JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Analyse von C#-Quellcode

BAKKALAUREATSARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bakkalaureus/Bakkalaurea der technischen Wissenschaften

in der Studienrichtung

INFORMATIK

Eingereicht von:
Thomas Thrainer, 0356015

Angefertigt am:
Institut für Systemsoftware

Betreuung:
Prof. Dr. H. Mössenböck
Dipl.-Ing. Markus Löberbauer

Linz, 06 2006

Inhaltsverzeichnis

1 Pflichtenheft

1.1 Allgemeines

Im Rahmen eines Projektes mit der Shared Source Implementierung der Common Language Infrastructure (CLI) (Codename Rotor) [www.microsoft.com,1] [ECMA,1] wurde der Compilergenerator Coco/R [ssw.jku.at,1] erweitert. Durch diese Erweiterungen kann er nun nicht mehr nur LL(1)-Grammatiken verwenden, sondern auch Schritte zur Auflösung von LL(1)-Konflikten setzen.

Für diese Version von Coco/R wurde eine attributierte Grammatik (ATG) für die Programmiersprache C# [ECMA,2] entwickelt, aus der Coco/R einen vollständigen Top-Down-Parser für C#-Programme erzeugt. Durch Einfügen zusätzlicher semantischer Aktionen in diese Vorlage können beliebige Analyse- und Compilerwerkzeuge für C#-Quellcode erzeugt werden.

1.2 Ziel

Das Ziel dieser Arbeit ist es, die C#-Grammatik-Vorlage so zu attributieren und mit semantischen Aktionen zu versehen, dass der daraus generierte Parser aus C#-Programmen verschiedenste Quellcode-Metriken berechnet. Dazu ist die C#-Version von Coco/R zu verwenden. Die errechneten Metriken sollen ansprechend dargestellt werden, wozu C# zu verwenden ist.

1.3 Zu berechnende Kennzahlen

- Lines of Code (mit/ohne Leerzeilen, Kommentare)
- Chars of Code (mit/ohne Leerzeilen, Kommentare)
- Verhältnis zwischen Kommentarzeilen und Codezeilen und Kommentarzeichen und Codezeichen
- Anzahl der Anweisungen
- Anzahl der Schleifen geordnet nach Typ
- Anzahl der Verzweigungen
- Anzahl der Namespaces, Klassen, Methoden, Felder, Properties
- Durchschnittliche und maximale Anzahl der Parameter von Methoden
- Durchschnittliche und maximale Anzahl der lokalen Variablen von Methoden
- Anzahl der Methodenaufrufe, die in einer Methode stattfinden
- Anzahl, wie oft eine Methode aufgerufen wird
- Anzahl des Auftretens der verschiedenen Schlüsselworte und Operatoren
- Durchschnittliche und maximale Schachtelungstiefe von Anweisungsblöcken
- DIT (depth of inheritance tree) Durchschnittliche und maximale Vererbungstiefe von Klassen
- NOC (number of children) Durchschnittliche und maximale Anzahl von Kindern von Klassen
- Komplexitätsmaße nach
  - McCabe
  - Rechenberg
  - Halstead
  - Henry & Kafura

Diese Kennzahlen sind jeweils für Methoden, Klassen, Namespaces und für das ganze Programm zu berechnen, je nachdem, was für die jeweiligen Maße angebracht ist.

1.4 Ausgabe

Die Ausgabe soll grafisch erfolgen. Dazu soll mittels C# eine Oberfläche erstellt werden, auf der man die verschiedenen Kennzahlen betrachten kann. Es soll möglich sein, die verschiedenen Bereiche Namespace, Klasse oder Methode auszuwählen, um die Kennzahlen für diese Einheit zu betrachten.

Die Daten sollen sich exportieren lassen, wobei ein Dateiformat zu wählen ist, das von einem Menschen intuitiv gelesen werden kann. Die Ergebnisse der Analyse sollen ebenfalls direkt als Kommentar in den Quellcode exportierbar sein. Dabei soll man auswählen können, ob die Statistik auf Dateiebene, auf Klassenebene und/oder auf Methodenebene zu exportieren ist. Zuvor exportierte Statistiken sollen zuerst entfernt werden, um redundante Informationen im Quellcode zu vermeiden.

2 Berechnung der Kennzahlen

Dieser Teil beschäftigt sich mit der Berechnung der geforderten Kennzahlen und mit der anschließender Speicherung in einer geeigneten Datenstruktur. Die grafische Ausgabe und Auswertung wird im Kapitel 3 auf Seite 30 präsentiert.

2.1 Analyse und Planung

2.1.1 Analyse der Kennzahlen

In den folgenden Abschnitten werden alle zu berechneten Kennzahlen vorgestellt, analysiert und Überlegungen zu deren Implementierung angestellt. Bei Unklarheiten beziehungsweise Entscheidungsfreiheiten einiger Kennzahlen bezüglich derer Relevanz oder Anwendung auf C#, werden die getroffenen Entscheidungen angegeben und begründet.

Lines of Code / Chars of Code

Die Berechnung der Lines of Code wird durch Zählen der End-of-Line-Marker realisiert. Alle Zeilen, Codezeilen, Kommentarzeilen und Leerzeilen müssen separat gezählt werden. Diese Zählung erfolgt im Scanner, da man in diesem Teil Zugriff auf die Kommentare und End-of-Line Marker hat.

Die Chars of Code werden ebenfalls im Scanner implementiert. Dieser analysiert jedes Zeichen der Eingabedatei und ordnet es der richtigen Kategorie (Code-Zeichen, Leerzeichen, Kommentar-Zeichen) zu.

Es wird gefordert, die Kennzahlen für Klassen, Methoden und das gesamte Programm gesondert zu berechnen. Daher ist es nötig, eine Kommunikation zwischen dem

Scanner und dem Parser herzustellen, da der Parser dem Scanner mitteilen muss, wann eine neue Klasse/Methode beginnt. Auf die Frage wie die Kommunikation zwischen Parser und Scanner funktioniert wird bei der Implementierung im Abschnitt 2.2.1 auf Seite 25 eingegangen.

Man könnte versuchen, die verschiedenen Namespaces, Klassen oder Methoden im Scanner zu identifizieren (z.B. durch Überprüfen auf die Schlüsselwörter *namespace* und *class*). Spätestens aber beim Identifizieren von Methoden (mit Spezialfällen wie Konstruktoren oder Properties) würde diese Methode kompliziert werden. Außerdem ist der Scanner nicht dafür zuständig, syntaktische oder semantische Analysen durchzuführen. Für diese Analysen ist der Parser vorgesehen.

Andererseits wäre es möglich, alle Zählungen im Parser durchzuführen. Dafür müsste man Pragmas für End-of-Line Marker, Kommentare, White-Spaces, und ähnliches definieren und diese im Parser auswerten. Codezeilen und -Zeichen würden durch Auswerten der *line* und *pos* Felder der gelesenen Token berechnet werden. Dies würde eine Attributierung aller Token in der Gramatik erfordern, weshalb von dieser Lösung abgesehen wurde.

Zählregeln Die Zuordnung von Zeilen und Zeichen zu den Kategorien Code, Kommentar und Whitespace und zu den Bereichen Datei, Namespace, Klasse und Methode ist nicht in allen Fällen selbstverständlich. Wir erklären daher hier die genauen Zählregeln, die verwendet werden:

- **Whitespace-Zeichen** sind Leerzeichen, Tabulatoren, vertikale Tabulatoren und Form-Feeds außerhalb von Kommentaren.
- **Whitespace-Zeilen** sind Zeilen, in denen nur Whitespace-Zeichen enthalten sind.
- **Kommentar-Zeichen** sind alle Zeichen zwischen /$\star$ und $\star$/ und zwischen // und dem Ende der Zeile, einschließlich den Zeichen in /$\star$, $\star$/ und //.
- **Kommentar-Zeilen** sind Zeilen, in denen nur Kommentar-Zeichen und Whitespace-Zeichen enthalten sind.
- **Code-Zeichen** sind die Zeichen aus denen Token bestehen. Pragmas werden ebenfalls als Token und daher als Code-Zeichen gezählt.

- **Code-Zeilen** sind Zeilen, in denen mindestens ein Code-Zeichen enthalten ist. Sie werden zu dem Bereich gezählt, in dem das erste Token der Zeile enthalten ist.
- **Bereichsänderungen** treten direkt nach dem Namen des neuen Bereiches in Kraft. In "`void main() {}`" wird "`void main`" noch zu dem umgebenden Klassen-Bereich gezählt, "`() {}`" jedoch schon zu dem Methoden-Bereich. Die ganze Zeile wird dem Klassen-Bereich zugeordnet, da der erste Token der Zeile noch zu diesem Bereich gehört. Die `main` Methode besteht demnach aus null Zeilen.

Verhältnis Code-Zeilen/Kommentar-Zeilen, Code-Zeichen/Kommentar-Zeichen

Dies ist eine Erweiterung zu den Lines of Code, wobei die nach dem oben erklärten Schema berechneten Kennzahlen verwendet werden, um die gewünschten Verhältnisse zu berechnen.

Anzahl der Anweisungen

Laut C#-Standard [ECMA,2] existieren zwei Typen von Statements: *statement* und *embedded-statement*. Diese Unterscheidung wurde getroffen, um bestimmte Statements (wie z.B. Deklarationen) in Rümpfen von *if*-, *for*-, und anderen Kontrollstrukturen zu verbieten.

Diese zwei Typen sind als Produktionen in der Grammatik enthalten, weshalb die Zählung durch Attributierung dieser Stellen realisiert wird.

Anzahl der Schleifen geordnet nach Typ

Diese Kennzahl ist als Zählung der Schlüsselwörter *for*, *foreach*, *while* und *do* implementiert. Da eine *do-while*-Schleife das Schlüsselwort *while* enthält, ist die Anzahl der *while*-Schleifen die Anzahl der *while*-Schlüsselwörter weniger die Anzahl der *do*-Schlüsselwörter. Die Anzahl der *do-while*-Schleifen ist die Anzahl der *do*-Schlüsselwörter.

