Karlsruhe Institute of Technology

Programmieren in natürlicher Sprache: Extraktion einer Alice-API-Ontologie

Bachelorarbeit
von

Oleg Peters

am Institut für Programmstrukturen und Datenorganisation
der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter:	Dipl. Inform.-Wirt Mathias Landhäußer

Bearbeitungszeit: 07. August 2012 – 07. November 2012

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, 07.11.2012

.....

(**Oleg Peters**)

Inhaltsverzeichnis

1. Motivation

Moderne Programmiersprachen erlauben es den Entwicklern, komplexe Verhaltensweisen von Software schnell umzusetzen. Dabei profitiert der Programmierer oft von Rahmenwerken (Frameworks), welche für bestimmte Aufgabenstellungen konzipiert sind und fertige Bausteine für deren Umsetzung liefern. Dadurch wird es möglich, Komponenten in den Anwendungen einzusetzen, ohne sich tief gehende technische Kenntnisse der darunter liegenden Architekturschichten aneignen zu müssen.

Das objektorientierte Rahmenwerk *Alice* [Car] bietet auch Nicht-Programmierern die Möglichkeit, 3D-Videos zu generieren und sogar einfache Spiele zu programmieren. Dabei kann der Benutzer auf vordefinierte 3D-Modelle aus der Alice-Bibliothek zugreifen, diese im Sichtfeld der virtuellen Kamera platzieren und Bewegungsabläufe für die Figuren per Drag-n-Drop definieren, indem er die nötigen Methodenaufrufe einfach in das Editierfenster zieht (Abb. 2.1) [Con97].

Der Ansatz der Programmierbarkeit von einfachen Zusammenhängen durch Nicht-Informatiker ist besonders interessant, und lässt sich noch weiter ausbauen. So wäre es sinnvoll, die Tatsache auszunutzen, dass Namen und Eigenschaften der Modelle in der *Alice*-Welt den Bezeichnungen aus der realen Welt entsprechen. Diese Informationen lassen sich mit den Begriffen aus der natürlichen Sprache assoziieren. So ließe sich eine Software bauen, welche aus einer textuellen Beschreibung ein Skript für das *Alice*-Rahmenwerk generieren könnte.

1.1. Programmieren mit natürlicher Sprache

Das Gesamtprojekt hat das Ziel, den Ansatz natürlichsprachlicher Programmierung zu validieren, indem man zunächst versucht, einfache Zusammenhänge in abgeschlossenen Anwendungsdomänen zu beschreiben. Als Domäne eignet sich das Alice-Rahmenwerk sehr gut, weil es ein abgeschlossenes Modell der realen Welt mit vielen bereits definierten Objekten zur Verfügung stellt [Con97]. Als Ergebnis des Projekts wird ein System angestrebt, welches aus gegebenen textuellen Beschreibungen eines Handlungsablaufs Alice-Videos erstellen kann.

Die Idee ist es, bereits vorhandene Modelle aus den Alice-Welten mit ihren Eigenschaften systematisch in eine Alice-API-Ontologie zu extrahieren. Dieses Teilziel verfolgen wir im Rahmen dieser Bachelorarbeit. Als zweiter Schritt muss man diese Ontologie mit Synonymen bzw. Umschreibungen anreichern, um noch weitere Begriffe der natürlichen Sprache

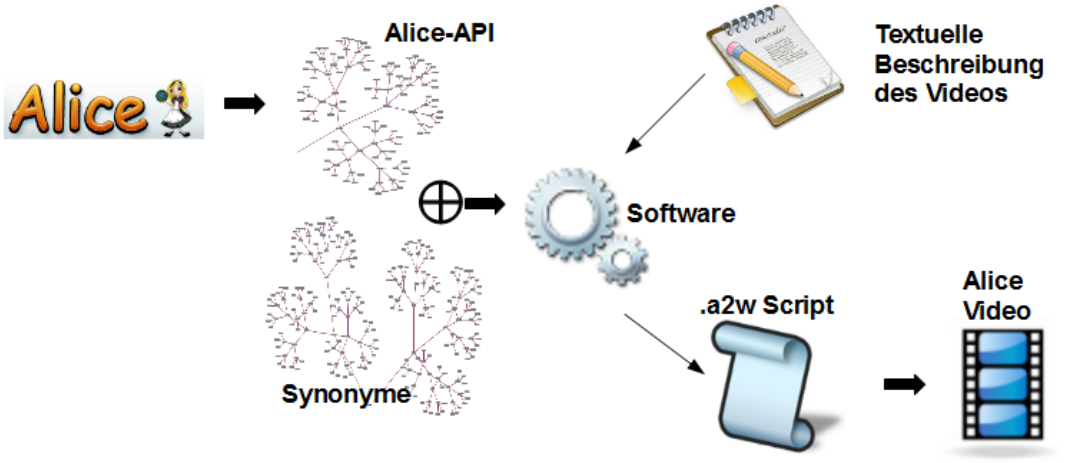


Abbildung 1.1.: Ideenskizze Programmieren in natürlicher Sprache

mit den Eigenschaften der Alice-Modelle verknüpfen zu können. Danach soll ein Anfrage-System entwickelt werden, welches imstande sein soll, für beliebige Wörter Entsprechungen in der Alice-Ontologie zu ermitteln. Dieses Anfrage-System soll das Herzstück einer Software bilden, welche textuelle Beschreibungen auswerten und Alice-Skripte für die Videos generieren kann (siehe Abbildung 1.1).

1.2. Extraktion der Alice-API-Ontologie

Ziel dieser Bachelorarbeit ist es, die Architektur des Alice-Rahmenwerks zu analysieren und alle für das Gesamtprojekt relevanten Eigenschaften zu identifizieren. Diese Informationen sollen sowohl der Umsetzung dieser Arbeit, als auch als Schnelleinstieg für die folgenden Arbeiten an dem Gesamtprojekt dienen.

Anschließend soll ein Grundgerüst der Alice-API-Ontologie entworfen werden, die es erlaubt, die Alice-Konstrukte eindeutig und strukturerhaltend abzubilden. Der ebenfalls in dieser Arbeit entstehende Extraktor soll in der Lage sein, die in Alice verfügbaren Konstrukte des Rahmenwerks und die existierenden 3D-Welten und -Objekte mit ihren Eigenschaften, Methoden und Parametern auf die Alice-API-Ontologie automatisch abzubilden.

1.3. Aufbau der Arbeit

Kapitel 2 erläutert die Grundlagen des behandelten Themas. In Kapitel 3 diskutieren wir die zu unserer Aufgabenstellung verwandten Arbeiten. In Kapitel 4 analysieren wir die Struktur des Rahmenwerks und entwerfen eine geeignete Ontologie, auf die das Alice-API abgebildet werden soll. Kapitel 5 stellt unsere Implementierung des Extraktors vor. Eine Evaluation des Ansatzes wird in Kapitel 6 durchgeführt und in Kapitel 7 fassen wir die Ergebnisse zusammen.

2. Grundlagen

2.1. Alice

Alice ist ein Werkzeug, das es dem Benutzer erlaubt, ein zeitbasiertes oder interaktives Verhalten von 3D-Objekten zu beschreiben. Damit kann der Benutzer von Alice Abläufe von Handlungen programmieren, die entweder in zeitlicher Reihenfolge, oder auch abhängig von Benutzerinteraktionen (z.B. Mausklicks) stattfinden. Das Programm, das so entsteht, kann ebenfalls in Alice ausgeführt und als Videosequenz wiedergegeben werden.

2.1.1. Allgemeiner Aufbau

Die Benutzeroberfläche des Alice-Rahmenwerk (Abb. 2.1) kann in drei Bereiche unterteilt werden: die linke Seite enthält eine Objekt-Übersicht (1), einen Bereich (2) mit den Eigenschaften des aktuell ausgewählten Objekts; in der unteren Hälfte der restlichen Fläche findet man den sogenannten Skript-Editor (3), in dem der eigentliche Programmcode erstellt wird und oberhalb davon die Vorschau der Szenerie (4) und der Ereigniseditor (5).

2.1.2. Programmierung in Alice

Grundsätzlich erfolgt die Programmierung mit dem Alice-Rahmenwerk in zwei Phasen: Als erstes muss eine Ausgangsszenerie mit allen benötigten Objekten erstellt werden. Im zweiten Schritt definiert man das Verhalten der Objekte in der Videosequenz.

Das Einfügen der Modelle in die 3D-Szene erfolgt aus der Objekt-Galerie, die nach dem anklicken der Schaltfläche *„Add Objects“* erscheint. Alice bietet eine umfassende Bibliothek vordefinierter Modelle, die mit dem Rahmenwerk mitgeliefert werden.

Sind die benötigten Objekte einmal in die Szene geladen, so können im Script-Editor Abläufe definiert werden. Dies erfolgt, indem der Benutzer die benötigten Methodenaufrufe, Attribute und Alice-Sprachkonstrukte per *Drag-N-Drop* in das Skript-Feld zieht. Zusätzlich kann der Benutzer eigene Methoden und Attribute definieren und beliebig aufrufen. Herkömmliches Programmieren mit der Tastatur ist in Alice nicht erlaubt, was Syntaxfehler aus dem Entwicklungsprozess eliminiert.

2.1.3. Modelle

Das Alice-Rahmenwerk bietet keine Unterstützung zum Erstellen von 3D-Körpern sondern konzentriert sich auf das Beschreiben des Verhaltens bereits modellierter Objekte.

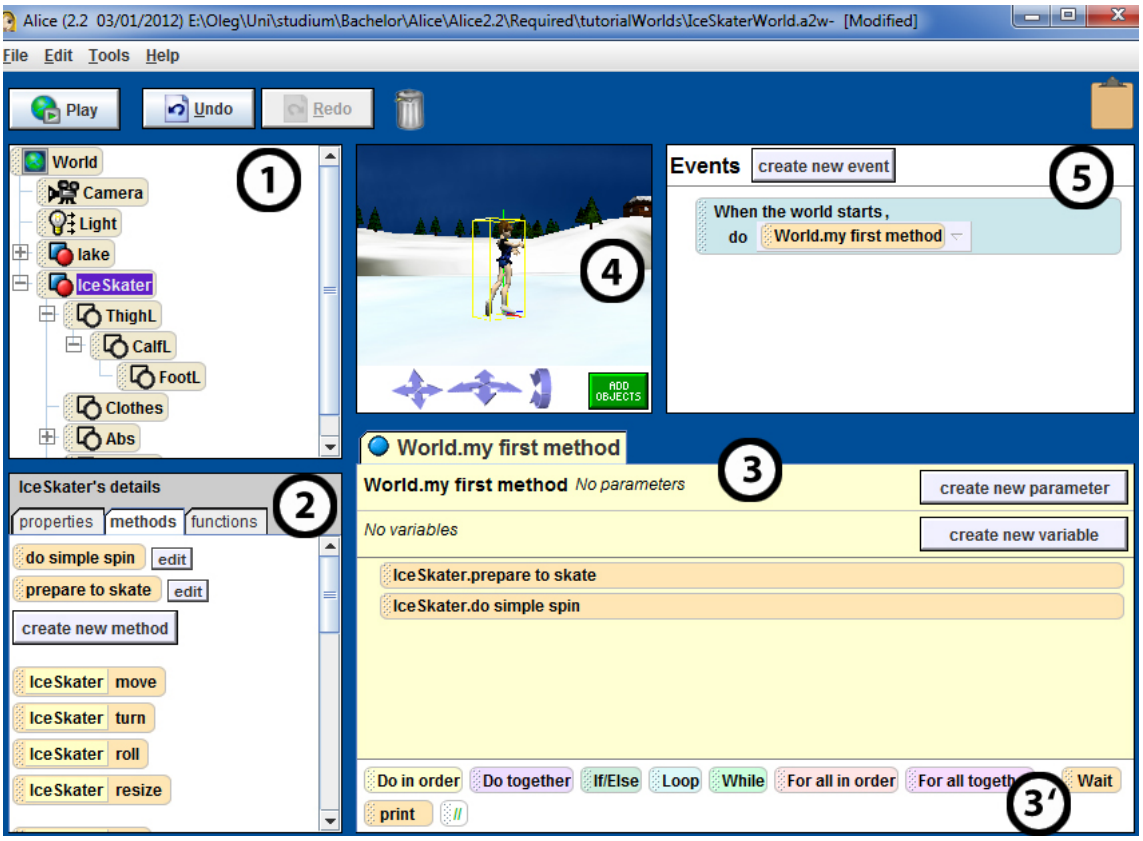


Abbildung 2.1.: Screenshot der Alice-Benutzeroberfläche

Diese werden in dem Objektbaum (Abb. 2.1, (1)) verwaltet, welcher alle in der Szenerie existierenden Objekte mit dem *Welt*-Objekt als Wurzel, enthält. Zusätzlich werden hier einige spezielle Objekte (wie z.B. Kamera, Licht, Boden) als Kind-Knoten erster Ordnung aufgeführt.

Alle Objekte sind ebenfalls hierarchisch aufgebaut und enthalten ihre Bestandteile als Unterknoten. So sind bspw. die Körperteile (Kopf, Beine, Hände) eines Eiskunstläufer-Modells als seine Kind-Knoten gespeichert. Der Eiskunstläufer selbst ist als Kind-Knoten des Objekts *Welt* im Objektdiagramm verankert, wie man in Abb. 2.1 (1) sehen kann.

2.1.3.1. Attribute

Objekte besitzen in Alice vordefinierte Attribute. Ist ein Modell im Objektbaum ausgewählt, erscheinen dessen Attribute in dem Eigenschaften-Bereich (Abb. 2.1, (2)) unter dem Reiter '*properties*'. Zu den Standardeigenschaften eines Objekts gehören zum Beispiel seine Farbe, Opazität oder Oberflächentextur. Die Attribute können in dem Skript zur Laufzeit der Videosequenz mit dem Aufruf der entsprechenden *Setter*-Methode verändert werden.

2.1.3.2. Methoden

Ebenso wie Attribute sind für alle Objekte Standardmethoden vordefiniert. Sie bilden die Grundlage für alle Methoden, die der Benutzer für die Objekte erstellt. Alice unterteilt die Methoden in zwei Kategorien und nennt sie Methoden (*methods*) und Funktionen (*functions*).

Unter Methoden versteht Alice alle Funktionen ohne Rückgabewert, die im Laufe der Videosequenz sichtbar ausgeführt werden. Sie sind für den Benutzer unter dem Reiter

methods aufgeführt und enthalten sowohl vordefinierte als auch vom Benutzer definierte Methoden. Beispiele für Methoden wären *move()*, *turn()* oder *play sound()*.

Mit Funktionen sind alle Funktionen gemeint, die einen Rückgabewert haben. Sie werden für Datenabfragen in den Skripten (z.B. *'get distance to()'*), mathematische Ausdrücke (z.B *sin*) oder boolesche Ausdrücke (z.B. *'<'*) verwendet. Diese Funktionen sind unter dem Reiter *functions* aufgelistet und können ebenfalls um benutzerdefinierte Funktionen erweitert werden.

Der Anwender kann eigene Methoden definieren indem er *'create new method'* bzw. *'create new function'* in dem Eigenschaften-Bereich anklickt. Der Methodenrumpf kann anschließend in dem Script-Editor programmiert werden und steht danach zur Verwendung bereit. Beispielsweise könnte eine zusammengesetzte Methode `Arm.wave()` definiert werden, die aus den beiden primitiven Methoden `Arm.move()` und `Arm.turn()` besteht.

2.1.4. Ereignisse

Im Ereigniseditor (Abb. 2.1, (5)) lassen sich Ereignisse (*Events*) definieren, die den Aufruf gewünschter Methoden auslösen können. Auf diese Weise kann im Rahmenwerk ein interaktives Verhalten der erstellten Anwendung modelliert werden.

Die Ereignisse sind im Rahmenwerk fest definiert und können nicht vom Benutzer selbst erstellt werden. Alice liefert die Möglichkeit, auf Maus- oder Tastatureingaben zu reagieren, logische Ausdrücke zu überwachen oder die Änderung einer Variable zu observieren.

