Sven Hodapp, Simon Schneeberger, Thorsten Niehues

Teamprojekt Dokumentation

Spray Editoren als RIA

Portierung der generierten Spray Editoren von Eclipse Graphiti auf Webtechnologie als Rich Internet Application, kurz „Spray Web“

Institution: Hochschule Konstanz
Bereich: Fakultät Informatik
Betreuer: Prof. Dr. Marko Boger
Markus Gerhart
Version: 22. Januar 2014

Inhaltsverzeichnis

1 Einleitung

Diese Ausarbeitung ist neben dem produzierten Quellcode das Resultat des Master Informatik Teamprojekts von Sven Hodapp, Simon Schneeberger, Thorsten Niehuesim Sommersemester 2013 bis Wintersemester 2013/2014.

Der grafische Editor, der durch Spray und Graphiti generiert wird, soll als Weblösung implementiert werden. Das Ziel von Spray ist es den Web Editor automatisiert zu generieren. Der Web Editor soll eine Alternative zum parallel funktionierenden grafischen Editor sein. Durch die Verwendung von Webtechnologien kann der Webeditor Multikollaboration unterstützen und soll zukünftig ein wesentlicher Funktionsbestandteil sein.

Sämtlicher Quellcode ist im Git-Repositorium tprj-msi-ss13[1] zu finden. Es gliedert sich in:

- `archive`
  Hier befindet sich alter Code, der hauptsächlich aus Experimenten besteht. Aus diesem Code ist zum Großteil der Produktionscode hervorgegangen, könnte aber noch nützlich sein für künftige Weiterentwicklungen.

- `docs`
  Der Quellcode dieser Dokumentation als LaTeX bzw. XeLaTeX Code.

- `generators`
  Dort befinden sich die Generatoren um Shapes und Spray-Logik in JSON umzuwandeln.

- `server`
  Der komplette lauffähige Produktions-Code. Ein `play 2.2.1` Server stellt anhand der beispielhaft vorgegebenen Spray-Defintionen den Spray Web Editor dar.

  - `public/javascripts/spray`
    Der Hauptteil des Spray Web Editors (JavaScript Code).
  - `app/controllers`
    Hauptanwendung von Spray, stellt insbesondere einen WebSocket sowie die Persistierung des vom Editor erstellten Modells bereit.

[1] Github Link: `https://github.com/magerhar/tprj-msi-ss13`

- `app/views`
  Hier werden alle Spray Web Editor Komponenten als HTML-Templates zusammengebracht.

Wenn das Playframework in Version 2.2.1 installiert ist, kann der Server wie folgt gestartet werden:

```
git clone https://github.com/magerhar/tprj-msi-ss13.git
cd tprj-msi-ss13/server
play run
```

Danach kann die Webseite mit dem Spray Web Editor im Webbrowser[2] auf `http://localhost:9000/` betrachtet werden.

1.1 Einführung in Spray

> The Graphiti framework is a new approach to create highly sophisticated visual editors on top of the GEF framework. Graphiti can easily be integrated with EMF as the domain modeling framework. [...] Spray aims to provide Domain Specific Languages (DSL) [...] to describe Visual DSL Editors [...] and provide code generation [...] to create the boilerplate code for [...] the Graphiti framework. [...] With the help of the tools created with Spray, Graphiti based diagram editors can be created much faster and reliable than doing it purely by hand. (Spray Webseite, 2014)

Das bedeutet: Spray bietet domänenspezifische Sprachen (DSLs) an, um einen grafischen Editor zu beschreiben. Man könnte sagen, dass man mit Spray eine Grammatik für grafische Editoren an die Hand bekommt. Man beschreibt textuell das Layout und Aussehen von Shapes, und beschreibt zudem was zulässige Verbindungen zwischen den einzelnen Shapes sind. Resultat ist ein domänenspezifischer grafischer Editor. Auf Abbildung 1 ist der prinzipelle Spray zu Editor Ablauf.

[2] Getestet mit Google Chrome 32.x, Apple Safari 7.x oder Mozilla Firefox 26.x unter Windows 7 bzw. Mac OS X 10.9.

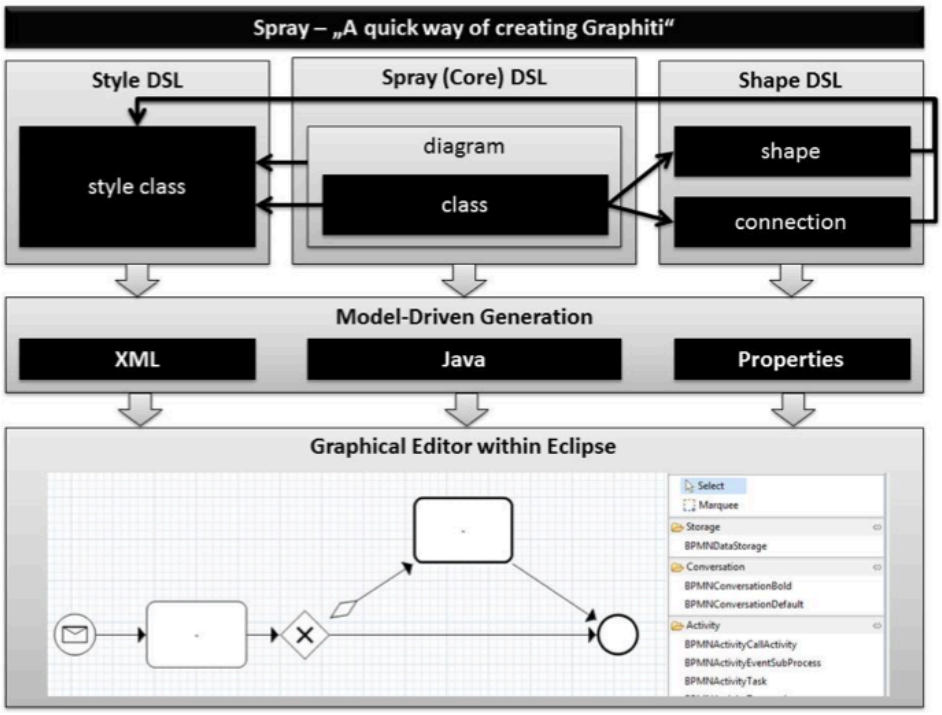

Abbildung 1: Prinzipelle Architektur von Spray. (aus Filippelli u. a., 2012, S. 3)

1.2 Ziele von „Spray Web“

Die Hauptaufgabe dieses Projekts ist herauszufinden, wie man ein solcher Editor möglichst generisch mit Webtechnologien umsetzen kann. Dazu gehört, dass die Shapes sowie Connections nach den von Spray vorgegebenen (Layout) Definitionen dargestellt werden können.

Spray gibt auch (Logik) Definitionen vor, was für Shapes miteinander über welche Connection verbunden werden darf. Es sollte also über den Editor abgefangen werden können, wenn versucht wird falsche Connections zu knüpfen.

Aus den Shapes und deren Connections entsteht ein konkretes Modell, welches vom Benutzer so modelliert wurde. Dieses Modell ist die Instanz des darüberliegenden Metamodells. Das Metamodell entsteht also implizit aus den durch Spray vorgegebene Logikdefinitionen. Genauer gesagt ensteht ein Ecore Metamodell, dazu mehr im Abschnitt 1.3. Diese Modellinstanz entsteht aus dem was der Benutzer „zusammenklickt“, dies muss durch den Web Editor ebenfalls persisiert werden.

Dieses Teamprojekt ist als Forschungsarbeit zu verstehen auf deren Erkenntnisse und Fundamente aufgesetzt werden kann. Die Hauptziele zusammengefasst sind:

- Shapes gemäß den Definitionen produzieren,
- Modellüberprüfung auf Korrektheit,
- Persistierung der Modellinstanz (sowie Koordinaten der gesetzten Shapes).

1.3 Ecore Metamodell

Um modellgetriebene Softwareentwicklung besser zu verstehen, sollte grob bekannt sein was ein *Modell* bzw. was ein *Metamodell* ist.

Laut (Drachenfels, 2013, S. 2-1) ist „Ein Modell [...] immer ein Abbild von etwas“, welches nur die Merkmale des Originals erfasst, welche dem Modellierer relevant erscheinen. Ein Modell „wird von bestimmten Nutzern innerhalb einer bestimmten Zeitspanne zu einem bestimmen Zweck für ein Original eingesetzt“.

Weiterhin wird in (Drachenfels, 2013, S. 2-5) beschrieben, dass „ein Metamodell [...] ein Modell von Modellen” ist, wobei dies beschreibt „aus welchen Arten von Enitäten und [...] Beziehungen [...] konforme Modelle bestehen“.

