Reverse Engineering generierten Quelltexts durch Analyse von Velocity Templates

Diplom I Arbeit

Fachgebiet Software Engineering

Prof. Dr. Albert Zündorf

Universität Kassel

im März 2007

von

Manuel Bork

Betreuer: Dipl. Inform. Christian Schneider
Prof. Dr. Albert Zündorf

Zusammenfassung

Heutzutage erfreuen sich CASE-Tools (Computer Aided Software Engineering) großer Beliebtheit bei der Entwicklung von Software. Neben dem einfachen Ansatz, nur Code aus UML-Klassendiagrammen generieren zu können, gibt es komplexe Tools, die auch aus weiteren Diagrammen Quelltext generieren können.

Die *FUJABA-Tool-Suite* [7] verwendet für den finalen Schritt der Quelltexterzeugung eine Template Bibliothek: *Apache Velocity Templates* [10]. Um ein *Roundtrip-Engineering* durchführen zu können, ist es auch notwendig, den umgekehrten Weg zu beschreiten: Den generierten Quelltext einzulesen und wieder als Diagramm im CASE-Tool zu visualisieren. Die vorliegende Diplomarbeit beschäftigt sich mit dem Ansatz, die zur Generierung verwendeten Templates zum *Reverse-Engineering* zu verwenden. Im Zuge dieser Diplomarbeit ist ein Plugin für die FUJABA-Tool-Suite entstanden, welches den vorgestellten Ansatz erfolgreich implementiert.

Abstract

Nowadays CASE tools (Computer Aided Software Engineering) are popular for the development of software. Aside from the simple approach to generate code only for UML class diagrams there are more complex tools that can generate code for other UML diagrams.

The *FUJABA Tool Suite* [7] uses the *Apache Velocity Template Engine* [10] in the final step of code generation. In order to perform *Roundtrip Engineering* it is neccessary to execute the process vice versa: to import the generated code and visualize it again as UML diagrams. The present diploma thesis deals with using the templates, which were used for code generation in the first place, for the *Reverse Engineering* process. A plugin for the FUJABA Tool Suite has been developed during this thesis, which implements the proposed approach successfully.

Inhaltsverzeichnis

Zusammenfassung **i**

Inhaltsverzeichnis **iii**

1 Einleitung **1**

1.1 Motivation 1

1.2 Voraussetzungen 2

1.2.1 Fujaba 2

1.2.2 CodeGen2 4

1.2.3 Velocity Template Engine 6

2 Umsetzung/ Methode **11**

2.1 Funktionsprinzip 14

2.2 Parser 15

2.2.1 Referenzen 16

2.2.2 Terminale 17

2.2.3 Set-Direktiven 19

2.2.4 If-Abfragen 20

2.2.5 Schleifen 23

2.2.6 Parse-Direktiven 24

2.3 Reasoning 24

2.4 Tokenerzeugung 28

2.5 Modellaktualisierung 31

3 Ergebnisse **33**

3.1 Laufzeitverhalten 33

3.2 Anforderungen an die Templates 34

3.3 Fazit und Ausblick 35

A Anhang **37**

A.1 Template Beispiele 37

A.2 Klassendiagramme 40

Literaturverzeichnis 43

Abbildungsverzeichnis 44

1 Einleitung

1.1 Motivation

Zur Umsetzung von Softwareprojekten ab mittlerer Größe gewinnt die Einhaltung eines gut strukturierten Entwicklungsprozesses an Bedeutung. Unabhängig von der Wahl des Vorgehensmodells (Wasserfallmodell, V-Modell, Spiralmodell, Unified Process, Extreme Programming, etc.) erhöht man sowohl die Produktivität der beteiligten Entwickler als auch die Qualität der zu erstellenden Software durch die Verwendung eines CASE[1] Tools. CASE Tools, also Werkzeuge zur computergestützten Softwareentwicklung, lassen sich dann am produktivsten einsetzen, wenn sie den gewählten Entwicklungsprozess möglichst gut unterstützen. Ausgehend von der Analyse der Anwendungsdomäne helfen sie den Entwicklern während der verschiedenen Phasen des eingesetzten Entwicklungsprozesses. Je nach eingesetztem Tool kann der Entwickler dabei die zu erstellende Software in unterschiedlicher Granularität modellieren, bei Erweiterungen *Refaktorierungen*[2] anwenden und dann aus den erstellten Diagrammen (beispielsweise in UML-Notation [5]) Quelltext generieren.

Unabhängig von der ursprünglichen Wahl des konkreten Vorgehensmodells stellen sich die kurz umrissenen Entwicklungsphasen in der Praxis als iterativer Prozess dar. Gerade nach vermeintlicher Fertigstellung der Software, also in der Wartungsphase, sind häufig Änderungen am Quelltext notwendig. Diese Änderungen müssen anschließend wieder konsequent in die Designdokumente einfließen - sonst ist bereits nach wenigen Änderungen die Dokumentation der Software nur noch eingeschränkt brauchbar. Weiterentwicklungen auf Basis des ursprünglichen Designs werden unnötig erschwert oder gar unmöglich gemacht.

[1] Computer Aided Software Engineering, vgl. [6]

[2] Als *Refaktorierung* bezeichnet man einen manuellen oder automatischen Umbau des Quelltexts eines Programms, wobei die Programmfunktionalität erhalten bleibt und gleichzeitig Struktur, Lesbarkeit und Wartbarkeit erhöht werden [4].

Abhilfe schafft *Roundtrip Engineering*. Roundtrip Engineering sorgt für die Aktualisierung der in den verschiedenen Phasen des Entwicklungsprozesses angefertigten Dokumente. So werden beispielsweise Quelltext-basierte Refaktorierungen sofort von den zur Codegenerierung verwendeten Designdokumenten reflektiert. Dadurch bleiben Implementierung und Design konsistent. Roundtrip Engineering besteht aus zwei Phasen. Als *Forward Engineering* wird die Phase beschrieben, in der Änderungen in Dokumenten einer höheren Abstraktionsebene auf Dokumente einer niedrigeren Abstraktionsebene angewendet werden. Verändert man beispielsweise die Dokumente des Feindesigns, wird anschließend der Quelltext durch erneute Codegenerierung aktualisiert. Die Rückrichtung bezeichnet man schließlich als *Reverse Engineering*. Hier werden umgekehrt Änderungen der niedrigeren Abstraktionsebene auf die darüberliegenden Ebenen angewendet. Insbesondere führen Änderungen am Quelltext zur automatischen Anpassung der Designdokumente. Die vorliegende Arbeit stellt ein Verfahren zum templatebasierten Reverse Engineering vor.

1.2 Voraussetzungen

1.2.1 Fujaba

Professor Zündorf beschreibt in [11] die Software Entwicklung mittels *Story Driven Modelling* (SDM) mit dem UML CASE-Tool Fujaba. Fujaba SDM ermöglicht dem Entwickler, die zu erstellende Software vollständig in UML zu modellieren - vom Grobdesign auf *Use Case*-Ebene bis zum Feindesign bei der Implementierung der Methoden. Der nach Wahl der Zielsprache (meist Java) generierte Quelltext ist kompilierbar und ausführbar. Fujaba unterstützt zur Codegenerierung UML Klassen-, Aktivitäts- und Zustandsdiagramme. Im Folgenden soll die Vorgehensweise bei der Anwendung von SDM kurz erläutert werden. Abbildung 1.1 illustriert den beschriebenen Ablauf.

1. **Use Cases:** Zunächst erstellt man für das zu lösende Anwendungsproblem im Rahmen einer Anforderungsanalyse mehrere *Use Cases*. Ein Use Case zielt dabei stets auf die Erfüllung eines einzelnen Tasks ab. Gewöhnlich sind mehrere Use Cases zur Abdeckung der gesamten Funktionalität der zu entwickelnden Software notwendig.

2. **Story Boarding:** In der nächsten Phase werden für jeden Use Case mehrere kon-

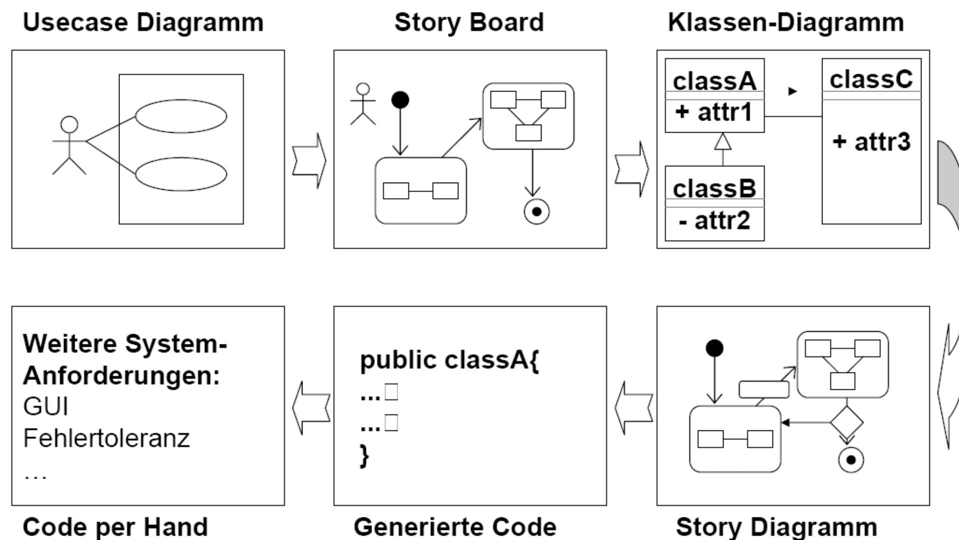

Abbildung 1.1: *FUJABA Story Driven Modeling*

krete Anwendungsfälle angelegt. Dieser Prozess wird als *Story Boarding* bezeichnet. *Story Boards* werden später zur automatischen Testgenerierung verwendet (*Test-First*-Paradigma).

3. **Klassendiagramm:** Aus den erstellten Story Boards leitet Fujaba automatisch ein UML Klassendiagramm ab, welches dann vom Entwickler angepasst wird (Einbau von Generalisierung, Festlegung der Kardinalitäten der Assoziationen, etc.). Änderungen am Klassendiagramm werden stets sofort in den betroffenen Story Boards reflektiert.

4. **Story Diagramme:** Nun kann der Entwickler die in den Story Boards geplanten Methoden implementieren. Dazu sieht Fujaba SDM wieder eine Mischform aus UML Kolloborations-, Sequenz,- und Aktivitätsdiagrammen vor.

5. **Tests:** Nach der Implementierung der Methoden können die aus den Story Boards automatisch generierten Tests durchgeführt werden. Bei Fehlern erlaubt es SDM, die obigen Prozessphasen in beliebiger Reihenfolge erneut zu durchlaufen, um schließlich die Tests zu bestehen.

6. **Codegenerierung:** Nach erfolgreichem Ausführen der Tests kann nun Quelltext erzeugt werden. Die dabei eingesetzte templatebasierte Codegenerierung (1.2.2) ist

zunächst programmiersprachenunabhängig. Momentan sind jedoch nur Templates für Java und EMF vorhanden.

Der Prozess wird iterativ angewendet - jede Phase kann nach Notwendigkeit wiederholt und dabei in beliebiger Reihenfolge angewendet werden. Änderungen in einer Phase schlagen sich in den anderen Phasen nieder - nur Änderungen am generierten Quelltext nicht. Obwohl der Name (*FUJABA: From UML to Java And Back Again*) auch das Roundtrip Engineering impliziert, ist dies derzeit nicht möglich. Mit Hilfe der vorliegenden Arbeit soll an dieser Stelle Abhilfe geschaffen werden, so dass generierter Quelltext wieder eingelesen werden kann und dabei Änderungen erhalten bleiben: Sowohl Klassendiagramme, Storydiagramme und Statecharts sollen aktualisiert werden.

Im Rahmen der Fujaba Tool-Suite sind zahlreiche Projekte des Fachgebiets Software Engineering der Universität Kassel[3] angesiedelt. Als besondere Voraussetzung für den Ansatz der vorliegenden Arbeit ist hier die Codegenerierung CodeGen2 [3] (siehe Abschnitt 1.2.2) hervorzuheben.

1.2.2 CodeGen2

In [3] beschreiben Geiger, Schneider und Reckord eine erfolgreiche Implementierung templatebasierter Codegenerierung als Plugin für Fujaba. CodeGen2 wandelt dabei das in Fujaba erstellte Modell der entwickelten Applikation in Quelltextdateien einer Zielsprache um. Als Templates werden Apache Velocity Templates verwendet.

Die Codegenerierung ist in folgende Phasen aufgeteilt (siehe Abbildung 1.2):

1. Aus dem bei der Modellierung der Software aufgebauten *Syntax Graph* in Fujaba werden in der Phase der *Decomposition* Elemente einer Zwischenschicht, sogenannte `Tokens`, erzeugt. Tokens repräsentieren einzelne zu generierende Quelltextfragmente und sind in Form eines Graphen miteinander verlinkt.

2. In der Kernphase der Codegenerierung werden Umstrukturierungen an der Token Schicht vorgenommen, um später aus der komplexen Graphstruktur Programmcode erzeugen zu können. Dazu wird im Laufe dieser Phase der Graph zu einem Baum umgeformt.

[3]Fujaba ist auch an anderen Universitäten Forschungsschwerpunkt; einen Überblick bietet hier [8].

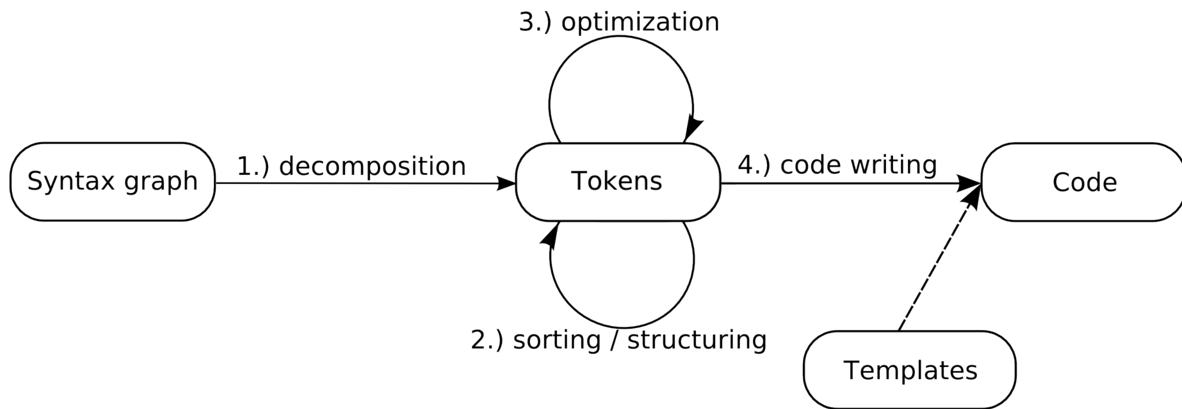

Abbildung 1.2: *Konzept der Codegenerierung von Codegen2 [3]*

3. Die Umstrukturierung der Token Schicht erfolgt desweiteren unter dem Gesichtspunkt möglichst geringer Laufzeit des erzeugten Quelltexts. Da es für ein gegebenes Verhaltensdiagramm mehrere Quelltextalternativen - mit jeweils unterschiedlicher Laufzeit - gibt, wird durch ein Kostenmodell versucht, optimalen Quelltext zu erzeugen.

