Entwicklung domänenspezifischer Sprachen mit ANTLR am Beispiel eines Pac-Man-Klons

Studiengang Medieninformatik Master
Beuth Hochschule für Technik Berlin

Marcel Brüning
s67176@beuth-hochschule.de

Simon Lischka
simon@lischka.co

Marcel Piater
s67357@beuth-hochschule.de

***Zusammenfassung*—Die nachfolgenden Seiten beschreiben die Entwicklung eines Pac Man Browserspiels mit JavaScript. Die Schwerpunkte des Projekts liegen in der automatischen Codegenerierung mit domänenspezifischen Sprachen (DSL) für die Levels und die artifizielle Intelligenz (AI). Die DSL der AI beschreibt verschiedene Strategien der vom Computer gesteuerten Spielfiguren.**

I. Über das Spiel

In diesem Absatz wird das Spielprinzip von Pac Man kurz vorgestellt und auf die Entwicklung eingegangen. Dabei werden auch die eingesetzten Technologien beschrieben.

A. Spielprinzip

Pac Man ist eine Spielfigur, die durch ein Labyrinth so schnell und effektiv wie möglich gesteuert werden soll um alle vorhanden Punkte einzusammeln. Auf dem Weg erschweren diverse Hindernisse den Siegeszug. Bewegliche Gegner (vier Geister) die von einer AI gesteuert werden sowie ein komplexes Labyrinth erschweren die Bewältigung des Spiels. Pro Spiel stehen verfügt Pac Man drei Leben. Wird er von einem Gegner gefasst, so geht ein Leben verloren. In dem Labyrinth gibt es durch das Sammeln von Münzen die Möglichkeit, die Gesamtpunktzahl zu erhöhen.

Über Früchte, welche Pac Man auf seinem Weg finden kann, gelangt dieser in den Angriffsmodus. In diesem Modus wird Pac Man für eine kurze Zeit vom Gejagten zum Jäger und kann seinerseits Geister fressen, was ihm wiederum zu zusätzlichen Punkten verhilft. Sollte es Pac Man innerhalb der drei Leben schaffen sämtliche Punkte auf dem Spielfeld zu konsumieren, so hat er das Level erfolgreich absolviert und startet ein neues Level.

B. Verwendete Technologien

Da das Spiel in JavaScript implementiert ist, muss der Codegenerator JavaScript-Code erzeugen, der von der Applikation angesteuert wird. Zur Interpretation von DSL-Dateien und Erzeugung von JavaScript-Code wird die Java-Implementierung von ANTLR4 verwendet.

Ein naheliegender Gedanke in der Planungsphase war es, die JavaScript-Implementierung von ANTLR zu verwenden. Auf diese Weise ließe sich eine einheitliche Programmiersprache in der gesamten Codebasis einsetzen.

Eine Rechtfertigung für dieses Vorgehen besteht darin, dass ein Team was sich nur auf eine Programmiersprache einarbeiten muss mit geringem Zeitaufwand einheitliche Coding-Standards etablieren kann, da diese nur für eine Sprache aufgestellt werden müssen. Ein programmiertechnischer Austausch zwischen allen Teammitgliedern, etwa durch Code Reviews oder die Klärung von sprachspezifischen Problemen würde vermutlich ebenfalls erleichtert werden. Das Ziel eines einheitlichen Einsetzens von JavaScript wäre also das Erlangen höherer Codequalität gewesen.

Da zur Java-Implementierung von ANTLR ausführlichere Dokumentation existiert als für die JavaScript-Variante und die Java-Implementierung etablierter erscheint, haben wir jedoch von der Verwendung von ANTLR für JavaScript abgesehen. Wir hatten die Vermutung, dass die zusätzliche Einarbeitungszeit und für uns unerwartetes Verhalten der JavaScript-Implementierung den zeitlichen Rahmen des Projektes übersteigen und in keinem Verhältnis zur höheren Codequalität durch den Einsatz einer einheitlichen Programmiersprache stehen würde.

Wir haben jedoch für die in JavaScript verfasste Codebasis gezielt Technologien zur Qualitätssteigerung ausgewählt. Hierzu gehört das Framework RequireJS, welches die Modularisierung und das Importieren von Klassen ähnlich wie in Java ermöglicht. Für das Einbetten von durch ANTLR erzeugte JavaScript-Klassen müssen diese als Module ladbar sein - RequireJS war für uns deshalb Voraussetzung um unser Vorhaben erfolgreich umsetzen zu können. Objektorientiertes Programmieren nach dem Paradigma Separation of Concerns (SoC) und der Aufbau einer übersichtlichen Projektstruktur werden durch Trennung von Klassen in einzelne Dateien durch RequireJS ebenfalls stark erleichtert.

Mit Underscore.js sind in kurzer Syntax und auf höherem Abstraktionsgrad insbesonders häufig eingesetzte Idiome wie Listeniterationen abbildbar. Hierdurch ergibt sich weniger Raum für Fehler. Underscore bietet eine Reihe von Helferfunktionen, die darauf ausgerichtet sind, funktionale Programmierung zu erleichtern[1].