Ecore ist ein in der Eclipse Welt beliebtes Format für Metamodelle. Es kann in XMI[3] serialisiert werden. Das Spray Framework fabriziert aus der Spray DSL ein Ecore Metamodell. Damit ist es möglich zu prüfen, ob das Modell welches vom Benutzer modelliert wurde korrekt ist.

1.4 Graphiti

Mit Graphiti[4] ist es möglich grafische Editoren, die innerhalb von Eclipse laufen, zu erstellen. Intern setzt es auf das *Graphical Editing Framework*[5] (GEF) und Draw2d, wobei Graphiti diese Frameworks komplett hinter seiner API versteckt. Spray generiert schlussendlich Graphiti-Quellcode. (sinngemäß aus Filippelli u. a., 2012, S. 2)

[3] XMI steht für XML Metadata Interchange
[4] Webseite: `http://www.eclipse.org/graphiti/`
[5] Webseite: `http://www.eclipse.org/gef/`

Achtung: Draw2d nicht mit *Draw2D touch* verwechseln! Ersters ist für Java, letzteres für JavaScript im Browser. Beide Frameworks haben nichts miteinander zu tun, außer dass Draw2D touch versucht die API von Draw2d nach Möglichkeit nachzuahmen.

2 Anforderungen

Die Anforderungsanalyse spezifiziert die Anforderungen an das Browser Shape Tool. Mithilfe einer Recherche und Brainstorming werden die Nutzungsanforderungen aufgestellt und daraus die funktionalen und nicht-funktionalen Anforderungen abgeleitet. Grafische Entwürfe der Benutzeroberfläche werden zeigen, wie die Nutzungsanforderungen aus Blackbox-Sicht umgesetzt werden können.

Projektvorgaben Nachstehend sind die Vorgaben der Projektaufttraggeber definiert:

- Implementation eines JS-Browser-Frameworks,
- Spray Definition annehmen, verarbeiten, interpretieren, darstellen,
- Darstellung ist Eclipse Graphiti-like,
- Shape-Darstellungen bearbeiten können,
- Diagramm-Modell muss validierbar und persistierbar sein.

2.1 Nutzungsanforderungen

Um die Funktionen des Systems zu definieren, müssen zuerst die Nutzungsanforderungen identifiziert werden. Diese sind sogenannte Akzeptanzkriterien und werden im Abnahmetest, welcher am Ende des Projekts durchgeführt wird, validiert. Auf Tabelle 1 werden die Nutzungsanforderungen gelistet, nummeriert und nach dem MoSCoW-Prinzip priorisiert:

- Must (M, unbedingt erforderlich)
- Should (S, sollte umgesetzt werden)
- Could (C, kann umgesetzt werden)

#	Anforderung	Prio.
1	Der Nutzer muss das Modell betrachten können.	C
2	Der Nutzer muss das Modell z.B. als Dokumentation einbinden können. (d.h. der View Mode muss druckbar seitens eines Webbrowsers sein).	C
3	Der Nutzer muss das Modell bearbeiten können.	M
4	Das System muss eine Spray Shape Definition laden können.	M
5	Das System muss die Semantik eines Shapes verstehen/abbilden können.	M
6	Der Nutzer muss Shape-Elemente gemäß der zugrunde liegenden Spray Shape-DSL zeichnen können. Der Nutzer muss eine Aufzählung, Auswahl bzw. Übersicht aller ermöglichten Shape-Elemente erhalten.	M
7	Der Nutzer muss die Bearbeitungssicht zoomen können.	C
8	Der Nutzer muss den Zustand speichern können. (i) Koordinaten eines jeden Elements, (ii) Das eigentliche (semantische) Modell.	M
9	Der Nutzer muss einen gespeicherten Zustand des Modells laden können. (i) Koordinaten eines jeden Elements, (ii) das eigentliche (semantische) Modell.	M
10	Der Nutzer muss erkennen können, ob der Zustand valide ist oder nicht.	C
11	Der Nutzer muss durch das System die Elemente automatisch ausrichten lassen können. „Ausrichtungsassistenz“	W

Tabelle 1: Nutzungsanforderungen für Spray Web, nach dem MoSCoW-Prinzip priorisiert.

- Won’t (W, wird nicht umgesetzt, aber für die Zukunft vorgemerkt)

2.2 Funktionale Anforderungen

Anhand von grafischen Entwürfen (siehe Abbildung 2 und 3) der Benutzeroberfläche wird dargestellt, wie die Nutzungsanforderungen aus Blackbox-Sicht umgesetzt werden sollen. Die Screens und die dazugehörigen Beschreibungen definieren, wie das Shape-Tool funktioniert und wie es zu bedienen ist. Diese Systemanforderungen werden im Systemtest verifiziert.
Anforderungen an die Werkzeugleiste die auf Abbildung 2 dargestellt wird:

- Undo, Redo
- Select, damit kann ein oder mehrere Shapes selektioniert werden

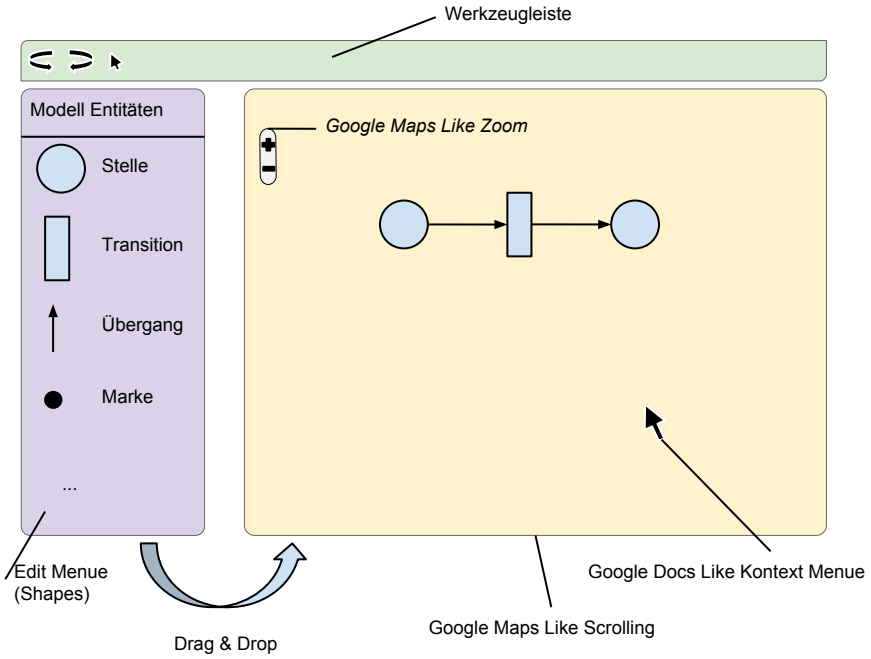

Abbildung 2: Entwurf GUI (Beispiel Modell: Petrinetz).

- Werkzeuge mit Hotkeys bedienbar (z.B. Undo mit CTRL+Z)
- In Modellentitäten: Formen entsprechen der zur Verfügung stehenden Spray Elemente
- In Modellentitäten: Die Formen können per Drag and Drop in die Fläche gezogen werden
- Optional: Reihenfolge der Elemente bestimmbar (Vordergrund, Hintergrund)
- Optional: Speichern, Laden

Primitive Shapes Es ist essentiell, dass folgende Shape Basistypen dargestellt werden können, und ineinander verschachtelbar sind (siehe Kapitel *The Shape grammar* Warmer u. a., 2013):

- Line
- PolyLine

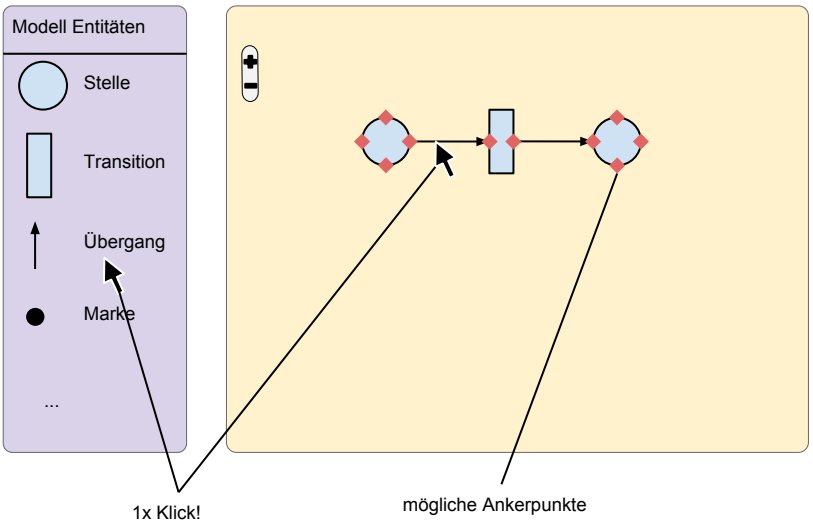

Abbildung 3: Entwurf GUI: Ankerpunkte Auswahl (Beispiel Modell: Petrinetz).

