HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Bearbeitungsbeginn: 01.03.2019

Vorgelegt am: 02.12.2019

Thesis

zur Erlangung des Grades

Master of Science

Design und Entwicklung des UIs für einen Spieleeditor sowie Optimierung der Userexperience

Monika Galkewitsch

Matr. Nr.: 258316

Erstbetreuer: Prof. Jirka Dell´Oro-Friedl

Zweitbetreuer: Prof. Christoph Müller

Abstract

Die Hochschul-intern entwickelte und auf Web-Technologien basierende Spiele-Engine FUDGE soll in ihrer Funktion durch einen Spiel-Editor erweitert und unterstützt werden. Zusätzlich wird ein User Interface System für FUDGE benötigt.
Diese Arbeit beschäftigt sich mit der Konzipierung und Umsetzung eines Spiele-Editor für den Einsatz in der Lehre.
Es werden die verwendeten Techniken und Technologien vorgestellt, anschließend werden die entwickelten Konzepte und Modelle im Detail betrachtet.
Im Zuge dieses Projekts entstand ein erster funktionsfähiger Prototyp des FUDGE Spiele-Editors auf Electron und Typescript-Basis.

The internally developed, web-based game-engine FUDGE needs a Game-Editor to expand and support its existing function.
Additionally, a User Interface System needs to be implemented.
This Paper intents document the concept- and development-phases of a game-editor intended for educational use.
The techniques and technologies used are described, as well are the concepts and models developed during the making of this game-editor.
During the course of this project, the first functional prototype of the electron- and typescript based FUDGE game editor was developed.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzugsverzeichnis

FUDGE	Furtwangen University Didactic Game Editor
GUI	Graphical User Interface
UI	User Interface
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
DOM	Document Object Model

1. Einführung

1.1. Problemstellung

FUDGE wird derzeit für den Einsatz in der Lehre entwickelt.
Damit FUDGE seinen vorgesehenen Nutzen erfüllen kann, ist eine grafische Benutzeroberfläche erforderlich.
Zusätzlich ist ein Editor gewünscht, der die Funktionalitäten von FUDGE abbildet und dem Nutzer zur Verfügung gestellt wird.
Diese Arbeit beschäftigt sich mit dem Entwurf und der Entwicklung eines User Interface Framework und eines grafischen Editors für FUDGE.

Das FUDGE Projekt schließt neben dieser Master-Arbeit zahlreiche weitere Arbeiten ein und wird von Professor Jirka Dell'Oro-Friedl betreut und geleitet. Das Projekt wurde ins Leben gerufen aus dem Wunsch heraus, eine geeignete Entwicklungsumgebung für die Module *Spielentwicklung 2D* und *Spielentwicklung 3D* zu haben. Zuvor wurden für diese jeweils *Adobe Animate* und *Unity* verwendet. Zwar boten beide Umgebungen viele Werkzeuge für die Erstellung von Spiele-Prototypen, da allerdings die Fächer einen starken Fokus auf die Erfüllung von praktischen Arbeiten hatten, fiel das Teilen und korrigieren der dadurch anstehenden Abgaben recht schwer.
FUDGE wurde konzipiert um diesen Problemen zu entgehen.[1]

1.2. Begriffserklärung

1.2.1. Graphical User Interface

Unter einer *Benutzerschnittstelle* versteht man im Generellen jede Art von Kommunikationsschnittstelle zwischen Mensch und Computer. Dieser Begriff kann sowohl Hardware als auch Software Schnittstellen einschließen. [2]
Diese Arbeit konzentriert sich speziell auf den Software-Teil dieses Ausdrucks, im besonderen liegt der Fokus auf grafische Benutzerschnittstellen.
Der englische Ausdruck *User Interface* wird synonym benutzt, um diese grafischen Benutzerschnittstellen zu bezeichnen.
Der Ausdruck *Grafische Benutzeroberfläche* (oder Benutzeroberfläche) referenziert sich auf die grafische Anzeige der Benutzerschnittstelle.

1.2.2. Graphical User Interface Framework (GUI Framework)

In den Anfangstagen der Software-Entwicklung wurden Anwendungen in der Regel mit sehr wenig Werkzeug-Unterstützung entwickelt.

Zu diesen gehörten Compiler, Standard Programmierbibliotheken (Libraries) und die Anwendungs-Programmierschnittstelle (Application Programming Interface, kurz API) des Betriebssystems für das entwickelt wurde.
Über die Jahre wurde mehr und mehr entdeckt, wie durch Abstraktion redundanter Code minimiert werden konnte. Mit der wachsenden Akzeptanz der Software Industrie für objektorientierte Sprachen entwickelte sich das Konzept von *Frameworks* wie es heute bekannt ist. (Siehe [3: 1ff])
Als Framework wird eine Programmierbibliothek bezeichnet, die objektorientiert und erweiterbar ist. Der Hauptunterschied zu anderen Software Bibliotheken besteht in der Erweiterbarkeit.
Nutzer des Frameworks können das Framework durch neue oder abgewandelte Funktionalitäten erweitern und für den jeweiligen Zweck der zu entwickelnden Anwendung anpassen.
Ein anderer Unterschied ist der Umfang eines Frameworks. In der Regel beschreibt der Ausdruck *Framework* eine sehr umfangreiche Bibliothek, welche eine Vielzahl an Anwendungsfällen abdeckt. Es gibt allerdings auch durchaus Frameworks, welche nur ein kleines Set an Usecases abbilden und somit mehr spezialisiert sind.
Ein Graphical User Interface Framework eine solche Programmbibliothek, welche Nutzer bei der Erstellung von Graphischen Benutzeroberflächen unterstützt.
Im Fall dieser Arbeit wird *FUDGE UI* als User Interface Framework bezeichnet, da es die oben genannten Kriterien gut beschreibt. FUDGE UI ist hierbei ein recht kleines und simples Framework und wurde explizit nur für die Verwendung mit FUDGE entwickelt. Zudem kommt der Zeitaufwand, der mit der Entwicklung eines Frameworks verbunden ist.
Aus diesen Gründen ist der Funktionsumfang von FUDGE UI relativ eingeschränkt.

1.3. Recherche

Die Recherchen bezogen sich primär auf die Methodenfindung zur technischen Umsetzung und Planung von Frameworks und und Software, die sich auf das Erstellen von grafischen Benutzeroberflächen fokusiert.
Ergebnisse dieser Recherchen werden in den folgenden Unterkapiteln zusammengefasst.
Auf tiefer gehende Erklärungen zu den recherchierten Themen wird hierbei verzichtet, da primär die Umsetzung des Projekts (Planung und Entwicklung) Gegenstand dieser Arbeit war.

1.3.1. Designziele eines Frameworks

Das Buch *Framework Design Guidelines* [2] behandelt generelle Richtlinien, die beim Entwurf und der Entwicklung eines Software Frameworks helfen sollen.
Das Buch verwendet das von Microsoft entwickelte .NET Framework als Beispiel, versucht allerdings für alle Größenordnungen relevant zu sein.
Um zu ermitteln was genau ein gut durchdachtes Framework ausmacht, wurden die folgenden Qualitäten konkretisiert.
Der nachfolgende Abschnitt referenziert sich auf [2: 3ff]

- *„Well-Designed Frameworks Are Simple“*
  Frameworks sind oft sehr mächtig, Einfachheit in der Benutzbarkeit sollte allerdings nie vergessen werden und immer angestrebt werden.
  Es ist besonders wünschenswert, dass Nutzer einfache Aufgaben mit Hilfe des Frameworks lösen können, ohne die Dokumentation in Anspruch nehmen zu müssen.

- *„Well-Designed Frameworks Are Expensive to Design“*
  Das Design von Frameworks ist mit viel Zeitaufwand, und subsequent Geldaufwand, verbunden. Ein gutes Design entsteht nicht automatisch und kann nicht als Nebenprodukt der Entwicklung entstehen.

- *„Well-Designed Frameworks are Full of Trade-Offs“*
  Ein Design ist niemals perfekt und Kompromisse müssen akzeptiert werden im Designprozess.

- *„Well-Designed Frameworks Borrow from the Past“*
  Die meisten erfolgreichen Frameworks sind auf Basis von bereits bewährten Konzepten gebaut. Die Implementierung von neuartigen Lösungen ist gewünscht, aber sollte nur mit Vorsicht angewandt werden.

- *„Well-Designed Frameworks Are Designed to Evolve“*
  Erweiterbarkeit ist ein wichtiger Kernpunkt für Frameworks. Jedoch kann dies das Framework schnell sehr komplex werden lassen.

- *„Well-Designed Frameworks Are Integrated“*
  Moderne Frameworks müssen entworfen werden, so dass sie in der Lage sind mit einem großen Software Ökosystem zusammenarbeiten zu können.

- *„Well-Designed Frameworks Are Consistent“*
  Konsistenz in Benutzungsschemas ist eine Schlüsselqualität beim Entwerfen eines guten Frameworks. Durch konsistente Schemas kann ein Benutzer leicht das Wissen von einem Teil des Frameworks zu einem anderen übertragen.
  Die in Kapitel 1.3.3 folgenden Design-Richtlinien, (welche dieses Buch aufstellt), versuchen sich nach diesen Qualitätskriterien zu richten.

1.3.2. Szenario-basiertes Framework Design

Die erste Version von .NET wurde von zahlreichen Nutzern als unhandlich empfunden, auf Grund der Unterschiede zu bisherigen verwendeten Bibliotheken im Windows-Raum, allerdings auch auf Grund von Usability Problemen.
Szenario-basiertes Design wurde entwickelt, um diese Probleme zu vermeiden in zukünftigen Versionen des Frameworks.
Die Idee des Szenario-basierten Framework Designs ist, dass aus einem großen Subset an Szenarios, die ein Framework behandelt, in der Regel nur ein kleiner Kernsatz an Anwendungsfällen die Bedürfnisse der meisten Anwender befriedigt. Somit sollten effektive Frameworks zunächst mit einem kleinen Kernsatz an Anwendungsszenarien im Fokus entworfen werden. Fortgeschrittene Szenarios sind sekundärer Priorität.
Diese Methode ist verwandt mit der Usecase Methodik, allerdings arbeiten

diese auf einer abstrakteren Ebene, da der Fokus mehr auf dem Endnutzer einer Software liegt. Szenario-basiertes Design versucht Szenarios zu beschreiben, die vorkommen in der Verwendung des Frameworks. Andernfalls sind die Ansätze sehr ähnlich.
Hilfreich in Szenario-basiertem Design ist es, die Anwendungsszenarien zunächst zu spezifizieren und anschließend Beispiel-Code zu schreiben, der diese Szenarien abbildet. Aus diesem Beispielcode kann Aufschluss über die benötigte Datenstruktur gewonnen werden.
Ziel hierbei ist es, nicht für die Wartbarkeit der Programmierschnittstelle innerhalb des Frameworks zu entwerfen, sondern für die Verwendbarkeit der Schnittstellen. [3: 13ff]

1.3.3. Technische Richtlinien im Framework Design

Framework Design Guidelines [3] gibt weiterführend Richtlinien an, die helfen sollen, die technische Umsetzung der entworfenen Schnittstellen robuster und erweiterbarer zu machen.
Im Folgenden werden einige dieser Richtlinien aufgeführt, die in Retrospektive sich für diese Arbeit als wichtig erwiesen haben.

- *„DO favor defining classes over interfaces“ [3: 81]*
  Klassen erlauben für leichtere Erweiterung als Interfaces. Grund hierfür ist die Schwierigkeit, neue Methoden und Attribute in Interfaces zu spezifizieren, ohne bereits existente Implementationen dieser zu brechen. Abstrakte Klassen können viele Aufgaben erfüllen, die sonst Interfaces übernehmen würden. Interfaces sind allerdings weiterhin für die Abbildung von Polymorphie und Mehrfach-Vererbung in vielen Sprachen erforderlich.

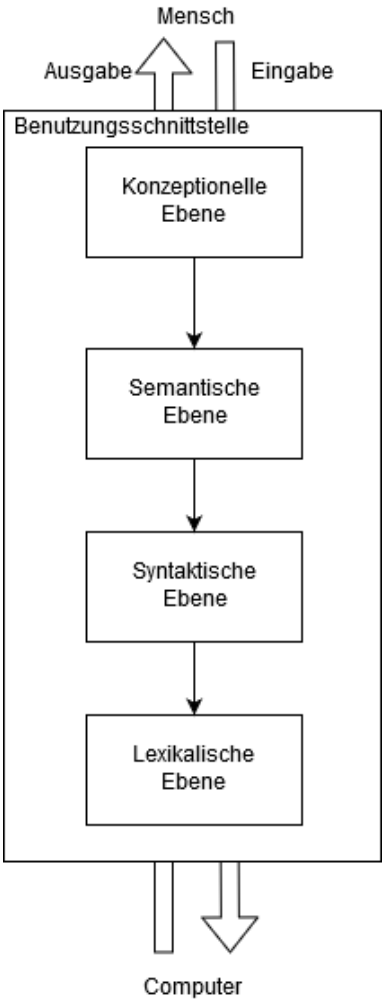
- *„DO NOT define public or protected-internal constructors for abstract types“ [3: 83]*
  Public Constructors sollten nur verwendet werden, wenn Nutzer tatsächliche Instanzen der Klasse anlegen sollen. Abstrakte Klassen sind nicht für diesen Zweck gedacht und somit sollten solche Konstruktoren für sie vermieden werden.
  Es bietet sich viel mehr an, einen protected constructor zu verwenden, werden falls benötigt.

- *„DO use static classes sparingly“ [3: 85]*
  Statische Klassen bieten eine gewisse Einfachheit in ihrer Benutzbarkeit. Zu dem bieten sie gute Abkürzungen, wenn es nur wenig Sinn macht, für eine bestimmte Klasse eine Instanz zu erzeugen.
  Statische Klassen sollten sparsam verwendet werden, hauptsächlich Unterstützungsfunktionen beherbergen und der Zweck der Klasse sollte klar kommuniziert werden.
  Attribute solcher Klassen sollten nie von außerhalb überschieben werden.

- *„... property-heavy design is generally preferable...“ [3: 116]*
  Im Generellen hat sich erwiesen, dass Nutzer schnell überfordert sind von Methoden mit vielen Parametern.
  Nutzer versuchen oft, den Umgang mit einem Framework durch Experimentieren kennenzulernen, deshalb sollte Augenmerk auf eine niedrige Einstiegsbarriere gelegt werden.
  Es ist bevorzugt, vorwiegend Klassen mit Eigenschaften anstelle von Methoden zu verwenden mit Methoden. Wenn Methoden sinnvoller sind als Eigenschaften, sollten sie möglichst wenige Parameter besitzen (bevorzugt mit Standard-Werten).
  Es ist ebenfalls vorzuziehen, Klassen-Eigenschaften ebenfalls mit Standard-Werten zu versehen.
  Zusätzlich sollte vermieden werden, für simple Anwendungsszenarien viele Typ-Initiationen durchführen zu müssen. Dies behindert Nutzer-Experimente und ist somit kontraproduktiv der Benutzbarkeit des Frameworks gegenüber.

1.3.4. Beschreibung einer grafischen Benutzerschnittstelle

Graphische Benutzungsschnittstellen [2] stellt zwei Modelle vor, die das Beschreiben einer Benutzerschnittstelle erleichtern soll: Das **linguistische Modell** und das **Seeheim-Modell**.



Das linguistische Modell [2: 9f] definiert eine Benutzerschnittstelle in vier Abstraktionsebenen. Diese modellieren die Interaktion zwischen Mensch und Computer. Der Kommunikationsstrom von Mensch zu Computer bildet hierbei die Eingabedaten und umgekehrt bildet der Strom von Computer zu Mensch die Ausgabe.
Durch die Abstraktionsebenen lässt sich diese Kommunikation als Ausgabe- und Eingabesprache verstehen.
Es folgt eine Beschreibung der Abstraktionsebenen und jeweils wie diese zum Beispiel eines Dateiverwaltungsprogramms korrelieren.

Abbildung 1: Linguistisches Modell [2]

- Konzeptionelle Ebene
  Auf dieser Ebene werden Konzepte der Anwendung ausgedrückt in konkreten Objekten und Klassen, sowie den Beziehungen zwischen Objekten und Operationen.

  Beispielhaft wären die Konzepte eines Dateiverwaltungsprogramms etwa Dateien, Verzeichnisse und Laufwerke, während Umbenennung, Verschieben, Kopieren und Löschen die Operationen bilden.

- Semantische Ebene
  Die semantische Ebene legt zu realisierende Funktionalitäten fest. Es wird spezifiziert, welche Operationen welche Parameter erfordern und

welche Wirkung diese besitzen.
Im aufgeführten Beispiel wird die *Kopieren* Operation betrachtet. Als Parameter erfordert die Operation eine oder mehrere Eingangsdateien, sowie ein Ziel-Verzeichnis. Vorbedingung dieser Operation ist die Existenz der besagten Dateien und Verzeichnisse.

