Diplomarbeit

Alexander Krohn

Ein Round-Trip-Engineering-Modul
am Beispiel von Python und UML

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Krohn

Ein Round-Trip-Engineering-Modul am Beispiel von Python und UML

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Softwaretechnik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Bernd Kahlbrandt
Zweitgutachter : Prof. Dr. rer.nat. Kai von Luck

Abgegeben am 10. Juli 2008

Alexander Krohn

Thema der Diplomarbeit

Ein Round-Trip-Engineering Modul am Beispiel von Python und UML

Stichworte

Round-Trip-Engineering, UML, Python, Quellcode, Modell

Kurzzusammenfassung

Diese Arbeit befasst sich mit den Möglichkeiten des Generierens von Pytho-Quellcode auf Basis von UML-Modellen und das Generieren von UML-Modellen auf Basis von Python-Quellcode

Alexander Krohn

Title of the paper

A Round-Trip-Engineering-Module for Python and UML

Keywords

Round-Trip-Engineering, UML, Python, Source code, Model

Abstract

This report describes the possibilities of generating code in Python based on UML-Models and generating UML-Models based on code in Python.

Danksagung

Ich danke den Betreuern dieser Arbeit Prof. Dr. rer. nat. Bernd Kahlbrandt und Prof. Dr. rer. nat. Kai von Luck für ihre Zeit und ihre Geduld.

Ich danke dem ArgoUML-Team für die Unterstützung, insbesondere Tom Morris für die Hilfe bei der Infrastruktur des Python-Moduls und Christian López Espínola für Tests und Kritik.

Weiterhin danke ich Frank Wierzbicki vom Jython-Projekt für seine Arbeit an der Python-Grammatik für ANTLR.

Diplomarbeit
Ein Round-Trip-Engineering-Modul am Beispiel von Python und UML

Alexander Krohn

9. Juli 2008

Inhaltsverzeichnis

Kapitel 1

Motivation

UML bietet die Möglichkeit, objektorientierte Klassen zu modellieren und Attribute und Methoden dieser Klassen festzulegen. Algorithmen lassen sich in UML bisher nicht modellieren, da dies nicht Aufgabe oder Inhalt eines Designs auf einem hohen Abstraktionslevel ist. Das heißt, dass ein Gerüst einer Software, nämlich die enthaltenen Klassen, bis zu einem bestimmten Grad definierbar sind, aber sobald nicht-triviale Algorithmen in der Software realisiert werden sollen, müssen diese per Hand im Quellcode realisiert werden. Wenn also Quellcode auf Basis eines UML-Modells generiert wird, so kann in den meisten Fällen davon ausgegangen werden, dass dieser Quellcode den Programmierer unterstützt, aber ihm seine Arbeit nicht gänzlich abnimmt.

Wenn ein Modell einer Software durch eine formale Notation erstellt wird, ist es nahe liegend, diese Informationen *automatisiert* in den Quellcode einfließen zu lassen, also auf Basis des Modells Quellcode zu generieren. Dies ist mit UML-Modellen nur in bedingtem Umfang möglich, da die *konkreten Implementationen* in UML nicht beschrieben werden können. UML legt eindeutig den Schwerpunkt auf die Modellierung der Konzepte einer Software, aber nicht auf die Beschreibung der eigentlichen Funktionalität.

Bei einem Vorgehensmodell wie dem *Wasserfallmodell*(siehe [5]) folgt die Phase der Implementierung auf die Phase des Designs. Wenn die Implementierung begonnen hat, sind keine Änderungen am Modell mehr möglich.

Bei anderen Vorgehensmodell, z. B. dem *V-Modell*(siehe [6]) sind diese Phasen auch strikt von einander getrennt, es ist aber vorgesehen, dass die Designphase im Verlauf des Projekts immer wieder stattfindet, um Änderungen durch neue oder geänderte Anforderungen einfließen zu lassen. Dies trägt dem Verlauf realer Projekte deutlich besser Rechnung, es bringt allerdings für automatisch generierten Code einige Schwierigkeiten mit sich.

Wenn Änderungen am Design Konsequenzen für bereits erstellten Quellcode hat, so muß dieser dem verändertem Modell angepasst werden. Bei bereits generiertem Quellcode muß also eine erneute Generierung erfolgen. Wenn der Quellcode bereits Implementierungen von Methoden enthält, die durch einen Programmierer eingefügt wurden, so müssen diese in den neu generierten Code eingepflegt werden.

Dies muß jedes mal entweder manuell durch einen Programmierer erfolgen, oder durch Sprachkonstrukte. Hierbei werden z.B. nur Basisklassen für die Implementationsklassen generiert oder Ableitungen von bereits existierenden Im-

plementationen generiert. In beiden Fällen ergibt sich aber Aufwand dadurch, dass Design und Implementation bewusst als völlig separate Dinge behandelt werden.

Es wäre also wünschenswert, wenn Design und Implementation enger verbunden werden, in dem Sinne, dass Modell und Quellcode verlustfrei in das entsprechende Pendant transformiert werden können.

In dieser Arbeit wird für das UML-Tool *ArgoUML*(siehe Abschnitt 7.1) ein Modul entworfen, dass Quellcode in der Programmiersprache *Python*(siehe Abschnitt 7.2) sowohl einlesen als auch generieren kann.

Dafür wird ein Metamodell entworfen, das die Grammatik der Sprache *Python* möglichst vollständig abbildet. Bei der Entwicklung des Metamodells wird auch die Grammatiken anderer Programmiersprachen eingegangen, z. B. *C++* und *Java*. Dieses Metamodell wird als Zwischenstufe zwischen UML-Modell und Quellcode arbeiten.

Kapitel 2

Metamodell

2.1 Einleitung

Im folgenden soll ein Metamodell entwickelt werden, welches die Konstruke aus Python-Quellcodedateien abbildet. Dieses Metamodell wird später durch den Parser verwandt und mit Daten befüllt. Die Klassen des Metamodells brauchen dazu keinerlei besondere Funktionalität, sie sollten aber natürlich die nötigen Zugriffsmethoden für ihre Attribute bereitstellen. Darum bietet es sich an, das Metamodell in MOF(siehe nächsten Abschnitt) zu definieren und die Klassen des Metamodells für die Verwendung durch den Parser automatisch generieren zu lassen.

In den folgenden Abschnitten werden die Konzept MOF und Roundtrip Engineering genauer erläutert.

2.2 MOF und UML

Die *Unified Modelling Language*(UML) ist eine Spezifikation der *Object Management Group*(OMG)[14]. Die Version 1.4.2 dieser Spezifikation ist als *ISO/IEC 19501* von der *International Organization for Standardization*(ISO)[8] als Standard akzeptiert worden. Die Bestandteile von UML, die in dieser Arbeit verwendet werden, beziehen sich auf UML 1.4[15].

Durch sie werden Modellelemente beschrieben, mit denen bestimmte Aspekte einer Software beschrieben werden können, z. B. Klassen und ihre Attribute und Methoden, Packete in denen Klassen zusammengefasst werden können, usw. Außerdem werden Diagrammtypen beschrieben. Die einzelnen Diagramme visualisieren das Zusammenwirken von Klassen, den sequentiellen Ablauf von Methodenaufrufen, Zustände und die entsprechenden Zustandsübergänge, usw. Zum Zeitpunkt dieser Arbeit ist die aktuelle Version von UML 2.1.2[16]-

Durch *Meta Object Facility*(MOF) wurde eine Methode spezifiziert, um die Modellierung selbst zu beschreiben, die Bestandteile von UML sind also durch MOF beschreibbar. Diese Beschreibung ermöglicht den Austausch zwischen verschiedenen Modellierungswerkzeugen und Werkzeugen, welche die Modellierungsdaten weiterverarbeiten, z. B. um Code zu erzeugen. MOF beschreibt also die Modellelemente, mit denen in UML gearbeitet werden kann: Pakete, Klassen, Assoziationen, usw. Die Version MOF 2.0[20] ist momentan aktueller Standard.

MOF kann dazu verwandt werden bestehende UML-Elemente zu erweitern oder auch um spezielle Metamodelle zu entwickeln.

Die Beschreibung der Modellelemente werden durch MOF in einem sogenannten *Repository* gehalten. Der Zugriff auf das Repository, also das Anlegen, Ändern und Löschen von Elementen, ist im Falle MOF 2.x durch *MOF Versioning and Development Lifecycle* spezifiziert. Dieses Repository erlaubt das Erstellen, Ändern und Löschen von Model-Objekten auf Basis des jeweiligen Metamodells.

Im Zuge dieser Arbeit wurden zwei verschiedene Implementationen von MOF 2.x evaluiert. Die an der HU Berlin erstellte Bibliothekt aMOF2ForJava[4] bietet eine Implementation des MOF 2.x Standards, sie kann Metamodelle per XMI importieren, als auch exportieren. Ebenso kann für ein Metamodell Java-Code generiert werden, um mit Objekten eines Metamodells zu arbeiten.

Außerdem wurde das Tool MOFLON[7] von der TU Darmstadt getestet. Es bietet neben der MOF-Implementation einen Metamodell-Editor, einen Code-Generator und eine eigene Transformationssprache(siehe Abschnitt 4.1.2).

2.3 Roundtrip Engineering

Das Generieren von Quellcode auf Basis eines Modells wird *Forward Engineering* genannt. Das Extrahieren von Modelldaten aus Quellcode wird als *Reverse Engineering* bezeichnet. Sind beide Möglichkeiten vorhanden, so spricht man vom *Roundtrip Engineering*.

Booch beschreibt Roundtrip Engineering für UML und weißt darauf hin, dass es beim *Forward Engineering* als auch beim *Reverse Engineering* zu Informationsverlusten kommt. In der Praxis, so Booch, würde man mit *stereotypes* und *tagged values* arbeiten. Ein *stereotype* würde dann einem Modellobjekt zugewiesen und ein Codegenerator könnte anhand des *stereotypes* und den *tagged values* den Code für das Objekt generieren. Der *stereotype* und die *tagged values* können also als Metainformationen über ein Modellobjekt betrachtet werden.

In dieser Arbeit wird versucht, möglichst auf den Einsatz von *stereotypes* und *tagges values* zu verzichten. Dies hat verschiedene Gründe. Zugunsten der Übersichtlichkeit wird auf den Einsatz von stereotypes möglichst verzichtet. Sollten sich Situation ergeben, wo der Einsatz ratsam wäre, wird dies entsprechend erwähnt. Ähnlich verhält es sich mit den *tagged values*, sie sollen nur als letzte Möglichkeit angesehen werden, das gewünschte Ziel zu erreichen. In den meisten Fällen, in denen sich der Einsatz von *stereotypes*, bzw. *tagged values* anbietet, handelt es sich um Fälle in denen entschieden werden muss, wie eine Information zu interpretieren oder abzubilden ist. Es wird in diesen Situationen eher Wert darauf gelegt, die möglichen Lösungswege und ihre Vor- und Nachteile zu beschreiben, als eine einzige Lösung mittels *stereotypes* und *tagged values* anzustreben.

Für eine vollständige Abbildung von Quellcode in Modellkonstrukten müssen nicht nur Klassendefinitionen und Methodensignaturen beschrieben werden, sondern auch Konstrukte, welche die Logik der Software repräsentieren. Die möglichen Anweisung in einer Programmiersprache müssen durch Objekte im Modell dargestellt werden können. Um diese Konstrukte abzubilden, wird im Folgenden ein Metamodell für die Sprache *Python* entworfen. Da dies hier aber auf einem abstrakten Niveau geschieht, sind die Elemente dieses Modells aber

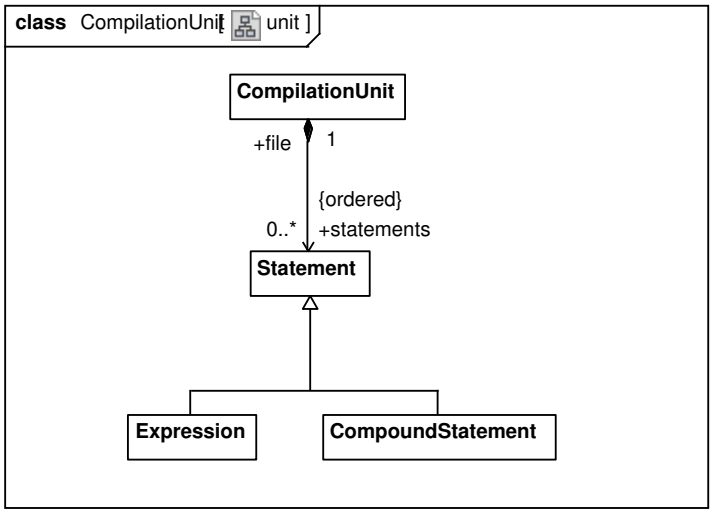

Abbildung 2.1: Das Metamodell einer Übersetzungseinheit

durchaus auch auf andere Programmiersprachen anwendbar.

2.4 Metamodell der Sprache *Python*

Im Folgenden wird ein Metamodell für die Sprache Python entwickelt. Hierbei wird versucht, das Modell möglichst generisch zu halten um die Möglichkeiten auszuloten, auch andere Sprachen in dieses Modell zu transformieren. Es werden also nicht alle sprachspezifischen Merkmale von Python abgebildet.

Die Folgenden Diagramme wurden mit MagicDraw(siehe 7.5) erstellt, da die Ausgabe am Besten für eine Dokumentation geeignet ist. Um das eigentliche Modell zu erstellen wurde MOFLON (siehe 7.4), dies hat den Vorteil, den Code für das Metamodell generieren zu können, die Ausgabe von Grafiken für Dokumentationen ist allerdings beschränkt. Außerdem verwendet MOFLON für ein Paket nur ein Diagramm, das gesamte Modell ist in einem Diagramm aber sehr unübersichtlich.

Da zum Zeitpunkt dieser Arbeit der MOF-Standard in der Version 2.x noch nicht verabschiedet ist, werden die Probleme durch fehlende Implementationen und Editoren durch die Verwendung von MagicDraw für die Diagramme umgangen.

2.4.1 Übersetzungseinheiten

In Abbildung 2.1 ist das Metamodell einer Übersetzungseinheit abgebildet, sie enthält im wesentlichen eine geordnete Menge von Anweisungen.

Python- und *Java*-Übersetzungseinheiten bestehen immer nur aus einer einzigen Datei, z. B. *C++* kann aber das Interface und die Implementation in getrennten Dateien definieren. Eine *CompilationUnit* enthält eine Menge von

Statements, diese müssen geordnet sein, damit die Ausführungsreihenfolge des Quelltextes erhalten bleibt.

