Comic Generierung - PJ 2

Fenja Harbke

Sommersemester 2015

Ziel der Masterarbeit ist eine Comic-DSL. Deren Input soll Panel für Panel einen Comic nach der in der DSL spezifizierten Beschreibung generieren.
Hierfür muss in erster Linie eine DSL umgesetzt werden. Die zu entwerfenden Sprache soll den Comic-Jargon aufgreifen, möglichst einfach zu erlernen sein und die dem Comic eigene Hierarchie von Seiten und Panels widerspiegeln.
Die Zielsprache in Java soll über eine Schnittstelle den ComicGenerator ansprechen und so das Generieren der Panel anstoßen. Das umfangreiche Erstellen einer im Panel darzustellenden Szene ist nicht Teil dieser Ausarbeitung.

1 Einführung

Zur Umsetzung einer Comic-DSL muss im ersten Schritt ein passendes Tool gefunden werden. Die zwei bedeutendsten Vertreter sind hier Xtext [17] und MPS [10], deren Unterschiede es gilt herauszufinden und abzuwägen.
Nach der Entscheidung für eine der so genannten Language Workbenches gilt es, die DSL umzusetzen. Dies setzt eine Einarbeitung in das Tool Language Workbench voraus, sowie den vorherigen Entwurf der finalen Struktur der abstrakten Syntax.

Zeitgleich soll das System *ComicGenerator* weiterentwickelt werden und eine Schnittstelle für den aus der DSL generierten Code bieten. Hierfür ist eine prototypische Implementierung desselben angedacht, in welcher die Grundfunktionalitäten vorhanden sind und der noch ausstehende Aufbau der Szene in einem Panel eingepasst werden kann.

1.1 Zielsetzung PJ2

Gesetzes Ziel für PJ2 ist die prototypische Implementierung der Comic-DSL, sowie das Einbinden des ComicGenerators. Hierfür müssen zunächst Entscheidungen für zu verwendende Software und Vorgehensweisen getroffen und das Erscheinungsbild der fertigen DSL entworfen werden. Immer im Vordergrund steht hierbei die Erweiterbarkeit, sodass es möglich sein sollte, zunächst Grundfunktionalitäten umzusetzen und im nächsten Schritt mehr Funktionalitäten und Regeln einzubauen.
Am Ende von PJ2 soll ein Comic mit mehreren Seiten und Panels beschrieben und mit Hilfe des Comic-Generators zu HTML exportiert werden, sodass die fertige Panel-Aufteilung des Comics im Browser angesehen werden kann.

2 Comic-DSL

Die Comic-DSL ist zunächst eine DSL der Domäne Comic. Entsprechend sollte soweit möglich Fachjargon verwendet werden und die comictypische Struktur wiedererkennbar sein.
Das Framework, mit dem die DSL erstellt wird, muss gewissen Anforderungen entsprechen und die erdachte Comic-Struktur damit umsetzbar sein.

2.1 Domain Specific Language

Eine DSL ist eine spezialisierte und problem-orientierte Sprache [11, S.4]. Eine so genannte *externe DSL* ist eine explizit entworfene Sprache zu einem Problembereich, der Domäne. Unter Verwendung des domänenspezifischen Jargons wird in konkreter Syntax die abstrakte Syntax der Domäne beschrieben. Letztere wird durch einen Graphen oder Baum repräsentiert, während die konkrete

Syntax menschenlesbar ist und vom User modifiziert wird, um das unterliegende abstrakte Modell anzupassen.

2.1.1 Aufbau einer DSL

Jede DSL hat drei Grundelemente, deren Gestaltung wichtiges Entscheidungskriterium bei der Auswahl eines Frameworks ist. Die konkrete Umsetzung ist dann je nach Werkzeug verschieden.

- Schema - Abstrakte Syntax, Beschreibung der Elemente einer Sprache.
- Editor - Quellsprache, in welcher der User den Code verfasst.
- Generator - Erstellt Code in der Zielsprache (z.B. Java).

2.1.2 Language Workbench (LWB)

Eine Language Workbench ist ein System zur Erstellung einer DSL. Sie ermöglicht die Spezifizierung der Grundelemente einer DSL, also den logischen Aufbau des Schemas, die Syntax des Editors und nach welchen Regeln der Generator aus dem Input Quellcode generieren soll.
Auf verschiedene Ansätze bei Language Workbenches wird noch einmal in Abschnitt 2.2.4 eingegangen.

2.2 Frameworks

Wie in 2.2.4 beschrieben wird gibt es zwei Hauptansätze für Language Workbenches. Die für diese Arbeit zur Auswahl stehenden Alternativen sind jeweils eine Textual bzw. Projectional Language Workbench. Gesucht wird das Tool, um die Comic-DSL umzusetzen.
Vor der Gegenüberstellung gilt es, Anforderungen an die DSL und somit die LWB festzuhalten und die zur Auswahl stehenden Tools entsprechend daraufhin zu untersuchen.

- **Zielsprache Java**, da der unterliegende Comic Generator durch die Verwendung von LibGDX in Java geschrieben ist.
- **Erweiterbarkeit** wird angestrebt, da im Rahmen der Masterarbeit ggf. nicht alle Möglichkeiten ausgeschöpft werden können und bereits jetzt viele Ideen für Erweiterungen vorliegen.
- Eine Einarbeitung in das Tool soll im Rahmen von Projekt 2 möglich sein, sodass am Ende des Semesters eine einsetzbare DSL erhalten wird.

Bei Xtext handelt es sich um ein *Language Development Framework* zur Erstellung von DSLs und GPLs, während MPS als Language Workbench auf die Entwicklung von DSLs spezialisiert ist.

