Automatisierte Analyse von C#-Programmen für das Pull-Up-Field-Refactoring in MonoDevelop

Abschlussarbeit im Studiengang

Bachelor of Science Wirtschaftsinformatik

von

Kevin Stumpf

(Matrikelnummer: 7528604)

vorgelegt der
Fakultät für Mathematik und Informatik
der FernUniversität in Hagen

Betreuer:	Prof. Dr. Friedrich Steimann Lehrgebiet Programmiersysteme Fakultät für Mathematik und Informatik
Beginn der Arbeit:	15.06.2011
Abgabe der Arbeit:	15.09.2011

Ich erkläre hiermit, die folgende Bachelorarbeit selbständig verfasst zu haben. Andere als die angegebenen Quellen und Hilfsmittel habe ich nicht benutzt. Wörtliche und sinngemäße Zitate sind kenntlich gemacht.

Köln, den 15.09.2011

Kevin Stumpf

Zusammenfassung

Refaktorisierungen sind Änderungen an Quellcode, die meist zum Ziel haben, den Code verständlicher und lesbarer zu gestalten. Die Semantik des Codes soll dabei unberührt bleiben. Werden Refaktorisierungen vom Entwickler von Hand durchgeführt, so besteht immer die Gefahr, dass unbemerkt Fehler gemacht werden und sich das Programm anschließend nicht mehr wie erwartet verhält. Refaktorisierungen, die automatisch vom Computer durchgeführt werden, können dieses Risiko minimieren. Mit Refacola existiert ein System, welches Constraint-basierte Refaktorisierungen automatisiert durchführen kann, während die syntaktische Korrektheit und semantische Unberührtheit des Codes sichergestellt wird. Im Rahmen dieser Arbeit wurde ein Plugin für die C#-Entwicklungsumgebung MonoDevelop entwickelt, welches zum Ziel hat, dem Refacola-System alle Informationen über den Quellcode bereitzustellen, die benötigt werden, um das „Pull-Up-Field"-Refactoring zu realisieren. Desweiteren ist das Plugin in der Lage, alle Änderungen, die Refacola zur Durchführung der Refaktorisierung vorsieht, in den Quellcode zurückzuschreiben. In der vorliegenden Arbeit wird zunächst näher auf Constraint-basierte Refaktorisierungen und Refacola eingegangen. Anschließend werden die Implementierung des Plugins sowie ausgewählte Probleme, die sich im Rahmen der Entwicklung ergeben haben, im Detail erläutert.

Inhaltsverzeichnis

1. Einleitung **1**

2. Theoretische Grundlagen **3**
2.1. Constraints . . . 3
2.2. Constraint-basierte Refaktorisierungen . . . 4
2.3. Refacola (in Anlehnung an [SKvP11]) . . . 5
2.4. Das „Pull-Up-Field“-Refactoring . . . 8

3. Schnittstellen von MonoDevelop **11**
3.1. ProjectDom . . . 11
3.2. NRefactoryOld . . . 14
3.3. NRefactory . . . 18
3.4. NRefactoryResolver . . . 19
3.5. FindMemberAstVisitor . . . 21

4. C#-Sprachdefinition **23**
4.1. Namespaces . . . 23
4.2. Structs, Enums und Delegates . . . 23
4.3. Propertys . . . 24
4.4. Zugriffsmodifizierer . . . 24
4.5. Initialisierer . . . 25
4.6. Zusammenfassung . . . 25

5. C#-Refacola-Plugin **27**
5.1. Schnittstelle zwischen dem Plugin und Refacola . . . 27
5.2. Architektur . . . 30
5.3. Komponente Contracts . . . 30
5.3.1. AST-Modell . . . 30
5.3.2. Refaktorisierungen . . . 37
5.3.3. Verschiedenes . . . 38
5.4. Komponente AST-Access . . . 38
5.4.1. Implementierung des AST-Modells . . . 39
5.4.2. RefactoringController . . . 48
5.5. Komponente Refacola-Interface . . . 53
5.5.1. Implementierung der Refacola-C#-Sprachdefinition . . . 54

5.5.2. Algorithmus zur Generierung der Faktenbasis 55
5.5.3. Export der Faktenbasis 66
5.5.4. Import der Change Sets 70
5.5.5. Abbildung der geforderten Modifikationen durch RefactoringInputs 73
5.6. Komponente MonoDevelop-Extension 75

6. Probleme während der Entwicklungsphase 77
6.1. Auflösen von ReturnTypes 77
6.2. ProjectDom-Database 79
6.3. FindMemberAstVisitor 79
6.4. Receiver im Fall von lokalen Variablen 80
6.5. Accessibility None für PropertyGetter und -Setter 81
6.6. Caching 82
6.6.1. NRefactoryResolverExt 82
6.6.2. ProjectDomCache 84
6.6.3. ProjectDomReferenceCache 85
6.6.4. Zusammenfassung 85
6.7. PropertyGetter und -Setter Referenzen 86
6.8. Generics 86
6.8.1. Mögliche Fälle 87
6.8.2. Abbildung der Generics in der Faktenbasis 88
6.8.3. Modifikation der Sprachdefinition zur Unterstützung von Generics 91

7. Ausblick und Fazit 95

A. C#-Sprachdefinition 97

B. Java-Sprachdefinition nach [SvP11, S. 18] 99

C. Sprachunabhängige Unterschiede zwischen den Refacola-Sprachdefinitionen 101
C.1. Typen 101
C.2. Typereference 101
C.3. This Referenzen 101
C.4. Methoden 101

D. AST-Modell Klassendiagramm 103

E. SolutionTraverser-Logik 107

F. ReferenceResolver-Logik 111

G. ReferenceReceiverResolver-Logik 113

H. Plugin Bedienungsanleitung **115**
H.1. Installation 115
H.2. Generierung einer Faktenbasis 116
H.3. Deinstallation 124

I. Inhalt der beigefügten CD **125**

Abbildungsverzeichnis

2.1. Typische Kinds einer Sprachdefinition 5
2.2. Refacola-Architektur, in Anlehnung an [VP11, S. 21] 7
2.3. Durchführung des „Pull-Up-Field“-Refactorings 9

3.1. Klassen der ProjectDom Architektur 13
3.2. NRefactoryOld-Parser Aufteilung 16
3.3. Abgebildete Baumstruktur des NRefactoryOld-Parsers 17

5.1. Schnittstelle zwischen Refacola und dem entwickelten Plugin 28
5.2. Architektur des entwickelten Plugins 31
5.3. Dem Plugin zu Grunde liegendes AST-Modell 32
5.4. Interaktion der AST-Access-Komponente mit MonoDevelop 40
5.5. Implementierung des AST-Baumes 41
5.6. Algorithmus zur Auflösung von Referenz Empfängern 43
5.7. Beispiele für die Schritte des Algorithmus aus Abb. 5.6 45
5.8. Zusammenspiel der AST-Access-Komponenten, um Refaktorisierungen durchzuführen 49
5.9. Implementierung der Member-Entität der Sprachdefinition 56
5.10. Zusammenspiel der Klassen zur Faktengenerierung 57
5.11. Analyse Schritte des SolutionTraversers 59
5.12. Export der Faktenbasis 67
5.13. Import der Change Sets 71

6.1. NRefactoryResolverExt-Implementierung 83
6.2. ProjectDomCache 84
6.3. Refacola-C#-Sprachdefinition Erweiterung 92

D.1. AST-Modell Teil 1 - Solution, Projects, Namespaces und Typen 104
D.2. AST-Modell Teil 2 - Members 105
D.3. AST-Modell Teil 3 - Referenzen und Receiver 106

H.1. Verzeichnisstruktur nach Installation des Plugins 115
H.2. Menü des Plugins nachdem die Solution geöffnet wurde 117
H.3. Menü des Plugins nach erfolgter Generierung der Faktenbasis 117
H.4. Export der Faktenbasis 118

H.5. Auswahl eines Change Sets 122
H.6. Ergebnis der Refaktorisierung 123
H.7. AST Visualisierung 123

Tabellenverzeichnis

3.1. Aufgerufene Methoden eines NRefactory AstVisitors bei der Analyse des Codes aus Listing 3.1 20

Listings

3.1. C#-Code zur Demonstration des NRefactoryOld-Parsers 14
3.2. Aufgerufene Methoden eines NRefactoryOld AstVisitors bei der Analyse des Codes aus Listing 3.1 18

5.1. Aufbau einer Faktenbasis 29
5.2. Aufbau eines Change Sets 29
5.3. Beispiele für die möglichen Receiver-Typen des AST-Modells 34
5.4. Beispiele für die möglichen Bezeichner-Typen des AST-Modells 36
5.5. C#-Code zur Demonstration des SolutionTraversers 59
5.6. Implementierung des InstanceMethod Kinds sowie des Binds Querys . . . 69
5.7. Demonstration des Fully Decorated Name 73

6.1. Verwendungsszenarien von Generics in C# 87
6.2. Sonderfall 1 beim Auflösen von Receivern 88
6.3. Sonderfall 2 beim Auflösen von Receivern 89
6.4. Sonderfall 3 beim Auflösen von Receivern 90
6.5. Sonderfall 4 beim Auflösen von Receivern 90

E.1. Pseudo-Code zur Demonstration der SolutionTraverser-Logik 107

F.1. Pseudo-Code zur Demonstration der ReferenceResolver-Logik 111

G.1. Pseudo-Code zur Demonstration der ReferenceReceiverResolver-Logik . . 113

H.1. Quellcode zur Demonstration des Plugins 116
H.2. Faktenbasis des Quellcodes aus Listing H.1 119
H.3. Change Set um das Feld zu verschieben 121

1. Einleitung

Agile Softwareentwicklung ist ein Versuch, starre Entwicklungsprozesse wie das Wasserfallmodell, durch flexiblere und leichtgewichtigere Vorgehensmodelle zu ersetzen. Das im Februar 2001 festgelegte „Agile Manifest“ hält als eine der Grundregeln der agilen Softwareentwicklung fest, dass die Bereitschaft, Änderungen an der Software durchzuführen, wichtiger ist, als einem festgelegten Plan zu folgen (vgl. [Bec01]).
Folgt man diesem Grundsatz, so werden zwangsläufig Änderungen am Code der Software notwendig. Diese Änderungen bergen jedoch ein hohes Fehlerpotenzial, wenn sie manuell vom Entwickler durchgeführt werden müssen. Aus diesem Grund bieten moderne Entwicklungsumgebungen die Möglichkeit, verschiedene Codeanpassungen automatisiert vorzunehmen. Diese automatisierten Änderungen werden als Refaktorisierungen bezeichnet und verfolgen das Ziel, die Semantik - also das Verhalten - von Software unberührt zu lassen, während sie den Aufbau des Quellcodes oder einzelne Codefragmente abändern.
Refaktorisierungen können scheinbar triviale Aufgaben sein, wie das Umbenennen von Typen, Methoden und Feldern. Es sind jedoch auch komplexere Veränderungen, wie das Verlagern eines Feldes von einer Subklasse in eine Superklasse, denkbar (das sog. „Pull-Up-Field“-Refactoring).
Leider kann auch unter Zuhilfenahme moderner Entwicklungsumgebungen nicht immer davon ausgegangen werden, dass die Refaktorisierung fehlerfrei durchgeführt wird. Fehlerfrei bedeutet hier nicht nur, dass der Code noch immer kompilierbar ist, sondern viel mehr, dass sich die Anwendung zur Laufzeit noch genauso verhält, wie vor der Modifikation. Aus diesem Grund hat der Lehrstuhl für Programmiersysteme der FernUniversität in Hagen ein System entwickelt, welches in der Lage ist, Refaktorisierungen unter Sicherstellung der Unberührtheit der Semantik durchzuführen. Kern des Systems ist die Abbildung der Refaktorisierungslogik in der eigens dafür entwickelten Sprache „Refacola“. Der Algorithmus stützt sich auf eine spezielle Repräsentation (die sog. „Faktenbasis“) des Abstract Syntax Tree (AST), die einem - für die Entwicklungssprache - festgelegten Modell folgt. Dieses Modell bildet alle für die Refaktorisierung relevanten Codeelemente ab und setzt diese zueinander in Beziehung. Anschließend wird auf Basis der so erzeugten AST-Abbildung eine Menge von Constraints generiert, die gemeinsam sicherstellen, dass der Code im Zuge der Codeänderung semantisch und syntaktisch korrekt bleibt. Abschnitt 2.3 geht genauer auf die Funktionsweise von Refacola ein.
Ziel dieser Arbeit ist es, Refacola für beliebige C#-Programme eine Faktenbasis zur Verfügung stellen zu können, die ausreicht, um das „Pull-Up-Field“-Refactoring fehlerfrei auf den AST anzuwenden. Außerdem soll die Möglichkeit bestehen, Modifikationen,

die Refacola zur Umsetzung der Refaktorisierung vorsieht, entgegenzunehmen und in den Code zurückzuschreiben.
Diese Aufgaben sollen dabei von einem Plugin für die freie Entwicklungsumgebung MonoDevelop übernommen werden. In den folgenden Abschnitten werden zunächst die Constraint-basierte Refaktorisierung sowie das Refaktorisierungssystem „Refacola“ erläutert. Anschließend werden die Schnittstellen MonoDevelops beschrieben, die für die Entwicklung des Plugins verwendet wurden, bevor im Anschluss daran die Architektur und Funktionsweise des Plugins skizziert wird[1]. In diesem Zusammenhang werden noch verschiedene Probleme, die sich im Rahmen der Entwicklung ergeben haben, aufgezeigt.

[1]Bei der Ausarbeitung dieser schriftlichen Arbeit wurde sehr großen Wert auf die Nachvollziehbarkeit der Entwicklung gelegt. Aus diesem Grund sind die Ausführungen zur Implementierung teilweise sehr detailliert.

2. Theoretische Grundlagen

2.1. Constraints

Ein Constraint stellt eine Bedingung dar, welche verschiedene Variablen zueinander in Beziehung setzt (vgl. [Bar05, S. 1]). Diese Bedingung definiert bestimmte Anforderungen an den Wert jeder Variable. Nur wenn diese Anforderungen eingehalten werden ist der Constraint erfüllt.
Constraints werden meistens nicht alleine betrachtet, sondern beschreiben gemeinsam mit anderen Constraints ein Constraint-logisches Problem. Jedes Constraint setzt dabei eine Teilmenge der insgesamt im Problem vorhandenen Variablen in Beziehung zueinander. Eine Lösung für das gesamte Problem ist gefunden, wenn jeder Variable ein Wert zugewiesen wurde und alle Constraints erfüllt sind. Dabei kann es für ein Constraint-logisches Problem durchaus mehrere Lösungen geben, wobei verschiedene Lösungen unterschiedlich bewertet werden können. Die Abbildung der Constraints erfolgt dabei in deklarativer Form, d.h. sie beschreiben die Einschränkungen, die für die annehmbaren Werte verschiedener Variablen gelten, ohne einen Algorithmus anzubieten, der das Einhalten dieser Einschränkung sicherstellt (vgl. [Kre11, S. 9]).
Constraint-logische Probleme bilden häufig Fragestellungen aus der realen Welt ab. Damit diese Probleme von Computern gelöst werden können, müssen die Fragestellungen in eine für den Computer verständliche Form gebracht werden. Dabei werden die verwendeten Begrifflichkeiten der Fragestellung durch Zahlen abgebildet (vgl. [Kre11, S. 10]).
Einfache Constraint-logische Probleme sind beispielsweise:

- „Der Geburtstag von Mona liegt zwischen dem Geburtstag von Svenja und Robin. Welche Tage kommen in Frage?“ Der Geburtstag dieser drei Personen müsste jeweils mittels einer Variable abgebildet werden. Der dazu passende Constraint könnte folgendermaßen lauten: $GeburtstagSvenja < GeburtstagMona < GeburtstagRobin$. Wenn für $GeburtstagSvenja$ und $GeburtstagRobin$ Werte festgelegt wurden, kann ein Computer dieses Problem bearbeiten.

- „Ordne drei Würfel so nebeneinander an, dass die Summe der Augenzahlen der ersten beiden Würfel gleich der Augenzahl des dritten Würfels ist“. Hierbei würde für jeden Würfel eine Variable definiert werden, die dessen Augenzahl abbildet. Anschließend könnte dieser Constraint in die für Computer verständliche Form

$X + Y = Z$ gebracht werden. X beschreibt die Augenzahl des ersten, Y die des zweiten und Z die des dritten Würfels. Desweiteren wäre es noch notwendig festzulegen, dass der Wertebereich jeder Variable den natürlichen Zahlen von 1 bis 6 entspricht.

Diese trivialen Probleme können auch ohne die Hilfe von Computern gelöst werden. Erst bei komplexeren Problemen wird der Vorteil deutlich, den Computer liefern können. So fragt das „N-Damen Problem“ beispielsweise, wie viele Möglichkeiten es gibt, N Damen auf einem $N * N$ Schachbrett zu positionieren, ohne dass eine Dame von einer anderen geschlagen werden kann. Damen können einander schlagen, wenn sie sich in der gleichen Zeile, Spalte oder Diagonale eines Schachbretts befinden. In [Bar05, S. 3] wird für dieses Problem ein Modell angegeben, welches jeder Dame eine eigene Spalte i zuordnet. Die Zeile, in der sich eine Dame befindet, wird dabei durch die Variable R_i abgebildet, wobei i für ihre Spalte steht und der Wertebereich von R_i alle vorhandenen Zeilen, und damit die natürlichen Zahlen zwischen 1 und N, abbildet. Der Constraint zur Lösung dieses Problems wird dann folgendermaßen definiert:

$$i \neq j \rightarrow (R_i \neq R_j \wedge |i - j| \neq |R_i - R_j|)$$

Die Bedingung $R_i \neq R_j$ gibt an, dass zwei Damen, die per Definition in verschiedenen Spalten sind, nicht in der gleichen Zeile sein dürfen. Durch die Bedingung $|i-j| \neq |R_i-R_j|$ wird festgelegt, dass sie auch nicht in der gleichen Diagonalen sein dürfen.

2.2. Constraint-basierte Refaktorisierungen

Constraint-basierte Refaktorisierungen stellen mit Hilfe von Constraints sicher, dass der Code nach Durchführung der Refaktorisierung syntaktisch korrekt sowie semantisch identisch geblieben ist. Dazu ist es vorab notwendig, die Constraints auf Basis des AST zu generieren. Variablen, auf die sich die Constraints beziehen, können dabei verschiedenste, für die Entwicklungssprache typische, Elemente beschreiben. So können zum Beispiel Einschränkungen auf den Zugriffsmodifizierer (engl. Access Modifier) einer bestimmten Methode getroffen werden. Die Menge der gültigen Modifizierer hängt dann u.a. ab von der Position der Referenzen auf diese Methode. Mittels unterschiedlicher Constraints kann so sichergestellt werden, dass die Methode für die verschiedenen Referenzen weiterhin sichtbar bleibt.
Bevor die Refaktorisierung durchgeführt wird, liegt mit dem Ausgangszustand bereits eine mögliche Lösung des Constraint-Systems vor. Das Ziel besteht nun nach erfolgter Anwendung der Refaktorisierungslogik darin, Werte für die Variablen des Constraint-Systems zu finden, damit das gesamte System wieder erfüllt ist. Das kann beispielsweise bereits durch die Änderung eines Zugriffsmodifizierers der Fall sein.

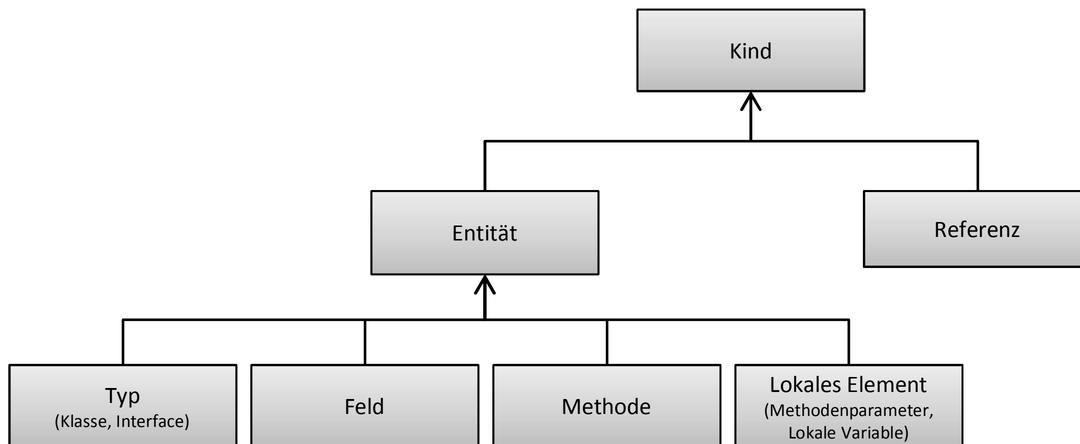


Abbildung 2.1.: Typische Kinds einer Sprachdefinition

Die Suche nach gültigen Werten für die Variablen ist Aufgabe des sog. „Constraint-Solvers“. Wenn keine Lösung gefunden werden kann, ist die Refaktorisierung nicht möglich.

2.3. Refacola (in Anlehnung an [SKvP11])

Mit Refacola wurde eine Sprache entwickelt, um den Algorithmus von Refaktorisierungen abbilden zu können. Die entwickelten Algorithmen sind dabei unabhängig von der Zielsprache, in welcher der zu refaktorisierende Code entwickelt wurde. Damit dies möglich ist, stützt sich die Refaktorisierung auf drei Säulen:

- Die Sprachdefinition der Zielsprache
- Die Constraint-Regeln
- Die Definition der Refaktorisierung

Die Sprachdefinition definiert, durch welche Element-Typen (sog. „Kinds“) eine Sprache abgebildet werden kann. Dabei werden diesen Element-Typen verschiedene näher charakterisierende Eigenschaften (engl. Propertys), gemeinsam mit einem Wertebereich (engl. Domain), zugeordnet. Zusätzlich beschreibt die Sprachdefinition die sog. „Querys“, die in ihrer abstrakten Definition verschiedene Kinds zueinander in Beziehung setzen. Kinds können bei objekt-orientierten Programmiersprachen gemäß Abb. 2.1 klassifiziert werden. Sie werden folglich in Entitäten und Referenzen auf Entitäten unterteilt. Entitäten repräsentieren beispielsweise Typen, Felder, Methoden oder lokale Elemente. Beispiele für

Eigenschaften dieser Elemente sind der Bezeichner (engl. Identifier) und der Zugriffsmodifizierer. Der Wertebereich von Zugriffsmodifizierern umfasst alle für die zugrundeliegende Sprache gültigen Werte. Im Falle von C# sind dies `public`, `internal`, `protected`, `protected internal` und `private`.
Sobald Kinds verwendet werden, um konkrete Elemente eines AST abzubilden, spricht man von Programmelementen. Analog dazu spricht man von Fakten, wenn auf Grundlage der definierten Querys verschiedene Programmelemente zueinander in Beziehung gesetzt werden. Somit kann man sich Programmelemente als Instanzen von Kinds und Fakten als Instanzen von Querys vorstellen. Programmelemente beschreiben damit beispielsweise einzelne Klassen des AST, während Fakten beschreiben können, dass ein Feld zu einem bestimmten deklarierten Typ gehört oder eine identifizierte Referenz sich auf eine Entität bezieht. Programmelemente und Fakten, die gemeinsam einen AST abbilden, ergeben zusammengefasst eine Faktenbasis.
Neben der Sprachdefinition werden die Constraint-Regeln definiert, welche sich auf die Kinds und Querys stützen. Wurde für ein Projekt eine vollständige Faktenbasis erzeugt, so lässt sich mittels dieser Constraint-Regeln ein darauf basierendes Constraint-System erstellen, welches den gesamten AST repräsentiert. Sind die Constraint-Regeln korrekt und vollständig definiert sowie das daraus erzeugte Constraint-System nach erfolgter Refaktorisierung erfüllt, so kann davon ausgegangen werden, dass der Code weiterhin syntaktisch und semantisch korrekt ist.
Eine Constraint-Regel könnte zum Beispiel definieren, dass für jede Referenz auf eine Entität der Bezeichner der Entität mit dem Bezeichner der Referenz übereinstimmen muss. Wird diese Regel auf einen vorhandenen Quellcode angewandt, um ein Constraint-System zu erzeugen, so wird für jedes Paar aus Entität und sich darauf beziehender Referenz ein Constraint erzeugt, der die Gleichheit der beiden Bezeichner verlangt.
Darüber hinaus benötigt Refacola für die Refaktorisierung noch einen Algorithmus der die gewünschten Änderungen am AST durchführt. Der Algorithmus wird dabei ebenfalls in der Refacola-Sprache verfasst.
Die Refacola-Infrastruktur besteht jedoch nicht nur aus einer Sprache, mit der die Refaktorisierungslogik, Constraint-Regeln und Sprachdefinition der Zielsprache abgebildet werden können. Sie ist ebenfalls in der Lage, für einen beliebigen Programmcode die Programmelemente und Fakten auf Basis der Sprachdefinition zu erzeugen. Dazu wendet sie sich an ein Plugin innerhalb der Integrated Development Environment (IDE), welches ein festgelegtes Interface anbietet und darüber Zugriff auf den AST gewährt. Desweiteren enthält die Infrastruktur einen Constraint-Generator, welcher basierend auf den erzeugten Programmelementen und Fakten sowie den sprachspezifischen Constraint-Regeln das Constraint-System erzeugt. Mit einem Constraint-Solver ist Refacola schließlich in der Lage, das erzeugte Constraint-System zu lösen. Sollte eine Lösung gefunden worden sein, so werden die notwendigen Änderungen an das Plugin gereicht, um die Änderungen am AST durchzuführen. Abb. 2.2 zeigt die Architektur der gesamten Refacola-Infrastruktur.

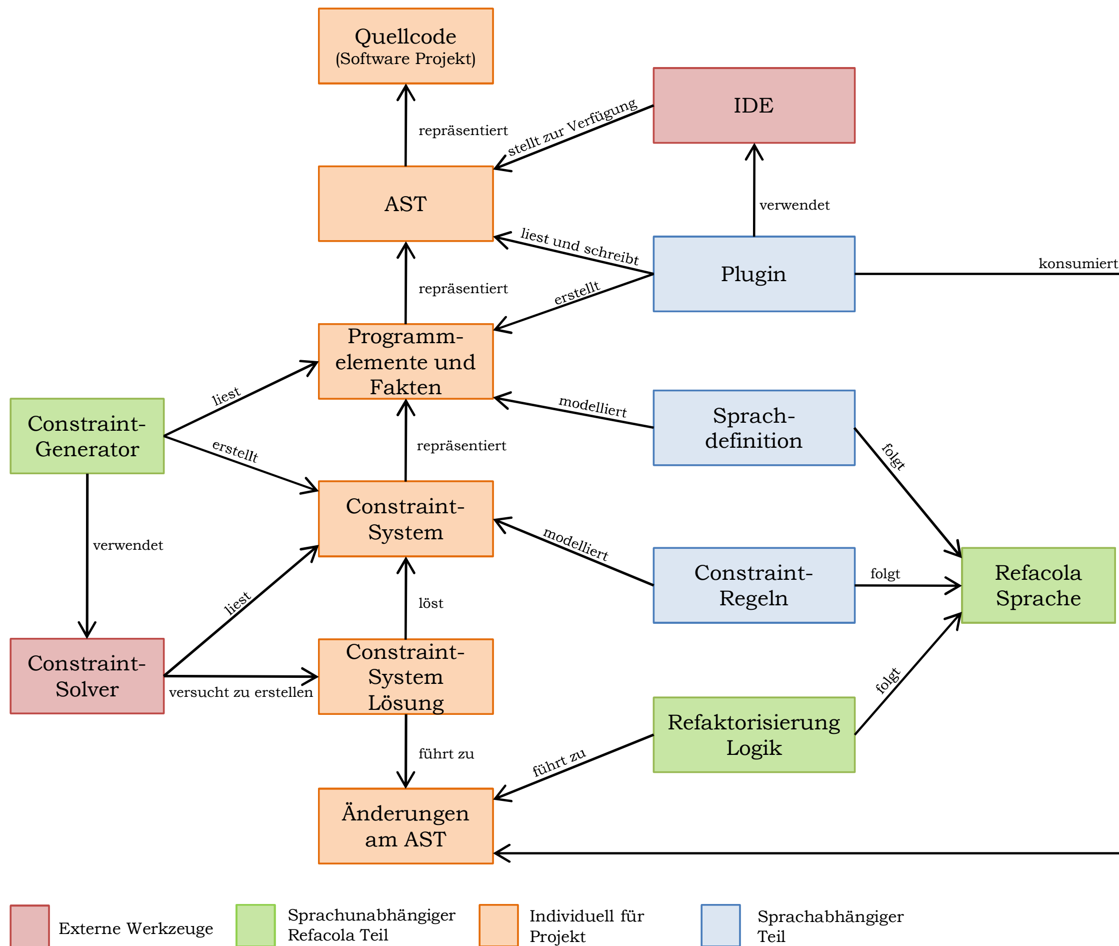


Abbildung 2.2.: Refacola-Architektur, in Anlehnung an [VP11, S. 21]

2.4. Das „Pull-Up-Field“-Refactoring

Diese Arbeit befasst sich mit dem automatisierten Erstellen einer Faktenbasis, die ausreicht, um ein Constraint-System zu erzeugen, welches die syntaktische und semantische Korrektheit nach erfolgtem „Pull-Up-Field“-Refactoring sicherstellt. Diese Refaktorisierung hat als Ausgangspunkt eine Klasse, die von einer Basisklasse ableitet und ein bestimmtes Feld deklariert. Das Ziel besteht darin, die Deklaration des Feldes in die Basisklasse zu verschieben. Abb. 2.3 zeigt diese Operation anhand eines Beispiels.
Bereits bei diesem trivialen Code wird deutlich, dass es meist nicht ausreicht, die Felddeklaration ohne weitergehende Modifikationen in die Basisklasse zu verschieben. So kann es beispielsweise notwendig sein, die Sichtbarkeit des Feldes zu verändern, damit die Superklasse weiterhin Zugriff darauf hat. Um die Refaktorisierung unter Gewährleistung der semantischen Unberührtheit und syntaktischen Korrektheit durchführen zu können, muss der Algorithmus auf folgende Informationen Zugriff haben:

- In welcher Klasse ist das Feld deklariert und welche Sichtbarkeit besitzt die Klasse?
- Wovon leitet die Klasse, die das Feld beinhaltet, ab? In welchem Namespace mit welcher Sichtbarkeit befindet sich diese Superklasse?
- Welche Referenzen verweisen momentan auf das Feld?
- Innerhalb welcher Entität sind die Referenzen angesiedelt, d.h. in welcher Methode, Property oder Feldinitialisierung wird auf dieses Feld verwiesen? Was ist die Sichtbarkeit dieser Entität und zu welchem Typ gehört sie?
- Wird auf das Feld direkt zugegriffen oder führt der Zugriff über Empfänger-Objekte?
- Wird das Feld bei der Deklaration direkt initialisiert? Auf welche anderen Referenzen wird bei dieser Initialisierung zugegriffen?

Basierend auf diesen Informationen ist es über die Constraint-Regeln möglich, für die Refaktorisierung Constraints zu generieren, die das zu refaktorisierende Programm repräsentieren. Mittels dieser Constraints wird nach erfolgtem Verschieben des Feldes versucht, über den Constraint-Solver eine Lösung für das gesamte System zu finden. Wurde eine Lösung gefunden, so kann das Verschieben samt allen begleitenden Codemodifikationen durchgeführt werden. Andernfalls muss die Refaktorisierung abgelehnt werden.

Damit sind die theoretischen Grundlagen, auf die sich das Plugin stützt, erläutert. Im folgenden Kapitel werden die Schnittstellen von MonoDevelop erklärt, die das Plugin verwendet, um die Faktenbasis zu generieren sowie Codemodifikationen durchzuführen.

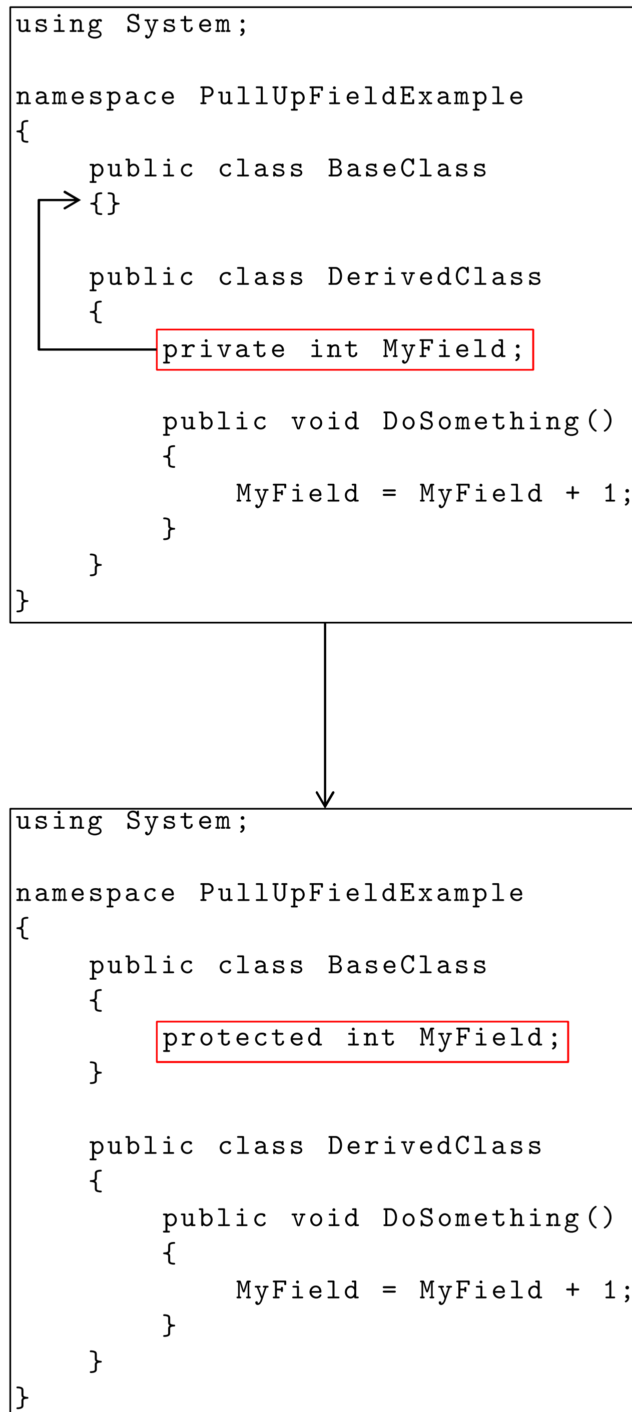


Abbildung 2.3.: Durchführung des „Pull-Up-Field“-Refactorings

3. Schnittstellen von MonoDevelop

MonoDevelop bietet geladenen Plugins verschiedene Schnittstellen, um den Aufbau des AST in unterschiedlicher Granularität und Informationsdichte zu analysieren.
In den folgenden Abschnitten werden folgende Schnittstellen näher beleuchtet, auf die sich das entwickelte Plugin zum Extrahieren der Fakten stützt:

- ProjectDom: Dieses Document Object Model (DOM) kann für eine oberflächliche Analyse von geladenen Projekten und .NET-Assemblys verwendet werden.

- NRefactoryOld: Dieses Modul gewährleistet einen detaillierten Überblick über den Aufbau einzelner C#-Dateien.

- NRefactory: Diese Klassensammlung ist ein Nachfolger des NRefactoryOld-Moduls und kann verwendet werden, um den gesamten Aufbau von C#-Dateien noch detaillierter zu analysieren.

- NRefactoryResolver: Mittels dieser Komponente können einzelne Codefragmente ihren entsprechenden Elementen aus dem ProjectDom zugeordnet werden.

- FindMemberAstVisitor: Diese Klasse ist in der Lage, im AST nach Referenzen auf Elemente zu suchen.

3.1. ProjectDom

Im Namespace `MonoDevelop.Projects.Dom` der Assembly `MonoDevelop.Core` befinden sich die Klassen der ProjectDom-Komponente. Dom steht hierbei für Document Object Model. Der Name ist allerdings irreführend, da sich dieses Model durchaus über mehrere Dokumente erstrecken kann.
Mithilfe dieser Klassen kann ein zusammenhängender AST - hinweg über die Grenzen von einzelnen Quellcode-Dateien und sogar Projekten - ausgelesen werden. Es ist dabei unerheblich, ob der Code ausschließlich in Form bereits kompilierter Dateien (die sog. „Assemblys") oder aber nur in Textform vorliegt. Da Assemblys immer als kompilierte

Intermediate Language (IL) vorliegen, ist es für das ProjectDom unwichtig, in welcher Sprache die referenzierten Assemblys ursprünglich entwickelt wurden.
Die Granularität der Informationen, die über diese Schnittstelle gewonnen werden können, ist dabei sehr beschränkt. Abb. 3.1 zeigt die für das Plugin relevanten Bestandteile des AST, die mit Hilfe dieser Komponente ausgelesen werden können. Dabei wird deutlich, dass eine Solution in MonoDevelop über die Klasse `Solution` repräsentiert wird. Darüber erhält man Zugriff auf alle darin enthaltenen Projekte. Über die `Project`-Instanzen kann schließlich die dazugehörende `ProjectDom`-Instanz angefordert werden, welche das ProjectDom des Projekts repräsentiert[1]. Über die `ProjectDom`-Instanz können die in der Abbildung dargestellten Details des AST ausgelesen werden:

- Es lässt sich auf die `ProjectDom`-Instanz aller referenzierten Assemblys zugreifen.
- Alle innerhalb des Projekts deklarierten Typen werden veröffentlicht. Über deren Property `ClassType` lässt sich die Art von Typ bestimmen. Dabei werden Klassen, Interfaces, Structs, Enums und Delegates unterschieden. Neben der Basisklasse kann außerdem auf die implementierten Interfaces zugegriffen werden. Auch der Zugriffsmodifizierer ist einsehbar. Desweiteren erhält man über `Type`-Instanzen Zugriff auf eine `CompilationUnit`-Instanz, welche die Datei, in der der Typ deklariert wurde, beschreibt. Diese Beschreibung enthält neben dem Pfad alle `using`-Instruktionen sowie die weiteren Typdeklaration innerhalb dieser Datei. Sollte der Typ partiell - und damit in verschiedenen Dateien - definiert sein, kann auf die anderen Teile mittels des `Parts`-Propertys zugegriffen werden.
- Felder werden durch Instanzen vom Typ `Field` repräsentiert. Diese geben Auskunft über den Namen, die Sichtbarkeit sowie deren Rückgabetyp.
- Methoden, die durch `Method`-Instanzen abgebildet werden, veröffentlichen die gleichen - für das Plugin relevanten - Informationen wie Felder. Weitergehende Informationen, wie die Deklaration einer Methode als `abstract` oder `virtual`, können über das Property `Modifiers` eingesehen werden.
- Der Zugriff auf Propertys erfolgt über Instanzen vom Typ `Property`. Neben den bereits erläuterten Details kann der Zugriffsmodifizierer des PropertyGetters und -Setters ausgelesen werden. Das ist besonders für die Generierung der Fakten wichtig, da es die C#-Spezifikation laut [ECM06, S. 300-305] erlaubt, den Zugriffsmodifizierer von Getter oder Setter weiter einzuschränken, als den des Propertys selbst.

