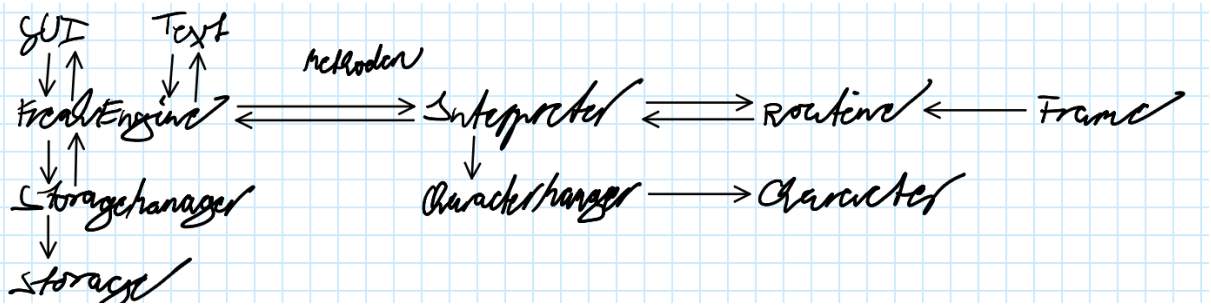
# *Projektdokumentation für A666*

Dies Dokument listet die Aufteilung der Projektarbeit und die Funktionsweise unterliegender Systeme in detaillierter Form auf. Für genauere Informationen über die einzelnen Klassen sollte stattdessen die API-Dokumentation zur Hilfe genommen werden.

## *Frühe Entwicklung*

Wir haben keine Vorgaben über die Inhalte und Funktionsweise des Spieles erhalten. Dies veranlasste und dazu, ein intensives Brainstorming zu betreiben. Schlussendlich landete die Wahl auf einer Idee: einem Text Adventure

Mit dieser Idee im Kopf fuhren wir fort. Erste Designpläne wurden entwickelt.




Noch sehr rudimentär lässt sich hier jedoch bereits die grobe Struktur des Programms erkennen. Eine Engine-Klasse im Zentrum, andere Helferklassen wie *Interpreter* und *StorageManager* die Funktionen zur Engine bereitstellen. Bei der Engine soll demnach alles zusammenlaufen.

An diesem Aufbau wird weitgehend festgehalten.

Der Name *FreakEngine* ist hier zusammengesetzt aus dem Paketnamen (intern „com.freakurl.engine“) und dem technischen Begriff „Engine“. Für einen Paketnamen wird von der Java Dokumentation ein eindeutiger Identifier empfohlen. Typischerweise wird hierfür eine umgekehrte Domain benutzt, deshalb wurde zu Beginn des Projekts das bereits im Besitz eines Projektmitglieds stehende Domain für diesen eingesetzt.

Die anfängliche Bekanntmachung mit dem BlueJ-System führte dazu, dass das Team sicher mit Dokumentation, also *Javadoc*, und dem allgemeinen Aufbau der BlueJ „Entwicklerumgebung“ umgehen kann. Wissenslücken können durch eine Internetsuche geschlossen werden.

Leider ist nicht äußerst viel Entwicklungsrückblick erhalten geblieben. Deshalb ist es ratsam, sollte besonderes Interesse am Fortschritt bestehen, dass ein Blick auf die Commit-Historie geworfen wird.

# *Interpreter*

Anfangen sollte man vorzugsweise mit dem Anfang. Dieser bildet sich in diesem Projekt durch den *Interpreter* ab. Der *Interpreter* ist dafür verantwortlich, die Geschichte, welche in einer Datei im XML-Format gespeichert ist, auszulesen und sie in ein *Routine*-Objekt umzuwandeln. Dieses enthält einige Einstellungen und die *Frame*s.

### *Feldübersicht*

**Felder**

| Modifizierer und Typ | Feld | Beschreibung |
|---|---|---|
| `final List<Frame>` | `frames` | Alle verfügbaren Frames. |
| `final Optional<File>` | `sound` | Optionale globale Hintergrundmusik. |
| `final String` | `summary` | Eine Zusammenfassung des Games. |

Um die Spieldateien zu laden, besitzt die *Interpreter*-Klasse eine statische Methode *loadRoutine*. Diese gibt ein neues *Routine*-Objekt zurück, das die Daten enthält. Es wird lediglich ein Pfad zum Asset-Ordner übergeben, da durch die Methode automatisch die *main.xml* Datei geladen wird. Es wäre möglich, einen Dateinamen zu übergeben, dies ist aber nicht genutzt.

*FreakEngine* ruft diese Funktion auf und speichert den Wert ab. Mehr Details in *FreakEngine*.

### *Methodendetails*

#### loadRoutine

```
static Routine loadRoutine(String parent)
                    throws EngineException
```

Lade eine Routine mit dem Standarddateinamen.