2.2.1 Xtext

Xtext ist ein 2006 entstandenes Open-Source-Framework, das seit 2008 im Rahmen des Eclipse Modeling Project weiterentwickelt wird. Die aktuelle Version ist 2.7.0 (September 2014). Mit Hilfe von Xtext können in einer Eclipse-basierten Umgebung DSLs und GPLs erstellt werden. Hierbei sorgt Xtext nicht nur, wie andere Parsergeneratoren, für einen Parser, sondern auch für folgende Bestandteile einer Sprache:

- **Parser** auf Basis einer Beschreibung der Sprache
- Modellierung des **semantischen Modells**
- Bau des **Abstrakten Syntax Baumes**
- **Eclipse-Plugin** zur Nutzung in einer Eclipse-Entwicklungsumgebung

[17][13, S.23f]

Vorteile von Xtext sind die einfache Beschreibung der Grammatik, das Eclipse-Plugin, mit dessen Hilfe sich die Sprache leicht nutzen lässt, sowie der Umstand, dass sich die Funktionalitäten von Xtext leicht erweitern lassen.

Nachteile sind die umständliche Erweiterung einer bereits erstellten Sprache, sowie die damit verknüpften Abhängigkeiten, sowie nicht zuletzt die Festlegung auf Eclipse [13, S.50f].

2.2.2 MPS

Vorteile von MPS sind die modulare Spracherweiterung und Java als Grundlage für dieselben, sowie der Umstand, dass keine Grammatik für die Spracherweiterung entwickelt werden muss, da der abstrakte Syntaxbaum (AST) direkt bei der Eingabe erzeugt wird.

Nachteile Durch fehlende textuelle Repräsentationen ist man an MPS als Entwicklungsumgebung gebunden, die Versionierung erfolgt in XML-Form der Modelle und als Zielsprache steht bisher nur Java zur Verfügung [13, S.51].

2.2.3 Alternativen

Martin Fowler beschreibt in *Language Workbenches: The Killer-App for Domain Specific Languages?* verschiedene LWBs, die zum Zeitpunkt der Arbeit (2005) alle noch am Anfang ihrer Entwicklung standen [6].

Intentional Software baut auf dem Projekt *Intentional Programming* auf, das unter der Führung von Charles Simonyi bei Microsoft Research lief. Sie ermöglicht das Schreiben und Ansehen von Programmen in verschiedenen Notationen und erlaubt ebenso die einfache Integration von domain-specific und general-purpose Programmen[2].

Software Factories will die Softwareentwicklung industrialisieren und bietet eine Produktlinie zur Einbettung von DSL-Pattern, -Frameworks und -Anleitungen an, um die Erstellung von DSLs zur individuellen und schnelleren Problemlösung zu erleichtern [1].

Model Driven Architecture (MDA) ist ein Ansatz zur Entwicklung von Software Systemen. Er enthält Richtlinien zur Strukturierung von Spezifikationen, die als *models* ausgedrückt werden. MDA verfolgt Ansätze die ähnlich sind zu denen von Fowlers Ansprüchen an eine Language Workbench. Die Einführung erfolgte 2001 durch die Object Management Group (OMG).

2.2.4 TLWB und PLWB

Merkle unterscheidet in seiner Arbeit [12] zwischen Textual Language Workbenches (TLWB), deren Hauptvertreter *Xtext* ist und als weitere Vertreter **TEF (Textual Editing Framework)**, **TCS (Textual Concrete Syntax)** und **EMFText**. Die Projectional Language Workbenches (PLWB) haben bisher nur *MPS* als nennenswerten Vertreter, die Entstehung von weiteren PLWBs wird allerdings prognostiziert [12].

2.2.5 Abwägung

Für eine Abwägung seien zunächst die zuvor festgelegten Anforderungen heran gezogen.
Java als Zielsprache wird von beiden Frameworks angeboten.
Die Erweiterbarkeit der DSL ist bei Xtext in der Form wie bei MPS nicht möglich und gestaltet sich umständlich, wenn überhaupt möglich.
Über den Umfang und die Einarbeitungszeit kann vor Beginn der Arbeit keine Aussage getätigt werden.

Eine Entscheidung für Xtext könnte bei einer vorliegenden Eclipse-Präferenz gewählt werden. Dies entfällt, da bevorzugt IntelliJ genutzt wird. Beide Frameworks fordern eine Festlegung bezüglich der Entwicklungsumgebung, sodass dieser Aspekt keinen Unterschied macht. Die Problematik von MPS bei der Versionskontrolle ist vorerst unwirksam, da zumindest für die Dauer der Masterarbeit nur ein Entwickler an der DSL arbeiten wird.

2.3 Realisierung mit MPS

DSL Design In ihrer Arbeit tragen Pane et al. Richtlinien für das Design neuer Programmier-Systeme zusammen. Schwerpunkt sind hierbei zwar Programmier-Neulinge, aber jede neue Programmiersprache profitiert davon, die Richtlinien so weit möglich einzuhalten [14, S.5].

```
< comic name = " Test " font_size = 12 project_path = " /usr/local " >
    < character name = " Fenja " src = " /usr/local/models/fenja.g3db " />

< page grid = 3 >
  < panel border = " false " width = " 1_3 " >
        < Fenja />
    << ... >>
  </ panel >
</ page >
</ comic >
```

Figure 1: Auszug Comic-DSL

- Sichtbarkeit des System-Status
- Übereinstimmung des Systems und der echten Welt
- Kontrolle und Freiheit des Users
- Konsistenz und Standards
- Wiedererkennung statt Erinnerung
- Ästhetik und minimalistisches Design
- User unterstützen bei der Wiedererkennung, Diagnose und Entdeckung von Fehlern
- Hilfe und Dokumentation

