Statische Extraktion von Programmabhängigkeiten in Microsoft .NET

Diplomarbeit

Michaela Bunke

Matrikelnummer: 1917760

25.02.2008

Universität Bremen

Fachbereich Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Dr. Berthold Hoffmann

Erklärung

Ich versichere, die Diplomarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 25.02.2008

. . . . . .

(Michaela Bunke)

Danksagung

Bedanken möchte ich mich besonders bei der Arbeitsgruppe Softwaretechnik für die Unterstützung bei der Erstellung und die gute Betreuung dieser Arbeit.

Allen weiteren Professoren und Lehrbeauftragten, sowie allen sonstigen Mitarbeitern der Universität Bremen gilt mein Dank für die gute Ausbildung, die mir dort zuteil wurde.

Der größte Dank gilt meiner Familie, die mir diese Ausbildung ermöglicht, und mir immer helfend mit Rat und Tat zur Seite gestanden hat. Auch bei meinen Freunden möchte ich mich für ihr stets offenes Ohr und Hilfe während meines Studiums herzlich bedanken.

INHALTSVERZEICHNIS

1 Einleitung **1**

1.1 Das Bauhaus-Projekt 2

1.2 Aufgabenstellung 3

1.3 Aufbau der Arbeit 5

2 Grundlagen **7**

2.1 Common Language Infrastructure 8

2.2 .NET 14

2.2.1 Das .NET-Framework 15

2.2.2 Open Source .NET-Projekte 17

2.3 C# 19

2.3.1 Programmaufbau 20

2.3.2 Namensräume 21

2.3.3 Zugriffsmodifizierer 21

2.3.4 Klassen, Strukturen und Schnittstellen 22

2.3.5 Vererbung 23

2.3.6 Delegaten 25

2.3.7 Ereignisse 26

2.3.8 Generics 28

2.3.9 Propertys und Indexer 29

2.3.10 Extern und DLLImport 31

2.4 Metamodelle zur Programmrepräsentation 33

2.4.1 RFG 33

2.4.2 FAMIX 36

2.4.3 Dagstuhl Middle Metamodel 38

3 Konzeption eines RFG-Schemas für C#-Programme **39**

3.1 Existierende RFG-Schemata 40
3.2 RFG-Schema für C# 41
3.2.1 Namespaces 41
3.2.2 Typen 42
3.2.3 Klassen und Schnittstellen 42
3.2.4 Klassenvariablen 44
3.2.5 Methoden 45
3.2.6 Propertys und Indexer 47
3.2.7 Events 48
3.2.8 Generics 50
3.2.9 Das vollständige RFG-Schema für C# 51
3.3 Anforderungen an die Analyse von C# 52

4 Analyse **55**
4.1 Assemblys als spezielle Form der *Portable Executable Images* 56
4.2 Analyse eines Assembly (*Portable Executable Image* mit CLI-Erweiterung) . . 58
4.2.1 File Header 58
4.2.2 CLI Header 60
4.2.3 CLI Data 61
4.3 Analyse der Metadaten Tabellen 64
4.3.1 Dekodierung von Informationen innerhalb der Metadaten Tabellen . . 65
4.3.2 Beschreibung der Metadaten Tabellen 68
4.3.3 Analyse des CIL-Codes 83
4.4 Gewinnung von Quelltextpositionen 86
4.4.1 Quelltextpositionsgewinnung mit Debuginformationen 86

5 Generierung des RFG **89**
5.1 Vorgehen bei der RFG-Generierung 90
5.1.1 Umsetzung der Informationen der Metadaten 90
5.1.2 Laufzeitinformationen aus den Methoden 91
5.2 Sichten 92
5.3 Linken 92

6 Test **95**

6.1 Test der Informationsextraktion 95
6.2 Test der RFG-Erzeugung 96

7 Fazit **99**
7.1 Überblick 99
7.2 Schwachstellen 99
7.3 Weiterentwicklung 100

A File Header eines Portable Executable Images **101**

B Metadatentabellen **105**

C DTD zur XML-Datei zum Speichern von Quelltextinformationen **107**

Abbildungsverzeichnis **110**

Tabellenverzeichnis **111**

Listings **113**

Literaturverzeichnis **118**

KAPITEL 1

Einleitung

“ . . . Visual Studio .NET and the .NET Framework will enable the next big wave of developer opportunity, creating XML Web services that will soon become the basis for all major new software development.“ Bill Gates (2002) [Mic02]

Fünf Jahre nach dem ersten Release des .NET-Framework, ist dieses in der nächsten Generation der Windows-Betriebssysteme[1] fest verankert und somit nicht mehr nur noch ein „kann“, sondern schon fast ein „muss“ für die Windows-Applikationsentwicklung geworden.

.NET ist ein Sammelbegriff für eine Softwareentwicklungsplattform der Firma Microsoft. Das dahinter stehende Konzept sollte ein Schritt in die Zukunft für die Windows-Applikationsentwicklung sein, welches die bis dato vorhandenen Probleme und Einschränkungen tiefgreifend verbessern sollte. Das im Jahre 2002 veröffentlichte Ergebnis ist eine einheitliche, objektorientierte Plattform mit Laufzeitumgebung.

Die Laufzeitumgebung (*Virtual Execution System*) ermöglicht plattformunabhängiges Entwickeln. Sie ist in einem öffentlichen ECMA/ISO Standard ([ECM06b, ISO06]) der *Common Language Infrastructure* (CLI) definiert und zeichnet sich unter anderem durch eine Sprachspezifikation (*Common Language Specification*), sowie einem einheitlichen Typsystem (*Common Type System*) aus. Beides ermöglicht das Entwickeln von Programmkomponenten in verschiedenen Sprachen und somit die Verwendung von Fertigkomponenten. Das .NET-Framework ist eine konkrete Implementierung des CLI-Standards und bildet die Basis von .NET.

Im Rahmen der .NET-Framework Entwicklung wurde von Microsoft die Programmiersprache C# entworfen und bildet die meisten CLI-spezifischen Besonderheiten ab. Darüber hinaus wurden Sprachderivate von Java (J#) und C++ (C++/CLI) für das .NET-Framework entwickelt. Sie sollten den Umstieg auf die neue Plattform erleichtern. Ebenso wurde die bis zu diesem Zeitpunkt am häufigsten verwendete Sprache zur Windows-Applikationsentwicklung, Visual Basic, komplett erneuert und an die Gegebenheiten des .NET-Frameworks angepasst.

Durch diesen radikalen Wechsel der Softwareentwicklungsstrategie von der plattformabhängigen hin zur plattformunabhängigen abstrakteren Entwicklung für

[1] Windows Vista und Windows Server 2008 [Mic06a]

Windows-Plattformen ergeben sich zwei interessante Szenarien. Im ersten Szenario werden neu geplante Softwaresystem für Windows-Betriebssysteme gleich für die .NET-Welt entwickelt und müssen in ihrem weiteren Lebenszyklus gewartet oder auch erweitert werden. Im zweiten Szenario werden ältere Programme die beispielsweise mit Visual Basic 6 geschrieben wurden aufgrund des auslaufenden Supportes[2] teilweise oder gänzlich auf die neue Plattform portiert. In beiden Szenarien entsteht irgendwann der Bedarf nach Analyse des bestehenden Programms aufgrund von Wartungsarbeiten oder Erweiterungen an der Software. An diesem Punkt setzt das Bauhausprojekt an.

1.1 Das Bauhaus-Projekt

Das Bauhaus-Projekt [RVP] ist ein Forschungsprojekt der Arbeitsgruppe Softwaretechnik der Universität Bremen[3] und der Abteilung Programmiersprachen und Übersetzer des Instituts für Softwaretechnologie der Universität Stuttgart[4]. Seit 2005 existiert zusätzlich die Firma Axivion GmbH[5], die die Bauhaus-Software kommerziell unter dem Namen Axivion Bauhaus Suite vertreibt. Innerhalb diese Projektes ist diese Diplomarbeit entstanden.

Ziel des Projektes ist es Wartungsprogrammierer bei der Analyse bestehender Softwaresysteme zu unterstützen und somit die Wartung zu erleichtern. Wartungsprogrammierer werden häufig damit konfrontiert ein ihnen unbekanntes Softwaresystem anpassen zu müssen. Meist geht dies einher mit wenig Dokumentation zum vorliegenden System und nicht verfügbaren Ansprechpartnern, die das System einmal entwickelt haben. Diese Voraussetzungen erschweren erheblich die Arbeiten an der Software, was sich in einem hohen Zeit- und Kostenaufwand wiederspiegelt. An diesem Punkt setzt das Bauhaus-Projekt an. Mit Hilfe der Bauhaus-Software können auf Grundlage des Quelltextes verschiedene Architektursichten rekonstruiert und visualisiert sowie Messungen mittels Metriken und Analysen durchgeführt werden.

Um Analysen durchführen zu können, wird der Quellcode in eine Zwischensprache überführt. Damit nicht für jede Programmiersprache eine eigene Zwischensprache entworfen werden muss, sind die in Bauhaus vorherrschenden Zwischensprachen ***I**nter**M**ediate **L**anguage* (IML) und ***R**esource **F**low **G**raph*(RFG) relativ abstrakt gehalten.

Die IML ist eine Repräsentationsform, die Softwaresysteme sehr nah an der Quelltextform abbildet. Hierbei werden fast alle Sprachelemente des Quelltextes in den abstrakten Syntaxbaum übertragen. In dieser Darstellungsform wird z.B. der Kontroll- und Datenfluss eines Programms nachgebildet. Sie dient vornehmlich dazu, einheitliche Analyse auf einer technisch sehr detaillierten Ebene durchzuführen.

[2] http://support.microsoft.com/lifecycle/
[3] http://www.informatik.uni-bremen.de/st/
[4] http://www.iste.uni-stuttgart.de/ps/
[5] http://www.axivion.com

Die zweite Repräsentationsform, der RFG, enthält ausschließlich die „gröberen“, technisch weniger detaillierten Informationen. Es werden ausschließlich architekturrelevante Elemente und deren Beziehungen abgebildet. Der RFG kann mit dem Visualisierungsprogramm *Gravis* dargestellt werden. Neben dem Programm an sich können auch Analysen visualisiert sowie detaillierte Informationen zu bestimmten Programmteilen dargestellt werden.

Für die Generierung der Repräsentationsformen gibt es verschiedene Tools zur Auswahl. Diese so genannten *Frontends* sind sprachspezifisch und erzeugen meist nur eine Repräsentationsform aus dem gegebene Quelltext. Eine Ausnahme bildet hier das Tool *java2rfg*; Es erzeugt einen RFG aus Java-Bytecode. Die zur Zeit verfügbaren Frontends können C, C++, Java und Ada Programme in die bauhauseigenen Formate überführen. Zusätzlich gibt es die Möglichkeit mit *iml2rfg* eine IML in einen RFG zu konvertieren.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist es, die mit dem .NET-Framework zusammen entwickelte Sprache C# in einem Resource Flow Graph (RFG) abzubilden. Ähnlich zum *java2rfg* Tool, das innerhalb des Bauhausprojektes aus Java-Bytecode einen RFG generiert, werden für die Analyse Kompilate der Sprache C# verwendet.

Dieses zu erstellende Analysewerkzeug soll eine plattformunabhängige Analyse der Kompilate ermöglichen. Aus diesem Grund wird nicht die Reflections[6]-Bibliothek des .NET-Frameworks benutzt, sondern die CLI-Informationen innerhalb eines Kompilates verwendet. Diese Kompilate enthalten die im Quelltext deklarierten Symbole sowie ihre Beziehungen. Die komplexe Struktur in der die Programmsymbole innerhalb eines Kompilates vorgehalten werden gestaltet die Analyse sehr aufwändig.

Problematisch ist das Fehlen von Zeilen- und Dateiinformationen zu den Symbolen innerhalb des Kompilates. Dadurch existiert keine Beziehung zwischen Symbolen innerhalb des Kompilates und der korrespondieren Quelltextdeklaration. Diese Beziehung wäre jedoch wünschenswert, damit im Visualisierungsprogramm des RFG zu einem Objekt der Quelltext angezeigt werden kann. Für die Abbildung von Quelltextinformationen auf die Elemente innerhalb des Kompilates soll in dieser Arbeit eine Lösung erarbeitet werden.

Im Rahmen dieser Arbeit soll ein Analysewerkzeug entstehen, das die in C#-Kompilaten enthaltene Programmdekomposition in einem RFG abbildet. Dazu ist es notwendig ein RFG-Schema zu entwickeln, das die Sprachbegebenheiten von C# abbildet. Damit im Anschluss das Kompilat hinsichtlich der benötigten Informationen untersucht werden kann.

[6] Reflections bieten die Möglichkeit während der Laufzeit Informationen zu geladenen Assemblys und ihren Objekten wie z.B. Klassen zu bekommen `http://msdn2.microsoft.com/en-us/library/cxz4wk15.aspx`

Rahmenbedingungen

Durch die Entwicklung im Rahmen des Bauhaus-Projektes ergeben sich spezifische Rahmenbedingungen für die Umsetzung der Arbeit. Die Arbeit soll in Ada, der vorherrschenden Programmiersprache des Bauhaus-Projektes, implementiert werden, damit ein reibungsloser Zugriff auf die Bauhaus-Bibliotheken gewährleistet ist. Die Entwicklung wird auf einem Linux-System stattfinden, da die Übersetzung der Ada-Quelltexte mit dem Ada-Complier nur in einer Linuxumgebung ohne weitere Anpassungen durchgeführt werden kann.

Das .NET-Framework ist die älteste und ausgeprägteste CLI-Implementierung. Deswegen wird sie zur Erzeugung von Testkompilaten verwendet. Allerdings werde ich mich auf die .NET-Frameworkversionen 2.0 und 3.0 beschränken, denn der Support für die Version 1.0 läuft im Jahr 2008 aus[7].

Durch die Verwendung des properitären .NET-Frameworks ergeben sich zusätzliche Bedingungen für diese Arbeit. Die meisten Programme, die auf Basis des .NET-Frameworks entwickelt werden, werden mit der Entwicklungsumgebung Visual Studio entwickelt. Das Visual Studio enthält eigene Compiler für die verfügbaren Sprachen. Aus diesem Grund wird zur Erzeugung der Testkompilate der Visual Studio Compiler für C# verwendet.

Testprogramme

Um sicher zu stellen, dass die Sprache C# möglichst vollständig im RFG abgebildet wird werden verschiedene Teststufen vorbereitet. Angelehnt an die Microsoft Programmierbeispiele aus dem *C# Programming Guide*([Mic05a]) und der *C# Reference*([Mic05b]) werden zuerst einzelne Objekt und Zusammenhänge in kleinen Testprogrammen getestet. Danach werden etwas komplexere Programm-Beispiele aus dem *Microsoft Developer Network - Visual C# Developer Center* die *Visual C#* Beispiele getestet, die spezielle Sprachelement behandeln. Abschließend wird das entwickelte Werkzeug mit Programmen von Open Source Projekten getestet.

Ziele der Arbeit

Die sich aus Aufgabenstellung ergebenden konkreten Ziele sind:

- **Abbildung von C# auf den RFG**
  Damit ein RFG zur Sprache C# generiert werden kann, muss zuvor ein RFG-Modell entworfen werden, dass die Sprachkonstrukte abbildet. Dieses muss in einer Form geschehen, in der die vorhandenen Konstrukte anderer schon existierender RFG-Schemata wiederverwendet und ggf. nur leichte Veränderungen vorzunehmensind, wenn in C# Sprachkonzepte vorhanden sind, die sich mit den

[7] http://support.microsoft.com/gp/lifedevtool

bestehenden Modellen nicht abbilden lassen.

- **Analyse von C#-Kompilaten**
  Um die für den RFG benötigten Daten erheben zu können, muss eine Analyse der CLI-spezifischen Informationen innerhalb eines Kompilates durchgeführt werden.

- **Gewinnung von Quelltextpositionen**
  Die CLI enthält keine Quelltextinformationen wie Zeile, Spalte oder Datei in der ein Objekt definiert wurde. Diese Informationen müssen auf einem anderen Weg gewonnen werden, damit der Benutzer mit dem in *Gravis* integrierten Quellcode-Browser den Quelltext zu einem Objekt ansehen kann.

- **Generierung des Resource Flow Graphs**
  Im Rahmen dieser Arbeit muss ein Programm entwickelt werden, das auf Grundlage der Analyse der CLI-Informationen einen RFG generiert.

1.3 Aufbau der Arbeit

Diese Arbeit gliedert sich in insgesamt sieben Kapitel auf. Nach der Einleitung wird das nächste Kapitel eine kleine Einführung in den Aufbau der CLI und seine Implementierungen wie das .NET-Framework geben. Außerdem wird die Sprach C# mit ihren, in dieser Arbeit verwendeten, Programmierkonstrukten vorgestellt. Im Anschluss wird der Resource Flow Graph sowie ähnliche Repräsentationsformen vorgestellt.

Das dritte Kapitel behandelt die Erstellung eines Sprachschemas, das C# auf einen RFG abbildet. Im vierten Kapitel wird der Aufbau eines C#-Kompilates vorgestellt und auf die für die RFG-Erstellung benötigten Informationen hin analysiert.

Nach der Analyse der Informationen behandelt das fünfte Kapitel die Generierung des Resource Flow Graphs auf Basis der gewonnenen Daten. Das sechste Kapitel beschreibt die Validierung des generierten RFG. Das siebte und letzte Kapitel fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick welche Möglichkeiten sich in Folge dieser Arbeit eröffnen.

KAPITEL 2

Grundlagen

In diesem Kapitel beschreibe ich die Grundlagen, die für diese Arbeit benötigt werden. Es soll einen Überblick über die für diese Arbeit relevanten Arbeitsgegenstände geben.

Zuerst wird das Konzept der *Common Language Infrastructure* (CLI) vorstellt. Dies bildet die Basis für das Microsoft .NET-Framework, welches im darauf folgenden Abschnitt unter anderem als konkrete Implementierung der *Common Language Infrastructure* beschrieben wird.

Danach wird auf die Sprache C# und ihre Besonderheiten eingegangen. C# ist der Ausgangspunkt der Analyse von .NET-Programmen in Bytecode-Form, da es die meisten Konstrukte der CLI-Zwischendarstellung verwendet. Schließlich werde ich das in dieser Arbeit verwendete RFG-Modell vorstellen sowie ähnliche Metamodelle kurz beschreiben und mit dem RFG-Modell vergleichen.

Kapitelinhalt

2.1 Common Language Infrastructure 8

2.2 .NET 14

2.2.1 Das .NET-Framework 15

2.2.2 Open Source .NET-Projekte 17

2.3 C# 19

2.3.1 Programmaufbau 20

2.3.2 Namensräume 21

2.3.3 Zugriffsmodifizierer 21

2.3.4 Klassen, Strukturen und Schnittstellen 22

2.3.5 Vererbung 23

2.3.6 Delegaten 25

2.3.7 Ereignisse 26

2.3.8 Generics 28

2.3.9 Propertys und Indexer 29

2.3.10 Extern und DLLImport 31

2.4 Metamodelle zur Programmrepräsentation 33

2.4.1 RFG 33

2.4.2 FAMIX 36
2.4.3 Dagstuhl Middle Metamodel 38

2.1 Common Language Infrastructure

Die Basis des .NET-Frameworks bildet die *Common Language Infrastructure* (CLI). Sie ist ein ISO[1] und ECMA[2] spezifizierter Standard (siehe dazu [ECM06b, ISO06]), der aus der Initiative der Microsoft Corporation in Zusammenarbeit mehrerer großer Firmen wie Intel Corporation, Hewlett-Packard und IBM Corporation entstanden ist.

Microsofts Intention mit diesem Konzept war, die Programmierung für Windows-Betriebssysteme zu revolutionieren. Es sollte viele derzeitige Probleme in der „Windows Welt“ verbessern und sich an den aktuellen Softwarekonzepten orientieren. Aus diesem Grund wurde bei der Konzipierung der CLI großer Wert auf die folgenden Eigenschaften gelegt:

- Interoperabilität
- Plattformunabhängigkeit
- Sprachunabhängigkeit
- einheitliche Schnittstelle zur Anwendungsprogrammierung (API)
- definierter Sprachstandard
- Code-Sicherheit
- Beseitigung der „DLL-Hölle“ (siehe dazu 2.1)

Herausgekommen ist ein öffentlicher Standard, der teilweise an das Javakonzept erinnert aber durchaus interessante neue Konzeptansätze verfolgt.

Die entstandene *Common Language Infrastructure* definiert Regeln und Komponenten, die es einer Laufzeitumgebung ermöglicht Programmcode, der in unterschiedlichen Programmiersprachen geschrieben wurde, auszuführen. Der Standard beschreibt somit die grundlegend wichtigen Aspekte des .NET-Frameworks: Das *Common Type System*, *Virtual Execution System*, die *Common Language Specification*, *Base Class Library* und den Aufbau eines Dateiformates (Assembly[3]), die vom *Virtual Execution System* benutzt werden.

[1] International Organization for Standardization
[2] European Computer Manufactures Association
[3] Ein Assembly ist eine Datei, die kompilierten Quelltext sowie weitere Informationen enthält. Sie können auch als fertige Softwarekomponente verstanden werden ([Ric06] S.30/S.70).

Common Type System (CTS)

Programmiersprachen haben in der Regel jeweils eigene Datentypen und somit ein eigenes Typsystem. Dies verhindert einen einfachen Austausch von Daten zwischen Programmen, die in verschiedenen Programmiersprachen geschrieben wurden. Um diesem Problem entgegenzuwirken wurde in der CLI ein allgemeines Typsystem (Common Type System) für CLI-kompatible Programme festgelegt.

CTS beschreibt den Aufbau einer Grundstruktur, die die sprachübergreifende Integration und Typsicherheit ermöglicht. Zudem wird ein objektorientiertes Modell bereitgestellt, das die Abbildung zahlreicher Programmiersprachen, egal ob objektorientiert, funktional oder prozedural, für die CLI unterstützt. Datentypen die innerhalb des Quelltextes deklariert werden, werden anhand der CTS und den im nächsten Abschnitt beschrieben *Common Language Specification* Regeln als Metadaten innerhalb eines Assembly gespeichert.

Zusammengefasst definiert die CTS wie Typen zur Laufzeit deklariert und verwaltet werden, so dass eine gemeinsame Nutzung von Bibliotheken und anderen Programmkomponenten sprachübergreifend möglich wird.

Common Language Specification (CLS)

Die *Common Language Specification* ist eine Zusammenstellung von Regeln, die die Interoperabilität von Programmiersprachen fördern soll. Sie ist eine Teilmenge der CTS.

Im ECMA-Standard umfasst die CLS insgesamt 47 Regeln, die eingehalten werden müssen, damit Programmcode in einer CLI-Implementierung ausgeführt werden kann. Hierbei ist es zwingend notwendig, dass Teile, die außerhalb eines Assembly sichtbar sind, diese Spezifikation einhalten, damit die Interoperabilität zwischen verschiedensprachigen Programmkomponenten gewährleistet werden kann.

Common Intermediate Language (CIL)

Die *Common Intermediate Language* ist eine Art Zwischencode ähnlich zum Java-Bytecode, jedoch enthält er nur die Anweisung die innerhalb von Methoden deklariert wurden.

Vor der Standardisierung hieß CIL auch MSIL (*Microsoft Intermediate Language*). Sie besteht aus 216 assemblerähnlichen Befehlen, die von der VES mit einem JIT (*Just in Time Compiler*) in Maschinencode übersetzt und ausgeführt wird.

Der geschriebene Quelltext wird dazu vom Entwurf bis zur Ausführung zweimal kompiliert. Dies veranschaulicht die Abbildung 2.1. Der Quelltext wird mit einem CLI-konformen Compiler in eine Bytecode-Form gebracht. Die dann mit Hilfe des JIT in eine plattformspezifische Form gebracht und ausgeführt wird. Mit diesem Konzept ist es möglich, CIL-Code ohne Veränderungen auf CLI-Implementierungen für unterschiedliche Hardware- und Betriebssystemplattformen laufen zu lassen.

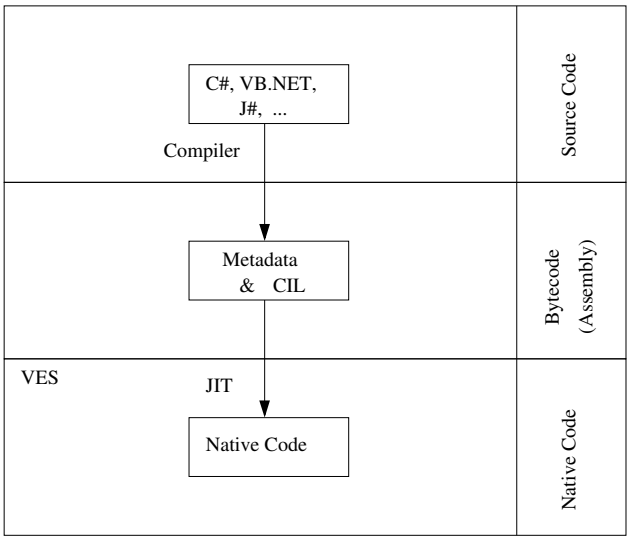

Abbildung 2.1: CLI-Konzept vom Quelltext bis zur Ausführung eines Programms

Virtual Execution System (VES)

Das *Virtual Execution System* ist für das Laden und die Ausführung von Programmcode zuständig. Es spezifiziert eine virtuelle Maschine zur Ausführung CLI-konformer Kompilate. Diese virtuelle Maschine wurde zusätzlich mit einer Programmablaufsteuerung und einem Fehlerbehandlungsmodell spezifiziert. Sie verwendet die Metadaten und den CIL-Code der aus dem Quelltext erzeugt wurde und in einem Assembly vorliegt.

Klassenbibliotheken und Profile

Zu den bisher allgemein beschriebenen Funktionalitäten, die eine CLI-Implementierung aufweisen soll, kommt ein Mindestmaß an Bibliotheken hinzu. Sie werden im ECMA-Standard 335 in obligatorische und optionale Profile und Bibliotheken unterteilt, die in Abbildung 2.2 dargestellt sind.

Eine Bibliothek ist hierbei eine konkrete Sammlung von Typen mit bestimmten Funktionen, die in einem Assembly zusammengefasst werden. Im Gegensatz dazu ist ein Profil eine Zusammenfassung mehrerer Bibliotheken, die als Einheit ein fest definiertes Funktionalitätsmaß definiert.

Eine konforme CLI-Implementierung muss als kleinsten gemeinsamen Nenner aller CLI-Implementierungen das *Kernel Profile* beinhalten. Dieses Profil besteht aus der *Runtime Infrastructure Library* und der *Base Class Library*. Die *Runtime Infrastructure Library* stellt Compiler-Dienste wie das Laden von CLI-konformen Dateiformaten zur Verfügung, während die *Base Class Library* unter anderem Funktionen zur Ein- und Ausgabe sowie die vordefinierten Datentypen der CLI enthält.

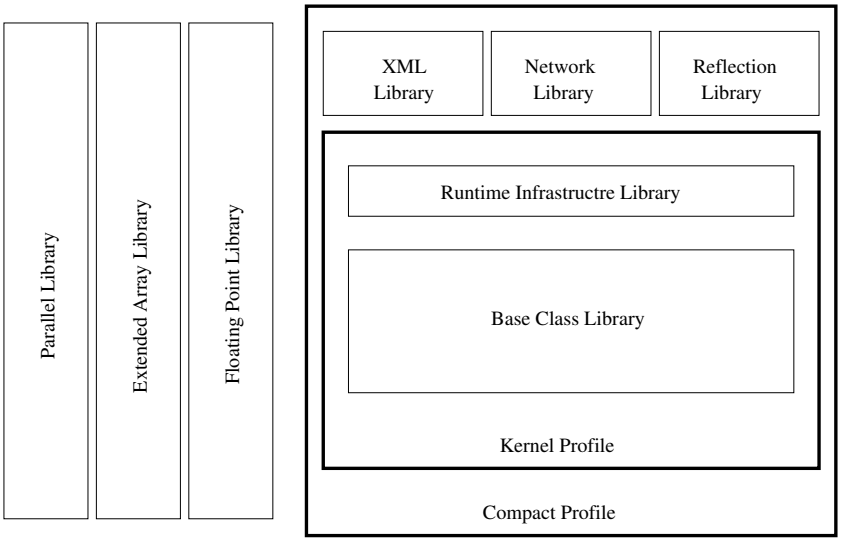

Abbildung 2.2: Beziehung zwischen Bibliotheken und Profilen in der CLI ([ECM06b], Partition IV, S.3)

Das *Compact Profile* umfasst das *Kernel Profile* und erweitert es um XML[4]-, Netzwerk- und eine *Reflection*-Bibliothek. Die XML-Bibliothek stellt Klassen zur XML-Verarbeitung bereit. Dies beinhaltet das Auslesen, die Manipulation und die Transformation von XML-Dateien. Die Netzwerk-Bibliothek enthält Klassen, die einfache Netzwerkfunktionalitäten bereitstellen. Mit Hilfe der *Reflection*-Bibliothek können unter anderem Datentypen eines geladenen Objektes instanziiert und untersucht werden.

Die übrigen drei Bibliotheken sind optionale Bibliotheken. Sie müssen nicht zwingend in einer CLI-Implementierung vorhanden sein beschreiben aber durchaus interessante Eigenschaften.

- Die *Floating Point Library* stellt die Gleitkomma-Datentypen *Single*, *Double* und *Decimal*, sowie deren Manipulationsfunktionen bereit.
- Die *Extended Array Library* ergänzt die in der *Base Class Library* definierten Arrays um die Funktionalität mehrdimensionaler Arrays sowie Arrays mit einer kleineren unteren Grenze als Null.
- Die *Parallel Library* bietet einfache Möglichkeiten zur gleichzeitigen oder nebenläufigen Ausführung von Programmteilen, so dass CPUs besser ausgenutzt werden können.

Assemblys

Zur neuen Softwareentwicklungsinfrastruktur ist noch das Konzept der .NET Assemblys hinzugekommen. Assemblys sollen das bisherige *Component Object Model* (COM) ablösen.

[4] XML steht für Extensible Markup Language

Auf die bisherige COM-Problematik und das Assembly-Konzept werde ich im Folgenden näher eingehen.

Soweit nicht anders ausgezeichnet, wurden folgende Quellen für diesen Abschnitt verwendet [And00, Pie00, Ric06].

Bis zur Einführung des .NET-Frameworks basierte das Softwarekomponentenmodell für Windows-Betriebssysteme auf dem COM-Konzept. COM-Objekte sind sprachneutrale Bibliotheken, die die Wiederverwendung von fertigen Softwareteilen ermöglichen. Sie werden in einem zentralen Platz im Windowssystem, der *Windows-Registry*, installiert und anschließend registriert. Durch diese zentrale Speicherung kann immer nur eine bestimmte Version eines COM-Objektes dort abgelegt und verwaltet werden. Dieses Konzept funktioniert solange, bis zwei Anwendungen verschiedene Versionen eines bestimmten COM-Objektes benutzen müssen oder ein älteres COM-Objekt durch eine neuere Version ersetzt wird. Dadurch können Fehler in den Programmen entstehen, die mit dem alten COM-Objekt arbeiten. Unter Umständen können diese Programme dann nicht mehr genutzt werden, da sie schlichtweg nicht mehr lauffähig sind. Diese prekäre Situation wird auch als „DLL-Hölle“[5] bezeichnet.

Assemblys sollen die Entwicklung vereinfachen und eine Lösung für das „DLL-Höllen“-Problem von Windows bieten. So bestehen COM-Objekte aus einer kompilierten Code-Datei. Diese Dateien haben einen speziellen Aufbau der *Portable Executable* Format[6] genannt wird. Assemblys unterscheiden sich von COM-Objekten darin, dass sie eine Zwischendarstellung enthalten. In dieser Zwischendarstellung, auch Metadaten genannt, sind alle deklarierten Objekte sowie die Anweisung innerhalb der Methoden des ursprünglichen Quelltextes enthalten. Es fehlen jedoch die Quelltextpositionen der deklarierten Symbole, da diese innerhalb der VES zum Ausführen des Programms nicht benötigt werden. Auf die enthaltenen Objekte und ihre konkrete Darstellung innerhalb der Metadaten werde ich in Kapitel 4 eingehen.

Diese Zwischendarstellung bildet den Ausgangspunkt der Analyse von .NET-Programmen in dieser Arbeit.

Es gibt zwei Arten von Assemblys, einmal Prozess-Assemblys mit der Endung „EXE“ und Bibliotheken-Assemblys mit der Endung „DLL“. Ein Assembly besteht aus einem oder mehreren Softwaremodulen, die den schon vorgestellten CIL-Code und Metadaten enthalten. Ein Compiler erzeugt jeweils ein Modul-Assembly. Damit mehrere Module zu einem Assembly verbunden werden können müssen die vorher kompilierten Softwaremodule mit dem Programm *AL.exe*[7] zusammengeführt werden. In dieser Arbeit wird auf der Basis von Assemblys bestehend aus einem Softwaremodul gearbeitet, da diese vollständig im ECMA Standard 335 beschrieben sind.

[5] siehe dazu [And00, Pie00, Ric06]

[6] Das *Portable Executable* Format beschreibt den Ausbau einer ausführbaren Datei in seiner Bytecode-Darstellung.

[7] `http://msdn2.microsoft.com/en-us/library/c405shex(VS.80).aspx`

Zusätzlich können in Assemblys Informationen, die keinen Programmcode enthalten, eingebunden werden. Diese *Ressourcen* können z.B. JPEG- oder XML-Dateien sein, die dann in den Assembly-Bytecode eingeflochten werden. Jedes Assembly hat einen Einstiegspunkt, der die Startmethode des Programms kennzeichnet. Der Aufbau und die spezifischen Informationen eines Assemblys werden später im Kapitel 4 beschrieben.

Anders als bei COM-Objekten existieren zwei Bereitstellungsarten für Assemblys: privat oder global. Bei der privaten Bereitstellung wird das referenzierte Assembly in einem Unterordner des Programms abgelegt und kann nur von Programmen verwendet werden, die explizit wissen, wo sich dieses Assembly befindet. Für die globale Bereitstellung von Komponenten wird das Assembly im *Global Assembly Cache* (GAC) installiert. Diese Konzept ist ähnlich zum zentralen Speicherort der COMs. Allerdings gibt es für jede Hardware-Plattform, die von der Laufzeitumgebung angesprochen werden kann, einen eigenen GAC in dem die dafür erzeugten Assemblys installiert werden. Ein großer Unterschied ist, dass die Assemblys im GAC nicht in der *Windows-Registry* eingetragen werden. Mit diesem Konzept wird schon ein Teil der „DLL-Höllen“-Problematik umgangen.

