

# Toolgestütztes Event Storming für das Requirements Engineering

fulibWorkflows

## B A C H E L O R A R B E I T

zur Erlangung des Grades eines Bachelor of Science  
im Fachbereich Elektrotechnik/Informatik  
der Universität Kassel

Eingereicht von:	Maximilian Freiherr von Künßberg
Anschrift:	Mönchebergstraße 31 34125 Kassel
Matrikelnummer:	33378673
Emailadresse:	maximilian-kuenssberg@uni-kassel.de
Vorgelegt im:	Fachgebiet Software Engineering
Gutachter:	Prof. Dr. Albert Zündorf Prof. Dr. Claude Draude
Betreuer:	M. Sc. Sebastian Copei
eingereicht am:	22. Februar 2022

# Kurzfassung

Diese Bachelorarbeit befasst sich mit der Methodik des Event Stormings zur Erarbeitung von Anforderungen in der Softwareentwicklung. Ziel der Arbeit ist es den Prozess des Event Stormings durch ein Tool zur Erfassung von Abläufen und daraus während dieses Prozesses zusätzliche Mockups und Diagramme bereitzustellen. Hierfür wird in dieser Arbeit eine Beschreibungssprache für Events, auf Grundlage des Event Stormings, erstellt. Weiterhin wird zur Nutzung dieser Sprache ein Web-basierter Editor entwickelt, in welchem diese Beschreibung vorgenommen werden kann und die generierten Mockups und Diagramme angezeigt werden können.

# Inhaltsverzeichnis

<b>Kurzfassung</b>	<b>I</b>
<b>Inhaltsverzeichnis</b>	<b>II</b>
<b>Listings</b>	<b>IV</b>
<b>Abbildungsverzeichnis</b>	<b>V</b>
<b>Abkürzungsverzeichnis</b>	<b>VI</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Ziele . . . . .	2
1.3 Existierende Konzepte . . . . .	2
1.4 Aufbau der Arbeit . . . . .	2
<b>2 Grundlagen</b>	<b>4</b>
2.1 Methoden . . . . .	4
2.1.1 Domain-Driven Design . . . . .	4
2.1.2 Event Storming . . . . .	5
2.1.3 Erweiterung . . . . .	6
2.2 Technologien . . . . .	7
2.2.1 fulibWorkflows . . . . .	7
2.2.2 Frontend des Web-Editors . . . . .	14
2.2.3 Backend des Web-Editors . . . . .	16
2.2.4 Deployment . . . . .	17
<b>3 Implementierung</b>	<b>19</b>
3.1 fulibWorkflows . . . . .	19
3.1.1 Workflow-Format . . . . .	19
3.1.2 JSON-Schema . . . . .	23
3.1.3 Antlr-Grammatik . . . . .	26
3.1.4 Generierung von Dateien . . . . .	30
3.2 Frontend des Web-Editors . . . . .	37
3.2.1 Code-Editor . . . . .	39
3.2.2 Darstellung generierter Dateien . . . . .	41

---

3.3	Backend des Web-Editors . . . . .	44
<b>4</b>	<b>Evaluation</b>	<b>46</b>
4.1	Expertengespräch . . . . .	46
<b>5</b>	<b>Fazit</b>	<b>48</b>
<b>6</b>	<b>Ausblick</b>	<b>50</b>
<b>7</b>	<b>Quellenverzeichnis</b>	<b>52</b>
<b>Anhang</b>		<b>A</b>
7.1	Workflow-Beispiel . . . . .	A
7.2	Repositories . . . . .	B
7.3	Aufnahme des Expertengespräches . . . . .	C

## Listings

1	Beispiel einer einfachen Grammatik in Antlr . . . . .	8
2	Eingabe . . . . .	8
3	“Hello World!” - Beispiel mittels StringTemplate . . . . .	9
4	Beispiel einer .stg-Datei . . . . .	10
5	Nutzung einer STG-Datei in Java . . . . .	11
6	STG-Ausgabe auf Konsole . . . . .	11
7	Objekt-Beispiel eines JSON-Schemas . . . . .	12
8	Begrenzung der Properties eines Schemas . . . . .	12
9	Listen-Beispiel eines JSON-Schemas . . . . .	13
10	Beispiel aller vorhandenen “Post-its” . . . . .	20
11	Referenzieren eines anderen Schemas . . . . .	24
12	Grammatik für Workflows . . . . .	26
13	Grammatik für Notes . . . . .	27
14	Schlüsselwörter zum Identifizieren von Notes . . . . .	27
15	Grammatik von Attributen . . . . .	27
16	Grammatik von Werten . . . . .	28
17	Grammatik einer Page . . . . .	28
18	exitPage-Methode . . . . .	30
19	FulibYaml.stg . . . . .	33
20	Generierungsmethode eines Objektdiagramms . . . . .	34
21	Beispiel eines richtigen Data-Notes . . . . .	35
22	Schritte zum Aufbau eines Klassenmodells . . . . .	36
23	Codemirror-Konfiguration . . . . .	39
24	Modell der vom Backend empfangenen Daten . . . . .	42
25	Bereitstellung einer Methode . . . . .	43
26	Definition der Endpunkte . . . . .	44
27	pm.es.yaml . . . . .	B

# Abbildungsverzeichnis

2.1	Event Storming (ES)-Board . . . . .	5
2.2	Interner Parse Tree . . . . .	8
2.3	Spring Initializr für eine Web-Anwendung . . . . .	16
3.1	Fehleranzeige in IntelliJ . . . . .	25
3.2	Autovervollständigung in IntelliJ . . . . .	25
3.3	Klassendiagramm fulibWorkflows . . . . .	29
3.4	Mittels fulibWorkflows generiertes Event-Storming-Board . . . . .	31
3.5	Mittels fulibWorkflows generierte Mockups . . . . .	32
3.6	Mittels fulibWorkflows generierte Objektdiagramme . . . . .	35
3.7	Mittels fulibWorkflows generiertes Klassendiagramm . . . . .	37
3.8	Oberfläche des Web-Editors für fulibWorkflows . . . . .	38
3.9	Download-Fenster . . . . .	38
3.10	Themes des Codemirrors . . . . .	40
3.11	Autovervollständigung . . . . .	40
3.12	Validierungsserror als Toast . . . . .	41
3.13	Event-Storming-Board in einem IFrame . . . . .	42
3.14	Inhalt eines heruntergeladenen Zip-Archivs . . . . .	45
7.1	QR-Codes der für diese Arbeit erstellten Repositories . . . . .	C
7.2	QR-Code zur Aufnahme des geführten Expertengesprächs . . . . .	C

# Abkürzungsverzeichnis

**Antlr** Another-Tool-for-Language-Recognition

**CPaaS** Cloud Platform as a Service

**DDD** Domain-Driven Design

**DSL** Domain Specific Language

**ES** Event Storming

**RE** Requirements Engineering

**SDK** Software Development Kits

**ST** String-Template

**STG** String-Template-Group

**YAML** YAML Ain't Markup Language

# 1 Einleitung

*Zwei kleine Änderungswünsche (jetzt, wo ich es sehe) ...*

Variationen dieses Satzes<sup>1</sup> begegnen Softwareentwicklern bei der agilen Softwareentwicklung für eine personalisierte Kundenanwendung häufiger. In der agilen Softwareentwicklung können sich Anforderungen an die Anwendung stetig verändern. Neue Features werden vom Kunden ausprobiert, wodurch schnelles Feedback entsteht. Dies verbessert zwar das Endprodukt, allerdings sorgt diese Art der Entwicklung für einen Mehraufwand aufseiten der Entwickler. In diesem Prozess ist die Meinung und technische Abschätzung der Entwickler unerlässlich. Allerdings stellt dies auch eine Herausforderung für Softwareentwickler dar, sollten diese keine Erfahrung im Requirements Engineering (RE) haben. In Gesprächen mit dem Kunden die korrekten Anforderungen zu verstehen und umzusetzen stellt ohne eine passende Methodik zum RE ein Problem dar.

## 1.1 Motivation

Um komplexe Arbeitsabläufe zu Beginn einer Softwareentwicklung zu verstehen, existiert die Methode des Event Stormings von Alberto Brandolini. Hierbei setzen sich Entwickler und Domänenexperten zusammen und erarbeiten in einem Workshop die in der Software darzustellenden Arbeitsabläufe. Sobald eine Anwendung Benutzeroberflächen enthält, benötigt es ebenso Mockups, wodurch die Entwickler Vorschläge für eine Benutzeroberfläche darlegen können, um ein schnelles Feedback der Kunden zu erlangen. Diese beiden Methodiken können die soeben beschriebenen Probleme lösen. Doch während dem Event Storming können bereits Aussagen über eine mögliche Benutzeroberfläche getroffen werden. Eine Kombination aus den beiden Methodiken könnte somit einen größeren Nutzen mit sich ziehen. Aus dieser Idee resultiert das Thema dieser Arbeit und der Versuch das Event Storming sinnvoll zu erweitern beziehungsweise den Prozess zu unterstützen.

---

<sup>1</sup>, welcher von einem Kunden aus einem Drittmittelprojekt stammt



## 1.2 Ziele

Aus der Motivation lassen sich mehrere Ziele extrahieren:

- Entwicklung einer Beschreibungssprache für Event Storming (ES)
- Verarbeitung der Beschreibungssprache in benutzerfreundliche Formate
- Entwicklung einer Web-Anwendung zur Erstellung einer ES-Beschreibung
- Darstellung der verarbeiteten Beschreibung in der Web-Anwendung

Das Ergebnis dieser Arbeit soll alle vorherigen Ziele erfüllen, somit soll eine Web-Anwendung entwickelt werden, welche den Ablauf eines ES-Workshops unterstützt. Zusätzlich soll auch nach einem solchen Workshop die Möglichkeit vorhanden sein auf diese Daten zuzugreifen und Aktualisierungen im Verlauf der Entwicklung vornehmen zu können.

## 1.3 Existierende Konzepte

Event Storming, wie es klassisch von Alberto Brandolini entwickelt wurde, basiert auf der Arbeit in einem Raum mit vielen Post-its. Dadurch kann jede Person, welche am ES teilnimmt, frei und jederzeit neue Events erstellen und anbringen. Sollte ein Zusammenkommen in einem Raum nicht möglich sein, sei es durch internationale Teilnehmer oder, wie es momentan der Fall ist, durch eine Pandemie, welche aufgrund von Sicherheitsmaßnahmen kein Treffen mit größeren Gruppen ermöglicht, können Online-Tools genutzt werden. Ein Beispiel für ein interaktives Online-Board ist Miro.<sup>2</sup> Hierbei kann jeder Teilnehmer ebenfalls selbstständig neue Events erstellen und auf dem Board platzieren. Zudem können Kommentare zu bestehenden Events hinzugefügt werden. Diese Aktualisierungen werden für alle Teilnehmer synchronisiert.

## 1.4 Aufbau der Arbeit

In Kapitel 2 werden zuerst die Grundlagen des ES und die verwendeten Technologien zur Erstellung der zuvor erwähnten Anwendungen erläutert. Hierbei sind die Technologien entsprechend der zugehörigen Anwendung zugeordnet. Anschließend befasst sich Kapitel 3 mit der Implementierung der Anwendungen. Dabei wird genauer auf die Anwendung der aus

---

<sup>2</sup><https://miro.com/>

Kapitel 2 erläuterten Technologien eingegangen. Um den Nutzen der erstellten Anwendungen zu prüfen, wird in Kapitel 4 eine Evaluierung mittels Expertengespräch durchgeführt. Die zuvor aufgestellten Ziele sollen somit, gemeinsam mit einem Experten, als erfüllt oder nicht erfüllt evaluiert werden. Im folgenden Kapitel 5 wird, unter Einbezug der in der Evaluation erlangten Erkenntnisse, ein Fazit zu den gestellten Fragen und Zielen erstellt. Zusätzlich soll die Umsetzung der zuvor festgelegten Funktionen anhand der Implementierung überprüft werden. Zuletzt wird in Kapitel 6 ein Ausblick zur weiteren Entwicklung der Anwendung gegeben. Unter anderem wird auch der mögliche Nutzen in der Lehre angesprochen.

## 2 Grundlagen

Im folgenden Kapitel werden zuerst die grundlegenden Konzepte des *Event Stormings* erläutert. Hierbei wird auf dessen Herkunft und Entwicklung eingegangen. Neben diesen Grundlagen werden anschließend die für diese Arbeit notwendigen Änderungen und Erweiterungen dargelegt. Weiterhin werden die wichtigsten Technologien erläutert, welche für die Implementierung der Anwendungen nötig sind. Um eine bessere Übersicht zu schaffen, sind die Technologien ihrem jeweiligen Anwendungsteil zugeordnet.

### 2.1 Methoden

In diesem Unterkapitel werden die grundlegenden Prinzipien und Ziele des Domain-Driven Design (DDD) erläutert, welche Vaughn Vernon in seinem Buch *Domain-Driven Design Distilled* definiert hat[Ver16]. Nachdem diese Grundlage vorhanden ist, wird darauf aufbauend erklärt, welche Symbiose aus dem DDD und dem ES entsteht und wie dies zu einer Softwareentwicklung beiträgt. Abschließend werden die neuen Ansätze zum ES, welche im Kontext dieser Arbeit vorgenommen wurden, erklärt.

#### 2.1.1 Domain-Driven Design

Das DDD ist nicht nur für die erste Phase der Softwareentwicklung praktisch, sondern ebenso für das Umstrukturieren bestehender Projekte. Ein grundlegendes Ziel des DDD ist es ein Projekt in sogenannte *Bounded Contexts* zu unterteilen und damit zu umgehen, dass die Anwendung aus einem riesigen aufgeblähten Modell besteht. Um dieses Ziel zu erreichen, ist es wichtig in Gesprächen mit Domänenexperten die wichtigsten Punkte eines *Bounded Context* zu evaluieren. Domänenexperten können in jedem Bereich eines Unternehmens gefunden werden. Es ist nötig ein möglichst breites Spektrum an Personen zu haben, um den gesamten zu entwickelnden Prozess zu verstehen und für die Entwickler verständlich zu machen. Dabei ist es wichtig, dass alle Personen, welche am Prozess des DDD teilnehmen, eine einheitliche Sprache entwickeln. Diese einheitliche Sprache beschreibt Vernon als *Ubiquitous Language*<sup>1</sup>[Ver16]. Diese Sprache zu entwickeln, ist eine fortlaufende Aufgabe für

---

<sup>1</sup>im Deutschen “allgegenwärtige Sprache”

die im Projekt angesiedelten Personen. Initial ist es wichtig, dass zwischen den verschiedenen Domänenexperten und den Entwicklern diese allgegenwärtige Sprache entsteht, welche nicht nur das Verständnis zwischen den beiden Parteien verbessern, sondern auch mit in das Modell einfließen soll.

### 2.1.2 Event Storming

Vernon selbst nennt Event Storming als mögliche Methode, um eine *Ubiquitous Language*[Ver16] zu entwickeln. Event Storming wurde von Alberto Brandolini entwickelt und resultiert aus mangelnder Zeit während der Nutzung von *event-driven modeling*. *Event-driven modeling* basiert ebenfalls auf Konversationen und konkreten Szenarien, allerdings mit der Verwendung von UML-Diagrammen zur Datenmodellierung. Dies hatte zur Konsequenz, dass in Gesprächen zur Anforderungsanalyse ab einem bestimmten Punkt nur noch die Entwickler daran teilnahmen. Brandolini verwarf UML-Diagramme, verwendete stattdessen Haftnotizen und legte damit den Grundbaustein für das ES[Ver16].

In seinem Buch “Introducing EventStorming” beschreibt Brandolini mehrere ES-Workshops und wie diese durchgeführt wurden[Bra17]. Hierbei stellt sich heraus, dass Event Storming kein starres Konstrukt aus Abläufen ist, sondern je nach Kontext angepasst werden kann. Dennoch gibt es Ähnlichkeiten, welche eine solide Grundlage für einen ES-Workshop bieten[Bra17]. Neben einem grenzenlosen Platz zum Modellieren benötigt es genügend Marker und Haftnotizen in verschiedenen Formen und Farben. Die teilnehmenden Domänenexperten benötigen eine kollaborative Einstellung zur Modellierung, ein offenes Miteinander ungeachtet ihrer Stellung. Zudem sollte es zu Beginn eines Projektes und dem damit einhergehenden ES-Workshop keine Grenzen zu dem Thema oder der Anwendung welche modelliert werden soll geben. Dadurch sollen weitere Probleme gelöst oder Fragen beantwortet werden. Ein Event Storming beginnt immer mit dem Erstellen von Domain Events und dem Platzieren dieser anhand eines Zeitstrahles. Zudem müssen alle Beteiligten am fortlaufenden Verfeinern eines Modells interessiert sein, da ein ES-Workshop zum Lernen und Verbessern von Anwendungen gedacht ist. Ein ES-Board nach einem solchen Workshop ist in Abbildung 2.1 dargestellt[Gop21].



Abbildung 2.1: ES-Board

Nachdem initial Domain Events beim ES erstellt wurden, soll jedem ein Command vorausgesetzt werden. Ein Command fungiert hierbei als die Aktion, welche das Event ausgelöst hat. Dies können Aktionen eines Nutzers oder eines externen Systems sein. Während eines Workshops können sogenannte *Hotspots* in Gesprächen entdeckt werden. Dabei kann es sich um Probleme, aber auch um Fragen bezüglich des Ablaufs handeln, welche wichtig für die spätere Anwendung werden.

Wie auch das DDD kann Event Storming jederzeit während des Entwicklungsprozesses verwendet werden. Für die verschiedenen Anwendungsbereiche hat Brandolini mehrere Typen des Event Stormings definiert[Bra17]. Hierbei bleibt die Methodik an sich die gleiche, allerdings wird das Ziel genauer definiert.

**Big Picture EventStorming** wird während einem Kick-off-Meeting eingesetzt, damit alle Teilnehmer den Inhalt und den Bereich der zu erstellenden Anwendung kennenlernen. Hierzu ist es nötig, dass alle Interessengruppen vertreten sind, welche innerhalb des Unternehmens existieren und ebenfalls eine Entscheidungsgewalt innehaben.

**Design Level EventStorming** findet auf einer tieferen Ebene einen Einsatz. Dabei handelt es sich um das Erstellen möglicher Implementierungen, zum Beispiel, ob Event Sourcing oder andere Techniken aus dem DDD verwendet werden sollen. Es werden somit Entscheidungen getroffen, welcher in erster Linie die Entwickler betrifft und von diesen am besten zu bewerten ist.

**Value-Driven EventStorming** bietet einen Einstieg in die Wertstromanalyse (englisch: value-stream mapping). Anhand einer solchen Analyse ist es möglich den Erhalt von Informationen und der Verarbeitung dieser darzustellen und Probleme zu erkennen.

