Diplomarbeit

Thema: Moderne Ansätze zur Oberflächengestaltung für hardwarenahe Programmierung

I Inhaltsverzeichnis

1 Motivation

[Diese Diplomarbeit ist aus einem Bedürfnis entstanden, ein geeignete Technologie für die hardwarenahe Programmierung von Benutzeroberflächen zu verwenden. Es gibt dafür bereits eine Vielzahl von Möglichkeiten, welche jedoch mir persönlich aus unterschiedlichen Gründen nicht zusagten, da sie mir häufig zu statisch vorkamen oder verbunden waren mit einem Vendor Lock-in. Wenn man in den Bereich der Webentwicklung blickt erkennt man, welche Möglichkeiten vorhanden sind. Ich hatte das Gefühl, dass für die hardwarenahe Programmierung Platz nach oben ist in Bezug auf Dynamik und Agilität. Mit dieser Arbeit möchte ich dies unter Beweis stellen.

Meine Haupt- oder Leitthese lautet daher: Es ist möglich, bisherige Technologien zur hardwarenahe Programmierung von graphischen Oberflächen durch neue, innovative Konzepte aufzuwerten, um eine agile Entwicklung zu vereinfachen.] → überarbeiten

2 Begriffsklärungen

Klärung Begriff „modern“

[Erklärung]
→ sehr agil entwickeln; ständige Anforderungsänderungen; Entscheidungen möglichst lange hinauszögern

Klärung Begriff „hardwarenah“

[Erklärung]
→ Nicht auf Anforderungen für Bewertungskriterien eingehen. Lediglich allgemein erklären, worum es sich handelt (limitierte Rechen- und Energieresourcen, sensitives Zeitverhalten usw.)

Klärung Begriff „Prototyp“

[Erklärung]
- Unterschied Prototyp + Software

Klärung Begriff „graphische Benutzeroberfläche“

Im laufe der Diplomarbeit wird häufig der Begriff „graphische Benutzeroberfläche“ o.ä. genutzt. Hierbei wird sich ausschließlich auf Benutzeroberflächen bezogen, welche eine gewisse visuelle Ästhetik besitzen. Darunter fallen Displays, welche Inhalte pixelgenau wiedergeben können, sowie über eine Farbtiefe von mindestens 24-Bit verfügen.

Klärung Begriff „Softwarearchitekturen“

3 Vorbetrachtungen zu Möglichkeiten der Programmierung von Benutzeroberflächen

3.1 Bisherige Architekturen für die Programmierung von Benutzeroberflächen

Graphische Benutzeroberflächen sind zu einem festen Bestandteil von jedem Softwaresystem geworden. Für die Realisierung ist eine Vielzahl an verschiedenen Lösungen entstanden, welche Ähnlichkeiten aufweisen [1]. Es ist daher schwierig, einzelne Paradigmen strikt voneinander zu trennen, da Überlappungen auftreten. Dennoch sollen in diesem Kapitel typische Konzepte vorgestellt werden, welche sich historisch herausgebildet haben.

3.1.1 Formulare und Kontrollelemente

Dieses Softwarekonzept ist relativ einfach aufgebaut. Die Grundidee besteht darin, graphische Oberflächen über Formulare abzubilden, welche mehrere Kontrollelemente beinhalten. Über die Kontrollelemente werden Daten angezeigt, sowie eine Interaktion ermöglicht.

Eine Grundproblematik in diesem Konzept besteht darin, die Daten synchron zu halten. Häufig befinden sich die Daten auf der GUI, intern in der Geschäftslogik und auf einem persistenten Speichermedium, teilweise mehrfach [1]. Ein populäres Muster, um dieses Problem zu lösen ist Datenbindung. Die Idee ist Änderungen in einem Datensatz sofort zu propagieren und so an allen nötigen Stellen zu ändern. Wenn die Übertragung abgeschlossen ist, wurde der Datensatz an allen Stellen geändert [2].

Dazu ein Beispiel: Gegeben sei eine Softwareanwendung, welche aus einem Formular mit jeweils einem Kontrollelement Textfeld, Button und einem Ausgabefeld besteht. Die Funktion des Textfeldes besteht darin, eine Namenseingabe zu ermöglichen. Über den Button soll die Eingabe bestätigt werden können. Nach der Bestätigung, soll in dem Ausgabefeld der Nutzer gegrüßt werden, indem vor dem eingegebenen Namen ein Begrüßungswort gehangen wird, also beispielsweise „Hallo Anna“. In dieser Anwendung könnten laut dieser Architektur je Kontrollfeld 3 Variablen existieren: Die erste bildet den Zustand auf dem Bildschirm ab. Die Zweite bildet den internen Zustand im Programm ab und eine dritte, bildet den Zustand auf einem Speichermedium ab, z. Bsp. eine Textdatei oder eine relationale Datenbank. Letzteres könnte den Zweck verfolgen, die Daten über einen Applikationsneustart zu erhalten.

Sobald der Nutzer anfängt, den Namen in das Textfeld einzugeben wird durch die Datenbindung nicht nur der Variablenwert für die GUI-Oberfläche, sondern gleichzeitig der Variablenwert innerhalb des Programms geändert. Nach Bestätigung des Namens kann sowohl der interne Variablenwert oder der GUI-Variablenwert für das Ausgabefeld geändert werden. Dies spielt keine weitere Rolle, weil die Änderungen in jedem Fall über das Muster Datenbindung durchgegeben werden.

Was in diesem simplen Anwendungsfall einfach aussieht, stellt sich in der Praxis jedoch als trickreich heraus, da Zyklen vermieden werden müssen. Daher falls die Daten intern geändert werden und diese Änderung an die GUI übergeben wird, darf diese nicht wiederum die Änderung an die interne Variable weitergegeben werden. Da jedoch grundsätzlich in diesem Konzept eine bidirektionale Änderung der Daten möglich sein muss, legt die Komplexität dieses Verfahrens einen Einsatz von Software-Frameworks nahe [1].

Dabei implementieren Software-Frameworks dieses Verfahren intern häufig über das Beobachter-Muster [1]. Dies bedeutet, dass entsprechende Ereignisse ausgelöst werden, wenn sich die Variablen ändern. In den Ereignisbehandlungsroutinen muss die Änderung der jeweils redundanten Datensätzen vorgenommen werden.

Historisch gesehen wurde diese Architektur gehäuft bei Client-Server Applikationen eingesetzt, da es die Funktionalität der Änderung von Datensätzen sehr gut abbildet, insbesondere die Änderung von Datensätzen in relationalen Datenbanken. In dieser Domäne wird dieses oft benötigt [1].

3.1.2 Model-View-Controller (MVC)

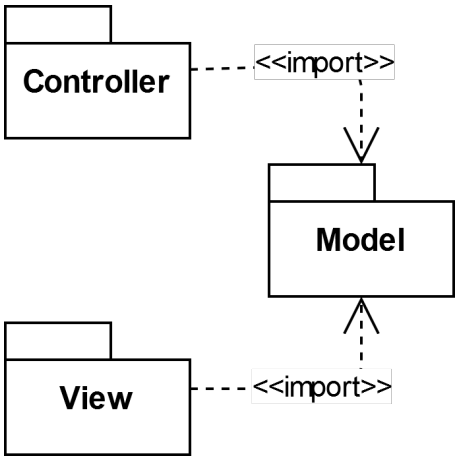
Der Model-View-Controller (zu dt. „Modell-View-Steuerung“) ist ein weiteres Konzept für die Strukturierung von Software zur graphischen Oberflächengestaltung. Es ist mit eines der historisch ältesten Konzepte für die graphische Programmierung. Zurückblickend, waren in den 1970er Jahren GUI‘s keineswegs gängig. Das MVC-Konzept ist mit eines der ersten Ansätze, Benutzeroberflächenprogrammierung in großem Umfang zu betreiben. Im Laufe der Zeit, wurden mehrfach Konzepte als MVC bezeichnet, welche jedoch eher wenig mit dem Ursprung gemein hatten [1]. Deswegen ist es umso wichtiger, dieses Paradigma explizit zu besprechen, um eine Abgrenzung von anderen Konzepten zu ermöglichen.

Die Anfänge sind stark mit der Programmiersprache Smalltalk verzweigt, welche zugleich als eine der ersten objektorientierten Programmiersprachen gilt [3]. Sie geht in die 80er Jahre zurück [1]. Im folgenden Verlauf der Diplomarbeit wird der Begriff MVC auf das Konzept, welches Smalltalk verwendete, eingegrenzt.

Eine der wichtigsten Errungenschaften von MVC ist die Separierung von graphischen Elementen und Domain-Objekten. Letztere sollten in der Lage sein, komplett losgelöst

von jeglichem graphischen Inhalt zu funktionieren. Sie bilden unsere Sicht auf die Welt ab. Sie müssen in der Lage sein, mehrere Präsentation-Schichten zu bedienen. In der Unix-Welt wird dies stark genutzt, da nahezu jede Applikation sowohl über GUI, als auch über Kommandozeile ansteuerbar ist. Dies ist nach aktuellem Stand immer noch der Fall.

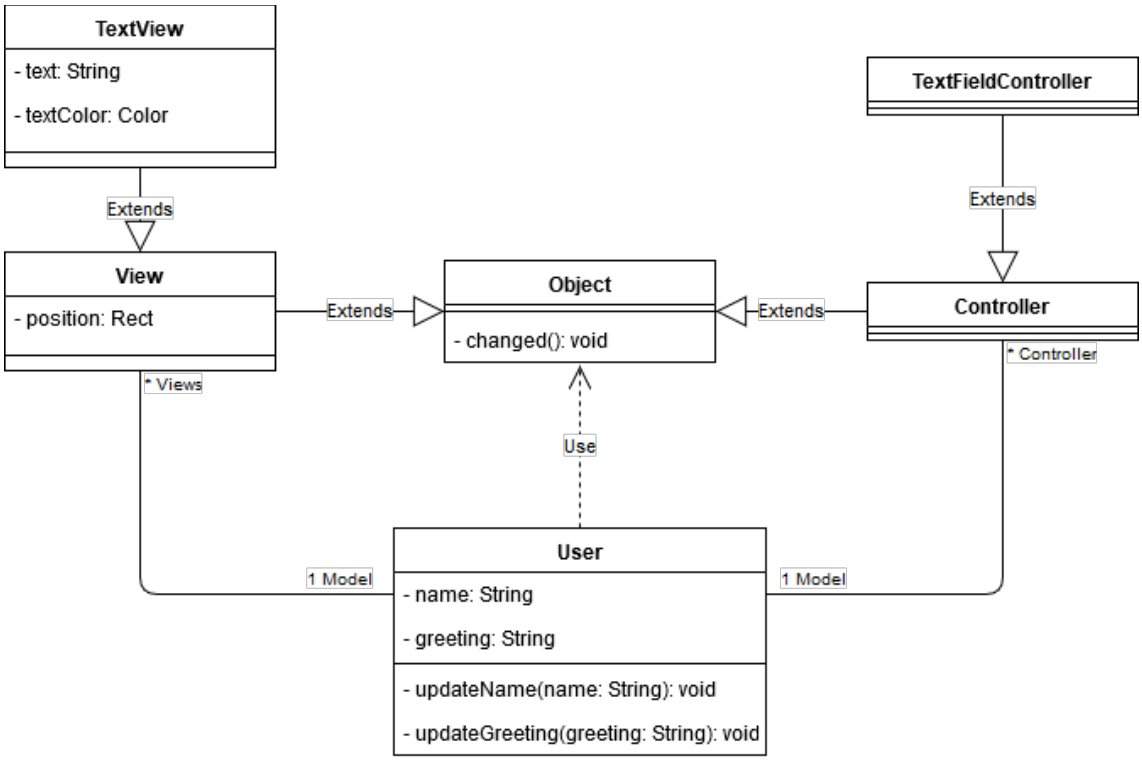
Um das Konzept im Groben zu beschreiben, werden Domain-spezifische-Objekte innerhalb des Musters als „Model“ bezeichnet. Die graphischen Elemente bilden den „View“. Die Aufgabe des „Controller“ ist es, Benutzereingaben zu verarbeiten. Dabei kennt im strengen Sinn das Model weder Controller noch View. Ebenso kennen sich Controller und View nicht. 1. Abbildung: Zusammenspiel zwischen Model, View und Controller soll dieses Zusammenspiel als Paketdiagramm verdeutlichen.



1. Abbildung: Zusammenspiel zwischen Model, View und Controller

Das Model kann dabei als Abbildung der realen Welt gesehen werden. Anders als in dem Formulare- und Kontrollelemente-Architektur-Muster geht dieses Model nicht von Datensätzen aus, die manipuliert werden sollen, sondern von Objekten, mit welchen man in der Anwendung agiert.

Um diese Architektur zu demonstrieren soll als Beispiel der gleiche Anwendungsfall dienen, welche unter Formulare und Kontrollelemente ausgewählt wurde. Wenn die Begrüßungsanwendung über das MVC-Konzept umgesetzt werden würde, so könnte das Model aus einem Benutzer (engl. „User“) bestehen, welches einen Namen beinhaltet und einen Grußtext. Dazu wurde ein Klassendiagramm erstellt, siehe 2. Abbildung: Begrüßungsanwendung im MVC-Konzept, Klassendiagramm.

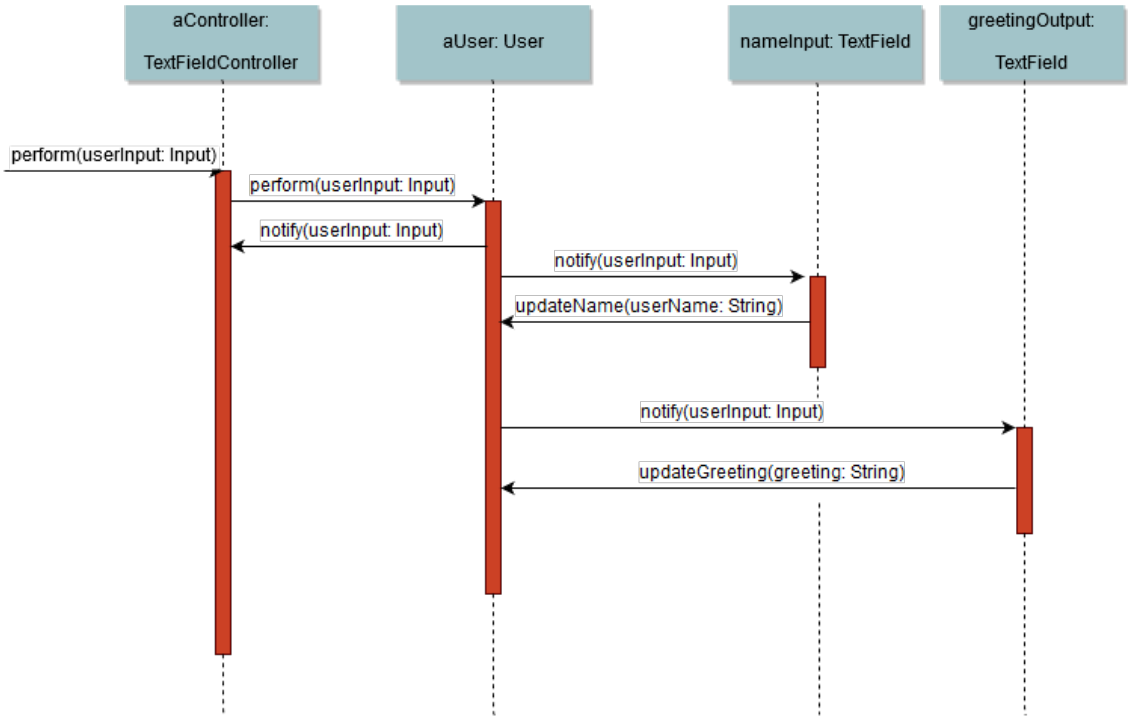


2. Abbildung: Begrüßungsanwendung im MVC-Konzept, Klassendiagramm

Innerhalb dieses Klassendiagramms repräsentieren die Klassen „TextView“ und „View“ den View, die Klassen „TextFieldController“ und „Controller“ den Controller und die Klasse „User“ das Modell. Die Klasse „Objekt“ befindet sich dabei entweder je nach Programmiersprache in der Standardbibliothek oder kann gegebenenfalls in einer eigenen Softwarebibliothek bereitgestellt werden.

