HAW HAMBURG

Bachelorarbeit

Felix Koch

Entwurf und Implementierung eines WYSIWYG-Web-Editors für SEF-Aufgaben im viaMINT-Projekt

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Felix Koch

Entwurf und Implementierung eines WYSIWYG-Web-Editors für SEF-Aufgaben im viaMINT-Projekt

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 16. Februar 2022

Felix Koch

Thema der Arbeit

Entwurf und Implementierung eines WYSIWYG-Web-Editors für SEF-Aufgaben im viaMINT-Projekt

Stichworte

WYSIWYG, Web-Editors, viaMINT, SEF, Usability, Interface Design, Vue, Antlr4, Graphischer Editor

Kurzzusammenfassung

Die Bachelorarbeit befasst sich mit dem Entwerfen und Implementieren eines "What you see is what you get"-Web-Editors, damit es Personen mit unterschiedlichem technischen Fachwissen ermöglicht wird, Aufgaben zu editieren. Dieser Editor wird in eine vorhandene Web-Architektur des viaMINT-Projektes integriert, welche einen Code-Editor und eine Voransicht besitzt. In dem Editor werden Slider-Exercise-Framework-Aufgaben bearbeitet und der Aufgaben-Code über alle Editoren und Ansichten bidirektional synchronisiert. Die technische Implementierung wird mit Django, Vue, JavaScript und Antlr4 umgesetzt, während die Benutzeroberfläche mit Paper Prototyping gestaltet wird. Für die Gestaltung wurden grundlegende Regeln und Heuristiken für das Interface Design und die Usability genutzt.

Felix Koch

Title of Thesis

Design and implementation of a WYSIWYG web editor for SEF exercises in the viaMINT project

Keywords

WYSIWYG, Web editor, viaMINT, SEF, Usability, Interface Design, Vue, Antlr4, Graphical editor

Abstract

The bachelor thesis deals with the design and implementation of a ” What you see is what you get ” web editor to offer people with different technical knowledge to edit exercises. This editor is integrated into an existing web architecture of the viaMINT project, which has a code editor and a preview. In the editor, Slider Exercise Framework exercises are processed and the exercises code is synchronized bidirectionally across all editors and views. The technical implementation is done with Django, Vue, JavaScript and Antlr4, while the user interface is designed with paper prototyping. Basic rules and heuristics for interface design and usability were used.

Inhaltsverzeichnis

Abbildungsverzeichnis **viii**

Tabellenverzeichnis **ix**

1 Einleitung **1**

1.1 Problemstellung und Motivation 1
1.2 Zielsetzung 2
1.3 Aufbau der Arbeit 2

2 Grundlagen **4**

2.1 Technologien 4
2.1.1 Django 4
2.1.2 Vue2 5
2.1.3 Antlr4 7
2.1.4 JavaScript 8
2.1.5 SEF 8
2.2 Benutzeroberfläche 8
2.2.1 Interface Design 9
2.2.2 Usability 11
2.2.3 Paper Prototyping 13

3 Analyse **15**

3.1 IST-Zustand 15
3.1.1 SEF 15
3.1.2 Editor 18
3.2 Anforderungen 20
3.2.1 Stakeholder 20
3.2.2 Funktionale Anforderungen 21
3.2.3 Nicht-funktionale Anforderungen 23

3.2.4 Anwendungsfälle 24
3.3 Technologietauglichkeit 28

4 Entwurf **30**
4.1 Gestaltungsentwurf 30
4.1.1 Methode 30
4.1.2 Ergebnis 32
4.2 Systementwurf 33
4.2.1 Integration 34
4.2.2 Datenstruktur 34
4.2.3 Visual-Editor-Komponenten 37
4.2.4 Konvertierung der Daten 39
4.3 Teststrategie 40

5 Umsetzung **42**
5.1 SEF-Code Konvertierung 42
5.1.1 Parser 42
5.1.2 Lexer 44
5.2 Datenstruktur 45
5.2.1 NodeType 46
5.2.2 SEF-Code Generierung 47
5.2.3 Boolesche Attribute 47
5.2.4 Mutierte Arrays 47
5.3 Benutzeroberfläche 48
5.4 Offene Probleme 51
5.4.1 Gemischte Texte 51
5.4.2 Paper Prototyping 52
5.4.3 Speichern 53
5.5 Was fehlt 53
5.6 Tests 54
5.7 Dokumentation 55

6 Fazit **56**
6.1 Zusammenfassung 56
6.2 Ausblick 57

Literaturverzeichnis **59**

Glossar **61**

Selbstständigkeitserklärung **62**

Abbildungsverzeichnis

2.1 Aktualisierungsablauf 7

3.1 Datenbank-Modell von Slider und Slide 16
3.2 Beispiel für SEF-Code 17
3.3 Verteilung und Bausteine des IST-Zustands 18

4.1 Gestaltungsentwurf des Visual-Editors 32
4.2 Integrationspunkt für den Visual-Editor 34
4.3 Klassendiagramm der Datenstruktur für den Systementwurf 35
4.4 Komponenten des Visual-Editors im Systementwurf 38

5.1 Namen von Leere-Element 43
5.2 Parser-Grammatik 44
5.3 Regeln der Lexer-Grammatik für Klartext und Kommentare 44
5.4 Verarbeitung von Klartext im Visitor 45
5.5 Verarbeitung eines Kommentars im Visitor 45
5.6 Klassendiagramm der Datenstruktur 46
5.7 Benutzeroberfläche des Visual-Editors 48
5.8 Komponenten des Visual-Editors 49
5.9 Eigenschaften-Komponente 50
5.10 Gemischter Text als SEF-Code 51
5.11 Gemischter Text im Visual-Editor 52

Tabellenverzeichnis

3.1 Element-Struktur im SEF-Code 17

4.1 Daten-Typen des Daten-Baums 35

1 Einleitung

1.1 Problemstellung und Motivation

Meine Erfahrungen im viaMINT Projekt haben mir gezeigt, dass es nicht nur wichtig ist, Material oder Übungen zur Verfügung zu stellen, sondern dass es noch wichtiger ist, diese verständlich aufzubereiten und benutzerfreundlich zu gestalten. Daraufhin habe ich bei meiner alltäglichen Rechnernutzung deutlich auf diese Aspekte geachtet und festgestellt, dass die Benutzerfreundlichkeit doch sehr schwankt. Da dieses Thema kaum im Studium erläutert wurde und es mir als sehr wichtig erscheint, wurde mein Interesse geweckt, einen näheren Einblick zu erhalten.

Es gibt im viaMINT Projekt aktuell eine große Umstrukturierung und Verbesserung der technischen Struktur; dadurch bietet sich die Gelegenheit, mein gewecktes Interesse in meiner Bachelorarbeit umzusetzen.

Ein Teil der Übungsaufgaben auf viaMINT wird mit dem Slider-Exercise-Framework als Programmcode umgesetzt, was sowohl Wissen für die Bedienung einer Programmier-Umgebung als auch über den Programmcode benötigt. Durch die technische Umstrukturierung werden das SEF erneuert und die SEF-Aufgaben in HTML-ähnlichem Format gespeichert, welches in einem Web-Code-Editor bearbeitet werden kann. Um eine sichere und einfache Umgebung für das Erstellen der SEF-Aufgaben zu schaffen und meine Kenntnisse im benutzerfreundlichen Designen zu vertiefen, kam ich auf die Idee, einen grafischen Editor für diese zu erstellen, welcher mit minimalem Wissen über SEF-Aufgaben bedient werden kann.

Zusätzlich bietet sich die Möglichkeit, Software für den produktiven Betrieb zu implementieren. Dies steht im Gegensatz zum Studium, in dem üblicherweise Einweg-Software-Projekte erstellt werden.

1.2 Zielsetzung

Das Ziel der Arbeit besteht in der Entwicklung und Implementation eines visuellen Editors nach dem "What you see is what you get"-Prinzips. Mit diesem soll es möglich sein, Slider-Exercise-Framework-Aufgaben im Kontext von viaMINT ohne technisches Fachwissen erstellen und bearbeiten zu können. Dafür soll die Benutzeroberfläche möglichst einfach und unkompliziert bedienbar sein.

Dieser visuelle Editor soll als möglichst eigenständige Komponente in ein vorhandenes System mit Code-Editor und Voransicht eingegliedert werden und wird als Visual-Editor bezeichnet. Um ein schnelles Wechseln zu ermöglichen, sollen die SEF-Aufgaben dabei bidirektional zwischen dem Visual-Editor, Code-Editor und der Voransicht synchronisiert werden.

Das vorhandene System ist ein in Entwicklung befindlicher Prototyp, welcher alle Anforderungen zum Bearbeiten von SEF-Aufgaben erfüllt. Dieser befindet sich zu Beginn der Arbeit in einer Phase zum Testen, technischen Refactoring und zum Ermitteln von fehlenden oder zusätzlichen Anforderungen. Daher wird der Entwicklungsstand zu Beginn der Arbeit genutzt.

Als Ergebnis für den Visual-Editor wird ein möglichst fortgeschrittener Prototyp angestrebt, wenn möglich mit anpassbaren Weiterentwicklungsmöglichkeiten für das vorhandene System.

1.3 Aufbau der Arbeit

Kapitel 2 beschäftigt sich mit den Grundlagen, welche für das Verständnis der Arbeit benötigt werden. Hierzu gehören Technologien, Frameworks, Programmiersprachen, Methoden und Richtlinien welche in der Arbeit verwendet werden. In Kapitel 3 wird zuerst der bereits vorhandene Editor analysiert, dann werden die Anforderungen an den Visual-Editor erstellt und anschließend die vorgesehenen Technologien getestet. In Kapitel 4 wird der Entwurf für den Visual-Editor beschrieben, welcher auf Basis der Analyse erstellt wurde. Aufgeteilt in die Gestaltung der Benutzeroberfläche, den technische Entwurf und wie dieser in den vorhandenen Editor integriert werden soll. Die Unterschiede, Besonderheiten und ungelösten Probleme von der Umsetzung des Entwurfs werden in

Kapitel 5 erläutert. Kapitel 6 fasst die Arbeit zusammen und gibt einen Ausblick auf die mögliche weitergehende Entwicklung.

2 Grundlagen

In diesem Kapitel werden Grundlagen genannt, welche für das Verständnis der Arbeit erforderlich sind und verwendet werden. Diese teilen sich auf in die verwendeten Technologien und Grundlagen zum erstellen von Benutzeroberflächen.

2.1 Technologien

Es werden Technologien, Frameworks und Programmiersprachen erläutert, welche in dieser Arbeit Verwendung finden. Bei diesen handelt es sich um Django, Vue, Antlr4, Javascript und das SEF.

2.1.1 Django

"Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source."[3]

Die Unterstützung bei der Webentwicklung beinhalten das Beschreiben von Datenbank-Models, den Zugriff auf die Datenbank, eine administrative Benutzeroberfläche, URL Schemata Management, erstellbare Templates für Webseiten, ein Authentifizierungssystem und vieles mehr. Zusätzlich gibt es viele Möglichkeiten Django mit Frameworks und Erweiterungen zu ergänzen.[3]

2.1.2 Vue2

"Vue (pronounced /vju:/, like **view**) is a **progressive framework** for building user interfaces. Unlike other monolithic frameworks, Vue is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications when used in combination with modern tooling and supporting libraries."[15]

In der Arbeit wird Vue mit Vutify[16] ergänzt, welches ist eine Sammlung von Standard-Komponenten und Utility für dessen Nutzung ist. In den folgenden Unterpunkten (Komponenten, Reaktiv und Limitierungen) werden grundlegende Eigenheiten von Vue zusammengefasst, diese Zusammenfassungen basieren auf der Dokumentation von Vue[15].

Komponenten

Vue-Anwendungen sind aus Komponenten zusammengesetzt, welche bei Bedarf wiederverwendet und geschachtelt werden können. Jede Komponente besitzt ein Template- und Skript-Teil. Mit diesen werden ein virtueller DOM deklariert und die Daten gebunden, diese können dann von einem Browser interpretiert und angezeigt werden.

Das Template ist in einer Vue-eigenen HTML-basierten Syntax geschrieben. Diese ermöglicht es, andere Komponenten und HTML zu verwenden. Die Komponenten werden hierbei wie HTML genutzt, wobei statt des HTML-Tags der Komponenten-Name verwendet wird.

Im JavaScript-basierten Skript können Eigenschaften wie Data (reaktive Werte/Objekte), Funktionen, Watcher und andere Logikelemente deklariert oder verändert werden. Diese befinden sich für gewöhnlich je in einer eigenen Instanz und können direkt im Skript oder Template genutzt werden.

Wenn in einem Template einer Komponente eine Kind-Komponente definiert wird, ist es möglich, Attribute wie in HTML zu verwenden. Im Skript von Kind-Komponenten, ist es möglich Props (Properties) zu definieren. Wenn der Name von Attribut und Prop gleich ist, werden diese miteinander verknüpft. Auf diese Weise ist das Übertragen von Werten, Funktionen, Objekten und Arrays von einer Eltern-Komponente zu deren Kind-Komponenten möglich, HTML selbst unterstützt nur einen Einzelwert pro Attribut.

Die Props in Komponenten sind nur für einen lesenden Zugriff vorgesehen, dieses soll einen klaren Datenfluss ermöglichen.

Im Skript einer Kind-Komponente ist es möglich, Events zu definieren, welche es ermöglichen, Werte in der Eltern-Komponente zu verändern. Diese Möglichkeit sollte man nur mit Bedacht verwenden, da der Datenfluss unübersichtlich werden kann.

Auf dieser Basis findet die Übertragung von Werten auf drei verschiedene Arten statt. Selten wird mit 'Attribut="Wert"' der Wert statisch übergeben. In den meisten Fällen wird der Wert mit 'v-bind:Attribut="Wert"' oder ':Attribut="Wert"' reaktiv/dynamisch gebunden/übergeben. Als dritte Option für Inputs wird 'v-model="Wert"' genutzt, hier wird der Wert dynamisch gebunden und zusätzlich in der Kind-Komponente ein Event zum Aktualisieren des Wertes in der Eltern-Komponente erzeugt.

Reaktiv

Vue kann reaktiv auf Datenänderungen reagieren. Wenn ein einfaches JavaScript-Objekt als Data an eine Vue-Komponente übergeben wird, durchläuft Vue alle seine Eigenschaften und konvertiert diese in Getter und Setter. Diese sind für den Nutzer nicht sichtbar, aber ermöglichen es Vue, Abhängigkeitsverfolgung und Änderungsbenachrichtigungen durchzuführen, wenn auf Eigenschaften zugegriffen oder diese verändert werden.

Wie in Abbildung 2.1 zu sehen ist, hat jede Komponenten entsprechende Watcher, welche beim Rendern alle Abhängigkeiten der Komponente als „Touch"-Eigenschaften wahrnehmen. Wenn der Setter einer Abhängigkeit später ausgelöst wird, benachrichtigt er seinen Watcher. Dieses führt wiederum dazu, dass die Komponente erneut gerendert wird.

Limitierungen

Aufgrund von Einschränkungen in JavaScript gibt es Arten von Änderungen, die Vue nicht erkennen kann. Bei **JavaScript-Objekt** kann Vue das Hinzufügen oder Löschen von Eigenschaften nicht erkennen. Da Vue dem Konvertierungsprozess der Getter und Setter während der Initialisierung von Data durchführt, muss eine Eigenschaft vorhanden sein. Bei einem späteren Hinzufügen oder Löschen werden Getter und Setter nicht aktualisiert. Bei **Arrays** kann Vue die Änderungen an einem Array nicht erkennen, wenn ein Element direkt mit dem Index gesetzt wird oder wenn die Länge des Arrays verändert wird.

