Dokumentation der Entwicklung des Spiels No Beer is oooch No Option (NBNO)

Tom Oberhauser
Beuth Hochschule für Technik Berlin
Medieninformatik Master
Software Engineering
Matrikelnummer: 859851

Robin Mehlitz
Beuth Hochschule für Technik Berlin
Medieninformatik Master
Software Engineering
Matrikelnummer: 857946

***Zusammenfassung*—Das ist ein toller abstract**

I. Wollen wir ein Datum einbinden

mds
September 17, 2014

II. Einleitung

Diese Dokumentation gibt einen Überblick über das Semesterprojekt im Modul Software-Engineering des Medieninformatik-Masterstudienganges. Ziel des Projekts ist es, eine anwendungsspezifische Sprache (domain-specific language) zu entwicklen und anhand dieser Code einer höheren Programmiersprache zu generieren. Der generierte Code soll hierbei eine zu implementierende Anwendung komplettieren und somit der Umgang mit anwendungsspezifischen Sprachen sowie der Codegenerierung praxisnah geübt und gefestigt werden.

Die Anforderung wird anhand eines Spiels und eines Levelgenerators umgesetzt. Der Levelgenerator übernimmt hierbei die Verarbeitung der anwedungsspezifischen Sprache.

A. Spielidee

Es soll ein 2D Spiel in Top-Down Perspektive entwickelt werden. Der Spielziel ist es, ein sich ständig leerendes Bier rechtzeitig durch den Besuch eines Spätis wieder aufzufüllen und bis zum Ende einer vorher definierten Zeit das Bier nicht leer werden zu lassen. Dazu bewegt der Spieler sich in einer blockbasierten Welt. Diese enthält sowohl Spätis zum nachfüllen des Bieres, als auch Gegner welche den Füllstand des Bieres oder die Geschwindigkeit des Spieler reduzieren können. In Abbildung **??** wird eine typische Szene des Spiels gezeigt. Die Statusbalken am unteren Bildrand geben Auskunft über den aktuellen Füllstand des Bieres (links) sowie die verbleibende Zeit (rechts), welche der Spieler überbrücken muss um das Level erfolgreich abzuschließen.

Späti erklären

B. Zweck der Codegenerierung

Das Spiel beinhaltet mehrere Level. Ein Level beinhaltet Informationen über die Anordnung der Blöcke auf der Karte, die Startpunkte des Spielers und eventueller Gegner, der Lauf- und Trinkgeschwindigkeit des Spielers, den Parametern der Gegner (Angriffsziel und Angrifswert) sowie der Zeitangabe, wie lange ein Level andauert. Dazu wurde eine anwendungsspezifische Sprache und ein Codegenerator entwickelt, auf welchen in Abschnitt **??** näher eingegangen wird.

Abbildung 1. Screenshot des Spiels

III. Aufbau der Infrastruktur

Das Spiel sowie auch der Levelgenerator werden mittels der Programmiersprache Java 8 entwickelt. Als Entwicklungsumgebung dient das IntelliJ IDEA System. Beide Programme werden jeweils in einem eigenen IntelliJ Projekt implementiert. Dadurch ist eine strikte Unabhängigkeit zwischen dem Levelgenerator und dem Spiel gewährleistet.
Um die Teamarbeit effizient zu gestalten, wird das Versionierungswerkzeug GIT eingesetzt. Es ermöglicht die Zusammenarbeit über die Plattform GitHub, auf der verschiedene nützliche Features angeboten werden. Sehr häufige Verwendung in diesem Projekt findet die Erzeugung und Zusammenführung von Pull Requests. Nachdem ein Entwickler ein Feature entwickelt oder einen Fehler behoben hat, wird so ein Pull Request erstellt. Ein weiteres Teammitglied überprüft diese Codeänderung und merged sie, bei positivem Ergebnis, auf die Master Branch.
Eine weitere wichtige GitHub Funktion besteht im Erstellen

von sogenannte Issues. Dies sind Repository interne Forumseinträge, die bestimmte Projektbezogene Themen behandeln sollen. Es können durch Labels Rubriken festgelegt werden, wie zum Beispiel "Bugöder "Question". So erhält das Projekt eine Struktur und Entscheidungen bzw. Hinweise die für das gesamte Team wichtig sind, werden an zentraler Stelle veröffentlicht und zur Diskussion freigegeben. Für das Bauen und die Abhängigkeiten Automatisierung von Bibliotheken und Frameworks wird im Spiele Projekt das Tool Maven verwendet. Im Generator Projekt dagegen besteht keine Notwendigkeit für ein solches Tool, weshalb hier Maven nicht eingesetzt wird.