**UX-Driven EventStorming** konzentriert sich auf das Erlebnis eines Nutzers/Kunden bei der Benutzung der Anwendung. Hierbei wird neben der Nutzerfreundlichkeit auch die fehlerfreie Ausführung abgebildet und überprüft.

**EventStorming as a Retrospective** konzentriert sich darauf einen Ablauf über Domain Events zu definieren und nach Erweiterungsmöglichkeiten zu suchen, welche Vorteile für die Anwendung bereitstellen können.

**EventStorming as a Learning Tool** zeigt auch die weiteren Lernchancen innerhalb eines Unternehmens. Mittels Event Storming ist es somit auch möglich neue Angestellte möglichst schnell auf den aktuellen Stand zu bringen. Diese Art von Lernchancen ist ebenfalls im **Big Picture EventStorming** enthalten.

### 2.1.3 Erweiterung

Im Rahmen dieser Arbeit werden verschiedene Bereiche des Event Stormings vertieft. Zum einen ist dies das Verbildlichen von Arbeitsabläufen, im Folgenden “Workflows” genannt. Ein komplexeres System soll durch die Beschreibung von mehreren Workflows genauer betrachtet werden. Dies orientiert sich stark an dem zuvor beschriebenen **Big Picture**

**EventStorming**, da ein großer Prozess in mehrere kleine Prozesse aufgeteilt wird und daraus ein ES-Board entsteht, welches weiterhin den großen Prozess zeigt.

Zum anderen wurde bei der Entwicklung der Anwendung darauf Wert gelegt, die Generierung von Mockups zu ermöglichen. Durch diese Möglichkeit ist das zuvor beschriebene **UX-Driven EventStorming** ein weiterer Typ des Event Stormings, welcher hierbei Verwendung findet. Die Generierung von Mockups und rudimentären *Clickdummies* soll es ermöglichen, während einem Event Storming genauer auf Oberflächen eingehen zu können und daraus verschiedene Wege darzustellen, in welcher eine Anwendung genutzt werden kann.

Auf die Generierungsmöglichkeiten von Mockups und deren weitere Funktionen im Web-Editor wird in Kapitel 3 genauer eingegangen.

## 2.2 Technologien

Dieses Kapitel gibt einen Überblick über die verwendeten Technologien in der Umsetzung. Da die Implementierung aus verschiedenen Komponenten besteht, ist dieses Kapitel in drei weitere Unterkapitel aufgeteilt. Es wird somit getrennt auf die Java-Bibliothek *fulibWorkflows*, das Spring-Boot-Backend des *fulibWorkflows Web-Editors* und das zum Editor dazugehörige Frontend, welches mit Angular umgesetzt wurde, eingegangen.

### 2.2.1 fulibWorkflows

Bei *fulibWorkflows* handelt es sich um eine Java-Bibliothek, welche Workflows in YAML<sup>2</sup>-Syntax notiert, als Eingabe erhält und daraus sowohl ein ES-Board als auch im Workflow beschriebene Mockups und Objekt-/ Klassendiagramme generiert. Welche Form die YAML-Eingabe haben muss und wie die Dateien aussehen und generiert werden, wird in Kapitel 3.1.1 erläutert.

#### 2.2.1.1 Antlr

Another-Tool-for-Language-Recognition (Antlr) bietet die Möglichkeit einen Parser über eine eigens geschriebene Grammatik zu generieren. Die Grammatik muss linksableitend sein und ist in erweiterter Backus-Naur-Form definiert. Der generierte Parser ermöglicht zudem das Aufbauen und Ablaufen eines *Parse trees*. Bei einem *Parse tree* handelt es sich um einen Syntaxbaum, in welchem über eine hierarchische Struktur ein Text in mehrere

---

<sup>2</sup>Kurzform von “YAML Ain’t Markup Language”

Knoten unterteilt wird. Hierdurch ergibt sich die Möglichkeit, während dem Parsen weitere Aktionen durchzuführen, welche den späteren Programmablauf eines Tools unterstützen können.

---

```

1  grammar AntlrExample;
2
3  file: prop+ ;
4  prop: ID '=' value '\n' ;
5  value: STRING | ID ;
6
7  ID: [a-zA-Z]+ ;
8  STRING: '"' .*? '"' ;
9  WS : [ \t]+ -> skip ;

```

---

Listing 1: Beispiel einer einfachen Grammatik in Antlr

In Listing 1 ist ein Beispiel für eine einfache Grammatik zur Erkennung von Wertzuweisungen beziehungsweise dem Erstellen von Properties dargestellt[Par13b]. Es existieren drei nicht-terminale Regeln, welche für den generierten Parser wichtig sind, und den Gesamtaufbau einer Eingabe definieren. Die Grammatik kann mehrere Properties verarbeiten, welche aus einer ID, einem Gleichheitszeichen und einem Wert bestehen. Hierbei ist in Zeile 4 zu erkennen, dass nach jeder Property eine neue Zeile begonnen werden muss. Ein Wert kann entweder ein String oder eine ID sein. *STRING* und *ID* sind terminale Regeln, welche einen finalen Knoten für einen Ast des Parse Trees darstellen. Eine zulässige Eingabe für die festgelegte Grammatik aus Listing 1 ist in Listing 2 dargestellt.

---

```

1  name=parrt
2  title="Supreme dictator for life"
3

```

---

Listing 2: Eingabe

Die zuvor beschriebene Grammatik kann mittels weiterer Tools auf eine Eingabe geprüft werden. Eines dieser Tools ist das Antlr-Plugin für *IntelliJ*. Die Überprüfung auf die Richtigkeit einer Eingabe oder auch der Grammatik kann über das Plugin bereits vor einer Generierung von Code durchgeführt werden. Weiterhin ist es möglich einen Parse Tree zu generieren, was in Abbildung 2.2 betrachtet werden kann.

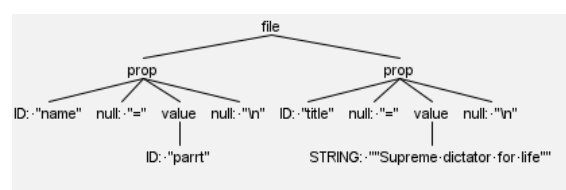


Abbildung 2.2: Interner Parse Tree

Hierbei ist ersichtlich, dass die Wurzel bei der obersten Regel **file** beginnt und alle weiteren Kindknoten durch **prop** Regeln kreiert wurden. Beide Properties besitzen eine ID und einen Wert, welcher durch ein Gleichheitszeichen und einen Zeilenumbruch umklammert ist. Der zugeordnete Wert ist bei den Properties dieses Beispiels allerdings unterschiedlich in *ID* und *STRING* aufgeteilt. Dies entsteht durch die Unterscheidung bei der Grammatik in Terminale und Nicht-Terminale. Hierbei werden wie in Listing 1 dargestellt nicht-terminale Regeln kleingeschrieben und terminale Regeln werden in Großbuchstaben verfasst. In der vereinfachten Grammatik sind lediglich ganze Zahlen als Eingabe erlaubt.

Dies sind einzig die Grundlagen von Antlr, auf die genauere Verwendung des generierten Parsers und Besonderheiten der Grammatik wird in Kapitel 3.1.3, anhand der Implementierung, eingegangen.

### 2.2.1.2 StringTemplate

Die Java-Bibliothek *StringTemplate* gehört, wie das vorherige Antlr, zum *Antlr Project*. Antlr verwendet ebenfalls String-Templates zur Generierung von formatiertem Text, im Folgenden als *Code* bezeichnet. Templates (übersetzt: Schablonen) ermöglichen es zum Beispiel die feste Syntax einer Programmiersprache mit variablen Werten für Variablen, Klassen und Methoden zu belegen. Somit können die generellen Bausteine einer Sprache beliebig gefüllt werden. Durch diese Funktionalität bieten sich String-Template (ST)s sehr gut zur Generierung von Dateien an. Ursprünglich ist *StringTemplate* eine Java-Bibliothek, jedoch wurden bereits Portierungen für C#, Objective-C, JavaScript und Scala erstellt.

Die folgenden Erläuterungen beziehen sich jedoch auf die ursprüngliche Java-Version von *StringTemplate*, da diese in dieser Arbeit verwendet wird. Die einfachste Möglichkeit für die Verwendung eines String Templates ist in Listing 3 zu sehen.

---

```
1  import org.stringtemplate.v4.*;
2  ...
3  ST hello = new ST("Hello, <name>!");
4  hello.add("name", "World");
5  String output = hello.render();
6  System.out.println(output);
```

---

Listing 3: "Hello World!" - Beispiel mittels StringTemplate

Die Klasse *ST* aus Zeile 3 kann mit einem String initialisiert werden. In diesem Beispiel wurden als Begrenzer für das zu ersetzende Stück des Textes die Zeichen "<" und ">" verwendet. Im Anschluss wird dem neuen *ST*-Objekt mithilfe der *add()*-Methode ein bestimmter Wert hinzugefügt. Der erste Parameter der Methode ist der Bezeichner innerhalb eines Templates, zu beachten ist die Angabe des Bezeichners ohne die Begrenzer. Der Wert



wird als zweiter Parameter übergeben und besitzt in diesem Beispiel den Text **World**. Um nun den fertig ersetzten Text aus dem Template und dem übergebenem Wert zu bekommen, muss auf dem *ST*-Objekt die Methode `render()` aufgerufen werden. Hierbei werden die Platzhalter des Templates durch den zuvor übergebenen Wert ersetzt und als String zurückgegeben. In Zeile 6 wird nun abschließend der fertige Text “Hello, World!” auf der Konsole ausgegeben. Dieses Beispiel entstammt der offiziellen Webseite von *StringTemplate*[Par13a].

Für ein strukturiertes Arbeiten mit vielen Templates bietet *StringTemplate* die Möglichkeit String-Template-Group (STG)s zu erstellen. Hierbei können mehrere Templates in einer Datei beschrieben werden, um aufeinander aufbauende Templates nicht im Code, sondern einer gesonderten Datei zu organisieren. In diesen Dateien, welche die Dateiendung `.stg` tragen, können die Begrenzer (eng.: Delimiters) frei gewählt werden. Dies ist je nach Kontext des Templates nötig, da zum Beispiel die Generierung von HTML-Dateien, welche “<” und “>” als Zeichen zum Abgrenzen von Bereichen verwenden, mit den Standard-Begrenzern für einen Mehraufwand sorgt. Bei der Wahl der Begrenzer sollte somit stets auf die Wahl der Zeichen im Kontext der zu generierenden Sprache geachtet werden. Zum Parsen einer STG wird ein mit Antlr generierter Parser verwendet[Ger15].

---

```
1  delimiters "{", "}"
2
3  example(topic, language) ::= <<
4  <span>
5      This test about {topic} is written in {language}.
6  </span>
7  >>
```

---

Listing 4: Beispiel einer .stg-Datei

Wie zuvor beschrieben ist in Listing 4 zu erkennen, dass in Zeile 1 die Begrenzer auf “{” und “}” gesetzt wurden. Dies hat den Hintergrund, dass in diesem Beispiel ein Text in eine HTML-Datei generiert werden soll. Hierfür könnten auch die Standard-Begrenzer verwendet werden, allerdings müssten dann für Schlüsselwörter wie `<span>` die Zeichen “<” und “>” mit einem führenden Backslash definiert werden. Da dies für HTML-Dateien allerdings einen immensen Aufwand bedeutet, ist die Nutzung anderer Begrenzer sinnvoll. In Zeile 3 werden für ein ST sowohl der Name des Templates als auch Übergabeparameter definiert. Ein ST wird durch “> >” geschlossen. Die Begrenzer in Zeile 5 zeigen, dass alles, was sich zwischen Ihnen befindet, einen Übergabeparameter in sich trägt. Somit ist das Wiederverwenden des Templates und die variable Befüllung gewährleistet.

Um diese Templates nun in einem Java-Programm zu verwenden, benötigt es unter anderem die zuvor beschriebene ST-Klasse, sowie die Klasse *STGroupFile*, welche für die Verwaltung der stg-Datei als auch deren Templates, benötigt wird. In Zeile 6 von Listing 5 ist zu erkennen, dass einem *STGroupFile*-Objekt bei der Initialisierung eine URL

übergeben werden muss. Diese URL verweist auf die stg-Datei. Im Anschluss kann, wie in Zeile 8 ersichtlich, über die `getInstanceOf()`-Methode auf ein bestimmtes Template in der stg-Datei zugegriffen werden. Hierbei ist es wichtig, dass der übergebene Bezeichner der `getInstanceOf()`-Methode mit dem Bezeichner in der stg-Datei übereinstimmt. Schließlich ist die weiterführende Verwendung bereits zuvor mittels der ST-Klasse beschrieben worden.

---

```
1  import org.stringtemplate.v4.ST;
2  import org.stringtemplate.v4.STGroupFile;
3  ...
4
5  URL resourceUrl = Class.class.getResource("Example.stg");
6  STGroupFile exampleGroup = new STGroupFile(resourceUrl);
7
8  ST st = exampleGroup.getInstanceOf("example");
9  st.add("topic", "the university");
10 st.add("language", "english");
11
12 String output = st.render();
13 System.out.println(output);
```

---

Listing 5: Nutzung einer STG-Datei in Java

Bei der Ausführung dieses Beispiels wird auf der Konsole der Text aus Listing 6 angezeigt.

---

```
1  <span>
2      This test about the university is written in english.
3  </span>
```

---

Listing 6: STG-Ausgabe auf Konsole

### 2.2.1.3 JSON-Schema

JSON-Schemas geben die Struktur und den Inhalt einer JSON-/YAML-Datei vor. Hierdurch ist es möglich, den Nutzer in seinen Eingaben zu begrenzen und bereits während des Schreibens einer Datei dabei zu unterstützen sinnvolle Eingaben zu erstellen. In dieser Arbeit wird alleinig auf die Nutzung der Schema-Version 7, die neueste Version, eingegangen, da diese in der Anwendung verwendet wird. **TODO: Also sind diese JSON-Schemas eine Art Standard? Woher kommt das? Wer hats erfundn? Anscheinend gibt es ja schon 7 Versionen davon?**

JSON-Schemas können Objektstrukturen in beliebiger Tiefe schachteln. Im folgenden Abschnitt werden die grundlegenden Elemente eines JSON-Schemas erläutert. Weiterführende Funktionalitäten werden anhand der Implementierung in Kapitel 3.1.2 näher beleuchtet.

Ein einzelnes Objekt kann zur Verbesserung der späteren Nutzung mit einem Titel und einer kurzen Beschreibung versehen werden. Diese sind in Listing 7 in Zeile 2 und 3 dargestellt. Die Einträge *title* und *description* dienen lediglich der verbesserten Lesbarkeit für den Entwickler.

---

```
1  {
2    "title": "Product",
3    "description": "A product from Acme's catalog",
4    "type": "object",
5    "properties": {
6      "productId": {
7        "description": "The unique identifier for a product",
8        "type": "integer"
9      }
10   },
11   "required": [
12     "productId"
13   ]
14 }
```

---

Listing 7: Objekt-Beispiel eines JSON-Schemas

Einem Element muss stets ein *type*, also ein Typ, zugeordnet werden. Dies kann entweder ein Objekt, Zeile 4 in Listing 7, oder eine Liste sein. Einem Objekt können nun *properties* hinzugefügt werden. Diese besitzen neben einem eindeutigen Bezeichner ebenfalls eine Beschreibung und einen Typen. Auf dieser Ebene kann der Typ eine Nummer, *integer* in Zeile 8, oder auch ein Text, welcher den Typ *string* bekommen würde, sein. Ist eine der *Properties* ein notwendiges Feld, kann dies mittels des Schlüsselwortes *required* realisiert werden. Hierbei wird eine Liste an Bezeichnern hinterlegt, welche dem Objekt bereits zugeordnet wurden und somit stets vorhanden sein müssen. Das Beispiel stammt von der offiziellen JSON-Schema-Webseite[Tea21c]. Sollten einem Objekt keine weiteren *properties* hinzugefügt werden dürfen, ist dies mit dem Ausdruck aus Listing 8 möglich.

---

```
1  "additionalProperties": false
```

---

Listing 8: Begrenzung der Properties eines Schemas

Wie zuvor bereits beschrieben, kann ein Element auch als Liste deklariert werden. Dies ist an einem kleinen Beispiel in Listing 9 dargestellt. Hierbei ist es möglich die *items* einer Liste genauer zu definieren. In diesem Beispiel müssen die Elemente einer Liste dem Schema aus dem Beispiel aus Listing 7 entsprechen.

Eine JSON- oder YAML-Datei, welcher dieses Schema zugrunde liegt, besteht somit aus einer Liste an Produkten. Durch die Verwendung des *oneOf*-Operators in Zeile 6 werden nur Elemente mit dem darunterliegenden Schema akzeptiert. Bei mehreren Einträgen in

der *items* Aufzählung muss immer eines dieser Elemente auf das Objekt in der JSON-/YAML-Datei zutreffen.

---

```

1  {
2    "title": "Products",
3    "description": "A list of products from Acme's catalog",
4    "type": "array",
5    "items": {
6      "oneOf": [
7        {
8          "type": "object",
9          "properties": {
10             "productId": {
11               "description": "The unique identifier for a product",
12               "type": "integer"
13             }
14           },
15           "required": [
16             "productId"
17           ]
18         }
19       ]
20     }
21   }

```

---

Listing 9: Listen-Beispiel eines JSON-Schemas

Durch ein fest definiertes Schema ist es vielen IDEs, darunter auch IntelliJ und VSCode, möglich den Entwickler durch Fehlerhervorhebung und Autovervollständigung zu unterstützen. Hierfür ist es möglich bereits erstellte JSON-Schemas im *SchemaStore* bereitzustellen. Dies ist eine zentrale Stelle, um JSON-Schemas für IDEs zur Verfügung zu stellen. Bei dem *SchemaStore* handelt es sich um ein Open-Source-Projekt, bei welchem die Einbringung eines neuen Schemas simpel gestaltet ist. Es ist möglich ein fertiges Schema fest dort zu hinterlegen, hierdurch muss für jede neue Änderung allerdings ein neuer Pull Request erstellt werden. Dieser bedarf einer Zustimmung einem der Verwalter des *SchemaStores*. Da dies stets mit einer Verzögerung passiert, ist es möglich eine Verlinkung zu einem Schema zu erstellen. Somit können Änderungen an einem Schema durchgeführt werden, um diese Änderungen nach dem Hochladen direkt zur Verfügung stellen zu können. Zum aktuellen Zeitpunkt existieren 439 Schemas, welche durch *SchemaStore.org* für diverse IDEs<sup>3</sup> bereitgestellt werden[Kri21].