2.4.2 Anweisungen

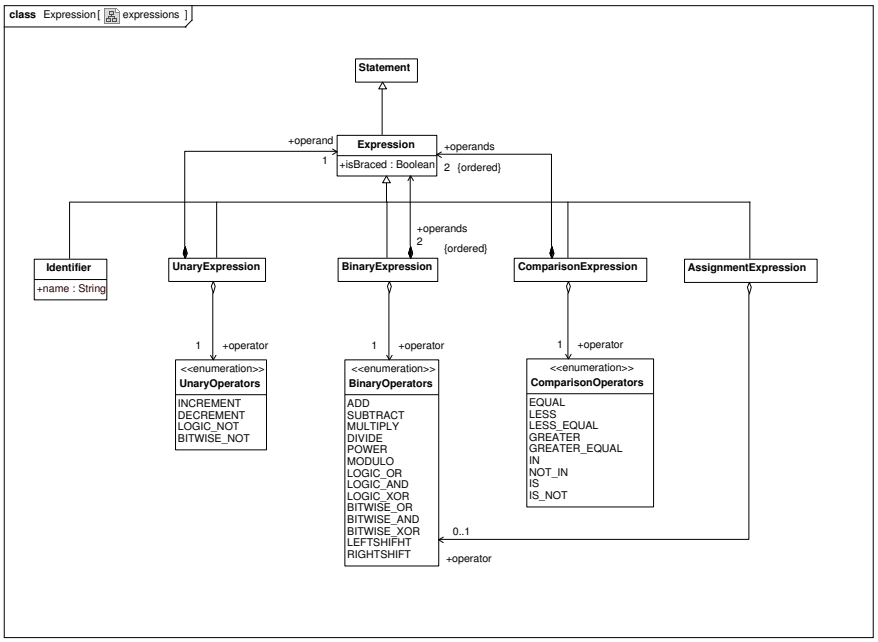

Abbildung 2.2: Das Metamodell verschiedener Ausdrücke

Die elementaren Anweisungen werden durch Objekte der Klasse *Expression* repräsentiert(Abbildung 2.2). Die Unterklassen definieren, welcher Operator verwendet wird und wie viele Operanden der Ausdruck hat. Die Operatoren stellen hier eine Auswahl der Operatoren gängiger Programmiersprachen dar, zur Vereinfachung wird angenommen, dass es zu jedem *BinaryOperator* einen passenden Zuweisungsoperator gibt. Der ternäre Operator *:?* existiert z. B. in *C* und *C++* aber nicht in *Python*, er kann dort aber über ein einfaches Hilfskonstrukt abgebildet werden.

2.4.3 Schleifen

ForLoop(Abbildung 2.3) ist ein Schleifenkonstrukt, das eine wiederholte Ausführung eines Anweisungsblocks ermöglicht. Das Assoziationsattribut *pre* stellt hierbei die Schleifeninitialisierung dar, häufig wird hier ein Zähler initialisiert. Das Attribute *post* ist ein Ausdruck, der nach jedem Schleifendurchlauf ausgeführt, z. B. um einen Zählerwert zu aktualisieren. Das Attribute *condition* stellt die Abbruchbedingung der Schleife dar.

Die *WhileLoop*(Abbildung 2.4) ist ebenfalls ein Schleifenkonstrukt. Das Attribut *postChecked* legt fest, ob es sich um eine *while-do*(*postChecked* ist Un-

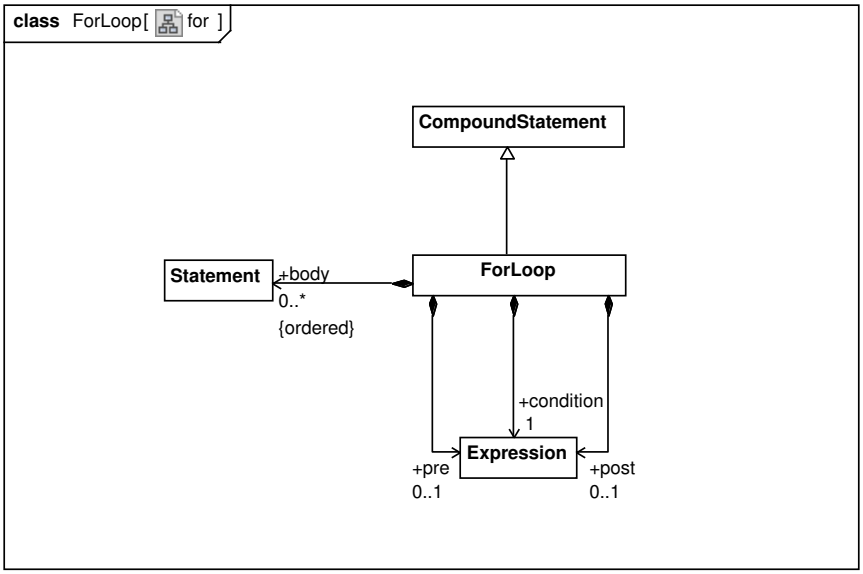

Abbildung 2.3: Das Metamodell einer *for*-Schleife

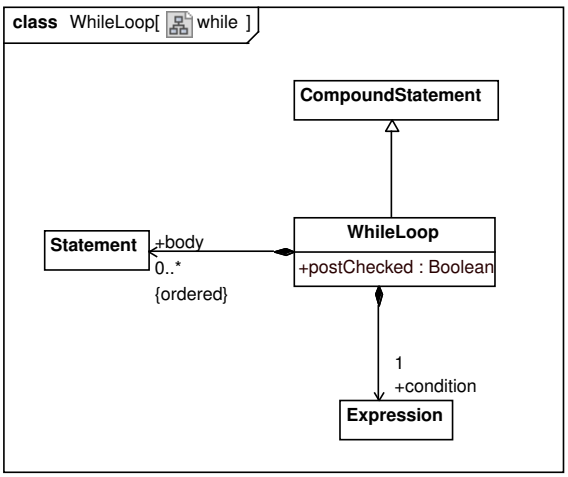

Abbildung 2.4: Das Metamodell einer *while*-Schleife

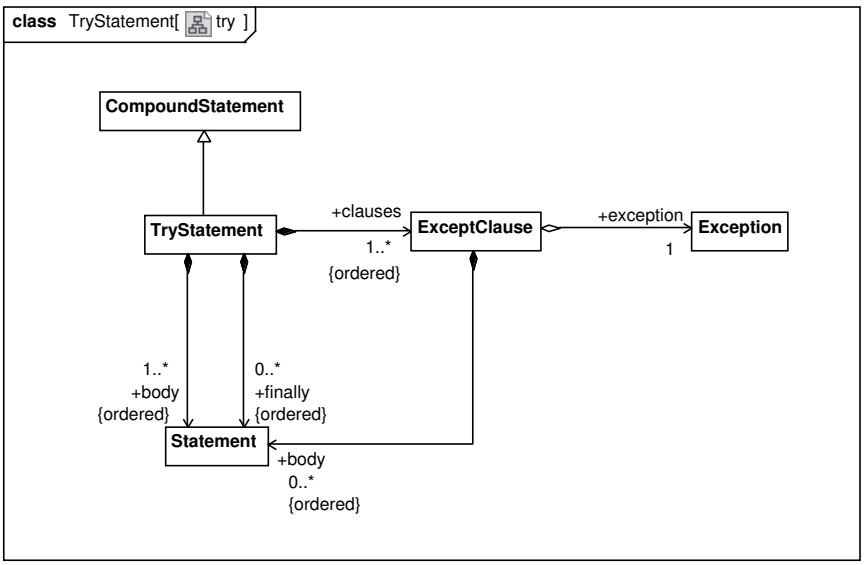

Abbildung 2.5: Das Metamodell einer *Try*-Anweisung

wahr) oder eine *do-while*(*postChecked* ist Wahr) handelt, also ob die *condition* vor oder nach einem Schleifendurchlauf geprüft werden soll.

2.4.4 TryStatement

Das *TryStatement*(Abbildung 2.5) ermöglicht die Ausführung eines Anweisungsblocks und die gezielte Behandlung von Ausnahmen, die dabei auftreten. Dieses Konstrukt ist in *Java*, *C++* und auch in *Python* vorhanden. In *C++* und *Java* spricht man beim Abfangen einer Ausnahme von einem *catch*, in *Python* wird dies durch das Schlüsselwort *except* beschrieben.

Ein *TryStatement* hat eine geordnete Menge von *ExceptClause*'s, diese stellen die Behandlungsroutinen für die entsprechende *Exception*s dar.

2.4.5 If-Then-Else

Abbildung 2.6zeigt einen *If-Then-Else*-Anweisungsblock. Der Ausdruck, der ausgewertet werden muss, um festzustellen, welcher Teil des Blocks ausgeführt werden muss, ist als *condition* benannt. Hier würde sich anbieten, eine *Comparison-Expression* einzusetzen, da Python aber an dieser Stelle auch andere Ausdrücke zulässt, ist die *condition* eine *Expression*.

2.4.6 Klassen

Ein Klassendefinition(Abbildung 2.7) besteht im wesentlichen aus einem Namen für die Klasse und einer Liste von Basisklassen, von denen diese Klasse erbt. Die Definitionsrumpf dieser Klasse wird an dieser Stelle nur als geordnete Menge von Anweisungen betrachtet. In Python können innerhalb einer Klassendefinition jegliche Form von Anweisungen auftauchen, also nicht nur Definitionen von

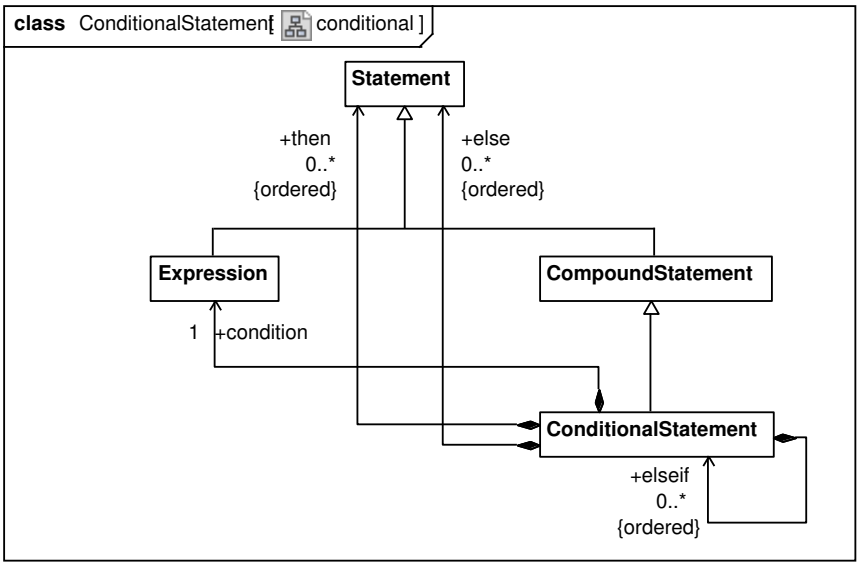

Abbildung 2.6: Das Metamodell einer *If-Then-Else*-Anweisung

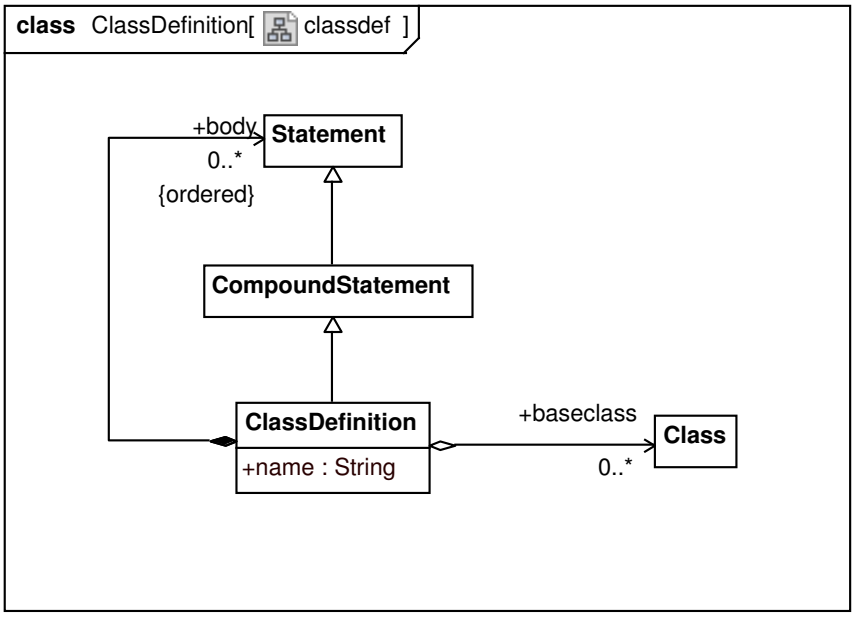

Abbildung 2.7: Das Metamodell einer Klassendefinition

Attributen, Methoden, usw. sondern jeglicher Ausführbarer Code. Dieser wird zur Laufzeit während der Definition der entsprechenden Klasse ausgeführt.

Die Angaben über die Basisklassen sind in diesem Modell stark vereinfacht, Python selbst unterstützt hier einige weitere Formen.

Das folgende Code-Beispiel zeigt eine Klassendefinition mit leerem Rumpf, bei der einige Basisklassen angegeben sind(*Python* unterstützt *Mehrfachvererbung*). Das Schlüsselwort *pass* ist lediglich ein Platzhalter, um den leeren Rumpf einer Klasse, Methode oder eines Blocks zu kennzeichnen. Da Python die Einrückungen am Zeilenanfang als Begrenzungen für diese Rümpfe versteht, würde ein leerer Rumpf den Parser durcheinander bringen.

```
class MyClass(MyBaseClassA, MyBaseClassB):
    pass
```

Die Klasse *MyClass* ist von den Klassen *MyBaseClassA* und *MyBaseClassB* abgeleitet. Diese Angaben ließen sich relativ einfach speichern und später in einem UML-Modell umsetzen, da auch hier Basisklassen angegeben werden können.

Python unterstützt jedoch sehr viel mehr Funktionalität bei der Angabe der Basisklasse(n). innerhalb der Klammern in der Klassendefinition kann auch ein Ausdruck stehen, der zur Laufzeit ausgewertet wird und somit erst zur Laufzeit feststeht, von welcher Basisklasse abgeleitet wird. Die wird am Beispiel eines Interfaces für XML-Parser gezeigt.

```
FastXMLParser = False
try:
    from xmlparser import FastXMLParser
except ImportError:
    # FastXMLParser konnte nicht importiert werden
    pass
class MyXMLParser(FastXMLParser or DefaultXMLParser):
    pass
```

Hier wird am Anfang versucht die Implementation eines XML-Parsers zu laden. Dieser könnte in C geschrieben sein, aber nicht auf allen Plattformen lauffähig sein, o.ä. Falls der Import fehlschlägt verbirgt sich hinter der Variablen *FastXMLParser* der Wert *False*. Falls der Import gelingt, ist unter diesem Namen der eigentlich XML-Parser verfügbar. Wenn der Basisklassenausdruck der Klasse *MyXMLParser* ausgewertet wird, so wird eine der beiden Basisklassen durch den Bool'schen Operator or ausgewählt, *FastXMLParser* falls der Import möglich wahr sonst *DefaultXMLParser*.

