HAW HAMBURG

BACHELORTHESIS
Stefan Schoon

Entwurf und Implementierung einer Erweiterung für eine Modellierungssprache zur Definition formaler Modelle um strukturierte Datentypen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Stefan Schoon

Entwurf und Implementierung einer Erweiterung für eine Modellierungssprache zur Definition formaler Modelle um strukturierte Datentypen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Lars Hamann
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 28. Juni 2024

Stefan Schoon

Thema der Arbeit

Entwurf und Implementierung einer Erweiterung für eine Modellierungssprache zur Definition formaler Modelle um strukturierte Datentypen

Stichworte

ANTLR, Compilerbau, KFG, OCL, strukturierte Datentypen, UML, USE

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Erweiterung des Open-Source-Projektes USE um ein Feature. Bei USE handelt es sich um ein Tool zur Erstellung von formalen Modellen in der UML und in der OCL. Es wird um das Sprachelement für die Verwendung von strukturierten Datentypen mit Wertesemantik erweitert. Das Vorgehen ist die Zusammenstellung relevanter theoretischer Grundlagen, eine Spezifikation und ein Entwurf für die Umsetzung des Vorhabens. Nach einer erfolgreichen Umsetzung wird evaluiert, ob die zuvor gesetzten Ziele erreicht wurden.

Stefan Schoon

Title of Thesis

Design and implementation of an extension for a modelling language to define formal models with structured data types

Keywords

ANTLR, compiler construction, CFG, OCL, structured data types, UML, USE

Abstract

This thesis deals with the extension of the open source project USE by a feature. USE is a tool for creating formal models in UML and OCL. It is extended by the language element for the use of structured data types with value semantics. The procedure is the compilation of relevant theoretical principles, a specification and a draft for the implementation of the project. After successful implementation, an evaluation is carried out to determine whether the previously set goals have been achieved.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

ANTLR	Another Tool for Language Recognition
AST	Abstract Syntax Tree (Abstrakter Syntaxbaum)
EBNF	Erweiterte Backus-Naur-Form
GUI	Graphical User Interface (Grafische Benutzeroberfläche)
KFG	Kontextfreie Grammatik
OCL	Object Constraint Language
OMG	Object Modelling Group
UML	Unified Modelling Language (Vereinheitlichte Modellierungssprache)
USE	UML Specification Environment

1 Einleitung

Dieses Kapitel befasst sich mit der Motivation, dem Ziel, der Abgrenzung und dem Aufbau dieser Arbeit.

1.1 Motivation

Das Open-Source-Project USE (*„UML Specification Environment“)*, wird seit 1999 von der Universität Bremen entwickelt. USE wird für die Erstellung von Modellen in der *Unified Modelling Language* (UML) verwendet. Es unterstützt außerdem das Festlegen von Einschränkungen in der *Object Constraint Language* (OCL).

USE wird stetig um UML- und OCL-konforme Features und Funktionen erweitert. Mit USE können Klassendiagramme, Objektdiagramme und Sequenzdiagramme modelliert werden. Durch OCL-Ausdrücke können zusätzliche Integritätsbeschränkungen für UML-Modelle festgelegt werden (siehe USE-Manual [1]).

Seit langem besteht von Entwicklerseite der Wunsch, dass USE zusätzlich die Syntax und die Semantik von **strukturierten Datentypen** unterstützt. Dies eröffnet neue Möglichkeiten bei der Modellierung und in der Laufzeitanwendung. Ein strukturierter Datentyp kann an jeder Stelle verwendet werden, an der auch ein primitiver Typ wie beispielsweise ein ganzzahliger Typ oder ein selbstdefinierter Typ stehen kann. Ein strukturierter Datentyp besteht im Gegensatz zu einem primitiven Typ, aus mehreren Werten, ähnlich wie bei einem Tupel. Die Werte eines strukturierten Datentyps sind im Gegensatz zu denen im Tupel jedoch noch zusätzlich benannt.

USE bietet eine umfangreiche und solide Basis im Bereich der UML- und OCL-Modellierung und eignet sich daher sehr gut für die Umsetzung des in dieser Arbeit beschriebenen Datentyp-Features.

1.2 Ziel

Ziel dieser Arbeit ist es, USE um ein Datentyp-Feature zu erweitern. Strukturierte Datentypen werden von der *Object Modelling Group* (OMG) im Standard für UML (siehe [2, pp. 167-168]) spezifiziert und sollen nun in USE integriert werden. Strukturierte Datentypen sind im Gegensatz zu Klassen **zustandslos**. Das bedeutet, es wird keine Referenz einer Datentypinstanz gespeichert. Dadurch sind diese nach ihrer Erzeugung unveränderbar, was im Englischen auch als *immutable* bezeichnet wird. Die Verwendung von Datentypen verändert den Systemzustand nicht und ist somit frei von Seiteneffekten.

Strukturierte Datentypen werden mit ihren Attributen und Operationen in einer USE-Spezifikation definiert. Sie können innerhalb der USE-Spezifikation (als Typ) verwendet und Instanzen direkt in der USE-Shell erzeugt werden. Über eine Instanz können die Attribute ausgelesen und die Operationen aufgerufen werden.

Bei der Umsetzung dieses Projektes wird darauf geachtet, die vorhandene Funktionalität von USE vollständig zu erhalten und die Kompatibilität zu allen USE-Spezifikationen älterer Versionen zu gewährleisten. Das angestrebte Ergebnis ist die Veröffentlichung einer neuen Version von USE mit dem neuen Datentyp-Feature.

1.3 Abgrenzung

Das in dieser Arbeit beschriebene Datentyp-Feature wird in ein seit 1999 bestehendes Projekt integriert. Der OMG-Standard für UML gibt vor, wo der Datentyp (*DataType*) im abstrakten UML-Metamodell verortet ist. USE hat den OMG -Standard als Vorbild, hält sich jedoch nicht strikt an das UML-Metamodell, sondern besitzt einige Anpassungen. Bei der Implementierung des Datentyp-Features wird somit eher auf eine Konsistenz zur jüngsten Version des Projekts, statt der genauen Struktur des UML-Metamodells geachtet.

Das UML-Metamodell definiert die grundlegenden Elemente für die Erstellung von UML-Diagrammen. Es ist jedoch nicht auf die Verwendung in einer Laufzeitumgebung wie USE ausgelegt. Die Aufgabe dieser Arbeit besteht nicht darin, Lücken im UML-Metamodell bezüglich der Laufzeit zu finden und Lösungen dafür zu entwickeln.

1.4 Aufbau

Diese Arbeit ist neben der *Einleitung* in die Kapitel *Grundlagen*, *Spezifikation*, *Entwurf*, *Ergebnis*, *Diskussion und Ausblick* sowie *Zusammenfassung* unterteilt.

Das Kapitel *Grundlagen* gibt einen Überblick über die im Projekt verwendeten Werkzeuge und den jeweiligen relevanten theoretischen Hintergrund. Für das Vorstellen der einzelnen Themen wird auf dazu passende Literatur zurückgegriffen.

Das Kapitel *Spezifikation* befasst sich mit der Anforderungsanalyse des zu entwickelnden neuen Datentyp-Features in USE. Hier wird ein Überblick darüber gegeben, wie das fertige Feature konkret verwendet wird. Bei einer erfolgreichen Umsetzung erfüllt die Implementierung alle zuvor aufgenommenen funktionalen und nichtfunktionalen Anforderungen. Die Spezifikation legt also fest, was genau umgesetzt werden soll.

Das Kapitel *Entwurf* befasst sich mit dem Entwurf und dem Design des neuen Features. Neben konkreten Beispielen zur Verwendung des neuen Features werden hier Modelle in Form von Klassendiagrammen zur Visualisierung wichtiger Code- und Architekturausschnitte für die Umsetzung dargestellt. Im Entwurf wird somit festgehalten, wie das Vorgaben umgesetzt wird.

Im Kapitel *Ergebnis* wird festgestellt, ob das fertig umgesetzte Datentyp-Feature die zuvor aufgenommenen Anforderungen erfüllt und ob die Umsetzung die formale Definition von strukturierten Datentypen des OMG-Standards für UML erfüllt. Darüber hinaus werden Herausforderungen und deren Lösungen bei der Umsetzung erläutert.

Im Kapitel *Diskussion und Ausblick* werden Designentscheidungen diskutiert. Hier wird auch ein Ausblick auf die zukünftige Weiterentwicklung von USE auf Grundlage dieser Arbeit gegeben.

Das Kapitel *Zusammenfassung* präsentiert den Stand nach der erfolgreichen Umsetzung des in dieser Arbeit beschriebenen Vorhabens USE um das Datentyp-Feature zu erweitern.

2 Grundlagen

Dieses Kapitel beschäftigt sich mit den theoretischen Hintergründen der für die praktische Umsetzung benötigten Themen. Dies beinhaltet jeweils die Grundlagen zum Compilerbau, zur Spracherkennung mit ANTLR, zu UML und OCL, zu USE, zu strukturierten Datentypen, zur Referenz- und Wertesemantik sowie zu positionellen und benannten Argumenten.

2.1 Compilerbau

Dieser Abschnitt basiert auf einigen Kapiteln des großen Drachenbuches (siehe „Compiler: Prinzipien, Techniken und Werkzeuge“ [3]) und soll einen präzisen Überblick darüber geben, wie ein Compiler aufgebaut ist und was seine Aufgaben sind. Dies ist relevant, da die formale Sprache von USE für das Vorhaben dieser Arbeit erweitert wird.

Ein Compiler übersetzt ein in einer Quellsprache geschriebenes Quellprogramm in ein gleichwertiges Zielprogramm. Bei der Sprache des Zielprogramms, der Zielsprache, handelt es sich in den meisten Fällen um Maschinencode. Benutzer können dieses Programm aufrufen, um damit Eingaben in Ausgaben verarbeiten zu lassen.

Jeder Compiler hat das Ziel, eine Codeoptimierung vorzunehmen, wobei es keine Garantie dafür gibt, dass der erzeugte Zielcode effizienter als der Quellcode ist. Das Problem, aus Code in einer Quellsprache den optimalen Zielcode zu erzeugen, gilt als nicht lösbar. [3, p. 18] Laut den Autoren des großen Drachenbuchs kann aber unter Verwendung einer mathematischen Grundlage aber bewiesen werden, dass eine Codeoptimierung bei allen möglichen Eingaben korrekt ist. [3, p. 20]

2.1.1 Aufbau eines Compilers

Die beiden Hauptaufgaben eines Compilers sind *Analyse* und *Synthese*. Bei der Analyse wird überprüft, ob die einzelnen Programmbestandteile einer bestimmten, als *Grammatik* bezeichneten Struktur folgen. Bei semantischen oder syntaktischen Fehlern muss der Compiler entsprechende Hinweise an den Benutzer weitergeben. [3, pp. 6-15]

Informationen über das Quellprogramm werden während der Analyse in einer als *Symboltabelle* bezeichneten Datenstruktur gesammelt. Bei der Synthese wird dann aus einer im Verlauf der Analyse entstandenen Zwischendarstellung und den Informationen aus der Symboltabelle das Zielprogramm konstruiert.

Der Kompilierungsvorgang verläuft in einer Abfolge von Phasen. Die wichtigsten Phasen sind dabei die *lexikalische Analyse*, die *Syntaxanalyse* und die *semantische Analyse*. Der vollständige Ablauf aller Phasen ist in Abbildung 1 dargestellt. [3, p. 6]

Bei der lexikalischen Analyse werden vom sogenannten *Lexer* aus dem Zeichenstream eines Quelltextes passende Sequenzen, sogenannte *Lexeme* gebildet. Für jedes Lexem gibt der Lexer ein *Token* aus.

Während der Syntaxanalyse erstellt der Parser aus den zuvor ausgegebenen Tokens einen *Syntaxbaum*, welcher die grammatische Struktur dieses Tokenstreams abbildet. Durch diesen Baum ist beispielsweise die Reihenfolge, nach der einzelne Operationen durchgeführt werden, festgelegt. Dabei repräsentieren die inneren Knoten Operationen und die Kindknoten, deren Argumente.

In den nachfolgenden Phasen wird das Zielprogramm mithilfe der grammatischen Struktur des Quellprogramms erstellt. Abschnitt 2.1.4 befasst sich näher mit dieser grammatischen Struktur, die durch eine sogenannte *kontextfreie Grammatik* (KFG) definiert wird.

Bei der semantischen Analyse wird das Quellprogramm auf semantische Konsistenz geprüft und es werden Typinformationen gesammelt. Durch eine Typprüfung erkennt der Compiler zum Beispiel, ob ein Operand zu einem Operator passt. Bei *statisch typisierten Sprachen* werden kompatible Typen, wie beispielsweise ganze Zahlen und Gleitkommazahlen, gegebenenfalls konvertiert.

Während der Übersetzung des Quellcodes wird dieser in eine Zwischendarstellung gebracht. Bei der syntaktischen und semantischen Analyse handelt es sich hierbei meist um den bereits erwähnten Syntaxbaum. [3, p. 11]

Informationen über Variablen wie die Speicheradresse, der Typ und der Gültigkeitsbereich werden in der Symboltabelle gespeichert. [3, p. 14] Die Symboltabelle wird dabei in allen Phasen des Kompiliervorganges genutzt. [3, p. 7]

2.1.2 Ziele eines Compilers

Die Codeoptimierung strebt die Erfüllung mehrerer Designziele an. Das wichtigste Ziel ist dabei die Korrektheit des erzeugten Zielprogramms. Damit ist gemeint, dass die Bedeutung des Quellprogramms erhalten bleiben muss. Die Leistung des optimierten Programms sollte bezüglich Geschwindigkeit, Größe und Energieverbrauch besser sein. Für einen flüssigen Entwicklungs- und Debuggingzyklus sollte die Kompilierungszeit möglichst kurz sein. Zuletzt sollte ein Compiler im Sinne der Wartbarkeit nicht übermäßig komplex sein. [3, pp. 20-21]

2.1.3 Unterschied zwischen statisch und dynamisch

Wenn ein Compiler für eine Programmiersprache Entscheidungen über das Programm treffen kann, dann verwendet diese Sprache eine *statische Verfahrensweise*. Kann eine Entscheidung hingegen nur zur Laufzeit getroffen werden, wird von einer *dynamischen Verfahrensweise* gesprochen. [3, p. 32]

Damit hängt auch der Gültigkeitsbereich einer beliebigen deklarierten Variablen *x* zusammen. Bei einem statischen Gültigkeitsbereich kann der Bereich einer Deklaration direkt aus dem Programm abgelesen werden. In Java ist dies beispielsweise eine Klassenvariable, von der es nur eine Kopie gibt, unabhängig von der Anzahl der Objekte einer Klasse. Bezieht sich *x* zur Laufzeit auf mehrere unterschiedlichen Deklarationen, handelt es sich um einen dynamischen Gültigkeitsbereich. In Java gäbe es für jedes Objekt einer Klasse eine eigene Kopie dieser Variablen. [3, p. 32]