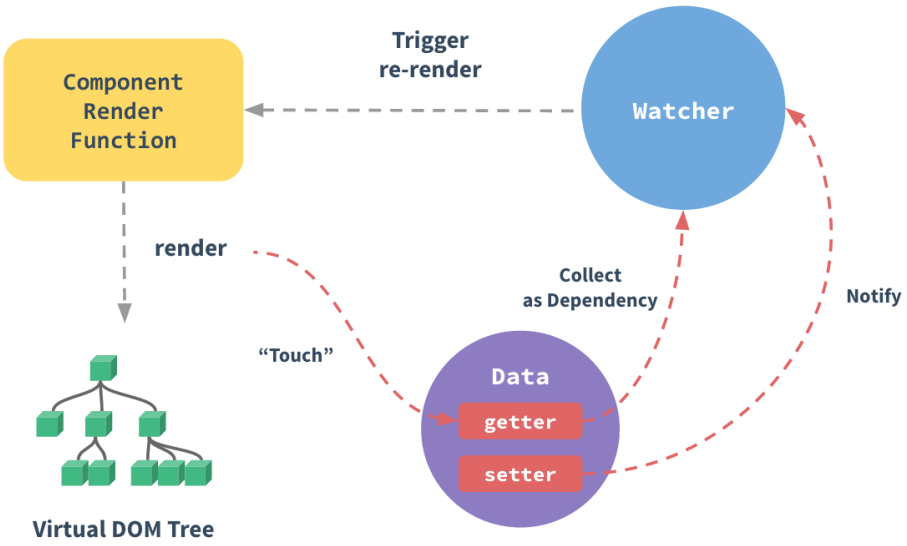

Abbildung 2.1: Aktualisierungsablauf[15]

2.1.3 Antlr4

"**ANTLR** is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build parse trees and also generates a listener interface (or visitor) that makes it easy to respond to the recognition of phrases of interest."[1]

Antlr4 ermöglicht es, einen definierten Parser, Listener und Visitor in verschiedenen Programmiersprachen zu generieren, wie zum Beispiel Python oder JavaScript.[1]

Mit einem Listener oder Visitor ist es möglich, einen durch das Parsen entstandenen Parse-Baum zu durchlaufen und geeignete Aktionen auszuführen, wie das Erstellen einer Datenstruktur oder das Verändern des Parse-Baumes.
In einem Listener wird beim Durchlaufen jedes Knotens eine entsprechende Listener-Methode aufgerufen, welcher das Ausführen von Aktionen ermöglicht. Im Visitor hingegen haben die Visitor-Funktionen einen Rückgabewert und die folgende Visitor-Funktion wird durch die aktuelle aufgerufen. Dieses ermöglicht es zusätzlich, die Reihenfolge zu verändern und Werte weiterzugeben, ohne diese in globalen Variablen zwischen zu speichern.

2.1.4 JavaScript

"**JavaScript** (**JS**) is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions. [...]. JavaScript is a prototype-based, multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles."[7]

Die folgenden Erläuterungen aus diesem Unterkapitel sind aus der Dokumentation von JavaScript[7] zusammengefasst.

Eine prototypenbasierte Programmiersprache erzeugt neue Objekte nicht durch die Instanziierung von Klassen, sondern durch die Kopie eines Objektes. Es ist möglich, diese Objekte während der Laufzeit zu verändern und zu ergänzen.
Wenn im Kontext von JavaScript Klassen verwendet werden, sind damit initial erstellte Objekte gemeint, welche als Grundlage zur Erstellung von Objekten der Klasse dienen. Für Umsetzung von Inhärenz nutzt JavaScript Mixins, was erlaubt, die Eigenschaften und Methoden eines Objektes zu denen eines anderen hinzuzufügen.

Funktionen erster Klasse können als Argument und Rückgabewert verwendet oder einer Variable zugewiesen werden.

2.1.5 SEF

Mit dem Slider-Exercise-Framework werden für das E-Learning auf der Online-Lernplattform viaMINT Aufgaben erstellt und ausgeführt. Das Ausführen der Aufgaben findet im Browser des Nutzers statt, daher sind die Aufgaben in erste Linie für Lern- und Übungszwecke geeignet.

2.2 Benutzeroberfläche

Für das Erstellen einer gut zu bedienenden Benutzeroberfläche benötigt man Qualitätsmerkmale und eine Methode zum Entwerfen. Die Merkmale werden mit den Regeln für Interface Design und Usability Heuristiken beschrieben. Als Methode zum Testen und Entwerfen von Benutzeroberflächen wird das Paper Prototyping vorgestellt.

2.2.1 Interface Design

Interface Design wird von Ben Shneiderman in seinem Buch "Designing the User Interface: Strategies for Effective Human-Computer Interaction"[11] beschrieben. Dort hat er die "8 Golden Rules for Better Interface Design"[10] formuliert.
Diese Regeln sollen bei der Gestaltung der Benutzeroberfläche des Visual-Editors genutzt werden, um eine möglichst hohe Qualität zu erzielen. Für eine Übersicht folgen die Regeln im Original.

IDR1: Strive for consistency.

"Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent color, layout, capitalization, fonts, and so on, should be employed throughout. Exceptions, such as required confirmation of the delete command or no echoing of passwords, should be comprehensible and limited in number"[10]

IDR2: Seek universal usability.

"Recognize the needs of diverse users and design for plasticity, facilitating transformation of content. Novice to expert differences, age ranges, disabilities, international variations, and technological diversity each enrich the spectrum of requirements that guides design. Adding features for novices, such as explanations, and features for experts, such as shortcuts and faster pacing, enriches the interface design and improves perceived quality."[10]

IDR3: Offer informative feedback.

"For every user action, there should be an interface feedback. For frequent and minor actions, the response can be modest, whereas for infrequent and major actions, the response should be more substantial. Visual presentation of the objects of interest provides a convenient environment for showing changes explicitly [...]."[10]

IDR4: Design dialogs to yield closure.

”Sequences of actions should be organized into groups with a beginning, middle, and end. Informative feedback at the completion of a group of actions gives users the satisfaction of accomplishment, a sense of relief, a signal to drop contingency plans from their minds, and an indicator to prepare for the next group of actions. For example, e-commerce websites move users from selecting products to the checkout, ending with a clear confirmation page that completes the transaction.”[10]

IDR5: Prevent errors.

”As much as possible, design the interface so that users cannot make serious errors; for example, gray out menu items that are not appropriate and do not allow alphabetic characters in numeric entry fields [...]. If users make an error, the interface should offer simple, constructive, and specific instructions for recovery. For example, users should not have to retype an entire name-address form if they enter an invalid zip code but rather should be guided to repair only the faulty part. Erroneous actions should leave the interface state unchanged, or the interface should give instructions about restoring the state.”[10]

IDR6: Permit easy reversal of actions.

”As much as possible, actions should be reversible. This feature relieves anxiety, since users know that errors can be undone, and encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data-entry task, or a complete group of actions, such as entry of a name-address block.”[10]

IDR7: Keep users in control.

”Experienced users strongly desire the sense that they are in charge of the interface and that the interface responds to their actions. They don’t want surprises or changes in familiar behavior, and they are annoyed by tedious data-entry sequences, difficulty in obtaining necessary information, and inability to produce their desired result.”[10]

IDR8: Reduce short-term memory load.

"Humans' limited capacity for information processing in short-term memory (the rule of thumb is that people can remember "seven plus or minus two chunks" of information) requires that designers avoid interfaces in which users must remember information from one display and then use that information on another display. It means that cellphones should not require reentry of phone numbers, website locations should remain visible, and lengthy forms should be compacted to fit a single display. "[10]

2.2.2 Usability

Usability beschreibt die intuitive Benutzbarkeit von Anwendungen oder anders ausgedrückt nach "DIN EN ISO 9241-11"[2]: "Die Usability eines Produktes ist das Ausmaß, in dem es von einem bestimmten Benutzer verwendet werden kann, um bestimmte Ziele in einem bestimmten Kontext effektiv, effizient und zufriedenstellend zu erreichen.". Jakob Nielson hat sich in seinem Buch "Usability inspection methods"[9] damit beschäftigt, Benutzeroberflächen zu analysieren, um Probleme in der Usability zu finden. Dabei hat er die "10 Usability Heuristics for User Interface Design"[8] formuliert.

Diese Heuristiken sollen bei der Gestaltung der Benutzeroberfläche des Visual-Editors genutzt werden, um eine möglichst zufriedenstellende Nutzbarkeit zu erhalten. Für eine Übersicht folgen die Heuristiken im Original.

UH1: Visibility of system status

"The design should always keep users informed about what is going on, through appropriate feedback within a reasonable amount of time."[8]

UH2: Match between system and the real world

"The design should speak the users' language. Use words, phrases, and concepts familiar to the user, rather than internal jargon. Follow real-world conventions, making information appear in a natural and logical order."[8]

UH3: User control and freedom

"Users often perform actions by mistake. They need a clearly marked ëmergency exit"to leave the unwanted action without having to go through an extended process."[8]

UH4: Consistency and standards

"Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform and industry conventions."[8]

UH5: Error prevention

"Good error messages are important, but the best designs carefully prevent problems from occurring in the first place. Either eliminate error-prone conditions, or check for them and present users with a confirmation option before they commit to the action."[8]

UH6: Recognition rather than recall

"Minimize the user's memory load by making elements, actions, and options visible. The user should not have to remember information from one part of the interface to another. Information required to use the design (e.g. field labels or menu items) should be visible or easily retrievable when needed."[8]

UH7: Flexibility and efficiency of use

"Shortcuts — hidden from novice users — may speed up the interaction for the expert user such that the design can cater to both inexperienced and experienced users. Allow users to tailor frequent actions."[8]

UH8: Aesthetic and minimalist design

"Interfaces should not contain information which is irrelevant or rarely needed. Every extra unit of information in an interface competes with the relevant units of information and diminishes their relative visibility."[8]

UH9: Help users recognize, diagnose, and recover from errors

"Error messages should be expressed in plain language (no error codes), precisely indicate the problem, and constructively suggest a solution."[8]

UH10: Help and documentation

"It's best if the system doesn't need any additional explanation. However, it may be necessary to provide documentation to help users understand how to complete their tasks."[8]

2.2.3 Paper Prototyping

Paper Prototyping ist eine Methode, um mit minimalem Aufwand die Usability von Benutzeroberflächen zu testen und somit auch zum Entwerfen und Verbessern verwendet werden kann. Da die Methode unkompliziert ist und effizient wirkt, soll unter Anwendung des Paper Prototypings der Entwurf für die Benutzeroberfläche des Visual-Editors erstellt werden.
Als Beispiel soll das Paper Prototyping aus der Sicht eines Nutzers wie in diesem Beispielvideo[17] ablaufen.

Die Autorin des beliebten Grundlagenbuches "Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces"[13], Carolyn Snyder, gibt in einem IBM-Artikel[17] eine Erläuterung der Methode.

Um ein Grundverständnis für die Methode zu geben, folgt eine grobe Zusammenfassung auf der Grundlage des Artikels[12] in den folgenden Unterpunkten (Übersicht[12, S.1, S.6], Anwendbarkeit[12, S.4] und Vorteile[12, S.4-5]).

Übersicht

Um das Paper Prototying anzuwenden, werden Vorbereitungen benötigt. Zuerst erfolgt das Festlegen einer Aufgabe, welche der Nutzer später erreichen soll. Dann wird die Benutzeroberfläche erstellt, mit welcher der Nutzer diese Aufgabe erreichen soll. Diese wird aus Skizzen, Entwürfe oder Screenshots für die einzelnen Komponenten, wie Fenster, Dialogboxen, Popup-Nachrichten etc. erstellt.

Des Weiteren gibt es verschiedene Rollen, die vergeben sein müssen: der bereits oben erwähnte **Nutzer** (user), welcher die Aufgaben ausführt und dessen Verhalten beobachtet wird; ein bis zwei Mitarbeiter als **Computer**, welche die Papierstücke bedienen und damit die Benutzeroberfläche simulierten, wobei zu beachten ist, dass sie keine Erklärungen geben; ein Mitarbeiter als **Moderator** (facilitator), welcher für gewöhnlich Erfahrungen mit dem Thema Usability hat. Dieser führt die Interaktion des Nutzers mit dem Prototyp durch, wie Erläuterungen und das Vorschreiben von Tätigkeiten. Schließlich gibt es einen oder mehrere Mitarbeiter als **Beobachter** (observer), welche Notizen zur Nutzerinteraktion tätigen.

Wenn alle ihre Rollen erfüllen, ist es möglich, noch während der Anwendung die Benutzeroberfläche anzupassen. Dies ermöglicht es, substantielle Änderungen zwischen dem initialen Konzept und der fertigen Version zu tätigen, ohne Code zu erstellen, welcher zum großen Teil wieder gelöscht wird. Es ermöglicht zudem, schon früh im Entwurf signifikante Probleme zu finden.

Anwendbarkeit

Die Anwendung von Paper Prototyping ermöglicht es, verschiedene Arten von Problemen zu identifizieren. Dazu gehören ein Fehlen oder eine missverständliche Gestaltung von Konzepten und Terminologien, Navigation und Workflow, Inhalten, Layout und Funktionalitäten in der Benutzeroberfläche.
Hingegen lassen sich Probleme wie die technische Realisierbarkeit, Lade-/Antwort-Zeiten, Scrolling, Farbgebung und Font lassen sich nicht mit einer abstrakten statischen Darstellung analysieren.

Vorteile

Das Paper Prototyping bietet einige Vorteile für das Entwerfen und Entwickeln. Durch das Testen vor dem Schreiben von Programmcode können Fehler in der Architektur unkompliziert korrigiert oder angepasst werden. Es ist möglich, schnell Änderungen an der Benutzeroberfläche durchzuführen, zum Beispiel während der Methodenausführung. Die Erwartungen der Nutzer an die Benutzeroberfläche werden schnell sichtbar, wodurch es möglich ist, Fehlentwicklungen durch eine andere Erwartung der Entwickler zu vermeiden. Es können während des Testens keine schweren technischen Fehler auftreten, da es keine Technik gibt, wie zum Beispiel Entwicklungsserver, die abstürzen könnten.

3 Analyse

Dieses Kapitel beschäftigt sich mit der Anforderungsanalyse für den Visual-Editor. Hierzu wird zuerst der bereits vorhandene Editor analysiert, dann werden auf Basis der Ergebnisse und des Zieles der Arbeit die Anforderungen an den Visual-Editor formuliert. Anschließend erfolgt eine Ermittlung für die Technologien, welche im Visual-Editor genutzt werden sollen.

3.1 IST-Zustand

Dieser beschreibt den Zustand des Editors für die Bearbeitung von SEF-Aufgaben zu Beginn der Bachelorarbeit. Um Komplikationen durch die noch aktive Entwicklung des Editors zu vermeiden, wird dieser als fester Stand verwendet. Der Editor und das Design der SEF-Aufgaben sind erweiterte Prototypen, welche die dritte Version des SEF nutzen.

Um einen Überblick über die Funktionalität des Editors zu erhalten, werden die Besonderheiten des SEF und alle für die Arbeit relevanten Komponenten des Editors und deren Zusammenspiel beschrieben. Diese Beschreibung ist die Grundlage, um den Visuellen-Editor entwerfen und geeignet einbinden zu können.

3.1.1 SEF

Im folgenden werden die Struktur der SEF-Aufgaben und die Funktionalität des SEF-Codes beschrieben. Diese Beschreibungen werden später dafür genutzt, den Aufgabencode auf geeignete Weise zu verarbeiten.

SEF-Aufgaben

Die SEF-Aufgaben sind in Slidern und Slides organisiert, diese enthalten alle benötigten Daten. Das Datenbankmodell in Abbildung 3.1 zeigt, dass ein Slider mehrere Slides haben kann. Dabei entsprechen die Slider einer Lernsequenz, welche gewöhnlich fünf bis fünfzehn Slides enthält. Jeder Slide enthält den SEF-Code einer SEF-Aufgabe, wobei auch andere HTML-Inhalte möglich sind. Für die Arbeit wird der Begriff "SEF-Aufgabe" für einen Slide verwendet, wenn dieser gültigen SEF-Code enthält.