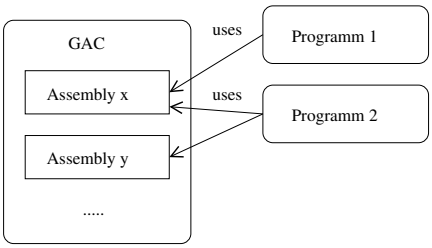

Abbildung 2.3: Der Global Assembly Cache (GAC)

Die Abbildung 2.3 zeigt die Verwendung des GAC als globale Komponentenspeicherstelle für die Zugriffe von verschiedenen Assemblys. Assemblys, die im GAC installiert werden sollen, müssen einen „starken Namen“ haben. Dies wird durch

- den Dateinamen
- eine Versionsnummer
- der Kultur[8]
- und einem Token

gewährleistet. Die Kombination dieser Merkmale sollen ein Assembly einmalig machen, so dass das Referenzieren eines falschen Assemblys relativ unwahrscheinlich ist. Zusätzlich wird damit die Koexistenz von gleichnamigen Assemblys in verschiedenen Versionen ermöglicht.

[8] Die Standardkultur eines Assembly ist neutral und. Es können aber auch „ortsgebundene“ Assemblys erzeugt werden. Beispielsweise ein Assembly mit deutscher und eins mit japanischer Kultur, die jeweils unterschiedliche Zeichensätze und kulturbedingte Bezeichnungen benutzen.

Um der Kompromittierung und Fälschung von Softwarekomponenten vorzubeugen, wird ein Assembly mit Hilfe eines öffentlichen und privaten Schlüssels signiert. Zudem wird eine Hashsumme aus den Assembly-Daten erzeugt, die dann als Token gespeichert wird. Diese Verschlüsselung schützt jedoch nicht deren Inhalt und somit nicht das enthaltene geistige Eigentum in einem Assembly. Es ist eher eine Art Validierungsmöglichkeit der Dateikonsistenz.

Das Assembly-Konzept ist eine gute Alternative zum bisherigen COM-Konzept der Windows-Betriebssysteme. Damit nicht ausschließlich eines der beiden Komponentenkonzepte genutzt werden muss, bietet das Microsoft .NET-Framework die Möglichkeit Softwarekomponenten aus beiden Konzepten zu verwenden. Auf diese Besonderheit werde ich im Abschnitt 2.3.10 eingehen.

2.2 .NET

Microsofts Initiative zur CLI-Entwicklung und ihr erstes Release ihrer CLI-Implementierung unter dem Namen .NET-Framework war ausschlaggebend für den Begriff .NET.

.NET umfasst aber nicht nur das .NET-Framework sondern beschreibt viel mehr eine gesamte Softwareentwicklungsinfrastruktur, dazu gehört eine CLI-Implementierung, eine Entwicklungsumgebung, Compiler und andere Hilfsprogramme. Beispielsweise umfasst Microsoft .NET folgende Elemente:

- eine Implementierung der CLI, das .NET-Framework
- das Visual Studio als eigene Entwicklungsumgebung: Dazu gehören Compiler für die CLI-konformen Programmiersprachen J#, C#, Visual Basic .NET und C++/CLI sowie Visual C++ als nicht CLI-kompatible Sprache. Zudem gibt es noch weitere unterstützende Werkzeuge wie zum Beispiel einen Disassembler[9] für Assemblys.

.NET ist so zusagen eine Art Markenname für die Zusammenfassung der genannten Komponenten. Der Name wir häufig als Schlagwort verwendet und wird teilweise auch für andere CLI-Implementierungen verwendet und bildet das Gegenstück zum Java-Konzept.

Microsoft stellt seit 2002 mit dem Visual Studio Compiler für die Sprachen J#, C# und C++/CLI sowie eine objektorientierte Neuauflage der beliebten Programmiersprache Visual Basic (VB.NET) zur Verfügung um Windows-Applikationen und dynamische Webseiten zu entwickeln. Besonders interessant in diesem Rahmen ist die Sprache C#. Sie wurde eigens für das .NET-Framework entwickelt. Derzeit können Entwickler unter ca. 70 weiteren Sprachen von Drittanbietern wählen, deren Compiler bereits an diese Plattform angepasst wurden (siehe dazu [Sch07]). Jede dieser Sprachen erzeugt ein Kompilat (Assembly), das konform zur *Common Language Infrastructure* ist und somit auf der Laufzeitumgebung ausgeführt werden kann.

[9] Ein Disassembler extrahiert Programminformationen aus einem Kompilat.

Als relevante Grundlagen dieser Arbeit werde ich im Folgenden die .NET-Technologie und im Anschluss Open Source .NET-Projekte vorstellen und miteinander vergleichen.

2.2.1 Das .NET-Framework

Das Herzstück der .NET-Technologie bildet das .NET-Framework. Es ist eine Implementierung des ECMA-Standards 335, der schon im Abschnitt 2.1 beschrieben wurde. Das .NET-Framework kann man sich als eine Art Abstraktionschicht zwischen dem Windows-Betriebssystem und der auszuführenden Applikation vorstellen. Diese Anordnung wird in der Abbildung 2.4 dargestellt.

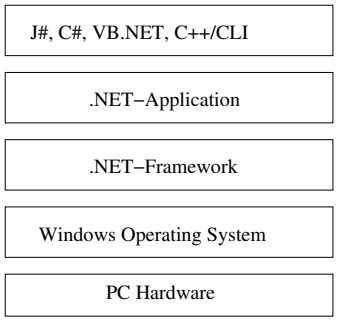

Abbildung 2.4: Der .NET-Framework Stack

Es wurde erstmals im Jahre 2000 erwähnt und schließlich im Jahr 2002 in Version 1.0 veröffentlicht. In den darauf folgenden Jahren wurde das Framework mit Verbesserungen und Erweiterungen ausgestattet. Derzeit liegt es in der Version 3.0 für Windows-Betriebssysteme vor.

Das .NET-Framework ist der Ausgangspunkt für die Entwicklung der aktuellen Windows-Applikationen, da es unter anderem Basisklassen für die Entwicklung von Desktop-, Consolen- und Web-Anwendungen und Webservices zur Verfügung stellt. Es ermöglicht auch ein plattformunabhängiges Programmieren.

Die .NET-Implementierung des CLI-Standards entspricht in seiner Basis der CLI-Spezifikation. Dieser Basis wurden noch weitere Teile hinzugefügt, die das Entwickeln für die Windows-Plattform weiter erleichtern sollen.

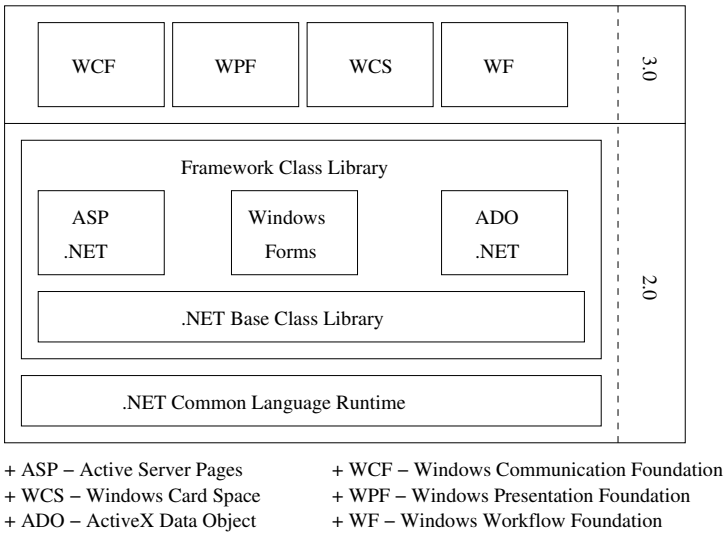

Abbildung 2.5: Aufbau des .NET-Framework 3.0 ([Mic06c])

Die Abbildung 2.5 veranschaulicht den Aufbau des .NET-Frameworks in der Version 3.0. Es besteht aus dem *Compact Profile* der CLI mit einigen Erweiterungen. Die dort veranschaulichten Teile werden in den nächsten Abschnitten näher beschrieben.

Die *Common Language Runtime* (CLR) ist die Implementierung des *Virtual Execution Systems* der CLI, welches eine Laufzeitumgebung für den Programmcode definiert. Die CLR verarbeitet eine spezielle Form von Bytecode, der ebenfalls in der CLI spezifiziert ist. Hierzu gehören die statischen Informationen (Metadaten) und der Methodenbytecode (CIL).

Die *Base Class Library* (BCL) im .NET-Framework ist die Implementierung der *Base Class Library*, *XML Library*, *Network Library* und *Reflection Library* aus dem *Compact Profile* der CLI sowie die *Extended Array Library*, *Parallel Library* und *Floating Point Library* (vgl. Abbildung 2.2).

In der Abbildung 2.5 wird die *Base Class Library* mit den weiteren Bibliotheken zur Erstellung von Datenbankzugriffen (ADO.NET), Web-Anwendungen (ASP.NET) und Windows-Applikationen (Windows Forms) zur *Framework Class Library* zusammen gefasst. Diese drei Bibliotheksteile bilden keine Teilmenge der CLI-Klassenbibliothek ab. Der Teil *ASP.NET* enthält Basisklassen für die Implementierung von dynamischen Webseiten und Webservices. Das Pendant dazu bildet auf der Applikationsseite der Teil *Windows Forms*, der Basisklassen für die Implementierung von Windows-Applikationen bereitstellt.

Das Funktionalitätsumfang der *Framework Class Library* und CLR wurden vom Release 2.0 auf 3.0 kaum verändert. Es ist lediglich eine zusätzliche Abstraktionsstufe zur Softwareentwicklung hinzugekommen. Dies ermöglicht ein einfacheres Trennen von Design und Funktionalität. Die folgenden Teile gehören nicht mehr zum CLI-Standard sind aber Bestandteil des aktuellen .NET-Frameworks:

- Windows Presentation Foundation (WPF)

WPF ist ein grafisches Subsystem für das .NET-Framework. Mit ihm ist eine strikte Trennung zwischen Benutzeroberfläche und Programmlogik möglich. Die Benutzeroberfläche wird mit der Sprache XAML[10], einer auf XML basierenden Sprache, definiert, die dann als Windows-Applikation oder als Web-Applikation ausgeführt werden kann.

- Windows Communication Foundation (WCF)
  WCF ist ein neues Kommunikationssubsystem. Es ermöglicht Interprozess- und Netzwerkkommunikation. Hierzu werden Service-Klassen abgeleitet, die den bereitzustellenden Service beschreiben. Dieser Service wird dann von einem Server bereitgestellt und wird als Endpunkt bezeichnet. Clients können sich zu diesem Endpunkt verbinden, indem sie ein Protokoll und die URL auswählen. Danach kann der Client die vom Endpunkt zur Verfügung gestellten Dienste nutzen.

- Windows Workflow Foundation (WF)
  WF ist eine spezielle Technik um Arbeitsabläufe (Workflows) abzubilden und auszuführen. Die Struktur eines Workflows wird mit XAML beschrieben.

- Windows CardSpace (WCS)
  Windows CardSpace ist ein Identitätsmetasystem. Im *Cardspace* werden die Referenzen zur Benutzeridentität gespeichert und als Karten virtuell für den Benutzer repräsentiert. Mit dem *Cardspace* soll der Benutzer die Möglichkeit haben seine Identität z.B. für das Anmelden auf einer Webseite selbst zu bestimmen und zu verwalten. Hierzu öffnet sich, wenn personenbezogene Daten angefordert werden, der *Cardspace* in dem die hinterlegten Identitätskarten aufgezeigt werden. Nachdem der Benutzer eine Karte ausgewählt hat, werden die Informationen in Form einer digital signierten XML-Datei geschickt. Dieses Konzept kann derzeit nur mit einigen Diensten von Microsoft genutzt werden.

Microsofts .NET-Framework ist die am weitesten ausgeprägte CLI-Implementierung. Außerdem werden Compiler für verschiedene Programmiersprachen bereitgestellt, die den vollen Umfang der CLI nutzen können. Zusätzlich gibt es umfangreiche Klassenbibliothek sowie einen Disassembler[11] für die Zwischendarstellung und gute Debugunterstützung für die Programmentwicklung. Aus diesem Grund wird diese Implementierung der CLI als Grundlage für die Analyse der spezifizierten Assemblys und der darin enthaltenen Zwischendarstellung in dieser Arbeit dienen.

2.2.2 Open Source .NET-Projekte

Neben dem proprietären .NET-Framework gibt es auch frei verfügbare Open Source Implementierungen des CLI-Standards. Drei dieser Implementierungen werde ich in diesem

[10] E**X**tensible **A**plication **M**arkup **L**anguage

[11] Der Disassembler wandelt das Binärformat von Kompilaten wie Assemblys in eine menschenlesbare Form um.

Abschnitt kurz vorstellen und mit dem .NET-Framework vergleichen.

Rotor Projekt

Rotor,[12] oder auch Shared Source CLI[13] genannt, ist eine Microsoft-Implementierung der *Common Language Infrastructure* und eines C# Compilers. Es bildet das Grundgerüst der CLI ab und wurde unter einer Shared Source-Lizenz veröffentlicht. Diese Lizenz beschreibt eine ausschließliche Nutzung des Quelltextes für Forschungszwecke, so dass der Quelltext nicht kommerziell genutzt werden darf. Die Implementierung wurde für folgende Betriebssysteme konzipiert: Windows XP, Windows 2000, FreeBSD und Mac OS X.

Das implementierte *Virtual Execution System* basiert auf der kommerziellen CLR des .NET-Frameworks. Die CLR ist speziell auf Windows-Betriebssysteme zugeschnitten, so dass zur Unterstützung anderer Betriebssysteme eine Zwischenschicht (Platform Abstraction Layer) zwischen Betriebssystem und CLR eingefügt wurde. Mit dieser Zwischenschicht werden die benötigten Windows-Funktionen für die anderen nicht Windows-Plattformen zur Verfügung gestellt.

Das Projekt Rotor stellt folgende Werkzeuge bereit:

- Laufzeitumgebung
- Klassenbibliothek mit minimalem Funktionsumfang
- Compiler für C#
- Disassembler für Assemblys
- Debuggerunterstützung
- Build Tools zum Bauen des Projektes auf den verschiedenen Plattformen

Diese beispielhafte Implementierung der CLI ist eher zum Lernen und Verstehen geeignet als für den konkreten Einsatz.

Mono Projekt

Mono[14] ist ein von der Firma Novell unterstütztes Projekt zur Implementierung des CLI-Standards für Unix-Betriebssysteme. Mittlerweile ist es in der Version 1.2.6 erhältlich und unterstützt die Betriebssysteme Linux, Solaris, Mac OS X, Windows und Unix.

Das Mono Projekt bietet folgende Entwicklungswerkzeuge an:

- Laufzeitumgebung inkl. JIT-Compiler

[12] `http://research.microsoft.com/programs/europe/rotor/`
[13] `http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&dis`
`en`
[14] `http://www.mono-project.com/Main_Page`

- Compiler für C# und einen, sich in der Betaphase befindlichen, VB.NET Compiler
- Klassenbibliothek, die auch ASP.NET, ADO.NET und *Windows Forms* umfasst
- Spezielle Bibliotheken, die für die grafische Darstellung unter Unix benötigt werden
- Entwicklungsumgebung Monodevelop[15] mit der zum Beispiel durch grafische Darstellung und Manipulation grafische Benutzeroberflächen erstellt werden können
- Disassembler
- Debuggerunterstützung

Dieses Projekt ist meines Erachtens nach neben dem .NET-Framework am weitesten mit der Umsetzung der CLI-Spezifikation. Es unterstützt schon viele Teile aus der .NET-Version 2.0 wie z.B. generische Objekte und wird kontinuierlich weiterentwickelt.

DotGNU Portable.NET

Die dritte Open Source Implementierung, die ich vorstelle, ist ein Teil des DotGNU Projektes[16]. Es hat sich zum Ziel gesetzt, eine Reihe freier Softwarewerkzeuge für die CLI-Applikationsentwicklung bereitzustellen. Hierzu gehört auch eine Implementierung des CLI-Standardes genannt DotGNU Portable.NET. Es unterstützt wie die schon vorgestellten Projekte primär Unix-Betriebssysteme wie Linux, Mac OS X, Solaris und AIX.

DotGNU Portable.NET bietet die folgenden Werkzeuge an:

- Laufzeitumgebung
- Basisklassen der *Base Class Library* und Funktionen aus der *XML Library* sowie Unterstützung der *Windows Forms*
- Compiler für C#

Portable.NET unterstützt teilweise schon spätere CLI-Bestandteile wie Generics, allerdings ist die Basisklassenunterstützung weniger umfangreich als die des Mono Projektes und die des .NET-Frameworks.

2.3 C#

C# wurde analog zum .NET-Framework entwickelt und sollte die CLI-Spezifikation komplett erfüllen ([ECM06a]). Sie sollte eine einfache, moderne, universell einsetzbare und objektorientierte Programmiersprache werden. Zudem sollte ihre Ähnlichkeit zu den Programmiersprachen Java und C++ das Erlernen der Sprache und somit den Umstieg auf

[15] `http://www.monodevelop.com/Main_Page`
[16] verfügbar unter `http://www.gnu.org/software/dotgnu/`

.NET attraktiver und leichter machen. C# ist direkt auf die Anforderungen und die Verwendung der CLI-Konzepte ausgelegt, deswegen habe ich sie als Sprache für die Analyse in dieser Arbeit ausgewählt. Vorteilhaft ist auch, dass sie im ECMA-Standard 334[ECM06a] konkret spezifiziert und umfassend beschrieben wird.

C# kann für die Entwicklung von Applikationen aber auch zur Entwicklung von Webseiten verwendet werden. Die Webseitenentwicklung bringt eigene Konzepte mit sich, die nicht ohne weiteres auf Applikationssoftware abgebildet werden können. Aus diesem Grund werde ich mich im Rahmen dieser Arbeit nur mit C#-Applikationen befassen.

Dieser Abschnitt soll einen allgemeinen Überblick über die Sprache geben, sie später zur Erzeugung der zu analysierenden Assemblys genutzt wird. Dazu gehören unter anderem der Programmaufbau und einige Spracheigenheiten.

Für dieses Kapitel wurden, soweit nicht anders gekennzeichnet, diese [ECM06a, Ric06, Mic05a, Mic05b, Gun00] Quellen verwendet.

2.3.1 Programmaufbau

C# ist, wie schon eingangs erwähnt, eine objektorientierte Programmiersprache. Damit wird die Grundidee der Flexibilität und Wiederverwendbarkeit von Programmen gefördert und manifestiert. In C# werden Daten und Funktionen in Objekten zusammengefasst. Konkrete Objekte sind in C# Klassen und Stukturen (*Structs*), abstraktere Objekt-Vertreter sind Schnittstellen (*Interfaces*), abstrakte Klassen und generische Objekte (*Generics*).

Das Beispiel in Listing 2.1 zeigt ein kleines Beispielprogramm in C#, das die Syntax und den Aufbau skizziert. C# ist eine von Groß- und Kleinschreibung abhängige Sprache. Die Objekte der Programmiersprache werden in Namensräumen (*Namespaces*) organisiert. Im Gegensatz zu Java sind die Dateinamen unabhängig von dem darin deklarierten Objektnamen wählbar. Objekte können seit dem .NET Framework 2.0 mit dem Schlüsselwort *Partial* über mehrere Dateien oder in verschiedenen Bereichen einer Datei definiert werden. (vgl. Listing 2.1).

Die Bereiche von Namensräumen-, Objekten- und Methoden-Definitionen werden mit geschweiften Klammern eingegrenzt. Zusätzlich ist es möglich spezielle Blöcke beispielsweise für Zeigeroperationen zu definieren. Dies wird im Listing 2.1 mit dem Schlüsselwort *unsafe* gezeigt. Innerhalb dieses Blockes dürfen Zeiger verwendet werden, außerhalb des Blockes nicht. Einzelne Anweisungen werden am Ende mit einem Semikolon abgeschlossen.

```
1 namespace HelloWorld{
2
3   // First part of MyClass
4   partial class MyClass{
5 static string s = "Hello World!";
6   }
7
8   // Second part of MyClass
```

```
9   partial class MyClass {
10    static void Main()
11    {
12      int i = 5;
13      // Define a unsafe block to allow pointers
14      unsafe {
15  int *p = &i;
16  for (int j = 1;j <= *p;j++){
17    System.Console.WriteLine(s+ "[{0}]", j.ToString());
18  }
19      }
20    }
21  }
22 }
```

Listing 2.1: Syntaxbeispiel C#

2.3.2 Namensräume

Namensräume (*Namespaces*) werden verwendet um C#-Programme zu strukturieren. Sie bieten, ähnlich wie *Packages* unter Java, die Möglichkeit Programmteile zusammenzufassen oder auch die Sichtbarkeit von Objekten auf bestimmte Programmteile zu beschränken. Zur Definition eines Namensraumes wird das Schlüsselwort „namespace“ verwendet.
Namespaces können Klassen, Schnittstellen, *Structs*, Aufzählungstypen (*Enums*), typsichere Methodenzeiger (Delegaten), generische Objekte und weitere Namensräume enthalten.

2.3.3 Zugriffsmodifizierer

Neben den aus Java bekannten Sichtbarkeitsattributen *Public*, *Private* und *Protected* für Objekte, gibt es in C# zusätzlich noch *Internal* und *Protected Internal*. Die Tabelle 2.1 gibt einen Überblick über die verfügbaren Zugriffsmodifizierer und ihre Beschränkungen.

Deklarative Zugriff	**Bedeutung**
Public	Der Zugriff ist nicht eingeschränkt.
Protected	Der Zugriff ist auf die enthaltene Klasse oder die von der enthaltenden Klasse abgeleiteten Typen begrenzt.
Internal	Internal beschänkt die Sichtbarkeit eines Typs auf das aktuelle Assembly.
Protected Internal	der Zugriff ist auf das aktuelle Assembly oder die von der enthaltenden Klasse abgeleiteten Typ begrenzt.
Private	Der Zugriff ist auf den enthaltenden Typ begrenzt.

Tabelle 2.1: Übersicht Zugriffsmodifizierer [Mic08a]

Die Beispiele in Listing 2.2 und Listing 2.3 verdeutlichen die Verwendung des

Internal-Modifizierers. In Listing 2.2 werden zwei Klassen definiert, eine mit einem *Public*- und eine mit *Internal*-Modifizierer. Dieses Listing wird als Bibliothek kompiliert und im Listing 2.3 referenziert. Dort kann dann von der öffentlichen Klasse *MyPublicClass* eine Instanz erzeugt werden. Von der assemblyinternen Klasse *MyInternalClass* kann jedoch keine Instanz erzeugt werden, da sie außerhalb der kompilierten Bibliothek nicht sichtbar ist. Ähnlich verhält sich dies für die Kombination *Protected Internal*. Es beschränkt die Zugriffe auf das aktuelle Assembly oder auf die von der Klasse abgeleiteten Typen.

```
1 // compile it with: /target:library
2 internal class MyInternalClass
3 {
4   public static int i = 0;
5 }
6 public class MyPublicClass
7 {
8   public static int j = 1;
9 }
```

Listing 2.2: Quelltext des Assembly *MyInternalAssembly*

```
1 // compile with: /reference:MyInternalAssembly.dll
2 class TestAccess
3 {
4   public static void Main()
5   {
6     MyPublicClass myPClass = new MyPublicClass(); // no error
7     MyInternalClass myIClass = new MyInternalClass();
8     /* Statement above produce a compile error,
9      * that's why MyInternalClass is not visible outside
10     * the assembly MyInternalAssembly.dl
11     */
12  }
13 }
```

Listing 2.3: Quelltext des Assembly *MyAssembly* welches das Assembly *MyInternalAssembly* verwendet.

2.3.4 Klassen, Strukturen und Schnittstellen

Zur Objektdefinition können in C# Klassen wie auch Strukturen und Schnittstellen verwendet werden.

Klassen stammen vom Objekt *System.Object* ab und können Datenfelder (Variablen und Konstanten) wie auch funktionale Mitglieder (Methoden, Eigenschaften (Propertys), Ereignisse) und geschachtelte Typen enthalten.

Structs sind Klassen recht ähnlich stammen aber vom Objekt *System.ValueType* ab. Sie können neue „primitive“ Datentypen[17] definieren. Die wesentlichen Unterschiede zwischen

[17] Zu den primitiven Datentypen zählen u.a. Integer, String, ...

Klassen und *Structs* werden in der Tabelle 2.2 zusammengefasst.

Characteristics	**Class**	**Struct**
Derives from	System.Object	System.ValueType
Can define new primitive types	no	yes
Can include nested types	yes	yes
Kind of data holding	reference to heap	direct
Can be abstract	yes	no
Is implicitly sealed	no	yes
Can extend another class	yes	no
Can implement an interface	yes	yes

Tabelle 2.2: Überblick über die Unterschiede von Klassen und *Structs*

Schnittstellen, können nur Signaturen von Methoden, Delegaten (siehe Abschnitt 2.3.6), Eigenschaften (Propertys) oder Ereignissen enthalten. Klassen und *Structs* können die konkrete Ausprägung einer Schnittstelle sein.

2.3.5 Vererbung

C# erlaubt Polymorphismus, so dass eine Klasse von einer Klasse (Basisklasse) abstammen aber auch mehrere Schnittstellen implementieren kann. Eine Schnittstelle kann ebenfalls von einer oder mehreren Schnittstellen abstammen, jedoch nicht von einer Klasse oder Struktur. Bei *Structs* ist der Polymorphismus eingeschränkt, so können sie nur eine Schnittstelle implementieren, aber nicht von einer anderen Basisklasse abstammen. Ein abgeleiteter Typ (Klasse, *Struct* oder Schnittstelle) übernimmt alle Methoden, Felder, Eigenschaften (Propertys) und Ereignisse des Basistyps.

Geerbte Klassenmitglieder eines Objektes können zusätzlich abstrakt, virtuell oder versiegelt sein. Versiegelte Klassenmitglieder können zwar vererbt aber nicht mehr durch Überschreiben verändert werden. Diese Eigenschaft wird mit dem Schlüsselwort *sealed* ausgedrückt. Abstrakte Methoden, Eigenschaften (Propertys) und Ereignisse einer abstrakten Klasse müssen, wenn sie geerbt werden, durch überschreiben implementiert werden. Virtuelle Eigenschaften einer Klasse können hingegen überschrieben oder neudefiniert werden. Der Unterschied zwischen überschreiben und neudefinieren äußert sich wie folgt:

1. Überschreiben mit dem Schlüsselwort *overrides*
   Das geerbte virtuelle oder abstrakte Basistypmitglied wird überschrieben.

2. Redefinition mit dem Schlüsselwort *new*
   Das geerbte Basistypmitglied wird durch ein Neues ersetzt und kann somit auch durch das Schlüsselwort *sealed* versiegelte Mitglieder erben und neudefinieren

([Ske04, Mic08c]). Weiterhin zeigt es dem Compiler ein explizites Überschreiben für geerbte Variablen an.

Das Listing 2.4 zeigt beispielhaft die Verwendung der beiden Schlüsselwörter. Bei der Neudefinition der Methode *MyMethod* wird bei einer Typumwandlung von *DerivedNew* zu *BaseClass* die Methode *MyMethod* der Basisklasse aufgerufen. Im Gegensatz dazu überschreibt *overrides* die Methode *MyMethod* für die Klasse *DerivedOverride* und seine Basisklasse. Dadurch wird auch bei einer Typumwandlung der Klasse *DerivedOverride* zu seinem Basistyp die überschriebene Methode aufgerufen. Zusammengefasst überschreiben also beide Schlüsselwörter die virtuelle Methode, jedoch haben sie unterschiedliche Auswirkungen bei Aufrufen durch die Vererbunghierarchie.

```
1  // Define the base class
2  public class BaseClass
3  {
4      // Define a virtual method
5    public virtual void MyMethod()
6    {
7      System.Console.WriteLine("base method");
8    }
9  }
10
11 public class DerivedNew : BaseClass
12 {
13 // Redefine MyMethod
14   public new void MyMethod()
15     {
16       System.Console.WriteLine("new method");
17   }
18     }
19
20 public class DerivedOverride : BaseClass
21 {
22  // Override the base class definition for MyMethod
23   public override void MyMethod()
24   {
25     System.Console.WriteLine("override method");
26   }
27 }
28
29 public class X{
30
31   public static void Main (){
32
33     BaseClass b = new DerivedNew();
34     DerivedNew d = new DerivedNew();
35     BaseClass bo = new DerivedOverride();
36     DerivedOverride dp = new DerivedOverride();
37
38     b.SomeOtherMethod();// Output: base method
```

```
39      d.SomeOtherMethod();// Output: new method
40
41      bo.SomeOtherMethod();// Output: override method
42      dp.SomeOtherMethod();// Output: override method
43    }
44  }
```

Listing 2.4: Polymorphismus in C#

2.3.6 Delegaten

Ein Delegat (*Delegate*) ist eine spezielle Klasse, deren Instanz eine Aufrufliste kapselt, welche eine oder mehrere Methoden als aufrufbare Instanz (*Callable Entity*) referenziert. Sie können zu statischen und nicht statischen Methoden zeigen unter der Voraussetzung, dass die Methodensignatur der Signatur des Delegaten entspricht. Wenn eine Delegat-Instanz aufgerufen wird, werden alle *Callable Entitys* des Delegaten aufgerufen. Der Delegat gibt hierbei seine Argumente an die *Callable Entitys* weiter. Abstrakt gesehen bildet ein Delegat einen typsicheren Funktionszeiger. Diese werden unter anderem für die Ereignisbehandlung in C# verwendet und sind in diesem Fall die Vermittler zwischen Ereignisauslöser und Ereignisbehandler.

Delegaten können in Single- und Multicast-Delegaten unterschieden werden.

Singlecast-Delegaten

Sie stammen vom Objekt *System.Delegate* ab und enthalten eine Aufrufliste mit einer Methode. Im Listing 2.5 wird das Vorgehen für die Nutzung verdeutlicht. Zuerst wird der Delegat *MyDelegate* definiert, seine Signatur umfasst einen *String* und keinen Rückgabewert. Danach wird der Delegat mit der Methode *MyFunction* instanziiert und mit *md("called by delegate..");* aufgerufen. Dadurch wird dann die Methode *MyFunction* aufgerufen und in der Console erscheint der übergebene *String*.

```
1  using System;
2
3  // 1. Declare the delegate
4  public delegate void MyDelegate(String s);
5
6  class TestDelegate
7  {
8    public static void MyFunction(String s)
9    {
10     Console.WriteLine(s);
11   }
12
13   public static void Main(){
14     // 2. Instanciate the delegate with method
15     //    MyFunction
```

```
16      MyDelegate md = new MyDelegate(MyFunction);
17      // 3. Invoke the delgate
18      md("called by delegate..");
19    }
20  }
```

Listing 2.5: Ein Singlecast Delegate

Multicast-Delegaten

Multicast-Delegaten stammen von der Klasse *System.MulticastDelegate* ab. Ihre Aufruflisten können mehreren Methoden enthalten. Die im Delegaten enthaltenen Methoden *Combine* und *Remove* werden für das An- und Abmelden der Methoden zu den Aufruflisten verwendet. Mittels *Combine* werden Methoden in die Liste aufgenommen und mit *Remove* wieder aus der Liste entfernt.

Das Haupteinsatzgebiet von Multicast-Delegaten ist die Ereignisbehandlung. Die Verwendung und Erzeugung eines Multicast-Delegaten ist ähnlich zum schon beschriebenen Singlecast-Delegaten. Es umfasst Deklaration, Instanzbildung und Aufruf, wobei der Aufruf beispielsweise durch das Auslösen eines Ereignisses erfolgen kann. Ein Beispiel für die Benutzung dieses Delegatentyps findet sich im Listing 2.6 (Abschnitt 2.3.7).

2.3.7 Ereignisse

Ein Ereignis (Event) ist ein Klassenmitglied, das es einer Klasse ermöglicht andere Klassen oder Objekte über bestimmte interessante Vorfälle zu informieren. Sie werden häufig zur Implementierung von grafischen Benutzeroberflächen benutzt, wobei hier mit den Ereignissen Aktionen des Benutzers wie das Drücken eines Buttons abgebildet wird, um im Anschluss an die Benutzeraktion die nötigen Vorgänge im Programm auszulösen.

Ereignisse in C# verwenden das Veröffentlichen-Abonnieren-Prinzip. In diesem Prinzip veröffentlicht eine Klasse (Herausgeber-Klasse) seine Ereignisse, die auftreten können, und andere Klassen (Abonnenten-Klassen), die Interesse an bestimmten Ereignissen haben, können diese dann abonnieren. Die Methode mit der ein Ereignis aufgerufen wird, wird durch einen Delegaten spezifiziert. Laut Entwurfskonventionen für Ereignisse sollen zur einfachen Handhabung von Ereignissen das Objekt, durch das das Ereignis ausgelöst wurde, und ein Objekt, das Informationen zum Ereignis erhält, übergeben werden. Hierzu gibt es den *System.EventHandler*-Delegaten und die *EventArgs*-Klasse in der Basisklassenbibliothek, der diese Funktionalität erfüllt. Nicht ausgeschlossen ist, dass Ereignisse auch mit selbstdefinierten Delegaten initialisiert werden können.

Klassen, die ein Ereignis abonnieren wollen, müssen für das Abonnement eine Methode mit der Signatur des Event-Delegaten bereitstellen und diese beim Abonnieren angeben (Ereignisbehandler). Nach der Ereignisinitialisierung wartet die Abonnenten-Klasse auf die Auslösung des Ereignisses, so dass dann die dazu definierte Methode aufgerufen wird, die

dann wiederum beliebig komplexe Verarbeitungsschritte nach sich ziehen kann. Ein Ereignis kann mehrere Abonnenten haben und ein Abonnent kann mehrere Ereignisse von mehreren Herausgebern behandeln.