Um solche spezifischen Möglichkeiten umzusetzen, gibt es verschiedene Möglichkeiten. Der Parser könnte so umgeschrieben werden, dass er zwar eine Liste von Basisklassen akzeptiert, aber die spezielleren Ausdrücke nicht. Dann wären diese Ausdrücke nicht im Metamodell abzubilden und das Modell könnte auf weiterhin für mehrere Sprachen genutzt werden. Dies hätte aber den Nachteil, dass der Parser bestimmte valide Python-Konstrukte nicht akzeptiert und er somit streng genommen nicht konform mit der Python-Grammatik ist. Für generierten Code wäre das kein Problem, sofern diese Ausdrücke nicht generiert werden. Bei Einlesen von bestehendem Code allerdings würden diese Ausdrücke nicht akzeptiert und im schlimmsten Falle der Vorgang für die gesamte Übersetzungseinheit abgebrochen.

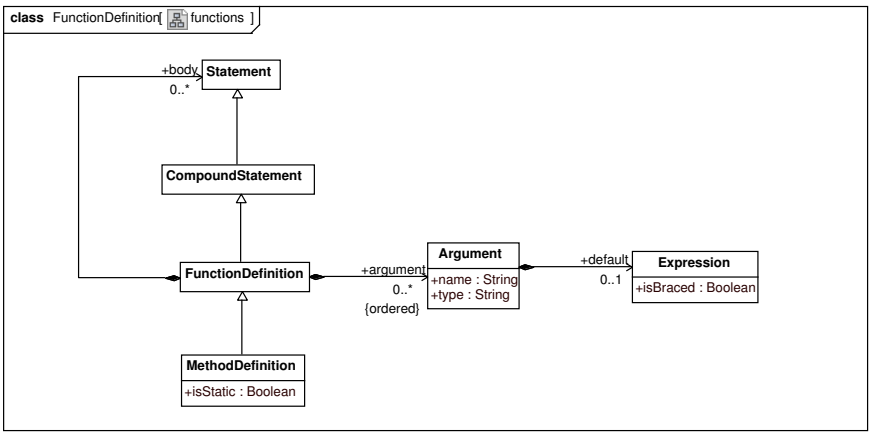

Abbildung 2.8: Das Metamodell von Funktions- und Methodendefinitionen

Eine weitere Möglichkeit wäre, das Metamodell um diese spezifischen Ausdrücke zu erweitern. Dann könnte der Parser Python-konform bleiben, aber das Metamodell wäre nicht mehr für mehrere Sprachen verwendbar. Der Nachteil hierbei wäre, das auch die Transformation von Metamodell nach UML nur für Python gültig wäre. Für eine andere Sprache müsste dann ein neues Metamodell erstellt werden und nicht nur die Transformation durch einen Parser, sondern auch die Transformation von Metamodell nach UML.

Ebenso wäre es möglich, den Ausdruck, welcher die Basisklasse(n) beschreibt in einer Notiz an das UML-Modellelement der Klasse zu hängen. Damit bliebe die Information erhalten und für den Benutzer zugreifbar.

2.4.7 Funktionen und Methoden

Funktionen und Methoden(Abbildung 2.8) werden in *Python* sehr ähnlich behandelt. Wird eine Funktion innerhalb einer Klasse definiert, dann ist sie als Methode für Objekte dieser Klasse verfügbar. Ihr erster Parameter wird dabei beim Aufruf als Referenz auf das Objekt selber verwandt. Per Konvention wird dieser Parameter *self* genannt, dies entspricht dem *this* in *Java* und *C++*. Dort ist diese Referenz aber nicht explizit in der Methodensignatur aufgeführt.

Beispiel:

```
class A(object):
    def myMethod(self, a):
        self.a = a
```

Python unterscheidet außer diesen Instanzmethoden noch Klassenmethoden und statische Methoden. Diese Methoden werden syntaktisch nicht gekennzeichnet, sondern durch einen bestimmten Funktionsaufruf zu einer Klassenmethode oder einer statischen Methode.

```
class A(object):
    def myClassMethod(cls, a):
        print a
```

```
myClassMethod = classmethod(myClassMethod)
def myStaticMethod(a):
    print a
myStaticMethod = staticmethod(myStaticMethod)
```

Die Funktion *classmethod(...)* macht aus der Methode *myClassMethod* eine Klassenmethode und weißt sie ihrem ursprünglichen Namen zu. Eine Klassenmethode bekommt als ersten Parameter die Klasse selbst übergeben, ähnlich dem *this* in einer Instanzmethode. Dieser wird per Konvention *cls* für *class* genannt. Eine statische Methode bekommt beim Aufruf weder eine Referenz auf die Klasse noch auf die Instanz übergeben.

Das Umwandeln in diese speziellen Methodentypen kann in *Python* über so genannte *Decorator* etwas kompakter geschrieben werden:

```
class A(object):
    @classmethod
    def myClassMethod(cls, a):
        print a
    @staticmethod
    def myStaticMethod(a):
        print a
```

Die Funktionen *classmethod*, *staticmethod*, usw. sind natürlich keine syntaktischen Mittel der Sprache Python, sondern normale Funktionen, die dem Interpreter zur Verfügung stehen. Der Umstand, ob eine Methode z. B. eine Klassenmethode ist, wird also nicht über Schlüsselwörter der Sprache angegeben, wie z. B. in Java durch das Schlüsselwort static, sondern erfolgt zur Laufzeit durch den Aufruf einer Funktion.

Dies bedeutet für den Parser, dass er diese Schlüsselwörter nicht kennt und somit kann auf dieser Ebene die Information, ob eine Methode eine statische oder eine Klassenmethode ist, nicht extrahiert werden. Dies muss also in der weiteren Transformation geschehen. Es müssten also für die obigen Beispiele die Dekoratoren für eine Methode betrachtet werden und weiterhin, ob im Namensraum der Klasse ein Aufruf der speziellen Funktionen erfolgt. Da der Rückgabewert der Methode die eigentliche Klassenmethode ist, kann diese auch unter einem anderen Namen zugewiesen werden. All diese Umstände erschweren natürlich die Extrahierung dieser Informationen.

Weitere Schwierigkeiten ergeben sich bei der Definition der Parameter für Funktionen und Methoden, im Zusammenhang mit Listentypen. Dazu werden diese Listentypen zunächst etwas näher beschrieben.

Die beiden Haupttypen für Listen heißen in Python *tuple* und *list*. Ein *tuple* ist eine nicht veränderbare Liste von Werten, d.h. die einzelnen Werte können nicht direkt verändert werden. Falls es sich um Objekte handelt, können aber deren Methoden aufgerufen werden und damit der Zustand der Objekte geändert werden, aber die Referenz auf das Objekt selbst kann nicht geändert werden. Außerdem wird die Länge eines *tuples* bei dessen Instantiierung vorgegeben und kann ebenfalls nicht verändert werden. Der Zugriff auf Elemente eines *tuples* erfolgt über numerische Indices. Die gilt auch für den Typ *list*, dieser kann allerdings verändert werden, sowohl dessen Inhalte, als auch die Länge der Liste.

Das folgende Beispiel zeigt einige Anwendungen des Typs *list*:

```
# eine leere Liste
my_list = []
# ein Wert wird angehängt, die Liste wächst
my_list.append("ein Wert")
# ein Wert wird per numerischem Index ausgelesen
a = my_list[0]
# ein neuer Wert wird zugewiesen
my_list[0] = "ein anderer Wert"
# eine neue Liste wird erstellt
my_list = [1, 3, 5, 2, 4, 6]
# die liste wird sortiert
my_list.sort()
```

Fast ähnlich ist die Verwendung des Typs *tuple*:

```
my_tuple = (1, 2, 3)
# Auslesen eines Wertes
a = my_tuple[0]
# ein Laufzeitfehler
# ein tuple kann nicht verändert werden
my_tuple[0] = 5
```

Diese Listentypen bieten in Python nun eine Möglichkeit, alle ihre Inhalte gleichzeitig an Variablen zuzuweisen. Dies sieht folgendermaßen aus:

```
# eine Liste mit drei Werten
my_list = [1, 2, 3]
# der Inhalt der Liste wird an Variablen zugewiesen
a, b, c = my_list
```

Nach Ausführung dieses Code-Fragments enthalten die Variablen a, b und c jeweils einen Wert aus der Liste. So kann z. B. eine Liste, welche die Werte eines dreidimensionalen Vektors enthält, in die Variablen x, y und z zugewiesen werden. Dies zeigt, das sich Python's Listentypen einige Tricks verbergen.

Eine solche Expandierung einer Liste kann nun auch ein einer Funktionsdeklaration auftreten, wie das folgende Beispiel zeigt:

```
class MyClass(object):
    def myMethod(self, a, b, (c, d, e)):
        pass
```

Hier stehen innerhalb der Methode *myMethod* die Variablen *self* (das Objekt selbst) und die Variablen a bis e zur Verfügung. Der Aufruf dieser Methode sieht dann folgendermaßen aus:

```
my_list = [3, 4, 5]
myInstance = MyClass()
myInstance.myMethod(1, 2, my_list)
```

Die ersten beiden Parameter werden direkt übergeben und die Parameter c, d und e werden aus *my_list* expandiert. Falls die übergebene Liste zu viele oder

zu wenige Werte enthält, erzeugt der Aufruf einen Laufzeitfehler. Im obigen Beispiel ist der Aufruf korrekt.

Dies ist ein gutes Beispiel für den *syntactic sugar* einer Sprache, in diesem Falle Python. Ob diese syntaktischen Möglichkeiten sinnvoll sind oder nicht, bleibt natürlich dem Programmierer überlassen. Der Parser allerdings muss diese Dinge berücksichtigen und es muss eine Lösung gefunden werden, wie ein solches syntaktisches Konstrukt in das Metamodell übertragen wird und wie es durch eine spätere Transformation behandelt wird.

Die Parameter einer Funktion in Python unterstützen noch einige Dinge mehr, so gibt es Standardwerte die in der Deklaration angegeben werden können, falls der Aufruf keinen Wert für Parameter enthält. Diese Standardwerte können ganz verschiedene Python-Ausdrücke sein. Außerdem unterstützt Python Funktionen mit variabler Anzahl von Parametern und zusätzlich sog. *keyword*-Parameter, die direkt durch ihren Namen übergeben werden können.

Alle diese Möglichkeiten hier zu beschreiben, würde zu weit führen. Es wird aber schnell deutlich, das eine komplette Übertragung jeglicher syntaktischer Konstrukte das Metamodell sehr viel komplizierter werden lässt, als es hier dargestellt ist. Es wird also in dieser Arbeit eine Grenze gezogen, um das Anschaulichkeit des Modells zu bewahren und trotzdem grundlegende Konstrukte der Sprache zu erfassen.

2.4.8 Basistypen

Der Grundstein einer Programmiersprache sind die Datentypen, mit denen die Sprache arbeiten kann(Abbildung 2.9). Im Falle von Python(und auch anderen Programmiersprachen) sind das Zeichenketten, Ganzzahlen, Fließkommazahlen und benutzerdefinierte Typen(Klassen). Das Modell der Basistypen zeigt eine einfache Klassenhierarchie. In konkreten Programmiersprachen werden die Ganz- und Fließkommazahlen nach ihrer Genauigkeit unterschieden. So gibt es z. B. einen Ganzzahltyp der 32 Bit im Speicher belegt und Einen, der 64 Bit im Speicher belegt. In *Python* ist diese Unterscheidung für den Nutzer kaum sichtbar. Wenn der Wertebereich eines Integers überschritten wird, dann wird dieser einfach in den Python-Typ *long* übertragen. Der Typ *long* hat keinerlei Größenbegrenzungen, solange genug Arbeitsspeicher vorhanden ist, um den Wert zu speichern. Der Python-Typ *long* ähnelt damit dem Java-Typ *BigInt*. Fließkommazahlen haben in Python immer den Typ *float*, die Größe in Bit, bzw. die Genauigkeit hängt von der Zielplattform ab. Auf aktuellen PC-Plattformen entspricht das dem C-Typ *double*, also 64 Bit.

In Abbildung 2.9 sind Basistypen von *Python* aufgeführt.

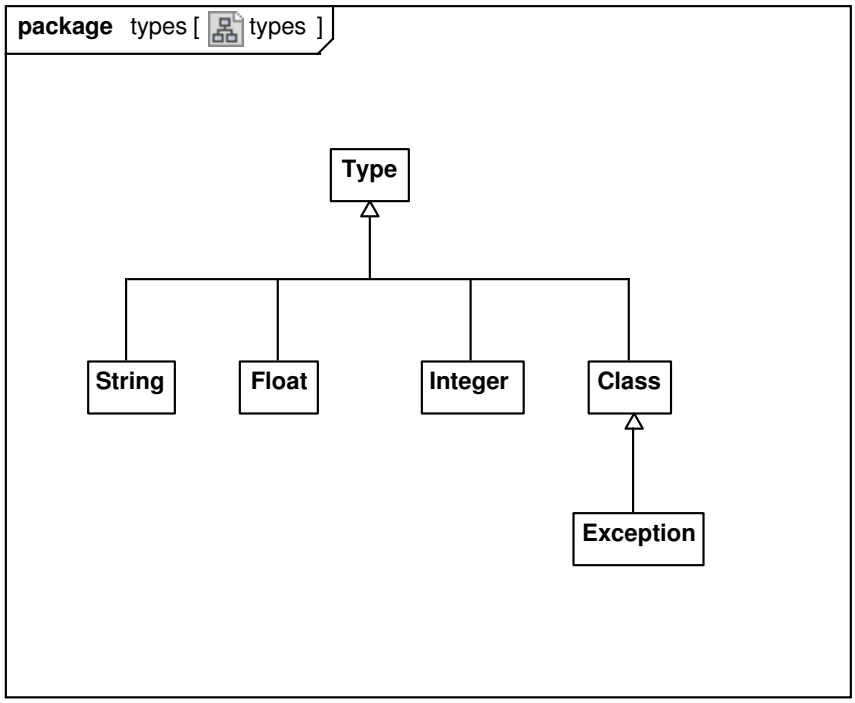

Abbildung 2.9: Das Metamodell der elementaren Datentypen

Kapitel 3

Parser

Das Einlesen von Quellcode übernimmt ein *Parser*. Dieser wird nicht von Hand geschrieben sondern durch einen *Parsergenerator* automatisch erstellt. Hierzu benötigt man eine Grammatik der Sprache, die durch den Parser verarbeitet werden soll.