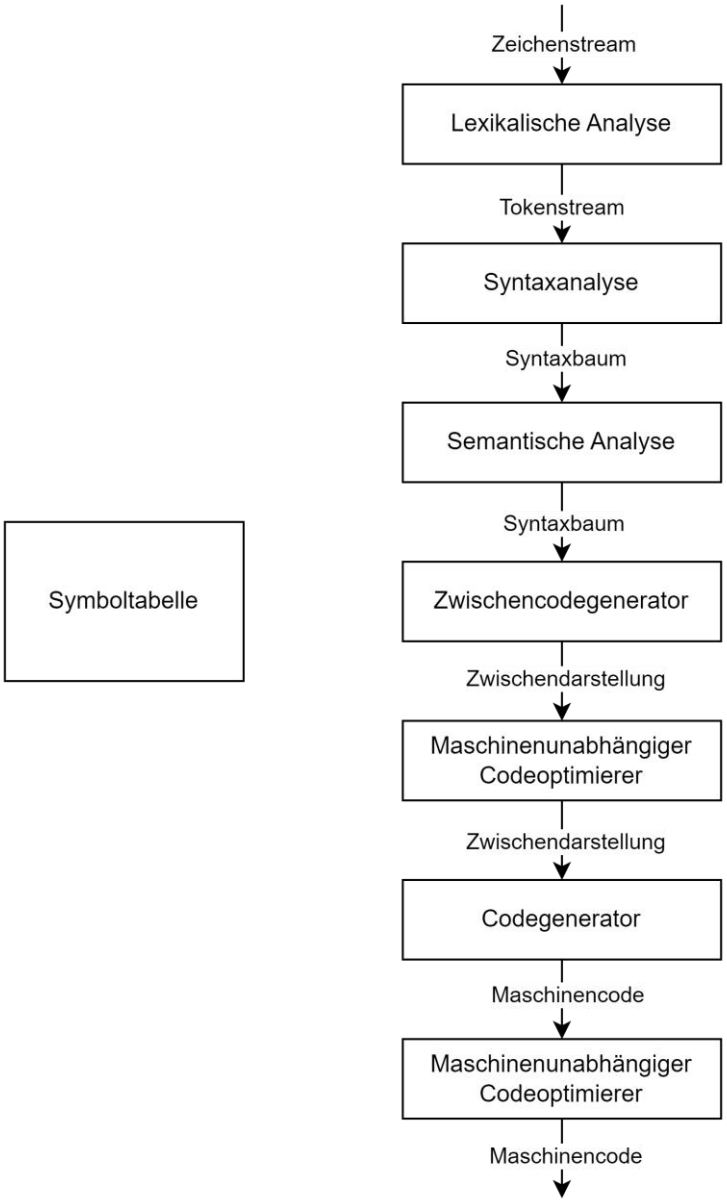

Abbildung 1: Phasen eines Compilers (Quelle: [3, p. 6])

2.1.4 Mechanismen zur Parameterübergabe

Es gibt grundsätzlich zwei Konzepte zur Übergabe von Argumenten an eine Prozedur, den *„Aufruf als Wert“* sowie den *„Aufruf als Referenz“*. *Tatsächliche Parameter* werden beim Aufruf einer Prozedur verwendet, während *formale Parameter* innerhalb einer Prozedur definiert werden.

Bei einem *„Aufruf als Wert“ (Call-by-Value)* werden die tatsächlichen Parameter im Falle eines Ausdrucks ausgewertet oder im Falle einer Variablen kopiert. Alle Berechnungen innerhalb der Prozedur werden lokal durchgeführt und die tatsächlichen Parameter nicht verändert. Dabei ist zu beachten, dass es sich bei einem tatsächlichen Parameter auch um einen Zeiger auf eine Variable handeln kann. In diesem Fall wird nur der Zeiger kopiert und die Prozedur kann Änderungen an dem durch den Zeiger referenzierten Wert vornehmen.

Beim *„Aufruf als Referenz“ (Call-by-Referenz)* erhält eine Prozedur die Adresse des tatsächlichen Parameters. Änderungen an einem korrespondierenden formalen Parameter erscheinen dann als Änderungen am tatsächlichen Parameter. Handelt es sich bei dem Parameter um einen Ausdruck, wird dieser zunächst ausgewertet und nur Änderungen an dem Ergebnis vorgenommen. In diesem Fall hat die Prozedur keinen Einfluss auf den tatsächlichen Parameter.

2.1.5 Kontextfreie Grammatik

Eine Grammatik ist neben einem Baum ein grundlegendes mathematisches Modell, mit dem die syntaktische Struktur von Programmiersprachen beschrieben wird. [3, p. 19] Sie wird auch als *kontextfreie Grammatik* (KFG) bezeichnet. Eine KFG dient somit als Spezifikation für eine formale Sprache. [4, p. 17]

Eine Grammatik folgt selbst einer formalen Notation und wird daher auch als *Metasprache* bezeichnet. Die Syntax einer formalen Sprache wird durch eine KFG beschrieben. Eine verbreitete Form zur Notation von KFGs ist die *erweiterte Backus-Naur Form* (EBNF). [4, p. 73]

Eine KFG besteht aus den folgenden vier Komponenten (vergleiche [3, p. 54]):

1. **Terminale**: Diese werden auch als *Token* bezeichnet und sind die grundlegenden *Symbole* der Sprache. Mehrere Terminale hintereinander bilden einen *Terminalstring*.
2. **Nichtterminale**: Diese stellen jeweils eine Menge von Terminalstrings dar.
3. **Produktionen**: Bestehen aus einem Nichtterminal auf der linken Seite eines Pfeils, gefolgt von einer Reihe von Terminalen und / oder Nichtterminalen auf der rechten Seite des Pfeils.
4. **Startsymbol**: Es ist ein dazu ausgewiesenes Nichtterminal. Es dient als Einstiegspunkt für die KFG.

Um mit einem Spracherkennungstool passende Lexer und Parser für eine durch eine Grammatik beschriebene formale Sprache zu generieren, muss die Grammatik zwei Eigenschaften erfüllen: Sie darf nicht mehrdeutig sein und keine Linksrekursionen aufweisen. [3, p. 59] Mehrdeutigkeiten sorgen für einen Nichtdeterminismus beim Parsen und durch Linksrekursionen gelangt ein Parser möglicherweise in eine Endlosschleife. Abschnitt 2.2 befasst sich näher mit dem bekannten Spracherkennungstool ANTLR, welches auch bei der Entwicklung von USE zum Einsatz kommt.

2.1.6 Syntaxanalyse

Bei der Syntaxanalyse wird ermittelt, wie mithilfe einer Grammatik ein Terrminalstring erstellt werden kann. Dieser Vorgang entspricht konzeptuell dem Erstellen eines *Parsebaumes*. [3, pp. 76, 233] Ein Parsebaum ist eine strukturierte Darstellung aller Elemente eines Quellcodes. Parser führen über eine Eingabe meist einen Durchlauf von links nach rechts durch. Dabei wird um ein Terminal vorausgeschaut und schrittweise der Parsebaum erstellt. [3, p. 76]

Syntaxanalysemethoden lassen sich in *Top-Down*- und *Bottom-Up*-Methoden einteilen. Diese Begriffe bezeichnen die Reihenfolge, in der die Knoten eines Parsebaumes erstellt werden. Bei Top-Down wird der Parsebaum von der Wurzel zu den Blättern aufgebaut, bei Bottom-Up wird der Parsebaum von den Blättern zur Wurzel aufgebaut. [3, p. 76]

Effiziente Top-Down- und Bottom-Up-Methoden funktionieren nur für bestimmte Untergruppen von Grammatiken. Die syntaktischen Konstrukte moderner Programmiersprachen können

durch die Klassen der sogenannten *LL*- und *LR-Grammatiken* beschrieben werden. Dabei sind Parser für die Klasse der LL-Grammatiken weniger umfangreich als Parser für die größere Klasse der LR-Grammatiken. [3, p. 234]

2.1.7 Variationen von LL-Grammatiken

Da die formale Sprache von USE durch eine LL-Grammatik beschrieben wird, wird die umfangreiche Klasse der LR-Grammatiken in dieser Arbeit nicht weiter behandelt. Dieser Abschnitt befasst sich aber mit drei unterschiedlichen Unterklassen von LL-Grammatiken.

Wenn es um die Einteilung von LL-Grammatiken geht, wird im Compilerbau von $LL(1)$- $LL(k)$- und $LL(*)$-Grammatiken gesprochen. Das erste „L“ steht dabei für eine Abtastung der Eingabe von links nach rechts und das zweite „L“ für das Anlegen einer Linksableitung. [3, p. 272] Die Zeichen 1, k und $*$ stehen für die Anzahl der Symbole, um die beim Parsen vorausgeschaut wird. Für jede Parsing-Entscheidung werden bei $LL(1)$ genau ein Symbol, bei $LL(k)$ eine bestimmte Anzahl k Symbole und bei $LL(*)$ immer beliebig viele Symbole vorausgeschaut.

Die unterschiedlichen Grammatikklassen sind notwendig, da sich aus einigen Grammatiken trotz zahlreicher Umstellungen wie beispielsweise das Entfernen von Mehrdeutigkeiten und das Entfernen von Linksrekursionen keine $LL(1)$-Grammatiken formen lassen. Bei linksrekursiven und mehrdeutigen Grammatiken handelt es sich nicht mehr um $LL(1)$-Grammatiken. [3, p. 272]

Die formale Sprache von USE entspricht einer $LL(*)$-Grammatik, welche mithilfe des Spracherkennungstools ANTLR entwickelt und übersetzt wird. Um USE weiterzuentwickeln, ist es daher hilfreich, sich über diesen Hintergrund bewusst zu sein.

2.2 Spracherkennung mit ANTLR

Für die Entwicklung der formalen Sprache von USE wird das Spracherkennungstool ANTLR (*„Another Tool for Language Recognition“*) genutzt. Bei der Erweiterung von USE um das Datentyp-Feature ist es notwendig, die Funktionsweise und das Vorgehen bei der

Sprachentwicklung mit ANTLR zu verstehen. Als maßgebliche Literatur dient dabei „The Definitive ANTLR Reference“ vom ANTLR-Entwickler Terence Parr.

In ANTLR wird eine formale Sprache über eine Grammatik definiert. Bei einer solchen Grammatik handelt es sich um eine oder mehrere Textdateien. Sie ähneln den in Abschnitt 2.1.5 erläuterten KFGs, sind aber nicht mit diesen gleichzusetzen, da sie einige spezifische Besonderheiten aufweisen.

Die formale Sprache von USE wird über mehrere Grammatiken definiert. Die beiden wichtigsten Grammatiken sind die für die Definition eines UML-Modells und die für die Verwendung von OCL-Ausdrücken.

Zur Erstellung passender Parser und Lexer aus diesen Grammatiken wird ANTLR verwendet. [4, p. 71] Wie in Abschnitt 2.1.4 erläutert, beschreibt eine Grammatik die Syntax einer Sprache. In der Praxis wird das Ziel verfolgt, ein Programm zu erhalten, welches die durch eine Grammatik beschriebene Sprache wiedererkennt. ANTLR wandelt Grammatiken in solche *Recognizer* (in Deutsch: „Erkenner“) um. [4, p. 73]

In Abschnitt 2.1.4 wird die *erweiterte Backus-Naur-Form* (EBNF) als Notation für Grammatiken erwähnt. Die Spezifikation von sich wiederholenden Elementen in einer beliebigen Regel ist in der herkömmlichen Backus-Naur-Form nicht erlaubt, da diese keine Rekursion vorsieht. Optionale und sich wiederholende Elemente können aber in der EBNF notiert werden. Bei Grammatiken in der EBNF handelt es sich um kontextfreie Grammatiken, da Regeln nicht auf einen bestimmten Kontext beschränkt werden können. [4, p. 74]

Da eine Sprache in der Praxis meistens nicht vollständig kontextfrei ist, werden *semantische* und *syntaktische Prädikate* zur effizienten Umsetzung einer kontextsensitiven Spracherkennung genutzt. [4, p. 74] Die Abschnitte 2.2.3 und 2.2.4 beschäftigen sich näher mit diesen Prädikaten.

2.2.1 Aktionen

Das Spracherkennungstool ANTLR unterstützt *Aktionen*. Bei Aktionen handelt es sich um Codeblöcke in einer Programmiersprache. Im Falle von USE ist dies Java. Aktionen können bei Bedarf an unterschiedlichen Stellen in einer Grammatik definiert werden. Sie sind von

geschweiften Klammern umgeben. ANTLR interpretiert innerhalb von Aktionen auch Symbole aus der Grammatik. [4, p. 78]

2.2.2 ANTLR's LL(*)-Parsing

In Abschnitt 2.1.7 werden die unterschiedlichen Grammatiktypen $LL(1)$, $LL(k)$ sowie $LL(*)$ erläutert. Dieser Abschnitt befasst sich nun mit dem Parsing von $LL(*)$-Grammatiken in ANTLR.

Das Kapitel „LL(*) Parsing" in „The Definitive ANTLR Reference" (siehe [4, p. 254]) beschreibt, welche Grammatiken $LL(*)$-konform oder vielmehr welche es nicht sind und wie mit diesen umgegangen wird. Obwohl $LL(*)$ deutlich mächtiger als $LL(k)$ ist, werden mitunter Grammatiken benötigt, die noch mächtiger als $LL(*)$ sind. Dies wird in ANTLR durch sogenanntes prädikatives $LL(*)$-Parsing erreicht.

2.2.3 Semantische Prädikate

Die Syntax einiger Sprachen erlaubt für bestimmte *Phrasen* mehr als nur eine Interpretation. Eine Phrase ist in diesem Fall eine bestimmte Abfolge von Symbolen. Reine KFGs sind nicht in der Lage, dies abzubilden. Für viele Sprachen kann daher keine korrekte KFG erstellt werden. ANTLR löst dies durch die Erweiterung der Grammatiken mit *semantischen Prädikaten*, welche den Parse-Vorgang zur Laufzeit an einen bestimmten Kontext anpassen. Semantische Prädikate werden so zur Behebung syntaktischer Mehrdeutigkeiten verwendet. Sie sind dabei nichts anderes als beliebige boolesche Ausdrücke, mit denen die semantische Gültigkeit einer Alternative beschrieben wird. [4, pp. 284, 309]

Aufgrund *kontextsensitiver Konstrukte* ist die Sprache von USE wie viele andere Sprachen schwierig zu parsen. Kontextsensitiv bedeutet hier, dass eine Eingabe nicht ohne das Wissen von umgebenen Statements interpretiert werden kann. [4, p. 284]

Im Beispiel in Listing 1 wird ein semantisches Prädikat zur Überprüfung genutzt, ob es sich bei einer Operationsdefinition um die Definition eines *Konstruktors* für einen Datentyp handelt. Eine Operation ist genau dann ein Konstruktor, wenn ihr Bezeichner in der Schreibweise mit dem *Classifier* übereinstimmt. Handelt es sich um einen Konstruktor, wird eine Action

ausgeführt. Konkret wird überprüft, ob der Text des Bezeichners `$IDENT` dem des übergebenen Classifiers `$c` gleicht und der boolesche Wert `isConstructor` wird entsprechend angepasst.

Ohne dieses semantische Prädikat wäre es in der Grammatik unmöglich, dem Recognizer mitzuteilen, dass die Schreibweise des Bezeichners, der des Classifiers entspricht und somit für die Definition eines Konstruktors steht.

