Moderne Ansätze zur Oberflächengestaltung für hardwarenahe Programmierung

zur Erlangung des akademischen Grades

Diplom-Wirtschaftsinformatiker (FH)

der Fakultät Mathematik/ Informatik

der Hochschule für Technik und Wirtschaft Dresden

Eingereicht von	Konstantin Merker
Matrikelnummer	38937
eingereicht am	18.12.2018
Betreuerin	Prof. Dr.-Ing. Anna Sabine Hauptmann
2. Gutachter	Prof. Dr.-Ing. Georg Freitag

Erklärung über die eigenständige Erstellung der Arbeit

Hiermit erkläre ich, dass ich die vorgelegte Arbeit mit dem Titel

Moderne Ansätze zur Oberflächengestaltung für hardwarenahe Programmierung

selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit als solche und durch Angabe der Quelle gekennzeichnet habe. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Mir ist bewusst, dass die Hochschule für Technik und Wirtschaft Dresden Prüfungsarbeiten stichprobenartig mittels der Verwendung von Software zur Erkennung von Plagiaten überprüft.

Ort, Datum	Unterschrift Studierender

Inhaltsverzeichnis

1 Motivation

Wir leben in einer Welt des permanenten technologischen Fortschritts. Die Art und Weise, wie wir dabei mit Technik interagieren hat sich in den letzten Jahrzehnten stark verändert. Neben aktuellen Trends wie virtuellen Brillen, erweiterter Realität (zu engl. „Augmented Reality“), Cloud-Computing und neuronalen Netzwerken nehmen graphische Displays eine zunehmende Präsens in unserem Alltag ein. Wir tragen sie mittlerweile nicht nur als Smartphone mit uns, sondern fangen an, Displays in unsere Autos, an unsere Kühlschränke, an Automaten für den öffentlichen Verkehr, an Smartwatches und vielem mehr zu bauen.

Dabei entstehen für neue Anwendungsfälle neue Anforderungen. Häufig sind die Rechnerkapazitäten, welche die graphischen Benutzeroberflächen betreiben sollen, eingeschränkt. Rechner, wie sie im Büroalltag eingesetzt werden, verfügen über immense Ressourcen, welche beispielsweise für einen Drucker eine reine Verschwendung darstellen würden. Der Aspekt der effizienten und damit hardwarenahen Programmierung verdient deswegen zunehmend Betrachtung, jedoch tauchen dadurch neue Probleme auf. Technologien, welche sich derzeit für die Oberflächengestaltung hoher Popularität erfreuen, sind nicht unbedingt Hardware-effizient. Dabei schaffen sie jedoch einen hohen Mehrwert an Entwicklerkomfort und vereinfachen die Entwicklung immens. Bestehende Lösungen können deswegen nicht verworfen werden, sondern müssen stattdessen überprüft werden, ob sie aktuellen technologischen Trends gerecht werden.

Inspiriert von diesen Problemstellungen und den bisherigen Technologien soll in dieser Diplomarbeit eine neue moderne Lösung unter dem Namen Flower entstehen, welche eine hardwarenahe Programmierung für graphische Benutzeroberflächen ermöglichen soll. Dies wird durch eine prototypische Entwicklung unter Beweis gestellt.

Die Leitthese der Diplomarbeit lautet: Es ist möglich, moderne Konzepte für graphische Benutzeroberflächenentwicklung für hardwarenahe Programmierung zugänglich zu machen und somit sowohl den Komfort in dem Bereich der Softwareentwicklung, als auch die Nähe zur Hardware beizubehalten, um eine effiziente Entwicklung zu ermöglichen.

Leitmotive aus der Softwareentwicklung, welche diese Arbeit vorangetrieben haben, sind Agilität, insbesondere Flexibilität, um einem sich stark ändernden Umfeld gerecht zu werden.

In der Diplomarbeit werden aus Gründen der Übersichtlichkeit Namen von Dateien, Klassen, Funktionen sowie Variablen kursiv geschrieben.

2 Begriffsklärungen und Begriffsunterscheidungen

Begriffsklärung „modern“

Der Begriff „modern“ wird im Rahmen der Diplomarbeit abhängig vom Kontext auf aktuelle Trends in Technologien und der Softwareentwicklung bezogen. Dies schließt eine möglichst agile Entwicklung, kurze Software-Iterationszyklen und das möglichst lange Hinauszögern des Treffens von Entscheidungen ein. Ebenso sind aktuelle Software-Paradigmen, -Konzepte und -Muster im Begriff inkludiert.

Begriffsklärung „hardwarenah“

Der Begriff „hardwarenah“ bezeichnet im Kontext der Programmierung eine Rechenkapazität-schonende Softwareentwicklung. Dies schließt eine Programmierung für limitierte Hardwareressourcen ein. Der Verzicht auf Entwicklerkomfort und Abstraktionen wird zugunsten eines effizienteren Quellcodes eingegangen.

Begriffsklärung „graphische Benutzeroberfläche“

Der Begriff „graphische Benutzeroberfläche“ bezieht sich ausschließlich auf Benutzeroberflächen, welche eine visuelle Ästhetik besitzen. Darunter fallen Displays, welche Inhalte pixelgenau wiedergeben können sowie über eine Farbtiefe von mindestens 24-Bit verfügen. Häufig benutzte Synonyme dafür sind Graphical User Interface (GUI, zu dt. „Graphische Benutzerschnittstelle“) oder kurz „Benutzeroberfläche“.

Begriffsklärung „Technologie“

Eine Technologie kann beliebig viele Software-Frameworks, Software-Bibliotheken sowie eigene Tools enthalten um eine Entwicklung zu vereinfachen. Teilweise kann eine Technologie auch eine eigene Entwicklungsumgebung enthalten um eine Komfortabilität für die Entwicklung bereitzustellen.

Begriffsunterscheidung „Softwarekonzept“ und „Softwarearchitektur“

Eine Softwarearchitektur gibt ein komplette Beschreibung zu einem Softwaresystem an. Ein Softwarekonzept stellt die Struktur von einem Teilaspekt des Softwaresystems dar.

Begriffsunterscheidung „Prototyp“ und „Software“

Ein Prototyp ist im Vergleich zu einer Software ein Programm, welches entwickelt wird um eine bestimmte Vorgehensweise oder Technologie unter bestimmten Kriterien zu testen. Der Unterschied zu einem Softwareprogramm besteht darin, dass bei einer prototypischen Entwicklung Abstriche in Bezug auf Robustheit und Wartbarkeit vorgenommen werden, weil die Erkenntnis im Vordergrund steht und nicht ein nutzbares Programm.

Begriffsunterscheidung „Software-Framework“ und „Software-Bibliothek“
Ein Software-Framework sowie eine Software-Bibliothek bieten beide Funktionalität an, welche in ein bestehendes Programm eingebunden werden kann. Ein Software-Framework enthält im Gegensatz zur Software-Bibliothek eine eigene Vorgehensmethodik. Diese besteht häufig darin, dass das Framework über eine interne Ablaufslogik verfügt und an entsprechenden Stellen Ereignisse wirft. Über die Ereignisse kann applikationsspezifische Funktionalität aufgerufen werden. Ein Programm, welches das Software-Framework nutzt, muss die vorgegebene Vorgehensmethodik anwenden. Eine Software-Bibliothek ist dabei unabhängig von einer bestimmten Vorgehensweise nutzbar.

Begriffsunterscheidung „Methode“ und „Funktion“
Als Methode wird in der Diplomarbeit eine Funktion ohne Rückgabewert bezeichnet. Der Begriff Funktion stellt damit den übergeordneten Begriff der Methode dar.

3 Vorbetrachtungen zu Möglichkeiten der Programmierung von Benutzeroberflächen

In diesem Kapitel sollen bisherige Erkenntnisse aus Wissenschaft und Forschung zusammengetragen werden, welche bezogen auf die Diplomarbeit Relevanz besitzen.

3.1 Bisherige Konzepte 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 übliches 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. Sobald die Übertragung abgeschlossen ist, wurde der Datensatz an allen Stellen geändert [2].

Dies soll anhand folgendes Beispiels erläutert werden: Eine Softwareanwendung besteht aus einem Formular mit jeweils einem Kontrollelement Textfeld, Button und einem Ausgabefeld. Durch das Textfeld soll eine Namenseingabe ermöglicht werden. 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 eingefügt wird, 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, beispielsweise eine Textdatei oder eine relationale Datenbank. Die Speicherung auf

einem persistenten Speichermedium 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. Weitere Auswirkungen sind zu vernachlässigen, weil die Änderungen in jedem Fall über das Muster Datenbindung durchgegeben werden.