Die Grammatik beschreibt die Syntax der Sprache, also den grundlegenden Aufbau. Es werden Bezeichner der Sprache(z. B. *if*, *for*, *print*) beschrieben, arithmetische und logische Operatoren(z. B. $+$, -, $>$, $<$) usw. Anhand der Grammatik erstellt der Parsergenerator zunächst den Code für die lexikalische Analyse, den sogenannten *Lexer*. Dieser überprüft einen gegebenen Quelltext, der als Folge von Zeichen vorliegt, anhand der syntaktischen Regeln der Grammatik auf Gültigkeit. Als Ausgabe produziert er eine Folge von *Tokens*, diese repräsentieren dann die Bezeichner der Sprache, Operatoren, Variablennamen, usw.

Auf Basis der Regeln der Grammatik wird zusätzlich der eigentliche *Parser* generiert, ein Programm, das als Eingabe eine Folge von syntaktisch korrekten *Tokens* verarbeitet. Der *Parser* überprüft die Gültigkeit der Folge der *Tokens*. Der Parser kann dann verschiedene Aufgaben erledigen, z. B. kann er die Eingabe für einen *Interpreter* liefern, der dann die Anweisungen der Sprache ausführt.

Dies ist bei der vorliegenden Aufgabenstellung nicht der Fall, der Parser wird stattdessen die Klassen des Metamodells(siehe Kapitel 2.4) instanzieren.

3.1 Grammatik

Eine Grammatik besteht aus sog. Produktionsregeln, diese verweisen entweder auf andere Produktionsregeln(*non-terminals*) oder auf Literale(*terminals*).

Beispiel:

```
Z := a | aZ | B
B := b | bB
```

Diese Grammatik besteht aus den Produktionsregeln Z und B und den Literalen a und b. Diese Grammatik beschreibt Sätze in der Form $aaabbb$, also eine beliebige Anzahl von a's gefolgt von einer beliebigen Anzahl b's. Das Zeichen / markiert Alternativen innerhalb einer Produktionsregel, Z produziert also a

oder *aZ* oder *B*. Dadurch ist *aaaabbb* ein gültiger Satz dieser Grammatik, *aaabbaa* hingegen nicht, weil die Produktionsregeln nach einem *b* keine Möglichkeit mehr bieten, das Literal *a* zu produzieren.

Die obige Grammatik ist durch die Alternativen *aZ* und *bB* eine *rechtsrekursive* Grammatik, der "Aufruf" von anderen Produktionsregeln steht auf der rechten Seite. Produktionen der Form *Za* und *Bb* wären *links-rekursiv*. Diese Unterscheidung hat erhebliche Auswirkung für Parser und Parsergeneratoren.

Für eine Grammatik, die links-rekursive Produktionen enthält, läßt sich kein Top-Down-Parser erstellen. Ein Top-Down-Parser arbeitet sich von der abstrahierten Produktionsregel auf hohem Level(Top) zu den Literalen(Bottom) durch. Ein Bottom-Up-Parser arbeitet sich von den Literalen zu den auf hohem Level angesiedelten Produktionsregeln durch.

In dieser Arbeit wird ein Parsergenerator verwandt, der Top-Down-Parser generiert. Deshalb darf die verwandte Grammatik keine links-rekursionen enthalten.

Eine tiefgehende Betrachtung dieses Themas wäre zu umfrangreich, für weitergehende Informationen zu diesem Thema siehe [3] und [17].

3.2 ANTLR

Als Parsergenerator wird die Software *ANTLR*(*ANother Tool for Language Recognition*) verwandt. Es gibt für *ANTLR* eine Reihe von Beispielgrammatiken, unter anderem auch eine Grammatik für Python. Diese Grammatik ist in ihrem Ausgangszustand bereits in der Lage große Teile valider *Python*-Quellcode-Dateien einzulesen. Allerdings akzeptiert diese Grammatik auch Code, der kein valides Python darstellt, da diese Grammatik weniger genau arbeitet, als die eigentlich Grammatik der Sprache *Python*.

Von diesem Ausganszustand aus, wird im Folgenden eine korrekte Form dieser Grammatik entwickelt. Dadurch ist *ANTLR* in der Lage einen Parser zu generieren, der die nötige Transformation von *Python*-Code in Objekte des Metamodells vornimmt.

Eine Grammatik für *ANTLR* besteht aus Regeln, diese werden unterschieden in Regeln für den Lexer und Regeln für den Parser.

Lexer-Regeln können z. B. folgendermaßen aussehen:

```
DIGIT: '0' .. '9';
INTEGER: DIGIT+;
LETTER: a .. z | A .. Z;
IDENTIFIER:  LETTER+;
TYPE: 'int' | 'string';
```

In der ersten Regel wird ein *Token* namens *DIGIT* defniniert, das aus einer Ziffer zwischen 0 und 9 besteht. Die zweite Regel beschreibt ein Token namens *INTEGER*, das aus einer Menge von *DIGIT*'s besteht. Der Operator + gibt hierbei an, das es sich um eine beliebige Zahl von *DIGIT*'s handeln kann, mindestens aber ein *DIGIT*. Ein Lexer, der auf Basis dieser Regeln erstellt wurde, würde die Zeichenfolge *123* zeichenweise als korrekt akzeptieren und die ganze Zahl als *Token* an der *Parser* übergeben. Die Zeichenfolge *12e3* würde allerdings nicht akzeptiert, der *Lexer* würde mit einer Fehlermeldung abbrechen, da diese Zeichenfolge nicht den defnierten Regeln entspricht. Da diese Regeln eine Sprache

beschreiben, sagt man, die Zeichenkette *123* ist ein gültiger Satz dieser Sprache und die Zeichenkette *12e3* ist kein gültiger Satz dieser Sprache. In der zweiten Regel beschreibt einen *IDENTIFIER*, dies könnte z. B. ein Variablenname in einer Programmiersprache sein.

Parser-Regeln bestehen aus Beschreibungen von Tokens und können andere Parser-Regeln aufrufen.

```
program: 'declare' decl+ 'begin' statement+ 'end';
decl: TYPE IDENTIFIER NEWLINE;
statement: expr NEWLINE;
```

Eine Gruppe der Parser sind die *LL(k)*-Parser, die Eingabe wird von Links nach Rechts gelesen und dabei jeweils die Parserregel, die am weitesten Links steht, als erstes verwandt. *LL(k)*-Parser unterscheiden sich durch die Anzahl der Tokens(k), die sie im voraus lesen müssen(*Look-Ahead*), um die richtige Regel auszuwählen. Ein *LL(1)*-Parser kann mit einem Token *Look-Ahead* die richtige Regel auswählen, ein *LL(2)*-Parser bräuchte zwei Tokens *Look-Ahead*. Die Anzahl der benötigten Tokens hängt von der Grammatik ab. In der folgenden Regel kann der Parser mit nur einem Token *Look-Ahead* nicht entscheiden, um welche Alternative es sich handelt.

```
decl: TYPE ID | TYPE ID '=' VALUE;
```

Das Zeichen / markiert Alternativen, die erste Alternative ist eine reine Deklaration, die Zweite weißt gleichzeitig einen Standardwert zu. Wenn der Parser das Token TYPE sieht, kann er nicht entscheiden, ob es um die erste oder die zweite Variante handelt. Bei einem *Look-Ahead* von zwei Tokens(*LL(2)*) wäre das immer noch nicht möglich, erst das dritte Token ermöglicht dies. Wenn das dritte Token ein = ist, dann handelt es sich um die zweite Variante, sonst um die Erste. Die obige Regel ist also *LL(3)*.

Die folgende Regel ist äquivalent, sie stellt aber den Zuweisungsteil als optional dar. Damit ergibt sich nicht die Wahl zwischen zwei Alternativen und der Parser kann mit einem Token Look-Ahead feststellen, wie weiter zu verfahren ist.

```
decl: TYPE ID ('=' VALUE)?;
```

ANTLR ist ein *LL(*)*-Parsergenerator. Die einzelnen Regeln können unterschiedliche Werte für k haben, also kann in der Grammatik festgelegt werden, wieviele Tokens *Look-Ahead* für diese Regel nötig ist. Das macht *ANTLR* sehr flexibel.

3.3 Transformation

Der generierte Parser allein kann nur feststellen, ob eine Eingabe korrekt im Sinne der Grammatik ist oder nicht. Um während des Einlesens die Obekte des Metamodells zu erstellen, müssen in die Grammatik *Actions* eingefügt werden. Eine *Action* ist ein Ausdruck in der Sprache, in welcher der Parser generiert wird. Während der Generierung wird dieser Code in den Parser eingefügt. Die folgende Regel stellt eine Klassendeklaration dar.

```
classdef: 'class' NAME suite;
```

In diese Regel muss folgendermaßen modifiziert werden:

```
classdef: 'class' n=NAME {createClassDefinition($n); } suite;
```

Das Token *NAME* wir hier einer Variable(n) zugewiesen, die später durch *$n* referenziert werden kann. Innerhalb der geschweiften Klammern steht Code den *ANTLR* in den generierten Parser kopiert. Dabei wird die Varible *$n* durch eine Referenz auf das Token *NAME* ersetzt.. Natürlich muß vorher die Methode *createClassDefinition()* defniert werden. Diese Methode erstellt dann ein entsprechendes Objekt aus dem Metamodell und übergibt dabei den Namen der zu deklarierenden Klasse.

Die Regel *suite* beschreibt im Falle der Python-Grammatik eine Liste von Anweisungen, den Rumpf der Deklaration. Die Liste muss der Klassendeklaration zugewiesen werden. *ANTLR* bietet hier eine Lösung an. Die Parser-Regeln können, ähnlich wie Funktionen, Argumente und Rückgabewerte haben. Die Regel *suite* muss also so angepasst werden, dass die eine Liste von *Statements* zurückgibt.

```
suite: simple_stmt | NEWLINE INDENT (stmt)+ DEDENT ;
```

Diese Regel wird so modifiziert, dass ihr Rückgabewert ein Objekt aus dem Metamodell ist.

```
suite returns [ArrayList<? extends Statement> statements]:
    s1=simple_stmt {statements.add($s1)}
    | NEWLINE INDENT (s2=stmt {statements.add($s2))+ DEDENT ;
```

Der Rückgabewert ist eine *ArrayList* aus den *Java Foundation Classes*(JFC), eine Liste variabler Größe. Der Parser weist die Rückgabewerte der Regeln *simple_stmt* und *stmt* den Variablen *s1* und *s2* zu. Diese enthalten dann *Statement*-Objekte. Der Inhalt der geschweiften Klammern ist Java-Code, der ausgeführt wird, sobald das davorliegende Parserelement zutrifft.

Die folgende Regel ist eine der möglichen Regeln, die durch *simple_stmt* aufgerufen werden können.

```
arith_expr returns [Expression expression]:
    t1=term op=(PLUS|MINUS) t2=term
    {$expression = createBinaryExpression($t1, $op, $t2); }
```

Diese Regel gibt ein *Expression*-Objekt zurück, das den entsprechenden Ausdruck repräsentiert.

Kapitel 4

Modelle und Diagramme

4.1 Transformationen

Wenn eine bereits existierende Quellcode-Datei eingelesen wird und sie in Objekte des Metamodells übertragen ist, können Transformationen an diesem Modell vorgenommen werden. Um die Objekte des Metamodells nach UML zu transformieren gibt es verschiedene Möglichkeiten, die im Folgenden vorgestellt werden. Beim Forward-Engineering muss diese Transformation dann in der Lage sein, UML in Objekte des Metamodells zu übertragen, eine zweite Transformation generiert dann den Quellcode auf Basis der Objekte des Metamodells.

Es gibt verschiedene Möglichkeiten, um diese Transformationen zu automatisieren, einige davon werden im Folgenden vorgestellt.

4.1.1 XSLT

Der einfachste Ansatz für die Modell-Transformation ist die Verwendung von *XSLT*. Dazu müssen die Objekte des Metamodells in *XMI*(*XML Metadata Interchange*) serialisiert werden, der generierte Code des Metamodells bringt diese Fähigkeit mit.

Wenn die Objekte des Metamodells in XML(XMI) vorliegen, können XSLT-Programme, sogenannte *stylesheets* die Transformationen übernehmen. XSLT ist eine Programmiersprache, die für die Transformationen von XML-Dokumenten geschaffen wurde.

Dies bedeutet aber, das XSLT nichts über die Zusammenhänge des Modells weiß, sondern nur die Struktur der XMI-Datei kennt. Die Ausgabe der Transformation ist dann valides XML, aber die Validität im Bezug auf das Zielmodell(UML) kann XSLT selbst nicht sicherstellen.

4.1.2 MOF-QVT

Die Object Management Group(OMG) hat einen Standard für MOF-Modell-Transformationen definiert(siehe [21]). *MOF-QVT*(*queries, views, transformations*) definiert ein deklarative und eine imperative Sprache, um MOF-Modelle zu transformieren.

Unter *queries* sind Anfragen zu verstehen, die an einzelne Elemente des Modells gerichtet werden, um diese auszuwählen. Für komplexere Anfragen gibt es

views, dies sind Sichten auf Mengen von Elementen, bzw. auf Teile des Modells. Die *transformations* setzen dann die Elemente der Modelle um, darauf kann aber auch verzichtet werden, wenn z. B. ein Modell nur auf Gültigkeit überprüft werden soll.

Wenn also das Metamodell in MOF definiert ist kann durch MOF-QVT eine Transformation der Metamodellinstanzen nach UML beschrieben werden, ebenso wie die Transformation von UML in Objekte des Metamodells.

Es gibt verschiedene Projekte, die Implementationen der MOF-QVT-Spezifikation bieten, z. B. die *Atlas Transformation Language*(siehe [13]). Sie wurde an der Université de Nantes entwickelt und ist zu einem Hauptbestandteil des Model-To-Model(M2M)-Projekts der Eclipse Foundation geworden(siehe [10]). M2M ist ein Unterprojekt des Eclipse Modeling Project(siehe [9]),

Das Tool MOFLON(siehe [7]), das an der TU Darmstadt entwickelt wurde, bietet die Möglichkeit mit einem grafischen Editor MOF-Modelle zu erstellen. Für so ein Modell kann MOFLON den Quellcode in Java generieren. Außerdem bietet MOFLON die Möglichkeit, auch Transformationen zwischen Metamodellen mit einem grafischen Editor zu modellieren. Hierbei werden sogenannte *Triple Graph Grammars* verwandt. Diese verwenden drei Graphen, um eine Transformation durchzuführen, diese bestehen aus dem Ausgangs- und dem Zielgraphen, sowie aus einem Graphen, der die Änderungen am Ausgangsgraphen darstellt. Eine detailierte Beschreibung dieser Technik würde den Rahmen dieser Arbeit sprengen, weiter Details finden sich unter [18].