#### 2.2.1.4 fulibTools

TODO: Siehe Github Issue

---

<sup>3</sup><https://www.schemastore.org/json/#editors>

FulibTools ist Teil der Fujaba Tool Suite. Auch das in dieser Arbeit erstellte fulibWorkflows ist ein Teil der Fujaba Tool Suite. Fulib bildet die Grundlage für fulibTools, wobei fulibTools erweiterte Möglichkeiten für die Nutzung von fulib bereitstellt. Fulib ist ein Codegenerator, welcher mittels einer Domain Specific Language (DSL) Modelle als Diagramme darstellen kann[Kas21a]. Dies begrenzt sich nicht nur auf Klassenmodelle, sondern ist auch für Objektmodelle einsetzbar. FulibTools ist eine Erweiterung, da die Generierung der Diagramme auch abseits der eben erwähnten DSL funktioniert[Kas21b]. Hierdurch bietet sich die Möglichkeit Objektmodelle über ein spezielles YAML-Format oder ein Java-Objektmodell zur Laufzeit zu generieren. Gleiches gilt für Klassenmodelle. Die Verwendung von FulibTools ist somit für diese Arbeit eine bessere Wahl als *Graphviz*, eine Bibliothek zur Generierung von Diagrammen, direkt zu verwenden. Dies ist der Fall, da FulibTools bereits die Verarbeitung einer Eingabe übernimmt und hierdurch leichter für ein weiteres Tool der Fujaba Tool Suite zu verwenden ist.

### 2.2.2 Frontend des Web-Editors

Der Web-Editor für fulibWorkflows besteht aus einem Frontend und einem Backend. Dieses Kapitel beschäftigt sich mit den Technologien, welche für das Frontend verwendet werden. Hierbei wurde die Entscheidung über die verwendeten Technologien für die Integration des Editors in die Webseite `fulib.org` getroffen. Auf diese Integration wird in Kapitel 6 näher eingegangen.

#### 2.2.2.1 Angular

Die Grundlage für das Frontend ist Angular. Angular ist ein Framework für das Designen von Applikationen und gleichzeitig eine Entwicklungsplattform[Goo21]. Für Entwickler ist das Angular Command-Line-Interface ein wichtiger Bestandteil bei der Entwicklung mit Angular. Neben der Generierung einer neuen Anwendung können ebenfalls neue Komponenten, Services und Module generiert werden. Die *Komponenten* dienen der Strukturierung einer Anwendung und enthalten verschiedene Abschnitte ebendieser. Hierbei besteht ein großer Vorteil darin, Komponenten modular zu gestalten. Dies bedeutet, dass Komponenten so entwickelt werden, dass diese in der Anwendung wiederverwendet werden können, sollte dies möglich sein. Eine Komponente besteht aus drei einzelnen Bereichen:

1. Der Logik, welche in Typescript verfasst wird.
2. Einem Template, welches eine HTML-Datei ist.
3. Styles, welche im Template eingebunden werden können, um die Oberfläche grafisch zu verändern.

Im Template werden neben den Styles auch Bestandteile des Code-Segments verwendet, um Daten dynamisch anzeigen zu können. Ein neu generiertes Angular Projekt bietet allerdings nur die Grundlage einer Anwendung. Um weitere Funktionalitäten in der Anwendung verwenden zu können, können sogenannte Package Manager wie *npm* oder *yarn* verwendet werden, um neue Bibliotheken einzubinden.

Auf weitere Erklärungen wird an dieser Stelle verzichtet, da Angular in Gänze zu erklären den Rahmen dieser Arbeit überschreiten würde. In Kapitel 3.2 wird auf weitere Funktionen genauer eingegangen und diese anhand der Implementierung erklärt. Folgend werden Bibliotheken erläutert, welche die wichtigsten Bestandteile der Anwendung ermöglichen.

#### 2.2.2.2 Bootstrap

Um eine Oberfläche zu gestalten, welche einheitlich mit der von *fulib.org* sein soll, wurde Bootstrap zum Stylen der Oberflächenelemente verwendet. Bootstrap bietet diverse Komponenten wie Eingabefelder, Buttons, Menüs, Pop-Ups und viele weitere. Neben den Styles enthalten diese auch zusätzliche Funktionen, welche im Kontext der Komponente sinnvoll sind. Dabei bleibt es weiterhin abänderbar, um dem Entwickler mehr Freiheiten zur Gestaltung einer Oberfläche zu geben. Weiterhin ermöglicht es Bootstrap das Layout, also die Anordnung, von Komponenten auf einer Seite festzulegen. Dies beginnt bei der Größe einer Komponente bis hin zu der Anordnung in der Horizontalen und Vertikalen[Tea21b].

Zusätzlich zu Bootstrap wurden Bootstrap Icons verwendet, um die Oberfläche mit Symbolen zu versehen. Hierbei handelt es sich um eine Sammlung von rund über 1500 Icons, welche frei verwendbar sind[Tea21a].

#### 2.2.2.3 CodeMirror

CodeMirror ist ein Texteditor, welcher in JavaScript geschrieben wurde und somit in Web-Anwendungen verwendet werden kann. Es gibt zahlreiche Optionen, um CodeMirror an die Bedingungen der zu bauenden Anwendung anzupassen. Neben zahlreichen Programmiersprachen, welche durch das Hervorheben von Schlüsselwörtern und der Überprüfung der Syntax emuliert werden können, ist es möglich CodeMirror zu einer eigenen, individualisierten IDE zu konfigurieren. Erweiterungen für einen CodeMirror sind die sogenannten Add-Ons. Hierbei gibt es neben vielen bereits vorhandenen Erweiterungen auch die Möglichkeit eigene Add-Ons zu erstellen. Hierzu zählt unter anderem das zuvor erwähnte farbliche Hervorheben von Schlüsselwörtern, auch Highlighting genannt, wie auch die Überprüfung des Codes auf die Syntax der eingestellten Programmiersprache[Hav21].

Für eine einfache Einbindung in ein Angular-Projekt existieren bereits mehrere Bibliotheken, welche eine CodeMirror-Komponente bereitstellen. Bei dieser Komponente können nicht nur Optionen übergeben, sondern auch der Inhalt eines CodeMirrors, also den geschriebenen Code, aus der Komponente extrahiert werden. In dieser Arbeit wird hierzu ngx-codemirror von Scott Cooper verwendet[Coo21]. **TODO: ist die besser als die ganzen anderen mehreren existierenden von denen du oben sprachst?**

### 2.2.3 Backend des Web-Editors

Das folgende Kapitel beschäftigt sich mit dem zweiten Bestandteil des Web-Editors, das Backend. Vom Frontend wird eine YAML-Beschreibung an das Backend geschickt, in diesem wird dies als Eingabe für fulibWorkflows verwendet. Da fulibWorkflows eine Java-Bibliothek ist, wird ein Backend benötigt, welches auf Java basiert.

#### 2.2.3.1 Spring Boot

Mittels *Spring Boot* ist es möglich ohne zusätzliche Konfiguration eine auf *Spring* basierende Applikation zu erstellen[VMw22a]. Spring ist ein Framework, welches sich als Ziel gesetzt hat Java-Programmierung zu vereinfachen und zu verschnellern, allerdings keine Einbußen bei Geschwindigkeit, Komplexität und Produktivität zu machen[VMw22b]. **TODO: Klingt wie ein Zitat (falls es eins ist: umformulieren oder korrekt zitieren)** In diesem Kapitel wird sich mit der Erstellung eines REST-Services befasst. Ein REST-Service stellt Endpunkte bereit, welche über REST angesprochen werden können. Diese eignen sich zur Nutzung als simples Backend.

The screenshot shows the Spring Initializr web application. The interface is divided into several sections:

- Project:** Includes radio buttons for "Maven Project" and "Gradle Project" (selected). Below this are radio buttons for "Spring Boot" versions: "2.7.0 (SNAPSHOT)", "2.6.3 (SNAPSHOT)", "2.6.2" (selected), and "2.5.9 (SNAPSHOT)". There is also a "2.5.8" option.
- Language:** Includes radio buttons for "Java" (selected), "Kotlin", and "Groovy".
- Project Metadata:** A form with fields for "Group" (com.example), "Artifact" (demo), "Name" (demo), "Description" (Demo project for Spring Boot), and "Package name" (com.example.demo).
- Packaging:** Includes radio buttons for "Jar" (selected) and "War".
- Java:** Includes radio buttons for "17" (selected), "11", and "8".
- Dependencies:** A section with a button "ADD DEPENDENCIES... CTRL + B". Below it, there is a "Spring Web" dependency listed with a "WEB" tag and a description: "Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container."

Abbildung 2.3: Spring Initializr für eine Web-Anwendung

Das Anlegen eines Spring-Boot-Projektes kann durch den *Spring Initializr* erledigt werden, dabei handelt es sich um eine Webseite, welche in Abbildung 2.3 zu sehen ist[VMw21]. Hier-

bei können diverse Einstellungen getätigt werden, um das zu generierende Projekt an die Anforderungen des Benutzenden anzupassen. Die Abbildung zeigt die getätigten Einstellungen, um ein Gradle-Projekt mit Java-Version 17 als Programmiersprache zu generieren. Ebenfalls kann die Version von Spring Boot eingestellt werden. Zusätzliche Dependencies können im rechten Bereich des *Spring Initializrs* hinzugefügt werden. In diesem Fall wurde **Spring Web** ausgewählt, wodurch die wichtigsten Dependencies für eine REST-Applikation bereits enthalten sind.

Das mit diesen Einstellungen generierte Projekt ist bereits ein fertiges Backend, welches direkt ausgeführt werden kann. Durch sogenannte Annotations können weitere Endpunkte hinzugefügt werden. In Kapitel 3.3 werden Beispiele hierzu gezeigt und erläutert. Auf die genaueren Funktionen, welche mittels Annotations implementiert werden können, wird in Kapitel 3.3 eingegangen, anhand von Beispielen der Implementierung.

## 2.2.4 Deployment

Im folgenden Kapitel werden die Technologien, welche für die Bereitstellung einer Java-Bibliothek und einer Web-Anwendung verwendet werden können, behandelt. Hierbei wurde für *fulib Workflows* die Plattform Maven Central und Heroku für Web-Anwendungen gewählt.

### 2.2.4.1 Maven Central

*Maven Central* ist ein Archiv von Software-Artefakten für die Softwareentwicklung mit Java. Artefakte können Software Development Kits (SDK) oder Bibliotheken sein, welche von Entwicklern bereitgestellt werden, um diese zentralisiert bereitzustellen und den Aufwand der Konfiguration für andere Entwickler möglichst gering zu halten. Veröffentlichte Artefakte können in verschiedenen Build Management Tools verwaltet werden, zum Beispiel Gradle oder Maven. *Maven Central* wurde von der Apache Software Foundation im Jahr 2002 ins Leben gerufen[Clo22].

### 2.2.4.2 Heroku

Der in dieser Arbeit erstellte Web-Editor soll frei zugänglich sein. *Heroku* ist eine Möglichkeit kostenlos und schnell Web-Anwendungen bereitzustellen. Bei Heroku handelt es sich um eine sogenannte Cloud Platform as a Service (CPaaS), welche es ermöglicht in einer Cloud-Umgebung Services verschiedener Arten und Programmiersprachen bereitzustellen[Sal22c].



Um eine Anwendung über Heroku bereitzustellen bedarf es eines Accounts bei Heroku. Dies ist für kleine Anwendungen oder zum Ausprobieren bereits ausreichend. Entwickler können bis zu fünf Anwendungen über einen kostenlosen Account gleichzeitig bereitstellen. Zudem erhalten Entwickler über die Heroku-CLI ein weiteres Tool zum Hochladen des Codes und damit auch zum automatischen Bauen und Bereitstellen der Anwendung. Über Logs können Fehler in der Web-Ansicht einer Applikation gesichtet und analysiert werden.

Im *Heroku Dev Center* gibt es zahlreiche Tutorials, welche als Anleitung für die Bereitstellung verschiedenster Anwendungen dienen. Hierunter zählen unter anderem Node.js-Applikationen und Java- / Gradle-Anwendungen, welches sowohl das Frontend als auch das Backend des Web-Editors dieser Arbeit abdeckt[Sal22b]. Über diese Tutorials ist es möglich mit geringem Aufwand eine lokale Anwendung für das Deployment vorzubereiten.

## 3 Implementierung

Dieses Kapitel erläutert die Implementierung der in dieser Arbeit erstellten Anwendungen. Diese werden anhand der im vorherigen Kapitel erläuterten Technologien genauer erklärt. Hierbei beschäftigt sich Kapitel 3.1 mit `fulibWorkflows`, Kapitel 3.2 mit dem Frontend des `fulibWorkflows` Web-Editors und abschließend Kapitel 3.3 mit dem dazugehörigen Backend.

### 3.1 `fulibWorkflows`

Bei *fulib Workflows* handelt es sich um eine Java-Bibliothek, welche Workflow-Beschreibungen als Eingabe erhält, diese Eingabe parst und daraus HTML-/FXML-Mockups, Objektdiagramme und Klassendiagramme generiert. Das Parsen wird über einen von Antlr generierten Parser übernommen, hierzu wird in Kapitel 3.1.3 genauer auf die zugrundeliegende Grammatik eingegangen. Zudem wird auf die Limitationen des Parsers und der generierten Mockups eingegangen.

#### 3.1.1 Workflow-Format

Bevor in Kapitel 3.1.3 auf den Parser und die Grammatik einer Workflow-Beschreibung eingegangen wird, ist es nötig das Format in welcher ein Workflow beschrieben werden muss, näher zu beleuchten. Die Grundlage des Formats bildet YAML Ain't Markup Language (YAML), hierbei handelt es sich allerdings um eine stark begrenzte YAML-Syntax, welche verwendet werden kann. Näheres zu den Beschränkungen der Syntax wird in Kapitel 3.1.2 erklärt.

In Listing 10 sind alle möglichen Post-its aufgelistet. Im weiteren Verlauf wird für einen Post-it das Wort Zettel verwendet, da es sich bei jedem Element in der Beschreibung um einen einzelnen Zettel aus dem Event Storming handelt. Als Beispiel für einen Workflow ist hierbei die Registrierung eines neuen Benutzers auf einer Webseite in vereinfachter Form dargestellt. Folgend werden alle verwendbaren Zettel aus der aktuellen Version, v0.3.5, von `fulibWorkflows` aufgelistet und deren Funktion erläutert.

---

```
1 - workflow: User Registration
2
3 - user: Karli
4
5 - page:
6   - pageName: Registration
7   - text: Please create a new account
8   - input: Username
9     fill: Gonpachiro1912
10  - input: E-Mail
11  - password: Password
12  - password: Repeat Password
13  - button: Register
14    targetPage: Login
15
16 - command: Click Register
17
18 - policy: check e-mail input
19
20 - externalSystem: E-Mail Checker
21
22 - problem: This part is always taking a lot of time somehow
23
24 - event: e-mail is valid
25
26 - service: User management
27
28 - policy: create user
29
30 - event: user data received
31   username: Gonpachiro1912
32   eMail: ds@taicho.com
33   psw: nezuko!DS42
34
35 - data: User gonpachiro1912
36   username: Gonpachiro1912
37   eMail: ds@taicho.com
38   psw: nezuko!DS42
39
40 - event: User created
41
42 - page:
43   - pageName: Login
44   - text: Successfully logged in
```

---

Listing 10: Beispiel aller vorhandenen “Post-its”

## workflow

Zum Darstellen eines oder mehrerer Workflows benötigt es den *workflow*-Zettel. Diesem kann nur ein Name zugeordnet werden, allerdings werden alle weiteren Zettel unter einem workflow-Zettel ebendiesem Workflow zugewiesen. Pro Workflow-Beschreibung benötigt es somit mindestens einen workflow-Zettel. Im vorherigen Beispiel gibt es einen Workflow mit dem Namen *User Registration*.

### **service**

Durch den *service*-Zettel werden Services bereitgestellt. Diese sind einzig zur Strukturierung des Workflows und dem später daraus entstehenden ES-Board da. Hiermit kann erreicht werden, dass die nach einem service-Zettel folgenden Zettel von dem vorherigen Service ausgeführt werden. Im Beispiel aus Listing 10 gibt es nur einen Service, dieser kümmert sich um die Nutzerverwaltung, daher stammt auch der Name *User management*.

### **problem**

Falls während einem Event Storming an einem bestimmten Punkt innerhalb eines Workflows Fragen bei den Beteiligten aufkommen, können diese mittels *problem*-Zettel festgehalten werden. Gleiches gilt für Probleme, welche bisher auftraten, allerdings noch nicht festgehalten wurden. Somit ist es möglich zusätzliche Informationen, welche nicht zum eigentlichen Workflow gehören, darzustellen und in der späteren Entwicklung genauer zu betrachten. Im obigen Workflow Beispiel, gibt es das Problem, dass das Validieren einer E-Mail enorm lange dauert, siehe Zeile 20.

### **event**

Aus dem *Domain Event* aus dem Event Storming ist die verkürzte Version *event* als Indikator für einen weiteren Zettel geworden. Hierbei wird eine Beschreibung eines Events in der Vergangenheitsform als Bezeichnung verwendet. Dies bezieht sich auf eine Aktion, welche innerhalb eines Workflows aufgetreten ist. Wie in Listing 10 in Zeile 24 und ab Zeile 30 zu sehen, können event-Zettel allein stehen oder mit weiteren Informationen angereichert werden. Weitere Informationen, die einem Event zugeordnet sind, repräsentieren Daten, welche zu der ausgeführten Aktion gehören.

### **user**

Ein *user*-Zettel ist sehr ähnlich zu einem *service*-Zettel. Einem User wird ein Name zugeordnet. Dieser Zettel dient ebenfalls der Strukturierung eines Workflows und leitet einen Abschnitt ein, welcher aktiv von einem Nutzer durchgeführt werden muss. Im Beispiel-Workflow existiert ein User, welcher den Namen Karli trägt, welches in Zeile 3 zu sehen ist.

## **policy**

Eine *policy* ist wie auch beim *event* eine Abstrahierung eines Begriffes aus dem Event Storming. Ein *policy*-Zettel beschreibt somit einen Automatismus, welcher aufgrund eines vorherigen Events einen bestimmten Ablauf an Schritten ausführt. Im Beispiel aus Listing 10 reagiert die *policy* aus Zeile 18 auf den vorherigen Command, um automatisch zu prüfen, ob die eingegebene E-Mail valide ist.

## **command**

Ein *command*-Zettel stellt die Interaktion eines Nutzers mit einer Webseite oder Applikation dar. Im Falle des obigen Beispiels wird in Zeile 16 der Klick auf den Knopf *Register* aus der darüber aufgeführten Page simuliert. Dies ist aus der kurzen Beschreibung des Commands zu entnehmen.