IV. Technologien

Das Spiel sowie der Levelgenerator benutzen Bibliotheken und Frameworks. Dabei spielt die Slick2d Abhängigkeit im Spiele Projekt und die Antl Benutzung im Generator die größte Rolle. Diese beiden Frameworks werden im Folgenden näher erläutert.

A. Slick2D

Zur Entwicklung des Spiels wird die *Slick2D*[1] Bibliothek verwendet. Slick2D basiert auf der *Lightweight Java Game Library (LWJGL)*[2]. LWJGL ist eine sehr umfangreiche Bibliothek, welche viele für unseren Anwendungsfall benötigte Funktionalitäten bereitstellt. Dazu gehören die hardwarenahe Erzeugung grafischer Bildschirmausgabe via OpenGL, sowie die Verarbeitung von Tastatureingaben. Slick2D bildet eine Abstraktionsschicht über der LWJGL, speziell für den Anwendungsfall der 2D Spieleentwicklung. Dies ermöglicht eine effiziente Spieleentwicklung.

B. Antlr

Der Levelgenerator soll aus einer Textbasierten Datei eine Java Klasse erstellen. Dafür wird ein Parser, sowie ein Lexer benötigt. Um diese nicht selbst zu schreiben, wird das Werkzeug Antlr (v4) verwendet.
Antlr ist ein Quellcodegenerator, der Parser und Lexer in einer beliebigen Sprache erstellt. Die Dateien werden anhand einer vom Entwickler vorgegebenen Grammatik generiert.
Dabei ist wichtig zu verstehen, was ein Lexer und was ein Parser ist und tut.
Ein Lexer vollzieht die lexikalische Analyse. Das bedeutet, er unterteilt den eingegebenen Text in Einheiten, sogenannte Tokens. Dies können zum Beispiel Schlüsselwörter, Zahlen oder auch Zeichenketten sein. Aufgrund dieser Einteilung kann eine, zur Prüfung gegebene, Datei vollständig in diese Tokens zerlegt werden.
Der Parser hingegen erstellt aus der gegebenen Datei und den erzeugten Tokens eine Struktur, die im Fall von Antlr, als Syntaxbaum sichtbar gemacht wird. So kann jede beliebige Datei anhand dieses Syntaxbaums auf die korrekte Struktur hin überprüft werden.

[1]http://slick.ninjacave.com
[2]https://www.lwjgl.org

Die Kombination aus Lexer und Parser ermittelt die syntaktische Korrektheit des gegeben Textes. Sollte ein Fehler auftreten kann anhand der vergebenen Tokens die genaue Zeile und Spalte in der gegebenen Datei bestimmt werden. Ist die Datei vollständig in Tokens zerlegbar und stimmt die Struktur mit den Parser Regeln überein, gilt die Datei als korrekt und kann benutzt werden.
Um die Regeln für den Lexer und Parser festzulegen, schreibt der Entwickler eine, auf der Erweiterten Barkus-Nauer-Form(EBNF) basierende, Grammatik und generiert mittels Antlr hieraus die Lexer und Parser Dateien. Durch Antlr werden in diesem Vorgang ebenfalls Listener erstellt, die benutzt werden können, um an bestimmten Stellen beim Parsen der Datei Zugriff auf die aktuelle Struktur und Daten zu erhalten. So kann der Programmierer semantische Prüfungen der Daten durchführen und entsprechend auf korrekte oder fehlerhafte Eingaben reagieren.
Um mit Antlr Code generieren zu können, ist in IntelliJ IDEA das Antlr Plugin zu installieren. Durch Ausführen des Plugins auf der Grammatik wird der gewünschte Java Code erstellt.
Weitere Einstellungen sind über die Konfiguration des Plugins möglich. Hier wird beispielsweise festgelegt, wohin die Lexer und Parser Dateien generiert werden sollen oder in welcher Programmiersprache diese Dateien erstellt werden. Sind die Lexer und Parser in das Sourcecodeverzeichnis eingebunden, so können sie in einem Java Programm aufgerufen und ausgeführt werden.