Als Bequemlichkeit gibt es sogar in der Ereignisauswahl die Option, die Maus oder die Pfeil-Tasten der Tastatur zur Steuerung der Kamera oder eines Objekts zu benutzen. Damit ist es selbst für Neulinge ein leichtes, eine kleines interaktives 3D-Spiel zu programmieren.

2.1.5. Programmkonstrukte

Alice unterstützt alle gängigen Steuerkonstrukte, die in der Welt der prozeduralen Programmierung zum Einsatz kommen. So kann neben der unentbehrlichen *if-else*-Verzweigung eine *for*-Schleife (in Alice *'Loop'* genannt) oder die *while*-Schleife eingesetzt werden.

Zudem verfügt das Rahmenwerk über die beiden Konstrukte *'Do in order'* und *'Do Together'*, die für die Ausführung beliebig vieler Methodenaufrufe nacheinander bzw. gleichzeitig (parallelisiert) sorgen. Ergänzend dazu gibt es noch die Methode *'Wait'*, die es erlaubt, Pausen während der Animation einzubauen.

Für die Iteration durch Listen von Objekten existiert außerdem das Konstrukt *'For all'* (als Pendant zu *'for each'* in den traditionellen Programmiersprachen), das ebenfalls in den beiden Ausführungen *'For all in order'* und *'For all together'* verfügbar ist.

2.2. Ontologien

Der Begriff Ontologie entstammt dem Griechischen Wort *Ontologia*, was soviel bedeutet wie „Lehre vom Sein“. Ursprünglich handelt es sich um eine philosophische Disziplin, die sich mit den Grundstrukturen der Wirklichkeit beschäftigt.

In der Informatik bezeichnet Ontologie allgemein ausgedrückt eine formale Konzeptualisierung eines Wissensbereichs („a formal conceptualization of a knowledge domain“, ([Gru93])). Im Gegenteil zum philosophischen Begriff wird in der Informatik nicht versucht, eine universale Ontologie für alle Begriffe und deren Beziehungen zu erreichen. Vielmehr werden hier viele anwendungsspezifische Ontologien aufgebaut, die das Wissen eines Wissensbereichs (oder Domäne) enthalten.

Allgemein werden Ontologien verwendet, um bereits bestehende Wissensbestände zu bündeln oder darin zu suchen und sie zu verändern. Der entscheidende Vorteil von Ontologien gegenüber herkömmlichen Datenbanken besteht jedoch in der automatischen Inferenz von Wissen. Anhand von dem bereits gespeicherten Wissen kann in der Ontologie auf nicht explizit beschriebene Beziehungen der Objekte geschlossen werden. Aus diesem Grund ist die Ontologie inzwischen zu einer weit verbreiteten Form der Wissensdatenbank geworden.

2.2.1. Klassen und Instanzen

Eine Ontologie wird typischerweise durch Klassen, ihre Instanzen und Beziehungen dazwischen beschrieben. Klassen (auch *Begriffe* oder *Konzepte* genannt) repräsentieren dabei die Allgemeinbegriffe, also abstrakte Konzepte eines Interessensgebiets. Klassen können hierarchisch über ‘*ist-ein*’ Beziehungen strukturiert werden (siehe [Wel09]).

Die Gesamtheit aller Klassen die zu einer einzigen hierarchischen Struktur zusammengefasst sind, heißt *Taxonomie* und bildet das Grundgerüst einer Ontologie. Die Taxonomie enthält keine Informationen zu den Relationen zwischen den Objekten und dient zur Klassifizierung der Objekte innerhalb einer Hierarchie.

Instanzen (auch *Individuen* oder Objekte) repräsentieren konkrete Vertreter der Klassen. Der Typ einer Instanz ist durch den Klassennamen festgelegt.

So kann beispielsweise eine Taxonomie mit der Klasse `Lebewesen` definiert werden, welche der Oberbegriff für die Subklassen `Mensch` und `Hund` ist. Dies würde aussagen, dass alle Menschen und Hunde gleichzeitig auch Lebewesen sind.
Als Instanzen wären dann die Individuen *Peter*, *Sara* (aus der Klasse `Mensch`) und *Mike* (aus der Klasse `Hund`) denkbar (siehe Abb. 2.2).

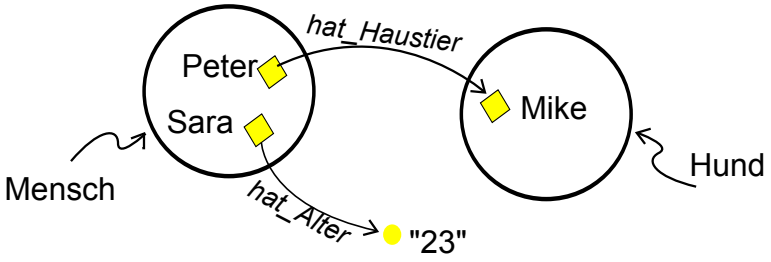


Abbildung 2.2.: Eine Beispielontologie

2.2.2. Properties

Begriffe oder ihre Instanzen können noch weiter spezifiziert werden, indem man sog. Properties einsetzt. Properties stellen semantische Relationen dar. Man kann Properties in zwei Kategorien unterteilen (siehe [OWL]):

1. Relationen zwischen zwei Klassen oder Instanzen und
2. eine Zuweisung einer Eigenschaft mit einem gewissen Wert einer Klasse/Instanz.

In dem Beispiel von oben definieren wir (im Sinne von 1.) eine Relation '*hat_Haustier*', die der Instanz '*Peter*' vom Typ `Mensch` ein Individuum '*Mike*' aus der Klasse `Hund` zuweist. Als eine weitere Property zu einer Instanzeigenschaft definieren wir für das Individuum '*Sara*' die Relation '*hat_Alter*' mit dem numerischen Wert "23" (Abb. 2.2).

2.2.3. Semantischer Reasoner

Um die Vorteile der Ontologien auszunutzen, werden sie mit Inferenzmaschinen (sogenannten semantischen Reasonern) untersucht. Ein Reasoner enthält Algorithmen, die die Ableitung von implizitem Wissen und Beantwortung von Suchanfragen an die Ontologie ermöglichen. Um implizite Informationen zu erhalten, wertet der Reasoner logische Ableitungsregeln aus. Voraussetzung für die Auswertung ist die Konsistenz der Ontologie.

Es sind bereits verschiedene Implementierungen wie Fact++, Pellet, RacerPro uvm. verfügbar. Eine Übersicht über moderne Reasoner gibt Liebig in dem technischen Bericht [Lie06].

2.2.4. Beschreibungssprachen

Eine erste Konsequenz der Verwendung von Ontologien als Datenmodell ist die Notwendigkeit standardisierter Modellierungssprachen, welche es erlauben würden, universale Werkzeuge und Methoden zur Unterstützung der Ontologiekonzeption und -befüllung zu entwickeln. Dazu wurden seit Ende der achtziger Jahre Beschreibungslogiken entwickelt, um terminologisches Wissen angemessen formalisieren zu können. In den letzten Jahren ist die Bedeutung von Beschreibungslogiken als Grundlage für die Darstellung von Ontologien enorm gestiegen. Dies ist vor allem auf die Standardisierung der Web Ontology Language (OWL) als Ontologiesprache für das Semantic-Web zurückzuführen. ([Stu09], SS. 98, 129, 148)

Als Ontologie-Beschreibungssprache wird in dieser Bachelorarbeit OWL verwendet. Vorgeschlagen vom World Wide Web Konsortium (W3C) basiert OWL ([W3C]) auf der RDF-Syntax, die zur formalen Beschreibung von logischen Aussagen über Ressourcen dient. OWL ist die Grundlage für viele Entwicklungen im Bereich Semantic Web und wurde hierdurch inzwischen zur meist benutzten Ontologiesprache aller Zeiten ([Stu09], S. 148).

2.2.5. Protege

Außer der Entwicklung von Ontologien mit Modellierungssprachen existieren inzwischen auch Werkzeuge, die eine graphisch unterstützte Modellierung erlauben. Zu solchen Werkzeugen gehört Protégé ([Sta]).

Protégé ist ein graphischer Editor, der den Anwender beim Erstellen von Ontologien durch spezielle Werkzeuge und fortgeschrittene Inferenzmethoden unterstützt ([Stu09], S. 99). In Abb. 2.3 ist ein Screenshot der Benutzeroberfläche von Protégé dargestellt. Neben diversen Sichten auf die Ontologie bietet Protégé die Möglichkeit mit unterschiedlichen Reasonern zu arbeiten (z.B. Fact++, HermiT). Die Ergebnisse der Inferenz werden anschließend auch in der Oberfläche dargestellt. Durch den Einsatz dieser Inferenzmethoden ist der Anwender zum Beispiel in der Lage, Modellierungsfehler schnell aufzudecken.

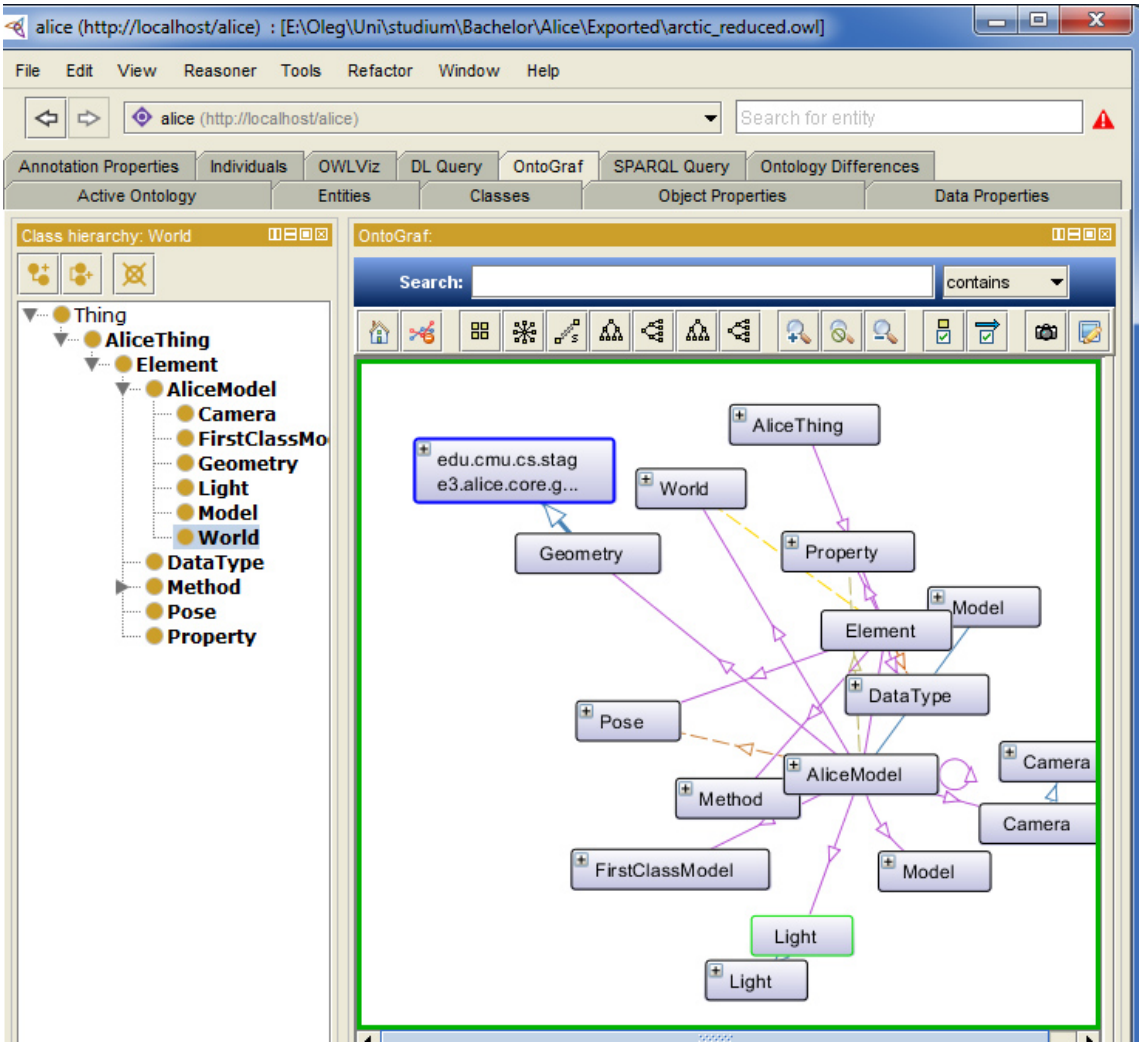


Abbildung 2.3.: Die graphische Oberfläche von Protégé

3. Verwandte Arbeiten

Eine zentrale Rolle in dieser Bachelorarbeit spielt der Aufbau einer Wissensdatenbank aus Modellen und Szenerien, die aus Alice gewonnen werden. In der Informatik werden für den Aufbau solcher Datenbanken Ontologien verwendet. Zu dem Thema des Aufbaus einer Ontologie aus bestehenden Anwendungen existieren bereits diverse Studien und Forschungsarbeiten.

So beschreiben Yang et al. in ihrem Paper [YCO99] die Extraktion von Wissen über Geschäftsprozesse in eine Ontologie als Startpunkt für das Re-Engineering komplexer Altsysteme. Dazu definieren die Autoren die Ontologie als einen Begriffsrahmen, bestehend aus Klassen, Relationen, Funktionen und Instanzen, auf welchen anschließend die Begrifflichkeiten aus den analysiertem Quellcode abgebildet werden. Der Aufbau der Ontologie mit der Grundstruktur des Alice-Rahmenwerks hat eine ähnliche Struktur erhalten, in der Klassen und deren Funktionen und Parameter durch Objektrelationen untereinander verknüpft sind.

Das von Zhang et al. beschriebene Projekt [ZWRH06] verfolgt das Ziel, Quellcode mit der dazugehörigen Dokumentation umfassend zu verlinken, um den Reverse-Engineering Prozess zu unterstützen. Dazu werden sowohl die Dokumentation, als auch Programmcode auf Ontologien abgebildet und danach miteinander verknüpft. Für diese Arbeit von Interesse ist dabei die Struktur der Quellcode-Ontologie, die sich sehr stark an der konkreten Implementierung orientiert. Die Ontologie wurde entworfen, um die wichtigsten Konzepte der objektorientierten Programmiersprachen abbilden zu können. Für den Aufbau der Alice-Ontologie ist eine derartige Unterteilung allerdings zu feingranular, da sich das Projekt „Programmieren mit natürlicher Sprache“ auf die Verwendung bestehender Szenarien beschränkt und keine Implementierung oder Analyse von Software beabsichtigt.

Sabou stellt in [Sab04] eine semi-automatische Methode zur Extraktion einer Domänen-Ontologie aus der Dokumentation einer Software vor. Der Vorgang geschieht in 4 Schritten: Zuerst wird aus der Dokumentation ein Korpus aufgebaut, der keine syntaktischen Ausdrücke mehr enthält. Danach wird der Korpus in Tokens zerlegt und daraus Verb-Nomen-Paare gebildet. Im dritten Schritt müssen händisch relevante Paare identifiziert und ausgewählt werden, aus welchen abschließend eine Ontologie gebaut wird. Anschließend wird jeder Schritt der Extraktion statistisch evaluiert, um Probleme und Verbesserungsmöglichkeiten aufzudecken. Dafür definieren die Autoren Metriken für die Bewertung der Zwischenergebnisse nach jedem Schritt. Die Auswertung wird nach jeder Feinabstimmung des Vorgangs vorgenommen, um sinnvolle Verbesserungen zu identifizieren. Da die

beschriebene Extraktionsmethode für die Verarbeitung von Dokumentationen konzipiert wurde, eignet sie sich nicht für unsere Aufgabenstellung. Der Ansatz statistischer Evaluation der Ergebnisse kann aber, angepasst an unsere Anforderungen, anschauliche Metriken und Ergebnisse liefern.