Kritische Funktionen wurden in JavaScript mit Unit Tests versehen. Hierzu setzen wir das Framework Jasmine ein. Eine

[1] Underscore.js Einführung, http://underscorejs.org/

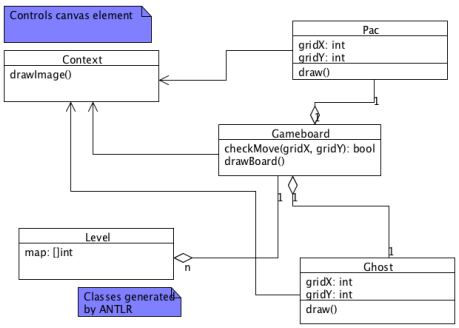

Abbildung 1. Reduziertes Klassendiagramm der zentralen Spielklassen

hohe Testabdeckung, wie sie mit Test Driven Development möglich ist, haben wir jedoch nicht priorisiert.

C. Umsetzung

Das zu spielende Level wird durch ein 2-Dimensionales Array abgebildet. Die Elementes des Spielfelds sind durch Zahlencodes im Array abgebildet. Eine Null steht dabei für ein freies Stück Weg. Die Eins und die Zwei stehen für ein begehbares Feld mit einem Punkt oder einer Frucht darauf. Wände werden durch eine Drei repräsentiert und sind von Pac Man und den Gegnern nicht begehbar. Auf diese Art und Weise können beliebig komplexe Levels erstellt und in das Spiel geladen werden.

Zur Darstellung werden für die genannten Elemente Bildobjekte in das Spiel geladen, die auf einem Canvas-Element des HTML Dokuments gezeichnet werden. Zum Zeichnen der Bildobjekte wird durch das Array iteriert und für jedes Array-Element das dem Zahlencode zugehörige Bildobjekt auf den 2D Kontext des Canvas gezeichnet.

Für die Spielfiguren wird ein zweites Canvas-Element verwendet, das transparent über den Canvas-Element des Levels liegt. Die Motivation hierfür ist, dass der Zustand der Spielfiguren sich in der Regel häufiger ändert als der des Spielfeldes. Ein Flackern, was durch zusätzliches Neuzeichnen des Levels bei jeder Figurveränderung entstehen kann, wird durch Trennung in zwei Canvas-Elemente reduziert.

Die Figuren werden genauso wie die Objekte des Levels als Bilder geladen. Sie operieren auf dem selben Koordinatensystem wie das Level. Abbildung 2 zeigt das ausgeführte Spiel.

Das Klassendiagramm in Abbildung 1 stellt den zuvor beschriebenen Sachverhalt dar. Die Objekte von Spieler und Gegner (Pac und Ghost) besitzen die Instanzvariablen gridX und gridY, die wie beschrieben den Wertebereich des Levels besitzen. Die Funktion checkMove() der Klasse Gameboard gleicht den nächsten Schritt einer Figur mit dem Array des Levels ab. Bei Zahlencode Null, Eins oder Zwei liefert sie true zurück und erlaubt den gewünschten Schritt der Figur. Entsprechend wird bei einer Drei, also einer Wand false zurückgeliefert und verhindert somit den nächsten Schritt der Spielfigur. Diese Funktion wird von Klassen Ghost und Pac bei der Umsetzung des nächsten Spielzugs verwendet.

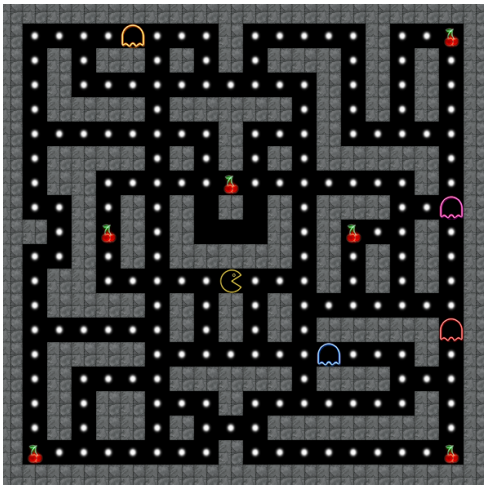

Abbildung 2. Screenshot des ausgeführten Pac-Man Spiels.

Um die Bewegungen der Figuren sichtbar zu machen existiert die Funktion updateOnInterval(), die durch den Scheduler des Browsers alle 150 Millisekunden aufgerufen wird. Sie zeichnet das Level und die Spielfiguren mit den entsprechenden Informationen und Positionen neu. In der Methode werden Kollisionsabfragen der Klasse GameBoard aufgerufen, die prüfen ob Pac Man gerade einen Punkt bzw. eine Frucht frisst (Methode checkPacsEating()) oder mit einem Geist kollidiert ist (Methode checkKills()). Die im Spiel agierende AI berechnet in jedem Interval die Richtung der Geister neu. Ihre Berechnung bezieht sich also immer genau auf einen Spielzug und hat dementsprechend die nächste zu gehenden Richtung als Rückgabewert.