Die Klassen „View“ und „Controller“ dienen dabei als abstrakte Basisklassen, um gemeinsam genutzte Funktionalität innerhalb der Views bzw. Controller abzubilden.

Zu dem Modell hinzuzufügen ist, dass das Modell, daher die Klasse „User“ weder View noch Controller kennt. Die Beziehung zwischen View und Modell bzw. Controller und Modell, welche in dem Klassendiagramm jeweils über eine 1 zu * Beziehung dargestellt wurde, ist dabei ausschließlich zur Veranschaulichung des MVC-Konzepts gedacht. In der Praxis würde die Klasse „TextView“ bzw. „TextFieldController“ die Klasse „User“ kennen, nicht jedoch in entgegengesetzter Richtung. Die Klasse „User“ würde lediglich über die Schnittstelle „Object“ ihre Änderungen mitteilen. Wie dieser Fall im Konkreten aussieht, soll gezeigt werden. Dazu wird der Änderungsfall betrachtet, dass eine Nutzereingabe erfolgt und ein Text eingegeben wird. Die Änderungen sollen über ein Sequenzdiagramm gezeigt werden, siehe 3. Abbildung: Begrüßungsanwendung im MVC-Konzept, Sequenzdiagramm für Nutzereingabe.



3. Abbildung: Begrüßungsanwendung im MVC-Konzept, Sequenzdiagramm für Nutzereingabe

Aus dem Sequenzdiagramm lässt sich entnehmen, dass die Objekte „aController“, „aUser“, „nameInput“ und „greetingOutput“ jeweils eine Instanz der Klassen „TextFieldController“, „User“ bzw. „TextField“ für die letzten beiden darstellen. Eine Nutzereingabe wird nach dem MVC-Konzept an den Controller weitergeleitet, in diesem Fall „aController“. Dieser gibt die Änderungen an das Modell, in diesem Fall „aUser“ weiter. Das Objekt „aUser“ wird nun alle benachrichtigen, welche sich für Änderungen registriert haben und den Nutzerinput entsprechend weiterleiten. Wie ebenso dem Sequenzdiagramm zu entnehmen ist, ist der Nutzerinput eine Instanz der Klasse „Input“. Sie enthält alle wichtigen Daten zu dem Textinput, vor allem, jedoch nicht ausschließlich, den eingegebenen Text.

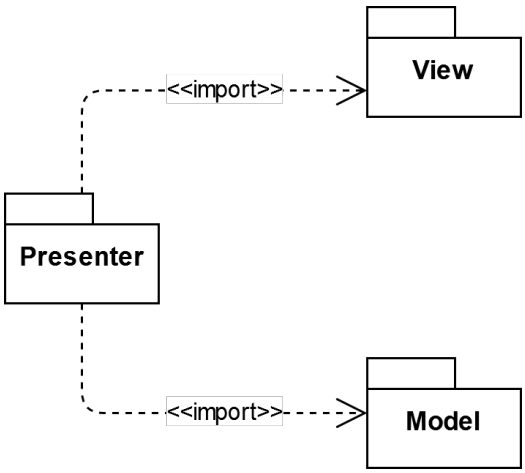
Wie dem Beispiel zu entnehmen ist wird der Textinput an die Objekte „aController“, „nameInput“, sowie „greetingOutput“ weitergeleitet. Diese können den Input nun verarbeiten. Dabei updaten die Objekte „nameInput“ und „greetingOutput“ die jeweiligen Variablenwerte für die Anzeigetexte intern und propagieren die Änderungen über die Funktionen „updateName“ bzw. „updateGreeting“ zurück an das Modell. Dadurch sind die Änderungen an allen notwendigen Stellen für den Anwendungsfall durchgeführt und der View kann die entsprechende Begrüßung anzeigen.

Auch wenn der Implementierungsaufwand relativ zu Formulare und Kontrollelemente etwas größer ist durch die Implementierung des Beobachter-Musters besteht der klare Vorteil in der Testbarkeit des Modells. Das Modell weist keine Abhängigkeiten zu View oder Controller auf und kann damit separat mittels Unit-Tests getestet werden. Ebenso wurde die Anwendungslogik sauber von der Anzeigelogik abgekapselt.

3.1.3 Model-View-Presenter (MVP)

Das Konzept des Model-View-Presenter (zu dt. „Modell-Präsentation-Steuerung“) soll als Abänderung des MVC-Konzepts, siehe Kapitel 3.1.2 Model-View-Controller (MVC) erklärt werden. Ebenso wie für das MVC-Konzept gibt es für das MVP-Konzept eine Vielzahl an Variationen. Für die Diplomarbeit wird daher der Begriff des MVP auf die in diesem Kapitel vorgestellte Variante des „Passive View“ (engl, zu dt. „Passive Anzeige“) beschränkt [4].

Im Gegensatz zu dem MVC-Konzept besitzt das MVP-Konzept anstelle eines „Controllers“ (engl., zu dt. „Kontrolleur“) einen „Presenter“ (engl, zu dt. „Präsentierer“). Dieser ist wie folgt zwischen Modell und View einzuordnen, siehe 4. Abbildung: Zusammenspiel zwischen Model, View und Presenter.



4. Abbildung: Zusammenspiel zwischen Model, View und Presenter

Wie in oben stehender Abbildung zu erkennen ist, besteht ein markanter Unterschied zu dem MVC-Konzept darin, dass keine direkte Verbindung zwischen Modell und View besteht und sämtliche Kommunikation stattdessen über den Presenter durchgeführt wird. Die Views verfügen dabei über eine Schnittstelle, worüber der Presenter kommunizieren kann [5].

Ein zweiter markanter Unterschied besteht darin, dass Nutzereingaben nicht mehr direkt an den Controller weitergeleitet werden, sondern stattdessen an den View. Dieser leitet den Input an den Presenter weiter, welcher sowohl den View, als auch das Modell und sich selbst intern entsprechend aktualisieren kann anhand der Nutzereingabe.

Diese beide Veränderungen scheinen zunächst die graphische Benutzeroberflächenprogrammierung komplexer werden zu lassen, bieten jedoch folgende zwei Vorteile:

- Der Presenter kann im Vergleich zu dem Controller einfacher durch Tests abgedeckt werden. Dies liegt in der Tatsache zu Begründen, dass die Views über Schnittstellen verfügen, welche für Tests durch Test-Funktionalität implementiert werden können. In dem strikteren MVC-Konzept greift der Controller direkt auf die View-Objekte zu.

- Die Views sind nicht mehr an das Domain-Modell geknüpft, wodurch eine höhere Wiederverwendbarkeit gegeben ist.

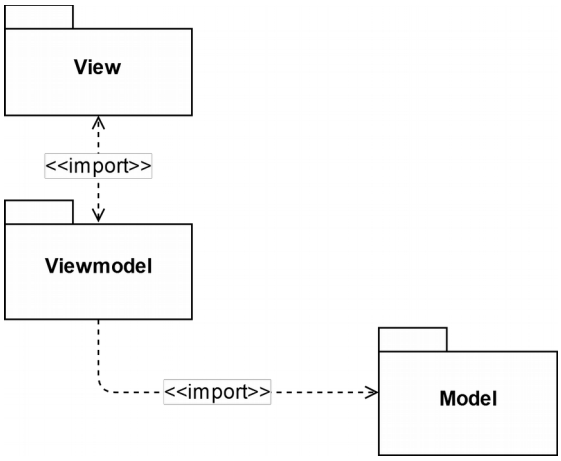
Dabei ist anzumerken, dass obwohl Presenter und Modell in dem MVP-Konzept gut testbar sind, der View nach wie vor schwer testbar ist. Hierbei ist anzumerken, dass das Testen von GUI-Funktionalität im Allgemeinen ein großes Problem darstellt. Deswegen hat sich eine Variante des MVP-Konzeptes entwickelt, welche im Fachjargon „Passive View“ (engl, zu dt. „Passive Anzeige“) heißt. Hierbei wird versucht, möglichst viel Funktionalität in Modell und Presenter auszulagern und den View möglichst klein zu halten. Im konkreten ist es am Besten, die Logik möglichst viel im Modell zu haben, da dort am meisten wiederverwendet werden kann. Im Presenter kann Quellcode nur bedingt wiederverwendet werden, ist jedoch noch testbar. Der View sollte im Idealfall lediglich übermittelte Informationen aus dem Presenter anzeigen. [4]

Da die Variante des „Passive View“ ein hochmoderner Ansatz für graphische Benutzeroberflächenentwicklung darstellt, wurde dieser für den Prototyp in dieser Diplomarbeit ausgewählt und verwendet.

3.1.4 Model-View-Viewmodel (MVVM)

Das Konzept des Model-View-Viewmodel (zu dt. „Modell-View-Viewmodell“) stellt neben dem MVP-Konzept einen weiteren modernen Ansatz für graphische Benutzeroberflächenentwicklung dar. Dieses Konzept weist Ähnlichkeiten zu MVC auf und beinhaltet das Muster der Datenbindung, welches in dem Kapitel 3.1.1 Formulare und Kontrollelemente erklärt wurde.

Das Zusammenspiel im Groben ist dabei wie folgt, siehe 5. Abbildung: Zusammenspiel zwischen Model, View und Viewmodel.



5. Abbildung: Zusammenspiel zwischen Model, View und Viewmodel

Die Ähnlichkeit zu dem MVC-Konzept besteht darin, dass der Quellcode in einen View, ein Modell und in diesem Fall ein Viewmodel zerlegt wird. Das Viewmodel übernimmt dabei die Aufgabe des Controllers. Es kann als Mittelmann zwischen View und Model gesehen werden [6].

Hervorzuheben ist für dieses Konzept, dass eine gegenseitige Abhängigkeit zwischen View und Viewmodel besteht. Dieser Umstand ist der Datenbindung geschuldet, welche zwischen beiden Teilen vorhanden ist. Dies bedeutet, dass Änderungen an Variablenwerten innerhalb des Views automatisch an das Viewmodel weiterpropagiert werden und entgegengesetzt. Das Viewmodel greift auf die Objekte der Anwendungsdomäne aus dem Model zurück und sorgt für die korrekte Anzeige.

Nutzereingaben werden ähnlich wie bei dem Controller in dem MVC-Konzept direkt an das Viewmodel übergeben und von dort aus weiterverarbeitet.

Die enge Bindung zwischen View und Viewmodel birgt den Vorteil, dass die Anzeige von Elementen sehr einfach zu implementieren ist. Der Nachteil besteht jedoch darin, dass View und Viewmodel praktisch ohne immensen Mehraufwand unmöglich voneinander getrennt werden können und sie damit nur im Block verwendet werden können. Ein Testen des Viewmodels wird dadurch ebenso erschwert und kann nur in seltenen Fällen überhaupt durchgeführt werden, weil die zugehörigen Views nicht mit Testfunktionalität austauschbar sind.

Ähnlich wie bei Formulare und Kontrollelemente legt die Datenbindung den Einsatz eines geeigneten Software-Frameworks nahe. Ein Beispiel dazu ist das Windows Presentation Framework (WPF) von der Firma Microsoft, welches über die Auszeichnungssprache „XAML“ eine Datenbindung nach Quellcode in die Programmiersprache „C#“ zulässt [7]. In diesem Fall ist das MVVM-Konzept fest in dem Framework integriert. Der View-Teil wird dabei ausschließlich mit der

Auszeichnungssprache „XAML“ entwickelt und das zugehörige Viewmodel ist dabei in C#-Code geschrieben.

3.2 Vorgehensweise für die Programmierung von Benutzeroberflächen

Für die Entwicklung von Benutzeroberflächen muss sowohl das graphische Design, als auch die Programmierung erfolgen. Da die Programmierung mit hohen Kosten belegt ist, ist es eine valide Vorgehensweise das graphische Design zuerst zu erstellen und dann anschließend mit der gewünschten Funktionalität zu versehen. Das graphische Design kann dazu mittels geeigneten Anwendungen, z. Bsp. Photoshop [8] oder proto.io [9] erstellt und im Roh-Zustand getestet werden. Wenn das Design zusagt, kann anschließend die Implementierung erfolgen. Dazu müssen die graphischen Elemente in geeigneter Form, z. Bsp. als *.png oder *.jpg Dateien exportiert und gespeichert werden. Ebenso müssen andere Ressourcen, wie Fonts oder Animationen in geeigneter Form abgelegt und exportiert werden. Diese können dann anschließend in der Software verwendet werden.

Ein Vorteil von dieser Vorgehensweise besteht darin, dass das graphische Design und die Implementation sauber getrennt voneinander erfolgen können. Die Schnittstelle zwischen Designer und Programmierer ist deutlich durch die extrahierten Dateien gegeben. Dies wird sich in der Diplomarbeit zunutze gemacht, sodass das graphische Design von einem Externen erstellt wird. Die Programmierung ist dabei Bestandteil der Diplomarbeit.

Es kann dadurch sowohl die visuelle Qualität des Endproduktes, als auch der wissenschaftliche Anspruch an die Diplomarbeit gewährleistet bleiben.

3.3 Moderne Anforderungen an die Programmierung von Benutzeroberflächen

Die Programmierung von Benutzeroberflächen unterliegt Veränderungen. Es kann als Trend festgehalten werden, dass zunehmend die Nutzer-orientierte Entwicklung von Software in den Vordergrund rückt [10]. Dies verlangt der Programmierung neue Vorgehensweisen ab. Es kann im Laufe der letzten Jahre eine Agile Bewegung festgestellt werden, mit welcher eine Vielzahl von neuen Tools und Vorgehensweisen entwickelt wurde, siehe [11]. Einzelne, jedoch bedeutende, Vertreter dieser Bewegung unterzeichneten, ein „Agiles Manifest“, in welchem die Ziele der Bewegung definiert wurden [12]. Darin lässt sich unter anderem eine hohe Bereitschaft finden, auf Änderungen einzugehen. Der nahe Kundenkontakt und das eingehen auf Veränderung gewinnt ebenso an Bedeutung. Starre Pläne sind dabei weniger zielführend.

In der Diplomarbeit muss deswegen bedacht werden, dass der Prototyp keine bewegungslose Software wird, die in einen Monolithen endet. Sie muss stattdessen die nötige Flexibilität besitzen, um auf Änderungen schnell und flexibel eingehen zu können.

Des Weiteren stellt dies ebenso neue Anforderungen an die Testabdeckung. Bei schnellen und kurzen iterativen Veränderung eines Softwaresystems, haben manuelle Tests keinen Platz. Sie würden kurze Entwicklungs-Zyklen verhindern, denn nach jedem Zyklus würden die manuellen Tests Zeit beanspruchen.