4. Zur Quelltextgenerierung wird der erstellte Token Baum *post-order* traversiert: Es wird zunächst Code für die untergeordneten Teilbäume erzeugt, dann schließlich Code für die Wurzel selbst (rekursiv). Für die verschiedenen Tokenarten existiert jeweils ein Template, welches von einem `TemplateCodeWriter` geladen wird. Dieser veranlasst dann die Velocity Template Engine mit Hilfe des gewählten Templates und den Daten des Tokens (gekapselt in einem `Context`, einem Container zur Datenhaltung, siehe 1.2.3), das jeweilige Stück Quelltext zu erzeugen. Der von den Kind-Token erzeugte Quelltext wird nun an die Eltern-Token weitergegeben, bis schließlich Quelltext für beispielsweise eine Java-Klasse vollständig ist. Anschließend wird die generierte Klasse als einzelne Datei gespeichert.

   Die Information, welche Templates welches Token erzeugen und welche Token welche Kinder haben, wird später von großer Wichtigkeit sein und muss dem Reverse Engineering Prozess von CodeGen2 zur Verfügung gestellt werden (vgl. Abschnitt 2.4).

Wie bereits erwähnt ist CodeGen2 zunächst programmiersprachenunabhängig und es existieren Templates zur Generierung von Java und EMF. Für weitere Sprachen ist es,

neben der offensichtlichen Notwendigkeit, neue Templates zu entwickeln, teilweise erforderlich, die erstellte Zwischenschicht anders zu organisieren oder andere `CodeWriter` zu verwenden. Beides kann aber relativ einfach durch Manipulation der verwendeten Logikstrukturen erfolgen - beispielsweise durch ein Plugin für CodeGen2 selbst. Änderungen am Kern von CodeGen2 sind dazu nicht erforderlich. Darüberhinaus sind die Informationen über die Struktur der Tokens und das Mapping der Token auf die Templates wichtig für das Reverse Engineering, wie bereits geschildert.

1.2.3 Velocity Template Engine

Das Apache Velocity Project[4] bietet mit Velocity eine Java-basierte Template Engine. Templates sind zunächst Vorlagen, die mit Platzhaltern versehen sind. Diese Platzhalter werden dann, beim Zusammenführen der Templates mit den Daten, mit Werten gefüllt. Besonders häufig trifft man im Bereich der Webentwicklung auf Templates, da auf diese Weise der Programm-Code (bspw. PHP, Perl, etc.) vom Markup (bspw. XHTML) getrennt werden kann. Ziel ist also zunächst eine Trennung zwischen Logik und Darstellung, gemäß *Model-View-Controller* Paradigma.

In der Realität sieht es allerdings so aus, dass Templates selbst Möglichkeiten zum Programmieren geben (müssen). Dadurch erfordert es beim Templateentwickler Disziplin, um nicht sowohl auf Programmiersprachenseite als auch im Template zu programmieren. Gerade Apache Velocity bietet mit der *Velocity Template Language* (VTL) eine ausdrucksstarke Programmiersprache, die es ermöglicht, komplexe Logik in die Templates zu integrieren.

Funktionsweise Velocity

Velocity kann sehr einfach in Java Projekte eingebunden werden. Zur Inbetriebnahme muss Velocity zunächst initialisiert werden. Anschließend wird ein `Context`-Objekt erzeugt. Der Context ist ein Container zur Datenhaltung und dient der Transferierung der Daten zwischen Java-Layer und Template-Layer. Er entspricht einer Hashtabelle, besteht also aus Tupeln von Schlüssel-Wert Paaren. Unter dem *Schlüssel* ist der *Wert* später bei der Zusammenführung mit dem Template adressierbar. Neben einfachen Werten, also bspw. Werten vom Typ `java.lang.String`, können dem Context auch ganze Objekte hinzugefügt werden. Sie sind komplett aus dem Template ansprechbar (siehe Abschnitt

[4]siehe [10]

zu VTL). Nach dem Hinzufügen der Daten zum Context-Objekt wird ein zuvor erstelltes Template ausgewählt und mit den Daten (Context-Objekt) zusammengeführt. Schließlich kann der so generierte Quelltext in einer Datei gespeichert werden.

Velocity Template Language (VTL)

Zum Verständnis des in Kapitel 2.2 vorgestellten Lösungsansatzes ist es erforderlich, kurz auf die einzelnen Sprachkonstrukte der Velocity Template Language einzugehen.

Terminale: Zunächst bestehen Templates aus Terminalen. Diese sind Zeichenketten, die ohne Veränderung direkt in den erstellten Quelltext einfließen. Insbesondere *Whitespaces* (' ', '\t', '\n') werden in den Zieltext übertragen, oft auch mit unerwünschten Nebeneffekten. Entweder man formatiert das Template, dann ist der generierte Text unformatiert - oder man formatiert das Template so, dass der generierte Text formatiert wird, was jedoch die Lesbarkeit des Templates stark beeinträchtigt.

Referenzen: Referenzen werden durch die Zeichenkette, auf die sie in der Hashtabelle des `Context`-Objekts zeigen, ersetzt. Es gibt keine Möglichkeit Sichtbarkeiten zu definieren, daher sind alle Referenzen global. VTL kennt drei Arten von Referenzen: *Variablen*, *Properties* und *Methoden*.

- *Variablen:* Für eine Variable `$foo` gibt es drei Notationen: die normale Notation `$foo`, eine formale Notation `${foo}` sowie eine „ruhige" Notation `$!foo`. Die ersten beiden Notationen sind semantisch gleichwertig. Die Besonderheit an der ruhigen Notation ist, dass, wenn für die Variable kein Wert in der Hashtabelle hinterlegt ist oder dieser Wert `null` ist, eine leere Zeichenkette im generierten Code eingefügt wird. Bei den anderen beiden Notationen würde im generierten Code die Stringrepräsentation der Variablen abgedruckt, also im Beispiel für `$foo` die Zeichenkette `$foo`.
- *Properties:* Properties entsprechen entweder einem Feldzugriff auf das Objekt, also bei `$foo.bar` einen Zugriff auf das Feld `bar`: `$foo.getBar()`, oder - falls das Objekt eine Hashtabelle ist - einen Wertzugriff: `$foo.get(bar)`.
- *Methoden:* Hier wird die entsprechende Methode des Objektes mitsamt Parametern aufgerufen: `$foo.myMethod( arg0, arg1, ..., argN )`.

Ist der Wert der Referenz keine Zeichenkette, sondern ein Objekt, so wird der Rückgabewert der `toString()` Methode in den Quelltext generiert.

Set-Direktiven: Set-Direktiven sind Wertzuweisungen, entweder zu einer Variablen (`#set( $foo = "bar")`) oder zu einem Property (`#set( $foo.bar = "bar")`). Sie bestehen also stets aus einer linken Seite (left-hand-side, LHS) und einer rechten Seite (right-hand-side, RHS). Während die LHS immer eine Variable oder ein Property sein muss, kann auf der RHS eine andere Referenz, ein String, eine Zahl, eine Liste, eine Map oder ein arithmetischer Ausdruck stehen.

If-Abfragen: Ein weiteres von VTL unterstütztes Sprachkonstrukt sind If-Abfragen. Sie bestehen aus einem `#if`, optional gefolgt von mehreren `#elseif` und abschließendem `#else`. Der `#if`- und die `#elseif`-Zweige enthalten jeweils eine Bedingung. Diese ist dann erfüllt, sobald der enthaltene logische Ausdruck zu dem Wahrheitswert `true` ausgewertet werden kann. Der Ausdruck besteht dabei aus Referenzen und logischen Operatoren.

```
#if( !$foo ) ..
#if( $foo && $bar ) ..
#if( $foo || $bar ) ..
#if( $foo == "123" ) .. ## vergleicht den Wert direkt,
#if( $foo == $bar ) ..  ## nicht die Referenz
```

Eine einzelne Referenz wird genau dann zu `false` ausgewertet, wenn sie entweder mit dem Wahrheitswert `false` oder gar nicht belegt ist. In allen anderen Fällen wird sie zu `true` ausgewertet.

Schleifen: In Programmiersprachen gibt es oft mehrere semantisch gleichwertige Varianten von Schleifen. VTL kennt nur eine Art von Schleifen, nämlich das *For-Each*.

```
#foreach( $foo in $bar )
  $foo
#end
```

Hierbei ist `$bar` eine Menge, über die iteriert wird, um für jedes Element der Menge die gleichen Operationen - welche sich im sogenannten Schleifenrumpf befinden - durchzuführen. Unterstützt werden Arrays sowie Subklassen

der Typen `java.util.Collection`, `java.util.Map`, `java.util.Iterator` und `java.util.Enumeration`.

Parse-Direktiven: Velocity erlaubt es ein Template auf mehrere Dateien aufzuteilen, um beispielsweise aus einem „Master"-Template verschiedene andere Templates je nach Bedarf - zum Beispiel innerhalb einer If-Abfrage - einzulesen. Dazu wird die Parse-Direktive verwendet. Als Argument nimmt sie den Namen des einzulesenden Templates entgegen (beispielsweise `#parse('otherTemplate.vm')`). Darüberhinaus ist es auch möglich den Namen des einzulesenden Templates durch eine Referenz erst zur Anwendungszeit zu bestimmen (zum Beispiel `#parse($foo)`). Im Resultat verhält sich ein aus mehreren eingebundenen Templates bestehendes Template wie ein großes. Die eingebundenen Templates verwenden das gleiche `Context`-Objekt wie das umgebene Template, inklusive aller zuvor durch Zuweisungen definierten Variablen.

Kommentare: Es gibt in VTL zwei Arten von Kommentaren. Der einzeilige Kommentar beginnt mit `##`. Mehrzeiliger Kommentar beginnt mit `#*` und endet mit `*#`. Kommentare werden bei der Anwendung des Templates von Velocity ignoriert. Allerdings können sie eine wichtige Rolle übernehmen, denn mit Hilfe der einzeiligen Kommentare kann man die geschilderte Whitespace-Problematik lösen. Indem man nämlich nach einem VTL Ausdruck einen Kommentar beginnt, kann man danach zur Strukturierung auch einen Zeilenumbruch einfügen, ohne dass dieser dann im generierten Text erscheint. Abbildung 1.3 versucht dies zu verdeutlichen.

```
${a}##
b##
c##
```

```
${a}bc
```

Abbildung 1.3: *Formatierung durch Kommentare: Diese beiden Templates erzeugen den gleichen Text.*

Beide Templates erzeugen im Beispiel den gleichen Text, wobei das linke aufgrund der Formatierung besser lesbar ist. Bei den folgenden Beispielen in dieser Arbeit wird dieses Formatierungsproblem der Templates jedoch ausgeklammert, da es für das Reverse Engineering nicht relevant ist und in diesem Fall die bessere Lesbarkeit der Korrektheit vorzuziehen ist.

Makros: Mit Hilfe von *Makros* kann der Templateentwickler eigene Methoden definieren. Dies ist jedoch gerade unter dem zu Beginn dieses Abschnitts geäußerten Gedankengang nicht als wünschenswert zu betrachten, da so unweigerlich viel Programmlogik in das Template integriert wird. Da Makros von der vorliegenden Arbeit nicht unterstützt werden, werden sie hier nun auch nicht näher erläutert.

2 Umsetzung/ Methode

Bisherige Ansätze verwenden zum Reverse Engineering eines Quelltextes einen auf der Grammatik der Programmiersprache basierenden Parser. Für Java gibt es schon viele Parser und Grammatiken - beispielsweise im Lieferumfang von JavaCC[1]. Nachteil dieser Herangehensweise wäre gewesen, dass man die Sprachunabhängigkeit von CodeGen2 verloren hätte. Man hätte für jede Zielsprache einen eigenen Parser entwickeln müssen - ein sehr hoher zeitlicher Aufwand, der jeweils zur Unterstützung einer weiteren Sprache betrieben werden müsste. Wünschenswert ist also eine stets gleiche Herangehensweise, unabhängig von der Zielsprache. Daher war die Aufgabenstellung für die vorliegende Arbeit die Entwicklung eines Parsers, der das Reverse Engineering mit Hilfe der zur Codegenerierung verwendeten Templates durchführt.

Sobald man nach der Initialisierung der Velocity Template Engine ein Template geladen hat, liegt dieses als Baumstruktur vor. Während des Traversierens dieser Baumstruktur kann man parallel den generierten Quelltext einlesen und so das ursprünglich verwendete `Context`-Objekt rekonstruieren. Zum Traversieren bietet Velocity das *Visitor-Design-Pattern* (siehe [2]) an. Das Visitor-Pattern erlaubt es, eine Objektstruktur zu traversieren, wobei man die auszuführenden Aktionen von den beteiligten Objekten trennt und in einem `Visitor`-Objekt kapselt.