Abbildung 3 stellt das FMC-TAM Diagramm des Spiels dar. Ghost und Pacman sind als Akteure dargestellt, die ihre Position in Abstimmung mit den Request-Antworten von checkMove aktualisieren. Akteur Strategy und Speicher Level werden durch ANTLR generiert. Im Zusammenhang mit Strategy ist Mittler GhostQuery relevant. GhostQuery stellt die Schnittstelle des generierten Codes zum Spiel dar und wird bei der Beschreibung der AI DSL in Abschnitt II-B näher erläutert.

Interessant ist, dass die Akteure sich nicht automatisch auf die gleiche Modularisierungseinheit, also Beispielsweise nur Klassen abbilden. Checkmove etwa wird im aktuellen Entwicklungsstand durch eine einzelne Methode repräsentiert, Hitdetection durch mehrere Methoden der Klasse Gameboard. Akteure Pacman und Ghost beispielsweise existieren als eigene Klassen, ebenso wie Strategy und GhostQuery.

Es ist wahrscheinlich, dass eine Auslagerung von Hitdetection in eine eigenständige Klasse bei zunehmender Komplexität sinnvoll wäre und dadurch das Design durch die Einhaltung des SoC-Paradigmas verbessert werden würde. Durch die Aufstellung des FMC-TAM Diagramms lässt sich diese bevorstehende Designänderung antizipieren, auch wenn wir uns in der vorliegenden Implementierung dazu entschlossen haben, sie nicht umzusetzen.

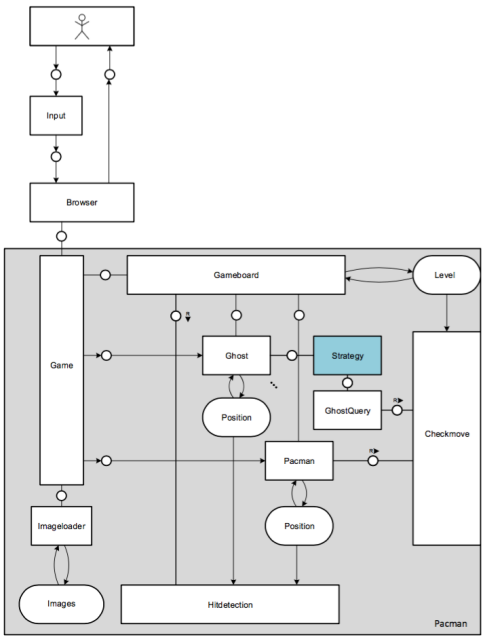

Abbildung 3. FMC-TAM Diagramm der umgesetzen Applikation

II. CODEGENERIERUNG

ANTLR4 generiert auf Grundlage der DSL Lexer zur lexikalischen Analyse und Parser zur syntaktischen Analyse. Wir haben uns bei der Umsetzung der Level- und der AI DSL für das Einsetzen von Listener Klassen zum Traversieren des Abstrakten Syntaxbaums (AST) entschlossen.

Für beide Grammatiken erstellten wir Subklassen der generierten Listener. Diese Klassen enthalten jeweils eine Datenstruktur als Instanzvariable, die beim Traversieren befüllt wird. Im Fall der Level DSL handelt es sich um eine Liste, bei der AI DSL um eine Baumstruktur.

Erst nach Beenden der Traversierung werden die Daten in der Klasse CodeGenerator durch Interpretation der Datenstrukturen, die in den Listenern erzeugt wurden, geschrieben. Abbildung 4 stellt die beteiligten Klassen dar. Bei totalValues und initialRoot handelt es sich um die erwähnten Datenstrukturen.

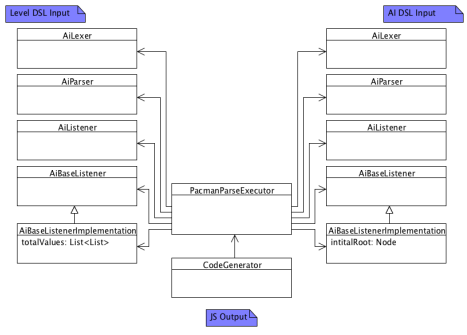

Abbildung 4. Beteiligte Klassen bei der Codegenerierung mit ANTLR

A. Level DSL

Der LevelListener ist für das Auslesen der Daten zuständig. In der Java Klasse LevelBaseListenerImplementation.java welche von der LevelListener.java erbt ist zum einen der Zugriff auf die Daten möglich zum anderen wurde an dieser Stelle die Überprüfung der Spielfeldgröße umgesetzt welches genau eine Arraygröße von 20x20 voraussetzt. Die parseLevel() Methode welche in der PacmanParseExecuter.java Klasse definiert ist erwartet eine .csv Datei als Argument. welche nach dem Schema wie in Tabelle 1 aufgebaut ist. Des Weiteren wird in dieser Methode die Initialisierung aller abhängigen Klassen, die für diesen Prozess nötig sind, erledigt.