Ratiu et al. beschreiben in [RFJ08] einen Ansatz zur Extraktion von Wissen aus mehreren, konkreten Implementierungen (oder APIs), die dieselbe Wissensdomäne adressieren. Diese Vorgehensweise resultiert aus der Beobachtung, dass relevante Konzepte die modelliert werden, sich in allen Implementierungen wieder finden lassen, wohingegen weniger relevante Informationen (Rauschen) in jeder konkreten Implementierung variieren. Zu diesem Zweck werden APIs in Graphen umgewandelt, welche anschließend effizient miteinander verglichen werden können, um ähnliche Strukturen zu identifizieren und als relevant einzustufen. Bei der Extraktion der Alice-API-Ontologie geht es in erster Linie darum, die bestehende Architektur des Rahmenwerks auf eine Ontologie abzubilden. Da es sich hierbei um die Extraktion einer konkreten Implementierung handelt und wir keine automatische Identifikation relevanter Konzepte anstreben, kann der vorgeschlagene Ansatz in dieser Arbeit nicht verwendet werden.

Da derzeit eine Wissensontologie nicht vollautomatisch erstellt werden kann, stellen Cimiano et al. in [Sta09] eine allgemeine Architektur für den halbautomatischen Aufbau von Ontologien vor. Sie unterstützt den Ontologie-Ingenieur in den Phasen der Einarbeitung in die Wissensdomäne, Vorbereitung der Daten, Modellierung, Evaluation und Verwendung des entstehenden Ontologie-Extraktors. Anschließend werden einige grundlegende Vorgehensweisen zum Aufbau einer Ontologie aus bestehenden Datenquellen aufgeführt und bereits existierende Rahmenwerke zum Aufbau von Ontologien aus textuellen Quellen beschrieben. Beim Aufbau der Alice-API-Ontologie mussten die genannten Phasen ebenfalls durchlaufen werden. Der Prozess der Datenaufbereitung in dieser Arbeit unterscheidet sich dabei am meisten von der Extraktion von Wissen aus Texten. Dies liegt daran, dass bei der Analyse eines API die benötigten Teile des Rahmenwerks programmatisch für die Extraktion vorbereitet werden müssen.

Simperl et al. schlagen in [ESV] eine ähnliche Methodik wie in [Sta09] vor, die sich zum Aufbau von Wissens-Ontologien eignet. Sie besteht aus acht Phasen: Ausführbarkeitsstudie; Anforderungsanalyse; Auswahl der Informationsquellen und Populationsmethoden und Werkzeugen; Vorbereitung und Ausführung der Population und abschließend die Bewertung und Integration der entstandenen Ontologie. So eine Vorgehensweise kann für den iterativen Aufbau der Ontologie verwendet werden. Simperl et al. beschreiben detailliert, welche Aktivitäten in jeder Phase durchgeführt, und welche Entscheidungen getroffen werden müssen. Die Autoren behaupten, dass ohne eine klare Methodologie die Aktivitäten der Ontologieingenieure nicht optimal durch existierende Extraktionstechniken unterstützt werden können.

Das Rahmenwerk *Text-To-Onto* [HV08] wurde für die Extraktion von Ontologien aus Texten konzipiert und verbindet Werkzeuge für deren Aufbau und Pflege. Hierzu bietet das Rahmenwerk eine große Auswahl von Algorithmen die sowohl auf statistischen als auch auf Datamining- und Clustering-Ansätzen basieren. Das Nachfolger-Werkzeug *Text2Onto* [CV05] bietet zudem eine abstrahierte Ontologie-Beschreibungssprache *Possible Ontologies Model* (POM), die sich anschließend in eine beliebige Beschreibungssprache (z.B. OWL oder RDFS) übersetzen lässt. Leider lassen sich die Algorithmen aus diesen Werkzeugen nicht für die Extraktion der Alice-Ontologie verwenden, da die Informationen in Alice nicht in Textform sondern als Skripten vorliegen. Stattdessen untersuchen wir in dieser Arbeit die Architektur des Rahmenwerks und entwickeln eigene Algorithmen, die speziell für die Extraktion der Alice-Skripte konzipiert sind.

4. Analyse und Entwurf

Wie bereits in Kapitel 1 motiviert, lassen sich Informationen über Objekte einerseits aus den Alice-Klassenbibliotheken, andererseits aus schon vorhandenen Alice-Welten extrahieren. Dazu analysieren wir in Kapitel 4.1 die Architektur des Alice-Rahmenwerks und beschreiben anschließend in Kapitel 4.2 die vorgeschlagene Struktur der Ontologie, auf die wir hinterher die extrahierten Konstrukte des Rahmenwerks und Objekte aus den existierenden Alice-Welten abbilden wollen.

In dieser Arbeit untersuchen wir die Version 2.0 des Alice-Rahmenwerks, die uns aktuell als Quellcode zur Verfügung steht.

4.1. Alice Architektur

Die Alice-Entwicklungsumgebung wurde in erster Linie dafür konzipiert, um es dem Benutzer zu erlauben, Verhalten von Objekten aus der realen Welt in einer virtuellen dreidimensionalen Szene zu modellieren und Videosequenzen zu generieren. Dabei sollen die Anwender des Rahmenwerks ihre ersten Erfahrungen mit dem Ansatz objektorientierter Programmierung machen.

Da die Zielgruppe der Anwender über wenig bis keine Programmierkenntnisse verfügt, ist die Alice-Skriptsprache stark an die Begriffe aus der realen Welt angenähert. So werden zum Beispiel die Klassennamen der Methoden (siehe Abschnitt 4.1.4) von dem Benutzer verborgen und auf der Benutzeroberfläche durch verständliche Begriffe aus dem Alltag ersetzt. Um den Programmierprozess für den Benutzer so einfach wie möglich zu gestalten, werden in dem Rahmenwerk für alle verwendeten Skriptbausteine eigene Konstrukte definiert. So werden z.B. Methoden und Kontrollstrukturen in Alice als eigene Klassen modelliert, die für Programmieranfänger verständlich aufgebaut sind.

Bei der Analyse der Softwarearchitektur des Rahmenwerks ist zu berücksichtigen, dass Java in vielen Fällen als Metasprache zur Definition der Alice-Skriptsprache verwendet wird. Im Folgenden geben wir aufgrund fehlender Dokumentation des Alice-Quellcodes eine Übersicht über relevante Entwurfsentscheidungen der Alice-Architektur.

4.1.1. Allgemeiner Aufbau

Das Alice-Rahmenwerk lässt sich in drei große Bausteine aufteilen: das Datenmodell, welches im Paket `edu.cmu.cs.stage3.alice.core` angesiedelt ist; die Benutzeroberfläche,

die sich im Paket `edu.cmu.cs.stage3.alice.authoringtool` befindet; und Szenegraphen, der Datenstrukturen für das eigentliche Rendering von Videosequenzen enthält und im Paket `edu.cmu.cs.stage3.alice.scenegraph` zu finden ist.

Für die Verwendung des Rahmenwerks zum Zweck der Programmierung mit natürlicher Sprache ist das Datenmodell von entscheidender Bedeutung. Dabei wird bei der automatischen Erstellung von Szenerien und Bewegungsabläufen aus natürlichsprachlichen Texten die im Laufe des Gesamtprojekts entstehende Übersetzungssoftware die Funktion der Benutzeroberfäche übernehmen.

Das Datenmodell, auf dem die Alice-Skriptsprache aufgebaut ist, lässt sich in drei Bereiche unterteilen:

1. 3D-Objekte: Modelle der realen Welt, die vom Benutzer des Rahmenwerks erstellt und zur Laufzeit der Videosequenz bewegt und verändert werden (Abschnitt 4.1.2);
2. Funktionen und Methoden: enthalten Berechnungen oder Anweisungen an die Objekte der Szenerie (Abschnitt 4.1.4);
3. Alice-Steuerkonstrukte: Klassen, die der Beschreibung des Kontrollflusses innerhalb der Methoden dienen (Abschnitt 4.1.6).

Die Klasse `Element`

`Element` ist die Elternklasse aller Modelle und Methoden, die ein Benutzer in Alice definieren und verwenden kann. Objekte der Klasse `Element` implementieren das Kompositionsmuster (siehe Abb. 4.1) und können somit beliebig geschachtelt werden. Dazu enthält `Element` das Feld *m_children*, in dem alle direkten Kind-Elemente des Objekts, als auch Methoden zum Hinzufügen, Finden und Entfernen von diesen gespeichert werden.

Das Feld *name* von `Element` ist ebenfalls relevant für den Aufbau der Alice-Welten. Es ist die Bezeichnung des Elements, die beim Erstellen des Elements vom Benutzer des Rahmenwerks angegeben wird. Die Auswertung dieses Felds ist für den Aufbau der Ontologie sehr wichtig, da der Name des Objekts in den meisten Fällen der Bezeichnung des Objekts in natürlicher Sprache entspricht.

In den folgenden Abschnitten beschreiben wir die Vererbungshierarchie der Alice-Elemente und gehen auf die Rollen der einzelnen Unterklassen genauer ein.

4.1.2. 3D-Objekte

Dreidimensionale Objekte haben eine zentrale Bedeutung im Alice-Rahmenwerk (siehe Kap.2.1.3). Auf Datenmodell-Ebene werden diese Objekte als Kompositionen aus weiteren Objekten oder primitiven geometrischen Figuren zusammengesetzt.

4.1.2.1. Die Klasse `Sandbox`

Die Klasse `Sandbox` definiert ein Grundgerüst aller dreidimensionalen Modelle, die in Alice zur Verfügung stehen. Zu den relevanten Feldern, die dabei in `Sandbox` deklariert werden, gehören:

- `questions` enthält alle benutzerdefinierten Funktionen eines Modells (siehe Kapitel4.1.4.3)
- `responses` enthält alle benutzerdefinierten Animationen eines Modells (siehe Kapitel 4.1.4.1)
- `behaviors` enthält die Ereignisse, die für ein Modell definiert wurden (siehe Kapitel 4.1.2.6)

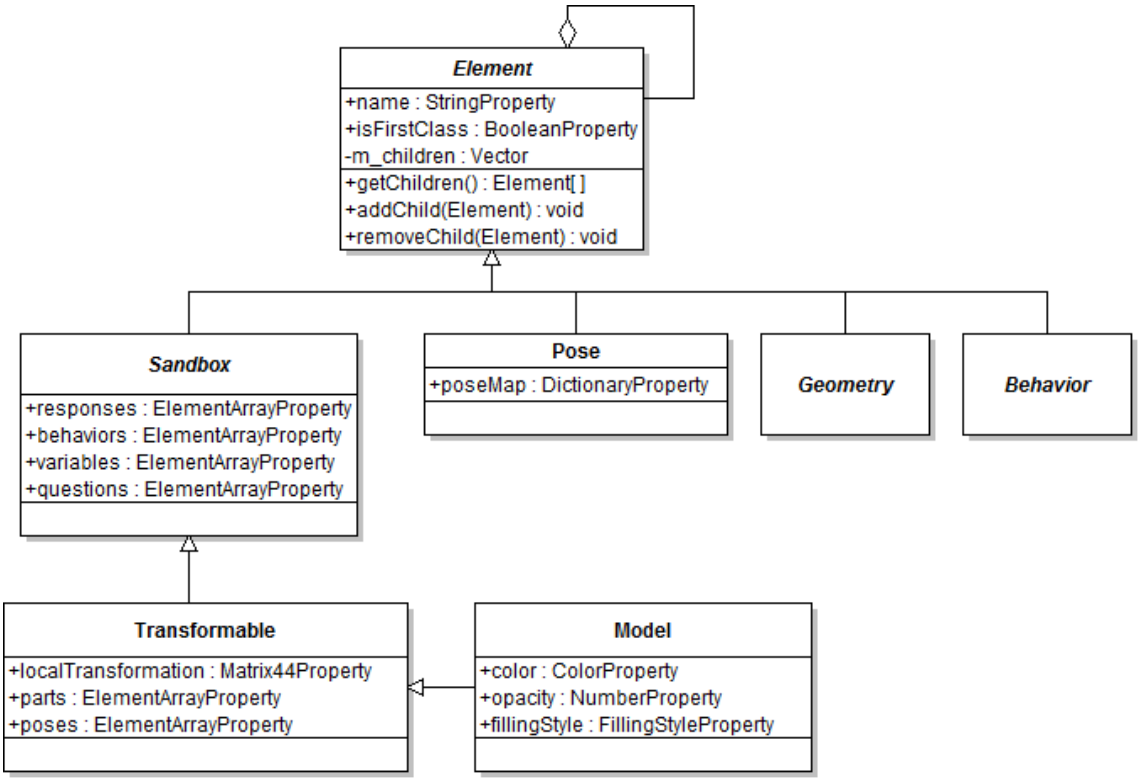


Abbildung 4.1.: Das Element-Kompositum und seine Unterklassen

– `variables`: Der Benutzer kann in jedem 3D-Objekt eigene Parameter (sog. Variablen) definieren, die z.B. einen internen Zustand des Objekts enthalten können. Sie werden in dem Feld `variables` gespeichert

Des Weiteren sind in Sandbox die Felder `textureMaps`, `sounds` und `misc` deklariert, welche Texturen, Tonspuren und die Parameter enthalten, welche in der Alice-Benutzeroberfläche unter dem Punkt *Misc* unter dem Reiter *Properties* (siehe Abb. 2.1, (2)) aufgeführt werden.

4.1.2.2. Die Klasse `Transformable`

Abgeleitet von der `Sandbox`-Klasse erweitert `Transformable` ein Element um die Eigenschaften, welche es ermöglichen, das Modell in der 3D-Szene zu positionieren und frei zu transformieren. Das Feld `localTransformation` beschreibt dabei die Transformationsmatrix von dem globalen zum lokalen Koordinatensystem und damit implizit die Position des Modells in der Szenerie.

`Transformable` implementiert folgende grundlegenden Objekttransformationen, auf die der Endnutzer des Alice-Rahmenwerks allerdings nicht direkt zugreifen kann. Sie bilden vielmehr das Herzstück der vordefinierten Alice-Animationen:

– *moveRightNow()*: löst eine Translation des Objekts im Raum aus, die alle Punkte des Objekts um einen konstanten Vektor verschiebt;

– *resizeRightNow()*: führt eine uniforme Skalierung des Modells aus;

– *rotateRightNow()*: rotiert das Objekt um eine Achse und einen vorgegebenen Winkel;

– *setOrientationRightNow()*: transformiert die Achsen des Objekts entsprechend der vorgegebenen 3x3-Matrix.

Darauf aufbauend bietet `Transformable` noch weitere zusammengesetzte Funktionen an, die auf die eben aufgelisteten Methoden zurückgreifen, hier jedoch nicht weiter betrachtet werden sollen.

Des Weiteren sind hier die Methoden zur Bestimmung der positionsbezogenen Angaben des Objekts definiert. Dazu gehören zum Beispiel *getLocalTransformation()* zum Abfragen der aktuellen Position des Objekts im Raum, aber auch Funktionen mit booleschen Rückgabewerten *isRightOf(·)*, *isLeftOf(·)*, *isAbove(·)* und *isBelow(·)*.

Ein Objekt der Klasse `Transformable` enthält zusätzlich Referenzen zu weiteren Eigenschaften der Modelle:

- alle Modelle in Alice besitzen Posen, welche das Modell einnehmen kann. Kind-Elemente eines Modells vom Typ `Pose` werden in dem Feld `poses` hinterlegt (siehe Abschnitt 4.1.2.4);
- die direkten Kind-Knoten eines Modell-Objekts vom Typ `Transformable` werden getrennt von den anderen Kind-Elementen in dem Feld `parts` abgespeichert.

4.1.2.3. Die Klasse `Model`

Als Unterklasse von `Transformable` erweitert `Model` die Objekte der Szenerie um Anzeigeeigenschaften. Dazu gehören Farbe (*color*), Textur (*emissiveColorMap*), Füllstil der Oberfläche (*fillingStyle*), Opazität (*opacity*) und weitere Parameter, die für die Berechnung des Beleuchtungsmodells benötigt werden (ambiente, diffuse und spiegelnde Komponenten der Beleuchtungsberechnung).