Es stellte sich heraus, dass es anhand des gegebenen Quelltextes oftmals mehrere äquivalente Möglichkeiten zur Belegung einzelner Referenzen des Templates gibt. Dies macht eine einfache lineare Traversierung des Templates nicht möglich, stattdessen müssen teilweise mehrere verschiedene mögliche Lösungen untersucht werden. Dies soll kurz an einem kleinen Beispiel verdeutlicht werden: Angenommen es liegt folgendes Template vor, mit welchem der Quelltext '`ab`' generiert wurde:

[1] Der *Java Compiler Compiler* (JavaCC) ist ein Werkzeug, das einen auf einer angegebenen Grammatik basierenden Parser generiert [9]

```
#if( $foo )
   a
#end
$bar
b
```

Es gibt genau zwei Möglichkeiten zur Belegung der beiden Referenzen `$foo` und `$bar`:

```
$foo := false
$bar := 'a'
Terminal: b
```

und

```
$foo := true
Terminal: a
$bar := ''
Terminal: b
```

Offensichtlich sind beide Belegungen richtig. Womöglich wird man erst nach Abschluss des Parsens, beim *Reasoning* (siehe Abschnitt 2.3) herausfinden können, welche Belegung korrekt ist und welche verworfen werden muss. Mit dem einmaligen, linearen Durchlauf des Templates ist es also nicht möglich alle möglichen Belegungen für die verwendeten Referenzen zu finden. Sobald anhand des Quelltexts nicht entscheidbar ist, welche der gegebenen Möglichkeiten die richtige ist, müssen alle Alternativen geprüft werden: es wird vermutlich oftmals zunächst mehrere richtige Lösungen für ein Template geben. Die hier beschriebene Vorgehensweise wird auch als *Backtracking*[2] bezeichnet. Mit diesem Problem hätte eine leicht modifizierte Version des Visitor-Patterns umgehen können, indem man es ermöglicht `Visitor`-Objekte zu kopieren, um dann jeweils die Alternativen zu prüfen.

Desweiteren stellte sich jedoch heraus, dass es teilweise notwendig ist, die lineare Traversierung des Templates abzubrechen, um sie an einem anderen Knoten fortzusetzen.

[2] *Backtracking* bezeichnet ein Vorgehen mit systematischem Ausprobieren: Zunächst wird ein Lösungsweg so lange verfolgt, bis er scheitert. Dann werden einzelne Schritte rückgängig gemacht und alternative Lösungswege ausprobiert. Dies geschieht rekursiv. Backtracking kann sehr schnell zu Laufzeitproblemen führen, da notfalls alle möglichen Wege ausprobiert werden (vgl. Abschnitt 3.1).

Dies ist nicht nur abhängig von der Art des Knotens der Baumstruktur (vom VTL Sprachkonstrukt), sondern auch von den Vorfahren des Knotens (seinem eigenen Kontext). Dies soll an folgendem Beispiel verdeutlicht werden:

```
1   a
2   #foreach( $foo in $bar )
3      $foo;
4   #end
5   $something
```

Nach dem Einlesen des Terminals 'a' (1) verarbeitet der Parser die Schleife (2). Da die Anzahl der Elemente in der Menge `$bar` nicht bekannt ist, kann der Parser nicht im Vorhinein bestimmen wie häufig der Schleifenrumpf (3) auf den Quelltext passen muss. Darüberhinaus reicht es nicht, einfach den Schleifenrumpf so häufig wie möglich anzuwenden, denn der Schleifenrumpf und das Template nach der Schleife könnten potentiell den gleichen Quelltext erzeugt haben (siehe dazu Abschnitt 2.2.5). Deshalb muss der Parser nach jeder Iteration der Schleife auch das restliche Template prüfen. Gelangt der Parser also an das Ende der Schleife, muss er einerseits im Template nach der Schleife anfangen (5) und andererseits eine weitere Iteration der Schleife testen, also im Schleifenrumpf (3) beginnen. Dies ist mit einem Visitor nicht realisierbar.

Aus diesen Überlegungen und ersten Tests entstand im Laufe der Arbeit ein Parser, der die jeweiligen Sprachkonstrukte einzeln und anhand ihres Kontextes verarbeiten kann. Diese Verarbeitungseinheiten sind in einzelnen sogenannten `ParserComponents` gekapselt, gemäß *Strategy-Design-Pattern* (siehe [2]).

Im Rahmen der vorliegenden Arbeit ist ein Plugin für Fujaba zur Realisierung des in Abschnitt 1.1 skizzierten Roundtrip Engineering entstanden. Die Hauptkomponente des Plugins ist der *Parser*, der die Belegung der Templates anhand des Quelltexts rekonstruiert. Desweiteren enthält das Plugin drei weitere Komponenten: einen Mechanismus zum Entfernen falscher Resultate und zur Zerlegung bool'scher Ausdrücke (*Reasoner*), ein Modul zur Erzeugung und Bereinigung der Token-Zwischenschicht (*ReverseTokenCreator*), sowie einer Komponente zur Erzeugung bzw. Aktualisierung des Fujaba-Metamodells[3] (*ModelUpdater*). Das Plugin wurde dabei größtenteils mit Fujaba

[3] Ein *Metamodell* ist das Modell eines Modells. In Fujaba wird eine Software modelliert - es wird also ein Modell erstellt. Dieses Modell wird in einer eigenen Datenstruktur repräsentiert: dem Metamodell.

selbst realisiert - nur der *ModelUpdater* und zur Qualitätssicherung erstellte *JUnit*-Tests wurden manuell mit Eclipse implementiert.

Im folgenden wird zunächst das Funktionsprinzip der Gesamtlösung erläutert, im Anschluss werden die einzelnen Komponenten detailliert vorgestellt.

2.1 Funktionsprinzip

Zunächst wird die generierte Quelltext-Datei eingelesen und es werden alle eventuell vorhandenen *Carriage Return*-Zeichen ('\r') entfernt. Dann wird die Velocity Template Engine initialisiert und die in Frage kommenden Templates eingelesen[4]. Jedes Template wird nun intern als Baumstruktur repräsentiert. Zur optimalen Verwendung in Fujaba ist es notwendig, dass die Knoten der Bäume *next*-, *previous*-, *parent*- und *children*-Assoziationen enthalten, damit die Baumstruktur je nach Notwendigkeit traversiert werden kann. Diese Assoziationen enthält die von Velocity verwendete Baumstruktur jedoch nicht, deshalb ist eine Kapselung durch eine eigene Knotenschicht notwendig.

Anschließend beginnt das Parsen: Es wird mit dem Wurzelknoten des Starttemplates[5] angefangen und der Lese-Zeiger auf den Anfang des Quelltexts gestellt. Abhängig vom Typ des vom Wurzelknoten gekapselten Velocity Knotens wird eine `ParserComponent` des Parsers mit Hilfe des Strategy-Patterns gewählt. Diese verarbeitet den Knoten und den Quelltext an der Leseposition. Dabei werden mögliche Belegungen gespeichert und weitere Alternativen vorgemerkt. Dies wird solange wiederholt, bis das gesamte Template abgearbeitet ist.

Wie in Abschnitt 1.2.2 über CodeGen2 erläutert, sind die zur Codegenerierung verwendeten Templates auf mehrere Dateien aufgeteilt: Pro Token der Zwischenschicht ein eigenes Template. Daher wird nun mit den gefundenen Belegungen derjenigen Variablen fortgefahren, die ihrerseits aus anderen Templates generiert ist. Die Information, welche Variablen dies sind, stellt CodeGen2 zur Verfügung. Wenn nun eine Belegung einer solchen Variablen nicht auf ein in Frage kommendes Template passt, so ist das gesamte Resultat nicht gültig und wird verworfen.

Nach Abschluss des Parsens müssen die gefundenen Resultate auf ihre Korrektheit geprüft werden. Da der Parser oft mehrere mögliche Ergebnisse für die Belegung eines

[4] Die Information, welche dies genau sind, wird von CodeGen2 zur Verfügung gestellt (vgl. Abschnitt 1.2.2).

[5] Das Starttemplate wird ebenfalls von CodeGen2 benannt.

Templates findet, können nun ungültige entfernt werden. Daher wird nun, im Rahmen eines einfachen Reasonings, ein Plausibilitätscheck durchgeführt. Zusammengesetzte logische Ausdrücke (Bedingungen von If-Abfragen, beispielsweise `$foo && $bar == true`) werden nach Möglichkeit aufgelöst (im Beispiel zu `$foo := true` und `$bar := true`) und die einzelnen Ausdrücke mit den gefundenen Belegungen der Referenzen verglichen. Enthält ein Resultat nun sich widersprechende Belegungen für eine Referenz, also beispielsweise `$foo == true` und `$foo == false`, so wird dieses Resultat verworfen. Soweit möglich geschieht dies auch schon während des Parsens, allerdings kann man gerade komplexe logische Ausdrücke oft erst nach Abschluss des Parsens zerlegen. Auf diese Weise wird im Idealfall die Anzahl der möglichen Lösungen auf eine Lösung reduziert.

Da CodeGen2 die Information enthält, welches Template aus welchem Token der Tokenschicht stammt, kann nun rückwärts aus der gefundenen Belegung der Templates das betroffene Token rekonstruiert werden (siehe Abschnitt 2.4) und dann wiederum ein Mapping auf das Fujaba-Metamodell durchgeführt werden (siehe Abschnitt 2.5). Damit kann dann das bestehende Modell des aktuell aktiven Projekts aktualisiert werden. Eine generische Lösung für diesen Vorgang zu implementieren ist jedoch schwierig und war nicht Teil der Aufgabe. Je nach Tokenart sind verschiedene Suchstrategien zum Finden verwendeter Nachbarn notwendig, denn letztendlich liefert der Parser nur Tupel bestehend aus Attributen und Zeichenketten. Um jedoch den Nutzen der vorliegenden Arbeit zu visualisieren und das Reverse Engineering zu komplettieren wurde eine einfache Logik zur Rekonstruktion des Modells in Fujaba implementiert.

2.2 Parser

Aus den Vorüberlegungen zu Beginn von Kapital 2 ergab sich die Notwendigkeit, *Backtracking* zu implementieren. Dazu ist es erforderlich, bei mehreren Möglichkeiten jede Alternative zu vermerken, um sie anschließend zu prüfen. Auf diese Weise werden dann alle gültigen Belegungen für die Referenzen des Templates gefunden. Dazu hat der Parser einen Stack, auf den alle noch abzuarbeitenden Alternativen abgelegt werden. Diese „Alternativen“ werden durch ein Objekt vom Typ `ParserResult` gekapselt und unterscheiden sich voneinander zunächst nur durch eine andere Belegung einer Referenz, sowie einer anderen Position des Lesezeigers im Quelltext. Zuvor gefundene Belegungen stimmen überein. Als Datenstruktur bietet sich daher ein Baum an: Zunächst gibt es nur ein einziges `ParserResult`. Dieses kapselt den Zustand des Parsers explizit; es enthält

also beispielsweise einen Zeiger auf den aktuellen Knoten im Template, die Leseposition im Quelltext sowie bereits gefundene Belegungen für Referenzen. Stößt der Parser auf eine Situation, in der es Alternativen für die weitere Vorgehensweise gibt, erzeugt er entsprechend viele neue `ParserResults` und fügt sie dem aktuellen `ParserResult` als Kinder hinzu. Handelt es sich beispielsweise um zwei mögliche Belegungen für eine Referenz, so werden diese auf zwei neue `ParserResults` aufgeteilt, die beiden Kinder auf den Stack gelegt und das Parsen mit dem aktuellen `ParserResult` beendet. Es geht weiter mit dem obersten `ParserResult` des Stacks, welches Informationen darüber enthält, mit welchem Knoten des Templates und an welcher Leseposition im Quelltext fortzufahren ist. Auf diese Weise wird sichergestellt, dass alle möglichen Alternativen zur Belegung der Referenzen gemäß vorliegendem Quelltext getestet werden.

Die Velocity Template Language (VTL) enthält mehrere Sprachkonstrukte, die in Kapitel 1.2.3 vorgestellt wurden. Beim Traversieren des Template Baums muss somit jeder Knoten gemäß repräsentiertem Sprachkonstrukt interpretiert und verarbeitet werden. Dazu bietet sich klassischerweise das *Strategy-Design-Pattern* (siehe [2]) an. Jede *Strategy* kapselt dabei die Funktionalität, die zur Verarbeitung des jeweiligen VTL-Konstrukts notwendig ist und enthält die Information, mit welchem Knoten fortzufahren ist. Die Entscheidung, welche *Strategy* zu verwenden ist, erfolgt durch Typprüfung des Template Knotens.

2.2.1 Referenzen

Das erste zu behandelnde Sprachkonstrukt sind Referenzen. Wie in Abschnitt 1.2.3 erläutert kennt VTL drei Arten von Referenzen: Variablen, Properties und Methoden. Sie sind Platzhalter im Template und werden von CodeGen2 mit Werten (entweder einer Zeichenkette oder einem Java-Objekt) im `Context`-Objekt abgelegt. Stößt man während des Parsens auf eine Referenz, so kann man sie zunächst nicht belegen. Eine Wertzuweisung ist erst möglich, sobald man ein Terminal im Template antrifft, sodass man dann den Zwischenraum zwischen letztem und nächstem Terminal der Referenz zuordnen kann. Daraus folgt auch, dass eine Unterscheidung zwischen Variablen, Properties und Methodenaufrufen (vgl. 1.2.3) zunächst nicht notwendig ist. Somit werden alle Referenzen gleich behandelt. Dabei ist es nicht wünschenswert, mit vielen Stringvergleichen arbeiten zu müssen, sondern eleganter, die Referenzen zu kapseln. Dies erfolgt durch `Expression` Objekte, die alle auftretenden Referenzen, aber auch zusammen-

gesetzte Ausdrücke (wie boolesche Formulierungen: `$a && $b`) kapseln. Da es später beim Reasoning (vgl. 2.3) notwendig wird, zusammengesetzte Ausdrücke wieder in ihre Bestandteile zu zerlegen, wird die dahinterliegende Datenstruktur (also der betroffene Knoten des Template Baums) vermerkt.

Da man einerseits nicht ständig neue Objekte anlegen will und andererseits verschiedene Repräsentationen der gleichen `Expression`[6] in VTL möglich sind, erfolgt der Zugriff auf die `Expression`-Objekte über eine `ExpressionFactory`. Diese ist zentral für die Erstellung von `Expression`-Objekten zuständig und merkt sich bereits erstellte Objekte in einer Hashtabelle, gemäß *Flyweight-Design-Pattern* (siehe [2]). Auf diese Weise ist auch der Objektvergleich (via ==) möglich. Vor der Verarbeitung eines übergebenen (String-) Ausdrucks wandelt die `ExpressionFactory` den Ausdruck in eine einheitlich Notation um. Dabei werden Variablen in die formale Notation transformiert (also aus `$foo` wird `${foo}`) und die ruhige Notation gänzlich entfernt (aus `$!foo` wird `${foo}`). Die Überlegung hierbei ist, dass die ruhige Notation eigentlich immer erwünscht ist, denn man will nie `$foo` im generierten Quelltext stehen haben. Daher kann man alle drei Notationen auch gleich behandeln und braucht an dieser Stelle nicht zu unterscheiden. Passiert es doch, dass der Quelltext `$foo` der Variablen `$foo` zugeordnet wird, so wird der Fall als Fehler des Templatedesigners gesehen und `$foo` der Wert `null` zugeordnet.