Die CodeGenerator.java Klasse ist neben der Generierung für die AI auch für die Erstellung einer level.js Datei zuständig welche nach JavaScript Syntax erstellt wird. Die Datei level.js (Abbildung 1) wird direkt in dem Ordner `/dsl_pacman/pacman/levels/` generiert und von der Gameboardklasse verwendet um das Level zu erzeugen.

```
define([], function () {
  return {
    floor: 0,
    point: 1,
    fruit: 2,
    wall: 3,
    map: [
      [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3],
      [3,1,1,1,1,1,1,1,1,3,1,1,1,1,1,3,1,1,2,3],
      [3,1,3,1,3,3,1,3,1,3,3,3,1,3,1,3,1,1,3],
      [3,1,3,1,1,1,1,1,1,1,1,1,1,3,1,3,1,3,1,3],
      [3,1,3,3,3,3,1,3,3,3,3,3,1,3,1,3,1,3,1,3],
      [3,1,1,1,1,1,1,1,1,3,1,1,1,3,1,1,1,1,1,3],
      [3,1,3,3,3,3,1,3,1,3,1,3,1,3,3,3,3,3,1,3],
      [3,1,3,3,1,1,1,1,1,2,1,1,1,1,1,1,3,1,3],
      [3,1,1,3,1,3,1,3,0,3,0,3,1,3,3,3,1,1,1,3],
      [3,3,1,3,2,3,1,3,0,0,0,3,1,3,2,1,1,3,1,3],
      [3,1,1,3,1,3,1,3,3,3,3,3,1,3,1,3,1,3,1,3],
      [3,1,3,3,1,1,1,1,1,1,1,1,3,1,3,1,3,1,3],
      [3,1,3,3,3,3,1,3,1,3,1,3,1,1,1,1,1,1,3],
      [3,1,1,1,1,1,1,3,1,3,1,3,1,3,3,3,3,3,1,3],
      [3,1,3,3,3,3,1,1,1,1,1,1,1,1,1,1,3,1,3],
      [3,1,3,1,1,1,1,3,3,3,3,3,1,3,3,3,1,1,1,3],
      [3,1,3,1,3,3,1,1,1,3,1,1,1,1,1,1,3,1,3],
      [3,1,3,1,3,3,1,3,1,1,1,3,3,3,3,3,3,1,3],
      [3,2,1,1,1,1,1,1,1,3,1,1,1,1,1,1,1,2,3],
      [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3]
    ]
  }
});
```

Listing 1. Generierte Level-Klasse in JavaScript

1) Spielfeldaufbau: Das Spielfeld bildet ein zweidimensionales Array ab, welches je nach Spielfeld-Design mit Zahlen von Null bis Drei befüllt wird. Für jeden Wert im Array werden dann die festgelegten Gegenstände auf der grafischen Oberfläche, dem Canvas Objekt dargestellt. Die Spezifikation sieht vor, dass ein Array der Größe von exakt 20x20 erstellt werden muss. Die Definition des Arrays wird in einer .csv Datei als Tabelle erstellt und ist die DSL. Eine beispielhafte Darstellung die das Prinzip für den Aufbau des Spielfeldes veranschaulicht ist in Listing I dargestellt.

Zulässig sind als *Value* definiert, nur die Zahlen Null, Eins, Zwei und Drei in beliebiger Reihenfolge. Als Trennsymbol einzelner Values ist allein das Semikolon als *Separator* zu-

Tabelle I
SPIELFELD-DEFINITION PER DSL

3	3	3	3	3	0	freier Weg
3	2	1	1	3	1	Punkt
3	1	0	1	3	2	Frucht
1	1	1	2	3	3	Mauer/Hindernis
3	3	3	3	3		

lässig. Diese Anreihung von Value und Trennsymbol kann beliebig oft vorkommen solange bis das Ende einer Zeile, welche als *row* festgelegt ist, erreicht wird. Das Zeilenende wird durch ein *LineBreak* oder aber durch ein *EOF* signalisiert. Letzteres dient als terminales Symbol und signalisiert das Ende des einzulesenden Spielfeld-Arrays beziehungsweise beendet das Einlesen der Zeilen. Alle Zeilen zusammen werden als *field* spezifiziert.

2) Grammatik: Um den korrekten Aufbau des Spielfelds zu gewährleisten werden zunächst domänenspezifische Gültigkeitsregeln festgelegt. In unserem Fall werden diese durch Zahlen, Sonderzeichen und reguläre Ausdrücke repräsentiert. Die Spezifikation der Arraygröße ist nicht Teil dieser DSL und die Überprüfung dafür findet an anderer Stelle statt welche im Kapitel "Codegenerierung mit ANTLR" beschrieben wird. Die in dem vorherigen Absatz, "Spielfeldeaufbau", festgelegten Regeln werden in einer level.g4 Datei (Listing 2) formal festgelegt um dann mit dem ANTLR4 Tool Java Code zu generieren welcher zur Überprüfung der Gültigkeit des aufzubauenden Spielfeldes (Vergl. Listing1) verwendet wird.

```
grammar Level;

field: row* EOF ;
row: value (Separator value)* (LineBreak | EOF) ;

value: Value ;
Separator: ';' ;
LineBreak: '\r'?'\n' | '\r';
Value: ('0'|'1'|'2'|'3')+ ;
```