## **externalSystem**

Wie auch bei dem *service* und *user* dient der *externalSystem*-Zettel dazu, einen Workflow zu strukturieren. In diesem Fall bedeutet dies, dass Aktionen oder Daten von einem System ausgeführt/gesendet werden, welche nicht Teil der zu entwickelnden Software sind, für welche das aktuelle Event Storming durchgeführt wird. In Zeile 20 des Beispiels existiert ein *externalSystem*, welches für die Validierung einer E-Mail-Adresse zuständig ist. Es kann sich dabei um ein System handeln, welches von einer anderen Firma oder einem separaten Team entwickelt wird.

## **data**

Wie in Kapitel 2.1 bereits erwähnt, ist *data* eine Erweiterung des Event Stormings. Ein *data*-Zettel benötigt als Bezeichner eine besondere Form, diese besteht zuerst aus einer Klassenbezeichnung und anschließend einem Namen für das Objekt. Dies ist nötig, um im späteren Verlauf das Aufbauen eines Datenmodells zu gewährleisten und ein Klassendiagramm und mehrere Objektdiagramme generieren zu können. Wie auch bei dem *event*-Zettel kann auch einem *data*-Zettel zusätzliche Informationen übergeben werden. Auf welche Art und Weise diese aufgebaut sein müssen und welche Funktionen diese innehaben, darauf wird in Kapitel 3.1.4.3 genauer eingegangen. In Zeile 35 unseres Beispiels wird ein *User* mit den aus dem Event stammenden Informationen zu Username, E-Mail und Passwort angelegt.

## **page**

Der *page*-Zettel ist ebenfalls eine Erweiterung zum klassischen Event Storming. Aus den *page*-Zetteln eines Workflows werden später Mockups generiert, wie dies genau funktioniert

wird in Kapitel 3.1.4.2 erläutert. An diesem Punkt ist es wichtig, die Restriktionen einer *page* zu erläutern. Eine Page besteht aus einer Liste an Elementen, hierbei steht *pageName* für den Bezeichner der Page und wird im Mockup nicht dargestellt. Um die Oberfläche einer Applikation zu beschreiben, existieren vier Elemente, welche beliebig oft verwendet werden können.

Ein Text-Element enthält einen Text, welcher entweder zur Verwendung als Überschrift, Bezeichner oder Trenner verwendet werden kann. Um Daten in einer Applikation eingeben zu können, gibt es das *input*- und das *password*-Element. Bei beiden handelt es sich um Eingabefelder, mit dem Unterschied, dass bei dem *password*-Element der eingetippte Inhalt mit Punkten ersetzt wird. Der Bezeichner, welcher einem Input oder Password hinterlegt wird, wird als Platzhalter im Eingabefeld und als Label über ebendiesem angezeigt. Um die Variationen der Mockups zu erweitern und diese mit beispielhaften Eingaben zu füllen, können sowohl dem *input*- als auch *password*-Element ein Attribut namens *fill* zugeordnet werden. Dabei handelt es sich um den Inhalt des Eingabefeldes, welches im Mockup angezeigt wird. Zuletzt kann ein Knopf mithilfe eines *button*-Elements erstellt werden, dabei wird der Bezeichner als Inhalt des Knopfes dargestellt. Das *button*-Element kann ebenfalls mit einem weiteren Attribut versehen werden. Mit dem Attribut *targetPage* kann dem Knopf der Name einer Seite übergeben werden, welche innerhalb der workflow-Datei definiert wurde. Dafür ist es notwendig, dass die erstellten Pages unterschiedliche Namen besitzen. Dieses Attribut entfaltet seine Funktionalität im Web-Editor zu fulibWorkflows, da es somit möglich ist eine lose Verlinkung zwischen einem Button und einer Page zu erzeugen. Näheres zu der Verwendung und der Umsetzung folgt in Kapitel 3.2.2.

Eine Oberfläche kann nur von oben nach unten beschrieben werden, eine weitere Limitierung ist, dass es nicht möglich ist mehrere Elemente in eine Reihe zu ordnen. Hierdurch ist das Designen eines Mockups durch die geringe Anzahl an Elementen stark begrenzt und kann lediglich für simple Benutzeroberflächen verwendet werden.

### 3.1.2 JSON-Schema

Wie im vorherigen Kapitel bereits erwähnt bedient sich die Beschreibung eines Workflows grundlegend der Syntax von YAML. In Kapitel 2.2.1.3 wurden bereits die Grundlagen für JSON-Schemas geschaffen, in diesem Kapitel wird das erstellte fulibWorkflows-Schema genauer betrachtet. Das komplette Schema ist in dem referenzierten fulibWorkflows-Repository im Anhang hinterlegt, da dieses zu lang ist, um es übersichtlich in diesem Kapitel zu erläutern. Dadurch wird nur auf die wichtigsten Punkte der Implementierung eingegangen. Um die Lesbarkeit für Entwickler zu verbessern, ist das Schema in zwei Dateien aufgeteilt.

Listing 11 ist eine minimale Version des eigentlichen Schemas, in welchem dennoch die wichtigsten Funktionen dargestellt sind. Das fulibWorkflows-Schema enthält die Definitio-

---

```
1  {
2    "type": "array",
3    "additionalItems": false,
4    "$defs": {
5      "pageItem": {
6        "description": "Defines a ui page",
7        "type": "object",
8        "properties": {
9          "page": {
10             "$ref": "page.schema.json"
11          }
12        },
13        "required": [
14          "page"
15        ],
16        "additionalProperties": false
17      }
18    },
19    "items": {
20      "oneOf": [
21        {
22          "$ref": "#/$defs/pageItem"
23        },
24        {
25          "$ref": "#/$defs/problemItem"
26        }
27      ]
28    }
29  }
```

---

Listing 11: Referenzieren eines anderen Schemas

nen und die Festlegung der erlaubten Elemente in der obersten Liste. Durch das Schema werden nur JSON-/YAML-Dateien akzeptiert, welche aus einer Liste an Elementen bestehen. Hierbei sind die Elemente jedoch festgelegt durch die in Zeile 20 und 21 dargestellten Zeilen. Ein *item* darf nur aus einem der Elemente bestehen, welche in der Auflistung ab Zeile 22 festgelegt sind. Um die Lesbarkeit zu vereinfachen und die Schachtelungstiefe möglichst gering zu halten, werden nur die erlaubten Elemente referenziert. Die referenzierten Elemente wurden im ersten Teil des Schemas, ab Zeile 4, definiert. Für jedes der im vorherigen Kapitel erwähnten Zettel gibt es ein Element im Schema. Die Definitionen der Elemente, welche nicht in Listing 11 dargestellt sind, enthalten Standardwerte, welche bereits in Kapitel 2.2.1.3 erläutert wurden. Die Definition der Page ist jedoch ein Sonderfall, welche durch das Aufteilen in mehrere Dateien entstanden ist. Es ist möglich weitere Schemas aus einer anderen Datei zu importieren, dies ist in Zeile 11 ersichtlich. Auch dort es Referenzen, allerdings referenzieren diese nicht auf eine im Schema befindliche Definition, sondern auf die Page-Definition aus der *page.schema.json*-Datei. Die Aufteilung wurde durchgeführt, da die Page eine Liste ist und ebenfalls fünf eigene Elemente definiert. Al-



((a)) Leere Datei

((b)) Fehlerhafte Eingabe

Abbildung 3.1: Fehleranzeige in IntelliJ



((a)) Alle Elemente

((b)) Page Elemente

Abbildung 3.2: Autovervollständigung in IntelliJ

lein das Page-Schema beläuft sich auf 93 Zeilen und umfasst somit allein die Hälfte des Parent-Schemas.

Wie in Kapitel 2.2.1.3 bereits erwähnt, ist das fulibWorkflows-Schema ebenfalls bei Schemastore.org hinterlegt. Das Schema wird automatisch auf Dateien mit der Dateierdung *.es.yaml* vom Editor angewendet. Dadurch ist es zum Beispiel in IntelliJ möglich Autovervollständigung für fulibWorkflows zu bekommen und auf Fehler hingewiesen zu werden.

In Abbildung 3.1 ist das Hervorheben von Fehlern in IntelliJ dargestellt, welches durch das JSON-Schema generiert wird. In Abbildung 3.1((a)) ist die Datei leer, wodurch die Schema Validierung anschlügt und den Entwickelnden darauf hinweist, dass ein Array, also eine Liste an Elementen, benötigt wird. Sobald ein Element begonnen wird, wird diese Warnung nicht mehr angezeigt. Sollte der Entwickelnde wiederum ein Element hinzufügen, welches keinem der definierten Elemente des Schemas entspricht, wird die Meldung aus Abbildung 3.1((b)) angezeigt. Da ein Command keine zusätzlichen Attribute/Properties akzeptiert, dennoch eines hinzugefügt wurde, besagt der Fehler, dass es nicht erlaubt ist weitere Attribute hinzuzufügen.

Wie zuvor erwähnt ermöglicht ein Schema jedoch nicht nur das Hervorheben von Fehlern, sondern unterstützt den Entwickelnden zusätzlich durch Autovervollständigung. Dies ist in Abbildung 3.2 dargestellt. Hierbei werden aufgrund des Kontextes verschiedene Möglichkeiten von zu erstellenden Elementen vorgeschlagen. Auf oberster Ebene werden alle erlaubten Schlüsselwörter für Elemente angezeigt, dies ist in Abbildung 3.2((a)) dargestellt. Hierbei fällt auf, dass die Elemente für eine Page nicht angezeigt werden, dies ist jedoch der Fall, sobald der Kontext dies zulässt. Während der Entwickelnde ein page-Element de-



finiert, werden wie in Abbildung 3.2((b)) dargestellt die Schlüsselwörter für Page-Elemente vorgeschlagen.

### 3.1.3 Antlr-Grammatik

Da die möglichen Eingaben durch das zuvor beschriebene JSON-Schema bereits verringert wurden, fiel die Wahl der Verarbeitung der YAML-Eingabe auf einen durch Antlr generierten Parser. Dieser bietet die Möglichkeit, während dem Parsen weitere Aktionen durchzuführen und in dem Fall dieser Anwendung ein Datenmodell aus der Eingabe zu erstellen. Das Datenmodell wird im folgenden weiterverarbeitet und bietet somit eine Grundlage für die Generierung, auf welche im folgenden Kapitel eingegangen wird.

Wie in Kapitel 2.2.1.1 bereits beschrieben, ist die Grundlage eines Antlr-Parsers die dazugehörige Grammatik, welche nun beleuchtet wird. Die komplette Grammatik ist im referenzierten fulibWorkflows-Repository im Anhang hinterlegt, in diesem Kapitel werden Ausschnitte daraus verwendet. Im Folgenden wird anstatt Zettel zur Beschreibung eines Post-its die englische Übersetzung “Note” verwendet, um einen direkten Bezug zu den nachfolgenden Listings herzustellen.

---

```
5  file: workflows+ ;
6
7  workflows: workflow NEWLINE eventNote* ;
8
9  eventNote: ( normalNote | extendedNote | page) NEWLINE? ;
```

---

Listing 12: Grammatik für Workflows

Zuerst wird die grundlegende Struktur einer Datei festgelegt, dies ist durch die drei Regeln in Listing 12 dargestellt. Da eine Datei mehrere Workflows beinhalten kann, ist die oberste Regel in Zeile 5 der Startpunkt des Parsers. Da als Eingabe eine YAML-Datei ist, heißt die oberste Regel *file* und erfordert mindestens einen *workflow*. Ein *workflow* besteht immer aus einem workflow-Note und beliebig vielen event-Notes, wobei diese immer mit einer Leerzeile von einander getrennt sind. Die Spezifikation ist aufgrund der YAML-Syntax notwendig. Ein event-Note ist immer einem von drei Typen zuzuordnen, wobei nach einem Note beliebig viele Leerzeilen folgen können.

Die Unterscheidung zwischen normal-/extended-Note, workflow und page erfolgt durch das Schlüsselwort, welches zwischen Bindestrich (**MINUS**) und Doppelpunkt (**NAME**) befindet. Sowohl ein workflow-Note als auch die normal-Notes besitzen nach dem Doppelpunkt einen Wert, welcher durch **NAME** gekennzeichnet ist. Dies ist in Listing 13 in Zeile 11 und 13 dargestellt. Ein extended-Note besitzt neben dem Wert zusätzliche Attribute, welche in einer neuen Zeile beschrieben werden. Die Anzahl der Attribute ist beliebig, es ist

---

```

11 workflow: MINUS 'workflow' COLON NAME ;
12
13 normalNote: MINUS NORMALNOTEKEY COLON NAME ;
14
15 extendedNote: MINUS EXTENDEDNOTEKEY COLON NAME NEWLINE attribute* ;
16
17 page: MINUS 'page' LISTCOLON NEWLINE pageList ;

```

---

Listing 13: Grammatik für Notes

somit erlaubt einen extended-Note ohne weitere Attribute anzugeben. Die Page ist wie zuvor bereits erwähnt ein Sonderfall, welches sich auch in der Grammatik widerspiegelt. Dem Schlüsselwort *page* folgt ein gesonderter Doppelpunkt und anschließend eine Liste von neuen Elemente.

---

```

30 NORMALNOTEKEY: 'externalSystem' | 'service' | 'command'
31                | 'policy' | 'user' | 'problem' ;
32
33 EXTENDEDNOTEKEY: 'event' | 'data' ;

```

---

Listing 14: Schlüsselwörter zum Identifizieren von Notes

Die zuvor erwähnten normal-Notes bestehen wie in Listing 14 Zeile 30 und 31 dargestellt aus *externalSystem*, *service*, *command*, *policy*, *user* und *problem*. Diese erhalten nur einen Bezeichner und erlauben keine weiteren Attribute. Zu den extended-Notes zählen *event* und *data*. Diese erhalten weitere Attribute, um Daten, welche zwischen Services verschickt werden, darstellen zu können.

---

```

19 attribute: INDENTATION NAME COLON value NEWLINE? ;
20
21 value: NAME | NUMBER | LIST;

```

---

Listing 15: Grammatik von Attributen

Attribute werden eingerückt und enthalten neben einem Bezeichner (**NAME**) einen dazugehörigen Wert (*value*). Ein Wert kann entweder ein Text, eine Nummer oder eine Liste sein, wobei eine neue Zeile optional ist. Die dazugehörigen Regeln sind in Zeile 19 und 21 aus Listing 15 vermerkt.

Wie in Listing 16 zu sehen ist, ist der akzeptierte Text auf eine feste Menge an verschiedenen Zeichen begrenzt. Ein Text muss stets mit einem Buchstaben beginnen, ungeachtet ob groß- oder kleingeschrieben. Darauf können Zahlen, Sonderzeichen und weitere Wörter folgen. Die Sonderzeichen sind in Zeile 36 genauer beschrieben.

---

```

36  NAME: ([A-Za-zäÄöÜß] [0-9]* [-/_.,'@!?!]* [ ]* [0-9]* [-/_.,'@!?!]* [ ]*)+ ;
37
38  LIST: '[' (.) * '?' ']' ;
39
40  NUMBER: [0-9]+ ;

```

---

Listing 16: Grammatik von Werten

Eine Nummer kann lediglich eine ganze Zahl sein, führende Nullen sind erlaubt. Die Möglichkeit als Wert eine Liste angeben zu können, basiert auf der Möglichkeit Objekt- und Klassendiagramme mit *fulibWorkflows* zu generieren. Hierzu wurde die Syntax von Java als Grundlage genommen. Zwischen den Klammern in Zeile 38 befindet sich eine sogenannte Wildcard, welche es erlaubt alle Symbole als Eingabe zu akzeptieren. Die Klammern erfüllen somit nicht nur den Zweck als Listendarstellung, sondern auch die Begrenzung der Wildcard. Eine Wildcard für die Eingabe eines Textes zu verwenden war für diese Grammatik aufgrund der Struktur vorerst nicht möglich, da es keine passenden Begrenzungen gab, welche keine anderen Regeln überschrieben hätte.

---

```

23  pageList: pageName NEWLINE element* ;
24
25  pageName: INDENTATION MINUS 'pageName' COLON NAME ;
26
27  element: text | inputField | button ;
28
29  text: INDENTATION MINUS 'text' COLON NAME NEWLINE;
30
31  inputField: INDENTATION MINUS ELEMENTKEY COLON NAME NEWLINE fill? ;
32
33  button: INDENTATION MINUS 'button' COLON NAME NEWLINE targetPage?;
34
35  fill: INDENTATION 'fill' COLON NAME NEWLINE;
36
37  targetPage: INDENTATION 'targetPage' COLON NAME NEWLINE;

```

---

Listing 17: Grammatik einer Page

Jeder Page muss ein *pageName*-Element zugeordnet werden, um sie später referenzieren zu können. Weiterhin können Pages beliebig viele Elemente beherbergen. Ein Element kann entweder ein Text, Eingabefeld oder Knopf sein. Ein Text-Element wird ein Text (**NAME**) zugeordnet. Dies ist bei den Eingabefeldern und dem Knopf anders, da diese wie in Kapitel 3.1.1 beschrieben zusätzliche Attribute besitzen können.

Zu dem aus der Grammatik generierten Parser gehört unter anderem ein Interface, welches für diese Grammatik den Namen *FulibWorkflowsListener* trägt. Um während des Parsens ein Datenmodell aufzubauen, wurde ein eigener Listener implementiert, welcher die Methoden des Interfaces überschreibt. Für jede Regel aus der Grammatik existiert eine

enter- und eine exit-Methode, welche nach enter/exit mit dem Namen der jeweiligen Regel verknüpft ist. Daraus entstehen Methoden wie zum Beispiel enterPage und exitPage. In den enter-Methoden werden ausschließlich neue Objekte angelegt und globale Variablen zurückgesetzt.