Da also beim Antreffen einer Referenz zunächst keine Aktion möglich ist, wird sich die Referenz zunächst nur im `ParserResult` gemerkt.

2.2.2 Terminale

Terminale sind in der Templatesprache VTL selbst weniger interessant, denn sie werden schlicht ohne Veränderung in das Zieldokument übernommen. Gerade aber dieses Merkmal macht sie für das templatebasierte Reverse Engineering unersetzlich: Durch sie kann man die Belegung zuvor gefundener Referenzen festlegen.

Stößt der Parser auf ein Terminal, so gibt es genau zwei Möglichkeiten: Im ersten, einfachen Fall wurde als letztes Element ebenfalls ein Terminal gefunden, oder das Parsen wurde gerade begonnen. Dann muss das Terminal an der aktuellen Leseposition im Quelltext passen. Wenn es nicht passt, ist das aktuelle `ParserResult` ungültig und kann entfernt werden. Darüberhinaus kann der Vater des aktuellen `ParserResult` ebenfalls entfernt werden, wenn er keine weiteren Kinder hat. Dieser Löschvorgang wird rekursiv,

[6] Also verschiedener Ausdrücke mit gleicher Semantik, bspw. `$foo` und `${foo}`

ggf. bis zur Wurzel, angewendet. Stimmt das Terminal hingegen mit dem Quelltext an der aktuellen Leseposition überein, kann mit dem Parsen fortgefahren werden.

Der andere, komplexere Fall ist, wenn zuvor Referenzen gefunden wurden. Angenommen, das Template besteht aus `${foo}*` bei gegebenem Quelltext '`aa*bb*`'. Nun müssen alle Vorkommnisse des Terminals ('`*`') im Quelltext ab der gegenwärtigen Leseposition untersucht werden. Existiert für die zu belegende Referenz (`${foo}`) noch keine Belegung, muss für alle Möglichkeiten (hier sind es zwei: `${foo} := 'aa'` sowie `${foo} := 'aa*bb'`) ein neues `ParserResult` erzeugt werden. Existiert jedoch schon eine Belegung für die Referenz und ist diese nicht als veränderlich markiert (siehe Abschnitt 2.2.3), kann getestet werden, ob die gefundene Belegung mit der alten übereinstimmt. Wenn nicht, so braucht folglich kein neues `ParserResult` angelegt werden.

Gerade bei kurzen, häufig vorkommenden Terminalen wie beispielsweise Leerzeichen oder Zeilenumbrüchen, läuft man Gefahr Laufzeitprobleme zu bekommen (vgl. Abschnitt 3.1). Angenommen man hat einen aus 600 Zeilen bestehenden Quelltext. In jeder Zeile des Templates steht einmal ein Zeilenumbruch, unter ungünstigen Umständen umgeben von zwei Referenzen. Wenn nun stets alle Vorkommnisse des Zeilenumbruchs im Quelltext ab der aktuellen Leseposition (599, 598, ...) untersucht werden müssten, würde die Laufzeit ins unermessliche steigen. Betrachtet man zur Codegenerierung eingesetzte Templates, fällt auf, dass im Regelfall einzelne Referenzen nur wenige Zeichen Quelltext abdecken und meistens nicht über einen Zeilenumbruch hinaus weitergehen. Nur Referenzen, die den aus anderen Templates generierten Quelltext enthalten (vgl. Abschnitt 1.2.2), enthalten auch Zeilenumbrüche. Daher erscheint es sinnvoll, zunächst davon auszugehen, dass Referenzen keine Zeilenumbrüche enthalten. Die Information, welche Referenzen von anderen Templates belegt sind, muss ja ohnehin von CodeGen2 stammen - deshalb kann CodeGen2 auch die Information übergeben, welche Referenzen über mehrere Zeilen gehen. Aus diesen Überlegungen wurden *Constraints* eingeführt, die für einzelne `Expression`-Objekte angegeben werden können. Per Default gilt nun, dass Referenzen keine Zeilenumbrüche enthalten (sogenanntes `SingleLineConstraint`). Dadurch kann eine Explosion des Suchraums vermieden werden.

Ein weiterer, gesondert zu behandelnder Fall tritt ein, wenn einem Terminal eine Folge von Referenzen vorangeht (bspw.: `${var1}${var2}...${varN}*` mit Quelltext '`aaaaa*`'). Problematisch daran ist, dass es viele Möglichkeiten gibt, die Zeichenkette auf die Folge von Referenzen aufzuteilen. Eine Auflösung ist nur dann möglich, wenn

bestimmte beteiligte Referenzen schon belegt sind, also im Idealfall mindestens jede zweite. Aber auch dies funktioniert nur eindeutig, wenn die belegten Referenzen eindeutig einem Substring zuordbar sind. Also könnte man - prinzipiell - stets alle möglichen Zerlegungen in separaten `ParserResults` anfügen. Dies würde allerdings unnötige Laufzeitkosten (vgl. Abschnitt 3.1)verursachen, denn das weitere Parsen nach dem Terminal ist bei allen Alternativen absolut identisch. Daher kommt diese Lösung nicht in Frage. Stattdessen erscheint es sinnvoller, zunächst nur die Konkatenation mit dem Gesamtstring zu belegen (im Beispiel also mit '`aaaaa`'). Desweiteren haben Tests gezeigt, dass die Auflösung dieser sogenannten `ConcatenatedExpression` sich erst später beim Reasoning (vgl. Abschnitt 2.3) lohnt, weil dann dadurch, dass alle belegbaren Referenzen auch belegt sind, die Erfolgswahrscheinlichkeit zum Belegen der Elemente der Konkatenation stark gestiegen ist. Leider ist es manchmal trotzdem nicht möglich, eine Konkatenation erfolgreich aufzulösen - dies stellt somit eine Anforderung an den Templateentwickler, der solche Konstrukte möglichst vermeiden sollte (vgl. Abschnitt 3.2).

2.2.3 Set-Direktiven

Referenzen sind innerhalb eines Templates global, da es keine Möglichkeiten gibt, im Velocity `Context`-Objekt Sichtbarkeiten zu modellieren. Außerdem nimmt man zunächst an, dass das `Context`-Objekt während der Codegenerierung beim Zusammenführen mit dem Template nicht durch Logik innerhalb des Templates verändert wird. Durch diese Einschränkung profitiert man sehr bei der Verwerfung von Belegungen für Referenzen (vgl. 2.2.2). Initialisierungen durch Set-Direktiven liegen zunächst quer zu dieser Annahme, denn sie erlauben es, Referenzen im `Context`-Objekt zu verändern. Daher wäre es aus Sicht der vorliegenden Arbeit logisch, diese schlichtweg zu verbieten. Auch für Velocity selbst wäre es konsequenterweise richtig, Zuweisungen nicht zu gestatten, denn man sollte - gemäß *Model-View-Controller* Paradigma - Programm-Logik und Text-Markup voneinander trennen. Allerdings werden Zuweisungen auch gerne aus Gründen der Lesbarkeit und zur Optimierung vom Templateentwickler eingesetzt. *Lokale Variablen* kapseln oftmals komplexe Operationen, die dann nur einmal berechnet werden müssen. Somit sind Zuweisungen in Templates notwendig und der Parser muss sinnvoll mit ihnen umgehen können.

Wird eine Referenz auf der linken Seite einer Zuweisung verwendet, obwohl sie bereits im Template verwendet wurde, widerspricht dies der Annahme, das das `Context`-Objekt

nicht verändert wird. Mit der Belegung der betroffenen Referenz kann im Folgenden, gerade beim Reasoning (vgl. Abschnitt 2.3) nicht mehr sinnvoll umgegangen werden: Angenommen, eine Referenz wird zunächst mit dem Wahrheitswert `true` belegt. Durch eine Zuweisung wird ihr später der Wert `false` zugeordnet. Für das Reasoning ist sie nun unbrauchbar; ebenso zum Ausschließen von Alternativen bei der Belegung von Referenzen (vgl. Abschnitt 2.2.2). Die Referenz muss folglich als *veränderlich* markiert werden und darf bei späteren Plausibilitätschecks nicht mehr zur Verwendung kommen. Wird die Zuweisung hingegen nur zur Initialisierung einer lokalen Variablen verwendet, kann man sie problemlos verwenden.

Um später die Tokens zu erstellen (vgl. Abschnitt 2.4) ist es erforderlich das `Context`-Objekt zu rekonstruieren. Dazu muss die Zuweisung, die die lokale Variable initialisiert hat, umgekehrt werden. Dies geschieht im Rahmen des *Reasonings* und wird in Abschnitt 2.3 erläutert. Zur Parsezeit reicht es zunächst, die Zuweisung zu notieren.

2.2.4 If-Abfragen

If-Abfragen dienen der Programmablaufsteuerung. Sie werden eingesetzt, um dynamisch - also je nach Belegung der in der Bedingung enthaltenen Variablen - im Programm zu verzweigen. Ähnlich werden sie auch in einem Template eingesetzt. Derjenige Zweig der If-Abfrage, dessen Bedingung erfüllt ist, wird ausgeführt. Beim Reverse Engineering hat man nun das Problem, oftmals noch nicht zu wissen, ob die Bedingung eines If-Zweiges erfüllt ist oder nicht.

In Abbildung 2.1 ist eine If-Abfrage mit ihrer Laufzeitrepräsentation illustriert. Angenommen, die Belegung von `$foo` ist noch nicht bekannt. Dann ist es erforderlich jeden Zweig einzeln zu prüfen. Selbst wenn ein Zweig für sich genommen passt, heißt dies noch nicht, dass dieser Pfad eine richtige Lösung darstellt. Zusätzlich muss das Parsen nämlich ab dem Ende der If-Abfrage (Zeile acht im Beispiel) fortgesetzt werden - beim Parsen jedes Zweiges.

Besonders beachtet werden müssen If-Abfragen, die keinen Else-Zweig enthalten. Einerseits muss jeder If-Zweig (bzw. ElseIf-Zweig) getestet werden. Andererseits darf nicht vergessen werden, den Weg zu untersuchen, in welchem das komplette If nicht passt, also alle Bedingungen zu `false` ausgewertet wurden.

Die Bedingungen der einzelnen Zweige kann man sich zu Nutze machen - allerdings nicht nur die Bedingung des aktiven Zweiges. Da im normalen Programmverlauf der erste

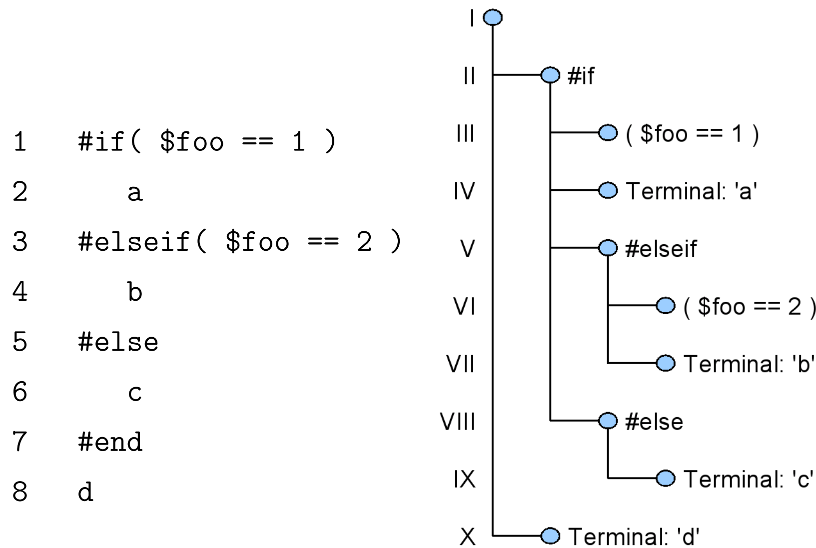

Abbildung 2.1: *Illustration einer If-Abfrage: in VTL (links) und als Baum zur Laufzeit (rechts)*

Zweig angewendet wird, dessen Bedingung erfüllt ist, kann man im Gegenzug verwenden, dass bei inneren Zweigen die Bedingungen der Vorgänger unerfüllt sein mussten.

If-Abfragen werden von der `IfStatementParserComponent` bearbeitet. Diese Parserkomponente hat ihrerseits wieder für jeden Zweig-Typ (If, elseif und else) eine Unterkomponente. Beim Bearbeiten der Zweige durch die entsprechende Unterkomponente wird jeweils ein neues `ParserResult` angelegt, als Startknoten der Inhaltsknoten (im Beispiel Knoten IV, VII oder IX) gesetzt und auf den Stack des Parsers gelegt. Dadurch wird ganz automatisch - sobald der Parser das `ParserResult` vom Stack zur Verarbeitung nimmt - der entsprechende Zweig geparst und das Parsen anschließend am Ende der If-Abfrage fortgesetzt. Dies funktioniert deshalb, weil der Parser nur eine Parserkomponente für das If-Statement hat, nicht aber für elseif oder else. Diese werden somit schlichtweg übersprungen. Darüberhinaus wird sich die Bedingung des aktiven Zweiges gemerkt, sowie - falls vorhanden - die Bedingungen der vorangegangenen Zweige (negiert). Diese Informationen sind wertvoll für das Reasoning (vgl. Abschnitt 2.3), denn man gewinnt aus einer If-Abfrage u.U. viele Belegungen für Referenzen.

Zur Codegenerierung erstellte Templates enthalten häufig If-Abfragen, deren Inhaltsknoten nur aus Zuweisungen bestehen.