Listing 2. Auszug aus der DSL spezifizierenden Grammatik level.g4

Zulässig sind als *Value* definiert, nur die Zahlen Null, Eins, Zwei und Drei in beliebiger Reihenfolge. Als Trennsymbol einzelner Values ist allein das Semikolon als *Separator* zulässig. Diese Anreihung von Value und Trennsymbol kann beliebig oft vorkommen solange bis das Ende einer Zeile, welche als *row* festgelegt ist, erreicht wird. Das Zeilenende wird durch ein *LineBreak* oder aber durch ein *EOF* signalisiert. Letzteres dient als terminales Symbol und signalisiert das Ende des einzulesenden Spielfeld-Arrays beziehungsweise beendet das Einlesen der Zeilen. Alle Zeilen zusammen werden als *field* spezifiziert.

B. AI DSL

Die durch Intrepretation der AI DSL generierten JavaScript-Klassen ermittelt auf Basis der aktuellen Richtung eines Geistes dessen Richtung für den folgenden Spielzug. Eingabe und Ausgabewert der AI-Klasse ist also die Richtung eines Geistes. Aus diesem Grund ist ein Verständnis der Tokens nötig, die für das Ausdrücken einer Richtung eingesetzt werden.

Tabelle II
RICHTUNGSTOKENS DER AI DSL MIT EXAMPLARISCHER BELEGUNG DER TOKENS

Token	Beschreibung	Richtung			
->	Aktuelle Richtung	R	L	U	D
<-	Entgegengesetzte Richtung	L	R	D	U
=>	Alternative Richtung	D	U	L	R
<=	Entgegengesetzt alternative Richtung	U	D	R	L

Tabelle III
OPERATOREN MIT SYNTAXBEISPIEL, DIE METHODE CHECKMOVE ANSTEUEREN

	Syntax	Beschreibung	Beispiel
if*	`if *(DIR) { //BLOCK A } else { //BLOCK B }`	*If-Free Operator* Ausführen von A, wenn Richtung DIR frei. Sonst Ausführen von B.	`if *(->) { -> } else { <- }`
**	`**{ DIR; DIR; }`	*Filter-Free Operator* Richtung die nicht begehbar sind, werden entfernt.	`**{ =>; <=; }`
*n	`*1{ DIR; DIR; }`	*Filter-FreeN Operator* Filter gemäß Filter-Free Operator, Auswahl des n-ten Elements.	`*1{ =>; <=; }`

Tabelle II stellt die Richtungs-Tokens dar. Hier wird durch die DSL insofern eine Abstraktion vorgenommen, als das eine Richtung relativ angegeben wird. Token `->` wird mit einem Wert aus der Menge `UP, DOWN, RIGHT, LEFT` belegt. Beim Verfassen der AI muss nicht mehr beachtet werden, welche Richtung dieser Token bei der Ausführung des Spiels entspricht. Relevant für die Entwicklerin oder den Entwickler ist es zu entscheiden ob ein Geist in der bisherigen Richtung weiterläuft, in eine der alternativen Richtungen ausweicht oder umkehrt. Die entstehende Simplifizierung wird durch den Pseudocode in Listing 3, der das Laufen in die entgegengesetzte Richtung implementiert.

```
if DIRECTION == RIGHT:
    return LEFT
elif  DIRECTION == LEFT:
    return RIGHT
elif DIRECTION == UP:
    return DOWN
elif DIRECTION == DOWN:
    return UP
```

Listing 3. Umkehren der Richtung in Pseudocode

Da die DSL die Notwendigkeit von Conditionals zur einzelnen Behandlung der Richtung entfernt, würde in Syntax der DSL ein `<-` Token genügen, um die Logik des Pseudocodes abzubilden.

Pac-Man AI DSLs beginnen mit dem Namen der DSL in runden Klammern. Innerhalb dieses Blocks werden Operatoren erwartet. Entscheidend für das Auswählen einer Richtung ist oftmals, ob diese frei ist. Wie in Absatz I-C beschrieben, ist im JavaScript-Spiel der Akteur *Checkmove* dafür zuständig, eine Auskunft darüber zu geben, ob eine Richtung frei ist. Die DSL bietet die in Tabelle III gelisteten Operatoren an, um Checkmove anzusteuern.

Bei den Operatoren ist zu beachten, dass der `If-Free-` und `Filter-FreeN`-Operator jeweils eine einzelne Richtung als Rückgabewert besitzen, wohingegen der `Filter-Free`-Operator eine Liste von Richtungen zurückgibt. Der Begriff des Rückgabewerts bildet sich auf aufgerufene JavaScript Methoden der von ANTLR compilierten DSL ab.