- Syntaktische Ebene
  Diese Ebene spezifiziert die logischen und zeitlichen Abläufe von Operationen. Hierzu zählen mögliche Programmzustände oder Abläufe von Operationen. Das Layout der Anwendung wird ebenfalls spezifiziert. Zusammensetzungen von Eingaben und Ausgaben werden als sogenannte Tokens abstrahiert.
  Das Beispiel kann auf dieser Ebene wie folgt betrachtet werden:
  Jedes Fenster zeigt ein Verzeichnis an. Der Kopiervorgang erfordert die Auswahl der gewünschten Dateien, das Ausführen des Kopierkommandos und anschließend die Wahl des Zielverzeichnisses.

- Lexikalische Ebene
  Diese Ebene löst die Token-Abstraktion in ihre Bestandteile auf. Hierzu gehört zum Beispiel auch, welche Kommandos über welche Wege erreicht werden können.
  Das oben geführte Beispiel kann auf dieser Ebene so weitergeführt werden, dass Dateien als Icons angezeigt werden. Die Auswahl von Dateien kann über den linken Mausklick erfolgen, während das Kopierkommando durch ein Ziehen und Ablegen der gewünschten Datei im Zielverzeichnis realisiert wird.

Dieses Modell hat eine große Schwäche. Es kann schwer fallen, die genaue Abgrenzung zwischen den Ebenen zu finden. Zum Beispiel kann die Entscheidung, wie das Kopierkommando erreichbar ist theoretisch sowohl in der Syntaktischen als auch der Lexikalischen Ebene stattfinden.
Diese Trennung kann noch schwerer für Ausgaben ausfallen, da Objekte, die von Nutzern direkt manipuliert werden können (in der HTML Welt sind hiermit zum Beispiel Textfelder gemeint), sowohl als Eingabeelement, als auch als direktes Ausgabeelement fungieren.

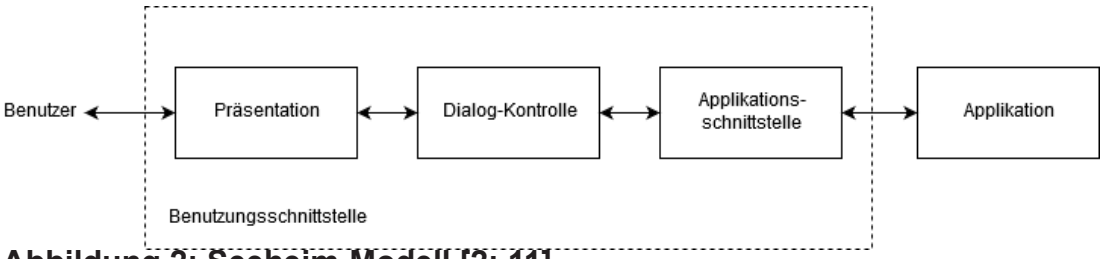


Abbildung 2: Seeheim-Modell [2: 11]

Das Seeheim-Modell [2: 11ff] beschreibt die Interaktion zwischen Mensch und Computer als Konversation, in der die Benutzungsschnittstelle den Vermittler spielt und stellt diesen Dialog in einem Schichtenmodell dar.

Die Präsentation-Schicht ist für die Ein- und Ausgabe von Daten zuständig. An ihr werden sogenannte *Devices* spezifiziert, welche als Eingabe und Ausgabegeräte fungieren. Diese können physischer Natur (Maus, Tastatur, Monitor) sein, aber auch logischer Natur (Menüfelder und andere grafische Ausgaben).

Die Dialogkontroll-Schicht steuert den Dialog zwischen Mensch und Computer, indem sie Tokens zwischen den Schichten austauscht.
Ziel ist es hier, Abläufe formell beschreibbar zu machen, um sie Geräte- und Anwendungs-Unabhängig zu machen.

Die Applikationsschnittstelle interpretiert die von der Dialogkontrolle übermittelten Tokens in Laufzeitdaten. Dies korreliert mit der Übersetzung von syntaktischer Ebene zur semantischen Ebene im zuvor beschriebenen linguistischen Modell.
Die Schwachstellen dieses Modells ist die Ungenauigkeit des Modells. Es wird zum Beispiel offengelassen, welche Form der Kommunikation benutzt wird. Genauso offen ist der genaue Kontrollfluss zwischen den Komponenten des Modells.

1.4. Methodik

Die gewählte Herangehensweise an das Projekt sieht die Spezifizierung und Konzeption des User Interface Frameworks vor.
Hierfür werden die in Kapitel *1.3.2. Szenario-basiertes Framework Design* und Kapitel *1.3.3. Technische Richtlinien im Framework Design* vorgestellten Modelle und Methoden verwendet.
Aus der Spezifikation der Schnittstellen soll ein möglichst vollständiges Set an Funktionalitäten entwickelt und umgesetzt werden.
Die hierdurch entstandenen Funktionen sollen in der Konzeption und Entwicklung des Editors unterstützende Funktionen erfüllen.
Das in Kapitel *1.3.4. Beschreibung einer grafischen Benutzerschnittstelle* vorgestellte Seeheim-Modell soll in der Konzeption unterstützend verwendet werden.
Zusätzlich werden gängige Praktiken aus der Anwendungs-Entwicklung und dem User Interface Design verwendet.

2. Grundlagen

2.1. FUDGE

FUDGE steht für „Furtwangen University Didactic Game Editor“. Es ist eine auf WebGL basierte Spiele-Engine, welche mit Typescript entwickelt wurde. Wie in der Zielsetzung erwähnt ist es Aufgabe dieser Arbeit, den namensgebenden Editor für diese Spiele-Engine zu entwickeln.
FUDGE macht weitreichend Gebrauch von Web-Technologien wie HTML, CSS und Javascript (oder in diesem Fall Typescript). [1]

2.2. Umfang und Verwendung von FUDGE

FUDGE ist primär ein Spiele-Editor und eine Engine, die für die Entwicklung von Spiele-Prototypen in der Lehre konzipiert wurde.
Die Engine kann verwendet werden, um sowohl 2D- als auch 3D-Spiele zu entwickeln.
Es wird angestrebt, FUDGE so auszubauen, dass alle notwendigen Aufgaben für die Erstellung von Spiele-Prototypen innerhalb des Editors erledigt werden können. Hierzu gehören sowohl die Erstellung von Leveln und Skripten, als auch die Erstellung von Grafiken, 3D Modellen und Animationen. Lediglich Sound und Musik sollen außen vor gelassen werden.
Darüber hinaus soll FUDGE über eine enge Bindung mit *GitHub* verfügen und damit Studenten und Lehrenden eine Verbindung bieten, über die Abgaben und Projekte besprochen und bewertet werden können. [1]

2.3. Funktionsweise von FUDGE

Im Folgenden wird die Funktionsweise von FUDGE betrachtet. Es werden nur Funktionalitäten betrachtet, die für diese Arbeit relevant sind.

2.3.1. Datenstruktur

Kern von FUDGE bildet die Knoten-basierte Datenstruktur.
In Anlehnung der im Document Object Model zu findenden Datenstruktur (welches näher betrachtet wird in Kapitel *2.4.2. Document Object Model*), werden Objekte in FUDGE in einer Knotenstruktur angeordnet.
Die Struktur ist gefüllt mit sogenannten *FUDGE Nodes*, welche mehrere *Component*-Objekte halten können. Nodes verfügen jeweils über einen Elternknoten und können mehrere Kindknoten angefügt bekommen. Hieraus entsteht eine Baumstruktur.

Jenseits von den für die Baumstruktur wichtigen Informationen enthalten Nodes selber kaum Eigenschaften oder Funktionalität und erhalten all diese von den angehängten Components, von der Position im Raum bis hin zur Texturierung und verwendeten Meshes.
Inhalte in FUDGE Projekten bauen sich vollkommen aus diesen Knotenstrukturen zusammen.
Strukturell können so Projekte als Wälder aus diesen Node-Strukturen gesehen werden, bei dem jedes *Level* oder *Szene* aus einem solchen Knoten-Baum besteht.

2.3.2. Interne Kommunikation und Datenaustausch

Zur Kommunikation zwischen Klassen und Objekten verwendet FUDGE das Eventsystem des Document Object Models (welches näher betrachtet wird in Kapitel *2.4.2. Document Object Model*).
Das Event-System funktioniert, indem Objekte mit sogenannten *Event-Listener* versehen werden können. Diese Event-Listener sind auf einen Event-Aufruf geprägt. Es können nun Event-Aufrufe ausgelöst werden, die durch die Datenstruktur nach oben steigen. Dieses aufsteigende Verhalten wird auch *bubbling* genannt.
Wenn ein solcher Aufruf am zuvor mit einem Event-Listener versehenen Knoten ankommt, wird eine zuvor spezifizierte Funktion ausgeführt.
Events können zusätzliche Daten halten, was die Kommunikation zwischen Objekten ermöglicht.
Zusätzlich verwendet FUDGE sogenannte *Mutatoren*, um Daten zwischen Objekten auszutauschen.
Mutatoren sind Assoziative Arrays, die Namen und Werte von Klassenattributen speichern. Mutatoren können von Mutable-Klassen während der Laufzeit generiert werden.
Die Kommunikation zwischen Objekten über Mutatoren ist indirekt.
Ein Mutable muss seine Attribute nicht offen teilen um sich von anderen Klassen und Objekten modifizieren zu lassen.

2.3.3. Serializer

Zur Speicherung von Daten werden diese serialisiert und in lesbare Textdaten umgewandelt. Als Datenformat wurde hierfür reines JSON verwendet.
Diese JSON Daten können dann bei Laufzeit wieder de-serialisiert und als

Objekte geladen werden.
Die genaue Methode der Serialisierung ist dem jeweiligen Objekt überlassen, Grundlegend funktioniert der Serialisierungsprozess wie in *Abbildung 3: Serialisierungsprozess [4]*.

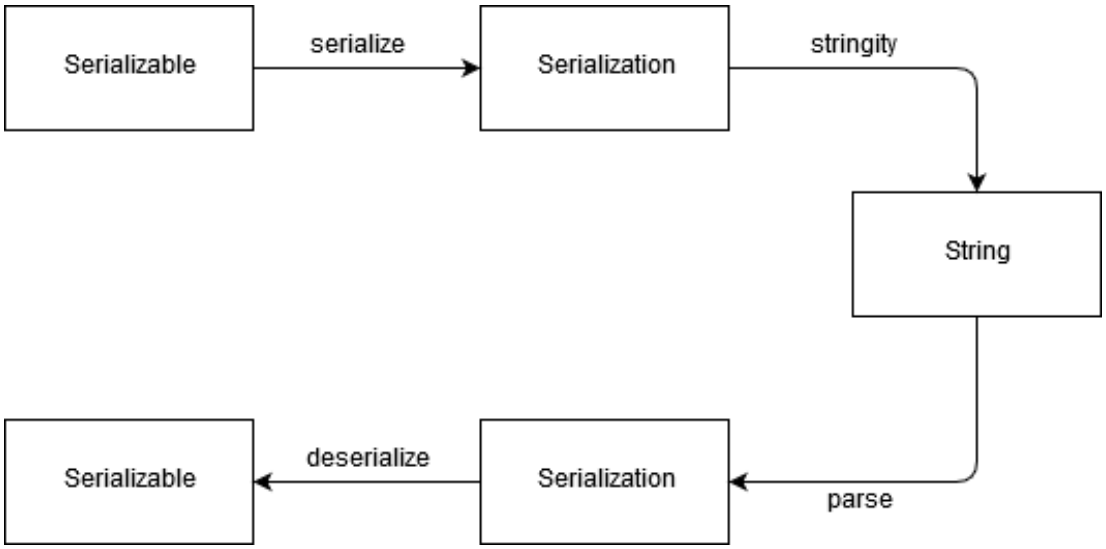


Abbildung 3: Serialisierungsprozess [4]

2.4. Verwendete Technologien

Außerhalb und Innerhalb des FUDGE Kerns werden zusätzlich noch Technologien von Dritten verwendet. Dieses Kapitel beschreibt diese in Kurzform.

2.4.1. Typescript

„Typescript is a typed superset of javascript that compiles into plain javascript“ [5]
Typescript ist eine auf Javascript aufbauende Programmiersprache, welche unter anderen Dingen Typisierung und Objektorientierung zu Javascript hinzufügt. Ziel ist es, Änderungsvorschläge, die im ECMA-Script 6 Standard gemacht wurden in Javascript einzuführen und nutzbar zu machen.
Code wird vom Compiler zu Standard Javascript kompiliert, welcher in jedem modernen Browser ausgeführt werden kann.
Dabei sind die neu eingeführten Sprachkonstrukte optional, was bedeutet, dass, Javascript Code ebenfalls gültiger Typescript Code ist und somit ein Subset von Typescript bildet. [5]
Typescript wird in allen Teilen von FUDGE verwendet, und wird deshalb sowohl in der Entwicklung des User Interface Frameworks, als auch in der Entwicklung des Editors verwendet.

2.4.2. Document Object Model

Das Document Object Model (kurz DOM) [6] ist eine Programmierschnittstelle, welche es erlaubt programmatische Manipulationen an der Struktur und Darstellungsweise von HTML und XML Dokumenten vorzunehmen.
Diese Schnittstelle wurde vom World Wide Web Consortium (Kurz W3C) spezifiziert und findet heute weitläufig Verwendung im Bereich von Webanwendungen.

HTML Daten werden im Document Object Model in einer Baumstruktur dargelegt. Programmierer können durch diese das Dokument beliebig verändern. Gleichzeitig ist es möglich, rein programmatisch Webseiten aufzubauen und dynamisch zu gestalten. Aus diesen Gründen wird eine Kombination aus HTML, CSS und Scripten von manchen auch als *Dynamic HTML* bezeichnet. [6]

Da FUDGE auf Webtechnologien basiert, wird vom Document Object Model an vielen Stellen Gebrauch gemacht. Selbiges gilt auch für das User Interface und den Editor.

2.4.3. GoldenLayout

Golden Layout [7] ist eine Open Source „a multi-screen layout manager“ Javascript Library. Es ermöglicht das strukturieren von Webinhalten in kleinere *Fenster*, die nach Belieben angeordnet werden können.
Es ermöglicht außerdem das Entkoppeln von Fenstern (sprich Inhalte einer Seite in seperaten Browserfenster anzeigen) und die Kommunikation zwischen diesen Golden Layout Fenstern, sogar über die Entkopplung hinweg.
Per Skript werden Bereiche definiert, und in welcher Form diese angeordnet werden sollen. Dies nennt sich das *Layout*.
Dieses Layout wird in einem JSON-ähnlichen Format beschrieben und enthält alle notwendigen Informationen zur Darstellung und Identifikation der Bereiche.
Für das Verständnis der Funktionsweise ist es wichtig diese Darstellungsformen näher zu erläutern.
Golden Layout unterscheidet zwischen Components, Stacks, Columns und Rows.

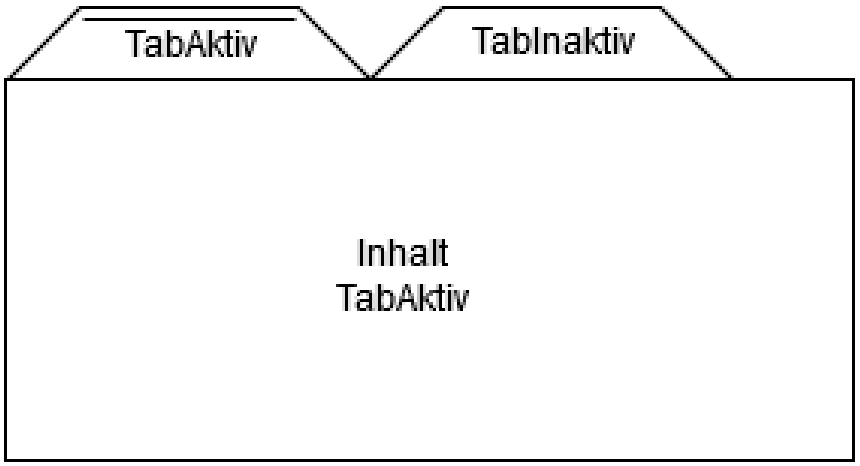


Abbildung 4: Stack-Darstellung

Stacks ordnen Inhalte in Tabs an. Jedes Fenster wird im Stack als kleiner Tab angezeigt. Während der Laufzeit können Fenster zum Stack hinzugefügt oder aus dem Stack gezogen werden.