C. Pixelart?

V. Aufbau der Spielarchitektur

A. Überblick

Slick2D stellt Techniken zur Umsetzung eines zustandsbasierten Spieles zur Verfügung. In Abbildung **??** werden die Zustände des Spiels und ihre Reihenfolge gezeigt.

Ein Spiel startet im `MainMenuState`. Hier kann der Spieler das Spiel entweder direkt beenden oder in den *LevelMenuState* wechseln. Der `LevelMenuState` bietet eine Auswahl aller zur Verfügung stehenden Levels an, welche dann im `PlayingState` gespielt werden können. Während des Spiels kann das Spiel durch Wechsel in den `PauseMenuState` pausiert werden. Nach erfolgreichem oder nicht erfolgreichem Abschluss eines Levels bietet der `GameFinishState` die Möglichkeit das Level zu wechseln, das Level erneut zu spielen oder das Spiel zu beenden.

Im FMC-Diagramm in Abbildung **??** wird die Bedeutung des `PlayingState` sowie der Aufbau eines Levels deutlich. Der `PlayingState` empfängt die Eingaben des Spielers und reicht diese an den `LevelController` weiter. Der `LevelController` kontrolliert und steuert die Bestandteile eines Levels.

B. Level

Jedes Level erbt von `AbstractLevel`. In Abbildung **??** wird der Aufbau eines Levels deutlich. Ein Level besteht

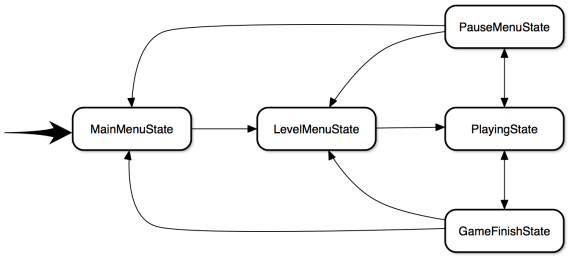

Abbildung 2. Übersicht der Zustände des Spiels

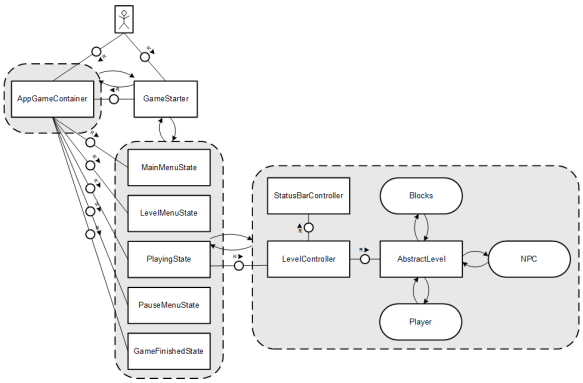

Abbildung 3. FMC Diagramm des Spiels

aus einer Menge an `NPC`-Objekten, welche die Gegner repräsentieren. Außerdem beinhaltet es ein `Player`-Objekt sowie die Menge der Blöcke(`Block`), also der Karte des Levels. Eine Übersicht über alle verwendbaren Spielobjekte wird in Abbildung **??** gegeben. Level sind komplexe Datentypen und verstehen sich als Komposition der, in Abbildung **??** gezeigten, spielspezifischen `GameObject` Datentypen. Zur Vermeidung von Redundanz und zur klaren Trennung zwischen Modell und Controller wird die Steuerung der Elemente eines Levels durch den `LevelController` übernommen. Durch diese Trennung eignen sich Implementierungen von `AbstractLevel` sehr gut zur Erzeugung durch eine anwendungsspezifische Sprache.