4.1.2.4. Die Klasse `Pose`

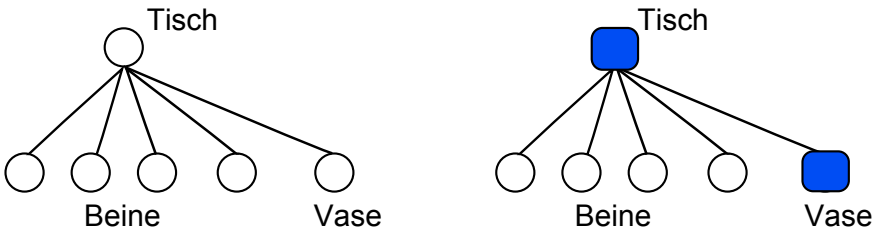
Wie bereits erwähnt, können für Modelle in Alice Posen definiert werden, die sie einnehmen können. Dazu verwendet das Rahmenwerk Objekte der Klasse `Pose`, welche von `Element` abgeleitet wird. Die Datenstruktur beschränkt sich auf das s.g. `poseMap` Feld, in dem alle nötigen Informationen zur Position aller Bestandteile des Modells abgelegt sind.

Die Klasse liefert auch die Methode *manufacturePose()*, die aufgerufen werden kann, um die aktuelle Körperhaltung eines Modells abzuspeichern. Der Benutzer des Alice-Entwicklungswerkzeugs kann eine Pose jederzeit mit einem Rechtsklick auf das Objekt in der Szenerie und der Auswahl von *„capture pose“* aus dem Kontextmenü aufnehmen (siehe Abb. 2.1, Bereich (4)).

4.1.2.5. Geometrische Primitive

Für die Darstellung der Alice-Modelle werden geometrische Primitive verwendet, die in dem Paket `alice.core.geometry` zu finden sind. Die Primitive werden im Alice-Objektbaum als Blätter gespeichert, da sie keine Kind-Knoten enthalten können.

Da sich alle Objekte in der Computergrafik durch Dreiecke im Raum beschreiben lassen, sind Objekte der Klasse `IndexedTriangleArray` (Array von Dreiecken) die meistverwendeten Primitive in den Alice-Welten. Zusätzlich ist in dem Paket die Typen `PolygonSegment` und `Polygon` definiert, die zu `PolygonGroup`s zusammengefasst werden können. Als zusätzliches 3D-Objekt haben die Entwickler von Alice eine Klasse `Text3D` erstellt, die in Alice ebenfalls zu geometrischen Primitiven gezählt wird und einen dreidimensionalen Schriftzug darstellt, der in der Szene platziert werden kann.



(a) gleichrangige Modelle (b) mit First-Class Modellen

Abbildung 4.2.: Objektstruktur der Alice-Modelle

4.1.2.6. Ereignisse

Wie bereits im Abschnitt 2.1.4 beschrieben, lassen sich in Alice Ereignisse definieren, die ein gewünschtes Verhalten des Objekts zur Laufzeit der Videosequenz auslösen. Für diese Funktionalität ist im Rahmenwerk die Klasse `Behavior` verantwortlich. Alle verfügbaren Ereignisse befinden sich in dem Paket `edu.cmu.cs.stage3.alice.core.behavior`.

Instanzen der Klasse `Behavior` werden in dem Objekt gespeichert, in dem das Ereignis definiert wurde. Standardmäßig werden alle Ereignisse im Welt-Objekt definiert, wenn man sie durch Klicken auf den Button *'create new event'* erstellt. Erzeugt man jedoch ein Ereignis durch das Ziehen eines Objekts in den Ereigniseditor, wird es in diesem Objekt angelegt.

Im Folgenden ist eine Liste der im Ereigniseditor verfügbaren Ereignisse mit einer kurzen Beschreibung aufgeführt.

- `WorldStartBehavior` wird beim Start der Videosequenz ausgeführt,
- `KeyClickBehavior` wird beim Drücken auf eine ausgewählte Taste auf der Tastatur ausgeführt,
- `MouseButtonClickBehavior` wird beim Klicken auf ein Objekt der Szenerie ausgeführt,
- `ConditionalBehavior` wird jedes Mal ausgeführt, wenn die definierte Bedingung wahr ist,
- `VariableChangeBehavior` wird beim Ändern einer überwachten Variable ausgeführt,
- `DefaultMouseInteractionBehavior` erlaubt es, Objekte mit der Maus zu steuern,
- `KeyboardNavigationBehavior` erlaubt es, Objekte mit den Pfeiltasten zu steuern,
- `MouseNavigationBehavior` erlaubt es, die Kamera mit den Pfeiltasten zu steuern,
- `MouseLookingBehavior` erlaubt es, die Kamera mit der Maus in alle Richtungen zu drehen.

4.1.2.7. Objekte erster Klasse

Die folgenden Ausführungen lehnen sich an den Bericht von Conway et al. ([CAB+00]) an.

Wie schon erwähnt, sind alle Objekte in Alice in einer Baum-Struktur aus Knoten und deren Kind-Knoten organisiert. Nehmen wir als Beispiel eine Vase, die auf einem Tisch steht. Wenn wir wollten, dass sich die Vase zusammen mit dem Tisch bewegt, würden wir sie als ein Kind-Objekt des Tisches modellieren (Abb. 4.2(a)).

Allerdings geht dabei Information darüber verloren, welche Elemente im Objektbaum selbständige Modelle und welche nur Bestandteile der Eltern-Knoten sind. So kann das Alice-Rahmenwerk in dem oberen Beispiel nicht mehr unterscheiden, ob die Vase ein Bestandteil vom Tisch ist, oder ein eigenständiges Objekt. Wie könnte man in diesem Fall sicherstellen, dass die Methode

`Tisch`.*setColor(***Red***)*

nur der Tisch, aber nicht die Vase rot färbt? Zu diesem Zweck werden in Alice eigenständige Objekte als Objekte erster Klasse (*first-class*-Objekte) markiert (zu sehen als blau eingefärbte Knoten in Abb. 4.2(b)). Elemente innerhalb dieser Objekte werden als Bestandteile (`parts`) deklariert. Dies erlaubt es, beim Ausführen von ausgewählten Operationen zwischen echten Bestandteilen eines Objekts und anderen eigenständigen Modellen zu unterscheiden, die aus anderen Gründen als Kind-Knoten in die Struktur eingefügt werden mussten.

In dieser Arbeit nehmen wir von der First-Class-Eigenschaft Gebrauch und stufen ein Element anhand von dem Feld `isFirstClass` als eigenständiges Objekt ein.

4.1.3. Vordefinierte Objekte

Alice verfügt über eine Reihe von vordefinierten Objekten, die von der Klasse `Model` abgeleitet sind und sich folglich in der Szenerie genauso verhalten, wie die anderen Modelle. Bei diesen Konstrukten handelt es sich jedoch um spezifische Objekte, ohne die die Bildsynthese von Videosequenzen nicht auskommen könnte. In dieser Arbeit sind das Licht- und Kameramodell von Interesse.

4.1.3.1. Die Klasse `Light`

Die abstrakte Klasse `Light` kapselt die Eigenschaften der Lichtquellen, die in der Szenerie positioniert werden können. Möchte man in den Skripten zusätzliche Lichtquellen modellieren, kommt man ohne diese Klasse nicht aus. Eine Lichtquelle kann als Objekt folgender Klassen instanziiert werden:

- `AmbientLight` beschreibt eine globale Umgebungsbeleuchtung, die keine gerichteten Lichtstrahlen aussendet, sondern die Objekte von allen Seiten gleichmäßig beleuchtet. Dieses Objekt wird standardmäßig beim Erstellen einer Alice-Welt angelegt;
- `DirectionalLight` sendet parallele Lichtstrahlen in eine bestimmte Richtung;
- `PointLight` modelliert eine Lichtquelle, die aus einem Punkt im Raum Lichtstrahlen in alle Richtungen aussendet und
- `SpotLight` kann als eine Punktlichtquelle betrachtet werden, die Licht nur in einen kegelförmigen Bereich aussendet.

4.1.3.2. Die Klasse `Camera`

Für die Modellierung dynamischer Szenen reicht eine fest positionierte virtuelle Kamera nicht aus. Für die Bewegung der Kamera bietet Alice alle dazu benötigten Eigenschaften des Kameraobjekts.

Abgeleitet von `Model`, erbt eine abstrakte `Camera` dessen Eigenschaften und Methoden. Deshalb kann die Kamera grundsätzlich genauso wie die anderen Objekte der Szenerie im Raum transformiert werden. Zusätzlich enthält `Camera` die Standard-Eigenschaften, welche

von einer Kamera für das Berechnen von Videosequenzen (Rendern) benötigt werden, an dieser Stelle aber von zweitrangiger Bedeutung sind.

Als konkrete Implementierungen werden in Alice alle gängigen Kamera-Modelle angeboten: `OrthographicCamera`, `PerspectiveCamera`, `ProjectionCamera` und `SymmetricPerspectiveCamera`. Standardmäßig wird die Kamera mit der symmetrischen, perspektivischen Projektion verwendet, da dieses Modell der menschlichen Wahrnehmung am nächsten kommt.

4.1.4. Funktionen und Methoden

Wie bereits im Abschnitt 2.1.3.2 beschrieben, werden die Methoden in Alice aus Benutzersicht in zwei Kategorien unterteilt: die Methoden (*methods*) und Funktionen (*functions*). Auf Ebene der Architektur werden Aufrufe von Methoden durch Objekte vom Typ `Animation`, und Funktionen durch Unterklassen von `Question` beschrieben.

4.1.4.1. Animationen

Animationen sind Befehle an Objekte, die im Laufe der Videosequenz sichtbar ausgeführt werden. Sie sind für den Benutzer unter dem Reiter *„methods“* aufgeführt und enthalten sowohl vordefinierte als auch vom Benutzer erstellte Methoden. Die Animationen findet man in dem Paket `edu.cmu.cs.stage3.alice.core.response`.

Für den Benutzer des Alice-Rahmenwerks bleibt der Begriff *Animation* verborgen. Es fasst die Animationen als Methoden der Objekte auf. Dazu wird in Alice eine Verknüpfung von den Animationsklassen auf den in der Benutzeroberfläche angezeigten Namen der Methode definiert. So wird bspw. `PointOfViewAnimation` zu *‘set point of view to’* übersetzt. Den angezeigten Namen für eine Animationsklasse liefert die Methode *getFormat()* der Klasse `AuthoringToolResources`.

Das Alice-Rahmenwerk verfolgt die Strategie, für alle ausgeführten Befehle eine Animation anzubieten [CAB+00]. Im Umkehrschluss bedeutet das, dass der Alice-Nutzer nur Funktionen ausführen kann, die auch als Animation modelliert sind. Insbesondere werden in der Alice-Skriptsprache alle Objekttransformationen mittels Animationen durchgeführt. Dies bedeutet, dass auch beim Aufbau der Skripten mit der Software, die im Laufe des Gesamtprojekts entwickelt wird, alle Objekttransformationen durch Subklassen von `Animation` realisiert werden müssen.

Da der Großteil der Animationen räumliche Transformationen beschreibt, bilden den Kern von Animationen auf Datenmodellebene zum einen die Eigenschaften der in Kapitel 4.1.2.2 beschriebenen Klasse `Transformable`. Tabelle C.3 zeigt eine Übersicht über Standardanimationen in Alice, begleitet von dem Namen in der Benutzeroberfläche und einer kurzen Funktionsbeschreibung.

Zum anderen werden alle Änderungen an den Objekteigenschaften ebenfalls animiert. Diese Funktionalität wird in der Klasse `PropertyAnimation` angeboten, welche einen weichen Übergang des Ursprungswerts zu dem Zielwert über eine Zeitspanne von ca. 1 Sekunde darstellt. Grundsätzlich können alle Objekteigenschaften in Alice geändert werden. Explizit werden im Kontextmenü der Benutzeroberfläche jedoch nur Anpassungen von Farbe (*color*), Opazität (*opacity*), Textur (*diffuseColorMap*) und Füllstil der Objektoberfläche (*fillingStyle*) angeboten.

Außerdem verfügt Alice über eine Reihe besonderer Animationen. Dazu gehören z.B.:

– die Methode `GetAGoodLookAtAnimation`, welche die Kamera automatisch zu einem Modell bewegt und es komplett anzeigt.

– die Klasse `PlaceAnimation`, die nicht in einer Videosequenz verwendet werden kann, sondern vom Rahmenwerk aufgerufen wird, um das Einfügen eines Objekts in die Szenerie darzustellen.

4.1.4.2. Die Klasse `Expression`

Die abstrakte Klasse `Expression` fasst im Rahmenwerk die Eigenschaften aller Datentypen mit einem (Rückgabe-) Wert zusammen. Zu ihren Unterklassen gehören alle Funktionen und die Variablen (Klasse `Variable`).

In dieser Arbeit sind unter den Eigenschaften von `Expression` die beiden Methoden *getValue()* und *getValueClass()* von Interesse. Da es sich bei den Variablen und Funktionen im Alice-Rahmenwerk um Objekte handelt, wird auch der Wert einer Alice-Variable (bzw. der Rückgabewert einer Funktion) durch *getValue()* erfragt, und deren Typ durch *getValueClass()*.

Alle Objekte und Funktionen können im Rahmenwerk mit Variablen versehen werden. Bei den Objekten dient zu diesem Zweck das von der Klasse `Sandbox` geerbte Feld *variables*; bei den Funktionen ist für diese Eigenschaft das Feld *localVariables* zuständig.

4.1.4.3. Funktionen

Funktionen (*Questions*) sind alle Funktionen der Modelle, die einen Rückgabewert haben. Sie sind unter dem Reiter *functions* (siehe Abb. 2.1, (2)) aufgelistet und können ebenfalls um benutzerdefinierte Funktionen erweitert werden. Alle Funktionen sind im Paket `edu.cmu.cs.stage3.alice.core.question` angesiedelt und werden von der Klasse `Expression` abgeleitet.

Die Funktionen in Alice kann man in objektbezogene und statische aufteilen. Während die ersteren zwangsläufig auf einem Objekt ausgeführt werden und für den Benutzer als Methoden der Szenerieobjekte dargestellt werden, operieren die statischen Funktionen auf primitiven Datentypen und sind an jeder Stelle des Skripts verfügbar.

Wenn der Benutzer im Objektbaum ein Modell (ausgenommen das Welt-Objekt) auswählt, werden unter dem Reiter *functions* die objektbezogenen Funktionen aufgelistet. Sie liefern sowohl Informationen zur Lage, Position und Distanz zu anderen Objekten, als auch zu weiteren Eigenschaften der Modelle. In der Tabelle A.1 sind alle objektbezogenen Funktionen aufgeführt.

Die statischen Funktionen werden bei der Auswahl des Welt-Objektes aufgelistet. Sie enthalten sowohl boolesche und mathematische, als auch Zeichenketten-, Zufallswert-, Zeitfunktionen und Funktionen zur Eingabeaufforderung. Eine Übersicht über alle statischen Funktionen ist in der Tabelle B.2 gegeben.

4.1.4.4. Parameter

Die Methoden in Alice können mit Parametern versehen werden. Die Parameter sind Elemente vom Typ `Variable` (siehe Abschnitt 4.1.4.2). Wie auch andere Komponenten der Alice-Skripte werden Parameter als Kind-Knoten der Methoden abgespeichert.

4.1.5. Benutzerdefinierte Funktionen

Analog zu den vordefinierten Methoden können Anwender des Rahmenwerks eigene Animationen und Funktionen definieren. Da beide Typen (wie auch alle anderen beschriebenen Klassen) von `Element` erben, erhalten sie unter anderem auch das Feld *name*, welches in dem graphischen Rahmenwerk obligatorisch für die vom Benutzer erstellten Funktionen ist und eine entscheidende Rolle im Aufbau der Alice-API-Ontologie spielt. Bei der Population der Ontologie hoffen wir darauf, dass die Benutzer ihre Funktionen möglichst treffend benennen, damit sie später im Gesamtprojekt sinnvoll verwendet werden können.

