

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 Mehltitz
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 entwickeln 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 anwendungsspezifischen Sprache.

A. Spielidee

Späti
er-
klären

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 1 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.

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 Angriffswert) sowie der Zeitangabe,



Abbildung 1. Screenshot des Spiels

wie lange ein Level andauert. Dazu wurde eine anwendungsspezifische Sprache und ein Codegenerator entwickelt, auf welchen in Abschnitt VI näher eingegangen wird.

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 Antlr 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.

Die Kombination aus Lexer und Parser ermittelt die syntaktische Korrektheit des gegebenen 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 Backus-Naur-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 Spiels zur Verfügung. In Abbildung 2 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 3 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 4 wird der Aufbau eines Levels deutlich. Ein Level besteht

¹<http://slick.ninjacave.com>

²<https://www.lwjgll.org>

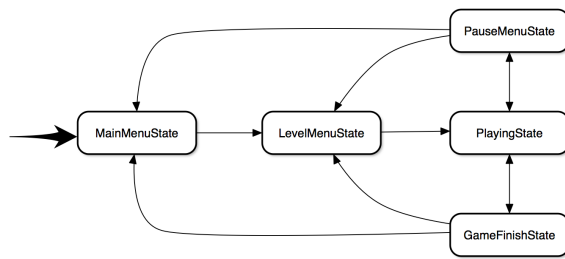


Abbildung 2. Übersicht der Zustände des Spiels

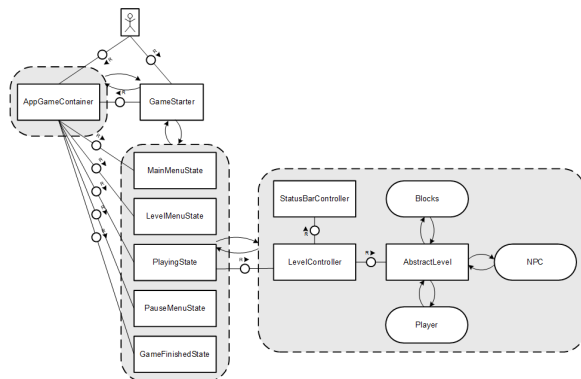


Abbildung 3. FMC Diagramm des Spiels

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 5 gegeben. Level sind komplexe Datentypen und verstehen sich als Komposition der, in Abbildung 5 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

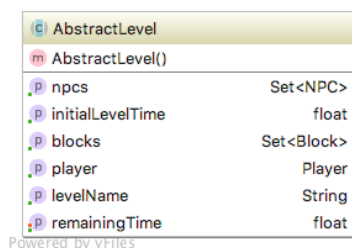


Abbildung 4. UML Darstellung der Klasse AbstractLevel

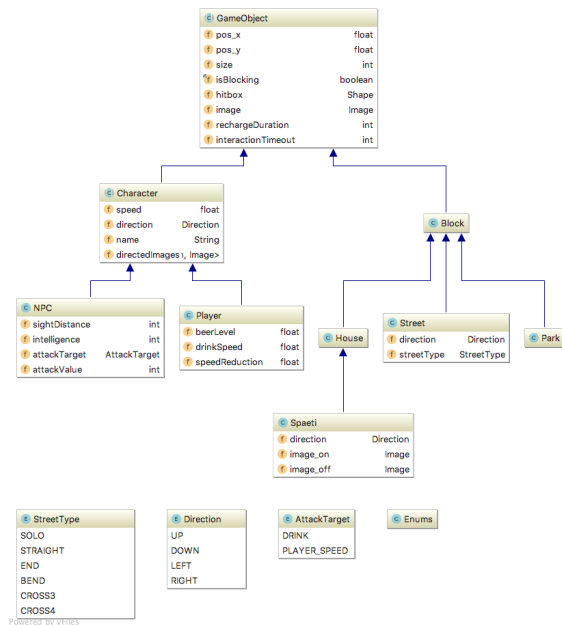


Abbildung 5. Spielobjekte

Änderungen des Levels um - steuert die KI

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 1). Der bisherige Projektstand benötigt als Leveleigenschaft nur die Spielzeit des Levels (entspricht dem rechten Balken des Spiels (vgl. Abb. 1)).

Der Spieler selbst erhält Eigenschaften die seine Geschwindigkeit, seine Trinkgeschwindigkeit und das initiale Bierlevel einstellen. Wobei das Bierlevel dem linken Balken entspricht (vgl. Abb. 1) und die Trinkgeschwindigkeit die Geschwindigkeit der Leerung dieses Balkens bestimmt.