Listing 1: Ausschnitt aus der Regel `operationDefinition` aus der USE-Grammatik

```
operationDefinition[ASTClassifier c] returns [ASTOperation n]
@init { boolean isConstructor = false; }
:
  -- ...
  name = IDENT
  -- ...
  {
    if ($IDENT.text.equals($c.getName())) {
      isConstructor = true;
      -- ...
    }
  }
  -- ...
```

Ein anderer Fall, bei dem semantische Prädikate in der USE-Grammatik Anwendung finden, ist das Erkennen von **Schlüsselwörtern**. Der Kontext legt dabei fest, ob es sich bei einem Bezeichner um ein Schlüsselwort handelt. Da eine KFG die Attribute eines Tokens nicht prüfen kann, kann hier mit einem semantischen Prädikat überprüft werden, ob ein Text einem bestimmten Schlüsselwort entspricht. [4, p. 287]

In der formalen Sprache von USE ist es zulässig, Bezeichner zu verwenden, die Schlüsselwörtern entsprechen. Dies ist bekanntermaßen in Java nicht so. Für jedes Schlüsselwort gibt es in der Grammatik eine Regel, in der mit einem semantischen Prädikat jeweils festgestellt wird, ob einem Token ein Schlüsselwort vorangegangen ist. Für die Implementierung von Datentypen wird das neue Schlüsselwort *dataType* ergänzt.

Das Beispiel in Listing 2 zeigt die Regel für das neu eingeführte Schlüsselwort *dataType* aus der USE-Grammatik. Sie ist in ihrer Struktur mit den Regeln für andere Schlüsselwörter wie beispielsweise *class* vergleichbar. Hier wird der Text des ersten Tokens eines Tokenstreams (`input.LT(1).getText()`) darauf geprüft, ob es sich um das Schlüsselwort *dataType* handelt. Ein semantisches Prädikat endet mit dem „?“-Symbol.

Listing 2: Beispiel für eine Regel zur Überprüfung vorhandener Schlüsselwörter

```
keyDataType:
  {input.LT(1).getText().equals("dataType")}? IDENT ;
```

2.2.4 Syntaktische Prädikate

Neben den semantischen Prädikaten unterstützt ANTLR *syntaktische Prädikate*. Syntaktische Prädikate beschreiben die syntaktische statt der semantischen Gültigkeit beim Anwenden einer Alternative. Syntaktische Prädikate passen basierend auf zur Laufzeit verfügbaren Informationen ebenfalls das Parsen an. Der Unterschied zu semantischen Prädikaten besteht darin, dass syntaktische Prädikate automatisch zukünftige Symbole untersuchen. Dieser Vorgang wird als *Lookahead* bezeichnet. Während semantische Prädikate syntaktische Mehrdeutigkeiten beseitigen, beheben syntaktische Prädikate auch Nichtdeterminismus. [4, p. 297]

Syntaktische Prädikate in ANTLR sind ein Spezialfall semantischer Prädikate, welche Parser-Backtracking-Methoden aufrufen. Dabei wird ein im syntaktischen Prädikat enthaltenes Grammatikfragment mit dem Eingangsstrom verglichen. Syntaktische Prädikate erhöhen die Erkennungskapazitäten eines $LL(*)$-Parsers dadurch enorm. Beim Backtracking handelt es sich zudem um einen gut bekannten Mechanismus. [4, p. 297]

Syntaktische Prädikate enden mit dem „=>“-Symbol. Syntaktische Prädikate finden in der USE-Grammatik nur seltenen Anwendung. Listing 3 zeigt die Verwendung eines syntaktischen Prädikats für *Tuple-Items*. Dabei wird vorausgeschaut, ob der nachfolgende Input zu dem Ausdruck `COLON type EQUAL` passt. Wenn dies der Fall ist, wird die Alternative hinter dem „=>“-Symbol gewählt.

Listing 3: Beispiel für die Verwendung eines syntaktischen Prädikats

```
tupleItem returns [ASTTupleItem n]
:
    name=IDENT
    (
      (COLON type EQUAL) => COLON t=type EQUAL e=expression
      { $n = new ASTTupleItem($name, $t.n, $e.n); }
    |
      --...
    )
    ;
```

Syntaktische Prädikate sind sinnvoll, falls ein $LL(*)$-Parser eine Grammatik in der gewünschten Form nicht verarbeiten kann oder die Präzedenz zwischen zwei Alternativen, welche derselben Eingangssequenz entsprechen, festgelegt werden soll.

2.3 UML

Bei der *Unified Modelling Language (UML)* handelt es sich um eine **Modellierungssprache**, die häufig in der Softwareentwicklung zum Einsatz kommt. [5, p. 20] Mithilfe von bestimmten Notationselementen und daraus aufgebauten Diagrammen können die statischen und die dynamischen Aspekte einer Anwendung modelliert werden. Das Einsatzgebiet der UML beschränkt sich dabei nicht ausschließlich auf die Softwareentwicklung.

Durch eine geprüfte und präzise Semantik und einfach gehaltenen Notationselementen zur grafischen Visualisierung modellierter Systeme ist die UML eindeutig und verständlich. Die UML wird weltweit von der *Object Management Group* (OMG) standardisiert und ist allgemein akzeptiert. Die UML hat ihre Stärken in der objektorientierten Welt und ist dabei unabhängig von Plattform und Programmiersprache. Die zum Zeitpunkt des Verfassens dieser Arbeit aktuelle Version ist UML 2.5.1, veröffentlicht im Dezember 2017. [5, p. 21]

Zu den von USE unterstützen Diagrammen der UML gehören das Klassendiagramm, das Objektdiagramm und das Sequenzdiagramm. Ein Klassendiagramm visualisiert die statischen Strukturbestandteile eines Systems. Mit dem Objektdiagramm wird der aktuelle Zustand von

aus den Klassen erstellten Objekten dargestellt. Durch das Sequenzdiagramm werden Interaktionen zwischen Objekten definiert, dabei liegt der Fokus auf dem Nachrichtenaustausch. [5, pp. 25-30]

Nachfolgend werden die für die Umsetzung des Datentyp-Features am relevantesten UML-Elemente *Modell*, *Classifier*, *Datentyp* und *Instanz* beschrieben.

2.3.1 Modell

Definition

Ein Modell (*model*) stellt die abstrakte Sicht eines Systems dar. Ihr Ziel ist es, die für den Zweck des Modells relevanten Aspekte des Systems in einem angemessenen Detaillierungsgrad zu beschreiben. [6, p. 21]

Notation

In USE ist keine Darstellung für ein Modell vorgesehen, da jede Spezifikation genau ein Modell beschreibt. Ein Modell wird mit dem Schlüsselwort *«model»* eingeleitet.

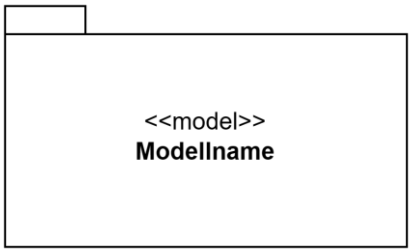

Abbildung 2: Darstellung eines Modells mit dem Paketsymbol

Beschreibung

Beim Beschreiben eines Systems wird auf abstrakter Ebene jedes Mal ein Modell erstellt. Mit diesem wird durch die UML ein dafür vorgesehenes Element bereitgestellt. Ein Modell ist ein Paket, das die Elemente zur vollständigen Beschreibung eines zu modellierenden Systems aus einer bestimmten Sicht enthält. [6, p. 21]

Anwendung

Ein Modell findet allgemein bei der Darstellung unterschiedlicher Sichten auf ein System Anwendung.

2.3.2 Classifier

Definition

Beim Classifier handelt es sich um eine abstrakte Metaklasse, durch die Instanzen mit gemeinsamen Merkmalen klassifiziert werden. [6, p. 25]

Notation

Zur Notation jedes Classifiers kann eine Standardnotation in Form eines einfachen Rechtecks verwendet werden. Die konkrete Notation wird durch die jeweilige Spezialisierung, also ob es sich beispielsweise um eine Klasse oder einen Datentyp handelt, festgelegt. [6, p. 25]

Beschreibung

Der Classifier ist kein Sprachelement, welches direkt vom Modellierer in einem UML-Diagramm verwendet werden kann, sondern es werden dessen Instanzen modelliert. [6, p. 26] Als Instanzen sind hier die von Classifier ableitenden Elemente wie *Class* oder *DataType* gemeint.

Instanzen von Classifier können unterschiedliche Eigenschaften haben, so besitzt eine Klasse Attribute und Optionen, während eine Assoziation Informationen darüber besitzt, ob sie eine Aggregation, eine Komposition oder eine einfache Assoziation ist. [6, p. 26]

Classifier und somit alle seine Instanzen können Teil einer Generalisierungshierarchie sein. [6, p. 26] So kann eine Klasse oder ein Datentyp eine Basisklasse oder einen Basisdatentyp haben, von welchem Eigenschaften wie Attribute und Operationen geerbt werden.

Anwendung

Abbildung 3 zeigt die Klasse *Person* mit den Eigenschaften *Vorname*, *Nachname* und *Geburtsdatum* als Beispiel eines Cassifiers.

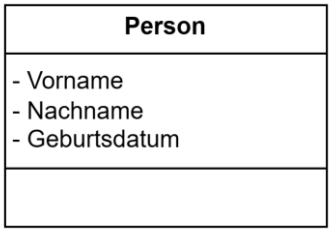

Abbildung 3: Eine Klasse *Person* mit den Eigenschaften *Vorname*, *Nachname* und *Geburtsdatum*

2.3.3 Datentyp

Definition

Die Instanzen eines Datentyps (*data type*) werden durch ihre Werte identifiziert. Besitzt ein Datentyp mehrere Attribute, wird von einem strukturierten Datentyp gesprochen. [6, p. 26]

Notation

Ein Datentyp wird durch das Rechtecksymbol mit dem Schlüsselwort *«dataType»* notiert. Daneben gibt es noch die speziellen Datentypen Aufzählungstyp (*«enumeration»*) und primitiver Typ (*«primitive»*). [6, p. 26]

Beschreibung

Ein Datentyp kann wie eine Klasse Attribute und Operationen enthalten, ausgenommen ist dabei der primitive Typ. Eine Datentypinstanz unterscheidet sich von einem Klassenobjekt darin, dass er keine Identität besitzt. Die Instanzen eines Datentyps gelten als gleich, wenn die Werte für alle Attribute jeweils gleich sind. Bei Instanzen einer Klasse handelt es sich um unterschiedliche Objekte, auch wenn alle ihre Attribute die jeweils gleichen Werte besitzen. [6, p. 27] In Abschnitt 2.6 wird näher auf die strukturierten Datentypen eingegangen.

Anwendung

Abbildung 4 zeigt die Notation eines Datentyps anhand des Beispiels *Date* mit den Attributen *day*, *month* und *year* sowie den Operationen *getMonths()*, *isLeapYear()* und *add(days)*.

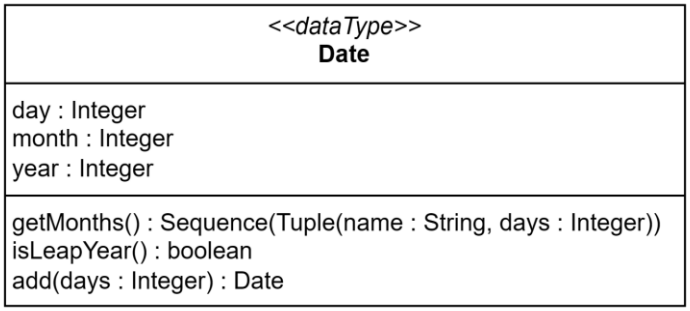

Abbildung 4: Datentyp *Date*

Das Konzept der *Generalisierung* (auch bekannt als *Vererbung*) ist für Classifier und somit auch für Datentypen definiert. Es ist also möglich, Generalisierungshierarchien zwischen unterschiedlichen Datentypen zu schaffen. Abbildung 5 zeigt eine einfache Generalisierungsstruktur zwischen mehreren Datentypen.

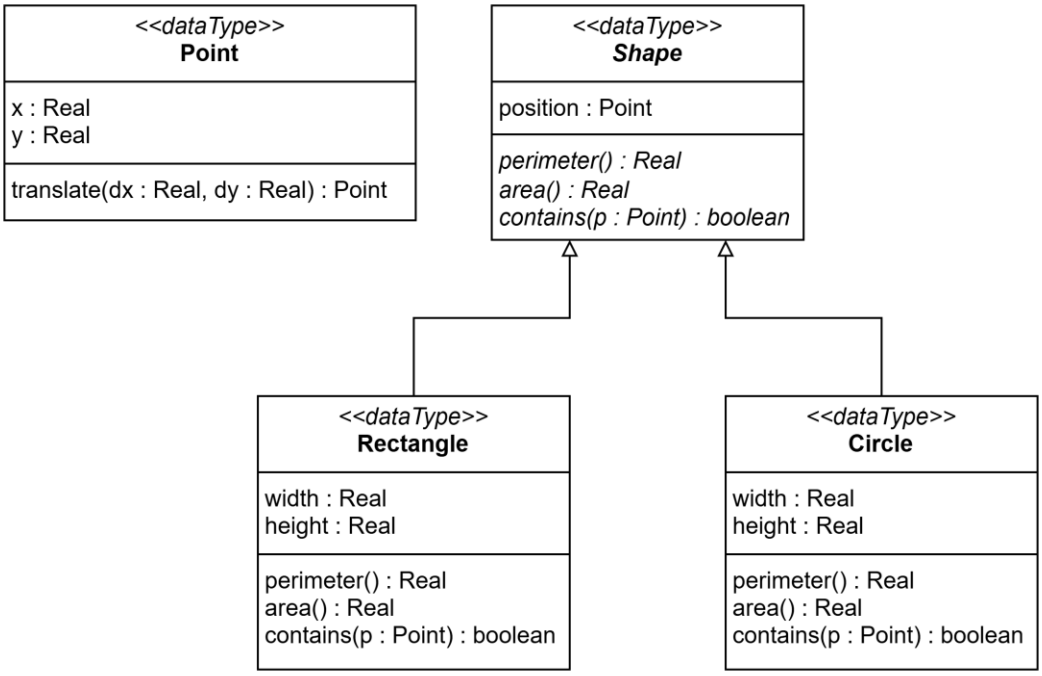

Abbildung 5: Einfache Generalisierungshierarchie zwischen Datentypen anhand des Modells *Shapes* aus Listing 9

2.3.4 Instanz

Definition

Eine Instanz repräsentiert die Instanziierung, also die Erzeugung eines Classifiers.

Notation

Die UML bietet keine allgemeine Notation für Instanzen. Bei Instanzen von Klassen handelt es sich um Objekte. Diese werden wie eine Klasse selbst als Rechteck dargestellt. Im Objektdiagramm wird auch von einer Instanzbeschreibung gesprochen. [6, p. 188]

Abbildung 6: Notation einer Klasseninstanz (Objekt)

Beschreibung

Bei einer Instanz handelt es sich um eine zur Laufzeit vorhandene Einheit des modellierten Systems. In den meisten Fällen handelt es sich bei Instanzen um Objekte von Klassen. In USE ist eine Instanz genau einem Classifier zugeordnet, im Falle eines Objektes hat dieses genau den Typ einer Klasse.