Was in diesem simplen Anwendungsfall einfach erscheint, stellt in der Praxis eine besondere Herausforderung dar, weil Zyklen vermieden werden müssen. Daher, falls die Daten intern geändert werden und diese Änderung an die GUI übergeben wird, darf diese wiederum nicht die Änderung an die interne Variable weitergeben. 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 Datensätze 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 diese Anforderung oft benötigt [1].

3.1.2 Model-View-Controller (MVC)

Der Model-View-Controller (zu dt. „Modell-Benutzeroberfläche-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 gehen in die 80er Jahre zurück [1]. In 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 auf Unix basierenden Betriebssystemen 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 [4].

Im Konzept werden Domain-spezifische Objekte innerhalb des Musters als Model bezeichnet. Die graphischen Elemente bilden den sogenannten 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. Das Zusammenspiel zwischen Model, View und Controller als Paketdiagramm wird in der Abbildung 1 verdeutlicht.

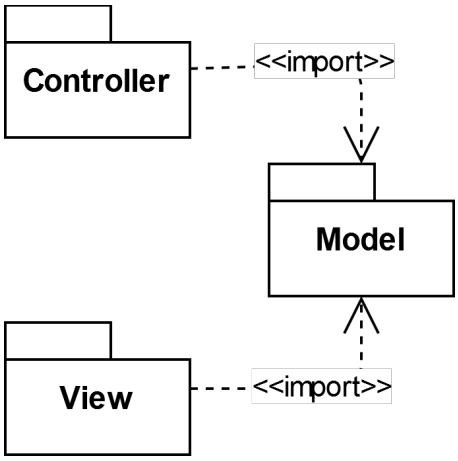

Abbildung 1: Zusammenspiel zwischen Model, View und Controller

Das Model kann dabei als Abbildung der realen Welt gesehen werden. Anders als in dem Konzept Formulare und Kontrollelemente geht dieses Model nicht von Datensätzen aus, die manipuliert werden können, sondern von Objekten, mit welchen der Benutzer in der Anwendung agiert [1].

Um dieses Konzept zu demonstrieren soll als Beispiel der gleiche Anwendungsfall dienen, welcher zur Erklärung der Formulare und Kontrollelemente ausgewählt wurde, siehe Kapitel 3.1.1. Wenn die Begrüßungsanwendung über das MVC-Konzept umgesetzt werden würde, so könnte das Model aus einem Benutzer (zu engl. „User“) bestehen, welches einen Namen beinhaltet und einen Grußtext. Für die Veranschaulichung soll ein Klassendiagramm dienen, siehe Abbildung 2: Begrüßungsanwendung im MVC-Konzept, Klassendiagramm.

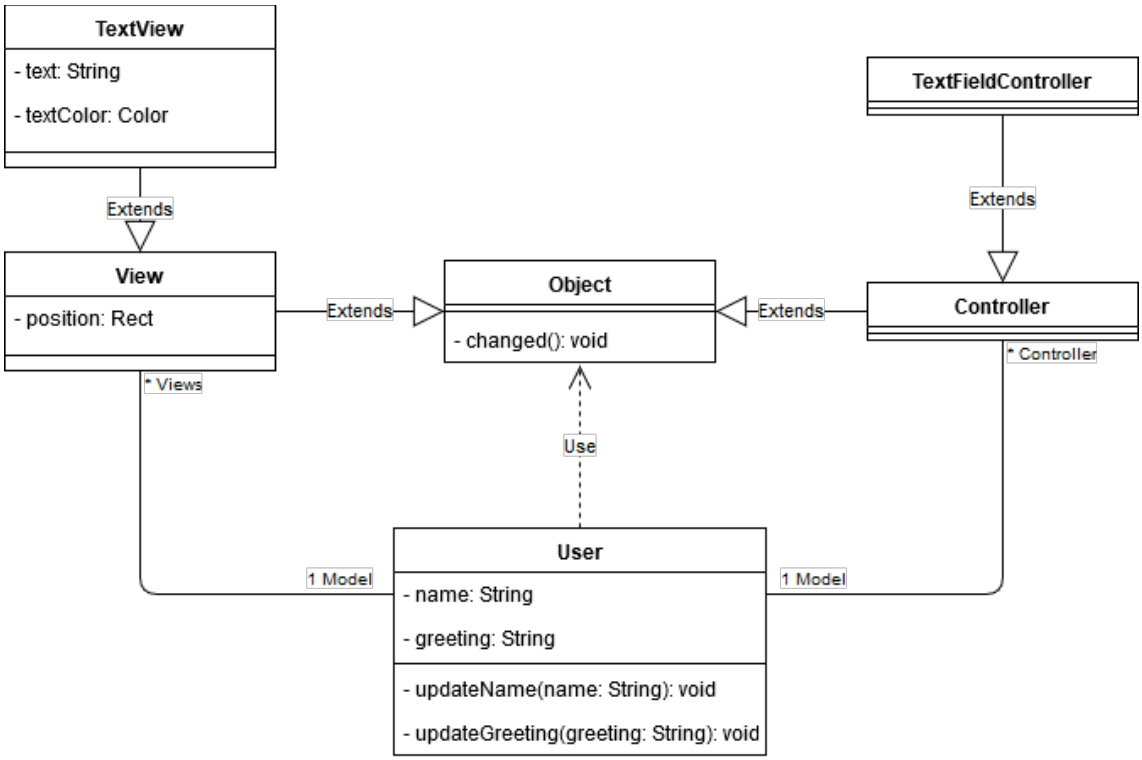

Abbildung 2: Begrüßungsanwendung im MVC-Konzept, Klassendiagramm

Innerhalb dieses Klassendiagramms repräsentieren die Klassen *TextView* und *View* den View, die Klassen *TextFieldController* und *Controller* den Controller und die Klasse *User* das Model. Die Klasse *Object* (zu dt. „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 des Views bzw. Controllers abzubilden.

Erwähnenswert ist, dass das Model weder View noch Controller kennt. Die Beziehung zwischen View und Model bzw. Controller und Model, 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. Zur Veranschaulichung wird der Änderungsfall betrachtet, dass eine Nutzereingabe erfolgt und ein Text eingegeben wird. Die Änderungen werden über ein Sequenzdiagramm dargestellt, siehe Abbildung 3: Begrüßungsanwendung im MVC-Konzept, Sequenzdiagramm für Nutzereingabe.

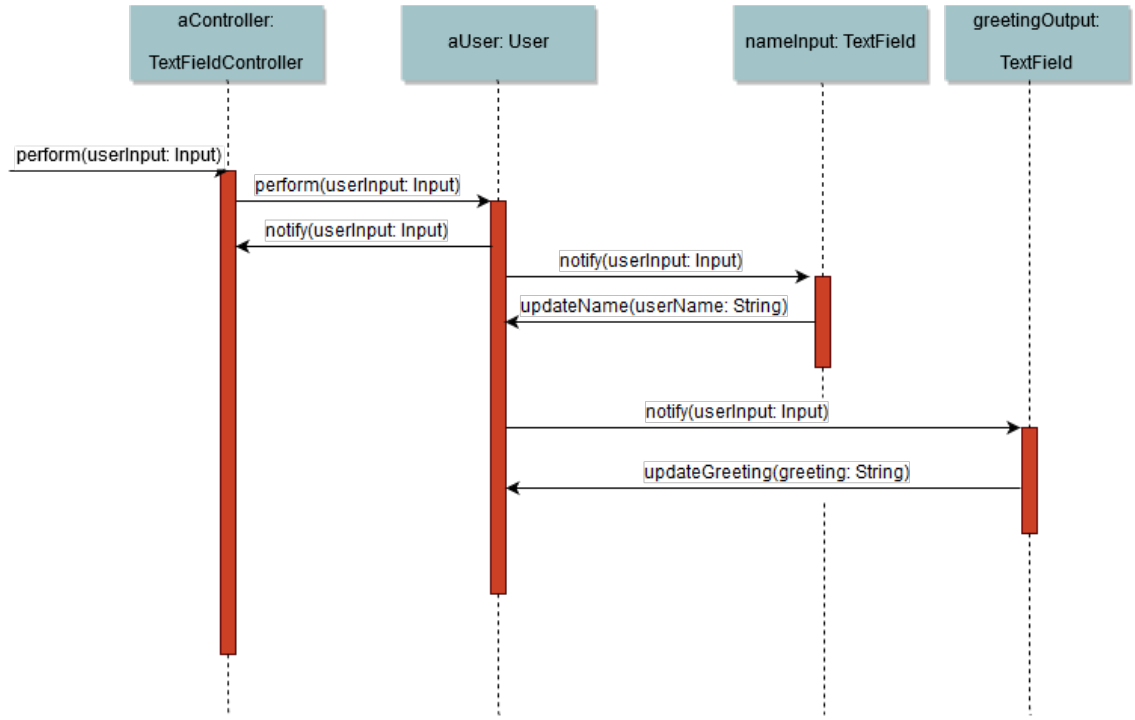

Abbildung 3: Begrüßungsanwendung im MVC-Konzept, Sequenzdiagramm für Nutzereingabe