**Parameter:**

`parent` - Ein absoluter Pfad zum übergeordneten Verzeichnis.

`name` - Dateiname mit Dateiendung im Verzeichnis `parent`.

**Gibt zurück:**

Die geparste `Routine` mit allen `Frame`s.

**Löst aus:**

`EngineException` - Wird ausgelöst, sollte der Ordner oder die Datei nicht existieren, die Datei nicht richtig formatiert sein oder diese einen nicht-evaluierbaren Wert enthalten. Siehe `Throwable.getMessage()` für mehr Details.

**Siehe auch:**

`loadRoutine(parent, name)`

Das Dateiformat, welches nötig ist, um die Geschichte korrekt auszulesen, ist in *Dateiformat* dokumentiert. Weitere Informationen über dieses dort.

# *Dateiformat*

Die Geschichte dieses Projekts, also der eigentliche Inhalt, ist in Form einer traditionellen XML-Datei gespeichert. Diese ist im Root-Verzeichnis des Asset-Ordners zu finden und ist *main.xml* genannt.

Die Hauptbestandteile sind der Doctype, die Syntax-Definition für Programme, um Fehler bereits in der IDE feststellen zu können, und ein routine-Tag. Zweiteres ist der eigentliche Inhalt.

Dieser hat weitere Unter-Tags, welche das tatsächliche Geschehen beeinflussen. Der Tag *summary* dient als Zusammenfassung des Dokuments, *sound* wird als Hintergrundmusik verwendet. *imports* ist genutzt, um andere Dokumente einzubinden. Dies ist nützlich, wenn man beispielsweise einen Geschichtsstrang auslagern möchte. *summary* und *sound* eines importierten Dokuments werden ignoriert. *characters* ist die Definition, welche an *CharacterManager* weitergegeben wird. Mehr dazu im Abschnitt *Charaktere*.

Der wichtigste Teil des routine-Tags ist der Unter-Tag *frames*. Dieser ist der Einzige, der zwingend vorhanden sein muss. Er enthält einige *frame*-Tags. Diese speichern die Informationen, die notwendig sind, um einen Frame anzuzeigen. Die ID muss einzigartig sein, da diese verwendet wird, um den Frame zu identifizieren, wie der Name schon sagt. Wenn eine doppelt vorkommt, wird ein Fehler geschmissen. Für weitere Felder sollte der Abschnitt *Frame* zur Hilfe genommen werden.

# *Frame*

Das *Frame*-Objekt ist dafür verantwortlich, die eingegebenen Dateien einzulesen, Pfade zu verifizieren und als einheitliche Plattform für die Renderer dienen. Das Objekt selbst hat keine Methoden und ist lediglich ein Container für Daten.

### *Feldübersicht*

**Felder**

| Modifizierer und Typ | Feld | Beschreibung |
|---|---|---|
| `final Optional<String>` | `author` | Der Autor dieses Frames. |
| `final Optional<String>` | `flag` | Eine Flag, welche nach dem Anzeigen dieses Frames in `FreakEngine` angestellt wird. |
| `final int` | `id` | Eine einzigartige ID, welche diesen Frame identifiziert. |
| `final Optional<File>` | `image` | Optional angezeigtes Bild. |
| `final List<FrameOption>` | `options` | Verfügbare Optionen. |
| `final Optional<File>` | `sound` | Optional abzuspielender Soundeffekt. |
| `final String` | `text` | Der Hauptinhalt. |
| `final String` | `title` | Als Zusammenfassung dienender Titel. |

Im Konstruktor der *Frame*-Klasse wird ein übergeordneter Ordner, der Ordner, in welchem die Hauptdatei gespeichert ist, und die Dateinamen für Audio und Sound übergeben. Die Pfade werden dann überprüft. Wenn vorhanden, wird das Feld gesetzt.

Das Autoren-Feld wurde eingeführt, um nachvollziehen zu können, wer diesen Frame geschrieben hat. Dieses Feld wird praktisch ignoriert und ist nur zu Vollständigkeitszwecken eingelesen.

Das *flag*-Feld ist ein Name für eine Flag – sehr kreativ. Diese wird gesetzt, nachdem dieser Frame ausgegeben wurde. Das heißt, beim ersten Mal, sollte dieser Frame die Flag abfragen, wird diese als nicht vorhanden ausgegeben, bei jedem weiteren Mal als vorhanden. Mehr Informationen zum Konzept der Flags in *Flags*.