Würde im Beispiel der Abbildung 2.2 oben skizzierter Algorithmus zum Einsatz kommen, würden zwei `ParserResult` erzeugt: Eines, das bei Knoten IV beginnt, und ein anderes, das bei Knoten V anfangen würde. Am Ende des Parsens würden sie sich in-

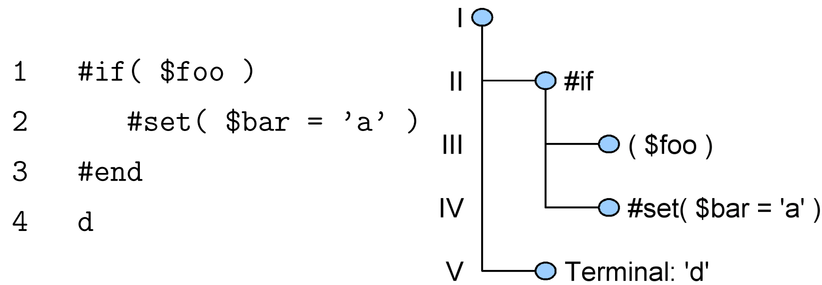

Abbildung 2.2: *If-Abfrage mit `#set`-Direktive: in VTL (links) und als Baum zur Laufzeit (rechts)*

haltlich nur um die Zuweisung aus Zeile zwei unterscheiden, die allerdings erst beim Reasoning (vgl. Abschnitt 2.3) Verwendung finden wird. Für diese Information muss also zweimal der komplette Parsevorgang durchgeführt werden. Dies ist zu kostenintensiv. Schneller wäre es, man würde bei der Zuweisung vermerken, dass sie nur unter einer Bedingung gilt (nämlich wenn `$foo` belegt oder wahr ist) und das Parsen nur einmal durchführen.

Dazu wird beim Betreten eines If-Zweiges zunächst untersucht, ob der Zweig Terminale oder Referenzen enthält. Enthält er diese nicht, wird der für den Teilbaum verwendete Parser umgebaut. Die nach dem oben skizzierten Algorithmus funktionierenden Strategien zur Bearbeitung von If-Zweigen werden durch andere ausgetauscht, welche den jeweiligen Zweig nur einmal traversieren. Beim Verarbeiten einer Zuweisung werden nun die Bedingungen, unter denen sie verwendet wird, mit ihr verknüpft gespeichert. Beim Reasoning (vgl. Abschnitt 2.3) werden diese Bedingungen dann ausgewertet um festzustellen, ob die Zuweisung gültig ist oder nicht.

Tests haben gezeigt, dass diese Vorgehensweise das Parsen immens beschleunigt. Denn bei vielen Templates finden sich mehrere Zuweisungen innerhalb einer If-Abfrage - gerade am Anfang des Templates - so dass das Parsen ohne die Optimierung mehrmals wiederholt werden muss.

Eine weitere mögliche Optimierung ist es, Bedingungen wenn möglich bereits zur Parsezeit auszuwerten, um sich das Parsen nicht in Frage kommender Zweige zu ersparen. Dies wurde realisiert, indem die für das Reasoning verwendete Logik mitverwendet wird (vgl. Abschnitt 2.3). Die in der Bedingung enthaltene `Expression` wird dazu in ihre Komponenten zerlegt. Besteht sie beispielsweise aus dem logischen Ausdruck `$foo && $bar`, wird zunächst `$foo` und dann `$bar` ausgewertet. Wird eine der beiden Referenzen zu `false` ausgewertet, kann der gesamte Zweig ignoriert werden. Die Auswertung der Bedingungen spielt in der Laufzeit gegenüber der Ersparnis, wenn eine

Bedingung erfolgreich ausgewertet werden kann, nur eine untergeordnete Rolle und stellt daher eine echte Optimierung des Laufzeitverhaltens dar (vgl. Abschnitt 3.1).

2.2.5 Schleifen

Schleifen werden verwendet, um mit allen Elementen einer Menge die gleichen Operationen durchzuführen. Ausgehend vom generierten Quelltext ist es dann jedoch schwierig, diese Menge zu rekonstruieren.

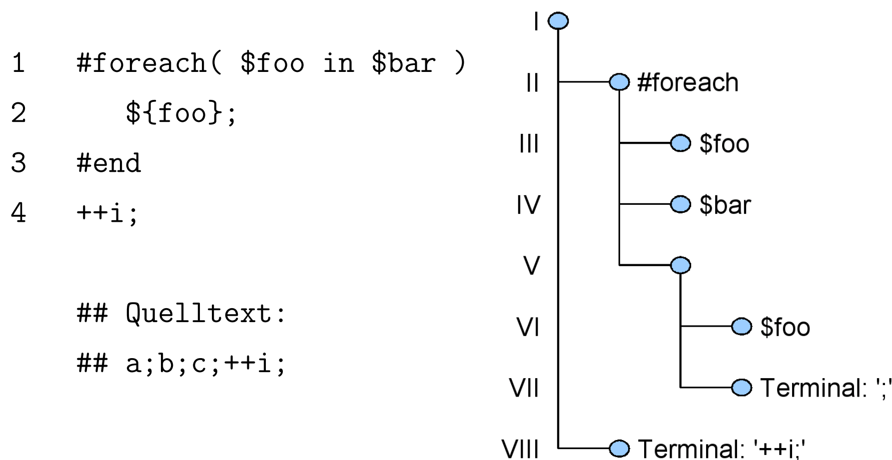

Abbildung 2.3: `#foreach`-*Loop: in VTL (links) und als Baum zur Laufzeit (rechts)*

Abbildung 2.3 zeigt ein einfaches Beispiel einer Schleife. Zunächst ist die Anzahl der Elemente in `$bar` unbekannt. Nun könnte man damit beginnen, den Schleifenrumpf (Knoten V) so häufig zu durchlaufen, bis er nicht mehr passt, um danach am Schleifenende (Knoten VIII) fortzufahren. Allerdings begegnet man schon im Beispiel einem Problem: `$foo` wird der Reihe nach mit '`a`', '`b`', '`c`' und schließlich '`++i`' belegt, obwohl '`++i`' bereits das Terminal nach Schleifenende sein sollte. Der Parser würde dann feststellen, dass der Quelltext komplett geparst wurde obwohl das Template noch nicht durchgearbeitet wurde und somit das Ergebnis verwerfen. Mit diesem einfachen Ansatz würde man also im Beispiel zu keinem Ergebnis kommen.

Es ist also erforderlich, *vor* jeder Iteration über den Schleifenrumpf zwei Alternativen zu prüfen: Erstens die weitere Iteration sowie zweitens die Schleifentermination und das Fortfahren nach dem Schleifenende. Auf diese Weise hat man auch den Fall abgedeckt, wenn die Menge `$bar` keine Elemente enthielt.

Nun fehlt noch eine Abbruchbedingung, denn der Algorithmus terminiert nicht: es werden stets neue `ParserResult` angelegt, die den Schleifenrumpf prüfen. Um dies zu

vermeiden untersucht man, ob beim *letzten* Schleifendurchlauf auch Terminale gefunden wurden. Wurden keine gefunden, so darf die Schleife nicht aufs Neue durchgearbeitet werden.

Enthält der Schleifenrumpf keine Terminale, wird das Reverse Engineering sehr schwierig. Enthält der Rumpf Referenzen (im Minimalfall nur die Iteratorvariable), kann man keine Abbruchbedingung definieren: Da eine Referenz zunächst immer „passt“, würde eine `ConcatenatedExpression` mit endlos vielen Mitgliedern entstehen. Daraus folgt, dass in diesem Fall der Schleifenrumpf nur einmal traversiert werden darf, wobei dann der enthaltenen Referenz die komplette Zeichenkette bis zum nächsten Terminal (nach der Schleife) bzw. zum Quelltextende zugeordnet wird, selbst wenn diese eigentlich von vielen Referenzen stammt. Bei der Rekonstruktion der Tokens ist diese Referenz nach Möglichkeit wieder zu zerteilen. Ist dies nicht möglich muss das Template an der entsprechenden Stelle geändert werden (vgl. Abschnitt 3.2).

2.2.6 Parse-Direktiven

Durch Parse-Direktiven werden Templates eingebunden und arbeiten dann mit dem gegenwärtigen Velocity Context. Beim Reverse Engineering kann man nun entweder das Parsen unterbrechen und mit dem neuen Template fortsetzen oder man verknüpft das neue mit dem gegenwärtigen. Bei der ersten Lösung ist es erforderlich, sich den Rücksprung in das alte Template zu merken, wobei andere Informationen, wie bereits gefundene Belegungen, auch beim Parsen mit dem neuen Template vorhanden sein müssen. Bei der zweiten Lösung ist dies nicht notwendig, allerdings muss da der Knoten, der das `#parse` kapselt, durch den Wurzelknoten des neuen Templates ersetzt werden. Die Entscheidung fiel auf die zweite Möglichkeit, da diese einfacher zu implementieren schien. Dadurch kann das Parsen nach dem Einbinden des neuen Templates fortgesetzt werden, als handele es sich von Beginn an nur um ein einzelnes Template.

2.3 Reasoning

Nach Abschluss des Parsens liegen für den eingelesenen Quelltext meist mehrere mögliche Belegungen vor, da es zur Parsezeit manchmal nicht möglich ist, Alternativen auszuschließen. Deshalb wird in der nächsten Phase des Reverse Engineering jedes gefundene Resultat einem Plausibilitätscheck unterzogen. Dabei werden beim Parsen belegte boo-

lesche Ausdrücke nach Möglichkeit in ihre einzelnen Bestandteile zerlegt, sowie nocht nicht belegte Ausdrücke durch logisches Schlussfolgern belegt. Dies, sowie das Aufteilen von `ConcatenatedExpression`-Objekten (vgl. Abschnitt 2.2.2), ist die Aufgabe des *Reasoners.*

Um diese Aufgaben zu bewältigen kann man entweder ein bestehendes Reasoning-System verwenden oder ein eigenes implementieren. Beispielsweise gibt es an der Universität Standford ein Reasoning System namens JTP[7], welches man zur Auswertung und Überprüfung der logischen Belegungen verwenden hätte können. Allerdings kommen in den Bedingungen von zur Codegenerierung eingesetzten Templates meist nur zusammengesetzte logische Ausdrücke, bestehend aus `and`, `or` und `not`, vor. Um diese Ausdrücke auszuwerten, ist das Einbinden einer externen Bibliothek nicht notwendig, da diese Auswertung schneller selbst implementiert ist als die Einarbeitung und Anbindung der zusätzlichen Bibliothek dauern würde.

Beim Parsen wurden aufeinanderfolgende Referenzen als eine `ConcatenatedExpression` zusammengefasst und mit der gefundenen Zeichenkette belegt. Aufgabe des Reasoners ist es zunächst, diese Zeichenkette auf die Referenzen aufzuteilen. Die Idee des dazu entwickelten Algorithmus ist es, dass es nicht notwendig ist, alle möglichen Zerlegungen zu testen, da man sich die in Abschnitt 2.2.2 vorgestellten *Constraints* zu Nutze machen kann. Der ersten Referenz wird zunächst eine leere Zeichenkette zugewiesen und dann getestet, ob sie eine gültige Belegung für die Referenz ist. Dies ist der Fall, wenn die Referenz entweder während des Parsens mit dem leeren String belegt wurde oder das assoziierte Constraint unter dieser Belegung gültig ist. Handelt es sich nicht um eine gültige Belegung, wird die Referenz mit dem ersten Zeichen der Gesamtzeichenkette belegt. Falls dies eine gültige Belegung ist, wird versucht, die nächste Referenz auf die gleiche Weise zu belegen. Angekommen bei der letzten Referenz muss diese mit dem Rest der Gesamtzeichenkette belegt werden. Wenn die Belegung der letzten Referenz ebenfalls gültig ist, wird die Zerlegung zur Ergebnissmenge hinzugefügt und anschließend wieder mit der Belegung der ersten Zeichenkette begonnen. Sie wird mit der nächst längeren Zeichenkette überprüft (im Beispiel mit den ersten beiden Zeichen der Gesamtzeichenkette). Dieser Vorgang wird so lange wiederholt, bis alle möglichen Zerlegungen getestet wurden. Auf diese Weise werden schnell alle gültigen Belegungen für die Referenzen gefunden.

[7] [1]

Praxistests haben gezeigt, dass es durch Vergabe von Constraints für Referenzen möglich ist, die Zerlegungen der gefundenen Zeichenkette auf exakt eine Möglichkeit zu beschränken. Deshalb gibt die Implementierung beim Finden mehrerer Belegungsmöglichkeiten nur eine Warnmeldung aus und erzeugt keine zusätzlichen Resultate. Bei Bedarf kann dies später leicht nachimplementiert werden, da das gefundene Resultat nur kopiert und um die neue Zerlegung ergänzt werden muss.

Nachdem die aufeinanderfolgenden Referenzen nach Möglichkeit zerlegt wurden, untersucht der Reasoner die Bedingungen der If-Abfragen. Soweit möglich geschieht dies zwar schon zur Parsezeit (vgl. Abschnitt 2.2.4), allerdings kann man komplexere logische Ausdrücke meistens erst nach vollständigem Parsen des Quelltextes in einzelne, auswertbare Ausdrücke zerlegen. Für das Auswerten dieser Bedingungen ist der `SimpleLogicHandler` zuständig. Seine Funktionsweise soll an folgendem Beispiel erläutert werden: Angenommen eine Bedingung, die vom Parser mit `true` belegt wurde, enthält den Ausdruck `($a || $b) && $c`. Zuerst wird untersucht, ob der Ausdruck gültig ist. Da logische Ausdrücke von Velocity stets als *Binärbaum*[8] repräsentiert werden, kann der Ausdruck an der Wurzel zerlegt werden und jeder Teilbaum seinerseits auf Gültigkeit geprüft werden. Im Beispiel wird der Ausdruck am `&&` getrennt und jeder Teilbaum muss wieder zu `true` ausgewertet werden, denn nur genau dann ist der Gesamtausdruck auch wahr. Dies geschieht so lange rekursiv, bis der Ausdruck in seine Blätter, die Referenzen (im Beispiel `$a`, `$b` und `$c`), zerlegt ist. Für eine Referenz kann der Reasoner dann untersuchen, ob sie vom Parser während des Parsens belegt wurde. Ist dies der Fall, wird der Wahrheitswert gemäß Belegung zurückgegeben. Wurde `$c` beispielsweise mit der Zeichenkette '`abc`' belegt, ist der Wahrheitswert `true`. Darüber hinaus ist es möglich, dass die Referenz als Bedingung in einer anderen If-Abfrage verwendet wurde. Dann kann der dort für sie festgestellte Wahrheitswert zurückgegeben werden. Sollte `$c` in einer anderen Bedingung mit `false` belegt worden sein, ist das gesamte Resultat ungültig und wird verworfen. Wird für `$c` keine Belegung gefunden, kann zunächst keine Aussage über `$c` gemacht werden. Zur Implementierung wurde dafür eine *ternäre Logik* verwendet, die die Zustände `true`, `false` und `undef` bietet.

