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 folgenden 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 definiert verschiedene Strategien der vom Computer gesteuerten Spielfiguren.**

I. Hinweis zur Begriffsverwendung

Im vorliegenden Text wird der Begriff *JavaScript* als Synonym für *JavaScript*-Implementierungen nach *ECMAScript5*-Standard verwendet.

II. Über das Spiel

Der Inhalt dieses Absatzes ist das Spielprinzip von Pac-Man und die Entwicklung der Spiellogik. Dabei werden die eingesetzten Technologien beschrieben.

A. Spielprinzip

Pac-Man ist eine Spielfigur, die durch ein Labyrinth so effektiv wie möglich gesteuert werden soll, um alle vorhandenen Punkte zu sammeln. Bewegliche Gegner (*vier* Geister), die vom Computer gesteuert werden, sowie ein komplexes Labyrinth erschweren den Siegeszug. Pro Spiel verfügt Pac-Man über *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 in den Ecken des Spielfelds gelangt Pac-Man in den Angriffsmodus. Er wird für kurze Zeit vom Gejagten zum Jäger und kann seinerseits Geister fressen. Das verhilft ihm zu zusätzlichen Punkten. Schafft es Pac-Man innerhalb der *drei* Leben sämtliche Punkte auf dem Spielfeld zu konsumieren, so hat er das Level erfolgreich absolviert und startet ein neues Level.

B. Verwendete Technologien

Das Spiel ist in *JavaScript* implementiert. Teile des Codes werden mit dem Parser-Generator `ANTLR4` auf Basis der DSL erzeugt. Dieser Code wird von der Applikation angesteuert.[1]

Ein naheliegender Gedanke in der Planungsphase war es, die *JavaScript*-Implementierung von `ANTLR4` zu verwenden. Auf diese Weise ließe sich eine einheitliche Programmiersprache in der gesamten Codebasis einsetzen. Ein programmiertechnischer Austausch zwischen allen Teammitgliedern könnte dadurch erleichtert werden. Dieser Austausch kann aus Code-Reviews und dem Klären von sprachspezifischen Problemen bestehen.

Da die *Java*-Implementierung von `ANTLR4` etablierter erscheint und über ausführlichere Dokumentation verfügt als die *JavaScript*-Variante, haben wir von der Verwendung von `ANTLR4` 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 würde.

Wir haben jedoch für die *JavaScript*-Codebasis gezielt Technologien zur Qualitätssteigerung ausgewählt. Hierzu gehört das Framework `RequireJS`, welches die Modularisierung und das Importieren von Klassen ähnlich wie in *Java* ermöglicht.[2] Von `ANTLR4` erzeugte *JavaScript*-Klassen müssen 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.

`Underscore.js` bietet eine Reihe von Helferfunktionen, die darauf ausgerichtet sind, funktionale Programmierung zu erleichtern[3]. Mit `Underscore.js` sind häufig eingesetzte Idiome wie Listeniterationen in kürzerer Syntax und auf höherem Abstraktionsgrad als durch native *JavaScript*-Sprachmittel abbildbar. Hierdurch ergibt sich weniger Raum für Fehler.

Kritische Funktionen wurden in *JavaScript* mit `Jasmine`-Unit Tests versehen. Eine hohe Testabdeckung, wie sie mit *Test Driven Development* möglich ist, haben wir jedoch nicht priorisiert.

C. Umsetzung

Das zu spielende Level besteht aus einem zweidimensionalem Array. Die Elementes des Spielfelds sind durch Zah-

[1] Einführung in ANTLR, http://www.antlr.org/

[2] Require.js Einführung, http://requirejs.org/

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

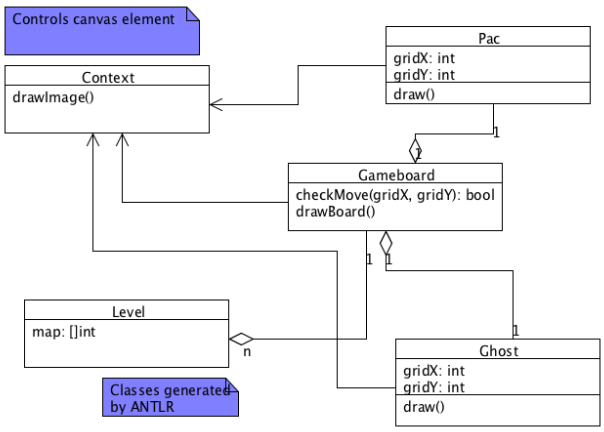

Abbildung 1. Reduziertes Klassendiagramm der zentralen Spielklassen

lencodes abgebildet. Eine *null* steht dabei für ein freies Stück Weg. Codes *eins* und *zwei* stehen für ein begehbares Feld mit einem Punkt oder einer Frucht. Wände werden durch Code *drei* repräsentiert und sind von Pac-Man und den Gegnern nicht begehbar.