[1] Um Verwirrungen zu vermeiden muss hierbei beachtet werden, dass mit ProjectDom zum einen das hier beschriebene AST-Modell bezeichnet wird. Zum anderen existiert noch die gleich benannte Klasse `ProjectDom`, die jedoch nur ein Teil des Modells ist. Die unterschiedliche Schreibweise soll im Rahmen der Arbeit anzeigen, wovon genau gesprochen wird.

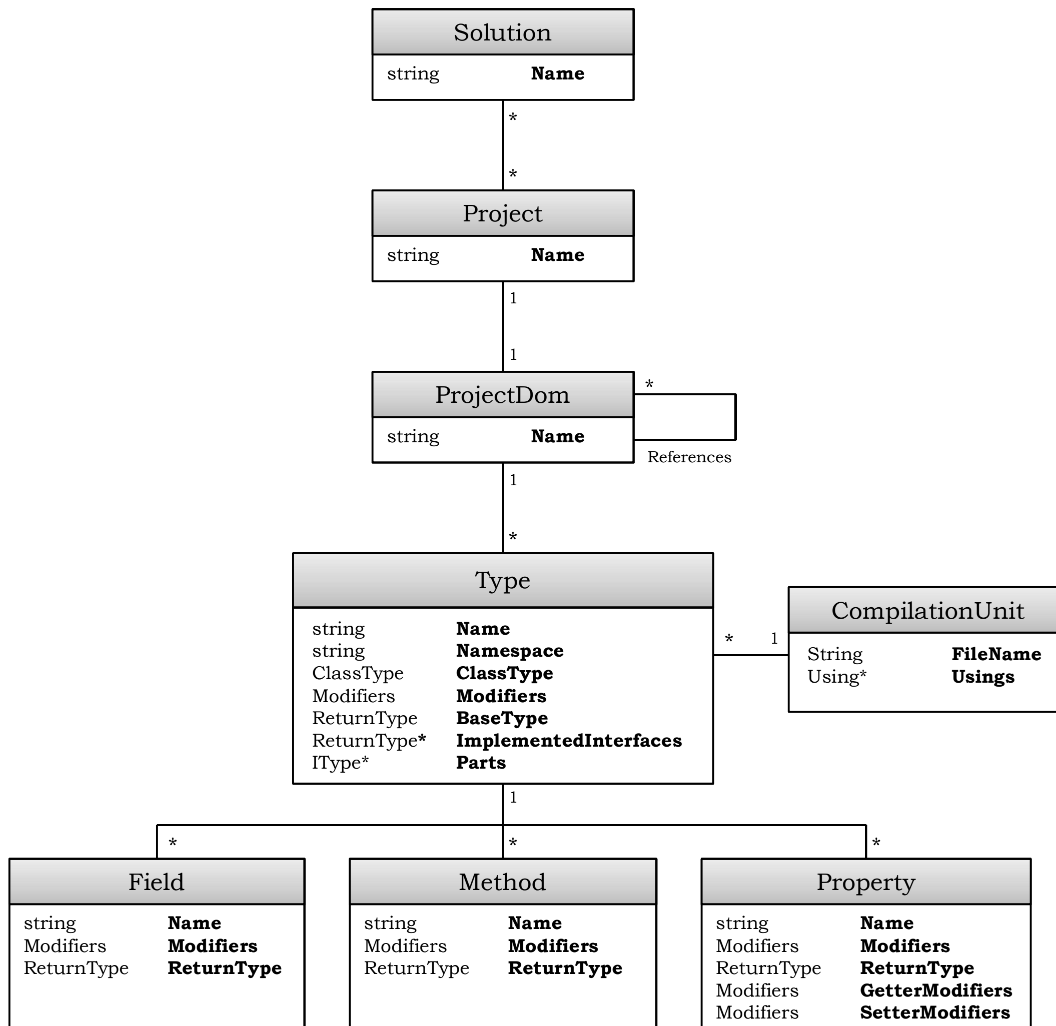


Abbildung 3.1.: Klassen der ProjectDom Architektur

Neben diesen beschriebenen AST-Knoten, die konkrete Elemente des Codes widerspiegeln, werden im ProjectDom alle Referenzen auf Typen durch `ReturnType`-Instanzen dargestellt, welche lediglich den Namen des referenzierten Typs beschreiben. Das betrifft zum Beispiel die Typen von Methodenargumenten oder auch die Rückgabetypen von Feldern, Propertys und Methoden. `ReturnType`-Instanzen können mit Hilfe der `CompilationUnit`-Klasse zu vollständigen `Type`-Instanzen aufgelöst werden. Hierbei wird versucht über die `using`-Instruktionen in der Datei, in welcher der Typ referenziert wird, sowie über die referenzierten Assemblys, den Typ zu finden.

Damit sind die Grenzen des ProjectDoms erreicht. Weitergehende Details, wie beispielsweise die Referenzen auf AST-Elemente, erhält man mit diesem Mittel alleine nicht. Um eine vollständige Faktenbasis generieren zu können, müssen die in den nächsten Abschnitten erläuterten Komponenten zur Hilfe genommen werden.

3.2. NRefactoryOld

Die im Namespace `ICSharpCode.OldNRefactory` der Assembly `NRefactory` befindliche Klassensammlung lässt sich verwenden, um einzelne C#-Dateien im Detail zu analysieren. Hierbei kann ein Parser verwendet werden, um aus Quellcode, der in Textform vorliegt, eine Baumstruktur zu generieren, welche die Textelemente in logischen Einheiten abbildet; diese Einheiten stehen in einer - für Baumstrukturen charakteristischen - Eltern-Kind-Beziehung zueinander, was bedeutet, dass alle Baumknoten - mit Ausnahme der Wurzel und der Blätter - immer einen übergeordneten Elternknoten und untergeordnete Kindknoten referenzieren.
Zur Erläuterung diesen Vorgehens soll der C#-Code aus Listing 3.1 helfen, welcher aus einer Klasse `MyClass` im Namespace `AstVisitorExample` besteht, die wiederum eine öffentliche Methode `SomeMethod` sowie eine private Methode `SomeOtherMethod` enthält. Innerhalb von `SomeMethod` wird die Variable `i` vom Typ `int` deklariert sowie die Methode `SomeOtherMethod` aufgerufen. Die gesamte Deklaration befindet sich in einer Datei mit einem einleitendem `using` Statement.

Listing 3.1: C#-Code zur Demonstration des NRefactoryOld-Parsers

```
1 using System;
2
3
4 namespace AstVisitorExample
5 {
6     public class MyClass
7     {
8         public void SomeMethod()
9         {
```

```
10              int i;
11
12              SomeOtherMethod();
13
14          }
15
16
17          private void SomeOtherMethod()
18          {
19
20          }
21      }
22 }
```

Abb. 3.2 zeigt, welche Codefragmente mit Hilfe logischer Einheiten vom NRefactoryOld-Parser abgebildet werden. Die eingezeichneten Klammern weisen daraufhin, welche Zeilen- und Spaltenangaben aus diesen Einheiten ausgelesen werden können.
Die gesamte Datei wird durch eine Instanz der Klasse `CompilationUnit`[2] abgebildet, welche das Ergebnis der Quellcode-Analyse durch den Parser ist.
Das `using`-Statement in Zeile 1 wird durch eine Instanz vom Typ `UsingDeclaration` abgebildet und beinhaltet eine `Using`-Instanz, die „System“ abbildet. Die gesamte Deklaration des Namespaces, die sich über die Zeilen 7 bis 42 erstreckt, wird von einer Instanz des Typs `NamespaceDeclaration` beschrieben. Die Deklaration der Klasse `MyClass` von Zeile 12 bis 40 verbirgt sich hinter einer `TypeDeclaration`-Instanz. Funktionsköpfe werden durch `MethodDeclaration`-Instanzen repräsentiert, wobei deren Rückgabetyp von einer `TypeReference` beschrieben wird. Die eingezeichnete Klammerung soll darauf hinweisen, dass die `MethodDeclaration` tatsächlich lediglich den Funktionskopf ohne dazugehörigen Rumpf beschreibt. Die gesamte Deklaration der Variable `i` in Zeile 21 ist in einer `LocalVariableDeclaration`-Instanz abgebildet, welche den Typ `int` als `TypeReference` und den Variablennamen als `VariableDeclaration` abbildet.
Der Funktionsaufruf `SomeOtherMethod()` in Zeile 27 wird durch eine Instanz des Typs `InvocationExpression` abgebildet. Diese enthält eine `InvocationExpression`, welche wiederum eine `IdentifierExpression` beinhaltet.
Damit wird insgesamt die in Abb. 3.3 dargestellte Baumstruktur im Speicher abgebildet, wobei die Knoten über die Indexnummer die in Abb. 3.2 dargestellten Blöcke referenzieren.

[2]Die hier verwendete `CompilationUnit`-Klasse ist dabei nicht mit der `CompilationUnit`-Implementierung aus dem ProjectDom zu verwechseln. Beide beziehen sich zwar auf Quellcode-Dateien, allerdings sind die enthaltenen Informationen unterschiedlich und die Implementierungen strikt voneinander getrennt.

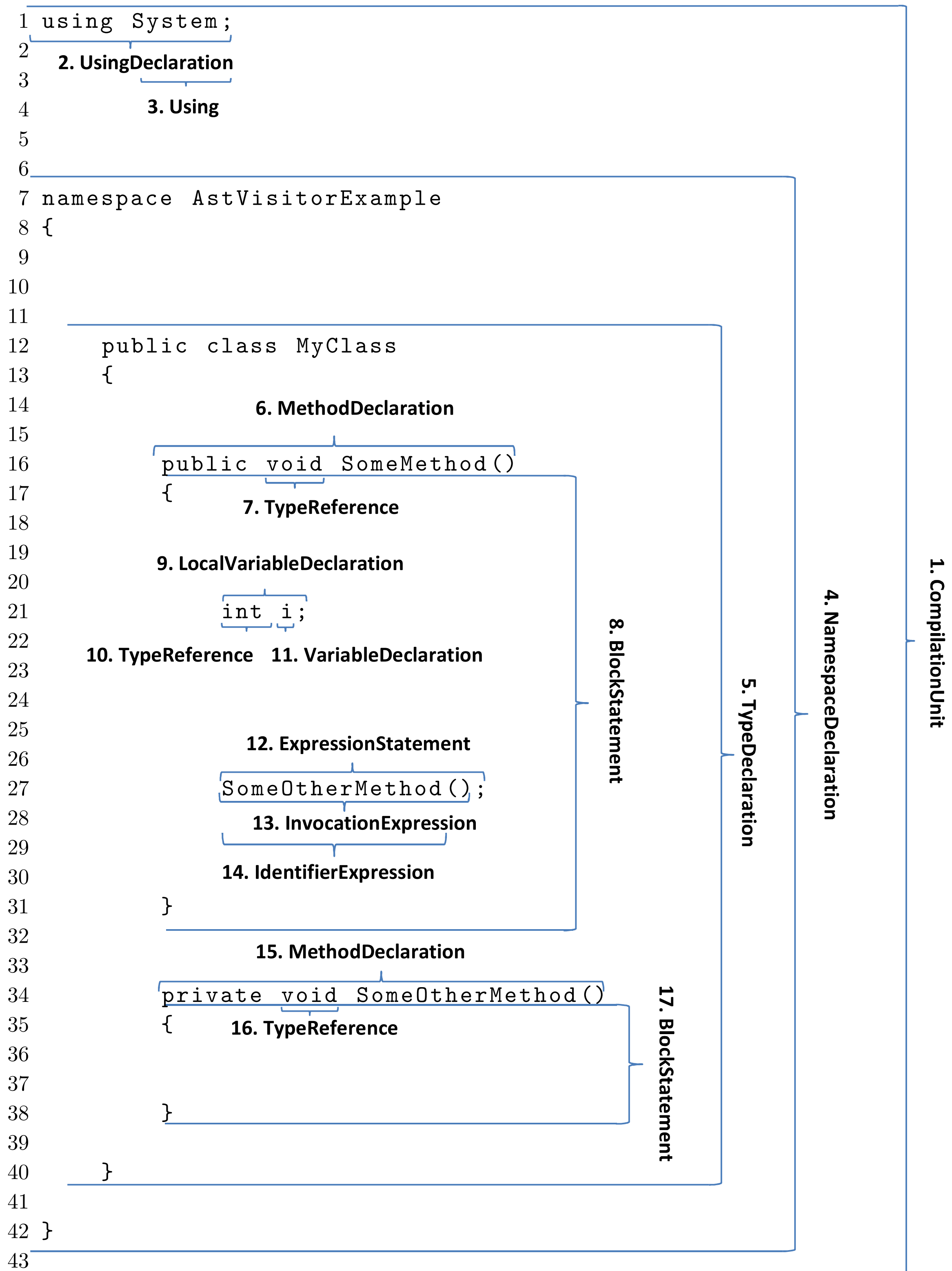


Abbildung 3.2.: NRefactoryOld-Parser Aufteilung

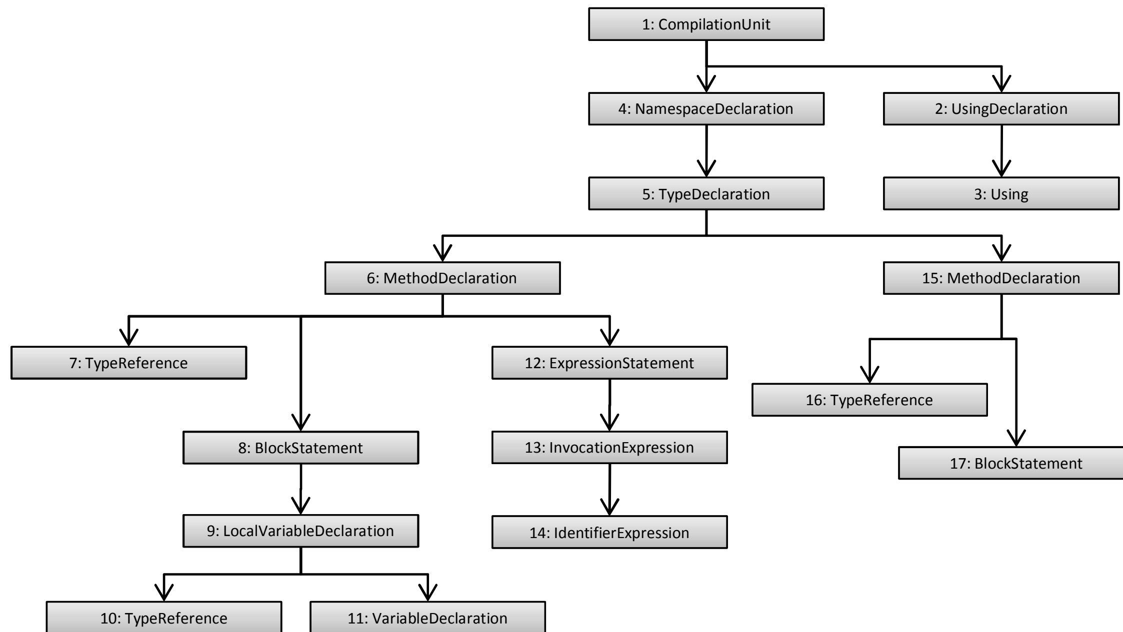


Abbildung 3.3.: Abgebildete Baumstruktur des NRefactoryOld-Parsers

Möchte man eine geparste Datei analysieren, so sieht NRefactoryOld einen Ansatz gemäß dem Visitor-Pattern (vgl. [GHJV94, S. 331-344]) vor. Hierbei soll eine Visitor-Klasse von der abstrakten Klasse `AbstractAstVisitor` ableiten. Diese ist in der Lage, alle Knoten einer `CompilationUnit`-Instanz zu durchlaufen. Wenn bestimmte Knoten von Interesse sind, muss lediglich die dafür zuständige Funktion überschrieben werden. Soll die Klasse zum Beispiel mit Typdeklarationen arbeiten, so sollte sie die Methode `VisitTypeDeclaration` überschreiben; Methodendeklarationen lassen sich in der Methode `VisitMethodDeclaration` analysieren.

Listing 3.2 zeigt alle Funktionsaufrufe, die abgefangen werden können, wenn das Code-Beispiel aus Listing 3.1 analysiert wird.

Listing 3.2: Aufgerufene Methoden eines NRefactoryOld AstVisitors bei der Analyse des Codes aus Listing 3.1

```
1 VisitCompilationUnit
2 VisitUsingDeclaration
3 VisitUsing
4 VisitNamespaceDeclaration
5 VisitTypeDeclaration
6 VisitMethodDeclaration
7 VisitTypeReference
8 VisitBlockStatement
9 VisitLocalVariableDeclaration
10 VisitTypeReference
11 VisitVariableDeclaration
12 VisitExpressionStatement
13 VisitInvocationExpression
14 VisitIdentifierExpression
15 VisitMethodDeclaration
16 VisitTypeReference
17 VisitBlockStatement
```

Die Zeilennummer referenziert auch hierbei die bekannten Indices aus Abb. 3.2 und 3.3. NRefactoryOld bildet zwar bereits deutlich mehr Details auf Dateiebene ab als das vorgestellte ProjectDom, allerdings werden auch bei diesem Ansatz nicht alle möglichen Sprachelemente durch eigene Klassen abgebildet und damit mit Zeilen- und Spalteninformationen versehen. So wird zum Beispiel der Zugriffsmodifizierer der Klasse `MyClass` sowie der Methoden `SomeMethod` und `SomeOtherMethod` nicht durch eigenständige Instanzen beschrieben. Benötigt man die Positionsangaben vorhandener Zugriffsmodifizierer, so ist es lediglich über eine Textsuche innerhalb des von `TypeDeclaration`- bzw. `MethodDeclaration`-Instanzen beschriebenen Abschnitts möglich, nach dafür gültigen Werten zu suchen. Desweiteren fehlen für `TypeReference`-Instanzen sämtliche Positionsangaben, die den Rückgabewert von Methoden beschreiben.
Diese Probleme werden durch den NRefactoryOld Nachfolger gelöst, der im nachfolgenden Abschnitts beleuchtet wird.

3.3. NRefactory

Der angesprochene Nachfolger befindet sich in der Assembly `ICSharpCode.NRefactory` im gleichnamigen Namespace und wird seit MonoDevelop 2.6 entwickelt. Die Möglichkeiten der Code-Analyse ähneln sehr stark NRefactoryOld. Auch hier wird, ausgehend von einer

C#-Quellcode-Datei, durch einen Parser eine `CompilationUnit`-Instanz[3] erzeugt, welche alle Codefragmente in logischen Einheiten abbildet, die wiederum in einer Eltern-Kind-Beziehung zueinander stehen. Der große Unterschied zu NRefactoryOld besteht in der Granularität der Informationen, die sich aus in Textform vorliegendem Quellcode gewinnen lassen. Es ist nicht mehr notwendig Volltextsuchen durchzuführen, um an gewünschte Daten zu kommen, da jedes für sich stehende Fragment durch ein Objekt abgebildet wird.
Die Analyse von `CompilationUnit`-Instanzen gestaltet sich wieder nach dem Visitor-Pattern. Dabei muss eine Klasse implementiert werden, die ein festgelegtes Visitor-Interface implementiert. Für jedes analysierte Element definiert das Interface eine eigene Methode, die aufgerufen wird, wenn das entsprechende Element besucht wird.
Tabelle 3.1 gibt alle Methoden wieder, die bei der Analyse des in Listing 3.1 angegebenen Beispiels aufgerufen werden. Dabei wird über die Spalten „Position“ und „Inhalt“ angegeben, auf welche Codestelle sich das Fragment bezieht.
Vergleicht man diese Tabelle mit den Möglichkeiten des NRefactoryOld Visitors aus Listing 3.2, so wird deutlich, dass die Analyse ausführlicher geworden ist. Auffällig ist vor allem die große Anzahl der besuchten `CSharpTokenNodes`, die alle C#-Schlüsselwörter, -Literale, -Operatoren und -Zeichensetzer (vgl. [ECM06, S. 69-77]) repräsentieren. Mit diesem Hilfsmittel hat man die Möglichkeit, die genauen Positionsangaben zu jedem einzelnen Codefragment auszulesen. Das ist insbesondere dann nützlich, wenn Code zuverlässig geändert werden muss.
Die bisher besprochenen MonoDevelop-Schnittstellen reichen aus, um Projekte oder einzelne Dateien als Bäume zu interpretieren. Möchte man jedoch einen Schritt weitergehen und Projekte als Graphen, d.h. als Bäume mit Zyklen, interpretieren, so benötigt man einen Mechanismus, um Referenzen auf Knoten des AST aufzulösen. Die folgenden Kapitel stellen Komponenten von MonoDevelop vor, die genau diesen Zweck erfüllen.

3.4. NRefactoryResolver

Wie gezeigt wurde, bietet das ProjectDom eine oberflächliche und dateiübergreifende Perspektive auf den AST, während es von den Details des Quellcodes losgelöst ist. Der `NRefactoryResolver` aus dem Namespace `MonoDevelop.CSharp.Resolver` der Assembly `CSharpBinding` hilft diese Lücke zu schließen, indem er einzelnen Codefragmenten ihre entsprechenden Objekte aus dem ProjectDom zuordnet. Dazu bedient er sich einer weiteren Klasse, dem `ResolveVisitor`. Dieser ist ein NRefactoryOld Visitor und ordnet zuerst dem übergebenen Codefragment den umschließenden Typ sowie - falls vorhanden - den umschließenden Member[4] aus dem ProjectDom zu. Anschließend überprüft er alle Typen sowie deren Member, die im ProjectDom selbst oder einem referenzierten Project-

[3] Diese Klasse ist wieder nicht mit den anderen bereits eingeführten CompilationUnit-Klassen zu verwechseln.
[4] Zum Beispiel die Methode, in der sich das Fragment befindet.

Index	Methode	Position	Beschriebener Inhalt
1:	VisitCompilationUnit	L01-L22	Gesamte Datei
2:	VisitUsingDeclaration	L01	'using System;'
3:	VisitCSharpTokenNode	L01	'using'
4:	VisitSimpleType	L01	'System'
5:	VisitIdentifier	L01	'System'
6:	VisitCSharpTokenNode	L01	';'
7:	VisitNamespaceDeclaration	L04-L22	Namespace 'AstVisitorExample'
8:	VisitCSharpTokenNode	L04	'namespace'
9:	VisitIdentifier	L04	'AstVisitorExample'
10:	VisitCSharpTokenNode	L05	'{'
11:	VisitTypeDeclaration	L06-L21	Klasse 'MyClass'
12:	VisitCSharpTokenNode	L06	'public'
13:	VisitCSharpTokenNode	L06	'class'
14:	VisitIdentifier	L06	'MyClass'
15:	VisitCSharpTokenNode	L07	'{'
16:	VisitMethodDeclaration	L08-L14	Methode 'SomeMethod'
17:	VisitCSharpTokenNode	L08	'public'
18:	VisitPrimitiveType	L08	'void'
19:	VisitIdentifier	L08	'SomeMethod'
20:	VisitCSharpTokenNode	L08	'('
21:	VisitCSharpTokenNode	L08	')'
22:	VisitBlockStatement	L09-L14	Methodenrumpf 'SomeMethod'
23:	VisitCSharpTokenNode	L09	'{'
24:	VisitVariableDeclarationStatement	L10	'int i;'
25:	VisitPrimitiveType	L10	'int'
26:	VisitVariableInitializer	L10	'i'
27:	VisitIdentifier	L10	'i'
28:	VisitCSharpTokenNode	L10	';'
29:	VisitExpressionStatement	L12	'SomeOtherMethod();'
30:	VisitInvocationExpression	L12	'SomeOtherMethod()'
31:	VisitIdentifierExpression	L12	'SomeOtherMethod'
32:	VisitIdentifier	L12	'SomeOtherMethod'
33:	VisitCSharpTokenNode	L12	'('
34:	VisitCSharpTokenNode	L12	')'
35:	VisitCSharpTokenNode	L12	';'
36:	VisitCSharpTokenNode	L14	'}'
37:	VisitMethodDeclaration	L17-L19	Methode 'SomeOtherMethod'
38:	VisitCSharpTokenNode	L17	'private'
39:	VisitPrimitiveType	L17	'void'
40:	VisitIdentifier	L17	'SomeOtherMethod'
41:	VisitCSharpTokenNode	L17	'('
42:	VisitCSharpTokenNode	L17	')'
43:	VisitBlockStatement	L18-L20	Methodenrumpf 'SomeMethod'
44:	VisitCSharpTokenNode	L18	'{'
45:	VisitCSharpTokenNode	L20	'}'
46:	VisitCSharpTokenNode	L21	'}'
47:	VisitCSharpTokenNode	L22	'}'

Tabelle 3.1.: Aufgerufene Methoden eines NRefactory AstVisitors bei der Analyse des Codes aus Listing 3.1

Dom enthalten sind, dahingehend, ob deren Name mit dem analysierten Codefragment übereinstimmt und ob deren Namespace im untersuchten CompilationUnit verwendet wird. Ist das der Fall, wird weiter überprüft, ob dieser Typ bzw. Member ausgehend vom umschließenden Typ und Member des Codefragments sichtbar ist. Wenn auch das zutrifft, wurde das referenzierte Objekt im ProjectDom gefunden.

3.5. FindMemberAstVisitor

Neben dem `NRefactoryResolver` befindet sich in der Assembly `CSharpBinding` im Namespace `MonoDevelop.CSharp.Refactoring` der `FindMemberAstVisitor`. Dieser Visitor ist in der Lage, `CompilationUnit`-Instanzen, die vom NRefactoryOld-Parser erzeugt wurden, nach Referenzen auf Typen und Member des AST zu durchsuchen, wobei die AST-Knoten durch Instanzen aus dem ProjectDom identifiziert werden.
Sollen alle Referenzen innerhalb einer Solution auf einen AST-Knoten gefunden werden, so ist zunächst zu untersuchen, innerhalb welcher Dateien die Entität überhaupt referenziert werden könnte. Dafür ist es notwendig, dessen Sichtbarkeit zu betrachten.
Ist ein Typ zum Beispiel `public`, so ist in allen Dateien der Solution nach Referenzen zu suchen. Ist der Typ hingegen `internal`, so reichen die Dateien des Projekts, in dem auch der Typ selbst deklariert ist. Bei ineinander geschachtelten Typen (sog. „nested Types“) sowie allen Membern eines Typs, ist zusätzlich zum Zugriffsmodifizierer der Entität, die Sichtbarkeit aller darüber liegenden Typen zu beachten. Eine private Methode innerhalb eines Typs kann zum Beispiel nur in der Datei referenziert werden, in der auch der Typ deklariert ist. Öffentliche Member innerhalb eines als `internal` deklarierten Typs können nur innerhalb von Dateien desselben Projekts referenziert werden. Sind damit alle Dateien gefunden, in denen das Objekt theoretisch referenziert werden könnte, so sind vorweg die dazugehörigen NRefactoryOld `CompilationUnit`-Instanzen zu erzeugen. Anschließend kann der `FindMemberAstVisitor` jede einzelne Instanz analysieren und nach Referenzen auf die übergebenen ProjectDom-Objekte suchen. Das Vorgehen des `FindMemberAstVisitor` sei im Folgenden vereinfacht skizziert:

- Der Visitor untersucht jedes durch den Parser aufgelöste Codefragment, das theoretisch das übergebene ProjectDom-Objekt referenzieren könnte.
  Da der NRefactoryOld-Parser jeden Zugriff auf lokale Felder und Propertys mit einer `IdentifierExpression` abbildet, wird bei diesen Instanzen überprüft, ob deren Bezeichner gleich dem Namen des gesuchten Objekts ist und es sich somit um eine potenzielle Referenz handelt. Lokal bedeutet dabei, dass das Feld bzw. Property innerhalb der aktuellen Klasse deklariert ist und keinen Receiver besitzt. Feld- und Propertyzugriffe, die sich auf einen Receiver beziehen, werden durch `MemberReferenceExpressions` abgebildet. Stimmt hier der Bezeichner mit dem gesuchten Objekt überein, so ist ebenfalls eine potenzielle Referenz gefunden.

Methodenaufrufe werden sowohl bei lokalen als auch entfernten Aufrufen durch `InvocationExpressions` abgebildet und sind bei der Suche nach Referenzen auf Methoden zu beachten.

- Hat der `FindMemberAstVisitor` über diese Namensvergleiche mögliche Referenzen gefunden, so wendet er sich an den `NRefactoryResolver`, um identifizierte Codefragmente eindeutigen Instanzen aus dem ProjectDom zuzuordnen. Wenn die zugeordnete Instanz der ursprünglich übergebenen Instanz entspricht, nach deren Referenzen gesucht werden sollte, ist eine Referenz gefunden.

Damit sind die Komponenten MonoDevelops, die bei der Analyse der Solution zum Erzeugen der Faktenbasis verwendet werden, erschöpft. Für weitergehende Details mussten eigene Code-Analysen implementiert werden.

4. C#-Sprachdefinition

Wie in 2.3 erläutert, ist Refacola generell unabhängig von der Sprache des Quellcodes, der zu refaktorisieren ist. Damit dies möglich ist, muss u.a. eine Sprachdefinition für die Zielsprache vorliegen. Aus diesem Grund wurde im Rahmen dieser Arbeit eine C#-Sprachdefinition entworfen, welche sich von dem Modell für Java gemäß [SvP11, S. 18] ableitet und alle Querys und Kinds abbildet, die zur Durchführung des „Pull-Up-Field"-Refactorings notwendig sind. In den folgenden Abschnitten werden die Unterschiede zur Java-Implementierung erläutert. Das vollständige Modell befindet sich im Anhang unter A.

4.1. Namespaces

In Java werden Klassen und Interfaces gemäß [Gos96, S. 153-172] in Paketen (engl. Package) strukturiert. Sie können hierarchisch angeordnet werden und dienen dazu Namenskonflikte zwischen Typen zu vermeiden. Außerdem kann es sinnvoll sein Klassen und Interfaces, die der gleichen Geschäftsdomäne zuzuordnen sind, in dem gleichen Paket zu platzieren. C# bietet mit Namespaces (vgl. [ECM06, S. 46]) ein ähnliches Konzept. Auch Namespaces helfen Namenskonflikte zu vermeiden und können hierarchisch angeordnet werden. Im Gegensatz zu Paketen haben Namespaces jedoch keinen Einfluss auf die Sichtbarkeit von Typen. Die Refacola-Sprachdefinition für Java aus [SvP11, S. 18] bildet Pakete mit Programmelementen des Kinds `Package` ab. Der Wertebereich dieses Elements erstreckt sich über alle im Programmcode verfügbaren Pakete. In dem Modell für C# werden Namespaces durch Programmelemente vom Kind `Namespace` abgebildet. Der Wertebereich ist hier - analog zu Java - als die Menge aller verfügbaren Namespaces definiert.

4.2. Structs, Enums und Delegates

Die Sprachdefinition für Java sieht als mögliche Typen lediglich Klassen, Interfaces und Enums als Spezialklassen vor. Nach [ECM06, S. 15-64] sieht die C#-Spezifikation darüber

hinaus noch Delegates und Structs vor.
Diese werden durch die Kinds `TopLevelDelegate` bzw. `NestedDelegate` und `TopLevelStruct` bzw. `NestedStruct` abgebildet.

4.3. Propertys

Neben Feldern, Methoden und Konstruktoren, die auch im Modell für Java abgebildet werden, stehen in C# Propertys zur Verfügung. Propertys implementieren mindestens einen Getter oder Setter. Es kann außerdem sowohl ein Getter als auch ein Setter definiert werden (vgl. [ECM06, S. 300-305]). Propertys besitzen einen eigenen Zugriffsmodifizierer. Darüber hinaus ist es möglich, entweder für den Getter oder den Setter - nicht jedoch für beide - eine Sichtbarkeit zu definieren, die eingeschränkter ist, als die Sichtbarkeit des dazugehörenden Propertys. Propertys werden durch Programmelemente des Kinds `Property` abgebildet. Getter bzw. Setter werden durch Programmelemente vom Kind `PropertyGetter` bzw. `PropertySetter` abgebildet. Darüber hinaus erhalten sie die Eigenschaft `Property`, welche vom Kind `Property` ist und den Bezug zum umschließenden Property herstellt.

4.4. Zugriffsmodifizierer

Java unterscheidet folgende Sichtbarkeiten (vgl. [Gos96, S. 138-144]):

$$public > protected > package > private$$

Der „Größer als“ Operator - > - stellt dabei zwei Sichtbarkeiten in Relation zueinander, wobei die Sichtbarkeit der rechten Seite weiter eingeschränkt ist als die der linken Seite. Die Zugriffsmodifizierer befinden sich in einer strengen Totalordnung, was bedeutet, dass jedes mögliche Paar von Sichtbarkeiten in Relation zueinander steht.
Gemäß [ECM06, S. 90] unterteilt C# dagegen folgende Sichtbarkeiten:

$$public > protected\,internal > \{protected, internal\} > private$$

Diese Anordnung soll suggerieren, dass *protected* und *internal* in keiner Relation zueinander stehen. Beide Sichtbarkeiten sind jedoch dem zusammengesetzten *protected internal* unterlegen, als auch *private* überlegen. Somit handelt es sich bei dieser Menge von Zugriffsmodifizierern um eine Halbordnung. Zum Zeitpunkt der Ausarbeitung dieser Arbeit ist die Refacola-Infrastruktur noch dahingehend anzupassen, dass sie Sichtbarkeiten, die durch eine Halbordnung abgebildet werden, unterstützt. Aufgrund dieser Unterschiede definiert die C#-Sprachdefinition den Wertebereich der `AccessModifiers`-Domäne u.a. über folgende Werte: `public, protected internal, protected, internal` und `private`.

4.5. Initialisierer

Im Refacola-Java-Modell werden Initialisierer durch Kinds abgebildet, die von `MemberOrConstructorOrInitializer` ableiten. In C# gibt es keinen dedizierten Initialisierer-Code. Alle Initialisierungen von Feldern, die direkt innerhalb ihrer Deklaration durchgeführt werden, sind implizit Teil des Initialisierers. Dieser wird vor jedem Konstruktoraufruf automatisch ausgeführt. Da es keinen expliziten Bereich gibt, wird der Initialisierer im Refacola-C#-Modell nicht abgebildet.

4.6. Zusammenfassung

Außer den beschriebenen Elementen gibt es weitere Merkmale im Refacola-C#-Modell, welche sich von der Java-Version unterscheiden. Da diese jedoch nichts mit der Spezifikation der Sprache selbst zu tun haben, sondern im Zuge von Verbesserungsmaßnahmen geändert wurden, befindet sich die Erläuterung im Anhang unter C.

5. C#-Refacola-Plugin

5.1. Schnittstelle zwischen dem Plugin und Refacola

Abschnitt 2.3 ist bereits auf die Infrastruktur eingegangen, in der Refacola eingebettet ist. Im zeitlichen Rahmen dieser Arbeit war es nicht möglich, Refacola vollständig in die Entwicklungsumgebung zu integrieren. Aus diesem Grund wird das entwickelte Plugin nicht unmittelbar von den Refacola-Komponenten verwendet. Stattdessen erfolgt die Kommunikation über eine dateibasierte Schnittstelle. Abb. 5.1 veranschaulicht das Zusammenspiel zwischen den verschiedenen Komponenten.

Das Plugin ist in der Lage, das geladene Projekt auf Wunsch zu analysieren. Dazu greift es auf den AST zu, an den es über die Schnittstelle zur IDE gelangt. Die Analyseergebnisse werden als Faktenbasis in eine Datei geschrieben. Der Inhalt der Datei hält sich dabei an einen Vertrag, der sowohl der Refacola-Komponente, als auch dem Plugin bekannt ist. Dieser Vertrag besteht zum einen aus der Sprachdefinition, welche die verfügbaren Kinds sowie die darauf basierenden Querys spezifiziert, als auch aus einer Festlegung, wie das Dateiformat auszusehen hat, d.h. in welcher Form die Programmelemente und Fakten enkodiert werden.
Da Refacola der Vertrag bekannt ist, ist es in der Lage, die Datei einzulesen. Dieser Vorgang wird manuell angestoßen. Basierend auf dieser Faktenbasis wird die Refaktorisierung durchgeführt und ein Change Set geschrieben.
Auch diese Datei hält sich an einen Vertrag, der sowohl dem Plugin, als auch Refacola bekannt ist. Das Change Set legt fest, welche Propertys von - in der Faktenbasis vorhandenen - Programmelementen geändert werden sollen. Anschließend muss das Plugin dazu veranlasst werden, das Change Set zu laden und die geforderten Änderungen am AST durchzuführen.
Sowohl der Vertrag der Faktenbasis, als auch der Vertrag der Change Set Dateien, sieht vor, dass die Informationen in Textform im UTF8-Format abgespeichert werden. Programmelemente und Fakten werden dabei gemäß dem in Listing 5.1 dargestellten Format abgebildet.

Abbildung 5.1.: Schnittstelle zwischen Refacola und dem entwickelten Plugin

Listing 5.1: Aufbau einer Faktenbasis

```
ProjectName {

        ProgramElementType1 UniqueProgramElementId1
            Property1    Value
            ...
            PropertyN    Value;

        ...

        ProgramElementTypeN UniqueProgramElementIdN
            Property1    Value
            ...
            PropertyN    Value;


        Fact1(Arg1, ..., ArgN);
        ...
        FactN(Arg1, ..., ArgN);
}
```

Anfangs wird das Projekt bezeichnet, hier „ProjectName“. Darauf folgen alle Programmelemente, die gemäß der Sprachdefinition den AST beschreiben. Jedes Programmelement wird von einer Reihe Propertys, die das Element auszeichnen, begleitet. Abschließend stellen die Fakten die Programmelemente in Relation zueinander. Listing 5.2 veranschaulicht das für Change Sets geltende Format.

Listing 5.2: Aufbau eines Change Sets

```
UniqueProgramElementId1?Property1    -> NewValue
...
UniqueProgramElementIdN?PropertyN    -> NewValue
```

Jede Zeile des Change Sets beschreibt die Änderung eines Propertys von einem Programmelement. Dabei ist der eindeutige Bezeichner des Programmelements vom Namen des zu ändernden Propertys durch ein - im Change Set Vertrag vereinbartes - Zeichen zu trennen. In diesem Fall wird als Trennzeichen das Fragezeichen „?“ gewählt. Das Property wird vom neuen Wert wieder durch ein vereinbartes Zeichen getrennt. In obigem Fall wird die Zeichenkette „->“ verwendet.