Im Rahmen dieser Arbeit wurde versucht, mithilfe von MOFLON ein Metamodell zu erstellen und die Transformation nach UML mittels TGG zu bewerkstelligen. Leider war dies aus verschiedenen Gründen nicht möglich. Grundsätzlich ist anzumerken, dass zum Zeitpunkt dieser Arbeit die aktuelle MOF-Spezifikation [20] noch fehlerbehaftet ist und die folgende Version des Standards noch noch nicht endgültig verabschiedet ist. Die Implementationen der Tools für diesen Standard versuchen daher, die Fehler der Spezifikation soweit wie möglich selbst zu beseitigen, es werden also evtl. Lösungen angeboten, die nicht dem Standard entsprechen.

Bei MOFLON sind die Transformationen mittels TGG zwar beschreibbar, um diese aber auszuführen, wird ein Tool benötigt, das noch nicht in MOFLON integriert ist. Dieses Tool befindet sich noch in einem sehr frühen Entwicklungsstadium und wird von den Entwicklern nicht so ohne weiteres zur Verfügung gestellt.

Die Möglichkeiten des MOFLON-Tools sind aber sehr interessant, eine weitere Beobachtung der Entwicklung in der nahen Zukunft ist in diesem Bereich sicherlich interessant.

4.1.3 Anmerkungen

Aufgrund der aufgetretenen Schwierigkeiten bei dem Versuch, automatisierte Transformationen zu verwenden, sollen hier einige Anmerkungen zusammengetragen werden.

Die Automatisierung der Transformationen sollte eigentlich den Zweck erfüllen, im Vergleich zu einer "handgeschriebenen" Transformation, den Aufwand zu verringern und durch theoretische Grundlagen die Richtigkeit und Vollständigkeit dieser Transformation gewährleisten. Die theoretischen Grundlagen der

Transformationsarten aus den vorhergehenden Abschnitten näher zu betrachten, würde den Rahmen dieser Arbeit deutlich überschreiten.

Es sollte also wohl überlegt sein, ob der Einsatz solcher automatisierten Transformation nötig ist. Hierzu muss festgestellt werden, welcher Nutzen sich daraus ergibt und natürlich welcher Aufwand dafür nötig ist.

Der Aufwand für den Einsatz von automatisierten Transformationen sollte nicht unterschätzt werden. Im Gegensatz zum Stand der benötigten Tools zum Zeitpunkt dieser Arbeit, wird der Aufwand in Zukunft sicherlich sinken und die Tools an Funktionsumfang und Benutzerfreundlichkeit zunehmen. Am Beispiel von MOFLON zeigt sich aber z. B., dass so ein Tool sehr umfangreich ist, einiges an Komplexität birgt und somit unter Umständen nicht leichtfertig in einem Projekt eingesetzt werden sollte.

Wenn ein Projekt einen gewissen Umfang überschreitet, so wird sicherlich die Automatisierung schnell interessant. Anzumerken ist aber, dass die Transformationen trotzdem defniniert werden müssen. Im Falle von MOFLON's TGG geschieht dies auf einem sehr hohen Abstraktionslevel, unterstützt durch grafische Werkzeuge, was einen gewissen Komfort mit sich bringt. Andere Transformationssprachen müssen allerdings "von Hand" geschrieben werden, es muss also mindestens eine, evtl. sehr umfangreiche Sprache erlernt werden.

Ein weiterer Aspekt solcher Tools ist, dass für bestimmte Zwecke Quellcode generiert wird. Dieser Quellcode enthält im Falle von MOFLON in jeder Quellcodedatei einen Lizenz-Kopf, der darauf verweist, dass dieser Code unter der *GNU Lesser General Public License* (siehe [12]) steht. Die rechtlichen Folgen für ein Projekt hier zu erörtern, würde ebenfalls den Rahmen dieser Arbeit sprengen, es sei aber darauf hingewiesen, dass solche Lizenzen und ihre Auswirkungen auf ein Projekt natürlich vor dem Einsatz geprüft werden müssen, insbesondere bei der Veränderung und Verbreitung von Quellcode und dessen Kompilaten.

4.2 UML Diagramme

4.2.1 Klassendiagramme

Um die Informationen, die der Parser in Objekte des Metamodells übertragen hat, in Klassendiagramme umzusetzen, sind verschiedene Schritte möglich. Aufgrund der aufgetretenen Probleme bei der Erstellung von automatischen Transformationen, werden diese Schritte im weiteren Verlauf der Arbeit nicht automatisiert. Es wird versucht, die Schritte zu analysieren und in der Implementation umzusetzen. Inwieweit diese Transformationen also durch die Möglichkeiten aus dem vorhergehenden Abschnitt möglich sind, kann nicht gezeigt werden. Es können aber die notwendigen Einzelheiten beschrieben werden.

Klassen in UML können in Paketen angeordnet werden, um ihren Zusammenhang zu verdeutlichen. Das entsprechende Gegenstück zu einem Paket in UML ist in Python ein *module*. Grundsätzlich ist jede Python-Datei ein *module* und somit auch ein Namensraum für die enthaltenen Elemente. Außerdem kann ein Verzeichnis im Dateisystem ein *module* sein, hierfür muss es eine Datei enthalten, die den Namen *__init__.py* trägt.

Wenn also eine Quellcode-Datei eingelesen wird, gibt der Dateiname das Paket vor, in das die Elemente gehören. Zusätzlich muss für das Verzeichnis, in dem die Datei liegt und für die Verzeichnisse darüber festgestellt werden, ob

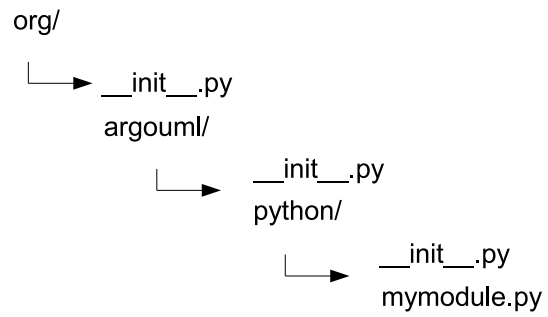

Abbildung 4.1: Ein Verzeichnisbaum

eine Datei namens __init__.py existiert. Falls das der Fall ist, dann muss das neue Paket als Element eines weiteren Pakets betrachtet werden.

Die Datei *mymodule.py* in Abbildung 4.1 liegt zusammen mit einer Datei namens *__init__.py* im Verzeichnis *python*. Das Verzeichnis *python* liegt wiederum im Verzeichnis *argouml* und dieses im Verzeichnis *org*. Wenn nun die Datei *mymodule.py* importiert wird, muss ein UML-Paket erstellt werden, dass *mymodule* heisst und die Elemente aus dieser Datei aufnimmt. Der vollständige Namensraum dieses Pakets ergibt sich aber aus dem Verzeichnis im dem die Datei liegt, da hier eine Datei *__init__.py* liegt und es sich beim dem Verzeichnis *python* dadurch ebenfalls um ein Modul handelt. Dies gilt auch für die beiden darüberliegenden Verzeichnisse. Sofern im Verzeichnis über *org* keine *__init__.py*-Datei zu finden ist, braucht der Verzeichnisbaum nicht weiter nach oben traversiert werden. Es ergeben sich also drei Pakete durch die Verzeichnisstruktur und ein Paket durch die zu importierende Datei selbst.

Die Datei *__init__.py* kann ebenfalls normalen Python-Code enthalten, dieser wird ausgeführt, wenn das Modul zur Laufzeit importiert wird. Sie wird häufig dazu benutzt, einzelne Untermodule zu importieren und diese durch eine spezielle Anweisung wieder zu exportieren. So könnte das Modul python aus dem obigen Beispiel eine Klasse aus mymodule.py importieren und diese wieder exportieren. Die Klasse wäre dann über den Namensraum *org.argouml.python* erreichbar, die Angabe des Untermoduls *mymodule* bräuchte also nicht angegeben werden.

Dieses exportieren kann natürlich auch in Abhängigkeit zu einer Bedingung(*if-then-else*) oder eines *try-except*-Blocks stattfinden, es würde dann erst zur Laufzeit feststehen, welche Elemente durch das Modul exportiert werden. Es bleibt also nur die Möglichkeit, das Exportieren ganz zu ignorieren, oder aber die gesamte *__init__.py*-Datei nach Elementen zu durchsuchen, die *möglicherweise* exportiert werden.

Um Informationen über die definierten Klassen in einem Modul zu erhalten, werden die Objekte der Klasse ClassDefinition, die in dem Modul gefunden wurden, befragt. Wie schon in Abschnitt 2.4.6erwähnt, lassen sich die Basisklassen nicht so einfach ermitteln.

Die Angabe der Basisklassen ist nicht einfach eine Liste von Bezeichner, sondern kann auch Ausdrücke enthalten. Diese Angaben werden, ähnlich wie bei den Modulen, erst zur Laufzeit festgelegt. Es können also, sofern ein Ausdruck vorliegt, nur die *möglichen* Basisklassen ermittelt werden. Dies kann natürlich Einfluss auf die Methoden haben, die vererbt werden.

In Java gibt es das Schlüsselwort *abstract*, das festlegt, ob eine Methode überschrieben werden muss. Wird diese Methode nicht überschrieben, dann wird der Compiler mit einem Fehler abbrechen. Ähnliches gilt für C++ und das Schlüsselwort *virtual*. In Python gibt es solche Beschränkungen nicht. Um ein ähnliches Verhalten zu erzeugen, kann die Basisimplementation eine Exception vom Typ *NotImplementedError* auslösen, dies geschieht aber wiederum erst zur Laufzeit und auch nur wenn die Methode aufgerufen wird.

Im ungünstigsten Fall können also das Verhalten(die Methoden) und auch die Attribute einer Klasse zur Laufzeit sehr unterschiedlich aussehen. Dies wird zusätzlich durch die dynamischen Fähigkeiten von Python bedingt, da Attribute nicht deklariert werden, sondern erst zur Laufzeit an eine Instanz oder eine Klasse gebunden werden. Dies passiert in vielen Fällen im Konstruktor, es muss aber nicht zwingend der Fall sein. Es müssen also alle Anweisungen des Konstruktors nach Zuweisungen der Form

```
self.x = ...
```

durchsucht werden. Diese Anweisungen können dann als Definitionen von Instanz-Attributen interpretiert werden. Sofern es sich bei der rechten Seite der Zuweisung nicht um einen zu komplizierten Ausdruck handelt, sondern z. B. um eine numerische Konstante oder eine Zeichenkette, kann auch ein Typ für das entsprechende Attribut erkannt werden- Da das Attribut aber nicht an diesen Typ gebunden ist, sondern auch Werte anderen Typs aufnehmen kann, ist es fraglich ob diese Interpretation sinnvoll ist. Ebenso schreibt UML nicht zwingend die Angabe eines Typs für ein Attribut vor, diese Information kann also auch ignoriert werden.

Der Klassenrumpf(außerhalb von Methoden) sollte ebenfalls nach Zuweisungen durchsucht werden. Hierbei kann es sich um Klassenattribute, oder sogar um spezielle Varianten von Methoden handeln(siehe Abschnitt 2.4.7). Außerdem kann der Rumpf auch Schleifen, *try-except*-Blöcke, usw. enthalten, eine sinnvolle Interpretation für die Transformation nach UML ist dann kaum möglich.

Das folgende Beispiel zeigt eine Klassendefinition.

```
class MyClass(object):
    my_class_attribute = 5
    my_unknown = someFunction()
```

Aus den Objekten des Metamodells lassen sich die binären Ausdrücke extrahieren. Im Falle von *my_ class_ attribute* ist auf der rechten Seite der Zuweisung ein Wert angegeben, damit steht eindeutig fest, dass es sich bei my_ *class_ attribute* wirklich um ein Attribute handelt. Im Falle von *my_ unknown* ist das nicht so einfach. Der Rückgabewert von *someFunction()* entscheidet, ob es sich um ein normales Attribute handelt. Es ist aber möglich, dass diese Funktion wiederum eine Funktion zurückgibt, die als Klassenmethode gedacht ist. Es müsste also der Typ des Rückgabewertes von *someFunction()* ermittelt werden, dies ist unter Umständen nicht möglich. Nur im Falle von *staticmethod*, usw. (siehe

2.4.7) läßt sich der Typ des Rückgabewertes ermitteln, da die Funktion vorher bekannt ist. Wenn die Transformation also nicht entscheiden kann, ob es sich um ein Attribut oder eine Funktion handelt kann sie die Anweisung also nur ignorieren, oder ein Attribut erstellen, wobei die Gefahr besteht, das dies nicht korrekt ist.

4.2.2 Automaten

Um die konkrete Implementation einer UML-Klasse, bzw. ihrer Methoden in UML abzubilden, gibt es verschiedene Möglichkeiten. Eine einfache Lösung wäre, eine Notiz mit einem speziellen *stereotype*(z. B. $<<$*implementation*$>>$) an eine Methode anzuhängen und als Inhalt dieser Notiz den gesamten Quellcode einer Methode in der jeweiligen Zielsprache, in diesem Falle Python. Damit wären die konkreten Implementationen nur *sprachspezifisch* vorhanden.

Im Folgenden soll der Versuch gemacht werden, die Implementation einer Methode in UML auszudrücken und sie dadurch in einer möglichst *sprachunspezifischen* Form darzustellen. Dies ist natürlich nur bis zu einem gewissen Grad möglich. Es wird darum versucht, nur die gängigen Konstrukte wie Schleifen, *if-then-else*-Blöcke, usw. in UML auszudrücken.

Als Modellelemente für diese Aufgabe wurde die UML 1.4 Statemachine Charts ausgewählt. In UML 1.4 gibt es *Statemachine Charts* um das Verhalten von verschiedenen Elementen zu beschreiben, dazu gehören auch Methoden. In dieser Arbeit wird versucht, die Implementationen von Methoden soweit wie möglich durch solche *statemachines* darzustellen. Bei diesen *statemachines* handelt es sich um endliche Automaten, diese werden im folgenden Abschnitt näher erläutert. Wie die Spezifikation von diesen Automaten in UML 1.4 aussieht wird in Abschnitt 4.2.2.2 kurz beleuchtet, die vollständige Spezifikation findet sich unter [15], Kapitel 2.12. Im Abschnitt 4.2.2.3 werden dann die Möglichkeiten zur Abbildung von Implementation unter Verwendung dieser Automaten behandelt.