Zur Darstellung laden wir Bildobjekte und zeichnen diese auf einem *Canvas*-Element des *HTML* Dokuments. Zum Darstellen der Bildobjekte wird durch das Array iteriert und für jedes Array-Element ein dem Zahlencode zugehörige Bildobjekt auf den 2D-Kontext des *Canvas* gezeichnet.

Ein *zweites Canvas*-Element stellt Spielfiguren dar. Es ist transparent über dem *Canvas*-Element des Levels positioniert. 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 operieren auf dem selben Koordinatensystem wie das Level. Abbildung 2 zeigt das ausgeführte Spiel.

Das Klassendiagramm in Abbildung 1 stellt die beteiligten *JavaScript*-Objekte dar. Die Objekte von Spieler und Gegner (*Pac* und *Ghost*) führen die Instanzvariablen `gridX` und `gridY`, die Koordinaten des Levels sind. 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.

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 nach vorgenommener Aktualisierung neu. In der Methode werden Kollisionsabfragen der Klasse `Gameboard` aufgerufen, die prüfen ob Pac-Man gerade einen Punkt beziehungsweise eine

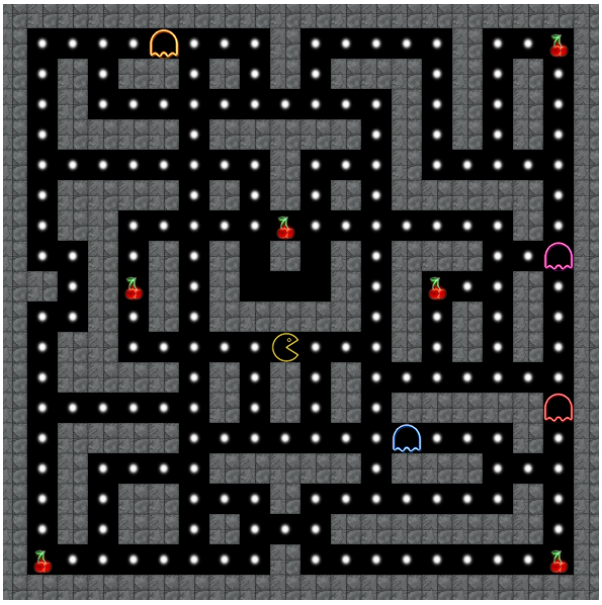

Abbildung 2. Screenshot des ausgeführten Pac-Man Spiels

Frucht frisst (Methode `checkPacsEating()`) oder mit einem Geist kollidiert (Methode `checkKills()`). Die im Spiel agierende AI berechnet in jedem Intervall die Richtung der Geister neu. Ihre Berechnung bezieht sich also immer genau auf einen Spielzug.

Abbildung 3 stellt das *FMC-TAM* Diagramm des Spiels dar. *Ghost* und *Pac-Man* sind als Akteure dargestellt, die ihre Position in Abstimmung mit dem Ergebnis von `checkMove()` aktualisieren. Akteur *Strategy* und *Speicher Level* werden durch `ANTLR4` 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 III-B näher erläutert.

Interessant ist, dass die Akteure sich nicht auf eine einheitliche Modularisierungseinheit, also Beispielsweise nur Klassen abbilden. *Checkmove* wird im vorliegenden Entwicklungsstand durch eine einzelne Methode repräsentiert, *Hitdetection* durch mehrere Methoden der Klasse `Gameboard`. Akteure *Pac-Man* und *Ghost* 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.