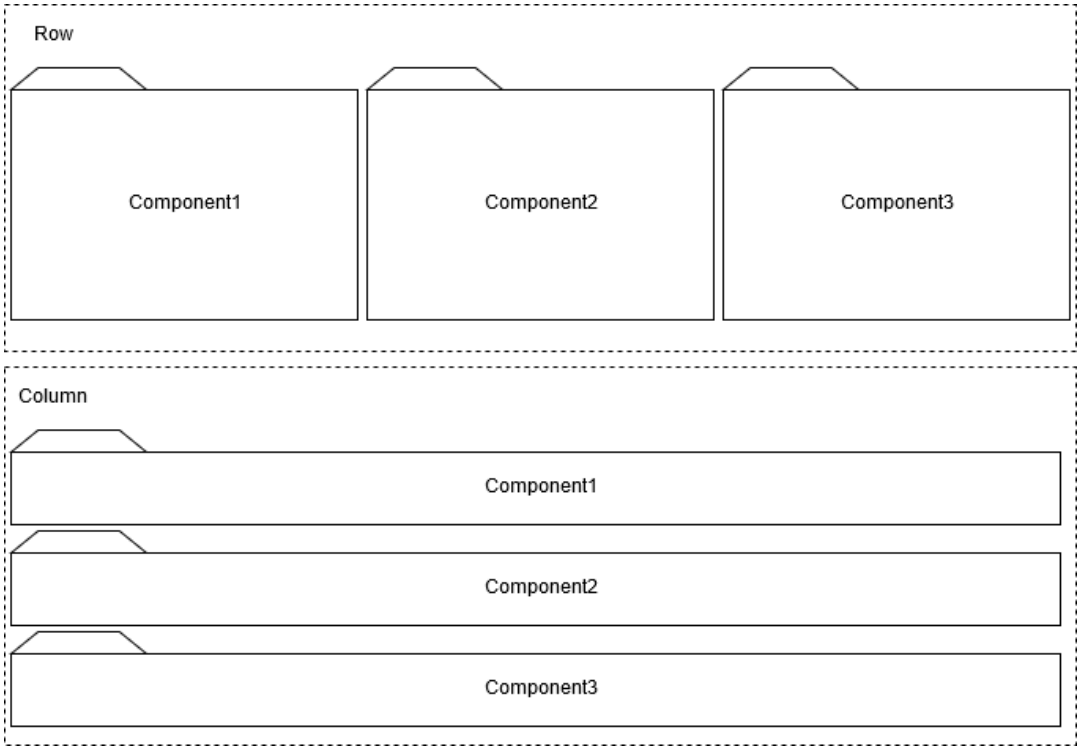


Abbildung 5: Row- und Column-Darstellung

Columns und Rows ordnen ihre Inhalte respektiv als Spalten und Reihen an. Diese Columns und Rows sind als eine Art Container für die eigentlichen Inhalte zu betrachten. Aus diesem Grund mag es den Anschein haben, dass sich in einer Column die Inhalte als Reihen untereinander ausrichten, und in einer Row als Spalten nebeneinander. Deswegen mag die Darstellung zunächst nicht sehr intuitiv erscheinen. Wenn man allerdings betrachtet, dass sich die Ausdrücke *Column* und *Row* nicht auf die Inhalte, sondern das Layout selber beziehen, erscheint es mehr sinnig.

Während die anderen Typen von Golden Layout Containern die Form des Layouts bestimmen, sind Components Container für die eigentliche Inhalte der Seite. Diese haben einen Namen zur Identifikation und einen Titel, der später als Tab-Name angezeigt wird. Zusätzlich können Components noch einen *State* speichern. Dies dient dazu, jegliche Form von Zusatzinformation zu speichern. Es können beim Erstellen des Layouts innerhalb des States Variablen definiert werden, die später befüllt werden können. Dies kann z.B. benutzt werden um Eingaben persistent zu speichern.
Components enthalten allerdings im Layout selber noch keinen Inhalt, sondern fungieren mehr als Gerüst für den eigentlichen Inhalt.
Dieser wird per Code befüllt. Es können im HTML Dokument definierte Elemente in den Container verschoben werden, in der Regel sollten Elemente allerdings ebenfalls im Code erstellt werden.

Aus diesen Layoutdaten wird dann bei Laufzeit die Seite umstrukturiert und in der zuvor definierten Form dargestellt.

Interne Kommunikation in Golden Layout läuft über ein eigenes, Event-gesteuertes Kommunikationssystem ab, das losgelöst ist vom Eventsystem des Document Object Models. Die Funktionsweise ist allerdings sehr ähnlich.
Es können EventListener auf Container gesetzt werden, die auf ein bestimmtes Event warten und eine entsprechende Funktion ausführen, wenn dieses Event auftritt.
Dies deckt die Grundlagen der Funktionsweise von Golden Layout ab.
Golden Layout verfügt über weitere Features, allerdings sind sie für diese Arbeit von minderer Wichtigkeit.

2.4.4. Electron

Electron ist eine von Github entwickelte Open Source library, die es ermöglicht cross-plattform Desktop Applikationen auf Basis von Webtechnologien wie Javascript, HTML/DOM und CSS zu erstellen.
Dies geschieht durch eine Kombination aus Googles *Chromium* Browser-Technologie und der *Node.js* Laufzeitumgebung.
Die Electron Library bietet viele Funktionalitäten, die beim Erstellen einer Desktop-Applikation nützlich sind. Unter den benutzten Features sind das Interne Kommunikationssystem von Electron und die Möglichkeit, dem erstellten Fenster eine programmierbare Menüleiste zu geben.
Electron dient in diesem Projekt als Basis für den Editor und wird hauptsächlich als *Container* für die Funktionen des Editors verwendet. Somit werden viele der Funktionalitäten von Electron nicht vom Editor genutzt und das tiefere Verständnis der Funktionsweise dieser Funktionalitäten ist für diese Arbeit von geringer Wichtigkeit.

2.5. Technologiewahl

Während die Wahl mancher Technologien (wie z.B. DOM, HTML, CSS) recht offensichtlich waren, war die Wahl anderer Technologien etwas schwieriger. In Zusammenarbeit mit Lea Stegk (siehe [9]) wurden Softwarebibliotheken für die Implementation der *Dockable Window* Funktionalität analysiert. Anschließend wurde dann eine dieser betrachteten Bibliotheken ausgewählt. Anforderungen an das Framework bestanden darin, dass es eine Dockable Windows-Funktionalität bietet und somit erlaubt, Inhalte in kleine Unterfenster aufzuteilen. Zusätzlich war gewünscht, dass sich diese Unterfenster herauslösen lassen, wo durch ein neues *Browser*-Fenster geöffnet wird.
Der Browser soll als Container für das User Interface des Editors dienen, im Fall der geplanten End-Anwendung dient Electron als dieser *Browser*.
Es folgt eine Auflistung der betrachteten Bibliotheken, zusammen mit deren Stärken und Schwächen.

- **PhosphorJS** ist ein Javascript basiertes User Interface Framework, welches unter anderem als Haupt-Feature mit hoher Performanz wirbt.
  Das Framework bietet Unterstützung für dockable Windows und Menüs, allerdings fehlt eine Funktionalität für das Öffnen der Interface-Fenster in

separaten Fenstern.
Das Framework ist außerdem sehr umfangreich und erscheint sehr unhandlich.

- **wcDocker [11]** ist ein „Page-Layout Framework“. Die Hauptfunktion des Frameworks ist die Dockable Window-Funktionalität. Zusätzlich bietet wcDocker eigene Context-Menüs, die auf einen Rechtsklick geöffnet werden können.
  Ähnlich zu PhosphorJS ist es Teil eines größeren Frameworks und erscheint für dieses Projekt wenig zweckgemäß. Zudem lassen sich Fenster zwar herauslösen, jedoch können sie nicht in separaten Browser-Fenstern geöffnet werden.
  Zudem wurde die Präsentation von wcDocker als unübersichtlich empfunden und die Context-Menü Funktionalität könnte zu Problemen mit Electron führen (Da Electron eine ähnliche Funktionalität bereitstellt).

- **Isotope [12]** ist eine Programmierbibliothek, die automatisierte Layout Lösungen ermöglicht. Das bedeutet, dass mit Hilfe von Isotope Inhalte einer Website automatisch anhand von zuvor festgelegten Parametern sortiert und angeordnet werden können.
  Es erfüllt somit keine der zuvor festgelegten Anforderungen.

- **Masonry [13]** ist verwandt in Zweck und Funktionsweise zu Isotope, verwendet allerdings ein anderes Sortierverfahren. So wie Isotope zuvor ist auch Masonry für die Zwecke dieses Projekts unbrauchbar.

- **jQuery Layout** war ein Javascript und jQuery basiertes Framework. Aufgabe des Frameworks war es, Inhalte von Webseiten in Unterfenster zu organisieren.
  Es unterstützt keine Drag-und-Drop Funktionalität zur Änderung eines bestehenden Layouts. Es wurde zudem zuletzt in 2012 aktualisiert und zum jetzigen Zeitpunk (26.11.2019) ist es nicht mehr verfügbar.

- **ExtJS [14]** ist ein Javascript-basiertes Framework, spezialisiert auf die schnelle Erstellung von Webanwendungen. Es bietet eine weite Palette an Funktionalitäten, unter anderem Dockable Windows-Layouts.

Es ist nur in einer kommerziellen Version erhältlich, weshalb gegen die Verwendung von ExtJS entschieden wurde.

- **Golden Layout [7]** ist eine Javascript „web app Layout manager“ Bibliothek, basierend auf jQuery.
  Die Bibliothek bietet dockable Windows, die Möglichkeit Unterfenster in neuen Browser-Fenstern zu öffnen, unter anderen Features.
  Golden Layout erfüllte alle gestellten Anforderungen.
  Die Dokumentation wurde als etwas unzureichend empfunden, und die Abhängigkeit zu jQuery erschien störend.

  Nach Abwägung der Optionen wurde sich schlussendlich für Golden Layout entschieden.

3. Konzeption

Die Entwicklung des User Interface für FUDGE besteht aus zweierlei Aufgabenpaketen: Das User Interface Framework und der Editor. Logisch baut der Editor auf dem Framework auf, da sich das User Interface des Editors mit den Funktionalitäten des Frameworks aufbaut, konzeptionell sind diese Pakete dennoch getrennt.
Beide Pakete sollen, ähnlich wie FUDGE, über eine starke Bindung zu Webtechnologien verfügen (speziell dem Document Object Model).
Im Fall des User Interface bedeutet dies, dass Technologien wie HTML, CSS und Javascript/Typescript starke Verwendung finden sollten. Darüber hinaus sollen Strukturen, die das DOM mit sich bringt, stark in die Konzeption des User Interface einfließen.

3.1. Designziele

FUDGE, als Game Editor, soll keine Trennung zwischen Designzeit-Daten und Laufzeit-Daten besitzen, um eine gewisse Transparenz zu schaffen. Wie beschrieben im FUDGE Wiki unter dem Punkt „What are the main features of FUDGE?“: „no separation of runtime and designtime data, games run right out of the source-repository“. [1]
Diese Transparenz soll sich (wenn möglich) durch alle Bereiche von FUDGE ziehen. Im erweiterten Sinne führt dies auch dazu, dass Funktionen, die das FUDGE Entwicklungsteam benutzt, auch den Anwendern von FUDGE zur Verfügung stehen und für diese nützlich sein sollten. Selbiges soll für das User Interface gelten.
Außerdem soll FUDGE primär für die Lehre und für die prototypische Entwicklung von Spielen eingesetzt werden.
Bei der Entwicklung des User Interface Systems müssen diese Gesichtspunkte beachtet werden.

3.2. User Interface Framework

Um die gesteckten Designziele zu erreichen, müssen bei der Konzeption des User Interface Frameworks einige Sachen berücksichtigt werden.
Um Transparenz zu schaffen, müssen *magische Funktionen* vermieden werden, was bedeutet, dass konzeptionelle Black-Boxes vermieden werden müssen. Funktionen innerhalb des Frameworks sollten einsehbar und verständlich für User sein. Einsehbarkeit ist gewährleistet, da FudgeCore und

FudgeUI
Javascript-Libraries mit offenem Sourcecode sind. Verständlichkeit muss angestrebt werden durch Funktionsdesign und sauberen Coding Stil.
Zudem muss das User Interface Framework so konzipiert werden, dass die Anwender Zugriff besitzt auf die selben Funktionalitäten, die das Entwicklerteam benutzt um den Spiele-Editor FUDGE zu bauen.
Enthalten in diese Funktionalitäten sind:

- Funktionen zur schnellen und problemlosen Abbildung von Daten aus der Engine heraus
- Abgebildete Daten sollen sich automatisch aktualisieren können
- Komfortable Funktionen zur Erstellung von typischen Game UI Elementen
- Die Behandlung von Interaktionen mit dem User Interface
- UI-Elemente sollten leicht themeable sein

Diese Funktionalitäten müssen also bereits so konzipiert werden, dass sie möglichst universell anwendbar sind.
Im Folgenden werden die besprochenen Funktionalitäten im genaueren betrachtet.

3.2.1. Automatisch generierte User Interfaces

Es soll die Funktion gegeben werden, bestehende Datenstrukturen schnell und einfach als User Interface abzubilden.
Hier stellen sich einige Probleme:

- Der Versuch, jede beliebige Datenstruktur abzubilden führt schnell dazu, dass die Funktionalität zu komplex wird.
- Welche Form ein beliebiger Datensatz nimmt muss klar festgelegt werden um Konsistenz zu schaffen
- Das generierte User Interface sollte möglichst einfach stylebar sein
- Durch die Funktionalität generierte Elemente müssen mit manuell generierten Elementen zusammenspielen können
- Generierte Elemente müssen mit anderen Funktionalitäten zusammenarbeiten können

Um die Komplexität dieser Funktionalität möglichst niedrig zu halten, muss sich hier für eine klare Struktur entschieden werden. Es bietet sich hierfür an, Strukturen zu verwenden, die weitgehend in anderen Teilen von FUDGE verwendet werden. *Mutatoren* sind hier eine gute Wahl.

Da Mutatoren weitestgehend Verwendung in FUDGE finden, sind sie ein guter Kandidat für die automatische Generierung von User Interfaces.
So kann ein Anwender z.B. für Debugzwecke ein User Interface für seine eigens geschriebenen Objekte schnell und einfach generieren.
Der Einfachheit halber und für Konsistenz sollten generierte User Interfaces auch aus HTMLElementen bestehen. Dies erlaubt es ebenfalls, die generierten Elemente leichter zu stylen, dank CSS.
Durch die Verwendung eines HTMLElements als Container ist das Modifizieren und hinzufügen von Elementen für den Nutzer recht einfach.
Zusätzlich bieten HTMLElemente eine vertraute Struktur, die von Nutzern und Entwicklern als Schnittstelle zu anderen Funktionalitäten verwendet werden kann.

Es ist hierbei wichtig zu betrachten, in welchen Formen solche automatisch generierte User Interfaces auftauchen können.
Zwar werden Mutatoren an den meisten Stellen in FUDGE verwendet, die Anwendungsfälle und die Anforderungen an das User Interface können sich allerdings weitgehend unterscheiden.
Die zugehörigen Components eines Nodes benötigen Eingabefelder, mit denen die zugehörigen Attribute der Components dargestellt und gegebenenfalls modifiziert werden können.
Der Übersichtlichkeit wegen bietet es sich an, diese Attribute anhand ihrer Zugehörigkeit zu Components zu gruppieren.
Fieldset-Elemente scheinen hier nützlich zu sein, da sie Elemente visuell trennen, ein beschreibendes Legenden-Element besitzen und leicht verschachtelt werden können.
Knotenstrukturen (wie z.B. die Strukturen von FUDGE-Nodes) sollten möglichst als Baumstruktur darstellbar sein. Da solche Strukturen meist als Baum dargestellt werden, erscheint diese Darstellungsform als besonders intuitiv.

Zur Darstellung von Baumstrukturen im User Interface bieten sich Listen-Elemente an. Geordnete Listen stellen ohnehin eine Art von Baumstruktur dar, somit sind sie ideale Kandidaten.

3.2.2. Automatisch aktualisierendes User Interface

User Interfaces, die Datenstrukturen aus FUDGE abbilden sollten ebenfalls in der Lage sein, sich automatisch zu aktualisieren. Hierzu gehört sowohl das Befüllen des User Interfaces mit neuen Daten, als auch das Synchronisieren der Datenstruktur mit User-Eingaben.
Für diesen Zweck bietet es sich an, wieder Gebrauch von Mutables zum machen. Gründe hierfür sind weitestgehend die selben wie bereits in Kapitel *3.2.1. Automatisch generierte User Interfaces* erläutert.

Zwar macht es Sinn, eine solche Funktionalität zusammen mit automatisch generierten Interfaces zu verwenden, es sollte allerdings nicht ausschließlich in dieser Verbindung vorkommen.
Grund hierfür ist, dass es Nutzern mehr Freiheit gibt, eigene User Interfaces zu bauen. Die Anbindung von Datensätzen in der Datenstruktur an ein vom User generiertes User Interface sollte ein möglichst einfacher Prozess sein. Eine Lösung, die bereits existierende Strukturen verwendet wäre zu bevorzugen.
HTMLElemente bieten durch die *Name, ID* und *Class* Attribute Möglichkeiten, Elemente im User Interface direkt und möglichst eindeutig zu identifizieren. Die *Queryselection*-Funktion, die HTMLElemente mit sich bringen hilft zusätzlich dabei, Elemente schnell und eindeutig auszuwählen und zu modifizieren.