- Rectangle
- RoundedRectangle
- Polygon
- Ellipse
- Text
- Anchor

Neben den visuellen Anforderungen gibt es noch weitere Anforderungen:

- Die Texte der Shapes können editierbar sein.
- Die Shapes sind per Maus positionierbar, drehbar, beliebig skalierbar, löschbar, bearbeitbar. Dafür werden Bearbeitungspunkte eingeblendet.
- Interaktive Verschachtelung von Shapes (Compartments)
  - Gruppieren, Entgruppieren

  - Compartments geben die Größe eines Shapes implizit vor. Ist vorgegeben von Shape-DSL. (z.B. Attribute die beim Klassendiagramm hinzukommen)

- Allgemeine Verschachtelung von Shapes (statisch) z.B. bereits verschachtelte Shapes durch DSL-Definition
- Eigenes Kontextmenü für Shapes
  - Add → new Shapes (für statische Verschachtelung)
  - Cut
  - Remove
  - Copy
  - Rename (wenn benennbar)
  - Align
  - Properties: Anzeige von Eigenschaften
- Eigenes Kontextmenü für Connections
  - Rename
  - Remove
  - Align
  - Properties: Anzeige von Eigenschaften
- Kontextmenüs mit Hotkeys bedienbar (z.B. Add mit Space)
- Optional: Spezifische Kontextmenüerweiterung für jede Entity
  - z.B. Randdicke oder Hintergrundfarbe bei Ellipse
  - z.B. Shapeumwandlung von Ellipse in Kreis
- Ankerpunkte werden durch die Shape-DSL definiert, Anker sind also vorgegeben
- Verbindungen zwischen Ankerpunkten werden vom Benutzer definiert
- Bei einfachem Klick auf eine Verbindung (Connection) im Edit Menü erscheinen die möglichen Ankerpunkte als Auswahl. (Siehe Abbildung 3)

- Bei einfachem Klick auf das Shape wird das Shape aktiv und die Bearbeitungspunkte (z.B zum Skalieren) werden angezeigt.
- Bei Doppelklick auf das Shape passiert nichts
- Bei Doppelklick auf das Label (z.B. Text) wird das Shape aktiv und das Label kann bearbeitet werden.
- Defaultwert Label „Name des Shapes“
- RapidButton bei onmouseover nur bei aktivem Shape, kann man auswählen welche Optionen zur Verfügung stehen. (Beispiel: bei einer UML-Assoziation kann man diese benennen, linke oder rechte Kardinalität setzen; bei z.B. einer Klasse Methoden, Properties hinzufügen/entfernen/ändern, ...)
- Shapes müssen durch „Kasten“ markiert und gruppiert werden können.
- Die Styles der Shapes sind dynamisch mit CSS konfigurierbar

Shape API

- Connections zwischen allen Shapes müssen möglich sein
- Die Shape API muss die DSL interpretieren können und entsprechende Werkzeuge anbieten (z.B. Kreis zeichnen)
- Die API muss eine Datenpersistierung anbieten, z.B. wird von der Klasse Kreis mehrere Kreise gezeichnet, müssen die Kreise in einer Datenstruktur festgehalten werden.
- Die API muss das Modell gegen die DSL prüfen können und ggf. Fehlermeldungen darstellen.

Connections Im einfachsten Fall sind Connections Verbindungslinien zwischen zwei Shapes. Somit bilden Shapes (Knoten) und Connections (Kanten) eine gerichtete Graphenstruktur. Prinzipiell können jedoch die Endpunkte der Connections auch aus den gleichen primitiven Shape-Arten zusammengesetzt werden, um ihre visuelle Bedeutung klar zu machen.

- CDLine

- CDPolyLine
- CDRectangle
- CDRoundedRectangle
- CDPolygon
- CDEllipse
- CDText

Compartments sind Shapes die zur Laufzeit in ein anderes Shape geschachtelt werden können, oder auch wieder entfernt werden können. Sie sind auch ein semantischer Teil, neben Shapes und Connections, und werden somit auch im Modell gespeichert. Es muss also Unterstützung dafür geben, wärend der Benutzer modelliert, Shapes verschachteln zu können (via Drag and Drop).

Generalisierbarkeit Dadurch, dass sämtlicher Code durch Spray-Generatoren automatisch produziert wird, ist es nötig, dass das Spray Web Framework keinen „fest verdrahteten“ Code aufweist, sondern vielmehr sehr generellen Code der mit verschiedensten Eingaben zurecht kommt. Beispielsweise ein Petrinetz hat andere Shapes und Validierungsregeln, als ein Heizungssystem. Die Diagramme unterscheiden sich in Anzahl und Aufbau der Shapes, Connections, Compartments und Validierungsregeln, daher soll der Code so generell wie möglich gehalten werden. (Stichwort Reflections)

2.3 Nicht-funktionale Anforderungen

Während funktionale Anforderungen definieren was das System leisten soll, müssen die Eigenschaften des Systems ebenfalls spezifiziert sein. Nicht-funktionale Anforderungen werden, wie die funktionalen Anforderungen, im Systemtest verifiziert und müssen daher messbar sein. Um nicht-funktionale Anforderungen zu spezifizieren, dient die ISO 25010 als Checkliste:

Zuverlässigkeit

- Das System ist an mindestens 51 Wochen im Jahr erreichbar.
- Bei unbeabsichtigtem Bearbeitungsende (durch Browserabsturz, etc.) soll der zuletzt gespeicherte Systemzustand wiederhergestellt werden.
- Systemdaten sollen bei Verlust nach 24 Stunden wiederhergestellt werden können.

Benutzbarkeit

- Das System ist nicht barrierefrei optimiert.
- Ein Nutzer mit guten Internetkenntnissen hat das System spätestens nach einer Stunde Schulung erlernt, verstanden und kann es selbständig fehlerfrei bedienen.

Sicherheit

- Das System muss den unauthorisierten Zugriff auf den Server verhindern.
- Das System erlaubt nur registrierten Benutzern den Zugriff.
- Das System muss Benutzerdaten vertraulich behandeln.

Leistung und Effizienz

- Die Übertragung der Signale vom Client an den Server und zurück ist maximal 1 Sekunde verzögert.
- Das System muss in der Lage sein, bei einer Änderung eines Shapes, während einer Sekunde die Validität zu prüfen.
- Das System muss in der Lage sein, auch komplexe Shape-Darstellungen, in maximal 10 Sekunden zu rendern.

Wartbarkeit

- Alle Modultests sollen automatisiert durchgeführt werden.
- Jede Klasse soll testbar sein.
- Alle Klassen und alle Methoden sind ausreichend dokumentiert.
- McCabe-Mass soll nicht höher sein als 8.

Übertragbarkeit

- Die Software ist mit einem Aufwand von maximal einem Tag auf ein System, z.B. Windows Server oder Linux, installierbar.
- Die Software ist auf ein anderes System, z.B. Windows Server, anpassbar.

Kompatibilität

- Die Software unterstützt alle aktuellen Browser, die auf der WebKit-Engine (Safari, Google Chrome) basieren.
- Die Software ist kompatibel mit allen Geräten und Betriebssystemen, die die genannten Browser unterstützen.

Funktionale Tauglichkeit

- Mindestens 99.5% aller Signale werden während einer Bearbeitung eines Shape-Diagramms übertragen.

3 Architektur

Das Design der Architektur ergibt sich aus den Anforderungen und wird auf der Architekturskizze Abb. 4 veranschaulicht.

Der Editor im Browser soll alle Shapes zeichnen, welche von einer Shape-Definition vorgegeben sind. Die Shape und Logik-Definitionen (class definitions) im JSON-Format kommen von Spray. Ein Generator in Spray generiert eine JSON-Datei welche die Regeln enthält. Der Server liefert diese JSON-Dateien statisch an den Browser aus. Anhand dieser JSON-Datei generiert die Shape Factory entsprechende Figuren mit Hilfe des Draw2D touch Frameworks.

Damit Draw2D touch auch die Regeln (die vorgeben wie ein valides Modell aussieht) beachtet, werden diese an Draw2D touch übergeben. In Draw2D touch gibt es Funktionen die bei bestimmten Events aufgerufen werden (Listener-Funktionen). Dazu gehören unter anderem die Policy-Funktionen. In diesen Funktionen wird überprüft ob die gegebene „Grammatik“ des Modells verletzt wird. Ist dies der Fall wird die entsprechende Aktion nicht ausgeführt. Damit der Nutzer eine nicht erlaubte Aktion erkennt, wird das Cursor Symbol zu einem Stop-Symbol geändert.