Einen Großteil der Arbeit nimmt die Beschaffenheit des in MPS erstellten Editors ab, indem unter anderem keine Fehler möglich sind und Vervollständigungen den User unterstützen. Auch Syntax Highlighting kann mit MPS umgesetzt werden und der Aufbau der Sprache bedient sich stark dem Comic-Jargon und durch das Aufgreifen der Comic-Hierarchie ist die Bedienung intuitiv.
Die DSL wurde zusätzlich durch den Gebrauch von spitzen Klammern an HTML angelehnt. So wird der Einstieg durch Verwendung von bereits bekannten Elementen vereinfacht, bzw. sollte jemand als erste Programmiersprache die Comic-DSL lernen, so ist ein Umschwung zum ebenfalls gestalterischen HTML einfacher möglich. Ein beispielhafter Auszug der DSL ist in Abbildung 1 zu sehen.

2.3.1 MPS-Komponenten

MPS enthält einige Basis-Notationen, die es vorzustellen gilt. Knoten ("nodes"), Begriffe ("concepts") und Sprachen ("languages") [3, S.11-13].

Abstract Syntax Tree (AST) Das erstellte Programm wird stets durch einen AST beschrieben. Änderungen werden direkt am AST vorgenommen und der Code als AST gespeichert. So müssen weder Grammatik, noch Parser erstellt werden. Die Sprachdefinition erfolgt anhand verschiedener Typen von AST-Knoten (im Folgenden nur Knoten) und Regeln zu deren Beziehung zueinander.

Knoten Jeder Knoten im AST hat einen Eltern- und Kind-Knoten, Eigenschaften und Referenzen auf mehrere andere Knoten. Wurzelknoten ("root nodes") besitzen keine Elternknoten. Bei der Arbeit mit MPS ist jedes Element, das man benutzt, ein Knoten.

Begriff (en. *concept*) Jeder Knoten hält eine Referenz auf seine Deklaration, seinen Begriff. Ein Begriff setzt den "Typ" des Knotens. Begriffe können Vererbungen nutzen und sind selbst auch AST-Knoten mit dem Begriff *ConceptDeclaration*.

Sprache Eine Sprache in MPS ist eine Sammlung von Konzepten mit zusätzlichen Informationen. Diese enthalten Details zum Autor, Typsystem, Intention, Generator etc., welche zur Sprache zugehörig sind. Diese Informationen formen mehrere Sprachaspekte. Eine Sprache kann von einer anderen Sprache erben. MPS unterstützt die volle Wiederverwertung der Komponenten.

```
concept Page extends    BaseConcept
             implements <none>

  instance can be root: false
  alias: <page
  short description: page of a comic book. is made of comic-panels

  properties:
  grid : integer

  children:
  pageContent : PageContent[0..n]
  attribute   : Attribute[0..1]
```

Figure 2: Struktur des Knoten "Page"

```
<default> editor for concept Page
  node cell layout:
    [/
      [> <page grid = { grid } > <]
      [> ---> (> % pageContent % <) <]
                /empty cell: <default>
      [> </page> <]
    /]
```

Figure 3: Editor des Knoten "Page"

2.3.2 MPS Workflow

MPS nutzt Datenstrukturen zur Sprach-Definition [13, S.53f - 6.2 Realisierung mit MPS], [3, 15]. Enthalten sind hier die in Abschnitt 2.1.1 beschriebenen Grundelemente, sowie ergänzend Einschränkungen, um eine valide abstrakte Repräsentation zu garantieren.

Struktur Die Struktur modelliert die Knoten des späteren AST. Jeder mögliche Knoten wird durch einen Begriff repräsentiert. Ein Begriff hat einen Namen, Eigenschaften und mögliche Kindknoten.
Als Beispiel sei der Begriff "Page" vorgestellt, in Abbildung 2. Kindknoten sind vom abstrakten Typ "PageContent", was sowohl "Panel", als auch "Transition", welche wiederum Panel enthalten, einschließt.

Editor Der Editor legt das Aussehen und den Aufbau des Userinputs fest. Zugehörig zu den einzelnen Knoten beschreibt ein Editor, wie die Notation des entsprechenden Knotens auszusehen hat. Beispielhaft für den Knoten "Page" in Abbildung 3 gezeigt. In der Regel werden konstante Begriffe, die Übergabe von Attributen, sowie die Auflistung der enthaltenen Kindknoten definiert. Syntax Highlighting wird in CSS-Syntax an den einzelnen Komponenten (Wörtern) definiert. So werden die Schlüsselwörter page, panel, usw. nach dem in MPS bereits vorhandenen Style "Keywords" dargestellt. Eine Definition eigener Styles ist möglich.

Generator Aus dem per Editor beschriebenen AST muss im letzten Schritt der DSL-Verarbeitung Code in der Zielsprache generiert werden. Hierfür wird, vom Wurzelknoten ausgehend, festgelegt, welche Knoten, Attribute und Kindknoten in welchen Code umgewandelt werden sollen. Beispielsweise wird über alle Knoten des Typs *page* iteriert und jeweils eine Methode *generatePage()* aufgerufen, in welcher die Kindknoten, also die Panel, jeweils ein *generatePanel()* erzeugen. Die Hauptklasse hierfür ist in Abbildung 4 zu sehen.

Constraints werden als zusätzliche Regeln angelegt und schränken die Eigenschaften eines einzelnen Knoten ein. Zum Beispiel muss ein Panel mit einem Kindknoten *Background* auch einen Rahmen (*border*) besitzen.