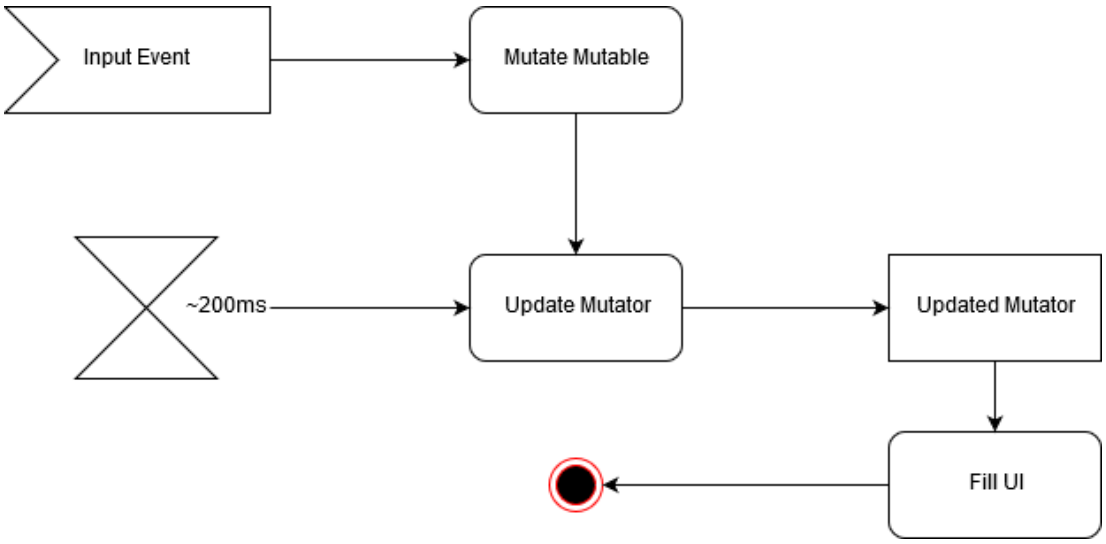


Abbildung 6: Skizze „Automatische Aktualisierung“

Das Aktualisieren sollte Event-basiert ablaufen. Das Document Object Model bietet bereits ein robustes und funktionstüchtiges Eventsystem, das von FUDGE als ganzes durchgängig benutzt wird. Da das User Interface aus HTMLElementen konstruiert wird, macht es Sinn, dieses Eventsystem auch für die Zwecke der automatischen Aktualisierung zu nutzen.

Gezeigt in Abbildung 6 ist eine Skizze, die den groben Ablauf des automatischen Aktualisierungs-Prozesses darstellt.
Hierbei müssen einige Dinge näher betrachtet werden.
Mutatoren werden bei Laufzeit generiert, indem das Mutable alle notwendigen Daten aus den zu übergebenden Attributen zusammen sammelt und in einem assoziativen Array zusammenfasst. Dieser Prozess kann sehr rechenintensiv sein, je nach Komplexität des Mutables und je nach Komplexität der Daten.

Zum Beispiel werden in einem Mutator für die Klasse ComponentTransform Position, Rotation und Skalierung im dreidimensionalen Raum des zugehörigen Objektes in für Nutzer leicht lesbaren relativen Werten gespeichert. Werte sind hierbei immer relativ zum Elternknoten.
Der FUDGE Core arbeitet allerdings nicht mit diesen Werten. So gibt es intern keine x,y,z Koordinaten, welche die Position eines Objektes geben. Stattdessen werden solche Werte in einer Matrix gespeichert. Die relativen Koordinaten Werte werden bei Laufzeit errechnet aus der Matrix, wenn sie erfordert werden.

Components besitzen Logik, die es ihnen erlauben diese Werte zu speichern solange diese aktuell sind. Beim Sammeln der Daten für den Mutator werden dann diese übergeben, falls sie noch aktuell sind.
Wenn ein Objekt seine Transformation jedoch ständig ändern sollte, müssen diese Werte teilweise pro Frame neu errechnet werden.
Menschliche Augen können Informationen ohnehin nicht so schnell wahrnehmen, wie ein Computerbildschirm Bilder diese wiedergeben kann.
Für die Nützlichkeit des User Interfaces müssen Werte ständig aktuell gehalten werden. Ein Nutzer (oder ein Spieler) bezieht nur mäßigen Nutzen aus einem User Interface, das veraltete Informationen anzeigt. Ein Nutzer kann nur

informierte Entscheidungen Treffen, wenn ihm auch aktuelle Informationen geboten werden. Das Fehlen von Informationen kann zu Fehleinschätzungen führen, was sich negativ auf die User Experience auswirkt.
Zusätzlich würde sich ein User Interface, dass zu langsam aktualisiert wird sich nicht responsive anfühlen, was sich zusätzlich negativ auf die User Experience auswirken würde.
Es ist allerdings, wie eben erläutert, nicht notwendig oder zweckmäßig das User Interface jeden Frame zu aktualisieren. Stattdessen sollte eine zeitgesteuerte Lösung bevorzugt werden.
Anzustreben ist es, eine Aktualisierungsrate minimal unterhalb durchschnittlicher menschlicher Wahrnehmung zu wählen.
Laut Online Studien, die von *Human Benchmark* durchgeführt werden befindet sich die durchschnittliche visuelle Reaktionszeit bei Menschen bei etwa 284 Millisekunden. [15] Dies ist allerdings nur ein Durchschnittswert.
Es sollte ein Intervall weit unterm Durchschnitt gewählt werden, um möglichst alle Menschen einzuschließen. Vorschlag hier ist ein Intervall von circa 200ms.
Auf diesem Weg wird Rechenzeit gespart und das User Interface sieht für den Nutzer immer aktuell aus.

Mutables bieten die *Mutieren* Funktionalität. Hierfür werden dem Mutable Mutatordaten übergeben, die dann vom Mutable übernommen werden. So erfüllen Mutables ihre Aufgabe sich modifizieren zu lassen, ohne dass Attribute direkt manipuliert wurden. Dies funktioniert auch, wenn der Mutator unvollständig ist.
Diese Funktionalität ist für die Übergebe von Nutzereingaben besonders interessant.
Das Document Object Model bietet die Möglichkeit, durch das Eventsystem direkt zu erfahren, wann eine Eingabe an einem HTMLElement erfolgte und welche Eingaben gemacht wurden. Dies schließt Mauseingaben ein.
Diese Daten können so in einen Mutator zusammen gefasst werden und dem Mutable zum mutieren übergeben werden.
Es ist vernünftig nach dem Mutieren des Mutables das User Interface zu aktualisieren.
Ebenfalls sollte beachtet werden, dass Nutzereingaben nicht überschrieben werden dürfen durch den Aktualisierungszyklus.

3.2.3. Komfortfunktionen

In Anbetracht der Anforderungen der zuvor besprochenen Aktualisierungs-Funktion und Generierung des User Interfaces, macht es Sinn Komfortfunktionen einzurichten, die User Interface-Elemente in der für diese Funktionalitäten erforderlichen Form anlegt. So würde Usern erlauben, eigene User Interface Elemente in den Aktualisierungszyklus einfacher einzupflegen.
Für komplexere User Interface Strukturen könnte die *Custom Element* Funktionalität des Document Object Models benutzt werden. Diese erlaubt es, eigene HTMLElemente und deren Form zu definieren. Eigene Elemente können dann auch mit Tags versehen werden und so direkt in einem HTML Dokument verwendet werden. Dadurch würde die Verwendung solcher Elemente um einiges vereinfachen.

3.2.4. Inputhandling

Die Verwaltung von Nutzereingaben ist ebenfalls eine der Aufgaben des User Interface Frameworks, wenn auch nur teilweise.
Das User Interface muss auf Maus und Tastatureingaben reagieren können und von FUDGE-Nutzern programmierbar sein.
Diese Funktionalitäten sollten allerdings vollkommen durch das existierende Eventsystem des Document Object Models abgedeckt sein.

3.2.5. Gestaltbarkeit von User Interface Elementen

Die visuelle Gestaltbarkeit des User Interfaces ist für das Framework sehr wichtig. Da die erstellten Elemente auch in Spielen Verwendung finden, sind Gestaltungsmöglichkeiten essenziell, aber auch im Editor kann die Möglichkeit, das Interface visuell anzupassen sehr hilfreich sein. Es ermöglicht nicht nur eine schönere Darstellung, sondern auch die Hervorhebung von Informationen.

Es macht Sinn, diese Gestaltung durch CSS umzusetzen. Die Möglichkeit, HTMLElemente via CSS anzupassen ist bereits gegeben, und da das User Interface diese vorwiegend verwendet, sollte das kein Problem sein, da der Nutzer beim anlegen solcher Elemente einfach die notwendigen Parameter manuell setzen kann.
Die automatische Generierung von Objekten wirft jedoch einige Fragen auf:

- Wie können generierte Elemente gezielt angesprochen werden (für manuelle Stilisierung)?
- Welche Methode sollte für die Identifikation von Elementen verwendet werden (CSS Selektoren)?
- Sollte der Nutzer Einfluss besitzen über die Konventionen dieser Identifikation?
- Wenn ja, wie sollte ein Nutzer Einfluss nehmen können?

Die meisten Probleme können sich lösen lassen, wenn es ein klares Schema gibt, in dem die Elemente identifiziert werden. Alle Elemente vom selben Typ zu selektieren sollte kein Problem sein mit Hilfe von Klassen-Selektoren.
Bestimmte Elemente anzusprechen ist komplizierter, da diese klar und eindeutig identifiziert werden müssen.
Da diese Elemente aus Mutatordaten generiert werden, und diese bekannt sind, gibt es einen klaren gemeinsamen Nenner. Hier würden sich ID-Selektoren anbieten, allerdings müssen IDs einzigartig im ganzen Dokument sein. Das bedeutet, dass sofern zwei Attribute den selben Namen haben sollten die selbe ID hätten. Dies geschieht selbst dann, wenn sie zu verschiedenen Elementen gehören. Das ist unzulässig.
Eine Kombination von Erkennungsmerkmalen sollte deshalb die ID des Elements bilden. Zum Beispiel könnte es hilfreich sein, die ID des Container-Elements mit einfließen zu lassen. Dann muss nur beachtet werden, dass die Containerelemente eine entsprechend einzigartige ID besitzen. So würden Nutzer auch eine gewisse Kontrolle über das Schema besitzen. Da der Containername vom Nutzer gegeben ist und die Namen der Mutatorendaten meistens bekannt sind, können Elemente leichter manuell gestyled werden.

3.3. Editor

Der Editor soll die bevorzugte Methode sein, mit der User mit FUDGE interagieren. Für diesen Zweck soll er die meisten Kernfunktionen von FUDGE abbilden. Kernaufgabe des Editors ist die komfortable Erstellung von Strukturen, die in der Produktion von Spielen in FUDGE wichtig sind. Zu diesen gehören Node-Strukturen, Animationen, sogenannte ‚Sketches' und später auch 3D-Modelle, welche die Kernfelder des Editors bilden.

Erstellte Objekte sollen zur weiteren Verwendung gespeichert werden in JSON Dateien. Dies dient der Lesbarkeit der erstellten Dateien. Es soll ermöglichen, dass die Nutzer einen Einblick in die Funktionsweise von FUDGE bekommen und ermöglichen, Dateien händisch zu modifizieren.
Zur Übersichtlichkeit sollten diese Kernfelder des Editors in mehrere Ansichten unterteilt werden, indem die Kernfunktionen visuell voneinander getrennt sind. Diese Trennung kann auf verschiedene Wege erfolgen, bevorzugt ist es jedoch, den Kernfeldern ein eigenes Fenster zu geben.

Das Standard UI Layout sollte eine gute User-Experience gewährleisten, trotzdem sollte es anpassbar sein auf die Wünsche des Nutzers. Ein User Interface, das in dockbare Fenster strukturiert ist, wird für diesen Zweck bevorzugt.

Das Endprodukt soll ein Editor sein, der die bereits erwähnten Kernfelder einschließt und als Web-basierte Desktop-App ausgeliefert werden.

3.3.1. Dockable Windows

Dockable Windows als Funktionalität neu zu entwickeln würde einiges an Zeit in Anspruch nehmen und erschien als nicht sehr zweckmäßig, da bereits Lösungen existieren. Aus diesen Gründen wurde sich früh dafür entschieden, eine bereits existierende Lösung zu verwenden. Hier soll Golden Layout in Anspruch genommen werden.
Erläuterung der näheren Funktionsweise wurde bereits in Kapitel *2.4.3. GoldenLayout* besprochen, Gründe für die Wahl von Golden Layout gegenüber Konkurrenzprodukten wurden bereits in Kapitel *2.5. Technologiewahl* erläutert.
Im Editor soll das User Interface vom Nutzer angepasst werden können mit Hilfe von Dockable Windows. Es ist hierfür sinnvoll zu betrachten, wie die Elemente des Editor-UIs aufgeteilt werden sollten.
Anhand der Aufgaben, die das User Interface des Editors erfüllen soll, können gewisse Anforderungen bereits erkannt werden.
An dieser Stelle wird nur das Node-Feld des Editors betrachtet, da die Ansichten der anderen Kernfelder nachfolgen sollen und Gegenstand von anderen Arbeiten sind. Jedoch ist das User Interface des Node-Feld die Basis für die anderen Ansichten.

Eine nähere Erläuterung, was ein Feld im Kontext des Editors ist, befindet sich in Kapitel *3.3.3. Struktur*.

Aufgabe des Node-Felds ist es, FUDGE Node Strukturen zu erstellen und zu modifizieren. Folgende Ansichten erscheinen für das Node-Feld sinnvoll. Jeder Punkt aus dieser Liste kann als ein Bereich im User Interface betrachtet werden.

- Ansicht der ganzen Node Struktur
- Ansicht der Components am aktuell betrachteten Knoten
- Renderfenster, das die Node-Struktur abbildet
- Ressourcenansicht

Die gesamte Node-Struktur ist sinnvoll anzuzeigen, um Anwendern einen Überblick über die Struktur des geöffneten Baums zu geben. Eine hierarchische Liste bietet sich an, um die Baumstruktur klar darzustellen.
Um das Navigieren der Struktur einfacher zu machen bietet es sich an, Kind-Elemente beliebig aus- und einblenden zu können.

Es ist adäquat, nicht alle Components für alle Knoten im Baum anzuzeigen, um das Interface übersichtlich zu halten. Stattdessen sollten immer nur die Components des aktuell ausgewählten Knotens angezeigt werden.
Um die Ansicht dieser Components übersichtlich zu halten, sollte jede Component visuell von den anderen getrennt gehalten sein. So sind alle Eingabefelder und Anzeigen in ihrer Zugehörigkeit zu den entsprechenden Components visuell klar kommuniziert. Für weitere Übersichtlichkeit wird es bevorzugt, die Eigenschaften von einzelnen Components aus- und einblenden zu können.

Das Renderfenster kann die Viewport Funktionalität von FUDGE benutzen. Es sollte jedoch zusätzliche Steuerungselemente geben, um die Navigation im dreidimensionalen Raum zu vereinfachen. Diese werden im Kapitel *3.3.5. Gizmos* etwas näher besprochen.

Um das aktuelle Projekt und die geöffnete Knotenstruktur besser verwalten zu können, sollte eine sogenannte *Ressourcen Ansicht* hinzugefügt werden.

Diese funktioniert wie ein herkömmlicher Datei-Explorer wie z.B. der *Explorer* der in Windows zu finden ist zum Beispiel.
Primäre Aufgabe ist es, die gesamten Daten des geöffneten Projekts anzuzeigen und dem Nutzer zu präsentieren. Dabei sollten bestimmte Dateiendungen eigene Icons besitzen, um dem Nutzer klar zu kommunizieren, welche Datei welche Daten enthält.

Es wird hierbei angestrebt, eine möglichst vertraute User Experience zu schaffen [9]. Im Besonderen wurde eine Anlehnung an das User Interface von Unity bevorzugt, da davon auszugehen ist, dass es das Spiel-Editor User Interface ist, mit dem Lernende am vertrautesten sind.