Kann die Gültigkeit eines Ausdrucks nicht widerlegt werden, wird davon ausgegangen, dass er gültig ist. Deshalb kann den nicht belegten Referenzen im nächsten Schritt der entsprechende Wahrheitswert zugewiesen werden, falls dies eindeutig möglich ist. Im

[8] In einem *Binärbaum* hat jeder Knoten maximal zwei Kinder.

Beispiel würde `$c` den Wert `true` zugewiesen bekommen; sind hingegen `$a` und `$b` nicht definiert, kann keine weitere Belegung vorgenommen werden.

Sind alle Bedingungen abgearbeitet, werden nun die Zuweisungen untersucht. Allerdings muss bei Zuweisungen beachtet werden, dass die Bedingungen, unter denen sie gelten, zuvor geprüft werden. Wie in Abschnitt 2.2.4 erklärt, werden Zuweisungen in If-Abfragen, die weder Terminale noch Referenzen enthalten, mit den für sie geltenden Bedingungen gespeichert. Deshalb müssen diese zunächst ausgewertet werden, bevor die Zuweisung weiterverarbeitet werden kann. Wird eine Bedingung zu `false` ausgewertet, wird die betroffene Zuweisung nicht weiter beachtet. Kann die Bedingung noch nicht ausgewertet werden, weil einzelne Komponenten der Bedingung nicht definiert sind, wird die Zuweisung zunächst nicht verarbeitet.

Der Reasoner verarbeitet nur Zuweisungen, die auf der rechten Seite einen logischen Ausdruck und auf der linken Seite einen Wahrheitswert enthalten. Diese werden von dem gleichen Mechanismus verarbeitet, wie die Bedingungen zuvor. Folglich können auch mit Hilfe von Zuweisungen, die als ungültig ausgewertet werden, Resultate verworfen werden. Angenommen, es liegt die Zuweisung `#set( $do = $foo && $bar)` vor. Wurde die Referenz `$do` während des Parsens zu `true` ausgewertet, `$foo && $bar)` jedoch als `false`, handelt es sich um ein ungültiges Resultat, welches verworfen werden kann. Sind `$foo` und `$bar` noch unbelegt, kann ihnen der Wahrheitswert `true` zugewiesen werden. Auf diese Weise werden weitere Belegungen der Referenzen ausgerechnet.

Das Ausrechnen der Belegungen für die Referenzen ist zunächst reihenfolgenabhängig. Deshalb wird der vorgestellte Algorithmus so lange wiederholt, bis sich die Ergebnismenge (die Belegungen für die Referenzen) nicht mehr verändert. Nach Abschluss des Reasoning liegt im Idealfall pro Template und Quelltext ein Resultat mit Belegungen aller vorkommenden Referenzen des Templates vor. Tests haben gezeigt, dass es trotzdem noch zu mehreren Resultaten pro Template und Quelltext kommen kann, vor allen Dingen, wenn das Template mehrere direkt aufeinanderfolgende Schleifen enthält, die den gleichen Schleifenrumpf haben. Da sich diese dann inhaltlich nur marginal voneinander unterscheiden (die Belegungen werden unterschiedlich auf die beteiligten Mengenvariablen aufgeteilt), werden im Folgenden die Alternativen nicht weiter betrachtet, sondern stets mit dem ersten Resultat fortgefahren. Theoretisch müssten alle Alternativen weiterverarbeitet werden, da sonst die spätere Rekonstruktion des Modells beeinträchtigt werden könnte. Dies ist bei den von CodeGen2 zur Quelltexterzeugung eingesetzten Templates jedoch nicht der Fall.

2.4 Tokenerzeugung

Nach Abschluss des Reasoning liegt pro Template und Quelltext ein Resultat mit den Belegungen aller vorkommenden Referenzen des Templates vor. Anhand dieser Resultate wäre es nun bereits möglich, das Fujaba Modell zu rekonstruieren. Allerdings enthalten die Resultate nicht nur Informationen aus dem `Context`-Objekt (vgl. Abschnitt 1.2.2), sondern ebenfalls Belegungen lokaler Variablen des Templates. Da sich diese lokalen Variablen potentiell in Templates verschiedener Zielsprachen unterscheiden, können sie nicht zur Modellrekonstruktion verwendet werden - sonst würde das folgende Mapping auf das Fujaba Metamodell abhängig vom konkreten Templatedesign werden, was nicht wünschenswert ist. Deshalb ist das Ziel dieser Phase die Rekonstruktion und Bereinigung der zur Codegenerierung verwendeten Zwischenschicht, den sog. `Token` (vgl. Abschnitt 1.2.2), mit welcher danach das Fujaba Metamodell aufgebaut bzw. aktualisiert werden kann.

Wie in Abschnitt 1.2.2 erläutert, repräsentieren Token einzelne zu generierende Quelltextfragmente (beispielsweise ein einzelnes Attribut oder eine Methodendeklaration) und werden während der Quelltexterzeugung in Form eines Baums miteinander verlinkt. Außerdem gibt es für jedes Token genau ein Template. CodeGen2 trennt in der vorliegenden Version nicht strikt zwischen Token und Fujaba Metamodell - vielmehr enthalten die `Token`-Objekte jeweils eine Referenz zu einem Objekt des Fujaba Metamodells. Diese Objektreferenz wird, zur Quelltexterzeugung, an Velocity im `Context`-Objekt übergeben. Anschließend kann aus den Templates über den Schlüssel `$elem`[9] auf das jeweilige Metamodellobjekt zugegriffen werden. Dadurch gelangen sämtliche zur Quelltexterzeugung notwendigen Informationen (zum Beispiel Name, Sichtbarkeit und Initialisierungswert eines Feldes) an Velocity und eine entsprechende Quelltextdatei kann erzeugt werden. Folglich kann umgekehrt, durch das Zusammenstellen aller gefundenen Belegungen für die Referenzen, die sich auf das Modellobjekt beziehen, das Token rekonstruiert werden. Dabei handelt es sich zunächst nur um einfache Zeichenketten, jedoch kann aus diesen dann im nächsten Schritt das Metamodell in Fujaba rekonstruiert bzw. aktualisiert werden.

Die Aufgabe dieser Phase des Reverse Engineering besteht also aus zwei Teilschritten.

[9]Die Bezeichnung `$elem` ist nur eine Konvention: In allen von CodeGen2 verwendeten Templates wird mit diesem Schlüssel auf das Objekt des Metamodells verwiesen. Der Name kommt von `FElement`, der Superklasse aller Fujaba Metamodellklassen.

Zunächst müssen wieder `Token`-Objekte erzeugt werden und diese korrekt als Baum miteinander verlinkt werden. Anschließend muss das jeweilige `Context`-Objekt rekonstruiert werden, wobei lokale Variablen nicht mehr vorkommen dürfen. Der erste Teil der Aufgabe ist trivial, denn es genügt pro Template ein `Token`-Objekt zu erzeugen und anschließend diese anhand der von CodeGen2 vorgegebenen Templatehierarchie miteinander zu verlinken. Für den zweiten Teil der Aufgabe sind zunächst einige Vorüberlegungen notwendig.

Abbildung 2.4: *Beispiel einer Assoziation.*

Angenommen es geht um die Quelltexterzeugung einer Assoziation, wie sie in Abbildung 2.4 illustriert ist. Um den Namen der Rolle zu bekommen, also den Namen des Feldes, über den die Gegenseite referenziert ist, erfolgt der Zugriff aus dem Template auf das `Context`-Objekt mit dem Schlüssel `$elem.role.AttrName`[10]. In den zur Quelltexterzeugung eingesetzten Templates werden häufig lokale Variablen verwendet, um Schreibarbeit zu sparen bzw. ein häufiges Traversieren komplexer Objektstrukturen zu vermeiden. Dazu könnte also im Beispiel der Rollenname mit `#set( $roleName = $elem.role.AttrName )` der lokalen Variablen `$roleName` zugewiesen werden. Oder die Rolle selbst, weil ihre Attribute im weiteren Verlauf des Templates öfters benötigt werden, könnte durch `#set( $role = $elem.role )` der lokalen Variablen `$role` zugewiesen werden. Angenommen beim Parsen wurde `$roleName` mit der Zeichenkette *leftRole* belegt. Aufgabe ist es nun, die lokale Variable zu eliminieren. Dies geschieht durch erneutes Anwenden der Zuweisungen. Dadurch wird das `Context`-Objekt wieder vollständig rekonstruiert und gleichzeitig lokale Variablen wieder entfernt: Wird nun die Zuweisung `#set( $roleName = $elem.role.AttrName )` angewendet, werden alle Belegungen, die die Zeichenkette `$roleName` als Referenz enthalten, durch `$elem.role.AttrName` ersetzt. Es bleibt die (korrekte) Belegung von `$elem.role.AttrName` mit der Zeichenkette *leftRole* übrig. Wurde im Template eine Referenz mit einem Wert Initialisiert, also beispielsweise `#set( $someValue = true )`, dann bewirkt das erneute Anwenden der Zuweisung die Entstehung von Belegungen der Form `true => true`. Diese können entfernt werden, indem nur Belegungen gestattet

[10] `$elem` ist das Metamodellobjekt (im Beispiel die Assoziation), `role` ein Feld der Assoziation (nämlich die Rolle) und `AttrName` schließlich der Name der Rolle.

werden, die als Schlüssel eine Referenz und kein Literal haben. Ein drittes Problem sind veränderliche Referenzen (vgl. Abschnitt 2.2.3). Da sie nicht für die Weiterverarbeitung geeignet sind, dürfen sie nicht im `Context`-Objekt verbleiben und müssen entfernt werden.

Nach diesen Vorüberlegungen kann im Folgenden die Umsetzung der geschilderten Lösungsstrategien erläutert werden. Zunächst wird für jedes gefundene Resultat ein `Token`-Objekt erzeugt; diese werden anschließend anhand der bekannten Template-Hierarchie (vgl. Abschnitt 1.2.2) verlinkt. Nun müssen die Belegungen der vom Parser gefundenen Resultate auf die Token übertragen werden, wobei lokale Variablen herausgerechnet werden müssen. Dies geschieht für alle erstellte Token in drei Arbeitsschritten:

- Im ersten Schritt werden alle Belegungen des Resultats dem `Token`-Objekt hinzugefügt. Diese sog. `Assignment`-Objekte enthalten als Schlüssel die Referenz und als Wert die gefundene Zeichenkette bzw. den beim Reasoning zugewiesenen booleschen Wert.

- Im zweiten Schritt werden die während des Parsens gefundenen Zuweisungen angewendet. Auf diese Weise werden, wie geschildert, lokale Variablen entfernt und die Belegungen des `Context`-Objekts rekonstruiert. Der zweite Schritt wird so lange wiederholt, bis es bei den `Assignment`-Objekten des `Token`-Objekts keine Veränderungen mehr gibt.

- Im dritten Schritt werden unbrauchbare Werte vom Token entfernt. Zum einen sind dies Werte veränderlicher Referenzen (vgl. Abschnitt 2.2.3), also Werte von Referenzen, die zunächst im Template verwendet wurden und anschließend durch eine Zuweisung anders belegt wurden. Zum anderen sind es `Assignment`-Objekte, die als Schlüssel ein Literal und keine Referenz haben.

Konnten während des Parsens und beim anschließenden Reasoning sämtliche Referenzen belegt werden, existieren nun keine lokalen Variablen mehr. Wenn nicht, so dürfen diese beim anschließenden Aktualisieren des Fujaba Metamodells nicht verarbeitet werden, da das Mapping sonst abhängig vom konkreten Templatedesign werden würde.

2.5 Modellaktualisierung

Nach Abschluss der Tokenerzeugung liegt eine Baumstruktur von Tokens vor, die jeweils Tupel, bestehend aus Referenzen und dazugehörendem Wert, enthalten. Mit Hilfe dieser Informationen muss nun das Fujaba Modell entweder neu erzeugt werden (im Falle von Reverse Engineering) oder das Modell eines aktuell geöffneten Projekts aktualisiert werden (im Falle von Roundtrip Engineering). Dies geschieht in der letzten Phase des Reverse Engineering.

Während des Parsens erfolgte die Zuordnung von Zeichenketten zu Referenzen. Jede Referenz stammte ursprünglich - während der Quelltexterzeugung - aus dem `Context`-Objekt des Tokens bzw. somit aus dem Fujaba Metamodell. Problematisch ist nun, dass das rekonstruierte `Context`-Objekt keine typisierten Objekte des Fujaba Metamodells enthält, sondern die belegten Referenzen lediglich Zeichenketten sind. Ein Beispiel für eine solche Referenz ist `$elem.PartnerRole.AttrName`, wie sie in Templates zur Erzeugung von Assoziationen verwendet wird. Während der Quelltexterzeugung griff Velocity auf die Variable `$elem` zu, also das assoziierte Objekt des Fujaba Metamodells. Anschließend erfolgte ein Zugriff auf das Feld `PartnerRole` und auf diesem Objekt dann der Feldzugriff auf `AttrName`, also den Namen des Attributs. Dieser Name wurde dann in den Quelltext geschrieben. Während des Reverse Engineering müssen die Nachbarobjekte (im Beispiel zunächst das Objekt zum Feld `PartnerRole` des Metamodell Objekts) zunächst gesucht werden, bevor sie weiterverarbeitet werden können. Diese Suche generisch für alle gefundenen Belegungen der Tokens zu implementieren ist nicht trivial und war nicht Teil der Aufgabe.

Die während der vorliegenden Arbeit entstandene Implementierung enthält explizite Regeln zum Mapping auf in Klassendiagrammen vorkommende Modellelemente, wie beispielsweise Klassen, Methoden, Felder und Assoziationen. Die Regeln sind, gemäß *Strategy-Design-Pattern* (siehe [2]), auf jeweils einzelne Klassen aufgeteilt.

Der *ModelUpdater* erhält aus den vorangegangenen Phasen für jede geparste Klasse einen Tokenbaum mit jeweils verwendetem Template. Dieser Baum wird traversiert und jedes Token einzeln verarbeitet. CodeGen2 liefert dazu die Information, welche Klasse des Fujaba Metamodells von dem jeweils vorliegenden Token repräsentiert wird. Anhand dieser wählt der *ModelUpdater* die zu verwendende Strategie. Handelt es sich beispielsweise um ein Token, welches ein Assoziationsende (eine *Rolle*) repräsentiert, wird die Aktualisierung des Modells an den `CustomRoleGenerator` delegiert. Seine Funktions-

weise soll im folgendem Exemplarisch für die anderen Mappingregeln erklärt werden. In Abbildung 2.5 ist eine Assoziation zur Anschaulichkeit gezeigt.