Actions definiert, wie mit im Editor neu erzeugten Knoten umgegangen werden soll. So können Knoten beim Anlegen mit default-Parametern befüllt werden.

```
root template
input Comic

public class map_Comic {

  public static void main(string[] args) {
    ComicGenerator generator = new ComicGenerator();
    generator.setComicName("$[Comic]");
    // generator.setProjectPath("path/to/dir");
    // generator.setPageParameters(400, 500);

    // generator.loadCharacter("Name", "path/to/model");
    $LOOP$[->$[createPage](generator);]
    generator.exportToHTML();

  }

  $LOOP$method[private static void $[createPage](ComicGenerator generator) {
                 generator.generatePage($[3]);
                 $LOOP$[$SWITCH$switch_pageContent [generator.generatePanel(true, 1 / 3);]]
               }
```

Figure 4: Hauptklasse des Generators

Behavior ermöglicht die Definition verschiedener Methoden, die von anderen Modellen der Sprache genutzt werden könne. Hier kann z.B. eine Methode *isValidPanelWidth* definiert werden.

2.3.3 Besonderheiten

Die Erstellung von Code mit dem Projectional Editor ist zunächst gewöhnungsbedürftig, da viel mit Autovervollständigung und einer festgelegten Auswahl von Eingaben gearbeitet wird. Nach den MPS-Entwicklern soll dies nach kurzer Eingewöhnungszeit in eine schnellere Arbeitsweise resultieren.[15]
Nach eigener Einschätzung ist die Arbeit mit dem Projectional Editor hauptsächlich ungewohnt. Eine Eingewöhnung hat aufgrund der bisher geringen Nutzung noch nicht stattgefunden. Ggf. haben sich zur nächsten Ausarbeitung mehr Erfahrungswerte ergeben.
Der große Vorteil mit MPS ist, dass eine Fehlerbehandlung von invalidem Code gänzlich entfällt, da nur valide Eingaben möglich sind. So kann garantiert werden, dass stets mit einem fehlerfreien AST gearbeitet wird.
Eine Erweiterung der bereits beschriebenen Sprache durch weitere Knoten und Attribute ist ohne Schwierigkeiten möglich und macht die Erweiterbarkeit der Comic-DSL somit einfach.

Nachteilig ist, dass mit erstelltem DSL-Code und dem dahinter liegenden AST nicht wie mit einem Textdokument umgegangen werden kann. Das Abspeichern funktioniert immer als AST, Löschen und Kopieren einzelner Code-Elemente ist nicht unbedingt ohne Weiteres möglich.

2.3.4 Schnittstelle zum ComicGenerator

Die Zielsprache der Comic-DSL soll die Schnittstelle des *ComicGenerator*s ansprechen und über einzelne Methodenaufrufen Parameter belegen und das Generieren von Panels anstoßen.

Um bei der Generierung der Zielsprache den ComicGenerator ansprechen zu können muss dieser als jar-Datei eingefügt werden. Die jar-Datei wird als Artefakt des Projektes erhalten, in das MPS-Projekt kopiert und dort als Stub des Generators eingebunden.

2.4 Ausführung

MPS ermöglicht es, den DSL-Code direkt ausführen zu lassen. Hierzu muss die Hauptklasse (Comic) ausführbar gemacht werden, indem das Interface *IMainClass* implementiert wird.
Nun kann mit Rechtsklick auf die DSL-Datei direkt ein "Run" angestoßen werden.

2.5 Lücken

Die DSL muss im nächsten Schritt noch eine Erweiterung erfahren, um die Panel mit Inhalt befüllen zu können. Hierzu müssen für die entsprechenden Knoten die Editoren und Generatoren umgesetzt werden.
Die Handhabung von MPS ist noch nicht vollends durchdrungen. Vor allem das Einbinden von Dependencys und Librarys in verschiedene Elemente der Sprache ist recht unübersichtlich gestaltet.
Zuletzt ist noch zu klären, wie der generierte Java-Code am nutzerfreundlichsten auszuführen ist und welche Ansätze es für eine umschließende GUI gibt, in der MPS und ggf. grafische Bedienelemente die Comic Generierung vereinfachen.

3 ComicGenerator

Das in PJ1 [9] begonnene System wird nun erweitert. Der bisherige Quellcode wird in eine passende Architektur eingepasst und entsprechend den neuen Anforderungen erweitert.

3.1 Systemarchitektur

Die ursprüngliche *Game Loop* von LibGDX ruft im Wechsel die Methoden *update* und *render* auf. Das Update, also der Aufbau der in einem Panel darzustellenden Szene, wird von der Komponente *SceneBuilder* übernommen. Das Rendering im Comic-Stil und das Hinzufügen von Hintergründen und Sprechblasen, sowie der finale Export als Comic-Panel erfolgt in der Komponente *PanelRenderer*. Die übergeordnete Komponente *ComicGenerator* ruft beide Komponenten im Wechsel auf und dient zudem als Schnittstelle für die DSL.

3.1.1 Module

Die einzelnen Module des ComicGenerators sind aus dem Schaubild (Abbildung 5) der vorherigen Ausarbeitungen abzulesen [8, 9].