Damit in Zukunft auch Kollaboration möglich ist, schickt der Client bei jeder Aktion eine Nachricht via Websocket an den Server. Die Nachricht an den Server enthält die genaue Beschreibung der Aktion. Aus dieser Beschreibung kann der Server eine Ecore Modellinstanz pflegen, diese wird zur Laufzeit als XMI persistiert. Bei jeder Nachricht aktualisiert der Server die XMI Datei. Ob die Aktion erlaubt ist wird bereits auf dem Client geprüft und muss deshalb nicht mehr auf dem Server validiert werden.

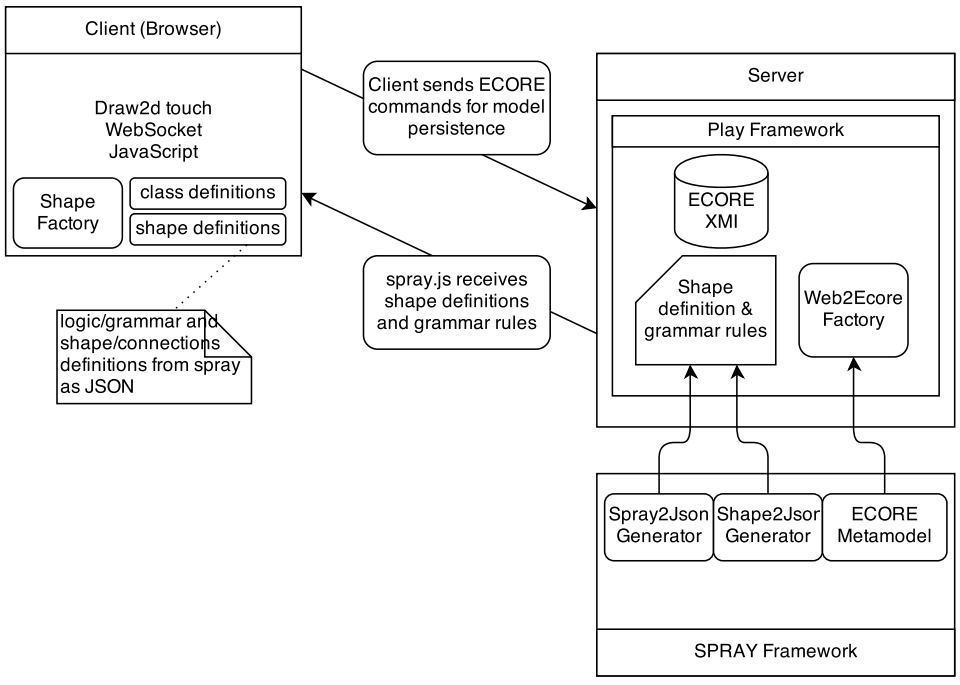

Abbildung 4: Übersicht über die wichtigsten Spray Web Komponenten.

spray.js Dort befindet sich der hauptsächliche Teil des Codes. Es befindet sich im Repositorium in `server/public/javascripts/spray`. Da Draw2D touch nicht allen Anforderungen genügt wird es um entsprechende Klassen erweitert. Diese Draw2D touch Klassen werden in `spray2d` gepflegt. Der `spray2d` Ordner ist somit der Struktur von Draw2D touch nachempfunden – Ordner als auch Namespaces.

Leider musste auch direkt Quellcode von Draw2D touch angepasst werden, da manche Änderungen über einfache Klassenvererbung oder Delegation nicht lösbar sind. Das heißt bei einem Update von Draw2D touch müssen diese Änderungen unbedingt migriert werden.

4 Umsetzung

In diesem Kapitel wird beschrieben, wie die Anforderungen praktisch umgesetzt wurden. Zunächst wird darauf eingegangen, wie Shapes gezeichnet werden, danach wie Spray passenden Code für Spray Web generieren kann und zuletzt wie Spray Web Modelle validiert und persistiert.

Hier zunächst eine Bildschirmaufnahme des entgültigen Resultats auf Abbildung 5. Diese hat zur Zeit das Regelwerk für Petrinetze geladen. Man beachte insbesondere die Ähnlichkeit zum GUI Entwurf auf Abbildung 2 die aus den Anforderungen hervorgegangen ist.

4.1 Shapes zeichnen

Es ist essentiell, dass die Basisshapes gezeichnet werden können. Zudem sollen die Basisshapes die in Kapitel 2.2 beschrieben sind, auch ineinander verschachtelbar sein. Da die Shapes auch u.a. Textfelder enthalten können, um z.B. das Shape zu benennen, ist es nötig, dass der Benutzer diese verändern kann auch innerhalb eines verschachtelten Shapes. Da die Shapes mit Connections zusammen eine Graphenstruktur bilden, müssen die Shapes über Connections miteinander verbunden werden können.

Es wäre ideal, wenn ein Toolkit existieren würde, welches all diese Anforderungen erfüllt. Falls ein solches Toolkit nicht existiert, müsste selbst eines mit den in Kapitel 2.2 gestellten Anforderungen gebaut werden.

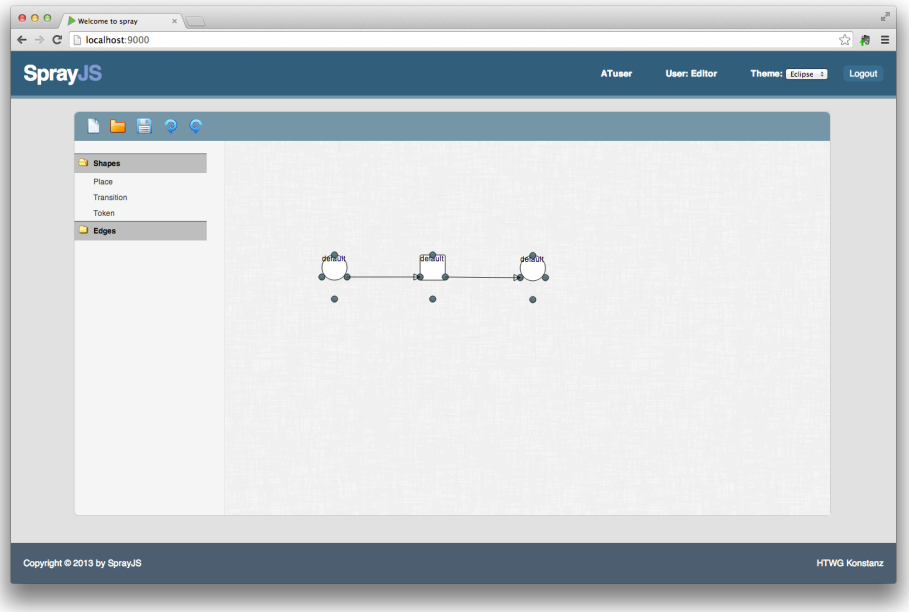

Abbildung 5: Screenshot vom gegenwärtigen Stand von Spray Web mit geladenem Petrinetz.

SVG oder Canvas? Für diesen konkreten Anwendungsfall ist SVG die bessere Wahl, siehe dazu Abbildung 6. Ein Hauptgrund[6] ist u.a. auch, dass weniger Zeilen Code benötigt werden als bei Canvas (selbst wenn man auf entsprechende Toolkits zurückgreift) und direkt Vektorgrafiken produziert werden.

	Vorteile	Nachteile
SVG	• Auflösung unabhängig • Animation möglich • DOM Zugang • XML Datenformat	• Langsames Rendern • nicht geeignet für Applikationen wie Spiele

Canvas	• High performance 2D Zeichenoberfläche
	• Alles ist ein Pixel
	• Bilder als jpg/png speichern
	• eignet sich für
	  ○ Rastergrafiken
	  ○ Bild editieren
	  ○ Pixelmanipulation
		• keine DOM Knoten
		• Text-Rendering
		• schlechte Accessibility
		• nicht geeignet für interaktive Webapplikation

Abbildung 6: SVG versus Canvas.

[6] Vgl. `http://dev.opera.com/articles/view/svg-or-canvas-choosing-between-the-two/`

4.1.1 Toolkits

Folgende Toolkits haben wir auf ihre Tauglichkeit überprüft:

- d3.js
  Ist ein Toolkit um im Web interaktive Diagramme darzustellen und verwendet dabei SVG.
- kinetic.js
  Ist eine Bibliothek, um eine komfortablere API zu Canvas anzubieten und wurde von Simon Schneeberger bereits verwendet.
- Raphael.js
  Ist ähnlich zu kinetic.js.
- LucidCharts
  Ist ein Editor ähnlich zu Visio, jedoch im Web mit HTML/CSS/JavaScript, setzt auf Canvas.
- Draw2D touch
  Bietet eine ähnliche API wie das aus der Java-Welt bekannte Draw2d, es ist in der Lage Shapes und Connections zu zeichnen. Verwendet SVG.
- go.js
  Ist ähnlich zu Draw2D touch, jedoch nicht quelloffen.