Abbildung 2.5: *Beispiel einer Assoziation.*

Zunächst werden die Namen der Assoziation, der Quellklasse, der Rolle, der Zielklasse und der Zielrolle aus dem Token durch direkten Zugriff auf die Belegung gewonnen. Ebenso werden die Sichtbarkeiten der Rollen und deren Kardinalitäten extrahiert. Nun muss die Quellklasse gesucht werden. Zunächst kann dazu auf das Vatertoken zugegriffen werden, da dieses (im Beispiel bei Assoziationen) die Klasse mit der Assoziation generiert hat. Allerdings ist es möglich, dass die Assoziation nicht in der derzeitigen Klasse deklariert ist, sondern beispielsweise in einem von der Klasse implementierten Interface. Daher muss in den Superklassen die Klasse gesucht werden, deren Name mit dem Namen der gefundenen Quellklasse übereinstimmt. Wird diese gefunden, so wird nun in dieser Klasse eine Rolle mit dem gefundenen Namen gesucht, um die Rolle dann entweder zu aktualisieren oder um sie neu anzulegen. Komplizierter ist die Suche nach der beteiligten Partnerklasse, zu der die Assoziation gezogen werden soll. In Fujaba existieren für alle Modellelemente Fabriken (gemäß *Factory-Method-Design-Pattern*, siehe [2]), die sich die erzeugten Elemente merken. Die `UMLClassFactory` enthält eine Liste der erzeugten Klassen. Existiert dort eine der Zielklasse gleichnamige Klasse, deren Paketname auch mit den importierten Klassen der aktuellen Klasse übereinstimmt (beispielsweise im gleichen Paket liegt oder explizit von der Quellklasse importiert wird), kann in der Zielklasse die Zielrolle gesucht werden und bei bedarf ebenfalls neu angelegt werden. Existiert die Zielklasse hingegen noch nicht im derzeitigen Projekt, muss sie neu erzeugt werden. Da Fujaba bei der Quelltexterzeugung sowohl JavaDoc als auch Annotationen erzeugt, können Assoziation gut gesucht bzw. neu erstellt werden.

Ähnlich der vorgestellten Strategie zum Mapping von Assoziationen funktionieren die anderen Mappings für die restlichen Modellelemente eines UML Klassendiagramms. Nach Abschluss der Aktualisierung des Fujaba Metamodells ist das Roundtrip Engineering vollzogen und Refaktorierungen des Quelltextes sind zurück auf das Feindesign in Fujaba abgebildet.

3 Ergebnisse

3.1 Laufzeitverhalten

Um templatebasiertes Reverse Engineering umzusetzen, ist es notwendig, Backtracking einzusetzen (vgl. Abschnitt 2). Prinzipiell erreicht man durch Backtracking jedoch eine Kostenexplosion, denn es werden - falls notwendig - alle Wege abgegangen, bis ein Ergebnis gefunden wird. Beim vorgestellten Ansatz ist es sogar notwendig, alle möglichen Belegungen zu finden, denn erst später, beim Reasoning, können einzelne Belegungen mit Sicherheit ausgeschlossen werden. Jede Alternative, die untersucht werden muss, bedeutet eine Steigerung der Laufzeit.

Während der Implementierung des Parsers haben sich nun an vielen Stellen Möglichkeiten zur Laufzeitoptimierung ergeben. Die größte Optimierung war es, `Constraints` einzuführen. Zunächst unterliegen alle Referenzen dem `SingleLineConstraint`, dürfen also keine Zeilenumbrüche enthalten. Wie in 2.2.2 erläutert ist der Gewinn immens, denn so wird verhindert, dass kurze Terminale, wie beispielsweise ein Semikolon am Zeilenende, an jeder Zeile des Quelltexts passen und viele ungültige Resultate, die zunächst alle geprüft werden müssen, hervorbringen. Durch die Vergabe von `SingleWordConstraints` konnte dies noch optimiert werden - dann darf eine Referenz nichtmal mehr Leerzeichen enthalten. Letztere sind jedoch sparsam einzusetzen, denn sonst verliert man die Flexibilität, die Templates zu verändern ohne die Konfiguration des Parsers verändern zu müssen.

Zweige von If-Abfragen, die nur Zuweisungen und weder Referenzen noch Terminale enthalten, sind eine weitere Möglichkeit, Laufzeit zu sparen, indem nicht - wie normalerweise - pro Bedingung ein neues Resultat erzeugt wird, sondern stattdessen mit der Zuweisung die Bedingung, unter der sie gilt, vermerkt wird (siehe 2.2.4).

Darüber hinaus können Bedingungen von If-Abfragen - soweit möglich - schon während des Parsens ausgewertet werden, sodass dann weitere Zweige nicht überprüft werden müssen (siehe 2.2.4).

Insgesamt ist die Laufzeit bei Quelltexten, die eine Klasse mit beispielsweise mehreren Assoziationen enthalten, befriedigend: sie liegt bei wenigen Sekunden. Um weitere Optimierungen vornehmen zu können, ist es erforderlich, den Quellcode des Parsers mit einem Profiler zu analysieren, um so Verursacher hoher Kosten ausfindig zu machen. Der Parser wurde unter dem Gesichtspunkt eines robusten und guten Designs implementiert, was nicht notwendigerweise mit einer guten Laufzeit einhergeht. Deshalb ist davon auszugehen, dass noch viel Potential in der Optimierung der Algorithmen liegt - beispielsweise durch Caching oder Einsatz von Nebenläufigkeiten.

3.2 Anforderungen an die Templates

Das Ziel bei der Entwicklung des Parsers war eine möglichst große Flexibilität für den Templateentwickler bei der Gestaltung der Templates. Dennoch müssen einige Anforderungen von den Templates erfüllt werden, damit sie für das Reverse Engineering eingesetzt werden können:

- Der Parser unterstützt keine *Makros*, also das Definieren von Methoden im Template. Unter dem Aspekt der sauberen Trennung zwischen Logik (in der Software) und Markup (im Template) sollten Makros nicht in Templates eingesetzt werden. Daher ist dieses Manko vernachlässigbar.

- Um das `Context`-Objekt rekonstruieren zu können, dürfen die im Template vorkommenden Referenzen nicht verändert werden (vgl. Abschnitt 2.2.3). Nachdem eine Referenz in einer Bedingung oder direkt im Text zur Quelltexterzeugung eingesetzt wurde, darf sie nicht mehr durch eine Zuweisung verändert werden. Sonst ist sie für die weitere Verarbeitung nicht von Nutzen.

- Schleifen müssen sorgsam konzipiert werden. Zunächst kann der Parser nicht sinnvoll mit Schleifen umgehen, die keine Terminale enthalten. Enthält eine Schleife nur Referenzen, wird das Parsen nach der Schleife fortgesetzt und der gesamte Zwischenraum den Referenzen zugeordnet. Eine spätere Aufteilung ist zur Zeit nicht möglich. In der derzeitigen Implementierung existiert darüber hinaus noch ein Problem bei der Verarbeitung von Schleifen, deren letztes Element im Rumpf die Iteratorvariable ist. Dies ist nach Möglichkeit zu vermeiden, da es sonst zu Problemen bei der Zuordnung zur Mengenvariablen kommen kann.

- Der Parser enthält keine Komponente, die mit arithmetischen Ausdrücken umgehen kann, wobei in den bisher zur Quelltexterzeugung eingesetzten Templates auch keine arithmetischen Ausdrücke vorkommen.
- Direkt aufeinanderfolgende Referenzen (`ConcatenatedExpression`, vgl. 2.2.2 und 2.3) können nur schwer wieder getrennt werden. Sie sind nach Möglichkeit zu vermeiden. Alternativ ist dafür zu sorgen, dass eine eindeutige Zerlegung nach Abschluss des Parsens möglich ist, beispielsweise durch Vergabe von zusätzlichen Constraints.

3.3 Fazit und Ausblick

Um qualitativ hochwertige, wart- und erweiterbare Software werkzeugunterstützt entwickeln zu können, müssen CASE Tools Roundtrip Engineering beherrschen. Das templatebasierte Reverse Engineering ist hierfür - durch seine Zielsprachenunabhängigkeit - ein sehr flexibles Konzept. Mit dem im Rahmen dieser Diplomarbeit entwickelten Plugin für die Fujaba Tool Suite lassen sich erfolgreich kleinere quelltextbasierte Refaktorierungen auf die Designdokumente in Fujaba übertragen. Dabei werden zunächst nur Klassendiagramme unterstützt, diese allerdings vollständig. Da die Implementierung des Plugins vom Parsen bis zur Erzeugung der Tokenstruktur generisch ist, funktionieren die anderen von Fujaba angebotenen Diagramme (Story Diagramme und Statecharts) prinzipiell ebenfalls. Lediglich die Aktualisierung des Fujaba Metamodells muss für die fehlenden Diagramme noch implementiert werden. Da dieses zunächst nicht Aufgabe dieser Diplomarbeit war, ist es nur exemplarisch implementiert. Sobald es generisch implementiert ist, wird es sich auch flexibel verhalten und die vollständige Unterstützung der restlichen Diagramme ermöglichen.

Ein prinzipielles Problem der Umsetzung ist es, dass der einzulesende Quelltext nicht vom Template abweichen darf. Bereits geringfügige Unterschiede zum Template bereiten Probleme und führen zur Verwerfung des Lösungswegs. Wünschenswert wäre hier eine gewisse Toleranz. Beispielsweise werden aufeinanderfolgende Leerzeichen momentan wie jedes andere Terminal zur Zuordnung von Quelltext zu einer Referenz verwendet. Dies könnte man derart ändern, dass alle Leerzeichen (sowohl im Quelltext als auch im Template) nur noch als einzelnes Leerzeichen verwendet werden. Auf diese Weise könnte der Entwickler die Quelltextdokumente nach den eigenen Vorlieben formatieren. Beim

Wiedereinlesen könnte man dem Entwickler dann beispielsweise Feedback geben, wie die Templates verändert werden müssten, damit bei erneuter Quelltexterzeugung der erzeugte Text mit dem eingelesenen übereinstimmt. Die Anpassung der Templates könnte daraufhin automatisch geschehen.

Eine Möglichkeit zur Laufzeitreduzierung wäre es, beim Parsen nicht linear vorzugehen, sondern zunächst größere, zusammenhängende Terminalstücke aus dem Template im Quelltext zu suchen, um anschließend die verbleibenden Lücken zu schließen. Da die Implementierung des Parsers sehr modular geschehen ist, könnte er mit überschaubarem Aufwand zu dieser Funktionsweise modifiziert werden.

Eine weitere sehr interessante Idee betrifft das Einlesen von Quelltext, der *nicht* mit Fujaba und damit *nicht* mit Templates generiert wurde. Diesen einzulesen vermag die vorliegende Arbeit nicht. Sobald die oben angesprochene Flexibilität gegenüber Leerzeichen implementiert ist, könnte man jedoch für diesen Fall Templates pro Zielsprache verfassen, die allgemeinen Quelltext generieren und damit einlesen können. So könnten, ausgehend von manuell geschriebenen Quelltext, jedenfalls Klassendiagramme relativ einfach rekonstruiert werden und von Methodenrümpfen zumindest der Kontrollfluss.

Es bleibt die Aufgabe zukünftiger Arbeiten, auf die vorliegende aufzubauen. Allein die beschriebenen Ideen enthalten das Potential für eine spannende Fortsetzung. Lohnend ist es allemal, schon deshalb, um Fujaba das 'Back Again' vollwertig wiederzugeben.

A Anhang

A.1 Template Beispiele

Um einen Eindruck von den zur Quelltexterzeugung eingesetzten Templates zu bekommen, befinden sich in diesem Abschnitt ausgesuchte Beispiele. Die folgenden Templates werden zur Erzeugung von Methodendeklarationen verwendet.

- Einstiegstemplate für Methodendeklarationen: *classDiag/method/declaration.vm*

```
#parse("classDiag/method/import.vm" )

#set( $return = $imports.addToImports($method.ResultType.FullClassName) )
#if( $comment )
#if( $parsed )
$comment
#elseif( $comment )
/**
$!comment
 **/
#end
#end
#if( $initializer )
static
#else
$visibility ##
#if( $abstract )abstract #end##
#if( $native )native #end##
#if( $synchronized )synchronized #end##
#if( $static )static #end##
#if( $final )final #end##
$returnType ##
$name (##
#parse( "classDiag/method/params.vm" )
```

```
)##
#set( $firstThrow = true )
#foreach( $throw in $throws )
#if( $firstThrow )
#set( $return = $imports.addToImports($throw.FullClassName) )
 throws #set( $firstThrow = false )
#else
, ##
#end
$throw.Name##
#end
#end
#if( $abstract || $interface )
;
#else
#if( $parsed && $body)
   $body
#else

{
#if( $body )
$body
#else
$!children
#end
#end
} // end CodeGen2 generated method ##

#end
```

- Mapping des `Context`-Objekts auf lokale Variablen: *classDiag/method/import.vm* (wird vom Einstiegstemplate mit einer `#parse`-Direktive eingelesen)

```
#set( $method = $elem )
#set( $name = $method.Name )
#set( $returnType = $utility.getTypeAsString($method.ResultType) )
#set( $initializer = ( $returnType.equals("Initializer") ) )
#set( $visibility = $utility.getVisibilityString($method.Visibility) )
#set( $abstract = $method.abstract )
#set( $native = $method.hasKeyInStereotypes("native") )
```

```
#set( $synchronized = $method.hasKeyInStereotypes("synchronized") )
#set( $static = $method.static )
#set( $final = $method.hasKeyInStereotypes("final") )
#set( $signal = $method.hasKeyInStereotypes("signal") )
#set( $params = $method.iteratorOfParam() )
#set( $throws = $method.iteratorOfThrowsTypes() )
#set( $body = $method.MethodBody )
#set( $parsed = $method.parsed )
#set( $comment = $method.Comment.Text )
#set( $start = $method.RevSpec )
#if( !$comment &&  $start )
#set( $comment = $start.Comment.Tex )
#end
#set( $interface = $utility.isInterface($method.Parent) )
```

- Template zur Erzeugung der Methodenparameter *classDiag/method/params.vm* (wird vom Einstiegstemplate mit einer `#parse`-Direktive eingelesen)

```
#set( $firstParam = true )
#foreach( $param in $params)
#if( !$firstParam ), #else
#set( $firstParam = false )#end
#set( $return = $imports.addToImports($param.ParamType.FullClassName) )
$utility.getTypeAsString($param.ParamType) $param.Name ##
#end
```

Die drei gezeigten Templates dienen der Generierung von Methodendeklarationen, wie der im folgenden Beispiel:

```
   public abstract boolean doSomething (String arg0 , int arg1 );
```

A.2 Klassendiagramme

In diesem Abschnitt werden einige Klassendiagramme des fertigen Parsers gezeigt. Sie dienen der Veranschaulichung der in den jeweiligen Abschnitten erläuterten Implementierungen.