4.1.5.1. Animationen

Die Klasse `UserDefinedResponse` stellt benutzerdefinierte Animationsanweisungen (siehe Abschnitt 2.1.3.2) dar. Im eigentlichen Sinne handelt sich bei diesem Konstrukt um eine von `DoInOrder` (siehe Abschnitt 4.1.6) abgeleitete Klasse, die mehrere beliebig geschachtelte Anweisungen nacheinander ausführen kann. Zusätzlich kann eine benutzerdefinierte Animation den eigenen Zustand in Form von Variablen in dem Feld *localVariables* speichern.

4.1.5.2. Funktionen

Analog zu Animationen können Anwender im Alice Rahmenwerk eigene Funktionen mit einem Rückgabewert anlegen. Dazu ist in Alice die Klasse `UserDefinedQuestion` implementiert, die alle Eigenschaften von benutzerdefinierten Funktionen kapselt. `UserDefinedQuestion` ist von der Klasse `Expression` abgeleitet (siehe Abschnitt 4.1.4.2) und erhält demnach ebenfalls die Methoden *getValue()* und *getValueClass()*, die den Rückgabewert der Funktion beschreiben.

In einer Funktion können keine Animationsanweisungen an Objekte stehen. Es kann nur auf Skript-Kontrollstrukturen (siehe Abschnitt 4.1.6) und bereits vorhandene Funktionen (vergleiche Abschnitt 4.1.4.3) zurückgegriffen werden. Damit wird im Rahmenwerk auch in den benutzerdefinierten Methoden eine strikte Trennung zwischen Animationen und Funktionen umgesetzt.

4.1.6. Steuerkonstrukte in Alice

Die Steuerungskonstrukte befinden sich, genauso wie die Animationen, im Paket `edu.cmu.cs.stage3.alice.core.response`. Sie werden von der Klasse `CompositeResponse` abgeleitet, welche im Feld `componentResponses` alle im Anweisungsblock enthaltenen Methodenaufrufe verwalten kann.

Die Klasse `IfElse` bildet das Gegenstück zur bedingten Verzweigung in Programmiersprachen.

Die Klassen `DoInOrder` und `DoTogether` führen die enthaltenen Methodenaufrufe nacheinander bzw. parallel aus.

Alice bietet auch mehrere Arten von Schleifen an. Zu den einfachsten gehören die *'for'*-Schleife (`LoopNInOrder`) und die *'while'*-Schleife (`While`). Ebenso bietet das Rahmenwerk die Möglichkeit, über Listen zu iterieren, und dabei Methodenaufrufe für jedes der in der Liste enthaltenen Objekte auszuführen. Zu diesem Zweck sind die Klassen `ForEachInOrder` für sequenzielle, und `ForEachTogether` für parallele Ausführung konzipiert.

4.1.7. Klasse AuthoringToolResources

Die Klasse `AuthoringToolResources` ist für diese Arbeit von besonderem Interesse, obwohl sie nicht zum Datenmodell des Rahmenwerks gehört, sondern Informationen für die Darstellung in der Benutzeroberfläche liefert. Sie übernimmt die oben erwähnte Abbildung von den Namen der Klassen aus dem Paket `alice.core` auf ihre vordefinierten Bezeichnungen in der Benutzeroberfläche. Zu diesem Zweck wird während der Initialisierung die Dateien „StandardResources.py“ und „Alice Style.py“ aus dem Ordner „Required/resources“ eingelesen und in dem Feld *resources* abgelegt.

Mit den Informationen aus `AuthoringToolResources` lassen sich die extrahierten Konstrukte der Alice-Skriptsprache mit Begriffen anreichern, die die Funktion der Konstrukte in einer für Menschen verständlichen Sprache beschreiben. In den Folgearbeiten können diese Informationen für die Suche nach weiteren Synonymen verwendet werden.

So gibt beispielsweise die Methode *getFormat(*`PointOfViewAnimation.class`*)* die Zeichenkette „<*subject*> *set point of view to* <*asSeenBy*>“ zurück.

Mit *getDesiredProperties(*`Class`*)* bekommt man die Parameter, die beim Aufruf einer Methode benötigt werden.

Zusätzlich verfügt `AuthoringToolResources` über die beiden Methoden *getOneShotStructure(*`Element`*)*, die alle für ein Element verfügbaren Animationen, und *getQuestionStructure(*`Element`*)* die alle Funktionen eines Elements zurückgeben.

4.1.8. Speichern von Alice-Welten

Videosequenzen und Programme (im Folgenden ”Alice-Welten”), die während der Arbeit mit dem Alice-Rahmenwerk entstehen, können in Form von Dateien abgespeichert werden.

Die Weltdateien enthalten die komplette Beschreibung einer Szenerie inklusive der vom Benutzer definierten Methoden. Dies wird durch das Serialisieren aller Elemente des Skriptes erreicht, welche anschließend in einer hierarchischen Verzeichnisstruktur gespeichert werden. Die Dateistruktur spiegelt dabei die baumartige Anordnung der aufgebauten Element-Komposition genau wider. Der so entstandene Verzeichnisbaum wird danach komprimiert und als eine `*.a2w`-Datei gespeichert.

Für die automatische Generierung von Alice-Skripten aus natürlicher Sprache wird es in manchen Fällen nötig sein, Objekten zugehörige Dateien (z.B. Texturen, Audio-Dateien etc.) in die generierte Alice-Welt zu kopieren. Im Folgenden soll ein kleiner Einblick in die Struktur der gespeicherten Dateien gewährt werden.

Dateihierarchie

Da alle Entitäten der Alice-Skriptsprache von der Klasse `Element` abgeleitet sind, welche das Kompositum-Muster implementiert, wird jede davon als Knoten des Objektbaums gespeichert. In der serialisierten Darstellung entspricht jeder Knoten einer xml-Datei namens `elementData.xml`.

Im Listing 4.1 ist ein Auszug aus einer Alice-Dateihierarchie dargestellt, die eine Welt mit einem Eiskunstläufer beschreibt. Jeder (Unter-) Ordner enthält genau eine `elementData.xml`-Datei, die einen Knoten aus der Alice-Welt darstellt. Darin sind einerseits alle Eigenschaften eines Objekts gespeichert, andererseits seine Kind-Knoten referenziert. Die Kind-Knoten befinden sich jeweils in einem Unterordner ihres Elternknotens, der den Namen des Kind-Elements trägt.

Im Beispiel enthält der Einskunstläufer drei Kindelemente *Koerper*, *springen*, und *pose1*, die seine Struktur, eine Methode und eine Pose enthalten. Diese Elemente sind in der `Eiskunstlaeufer/elementData.xml` referenziert. Seinerseits besteht *Koerper* aus zwei unbenannten Objekten, die beide neben einer xml-Beschreibung noch kompilierte Dateien ‘indices.bin’ und ‘vertices.bin’ mit der Geometriebeschreibung des Körpers enthalten.

Die Methoden *springen* und *beginne Schau* sind ebenfalls hierarchisch aufgebaut und enthalten lauter unbenannte Skriptbausteine, welche Kontrollstrukturen und Methodenaufrufe enthalten können.

Die Posenbeschreibung *pose1* besteht aus einer einzigen xml-Datei, in der die Lage des Objekts und aller seiner Bestandteile gespeichert sind.

Die Einstellungen der beiden vorgeschriebenen Objekte `Kamera` und `Licht` sind ebenfalls in der *.a2w-Datei gespeichert und parametrisieren die vordefinierten Alice-Objekte (siehe 4.1.3) auf die Verwendung in der konkreten Szenerie.

```
|   elementData.xml
+---Camera
|       elementData.xml
+---Eiskunstlaeufer
|   |   elementData.xml
|   +---Koerper
|   |   |   elementData.xml
|   |   +---__Unnamed0__
|   |   |       elementData.xml
|   |   |       indices.bin
|   |   |       vertices.bin
|   |   \---__Unnamed1__
|   |           elementData.xml
|   |           indices.bin
|   |           vertices.bin
|   +---springen
|   |   |   elementData.xml
|   |   +---__Unnamed0__
|   |   |   |   elementData.xml
|   |   |   +---__Unnamed0__
|   |   |   |       elementData.xml
|   |   |   \---__Unnamed1__
|   |   |           elementData.xml
|   |   \---__Unnamed1__
|   |           elementData.xml
|   \---pose1
|           elementData.xml
+---Light
|       elementData.xml
\---beginne Schau
    |   elementData.xml
    \---__Unnamed0__
        |   elementData.xml
        +---__Unnamed0__
        |       elementData.xml
        \---__Unnamed1__
                elementData.xml
```

Listing 4.1: Aufbau einer Alice-Weltdatei

4.2. Die Alice-Ontologie

Um die Konzepte aus dem Alice-Rahmenwerk abzubilden, bauen wir zunächst eine Grundstruktur der Ontologie auf. Dabei müssen die Klassen aus dem Alice-Rahmenwerk sinnvoll in entsprechenden Entitäten der Ontologie widergespiegelt werden. Hierzu erstellen wir in der Entwurfsphase ein Grundgerüst der Ontologie, das später von dem Alice-API-Extraktor automatisch befüllt wird.

4.2.1. Taxonomie des Alice-API

Wie bereits erwähnt, muss die im Laufe dieser Bachelorarbeit erstellte Klassenhierarchie der Ontologie eindeutig die relevanten Bereiche der Architektur des Alice-Rahmenwerks beschreiben. Zu diesem Zweck betrachten wir nur diejenige Teilmenge der Alice-Klassen, die im Gesamtprojekt zum Aufbau der Alice-Skripte verwendet wird.

Da das Alice-Rahmenwerk in Java implementiert ist, verzichten wir in der Modellierung der Alice-Ontologie auf die Möglichkeit der Mehrfachvererbung der beteiligten Klassen. Die Abbildung 4.3 zeigt die Taxonomie bestehend aus den Konzepten des Rahmenwerks. Als Oberbegriff aller Alice-Konzepte definieren wir die Klasse `AliceThing` und grenzen damit die Wissensdomäne um Alice von allen anderen ab.

Im Folgenden betrachten wir die einzelnen Klassen der Ontologie.

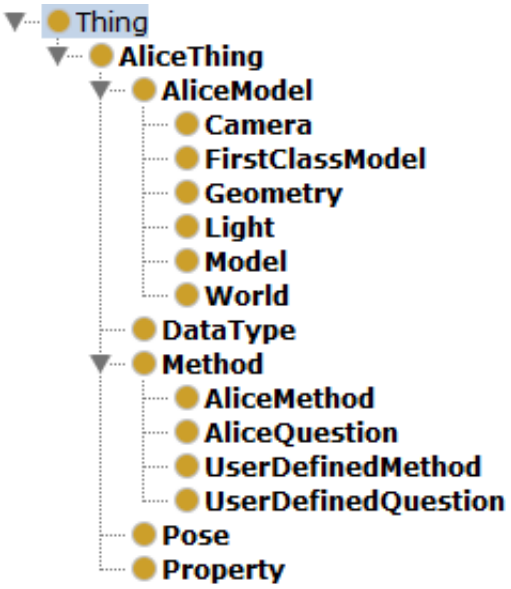


Abbildung 4.3.: Die Taxonomie der Alice-Konzepte

`AliceModel`

Als Gegenstück zu den Modellen in den Alice-Welten vereinigt die Klasse `AliceModel` sowohl in der Szenerie sichtbare Objekte (siehe Abschnitt 4.1.2) als auch vom Rahmenwerk vordefinierte Objekte (vergleiche Abschnitt 4.1.3):

- `World` enthält alle Szenerien, die extrahiert in der Ontologie enthalten sind;
- `FirstClassModel` enthält alle eigenständigen Objekte (siehe Abschnitt 4.1.2.7) der Szenerien;
- `Model` umfasst alle übrigen 3D-Objekte, aus welchen Objekte erster Klasse zusammengesetzt sind;

- `Geometry` enthält die geometrischen Primitive der Szenerien (siehe Abschnitt 4.1.2.5);
- `Camera` enthält die verwendeten Kamera-Objekte (vergleiche Abschnitt 4.1.3.1)
- und `Light` umfasst alle in den Alice-Welten verwendeten Lichttypen (siehe Abschnitt 4.1.3.1).

`Pose`

Die in den Alice-Objekten definierten Posen (siehe Abschnitt 4.1.2.4) bildet der Extraktor auf die Instanzen der Klasse `Pose` ab.

`Method`

In dem Konzept Method fassen wir alle Methoden und Funktionen (siehe Abschnitt 4.1.4) zusammen, die in den Welten verfügbar sind. Dazu unterscheiden wir genauso wie im Rahmenwerk zwischen vordefinierten und benutzerdefinierten Funktionen und Methoden. Wir führen folgende Klassen ein:

- `AliceMethod`, die alle im Rahmenwerk definierten Methoden als Instanzen enthält;
- `AliceQuestion` für die im Rahmenwerk verfügbaren Funktionen;
- `UserDefinedMethod`, auf die wir die benutzerdefinierten Methoden abbilden und
- `UserDefinedQuestion`, in der wir die in den Welten gefundenen Funktionen speichern.

`Property`

Die Übergabeparameter für die Methoden und Parameter der Alice-Objekte werden als Instanzen der Klasse `Property` verwaltet. Dabei tragen die Instanzen die Namen der zugehörigen Parameter.

`DataType`

Um die Typsicherheit im Aufbau der Alice-Welten im Gesamtprojekt zu gewährleisten, führen wir die Klasse `DataType` ein, deren Individuen die Datentypen repräsentieren.

4.2.2. Relationen

Auf der Taxonomie, die aus den oben beschriebenen Klassen besteht, definieren wir Relationen, mit welchen die Verknüpfung extrahierter Konzepte des Rahmenwerks untereinander ermöglicht wird.

Die Relation *consistsOf* verknüpft `AliceModel`-Instanzen untereinander. Mit dieser Verknüpfung definieren wir die Teil-Ganzes-Beziehung unter den Modellen und können damit deren Aufbau auf eine Baumstruktur abbilden (vergleiche Kapitel 2.1.3). Beispielsweise können wir damit im Kontext eines Alice-Objekts „Eiskunstläufer“ (Instanz von `FirstClassModel`) seinen „Oberkörper“ als Bestandteil abbilden, der seinerseits mit den *consistsOf*-Relationen auf die Instanzen „Rumpf“, „LinkeHand“ und „RechteHand“ versehen ist.

Mit *hasPose* erlauben wir die Verknüpfung der `AliceModel`-Instanzen mit den der Klasse *Pose*. In unserem Beispiel könnten wir „Eiskunstläufer“ mittels *hasPose* mit der Figur `Spagat` verknüpfen.

Alice-Objekte können Parameter enthalten, welche den inneren Zustand des Objekts beschreiben. Die möglichen Parameter eines Objekts verbindet der Extraktor mittels der

Relation *hasObjectProperty* zu der Instanz des Objekts, welchem die Eigenschaft zugeordnet ist. Zielbereich der Relation sind Instanzen der Klasse `Property`.

Die Relation *fromWorld* ermöglicht es, beliebige Objekte der Ontologie mit einer Welt zu verknüpfen. Zum Beispiel würden wir alle 3D-Objekte der Alice-Welt „IceWorld.a2w“ mit der gleichnamigen Instanz der Klasse `World` verbinden.

Da die Namen der Individuen während der Extraktion normalisiert werden, erhalten alle Elemente aus den Alice-Welten über die Relation *elementPath* ihren vom Wurzel-Knoten ausgehenden relativen Pfad innerhalb des Objektbaums. Dieser Pfad wird es ermöglichen, von den Instanzen der Alice-API-Ontologie auf konkrete Elemente der Alice-Welten zurückzuschließen.