4.1.2 Draw2D touch

Erst relativ spät haben wir das Draw2D touch Toolkit gefunden. Es erfüllt quasi alle Anforderungen (siehe Abb. 7) und ist damit der Gewinner unter den durch unsere Recherche untersuchten Toolkits.
Draw2D touch wurde von (Herz, 2013) entwickelt und steht u.a. unter der GNU General Public Lizenz (GPL) und ist daher auch rein rechtlich geeignet für das ebenfalls unter einer Open Source Lizenz stehende Spray-Projekt.

4.1.3 Shape Factory

Die eigens entwickelte Shape Factory erstellt gemäss der Shape Definition aus Spray das Spray Shape in der Zeichenfläche. Die Factory kann alle Basis Shapes wie Rectangle, Rounded-Rectangle, Label und weitere als Draw2D

Was	Möglich	Wie
Basisshapes darstellbar, Linie/Polylinie, Kreis, Rechteck, Abgerundete Recktecke, Quadrat, Ellipse, Label, Image	JA alle	z.B. draw2d.shape.basic.Oval
Connections zwischen Shapes	JA	zwischen ConnectionAnchors kann ein ConnectionRouter erstellt werden
Texte der Shapes editierbar, platzierbar	JA	siehe auch Locator for children shapes
Shapes müssen durch Kasten markiert und gruppiert werden können	JA	multiple selection
Interaktive Verschachtelung	(NEIN) - vom Framework nicht unterstützt.	Funktionalität wird von uns implementiert
Allgemeine Verschachtelung	JA	children figures
Eigenes Kontextmenü für Shapes	JEIN	siehe aber Connection with context menu
Eigenes Kontextmenü für Conenctions	JA	Conntection with context menu
spezifische Kontextmenüerweiterungen	NEIN	
Ankerpunkte	JA, mehrere Ankerpunkte pro Shape möglich, mehrere Connections zu einem Ankerpunkt möglich	Connections
Nach Klick auf Shape wird Shape aktiv und die Bearbeitungspunkte werden angezeigt	JA	
Erscheinen einer Auswahl Ankerpunkte bei Klick auf eine Verindung im Edit menü	JA	Canvas Interaction Decoration
Doppelklick	JA	
Doppelklick in das Label -> bearbeitbar	JA	
Defaultwerte Label	JA	
Rapidbuttons	NEIN	Muss selbst implementiert werden
Styles anpassbar	JA	setCSSClass für Shapes
beliebige Eigenschaften für Shape	JA	setUserData für Shapes
Undo/Redo	JA	
Select	JA	
Hotkeys	JA	
Rotieren	NEIN	preAlpha
DragnDrop aus Toolleiste	JA	
Speichern, Laden	JA	JSON writer
Zoom	JA	
An Grid ausrichten	JA	
Tooltip	JA	
Highlighting bei Mousover	JA	
Reihenfolge der Elemente	NEIN	allerdings getZOrder für Shape

Abbildung 7: Draw2D touch Anforderungsüberprüfung.

touch Shapes zeichnen. Sie unterstützt auch die Erzeugung von Anchors. Durch rekursive Iteration der Spray Definition des Shapes wird das Spray Shape für die Zeichenfläche generisch aufgebaut. So können Draw2D touch Shapes beliebig ineinander verschachtelt werden. Als Mutterobjekt des Shapes wird ein Rechteck als Boundingbox verwendet. Alle weiteren Shapes werden dieser Boundingbox untergeordnet. Jedes so erzeugte Spray Shape erhält eine Draw2D touch ID und kann somit für weitere Zwecke angesteuert werden.

4.1.4 Compartments

Compartments sind Verschachtelungen einzelner Shapes (oder Gruppen von Shapes).

Draw2d bietet bereits die Möglichkeit zur Verschachtelung von Shapes. Wenn allerdings Gruppen von Shapes verschachtelt werden sollen, müssen alle Elemente einzeln dem neuen Parent hinzugefügt werden.

Auch die Commands für Undo und Redo mussten implementiert werden. Dieser Prozess ist etwas aufwändiger weil bereits auf abgeschlossene Commands zugegriffen werden muss. Manche Commands we Move werden automatisch erstellt, ausgeführt und auf den CommandStack gelegt. Insbesondere der Move und Create Command müssen modifiziert/ausgeführt werden. Hier wurden viele Fälle bereits implementiert.

Beim Laden einer abgespeicherten Zeichenfläche werden Compartments aktuell nur sehr rudimentär unterstützt.

Alternativ kann auch das nur das Laden der Compartments angepasst werden.

Compartments / Verschachtelungen / Komplexe Shapes können durch die Methode `Canvas.addFigure(child, locator)` erstellt werden.

```
var locator = new spray2d.layout.locator.FigureLocator(p);
locator.setPos(p, this.x, this.y);
this.parent.addFigure(this.figure, locator);

// Durch setDraggable(true) kann das hinzugefügt Shape
// eigenständig bewegt werden
this.figure.setDraggable(true);
```

4.2 Code-Generierung

Spray Web ist so aufgebaut, dass es von Spray generierten Code annimmt und verarbeitet. Da Spray intern um andere Generator-Implementierungen erweiterbar ist, ist es auch möglich, dass ideal für Spray Web zugeschnittenen Code erzeugt werden kann.

Die Generatoren sind in Xtend geschrieben, dieses ist wiederum Teil der Language Workbench *Xtext*[7].

Diese Generatoren sind noch nicht ins Spray-Projekt integriert, und müssen daher noch in die Spray Projektstruktur eingebracht werden. Man kann den Shape-Generator als Standalone Xtext Projekt zum laufen bekommen, der Spray-Generator (Logik) muss jedoch in der Eclipse-Umgebung direkt im Spray-Projekt eingebunden werden, da die Spray-Grammatik nicht ohne weiteres als Standalone Xtext Projekt extrahiert werden kann. Um Eclipse Spray (Developer) lauffähig zu bekommen, sind ggf. die fogenden Links hilfreich:

- `http://stackoverflow.com/questions/13575695/`
- `http://stackoverflow.com/questions/17527096/`
- `http://stackoverflow.com/questions/15374697/`

In der ersten Spray Eclipse-Instanz liegt der eigentliche Spray-Quellcode. Auf dieser Ebene liegen auch sämtliche Generatoren. Aus diesem Quellcode kann eine zweite Eclipse-Instanz generiert werden, in dieser Instanz werden z.B. die Shape- und Spray-Grammatik geschrieben und Eclipse stößt die entsprechenden Code-Generatoren an um Graphiti-Code zu generieren – bzw. für Spray Web: JSON Code. Dieser Graphiti Code kann wiederum als Eclipse-Instanz ausgeführt werden; dort ist dann der eigentliche grafische Editor auf Eclipse-Basis – eben der gewollte Graphiti Editor.

4.2.1 Shape Layout Definitionen

Es soll JSON-Code aus der Shape Grammatik (gemäß Warmer u. a., 2013) generiert werden. Der Code befindet sich im Repositorium in `generators/shape_dsl/ShapeGenerator.xtend`. Ziel ist es z.B. aus folgender Spray-Shape-Grammatik:

[7] Siehe `http://www.eclipse.org/Xtext/`

```
shape TransitionShape {
  text {
    position (x=0, y=0)
    size (width=30, height=30)
    id = transitionText
  }
  rectangle {
    position(x=0, y=30)
    size (width=40, height=40)
  }
}
```

Ein solches JSON zu generieren:

```
{
  name: ”TransitionShape”,
  params: {
    size: {witdh: 40, height: 70}
  },
  shapes: [
    {
      name: ”Text”,
      params: {
        size: {width: 30, height: 30},
        align: {
          horizontal: ”left”,
          vertical: ”top”
        }
      }
    },
    {
      name: ”Rectangle”,
      params: {
        size: {width: 40, height: 40},
      }
    }
  ]
```

```
}
```

JSON Shape Spezifikation Komplexe Shapes können aus primitiven Shapes zusammengesetzt werden. Es gibt folgende primitive Shapes:

- Line
- PolyLine
- Rectangle
- RoundedRectangle
- Polygon
- Ellipse
- Text
- Anchor

Diese komplexen Shapes werden in einer Datei `genshape.js` in einer JSON-Datenstruktur abgelegt. Die Datei enthält eine Variable, worin die Shapes in einer Liste abgespeichert werden:

```
var shapedefs = [
  {...},
  {...},
  ...
]
```

Jedes `{...}` Objekt entspricht genau einem komplexen Shape. Jedes dieser Shapes kann folgende Eigenschaften (JSON Properties) besitzen:

```
  name    : "..."
  params  : {...}
  anchors : [...]  (only in the top level!)
  shapes  : [...]
```