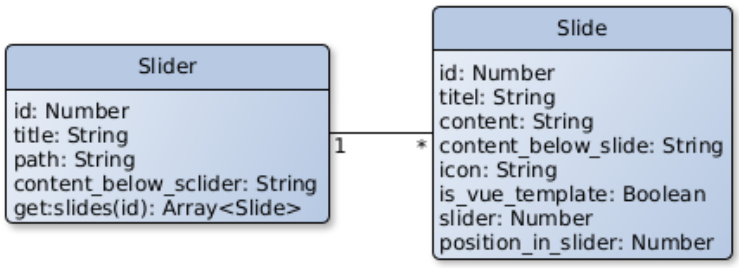

Abbildung 3.1: Datenbank-Modell von Slider und Slide

Im Code des Editors wird die Struktur des Datenbankmodells in Form von Objekten genutzt, wobei der Getter für die zugehörigen Slides im Slider durch ein Array mit den enthaltenen Slides ersetzt ist.

Die relevanten Attribute im Slide sind 'content' für den Inhalt und 'is_vue_template' für die Art des Inhalts. In einer SEF-Aufgabe ist 'is_vue_template' immer wahr. Alle anderen Attribute werden für die Auswahl des Slides vor dem Bearbeiten im Code-Editor und zur Wiedergabe in der Voransicht benötigt.

SEF-Code

Der SEF-Code ist ein Text, welcher in Vue-Template-Syntax geschrieben ist, welche eine erweiterte Form von HTML ist und in den Vue-Komponenten (Abschnitt 2.1.2) genutzt wird.

Die Tabelle 3.1 zeigt, welche Struktur die Elemente im SEF-Code besitzen. Diese leitet sich von HTML ab, welches eine limitierte Menge an Tags zulässt.
In Vue-Templates werden zusätzlich auch die Namen von Komponenten verwendet. Kommentare und Klartext werden wie in regulärem HTML genutzt.

Element Bezeichnung	Struktur
Leere	< Name/Tag Attribute >
Leere und geschlossen	< Name/Tag Attribute />
Block	< Name/Tag Attribute > Elemente </ Name/Tag >
Kommentar	<!-- Text -->
Klartext	Text

Tabelle 3.1: Element-Struktur im SEF-Code

Durch Attribute können die Elemente weiter definiert werden, bei Komponenten ist es so möglich, über die Props initiale Werte zu definieren.

```
<!-- Kommentar -->
<SefContainer renderCheckAnswerButton>
  <MultipleChoiceExercise>
    <MultipleChoiceExerciseAnswer :is_correct='true'>
      <b>Plain Text</b><br>
      <KatexElement expression='4*x'/>
    </MultipleChoiceExerciseAnswer>
  </MultipleChoiceExercise>
  <DragAndDropExercise valid='[[["dest_0", "item_0"], ...], ...],]'>
  </DragAndDropExercise>
</SefContainer>
```

Abbildung 3.2: Beispiel für SEF-Code

Die Abbildung 3.2 zeigt ein kompaktes Beispiel des SEF-Codes. Die HTML-Tags oder Komponenten-Namen sind mit den Steuerzeichen zusammen grün eingefärbt, Attribute blau und deren Werte rot.
Die Namen der Komponenten werden als Konvention in Upper-Camel-Case geschrieben. Benötigte und alternative Werte werden über die Attribute angegeben, welche je nach Komponenten-Definition statisch oder dynamisch/reaktiv sind.

Die verwendeten HTML-Elemente werden hauptsächlich für die Formatierung und Gestaltung von Text verwendet, die Komponenten-Elemente des SEF für die Beschreibung der Auswertung und die verschiedenen Arten von Aufgaben. Bei komplexeren SEF-Aufgaben werden auch Komponenten von Vutify[16] für das Layout genutzt.

3.1.2 Editor

Zum Bearbeiten der Slider gibt es ein komplexes System, dieses beinhaltet ein serverseitiges Backend, das in Django mit Python implementiert ist, und ein im Browser ausgeführtes Frontend, welches in Vue geschrieben ist und Vutify-Komponenten nutzt. Später soll auf Basis der Beschreibung des Systems die Integration des Visual-Editors bestimmt werden.

In der Frontend-Applikation befinden sich die JavaScript (Axios) Rest-API und der Editor als Vue-Komponente. Der Editor besteht aus einer Vielzahl von Komponenten die ineinander verschachtelt genutzt werden. Mit diesen ist es möglich, Slider und Slides zu laden, zu speichern und zu bearbeiten. Die Abbildung 3.3 zeigt eine Übersicht über die Komponenten, welche benötigt werden um SEF-Code zu bearbeiten.

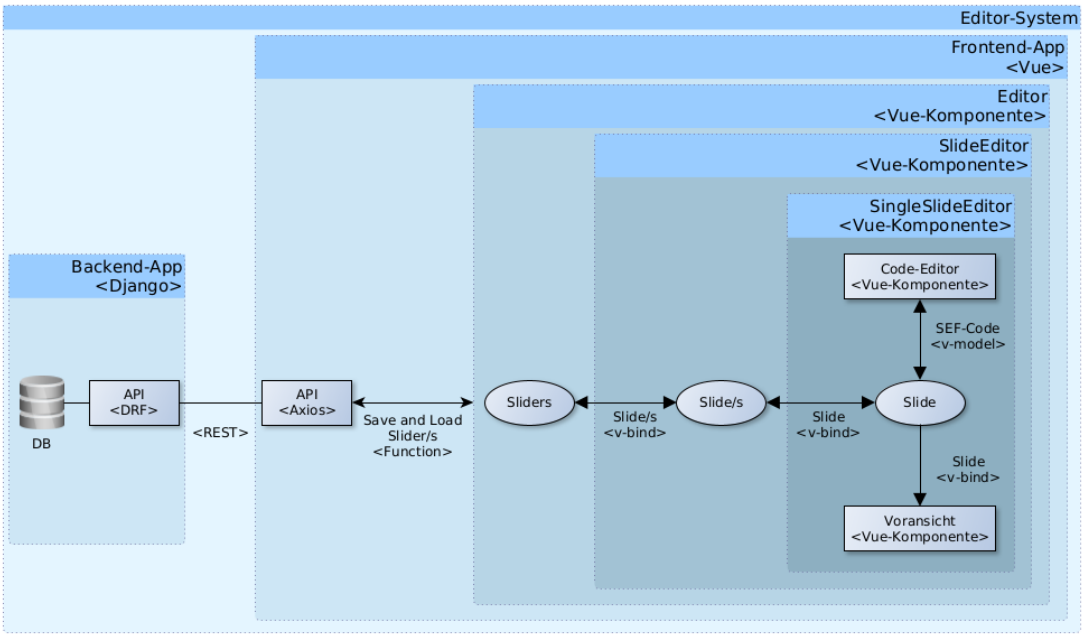

Abbildung 3.3: Verteilung und Bausteine des IST-Zustands

Bearbeitungsablauf

Über nicht dargestellte Vue-Komponenten des Editors können Slider und danach die enthaltenen Slides ausgewählt und somit geladen werden. Dafür wird im Editor eine Funktion aufgerufen, welche über eine REST-Schnittstelle die Slider lädt. Das Backend lädt die

Slider direkt aus der Datenbank über das verwendete Django-Rest-Framework[4]. Nun werden die Slider im Editor gespeichert und über dynamische Attribute die enthaltenen Slides an den SlideEditor weitergereicht, welcher wiederum einen einzelnen ausgewählten Slide an den SingleSlideEditor gibt, der diesen Slide an die Voransicht und den SEF-Code an den Code-Editor weiterreicht.

Hierbei nutzt der Code-Editor für die dynamische Bindung v-model, wie es auch Input-Felder verwenden. Dieses ermöglicht es dem Code-Editor, Änderungen am SEF-Code direkt wieder zurückzugeben, daher ist der SEF-Code im Slide immer aktuell.
Für die dynamische Bindung zwischen den anderen Komponenten wird v-bind verwendet. Da es sich beim Slider und Slide um Objekte handelt, registrieren die Komponenten Änderungen der Referenzen in ihren Props. Bei einer Veränderung am SEF-Code durch den Code-Editor wird dieser überall aktualisiert. Es wird allerdings nur dort reaktiv auf die Änderung reagiert, wo der Slide in Data gespeichert wurde oder es einen Watcher auf das SEF-Code Attribut des Slides gibt.

Dieses Verhalten ermöglicht einen Datenfluss von Kind-Komponente zu Eltern-Komponente ohne die Nutzung von Events, was nicht von Vue empfohlen wird. Da nur an einer Stelle das verändern des SEF-Codes stattfindet, ist der einzige Nachteil die unerwartete Implementation.

Zum Speichern wird über einen Knopf oder eine Tastenkombination im Editor eine Funktion aufgerufen, welche über eine REST-Schnittstelle den selektierten Slider speichert.

Code-Editor

Diese Ansicht ist eine Vue-Komponente mit zwei Unterkomponenten. Eine ist ein importierter Code-Editor mit dem Namen Vue-Codemirror[14], in dem der SEF-Code bearbeitet wird. Es ist möglich, den Code-Editor auf vielfältige Art einzustellen. Unter anderem kann er den Code farblich angepasst anzeigen, wie es in Abbildung 3.2 zu sehen ist. Und die andere Unterkomponente ist ein Auswahlmenü, über das man vorgefertigte Code-Segmente in den zuvor markierten Bereich des Code-Editors einfügen kann. Das Nutzen der Code-Ansicht zum Bearbeiten von SEF-Aufgaben erfordert technisches Wissen.

Voransicht

Hier handelt es sich um eine eigene Vue-Komponente, welche einen Slider mit seinen Slides (Abbildung 3.1) anzeigt. Diese Komponente ist auch für das Darstellen der Slider im produktiven Einsatz gedacht. Hier wird die Komponente verwendet, um den ausgewählten Slide als Voransicht anzuzeigen.

Für die Anzeige des 'content' wird eine von zwei Subkomponenten verwendet, ausgewählt durch den Boolean 'is_vue_template'. Ist dieses 'false', dann wird der Inhalt direkt in das Template der Komponente eingefügt. Damit ist es möglich, enthaltenes HTML anzeigen zu lassen, der Template-Code von Vue-Komponenten wird dabei ignoriert.
Ist der Bolean 'true', dann ist der Inhalt SEF-Code und wird über eine dynamische Komponente zur Laufzeit umgewandelt. Damit ist es möglich, enthaltenes HTML und die in der dynamische Komponente definierten Vue-Komponenten anzeigen zu lassen, hier die Komponenten des SEF.

3.2 Anforderungen

Für die Entwicklung des Visual-Editors zum Bearbeiten von SEF-Aufgaben müssen zuerst die Anforderungen an diesen bestimmt werden. Diese beschreiben die Zielsetzung der Arbeit und ermöglichen eine Beurteilung.

Für die Ermittlung der Anforderungen werden die Stakeholder identifiziert. Mit Hilfe dieser und der Ziele der Arbeit (Abschnitt 1.2) werden anschließend die funktionalen und nicht-funktionalen Anforderungen definiert. Um das Erreichen der Anforderungen und der Funktionalität des Visual-Editors prüfen zu können, werden anschließend die grundlegenden Anwendungsfälle definiert.

3.2.1 Stakeholder

Die Stakeholder oder auch Akteure sind Personen welche ein Interesse an der Bearbeitung und Entwicklung von SEF-Aufgaben haben. Diese lassen sich in fachfremde und technische Nutzer aufteilen.

Fachfremder Nutzer

Ein fachfremder Nutzer ist in der Regel mit dem Programmieren von Code nicht vertraut und möchte ohne Fachkenntnisse in diesem Bereich SEF-Aufgaben bearbeiten und entwerfen. Hierzu zählen zum Beispiel Lehrkräfte, welche ihre Aufgaben und Ideen selbst direkt implementieren wollen.

Wissen: Ein solcher Nutzer besitzt keine oder nur wenige Informatikkenntnisse und wurde in das Entwerfen von SEF-Aufgaben eingewiesen.
Ziel: Er möchte unkompliziert SEF-Aufgaben erstellen ohne seine Kenntnisse in der Informatik zu vertiefen.
Beispiel: Tutor, Lehrkraft, Berater des viaMINT-Teams für das Erstellen von Online-Aufgaben.

Technischer Nutzer

Ein technische Nutzer besitzt diese Kenntnisse und Detailwissen, welches es ermöglicht SEF-Aufgaben in SEF-Code zu schreiben und Spezialfälle umzusetzen. Für diesen ist das Implementieren eher trivial, aber wegen der benötigten und zeitaufwendigen Kommunikation wird er von anderen Arbeitsaufgaben abgehalten, die technisches Fachwissen benötigen.

Wissen: Er ist mit den SEF-Aufgaben gut vertraut, kann diese in SEF-Code umsetzen und besitzt Detailwissen in diesem Bereich.
Ziel: Die Wartung und Erweiterung von IT-Infrastruktur und möglichst wenig Zeit für das Umsetzen von SEF-Aufgaben zu verwenden.
Beispiel: Die technischer Mitarbeiter des viaMINT Teams, Informatiker, Tutor mit Einweisung im SEF-Code.

3.2.2 Funktionale Anforderungen

Die funktionalen Anforderungen an den Visual-Editor beschreiben die gewünschten Fähigkeiten sowie dessen Verhalten. Im Folgenden sind diese beschrieben.

FA1: Bidirektional

In dem System des Editors muss der SEF-Code einer ausgewählten SEF-Aufgabe bidirektional zwischen vorhandenen Editoren und Ansichten synchronisiert sein. Hierzu zählen der Visual-Editor, der Code-Editor und die Voransicht.

FA2: Auswählen

In dem Visual-Editor muss es einem Nutzer möglich sein, über die Benutzeroberfläche einzelne Komponenten auszuwählen, welche im SEF-Code der ausgewählten SEF-Aufgabe beschrieben werden.

FA3: Editieren

In dem Visual-Editor muss es einem Nutzer möglich sein, über die Benutzeroberfläche ausgewählte Attribute von Komponenten zu editieren, welche im SEF-Code der SEF-Aufgaben beschrieben werden. Mit den auswählbaren Attributen muss es möglich sein, funktionierende SEF-Aufgaben zu erstellen.

FA4: Operationen

In dem Visual-Editor muss es einem Nutzer möglich sein, über die Benutzeroberfläche Komponenten mit Operationen zu bearbeiten. Das Minimum an Operationen ist: hinzuzufügen, löschen und versetzen. Die Komponenten werden im SEF-Code der SEF-Aufgaben beschrieben.

FA5: Integration

Der Visual-Editor soll als möglichst eigenständige Komponente oder als Plugin in die Struktur des vorhandene Editors integriert werden und dabei die vorhandenen Funktionalitäten nutzen.

FA6: Code umwandeln

Das System des Visual-Editors muss den vorhandenen SEF-Code der ausgewählten SEF-Aufgabe in eine Form umwandeln können, mit der die beschriebenen Komponenten in einer Benutzeroberfläche für die Bearbeitung angezeigt werden. Bei einer Bearbeitung müssen entsprechende Änderungen im SEF-Code vorgenommen werden, wie es in WYSIWYG-Editoren üblich ist.

FA7: Technik

Der Visual-Editor soll mindestens mit dem Browser Firefox auf Desktop-Rechnern fehlerfrei angezeigt werden. Hierbei soll eine Standartauflösung wie Full HD angestrebt werden.

3.2.3 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen an den Visual-Editor beschreiben die Qualitäts-Merkmale an die geforderten Funktionalitäten. Im Folgenden sind diese beschrieben.

NFA1: Usability

Die Benutzeroberfläche Visual-Editor soll eine möglichst gute Usability haben. Als Grundlage für die Umsetzung sollen die Regeln für die Usability nach Jakob Nielson aus Unterabschnitt 2.2.2 angewandt werden.