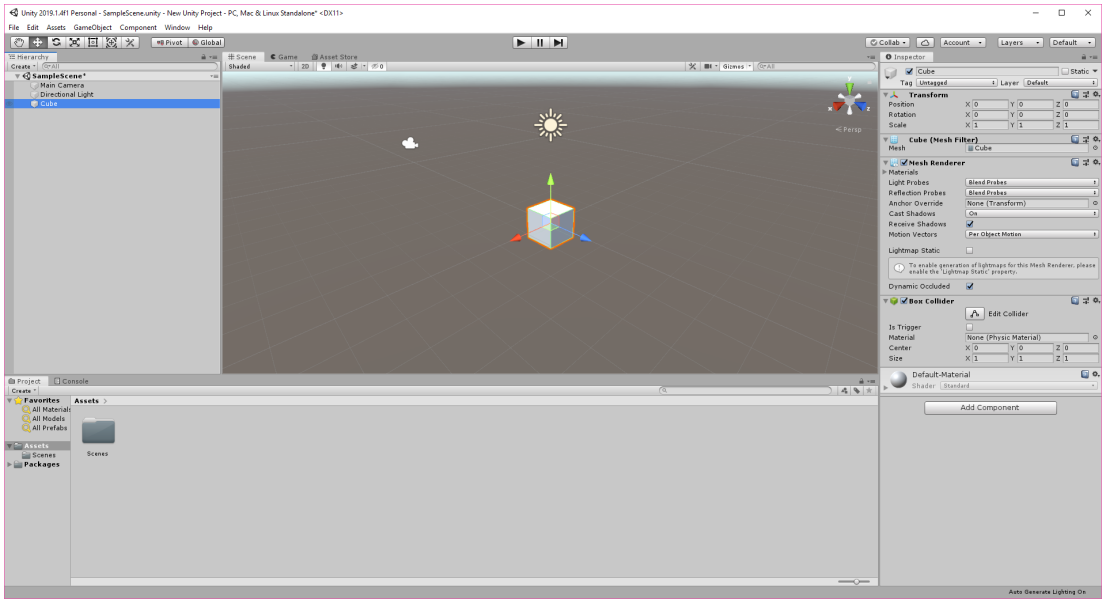


Abbildung 7: Unity Standard User Interface

Abbildung 8 zeigt die schematische Ansicht eines Design-Vorgeschlags für einen Node Editor, basierend auf dem Design des User Interface von Unity. [9]

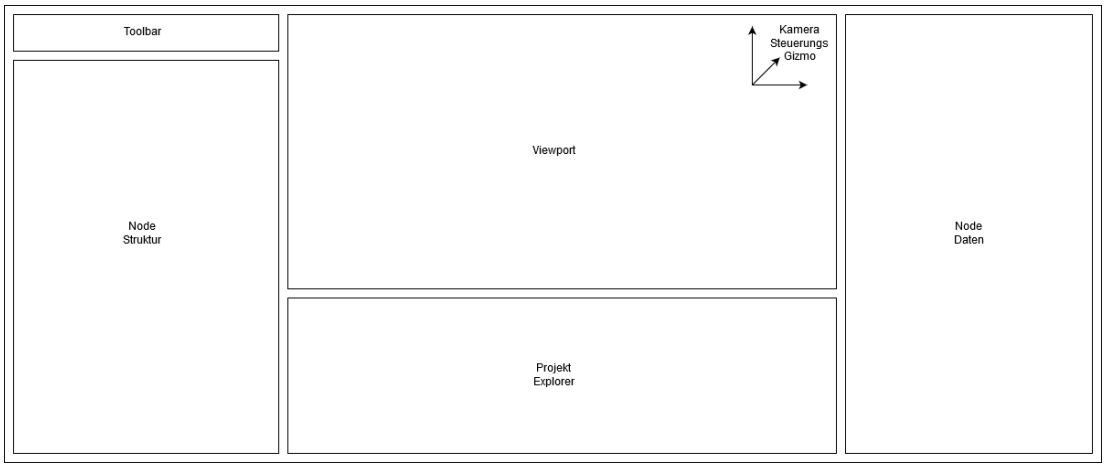


Abbildung 8: Node Editor Designskizze

3.3.2. Fenster-basiert gegen Tab-basiert

Der FUDGE Editor soll über mehrere Bedienfelder verfügen, welche alle verschiedene Aufgaben lösen und ihre eigenen User Interfaces besitzen. Eine klare Strukturierung dieser Felder ist erforderlich für eine gute Benutzbarkeit des Programms.
Hierfür wurden zwei Ansätze konzipiert: Ein *Fenster-basierter* Ansatz und ein *Tab-basierter* Ansatz. Beide haben ihre Vor- und Nachteile, die hier betrachtet werden sollen. Im Folgenden werden die beiden Ansätze näher erläutert und die Stärken und Schwächen der jeweiligen Ansätze aufgezeigt.

Der Fenster-basierte Ansatz betrachtet FUDGE weniger als einen großen Editor, sondern mehr als Sammlung an Editoren, die alle sehr verschiedene Aufgaben erledigen. Jedes der Kernfelder ist hierbei als sein eigener Editor zu betrachten, der zum Gesamtpaket *FUDGE* zugehörig ist.
Um dies zu kommunizieren, würde dieser Ansatz jedem dieser Kernfelder ein eigenes Anwendungsfenster geben.
Diese Anwendungsfenster würden, wenn notwendig, miteinander kommunizieren und ansonsten vollkommen selbstständig agieren.
Electron bietet uns die Möglichkeit, mehrere Fenster zu öffnen, die alle zur selben Electron-Instanz gehören. Diese können dann durch das Electron-Interne Kommunikationssystem Informationen austauschen.

Positiv hierbei ist, dass die Fenster-basierte Lösung eine viel klarere Trennung zwischen den Kernfeldern schafft und verhindert, dass Ansichten aus einem Feld in ein anderes Feld geschoben werden können.

Dies trägt hauptsächlich zur konsistenten Übersichtlichkeit der Editor Anwendung bei. Dadurch, dass Anwendungsfelder nicht vermischt werden können, bleibt die Aufgabe jedes Fensters immer klar.
Die Anwendung auf mehrere Anwendungsfenster aufzuteilen bedeutet allerdings auch, dass die Navigation zwischen diesen unübersichtlich werden kann. Dazu kommt, dass ein weiterer Kommunikationsweg benutzt werden muss, was suboptimal ist. Die Komplexität der Anwendung würde hierdurch ansteigen.
Außerdem geht die logische Zusammengehörigkeit der Anwendungsfenster unter. Alle Fenster gehören zu FUDGE, aber welches dieser Fenster würde in diesem Fall das Hauptfenster bilden? Gibt es ein Hauptfenster? Was passiert wenn ein Fenster geschlossen wird?
Um dieses spezifische Problem zu lösen, wurde entschieden für eine solche Lösung ein *Hauptfenster* einzuführen. Von hier aus werden Projekte geöffnet und andere Anwendungsfenster gestartet. Wenn das Hauptfenster geschlossen wird, schließt die gesamte Anwendung mitsamt aller geöffneten Anwendungsfenster.

Der Tab-basierte Ansatz betrachtet FUDGE kollektiv als einen Editor, der viele Aufgabenfelder abdeckt und diese alle in einer Anwendung vereint. Jedes Kernfeld ist hierbei ein Bedienfeld, das von den anderen getrennt ist. Um die Übersichtlichkeit über die Bedienfelder zu bewahren und Navigation einfacher zu machen, werden alle Kernfelder (ähnlich wie in einem Aktenschrank) durch Registerkarten, sogenannte Tabs, unterteilt.
Golden Layout bietet durch seine *Stack* Funktionalität eine Möglichkeit das eben beschriebene Verhalten abzubilden.
Für diese Lösung ist nur ein Anwendungsfenster erforderlich. Das bedeutet auch, das technisch gesehen alle Felder zum selben HTML-Dokument gehören, wo durch die Kommunikation zwischen Feldern durch das übliche DOM Eventsystem geregelt werden kann.

Die Tab-basierte Lösung hat den Vorteil, dass sie technisch voraussichtlich einfacher ausfällt, da keine zusätzlichen Kommunikationswege verwendet werden müssen.
Dafür könnten Anwender potenziell Ansichten zwischen Feldern hin und her schieben, was sich negativ auf die Benutzbarkeit der Anwendung auswirken

könnte.
Insgesamt ist die Trennung zwischen den Kernfeldern weniger stark und es kann dadurch zu Verwirrung bei den Anwendern kommen.
Golden Layout besitzt von Haus aus keine Möglichkeit, Golden Layout Fenster in ihrer Anordnung einzuschränken, ohne die Möglichkeit vollständig zu entfernen. Es müssen bei dieser Lösung zusätzliche Überlegungen angestellt werden, inwiefern eine solche Begrenzung implementiert werden kann und wie so etwas implementiert werden kann.

Die *Fenster-Lösung* wird an dieser Stelle bevorzugt, da eine gute User Experience höhere Priorität hat als die technische Komplexität der Anwendung.

3.3.3. Struktur

Strukturell wird der Editor in Felder, so genannte *Panels*, und Ansichten, sogenannte *Views*, unterteilt.
Fortan werden die Ausdrücke *Panel* und *Views* verwendet um konsistent mit der Umsetzung zu bleiben und spätere Verwirrung zu vermeiden.

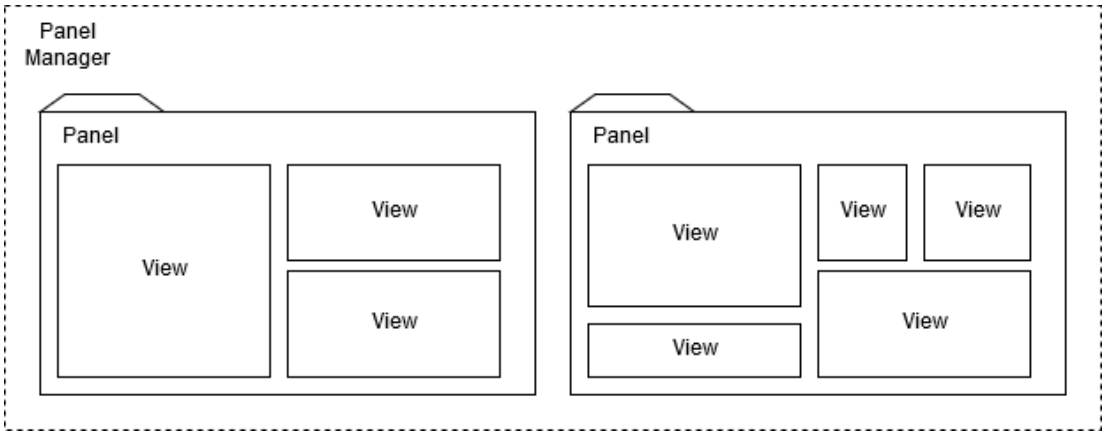


Abbildung 9: Panel-View Struktur

Panel sind logische Container, die einen gewissen Aufgabenbereich abstecken. Diese Panel enthalten Views.
Views können als Teilaufgaben des Arbeitsbereichs gesehen werde.
Views enthalten User Interface Elemente, die zur Erfüllung der Teilaufgabe der View dienen.
Ein Feld kann jeweils mehrere Ansichten enthalten, wobei jede dieser Views mehrere Anzeigeelemente besitzen kann.

Panels besitzen wenig Anwendungslogik, welche hauptsächlich der Kommunikation zwischen den Views im Panel dienen.
Die meiste Logik wird von den Views gehalten, da diese den Großteil der Aufgaben des Panels bewältigen.

Diese logischen Elemente werden dazu benutzt, um den Zusammenhang der im User Interface des Editors gezeigten Elemente zu ordnen, primär um sie mit Golden Layout in die benötigte Form zu bringen.
In dieser Struktur werden User Interface Elemente verwaltet durch ihre Views, während Ansichten verwaltet werden durch ihre Panel.

Zur Verwaltung aller Panel benötigt es ein höheres Verwaltungselement. Es wird hier der *Panel Manager* genannt.
Die Aufgaben des Panel Managers sind verwaltungstechnischer Natur. Er kümmert sich um das Anlegen und Entfernen von Panels, das Halten aller Panel-Instanzen und die Kommunikation zwischen Panels.
Es scheint sinngemäß, dass es immer nur eine Panel Manager Instanz innerhalb von FUDGE geben kann.
Zudem erscheint es sinnvoll, dem Panel Manager zusätzlich die Verwaltung von Golden Layout anzuvertrauen.

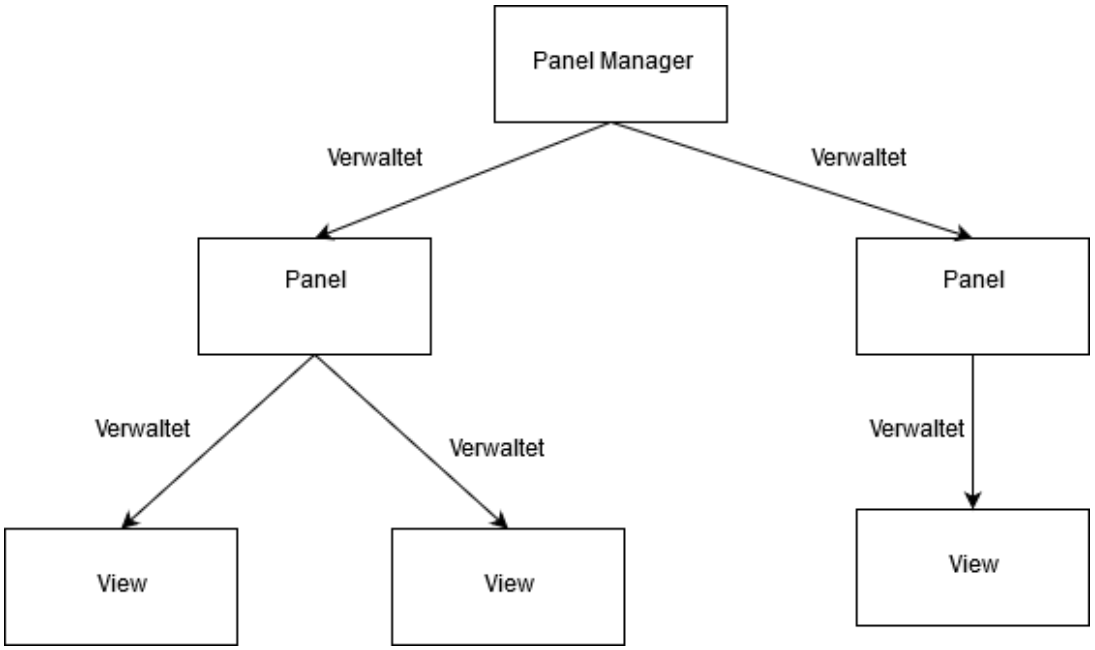


Abbildung 10: Panel-View Hierarchie

3.3.4. Interne Kommunikation

Kommunikation innerhalb des Editors findet mit Hilfe des DOM Eventsystems statt.
Event Aufrufe, die an den User Interface Elementen ausgelöst werden, erreichen die View (durch das Bubbling-Verhalten).
Die View gibt diese Eventdaten dann weiter an das zugehörige Panel, welches es an den Panel Manager weiterreicht.

Views und Panels können Event Listener an das jeweilig hierarchisch höher gelegene Element setzen, um auf entsprechende Eventaufrufe reagieren zu können.
Auf diese Art können alle Elemente im User Interface miteinander über Events kommunizieren.

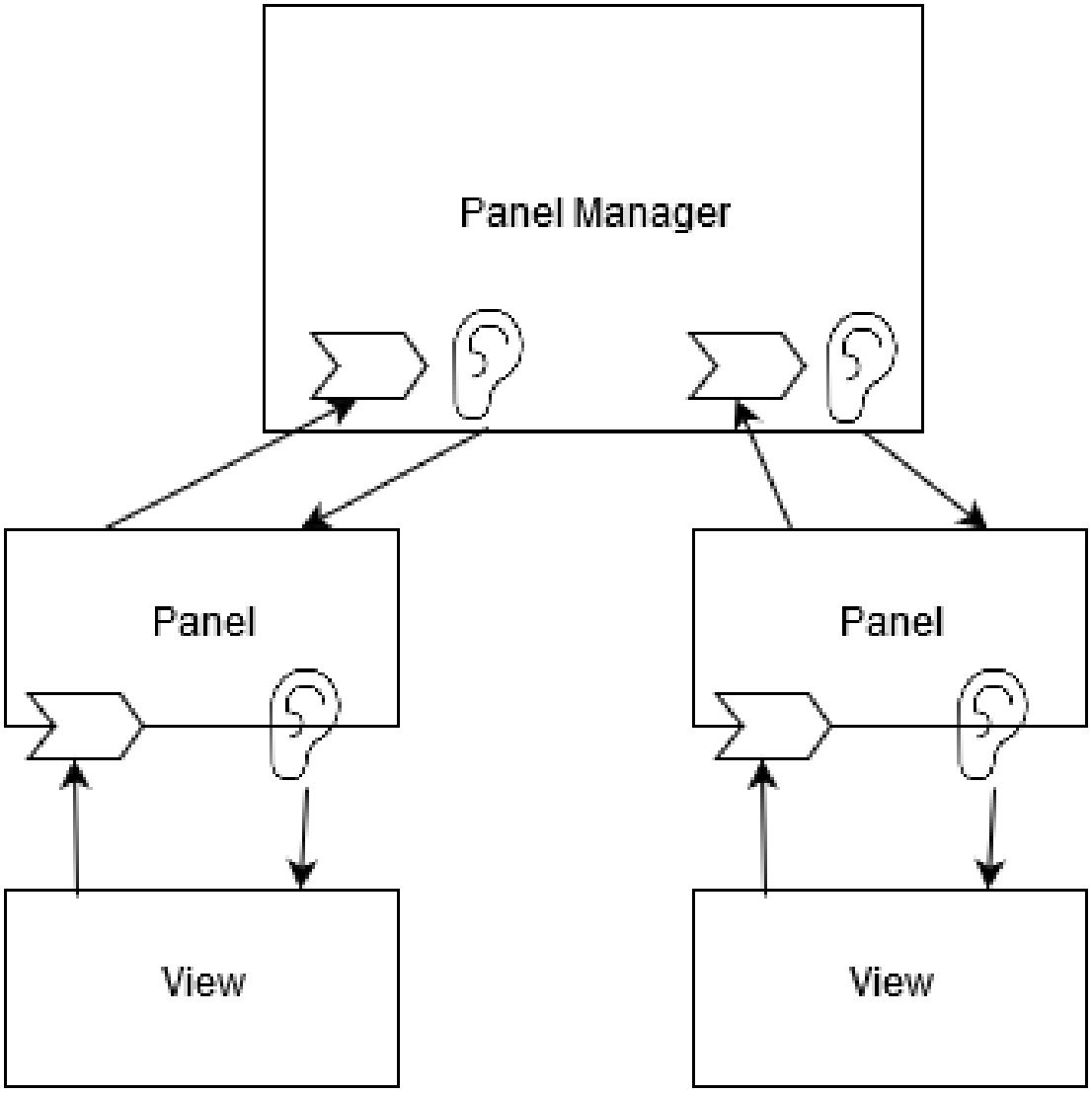


Abbildung 11: Interne Kommunikation Panel-View Struktur

3.3.5. Gizmos

Als *Gizmos* bezeichnet man kleine Steuerelemente, mit denen Nutzer meist per Mausklick interagieren können um somit Objekte manipulieren.
Im Kontext des Editors werden alle Hilfselemente, die bei der Steuerung von Kamera und Perspektive des Renderfensters unterstützen, als Gizmos bezeichnet. Diese Definition fasst allerdings auch andere Hilfselemente ein, die bei der Entwicklung eines Spiels den Nutzer unterstützen können.