Listing 1. Level Metadaten und Spieler

```

1 levelName: TestName
2 levelConfiguration {
3     levelTime: 40
4 }
5 player {
6     speed: 50,
7     drinkSpeed: 6,
8     beerLevel: 30
9 }

```

An diesem Auszug ist der Aufbau einer nbno-Datei bereits zu erkennen. Bestimmte Attribute besitzen Eigenschaften die³ angelehnt an das JSON-Format in geschweiften Klammern aufgelistet werden. Separiert werden diese Eigenschaften⁴ durch Kommata.

Dies ist auch im nächsten Auszug gut zu erkennen. Es⁵ werden die Gegnerklassen angelegt und mit bestimmten Eigenschaften ausgestattet (vgl. Listing 2). Es wird ein⁶ Name an die Klassen verteilt, durch den zur Laufzeit eine⁷ passende Darstellung des Charakters erreicht wird. Weitere Eigenschaften sind die Geschwindigkeit, das Angriffsziel⁸ 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

```

1 enemies {
2     1 {
3         name: Schnorrer,
4         speed: 40,
5         attackTarget: DRINK,
6         damage: 30,
7         ki: (10|30)
8     },
9     2 {
10        name: Polizist,
11        speed: 25,
12        attackTarget: PLAYER_SPEED,
13        damage: 70,
14        ki: (24|40)
15    }
16 }

```

Wichtig sind auch die Ziffern vor den eigentlichen Geg-⁸ nereigenschaften. Sie stellen die Repräsentation auf der Map⁹ dar, die in der Leveldatei angelegt wird (vgl. Listing 3). 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 "X" ist die Startposition des Spielers. Ein "H" steht für ein Haus, ein "P" für ein Block Park und das "S" ist das Symbol für eine Straße. Die Späti 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

```

1 map {
2     H, H, H, H, H, H, H, H, H, H, S, S, S, S, S, S, S, S, S, S,
3         S, S, S, S, S, S, S,
4     P, P, S, S, S, S, S, S, 2, H, S, S, S, S, S, S, 1, S, S,
5         S, S, S, S, S, S, S,
6     P, P, S, S, S, S, S, S, S, H, S, S, S, S, H, H, H, S, S,
7         S, S, S, S, S, S, S,
8     H, H, V, H, H, H, S, S, S, H, S, S, S, S, H, H, H, S, S,
9         S, S, S, S, S, S, S,
10    [...]
11    S, S, S, S, S, S, S, S, S, S, S, S, X, S, S, S, S, S, S, S,
12        S, S, S, S, S, S, S, S
13 }

```

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 festlegen. Außerdem ist eingestellt, dass alle Leerzeichen und Abstände beim Parsen übersprungen werden.

Listing 4. Aufbau und Tokens der DSL

```

1 file : levelName levelConfigs player
2     enemies map EOF;
3
4 // helper definitions
5 ALPHABET: ('a'..'z' | 'A'..'Z')+;
6 DIGITS : [0-9]+;
7 ObjectBegin: '{';
8 ObjectEnd: '}';
9 Separator: ',';
10 WS : [ \t\n\r ]+ -> skip ; // skip
11     whitespaces 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 "player" angegeben 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 anzuhalten und sich den Wert ausgeben zu lassen. Dies erleichtert die semantische Prüfung im Levelgenerator. Die anderen Schlüsselattribute der NBNO-DSL sind nach dem gleichen Schema aufgebaut.

Listing 5.

```
1 //the player and its attributes
2 player: 'player' ObjectBeginn
   playerAttributes ObjectEnd ;
3 playerAttributes: speed Separator
   drinkSpeed Separator beerLevel;
4 speed: 'speed:' speedValue;
5 speedValue: DIGITS;
6 drinkSpeed: 'drinkSpeed:' drinkSpeedValue
   ;
7 drinkSpeedValue: DIGITS;
8 beerLevel: 'beerLevel:' beerLevelValue;
9 beerLevelValue: DIGITS;
```

- unterteilung in Lexer und Parser regeln - lexer Regeln sind alphabete - parser regeln legen struktur fest - feste reihenfolge!
- gewisser aufbau durch klammern etc. - alle whitespaces werden geskipped - jedes attribut besitzt namen und eine extra Regel,terminal,non-terminal, auf das ich durch antlr's listener Erzeugung direkt zugreifen kann

C. Codegenerator

- modellinterpret - Fehler Handling! (Extra n exception handler beigelegt) - Semantik Prüfung - level erzeugung per string maps - methoden zur übersichtlichkeit und entkopplung aus anderen methoden zum erzeugen von strings - stream als java datei speichern

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 L^AT_EX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.