4.2.2.1 Endliche Automaten

Ein endlicher Automat ist ein mathematisches Modell eines Systems. Ein System wird durch eine endliche Menge von Zuständen beschrieben, die auch als "Konfigurationen" bezeichnet werden können. Ein endlicher Automat befindet sich immer in genau einem dieser Zustände. Der Automat hat eine endliche Menge von Eingabesymbolen, eine endliche Menge von Ausgabesysmbolen, einen Anfangszustand und eine Menge von Endzuständen. Zusätzlich wird eine Menge von Kanten beschrieben, diese führen von einem Zustand zu sich selbst oder zu einem anderen Zustand. Erhält der Automat ein Eingabesymbol entscheidet der aktuelle Zustand und die von dort ausgehenden Kanten anhand des Symbols den Folgezustand. Die Kanten repräsentieren also die Übergangsfunktion.

Die Menge der Zeichenketten, die durch einen Automaten akzeptiert werden, bezeichnet man als Sprache. Wenn eine Sprache von einem endlichen Automaten abgebildet werden kann, so ist dies ein *reguläre* Sprache. Die hier beschriebenen Programmiersprachen sind nicht *regulär*, es kann also nicht erwartet werden, das diese mit den Automaten aus UML 1.4 vollständig abbildbar sind. Trotzdem sollte sich eine gute Näherung ergeben, bzw. sollten sich die grundlegenden Konstrukte abbilden lassen.

Eine vollständige Betrachtung der Automatentheorie ist hier nicht möglich, für eine detailiertere Beschreibung diese Themas siehe [17].

Im folgenden wird nicht der Zustand eines Objekts betrachtet, sondern die Zustände ergeben sich aus den Anweisungen einer Programmiersprache. Die Kanten und damit die Folgezustände ergeben sich aus einer Bedingung, z. B. bei einer *if*-Anweisung, oder einfach aus der Tatsache, das die aktuelle Anweisung im Programmfluss abgearbeitet wurde. Das Eingabesymbol ist dadurch nicht explizit vorhanden, implizit ergibt es sich aus dem Kontrollfluss eines Programms, wenn es ausgeführt würde.

Aus dieser Betrachtung heraus ergibt sich die Möglichkeit, den Kontrollfluss eines Programms schematisch in Form eines endlichen Automaten abzubuilden.

4.2.2.2 Automaten in UML 1.4

Die Kanten in einem UML 1.4 Automaten werden als *Transitions* bezeichnet. Diese *Transitions* können über einen sogenannten *Guard* verfügen, einen Bool'schen Ausdruck, der festlegt, ob dieser Übergang möglich ist oder nicht. Die Spezifikation von UML 1.4 sieht verschiedene Zustandstypen vor, die allesamt als *Vertices* bezeichnet werden:

PseudoState Beschreibt einen "flüchtigen" Zustand, einen Zustand mit einer bestimmten Funktion, der aber nur als Übergang dient. Es wird ein bestimmter Typ zugeordnet, der das genaue Verhalten eines *PseudoStates* definiert. Die möglichen Typen sind u. A.:

- **initial** Mit diesem Typ beschreibt dieser PseudoState einen Anfangszustand, der keine eingehenden Kanten erlaubt
- **choice** Kann mehrere eingehende Kanten zusammenführen. Bei den ausgehenden Kanten werden die *Guards* ausgewertet. Wenn der Bool'sche Ausdruck eines *Guards* einen wahren Wert ergibt, ist dieser Übergang erlaubt. Wenn dies für mehrere ausgehende Kanten der Fall ist, wird nur eine davon ausgewählt. Wenn die *Guards* aller Kanten zu einem unwahren Wert evaluieren, gilt das Modell als nicht Korrekt, darum sollte ein *Guard* namens *else* vorhanden sein
- **fork** Ermöglicht das Aufteilen in mehrere nebenläufige Pfade
- **join** Führt nebenläufige Pfade zusammen

SimpleState Ein "normaler" Zustand. Dieser kann verschiedene Attribute haben., z. B.:

- **entry activity,** eine Anweisung die ausgeführt wird, wenn dieser Zustand eintritt
- **do activity,** eine Anwweiung die ausgeführt wird, wenn dieser Zustand eingetreten ist
- **exit activity** eine Anweisung die ausgführt wird, bevor der Zustand verlassen wird

CompositeState Ein zusammengesetzter Zustand, oder ein Container, der alle anderen Arten von *Vertices* enthalten kann, also auch vollständige Automaten. Jeder Automat in UML 1.4 ist ein *CompositeState*.

FinalState Beschreibt einen Endzustand. Ein FinalState kann keine ausgehenden Kanten haben.

4.2.2.3 Abbildung von Implementationen mit endlichen Automaten

Der Eintritt in eine Methode wird durch einen *PseudoState* vom Typ *initial* dargestellt, dies entspricht dem Startzustand. Das Ende einer Methode ist dann der Endzustand, in UML 1.4 Terminologie ein *FinalState*.

Um die einzelnen Anweisungen innerhalb einer Methode abzubilden, gibt es nun verschiedene Möglichkeiten. Wenn man einen Automatenzustand als einen Zustand einer Instanz betrachtet, so würde sich der Zustand ändern, wenn sich die Attribute der Instanz ändern, z. B. über den Aufruf einer Methode, die einen Attributwert verändert. Vorausgesetzt, die Attribute einer Instanz sind von aussen nur über Methoden erreichbar(z. B. *getter*- und *setter*-Methoden), dann wäre ein Methodenaufruf also ein Eingabesymbol für den Automaten, anhand dessen festgestellt wird, welcher Folgezustand angesprungen wird. Da die Methoden das Verhalten eines Objektes darstellen, ergibt sich der Zustand eines Objekts aus seinen Attributen. Wenn ein Objekt also nur einen Bool'schen Wert als Attribut hat, kennt das Objekt nur zwei Zustände: entweder ist das Bool'sche Attribut wahr oder nicht wahr. Wenn nun ein zweiter Bool'scher Wert als Attribut hinzukommt, ergeben sich schon einige Zustände mehr, nämlich $2^2 = 4$ Zustände. Wenn aber ein Ganzzahliges Attribut hinzukommt, z. B. ein 32bit Integer, so würden genaugenommen nochmal 2^{32} mögliche Zustände hinzukommen. Ein Automat mit einer solchen Anzahl an Zuständen ist für ein Diagramm kaum sinnvoll. Eine Ausnahme hierzu würde ein expliziter Zustand eines Objektes sein, z. B. bei einer Netzwerkanfrage mit den Zuständen *processing*, *finished*, *error*, usw.

Um trotzdem zu einer sinnvollen Abbildung zu kommen, werden im Folgenden einzelne Anweisungen als Zustände betrachtet. Die Zustandswechsel symbolisieren also den Programmfluss, der sich durch die Anweisungen ergibt. Wird ein Zustand angesprungen, bedeutet das, die entsprechende Anweisung wird ausgeführt, danach erfolgt der Wechsel zum nächsten Zustand(der nächsten Anweisung). Dadurch läßt sich der Programmfluss visualisieren.

Bei einfachen Anweisungen ergibt sich der Folgezustand aus der darauffolgenden Anweisung. Bei einem if-then-else-Block erfolgt der Zustandswechsel anhand der Auswertung des Ausdrucks im *if*-Teil, also wird zum *then* gesprungen, zu weiteren *else-if*-Blöcken oder zu einem *else*-Block.

Für zusammengesetzte Anweisungen, bzw. Anweisungsblöcken wie Schleifen, *if-then-else*-Blöcken, usw. bieten sich dann *CompositeStates* an. Hierbei entsteht aber relativ schnell ein Problem in Verbindung mit der Anweisung *return*, dies soll folgendes Beispiel zeigen.

```
def aMethod(self, b):
    if b:
        return 5
    return -5
```

Der Automat dieser Methode würde einen *CompositeState* enthalten, der den *if-then-else*-Block(hier ohne *else*-Teil) darstellt und einen *SimpleState* der die Anweisung "*return -5*" darstellt. Letztere könnte als Endzustand dieser Methode betrachtet werden, da er deren letzte Anweisung ist. Da ein *CompositeState*

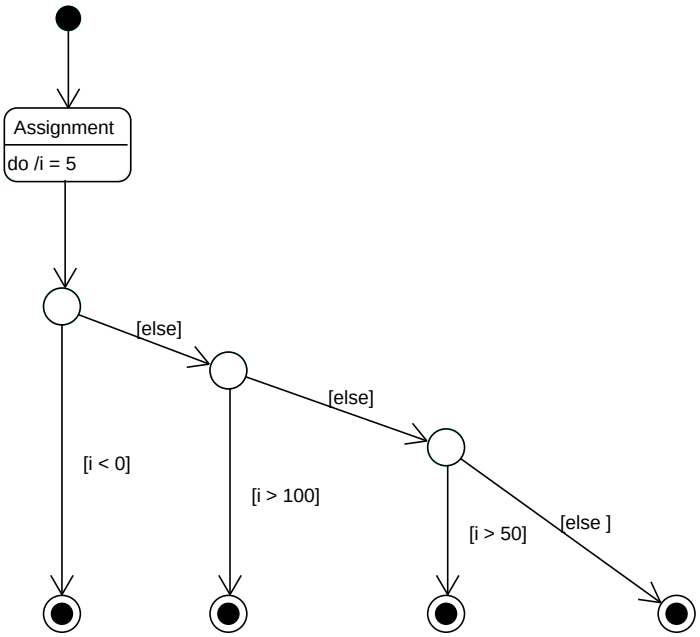

Abbildung 4.2: Ein if-then-else-Statement als Automat

wiederum einen Automaten darstellt, wäre die Anweisung "*return 5*" der Endzustand dieses eingebetteten Automaten. Der Folgezustand des *CompositeStates* wäre die Anweisung "*return -5*". Dies entspricht aber nicht dem Kontrollfluss eines Programms, da er durch die *return*-Anweisung sofort die Methode verlassen würde, gegebenfalls erst nach Auswertung des Ausdrucks, welcher hinter dem *return* steht, in diesem Falle eine numerische Konstante. Für beide *return*-Anweisungen gilt also, dass sie entweder die Endzustände dieser Methode darstellen, oder für einen direkten Sprung zu einem expliziten Endzustand sorgen müssen. Wenn aber die Anweisung "*return -5*" innerhalb eines *CompositeStates* ist, kann kann sie weder ein Endzustand des äußeren Automaten sein, noch kann von dort ein direkter Übergang zu einem Zustand außerhalb des *CompositeStates* erfolgen.

Von der Verwendung von *CompositeStates* für Anweisungsblöcke, bzw. Unterklassen der Klasse *CompoundStatement* aus dem Metamodell muss also abgesehen werden.

Abbildung 4.2 zeigt folgendes Quellcode-Fragment als Zustandsautomat:

```
i = 5
if i < 0:
    return “negative”
elif i > 100:
    return “bigger than 100”
elif i > 50:
    return “bigger than 50”
else:
    return “between 0 and 50”
```

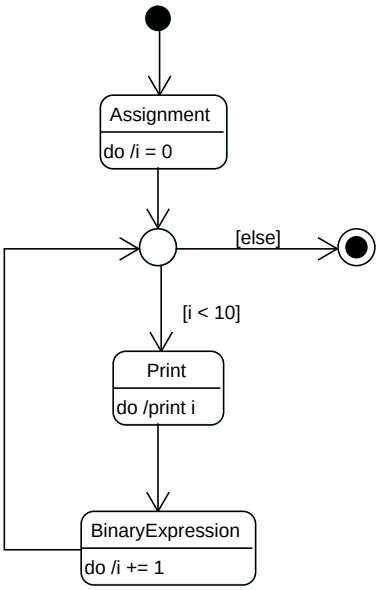

Abbildung 4.3: Eine *while*-Schleife als Automat

Da diese Automaten hier als Darstellung der Implementationen von Methoden verwandt werden, enthält das Quellcode-Fragment *return*-Anweisungen. Der Automat beginnt mit einem *InitalState*, danach folgt ein *SimpleState*, der die Zuweisung darstellt. Die Zuweisung $i = 5$ wird als eigener Zustand dargestellt. Er enthält eine *DoActivity*, also eine Anweisung, die ausgeführt wird, während der Automat in diesem Zustand ist. Die Anweisung $i = 5$ ist hier aus dem Quellcode übernommen und ist daher in der Sprache Python notiert. Danach folgt ein *Choice*-Zustand, der die Bedingung $i < 0$ repräsentiert. Die eigentliche Bedingung steht als *Guard* an einer der ausgehenden Kanten. Wenn diese Bedingung nicht zutrifft, wird die Kante ausgewählt, die den *else*-Guard enthält, dies ist aber nicht das *else* aus dem Code-Fragment. Die beiden folgenden *Choices* symbolisieren die *elif*-Zweige. Wichtig ist dabei, dass die *elif*-Zweige in dieser Form aneinandergehängt sind. Würden die Übergänge alle am ersten *Choice*-Zustand untergebracht sein, würde die Reihenfolge dieser *elif*-Zweige verloren gehen. Dies gilt auch für den *else*-Teil des Code-Fragements, dieser ist also als *else-Guard* am letzten *Choice*-Zustand untergebracht. Die *FinalStates* am unteren Rand repräsentieren die *return*-Anweisungen. Im Diagram nicht sichtbar sind die Werte, die von den *return*-Anweisungen zurückgegeben werden sollen. Diese sind aber als *Action* angehängt und enthalten die Ausdrücke in Python-Notation. Die Ausdrücke bestehen hier lediglich aus konstanten Zeichenketten.