Außerdem verknüpft der Extraktor die Alice-Objekte mit ihren Funktionen und Methoden mittels der Relation *hasMethod*. Dazu gehören sowohl benutzerdefinierte als auch im Rahmenwerk verfügbare Funktionen. Für den Eiskunstläufer wären zum Beispiel die Methoden *springen* und *fahren*, sowie die Standardmethoden *move*, *turn* usw. verfügbar (siehe Tabelle C.3 im Anhang).

Ihrerseits kann die Signatur einer Methode Übergabeparameter enthalten. Diese Beziehung modellieren wir in der Ontologie durch die Relation *hasMethodProperty*, welche Instanzen der Klasse `Method` mit denen der Klasse `Property` verknüpft. So verfügt zum Beispiel die Methode *fahren* über den Parameter *distance*, der angibt wie weit der Eiskunstläufer fahren soll. Deshalb verbindet der Extraktor die Methodeninstanz über die Relation *hasMethodProperty* mit der Instanz namens *distance*.

Alle Objekte der Ontologie können über die Relation *ofType* mit den Individuen des Konzepts `DataType` verknüpft werden. Für Alice-Objekte und Parameter würde diese Beziehung die Information ihren Datentyp festlegen; bei Funktionen kennzeichnet diese Beziehung den Typ des Rückgabewerts.

4.2.3. Individuen

Als Individuen werden in der Ontologie nur extrahierte Komponenten des Alice-Rahmenwerks gespeichert. Damit ist sichergestellt, dass während der Benutzung der fertigen Ontologie immer klar ist, auf welche Objekte und deren Eigenschaften in den Alice-Skripten zurückgegriffen werden kann.

Ein Individuum soll den gleichen Namen tragen, mit dem es auch im Skript aufgerufen werden kann. So erhalten die Konstrukte des Rahmenwerks ihren eindeutigen Klassennamen (inklusive Paketbezeichnung). Die Objekte aus den Alice-Welten werden in der Ontologie unter den gleichen Bezeichnungen abgelegt, die auch in der Benutzeroberfläche des Rahmenwerks verwendet werden, welche jedoch zuvor auf die mit Ontologien verträgliche Form normalisiert werden. Da diese Bezeichnungen meistens von Anwendern eingegeben werden, können sie wertvolle natürlichsprachliche Informationen zu den Objekten enthalten.

5. Implementierung

Die Extraktion der Alice-API-Ontologie erfolgt mit Java-Bordmitteln. Dabei werden möglichst viele in Alice bereits enthaltene Funktionen und Datenstrukturen wiederverwendet. Die vorhandene Objektstruktur analysieren wir zur Laufzeit über das *Java Reflection-API* und bilden sie auf die Ontologie ab.

5.1. Ontologie

Die Implementierung beabsichtigt automatische Population der Alice-API-Ontologie durch den Extraktor. Als Grundgerüst dafür wurde die zugehörige Taxonomie und die benötigten Relationen mit dem graphischen Editor *Protégé* erstellt. Dabei wurden die in Kapitel 4.2 beschriebenen Konzepte unverändert in die Struktur der Ontologie aufgenommen.

Als Beschreibungssprache der Alice-API-Ontologie haben wir OWL (Web Ontology Language) gewählt. Zur Laufzeit des Extraktors bilden wir die extrahierten Konstrukte des Rahmenwerks als Individuen auf die Konzepte der Ontologie ab und verknüpfen sie durch Relationen untereinander. Zum Befüllen der Ontologie verwenden wir die für Java verfügbare *OWL API*-Bibliothek ([OWL]).

5.2. Extraktor

Die für die Population der Ontologie zuständige Anwendung wird aus der Klasse `Extractor` initialisiert und führt die Extraktion des Alice-Rahmenwerks in zwei Stufen aus:

1. Als erstes werden alle vom Rahmenwerk vordefinierten Eigenschaften der Objekte in besondere Instanzen der Ontologie extrahiert. Diese Eigenschaften sind dann für alle anderen Instanzen der jeweiligen Ontologie-Klasse verfügbar.

2. In der zweiten Phase kann die Ontologie mit weiteren Daten angereichert werden, die aus beliebigen zur Verfügung stehenden Alice-Welten extrahiert werden.

Das Ergebnis der Extraktion ist eine OWL-Ontologie, die alle für den Anwender verfügbaren Eigenschaften von Alice enthält. Diese Ontologie wird als `*.owl`-Datei in dem Export-Verzeichnis abgelegt.

5.2.1. Extraktion der Alice-Architektur

Während der Extration der Alice-Architektur werden einmalig die in Kapitel 4.1 beschriebenen Eigenschaften der Alice-Objekte extrahiert. Dies erfolgt durch die Ausführung der Methode *extract()* der Klasse `AliceExtractor`, welche für eine definierte Liste von Tupeln der Form „Alice-Klassenname – OWL-Instanzname“ alle Animationen und Funktionen aus der Alice-Architektur extrahiert und in die Ontologie speichert.

Der eigentliche Vorgang der Ontologiepopulation erfolgt in den Methoden *extractResponses()* und *extractQuestions()* der Klassen `AliceResponseExtractor` bzw. `AliceQuestionExtractor`. Im Wesentlichen weisen die beiden Klassen eine sehr ähnliche Struktur auf. Sie benutzen die Klasse `AuthoringToolResources` (siehe Kapitel 4.1.7), die in der Lage ist, die Liste aller verfügbaren Methoden (Funktionen) zu einer gegebenen Alice-Klasse anzugeben. Diese werden ausgewertet, um zugehörige Parameter erweitert und anschließend in die OWL-Ontologie exportiert.

5.2.2. Extraktion der Alice-Welten

Die Herausforderung bei der Extraktion der Alice-Welten besteht darin, dass die von unterschiedlichen Anwendern erstellten Szenerien nicht standardisiert sind und deshalb der Algorithmus zu deren Extraktion erweiterbar und anpassbar sein muss.

Da alle zu extrahierenden Elemente der Alice-Welten von der Klasse `Element` abgeleitet sind, lässt sich der Vorgang der Extraktion anhand dieser Eigenschaft strukturieren. Der Extraktionsalgorithmus läuft alle Elemente einer Welt rekursiv ab und bearbeitet sie abhängig von ihrem Typ (und ggf. anderen Eigenschaften). Zu diesem Zweck besteht der Welt-Extraktor aus mehreren Bearbeitern (Klassen welche für die Extraktion bestimmter Elementtypen zuständig sind), die von der Klasse `ElementHandler` abgeleitet sind.

ElementHandler

Die abstrakte Klasse `ElementHandler` implementiert die Extraktion der Elemente einer Alice-Welt.

Aufgrund des hierarchischen Aufbaus der Objekte (siehe Kapitel 4.1.1) eignet sich bei der Analyse der Alice-Welten der Einsatz von Rekursion am besten. Dabei wird das Element selbst mit den bereits vorhandenen Instanzen der Alice-Ontologie verglichen. Wird ein Objekt mit der gleichen Struktur in der Ontologie gefunden, muss es nicht mehr aufgenommen werden. Stattdessen fassen wir alle Eigenschaften gleicher Objekte in einer Instanz zusammen.

Zuständigkeitskette

Für die Analyse eines Elements wird eine Variante des Entwurfsmusters „Zuständigkeitskette“ (siehe [Gam04], S. 410ff) verwendet. Dieses Verhaltensmuster reduziert die Kopplung der einzelnen Bausteine des Extraktionsalgorithmus. Dadurch wird der Extraktionsalgorithmus sehr leicht erweiterbar, ohne dass die Struktur bestehender Klassen verändert werden muss.

Dazu werden alle benötigten Bearbeiter miteinander über das Feld *next* verknüpft, ähnlich einer einfach verketteten Liste. Die Klasse `ElementHandler` liefert eine abstrakte Methode *handleElement(·)*, welche ein Element aus einer Alice-Welt als Parameter erhält und es auf die Ontologie abbilden kann. Jeder Bearbeiter implementiert des Weiteren die boolesche Methode *accepts(·)*, deren Rückgabewert besagt, ob der Bearbeiter ein ihm übergebenes Element-Objekt bearbeiten kann. Ist dies der Fall, so wird das Element durch den Aufruf

von *handleElement(·)* von diesem Bearbeiter extrahiert. Anderenfalls leitet der Bearbeiter das Element an seinen Nachfolger weiter.

In der Klasse `WorldExtractor` wird die Zuständigkeitskette einmalig initialisiert und die Reihenfolge der Bearbeiter festgelegt. Das Konstrukt kann jederzeit um zusätzliche Bearbeiter-Klassen erweitert werden, ohne dass eine Änderung in den bestehenden Klassen nötig wird. Dazu reicht es, den neu implementierten Bearbeiter in `WorldExtractor` bei der Zuständigkeitskette zu registrieren.

Zu den konkreten Unterklassen von `ElementHandler` gehören:

- `WorldHandler`, verantwortlich für die Abbildung der Welt-Wurzelknoten auf die Ontologie;
- `ModelHandler`, der für die Extraktion aller Alice-Objekte und ihrer Bestandteile zuständig ist;
- `PoseHandler`, der die Posen (siehe 4.1.2.4) der Objekte extrahiert;
- `UserDefinedResponseHandler`, der die benutzerdefinierten Methoden der Objekte bearbeitet und
- `DefaultHandler`, der immer akzeptiert und somit alle von der Zuständigkeitskette nicht bearbeiteten Elemente abfängt.

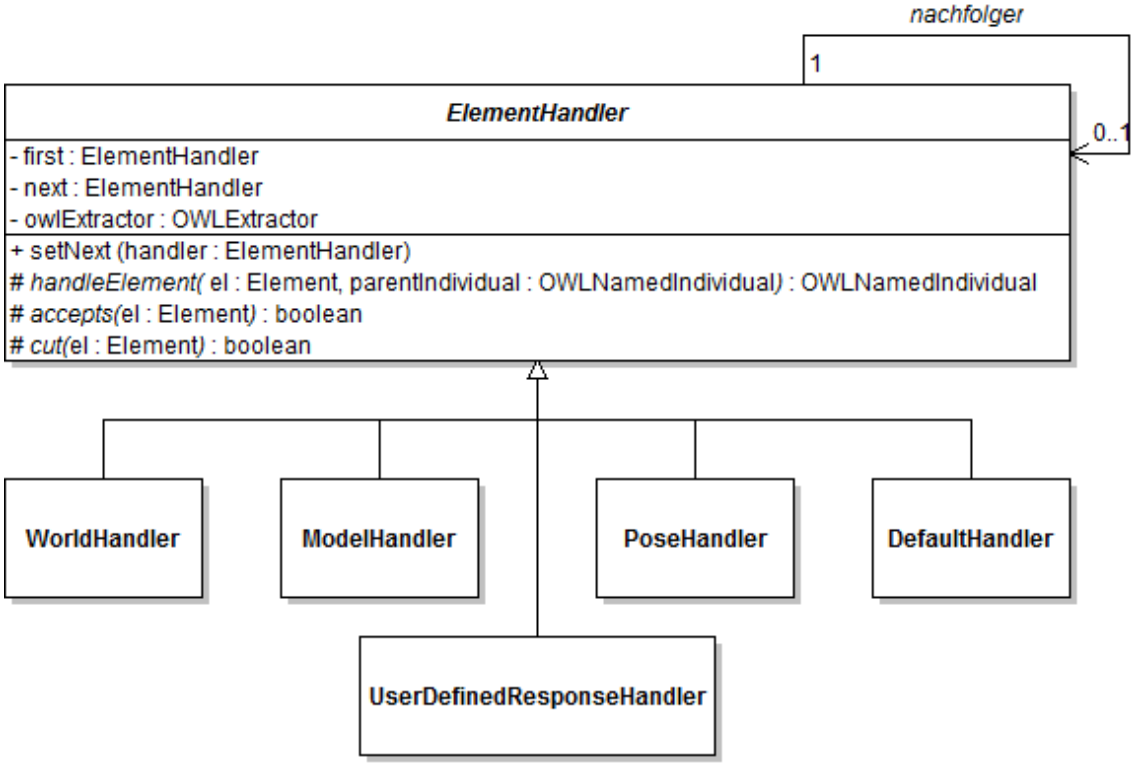


Abbildung 5.1.: Klassendiagramm der ElementHandler

Schablonenmethode *handleBranch(·)*

Der Rekursionsschritt beim Extrahieren einer Alice-Welt ist bereits in der Methode *handleBranch(·)* des `ElementHandlers` implementiert. Dazu wird das Entwurfsmuster „Schablonenmethode“ angewendet (siehe Abb. 5.2). Darin werden sowohl konkrete Methoden der Klasse `ElementHandler`, als auch abstrakte Methoden verwendet, die erst in ihren Subklassen definiert werden. Die Schablonenmethode ermöglicht es konkreten Implementierungen

der `ElementHandler` die Extraktion eines Elementes zu übernehmen, ohne die Struktur der Bearbeitung eines Teilbaums zu verändern ([Gam04], S. 366).

Die Methode *cut(·)* eines Bearbeiters besagt dabei, ob der ElementHandler die Kind-Knoten des aktuellen Elements weiterverfolgen soll, oder diesen Teilbaum abschneiden und verwerfen kann. Wird ein Teilbaum bearbeitet, so erfolgt der Aufruf von *handleBranch(·)* für alle Kind-Knoten des Eltern-Elements (siehe Abb. 5.2)

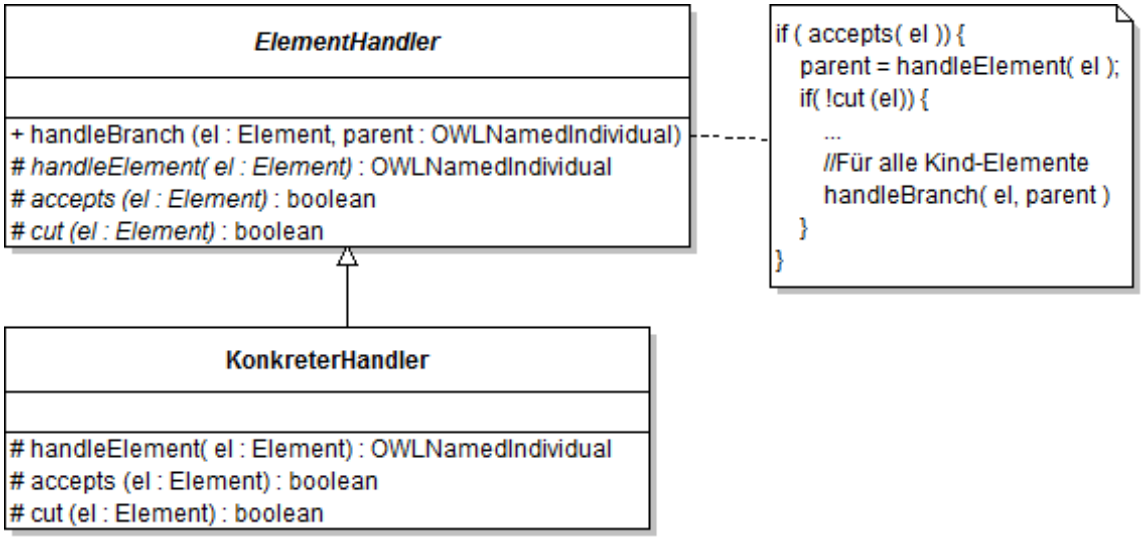


Abbildung 5.2.: Schablonenmethode *handleBranch(·)*

5.3. Modell-Extraktion

Die extrahierte Struktur der Alice-Objekte bildet das Herzstück der Ontologie. Um die entstehende Ontologie für die weitere Verwendung im Gesamtprojekt vorzubereiten, holen wir alle nützlichen Informationen aus den Objekten heraus und führen bereits in diesem Schritt eine Optimierung der extrahierten Konzepte durch.

5.3.1. First-Class Modelle

Der `ModelHandler` bearbeiter die Elemente vom Typ `Model` und unterscheidet zwischen den Objekten erster Klasse (siehe Abschnitt 4.1.2.7) und deren Bestandteilen. Diese Unterscheidung wird anhand des Felds *isFirstClass* des Modells vorgenommen.