Anzahl der Verzweigungen

Diese Kennzahl ist die Anzahl der *if* und *switch* Schlüsselwörter in einem Programm. Dabei wird eine *else if*-Anweisung als eine Verzweigung gezählt, da sie eine 2. Verzweigung nach einer schon getroffenen Verzweigung darstellt. Die *switch*-Anweisung stellt jedoch nur eine Verzweigung dar, da sie zwar zwischen mehreren Möglichkeiten wählt, jedoch zur ausgewählten sofort springt.

Anzahl der Namespaces, Klassen, Methoden und Felder

Hierbei wird das Auftreten der durch den Parser identifizierten Bereiche Namespace, Klasse, Methode und der deklarierten Datenfelder gezählt. Dabei ist darauf zu achten, dass es möglich ist, mehrere Klassen mit dem gleichen Namen in unterschiedlichen Namespaces zu deklarieren. Gleiches gilt für Methoden, Felder und Klassen, wobei diese Elemente aber eigene Einträge darstellen und als solche gezählt werden müssen.

Die zum Zählen und Speichern der Kennzahlen verwendeten Datenstrukturen, die im Abschnitt 2.1.2 auf Seite 21 vorgestellt werden, stellen eine korrekte Zählung sicher.

Durchschnittliche und Maximale Anzahl der Parameter von Methoden

Es muss die Anzahl der Parameter jeder Methode gezählt und am Ende eines Analysedurchlaufes die durchschnittliche Parameteranzahl berechnet werden. Die Maximalanzahl wird ebenfalls am Ende des Analysedurchlaufes ermittelt.

Durchschnittliche und Maximale Anzahl der lokalen Variablen von Methoden

Zur Berechnung dieser Kennzahl müssen die Deklarationen von lokalen Variablen erkannt und gezählt werden. Diese Deklarationen sind in der Grammatik in der Produktion *LocalVariableDeclaration* erfasst und dort attributiert. Man kann in einer Methode mehrere Male eine Variable mit dem gleichen Namen deklarieren. Dies kann entweder durch Verdecken einer bereits deklarierten Variable in eingebetteten Blöcken oder durch mehrmalige Deklarieren von Variablen gleichen Namens in unterschiedlichen Geltungsbereichen geschehen. Letzteres ist oft bei Laufvariablen in Schleifen der Fall. Folgender Codeausschnitt zeigt typische Beispiele:


```
...
int i = 0;
if (...) {
  int i = 2; // Verdeckung
}
for (int j=0; j<10; ++j) ... ;
for (int j=0; j<5; ++j) ... ; // Erneute Definition von j
...
```

In solchen Fällen werden alle Deklarationen gezählt.

Die durchschnittliche und maximale Anzahl der lokalen Variablen lassen sich in analoger Art und Weise wie die durchschnittliche und maximale Anzahl der Parameter am Ende eines Analysedurchlaufes berechnen.

Anzahl der Methodenaufrufe einer Methode

Die Methodenaufrufe, die eine Methode durchführt, werden in der Grammatik in der Produktion *Primary* behandelt. Dort werden Ausdrücke, die am Ende eine öffnende Klammer, optional Argumente und eine schließende Klammer haben, als Methodenaufrufe identifiziert.

Mit C# ist es möglich, Operatoren zu überladen und Properties zu definieren. Daher können nicht alle Methodenaufrufe durch diese Methode erkannt werden. Dazu wären umfassende Typinformationen nötig, die aber den Rahmen dieser Arbeit gesprengt hätten.

Anzahl, wie oft eine Methode aufgerufen wird

Es ist nötig, für alle Methodenaufrufe, die nach oben erläuterter Methode identifiziert wurden, einen der aufgerufenen Methode zugeordneten Zähler zu erhöhen. Dazu werden am Ende des Analysedurchganges alle Aufrufe analysiert und versucht, die aufgerufenen Methoden zu finden. Mehrere Methoden mit gleichem Namen können in unterschiedlichen Klassen/Namespaces existieren, wobei es mehrere Möglichkeiten gibt, diese Methoden aufzurufen. Auf Grund dieser Mehrdeutigkeiten ist das Auffinden dieser Namen nicht trivial und ist wie im Abschnitt 2.2.2 auf Seite 26 erläutert implementiert.

Es ist möglich, dass es mehrere gleich heißende Methoden in einer Klasse gibt, die jedoch in ihrer Signatur unterschiedlich sind. Diese überladenen Methoden können nicht richtig erkannt werden, da ohne Typinformationen eine Überprüfung der Argumenttypen nicht möglich ist. So ist es möglich, den Typ *String* mit *System.String* anzusprechen. Umfassende Typinformation würden benötigt um zu erkennen, dass es sich bei diesen Namen um den gleichen Typ handelt. Es ist des Weiteren auch möglich, dass Typen automatisch konvertiert werden, z.B. wenn einem int-Parameter eine Variable des Typs short übergeben wird. Und schließlich kann bei virtuellen Methoden zur Compilierzeit nicht vorhergesagt werden, welche Methode tatsächlich aufgerufen wird.

Eine akzeptable Abschätzung trotz fehlender Typinformationen wird erreicht, indem die Anzahl der Parameter des Aufrufs mit der Anzahl der Parameter der Methodendefinitionen verglichen wird, um überladenen Methoden zu unterscheiden. Daher wird diese Zählung nur eine Abschätzung für die Menge der Aufrufe einer Methode geben, aber keineswegs exakt sein.

Anzahl des Auftretens der verschiedenen Schlüsselwörter und Operatoren

Diese Kennzahl ist als Zählung der Vorkommnisse der verschiedenen Schlüsselwörter und Operatoren implementiert. Eine Schwierigkeit ist die große Anzahl dieser Operatoren mit der damit verbundenen Organisation der Daten. Dabei ist es von Vorteil, eine Map zu verwenden, in der die Vorkommnisse gezählt werden, anstatt für jeden Operator eine eigene Variable zu verwenden.

Durchschnittliche und Maximale Schachtelungstiefe von Methoden

Eingebettete Statements, die eine neue Verschachtelungsstufe im Code darstellen, werden in der Grammatik durch die Produktion *EmbeddedStatement* ausgedrückt. Der Anfang eines eingebetteten Blockes wird daher dort erkannt und attributiert.

Am Anfang einer Methode wird ein Zähler initialisiert, der beim jedem Betreten eines neuen eingebetteten Blockes erhöht wird. Beim Verlassen eines solchen Blockes wird dieser Zähler wieder verringert, wobei der Vergleich mit der maximalen Verschachtelungstiefe durchgeführt wird. Nach dem Ende eines Analysedurchganges wird aus den maximalen Verschachtelungstiefen aller Methoden die durchschnittliche und maximale

Verschachtelungstiefe für Klassen, Namespaces und das ganze Programm berechnet.

DIT - Depth of inheritance Tree

Für die Berechnung der DIT wird für jede Klasse ihre Vererbungstiefe mitgeführt. Wird sie nicht abgeleitet, ist ihre Vererbungstiefe null. Sonst wird die Vererbungstiefe der Vater-Klasse ermittelt, und die Vererbungstiefe der aktuell analysierten Klasse um eins höher als diese ermittelte Tiefe gesetzt. Nach dem Ende des Analysedurchganges wird die durchschnittliche und maximale Vererbungstiefe aus den Vererbungstiefen aller Klassen berechnet.

Es sind keine Informationen über Klassen vorhanden, die außerhalb der zu analysierenden Datei definiert sind (dazu gehören auch Klassen in der Klassenbibliothek). Deshalb kann bei Klassen, die von solchen externen Klassen ableiten, die Vererbungstiefe nicht berechnet werden.

NOC - Number of children

Beim Deklarieren einer neuen Klasse wird für die Vater-Klasse ein Zähler erhöht, der die Anzahl der Kinder angibt. Wie bei der Berechnung der DIT gilt, dass diese Zählung nicht exakt ist, wenn es Definition von Kind-Klassen außerhalb der analysierten Datei gibt.

Bei beiden Kennzahlen wird darauf geachtet, immer nach dem voll deklariertem Namen (mit Namespacenamen) zu suchen, da es möglich ist, mehrere Klassen gleichen Namens in verschiedenen Namespaces zu deklarieren. Ähnlich wie beim Aufruf von Methoden gibt es mehrere Möglichkeiten, eine bestimmte Klasse anzusprechen (durch Aliases oder Usings). Im Abschnitt 2.2.2 auf Seite 26 wird auf die Problematik des Suchens von Namen eingegangen.

Komplexitätsmaß nach McCabe

Das Komplexitätsmaß nach McCabe wird vom Kontrollfluss-Graphen eines Programms abgeleitet und gibt die Anzahl der linear unabhängigen Kontrollpfade durch ein Programm an. Dieses Maß lässt sich nur auf Methoden anwenden, da es in Klassen und

Namespaces keine Kontrollflüsse gibt. Die Berechnung beruht auf folgender Formel:

$$M = e - n + 2$$

mit

e ... Kanten im Graph

n ... Knoten im Graph

M ... Anzahl der möglichen Wege durch eine Methode

Die Anzahl der Kanten werden durch das Zählen der *if*-Schlüsselwörter$*2$ errechnet, da bei einem if 2 Kanten hinzukommen (then- und else-Teil). Weiters durch Zählen der *case*-Schlüsselwörter, da bei jedem *case* eine neue Kante hinzukommt. Das *case*-Schlüsselwort kommt auch in *goto*-Statements vor, wird aber von uns nur in *switch* Blöcken als Sprung-Label berücksichtigt.