Das folgende Quellcode-Fragment zeigt eine *while*-Schleife:

```
i = 0
while i < 10:
    print i
    i += 1
```

Der entsprechende Automat ist in Abbildung 4.3 dargestellt, die *while*-

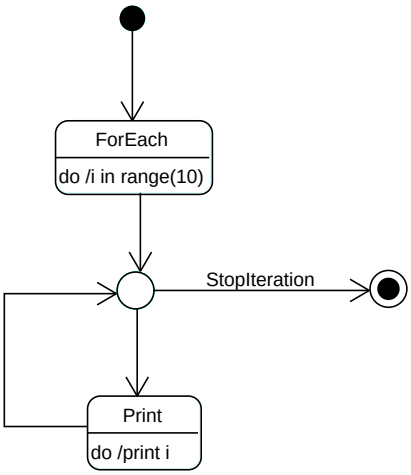

Abbildung 4.4: Eine *for*-Schleife als Automat

Bedingung ist hier als *Guard* untergebracht. Diese Form der Schleife ist sehr gut abzubilden, anders sieht es mit einer *for*-Schleife aus. Eine *for*-Schleife in Python ist nicht mit einer for-Schleife in C++ zu vergleichen, sie entspricht eher einer Sonderform der *for*-Schleife in Java. Eine *for*-Schleife in C++ und in der Grundform auch in Java, hat eine explizite Abbruchbedingung, dies ließe sich sehr einfach durch einen Guard abbilden. In Python allerdings ist eine for-Schleife eher mit einem *foreach* in PHP zu vergleichen. Das folgende Beispiel zeigt eine for-Schleife in Python:

```
for i in range(10):
    print i
```

Nach dem Schlüsselwort *for* folgt die Iterationsvariable, dies kann auch eine Liste von Variablen sein. Nach dem Schlüsselwort *in* folgt ein Ausdruck, der ein Objekt zurückgibt, der vom Typ *iterable* abgeleitet ist. Die Klasse iterable definiert eine Methode next(), und gibt das nächste Element der Iteration zurück. Sollte kein Element mehr vorhanden sein, wird intern eine *Exception* vom Typ *StopIteration* erzeugt, welche der Schleife signalisiert, dass sie am Ende der Iteration angelangt ist. Der Programmierer muss diese *Exception* nicht behandeln, der entsprechende *try-except*-Block verbirgt sich hinter der *for*-Schleife. Durch diesen Mechanismus hat eine for-Schleife in Python also keine explizite Abbruchbedingung, die Funktion *range(10)* gibt in diesem Falle ein iterierbares Objekt zurück, das sukzessive die Werte von 0 bis 9 zurückgibt. Diese werden dann der Iterationsvariable zugewiesen. In Java gibt es diese Form der for-Schleife seit der Version 1.5 auch.

Abbildung 4.4 zeigt die Umsetzung dieses Schleifenkonstrukts in einem Automaten. Da hier keine Bool'schen Abbruchbedingungen existieren, können keine *Guards* eingesetzt werden, es wurde ein Hilfskonstrukt gewählt. An eine der Kanten wurde ein Signal mit dem Namen *StopIteration* angefügt, um zu symbolisieren, dass hier das Schleifenende ist. Diese Lösung ist sehr spezifisch für Python, aber sie hält den Zustandsautomaten sehr übersichtlich. Es könnte auch kompliziertere Konstrukte gewählt werden, z. B. könnte eine Zuweisung des *ite-*

rables an eine neue Variable erfolgen, der Aufruf *next()* und die Behandlung der *StopIteration*-Ausnahme explizit darzustellen. Dies würde aber den Automaten verkomplizieren und wird daher zugunsten der Übersichtlichkeit verworfen. Anzumerken ist, dass die Verwendung eines *SignalEvents* ist an dieser Stelle zwar übersichtlich ist, aber die UML-Spezifikation diesen Event-Typ als *asynchron* bezeichnet, in diesem Falle aber nicht so verwendet wird.

Kapitel 5

Quellcodegenerierung

Die Modellelemente und ihre Beziehungen müssen bei einer Quellcodegenerierung sinnvoll umgesetzt werden. Ebenso müssen Strukturen im Dateisystem erzeugt werden, welche die Pakete der entsprechenden Komponenten repräsentieren. Beim *reverse engineering* von Quellcode in dieser Arbeit wurden nur Elemente für Klassendiagramme und Zustandsautomaten betrachtet, dies gilt auch für diesen Abschnitt. Die Implementation in Form des Python-Moduls für ArgoUML bekommt für die Quellcode-Generierung auch nur Klassen übergeben, dies sind entweder vom Benutzer ausgewählte Klassen oder alle Klassen des aktiven Projekts in ArgoUML. Die Zustandsautomaten sind für die Generierung trotzdem erreichbar, sofern sie, wie beim *reverse engineering* erzeugt, als Unterelement der entsprechenden Methoden vorhanden sind.

Eine allgemeine Anforderung an die Generierung von Python-Code ist die Kennzeichnung von Blöcken. Die Rümpfe von Klassendefinitionen, Funktionen/Methoden, Schleifen, usw. werden in Python durch Einrückungen gekennzeichnet. Es ist also sehr wichtig, dass der generierte Code sehr sauber formatiert ist. Dies dient nicht nur der Lesbarkeit, sondern ist sogar für die Korrektheit des Codes nötig. Für Beispiele hierzu wird auf die Beispiele im Abschnitten 7.2und dem Kapitel 2verwiesen.

5.1 Pakete

Wenn der Quellcode von Klassen generiert werden soll, müssen zuerst die Verzeichnisstruktur vorhanden sein, welche die Pakethierarchie darstellt. Ist dies nicht der Fall, müssen diese Verzeichnisse natürlich angelegt werden. Eine Python-Quellcode-Datei selbst ist ein Modul, bzw. ein Namensraum. Ein Verzeichnis ist genau dann ein Modul, wenn darin eine Datei namens *__init__.py* vorhanden ist(siehe Abschnitt 4.2.1).

In Java ist es zwingend notwendig, dass es nur eine Klassendefinition pro Datei gibt, außer es werden innerhalb dieser Klasse weitere Klassen definiert. In Python ist dies nicht notwendig.

Es gibt also verschiedene Möglichkeiten, um Paketstrukturen und die enthaltenen Elemente abzubilden. Es kann für jede Klasse eine eigene Datei erzeugt werden, ähnlich wie in Java. Dies hält den Inhalt dieser Dateien übersichtlich. Es wird aber ein zusätzlicher Namensraum erzeugt, da die Datei selbst ein Modul

ist. Wenn es also eine Klasse mit dem Namensraum *org.argouml.python.MyClass* gibt und für diese Klasse eine eigene Datei myclass.py erzeugt wird, dann ist der Namensraum der enthaltenen Klasse *org.argouml.python.myclass.MyClass*. Um die Klasse trotzdem im ursprünglichen Namensraum verfügbar zu machen, muss die Datei *__init__.py* im Verzeichnis *org/argouml/python/* die Klasse importieren und als verfügbares Element exportieren. Dies ist gängige Praxis in Python.

Die zweite Alternative wäre, nicht für jede Klasse eine eigene Datei zu erzeugen, sondern alle Klassen in einem Namensraum auch einen einer Datei zu speichern. Dies würde aber sehr unübersichtliche Dateien erzeugen, wenn ein Namensraum sehr viele Klassen enthält. Dies wäre zwar eine valide Lösung, aber zugunsten der Lesbarkeit, sollte der Aufwand der ersten Alternative in Kauf genommen werden.

5.2 Klassen und Klassendiagramme

Eine Klassendefinition in UML 1.4 enthält hauptsächlich Methoden und Attribute der Klasse, bzw. ihrer Objekte. Da Python eine objektorientierte Sprache ist, lassen sich diese Konzepte grundsätzlich abbilden. Bei näherer Betrachtung finden sich aber Dinge, die nicht unbedingt in Python abzubilden sind, evtl. nur durch Interpretation und auch Elemente die definitiv nicht abzubilden sind.

In Sprachen wie Java und C++ wird die Sichtbarkeit von Methoden und Attributen bei der Deklaration, bzw. bei der Definition festgelegt. Ein Attribut, das in Java mit der Sichtbarkeit *public* markiert ist, kann von anderen Klassen direkt ausgelesen und geschrieben werden. Bei einem Attribut, das als *private* deklariert ist, ist dies nur für Objekte der Klasse selbst möglich(oder nur für die Klasse im Falle eines Klassenattributs). Außerdem gibt es noch die Sichtbarkeit *protected*, welchen den Zugriff für die Klasse selbst und zusätzlich für Unterklassen erlaubt. Diese drei Sichtbarkeitsattribute gibt es auch in UML 1.4, sie gelten für Attribute und Methoden. Außerdem gibt es noch die Sichtbarkeit *package*, die den Zugriff nur für Objekte im selben Paket erlaubt.

Python unterscheidet nur zwischen den Sichtbarkeiten *public* und *private*. Außerdem werden diese Sichtbarkeiten nicht durch Schlüsselwörter festgelegt, sondern durch spezielle Namen der betroffenen Methoden, oder des Attributs. Der Zugriff auf ein privates Element ist, wenn auch etwas umständlich, trotzdem möglich, da dem Programmierer nur mitgeteilt werden soll, das der direkte Zugriff nicht erlaubt ist. Weder der Compiler noch der Interpreter unterbinden aber diesen Zugriff. Folgendes Beispiel zeigt eine Klasse mit einem privaten Attribut:

```
class MyClass(object):
    def __init__(self):
        self.__privateAttribute = "privat"
    def myMethod(self):
        print self.__privateAttribute
```

Im Konstruktor *__init__()* dieser Klasse wird ein Attribut namens *__privateAttribute* definiert, welches eine Stringkonstante enthält. Die Unterstriche am Anfang des eigentlichen Variablennamens legen nun die private Sichtbarkeit fest, sobald der Python-Compiler auf diese beiden Unterstriche stößt, wird

die Variable speziell behandelt. Die Methode myMethod() kann ungehindert auf diese Variable zugreifen, ein der Zugriff von außen würde mit einem *AttributeError* fehlschlagen, mit dem Hinweis, das ein Attribut mit dem Namen *__privateAttribute* nicht vorhanden ist. Python behandelt den speziellen Variablennamen, indem es die Variable umbenennt. Das Ergebnis ist in diesem Falle eine Variable mit dem Namen *_MyClass__privateAttribute*. Beim Zugriff innerhalb der Klasse wird dies erkannt und auf die umbenannte Variable umgelenkt, bei einem Zugriff von außen ist dies nicht der Fall. Dies gilt auch für Unterklassen der Klasse MyClass, sie können ebenfalls nicht direkt auf *__privateAttribute* zugreifen. Da der Zugriff aber nur durch die Umbenennung geregelt wird, kann der Programmierer das Attribut einfach unter dem Namen *_MyClass__privateAttribute* erreichen, weder der Compiler noch der Interpreter werden ihn daran hindern.

Für die Sichtbarkeit *protected* gibt es in Python keine entsprechende Möglichkeit. Es kann aber, in Anlehnung an den doppelten Unterstrich bei privaten Elementen ein einfacher Unterstrich vor den Variablen- oder Methodennamen gestellt werden, um zu kennzeichnen, dass der Zugriff von außen nicht vorgesehen ist. Diese Form wird von Python nicht speziell behandelt und kann somit nur als Hinweis für den Programmierer dienen.

Die Sichtbarkeit *package* kann in Python direkt nicht abgebildet werden, dies wäre nur durch speziellen Code möglich.

Die Benutzung von Unterstrichen ist in Python für Attribute als auch für Methoden möglich. Es ist allerdings zu beachten, das Methoden deren Namen mit einem doppelten Unterstrich beginnen, als auch mit einem doppelten Unterstrich enden, ein andere spezielle Bedeutung haben. So ist z. B. *__init__()* der Konstruktor einer Klasse, *__del__()* deren Destructor und die Methoden *__add__()* und *__sub__()* die Implementation der Operatoren + und -.

Für Instanzattribute generell gilt, das diese in Python nicht explizit zur Übersetzungszeit deklariert werden müssen. Ein Attribut kann zu einem beliebigen Zeitpunkt an eine Instanz zugewiesen werden. Bei der Generierung wird diese Zuweisung in den Konstruktor geschrieben. Sollte kein Ausdruck für den initialen Wert einer Variable vorhanden sein, bekommt dieser den Wert *None* zugewiesen. Eine Variablen-Deklaration ohne Zuweisung ist in Python nicht möglich.

Bei den Assoziationen zwischen Klassen gilt ähnliches, wie für die Attribute. Wenn eine Assoziation navigierbar ist, wird im Konstruktor ein entsprechendes Attribut generiert. Bei einer Assoziation mit einer Kardinalität, die Werte größer als eins annehmen kann, wird eine leere *list* zugewiesen. Dabei geht allerdings die Typinformation verloren, da der *list*-Typ in Python keine Vorgaben zu den enthaltenen Elementen macht.

5.3 Zustandsautomaten

In Abschnitt 4.2.2.3 wurde die Verwendung von Zustandsautomaten zur Abbildung von Methodenimplementationen dargestellt. Da die Zustandsautomaten in UML 1.4 ein abstraktes Konzept darstellen, ist die Generierung von Quellcode nur sinnvoll, wenn die Automaten in der Form angelegt sind, wie es in Abschnitt 4.2.2.3 beschrieben wurde. Ist dies nicht der Fall, lassen sich aus den Zustandsautomaten nicht Informationen gewinnen, die konkret genug sind,

um Quellcode daraus zu generieren. Dieser Abschnitt beschreibt also nur die Quellcode-Generierung dieser speziellen Automaten.

Um der Codegenerierung anzuzeigen, das hier ein Implementationsautomat vorliegt, sollte dieser speziell gekennzeichnet sein. Hierzu wird ein *stereotype* mit dem Namen *implementation* eingeführt, die nur auf Zustandsautomaten anwendbar ist. Diese Markierung könnte auch über einen speziellen Namen des Zustandsautomaten geschehen, die Lösung über einen stereotype ist aber sehr viel deutlicher für den Anwender. Wenn eine Methode allerdings über zwei Automatenelemente verfügt, die mit diesem stereotype versehen sind, ist für die Code-Generieren nicht zu entscheiden, wie damit zu verfahren ist, sie kann also nur einen der Automaten verwenden und eine entsprechende Warnung ausgeben.

Ähnliches gilt für die Schleifenkonstrukte innerhalb der Automaten. Ein Zustand kann den Namen "*ForEach*" tragen, um zu kennzeichnen, dass es sich um den ersten Zustand einer *for*-Schleife handelt, oder es kann ein *stereotype* namens *ForEach* verwendet werden, der denselben Umstand anzeigt. Wenn an dieser Stelle auch ein stereotype verwandt wird, besteht keine Möglichkeit mehr, den eigentlichen Namen des betroffenen Zustands abzubilden. Wenn der Benutzer also eine Information in diesem Namen untergebracht hat, geht diese bei der Generierung verloren.

Diese Probleme haben ihre Quelle in der Tatsache, das für die Zustandsautomaten eine Semantik eingeführt wurde, um Implementationen abzubilden. Diese Semantik kann aber vorerst nicht überprüft werden, da diese nicht formal vorhanden ist.

Eine mögliche Lösung des Problems wäre MOF. Es könnten Ableitungen der Automatenelemente gebildet werden, welche die Implementationssemantik überprüfen, bzw. deren Korrektheit sicherstellen. Dies stellt aber Anforderungen an den verwandten UML-Modellierer, denn dieser muss mit den erweiterten Elementen umgehen können. ArgoUML kann das bisher nicht, dieser Lösungsweg ist daher vorerst leider nicht möglich.

ArgoUML bietet die Möglichkeit zur *Kritik* an Modellen. Diese Kritiken sind Textmeldungen, die in einer Liste zusammengefasst werden und dem Nutzer angezeigt werden. Hierzu gehören z. B. die Hinweise, unbenannten Elementen einen Namen zu geben. Die Semantik der Automaten könnte also zumindest überprüft werden und ggf. dem Nutzer mitgeteilt werden, wenn eine nicht korrekte Benutzung vorliegt.