Aus dem Sequenzdiagramm lässt sich entnehmen, dass die Objekte *aController*, *aUser*, *nameInput* und *greetingOutput* jeweils eine Instanz der Klassen *TextFieldController*, *User*, *TextField* bzw. *TextField* 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 verarbeiten. Dabei updaten die Objekte *nameInput* und *greetingOutput* die jeweiligen Variablenwerte für die Anzeigetexte Objekt-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 im Vergleich zum Konzept Formulare und Kontrollelemente etwas größer ist, besteht der klare Vorteil durch die Implementierung des Beobachter-Musters in der Testbarkeit des Models. Das Model 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-Presenters (zu dt. „Modell-Benutzeroberfläche-Präsentation“) weist eine starke Ähnlichkeit zum MVC-Konzept auf, siehe Kapitel 3.1.2 Model-View-Controller (MVC). Ebenso wie für das MVC-Konzept gibt es für das MVP-Konzept eine Vielzahl an Variationen. Diese Diplomarbeit verwendet ausschließlich die Variante des Passive View (zu dt. „Passive Anzeige“) [5].

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

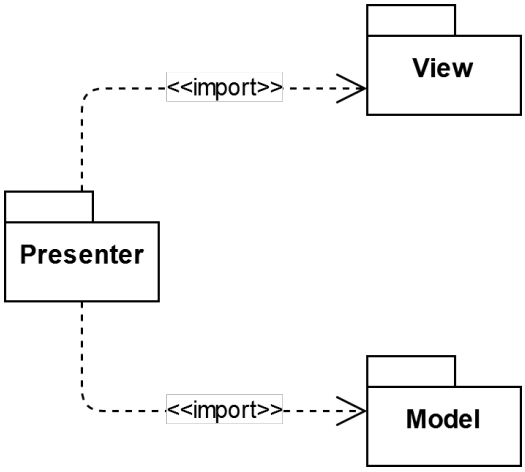

Abbildung 4: Zusammenspiel zwischen Model, View und Presenter

Wie in oben stehender Abbildung zu erkennen ist, liegt ein markanter Unterschied zu dem MVC-Konzept darin, dass keine direkte Verbindung zwischen Model und View besteht sowie sämtliche Kommunikation stattdessen über den Presenter durchgeführt wird. Die Views verfügen dabei über eine Schnittstelle, worüber der Presenter kommunizieren kann [6].

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

Die beiden Veränderungen fügen Komplexität für die Programmierung von graphischen Benutzeroberflächen hinzu, bieten jedoch folgende zwei Vorteile:

- Der Presenter kann im Vergleich zu dem Controller einfacher durch Tests abgedeckt werden, weil die Views über Schnittstellen verfügen, welche für Tests genutzt werden können. In dem strikteren MVC-Konzept greift der Controller direkt auf die Objekte der Views zu, wodurch eine Testabdeckung erschwert wird.

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

Obwohl Presenter und Model in dem MVP-Konzept gut testbar sind, bleibt der View schwer testbar ist. Hierbei ist anzumerken, dass das Testen von GUI-Funktionalität ein generelles Problem darstellt. Deswegen wurde die Konzeptvariante Passive View entwickelt, welche vorgibt möglichst viel Funktionalität in Model und Presenter auszulagern und den View ohne Anwendungslogik zu versehen. Es ist am günstigsten die Logik stark im Model vertreten zu haben, da dort der größte Anteil wiederverwendet werden kann. Im Presenter kann der Quellcode nur bedingt wiederverwendet werden, ist jedoch noch testbar. Der View sollte im Idealfall lediglich übermittelte Informationen aus dem Presenter anzeigen. [5]

3.1.4 Model-View-ViewModel (MVVM)

Das Konzept des Model-View-ViewModel (zu dt. „Modell-Benutzeroberfläche-Benutzeroberflächenmodell“) stellt neben dem MVP-Konzept einen weiteren modernen Ansatz für graphische Benutzeroberflächenentwicklung dar. Dieses Konzept weist Ähnlichkeiten zu dem MVC-Konzept auf und beinhaltet das Muster der Datenbindung, welches im Kapitel 3.1.1 Formulare und Kontrollelemente erörtert ist.

Das Zusammenspiel der einzelnen Bestandteile des MVVM-Konzepts ist dargestellt in der Abbildung 5: Zusammenspiel zwischen Model, View und ViewModel.

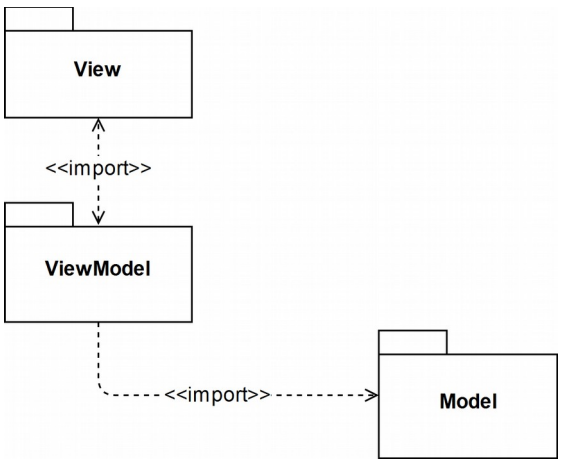

Abbildung 5: Zusammenspiel zwischen Model, View und ViewModel

Die Ähnlichkeit zu dem MVC-Konzept besteht darin, dass der Quellcode in einen View, ein Model und in diesem Fall ein ViewModel zerlegt wird. Das ViewModel übernimmt dabei die Aufgabe des Controllers. Es kann als Mittelsmann zwischen View und Model gesehen werden [7].

Hervorzuheben für dieses Konzept ist die gegenseitige Abhängigkeit zwischen View und ViewModel. Diese Tatsache basiert auf der Datenbindung, welche zwischen beiden Teilen vorhanden ist. Dies bedeutet, dass Änderungen an Variablenwerten innerhalb des Views automatisch und wechselseitig an das ViewModel propagiert werden. 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 dem Konzept 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 [8]. In diesem Fall ist das MVVM-Konzept fest in dem Framework integriert. Der View wird dabei ausschließlich mit der Auszeichnungssprache XAML entwickelt und das zugehörige ViewModel ist 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 hohe Kosten verursacht und das graphische Design nicht direkt von der Programmierung abhängig ist, ist es eine logische Vorgehensweise das graphische Design zuerst zu erstellen und anschließend mit der gewünschten Programmfunktionalität zu versehen. Das graphische Design kann mittels geeigneter Anwendungen, z. Bsp. Photoshop [9] oder proto.io [10] erstellt sowie im Roh-Zustand getestet werden. Wenn das Design den Anforderungen entspricht, erfolgt anschließend die Implementierung. Dazu müssen die graphischen Elemente in geeigneter Form, beispielsweise in dem Grafikformat „Portable Network Graphics (PNG, zu dt. „protable Netzwerkgrafik“) [11], 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 dieser Vorgehensweise besteht darin, dass das graphische Design und die Implementierung sauber getrennt voneinander erfolgen können. Die Schnittstelle zwischen Designer und Programmierer ist deutlich durch die extrahierten Dateien gegeben. Das graphische Design für den Prototyp wurde von einem Externen erstellt [12]. Dadurch wird die visuelle Qualität des Endproduktes gewährleistet.

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 [13]. Dies verlangt der Programmierung neue Vorgehensweisen ab. Eine agile Bewegung im Laufe der letzten Jahre brachte eine Vielzahl von neuen Tools und Vorgehensweisen hervor [14]. Einzelne Vertreter dieser Bewegung unterzeichneten ein Agiles Manifest, in welchem die Ziele der Bewegung definiert wurden [15]. Darin lässt sich unter anderem der nahe Kontakt zum Kunden und eine hohe Bereitschaft finden auf Änderungen einzugehen. Starre Pläne sind dabei weniger zielführend.