III. 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

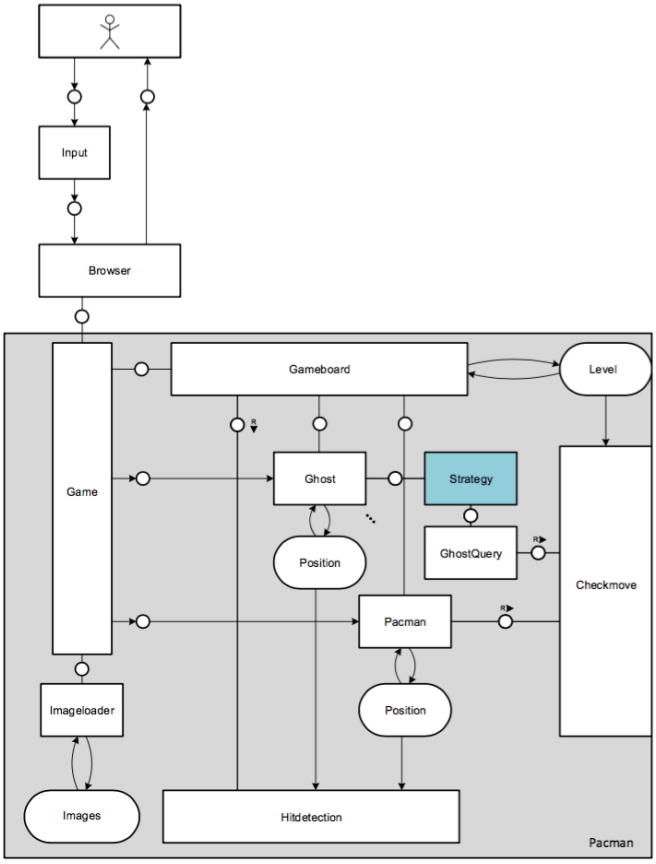

Abbildung 3. *FMC-TAM* Diagramm der umgesetzten Applikation

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 des AST befüllt wird. Im Fall der Level DSL handelt es sich um eine `Liste` und 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 diese Datenstrukturen.

A. Level DSL

Eine Level DSL entspricht dem *CSV*-Format. In der Java Klasse `LevelBaseListenerImplementation`, welche von `LevelListener` erbt, wird eine verschachtelte Listenstruktur erzeugt, die das Spielfeld Zeilen- und Felderweise abbildet. Die `parseLevel()` Methode, welche in der `PacmanParseExecuter` Klasse definiert ist, erwartet den Pfad zu einer *.csv*-Datei als Argument. Des Weiteren wird in dieser Methode die Initialisierung aller abhängigen Klassen, die für diesen Prozess nötig sind, erledigt.

Die `CodeGenerator` Klasse ist neben der Generierung für die AI auch für die Erstellung einer `level.js` Datei