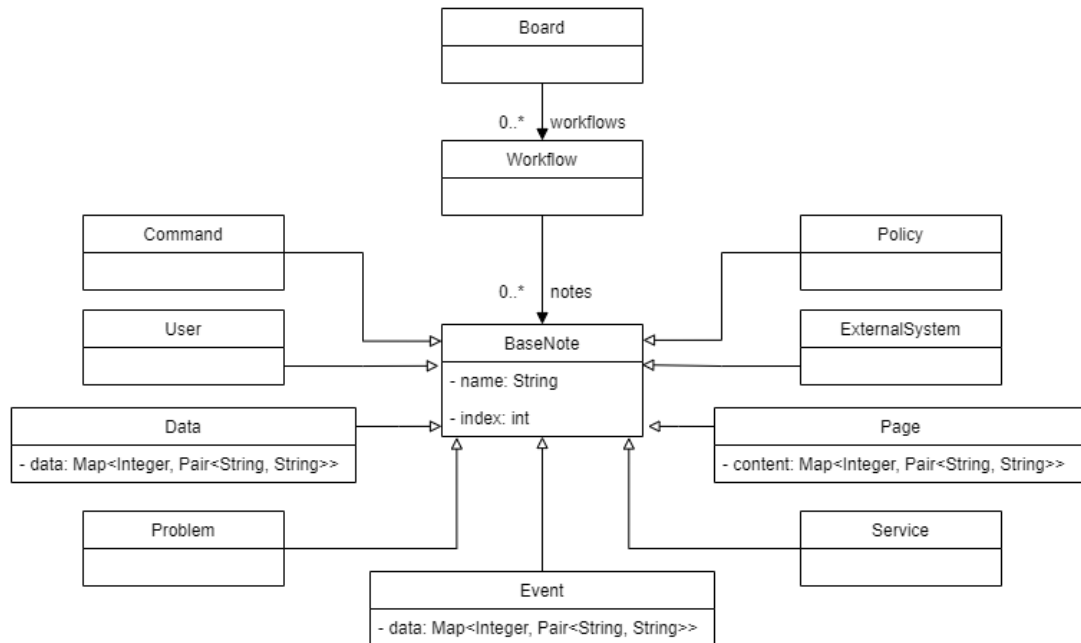


Abbildung 3.3: Klassendiagramm fulibWorkflows

Bevor anhand eines Beispiels die Verwendung einer exit-Methode erläutert wird, ist es notwendig das Datenmodell genauer zu betrachten. In Abbildung 3.3 ist das Klassendiagramm abgebildet, welches die Struktur eines ES-Boards nach dem Parsen der YAML-Eingabe widerspiegelt. Jeder Note besitzt eine dazugehörige Klasse, welche von **BaseNote** erbt. Für jeden Note existiert somit ein Name und ein Index, auf welchen im folgenden Abschnitt eingegangen wird. Da es im Event Storming ein dazugehöriges *Board* gibt, ist dies ebenfalls eine Klasse, welche alle *workflows* einer Eingabe hält. Ein Workflow besteht weiterhin aus vielen Notes. Wie zuvor bereits erläutert, sind **Data**, **Event** und **Page** Sonderfälle unter den Notes, da diese weitere Daten beherbergen. Daher haben diese Klassen ein Attribut, welches diese Daten organisiert in einer Map hält. Hierbei wird als key der Index eines Notes verwendet und das value ist ein Pair. Das Pair beinhaltet den Bezeichner und den dazugehörigen Wert einer zusätzlichen Property eines Notes.

In Listing 18 wird ein neues **Page**-Objekt erstellt. Bevor jedoch die `exitPage`-Methode aufgerufen wird, werden alle Elemente der **Page** in der Variable `noteData` gespeichert. Diese Elemente werden in der `exitElement`-Methode hinzugefügt, der Name einer **Page** wird hingegen in der gesonderten `exitPageName`-Methode zu `noteData` hinzugefügt. Zusätzlich zu den Daten eines Notes, wird diesem ein Index gegeben und anschließend zur Liste aller Notes hinzugefügt. Der Index ist notwendig, um die Reihenfolge von Notes und deren Attributen beizubehalten.

---

```
106  @Override
107  public void exitPage(FulibWorkflowsParser.PageContext ctx){
108      Page newPage = new Page();
109
110      newPage.setContent(noteData);
111      newPage.setIndex(noteIndex);
112      noteIndex++;
113
114      notes.add(newPage);
115  }
```

---

Listing 18: exitPage-Methode

### 3.1.4 Generierung von Dateien

Aus einer Workflow-Beschreibung können bis zu fünf verschiedene Typen von Dateien generiert werden. Der Einstiegspunkt und somit der Start des Parsens und der Generierung ist die *BoardGenerator*-Klasse. Diese hat Methoden, um Datei- oder String-Eingaben zu verarbeiten. Nachdem mittels des Parsers aus der Eingabe ein fertiges Board-Objekt erstellt wurde, werden weitere Klassen zur Generierung verwendet. HTML-Dateien werden vom *HtmlGenerator* generiert, hierbei handelt es sich um Mockups und das ES-Board. Mockups können allerdings auch als FXML-Datei generiert werden, um eine Grundlage für eine JavaFx-Anwendung zu bilden, diese Generierung übernimmt der *FxmlGenerator*. Zuletzt vereint der *DiagramGenerator* die Generierung von Objekt- und Klassendiagrammen. Außer der *BoardGenerator*-Klasse sind die restlichen Generator-Klassen für die Vorbereitung der Daten zuständig. Diese erhalten Eingaben von dem *BoardGenerator*, bereiten diese Eingabe auf, je nachdem welche Daten benötigt werden und enthalten eine separate Methode zum Erstellen von Dateien im Dateisystem. Das Bauen einer Datei in Form eines Strings wird in einer gesonderten *Constructor*-Klasse erledigt. Da es fünf verschiedene Typen von Dateien gibt und das Bauen für jede Datei unterschiedlich ist, existiert für jeden Typ eine eigene *Constructor*-Klasse. Im Folgenden werden die Aufbereitungsschritte genauer beleuchtet.

#### 3.1.4.1 Event-Storming-Board

Für die Generierung des Event-Storming-Boards bedarf es keiner Bearbeitung des *HtmlGenerator*s, da das gesamte Board generiert werden soll und die Daten, welche vom Parser erstellt wurden, bereits optimiert sind. Die HTML-Datei wird mittels STs, welche in einer STG organisiert sind, zusammengebaut. Für jeden Workflow im Board-Objekt wird eine neue Reihe in der HTML-Datei angelegt. Innerhalb einer Workflow-Reihe werden alle dazugehörigen Notes gebaut. Hierbei wird zwischen den verschiedenen Notes unterschieden, um verschiedene Darstellungen zu ermöglichen. Je nach Note wird eines von drei STs

verwendet. Für die Standard-Notes wird eine neue *Card*, eine Bootstrap-CSS-Klasse, erstellt, welche den Typen des Notes, dessen Content und eine bestimmte Farbe übergeben bekommt. Für die organisatorischen Notes, User, Service und ExternalSystem, wird eine Card erstellt, welche kleiner als die eines normalen Notes ist. Zudem wird in organisatorischen Notes nur ein Icon und der Bezeichner angezeigt, wobei das Icon ein Bootstrap-Icon ist. Data- und Page-Notes werden gesondert mit einem dritten ST behandelt, da es neben Typ, Content und Farbe noch einen Link gibt, welcher als Button definiert und für die Verwendung im Web-Editor genutzt wird.

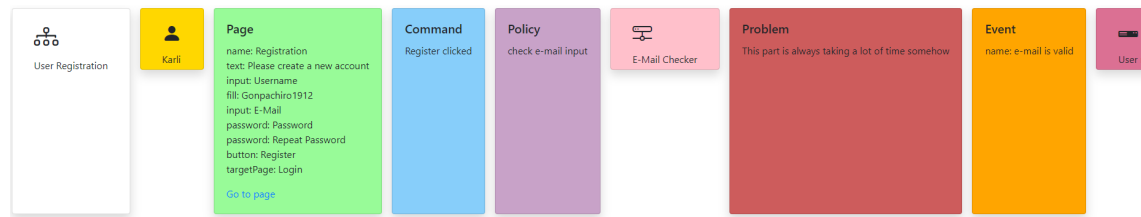


Abbildung 3.4: Mittels fulibWorkflows generiertes Event-Storming-Board

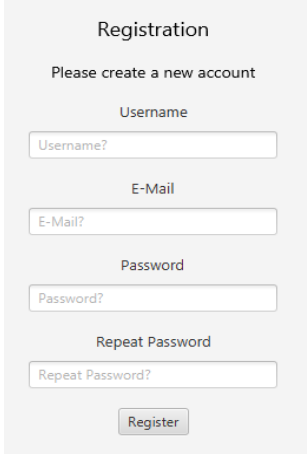
**TODO:** der kram auf den notes könnte auch schwer zu lesen sein

In Abbildung 3.4 ist ein Ausschnitt des in Listing 10 beschriebenen Workflows in Form eines generierten ES-Boards dargestellt. Hierbei sind die zuvor beschriebenen Unterschiede zwischen den verschiedenen Notes erkennbar. Jeder Note-Typ besitzt eine eigene Farbe, wobei für User, Service und ExternalSystem eine kleinere Card und jeweils ein eigenes Icon erkennbar sind. Der Page-Note enthält die meisten Informationen, da nur dort zusätzliche Informationen in der Workflow-Beschreibung existierten.


#### 3.1.4.2 Mockups (HTML und FXML)

Die Generierung von HTML-Mockups wird durch die *HtmlGenerator*-Klasse übernommen. Diese filtert aus allen Workflows und den dazugehörigen Notes die Pages heraus und übergibt diese an die *PageConstructor*-Klasse. FXML-Mockups erhalten eine gesonderte Generator- und Constructor-Klasse. Die in diesen Klassen befindlichen Funktionen ähneln der Funktionsweise des HtmlGenerators und PageConstructors stark. Eine Unterscheidung in HTML und FXML wurde vorgenommen, um bei der Generierung die Möglichkeit von verschiedenen Optionen offenzulassen. Somit könnten entweder nur HTML- oder FXML-Mockups erstellt werden. Der FxmlConstructor und PageConstructor haben somit sowohl eine ähnliche Funktionsweise als auch einen ähnlichen Aufbau, da die zugrundeliegenden Daten gleich sind. Lediglich die zugrunde liegende STG unterscheidet die beiden Klassen. In beiden STGs gibt es ein ST zum Aufbau der generellen Struktur der jeweiligen Datei und für jedes verfügbare Oberflächenelement ein weiteres ST. Es existieren somit für die Oberflächenelemente STs für Text, Eingabefeld, Passwortfeld und Knopf.

**TODO:** bilder etwas größer machen?



((a)) FXML-Mockup



((b)) HTML-Mockup

Abbildung 3.5: Mittels fulibWorkflows generierte Mockups

In Abbildung 3.5 sind die generierten Mockups aus dem vorherigen Beispiel dargestellt. Diese bestehen, entsprechend der Beschreibung, aus einem Text, drei Eingabefeldern, einem Passwortfeld und einem Knopf. Bei der Generierung wird vor allem darauf geachtet, dass sich die Oberflächen möglichst stark ähneln. Der Aufbau der in Abbildung 3.5((a)) und Abbildung 3.5((b)) dargestellten Oberflächen ist somit grundlegend gleich. Unterschiede entstehen einzig durch die verschiedenen Styles, da in dem HTML-Mockup Bootstrap zum Stylen der Oberfläche verwendet wurde. Zudem ist erkennbar, dass das Username-Eingabefeld bei dem FXML-Mockup (links) leer ist, allerdings ist dieses mit den im Beispiel definierten Daten im HTML-Mockup (rechts) bereits befüllt. Diese Wahl wurde getroffen, da die beiden Mockup-Typen verschiedene Anwendungszwecke haben. Die HTML-Mockups können im Web-Editor angezeigt werden und somit direkt mit den am Event Storming teilnehmenden Personen angeschaut werden. Dies ist mit den FXML-Mockups nicht möglich, da diese ausschließlich zur Grundlage einer Benutzeroberfläche in einer Java-Anwendung werden.

### 3.1.4.3 Objektdiagramme

Zuletzt gibt es die *DiagramGenerator*-Klasse, welche die Eingabe für Objekt- und Klassendiagramme aufarbeitet. Jeder Data-Note erhält sein eigenes Objektdiagramm. Damit die zeitliche Abfolge und somit ein korrektes Objektdiagramm entsteht, wird jeder Data-Note zu einer Liste hinzugefügt und diese anschließend an die *ObjectDiagramConstructor*-Klasse übergeben. Hierdurch werden pro Data-Note mehr Objekte zu dem dazugehörigen Objektdiagramm hinzugefügt, sollte es eine Verbindung zu einem bestehenden Objekt geben. FulibTools verwendet Graphviz zur Erstellung von Diagrammen, zusätzlich gibt es die Möglichkeit Objekt- und Klassendiagramme anhand einer bestimmten Eingabe zu generieren. Diese Funktionalität macht sich fulibWorkflows im *ObjectDiagramConstructor* zunutze. Aus der übergebenen Liste an Data-Notes wird eine YAML-Datei erstellt, welche

den Spezifikationen von `fulibYaml` entspricht. In Listing 19 ist die zum `ObjectDiagram`-Constructor gehörende STG-Datei abgebildet. Ein Objekt benötigt immer einen Namen, eine Klasse, in Zeile 5 als *type* gekennzeichnet, und optionale Attribute. Die Form eines Attributes ist in dem ST ab Zeile 10 abgebildet, ein Attribute besteht lediglich aus einer Klasse, erneut *type* genannt, und dem dazugehörigen Wert.

---

```
1  delimiters "{", "}"
2
3  object(name, type, attributes) ::= <<
4  - {name}: .Map
5    type: {type}
6    {attributes}
7
8  >>
9
10 attribute(type, value) ::= <<
11 {type}: {value}
12
13 >>
```

---

Listing 19: `FulibYaml.stg`

TODO: vlt kann man das hier auf die Seite davor schieben und Listing 20 auf diese Seite? Damit dann die nächste Abbildung nicht >1 Seite vom Text entfernt ist?

Nachdem ein Objektdiagramm in `fulibYaml`-Notation vorhanden ist, wird `FulibTools` verwendet, um daraus ein Diagramm zu generieren. In Listing 20 ist diese Generierungsmethode abgebildet. Der erste Parameter der Methode enthält die Objektstruktur in Form von `fulibYaml` als String. Daraus wird in Zeile 99 und 100 ein `root`-Objekt erstellt, welches die Klasse `YamlIdMap` aus der Bibliothek `fulibYaml` nutzt. Dieses `root`-Objekt wird gemeinsam mit dem in Zeile 96 festgelegten Dateinamen durch `FulibTools` in Zeile 102 generiert. `FulibTools` erlaubt bei der Generierung nicht, dass der Inhalt der zu generierenden Datei zurückgegeben wird, die Datei wird sofort im Dateisystem erstellt.

Im Anschluss an die Generierung durch `FulibTools` wird der Inhalt der generierten Datei als String ausgelesen und von der Methode zurückgegeben. Danach wird die generierte Datei, sowie ein für die Generierung temporär angelegter Ordner wieder gelöscht. Dies hat den Hintergrund, dass die Generierung von Dateien, in Form vom Speichern im lokalen Dateisystem, gesammelt in der `BoardGenerator`-Klasse vollzogen werden soll. Zudem wird im `BoardGenerator` die Möglichkeit geboten, Dateien aus einer Workflow-Beschreibung zu generieren und diese als String zurückzugeben. Diese Methoden existieren für die Nutzung im `fulibWorkflows`-Web-Editor, welche in einem nachfolgenden Kapitel näher betrachtet werden.



---

```

95  private String generateObjectDiagram(String objectYaml, int index) {
96      String fileName = "tmp/test/diagram_" + index;
97      String result = "";
98
99      YamlIdMap idMap = new YamlIdMap();
100     Object root = idMap.decode(objectYaml);
101
102     fileName = FulibTools.objectDiagrams().dumpSVG(fileName, root);
103
104     try {
105         result = Files.readString(Path.of(fileName + ".svg"));
106
107         Files.deleteIfExists(Path.of(fileName + ".svg"));
108
109         Files.deleteIfExists(Path.of("tmp/test/"));
110     } catch (IOException e) {
111         e.printStackTrace();
112     }
113
114     return result;
115 }

```

---

Listing 20: Generierungsmethode eines Objektdiagramms

TODO: puh da muss man aber ganz schön blättern jetzt, abbildung ist ganz weit woanders

In Abbildung 3.6 sind zwei Objektdiagramme dargestellt, welche mit `fulibWorkflows` generiert wurden. Die Beschreibung des Workflows stammt von einem Beispiel, welches zum Testen dieser Funktionalität in `fulibWorkflows` verwendet wurde. Das Beispiel ist im Anhang in Listing 27 hinterlegt. Zu dem Objektdiagramm aus Abbildung 3.6((a)) ist ein weiteres Room-Objekt in Abbildung 3.6((b)) hinzugefügt worden. Dies basiert auf der vorher erwähnten Funktion, dass Objekte in einer Liste gespeichert werden und jedes neue Objekt dort hinzugefügt wird, bevor ein Objektdiagramm generiert wird.

### 3.1.4.4 Klassendiagramme

Der im vorherigen Kapitel eingeführte DiagramGenerator verwaltet nicht nur die Objektdiagramme. Nachdem alle Data-Notes aus allen Workflows durchlaufen wurden, enthält der Generator eine Liste aller Data-Notes eines ES-Boards. Diese Liste wird an die *ClassDiagramConstructor*-Klasse weitergegeben.

Für das Erstellen eines Klassenmodells müssen die Data-Notes bestimmte Spezifikationen erfüllen, um das gewünschte Ergebnis zu erzielen.

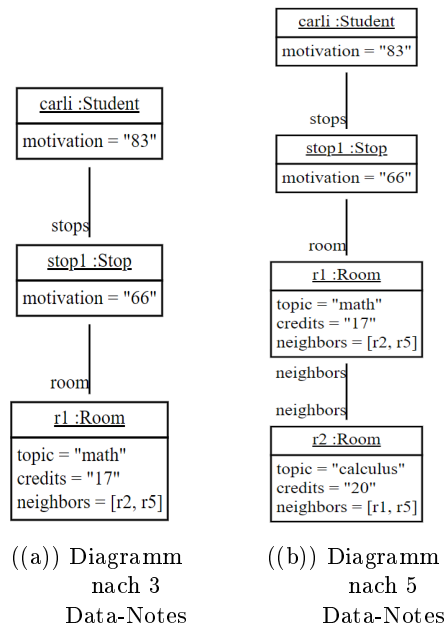


Abbildung 3.6: Mittels fulibWorkflows generierte Objektdiagramme

---

```

5 - data: Student carli
6   motivation: 83
7   stops: [stop1]
8   stops.back: student
9
10 - data: Stop stop1

```

---

Listing 21: Beispiel eines richtigen Data-Notes

TODO: vlt listing 21 nach abbildung 3.6, brauchst du die abbildung hier oder war die nur für oben?

Anhand des Data-Notes aus Listing 21 werden folgend die eben erwähnten Spezifikationen näher betrachtet. Der *Name*, welcher in Zeile 5 nach dem Doppelpunkt vergeben wird, muss immer die Form: <Klassenname> <Objektnamen> besitzen. Zudem muss beim ersten Aufkommen einer Assoziation sowohl die Hin- als auch Rückrichtung beschrieben werden. Eine Assoziation wurde in Zeile 7 und 8 beschrieben, bei dieser wird mittels *stops* der Bezeichner für die Hinrichtung festgelegt. Mit den eckigen Klammern um den Text *stop1* wird ausgesagt, dass die Hinrichtung eine To-Many-Assoziation ist, somit ein Student mehrere Stops haben kann. Die Rückrichtung wird in Zeile 8 durch *stops.back* definiert, gemeinsam mit dem Wert *student* wird der Bezeichner der Rückrichtung auf *student* gesetzt. Durch die fehlenden eckigen Klammern ist die Kardinalität der Rückrichtung 0 oder 1.