2.4 OCL

Die *Object Constraint Language* (OCL) ist eine Erweiterung der UML. Sie wird wie die UML selbst von der OMG standardisiert (siehe [7]). Es handelt sich dabei um eine formale Sprache, welche vor allem zur Definition von *Einschränkungen* (gängiger englischer Begriff: *Constraints*) in UML-Modellen dient. Die OCL ist *deklarativ* und *seiteneffektfrei*. *Deklarativ* bedeutet, dass sie spezifiziert, was das gewünschte Ergebnis ist und nicht wie dieses erreicht wird. *Seiteneffektfrei* bedeutet, dass ihre Ausführung keinerlei Änderungen an Objekten vornimmt. Die OCL ist ähnlich wie die UML allgemein akzeptiert und ergänzt UML-Modelle um eine präzise Semantik. Sie verfügt über viele Erweiterungen, wie zum Beispiel temporäre

Constraints und dient als Kernsprache anderer von der OMG standardisierter Sprachen. [8, pp. 6-7]

2.4.1 Constraints

„Ein **Constraint** ist eine Einschränkung eines oder mehrerer Werte (von Teilen) eines objektorientierten Modells oder Systems.“ [8, p. 10]

Ein Constraint wird auf der Klassenebene definiert, seine Semantik wird aber auf Objektebene angewendet. [8, p. 10]

Invarianten

Eine Invariante ist ein Constraint, die für ein Objekt während seiner gesamten Lebensdauer erfüllt sein muss. Invarianten sind Regeln, welche für die realen Objekte, nach denen eine Software modelliert wird, gelten sollen. [8, p. 11] Listing 4 zeigt die allgemeine Syntax für eine OCL-Invariante, so wie sie auch in USE verwendet wird.

Listing 4: Syntax für OCL-Invarianten

```
context <classifier>
inv [<constraint name>]: <Boolean OCL expression>
```

In USE können Invarianten für beliebige Instanzen von Classifier wie beispielsweise für eine Klasse oder für einen Datentyp definiert werden. Listing 5 zeigt die Definition einer abstrakten Klasse *Person* mit den Attributen *firstname*, *lastname* und *age*. Dort wird außerdem das Contraint *Person1* für das Attribut *age* festgelegt. Für das Attribut *age* gilt für die Klasse *Person* die Einschränkung, immer einen Wert größer Null haben zu müssen.

Listing 5: Definition des Constraints *Person1* für eine Klasse *Person* in einer USE-Spezifikation

```
abstract class Person
attributes
  firstname : String
  lastname : String
  age : Integer
-- ...
end

-- ...

context Person
-- The age attribute of persons is greater than zero
inv Person1: age > 0
```

Vor- und Nachbedingungen

Vor- und Nachbedingungen sind Constraints, welche die Anwendbarkeit und die Auswirkung einer Operation spezifizieren. Sie enthalten dabei weder eine Implementierung noch einen Algorithmus. [8, p. 14]

Eine Vorbedingung muss vor der Ausführung einer Operation erfüllt sein. [8, p. 15] Listing 6 zeigt die allgemeine Syntax einer Vorbedingung.

Listing 6:Syntax für OCL-Vorbedingungen

```
context <classifier>::<operation> (<parameters>)
pre [<constraint name>]: <Boolean OCL Expression>
```

Eine Nachbedingung muss nach der Ausführung einer Operation erfüllt sein. [8, p. 17] Listing 7 zeigt die allgemeine Syntax einer Nachbedingung.

Listing 7: Syntax für OCL-Nachbedingungen

```
context <classifier>::<operation> (<parameters>)
post [<constraint name>]: <Boolean OCL Expression>
```

USE unterstützt die Definition von Vor- und Nachbedingungen für beliebige Operationen von Classifiern mit OCL-Ausdrücken. Listing 8 zeigt jeweils die Definition einer Vor- und einer Nachbedingung für die Operation *raiseSalary()* der zuvor definierten Klasse *Employee*.

Die Vorbedingung für das der Operation *raiseSalary()* übergebene Argument *amount* ist, dass es sich um einen positiven Wert handeln muss. Die Nachbedingung ist, dass das Attribut *salary* nach dem Aufruf dieser Operation um den Wert von *amount* inkrementiert wird.

Listing 8: Definition einer Vor- und Nachbedingung für die Operation *raiseSalary()* der Klasse *Employee* innerhalb einer USE-Spezifikation

```
class Employee < Person
attributes
  salary : Real
operations
  raiseSalary(amount : Real) : Real
end

-- ...

context Employee::raiseSalary(amount : Real) : Real
  -- If the amount is positive, raise the salary by the given amount
  pre:  amount > 0
  post: self.salary = self.salary@pre + amount and result = self.salary
```

2.5 USE – UML Specification Environment

Bei USE („*UML Specification Environment*“) handelt es sich um ein Open-Source-Projekt zum Modellieren und Ausführen von UML-Modellen und dem Festlegen von Einschränkungen in der OCL für diese UML-Modelle.

USE ist zum größten Teil in Java geschrieben. Für die Entwicklung der formalen Sprache von USE wird das in Abschnitt 2.2 vorgestellte Spracherkennungstool ANTLR (Version 3) verwendet. Die aktuelle USE-Version 7.1.0 verwendet Maven als Build-Tool.

Mit USE lassen sich ganz allgemein Software-Systeme spezifizieren. Genauer gesagt basiert es auf einer Untermenge der UML, die unter anderem zum Modellieren von Klassen, Komponenten und Abhängigkeiten in einem Projekt verwendet wird. [1]

In einer Datei, der USE-Spezifikation, werden UML-Klassendiagramme in der eigens dafür entwickelten Sprache definiert. Zusätzlich können dort Constraints für das Modell in der OCL festgelegt werden.

Die in einer USE-Spezifikation erstellten Modelle lassen sich mit USE als Klassendiagramme visualisieren. Darüber hinaus lassen sich erstellte Objekte als Objektdiagramm und ausgeführte Operationen als Sequenzdiagramm visualisieren.

2.6 Strukturierte Datentypen

Dieser Abschnitt beschäftigt sich konkret mit der Theorie der strukturierten Datentypen, über die in Abschnitt 2.3.3 bereits ein Überblick gegeben wird. Dabei geht es um die Definition, eine Beschreibung aus Sicht der Mengenlehre sowie die Vorteile gegenüber einem Tupel.

2.6.1 Definition

Es werden mehrere Definitionen des strukturierten Datentyps von unterschiedlichen Autoren vorgestellt, wobei sich die Definitionen nicht widersprechen und im Kern das Gleiche beschreiben.

Nach dem OMG-Standard (vergleiche [2, pp. 167-168]) unterscheiden sich Datentypen ausschließlich in ihren Werten. Somit werden Instanzen eines Datentyps mit dem gleichen Wert als identisch betrachtet. Besitzt ein Datentyp eine beliebige Anzahl an Attributen, dann wird er als *strukturierter Datentyp* bezeichnet. Besitzen unterschiedliche Instanzen desselben strukturierten Datentyps die gleichen Werte für die jeweiligen Attribute, werden diese ebenso als identisch angesehen. Ein strukturierter Datentyp kann neben den Attributen auch Operationen enthalten. Er unterscheidet sich von der Klasse darin, dass er ohne einen Systemzustand verwendet werden kann.

Laut Craig Larman (vergleiche [9, p. 162]) sind Datentypen in der UML-Terminologie eine Menge von Werten, ohne dass deren Identität von Bedeutung wäre. Gleichheitstests basieren

also nicht auf der Identität, sondern auf den Werten. Solche Konstrukte sind sehr hilfreich, da es meist nicht sinnvoll ist, zwischen unterschiedlichen Instanzen derselben natürlichen Zahl, derselben Zeichenfolge oder desselben Datums zu unterscheiden. Dagegen ist es sinnvoll, zwischen zwei unterschiedlichen Instanzen von Personen mit dem gleichen Vor- und Nachnamen zu unterschieden, da beide Instanzen unterschiedliche Individuen mit demselben Namen repräsentieren können. Während Instanzen von Datentypen wie Zahlen unveränderlich sind, sind Instanzen von einer Klasse Person (siehe Listing 5) veränderlich, da diese beispielsweise ihren Nachnamen bei bestimmten Anlässen ändern können.

Für Bernd Oesterreich (vergleiche [10, p. 75]) sind Datentypen Klassen mit einfachen Attributen, deren Typen wiederum Klassen oder andere Datentypen sind. Durch Datentypen werden demnach selbstentwickelte, fachlich neutrale Standardklassen repräsentiert. Sie sind nicht persistent und besitzen neben wenigen Attributen oftmals einfache Lese- und Schreiboperationen. Gelegentlich besitzen sie einfache Berechnungsfunktionen und seltener Operationen zur Umwandlung in andere Primitive, wie die Umwandlung in eine lesbare Zeichenkette.

Aus den oben beschriebenen Definitionen lässt sich eine eigene Definition für strukturierte Datentypen entwickeln, welche als Grundlage für das Datentyp-Feature dient:

Strukturierte Datentypen enthalten wie Klassen Attribute und Operationen. Instanzen eines Datentyps besitzen keine Identität und unterscheiden sich einzig in den Werten der Attribute. Zwischen Datentypen kann es wie bei Klassen eine Generalisierungshierarchie geben.

2.6.2 Mengenlehre

Strukturierte Datentypen sind Elemente mehrdimensionaler Mengen. Jedes Attribut ist ein Wert einer Menge, zum Beispiel aus der Menge der natürlichen Zahlen $\mathbb{N}$. Beispielsweise ist der Datentyp *Date* aus den drei Attributen *day*, *month* und *year* aufgebaut. Dabei wird hier davon ausgegangen, dass die drei Attribute jeweils einen beliebigen Wert aus der Menge der natürlichen Zahlen $\mathbb{N}$ annehmen können (bekanntlich kann *day* Werte zwischen 1 und 31 und *month* Werte zwischen 1 und 12 annehmen). Ein beliebiger Wert für ein Datum ist in diesem Fall dann ein Element der Menge $\{\mathbb{N} \times \mathbb{N} \times \mathbb{N}\}$, wobei „×“ für das Kreuzprodukt zweier Mengen steht. In diesem Fall sind alle Komponenten des Datentyps vom gleichen Typ.

Wenn alle Komponenten eines Datentyps vom gleichen Typ sind, ist es ein homogener Datentyp. Bei inhomogenen Datentypen können die Komponenten von unterschiedlichen Typen sein.

2.6.3 Vorteile gegenüber Tupel

USE unterstützt bereits das Element Tupel. Ein Tupel ist in der Informatik bekanntermaßen eine Gruppe mehrerer Werte. Dabei können die Werte entweder alle vom gleichen Typ (homogen) oder von unterschiedlichen Typen (inhomogen) sein. USE unterstützt inhomogene und damit auch homogene Tupel.

Die Werte eines Tupels sind im Gegensatz zu den Attributen eines Datentyps nicht benannt, was die Bedeutung der einzelnen Werte erschwert. Ein Datentyp ist ein selbst definierter Typ mit benannten Attributen und gegebenenfalls Operationen, wohingegen ein Tupel ausschließlich Daten enthält. Ein Datentyp kapselt somit mehr Logik als ein Tupel und hat eine klare Semantik. Zum Beispiel ist ein Datentyp *Person* mit den Attributen *name*, *firstname* und *dateOfBirth* intuitiver als ein Tupel *(„Mustermann“, „Max“, (1900, 01, 01))*

Ein Tupel ist zudem immer vom Typ *Tuple*. Ein benutzerdefinierter Datentyp sorgt im Gegensatz zu einem Tupel dadurch für mehr Typsicherheit. Diese Typisierung vereinfacht die Lesbarkeit und reduziert die Fehleranfälligkeit.

Ein weiterer Vorteil von benutzerdefinierten Datentypen ist, dass sie an mehreren Stellen wiederverwendet werden können. In Bezug auf USE kann ein Datentyp *Person* überall in einem Modell verwendet werden. Beim Ändern der Definition eines Datentyps geschieht dies an nur einer Stelle, was die Wartbarkeit vereinfacht.

2.7 Referenz- und Wertsemantik

Bei der Zuweisung von Werten an Variablen gibt es die *Referenz*- und die *Wertesemantik*. Bei der Referenzsemantik verweist eine Variable auf ein im Speicher abgelegtes Objekt. Beim Kopieren wird nicht das Objekt selbst, sondern die Referenz darauf kopiert, sodass alle Referenzen auf dasselbe Objekt zeigen. Veränderungen am Objekt über eine dieser Referenzen sind daher von allen Referenzen aus sichtbar.

Bei der Wertesemantik wird bei einer Zuweisung das gesamte Objekt kopiert. Die zugewiesenen Objekte sind dabei unabhängige Kopien mit jeweils eigenem Zustand. Änderungen an einem Objekt betreffen nur diese spezifische Kopie und nicht andere Kopien.

Die in dieser Arbeit umgesetzten strukturierten Datentypen besitzen eine Wertsemantik. Da aus Datentypen keine Objekte, sondern zustandslose Instanzen erstellt werden, wird bei der Änderung eines Attributs eine neue Instanz erstellt.

2.8 Positionelle und benannte Argumente

Es gibt grundsätzlich zwei unterschiedliche Formen, nach denen Argumente einer Operation übergeben werden können: die *positionellen* und die *benannten* Argumente.

Bei den positionellen Argumenten muss die Reihenfolge der übergebenen Argumente mit der Reihenfolge der entsprechenden Parameter in der Signatur der Operation übereinstimmen. Die Argumente werden ohne zusätzliche Information darüber, zu welchem Parameter sie gehören, übergeben. Die Zuordnung der Argumente zu den Parametern erfolgt hierbei ausschließlich über die Reihenfolge, in der sie übergeben werden. Dieses Konzept findet sich beispielsweise in der Sprache Java.

Bei den benannten Argumenten spielt die Reihenfolge, in der die Argumente der Operation übergeben werden, keine Rolle. In diesem Fall wird jedem Argument seine jeweilige Parameterbezeichnung vorangestellt, wodurch klar ist, zu welchem Parameter ein Argument gehört. Die Parameterbezeichnung und das Argument werden oft durch ein Gleichheitszeichen getrennt, wobei dies je nach Programmiersprache unterschiedlich gehandhabt werden kann. Dieses Konzept findet zum Beispiel in Python Anwendung.

3 Spezifikation

Dieses Kapitel befasst sich mit den Anforderungen, die an das zu implementierende Datentyp-Feature gestellte werden. Es wird klassischerweise zwischen *funktionalen* und *nichtfunktionalen* Anforderungen unterschieden. Die funktionalen Anforderungen beschreiben, was ein System leisten soll, während die nichtfunktionalen Anforderungen die angebotenen Dienste und Funktionen beschränken. [11, pp. 116-123]

Die beiden nachfolgenden Abschnitte beschäftigen sich genauer jeweils mit den wichtigsten funktionalen und nichtfunktionalen Anforderungen.

3.1 Funktionale Anforderungen

Allgemein ist das Ziel dieser Arbeit die akkurate Umsetzung der Verwendung strukturierter Datentypen in USE. Das erweiterte Programm soll die bereits vorhandene Funktionalität in keiner Weise einschränken, alte USE-Spezifikationen sollen vollständig mit dieser kompatibel sein.

Die Idee für die Möglichkeit der Verwendung von zustandslosen Datentypen in USE steht seit langem im Raum. Im Projekt existieren daher bereits USE-Spezifikationen, die für ein erfolgreiches Parsen und eine sinnvolle Anwendung das bisher nicht vorhandene Datentyp-Feature benötigen. Die in diesen Dateien vorgegebene Syntax wird für die Umsetzung übernommen und erweitert.