Im Listing 2.6 wird die Programmablaufsteuerung mittels Ereignis verdeutlicht. Zuerst wird der Delegat für das Ereignis definiert. Danach wird in der Klasse X das Ereignis *myEvent* mit dem Delegaten *MyDelegate* zusammen definiert, sowie eine Methode, die das Ereignis mit den entsprechenden Parametern auslöst. In der Klasse Y wird eine Methode mit einer zur Delegatensignatur passenden Signatur definiert. Anschließend wird in der *Main*-Methode eine neue Instanz der Klasse X erzeugt, sowie zu dem Ereignis mit dem Operator $+=$ der mit der Methode M initialisierte Delegat abonniert. Wenn nun die Methode *xx.Fire()* aufgerufen wird, wird das Ereignis *myEvent* ausgeführt. Alle Abonnenten werden benachrichtigt und bekommen die bei der Auslösung übergebenen Werte übergeben. Darauf hin wird dann in diesem Beispiel die Funktion M mit dem übergebenen String ausgeführt.

```
1  // Declare a custom delegate
2  public delegate void MyDelegate(string s);
3
4  public class X{
5    // Declare a event with custom delegate
6    public event MyDelegate myEvent;
7
8    // Method that raises the event
9    public void Fire(){
10     myEvent("Hello Event1");
11   }
12 }
13
14 public class Y {
15   //declare the handlermethod
16   private static void M(string s){
17     System.Console.WriteLine(s);
18   }
19
20   public static void Main(){
21     // Create a instance from class X
22     X xx = new X();
23     // Register the event with handlermethod
24     xx.myEvent += new MyDelegate(M);
25     // Fire the event
26     xx.Fire();
27   }
28
29 }
```

Listing 2.6: Ereignisdefinition und -behandlung

Die Programmsteuerung mit Ereignissen ist bei selbstdefinierten Delegaten etwas

aufwändiger können aber auf die benötigten Informationen genau zugeschnitten werden. Dadurch bildet das Ereigniskonzept ein sehr gutes Kommunikationsmittel für grafische Oberflächen aber auch für Programmteile.

2.3.8 Generics

Seit .NET-Version 2.0 gibt es generische Objekte (*Generics*) in C#. Sie ermöglichen einen parametrisierten Polymorphismus für Schnittstellen, Klassen, Strukturen, Delegaten und Methoden. Generische Objekte erhöhen die Wartbarkeit und Quelltextwiederverwendung innerhalb eines Programms.

Der Code für generische Objekte wird nach Bedarf von der CLR erzeugt, so dass eine Instanz des *Generics* erst im Programmablauf erstellt wird. Wenn mehrere Instanzen eines *Generics* im Programm verwendet werden, versucht die Laufzeitumgebung den Programmcode zwischen den Instanzen zu teilen (siehe [Mic08b]). Damit in diesem Konzept möglichst wenig Laufzeitfehler auftreten, werden *Generics* während der Kompilierung auf Typkonsistenz überprüft, so dass Fehler schon beim Kompilieren gefunden werden (vgl. Listing 2.7).

Generische Objekte sind eine zentrale Definition für eine Aufgabe oder eines Objektes, welche unabhängig von einem konkreten Typ wie *System.Integer* ist. Im Listing 2.7 wird dies mit einer Stackimplementierung verdeutlicht. Ein Stack sollte für alle möglichen Objekte verwendet werden können, so dass die Implementierung am besten keine Typbindung aufweisen sollte. Der Typ eines Generics wird mit einem in spitzen Klammern stehenden Platzhalter definiert. Dieser Platzhalter steht für den übergebenen Typ bei der Initialisierung des Generics. Generics können auch mehrere Platzhalter haben. Diese werden durch Kommata getrennt und können im Rahmen von Vererbung auch separat typisiert werden.

```
1  // generic interface
2  interface IGenericInterface <T>{
3    T Pop();
4    void Push(T item);
5  }
6  // generic class implementing a generic interface
7  class Stack<T> : IGenericInterface <T>{
8     private T[] ItemsArray;
9     private int Index = 0;
10    public Stack(int max_size){ ItemsArray = new T[max_size]; }
11    public T Pop(){
12       if (Index == 0)
13   throw new System.Exception ("Stack is empty");
14      else{
15   return ItemsArray[--Index];
16       }
17    }
18    public void Push(T item) {
```

```
19        ItemsArray[Index++] = item;
20     }
21 }
22
23 class MyClass{
24     static unsafe void Main(){
25     // Declare the generic of type int
26      Stack<int> stack = new Stack<int>(10);
27      stack.Push(1);
28      int number1 = stack.Pop(); // typeconform, no compiler error
29      string s = stack.Pop(); // this raises a compiler error
30      System.Console.WriteLine(number1);
31     }
32 }
```

Listing 2.7: Beispiel: Generics

2.3.9 Propertys und Indexer

Eigenschaften (Propertys) sind Typmember zugehörig zu Klassen oder Structs, die mit einer Set-/Get-Methode implementiert werden. Propertys ersetzen selbstgeschriebene Get-/Set-Methoden (Accessoren) für private Variablen innerhalb einer Klasse, wie man sie z.B. aus der Java-Programmierung kennt. Der Compiler erzeugt automatisch die Signatur und der Rückgabewert der Set- und Get-Methoden des Property. Ein Property wird wie eine öffentliche Variable benutzt. Das Listing 2.8 zeigt beispielhaft den Einsatz von Propertys. Zuerst wird eine private Variable *px* deklariert. Danach wird das Property *X* vom gleichen Typ wie die private Variable deklariert und die Funktionen der Get-/Set-Methoden definiert. Auffällig ist, dass die Set-Methode einen implizit vorhandenen Parameter *value* hat. Dieser wird vom Compiler automatisch generiert und kann für die Wertzuweisung verwendet werden. Die private Variable *px* wird nun indirekt durch die Verwendung des Property *X* manipuliert. Dabei wird impliziert die jeweilig benötigte Methode des Property aufgerufen.

```
1 class MyProp{
2  // private member
3  private int px;
4  // property definition
5  public int X{
6  // get-method definition
7  get{
8   return px;
9  }
10 // set-method definition
11 set{
12  px = value;
13 }
14 }
15 }
```

```
16
17 class MyClass{
18   public static void Main(){
19    MyProp mc = new MyProp();
20
21    mc.X = 10; // calls property-set method
22    int xVal = mc.X; // calls property-get method
23   }
24 }
```

Listing 2.8: Propertys in C#

Der Indexer ermöglicht das Objekt in dem er definiert ist zu indizieren wie ein Array.

Das Indexer-Konzept ist dem der Propertys recht ähnlich. Indexer werden analog zu Propertys mit einer Get-/Set-Methode definiert. Der Bezeichner des Indexers ist immer *this* und kann nicht wie bei Propertys frei gewählt werden. Das zum Indexer korrespondierende Array wird über die Signatur des Indexers identifiziert. Der Einsatz eines Indexers wird im Listing 2.9 veranschaulicht. Nachdem eine Instanz der Klasse *MyClass* gebildet wurde kann die Instanz der Klasse als Array verwendet werden. Die Zugriffe auf die Arrayelemente werden in der Klasse über das gekapselte Array und den dazugehörigen Indexer verwaltet.

```
1 class MyClass
2 {
3  // The private array
4  private string []data = new string[5];
5  // Declare the indexer to []data
6  public string this [int index]
7  {
8   get
9   {
10   return data[index];
11  }
12  set
13  {
14   data[index] = value;
15  }
16 }
17 }
18
19 class MyClient
20 {
21  public static void Main()
22  {
23   // Make a instance from MyClass
24   MyClass mc = new MyClass();
25   // Fill the array
26   mc[0] = "Nele";
27   mc[1] = "Zoe-Lina";
28   // Read the items from array
29  Console.WriteLine("{0},{1}",mc[0],mc[1]);
```

```
30  }
31 }
```

Listing 2.9: Indexer in C#

Die Tabelle 2.3 zeigt die wesentlichen Unterschiede der beiden Programmierkonstrukte.

Characteristics	**Property**	**Indexer**
Member capsulation	Member	Array
Is defined by	It's name	It's signature
Is instance member	Yes	Yes
Can be static	Yes	No
Access through	A member access	A element access

Tabelle 2.3: Überblick über die Unterschiede von Indexern und Propertys

2.3.10 Extern und DLLImport

C# bietet neben der Verwendung von *Managed-Code*[18]-Komponenten aber auch die Möglichkeit *Unmanaged-Code*[19] einzubinden. Bei der Verwendung von *Unmanaged-Code*-Komponenten verlässt man die Sicherheitsbeschränkungen der Laufzeitumgebung, so dass für eventuell auftretende Fehler Vorkehrungen getroffen werden müssen. Einige dieser Vorkehrungen werden durch den *.NET-Managed-Wrapper* getroffen, andere müssen explizit im Code getroffen werden. Die tiefgreifende Einführung in diese umfangreiche Thematik ist nicht Gegenstand dieser Arbeit, deswegen verweise ich an dieser Stelle auf weiterführende Informationen in [ZG08, Mic05a, Micc].

Die Abbildung 2.6 zeigt die Verwendung von *Unmanaged-Code* mit *Managed-Code*. Die Kommunikation der beiden Komponenten und der Austausche von Daten wird über die CLR gesteuert. Hierbei stellt die CLR sicher, dass die *Managed-Daten* richtig bei der Komponente ankommen, wie auch invers. Die internen Vorgänge hierbei werden ausführlich in [Micb] behandelt. Allgemein ausgedrückt wird ein Rückruf (Callback) auf das COM-Objekt erstellt mit dem dann über die CLR zugegriffen werden kann.

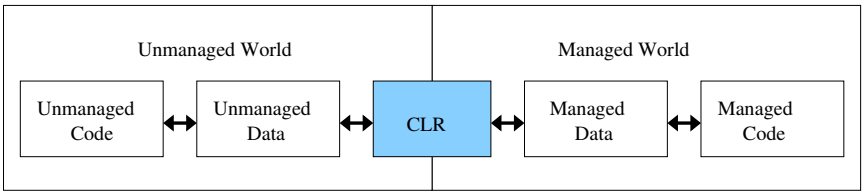

Abbildung 2.6: Kommunikation zwischen *Managed-* und *Unmanaged-Code* im .NET-Framework (aus [ZG08])

[18] *Managed-Code* wird auf einer Laufzeitumgebung ausgeführt

[19] *Unmanaged-Code* wurde plattformabhängig erzeugt und kann nicht in der Laufzeitumgebung ausgeführt werden. Dies sind z.B. alte COM-Komponenten oder DLL-Bibliotheken.

Damit in C# eine *Unmanaged*-Komponente angesprochen werden kann, wird der Compiler darüber informiert, welches Objekt verwendet werden soll. Diese Deklaration erfolgt mit dem Attribut *DLLImport*. Nach dem spezifiziert wurde welches DLL-Objekt verwendet werden soll, wird mit der Deklaration *extern* die Methode spezifiziert, die in C# dieses Objekt aufrufen soll. Diese Funktion kann dann wie eine normale Methode in C# aufgerufen werden.

Im Listing 2.10 wird der Aufruf einer Methode aus einer *Unmanaged-Code*-Komponente, hier die *user32.dll*, gezeigt. Die Methode *MessageBoxA* wird mit ihrer Signatur spezifiziert und später mit den entsprechenden Parametern aufgerufen. In diesem Beispiel wurden keine Aspekte zur Kommunikationssicherheit verwendet.

```
1  using System.Runtime.InteropServices; //contains .NET-Managed-Wrapper
2
3  public class zzz
4   {
5    [DllImport("user32.dll")]
6    public static extern int MessageBoxA
7     (int hWnd,string text,string caption,uint type);
8
9    public static void Main(){
10    // call external method
11    MessageBoxA(0,"text", "caption", 0);
12
13   }
14 }
```

Listing 2.10: Beispiel für das Benutzen von *Unmanaged Code*

2.4 Metamodelle zur Programmrepräsentation

Softwaresysteme in ihrer Quelltextrepräsentation oder in ihrer Bytecode-Darstellung sind keine adäquaten Mittel um Software auf Architekturebene zu untersuchen. Die Komplexität eines Softwaresystems kann beliebig groß sein, so dass die eingangs genannten Repräsentationsformen nicht geeignet sind, um sich einen Überblick über das aktuell vorliegende Problem bilden zu können. Dazu muss von der konkreten Vorlage abstrahiert werden. Zudem sollte eine Abstraktion die Möglichkeit bieten kontextabhängig nicht relevante Objekte auszublenden oder Vergleiche zu ermöglichen.

Solche Modelle werden Metamodelle genannt. Sie sind abstrahierende Abbildungen der Realität, die nur ausschnittsweise oder kontextabhängig Dinge der Realität abstrahiert abbilden. Diese Abstraktion kann man sich bei großen Softwaresystemen zunutze machen um eine übersichtliche Darstellung des Programminhaltes zu erhalten.

Hierbei stellt sich jedoch die Frage welche Elemente im konkreten Kontext wirklich interessieren und somit in die abstrahierende Darstellung müssen. Eine solche Abstraktion kann relativ Quelltext nah wie die in [RVP] beschriebene *Intermediate Language* sein oder sich auf wenige Elemente beschränken. Die Wahl des richtigen Modells hängt auch immer von der gewünschten Granularität der Abbildung sowie ihrem Verwendungszweck ab. Aus diesem Grund gibt es verschiedene Programmrepräsentationsformen für Softwaresysteme. Eine kleine Übersicht über gängige Programmrepräsentationsformen im Software-Reengineering mit ihren Anforderungen und Abbildungsmöglichkeiten sind in [JCD02] zu finden.

In diesem Abschnitt werde ich drei Metamodelle vorstellen die im Software-Reengineering verwendet werden um Softwaresysteme zu abstrahieren. Diese Metamodelle haben alle eine ähnliche Abstraktionsstufe, in der im wesentlichen globale Programmbestandteile wie Klassen, Methoden, Variablen und ihre Beziehungen abgebildet werden. Der Schwerpunkt liegt beim RFG-Modell, welches im Bauhaus-Projekt und in dieser Arbeit zur Abbildung von C# verwendet wird. Zusätzlich werden zwei ähnliche Metamodelle das FAMIX und Dagstuhl Middle Metamodel kurz vorgestellt und miteinander verglichen.

2.4.1 RFG

Der Resource Flow Graph (RFG), beschrieben in [RVP, KEPG99], ist eine stark abstrahierende Zwischendarstellung von Programmquelltext oder Bytecode. Er dient im Bauhaus-Projekt zur Darstellung und Untersuchung der architektonischen Softwarekomponenten und bildet dessen logische Dekomposition ab. In Abschnitt 1.1 wurde beschrieben, dass es möglich ist, einen RFG auf der Basis von Quelltext aber auch eines Bytecodes zu gewinnen. Im Weiteren wird die Erstellung aus Quelltext synonym für die Erstellung eines RFG aus Bytecode oder Quelltext verwendet.

Der RFG ist ein gerichteter Graph, der die globalen Bestandteile eines Softwaresystems

abbildet. Es existieren hierfür abstrakte Konzepte, die möglichst sprachunabhängig die Bestandteile einer Programmiersprache abbilden. In diesem Konzept gibt es Knoten, die Typen, Variablen und globale Unterprogramme abbilden. Zusammenhänge wie Aufrufe oder die Typzugehörigkeit eines Elementes werden mit Kanten zwischen den Knoten abgebildet. Sie bilden die spezifischen Beziehungen der Knoten untereinander ab.

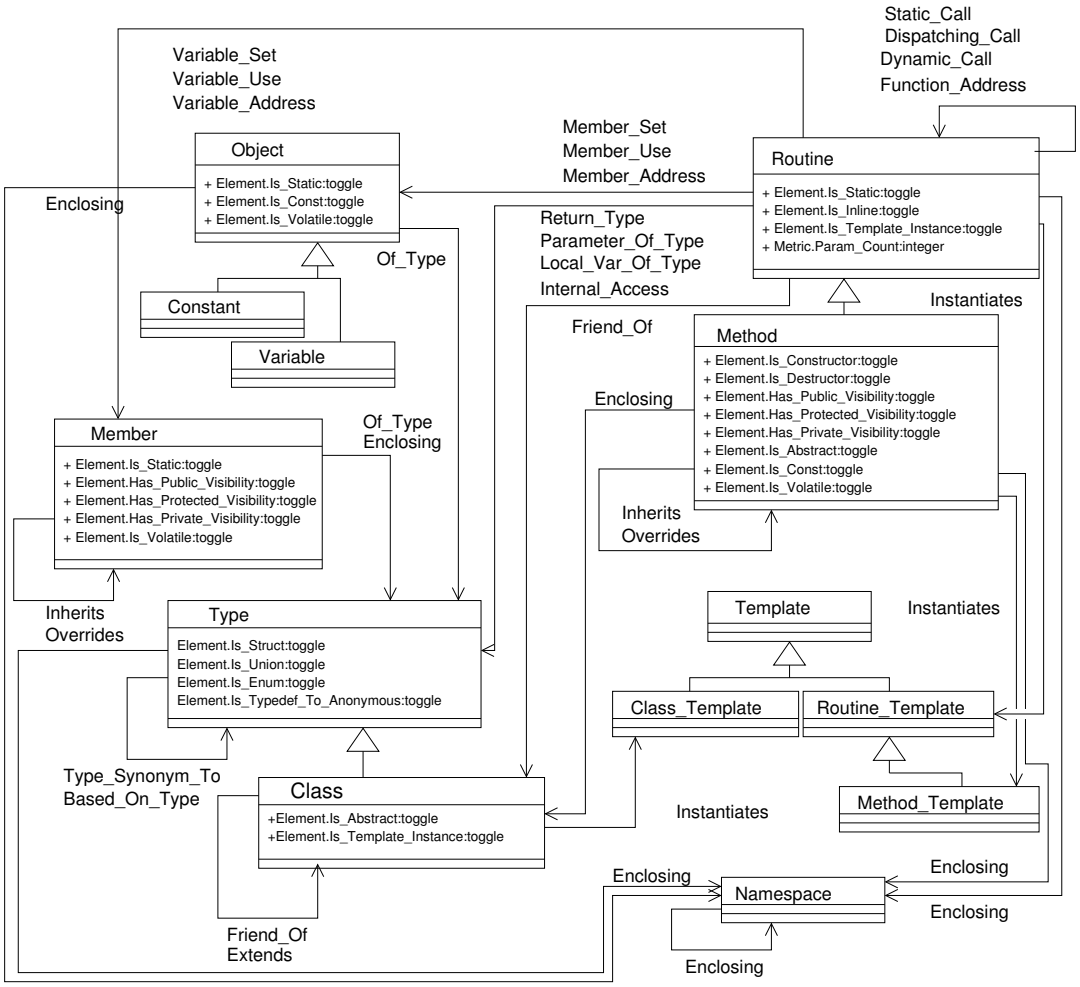

Abbildung 2.7: Das RFG-Modell für C++ [Axi06]

Die unterschiedlichen Knoten und Kantentypen des Metamodells sind in Hierarchien organisiert. Zur Darstellung des RFGs wird eine UML[20]-Notation verwendet. Die aus der Bauhaus-Dokumentation entnommene Abbildung 2.7 zeigt beispielhaft den Teil des RFG-Modells für die Sprache C++ ohne Multiplizitäten.

Klassen-Entitäten bilden die verschiedenen Knotentypen und ihre Vererbungshierarchie ab. Hinzu kommen die Assoziationen, die die Kantentypen des RFGs repräsentieren, wobei die Pfeilspitzen der Kanten immer die Richtung der jeweiligen Beziehung angeben. Ein Kantentyp kann mehrmals im Schema enthalten sein, jedoch darf ein Knotentyp nur einmal vorkommen. Namensgleiche Kanten gehören immer dem gleichen Kantentyp an. Außerdem

[20] Unified Modeling Language

sind Assoziationen mit mehreren Bezeichnern als zusammenfassende Darstellung für alle einzelnen Kanten zu verstehen. Im RFG-Schema wird zudem von den generell verfügbaren Typen im RFG abstrahiert und nur die zur Abbildung der Sprache wichtigen Teile verwendet. Ausnahmen bilden hiervon Generalisierungen wie *Type* und *Template*. Sie werden zur strukturierteren Sprachabbildung mit im Diagramm dargestellt. Verzichtet wird jedoch in dieser Darstellung auf obligatorische Attribute wie Entitätsnamen oder Quelltextpositionen.

Attribute der Knotentypen werden als boolsche Werte repräsentiert. Besitzt ein Knotentyp eine bestimmte Eigenschaft wird das jeweilige Attribut am Knotentyp angezeigt und ansonsten weggelassen. Sie werden genutzt, um zusätzliche Informationen der Knotentypen abzubilden. Diese Attribute können durchaus auch nur rein informativ zur Unterstützung des Programmverständnisses sein.

Knotentypen wie *Namespace*, *Routine*, *Type*, *Template* und *Object* sowie deren Ableitungen bilden die Basiselemente der Sprache C++ ab (vgl. Abbildung 2.7). Zu den Knoten zählen auch deren Beziehungen, dargestellt in Form von Assoziationen. Diese Basiselemente verdeutlichen, dass die Repräsentation auf einer konzeptionellen Ebene stattfindet und nicht auf einer quelltextnahen Ebene. Dies wird deutlich, wenn die Methodenkonzeption betrachtet wird. Hier werden nur spezielle Eigenschaften einer Methode wie ihre Sichtbarkeit im Schema abgebildet, jedoch keine weiteren Informationen über lokal definierte Variablen innerhalb einer Methode.

Ungeachtet der Beschränkung auf globale Programmelemente, werden auch Elemente dargestellt, die vielleicht nicht unbedingt global sichtbar sind, aber von Interesse sein könnten. Ein Beispiel dafür sind private *Member* die auch im RFG dargestellt werden. Darüber hinaus sind im RFG auch strukturelle Elemente wie *Namespaces* enthalten, die zusätzliche Informationen über die Programmorganisation geben. Neben dem strukturellen Programmaufbau werden auch Informationen zur Vererbungsstruktur abgebildet. Diese werden durch die *Extends*- und *Implementation_Of*-Kante abgebildet. Zusätzlich wird das Erben oder Überschreiben von *Membern* und *Methoden* mit den Kanten *Inherits* und *Overrides* abgebildet. Die Zugehörigkeit von verschiedenen Elementen zu einem *Type* oder *Namespace* wird durch die *Enclosing*-Kante angezeigt. Diese Kante drückt aus, dass ein Element innerhalb eines *Type* oder *Namespace* deklariert wurde.

Die Knoten und ihre Attribute im RFG geben die aus dem Programm hervorgehenden Deklarationen wieder. Ergänzend dazu bilden die Kanten die semantischen Aspekte des Programms ab. Sie modellieren den hierarchischen Aufbau, die Verwendung von Variablen und geben Aufschluss welche Methoden aufgerufen werden. Diese Methodenaufrufe können zum Beispiel einen statischen oder dynamischen Charakter haben, der davon abhängig ist, zu welchem Zeitpunkt bekannt ist, welche Funktion aufgerufen werden soll. Diese Kanten werden anhand der Aufrufe innerhalb des Quelltextes abgebildet. Hierbei werden jedoch keine Informationen, die lokal in der Methode abgearbeitet werden, dargestellt, sondern nur die Informationen, die über die Lokalität der Methode hinausgehen wie der Aufruf anderer

Methoden. Das Schema gibt auch Aufschluss über die Rückgabeparameter und die Signatur einer Methode.

Innerhalb des Bauhaus-Projektes kann der RFG mit dem Programm *Gravis* visualisiert und manipuliert werden. Diese Darstellung bietet dem Benutzer einen Überblick über den architektonischen Aufbau eines Programms, der dann explorativ erforscht oder analysiert werden kann. Eine weitere Möglichkeit zur Analyse ist die Anwendung von Metriken zur Vermessung des vorliegenden Softwaresystems. Eine Besonderheit der Visualisierung mit *Gravis* ist, das über die visualisierten Knoten und Kanten an die jeweilige Quelltextposition der Definition gesprungen werden kann.

Selbst in der Architektursicht können Softwaresysteme sehr groß sein. Aus diesem Grund gibt es im RFG ein Sichtenkonzept. Dieses Konzept beinhaltet, dass nicht immer auf gesamten Knoten- und Kanten-Menge operiert werden muss, sondern das Softwaresystem in kontextabhängigen Sichten (*Views*) dargestellt wird. *Views* sind Teilgraphen, die nur eine Teilmenge von Knoten und Kanten enthalten. Die vier Standardsichten des RFG bilden:

- **Base-View:** Enthält alle Knoten, die im Quelltext definiert wurden, sowie deren Beziehungen.
- **Environment-View:** Enthält alle Knoten, die nicht im Quelltext selbst definiert wurden, sondern aus der Umgebung des Programms stammen. Dies können z.B. API-Klassen oder externe Assemblys sein.
- **Module-View:** In dieser View werden die Programmbestandteile nach ihrer physikalischen Ursprungsquelle der Quelltextdatei geordnet angezeigt.
- **Call-View:** Stellt den Aufrufgraph des Programms dar.

Zu diesen Views können, nach dem Ausführen von Analysen auf dem RFG oder dem Erstellen von eigenen Views, neue Views hinzukommen, die wieder einen spezifischen Kontext des Programms veranschaulichen.

Zusammenfassend ist der RFG ein Modell zur Darstellung globaler Quelltextelemente und ihrer Beziehungen zueinander. Durch die Abbildung der Elemente auf einer höheren Abstraktionsstufe ist es möglich eine Softwarearchitektur abzubilden, die sich noch durch weitere Informationen anreichern lässt. Im Kapitel 3 werde ich ein RFG-Schema für die Programmiersprache C# entwickeln, dass die hier beschriebenen Konzepte aufgreift und wiederverwendet.

2.4.2 FAMIX

Zwischen den Jahren 1996 und 1999 wurde an der Universität Bern im Rahmen des FAMOOS-Projektes[21] ein Konzept entwickelt, wie man objektorientierte Softwaresysteme

[21] FAMOOS steht für **F**ramework-based **A**pproach for **M**astering **O**bject-**O**riented **S**oftware Evolution

mit Reengineering-Methoden und daraus resultierenden Programmen analysieren kann (siehe [SCG02]). Im Laufe des Projektes wurde das FAMIX-Modell[22] entwickelt. Diese Modell sollte den ganzen Reengineering-Lebenszyklus von der Softwareanalyse und ggf. darauf folgende Reengineering-Operationen abbilden können. FAMIX wird genauer in [Tic01, SCG00] beschrieben und wird kontinuierlich weiterentwickelt[23].

Das FAMIX-Modell modelliert Programmquelltexte auf einer zum RFG ähnlichen Abstraktionsstufe. Es besteht aus einer Hierarchie von Konzepten und stellt Entitäten sowie deren Beziehungen zueinander dar. FAMIX modelliert zum Teil auf einer quelltextnäheren Ebene als der RFG, so dass unter anderem auch lokale Variablen von Methoden abgebildet werden. Ein Unterschied zum RFG ist, dass alle klassenähnlichen Konstrukte wie Schnittstellen als Klassen im FAMIX-Modell modelliert werden. Im RFG existiert dafür die Generalisierung *Type*, so dass Klassen und Schnittstellen im RFG-Modell von *Type* abgeleitet werden und eigenständige Entitäten bilden.

Damit das Basismodell des FAMIX auch sprachspezifische Konzepte abbilden kann, sofern diese analyserelevant sind, wurde beim Entwurf des FAMIX-Modells vorgesehen, dass das Modell erweitert werden kann. Hierbei bleibt lediglich das Grundmodell fest vorgegeben, so dass die wichtigen Kerninformationen des Modells immer zwischen den verschiedenen Werkzeugen ausgetauscht werden können. Ein Beispiel dafür sind die javaspezifischen Erweiterungen des Klassenobjektes für die Sprachattribute *Final* und *Public*, beschrieben in [Tic01](S.41ff). Zusätzlich wurde auf den Austausch von Informationen zwischen verschiedenen Tools geachtet, so dass Standardformate wie CDIF[24] und XMI[25] dafür verwendet wurden.

FAMIX bildet anders als der RFG keinen Graphen zur Darstellung der Beziehungen zwischen den Entitäten. Die Beziehungen von Elementen werden textuell als Attribute oder durch eigene Konzepte modelliert. Beispielsweise werden Typzugehörigkeiten durch Attribute, aber Aufrufe und Variablenverwendungen durch eigene Konzepte modelliert. Beides wird im RFG durch spezifische Kanten zwischen den korrespondierenden Elementen dargestellt.

Der RFG verfolgt im Gegensatz zu FAMIX das Ziel vom konkreten Quelltext zu abstrahieren, so dass das Programmverstehen mit Hilfe einer globalen Sicht verbessert wird. FAMIX hingegen wurde konzipiert um den gesamten Software-Reengineeringprozess vom Ausgangsquelltext bis hin zum geänderten Quelltext zu begleiten.

[22] FAMIX steht für FAMOOS Information Exchange Modell
[23] Informationen zur Weiterentwicklung gibt es unter [SCG07]
[24] CDIF - **C**ASE **D**ata **I**nterchange **F**ormat, `http://www.cdif.org/`
[25] XMI - **X**ML **M**etadata **I**nterchange, `http://www.omg.org/technology/documents/formal/xmi.htm`

2.4.3 Dagstuhl Middle Metamodel

Das Dagstuhl Middle Metamodel (DMM), beschrieben in [PLT+02, LTP04, Fli06], ist in internationaler Zusammenarbeit während dem Seminar *Interoperability of Reengineering Tools* in Dagstuhl im Jahre 2001 entstanden. Es ist aus der Idee erwachsen, ein allgemein konzipiertes Metamodell zu entwickeln, dass einen Austausch von Modellen zwischen möglichst vielen Anwendungen im Reengineering ermöglicht.

DMM modelliert einen gemeinsamen Nenner von bestehenden Programmrepräsentationskonzepten. Programmelemente, Quelltextfragmente und ihre Beziehungen werden in diesem Schema sprachunabhängig abgebildet. Die Elemente im DMM werden nicht so quelltextnah wie im FAMIX-Modell modelliert und es enthält keine abstrakten Architekturkonzepte wie das RFG-Modell.

Das Modell zeichnet sich durch eine einfache Modellerweiterung mit so genannten *Extensions* aus (vgl. [LTP04]). Mit diesen individuellen Erweiterungen ist es möglich dem Modell beliebige Abstraktionsgrade hinzuzufügen. Außerdem gibt es eine optionale Schicht, die es ermöglicht die Struktur des Quelltextes zu Modellkonzepten zuzuordnen. Die Erweiterungsmöglichkeit und die Zuordnung von Quelltextelementen zu den Modellkonzepten macht das DMM attraktiv für die Ausführung von Refactorings[26] im Rahmen von Reverse- und Reengineeringprogrammen, beschrieben in [Fli06].

[26] Refactorings sind Strukturverbesserungen am Quelltext, wobei das bestehende Programmverhalten beibehalten wird.

KAPITEL 3

Konzeption eines RFG-Schemas für C#-Programme

Im Abschnitt 2.4.1 wurde schon das Konzept des RFG-Modells vorgestellt. In diesem Kapitel werde ich nun aus den schon vorgestellten Konzepten ein eigenes RFG-Schema für C# entwickeln, dass auf Grund seiner Nähe zur CLI stellvertretend die Basiskonzepte für .NET-Programme abbilden soll.

Dazu wird zuerst der Ausgangspunkt meines Schemas vorgestellt. Im Abschnitt 3.2 werden die in Abschnitt 2.3 vorgestellten Konzepte der Sprache C# im RFG modelliert. Hierbei wird Wert auf die Wiederverwendung vorhandener Konzepte gelegt.

Die Sprache C# hat in [ECM06a] eine feste Definition die abgebildet werden kann und die deklarierten Symbole und Konzepte müssen in den Metadaten abgebildet sein damit sie von einer VES ausgeführt werden können. Aus diesem Grund ist es möglich vor der Analyse der Metadaten ein RFG-Schema für die Abbildung von C# zu konzipieren. Eine Vorabkonzeption stellt zudem klare Anforderungen an die nachfolgende Analyse der Zwischendarstellung.

Kapitelinhalt

3.1 Existierende RFG-Schemata 40

3.2 RFG-Schema für C# 41

3.2.1 Namespaces 41

3.2.2 Typen 42

3.2.3 Klassen und Schnittstellen 42

3.2.4 Klassenvariablen 44

3.2.5 Methoden 45

3.2.6 Propertys und Indexer 47

3.2.7 Events 48

3.2.8 Generics 50

3.2.9 Das vollständige RFG-Schema für C# 51

3.3 Anforderungen an die Analyse von C# 52

3.1 Existierende RFG-Schemata

In der Bauhausdokumentation werden Sprachrepräsentation für die Sprachen C, C++ und Java, sowie in der Diplomarbeit [Had06] ein RFG-Schema für Visual Basic 6, beschrieben.

Aus diesen Schemata werden schon existierende Teile wiederverwendet, um große Veränderungen am Grundkonzept zu vermeiden und damit schon gewonnene Erfahrungen in der Analyse der vorhandenen Sprachen auf C# angewendet werden können. Damit gehört die Umbenennung oder Veränderung solcher Konzepte nicht zu den Aufgaben dieser Arbeit. Aus diesem Grund werden die in den Schemata verwendeten Namen übernommen, auch wenn diese namentlich nicht den Bezeichnern der Sprache entsprechen, sie aber die gleiche Eigenschaft repräsentieren.

Die Basis des C#-Schemas

Als Grundlage des C#-Schemas wird das in Abbildung 3.1 abgebildete Java-Schema dienen. C# und Java haben einen ähnlichen Aufbau. Beide Sprachen wurden nach objektorientierten Paradigmen entworfen. Dadurch gibt es eine klare Typhierarchie, die durch die Generalisierung *Type* abgebildet wird. In Java werden diese Typen in *Packages* deklariert und können durch Klassen oder *Interfaces* ausgeprägt sein. Klassen können, wie in der Darstellung Abbildung 3.1 abgebildet, *Member* und Methoden enthalten sowie Schnittstellen implementieren und von anderen Klassen abstammen. Damit bildet diese Hierarchie die grundlegenden Elemente sowie das Grundkonzept der Sprache C# ab.