Der `Filter-Free`-Operator darf also beispielsweise nicht als äußerster Block einer DSL stehen, da mehrere Werte zurückgegeben werden aber nur eine einzige Richtung den Folgespielzug bestimmt. Da JavaScript eine dynamisch typisierte Sprache ist, würde ein Laufzeitfehler aufgrund eines unerwarteten Listen-Typs auftauchen. Dies ist unerwünscht, weil der Fehler spät in der Verarbeitungskette auftaucht und dessen Ursprung schwer zurückzuverfolgen ist. Eine entsprechende Validierung bei der Kompilierung der DSL wäre möglich und sinnvoll, wurde in der vorliegenden Implementierung jedoch nicht umgesetzt.

Die DSL in Listing 4 benutzt den `Filter-FreeN`-Operator um die erste der übergebenen Richtungen auszuwählen, die frei ist. Klar sichtbar ist, dass ein weiterlaufen in der aktuellen Richtung (`->`) vorzuziehen ist, und versucht werden soll in eine der alternativen Richtungen (`=>`, `<=`) auszuweichen, bevor umgekehrt wird (`<-`).

```
simple(
    *1{
      ->;
      =>;
      <=;
      <-;
    }
)
```

Listing 4. AI DSL mit F`Filter-FreeN`-Operator

Zum zufälligen Auswählen von Richtungen führen wir den Random-Operator (Tabelle IV) ein.

Tabelle IV
RANDOM OPERATOR MIT SYNTAXBEISPIEL

	Syntax	Beschreibung	Beispiel

%	`%(r:r) {`
	`  DIR;`
	`  DIR;`
	`}`
		Random Operator
		Aus den Richtungen `DIR`
		wird zufällig ein Wert ausgewählt.
		Die Wahrscheinlichkeit
		ist durch den ratio-Parameter
		`r` festgelegt.
			`if *(->) {`
			`  ->`
			`} else {`
			`  <-`
			`}`

Durch die vorgestellte Sprachmitteln lassen sich nun auch komplexere Strategien abbilden. In der DSL aus Listing 5 ist ein `If-Free`-Block zu sehen. Ist die aktuelle Richtung des Geistes nicht frei, so wird der `else`-Block ausgewertet und mit Hilfe des `Filter-FreeN`-Operators die erste freie Richtung gemäß der gleichen Rangordnung von Listing 4 gewählt. Ist die aktuelle Richtung frei, so soll mit Hilfe einer Zufallsverteilung entschieden werden, ob diese weitergegangen oder eine der freien alternativen Richtungen eingeschlagen werden soll.

```
random(
    if *(->) {
        %(50:25:25) {
            ->;
            **{
              =>;
              <=;
            }
        }
    } else {
      *1 {
        =>;
        <=;
        <-;
      }
    }
)
```

Listing 5. AI DSL mit `If-Free`-Block und Zufallsverteilung

In Listing 5 ist sichtbar, dass der Filter-Free Operator mit dem Random Operator geschachtelt wurde. Die DSL erlaubt beliebige Schachtelungstiefen. Beispielsweise die Syntax von Listing 6 ebenfalls zulässig.

```
if *(->) {
  %(50:25:25) {
    ->;
    **{
      %(50:25:25) {
        ->;
        **{
          =>;
          *1{
            <-;
            ->;
          }
          <=;
        }
    }
    =>;
    <=;
  }
}
```

Listing 6. Komplexere Verschachtelung in AI DSL

Eine beliebige Verschachtelungstiefe, wie durch Listing 6 demonstriert, wird nicht *direkt* durch die Grammatik definiert. Die AI-Grammatik erkennt lediglich die notwendigen Tokens (Listing 7).

```
ai_body: WS* (LINE_BREAK|COMMENT|bracket_close|
    block_declaration||direction_statement);
bracket_open: '{';
bracket_close: '}';
block_declaration: (if_free_statement|
    else_free_statement|random_statement|
    leave_free_statement|get_nth_free_statement);
```

Listing 7. Ausschnitt der AI Grammatik

Das Aufbauen einer rekursiven Baumstruktur erfolgt in der Listener Implementation. Hierzu wird für jeden geparsten *Operator* eine Klasse erstellt, die von der abstrakten Klasse `Node` erbt.

Alle Unterklassen von Node (Abbildung 5) enthalten intern eine Listen-Struktur, in der Kinder-Nodes geführt werden. Jeder Node implementiert eine `renderCode()`-Methode, die JavaScript-Quellcode erzeugt. Der Quellcode der Kinder-Nodes wird durch Methode `childCode()` erzeugt, die über die Kinder-Nodes iteriert, jeweils `renderCode()` aufruft und die Ergebnisse konkateniert.