`name` beschreibt entweder das primitive Shape oder wenn es ganz oben in der Hierarchie steht den Namen des komplexen Shapes. In `params` können wieder folgende Attribute vorkommen:

```
position : [{x: Int, y: Int, radius: Int, angle: Int, offset: Float}*]
size : {width: Int, height: Int}
stretching : {horizontal, vertical}
align : {horizontal, vertical}
  horizontal : "left" | "center" | "right"
  vertical :  "top" | "middle" | "center"
curve : {width: Int, height: Int}
size-min : {width: Int, height: Int}
size-max : {width: Int, height: Int}
proportional : Bool
layout : {stretching | spacing : Int | margin : Int | invisible}
points : [{x: Int, y: Int, curveBefore: Int, curveAfter: Int}*]
```

Nur in der obersten Ebene der Hierarchie können Anker `anchors` definiert werden, diese betreffen also nur das gesamte komplexe Shape (Anker können also nicht verschachtelt werden). Es gibt diese Formen von Ankern:

```
{type: "center"}
{type: "corners"}
{type: "relative", x: Int, y: Int}
{type: "fixpoints", x: Int, y: Int}
```

Mit der `shapes` Property kann die Verschachtelung des Shapes beschrieben werden, es darf also die gesamte Shape-Definition hier nochmals (also rekursiv) vorkommen.

Folgende Parameter dürfen nur in der obersten Ebene der Hierarchie innerhalb der `params`-Liste vorkommen:

```
minWidth: Int
minHeight: Int
maxWidth: Int
maxHeight: Int
stretchH: Bool
stretchV: Bool
proportional: Bool
```

Zudem wird immer für das gesamte komplexe Shape eine Boundingbox berechnet, d.h. es existiert in der obersten Ebene immer ein `size`-Objekt innerhalb der `params`.

Compartments sind spezielle Shapes, die andere Shapes wiederum zur Laufzeit aufnehmen können. Es können nur Rectangle und Ellipse als Compartment markiert werden, d.h. sie enthalten innerhalb ihres `params` Property folgendes Attribut:

```
compartment: {
  locationId: String,
  layout: fixed|vertical|horizontal|fit,
  spacing: Int,
  margin: Int,
  insisible: Bool,
  stretchH: Bool,
  stretchV: Bool
}
```

Connections sind ebenfalls ein Spezialfall von Shape, welche sich jedoch explizit von den „normalen“ Shapes unterscheiden, denn sie können nur folgende Eigenschaften besitzen:

```
name: "..."
connectionType: "freeform" | manhatten
placings: [ {...}, {...}, ... ]
```

Ein `{...}`-Objekt welches in den `placings` vorkommt kann folgende Attribute enthalten:

```
offset: Double
distance: Double
angle: Double
shape: CDShape
```

Ein CDShape ist ein spezielles primitives Shape, welches für Connections zugelassen ist. Diese kommen auch nur in *nicht* verschachtelter Form vor:

- CDLine
- CDPolyLine
- CDRectangle

- CDRoundedRectangle
- CDPolygon
- CDEllipse
- CDText

4.2.2 Spray Logik Definitionen

Es soll JSON-Code aus der Spray Grammatik generiert werden. Mit diesen Definitionen kann Spray Web die Benutzereingaben überprüfen, ob diese zu ein zulässiges Modell führen. Der Code dazu befindet sich im Repositorium in `generators/spray_dsl/SprayJsonGenerator.xtend`. Ziel ist es z.B. aus folgender Spray-Logik-Grammatik:

```
class Place {
  shape PlaceShape {
    name into placeText
    reference tokens into tokensCompartment
  }
  behavior {
    create into mapelements palette ”Shapes”
  }
}
```

Solch ein JSON zu generieren:

```
{
  name: ”Place”,
  shape: ”PlaceShape”,
  compartments: [
    {
      canContain: ”Token”,
      atLocationId: ”tokensCompartment”,
    },
  ],
  palette: ”Shapes”,
}
```

JSON Spray (Logik) Spezifikation Die Spray Grammatik definiert ein Diagramm, welches sich in einzelne Klassen gliedert. Die Klassen spezifizieren welches Verhalten, welchem Shape zugeordnet wird bzw. welche Connections zwischen Shapes zulässig sind.

Diese Logik Definitionen werden in einer Datei `genspray.js` abgelegt. Die Datei enthält eine Variable, worin die einzelnen Klassendefinitionen in einer Liste gespeichert werden:

```
var classdefs = [
  {...},
  {...},
  ...
]
```

Es gibt zwei Sorten von `{...}`-Objekten. Die erste Sorte verbindet Shape und Klasse und kann folgende Eigenschaften besitzen:

```
  name: String
  shape: String
  compartments: [ {...}, ... ]
```

Das `compartments` Property enthält die Referenz, welches Shape in das Compartment gelegt werden darf, sowie das Shape welches der Compartment-Behälter sein soll:

```
  canContain: String
  atLocationId: String
```

Die zweite Sorte definiert, welche Klassen mit welchem Connection-Shape verbunden werden dürfen. Es kann folgende Eigenschaften besitzen:

```
  name: String
  connection: String
  from: String
  to: String
```

Wobei `connection` auf das Connectionshape referenziert, `from` referenziert von welcher Klasse aus die Connection ausgehen darf und `to` referenziert zu der Klasse zu der die Connection führen darf.

Jede der beiden Sorten muss das `palette` Propterty enthalten. Dieses definiert, unter welcher Auswahlpalette die Klasse im grafischen Editor zur Verfügung gestellt wird.

4.3 Validierung und Persistierung

Mit dem Spray Framework kann über drei Grammatiken ein grafischer DSL Editor definiert werden. Mit der Spray Grammatik wird spezifiziert, wie ein gültiges Modell aussieht.

Der Benutzer des so entstandenen grafischen Editors möchte eine Instanz dieses Modells erstellen, welche er speichern bzw. laden kann; zudem sollte vom Editor gewährleistet werden, dass der Benutzer nur gültige Modelle produzieren kann.

4.3.1 Validierung im Client

Der Client erhält die Regeln für ein gültiges Modell im JSON-Format (`classdefs`). Mit diesen Informationen wird die Shape Factory gefüttert und kann somit schon bei der Erstellung eines Shapes bzw. einer Connection überprüfen, ob dies eine zulässig Aktion ist.

4.3.2 Persistierungsverfahren

In diesem Kapitel werden drei Verfahren diskutiert, mit denen Modellinstanzspeicherung ermöglicht werden kann. Ein Verfahren wurde exemplarisch programmiert, da die anderen Verfahren ggf. noch Mängel aufweisen.

EMF-REST

> EMF-REST generates truly RESTful APIs for your EMF models. EMF-REST complements the existing (Java-based) API generation facilities EMF already provides and extends them to the Web.
>
> Thanks to EMF-REST, web developers can quickly get a JSON-based RESTful API derived from their models plus a JavaScript library to simplify the interaction with the API.[8]

[8] EMF-REST Webseite: `http://emf-rest.com/index.html`

EMF-REST stellt ein Eclipse-Plugin bereit, welches aus einem Ecore Metamodell Quellcodedateien generieren kann, die mit einem Tomcat Server lauffähig sind. Neben der RESTful API wird auch noch ein passendes JavaScript-Binding generiert.

Bisher kann man jedoch nur eine schon existierende XMI Modellinstanz via RESTful Abruf *lesen*; das *Schreiben* mit z.B. einem HTTP-PUT ist noch nicht möglich.

Dazu schrieben wird eine E-Mail an das Entwicklerteam, Camilo Alvarez antwortete:

> [...] we are implementing the PUT/POST/DELETE methods to complete the API. There will be an option to save the changes in the XMI file. We plan to release it in february of the next year.

Das bedeutet diese wichtigen Funktionen stehen vorr. erst Februar 2014 bereit. Daher haben wir uns vorerst gegen EMF-REST entschieden, auch wenn es ein sehr vielversprechender Ansatz gewesen wäre.

Ecore.js Ecore.js ist eine Ecore (EMOF) Implementierung in JavaScript von Guillaume Hillairet. Der Quellcode ist verfügbar auf GitHub unter `https://github.com/ghillairet/ecore.js`. Ecore.js läuft sowohl als JavaScript Clientcode (Browser), als auch als node.js Servercode.

Der Vorteil hier ist, dass quasi keinerlei Kommunikation zu einem Server nötig ist. Es ist möglich aus den Ecore Objekten die in JavaScript gehalten werden, direkt ein XMI oder JSON erstellen zu lassen. Das würde bedeuten Spray Web könnte komplett autonom, also ohne externe Abhängigkeiten oder Synchronisierungsprobleme, im Browser arbeiten. Eine Kollaboration könnte dennoch mit z.B. Together.js[9] erfolgen.