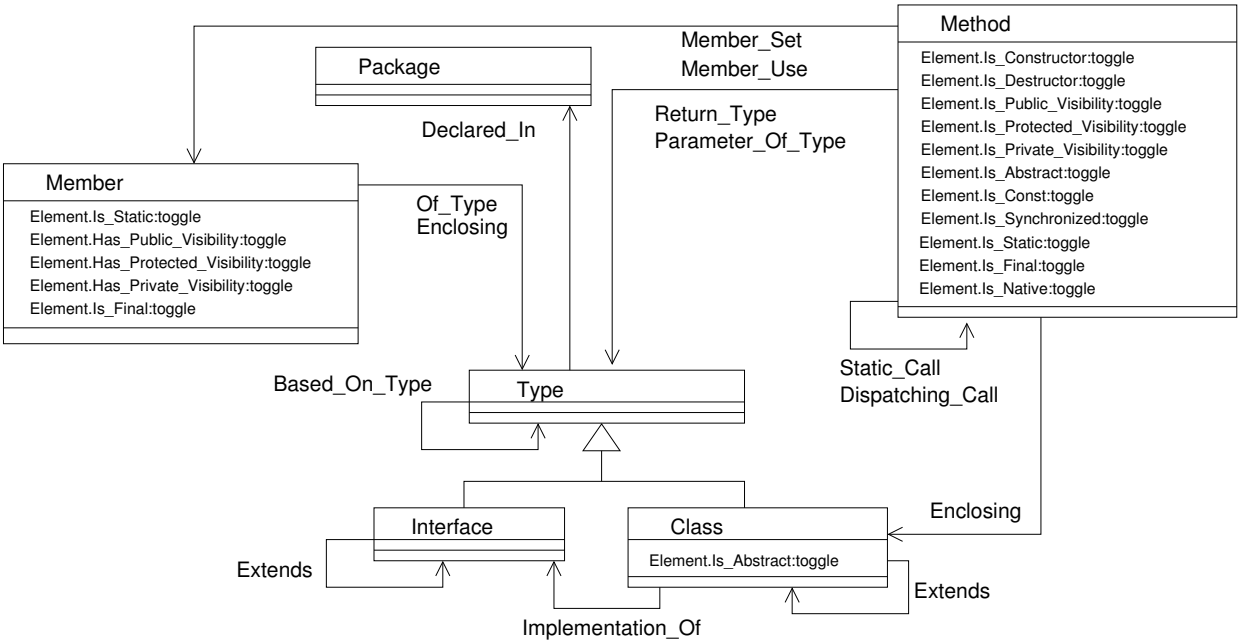

Abbildung 3.1: Das RFG-Schema für Java

Leider reicht das Java-Schema nicht gänzlich um den Sprachumfang von C# abzubilden. Aus diesem Grund werden einige Konzepte aus dem im Abschnitt 2.4.1 schon vorgestellten C++-Schema übernommen. Diese Konzepte erweitern das verwendete Basisschema oder ersetzen gegenebenfalls einige Teile. Beispielsweise ersetzt die Organisation in *Namespaces* die Deklaration in *Packages*. Ähnlich verhält es sich mit den Attributen der Knoten, beispielsweise für Typen, die dann mit den Attributen *Element.Is_Enum* näher spezifiziert werden können. Schemaerweiterungen durch Attribute stellen kein Problem dar. Sie sind unabhängig von den sprachunabhängigen Analysen des RFG. Aus diesem Grund ist es problemlos möglich, das Schema an dieser Stelle zu erweitern ohne dabei unerwünschte Seiteneffekte hervorzurufen.

Im Folgenden werden die wiederverwendeten Teile der bestehenden RFG-Schemata beschrieben und wie sie sich in ein Schema für die Sprache C# zusammensetzen lassen. Zusätzlich werde ich auch Regeln beschreiben, wie einige Sprachkonzepte abgebildet werden sollen. Letztlich werden auch hinzugefügte Attribute beschrieben, die das Programmverstehen von C#-Programmen zusätzlich unterstützen sollen. Für das in diesem Kapitel beschriebene Schema gelten die in Abschnitt 2.4.1 beschriebenen Paradigmen.

3.2 RFG-Schema für C#

Ein großer Teil der in Abschnitt 2.3 vorgestellten Konzepte von C# ist bereits im RFG-Modell vorhanden. In diesem Abschnitt wird untersucht in wie weit sich die vorhanden Konzepte des RFG auf die Sprachelemente von C# abbilden lassen und ob gegebenenfalls Erweiterungen des Modells notwendig sind. Zudem gilt festzulegen welche Teile der C#-Programme relevant für den RFG sind.

Diese Festlegung ist relativ einfach. Alle Deklarationen, die innerhalb von Methoden stattfinden sind grundsätzlich lokal und werden nicht abgebildet. Als Entitäten können also nur Programmelemente dienen, die nicht lokal sind und damit global sein können. Im Folgenden sollen die noch relevanten Bestandteile von C#-Programmen einzeln betrachtet und die Möglichkeiten ihrer Abbildung auf den RFG diskutiert werden. Am Ende jedes Unterabschnitts werden alle erarbeiten Abbildungen von C# auf den RFG in Form einer Tabelle abschließend zusammengefasst. Bei dieser Auflistung werden Neuerungen im RFG-Modell gesondert ausgezeichnet.

3.2.1 Namespaces

C#-Programme werden, wie in Abschnitt 2.3.2 beschrieben, in *Namespaces* organisiert. Deswegen wird im Java-Schema der Knotentyp *Package* zur Programmstrukturierung gegen den Knoten *Namespace* aus dem C++-Schema ausgetauscht. Die dazugehörige Kante *Declared_In* wird ebenfalls durch eine *Enclosing-Kante* ersetzt, die abbildet welcher Typ in einem *Namespace* enthalten ist. Damit geschachtelte Namespaces mit dem Schema

abgebildet werden, wird eine weitere *Enclosing*-Kante auf den *Namespace*-Knoten hinzugefügt.

C#	→	**RFG**	**Neu**
Organisation in Namensräumen		*Namespace*-Knoten	
Namensraumzugehörigkeit eines Typs		*Enclosing*-Kante	
Geschachtelte Namensräume		*Enclosing*-Kante	

3.2.2 Typen

Wie im Java-RFG sind Typen als Generalisierung von Schnittstellen und Klassen zu verstehen. Der Aufzählungstyp *Enum* wird im C++-Schema mit dem Attribut *Is_Enum* dargestellt wird. Dieses Attribut wird für Aufzählungen in C# übernommen.

Neben Aufzählungstypen werden auch alle nicht klar definierbaren Typen als *Type* dargestellt. Dieses Konzept kann bei referenzierten Typen aus anderen Softwarekomponenten Anwendung finden, denn diese werden als fertige abgeschlossene Objekte referenziert und können nicht als Klasse, *Struct* oder *Interface* identifiziert werden. Dazu zählen auch referenzierte DLL-Objekte, die wie in Abschnitt 2.3.10 beschrieben, Methoden zur Verwendung bereitstellen.

C#	→	**RFG**	**Neu**
Enum		*Type*-Knoten mit *Enum*-Attribut	
Nicht klar definierbare Typen		*Type*-Knoten	

3.2.3 Klassen und Schnittstellen

In Abschnitt 2.3.1 wurde darauf hingewiesen, dass es möglich ist Objekte im Quelltext partiell zu definieren. Ungeachtet dessen werden diese Objekte im RFG als ein Objekt betrachtet.

Klassen und Schnittstellen werden entsprechend zum Java-Schema als Ableitung von *Type* modelliert und übernommen. *Structs* können wie Klassen eine Schnittstelle implementieren, deswegen wird ein *Struct* entgegen der Modellierung von *Stucts* im C++-Schema als Klasse mit dem Attribut *Is_Struct* abgebildet.

Die weitere Ableitung einer Klasse wird in C# mit einer *Sealed*-Deklaration unterbunden. Diese Eigenschaft ist vergleichbar mit dem *Final*-Attribut in der Java-Programmierung, so dass im RFG das vorhandene *Is_Final*-Attribut für die Abbildung der *Sealed*-Eigenschaft verwendet wird. Neben dem ausgeprägten Polymorphismus ist es auch möglich Klassen ohne eine explizite Instanzbildung zu benutzen. Diese statischen Klassen werden als *Class*-Knoten mit einem *Is_Static*-Attribut abgebildet. Eine weitere Klassenform von der keine Instanz gebildet werden kann, sind abstrakte Klassen. Sie stellen partielle

Implementierungen bereit, die durch das Vererben an eine andere Klasse konkret ausgeprägt werden können. Dieser Klassentypus kann im RFG von anderen Klassen mittels *Is_Abstract*-Attribut unterschieden werden. Beide Attribute sind schon bei den Methodenattributen vorhanden und werden für den *Class*-Knoten übernommen.

Klassen können Klassenvariablen *Member* enthalten, sowie Methoden und *Events*. Zusätzlich können aber auch wiederum weitere Klassen in Klassen enthalten sein (*Nested Types*). Die Klassenzugehörigkeit dieser Elemente wird mit der *Enclosing*-Kante ausgedrückt. Diese Typschachtelung ermöglicht unter anderem eine explizite Verwendung von Sichtbarkeitsattributen. Klassen können wie in Abschnitt 2.3.3 beschrieben vier Sichtbarkeiten im Programm haben. Deswegen werden für den *Class*-Knoten die schon bestehenden Sichtbarkeitsattribute der Methoden übernommen. Damit die Sichtbarkeitsbeschränkung auf ein Assembly und auf abgeleitete Typen ausgedrückt werden kann, werden die Attribute *Has_Internal_Visibility* und *Has_Protected_Internal_Visibility* hinzugefügt. Die Sichtbarkeit *Protected Internal* wird nicht als Kombination der Attribute *Has_Internal_Visibility* und *Has_Protected_Visibility* abgebildet, weil die Sichtbarkeit von Klassen im RFG jeweils mit einem Icon an einer bestimmten Stelle des Knoten dargestellt wird (vgl. Abbildung 3.2).

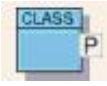

Abbildung 3.2: Ein Klassenicon mit visualisiertem Sichtbarkeitsattribut.

In Abschnitt 2.3.6 wurde das Delegaten-Konzept beschrieben. Diese spezielle Klasseform enthält die eine Aufrufliste für Methoden. Diese Methoden werden aufgerufen, sobald der Delegat aufgerufen wird. Für dieses Konzept macht es keinen Sinn einen eigenen Knotentyp zu verwenden, da eine explizite Klassenähnlichkeit gegeben ist. Aus diesem Grund werden Delegaten als *Class* modelliert und erhalten das Attribut *Is_Delegate*, damit sie von Klassen unterscheidbar sind.

C# →	**RFG**	**Neu**
Interface	*Interface*-Knoten	
Klasse	*Class*-Knoten	
Struct	*Class*-Knoten mit *Is_Struct*-Attribut	
Delegat	*Class*-Knoten mit *Is_Delegate*-Attribut	+
Sichtbarkeit von Klassen	*Has_..._Visibility*-Attribut	
Sichtbarkeit innerhalb eines Assemblys	*Internal_Visibility*-Attribut	+
Sichtbarkeit innerhalb eines Assemblys und beschränkt auf abgeleiteten Typ	*Has_Protected_Internal_Visibility*-Attribut	+
Nicht weiter ableitbare Klassen	*Is_Final*-Attribut	

Statische Klassen	*Is_Static*-Attribut
Abstrakte Klassen	*Is_Abstract*-Attribut
Interface-Implementierung	*Implementation_Of*-Kante
Klassenvererbung	*Extends*-Kante
Geschachtelte Klasse	*Enclosing*-Kante

3.2.4 Klassenvariablen

Variablen und Konstanten lassen sich in C# nur als Klassenvariablen (*Member*) verwenden. Sie werden innerhalb eines Objektes deklariert und werden analog zum Java-RFG als *Member* abgebildet. Dazu gehört die *Enclosing*-Kante, die zeigt in welchem Typ der *Member* deklariert wurde, sowie die *Of_Type*-Kante, die den Typ des *Members* näher spezifiziert. Konstanten können von Variablen anhand des Attributes *Is_Const* unterschieden werden.

Sie können von Methoden ausgelesen, gesetzt oder über ihre Adresse angesprochen werden. Darüber hinaus ist es möglich, dass *Member* von Typen geerbt werden oder diese überschreiben. Diese Konstrukte werden mit einer *Inherits*- und *Overrides*-Kante zwischen zwei *Membern* abgebildet. Auf statische *Member* kann ohne eine Instanz des deklarierenden Typen zugegriffen werden. Diese Eigenschaft wird mit dem Attribut *Is_Static* ausgedrückt.

Member können mit verschiedenen Sichtbarkeitsmodifizierern (siehe Abschnitt 2.3.3) versehen werden. Aus diesem Grund werden wie bei Klassen die Attribute *Has_Internal_Visibility* und *Has_Protected_Internal_Visibility* zu den vorhandenen Sichtbarkeitsattributen hinzugefügt.

Die Laufzeitumgebung bietet die Möglichkeit Programme mit verschiedenen Threads[1] laufen zu lassen. Wenn ein Member von mehreren Threads benutzt wird, können die enthaltenen Daten durch unkontrollierten Threadzugriff inkonsistent werden. Um dieses Verhalten zu vermeiden, kann ein Member mit dem Schlüsselwort *Volatile* versehen werden. Hierbei wird dann bei der Optimierung der Zwischendarstellung darauf geachtet, dass ein Thread erst wieder lesend oder schreiben auf den Member zugreifen darf, wenn dieser vom vorherigen Thread geschrieben wurde. Diese Eigenschaft wird mit dem Attribut *Is_Volatile* abgebildet.

C# →	**RFG**	**Neu**
Variable oder Konstante innerhalb einer Typdefinition	*Member*-Knoten	
Variable oder Konstante definiert in Typ	*Enclosing*-Kante	
Typ der Variable oder Konstante	*Of_Type*-Kante	
Eine Konstante	*Is_Const*-Attribut	
Sichtbarkeit von Membern	*Has_..._Visibility*-Attribut	

[1] Ein Thread ist Teil eines Prozesses, der sich mit anderen Threads innerhalb eines Prozesses Codesegmente teilt.

Sichtbarkeit innerhalb eines Assemblys	*Has_Internal_Visibility*-Attribut	+
Sichtbarkeit innerhalb eines Assemblys und beschränkt auf abgeleiteten Typ	*Has_Protected_Internal_Visibility*-Attribut	+
Statische Member, die ohne eine Instanz des Typs verwendet werden können	*Is_Static*-Attribut	
Threadsichere Member	*Is_Volatile*-Attribut	
Die Zugehörigkeit von Membern	*Enclosing*-Kante	
Vererbung von Membern	*Inherits*-Kante	
Überschreiben von geerbten Membern	*Overrides*-Kante	

3.2.5 Methoden

C#-Methoden werden analog zu den im Java-RFG definierten Methoden abgebildet, weil sie die gleichen Eigenschaften aufweisen. Zum Beispiel können statische Methoden ohne eine Instanz ihres deklarierenden Typs benutzt werden oder bei der Verwendung mehrere Threads synchronisiert werden. Die Attribute und Kanten der Methoden können direkt aus dem Java-RFG übernommen werden. Dennoch reicht die Darstellung der Methoden im Java-RFG nicht vollständig für die Abbildung der C#-Methoden, so dass noch Attribute ergänzt werden müssen.

Zu den vorhandenen Methodensichtbarkeiten kommt analog zu den Klassensichtbarkeiten noch das *Has_Internal_visibility*- und *Has_Protected_Internal_Visibility*-Attribut hinzu.

Wie schon im Abschnitt 2.3.5 veranschaulicht wurde, können Methoden als *virtual* deklariert werden. Dies ermöglicht ein explizites Überschreiben und Neudefinieren von geerbten Methoden. Dies Eigenschaft wird im RFG-Modell mit dem Attribut *Is_Virtual* abgebildet. Um die Beziehungen zwischen den sich überschreibenden oder geerbten Methoden abzubilden werden die Kanten *Overrides* und *Inherits* verwendet. In C# können wie schon angesprochen geerbte virtuelle Methoden auch neudefiniert werden. Die Neudefinition unterscheidet sich nur bei einer expliziten Typumwandlung zum Basistyp der abgeleiteten Klasse vom Überschreiben der Methode (siehe Listing 2.4. Im Prinzip überschreiben beide Schlüsselwörter die vorherige Definition, nur das bei einem *Override* nicht mehr die Methodendefinition der Basisklasse benutzt werden kann. Aus diesem Grund wird für die Neudefinition einer virtuellen Methode keine neue Kante eingeführt, da nur das überschreiben abgebildet wird und nicht die unterschiedliche Ausführung der Instanzmethoden.

Im Abschnitt 2.3.10 habe ich beschrieben, wie Methoden von COM-Objekten als C#-Methoden definiert und verwendet werden können. Damit klar dargestellt wird, das diese Methode nicht im vorliegenden Quelltext definiert wurde, sondern extern, wird eine solche Methode mit dem Attribut *Is_Extern* versehen.

In C# kann wie in Abschnitt 2.3.1 gezeigt innerhalb eines *Unsafe*-Blocks mit Zeigern gearbeitet werden. Daraus ergibt sich als Zusatz zu den *Member_Set*- und *Member_Use*-Kanten zur Modifikation von *Member*, die *Member_Address*-Kante. Damit werden Zeigeroperationen innerhalb einer Methode auf eine Klassenvariable modelliert.

Das in dieser Arbeit entwickelte Werkzeug basiert auf der Analyse der Zwischendarstellung von C#-Programmen. Dies impliziert, das der Quelltext schon einmal von einem Compiler verarbeitet wurde. Dadurch können auch Methoden erzeugt werden, die im Quelltext nicht definiert wurden. Damit diese Methoden beim Programmverstehen nicht zu Problemen führen, werden diese Elemente mit dem Attribut *Element.Is_Artificial* ausgezeichnet.

Methodenaufrufe werden im RFG mit drei Konzepten modelliert. Jedes Konzept bildet damit eine spezielle Form des Bindens ab.

- *Static_Call*
  Bei statischen Aufrufen von Methoden steht schon während der Übersetzungzeit fest, welche Methode aufgerufen werden soll. Es entspricht einem frühen Binden.
- *Dynamic_Call*
  Ein dynamischer Aufruf modelliert das Aufrufen von Methoden mit Hilfe eines Funktionszeigers. Hierbei ist der Aufruf laufzeitabhängig von einem Zeiger.
- *Dispatching_Call*
  Dieser Aufruf modelliert Aufrufe von Funktionen innerhalb einer Vererbungshierarchie.

Für das C#-Schema werden *static*- wie auch *Dispatching*-Calls verwendet. Es C# gibt zwar die Möglichkeit Zeiger einzusetzen, allerdings gilt dies nur für Variablen. Als Alternative zu dynamischen Methodenaufrufen gibt das in Abschnitt 2.3.6 beschrieben Delegatenkonzept.

C# →	**RFG**	**Neu**
Methode	*Method*-Knoten	
Methode innerhalb einer Typdefinition	*Enclosing*-Kante	
Methode ist ein Konstruktor/Dekonstruktor	*Is_Constuctor*/*Is_Deconstructor*-Attribut	
Parameter der Methode	*Parameter_Of_Type*-Kante	
Rückgabewert der Methode	*Return_Type*-Kante	
Virtuelle Methoden	*Is_Virtual*-Attribut	+
Extern deklarierte Methoden	*Is_Extern*-Attribut	+
Manipulation von *Membern*	*Member_Address*-/*Member_Set*-/*Member_Use*-Kante	
Aufruf anderer Methoden	*Dispatching_Call*- und *Static_Call*-Kante	
Von der Laufzeitumgebung erzeugte Methoden	*Is_Artificial*-Attribut	+

Statische Methoden	*Is_Static*-Attribut	
Versiegelte Methoden	*Is_Final*-Attribut	
Threadsynchronisierte Methoden	*Is_Synchronized*-Attribut	
Sichtbarkeit innerhalb eines Assemblys	*Has_Internal_Visibility*-Attribut	+
Sichtbarkeit innerhalb eines Assemblys und beschränkt auf abgeleiteten Typ	*Has_Protected_Internal_Visibility*-Attribut	+

3.2.6 Propertys und Indexer

Wie schon in Abschnitt 2.3.9 beschrieben bilden Propertys und Indexer jeweils ein besonderes Konstrukt. Betrachtet man die Verwendung eines Propertys beim Programmieren, dann verhält es sich wie eine öffentliche Variable. Analog dazu der Indexer, der wie ein öffentliches Array verwendet wird. Eigentlich sind beide Konstrukte nur Hüllen für gekapselte Zugriffe auf private Variablen. Im Hinblick auf ein konsistentes Programmverstehen beim Einsatz der beiden Konstrukte wurde von einer separaten Modellierung als eigener Knoten abgesehen. Propertys und Indexer werden im RFG als *Member* im RFG abgebildet und erhalten als Unterscheidungsmerkmal zu normalen Variablen das Attribut *Is_Property*/*Is_Indexer*.

Zu Propertys und Indexern gehören jeweils Get- und Set-Methoden, welche die konkreten Manipulationen auf dem privaten *Member* definieren. Diese Methoden werden mit dem Attribut *Is_Property_Accessor* bzw. *Is_Indexer_Accessor* gekennzeichnet, damit der Zusammenhang zwischen Property/Indexer und den dazu definierten Methoden eindeutig ist. Die genaue Funktion einer Get- oder Set-Methode wird durch die Kanten *Member_Set* und *Member_Use* abgebildet.

Propertys und Indexer können auch als abstrakte und virtuelle *Member* definiert werden. Ein abstrakter Property/Indexer wird später erst implementiert. Ähnlich zu virtuellen Methoden können virtuelle Propertys/Indexer überschrieben und neudefiniert werden. Daraus ergibt sich die Notwendigkeit von speziellen Attributierungen, die diese Eigenschaften aufzeigen. Dazu werden die schon bei Methoden eingesetzten Attribute *Is_Virtual* und *Is_Abstract* verwendet. Zugleich gilt auch die Definition für *sealed* Methoden die mit dem *Is_Final*-Attribut gekennzeichnet werden.

C# →	**RFG**	**Neu**
Property	*Member*-Knoten mit *Is_Property*-Attribut	+
Indexer	*Member*-Knoten mit *Is_Indexer*-Attribut	+
Get- und Set-Methoden eines Propertys	*Methoden*-Knoten mit *Is_Property_Accessor*-Attribut	+

Get- und Set-Methoden eines Indexer	*Methoden*-Knoten mit *Is_Indexer_Accessor*-Attribut	+
Virtueller Property/Indexer	*Is_Virtual*-Attribut	+
Abstrakter Property/Indexer	*Is_Abstract*-Attribut	
Versiegelte Property/Indexer	*Is_Final*-Attribut	

3.2.7 Events

Die in C# verwendbare Programmsteuerung mittels Ereignissen wird ähnlich zum Event-Konzept des Visual Basic 6 RFG modelliert. Sprachbedingt gibt es kleine Unterschiede, so dass die Kanten und Knoten wiederverwendet werden können, aber das Konzept leicht abweicht. Dies liegt unter anderem an der Ereignisdefinition mittels Delegaten und ihrem expliziten Konzept des Abonnierens.

Die in Abschnitt 2.3.7 vorgestellten Events bestehen zusammengefasst aus folgenden Konstrukten:

- Die Eventdeklaration innerhalb eines Typen
- Einen Delegat, der die Signatur des Events spezifiziert
- Eine Instanz der Klasse, die das Event definiert
- Eine Methode, deren Signatur dem Delegaten entspricht, um das Event zu behandeln
- Eine Methode, die das Event auslöst

Events sind maßgeblich an der Programmsteuerung beteiligt, wenn sie in einem Programm definiert werden. Deswegen ist ihre Abbildung im RFG sehr interessant für die Erstellung einer Call-View. Denn Events können, wenn sie ausgelöst werden, mehrere Abonnenten haben, die dann mit spezifischen Methoden auf das Event reagieren, es verarbeiten und dabei wiederum weitere Methoden im Programmablauf anstoßen können. Das *Event*-Konstrukt ist nicht vergleichbar mit einem *Member*, obwohl es auch Bestandteil einer Klasse ist. Events können nur abonniert und abbestellt werden. Zugriffe wie *Set*- und *Use*-Kanten würden diesem Vorgang nicht gerecht werden. Angenommen der Abonnentenvorgang würde mit zwei *Set*-Kanten für das Abonnieren und Abbestellen abgebildet werden, so fehlte die Klarheit über den konkreten Programmvorgang und das Konzept wäre für Benutzer nicht schlüssig. Aus diesem Grund sehe ich von einer Event-Modellierung als *Member* mit einem ausweisenden Attribut ab und werde abgeleitet von einer *Set*-Kante die Kanten *Event_Register* und *Event_Unregister* zur Abbildung des Abonnentenprinzips verwenden.

Ereignisse haben, anders als in Visual Basis 6, eine konkrete Typzugehörigkeit. Dieser Typ ist immer eine Ableitung von *System.MulticastDelegate* und somit ein Delegat. Die

Typzugehörigkeit eines Events wird mittels *Of_Type*-Kante vom Event zum *Type*-Knoten abgebildet. Die Eventdeklaration innerhalb einen Typs wird ähnlich zu *Membern* mit der *Enclosing*-Kante abgebildet.

Eine ereignisbehandelnde Methode behandelt immer das Ereignis eines bestimmten *Member*. Diese Beziehung wird mit der Kante *Handles_Event_Of* von der behandelnden Methode zu einem *Member* abgebildet.

Die benötigten Kanten und Knoten zur Modellierung sind damit festgelegt. Die Abbildung 3.3 zeigt das Teilschema zur Event-Modellierung. Eine Methode, die ein Event

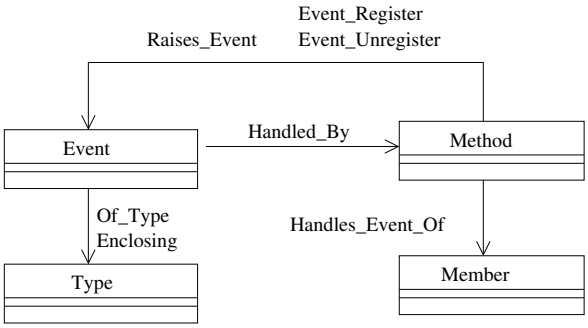

Abbildung 3.3: Teilschema zur Eventmodellierung

auslöst wird mit der *Raises_Event*-Kante angezeigt. Dazu gehört, wenn das Event abonniert wurde, eine Methode, die das Event behandelt. Solche Methoden werden mit der *Handled_By*-Kante vom Event zur Methode abgebildet. Beide Kanten wurden im Visual Basic 6 RFG von einer *Call*-Kante abgeleitet und bilden einen Aufrufkreislauf ab, so dass Aufrufe anhand der Kanten eindeutig verfolgt werden können.

Events können, wie andere Objekte der Sprache C#, verschiedene Sichtbarkeiten und weitere Attribute haben. Die Attribute wurden schon bei den Propertys vorgestellt, deswegen werde ich hier nicht weiter auf sie eingehen, sie aber in der Zusammenfassungtabelle aufführen.

C# →	**RFG**	**Neu**
Event	Event-Knoten	
Typzugehörigkeit des Ereignisses	*Of_Type*-Kante	
Typ in dem das Ereignis deklariert ist	*Enclosing*-Kante	
Ereignisauslösung	*Raises_Event*-Kante	
Eine Methode behandelt ein Event	*Handled_By*-Kante	
Eine Methode behandelt das Event einer Klasseninstanz	*Handles_Event_Of*-Kante	
Abonnieren eines Events	*Event_Register*-Kante	+
Abbestellen eines Events	*Event_Unregister*-Kante	+
Abstraktes Ereignis	*Is_Abstract*-Attribut	
Virtuelles Ereignis	*Is_Virtual*-Attribut	

Statisches Ereignis	*Is_Static*-Attribut	
Versiegeltes Ereignis	*Is_Final*-Attribut	
Sichtbarkeit des Ereignisses	*Has_. . . _Visibility*-Attribut	
Sichtbarkeit innerhalb eines Assemblys	*Has_Internal_Visibility*-Attribut	+
Sichtbarkeit innerhalb eines Assemblys	*Has_Protected_Internal_Visibility*-Attribut	+

3.2.8 Generics

Generische Objekte sind ein fester Bestandteil von C#. Die es ermöglichen parametrisierte Klassen, Schnittstellen, *Structs*, Methoden und Delegaten zu definieren. Ähnlich zu *Templates* in C++, werden sie mit einem Platzhalter für die spätere Typisierung versehen. Im Gegensatz zu C++ wird aber schon bei der Kompilierung des C#-Quelltextes auf eine zulässige Typisierung geprüft. Ein weiterer Unterschied zu C++-*Templates* ist auch der Instanziierungszeitpunkt von C++-*Templates* und *Generics*. C#-*Generics* werden zur Laufzeit erzeugt und mit einem Typen besetzt. Dies ermöglicht Speicherwiederverwendung, wenn ein *Generic* mit mehreren Typen im Programmablauf genutzt wird. Dies ist konträr zu C++-*Templates*, die schon während der Kompilierung oder zur Linkzeit instanziiert werden und somit keine gemeinsame Speichernutzung für C++-*Templates* möglich wird. Ungeachtet der kleinen Unterschiede werde ich die Modellierung der C++-*Templates* aus dem C++-RFG für die Abbildung der C#-*Generics* verwenden.

Abbildung 3.4: Icon für generische Schnittstellen

Die einzeln ausgeprägten Templates *Method*- und *Class*-Templates werden wie im C++-Schema vom Knoten *Template* abgeleitet. Zusätzliche können sie innerhalb von *Namespaces* definiert sein. Neben generischen Klassen auch möglich generische *Structs* zu definieren. Diese werden analog zu dem schon in Abschnitt 3.2.3 beschriebenen Klassenkonzept als *Class_Templates* mit dem *Is_Struct*-Attribut abgebildet, sowie ein Delegat mit dem *Is_Delegate*-Attribut. Zu den von *Template* abgeleiteten Knoten kommt der Knoten *Interface_Template* hinzu, dargestellt in Abbildung 3.4. Er repräsentiert die generische Schnittstelle in C#. Sie verhalten sich ähnlich zu normalen Schnittstellen und erhalten zur besseren Repräsentation einen eigenen Knoten, denn mit dem *Class_Template*-Knoten würden sie nicht ausreichend repräsentiert werden.

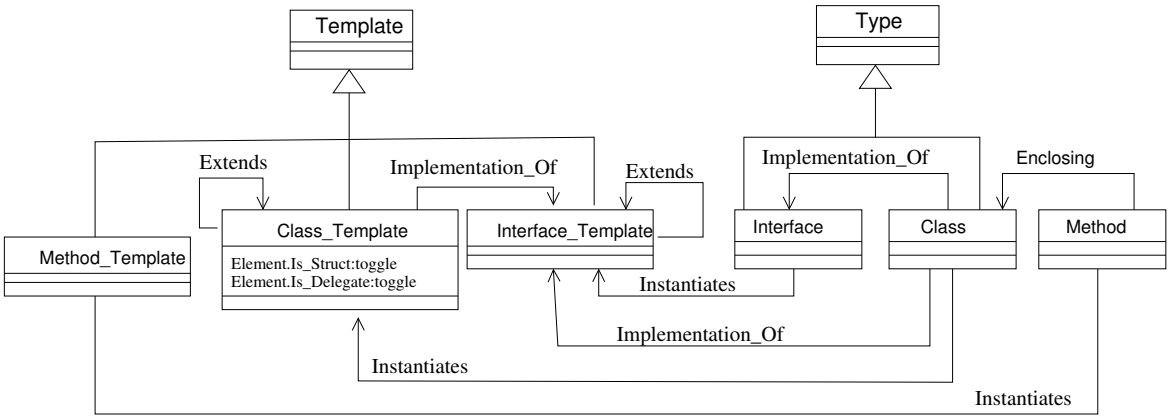

Abbildung 3.5: Teilschema für Generics

Um dem Polymorphismus in C# gerecht zu werden, werden bei den Knoten *Interface_Template* und *Class_Template* entsprechende *Extends*-Kanten hinzugefügt (siehe Abbildung 3.5). Hierbei gilt wie bei Typen ein *Class_Template* kann ein *Interface_Template* implementieren. Beide *Template*-Ableitungen können durch Klassen oder *Interfaces* instanziiert werden, wobei eine Klasse auch ein *Interface_Template* implementieren kann. *Method_Templates* werden durch die konkrete Typisierung einer Methode instanziiert. Somit ergibt sich für die Genericmodellierung das in der Abbildung 3.5 dargestellte Teilschema.

C# →	**RFG**	**Neu**
Generalisierung der drei Templates	*Template*-Knoten	
Generische Klasse	*Class_Template*-Knoten	
Generisches *Struct*	*Class_Template*-Knoten mit *Is_Struct*-Attribut	
Generisches Interface	*Interface_Template*-Knoten	+
Generische Methode	*Methoden_Template*-Knoten	
Vererbung innerhalb der Templates	*Extends*-Kante	
Implementierung eines Interface-Templates	*Implementation_Of*-Kante	
Instantiierung eines Templates	*Instantiates*	

3.2.9 Das vollständige RFG-Schema für C#

Nach den vorgestellten Änderungen und Abbildungskonventionen für das C#-Schema wird das vollständige Schema in Abbildung 3.6 dargestellt. Wenn man dieses Schema mit den existierenden Schemata vergleicht, bildet es eine Art Hybrid-Schema aus dem Java- und C++-RFG mit wenigen Auszügen des Visual Basis 6 Schemas. Dies Unterstreicht die in Abschnitt 2.3 erwähnten Entwurfsprinzipien der Sprache und die Fortführung des Ereigniskonzeptes, der vor dem Erscheinen des .NET-Framework hauptsächlich für die Windows-Applikationsentwicklung verwendete Sprache Visual Basic 6.

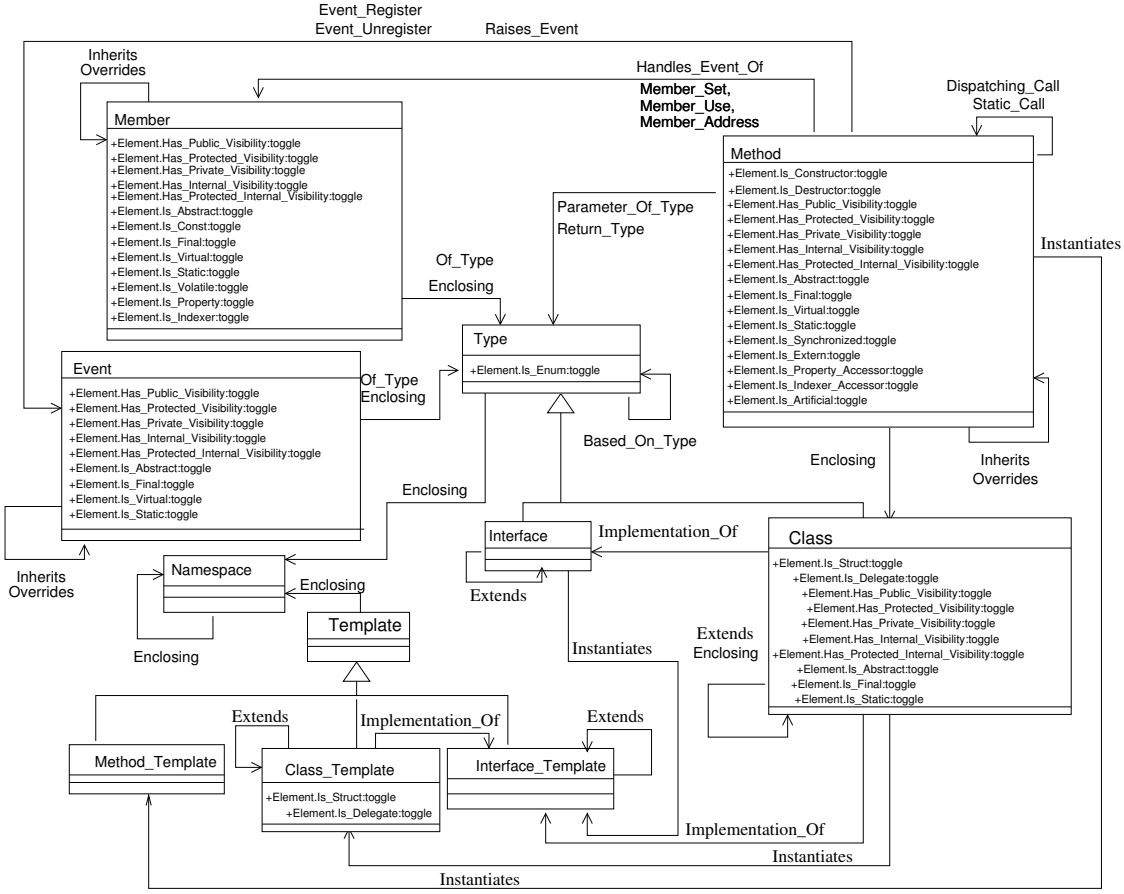

Abbildung 3.6: Das RFG-Schema für C#

Für die Modellierung der Sprachaspekte wurden lediglich zwei neue Kanten für die Modellierung des Ereignis-Abonnieren-Prinzips, sowie ein neuer Knotentyp für die Abbildung von generische Schnittstellen definiert. Weiterhin wurden sieben neue Attribute hinzugefügt. Die Möglichkeit dem RFG dynamisch neue Attribute hinzuzufügen ohne sie explizit zu deklarieren, machen diese Änderungen unerheblich. Diese Attribute haben einen rein informativen Charakter und keinen wesentlichen Effekt auf das RFG-Modell. Alle weiteren verwendeten Element waren bereits vorher im RFG-Modell enthalten und konnten daher wieder aufgegriffen werden.