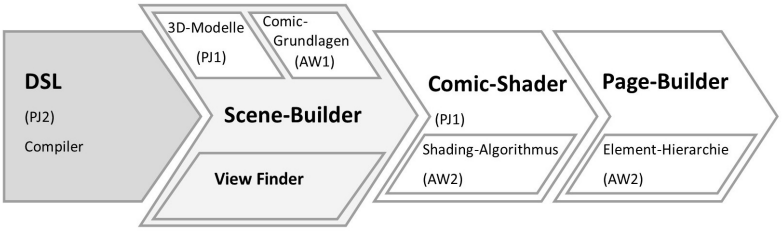

Figure 5: Teilschritte der MA "Comic Generierung"

3.1.2 LibGDX Architektur

Da das verwendete Framework LibGDX ist bietet es sich an, entsprechende Projekte als Vorbild für den eigenen Architekturentwurf zu wählen. Das Klassendiagramm in Abbildung 6 gehört zum Spiel "Canyon Bunny", welches in dem LibGDX-Werk *Learning LibGDX Game Development - Second Edition* [4] vorgestellt wird. Das Zusammenwirken von Render-, Controller- und Hauptklasse kann für den eigenen Ansatz übernommen werden.

3.1.3 Eigene Architektur

Die Architektur des ComicGenerators orientiert sich an der Standard-LibGDX-Architektur. Zu sehen ist sie in Abbildung 7. Zusätzlich werden hier Interfaces verwendet, um die einzelnen Komponenten bei Bedarf leichter austauschen zu können.

ComicGenerator die Main-Klasse dient als Schnittstelle für die DSL. Hier werden Instanzen von SceneBuilder und PanelRenderer erstellt und diese entsprechend aufgerufen.

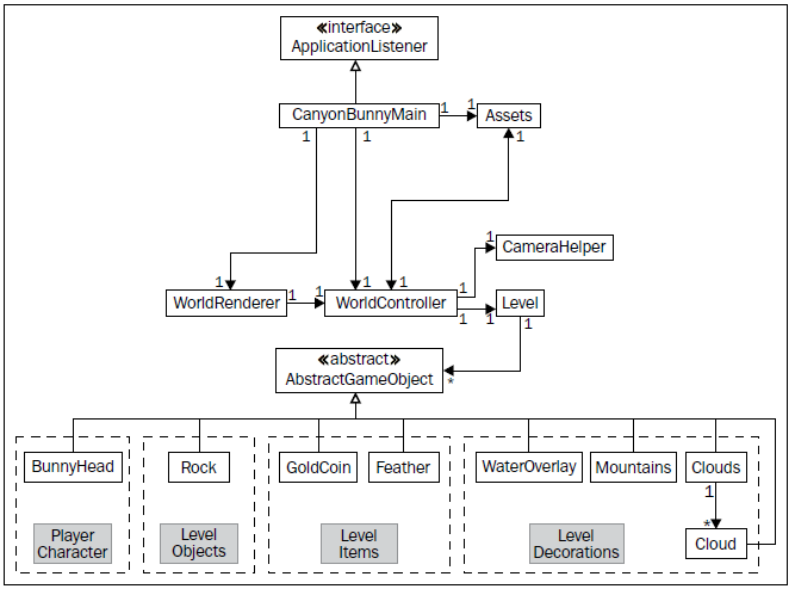

Figure 6: Architektur des LibGDX-Spiels "Canyon Bunny" [4, S.110]

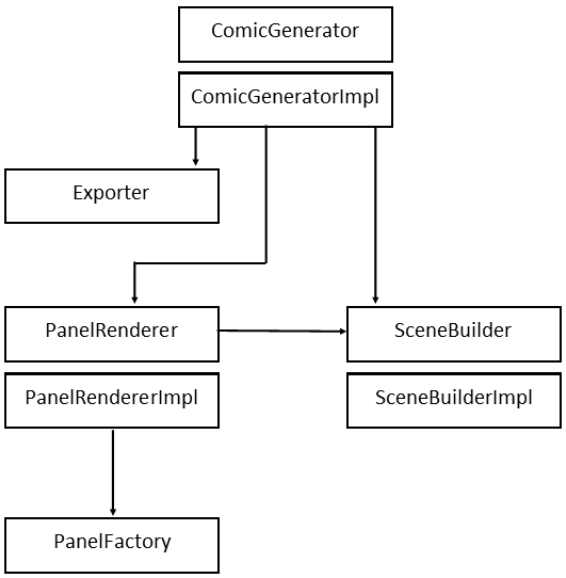

Figure 7: Architektur Comic Generator

SceneBuilder entstpricht der Controller-Klasse. Hier wird der Zugriff auf die 3D-Modelle und Sprechblasen benötigt (Assets), um die vom Renderer darzustellende Szene zu erstellen. Mit Hilfe des View-Finders wird der passende Bildausschnitt gefunden und die Szene, sowie die zugehörige Kamera-Positionierung an den PanelRenderer weiter gegeben.

PanelRenderer ist die Rendering-Komponente. Die zuvor erstellte Szene wird gerendert, sodass Hintergrund, Charaktere und Sprechblasen sich in der richtigen Reihenfolge überlagern und das Ergebnis als fertiges Panel abgespeichert. Die Umsetzung dieser Komponente wurde in PJ1 [9] vorgenommen.

Exporter ermöglicht eine Repräsentation der generierten Panel in einem gewünschten Format. Im Rahmen von PJ2 wird ein Export ins HTML-Format umgesetzt. Auf Basis der HTML-Datei ist ein Export in ein Textdokument, PDF oder ePub ohne großen Mehraufwand möglich.

Constants hält Variablen und Werte, auf die alle (oder mehrere) Klassen zugreifen müssen. Grundsätzlich führt ein Pfeil von jeder Klasse aus zur Constants-Klasse.

Assets lädt sämtliche benötigte Ressourcen. Im Falle des ComicGenerators also hauptsächlich 3D-Modelle und ggf. benötigte Hintergrundbilder.