Schlussfolgernd soll mit dem Prototyp keine bewegungslose Software entstehen, die in einem Monolithen endet. Er muss stattdessen die nötige Flexibilität besitzen, um auf Änderungen schnell und flexibel eingehen zu können.

Des Weiteren stellt dies neue Anforderungen an die Testabdeckung. Bei schnellen und kurzen iterativen Veränderungen 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 Änderungen und eine gewisse Robustheit zu garantieren.

Ein weiterer Trend in der GUI-Programmierung 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 Innovationen essentiell vorantreibt, beispielsweise das Smartphone. Wie auf der Startseite zu erkennen ist, wird kein Textblock angezeigt, sondern stattdessen Schlagworte und Bilder zu Produkten [16]. Die psychologische Wirkung von graphischen Benutzeroberflächen auf den Konsumenten im Kontext des Marketing soll an dieser Stelle vernachlässigt werden, weil es den Rahmen der Diplomarbeit sprengen würde. Es kann jedoch festgehalten werden, dass ein guter Support von Text weniger stark benötigt wird, sondern stattdessen ein hoher Anteil von medialer Funktionalität für eine moderne Programmierung von Benutzeroberflächen nötig ist.

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. In diesem Kapitel wird auf typische Besonderheiten eingegangen und der Begriff der hardwarenahen Programmierung definiert.

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 [17], 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 die Programmierung für eingebettete Systeme umfasst und zusätzlich auch die Entwicklung von 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 im Vordergrund [18, S. 14 f.]:

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

Es folgen Erklärungen für jeden Aspekt 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 der Rechenkapazität ist auch der Arbeitsspeicher eingeschränkt. Es ist üblich, den Arbeitsspeicherbedarf einer Anwendung zu limitieren.

Entwicklungskosten im Widerspruch zu Produktionskosten
Die Ziele niedrige Entwicklungs- bzw. Produktionskosten stehen im wechselseitigem Widerspruch. Entwicklungskosten bezeichnen 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 Entwicklungs- und Produktionskosten statt. Die Extreme sind in der Praxis selten zielführend. Diese Diplomarbeit verfolgt mit dem Prototyp 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.

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.

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 und in folgender Tabelle in Bezug zu den Kriterien dargestellt, siehe Tabelle 1: Typische Anforderung an hardwarenahe Programmierung, übernommen von [18, S. 16].

Tabelle 1: Typische Anforderung an hardwarenahe Programmierung, übernommen von [18, 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 aufgrund des Zeitpunktes der Quelle nicht zwangsläufig aktuell sein müssen, jedoch das Verhältnis der Entwicklungs- und Produktionskosten aufzeigen. Die Kategorie Niedrig repräsentiert beispielsweise einen Mikrocontroller, welcher in Digitaluhren verbaut wird [18, S. 16]. Die Kategorie Mittel könnte für das Kontrollsystem eines Fahrstuhls stehen, bei welchem die graphische Benutzeroberfläche aus einem einfachen LED-Display besteht. Die Kategorie Hoch umfasst unter anderem Anwendungen für mobile Endgeräte, daher auch Smartphones.

Detaillierte Beispiele zu den einzelnen Kategorien sind verfügbar [18, S. 16 ff.]. Der Prototyp befindet sich in der rechten Spalte der Tabelle 1: Typische Anforderung an hardwarenahe Programmierung, übernommen von [18, S. 16]. Damit liegt er in der Kategorie Hoch. Diese Einschränkung wurde getroffen, weil nur in dieser Kategorie ansprechend visuell aufbereitete graphische Benutzeroberflächen zum Einsatz kommen. Die anderen beiden Kategorien haben ihre Berechtigung für eingebettete Systeme, 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 deswegen sehr aufwendig. 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, in welchen nicht referenzierte Objekte mittels eines Garbage Collectors (zu dt. „Müllsammler“) zerstört werden. Der Garbage Collector bezeichnet dabei einen Algorithmus, welcher sämtliche Abläufe im Programm stoppt sowie überprüft, welche Objekte nicht mehr benötigt werden und diese anschließend zerstört. Eine Anwendung dieser Funktionalität ist aufgrund der benötigten Rechenleistung nicht im Interesse von hardwarenaher Programmierung, obwohl sie zweifellos Entwicklerkomfort bietet. Ebenso ist das dynamische Erstellen von Objekten ungünstig, weil der dynamische Speicherplatz über die Datenstruktur eines Heaps reserviert werden muss [19].

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 Erstellen 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 in dieser Vorgehensweise besteht darin, dass die Freigabe des allokierten Speichers beim Verlassen des Kontexts erfolgt und der Speicher nicht weiter verwendet werden darf. Ein Beispiel dafür sind Funktions-interne Variablen, welche beim Verlassen der Funktion freigegeben werden. Die statische Allokation erfolgt deterministisch und ist keine rechenintensive Operation und somit relativ bedenkenlos durchführbar. Die dynamische Allokation ist im Vergleich deutlich rechenintensiver, 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 bekannt [20].

Zusammenfassend 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 und eine allgemeine Rechnerkapazitäten-schonende Softwareentwicklung. Auf Hardware-intensive Algorithmen, wie beispielsweise den Garbage Collector, soll verzichtet werden.

4 Auszeichnungssprachen

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

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 [21].

Eine Auszeichnungssprache kann nicht wie bei Programmiersprachen üblich direkt nach Maschinencode kompiliert werden. Stattdessen wird ein 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 [22]. 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 Textdokumenten (z. Bsp. „Markdown“)
- das Abbilden von Websites (z. Bsp. „HTML“)
- den Datenaustausch zwischen Rechnerprogrammen (z. Bsp. „XML“)
- die Beschreibung von Serverkonfigurationen (z. Bsp. „YAML“)
- die Definition der Metadaten einer Datei (z. Bsp. „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 graphischen Benutzeroberflächen. 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

Die Auswahl der vorgestellten Sprachen orientiert sich an der Relevanz von graphischen Benutzeroberflächen für die Anwendungsdomäne der Programmierung, der Präsenz im Internet und der Wichtigkeit für die Diplomarbeit. Die Auszeichnungssprache JavaScript Object Notation (JSON) fiel aufgrund von ersterem beispielsweise nicht in die Auswahl, weil sie hauptsächlich für die Serialisierung von Daten genutzt wird und nicht um Oberflächen zu definieren [23].

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 [24] [25]. XML basiert dabei auf der Standard Generalized Markup Language (SGML, zu dt. „Standardisierte Verallgemeinerte Auszeichnungssprache“) [25], welche eine Norm festlegt um Auszeichnungssprachen zu definieren. SGML ist in der International Standard Organisation (ISO) 8879:1986 definiert [26]. Die Norm legt zwei Grundsätze zu Auszeichnungssprachen fest:

- Generalisierte Auszeichnungssprachen sollen 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.
- Generalisierte Auszeichnungssprachen sollen 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 [27], [28], weswegen nur kurz auf markante Besonderheiten der Sprache eingegangen werden soll. XML ist eine Sprache für die Datenserialisierung. 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 oder Zeichenketten repräsentiert werden. Jedes Element 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 eingegangen werden, weil die Auszeichnungssprache nicht primärer Bestandteil der Diplomarbeit ist. Es kann sich an den aufgeführten Quellen weiter informiert werden.

4.2.2 Extensible Application Markup Language (XAML)

XAML ist laut offizieller Entwicklerdokumentation eine von Microsoft entwickelte deklarative Auszeichnungssprache [29]. 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 [30]. 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 an der Definition der graphischen Benutzerschnittstelle und der Implementation der Programmfunktionalität gleichzeitig arbeiten können [29].

XAML-Dateien sind gleichzeitig XML-Dateien mit der Endung „*.xaml“ [29]. XAML kann daher als Untermenge von XML angesehen werden. Microsoft stellt zahlreiche Lehrmaterialien und eine umfangreiche Dokumentation zur Verfügung [31], weswegen an dieser Stelle nur auf für die Diplomarbeit wichtige Besonderheiten eingegangen wird.

Eine der Besonderheiten von XAML ist die Verbindung zu dem dahinterliegenden Quellcode. Über das Muster der Datenbindung, welches in Kapitel 3.1.1 Formulare und Kontrollelemente vorgestellt ist, 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 sofort auf die Benutzeroberfläche überführt werden. Es spielt dabei keine Rolle, ob die Änderungen von außerhalb des Objektes oder von innerhalb angestoßen werden. Ein Einsatz des Softwarekonzepts MVVM, siehe dazu Kapitel 3.1.4 Model-View-ViewModel (MVVM), ist daher naheliegend für eine Applikationsentwicklung mit XAML.