3.3 Anforderungen an die Analyse von C#

Das definierte RFG-Schema für C#-Programme gibt einen Überblick welche Elemente der Sprache und ihrer Semantik abgebildet werden sollen. Daraus können nun konkrete Anforderungen an die Analyse von C# gestellt werden.

- **Analyse der im Programm deklarierten Elemente**

Die Knoten des RFG ergeben sich aus den im Programm deklarierten Elementen. Hierzu zählen alle globalen Elemente, dies schließt auch Elemente mit einer lokal beschränkten Sichtbarkeit ein. Jedoch werden Elemente, die im Kontext einer Methode deklariert werden, vernachlässigt. Die Zugehörigkeiten von Elementen zu Klassen und anderen Typen sind ebenfalls zu ermitteln. Außerdem müssen elementspezifische Informationen, die als Attribute abgebildet werden sollen, ermittelt werden.

- **Auflösung der Element-Beziehungen**
  Zusätzlich zur Analyse der Sprachelemente müssen auch die Beziehungen zwischen den RFG-Knoten gewonnen werden. Dies beinhaltet die Ermittlung der Typzugehörigkeiten und Vererbungshierarchie. Mit diesen Informationen können die entsprechenden Kanten zwischen Typen, Membern, Methoden und Events abgebildet werden. Hierbei muss auch die Instanzbildung von generischen Typen identifiziert werden.

- **Auflösung von Aufrufen, Variablenverwendungen und Ereignisbehandlung**
  Nachdem die syntaktischen Informationen vorliegen können die semantischen Aspekte im RFG abgebildet werden. Diese Informationen werden aus den Methoden gewonnen. Hierbei werden nur Aufrufe von anderen Methoden, und die Manipulationen von *Membern* betrachtet. Daraus ergibt sich, dass die Verarbeitung von lokalen Variablen von der Verarbeitung von Klassenvariablen und Methodenaufrufen separiert werden muss. Desweiteren muss analysiert werden welche Manipulation von *Membern* durchgeführt wird, damit die entsprechende Kante gesetzt werden kann. Bei der Ereignisbehandlung muss zudem zwischen den abonnierenden, auslösenden und behandelnden Methoden unterschieden werden. Zu dem muss analysiert werden zu welchem *Member* das Ereignis gehört.

- **Identifizierung externer Methoden**
  Extern deklarierte Methoden müssen identifiziert und dementsprechend ausgewiesen werden.

- **Gewinnung von Quelltextpositionen**
  Eine implizite Anforderung bei der Analyse ist die Gewinnung von Quelltextpostitionen. Sie hat konkret nichts mit der Konzeption des RFGs zu tun, ist aber, wie in Abschnitt 2.4.1 beschrieben, ein wichtiger Bestandteil für die spätere Visualisierung mit dem Programm *Gravis*.

KAPITEL 4

Analyse

Bevor der Resource Flow Graph generiert werden kann, müssen zunächst die benötigten Informationen gewonnen werden. Gegenstand der Analyse ist hierbei nicht der Quelltext, sondern das daraus kompilierte Erzeugnis. Dazu wird die aus der in Abschnitt 2.3 vorgestellte Programmiersprache C# erzeugte Zwischendarstellung analysiert, dargestellt in Abbildung 4.1.

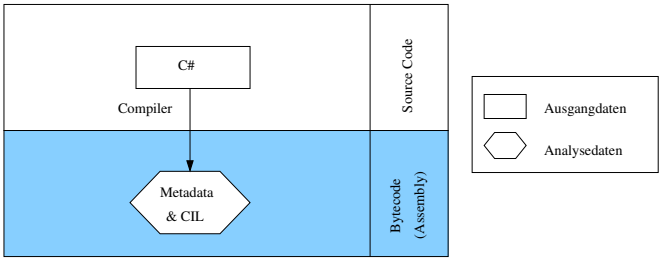

Abbildung 4.1: Aufgabenbereich der Analyse

Zu Anfang wird in Abschnitt 4.1 allgemein die Darstellungsform eines Assemblys beschrieben. So ein Kompilat enthält nicht nur Metadaten und CIL-Code. Deswegen wird in Abschnitt 4.2 der komplette Aufbau eines Assemblys vorgestellt. Die Analyse des Aufbaus ist maßgeblich wichtig damit später die Zwischendarstellung der Quelltextinformationen im Kompilat gefunden wird. Der Weg zu diesen Informationen wird im Abschnitt 4.2 beschrieben.

Im Anschluss wird die Zwischendarstellung der im Quelltext definierten Elemente, die Metadaten, werden im Abschnitt 4.3 gesondert betrachtet. Metadaten haben eine in sich komplexe Struktur zur Abbildung der Quelltextinformationen. Die Analyse der Datenstruktur erfolgt anhand der aus Abschnitt 3.3 gewonnenen Analyseschwerpunkte zur Erstellung des RFG.

Kapitelinhalt

4.1 Assemblys als spezielle Form der *Portable Executable Images* . 56

4.2 Analyse eines Assembly (*Portable Executable Image* mit CLI-Erweiterung) 58

4.2.1 File Header 58

4.2.2 CLI Header 60
4.2.3 CLI Data 61
4.3 Analyse der Metadaten Tabellen 64
4.3.1 Dekodierung von Informationen innerhalb der Metadaten Tabellen . 65
4.3.2 Beschreibung der Metadaten Tabellen 68
4.3.3 Analyse des CIL-Codes 83
4.4 Gewinnung von Quelltextpositionen 86
4.4.1 Quelltextpositionsgewinnung mit Debuginformationen 86

4.1 Assemblys als spezielle Form der *Portable Executable Images*

Im Abschnitt 2.1 wurden schon kurz Assemblys als Informationsträger der Zwischendarstellung vorgestellt. Sie werden auch als *Portable Executable (PE) Image* mit CLI-Erweiterung bezeichnet.

PE ist ein Dateiformat das mit Windows NT 3.1 eingeführt wurde. Seit dieser Zeit unterstützen alle späteren Windows-Betriebssysteme dieses Dateiformat. PEs bilden Bibliotheken oder ausführbare Dateien. Das Dateiformat enthält alle Datenstrukturen, die zur Programmausführung benötigt werden. Dazu gehören Referenzen über verwendete Bibliotheken, Ressourcenmanagement und der kompilierte Programmcode. Die enthaltenen Datenstrukturen werden als *Image* bezeichnet.

Sie sind der „Auslieferzustand“ für Programme, die auf einem Windows-Betriebssystem ausgeführt werden können. Assemblys sind PEs die mit CLI-Informationen angereichert wurden und später auf einer VES-Implementierung ausgeführt werden. Dieses Format ist explizit im ECMA-Standard 335 und [Mic06b] beschrieben. Dabei gilt, jedes CLI-Image ist ein PE-Image aber nicht jedes PE-Image ist auch ein CLI-Image. Das resultiert daraus, dass auch kompilierte Programme und Bibliotheken, die nicht mit einem CLI-Compiler erzeugt werden, dieses Datei-Format haben. Allerdings fehlen dann die CLI-spezifischen Metadaten und können nicht auf einem *Virtual Execution System* ausgeführt werden.

Das PE Format liegt in Binärform vor. Die enthaltenen Informationen werden durch das Auslesen einzelner oder mehrerer Bytes aus dem Bytestrom der Datei gewonnen. Bei der Informationsgewinnung gibt es zum Beginn des Dateiformates eine Vielzahl von Informationsfeldern, die zu so genannten *Headern* zusammen gefasst sind. Beispielsweise sind dort Informationen über das Erstellungsdatum oder auch Versionsinformationen hinterlegt.

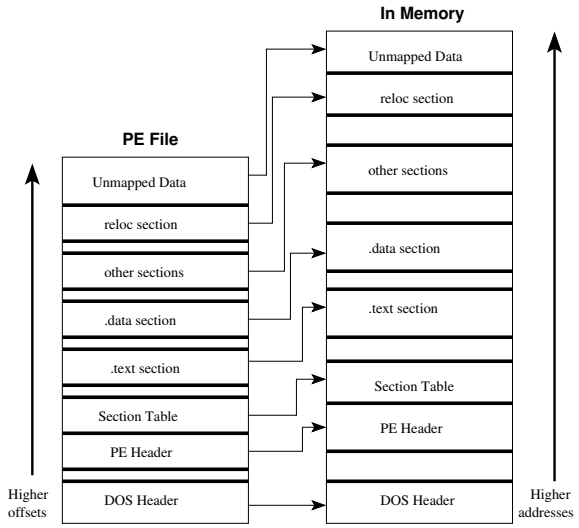

Abbildung 4.2: *Portable Executable Image* im Speicher und im Dateiformat ([Pie02])

Die Informationen des Dateiformates sind in verschiedene Abschnitte unterteilt. Sie werden zur Ausführungszeit in den Speicher der virtuellen Maschine oder den Hauptspeicher geladen. Hierbei werden die Abschnitte, die im Bytestrom vorzufinden sind, in den Speicher geladen. Die Adressierung von Abschnitten im Bytestrom erfolgt mit einem Abstandsmaß zu einem Startpunkt der Datei (*Offset*). Die Adressierung der in den Speicher geladenen Informationen erfolgt mittels relativer virtueller Adressen (RVA). Diese virtuellen Adressen sind relativ zum Startpunkt der Informationen im Speicher. Zwischen beiden Adressierungsarten lässt sich eine Beziehung herstellen, wie sie in der Abbildung 4.2 dargestellt wird. Auf die genaue Erstellung dieser Abbildung von RVA auf *Offset* werde ich am Ende des nächsten Abschnittes näher eingehen. Diese Abbildung ermöglicht eine Analyse des Binärformates ohne ein separates Speicherbild der Informationen vorhalten zu müssen, denn viele Informationen des Dateiformates sind mit virtuellen Adressen versehen.

Assemblys können als Softwarekomponente verstanden werden. Sie können aus:

- einem Modul (das eine eigenständige Software/Bibliothek enthält), oder
- mehreren zusammengefügten einzelnen Modulen

bestehen.

Assemblys, die aus verschiedenen einzelnen Modulen mit dem Programm *AL.exe*[1] zusammengelinkt werden, werden in der Analyse nicht betrachtet. Sie haben einen ähnlichen Aufbau zu Assemblys mit einem Modul, jedoch werden die Informationen der Metadaten auf ein Manifest und die zur Verfügung gestellten ExportedTypes beschränkt. In den folgenden Abschnitten bezeichnen Assembly und *Portable Executable Image* immer ein *Portable Executable Image* mit CLI.

[1] `http://msdn2.microsoft.com/en-us/library/c405shex(VS.80).aspx`

Zum Analysieren dieses Formates wird der ganze Bytestrom des Binärformates eingelesen und verarbeitet. Dazu wird im nächsten Abschnitt der Aufbau eines PE-Image mit CLI-Informationen detaillierter beschrieben.

4.2 Analyse eines Assembly (*Portable Executable Image* mit CLI-Erweiterung)

In diesem Abschnitt werde ich den Aufbau eines Assembly vorstellen. Wie schon in Abbildung 4.2 dargestellt wurde, besteht ein *Portable Executable Image* aus verschiedenen Bereichen.

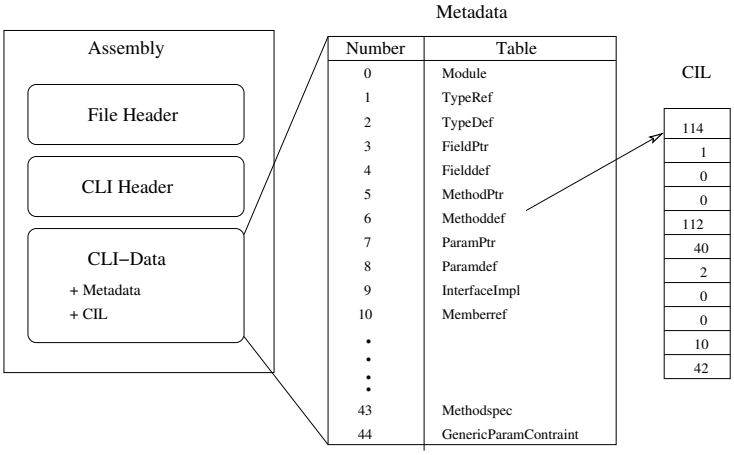

Abbildung 4.3: Aufbau eines *Portable Executable Files*

Die Abbildung 4.3 zeigt stark vereinfacht den Aufbau eines Assembly. Es besteht aus mehreren Abschnitten die jeweils spezielle Informationen bereitstellen. Der *File Header* enthält verschiedene Informationen über das Kompilat, aber auch über darauffolgende Informationsabschnitte. *CLI Header* und *CLI Data* enthalten die Informationen über die Metadaten und die Metadaten selbst. Die Abbildung 4.3 veranschaulicht das in den nächsten Abschnitten beschriebene Vorgehen zur Analyse. Zuerst werden die Informationen aus *File Header* und *CLI Header* gewonnen. Im Anschluss daran wird der *CLI Data* analysiert, der die Metadaten und Verweise zum CIL-Code enthält.

4.2.1 File Header

Ein *Portable Executable Image* beginnt immer mit einem speziellen Dateikopf (*File Header*). Dieser Dateikopf unterteilt sich in weitere kleine Bereiche (*Header*), dargestellt in Abbildung 4.4. Jeder dort enthaltene Bereich besteht aus einer Ansammlung von Werten, die jeweils bestimmte Eigenheiten des *Portable Executable Image* beschreiben. Die Bereiche und Werte werden anhand ihrer Feldgröße nacheinander aus dem Bytestrom der Datei ausgelesen, veranschaulicht in Abbildung 4.4 mit der Richtung der höheren Bytecode

Offsets. Die Felder können Zahlenwerte oder textuelle Informationen enthalten. Im Anhang A sind alle Felder inkl. Größe, Offset und Bezeichner aufgeführt, die zum *File Header* gehören (siehe auch [Mic06b, ECM06b]). Ich werde hier nur auf wichtige Felder eingehen, die Informationen für die Gewinnung weiterer Informationen im Bytecode enthalten.

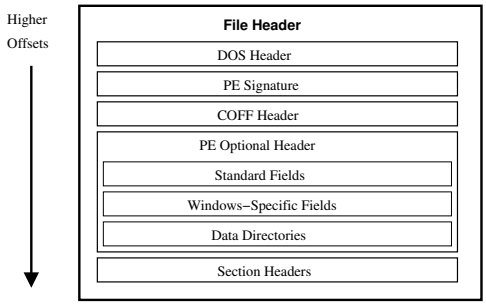

Abbildung 4.4: Der Aufbau des PE-Headers

Der *DOS Header* ist ein 128-Byte Block, der Werte enthält, an denen bestimmt werden kann, ob die gegebene Datei DOS-kompatibel ist und wo der *COFF Header* im Byte-Strom beginnt. Ab der im *DOS Header* definierten *COFF Header* Startadresse ist ein vier Byte Block zu finden, der den Wert PE00 enthält. Mit diesem Wert kann die Richtigkeit der Sprungadresse überprüft werden.

Im Anschluss folgt der *COFF File Header*. Er umfasst 20-Byte und enthält zwei wichtige Felder für die weitere Informationsgewinnung.

- NumberOfSections - Anzahl der definierten Abschnitte (Sections) im *Section Table*.
- SizeOfOptionalHeader - Größe des *PE Optional Header*.

Der *PE Optional Header* enthält drei Konstrukte: *Optional Header Standard Fields* mit 28-byte, *Optional Header Windows-Specific Fields* mit 68-Byte und einen 128-byte *Optional Header Data Directories*.

Die Felder der ersten beiden *PE Optional Header* enthalten keine relevanten Werte für das weitere Vorgehen, anders der *Optional Header Data Directory*. Er enthält eine Serie von acht Byte Einträgen. Bei diesen Einträgen repräsentieren die ersten vier Bytes eine RVA und die anderen vier Bytes eine Größe. Insgesamt gibt es von diesen Einträgen so viele wie im *COFF Header* im Feld *SizeOfOptionalHeader* ausgelesen wurden. Der wichtigste Eintrag ist das Feld *CLI Header*. Es enthält die Informationen, wo der *CLI Header* im Bytestrom gefunden werden kann.

Die *Section Header* folgen direkt auf den *PE Optional Header* und listen weitere Abschnitte (*Sections*) der Datei auf. Jeder *Section Header* ist 40 Byte groß und enthält folgende Felder:

- Name - Name des Abschnittes

- VirtualSize - Größe des Abschnittes im Speicher
- VirtualAddress - Startadresse des Abschnittes im Speicher
- SizeOfRawData - Größe der im Abschnitt enthaltenen Daten in Bytes
- PointerToRawData - Startoffset der Daten im Bytestrom
- Characteristics - Ein Bitvektor, der spezielle Eigenschaften wie Lesezugriff kodiert

Die Abschnittsnamen beginnen immer mit einem „.“. Normalerweise sind die Abschnitte *.text*, *.reloc* und *.rsrc* in einem *Portable Executable* vertreten.

Das Feld *VirtualAddress* gibt die Adresse des Abschnittes im Speicher an und das Feld *PointerToRawData* gibt den dazugehörigen *Offset* im Bytestrom der Datei an. Diese Abbildung zwischen Speicheradresse und Bytecode *Offset* wird in Abbildung 4.5 am Beispiel des *.text* Abschnittes gezeigt. Mit dieser Zuordnung ist es möglich virtuelle Adressen in *Offsets* des Bytestroms umzurechnen, sofern diese innerhalb eines Abschnittes liegen. Diese Umrechnung wird später häufiger benötigt, da im Weiteren fast ausschließlich virtuelle Adressen benutzt werden. Im Folgenden wird ein Verwenden dieser Umrechnung implizit vorrausgesetzt, da auf dem Bytestrom und nicht auf einem Speicherabbild gearbeitet wird.

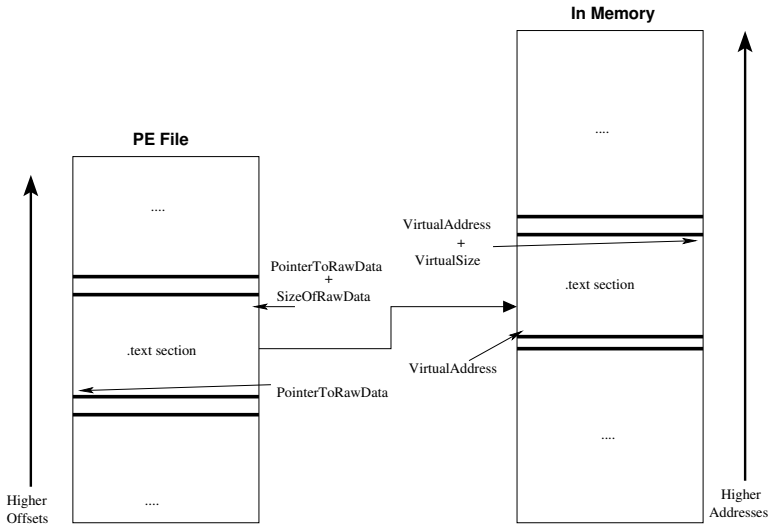

Abbildung 4.5: RVA zu *Offset* Abbildung von *Sections* (angelehnt an [Pie02])

4.2.2 CLI Header

Mit den Informationen *CLI Header Size* und RVA aus dem *Optional Header Data Directory* können unter Verwendung der Umrechnung von RVA in *Offsets* der *CLI Header* gefunden und ausgelesen werden.

Der *CLI Header* enthält laufzeitspezifische Informationen. Er besteht aus Dateneinträgen und weiteren Informationen. Normalerweise ist er in einem nur lesbaren Abschnitt, häufig *.text* untergebracht. Der *CLI Header* enthält folgende relevante Felder:

- MetaData - RVA und Größe der physikalischen Metadaten in diesem Image
- Flags - Bitvektor für metadatenspezifische Eigenschaften
- EntryPointToken - Enthält einen Verweis auf die Anfangsmethode, die in den Metadaten definiert ist

4.2.3 CLI Data

Die Analyse der *CLI Daten* sind essentieller Bestandteil dieser Arbeit. Sie sind, wie schon beschrieben, in einem bestimmten Bereich innerhalb eines Assemblys zu finden und bestehen aus einem Informationskopf am Anfang (*Metadatenheader*), *Metadatenstreams* und Metadatentabellen. Das *MetaData* Feld des *CLI Header* enthält die RVA der Metadaten im Speicher. Ab dieser Startadresse können die eingangs genannten Informationen ausgelesen werden. Die Gewinnung dieser Informationen wird im Folgenden näher beschrieben.

Metadatenheader und -streams

Nachdem der *CLI Header* im Bytestrom ausgelesen wurde, hat man den Anfangspunkt der Metadaten im Feld *MetaData* erhalten. Diese RVA wird auf ein *Offset* im Bytestrom umgerechnet. Nachdem der Einstiegspunkt zu den Metadaten berechnet wurde, kann die Richtigkeit der gefundenen Sprungmarke durch das Auslesen von vier Bytes ab der Sprungmarke validiert werden. Diese Werte der vier Bytes werden in vier Buchstaben konvertiert und müssen als Startpunkt des *Metadatenheader BSJB* enthalten.

Offset[2]	**Größe in Byte**	**Name**	**Beschreibung**
0	4	Signature	*BSJB*
4	2	MajorVersion	
6	2	MinorVersion	
8	4	Reserved	
12	4	Length	Länge der Versionsnummer in Bytes
16	Length	Version	Die Versionsnummer
16+Length	2	Flags	
18+Length+2	2	Streams	Anzahl der enthaltenen Byteströme

Die Informationen der Metadaten sind aufgeteilt in separate Byteströme (*Streams*). Diese einzelnen Byteströme sind kleine Teilströme aus dem Bytestrom des Dateiformates und beinhalten jeweils bestimmte Informationen.

- #~
  Dieser Stream enthält die Metadatentabellen, wenn es sich um optimierte Metadaten[3]

[2] Dieser Offset ist absolut zum Startpunkt der Metadaten

[3] Optimierte Metadaten sind hinsichtlich ihrer Verweise untereinander geordnet. Beispielsweise liegt eine geschachtelte Klasse nach der Klasse in der sie deklariert wurde in den Metadaten.

handelt.

- #String
  Dieser Stream enthält Namen und Bezeichner die in den Metadatentabellen verwendet werden. Sie werden aneinandergereiht und durch ein Leerzeichen getrennt.

- #US
  Der Userdefined String (US) Stream enthält Strings, die innerhalb des Programms definiert werden. Beispielsweise innerhalb von Methoden wenn ein String mit einer Textzuweisung definiert wird. Dieser Streams wird alleinig von CIL-Instruktionen benutzt und ist für die weitere Analyse nicht relevant.

- #Blob
  Dieser Stream enthält Signaturen oder andere nicht Stringojekte.

- #GUID
  Dieser Stream enthält GUID (Global Unique IDentifier) Informationen. GUID bestehen aus 16 Bytes. Diese 128-Bit-Nummer wird unter anderem zur Identifizierung von Softwarekomponenten genutzt.

- #-
  Dieser Stream existiert nur, wenn die Metadaten nicht optimiert sind und somit der ∼ Stream nicht vorhanden ist.

Jeder Stream wird durch die in Tabelle 4.3 aufgelisteten Felder näher spezifiziert.

Offset	**Größe in Byte**	**Name**	**Beschreibung**
0	4	Offset	Zum Metadatenstartpunkt
4	4	Size	Größe des Bytestromes
8	x	Name	Names des Bytestromes

Tabelle 4.3: Header eines Streams

Diese Felder werden im Anschluss an den *Metadatenheader* im Bytestrom ausgelesen. Diese Felder zeigen an welcher Stelle und wie groß der jeweilige *Stream* ist.

Schwierig beim Auslesen der *Streams* ist, dass die zu dem Stream gehörenden Felder vor dem Namen des Streams im Bytecode zu finden sind. Es gibt auch keine eindeutige Reihenfolge, in der die Streams im Bytecode zu liegen haben. Dadurch wird das Finden und zuordnen der Felder zu den Streams erschwert.

Gängige Compiler erzeugen meistens optimierte Metadaten. Aus diesem Grund werde ich mich in der Analyse auf optimierte Metadaten beschränken. Das bedeutet, dass die zu analysierenden Metadaten im #∼-Stream zu finden sind. Zum Beginn des #∼-Streams befindet sich ein *Header*, der Informationen über bestimmte Eigenschaften der Metadaten

enthält, dargestellt in Tabelle 4.5. Dieser *Header* enthält einen Schlüssel zur Dekodierung der Metadatentabellen und im Anschluss die Tabellen selbst.

Offset[4]	**Größe in Byte**	**Name**
0	4	Reserved
4	1	MajorVersion
5	1	MinorVersion
6	1	HeapSize
7	1	Reserved
8	8	Valid
16	8	Sorted
24	4*x	Rows
24+4*x		Tabellen

Tabelle 4.5: Metadatentabellen

Das *Valid*-Feld ist ein Bitvektor. Er kodiert welche Metadatentabellen der 64 möglichen in diesem Assembly tatsächlich benutzt werden. Die Tabellen sind von 0 bis 63 durchnummeriert. Jede 1 im Bitvektor gibt an, dass die Tabelle die durch die Stelle des Bits repräsentiert wird vorhanden ist. Diese Dekodierung ist dargestellt in Tabelle 4.7. In diesem Beispiel sind unter anderem die Tabellen mit den Nummern 0,1,2,4,... im Assembly enthalten.

Tabelle	63	62	61	...	4	3	2	1	0
Bitvektor	0	0	0	...	1	0	1	1	1

Tabelle 4.7: Bitvektor zur Dekodierung der Metadatentabellen

Jede mit dem Bitvektor markierte Tabelle enthält Spalten. Die Anzahl der Spalten jeder vorkommenden Tabelle ist im Feld *Rows* zu enthalten. Die Spaltenanzahl jeder vorhandenen Tabelle wird im Anschluss an den in Tabelle 4.5 gezeigten *Header* ausgelesen. Dazu wird der Bitvektor von der Bitnummer 0 bis zur 64 durchlaufen und immer wenn das betrachtete Bit den Wert *1* enthält werden vier Bytes aus dem #∼-Stream ausgelesen. Diese vier Bytes enthalten die Anzahl der Spalten für die Tabelle mit der Nummer des betrachteten Bits.

Das Metadatenformat mit seinen Tabellen ist an dieser Stelle ausbaufähig gehalten worden. Es existieren insgesamt 64 Tabellen von denen aber im derzeitigen CLI-Standard nur 45 Tabellen genutzt werden. Im Anhang B werden die genutzten Tabellen mit Name und Nummer aufgelistet. Diese Erweiterungsmöglichkeit ist vorteilhaft, da bei kleineren Erweiterungen des Formates oder fehlenden Abbildungsmöglichkeiten von Programmaspekten nicht gleich ein neues Format entworfen werden muss, sondern einfach neue Tabellen eingeführt werden können.

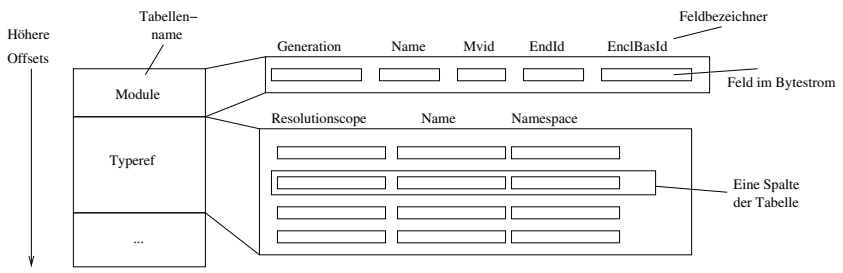

Abbildung 4.6: Metadatenstruktur

Jede Tabelle enthält Spalten, die jeweils tabellenspezifische Felder enthalten, dargestellt in Abbildung 4.6. Die Tabellen und ihre Felder werde ich in Abschnitt 4.3 beschreiben.

Felder können Zeiger in andere Tabellen, feste Werte, oder Zeiger in die Streams #String, #Blob und #GUID enthalten. Nur wenige dieser Felder haben eine fest definierte Größe. Die meisten Felder haben dynamische Größen die abhängig von Tabellen- oder *Stream*-Größen sind.

Das *HeapSize*-Feld kodiert die Größe der #String-, #Blob- und #GUID-Zeigergröße. Diese Werte geben Aufschluss darüber, ob der Zeiger auf die Streams zwei Bytes oder vier Bytes groß ist. Zusätzlich werden die Zeigergrößen zur Referenzierung anderer Tabellen ermittelt. Die Standardgröße von Feldern die Tabellen referenzieren ist zwei Bytes. Sollte nun eine referenzierte Tabelle mehr Einträge haben als mit zwei Bytes referenziert werden können, vergrößert sich das Feld in der Tabelle auf vier Bytes. Somit können alle Einträge in der referenzierten Tabelle erreicht werden. Für diese Berechnung der Feldgröße werden die im Assembly vorhandenen Tabellen sowie die Anzahl ihrer Einträge benötigt. Nachdem die Größe der verschiedenen Felder in den Tabellenspalten berechnet wurden, werden mit der Tabelle 0 beginnend für jede Spalte der Tabelle die Felder aus dem Bytestrom entnommen.

Mit dem Auslesen der Tabellen hat man alle wichtigen Informationen über die Zwischendarstellung gewonnen. Somit ist die technische Analyse des PE Formates abgeschlossen, so dass im nächste Abschnitt auf die einzelnen Tabellen eingegangen sowie die darin enthaltenen Informationen beschrieben. Diese Informationen bilden die Grundlage für die Generierung des RFG.

4.3 Analyse der Metadaten Tabellen

Die Metadaten sind das Herzstück der CLI. Sie repräsentieren alle wichtigen Informationen, die zum Ausführen auf einem *Virtual Execution System* benötigt werden. Hierzu zählen unter anderem verwendete Typen aus anderen Assemblys oder auch Klassen mit ihren Methoden und Variablen.

4.3.1 Dekodierung von Informationen innerhalb der Metadaten Tabellen

Zur Dekodierung von einigen Feldern in Tabellenspalten sind spezielle Techniken notwendig. Diese werden vorab vorgestellt, da diese Konzepte im Abschnitt 4.3.2 bei Analyse benötigt werden.

Decodierung von Zeigern auf andere Tabellen

Einige Zeiger in andere Metadatentabellen können nicht explizit benutzen werden, da sie in mehrere Tabellen zeigen können. In einem solchen Fall besteht das Feld aus einem Bitvektor, der mit einer speziellen Tabelle dekodiert wird. Beispielsweise kann ein Feld Zeiger auf die Tabellen *TypeRef*, *TypeDef* oder *TypeSpec* enthalten. Anhand der Felder wird aus [ECM06b] S. 170 ff. die entsprechende Dekodiertabelle, in diesem Fall *TypeDefOrRef*, ausgewählt. In diesem Fall können drei Tabellen im Feld kodiert sein, deswegen wird der Bitvektor bitweise mit dem Wert drei verundet. Das Ergebnis (0, 1 oder 2) wird aus der *TypeDefOrRef*-Auflistung entnommen und spezifiziert die referenzierte Metadatentabelle. Diese Information reicht aber noch nicht als konkrete Referenz aus, so dass die zwei dekodierten Bits um zwei Bits nach recht aus dem Bitvektor geschoben. Die so entstandene Zahl spezifiziert die Spalte der vorher ermittelten Tabelle.

Im Folgenden werde ich nicht auf jedes zu dekodierende Feld eingehen. Die Schema zur Referenzgewinnung ist für alle Felder die Zeiger in andere Tabellen kodieren ähnlich. Sie unterscheiden sich lediglich in der Anzahl der kodierten Tabellen. Die Dekodiertabellen sind abhängig von der Tabelle und dem enthalten Feld. Sie sind in [ECM06b] S. 170 ff. beschrieben und werden im Weiteren nicht explizit beschrieben. Felder, die Bitvektoren enthalten, werden mit ihrem Feldnamen benannt und erhalten in der Beschreibung die möglichen kodierten Referenzen auf andere Tabellen.