Die funktionalen Anforderungen spezifizieren, wie das neu zu programmierende Datentyp-Feature zu verwenden ist. Diese Anforderungen sind:

- Definition eines Datentyps in einem Modell in einer USE-Spezifikation
- Definition genau eines Konstruktors für einen Datentyp
- Erzeugung einer Datentypinstanz auf der USE-Shell
- Verwendung eines Datentyps als **Typ** innerhalb eines Modells
- Verwendung eines Datentyps wie ein **Wert** innerhalb eines Modells

Es gibt drei Programmkomponenten in USE, die von den genannten Anforderungen betroffen sind: die Definition von USE-Spezifikationen, die USE-Shell und die grafische Benutzeroberfläche. Die nachfolgenden Abschnitte befassen sich detailliert mit den Anforderungen an die gerade genannten Punkte.

3.1.1 Anforderungen an die Erstellung von USE-Spezifikationen

Ein Datentyp wird in einem Modell innerhalb einer USE-Spezifikation mithilfe des dafür neu eingeführten Schlüsselworts *dataType* definiert. Datentypen lassen sich wie Klassen instanzieren, besitzen jedoch keinen Systemzustand und sind nicht persistent. Ein Datentyp besitzt wie eine Klasse beliebige Attribute und Operationen. Die Attribute werden über eine spezielle Operation, dem sogenannten Konstruktor definiert. Basisattribute werden der Signatur in Klammern als kommaseparierte Liste angehängt.

Listing 9 zeigt ein einfaches USE-Modell *Shapes*, welches die Datentypen *Point*, *Shape* und *Circle* enthält. *Circle* erbt vom abstrakten Datentyp *Shape* und besitzt damit automatisch das Attribut *position,* welches im Datentyp *Point* definiert wurde. Der Datentyp *Point* repräsentiert eine beliebige Position in $\mathbb{R}^2$. *Circle* bekommt zusätzlich noch das Attribut *radius*. Bei einer Instanziierung von *Circle* müssen sowohl das von *Shape* stammende Basisattribut *position* als auch das direkt in *Circle* definierte Attribut *radius* initialisiert werden. Daher sind beide Attribute als Parameter bei der Konstruktordefinition von *Circle* angegeben.

Um dem Compiler beim Übersetzen der USE-Spezifikation mitzuteilen, dass es sich bei dem ersten Attribut *position* um ein Basisattribut aus einem in der Hierarchie über *Circle* stehenden Datentypen handelt, wird dieses separat in Klammern hinter der Parameterliste angehängt. Bei

mehreren Basisattributen würden diese kommasepariert dort stehen. Die Reihenfolge der Parameter und der angehängten Basisattribute ist unabhängig und kann vom Modellierer nach Bedarf festgelegt werden. Die einem Konstruktor übergebenen Argumente sind hingegen positionell, das heißt die Reihenfolge der übergebenen Argumente bei einer Instanziierung muss eingehalten werden.

Listing 9: Ein USE-Modell *Shapes* bestehend aus den Datentypen *Point*, *Shape* und *Circle*

```
model Shapes

dataType Point
operations
  Point(x : Real, y : Real)
  translate(dx : Real, dy : Real)(dx) : Point =
    Point(self.x + dx, self.y + dy)
end

abstract dataType Shape
operations
  Shape(position : Point)
  perimeter() : Real
  area() : Real
end

dataType Circle < Shape
operations
  Circle(position : Point, radius : Real)(position)
  perimeter() : Real = 2.0 * 3.14 * radius
  area() : Real = 3.14 * radius * radius
end
```

Herkömmliche Operationen werden bis dato immer auf einem Bezeichner eines Klassenobjektes aufgerufen. Konstruktoren können direkt in einem Modell oder in der Shell aufgerufen werden. Für Konstruktoren wird im Gegensatz zu Operationen kein Rückgabetyp definiert (auch nicht *void*). Ihr Ergebnis ist eine Instanz des zugehörigen Datentyps, auf welcher dann alle anderen im Datentyp definierten Operationen aufgerufen werden können. Ein Operationsaufruf

auf einer Datentypinstanz oder einer Klasseninstanz ist identisch bis auf den Unterschied, dass im ersten Fall keine Änderungen an dem Systemzustand vorgenommen werden. Ändert eine Operation die Attribute einer Datentypinstanz, ist das Ergebnis eine neue Instanz dieses Datentyps.

Für Konstruktoren können Vorbedingungen in der OCL festgelegt werden. Da der Aufruf eines Konstruktors keine Änderung am Systemzustand vornimmt, kann es keine Nachbedingungen dafür geben.

3.1.2 Anforderungen an die USE-Shell

USE verfügt über eine Kommandozeilenoberfläche, mit der Befehle verarbeitet oder Skriptdateien ausgeführt werden können. Mit der Einführung von Datentypen sollen Konstruktoraufrufe direkt in dieser Shell ausgeführt werden können und ein passender Ergebnistext zurückgegeben werden. In der USE-Shell wird allgemein zwischen einer mit einem „!“-Symbol eingeleiteten **seiteneffektbehafteten Aktion** und einer mit einem „?“-Symbol eingeleiteten **seiteneffektfreien Abfrage** unterschieden.

Listing 10 zeigt eine Abfrage mit einem Konstruktoraufruf des im Modell aus Listing 9 definierten Datentyps *Point*, der einen einfachen Punkt in 2D repräsentiert. Dem Konstruktor werden die Werte für *x* und *y* mitgegeben. USE muss den richtigen Datentypen anhand der Schreibweise des Konstruktors erkennen. Das Ergebnis ist dann die Textdarstellung der zugehörigen Datentypinstanz.

Listing 10: Beispiele für Eingaben in die USE-Shell mit Ergebnis

```
use> ?Point(1, 1)
-> Point{x=1, y=1} : Point
use> ?Point(1, 2).translate(1, -4)
-> Point{x=2, y=-2} : Point
```

Beim zweiten Befehl in Listing 10 handelt es sich um einen an einen Konstruktor angehängten Operationsaufruf. Die Operation *translate()* verschiebt den ursprünglichen Punkt um die entsprechenden übergebenen Werte. Das Ergebnis ist eine neue Instanz von *Point* mit den jeweils berechneten Werten.

3.1.3 Anforderungen an die grafische Benutzeroberfläche

USE verfügt neben der Shell über eine grafische Benutzeroberfläche (englisch: *Graphical User Interface*, kurz: *GUI*). Die Hauptbestandteile der GUI sind neben einer Menüleiste und einer Schnellzugriffsleiste eine Baumanzeige mit einer Informationsvorschau des geladenen Modells, einem Bereich für die Anzeige unterschiedlicher Diagramme und einem Logbereich.

Die neuen Datentypen sollen neben den anderen Elementen im Modell in der Baumanzeige korrekt angezeigt werden. Es ist erforderlich, dass die Datentypen wie Klassen beim Anzeigen des Klassendiagramms einschließlich ihrer Vererbungshierarchie in der korrekten Notation mit dem Schlüsselwort *«dataType»* (siehe Abschnitt 2.3.3) dargestellt werden.

In Abbildung 7 wird der Hauptbildschirm der GUI von USE dargestellt. Dort ist das Modell *Shapes* aus Listing 9 geladen. Die enthaltenen Datentypen *Point*, *Shape*, *Rectangle* und *Circle* werden im Fenster *Class diagram* in der korrekten Notation dargestellt.

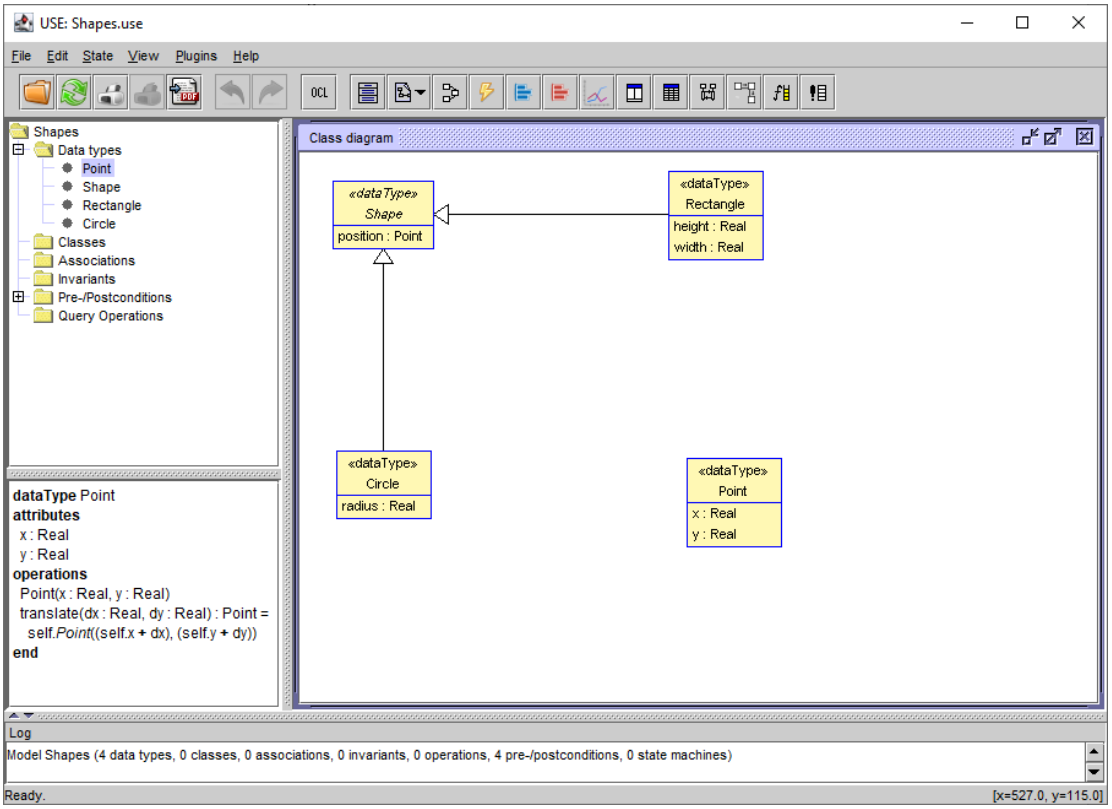

Abbildung 7: Die GUI von USE mit dem Modell *Shapes* aus Listing 9

3.2 Nichtfunktionale Anforderungen

Die in dieser Arbeit neue entwickelte Version von USE soll nach wie vor alle nichtfunktionalen Anforderungen an das Projekt erfüllen. In jedem Softwareprojekt werden unter anderem folgende nichtfunktionale Anforderungen angestrebt: Kompatibilität, Erweiterbarkeit und Wartbarkeit.

3.2.1 Kompatibilität

Ältere USE-Spezifikationen sollen vollständig mit der neuen Version kompatibel sein. Nach dem Laden dieser Spezifikationen sollen sie sich bei der Verwendung nicht anders als in der vorherigen Version verhalten. Es wird angestrebt, dass bei der Benutzung des Systems zu keinem Zeitpunkt interne Fehler auftreten. Kommt es zu einem internen Fehler in USE, bedeutet dies, dass es Lücken in der korrekten Fehlerbehandlung gibt.

Als Beispiel hierfür wird die Definition von Vor- und Nachbedingungen für die neuen Konstruktoren aus Abschnitt 2.4.1 betrachtet. Während für einen Konstruktor wie für eine klassische Operation Vorbedingungen definiert werden können, kann es aufgrund der Zustandslosigkeit für diese keine Nachbedingungen geben. Wenn ein Benutzer für einen beliebigen Konstruktor eine oder mehrere Vorbedingungen definiert, wird dies wie gewohnt behandelt. Falls der Benutzer für einen Konstruktor auch Nachbedingungen definiert, stellt sich die Frage, wie damit umgegangen wird. Das Wichtigste dabei ist, dass ein solcher Fall robust verarbeitet wird und es bei der Auswertung zu keinem internen Fehler kommt, da das Auswerten von Nachbedingungen für die Instanzen von Datentypen technisch nicht möglich ist (siehe Abschnitt 3.1.1). Mögliche Behandlungen wären das Ignorieren von Nachbedingungen von Konstruktoren oder die Ausgabe einer Fehlermeldung oder Warnung, dass Nachbedingungen nicht erlaubt sind.

3.2.2 Erweiterbarkeit

Eine wichtige nichtfunktionale Anforderung an das System ist die Erweiterbarkeit. So wie USE um die Verwendung von Datentypen erweitert werden soll, ohne dass dabei große Teile des Systems vollständig neu entwickelt werden müssen, so soll die Erweiterbarkeit auch nach der

Umsetzung dieses Datentyp-Features gegeben sein. Dies gelingt durch die Trennung von Zuständigkeiten und die Einhaltung der Architekturvorgaben. USE verfügt über eine umfangreiche Paketstruktur. Klassen und Gruppen von Klassen für bestimmte Aufgaben befinden sich in separaten Paketen. Klassen, Variablen und Methoden besitzen zudem konsistente Namenskonventionen. Dies erleichtert die Erweiterung des Systems, da aus der Paketstruktur hervorgeht, in welchen Programmabschnitten neuer Sourcecode hinzugefügt werden sollte.

Der Sourcecode, der für das Erstellen des *abstrakten Syntaxbaumes* (AST) zuständig ist, befindet sich in einem separaten Paket. Ebenso befinden sich die Klassen zum Erstellen des Metamodells in einem eigenen Paket. Ein weiteres Paket enthält alle Klassen, die mit OCL zusammenhängen. Außerdem gibt es ein Paket, welches für die Erzeugung von Objekten und der Verwaltung des Systemzustandes eines USE-Modells zuständig ist. Diese Kapselung der Zuständigkeiten ermöglicht es, dass zukünftige Entwickler verstehen, an welchen Stellen Erweiterungen der Codebasis stattfinden sollen.

3.2.3 Wartbarkeit

Eine bestmögliche Wartbarkeit wird allgemein in jedem Softwareprojekt angestrebt. Ist ein Projekt gut wartbar, so lässt es sich meist auch gut erweitern.

USE verfügt über eine Architektur, in der die Aufgaben klar voneinander getrennt sind (siehe Abschnitt 3.2.2). Durch diese Trennung und durch klare Generalisierungshierarchien werden Abhängigkeiten zwischen einzelnen Komponenten minimiert. Dies ist für eine gute Wartung ebenfalls unerlässlich.

In der neuen Version von USE wird der in der UML als *Classifier* eingeordnete *DataType* eingeführt. Dieser ähnelt der vorhandenen *Class*. *Class* besitzt im UML-Metamodell *Classifier* als Basisklasse. Dies spiegelt sich zu einem gewissen Teil im Sourcecode von USE wider. Bei der Entwicklung von *DataType* kann dieser ebenfalls als eine Ableitung von *Classifier* umgesetzt werden. *Class*, *DataType* und die anderen bereits in USE umgesetzten *Classifier* lagern ihr gemeinsames Verhalten also in *Classifier* aus.

Bei der praktischen Umsetzung wird darauf geachtet, an den vorhandenen Java-Klassen im Sourcecode möglichst wenig zu verändern. Das Verändern von vorhandenen Codeteilen lässt

sich für eine adäquate Umsetzung dieses Vorhabens jedoch nicht ganz vermeiden. Ein typisches Beispiel hierfür ist die Einführung einer neuen Basisklasse zur Generalisierung einer bestehenden Klasse. In diesem Zusammenhang muss im gesamten Quellcode die Referenz zur ursprünglichen Klasse an allen relevanten Stellen durch die neue Basisklasse ersetzt werden.

4 Entwurf

In der bisherigen Veröffentlichung von USE mit der Versionsnummer 7.1.0 werden die nach OMG spezifizierten strukturierten Datentypen noch nicht unterstützt. Um dieses Feature hinzuzufügen, werden an unterschiedlichen Stellen Anpassungen und Erweiterungen vorgenommen. Dazu gehören unter anderem das Erstellen neuer Klassen, die Anpassung der Architektur und der Vererbungshierarchien sowie die Modifikation vorhandener Programmkomponenten.