Beispielcode der durch unsere Nachforschungen entstanden ist, wird im Repositorium unter `archive/ecore_js` festgehalten.

Schlussendlich haben wir uns dagegen entschieden, da die Kollaboration über den Server von den Projektauftraggebern bevorzugt wurde und Ecore für die Java-Platform von einer starken Gemeinschaft gepflegt wird.

Ecore mit Server Als Serverlösung wurde ein Web2Ecore-Interface implementiert, welches durch Spray für jedes Modell automatisiert generiert

[9] `https://togetherjs.com/`

werden muss. Die Ecore Serverlösung ist mit Reflections weitgehend dynamisch gehalten. Der Spray Generator muss nur einige wenige Anpassungen hinsichtlich Modelltyps (z.B. Petrinet) und der Imports der generierten Implementierungsklassen des Ecore Modells vornehmen.

Jede Aktion im Webeditor betreffend Erstellen und Löschen, sei es ein Compartment, Connection oder ein einzelnes Shape, wird über eine Websocketnachricht dem Web2Ecore-Interface auf dem Server gemeldet. Das Interface verarbeitet die Benachrichtigung und aktualisiert entsprechend das Ecore-Modell auf dem Server.

Das eigens entwickelte Ecore-Interface unterstützt das Erstellen von Ecore-Objekten auf allen Ebenen. Wird ein Objekt auf zweiter oder tieferer Ebene erstellt, sprich Compartment, so muss immer das Elternelement mitangegeben werden. Das Ecore-Interface unterstützt auch die Erstellung von Connections. Dafür muss bekannt sein welche Connection erstellt werden soll und dessen Start und Zielobjekt.

Ebenso kann das Ecore-Interface Objekte löschen. Alle Fälle sind berücksichtigt: Löschen eines einzelnen Objekts auf erster Ebene, sowie auf allen nachfolgenden Ebenen (Compartment) und das Löschen von Connections. Wird ebenso ein Objekt gelöscht, was als Start oder Zielobjekt einer Connection agiert, so wird auch die Connection aus dem Ecore-Modell gelöscht.

Damit das erstellte oder gelöschte Shape in der Weboberfläche auch dem Shape im Ecore-Modell entspricht, ist ein Mapping zwischen den beiden Elementen nötig. Das Mapping erfolgt über einen Index. Jedes Element in der Zeichenfläche besitzt einen Index welche der Erstellungsreihenfolge entspricht. Bei einem Compartment muss der Index des Elternelements mitangegeben werden. Wird ein Shape innerhalb eines Shapes erstellt (Compartment) so wird der Index wieder von vorne gezählt, dazu muss zusätzlich der Index ders Elternelements mitangegeben werden. Das Mapping ist keine triviale Angelegenheit und muss optimiert werden. Die Berechnung des Index kann unter Umständen unnötig viel Rechenzeit beanspruchen.

Da nur valide Modelle in Spray Web gezeichnet werden können, ist auch das Ecore-Modell stets valide. Das Ecore-Modell wird als XMI-Dokument abgespeichert und enthält lediglich die Objekte ohne Eigenschaften zur Zeichenfläche. Die Zeichenfläche wird in JSON festgehalten und beinhaltet alle Informationen die der User selbst ändern kann. Das umfasst die Position der

Elemente, deren Skalierung, das Label und über welche Anker welche Connection gesetzt sind. Die Speicherung der Compartments sind an dieser Stelle noch nicht implementiert worden. Die Zeichenfläche muss zusätzlich zum Ecore-Modell auf dem Server persistiert werden. Die Persistierung der Zeichenfläche muss bei jeder Benutzeraktion erfolgen, die auch eine Aktualisierung des Ecore-Modells bewirkt. Wenn der Web Editor neu geladen wird, so muss der letzte Zustand der Zeichenfläche vom Server an den Client gesendet, im Editor geladen und somit dargestellt werden. Ansonsten würde der Benutzer beim Neuladen des Editors mit einer leeren Zeichenfläche starten, das zugrundeliegende Ecore-Modell ist aber in einem anderen Zustand. So wäre das Ecore-Modell nicht mehr mit der Zeichenfläche konsistent. Zukünftig muss an dieser Stelle eine sinnvolle Synchronisierung implementiert werden, so dass Ecore-Modell und Zeichenfläche auf dem Server zu jedem Zeitpunkt konsistent sind.

In Spray Web wird die Zeichenfläche erst beim expliziten Speichern (Klick auf das Speichern-Icon) auf dem Server persisistiert. Unter Umständen kann es aber sein, dass der Benutzer nach dem Speichern noch Aktionen tätigt, die das Ecore-Modell aktualisiert. Wenn der Benutzer nun die Zeichenfläche neu lädt, so stimmt sie nicht mehr mit Ecore-Modell überein.

Durch die umgesetzte Web2Ecore-Lösung bleibt das Ecore-Modell stets auf dem Server aktuell. Damit ist ein wesentlicher Baustein für Multikollaboration gesetzt. Jeder Benutzer der mit derselben Zeichenfläche arbeitet, verwendet so auch das gleiche zugrunde liegende Ecore-Modell.

4.4 Änderungen am Framework (Patches)

Draw2D touch deckt nicht alle Funktionen ab um die benötigten Anforderungen an Spray Web zu erfüllen. Aus dem Grund musste das Draw2D touch Framwork an einigen Stellen erweitert werden. So weit es das Framwork zugelassen hat, wurden Klassen abgeleitet um das Framework so wenig wie möglich unverändert zu lassen. Somit kann das Framework problemlos aktualisiert werden.

An einigen Stellen ist komponentengetrenntes Entwickeln nicht realisierbar, da abstrakte Klassen auch implementierte Methoden besitzen und JavaScript keine vollständige objekt-orientierte Programmierung unterstützt. Unter Umständen hätten ganze Vererbungshierarchien abgeleitet werden

müssen um Basis-Funktionalität vom Framework getrennt zu erweiteren.

Die Differenz von der Originalversion 3.0.0 und den Änderungen von uns kann mit dem Befehl `git diff 57a7d13 -- sprayServer/public/javascripts/spray/lib/draw2d_3.0.0/src` ermittelt und visualisiert werden.

5 Nächste Ausbaustufen

Hier werden für ggf. künftige Teamprojekte interessante Aufgaben im Bezug zu Spray Web vorgestellt und kurz beschrieben.

Zeichentoolkits Obwohl Draw2D touch viele Vorteile bietet wie Open Source Lizenz, ein umfassendes Framework und eine z.Z. aktive Entwicklung, gibt es auch ärgerliche Aspekte, wie z.B. fehlende Funktionalitäten oder Bugs oder auch dass die Software nicht öffentlich von einer Gemeinschaft gepflegt wird, sondern nur von Andreas Herz.

Gibt es besseren Alternativen? Wäre eine nicht Open Source Lösung denkbar? Die Frameworks go.js, diagram.js sind mit Sicherheit Alternativen. Oder gibt es andere potentielle Frameworks? Oder eine schlanke Eigenentwicklung die genau auf Sprays Bedürfnisse zugeschnitten ist?

Alternativ dazu, könnte man die Änderungen wieder vernünftig in den regulären Draw2D touch Quellcode-Baum einpflegen oder eine Kooperation mit Andreas Herz eingehen, um das Framework von sich noch stärker an die Bedürfnisse von Spray auszurichten.

Pflege Codebasis Die bisherige Codebasis ausbauen, verbessern, stabilisieren und testen bzw. allgemeine Verbesserungen der Architektur insbesondere auf JavaScript-Seite. Ein automatisiertes Testsystem[10] aufbauen, sowie eine solide Dokumentation auf Basis von z.B. Sphinx oder Docco.

Portierung des Codes von Draw2D touch Version 3.0.0 auf die akuellste Version. Können bessere Abstraktionen als bisher gefunden werden, um den Draw2d-Patchcode leichter in neue Versionen zu überführen? Oder kann man sich den Patchcode gänzlich sparen bzw. komplett von extern injizieren?

[10] jasmine.js, vows.js, grunt.js, phantom.js, testling.com, yeoman.io

Wie könnte eine neue Funktion wie Zooming eingebaut werden? Am besten so, dass man sich wie in Google Maps durch das Modell hangeln und zoomen kann.

Standalone Wenn das Modell mit z.B. ecore.js als JSON gespeichert werden würde, könnte der Spray Web Editor vollständig ohne Server, d.h. auch ohne Kommunikation zwischen Server und Client, auskommen.