5.2. Architektur

Das im Rahmen dieser Arbeit entwickelte Plugin basiert vollständig auf .NET 4.0 und setzt sich aus folgenden vier Modulen zusammen, deren Interaktion Abb. 5.2 darstellt:

- `Contracts`: Definition aller gemeinsam genutzten Interfaces und Datenstrukturen.
- `AST-Access`: Stellt lesenden und schreibenden Zugriff auf den AST zur Verfügung.
- `Refacola-Interface`: Generiert die Faktenbasis und verarbeitet Change Sets.
- `MonoDevelop-Extension`: Verantwortlich für die Einbettung in MonoDevelops IDE.

Bei der Entwicklung wurde großen Wert auf eine modulare Architektur gelegt, um eine lose Kopplung zwischen den Komponenten zu schaffen und dadurch ein leichtes Austauschen einzelner Module zu ermöglichen. Entsprechend befinden sich alle Komponenten in eigenen .NET-Assemblys, wobei sich die Klassen aller Assemblys in einem übergeordneten Namespace[1] befinden. Die folgenden Abschnitte gehen im Detail auf die Module ein.

5.3. Komponente Contracts

In der `Contracts`-Assembly werden alle Verträge definiert, die zwischen den unterschiedlichen Komponenten bestehen. Diese Verträge liegen in Form von Interfaces und Datenstrukturen vor, die über die Interfacegrenzen ausgetauscht werden. Dieses Modul ist komplett autark, besitzt also keinerlei Abhängigkeiten zu anderen Komponenten. Im Folgenden werden die unterschiedlichen Verträge erläutert.

5.3.1. AST-Modell

Das Modul definiert ein AST-Modell, über das alle Informationen abgefragt werden können, die für die Generierung einer vollständigen Faktenbasis notwendig sind. Dieses Modell wird mit Hilfe von Interfaces beschrieben und ist damit von einer konkreten IDE losgelöst. Für jede Entwicklungsumgebung, die unterstützt werden soll, muss somit lediglich ein Modul bereitgestellt werden, welches eine AST-Repräsentation gemäß dem festgelegten Modell zur Verfügung stellt. Abb. 5.3 illustriert schematisch die Elemente des Modells sowie deren Zusammenhang.

[1] `FernUniHagen.Programmiersysteme.Refacola.MonoDevelop.CS`

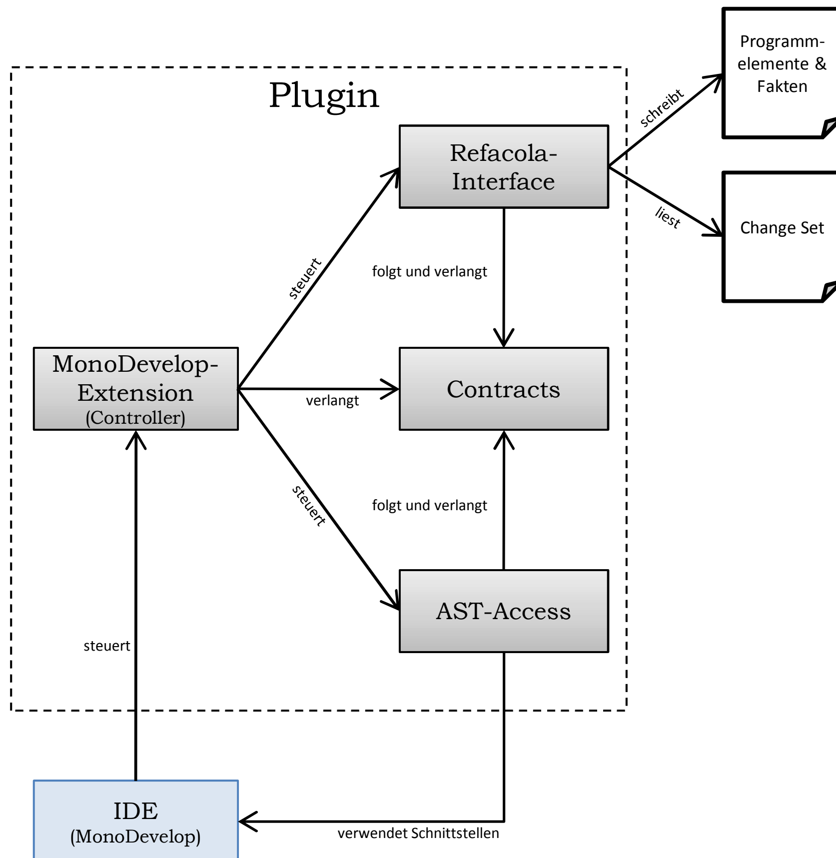


Abbildung 5.2.: Architektur des entwickelten Plugins

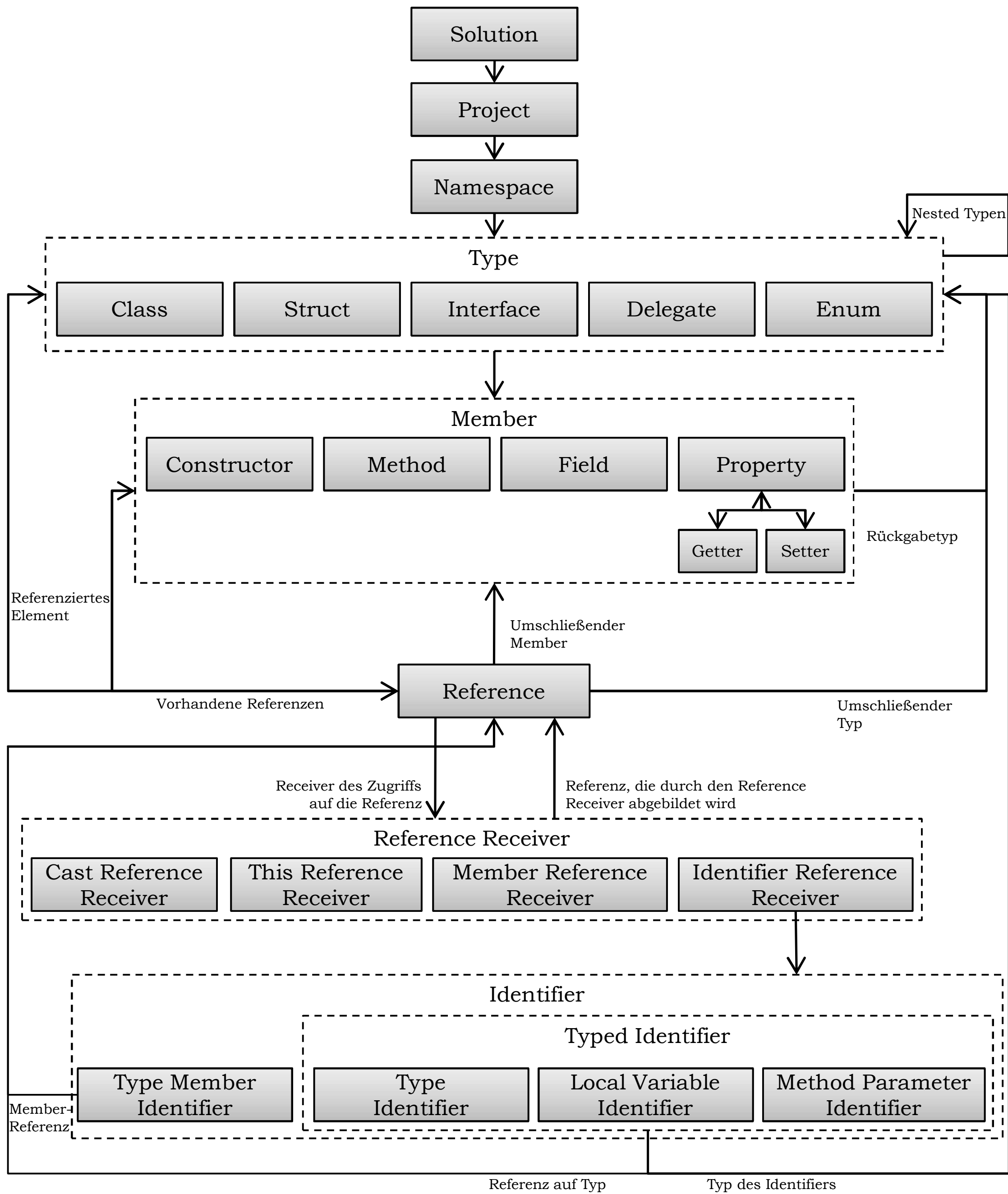


Abbildung 5.3.: Dem Plugin zu Grunde liegendes AST-Modell

Der Einstiegspunkt in den AST beginnt bei der geladenen Solution. Diese gibt Zugriff auf alle geöffneten Projekte, die Teil der Solution sind. Die Projekte verweisen auf enthaltene Namespaces. Über die Namespaces gelangt man an alle darin deklarierten Typen, welche selbst wiederum alle darin enthaltenen verschachtelten Typen referenzieren. Das Modell unterscheidet dabei die C#-Typen `Class`, `Struct`, `Interface`, `Delegate` und `Enum`. Die jeweiligen Typ-Instanzen gewähren Zugriff auf darin deklarierte Member. Die Familie der Member umfasst `Constructor`, `Method`, `Field` und `Property`, wobei Propertys noch Referenzen auf ihren optionalen Getter und Setter halten. Bis auf `Constructor` ist es außerdem möglich, zu allen Membern den Rückgabetyp zu erhalten. Die möglichen Member, die über Typ-Instanzen abgefragt werden können, sind je nach Typ unterschiedlich. So können Interfaces zum Beispiel keine Konstruktoren definieren. Aus Gründen der besseren Lesbarkeit verzichtet die Darstellung allerdings auf solche Details. Daneben veröffentlichen die Typ- und Member-Instanzen den Pfad zu der Datei, in der sie deklariert sind, wenn diese innerhalb der Solution vorliegt.
Über Typ- und Member-Instanzen ist man weiter in der Lage, alle auf sie verweisenden Referenzen innerhalb der Solution zu erhalten. Referenzen gewähren Zugriff auf den umschließenden Typ sowie, falls vorhanden, den Member, innerhalb dessen Deklaration die Referenz existiert. Desweiteren erhält man über Referenz-Instanzen auch wieder Zugriff auf die Entität (d.h. Typen oder Member), auf die verwiesen wird. Das AST-Modell sieht außerdem vor, dass man über Referenz-Instanzen Zugriff auf den Empfänger der Referenz, den `Reference Receiver`, erhält. Listing 5.3 veranschaulicht die verschiedenen Receiver-Typen, die das Modell unterscheidet. Die Demonstration verwendet dabei exemplarisch den schreibenden Zugriff auf das Feld `MyField` der Klasse `MyClass`.

Listing 5.3: Beispiele für die möglichen Receiver-Typen des AST-Modells

```
1 namespace ReceiverScenarios
2 {
3     class MyClass
4     {
5         int MyField;
6
7         void CastReferenceReceiverDemo()
8         {
9             object k = new MyClass();
10
11            ((MyClass)k).MyField = 0;
12        }
13
14        void ThisReferenceReceiverDemo()
15        {
16            MyField = 0;
17            this.MyField = 0;
18        }
19
20
21        void MemberReferenceReceiverDemo()
22        {
23            InstanceProviderMethod().MyField = 0;
24        }
25
26        MyClass InstanceProviderMethod()
27        {
28            return new MyClass();
29        }
30
31
32        void IdentifierReferenceReceiverDemo(...)
33        {
34            ...
35
36            someIdentifier.MyField = 0;
37        }
38    }
39 }
```

In Zeile 11 wird der Fall eines `CastReferenceReceivers` (vgl. Abb. 5.3) demonstriert. Diese Receiver beziehen sich auf Empfänger-Objekte, die das Ergebnis einer expliziten Typumwandlung (engl. Cast) sind. Die Typumwandlung stellt selbst eine Referenz auf den Zieltyp, hier `MyClass`, dar. Genau diese Referenz ist die empfangende Referenz des Zugriffs auf das Feld `MyField`. Sie lässt sich aus Instanzen vom Typ `CastReferenceReceiver` auslesen.
Die Zeilen 16 und 17 demonstrieren den Zugriff auf das Feld mittels eines `this`-Empfängers. Das Modell unterscheidet dabei sowohl implizite (vgl. Zeile 16), als auch explizite (vgl. Zeile 17) `this`-Referenzen. Dieser Receiver wird durch Instanzen des Typs `ThisReferenceReceiver` abgebildet. Diese Instanzen geben zum einen an, ob es sich dabei um eine implizite oder explizite Referenz handelt, und zum anderen gewähren sie Zugriff auf die `this`-Referenz selbst.
Darüber hinaus wird der Fall unterschieden, dass der Empfänger einer Referenz als Ergebnis eines Methodenaufrufs vorliegt. Hierbei ist die Empfänger-Referenz der Verweis auf die Methode. In Zeile 23 gibt die Methode `InstanceProviderMethod` eine Instanz vom Typ `MyClass` zurück, die dann weiterverwendet wird, um das Feld `MyField` zu setzen. Die empfangende Referenz ist in diesem Beispiel die Referenz auf die Methode `InstanceProviderMethod` aus Zeile 23. Diese Konstellation wird durch Instanzen des Typs `MemberReferenceReceiver` abgebildet.
Neben den beschriebenen Fällen existiert noch das in Zeile 36 angegebene Szenario. Hierbei ist der Empfänger des Feldzugriffs ein Bezeichner, hier `someIdentifier`. Die Referenz auf diesen Bezeichner wird in `IdentifierReferenceReceiver`-Instanzen festgehalten. Diese Bezeichner können sich nun auf verschiedene Arten von Objekten beziehen. Deswegen sieht das AST-Modell vor, dass über Instanzen des Typs `IdentifierReferenceReceiver` der zugrunde liegende Bezeichner näher bestimmt werden kann (vgl. hierzu Abb. 5.3). Listing 5.4 veranschaulicht die unterschiedenen Szenarien.

Listing 5.4: Beispiele für die möglichen Bezeichner-Typen des AST-Modells

```
1 namespace IdentifierReceiverScenarios
2 {
3     class MyClass
4     {
5         int MyField;
6
7         void TypeMemberIdentifierReferenceReceiverDemo()
8         {
9             InstanceProviderGetter.MyField = 0;
10        }
11
12        MyClass InstanceProviderGetter
13        {
14            get
15            {
16                return new MyClass();
17            }
18        }
19
20        static int MyStaticField;
21        void TypeIdentifierReferenceReceiverDemo()
22        {
23            MyClass.MyStaticField = 0;
24        }
25
26        void LocalVariableIdentifierReferenceReceiverDemo()
27        {
28            MyClass myLocalVariable = new MyClass();
29            myLocalVariable.MyField = 0;
30        }
31
32        void ArgIdentifierReferenceReceiver(MyClass arg)
33        {
34            arg.MyField = 0;
35        }
36    }
37 }
```

Wenn es sich bei dem Bezeichner, wie in Zeile 9, um ein Member eines Typs handelt, so wird dieser mittels einer `IdentifierAsMemberDeclaration`-Instanz charakterisiert. Diese Instanz enthält die Referenz auf den empfangenden Member des Ausdrucks. In diesem Beispiel enthält die `IdentifierAsMemberDeclaration`-Instanz somit die Referenz aus Zeile 9 auf das Property `InstanceProviderGetter`.
Im Szenario aus Zeile 23 verweist der empfangende Bezeichner, hier „MyClass", auf die Klasse `MyClass`. Er wird durch eine `IdentifierAsTypeDeclaration`-Instanz beschrieben. Diese enthält einen Verweis auf den Typ selbst, hier `MyClass`, sowie die Referenz auf die Klasse `MyClass` aus Zeile 23. Dieser Fall tritt beim Zugriff auf statische Member auf.
Desweiteren ist es auch möglich, dass sich ein Bezeichner auf eine lokale Variable bezieht. Im Beispiel aus Zeile 29 verweist der Bezeichner „myLocalVariable" auf die Instanz `myLocalVariable` vom Typ `MyClass`, aus Zeile 28. In solchen Fällen wird der Bezeichner über `IdentifierAsLocalVariableDeclaration`-Instanzen repräsentiert. Diese enthält zum einen den Typ der Variable - hier `MyClass` - und zum anderen die Referenz auf den Typ, der im Zuge der Deklaration in Zeile 28 verwendet wird.
Bezieht sich der Bezeichner nicht auf eine lokal deklarierte Variable, sondern auf ein Übergabeargument einer Methode, so wird er über Instanzen der Klasse `IdentifierAsMethodParameterDeclaration` abgebildet. Analog zu `IdentifierAsLocalVariableDeclaration` enthält auch diese Instanz einen Verweis auf den Typ des Arguments sowie auf die Typ-Referenz selbst. Im Anhang unter D ist das vollständige Modell als Klassendiagramm abgebildet.

5.3.2. Refaktorisierungen

Mit Hilfe des AST-Modells können alle Programmelemente und Fakten erzeugt werden, die für die vollständige Repräsentation einer Solution notwendig sind. Ziel der Refacola-Infrastruktur ist es nun auf Basis dieser Daten eine fehlerfreie Refaktorisierung durchzuführen. Dazu muss schreibend auf den AST zugegriffen werden können. Zu diesem Zweck enthält die `Contracts`-Assembly eine Sammlung von Datenstrukturen, die Änderungen der AST-Knoten beschreiben. Im Rahmen dieser Arbeit wurden folgende Modifikationen vorgesehen:

- Änderungen des Zugriffsmodifizierers von Typen und Membern werden über die Klasse `ChangeEntityAccessModifierRefactoringInput` beschrieben.
  Instanzen von diesem Typ enthalten zum einen die Entität, die geändert werden soll, und zum anderen den neuen Zugriffsmodifizierer. Der Zugriffsmodifizierer umfasst neben den für C# gültigen Werten (vgl. Abschnitt 4.4) noch den Modifizierer `None`. Dieser ist für den Fall vorgesehen, dass bei einem PropertyGetter oder PropertySetter die Sichtbarkeit nicht explizit angegeben werden soll. In diesem Fall ist sie identisch zu dem definierenden Property (vgl. Abschnitt 4.3).

- Soll der Bezeichner einer Entität (also eines Typs oder Members) geändert werden, so werden die Parameter dieser Modifikation in Instanzen der Klasse `ChangeEntityIdentifierRefactoringInput` vorgehalten. Dazu wird sowohl die betreffende Entität, als auch der neue Bezeichner festgelegt.

- Sollen Refaktorisierungen wie das „Pull-Up-Field"-Refactoring durchgeführt werden, so ist es notwendig, dass sich die Deklaration einer Entität in einen anderen Typ verschiebt. Instanzen der Klasse `ChangeEntityOwnerRefactoringInput` beschreiben diese Änderungen. In dem Datencontainer werden die betroffene Entität, die verschoben werden soll, sowie der Typ, in den die Deklaration verschoben werden soll, angegeben.

- Wenn der Bezeichner von Entitäten geändert wird, ist es meistens nicht ausreichend, den Bezeichner ausschließlich in der Deklaration zu ändern. Sobald Referenzen auf die Entität existieren, müssen auch diese geändert werden. Diese Modifikationen werden durch `ChangeReferenceIdentifierRefactoringInput`-Instanzen beschrieben. Sie enthaltenen zum einen die Referenz, die geändert werden soll, und zum anderen den neuen Bezeichner.

Die erläuterten Datencontainer beschreiben dabei lediglich die durchzuführenden Modifikationen. Sie enthalten keinerlei Logik und sind damit - genau wie das AST-Modell - von konkreten Entwicklungsumgebungen losgelöst.

5.3.3. Verschiedenes

Neben dem AST-Modell und den möglichen Quellcodeänderungen enthält die `Contracts`-Assembly noch eine Reihe weiterer, gemeinsam genutzter, Datenstrukturen und Interfaces. So wird beispielsweise das `IProgressCallback`-Interface modulübergreifend verwendet, um den Fortschritt einer Operation auszugeben. Auch der in allen Komponenten benutzbare Protokolliermechanismus (engl. Logging) ist hier angesiedelt. Da diese Konstrukte jedoch nichts mit der Thematik dieser Arbeit direkt zu tun haben, wird an dieser Stelle auf eine ausführlichere Erläuterung verzichtet.

5.4. Komponente AST-Access

Die Komponente `AST-Access` ist die konkrete Implementierung der in der `Contracts`-Assembly definierten Verträge für die Entwicklungsumgebung MonoDevelop. Über die Methode `RetrieveSolution` der statischen Klasse `MonoDevelopAstProvider` stellt sie einen AST zur Verfügung, der dem vereinbarten Modell folgt. Außerdem nimmt sie über

die Klasse `RefactoringController` (vgl. Abschnitt 5.4.2) die Beschreibung gewünschter Codemodifikationen (vgl. 5.3.2) entgegen und führt diese durch. Abb 5.4 illustriert, wie `AST-Access` mit den in Abschnitt 3 beschriebenen Schnittstellen von MonoDevelop interagiert.

Die folgenden Abschnitte erläutern im Detail, wie die Komponente den AST gemäß dem vereinbarten Modell zur Verfügung stellt und Modifikationen an diesem vornimmt.

5.4.1. Implementierung des AST-Modells

Die Implementierung der AST-Repräsentation lässt sich in vier Teilaufgaben unterscheiden:

- Eine geladene Solution muss in eine Baumstruktur aufgelöst werden, wobei die Solution den Wurzelknoten, und die Member der Typen die Blätter des Baumes darstellen.
- Der Baum muss in einen Graph, der Zyklen enthält, abgebildet werden. Das bedeutet, dass die Rückgabetypen der Baumblätter (also Member der Typen) auf Knoten des Baumes verweisen.
- Der Graph muss dahingehend erweitert werden, dass - ausgehend von den Baumknoten - auf die darauf verweisenden Referenzen im Quellcode zugegriffen werden kann.
- Im letzten Schritt müssen die Receiver der Referenzen aufgelöst werden können. Hierbei befindet sich der `Identifier Receiver` in einer besonderen Rolle, da sich ein Bezeichner nach dem AST-Modell auf lokale Variablen, Methodenargumente, statische Klassen sowie lokale Felder und Propertys beziehen kann.

Die erste Teilaufgabe lässt sich durch das in Abschnitt 3.1 beschriebene ProjectDom lösen. Das ProjectDom definiert bereits Klassen, welche die Typen und deren Member eines ASTs abbilden. `AST-Access` implementiert für alle Typen und Member `Container`-Klassen, welche die vertraglich festgelegten Interfaces des AST-Modells implementieren und den Zugriff auf die Knoten an die Instanzen des ProjectDoms delegieren. Da das ProjectDom erst auf Projektebene beginnt (vgl. Abb. 3.1), muss für die Repräsentation des Wurzelknotens, also der Solution, eine besondere Klasse implementiert werden, welche den Zugriff über die IDE-API auf die Informationen der geladenen Solution sowie der darin enthaltenen Projekte kapselt. Abb. 5.5 veranschaulicht dieses Vorgehen.

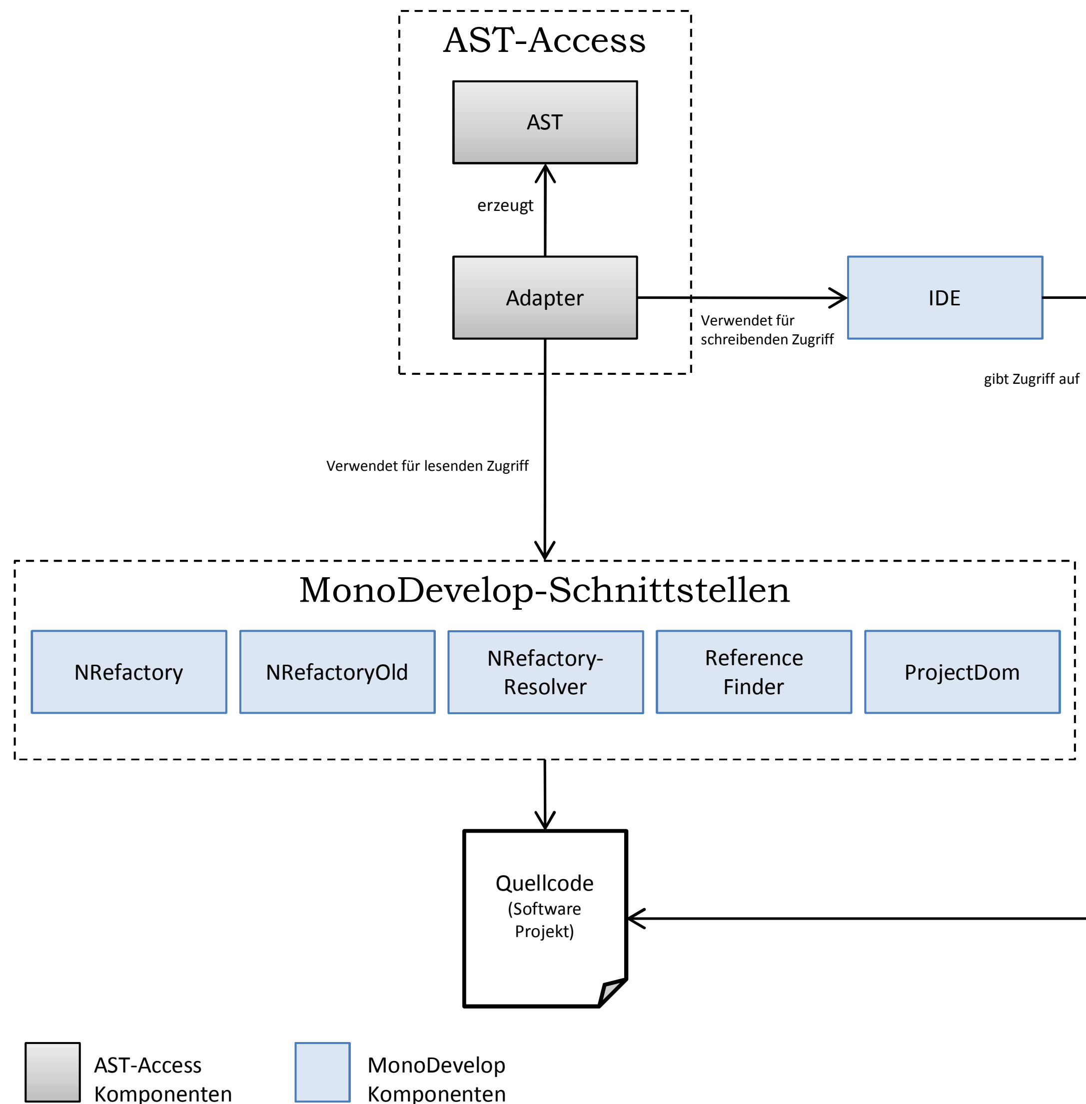
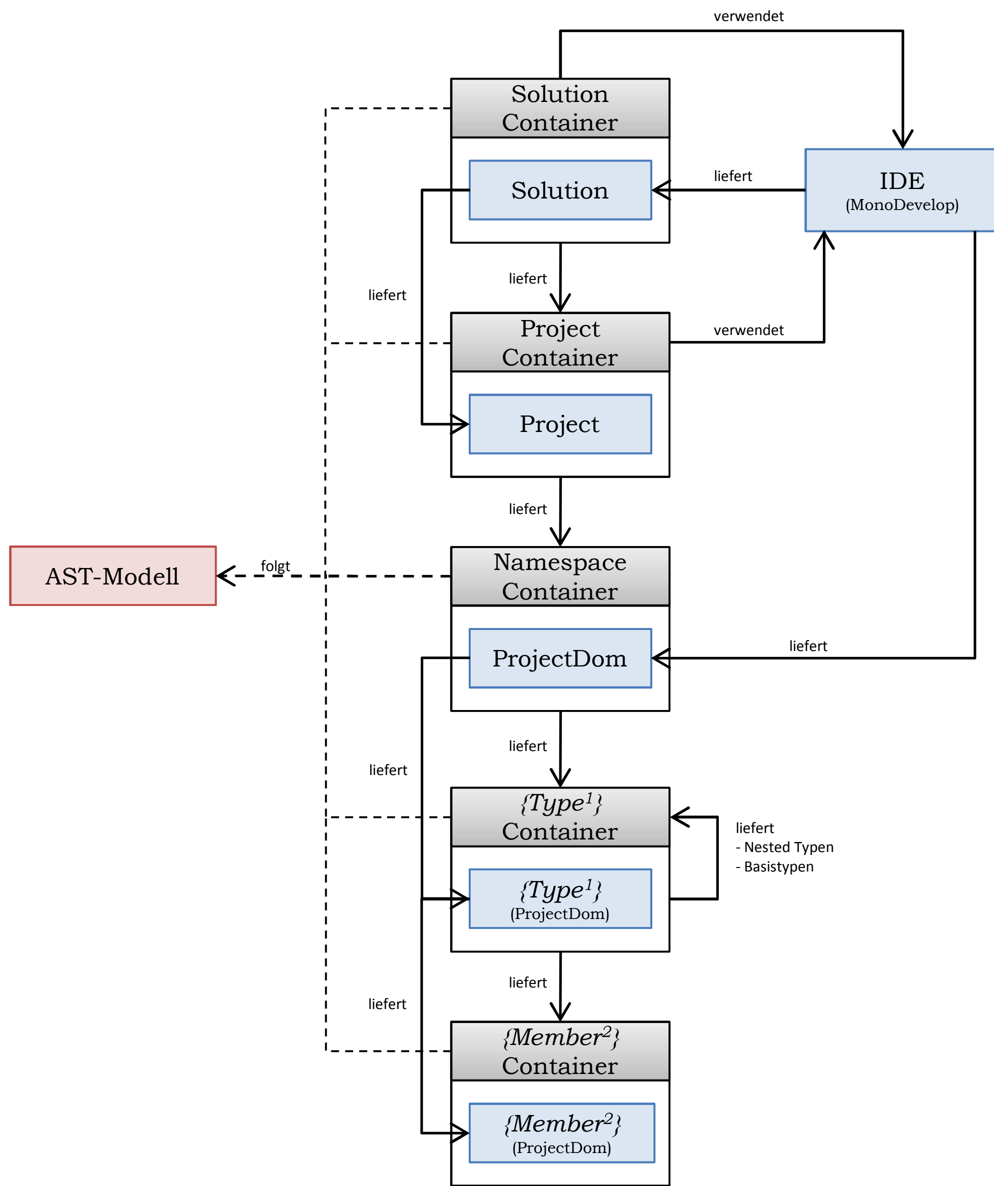


Abbildung 5.4.: Interaktion der AST-Access-Komponente mit MonoDevelop



1: Class, Interface, Struct, Enum, Delegate

2: Method, Property, Field

Abbildung 5.5.: Implementierung des AST-Baumes

Die Abbildung verdeutlicht, dass zuerst von der IDE eine `Solution`-Instanz angefordert wird. Diese wird in einem `SolutionContainer` gekapselt. Der `SolutionContainer` stellt für alle darin enthaltenen Projekte einen `ProjectContainer` zur Verfügung. Über die `Project`-Instanzen gewährt die IDE Zugriff auf das dazugehörende ProjectDom. Da das Objektmodell von MonoDevelop Namespaces nicht als eigene Entitäten abbildet, enthält der `NamespaceContainer` direkt eine Instanz des Typs `ProjectDom`. Werden von einem `NamespaceContainer` die darin enthaltenen Typen angefordert, so gibt dieser nur die Typen zurück, die im entsprechenden Namespace deklariert wurden. Aus Gründen der Übersichtlichkeit werden hier nicht die Container aller Typen aufgeführt, sondern stattdessen die abstrakte Basisklasse `TypeContainer` dargestellt. Für die verschiedenen Typen liegen jedoch konkrete `Container`-Klassen vor. Über diese `Container`-Klassen erhält man schließlich - wenn vorhanden - die zum Typ gehörenden Member, welche ebenfalls nicht einzeln aufgeführt werden. Die `Container`-Klassen implementieren dabei die in der `Contracts`-Assembly definierten Interfaces und stellen somit gemeinsam eine Repräsentation des AST-Modells dar.
Wie in Abschnitt 3.1 bereits angesprochen, werden im ProjectDom Rückgabetypen von Membern, sowie Basisklassen und implementierte Interfaces über den Typ `ReturnType` abgebildet. Diese Klasse ist keine vollständige Repräsentation des Typs, sondern bildet lediglich den Namen des Typs ab. Deshalb enthält der AST, der vom ProjectDom zur Verfügung gestellt wird, auch keine Zyklen. Im AST-Modell, das diesem Plugin zu Grunde liegt, werden hingegen immer konkrete Typen referenziert. Das hat zur Folge, dass die `ReturnType`-Instanzen des ProjectDoms beim Zugriff auf Rückgabetypen oder Basisklassen direkt vom jeweiligen `Container` aufgelöst werden. Dies geschieht für den Aufrufenden vollkommen transparent.
Darüber hinaus sieht das AST-Modell vor, dass ausgehend von `Type`- und `MemberContainern`, auf die darauf verweisenden Referenzen zugegriffen werden kann. Hierzu wurde die Klasse `ProjectDomReferenceFinder` im Namespace `ProjectDomAstModel.ReferenceResolving` implementiert. Diese erwartet Typ- oder Member-Instanzen des ProjectDoms und reicht diese an den in Abschnitt 3.4 vorgestellten `FindMemberAstVisitor` weiter. Da die `Container`-Klassen die Instanzen des ProjectDoms kapseln, können sie diese problemlos an den `ProjectDomReferenceFinder` übergeben, sobald die Referenzen abgefragt werden. Gefundene Referenzen werden durch `EntityReferenceContainer` repräsentiert, welche das entsprechende Interface des AST-Modells implementieren.
Das AST-Modell sieht desweiteren vor, dass für die Referenzen der dazu gehörende Receiver ausgelesen werden kann. Hierfür konnte sich keiner MonoDevelop-Komponente bedient werden. Die Analyse des Referenzempfängers geschieht mit Hilfe der in Abschnitt 3.3 beschrieben NRefactory-Komponente. Hierbei wird eine Visitor-Klasse implementiert, die die Datei, in der die gesuchte Referenz enthalten ist, analysiert. Abb. 5.6 veranschaulicht den Algorithmus, der den Empfänger ausliest. Abb. 5.7 veranschaulicht die in Abb. 5.6 dargestellten Schritte des Algorithmus an ausgewählten Codefragmenten. Die Beispiele illustrieren die verschiedenen - vom Algorithmus identifizierbaren - Möglichkeiten, um auf ein Feld in C# (in diesem Fall `MyField`) zuzugreifen.

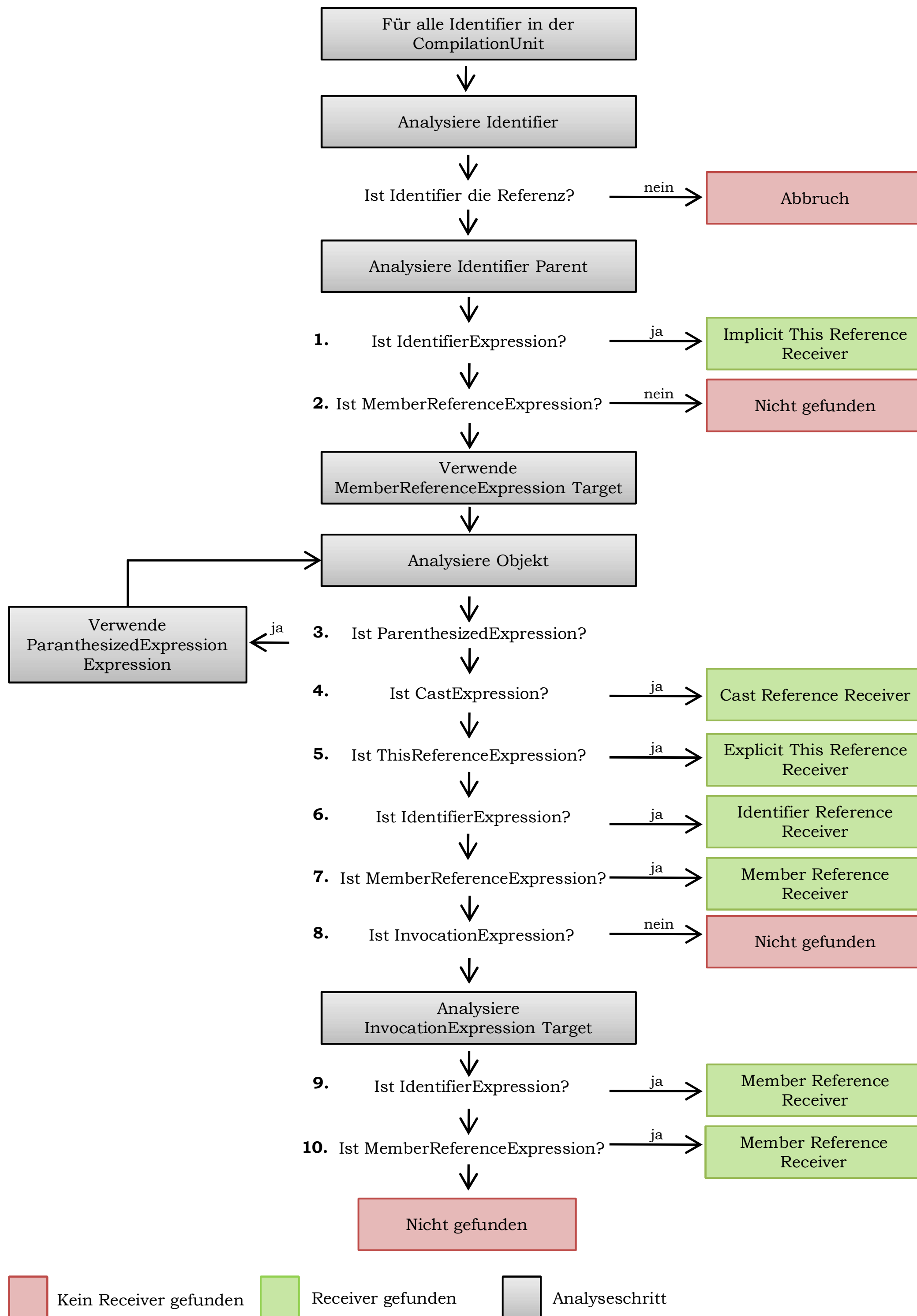


Abbildung 5.6.: Algorithmus zur Auflösung von Referenz Empfängern

```
1    class MyClass
2    {
3        int MyField;
4
5
6        void SimpleIdentifierExpressionDemo()
7        {
8              IdentifierExpression
9    1.        MyField = 0;
10             [...].Target
11       }
12
13
14       void GeneralMemberReferenceReceiverDemo()
15       {
16             MemberReferenceExpression
17   2.        [...].MyField = 0;
18             [...].Target  Identifier
19       }
20
21
22       void ParenthesizedExpressionDemo()
23       {
24             ParenthesizedExpression
25   3.        ([...]).MyField = 0;
26             [...].Expression
27       }
28
29
30       void CastExpressionDemo()
31       {
32           object k = new MyClass();
33
34   4.      ((MyClass)k).MyField = 0;
35             CastExpression
36       }
37
38
39       void ThisReferenceExpressionDemo()
40       {
41   5.      this.MyField = 0;
42           ThisReferenceExpression
43       }
44
45
```

```
46        void IdentifierExpressionDemo()
47        {
48   6.       someIdentifier.MyField = 0;
49             IdentifierExpression
50        }
51
52
53        void MemberReferenceExpressionDemo()
54        {
55              MemberReferenceExpression
56   7.       [...].FieldOrProperty.MyField = 0;
57                    [...].MemberName
58        }
59
60
61        void InvocationExpressionDemo()
62        { InvocationExpression
63
64   8.       [...]().MyField = 0;
65            [...].Target
66        }
67
68
69        void InvExpTargetAsIdentifierExpressionDemo()
70        {
71   9.       LocalMethod().MyField = 0;
72            IdentifierExpression
73        }
74
75
76        void InvExpTargetAsMemberReferenceExpressionDemo()
77        {
78   10.      [...].RemoteMethod().MyField = 0;
79            MemberReferenceExpression
80        }
81    }
```

Abbildung 5.7.: Beispiele für die Schritte des Algorithmus aus Abb. 5.6

Der Visitor besucht jeden einzelnen Bezeichner in der `CompilationUnit` und überprüft, ob es sich dabei um die Referenz handelt, für die der Receiver gesucht werden soll. Wenn dies nicht der Fall ist, beendet der Algorithmus die weitere Analyse des Bezeichners. Sollte es sich jedoch um die Referenz handeln, wird der Ausdruck analysiert, der die Referenz enthält. Hierbei werden die folgenden beiden Fälle unterschieden:

- Wenn es sich bei diesem Ausdruck um eine `IdentifierExpression` handelt, wird direkt auf das referenzierte Objekt zugegriffen. Das bedeutet, dass kein Empfänger explizit angegeben wurde. Somit handelt es sich um eine implizite `this`-Referenz und die `CompilationUnit` muss nicht weiter untersucht werden. Als Ergebnis wird ein `ThisReferenceReceiver` zurückgeliefert, der beschreibt, dass es sich in diesem Fall um eine implizite `this`-Referenz handelt (siehe Schritt 1).