Um dem entgegenzuwirken, haben automatisierte Tests an Bedeutung gewonnen. Es ist unabdingbar, dass ein Software System automatisch testbar ist, um schnelle Änderung und eine gewisse Robustheit zu garantieren. Die Testbarkeit ist der Grund, weswegen das MVP-Konzept für den Prototypen ausgewählt wurde, weil es diese am stärksten ermöglicht.

Ein weiterer Trend in der Benutzerschnittstellenprogrammierung ist eine steigende mediale Aufbereitung der Oberflächen. Es lässt sich feststellen, dass Texte an Bedeutung verlieren und diese durch Bilder und dynamische Inhalte ersetzt werden. Als Beispiel soll die Startseite des Webauftritts der Firma „Apple“ dienen, welche bekannt ist für Ihren hohen Standard an Design und viele Innovationen, wie beispielsweise das Smartphone essentiell vorangetrieben hat. Wie auf der Startseite zu erkennen ist, wird kein Textblock angezeigt, sondern stattdessen Schlagworte und Bilder zu Produkten [13]. Auf die Wirkung auf den Benutzer im Kontext des Marketing soll an dieser Stelle nicht eingegangen werden, weil es den Rahmen dieser Diplomarbeit sprengen würde. Es kann aber festgehalten werden, dass ein guter Support von Text weniger stark benötigt wird, sondern stattdessen ein hoher Anteil von medialen Support nötig ist für eine moderne Programmierung von Benutzeroberflächen.

3.4 Anforderungen durch hardwarenahe Programmierung

Hardwarenahe Programmierung weist im Vergleich zur regulären Softwareentwicklung Besonderheiten auf. Diese sind hauptsächlich durch das Umfeld bedingt, in dem hardwarenahe Programmierung Anwendung findet. Auf typische Besonderheiten soll in diesem Kapitel eingegangen werden. Außerdem soll der recht allgemeine Begriff der hardwarenahen Programmierung konkret definiert werden.

Neben dem Begriff der hardwarenahen Programmierung lässt sich in der Literatur der Begriff der Programmierung für eingebettete Systeme (zu engl. „Embedded Systems Programming“) finden. Embedded Systems Programming bezeichnet dabei die Programmierung von Software für eingebettete Systeme [14], wobei eingebettete Systeme als eine Kombination von Computer Hard- und Software bezeichnet werden,

welche für eine spezifische Funktionalität innerhalb eines größeren Softwaresystems konzeptioniert wurde.

Der Begriff der hardwarenahen Programmierung wird für die Diplomarbeit so eingegrenzt, dass er den Begriff der Programmierung für eingebetteten Systemen umfasst und zusätzlich auch die Entwicklung von All- bzw. Mehrzweck-Anwendungen enthält. Hardwarenahe Programmierung ist dabei als eine grundsätzliche Vorgehensweise im Bereich der Softwareentwicklung zu verstehen, in welcher der ressourcenschonende Einsatz von Rechnerkapazitäten im Vordergrund steht.

Folgende Aspekte stehen bei der Programmierung für eingebettete Systeme bzw. bei hardwarenaher Programmierung im Vordergrund [15, S. 14 f.]:

- Geringe Rechenkapazität
- Wenig Arbeitsspeicher
- Entwicklungskosten im Widerspruch zu Produktionskosten
- Erwartete Lebensdauer
- Ausfallsicherheit

Es folgen Erklärungen für jeden Punkt einzeln.

Geringe Rechenkapazität
Die Rechenkapazität im Bezug auf den Prozessor ist häufig eingeschränkt. Die Programmierung findet nicht für leistungsfähige Prozessorchips statt, sondern reicht gerade für die notwendige Anwendung aus.

Wenig Arbeitsspeicher
Ähnlich wie die Rechenkapazität ist auch der Arbeitsspeicher eingeschränkt. Es ist nicht unüblich, den Arbeitsspeicherbedarf einer Anwendung zu limitieren.

Entwicklungskosten im Widerspruch zu Produktionskosten
Entwicklungskosten bezeichnet die Kosten in einem Projekt, welche für die Entwicklung des Produktes und damit der Software entstehen. Produktionskosten bezeichnen die Kosten, welche für die Produktion von einer Produkteinheit entstehen. Durch eine hardwarenahe Programmierung entstehen initial höhere Entwicklungskosten. Diese können jedoch durch geringere Produktionskosten, da weniger Arbeitsspeicher und Rechenkapazität in dem Produkt benötigt werden, ausgeglichen werden. Es findet je nach praktischem Anwendungsfall und Herstellmenge ein Kompromiss zwischen Entwicklungsinvestition und Produktkosten statt. Die Ziele niedrige Entwicklungs- bzw. Produktionskosten stehen im Widerspruch. Die Extreme sind in der Praxis selten zielführend. Diese Diplomarbeit verfolgt mit dem Prototypen und der dazugehörigen Lösung das Ziel, durch eine Aufwertung der Technologie mehr Entwicklerkomfort zu schaffen, um die Entwicklungskosten zu senken. Dabei sollen die Produktionskosten möglichst nicht ansteigen, jedoch werden minimale Änderungen in Kauf genommen.

Erwartete Lebensdauer
Die erwartete Lebensdauer der Software spielt bei der hardwarenahen Programmierung ebenso eine Rolle. Ein Softwaresystem mit nur geringer Lebensdauer ist deutlich günstiger zu implementieren, als ein System, welches Jahrzehnte verwendet werden soll. Es sei dabei angemerkt, dass in der Praxis häufig ein Softwaresystem länger Bestand hat, als die Entwickler ursprünglich im Sinn hatten.

Ausfallsicherheit
Je nachdem wie entscheidend die Ausfallsicherheit der Software ist, können die Entwicklungskosten gesenkt bzw. erhöht werden.

Neben den Aspekten, welche bei hardwarenaher Programmierung eine Rolle spielen, existieren typische Anforderungen an Softwareprojekte. Diese Anforderungen sind in die 3 Kategorien „Niedrig“, „Mittel“ und „Hoch“ eingeteilt worden und in folgender Tabelle in Bezug zu den Kriterien dargestellt, siehe 1. Tabelle: Typische Anforderung an hardwarenahe Programmierung, übernommen von [15, S. 16].

1. Tabelle: Typische Anforderung an hardwarenahe Programmierung, übernommen von [15, S. 16]

Kriterium	Niedrig	Mittel	Hoch
Prozessor	4- oder 8-bit	16-bit	32- oder 64-bit
Arbeitsspeicher	< 16 KB	64 KB bis zu 1 MB	> 1 MB
Entwicklungskosten	< $100.000	$100.000 bis $1.000.000	> $1.000.000
Produktionskosten	< $10	$10 bis $1.000	> $1.000
Anzahl an Einheiten	< 100	100 – 10.000	> 10.000
Erwartete Lebensdauer	Tage, Wochen oder Monate	Jahre	Jahrzehnte
Zuverlässigkeit	Darf gelegentlich ausfallen	Muss zuverlässig arbeiten	Muss fehlersicher arbeiten

Es sei angemerkt, dass die Geldbeträge nur im Groben zu interpretieren sind und die Entwicklungs- und Produktionskosten aufgrund des Zeitpunktes der Quelle nicht zwangsläufig aktuell sein müssen. Die Anforderung „Niedrig“ repräsentiert beispielsweise einen Mikrocontroller, welcher in Digitaluhren verbaut wird [15, S. 16]. Die Anforderung „Mittel“ könnte für das Kontrollsystem eines Fahrstuhls stehen, bei welchem die Graphische Benutzeroberfläche aus einem einfachen LED-Display besteht. Die Anforderung „Hoch“ umfasst unter anderem Anwendungen für mobile Endgeräte, daher auch Smartphones.

Für detailliertere Beispiele zu den 3 Kategorien sei an dieser Stelle auf [15, S. 16 ff.] verwiesen. In dieser Diplomarbeit soll lediglich die rechte Spalte der 1. Tabelle:

Typische Anforderung an hardwarenahe Programmierung, übernommen von [15, S. 16] betrachtet werden und damit die Anforderungskategorie „Hoch“. Diese Einschränkung wurde getroffen, weil nur in dieser Kategorie ansprechend visuell aufbereitete graphische Benutzeroberflächen zum Einsatz kommen. Die anderen beiden Kategorien haben auch ihre Berechtigung, jedoch findet eine Programmierung einer Benutzerschnittstelle nicht in dem Umfang statt, wie er für die Forschung im Rahmen der Diplomarbeit relevant ist.

Aus dieser Anforderungskategorie lassen sich Rückschlüsse für die praktische Programmierung ziehen. Zum einen liegt eine Vielzahl von unterschiedlichen Chip-Architekturen vor. Jede Chip-Architektur enthält ihre eigenen Befehlssätze. Für eine moderne Lösung würde eine Programmierung in Assembler, eine Programmiersprache der 2. Generation, wenig Sinn ergeben, weil für jede Chip-Architektur eine eigene Implementierung stattfinden müsste. Dieser Ansatz ist sehr mühselig. Der Einsatz einer portablen Hochsprache, beispielsweise „C“, liegt daher nahe, weil sie auf allen Chip-Architekturen lauffähig ist. Die Portabilität muss durch eine neue Technologie im größtmöglichen Umfang erhalten bleiben, damit dieser Vorteil bestehen bleibt.

Ein weiterer Rückschluss aus diesen Anforderungen ist, dass der rechnerische Mehraufwand verursacht durch die Benutzung einer Technologie gering bleiben muss. Es existieren Hochsprachen, wie beispielsweise „Java“, welche nicht referenzierte Objekte mittels eines „Garbage Collectors“ (engl, zu dt. „Müllsammler“) zerstören. Der Garbage Collector bezeichnet dabei einen Algorithmus, welcher sämtliche Abläufe im Programm stoppt und überprüft, welche Objekte nicht mehr benötigt werden und diese anschließend zerstört. Eine Anwendung dieser Funktionalität ist nicht im Interesse von hardwarenaher Programmierung, obwohl sie zweifellos Entwicklerkomfort bietet. Ebenso ist das dynamische Erstellen von Objekten mit Vorsicht zu genießen. Der Grund liegt darin, dass dynamischer Speicherplatz über die Datenstruktur eines Heaps reserviert werden muss.

Auf die Problematik der dynamischen Allokation bei einem Heap soll näher eingegangen werden. Der Heap verfügt über Speicher, welcher als lange eindimensionale Reihe aus Bytes gesehen werden kann. Beim dynamischen Erstellen muss anders als beim statischen zunächst geprüft werden, welcher Speicherbereich noch frei ist und ob er auch die benötigte Breite für das gewünschte Objekt besitzt. Dieser Vorgang ist nicht deterministisch, weil nicht im Vorfeld feststeht, wann verfügbarer Speicher gefunden wird. Anders ist es bei einer statischen Allokation von Speicher. Das statische allokieren von Speicher erfolgt, wenn ein neuer Kontext, beispielsweise eine Funktion, betreten wird. Dabei wird die statische Allokation in der Regel über einen Stack implementiert, welcher lediglich um die benötigte Länge erweitert werden muss. Der Nachteil dabei ist, dass die Freigabe des allokierten Speichers beim Verlassen des Kontexts erfolgt. Deswegen sind auch Funktions-interne Variablenwerte beispielsweise nicht von außerhalb zugänglich, weil sie statisch allokiert werden. Die statische Allokation erfolgt deterministisch und ist keine teure Operation und somit relativ bedenkenlos durchführbar. Die dynamische Allokation muss mit Vorsicht genutzt werden, zumal es während des Programmablaufes zu einer Fragmentierung in dem

Heap kommen kann. Dies bedeutet, dass durch eine dauerhafte Allokation und Freigabe von dynamischen Speicher, belegte Speicherbereiche zunehmend unorganisierter verteilt werden können und freier Speicher aufwändigerer zu finden ist. Diese Problematik ist für hardwarenahe Sprachen, wie beispielsweise C, sehr gut bekannt und ist bei Bedarf in der Quelle [16] näher erklärt.

Um dieses Kapitel zusammenzufassen beschränken sich die Anforderungen an hardwarenahe Programmierung in der Diplomarbeit auf die Verwendung einer portablen Hochsprache, wobei die Portabilität durch eine entwickelte Technologie erhalten bleiben muss, der Verzicht auf Hardware-intensive Algorithmen, wie beispielsweise einen Garbage Collector, und eine allgemeine Rechnerkapazitäten-schonende Softwareentwicklung.

4 Auszeichnungssprachen

Auszeichnungssprachen finden für die Programmierung von Benutzeroberflächen in vielen Technologien, beispielsweise WPF, HTML und Android, Anwendung, weswegen Auszeichnungssprachen in diesem Kapitel und den dazugehörigen Unterkapiteln genauer untersucht werden sollen.

4.1 Definition und Funktion von Auszeichnungssprachen

Eine Auszeichnungssprache (zu engl. „Markup Language“) ist eine Computersprache, welche ein Dokument um syntaktische Elemente erweitert, um eine maschinelle Verarbeitung zu ermöglichen [17].

Eine Auszeichnungssprache kann nicht wie bei Programmiersprachen üblich direkt nach Maschinencode kompiliert werden. Stattdessen wird eine Dokument, welches mithilfe einer Auszeichnungssprache erstellt wurde von einem Computerprogramm geladen und interpretiert. Das Computerprogramm führt anhand des eingelesenen Dokumentes weitere Computerabläufe aus.

Auszeichnungssprachen werden für die Verlinkung von Dokumenten untereinander, für die Strukturierung von Inhalt innerhalb eines Dokumentes, für den Datenaustausch oder für die Definition von anderen Auszeichnungssprachen eingesetzt [18]. Dabei hat jede Auszeichnungssprache eigene Syntax-Elemente. Häufig kann zwischen einzelnen Syntax-Elementen unformatierter Fließtext stehen.

Auszeichnungssprachen haben in verschiedenen Domänen Anwendung gefunden. Sie werden u.a. eingesetzt für:

- das Schreiben von Textdokumente (z. Bsp. „LaTeX“ oder „Markdown“)
- das Abbilden von Websites (z. Bsp. „HTML“)
- den Datenaustausch zwischen Rechnerprogrammen (z. Bsp. „JSON“ oder „XML“)
- die Beschreibung von Serverkonfigurationen (z. Bsp. „YAML“)
- die Definition der Metadaten einer Datei (z. Bsp. „JSON“ oder „XML“)
- die Graphische Oberflächengestaltung von Anwendungen (z. Bsp. „XAML“)

Diese Liste hat keinen Anspruch auf Vollständigkeit.

Für die Diplomarbeit ist lediglich die letzte Domäne relevant, nämlich für die Programmierung von Oberflächen. In dieser Domäne haben Auszeichnungssprachen eine starke Verbreitung gefunden, weswegen Sie in dieser Arbeit näher betrachtet werden. Die verbleibenden Anwendungsdomänen sollen keine weitere Bedeutung besitzen.

4.2 Vorstellung ausgewählter Auszeichnungssprachen

Es sollen markante Vertreter von Auszeichnungssprachen in diesem Kapitel vorgestellt werden und auf deren Besonderheiten eingegangen werden. Dabei wurden ausschließlich Sprachen ausgewählt, welche für die Anwendungsdomäne der Programmierung von graphischen Benutzeroberflächen relevant sind. Die Auswahl viel dabei auf Sprachen, welche eine allgemeine Präsenz im Internet besitzen oder für die Diplomarbeit relevant sind. Die Auszeichnungssprache „JSON“ viel aufgrund von letzterem beispielsweise nicht mit in die Auswahl, weil sie eher für die Serialisierung von Daten genutzt wird und nicht um Oberflächen zu definieren.