Zeiger in einen #String-Stream

Wie schon im vorherigen Kapitel kurz erwähnt, enthält der #String-Stream Bezeichner wie Klassennamen, die in den Metadaten verwendet werden. Diese Bezeichner werden mit einem Zeiger in den Spalten von Tabellen referenziert. Anhand dieses Zeigers wird Byte für Byte aus dem #String-Stream ausgelesen und in Zeichen umgewandelt. Das Ende eines Bezeichners ist mit einem Nullbyte gekennzeichnet, veranschaulicht in Abbildung 4.7.

```
0000:0490 6a 65 63 74 00 53 79 73 74 65 6d 00 2e 63 74 6f ject.System..cto
0000:04a0 72 00 53 74 72 69 6e 67 00 43 6f 6e 63 61 74 00 r.String.Concat.
0000:04b0 43 6f 6e 73 6f 6c 65 00 57 72 69 74 65 4c 69 6e Console.WriteLin
0000:04c0 65 00 48 65 6c 6c 6f 57 6f 72 6c 64 00 3c 4d 6f e.HelloWorld.<Mo
0000:04d0 64 75 6c 65 3e 00 48 65 6c 6c 6f 57 6f 72 6c 64 dule>.HelloWorld
0000:04e0 2e 65 78 65 00 4d 79 43 6c 61 73 73 00 73 00 2e .exe.MyClass.s..
0000:04f0 63 63 74 6f 72 00 4d 61 69 6e 00 52 75 6e 74 69 cctor.Main.Runti
0000:0500 6d 65 43 6f 6d 70 61 74 69 62 69 6c 69 74 79 41 meCompatibilityA
0000:0510 74 74 72 69 62 75 74 65 00 53 79 73 74 65 6d 2e ttribute.System.
0000:0520 52 75 6e 74 69 6d 65 2e 43 6f 6d 70 69 6c 65 72 Runtime.Compiler
```

Abbildung 4.7: Teile eines #String-Streams angezeigt in einem Hexadezimal-Editor

Zeiger in einen #GUID-Stream

Eine GUID hat immer eine Länge von 16 Byte sowie eine spezielle Darstellungsform der Bytes[5]. Ein Zeiger in diesen *Stream* impliziert somit das Auslesen von 16 Bytes aus denen sich eine #GUID zusammensetzt.

Zeiger in einen #Blob-Stream

Der #Blob-Stream enthält unter anderem die Signaturen für die in der Tabelle *MethodDef* enthaltenen Methoden. Diese Signaturen sind von Tabelle zu Tabelle unterschiedlich und können im Detail der [ECM06b] Partion II S.152 ff. entnommen werden.

Der #Blob-Stream ist wie alle anderen Streams ein Bytestrom. In diesem können die enthaltenen Informationen aber zusätzlich in einer gepackten oder nicht gepackten Form enthalten sein. Gepackt heißt, das mehrere Bytes eine Information enthalten. Um dies zu bestimmten, wird das aktuelle Byte angeschaut und auf seinen Wert geprüft.

- Ist der Wert zwischen 0 und 127 ist die Information nicht gepackt und kann direkt verwendet werden.
- Ist der Wert zwischen 128 und 192 müssen zwei Bytes für die Information zusammengefügt werden.
- Ist der Wert größer als 192 müssen vier Bytes zusammengefügt werden.

Diese Prüfung muss bei jedem Auslesen von Informationen aus dem #Blob-Stream berücksichtigt werden.

Ich werde das Auslesen einer Signatur am Beispiel der *Field*-Signatur zeigen. Die *Blob*-Signatur für die *Field*-Tabelle wird in Abbildung 4.8 dargestellt.

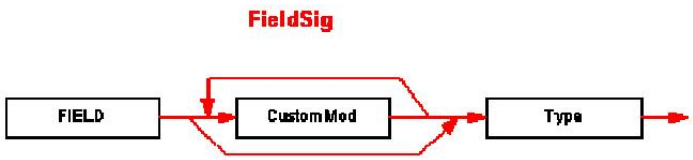

Abbildung 4.8: *Blob*-Signatur für die *Field*-Tabelle ([ECM06b] Partition II S.157)

In dieser Darstellung steht jeder Kasten für das Auslesen einer *Blob*-Information und die Pfeile für die Leserichtung. Der *Field*-Kasten zeigt an, dass die *Blob*-Signatur mit einer 6 beginnen muss. Danach kann ein weiteres separat definiertes Konstrukt der *CustomMod* folgen und zum Abschluss folgt der Typ des Feldes. Der Typ ist dabei wieder ein separates Konstrukt zum Auslesen von *Blob*-Informationen.

[5] Beschreibung der Darstellungsform unter `http://msdn2.microsoft.com/de-de/library/ms131157(VS.80).aspx`

Nachdem die Struktur der *Blob*-Signatur für das Feld analysiert wurde, kann mit der Informationsgewinnung begonnen werden. Der Zeiger in einen #Blob-Stream markiert den Anfangspunkt zum Auslesen der Signatur. Das dort vorgefundene Feld enthält die Anzahl der Bytes, in denen die Signatur kodiert ist. Danach wird über die Anzahl der Bytes iteriert und die in Abbildung 4.8 beschriebene Signatur dekodiert. Die Abbildung 4.9 stellt die Dekodierung mit den Werten des Bytecodes dar. Hierbei muss der letzte Wert, der den Typ repräsentiert, noch mit der Typsignatur dekodiert werden. Sie hat wiederum eine eigene *Blob*-Signatur zur Informationsgewinnung. In diesem Beispiel steht der Wert *8* für einen 32-Bit-Integer und es war keine *CustomMod*-Konstrukt in der Signatur enthalten.

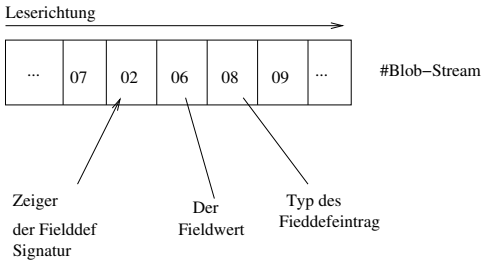

Abbildung 4.9: Darstellung einer *Field*-Signatur im #Blob-Stream

Dekodierung von Attributen

Einig Metadatentabellen enthalten auch Attributfelder (*Flag*-Felder). Diese Felder haben eine feste Bytegröße und enthalten einen Bitvektor. Für jedes dieser *Flag*-Felder gibt es eine Dekodiertabelle mit der die dort enthaltenen Attribut dekodiert werden können. Anders als bei der schon beschriebenen Zeigerdekodierung wird der Bitvektor mit den vorgegebenen aus der Dekodiertabelle hervorgehenden Werten dekodiert. Diese Werte sind in der zu finden und abhängig vom *Flag*-Feld und der Tabelle zudem das Feld gehört, beispielsweise das *Flag*-Feld für Einträge der *TypeDef*-Tabelle. Einträge in dieser Tabelle enthalten unter anderem Klassendefinitionen. Nehmen wir an das *Flag*-Feld enthält den Bitvektor *10000000000000000001* und es soll die Sichtbarkeit der Klasse ermittelt werden. Ein Ausschnitt der dazugehörigen Dekodiertabelle ist in Abbildung 4.10 dargestellt. In der Tabelle wird angegeben, das die Sichtbarkeit (*VisibilityMask*) durch die Verundung des Bitvektors mit dem Wert 7 dekodiert werden kann. Das Ergebnis der Dekodierung ist beim Beipielvektor *1*, so dass sich anhand der Tabelle ablesen lässt, das das Attribut *Public* im Bitvektor enthalten ist. *VisibilityMask*, die die Sichtbarkeit kodiert.

23.1.15 Flags for types [TypeAttributes]

Flag	Value	Description
Visibility attributes		
VisibilityMask	0x00000007	Use this mask to retrieve visibility information. These 3 bits contain one of the following values:
NotPublic	0x00000000	Class has no public scope
Public	0x00000001	Class has public scope
NestedPublic	0x00000002	Class is nested with public visibility
NestedPrivate	0x00000003	Class is nested with private visibility
NestedFamily	0x00000004	Class is nested with family visibility
NestedAssembly	0x00000005	Class is nested with assembly visibility
NestedFamANDAssem	0x00000006	Class is nested with family and assembly visibility
NestedFamORAssem	0x00000007	Class is nested with family or assembly visibility

Abbildung 4.10: Ausschnitt aus der Dekodiertabelle für das *Flag*-Feld der *TypeDef*-Tabelle (Tabelle 23.1.15 aus [ECM06b] Partition II S. 143 ff.)

Die Attributnamen sind teilweise anders als die Attribute während der Programmierung (vgl.fig:net:visibi). Sofern starke Unterschiede zwischen den Attributbezeichnern der CLI und der Attribute beim Programmieren vorhanden sind, werden ich entsprechend auf die Namensgebung, die im RFG verwendet wird, hinweisen.

4.3.2 Beschreibung der Metadaten Tabellen

Insgesamt gibt es zu Informationsrepräsentation 64 Tabellen von denen aber nur effektiv 45 genutzt werden. Eine Übersicht der genutzten Tabellen befindet sich in Anhang B. Die Metadatentabellen und ihre Felder werden in [ECM06b] Partition II beschrieben und ich werde genauer auf die, für die Darstellung im RFG relevanten, Tabellen eingehen. Jeder Unterabschnitt enthält jeweils die Felder der behandelten Tabelle, sofern diese wichtige Analyseinformationen enthält.

Module

Die erste Tabelle der Metadaten ist die *Modul*-Tabelle. Sie enthält eine GUID, die die Instanz dieses Moduls definiert. Verschieden kompilierte Assemblys lassen sich anhand ihres aus dem #GUID-Stream ausgelesenen GUID-Wertes unterscheiden.

Feldname	Beschreibung
Generation	Enthält einen Wert der meist 0 ist
Name	Zeiger in den #String-Stream
Mvid	Zeiger in den #GUID-Stream
EndId	Enthält einen Wert der meist 0 ist
EnclBaseId	Enthält einen Wert der meist 0 ist

TypeRef

Die *TypeRef*-Tabelle enhält alle verwendeten Klassenreferenzen. Die Referenzen können auf einen anderen Eintrag in der selben Tabelle, externe Module oder Assemblys verweisen. In den *TypRefs* werden somit alle extern verwendeten Typen abgebildet, gleich ob sie aus der *Base Class Library* oder anderen Bibliotheken kommen und von Einträgen der *Typdef*-Tabelle referenziert werden.

Feldname	**Beschreibung**
Resolutionscope	Zeiger in die *TypeRef*-, *AssemblyRef*-, *ModuleRef*- oder *Module*-Tabelle
TypeName	Zeiger in den #String-Stream
TypeNamespace	Zeiger in den #String-Stream

TypeDef

TypeDefs sind Typdefinitionen. Sie bilden im Quelltext definierte Schnittstellen, Strukturen, Delegaten oder Klassen ab. Jede Spalte in dieser Tabelle enthält den Namen des Typs und den Namensraum in dem er deklariert wurde. Die in Abschnitt 2.3.1 angesprochenen partiellen Klassen sind hier nicht mehr zu identifizieren. Sie wurden vom Compiler zu einer zusammengefügt und werden in dieser Tabelle als eine Klasse abgebildet.

Feldname	**Beschreibung**
Flags	Bitvektor der Attribute kodiert
TypeName	Zeiger in den #String-Stream
TypeNamespace	Zeiger in den #String-Stream
Extends	Zeiger in die *TypeDef*-, *TypeRef*- oder *TypeSpec*-Tabelle
FieldList	Zeiger in die *Field*-Tabelle
MethodList	Zeiger in die *MethodDef*-Tabelle

Das Flagfeld enthält einen Bitvektor der Attribute der Klasse kodiert. Dieser Bitvektor dekodiert folgende analyserelevante Eigenschaften:

- Sichtbarkeitsattribute
  - „Normale“ Klassen mit *Public* und *NotPublic*
  - Geschachtelte Klassen mit *NestedPublic, NestedPrivate, NestedFamily, NestedAssembly, NestedFamAndAssem* und *NestedFamOrAssem*
- Spezielle Semantik
  Attribute *Abstract* und *Sealed*
- Klassensemantik
  Unterscheidung zwischen Klasse und Schnittstelle

Anhand dieser Flags können schon einige Informationen zu der vorliegenden Klasse gewonnen werden. Auffällig ist das der Bitvektor des *Flag*-Feldes kein *Static*-Attribut

Attributbezeichnung der CLI	Attributbezeichnung im RFG
NestedPublic	Public
NestedPrivate	Private
NestedFamily	Protected
NestedAssembly	Internal
NestedFamAndAssem	Protected Internal
NestedFamORAssem	Protected Internal

Tabelle 4.10: Abbildung der CLI-Sichtbarkeiten auf Sichtbarkeiten im RFG

enthält, dies aber bei der Programmierung durchaus verwendet werden kann. *Static* für Klassen wurde nachträglich in den Sprachschatz von C# aufgenommen, deswegen wird es mit der Kombination der Attribute *Abstract* und *Sealed* zusammen dargestellt. Dies ist kein Widerspruch zur Programmierung, denn *Abstract* und *Sealed* dürfen niemals zusammen eine Klasse attributieren.

Anhand der Sichtbarkeitsattribute kann eine Klasse leicht als geschachtelte Klasse identifiziert werden. Die Bezeichnungen der Sichtbarkeitsattribute werden dazu in die schon bekannten Sichtbarkeiten umbenannt (siehe Tabelle 4.10) und bei der Abbildung von geschachtelten Klassen im RFG verwendet.

Alle Typen, die in einem Programm definiert werden, werden in der *TypeDef*-Tabelle aufgeführt. *TypeDef*-Tabelleneinträge können folgende Typen abbilden:

- Klasse
- *Struct*
- Schnittstelle
- Aufzählungstyp (*Enum*)
- Generische Klasse
- Generisches *Struct*
- Generische Schnittstelle
- Geschachtelter Typ

Anhand des *Flag*-Feldes können lediglich Klassen von Schnittstellen unterschieden werden. Für weitere Unterscheidungen müssen andere Metadatentabellen hinzugezogen werden. Klassen, *Structs* und *Enums* können auf Basis ihres Abstammungstypen unterschieden werden. Bei *Structs* ist dies *System.ValueType* und bei *Enums System.Enum*. Diese Information ist im *Extends*-Feld enthalten. Es enthält von welcher Basisklasse das Objekt abstammt.

Zur Abgrenzung generischer Typen von normalen Typen muss ein Blick in die Einträge der *GenericParam*-Tabelle geworfen werden. Diese Tabelle enthält alle im Quelltext definierten

generischen Parameter mit einem Verweis zu welchem Objekt der generische Parameter gehört. Durch einen Abgleich der dort referenzierten Typen kann der Eintrag der *TypeDef*-Tabelle als generisch oder nicht generisch identifiziert werden. Eine weitere Unterscheidung in Schnittstelle, *Struct* und Klasse erfolgt bei generischen Typen analog zu normalen Typen.

Die zu einer Typ gehörenden Variablen werden mit einem Zeiger in die *Field*-Tabelle angezeigt. Die *Field*-Tabelle enthält alle Variablen die im Quelltext als Klassenvariablen definiert wurden. Das Feld *FieldList* ist markiert lediglich den Startpunkt auf eine Spalte in der *Field*-Tabelle. Dies reicht nicht aus um alle Variablen, die zu einem Typ gehören, zuzuordnen. Das wird möglich wenn die nächste Spalte in der *TypeDef*-Tabelle mit ihrem *FieldList*-Feld betrachtet wird. Die Differenz der beiden Felder legt fest, welche Variablen zum gerade betrachteten Eintrag der Tabelle gehören, veranschaulicht in Abbildung 4.11.

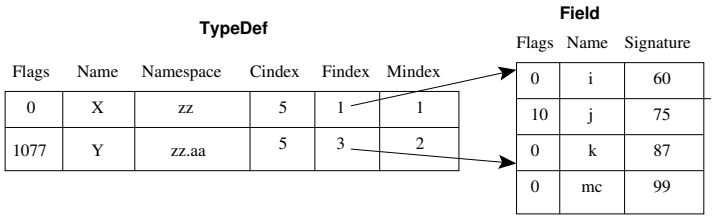

Abbildung 4.11: Zuordnung der Klassenvariablen

Analog zur Gewinnung der Klassenvariablen wird bei der Zuordnung der Methoden vorgegangen. Der einzige Unterschied ist, dass die Spalten durch das Feld *MethodList* angezeigt werden.

Die *TypeDef*-Tabelle enthält immer zuerst den Platzhaltereintrag <*Module*>. Dieser Eintrag wird vom Compiler erzeugt und ist ein künstliches Konstrukt. Das in Kapitel 3 entwickelte C#-Schema bildet vornehmlich Konstrukte ab, die im Quelltext definiert wurden. Trotzdem wird dieser Eintrag der Vollständigkeit halber in der RFG-Erzeugung verwendet wird.

Field

Die Einträge in dieser Tabelle repräsentieren Variablen und Ereignisse, die in Klassen definiert wurden. Mit dem Bitvektor des *Flag*-Feldes können unter anderem die Sichtbarkeit und die Eigenschaft *Static* des Eintrags dekodiert werden.

In Abschnitt 3.2.4 wurde beschrieben, das Variablen mit dem Attribut *volatile* ausgezeichnet sein können, so dass die Datenkonsistenz bei Threadbenutzung gewährleistet wird. Diese Attribut kann nicht mit dem *Flag*-Bitvektor dekodiert werden es kann nur durch die Analyse der CIL-Anweisungen gewonnen werden. Die Gewinnung diese Attributs wird später in Abschnitt 4.3.3 beschrieben.

Der Typ der Variable wird mit dem Auslesen der *Field*-Signatur aus dem #Blob-Stream gewonnen.

Feldname	Beschreibung
Flags	Bitvektor der Attribute kodiert
Name	Zeiger in den #String-Stream
Signature	Zeiger in den #Blob-Stream

MethodDef

Die *MethodDef*-Tabelle enthält alle im Quelltext definierten Methoden.

Feldname	Beschreibung
RVA	Adresse, wo der Methodenrumpf zu finden ist
ImplFlags	Bitvektor mit speziellen Implementierungsinformationen
Flags	Bitwert mit Attributen der Methode
Name	Zeiger in den #String-Stream
Signature	Zeiger in den #Blob-Stream

Der *ImplFlags*-Bitvektor enthält Informationen über die Art des in der Methode enthaltenen Codes wie *Optimized IL*[6] oder *IL*. Außerdem kann die Eigenschaft *Synchronized* mit diesem Bitvektor dekodiert werden.

Die Methodenflags geben Aufschluss über folgende Eigenschaften der vorliegenden Methode:

- Sichtbarkeit (*Public*, ...)
- Spezielle Eigenschaften wie *Static*, *Sealed*, *Virtual*, *Abstract*
- *Reuse slot* und *New slot*

In Abschnitt 2.3.5 wurde beschrieben wie virtuelle Methoden einmal neudefiniert und überschieben werden können. Diese unterschiedliche Behandlung des Methodenüberschreibens wird mit den Eigenschaften *New Slot* oder *Reuse Slot* ausgedrückt. Bei der Redefinition einer virtuellen Methode wird ein neuer Speicherbereich für die Methode in der Instanz erzeugt. Im anderen Fall wird der schon reservierte Speicherbereich der Methode der Basisklasse genutzt und sie überschreibt die Definition der geerbten Methode.

Die Methodensignatur wird ab der im Feld *Signature* angegeben Stelle aus dem #Blob-Stream extrahiert. Die *Blob*-Signatur enthält Informationen über den Rückgabetyp sowie alle Parameter der Methode.

MethodDef-Einträge deren RVA-Feld nicht *null* ist enthalten CIL-Code. Dieser ist nicht in den Einträgen oder Feldern der *MethodDef*-Tabelle enthalten, sondern liegt gesondert im Bytestrom vor. Zum Auslesen wird im Bytestrom zum berechneten *Offset* des RVA-Feldes

[6] IL - Intermediate Language, eine andere Bezeichung für den CIL-Code

gesprungen. Der Methodeninhalt kann in zwei Formaten repräsentiert werden. Aufschluss über das Format des Inhaltes geben die ersten zwei Bits. Ergeben die Bits den Wert *2* liegen die Informationen im *Tiny*-Format vor und bei einem Wert von *3* im *Fat*-Format.

Methoden im *Tiny*-Format enthalten keine Ausnahmebehandlungen (Exceptions) und der Methodenstack ist kleiner als acht Einträge. Der Methoden-*Header* umfasst insgesamt sechs Bits. Hierbei stehen zwei Bits zur Dekodierung des Headertyps und die darauffolgenden vier Bits geben die Länge des enthaltenen Bytecodes in Byte an. Methoden im *Fat*-Format können lokale Variablen und Ausnahmebehandlungen enthalten sowie einen größeren Stack benötigen. Dazu enthält der *Header* neben der Länge des Bytecodes die maximale Stackgröße der Methode und ein Token, das die lokalen Variablen kodiert. Der Methoden-*Header* umfasst 12 Bytes.

Nachdem das Methodenformat bestimmt wurde kann anhand des Formates der jeweilige Methoden-Header ausgelesen werden. Im Anschluss an den jeweiligen Methoden-*Header* folgt der CIL-Code. Die Länge des Bytestroms der den CIL-Code enthält, ist von den vorher ermittelten Feldern abhängig. Die Analyse des CIL-Codes werde ich später in Abschnitt 4.3.3 beschreiben.

Neben explizit deklarierten Methoden gibt es auch compilergenerierte Methoden. Hierzu zählen die Methoden für das Anmelden und Abmelden von Ereignissen. Die Namen der Methoden werden aus dem jeweiligen Vorgang den die Methode abbildet (add/remove) und dem Eventnamen erzeugt. Beispielsweise für das Abmelden des Events *e* heißt die Methode *remove_e*.

Methoden können auch Konstruktor- und Dekonstruktormethoden sein. Konstruktormethoden können innerhalb der Metadaten in zwei verschiedene Konstruktorarten unterschieden werden. Es gibt zum einen Instanzkonstruktoren die immer den Namen *.ctor* tragen und für die Erzeugung einer Instanz verantwortlich sind. Zum anderen gibt es den Initialisierungskonstruktor *.cctor*, der Instanzvariablen, nach der Instanziierung eines Objektes initialisiert. Dekonstruktoren tragen immer den Name *Finalize*.

Param

In der *Param*-Tabelle sind die lokalen Variablen von Methoden enthalten. Die Tabelle ist ähnlich zur *Field*-Tabelle aufgebaut und enthält einen Bitvektor zur Attributdekodierung, den Parameternamen und seinen Typ. Diese lokalen Elemente werden im RFG nicht abgebildet, so dass diese Tabelle für die Analyse weitgehend uninteressant ist.

InterfaceImpl

Diese Tabelle zeigt an, welche Klasse eine Schnittstelle implementiert. Sie enthält die Felder *Class* und *Interface*. Das Feld *Interface* zeigt auf die Schnittstelle, die von der in *Class*

angegeben Klasse implementiert wird.

Feldname	**Beschreibung**
Class	Zeiger in die *TypeDef*-Tabelle
Interface	Zeiger in die *TypeDef*-, *TypeSpec*- oder *TypeRef*-Tabelle

MemberRef

Die Spalten dieser Tabelle beschreiben Methoden oder Felder, die aus anderen Tabelleneinträgen referenziert werden. Beispielsweise wird eine Klasse erzeugt, die von der Klasse *System.Objekt* abstammt. Beim Aufruf des Konstruktors wird der Konstruktor der Basisklasse durch einen Eintrag dieser Tabelle aufgerufen.

Feldname	**Beschreibung**
Class	Zeiger in die *TypeSpec*-, *MethodDef*-, *ModuleRef*-, *TypeRef*- oder *TypeDef*-Tabelle
Name	Zeiger in den #String-Stream
Signature	Zeiger in den #Blob-Stream

Constant

Die *Constant*-Tabelle enthält Verweise auf Einträge in der *Field*-, *Param*- oder *Property*-Tabelle. Ein Eintrag in dieser Tabelle signalisiert, dass es sich bei der im *Parent*-Feld kodierten Tabelleneintrag um eine Konstante handelt.

Mit dem Zeiger in den #Blob-Stream und dem Wert des Feldes *Dtype* kann der Wert ermittelt werden, den die Konstante hat. Bei einem *Dtype*-Wert unter 12 besteht die Konstante aus einer Zahl. Beträgt der Wert 14 repräsentiert die Konstante einen String.

Feldname	**Beschreibung**
Dtype	Hilfswert zur Ermittlung der Signatur
Zero	Enthält immer eine 0
Parent	Zeiger in die *Field*-, *ParamDef*- oder *Property*-Tabelle
Signature	Zeiger in den #-Blob-Stream

CustomAttribute

Die *CustomAttribute*-Tabelle enthält Einträge sobald Klassen oder Methoden mit speziellen Attributen versehen werden. Diese Attribute sind laufzeitspezifisch und nicht vergleichbar mit Attributen wie *Public* oder *Volatile*. Diese Attribute werden nicht im RFG abgebildet, so dass diese Tabelle nicht für die RFG-Generierung verwendet wird.

FieldMarshal

In Abschnitt 2.3.10 wurde darauf hingewiesen, das es spezielle Feldbehandlungen zur Absicherung der Informationen bei der Verwendung von *Unmanaged-Code* definiert werden können. Diese Informationen sind in der *FieldMarshal*-Tabelle enthalten. Sie verknüpft Einträge aus der *Field*- oder *Property*-Tabelle mit einer Signatur im #Blob-Stream, welcher definiert wie dieses Feld bei einem Aufruf verwendet werden soll. Diese Tabelle ist nicht relevant für die Analyse im Sinne des RFG.

DeclSecurity

Der Zugriff auf Ressourcen kann mit Sicherheitshinweisen versehen werden. Damit können beispielsweise Regeln definiert werden, die eine Codeausführung legitimieren. Diese Sicherheitshinweise werden in der *DeclSecurity*-Tabelle definiert. Solche Informationen werden nicht im RFG dargestellt, so dass diese Tabelle keiner weiteren Analyse bedarf.

ClassLayout, FieldLayout

Die *ClassLayout*-Tabelle enthält Informationen über die spezielle Anordnung von Feldern innerhalb einer Klasse. Manchmal werden Felder bei der Metadatenerzeugung zur Optimierung neugeordnet und dieses Feldlayout bei Vererbung an eine andere Klasse weitergegeben. Ein explizites Abweichen davon wird in der *ClassLayout*-Tabelle festgehalten.

Innerhalb von C#-Programmen können Variablen explizit zu einem bestimmten Zeitpunkt deklariert werden, um den Initialisisierungszeitpunkt von Variablen im Programmablauf auszunutzen. Dieses explizite Variablenlayout wird in der *FieldLayout*-Tabelle festgehalten.

Diese Tabelle, wie auch die *ClassLayout*-Tabelle, wird bei der Analyse nicht berücksichtigt. Der Zeitpunkt einer Initialisierung oder Variablenlage innerhalb einer Klasse ist nicht relevant für die Erstellung des RFG.

StandAloneSig

Einträge in der *StandAloneSig*-Tabelle enthalten nur einen Zeiger in den #Blob-Stream. Diese Einträge spezifizieren entweder Methoden oder lokale Variablen. Diese Einträge werden vom Compiler eigens für die VES erzeugt und enthalten ausschließlich Informationen die während der Laufzeit schnell zur Verfügung stehen müssen. Zum Beispiel die lokalen Variablen einer Methode die in einer spezielle Signatur zusammengefasst werden, damit sie bei einem Methodenaufruf geladen werden können. Im RFG werden keine lokalen Variablen sowie Laufzeitinformationen abgebildet, so dass diese Tabelle keiner weiteren Analyse bedarf.

EventMap

Die *EventMap*-Tabelle definiert die Assoziation zwischen Ereignissen und ihrer Klassenzugehörigkeit, dargestellt in Abbildung 4.12.

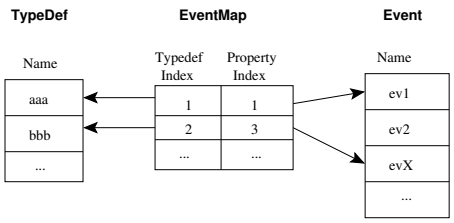

Abbildung 4.12: Zuordnung von Events

Aufeinanderfolgende Einträge in dieser Tabelle grenzen wie beim *FieldList*-Feld der *TypeDef*-Tabelle die Anzahl der zum Typ zugehörenden Events ein. Der Typ wird direkt mit dem Wert des *Parent*-Feldes angegeben.

Feldname	**Beschreibung**
Parent	Zeiger in die *TypeDef*-Tabelle
EventList	Zeiger in die *Event*-Tabelle

Event

Events werden zum Teil als Felder in der *Field*-Tabelle aufgeführt. Jedoch können sie dort nicht von anderen Variablen unterschieden werden. Zur Differenzierung gibt es diese Tabelle. Sie enthält neben dem Namen des Events auch den Zeiger *EventType*, der die Typzugehörigkeit des Events definiert. Das *EventFlags*-Feld enthält keine relevanten Informationen für den RFG. Ereignisse sind wie Variablen in der *Field*-Tabelle enthalten, so dass die Attribute eines Ereignisses aus dem korrespondierenden *Field*-Eintrag entnommen werden können.

Feldname	**Beschreibung**
EventFlags	Bitvektor
Name	Zeiger in den #String-Stream
EventType	Zeiger in die *TypeDef*-, *TypeRef*- oder *TypeSpec*-Tabelle

PropertyMap

Analog zur *EventMap*-Tabelle definiert die *PropertyMap*-Tabelle welche Propertys der *Property*-Tabelle zu welcher Klasse in der *TypeDef*-Tabelle gehören, veranschaulicht in Abbildung 4.13.

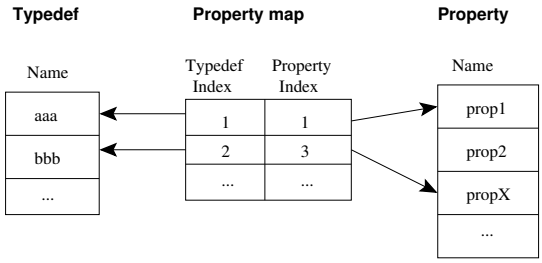

Abbildung 4.13: Zuordnung von Propertys

Feldname	Beschreibung
Parent	Zeiger in die *TypeDef*-Tabelle
Propertylist	Zeiger in die *Event*-Tabelle

Property

Anders als Ereignisse werden Propertys nicht in der *Field*-Tabelle aufgeführt. Sie sind nur in dieser Tabelle definiert und enthalten Property- wie auch Indexer-Definitionen. Mithilfe des Zeigers in den #Blob-Stream kann die Typzugehörigkeit eines Propertys ermittelt werden.

Der Attributvektor kodiert keine Informationen, die für die RFG-Erstellung interessant sind, deswegen müssen die Attributinformationen des Propertys über seine Get- und Set-Methoden gewonnen werden. Die zum Eintrag korrespondierenden Methoden können durch Einträge in der *MethodSemantics*-Tabelle gefunden werden.

Indexer und Propertys sind in dieser Tabelle enthalten. Es gibt aber kein speziell ausgewiesenes Attribut, das eine Unterscheidung ermöglicht. Deswegen muss ein anderer Weg zur Unterscheidung der beiden Konstrukte gefunden werden. Indexer erhalten in der Zwischendarstellung den speziellen Namen *Item,* wenn sie in der *Property*-Tabelle aufgeführt werden. Dieser Name kann, wenn ein Indexer definiert wurde, nicht anderweitig als Variablenname genutzt werden. Dies ist jedoch kein eindeutiges Unterscheidungsmerkmal für Indexer und Propertys, denn wenn kein Indexer definiert wurde kann auch ein Property oder eine Variable mit diesem Namen existieren. Beide Konstrukte können auch nicht anhand ihrer #Blob-Informationen unterschieden werden. Sie enthält keine Information, ob der Eintrag eine Variable oder ein Array kapselt. Die #Blob-Information liefert einzig und alleine die Typinformation die ausschlaggebend für die Get- und Set-Methodensignatur ist. Eine Lösung für die Unterscheidung findet sich durch eine weitere Analyse unter zu Hilfenahme der *MethodSemantics*-Tabelle die im nächsten Abschnitt beschrieben wird.

Feldname	Beschreibung
Attribut	Bitvektor
Name	Zeiger in den #String-Stream
Signatur	Zeiger in den #Blob-Stream

MethodSemantics

Die *MethodSemantic*-Tabelle beschreibt Zusammenhänge zwischen Ereignissen, Indexern und Propertys und deren spezifischen Methoden. Aus dieser Tabelle geht hervor, welche Methode aus der *MethodDef*-Tabelle eine Get- oder Set-Methode für ein bestimmtes Property oder Indexer besitzt sowie welche Methoden ein Ereignis bestellen und abbestellen können, dargestellt in der Abbildung 4.14. Dieser Kontext wird mit dem Bitvektor im *Attribut*-Feld spezifiziert. Bezeichnend für Property- und Indexer-Methoden sind *Getter* und *Setter* sowie für Ereignisse die Attribute *Remove* und *Add*. Die Bitvektor kann zwei weiteren Attribute *Fire* und *Other* dekodieren, allerdings werden diese nicht genutzt.

Innerhalb der Get- und Set-Methodensignaturen für Propertys und Indexer werden keine expliziten Bezeichner für übergebene Variablen definiert. Sie werden vom Compiler generiert und lauten für Propertyfunktionen *Value* und für Indexer *Index*. Bei der Betrachtung der Get- und Set-Methoden eines Propertys über die *MethodSemantics*-Tabelle können die zugehörigen Methoden bestimmt werden. Diese können dann auf die Parameter *Value* und *Index* hin untersucht werden, so dass damit eine eindeutige Unterscheidung von Propertys und Indexern möglich wird.

Feldname	**Beschreibung**
Attribut	Bitvektor
Method	Zeiger in den *MethodDef*-Tabelle
Association	Zeiger in die *Event*- oder *Property*-Tabelle

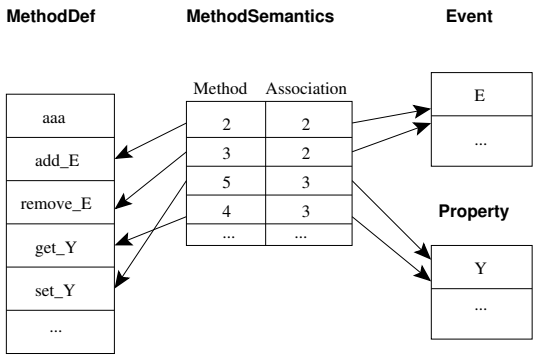

Abbildung 4.14: Aufgabe der *MethodSemantic*-Tabelle

MethodImpl

Erbt eine Klasse aus der *TypeDef*-Tabelle Methoden und implementiert diese, wird dies in der *MethodImpl*-Tabelle festgehalten. Ein Eintrag in der *MethodImpl*-Tabelle enthält den Zeiger *Class* auf die Klasse, die etwas implementiert, den Zeiger *MethodBody* auf die geerbte Methode und den Zeiger *MethodDeclaration* auf die implementierende Methode.

Feldname	Beschreibung
Class	Zeiger in die *TypeDef*-Tabelle
MethodBody	Zeiger in die *MethodDef*- oder *MemberRef*-Tabelle
MethodDeclaration	Zeiger in die *MethodDef*- oder *MemberRef*-Tabelle

ModuleRef

Einträge in dieser Tabelle enthalten nur das Feld *Name*. Der Name eines referenzierten Moduls wird aus dem #String-Stream gelesen. In dieser Tabelle werden Module, die mit *ImportDLL* im Quelltext referenziert werden, aufgeführt.