- Die Alternative dazu ist, dass über ein Empfänger-Objekt auf das referenzierte Element zugegriffen wird. Der Zugriff auf Member eines Objekts wird in C# über einen Punkt („.“) eingeleitet. Der NRefactory-Parser beschreibt diese Zugriffsausdrücke in Instanzen der Klasse `MemberReferenceExpression` (siehe Schritt 2). Sollte es sich bei dem Ausdruck jedoch auch nicht um eine `MemberReferenceExpression` handeln, so kann der Empfänger nicht aufgelöst werden und die Suche wird erfolglos abgebrochen. Andernfalls wird dieser Zugriff auf das Member näher untersucht.

Das `Target`-Property von `MemberReferenceExpressions` beschreibt den C#-Ausdruck, der das Empfänger-Objekt zurückliefert. Dieser Ausdruck beschreibt also den Teil, der im Code links von dem Punkt steht. Er wird in den nachfolgenden Schritten analysiert.

- In Schritt 3 wird überprüft, ob sich der Ausdruck in Klammern befindet. In diesem Fall wird der Ausdruck, der innerhalb der Klammern steht, näher analysiert. Abb. 5.6 zeigt bereits, dass es sich in diesem Fall um einen rekursiven Algorithmus handelt. Schließlich können Klammerausdrücke auch mehrfach ineinander verschachtelt werden. Aus diesem Grund startet die Analyse des inneren Ausdrucks wieder bei Schritt 3.

- In Schritt 4 wird untersucht, ob es sich bei dem Ausdruck um einen expliziten Cast handelt. Diese Ausdrücke werden durch `CastExpression`-Instanzen beschrieben. Sollte das der Fall sein, ist der Empfänger gefunden. Als Ergebnis wird der vom Modell vorgeschriebene `CastReferenceReceiver` zurückgeliefert und der Algorithmus wird beendet.

- Schritt 5 stellt fest, wenn das Empfänger-Objekt eine explizite `this`-Referenz ist, indem überprüft wird, ob der Ausdruck vom Typ `ThisReferenceExpression` ist. Wenn dem so ist, wird ein `ThisReferenceReceiver` mit dem Vermerk, dass es sich um einen explizite `this`-Referenz handelt, zurückgegeben.

- Wenn das Empfänger-Objekt nur durch einen Bezeichner beschrieben wird, ist der Ausdruck vom Typ `IdentifierExpression` und der Algorithmus terminiert mit der Rückgabe eines `IdentifierReferenceReceivers` (siehe Schritt 6). Instanzen dieses Typs enthalten lediglich den Bezeichner sowie dessen genaue Stelle in der Datei. Dieser Bezeichner kann weiter analysiert werden, da es sich bei ihm um eine lokale Variable, ein Methodenargument, eine Referenz auf ein lokales Feld oder Property, sowie um den Zugriff auf eine statische Klasse handeln kann. Dieser Umstand geht auch aus dem Beispiel aus Abb. 5.7 hervor: Der hier dargestellte Bezeichner `someIdentifier` ist nicht näher deklariert worden. Für den NRefactory-Parser ist das jedoch auch unerheblich, denn unabhängig von dem Objekt, das sich hinter dem Bezeichner verbirgt, wird der Ausdruck immer mit einer `IdentifierExpression`-Instanz beschrieben. Die weitere Analyse des Bezeichners ist allerdings nicht mehr Aufgabe des Visitors, sondern wird durch die `IdentifierReferenceReceiver`-Instanzen ermöglicht. Darauf wird im nächsten Absatz genauer eingegangen.

- Sollte das Empfänger-Objekt ein Feld oder Property eines Objekts sein, so wird der Zugriff auf dieses Member durch eine `MemberReferenceExpression` beschrieben. Diese ist bereits aus Schritt 2 bekannt, jedoch mit dem Unterschied, dass in Schritt 2 das Member die gesuchte Referenz darstellt, wohingegen in Schritt 7 das Member das Empfänger-Objekt der Referenz abbildet. Der Algorithmus gibt an dieser Stelle eine Instanz des Typs `MemberReferenceReceiver` zurück, welche den Ort und Bezeichner der empfangenden Referenz beschreibt.

- Sollte keiner der vorangehenden Analyseschritte Erfolg haben, wird in Schritt 8 schließlich überprüft, ob es sich bei dem Ausdruck um eine `InvocationExpression` handelt. Wenn auch das nicht zutrifft, bricht der Algorithmus erfolglos ab. Methodenaufrufe werden durch `InvocationExpressions` beschrieben und enthalten in ihrem `Target`-Property den Ausdruck, über den die Methode angesprochen wird. Methodenreferenzen sind folglich der letzte, vom Algorithmus identifizierte, Typ von Empfänger-Objekt. Diese Methodenaufrufe werden in zwei Gruppen unterschieden.

- Wenn die Methode lokal ist und über den impliziten `this`-Receiver aufgerufen wird, enthält das `Target`-Property eine `IdentifierExpression`. In diesem Fall gibt der Algorithmus in Schritt 9 wieder einen `MemberReferenceReceiver` zurück. Dieser enthält den Namen der aufgerufenen Methode sowie die genaue Positionsangabe des Bezeichners.

- Die Alternative dazu ist, dass die Methode nicht lokal, sondern auf einem Empfänger-Objekt aufgerufen wird. In diesem Fall wird der Methodenaufruf durch eine `MemberReferenceExpression` beschrieben und der Algorithmus kann analog zu Schritt 10 einen `MemberReferenceReceiver` zurückgeben. Fällt die Beschreibung des Methodenaufrufs in keine der zwei Gruppen, bricht der Algorithmus erfolglos ab.

Zur Vervollständigung des Modells fehlt an dieser Stelle noch das Auflösen der Bezeichner, die durch die `IdentifierReferenceReceiver` beschrieben werden. Das AST-Modell schreibt vor, dass ausgehend von einer `IdentifierReferenceReceiver`-Instanz die Deklaration des Bezeichners aufgelöst werden kann. Hierfür werden lokale Variablen, Methodenargumente, Klassen sowie Member unterschieden. Die Analyse des Bezeichners geschieht durch den in Abschnitt 3.4 beschriebenen NRefactoryResolver. Das Ergebnis dieser Analyse wird in Klassen gekapselt, welche die entsprechenden Interfaces des AST-Modells aus der `Contracts`-Assembly implementieren. Konkret sind dies die im Abschnitt 5.3.1 erläuterten Interfaces `IdentifierAsLocalVariableDeclaration`, `IdentifierAsMethodParameterDeclaration`, `IdentifierAsMemberDeclaration` und `IdentifierAsTypeDeclaration`.

Durch die erläuterten Techniken konnte eine vollständige Repräsentation des AST-Modells für MonoDevelop implementiert werden. Sie ist ausreichend, um die Faktenbasis für Refacola zu erzeugen. Da das AST-Modell jedoch nur für den lesenden Zugriff geeignet ist, musste noch eine Komponente implementiert werden, welche die in Abschnitt 5.3.2 eingeführten Refaktorisierungen durchführt. Sie wird im folgenden Kapitel näher erläutert.

5.4.2. RefactoringController

Der `RefactoringController` aus dem `Refactorings`-Namespace der `AST-Access`-Assembly nimmt die vereinbarten Datencontainer entgegen, welche die durchzuführenden Refaktorisierungen beschreiben (vgl. 5.3.2), und veranlasst die Modifikation des Quellcodes durch weitere Klassen. Abb. 5.8 veranschaulicht, welche Klassen an der Refaktorisierung beteiligt sind und wie diese zusammenhängen.

Zunächst unterscheidet der `RefactoringController`, ob es sich um die Änderung einer `Entity`, d.h. eines Typs oder Members, handelt oder ob eine Referenz geändert werden soll. Ist Letzteres gegeben, so wird die Refaktorisierung direkt von der dafür zuständigen Klasse ausgeführt. Da im Rahmen dieser Arbeit Bezeichner von Referenzen geändert werden können, existiert für diesen Zweck die `ChangeReferenceIdentifier`-Klasse. Diese verarbeitet `ChangeReferenceIdentifierRefactoringInput`-Instanzen und modifiziert den Bezeichner wie gewünscht direkt in der angegebenen Datei (siehe Abb. 5.8). Sollen hingegen Typen oder Member geändert werden, so gibt der `RefactoringController` die erhaltenen Datencontainer zunächst an Verteilerklassen (engl. dispatcher class) weiter, die abhängig vom zu ändernden Objekt die Arbeit weiter delegieren. Dabei steht für jeden `RefactoringInput`-Typ ein eigener Dispatcher zur Verfügung, der analog benannt ist. So verarbeitet die `ChangeEntityAccessModifierDispatcher`-Klasse beispielsweise Refaktorisierungsbeschreibungen vom Typ `ChangeEntityAccessModifierRefactoringInput`. Entsprechende Dispatcher stehen auch für `ChangeEntityIdentifier` und `ChangeEntityOwner` zur Verfügung. Da das AST-Modell Referenzen lediglich über die Datei und die

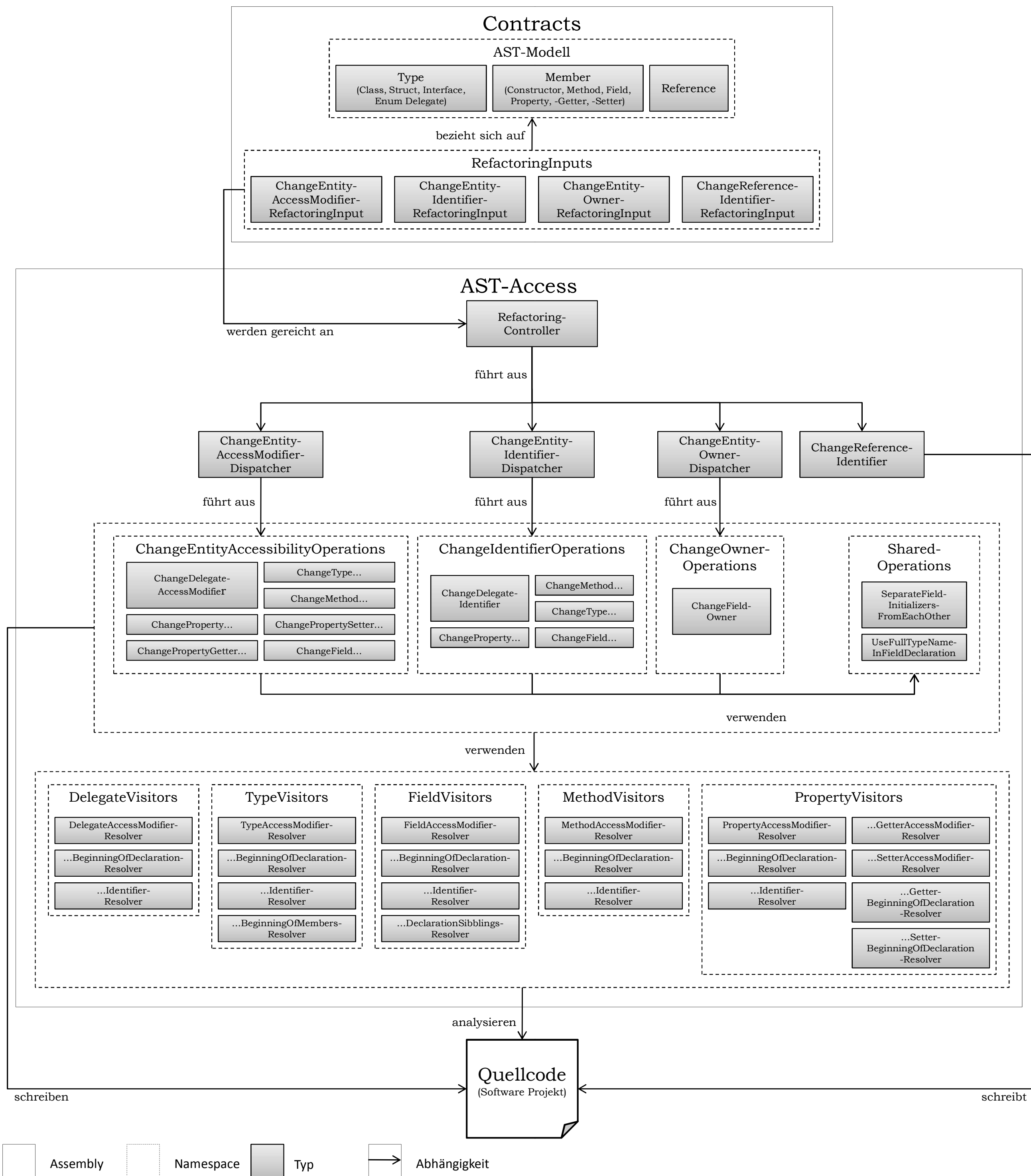


Abbildung 5.8.: Zusammenspiel der AST-Access-Komponenten, um Refaktorisierungen durchzuführen

genaue Positionsangabe innerhalb der Datei identifiziert und somit keine verschiedenen Referenz-Typen unterscheidet, entfällt hierfür die Notwendigkeit dieser Dispatcher und die Modifikation kann, wie oben beschrieben, direkt durchgeführt werden.
Im nächsten Schritt untersuchen die Dispatcher das Objekt, welches modifiziert werden soll. Abhängig von dem Objekt-Typ wird eine eigens dafür vorgesehene Klasse aus den `Operations`-Namespaces verwendet, um die geforderte Refaktorisierung durchzuführen. Entsprechend den verschiedenen möglichen Refaktorisierungen existieren die folgenden Namespaces:

- `ChangeEntityAccessibilityOperations`: In diesem Namespace liegen die Klassen, die die Logik enthalten, um den Zugriffsmodifizierer von Typen und Membern zu ändern. Klassen, Structs, Interfaces und Enums werden durch die `ChangeTypeAccessModifier`-Klasse abgedeckt. Delegates hingegen von einer eigens dafür vorgesehenen Klasse. Diese Unterscheidung liegt darin, dass Delegates gesondert behandelt werden müssen. Auf diese Besonderheit wird weiter unten eingegangen. Außerdem liegen nicht nur für Propertys, sondern außerdem für deren Getter und Setter eigene Klassen vor, da der Zugriffsmodifizierer für diese drei Einheiten unterschiedlich gesetzt werden kann.

- `ChangeEntityIdentifierOperations`: In diesem Namespace befinden sich Klassen, die den Bezeichner von Typen und Membern ändern.

- `ChangeEntityOwnerOperations`: Dieser Namespace enthält Klassen, die Member von einer Klasse in eine andere verschieben.

Neben diesen `Operations`-Klassen, welche die eigentliche Logik der Refaktorisierung für konkrete Objekt-Typen enthalten, existiert der Namespace `SharedOperations`. In diesem befinden sich ausgelagerte Operationen, die von mehreren Refaktorisierungen verwendet werden können. Im Rahmen dieser Arbeit konnten als gemeinsam verwendete Operationen `SeparateFieldInitializersFromEachOther` sowie `UseFullTypeNameInFieldDeclaration` ausgelagert werden. `SeparateFieldInitializersFromEachOther` ist notwendig, da es in C# zulässig ist Felder gemeinsam, unter einmaliger Angabe eines Zugriffsmodifizierers, zu deklarieren und initialisieren. Mit Hilfe dieser Operation werden diese Initialisierungen voneinander getrennt. Das ist beispielsweise nötig, wenn von einem einzigen Feld innerhalb dieser gemeinsamen Deklaration der Zugriffsmodifizierer geändert oder es in eine andere Klasse verschoben werden soll. `UseFullTypeNameInFieldDeclaration` ersetzt die Angabe des Feldtyps durch eine Bezeichnung, welche den Namespace, in dem sich der Typ befindet, vor den Typnamen stellt. Diese Operation wird benötigt, wenn ein Feld in eine andere Klasse verschoben wird. Da nicht davon ausgegangen werden kann, dass die `using`-Statements in der Datei der neuen Ownerklasse den Namespace des Feldtyps enthalten, wird der volle Typbezeichner verwendet. Dadurch wird sichergestellt, dass der Typ vom Compiler auf jeden Fall aufgelöst werden kann, solange die Assembly, in der sich der Typ befindet, referenziert wird.

Die Klassen aus den `Operations`-Namespaces müssen zur Durchführung der Refaktorisierung den Quellcode vorab analysieren und entsprechend der geforderten Modifikation direkt in den Dateien ändern. Zur Untersuchung des bestehenden Quellcodes bedienen sie sich der Analyseklassen aus den `Visitor`-Namespaces. Dabei steht für alle C#-Konstrukte, die Ziel einer Refaktorisierung sein können, ein eigener Namespace zur Verfügung. Hierbei wird in `Delegate-`, `Type-`, `Field-`, `Method-`, und `PropertyVisitors` unterschieden[2]. Die jeweiligen Klassen sind Visitor-Klassen der in Abschnitt 3.3 vorgestellten NRefactory-Komponente der MonoDevelop-API. Die Visitors folgen dem Single Responsibility Principle (vgl. [MSA09, S. 11]). Demnach haben Visitors die Verantwortung über nur einen Element-Typ des DOM und wurden entsprechend separat implementiert.
Damit die Visitors ihre Aufgabe durchführen können, ist es zunächst notwendig, die Datei, in der das Zielobjekt deklariert ist, zu parsen[3]. Die so erzeugte `CompilationUnit`-Instanz wird anschließend durch den jeweiligen Visitor analysiert. Wie sich aus der Abbildung ablesen lässt, wurden für Delegates, Types, Fields, Methods und Propertys die folgenden Visitors implementiert:

- `AccessModifierResolver`: Diese Klassen geben die genauen Positionen der Zugriffsmodifizierer in der Deklaration an[4]. Diese Resolver sind notwendig, wenn der Zugriffsmodifizierer eines Objekts im Zuge einer Refaktorisierung geändert werden soll.

- `BeginningOfDeclarationResolver`: Über diese Visitors erhält man die genaue Position innerhalb der Datei, bei der die Deklaration des Objekts beginnt. Auch das ist notwendig, wenn durch eine Refaktorisierung der Zugriffsmodifizierer geändert werden soll. Denn hierbei werden zuerst alle Zugriffsmodifizierer entfernt und im Anschluss der gewünschte neue Wert an den Anfang der Deklaration geschrieben.

- `IdentifierResolver`: Diese Visitors geben Auskunft über die genaue Position des Bezeichners innerhalb der Deklaration des Objekts. Sie kommen zum Einsatz, wenn der Bezeichner geändert werden soll.

Neben diesen - für alle Typen und Member vorhandenen - Visitors gibt es noch folgende spezielle Implementierungen:

[2]Visitors werden für Delegates und Typen getrennt voneinander implementiert, da das DOM, welches vom NRefactory-Parser erzeugt wird, die Deklaration von Klassen, Structs, Interfaces und Enums gemeinsam durch `TypeDeclaration`-Instanzen abbildet, wohingegen `DelegateDeclaration`-Instanzen für die Deklarationen von Delegates verwendet werden.

[3]Die Deklaration kann geparst werden, da jede Typ- und Member-Repräsentation des AST-Modells auch den Pfad zur Datei, in der es deklariert wurde, veröffentlicht (vgl. AST-Modell in Abschnitt 5.3.1).

[4]Sollte das Objekt als `protected internal` markiert sein, so wird sowohl für `protected` als auch für `internal` die Position zurückgeliefert.

- TypeBeginningOfMembersResolver: Dieser Visitor liefert den Beginn des Rumpfs (engl. Body) eines Typs. Dies ist wichtig, wenn Member durch die Refaktorisierung verschoben, und damit deren Deklaration in einen anderen Typ eingefügt werden sollen. Neue Member werden hierbei an der Stelle, die dieser Visitor liefert, eingefügt.

- FieldDeclarationSibblingsResolver: Dieser Visitor analysiert die Deklaration von einem Feld hinsichtlich weiterer Felder, die Teil der gleichen Deklaration sind. Die Analyseergebnisse werden in einer FieldDeclarationSibbilingsResult-Instanz beschrieben[5]. Die darin enthaltenen Informationen werden u.a. von der SeparateFieldInitializersFromEachOther-Operation benötigt. Denn um die Felddeklaration aufzuspalten werden für jedes Feld sowohl die Angabe des Typs, als auch die Modifizierer anhand der Positionsangaben kopiert. Anschließend wird die Felddeklaration, welche neben dem Feldnamen auch den Initialisierer beinhaltet, übernommen. Zum Abschluss wird die gesamte frühere Felddeklaration entfernt.

- PropertyGetter/SetterAccessModifierResolver: Diese Visitors liefern die Position des Zugriffsmodifizierers vom Getter bzw. Setter. Diese Angabe ist notwendig, wenn der Zugriffsmodifizierer gelöscht werden soll.

- PropertyGetter/SetterBeginningOfDeclarationResolver: Diese Visitors liefern den Anfang der Deklaration des Getters bzw. Setters. Soll der Zugriffsmodifizierer explizit auf einem Getter oder Setter gesetzt werden, so wird er an den Anfang dessen Deklaration gesetzt.

Sobald die Dateien mit Hilfe der Visitors analysiert wurden, übergeben die Operations-Klassen die durchzuführenden Änderungen an die Klasse RefactoringService im Namespace MonoDevelop.Refactoring, welche die Dateien schließlich modifiziert.
Damit wurden alle Klassen, die an einer Refaktorisierung beteiligt sind, einzeln und voneinander abgegrenzt erläutert. Abschließend soll noch anhand dem „Pull-Up-Field“-Refactoring aufgezeigt werden, wie die Klassen genau zusammenhängen.

- Die Refaktorisierung muss zunächst durch eine Instanz der Klasse ChangeEntityOwnerRefactoringInput beschrieben werden. Diese Instanz beschreibt das Feld sowie die Klasse, in die das Feld verschoben werden soll.

- Im nächsten Schritt wird dieser Datencontainer an den RefactoringController übergeben.

[5]Sie enthält zum einen die genaue Positionsangabe des Typs des Felds, der in der Deklaration angegeben werden muss. Zum anderen enthält sie alle Modifizierer, die Teil der Deklaration sind. Das schließt neben Zugriffsmodifizierern zusätzlich const und static mit ein. Außerdem enthält sie für jedes Feld die Positionsangabe für dessen Deklaration und Initialisierer.

- Anschließend erzeugt der RefactoringController eine Instanz vom Typ ChangeEntityOwnerDispatcher, übergibt ihr die Beschreibung und stößt die Refaktorisierung an.

- Die ChangeEntityOwnerDispatcher-Instanz analysiert daraufhin das Objekt, welches verschoben werden soll. Da es sich um ein Feld handelt, wird die Klasse ChangeFieldOwner instanziert und ihr das Feld sowie die Zielklasse übergeben.

- ChangeFieldOwner veranlasst zunächst die Klasse SeparateFieldInitializersFromEachOther dazu, die Felddeklaration - wenn möglich - aufzuspalten. Das bedeutet, dass alle Felder, die Teil der Deklaration sind, in eine separate Deklaration verschoben werden. Dazu bedient sich diese Klasse des FieldDeclarationSibblingsResolverVisitors, analysiert durch ihn die Deklaration des Feldes, und übergibt die notwendigen Änderungen an den RefactoringService.

- Anschließend veranlasst die ChangeFieldOwner-Instanz die Klasse UseFullTypeNameInFieldDeclaration dazu, den Typen des Felds vollständig, d.h. inklusive dessen Namespace, zu beschreiben. Diese analysiert daraufhin wieder mit Hilfe des FieldDeclarationSibblingsResolver die derzeitige Deklaration des Feldes und ersetzt schließlich den Typnamen.

- Im nächsten Schritt bedient sich auch ChangeFieldOwner am FieldDeclarationSibblingsResolver, um den gesamten Bereich der Felddeklaration zu erfragen und ihn anschließend zu entfernen.

- Anschließend erhält ChangeFieldOwner mit Hilfe des TypeBeginningOfMembersResolvers die Stelle in der neuen Owner-Klasse, an der Felder deklariert werden können, und fügt die Felddeklaration dort ein.

Mit dem RefactoringController ist nun auch der schreibende Zugriff auf den AST abgedeckt und damit die AST-Access-Assembly vollständig beschrieben.

5.5. Komponente Refacola-Interface

Die Refacola-Interface-Komponente stellt die Schnittstelle des Plugins zu Refacola bereit. In Abschnitt 5.1 wurde bereits erläutert, dass die Kommunikation zwischen Refacola und dem Plugin über zwei Dateien abläuft: Die Faktenbasis als Repräsentation des AST und das Change Set als Abbildung der geforderten AST-Modifikationen. Somit besteht die Aufgabe der Refacola-Interface-Assembly zum einen aus der Erzeugung der Faktenbasis und zum anderen aus der Verarbeitung der Change Sets.

Zur Lösung dieser Aufgabe gliedert sich die Komponente in folgende Bereiche auf:

- Implementierung der Refacola-C#-Sprachdefinition.
- Algorithmus zur Generierung der Faktenbasis.
- Export der Faktenbasis.
- Import der Change Sets.
- Abbildung der geforderten Modifikationen durch `RefactoringInput`-Instanzen.

Die folgenden Kapitel gehen auf die genannten Bereiche genauer ein.

5.5.1. Implementierung der Refacola-C#-Sprachdefinition

Die Refacola-Sprache sieht vor, dass definierte Kinds einer Sprachdefinition von mehreren anderen Kinds erben können. Sie unterstützt somit die Mehrfachvererbung. Sollen die Kinds unter Berücksichtigung der Vererbungshierachie durch C#-Typen abgebildet werden, so bietet es sich an Interfaces zu verwenden. Denn die C#-Sprachdefinition (vgl. [ECM06, S. 41-42]) legt fest, dass Klassen und Interfaces von mehreren Interfaces erben können, wohingegen die Mehrfachvererbung unter Klassen verboten ist. Aus diesem Grund wird im Namespace `LanguageModel.Kinds.Interface` jeder Kind der Refacola-Sprachdefinition durch ein Interface abgebildet. Die Refacola-Propertys der Typen werden durch entsprechende C#-Propertys abgebildet. Desweiteren können den Refacola-Propertys Wertebereiche zugewiesen werden. Beschränkt sich der Wertebereich auf einen bestimmten Kind, so entspricht der Typ des C#-Propertys dem Interface, welches diesen Kind abbildet. Definiert der Wertebereich jedoch eine Menge von Konstanten, so wird die Domäne durch ein Enum im Namespace `LanguageModel.Domains` repräsentiert. Das Property `Identifier` wird standardmäßig durch einen `string` abgebildet.
Darüber hinaus sieht die Implementierung der Sprachdefinition vor, dass jedes C#-Interface, welches einen Kind repräsentiert, vom `IKind`-Interface erben muss. Dieses Interface definiert zwei `string`-Propertys: `KindName` und `Id`. Diese Propertys entsprechen keinem Refacola-Property aus der Sprachdefinition, sondern werden für den späteren Export benötigt. `KindName` liefert den Namen des Kinds zurück und `Id` ist eine eindeutige Bezeichnung des instanzierten Kinds, also des Programmelements.
Querys, die Kinds zueinander in Beziehung setzen, können nicht voneinander erben. Aus diesem Grund wurden für die verschiedenen Arten von Querys auch keine Interfaces eingeführt. Stattdessen werden Querys direkt durch konkrete Klassen abgebildet, wobei jeder Parameter der Querys durch ein Property in der Klassendeklaration dargestellt wird. Desweiteren müssen alle Querys das `IQuery`-Interface implementieren, welches lediglich das

`string`-Property `QueryName` festlegt. Die Query-Implementierungen liefern bei diesem Property den Namen des Querys zurück.
Diese theoretischen Ausführungen sollen anhand dem Klassendiagramm in Abb. 5.9 verdeutlicht werden, welches die Klassenhierarchie der C#-Implementierung des Entitys `Member` darstellt.
Wie zu erkennen ist, erbt das `IMember`-Interface analog zur Sprachdefinition von `IMemberOrConstructor` und `INamedEntity`. `INamedEntity` erbt direkt von `IEntity`. `IMemberOrConstructor` leitet von `IAccessibleEntity`, `IDeclaredEntity` und `IOwnedEntity` ab, welche alle wieder von `IEntity` erben. `IEntity` implementiert das `IKind`-Interface.

Damit ist die gesamte Sprachdefinition durch C#-Typen abgebildet. Die Interfaces, die Kinds repräsentieren, können jedoch nicht instanziert werden. Deshalb ist es noch für jedes Kind notwendig, aus dem im Rahmen der Faktengenerierung Programmelemente erzeugt werden könnten, eine Klasse zu deklarieren. Diese Klassen implementieren dabei lediglich das Interface des Kinds. Konkret handelt es sich dabei um folgende Kinds:

- `TopLevelClass`, `-Interface`, `-Struct`, `-Enum` und `-Delegate` sowie `NestedClass`, `-Interface`, `-Struct`, `-Enum` und `-Delegate` zur Abbildung der C#-Typen.
- `Namespace`, um Namespaces darzustellen.
- `Constructor`, `Field`, `InstanceMethod` und `Property` zur Darstellung aller Member.
- `PropertyGetter` und `PropertySetter` um die Getter und Setter der Propertys abzubilden.

Da Querys durch Klassen repräsentiert werden, können durch deren Instanzierung Fakten bereits problemlos erzeugt werden. Damit liegen nun ausreichend Klassen vor, um eine vollständige Faktenbasis generieren zu können. Abschließend ist noch zu erwähnen, dass die Typen, die zur Abbildung der Sprachdefinition implementiert wurden, von der Refacola-Infrastruktur generiert werden sollten. Da zum Zeitpunkt der Erstellung dieser Arbeit diese Infrastruktur noch nicht vorlag, wurden die Typen prototypisch von Hand entwickelt. Die automatische Generierung wird als Teil einer weiteren Arbeit implementiert [Her11].

5.5.2. Algorithmus zur Generierung der Faktenbasis

Die Faktengenerierung wird gemeinsam von drei Klassen aus dem Namespace `FactsGeneration` der `Refacola-Interface`-Assembly durchgeführt. Abb. 5.10 veranschaulicht deren Zusammenspiel.

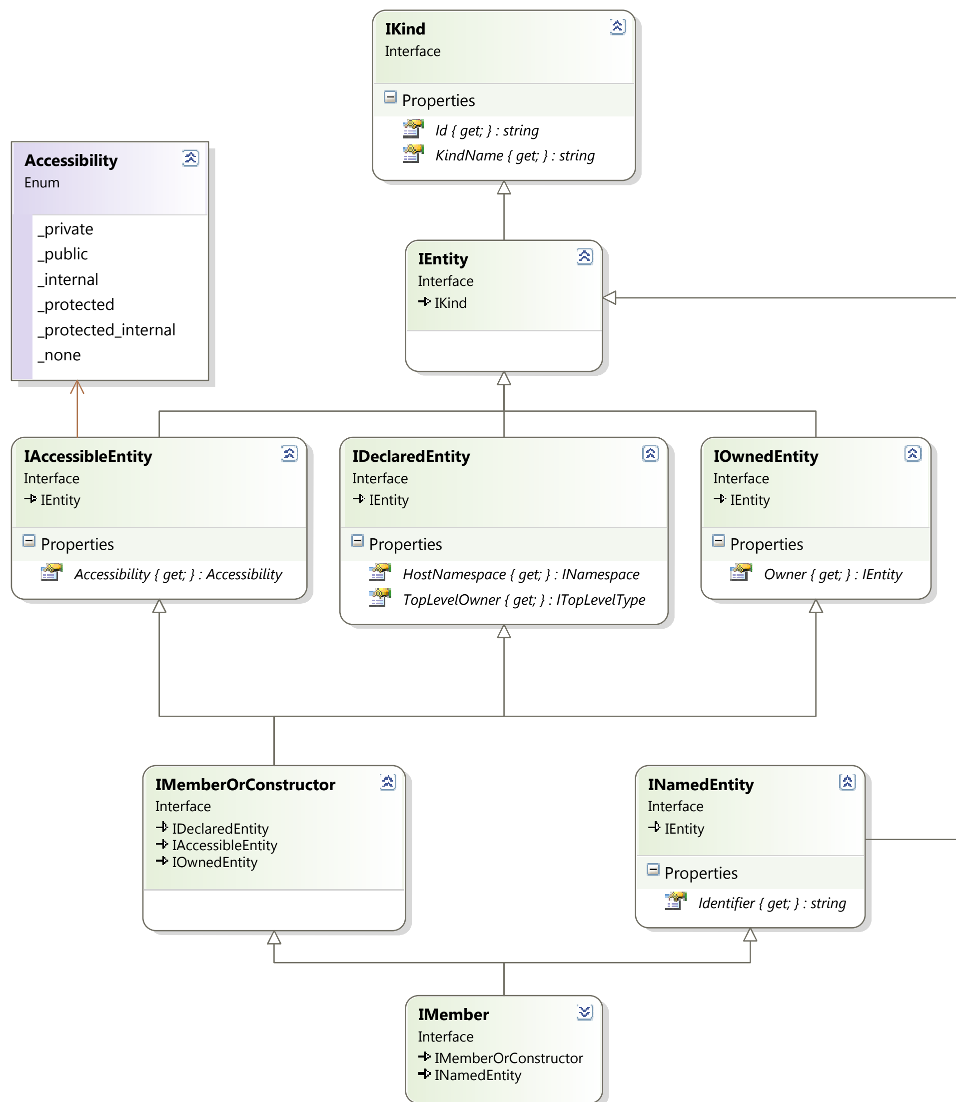


Abbildung 5.9.: Implementierung der Member-Entität der Sprachdefinition

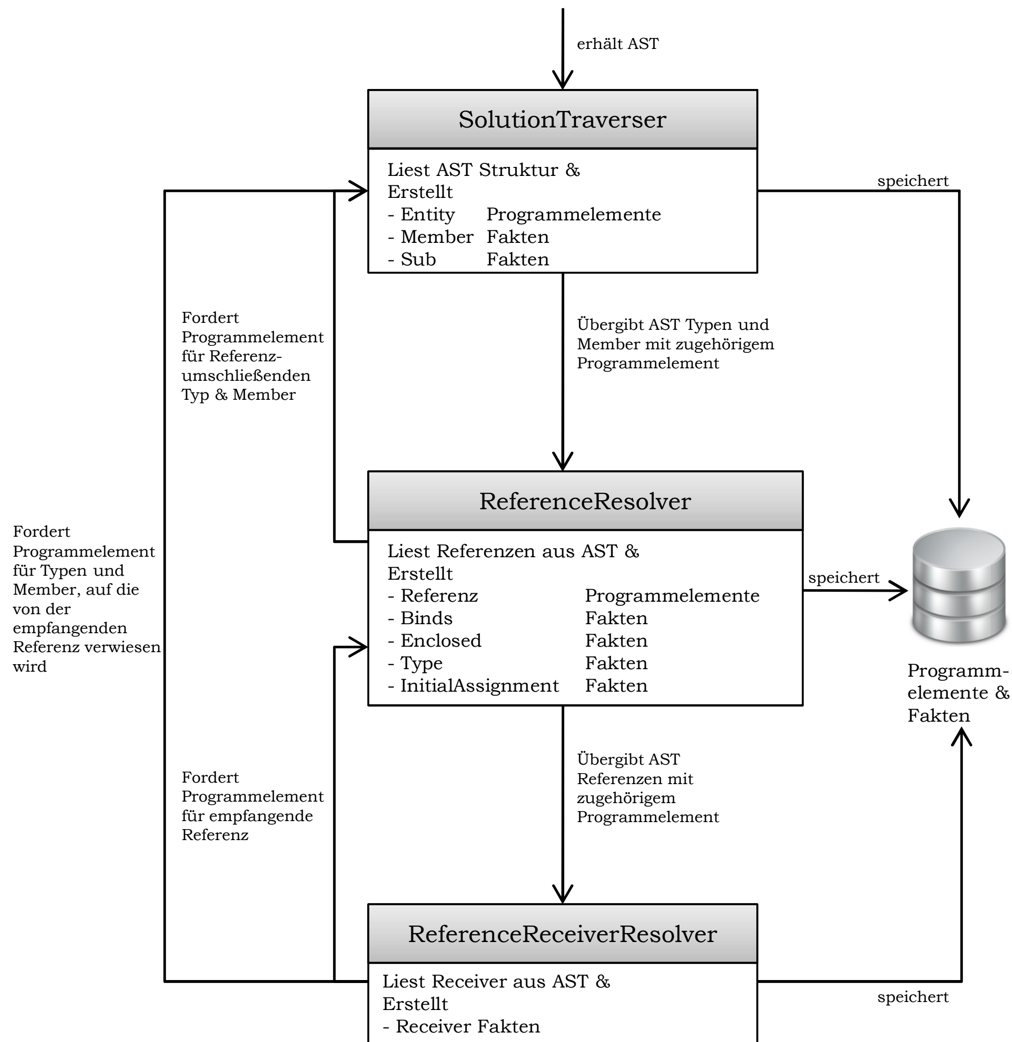


Abbildung 5.10.: Zusammenspiel der Klassen zur Faktengenerierung

Alle Klassen arbeiten dabei auf dem vereinbarten AST-Modell. Die generierten Programmelemente und Fakten werden nicht unmittelbar in eine Datei geschrieben, sondern vorerst im Speicher, und zwar in einer Instanz der Klasse `Factsbase` aus dem Namespace `FactsGeneration`, gehalten. Die Faktengenerierung kann von anderen Assemblys über die öffentliche statische Klasse `FactbaseGenerator` und dessen Methode `GenerateFactsFromSolution` aus demselben Namespace angestoßen werden.