C. LevelController

- Initialisiert ein Level zur Benutzung (Blockpositionen ...) - Erhält die Eingaben des Nutzers und wandelt diese in Änderungen des Levels um - steuert die KI

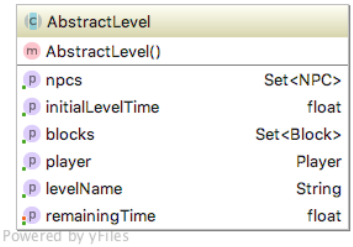

Abbildung 4. UML Darstellung der Klasse AbstractLevel

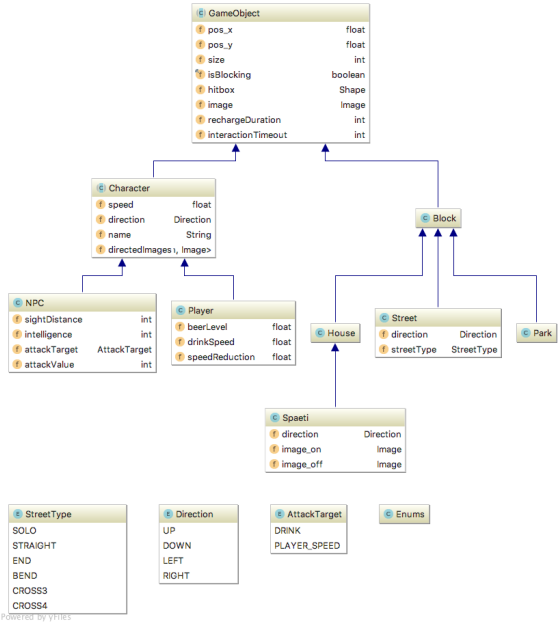

Abbildung 5. Spielobjekte

stubs

D. Highlights der Implementierung

Ideen: - LevelMenu - Kollisionskontrolle - KI

VI. Levelerzeugung

In diesem Abschnitt wird das Projekt des Levelgenerators näher erläutert. Die Schwerpunkte liegen dabei auf dem Level Metamodell sowie auf der Codegenerierung aus einer gegebenen Leveldatei.
Dabei wird erläutert wie so eine textuelle Leveldatei aussehen muss, um anschließend die beschreibende Domain Specific Language (DSL) verständlich darzustellen.
Abschließend wird in diesem Kapitel der Levelgenerator vorgestellt, mit Augenmerk auf seinen Aufbau und seine Funktionsweise.

A. Aufbau einer nbno-Datei

Um ein Level mit dem Levelgenerator zu erzeugen, muss eine textuelle Datei im NBNO Format an den Generator übergeben werden. Eine solche Datei besitzt eine durch die DSL fest vorgegebene Struktur. Diese wird durch die benötigten Levelattribute und Eigenschaften bestimmt.
Zunächst enthält ein Level einen Namen und Levelkonfigurationen (vgl. Listing **??**). Der bisherige Projektstand benötigt als Leveleigenschaft nur die Spielzeit des Levels (entspricht dem rechten Balken des Spiels (vgl. Abb. **??**)).
Der Spieler selbst erhält Eigenschaften die seine Geschwindigkeit, seine Trinkgeschwindigkeit und das initiale Bierlevel einstellen. Wobei das Bierlevel dem linken Balken entspricht (vgl. Abb. **??**) und die Trinkgeschwindigkeit die Geschwindigkeit der Leerung dieses Balkens bestimmt.