3.1.4 Game Loop

Vor allem für Spiele ist es üblich, dass eine so genannte *Game Loop* in einem vorgegebenen Intervall das darzustellende Spiel aktualisiert, um Änderungen, wie User-Eingaben wirksam zu machen, und daraufhin rendert, um die Änderungen darzustellen.
Im Falle des ComicGenerators ist eine solche Loop nicht nötig, bzw. nicht erwünscht. Pro Panel soll ein Update geschehen, in welchem die Szene aufgebaut wird, sowie ein Render, in welchem der Comic-Shader angewendet und das fertige Panel exportiert wird. Anstatt einer Dauerschleife wird der Darstellungsablauf also einmal pro Panel durchlaufen, veranschaulicht ist dies in Abbildung 8. Danach wird das System wieder gestoppt (dispose).
Dies ist möglich, indem die über die Schnittstelle aufgerufene Methode einmal update und einmal render aufruft, ohne eine Schleife zu verwenden.

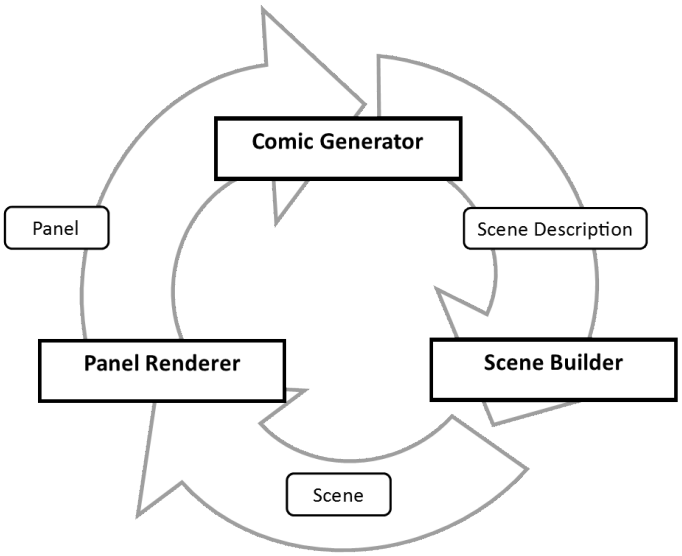

Figure 8: Comic Generator Game Loop

3.1.5 Schnittstelle

Interface-Klassen werden zum einen innerhalb des ComicGenerator-Systems verwendet, um Komponenten leichter austauschen zu können. Die wichtigste Schnittstelle stellt allerdings die der Klasse ComicGenerator dar. Gegen sie wird der durch die DSL generierte Code implementiert. Per DSL eingehende Befehle sollten also möglichst intuitiv und übersichtlich an den ComicGenerator weitergegeben werden.

Die hauptsächlich genutzten Methoden sind

- **generatePage(int grid)** Setzt Parameter wie aktuelle Seitenzahl und Grid als Vorbereitung auf das Generieren der Panels.
- **generatePanel(boolean border, float width)** Hier werden update und render aufgerufen und ein Panel erhalten. Die Methodendefinition ist noch nicht vollständig, es sollen noch enthaltene Charaktere, Sprechblasen, etc. übergeben werden. Für das prototypische Setzen der Panel reicht es jedoch aus.

3.2 Prototyp

Der im Rahmen von PJ2 implementierte Prototyp ist in der Lage durch die Methodenaufrufe über die Schnittstelle Panel zu generieren. Die Höhe der einzelnen Panel hängt hierbei von dem Grid der Seite ab (also der Zeilenanzahl), die Breite wird als eigener Parameter übergeben. Diese Werte orientieren sich zudem an einer vorher gesetzten Seitenbreite (z.B. Din A4).

Jedes generierte Panel wird in einer vom System erstellten Ordnerstruktur abgelegt. Für jede Seite wird ein eigener Ordner *page_x* mit Seitenzahl generiert, in welchem die zugehörigen Panels abgelegt sind. Der übergeordnete Ordner trägt den Namen des Comics und beinhaltet ggf. weitere generierte Dateien. Für den Export nach HTML werden hier eine HTML- und eine CSS-Datei erzeugt.

3.2.1 Lücken

Die Komponente SceneBuilder muss noch implementiert werden. Bisher wird lediglich ein vorgefertigtes Modell geladen und ohne weitere Modifizierung angezeigt. Hier müssen die Auswahl der Perspektive und die Anpassung des Modells an gewünschte Gestik und Mimik erfolgen.

3.2.2 QA

Regelmäßiges und gründliches Testen erhöht die Qualität der Software. Entsprechend muss möglichst früh mit umfangreichen Tests begonnen werden, man vergleiche die Extreme-Programming Methode Test Driven Development (TDD). Mit dem Programm sollte auch der Umfang der Tests wachsen.

JUnit Unit Tests werden an das core-Projekt angehängt. Für eine übersichtliche Struktur wurde hierzu der Code in ein *main*-Package verschoben und ein neues *test*-Package angelegt.
Hier werden nun JUnit-Tests gepflegt, welche die korrekte Funktion des Comic-Generators überprüfen. Hierbei werden hauptsächlich das korrekte Abspeichern und Berechnen von Parametern überprüft, da ein Test des fertigen Panels (sind die richtigen Objekte in richtiger Perspektive zu sehen?) technisch kaum umsetzbar ist. Ebenfalls sind variierende Ergebnisse beim Erstellen der Szene erwünscht, um dem Leser Variabilität und Abwechslung zu bieten, sodass auch kein genauer Abgleich vorgenommen werden kann [5].