5.3.2. Modelläquivalenz

Während der Extraktion von Alice-Welten werden die neuen Konstrukte mit den bereits in der Ontologie bestehenden verglichen. Hiermit erkennen wir die bereits extrahierten Konstrukte und gruppieren gleiche Objekte aus den Alice-Welten zu einer Objekt-Instanz in der Alice-API-Ontologie.

Der Vergleich eines Elements mit Objekten der Ontologie erfolgt rekursiv. In einem Rekursionsschritt wird das Element selbst mit den bereits vorhandenen Instanzen der Alice-Ontologie verglichen. Kommt es zu einer Übereinstimmung, so werden nach dem rekursiven Abstieg die Kind-Knoten des Elements mit den Bestandteilen des Objekts aus der Ontologie verglichen. Die Rekursion wird bis zu den Blatt-Knoten des Objekts durchgeführt. Findet der Extraktor ein baugleiches Objekt in der Ontologie, so wird kein neues Objekt angelegt, sondern das bestehende um die neuen Eigenschaften des zu extrahierenden Modells erweitert.

In dem Äquivalenztest müssen die Bestandteile eines Objekts (siehe Relation *consistsOf* im Abschnitt 4.2.2) inferiert werden. Zu diesem Zweck verwenden wir den Reasoner *Pellet 2.3.0* (siehe [SPG+07]).

5.3.3. Namensschema

Da in der Alice-API-Ontologie die extrahierten Instanzen nur anhand von ihren Namen identifiziert werden, ist es nicht möglich, mehrere unterschiedliche Objekte mit gleichen Namen zu erstellen. Stattdessen liefert das OWL-API beim Versuch, ein neues Objekt zu erstellen, das bereits in der Ontologie existente gleichnamige Objekte zurück. Dieser Umstand würde die Struktur der Ontologie durcheinander bringen: so würde zum Beispiel eine Instanz *Kopf*, die bereits als Bestandteil eines Menschen referenziert war, bei wiederholtem Einfügen im Kontext eines Hundes eine zweite Referenz als Bestandteil des Hundes erhalten.

Um diesem Problem zu entgehen, verwenden wir bei den Objekten das Konzept der vollständig angegebenen Namen (Fully-Qualified Name). Damit lassen sich Objekte immer eindeutig ihrem Ursprung zuordnen. Wir definieren eine (durch ein vordefiniertes Trennzeichen) verkettete Namensgebung, die dem Namen des Elements immer den Namen seines Elternknotens voranstellt. Damit ist anhand von dem Namen des Elements immer der Name seines Eltern-Knotens im Objekt-Baum festgehalten.

6. Evaluation

In diesem Kapitel evaluieren wir die Ergebnisse des Extraktionsalgorithmus, der im Laufe dieser Bachelorarbeit implementiert wurde. Die Qualität des Alice-Extraktors bewerten wir durch dessen Anwendung auf bestehende Alice-Welten, deren Auswahl wir im Abschnitt 6.1 treffen. Dabei legen wir einerseits Wert auf die Vollständigkeit der Ontologien (Abschnitt 6.2), als auch die Qualität und Flexibilität der Aggregation redundanter Informationen (Abschnitt 6.3), die in den bestehenden Welten enthalten sind. Des Weiteren bewerten wir die Qualität der Begriffe, die schließlich in der Alice-API-Ontologie zur Verfügung stehen (Abschnitt 6.4). Im Abschnitt 6.5 fassen wir die Ergebnisse der Evaluation zusammen.

6.1. Auswahl der Test-Welten

Zu dem Alice-Rahmenwerk existieren bereits einige Welten, die von ihren Urhebern im Internet (zum Beispiel [duk]) zum Download bereitgestellt wurden. Zur Evaluation haben wir 12 geeignete Welten ausgesucht, die folgende Grundanforderungen erfüllen: jedes eigenständige Objekt der Szenerie muss in einer baumartigen Struktur mit einem Wurzelknoten gespeichert sein. Außerdem muss jedes Objekt seine Eigenschaften und Methoden kapseln. Damit wir im Gesamtprojekt Objekte der Alice-API-Ontologie mit Begriffen aus der natürlichen Sprache sinnvoll verbinden können, setzen wir eine natürlichsprachige Namensgebung für Objekte und Methoden der Alice-Welt.

In der Tabelle 6.1 sind die Alice-Welten mit den Statistiken zu ihrer Zusammensetzung aufgeführt. Die Anzahl der Elemente setzt sich aus allen Knoten der jeweiligen Welt zusammen. Dazu zählen unter Anderem die vollständigen Syntaxbäume der benutzerdefinierten Methoden und alle Eigenschaften der Objekte (siehe auch Abschnitt 4.1.8). Die Modelle unterscheiden wir hier nach Modellen erster Klasse (siehe Abschnitt 4.1.2.7) und deren Bestandteilen, die wir hier als Modelle zweiter Klasse bezeichnen.

6.2. Vollständigkeit

Ein wichtiges Kriterium für die Bewertung der Alice-API-Ontologie ist die vollständige Übertragung aller verfügbaren Konstrukte aus dem Rahmenwerk und den existierenden Welten.

Die Ergebnisse der Extraktion vordefinierter Konstrukte des Rahmenwerks wurden hauptsächlich durch Sichtprüfungen und Vergleiche mit der Alice-Benutzeroberfläche validiert.

Welt	#Elemente	#Modelle		#Posen	#Methoden
		1.Kl	2.Kl		
arcticWorld.a2w	6607	29	354	119	149
ballerinaDance.a2w	974	11	122	0	16
bigchickentc3.a2w	871	15	114	0	29
classroom.a2w	696	18	108	4	14
Haloween.a2w	2954	108	774	0	114
kittyStory.a2w	4190	160	753	18	45
moonQuestions.a2w	510	5	59	0	15
nemoMath.a2w	498	11	43	1	13
pjsDream.a2w	1988	67	280	1	27
schoolSafety.a2w	1853	26	345	0	36
TerenceDance.a2w	1268	50	96	50	27
waterWorld.a2w	1476	24	307	0	30

Tabelle 6.1.: Ergebnisse der Evaluation

Da die Menge dieser Konstrukte überschaubar ist, konnte jedes davon einzeln überprüft werden. Hiermit ist die Vollständigkeit der Alice-API-Ontologie auf dieser Stufe der Extraktion sichergestellt.

Die Vollständigkeit der importierten Alice-Welten begründen wir mit der Architektur des Extraktionsalgorithmus. Alle zu extrahierenden Elemente der Alice-Welten (Modelle, Methoden, Parameter etc.) befinden sich in dem Objektbaum, der mit Alice-eigenen Funktionen aus der `*.a2w`-Datei geladen wird (siehe Abschnitt 2.1.3).

Der Entwurf des Extraktors als Zuständigkeitskette (vergleiche Abschnitt 5.2.2) erlaubte es, bei der Implementierung iterativ vorzugehen. So haben wir als erstes den `DefaultHandler` implementiert, der in der Zuständigkeitskette ein Element des Objektbaums verarbeitet, wenn kein für dieses Element zuständiger Bearbeiter gefunden wurde. `DefaultHandler` verändert die Ontologie nicht, gibt dem Entwickler aber eine Rückmeldung über ein nicht behandeltes Element. Dieses Verhalten stellt sicher, dass kein Element einer Alice-Welt unbemerkt übergangen wird.

Im Laufe der iterativen Entwicklung des Extraktors wurden wiederholt Durchsichten der entstandenen Alice-API-Ontologie vorgenommen, um deren Qualität stichprobenartig bewerten zu können. Anhand von mehreren Alice-Welten haben wir die Ontologie auf Konsistenz und Übereinstimmung mit der Struktur der Objekte in der Alice-Benutzeroberfläche untersucht und konnten für alle Stichproben lückenlose Deckung der OWL-Ontologie mit der Struktur der Objekte feststellen.

Die Statistiken aus Tabelle 6.1, die ebenfalls von dem Extraktor erzeugt wurden, haben wir ebenfalls durch Nachzählen in der Alice-Benutzeroberfläche stichprobenartig überprüft und konnten die Resultate des Extraktors für alle Stichproben validieren. Dies bestätigt die Richtigkeit unserer Überlegungen zu der vollständigen Überdeckung der Elemente betrachteter Alice-Welten.

6.3. Eliminierung redundanter Objekte

Wie in Kapitel 5.3.2 beschrieben, versuchen wir bereits bei der Extraktion redundante Objekte in der Ontologie zusammenzufassen.

In der Alice-Welt `arcticWorld.a2w` wird der rekursive Objektvergleicher explizit getestet. Hierzu enthält die Welt mehrere gleiche Pinguine. Ein Pinguin (`Penguin2`) unterscheidet

sich jedoch von den anderen in der dritten Rekursionsebene durch ein fehlendes Bestandteil `LeftFoot`. Er wird vom Extraktor wie gewünscht als ein unbekanntes Modell aufgefasst und als ein neues Individuum der OWL-Klasse `FirstClassModel` in die Ontologie aufgenommen. Die anderen Pinguine weisen dieselbe Struktur wie der erste Pinguin auf, und werden verworfen.

Als weiterer Testfall wurde eine Alice-Welt mehrfach in die Alice-API-Ontologie importiert. Dabei konnte festgestellt werden, dass die Ontologie nach dem ersten Import-Vorgang vom Extraktor nicht mehr geändert wurde, was die Korrektheit des rekursiven Vergleichs bestätigt.

Animationen und Funktionen (siehe Abschnitt 4.1.5) und weitere Kind-Knoten redundanter Objekte werden trotzdem in die Ontologie übernommen, und erweitern die Funktionalität bereits vorhandener baugleicher Modelle.

In der Tabelle 6.2 sind die Ergebnisse der Eliminierung mehrfach vorkommender Objekte zu sehen. Die Anzahl von First-Class Objekten meint dabei die Gesamtanzahl in der jeweiligen Welt vor Reduktion. Mit lokalen Duplikaten sind Vorkommen gleicher Objekte innerhalb einer Welt gemeint, während die globalen Duplikate die in der Ontologie bereits enthaltenen Objekte berücksichtigen. In dem Testlauf wurden die Welten in alphabetischer Reihenfolge in die Ontologie importiert.

Die Anzahl der gefundenen Duplikate unterscheidet sich lokal und global mindestens um eins. Dies liegt daran, dass der Boden der Szenerie (Objekt `ground`) standardmäßig vom Rahmenwerk automatisch eingefügt wird und in allen Welten daher gleich heißt.

Außerdem wurden in 11 aus 12 untersuchten Welten entweder lokale oder globale Duplikate (außer `ground`) gefunden, was die Notwendigkeit der Reduktion bestätigt.

Tendenziell sind in größeren Welten höhere Anzahl an Duplikaten zu verzeichnen. Das hängt oft damit zusammen, dass Tiergruppen oder Gärten durch Kopieren von Tieren oder Bäumen umgesetzt werden.

Welt	**#FirstClass Objekte**	**#Duplikate (lokal)**	**#Duplikate (global)**
arcticWorld.a2w	29	13	13
ballerinaDance.a2w	11	5	6
bigchickentc3.a2w	15	7	11
classroom.a2w	18	8	9
Haloween.a2w	108	94	96
kittyStory.a2w	160	68	71
moonQuestions.a2w	5	0	2
nemoMath.a2w	11	0	2
pjsDream.a2w	67	16	28
schoolSafety.a2w	26	12	16
TerenceDance.a2w	50	37	40
waterWorld.a2w	24	6	8

Tabelle 6.2.: Ergebnisse der Modell-Aggregation

Ein weiteres Beispiel für die Arbeit des Vergleichers sind die Objekte `bonzai` aus den Welten `kittyStory.a2w` und `waterWorld.a2w`. Die beiden Objekte besitzen die gleiche hierarchische Struktur und man würde erwarten, dass sie in der Alice-API-Ontologie zu einem Objekt verschmolzen werden. Allerdings variiert die Benennung eines Bestandteils des Baums in der ersten Rekursionsebene. Die Instanz `root3` wurde in der Welt `kittyStory.a2w` zu `root03` umbenannt.

Der rekursive Vergleicher ist in unserer Implementierung so definiert, dass die Namen der First-Class-Modelle *ähnlich* sein können, um der Gleichheit zweier Objekte zu genügen. *Ähnlich* bedeutet in dem Fall, dass wir uns beim Vergleich der Instanznamen aus der Klasse `FirstClassModel` von Sonderzeichen, Zahlen sowie Groß- und Kleinschreibung abstrahieren. Die zweite Voraussetzung ist die absolute Gleichheit der Namen aller Bestandteile des Objekts. Dieser Vergleich wird rekursiv fortgesetzt, bis die Blätter des Objektbaumes erreicht sind, oder eine Ungleichheit auftritt. So wären zum Beispiel zwei Alice-Objekte `Auto23` und `AUTO` mit gleichen Bestandteilen `Reifen1` bis `Reifen4` gleich.

Diese Entscheidung beruht auf der Überlegung, dass wir zwei Objekte zusammenfassen wollen, wenn sie vom Benutzer des Rahmenwerks kopiert und unverändert gelassen werden. Verändert der Benutzer die Bestandteile des kopierten Objekts, so will er offensichtlich etwas an dem Objekt ändern und wir erhalten ein neues Objekt, welches von dem ersten abgeleitet ist.

Mit anderen Worten nehmen wir an, dass es sich bei einem `Auto1` mit den Bestandteilen `REIFEN01` bis `REIFEN04` und `tuer1`, `tuer2`, und einem anderen `Auto2` mit Bestandteilen `Reifen1` bis `Reifen4` und `Tuer01` und `Tuer02` mit hoher Wahrscheinlichkeit um unterschiedliche Autos handelt. Andererseits kann eine reine Umbenennung der Bestandteile eines Objekts nicht trivial wie im oberen Beispiel ausfallen, wenn man zum Beispiel den `Reifen1` zu `Rad1` umbenennt. In diesem Fall lässt sich mit der reinen Normalisierung der Zeichenketten kein zufriedenstellendes Ergebnis erreichen.

Aus diesen Gründen erhalten wir auch in der resultierenden Alice-API-Ontologie die beiden Instanzen `bonzai` und `bonzai1`, die sich in der Namensgebung eines ihrer Bestandteile unterscheiden.

6.4. Qualität der Begriffe

Die Qualität der Begriffe in der extrahierten Ontologie ist stark von den verwendeten Begrifflichkeiten in den Alice-Weltdateien abhängig. Beim Programmieren des Extraktors wurde darauf Wert gelegt, dass die Bezeichnungen der Objekteigenschaften nicht durch ungeschickte Normalisierung o.Ä. eine ineffiziente Auswertung des Anfrage-Systems nach sich ziehen. Endgültig lässt sich die Ontologie unter diesem Aspekt jedoch erst beim Aufbau des Anfrage-Systems bewerten und anpassen.

In der Studienarbeit von Sebastian Weigelt, die sich aufbauend auf den Ergebnissen dieser Bachelorarbeit mit der Erweiterung der Alice-API-Ontologie um Synonyme beschäftigt, wird die Synonym-Abdeckung der extrahierten OWL-Individuen bewertet. Mit dieser Metrik können wir die Brauchbarkeit der Begriffe beurteilen. Die relative Synonym-Abdeckung, also das Verhältnis der Individuen, die kein Synset erhalten haben zu der Gesamtanzahl der Individuen, erreicht für die Ontologien einzelner Alice-Welten Werte zwischen 97,80% bis 98,55%. Für die aggregierte Alice-API-Ontologie beträgt dieser Wert 99,57%, was positiv zu bewerten ist (vergleiche [Wei12], Kapitel 6.1).