Zusätzlich zu XML verfügt XAML über spezifische Erweiterungen der Auszeichnungssprache, 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 Spracherweiterungen in XAML ermöglichen unter anderem folgende Zwecke [32]:

- 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 [32].

Es ist in XAML ebenso möglich, Quellcode zu hinterlegen und auszuführen, beispielsweise bei einem Klick-Ereignis [33] 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, beispielsweise ausgeprägte Template-Systeme, welche dafür genutzt werden können, die Gestaltung von generischen Elementen auf abstrakte Weise zu bestimmen [34]. Generische Elemente sind Elemente, welche nicht im Konkreten bekannt sind. Für die Diplomarbeit ist jedoch dieses Wissen nicht weiter relevant, weswegen lediglich darauf verwiesen wird.

4.2.3 FX Markup Language (FXML)

FXML ist eine programmierbare, XML-basierte Auszeichnungssprache um Objektgraphen für die Programmiersprache Java zu beschreiben. Sie ist ideal für die Programmierung von Benutzeroberflächen geeignet. [35]

Dabei kann der Prozess der Objektorientierung innerhalb der Auszeichnungssprache über sogenannte Processing Instructions (PI, zu dt. „Prozessinstruktionen“) beeinflusst werden. Die PI stellen dabei eine Erweiterung zu der Auszeichnungssprache XML dar und können genutzt werden um Java-Packete zu importieren. [35]

Die Sprache ist vom Umfang her mit XAML vergleichbar, siehe Kapitel 4.2.2 Extensible Application Markup Language (XAML). So wird ebenso das Muster der Datenbindung unterstützt über Expression Binding (zu dt. „Bindung eines Ausdrucks“). Es ist ebenso möglich Skripte in der Auszeichnungssprache zu integrieren. Dafür wird als Skriptsprache JavaScript verwendet. [35]

Für die Diplomarbeit sind die Unterschiede in der Funktionalität zu XAML nicht relevant. Die Unterschiede in der Implementierung der beiden Auszeichnungssprachen haben jedoch Auswirkungen. Während XAML kompiliert wird und somit nativ lauffähig ist, wird FXML zur Laufzeit geladen und interpretiert [36]. Dieser Unterschied sorgt dafür, dass FXML an Nähe zur Hardware verliert.

4.2.4 Hypertext Markup Language (HTML)

HTML ist die Kernsprache des Internets [37]. Nahezu jede Website nutzt HTML. Die Besonderheit von HTML besteht darin, dass sie viele Elemente und Attribute enthält, welche für die Gestaltung von Websites geeignet sind. Der Standard für diese Auszeichnungssprache ist wie XML von dem World Wide Web Consortium (W3C) vorgeschlagen worden [37]. Die Auszeichnungssprache weist Ähnlichkeit zu XML auf, ist jedoch deutlich weniger restriktiv.

Die Verwendung im World Wide Web (WWW) hat die Auszeichnungssprache 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 fehlerhafte Dokumente oder nicht bekannte Funktionalität möglichst gut dem Nutzer darzustellen. Die Fehlertoleranz hat eine hohe Priorität, welche in der Konsequenz dazu führen kann, dass ein Fehler in einer HTML-Seite nicht während der Entwicklung 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) [38]. XHTML kann somit als Untermenge von XML gesehen werden. Dieser Standard wird wie HTML ebenso von gängigen Browsern unterstützt [38]. Dabei hat jedoch der HTML Standard derzeit eine deutlich größere Verbreitung [39].

4.2.5 Qt Modeling Language (QML)

Die Auszeichnungssprache QML ist laut offizieller Entwicklerdokumentation eine deklarative Auszeichnungssprache, konzeptioniert für die Beschreibung von graphischen Oberflächenkomponenten [40]. QML wurde von der QT Group, einer internationalen Softwarefirma, entwickelt. Mit ihr ist es möglich eine Objekthierarchie zu erstellen, welche anschließend in das Programm geladen werden kann.

Die Auszeichnungssprache QML ist, ähnlich wie XAML, trotz Beschreibung in der offiziellen Dokumentation nicht streng deklarativ. Es ist möglich in ihr Programmfunktionalität aufzunehmen [41]. Ebenso ist es möglich das Muster der Datenbindung zu nutzen. In der offiziellen Entwicklungsdokumentation wird dieses Muster als Property Binding (zu dt.: „Bindung des Eigentums“ bzw. „Bindung der Attribute“, bezogen auf Objekte) bezeichnet [41].

QML baut nicht auf XML auf. Die Sprache nutzt JavaScript-Statements für Funktionsdefinitionen, jedoch enthält sich eigens für die Sprache entwickelte und somit einzigartige Syntaxelemente. Dies hat den Vorteil, dass Lasten von XML nicht mitgeführt werden. So müssen beispielsweise die Element-Typen nicht immer angegeben werden, sondern werden von dem dahinterliegenden Quellcode bezogen [41].

Es sei angemerkt, dass QML Teil der Qt-Plattform ist. Die Plattform steht teilweise unter einer Open-Source-Lizenz und teilweise kommerziell zur Verfügung [42].

4.2.6 YAML Ain‘t Markup Language (YAML)

YAML ist eine benutzerfreundliche Datenserialisierungs-Sprache, welche von Oren Ben-Kiki, Clark Evans und Ingy döt Net entwickelt wurde [43]. Als Sprache für die Datenserialisierung gibt es Anbindungen zu verschiedenen Programmiersprachen, welche den Standard nutzen können. Sie ist mit Focus auf gewöhnliche native Elemente entworfen worden [43]. Das Attribut benutzerfreundlich bezieht sich auf die menschliche Lesbarkeit im Verhältnis zu anderen Auszeichnungssprachen. Die Auszeichnungssprache findet vor allem im Bereich der Serverkonfiguration derzeit eine hohe Anwendung. Ein Beispiel dafür ist Kubernetes, ein modernes Software-System für die Verwaltung von Container-basierten Applikationen über mehrere Rechner [44] [45].

Es wird auf markante Elemente der Sprache eingegangen. Als Grundlage dient folgendes Beispiel, welches aus einer Quelle [46] 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 drei Anstrichen ‚---‘ können Metainformationen zu dem YAML-Dokument stehen.

In dem Beispiel ist zu erkennen, dass ein YAML-Dokument drei Elementtypen enthält: Maps, Arrays und Skalar-Werte. 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 verwendet. 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-Elements 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.

Die weiteren Kapiteln beziehen sich auf Maps, Arrays und Skalar-Werte über YAML-Maps, YAML-Arrays bzw. YAML-Skalar-Werte. Ebenso findet eine Referenzierung von Schlüsseln und Werten einer YAML-Map über YAML-Map-Schlüssel bzw. YAML-Map-Wert statt.

Der YAML-Standard enthält weitere Elemente, beispielsweise Anker und Textblockabsätze, welche für diese Arbeit keine Relevanz haben und deswegen nicht weiter aufgeführt werden.

4.3 Problematik der Typisierung bei Auszeichnungssprachen

Auszeichnungssprachen sollen möglichst strikt sein, siehe dazu Kapitel 4.2.1 Extensible Markup Language (XML). Hinzu kommt, dass aus programmiertechnischer Sicht Quellcode im Allgemeinen und auch bei der Definition einer Datenstruktur innerhalb einer Auszeichnungssprache weitestgehend aussagekräftig sein soll. Überflüssiger Programmcode, welcher keinen nützlichen Informationsgehalt bietet, erschwert die Verwaltung einer Datei. Die für den Programmablauf relevanten Teile sind schwieriger zu finden und der redundante Informationsgehalt bleibt unnötig bestehen. Der Wunsch nach Striktheit in einer Auszeichnungssprache und Ausdrucksstärke in Dokumenten stellt einen Widerspruch dar.

Das Problem der Typisierung von Elementen in Auszeichnungssprachen verdeutlicht diesen Widerspruch. Häufig ist der Typ eines Elementes redundant und anhand von vorherigen Elementen aus dem Kontext eindeutig bestimmt. Beispielsweise sind die RGB-Farbwerte einer Farbe in der Regel ganze Zahlen und werden durch einen entsprechenden Datentyp dargestellt. Wird der Datentyp an dieser Stelle in der Auszeichnungssprache angegeben, ist diese Information redundant. Die Auszeichnungssprache XML sieht dabei jedoch vor, dass jedes Element mit dem Element-Typ angegeben ist um eine strenge Striktheit zu gewährleisten.