Es liegt dann an dem Benutzer, ob die semantischen Fehler behoben werden oder nicht, die Quellcode-Generierung muss also trotzdem eine Überprüfung durchführen. Diese Überprüfung ist aber nur bis zu einem gewissen Grad möglich, daher können auch noch bei der Übersetzung des Quellcodes und zur Laufzeit Fehler auftreten, die auf die nicht korrekte Verwendung der speziellen Form dieser Zustandsautomaten im UML-Modell zurückzuführen ist.

Kapitel 6

Fazit

Die vorliegende Lösung bietet einige Funktionen, um existierenden Code zur Analyse oder zur Dokumentation in UML 1.4 abzubilden und außerdem aus UML 1.4 Modellen Quellcode zu erzeugen.

Beim Erarbeiten dieser Lösung wurde sowohl einige Probleme aufgezeigt, die speziell mit der Sprache Python verbunden sind, als auch Probleme, die durch UML als Modellierungssprache für Software im Allgemeinen auftreten.

Ein wirkliches Round-Trip-Engineering müsste allerdings noch mehr leisten, als nur Modellelemente, bzw. Quellcode zu erstellen. Dieses müsste auch möglich sein, ohne ganze Elemente oder ganze Quellcode-Dateien bei Änderungen an ihrem entsprechendem Gegenstück nur zu erstellen. Dies stellt aber hohe Anforderungen an die Transformationen, den UML-Modellierer(ArgoUML) und letztlich auch an die Zielsprache(Python). Um die Auszuloten, wie weit es möglich ist, diese Anforderungen umzusetzen hätte den Rahmen dieser Arbeit gesprengt.

Zusätzlich wird durch den Informationsverlust, bzw. die Punkte an denen Information nur teilweise abbildbar ist, eine wirklich praktikable Lösung in Frage gestellt.

Die Abbildung von gängigen Programmierkonstrukten in Zustandsautomaten ist eine Lösung, die es ermöglicht, Programmierkonstrukte bis zu einem gewissen Grad auf eine abtrakte Ebene zu heben. Durch die Transformation in UML-Modellelemente lassen sich Implementationen mehr oder weniger anschaulich auf einem sprachunabhängigen Niveau darstellen. Dies funktioniert natürlich nicht ohne sprachspezifsiche Ausdrücke, eine sprachunabhängige Lösung ist dies also nicht. Trotzdem bietet diese Form der Transformation einige interessante Perspektiven. Die Transformation in andere Modellelemente, oder Diagramme(z. B. *Activity Charts* oder *Sequence Diagrams*) wären Möglichkeiten der weiteren Entwicklung.

Durch die Abbildung von Programmierkonstrukten wurden semantische Anforderungen gestellt, für welche die Zustandsautomaten in UML 1.4 ausgelegt sind. Die in Abschnitt 5.3 angesprochenen MOF-Erweiterungen würden einige semantische Probleme bereinigen, leider bleibt die Frage offen, wie weit eine solche Lösung tragen würde. Es wäre zu betrachten, ob die nötigen semantischen Vorgaben z. B. durch OCL-Ausdrücke zu realisieren wären.

Ebenfalls ungeklärt bleibt die Verwendung von MOF-Transformationen, da die entsprechenden Tools in dieser Arbeit nicht vollständig geprüft werden konnten. Anzumerken bleibt, das bei einem Metamodell, wie es in Abschnitt 2.4 vor-

gestellt wurde, die Transformation nicht ohne ein gewisses Maß an Komplexität zu bewerkstelligen ist.

Das vorliegende Metamodell ähnelt eher einem abstrakten Syntaxbaum eines Parsers, als den Elementen eines UML-Modells. Es könnte ein Metamodell gewählt werden, das "näher" an UML-Modellen liegt, bzw. dessen Elementen mehr ähnelt. Dies würde aber höhere Anforderungen an den Parser stellen, der mit dem Metamodell dieser Arbeit aber durchaus komplex genug ist.

Als weitere Alternative wäre es möglich, ein weiteres Zwischenmodell einzuführen, da dies die Transformationen vereinfachen würde und einer automatisierten Form der Transformation sicherlich entgegenkommen würde. Offen bleibt also die Frage ob und wie diese Formen der Transformationen durch Automatisierungen zu bewerkstelligen sind und welche Aussagen sich darüber durch die verwendeten Theorien und Algorithmen treffen lassen.

Kapitel 7

Software und Tools

7.1 ArgoUML

ArgoUML ist in Java implementiert und unterstützt Diagramme in UML 1.4. Es verfügt über verschiedene Module, um Quellcode auf Basis von Diagrammen zu generieren und Diagramme zur erstellen, in dem es Quellcode einliest. Es unterstützt bisher u.a. die Sprachen Java, C++, C# und SQL. Die Implementationen für diese Sprachen sind unterschiedlich ausgereift, einige unterstützen nur das *Forward Engineering*, andere nur das *Reverse Engineering*.

Das Einbinden von Modulen in ArgoUML[2] wird durch das Interface *ModuleInterface* ermöglicht. Eine Klasse, die dieses Interface implementiert und deren *.class*-Datei in einem Verzeichnis abgelegt ist, dass ArgoUML als Verzeichnis für Erweiterungen bekannt ist, wird beim Starten von ArgoUML initialisiert. Für das Importieren von Dateien, also das Einlesen von Quellcode muß das spezielle Interface *ImportModule* verwandt werden, das zu erstellende Modul muß also aus Sicht von ArgoUML aus zwei Modulen bestehen. Das Generieren von Code erfolgt explizit. Es gibt Menüpunkte um Quellcode für ausgewählte oder alle Klassen des Projektes zu generieren, das Modul bekommt dann die Klassen übergeben. Ebenso wird beim Importieren von Quellcode verfahren, dass Modul bekommt eine Liste von Dateinamen übergeben, die der Benutzer über einen Datei-Dialog ausgewählt hat.

7.2 Python

Python[11] ist eine objektorientierte Programmiersprache, die durch einen Compiler in Bytecode übersetzt wird und von einer *Virtual Machine* interpretiert wird. Ein etwas ungewöhnlicher Aspekt dieser Sprache ist, dass die Formatierung des Quellcodes eine wichtige Bedeutung hat. Während in anderen Sprachen Blöcke wie z.B. der Rumpf einer Methode oder der auszuführende Code bei einer Bedingung(*If-Then-Else*) durch das Einbetten in z.B. geschweifte Klammern(z.B. *Java* und *C/C++*) gekennzeichnet werden, geschieht dies in Python durch die Einrückungen am Zeilenanfang.

Beispiel:

```
class MyClass(object):
```

```
    def myMethod(self, a, b):
        if a > b:
            return a
        else:
            return b
```

Durch die dynamische Typisierung entstehen Probleme beim Generieren und Wiedereinlesen von Quellcode. Neben der Vorgehensweise der Objektorientierung bietet Python auch die

Möglichkeit der prozeduralen Programmierung. Diese wird zur Vereinfachung im folgenden Beispiel gewählt.

```
def myFunction(a, b = 0.0):
    return a + b
```

Die obige Funktion hat zwei Argumente. Der Typ des ersten Arguments ist nicht näher beschrieben, die Variable a nimmt also zur Laufzeit den Typ des übergebenen Wertes an. Die Variable b hingegen

läßt , dass es sich hier um eine Fließkommazahl handelt, da sie über einen Default-Parameter verfügt mit diesem Typ verfügt. Trotzdem kann zur Laufzeit ein anderer Typ übergeben werden. Ein Fehler entsteht erst dann, wenn der Operator $+$ für die Laufzeitwerte der Variablen a und b nicht definiert ist. Beispielsweise würde die Addition einer Zeichenkette und eines numerischen Wertes eine Exception des Typs *TypeError* auslösen. Die Bedingung eines UML-Designs *"Argument x muß vom Typ X sein"* kann also erstmal nicht direkt durch den Quellcode erfüllt werden, die Typinformation geht also bei der Generierung des Codes verloren. Bei einem erneuten Einlesen des Codes würde das UML-Modell eine Information wie *"der Typ des Argumentes x ist nicht definiert"*, wenn es dies direkt umsetzt, würde die Information auch im Modell verloren gehen.

Wenn also das UML-Modell Typinformationen für Argumente von Methoden enthält, aber keinen vorgegebenen Standardwert, geht diese Information beim Generieren des Quellcodes verloren. Umgekehrt kann die Typinformation aus dem Quellcode nur ermittelt werden, wenn in der Methodendeklaration ein Standardwert angegeben ist. Es läßt sich aber nicht ermitteln, ob diese Information auch wirklich ihre Gültigkeit behält, da an anderer Stelle im Code einfach ein Wert eines anderen Typs an die Variable zugewiesen werden kann.

Ein weiterer Aspekt sind Attribute der Klasse und der Instanzen. In Sprachen wie *Java* und *C++* werden diese immer mit der Klasse deklariert, zur Laufzeit können diese Attribute weder hinzugefügt, noch entfernt werden. In Python geschieht diese Deklaration erst zur Laufzeit. Die folgende Klasse hat eine spezielle Methode mit dem Namen *__init__(...)*. Methoden die mit einem doppelten Unterstrich beginnen, haben in Python eine besondere Bedeutung, im Falle von *__init__(...)* handelt es sich um den Konstruktor.

```
class MyClass(object):
    def __init__(self, a):
        self.a = a
    def myMethod(self, b):
        self.b = b
```

Im Konstruktor wird ein Instanzattribut *a* erzeugt und bekommt einen Wert zugewiesen, in der Methode *myMethod(...)* geschieht dasselbe. Der Unterschied ist, dass jede Instanz dieser Klasse zunächst ein Attribut namens *a* hat, ein Attribut namens *b* aber nur erhält, wenn die Methode myMethod aufgerufen wird. Falls diese Methode für eine Instanz niemals aufgerufen wird, so wird diese Instanz niemals ein Attribut *b* besitzen.

Außerdem lassen sich zur Laufzeit auch Attribute entfernen. Das ein Attribut im Konstruktor auftaucht ist also kein Garant dafür, dass dieses Attribut auch für diese gesamte Lebensdauer des Objekts existiert. Weiterhin lassen sich zur Laufzeit sogar Methoden hinzufügen und entfernen. Dies gilt für alle Instanzen einer Klasse, wie für die Klasse selbst, also auch für Konstruktoren.

7.3 ANTLR

ANTLR(ANother Tool for Language Recognition)[1] ist ein Parsergenerator. Durch erstellen einer Grammatik für Python kann *ANTLR* einen Lexer für das syntaktische Scannen, als auch einen Parser generieren, mithilfe dessen die Transformation in UML-Modellobjekte ausgeführt werden kann. *ANTLR* ist in *Java* geschrieben und kann Lexer und Parser in verschiedenen Sprachen generieren, u.A. *Java*, *Python* und *C#*.

7.4 MOFLON

Das Tool MOFLON wurde an der TU Darmstadt entwickelt und ist verfügbar unter [7]. Es bietet einen grafischen Editor für Modelle in MOF 2.x, es kann Modell in XMI exportieren und importieren. Es enthält einen Code-Generator(Java) für Metamodelle und außerdem einen grafischen Editor für Modell-Transformationen.

Zum Zeitpunkt dieser Arbeit ist der Editor durchaus einsetzbar, die Transformationen sind leider noch nicht vollständig implementiert.

7.5 MagicDraw

MagicDraw ist ein UML-Modellierer der Firma No Magic, Inc. und ist verfügbar unter [19]. Es unterstützt UML 2.x und es sind eine Vielzahl an Erweiterungen für verschiedenste Aspekte der Modellierung verfügbar.

Da MagicDraw viel Wert auf optische Details legt, sind exportierte XMI-Dokumente sehr umfangreich, da viele Darstellungsdetails mit exportiert werden.

Literaturverzeichnis

[1] Antlr-website. http://www.antlr.org. Zuletzt geprüft: 07.07.2008.

[2] Argouml-website. http://argouml.tigris.org. Zuletzt geprüft: 07.07.2008.

[3] J.D. Ullman A.V Aho, R. Sethi. *Compilerbau.* Addison-Wesley, 1986.

[4] HU Berlin. amof2forjava. http://www2.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava/index.html. Zuletzt geprüft: 07.07.2008.

[5] B. Boehm. *Software Engineering Economics.* Prentice-Hall Inc., London, 1981.

[6] Dröschel Bröhl. *Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden.* Oldenbourg R. Verlag GmbH, 1993.

[7] TU Darmstadt. Moflon tool. http://www.moflon.org. Zuletzt geprüft: 07.07.2008.

[8] International Organization for Standardization. Iso-website. http://www.iso.org/. Zuletzt geprüft: 07.07.2008.

[9] Eclipse Foundation. Eclipse modeling project. http://www.eclipse.org/modeling/. Zuletzt geprüft: 07.07.2008.

[10] Eclipse Foundation. Model-to-model(m2m) project. http://www.eclipse.org/m2m/. Zuletzt geprüft: 07.07.2008.

[11] Python Foundation. Python-website. http://www.python.org. Zuletzt geprüft: 07.07.2008.

[12] Inc. Free Software Foundation. Gnu lesser general public license. http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html, Februar 1999. Zuletzt geprüft: 07.07.2008.

[13] Atlas Group. Atlas transformation language(atl). http://www.eclipse.org/m2m/atl/. Zuletzt geprüft: 07.07.2008.

[14] Object Management Group. Omg-website. http://www.omg.org. Zuletzt geprüft: 07.07.2008.

[15] Object Management Group. Unified modeling language, version 1.4. http://www.omg.org/spec/UML/1.4/, September 2001. Zuletzt geprüft: 07.07.2008.

[16] Object Management Group. Uml infrastructure specification, uml superstructure specification. `http://www.omg.org/spec/UML/2.1.2/`, November 2007. Zuletzt geprüft: 07.07.2008.

[17] Jeffrey D. Ullman John E. Hopcroft. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, 1979.

[18] A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In R. Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Amsterdam, 2006. Elsevier Science Publ.

[19] Inc. No Magic. Magicdraw uml-tool. `http://www.magicdraw.com`. Zuletzt geprüft: 07.07.2008.

[20] OMG. Mof core specification. `http://www.omg.org/spec/MOF/2.0/`, Januar 2006. Zuletzt geprüft: 07.07.2008.

[21] OMG. Meta object facility (mof) 2.0 query/view/transformation specification. `http://www.omg.org/cgi-bin/doc?ptc/2007-07-07`, Juli 2007. Zuletzt geprüft: 07.07.2008.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit
im Sinne der Prüfungsordnung nach §24(4) ohne fremde Hilfe selbstständig verfasst
und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 8. Juli 2008

Ort, Datum | Unterschrift