NFA2: Interface Design

Die Benutzeroberfläche des Visual-Editor soll ein möglichst gutes Interface Design haben. Als Grundlage für die Umsetzung sollen die Regeln für das Interface Design nach Ben Shneiderman aus Unterabschnitt 2.2.1 angewandt werden.

NFA3: Modularität

Die einzelnen Teile der Architektur des Visual-Editors sollen in möglichst unabhängige Module implementiert werden, um spätere Ergänzungen zu ermöglichen.

NFA4: Reaktionszeit

Der Visual-Editor soll in einem für den Nutzer angenehmen Zeitrahmen reagieren und Aktionen ausführen.

3.2.4 Anwendungsfälle

Alle Anwendungsfälle basieren auf den funktionalen Anforderungen und bilden die grundlegenden Funktionalitäten für den Betrieb des Visual-Editors ab. Mit diesen ist es möglich, die Funktionalität von Anforderungen zu prüfen.

Über die nicht-funktionalen und funktionalen Anforderungen ist beschrieben, dass der Visual-Editor Komponenten besitzt. Diese sind eine visuelle Darstellung der im SEF-Code beschriebenen Komponenten einer SEF-Aufgabe und dienen der Bearbeitung dieser. Für die Anwendungsfälle werden beide als Komponente bezeichnet. Es folgend die Anwendungsfälle.

AF1: Visual-Editor erzeugen

Generierung der Komponenten für die Benutzeroberfläche des Visual-Editors auf Grundlage des SEF-Codes der ausgewählten SEF-Aufgabe, basierend auf FA6: Code umwandeln.
Akteure: Fachfremder Nutzer oder Technischer Nutzer.
Auslöser:
Ein Nutzer möchte eine SEF-Aufgabe im Visual-Editor öffnen.
Vorbedingung:
Ein im Editor ausgewählte SEF-Aufgabe die mit SEF-Code befüllt ist und der Visual-Editor ist nicht aktiv.
Nachbedingung:
Alle Komponenten werden in der Benutzeroberfläche des Visual-Editor angezeigt.

Erfolgsszenario:
1. Der Nutzer aktiviert den Visual-Editor.
2. Das System generiert aus dem SEF-Code Komponenten.
3. Das System zeigt die Komponente/n in der Benutzeroberfläche des Visual-Editors an.
Erweiterungen:
2.a Für syntaktisch korrekte, aber dem Visual-Editor unbekannte Komponenten werden geeignete Komponenten erzeugt.
Fehlerfälle:
2.a Bei ungültigem SEF-Code bricht das System ab und erstellt eine Komponente, welche den Nutzer geeignet informiert und den kompletten SEF-Code enthält.

AF2: SEF-Code generieren

Generierung von SEF-Code aus den Komponenten des Visual-Editors.
Akteure: Fachfremder Nutzer oder Technischer Nutzer.
Auslöser:
Ein Nutzer möchte den Visual-Editor schließen oder verändert die aktuell geöffnete SEF-Aufgabe über den Visual-Editor.
Vorbedingung:
Ein mit Komponente befüllter Visual-Editor, der aktiv ist.
Nachbedingung:
Der aktuelle SEF-Code wird im Editor gespeichert und liegt dessen Editoren und Ansichten vor.
Erfolgsszenario:
1. Der Nutzer deaktiviert den Visual-Editor oder die Komponenten werden verändert. (Siehe AF4 bis AF7)
2. Das System generiert aus den Komponenten SEF-Code.
3. Das System speichert den SEF-Code im Editor.

AF3: Komponente auswählen

Auswählen einer Komponente über die Benutzeroberfläche des Visual-Editors, basierend auf FA2: Auswählen.
Akteure: Fachfremder Nutzer oder Technischer Nutzer.
Auslöser:

Ein Nutzer möchte eine andere Komponente im Visual-Editor auswählen, um die Eigenschaften zu sehen oder diese bearbeiten zu können.
Vorbedingung:
Ein mit Komponenten befüllter Visual-Editor, der aktiv ist.
Nachbedingung:
Die ausgewählte Komponente wird angezeigt und ist visuell markiert und ihre Eigenschaft/en und ausführbaren Operationen werden in der Benutzeroberfläche angezeigt.
Erfolgsszenario:
1. Der Nutzer wählt eine vorhandene Komponente aus, welche zuvor nicht ausgewählt war.
2. Das System markiert nur die ausgewählte Komponente visuell.
3. Das System zeigt die Eigenschaft/en der Komponente an.
4. Das System zeigt geeignete Operationen an, welche mit der Komponente ausgeführt werden können.

AF4: Operation Komponente hinzufügen

Erstellung einer Komponente über die Benutzeroberfläche des Visual-Editors, basierend auf FA4: Operationen und FA6: Code umwandeln.
Akteure: Fachfremder Nutzer oder Technischer Nutzer.
Auslöser:
Ein Nutzer möchte neue Komponenten über den Visual-Editor hinzufügen.
Vorbedingung:
Ein leerer oder mit Komponenten befüllter Visual-Editor, der aktiv ist.
Nachbedingung:
Die neue Komponente ist zu sehen und auswählbar.
Erfolgsszenario:
1. Der Nutzer wählt eine Komponente aus, in der es möglich ist, neue Komponenten hinzuzufügen.
2. Der Nutzer wählt eine Art von Komponente aus, welche erzeugt werden soll.
3. Der Nutzer wählt die Positionierung der neuen Komponente aus.
4. Der Nutzer betätigt den Knopf zum Hinzufügen der Komponente.
5. Das System erstellt eine Komponente mit der ausgewählten Art und Position.
6. Das System aktualisiert den SEF-Code und die Benutzeroberfläche.

AF5: Operation Komponente verschieben

Verschieben eine Komponente über die Benutzeroberfläche des Visual-Editors, basierend auf FA4: Operationen und FA6: Code umwandeln.
Akteure: Fachfremder Nutzer oder Technischer Nutzer.
Auslöser:
Ein Nutzer möchte eine Komponente über den Visual-Editor verschieben.
Vorbedingung:
Ein mit mindestens zwei Komponenten befüllter Visual-Editor, der aktiv ist.
Nachbedingung:
Die ausgewählte Komponente ist an der neuen Position zu sehen und auswählbar.
Erfolgsszenario:
1. Der Nutzer wählt eine vorhandene Komponente aus, welche verschoben werden kann.
2. Der Nutzer wählt eine andere Komponente als Ziel aus.
3. Der Nutzer wählt eine neue Positionierung aus.
4. Der Nutzer betätigt den Knopf zum Verschieben der Komponente.
5. Das System verschiebt die Komponente an die neue Position.
6. Das System aktualisiert den SEF-Code und die Benutzeroberfläche.
Erweiterungen:
3.a Die Position ist vor oder hinter dem Ziel.
3.b Die Position kann in dem Ziel sein, insofern es Komponenten enthalten kann.

AF6: Operation Komponente löschen

Löschen einer Komponente über die Benutzeroberfläche des Visual-Editors, basierend auf FA4: Operationen und FA6: Code umwandeln.
Akteure: Fachfremder Nutzer oder Technischer Nutzer.
Auslöser:
Ein Nutzer möchte eine Komponenten über den Visual-Editor löschen.
Vorbedingung:
Ein mit Komponenten befüllter Visual-Editor, der aktiv ist.
Nachbedingung:
Die ausgewählte Komponente ist gelöscht und nicht mehr zu sehen oder auswählbar.
Erfolgsszenario:
1. Der Nutzer wählt eine Aufgaben-Komponente aus, die gelöscht werden kann.

2. Der Nutzer betätigt den Knopf zum Löschen der Komponente.
3. Das System löscht die ausgewählte Komponente.
4. Das System aktualisiert den SEF-Code und die Benutzeroberfläche.

AF7: Komponente editieren

Editieren einer Eigenschaft einer Komponente über die Benutzeroberfläche des Visual-Editors, basierend auf FA3: Editieren und FA6: Code umwandeln.
Akteure: Fachfremder Nutzer oder Technischer Nutzer.
Auslöser:
Ein Nutzer möchte die Eigenschaften einer Komponente über den Visual-Editor editieren.
Vorbedingung:
Ein mit Komponenten befüllter Visual-Editor, der aktiv ist.
Nachbedingung:
Die editierte Eigenschaft ist in der ausgewählten Komponente sichtbar und aktualisiert.
Erfolgsszenario:
1. Der Nutzer wählt eine vorhandene Komponente aus, welche änderbare Eigenschaften besitzt.
2. Der Nutzer ändert eine Eigenschaft der ausgewählten Komponente über eine geeignete und angezeigte Eingabemöglichkeit.
3. Das System editiert die Eigenschaft in der Komponente.
4. Das System aktualisiert den SEF-Code und die Benutzeroberfläche.

3.3 Technologietauglichkeit

Um festzustellen welche, Technologien und Programmiersprachen für eine Umsetzung des Visual-Editors am besten geeignet sind, wird eine Auswahl getroffen und diese getestet. Dabei ist eine möglichst gute Kompatibilität mit dem Editor (3.1.2) von Vorteil und es müssen die Anforderungen (3.2) umsetzbar sein.

Technologieauswahl

Da es bereits möglich ist, die SEF-Aufgaben über das Frontend zu laden und zu speichern, ist es voraussichtlich nicht notwendig, neue Strukturen im Backend zu erstellen.

Falls dies doch der Fall ist, sollte die genutzte Technologie auf Django/Python (2.1.1) basieren, da das Backend diese nutzt.
Das Frontend verwendet Vue/Vuetify (2.1.2), welche auf JavaScript (2.1.4) basieren. Zusätzlich sind die vorhandenen Hilfsstrukturen in JavaScript geschrieben, daher sollten die zu verwendenden Technologien auch auf darauf basieren.

Für die Umsetzung des Visual-Editors sind einige Strukturen nötig, die im folgenden mit einer kurzen Erläuterung und der angedachten Technologie beschrieben werden. Für eine Einbettung in den Editor bietet sich eine Vue-Komponente an, diese kann an fast jeder Stelle ohne weitere Anpassungen verwendet werden und ist in sich geschlossen. Zusätzlich ist es möglich, die Benutzeroberfläche der SEF-Aufgabe und Operationen für die Bearbeitung als Subkomponenten zu erstellen. Damit diese die Komponenten den SEF-Code bearbeiten können, ist es nötig, diesen von seiner Textform in eine Objektstruktur zu überführen, dies könnte sowohl im Frontend als auch im Backend geschehen. Für die Objektstruktur bieten sich Python und JavaScript an, da diese konsistent mit der vorhandenen Codebasis des Editors sind. Der vermutlich komplexeste Teil ist das Umwandeln des SEF-Codes in die Objektstruktur. Hierbei betet sich der Parsergenerator Antlr4 (2.1.3) an, da er es ermöglicht, einen definierten Parser sowohl in Python als auch in JavaScript zu transpilieren.

Technologietest

Um zu testen, ob die ausgewählten Technologien und Programmiersprachen geeignet sind, wurde ein Technologietest durchgeführt. Das Ziel war es, Teile eines simplen SEF-Codes in der Benutzeroberfläche anzuzeigen, zu ändern und anschließend zu speichern. Dabei ging es lediglich um die Machbarkeit und das Sammeln von Erfahrungen, eine Architektur für den späteren Entwurf lag nicht im Fokus.

Die Umsetzung des Tests fand komplett im Frontend mit JavaScript, Vue und Vutify möglich. Als Besonderheit zeigte sich, dass Antlr4 komplex und für JavaScript nur mäßig dokumentiert ist. Allerdings bietet es die Möglichkeit, einen Visitor zu generieren, welcher sich für eine Grundstruktur anbietet, um das Ergebnis des Parsers in eine Datenstruktur zu überführen.

4 Entwurf

Der Entwurf für den Visual-Editor besteht aus Gestaltungsentwurf, Systementwurf und Teststrategie. Die Entwürfe zur Gestaltung der Benutzeroberfläche und des dahinter liegenden Systems basieren auf den zuvor ermittelten Anforderungen. Die Strategie für das Testen ermöglicht es, nach einer Umsetzung des Entwurfs zu prüfen, ob die Anforderungen erfolgreich umgesetzt wurden.

4.1 Gestaltungsentwurf

Dieses Kapitel widmet sich dem Entwurf für die Gestaltung der Benutzeroberfläche des Visual-Editors. Der erste Teil bezieht sich auf die verwendete Methode Paper Prototyping (2.2.3) und beschreibt, wie diese an die Gegebenheiten angepasst und angewendet wurde. Der zweite Teil erläutert das Ergebnis und die getroffenen Entscheidungen.

4.1.1 Methode

Die Methode, mit der die Benutzeroberfläche gestaltet wird, basiert auf dem Paper Prototyping (2.2.3), das auf Grund der Gegebenheiten angepasst werden musste.

Anpassungen

Da ich die Bachelorarbeit als Einzelperson ausgeführt habe, war es nötig, die Rollen für den Moderator, Computer und Beobachter selbst auszuführen. Durch die Corona-Pandemie war es zu riskant, Menschen zu treffen und einen Papier Prototype zu nutzen, daher erfolgte das Darstellen der Benutzeroberfläche mit dem kollaborativen online-Whiteboard Miro[6]. In diesem ist es möglich, die einzelnen Komponenten der Benutze-

roberfläche aus Linien, Vierecken, Freitext und Ähnlichem zu erstellen, zu bewegen, zu modifizieren und zu duplizieren.

Vorbereitung

Als Vorbereitung gab es einen groben Vorentwurf für die Benutzeroberfläche des Visual-Editors und einige typische Tätigkeiten als Aufgaben für den Nutzer. Die einzelnen Benutzeroberflächen-Komponenten des Vorentwurfs wurden in einer weniger detaillierten Form in Miro[6] umgesetzt, da viele Änderungen während der Anwendung erwartet wurden. Bei den Tätigkeiten handelte es sich um das Erstellen von verschiedenen Aufgaben und das Verändern dieser, dabei wurden alle Operationen, wie das Auswählen und Editieren aus den Anforderungen (3.2) beachtet. Die Nutzer war durch Personen aus dem Kreis der fachfremden Nutzer repräsentiert, da das Bedienen der Benutzeroberfläche für diese vorgesehen ist. Technische Nutzer wurden nicht befragt, da für diese bereits der Code-Editor vorhanden ist. Die Teilnahme der Nutzer war freiwillig und unentgeltlich.

Ausführung und Auswirkungen

Die Änderungen an der Methode hatten die folgenden Auswirkungen: Während des Ausführens der Computer-Aktionen, wurde eine offene Diskussion in einem Dialog geführt. In diesem konnte der Nutzer seine Gedankengänge zu seinen Aktionen erläutern und seine Erwartungen an die Reaktion des Visual-Editors beschreiben.
Während aufwendigerer Änderungen an der Benutzeroberfläche, wie dem Vorbereiten der nächsten Aufgabe, wurden die Komponenten teils auf Basis der Nutzervorschläge angepasst und in einem parallelen Dialog Fragen auf Basis der Regeln vom Interface Design und der Usability gestellt, um weitere Änderungsvorschläge durch den Nutzer anzuregen. Nachdem es keine oder nur noch triviale Änderungsvorschläge gegeben hatte, wurde die Sitzung beendet.

Einschätzung

Das Ausführen des Paper Prototyping in veränderter Form hat sich als praktikabel und dynamisch für das Entwerfen der Benutzeroberfläche herausgestellt. Sowohl an der Art der Änderungsvorschläge als auch an den Dialogen war es gut nachvollziehbar, wann ein Ergebnis vorlag.

Durch das Besetzen aller Mitarbeiterrollen von einer Person war die Ausführung langsamer, was ausführliche und intensive Gespräche mit dem Nutzer ermöglichte. Daher ist diese Art der Ausführung allerdings mehr für das Gestalten als das Testen geeignet. Für das Testen werden spontane und impulsive Reaktionen benötigt, welche nur mit einer schnellen Änderung durch den Computer erreicht werden und durch fokussierte Beobachter besser einschätzt werden können.
Durch die Verwendung von Miro[6] musste der Nutzer seine Aktionen beschreiben und konnte sie nicht zeigen, daher war dieser Aspekt weniger intuitiv.