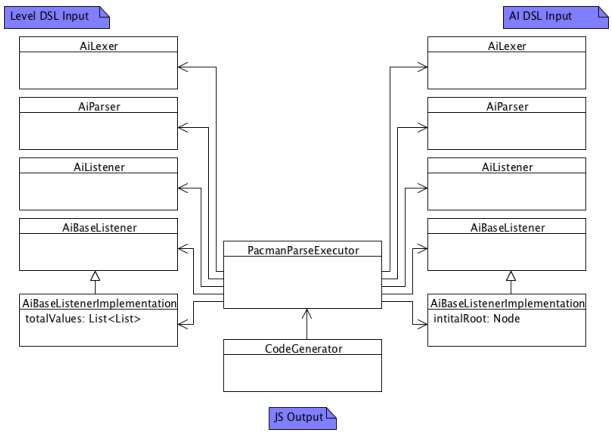

Abbildung 4. Beteiligte Klassen bei der Codegenerierung mit `ANTLR4`

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 `Gameboard`-Klasse importiert.

```
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],
      [3,1,1,1,1,1,1,1,1,1,3,1,1,1,1,1,3,1,1,2,3],
      [3,1,3,1,3,3,1,3,1,3,1,3,3,3,1,3,1,3,1,3],
      [3,1,3,1,1,1,1,1,1,1,1,1,1,1,3,1,3,1,3,1,3],
      [3,1,3,3,3,3,1,3,3,3,3,3,1,3,1,3,1,3,1,3],
      [3,1,1,1,1,1,1,1,1,1,3,1,1,1,3,1,1,1,1,1,3],
      [3,1,3,3,3,3,3,1,3,1,3,1,3,1,3,3,3,3,3,1,3],
      [3,1,3,3,1,1,1,1,1,2,1,1,1,1,1,1,1,3,1,3],
      [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],
      [3,1,3,3,1,1,1,1,1,1,1,1,1,3,1,3,1,3,1,3],
      [3,1,3,3,3,3,3,1,3,1,3,1,3,1,1,1,1,1,1,3],
      [3,1,1,1,1,1,1,1,3,1,3,1,3,1,3,3,3,3,3,1,3],
      [3,1,3,3,3,3,3,1,1,1,1,1,1,1,1,1,1,3,1,3],
      [3,1,3,1,1,1,1,3,3,3,3,3,1,3,3,3,1,1,1,3],
      [3,1,3,1,3,3,1,1,1,3,1,1,1,1,1,1,1,3,1,3],
      [3,1,3,1,3,3,3,1,3,1,3,1,3,3,3,3,3,3,1,3],
      [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. Die Spezifikation sieht vor, dass ein Array der Größe von exakt 20x20 erstellt werden muss. Eine beispielhafte Darstellung, die das Prinzip für den Aufbau des Spielfeldes veranschaulicht, ist in Listing I zu finden.

Zulässige Tokens sind als *Value* definiert. Dies sind nur die Zahlen *null, eins, zwei* und *drei* in beliebiger Reihenfolge. Als Trennsymbol einzelner Values ist allein das Semikolon als

Tabelle I
SPIELFELD-DEFINITION PER DSL

3	3	3	3	3
3	2	1	1	3
3	1	0	1	3
1	1	1	2	3
3	3	3	3	3

0	freier Weg
1	Punkt
2	Frucht
3	Mauer/Hindernis

Tabelle II
RICHTUNGSTOKENS DER AI DSL MIT EXEMPLARISCHER 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

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. Alle Zeilen zusammen bilden ein *field*.

2) Grammatik: Um den korrekten Aufbau des Spielfelds zu gewährleisten werden zunächst domänenspezifische Gültigkeitsregeln festgelegt. In unserem Fall sind diese durch Zahlen, Sonderzeichen und reguläre Ausdrücke repräsentiert. Die Anzahl der erlaubten Spalten ist nicht in der DSL spezifiziert. Sie wird in der `ListenerImplementation` überprüft. Die im vorherigen Absatz III-A1 festgelegten Regeln werden in der Datei `level.g4` (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`

B. AI DSL

Die durch Interpretation der AI DSL generierten *JavaScript*-Klassen ermittelt auf Basis der aktuellen Richtung eines Geistes dessen Richtung für den folgenden Spielzug. Eingabe und Ausgabewert einer AI 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 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,

Tabelle III
OPERATOREN MIT SYNTAXBEISPIEL, DIE DIE METHODE `checkMove()` ANSTEUERN

	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{ =>; <=; }`

welcher 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 demonstriert, der das Laufen in die entgegengesetzte Richtung ohne Abstraktionen der AI DSL darstellt.

```
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.

1) Überblick der Operatoren: Pac-Man AI DSLs beginnen mit ihrem Namen in runden Klammern. Innerhalb dieses Blocks befinden sich Operatoren, die eine Richtung als Rückgabewert liefern. Entscheidend für das Auswählen einer Richtung ist oftmals, ob diese frei ist. Wie in Absatz II-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, die eine Prüfung durch *Checkmove* auslösen.

Bei den Operatoren ist zu beachten, dass die `If-Free-` und `Filter-FreeN`-Operatoren jeweils eine einzelne Richtung als Rückgabewert besitzen, wohingegen der `Filter-Free`-Operator eine Liste von Richtungen zurückgibt. Der Begriff des Rückgabewertes im Kontext der DSL bildet sich bei der Ausführung mit `JavaScript`-Methoden ab.

Der `Filter-Free`-Operator darf 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 unerwar-

teten 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. Erkennbar 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 `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 der `Filter-Free` Operator mit dem `Random` Operator geschachtelt. Die DSL erlaubt beliebige Schachtelungstiefen. Die Syntax von Listing 6 ist deshalb ebenfalls zulässig.

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

Listing 6. Komplexere Verschachtelung in AI DSL

2) Aufbauen der Datenstruktur zur Codegenerierung: Die Verschachtelung von Operatoren 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

Die Listener Implementation baut eine Baumstruktur auf. Hierzu wird für jeden geparsten *Operator* eine Klasse erstellt, die von der abstrakten Klasse `Node` erbt.

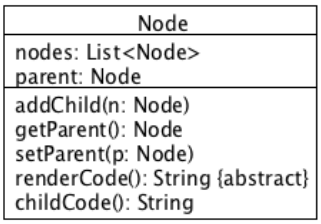

Abbildung 5. `Node`-Basisklasse als Halter der Datenstruktur zu Codegenerierung

Alle Unterklassen von `Node` (Abbildung 5) enthalten intern eine Listen-Struktur mit Kinder-Nodes. 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 als String konkateniert. Methode `childCode()` wird per Konvention in jeder `Node`-Klasse bei `renderCode()` aufgerufen. Eine `Node`-Klasse erzeugt

so beim Code-Generieren ebenfalls den *JavaScript*-Quellcode seiner Kinder. Beim Generieren einer AI Strategie wird `renderCode()` auf dem Wurzelknoten aufgerufen, der im Listener als Instanzvariable `initialNode` geführt ist (siehe Abbildung 4). Abbildung 6 zeigt die Baumstruktur zur DSL von Listing 5, Listing 8 den generierten *JavaScript*-Code.