4.2.1 Extensible Markup Language (XML)

XML wurde von dem World Wide Web Consortium (W3C), einer internationalen Community, welche sich zum Ziel gesetzt hat Standards für das Web voranzutreiben, entwickelt [19] [20]. XML basiert dabei auf „Standard Generalized Markup Language“ (SGML) [20], welcher zum Ziel hat zu normieren, wie generalisierte Auszeichnungssprachen aufgebaut sind. SGML ist in der ISO 8879:1986 definiert [21]. Dabei werden in der ISO-Norm zwei Grundsätze zu Auszeichnungssprachen festgelegt:

- Zum einen sollen generalisierte Auszeichnungssprachen deklarativ sein. Dies bedeutet, dass sie nur die Struktur der Dokumente und andere Attribute beschreiben sollen und nicht die konkreten Programmabläufe, welche zum Anzeigen benötigt werden.
- Zum zweiten sollen sie strikt sein, sodass eine prozessgesteuerte Auswertung in Programmen und innerhalb von Datenbanken möglich ist.

Diese beiden Grundsätze spiegeln sich in XML wider.

Zu XML gibt es bereits eine Vielzahl von frei verfügbaren Lernmaterialien, siehe [22], [23], weswegen nur kurz auf markante Besonderheiten der Sprache eingegangen werden soll. XML ist eine Datenserialisierungssprache. Dies bedeutet, dass die Sprache so konzeptioniert wurde, dass ein einfacher Datenaustausch zwischen Geräten möglich ist. Dabei steht beide, die menschliche und die maschinelle Lesbarkeit im Vordergrund.

Jedes Dokument beginnt mit einer XML-Deklaration, in welcher die XML-Version und das Textformat, z. Bsp. „UTF-8“ angegeben werden. Diese Informationen werden ausschließlich für die Verarbeitung des Dokumentes berücksichtigt und haben keine weitere Relevanz. Darauf folgt der eigentliche Inhalt. Für die Beschreibung der Datenstruktur stehen grundsätzlich die Sprachbestandteile Elemente, Attribute und Namensräume zur Verfügung. Elemente repräsentieren dabei ein Datum und können beliebig tief rekursiv ineinander geschachtelt werden. Sie besitzen einen konkreten Typ,

welcher angegeben werden muss. Jedem Element können mehrere Attribute zugeordnet werden. Einzelne Attribute können dabei durch Variablenwerte repräsentiert werden oder Zeichenketten. Jedes Elementes kann, solange keine weiteren Unterelemente folgen, unstrukturierten Text beinhalten. Die Namensräume dienen dem Zweck, bei der Verwendung mehrerer Typen, wobei mindestens zwei den gleichen Namen besitzen, eine Unterscheidung zu ermöglichen.

An dieser Stelle soll auf XML nicht weiter detailliert eingegangen werden, weil er nicht primärer Bestandteil der Diplomarbeit ist. Es kann sich an den verwiesenen Stellen näher darüber informiert werden.

4.2.2 Extensible Application Markup Language (XAML)

XAML ist laut offizieller Entwicklerdokumentation eine von Microsoft entwickelte deklarative Auszeichnungssprache [24]. Auf die deklarative Eigenschaft der Sprache wird später in diesem Kapitel Bezug genommen werden. XAML ist stark in das .NET Framework integriert. Das .NET Framework wurde ebenso durch Microsoft konzeptioniert und implementiert und ist primär auf dem Betriebssystem Windows lauffähig, kann jedoch auch auf andere Plattformen portiert werden [25]. In diesem Rahmen verfolgt die Sprache sowie das Framework den primären Zweck eine Entwicklung von Applikationen zu vereinfachen. XAML hat dabei den Hauptnutzen visuell sichtbare, graphische Elemente von dem eigentlichen Programmcode zu trennen. Dies ermöglicht, dass unterschiedliche Parteien zum einen für die Definition der graphischen Benutzerschnittstelle und zum anderen für die Implementation der Programmfunktionalität gleichzeitig an dem Produkt arbeiten können [24].

XAML-Dateien sind gleichzeitig XML-Dateien mit der Endung „.xaml“ [24]. XAML kann daher als Untermenge von XML angesehen werden. Ähnlich wie bei XML ist der XAML-Standard relativ weit verbreitet und Microsoft stellt eine große Fülle an Lehrmaterialien und Dokumentation zur Verfügung [26], weswegen ebenso lediglich markante Besonderheiten der Auszeichnungssprache hervorgehoben werden sollen.

Eine der Besonderheit von XAML ist dabei die Verbindung zu dem dahinterliegenden Quellcode. Über das Muster der Datenbindung, welches bereits in Kapitel 3.1.1 Formulare und Kontrollelemente vorgestellt wurde, wird eine Anbindung zu der Programmfunktionalität hergestellt. Dabei existiert für jede XAML-Datei ein Objekt in der dahinterliegenden Programmiersprache, an welches die Datenstruktur gebunden wird. Dies bedeutet, dass sobald Änderungen an diesem Objekt durchgeführt werden, diese 1:1 auf die Benutzeroberfläche überführt werden. Es spielt dabei keine Rolle, ob die Änderungen außerhalb des Objektes oder innerhalb angestoßen werden. Ein Einsatz des Softwarekonzeptes MVVM, siehe Kapitel 3.1.4 Model-View-Viewmodel (MVVM), ist daher naheliegend für eine Applikationsentwicklung mit XAML.

Zusätzlich zu XML verfügt XAML über spezifische Sprach-Erweiterungen, welche für Elemente und Attribute genutzt werden können. Dabei werden sie durch geschweifte Klammern ‚{' bzw. ‚}' gekennzeichnet und heben sich so eindeutig von den aus XML üblichen Attribut- und Elementwerten ab, welche weiterhin genutzt werden können. Die zusätzlichen Spracherweiterungen in XAML ermöglichen unter anderem folgende Zwecke [27]:

- statische Variablenwerte innerhalb der Auszeichnungssprache erstellen
- Arrays innerhalb der Auszeichnungssprache erstellen
- auf statische oder dynamische Ressourcen in den Softwarecode verweisen
- eine Datenbindung zu Variablenwerte des dahinterliegenden Objektes herstellen
- auf ein allgemeine Darstellungsform der Applikation verweisen und nutzen

Für eine vollständige Liste sei an dieser Stelle auf den entsprechenden Abschnitt in der Entwicklerdokumentation von XAML verwiesen, siehe [27].

Es ist in XAML ebenso möglich, Quellcode zu hinterlegen und auszuführen, beispielsweise bei einem Klick-Ereignis [28] oder darauf zu verweisen. Dadurch ist in XAML nicht nur eine Auszeichnungssprache, sondern es kann ebenso in einer XAML-Datei Programmfunktionalität hinterlegt werden. Es ist daher festzuhalten, dass die deklarative Eigenschaft, welche von XML und SGML, siehe Kapitel 4.2.1 Extensible Markup Language (XML) geerbt wurde, teilweise aufgebrochen wurde. Die Beschreibung in der Entwicklerdokumentation erwähnt diesen Aspekt nicht, weswegen an dieser Stelle explizit darauf hingewiesen wird.

XAML enthält noch weitere Funktionalität, wie beispielsweise ausgeprägte Template-Systeme, welche dafür genutzt werden können, um auf abstrakte Weise die Gestaltung von generische Elemente, daher Elemente welche nicht im Konkreten bekannt sind, zu bestimmen [29]. Für die Diplomarbeit ist jedoch dieses Wissen nicht weiter relevant, weswegen lediglich darauf verwiesen wird.

4.2.3 Hypertext Markup Language (HTML)

HTML ist die Kernsprache des Internets [30]. Nahezu jede Website nutzt HTML. Der Standard für diese Auszeichnungssprache ist dabei ebenso wie XML von dem World Wide Web Consortium (W3C) vorgeschlagen worden. Die Auszeichnungssprache weist Ähnlichkeit zu XML auf, ist jedoch deutlich weniger restriktiv.

Die Auszeichnungssprache ist stark durch die Verwendung im World Wide Web (WWW) geprägt. HTML wird von einer Vielzahl von Browsern, beispielsweise Firefox, Google Chrome, Internet Explorer, Opera oder Safari unterstützt. Dabei wird in HTML der Grundaufbau einer Website definiert, welcher von den jeweiligen Browsern angezeigt wird. Aus dieser Domäne heraus besteht das Ziel, dass trotz der unterschiedlichen Verarbeitungsalgorithmen der jeweiligen Browser der Benutzer

möglichst immer ein gewünschtes Ergebnis erhält. Dabei versuchen die Browser, sogar fehlerhafte Dokumente oder nicht bekannte Funktionalität möglichst gut dem Nutzer darzustellen. Die Fehlertoleranz hat eine hohe Priorität, was schlussfolgerlich auch dazu führen kann, dass ein Fehler in einer HTML-Seiten nicht von den Entwicklern selber wahrgenommen wird.

Es wurde eine Abwandlung des HTML Standards geschaffen, um eine Antwort auf dieses Problem zu finden. Die Abwandlung des Standards verfügt die gleiche Restriktion wie XML und trägt den Namen „Extensible Hypertext Markup Language (XHTML) [31]. XHTML kann somit als Untermenge von XML gesehen werden. Dieser Standard wird wie HTML ebenso von gängigen Browsern unterstützt [31]. Dabei hat jedoch der HTML Standard derzeit eine deutlich größere Verbreitung [32].

Die Besonderheit von HTML besteht darin, dass es viele Elemente und Attribute enthält, welche für die Gestaltung von Websites geeignet sind. Es gibt bereits eine Fülle an Material zu der Sprache, weswegen nicht weiter auf syntaktische Elemente eingegangen werden soll, jedoch wurde eine kurze Betrachtung dieser Sprache mit in ein Unterkapitel in 4.2 Vorstellung ausgewählter Auszeichnungssprachen aufgenommen, weil die Sprache durch den Aufstieg des Internets eine sehr weite Verbreitung gefunden hat und deswegen erwähnt werden sollte.

4.2.4 YAML Ain‘t Markup Language (YAML)

YAML ist eine benutzerfreundliche Datenserialisierungs-Sprache, welche ursprünglich von Oren Ben-Kiki, Clark Evans und Ingy döt Net entworfen wurde. Als Datenserialisierungs-Sprache gibt es Anbindungen zu verschiedenen Programmiersprachen, welche den Standard nutzen können. Sie ist mit Focus auf gewöhnliche native Elemente entworfen worden [33]. Das Attribut „benutzerfreundlich“ bezieht sich dabei auf die menschliche Lesbarkeit der Dokumente relativ im Verhältnis zu anderen Auszeichnungssprachen. Gerade durch die einfache menschliche Lesbarkeit hebt sie sich von anderen Auszeichnungssprachen ab. Sie findet vor allem im Bereich der Serverkonfiguration derzeit eine hohe Anwendung, beispielsweise für Kubernetes, einem modernen System für die Verwaltung von Container-basierten Applikationen über mehrere Rechner [34], [35].

Es soll auf markante Elemente der Sprache eingegangen werden. Als Grundlage soll folgendes einfaches Beispiel dienen, welches aus einer Quelle, siehe [36], übernommen wurde:

```
---
id: 731
date: 16th of May 2011
customer: 17
items:
- quantity: 5
```

```
  description: Oolong
  price: 5.98
  in-stock: true

- quantity: 2
  description: Assam
  price: 2.95
  in-stock: false
```

Über den 3 Anstrichen ‚---‘ können Metainformationen zu dem YAML-Dokument stehen. Dies ist jedoch für die Diplomarbeit nicht weiter relevant, weswegen darauf nicht weiter eingegangen wird. Man kann in dem Beispiel erkennen, dass ein YAML-Dokument im Kern aus 3 Elementtypen besteht: Maps, Arrays und Skalar-Werten. Ein YAML-Dokument besteht dabei aus einer Map, einem Array oder einem Skalar-Wert. Eine Map besteht aus Schlüssel-Werte-Paaren. Dabei kann jeder Schlüssel und jeder Wert wiederum eine Map, ein Array oder ein Skalar-Wert sein. In dem Beispiel sind als Schlüssel lediglich Skalar-Werte angegeben. Schlüssel werden von Werten mithilfe eines Doppelpunktes ‚:‘ voneinander getrennt. Ein Array besteht aus beliebig vielen Maps, Arrays und Skalar-Werten. Jedes Element eines Arrays wird dabei über einen Anstrich ‚-‘ angegeben. Jeder Wert einer Map und jedes Array-Element muss auf fortlaufenden Zeilen eingerückt werden. Ein Skalar-Wert besteht aus einem Ausdruck. Dieser Ausdruck kann beispielsweise ein boolescher Wert, eine Zahl oder ein Text sein.

In weiteren Kapiteln wird sich auf Map(s), Array(s) und Skalar-Werte über YAML-Map(s), YAML-Array(s) bzw. YAML-Skalar-Wert(e) bezogen. Ebenso findet eine Referenzierung von Schlüsseln und Werten einer YAML-Map über YAML-Map-Schlüssel bzw. YAML-Map-Wert statt.

Es gibt in dem YAML-Standard noch weitere grundlegende Elemente, wie beispielsweise Anker und Textblockabsätze, welche für diese Arbeit aber keine Relevanz haben und deswegen an dieser Stelle nicht weiter aufgeführt werden.

4.2.5 Qt Modeling Language (QML)

QML ist eine deklarative Programmiersprache

4.3 Problematik der Typisierung bei Auszeichnungssprachen

Auszeichnungssprachen sollen möglichst strikt sein, siehe dazu Kapitel 4.2.1 Extensible Markup Language (XML). Aus programmiertechnischer Sicht soll jedoch Quellcode im Allgemeinen und auch bei der Definition einer Datenstruktur innerhalb einer Auszeichnungssprache weitestgehend aussagekräftig sein. Überflüssiger Programmcode, welcher keinen weiteren Informationsgehalt bietet, lässt eine Datei nur

unnötig aufblähen. Die für den Programmablauf relevanten Teile sind schwieriger zu finden und der redundante Informationsgehalt bleibt nur unnötig bestehen. Der Wunsch nach Striktheit in einer Auszeichnungssprache und Ausdrucksstärke in Dokumenten resultiert in einen Widerspruch, welcher in der Thematik der Typisierung deutlich wird. Typisierung ist dabei bezogen auf die Bestimmung eines konkreten Typs für ein Element. Die Aufgabe besteht deswegen darin, für ein innerhalb einer Auszeichnungssprache definiertes Element den Typ zu bestimmen.

Dabei gibt es Auszeichnungssprachen

5 Bewertung von bisherigen Technologien zur Eignung für hardwarenahe Benutzeroberflächenerstellung

In diesem Kapitel sollen bisherige Technologien für die Programmierung von Benutzeroberflächen analysiert werden und hinsichtlich ihrer Eignung für die hardwarenahe Programmierung ausgewertet werden. Als Mittel, um dies zu bewerkstelligen wurde der Vergleich gewählt.

5.1 Bewertungskriterien

Die Bewertungskriterien werden sowohl aus Anforderungen für die Programmierung von graphischen Benutzeroberflächen, als auch aus Anforderungen für die hardwarenahe Programmierung abgeleitet. Deswegen werden diese beiden Bereich zunächst separiert betrachtet. siehe: Anforderungen an die Programmierung von Benutzeroberflächen

5.2 Anforderungen für die Programmierung von graphischen Benutzeroberflächen

- Formulare und Kontrollelemente
- MVP
- MVC
- MVVM
- No Vendor Lock-In [37]
- Open Source?