Bei jedem *switch*-Schlüsselwort wird wegen des *default*-Zweiges (zusätzlich zu den bereits behandelten case's) eine neue Kante hinzugefügt. Schlussendlich werden Schleifen gleich behandeln wie *if*'s, da bei jedem *for*, *foreach* oder *while* entschieden wird, ob in die Schleife verzweigt wird, oder nicht. *do*-Schlüsselwörter müssen nicht extra behandelt werden, da *do-while* Schleifen ein *while*-Schlüsselwort enthalten und somit bereits mitgezählt werden. Am Anfang ist immer eine Kante vorhanden, somit kommt man zu folgender Formel:

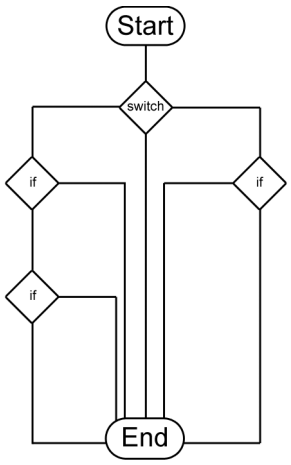
$$\begin{aligned} e = {} & 1 + 2 * \#\{\text{if}\} + \#\{\text{case}\} + \#\{\text{switch}\} + \\ & 2 * \#\{\text{for}\} + 2 * \#\{\text{foreach}\} + 2 * \#\{\text{while}\} \end{aligned}$$

Die Anzahl der Knoten im Graph errechnet sich durch die Anzahl der Verzweigungen, also die Anzahl der *if*-Schlüsselwörter, der *for*-, *foreach*- und *while*-Schlüsselwörter plus die Anzahl der *switch*-Schlüsselwörter. Darüber hinaus gibt es in jeder Methode einen Initial- und einen End-Knoten. Die Formel lautet also:

$$\begin{aligned} n = {} & 2 + \#\{\text{if}\} + \#\{\text{switch}\} + \\ & \#\{\text{for}\} + \#\{\text{foreach}\} + \#\{\text{while}\} \end{aligned}$$

Beispiel:

```
switch(i) {
 case 1:
   if(x) {
     if(y) ...
     else ...
   } else ...
   break;
 case 2: break;
 default:
   if(z) ...
   else ...
}
```



$$
\begin{aligned}
e &= 1 + 2 * 3 + 2 + 1 &&= 10 \\
n &= 2 + 3 + 1 &&= 6 \\
M &= e - n + 2 \\
&= 10 - 6 + 2 &&= 6
\end{aligned}
$$

Abbildung 2.1: Kontrollflußgraph

Die gesamte Berechnung wird auf das Zählen von *if*-, *for*-, *foreach*-, *while*-, *case*- und *switch*-Schlüsselwörtern zurückgeführt, wobei das bereits im Zuge der Berechnung anderer Komplexitätsmaße geschieht.
[www.dcs.qmw.ac.uk,1], [www.dacs.dtic.mil,1], [de.wikipedia.org,1], [HOL02]

Komplexitätsmaß nach Rechenberg

Das Komplexitätsmaß nach Rechenberg setzt sich aus 3 Komponenten zusammen:

$$\text{Gesamtkomplexität } CC = SC + EC + DC$$

Diese Komponenten sind wie folgt definiert:

SC (statement complexity) ist die Summe aller Anweisungen, die mit Erfahrungswerten gewichtet sind. Außerdem geht die Verschachtelungstiefe mit einem exponentiell steigenden Faktor mit ein. Die Gewichte wurden von Rechenberg wie folgt definiert, sind aber Erfahrungswerte und demnach anpassbar:

Anweisungsart	Gewichtung
Wertzuweisung	1
Prozeduraufruf	$1 +$ Parameterzahl
If-Anweisung	2 für if +
	1 für else
Select-Anweisung	3
While- und Do-Anweisung	3
For-Anweisung	3
Return-Anweisung	1
Goto-Anweisung	5
Break-Anweisung*	4
Continue-Anweisung*	3
Throw-Anweisung*	2
Try-Anweisung*	1

Rechenberg hat keine Werte für z.B. Break- und Continue-Anweisung angegeben, daher wurde diese Tabelle mit sinnvollen Werten ergänzt. Diese sind mit einem * gekennzeichnet. Andere von Rechenberg definierte Werte, wie für Loop- und With-Anweisungen, existieren in C# nicht und wurden deshalb nicht angeführt.

Der Faktor für die Verschachtelungstiefe ist $1.5^{\text{Verschachtelungstiefe}}$. Die Berechnung der statement complexity stellt keine großen Probleme dar, da die genannten Anweisungen in der Grammatik erkannt werden und mit den jeweiligen Gewichten multipliziert zur SC addiert werden.

Die Berechnung der Verschachtelungtiefe wurde im Abschnitt 2.1.1 auf Seite 12 erläutert.

EC (expression complexity) ist, ähnlich wie die SC, eine Summe aller Operatoren, die mit Erfahrungswerten gewichtet sind und mit einem Faktor für die Verschachtelungstiefe multipliziert werden. Die folgenden Werte sind wie bei der SC Erfahrungswerte:

Ausdrucksart	Gewichtung
Literale, Konstanten, Variablen	0
Funktionsaufrufe	$1 +$ Parameterzahl
Operatoren +, -	1
Operatoren *, /	2
Operator %	3
Relationsoperatoren (<, >, ...)	1
Operator !	2
Operatoren &&, ||	3
Indizierung (für jeden Index)	2
Feldzugriff	1
Binäre Operatoren (&, |, ...)*	2
Shift Operatoren (<<, >>)*	3
Dereferenzierung und Adresszugriff (&, *)	3

Jeder Teil eines Ausdrucks muss mit dem zugehörigen Faktor gewichtet werden. Die Verschachtelungstiefe geht in die Berechnung exponentiell mit $1.5^{\text{Verschachtelungstiefe}}$ ein. Selbst gewählte Faktoren in der Tabelle sind mit einem * markiert.

DC (data complexity) bewertet alle Variablen, die in einer Funktion vorkommen, mit einem Gewicht, wobei der Gewichtungsfaktor von der Position der Deklaration abhängt. Da Rechenberg keine Klassenvariablen und geerbte Variablen berücksichtigt hat, wurden passende Werte ausgewählt und mit einem * markiert. Die Gewichte sind folgende:

Variablenart	Gewichtung
lokale Variable	1
Parameter	2
Klassenvariablen*	3
geerbte Variablen*	$4+i$

i bezeichnet die Entfernung zu der Vaterklasse, in der die gesuchte Variable deklariert ist. Ist sie in der direkten Vaterklasse deklariert, ist $i = 0$. Ist sie in der Vaterklasse der Vaterklasse so ist $i = 1$, usw.

In einem Ausdruck werden Variablen immer nur einmal gezählt. Rechenberg ging davon aus, dass ein Programmierer sich beim lesen des Codes nur einmal über die Bedeutung einer Variable informieren muss. Wiederholende Vorkommnisse stören ihn hingegen nicht.

Die DC wird am Ende eines Analysedurchganges berechnet, indem der Deklarationsort aller von einer Methode verwendeten Variablen gesucht und auf Grund dieser Informationen die richtige Gewichtung zur Datenkomplexität hinzu addiert wird. [HOL02], [REC86]

Komplexitätsmaß nach Halstead

Halstead's Komplexitätsmaß wird aus folgenden 4 Komponenten berechnet:

n_1 . . . Anzahl der unterschiedlichen Operatoren

n_2 . . . Anzahl der unterschiedlichen Operanden

N_1 . . . Totale Anzahl der Operatoren

N_2 . . . Totale Anzahl der Operanden

Dabei wird bei jedem gelesenem Token entschieden, ob es sich um einen Operator oder einen Operanden handelt. Diese Entscheidung ist aber nicht immer eindeutig zu treffen, wie aus [BOO96], [www.cis.ksu.edu,1] und [CHR81] hervorgeht.

Für die Zählung wurden folgende Entscheidungen getroffen:

- Schlüsselwörter, Arithmetische/Logische/Binäre Operatoren (+, -, <<, ||, etc.) werden als Operatoren gezählt.
- Konstanten werden als Operanden gezählt.
- Vom Benutzer vergebenen Namen (*ident*) werden als Operanden gezählt.
- Klammernpaare (( ), [ ], { }) werden als ein Operator gezählt.
- *do-while* Schleifen werden als ein Operator gezählt.

- Es werden alle Sprachkonstrukte, bis auf Kommentare, gezählt. Es werden also auch Deklarationen, Typ-Deklarationen, Präprozessor- und Compiler-Direktiven berücksichtigt.

Aus diesen Regeln folgt, dass Methodenaufrufe als ein Operand (der Methodenname) und einen „apply“-Operator (den Klammern des Methodenaufrufes "( )") gezählt werden. In der Literatur ist dieser Ansatz genauso zulässig, wie das Zählen des Methodennamens als Operator. Einige Autoren ([www.cs.technion.ac.il,1]) fordern, Operatoren wie + als unterschiedlich zu zählen, sobald sie auf unterschiedliche Typen angewandt werden. Damit wäre das + in `(int)i + (int)j` nicht der gleiche Operator wie in `(float)f + (float)g`. Da keine ausreichenden Typ-Informationen bereitstehen, werden diese beiden Operatoren als ein einziger behandelt.

Jeder einzelne Token wird unabhängig von seiner Position in der Grammatik behandelt, wobei diese Zählung im Scanner als Look-Up in Zuordnungstabellen implementiert ist. Dabei werden auch Token behandelt, die dem Parser als Pragmas übergeben werden, und somit in der Grammatik nicht berücksichtigt würden. Der Right-Shift Operator (>>) ist ein Spezialfall, da er erst in der Grammatik identifiziert wird.

Sind diese vier Kennzahlen bestimmt, lassen sich die Kennzahlen von Halstead mit folgenden Formeln ermitteln:

$$
\begin{array}{lcll}
n & = & n_1 + n_2 & \ldots \text{ Vokabular} \\
N & = & N_1 + N_2 & \ldots \text{ Länge} \\
V & = & N \log_2 n & \ldots \text{ Volumen} \\
D & = & \frac{n_1}{2} \frac{N_2}{n_2} & \ldots \text{ Schwierigkeit} \\
E & = & D \cdot V & \ldots \text{ Aufwand} \\
L & = & \frac{2}{n_1} \frac{n_2}{N_2} & \ldots \text{ Level} \\
T & = & \frac{E}{18} & \ldots \text{ Programmierzeit}
\end{array}
$$

[www.di.uminho.pt,1]

Zum Berechnen von Halstead auf Klassen- Namespace- und Datei-Ebene werden N_1 und N_2 aufsummiert. n_1 und n_2 hingegen müssen angepasst werden, da die mehrfach vorkommenden Operatoren/Operanden nicht mitgezählt werden.