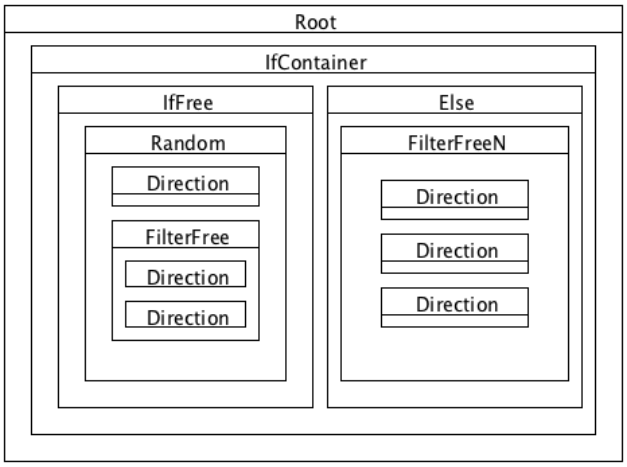

Abbildung 6. Schematische Darstellung der Datenstruktur beim Parsen einer AI DSL (Laufzeit)

```
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

3) Einbindung der generierten Klassen: Klasse `GhostQueries` dient als Mittler der generierten 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.

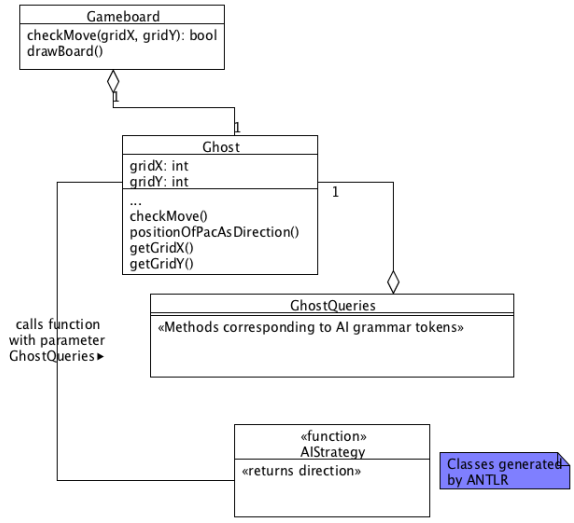

Abbildung 7. Aufruf der generierten Strategie-Methode durch Klasse `Ghost`

Tabelle V zeigt die Abbildung der Tokens der DSL auf *JavaScript*-Methoden der Klasse `GhostQueries`. In der Darstellung werden ebenfalls die Parameter- und Rückgabetypen dieser Methoden aufgeführt.

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>*)

4) JavaScript-Datenstrukturen: Neben der Abstraktion bezüglich der Richtungen (siehe Absatz III-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 Vereinigung 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`

5) Validierung: Bei der Implementierung der AI DSL hat sich bewährt, eine Baumstruktur in der entsprechende Listener Klasse aufzubauen um diese bei der Codegenerierung zu traversieren. Hierbei taucht allerdings das Problem auf, dass `ANTLR4` 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 `ANTLR4` empfehlenswert, um womöglich einen standardisierten 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 `Node` in `AiBaseListenerImplementation`

IV. Fazit und Ausblick

Ziel dieses Projekts war die Umsetzung eines funktionsfähigen Pac-Man-Klons. Ein Teil des Quellcodes sollte aus vorher 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 Level Designs beziehungsweise unterschiedliche Bewegungsstrategien (Erhöhung des Schwierigkeitsgrades) unabhängig vom Basiscode des Spiels zu generieren. Alle im Vorfeld gesetzten Kriterien wurden erfolgreich umgesetzt.

Ausbaubedarf besteht im Bereich der AI DSL insbesondere bei der Validierung (siehe Abschnitt III-B5). Sprechende Fehlermeldungen mit entsprechenden Zeilennummern erleichtern den Einsatz der DSL erheblich. 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 sicherzustellen, wäre im realen Anwendungsfall eine Notwendigkeit.

Das Erzeugen von Code mit Hilfe von DSLs scheint in dem gegebenen 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 des eigentlichen Spiels weitestgehend 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.