5.3 Anforderungen für hardwarenahe Programmierung

- Verzicht auf Garbage-Collector
- Portabilität für Cross-Plattform-Entwicklung
- Zugriff auf statische Resourcen

5.4 Ausgewählte Technologien

[Zu jeder Möglichkeit min. einen prominenten Vertreter auswählen, Technologien kurz vorstellen]
- QT
- WPF mit XAML
- Xamarin mit XAML

- React Native (Javascript + HTML + CSS)

5.5 Vergleich

[Punkte vergeben, jede Punktevergabe kurz begründen]
[inklusive kurzes Fazit]

6 Prototypische Entwicklung

6.1 Anforderungen an den Prototyp „XXX“

Ziel der Diplomarbeit ist es neue Ansätze für die hardwarenahe Programmierung von Benutzeroberflächen zu erforschen. Diese Ansätze sollen mithilfe eines Prototypen praxisnah veranschaulicht werden. Im Rahmen der Anforderungsanalyse konnten grundsätzlich folgende Bereich von Interesse ermittelt werden:

- Smart-Home-Anwendungen
- Anwendungen für Automaten jeglicher Art
- Anwendungen für Bürogeräte
- Mobile Anwendungen

Smart-Home-Anwendungen

Unter den Begriff „Smart-Home-Anwendungen“ wird jegliche Software gezählt, welche lokal vor Ort bei Endnutzer auf den Geräten installiert ist, die zum Zweck hat hauswirtschaftliche Abläufe zu steuern und zu automatisieren. Dazu zählt unter anderem die Kontrolle der Heizung, der Fenster, der Türen, sowie sonstigen Haushaltsgeräten, wie beispielsweise Kühlschrank, Geschirrspüler und Waschmaschine.
Für den Prototyp sind Smart-Home-Anwendungen interessant, welche über eine graphische Oberfläche mit dem Endnutzer interagieren.

Anwendungen für Automaten jeglicher Art

Unter „Automaten jeglicher Art“ fallen sämtliche Automaten, welche nach Anbindung an ein Softwaresystem direkt nutzbar sind und über einen vordefinierten Funktionsumfang verfügen. Darunter fallen beispielsweise Fahrkartenautomaten, Getränkeautomaten und Selbstbezahlungsautomaten.

Anwendungen für Bürogeräte

Mit Bürogeräten werden Geräte für den Bürobedarf abgedeckt. Darunter fallen Drucker und Scanner.

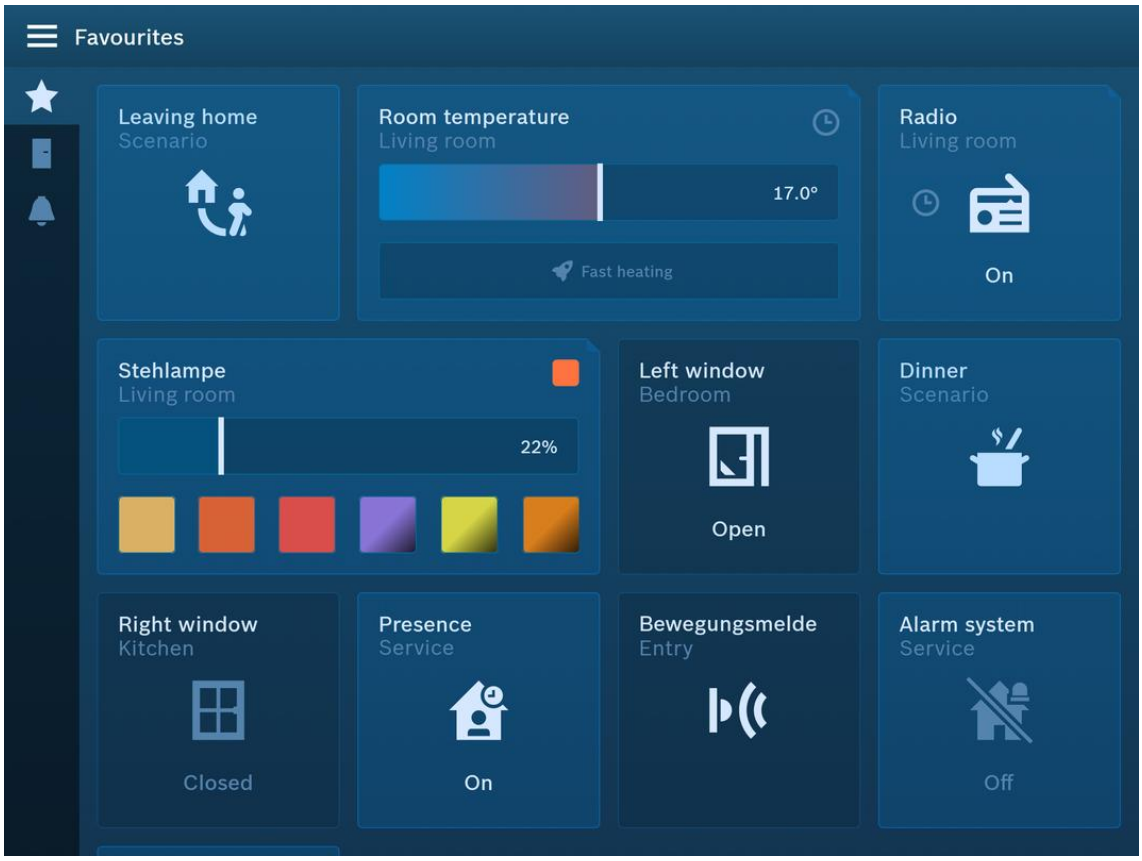
Mobile Anwendungen

Mobile Anwendungen sind Anwendungen für Geräte, welche ohne erheblichen Mehraufwand im Alltag mitgeführt werden können. Darunter fallen Smartphones und Smartwatches. Nicht hinzu zählen Notebooks.

Alle Bereiche eint, dass stets limitierte Hardware zu berücksichtigen ist. Der Prototyp soll exemplarisch häufig vorkommende Funktionalität in der Benutzerführung abbilden.

Dementsprechend müssen funktionale und nicht-funktionale Anforderungen festgelegt werden.

Um eine möglichst genaue Abbildung der Anforderungen aus der Praxis zu gewährleisten, wurde als Grundlage ein Screenshot der Applikation "Bosch Smart Home" von der Robert Bosch GmbH für Android gewählt, siehe Abbildung 6. Dieser Screenshot wird aktuell benutzt, um die Applikation im Google Play Store zu vertreiben und deckt damit sowohl das Feld „Mobile Applikationen“, als auch „Smart-Home-Anwendungen“ ab.



6. Abbildung: Screenshot der Android-Applikation "Bosch Smart Home" von der Robert Bosch GmbH

Die Anforderungen wurden anhand des Screenshots festgelegt.

6.1.1 Funktionale Anforderungen

/101/ Overlay:

Der Prototyp besteht aus einem Overlay. Innnerhalb des Overlays müssen Titelleiste, Navigationsbar und Inhalt angeordnet werden.

/102/ Titelleiste:

Die Titelleiste muss aus einem Symbol bestehen und einem Textbanner. Das Symbol muss klick-bar sein. Bei Klick muss in der Konsole eine Benachrichtigung erfolgen.

/103/ Navigationsbar:

Die Navigationsbar muss aus einem Tab "Favoriten" und weiteren Tabs bestehen. Der Tab "Favoriten" ist bei Start der Software ausgewählt. Jeder Tab ist durch ein unterschiedliches Symbol dargestellt. Bei Klicken eines Tabs muss sich die Titelleiste entsprechend ändern, sowie der Inhalt angepasst werden. Es muss visuell markiert sein, welcher Tab ausgewählt ist.

/104/ Inhalt:

Der Inhalt ist leer, solange ein anderer Tab als "Favoriten" ausgewählt ist. Falls der Tab "Favoriten" in der Navigationsbar ausgewählt ist, so besteht der Inhalt aus einem Scrollview, welcher ein Grid-Layout beinhaltet. In dem Gridlayout müssen unterschiedlich große Kacheln angeordnet werden.

/201/ Kacheln [TODO]:
[An dieser Stelle würde ich die Kacheln, welche man auf dem Screen sieht definieren, sowie eine Kachel "Add to favorites", welche bei klick ebenso eine Meldung in der Konsole wirft.]

Leaving home - Einfache Meldung in der Konsole
Room temperature - Leiste für temperature. Wenn fast heating aktiviert wird, steigt temperatur simulationsmäßig langsam an.
Radio - on/off switch
Stehlampe - Scrolleiste für Helligkeit in %. Farbauswahl.
Left window - open/close switch, evt. mit verzögerung
Dinner - Einfache Meldung in der Konsole
Right window - open/close switch
Presence - on/off switch
Bewegunsmelder - statisches feld
Alarm system - on/off switch

6.1.2 Qualitätsanforderungen

/301/ Visuelle Aufbereitung:

Der Prototyp muss ansprechend visuell aufbereitet sein.

6.1.3 Rahmenbedingungen

Dieses Kapitel umfasst die technisch/technologischen Rahmenbedingungen.

6.1.3.1 Technisch/ Technologische Rahmenbedingungen

/401/ Statische Favoriten:

Die Favoriten dürfen für den Prototyp als statisch angenommen werden und müssen nicht dynamisch geladen werden.

/402/ Portabilität:

Die Portabilität der zugrunde liegenden Programmiersprache soll gewährleistet bleiben.

/403/ Hardwarenahe Programmierung:

Die Programmierung muss mit einer hardwarenahen Programmiersprache, wie „C“ oder ähnlich umgesetzt werden.

6.1.3.2 Rechtliche Rahmenbedingungen

6.1.3.3 Organisatorische Rahmenbedingungen

6.2 Lösungskonzepte bzgl. der Technologie

In diesem Kapitel sollen die Lösungskonzepte bzgl. der genutzten Technologie für den Prototypen vorgestellt werden.

6.2.1 Verwendete Technologien und Software-Bibliotheken

Um das Rad nicht neu zu erfinden, wurde für den Prototypen auf verschiedene Technologien und Software-Bibliotheken zurückgegriffen. Diese sollen in diesem Kapitel vorgestellt werden.

6.2.1.1 CMake

CMake ist eine Kollektion an Werkzeugen, welche erstellt wurde um Software zu kompilieren, zu testen und zu archivieren. CMake ist dabei Compiler- und Plattform-unabhängig und unter einer Open-Source-Lizenz verfügbar [38].

Für das Projekt wurde CMake Version „3.12.1“ verwendet für jenen Zweck. Dadurch kann eine IDE- und Plattform-Unabhängigkeit gewährleistet bleiben. Ebenso können über CMake verschiedene Code-Generierungsskripte in den Compilierungsprozess integriert werden. Diese Funktionalität wurde ebenso genutzt.

6.2.1.2 Imps Engine, Expat, SFMT

Die Imps Engine ist eine Sammlung von Software-Bibliotheken für die Plattform-unabhängige Entwicklung von Applikationen in der Programmiersprache „C“ [39]. Sie ist verfügbar unter einer Open-Source-Lizenz, der MIT-Lizenz.

Die Imps Engine hat als interne Abhängigkeiten Expat und SFMT, welche durch die Benutzung vererbt werden, aber keine weitere Verwendung im Quellcode des Prototypen finden. Die beiden Bibliotheken sollen nur kurz angerissen werden. Expat ist eine C-Software-Bibliothek um XML-Dokumente zu parsen, siehe [40]. SFMT (SIMD oriented Fast Mersenne Twister, wobei SIMD für „Single Instruction Multiple Data“ steht) ist eine C-Software-Bibliothek für die Erzeugung von Pseudo-Zufallszahlen [41]. Dazu wurde eine Variante des Mersenne Twister (MT) implementiert, welche von Mutsuo Saito und Makoto Matsumoto im Jahr 2006 vorgestellt wurde. Es verspricht eine bessere Performance im Vergleich zu ähnlichen Zufallsgeneratoren [42]. Die Generierung von Zufallszahlen ist für die Diplomarbeit jedoch nicht weiter Relevant, weswegen auf den Algorithmus nicht weiter eingegangen wird.

6.2.1.3 OpenCV

OpenCV (Open Source Computer Vision Library) ist ebenso unter einer Open-Source-Lizenz, der BSD-Lizenz verfügbar [43]. Sie wurde genutzt, um das Laden von Bilddateien zu ermöglichen. Die Software-Bibliothek ist für die rechnergestützte Erkennung von Bildern optimiert, wodurch sie wesentlich umfangreicher ist und nur ein minimaler Teil des Umfangs genutzt wurde. Es besteht die Möglichkeit, den Umfang des Prototypen zu reduzieren, indem das Laden der Bilddateien durch eine andere

Software-Bibliothek unterstützt wird. Jeglicher Quellcode dieser Funktionalität, welcher ersetzt werden müsste, befindet sich dabei unter „deps/opencv-glue“.

6.2.1.4 OpenGL und GLEW

OpenGL ist ein Graphikkarten Standard für hohe graphische Qualität und Performance [44]. Er wird genutzt, um graphische Elemente darzustellen. Um die Entwicklung zu vereinfachen, werden Hilfsobjekte aus der Imps Engine genutzt.

GLEW (OpenGL Extension Wrangler Library) ist eine Plattform-unabhängige Open-Source C/C++ Software-Bibliothek für das Laden von OpenGL-Erweiterungen. Sie ist dafür geeignet, zur Laufzeit zu bestimmen welche OpenGL Versionen unterstützt werden und welche Erweiterungen verfügbar sind und wurde für diesen Zweck genutzt [45].

6.2.1.5 CLion, MinGW, Windows 8.1

Für die Entwicklung wurde die IDE „CLion“ von JetBrains genutzt, siehe [46]. CLion ist eine kommerzielle Anwendung, spezialisiert für die Programmierung in C und C/C++, inklusive nativem Support für CMake. Die Versionsnummer lautet „2018.2.3“.

Als Compiler wurde MinGW verwendet, siehe [47], Version „w64 6.0“. MinGW beinhaltet den Debugger „MinGW-w64 GDB“ mit Versionsnummer „8.1“.

Die Plattform-Umgebung ist Windows 8.1. Aufgrund der Tatsache, dass CMake verwendet wurde ist es kein Problem, IDE, Compiler oder Plattform auszuwechseln. Lediglich bei einer Änderung der Plattform müssen Windows-spezifische Teile angepasst werden, beispielsweise für das Laden von Bildern. Diese Anpassungen sind für Android und iOS bereits in der Imps Engine enthalten und müssen lediglich korrekt verlinkt werden.

6.2.2 Objektorientierte Programmierung in C

Der Prototyp repräsentiert die praktische Anwendung von objektorientierter Programmierung (OOP) in der Programmiersprache C als hardwarenahe Programmiersprache. Dabei bietet die Programmiersprache C native keine Unterstützung für OOP [48].

Es ist festzuhalten, dass OOP nicht an syntaktische Elemente gebunden ist, sondern lediglich ein Programmierparadigma darstellt, nach welchem Daten und zugehöriger

Code bzw. Funktionen gekapselt werden [49]. Die Objekte kommunizieren dabei über Funktionen miteinander.

Deswegen ist es grundsätzlich kein Problem, OOP in C durchzuführen. Es ist sogar eine Tatsache, dass populäre Bibliotheken in C, wie beispielsweise POSIX Threads [50] einen objektorientierten Aufbau verfolgen, siehe dazu die Spezifikation [51]. Man kann erkennen, dass Funktionen zu einem bestimmten Datentyp, wie „pthread_mutex_t“, über die Namensgebung, in diesem Falle „pthread_mutex*“, gruppiert und auf diese Weise an den Datentyp gekapselt werden. Daher ist diese Bibliothek mindestens teilweise von einem objektorientierten Aufbau geprägt.