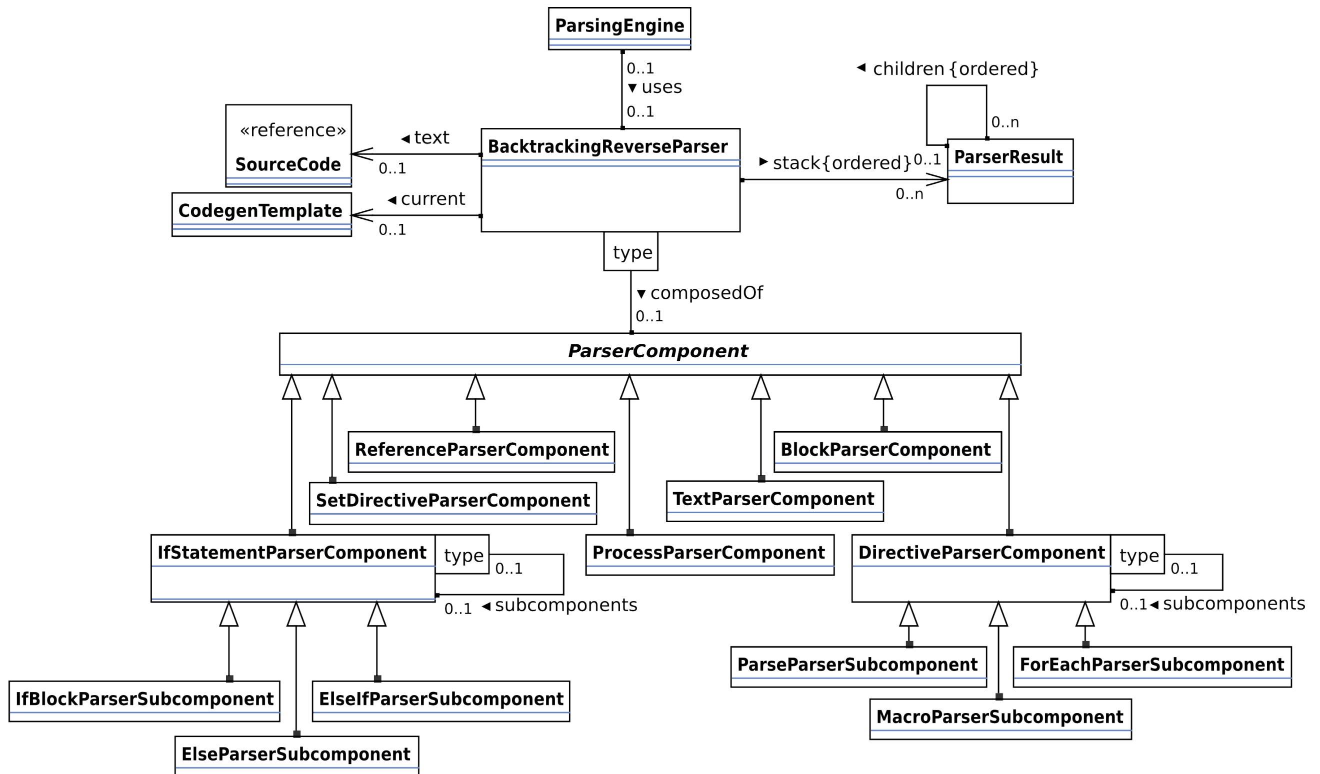

Abbildung A.1: *Klassendiagramm des Parsers.*

Abbildung A.1 zeigt das Klassendiagramm der Kernkomponente der vorliegenden Arbeit, des Parsers. Die zentrale Klasse ist der `BacktrackingReverseParser`, welcher die diversen `ParserComponents` (gemäß *Strategy-Design-Pattern*, siehe [2]), verwaltet. Die Strategien sind über den Typ des VTL Sprachkonstrukts qualifiziert, es erfolgt also ein Zugriff mit *O(1)* auf die jeweils zuständige Strategie.

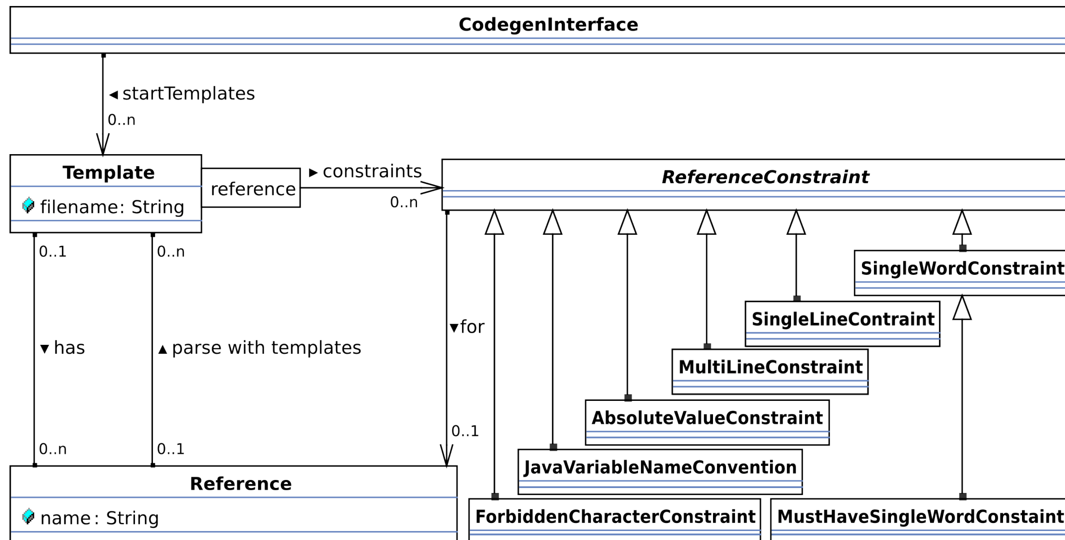

Abbildung A.2: *Klassendiagramm des von CodeGen2 genutzten Interfaces.*

Abbildung A.2 zeigt das Interface (Klasse `CodegenInterface`), über welches auf Informationen aus CodeGen2 zugegriffen wird. Da dies ursprünglich nicht von CodeGen2 vorgesehen war, entstand es ebenfalls im Laufe dieser Arbeit. Es enthält eine Liste von Start-Templates, mit welchen der Parser das Analysieren einer neuen Klasse beginnen muss. Jedes Template enthält dann wiederum eine Liste von Referenzen, die von anderen Templates geparst werden müssen. Darüberhinaus enthält jedes Template eine Liste von Constraints (Klasse `ReferenceConstraint`), die für einzelne Referenzen des Templates vergeben werden können. Der Zugriff erfolgt qualifiziert mit dem Namen der Referenz, also ebenfalls mit *O(1)*. Dies ist wichtig, denn während des Parsens muss beim Belegen jeder Referenz die Einhaltung eventueller Constraints geprüft werden.

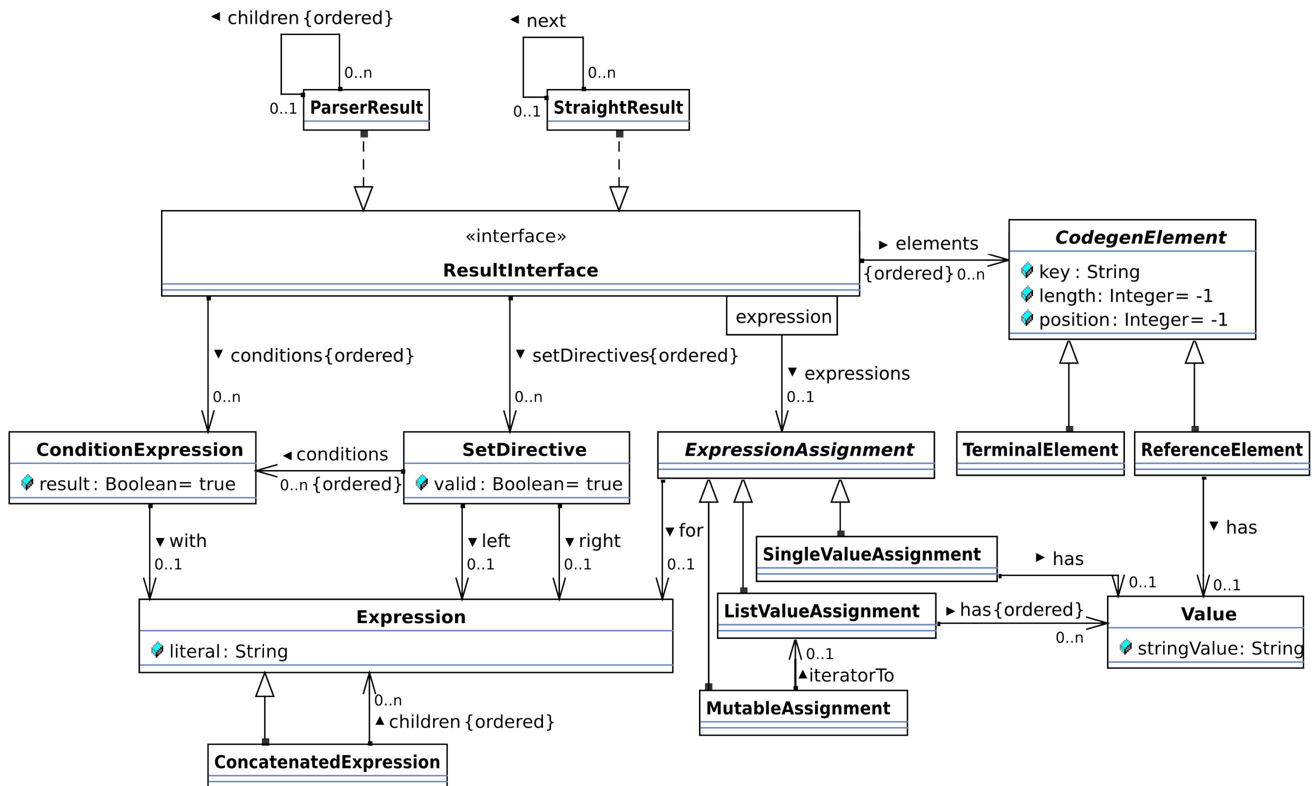

Abbildung A.3: *Klassendiagramm der während des Parsens erzeugten Datenstruktur.*

Abbildung A.3 zeigt die vom Parser angelegte Datenstruktur, in der gefundene Belegungen gespeichert werden. Während des Parsens ist das `ParserResult` die zentrale Klasse. Über die `children`-Assoziation werden beim Antreffen von Alternativen mehrere neue Kind-Resultate dem gegenwärtigen Ergebnis hinzugefügt. Bedingungen werden als `ConditionExpression`-, Zuweisungen als `SetDirective`- und Belegungen als `ExpressionAssignment`-Objekte dem aktuellen Resultat hinzugefügt. `CodegenElement`-Objekte sind im Template vorkommende Terminale und Referenzen. Sie wurden ursprünglich nur zu Testzwecken vermerkt, können dann aber zukünftig auch zwecks Anpassung des Templates verwendet werden. Wichtig ist, dass die Referenzen als `Expression`-Objekte gekapselt werden. Der Zugriff erfolgt auf sie gemäß *Flyweight-Design-Pattern* (siehe [2]). Da nach Beenden des Parsens sehr hohe Bäume aus `ParserResult`-Objekten vorliegen, werden diese in `StraightResult`-Objekte „geplättet“: Belegungen aller `ParserResult`-Objekte eines Pfades von der Wurzel bis zu einem Blatt werden einem `StraightResult`-Objekt hinzugefügt. Dies ermöglicht eine effizientere Weiterverarbeitung, da dann die einzelnen Belegungen nicht stets transitiv

in allen Eltern gesucht werden müssen. Über die `next`-Assoziation sind dann die einzelnen Alternativen (ehemalige Blatt-Nachbarn) erreichbar.

Literaturverzeichnis

[1] R. Fikes, J. Jenkins, and G. Frank. JTP: A System Architecture and Component Library for Hybrid Reasoning. In *Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, Florida, USA, July 2003.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[3] L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-Tools. In *3rd International Fujaba Days 2005*, Paderborn, Germany, September 2005.

[4] R. Johnson and W. Opdyke. Refactorings: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposion on Object-Oriented Programming Emphasizing Practical Applications*, September 1990.

[5] OMG: Object Management Group. Unified Modeling Language Specification: Version 2, Revised Final Adopted Specification (ptc/04-10-02), 2004.

[6] What is a CASE environment. http://www.sei.cmu.edu/legacy/case/case_whatis.html, 2007.

[7] Fujaba Website. http://www.fujaba.de, 2007.

[8] Fujaba Partners. http://wwwcs.uni-paderborn.de/cs/fujaba/partners/, 2007.

[9] Java Compiler Compiler. http://javacc.dev.java.net/, 2007.

[10] The Apache Velocity Project. http://velocity.apache.org/, 2007.

[11] A. Zündorf. Rigorous Object Oriented Software Development. Habilitation Thesis, University of Paderborn, 2001.

Abbildungsverzeichnis

1.1 FUJABA Story Driven Modeling 3
1.2 Konzept der Codegenerierung von Codegen2 5
1.3 Formatierung durch Kommentare 9

2.1 Illustration If-Abfrage 21
2.2 Illustration If-Abfrage mit Zuweisung 22
2.3 Illustration foreach-Loop 23
2.4 Beispiel einer Assoziation 29
2.5 Beispiel einer Assoziation 32

A.1 Klassendiagramm des Parsers 40
A.2 Klassendiagramm des Interfaces zu CodeGen2 41
A.3 Klassendiagramm der Datenstruktur des Parsers 42

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Diplomarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum Unterschrift