Komplexitätsmaß nach Henry & Kafura

Diese Metrik basiert auf dem Informationsfluss von Methoden und berechnet sich nach folgender Formel:

$$\text{Komplexität} = LOC * (\text{fan-in} * \text{fan-out})^2$$

Wobei *LOC* die Länge der Methode in Zeilen, *fan-in* die Informationsflüsse in die Methode und *fan-out* die Informationsflüsse aus der Methode hinaus bezeichnen.

Informationsflüsse in eine Methode sind die übergebenen Parameter und dieVariablen, auf die die Methode zugreift (wobei Klassenvariablen, globale Variablen, statische Felder von Klassen, etc. *fan-in*'s darstellen). Informationsflüsse aus einer Methode heraus sind der Rückgabewert, die übergebenen Parameter an andere Methoden und Zuweisungen an öffentliche Variablen.

Datenflüsse werden nur ein Mal gezählt, 2 Zugriffe auf eine öffentliche Variable werden beispielsweise nur als ein *fan-in* gezählt.
[HEN81]

2.1.2 Planung

In diesem Abschnitt wird behandelt, wie die Kennzahlen gespeichert und verwaltet werden und wie die Datenstrukturen organisiert sind, um die Berechnungen durchzuführen. Es wird erwähnt, welche Teile von Coco/R zu bearbeiten waren und wie sich die Implementierung vollzog.

Coco/R - Erzeugte Klassen

Coco/R erzeugt aus einer ATG-Datei und zwei Frame-Dateien die Klassen Scanner und Parser, die von einem Main-Programm aufgerufen werden. Die Abbildung 2.2 auf der nächsten Seite zeigt diesen Zusammenhang.

Es wurde hauptsächlich die ATG-Datei verändert, die die attributierte Grammatik enthält, aus der der Parser erzeugt wird. Wie bei der Diskussion der Metriken (in Abschnitt 2.1.1 auf Seite 7) erwähnt wurde, ist es nötig dem Scanner mitzuteilen, in welchem Bereich er gerade ist. Der Scanner soll mit diesen Informationen die Quellcode-Zeilen

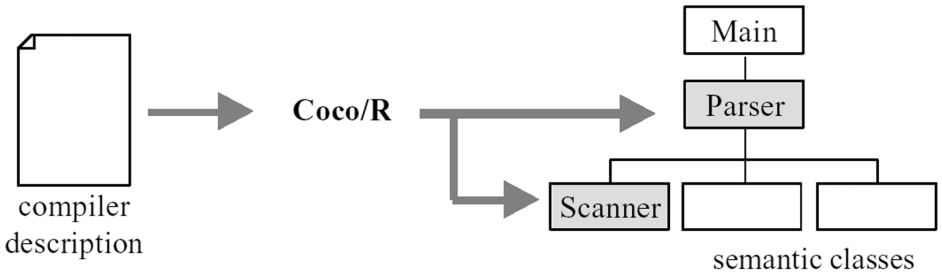


Abbildung 2.2: Klassen erzeugt von Coco/R [MÖS01]

und -Zeichen für jeden Namespace, Klasse und Methode separat zählen, da dies für den Parser sehr umständlich wäre. Zu diesem Zweck wurden die Frame-Dateien des Scanners und des Parsers geändert. Einerseits wurden Methoden hinzugefügt, die es dem Scanner und dem Parser erlauben, den aktuellen Kontext auszutauschen. Andererseits wurde Code geschrieben, der die verschiedenen Kennzahlen berechnet, die im Scanner behandelt werden.

Die Klassenstruktur

Bei der hier vorgestellten Klassenstruktur wird das Programm in einem Baum dargestellt. Die Wurzel ist ein *CodeInfo*-Knoten, der als Kinder *NamespaceInfo*-Knoten aufnimmt. Diese haben *ClassInfo*-Knoten als Kinder, die wiederum *MethodInfo*-Knoten als Blätter enthalten. Es ist mit dieser Struktur möglich, innere Klassen darzustellen. Dabei hat ein *ClassInfo*-Knoten wieder einen *ClassInfo*-Knoten als Kind. Die Abbildung 2.3 auf der nächsten Seite zeigt die verwendete Klassenstruktur und in Abbildung 2.4 auf Seite 23 ist Beispiel-Code mit dem erzeugten Informations-Baum angegeben.

Die große Anzahl der Kennzahlen und die große Anzahl der Hilfsgrößen, die benötigt werden, um die Metriken zu berechnen, lassen das Modell kompliziert erscheinen. Es enthält jedoch einen sehr großen Anteil von reinen Datenfeldern und Klassen, die nur zu Gruppierung von Informationen dienen. Eine unüberschaubare Anzahl von Gettern und Settern wurde vermieden, indem viele Felder als *public* definiert wurden.

Neben den Verwaltungsinformationen für den Baum (*parent*, *children*), gibt es eine Reihe von Feldern, die für bestimmte Kennzahlen zuständig sind. Darüber hinaus

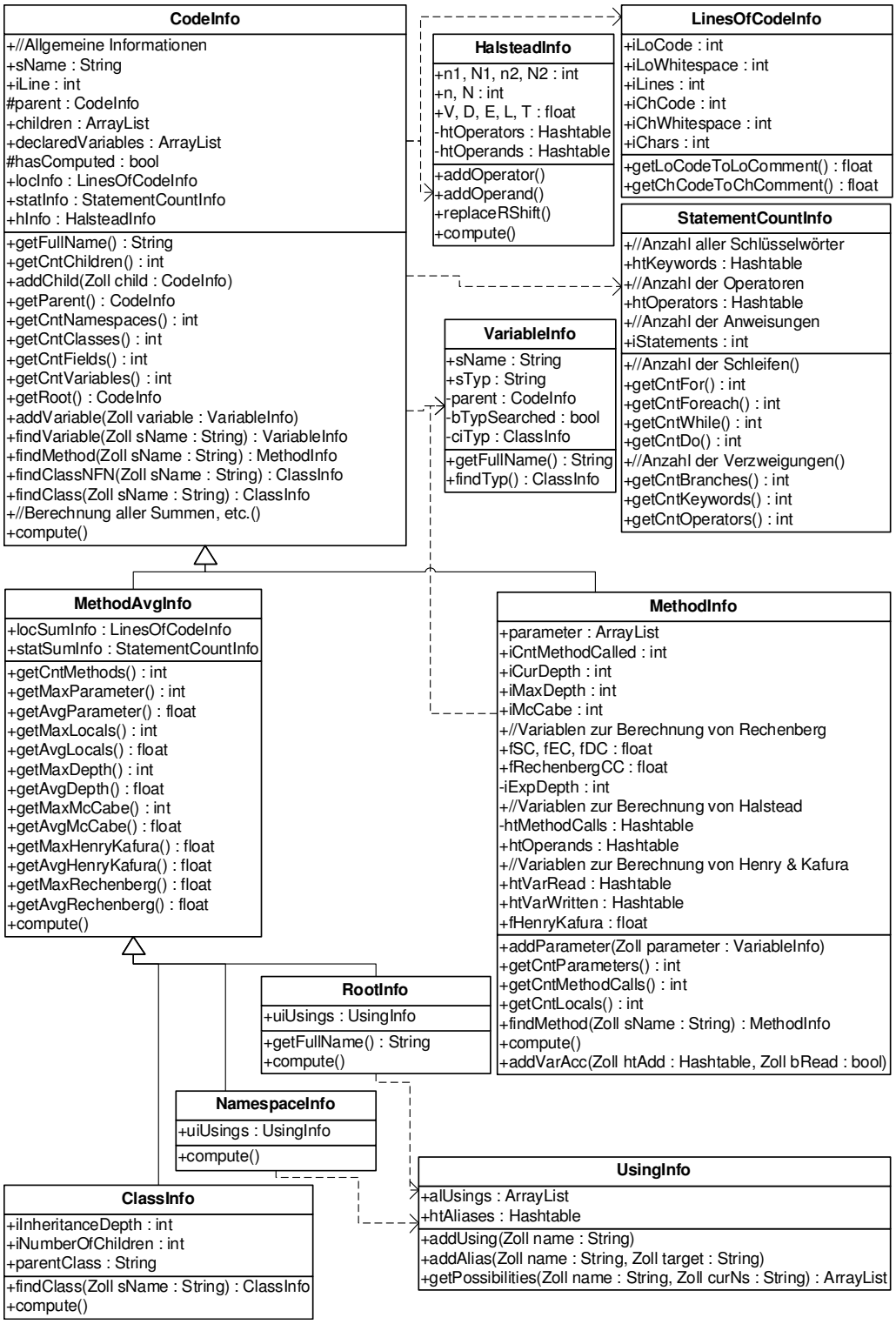


Abbildung 2.3: Klassendiagramm für die Berechnung der Kennzahlen

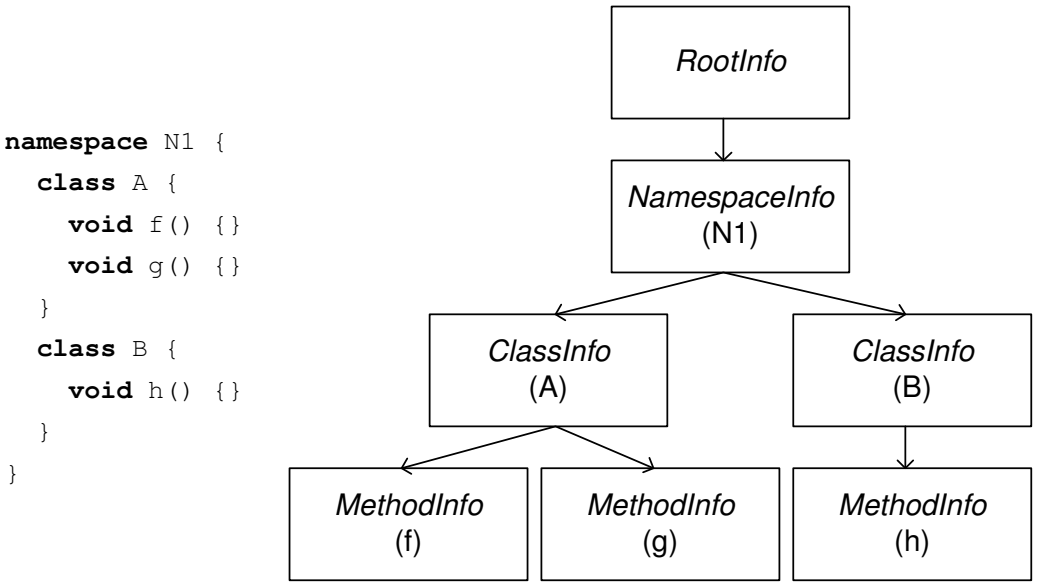


Abbildung 2.4: Beispiel eines Informations-Baums

gibt es noch einige Einträge, die besondere Aufmerksamkeit verdienen und deshalb hier kurz vorgestellt werden:

- *iLine*: Die Zeile, in der diese Einheit (Namespace, Klasse, Methode) definiert wurde. Damit können später die Informationen in die Datei exportiert werden.
- *declaredVariables*: Die Variablen, die in diesem Scope deklariert wurden. Für Methoden sind dies lokale Variablen, für Klassen Klassenvariablen. Diese Informationen werden für einige Kennzahlen benötigt.
- *getFullName()*: Diese Methode gibt den vollen Namen eines Objekts zurück, also z.B. <*Namespace*>.<*Klasse*>.<*Methode*>(<*Anz. Parameter*>). Dieser wird benötigt, um Probleme mit gleichnamigen Methoden und Klassen, die in verschiedenen Namespaces definiert wurden, zu vermeiden.
- *compute()*: Diese Methode berechnet nach einem beendeten Parser-Lauf die Kennzahlen. Dazu ruft sie erst alle *compute()*-Methoden der Kinder des aktuellen Knotens auf, und berechnet dann alle Metriken, Summen, Mittelwerte, etc. auf Grund der bereits berechneten Werte seiner Kinder.
- die Klasse *MethodAvgInfo*: für Klassen, Namespaces, und das gesamte Programm

lassen sich Durchschnitte und Maximalwerte für die einzelnen Kennzahlen bilden, was mit dieser Klasse zum Ausdruck gebracht wird.

- die Klasse *VariableInfo*: diese Klasse fasst Informationen zu einer Variable zusammen. Mit ihrer Methode *findTyp()* lässt sich das *CodeInfo*-Objekt finden, das den Typ dieser Variable darstellt. Über das Feld *parent* ist der Deklarationsort auffindbar.

- *findClass()* und *findClassNFN()*: diese beiden Methoden suchen nach einer Klasse im Baum der Informationen. Die Methode *findClass* sucht nach einer Klasse mit ihrem vollen Namen. Die Methode *findClassNFN()* hingegen sucht nach einer Klasse mit einem Namen, wie er im Quellcode vorkommt. Dazu berücksichtigt diese Methode alle Usings und Aliases. Im Abschnitt 2.2.2 auf Seite 26 wird auf die Funktionsweise genauer eingegangen.

- die Klasse *UsingInfo*: diese Klasse speichert Informationen über die Using-Direktiven einer Datei oder eines Namespaces und wird von der Methode *findClassNFN* zum Auffinden von Klassen verwendet.

Zu beachten ist, dass während eines Parser-Durchlaufes alle Informationen und Kennzahlen für jede Methode, Klasse, Namespace und für das Programm separat berechnet und gespeichert werden, und erst durch den finalen Aufruf der *compute()* Methode des Wurzel-Knotens zusammengefügt werden. So zählt z.B. der Scanner die Anzahl der Zeilen für jede Methode, jede Klasse und jeden Namespace separat, und die Gesamtzahl der Zeilen wird erst durch *compute()* berechnet.

2.2 Implementierung

Die Implementierung wurde wie im Abschnitt 2.1 auf Seite 7, der Analyse und Planung, beschrieben durchgeführt. Es sind jedoch einige Besonderheiten und interessante Schwierigkeiten aufgetreten, die in diesem Abschnitt mit den jeweiligen Lösungen vorgestellt werden.

2.2.1 Kommunikation zwischen Scanner und Parser

Die Problemstellung war, die Kommunikation zwischen Scanner und Parser so herzustellen, dass der Scanner seine Zählungen immer im richtigen Kontext durchführt. Dabei sollen Lines of Code, Operatoren-Anzahlen, Schlüsselwörter-Anzahlen, etc. für Namespaces, Klassen und Methoden separat gezählt werden. Stellt der Parser beispielsweise fest, dass der aktuell analysierte Code in einer Methode ist, dann muss der Scanner die dazugehörigen Code-Informationen für diese Methode bearbeiten.

Wie im Abschnitt 2.1.1 auf Seite 7 diskutiert, wurde dieses Problem dadurch gelöst, dass der Parser dem Scanner bei jedem Wechsel des Kontextes das aktuelle *CodeInfo*-Objekt übergibt. Dabei ergab sich das Problem, dass der Parser immer ein Look-Ahead Symbol hält, das er zur Analyse benötigt. Dieses Symbol wird vom Scanner geliefert, und folglich vom Scanner bereits „gesehen“ und im falschen Kontext gezählt, falls dieses Look-Ahead Symbol einen Kontextwechsel darstellt.

Als Lösung wird ein Buffer-Token erstellt, das das letzte dem Parser übergebene Token enthält. Bei jedem Aufruf von *Scan()* wird dieses Buffer-Token verarbeitet, ein neues Token gelesen und das Buffer-Token aktualisiert. Mit Hilfe nur dieses Buffer-Tokens lassen sich jedoch keine Zählungen von Kommentaren und Leerzeichen durchführen.

Zur Umgehung dieses Problems wurde der Token-Datenstruktur ein *LinesOfCodeInfo*-Feld hinzugefügt. Dieses Feld zählt alle Leer- und Kommentarzeichen und -Zeilen von einem Zeichen nach dem Ende des Vorgänger-Tokens bis zum Ende des aktuellen Tokens. Diese Informationen werden beim Verarbeiten des Buffer-Tokens zum aktuellen Bereich hinzuaddiert. Der dazugehörige Codeausschnitt ist folgender:

```
public Token Scan () {
  if (lastTok != null) {
    // Token auf Schlüsselwörter, Operatoren, und
    // im Sinne von Halstead analysieren
    ...
    // LOC-Informationen des Buffer-Tokens zum
    // aktuellen Bereich addieren
    ciCur.locInfo = ciCur.locInfo + lastTok.loc;
  }
```



```
  Token ret;
  // Neuen Token ermitteln und LOC-Informationen
  // in seinem loc-Feld speichern
  ...

  lastTok = ret; // Buffer-Token aktualisieren
  return ret;
}
```

2.2.2 Auffinden bereits deklarierter Namen

Für die Berechnung einiger Kennzahlen ist es nötig, andere bereits identifizierte Objekte zu finden. Wird z.B. eine Klasse B von einer Klasse A abgeleitet, ist es nötig die Informationen der Klasse A zu finden, um die Vererbungstiefe der Klasse B berechnen zu können. Dieses Auffinden von anderen Objekten ist schwierig, da es einerseits mehrere Objekte mit gleichen Namen in unterschiedlichen Namespaces geben kann, und andererseits ein Objekt mit mehreren Namen angesprochen werden kann, wenn Using- und Alias-Direktiven verwendet werden.

Das Problem wurde gelöst, indem für jedes Objekt ein „voller Name“ ermitteln werden kann. Dieser Name ist der Namespace-Name plus eventuell dem Namen der Klasse, der Methode und dem Variablennamen. Eine Sonderstellung nehmen Methoden ein, da hierbei die Parameteranzahl ebenfalls zum vollen Namen gehört, da Methoden überladen werden können. Da wir nicht über volle Typinformationen verfügen, können überladene Methoden nur dann unterschieden werden, wenn sie unterschiedliche Parameteranzahlen haben.

Der Name, mit dem das Objekt im Code angesprochen wurde, muss zu einem vollen Namen erweitert werden um ein Objekt im Informations-Baum wiederzufinden. Dazu wurde die Klasse *UsingInfo* erstellt, welche die Informationen über Usings und Aliases eines Namespaces oder der ganzen Datei enthält.

Will man die Vaterklasse einer Klasse finden, beginnt man bei der am nächsten gelegenen Using-Direktive und versucht anhand dieser den vollen Namen zu konstruieren. Die am nächst gelegen Using-Direktive ist jene, die man beim Aufsteigen im Informationen-Baum als erstes findet, im Beispielfall also die Using-Direktive des die

Klasse umgebenden Namespaces. Findet man keine, wird weiter zur nächsten Using-Direktive gegangen. Gibt es keine solche mehr, dann kann das gewünschte Objekt nicht gefunden werden. Dies ist der Fall, wenn auf Objekte verwiesen wird, die in anderen Dateien definiert wurden und folglich nicht analysiert wurden. Diese Vorgehensweise wird auch zum Auffinden von Methodennamen und Variablen verwendet.