TypeSpec

Die Spalten der *TypeSpec*-Tabelle enthalten nur ein Signaturfeld, das in den #Blob-Stream zeigt. Mit dem Wert aus dem #Blob-Stream wird dann ein Typ spezifiziert, der im Programmablauf häufig verwendet wird oder die Instanz eines generischen Typen darstellt.

ImplMap

Diese Tabelle enthält Informationen über die Benutzung von *Unmanaged-Code*-Methoden. Einträge in dieser Tabelle geben Auskunft darüber, ob eine Methode der *MethodDef*-Tabelle eine extern definierte Methode ist. Das Softwaremodul, das die Implementierung der Methode enthält, wird mit dem Feld *Importscope* angegeben.

Feldname	Beschreibung
MappingFlags	Bitvektor
MemberForwarded	Zeiger in die *MethodDef*-Tabelle
ImportName	Zeiger in den #String-Stream
ImportScope	Zeiger in die *ModuleRef*-Tabelle

FieldRVA

Ein Eintrag in dieser Tabelle gehört jeweils zu einem Eintrag in der *Field*-Tabelle. Er spezifiziert zusätzlich Informationen zum *Field*-Eintrag, die für die RFG-Erstellung nicht benötigt werden.

Assembly

Assembly enthält Informationen über das aktuell vorliegende Assembly. Innerhalb dieser Tabelle sind die eingangs beschriebenen Assembly-Merkmale zu finden. Dazu gehören Informationen zur Assembly-Erstellung, der *PublicKey* der Assembly-Name und seine

Kultur. Diese Informationen sind im Rahmen der RFG-Generierung nicht relevant.

AssemblyRef

Einträge dieser Tabelle enthalten Informationen über referenzierte Assemblys, die für Objekte dieses Assemblys benötigt werden.

File

Diese Tabelle enthält nur Informationen wenn das vorliegenden Assembly aus mehreren Modulen zusammengesetzt wurde . Hier werden die einzelnen Module mit Attributen, ihrem Namen und einem Hashwert aufgeführt. Die Einträge dieser Tabelle werden nicht betrachtet, da im Rahmen dieser Arbeit nur Assemblys bestehend aus einem Modul behandelt werden (siehe Abschnitt 4.1).

ExportedType

Innerhalb dieser Tabelle werden alle öffentlichen Objekte aufgeführt, die von anderen Assemblys benutzt werden können. Diese Einträge existieren nur, wenn das Assembly eine Bibliothek, zusammengesetzt aus mehreren Modulen, ist und somit Einträge in der *File*-Tabelle vorhanden sind. Diese Tabelle wird wie die *File*-Tabelle nicht für die Generierung des RFG verwendet.

ManifestResource

Die Einträge der *ManifestResource*-Tabellen enthält einen *Offset* ab dem eingefügte Programm-Ressourcen wie Bilder oder andere Dateien innerhalb des Bytestroms gefunden werden können. Diese Tabelle ist für die RFG-Generierung uninteressant, da in der Darstellung solche Informationen nicht abgebildet werden.

NestedClass

Die Spalten dieser Tabelle enthalten zwei Verweise auf Klassen. Das Feld *NestedClass* identifiziert eine geschachtelte Klasse und das Feld *EnclosingClass* zeigt an in welcher Klasse sie liegt.

Feldname	**Beschreibung**
NestedClass	Zeiger in die *TypeDef*-Tabelle
EnclosingClass	Zeiger in die *TypeDef*-Tabelle

GenericParam

In dieser Tabelle sind die deklarierten generischen Platzhalter enthalten. Eine Spalte enthält jeweils einen Platzhalter spezifiziert mit Namen, Position in der Signatur und einen Verweis auf den generischen Typen zu dem es gehört. Diese Tabelle wird zur Unterscheidung von generischen und normalen Typen, die in der *TypeDef*-Tabelle aufgeführt sind, verwendet.

Feldname	**Beschreibung**
Number	Position in der Signatur an der dieser Parameter steht
Flags	Bitvektor
Owner	Zeiger in die *TypeDef*- oder *MethodDef*-Tabelle
Name	Zeiger in den #String-Stream

MethodSpec

Feldname	**Beschreibung**
Method	Zeiger in die *MethodDef*- oder *MemberRef*-Tabelle
Instantiation	Zeiger in den #Blob-Stream

Der *MethodSpec* enthält die Informationen darüber welche Methode eine generische Methode instanziert. Hierbei identifiziert der Zeiger des Feldes *Method* die generische Methode und das Feld *Instantiation* zeigt in den #Blob-Stream zur Signatur der Instanz.

GenericParamConstraint

GenericParamConstraint-Einträge beschreiben bestimmte Typeigenschaften von generischen Parametern. Somit kann ein generischer Parameter einer Klasse z.B. explizit nur für Klassen oder andere Objekte zugelassen werden. Dieser Kontext wird im RFG nicht abgebildet, so dass diese Tabelle nicht weiter analysiert wird.

FieldPtr, MethodPtr, ParamPtr, EventPtr, PropertyPtr, EncLog, EncMap

Diese Tabellen werden in den von dieser Arbeit genutzten optimierten Metadaten nicht generiert und dementsprechend nicht verwendet.

AssemblyProcessor, AssemblyOS, AssemblyRefProcessor, AssemblyRefOS

Diese Tabellen werden selten generiert und enthalten rein informative Felder, die nicht von der VES genutzt werden (siehe [ECM06b] Partition II. S.106-107).

4.3.2.1 Zusammenfassung

Die im vorherigen Abschnitt beschriebenen Verweisinformationen der Metadatentabellen werden in Abbildung 4.15 dargestellt.

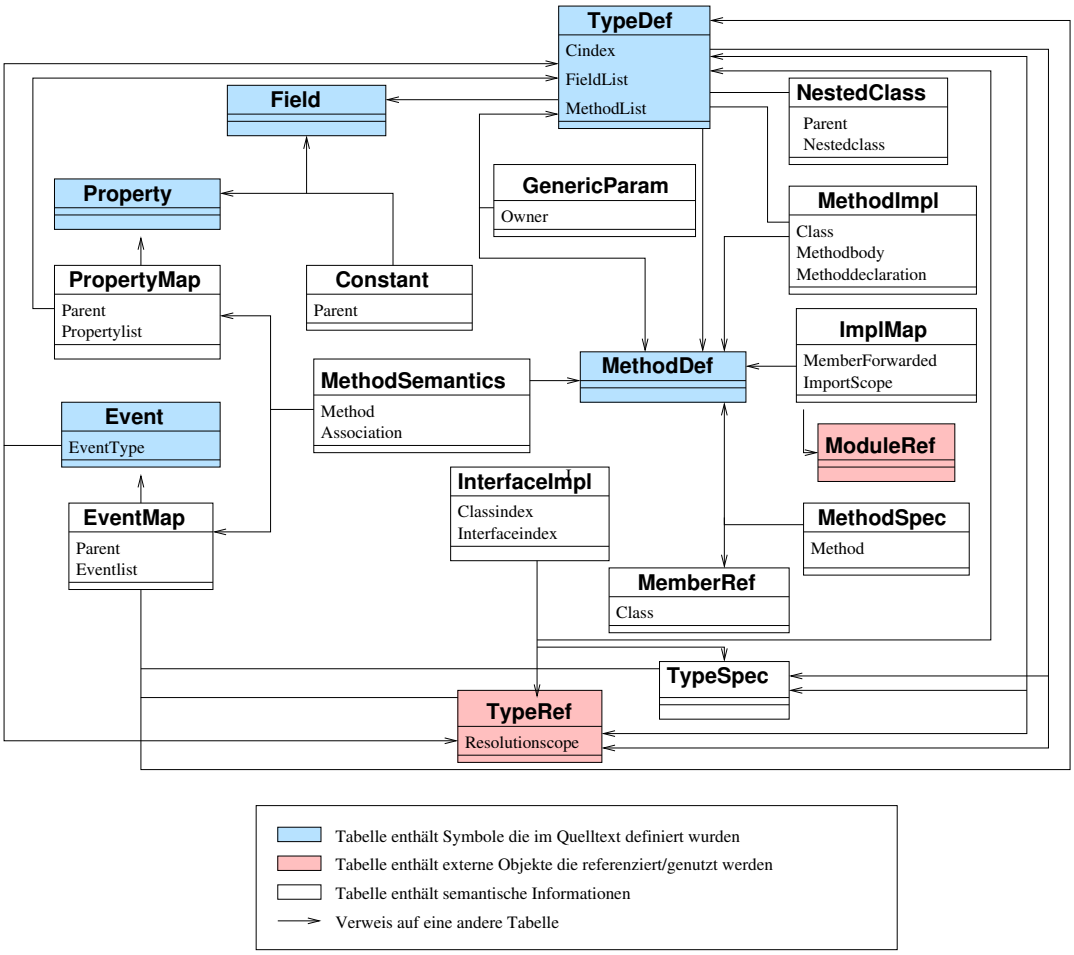

Abbildung 4.15: Übersicht der relevanten Verweise innerhalb der Tabellen

Die Abbildung 4.15 zeigt welche Tabellen im Quelltext deklarierte Objekte und externe Objekte enthalten. Zusätzlich sind Tabellen, die semantische Informationen zu den Objekte enthalten dargestellt. Die Attribute in der Darstellung zeigen an welche Tabellenfelder Zeiger in andere enthalten und die Pfeile visualisieren die Tabellen, die adressiert werden können.

Die dort abgebildeten Verweise und Tabellen bilden die Komposition der im Quelltext deklarierten und extern verwendeten Symbole ab. Sie werden in der RFG-Generierung für die Abbildung der Symbole und ihrer Vererbunghierarchie verwendet.

Zur besseren Übersicht sind die Attribute und Signaturinformationen der einzelnen Tabellen in der Abbildung 4.15 nicht abgebildet. Sie werden aus den zuvor beschriebenen Informationen zu den Tabellen bei der Erstellung der RFG-Elemente miteinbezogen.

4.3.3 Analyse des CIL-Codes

Der CIL-Code ist wie alles andere in einem *Portable Executable* auch ein Bytestrom. Er gehört jeweils zu einer Methode aus der *MethodDef*-Tabelle. Insgesamt gibt es 216 assemblerähnliche Anweisungen die im CIL-Code enthalten sein können. Die Anweisungen sind entweder in einem Byte kodiert oder setzen sich aus zwei aufeinander folgenden Bytes zusammen. Zusammengesetzte Instruktionen werden im ersten Byte mit der Zahl *254* gekennzeichnet.

Auf Anweisungen können verschiedene Werte folgen, dargestellt in Abbildung 4.17. Nach einer *ldgarg.0* Anweisung folgt gleich eine weitere Anweisung, hingegen bei einer *call* Anweisung kodieren die folgenden vier Bytes die Funktion, die aus den Metadatentabellen aufgerufen werden soll. Diese vier Bytes werden Token genannt. Die ersten drei Bytes repräsentieren die Spalte und das vierte Byte die Tabelle. In der Abbildung 4.17 kodiert das Token Spalte 1 der *MemberRef*-Tabelle (Tabelle 10). Darauf folgt dann die nächste Instruktion.

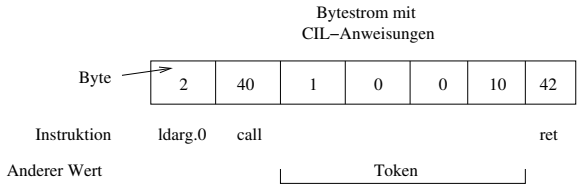

Abbildung 4.16: Der CIL Bytecode

Neben den vorgestellten Token können hinter Instruktionen Zahlenwerte mit ein bis acht Byte kodiert sein. Eine Ausnahme bildet die Anweisung für *Switch*-Konstrukte, auf sie können N+1 vier Bytefelder folgen.

Nicht alle Anweisungen sind interessant im Hinblick auf die Generierung des RFG. Jedoch müssen für die Analyse alle Instruktionen erkannt und die darauf folgenden Werte gelesen werden. Wird dies nicht berücksichtigt, können Instruktionen an Stellen identifiziert werden, die eigentlich einen Wert oder ein Token repräsentieren.

Im Rahmen des im Kapitel 3 aufgestellten C#-RFG sind die Kanten *Member_Set*, *Member_Use*, *Member_Address*, *Static_Call* und *Dispatching_Call* sowie die Ereignisbehandlung im Mittelpunkt der CIL-Code Analyse. Zusätzlich muss für Variablen die Eigenschaft *Volatile* aus dem CIL-Code gewonnen werden. Diese Variableneigenschaft kann wie schon beschrieben, nicht durch das *Flag*-Feld der *Field*-Tabelle bestimmt werden.

Gewinnung der *Member_Set*, Member_Use- und *Member_Address*-Kanten

Das Verarbeiten von methodeninternen Variablen und externen Variablen wird im CIL-Code klar durch bestimmte Instruktionen unterschieden. Aus diesem Grund können RFG-relevante Variablenzugriffe eindeutig anhand der dazugehörigen Instruktion

identifiziert werden. Welche Variable durch die Instruktion manipuliert wird, kann dem darauf folgenden Token entnommen werden.

Anweisung	**Beschreibung**
ldfld	Sie zeigt an, dass der Inhalt einer Variable aus der *Field*-Tabelle auf den Methodenstack geladen wird.
ldsfld	Mit dieser Anweisung wird der Inhalt eines statischen *Field*-Eintrags geladen.
stfld	Speichert etwas in eine Klassenvariable zurück.
stsfld	Speichert etwas in eine statische Klassenvariable zurück.
ldflda	Läd einen Zeiger auf den Methodenstack.
ldsflda	Läd den Zeiger einer statischen Variable.

Tabelle 4.27: CIL Anweisungen die Klassenvariablen manipulieren

Die ersten zwei Anweisungen aus der Tabelle 4.27 entsprechen der *Member_Use*-Kante. Die darauf folgenden zwei Anweisungen sind vergleichbar mit der *Member_Set*-Kante und die letzten beiden mit der *Member_Address*-Kante.

Identifizierung von Variablen mit *Volatile*-Attribut

Das *Member*-Attribut *volatile* kann nicht anhand des *Flag*-Feldes der *Field*-Tabelle bestimmt werden. Es wird durch eine eigene CIL-Anweisung repräsentiert. Sie setzt sich aus zwei Bytes zusammen und ist jedesmal vor einer der oben aufgeführten Variablenmanipulationsinstruktionen zu finden. Dadurch kann bei einer Analyse der *Member*-Manipulationen das entsprechende Attribut im RFG gesetzt werden.

Gewinnung der *Call*-Kanten

Direkte Aufrufe von Methoden lassen sich innerhalb des CIL-Code anhand der Instruktionen *Call* und *Callvirt* im CIL-Code identifizieren. Die Instruktion *Call* ist gleichzusetzen mit einem *Static_Call* im RFG sowie *Callvirt* mit *Dispatching_Call*. Auf beiden Instruktionen folgt jeweils ein Token, das die aufgerufene Methode angibt.

Abbildung des Ereignis-Schemas

Die Analyse hinsichtlich der Kanten aus dem Ereignis-Teilschema Abbildung 3.3 ist etwas aufwändiger. Für diese Analyse ist die Stapelverarbeitung der CIL-Anweisungen von Vorteil. Ähnlich zum beschriebenen *Volatile*-Attribut das vor der geladenen Variable geladen wird, müssen auch andere Instruktionen in einer bestimmten Reihenfolge auf den Methodenstack gelegt werden.

Diese Reihenfolge beginnt für die Analyse der Ereigniskanten mit der Anweisung *ldnull*. Darauf folgt die Anweisung *ldftn*. Mit diesen beiden Anweisungen wird ein Funktionszeiger zu der nach *ldftn* folgenden Funktion erzeugt. Dieser Zeiger wird dann mittels *newobj* in ein neues Delegatenobjekt übergeben. Darauf folgt der Aufruf einer *Event_add*-Funktion, die mit *callvirt* eingeleitet wird. Die Abfolge dieser Aufrufe ermöglicht das Setzen der *Handled_By*-Kante.

Klassen, die Events bereitstellen, können innerhalb einer Methode oder als *Member* instanziiert werden. Dadurch wird eine zusätzlich Analyse nach der Identifizierung der *Handled_By*-Kante nötigt. Dafür müssen im CIL-Code sechs Bytes vor der *ldftn*-Anweisung gelesen und geprüft werden, ob ein Feld mit der Anweisung *ldfld* oder *ldsfld* auf den Methodenstack geladen wurde. Dies heißt im Kontext des Events, dass der *callvirt*-Aufruf von einer Instanz aus getätigt wurde und das Event zu diesem Member gehört. Dies ermöglicht ein setzen der *Handles_Event_Of*-Kante. Sollte kein Feld vorher geladen werden, ist davon auszugehen, dass die Instanz deren Event behandelt werden soll lokal in der Methode erzeugt wurde und somit keine *Handles_Event_Of*-Kante gesetzt werden muss.

Die *Raises_Event*-Kante wird gesetzt, wenn eine Methode einen Member läd, der ein Event ist. Hierbei muss in der Analyse jedoch geprüft werden, ob diese Methode einen Eintrag in der *MethodSemantics*-Tabelle hat. Sollte dies der Fall sein, wird keine entsprechende Kante gesetzt, weil diese Methoden auch Ereignisfelder laden und speichern, sie aber keine entsprechende Delegaten-Methode danach anstoßen, die letztlich für das Anstoßen des Ereignisses verantwortlich ist.

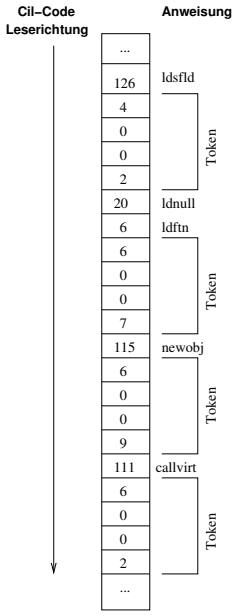

Abbildung 4.17: Ausschnitt des CIL Bytecode für die Ereignisbehandlung

4.4 Gewinnung von Quelltextpositionen

Aus den bisher beschriebenen Informationen der Metadaten geht hervor, dass diese keine Informationen zu Quelltextpositionen enthalten. Auch wenn es in der *Portable Executable* Spezifikation ein Abschnitt mit dem Namen *.debug* gibt wird dieser von Compilern, die nach 2003 erschienen sind, nicht mehr genutzt (siehe [Mic06b] S.20,[ECM06b, Mica]).

Dennoch wäre es schön wenn man zu der RFG-Darstellung Bezüge auf den Quelltext hätte. Dies wird vom RFG-Visualisierungsprogramm unterstützt, so dass von einem RFG-Knoten in den dazugehörigen Quelltext gesprungen werden kann. Dafür müssen aber bestimmte Knotenattribute gesetzt werden die dies ermöglichen. Daraus ergibt sich neben der Analyse der Zwischendarstellung noch eine weiter Aufgabe, die Gewinnung von Quelltextpositionen der in den Metadaten definierten Objekte.

Zur Gewinnung der fehlenden Informationen sind zwei Ansätze denkbar: ein Sprachunabhängiger und ein Sprachabhängiger. Beim sprachabhängigen Ansatz müsste für jedes Assembly die entsprechende Sprache zur Analyse der Quelltextpositionen ausgewählt werden können. In einem Assembly sind jedoch keine Informationen über die Quelltextsprache, aus der die Zwischendarstellung erzeugt wurde, enthalten. Außerdem müsste ein passender Quelltextparser gebaut werden, der abhängig von der jeweiligen Programmiersprache die entsprechenden Quelltextinformationen liefert. Dies ist bei fast 70 CLI-konformen Sprachen (siehe [Sch07]) keine adäquate Lösung. Aus diesem Grund wird im nächsten Abschnitt ein sprachunabhängiger Ansatz unter zu Hilfenahme von Debuginformationen beschrieben.

4.4.1 Quelltextpositionsgewinnung mit Debuginformationen

Die beiden ausgereiftesten CLI-Implementierungen, das .NET-Framework und das Mono-Projekt, stellen eigene Debuginformationen für die Fehleranalyse in Programmen bereit. Für die Analyse der Debuginformationen werde ich mich auf die Debuginformationen des Visual Studios beschränken, da dies primär zur Erstellung der Assemblys, die in dieser Arbeit verwendet werden, genutzt wird.

Die Debuginformationen enthalten Informationen zu Programmausführung. Damit kann im Rahmen des Visual Studios mit einem speziellen JIT-Compiler als Debugger schrittweise durch Quelltextpositionen nach einem Breakpoint[7] gesprungen werden. Debuginformationen werden beim Compilieren des Quelltextes erzeugt. Das Konzept ist dargestellt in Abbildung 4.18.

[7] Spezielle Markierung zum Beginn des visuellen Debuggens

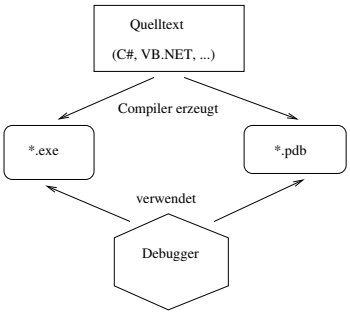

Abbildung 4.18: Debuggerkonzept des Visual Studios

Die erstellten Debuginformationen liegen in einer separaten Datei in einem eigenen binären Format vor. Dieses proprietäre Format hat keine öffentliche Spezifikation. Deswegen basiert das Analyseverfahren auf den Informationen von [Sta05, Jor06] zum Debugformat und wie es mittels *Reflection*-Bibliothek analysiert werden kann.

Unter der Verwendung der *Reflection*-Bibliothek des .NET-Frameworks können die darin enthaltenen Informationen in Kombination mit dem dazugehörigen Assembly analysiert werden. Methoden die CIL-Code enthalten können positionsgenau mit Zeile und Spalte ermittelt werden, sowie ihre enthaltenen Instruktionen. Zusätzlich kann die Datei in der die Methode definiert wurde ermittelt werden. Ausgehend von der Methode kann die Klasse in der sie definiert wurde sowie die enthaltenen Variablen, Ereignisse und Propertys gefunden werden. Für diese Elemente kann keine genaue Quelltextposition ermittelt werden, aber zumindest kann die Datei angegeben werden, in der, abhängig von der Methode, die Klasse definiert wurde.

Auf Basis der Beispiele in [Sta05, Jor06] ist ein prototypisches Werkzeug für die Gewinnung von Quelltextinformationen unter zu Hilfenahme des Visual Studio Debugformats entstanden. Die gewonnenen Quelltextinformationen werden in einer XML-Datei nach dem im Anhang C beschriebenen Schema abspeichert. Damit später die Informationen der XML-Datei und der Metadaten bei der RFG-Generierung wieder zusammengefügt werden können, werden zu jedem Objekt in der XML sein Tokenwert in Tabelle und Spalte aufgeschlüsselt gespeichert. Wie schon in der CIL-Analyse beschrieben wurde, kodiert ein Token die Tabelle und die Spalte eines Objektes. Dies bildet innerhalb der Metadaten eines Assemblys einen eindeutigen Bezeichner.

Nachteilig an dieser Art von Quelltextpositionsgewinnung ist, das nur Quelltextinformationen über Methoden mit CIL-Code in den Debugdateien enthalten sind. Somit können die tatsächlich ausgeführten Methoden und ihre Klassen einer Datei zugeordnet werden, jedoch können abtraktere Objekte wie Schnittstellen oder auch abstrakte Klassen keiner Datei zugeordnet werden. Letztlich kann für die Klassen und seine *Member* auch nur geschätzt werden das sie in der Datei von der Methode deklariert wurden, denn wie schon in Abschnitt 2.3.1 angesprochen wurde können Klassen auch über mehrere Dateien hinweg definiert sein. So ein Fall lässt sich weder über die Metadaten noch über das

Debugformat feststellen.

KAPITEL 5

Generierung des RFG

In diesem Kapitel wird das Vorgehen zur Erstellung des RFG beschrieben. Der RFG selbst ist im Bauhausprojekt in Form einer Ada-Bibliothek realisiert, die bereits alle benötigten Funktionen zum Aufbau des Graphen bereitstellt.

Zur Generierung des RFG werden die aus der Analyse gewonnenen Informationen verwendet. Einige Informationen können explizit aus den ausgelesenen Informationen der Metadaten übernommen werden, andere werden erst zur Erstellung des RFG ausgewertet.

Quelltextsymbole und Teile der Vererbungshierarchie werden in den Metadaten explizit modelliert (vgl. Abbildung 4.15). Mit diesen Informationen ist es werden die Basisknoten wie *Namespaces*, Klassen und Methoden und ihre Beziehungen im RFG abgebildet. Im Anschluss werden die Kanten, die nicht aus diesen Informationen gewonnen werden können, aus den in Abschnitt 4.3.3 beschriebenen CIL-Informationen erzeugt. Abschließend gilt es noch die nötigen Sichten zu erzeugen sowie das Linken mehrerer RFGs zu einem Gesamtgraphen zu ermöglichen.

Kapitelinhalt

5.1 Vorgehen bei der RFG-Generierung 90

5.1.1 Umsetzung der Informationen der Metadaten 90

5.1.2 Laufzeitinformationen aus den Methoden 91

5.2 Sichten 92

5.3 Linken 92

5.1 Vorgehen bei der RFG-Generierung

Die Generierung des RFG ist, nach dem die Metadaten ausgelesen sind, ein relativ einfacher Vorgang. Die Datenrepräsentation der Metadaten wird so verwendet wie sie ausgelesen wurde, in Tabellenform. Die Tabelleneinträge wurden lediglich um die Analyseergebnisse aus Abschnitt 4.3 ergänzt.

Zuerst werden die Symbolinformationen der Metadatentabellen sowie der hierarchischen Aufbau im RFG abgebildet. Anschließend wird der CIL-Code auf den Methoden der *MethodDef*-Tabelle analysiert um *Aufruf*-, *Get*- und *Set*-Kanten zu setzen. Zusätzlich wird bei der Generierung, wenn eine XML-Datei mit Informationen zu den Quelltextpositionen vorliegt, automatisch die Quelltextposition zum RFG-Element hinzugefügt.

5.1.1 Umsetzung der Informationen der Metadaten

Die RFG-Generierung beginnt mit der Namensraumerzeugung. Dazu werden die *TypeNamespace*-Felder der Tabellen *TypeRef* und *TypeDef* verwendet. Geschachtelte Namensräume werden dort mit einem Punkt getrennt dargestellt und als einzelne *Namespace*-Knoten abgebildet, deren Beziehung mit der *Enclosing*-Kante dargestellt wird.

Nachdem die Namensräume im RFG enthalten sind, werden aus den Einträgen der *TypeRef*-Tabelle *Type*-Knoten erzeugt und dem im Feld *TypeNamespace* angegebenen Namensraum mit der *Enclosing*-Kante zugeordnet.

Anschließend werden die in der *TypeDef*-Tabelle aufgeführten Objekte erzeugt. Die *TypeDef*-Tabelle enthält nicht nur Klassen, so dass vor der Erzeugung geprüft werden muss, ob es sich um eine Klasse, Struktur, generische Schnittstelle, generische Klasse, generische Struktur oder eine Aufzählung handelt. Dazu wird das Feld *Extends*, das Feld *Flags* und die Einträge der *GenericParam*-Tabelle untersucht. Anhand der gewonnen Informationen wird der entsprechende Knoten im RFG erzeugt. Die Basisklasse des Objektes wird dem Feld *Extends* entnommen und zum Setzen der *Extends*-Kante verwendet. Geschachtelte Klassen werden aus der *NestedClass*-Tabelle herausgesucht und mit der *Enclosing*-Kante der Klasse in der sie enthalten sind zugeordnet. Die Knotenattribute werden anhand der dekodierten *Flag*-Informationen und bei *Structs*, Delegaten und *Enums* anhand ihrer Basisklasse gesetzt.

Nachdem die leere Klasse im RFG hinzugefügt wurde, werden die in ihr enthaltenen Felder ermittelt. Die Felder einer Klasse werden durch ihr *FieldList*-Feld gekennzeichnet und können durch das Ermitteln des nächsten *FieldList*-Feldes in der nächsten *TypeDef*-Spalte herausgefunden werden. Die beiden *FieldList* Einträge geben an welche Einträge aus der *Field*-Tabelle zu dem aktuellen *TypeDef*-Eintrag gehören. Neben Variablen und Konstanten werden dort auch Ereignisse aufgeführt. Ereignisse werden durch den Eintrag in der *EventMap*-Tabelle der deklarierenden Klasse zugeordnet. Dies ermöglicht durch einen Namensvergleich der Felder und zur Klasse eingetragenen Ereignisse eine Variable von einem Event zu unterscheiden und mit unterschiedlichen Knoten zu erzeugen. Konstante

Felder werden durch ihren Eintrag in der *Constant*-Tabelle ausgezeichnet und mit dem Attribut *Is_Const* versehen. Weitere Attribute werden durch die im *Flag*-Feld angegebenen Attribute im RFG abgebildet. Der Typ des *Members* wird mit der dekodierten #Blob-Information erzeugt.

Das Hinzufügen der Methoden funktioniert analog zu den Feldern. Bei Methoden muss geprüft werden, ob es sich um eine generische Methode handelt. Dies kann mit einem Blick in die *GenericParam*-Tabelle überprüft werden. Sollte die Methode einen generischen Parameter enthalten, wird sie als *Method_Template* im RFG abgebildet, ansonsten als normale Methode. Die Attribute der Methoden werden anhand ihres *Flag*-Feldes gesetzt. Der Rückgabewert und die Parameter der Methode werden aus den ausgelesenen #Blob-Informationen gesetzt.

Propertys und Indexer werden in der *Field*-Tabelle nicht aufgeführt. Sie sind separat in der *Property*-Tabelle vertreten und werden anhand der Zuordnunginformationen der *PropertyMap*-Tabelle als *Member* zu der jeweiligen Klasse hinzugefügt. Die Attribute dieser *Member* werden anhand der schon beschrieben Analysemöglichkeiten gesetzt.

Mit den Informationen aus der *InterfaceImpl*-Tabelle werden die Kante *Implementation_Of* zwischen Klassen und Schnittstellen gesetzt. Externe Methoden werden in der *ImplMap*-Tabelle deklariert. Sie haben einen Zeiger in die *MethodDef*-Tabelle, die die Signatur der referenzierten Methode und ihre Attribute wiedergibt. In Verbindung mit dem dazugehörigen Eintrag der *ModuleRef*-Tabelle wird die referenzierte extern DLLs als *Type* abgebildet und erhält die externe Methode. Beide Methoden werden mit einem *Static_Call* verbunden, damit verdeutlicht wird, das der Aufruf an ein externes Objekt weitergeleitet wird.

Aus den Metadaten und ihren Verweisen können leider keine expliziten Informationen zu überschriebenen oder geerbten Variablen und virtuellen Methoden entnommen werden. Deswegen werden diese Informationen durch einen Abgleich der Variablen und Methoden innerhalb der Vererbungshierarchie bestimmt. Die Übereinstimmung bei Namen und Signatur/Typ wird bei geerbten *Membern* mit einer *Overrides*-Kante im RFG abgebildet. Damit Ereignisse, Propertys und Indexer als spezielle Konstrukte auch mit diesen Kanten versehen werden können, werden die dazugehörigen Methoden betrachtet und die jeweilige Vererbungseigenschaft für das Konstrukt übernommen.

5.1.2 Laufzeitinformationen aus den Methoden

Nachdem alle statischen Informationen aus den Tabelle abgebildet wurden, fehlen noch die Kanten *Call*, *Raises_Event*, *Handles_Event* und die *Member*-Zugriffe. Die dazu benötigten Informationen befinden sich im CIL-Code spezifiziert durch die *MethodDef*-Tabelleneinträge. Zur Gewinnung der Kanteninformationen wird anhand der in Abschnitt 4.3.3 beschrieben Schlüsselwörter der Bytecode durchlaufen und die entsprechenden Kanten gesetzt.

5.2 Sichten

Innerhalb des Bauhausprojektes gibt es neben der Ada-Bibliothek zur Generierung eines RFG auch Bibliotheken, die das automatische Erzeugen der RFG-typischen Sichten ermöglichen. Hierbei dient die Base-View als Ausgangspunkt, denn sie enthält alle Informationen über das analysierte Assembly.

Leider ist das Ergebnis der automatischen *Environment-View*-Generierung nicht sehr zufriedenstellend (vgl. Abbildung 5.1), denn es werden neben den extern verwendeten Programmteilen auch intern definierte Programmteile angezeigt. Aus diesem Grund wird neben der *Base-View* auch die *Environment-View* erzeugt. Die *Environment-View* erhält hierbei alle Einträge aus der *TypeRef*- und *ModuleRef*-Tabelle, die beide extern genutzte Objekte abbilden.

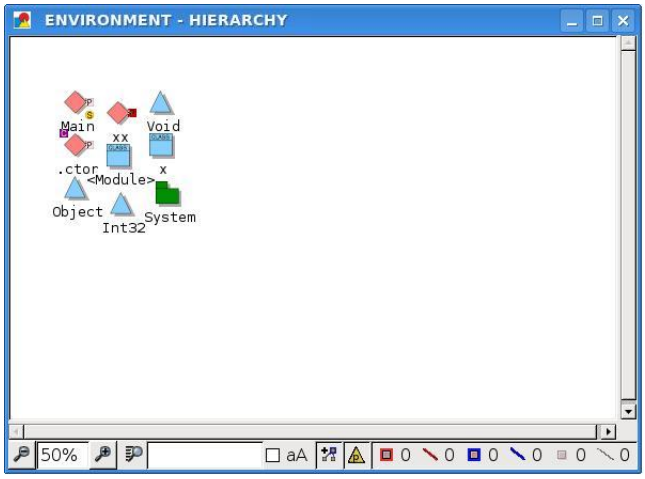

Abbildung 5.1: Eine automatisch erzeugte Environment-View

Für das C#-Schema sind außer des *Interface_Template*-Knotens und der Kanten zum Event-Abonnieren und -Abbestellen keine weiteren Elemente zum RFG hinzugekommen. Die hinzugefügten Kanten wurden von *Member_Set* abgeleitet und bilden keine Aufrufe ab. Dadurch lässt sich der vorhandene Algorithmus zur Erzeugung der *Call-View* ohne weiteres verwenden.

C#-Programme sind in Namensräumen strukturiert. Diese Struktur wird schon in der *Base-View* abgebildet, so dass keine separate *Module-View* erzeugt werden muss.