TODO: Zeile 10 aus Listing 21 beschreiben

---

```
39  ....
40  ClassModelManager mm = new ClassModelManager();
41
42  // Create a map String,Clazz containing every possible class from the Data notes
43  createClazz(mm);
44
45  // Build all associations and put it into a global list
46  // Also Build a list of attributes, that are not allowed to be created
47  buildAssociations();
48
49  // Create all attributes
50  createAttributes(mm);
51
52  // Create all associations
53  createAssociations(mm);
54  ....
```

---

Listing 22: Schritte zum Aufbau eines Klassenmodells

Anhand dieser Anforderungen an einen Data-Note ist es möglich ein Klassenmodell aus den Notes zu abstrahieren. In Listing 22 sind die Schritte, welche zum Aufbau eines Klassenmodells nötig sind, aufgelistet. Diese stammen samt Kommentaren aus der Implementierung, die *ClassModelManager*-Klasse ist in der fulib-Bibliothek enthalten. Im ersten Schritt, Zeile 43, werden alle Klassen in einer Map abgelegt, wobei als Key der Name der Klasse und als Value ein *Clazz*-Objekt fungiert. Hierbei wird über alle Data-Notes iteriert und die darin enthaltenen Klassen aus dem Namen des Data-Notes extrahiert. In Schritt zwei, Zeile 47, werden die vorhandenen Assoziationen erstellt. Hierfür wurde eine Klasse namens *Association* erstellt, welche zum Zwischenspeichern der für eine Assoziation wichtigen Daten speichert. Darunter zählen Klasse, Bezeichner und Kardinalität für Hin- und Rückrichtung. Neben diesen Daten werden in einer Liste von Strings Schlüsselwörter gespeichert, welche als Assoziation fungieren und nicht als Attribut betrachtet werden dürfen. Die wichtigsten Spezifikationen für eine funktionierende Assoziation wurde bereits zuvor anhand von Listing 21 erläutert. Weiterhin ist es wichtig, dass die verwendeten Objektnamen in der Workflow-Beschreibung konsistent sind, da ansonsten keine Zielklasse ermittelt werden kann. Während des Aufbaus einer Assoziation wird über alle Data-Notes iteriert, um die Informationen der Zielklasse zu erhalten. Der nächste Schritt, Zeile 50, erstellt Attribute für alle Klassen. Hierbei kommt die zuvor erstellte Liste an Schlüsselwörtern zum Tragen, um auszuschließen, dass eine Assoziation ebenfalls als Attribut aufgefasst wird. Im letzten Schritt zur Abstrahierung eines Klassenmodells aus den Data-Notes werden die zuvor in Zeile 47 gebauten Assoziationen im Klassenmodell erstellt. In diesem Schritt wird darauf geachtet, dass alle benötigten Informationen für eine Assoziation vorhanden sind. Durch die Nutzung von fulib und dem *ClassModelManager* muss nicht auf Duplikate bei Attributen oder Assoziationen geachtet werden.

Nachdem das Klassenmodell erstellt wurde, wird dieses ebenfalls mit FulibTools zu einem Diagramm weiterverarbeitet. Der Ablauf hierbei ist ähnlich zu der Generierung von Objektdiagrammen. Nachdem das Diagramm generiert wurde, wird dieses ebenfalls auf dem Dateisystem gelöscht und der Inhalt des Diagramms als String zurückgegeben.

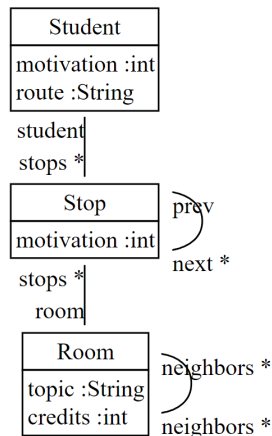


Abbildung 3.7: Mittels fulibWorkflows generiertes Klassendiagramm

In Abbildung 3.7 ist das aus der vorherigen Kapitel verwendete Beispiel zur Grundlage der Generierung eines Klassendiagramms genutzt worden. Das Klassendiagramm besteht aus drei Klassen und beinhaltet alle beschriebenen Assoziationen bestehend aus Kanten, Bezeichnern an beiden Seiten und den entsprechenden Kardinalitäten. Bei den Kardinalitäten kennzeichnet ein “\*” eine To-Many-Assoziation. Die Typen der Felder einer Klasse können lediglich zwischen “String” und “int” unterschieden werden.

## 3.2 Frontend des Web-Editors

Nachdem die erste Hälfte der Implementierung durch fulibWorkflows abgeschlossen ist, konzentrieren sich dieses und das folgende Kapitel um den dazugehörigen Web-Editor. Der Web-Editor besteht aus einem Frontend und einem Backend, welche beide über Heroku deployt wurden und somit<sup>1</sup> erreichbar sind.

In Abbildung 3.8 ist die Oberfläche des Web-Editors dargestellt. Dieser besteht aus vier verschiedenen Bereichen. Der erste dieser Bereiche ist die Navigationsleiste, welche den oberen Rand der Oberfläche einnimmt. In dieser existieren zuerst, von links nach rechts, zwei Buttons, welche das Theme des Code-Editors abändern. Damit ist es möglich einen Light- oder Dark-Mode zum Schreiben des Codes zu verwenden. Danach folgt ein Dropdown-Menü, mit welchem es möglich ist verschiedene vorgefertigte Beispiele zu laden. Das ausgewählte Beispiel wird automatisiert nach der Auswahl ans Backend geschickt und dort generiert, sodass nach einer kurzen Wartezeit ein ES-Board und falls vorhanden Mockups

<sup>1</sup>unter <https://workflows-editor-frontend.herokuapp.com/>

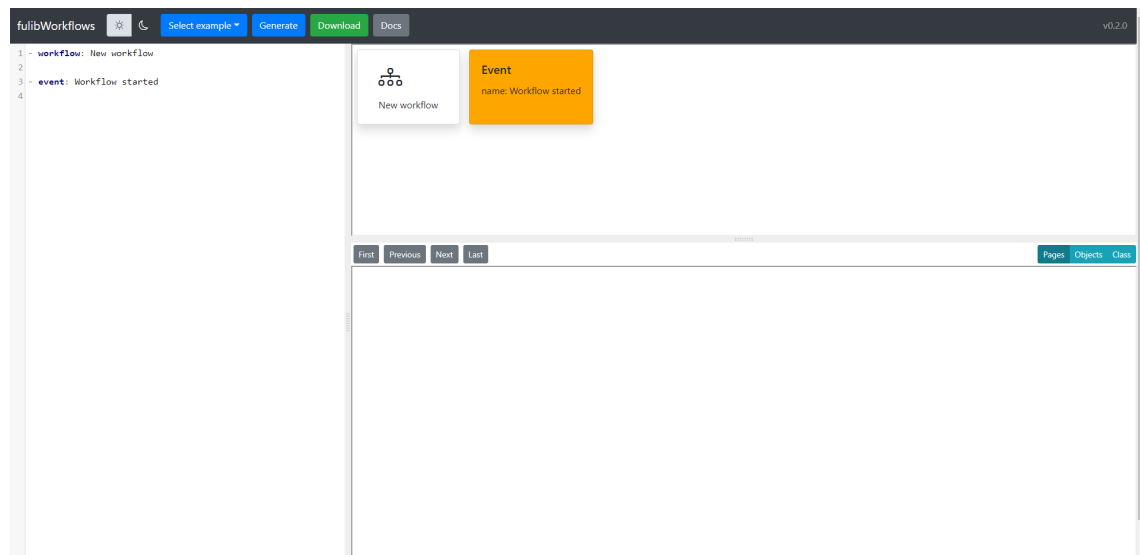


Abbildung 3.8: Oberfläche des Web-Editors für fulibWorkflows

und Diagramme angezeigt werden können. Als Nächstes folgt ein Knopf zum Anstoßen einer Generierung, nachdem dieser Knopf gedrückt wurde und die Generierung angestoßen ist, erscheint ein Ladekreis in dem Knopf, um als Indikator dafür zu dienen, dass der Prozess noch nicht abgeschlossen ist. Die Web-Anwendung ermöglicht es zusätzlich die in der Oberfläche erstellten Dateien herunterladen. Hierfür öffnet sich ein Pop-Up, nachdem der Download-Knopf betätigt wurde. Dieses Fenster ist in Abbildung 3.9 dargestellt.

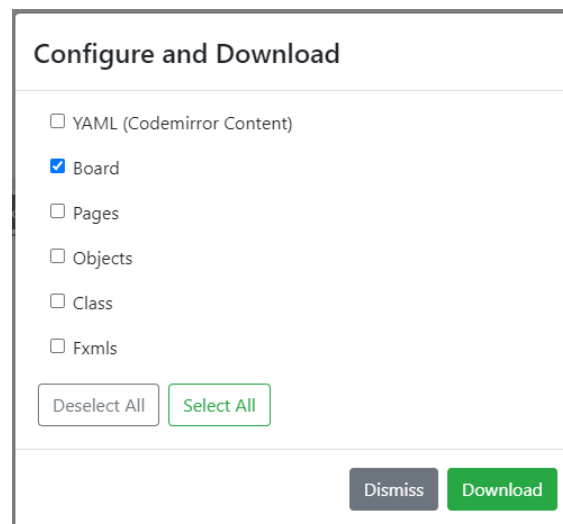


Abbildung 3.9: Download-Fenster

Im sich öffnenden Fenster erhält der Benutzende die Möglichkeit auszuwählen, welche Dateien heruntergeladen werden sollen. Es können einzelne Dateien heruntergeladen werden, wie der Inhalt des Code-Editors, das generierte ES-Board oder aber alle Dateien die durch die aktuelle Workflow-Beschreibung im Code-Editor generiert wurden. Mit einem Klick auf den Download-Knopf, welcher sich neben dem Dismiss-Knopf befindet, wird eine Zip-Datei generiert und automatisch durch den jeweiligen Browser heruntergeladen.

Für den Fall, dass der Nutzende noch keine Erfahrungen mit der Syntax von `fulibWorkflows` gemacht hat, kann die Dokumentation, welche auf Englisch verfasst ist, mit einem Klick auf den grauen Docs-Knopf aus Abbildung 3.8 geöffnet werden. Die Dokumentation stammt aus dem GitHub-Repository von `fulibWorkflows`. Letztlich befindet sich ganz rechts die aktuelle Versionsnummer des Web-Editors.

### 3.2.1 Code-Editor

In diesem Kapitel wird der Code-Editor, welcher das Herzstück der Anwendung ist, erläutert. Dieser befindet sich auf der linken Seite der in Abbildung 3.8 dargestellten Oberfläche. Wie zuvor bereits beschrieben wurde hierbei ein Codemirror verwendet. Durch die Verwendung der `ngx-codemirror` Bibliothek vereinfacht sich das Einbinden eines Codemirrors in eine Angular-Anwendung. Dadurch kann eine Konfiguration des Codemirrors über ein Options-Objekt übergeben werden. Die Konfiguration des Codemirrors ist in Listing 23 dargestellt.

---

```
49  this.codemirrorOptions = {
50    lineNumbers: true,
51    theme: this.currentCodemirrorTheme,
52    mode: 'yaml',
53    extraKeys: {
54      'Ctrl-Space': 'autocomplete',
55      'Ctrl-S': generateHandler,
56    },
57    autofocus: true,
58    tabSize: 2,
59  };
```

---

Listing 23: Codemirror-Konfiguration

In Zeile 50 werden die Zeilennummern am linken Rand des Codemirrors aktiviert. Das aktuelle Theme wird in der folgenden Zeile ebenfalls übergeben. Da es zwei verschiedene Themes gibt, welche verwendet werden können, wird der Wert aus einer weiteren Variable übernommen. Codemirror stellt bereits zahlreiche verschiedene Themes für Light-Mode oder Dark-Mode bereit. Initial wird der Codemirror mit dem Light-Theme geladen, das Umstellen des Themes erfolgt über die entsprechenden Knöpfe in der Navigationsleiste, wie bereits zuvor beschrieben. Abbildung 3.10 zeigt den Codemirror in beiden Modi, wobei sich sowohl Light- als auch Dark-Mode nah an den Standard-Modi von IntelliJ orientieren.

In Zeile 52 aus Listing 23 wird die Programmiersprache des Editors festgelegt. Da die Workflow-Beschreibungen von `fulibWorkflows` in `.es.yaml`-Dateien angelegt/gespeichert werden, wird die Programmiersprache auf `yaml` festgelegt. Dieser Modus wird von Codemirror selbst bereitgestellt und bedarf zur Verwendung einen Import in der `main.ts`-Datei

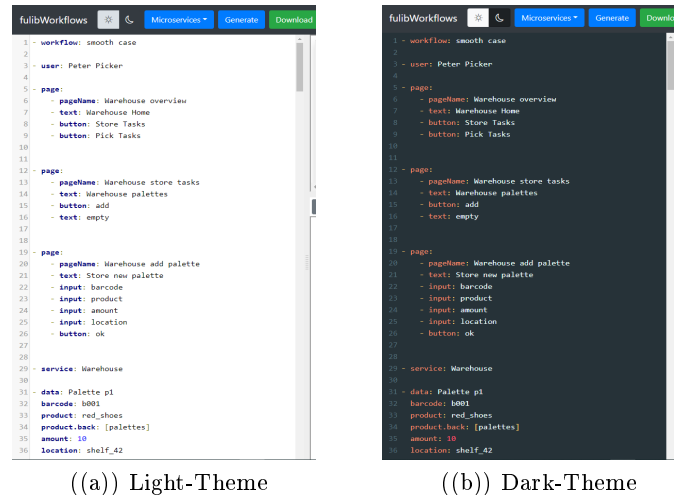


Abbildung 3.10: Themes des Codemirrors

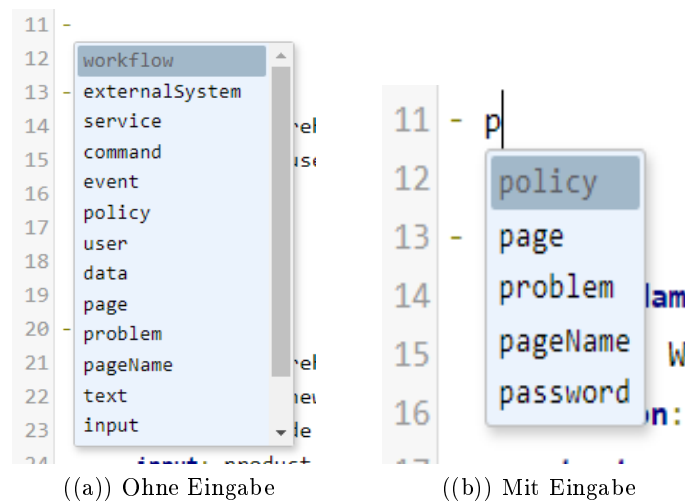


Abbildung 3.11: Autovervollständigung

der Angular-Anwendung. Über die Option *extraKeys* können Tasten oder Tastenkombinationen an weitere Funktionen gekoppelt werden. In Zeile 54 wird das Öffnen einer Liste an vorgeschlagenen Wörtern geöffnet, nachdem der Nutzende die Tastenkombination “Strg+Leertaste” gedrückt hat. Damit dies funktioniert, benötigt es das Importieren des *show-hint*-AddOns von Codemirror in der *main.ts*-Datei.

Dieses Add-On erstellt die in Abbildung 3.11((a)) angezeigte Liste. Der Inhalt der Liste wird über ein in dieser Arbeit erstelltes Codemirror Add-On gefüllt. Hierbei fällt auf, dass auch Schlüsselwörter angezeigt werden, welche nur im Kontext eines Page-Notes sinnvoll sind. Das eigens geschriebene Add-On ist nicht kontextsensitiv und besitzt somit nicht die gleichen Funktionen wie die Autovervollständigung, welche eine IDE durch das JSON-Schema bereitstellt. Der Code des geschriebenen Add-Ons befindet sich im Anhang dieser Arbeit, hierbei wird über die aktuelle Position des Cursors das aktuelle Wort extrahiert. Damit ist es möglich über die Liste der Schlüsselwörter zu iterieren und zu prüfen, welche Vorschläge sinnvoll sind, sollte ein Wort bereits begonnen sein. Dies ist anhand von Ab-

bildung 3.11((b)) genauer zu erkennen. Hierbei wurde bereits der Buchstabe *p* eingetippt und das Add-On bietet zur Vervollständigung nur Wörter an, welche mit *p* beginnen.

Des Weiteren wird durch das Betätigen der Tastenkombination “Strg+S” die Generierung angestoßen. Bevor die Daten aus dem Codemirror zur Generierung an das Backend gesendet werden, werden diese auf Richtigkeit überprüft. Da bereits ein JSON-Schema für *fulibWorkflows* existiert, wurde eine Bibliothek ausgewählt, welche einen Text über ein JSON-Schema validieren kann. *Ajv* ist ein solcher Validierungsmechanismus, allerdings kann mit *Ajv* lediglich ein JSON-Objekt mittels JSON-Schema validiert werden[Pob21]. Somit wurde zum Umwandeln des Textes aus dem Codemirror zu einem JSON-Objekt die Bibliothek *js-yaml* verwendet[Zap21].

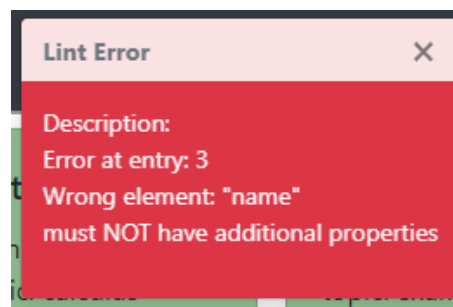


Abbildung 3.12: Validierungserror als Toast

Sobald während der Validierung ein Problem mit der Eingabe erkannt wird, gibt diese einen Fehler zurück. Dieser Fehler wird anschließend als Toast in der Oberfläche angezeigt, um den Nutzenden darauf hinzuweisen, an welcher Stelle die Eingabe im Codemirror nicht dem JSON-Schema von *fulibWorkflows* entspricht. Eine solche Fehlermeldung ist in Abbildung 3.12 dargestellt. Hierbei wurde einem Note ein Attribut *name* zugeordnet. Da der besagte Note allerdings ein *User* ist und im JSON-Schema festgelegt ist, dass ein User-Note keine zusätzlichen Attribute besitzen darf, entsteht der in der Abbildung gezeigte Fehler. Diese Fehlermeldung wird nach 20 Sekunden automatisch geschlossen. Für die Darstellung von Toasts wurde die gleichnamige Komponente von *ng-bootstrap* verwendet. Sollte die Eingabe valide sein, so wird diese über einen Service an das Backend geschickt.