Das hat den Vorteil, dass ein solcher Modellierungseditor ganz leicht in Webseiten integriert werden kann, ohne aufwändige Konfiguration und Installation. Ein Webseitenbetreiber müsste nur seinen existierenden Server pflegen und könnte mit diesem auf dem bevorzugten Weg direkt mit dem Editor kommunizieren. Aber gleichzeitig bleibt der Editor so portabel, dass er einfach auf ein USB-Stick geladen werden und ohne irgendwelche Hilfsmittel ausgeführt werden kann.

Beim Modellieren mit einem grafischen Editor kann es passieren, dass das Modell sehr groß wird, es wäre daher sehr sinnvoll wenn das Modell gut dokumentiert werden kann. Daher könnte ein Spray Web Editor in einem *Nur-Anzeige-Modus*, der standalone auf einer Dokumentationswebseite läuft, sehr nützlich sein. Für Dokumentationszwecke wäre es sicher auch sinnvoll, wenn der Editor auch nur *Teilmengen des Modells* anzeigen kann.

Kollaborationsfähigkeit Eine weitere Ausbaustufe ist es den Server mit Kollaborationsfunktionalität auszubauen. Damit können mehrere Benutzer parallel auf der gleichen Zeichenfläche arbeiten. Die bisherige Implementierung ist für Kollaboration in Grundzügen vorbereitet. Die Websocketkommunikation sorgt bisher für die Persistierung der Zeichenfläche auf dem Server und der Aktualisierung des Ecore-Modells.

Eine Idee ist es mehrere Benutzer auf dem Server anzulegen, welche einen Speicherplatz für ihre Modelle haben, aber auch das Modell gleichzeitig mit anderen Benutzer zu bearbeiten. Dank dem verwendeten Webframework Play und Scala kann Akka verwendet werden. Mit Akka lassen sich verteilte und nebenläufige Applikationen entwickeln. Die implementierte Websocketkommunikation lässt sich hervorragend mit Akka verknüpfen. Die Herausforderung ist es die Synchronizität zwischen den Benutzer zu implementieren. Draw2D touch liefert einen Commandstack mit, der für diesen Zweck verwendet werden kann. Commands könnten zwischen den Benutzer geschickt werden um

die Zeichenfläche auf allen Clients zu aktualisieren. Als Vorbild sei hier Google Docs erwähnt.

Die andere Möglichkeit wäre mit der Together.js JavaScript-Bibliothek, eine Kollaboration zu verwirklichen.

Validierung ausbauen Gibt es andere Möglichkeiten zu validieren oder zu persistieren? Ist es möglich und wäre es sinnvoll den Client komplett standalone zu machen? Wie kann man Modelle importieren, exportieren bzw. wie unter Versionskontrolle stellen?

Spray Integration Den bisherigen Code in Spray vernünftig integrieren, mit Tests versehen, sowie den Developer- bzw. User-Guide entsprechend erweitern sind weitere Aufgaben. Ein weiteres Teamprojekt hat bereits die Spray-Grammatik verbessert, hier könnte man den Generator auf die neue Spray-Grammatik portieren.

Generierung und Integration der Styles Die Grammatik der Spray Styles wird nicht vom Generator nach JSON konvertiert. Dieses Feature fehlt komplett. Mit Ausnahme von Stricharten (dot, dash-dot, etc.) und Positionierung der Labels, vertikal sowie horizontal, sind keine Styles implementiert.

Nicht umgesetzte Features Einige Features sind nicht umgesetzt worden oder müssen optimiert werden. Die folgende Auflistung soll einen Überblick verschaffen.

- Bedingt durch das Draw2D touch Framwork können Shapes, die zur Connection gehören, nicht rotiert werden.
- Speichern/Laden der Compartments funktioniert experimentell.
- Bessere Synchronisierung zwischen Ecore-Modell und der Zeichenfläche.
- Optimiertes Mapping zwischen Elementen des Ecore-Modells und der Zeichenfläche.
- Die Persistierung des Zeichenflächenmodells könnte durch Commands gelöst werden. Anstatt die Zeichenfläche vollständig zu übertragen, würden sich Commands besser eignen, da diese wesentlich kürzer sind.

- Undo/Redo für Label ändern. Dieses Feature wird von Draw2D touch nicht unterstützt.
- Im Petrinetz Beispiel ist ein Shape (Edge/Place) auch ein Anker. Dieser Umstand wurde mit 4 Ankern am Rand des Shapes gelöst.
- Labels sind manchmal fehlerhaft bzw. schwierig zu editieren. Das könnte man noch „runder“ machen.
- Web2Ecore-Interface muss vom Spray Generator für das entsprechende Projekt erzeugt werden.
- Weitere fehlende Features z.B. Rapidbuttons.

6 Zusammenfassung

Spray ist ein Framework mit dem automatisiert grafische Editoren für quasi beliebige Domänen produziert werden können. Das bedeutet, dass der Spray Web Editor eine gewisse Flexibilität mitbringen muss, um anhand eines von Spray automatisch generierten Regelwerk alle essentiellen Editoreigenschaften bereitstellen zu können.

Ziel der Projektaufttraggeber war es einen funktionierenden grafischen Spray Web Editor zu erhalten. Das Modell soll nach vorgegebener Grammatik aus Spray im Web Editor gezeichnet werden. In einem ersten Schritt wurden Generatoren implementiert welche Spray-Grammatiken (Beispiele Petrinetz und Heizkesselsystem) via Xtext nach JSON konvertiert, damit der Webeditor die Grammatik lesen kann. Diese JSON Dateien dienen als Regelwerk, anhand dessen Spray Web sämtliche Shapes, Connections und die Modellvalidierung ableitet. Der Benutzer des Spray Web Editors kann aus den Shapes, Connections und Compartments ein Modell „zusammenklicken“. Dieses Modell wird nach einem vorgegebenen Regelwerk auf Korrektheit überprüft, d.h. der Benutzer ist nur in der Lage valide Modelle zu erstellen. Zudem kann das so entstandene Modell sowohl als Zeichenfläche als auch als Ecore-Modell über das implementierte Web2Ecore-Interface auf dem Server gespeichert werden. Dazu ist eine Webserverkommunikation mit Websockets realisiert worden. Damit das Ecore-Modell tatsächlich mit der Zeichenfläche konsistent ist, muss jedes Ecore-Element eindeutig seinem Zeichenflächen-Element zuordenbar sein. Die Zuordnung wird mit einem Mapping des Index realisiert.

All diese Anforderungen wurden im Zuge dieses Teamprojekts erfasst und konkretisiert. Es wurden Nachforschungen angestellt, wie man die gesammelten Anforderungen umsetzen kann. Dabei hat sich das Draw2D touch-Framework auf Basis von JavaScript als geeignet herausgestellt. Dieses dient nun als Grundlage für Spray Web. Es übernimmt hauptsächlich die Aufgabe Shapes, Connections und Compartments zu zeichnen.

Das Teamprojekt konnte all diese Schritte letztendlich erfolgreich umsetzen. Dazu gehörte die Einarbeitung in die Thematik, Zusammentragen der Anforderungen, Recherchearbeit zu Technologien, Überlegungen zur Architektur, die praktische Erstellung eines funktionierenden Prototyps, sowie die entsprechende Dokumentation nach wissenschaftlichen Maßstäben.

Literatur

[Drachenfels 2013] Drachenfels, Heiko: *Modellgetriebene Softwareentwicklung – Teil 2: Modellierung*. http://www-home.htwg-konstanz.de/~drachen/mgse/Teil-2.pdf. Version: 12. August 2013, Abruf: 2014-01-09. – Vorlesungsskript

[Filippelli u. a. 2012] Filippelli, Fabio ; Kollosche, Steffen ; Bauer, Michael ; Gerhart, Markus ; Boger, Marko ; Thoms, Karsten ; Warmer, Jos: *Concepts for the model-driven generation of graphical editors in Eclipse by using the Graphiti framework*. Version: 2012. https://spray.eclipselabs.org.codespot.com/files/SprayPaper.pdf, Abruf: 2014-01-09

[Herz 2013] Herz, Andreas: *API Documentation – Draw2D touch JavaScript GraphLib (version 3.0.0)*. Version: 19. Oktober 2013. http://draw2d.org/draw2d_touch/jsdoc, Abruf: 2014-01-05

[Spray Webseite 2014] *spray – A quick way of creating Graphiti*. https://code.google.com/a/eclipselabs.org/p/spray/. Version: 2014, Abruf: 2014-01-09

[Warmer u. a. 2013] Warmer, Jos ; Thoms, Karsten ; Boger, Marko ; Filippelli, Fabio ; Bauer, Michael ; Reichert, Joerg: *Spray User Guide*. Version: 2013. https://spray.ci.cloudbees.com/job/spray-docs-build/lastSuccessfulBuild/artifact/docs/org.eclipselabs.spray.doc.user/docs/html/SprayUserGuide.html, Abruf: 2014-01-05