Das Paradigma OOP bietet jedoch nicht nur die Kapselung von Daten und Funktionen, sondern auch erweiterte Konzepte und Muster, wie beispielsweise Vererbung oder bzw. das Beobachter-Muster. Um diese Funktionalität bereitzustellen, wurde die Imps Engine verwendet. Über eine Reihe von Annotationen in den Kommentaren können reguläre C-Structs mit Objektfunktionalität aufgewertet [52]. Dies wurde für die Diplomarbeit genutzt.

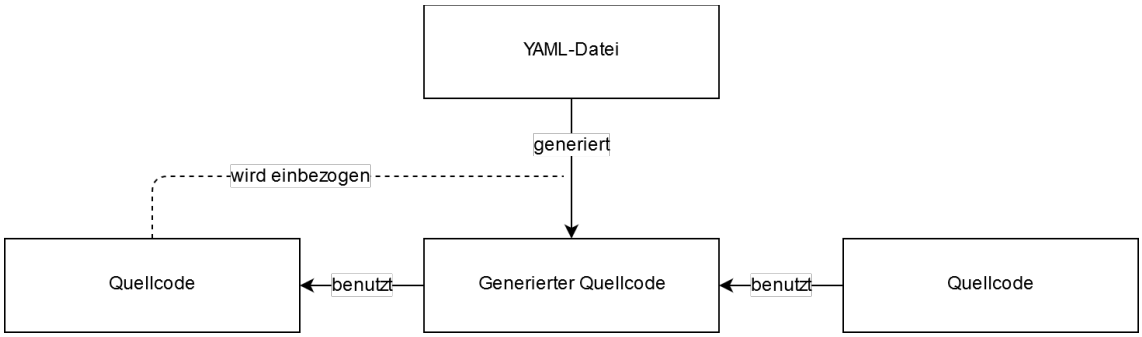
Todo: Header und C Datei example. Vererbung und Beobachter-Muster am Beispiel erklären, Annotations genauer erklären, Standardaufbau von Klassen, Namespaces, (Optionale) Attribute für Klassen und Funktionen

Da OOP in der Programmiersprache C nicht primärer Forschungsgegenstand ist, sondern lediglich für die Lösungskonzepte Anwendung findet, soll an dieser Stelle nicht weiter darauf eingegangen werden.

6.2.3 Verwendung einer Auszeichnungssprache

Es soll für die graphische Oberflächengestaltung eine Auszeichnungssprache angewendet werden. Als Grundlage dient der YAML-Standard. Dieser wird weiter eingeschränkt.

Das Ziel ist es, über die Auszeichnungssprache eine Objekterstellung zu ermöglichen. Dazu soll das YAML-Dokument in C-Code übersetzt werden und innerhalb des Programms aufgerufen werden können, um die spezifizierte Oberfläche zu erstellen, wie die 7. Abbildung: Übersicht Integration der Auszeichnungssprache in den Quellcode darstellt.



7. Abbildung: Übersicht Integration der Auszeichnungssprache in den Quellcode

Wie in der Abbildung zu erkennen ist, wird aus der YAML-Datei neuer Quellcode generiert. Der generierte Quellcode kann von außerhalb eingebunden und verwendet werden. Dies ist durch den Block „Quellcode“ rechts unten in der 7. Abbildung visualisiert. Weiterhin ist festzuhalten, dass in den Schritt der Generierung zugrunde liegender Programmcode mit einbezogen wird. Dies bedeutet, dass die Funktionalität, welche bereits implementiert wurde, Auswirkungen auf das Generierung-Verhalten hat.

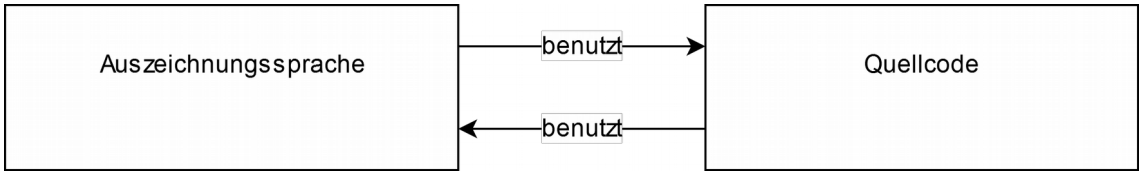
Dieser Umstand lässt sich einfach erklären. Wie bereits genannt, soll durch die Auszeichnungssprache eine Objekterstellung ermöglicht werden. Damit die Objekte entsprechend erstellt werden können, ist es zwingend notwendig, Kenntnis über den Aufbau der Klassen zu besitzen. Diese Kenntnis, welche benötigt wird besteht im Großteil, aber nicht ausschließlich, aus dem Wissen über die Art und Weise, wie ein Objekt dieser Klasse instanziiert werden kann, sowie welche Attribute diese Klasse besitzt, welche davon zwingend benötigt werden und welche optional sind. Diese Informationen werden für die Generierung des Programmcodes genutzt, damit die Objekte erstellt werden können.

Die Brillianz in diesem Lösungskonzept steckt in der Tatsache, dass anhand des objektorientierten Aufbaus der Klassen diese Kenntnis automatisch aus dem Quellcode gewonnen werden kann. Dies wird durch ein Perl-Script ermöglicht. Dieses Perl-Script liest spezifizierte Header-Dateien des Programms ein und interpretiert diese. Die Header-Dateien sind mit in dem Build-Tool „CMake“ hinterlegt. Es findet also auch keine Dopplung der Spezifizierung der genutzten Header-Dateien statt, weil sie lediglich in dem Build-Tool hinterlegt sind, welches für die Programmerstellung genutzt werden soll. Ein starker Vorteil dieser Lösung im Vergleich zu analysierten technologischen Lösungen besteht darin, dass wenn ein Klasse für die Auszeichnungssprache genutzt werden soll, kein zusätzlicher Anpassungsaufwand notwendig ist. Die Art und Weise, wie diese Klasse über programmiertechnischen Weg erstellt wird, wird 1:1 genutzt. Dies setzt einen gewissen Standardaufbau von Klassen vorraus, welcher durch die Nutzung der Imps Engine gegeben ist.

Der Spezifikation der Auszeichnungssprache wird in einem Unterkapitel genauer beschrieben, weswegen an dieser Stelle nicht weiter darauf eingegangen wird.

6.2.4 Auflösung der gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode

Innerhalb eines YAML-Dokumentes soll keine Referenzierung auf den aufzurufenden Code durchgeführt werden. Dies impliziert, dass Funktionsreferenzen, wie beispielsweise eine „onClick“-Methode eines Buttons, nicht direkt verlinkt werden können. Hintergrund dazu ist folgender: Bei allen analysierten bisherigen Lösungen, z.B. XAML oder QML kann eine gegenseitige Referenzierung zwischen Auszeichnungssprache und Quellcode erfolgen, siehe 8. Abbildung: Darstellung der ungünstigen gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode.

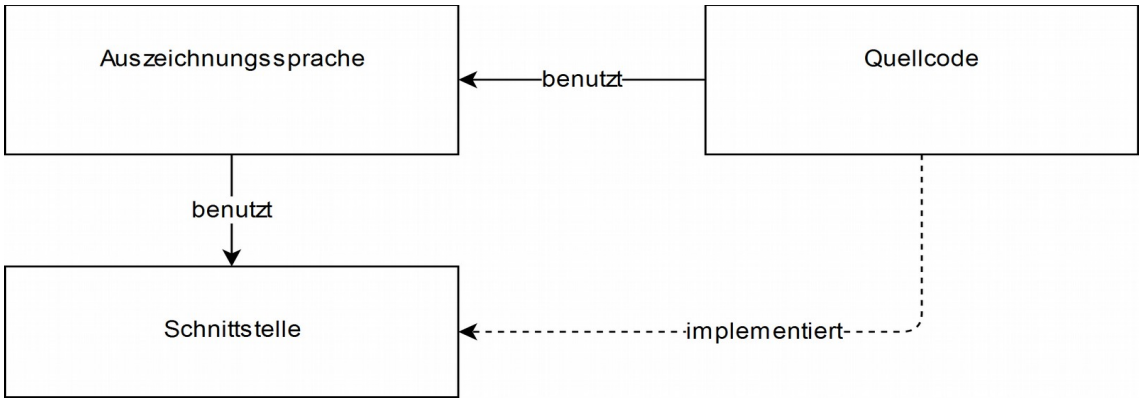


8. Abbildung: Darstellung der ungünstigen gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode

Aus softwaretechnischer Sicht ist dies gefährlich, weil es jede Datei aus der Auszeichnungssprache fest an eine mindestens eine Datei aus dem Quellcode bindet. Man kann daher beide Bestandteile nicht getrennt voneinander wiederverwenden, sondern immer nur im ganzen Block. Dieser Fall ist bei weitem nicht theoretisch. Ein erstellter View kann nur im Zusammenhang mit dem dahinter liegendem Quellcode verwendet werden. Wenn er mit verschiedener Funktionalität versehen werden soll, muss er entweder dupliziert werden oder es muss die nötige Abstraktion im Quellcode geschaffen werden.

Ein weiterer Nachteil, welcher sich aus diesem Aufbau heraus ergibt ist, dass in den Auszeichnungssprachen automatisch Quellcodefunktionalität enthalten sein muss, um die entsprechenden Aufrufe zu tätigen. Der Nachteil besteht darin, dass ein Designer, welcher möglicherweise etwas in der Datei anpassen will und kein programmiertechnisches Fachwissen hat unnötigerweise mit Codefunktionalität auseinandergesetzt wird.

Um diese Nachteile zu umgehen wird folgende Auflösung vorgeschlagen, siehe 9. Abbildung: Auflösung der gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode durch Schnittstelle.



9. Abbildung: Auflösung der gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode durch Schnittstelle

Die Grundidee besteht darin, eine gegenseitige Referenzierung über das Einführen einer Schnittstelle auszuhebeln. Dabei kennt der Quellcode sowohl Schnittstelle als auch Auszeichnungssprache und implementiert die Schnittstelle, in diesem Fall die „onClick“-Methode. Die Auszeichnungssprache muss nun lediglich auf die Schnittstelle zurückgreifen und benötigt kein Wissen über den dahinterliegenden Quellcode.

Festzuhalten ist, dass es keine direkten Aufrufe von der Auszeichnungssprache in den Quellcode geben darf, welcher die Objekterstellung ausführt und damit die Auszeichnungssprache kennt. Es dar lediglich Funktionalität angeboten werden, mit deren Hilfe sich der Quellcode die gewünschte Funktionalität an gewünschter Stelle über eine Schnittstelle nutzen darf.

Der Vorteil besteht darin, dass die Auszeichnungssprache wiederverwendet werden kann unabhängig vom Quellcode. Softwaretechnisch ist dies sauberer und damit besser.

6.2.5 Spezifikation der Auszeichnungssprache

Es wurde bereits im Kapitel 6.2.3 die grobe Funktionsweise der Auszeichnungssprache erklärt. In diesem Kapitel soll auf den konkreten Aufbau der Auszeichnungssprache genauer eingegangen werden und eine Grammatikspezifizierung durchgeführt werden. Ergänzend zu Kapitel 6.2.3 soll festgehalten werden, dass die Objekterstellung mithilfe der Auszeichnungssprache vollständig rekursiv möglich sein soll. Dies bedeutet, dass Objekte beliebig tief ineinander geschachtelt werden können sollen.

Die Grammatik wird dabei wie folgt festgelegt und näher erklärt. Dabei stellen nicht rekursiv geschriebene Wörter die Definition eines Grammatikelements dar. Der Zuweisungsoperator „:=“ ist dabei zu lesen als „besteht aus“, sofern dies möglich ist. Zusätzlich liegen durch den dahinterliegenden Programmcode weitere Einschränkungen vor. Es wurde aus Gründen der Übersichtlichkeit darauf verzichtet, diese

Einschränkungen in die Grammatikdefinition aufzunehmen. Stattdessen wird darauf innerhalb der Erklärungen eingegangen:

Dokument := *YAML-Map-Schlüssel['__generate__'] : Objekt-Typ, (YAML-Map-Schlüssel : Element)**
Objekt-Typ := *Name*
Name := *\w[\w\d]**
Element := *Objekt | Array | Struktur | Konstanter-Ausdruck*
Objekt := *(YAML-Map-Schlüssel['__type__'] : Objekt-Typ)?, (YAML-Map-Schlüssel['__ref__'] : Objekt-Referenz)?, (YAML-Map-Schlüssel : Element)** | *YAML-Map-Schlüssel['__resource__'] : Resource-Provider, (YAML-Map-Schlüssel : (Objekt | Struktur | Konstanter-Ausdruck))**
Resource-Provider := *Name*
Objekt-Referenz := *Name*
Array := *YAML-Array[Array-Objekt]*
Array-Objekt := *(YAML-Map-Schlüssel : Element)**
Struktur := *(YAML-Map-Schlüssel : Konstanter-Ausdruck)** | *YAML-Array[Konstanter-Ausdruck]* | *Konstante*
Konstanter-Ausdruck := *Zeichenkette* | *Nummerischer-Ausdruck* | *Boolescher-Ausdruck* | *Konstante*
Zeichenkette := *Name*
Nummerischer-Ausdruck := *\d+(\s*[\+\-*\/]\s*\d+)**
Boolescher-Ausdruck := *true* | *false*
Konstante := *Name*

Festzuhalten ist, dass die Verarbeitung der Auszeichnungssprache kontextorientiert erfolgt. Dies bedeutet, abhängig davon in welchem Kontext der Parser sich befindet sind unterschiedliche grammatikalische Ausdrücke (GA) zulässig und werden anders interpretiert. Der Kontext wird anhand der vorhandenen Klassen und Funktionalitäten bestimmt. Wie der Grammatikdefinition zu entnehmen ist, können teilweise mehrere unterschiedliche GA aufeinander folgen. In diesem Fall hat der Kontext den entscheidenden Einfluss. In der folgenden Erklärung zu den grammatikalischen Ausdrücken wird auf den Kontext an den entsprechenden Stellen im Einzelnen eingegangen.

Dokument := *YAML-Map-Schlüssel['__generate__'] : Objekt-Typ, (YAML-Map-Schlüssel : Element)**

Ein Dokument besteht aus einer YAML-Map. Die YAML-Map muss dabei aus einem Schlüssel „__generate__“ bestehen, welcher auf den GA „Objekt-Typ“ referenziert. Anschließend können beliebig viele weitere Map-Einträge folgen. Diese müssen auf den GA „Element“ verweisen. Der Objekt-Typ stellt dabei die initiale Klasse dar, welche instanziiert werden soll. Die Schlüssel-Werte paare Stellen Attribute dar, welche dem Objekt bei der Erstellung übergeben werden. Dabei sind manche Attribute zwangsweise

notwendig und andere optional. Dieses Wissen wird aus dem Quellcode entnommen. Falls Pflicht-Attribute fehlen oder Attribute nicht zugeordnet werden können, liegt eine ungültige YAML-Datei vor. Dies resultiert in einem Fehler.

Objekt-Typ := *Name*

Ein Objekt-Typ besteht aus einem GA „Name“. Dabei ist festzuhalten, dass der Objekt-Typ einer Klasse im Programmcode entsprechen muss, auf welche ein Objekt-Typ verweist.