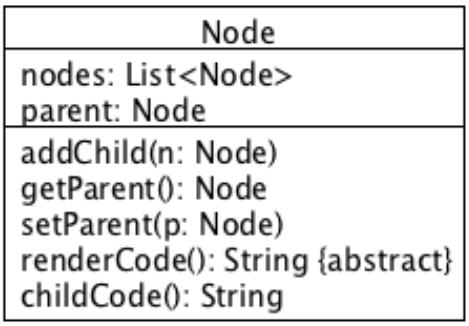

Abbildung 5. Node-Basisklasse als Halter der Datenstruktur zu Codegenerierung

Methode `childCode()` wird per Konvention in jeder Node-Klasse bei `renderCode()` aufgerufen. Eine Node Klasse, erzeugt beim Code-Generieren ebenfalls den JavaScript-Quellcode seiner Kinder. Zum Erzeugen der kompletten AI-JavaScript Klasse reicht das Aufrufen von `renderCode()` auf dem Wurzelknoten, der im Listener als Instanzvariable `initialNode` geführt ist (siehe Abbildung 4). Abbildung 6 zeigt die entstehende Baumstruktur zur DSL von Listing 5.

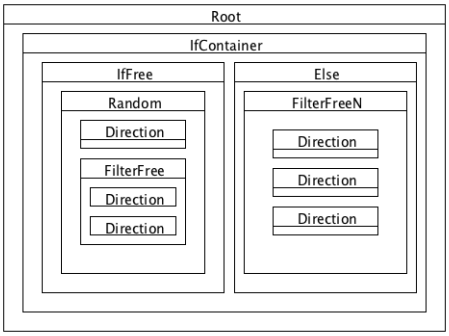

Abbildung 6. Schematische Darstellung der Datenstruktur beim Parsen einer AI DSL (Laufzeit).

Der entstehende JavaScript-Code zur DSL in Listing 5 ist in Listing 8 aufgeführt.

```
define([], function() {
   var strategy = function(queries) {
       var direction = queries.currentDirection();
       return (function() {
           if (queries.isFree(direction)) {
               return queries.
                   randomWithDistribution([
                   50, 25, 25
               ], [
                   direction,
                   queries.filterFree([
                       queries.alternative(
                           direction),
                       queries.alternativeOpposite(
                           direction)
                   ])
               ]);
           } else {
               return queries.filterFreeN(1, [
                   queries.alternative(direction),
                   queries.alternativeOpposite(
                       direction),
                   queries.opposite(direction)
               ]);
           }
       })();
   }
   return strategy;
});
```

Listing 8. Beispiel zu generiertem JavaScript-Code einer AI DSL

Tabelle V zeigt die Abbildung der Tokens der DSL auf JavaScript-Methoden der Klasse GhostQueries, die der Funktion strategy der generierten AI-Klasse als Parameter queries übergeben wird. In der Darstellung werden ebenfalls die Parameter- und Rückgabetypen dieser Methoden aufgezeigt.

Tabelle V
MAPPING VON AI-TOKENS AUF JAVASCRIPT METHODEN DER KLASSE GHOSTQUERIES

Op	Methode in GhostQueries
->	currentDirection(): *String*
<-	opposite(direction: String): *String*
=>	alternative(direction: String): *String*
<=	alternativeOpposite(direction: String): *String*
if*	isFree(direction: String): *Boolean*
**	filterFree(directions: List<String>): *List<String>*
*n	filterFreeN(n: Number, directions: List<String>): *String*
%	randomWithDistribution(ratios: *List<Number>*, dirs: *List<String>*)

Klasse `GhostQueries` dient als Mittler der generieten Strategie mit dem Ghost-Objekt. Sie ruft Methoden `currentDirection`, `gridX()`, `gridY()` und `checkMove()` auf. Der Aufruf von `checkMove()` wird von Klasse `Ghost` an `Gameboard` deligiert. Der Aufruf der generierten Strategie mit Übergabe des Parameters GhostQueries wird in Abbildung 7 dargestellt.

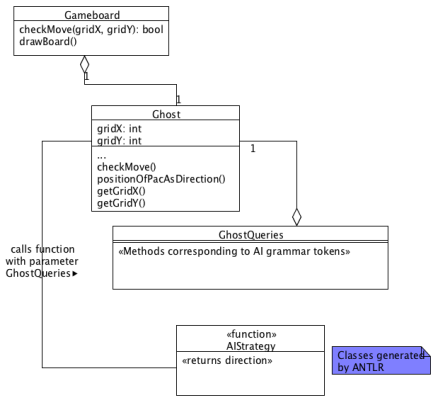

Abbildung 7. Aufruf der generierten Strategie-Methode durch Klasse Ghost.