Listing 1. Level Metadaten und Spieler

```
/levelName: TestName
levelConfiguration {
    levelTime: 40
}
player {
    speed: 50,
    drinkSpeed: 6,
    beerLevel: 30
}
```

An diesem Auszug ist der Aufbau einer nbno-Datei bereits zu erkennen. Bestimmte Attribute besitzen Eigenschaften die angelehnt an das JSON-Format in geschweiften Klammern aufgelistet werden. Separiert werden diese Eigenschaften durch Kommata.
Dies ist auch im nächsten Auszug gut zu erkennen. Es werden die Gegnerklassen angelegt und mit bestimmten Eigenschaften ausgestattet (vgl. Listing **??**). Es wird ein Name an die Klassen verteilt, durch den zur Laufzeit eine passende Darstellung des Charakters erreicht wird. Weitere Eigenschaften sind die Geschwindigkeit, das Angriffsziel und der Schaden des Gegners. Die künstliche Intelligenz des Gegenspielers wird hier ebenfalls vergeben. Dabei ist die erste Zahl die Sichtweite des Charakters und die Zweite die Wahrscheinlichkeitsangabe, wie oft der Gegner seine Richtung ändert (Die KI wurde genauer erläutert in **??**).

Listing 2. Gegnerklassen

```
/enemies {
    1 {
        name: Schnorrer,
        speed: 40,
        attackTarget: DRINK,
        damage: 30,
        ki: (10|30)
    },
    2 {
        name: Polizist,
        speed: 25,
        attackTarget: PLAYER_SPEED,
        damage: 70,
        ki: (24|40)
    }
}
```

Wichtig sind auch die Ziffern vor den eigentlichen Gegnereigenschaften. Sie stellen die Repräsentation auf der Map dar, die in der Leveldatei angelegt wird (vgl. Listing **??**). Der Ausschnitt zeigt einige Zeilen einer Map Konfiguration. Die Map wird durch Buchstaben und Ziffern repräsentiert, wobei jedes Zeichen dabei eine eigene Bedeutung hat. Die zuvor erwähnten Ziffern der Gegner stellen hier die Startpositionen eines Gegners der entsprechenden Klasse dar. Auf diese Weise ist es möglich, von einer Klasse mehrere Feinde starten zu lassen.

Das ”Xïst die Startposition des Spielers. Ein ”Hßteht für ein Haus, ein ”P”für ein Block Park und das SSïst das Symbol für eine Straße. Die Spätis werden per Pfeilspitze dargestellt. Je nachdem in welche Richtung die Spitze zeigt, öffnet sich der Späti im Spiel und kann vom Spieler angelaufen werden. In diesem Beispiel bedeutet demnach das ”V”, dass der Späti sich nach unten öffnet.

Listing 3. Level Map

```
/map {
    H,H,H,H,H,H,H,H,H,H,S,S,S,S,S,S,S,S,S
        ,S,S,S,S,S,S
    P,P,S,S,S,S,S,S,2,H,S,S,S,S,S,S,1,S,S
        ,S,S,S,S,S,S
    P,P,S,S,S,S,S,S,S,H,S,S,S,S,H,H,H,S,S
        ,S,S,S,S,S,S
    H,H,V,H,H,H,S,S,S,H,S,S,S,S,H,H,H,S,S
        ,S,S,S,S,S,S
  [...]
    S,S,S,S,S,S,S,S,S,S,S,X,S,S,S,S,S,S,S
        ,S,S,S,S,S,S
}
```

B. Metamodell eines Levels

Das Metamodell dieses Projekts beschreibt die Struktur und den Aufbau eines Levels des Spiels. Das Modell ist in EBNF verfasst und generiert durch Antlr die Compiler für die gewünschte NBNO-DSL.
Die DSL muss immer gleich aufgebaut sein (vgl. Listing **??**). Es muss mit dem Levelnamen begonnen werden und anschließend die Levelkonfigurationen festgelegt werden. Danach folgt der Spieler und die Gegnerklassen. Abschließend muss ein Spielfeld definiert werden.
Um Wiederholungen zu vermeiden, sind Lexerregeln aufgestellt, die Alphabete und Trenner festgelegen. Außerdem ist eingestellt, dass alle Leerzeichen und Abstände beim Parsen übersprungen werden.