Name := *\w[\w\d]**

Ein GA „Name“ besteht aus einem YAML-Skalar-Wert. Dieser Wert beinhaltet ein Zeichen gefolgt von einer Aneinanderreihung von beliebig vielen Zeichen und Ziffern. Dabei ist ein Zeichen definiert aus dem Zeichenraum von A-Z (Großbuchstaben), a-z (Kleinbuchstaben), sowie einem Unterstrich „-“. Eine Ziffer ist definiert aus dem Zeichenraum von 0-9. Dieser Sachverhalt wird in der Definition des GA „Name“ mithilfe einer regulären Expression erklärt.

Element := *Objekt* | *Array* | *Struktur* | *Konstanter-Ausdruck*

Ein GA „Element“ ist im Vergleich zu anderen Grammatik Ausrücken relativ abstrakt. Ein Element kann ein Objekt, ein Array, eine Struktur oder ein konstanter Ausdruck sein. In welche der 4 Kategorien ein Element fällt wird aus dem aktuellen Kontext wie folgt bestimmt. Ein Element wird immer für ein Objektattribut verwendet. Dieses Objektattribut kann ein anderes Objekt sein, ein Array, eine Struktur oder ein konstanter Ausdruck. Je nachdem, welcher Typ verlangt wird, wird das Element entsprechend interpretiert. Die einzelnen grammatikalischen Ausdrücke werden individuell an den entsprechenden Stellen erklärt.

Objekt := *(YAML-Map-Schlüssel['__type__'] : Objekt-Typ)?,*
(YAML-Map-Schlüssel['__ref__'] : Objekt-Referenz)?,
*(YAML-Map-Schlüssel : Element)**
| YAML-Map-Schlüssel['__resource__'] : Resource-Provider,
*(YAML-Map-Schlüssel : (Objekt | Struktur | Konstanter-Ausdruck))**

Der GA „Objekt“ weist eine hohe Komplexität auf, weil er über eine hohe Abstraktion verfügt und auf unterschiedliche Weise verwendet werden kann. Ein Objekt kann auf zwei verschiedene Weisen erstellt werden. Entweder über die Erstellung mithilfe einer Klasse oder über einen Resource-Provider. Welcher Weg dabei gewählt wird ist abhängig davon, ob der YAML-Map-Schlüssel „__resource__“ vorhanden ist. Wenn ja, wird das Objekt über einen Resource-Provider zur Verfügung gestellt. Falls nein, wird

es über eine Klasse instanziiert. Die Instanziierung über eine Klasse erfolgt dabei ähnlich wie beschrieben unter „Dokument“. Sie hat notwendige und optionale Attribute. Diese werden über beliebig viele Schlüssel-Werte-Paare angegeben. Die Paare verweisen auf mehrere „Elemente“. Optional kann über den YAML-Map-Schlüssel „__type__“ ein Objekt-Typ angegeben werden. Zusätzlich optional kann über den Schlüssel „__ref__“ eine Objekt-Referenzierung von außerhalb angegeben werden. An dieser Stelle soll auf beide Konzepte nicht weiter eingegangen werden, da sie Komplex sind und in ein Unterkapitel ausgelagert wurden. Das Resource-Provider Konzept befindet sich ebenfalls in einem eigenen Unterkapitel wieder.

Resource-Provider := *Name*

Ein GA „Resource-Provider“ wird über einen GA „Name“ angegeben. Der Resource-Provider muss dabei im Quellcode hinterlegt sein. Das Resource-Provider Konzept soll in einem Unterkapitel näher erklärt werden.

Objekt-Referenz := *Name*

Ein GA „Objekt-Referenz“ wird über einen GA „Name“ angegeben. Eine Objekt-Referenz ist nur bezogen auf Objekte zulässig. Die Objekte sind von außerhalb referenzierbar. Dieses Konzept wird in einem Unterkapitel näher erklärt.

Array := *YAML-Array[Array-Objekt]*

Ein GA „Array“ besteht aus einem YAML-Array, welcher auf einzelne grammatikalische Ausdrücke „Array-Objekt“ verweist. Der Hintergrund, weswegen ein extra „Array-Objekt“ eingeführt wurde und nicht auf „Objekt“ verwiesen wird, ist durch die Programmiersprache C bedingt. Ein Array in C ist in der Länge fest definiert. Dadurch hat jedes Element eines Arrays ebenso eine feste Länge für einen Datentyp. Aufgrund der Tatsache, dass die Länge des Datentyps und damit des Objektes fest definiert ist, können nicht beliebige Objekte instanziiert werden, sondern nur Objekte von einem Typ. Um diese Einschränkung darzustellen, wurde der GA „Array-Objekt“ eingeführt. Es sei an dieser Stelle erwähnt, dass eine Möglichkeit in Betracht gezogen wurde, wie man diese Einschränkung umgehen könnte wäre. Das Array könnte durch Zeiger auf „void“ (zu deutsch: „Leere“) definiert werden. Die Zeiger könnten dann über „casts“ (zu deutsch: „Besetzen“) auf andere Objekt-Typen verweisen und so eine Abstraktion bieten, wodurch einzelne Objekt-Typen wieder frei definierbar wären. Diese Variante wurde bewusst ausgeschlossen, weil sie unnötigerweise eine Abstraktionsschicht einführt, welche nicht umgangen werden kann. In der Diplomarbeit hat sich der Ansatz mit gleichen Datentypen innerhalb eines Arrays bewährt, zumal der Datentyp ebenso aus einem Zeiger auf „void“ oder anderen Kollektionen aufgebaut sein kann und somit jener Fall abgebildet werden kann.

Array-Objekt := *(YAML-Map-Schlüssel : Element)**

Ein GA „Array-Objekt“ kann als abgespeckte Variante des GA „Objekt“ gesehen werden. Im Vergleich zum Objekt, ist dieser GA lediglich durch eine YAML-Map auf grammatikalische Ausdrücke „Element“ definiert. Diese müssen ähnlich wie bei dem GA „Objekt“ den notwendigen und optionalen Klassenattributen entsprechen. Fancy Funktionalität, wie die Typisierung kann nicht verwendet werden, aufgrund der beschränkten Datentyp-Länge. Dieser Sachverhalt ist unter dem GA „Array“ genauer beschrieben.

Struktur := *(YAML-Map-Schlüssel : Konstanter-Ausdruck)* | YAML-Array[Konstanter-Ausdruck] | Konstante*

Eine Struktur bezeichnet einen Datentyp, welcher Attribute besitzt auf welche direkt zugegriffen werden kann. In der Programmiersprache Java würde dieser Datentyp als „Plain old Java Object“ (POJO, zu deutsch: Bloßes, altes Java Objekt) bezeichnet werden. In der Programmiersprache C sind dies übliche C-Structs ohne Objektfunktionalität durch die Imps Engine aus dem C-Standard [48]. Der direkte Zugriff auf Attribute ist eine klare Verletzung gegen OOP, die aussagt, dass Daten nur über Funktionen verändert und manipuliert werden dürfen. In der Praxis haben sich Strukturen jedoch bewährt, da sie für sehr einfache Datentypen, wie beispielsweise einen 2-dimensionalen Punkt, welcher lediglich durch je einen X- und Y-Koordinatenwert definiert ist, sehr geeignet sind. Die Abstraktion, welche OOP in diesem Falle schafft, bringt keinen nennenswerten Vorteil. Die Stärke von OOP besteht in der einfachen Austauschbarkeit von internen Klassenattributen ohne die Funktionalität nach außen zu verändern. Bei einem 2-dimensionalen Punkt kommt eine Änderung praktisch nicht vor, jedoch wird hingegen der Zugriff über Funktionen zur unnötigen Last an dieser Stelle. Ebenso ist es Rechen- und Speicherkapazitäts-intensiver eine Klasse zu instanziieren als eine Struktur anzulegen.

Eine Struktur kann dabei auf dreifache Weise erstellt werden. Die erste Möglichkeit ist über eine YAML-Map. Dabei entspricht jeder Schlüsselname einem Attribut der Struktur. An dieser Stelle gibt es keine optionalen Attribute. Jedes Attribut der Struktur muss definiert sein. Der Umstand, dass Strukturen sehr klein gehalten werden sollen und sobald sie größer werden, der direkte Zugriff auf Attribute verboten werden soll begründet die Entscheidung, optionale Attribute zu verbieten. Optionale Attribute stellen eine höhere Komplexität dar, als welche für Strukturen vorgesehen ist. Deswegen ist diese Entscheidung nur konsequent.

Die zweite Möglichkeit ist dabei über einen YAML-Array. Dabei muss der YAML-Array genau der Anzahl der Attribute entsprechen. Jedes Array Element wird an dieser Stelle chronologisch genau einem Attribut zugeordnet. Die ersten beiden Möglichkeiten verwenden dabei den GA „Konstanter-Ausdruck“ für die Erstellung der Strukturattribute. Der Umstand, dass nicht der GA „Objekt“ benutzt wird, ist ebenso

durch die Einfachheit einer Struktur bestimmt. Eine Struktur soll keine komplexen Objekte beinhalten, weil dadurch die Komplexität auf die Struktur übertragen werden würde. Um ein Beispiel zu präsentieren könnten Strukturen, welche Objekte beinhalten nicht einfach kopiert werden, da die Objekte ebenso kopiert werden müssten. Jedoch spielt es für dieses Beispiel keine Rolle, wenn eine Struktur über andere Strukturen definiert ist. Dies ist daher in der Grammatikspezifikation ermöglicht worden.

Die dritte Möglichkeit besteht daraus, die Struktur aus einem GA „Konstante“ zu erstellen. Dabei wird lediglich auf eine Konstante aus dem Quellcode verwiesen, welche genutzt werden soll.

Egal über welche der 3 Möglichkeiten eine Struktur erstellt wird, sie ist stets auf die gleiche Weise nutzbar.

Konstanter-Ausdruck := *Zeichenkette* | *Nummerischer-Ausdruck* | *Boolescher-Ausdruck* | *Konstante*

Ein GA „Konstanter-Ausdruck“ repräsentiert eine Zeichenkette, einen numerischen Ausdruck, einen booleschen Ausdruck oder eine Konstante. Ähnlich wie bei dem GA „Element“ hängt es von dem Kontext ab, welcher GA gewählt wird. Es wird darauf geachtet, welcher GA zur Objekterstellung benötigt wird und dementsprechend ausgewählt. Daher wird beispielsweise für das aktuelle Element eine Integer verlangt, wird entweder ein GA „Nummerischer-Ausdruck“ oder ein GA „Konstante“ gewählt. Zeichenkette und Boolescher-Ausdruck kommen dafür nicht in Frage. Diese beiden genannten grammatikalischen Ausrücke unterscheiden sich in ihrer Syntax eindeutig voneinander, sodass ein deterministisches Verhalten garantiert werden kann. Festzuhalten ist, dass für einen GA „Konstanter-Ausdruck“ lediglich YAML-Skalar-Werte erlaubt sind. Eine YAML-Map oder ein YAML-Array sind ungültig und resultieren in einem Error.

Zeichenkette := .*

Ein GA „Zeichenkette“ besteht aus einem YAML-Skalar-Wert. Dieser Wert wird nicht weiter eingeschränkt und kann daher sehr beliebig sein. Der GA wird für die Objekterstellung verwendet, wenn ein Datentyp „char *“ (char Abk. für „Character“ (engl.), zu deutsch „Zeichen“), daher ein String in C erwartet wird. Strings werden in C über einen Array aus einzelnen Zeichen dargestellt. Da die Länge eines Arrays in C aus Gründen der Performance nicht in das Array mit aufgenommen wird und deswegen aufwendig extra mitgeführt werden müsste, sind Strings in C nullterminiert. Dies bedeutet ein String ist zu Ende, wenn alle 8 Bits eines Zeichens 0 betragen. So kann die Länge eines Strings in C einfach anhand der Zeichenkette ermittelt werden. Dieser Fall wurde berücksichtigt.

Nummerischer-Ausdruck := *\d+(\s*[\+\-*\/]\s*\d+)**

Ein GA „Numerischer-Ausdruck“ besteht ebenso aus einem YAML-Skalar-Wert. Dieser GA wird verwendet, falls Datentypen verlangt werden, die eine Zahl repräsentieren. Für folgende Datentypen trifft dies zu in der Programmiersprache C zu: short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t. Diese Liste hat keinen Anspruch auf Vollständigkeit. Da es sich jedoch um eine prototypische Umsetzung handelt, ist diese Liste hinreichend. Der YAML-Skalar-Wert wird als Ausdruck interpretiert und für die Erzeugung des entsprechenden Datentyps verwendet. Dabei ist er über einen relativ komplexen regulären Ausdruck definiert und soll erklärt werden. Ein Numerischer Ausdruck muss mit einer Zahl beginnen. Diese Zahl enthält beliebig viele Zeichen, mindestens jedoch eins: „\d+“. Anschließend können beliebig viele einfache Rechenoperationen folgen: „(\s*[\+\-*\/]\s*\d+)*“. Im konkreten kann eine einfache Rechenoperation dabei eine Addition „+“, eine Subtraktion „-“, eine Multiplikation „*“ oder eine Division „/“ sein. Anschließend folgt wieder eine Zahl: „\d+“. Zwischen Zahlen und Rechenoperationen können beliebig viele Leerzeichen stehen, oder keine: „\s*“.

Boolescher-Ausdruck := *true* | *false*

Ein GA „Boolescher-Ausdruck“ besteht immer aus einem YAML-Skalar-Wert „true“ oder „false“. Dieser GA wird verwendet, wenn der Datentyp „bool“ erwartet wird. Boolesche Werte sind mit dem C-99-Standard eingeführt worden [48]. Für die Implementierung reicht für einen booleschen Wert ein Bit (0 für falsch, 1 für richtig) aus. Da der kleinste Datentyp in der Programmiersprache C jedoch ein Byte, daher 8 Bit groß ist, sollte der Datentyp „bool“ in C mit Bedacht eingesetzt werden. Dieser Datentyp ist nämlich 8-bit groß. Die restlichen 7 Bits bleiben ungenutzt und werden verschwendet. Aus programmiertechnischer Sicht ist dieser Datentyp jedoch praktisch, da er in der Praxis viele Anwendungsgebiete gefunden hat.

Konstante := *Name*

Ein GA „Konstante“ besteht aus einem GA „Name“. Dabei kann eine Konstante vielseitig eingesetzt werden: Für Objekte, Strukturen und Interne Datentypen der Programmiersprache C. Die Funktionsweise besteht dabei immer daraus, dass die Konstante im Quellcode hinterlegt wird und über den Namen angesprochen werden kann. Dieses Wissen über die Konstanten wird aus dem Quellcode entnommen. Die Funktionsweise einer Konstanten wurde dabei nicht verändert: Eine Konstante ist eine Variable, deren Variablenwert sich nach Initialisierung nicht ändert. In der Programmiersprache C gibt es auch Möglichkeiten Variablen zu ändern, welche als konstant definiert wurden zu verändern, dies wird aber als sehr schlechtes Programmierverhalten angesehen und deswegen wird diese Möglichkeit für die Diplomarbeit nicht weiter in Betracht gezogen.