SolutionTraverser

Soll die Faktenbasis erzeugt werden, so wird der `SolutionTraverser` über dessen Methode `TraverseSolutionAndGenerateFacts` vom `FactbaseGenerator` aufgefordert, die übergebene Solution zu analysieren und diese durch Programmelemente und Fakten abzubilden. Der `SolutionTraverser` verfolgt bei der Analyse einen „Top-Down“-Ansatz. Das bedeutet, dass er den gesamten AST, ausgehend von der Solution als Wurzelknoten, bis zu den Membern der Typen als Blattknoten, abarbeitet und für diese Knoten die entsprechenden Programmelemente aus den Kinds-Klassen erzeugt. Dabei werden auch beliebig tief verschachtelte Typen analysiert. Für Member muss neben dem Programmelement, das sie repräsentiert, auch ein Programmelement für den Typ ihres Rückgabewertes erzeugt werden. Dieser Typ kann jedoch in einer externen, referenzierten Assembly - und damit außerhalb von der Solution - deklariert worden sein. Um in diesem Fall dennoch eine vollständige Faktenbasis generieren zu können, ist es notwendig, dessen Deklarationshierarchie ebenfalls abzubilden. Dazu wird ausgehend von diesem Typ, über dessen eventuell vorhandene Elterntypen, bis schließlich zum umschließenden Namespace, für jeden angetroffenen Knoten ein Programmelement erzeugt. Durch die Auflösung externer Typen nach diesem „Bottom-Up“-Ansatz wird eine minimale und gleichzeitig ausreichende Faktenbasis erzeugt, da nur die Typen aus referenzierten Assemblys durch Programmelemente abgebildet werden, die auch tatsächlich verwendet werden. Alle anderen Typen werden nicht beachtet und sind für eine vollständige Abbildung des AST auch irrelevant[6].
Die beschriebenen „Top-Down“- und „Bottom-Up“-Ansätze sollen im Folgenden veranschaulicht werden. In Listing 5.5 ist die Deklaration der Klasse `MyClass` abgebildet, deren Methode `MyMethod` eine Instanz vom Typ `string`, der im Namespace `System` der .NET-Assembly `mscorlib` definiert ist, zurückliefert.

[6] Als Alternative zu diesem „Bottom-Up“-Ansatz könnten alle referenzierten Assemblys in ihrer Gesamtheit nach dem „Top-Down“-Ansatz abgebildet werden. Dieses Vorgehen würde jedoch länger dauern und mehr Daten generieren, als zur Abbildung des AST notwendig sind, wenn nicht alle Typen der referenzierten Assemblys innerhalb der Solution verwendet werden.

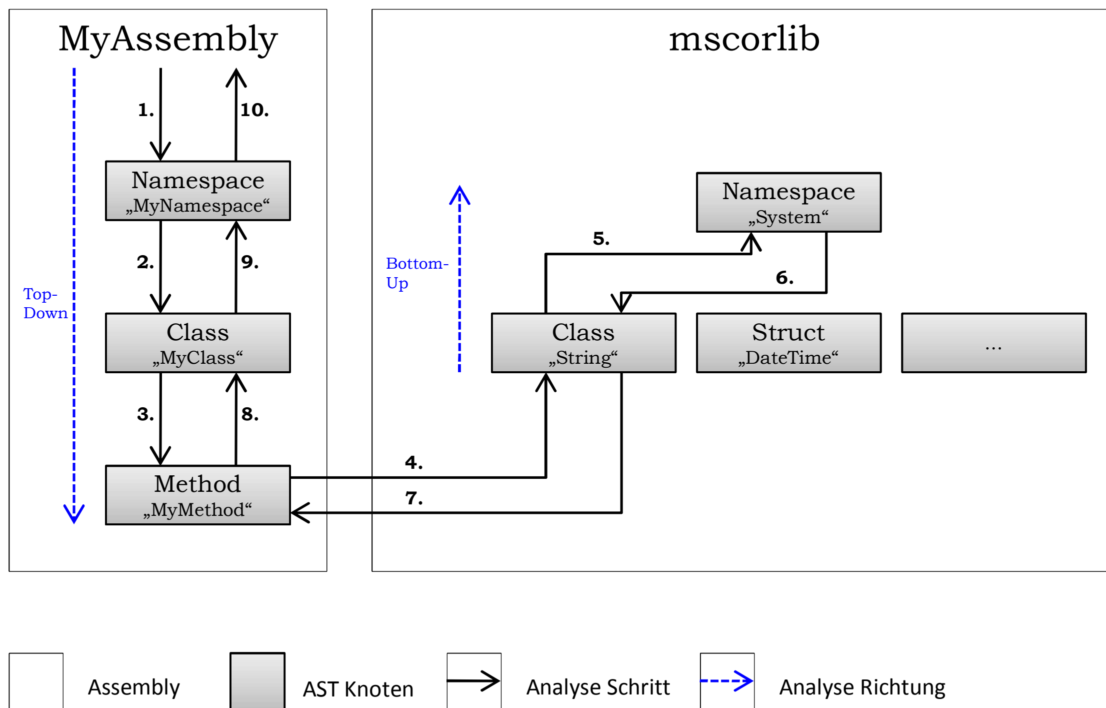


Abbildung 5.11.: Analyse Schritte des SolutionTraversers

Listing 5.5: C#-Code zur Demonstration des SolutionTraversers

```
using System;


namespace MyNamespace
{
    public class MyClass
    {
            public string MyMethod()
            {
                return "Text";
            }
    }
}
```

Abb. 5.11 illustriert die Schritte, die der `SolutionTraverser` durchläuft, um die Programmelemente zu erzeugen.

Diese seien nun genauer beleuchtet:

1. Der `SolutionTraverser` arbeitet die Solution nach dem „Top-Down“-Ansatz ab und trifft auf den Namespace `MyNamespace`.

2. Zur Abbildung des gefunden Namespaces wird ein Programmelement vom Kind `Namespace` erzeugt. Anschließend werden alle darin deklarierten Typen analysiert.

3. Es wird ein Programmelement vom Typ `TopLevelClass` erzeugt, um die gefundene Klasse `MyClass` abzubilden. Anschließend werden die darin enthaltenen Member durchsucht, wobei `MyMethod` gefunden wird.

4. Der Rückgabetyp der Methode muss analysiert werden, da dessen Programmelement zur Abbildung der Methode benötigt wird. Die Analyse geschieht nach dem „Bottom-Up“-Ansatz.

5. Zur Abbildung der Klasse `string` muss erst dessen umschließender Namespace verarbeitet werden.

6. Der Namespace `System` wird durch ein Programmelement vom Typ `Namespace` abgebildet.

7. Damit kann nun auch ein Programmelement zur Repräsentation der Klasse `string` vom Typ `TopLevelClass` erzeugt werden. Somit ist die Analyse dieser Klasse abgeschlossen, da keine weiteren Details, wie zum Beispiel dessen Member, relevant sind.

8. Mit dem Programmelement der Klasse `string` kann schließlich das Programmelement vom Typ `InstanceMethod` zur Abbildung der Methode erzeugt werden.

9. Die Analyse der Klasse `MyClass` ist abgeschlossen, da keine weiteren Member existieren.

10. Auch die Analyse des Namespaces `MyNamespace` ist beendet, da alle darin enthaltenen Klassen abgebildet wurden.

Dabei wird deutlich, dass bei der „Bottom-Up“-Analyse bestimmter Knoten keine Details dieser Knoten verarbeitet werden, sondern nur die Informationen aus dem AST gelesen werden, die notwendig sind, um das Programmelement zu erzeugen. So erfolgt bei obigem Beispiel keine Analyse des Structs `DateTime`, da dieses im Rahmen der Solution nicht verwendet wird.
Außer den Programmelementen, die sich aus dem AST ergeben, erzeugt der `SolutionTraverser` noch Fakten der beiden Query-Typen `Member` und `Sub`. `Member`-Fakten werden für

alle analysierten Member erzeugt und stellen deren Programmelement in Relation zu dem Programmelement des umschließenden Typs. `Sub`-Fakten werden für alle Vererbungen zwischen zwei Typen erstellt und setzen ihre Programmelemente zueinander in Verbindung. Neben der erläuterten aktiven Faktengenerierung, die direkt vom `SolutionTraverser` angestoßen wird, bietet diese Klasse noch zwei öffentliche Methoden, die vom `ReferenceResolver` und `ReferenceReceiverResolver` in Anspruch genommen werden, um für AST-Knoten die Programmelemente passiv zu erzeugen:

- `GetTypeKindAndTraverseToRoot`: Diese Methode erwartet einen Knoten aus dem AST, der einen Typ abbildet. Dieser wird gemäß dem oben erläuterten „Bottom-Up"-Ansatz analysiert, um ein Programmelement zu erzeugen. Dieses Programmelement wird an den Aufrufenden zurückgegeben.

- `GetMemberKindAndTraverseToRoot`: Über diese Methode wird für den übergebenen Member-Knoten nach dem „Bottom-Up"-Ansatz ein Programmelement erzeugt und zurückgeliefert.

Die Programmelemente und Fakten, die vom `SolutionTraverser` erzeugt werden, sind noch nicht ausreichend, um den gesamten AST abzubilden, da Referenzen bisher ausgeblendet wurden. Um diese Lücke zu schließen reicht der `SolutionTraverser` Typen und Member samt deren Programmelement an den `ReferenceResolver` weiter. Dabei werden jedoch nur Typen und Member berücksichtigt, die innerhalb der Solution deklariert wurden und damit im Rahmen der „Top-Down"-Analyse verarbeitet werden.
Abschließend ist die beschriebene Logik im Anhang unter E nochmal als stark vereinfachter Pseudo-Code dargestellt.

ReferenceResolver

Über die öffentlichen Methoden

- `void ResolveReferencesOnTypeAndBuildQueries (ITypeContainer, IType)`

- `void ResolveReferencesOnFieldAndBuildQueries (IFieldContainer, IField)`

- `void ResolveReferencesOnMethodAndBuildQueries (IMethodContainer, IInstanceMethod)`

- `void ResolveReferencesOnPropertyGetterAndBuildQueries (IPropertyGetterContainer, IPropertyGetter)`

- `void ResolveReferencesOnPropertySetterAndBuildQueries (IPropertySetterContainer, IPropertySetter)`

erhält der `ReferenceResolver` vom `SolutionTraverser` Typen und Member des AST sowie deren abbildende Programmelemente. Die Aufgabe besteht nun darin, für die übergebenen Elemente alle Referenzen in der Solution ausfindig zu machen, und für diese - wenn möglich - Fakten aus folgenden Querys zu erzeugen:

- `Binds`: Fakten von diesem Query setzen `Reference`- und `Entity`-Programmelemente zueinander in Verbindung und drücken damit aus, auf welche konkrete Entität ein erzeugtes `Reference`-Programmelement verweist.

- `Enclosed`: Dieses Query setzt den `Reference`- und `MemberOrConstructor`-Kind in Relation zueinander. Erzeugte Fakten zeigen an, in welchem Member sich eine konkrete Referenz befindet.

- `Type`: Fakten von diesem Query halten fest, von welchem Typ eine bestimmte Referenz ist. Dazu werden `TypedEntityReference`- und `Type`-Programmelemente zueinander in Verbindung gesetzt. Referenzen vom Kind `TypedEntityReference` beziehen sich auf Entitäten, die entweder selbst ein Typ sind (zum Beispiel Klassen oder Interfaces), einen Typ zurückgeben (Methoden und Propertys) oder Instanzen eines Typs sind (Felder). Das `Type`-Programmelement bildet dann entweder den Typ selbst, den zurückgegebenen Typ oder aber den Typ der Instanz ab.

- `InitialAssignment`: Dieses Query setzt `Field`- und `TypedEntityReference`-Kinds zueinander in Verbindung. Die damit erzeugten Fakten weisen darauf hin, dass sich die angegebene Referenz innerhalb des Feldinitialisierers des Feldes befindet.

Um diese Fakten generieren zu können, ist es zunächst notwendig, alle Referenzen auf den übergebenen Typ oder Member aufzulösen. Das in Abschnitt 5.3.1 vorgestellte AST-Modell schreibt vor, dass die Typ- und Member-Instanzen unmittelbar Zugriff auf deren Referenzen gewähren. Für alle erhaltenen Referenzen muss zunächst ein Programmelement erzeugt werden. Die Sprachdefinition sieht dafür die folgenden Referenz-Kinds vor:

- `TypeReference`: Referenzen auf alle Arten von Typen.

- `PropertySetterReference`: Referenzen auf PropertySetter.

- `PropertyGetterReference`: Referenzen auf PropertyGetter.

- `MethodReference`: Referenzen auf Methoden.

- `FieldReference`: Referenzen auf Felder.

Gemäß der Sprachdefinition erben diese Referenz-Kinds die Propertys `Owner`, `TopLevelOwner`, `HostNameSpace` und `InferredType`. Das Property `Owner` zeigt an, innerhalb welchem Typ sich die Referenz befindet. Die Propertys `TopLevelOwner` und `HostNameSpace` ergeben sich automatisch mit dem Programmelement des umschließenden Typs, da dieses die gleichen Propertys abbildet und die Werte identisch sein müssen. Über `InferredType` wird angegeben, von welchem Typ die Entität ist, auf welche die Referenz verweist. Folglich muss zur Abbildung der erhaltenen Referenzen lediglich das Programmelement des umschließenden Typs sowie das Programmelement des Typs der Entität, auf den die Referenz verweist, aufgelöst werden. Letzteres liegt bereits vor, da im Fall von Referenzen auf Typen, der `InferredType` identisch mit dem Programmelement des Typs selbst ist. Im anderen Fall von Referenzen auf Member, erhält man den `InferredType` über das Property `DeclaredType` vom übergebenen Programmelement des Members. Der Typ, innerhalb dessen Deklaration sich die Referenz befindet, muss hingegen erst aufgelöst werden. Über die Methode `RetrieveOwnerType`, die das AST-Modell für Referenzen definiert, ist genau das möglich. Um das Programmelement des gelieferten Typs zu erhalten, bedient sich der `ReferenceResolver` der öffentlichen Methode `GetTypeKindAndTraverseToRoot` des `SolutionTraversers`, die im vorangegangenen Abschnitt erläutert wurde.
Mit diesen Daten ist der `ReferenceResolver` nun in der Lage, die Referenzen durch Programmelemente abzubilden. Im nächsten Schritt wird versucht, für die abgebildeten Referenzen Fakten zu erzeugen:

- `Binds`-Fakten werden für jede Referenz erzeugt. Das notwendige Programmelement der Referenz liegt durch die vorangegangenen Schritte vor. Das Programmelement der referenzierten Entität wurde dem `ReferenceResolver` mit der Aufforderung übergeben, darauf verweisende Referenzen aufzulösen, und liegt damit ebenfalls vor.

- `Enclosed`-Fakten werden für alle Referenzen erzeugt, die sich innerhalb eines Members befinden. Das AST-Modell sieht für Referenzen die Methode `RetrieveEnclosingMember` (vgl. D) vor, um den umschließenden Member aufzulösen. Über die öffentliche Methode `GetMemberKindAndTraverseToRoot` des `SolutionTraversers` wird das Programmelement zur Abbildung des Members angefordert und anschließend der Fakt erzeugt.

- Fakten des `Type`-Querys können ohne weiteren Aufwand erzeugt werden, da sie lediglich die Referenzen zum Typ der Entität, auf den die Referenzen verweisen, zueinander in Verbindung setzen. Letzteres liegt durch die beschriebene Auflösung des `InferredType`-Propertys bereits vor.

- `InitialAssignment`-Fakten werden erzeugt, wenn sich die Referenz innerhalb eines Feldinitialisierers befindet. Über die `RetrieveEnclosingMember`-Methode lässt sich, wie schon im Rahmen des `Enclosed`-Querys beschrieben, das umschließende Member anfordern. Wenn diese Methode ein Feld zurückliefert, befindet sich die Referenz in dessen Initialisierer und es wird ein `InitialAssignment`-Fakt erstellt.

Durch die beschriebenen Aufgaben ergänzt der `ReferenceResolver` den `SolutionTraverser` um die Analyse der Referenzen. Zur vollständigen Repräsentation des AST fehlt nun nur noch die Erzeugung der `Receiver`-Fakten. Dazu reicht der `ReferenceResolver` alle Referenzen auf Typen und Member samt dem dazu gehörenden Programmelement an den `ReferenceReceiverResolver` weiter. Außerdem stellt er noch folgende öffentliche Methoden zur Verfügung, um gefundene `Receiver`-Referenzen durch Programmelemente abzubilden:

- `IThisReference ResolveThisReferenceAndBuildQueries (IEntityReferenceContainer, isExplicit)`
- `ITypeReference ResolveReferenceOnTypeAndBuildQueries (IType, IEntityReferenceContainer)`
- `IFieldReference ResolveReferenceOnFieldAndBuildQueries (IField, IEntityReferenceContainer)`
- `IMethodReference ResolveReferenceOnMethodAndBuildQueries (IInstanceMethod, IEntityReferenceContainer)`
- `IPropertyGetterReference ResolveReferenceOnPropertyGetter-AndBuildQueries(IPropertyGetter, IEntityReferenceContainer)`
- `IPropertySetterReference ResolveReferenceOnPropertySetter-AndBuildQueries(IPropertySetter, IEntityReferenceContainer)`

Abschließend ist die implementierte Logik des ReferenceResolvers im Anhang unter F als Pseudo-Code dargestellt.

ReferenceReceiverResolver

Der `ReferenceReceiverResolver` kann über folgende öffentliche Methode dazu aufgefordert werden, den Receiver einer Member-Referenz aufzulösen und in der Faktenbasis einzutragen:

- `void ResolveReceiverAndBuildQueryForReference (IMemberReference, IEntityReferenceContainer)`

Dieser Methode wird sowohl die Referenz als auch dessen abbildendes Programmelement übergeben. In Abschnitt 5.3.1 wurde bereits erläutert, dass das AST-Modell vorsieht, dass über Referenz-Instanzen der Receiver unmittelbar aufgelöst werden kann. Zu

diesem Zweck bieten Referenzen die Methode `RetrieveReferenceReceiver` (vgl. Klassendiagramm unter D). Es ist nun Aufgabe des `ReferenceReceiverResolvers`, diesen zurückgegebenen Receiver zu analysieren und die Analyseergebnisse in der Faktenbasis zu erfassen. Das AST-Modell unterscheidet vier verschiedene Receiver-Typen, die unterschiedlich behandelt werden müssen:

- `ThisReferenceReceiver`: Handelt es sich bei dem Receiver der Referenz um eine `this`-Referenz, so muss für diese zunächst ein Programmelement angelegt werden. Dazu wendet sich der `ReferenceReceiverResolver` an die Methode `ResolveThisReferenceAndBuildQueries` vom `ReferenceResolver`[7]. Das erzeugte Programmelement wird anschließend gemeinsam mit dem Programmelement der Member-Referenz in einem `Receiver`-Fakt abgebildet.

- `CastReferenceReceiver`: Wenn der Empfänger der Member-Referenz ein Cast-Ausdruck ist, so wird die Angabe des Typs, auf den „gecastet“ wird, als Receiver interpretiert. Der Bezeichner des Typs innerhalb des Casts stellt dabei eine Referenz auf diesen Typ dar. Das entsprechende Programmelement wird über die Methode `ResolveReferenceOnTypeAndBuildQueries` des `ReferenceResolvers` erzeugt. Mit diesem Programmelement kann anschließend der `Receiver`-Fakt erstellt werden.

- `MemberReferenceReceiver`: Handelt es sich bei dem Empfänger des Memberzugriffs um einen Methodenaufruf, ein abgefragtes Feld oder Property, so wird dieser über `MemberReferenceReceiver`-Instanzen abgebildet. `MemberReferenceReceiver` enthalten die Referenz auf das empfangende Member. Das entsprechende Programmelement wird wieder über den `ReferenceResolver` erzeugt[8]. Mit dem zurückgelieferten Programmelement kann der `Receiver`-Fakt erzeugt werden.

- `IdentifierReferenceReceiver`: Ist der Empfänger des Members nur ein einzelner Bezeichner, so muss dieser zunächst weiter analysiert werden. Dafür kann über die `IdentifierReferenceReceiver`-Instanz auf dessen `IdentifierReference`-Property zugegriffen werden. Diese `IdentifierReference` bietet die Methode `ResolveDeclaration`, um die Deklaration des Bezeichners aufzulösen. Das AST-Modell definiert vier mögliche Typen von Bezeichnern, die jeweils unterschiedlich behandelt

[7] Diese Methode erwartet neben der Referenz noch eine Angabe darüber, ob es sich um eine explizite oder implizite `this`-Referenz handelt. Diese Unterscheidung ist notwendig, weil die Sprachdefinition für beide Varianten unterschiedliche Kinds vorsieht (`ImplicitThisReference` und `ExplicitThisReference`).

[8] Dazu muss zunächst untersucht werden, ob es sich bei dem Member um ein Feld, eine Methode oder einen PropertyGetter handelt (PropertySetter können keine Objekte zurückgeben und kommen deshalb nicht als Receiver in Frage). Abhängig vom Membertyp wird eine der Methoden `ResolveReferenceOnFieldAndBuildQueries`, `ResolveReferenceOnMethodAndBuildQueries`, und `ResolveReferenceOnPropertyGetterAndBuildQueries` aufgerufen.

werden müssen (vgl. zur näheren Erläuterung der Typen das AST-Modell im Abschnitt 5.3.1):

- `IIdentifierAsLocalVariableDeclaration`: Handelt es sich um eine lokale Variable, so wird der Bezeichner des Variablentyps aus der Variablendeklaration, welcher eine Referenz auf eben diesen Typen darstellt, als Receiver interpretiert und damit der `Receiver`-Fakt erstellt.

- `IIdentifierAsMethodParameterDeclaration`: Für den Fall, dass sich der Bezeichner auf ein Methodenargument bezieht, verläuft die Verarbeitung analog zur `IIdentifierAsLocalVariableDeclaration`: Der Typbezeichner innerhalb des Methodenkopfs wird als Referenz auf diesen Typen wahrgenommen. Das Programmelement, welches diese Referenz abbildet, wird schließlich als Receiver im `Receiver`-Fakt verwendet.

- `IIdentifierAsTypeDeclaration`: Handelt es sich bei dem Receiver um eine Klasse[9], so wird der Bezeichner durch eine `IIdentifierAsTypeDeclaration`-Instanz abgebildet. In diesem Fall ist der Bezeichner selbst die Referenz auf die Klasse und wird für den `Receiver`-Fakt verwendet.

- `IIdentifierAsMemberDeclaration`: Referenziert der Bezeichner jedoch einen lokalen Member, so sieht das AST-Modell vor, dass der Bezeichner über eine `IIdentifierAsMemberDeclaration`-Instanz charakterisiert wird. In diesem Fall geht der `ReferenceReceiverResolver` identisch vor, wie im Falle eines `MemberReferenceReceivers` (siehe oben).

Die hier skizzierte Logik ist in Abschnitt G nochmal im Gesamten als Pseudo-Code veranschaulicht.
Damit sind die wichtigsten Punkte des `ReferenceReceiverResolvers` erläutert. Die vorgestellten Klassen sind damit gemeinsam in der Lage, eine vollständige Faktenbasis zu erzeugen. Der nächste Abschnitt befasst sich mit dem Export dieser Daten in eine Datei, damit diese von Refacola eingelesen und verarbeitet werden können.

5.5.3. Export der Faktenbasis

Wie im vorangegangen Kapitel erläutert, werden die Programmelemente und Fakten zunächst nur im Speicher, in einer Instanz der Klasse `Factsbase`, gehalten. Da der Austausch von Daten mit Refacola jedoch über das Dateisystem stattfindet, muss die Faktenbasis in eine Datei geschrieben werden.

[9] Wie es beim Aufruf statischer Member der Fall ist.

In Abschnitt 5.1 wurde bereits auf den Aufbau der Datei eingegangen. Aus diesem Grund soll hier nur noch die technische Umsetzung aufgezeigt werden. Abb. 5.12 veranschaulicht das Zusammenspiel der einzelnen Klassen.

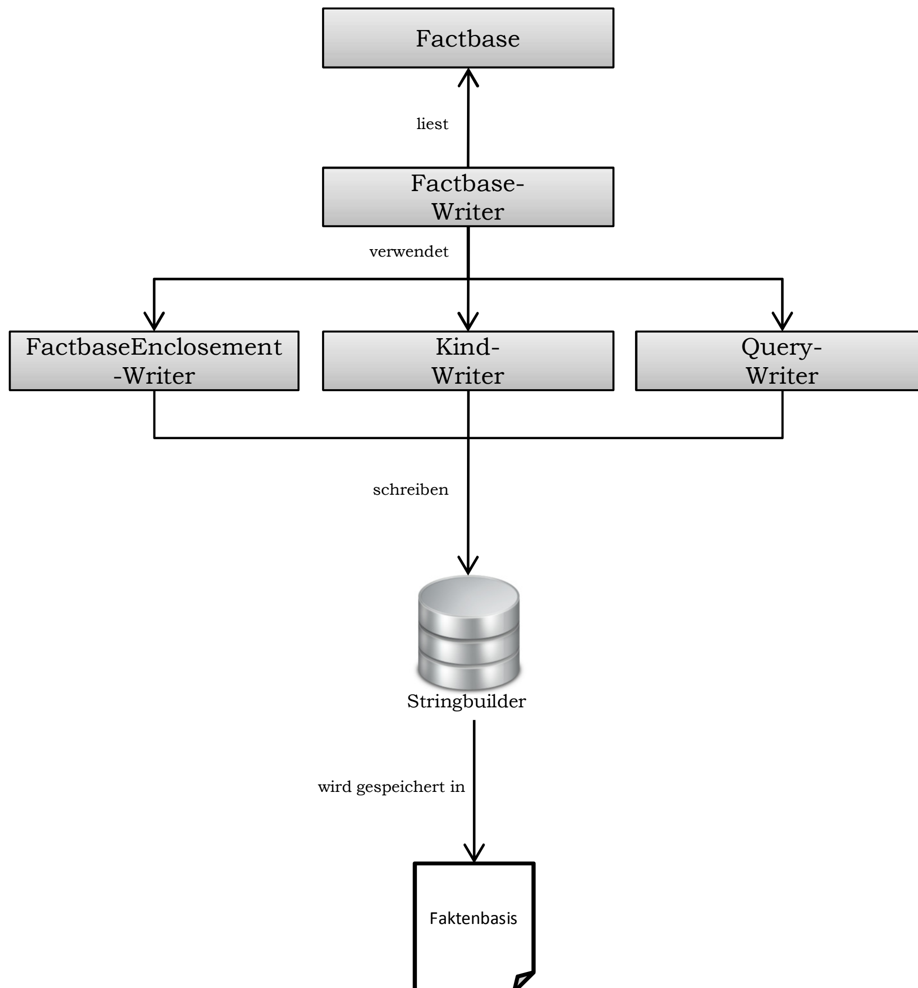


Abbildung 5.12.: Export der Faktenbasis

Im ersten Schritt wird die Klasse `FactbaseWriter` aus dem Namespace `IO.Factbase` dazu aufgefordert, alle Programmelemente und Fakten aus einer `Factbase`-Instanz als Text in einer `Stringbuilder`-Instanz abzulegen. Der `FactbaseWriter` bedient sich folgender Helfer-Klassen, um den Text zu erzeugen:

- `FactbaseEnclosementWriter`: Diese Klasse erzeugt den umschließenden Bereich einer Faktenbasis. Dieser besteht aus dem Namen der Faktenbasis und umklammert die erzeugten Programmelemente und Fakten.

- `KindWriter`: Mit Hilfe dieser Klasse werden Programmelemente geschrieben.

- `QueryWriter`: Mit Hilfe des `QueryWriters` werden Fakten (die schließlich Instanzen der Query-Klassen sind) in Textform übertragen.

Jedes Kind unterscheidet sich nun jedoch hinsichtlich seiner Propertys sowie deren Wertebereiche. Ebenso unterscheiden sich die Querys hinsichtlich der Argumente. Bei der Umsetzung war nun das Ziel, dass der `KindWriter` und `QueryWriter` keine konkreten Kind- bzw. Query-Klassen kennen muss. Stattdessen geben diese Klassen selbst alle Informationen bekannt, die benötigt werden, um die Übersetzung in Textform durchzuführen. Zu diesem Zweck wurden C#-Attribute eingeführt, welche den Propertys der Kind- und Query-Klassen zugewiesen werden, um die notwendigen Informationen festzuhalten. Listing 5.6 zeigt dies am Beispiel der Implementierung des Kinds `InstanceMethod` sowie des Querys `Binds`.

Listing 5.6: Implementierung des InstanceMethod Kinds sowie des Binds Querys

```
1 public class InstanceMethod : IInstanceMethod
2 {
3     public string KindName
4     {
5         get { return "InstanceMethod"; }
6     }
7
8     public string Id { get; set; }
9
10    [KindProperty]
11    public INamespace HostNamespace { get; set; }
12
13    [KindProperty]
14    public ITopLevelType TopLevelOwner { get; set; }
15
16    [KindProperty(typeof(AccessibilityToStringConverter))]
17    public Accessibility Accessibility { get; set; }
18
19    [KindProperty]
20    public IEntity Owner { get; set; }
21
22    [KindProperty(true)]
23    public string Identifier { get; set; }
24
25    [KindProperty]
26    public IType DeclaredType { get; set; }
27 }
28
29 public class Binds : IQuery
30 {
31     public string QueryName
32     {
33         get { return "binds"; }
34     }
35
36     [QueryProperty(0)]
37     public IReference Reference { get; set; }
38
39     [QueryProperty(1)]
40     public IEntity Entity { get; set; }
41 }
```

Alle C#-Eigenschaften, die das Attribut `KindProperty` tragen, werden vom `Factbase-Writer` als Propertys des Kinds interpretiert. Das `KindProperty`-Attribut definiere folgende Eigenschaften:

- `string PropertyName`: Wenn diese Eigenschaft gesetzt ist, wird der angegebene Name beim Schreiben des Programmelements für das Property verwendet.
- `bool IsStringProperty`: Diese Eigenschaft sagt aus, ob der Wert des Propertys in Anführungszeichen gesetzt werden soll. Im Listing 5.6 ist dies beim Property `Identifier` in Zeile 22 der Fall.
- `Type TypeConverterType`: Über diese Eigenschaft kann ein `TypeConverter` angegeben werden, der für die Umwandlung des Propertywerts in Textform zuständig ist. In Zeile 16 übernimmt die Klasse `AccessibilityToStringConverter` die Umwandlung der `Accessibility` in Textform.

Wenn der Rückgabetyp einer Eigenschaft ein Kind darstellt (und somit das Interface `IKind` implementiert), wird beim Schreiben des Propertywerts die Id der `IKind`-Instanz verwendet. Das ist in Listing 5.6 für die Eigenschaften `HostNamespace`, `TopLevelOwner`, `Owner` und `DeclaredType` der Fall. Darüber hinaus müssen beim Schreiben der Programmelemente der Name des Kinds sowie die eindeutige Id des Programmelements bekannt sein. Beides erhält der Writer über die Eigenschaften `KindName` und `Id` des Interfaces `IKind`.
Für Querys existiert analog das C#-Attribut `QueryProperty`. Über dessen Eigenschaft `QueryPropertyIndex` vom Typ `Integer` wird definiert, in welcher Reihenfolge die Query-Argumente beim Schreiben der Query-Instanz beachtet werden sollen. Im angegebenen Beispiel wird das `Reference`-Argument aus Zeile 37 folglich vor dem `Entity`-Argument aus Zeile 40 geschrieben. Beim Schreiben der Query-Argumente wird davon ausgegangen, dass alle Argumente vom Interface `IKind` ableiten. Entsprechend wird der Wert deren `Id`-Eigenschaft als Wert des Arguments verwendet. Über die Eigenschaft `QueryName` des Interfaces `IQuery` erhält der Writer den Namen, der für den Fakt verwendet werden soll. Der so entstandene Text, der vorerst noch in einer `Stringbuilder`-Instanz liegt, kann anschließend in eine Datei geschrieben werden. Das liegt allerdings in der Verantwortung des Aufrufenden des `FactbaseWriters`.

5.5.4. Import der Change Sets

Sobald die Faktenbasis generiert wurde, ist es Aufgabe der Refacola-Infrastruktur, diese zu verarbeiten und die gewünschten Änderungen am AST in einem Change Set abzulegen. Das Format der Change Sets wurde bereits in Abschnitt 5.1 erläutert. Aus diesem Grund wird im Folgenden nur auf den in Abb. 5.13 dargestellten Importvorgang eingegangen.

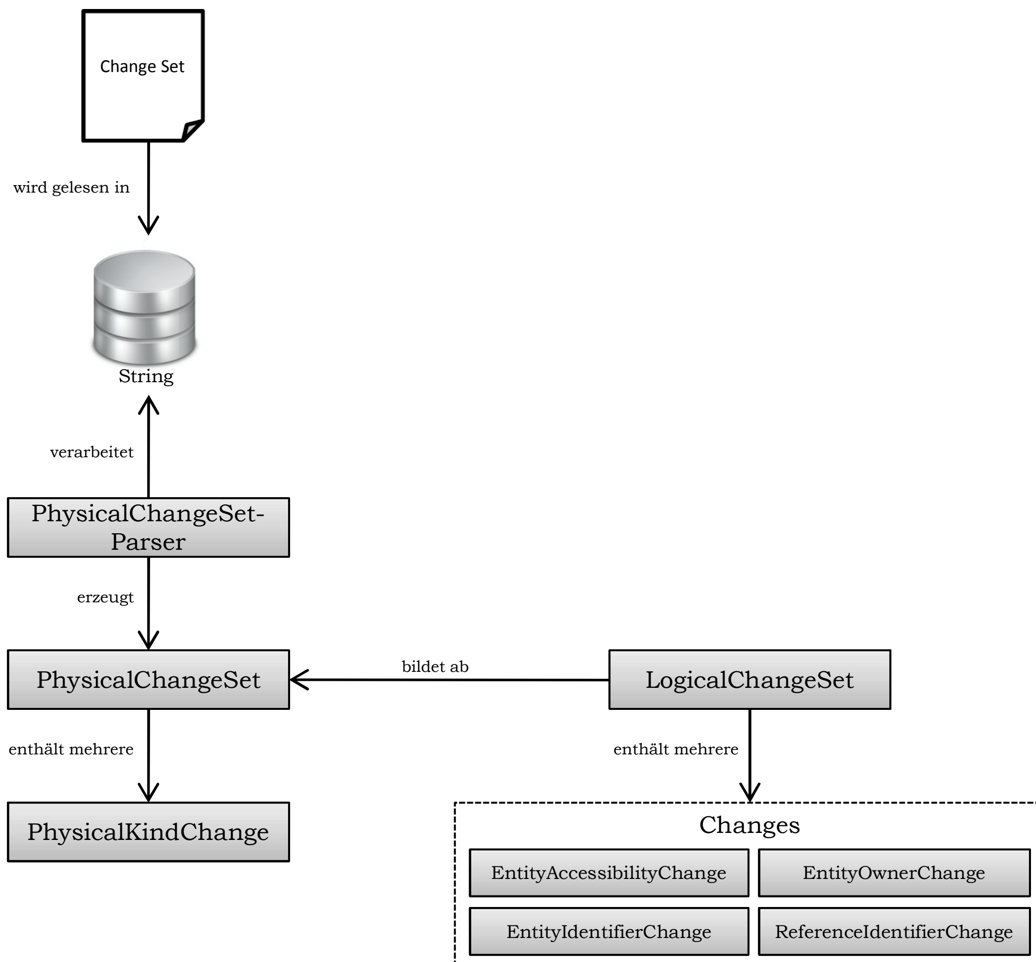


Abbildung 5.13.: Import der Change Sets

Im ersten Schritt ist es notwendig, den Text aus der Datei auszulesen.
Im zweiten Schritt wird dieser Text zur Verarbeitung an den `PhysicalChangeSetParser` aus dem Namespace `IO.Changeset.Physical` übergeben. Dieser verarbeitet jede einzelne Zeile des Change Sets. Die Changes aus dem Change Set sind identisch aufgebaut. Auf die Id des Programmelements folgt das Property, welches geändert werden soll, sowie dessen neuer Wert. Der `PhysicalChangeSetParser` versucht diese drei Teile aus jeder Zeile auszulesen und bildet sie als strings in einer Instanz vom Typ `PhysicalKindChange` ab. Nachdem alle Zeilen verarbeitet wurden liegt das Ergebnis in einer Instanz vom Typ `PhysicalChangeSet` vor.
Im dritten Schritt wird die erzeugte `PhysicalChangeSet`-Instanz an die statische Methode `CreateFromPhysicalChangeSet` der Klasse `LogicalChangeSet` übergeben. Diese Methode erwartet als weiteres Argument die Faktenbasis (also eine Instanz der `Factbase`-Klasse), um die Programmelemente, die geändert werden sollen, auflösen zu können. Für jede `PhysicalKindChange`-Instanz wird anschließend überprüft, ob das Programmelement in der übergebenen Faktenbasis wiedergefunden werden kann. Wenn dies der Fall ist, wird als nächstes das zu ändernde Property sowie dessen neuer Wert analysiert. Abhängig von dem Analyseergebnis wird eine der folgenden Klassen zur Abbildung der gewünschten Änderung verwendet:

- `EntityAccessibilityChange`: Wenn sich das zu ändernde Programmelement in der Faktenbasis wiederfinden lässt, es vom Typ `Entity` ist und `Accessibility` das zu ändernde Property ist sowie der neue Wert aus der `AccessModifier`-Domäne kommt, dann wird die Änderung mit Hilfe dieser Klasse abgebildet.

- `EntityOwnerChange`: Heißt das zu ändernde Property dagegen `Owner`, so wird zuerst überprüft, ob der neue Propertywert als Programmelement innerhalb der Faktenbasis existiert. Wenn dies der Fall ist, so wird das Programmelement, dessen Owner zu ändern ist, sowie das Programmelement des neuen Owners in einer Instanz der Klasse `EntityOwnerChange` abgebildet.

- `EntityIdentifierChange`: Soll dagegen das Property `Identifier` geändert werden, und bezieht sich die Änderung auf ein Programmelement vom Typ `Entity`, so wird die `PhysicalKindChange`-Instanz mit Hilfe dieser Klasse abgebildet.

- `ReferenceIdentifierChange`: Soll zwar das Property `Identifier` geändert werden, handelt es sich bei dem Programmelement allerdings um eine Referenz, so wird die Änderung mit dieser Klasse abgebildet. Diese Klasse enthält neben dem neuen Bezeichner und dem Programmelement der Referenz noch das Programmelement der Entität, auf die sich die Referenz bezieht.