Es gibt Auszeichnungssprachen, welche diesen Widerspruch eleganter auflösen, beispielsweise QML. Da QML nicht auf XML aufbaut, kann eine andere Funktionalität angewandt werden. Bei QML werden die Element-Typen implizit angenommen. Dazu wird der dahinterliegende Quellcode mit einbezogen um den Typ eines Elementes zu bestimmen. Lediglich in Fällen in denen der Element-Typ nicht implizit bestimmbar ist, muss er angegeben werden [41]. Da diese Variante in wesentlich weniger redundantem Quellcode resultiert, wird sie für die Entwicklung des Prototyps genutzt.

5 Bewertung von aktuellen Technologien zur Eignung für hardwarenahe Benutzeroberflächenerstellung

In diesem Kapitel werden aktuelle Technologien für die Programmierung von Benutzeroberflächen analysiert und hinsichtlich ihrer Eignung für die hardwarenahe Programmierung ausgewertet. Als Methode dient der Vergleich.

5.1 Zielstellung des Vergleichs

Der Vergleich muss Auskunft geben, inwiefern sich bisherige Technologien für eine hardwarenahe Programmierung von Benutzeroberflächen eignen. Als Grundlage für die Vergleichskriterien werden die Vorbetrachtungen genutzt, siehe dazu Kapitel 3 Vorbetrachtungen zu Möglichkeiten der Programmierung von Benutzeroberflächen.

Insbesondere werden durch den Vergleich die Stärken und Schwächen einzelner Lösungen aufgezeigt.

5.2 Bewertungskriterien

Um einen Vergleich durchzuführen müssen objektive Bewertungskriterien festgelegt werden. Da in dieser Diplomarbeit sowohl die Programmierung von Oberflächen, als auch die hardwarenahe Programmierung im Mittelpunkt stehen, ist es plausibel die Bewertungskriterien in diese beiden Bereiche aufzuschlüsseln. Dies geschieht jeweils in einem eigenen Unterkapitel.

Für jedes Bewertungskriterium wird eine Bewertungsskala von 1 – schlechte Unterstützung bis 3 – sehr gute Unterstützung definiert.

Die Eignung der zu vergleichenden Technologien für die Softwareentwicklung von graphischen Benutzeroberflächen wird anhand der Unterstützung der jeweiligen Softwarekonzepte bestimmt, siehe dazu Kapitel 3.1 Bisherige Konzepte für die Programmierung von Benutzeroberflächen. Der Fokus auf die Softwarekonzepte ist dadurch begründet, dass diese zahlreiche Methoden und Vorgehensweisen der Softwareentwicklung enthalten. Beispielsweise finden sich Vorgehensweisen aus der agilen Softwareentwicklung, den Quellcode gut zu strukturieren und einzelne Softwarebestandteile sauber voneinander zu trennen, in den Softwarekonzepten wieder. Die Technologien unterscheiden sich dabei in der Art und Weise die entsprechenden Konzepte umzusetzen.

5.2.1 Bewertungskriterien für die Unterstützung von Softwarekonzepten für die Erstellung von Benutzeroberflächen

Wie unter Kapitel 5.2 Bewertungskriterien bereits erwähnt, werden in diesem Kapitel die verschiedenen Bewertungskriterien für jedes in der Diplomarbeit vorgestellte Softwarekonzept aufgestellt. Diese sind folgende:

- Formulare und Kontrollelemente, siehe Kapitel 3.1.1
- MVC, siehe Kapitel 3.1.2
- MVP, siehe Kapitel 3.1.3
- MVVM, siehe Kapitel 3.1.4

Jedes Bewertungskriterium wird in einem Unterkapiteln einzeln konkret definiert. Es sei angemerkt, dass pro Kriterium ausschließlich die Eignung der Technologie für das Software-Konzept betrachtet wird. Wie bereits in Kapitel 5.2 Bewertungskriterien angemerkt, stellen die Softwarekonzepte Vorgehensweisen dar den Programmcode zu strukturieren. Es ist daher grundsätzlich möglich jedes vorgestellte Softwarekonzept mit jeder Technologie umzusetzen. Lediglich ist die Technologie nicht immer geeignet.

5.2.1.1 Bewertungskriterium Formulare und Kontrollelemente (F&K)

Das Bewertungskriterium F&K erfüllt eine Technologie, welche das Muster der Datenbindung über alle Schichten von der Datenbankschicht bis in die graphische Benutzerschnittstelle ermöglicht. Eine manuelle Implementation über das Beobachter-Muster ist dabei immer möglich, wird aber für den Vergleich nicht anerkannt, weil lediglich die Technologien untersucht werden sollen.

Die Abstufungen werden wie folgt festgelegt:

(1) Es ist keine Funktionalität für die Datenbindung in der Technologie vorhanden.
(2) Es ist Funktionalität für die Datenbindung zwischen View und Quellcode in der Technologie vorhanden.
(3) Es ist Funktionalität für die Datenbindung zwischen View, Quellcode und Datenbank in der Technologie vorhanden.

5.2.1.2 Bewertungskriterium MVC

Für die Diplomarbeit soll festgelegt werden, dass das Bewertungskriterium MVC von einer Technologie unterstützt wird, wenn eine Einteilung des Quellcodes in die Bereiche Model, View und Controller möglich ist. Es sei dabei festgelegt, dass für den View eine Auszeichnungssprache verwendet werden muss. Es ist darauf zu achten, dass keine Datenbindung zwischen View und Quellcode erzwungen wird, denn das würde dem

Konzept widersprechen. Für eine strikte Implementation dürfen lediglich Nutzereingaben an den dahinterliegenden Quellcode weitergegeben werden.

Im Konkreten bestehen folgende Abstufungen:

(1) Keine Auszeichnungssprache ist in der Technologie vorhandenen.
(2) Es ist eine Auszeichnungssprache in der Technologie vorhanden, jedoch nur nutzbar mithilfe des Musters der Datenbindung.
(3) Es ist eine Auszeichnungssprache in der Technologie vorhanden und keine Datenbindung erforderlich.

5.2.1.3 Bewertungskriterium MVP

Da das MVP-Konzept eine Nähe zu dem MVC-Konzept besitzt, sind die Bewertungskriterien ebenso ähnlich. Eine Technologie unterstützt dieses Kriterium, wenn es eine Separierung des Quellcodes in die Bestandteile Model, View und Presenter zulässt. Dabei gelten gleiche Einschränkungen für den View wie für das Bewertungskriterium MVC, siehe Kapitel 5.2.1.2 Bewertungskriterium MVC. Es muss möglich sein den View über eine Auszeichnungssprache zu definieren.

Der Unterschied zwischen den Bewertungskriterien MVC und MVP ist dadurch gekennzeichnet, dass für die Unterstützung des MVP-Konzeptes keine Referenzierung zum dahinterliegenden Quellcode erfolgt. Der Presenter darf den View kennen, aber der View nicht den Presenter, siehe dazu Kapitel 3.1.3 Model-View-Presenter (MVP). Diese Einschränkung ist dann gegeben, wenn die Auszeichnungssprache streng deklarativ ist, folglich keine Prozesslogik enthält.

Aus diesen Gründen wird für das Bewertungskriterium MVP folgender Maßstab festgelegt:

(1) Keine Auszeichnungssprache ist in der Technologie vorhandenen.
(2) Es ist eine Auszeichnungssprache in der Technologie vorhanden, jedoch nur nutzbar mithilfe des Musters der Datenbindung.
(3) Es ist eine Auszeichnungssprache in der Technologie vorhanden und keine Datenbindung erforderlich. Zusätzlich ist die Auszeichnungssprache streng deklarativ.

5.2.1.4 Bewertungskriterium MVVM

Bei dem Bewertungskriterium MVVM ist die Möglichkeit der Datenbindung zwischen Model und ViewModel entscheidend, siehe dazu Kapitel 3.1.4 Model-View-ViewModel (MVVM). Falls das Muster der Datenbindung nicht in der Technologie vorgesehen ist, jedoch eine Auszeichnungssprache angeboten wird, kann dieses Muster manuell über

das Beobachter-Muster implementiert werde. Die Technologie würde demzufolge in dieser Kategorie nicht die beste Punktzahl erhalten.