Neben der Abstraktion bezüglich der Richtungen (siehe Absatz II-B) sorgt Klasse GhostQueries dafür, dass Listenstrukturen bei verschachtelten Aufrufen korrekt vereint werden. Eine Verschachtlung des `Random-Operators` mit `Filter-Free-Operator`, wie in Abbildung 6 gezeigt, hat zur Folge dass Parameter *directions* der

`randomWithDistributions()`-Methode mit einer verschachtelten Liste befüllt wird. Die Ursache ist, dass es sich beim Ergebnis des `Filter-Free-Operator` ebenfalls um eine Liste handelt. Ein Beispiel einer entstehende Datenstruktur des directions-Parameter ist in Listing 9 zu sehen.

```
[
  "UP",
  "DOWN",
  [
    "LEFT",
    "RIGHT"
  ]
]
```

Listing 9. Beispiel einer verschachtelten Datenstruktur als directions-Parameter von randomWithDistributions()

Die Vereingung der Liste wird durch die underscore.js-Methode `flatten` vorgenommen. Die Idiome zur Iteration von underscore.js bestehen aus Funktionen, die als Parameter zum einem die zu iterierende Listenstruktur übergeben bekommen und zum anderen eine Funktion die das jeweilige Listenelement als Parameter empfängt. Für die Implementierung der Klasse GhostQueries war es sehr hilfreich, sprechende Schreibweisen wie die aus Listing 10 einsetzen zu können. Die Syntax erinnert an moderne Programmiersprachen wie Python oder Scala, die funktionale Idiome unterstützen.

```
return _.filter(_.flatten(directions), function(d) {
                return isFree(d);
            });
```

Listing 10. Implementierung der Filter-Free Methode mit Hilfe von funktionalen Hilfsmitteln von underscore.js

III. Fazit und Ausblick

Ziel dieses Projekts war die Umsetzung eines funktionsfähigen Pacman Klons. Ein Teil des Quellcodes sollte aus voher detailliert spezifizierten Modellen automatisch generiert werden. Dabei wurden zwei DSLs eingesetzt. Zum einen eine DSL für die Erstellung des Spielfeldes und zum anderen eine AI DSL für die Umsetzung verschiedenen Strategien für die Bewegung der von Computer gesteuerten Figuren. Somit ist es möglich verschiedene Leveldesigns bzw. unterschiedliche Bewegungsstrategien (Erhöhung des Schwierigkeitsgrades) unabhängig vom Basiscode des Spiels zu generieren. Alle im Vorfeld gesetzten Kriterien wurden erfolgreich umgesetzt.

Bei der Implementierung der AI DSL hat sich bewährt, eine Baumstruktur in der entsprechende Listener-Klasse aufzubauen um diese bei der Codegeneriung zu traversieren. Hierbei taucht allerdings das Problem auf, dass `ANTLR` keine syntaktische Validierung in Bezug auf den Aufbau dieser Baumstruktur vornimmt. Es ist also ohne manuell implementierte Validierungen nicht zu gewährleisten, dass die Operatoren der AI DSL korrekt miteinander kombiniert werden - im schlimmsten Fall resultiert ein Laufzeitfehler im JavaScipt Code.

In der Klasse `AiBaseListenerImplementation` wurde ein erster Ansatz implementiert, eine solche Validierung vorzunehmen (Listing 11). Um eine Aussagekraft für den Anwender zu haben, müsste jedoch eine präzisere Fehlermeldung mit entsprechender Zeilennummer der interpretierten DSL ausgegeben werden. Es ist abzusehen, dass der verfolgte Ansatz bei einer Erweiterung der Grammatik ungenügend ist. Hier wäre eine noch tiefere Auseinandersetzung mit ANTLR empfehlenswert, um womöglich einen standartisierten Lösungsweg verfolgen zu können.

```
private void add(Node n) {
    if ((n instanceof Else) && !(currentNode
        instanceof IfContainer)) {
        System.out.println("Tried␣to␣add␣else␣
            without␣preceding␣if␣block");
    }
    if (!(n instanceof Else) && !(n instanceof
        IfFree) && currentNode instanceof
        IfContainer) {
        System.out.println("Illegal␣state,␣possible␣
            programming␣error:␣Opened␣an␣IfContainer
            ␣and␣trying␣to␣add␣other␣than␣if␣or␣else
            ␣block.");
    }
    this.currentNode.addChild(n);
    // Flat elements that can't contain child nodes
        should not set themselves as currentNode
    if (!(n instanceof Direction || n instanceof
        Reference || n instanceof Assignment)) {
        this.currentNode = n;
    }
}
```

Listing 11. Grundliegende Validierung beim Hinzufügen eines einer Node in `AiBaseListenerImplementation`

Spezialisierte Unit-Tests für die AI DSL aufzustellen wäre ein weiterer Schritt, um die Qualität der Implementierung zu erhöhen. Eine hohe Zuverlässigkeit auf Seite der DSL Implementation zu beheben wäre im realen Anwendungsfall eine Notwendigkeit.

Das Erzeugen von Code mit Hilfe von DSLs scheint in dem gegegeben Anwendungsfall in Hinblick auf Arbeitsteilung im Team äußerst sinnvoll. Es ist wahrscheinlich, dass bei einer komplexeren Pac-Man Anwendung eigenständige Teams für Level-Gestaltung und AI-Entwurf eingesetzt werden würden. Diese Teams sollten von programmiertechnischen Aspekten der eigentlichen Spiels weitesgehend entkoppelt werden um effektiv arbeiten zu können. Durch Anpassung der Codegeneratoren wären die Anwendungsentwickler in der Lage, nach Bedarf neue Technologien einsetzten zu können ohne die anderen Teams in ihrer Arbeit zu stören.