Gizmos besitzen eine geringere Priorität für die Zwecke dieser Arbeit.

4. Umsetzung

4.1. User Interface

Es folgt eine Spezifikation der in FUDGE UI enthaltenen Schnittstellen.
Dieses Kapitel geht auf den jeweilig vorgesehenen Zweck einer Klasse bzw. Methode ein und beschreibt in Kürze, welche Funktionalitäten enthalten sind.
Anschließend werden die Klassen und Methoden-Spezifikationen gelistet.
Dies erfolgt entweder als beschreibende Spezifikation, oder als grafische Abbildung der Funktionalitäten einer Klasse.

4.1.1. UIGenerator

Die Aufgabe der `UIGenerator` Klasse besteht aus der komfortablen Generierung von User Interface Elementen.
Es ist eine statische Klasse, die hauptsächlich Komfortfunktionen für die Erstellung von typischen Elementen beherbergt. Zusätzlich verfügt diese Klasse über eine Methode, die aus Mutables automatisch User Interfaces generiert. Diese bildet die in Kapitel *3.2.1 Automatisch generierte Userinterfaces* beschriebene Funktionalität ab.

`public static CreateFromMutable(_mutable: ƒ.Mutable, _name?: string, _mutator?: ƒ.Mutator):HTMLElement`	
Verhalten	Erzeugt ein User Interface anhand der Daten des gegebenen Mutables.
Anmerkungen	Benutzt *createFromMutator(...)* für Rekursion. Der optional übergebene Mutator dient zur Optimierung der Performanz der Methode und wird in den meisten Anwendungsfällen nicht benötigt.

`private static createFromMutator(_mutator: ƒ.Mutator, _mutatorTypes: ƒ.MutatorAttributeTypes, _mutable: ƒ.Mutable): HTMLElement`	
Verhalten	Erzeugt ein User Interface anhand des gegebenen Mutators und des gegebenen mutatorTypes.
Anmerkungen	Erfordert einen passenden Mutable. Diese Methode fungiert primär für Rekursion und sollte von Anwendern nicht aufgerufen werden können.

`public static createDropdown(_id: string, _content: Object, _value: string, _cssClass?: string): HTMLSelectElement`	
Verhalten	Erzeugt eine Selection Box mit den Elementen des übergebenen Objects `_content` als Einträgen. `_value` gibt hier den ausgewählten Eintrag.
Anmerkungen	Der Name wurde gewählt, da er als intuitiver empfunden wurde. In `_content` wird ein mit Strings befülltes Element erwartet.

`public static createFieldset(_legend: string, _cssClass?: string): HTMLFieldSetElement`	
Verhalten	Erzeugt ein Fieldset mit einem Legenden Element
Anmerkungen	

`public static createFoldableFieldset(_legend: string): HTMLFieldSetElement`	
Verhalten	Erzeugt ein einklappbares Fieldset mit einem Legenden Element
Anmerkungen	

`public static createLabelElement(_id: string, params: { _value?: string, _cssClass?: string } = {}): HTMLElement`	
Verhalten	Erzeugt ein Label Element. Verwendet `_id` als Text wenn kein Wert in `_value` übergeben wird.
Anmerkungen	Dient hauptsächlich als Komfortfunktion für das Bauen von automatisch generierten Interfaces, es erschien allerdings sinnvoll Nutzern ebenfalls Zugriff zu der Methode zu geben.

`public static createFoldableFieldset(_legend: string): HTMLFieldSetElement`	
Verhalten	Erzeugt ein einklappbares Fieldset mit einem Legenden Element
Anmerkungen	

`public static createCheckboxElement(_id: string, _value: boolean, _cssClass?: string): HTMLInputElement`	
Verhalten	Erzeugt ein Checkbox Element
Anmerkungen	

`public static createStepperElement(_id: string, params: { _value?: number, _min?: number, _max?: number, _cssClass?: string } = {}): HTMLSpanElement`	
Verhalten	Erzeugt ein Zahleingabefeld mit den eingegebenen Parametern
Anmerkungen	

4.1.2. GenerateFromMutable

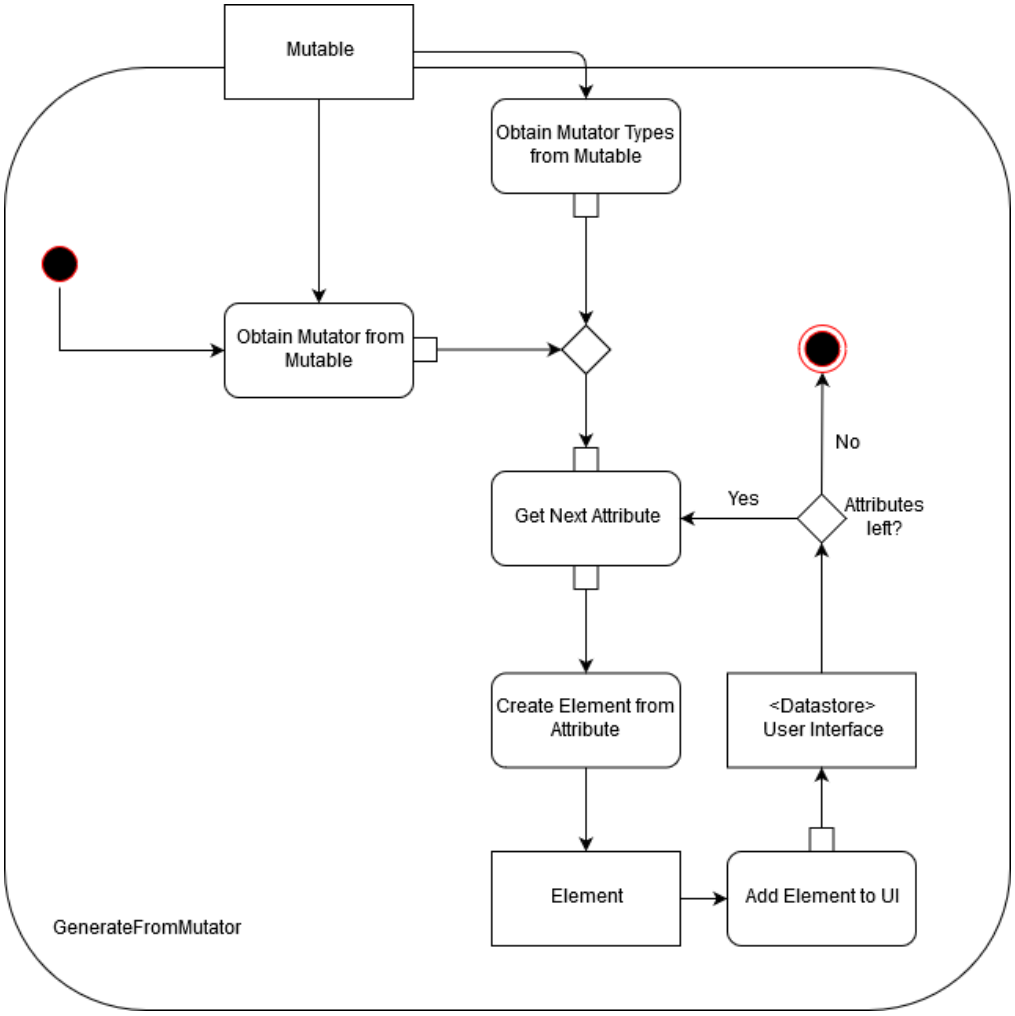


Abbildung 12: Aktivität „Generate From Mutable“

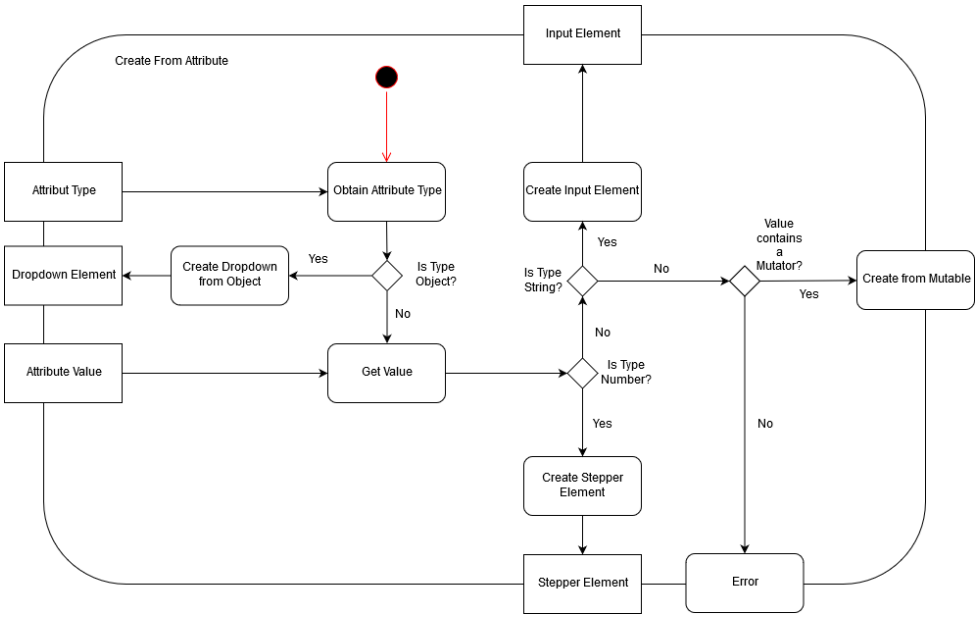


Abbildung 13: Aktivität „Create From Attribute“

Die in Abbildung 12 und Abbildung 13 gezeigten Aktivitätsdiagramme beschreiben das Verhalten der automatischen Generierung eines User Interfaces aus einem Mutable heraus.

Jedes Attribut, dass im Mutator enthalten ist wird zu einem User Interface Element verarbeitet. Hierbei wird das *MutatorTypes* Objekt benutzt, um fest zu stellen, welche Elemente generiert werden müssen.
Im Falle einer Mutator-Verschachtelung wird das Attribut am Mutable als neues (ein sogenanntes SubMutable) interpretiert und der Generierungsprozess wird Rekursiv ausgeführt.
Um Rechenzeit zu sparen wird das Attribut (in diesem Fall ein Mutator) mit übergeben, so dass kein Mutator vom SubMutable generiert werden muss.
Dropdown (auch Selectboxen genannt) bilden einen Sonderfall, da die Typdaten verwendet werden um den Inhalt des Dropdowns zu generieren.

4.1.3. UIMutable

`UIMutable` ist eine abstrakte Klasse, die als Bauplan für eine Implementierung der in Kapitel *3.2.2 Automatische aktualisierendes Userinterface* besprochenen Funktionalität.
Die Klasse beinhaltet alle notwendigen Funktionen zur Ausführung der besprochenen Funktionalität, allerdings wird die genaue Implementation dem Anwender überlassen. Es reicht für eine Implementation allerdings auch vollkommen aus, ein User Interface zu generieren. Nutzer müssen lediglich sicherstellen, dass Eingabefelder, die Attribute des Mutables abbilden, den Attributnamen im *id*-Attribut des Elements führen.
Das *id*-Attribut des Elements dient zur eindeutigen Identifikation. Es wurde gewählt, da es die einfachste Art der eindeutigen Identifikation ist, die Funktion kann von fortgeschrittenen Nutzern in der Implementation überschrieben werden, wodurch andere Identifikationsmethoden verwendet werden können.
Es wird eine abstrakte Klasse verwendet, damit Nutzer keine Instanzen dieser Klasse anlegen können. Es kommuniziert, dass eine eigene Implementation erwartet wird. Dieses Vorgehen dient dazu, Nutzern mehr Freiheit zu geben.
Es ist allerdings keine rein abstrakte Klasse, wodurch die gewünschte Funktionalität einfach geerbt werden kann.
Eine Standard-Implementation ist geboten in `UINodeData`.

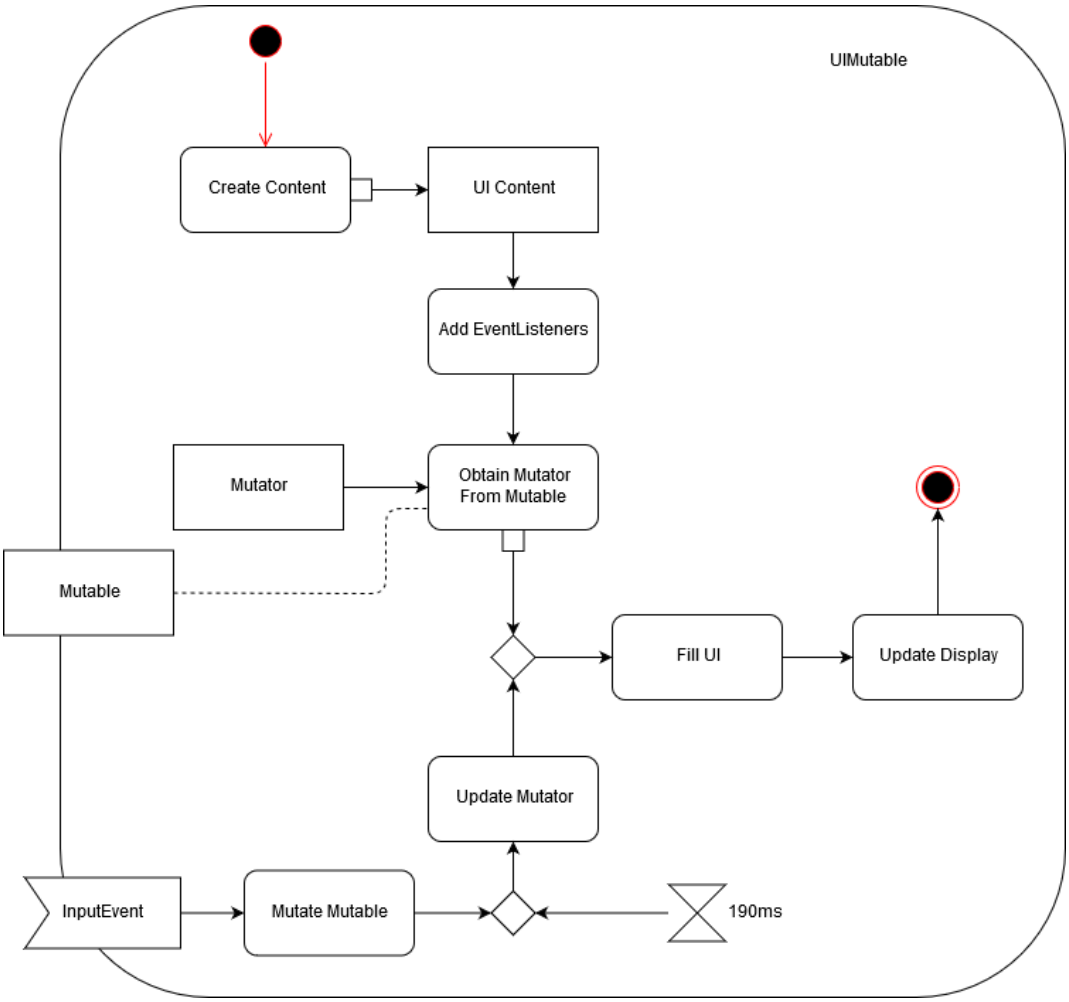


Abbildung 14: Aktivität „UIMutable“

4.1.4. UIElements

UIElements enthält eine Sammlung an Custom HTMLElements, welche die Funktionsweise bestehender HTMLElemente erweitert.
Alle beschriebenen Elemente können auch durch HTML-Tags erstellt werden.

ToggleButtons sind Buttons, die visuell ihren Zustand anzeigen können. Sie funktionieren hierbei ähnlich wie Checkbox-Elemente, welche ihren Zustand als *checked* oder *unchecked* darstellen können.
Sie sind gedacht für Fälle, in denen ein Nutzer eine Funktionalität ein- und abschalten kann. Dies schließt zum Beispiel auch Fälle ein, in denen Content ein- und ausgeklappt werden kann.

`export class ToggleButton extends HTMLButtonElement`	
Verhalten	Ein Button Element, das seinen Zustand speichert. Dieser Zustand wird als boolean intern gespeichert und kann gesetzt werden. Das Element kann per CSS gestyled werden.
Anmerkungen	Der ToggleButton wird in CSS unterschieden durch die Klassen *ToggleOn* und *ToggleOff*

`public constructor(style: string)`	
Verhalten	Erzeugt einen neuen ToggleButton, fügt den übergebenen *stlye* Parameter zu den CSS Klassen des Buttons hinzu.
Anmerkungen	Style kann verwendet werden, um den Button per CSS anzusprechen

`public setToggleState(toggleState: boolean): void`	
Verhalten	Verändert den Togglezustand des Buttons auf den übergebenen Wert
Anmerkungen	Kann verwendet werden, wenn ein ToggleButton programmatisch ausgelöst werden soll.