Folgender Maßstab wird für das Bewertungskriterium MVVM definiert:

(1) Keine Auszeichnungssprache ist in der Technologie vorhandenen.
(2) Es ist eine Auszeichnungssprache in der Technologie vorhanden, jedoch keine Möglichkeit der Datenbindung zum dahinterliegenden Quellcode möglich.
(3) Es ist eine Auszeichnungssprache in der Technologie vorhanden mit Möglichkeit der Datenbindung zum dahinterliegenden Quellcode.

5.2.2 Bewertungskriterien für den Aspekt der hardwarenahen Programmierung

Die Bewertungskriterien, welche für den Aspekt der hardwarenahen Programmierung gewählt wurden, sollen möglichst genau und objektiv die Anforderungen unter Kapitel 3.4 Anforderungen durch hardwarenahe Programmierung widerspiegeln. Dazu wurden folgende 2 Kriterien bestimmt:

- Nähe zur Hardware
- Portabilität

Ähnlich wie in Kapitel 5.2.1 Bewertungskriterien für die Unterstützung von Softwarekonzepten für die Erstellung von Benutzeroberflächen findet die genaue Definition jeweils in einem Unterkapitel statt.

5.2.2.1 Bewertungskriterium Nähe zur Hardware

Das Bewertungskriterium Nähe zur Hardware gibt Auskunft, inwieweit eine hardwarenahe Programmierung durch die Technologie unterstützt wird. Es wird dabei jede Softwareschicht bestraft, welche die Lösung einführt. Dies schließt die verwendete Hochsprache der Technologie mit ein. Wird beispielsweise der Programmcode in einer virtuellen Umgebung ausgeführt, wie es bei Java der Fall ist und deswegen nicht direkt auf der Hardware, wird dies bestraft. Ebenso werden Algorithmen innerhalb der verwendeten Programmiersprache bestraft, welche sehr viel Rechenkapazität binden. Ein Beispiel ist dabei der Garbage Collector, welcher das gesamte Programm stoppt und anschließend nicht mehr benötigte Objekte findet und freigibt.

Der Maßstab ist wie folgt festgelegt:

1) Die Programmausführung findet in einer virtuellen Maschine statt.
2) Die Programmausführung findet nativ statt, jedoch sind Abstraktionsschichten für die Softwareentwicklung vorhanden

3) Die Programmausführung findet nativ statt ohne Abstraktionsschicht.

5.2.2.2 Bewertungskriterium Portabilität

Das Bewertungskriterium Portabilität gibt Auskunft über die Übertragbarkeit der Technologie auf andere Betriebssysteme und Prozessortypen. Die Bedeutung der Portierbarkeit ist bereits begründet, siehe Kapitel 3.3 Moderne Anforderungen an die Programmierung von Benutzeroberflächen.

Die Portabilität kann durch unterschiedliche Faktoren eingeschränkt werden. Jede Technologie verfügt über Voraussetzungen, welche erfüllt sein müssen, damit die Technologie ausgeführt werden kann. Diese Voraussetzungen resultieren aus der Laufzeitumgebung der Technologie. Je nachdem wie quelloffen die Laufzeitumgebung ist und wie einfach es ist die Laufzeitumgebung auf verschiedene Plattformen zu portieren, werden unterschiedliche Bewertungen festgelegt. Die Programmiersprache C beispielsweise kann auf nahezu allen Betriebssystemen und Geräten ausgeführt werden. Die Programmiersprache C# verfügt über eine Laufzeitumgebung, welche nicht nativ auf Android und iOS vorinstalliert ist. Diese kann, um eine Entwicklung auf den jeweiligen Betriebssystemen zu ermöglichen, portiert werden [47]. Die aufwendigere Portierung hat eine schlechtere Bewertung zur Folge.

Ebenso werden Tools, welche für eine Entwicklung zwangsweise notwendig sind, geahndet. Dies ist dadurch zu begründen, dass ein sogenannter Vendor Lock-in (zu dt. „Lieferanten-Lock-In“) geschaffen wird, welcher die Flexibilität und damit die Portierbarkeit der Technologie auf Plattformen einschränkt. Vendor Lock-In bezeichnet dabei das Eingehen einer Abhängigkeit zu dem Hersteller [48].

Deswegen wird folgender Maßstab festgelegt:

1) Die Technologie ist für eine Plattform oder ein bestimmtes Betriebssystem konzeptioniert worden und bietet keine Portierung an.
2) Die Technologie bietet eine Unterstützung für diverse Plattformen. Eine Laufzeitumgebung ist dabei allerdings nicht immer nativ vorhanden und muss portiert werden. Häufig sind Einschränkungen durch Vorgabe der Tools etc. vorhanden. Teile der Technologie sind meistens unter einer Open-Source-Lizenz verfügbar. Letzteres ist jedoch keine Bedingung zur Erfüllung des Kriteriums.
3) Die Technologie bietet eine Unterstützung für zahlreiche Plattformen. Eine Entwicklung ist mit einer Vielzahl von Tools möglich. Die Lösung ist vollständig unter einer Open-Source-Lizenz verfügbar.

5.3 Ausgewählte Technologien

Für die Erstellung von graphischen Benutzeroberflächen existiert eine Vielfalt an Technologien. Diese Tatsache ist dadurch zu begründen, dass eine große Menge an Programmiersprachen entwickelt wurde und häufig pro Programmiersprache mehrere Technologien existieren um graphische Benutzeroberflächen zu erstellen. Die Auswahl der Technologien fand so statt, dass aus einer Teilmenge von populären Programmiersprachen jeweils ein Vertreter herausgesucht wurde. Bei der Auswahl der Programmiersprachen wurde der TIOBE Index als Grundlage verwendet, ein Index um die Popularität von Programmiersprachen zu bestimmen. Der Index wird anhand von unterschiedlichen Suchmaschinenergebnissen ermittelt von der Firma TIOBE [49].

Dabei ist anzumerken, dass ebenso Technologien ausgewählt wurden, welche nicht hardwarenah sind, weil diese im Bereich der graphischen Programmierung von Benutzeroberflächen durchaus interessant sein können. Ein Beispiel dazu ist die auf JavaScript aufbauende Technologie React Native.

5.3.1 Windows Presentation Foundation (WPF) mit XAML – C#

WPF ist eine von Microsoft entwickelte Technologie für die Applikationsentwicklung für die Windows Plattform. WPF ist ein Teil des .NET Frameworks. [50]

Die Technologie ermöglicht dabei eine Applikationsentwicklung in der Auszeichnungssprache XAML und der Programmiersprache C#. Es besteht eine enge Bindung zwischen der Auszeichnungs- und Programmiersprache. Eine Implementation des MVVM-Konzeptes ist für diese Technologie daher sehr naheliegend, weil der View über die Auszeichnungssprache und das ViewModel elegant über die Programmiersprache C# implementiert werden können.

Eine Portierung auf mobile Geräte wird über Xamarin ermöglicht. Dazu wird eine Laufzeitumgebung von C# jeweils auf die Betriebssysteme Android bzw. iOS in der Applikation eingebettet. Xamarin ist eine Technologie, welche von Microsoft aufgekauft wurde.

5.3.2 JavaFX – Java

JavaFX oder OpenJFX ist eine Technologie für die Applikationsentwicklung für Desktop und eingebettete Geräte mithilfe von Java [51]. JavaFX wurde von Oracle entwickelt, ist aber mittlerweile vollständig Open Source [52].

Da die Technologie auf Java aufbaut, setzt sie eine JVM („Java virtual machine“) voraus um ausgeführt zu werden. Eine Portierung auf andere Plattformen wird ermöglicht, indem die JVM eingebettet wird, wenn sie nicht nativ vorhanden ist.

JavaFX nutzt als Auszeichnungssprache FXML um Objekthierarchien abzubilden [36]. Die Auszeichnungssprache wird zur Laufzeit geladen.

5.3.3 React Native – JavaScript

React Native ist eine Technologie, welche von Facebook Inc. entwickelt wurde und unter einer Open-Source-Lizenz verfügbar ist. React Native ermöglicht eine native Entwicklung von Applikationen für mobile Endgeräte unter Einbezug des React Software-Frameworks [53]. Das Software-Framework bietet Funktionalität in der Programmiersprache JavaScript für die Erstellung von Benutzeroberflächen an [54].

Während das React Framework hauptsächlich für die Einbindung in Websites konzeptioniert ist, wird React Native direkt auf dem Android oder iOS Betriebssystem ausgeführt. Dafür wird beim Start einer Applikation ein eigener Prozessablauf für die Ausführung des JavaScript-Quellcodes gestartet [55]. Die nativen Komponenten für die Oberfläche werden direkt in JavaScript angesprochen. Die Technologie implementiert die verschiedenen Überbrückungen zu nativen Quellcode in Java auf Android bzw. in Objective-C auf iOS. Als Applikationsschnittstelle sind die verschiedenen View-Elemente in JavaScript ansprechbar. Es besteht die Möglichkeit eigene Erweiterungen zu schreiben, sodass Performance-kritische Programmteile ausgelagert werden können.