Wie in Abschnitt 2.6 erläutert, handelt es sich bei *DataType* und *Class* laut der OMG um Untertypen von *Classifier* (siehe im OMG-Standard für UML [2, p. 167]). Andere bereits in USE umgesetzte Classifier sind neben *Class Association*, *AssociationClass*, *EnumType* und *Signal*.

Vor der praktischen Umsetzung sind neben einer Einarbeitung in die Grundlagen des Projekts und das Projekt selbst der Entwurf des neuen Datentyp-Features von großer Bedeutung. Das Vorgehensmodell für die praktische Umsetzung ist agil. Das bedeutet, der Entwurf wird während der Implementierung evaluiert und gegebenenfalls angepasst. Dieses Vorgehen bietet sich bei einem Legacy-Projekt wie USE an, bei welchem die Architektur weitestgehend vorgegeben ist.

4.1 Modellierung

Dieser Abschnitt enthält Modelle in Form von Klassendiagrammen, die zur Unterstützung bei der praktischen Umsetzung dienen. Diese Modelle werden vor und während der praktischen Umsetzung entwickelt, mit dem Ziel, die Übersicht über das Vorhaben zu erleichtern und zu strukturieren.

4.1.1 Grammatik

Die Definition eines Datentyps in einem Modell soll vom Compiler fehlerfrei geparst werden. Dazu wird die Grammatik von USE angepasst und erweitert. Aus der Grammatik werden mit ANTLR die entsprechenden Parser- und Lexer-Java-Klassen generiert.

Die bisherige Grammatik wird dahingehend angepasst, dass nach dem Schlüsselwort *dataType* analog zum Schlüsselwort *class* die Definition eines Datentyps möglich ist. Es wird eine neue Regel für die Definition von Datentypen eingeführt. Dabei wird sich an der Regel zur Definition von Klassen orientiert. Jede Regel in der Grammatik besitzt zur Dokumentation eine an die EBNF angelehnte Darstellung. Listing 11 zeigt die EBNF für diese neue Regel *dataTypeDefinition*.

Listing 11: EBNF der neu eingeführten Regel *dataTypeDefinition* für die Definition von Datentypen

```
dataTypeDefinition ::=
  [ "abstract" ] "dataType" id [ specialization ]
  [ operations ]
  [ constraints ]
  "end"

specialization ::= "<" idList
operations ::= "operations" { operationDefinition }
constraints ::= "constraints" { invariantClause }
```

Mit dem optionalen Schlüsselwort *abstract* wird angegeben, ob es sich um einen abstrakten Datentyp handelt. *dataType* ist das maßgebende Schlüsselwort für die Datentypdefinition. Bei *id* handelt es sich um einen beliebig festgelegten Bezeichner des Datentyps. *specialization* ist die Liste mit den Bezeichnern der Basisdatentypen, von denen Attribute und Operationen geerbt werden. Nach dem Schlüsselwort *operations* folgen die Definitionen der Operationen einschließlich genau eines Konstruktors. Nach *constraints* werden gegebenenfalls Invarianten für den Datentyp und seine Operationen definiert. Schließlich markiert *end* das Ende der Datentypdefinition.

Die Attribute eines Datentyps werden in dieser Implementierung über einen Konstruktor definiert. Bei einer Klasse geschieht dies über das Schlüsselwort *attributes*. Das Schlüsselwort *attributes* wird für Datentypen nicht unterstützt. Der Konstruktor ist eine Operation, welche die Attribute als Argumente entgegennimmt.

Der Name des Konstruktors ist in seiner Schreibweise immer identisch mit dem Namen des Datentyps. Dies sorgt für eine neue Einschränkung bei der Namengebung von Operationen, welche es zuvor nicht gab. Der Name einer Operation in einer Klasse darf in der Schreibweise nicht mit der Bezeichnung der Klasse übereinstimmen. Dies hängt damit zusammen, dass die Regel zur Definition von Operationen allgemein für Classifier gilt, zu denen neben dem Datentyp bekanntermaßen auch die Klasse gehört. Ein Konstruktor besitzt außerdem keinen Rückgabetyp und keinen Operationsrumpf.

Die Regel zur Definition von Operationen wird dahingehend angepasst, dass hinter der Parameterliste einer beliebigen Operation noch eine Liste mit den Basisattributen von Basisdatentypen angehängt werden kann. Diese Liste hat nur bei der Definition von Konstruktoren eine Funktion und dient dazu, anzuzeigen, welche Attribute bereits in übergeordneten Datentypen vorhanden sind.

Beim Parsen einer USE-Spezifikation werden einer abgeleiteten Klasse oder einem abgeleiteten Datentyp dann automatisch alle Basisattribute und Basisoperationen hinzugefügt. Diese Attribute dürfen der Klasse oder dem Datentyp nicht noch einmal hinzugefügt werden. Im Falle des Datentyps müssen beim Aufruf eines Konstruktors neben den neu ergänzten auch alle übergeordneten Attribute als Argumente angegeben werden. Zusätzlich wird ein semantisches Attribut und eine Action benötigt, um die Definition eines Konstruktors zu handhaben (siehe Listing 1).

4.1.2 Abstrakter Syntaxbaum

Der Parser erzeugt mithilfe der AST-Klassen den *abstrakten Syntaxbaum (AST)*. Analog zur bereits vorhandenen Klasse `ASTClass` wird die Klasse `ASTDataType` erstellt. Im UML-Metamodell des OMG-Standards ist *Classifier* unter anderem die Oberklasse von *Class* und *DataType*. [2, p. 167] Da sich auf AST-Ebene bisher noch keine entsprechende Klasse

`ASTClassifier` befindet, wird eine Klasse dieses Namens hinzugefügt und Gemeinsamkeiten der anderen Classifier (wie beispielsweise die Felder `attributes` und `operations`) in diese ausgelagert.

Neben *Class* und *DataType* werden außerdem *Association*, *AssociationClass*, *EnumType* und *Signal* von *Classifier* abgeleitet. Diese Arbeit befasst sich ausschließlich mit den Classifiern *Class* und *DataType*.

Die AST-Klassen bilden den abstrakten Syntaxbaum der textuell definierten USE-Spezifikation ab. Daraus wird ein Metamodell erstellt. In der AST-Klasse zur Repräsentation einer Operation wird eine neue boolesche Klassenvariable `isConstructor` eingeführt. Wie in Abschnitt 4.1.1 erläutert, handelt es sich bei einem Konstruktor syntaktisch um eine Operation. Ein Konstruktor wird in der Grammatik und im AST wie eine Operation behandelt. Diese Designentscheidung verhindert das Einführen einer neuen Regel in der Grammatik und einer neuen AST-Klasse. Hätte man neben `operationDefinition` eine neue Regel `constructorDefinition` eingeführt, wäre diese nahezu identisch zur bereits vorhandenen Regel. Ebenso lohnt sich die Einführung einer neuen AST-Klasse `ASTConstructor` nicht, da die Information darüber, ob es sich um einen Konstruktor handelt, der einzige Unterschied zur Operation ist und die Anpassungen im Code daher minimal sind. Der Vorteil ist, dass für einen Konstruktor keine separate Regel in der Grammatik oder eine neue AST-Klasse eingeführt werden muss.

Mit der angepassten Grammatik und der neuen Klasse `ASTDataType` wird beim Parsen einer USE-Spezifikation mit mindestens einem Datentyp dieser korrekt erkannt und dem AST hinzugefügt.

Abbildung 8 zeigt die beiden neu eingeführten Klassen `ASTClassifier` und `ASTDataType` in Beziehung zu den anderen in USE vorhandenen Classifiern.

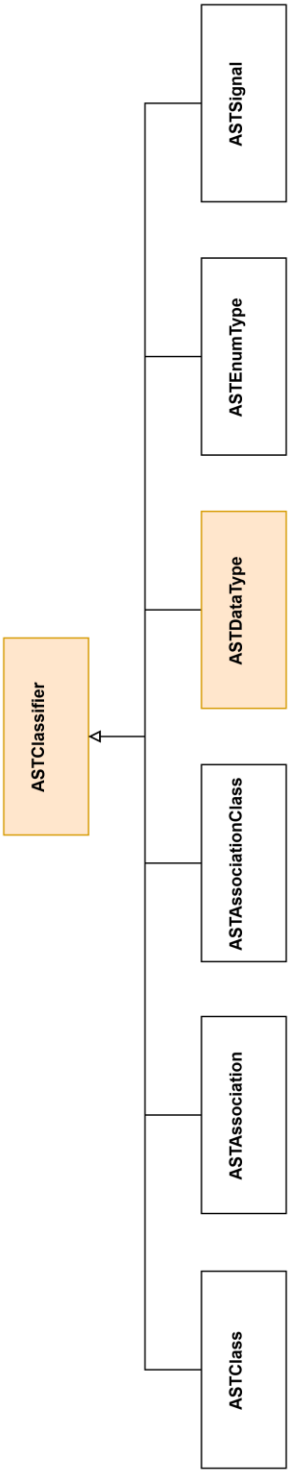

Abbildung 8: Einführung der Klassen `ASTClassifier` und `ASTDataType`

4.1.3 Metamodell

Das Metamodell kann als statische Repräsentation der USE-Spezifikation gesehen werden. Mögliche Ausdrücke in Attributen oder Operationen werden während der Erstellung des Metamodells bereits ausgewertet. Zum Metamodell gehört ein Systemzustand, welcher alle zur Laufzeit erstellten Objekte sowie deren Attributwerte beinhaltet.

Ein umfassender Teil der Implementierung beschäftigt sich mit dem Erstellen und Anpassen der Meta-Klassen. Den Namen aller Meta-Java-Klassen im Projekt wird zur Kennzeichnung ein großes „M“, vorangestellt, zum Beispiel `MClass`. Neben der Erstellung neuer passender Interfaces und Klassen für die Abbildung von Datentypen werden einige Anpassungen in den vorhandenen Klassen vorgenommen. Eine wichtige Anpassung ist, dass Attribute und Operationen jetzt nicht nur in Klassen, sondern auch in Datentypen vorkommen. Das wird beispielsweise durch das Auslagern von Attributen und Operationen in die Basisklasse `MClassifier` erreicht, wobei es nach wie vor Unterklassen von `MClassifier` gibt, welche keine Attribute und keine Operationen unterstützen.

Ein Konstruktor wird hier ebenfalls als Operation behandelt. Die Information darüber, ob es sich bei einer Operation um einen Konstruktor handelt, befindet sich bereits in der entsprechenden AST-Klasse und wird daraus weitergegeben.

Mit der Einführung von zustandslosen Datentypen und der damit verbundenen Möglichkeit von Konstruktoraufrufen in USE-Spezifikationen und auf der USE-Shell ergibt sich besonders eine Herausforderung. Bei dieser handelt es sich um die Einführung einer Repräsentation für zustandlose Classifier-Instanz. Dies wird näher in Abschnitt 4.1.6 beschrieben.

4.1.4 Ausdrücke

In USE werden unterschiedliche Arten von Ausdrücken separat behandelt. So werden Operationsaufrufe anders ausgewertet als arithmetische Ausdrücke. Die Auswertung der neuen Konstruktoraufrufe und der herkömmlichen Operationsaufrufe unterscheidet sich ebenfalls. Einer Operation wird normalerweise immer der Bezeichner einer Instanz vorangestellt, auf der diese aufgerufen werden soll. Bei der Auswertung eines Konstruktors gibt es hingegen keinen vorangestellten Bezeichner. Das Ergebnis eines Konstruktors ist selbst die Instanz eines

Datentyps. Aus diesem Grund kann eine Operation direkt auf einem Konstruktor aufgerufen werden. Da Konstruktoren zustandslos sind, muss darauf geachtet werden, dass eine auf einen Konstruktor aufgerufene Operation alle für die weitere Auswertung relevanten Variablen kennt, da für diese nicht wie bei einem zustandsbehafteten Objekt ein Lookup aus dem Systemzustand durchgeführt werden kann.

4.1.5 Werte

Die abstrakte Basisklasse `Value` ist für unterschiedliche Arten von Werten zuständig. Für die Repräsentation eines zustandsbehafteten Objektes dient bisher die davon abgeleitete Klasse `ObjectValue`.

Das Klassendiagramm in Abbildung 9 zeigt die Einführung der Klasse `InstanceValue` für die allgemeine Repräsentation von beliebigen Instanzen von Datentypen oder Klassen. `InstanceValue` wird die Klasse zur Repräsentation eines zustandsbehafteten Objektes `ObjectValue` und eine neue Klasse zur Repräsentation der Werte eines zustandslosen Datentyps `DataTypeValueValue` untergeordnet.

4.1.6 Instanzen

In der Version USE 7.1.0 werden für die Erzeugung von Classifiern bisher nur zustandsbehaftete Objekte unterstützt. Mit der Einführung von zustandslosen Datentypen wird jedoch eine Möglichkeit benötigt, eine zustandslose und nicht persistente **Instanz** eines Datentyps zu erzeugen. Zu diesem Zweck wird in der Architektur der Metaebene eine neue Klasse mit dem Namen `MInstance` eingeführt.

`MInstance` dient als neue Basisklasse für die bereits vorhandene Klasse `MObject`, welche zustandsbehaftete Klassenobjekte repräsentiert und einer neuen Klasse `MDataTypeValue`, welche Datentypeninstanzen repräsentiert. Das Konzept von Instanzen war bis dato im System nicht verankert. Die Einführung von `MInstance` ist unabhängig vom Datentyp-Feature ein wichtiger Schritt in der Entwicklung von USE. Es vereinfacht den Umgang mit Instanzen jeglicher Art deutlich. Die Einführung von `MInstance` und `MDataTypeValue` wird im Klassendiagramm in Abbildung 10 dargestellt.