4.1.2 Ergebnis

Aus der Anwendung der veränderten Methode folgte Abbildung 4.1 als Ergebnis, dieser Gestaltungsentwurf soll für die spätere Umsetzung des Visual-Editors verwendet werden.

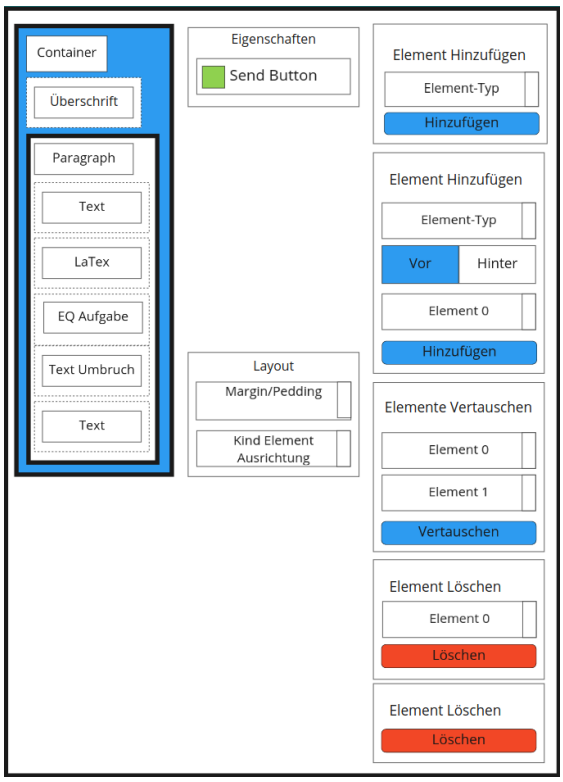

Abbildung 4.1: Gestaltungsentwurf des Visual-Editors

Die Benutzeroberfläche des Gestaltungsentwurfs ist in drei Teile gegliedert: die Auswahl von Komponenten, die Bearbeitung von Eigenschaften und das Einstellen/Ausführen von Operationen. Auf der linken Seite befindet sich der Auswahlbaum, dieser ermöglicht es, einzelne Komponenten auszuwählen und somit durch die Aufgabe zu navigieren. Die ausgewählte Komponente wird für die visuelle Unterscheidung farblich markiert. Auf der rechten Seite befinden sich Operationen, welche mit der ausgewählten Komponente verwendet werden können. Diese sind wiederum aufgeteilt in Operationen mit Bezug auf die Kind-Komponenten (Hinzufügen, Vertauschen und Löschen) und die ausgewählte Komponente selbst (Verschieben und Löschen). In der Mitte werden die Eigenschaften, Inhalte und das Layout der ausgewählten Komponente angezeigt, sodass es möglich ist, diese zu bearbeiten. Dabei sind die individuellen Eigenschaften und Inhalte oben und das Layout unten angeordnet.

Das Trennen der einzelnen Aktionsbereiche erzeugt eine logische Gliederung. Dabei fällt der Blick zuerst auf den Auswahlbaum, welcher das Kernelement bildet. Da fertige Aufgaben oftmals nur angepasst werden, befindet sich die Bearbeitung der Eigenschaften mittig. Die Operationen sind abseits auf der rechten Seite angeordnet, da sie meist nur zum Erstellen von Aufgabe benötigt werden.

Die Gruppierung von Komponenten zu Aktionsbereichen ermöglicht es später, diese ohne größere Probleme neu zu platzieren oder sie zum Beispiel einzeln auf einem Handydisplay anzuzeigen. Von der farblichen Gestaltung her sind die nicht destruktive Auswahloptionen und Aktionen blau und die destruktiven rot gestaltet. Dies ermöglicht eine schnelle Unterscheidung und entspricht auch anderen Designs, was die Bedienung intuitiver gestaltet. Um die einzelnen Elemente gut voneinander unterscheiden zu können, haben alle Grundelemente einen Abstand zueinander und eine Schattierung. Durch die Begrenzung der Aktionsmöglichkeiten in der Benutzeroberfläche soll es möglich sein, Fehleingaben des Nutzers zu unterbinden, und somit das Auftreten von Fehlern weitestgehend umgangen werden.

4.2 Systementwurf

Diese beschreibt das System des Visual-Editors und wie dieses gestaltet ist und in den vorhandenen Editor integriert werden soll. Das System wird als eine eigenständige Vue-Komponente mit den bereits bekannten Technologien (3.3) und Anforderungen (3.2) entworfen, dabei sollen die Regeln für das Interface Design (2.2.1) angewendet werden.

4.2.1 Integration

Der beste Ort für die Integration der Vue-Komponente für den Visual-Editor ist neben dem Code-Editor und der Voransicht im SingleSlideEditor (3.3, 4.2). Dieser ist nur sichtbar, wenn exakt ein Slide ausgewählt ist, welcher dann bearbeitet werden kann. Änderungen in anderen Komponenten oder im Backend werden nicht benötigt, da der Visual-Editor eine eigenständige Komponente ist.

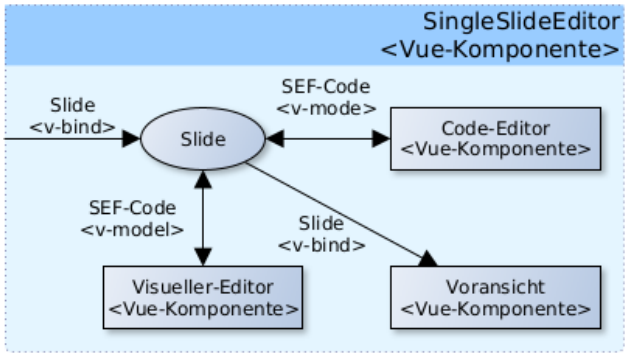

Abbildung 4.2: Integrationspunkt für den Visual-Editor

Der für die Bearbeitung benötigte SEF-Code der ausgewählten SEF-Aufgaben kann wie beim Code-Editor als reaktives (v-model) Attribut an den Visual-Editor übergeben werden, wie in Abbildung 4.2 dargestellt.
Durch eine solche Integration bleibt das im Bearbeitungsablauf (3.1.2) beschriebene Speicherverhalten des SEF-Codes unverändert und ist durch den reaktiven Zugriff aller beteiligten Komponenten in jeder Komponente immer aktuell, wodurch der SEF-Code Bidirektional zur Verfügung steht.

4.2.2 Datenstruktur

Der Visual-Editor benötigt eine Datenstruktur, welche es ermöglicht den SEF-Code sinnvoll in Vue-Komponenten zu nutzen. Dafür muss die Datenstruktur es ermöglichen den SEF-Code zu importieren und nach einer Bearbeitung wieder als SEF-Code zu exportieren, ohne dass dabei Informationen verloren gehen.
Dafür ist die Datenstruktur in einem Baum organisiert, da SEF-Code auf der kontextfreien Sprache HTML basiert, welche beliebig viele Verschachtelungen haben kann. Dabei entspricht eine Schachtelung einem Zweig im Baum.

Dieser wird im Folgenden auch als Daten-Baum bezeichnet und ist zwecks Kompatibilität und Konsistenz in nativem JavaScript (2.1.4) implementiert (3.3, Abbildung 4.3). In diesem sind Interfaces nicht möglich und abstrakte Klassen nur über JSDoc definierbar.

Daten-Typ	Attribute	Besonderheit
Element	Attribute	
Container	Attribute Kindknoten	
Inhalt	Inhalt	
Wurzel	Kindknoten	Keine Eltern
Fehler	Inhalt Fehlernachricht	Keine Eltern

Tabelle 4.1: Daten-Typen des Daten-Baums

Die Struktur des Daten-Baums basiert auf einer Abstraktion der Vue-Template-Struktur (Tabelle 3.1) und bildet jeden Daten-Typ wie in Tabelle 4.1 ab, dabei werden einige Elemente aus der Vue-Template-Struktur in der Daten-Baum-Struktur zusammengefasst. Die entstandene Daten-Baum-Struktur ist in Abbildung 5.6 als Klassendiagramm abgebildet und wird in den folgenden Unterabschnitten erläutert.

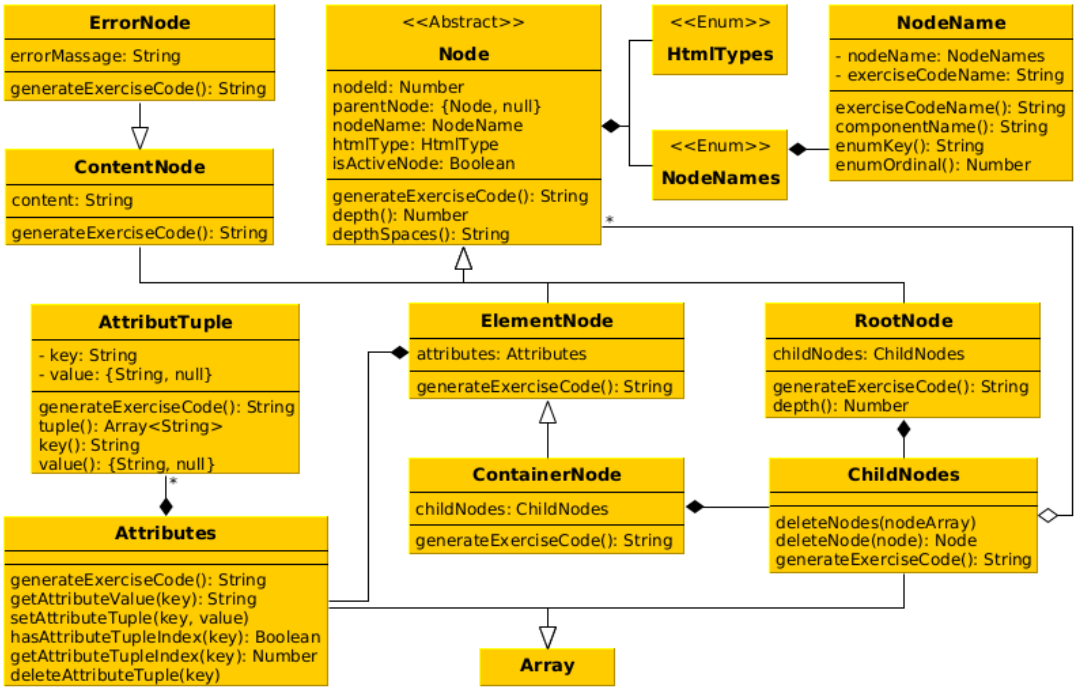

Abbildung 4.3: Klassendiagramm der Datenstruktur für den Systementwurf

Node

Als Grundlage dient eine als abstrakt deklarierte Klasse welche alle grundlegenden Daten enthält, wie eine einzigartige Identifikationsnummer, den Elternknoten und ein Bool ob der Knoten aktuell ausgewählt ist. Alle anderen Knoten erben diese Attribute und fügen ihre eigenen entsprechend der Tabelle 4.1 hinzu, wie es in Abbildung 4.3 zu sehen ist.

Eine Ausnahme bildet hierbei der Wurzelknoten, da es nicht möglich ist, dass dieser die Kinder-Liste erbt, ohne auch die Map der Attribute zu erben. Wenn man eine abstrakte Klasse zum Erben von der Kinder-Liste implementieren würde, müsste der Knoten, welcher die Kinder-Liste und die Attribut-Map hält, von zwei Klassen erben. Dieses ist in JavaScript nicht möglich und könnte ansonsten einen 'Diamond of Death' zur Folge haben.

Attributes und ChildNodes

Als native reaktive Listenstrukturen bietet Vue nur Arrays an, andere Strukturen wie Maps sind schwierig umzusetzen. Daher wird als Grundlage für die Liste der Attribute und Kind-Knoten vom Array geerbt und dann die benötigten Funktionalitäten erweitert. Für das Speichern einzelner Attribute wird das Array als Map genutzt, welches die Attribute als Tupel in einem Objekt mit Key und Value speichert.

NodeNames

Damit wird ein Attribut bezeichnet, welches die interne Bezeichnung als Enum und die Benennung (Name/Tag) aus dem SEF-Code (3.1.1) als String enthält. Die Benennung kann nachträglich nicht verändert werden, bildet aber keine statischen Tupel mit der internen Bezeichnung, da unbekannte Elemente oder Komponenten aus dem SEF-Code beliebig benannt sein können.

HtmlType

Um festzuhalten, aus welchem Typ von Vue-Template-Struktur (Tabelle 3.1) der Knoten entstanden ist, wird dieses als ein Enum festgehalten. Diese Information wird benötigt, um aus dem Knoten korrekten SEF-Code zu generieren.

4.2.3 Visual-Editor-Komponenten

Der Visual-Editor (Abbildung 4.4) ist eine Vue-Komponente und enthält drei weitere Vue-Komponenten für die Operationen, Eigenschaften und den Auswahlbaum, diese sind auf Basis des Gestaltungsentwurfs (4.1.2) und der Anforderungen (3.2) entworfen worden und in den Unterabschnitten erläutert.

Die Visual-Editor-Komponente enthält eine Fabrik zum Erzeugen von Knoten, den selektierten Slide, den aktuell ausgewählten Knoten und den Wurzelknoten der Datenstruktur. Da Vue-Komponenten nur ihre eigenen Attribute und die ihrer Kinder ändern sollen, befinden sich die Funktionen zum Verschieben, Löschen und Setzen des ausgewählten Knotens auch in dieser Komponente; diese Funktionen werden über Attribute und Props an ihre Kinder-Komponenten weitergereicht.

Die Methoden zum Serialisieren und Deserialisieren des SEF-Codes befinden sich auch in der Visual-Editor-Komponente. Die Funktion zum Deserialisieren wird beim Erzeugen der Visual-Editor-Komponente und bei Änderungen am SEF-Code aufgerufen. Falls es einen Fehler beim Deserialisieren gibt, ist der Wurzelknoten ein Fehlerknoten. Beim Zerstören der Visual-Editor-Komponente und bei Änderungen an dem ausgewählten Knoten wird die Funktion zum Serialisieren aufgerufen. Nach jeder Serialisierung wird der entstandene SEF-Code im Slide des SingleSlideEditor und getrennt im Visual-Editor gespeichert. Da eine Änderung am SEF-Code des Slides getätigt wurde, reagiert der SingleSlideEditor reaktiv und gibt den veränderten SEF-Code an den Visual-Editor, den Code-Editor und die Voransicht weiter. Um eine Dauerschleife zu verhindern, wird neuer SEF-Code in den Props des Visual-Editors mit dem getrennt gespeicherten SEF-Code aus der letzten Serialisierung abgeglichen.

Auswahlbaum-Komponente

Diese ist eine Wrapper-Komponente für die Platzierung des Auswahlbaumes und enthält eine weitere Komponente welche sich auf Basis der Kind-Knoten im Wurzelknoten rekursiv aufruft. Dabei gibt es für jeden Knoten eine Komponente. Wenn einer dieser Knoten als aktiv markiert ist, wird er farblich hinterlegt. Falls in der Benutzeroberfläche eine Komponente angeklickt wird, wird die Funktion zum Aktiv-Setzten mit dem enthaltenen Knoten ausgelöst und in der Visual-Editor-Komponente ausgeführt.