Es passiert auch ein wenig Magie in dieser Klasse. Ein Frame hat nämlich noch zwei weitere private Attribute namens *textNodes* und *titleNodes*. Diese sind die reinen Elemente, welche direkt aus dem XML-Dokument ausgelesen wurden. Warum? In frühen Versionen der Engine wurde der Text bereits im *Interpreter* evaluiert. Das Problem hierbei ist, dass die Information über die Elemente in den XML-Tags hierbei verloren geht. Als also Flags implementiert wurden, brauchte es einen Weg, um den Text auch nach Auslesen nochmals zu evaluieren. In *FreakEngine*s *getFrame*-Methode wird in Wirklichkeit nicht das eigentliche *Frame*-Objekt zurückgegeben, sondern nur eine Kopie, welche mit der Methode *copyWithFlags* des eigentlichen *Frame*-Objekts erstellt wurde. In dieser werden beispielsweise die Flags neu evaluiert, wie der Name vermuten lässt. Alle anderen Attribute werden dabei übernommen.

# *FreakEngine*

Die Engine selbst ist das Herz des Projekts; sie schweißt alle Bauteile zusammen. Sie sollte benutzt werden, um das eigentliche Spiel zu erstellen. Klassen von anderen Paketen sind nicht erwünscht auf andere Klassen dieses Paketes zuzugreifen.

Der Konstruktor der *FreakEngine*-Klasse ruft die *loadRoutine* Methode der *Interpreter*-Klasse auf. Dadurch wird die *main.xml* Datei im Asset-Ordner ausgelesen und das daraus resultierende *Routine*-Objekt in das (private) routine-Attribut gespeichert. Auf dieses kann von anderen Paketen aus nicht zugegriffen werden. Stattdessen wird der Inhalt durch einige Helfermethoden vom *FreakEngine*-Objekt bereitgestellt. Siehe unten.

Der Pfad des Asset-Ordners wird bestimmt dadurch, dass der Pfad der Klasse ausgelesen wird. Dies gibt jedoch kein korrektes Ergebnis aus, sollte das Programm in eine JAR-Datei verpackt werden. Deshalb ist ein Workaround implementiert, welcher den Pfad korrigiert, sollte der Dateiname auf *.jar* enden. Es wird daher zu einem Fehler führen, wollte die JAR-Datei einen Dateinamen ohne die Endung haben, was theoretisch in Java allein noch valider Code wäre.

Ist das *Routine*-Objekt ausgelesen, macht die Engine selbst nichts mehr. Die Zusammensetzung des Spieles selbst ist den Renderern überlassen. Mehr Details über deren Implementation ist in *Renderer* nachzulesen.

Die Methode *getFrame* ist die Schlüsselfunktion der Engine. Sie scheint erstmal etwas Einfaches zu tun; durch das Array *routine.frames* zu gehen und den *Frame* mit der richtigen ID zurückzugeben. Im Kern ist das auch genau, was diese Methode macht. Doch sie hat eine weitere wichtige Aufgabe, neben der Abstraktion für Renderer: Flags.

Sobald die Methode aufgerufen wird und der geforderte *Frame* gefunden wurde, wird der Wert einer Speichervariable zur Flag diesen Frames gesetzt. Sie wird jedoch nicht in die Liste an aktuell gesetzten Flags aufgenommen. Dies geschieht erst im nächsten Aufruf. Denn bevor die Speichervariable gesetzt wird, wird der Wert dieser in besagte Liste aufgenommen. Dies macht es möglich, in einen Frame, welcher eine Flag setzt, die eigene Flag zu testen. Beim ersten Mal ist die Flag nicht vorhanden, beim zweiten Laden jedoch schon.

# *Flags*

Wir hatten ein Problem: Die Geschichte war sehr statisch. Da man sich nur plump durch die einzelnen Kapitel klicken kann, fühlt sich die Spielerfahrung nur sehr stumpf an. Eine Entscheidung, welche erinnert werden soll, wäre nur möglich, würde man jeweils zweimal die exakt gleichen Stränge aufsetzen und die Änderungen dort machen. Hard-Coding ist aber nicht sehr gerne gesehen. Eine Lösung musste her, und das dringend.

Eine Brainstorming-Session später die Lösung: Flags.

Ein Frame kann immer maximal eine Flag haben. Diese Flag bestimmt dann die Anzeige von verschiedenen Feldern oder Optionen. Eine (immer noch recht limitierte, aber weitaus bessere) Lösung, welche dazu noch einigermaßen einfach zu implementieren ist. Wie werden jetzt also Flags eingesetzt? Grundsätzlich können sie im Titel des Frames, Text oder einer Option vorkommen.

Bei ersteren beiden ist dies möglich, indem ein *if*-Tag ins in den Text- oder Titel-Tag eingefügt wird. Dieser hat ein Attribut namens *flag*. Es ist dabei egal, ob die Flag jemals umgeschaltet wird, anders als bei der Frame ID. Ist diese Flag vorhanden, wird der Text im *if*-Tag angezeigt. Sollte man einen Gegenfall einsetzen wollen, fügt man einen einfachen, selbst-schließenden *else*-Tag in den *if*-Tag ein. Alles vor dem *else*-Tag wird angezeigt, ist die Flag vorhanden, alles danach, falls nicht.