Diese Methode wird, wie in folgendem Beispiel gezeigt, in der Methode *compute()* der Klasse *ClassInfo* eingesetzt, um die Vaterklasse zu finden und die Vererbungstiefe zu berechnen:

```
...
while (ciTmp != null && bContinue) {
  // Ermitteln, ob ciTmp Informationen über
  // Usings enthält
  ...
  // Falls ja, versuchen den vollen Namen
  // der Vaterklasse zu konstruieren und zu finden
  if (uiTmp != null) {
    foreach (String s in
     uiTmp.getPossibilities(parentClass, getFullName())) {
      if ((clsTmp=getRoot().findClass(s)) != null) {
        // Falls Vaterklasse gefunden, dann zuerst deren
        // compute()-Methode aufrufen um zu verhindern,
        // falsche Werte im iInheritanceDepth-Feld zu haben
        clsTmp.compute();
        iInheritanceDepth = clsTmp.iInheritanceDepth + 1;
        clsTmp.iNumberOfChildren++;
        // aufhören, da bereits gefunden
        bContinue = false;
        break;
      }
    }
  }
  // suchen der nächsten Using-Direktive
  ciTmp = ciTmp.getParent();
```



```
}
...
```

Innerhalb dieser Methode wird die Methode *getPossibilities()* der Klasse *UsingInfo* aufgerufen. Diese Methode konstruiert, wie oben beschrieben, die möglichen vollen Namen der Vaterklasse.

```
public ArrayList getPossibilities
                              (String name, String curNs) {
  ArrayList alPos = new ArrayList();
  // aufnehmen des Names wie er im Code steht
  // als eine Möglichkeit
  alPos.Add(name);
  // curNs enthält den vollen Namen des umgebenden
  // Code-Teiles (Klasse, Methode). Ersetzen des
  // Strings hinter dem letzten Punkt um einen vollen
  // Namen im gleichen Namespace zu erzeugen.
  int iTmp = curNs.LastIndexOf(".");
  if (iTmp != -1)
    alPos.Add(curNs.Substring(0, iTmp+1) + name);
  // ersetzen des Aliases wenn verwendet
  foreach (DictionaryEntry de in htAliases) {
    if (name.StartsWith((String)de.Key)) {
      //Alias gefunden
      name = de.Value +
                name.Substring(((String)de.Key).Length);
      alPos.Add(name);
      break;
    }
  }
  // voranstellen aller definierten Usings.
  foreach (String s in alUsings) alPos.Add(s + "." + name);
  return alPos;
}
```


2.2.3 Einige Beschränkungen

Durch den Mangel von Typinformationen gibt es einige Informationen, die nicht oder nicht exakt berechenbar sind. Werden beispielsweise Funktionen der Klassenbibliothek aufgerufen, sind keinerlei Informationen über deren Rückgabetyp bekannt. Trifft man z.B. folgenden Code an

```
...
StringBuilder sb = new StringBuilder();
sb.Append("Hallo ").Append("Welt!");
...
```

ist man nicht in der Lage festzustellen, mit welchem Typ und welchem Objekt das 2. *Append()* aufgerufen wird. Dieses Problem spielt eine Rolle, wenn wir die Anzahl der Aufrufe einer Methode berechnen. Diese Einschränkung wurde jedoch nicht weiter behandelt. Einerseits wäre es schwierig (im Falle von Methodenaufrufen der Klassenbibliothek ohne umfassende Informationen unmöglich) und aufwendig, Konstrukte dieser Art zu analysieren. Andererseits kann die Anzahl der Aufrufe einer Methode nur eine Schätzung sein, da es bei virtuellen Methodenaufrufen definitiv unmöglich ist, die aufgerufene Methode zu ermitteln.

Weiters wurde auf eine Berücksichtigung von Cast's verzichtet, da hier umfassende Typinformationen nötig wären die den Rahmen dieser Bakkalaureatsarbeit sprengen würden. Daher werden Konstrukte wie

```
...
((String)str).IndexOf("Hallo");
...
```

nicht richtig als Methodenaufruf der Methode *IndexOf()* der Klasse String interpretiert.

3 Grafische Oberfläche

In diesem Teil geht es darum die schon berechneten Kennzahlen, die eine umfangreiche Datenmenge darstellen, grafisch gegliedert und geordnet darzustellen. Dem Benutzer soll ermöglicht werden, die errechneten Daten einfach und intuitiv zu verstehen und zu analysieren.

Außerdem ist die grafische Oberfläche dafür zuständig, die ermittelten Informationen als Text-Datei zu exportieren oder direkt als Kommentare in die Quellcode-Dateien zu schreiben.

3.1 Analyse und Planung

Die naheliegenste Möglichkeit die gesammelten Daten auszugeben ist, sie in einer Baumstruktur aufzubereiten. In dieser Struktur sollen die Einheiten Datei, Namespace, Klasse und Methode in der Art und Weise dargestellt werden, in der sie auch im Code vorhanden sind. Das heißt, eine Klasse ist ein Unterknoten eines Namespaces und hat als Unterknoten mehrere Methoden oder auch Unterklassen.

Jede dieser Einheiten besitzt eigene Informationen, die sich durch Auswählen dieser Einheit anzeigen lassen. Manche Einheiten, wie Methoden, besitzen sehr viele Informationen. Bei ihnen ist es von Vorteil, diese Informationen thematisch gruppiert auszugeben.

Durch diese Zweiteilung der Informationen (Struktur- und Detailinformationen) ist eine Zweiteilung des Ausgabefensters die beste Lösung zur Darstellung. In einer Baumansicht wird die Struktur der Daten dargestellt, in der Elemente ausgewählt werden können. In einem Detailfenster wird eine geeignete Darstellung für das ausgewählte

Element angezeigt. Für eine Methode bietet es sich beispielsweise an, mehrere Reiter mit allgemeinen Informationen (Name, deklarierte Variablen, aufgerufenen Methoden, etc.), statistischen Informationen (Codezeilen, Anzahl der Schlüsselwörter, etc.) und errechneten Informationen (Kennzahlen) zu erstellen.

Die Zusatzfunktionen der grafischen Oberfläche wie Öffnen einer Datei, Exportieren der Informationen in die Source-Code-Datei oder Speichern der Informationen werden in den Menüs des Programmes untergebracht, in dem sich auch ein kurzer Hilfetext zur Einführung in das Programm befindet.

3.2 Implementierung

Die Implementierung wurde mit C# durchgeführt, und beruht auf einzelnen grafischen Komponenten. Das heißt, dass sich die Oberfläche aus „Bausteinen“ zusammensetzt, die miteinander kommunizieren um so die Komplexität des Programmes aufzuteilen.

Es wurde für die Baumansicht, in der die Struktur des analysierten Codes zu sehen ist, ein *TreeView* verwendet, das auf der linken Seite des Hauptfensters „klebt“. Für die Hauptansicht wurde eine neue, von *UserControl* abgeleitete, Komponente definiert. Darin ist ein *TabControl* untergebracht, das mehrere *TabPage*'s beinhaltet. Für jede darzustellende Informationsgruppe (ganze Datei, Namespace, Klasse, Methode) wurde noch einmal weiter abgeleitet, um Gruppenspezifische Informationen anzuzeigen indem die einzelnen *TabPage*'s unterschiedlich gefüllt werden.

Die Kommunikation zwischen den Komponenten wird ausgelöst, wenn der Benutzer auf ein Element im *TreeView* klickt. Daraufhin wird festgestellt, welchen Typ das ausgewählte Objekt hat und es wird ein neues *TabControl* im Hauptfenster erzeugt, das mit den korrespondierenden Informationen gefüllt wird.

Die Oberfläche wird in Abbildung 3.1 auf der nächsten Seite dargestellt.

Die berechneten Daten können als Kommentare in die Quellcode-Datei exportiert werden, wobei man aussuchen kann, welche Informationen geschrieben werden sollen (wie in Abbildung 3.2 auf der nächsten Seite dargestellt). Es werden dabei die eventuell vorhandenen alten Informationen entfernt. Dies erlaubt ein Löschen von allen bereits exportierten Informationen indem ein Export gemacht wird, bei dem nichts exportiert wird.

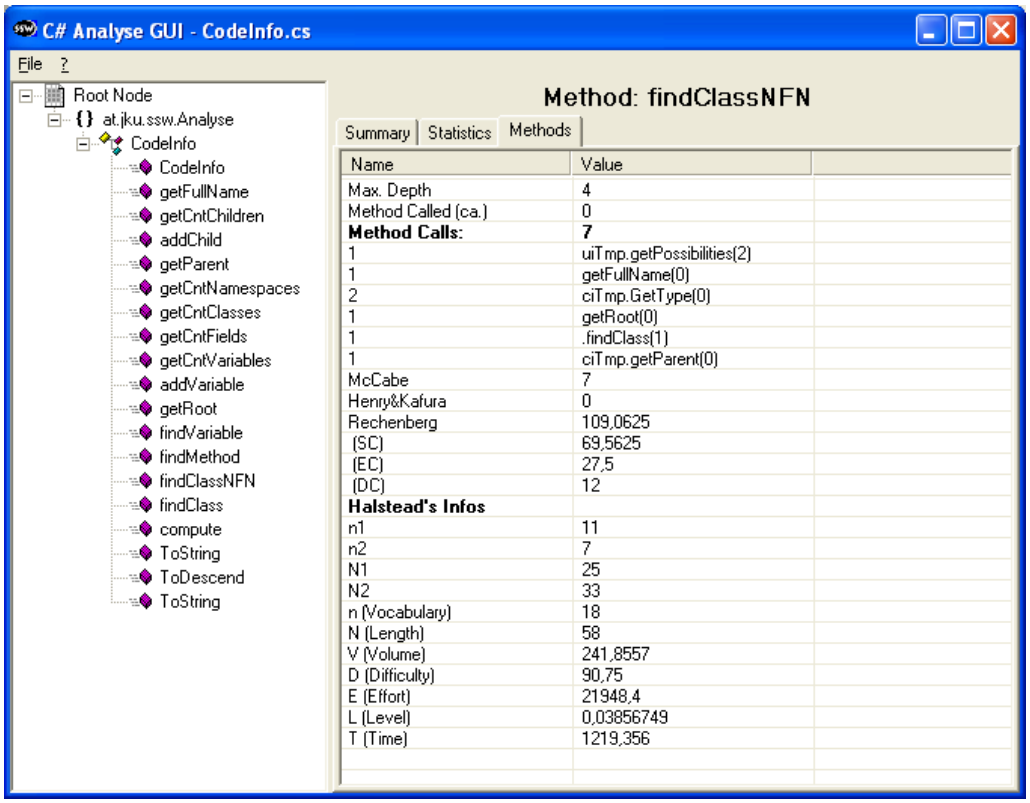


Abbildung 3.1: Screenshot der Anwendung

Abbildung 3.2: Auswahl der zu exportierenden Informationen

Die Anwendung ist im wesentlichen selbsterklärend und ihre Funktionsweise ist in diesem Dokument beschrieben. Deshalb wurde nur eine rudimentäre Hilfe (Abbildung 3.3 auf der nächsten Seite) hinzugefügt, die die ersten Schritte der Anwendung beschreibt und einige allgemeine Informationen bereitstellt.