Als Ergebnis liefert die `CreateFromPhysicalChangeSet`-Methode schließlich ein `LogicalChangeSet` zurück, welches alle erstellten `Change`-Instanzen enthält.

5.5.5. Abbildung der geforderten Modifikationen durch RefactoringInputs

Wie in Abschnitt 5.4.2 erläutert, kann der AST über den `RefactoringController` der `AST-Access`-Assembly modifiziert werden. Da dieser jedoch `RefactoringInput`-Instanzen erwartet, ist es noch notwendig, die Changes aus dem `LogicalChangeSet` umzuwandeln. Diese Aufgabe übernimmt die `RefactoringInputFactory`-Klasse aus dem Namespace `Refactoring`. Deren statische Methode `GenerateRefactoringInputsFromLogicalChangeSet` erwartet die `LogicalChangeSet`-Instanz sowie die Solution, auf die sich die Änderungen beziehen. Die Solution wird dabei wieder als Instanz des AST-Modells übergeben und dient dazu, die Entitys und Referenzen aus dem `LogicalChangeSet` in die entsprechenden Knoten des AST aufzulösen. Als Ergebnis liefert die Methode eine Liste von `RefactoringInput`-Instanzen. Im Folgenden ist die Auflösung der verschiedenen `Change`-Klassen durch die `RefactoringInputFactory` näher beschrieben:

- `EntityAccessibilityChange`: Diese Änderung wird durch die Klasse `ChangeEntityAccessModifierRefactoringInput` abgebildet. Da die `RefactoringInput`-Klasse auf einem Typ oder Member aus dem AST-Modell basiert, muss das Programmelement aus der `EntityAccessibilityChange`-Instanz zunächst in den abgebildeten Knoten des AST aufgelöst werden. Diese Auflösung geschieht mittels des „Fully Decorated Name" von Typen und Membern, da dieser eindeutig in der gesamten Solution ist und sich auch aus Programmelementen herleiten lässt. Dieser vollständige Name beinhaltet neben dem Namen des Typs oder Members noch die Namen aller umschließenden Typen sowie den Namen des Namespaces. Zur Demonstration dient die Methode `SomeMethod` aus Listing 5.7.

  Listing 5.7: Demonstration des Fully Decorated Name

  ```
  namespace MyNamespace.SubNamespace
  {
      class MyOuterClass
      {
          class MyInnerClass
          {
              void SomeMethod() { }
          }
      }
  }
  ```

  Der „Fully Decorated Name" dieser Methode lautet „MyNamespace.SubNamespace.MyOuterClass.MyInnerClass.SomeMethod" und muss einmalig innerhalb der Solution sein, wenn der Quellcode kompiliert werden kann.
  Möchte man diesen vollständigen Namen aus Programmelementen herleiten, so ist es lediglich notwendig, über das Property `Owner` der Entität alle umschließenden En-

titäten rekursiv zu durchlaufen, bis schließlich der `TopLevelType` erreicht ist. Von diesem ist dann noch der Name des Namespaces hinzuzunehmen. Die `Solution`-Instanz veröffentlicht gemäß dem AST-Modell die Methode `TryGetEntityFromFullyDecoratedName`, welche versucht, einen Knoten über dessen vollständigen Namen ausfindig zu machen. Kann die Entität aufgelöst werden, so ist noch der neue Zugriffsmodifizierer in den Wertebereich aufzulösen, den das AST-Modell für Zugriffsmodifizierer vorsieht. Diese Umwandlung ist nötig, weil der `AccessModifier`-Wertebereich der Refacola-Sprachdefinition durch ein eigenes, unabhängiges, Enum abgebildet wird. Da beide Wertebereiche die gleichen Werte beinhalten ist die Umwandlung unproblematisch. Wurden die Entität sowie der neue Zugriffsmodifizierer erfolgreich aufgelöst, so kann die `RefactoringInput`-Instanz erzeugt werden.

- `EntityOwnerChange`: Diese Änderung wird durch Instanzen der Klasse `ChangeEntityOwnerRefactoringInput` abgebildet. Sie erwartet neben dem Member, das verschoben wird, noch den neuen umschließenden Typ. Sowohl das Member, als auch der neue Typ werden wieder als Instanzen aus dem AST-Modell abgebildet. Dazu müssen lediglich die Programmelemente, die das Member und den neuen Owner repräsentieren, aus der `EntityOwnerChange`-Instanz über ihren vollständigen Namen nach dem bereits erläuterten Vorgehen aufgelöst werden. Wenn dies gelingt, kann die `RefactoringInput`-Instanz erzeugt werden.

- `EntityIdentifierChange`: Die Klasse `ChangeEntityIdentifierRefactoringInput` bildet diese Änderung ab. Zur Umwandlung ist es lediglich notwendig, das Programmelement der Entität in den entsprechenden AST-Knoten aufzulösen. Wenn das erfolgreich war, kann der neue Bezeichner ohne Weiteres übernommen werden, da er vom Typ `string` ist.

- `ReferenceIdentifierChange`: Diese Refaktorisierung wird durch `ChangeReferenceIdentifierRefactoringInput`-Instanzen abgebildet. Um die Referenz im AST wiederzufinden, ist es nicht möglich einen „Fully Decorated Name“ zu verwenden, da so etwas für Referenzen nicht existiert. Aus diesem Grund wurde folgender Weg gewählt: Wenn die Faktenbasis generiert wird und dabei Referenzen durch Programmelemente abgebildet werden, so bekommen diese neben den üblichen Propertys des Referenz-Typs noch das Property `LocationInformation` vom Typ `string`. Der Wert dieses Propertys ergibt sich wiederum aus der Referenz, die abgebildet werden soll. Das AST-Modell sieht dazu vor, dass jede Referenz ein `string`-Property implementiert, das ebenfalls `LocationInformation` heißt (vgl. Klassendiagramm unter D). Dieser Wert ist einmalig für jede Referenz und enthält neben dem Namen der Datei, innerhalb der sich die Referenz befindet, auch die Zeile und Spalte sowie den Bezeichner der Referenz[10].

[10] Man hätte alternativ einen Globally unique identifier (GUID) generieren können, jedoch ist die gewählte Art der Id verständlicher, da sich so direkt feststellen lässt, wo die Referenz liegt.

Da `ReferenceIdentifierChange`-Instanzen nicht nur das Programmelement der Referenz enthalten, sondern auch das Programmelement der Entität, auf die sich die Referenz bezieht, kann über die Entität zunächst der entsprechende AST-Knoten über dessen vollständigen Name aufgelöst werden. Anschließend wird jede Referenz auf diese Entität überprüft. Sobald eine Referenz gefunden wurde, bei der das `LocationInformation`-Property identisch mit dem Wert ist, den das Referenz-Programmelement liefert, konnte die Referenz erfolgreich aufgelöst werden. Mit der aufgelösten Referenz und dem neuen Bezeichner kann schließlich die `ChangeReferenceIdentifierRefactoringInput`-Instanz erstellt werden.

Die zurückgelieferten `RefactoringInput`-Instanzen können anschließend zur Weiterverarbeitung an den `RefactoringController` übergeben werden.

5.6. Komponente MonoDevelop-Extension

Das Modul `MonoDevelop-Extension` ist das Herzstück des Plugins und hat die Aufgabe, ein Interface für MonoDevelop zu implementieren, damit es als Plugin erkannt und geladen wird. Desweiteren registriert es verschiedene Kommandos im Menü der IDE, nimmt deren Aufruf entgegen und delegiert diese an die anderen vorgestellten Komponenten weiter. Es werden folgende Kommandos registriert:

- `Generate Factbase`: Wird diese Aktion über das Menü der MonoDevelop-IDE ausgeführt, so fordert die `MonoDevelop-Extension` über den `MonoDevelopAstProvider` der `AST-Access`-Assembly zunächst den Zugriff auf die geladene Solution an. Anschließend wird die Solution an den `FactbaseGenerator` der `Refacola-Interface`-Assembly gereicht. Dieser erstellt daraufhin - zunächst noch im Speicher - mit Hilfe des bereits beschriebenen `SolutionTraversers` die Faktenbasis und liefert sie an den Aufrufer zurück.

- `Export Factbase`: Soll die erzeugte Faktenbasis exportiert werden, so ist es Aufgabe der `MonoDevelop-Extension`, dem Benutzer einen Dialog zur Auswahl des Dateipfades anzubieten. Anschließend wird die erläuterte `FactbaseWriter`-Klasse der `Refacola-Interface`-Assembly angewiesen, die erzeugte Faktenbasis in Textform zu übersetzen. Ist dies erfolgt, wird der Text in die ausgewählte Datei geschrieben.

- `Apply Change Set`: Zunächst bietet die `MonoDevelop-Extension` bei Auswahl dieser Aktion einen Dialog zur Selektion des Change Sets an. Anschließend wird mit Hilfe der erläuterten Klassen aus der `Refacola-Interface`-Assembly eine `LogicalChangeSet`-Instanz erzeugt, welche über die `RefactoringInputFactory` in mehrere `RefactoringInput`-Instanzen übersetzt wird.

Schließlich werden die erzeugten Instanzen an den `RefactoringController` der `AST-Access` Komponente übergeben, um die Änderungen am Quellcode durchzuführen.

Mit den Erläuterungen zur `MonoDevelop-Extension` Komponente sind alle Kernbestandteile des Plugins beschrieben.

6. Probleme während der Entwicklungsphase

In den folgenden Abschnitten werden ausgewählte Probleme, die sich während der Implementierung des Plugins ergeben haben und gelöst werden mussten, erläutert.

6.1. Auflösen von ReturnTypes

Die Implementierung des AST-Modells für MonoDevelop stützt sich, wie bereits erläutert, auf das ProjectDom. Zugriff auf `ProjectDom`-Instanzen erhält man vom `ProjectDomService` aus dem Namespace `MonoDevelop.Projects.Dom.Parser`[1]. Innerhalb des ProjectDoms werden Rückgabetypen von Methoden, Feldern und Propertys durch `ReturnType`-Instanzen beschrieben, welche lediglich den Namen des Typs beschreiben. Da das im Rahmen dieser Arbeit entworfene AST-Modell jedoch immer konkrete Typen referenziert, müssen diese `ReturnType`-Instanzen in konkrete Typen aufgelöst werden. Die verschiedenen MonoDevelop-APIs sehen dafür mehrere Möglichkeiten vor. Jedoch hat sich nur eine als zuverlässig herausgestellt:
Die Datei, in welcher der `ReturnType` verwendet wird[2], muss über die Methode `ParseFile` des `ProjectDomService` geparst werden. Diese liefert eine `CompilationUnit`-Instanz, die gemeinsam mit der `ReturnType`-Instanz an die Methode `SearchType` der `ProjectDom`-Klasse übergeben wird. Diese Methode löst anhand der `using` Instruktionen sowie der referenzierten Assemblys des Projekts, den Typen auf.
Mit diesem Vorgehen haben sich jedoch weitere Probleme ergeben. Vor deren Erläuterung ist zunächst zu verstehen, dass `ProjectDom` eine abstrakte Klasse ist. Die tatsächliche Implementierung ist eine Instanz vom Typ `DatabaseProjectDom`, die sich zur Aufbewahrung von geparsten Typen auf die Klasse `SerializationCodeCompletionDatabase` aus dem Namespace `MonoDevelop.Projects.Dom.Serialization` stützt. Diese Klasse behält geparste Typen im Speicher und persistiert diese mit dem Überschreiten einer Maximalanzahl in einer Datei. Verwendet man nun die `DatabaseProjectDom`-Klasse, um Typen aufzulösen, so ergeben sich die folgenden Probleme:

[1] Hierbei ist wieder die in Abschnitt 3.1 erläuterte Unterscheidung zwischen der `ProjectDom`-Klasse und dem ProjectDom AST-Modell zu beachten.
[2] Es handelt sich somit um die Datei, in der das Member deklariert ist.

- Typen, die innerhalb der Solution deklariert wurden, können nur aufgelöst werden, wenn die Datei, in der sich die Typdeklaration befindet, bereits geparst wurde. Das liegt daran, dass sich die `DatabaseProjectDom`-Klasse an die `SerializationCodeCompletionDatabase`-Klasse mit der Bezeichnung des aufzulösenden Typs wendet. Diese wiederum überprüft sowohl die im Speicher gehaltenen Typdeklarationen, als auch die Typdeklarationen, die bereits ausgelagert wurden. Das Problem besteht nun darin, dass MonoDevelop das Parsen aller Dateien der Solution in einem Hintergrundthread durchführt. Versucht man nun einen Typen aufzulösen, dessen Deklaration noch nicht geparst wurde und auch nicht in früheren Sitzungen ausgelagert wurde, so schlägt die Auflösung fehl. Entsprechend bricht die Faktengenerierung mit einer `Exception` ab, weil keine vollständige AST-Repräsentation erzeugt werden kann.
  Gelöst wurde das Problem dadurch, dass alle Dateien der Solution geparst werden, sobald die Faktengenerierung angestoßen wird. Die Analyse des AST startet erst, sobald die Solution vollständig abgearbeitet wurde.

- Das nächste Problem besteht darin, dass die `SerializationCodeCompletionDatabase`-Klasse geparste Typen auslagert, sobald sich eine gewisse maximale Anzahl von Typdeklarationen im Speicher befindet. Diese Maximalzahl ist standardmäßig auf `10` festgelegt. Probleme können nun auftauchen, wenn Typdefinitionen angefragt werden, die in eine Datei ausgelagert wurden und im Zuge der Anfrage wieder eingelesen werden. Die Ursache liegt darin, dass die Implementierung der Klasse, die geladene Typen abbildet[3], fehlerhaft ist. Besonders problematisch ist dies, da die Fehler schwer zu reproduzieren sind. Denn wenn sich ein geladener Typ im Speicher befindet und nicht ausgelagert wurde, treten diese Fehler nicht auf[4]. Die einzige Möglichkeit die `SerializationCodeCompletionDatabase`-Klasse dazu zu zwingen, keine Auslagerungen zur Laufzeit vorzunehmen, besteht darin, die Maximalzahl der im Speicher gehaltenenen Typdefinitionen auf einen ausreichend großen Wert zu setzen. Im Rahmen dieser Arbeit wurde er auf `100.000` gesetzt. Der Nachteil besteht nun darin, dass bei einer Solution, die mehr als `100.000` Typen umfasst, wieder Probleme auftauchen können. Desweiteren kann sich MonoDevelop mit einer `OutOfMemoryException` beenden, wenn der Speicher nicht ausreicht, um alle Typen darin zu behalten.

- Ein weiteres Problem ergibt sich, wenn Quellcode-Dateien vom Parser, der daraus die `CompilationUnit`-Instanz erzeugt, nicht fehlerfrei verarbeitet werden können. Das ist insbesondere dann problematisch, wenn der Quellcode korrekt ist und damit vom C#-Compiler kompiliert werden kann. Können nun Dateien nicht vollständig

[3] `DomTypeProxy` aus `MonoDevelop.Projects.Dom.Serialization`.
[4] Das Entwicklerteam von MonoDevelop wurde im Rahmen dieser Arbeit auf gefundene Fehler hingewiesen. Trotz der Fehlerbehebung haben die Entwickler davon abgeraten, sich auf die `DomTypeProxy`-Instanzen und auf den Inhalt der Auslagerungsdatei im Gesamten zu verlassen, da bei der Serialisierung und Deserialisierung Probleme auftauchen können.

analysiert werden, in denen Typen deklariert werden, die an anderer Stelle in der Solution verwendet werden, so kann dieser Typ später nicht zuverlässig aufgelöst werden. Dies hat zur Folge, dass die Erstellung der Faktenbasis fehlschlägt[5].

6.2. ProjectDom-Database

Im vorangegangenen Abschnitt wurde bereits erläutert, dass eine `SerializationCodeCompletionDatabase`-Instanz die geladenen Typinformationen auslagert, sobald eine gewisse maximale Anzahl von Typdeklarationen im Speicher gehalten wird. Darüber hinaus findet die Auslagerung jedes Mal statt, wenn MonoDevelop ordnungsgemäß beendet wird. Wird die Solution später wieder geladen, werden die Typinformationen zunächst aus der zuvor ausgelagerten Datei eingelesen. Im Regelfall sorgt das bereits erläuterte initiale Parsen aller Quellcode-Dateien dafür, dass alle von der Datei eingelesenen Typinformationen durch die so neu erzeugten Typinformationen ersetzt werden. Allerdings ist es während der Entwicklung des Plugins vereinzelt dazu gekommen, dass MonoDevelop die alten Typinformationen nicht ersetzt hat. Wenn in der Zwischenzeit der Quellcode geändert wurde, führt das dazu, dass das ProjectDom, auf dem die AST-Modell-Implementierung und damit die gesamte Faktengenerierung basiert, einen alten Quellcode-Zustand repräsentiert. Dieser Umstand ist auf eine fehlerhafte Implementierung im ProjectDom von MonoDevelop zurückzuführen[6]. Das beschriebene Problem kann verhindert werden, indem die Datei, in welche die Typinformationen ausgelagert werden, von Hand gelöscht wird, bevor das Projekt in MonoDevelop geladen wird[7]. So wird - gemeinsam mit den im vorangegangenen Abschnitt erläuterten Modifikationen - dafür gesorgt, dass überhaupt keine Typinformationen mehr aus einer Auslagerungsdatei zur Anwendung kommen können.

6.3. FindMemberAstVisitor

Wie bereits bekannt ist, stützt sich die MonoDevelop-Implementierung des AST-Modells zum Auflösen von Referenzen auf Typen und Member auf die `FindMemberAstVisitor`-

[5] Es wurden im Rahmen dieser Arbeit Fehler am Parser behoben und an das Entwicklerteam weitergegeben. Dennoch war es im zeitlichen Rahmen dieser Arbeit nicht möglich, alle Fehler des Parsers zu beheben. Deshalb ist es wahrscheinlich, dass für verschiedene Solutions keine Faktenbasis generiert werden kann.

[6] Aufgrund des nicht zuverlässig reproduzierbaren Verhaltens konnte der Fehler nicht behoben und an das Entwicklerteam weitergegeben werden.

[7] Die Auslagerungsdatei befindet sich für jedes Projekt in dessen Hauptverzeichnis und endet auf „.pidb“.

Klasse. Die Standardimplementierung wies jedoch mehrere Probleme auf. Ein Problem bestand darin, dass bei der Suche nach Referenzen auf eine bestimmte Methode die Signatur der Methode ignoriert wurde. Das hatte zur Folge, dass die ursprüngliche Implementierung des Visitors jede Referenz auf Überladungen der gesuchten Methode ebenfalls zurücklieferte. Dieses Problem konnte behoben werden.
Ein weiteres Problem ist die Geschwindigkeit des `FindMemberAstVisitors`. Durch die Einführung verschiedener Abbruchkriterien konnte diese gesteigert werden[8].

6.4. Receiver im Fall von lokalen Variablen

In Abschnitt 5.5.2 wurde darauf eingegangen, dass der Receiver von Member-Referenzen eine lokale Variable sein kann. In diesem Fall wird die Referenz auf den Typ der Variable, die sich in der Deklaration der Variable befindet, als empfangende Referenz verwendet. Im Rahmen dieser Arbeit sind dabei folgende zwei Sonderfälle von C# aufgetaucht:

- Der Typ einer Variablen kann bei dessen Deklaration mittels der sog. „Type Inference“ ermittelt werden. Das bedeutet, dass sich der Typ durch den Ausdruck ergibt, welcher der Variable ihren Wert zuweist. In diesem Fall wird der Typ in der Variablendeklaration nicht explizit angegeben, sondern durch das Schlüsselwort `var` ersetzt. Die Auflösung des Typs geschieht dabei zum Zeitpunkt des Kompilierens. Wird während der Generierung der Faktenbasis beim Auflösen des Typs einer empfangenden Variable festgestellt, dass diese mit dem Schlüsselwort `var` deklariert wurde, so wird in der gegenwärtigen Implementierung des Plugins kein `Receiver`-Fakt angelegt. Als Alternative hätte man den Text „var“ als Typ-Referenz festlegen, und entsprechend darauf einen `Receiver`-Fakt anlegen können. Dies ist jedoch nicht ganz exakt, da an dieser Stelle im Quellcode keine Referenz auf den Typ existiert. Erst im - durch den Compiler erzeugten - IL Code liegt an dieser Stelle eine wirkliche Typ-Referenz vor. Wenn die Refacola-Infrastruktur mit der „Type Inference“ umgehen kann, macht es Sinn, die Refacola-Sprachdefinition für C# zu erweitern, um diesen Fall abzudecken.
  Vorstellbar wäre beispielsweise, ein Kind `InferredTypeReference` einzuführen. Dieses würde lediglich von `TypedEntityReference` erben und damit im Vergleich zur `TypeReference` auf das `Identifier`-Property verzichten, da ein Bezeichner hier nicht notwendig ist. Programmelemente von diesem Kind könnten als Empfänger in `Receiver`-Fakten verwendet werden und würden außerdem wiedergeben, dass es

[8] Dennoch war es im zeitlichen Rahmen dieser Arbeit nicht möglich, die gesamte Implementierung auf Geschwindigkeitspotenziale zu untersuchen. Rücksprachen mit dem Entwicklerteam von MonoDevelop haben ergeben, dass die gegenwärtige Implementierung lediglich versucht, alle Referenzen möglichst zuverlässig zu finden. Auf Performance wurde daher keinen Wert gelegt. Insofern ist anzunehmen, dass hier noch weitere Verbesserungen möglich sind.

sich dabei nicht um eine explizite Typ-Referenz, die im Quellcode wiedergefunden werden kann, handelt.

- Desweiteren können lokale Variablen in Lambda-Ausdrücken als Argumente deklariert werden. Dabei wird der Typ der Argumente nicht explizit angegeben. Das funktioniert wieder durch die angesprochene „Type Inference“. Aufgrund des Kontexts, indem der Lambda-Ausdruck verwendet wird, erkennt der Compiler, von welchem Typ das Argument sein muss. Da auch in diesem Fall keine explizite Typ-Referenz existiert, wird bei der Faktengenerierung wieder auf die Erstellung eines `Receiver`-Fakts verzichtet. Auch hier wäre es denkbar, die fehlende explizite Typ-Referenz mit einem Programmelement vom bereits erläuterten Kind `InferredTypeReference` zu ersetzen. Dieses Programmelement könnte dann ebenfalls in `Receiver`-Fakten als Empfänger dienen.

6.5. Accessibility None für PropertyGetter und -Setter

Es wurde bereits darauf eingegangen, dass die C#-Sprachspezifikation vorsieht, dass für PropertyGetter -oder -Setter, ein Zugriffsmodifizierer festgelegt werden kann, der einschränkender ist, als die Sichtbarkeit der Propertydeklaration (vgl. [ECM06, S. 300-305]). Dabei ist es nicht möglich, sowohl für den Getter, als auch für den Setter eine Sichtbarkeit explizit anzugeben. Im Zuge von Refaktorisierungen kann es deshalb notwendig sein, den explizit angegebenen Zugriffsmodifizierer wieder zu entfernen, damit die Sichtbarkeit wieder der des umschließenden Propertys entspricht. Aus diesem Grund wurde die Domäne `AccessModifier` in der Refacola-C#-Sprachdefinition um den Wert `none` erweitert[9]. Dieser ist ausschließlich für PropertyGetter und -Setter vorgesehen und darf nicht verwendet werden, um den Zugriffsmodifizierer von Typen oder Membern vermeintlich zu entfernen. Die Komponente `Refacola-Interface` weist alle Änderungen aus Change Sets zurück, die diese Vorgabe nicht einhalten. Deshalb würde es sich anbieten, für das `Accessibility`-Property der PropertyGetter -und -Setter Kinds eine eigene Domäne festzulegen, welche alle Werte der `AccessModifier`-Domäne erbt und diese um den Wert `none` erweitert. Dadurch würde bereits die Sprachdefinition verbieten, diesen Wert auf das `Accessibility`-Property von Typen oder Membern anzuwenden. Damit die Sprachdefinition auf diese Weise modifiziert werden könnte, müsste Refacola die Vererbung zwischen Domänen unterstützen.

[9]Desweiteren ist - um Verwechslungen zu vermeiden - noch darauf hinzuweisen, dass im Rahmen der Arbeit [ST09, S. 12] der Wert `absent` innerhalb der Domäne `AccessModifier` definiert wurde. Dieser unterscheidet sich jedoch von dem hier eingeführten Wert `none`. Wenn in Change Sets dem `Accessibility`-Property von Entitäten der Wert `absent` zugewiesen wird, so bedeutet das, dass die gesamte Entität aus dem Quellcode entfernt werden soll. Der Wert `none` besagt hingegen nur, dass die explizite Angabe des Zugriffsmodifizierers entfernt werden soll.

6.6. Caching

Während die Faktenbasis für eine geladene Solution erstellt wird, werden die verschiedensten Teile des AST mehrfach analysiert; jedes Mal jedoch unter einer unterschiedlichen Perspektive. Die Vorbereitung der Analyse dauert dabei häufig deutlich länger, als die Analyse selbst. So müssen Quellcode-Dateien beispielsweise erst vom C#-Parser verarbeitet werden, um auf der daraus erzeugten `CompilationUnit`-Instanz die Analyse durchführen zu können. Hierbei dauert der Vorgang des Parsens, bei dem jedes einzelne Textzeichen interpretiert werden muss, deutlich länger als die Analyse, die auf den erzeugten Informationen dann tatsächlich stattfindet. Man kann nun die Annahme treffen, dass - während die Faktenbasis für eine geladene Solution erzeugt wird - der AST nicht geändert, d.h. der Quellcode nicht editiert, wird. Unter dieser Voraussetzung ist es möglich, verschiedenste vorbereitende Maßnahmen nur ein einziges Mal durchzuführen, das Ergebnis danach kurzfristig im Speicher zu halten (engl. Caching), und bei erneutem Bedarf wieder aus dem Speicher zu holen. Im Rahmen dieser Arbeit wurde dieses Prinzip an verschiedenen Stellen angewendet, um die Geschwindigkeit der Faktengenerierung zu steigern.

6.6.1. NRefactoryResolverExt

Es wurde darauf hingewiesen, dass der `FindMemberAstVisitor` verwendet wird, um Referenzen auf Typen und Member innerhalb von `CompilationUnits` zu finden. Während der Visitor alle Knoten der `CompilationUnit` auf in Frage kommende Referenzen untersucht, wendet er sich an den eingeführten `NRefactoryResolver` (vgl. Abschnitt 3.4), um gefundene Codefragmente näher zu analysieren. Bei der Analyse überprüft der `NRefactoryResolver` beispielsweise, ob sich das übergebene Codefragment auf ein Objekt aus dem ProjectDom bezieht. Das Ergebnis der Analyse wird an den `FindMemberAstVisitor` zurückgegeben, welcher damit feststellen kann, ob eine gültige Referenz gefunden wurde. Da sich der AST nicht ändert, ist es ausreichend, die zeitintensive Analyse der Codefragmente nur ein einziges Mal durchzuführen und das Ergebnis zwischenzuspeichern. Verlangt der `FindMemberAstVisitor` zu einem späteren Zeitpunkt in der Faktengenerierung erneut nähere Informationen über ein Codefragment, so kann das zwischengespeicherte Ergebnis zurückgeliefert und dadurch Zeit gespart werden. Die Implementierung dieses Caching-Mechanismus wird in Abb. 6.1 skizziert. Es war notwendig, die öffentlichen Methoden des `NRefactoryResolvers`, die zur Analyse von Codefragmenten aufgerufen werden, als `virtual` zu deklarieren. Die `NRefactoryResolverExt`-Implementierung erbt vom `NRefactoryResolver`, überschreibt diese Methoden, und überprüft bei jedem Aufruf, ob für das Codefragment bereits ein Ergebnis vorliegt. Wenn nicht, wird der Aufruf an die Basisklasse durchgereicht und das Ergebnis im Speicher abgelegt.

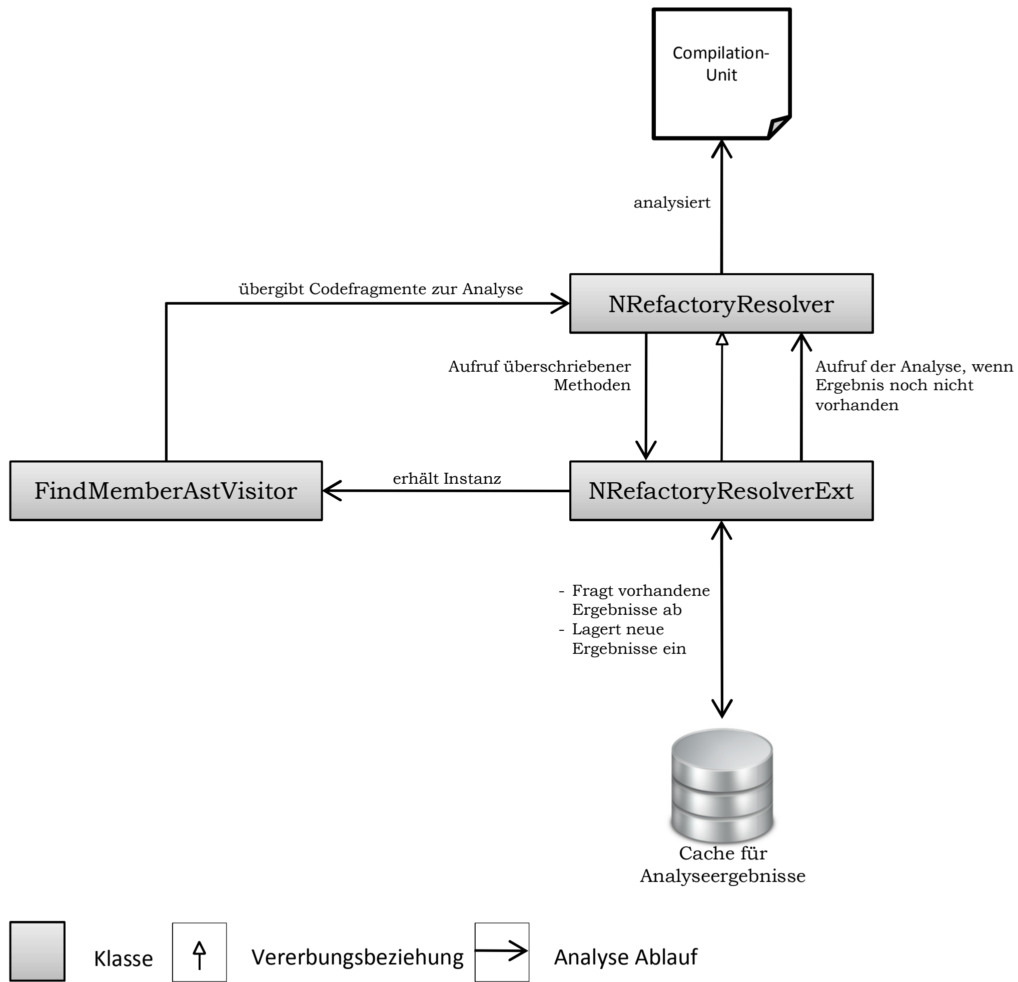


Abbildung 6.1.: NRefactoryResolverExt-Implementierung

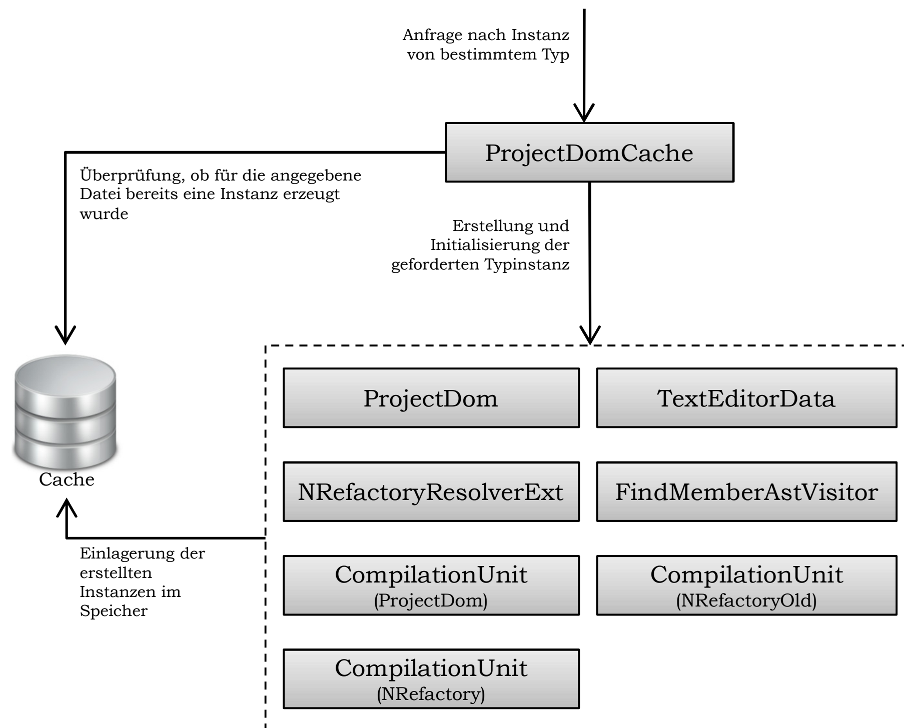


Abbildung 6.2.: ProjectDomCache

6.6.2. ProjectDomCache

Die Annahme, dass sich der Quellcode während der Analyse nicht verändert, kann noch an anderen Stellen ausgenutzt werden. Zu diesem Zweck wurde die Klasse `ProjectDomCache` im Namespace `ProjectDomAstModel` der `AST-Access`-Assembly implementiert. Sie ist der zentrale Cache, über den Instanzen von Klassen angefordert werden, die direkt auf den Quellcode-Dateien arbeiten. Wird für eine Quellcode-Datei mehrfach die gleiche Klasse angefordert, so muss diese nur einmal instanziert werden, wodurch die damit verbundene - teilweise sehr aufwendige - Initialisierung ebenfalls nur einmal durchgeführt werden muss. Abb. 6.2 skizziert den Aufbau des Caches.

Die Abbildung zeigt, welche Instanzen durch den `ProjectDomCache` verwaltet werden. Dabei sind folgende Dinge anzumerken:

- Muss eine `ProjectDom`-Instanz für eine Quellcode-Datei erzeugt werden, so wird zunächst überprüft, welchem Projekt die Datei zugeordnet ist, und anschließend die `ProjectDom`-Instanz zu diesem Projekt zurückgeliefert.

- `TextEditorData` ist eine Klasse von MonoDevelop aus dem Namespace `Mono.TextEditor.TextEditorData`. Über diese Klasse erhält man immer den aktuellen Stand des Quellcodes. Ist die Datei beispielsweise im Editor geöffnet und wurde bereits bearbeitet, so liefert diese Klasse den editierten Zustand zurück. Ist die Datei hingegen nicht geöffnet, wird der Inhalt direkt aus dem angegebenen Dateipfad ausgelesen.

- Der Cache verwaltet die `CompilationUnit`-Instanzen zur Abbildung von geparstem Quellcode aus folgenden Bereichen:
  - ProjectDom (vgl. Abschnitt 3.1)
  - NRefactoryOld (vgl. Abschnitt 3.2)
  - NRefactory (vgl. Abschnitt 3.3)

6.6.3. ProjectDomReferenceCache

In Abschnitt 5.5 wurde darauf eingegangen, dass innerhalb der `AST-Access`-Assembly der `ProjectDomReferenceFinder` zum Auflösen von Referenzen auf Typen und Member verwendet wird. Um das aufgelöste Ergebnis im Speicher zu halten, wurde der `ProjectDomReferenceCache` implementiert. Werden also für ein Member oder einen Typ im Zuge der Faktengenerierung mehrfach die Referenzen abgefragt, so muss die zeitintensive Auflösung nur einmal geschehen[10].

6.6.4. Zusammenfassung

Da die Cache-Implementierungen auf der Annahme basieren, dass sich der AST während der Analyse nicht ändert, ist es notwendig die Caches zu leeren, sobald der AST geändert wurde. Aus diesem Grund setzt das Plugin vor der Analyse des AST im Zuge der Faktengenerierung alle Caches explizit zurück. Außerdem ist noch darauf hinzuweisen, dass bei der Verarbeitung von Change Sets keine Caches zur Anwendung kommen dürfen, da sich hier der AST mit jeder abgearbeiteten Modifikation ändert.

[10] Für das entwickelte Plugin ergibt sich dadurch kein direkter Geschwindigkeitsvorteil, da der implementierte `SolutionTraverser` Referenzen von Typen und Membern nur auflöst, wenn er auf diese bei der Analyse des Quellcodes der Solution im Zuge des erläuterten „Top-Down"-Ansatzes trifft. Dies ist für die Typen und Member der Solution nur einmal der Fall, da die „Top-Down"-Analyse alle Knoten des AST - von der Wurzel bis zu den Blättern - nur einmal besucht. Der Cache wurde dennoch implementiert, um für spätere Entwicklungen vorzusorgen, die über das Ergebnis dieser Arbeit hinausgehen.

6.7. PropertyGetter und -Setter Referenzen

Die Refacola-C#-Sprachdefinition sieht vor, dass PropertyGetter und -Setter explizit neben ihrem umschließenden Property erfasst werden. Dadurch entsteht die Notwendigkeit, dass für Getter und Setter, die innerhalb der Solution deklariert wurden, auch die darauf verweisenden Referenzen aufgelöst und in der Faktenbasis eingetragen werden. Das Problem bestand nun darin, dass der `FindMemberAstVisitor` lediglich in der Lage ist, Referenzen auf Propertys im Allgemeinen ausfindig zu machen. Er kann nicht unterscheiden, ob es sich dabei um eine Referenz auf den Getter oder Setter des Propertys handelt. Aus diesem Grund wurde der `PropertyReferenceTypeResolvingAstVisitor` im Namespace `ProjectDomAstModel.NRefactoryVisitors` implementiert. Dabei handelt es sich - wie beim `FindMemberAstVisitor` - um einen NRefactoryOld Visitor zur Analyse von NRefactoryOld `CompilationUnits`. Wenn eine Referenz auf ein Property vom `FindMemberAstVisitor` gefunden wurde, so wird sie zur weiteren Analyse gemeinsam mit der `CompilationUnit`, in der sie sich befindet, an den `PropertyReferenceTypResolvingAstVisitor` gereicht. Dieser analysiert den Ausdruck, in dem sich die Referenz befindet, und kann dadurch feststellen, ob es sich um eine Referenz auf den Setter oder Getter handelt. Ist der Ausdruck vom Typ `AssignmentExpression`, so bedeutet das, dass der Referenz ein Wert zugewiesen wird. Entsprechend ist das Ergebnis, dass es sich um eine Referenz auf den PropertySetter handeln muss. Andernfalls wird davon ausgegangen, dass lesend auf das Property zugegriffen wird und es sich entsprechend um eine Referenz auf den PropertyGetter handelt.