Listing 4. Aufbau und Tokens der DSL

```
/file : levelName levelConfigs player
   enemies map EOF;

// helper definitions
ALPHABET: ('a'..'z' | 'A'..'Z')+;
DIGITS : [0-9]+;
ObjectBeginn:'{';
ObjectEnd: '}';
Separator: ',';
WS : [ \t\n\r]+ -> skip ; // skip
   witepsaces tabs and linebreaks
```

Die Definitionen der oben genannten Attribute ähnelt sich sehr, weshalb hier beispielhaft der Ausschnitt der Spielerdefinition erklärt wird. Ein Spieler soll in der DSL als ”playerängeben sein, was hier als Schlüsselwort festgelegt wird. Anschließend folgen die geschweiften Klammern. Die Parserregel legt fest, dass ein Spieler Attribute innerhalb

der geschweiften Klammern besitzt. Diese Attribute sind durch Kommata getrennt und beginnen immer mit ihrem Schlüsselwort und einem Doppelpunkt dahinter.
Der Wert des Attributs ist durch einen Verweis auf eines der möglichen Alphabete angegeben. Durch diese strukturelle Einschränkung werden Listener durch Antlr für jeden Attributwert erstellt. Dadurch ist es möglich beim Parsen im Syntaxbaum direkt an diesem Attribut änzuhaltenünd sich den Wert ausgeben zu lassen. Dies erleichtert die semantische Prüfung im Levelgenerator. Die andern Schlüsselattribute der NBNO-DSL sind nach dem gleichen Schema aufgebaut.

Listing 5.

```
///the player and its attributes
player: 'player' ObjectBeginn
    playerAttributes ObjectEnd ;
playerAttributes:  speed Separator
    drinkSpeed Separator beerLevel;
speed: 'speed:' speedValue;
speedValue: DIGITS;
drinkSpeed: 'drinkSpeed:' drinkSpeedValue
    ;
drinkSpeedValue: DIGITS;
beerLevel: 'beerLevel:' beerLevelValue;
beerLevelValue: DIGITS;
```

C. Levelgenerator

Als Modellinterpretierer und damit Levelgenerator fungiert eine Java Klasse ”LevelGenerator.java”. In ihr werden die durch Antlr erzeugten Lexer und Parser ausgeführt, eine semantische Prüfung der übergebenen Daten vollzogen und abschließend eine Java Leveldatei erzeugt.
Ein sehr wichtiger Bestandteil des Generators ist das Errorhandling. Die Antlr Parser und Lexer werfen standardisiert keine Fehler sondern geben lediglich auf der Konsole eine Nachricht aus, das ein Fehler beim Parsen aufgetreten ist. Gewünscht ist jedoch, dass der Levelgenerator fehlerhaft abbricht bei jeglicher Art eines Syntaxfehlers. Aus diesem Grund wird der Antlr ErrorListener beerbt und der eigene ”LevelErrorListener”dem Parser und Lexer im Levelgenerator übergeben. Er überschreibt die bei einem Syntaxfehler aufgerufene Methode und kann so eine Exception werfen, die den Levelerzeugungsvorgang abbricht.
Der Generator liest zu beginn eine .nbno-Datei aus dem ”levelfiles”Projektverzeichnis ein, deren Dateiname über die Konsole übergeben wird. Dieses File wird dann geparst und auf Fehlerhafte Einträge oder Strukturen geprüft. Entspricht die Datei der Grammatik, wird eine Java Klasse mit dem in der Leveldatei spezifizierten Namen angelegt und gespeichert. Durch das Metamodell ist die syntaktische Analyse bereits abgedeckt, weshalb im Generator die semantische Prüfung fokussiert wird. Dafür werden die erzeugten Listener Methoden für die Attribute und Attributwerte überschrieben. Im folgenden Codeauszug (vgl. Listing **??**) wird der Levelname auf Länge (maximal 100 Zeichen) geprüft. Des Weiteren wird bei Korrektheit das erste Zeichen zu einem Großbuchstaben gewandelt, so dass die Javaklasse den Javakonventionen entspricht. Ist der Levelname zu lang, so wird ein Fehler geworfen und der Generierungsvorgang abgebrochen.