5.3 Linken

Das in dieser Arbeit entwickelte Werkzeug ist nur dazu ausgelegt, ein einziges Assembly zu analysieren und in einem RFG abzubilden. Damit bei größeren Softwaresystemen mit vielen Komponenten eine Gesamtarchitektur des Ganzen erstellt werden kann, gibt es im Bauhausprojekt das Hilfsprogramm *rfglink*. Dieses Hilfsprogramm fügt mehrere RFGs zu

einem zusammen, beschrieben in [Axi06].

Damit das Linken der einzelnen RFGs funktioniert, muss jeder Graph Informationen über die Bereiststellung seiner Knoten haben. Hierbei ist zum Einen wichtig, ob ein Knoten eine Definition innerhalb des RFG darstellt oder nicht. Diese Eigenschaft wird mit dem Attribut *Linkage.Is_Definition* repräsentiert. Diese Markierung reicht nur aus, um gleiche Symbole in verschiedenen RFGs zu unterscheiden, so dass nur die Definition in den gelinkten RFG übernommen werden. Zum Anderen ist auch ein eindeutiger Name der Objekte wichtig. Dieser wird mit dem Attribut *Linkage.Name* an jedem Objekt dargestellt. Er ist der eindeutige Bezeichner für alle RFG-Knoten und wird anhand der Objektinformationen des Knotens gebildet. Durch Programmstrukturierung in Namensräumen und der Eindeutigkeit der dort enthaltenen Bezeichnern kann unter zur Hilfenahme von Trennzeichen ein eindeutiger Namen zum Linken erstellt werden. Dazu wird die Umgebung betrachtet in der ein Objekt definiert wurde. Dazu werden die Namen der Umgebungen wie Namensräume oder Klassen durch Trennzeichen voneinander getrennt. Beispielsweise erhält ein Feld *F*, das in der Klasse *C* innerhalb des Namensraums *N1* deklariert wurde, den *Linkage.Name* `N1.C#F`.

Wenn diese Informationen im RFG vorhanden sind, kann der Linker im Allgemeinen ohne Weiteres für die aus Assemblys erzeugten RFGs verwendet werden.

KAPITEL 6

Test

Im Kapitel 4 und Kapitel 5 wurde die Analyse und die RFG-Erzeugung beschrieben. In diesem Kapitel wird das entstandene Werkzeug hinsichtlich Qualität und Robustheit untersucht. Dazu wird das Werkzeug verschiedenen Tests unterzogen. Zuerst wird die Extraktion der Assemblyinformationen überprüft. Im Abschnitt 6.2 wird überprüft, wie gut die in Kapitel 3 definierten Anforderungen im RFG umgesetzt wurden. Im Anschluss wird das Werkzeug hinsichtlich der Verarbeitung komplexerer Softwaresysteme getestet.

Kapitelinhalt

6.1 Test der Informationsextraktion **95**

6.2 Test der RFG-Erzeugung **96**

6.1 Test der Informationsextraktion

Zur Verifikation der ausgelesenen Bytes der Zwischendarstellung innerhalb der Assemblys wurden verschiedene Disassembler verwendet. Viele Disassembler sind frei verfügbar oder werden bei CLI-Implementationen mitgeliefert (vgl. Abschnitt 2.2.2). Mitgelieferte Disassembler sind meist auf das annähernde Wiederherstellen der Quelltextrepräsentation spezialisiert und liefern so nur Informationen zu den enthaltenen Metadaten. Eine Ausnahme bildet der Disassembler *ildasm* für das .NET-Framework von Microsoft. Dieses Werkzeug kann die kompletten Informationen eines Assemblys ausgeben. Neben *ildasm* habe ich zusätzlich Disassemblerprogramme des Mono-Projektes verwendet. Die verwendeten Programme haben keine einheitliche Ausgabe, so dass die Werte der Implementierung manuell mit den Disassemblerausgaben verifiziert wurden.

Die in Abschnitt 4.2.1 beschriebenen Header und ihre enthaltenen Werte wurden mit den Ausgaben der Programme *Ildasm* und *pedump*[1] verglichen. Hierbei wurden die *File Header* Informationen zum Beginn des PE-Dateiformates bei Stichproben jedes mal richtig ausgelesen.

Die Gewinnung und Dekodierung der Metadaten, beschrieben in Abschnitt 4.3.2, wurde mit

[1] Mono-Disassembler für den Aufbau eines PEs

kleinen Programmbeispielen nachvollzogen. Dazu wurden kleine C#-Programme geschrieben und Beispiele aus [Mic05b] und [Mic05a] verwendet. Sie bilden jeweils bestimmte Programmteile oder Eigenschaften in den Metadaten ab, die für die Analyse relevante Teile enthalten. Die Testbeispiele wurden mit dem Compiler des Visual Studio 2005 und dem Compiler des Mono-Projektes (Version 1.2.6) erzeugt. Spezielle wurden mit diesen Beispielen die Verweise innerhalb der Metadatentabellen, die Extraktion der Signaturen aus dem #Blob-Stream und die Zusammensetzung der CIL-Instruktionen überprüft. Zur Überprüfung wurden die extrahierten Informationen mit den Ausgaben von *Ildasm* und *monodis*[2] verglichen. Die zur RFG-Generierung benötigten Informationen wurden in den verwendeten Beispielen zu 100% richtig ausgelesen und dekodiert.

6.2 Test der RFG-Erzeugung

Die Richtigkeit der Erstellung des RFG wurde mit kleinen Spezialfällen getestet, die die aus dem Abschnitt 3.3 gewonnen Anforderungen abdecken sollen. Sie sind angelehnt an die in Abschnitt 2.3 vorgestellten Sprachbesonderheiten. Zusätzlich wurden stichprobenartig Beispiele aus [Mic05b] und [Mic05a] getestet. Bei diesen Tests wurden alle Quelltextsymbole und Beziehungen korrekt im RFG abgebildet.

Microsoft veröffentlicht neben den oben genannten Programmbeispielen auch Beispiele zur Benutzung der Entwicklungsumgebung Visual Studio. Sie umfassen jeweils 101 Beispiele zu einer unterstützten Programmiersprache. Diese Beispiele decken unterschiedliche Besonderheiten des .NET-Frameworks und der unterstützten Programmiersprache ab. Zum Testen konkreterer Beispiele wurde die 101 C#-Programmbespiele für Visual Studio 2003 (VS03) und Visual Studio 2005 (VS05) verwendet. Bei der konkreten Auswahl der Testbeispiele wurden nur Beispiele verwendet die eine Windows-Applikation erstellen, da sich das entwickelte Werkzeug wie in Abschnitt 2.3 beschrieben nur mit Windows-Applikationen auseinander setzt. Die Beispiele zum VS03 wurden in Verbindung mit dem .NET-Framework 2.0 und die Beispiele zum VS05 mit dem .NET-Framework 3.0 erstellt. Diese Unterscheidung ist wichtig, weil die Beispiele teilweise versionsspezifische Funktionen des .NET-Frameworks verwenden. Insgesamt wurde mit 140 Beispielen getestet. Sie umfassen jeweils zwischen 280 und 2600 Zeilen Quelltext und verwenden die unterschiedlichsten Besonderheiten der .NET-Frameworks sowie der Sprache C#.

[2] Metadaten-Disassembler des Mono-Projekt

Beispiele zu	Aus Bereich	Anzahl	Durchschnittliche Größe in SLOC	Korrekte Stichproben
VS03	Base Class Library	27	448	35 von 40
VS03	Data Access	15	826	28 von 30
VS03	Windows Forms	13	851	25 von 26
VS05	Data Access	12	1409	22 von 24
VS05	Advanced .NET	14	1121	25 von 28
VS05	Language	7	995	15 von 15
VS05	.NET-Framework	9	875	20 von 20
VS05	Windows Forms	22	983	44 von 44
VS05	Security	4	776	8 von 8
VS05	Framework	17	920	31 von 34

Tabelle 6.2: Testergebnisse des entwickelten Werkzeugs mit übersetzten C#-Quelltextbeispielen zu Visual Studio 2003 und 2005

Die Tabelle 6.2 zeigt die Ergebnisse der RFG-Generierung mit den Testbeispielen. Die Testbeispiele sind in verschiedene Bereiche unterteilt, die jeweils verschiedene Beispiele zu einem bestimmten Kontext enthalten. Beispielsweise umfasst der Bereich *Base Class Library* Beispiele zur Verwendung der .NET-Klassenbibliotheken. Zur Übersichtlichkeit wurden die Beispiele nicht im einzelnen aufgeführt, sondern in den Bereichen zusammengefasst. Der Quelltextumfang der Testbeispiele wurde in SLOC (Source Lines Of Code) ohne Kommentare gemessen und wird als durchschnittliche Größe der Beispiele im jeweiligen Bereich angegeben. Zum Abschluss wurden die entstandenen RFGs stichprobenartig mit zwei Proben pro RFG überprüft, deren Ergebnis in der Spalte Korrekte Stichproben dokumentiert ist. Das Testergebnis zeigt, das die Sprach- und .NET-Frameworkbesonderheiten mit dem Werkzeug recht gut abgebildet werden.

Programm	SLOC	Zeit	Korrekte Stichproben
DaveChart Library[a]	1617	0,25 sec	5 von 5
ZedGraph[b] 4.6.4	26308	14 sec	5 von 5
ZedGraph 5.1.4	26414	14 sec	3 von 5
Quartz.NET 0.6[c]	21191	12 sec	4 von 5
Netz[d]	4148	0,6 sec	5 von 5

[a] `http://www.3dman.org/chart/`
[b] `http://sourceforge.net/projects/zedgraph/`
[c] `http://quartznet.sourceforge.net/`
[d] `http://madebits.com/netz/index.php`

Tabelle 6.3: Testergebnisse des entwickelten Werkzeugs mit Open Source Programmen

Die Beispiele aus der Microsoftkollektion sind vom Quelltextumfang relativ klein. Aus

diesem Grund wurden fünf größere C#-Programme zum Testen der Programmrobustheit verwendet. Einige der in Tabelle 6.3 ausgeführten Programme sind auf mehrere Assemblys verteilt, so dass alle Teile einzeln getestet und die Zeit akkumuliert wurde. Die durchschnittliche Generierungszeit wurde in einer Testumgebung mit folgenden Systemeigenschaften gemessen.

- Prozessor: Intel Core Duo 2,0GHz
- Hauptspeicher: 2GB
- Betriebssystem: (32-Bit) Linux

Das entstandene Werkzeug konnte aus allen aufgeführten Beispielen einwandfrei in einen RFG erzeugen. Die Erzeugung ist sogar bei etwas größeren Programmen noch sehr performant. In den Stichproben wurden insgesamt keine Fehler bezüglich der definierten Quelltextsymbole wie Klassen oder Namensräume entdeckt. Es fehlen jedoch einige Aufrufkanten. Ihre Abwesenheit könnten daher rühren, dass einige Methoden keinen CIL-Code haben und somit keine von dieser Methode ausgehenden Aufrufe abgebildet werden können.

KAPITEL 7

Fazit

Nachdem in den vorangegangenen Kapiteln Grundlagen, Anforderungen und die konkrete Realisierung der Aufgabe beschrieben wurden, soll dieses Kapitel einen Überblick über die Diplomarbeit geben. Abschnitt 7.1 beschreibt kurz erreichten Ziele. Im Anschluss werden die Schwachstellen des entstandenen Werkzeugs reflektiert. Abschließend werde ich in Abschnitt 7.3 ein paar Ideen zur Weiterentwicklung des entstandenen Werkzeugs aufzeigen.

Kapitelinhalt

7.1 Überblick **99**

7.2 Schwachstellen **99**

7.3 Weiterentwicklung **100**

7.1 Überblick

Das Ergebnis dieser Arbeit ist ein Werkzeug, das die in Assemblys enthaltenen Programminformationen in einem RFG abbildet. Zur Abbildung der Besonderheiten der Sprache C# wurde das RFG-Modell hierfür geringfügig erweitert. Betrachtet man die Testergebnisse aus Abschnitt 6.2 deckt das entstandene Werkzeug fast gänzlich die im Quelltext deklarierten Elemente und Beziehungen ab.

Zusätzlich wurde eine Lösung für die Gewinnung der fehlenden Quelltextinformationen gefunden. Die Lösung ist sprachunabhängig und verwendet Informationen die zum Debuggen eines Programms verwendet werden. Zur Gewinnung der Quelltextinformationen wurde ein eigenes Werkzeug entwickelt, das die Quelltextinformationen separat in einer XML-Datei abspeichert.

7.2 Schwachstellen

Schon während der Planung und Programmierung des Werkzeugs sind mir einige Schwachstellen aufgefallen, die teilweise in Kauf genommen wurden.

Der sprachunabhängige Ansatz zur Quelltextpositionsgewinnung ist eher unbefriedigend. Es

können nur Positionen für implementierte Methoden gewonnen werden. Weitere Informationen, die für die dazugehörige Klasse und ihre weiteren Mitglieder wie Variablen und Events werden aus den Methodeninformationen abgeleitet und werden somit nur geschätzt. Nachteilig ist, das Methoden über mehrere Dateien hinweg definiert werden können und somit die Positionsbestimmung für Klassen und ihre weiteren Mitglieder auf Basis der Methodeninformationen falsch sein können. Außerdem erhalten Objekte wie Schnittstellen oder abstrakte Klassen keine Quelltextpositionen, da sie keine Methoden implementieren.

Eine weitere Schwachstelle ist, das einige Methoden keinen CIL-Code enthalten aber von anderen Methoden aufgerufen werden. Diese Aufrufe können nicht weiterverfolgt werden, so dass Unterbrechungen im Aufrufgraph entstehen. Dazu zählt auch das Delegatenkonstrukt. Bei der Instanziierung eines Delegaten wird innerhalb des CIL-Codes ein Funktionszeiger für die dem Delegaten zugewiesene Methode erzeugt. Wird der Delgate aufgerufen wird, kann bei der implementierten CIL-Code-Analyse nicht festgestellt werden welche Methode der Delegat aufruft. Zur Abbildung solcher Aufrufe müsste eine vollständige virtuelle Maschine nachgebildet werden.

7.3 Weiterentwicklung

Mit dem nächsten Release den Visual Studio wird für die Sprachen Visual Basic.NET und C# ein neuer Compiler mitgeliefert, der die nächsten Erweiterungen der CLI und somit dem .NET-Framework liefert. Hierbei werden neue Sprachkonstrukte eingeführt die ggf. dazu führen, dass weitere Metadatentabellen eingeführt werden. Diese würde dann unter Umständen nicht bei der Analyse berücksichtigt und könnten später im Werkzeug nachträglich eingepflegt werden, damit auch Programme die mit C# in Version 3.0 geschrieben wurden über ihre Zwischenabbildung als RFG generiert werden können.

Eine weitere Möglichkeit zur Weiterentwicklung ist auch die Analyse von Assemblys, die aus mehreren Modulen bestehen. Diese zusammengelinkten Module können bisher nur mit den Einträgen der *ExportType*-Tabelle dargestellt werden. Es fehlen jedoch die inneren Definitionen wie Methoden und Variablen dazu. Das Assemblyformat könnte auf die Lage dieser Programmteile hin untersucht werden. Somit wäre es möglich größere Bibliotheken zu untersuchen, die aus mehreren Einzelteilen bestehen.

Das entstandene Werkzeug könnte auch dahin gehend erweitert werden, das die andere im Bauhaus verwendete Zwischendarstellung, die IML, aus den Assemblyinformationen gewonnen wird. Möglich wäre auch eine automatische Generierung für Softwaremetriken wie sie im Bauhausprojekt verwendet werden. Allerdings müssten für beide Erweiterungen eine bessere Lösung der Quelltextpostitionsgewinnung gefunden werden.

ANHANG A

File Header eines Portable Executable Images

Die folgenden Tabellen beschreibt den genauen Aufbau der *Header* in einem Portable Executable (siehe dazu auch [ECM06b] Partition II und [Mic06b]). Hierbei liegen nicht immer alle Felder hintereinander im Bytestrom, deswegen gibt es einmal das Feld Offset, das das Einstiegsbyte für jedes Feld markiert und das Feld Size, welches die Anzahl der Bytes für das Feld angibt. Fett markierte Felder sind für die Informationsgewinnung der Metadaten wichtig.

Header	Offset	Size	Field
DOS	0	2	Msdos_Sig
	2	2	Nlast_Page
	4	2	Npages
	6	54	Msdos_Header
	60	4	**Pe_Offset**
	64	64	Msdos_Header2
	Pe_Offset	4	PE Signature
COFF File Header	Pe_Offset + 4	2	Machine
	Pe_Offset + 6	2	**NumberOfSections**
	Pe_Offset + 8	4	TimeDateStamp
	Pe_Offset + 12	4	PointerToSymbolTable
	Pe_Offset + 16	4	NumberOfSymbols
	Pe_Offset + 20	2	SizeOfOptionalHeader
	Pe_Offset + 22	2	Charactericics
Optional Header Standard Fields	Pe_Offset + 24	2	Magic
	Pe_Offset + 26	1	MajorLinkerVersion
	Pe_Offset + 27	1	MinorLinkerVersion
	Pe_Offset + 28	4	SizeOfCode
	Pe_Offset + 32	4	SizeOfInitializedData
	Pe_Offset + 36	4	SizeOfUninitializedData
	Pe_Offset + 40	4	AdressOfEntryPoint

	Pe_Offset + 44	4	BaseOfCode
	Pe_Offset + 48	4	BaseOfCode
Optional Header Windows-Specific Fields	Pe_Offset + 52	4	ImageBase
	Pe_Offset + 56	4	SectionAlignment
	Pe_Offset + 60	4	FileAlignment
	Pe_Offset + 64	2	MajorOperationsystemVersion
	Pe_Offset + 66	2	MinorOperatinSystemVersion
	Pe_Offset + 68	2	MajorImageVersion
	Pe_Offset + 70	2	MinorImageVersion
	Pe_Offset + 72	2	MajorSubsystemVersion
	Pe_Offset + 74	2	MinorSubsystemVersion
	Pe_Offset + 76	4	Win32VersionValue
	Pe_Offset + 80	4	SizeOfImage
	Pe_Offset + 84	4	SizeOfHeaders
	Pe_Offset + 88	4	CheckSum
	Pe_Offset + 92	2	Subsystem
	Pe_Offset + 94	2	DLLCharacteristics
	Pe_Offset + 96	4	SizeOfStackReserve
	Pe_Offset + 100	4	sieOfStackCommit
	Pe_Offset + 104	4	SizeOfHeapReserve
	Pe_Offset + 108	4	SizeOfHeapCommit
	Pe_Offset + 112	4	LoaderFlags
	Pe_Offset + 116	4	NumberOfRvaAndSizes
Optional Header Data Directories	Pe_Offset + 120	4	Export Table RVA
	Pe_Offset + 124	4	Export Table Size
	Pe_Offset + 128	4	Import Table RVA
	Pe_Offset + 132	4	Import Table Size
	Pe_Offset + 136	4	Resource Table RVA
	Pe_Offset + 140	4	Resource Table Size
	Pe_Offset + 144	4	Exception Table RVA
	Pe_Offset + 148	4	Exception Table Size
	Pe_Offset + 152	4	Certification Table RVA
	Pe_Offset + 156	4	Certification Table Size
	Pe_Offset + 160	4	Base Relocation Table RVA
	Pe_Offset + 164	4	Base Relocation Table Size
	Pe_Offset + 168	4	Debug RVA
	Pe_Offset + 172	4	Debug Size
	Pe_Offset + 176	4	Architecture RVA
	Pe_Offset + 180	4	Architecture Size
	Pe_Offset + 184	4	Global Ptr RVA
	Pe_Offset + 188	4	Global Ptr Size

	Pe_Offset + 192	4	TLS Table RVA
	Pe_Offset + 196	4	TLS Table Size
	Pe_Offset + 200	4	Load Config Table RVA
	Pe_Offset + 204	4	Load Config Table Size
	Pe_Offset + 208	4	Bound Import RVA
	Pe_Offset + 212	4	Bound Import Size
	Pe_Offset + 216	4	IAT RVA
	Pe_Offset + 220	4	IAT Size
	Pe_Offset + 224	4	Delay Import Descriptor RVA
	Pe_Offset + 228	4	Delay Import Desccriptor Size
	Pe_Offset + 232	4	**CLI Header RVA**
	Pe_Offset + 236	4	**CLI Header Size**
	Pe_Offset + 240	4	Reserved RVA
	Pe_Offset + 244	4	Reserved Size

Im Anschluss an die Header folgen die Section Tables. Die Anzahl der Section Tables wurde im COFF Header ausgelesen. Alle Sections liegen im Bytestrom hintereinander und jede enthält folgende Felder:

Offset	Size	Name
x	8	Name
x+8	4	**VitualSize**
x+12	4	**VirtualAddress**
x+16	4	**SizeOfRawData**
x+20	4	**PointerToRawData**
x+24	4	PointerToRelocations
x+28	4	PointerToLinenumbers
x+32	2	NumberOfRelocations
x+34	2	NumberOfLinenumbers
x+36	4	Characteristics

Tabelle A.2: Felder eines Section Tables

ANHANG B

Metadatentabellen

Die Tabelle B.1 zeigt die bisher genutzten Metadatentabellen nach [ECM06b] mit ihrer zugehörigen Nummer.

Nummer	**Name**
0	Module
1	Typeref
2	Typedef
3	Fieldptr
4	Field
5	Methodptr
6	Methoddef
7	Paramptr
8	Paramdef
9	Interfaceimpl
10	Memberref
11	Constant
12	Customattribute
13	Fieldmarshal
14	Declsecurity
15	Classlayout
16	Fieldlayout
17	Standalonesig
18	Eventmap
19	Eventptr
20	Event
21	Propertymap
22	Propertyptr
23	Property
24	Methodsemantic
25	Methodimpl
26	Moduleref
27	Typespec

28	Implmap
29	Fieldrva
30	Enclog
31	Encmap
32	Assembly
33	Assemblyprocessor
34	Assemblyos
35	Assemblyref
36	Assemblyrefprocessor
37	Assemblyrefos
38	File
39	Exportedtype
40	Manifestresource
41	Nestedclass
42	Genericparam
43	Methodspec
44	Genericparamconstraint

Tabelle B.1: Metadatentabellen, die im [ECM06b] tatsächlich genutzt werden

ANHANG C

DTD zur XML-Datei zum Speichern von Quelltextinformationen

Das Listing C.1 spezifiziert die Informationen, die das Programm zur Analyse von Quelltextpositionen in der XML-Datei ausgibt.

```
1
2 <!ELEMENT symbols (files,objects)> <!-- root node -->
3 <!ELEMENT files (file*)> <!-- collects all filereferences -->
4 <!ELEMENT file EMPTY> <!-- present one filereference -->
5 <!ATTLIST file id CDATA #REQUIRED
6     name CDATA #REQUIRED
7           fname CDATA #REQUIRED
8           path CDATA #REQUIRED>
9 <!ELEMENT objects (item*)> <!-- collects all objects -->
10 <!ELEMENT item (cil*)> <!-- collects infos for class, field, method .. -->
11 <!ATTLIST item fullname CDATA #REQUIRED
12     name CDATA #REQUIRED
13     start_row CDATA #REQUIRED
14     start_column CDATA #REQUIRED
15     file_ref CDATA #REQUIRED
16           table CDATA #REQUIRED
17           row CDATA #REQUIRED>
18 <!ELEMENT cil EMPTY> <!-- collects info for methodinstructions -->
19 <!ATTLIST cil offset CDATA #REQUIRED
20     start_row CDATA #REQUIRED
21     start_column CDATA #REQUIRED
22     end_row CDATA #REQUIRED
23     end_column CDATA #REQUIRED>
```

Listing C.1: DTD zum pdb2xml-Tool

ABBILDUNGSVERZEICHNIS

2.1 CLI-Konzept vom Quelltext bis zur Ausführung eines Programms 10

2.2 Beziehung zwischen Bibliotheken und Profilen in der CLI ([ECM06b], Partition IV, S.3) 11

2.3 Der Global Assembly Cache (GAC) 13

2.4 Der .NET-Framework Stack 15

2.5 Aufbau des .NET-Framework 3.0 ([Mic06c]) 16

2.6 Kommunikation zwischen *Managed-* und *Unmanaged-Code* im .NET-Framework (aus [ZG08]) 31

2.7 Das RFG-Modell für C++ [Axi06] 34

3.1 Das RFG-Schema für Java 40

3.2 Ein Klassenicon mit visualisiertem Sichtbarkeitsattribut. 43

3.3 Teilschema zur Eventmodellierung 49

3.4 Icon für generische Schnittstellen 50

3.5 Teilschema für Generics 51

3.6 Das RFG-Schema für C# 52

4.1 Aufgabenbereich der Analyse 55

4.2 *Portable Executable Image* im Speicher und im Dateiformat ([Pie02]) 57

4.3 Aufbau eines *Portable Executable Files* 58

4.4 Der Aufbau des PE-Headers 59

4.5 RVA zu *Offset* Abbildung von *Sections* (angelehnt an [Pie02]) 60

4.6 Metadatenstruktur 64

4.7 Teile eines #String-Streams angezeigt in einem Hexadezimal-Editor 65

4.8 *Blob*-Signatur für die *Field*-Tabelle ([ECM06b] Partition II S.157) 66

4.9 Darstellung einer *Field*-Signatur im #Blob-Stream 67

4.10 Ausschnitt aus der Dekodiertabelle für das *Flag*-Feld der *TypeDef*-Tabelle (Tabelle 23.1.15 aus [ECM06b] Partition II S. 143 ff.) 68

4.11 Zuordnung der Klassenvariablen 71
4.12 Zuordnung von Events 76
4.13 Zuordnung von Propertys 77
4.14 Aufgabe der *MethodSemantic*-Tabelle 78
4.15 Übersicht der relevanten Verweise innerhalb der Tabellen 82
4.16 Der CIL Bytecode 83
4.17 Ausschnitt des CIL Bytecode für die Ereignisbehandlung 85
4.18 Debuggerkonzept des Visual Studios 87

5.1 Eine automatisch erzeugte Environment-View 92

TABELLENVERZEICHNIS

2.1 Übersicht Zugriffsmodifizierer [Mic08a] 21

2.2 Überblick über die Unterschiede von Klassen und *Structs* 23

2.3 Überblick über die Unterschiede von Indexern und Propertys 31

4.3 Header eines Streams 62

4.5 Metadatentabellen 63

4.7 Bitvektor zur Dekodierung der Metadatentabellen 63

4.10 Abbildung der CLI-Sichtbarkeiten auf Sichtbarkeiten im RFG 70

4.27 CIL Anweisungen die Klassenvariablen manipulieren 84

6.2 Testergebnisse des entwickelten Werkzeugs mit übersetzten C#-Quelltextbeispielen zu Visual Studio 2003 und 2005 97

6.3 Testergebnisse des entwickelten Werkzeugs mit Open Source Programmen . . 97

A.2 Felder eines Section Tables 103

B.1 Metadatentabellen, die im [ECM06b] tatsächlich genutzt werden 106

LISTINGS

2.1 Syntaxbeispiel C# 20

2.2 Quelltext des Assembly *MyInternalAssembly* 22

2.3 Quelltext des Assembly *MyAssembly* welches das Assembly *MyInternalAssembly* verwendet. 22

2.4 Polymorphismus in C# 24

2.5 Ein Singlecast Delegate 25

2.6 Ereignisdefinition und -behandlung 27

2.7 Beispiel: Generics 28

2.8 Propertys in C# 29

2.9 Indexer in C# 30

2.10 Beispiel für das Benutzen von *Unmanaged Code* 32

C.1 DTD zum pdb2xml-Tool 107

LITERATURVERZEICHNIS

[And00] Anderson, Rick: *The End of DLL Hell.* Technischer Bericht, Microsoft Cooperation, Januar 2000.

[Axi06] Axivion: *Language Guide.* Axivion GmbH, 5.3 Auflage, 2006.

[ECM06a] ECMA: *Standard ECMA-334 - C# Language Specification.* `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, Juni 2006. Stand: Juli 2007.

[ECM06b] ECMA: *Standard ECMA-335 - Common Language Infrastructure (CLI).* `http://www.ecma-international.org/publications/standards/Ecma-335.htm`, Juni 2006. Stand: Juli 2007.

[Fli06] Flick, Steffen: *Das Dagstuhl Middle Metamodel im Kontext sprachunabhängigen Refactorings.* Diplomarbeit, Universität Koblenz, August 2006.

[Gun00] Gunnerson, Eric: *C#.* Galileo OpenBook. Galileo Computing, 1 Auflage, September 2000.

[Had06] Hader, Jan: *Generierung und Erweiterung des Resource Flow Graph für Visual Basic 6 Programme.* Diplomarbeit, Universität Bremen, Dezember 2006.

[ISO06] ISO/IEC: *ISO/IEC 23271:2006 - Information Technology - Common Language Infrastructure (CLI).* `http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`, Oktober 2006.

[JCD02] Jin, Dean, James R. Cordy und Thomas R. Dean: *Where's the Schema? A Taxonomy of Patterns for Software Exchange.* In: *WPC 2002 - IEEE 10th International Workshop on Program Comprehension*, Seiten 65–74, Juni 2002.

[Jor06] Jordan, Robert: *[Mono-list] VBNC in Mono.* `http://lists.ximian.com/pipermail/mono-list/2006-September/032692.html`, September 2006. Ausschnitt aus einer Mailingliste.

[KEPG99] Koschke, Rainer, Thomas Eisenbarth, Erhard Plöderer und Jean Francois Girard: *Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnugn von SW-Architekturen.* In: *Workshop Software-Reengineering*,

Band 7-99 der Reihe *Fachberichte Informatik*, Seiten 17–26, Universität Koblenz-Landau, 1999.

[LTP04] Lethbridge, Timothy, Sander Tichelaar und Erhard Plödereder: *The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering.* Electronic Notes in Theoretical Computer Science, 94:7–18, 2004.

[Mica] Microsoft: *The COFF type debug information no longer contains line numbers in Visual C++.* `http://support.microsoft.com/kb/259924`.

[Micb] Microsoft: *An Overview of Managed/Unmanaged Code Interoperability.* `http://msdn2.microsoft.com/en-us/library/ms973872.aspx`. Stand: 17.01.2008.

[Micc] Microsoft: *Security in Native and .NET Framework Code.* `http://msdn2.microsoft.com/en-gb/library/1787tk12(VS.80).aspx`. Stand: 17.01.2008.

[Mic02] Microsoft: *Microsoft Launches XML Web Services Revolution With Visual Studio .NET and .NET Framework.* `http://www.microsoft.com/presspass/press/2002/feb02/02-13RevolutionPR.mspx`, Februar 2002. Abruf: 10.10.2007.

[Mic05a] Microsoft: *C# Programming Guide.* `http://msdn2.microsoft.com/en-us/library/67ef8sbd(VS.80).aspx`, 2005. Stand: 12.12.2007.

[Mic05b] Microsoft: *C# Reference.* `http://msdn2.microsoft.com/en-us/library/618ayhy6(VS.80).aspx`, 2005. Stand 12.12.2007.

[Mic06a] Microsoft: *Microsoft .NET Framework 3.0 Deployment Guide.* `http://msdn2.microsoft.com/en-us/library/aa480173.aspx`, Juli 2006. Stand: 8.11.2007.

[Mic06b] Microsoft: *Microsoft Portable Executable and Common Object File Format Specification.* `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, Mai 2006.

[Mic06c] Microsoft: *.NET Framework 3.0 Versioning and Deployment Q&A.* `http://msdn2.microsoft.com/en-us/netframework/aa663314.aspx`, 2006. Stand: 15.12.2007.

[Mic08a] Microsoft: *Accessibility Levels (C# Reference).* `http://msdn2.microsoft.com/en-us/library/ba0a1yw2(VS.80).aspx`, 2008. Stand: 17.01.2008.

[Mic08b] Microsoft: *Generics in the Run Time (C# Programming Guide).* http://msdn2.microsoft.com/en-us/library/f4a6ta2h.aspx, 2008. Abruf: 08.02.2008.

[Mic08c] Microsoft: *Versioning with the Override and New Keywords (C# Programming Guide).* http://msdn2.microsoft.com/en-gb/library/6fawty39.aspx, 2008.

[Pie00] Pietrek, Matt: *Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework.* MSDN Magazine, Oktober 2000.

[Pie02] Pietrek, Matt: *An In-Depth Look into the Win32 Portable Executable File Format.* http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx, Februar 2002. Stand: 18.1.2008.

[PLT+02] Plödereder, Erhard, Timothy C. Lethbridge, Sander Tichelaar, Claudio Riva, Panos Linos und Sergei Marchenko: *The Dagstuhl Middle Model (DMM).* http://www.site.uottawa.ca/~tcl/dmm/DMMDescriptionV0006.pdf, Juli 2002.

[Ric06] Richter, Jeffrey: *Microsoft .NET Framework-Programmierung in C#.* Microsoft Press Deutschland, 2 Auflage, 2006.

[RVP] Raza, Aoun, Gunter Vogel und Erhard Plödereder: *Bauhaus - A Tool Suite for Program Analysis and Reverse Engineering.* http://www.iste.uni-stuttgart.de/ps/bauhaus/papers/bauhaus.pdf. Stand: 8.11.2007.

[SCG00] SCG, Software Composition Group: *SCG - FAMOOS - FAMIX (Information Exchange Model).* http://iamwww.unibe.ch/~famoos/FAMIX/, Januar 2000. Stand: 27.12.2007.

[SCG02] SCG, Software Composition Group: *FAMOOS Archive.* http://www.iam.unibe.ch/~famoos/, März 2002. Stand: 27.12.2007.

[SCG07] SCG, Software Composition Group: *MooseWiki – Famix 3.0.* http://smallwiki.unibe.ch/moose/famix3.0/, November 2007. Stand: 12.01.2008.

[Sch07] Schwichtenberg, Holger: *Markübersicht: .Net-Programmiersprachen - Babylonische Vielfalt.* IX, (10):102–108, Oktober 2007.

[Ske04] Skeet, Jon: *C# Frequently Asked Questions - What's the difference between override and new?* http://blogs.msdn.com/csharpfaq/archive/2004/03/12/88453.aspx, März 2004.

[Sta05] Stall, Mike: *Converting a managed PDB into a XML file.* `http://blogs.msdn.com/jmstall/archive/2005/08/25/pdb2xml.aspx`, August 2005.

[Tic01] Tichelaar, Sander: *Modeling Object-Oriented Software for Reverse Engineering and Refactoring.* Doktorarbeit, Universität Bern, Dezember 2001.

[ZG08] Zhang, Yi und Xiaoying Guo: *CLR Indside Out: Marshaling between Managed and Unmanaged Code.* `http://msdn.microsoft.com/msdnmag/issues/08/01/CLRInsideOut/default.aspx`, Januar 2008.