Bei Optionen können die Attribute *ifFlag* und *unlessFlag* verwendet werden. Beide dürfen nicht vorhanden sein. Wenn die Flag in *ifFlag* gesetzt ist, wird die Option angezeigt. Wenn jedoch die Option *unlessFlag* vorhanden ist und diese Flag gesetzt ist, wird die Option versteckt.

Eine Flag wird umgeschaltet oder auch gesetzt, nachdem ein Frame, welcher diese Flag in seinem *flag*-Attribut stehen hat, von der Methode *getFrame* der *FreakEngine* zurückgegeben wurde. Sollte der Frame ein zweites Mal angefordert werden, ist auch für ihn selbst die Flag gesetzt.

Für Informationen über die technische Umsetzung sollte der Abschnitt *Frame* zur Hilfe genommen werden. Dieser erklärt die anfänglichen Hindernisse und wie diese überkommen wurden.

# *Renderer*

Renderer ist im Kontext dieses Programms das eigentliche Programm selbst. Jeder Renderer sollte unabhängig von den jeweils anderen Renderern sein und auch ohne eine Abhängigkeit auf diese sich zu einem JAR-Archiv packen lassen, sodass dieses dann auch ausführbar ist. Er ist dafür verantwortlich, ein *FreakEngine*-Objekt zu kreieren. Die Options-Auswahl des jeweiligen Frames anzuzeigen und den korrekten nächsten Frame zu laden.

Ein Renderer sollte wie folgt agieren:

1. Zuerst muss dieser ein *FreakEngine*-Objekt mit dessen Konstruktor erstellen. Dieser benötigt keine Argumente. Dabei werden alle Assets geladen. Vorsicht ist geboten, der Konstruktor kann einen Fehler werfen.
2. Daraufhin muss der Renderer den Frame mit der ID `0` laden. Dies macht er, indem er die Methode *getFrame* des erzeugten *FreakEngine*-Objekts aufruft. Wichtig zu beachten ist hier, wie auch bei allen folgenden Ladungen, dass das erste Mal Laden des Frames eventuell ein anderes Ergebnis hervorbringt als die Folgenden. Bei fehlerhafter Eingabe oder Ähnlichem sollte also auf ***keinen Fall*** der Frame mit der gleichen ID erneut geladen werden! Stattdessen sollte das bereits geholte *Frame*-Objekt erneut angezeigt werden. Mehr Informationen über das Problem im Abschnitt *Frame*.
3. Darauffolgend wird der Inhalt des Frames ausgegeben. Für mehr Informationen und verfügbare Felder sollte hier der Abschnitt *Frame* verwendet werden.
4. Es muss eine Eingabe von nächsten Frames verfügbar sein. Dies kann beispielsweise durch eine Terminal-Eingabe oder Buttons gelöst werden. Wichtig ist dabei, dass die ID korrekt aufgelöst wird. Nach Eingabe wird wieder mit Schritt 2 fortgefahren, jetzt nur mit der eingegebenen ID statt `0` als Wert.

Aktuell gehören zwei verschiedene Renderer zu diesem Projekt, *TextRenderer* und *GuiRenderer*. Beide können in eine JAR-Datei verpackt werden. Wichtig ist hierbei, dass der Asset-Ordner aus dem Quellcode sich zusätzlich im gleichen Verzeichnis wie die JAR-Datei befinden muss.

# *Charaktere*

Die Klasse *CharacterManager* hat einen leicht zu erratenden Job: das Managen von Charakteren. Sie dient dabei als Registry für diese.

Die Charaktere werden zuerst im *Interpreter* eingelesen dabei werden diese anders als alle anderen Felder nicht direkt in einem Container-Objekt gespeichert, stattdessen werden diese zentral bei *CharacterManager* registriert.

Um dies zu tun, gibt es eine statische Methode namens *createCharacter*. Dieser Methode werden dann alle Daten übermittelt. Dazu gehört eine ID, welche wieder einzigartig sein muss, ein anzuzeigender Name und eine Zusammenfassung für diesen Charakter. Die Methode erstellt dann ein *Character*-Objekt und speichert dieses intern ab. All dies wird allerdings vom *Interpreter* selbstständig durchgeführt.

Die Methode *getCharacter* wird dem entgegengesetzt dafür verwendet, um ein *Character*-Objekt, welches davor registriert wurde, wieder aufzurufen.

Charaktere werden dafür verwendet, um eine einheitliche Plattform für die Personen in der Geschichte zu schaffen. Es ist dabei vorgesehen, dass jeder genannte Charakter in der Geschichte mit einem registrierten Character-Objekt hinterlegt wird und somit eine einheitliche Erfahrung gewährleistet wird.