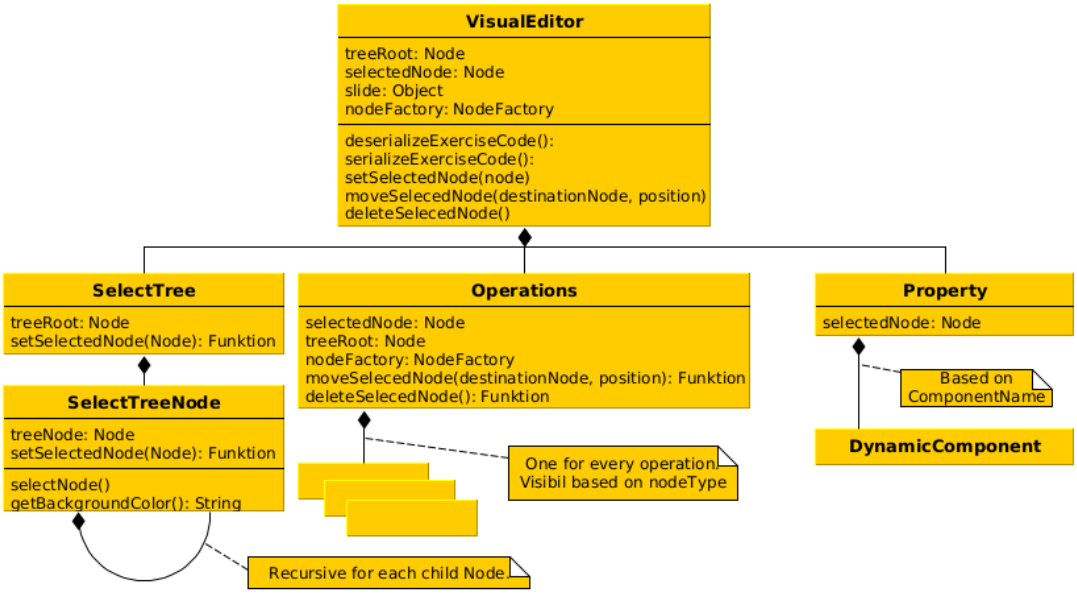

Abbildung 4.4: Komponenten des Visual-Editors im Systementwurf

Eigenschaften-Komponente

Für jeden Knotennamen gibt es eine passende Eigenschafts-Komponente, mit der es möglich ist, die Eigenschaften des Knotens anzuzeigen und zu editieren. Daher haben alle den aktuell ausgewählten Knoten als Attribut/Prop, und wenn sich dieser in der Eigenschaften-Komponente ändert, wird über das 'NodeNames'-Attribut der zugehörige Name ausgelesen und dann die passende Eigenschafts-Komponente als Dynamische-Komponente geladen.

Operationen-Komponente

Diese Komponente beinhaltet den ausgewählten Knoten, die Fabrik zum Erzeugen von Knoten und die Funktionen zum Verschieben und Löschen als Attribut/Prop. Für jede Operation gibt es eine eigene Operation-Komponente. Diese unterteilen sich in zwei Gruppen: eine zum Hinzufügen, Löschen und Wechseln der Position von Kind-Knoten und die andere zum Bewegen und Löschen des ausgewählten Knotens. Alle haben den ausgewählten Knoten und die nötigen Funktionen und Attribute als Attribut/Prop.
Die Operation-Komponenten werden angezeigt auf Basis des Knotentyps und der Anzahl von Kind-Knoten.

4.2.4 Konvertierung der Daten

Um den SEF-Code, welcher unbegrenzt geschachtelt sein kann, in einen Daten-Baum der Datenstruktur zu überführen, wird zuerst ein Lexer, dann ein Parser und zum Schluss ein Visitor genutzt. Diese drei Strukturen werden zuvor mit dem Parser-Generator Antlr4[1] definiert und erzeugt. Damit Fehlinterpretationen vermieden werden, wird eine Bailout-Strategie angewendet. Durch diese wird bei jedem Fehler der Auftritt die Verarbeitung des SEF-Codes abgebrochen und ein Fehlerknoten mit dem unveränderten SEF-Code als Inhalt zurückgegeben. Bei einem erfolgreichen Durchlauf wird ein Daten-Baum erstellt. Dadurch sind Lexer, Parser und Visitor entkoppelt, was ein Austauschen ermöglicht.

Lexer

Der Lexer verarbeitet den SEF-Code (3.1.1) als Zeichenstrom in logisch zusammengehörigen Einheiten, Token. Die HTML-Tag-/Komponenten-Namen-Blöcke werden dabei gruppiert und in einem Sublexer verarbeitet, welcher über '<' geöffnet und mit '>' oder '/>' geschlossen wird; dies ermöglicht es, eine Schachtelung zu verarbeiten.

Parser

Die Datenstruktur nutzt hauptsächlich die Klassen 'ContainerNode', 'ElementNode' und 'ContentNode' (Abbildung 4.3). Daher ist das Ziel, mit der Grammatik die Daten möglichst entsprechend zu parsen. Beim Parsen erzeugt der Parser einen Parse-Baum. Für jede Regel, die erfolgreich angewendet werden kann, wird ein Knoten erstellt, in dem die enthaltenden Regeln und Token gespeichert werden.
Damit alle Token verarbeitet werden, muss in der ersten Regel 'EOF' als letzter Token abgefragt werden. Falls es nicht möglich ist, alle Token zu verarbeiten, wird ein Error geworfen.

Visitor

Nach dem Parsen wird der Parse-Baum durch einen Visitor verarbeitet. Mit diesem ist es möglich, den Parse-Baum in einer beliebigen Reihenfolge zu durchlaufen. Hier wird

parallel zum Durchlaufen ein zweiter Baum mit den Knoten und Klassen der Datenstruktur erstellt. Dafür wird beim Durchlaufen des entsprechenden Parse-Baum-Knoten ein Daten-Baum-Knoten in einer Fabrik erstellt.

Generierung von SEF-Code

Jede Klasse in der Datenstruktur (Abbildung 4.3), hat eine Funktion zum generiere ihres eigenen SEF-Codes. Dafür werden die Funktion zum Erzeugen führender Leerzeichen und die Benennung aus dem Attribut 'NodeNames'(4.2.2) des Knotens verwendet. Beim Generieren ruft jeder Knoten rekursiv seine Kinder und Attribute auf, wodurch der komplette SEF-Code zusammengesetzt wird, insofern der erste Aufruf im Wurzelknoten stattfand.

4.3 Teststrategie

Komponenten, Integration und System Tests prüfen die vorhandenen Funktionalitäten, die Abnahmetests stellen sicher, ob die Anforderungen (3.2) umgesetzt wurden. Die Teststrategie ist noch vorläufig und muss möglicherweise in der Umsetzung angepasst und konkretisiert werden, da wenig Erfahrung mit den verwendeten Technologien besteht.

Komponententests

Die Klassen der Datenstruktur und die für die Konvertierung des SEF-Codes sollen in der Entwicklungsumgebung über eine Äquivalenzklassenbildung und Grenzwertanalyse getestet werden. Bei diesen Blackbox-Tests ist zu beachten, dass mögliche Outputs auch Inputs für die Nachfolge-Klassen sein können.

Integrationstests

Die Vue-Komponenten unter der Verwendung des Daten-Baums der Datenstruktur sollen in der Entwicklungsumgebung über eine Pfadüberdeckung oder Mehrfachbedingungsüberdeckung getestet werden. Bei diesen Whitebox-Tests liegen die Funktionen der Komponenten und die entsprechenden Änderungen im Datenbaum im Fokus, Eingaben über die Benutzeroberfläche hingegen nicht.

Systemtests

Auf Basis der Eingabe- und Auswahl-Möglichkeiten der Benutzeroberfläche sollen Entscheidungstabellentests unter möglichst nahen Realbedingungen ausgeführt werden. Diese sollen die allgemeine Funktionalität des Systems sicherstellen.

Abnahmetests

Diese werden unter Realbedingungen mit Anwendungsfall basierten Tests ausgeführt. Diese können bei Bedarf durch intuitives und exploratives Testen ergänzt werden.

5 Umsetzung

Dieses Kapitel beschreibt die Abweichungen bei der Umsetzung des Entwurfs, Besonderheiten, Probleme und Fehlendes. Unter anderem hat sich das Umwandeln des SEF-Codes als schwierig herausgestellt, die Datenstruktur wurde den Gegebenheiten angepasst und es haben sich neue Probleme gezeigt.

5.1 SEF-Code Konvertierung

Während des Umsetzens der Konvertierung des SEF-Code zum Daten-Baum (4.2.4) haben sich Schwierigkeiten beim Parser und Lexer ergeben. Beim Parser handelt es sich hierbei um die Differenzierung von zwei Alternativen in einer Regel, dabei ist eine Alternative auch komplett in der anderen enthalten. Beim Lexer war es problematisch, die Token für Kommentare und Klartext einheitlich zu lexen, wodurch zwei Parallellösungen entstanden sind.

5.1.1 Parser

Beim Umsetzen der Grammatik in Antlr4 zeigten sich Schwierigkeiten bei der generischen Unterscheidung der Struktur von Leere-Elementen und Block-Elementen, welche in Tabelle 3.1 gezeigt wird. Die Ursachen werden anhand der folgenden Lösungsstrategien erläutert.

Der erste Ansatz war eine generische Lösung, in der die Name-Token (Tags/Namen) beliebig sein können, dafür muss zunächst geprüft werden, ob die Struktur eines Block-Elements vorliegt. Hierfür muss es ein gleiches Paar Name-Token geben, welches sich auf beide Stellen in der Struktur einpasst. Wenn das nicht der Fall ist, dann handelt es sich bei dem ersten Teil um die Struktur eines Leere-Elements.
Für eine solche Prüfung ist die Standardlösung (in Antlr4) ein semantisches Prädikat vor

der Regelalternative für die Block-Elemente, welches prüft, ob beide Name-Token gleich sind. Dafür wird für gewöhnlich im Stream des Lexers eine feste Anzahl von Token vorausgeschaut und dann beide Token verglichen. Da sich zwischen den beiden Name-Token aber unbekannt viele Token befinden, müssten alle verbliebenen durchsucht werden, bis ein zweiter gleicher oder kein Name-Token gefunden wird. Im ungünstigsten Fall entspricht das einem Aufwand von $(Anzahl\ Prüfungen)^{(Anzahl\ Token)}$, diese Berechnung ist häufig, benötigt insgesamt ein merkbares Maß an Berechnungszeit und ist daher ungeeignet.
Als Alternative ist es möglich, die Name-Token mit einem Prädikat zu prüfen, nachdem sie in der Regelalternative für die Block-Elemente bekannt sind. Hierfür ist es aber nötig, die Standard-Error-Strategie zu verwenden, dann wird bei ungleichen Name-Tokens ein Fehlerknoten erstellt und über Backtracking die Struktur eines Leere-Elements erkannt. Hierfür erfordert das richtige Verarbeiten und Beheben des Fehlerknotens ein komplexes Fehlerhandling und zusätzliche komplexe Programmstrukturen. Und es besteht ein Risiko, dass andere unbekannte Fehler durch diese Strategie falsch behandelt oder interpretiert werden, was wiederum den SEF-Code verfälschen kann, daher ist diese Lösung ebenfalls nicht geeignet.

```
VOID_NAME   : 'img' | 'br' | 'hr' | 'meta' | 'input'
            | 'area' | 'base' |  'col' | 'embed' | 'wbr'
            | 'link' |  'param' | 'source' | 'track'
            ;
```

Abbildung 5.1: Namen von Leere-Element

Der zweite Ansatz ist eine teilweise generische Lösung, in dieser sind entweder die Name-Token für Leere-Elemente oder Block-Elemente generisch und die jeweils anderen statisch definiert.
Da es in HTML4 und HTML5 nur eine begrenzte Menge an gültigen Name-Tags für die Struktur von Leere-Elemente gibt und Vue-Komponenten diese für gewöhnlich nicht nutzen, werden diese aktuell als Lösung statisch definiert. Dafür werden diese im Lexer als eigene Token verarbeitet, wie es Abbildung 5.1 zeigt.
Die entstandene Grammatik des Parsers ist in Abbildung 5.2 dargestellt. Diese prüft in der 'element'-Regel zuerst, ob es sich um die Struktur eines Leere-Elements handelt und falls dieses nicht der Fall ist, später, ob es sich um die Struktur eines Block-Elements handelt. Zur Sicherheit, falls ein unbekanntes Leere-Element im SEF-Code sein sollte, werden mit einer Aktion die Name-Token in der Regel für die Struktur des Block-Elements

geprüft. Durch das Verwenden der Bailout-Error-Strategie wird in diesem Fall und in anderen unbekannten Fällen das Parsen abgebrochen.

```
root      : content EOF ;
content   : (element | text | COMMENT)*;
element   : OPEN VOID_NAME attribute* (CLOSE | SLASH_CLOSE)
          | OPEN NAME attribute* SLASH_CLOSE
          | OPEN openName=NAME attribute* CLOSE content OPEN SLASH closeName=NAME CLOSE
            {if ($openName.text !== $closeName.text) {
              throw new Error('Illegel void name: "' + $openName.text) + '"'
            }}
          ;
text      : TEXT_LINE+;
attribute : NAME EQUALS VALUE
          | NAME
          ;
```

Abbildung 5.2: Parser-Grammatik

Eine Lösung, in der die Struktur für Block-Elemente statisch festgelegt wird, ist ungeeignet, da die Aufgaben im SEF-Code über Vue-Komponenten definiert werden. Bei diesen entstehen häufiger neue Aufgabentypen oder Änderungen an den aktuellen, was einen großen Wartungsaufwand durch eine stärkere Kopplung nach sich zieht, da es für jeden Aufgabentyp eine statische Alternative in der 'element'-Regel benötigt, welche auch geändert werden müsste.

5.1.2 Lexer

Im Lexer hat es sich als schwierig herausgestellt, Token für Kommentare und Klartext zu erstellen, wodurch zwei verschiedene Arten der Verarbeitung verwendet wurden.

```
TEXT_LINE  : ~[ \t\n<] ~[\n<]*;
COMMENT: '<!--' .*? '-->' ;
```

Abbildung 5.3: Regeln der Lexer-Grammatik für Klartext und Kommentare

In Abbildung 5.3 wird jede Zeile Klartext ohne führende Leerzeichen zu einem Token verarbeitet. Ein Kommentar wird hingegen komplett mit alle Symbolen, wie dem Öffnen, Inhalt und Schließen des Kommentars zu einem Token verarbeitet. Dem Lexer ist es nicht möglich, den Inhalt als eigene Regel zu definieren, da der nicht-gierige Operator eine Suchbeschränkung (Schließen) benötigt.

```
case SEF3grammarParser.TEXT_LINE:
    if (!firstTextLine) {
        text += "\n"
    } else {
        firstTextLine = false
    }
    text += ctxChild.getText()
    break
```

Abbildung 5.4: Verarbeitung von Klartext im Visitor

Beim Verarbeiten des Klartextes im Visitor (Abbildung 5.4) muss nun nach jeder Zeile mit Ausnahme der ersten ein Zeilenumbruch hinzugefügt werden.

```
case SEF3grammarParser.COMMENT:
    node = this.visualEditorTreeNodeFactory.createNode(VisualEditorTreeNodeNames.Comment,
                                                       VisualEditorTreeNodeHtmlTypes.Comment,
                                                       parentNode)
    node.visualEditorTreeNodeContent = ctxChild.getText()
        // Remove "<!--" and "-->"
        .substring(4, ctxChild.getText().length - 3)
        // Remove leading spaces after newline
        .replace( searchValue: /\n\s*/g,  replaceValue: "\n")
    nodes.push(node)
    break
```

Abbildung 5.5: Verarbeitung eines Kommentars im Visitor

Beim Verarbeiten des Kommentars (Abbildung 5.5) werden über String-Operationen das Öffnen und Schließen entfernt.

Das Entfernen der Leerzeichen beim Klartext und das Öffnen und Schließen beim Kommentar sind nötig, um eine korrekte und effiziente Darstellung für das Bearbeiten der Texte zu ermöglichen. Eine einheitliche und effiziente Lösung sollte für beide angestrebt werden, um die Komplexität und Lesbarkeit des Codes zu verbessern.

5.2 Datenstruktur

Es hat sich gezeigt, dass die Datenstruktur im Entwurf (Abbildung 4.3) zu feingranular und komplex war. Die umgesetzte Datenstruktur in Abbildung 5.6 ist zentralisierter

gestaltet und vereinfacht die Übersicht und damit das weitere Entwickeln des Prototyps. Die folgenden Unterkapitel erläutern die jeweiligen Änderungen oder Optionen für die zukünftige Entwicklung.