### 3.2.2 Darstellung generierter Dateien

Die anderen beiden Bereiche der Oberfläche, welche bisher nicht erläutert wurden, stellen jeweils einen bestimmten Teil der generierten Dateien dar. Beide Bereiche aus Abbildung 3.8 bestehen aus einem *IFrame*, wobei der obere *IFrame* das generierte ES-Board anzeigt. Andererseits übernimmt der untere *IFrame* die Anzeige der HTML-Mockups, den Objektdiagrammen und dem Klassendiagramm.

Bevor auf die Darstellung und Funktionen der *IFrames* eingegangen wird, wird zuerst betrachtet in welcher Form die generierten Dateien vom Backend im Frontend verarbeitet



---

```

1  export interface GenerateResult {
2    board: string,
3    pages: Map<number, string>,
4    numberOfPages: number,
5    diagrams: Map<number, string>,
6    numberOfDiagrams: number,
7    classDiagram: string,
8  }

```

---

Listing 24: Modell der vom Backend empfangenen Daten

werden. Die Form ist in Listing 24 dargestellt, in einem `GenerateResult`-Objekt werden alle generierten Dateien und Zusatzinformationen abgespeichert. Bei den Zusatzinformationen handelt es sich um die Anzahl der generierten Diagramme und HTML-Mockups. Die generierten Dateien werden als reiner String behandelt. Um den Zugriff auf einzelne Objektdiagramme oder Mockups zu vereinfachen, sind diese jeweils in einer Map abgespeichert. Bei den Maps ist jedem Diagramm/Mockup eine eindeutige Nummer zugeordnet.

Der obere `IFrame` erhält als Eingabe das ES-Board und stellt dieses dar. Dies ist möglich, da das ES-Board eine valide HTML-Datei ist, welche durch einen `IFrame` dargestellt werden kann. Hierbei war es nötig eine Pipe zu erstellen, welche den *DomSanitizer* von Angular auf der Eingabe umgeht[Swa17]. Der *DomSanitizer* ist ein von Angular bereitgestellter Service, welcher Elemente aus der Eingabe entfernt die potenziell für Angreifer genutzt werden könnten, um Skripte auf der Anwendung auszuführen. Dies ist ein Sicherheitsmechanismus, um das sogenannte *Cross-Site-Scripting* auszuhebeln. Beim *Cross-Site-Scripting* können Angreifer JavaScript-Code in den Browsern anderer Nutzer ausführen und somit personenbezogene Daten erhalten[Jak13]. Durch die Pipe wird dieser Sicherheitsmechanismus umgangen und die Eingabe wird unverändert im `IFrame` geladen.

TODO: Heißt das, dass die Anwendung jetzt angegriffen werden kann? Das wäre ja blöd oder? -> Ja das heißt es. Lösungsansatz im Ausblick beschreiben und darauf hier verweisen

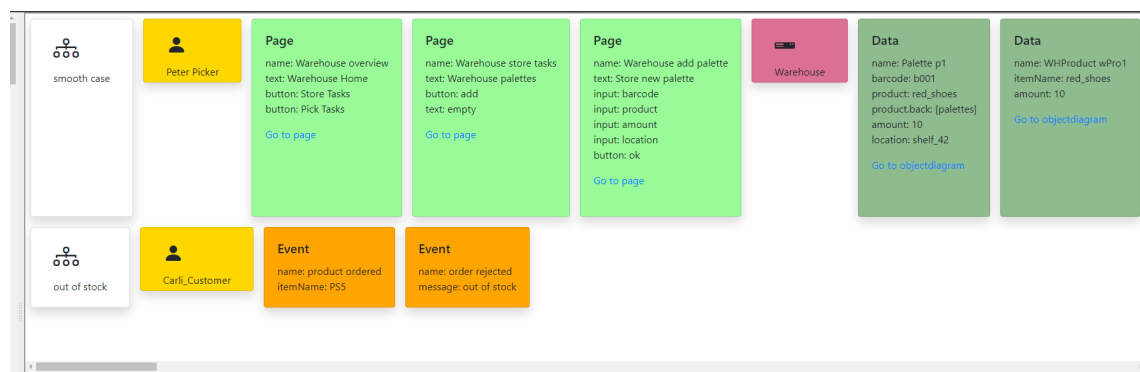


Abbildung 3.13: Event-Storming-Board in einem IFrame

In Abbildung 3.13 ist ein ES-Board für das im Web-Editor vorhandene Microservices-Beispiel dargestellt. Hierbei ist zu erwähnen, dass die Bereiche: Code-Editor, ES-Board-IFrame und der IFrame zur Anzeige der Mockups und Diagramme beliebig vergrößert oder verkleinert werden kann. Um dies zu ermöglichen wurde die Bibliothek *angular-split* verwendet. Diese ermöglicht es Bereiche zu definieren und darin Inhalt zu platzieren, sowie die Veränderung der Größen der Bereiche bereitzustellen[Gai21]. Das Abändern der Größe von Bereichen ermöglicht das Hervorheben des Editors, des ES-Boards oder der Mockups/Diagramme.

Für den unteren IFrame gelten die gleichen Gegebenheiten wie für den oberen IFrame. Da es mehrere Mockups oder Diagramme geben kann, existieren vier Knöpfe, mit welchen es möglich ist zwischen den Mockups/Diagrammen zu wechseln. Doch der Wechsel zwischen diesen Elementen ist nicht nur über die vier Knöpfe möglich, sondern ebenfalls über die entsprechenden Notes aus dem ES-Board-IFrame. Für jeden Page- oder Data-Note existiert in der Anzeige ein Link, welcher als Knopf fungiert, mit welchem zu dem Mockup oder Diagramm des Notes gewechselt werden kann.

---

```
1  (<any>window).setIndexFromIframe = this.setIndexFromIframe.bind(this);
```

---

Listing 25: Bereitstellung einer Methode

Hierfür ist eine Methode erstellt worden, welche für die gesamte Oberfläche sichtbar ist. In der Generierung mittels *fulibWorkflows* wird dieser Link erstellt, darin wird die bereitgestellte Methode mittels `window.parent.setIndexFromIframe(0, 'pages');` aufgerufen. **TODO: hmmm vielleicht auch ein snippet zur erklärung einfügen, statt das direkt in den text zu schreiben?** Hierbei wird über das Fenster auf die oberste Komponente der Anwendung zugegriffen, dort ist durch den Code aus Listing 25 die Methode *setIndexFromIframe* bereitgestellt. Neben dem Index wird der Methode ebenfalls ein String übergeben, mit diesem ist es möglich zwischen Page und Object-Darstellung automatisch beim Klicken des Links zu wechseln.

**TODO: so generell zu diesem absatz... ist das sehr wichtig? dann vielleicht ausführlicher erklären... wenn unwichtig.. vielleicht weglassen? kA also ich wurde da jetzt nicht so richtig schlau draus beim lesen**

Zuletzt existieren drei weitere Buttons auf Höhe der Buttons zum Wechseln der Page oder des Diagramms, welche die Anzeige der generierten Dateien manuell umschaltet. **TODO: ja wo denn ich seh nix** Hierbei kann zwischen dem Anzeigen der Mockups (Pages), der Objektdiagramme und des Klassendiagramms umgeschaltet werden.

### 3.3 Backend des Web-Editors

Das Backend des Web-Editors basiert auf einem mit Spring Initializr generiertem Java-Projekt. Zusätzlich wurden bei der Konfiguration die Dependencies für eine Spring-Web-Anwendung hinzugefügt. Neben den eben genannten Dependencies wurde lediglich fulib-Workflows als weitere Bibliothek zum Backend hinzugefügt.

Die verfügbaren Endpunkte des Backends werden in einem Controller bereitgestellt. In diesem Fall ist dies der FulibWorkflowsController, welcher mit zwei Annotations versehen ist. Die erste Annotation ist `@Controller`, welche eine Klasse als Controller deklariert. Bei der zweiten Annotation handelt es sich um `@CrossOrigin()` mit welcher es ermöglicht wird, problemlos mit dem Frontend zu interagieren. **TODO: wenn das schon so dort steht, könnte sich dem leser die frage stellen, welche probleme man hat, wenn mans nicht macht?** Im FulibWorkflowsController sind Endpunkte für die Generierung und den Download definiert. Diese Definitionen werden in Listing 26 zur Veranschaulichung dargestellt.

---

```
15 @Controller
16 @CrossOrigin()
17 public class FulibWorkflowsController {
18     @PostMapping(path = "/generate", consumes = MediaType.ALL_VALUE)
19     @ResponseBody
20     public String generate(@RequestBody String yamlData) {
21         return fulibWorkflowsService.generate(yamlData);
22     }
23
24     @PostMapping(path = "/download",
25                 consumes = MediaType.ALL_VALUE,
26                 produces = "application/zip")
27     @ResponseBody
28     public byte[] download(@RequestBody String yamlData,
29                          @RequestParam Map<String, String> queryParams) {
30         return fulibWorkflowsService.createZip(yamlData, queryParams);
31     }
32 }
```

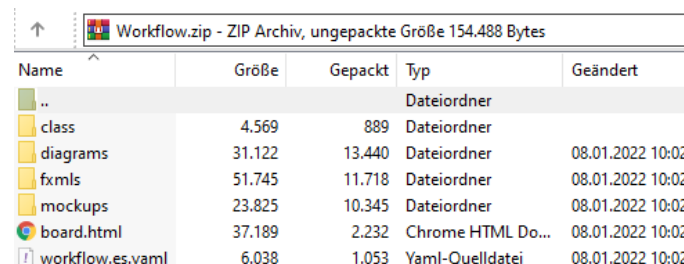
---

Listing 26: Definition der Endpunkte

Beide Endpunkte können mit einem POST-Request angesprochen werden, da sowohl beim Download als auch der Generierung der Inhalt der YAML-Datei vom Frontend mitgeschickt wird. Dies sorgt dafür, dass bei beiden Endpunkten die Generierung durchgeführt wird, damit kein Abspeichern von Dateien stattfinden und keine IDs für eine Beschreibung angelegt und verwaltet werden müssen. Damit die Endpunkte auf diesen Inhalt zugreifen können, ist der erste Methodenparameter mit der `@RequestBody`-Annotation versehen. Hierbei wird auf den Inhalt zugegriffen, welcher vom Frontend in der Post-Anfrage mitgesendet wurde. Der Download-Endpunkt erhält zusätzlich zu der YAML-Beschreibung

ebenfalls Query-Parameter. Diese enthalten die aus dem Download-Pop-Up des Frontends angegebenen Dateien, welche heruntergeladen werden sollen. Mittels der `@ResponseBody`-Annotation können der Antwort des Backends weitere Daten hinzugefügt werden. In welcher Form diese sind, resultiert aus dem Rückgabebetyp der jeweiligen Methode unterhalb der Annotation. Bei der `generate`-Methode wird ein JSON-Objekt als String zurückgegeben. Die `download`-Methode wiederum generiert ein Zip-Archiv und versendet die Daten als `ByteArray`. Beide Methoden des Controllers reichen die erhaltenen Daten an den `FulibWorkflowsService` weiter, welcher sich um die Generierung über `fulibWorkflows` und der Erstellung des Zip-Archives kümmert.

Im `FulibWorkflowsService` wird sowohl in der `generate`- als auch der `createZip`-Methode, welche vom Controller aufgerufen werden, der YAML-String über `fulibWorkflows` generiert. Dabei nutzt das Backend die `generateAndReturnHTMLsFromString`-Methode des `BoardGenerators` von `fulibWorkflows`. **TODO: wenn es kein Snippet gibt, wo die auftaucht, brauchst du die Methode nicht beim Namen nennen, sondern nur erklären, was die macht - > Schauen wie ich das Umschreibe** Im Anschluss wird aus der Map von generierten Dateien ein `GenerateResult` erstellt, welches von der `generate`-Methode zu einem JSON-String weiterverarbeitet und zurück an den Controller gegeben wird. Nachdem das `GenerateResult`-Objekt in der `createZip`-Methode erstellt wurde, beginnt das Erstellen des Zip-Archivs. Auf Grundlage der Query-Parameter werden die Dateien zum Zip-Archiv hinzugefügt, welche im Frontend ausgewählt werden. **TODO: hab das gefühl ein ganz ähnlicher satz ist nicht lange her -> Ich sehs nicht xD** In Abbildung 3.14 wurden alle Dateien heruntergeladen, welche von `fulibWorkflows` generiert wurden.



Name	Größe	Gepackt	Typ	Geändert
..			Dateiordner	
class	4.569	889	Dateiordner	
diagrams	31.122	13.440	Dateiordner	08.01.2022 10:02
fxmls	51.745	11.718	Dateiordner	08.01.2022 10:02
mockups	23.825	10.345	Dateiordner	08.01.2022 10:02
board.html	37.189	2.232	Chrome HTML Do...	08.01.2022 10:02
workflow.es.yaml	6.038	1.053	Yaml-Quelldatei	08.01.2022 10:02

Abbildung 3.14: Inhalt eines heruntergeladenen Zip-Archivs

Um die Dateien strukturiert zu halten, werden Diagramme und Mockups in eigenen Unterordnern abgelegt. Hierbei wird sowohl zwischen Objektdiagrammen und dem Klassendiagramm, als auch zwischen HTML- und FXML-Mockups unterschieden. Auf oberster Ebene wird die Workflow-Beschreibung und das generierte ES-Board abgelegt, da diese die Grundlage für alle weiteren Dateien bilden. Das Klassendiagramm ist im Ordner `class` abgelegt, die Objektdiagramme im `diagrams`-Ordner. Im Gegensatz hierzu werden die HTML-Mockups im `mockups`-Ordner hinterlegt, die gleichen Mockups im FXML-Format werden im `fxmls`-Ordner platziert.

## 4 Evaluation

In diesem Kapitel werden mittels eines Expertengesprächs die bisher in dieser Arbeit gesammelten Erkenntnisse über Event Storming diskutiert. Aus dieser Diskussion und einer anschließenden Präsentation des erstellten Web-Editors wird evaluiert, ob die zu Beginn in Kapitel 1.2 **TODO: steht anscheinend erst in Kapitel 5?** gesetzten Ziele aus der Sicht des Experten erfüllt wurden.

Eine Aufnahme des gesamten Gesprächs ist über den QR-Code im Anhang erreichbar.

### 4.1 Expertengespräch

Bevor dem Experten die in dieser Arbeit erstellte Anwendung präsentiert wurde, wurden Probleme beim RE, welche aus den Erfahrungen des Experten hervorgingen, evaluiert. Direkt zu Beginn des Gesprächs bestätigte sich, dass es bei Gesprächen mit Kunden häufig zu einer Diskrepanz zwischen beschriebenen Benutzeroberflächen und der tatsächlichen Implementierung kommt<sup>1</sup>. Dabei fallen, wie in Kapitel 1.1 beschrieben, Probleme bei der Benutzung einer Anwendung erst nach einer Implementierung auf, wodurch weitere Anpassungen getätigt werden müssen. Durch eben diese Anpassungen entstehen während der Softwareentwicklung weitere Kosten, da erneut Gespräche mit dem Kunden darüber geführt werden müssen, wie Oberflächen auszusehen haben, welche Funktionen noch fehlen oder falsch verstanden wurden. Bei bisherigen ES-Sessions wurden bisher keine Anwendungen zur Erstellung von Mockups verwendet, da der Fokus auf dem Ablauf und nicht den Oberflächen liegt. Daraus resultiert, dass es mit einem entsprechendem Tooling ermöglicht werden kann eine Kostenreduzierung für das Unternehmen zu erreichen<sup>2</sup>.

Da als Methodik für diese Arbeit das ES gewählt wurde, wurde ebenfalls darüber diskutiert, ob das DDD und ES auch in der praktischen Anwendung während der Arbeit des Experten für ein produktiveres Arbeiten gesorgt haben. Die Erkenntnisse und Beschreibung aus Kapitel 2.1 konnten vom Experten bestätigt werden<sup>3</sup>. Dies bezieht sich sowohl auf Bestandteile des DDD als auch die Verwendung von Event Storming für initiale Meetings. Hierbei wurde vor allem auf das Verständnis des Prozesses während einer ES-Session eingegangen. Nicht nur Entwickler konnten dadurch besser verstehen, was für eine Anwendung

---

<sup>1</sup>Siehe 7.2 Expertengespräch ab Minute 1:30

<sup>2</sup>Siehe 7.2 Expertengespräch ab Minute 3:43

<sup>3</sup>Siehe 7.2 Expertengespräch ab Minute 4:58

benötigt wird, auch Teilnehmer aus dem Unternehmen erlangten einen besseren Überblick über Bereiche in welchen diese nicht arbeiten.

Für ES-Sessions wurden bereits Online-Anwendungen verwendet, dies geschah in erster Linie aufgrund der COVID-19-Pandemie<sup>4</sup>. Dadurch wurde Miro, ein interaktives Online-Whiteboard, für Event Stormings verwendet.

Beim ES werden bei einem Gespräch mit Kunden keine UML-Diagramme zur Ermittlung eines Datenmodells verwendet, da diese für nicht technisch versierte Personen keinen Mehrwert bieten<sup>5</sup>. Dennoch wird der generelle Einsatz von UML-Diagrammen vor Entwicklungsstart einer Anwendung als sinnvoll erachtet, da sich Entwickler im Vorhinein ein Datenmodell überlegen können, um daraus weitere Fragen gegenüber den Kunden zu stellen. **TODO: Gibts dafür auch ne Quelle? -> Gutes Argument, nein gibts nicht**

Nachdem grundlegende Probleme aus der Praxis und verwendete Technologien und Techniken während des ersten Teils des Gespräches festgelegt wurden, folgte die Vorstellung der Anwendung mit allen dazugehörigen Funktionen. Da eine der Funktionen der Anwendung das Erstellen von Datenmodellen mittels UML-Diagrammen ist, verwies der Experte an den ähnlichen Aufbau der Anwendung zu anderen Datenmodellierungs-Tools wie *dbdiagram.io*. Dies kam beim Experten bereits in der Praxis zum Einsatz, um ein Datenmodell für Datenbanken zu erstellen. Ebenso wie das in dieser Arbeit erstellte Tool, existiert dort ebenfalls ein Editor, in welchem mittels einer Beschreibungssprache ein Diagramm erzeugt wird, welches ein Datenmodell darstellt. Dies sei wie zuvor erwähnt besonders in der ersten Phase der Softwareentwicklung, dem RE, von Nutzen<sup>6</sup>.

Ebenfalls fiel auf, dass die Darstellung des ES-Boards in mehreren Workflows einer Anordnung an einem Zeitstrahl gleicht. Es wird somit ein zeitlicher Ablauf über Events modelliert. Dies ist ein Grundbaustein des ES, somit geht dieser durch die Unterteilung in Workflows nicht verloren<sup>7</sup>.