Zusätzlich kann aufgrund von kaskadierender Namensgebung der OWL-Individuen die Bestimmung des Begriffskontextes effizient stattfinden. Da zum Aussortieren von passenden Synonymen die hierarchische Struktur der Modelle bekannt sein muss (siehe [Wei12], S.51), vermeiden wir mit der kaskadierenden Benennung der OWL-Individuen zusätzliche Anfragen an die Ontologie über die Relation *consistsOf*. Dieser Umstand steigert die Effizienz bei der Erweiterung der Alice-API-Ontologie um Synonyme.

6.5. Zusammenfassung

In Anbetracht der in diesem Kapitel aufgeführten Ergebnisse der Evaluierung lässt sich die Umsetzung des Alice-API-Extraktors hinsichtlich der Vollständigkeit, Eliminierung

redundanter Objekte und Qualität der extrahierten Begriffe insgesamt positiv bewerten. Natürlich hängt insbesondere das letzte Evaluationskriterium stark mit der Wahl der untersuchten Alice-Welten zusammen. Da wir jedoch im Abschnitt 6.1 bestimmte Forderungen an die extrahierten Alice-Welten stellen, darf an dieser Stelle das Ziel der Implementierung, eine vollständige und in sich konsistente Alice-API-Ontologie ohne Informationsverlust zu erzeugen, als erreicht betrachtet werden.

7. Zusammenfassung

Der Ansatz natürlichsprachlicher Programmierung ist ein spannendes Forschungsfeld, welches vor dem Hintergrund der sich weiterentwickelnder Spracherkennungssoftware breite Anwendung unter Nicht-Informatikern finden kann. So wäre es zukünftig möglich, einfache sicherheitsunkritische Vorgänge mit natürlicher Sprache zu beschreiben und einem Informationssystem zur Ausführung zu geben.

Das Projekt „Programmieren mit natürlicher Sprache“ befasst sich mit der Konstruktion von 3D-Videosequenzen für das Alice-Rahmenwerk. Alice bietet uns als ein abgeschlossenes System die Möglichkeit, Texte in natürlicher Sprache auf Konstrukte des Rahmenwerks und bereits existierenden Alice-Welten abzubilden und auf diese Art neue Alice-Welten (Videosequenzen) automatisch zu erzeugen.

Im Laufe dieser Bachelorarbeit haben wir die Architektur des Rahmenwerks für den Einsatz im Gesamtprojekt analysiert und alle für die Erzeugung von Alice-Welten relevanten Eigenschaften dokumentiert. Sicherlich wird in den Folgearbeiten Bedarf bestehen, die Architektur des Alice-Rahmenwerks noch detaillierter zu analysieren, um zum Beispiel die genaue Implementierung des Alice-Welt-Generators an die Vorgaben des Rahmenwerks anzupassen. Da der Quellcode des Alice-Rahmenwerks aber über keinerlei Dokumentation verfügt, bietet die in dieser Arbeit erstellte Dokumentation einen guten Einstiegspunkt für weitere Analyse des Rahmenwerks.

Des weiteren entstand eine Extraktionssoftware, welche als Grundlage für alle in diesem Projekt folgenden Arbeiten eine Alice-API-Ontologie aus dem Alice-Rahmenwerk und beliebigen geeigneten Alice-Welten erzeugen kann. Da die Alice-Welten auch von Nicht-Informatikern programmiert werden können, bezeichnen wir eine Welt als *geeignet*, wenn jedes eigenständige Objekt der Szenerie in einer baumartigen Struktur mit einem Wurzelknoten gespeichert ist, und seine Eigenschaften und Methoden kapselt. Damit wir im Gesamtprojekt Objekte der Alice-API-Ontologie mit Begriffen aus der natürlichen Sprache sinnvoll verbinden können, erwarten wir außerdem treffende Namensgebung für Objekte und Methoden einer geeigneten Alice-Welt.

Bei Bedarf der Modellierung eines interaktiven Verhaltens der Alice-Welten lässt sich die Ontologie um die von Behavior abgeleiteten Klassen (siehe Abschnitt 4.1.2.6) oder weitere benötigte Strukturen erweitern. Sollte es nötig werden, weitere bis jetzt nicht benötigten Eigenschaften der Alice-Welten zu extrahieren, kann die verwendete Architektur leicht erweitert und angepasst werden.

In der Evaluation haben wir die Ergebnisse des Extraktors untersucht und konnten zeigen, dass es die Anforderung erfüllt, die Alice-Welten vollständig und strukturerhaltend auf Konzepte der Ontologie abzubilden. Abschließende Beurteilung und Verbesserungen des Systems können allerdings erst im weiteren Verlauf des Gesamtprojekts vorgenommen werden.

Literaturverzeichnis

[CAB+00] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove und Kevin Christiansen: *Alice: lessons learned from building a 3D system for novices.* In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '00, Seiten 486–493, New York, NY, USA, 2000. ACM, ISBN 1-58113-216-6. http://alice.org/publications/chialice.pdf.

[Car] Carnegie Mellon University: *Alice.* http://www.alice.org.

[Con97] Matthew J. Conway: *Alice: Easy-to-Learn 3D Scripting for Novices.* Dissertation, Faculty of the School of Engineering and Applied Science, University of Virginia, Dezember 1997. http://alice.org/publications/ConwayDissertation.PDF.

[CV05] Philipp Cimiano und Johanna Völker: *Text2Onto.* In: *Natural Language Processing and Information Systems*, Band 3513 der Reihe *Lecture Notes in Computer Science*, Seiten 257–271. Springer Berlin / Heidelberg, 2005, ISBN 978-3-540-26031-8. http://dx.doi.org/10.1007/11428817_21, 10.1007/11428817_21.

[duk] http://www.cs.duke.edu/csed/alice09/examples.php, besucht: 29.08.2012.

[ESV] C. Tempich E. Simperl und D. Vrandecic.: *A methodology for ontology learning.*, Band Bridging the Gap between Text and Knowledge: Selected Contributions to Ontology Learning and Population from Text, 167.

[Gam04] Dirk [Uebers.] Gamma, Erich ; Riehle (Herausgeber): *Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software.* Professionelle SoftwareentwicklungProgrammers choice. Addison-Wesley, Muenchen [u.a.], 2004, ISBN 3-8273-2199-9.

[Gru93] Thomas R. Gruber: *A translation approach to portable ontology specifications.* Knowledge Acquisition, 5(2):199 – 220, 1993, ISSN 1042-8143. http://www.sciencedirect.com/science/article/pii/S1042814383710083.

[HV08] Peter Haase und Johanna Voelker: *Ontology Learning and Reasoning - Dealing with Uncertainty and Inconsistency.* In: Paulo da Costa, Claudia d'Amato, Nicola Fanizzi, Kathryn Laskey, Kenneth Laskey, Thomas Lukasiewicz, Matthias Nickles und Michael Pool (Herausgeber): *Uncertainty Reasoning for the Semantic Web I*, Band 5327 der Reihe *Lecture Notes in Computer Science*, Seiten 366–384. Springer Berlin / Heidelberg, 2008, ISBN 978-3-540-89764-4. http://dx.doi.org/10.1007/978-3-540-89765-1_21, 10.1007/978-3-540-89765-1_21.

[Lie06] Thorsten Liebig: *Reasoning with OWL - System Support and Insights.* Technischer Bericht, Universität Ulm, 2006.

[OWL] *OWL API*. http://owlapi.sourceforge.net/, besucht: 30.09.2012.

[RFJ08] D. Ratiu, M. Feilkas und J. Jurjens: *Extracting Domain Ontologies from Domain Specific APIs*. In: *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, Seiten 203 –212, april 2008.

[Sab04] Marta Sabou: *Extracting ontologies from software documentation: a semi-automatic method and its evaluation*. In: *Workshop on Ontology Learning and Population (ECAI-OLP)*, Valencia, Spain, August 2004. http://oro.open.ac.uk/id/eprint/3028.

[SPG+07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur und Yarden Katz: *Pellet: A practical OWL-DL reasoner*. Web Semantics: Science, Services and Agents on the World Wide Web, 5(2):51 – 53, 2007, ISSN 1570-8268. http://www.sciencedirect.com/science/article/pii/S1570826807000169, <ce:title>Software Engineering and the Semantic Web</ce:title>.

[Sta] Stanford Center for Biomedical Informatics Research: *protégé*. http://protege.stanford.edu/, besucht: 29.08.2012, Stanford Center for Biomedical Informatics Research.

[Sta09] Steffen Staab: *Handbook on Ontologies*, 2009, ISBN 978-3-540-92673-3. http://dx.doi.org/10.1007/978-3-540-92673-3.

[Stu09] Heiner Stuckenschmidt: *Ontologien : Konzepte, Technologien und Anwendungen*. Informatik im Fokus. Springer, Berlin, 2009, ISBN 978-3-540-79330-4. http://dx.doi.org/10.1007/978-3-540-79333-5.

[W3C] *OWL Web Ontology Language Reference*. W3C Recommendation. http://www.w3.org/TR/owl-ref/.

[Wei12] S. Weigelt: *Programmieren in natürlicher Sprache: Aufbau einer Alice-Ontologie – Korpus-Ontologie-Assoziation*. Studienarbeit, Karlsruher Institut für Technologie (KIT) – IPD Tichy, 2012. http://www.ipd.kit.edu/Tichy/theses.php?id=204.

[Wel09] Kathrin Weller: *Ontologien: Stand und Entwicklung der Semantik für WorldWideWeb*. 2009.

[YCO99] Hongji Yang, Zhan Cui und P. O'Brien: *Extracting ontologies from legacy systems for understanding and re-engineering*. In: *Computer Software and Applications Conference, 1999. COMPSAC '99. Proceedings. The Twenty-Third Annual International*, Seiten 21 –26, 1999.

[ZWRH06] Yonggang Zhang, René Witte, Juergen Rilling und Volker Haarslev: *An Ontology-based Approach for Traceability Recovery*. In: *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006*, Seiten 36–43, 2006. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.3547&rep=rep1&type=pdf.

Anhang

A. Objektbezogene Funktionen in Alice

Klasse	**Format**
IsCloseTo	<subject> is within <threshold> of <object>
IsFarFrom	<subject> is at least <threshold> away from <object>
DistanceTo	<subject> distance to <object>
DistanceToTheLeftOf	<subject> distance to the left of <object>
DistanceToTheRightOf	<subject> distance to the right of <object>
DistanceAbove	<subject> distance above <object>
DistanceBelow	<subject> distance below <object>
DistanceInFrontOf	<subject> distance in front of <object>
DistanceBehind	<subject> distance behind <object>
Width	<subject>'s width
Height	<subject>'s height
Depth	<subject>'s depth
IsSmallerThan	<subject> is smaller than <object>
IsLargerThan	<subject> is larger than <object>
IsNarrowerThan	<subject> is narrower than <object>
IsWiderThan	<subject> is wider than <object>
IsShorterThan	<subject> is shorter than <object>
IsTallerThan	<subject> is taller than <object>
IsLeftOf	<subject> is to the left of <object>
IsRightOf	<subject> is to the right of <object>
IsAbove	<subject> is above <object>
IsBelow	<subject> is below <object>
IsInFrontOf	<subject> is in front of <object>
IsBehind	<subject> is behind <object>
PointOfView	<subject>'s point of view
Position	<subject>'s position
Quaternion	<subject>'s quaternion
Pose	<subject>'s current pose
PartKeyed	<owner>'s part named <key>
VariableNamed	<owner>'s variable named <variableName> of type <valueClass>

Tabelle A.1.: Vordefinierte Funktionen eines Objekts

B. Globale Funktionen des Alice-Rahmenwerks

Klasse	Format
Not	not `a`
And	both `a` and `b`
Or	either `a` or `b`, or both
NumberIsEqualTo	`a==b`
NumberIsNotEqualTo	`a!=b`
NumberIsGreaterThan	`a>b`
NumberIsGreaterThanOrEqualTo	`a>=b`
NumberIsLessThan	`a<b`
NumberIsLessThanOrEqualTo	`a<=b`
RandomBoolean	choose true <probabilityOfTrue> of the time
RandomNumber	random number
StringConcatQuestion	`a` joined with `b`
ToStringQuestion	<what> as a string
AskUserForNumber	ask user for a number <question>
AskUserYesNo	ask user for yes or no <question>
AskUserForString	ask user for a string <question>
DistanceFromLeftEdge	mouse distance from left edge
DistanceFromTopEdge	mouse distance from top edge
TimeElapsedSinceWorldStart	time elapsed
Year	year
MonthOfYear	month of year
DayOfYear	day of year
DayOfMonth	day of month
DayOfWeek	day of week
DayOfWeekInMonth	day of week in month
IsAM	is AM
IsPM	is PM
HourOfAMOrPM	hour of AM or PM
HourOfDay	hour of day
MinuteOfHour	minute of hour
SecondOfMinute	second of minute
Min	minimum of `a` and `b`
Max	maximum of `a` and `b`
Abs	absolute value of `a`
Sqrt	square root of `a`
Floor	floor `a`
Ceil	ceiling `a`
Sin	sin `a`
Cos	cos `a`
Tan	tan `a`
ACos	arccos `a`
ASin	arcsin `a`
ATan	arctan `a`
ATan2	arctan2 `ab`
Pow	`a` raised to the `b` power
Log	natural log of `a`
Exp	e raised to the `a` power
IEEERemainder	IEEERemainder of `a/b`

Round	round `a`
ToDegrees	`a` converted from radians to degrees
ToRadians	`a` converted from degrees to radians
SuperSqrt	the `b`th root of `a`
RightUpForward	<right>, <up>, <forward>

Tabelle B.2.: Vordefinierte Funktionen

C. Vordefinierte Animationen der Alice-Objekte

Klasse	**Format und Beschreibung**
MoveAnimation	<subject> move <direction><amount> Bewegung des gesamten Modells entlang eines Vektors
TurnAnimation	<subject> turn <direction><amount> Rotation des Modells rechts- und links herum oder vor- und rückwärts
RollAnimation	<subject> roll <direction><amount> Kippen des Modells nach Rechts oder Links
ResizeAnimation	<subject> resize <amount> Skalieren des Modells um ein Faktor *amount*
SayAnimation	<subject> say <what> Animation eines Textes in einer Sprechblase
ThinkAnimation	<subject> think <what> Animation eines Textes in einer Gedankenblase
SoundResponse	<subject> play sound <sound> Abspielen einer Audiosequenz
PositionAnimation	<subject> move to <target> Bewegung zu einem anderen Objekt der Szene
MoveTowardAnimation	<subject> move <amount> toward <target> Bewegung um *amount* Längeneinheiten in Richtung eines anderen Objekts
MoveAwayFromAnimation	<subject> move <amount> away from <target> Entfernung um *amount* Längeneinheiten weg von einem anderen Objekt
QuaternionAnimation	<subject> orient to <asSeenBy> Modell an einem anderem Objekt ausrichten, sodass die Blickrichtung übereinstimmt
TurnToFaceAnimation	<subject> turn to face <target> Modell sich einem anderen Objekt zuwenden lassen
PointAtAnimation	<subject> point at <target> Blickrichtung eines Modells zum Zentrum eines anderen Modells wenden
PointOfViewAnimation	<subject> set point of view to <object> Position und Blickrichtung des Modells mit den eines anderen Modells überschreiben
PoseAnimation	<subject> set pose <pose> Objekt eine vordefinierte Pose einnehmen lassen
StandUpAnimation	<subject> stand up Den Up-Vektor eines Modells am Up-Vektor der Welt ausrichten
MoveAtSpeed	<subject> move at speed <direction><speed> Bewegung des gesamten Modells mit einer Geschwindigkeit *speed*
TurnAtSpeed	<subject> turn at speed <direction><speed> *Turn*-Animation mit einer Geschwindigkeit *speed*
RollAtSpeed	<subject> roll at speed <direction><speed> *Roll*-Animation mit einer Geschwindigkeit *speed*
TurnToFaceConstraint	<subject> constrain to face <target> Stellt sicher, dass das Modell einem Objekt zugewandt ist
PointAtConstraint	<subject> constrain to point at <target> Stellt sicher, dass das Modell zum Zentrum eines anderen Objekts zugewandt ist

Tabelle C.3.: Vordefinierte Animationen