6.8. Generics

Die C#-Sprachdefinition erlaubt die Verwendung von generischen Typen (engl. Generics) (vgl. [ECM06, S. 52-56]). Ein generischer Typ definiert bei dessen Deklaration einen oder mehrere Typparameter. Soll der Typ instanziert werden, so muss für jeden festgelegten Typparameter ein Typ übergeben werden, wodurch aus dem generischen ein konkreter Typ wird. Desweiteren kommt den Methoden im Rahmen der Generics eine Sonderrolle unter den Membern zu. So können Methodendeklarationen mit eigenen Typparametern (d.h. unabhängig von den Typparametern des umschließenden Typs) versehen werden, die bei deren Aufruf mit übergeben werden müssen. Da die Refacola-Infrastruktur zum Zeitpunkt der Entwicklung des Plugins noch nicht in der Lage war, mit Generics umzugehen, sieht die Refacola-C#-Sprachdefinition auch keine Elemente für deren Abbildung vor. Um dennoch für Solutions, die mit Generics arbeiten, eine Faktenbasis erzeugen zu können, mussten generischen Typen und Methoden durch Programmelemente repräsentiert werden, die sich an der festgelegten Sprachdefinition orientieren. In den folgenden Abschnitten werden zunächst die möglichen Fälle aufgezeigt, die Generics ermöglichen. Anschließend

wird deren derzeitige Abbildung bei der Faktengenerierung beleuchtet, bevor schließlich ein Vorschlag zur Modifikation der Sprachdefinition gemacht wird, mit der die gezeigten Fälle abgebildet werden können.

6.8.1. Mögliche Fälle

Listing 6.1 zeigt mögliche Szenarien, die mit Hilfe von Generics abgebildet werden können. Dabei ist zu beachten, dass die Rückgabetypen der dargestellten Felder auch für Propertys und Methoden verwendet werden können.

Listing 6.1: Verwendungsszenarien von Generics in C#

```
class GenericClass<TClassParameter>
    where TClassParameter : [Constraints]
{
    TClassParameter field1;

    OtherGenericClass<TClassParameter> field2;

    OtherGenericClass<string> field3;

    TMethodParameter Method<TMethodParameter>()
    {
        return default(TMethodParameter);
    }
}

class OtherGenericClass<T>
{}
```

Die generische Klasse `GenericClass` definiert mit `TClassParameter` einen Typparameter. Das Feld `field1` ist von dem Typ, welcher der Klasse bei deren Instanzierung im Typparameter `TClassParameter` übergeben wird. Das Feld `field2` hingegen ist vom generischen Typ `OtherGenericClass`, der durch Übergabe des Typparameters `TClassParameter` konkretisiert wird. Das Feld `field3` ist vom konkreten Typ `OtherGenericClass<string>`.
Die Methode `Method` deklariert den eigenen Typparameter `TMethodParameter`. Die Angabe eigener Typparameter ist Methoden vorbehalten und für Felder und Propertys nicht möglich.
Neben den hier dargestellten Fällen erlaubt die C#-Sprachspezifikation, für Typparameter bestimmte Einschränkungen festzulegen, wie dies am Typparameter `TClassParameter` angedeutet ist. Dazu werden die folgenden sog. „constraints“ definiert, um den Typ einzuschränken:

- „where T : struct“: Der Typ muss ein struct sein.
- „where T : class“: Der Typ muss eine Klasse sein.
- „where T : new()“: Der Typ muss einen öffentlichen und parameterlosen Konstruktor enthalten.
- „where T : <class name>“: Der Typ muss entweder der definierten Klasse entsprechen oder von ihr ableiten.
- „where T : <interface name>“: Der Typ muss das angegebene Interface implementieren.
- „where T : U“: Der Typ muss von dem Typ, der im Typparameter U angegebenen ist, ableiten.

6.8.2. Abbildung der Generics in der Faktenbasis

Um den in Listing 6.1 dargestellten Fall mit der definierten Sprachdefinition abbilden zu können, mussten folgende Vereinfachungen getroffen werden:

- Generische Typen werden genau wie nicht generische Typen behandelt. Das bedeutet, dass die Typparameter nicht beachtet werden. Somit wird lediglich ein `TopLevelClass`-Programmelement für die Klasse `GenericClass<>` erzeugt.
- Muss der Rückgabetyp von Membern aufgelöst werden, wird zuerst überprüft, ob dieser generisch ist, wie das für die Felder `field1` und `field2` sowie für die Methode `Method` der Fall ist oder ob es sich um einen vollständig typisierten generischen Typ, wie im Beispiel für das Feld `field3`, handelt. In all diesen Fällen wird der Rückgabetyp als Klasse `Object` aus dem Namespace `System` betrachtet.

Desweiteren ergeben sich auch bei der Auflösung der Receiver durch die Verwendung von Generics verschiedene Sonderfälle, die im Folgenden aufgezeigt werden. Die Liste erhebt dabei keinen Anspruch auf Vollständigkeit.

Listing 6.2: Sonderfall 1 beim Auflösen von Receivern

```
class GenericClass<TClassParameter>
{
    public TClassParameter field1;
}
```

```
class OtherClass
{
    public int SomeField = 0;
}

class CallingClass
{
    public void SomeMethod()
    {
        GenericClass<OtherClass> k =
            new GenericClass<OtherClass>();

        k.field1.SomeField = 5;
    }
}
```

In Listing 6.2 wird in der Methode `SomeMethod` eine Variable `k` vom konkreten Typ `GenericClass<OtherClass>` deklariert. Über dessen Feld `field1` - welches aufgrund der Konkretisierung vom Typ `OtherClass` ist - wird auf eine Instanz vom Typ `OtherClass` zugegriffen. Schließlich wird das Feld `SomeField` der Instanz gesetzt.
Der Receiver vom Zugriff auf das Feld `SomeField` ist die Referenz auf das Feld `field1` der Klasse `GenericClass`. Da die Variable `k` vom konkreten Typ `GenericClass<OtherClass>` ist, ist der Rückgabetyp des Feldes `field1` vom Typ `OtherClass`. Da die Refacola-C#-Sprachdefinition keine generischen Typen, die durch die Angabe der Typparameter zu konkreten Typen werden, abbilden kann, wird in diesem Fall kein `Receiver`-Fakt erzeugt.

Listing 6.3: Sonderfall 2 beim Auflösen von Receivern

```
class GenericClass<TClassParameter>
 {
    public TClassParameter field1;
 }

class CallingClass
{
    public void SomeMethod()
    {
        GenericClass<string> k =
            new GenericClass<string>();

        k.field1 = "test";
    }
}
```

In dem abgebildeten Fall aus Listing 6.3 wird über eine Variable `k`, die vom konkretisierten Typ `GenericClass<string>` ist, auf das Feld `field1` zugegriffen. Nach üblichem Vorgehen ist die empfangende Referenz die Typ-Referenz auf `GenericClass<string>` innerhalb der Deklaration der Variable `k`. Da die Sprachdefinition jedoch keine typisierten generischen Typen - und damit auch keine Referenzen auf diese - kennt, wird auch in diesem Fall kein `Receiver`-Fakt erzeugt.

Listing 6.4: Sonderfall 3 beim Auflösen von Receivern

```
class GenericClass<TClassParameter>
 {
    public TClassParameter field1;
 }

class CallingClass
{
    public void SomeMethod()
    {
        ((GenericClass<string>)new object()).field1
            = "test";
    }
}
```

Auch im Szenario aus Listing 6.4 wird aus den bekannten Gründen kein `Receiver`-Fakt angelegt, da der Typ, auf den gecastet wird, ein typisierter generischer Typ ist.

Listing 6.5: Sonderfall 4 beim Auflösen von Receivern

```
class GenericClass
 {
    public string field1;
 }

class CallingClass<TParameter>
    where TParameter : GenericClass
{
    public void SomeMethod(TParameter m)
    {
        m.field1 = "test";
    }
}
```

In dem Fall aus Listing 6.5 müsste die Referenz auf `GenericClass` bei den Constraints des Typparameters `TParameter` die Receiver-Referenz des Zugriffs auf Feld `field1` sein. Da

dies mit den vorhandenen Mitteln wieder nicht dargestellt werden kann, wird in solchen Fällen auch kein `Receiver`-Fakt erstellt.

6.8.3. Modifikation der Sprachdefinition zur Unterstützung von Generics

Aufgrund der betrachteten verschiedenen Fälle ergeben sich folgende Anforderungen an die Refacola-C#-Sprachdefinition:

- Generische Typen müssen als solche erfasst werden.
- Typparameter müssen erfasst werden und ihren generischen Typen zugeordnet werden können.
- Die möglichen Constraints für Typparameter müssen abgebildet werden können.
- Methoden müssen eigene Typparameter zugeordnet werden können (vgl. Methode `Method` in Listing 6.1).
- Werden Typparameter als Rückgabetypen von Membern verwendet, muss das abgebildet werden können (vgl. Feld `field1` in Listing 6.1).
- Generische Typen, die durch Angabe von konkreten Typen für die Typparameter konkretisiert werden, müssen abbildbar sein (vgl. Feld `field3` in Listing 6.1).
- Generische Typen, die durch die Angabe von generischen Typen für die Typparameter konkretisiert werden, müssen abbildbar sein (vgl. Feld `field2` in Listing 6.1).
- Referenzen auf Typparameter und typisierte generische Typen müssen abgebildet werden können.

Abbildung 6.3 zeigt einen Vorschlag zur Erweiterung der bisherigen Refacola-C#-Sprachdefinition (siehe Anhang unter A), mit dessen Hilfe die Anforderungen erfüllt werden könnten.

- Generische Typen werden über Programmelemente vom Kind `GenericType` abgebildet. Dabei bietet es sich wieder an - analog zur bisherigen Sprachdefinition - eine weitere Unterteilung in `GenericNested-` und `GenericTopLevelTypes` vorzunehmen und zwischen den verschiedenen Arten von Typen (Class, Struct, Interface, Enum, Delegate) zu unterscheiden.
- Generische Methoden werden mit dem Kind `GenericMethod` abgebildet.

```
Language C#
Kinds
  GenericEntity                    <: ENTITY
  GenericType                      <: GenericEntity, Type
  GenericMethod                    <: GenericEntity, InstanceMethod
  InstantiatedGenericEntity        <: ENTITY  GenericEntity
  InstantiatedGenericType          <: InstantiatedGenericEntity, Type
  InstantiatedGenericMethod        <: InstantiatedGenericEntity, InstanceMethod
  TypeParameter                    <: Type, NamedEntity   { GenericEntityOwner, MustBeClass,
                                                            MustBeStruct,
                                                            MustHavePublicDefaultConstructor }
Properties
  GenericEntityOwner                    :   GenericEntity
  MustBeClass                           :   Boolean
  MustBeStruct                          :   Boolean
  MustHavePublicDefaultConstructor      :   Boolean
Domains
  GenericEntity     = [ GenericEntity ]
  GenericType       = [ GenericType ]
  GenericMethod     = [ GenericMethod ]
  Boolean           = {true, false}
Queries
  genericConstraintSub                          (typeParameter: TypeParameter,
                                                baseType: Type)
  instantiatedGenericEntityTypeParameter        (entity: InstantiatedGenericEntity,
                                                param: TypeParameter, type: Type)
```

Abbildung 6.3.: Refacola-C#-Sprachdefinition Erweiterung

- Typparameter werden über Programmelemente vom Kind `TypeParameter` abgebildet. Über das Property `GenericEntityOwner` wird festgelegt, auf welchen generischen Typ oder welche generische Methode es sich bezieht. Die `MustBeClass`, `MustBeStruct`, und `MustHavePublicDefaultConstructor`-Propertys geben an, ob die möglichen Constraints `class`, `struct` oder `new` gesetzt sind. Über das `genericConstraintSub`-Query kann außerdem angegeben werden, von welchem Typ oder Typparameter der Typ, der für diesen Typparameter übergeben wird, erben muss.

- Da der Kind `TypeParameter` von `Type` erbt, können dessen Programmelemente zur Angabe des Rückgabetyps von Membern verwendet werden.

- Wird ein generischer Typ durch die Angabe von Typen für die Typparameter konkretisiert, so kann das durch ein Programmelement des Kinds `InstantiatedGenericType` dargestellt werden. Dieser Kind leitet von `InstantiatedGenericEntity` ab. Über dessen Property `GenericEntity` wird angegeben, auf welchen generischen Typ sich der konkretisierte Typ stützt.
  Analoges gilt für konkretisierte Methoden, die mit `InstantiatedGenericMethod`-Programmelementen abgebildet werden. Die übergebenen Typen für die Typparameter werden über das Query `instantiatedGenericEntityTypeParameter` abgebildet. Über dessen Argument `entity` wird die Verbindung zum `InstantiatedGenericEntity`, welches den konkreten Typ bzw. die konkrete Methode abbildet,

hergestellt. Das Argument `param` gibt an, auf welchen Typparameter sich der Fakt bezieht. Über das Argument `type` wird schließlich angegeben, welcher Typ für den Typparameter übergeben wurde. Da das Argument `type` vom Kind `Type` ist, und das Kind `TypeParameter` von `Type` erbt, kann mit diesem Query auch der Fall abgebildet werden wenn ein Typparameter bei Konkretisierung eines anderen Typs verwendet wird. Dies ist in Listing 6.1 beispielsweise für das Feld `field2` der Fall. Hier wird der Typparameter `TClassParameter` zur Konkretisierung des Feldtyps `OtherGenericClass` verwendet.

- Da `InstantiatedGenericType` und `TypeParameter` vom Kind `Type` erben, kann zur Abbildung von Referenzen auf konkretisierte generische Typen bzw. Typparameter das bereits vorhandene Kind `TypeReference` verwendet werden.

7. Ausblick und Fazit

In der Einleitung wurde aufgezeigt, welchen Mehrwert Refaktorisierungstools für Entwickler bieten, wenn sie sich auf deren Funktionsweise verlassen können. Mit Refacola wurde ein System geschaffen, welches sich genau das zum Ziel gesetzt hat: Refaktorisierungen intelligent durchzuführen, während syntaktische Korrektheit und semantische Unberührtheit des Quellcodes garantiert wird. Die Sprachunabhängigkeit Refacolas ermöglicht es, die Logik der Refaktorisierung nur einmal zu implementieren und anschließend für alle unterstützten Programmiersprachen wiederzuverwenden.
Mit dieser Arbeit wurde der Grundstein zur Unterstützung der Programmiersprache C# gelegt. Dabei musste zunächst eine Sprachdefinition entworfen werden, mit Hilfe derer sich der AST eines C#-Projekts abbilden lässt. Die Zielsetzung beim Entwurf der Sprachdefinition bestand darin, dass alle Elemente des AST abgebildet werden können, die benötigt werden, um das „Pull-Up-Field“-Refactoring durchzuführen. Im nächsten Schritt wurde ein - von einer konkreten Entwicklungsumgebung unabhängiges - Interface entworfen, über das der AST analysiert und modifiziert werden kann. Dieses Interface wurde anschließend für MonoDevelop implementiert.
Darüber hinaus wurde eine Komponente entwickelt, welche in der Lage ist, basierend auf den Daten, die das entworfene Interface liefert, eine Faktenbasis für Refacola zu erzeugen. Desweiteren kann diese Komponente Change Sets, die von Refacola erzeugt wurden, analysieren und die geforderten Modifikationen so aufbereiten, dass sie an eine beliebige Implementierung des entworfenen AST-Interfaces gereicht werden können.
Schließlich wurde ein Plugin für MonoDevelop entwickelt, welches dem Anwender über das Menü der IDE die Möglichkeit gibt, eine Faktenbasis zu erzeugen und Change Sets zu verarbeiten.
Die Kommunikation zwischen dem Plugin und der Refacola-Infrastruktur erfolgt dabei in indirekter Form, indem erzeugte Faktenbasen und Change Sets als Dateien über das Dateisystem ausgetauscht werden.
Um die Funktionstüchtigkeit des Plugins zu prüfen, wurden für Open Source Projekte (Json.NET[1], Protobuf[2]) Faktenbasen generiert. Die Change Sets wurden anschließend von Hand geschrieben und wieder an das Plugin zur Verarbeitung gereicht.
Um mit Hilfe von Refacola Refaktorisierungen im Produktivbetrieb von MonoDevelop durchführen zu können, müsste Refacola direkt im Plugin integriert werden, statt, wie

[1] http://json.codeplex.com/
[2] http://code.google.com/p/protobuf-net/

im gegenwärtigen Zustand, die Kommunikation über die Dateien durchzuführen. Desweiteren müsste die Faktenbasis, die für die Refaktorisierung verwendet wird, intelligent erzeugt werden. Denn für die meisten Refaktorisierungen dürfte es nicht notwendig sein, den gesamten AST in Form von Programmelementen und Fakten vorliegen zu haben. Stattdessen müsste eine Faktenbasis erzeugt werden, die ausgehend von dem zu refaktorisierenden Element alle darüber liegenden Knoten des AST sowie alle Referenzen des Elements und deren Receiver abbildet. Dieser iterative Vorgang dürfte eine für den Entwickler akzeptable Laufzeit aufweisen. Als Ausgangspunkt könnte dafür der bereits implementierte „Bottom-Up"-Ansatz des `SolutionTraversers` verwendet werden.
Um die Refaktorisierung weiter zu beschleunigen könnte außerdem ein Hintergrundthread Leerlaufzeiten nutzen, in denen der Prozessor nicht ausgelastet ist, um eine Faktenbasis des gesamten AST zu erzeugen. Werden Codeänderungen festgestellt, müssten diese in der Faktenbasis nachgezogen werden. Desweiteren würde es sich anbieten, die Faktenbasis nicht nur im Speicher zu halten, sondern auszulagern, damit die Daten bei späterem erneutem Öffnen des Projekts wiederverwendet werden können und nicht nochmals generiert werden müssen. Weitere Geschwindigkeitsvorteile ließen sich erzielen, wenn mehrere Threads gleichzeitig (engl. Multithreading) die Faktengenerierung übernehmen würden.
Um neben dem „Pull-Up-Field"-Refactoring noch weitere Refaktorisierungen zu unterstützen wäre es außerdem notwendig, die entwickelte Sprachdefinition um Programmelemente und Fakten zu erweitern, die von diesen Refaktorisierungen benötigt werden[3].
Um dem Entwickler ein wirklich zuverlässiges und nützliches Refaktorisierungstool bieten zu können, wäre es desweiteren noch notwendig, die Verwendung von Generics zu unterstützen.
Sobald verschiedene Refaktorisierungen mittels Refacola vollständig in MonoDevelop integriert wurden und im Produktivbetrieb verwendet werden können, wäre es außerdem denkbar, die Implementierungen auf andere populäre C#-Entwicklungsumgebungen, wie SharpDevelop oder Visual Studio, zu übertragen. Bei dem Entwurf der Architektur des Plugins wurde Wert darauf gelegt, die Implementierungen nicht an eine bestimmte Entwicklungsumgebung zu koppeln. Deshalb wäre es für die Unterstützung weiterer IDEs lediglich notwendig, eine eigene `AST-Access`-Assembly zu entwickeln.
Mit dem Einzug Refacolas in die gegenwärtig meistverwendeten Entwicklungsumgebungen wäre ein wertvoller Beitrag zu modernen agilen Entwicklungsprozessen geleistet, da dem Entwickler damit ein verlässliches Hilfsmittel zur Verfügung stehen würde, das ihm die zeitraubenden und fehleranfälligen Routinearbeiten von Hand durchgeführter Refaktorisierungen abnehmen könnte.

[3] Entsprechend müssten die Klassen `SolutionTraverser`, `ReferenceResolver` und `ReferenceReceiverResolver`, die gemeinsam die Faktenbasis erzeugen, angepasst werden. Wenn diese Refaktorisierungen neben den bereits implementierten Änderungen an Programmelementen noch weitere Properties ändern, müsste zusätzlich die Gruppe der `RefactoringInput`-Datencontainer erweitert werden. Außerdem wäre noch der `RefactoringController` zu erweitern, um die neu eingeführten Änderungen durchführen zu können.

A. C#-Sprachdefinition

Language C#
Kinds

NamedReference	<: REFERENCE	{ Identifier }
TypeReference	<: NamedReference, TypedEntityReference	{ HostNameSpace }
ReferenceInType	<: NamedReference	{ Owner, TopLevelOwner, HostNameSpace }
TypedEntityReference	<: ReferenceInType	{ InferredType }
MemberOrConstructorReference	<: NamedReference, ReferenceInType	
AssignableReference	<: TypedEntityReference	
MemberReference	<: MemberOrConstructorReference, TypedEntityReference	
FieldReference	<: MemberReference, AssignableReference	
MethodReference	<: MemberReference	
PropertyGetterReference	<: MemberReference	
PropertySetterReference	<: MemberReference, AssignableReference	
ThisReference	<: TypedEntityReference	
ImplicitThisReference	<: ThisReference	
ExplicitThisReference	<: ThisReference	
DeclaredEntity	<: ENTITY	{ HostNameSpace, TopLevelOwner }
NamedEntity	<: ENTITY	{ Identifier }
AccessibleEntity	<: ENTITY	{ Accessibility }
TypedEntity	<: ENTITY	{ DeclaredType, Owner }
MemberOrConstructor	<: DeclaredEntity, AccessibleEntity	{ Owner }
Constructor	<: MemberOrConstructor	
Member	<: MemberOrConstructor, NamedEntity	
HideableMember	<: Member	
TypedMember	<: Member, TypedEntity, DeclaredEntity	
InstanceMethod	<: TypedMember	
Namespace	<: NamedEntity	
Field	<: TypedMember, HideableMember, NamedEntity, AccessibleEntity	
Property	<: TypedMember	
PropertyGetter	<: TypedMember	{ Property }
PropertySetter	<: TypedMember	{ Property }
Type	<: AccessibleEntity, DeclaredEntity	
NestedType	<: Type, NamedEntity	{ Owner }
NestedClass	<: NestedType	
NestedInterface	<: NestedType	
NestedStruct	<: NestedType	
NestedEnum	<: NestedType	
NestedDelegate	<: NestedType	
TopLevelType	<: Type, NamedEntity	
TopLevelClass	<: TopLevelType	
TopLevelInterface	<: TopLevelType	
TopLevelStruct	<: TopLevelType	
TopLevelEnum	<: TopLevelType	
TopLevelDelegate	<: TopLevelType	

Properties

InferredType	"\\iota":	Identifier
Accessibility	"\\alpha":	AccessModifier
Owner	"\\lambda":	Type
TopLevelOwner	"\\lambda_T":	TopLevelType
HostNameSpace	"\\pi":	Namespace
DeclaredType	"\\tau":	Type
InferredType	"\\tau":	Type
Property	"\\prop":	Property

Domains

AccessModifier	= {private, protected, internal, protected internal, public, none}
TopLevelType	= [TopLevelType]
Namespace	= [Namespace]
Type	= [Type]
Property	= [Property]

Queries

binds	(reference: Reference, entity: Entity)
receiver	(memberAccess : MemberReference, receiver: TypedEntityReference)
sub	(t1 : Type, t2 : Type)
initialassignment	(f: Field, r : TypedEntityReference)
enclosed	(r: Reference, m: MethodOrConstructor)
member	(T : Type, M : Member)
type	(e : TypedEntityReference, T : Type)

B. Java-Sprachdefinition nach [SvP11, S. 18]

Language Java

Kinds

NamedReference	<: REFERENCE	{ identifier }
TypeReference	<: NamedReference	{ hostPackage }
ReferenceInType	<: REFERENCE	{ owner, tlowner, hostPackage }
TypedEntityReference	<: ReferenceInType	{ inferredType }
DeclaredEntity	<: ENTITY	{ hostPackage, tlowner }
NamedEntity	<: ENTITY	{ identifier }
AccessibleEntity	<: ENTITY	{ accessibility }
TypedEntity	<: ENTITY	{ declaredType, owner }
MemberOrConstructor	<: DeclaredEntity, AccessibleEntity	{ owner }
MethodOrConstructorOrInitializer	<: Entity	{ owner }
Package	<: ENTITY	{ identifier }
MemberOrConstructorReference	<: NamedReference, ReferenceInType	
AssignableReference	<: TypedEntityReference	
MemberReference	<: MemberOrConstructorReference, TypedEntityReference	
FieldReference	<: MemberReference, AssignableReference	
ThisReference	<: TypedEntityReference	
Member	<: MemberOrConstructor, NamedEntity	
HideableMember	<: Member	
TypedMember	<: Member, TypedEntity, DeclaredEntity	
InstanceMember	<: TypedMember	
Field	<: TypedMember, HideableMember, NamedEntity, AccessibleEntity	
Type	<: AccessibleEntity, DeclaredEntity	
Interface	<: Type	
TopLevelType	<: Type, NamedEntity	

Properties

identifier	”\\iota”:	Identifier
accessibility	”\\alpha”:	AccessModifier
location	”\\lambda ”:	Type
location_T	”\\lambda_T”:	LocationType
package	”\\pi”:	LocationPackage
declaredType	”\\tau”:	Type
inferredType	”\\tau”:	Type

Domains

AccessModifier	= {private, package, protected, public}
LocationType	= [TopLevelType]
LocationPackage	= [Package]
Type	= [Type]

Queries

binds	(reference: Reference, entity: Entity)
receiver	(memberAccess : MemberReference, receiver: TypedEntityReference)
sub	(t1 : Type, t2 : Type)
initialassignment	(f: Field, r : TypedEntityReference)
enclosed	(r: Reference, m: MethodOrConstructorOrInitializer)
member	(T : Type, M : Member)
type	(e : TypedEntityReference, T : Type)

C. Sprachunabhängige Unterschiede zwischen den Refacola-Sprachdefinitionen

C.1. Typen

Allgemein wird bei allen Typen im C#-Modell differenziert, ob sie innerhalb eines anderen Typs deklariert sind (nested) oder nicht (top level). TopLevel-Typen leiten von `TopLevelType` ab, während verschachtelte Typen von `NestedType` erben. Sowohl `TopLevelType`, also auch `NestedType`, stammen von `Type` ab. Das Java-Modell in B definiert kein Kind für verschachtelte Typen.

C.2. Typereference

Programmelemente vom Kind `TypeReference` werden verwendet, um alle Referenzen auf Klassen, Interfaces, Structs, Enums und Delegates abzubilden. Da sich diese Referenzen ebenfalls innerhalb eines bestimmten Typen befinden und einen Typen abbilden, leitet `TypeReference` im C#-Modell nicht direkt von `NamedReference`, sondern von `TypedEntityReference` ab. Damit werden die Referenzen um die Propertys `Owner`, `TopLevelOwner`, und `InferredType` erweitert.

C.3. This Referenzen

Das C#-Modell unterscheidet zwischen expliziten und impliziten this Referenzen. Explizite Referenzen werden durch Programmelemente des Kinds `ExplicitThisReference` abgebildet, wohingegen implizite Referenzen durch das Kind `ImplicitThisReference` dargestellt werden.

C.4. Methoden

Das Java-Modell definiert ein Kind `InstanceMember`. Dieser Name suggeriert, dass diese Elemente jegliche Member, die in einem Typ deklariert werden können, abbilden. Da jedoch beispielsweise `Field` nicht von diesem Element, sondern direkt von `TypedMember`, ableitet, wird im C#-Modell komplett darauf verzichtet. Für Methoden wird stattdessen das Kind `InstanceMethod` eingeführt, welches wiederum ein `TypedMember` ist. Dieses ist vor allem vor dem Hintergrund notwendig, dass Feldinitialisierer auch statische Methoden aufrufen können. Somit müssen diese Referenzen für ein erfolgreiches „Pull-Up-Field“-Refactoring abgebildet werden können.

D. AST-Modell Klassendiagramm

Das Klassendiagramm ist in 3 Teile aufgeteilt, um die Übersichtlichkeit zu wahren. Abb. D.1 zeigt ausgehend von der Solution, als Wurzelknoten des AST, alle weiteren Knoten bis zu den verschiedenen Typen. Abb. D.2 illustriert die verschiedenen Member und die darunter bestehenden Vererbungsbeziehungen. Abb. D.3 zeigt letztlich die Modellentitäten, die Referenzen und deren Receiver abbilden. Die Interfaces tragen alle das Suffix „Container", um verwirrende Namenskonflikte mit bereits bestehenden Typnamen aus den MonoDevelop-Namespaces zu vermeiden. In den Abbildungen sind lediglich Vererbungshierarchien eingezeichnet, während die Abhängigkeiten - der Übersichtlichkeit wegen - ausgeblendet sind.

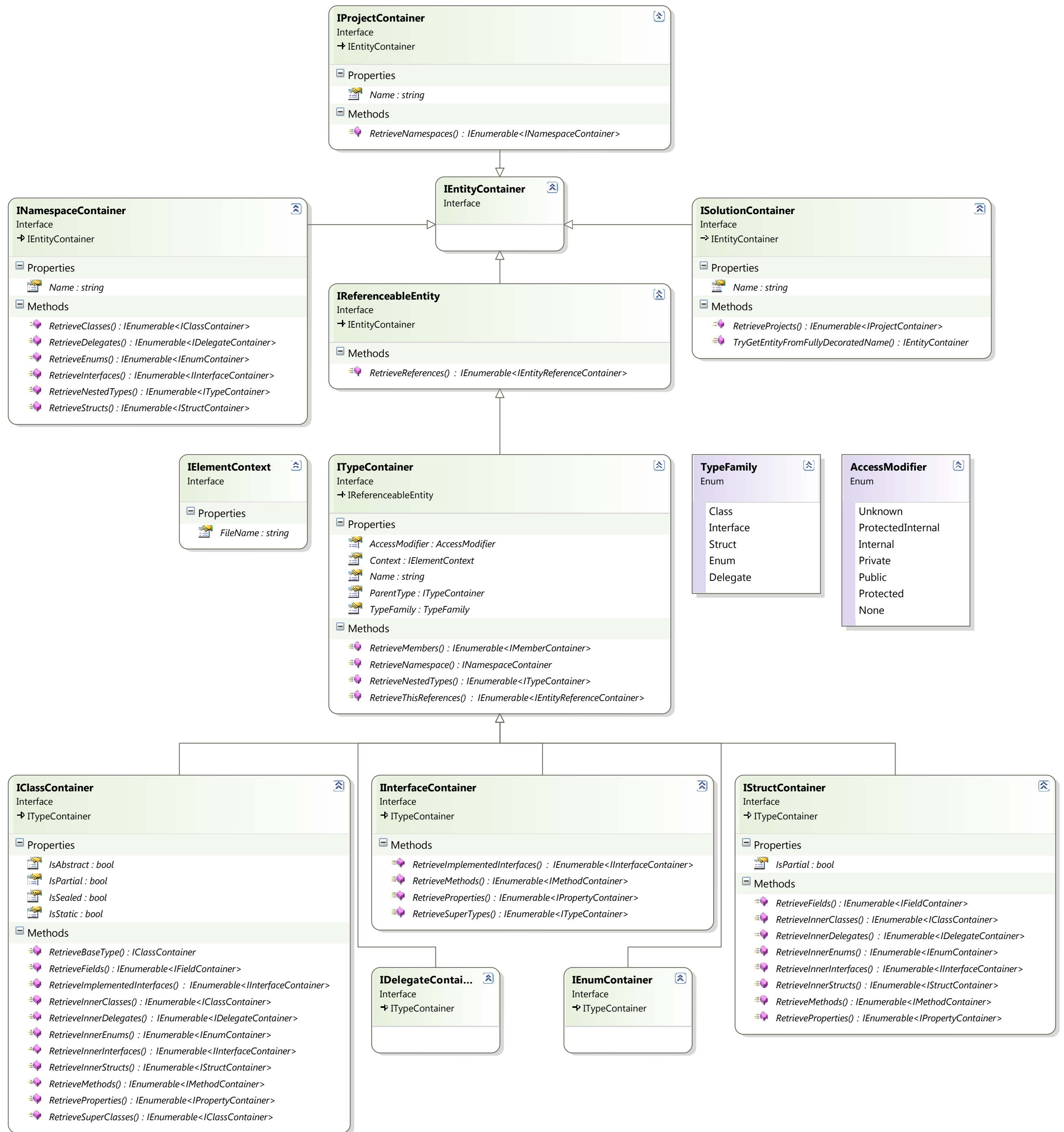


Abbildung D.1.: AST-Modell Teil 1 - Solution, Projects, Namespaces und Typen

Abbildung D.2.: AST-Modell Teil 2 - Members

Abbildung D.3.: AST-Modell Teil 3 - Referenzen und Receiver

E. SolutionTraverser-Logik

Anmerkung: Der im Anhang verwendete Pseudo-Code lehnt sich an die C#-Syntax an, ohne Variablen explizit deklarieren zu müssen. Es wurde versucht, die Syntax auf das Wesentliche zu reduzieren, um maximale Lesbarkeit zu erreichen. Desweiteren werden beim Aufruf von Methoden, die nicht Teil des Pseudo-Codes sind, die Übergabeparameter explizit benannt, um zu verdeutlichen, zu welchem Sinn die Parameter übergeben werden. Die Methodennamen wurden dabei so gewählt, dass sie möglichst gut die Aufgabe der aufgerufenen Methoden beschreiben.

Listing E.1: Pseudo-Code zur Demonstration der SolutionTraverser-Logik

```
function TraverseSolutionAndGenerateFacts(Solution)
{
    foreach(Project in Solution)
    {
        foreach(Namespace in Project)
        {
            NamespaceProgramElement =
                CreateProgramElementForNamespace(Namespace);

            foreach(Type in Namespace)
            {
                TypeProgramElement =
                    CreateProgramElementForTopLevelType(Type,
                                                        NamespaceProgramElement);

                TraverseTypeToLeafs(Type, TypeProgramElement);
            }
        }
    }
}

function CreateProgramElementForTopLevelType(Type, NamespacePE)
{
    switch(Type.Kind)
    {
        case Class:
            CreateProgramElementForTopLevelClass(Type: Type,
                                                 HostNamespace: NamespacePE);
            TraverseClassHierarchie(Type, CreatedProgramElement);
        case Struct:
            CreateProgramElementForTopLevelStruct(Type: Type,
                                                  HostNamespace: NamespacePE);
        case Interface:
            CreateProgramElementForTopLevelInterface(Type: Type,
                                                     HostNamespace:NamespacePE);
        case Enum:
            CreateProgramElementForTopLevelEnum(Type: Type,
                                                HostNamespace: NamespacePE);
        case Delegate:
            CreateProgramElementForTopLevelDelegate(Type: Type,
                                                    HostNamespace: NamespacePE);
```

```
    }

    return CreatedProgramElement;
}

function TraverseClassHierarchie(Class, ClassProgramElement)
{
    if(Class.HasBaseClass)
    {
        BaseClassProgramElement = GetTypeKindAndTraverseToRoot(Class.BaseClass);

        CreateSubFact(SubClass: ClassProgramElement,
                      BaseClass: BaseClassProgramElement);
    }
}

public function GetTypeKindAndTraverseToRoot(Type)
{
    if(Type.IsTopLevelType)
    {
        NamespaceProgramElement =
            CreateProgramElementForNamespace(Type.Namespace);

        CreateProgramElementForTopLevelType(Type, NamespaceProgramElement);
    }
    else
    {
        OwnerProgramElement = GetTypeKindAndTraverseToRoot(Type.ParentType);

        CreateProgramElementForNestedType(Type, OwnerProgramElement);
    }

    return CreatedProgramElement;
}

function CreateProgramElementForNestedType(Type, Owner)
{
    switch(Type.Kind)
    {
        case Class:
            TraverseClassHierarchie(Type);
            CreateProgramElementForNestedClass(Type: Type, Owner: Owner);
        case Struct:
            CreateProgramElementForNestedStruct(Type: Type, Owner: Owner);
        case Interface:
            CreateProgramElementForNestedInterface(Type: Type, Owner: Owner);
        case Enum:
            CreateProgramElementForNestedEnum(Type: Type, Owner: Owner);
        case Delegate:
            CreateProgramElementForNestedDelegate(Type: Type, Owner: Owner);
    }

    Return CreatedProgramElement;
}

function TraverseTypeToLeafs(Type, TypeProgramElement)
{
    foreach(NestedType in Type)
    {
        NestedTypeProgramElement =
            CreateProgramElementForNestedType(NestedType,
```

```
                                              TypeProgramElement);

        TraverseTypeToLeafs(NestedType, NestedTypeProgramElement);
    }

    foreach(Member in Type)
    {
        DeclaredTypeProgramElement =
            GetTypeKindAndTraverseToRoot(Member.ReturnType);

        MemberProgramElement =
            CreateProgramElementForMember(Member,
                                          TypeProgramElement,
                                          DeclaredTypeProgramElement);

        CreateMemberFact(Type: TypeProgramElement, Member: MemberProgramElement);

        ReferenceResolver.ResolveReferencesOnMember(Member,
                                                    MemberProgramElement);

        If(Member is Property)
        {
            GetterProgramElement =
                CreateProgramElementForMember(Member.PropertyGetter,
                                              TypeProgramElement,
                                              DeclaredTypeProgramElement);

            ReferenceResolver.ResolveReferencesOnMember(Member.PropertyGetter,
                                                        GetterProgramElement);

            SetterProgramElement =
                CreateProgramElementForMember(Member.PropertySetter,
                                              TypeProgramElement,
                                              DeclaredTypeProgramElement);

            ReferenceResolver.ResolveReferencesOnMember(Member.PropertySetter,
                                                        SetterProgramElement);

        }
    }

    ReferenceResolver.ResolveReferencesOnType(Type, TypeProgramElement);
}

function CreateProgramElementForMember(Member, Owner, DeclaredType)
{
    switch (Member.Kind)
    {
        case Field:
            CreateProgramElementForField(
                Field: Member,
                Owner: Owner,
                DeclaredType: DeclaredType);
        case Method:
            CreateProgramElementForMethod(
                Method: Member,
                Owner: Owner,
                DeclaredType: DeclaredType);
        case Constructor:
            return CreateProgramElementForConstructor(
                Constructor: Member,
```

```
                Owner: Owner,
165             DeclaredType: DeclaredType);
        case Property:
            CreateProgramElementForProperty(
                Property: Member,
                Owner: Owner,
170             DeclaredType: DeclaredType);
        Case PropertyGetter:
            CreateProgramElementForPropertyGetter(
                PropertyGetter: Member,
                Owner: Owner,
175             DeclaredType: DeclaredType);
        Case PropertySetter:
            CreateProgramElementForPropertySetter(
                PropertySetter: Member,
                Owner: Owner,
180             DeclaredType: DeclaredType);
    }

    return CreatedProgramElement;
}
185
public function GetMemberKindAndTraverseToRoot(Member)
{
    Owner = GetTypeKindAndTraverseToRoot(Member.EnclosingType);
    DeclaredType = GetTypeKindAndTraverseToRoot(Member.ReturnType);
190
    return CreateProgramElementForMember(Member, Owner, DeclaredType);
}
```

F. ReferenceResolver-Logik

Die Methode `ResolveReferencesOnMemberAndBuildQueries` steht hier stellvertretend für die Methoden, welche die Referenzen auf Felder, PropertyGetter, PropertySetter und Methoden auflösen. Da die Logik identisch ist, wurde sie hier nur einmal abgebildet.