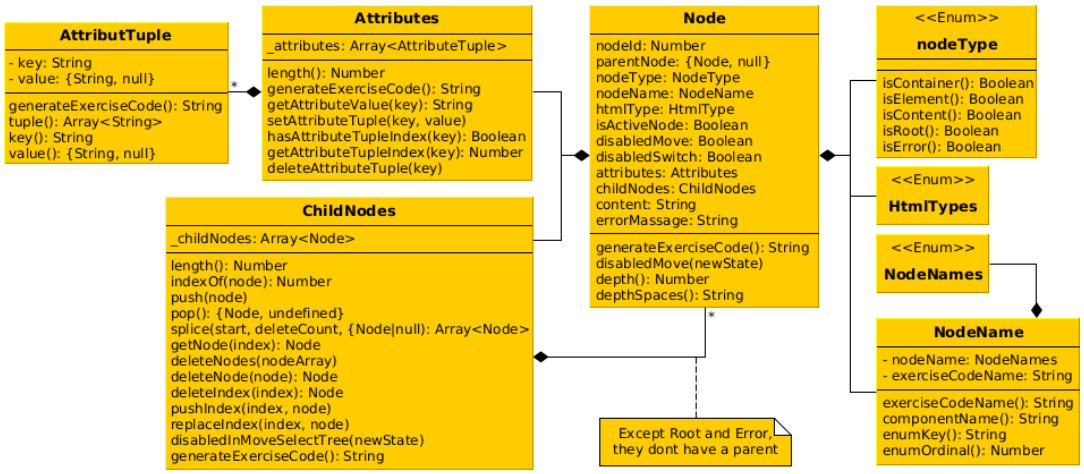

Abbildung 5.6: Klassendiagramm der Datenstruktur

5.2.1 NodeType

Im Regelfall wird Vererbung verwendet, wenn man von Typen spricht. In diesem Fall wurde aber eine schon kleine Klasse gespalten, was den Entwurf unnötig komplex gestaltete. Zusätzlich würde die eine Typprüfung in den Props der Vue-Komponenten einen relativ großen Aufwand erfordern. Und eine Typprüfung im Template-Teil der Vue-Komponenten ist nur schwer umzusetzen, hier müssten zusätzliche Funktionen im Script-Teil eingepflegt werden, welche eine hohe Kopplung zwischen Knoten und Komponenten haben.

Wenn man die Typen in einem Attribut ('nodeType') als Enum verwendet und mit Funktionen für eine Typprüfung ergänzt, ist es möglich, diese Funktionen direkt im Template-Teil der Vue-Komponenten zu verwenden. Diese Umsetzung ist einfacher und performanter. Zusätzlich ist es möglich, diese ohne größeren Aufwand in eine Variante zu transferieren, welche Vererbung oder Komposition nutzt.

5.2.2 SEF-Code Generierung

Aktuell kann jede Klasse in der Datenstruktur (Abbildung 5.6) ihren eigenen SEF-Code generieren, rekursiv für Klassen innerhalb der Klasse. Dies ermöglicht, bei Änderungen in den Klassen das Generieren des SEF-Codes unkompliziert mit anzupassen.

Dadurch liegt aber eine Funktionalität in der Datenstruktur, welche eigentlich nur Daten halten sollte. Es empfiehlt sich im späteren Verlauf, für eine bessere Kohäsion die Generierung des SEF-Codes in eine eigene Klasse zu extrahieren, welche den Datenbaum einlesen und verarbeiten kann.

5.2.3 Boolesche Attribute

Für einige Vue-Komponenten werden boolesche Attribute in den Datenbaum-Knoten benötigt, mit diesen werden Komponenten zum Auswählen in der Benutzeroberfläche deaktiviert. Da diese nicht reaktiv sein müssen, können sie nachträglich in den Vue-Komponenten an die Knoten angefügt werden, was unübersichtlich ist und zu Konflikten führen kann. Für eine bessere Übersicht sind diese in den Knoten eingefügt worden. Dies gilt auch für eine Funktion im Kind-Knoten sowie der Knoten-Klasse, um das rekursive Ändern eines der booleschen Attribute zu ermöglichen Für eine bessere Kohäsion sollten diese Funktionen später in einer Hilfsklasse oder in den entsprechenden Vue-Komponenten platziert werden.

5.2.4 Mutierte Arrays

Aufgrund der Limitierungen bei reaktiven Arrays in Vue (Abschnitt 2.1.2) war es nötig, die Klassen zur Haltung der Kind-Knoten und Attribute anzupassen. Es hat sich gezeigt, dass Vue nicht nur Arrays, sondern auch Klassen, welche von Array erben mutiert werden, wobei alle hinzugefügten Funktionen und Attribute verloren gehen und nur die dann reaktiven Standardfunktionen erhalten bleiben.

Beide Klassen sind nun Wrapper, welche je ein Array enthalten und geeignete Methoden/Funktionen für die angedachte reaktive Nutzung enthalten. Diese Arrays dürfen für eine Nutzung der Reaktivität von Vue nicht privat sein. Da aber keine direkte Nutzung angedacht ist, sind diese über ihre Benennung als 'soft-private' gekennzeichnet.

5.3 Benutzeroberfläche

Für die Umsetzung des Gestaltungsentwurfs (Abbildung 4.1) mit den Visual-Editor-Komponenen (Abbildung 4.4), wurden einige Details angepasst oder ergänzt. In Abbildung 5.7 ist die entstandene Benutzeroberfläche zu sehen und Abbildung 5.8 zeigt die dafür erstellten Komponenten.

Da es nur möglich ist, im Visual-Editor, Code-Editor oder in der Voransicht zu arbeiten und im Editor ein Wechsel nur durch Scrollen möglich war, wurde eine Auswahl über umschaltbare Knöpfe in einer Leiste hinzugefügt. Dadurch ist das Wechseln angenehmer und es werden nur relevant Informationen angezeigt (2.2.2). Hierbei ist der Visual-Editor nur auswählbar, wenn der Slide Inhalt als Vue-Template/SEF-Code gekennzeichnet ist.

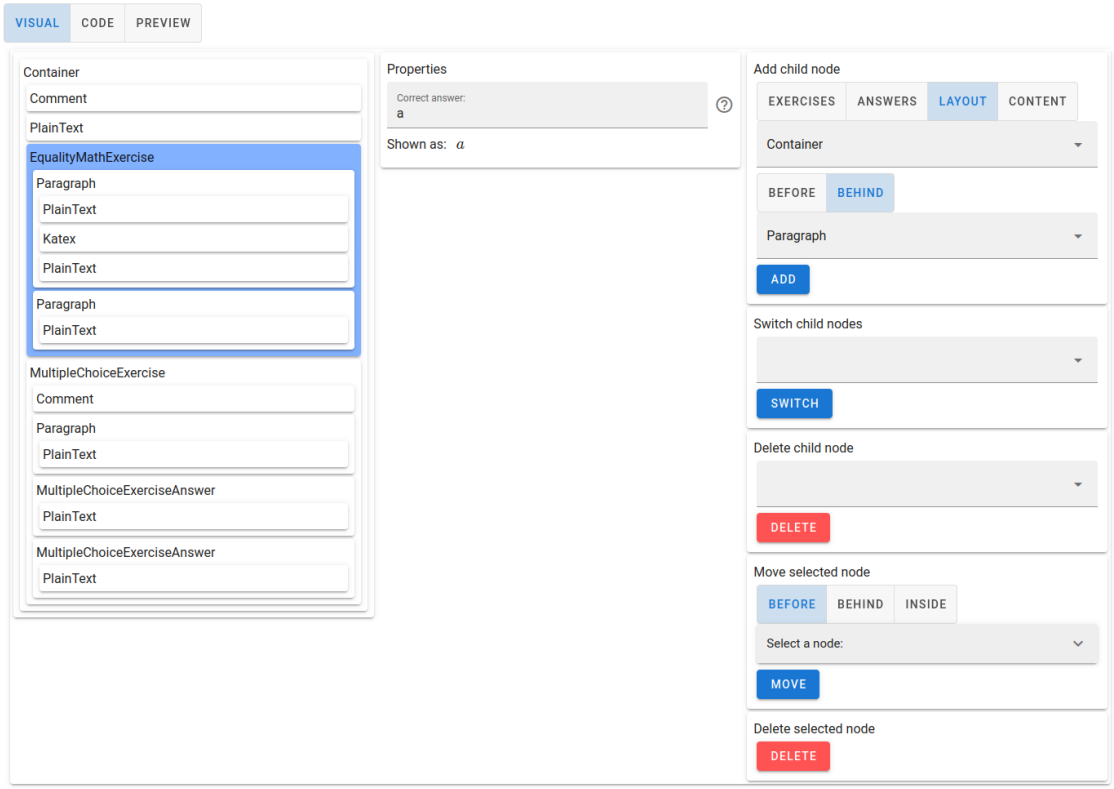

Abbildung 5.7: Benutzeroberfläche des Visual-Editors

Für eine Bedienung, welche noch intuitiver ist, wurde die Farbgestaltung leicht ergänzt. Die ausgewählte Komponente und die aktiven Knöpfe von Auswahloptionen sind nun in hellblau gestaltet. Alle Knöpfe, mit denen nicht destruktive Aktionen ausgeführt werden,

sind dunkelblau und alle Knöpfe, welche destruktive Aktionen ausführen, sind weiterhin rot.

Es wurden in der Operation zum hinzufügen von Kind-Knoten Auswahloptionen hinzugefügt, mit diesen ist es möglich, die zur Verfügung stehenden Komponenten zu gliedern, was ein schnelleres Finden ermöglicht.

Da Zweifel an der Sinnhaftigkeit aufkamen, wurden die Möglichkeiten zum bearbeiten des Layouts nicht umgesetzt. Eine nähere Erläuterung ist im Abschnitt 5.5 zu finden.

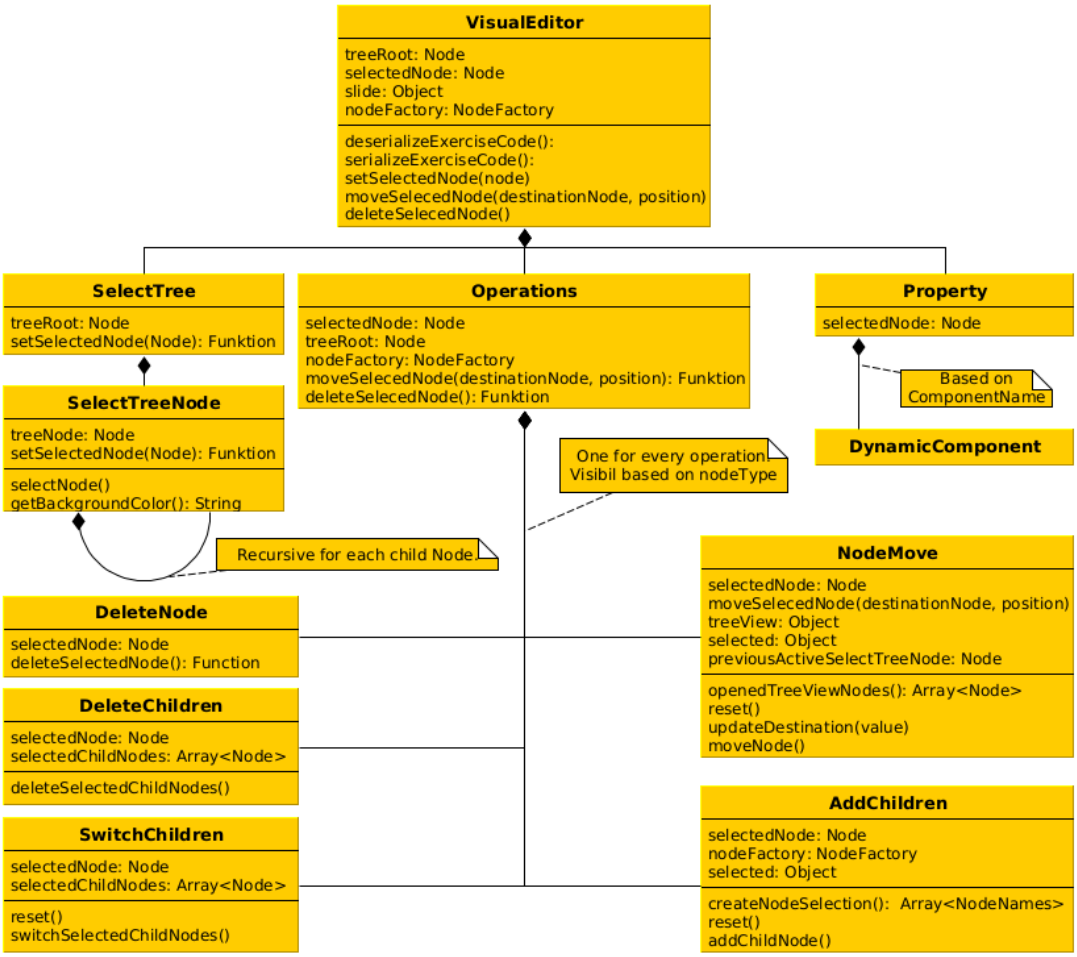

Abbildung 5.8: Komponenten des Visual-Editors

Die Anzeige der Eigenschaften aus Abbildung 5.8 ist mit Hilfe einer Dynamischen-Komponente von Vue umgesetzt.

```
<template>
  <v-container class="pa-0 ma-0">
    <component
        class="pa-2 ma-1"
        :is="this.activeSelectTreeNode.nodeName.componentName"
        :activeSelectTreeNode="activeSelectTreeNode"
    ></component>
  </v-container>
</template>

<script>
import ...

export default {
  name: "VisualEditorProperty",
  components: {
    // Container
    VisualEditorPropertyContainer,
    VisualEditorPropertyMultipleChoiceExercise,
    VisualEditorPropertyMultipleChoiceExerciseAnswer,
    VisualEditorPropertyEqualityMathExercise,
    VisualEditorPropertyParagraph,
    // Element
    VisualEditorPropertyKatex,
    // Content
    VisualEditorPropertyComment,
    VisualEditorPropertyPlainText,
    // Special cases
    VisualEditorPropertyRoot,
    VisualEditorPropertyUnknown,
    VisualEditorPropertyError
  },
  props: {
    activeSelectTreeNode: {
      type: [VisualEditorTreeNode],
      required: true,
    },
  },
}
</script>
```

Abbildung 5.9: Eigenschaften-Komponente

Die Abbildung 5.9 zeigt die Anwendung dieser Komponente. Über das gebundene Spezial-Attribut 'is' vom Element 'component' im Template kann über den Komponenten-Namen

eine im Script registrierte Komponente dynamisch geladen werden. Ein Wechsel des aktiven ausgewählten Knotens wird über das gebundene Attribut erkannt und die Komponente dynamisch ausgetauscht.

5.4 Offene Probleme

5.4.1 Gemischte Texte

Mit der aktuellen Implementierung ist es schwierig, Aufgabentexte zu schreiben, wenn diese mathematische Ausdrücke und Html für die Darstellung/Umbrüche enthalten. Ein Beispiel hierfür ist das folgende:

”Pythagoras behauptet a^2 plus b^2 ist gleich c^2.
Ist diese Behauptung richtig?”

Im SEF-Code ist es noch möglich, eine relativ gute Übersicht zu haben, wie es Abbildung 5.10 zeigt.

```
<p>
  Pythagoras behauptet
  <KatexElement expression='a^2'/>
  plus
  <KatexElement expression='b^2'/>
  ist gleich
  <KatexElement expression='c^2'/>
  .
  <br>
  Ist diese behauptung richtig?
</p>
```

Abbildung 5.10: Gemischter Text als SEF-Code

Hingegen ist im Visual-Editor (Abbildung 5.11) keine vernünftige Übersicht mehr gegeben und es entsteht eine Vielzahl von Komponenten, welche den Auswahl-Baum unnötig vergrößern.