Das zu Beginn beschriebene Problem von Diskrepanzen bei der Entwicklung von Oberflächen wurde durch die Möglichkeit der Generierung von Mockups, welche grundlegende Funktionalität aufweisen, positiv aufgefasst. Durch die Möglichkeit Mockups über Buttons untereinander zu verknüpfen, kann in der Anwendung ein Clickdummy erstellt werden. Bei einem Clickdummy handelt es sich um Oberflächen, welche so miteinander verbunden sind, dass diese eine Anwendung simulieren. Durch eine Generierung dieser Oberflächen während einer ES-Session könne man den späteren Nutzer, solange dieser innerhalb der Teilnehmer ist, ihren Arbeitsablauf anhand des Clickdummies durchführen lassen. Daraus können bereits vor Start der Entwicklung Probleme und Ungereimtheiten bei der Modellierung gelöst werden<sup>8</sup>.

Es wurden zudem weitere Funktionen besprochen, welche einen positiven Effekt auf die Verwendung haben können, diese werden in Kapitel 6 ausgeführt.

---

<sup>4</sup>Siehe 7.2 Expertengespräch ab Minute 12:20

<sup>5</sup>Siehe 7.2 Expertengespräch ab Minute 9:35

<sup>6</sup>Siehe 7.2 Expertengespräch ab Minute 29:29

<sup>7</sup>Siehe 7.2 Expertengespräch ab Minute 39:17

<sup>8</sup>Siehe 7.2 Expertengespräch ab Minute 33:40

## 5 Fazit

**TODO: Fazit und Ausblick zusammenführen?**

Neben der Beantwortung der Frage, ob die Ziele aus Kapitel 1.2 erreicht wurden, wird auch auf Probleme während der Implementierung eingegangen.

Anhand der Evaluierung zeigt sich, dass das Ziel erreicht wurde. Es wurde eine Web-Anwendung erstellt, welche ein Event Storming durch die Generierung und zeitnahe Darstellung von Dateien aus Events unterstützen kann<sup>1</sup>. Hierbei kann ebenfalls eine Datenmodellierung vorgenommen werden, welche initial bei einem Projekt von Entwicklern genutzt werden kann. Die Beschreibungssprache, auf welcher die Generierung basiert, erfüllt die Ansprüche der Event-Notes für das Event Storming.

Dennoch ist zu sagen, dass die in dieser Arbeit erstellte Anwendung kein Ersatz für Stifte und Post-its oder ein Online-Tool wie Miro ist. In seiner aktuellen Form ist die Anwendung lediglich ein unterstützendes Werkzeug für ein Event Storming, da ein Board nur von einer Person bearbeitet werden kann. Dies verstößt gegen eine Grundlage des Event Stormings, die Freiheit der Teilnehmer jederzeit neue Events zu erstellen. In dieser Arbeit wurde allerdings die Basis für ein Online-Tool geschaffen. Dies wurde ebenfalls während des Expertengesprächs bestätigt<sup>2</sup>. Aufbauend auf dieser Basis wurden Erweiterungen besprochen, damit das Online-Tool in der Praxis einsetzbar ist. Die Erweiterungen werden in Kapitel 6 erläutert.

**TODO: Limitierungen von Heroku hervorheben** <https://railsautoscale.com/heroku-free-dynos/> Hierbei ist zu beachten, dass das initiale Laden der Webseite etwas Zeit in Anspruch nimmt, da die Anwendung auf Heroku nach einer gewissen Inaktivität in einen Ruhezustand versetzt wird und bei einem neuen Aufruf erst hochgefahren werden muss.

Während des Implementierens der Anwendung traten keine nennenswerten Probleme auf. Dies änderte sich allerdings, nachdem die Anwendung über Heroku bereitgestellt wurde. Das Problem hierbei war die Nutzung von *Graphviz* zur Generierung der UML-Diagramme im Backend. Während der Generierung wurde eine zusätzliche Javascript-Engine gestartet, welche für *Graphviz* benötigt wurde. Dadurch stieg der Speicherverbrauch des Backends

---

<sup>1</sup>Siehe 7.2 Expertengespräch ab Minute 44:54

<sup>2</sup>Siehe 7.2 Expertengespräch ab Minute 51:06

und sorgte für einen Fehler, welcher die Generierung aller Dateien beeinflusste. Somit konnten weder neue ES-Boards noch alle Funktionen bei den Mockups richtig generiert werden. Dies sorgte während des Expertengesprächs für Pausen, in denen das Backend neu gestartet werden musste. Diese Neustarts mussten nach fast jeder Generierung wiederholt werden. Um diese Problematik zu umgehen, wurde aus der Spring-Boot-Anwendung eine Jar-Datei erstellt und diese zusammen mit einer *Graphviz*-Installation in einem Docker-Image vereinigt[Cat20]. Somit muss keine zusätzliche Javascript-Engine gestartet werden, da *Graphviz* über die vorinstallierte Instanz genutzt werden kann. Durch die Verwendung eines Docker-Images musste das Deployment auf Heroku angepasst werden. Heroku bietet eine eigene Registry an, in welcher Docker-Images bereitgestellt werden können[Sal22a]. Nachdem dies funktionierte, wurde die Anwendung mittels mehreren Generierungen großer Workflow-Beschreibungen getestet und das Speicherproblem gelöst.

Die im Web-Editor vorhandenen Beispiele wurden in dem Editor selbst verfasst. Bei der Arbeit mit der Anwendung sind mehrere Feature-Anfragen aufgekommen. Die Autovervollständigung im Codeeditor ist bisher nicht kontextabhängig. Es werden immer alle möglichen Schlüsselwörter angeboten, auch wenn diese nach dem JSON-Schema nicht valide sind. Hier erhält der Nutzende erst bei dem Beginn der Generierung und der damit einhergehenden Validierung der Beschreibung ein Feedback. Um die Arbeit mit dem Tool für den Entwickelnden zu erleichtern, sollte diese Autovervollständigung verbessert werden und abhängig vom Kontext sinnvolle Vorschläge machen. Das eben erwähnte Feedback wird im Fehlerfall als Toast angezeigt, hierbei hat die Fehlermeldung eine bestimmte Form, welche nicht in jedem Fall aussagekräftig ist. Zudem wird zwar angezeigt, welches Element falsch ist, diese Stelle wird aber weder farblich hervorgehoben noch fokussiert.



## 6 Ausblick

Im Gespräch mit dem Experten aus Kapitel 4.1 entstanden ebenfalls weitere Funktionen, welche für die Verwendung des Tools in der Praxis von Vorteil wären. Eine Ausweitung der verwendbaren Elemente in den Mockups ist eine dieser Funktionen<sup>1</sup>. Dabei wurde angesprochen, dass es sich dabei um Standardelemente wie Checkboxes, Dropdowns und Datepicker handelt. Es wurde auch das Erstellen von Tabellen angesprochen, allerdings bedarf es hierbei einer Abwägung, ob es möglich ist eine Tabelle in der YAML-syntax zu definieren, welche nicht zu kompliziert für die Nutzenden ist und damit zu zeitaufwändig in einer ES-Session wäre. Neben diesen Elementen könne es in Betracht gezogen werden, URLs für Bilder oder sonstige Daten in Pages mit anzugeben und daraus bei der Generierung dynamisch Bilder in den HTML-Mockups hinzuzufügen. Diese Bilder könnten als Platzhalter oder zum Branding eines Mockups verwendet werden. Alternativ zu einer URL ist eine weitere Überlegung, dass man Bilder hochladen kann und diese dann anstelle einer URL hinterlegt<sup>2</sup>. Eine URL könnte auch eine API ansprechen, um Daten eines anderen Services zu erhalten. Weiterhin wünschte sich der Experte, dass es möglich ist Kommentare zu Notes hinzuzufügen<sup>3</sup>. Dies ist bei dem Online-Tool Miro, welches in Kapitel 1.4 erwähnt wurde, bereits möglich und wurde auch nach einer Session als Möglichkeit zum Verfeinern eines ES-Boards genutzt<sup>4</sup>.

Im Fazit wurde bereits klargestellt, dass das erstellte Online-Tool momentan kein Ersatz für Tools wie Miro ist. Um dies zu ändern, ist es nötig, dass jeder Teilnehmer die Möglichkeit hat Events zu einer Workflow-Beschreibung hinzuzufügen. Eine Umsetzung dieser Funktion benötigt jedoch weitreichende neue Technologien. Zwischen den Teilnehmern einer Session müssen die Änderungen im Editor synchronisiert werden, dafür muss zuvor die Funktion implementiert werden, dass es so etwas wie Sessions gibt, bei denen neue Personen über einen Link oder ein Token eingeladen werden können. Bevor diese Funktion umgesetzt wird, muss evaluiert werden, ob die erstellte Beschreibungssprache intuitiv und nutzbar ist. Zudem ist festzustellen, ob ein Bedarf für ein solches Tool existiert, oder die bisher erstellte Anwendung zur Unterstützung eines Event Stormings genügt.

Wie bereits in Kapitel 2.1.3 angedeutet wurde, soll das in dieser Arbeit erstellte Tool auch einen Einsatz in der Lehre bekommen. In der Veranstaltung *Programmieren und Model-*

<sup>1</sup>Siehe 7.2 Expertengespräch ab Minute 49:45

<sup>2</sup>Siehe 7.2 Expertengespräch ab Minute 40:50

<sup>3</sup>Siehe 7.2 Expertengespräch ab Minute 48:02

<sup>4</sup>Siehe 7.2 Expertengespräch ab Minute 39:47

*lieren* wird den Studierenden die Methodik der objektorientierten Programmierung beigebracht. Hierfür wurde bereits in vorherigen Semestern zur Generierung von Datenmodellen in Java die Bibliothek *fulib* verwendet. Doch auch *fulibScenarios*, welches die Datenmodellgenerierung über eine natürliche Sprache übernimmt, wurde als eine mögliche Alternative ausprobiert. Ebenso soll *fulibWorkflows* als eine Alternative zur *Fulib-Notation* ausprobiert werden. Eine weitere Lehrveranstaltung ist *Microservices*, bei welcher Studierende eine Einführung in die Web-Entwicklung und die Verwendung von Microservices erhalten. Da es sich dabei um separate Systeme handelt, welche miteinander kommunizieren, um Daten zu übertragen, ist die Architektur schwierig greifbar. Hierfür soll die Verwendung des mittels *fulibWorkflows* generierten ES-Boards als Unterstützung zum Verständnis der Abläufe dienen.

Studierende haben bereits in vergangenen Instanzen der Veranstaltung *Programmieren und Modellieren* die Web-Anwendung *fulib.org* kennengelernt. Dort ist es möglich auf *fulib* basierende Anwendungen zu verwenden. Neben einer natürlichsprachigen Beschreibung eines Datenmodells besteht ebenfalls die Möglichkeit ein vorgefertigtes Gradle-Projekt zu exportieren. Da auch *fulibWorkflows* ein Teil der *Fujaba Tool Suite* ist, soll der in dieser Arbeit erstellte Web-Editor in *fulib.org* integriert werden. Somit werden Anwendungen, welche zur *Fujaba Tool Suite* gehören, an einem Ort platziert. Da der Web-Editor für *fulibWorkflows* und *fulibScenarios* Objekt- und Klassendiagramme generieren und anzeigen, liegt ein Zusammenlegen nahe. Dies setzt voraus, dass *fulibWorkflows* ebenfalls die Generierung eines Klassenmodells mittels *fulib* in einem Gradle-Projekt bietet. Diese Funktion soll ebenfalls bereitgestellt werden, um den zuvor erwähnten Einsatz in der Lehre zu ermöglichen.

## 7 Quellenverzeichnis

- [Jak13] Irene Lobo Valbuena Jakob Kallin. *Excess XSS*. 2013. URL: <https://excess-xss.com/>.
- [Par13a] Terence Parr. *StringTemplate*. 2013. URL: <https://www.stringtemplate.org/>.
- [Par13b] Terence Parr. *The Definitive ANTLR 4 Reference*. 2013. URL: <https://youtu.be/0AoA3E-cyug?t=69>.
- [Ger15] Terence Parr und Gerald Rosenberg. *StringTemplateGroupGrammar*. 2015. URL: <https://github.com/antlr/grammars-v4/blob/master/stringtemplate/STGParser.g4>.
- [Ver16] Vaughn Vernon. *Domain-Driven Design Distilled*. Addison-Wesley, 2016. ISBN: 978-0-13-443442-1.
- [Bra17] Alberto Brandolini. *Introducing Event Storming, v0.1.0.2*. 2017.
- [Swa17] Swarna. *Angular safe pipe implementation to bypass DomSanitizer stripping out content*. 2017. URL: <https://medium.com/@swarnakishore/angular-safe-pipe-implementation-to-bypass-domsanitizer-stripping-out-content-c1bf0f1cc36b>.
- [Cat20] Beppe Catanese. *1, 2, 3: Docker, Heroku, MongoDB Atlas, Python*. 2020. URL: <https://medium.com/analytics-vidhya/1-2-3-docker-heroku-mongodb-atlas-python-952958423bea#7edf>.
- [Coo21] Scott Cooper. *ngx-codemirror*. 2021. URL: <https://github.com/scottcper/ngx-codemirror>.
- [Gai21] Bertrand Gaillard. *angular-split*. 2021. URL: <https://angular-split.github.io/>.
- [Goo21] Google. *Introduction to the Angular Docs*. 2021. URL: <https://angular.io/docs>.
- [Gop21] Eakan Gopalakrishnan. *Event Storming*. 2021. URL: <https://www.softwarecraftsperson.com/2021/04/25/event-storming/>.
- [Hav21] Marijn Haverbeke. *CodeMirror*. 2021. URL: <https://codemirror.net/>.
- [Kas21a] Fujaba Team Kassel. *fulib - Fujaba library*. 2021. URL: <https://github.com/fujaba/fulib#fulib---fujaba-library>.

- [Kas21b] Fujaba Team Kassel. *fulibTools - Additional features for fulib*. 2021. URL: <https://github.com/fujaba/fulibTools#fulibtools---additional-features-for-fulib>.
- [Kri21] Mads Kristensen. *JSON Schema Store*. 2021. URL: <https://www.schemastore.org/json/>.
- [Pob21] Evgeny Poberezkin. *Ajv JSON schema validator*. 2021. URL: <https://ajv.js.org/>.
- [Tea21a] Bootstrap Team. *Bootstrap Icons*. 2021. URL: <https://icons.getbootstrap.com/>.
- [Tea21b] Bootstrap Team. *Build fast, responsive sites with Bootstrap*. 2021. URL: <https://getbootstrap.com/>.
- [Tea21c] JSON Schema Team. *Getting Started Step-By-Step*. 2021. URL: <https://json-schema.org/learn/getting-started-step-by-step#defining-the-properties>.
- [VMw21] Inc. VMware. *spring initializr*. 2021. URL: <https://start.spring.io/>.
- [Zap21] Alexey Zapparov. *js-yaml*. 2021. URL: <https://github.com/nodeca/js-yaml>.
- [Clo22] CloudRepo. *Public Maven Repositories: Maven Central and More*. 2022. URL: <https://www.cloudrepo.io/articles/public-maven-repositories-maven-central-and-more.html>.
- [Sal22a] Salesforce.com. *Container Registry Runtime (Docker Deploys)*. 2022. URL: <https://devcenter.heroku.com/articles/container-registry-and-runtime>.
- [Sal22b] Salesforce.com. *Language Support*. 2022. URL: <https://devcenter.heroku.com/categories/language-support>.
- [Sal22c] Salesforce.com. *What is Heroku?* 2022. URL: <https://www.heroku.com/what>.
- [VMw22a] Inc. VMware. *Spring Boot*. 2022. URL: <https://spring.io/projects/spring-boot>.
- [VMw22b] Inc. VMware. *Why Spring?* 2022. URL: <https://spring.io/guides/gs/rest-service/>.

# Anhang

## 7.1 Workflow-Beispiel

---

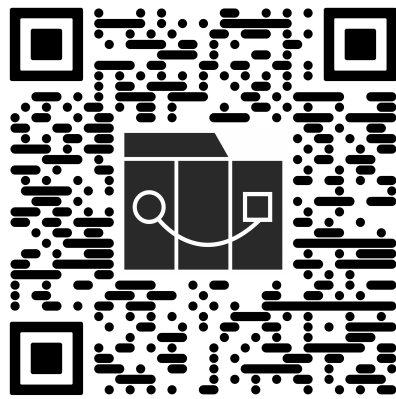
```
1 - workflow: Study Right
2
3 - service: StudyGuide
4
5 - data: Student carli
6   motivation: 83
7   stops: [stop1]
8   stops.back: student
9
10 - data: Stop stop1
11   motivation: 66
12   room: r1
13   room.back: [stops]
14
15 - data: Room r1
16   topic: math
17   credits: 17
18   neighbors: [r2, r5]
19   neighbors.back: [neighbors]
20
21 - data: Room r2
22   topic: calculus
23   credits: 20
24   neighbors: [r1, r5]
25
26 - data: Room r4
27   topic: exam
28   neighbors: [r5]
29
30 - data: Room r5
31   topic: modeling
32   credits: 29
33   neighbors: [r1, r2, r4]
34
35 - command: findRoute
36
37 - data: Stop stop2
38   room: calculus
39   motivation: 56
40   prev: stop1
41   prev.back: [next]
42
43 - data: Student carli
44   route: [math, calculus, math, modeling, exam]
```

---

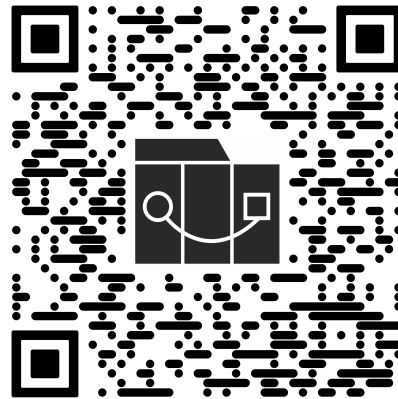
Listing 27: pm.es.yaml

## 7.2 Repositories

TODO: Hier Text hin und noch mal irgendwie die URLs zusätzlich zu den qr-Codes?



((a)) fulibWorkflows



((b)) fulibWorkflows Web-Editor

Abbildung 7.1: QR-Codes der für diese Arbeit erstellten Repositories

### 7.3 Aufnahme des Expertengesprächs



Abbildung 7.2: QR-Code zur Aufnahme des geführten Expertengesprächs

# Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit den nach der Prüfungsordnung der Universität Kassel zulässigen Hilfsmitteln angefertigt habe. Die verwendete Literatur ist im Literaturverzeichnis angegeben. Wörtlich oder sinngemäß übernommene Inhalte habe ich als solche kenntlich gemacht.

Kassel, 22.02.2022

---

Maximilian Freiherr von Künßberg