`public getToggleState(): boolean`	
Verhalten	Gibt Togglezustand zurück
Anmerkungen	

`public toggle(): void`	
Verhalten	Verändert den Togglezustand.
Anmerkungen	Simuliert quasi ein Auslösen des Buttons.

Stepper sind Eingabefelder (`HTMLInputElement`), welche nur Zahlen als gültige Eingaben akzeptieren.
Die Funktion, Eingabefelder für Zahlen zu erstellen ist nativ gegeben als Subfunktion des HTMLInputElements, allerdings erfordert es mehrere Schritte. Das Stepper Objekt dient als Komfortobjekt zur schnellen Erstellung solcher Eingabefelder.
Der Name Stepper wurde gewählt, da er als intuitiver empfunden wurde als *Number Input Element*.

`export class Stepper extends HTMLInputElement`	
Verhalten	Ein Zahlen-Eingabefeld.
Anmerkungen	Darstellungsform ändert sich und zeigt Buttons an der Seite des Eingabefelds an.

`public constructor(_name: string, params: { _min?: number, _max?: number, _step?: number, _value?: number } = {})`	
Verhalten	Erzeugt ein neues Stepper Element. Es können als zusätzliche Parameter übergeben werden, welche die Eigenschaften des Eingabefelds bestimmen (Minimalwert, Maximalwert, Inkrement, Startwert).
Anmerkungen	

FoldableFieldsets sind eine Sonderform von Fieldsets, die es erlauben ihren Inhalt zu verstecken. Hierbei wird ein ToggleButton verwendet, um dieses Verhalten zu ermöglichen.
FoldableFieldsets dienen zur Organisation von User Interface Elementen. Sie werden bei der automatischen Generierung von User Interface Elementen (siehe *4.1.1. UIGenerator*) verwendet, um die Attribute des Mutables in einer organisierten Form darzustellen.
FoldableFieldsets sind (genau wie Fieldsets) verschachtelbar, wodurch eine hierarchische Darstellung von Daten gebildet werden kann, was sie für die Darstellung von Objektdaten besonders interessant macht.
Der Name wurde gewählt, um das Verhalten sprachlich darzustellen: es ist ein Fieldset, welches sich *ein- und ausfalten* lässt.

Abbildung 15: Beispiel „Foldable Fieldset“

Der verwendete ToggleButton erhält standardmäßig eine feste CSS Klasse, um das Falt-Verhalten konsistent für alle FoldableFieldsets darzustellen. Nutzer erhalten durch einen optionalen Parameter allerdings die Möglichkeit, den Stil des Buttons nach Belieben zu verändern (für den Fall, dass mehrere FoldableFieldsets auftauchen sollten, die verschiedene Styles verwenden)

`export class FoldableFieldSet extends HTMLFieldSetElement`	
Verhalten	Ein ein- und ausklappbares Fieldset
Anmerkungen	Kann verschachtelt werden.

`public constructor(_legend: string, _buttonStyle?: string)`	
Verhalten	Erzeugt ein neues ein- und ausklappbares Fieldset
Anmerkungen	Der _legend Parameter gibt den Titel des Fieldsets an, der im Legenden Element angezeigt wird. _buttonStyle dient zur Stilisierung des zugehörigen ToggleButtons.

DropMenus sind Menüs, die sich auf Knopfdruck ausklappen lassen.
Sie sind konzipiert worden aus der Notwendigkeit heraus, Untermenüs im User Interface des Editors zu besitzen.
Die Menüpunkte werden aus einem Mutator gelesen und erzeugt. Jeder dieser Menüpunkte bekommt eine Signatur anhand des Untermenüs, in dem es sich zugewiesen. Welcher Menüpunkt ausgewählt wurde, kann dann anschließend mit Hilfe eines EventListeners ermittelt werden. (Mehr hierzu in Kapitel *4.1.6. UIEvent*)
Die Erzeugung des Mutators kann händisch vom Nutzer erfolgen, es ist allerdings auch möglich ein Menü aus einer Reihe an Strings zu generieren durch eine Komfortfunktion. Die statische Methode buildFromSignature(...) ermöglicht es, aus einem in Punkt-Notation geschriebenen String einen passenden Mutator anzufertigen. Eine passende Punktnotation sieht beispielhaft wie folgt aus:

„`Obermenü.Untermenü.Menüpunkt`“

Abbildung 16: Codesnippet „Beispielsignatur“

Die Funktion würde aus diesem String folgenden Mutator geben:

```
{ Obermenü: {
        Untermenü: {
            Menüpunkt: „Menüpunkt“
        }
}
```

Abbildung 17: Codesnippet „Beispielmutator“

(Dies ist die benötigte Mutatorstruktur).
Zusätzlich zum DropMenu existieren noch DropButton und DropContent. Diese Elemente bilden jeweils den Button zum Ausklappen des Menüs und den Container für die Menüpunkte. Diese werden nicht exportiert, da sie nur für

die Verwendung innerhalb der DropMenu Struktur verwendet werden sollen. Sie sind somit effektiv nicht zugänglich für die Nutzer.
Grund hierfür ist, dass sie zusätzliche Logik besitzen die für die Funktionalität von DropMenu wichtig ist, für Nutzer allerdings gänzlich uninteressant da sie nur für diesen einen Zweck konzipiert wurden.

`export class DropMenu extends HTMLDivElement`	
Verhalten	Ein Dropdown Menü, das über einen Button aufgeklappt wird.
Anmerkungen	Besitzt zwei Untertypen (DropButton und DropContent), die dem Nutzer nicht zugängig gemacht werden, da sie nur in Kombination mit diesem Typ verwendet werden sollten.

`public constructor(_name: string, _contentList: ƒ.Mutator, params: { _parentSignature?: string, _text?: string })`	
Verhalten	Erzeugt ein neues DropMenu
Anmerkungen	Die optionalen Parameter werden hauptsächlich für das rekursive Verhalten der Methode verwendet.

`public static buildFromSignature(_signature: string, _mutator?: ƒ.Mutator): ƒ.Mutator`	
Verhalten	Konstruiert einen Mutator in der für DropMenu benötigten Form aus einem String in Punkt-Notation
Anmerkungen	Statische Methode. Der optionale _mutator Parameter der Methode wird hauptsächlich für Rekursion verwendet, kann aber auch benutzt werden, um zusätzliche Menüpunkte an einen bereits existierenden Mutator hinzuzufügen.

4.1.5. UIListElements

UIListElements enthält eine Sammlung an Klassen, welche der Darstellung von hierarchischen Listenstrukturen dienen.
Primär sollen FUDGE Node-Strukturen abgebildet werden in einer sogenannten *CollapseableList*. Die Idee hinter CollapseableLists ist, dass die Darstellung einer Datenstruktur unabhängig von dieser ist und nur die nötigsten Informationen anzeigt. Darstellungsdaten bleiben nur erhalten, solange sie sichtbar sind und werden verworfen, sobald sie nicht mehr gezeigt werden.

Diese Herangehensweise sollte primär Speicherplatz sparen und das Anzeigen der Daten performanter machen.
Es fielen jedoch Probleme mit diesem Verfahren während der Entwicklung des Editors auf, weshalb ein anderes Verfahren gewählt wurde.
Statt die Daten immer zu verwerfen, sobald sie nicht angezeigt werden, sollen Daten gespeichert bleiben, bis sie erneuert werden. Dieses Vorgehen war sehr ähnlich zum *Falt*-Verhalten der FoldableFieldsets.
Um die Änderung im Vorgehen zu reflektieren wurden die *CollapseableLists* in *FoldableLists* umbenannt.
Zur Abbildung der beschriebenen Funktionalität soll ein *Model View Controller*-Designpattern verwendet werden.

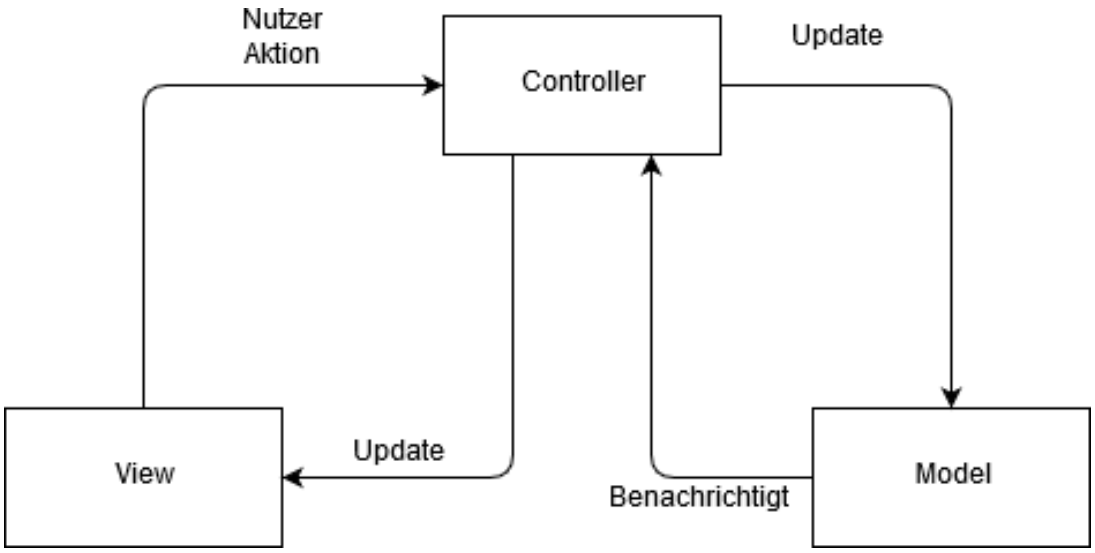


Abbildung 18: Model View Controller Pattern

Das Pattern ist verwand mit dem in Kapitel 1.3.4 vorgestellten Seeheim-Modell und basiert auf dem selben Prinzip der Trennung zwischen Anzeige, Anwendungslogik und Datenmodell.

Die Darstellung einer Datenstruktur ist sehr von selbiger abhängig. Aus diesem Grund wird die Implementation des Modells den Anwendern überlassen. Die Klassen, welche die Grundstruktur der Listen-Elemente bilden, sind abstrakt und können nicht instanziiert werden.
Der Listen-Controller muss der Datenstruktur entsprechend programmiert werden.

Für Animationsdaten und FUDGE Nodes existieren Implementationen des Modells (jeweils in UIAnimationList und UINodeList).

4.1.6. UIEvent

UIEvents beinhaltet ENUMs, welche die Namen der im Framework aufgerufenen Events beschreibt. Die meisten dieser Events sind zur Zeit spezifisch für das Editor Interface.
EventListener können auf die hier beschriebenen Events gelegt werden, um Funktionalitäten des Frameworks zu nutzen.
Die ENUMs sind benannt nach dem Betreff des Events. Events, die FUDGE Nodes betreffen, sind zum Beispiel in *EVENTNODE* zu finden.
Diese Form wurde gewählt, um das Finden von spezifischen Events einfacher zu gestalten. Ursprünglich waren alle Eventaufrufe in einem ENUM namens *UIEVENT* vereint, durch die Benennung wurde es jedoch schnell unhandlich. Durch die Trennung der Events in verschiedene ENUMs existieren allerdings nun ENUMs, die nur einen Wert besitzen, welche auch als Konstante realisiert werden könnten.
Es wurde sich für ENUMs entschieden, um spätere Erweiterungen des Frameworks einfacher zu gestalten.
Es folgt eine kurze Beschreibung der Events, welchen Zweck sie erfüllen und welche Daten mit dem entsprechenden Event übermittelt werden sollen.

EVENTNODE enthält Events, die FUDGE Node betreffen. Sie werden hauptsächlich in FoldableLists verwendet im Editor.

- **SELECTION** signalisiert die Auswahl eines Nodes. Es wird der selektierte Node übergeben.
- **REMOVE** signalisiert das Entfernen eines Nodes. Es wird der zu entfernende Node übergeben.
- **HIDE** signalisiert das Ausblenden eines Nodes. Es wird der auszublendende Node übergeben.

EVENTVIEWPORT enthält Events für die Interaktion mit Viewports. Diese Events sind hauptsächlich für den Editor relevant.

- **ACTIVEVIEWPORT** signalisiert eine Änderung des derzeit aktiven Viewports. Dieses Event wird im Editor für die Kamerasteuerung verwendet. Es wird der aktive Viewport übergeben.

EVENTLIST enthält Events für die Interaktion mit FoldableList.

- **FOLD** signalisiert, dass ein Listenelement eingefaltet werden soll. Es wird das Listenelement übermittelt.

EVENTMUTATOR enthält Events für die generelle Interaktion mit Mutables und Mutatoren.

- **UPDATE** signalisiert, dass Mutatordaten aktualisiert werden sollen. Es wird der Mutator übergeben.
  Dieses Event ist für AnimationLists wichtig.

EVENTDROPMENU enthält Events für die Interaktion mit DropMenues.

- CLICK signalisiert die Auswahl eines Menüpunkts. Es wird die Signatur des Menüpunkts übergeben.
- COLLAPSE signalisiert das Einklappen des Menüs.

4.2. Editor

4.2.1. Multi-Window Lösung

Wie in Kapitel *3.3.2. Fenster-basiert gegen Tab-basiert* beschrieben und erklärt, wurde eine “Fenster-basierte“ Lösung bevorzugt.
Der Ansatz sieht vor, dass Kernfelder des Editors als eigener „Sub-Editor“ implementiert werden. Diese sollen in eigenen *Electron-Fenster* angezeigt werden.
Durch die Funktionsweise von Electron ist es möglich, mehrere Desktop-Fenster innerhalb einer Electron-Instanz zu erzeugen.
Für die Kommunikation zwischen diesen Fenstern muss Electrons interne Kommunikationsschnittstelle *IPC* verwendet werden.
Während der Prototypisierung dieser Umsetzung fiel auf, dass durch IPC versendete Objekte ihren Objekt-Bezug verloren.
Der Auslöser hierfür war, dass Objekte, die mit IPC versendet werden, serialisiert und deserialisiert werden. Bei diesem Deserialisierungs-Prozess werden übermittelte Objekte als Plain Javascript Objects rekonstruiert, wodurch jegliche Objekt-Referenz verloren geht.
Es wurde diskutiert, ob es sinnvoll erscheint, einen Workaround für dieses Problem zu finden, allerdings wurde sich schnell dagegen entschieden.
Zusätzlich wurde entdeckt, dass die *Pop-out Window* Funktionalität von Golden Layout, welche es erlaubt Unterfenster aus dem Layout

herauszulösen und als eigene Fenster zu öffnen, nicht korrekt in Electron funktioniert. Hierfür konnte kein Workaround gefunden werden, somit musste auf diese Funktionalität ebenfalls verzichtet werden.
Die *Tab-basierte* Lösung soll als Ausweichlösung dienen.

4.2.2. Panel-View Struktur

Die in Kapitel *3.3.3. Struktur* beschriebene Panel-View Struktur wurde entwickelt, um Inhalte effektiv organisieren zu können.

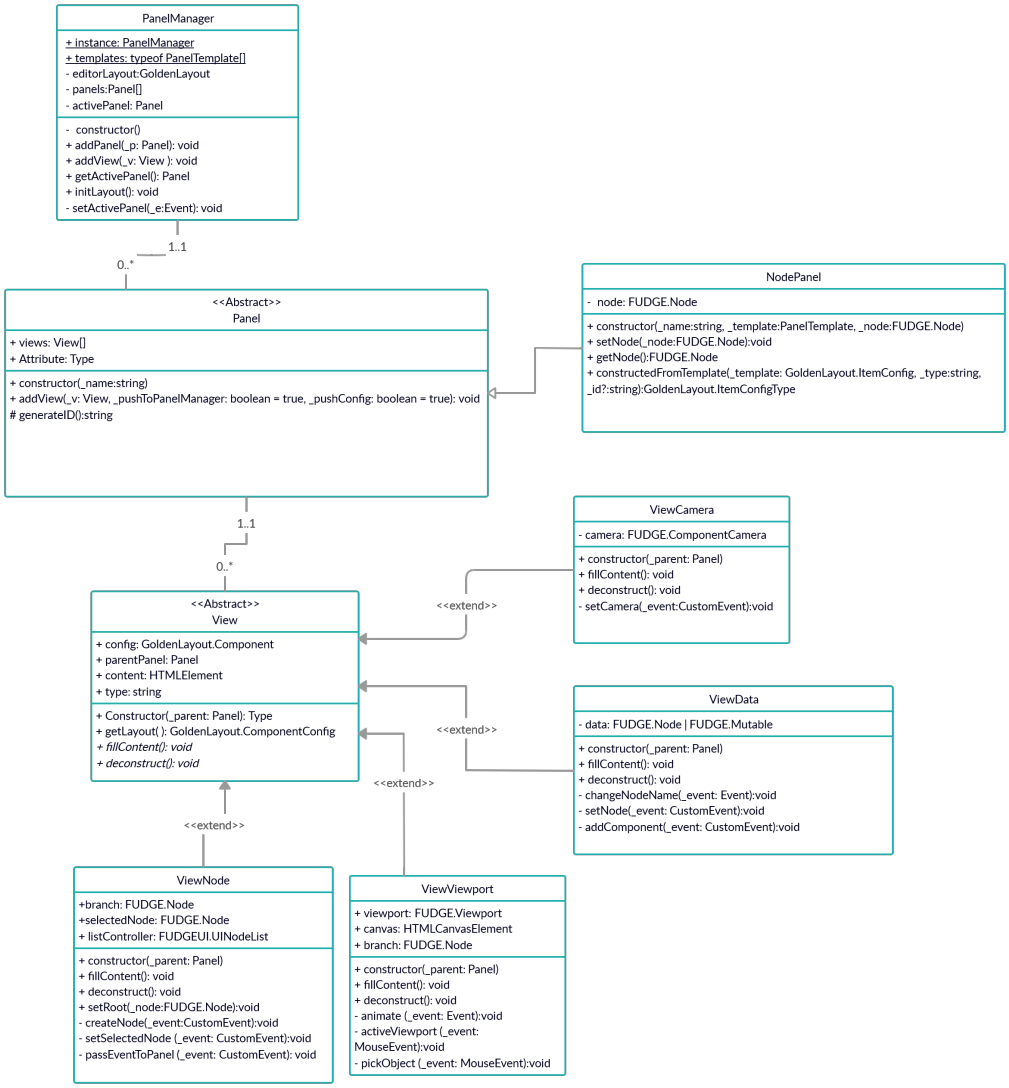


Abbildung 19: Klassendiagramm „FUDGE Editor“

Die gezeigte Struktur kann grafisch folgendermaßen verstanden werden.