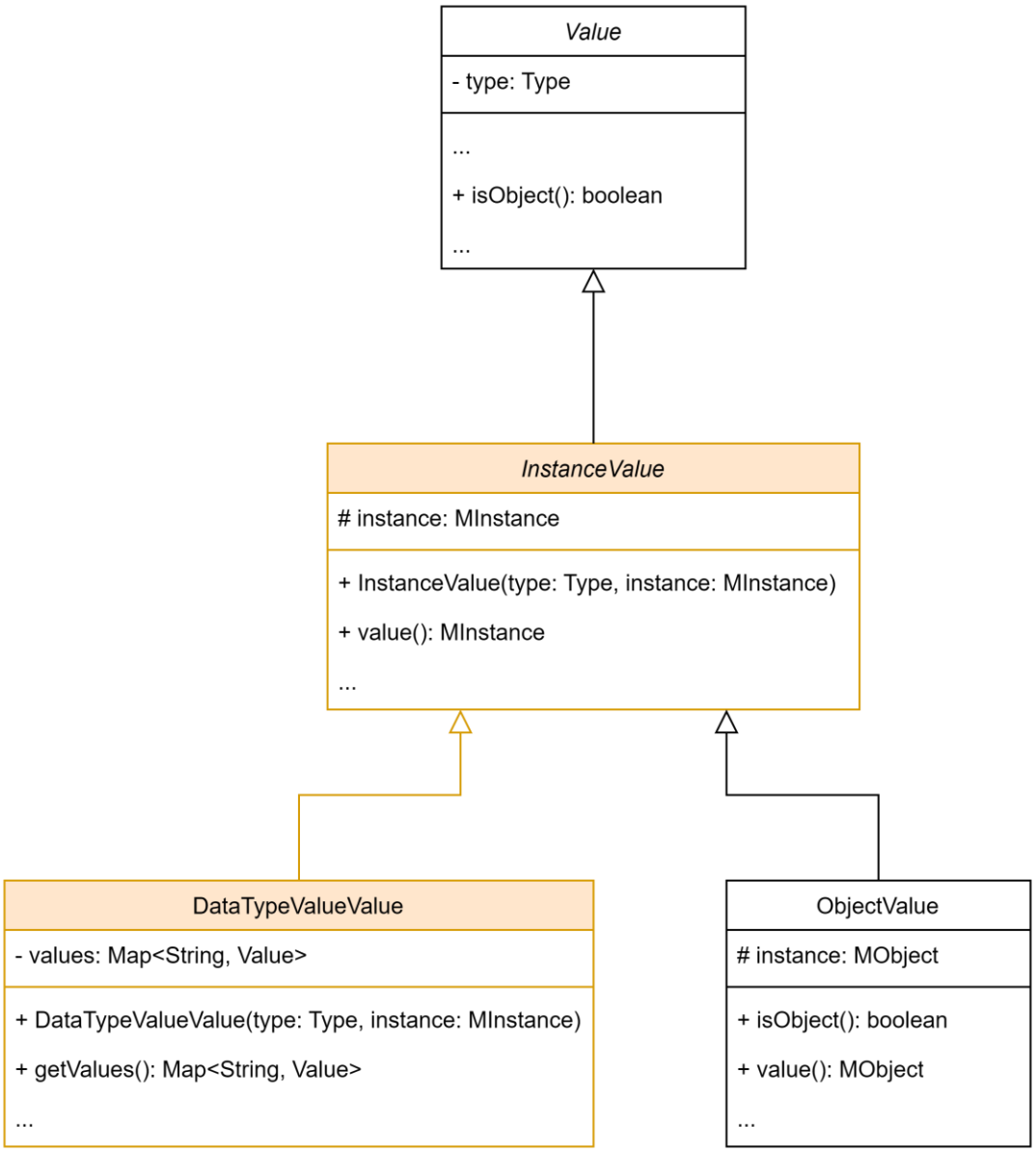

Abbildung 9: Einführung der abstrakten Klassen `InstanceValue` und `ObjectValue`

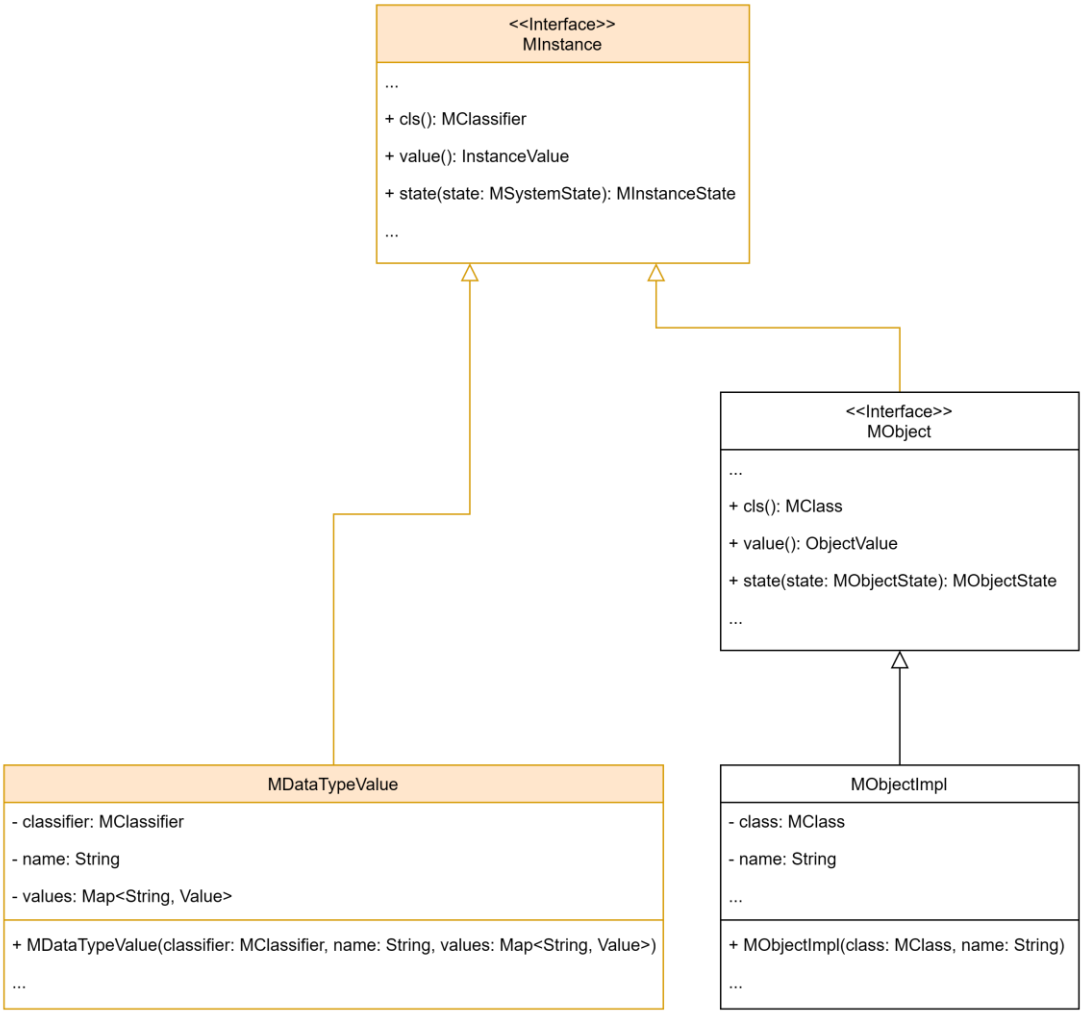

Abbildung 10: `MInstance` und `MDataTypeValue` bilden eine zustandslose Instanz ab

4.2 Dokumentation

USE besitzt eine umfangreiche Dokumentation in Form eines im Quellcode enthaltenen Markdown-Dokuments. Dort wird anhand von Beispielen beschrieben, wie eine USE-Spezifikation erstellt und in der USE-GUI und der USE-Shell verwendet wird.

Diese Dokumentation wird um eine Beschreibung für die Benutzung der neuen Datentypen anhand des Beispiels *Shapes* (siehe Listing 9) erweitert, in dem die Verwendung innerhalb einer USE-Spezifikation und auf der Shell erläutert wird.

4.3 Compiler-Test

USE besitzt umfangreiche Tests. Es verfügt über Unit-Tests für die einzelnen Programmkomponenten, wie beispielsweise die Erstellung des AST und des Metamodells. Da bei der Umsetzung des Datentyp-Features eine Erweiterung der formalen Sprache und somit Änderungen am Parser vorgenommen werden, sollte besonders dieser getestet werden. Dafür gibt es bereits einen umfassenden Compiler-Test, der einfach erweitert werden kann.

Der Ablauf des Tests ist, dass eigens dafür erstellte einfache USE-Spezifikationen automatisch vom USE-Compiler geparst werden. Dabei gibt es zwei Szenarien, die auftreten können:

1. Die jeweilige Spezifikation ist syntaktisch korrekt und wird ohne Fehler geparst.
2. Die jeweilige Spezifikation enthält bewusst einen oder mehrere syntaktische Fehler und der Compiler gibt entsprechende Hinweise (Fehlermeldungen und Warnungen) aus.

Das erste Szenario ist hilfreich, um zu testen, ob USE eine gültige Spezifikation ohne Fehler verarbeiten kann. Das zweite Szenario dient zum Testen der Fehlerbehandlung in USE. Neben der bewusst fehlerhaften Spezifikation wird eine Datei erstellt, welche den Fehlertext enthält, der beim Kompilieren erwartet wird. Damit wird getestet, ob ungültige Spezifikationen auch als solche erkannt werden und ob das System dafür eindeutige Hinweise ausgibt.

Im Falle der neu eingeführten Datentypen werden für diesen Test neue einfache Spezifikationen erstellt, welche die Definition von Datentypen und Konstruktoren enthalten. Hierbei werden auch Besonderheiten wie die Vererbung und der damit verbundenen Berücksichtigung der Basisattribute im Konstruktor getestet. Für die neuen Datentypen wird beispielweise getestet, ob die wiederholte Definition des Konstruktors einen Fehler auslöst, da dies nicht zulässig ist. Ferner wird die korrekte Fehlerbehandlung von Inkonsistenzen bei Generalisierungen so überprüft. Mit diesem Test wird also zugleich die in Kapitel 3 festgelegte korrekte Fehlerbehandlung des Systems sichergestellt.

Zusätzlich ist die Erweiterung des Compiler-Tests ein wichtiger Faktor zur Sicherstellung der in Abschnitt 3.1 gestellten funktionalen Anforderung, dass die Kompatibilität mit alten USE-Spezifikationen in USE nach wie vor gegeben ist.

4.4 Integrations-Tests

Neben dem Compiler-Test verfügt USE über zahlreiche Integrationstests. Diese haben das Ziel, das exakte Verhalten der USE-Shell zu repräsentieren. Für eine erfolgreiche Veröffentlichung einer neuen Version von USE müssen alle Integrationstests ohne Fehler verarbeitet werden.

Die Integrationstests laufen folgendermaßen ab:

1. Es wird eine ausführbare Datei zum Starten von USE aus dem Projekt erzeugt.
2. USE wird ausgeführt.
3. Ein Modell aus einer Spezifikation wird als Input in USE geladen.
4. Es werden mehrere Shell-Befehle aus einer zusätzlichen Input-Datei auf dem geladenen Modell ausgeführt. Diese Datei enthält außerdem den erwarteten Output.

Nach der Implementierung des Datentyp-Features schlägt einer der insgesamt 123 Tests fehl. Der Auslöser dafür findet sich bei der Inspektion der entsprechenden USE-Spezifikation. In dieser werden in mehreren Fällen Klassenoperationen definiert, die in der Schreibweise mit den Klassennamen übereinstimmen. In der Vergangenheit war dies zulässig. Durch das Datentyp-Feature wird die formale Sprache von USE jedoch dahingehend angepasst, dass es sich bei einer Operation, die in der Schreibweise einem Classifier-Namen gleicht, immer um einen Konstruktor handelt. Dies gilt also für alle Classifier, in denen Operationen zulässig sind, auch wenn Konstruktoren derzeit nur für den Classifier Datentyp unterstützt werden. Das Fehlschlagen des Tests wird in diesem Fall durch eine Kleinschreibung der betroffenen Operationen in der besagten USE-Spezifikation gelöst.

Wie bereits aus Abschnitt 4.1.1 hervorgeht, sollten sich Benutzer von USE darüber im Klaren sein, dass es zukünftig nicht mehr möglich ist, Operationen zu definieren, die in der Schreibweise mit einem im Modell enthaltenem Classifier übereinstimmen.

5 Ergebnis

In diesem Kapitel wird das Ergebnis der praktischen Realisierung dieser Arbeit zusammengefasst und präsentiert.

5.1 Umsetzung der Anforderungen

In diesem Abschnitt wird beurteilt, ob im umgesetzten Projekt alle in Kapitel 3 spezifizierten Anforderungen eingehalten werden. Dies wird durch das Testen mit den dafür angefertigten Unit-Tests und durch manuelles Testen der um das Datentyp-Feature erweiterten Software festgestellt.

Es lässt sich festhalten, dass alle funktionalen Anforderungen wie erwünscht erfüllt werden. Datentypen lassen sich mit dem Schlüsselwort *dataType* innerhalb eines Modells definieren. Für einen Datentyp wird ein beliebiger Bezeichner gewählt und Attribute sowie Operationen definiert. Dabei werden anders als bei der Klasse, die Attribute über den Konstruktor definiert. Jeder Datentyp besitzt genau einen Konstruktor.

Technisch ist das Datentyp-Feature so umgesetzt, dass die Instanziierung eines Datentyps zustandslos bleibt. Beim Versuch, ein Objekt aus einem Datentyp zu erzeugen, erscheint eine entsprechende Fehlermeldung.

Mit der neuen Version kann ein Konstruktor in der USE-Shell wie gewünscht überall dort eingesetzt werden, wo auch der Bezeichner eines Objektes oder ein Wert eingesetzt werden kann. Überdies ist es möglich, strukturierte Datentypen genau wie primitive Typen oder selbst definierte Typen, die Klassen oder andere Datentypen sein können, innerhalb eines Modells zu verwenden. Für Konstruktoren können Vor- aber keine Nachbedingungen definiert werden, da es durch die Zustandslosigkeit bei durch Konstruktoren erzeugte Datentypen keine

Nachbedingungen geben kann. Beim Definieren einer Nachbedingung für einen Konstruktor wird eine entsprechende Fehlermeldung ausgegeben.

Die GUI wurde dahingehend angepasst, dass analog zu den anderen Elementen aus einem geladenen Modell auch Datentypen in der seitlichen Baumstruktur (siehe Abbildung 7) und in korrekter Notation im Klassendiagramm angezeigt werden.

Es ist eine vollständige Kompatibilität mit älteren USE-Spezifikationen gegeben. Bei der Entwicklung des Features wurde auf eine lückenlose Fehlerbehandlung geachtet, sodass USE stabil bleibt und es zu keinen Fehlern kommt, die durch das neue Datentyp-Feature ausgelöst werden.

Aufgrund der Einhaltung der durch das Projekt vorgegebenen Architektur, eine dazu konsistente Erweiterung sowie eine klare Aufgabentrennung sind Erweiterbarkeit und Wartbarkeit auch für die Zukunft des Projektes gegeben.

5.2 Umsetzung der formalen Definition

Die formale Definition von Datentypen wird in Abschnitt 2.6 aus unterschiedlichen Quellen zitiert. Daraus geht hervor, dass ein strukturierter Datentyp eine Menge von Attributen ohne Identität ist. Ein Attribut besitzt einen Namen, einen Typ und kann einen beliebigen Wert seines Typs annehmen. Zwei Instanzen des gleichen Datentyps sind genau dann identisch, wenn alle ihre Attribute die jeweils gleichen Werte haben. Eine bestehende Instanz eines Datentyps wie beispielsweise ein bestimmtes Datum ist unveränderlich. Datentypen besitzen optional Operationen.

Die Datentypen in USE besitzen Attribute und Operationen. Datentypinstanzen besitzen im Gegensatz zu Klassenobjekten keine Identität. Beim Vergleich zweier Instanzen desselben Datentyps, werden die jeweiligen Werte der Attribute verglichen. Die Datentypen in USE sind zustandslos und somit unveränderlich. Operationen, die Attribute ändern, erzeugen als Ergebnis eine neue Datentypinstanz.

5.3 Einführung der Classifier-Instanz

Dieser Abschnitt beschreibt eine größere Hürde, die es bei der Umsetzung des Datentyp-Features gab. Dies beinhaltet eine Einschränkung in der Architektur. Die Architektur war rein auf den Umgang mit zustandsbehafteten Klassenobjekten ausgerichtet. Die Herausforderung war die Einführung der sogenannten Classifier-Instanz.

Wie in Abschnitt 4.1.6 erwähnt, ist das Konzept der Instanzen in USE zuvor nicht bekannt gewesen. Es gab ausschließlich zustandsbehaftete Objekte. Dieser Umstand zog sich durch das System.