MPS Testing Im Rahmen von MPS können Tests durchgeführt werden. Diese werden als zusätzliche Bausteine in die Solution einer Sprache integriert und können Knoten oder den Editor testen. Das Aussehen dieser Testfälle stimmt zum Großteil mit Unit-Tests überein.

Usability Lab Sobald der ComicGenerator soweit fertig zur Nutzung ist bieten sich Tests im Usability-Lab an. Nutzer, die aus dem Programmier- oder Comic-Fachbereich kommen oder noch gänzlich unerfahren sind können die Intuitivität und Nutzerfreundlichkeit des Systems durch Ausprobieren testen.
Da diese Tests an den Fortschritt der Arbeit geknüpft sind können sie erst im nächsten Semester durchgeführt werden, wenn der ComicGenerator weitestgehend einsatzbereit ist. Entsprechendes Feedback kann in letzte Änderungen und Anpassungen einfließen.

4 Schluss

Die im Rahmen dieser Arbeit präsentierten Fortschritte und Ergebnisse finden sich zum Großteil auch im parallel stattfindenden Masterseminar wieder. Hier wird der aktuelle Stand mit einem Rückblick auf zurückliegende Arbeiten präsentiert [7, 8, 9].
Noch ausstehende Arbeitsschritte erreichen den Abschluss der Masterarbeit und werden in derselben beschrieben werden.

4.1 Nächste Schritte

Allgemein gilt es, die DSL und den ComicGenerator um den Inhalt der Panel zu erweitern. Die DSL muss entsprechende Knoten spezifizieren, die Beschreibung an den ComicGenerator weiter geben und dieser eine Umsetzung im SceneBuilder schaffen.
Vorbereitend wurden bereits Datentypen *CharacterDescription* und *PanelDescription* angelegt, um die Schnittstellendefinition mit möglichst wenig Parametern zu versehen.

Die große Komponente *SceneBuilder* steht noch aus und stellt, wie im nächsten Abschnitt erläutert, die größte Herausforderung dar. Modelle müssen geladen, platziert und angepasst werden. Nicht zuletzt müssen Sprechblasen und weitere Textboxen in die Auswahl des Szenenausschnitts mit eingebunden werden, damit kein Inhalt überdeckt wird.

Ein ständiges Testen der neu hinzukommenden Funktionen darf natürlich nicht vernachlässigt werden.

4.2 Risiken

Das noch fehlende Modul SceneBuilder ist nach wie vor das größte Risiko. Es gilt die Bones eines importierten Meshes so zu rotieren, dass Gesten dargestellt werden können, die wiederum möglichst auf sämtliche Modelle anwendbar sein sollen. Die Beschreibung solcher Gesten kann zu umfangreicher Fleißarbeit führen und eine Datenmenge erzeugen, die es gilt elegant in das System einzubinden und dabei erweiterbar zu halten, um später weitere Gesten ohne großen Aufwand hinzufügen zu können.

Die Mimik der Charaktere sei hier noch einmal gesondert genannt, da bisher noch nicht feststeht, wie eine Umsetzung aussehen soll. In Blender existieren ShapeKeys, um das Mesh anzupassen und so ein Lächeln oder Stirnrunzeln darzustellen. Inwieweit diese für die Zwecke des Comic-Generators genutzt werden können, und vor allem ob ein Ansprechen aus LibGDX heraus möglich ist, gilt es noch zu klären.

References

[1] Software Factories - Industrialized Software Development.

[2] Intentional, 2014.

[3] MPS User ' s Guide, 2014.

[4] Balakrishnan Nair, S., and Oehlke, A. *Learning Libgdx Game Development*, second edition ed. Packt Publishing Ltd., Birmingham, 2015.

[5] Beck, K., and Gamma, E. Test Infected: Programmers Love Writing Tests. *Java Report 3*, 7 (1998), 51–56.

[6] Fowler, M. Language Workbenches: The Killer-App for Domain Specific Languages?, 2005.

[7] Harbke, F. Ausarbeitung: Comic Generierung.

[8] Harbke, F. Comic Generierung - AW 2.

[9] Harbke, F. Comic Generierung - PJ 1.

[10] Jetbrains. Meta Programming System: Language Oriented Programming environment and DSL creation tool, 2015.

[11] Langlois, B., Jitia, C.-E., and Jouenne, E. DSL classification. *OOPSLA 7th Workshop on Domain specific modeling* (2007).

[12] Merkle, B. Textual Modeling Tools: Overview and Comparison of Language Workbenches. *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (2010), 139–148.

[13] Pahl, S.-A. *Entwicklung einer domänenspezifischen Sprache zur Modellierung und Validierung eines Architekturentwurfes nach den Regeln der Quasar-Standardarchitektur*. Bachelor thesis, Hamburg University of Applied Science, 2011.

[14] Pane, J. F., and Myers, B. Usability Issues in the Design of Novice Programming Systems. *Education*, August (1996), 78.

[15] Pech, V. Fast Track to MPS, 2014.

[16] Pech, V. Introduction to Jetbrains MPS, 2015.

[17] Xtext2.5Documentation. Xtext 2.5 Documentation.