Es ist festzuhalten, dass Zeichenketten nicht über Konstanten ansprechbar sind. Der Grund liegt darin, dass wenn eine Zeichenkette erwartet wird, nicht unterschieden werden kann, ob eine Zeichenkette als Zeichenkette oder als Konstante zu interpretieren ist. Beispielsweise könnte „Hintergrundfarbe“ ebenso für eine Konstante „Hintergrundfarbe“ aus dem Quellcode oder den Text „Hintergrundfarbe“ stehen. Um diesen Konflikten vorzubeugen ist die Referenzierung von konstanten Zeichenketten nicht möglich. In allen anderen Fällen ist es eindeutig, dass wenn eine Referenz auf eine Konstante vorliegt, daher ein YAML-Skalar-Wert, welcher einem GA „Name“ entspricht, dieser eine Referenz auf eine Konstante ist. Wenn auf eine Konstante verwiesen wird, welche im Quellcode nicht hinterlegt wurde, daher nicht gefunden werden kann, resultiert dies in einem Fehler.

Zusätzlich verfügt der GA „Konstante“ noch über eine Besonderheit. Konstanten, welche öffentlich zugreifbar sind, sind über einen Namensraum, in der Regel dem Dateinamen, gekennzeichnet. Beispielsweise „IAColor_red“, wobei „IAColor“ für den Namensraum steht und „red“ (zu deutsch: rot) für den Variablennamen, getrennt mittels Unterstrich „_“. Wenn die Konstante in der Datei hinterlegt ist, in welcher ebenso der Dateityp definiert wurde, daher für das Beispiel „IAColor_red“ wurde in „IAColor.h“ definiert, kann bei der Referenzierung der Namensraum weggelassen werden. Es ist daher möglich, eine rote Farbe des Datentyps „IAColor“ direkt über „red“ zu referenzieren. Wenn weitere Konstanten des Datentyps „IAColor“ in anderen Quellcodedateien festgelegt werden, sind diese stets über den Namensraum zu referenzieren, daher den vollständigen Variablennamen.

6.2.6 Referenzierung von Objekten in der Auszeichnungssprache von außerhalb

Die Referenzierung von Objekten soll von außerhalb möglich sein. Dabei ist zunächst festzuhalten, dass der Zweck der Auszeichnungssprache darin besteht eine Objekterstellung durchzuführen. Dabei wird pro YAML-Datei ein Wurzel-Objekt erzeugt. Dieses Wurzel-Objekt kann, wie in Kapiteln zuvor beschrieben über weitere Objekte und Strukturen verfügen, welche nach einer Grammatikspezifikation rekursiv ineinander geschachtelt werden können.

Es soll nun also ermöglicht werden, auf sämtliche Objekte, sowohl das Wurzel-Objekt, als auch verschachtelte Objekte zuzugreifen. In der Programmiersprache C lassen sich dazu verschiedene Mechanismen finden, welche für diesen Anwendungsfall geeignet sind. Die Objekterstellung soll in einer Funktion erfolgen. Funktionen in C haben genau einen Rückgabewert. Dieser Rückgabewert wird häufig in C-Projekten für Fehlercodes verwendet. Fehler in der Auszeichnungssprache werden jedoch bereits zur Kompilierzeit aufgedeckt. Zur Laufzeit würde höchstens ein Fehler bzgl. nicht ausreichendem Arbeitsspeichers auftreten können, doch dieser wird bereits von der Imps Engine über eine Methode „onMemoryWarning“ (zu deutsch: „eintreten einer

Speicherwarnung“) gehandhabt. Deswegen kann der Rückgabewert problemlos und komfortabel für das Wurzel-Objekt genutzt werden.

Schwieriger wird es für verschachtelte Objekte. Um hier eine Referenzierung zu ermöglichen wurde folgendes System entwickelt. Objekte, welche referenzierbar sein sollen, müssen einen YAML-Map-Schlüssel „__ref__“ besitzen, siehe dazu Kapitel 6.2.5 Spezifikation der Auszeichnungssprache. Der zugehörige YAML-Map-Wert gibt dabei den Referenznamen an, über welchen das Objekt referenzierbar sein soll. Anschließend kann dem Objekterstellungsalgorithmus für jede Referenz, optional ein Zeiger übergeben werden, an welcher Stelle das Objekt referenziert werden soll. Daher kann anhand des Zeigers das Objekt an der gewünschten Stelle abgelegt werden. Da diese Stelle von außerhalb bekannt ist, kann nachdem der Objekterstellungs-Algorithmus beendet ist, auf das Objekt zugegriffen werden.

Es ist dabei anzumerken, dass dieser Mechanismus lediglich für ein Objekt pro Referenznamen funktioniert, weil es ansonsten den jeweiligen Objektzeiger überschreiben würde. Hinzu kommt, dass YAML-Maps unsortiert sind, weswegen die Reihenfolge, in welcher Objekte des gleichen Referenznamens erstellt werden zufällig sein kann. Um diesen Problemen aus dem Weg zu gehen, wurde an dieser Stelle die Einschränkung getroffen, dass Referenznamen eindeutig innerhalb einer YAML-Datei sein müssen. Es ist Gegenstand weiterer Forschung geeignete Verfahren zu finden, um Möglichkeiten einer mehrfachen Referenzierung von Objekten zu untersuchen.

6.2.7 Erzeugung von Klassen durch Subklassen

6.2.8 Objekterstellungsreihenfolge

6.2.9 Objekterstellungs-Injektion

6.2.10 Resource-Provider Konzept

Abgrenzung zu konstanten

6.2.11 Schritte zum kompilieren des Prototypen

Um den Prototypen auszuführen, muss sämtlicher Code kompiliert werden. Dies erfolgt vereinfacht dargestellt in 3 Schritten.

6.2.11.1 Generierung der Funktionalität für Objektorientierung

6.2.11.2 Generierung der Objekt-Erstellungsalgorithmen von YAML

6.2.11.3 Kompilierung des C-Codes inklusive generiertem Code

6.3 Lösungskonzepte bzgl. des Prototyps

6.3.1 Grober Softwareentwurf mittels MVP

6.3.2 Feinentwurf Model

6.3.3 Feinentwurf View

6.3.4 Feinentwurf Presenter

6.4 Implementation

6.4.1 Ausgewählte Basisklassen

6.4.2 Eleganter Aufbau der Hierarchie in YAML

Attribute geschickt sortieren, sodass schöner Aufbau entsteht.

6.4.3 Implementation der jeweiligen Konzepte mit ausgewählten Codebeispielen (Kapitel aufsplitten je Konzept)

→ spezifisch schwerpunkte setzen, bei welchen Konzepten Probleme in der Implementation aufgetreten sind. Wenn alles glatt gelaufen ist eher nicht darauf eingehen.
- Allokation Count Problematik
- Objekt, Struktur zeigen, Konstante

6.5 Validierung

[Überprüfen, ob Rahmenbedingungen erfüllt worden sind]

7 Open-Source-Aktivitäten und Abschließende Bemerkungen des Projektes

8 Quellenverzeichnis

[1] „GUI Architectures“, *martinfowler.com*. [Online]. Verfügbar unter: https://martinfowler.com/eaaDev/uiArchs.html. [Zugegriffen: 14-Sep-2018].

[2] dotnet-bot, „Data Binding Overview“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview. [Zugegriffen: 15-Okt-2018].

[3] „The Early History of Smalltalk“. [Online]. Verfügbar unter: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html. [Zugegriffen: 11-Okt-2018].

[4] „Passive View“, *martinfowler.com*. [Online]. Verfügbar unter: https://martinfowler.com/eaaDev/PassiveScreen.html. [Zugegriffen: 21-Nov-2018].

[5] „MVC vs. MVP vs. MVVM on Android“. [Online]. Verfügbar unter: http://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/. [Zugegriffen: 21-Nov-2018].

[6] „Understanding the basics of MVVM design pattern – Microsoft Gulf Technical Community“. [Online]. Verfügbar unter: https://blogs.msdn.microsoft.com/msgulfcommunity/2013/03/13/understanding-the-basics-of-mvvm-design-pattern/. [Zugegriffen: 21-Nov-2018].

[7] dotnet-bot, „Getting Started (WPF)“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/dotnet/framework/wpf/getting-started/. [Zugegriffen: 21-Nov-2018].

[8] „Photo Editing Software - Photo Editor for Online, Mac & PC | Adobe“. [Online]. Verfügbar unter: https://www.adobe.com/products/photoshopfamily.html. [Zugegriffen: 09-Nov-2018].

[9] „Proto.io - Prototypes that feel real“. [Online]. Verfügbar unter: https://proto.io/. [Zugegriffen: 09-Nov-2018].

[10] „Top 10 IT Entwicklung Trends in 2018 | ASFAST-EDV“. .

[11] „Agile Methodology“. .

[12] „Manifesto for Agile Software Development“. [Online]. Verfügbar unter: https://agilemanifesto.org/. [Zugegriffen: 22-Nov-2018].

[13] „Apple“, *Apple*. [Online]. Verfügbar unter: https://www.apple.com/. [Zugegriffen: 22-Nov-2018].

[14] „What is embedded systems programming? - Definition from WhatIs.com“, *IoT Agenda*. [Online]. Verfügbar unter: https://internetofthingsagenda.techtarget.com/definition/embedded-systems-programming. [Zugegriffen: 22-Nov-2018].

[15] M. Barr, *Programming Embedded Systems in C and C++*. O’Reilly Media, Inc., 1999.

[16] „Heap-Fragmentierung – Mikrocontroller.net“. [Online]. Verfügbar unter: https://www.mikrocontroller.net/articles/Heap-Fragmentierung. [Zugegriffen: 22-Nov-2018].

[17] „Markup Language Definition“. [Online]. Verfügbar unter: https://techterms.com/definition/markup_language. [Zugegriffen: 04-Okt-2018].

[18] „Auszeichnungssprache :: ML (markup language) :: ITWissen.info“. [Online]. Verfügbar unter: https://www.itwissen.info/Auszeichnungssprache-markup-language-ML.html. [Zugegriffen: 04-Okt-2018].
[19] „About W3C“. [Online]. Verfügbar unter: https://www.w3.org/Consortium/. [Zugegriffen: 24-Nov-2018].
[20] „Extensible Markup Language (XML) 1.0 (Fifth Edition)“. [Online]. Verfügbar unter: https://www.w3.org/TR/REC-xml/. [Zugegriffen: 24-Nov-2018].
[21] 14:00-17:00, „ISO 8879:1986“, *ISO*. [Online]. Verfügbar unter: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/01/63/16387.html. [Zugegriffen: 24-Nov-2018].
[22] „XML Tutorial“. [Online]. Verfügbar unter: https://www.w3schools.com/xml/. [Zugegriffen: 24-Nov-2018].
[23] K. Schild und M. Mochol, „Aufbau von XML- Dokumenten“, S. 69, 2006.
[24] dotnet-bot, „XAML overview (WPF)“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/xaml-overview-wpf. [Zugegriffen: 24-Nov-2018].
[25]. „NET | Free. Cross-platform. Open Source.“, *Microsoft*. [Online]. Verfügbar unter: https://www.microsoft.com/net. [Zugegriffen: 24-Nov-2018].
[26] dotnet-bot, „XAML in WPF“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/xaml-in-wpf. [Zugegriffen: 24-Nov-2018].
[27] dotnet-bot, „Markup Extensions and WPF XAML“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/markup-extensions-and-wpf-xaml. [Zugegriffen: 24-Nov-2018].
[28] dotnet-bot, „Code-Behind and XAML in WPF“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/code-behind-and-xaml-in-wpf. [Zugegriffen: 24-Nov-2018].
[29] dotnet-bot, „Inline Styles and Templates“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/inline-styles-and-templates. [Zugegriffen: 24-Nov-2018].
[30] „HTML 5.2“. [Online]. Verfügbar unter: https://www.w3.org/TR/html52/index.html#contents. [Zugegriffen: 24-Nov-2018].
[31] „HTML and XHTML“. [Online]. Verfügbar unter: https://www.w3schools.com/html/html_xhtml.asp. [Zugegriffen: 24-Nov-2018].
[32] „Usage Statistics and Market Share of HTML for Websites, November 2018“. [Online]. Verfügbar unter: https://w3techs.com/technologies/details/ml-html/all/all. [Zugegriffen: 24-Nov-2018].
[33] „YAML Ain’t Markup Language (YAML™) Version 1.2“. [Online]. Verfügbar unter: http://yaml.org/spec/1.2/spec.html. [Zugegriffen: 12-Nov-2018].
[34] „Kubernetes Documentation“. [Online]. Verfügbar unter: https://kubernetes.io/docs/home/. [Zugegriffen: 24-Nov-2018].
[35] „Run a Stateless Application Using a Deployment“. [Online]. Verfügbar unter: https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/. [Zugegriffen: 24-Nov-2018].
[36] „XML, JSON und YAML im Vergleich“, *predic8.com*. [Online]. Verfügbar unter: http://www.predic8.de/xml-json-yaml.htm. [Zugegriffen: 12-Nov-2018].

[37] „Trends in der modernen Software-Entwicklung“. [Online]. Verfügbar unter: https://www.microsoft.com/germany/techwiese/know-how/trends-in-der-modernen-software-entwicklung.aspx. [Zugegriffen: 14-Sep-2018].
[38] „CMake“. .
[39] *Contribute to impsapps/impsengine development by creating an account on GitHub*. impsapps, 2018.
[40]: *herb: Expat library: Fast streaming XML parser written in C; in the process of migrating from SourceForge to GitHub: libexpat/libexpat*. Expat development team, 2018.
[41] *SIMD-oriented Fast Mersenne Twister. Contribute to MersenneTwister-Lab/SFMT development by creating an account on GitHub*. MersenneTwister-Lab, 2018.
[42] „SIMD-oriented Fast Mersenne Twister (SFMT)“. [Online]. Verfügbar unter: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/. [Zugegriffen: 12-Nov-2018].
[43] „OpenCV library“. [Online]. Verfügbar unter: https://opencv.org/. [Zugegriffen: 12-Nov-2018].
[44] „OpenGL - The Industry Standard for High Performance Graphics“. [Online]. Verfügbar unter: https://www.opengl.org/. [Zugegriffen: 12-Nov-2018].
[45] „GLEW: The OpenGL Extension Wrangler Library“. [Online]. Verfügbar unter: http://glew.sourceforge.net/. [Zugegriffen: 12-Nov-2018].
[46] „CLion: A Cross-Platform IDE for C and C++ by JetBrains“, *JetBrains*. [Online]. Verfügbar unter: https://www.jetbrains.com/clion/. [Zugegriffen: 12-Nov-2018].
[47] „MinGW | Minimalist GNU for Windows“. [Online]. Verfügbar unter: http://www.mingw.org/. [Zugegriffen: 12-Nov-2018].
[48] „C99 Standard PDF Entwurf“. [Online]. Verfügbar unter: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n897.pdf. [Zugegriffen: 12-Nov-2018].
[49] „Was ist OOP?“ [Online]. Verfügbar unter: https://www.dev-insider.de/was-ist-oop-a-677737/. [Zugegriffen: 13-Nov-2018].
[50] „pthreads(7) - Linux manual page“. [Online]. Verfügbar unter: http://man7.org/linux/man-pages/man7/pthreads.7.html. [Zugegriffen: 13-Nov-2018].
[51] „<pthread.h>“. [Online]. Verfügbar unter: http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html. [Zugegriffen: 13-Nov-2018].
[52] „ImpsEngine: Annotations documentation“. [Online]. Verfügbar unter: https://impsapps.github.io/impsengine-doc/d9/d3b/md_docs__annotations-_documentation.html. [Zugegriffen: 13-Nov-2018].

II Abbildungsverzeichnis

III Tabellenverzeichnis

IV Abkürzungsverzeichnis

GUI
YAML
MIT-Lizenz
SFMT
BSD
OOP (objekt orientierte programmierung)