Da Datentypen laut Definition zustandslos sind und die praktische Umsetzung den Anspruch hat, diese Eigenschaft adäquat umzusetzen, musste eine Lösung gefunden werden, welche allgemein die Instanziierung von Classifiern und damit auch von Datentypen zulässt.

Dafür wurden einige Teile der Architektur von USE angepasst. Diese Anpassung war eine Herausforderung, da an vielen Stellen im Projekt ein Objekt erwartet wird, an denen der Zustand des Objektes keine Rolle spielt. Abbildung 10 in Abschnitt 4.1.6 zeigt, wie dies wurde durch die Einführung des Interfaces `MInstance`, welches eine Instanz repräsentiert, behoben wird. Von diesem Interface leitet sich eine neue Klasse für Datentypinstanzen sowie ein bereits vorhandenes Interface für Klassenobjekte ab.

Die Umsetzung des Datentyp-Features wäre vermutlich deutlich einfacher gewesen, wenn die Architektur von USE von Anfang an die Existenz von zustandlosen Instanzen vorgesehen hätte. Durch die nachträgliche Einführung mussten zahlreiche kleine Änderungen in vielen Klassen vorgenommen werden.

6 Diskussion und Ausblick

In diesem Kapitel wird die Definition der Attribute und des Konstruktors diskutiert und dargelegt, wie ein Konstruktor neben der Erzeugung von Datentypinstanzen auch zur Erzeugung von Klassenobjekten verwendet werden könnte.

6.1 Definition der Attribute und des Konstruktors

Wie in Abschnitt 4.1.1 beschrieben, werden Attribute in Datentypen anders als bei Klassen nicht hinter dem Schlüsselwort *attributes*, sondern über die Parameterliste des Konstruktors definiert.

Es erscheint konsistenter, die Attribute für Datentypen genau wie bei Klassen über das Schlüsselwort *attributes* zu definieren. In diesem Fall stellt sich die Frage, wie mit der Definition des Konstruktors umgegangen werden soll. Es wird davon ausgegangen, dass der Konstruktor dann ebenfalls definiert wird. Der Vorteil einer solchen Variante ist, dass bei der Definition des Konstruktors dann auf die angehängte Liste mit den Basisattributen verzichtet werden kann, wodurch die Definition kürzer ist. Der Nachteil ist, dass die Definition des Konstruktors dann redundant erscheint. Listing 12 zeigt am Beispiel des in Listing 9 vorgestellten Modells *Shapes* anhand des Datentyps *Rectangle*, wie diese alternative Lösung aussähe. Hier werden die Attribute *width* und *heigth* von *Rectangle* nach dem Schlüsselwort *attributes* ohne das Basisattribut *position* definiert. Der Konstruktor enthält alle Attribute als Parameter.

Listing 12: Alternative Definition der Attribute nach dem Schlüsselwort *attributes* und zusätzlicher Definition des Konstruktors

```
model Shapes

-- ...

dataType Rectangle < Shape
attributes
  width : Real
  height : Real
operations
  Rectangle(position : Point, width : Real, height : Real)
  perimeter() : Real = 2.0 * width + 2.0 * height
  area() : Real = width * height
end
```

Eine weitere Möglichkeit wäre, die Attribute zwar über das Schlüsselwort *attributes* zu definieren, die explizite Konstruktordefinition jedoch auszulassen und den Konstruktor im Hintergrund automatisch zu erstellen. Der Nachteil dieser Variante ist, dass ein Modellierer wissen muss, dass der Konstruktor implizit als Operation vorhanden ist und bei dessen Verwendung neben den direkt im Datentyp definierten Attributen gegebenenfalls zunächst alle Basisattribute als Argumente übergeben werden müssen. Diese Variante ist für das Beispiel *Shapes* in Listing 9 dargestellt.

Listing 13: Alternative Definition der Attribute nach dem Schlüsselwort *attributes* ohne explizite Definition des Konstruktors

```
model Shapes

-- ...

dataType Rectangle < Shape
attributes
  width : Real
  height : Real
operations
  perimeter() : Real = 2.0 * width + 2.0 * height
  area() : Real = width * height
end
```

Die in dieser Arbeit umgesetzte Lösung, bei der die Attribute ausschließlich über den Konstruktor definiert werden, ist ein guter Kompromiss zwischen der expliziten Konstruktordefinition einerseits und einer redundanten Attributdefinition andererseits. Der Vorteil dieser Variante ist, dass der Konstruktor als spezielle Operation mit allen notwendigen Parametern definiert ist. Die Nachteile sind, dass die Definition der Attribute in Datentypen inkonsistent zu der in Klassen ist und dass hinter der Signatur des Konstruktors gegebenenfalls noch eine Liste mit Basisattributen steht. Diese Liste beeinträchtigt möglicherweise die Übersicht.

In der zukünftigen Entwicklung von USE könnte angestrebt werden, die Notwendigkeit dieser Liste allgemein zu entfernen. Die Herausforderung dabei ist, dass der Compiler beim Parsen der Konstruktorsignatur für jeden Parameter einen Lookup durchführen und feststellen muss, ob das zugehörige Attribut bereits in einem Basisdatentyp definiert wurde.

6.2 Konstruktoren für Klassen

Das Konzept der Konstruktoren wurde zusammen mit den Datentypen eingeführt. Die Verwendung von Konstruktoren muss aber nicht in Bezug auf die Verwendung von Datentypen beschränkt sein, sondern kann beispielsweise auch bei der Erzeugung von Klassenobjekten zum Einsatz kommen. Listing 14 zeigt die Initialisierung eines Objektes einer Klasse *Person*

mit den Attributen *lastname*, *firstname* und *dateOfBirth* ohne die Verwendung eines Konstruktors.

Listing 14: Klassische Initialisierung eines Objektes der Klasse *Person*

```
use> !create myPerson : Person
use> !set myPerson.lastname := Mustermann
use> !set myPerson.firstname := Max
use> !set myPerson.dateOfBirth := Date(1, 1, 1900)
```

In Listing 15 wird diese Initialisierung mithilfe eines Konstruktors durchgeführt. Statt den bisher vier Befehlen ist damit nur noch ein Befehl nötig.

Listing 15: Vereinfachte Initialisierung eines Objektes *Person*

```
use> !create myPerson : Person(Max, Mustermann, Date(1, 1, 1900))
```

7 Zusammenfassung

Das Ziel dieser Arbeit war die Erweiterung des UML- und OCL-Modellierungswerkzeugs **USE**, sodass es zukünftig **strukturierte Datentypen** unterstützt. Dazu wurden die relevanten theoretischen Grundlagen aus der Literatur zusammengetragen. Zunächst wurden die Themen Compilerbau und Spracherkennung detailliert betrachtet, weil diese für die praktische Weiterentwicklung der formalen Sprache von USE von großer Relevanz waren. Es wurden zudem die wichtigsten Elemente der UML und OCL beschrieben und das System USE erläutert. Separat wurde sich detailliert mit dem theoretischen Hintergrund von strukturierten Datentypen beschäftigt sowie ein Überblick über die Konzepte der Referenz- und Wertesemantik und positionelle und benannte Attribute gegeben.

Die Spezifikation besteht aus dem konkreten Festlegen der Funktionen und der Art der Verwendung des neuen Datentyp-Features. Darin wurden alle Teile des Programms identifiziert, die verändert und angepasst werden müssen. Die Spezifikation beschreibt dabei, was genau umgesetzt werden sollte.

Der Entwurf befasste sich besonders mit der Modellierung aber auch mit der Dokumentation und dem Testen der Umsetzung. Bei der Modellierung wurde nach Lösungen für diejenigen Herausforderungen gesucht, die sich durch die bestehende Architektur des Projektes ergeben und die genaue Struktur sowie die Generalisierungshierarchie bestimmter Teile im Projekt betrachtet, um eine solide Lösung für die strukturierten Datentypen entwerfen zu können. Bei diesem agilen Prozess wurden Klassendiagramme bestimmter Komponenten entwickelt, die einen Beitrag zur erfolgreichen praktischen Umsetzung leisten. Im Entwurf wurde somit dargelegt, wie bei der praktischen Umsetzung vorgegangen wurde. Das Kapitel beinhaltet ebenfalls Informationen über die Dokumentation und das Testen des neuen Datentyp-Features.

Im Ergebnis wurde evaluiert und bestätigt, dass das fertig umgesetzte Datentyp-Feature den aufgenommenen Anforderungen entspricht und dabei der formalen Definition von Datentypen gerecht wird.

Durch die Einführung des Datentyp-Features wurde ebenfalls das Konzept der Konstruktoren für Datentypen eingeführt. Im Kapitel „Diskussion und Ausblick“ wurde zunächst diskutiert, welche Variationen es bei der Implementierung des Konstruktor-Features gibt. Es werden Vor- und Nachteile dieser Variationen genannt und es wurde begründet, welche von den Möglichkeiten letztendlich praktisch umgesetzt wird. Darüber hinaus wurde ein Ausblick darüber gegeben, wie dieses Konstruktor-Feature in Zukunft auch für Klassen in USE umgesetzt werden könnte. Dabei wurde klar, dass Konstruktoren für Klassen eine sinnvolle Erweiterung sind, um schneller Klassenobjekte mit bestimmten Attributen zu erzeugen.

Literaturverzeichnis

[1] „USE - UML-based Specification Environment,“ [Online]. Available: https://github.com/useocl/use/. [Zugriff am 28 05 2024].

[2] Object Modelling Group (OMG), „OMG Unified Modeling Language (OMG UML),“ Dezember 2017. [Online]. Available: http://www.omg.org/spec/UML/. [Zugriff am 05 06 2024].

[3] A. Aho, M. S. Lam, R. Sethi, J. D. Ullman und M. Leuschel, Compiler : Prinzipien, Techniken und Werkzeuge, 2., aktualisierte Aufl. [der engl. Ausg.] Hrsg., München u. a.: Pearson Studium, 2008.

[4] T. Parr, The definitive ANTLR reference : building domain-specific languages, Raleigh, NC u. a.: Pragmatic Bookshelf, 2007.

[5] C. Kecher und A. Salvanos, UML 2.5: Das umfassende Handbuch, Bonn: Rheinwerk Computing, 2015.

[6] C. Rupp und S. Queins, UML 2 glasklar : Praxiswissen für die UML-Modellierung, München: Hanser Verlag, 2012.

[7] Object Modelling Group (OMG), „Object Constraint Language,“ Februar 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4. [Zugriff am 05 06 2024].

[8] B. Demuth, „OCL By Example Lecture - TU Dresden,“ 2009. [Online]. Available: https://st.inf.tu-dresden.de/files/general/OCLByExampleLecture.pdf. [Zugriff am 05 06 2024].

[9] C. Larman, Applying UML and Patterns, Upper Saddle River: Pearson, 2005.

[10] B. Oestereich und S. Bremer, Analyse und Design mit der UML 2.5 - Objektorientierte Softwareentwicklung, München: Oldenbourg Verlag, 2012.

[11] I. Sommerville, Software Engineering, Hallbergmoos: Pearson, 2018.

A Anwendungsfälle

Titel	Datentyp definieren
Akteur	Modellierer
Ziel	Es wird ein Datentyp in einer Spezifikation modelliert
Auslöser	Keinen
Vorbedingungen	Es gibt eine Spezifikation mit einem Modell
Nachbedingungen	Der Datentyp befindet sich in dem Modell
Erfolgsszenario	1. Der Modellierer öffnet die Spezifikation mit einem beliebigen Texteditor. 2. Der Modellierer definiert den Namen eines neuen Datentyps mithilfe des Schlüsselworts *dataType* innerhalb des Modells 3. Der Modellierer speichert die Änderungen an der Spezifikation 4. Die Spezifikation mit dem Modell wird fehlerfrei geparst

Titel	Konstruktor definieren
Akteur	Modellierer
Ziel	Es wird genau ein Konstruktor für jeden Datentyp in einem Modell definiert
Auslöser	Keinen
Vorbedingungen	Es befindet sich mindestens ein Datentyp in einem Modell
Nachbedingungen	Die Attribute und der Konstruktor wurden definiert
Erfolgsszenario	1. Der Modellierer definiert einen zum Datentyp gleichnamigen Konstruktor mit beliebigen Attributen als Parameter 2. Der Modellierer speichert die Änderungen an der Spezifikation 3. Die Spezifikation mit dem Modell wird fehlerfrei geparst

Titel	Datentyp mittels eines Konstruktors erzeugen
Akteur	Modellierer
Ziel	Eine zustandslose Instanz eines Datentyps erstellen
Auslöser	Keiner
Vorbedingungen	Es gibt ein Modell mit mindestens einem Datentyp und genau einem Konstruktor
Nachbedingungen	Es wurde eine zustandslose Instanz des Datentyps erstellt.
Erfolgsszenario	1. Der Benutzer gibt einen Query-Befehl („?“) gefolgt von einer Konstruktorbezeichnung mit den geforderten Argumenten in die USE-Shell ein. 2. Das System erzeugt eine Instanz des zugehörigen Datentyps und gibt dazu eine passende Zeichenkette auf der Konsole aus.

B Dokumentation

Die offizielle USE-Dokumentation wurde im Zuge des neuen Datentyp-Features um den nachfolgenden Teil erweitert:

Data types

Data types may be added at the top of the model body.

Syntax

```
<datatypedefinition> ::= [ abstract ] dataType <datatypename>
                         [ < <datatypename> { , <datatypename> } ]
                         [ operations { <operationdeclaration>
                         [ = <oclexpression> ]
                         { <preconditiondefinition>
                          | <postconditiondefinition> } } ]
                         [ constraints { <invariantdefinition> } ]
                         end
      <datatypename> ::= <name>
```

Example:

The example shows four different data type definitions. The data types 'Rectangle' and 'Circle' inherit from abstract data type 'Shape'. 'Shape' defines the attribute 'position' which is of type 'Point' in its constructor. 'Point' defines the attributes 'x' and 'y' which are both of primitive type 'Real'. It also defines the operation 'translate()' which takes two parameters of type 'Real' for shifting 'x' and 'y' with no result type. 'Shape' defines the two operations 'perimeter()' and 'area()'. Since 'Shape' is abstract both 'Rectangle' and 'Circle' implement their specific logic for

these operations. Note that a constructor can only have pre- and no postconditions, since it is a stateless operation.

```
dataType Point
operations
  Point(x : Real, y : Real)
  translate(dx : Real, dy : Real)(dx) : Point =
    Point(self.x + dx, self.y + dy)
end

abstract dataType Shape
operations
  Shape(position : Point)
  perimeter() : Real
  area() : Real
end

dataType Rectangle < Shape
operations
  Rectangle(position : Point, width : Real, height : Real)(position)
  perimeter() : Real = 2.0 * width + 2.0 * height
  area() : Real = width * height
end

dataType Circle < Shape
operations
  Circle(position : Point, radius : Real)(position)
  perimeter() : Real = 2.0 * 3.14 * radius
  area() : Real = 3.14 * radius * radius
end
```

Following example shows, how to query data types from the above model 'Shapes' in the shell:

```
use> ?Point(1, 1)
-> Point{x=1, y=1} : Point
use> ?Point(1, 1).translate(3, 4)
-> Point{x=4, y=5} : Point
use> ?Rectangle(Point(2, 2), 2, 3).perimeter()
-> 10.0 : Real
use> ?Circle(Point(2, 6), 2, 3).area()
-> 12.56 : Real
```

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original