Listing F.1: Pseudo-Code zur Demonstration der ReferenceResolver-Logik

```
function ResolveReferencesOnTypeAndBuildQueries(Type, TypeProgramElement)
{
    foreach(Reference in Type.References)
    {
        OwnerProgramElement = SolutionTraverser.
            GetTypeKindAndTraverseToRoot(Reference.RetrieveEnclosingType());

        TypeReferenceProgramElement =
            CreateProgramElementForTypeReference(
                Owner: OwnerProgramElement,
                InferredType: TypeProgramElement);

        CreateBindsFact(Reference: TypeReferenceProgramElement,
                        Entity: TypeProgramElement);

        EnclosingMemberProgramElement = SolutionTraverser.
            GetMemberKindAndTraverseToRoot(Reference.RetrieveEnclosingMember());

        CreateEnclosedFact(Reference: TypeReferenceProgramElement,
                           EnclosedBy: EnclosingMemberProgramElement);

        CreateTypeFact(Reference: TypeReferenceProgramElement,
                       ReferenceType: TypeProgramElement);

        if(EnclosingMemberProgramElement is Field)
        {
            CreateInitialAssignmentFact(Field: EnclosingMemberProgramElement,
                                        Reference: TypeReferenceProgramElement);
        }
    }
}

function ResolveReferencesOnMemberAndBuildQueries(Member, MemberProgramElement)
{
    foreach(Reference in Member.References)
    {
        OwnerProgramElement = SolutionTraverser.
            GetTypeKindAndTraverseToRoot(Reference.RetrieveEnclosingType());

        MemberReferenceProgramElement =
            CreateMemberReference(
                Owner: OwnerProgramElement,
                InferredType: MemberProgramElement.DeclaredType);
```

```
45          CreateBindsFact(Reference: MemberReferenceProgramElement,
                            Entity:MemberProgramElement);

            EnclosingMemberProgramElement = SolutionTraverser.
                GetMemberKindAndTraverseToRoot(Reference.RetrieveEnclosingMember());
50
            CreateEnclosedFact(Reference: MemberReferenceProgramElement,
                               EnclosedBy: EnclosingMemberProgramElement);

            CreateTypeFact(Reference: MemberReferenceProgramElement,
55                         ReferenceType: MemberProgramElement.DeclaredType);

            if(EnclosingMemberProgramElement is Field)
            {
                CreateInitialAssignmentFact(
60                  Field: EnclosingMemberProgramElement,
                    Reference: MemberReferenceProgramElement);
            }
        }
    }
```

G. ReferenceReceiverResolver-Logik

Listing G.1: Pseudo-Code zur Demonstration der ReferenceReceiverResolver-Logik

```
function ResolveReceiverAndBuildQueryForReference(MemberReferenceProgramElement,
                                                   Reference)
{
    Receiver = Reference.RetrieveReferenceReceiver();

    if(Receiver is ThisReferenceReceiver)
    {
        ThisReferenceProgramElement = ReferenceResolver.
            ResolveThisReferenceAndBuildQueries(
                Receiver.ReceivingReference,
                Receiver.IsExplicit);

        CreateReceiverFact(MemberReferenceProgramElement,
                           ThisReferenceProgramElement);
    }

    if(Receiver is CastReferenceReceiver)
    {
        ReceivingType = Receiver.ReceivingTypeReference.RetrieveTargetEntity();

        ReceivingTypeProgramElement = SolutionTraverser.
            GetTypeKindAndTraverseToRoot(ReceivingType);

        ReceivingTypeReferenceProgramElement = ReferenceResolver.
                ResolveReferenceOnTypeAndBuildQueries(
                    ReceivingTypeProgramElement,
                    Receiver.ReceivingTypeReference);

        CreateReceiverFact(MemberReferenceProgramElement,
                           ReceivingTypeReferenceProgramElement);
    }

    if(Receiver is MemberReferenceReceiver)
    {
        ReceivingReferenceProgramElement =
                ResolveProgramElementForMemberReference(
                    Receiver.ReceivingReference);

        CreateReceiverFact(MemberReferenceProgramElement,
                           ReceivingReferenceProgramElement);

    }

    if(Receiver is IdentifierReferenceReceiver)
    {
        IdentifierDeclaration =
            Receiver.IdentifierReference.ResolveDeclaration();

        if(IdentifierDeclaration is IdentifierAsLocalVariableDeclaration or
```

```
50                                    IdentifierAsMethodParameterDeclaration or
                                      IdentifierAsTypeDeclaration)
          {
              TypeProgramElement = SolutionTraverser.
                  GetTypeKindAndTraverseToRoot(IdentifierDeclaration.Type);
55
              TypeReferenceProgramElement = ReferenceResolver.
                  ResolveReferenceOnTypeAndBuildQueries(
                      TypeProgramElement,
                      IdentifierDeclaration.TypeReference);
60
              CreateReceiverFact( MemberReferenceProgramElement,
                          TypeReferenceProgramElement);
          }

65        if(IdentifierDeclaration is IdentifierAsMemberDeclaration)
          {
              ReceivingReferenceProgramElement =
                  ResolveProgramElementForMemberReference(
                      IdentifierDeclaration.MemberReference);
70
              CreateReceiverFact(MemberReferenceProgramElement,
                                 ReceivingReferenceProgramElement);

          }
75    }

  }

  function ResolveProgramElementForMemberReference(MemberReference)
80 {
      Member = MemberReference.RetrieveTargetEntity();
      MemberProgramElement = SolutionTraverser.
          GetMemberKindAndTraverseToRoot(Member);

85    if(Member is Field)
      {
          MemberReferenceProgramElement = ReferenceResolver.
              ResolveReferenceOnFieldAndBuildQueries(
                  MemberProgramElement,
90                MemberReference);
      }

      if(Member is Method)
      {
95        MemberReferenceProgramElement = ReferenceResolver.
              ResolveReferenceOnMethodAndBuildQueries(
                  MemberProgramElement,
                  MemberReference);
      }
100
      if(Member is PropertyGetter)
      {
          MemberReferenceProgramElement = ReferenceResolver.
              ResolveReferenceOnPropertyGetterAndBuildQueries(
105               MemberProgramElement,
                  MemberReference);
      }

      return MemberReferenceProgramElement;
110 }
```

H. Plugin Bedienungsanleitung

In den folgenden Abschnitten wird die Verwendung des Plugins erläutert. Vorab ist zu erwähnen, dass das Plugin neben der Generierung von Faktenbasen sowie dem Verarbeiten von Change Sets noch die Möglichkeit bietet, den AST der geladenen Solution zu visualisieren. Da dies jedoch keine direkte Relevanz für das Thema dieser Arbeit hatte, wurde die Implementierung der Visualisierung nicht beleuchtet.
Desweiteren wird darauf hingewiesen, dass das Plugin gegen die MonoDevelop Version 2.6 RC1 (2.5.93), welche am 11.06.2011 den aktuellsten Stand in der MonoDevelop Versionskontrolle[1] darstellte, entwickelt wurde. Dabei handelt es sich um keine Version, die in Form eines offiziellen Releases veröffentlichte wurde. Es musste mit der aktuellen Entwicklungsversion gearbeitet werden, da die Komponente NRefactory (vgl. Abschnitt 3.3) zum Zeitpunkt der Entwicklung des Plugins noch immer vom MonoDevelop Team entwickelt wurde und damit noch nicht Bestandteil eines veröffentlichten Releases war. Es wurde darauf verzichtet, aktuellere Entwicklungsstände des MonoDevelop Teams, die nach dem 11.06.2011 in die Versionskontrolle aufgenommen wurden, zu verwenden, da diese immer einen deutlich instabileren Eindruck machten. Sobald die NRefactory-Komponente Teil eines offiziellen Releases wird, kann es durchaus Sinn machen, das entwickelte Plugin mit dieser Version zu verwenden. Dabei könnten allerdings Änderungen am Plugin notwendig werden, wenn sich die genutzten Schnittstellen MonoDevelops geändert haben.

H.1. Installation

Um das Plugin in MonoDevelop verwenden zu können, ist es vorab nötig, im Unterordner „AddIns“ des MonoDevelop-Installationsverzeichnisses ein beliebiges Verzeichnis, wie beispielsweise „Refacola“ zu erstellen. In diesen sind die kompilierten Assemblys `Contracts`, `AST-Access`, `Refacola-Interface`, `MonoDevelop-Extension` und `Visualization` sowie deren Abhängigkeiten zu kopieren. Damit ist die Installation komplett und das Plugin nach dem nächsten Start der IDE verwendbar. Abb. H.1 zeigt beispielhaft den Ordner „Refacola“ nach der Installation.

Abbildung H.1.: Verzeichnisstruktur nach Installation des Plugins

[1] `https://github.com/mono/monodevelop/`

H.2. Generierung einer Faktenbasis

Wenn das Plugin korrekt installiert wurde erscheint im Menü der IDE der Eintrag „FernUniHagen“, über das folgende Kommandos ausgeführt werden können:

- „Explore Loaded Solution“: Über diesen Eintrag kann die geöffnete Solution analysiert werden. Wenn keine Solution geöffnet ist, ist der Menüpunkt ausgegraut.
- „Generate Factbase“: Über dieses Kommando wird die Faktenbasis für die geöffnete Solution - zunächst nur im Speicher - erzeugt. Der Menüpunkt ist ausgegraut, wenn keine Solution geöffnet ist.
- „Export Factbase“: Über diesen Eintrag lässt sich die zuvor erzeugte Faktenbasis exportieren. Dieser Menüpunkt ist nur aktiv, wenn eine Faktenbasis generiert wurde.
- „Apply Change Set“: Über dieses Kommando lassen sich von Refacola erzeugte Change Sets verarbeiten. Auch dieser Menüpunkt ist nur aktiv, wenn eine Faktenbasis generiert wurde, da sonst keine Relation von den Programmelementen, die im Change Set modifiziert werden, zum AST existiert.

Die Verwendung des Plugins wird dabei anhand des Quellcodes aus Listing H.1 demonstriert. Ziel wird es sein, zunächst die Faktenbasis zu generieren und anschließend ein Change Set manuell anzulegen, über welches das Feld `TestField` in die Basisklasse `BaseClass` verschoben wird.

Listing H.1: Quellcode zur Demonstration des Plugins

```
using System;

namespace DemonstrationSolution
{
    class MyClass : BaseClass
    {
        int TestField;

        void TestMethod()
        {
            this.TestField = 15;
        }
    }

    class BaseClass
    {

    }
}
```

In Abb. H.2 ist der initiale Zustand dargestellt, wenn eine Solution gerade geöffnet oder neu angelegt wurde. Da noch keine Faktenbasis erzeugt wurde, sind die Menüpunkte „Export Factbase“ und „Apply Change Set“ ausgegraut.

Bevor die Faktenbasis generiert wird, ist unbedingt sicherzustellen, dass MonoDevelop keine CodeCompletion Datenbank ausgelagert hat. Dazu ist im Verzeichnis jeden Projekts der Solution die „.pidb“ Datei zu löschen, falls vorhanden. Desweiteren sollte eine neu erstelle Solution unbedingt vorher geschlossen und neu geöffnet werden, da MonoDevelop andernfalls mit einem alten ProjectDom arbeiten könnte.
Wird nun der Menüpunkt „Generate Factbase“ ausgewählt, so generiert das Plugin in einem Hintergrundthread die Faktenbasis. Der Fortschritt wird dabei in der Statusleiste der IDE angezeigt. Im Verzeichnis der Solution wird eine Logdatei mit dem Namen „FactsGeneration.log“ angelegt, in der Informationen, Warnungen und Fehler, die während der Generierung aufgetreten sind, abgespeichert werden. Anschließend kann über das Kommando „Export Factbase“ die Faktenbasis exportiert werden, wie in den Abbildungen H.3 und H.4 dargestellt.

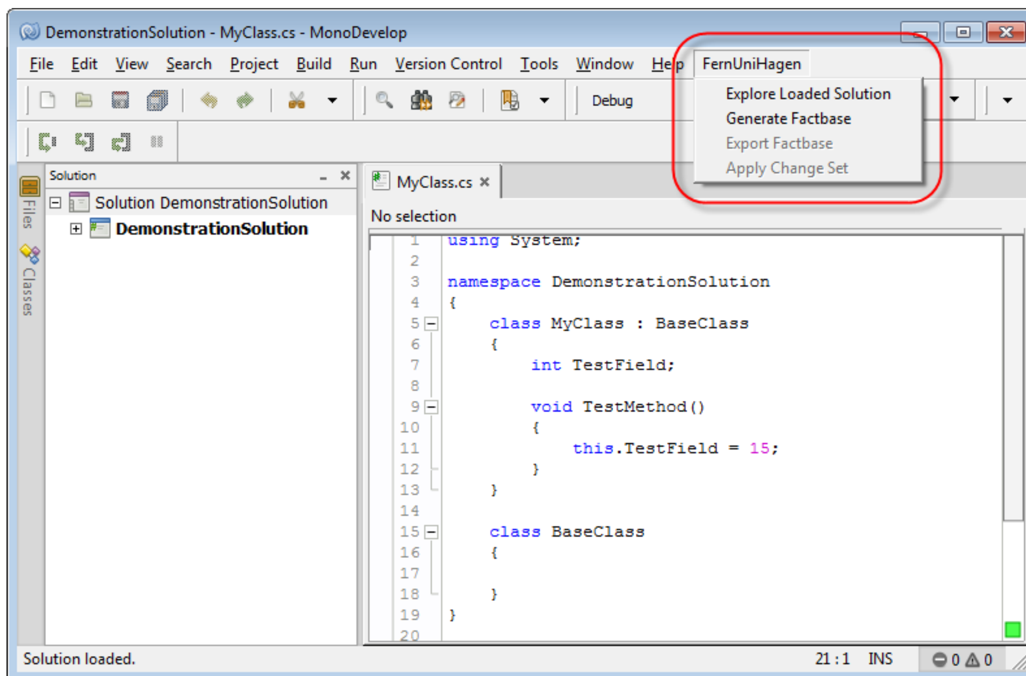


Abbildung H.2.: Menü des Plugins nachdem die Solution geöffnet wurde

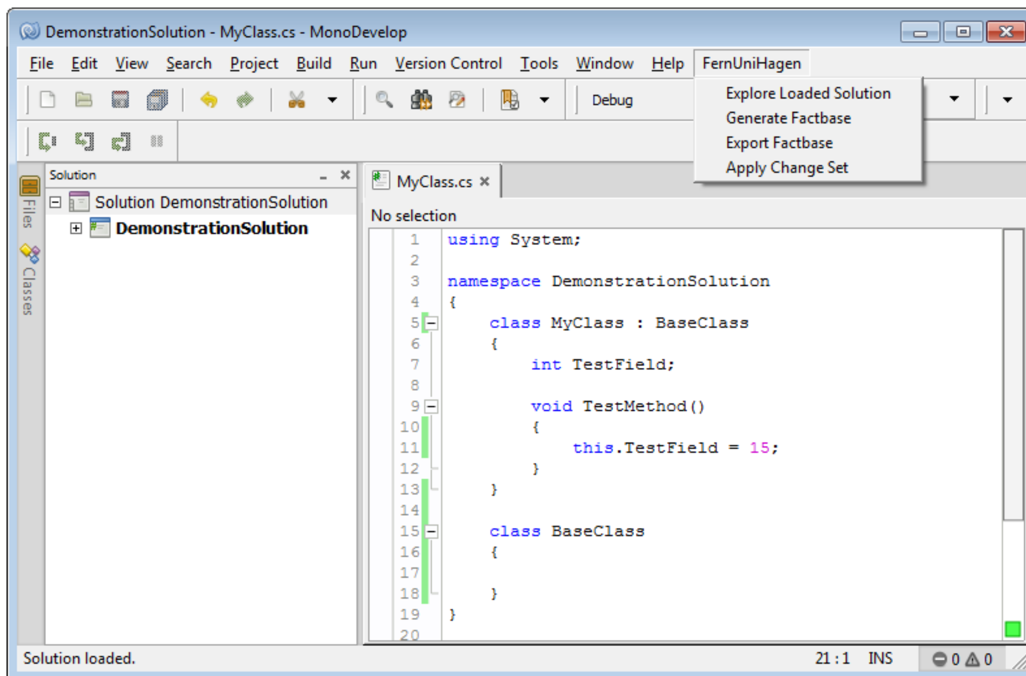


Abbildung H.3.: Menü des Plugins nach erfolgter Generierung der Faktenbasis

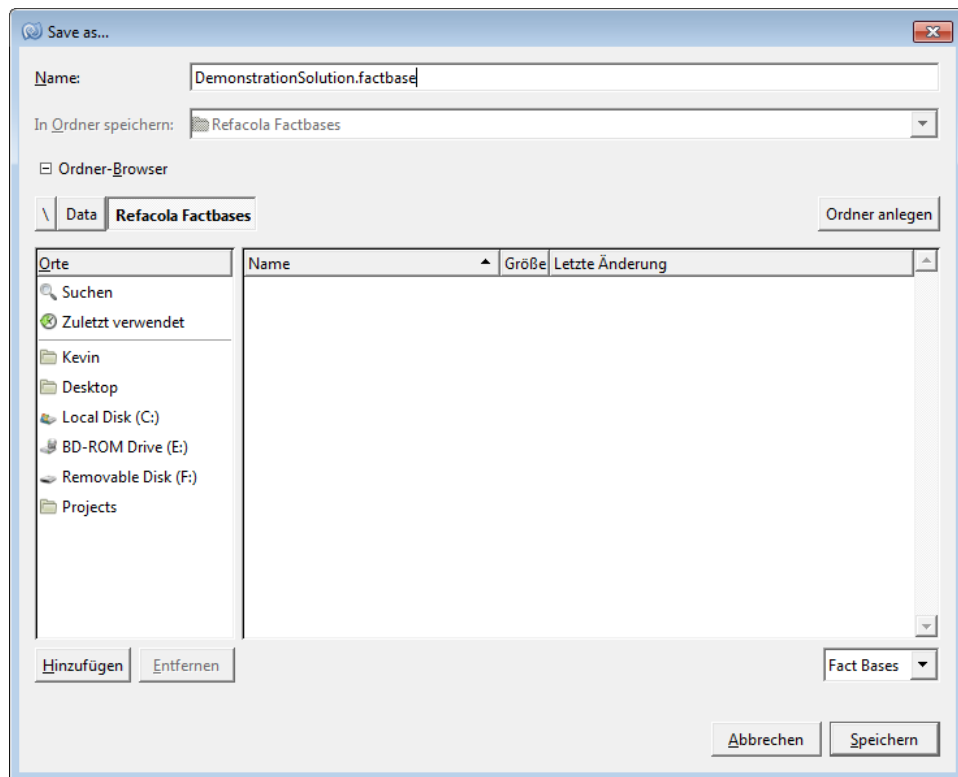


Abbildung H.4.: Export der Faktenbasis

Das Ergebnis des Exports ist in Listing H.2 dargestellt.

Listing H.2: Faktenbasis des Quellcodes aus Listing H.1

```
DemonstrationSolution {
    ExplicitThisReference ExplicitThisReference_MyClass.cs_L:11_C:4_this
        Identifier "this"
        Owner TopLevelClass_DemonstrationSolution.MyClass
        TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
        HostNamespace Namespace_DemonstrationSolution
        InferredType TopLevelClass_DemonstrationSolution.MyClass
        LocationInformation MyClass.cs_L:11_C:4_this;

    Field Field_DemonstrationSolution.MyClass.TestField
        HostNamespace Namespace_DemonstrationSolution
        TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
        Accessibility private
        Owner TopLevelClass_DemonstrationSolution.MyClass
        Identifier "TestField"
        DeclaredType TopLevelStruct_System.Int32;

    FieldReference FieldReference_MyClass.cs_L:11_C:9_TestField
        Identifier "TestField"
        Owner TopLevelClass_DemonstrationSolution.MyClass
        TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
        HostNamespace Namespace_DemonstrationSolution
        InferredType TopLevelStruct_System.Int32
        LocationInformation MyClass.cs_L:11_C:9_TestField;

    FieldReference FieldReference_MyClass.cs_L:7_C:7_TestField
        Identifier "TestField"
        Owner TopLevelClass_DemonstrationSolution.MyClass
        TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
        HostNamespace Namespace_DemonstrationSolution
        InferredType TopLevelStruct_System.Int32
        LocationInformation MyClass.cs_L:7_C:7_TestField;

    InstanceMethod InstanceMethod_DemonstrationSolution.MyClass.TestMethod
        HostNamespace Namespace_DemonstrationSolution
        TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
        Accessibility private
        Owner TopLevelClass_DemonstrationSolution.MyClass
        Identifier "TestMethod"
        DeclaredType TopLevelStruct_System.Void;

    MethodReference MethodReference_MyClass.cs_L:9_C:8_TestMethod
        Identifier "TestMethod"
        Owner TopLevelClass_DemonstrationSolution.MyClass
        TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
        HostNamespace Namespace_DemonstrationSolution
        InferredType TopLevelStruct_System.Void
        LocationInformation MyClass.cs_L:9_C:8_TestMethod;

    Namespace Namespace_DemonstrationSolution
        Identifier "DemonstrationSolution";

    Namespace Namespace_System
        Identifier "System";

    TopLevelClass TopLevelClass_DemonstrationSolution.BaseClass
        HostNamespace Namespace_DemonstrationSolution
        TopLevelOwner TopLevelClass_DemonstrationSolution.BaseClass
        Accessibility internal
```

```
60          Identifier "BaseClass";

        TopLevelClass TopLevelClass_DemonstrationSolution.MyClass
            HostNamespace Namespace_DemonstrationSolution
            TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
65          Accessibility internal
            Identifier "MyClass";

        TopLevelStruct TopLevelStruct_System.Int32
            HostNamespace Namespace_System
70          TopLevelOwner TopLevelStruct_System.Int32
            Accessibility public
            Identifier "Int32";

        TopLevelStruct TopLevelStruct_System.Void
75          HostNamespace Namespace_System
            TopLevelOwner TopLevelStruct_System.Void
            Accessibility public
            Identifier "Void";

80      TypeReference TypeReference_MyClass.cs_L:15_C:8_BaseClass
            Identifier "BaseClass"
            Owner TopLevelClass_DemonstrationSolution.BaseClass
            TopLevelOwner TopLevelClass_DemonstrationSolution.BaseClass
            HostNamespace Namespace_DemonstrationSolution
85          InferredType TopLevelClass_DemonstrationSolution.BaseClass
            LocationInformation MyClass.cs_L:15_C:8_BaseClass;

        TypeReference TypeReference_MyClass.cs_L:5_C:18_BaseClass
            Identifier "BaseClass"
90          Owner TopLevelClass_DemonstrationSolution.MyClass
            TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
            HostNamespace Namespace_DemonstrationSolution
            InferredType TopLevelClass_DemonstrationSolution.BaseClass
            LocationInformation MyClass.cs_L:5_C:18_BaseClass;
95
        TypeReference TypeReference_MyClass.cs_L:5_C:8_MyClass
            Identifier "MyClass"
            Owner TopLevelClass_DemonstrationSolution.MyClass
            TopLevelOwner TopLevelClass_DemonstrationSolution.MyClass
100         HostNamespace Namespace_DemonstrationSolution
            InferredType TopLevelClass_DemonstrationSolution.MyClass
            LocationInformation MyClass.cs_L:5_C:8_MyClass;

        binds(FieldReference_MyClass.cs_L:7_C:7_TestField,
105         Field_DemonstrationSolution.MyClass.TestField);

        binds(FieldReference_MyClass.cs_L:11_C:9_TestField,
            Field_DemonstrationSolution.MyClass.TestField);

110     binds(ExplicitThisReference_MyClass.cs_L:11_C:4_this,
            TopLevelClass_DemonstrationSolution.MyClass);

        binds(MethodReference_MyClass.cs_L:9_C:8_TestMethod,
            InstanceMethod_DemonstrationSolution.MyClass.TestMethod);
115
        binds(TypeReference_MyClass.cs_L:5_C:8_MyClass,
            TopLevelClass_DemonstrationSolution.MyClass);

        binds(TypeReference_MyClass.cs_L:5_C:18_BaseClass,
120         TopLevelClass_DemonstrationSolution.BaseClass);
```

```
        binds(TypeReference_MyClass.cs_L:15_C:8_BaseClass,
            TopLevelClass_DemonstrationSolution.BaseClass);

125     enclosed(FieldReference_MyClass.cs_L:7_C:7_TestField,
            Field_DemonstrationSolution.MyClass.TestField);

        enclosed(FieldReference_MyClass.cs_L:11_C:9_TestField,
            InstanceMethod_DemonstrationSolution.MyClass.TestMethod);
130
        enclosed(ExplicitThisReference_MyClass.cs_L:11_C:4_this,
            InstanceMethod_DemonstrationSolution.MyClass.TestMethod);

        enclosed(MethodReference_MyClass.cs_L:9_C:8_TestMethod,
135         InstanceMethod_DemonstrationSolution.MyClass.TestMethod);

        member(TopLevelClass_DemonstrationSolution.MyClass,
            Field_DemonstrationSolution.MyClass.TestField);

140     member(TopLevelClass_DemonstrationSolution.MyClass,
            InstanceMethod_DemonstrationSolution.MyClass.TestMethod);

        receiver(FieldReference_MyClass.cs_L:11_C:9_TestField,
            ExplicitThisReference_MyClass.cs_L:11_C:4_this);
145
        sub(TopLevelClass_DemonstrationSolution.BaseClass,
            TopLevelClass_DemonstrationSolution.MyClass);

        type(FieldReference_MyClass.cs_L:7_C:7_TestField,
150         TopLevelStruct_System.Int32);

        type(FieldReference_MyClass.cs_L:11_C:9_TestField,
            TopLevelStruct_System.Int32);

155     type(ExplicitThisReference_MyClass.cs_L:11_C:4_this,
            TopLevelClass_DemonstrationSolution.MyClass);

        type(MethodReference_MyClass.cs_L:9_C:8_TestMethod,
            TopLevelStruct_System.Void);
160
        type(TypeReference_MyClass.cs_L:5_C:8_MyClass,
            TopLevelClass_DemonstrationSolution.MyClass);

        type(TypeReference_MyClass.cs_L:5_C:18_BaseClass,
165         TopLevelClass_DemonstrationSolution.BaseClass);

        type(TypeReference_MyClass.cs_L:15_C:8_BaseClass,
            TopLevelClass_DemonstrationSolution.BaseClass);
    }
```

Ziel ist es nun, das Feld in die Basisklasse zu verschieben. Dazu wird ein Change Set, wie in Listing H.3, von Hand angelegt.

Listing H.3: Change Set um das Feld zu verschieben

```
1 Field_DemonstrationSolution.MyClass.TestField?Accessibility
2     -> protected
3
4 Field_DemonstrationSolution.MyClass.TestField?Owner
5     -> TopLevelClass_DemonstrationSolution.BaseClass
```

In der ersten Zeile wird angegeben, dass der Zugriffsmodifizierer auf `protected` gesetzt werden soll. Andernfalls wäre das Feld nicht mehr in der Klasse `MyClass` sichtbar. Die vierte Zeile legt fest, dass das Feld in die Klasse `BaseClass` verschoben werden soll.
Das Change Set kann schließlich über das Kommando „Apply Change Set“ eingelesen und verarbeitet werden. Informationen, Warnungen und Fehler, die während der Verarbeitung des Change Sets auftreten, werden in einer Datei mit dem Namen „ChangeSetApplication_[NameOfChangeSet].log“ im Verzeichnis der Solution abgelegt ([NameOfChangeSet] wird dabei durch den Namen der Change Set Datei ersetzt). Abb. H.5 zeigt den Dialog zur Auswahl des Change Sets.

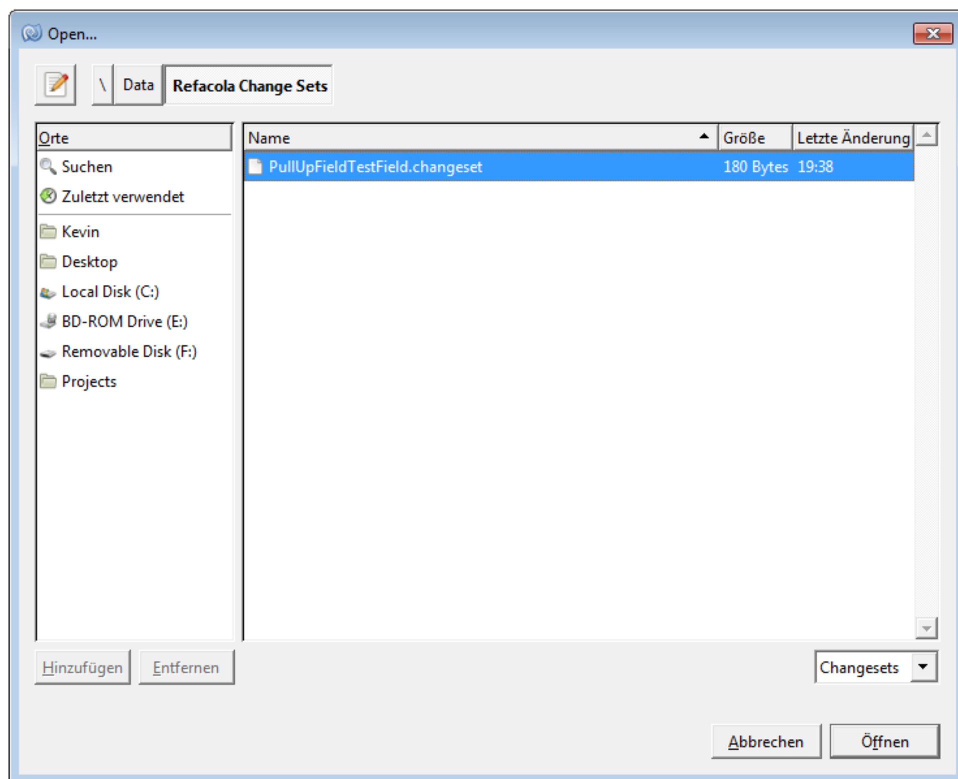


Abbildung H.5.: Auswahl eines Change Sets

Das Ergebnis der Refaktorisierung ist in Abb. H.6 dargestellt. Darauf ist zu erkennen, dass das Feld erfolgreich in die Klasse `BaseClass` verschoben wurde. Außerdem wurde der Zugriffsmodifizierer auf `protected` gesetzt.

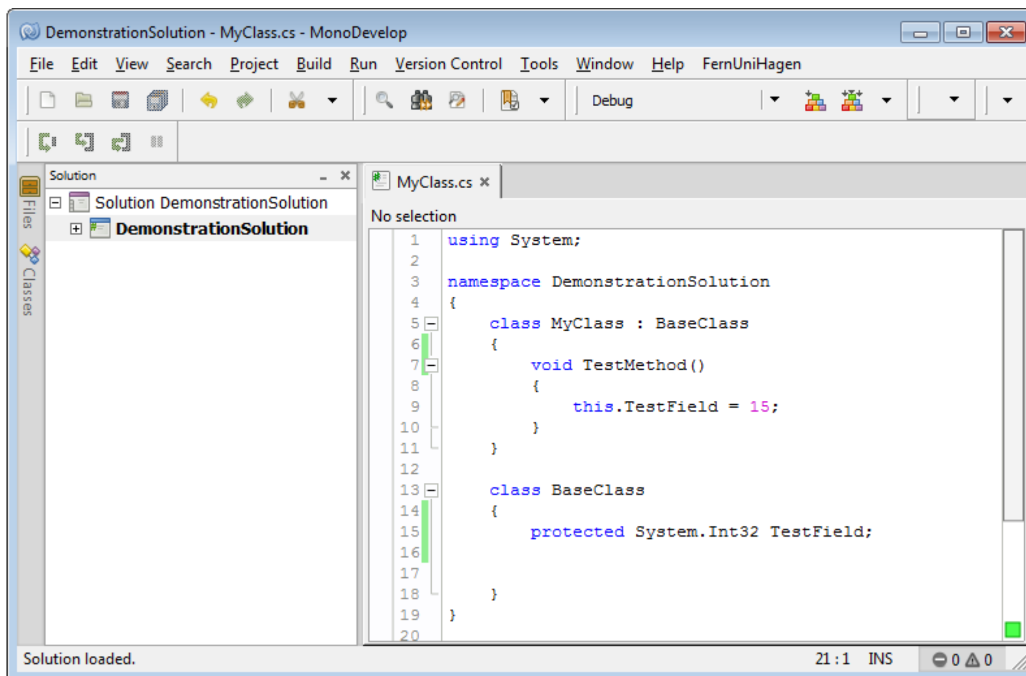


Abbildung H.6.: Ergebnis der Refaktorisierung

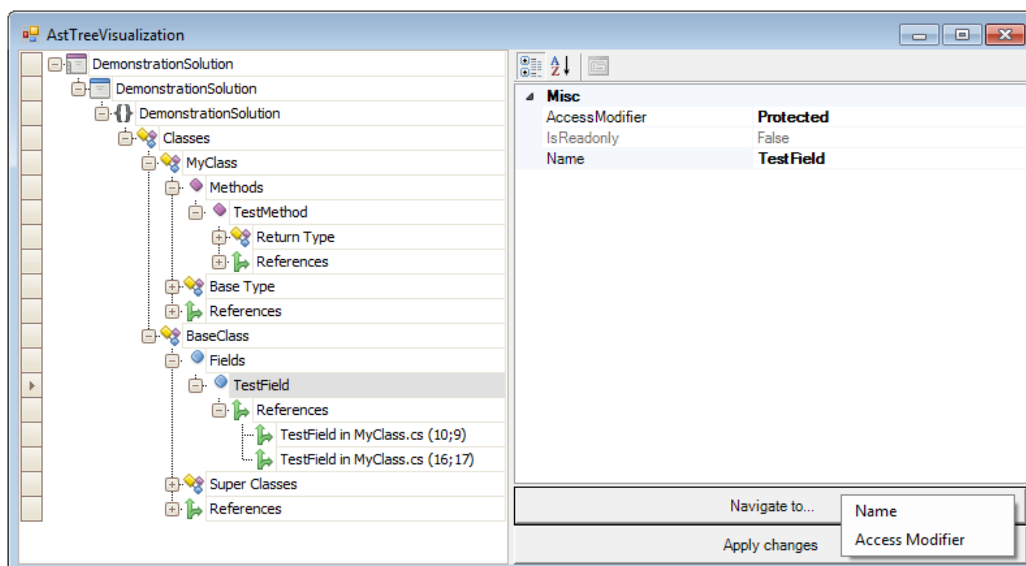


Abbildung H.7.: AST Visualisierung

Wie eingangs erwähnt, kann über den Menüpunkt „Explore Loaded Solution“ der AST der geladenen Solution visualisiert werden. In Abb. H.7 ist die Visualisierung dargestellt.

Im linken Control wird der AST mittels einer Baumstruktur visualisiert. Im rechten PropertyGrid werden verschiedene Eigenschaften des ausgewählten Knoten dargestellt. Sind Eigenschaften fett markiert, kann deren Wert geändert werden. Geänderte Eigenschaften können über den Button „Apply changes“ in den AST zurückgeschrieben werden. Über den Button „Navigate to...“ können verschiedene Komponenten des ausgewählten AST-Knotens direkt in der IDE im Quellcode selektiert werden.

H.3. Deinstallation

Das Plugin kann deinstalliert werden, indem der Ordner, der bei der Installation im Verzeichnis „AddIns“ des MonoDevelop-Installationsverzeichnisses angelegt wurde, samt aller darin enthaltenen Assemblys gelöscht wird.

I. Inhalt der beigefügten CD

- **Compiled**: Dieser Ordner enthält das Plugin und MonoDevelop in kompilierter Form. Das Plugin liegt im Unterordner **Addins\Refacola** und ist somit bereits installiert.

- **Example Files**: Dieser Ordner enthält Faktenbasen und Change Sets für die Open Source Projekte, mit denen das Plugin getestet wurde.

- **MonoDevelop Dependencies**: Dieser Ordner enthält alle Abhängigkeiten MonoDevelops, die installiert werden müssen, um MonoDevelop kompilieren zu können.

- **OpenSource Target Projects**: In diesem Ordner liegen die beiden Open Source Projekte protobuf-net[1] und Json.NET[2], mit denen das Plugin getestet wurde.

- **Source**: Dieser Ordner enthält im Unterordner **PluginDevelopment** den Quellcode des Plugins sowie die Dateien aus der MonoDevelop-Solution, die im Zuge der Entwicklungen geändert werden mussten. Im Unterordner **MonoDevelop** liegt der MonoDevelop-Quellcode. Die Solution **Main.sln** kann nur in der **x86**-Konfiguration kompiliert werden.

[1] http://code.google.com/p/protobuf-net/
[2] http://json.codeplex.com/

Abkürzungsverzeichnis

AST Abstract Syntax Tree
IDE Integrated Development Environment
DOM Document Object Model
GUID Globally unique identifier
IL Intermediate Language

Literaturverzeichnis

[Bar05] Bartak, Roman: *Constraint Propagation and Backtracking-based Search.* Charles Universität, Prag, 2005.

[Bec01] Beck, Kent: *Manifesto for Agile Software Development.* Februar 2001. URL: `http://agilemanifesto.org`, Abgerufen: 11.09.2011.

[ECM06] *ECMA-334: C# Language Specification.* ECMA (European Association for Standardizing Information and Communication Systems), Vierte Auflage, Juni 2006.

[GHJV94] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design patterns : elements of reusable object-oriented software.* Addison-Wesley, 1994.

[Gos96] Gosling, James: *The Java language specification.* Addison-Wesley, Reading, Mass, Dritte Auflage, 1996.

[Her11] Hertel, Marcel: *Portierung der RefaCoLa-API auf die .NET-Plattform.* FernUniversität Hagen, Hagen, 2011.

[Kre11] Kreis, Marius: *Systematisches Testen von Constraintregeln.* FernUniversität Hagen, Hagen, 2011.

[MSA09] *Microsoft Application Architecture Guide.* Microsoft, Redmond, Wash, Zweite Auflage, 2009.

[SKvP11] Steimann, Friedrich, Christian Kolle und Jens von Pilgrim: *A Refactoring Constraint Language and its Application.* FernUniversität Hagen, Hagen, 2011.

[ST09] Steimann, Friedrich und Andreas Thies: *From Public to Private to Absent: Refactoring JAVA Programs under Constrained Accessibility.* FernUniversität Hagen, Hagen, 2009.

[SvP11] Steimann, Friedrich und Jens von Pilgrim: *Constraint-Based Refactoring with Foresight.* FernUniversität Hagen, Hagen, 2011.

[VP11] Von Pilgrim, Jens: *Refacola. Konzepte, Frameworks, Umsetzung.* Clare Workshop, Hagen, 2011. URL: `http://www.fernuni-hagen.de/ps/prjs/refacola/RefacolaJanuar2011mitAnmerkungen.pdf`, Abgerufen: 11.09.2011.