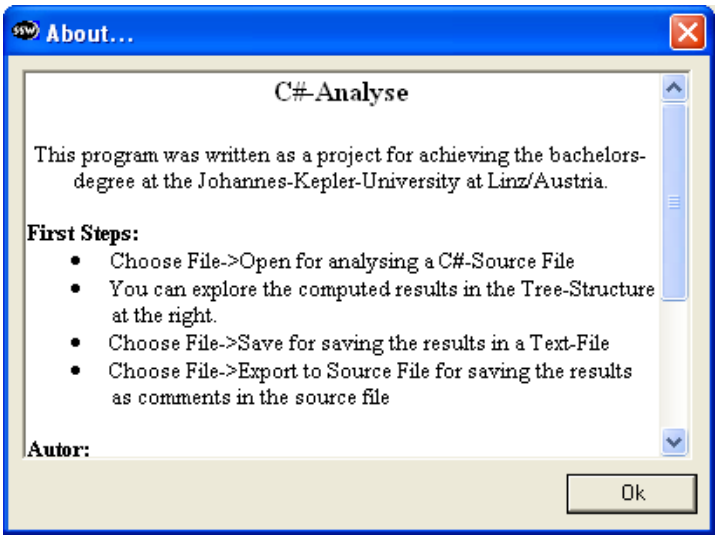


Abbildung 3.3: Der Hilfe-Dialog

4 Testfälle

Folgend werden einige Testfälle und die dazugehörigen Ausgaben angeführt, die zu den jeweilig getesteten Kennzahlen gehören.

4.1 Lines of Code

Testcode

```
// A Comment with '//' followed by a Newline

/* A Comment with '/*' in only one line */
/* A Multiline-Comment with '/*'
   which goes over 3 lines and which
   is followed by a Newline */

class A {
  void main() {
  }
}
```

Ergebnisse

	Datei (summiert)		Klasse		Methode	
	Zeilen	Zeichen	Zeilen	Zeichen	Zeilen	Zeichen
Code	4	20	2	10	1	4
Whitespace	2	8	0	4	0	3
Comment	5	184	0	0	0	0
Total	11	212	2	14	1	7
Code/Comment	0,8	0,109	0	0	0	0

4.2 Komplexitätsmaße

Testcode

```
class A {
  void f(ref int x, ref int y) {
    int z = x;
    x = y; y = z;
  }

  void g(int n) {
    char[] a = new char[n];
    for (int i=0; i<n; i++)
      a[i] = 'x';
  }

  bool h(int n, char[] a, char c) {
    for (int i=0; i<n; i++)
      if (a[i] == c) return true;
    return false;
  }

  void i(int n, char[] a) {
```



```
    for (int i=0; i<n; i++)
      for (int j=i+1; j<n; j++)
        if (a[i] > a[j]) {
          char t = a[j];
          a[j] = a[i];
          a[i] = t;
        }
  }
}
```

Ergebnisse

	f()	g()	h()	i()
Max. Depht	0	1	2	3
McCabe	1	2	3	4
Henry & Kafura	48	0	36	32
Rechenberg	8	17,5	21,92	47,63
SC (Statement Compl.)	3	6,5	10,25	24,63
EC (Expression Compl.)	0	3	4,67	16
DC (Data Compl.)	5	8	7	7
Halstead				
n_1	7	11	16	14
n_2	3	5	5	7
N_1	14	21	26	40
N_2	8	11	11	27
n (Vocabulary)	10	16	21	21
N (Length)	22	32	37	67
V (Volume)	73,08	128	162,52	294,29
D (Difficulty)	9,33	12,1	17,6	27
E (Effort)	682,1	1548,8	2860,28	7945,7
L (Level)	0,107	0,082	0,057	0,037
T (Time)	37,89	86,04	158,9	441,43

4.3 Klassenhierarchie

Testcode

```
namespace n1 {
  class A {
    public void f() { }
  }

  class B : A {
    public void g() { f(); }
  }
}

namespace n2 {
  class C : n1.B {
    public void h() { f(); g(); }
  }
}

namespace n3 {
  using alias = n1;
  using n2;

  class D : alias.A {
    public void i() { f(); }
  }

  class E : C {
    public void j() { h(); }
  }
}
```


Ergebnisse

	A	B	C	D	E
Voller Name	n1.A	n1.B	n2.C	n3.D	n3.E
Mutterklasse		A	n1.B	alias.A	C
Vererbungstiefe	0	1	2	1	3
Anzahl Kinder	2	1	1	0	0

	f()	g()	h()	i()	j()
Voller Name	n1.A.f(0)	n1.B.g(0)	n2.C.h(0)	n3.D.i(0)	n3.E.j(0)
Aufgerufen (ca.)	3	1	1	0	0
Anzahl der Aufrufe	0	1	2	1	1

5 Vergleich mit bestehenden Lösungen

In diesem Kapitel werden bestehende Lösungen vorgestellt, die sich auf die Berechnung von Software-Metriken konzentrieren oder die die Berechnung von Software-Metriken für weitergehende Aufgaben nutzen. Es wurden bevorzugt Produkte ausgewählt, die mindestens C# unterstützen.

5.1 McCabe IQ

Dieses kommerzielle Produkt ist spezialisiert auf die von McCabe definierten Metriken. Neben der McCabe Cyclomatic Complexity, die auch in unserem Projektes berechnet wird, berechnet McCabe IQ die McCabe Essential Complexity, die Module Design Complexity, die Integration Complexity, die Komplexität nach Halstead und noch weitere. Außer C# werden C/C++, Java, Perl und weitere Programmiersprachen unterstützt. Das Programm kann den Kontrollfluss von Methoden grafisch darstellen um die Komplexität von gegebenen Code anschaulich zu präsentieren.

Der eigentliche Nutzen dieses Programmes ist es aber, die Komplexität verschiedener Programmteile und Module zu bewerten und grafisch darzustellen. McCabe IQ kann unter anderem die Struktur eines Programmes darstellen und simple Teile grün, schwierige gelb und sehr komplexe rot einfärben. Diese Darstellung kann dem Qualitätsmanagement helfen, fehlerträchtigen Code zu identifizieren und Entwicklungs-Ressourcen auf fehleranfällige Teile zu konzentrieren.

Das Programm unterstützt eine grafische Darstellung der Verbesserung der Qualität. Es versucht festzustellen, welche Codeteile am fehleranfälligsten sind, indem die Häufigkeit der Änderungen und die Komplexität dieser geänderten Codeabschnitte analysiert werden.

Ein anderes Feature von McCabe IQ ist das Auffinden von redundantem Code, bzw. redundanten Modulen. Dabei versucht das Programm ähnlichen Code/Module aufzufinden und dem Benutzer zum Überarbeiten anzubieten. Falls sich die Entwickler entscheiden, den Code zu vereinheitlichen, kann das Programm die neue Komplexität mit der Vorherigen vergleichen und so können Verbesserungen bzw. Verschlechterungen festgestellt werden.

Zusammenfassend kann gesagt werden, dass McCabe IQ als kommerzielles Produkt weit über die Möglichkeiten unseres Programmes hinausgeht. Einerseits wird eine größere Anzahl von Kennzahlen berechnet (die auch auf Projekt-Ebene, und nicht nur auf Dateiebene, wie in dem erstellten Programm, berechnet werden), und andererseits werden dem Benutzer umfangreiche Möglichkeiten geboten, die berechneten Daten zu nutzen. [www.mccabe.com,1]

5.2 LDRA Testbed

Dieses ebenfalls kommerzielle Produkt ist spezialisiert auf die statische Analyse von Programmcode. Zu den unterstützten Sprachen von LDRA Testbed gehören neben C#, C/C++, Java und Visual Basic noch weitere objektorientierte und imperative Programmiersprachen. Es berechnet Komplexitätsmaße wie die Cyclomatic Complexity oder die Anzahl der Knoten im Code, aber es unterstützt auch anderen Aufgaben, die nicht Bestandteil dieser Bakkalaureatsarbeit waren.

LDRA Testbed kann Source-Code auf benutzerdefinierbare Programmier-Standards überprüfen und die Struktur des Codes verifizieren. Das Programm kann lokale und globale Variablen verfolgen und deren Zugriff auch über Datei-Grenzen hinweg protokollieren. Es kann, ähnlich wie McCabe IQ, redundanten Coder identifizieren und unerreichbaren Code finden (wie z.B. Code hinter einem *return*).

Darüber hinaus können noch verschiedene Analysen durchgeführt werden, wie Datenfluss-Analysen oder Informationsfluss-Analysen. Es werden die Interfaces von Methoden auf eventuelle Fehleranfälligkeit hin überprüft und das korrekte Aufrufen von ihnen im Code verifiziert.

Wie im Falle von McCabe IQ ist auch LDRA Testbed als kommerzielles Produkt umfangreicher als unser Programm. Bei diesem Produkt ist die Berechnung von Software-

Metriken jedoch nur ein kleiner Teil des Funktionsumfanges, der als solcher nicht so umfangreiche Berechnungen anstellt wie es in unserer Lösung der Fall ist.
[www.ldra.co.uk,1]

5.3 CCCC - C and C++ Code Counter

Dieses Programm ist als OpenSource verfügbar und ist im Rahmen einer Diplomarbeit entstanden. Dieses Produkt wurde ausgewählt, um einen Vergleich mit einem nichtkommerziellen Programm geben zu können. Da kein vergleichbares Programm, das C# unterstützt, öffentlich zugänglich ist, beschränken wir uns auf den Vergleich mit diesem älteren, nicht mehr weiterentwickelten Tool für C/C++.

CCCC ist ein Kommandozeilen-Programm, das aus gegebenen Source-Dateien einen HTML-Report generieren kann. Zu diesem Zweck analysiert das Programm mit einem Parser für C oder C++, ausgewählt nach der Dateiendung, die Quellcode-Dateien und erstellt eine interne Datenbank. Basierend auf dieser Datenbank werden anschließend die Kennzahlen berechnet und als HTML formatiert ausgegeben.