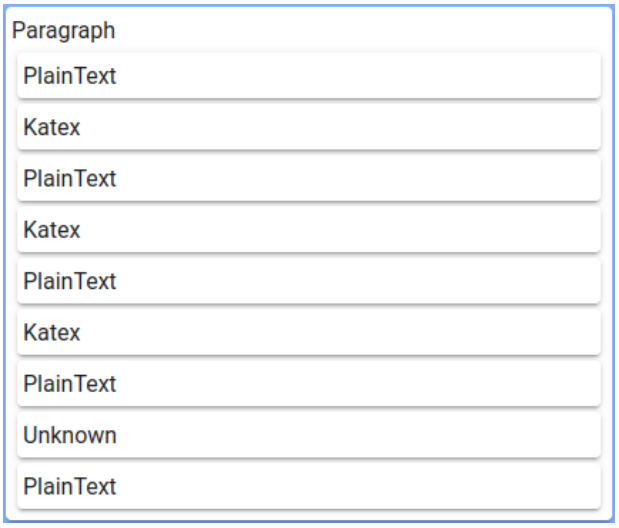

Abbildung 5.11: Gemischter Text im Visual-Editor

Eine mögliche Lösung wäre es, den umgebenen Paragrafen als Text-Komponente zu nutzen und dort eine gemischte Eingabe zu ermöglichen. Um unnötig Code zu vermeiden, sollten Komponenten wie das 'KatexElement' anders angezeigt werden. Denkbar wäre eine Lösung ähnlich wie in LaTeX:

"Pythagoras behauptet a^2 plus b^2 ist gleich c^2.
Ist diese Behauptung richtig?"

Eine solche Schreibweise ist sehr kompakt und einfach zu schreiben und zu lesen. Allerdings muss hier ein geeigneter Weg gefunden werden, um den Text in dieser Form zu parsen. Und ob es möglich ist ein 'KatexElement' für die Darstellung in die $...$ Form überführen bzw. es direkt in SEF-Code einzubinden ist noch unklar.

5.4.2 Paper Prototyping

Die Anzahl der Teilnehmer war gering, da es problematisch war Probanden zu finden. Zusätzlich hat es sich gezeigt, dass die Motivation der Teilnehmer sehr wichtig ist. Auf Grund der Gegebenheiten musste die Methode angepasst werden, was möglicherweise zu schlechteren Ergebnissen geführt hat.

5.4.3 Speichern

Um die Konvention einzuhalten, dass der SEF-Code im Slide immer aktuell ist, wird der Daten-Baum nach jeder Änderung und beim Verlassen des Visual-Editors serialisiert. Aus Performance-Gründen sollte hier eine bessere Lösung gefunden werden.
Eine Möglichkeit ist es, beim Verlassen des Visual-Editors und beim Speichern der Aufgabe eine Serialisierung anzustoßen. Da die Funktion außerhalb des Visual-Editors liegt, wird hierbei eine Abhängigkeit erzeugt. Es gibt auch die Möglichkeit, dass die Vue-Komponenten für die Bearbeitung der Eigenschaften und die Ausführung von Operationen das Serialisieren anstoßen. Hierbei gibt es keine Abhängigkeit nach außen, aber die Performance ist nur mäßig besser.
Bei beiden Möglichkeiten wird technisch die Konvention verletzt, dass der SEF-Code im Slide immer aktuell ist, praktisch kann ein Nutzer während der Bedienung der Editoren keinen Unterschied feststellen. Da diese Eigenschaft die Bidirektionalität zwischen den Editoren ausmacht, sollte eine adäquate Lösung ermittelt werden.

5.5 Was fehlt

Internationalisierung

Für eine Internationalisierung der Vue-Komponenten ist das Plugin Vue-I18n[5] die Standardlösung. Mit dieser ist es möglich, Übersetzungen in mehrere Sprachen zu erstellen, sowohl komponentenbasiert als auch global.

Exception-Handling

Aktuell gibt es nur ein einfaches Exception-Handling, welches ausreichend für einen Prototypen ist. Eine Ausweitung und Präzisierung in Komponenten, Klassen und Parser ist sinnvoll. Und ein Ausbau der Error-Strategie des Parsers könnte zusätzlich sinnvoll sein.

Verifizierung von Eingaben

Die Eingaben in Textfelder werden aktuell nicht auf ihre Korrektheit als SEF-Code verifiziert. Daher ist es möglich, Eingaben zu tätigen, mit denen sich ein Nutzer aussperren kann, da bei einem wiederholten Öffnen des Visual-Editors ein Error beim Parsen entsteht.
Es könnte hier zum Beispiel möglich sein, einen Teil des Parsers zu nutzen, um die Korrektheit der Eingabe zu verifiziert. Allerdings sollte zuvor das Problem der gemischten Texte (5.4.1) gelöst werden, da diese einen großen Teil der Eingaben ausmachen.

Komponenten-Limitierungen

Es gibt keine Limitierungen beim Erzeugen von Komponenten über die Benutzeroberfläche. Zum Beispiel ist es möglich, in einem Paragrafen einen weiten Paragrafen zu platzieren. Für eine sinnvolle Umsetzung sollten mehr oder alle Arten von Komponenten bekannt sein. Da es eine sehr große Menge an Kombinationen gibt, kann eine Umsetzung komplex und unübersichtlich werden; eine geeignete Lösung ist bisher nicht bekannt.

Layout

Die Funktionalität zum Hinzufügen, Ändern und Entfernen des Layouts wurde nicht implementiert. Dies beinhaltet Änderungen von CSS und die horizontale/vertikale Anordnung von Kind-Komponenten. Es kam die Frage auf, ob ein fachfremder Nutzer diese Freiheit haben sollte. Es besteht die Möglichkeit, dass jeder Nutzer seine Aufgaben unterschiedlich gestaltet und es dadurch keine einheitliche Gestaltung mehr über alle Aufgaben gibt. Das könnte zu Verwirrungen und einem schlechteren Nutzungserlebnis führen. Eine Lösung wäre, das Layout komplett über eine begrenzte Auswahl von Komponenten zu steuern.

5.6 Tests

Auf Grund von Zeitmangel wurde der Prototyp minimal durch mich getestet. Hierzu wurden auf der Ebene der System-Tests die Abnahmetests auf Basis der Anwendungsfälle durchgeführt, ergänzt durch eine Mischung aus intuitiven und explorativen Tests, dies

jedoch ohne ein Festhalten der Ergebnisse, stattdessen wurden die gefundenen Fehler sofort behoben. Eine Ausweitung der Tests wie in der Teststrategie beschrieben sollte angestrebt werden.

5.7 Dokumentation

Die Dokumentation der JavaScript-Klassen und Methoden wurde im Code mit JSDoc realisiert, was es der IDE ermöglicht, Types zu prüfen. Komplexe Funktionalitäten und Abläufe sind mit Code-Kommentaren erläutert, sowohl in JavaScript-Klassen als auch in den Vue-Komponenten. Probleme, mögliche Lösungen und noch Offenes sind an entsprechender Stelle im Code als Todos hinterlegt.

Die weiterführende Dokumentation besteht aktuell nur in der Bachelorarbeit, da es sich um einen Prototypen mit noch offenen Gestaltungs-Entscheidungen handelt.

6 Fazit

6.1 Zusammenfassung

Das Ziel der Arbeit war das Integrieren, Entwerfen und Umsetzen eines visuellen Editors in ein vorhandenes System des viaMINT zum Editieren und Anzeigen von Aufgaben für das Slider-Exercise-Framework. Das vorhandenen System ist ein Prototyp, welcher es ermöglicht, SEF-Aufgaben in Form von SEF-Code zu editieren, weshalb technisches Fachwissen für den Umgang mit dem Code benötigt wird. Der visuelle Editor hingegen soll nach dem 'What you see is what you get'-Prinzip funktionieren und es Nutzern damit ermöglichen, ohne technisches Fachwissen SEF-Aufgaben zu editieren. In der Arbeit wird das vorhandene System als Editor bezeichnet und der visuelle Editor als Visual-Editor.

Für das Erreichen des Ziels wurde als erster Schritt eine Anforderungsanalyse durchgeführt. In dieser wurden zuerst das vorhandene System und das enthaltene SEF analysiert. Und anschließend wurden auf der Basis der Analyse und des Ziels der Arbeit die Anforderungen an den Visual-Editor formuliert. Danach wurden einige Technologien (Antlr4[1], Django[3], JavaScript[7] und Vue2[15]), welche für die Umsetzung und Integration geeignet sind, auf ihr Tauglichkeit geprüft.

Unter Einbeziehung der Anforderungen und Technologien wurde ein Entwurf erstellt. In diesem wurde zuerst ein Gestaltungsentwurf für die Darstellung der Benutzeroberfläche mittels einer angepassten Methode des Paper Prototypings[12] und unter Beachtung der "10 Usability Heuristics for User Interface Design"[8] von Jakob Nielson erstellt. Dabei hat sich die angepasste Methode als praktikabel und dynamisch für das Entwerfen der Benutzeroberfläche herausgestellt. Anschließend wurde der Systementwurf für den technischen Teil unter Einbeziehung des Gestaltungsentwurfs und der "8 Golden Rules for Better Interface Design"[10] von Ben Shneiderman erstellt; dieser beinhaltet das Konvertieren des SEF-Codes in eine Baum-Datenstruktur für die Bearbeitung in der Benutzeroberfläche. Für eine Möglichkeit zur Verifizierung und Validierung der Umsetzung wurde zum Schluss noch eine vorläufige Teststrategie erstellt.

Folgend wurde der Entwurf umgesetzt und dabei aufgetretene Abweichungen, Besonderheiten und Probleme werden im Bezug auf das Ziel der Arbeit erläutert. Bei der Umsetzung haben sich Schwierigkeiten mit der Technologie (Antlr4) zum Konvertieren des SEF-Codes in die Baum-Datenstruktur ergeben. Und die entworfene Datenstruktur wurde angepasst, um in Prototypen genutzt zu werden. Zudem wurde aus Zeitmangel das Testen auf ein Minimum reduziert.
Zusätzlich haben sich Probleme ergeben, aus denen zukünftig neue Anforderungen entstehen können, welche im Ausblick aufgeführt werden.

Als Ergebnis ist ein Prototyp des Visual-Editors entstanden, welcher den Anforderungen genügt, erfolgreich in das vorhandene System integriert werden konnte und grundlegend für das Editieren von SEF-Aufgaben geeignet ist. Dieser bietet eine gute Grundlage für eine weitere Entwicklung.

6.2 Ausblick

Für eine Weiterentwicklung des Visual-Editors und eine Verbesserung der Usability gibt es verschiedene Möglichkeiten.

Einige Probleme aus der Umsetzung sollten als neue Anforderungen ausformuliert und umgesetzt werden. Es ist aktuell nicht möglich ist, Texten unkompliziert zu bearbeiten, welche mathematische Ausdrücke und HTML für die Darstellung/Umbrüche enthalten. Und Nutzereingaben in Textfeldern werden nicht auf ihre Korrektheit hin verifiziert, wodurch sich Nutzer aussperren können. Es gibt keine Limitierung, wie die Bestandteile von SEF-Aufgaben ineinandergeschachtelt werden dürfen, daher ist das Erstellen von unsinnigen Konstrukten wie ineinandergeschachtelte Paragraphen möglich. Und hier kam die Frage auf, inwiefern es einem Nutzer erlaubt sein sollte, das Layout von Aufgaben zu verändern.

Weiter sind folgende Verbesserungsmöglichkeiten denkbar: Der Editor könnte intuitiver mit Drag & Drop bedient werden, Aufgaben könnten kooperativ editiert werden und eine zusätzliche Benutzeroberfläche für die Bedienung auf Tabletts und Handys könnte entworfen werden.
Einige kleinere Funktionen für eine bessere Usability wie Tutorials, mehr Feedback für den Nutzer, Rückgängig-Machen von Aktionen, die Implementierung von Tastenkürzeln sowie

die Möglichkeit, vorgefertigte Gruppierungen für SEF-Aufgaben zu erstellen, erscheinen ebenfalls sinnvoll.

Literaturverzeichnis

[1] ANTLR4: *antlr4 git.* – URL https://github.com/antlr/antlr4. – Zugriffsdatum: 29.08.2021

[2] : *Usability: Definitions and concepts.* 2018

[3] DJANGO: *Django web framework.* – URL https://www.djangoproject.com/. – Zugriffsdatum: 30.11.2021

[4] FRAMEWORK, Django R.: *Django REST framework.* – URL https://www.django-rest-framework.org/. – Zugriffsdatum: 30.11.2021

[5] KAWAGUCHI kazuya: *Vue I18n.* – URL https://kazupon.github.io/vue-i18n/. – Zugriffsdatum: 01.01.2022

[6] MIRO: *Miro.com.* – URL https://miro.com. – Zugriffsdatum: 15.11.2021

[7] MOZILLA: *JavaScript.* – URL https://developer.mozilla.org/en-US/docs/Web/JavaScript. – Zugriffsdatum: 01.01.2022

[8] NIELSEN, Jakob: *10 Usability Heuristics for User Interface Design.* – URL https://www.nngroup.com/articles/ten-usability-heuristics/. – Zugriffsdatum: 15.11.2021

[9] NIELSEN, Jakob 1. (Hrsg.): *Usability inspection methods.* Wiley, 1994. – URL http://www.gbv.de/dms/hbz/toc/ht006341384.pdfandhttp://bvbr.bib-bvb.de:8991/F?func=service&doc_library=BVB01&doc_number=006646045&line_number=0001&func_code=DB_RECORDS&service_type=MEDIAandhttp://www.loc.gov/catdir/toc/onix05/93048412.htmlandhttp://www.loc.gov/catdir/description/wiley034/93048412.htmlandhttp://www.loc.gov/catdir/bios/wiley042/93048412.html

[10] Shneiderman, B. ; Plaisant, C. ; Cohen, M. ; Jacobs, S. ; ; Elmqvist, N.: *The Eight Golden Rules of Interface Design.* – URL https://www.cs.umd.edu/users/ben/goldenrules.html. – Zugriffsdatum: 15.11.2021

[11] Shneiderman, B. ; Plaisant, C. ; Cohen, M. ; Jacobs, S. ; ; Elmqvist, N.: *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* 6th edition. Pearson, 2016. – URL http://www.cs.umd.edu/hcil/DTUI6/

[12] Snyder, Carolyn: *Paper prototyping.* – URL https://www.csee.umbc.edu/courses/undergraduate/345/spring11/mitchell/Assignments/CSnyderPaperPrototyping.pdf. – Zugriffsdatum: 20.11.2021

[13] Snyder, Carolyn: *Paper prototyping: The fast and easy way to design and refine user interfaces.* Morgan Kaufmann, 2003

[14] Surmon: *Vue-Codemirror.* – URL https://www.npmjs.com/package/vue-codemirror. – Zugriffsdatum: 01.12.2021

[15] Vuejs: *Vue v2 docs.* – URL https://vuejs.org/v2/guide/. – Zugriffsdatum: 29.08.2021

[16] vuetifyjs: *vuetify v2.5.9 docs.* – URL https://vuetifyjs.com/en/. – Zugriffsdatum: 29.08.2021

[17] Yu, Channy: *Hanmail Paper Prototype UX (User Exprience).* – URL https://youtu.be/GrV2SZuRPv0. – Zugriffsdatum: 20.11.2021

Glossar

SEF Ist eine Abkürzung für 'Slider-Exercise-Framework' und steht für ein Frameworkt für das Erstellen und Ausführen von E-Learning Aufgaben in einem Browser und wird von der Online-Lernplattform viaMINT entwickelt.

Visual-Editor Ist die Bezeichnung des visuellen Editors, welcher in dieser Bachelorarbeit Entworfen, Umsetzt und in das vorhandene System Integriert wird.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Entwurf und Implementierung eines WYSIWYG-Web-Editors für SEF-Aufgaben im viaMINT-Projekt

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original