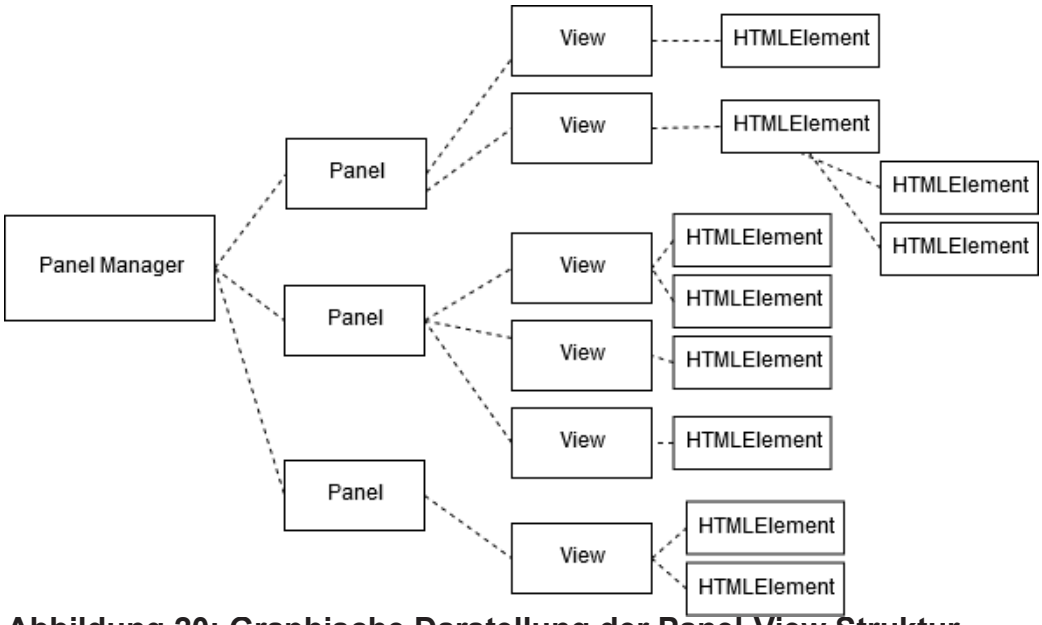


Abbildung 20: Graphische Darstellung der Panel-View Struktur

Das tatsächliche User Interface wird aus HTML Elementen gebildet und besitzt seine eigene DOM-Baumstruktur. Logisch wird diese Baumstruktur aufgespalten durch die hier dargestellte Datenstruktur.
Views besitzen *en zu ihren zugehörigen Elementen und verwalten diese. Views besitzen ebenfalls einen Großteil der notwendigen Logik, um den Datenaustausch zwischen User Interface und dem FUDGE Kern zu handhaben. Ebenfalls besitzen sie die nötige Logik, um ihren Inhalt zu generieren.
Die Klasse View ist abstrakt, da die Implementation abhängig vom Zweck der jeweiligen View ist. `ViewNode` besitzt z.B. andere Anforderungen als `View-Viewport`.

Views werden durch ihre zugehörigen Panels verwaltet. Das Panel beinhaltet hierbei die Layout Struktur, in das die ihm zugehörigen Panels eingeordnet werden. Zwar besitzen Views eine Referenz ihres eigenen Layouts, allerdings wird dieses ihnen im Regelfall vom Panel generiert und gegeben.
Die Panel Klasse ist ebenfalls abstrakt.
Ursprünglich war dem nicht so, da es vorgesehen war, dass einzig und allein Views Informationen über ihren Editor-Kontext besitzen (welcher Node bearbeitet wird, z.B.). Panels verfügten über keine Anwendungsfall-Spezifische Logik. Dies entpuppte sich schnell als Schwäche im Design. Der fehlende Kontext an den Panel machte das hinzufügen neuer Views zu diesen sehr schwierig und intuitiv.

Es war zudem nicht ersichtlich, welchen Zweck ein spezifisches Panel erfüllen sollte und somit war die gewünschte klare Trennung zwischen den Aufgabenfeldern durch die Panels nicht gegeben.
Das Problem wurde gelöst, indem Panels als abstrakte Klasse definiert wurden und somit Implementationen für spezifische Aufgabenfelder benötigt werden.
Panels erhalten damit zusätzliche Logik, die das Halten des Editor-Kontextes erlauben.
Panels werden am `PanelManager` registriert. Die von den Panels gehaltene Layout Struktur wird vom `PanelManager` zusammengeführt und bilden das Layout, das von Golden Layout verwendet wird zur Strukturierung der Inhalte.

`PanelManager` bildet ein Singleton Design Pattern ab, es kann also nur eine Instanz der Klasse existieren. Der Zweck des `PanelManager` ist die Verwaltung des Editor User Interfaces, deshalb erschien es vernünftig genau eine Instanz einer solchen Klasse zu besitzen. Auf diesem Weg ist die Beziehung zwischen dem User Interface und dem `PanelManager` klar gegeben.
Der `PanelManager` verfügt zusätzlich aus eine Sammlung an sogenannten *PanelTemplates*. Diese sind Layout Konfigurationsobjekte, welche das gewünschte Layout des Panels darstellen.
Panels können ihre Konfiguration aus diesem Template erzeugen.

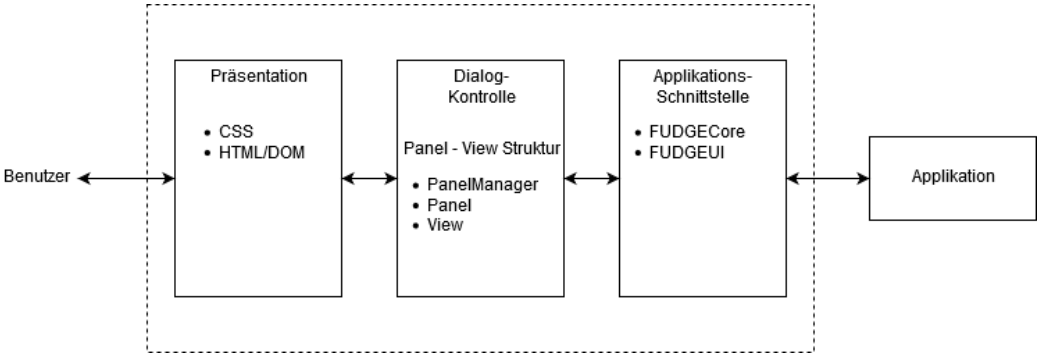


Abbildung 21: Seeheim-Modells „FUDGE Editor“

Im Seeheim-Modell (siehe *1.3.4. Beschreibung einer grafischen Benutzerschnittstelle*) gliedert sich das beschriebene Modell in der Dialogkontroll-Schicht ein. Der FUDGE Kern und das FUDGE UI Framework würden hierbei die Applikationsschnittstelle bilden, während HTML und CSS sich in

der Präsentation-Schicht ansiedeln.

4.2.3. PanelNode

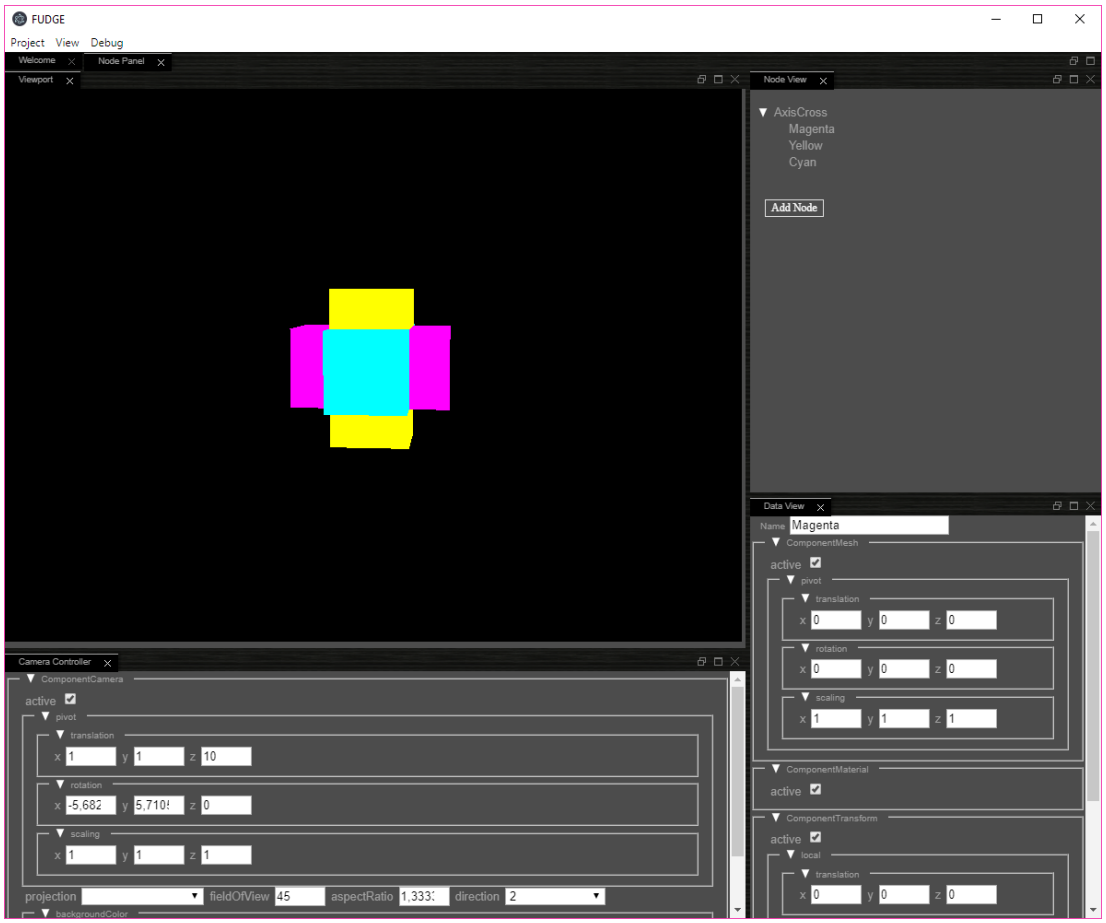


Abbildung 22: Screenshot „Node Panel“-Prototyp

Das Node Panel bildet das Aufgabenfeld der FUDGE Node Manipulation ab, das bedeutet, mit diesem Panel sollen Nutzer Node Strukturen erstellen und verändern können.
Im Folgenden werden die Bestandteile des Node Panels näher betrachtet.

`ViewViewport` (Viewport) enthält alle Funktionalitäten, welche erforderlich sind um Node Strukturen im dreidimensionalen Raum darzustellen.
Der Viewport rendert neue Bilder jeden Frame.

`ViewCamera` (Camera Controller) bildet eine primitive Kamerasteuerung ab. Aus der ComponentCamera des aktiven Viewports wird ein User Interface automatisch generiert. Diese View verwendet die Klasse `UINodeData` als Implementation von `UIMutable`.

`ViewNode` (Node View) verwendet Foldable Lists um die Node Struktur an-

zuzeigen, die vom Panel bearbeitet wird. Es ist zukünftig vorgesehen, von dieser Ansicht Knoten direkt ausblenden und entfernen zu können. Zusätzlich existiert ein DropMenu, mit dem neue Knoten hinzugefügt werden können.

`ViewData` (Data View) stellt die Attribute der Componenten am aktuell ausgewählten Knoten dar. Jede Komponente am selektierten Knoten wird von einer Instanz von `UINodeData` abgebildet.

5. Fazit und Ausblick

5.1. Fazit

Während der Projektlaufzeit wurde das angestrebte Projektziel eines vollständigen Editors und eines möglichst “Feature-complete” User Interface Frameworks nicht vollständig erreicht, es wurde allerdings ein Grundstein für zukünftige Entwicklungen an FUDGE gelegt.
Der Editor hatte von Beginn des Projekts ab die höchste Priorität, somit wurden viele Teile von FUDGE UI mit einem besonderen Augenmerk auf die Entwicklung des Editors konzipiert und entwickelt.
Ein erster lauffähiger Editor Prototyp entstand am Ende der Projektzeit, welcher das NodePanel abbilden kann. Dieses Panel ist zum Großteil funktionsfähig. Es können Node Strukturen erstellt, dargestellt, geändert, gespeichert und geladen werden.
Es fehlen die Umsetzung einer direkten Kamera-Steuerung, Gizmos, der Projekt Explorer und die Selektion von Nodes über den Viewport.

5.2. Ausblick

Mit Fertigstellung des Projektes folgt die Erweiterung der Funktionalitäten von FUDGE.
FUDGE als solches steckt noch in den Kinderschuhen und wird zum ersten Mal in Veranstaltungen während des Wintersemesters 2019/2020 eingesetzt.

Der Funktionsumfang von FUDGE soll unter anderem in naher Zukunft um eine Physik-Engine und ein Sound System erweitert werden. Zusätzlich soll eine rudimentäre 3D Modellierungs Funktionalität hinzugefügt werden, welche ihr eigenes Panel besitzen soll.
Zudem soll das Animation Panel implementiert werden.
Hierfür muss der Editor um zahlreiche Funktionen erweitert werden.

Es ist abzusehen, dass in näherer Zukunft der Funktionsumfang von FUDGE stark erweitert wird.

Literatur

[1] Prof. J.Dell O'ro-Friedl, "FUDGE Wiki" [Online], Feb. 2019,https://github.com/JirkaDellOro/FUDGE/wiki [Zugriff Nov 2019]

[2] Dr. J.Voss, Dr. D. Nentwig, Entwicklung von graphischen Benutzungsschnittstellen : Modelle, Techniken und Werkzeuge der User-Interface-Gestaltung, München; Wien; Hanser, 1998

[3] K. Cwalina, B. Abrams, Framework design guidelines : conventions, idioms, and patterns for Reusable .NET libraries, Upper Saddle River, N.J. London; Addison-Wesley, 2006

[4] Prof. J.Dell O'ro-Friedl, "FUDGE Core API"[Online], Nov 2019, https://jirkadelloro.github.io/FUDGE/Documentation/Reference/Core/ [Zugriff Nov 2019]

[5] Microsoft, "Typescript" [Online], 2019, https://www.typescriptlang.org/ [Zugriff Nov 2019]

[6] W3C DOM Working Group, "Document Object Model (DOM)" [Online], 2009, https://www.w3.org/DOM/ [Zugriff Nov 2019]

[7] deepstreamHub GmbH, "Golden Layout" [Online], 2019, http://golden-layout.com/ [Zugriff Nov 2019]

[8] GitHub Inc., "Electronjs" [Online], 2019, https://electronjs.org/ [Zugriff Nov 2019]

[9] L. Stegk, „UX-Design und Entwicklung der UI für einen Spieleeditor und Optimierung der Userexperience", Bachelor Thesis, HFU Furtwangen, Furtwangen, 2019

[10] PhosphorJS, "PhosphorJS" [Online], 2019, http://phosphorjs.github.io/about.html [Zugriff Nov 2019]

[11] Jeff Houde, "wcDocker" [Online], 2016, http://docker.webcabin.org/ [Zugriff Nov 2019]

[12] Metafizzy, "Isotope" [Online], 2019, https://isotope.metafizzy.co/ [Zugriff Nov 2019]

[13] David DeSandro, "Masonry" [Online] 2019, https://masonry.desandro.com/ [Zugriff Nov 2019]

[14] Sencha Inc., "ExtJS" [Online], 2019, https://www.sencha.com/products/extjs/ [Zugriff Nov 2019]

[15] Human Benchmark,"Human Benchmark - Statistics" [Online], 2019, https://www.humanbenchmark.com/tests/reactiontime/statistics [Zugriff Nov 2019]

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Thesis selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel sind angegeben.

_____ Furtwangen, den 02.12.2019