[18] Zechner, M. LibGDX - Homepage, 2013.

```
< comic name = " Test " font_size = 12 project_path = " /usr/local " >
  << ... >>

< page grid = 3 >
  < panel border = " true " width = " 1_3 " >
    << ... >>
    << ... >>
  </ panel >
  < panel border = " true " width = " 1_3 " >
    << ... >>
    << ... >>
  </ panel >
  < panel border = " true " width = " 1_3 " >
    << ... >>
    << ... >>
  </ panel >
  < transition type = " Action " >
    < panel border = " true " width = " 1_2 " >
      << ... >>
      << ... >>
    </ panel >
    < panel border = " true " width = " 1_2 " >
      << ... >>
      << ... >>
    </ panel >
  </ transition >
  < panel border = " true " width = " 1_4 " >
    << ... >>
    << ... >>
  </ panel >
</ page >
</ comic >
```

(a) DSL-Code in MPS

```
package ComicDSL.sandbox;

/*Generated by MPS */

import de.haw.ma.interfaces.ComicGenerator;
import de.haw.ma.ComicGeneratorImpl;

public class map_Comic {

  public static void main(String[] args) {
    ComicGenerator generator = new ComicGeneratorImpl();
    generator.setComicName("Project 2");
    // <node>
    // <node>

    // <node>
    createPage_a(generator);
    generator.exportToHTML();

  }

  private static void createPage_a(ComicGenerator generator) {
    generator.generatePage(3);
    generator.generatePanel(true, 13);
    generator.generatePanel(true, 13);
    generator.generatePanel(true, 13);
    generator.generatePanel(true, 12);
    generator.generatePanel(true, 12);
    generator.generatePanel(true, 14);
  }


}
```

(b) Generierter Java-Code in MPS

Figure 9: Quellcode zur Panelgenerierung

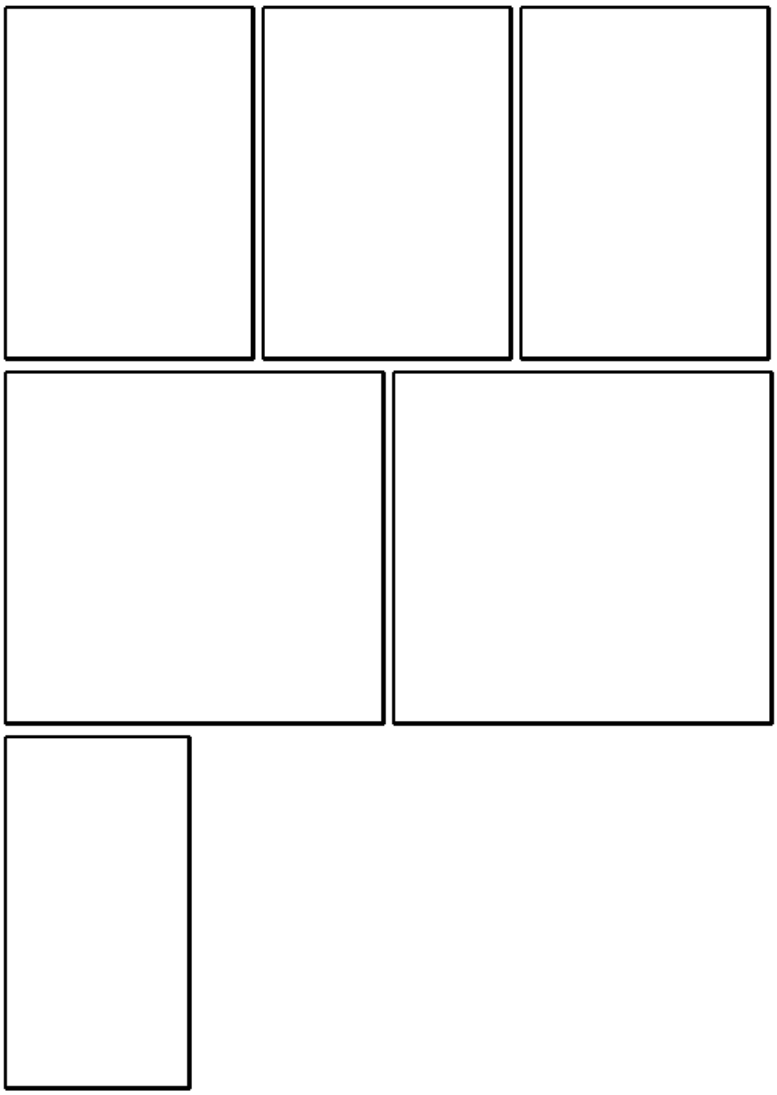

Figure 10: Generierte Panels im Browser

Glossar

Actions MPS-Modell. Definiert Aktionen, die beim Einfügen neuer Knoten ausgeführt werden.

AST Abstrakter Syntaxbaum (en. "abstract syntax tree").

Constraint MPS-Modell. Definiert Regelungen, die innerhalb eines Knoten eingehalten werden müssen.

DSL Domänenspezifische Sprache (en. "domain specific language").

Editor MPS-Modell. Definiert die Quellsprache, in welcher der User den Code verfasst.

Generator MPS-Modell. Erstellt Code in der Zielsprache (z.B. Java).

Hypertext Markup Language (HTML) Auszeichnungssprache, die von Browsern interpretiert werden kann.

Language Workbench (LWB) Software zur Erstellung von DSLs.

Meta Programming System (MPS) PLWB von JetBrains.

Projectional Editor Mögliche Knoten sind vorgegeben, die Eingabe modifiziert ohne Parser den Syntaxbaum. Gegenstück ist ein Texteditor.

Projectional Language Workbench (PLWB) LWB mit projektionaler Anpassung der abstrakten Syntax. Nutzt einen projectional Editor.

Struktur MPS-Modell. Abstrakte Syntax, Beschreibung der Elemente einer Sprache.

Textual Language Workbench (TLWB) LWB mit textueller Beschreibung der abstrakten Syntax. Umformung unter Einsatz eines Parsers.

Xtext TLWB von Eclipse.