Listing 6. LevelGenerator: Levelnamensprüfung

```
@Override
public void enterLevelNameValue(
    LevelGrammarParser.
    LevelNameValueContext ctx) {
  final String name = ctx.getText();
  if(name.length() > 100){
    throw new IllegalArgumentException(
      GeneratorUtils.
          getFormattedErrorMessage("Level
          name is to long!", ctx.getStart
          ()));
  }
  levelName = name.substring(0,1).
     toUpperCase() + name.substring(1);
}
```

Die weiteren Attribute und Eigenschaften werden auf die gleiche Art und Weise geprüft. Beispielsweise wird die Geschwindigkeit des Spielers auf einen Wertebereich zwischen 0 und 100 in der überschriebenen Methode *enterSpeedValue()* geprüft. So ist gewährleistet das am Ende alle Attribute auch die inhaltlich korrekten Werte enthalten.
Eine weitere Interessante Prüfung geschieht bei der Spielkartenanalyse (vgl. Listing **??**). Die Map muss immer genau 20 Zeilen und 25 Spalten enthalten. Wobei Spalten in diesem Projekt als Blöcke interpretiert werden. Ein Block wird im Spiel zu einem 32x32 Pixel Block umgerechnet. Auf diese Weise besitzt das Spielfeld immer die gleiche Größe. Die Leveldateien müssten so auch nicht bearbeitet werden, falls die Größen im Spiel einmal skaliert werden sollten.

Listing 7. Codegenerator: Map Prüfung

```
@Override
public void enterMapValue(
    LevelGrammarParser.MapValueContext ctx
    ) {
  final int maxRows = 20;
  final int maxBlocks= 25;
  if(ctx.row().size() != maxRows){
      throw new IllegalArgumentException(
         "The map must have 20 rows. Not
         more or less.");
  }
  int rowCounter = 0;
  int blockCounter = 0;
  for(LevelGrammarParser.RowContext row :
      ctx.row()){
      if(row.block().size() != maxBlocks)
         {
          throw new
             IllegalArgumentException("
             Each row in the map must
             contain " + maxBlocks + "
             Blocks. "
                  + "In Line " + row.
                     getStart().getLine()
                      + "is an incorrect
                     number of blocks.");
      }
      for(LevelGrammarParser.BlockContext
          block : row.block()){
          final int xPos = blockCounter;
          final int yPos = rowCounter;
          final String blockVal = block.
             getText();
          switch (blockVal){
            case GeneratorUtils.HOUSE:
               generateHouseString(xPos,
                  yPos);
               break;
            [...]
            default:        // wenn keins
                der oberen matched, muss
               es eine Zahl sein
               generateEnemyString(
                  Integer.parseInt(
                  blockVal), xPos, yPos);
               generateStreetString(xPos,
                   yPos);
               break;
          }
          blockCounter ++;
      }
      blockCounter = 0;
      rowCounter ++;
  }
}
```

Ein weiteres Detail wird in dieser Methode ebenfalls sichtbar. Beim Durchlaufen der einzelnen Blöcke der Karte wird für jeden Block ein String angelegt. Diese Strings werden in einer Liste von Strings gesammelt.
Alle variablen Attribute, wie der Spieler oder die Gegnerklassen, werden als Strings erzeugt. Wobei unter jedem Gegner und dem Spieler jeweils noch ein Straßenblock erzeugt werden muss. All diese erstellten Zeichenketten werden in Listen gespeichert.
Die Codegenerierung erfolgt am Ende durch das Zusammensetzen aller gespeicherten Strings. Alle Konstanten Angaben, wie der Import von Datentypen oder das Überschreiben des Konstruktors, werden zuvor in der Methode *generateLevelClass()* mit den anderen variablen Strings in einer Liste zusammengeführt. Diese enthält dann die komplette Levelklasse als Stringrepräsentation.
Über die final entstandene Stringliste wird in der *writeFile()* Methode iteriert und für jeden String eine eigene Zeile in die neue Javaklasse geschrieben.

VII. Fazit

VIII. Ausblick

- mehr verschiedene Blöcke - dynamische Spielfeldgröße (bei anpassbarem Fenster mit Scrolling)

Literatur

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.