Die berechneten Kennzahlen sind Lines of Code, McCabe's Complexity, Kommentarzeilen und Kommentaranteil im Sourcecode und Fan-in und Fan-out, die zur Berechnung von Henry und Kafura eingesetzt werden. Anders als in unserem Projekt, bei dem Henry und Kafura auf Methoden-Ebene berechnet wird, berechnet CCCC Henry und Kafura auf Modul-Ebene. Zusätzlich zu diesen Maßen werden noch die Anzahl der Module gezählt und die durchschnittliche Anzahl der Methoden pro Klasse.

Im Vergleich zu unserem Programm fällt auf, dass CCCC keinerlei Komplexitätsmaße berechnet, die die Expression-Komplexität berücksichtigen, wie z.B. Halstead oder Rechenberg. Als Vorteil gegenüber unserer Arbeit wertet CCCC mehrere Informationen über die Bindung von Modulen bzw. Klassen aus, wodurch ein besserer Überblick über die Komplexität von Projekten ermöglicht wird.
[sourceforge.net,1]

5.4 Lachesis

Als zweiten Vertreter aus der OpenSource-Szene wurde Lachesis ausgewählt, das ebenfalls im Rahmen einer Diplomarbeit entstanden ist. Dieses Programm analysiert Java-Sourcecode und ist daher nicht 100%-ig vergleichbar mit unserer Arbeit. Anders als CCCC wird an Lachesis weitergearbeitet und es soll in der Zukunft auch andere objektorientierte Sprachen unterstützen.

Dieses Programm berechnet auf Ebene von Methoden im Wesentlichen nur die McCabe und Halstead Komplexität. Darüber hinaus werden aber noch eine Reihe von objektorientierten Komplexitätsmaßen wie Chidamber und Kemerer oder die Tiefe des Vererbungs-Baumes berechnet. Lachesis versucht, ähnlich wie das in dieser Bakkalaureatsarbeit entstandene Programm, Funktionsaufrufe zu analysieren und somit Komplexitätsmaße wie Abhängigkeit von Klassen zu berechnen. Laut dem Autor des Programms [lachesis.sourceforge.net,1] sind die Erkennungsquoten dieser Funktionsanalyse relativ weit fortgeschritten und arbeiten zufriedenstellend.

Zusammenfassend kann man sagen, dass Lachesis, ähnlich wie CCCC, mehr Wert auf die Analyse von Komplexitäten von Modulen und Klassen legt, als auf die Analyse von Methoden. Im Vergleich zu unserer Arbeit werden nur rudimentäre „klassische“ Komplexitätsmaße wie McCabe und Halstead unterstützt, dafür aber recht umfangreich die objektorientierten Merkmale analysiert, die in unserer Arbeit nur am Rande gestreift wurden.
[sourceforge.net,2]

6 Mögliche Erweiterungen

Während der Arbeit an dieser Bakkalaureatsarbeit und im Zuge der Beschäftigung mit der Qualität von Software sind uns einige Möglichkeiten aufgefallen, wie unser Programm noch weiterentwickelt und vervollständigt werden könnte. Diese möglichen Erweiterungen und Ergänzungen werden an dieser Stelle kurz präsentieren:

- Unser Programm ist auf die Analyse einer einzelnen Quellcode-Datei beschränkt. In der Realität werden Programme jedoch immer auf mehrere Dateien aufgeteilt, und zu Projekten zusammengefasst. Eine Unterstützung von mehreren Dateien, mit damit verbundenen Analysen wie beispielsweise das Erkennen von Ableitungshierarchien über mehrere Dateien hinweg, würde dem Analyse-Tool zu mehr praktischem Nutzen verhelfen.

- Wenn eine Unterstützung für mehrere Quellcode-Dateien implementiert wird, dann wäre es auch sinnvoll, objektorientierte Metriken wie Henry & Kafura auf Klassen-Ebene, Bindung zwischen Klassen, etc. zu berechnen.

- Es wäre sinnvoll, den Begriff eines „Projektes“ einzuführen. Man könnte in einem Projekt alle zu analysierenden Dateien speichern, und die Ergebnisse der Analyse nach Datum geordnet verwalten. Damit wäre es möglich, den Verlauf der Komplexität eines Projektes nachzuvollziehen. Grafische Darstellungen wären damit vorstellbar.

- Die Darstellung der berechneten Daten könnte noch verbessert werden. Eine Möglichkeit wäre, den Code anzuzeigen und die komplexen Methoden rot, die Simpleren grün zu hinterlegen. Es wäre vorstellbar, die Klassenhierarchie grafisch darzustellen (was wieder in Verbindung mit der Analyse von mehreren Dateien geschehen sollte). Methoden könnten als Flussgrafen dargestellt werden, und dort könnten

wiederum komplexe Bereich rot und simple grün markiert werden. Dafür würde sich z.B. die Expression Complexity nach Rechenberg anbieten.

- Man könnte noch verschiedene Export-Möglichkeiten hinzufügen. Es wäre sinnvoll, HTML-Dateien zu generieren, oder die Daten in einem von Excel oder OpenOffice lesbaren Format zu speichern, um sie nachher individuell weiterverarbeiten zu können (z.B. in Diagrammen).

- Eine Verbesserung der Analyse von Methodenaufrufen wäre wünschenswert. Dazu müssten beispielsweise Cast's berücksichtigt werden (wie in `((A)x).f(...)`), oder Rückgabetypen (wie in `x.f(...).g(...)`). Auch eine genauere Analyse von überladenen Methoden wäre sinnvoll (momentan wird anhand der Anzahl der Parameter unterschieden, nicht nach ihren Typen).

- Im Zuge des vorherigen Vorschlags wäre es sinnvoll, sich ein ausgefeilteres Typensystem zu überlegen. Es sollten Typen auf Gleichheit, Kompatibilität, etc. geprüft werden können. Damit ließen sich genauere Analysen realisieren (besonders auf dem Niveau der Methodenaufrufe) und weiterführende Aufgaben durchführen.

- Bei einer größeren Anzahl von analysierten Methoden kann es sehr schwierig sein, den Überblick zu behalten. Es wäre deswegen sinnvoll, wenn der Anwender für die verschiedenen Kennzahlen Grenzwerte angeben könnte, ab welchen das Programm Warnungen ausgeben soll. Somit wäre es für den Benutzer einfacher, komplexe Projekte schnell zu überblicken.

7 Literatur

- [**ECMA,1**] Standard ECMA-335
  Common Language Infrastructure (CLI)
  3^{rd} edition (June 2005)
  `http://www.ecma-international.org/publications/standards/Ecma-335.htm`, 03.05.2006

- [**ECMA,2**] Standard ECMA-334
  C# Language Specification
  3^{rd} edition (June 2005)
  `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, 03.05.2006

- [**ssw.jku.at,1**]
  `http://ssw.jku.at/Research/Projects/Coco/index.html`, 03.05.2006

- [**www.microsoft.com,1**]
  `http://www.microsoft.com/downloads/details.aspx?familyid=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en`, 03.05.2006

- [**www.dcs.qmw.ac.uk,1**]
  `http://www.dcs.qmw.ac.uk/~norman/papers/qa_metrics_article/section_5_key_metrics.html`, 05.10.2005

- [**www.dacs.dtic.mil,1**]
  `http://www.dacs.dtic.mil/techs/baselines/reliability.html`, 05.10.2005

- [**de.wikipedia.org,1**]
  Artikel McCabe-Metrik.
  In: Wikipedia, Die freie Enzyklopädie.
  Bearbeitungsstand: 1. Februar 2006, 07:58 UTC.
  URL: `http://de.wikipedia.org/w/index.php?title=McCabe-Metrik&oldid=13248337`, 15.05.2006

- [**www.di.uminho.pt,1**]
  `http://www.di.uminho.pt/~joostvisser/software/UMinhoHaskellSoftware-1.0/Language.Sdf.Metrics.Halstead.html`, 05.10.2005

- [**BOO96**] Simon P. Booth, Simon B. Jones, „Are Ours Really Smalle Than Theirs?“, Departement of Computer Science and Mathematics, University of Stirling, Technical Report CSM-141, November 1996

- [**www.cis.ksu.edu,1**]
  `www.cis.ksu.edu/~dag/540fall05/materials/measure23.ppt`, 05.05.2006

- [**CHR81**] K. Christensen, G. P. Fitsos, C. P. Smith, „A perspective on software science“, IBM SYST J, Vol. 20, No. 4, Pages: 372-387, 1981

- [**www.cs.technion.ac.il,1**]
  `http://www.cs.technion.ac.il/Courses/OOP/slides/export/236804-Fall-1997/metrics/part1.html`, 05.05.2006

- [**HOL02**] Seminar Softwareentwicklung (Programmierstil), WS2002/2003
  Johannes Kepler Universität Linz, System Software Group
  Betreuer: Prof. Hanspeter Mössenböck
  Autor: Clemens Holzmann

- [**REC86**] Peter Rechenberg, „Ein neues Maß für die softwaretechnische Komplexität von Programmen“, Informatik - Forschung und Entwicklung, Heft 1, Seiten: 26-37, 1986

- [**HEN81**] Sallie Henry, Dennis Kafura, Kathy Harris, „On the relationships among three software metrics“, Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality, Pages: 81-88, 1981, ISBN:0-89791-038-9

- [**MÖS01**] Hanspeter Mössenböck, „The Compiler Generator Coco/R, User Manual“, Johannes Kepler Universität Linz

- [**www.mccabe.com,1**]
  `http://www.mccabe.com/iq_qa.htm`, 11.02.2006

- [**www.ldra.co.uk,1**]
  `http://www.ldra.co.uk/staticanalysis.asp`, 11.02.2006

- [**sourceforge.net,1**]
  `http://sourceforge.net/projects/cccc`, 12.02.2006

- [**lachesis.sourceforge.net,1**]
  `http://lachesis.sourceforge.net/news/changelist.html`, 12.02.2006

- [**sourceforge.net,2**]
  `http://sourceforge.net/projects/lachesis`, 12.02.2006