Weitere Informationen zu React Native sind in einer Quelle verfügbar [53].

5.3.4 Qt – C++

Qt ist eine weitere Technologie um eine Entwicklung von Applikationen zu ermöglichen [56]. Die Technologie verwendet als Programmiersprache C++ und als Auszeichnungssprache QML. Qt enthält zahlreiche Tools und Software-Bibliotheken, welche Funktionalität für unterschiedlichste Domänen anbieten. Qt umfasst eine eigene integrierte Entwicklungsumgebung, Softwarebibliotheken für 2d- und 3d-Graphiken, Multimediafunktionalität, Funktionalität für die Unterstützung verschiedener Eingabemethoden, Integration zu Webanwendungen, Möglichkeiten der Lokalisierung und vieles mehr [57]. Die Bibliotheken sind teilweise kommerziell und teilweise unter einer Open-Source-Lizenz verfügbar.

In der Diplomarbeit liegt das Hauptaugenmerk von Qt auf der Verwendung von QML und C++, jedoch wird Funktionalität, welche über Bibliotheken integriert ist, mit berücksichtigt. Die QML-Dateien werden zur Programmlaufzeit geladen [58].

5.4 Bewertung der Technologien

In diesem Kapitel ist die Bewertung der Technologien bezüglich der einzelnen Kriterien zu finden. Die Tabelle 2 zeigt eine Übersicht der Bewertungen. In der letzten Zeile Gesamtpunktzahl steht die Summe der einzelnen Punktevergaben.

Tabelle 2: Bewertungsübersicht der Technologien zur Implementierung von graphischen Benutzeroberflächen

Kriterium	JavaFX	WPF	React Native	Qt
Eignung für F&K	2	2	1	3
Eignung für MVC	3	3	3	3
Eignung für MVP	2	2	3	2
Eignung für MVVM	3	3	2	3
Nähe zur Hardware	1	2	1	3
Portabilität	2	2	2	2
Gesamt-punktzahl	13	14	12	16

Da die Bewertungsmaßstäbe objektiv definiert wurden, ist der Interpretationsspielraum gering. Die Begründung für einzelne Bewertungsvergaben erfolgt dennoch. Hierbei liegt das Hauptaugenmerk auf besonders guten bzw. schlechten Bewertungen.

Die Technologie Qt unterstützt das Softwarekonzept Formulare und Kontrollelemente als einziges sehr gut. Diese Bewertung ist dadurch zu begründen, dass die Technologie nativ ermöglicht Daten aus einer Datenbank in einem View anzuzeigen und editierbar zu machen. Die Datenpräsentation ist dabei konfigurierbar [59].

Die Technologie React Native ist als einzige für das Softwarekonzept „MVP“ sehr gut geeignet. Der Grund besteht in dem Aufbau der Technologie auf HTML. Es wird keine Datenbindung zwischen HTML und JavaScript von der Technologie unterstützt. JavaFX, WPF und Qt unterstützen alle eine Datenbindung zwischen Auszeichnungssprache und Quellcode. Da die Datenbindung grundsätzlich dem Softwarekonzept MVP widerspricht, wurde an dieser Stelle diese Besonderheit durch eine Vergabe von drei Punkten hervorgehoben. Auf der anderen Seite wird diese Eigenschaft für das Softwarekonzept MVVM benötigt. Deswegen erhält React Native an dieser Stelle nur zwei und die anderen drei Technologien jeweils drei Punkte.

Die Nähe zur Hardware ist bei React Native und JavaFX am schlechtesten, weil hier der Code interpretiert bzw. in einer virtuellen Maschine ausgeführt wird. In WPF wird der C# Code zwar nativ ausgeführt, wird aber dennoch verwaltet. Es existiert in der Sprache beispielsweise ein Garbage Collector, welcher nicht benötigte Objekte entsorgt. Lediglich in Qt und damit C/C++ wird der Code nativ ausgeführt und erhält somit drei Punkte.

Es ist empfehlenswert zu einem gewählten Softwarekonzept eine zugehörige Technologie mit ausschließlich drei Punkten zu nutzen, weil ansonsten unnötig zusätzliche Funktionalität implementiert werden muss oder benötigte Einschränkungen nicht vorliegen.

Es sei an dieser Stelle auf Grenzen des Vergleiches hingewiesen. Das Vorhandensein von qualitativ hochwertigen Software-Bibliotheken ist für die Entscheidung für eine Technologie bedeutsam, kann jedoch objektiv nur schwer bestimmt werden. Deswegen wurde dieses Kriterium in dem Vergleich nur indirekt berücksichtigt. Ebenso wird auf Programmiersprachen-spezifische Eigenschaften nur im Rahmen der Softwarekonzepte eingegangen. JavaScript ist beispielsweise als Skriptsprache eher für kleine bis mittelgroße Projekte geeignet. Die Technologie Qt, welche auf C++ zurückgreift, dagegen durchaus auch für große Projekte mit mehreren Hunderttausend Quellcodezeilen.

5.5 Auswertung

In diesem Kapitel erfolgt die Auswertung des Vergleichs.

Es ist festzuhalten, dass keine Technologie in allen Kriterien die höchste Punktzahl erreicht hat. Daraus lässt sich schlusszufolgern, dass je nach Anwendungsfall eine spezifische Technologie ausgewählt werden muss und keine allgemeingültige Aussage bezüglich einer optimalen Technologiewahl getroffen werden kann.

Des Weiteren ist festzuhalten, dass die Gesamtpunktzahl bei der Technologie Qt am höchsten ausgefallen ist. Dies lässt nachweisen, dass diese Technologie in dem Vergleich am besten abgeschnitten hat und für eine hardwarenahe Programmierung von Benutzeroberflächen Anwendungsfall-spezifisch mindestens in Betracht gezogen werden sollte. Sie zeichnet sich durch eine hohe Hardwarenähe und eine Vielzahl an Software-Bibliotheken für unterschiedliche Funktionalitäten aus, welche in die Technologie integriert sind.

Eine weitere Feststellung kann getroffen werden durch die Tatsache, dass alle in dem Vergleich ausgewählten Technologien im Bereich der Portabilität zwei Punkte erhalten haben. Dieser Umstand ist dadurch zu erklären, dass jede Technologie ihre eigenen Tools und Software-Bibliotheken enthält, wodurch eine Kombination aus mehreren vorgestellten Technologien in der Praxis eine große Herausforderung darstellt und selten

einen Mehrwert schafft. Der Vendor Lock-in ist in jeder vorgestellten Technologie zumindest in Ansätzen da, wodurch die Flexibilität und damit auch die Portabilität jeder Technologie eingeschränkt ist.

Für eine innovative, neue und moderne Lösung soll dabei die Portabilität nicht eingeschränkt werden. Es ist daher eine Lösung zu entwerfen, welche unabhängig von einer bestimmten Plattform, diversen Tools und einer bestimmten Entwicklungsumgebung genutzt werden kann. Ebenso ist festzuhalten, dass keine Lösung drei Punkte in Hardwarenähe und Eignung für das Konzept MVP hat. Dieser Missstand soll durch eine neue Lösung im Praxisteil der Diplomarbeit behoben werden.

6 Prototypische Entwicklung

Dieses Kapitel befasst sich mit einer prototypischen Entwicklung für eine neu zu entwickelnde Technologie. Dabei werden zunächst die Anforderungen an den Prototyp definiert und anschließend die dafür entwickelten Lösungskonzepte vorgestellt. Die Lösungskonzepte sollen durch die prototypische Entwicklung unter Beweis gestellt und in Form einer neuen technologischen Lösung bereitgestellt werden. Der Prototyp trägt den Namen Flower (zu dt. „Blume“).

6.1 Anforderungen an den Prototyp Flower

Leitthese der Diplomarbeit ist es neue Ansätze für die hardwarenahe Programmierung von graphischen Benutzeroberflächen zu erforschen. Diese Ansätze werden mithilfe des Prototyps praxisnah veranschaulicht. Im Rahmen der Anforderungsanalyse wurden grundsätzlich folgende Anwendungsdomänen ermittelt:

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

Smart-Home-Anwendungen

Unter den Begriff Smart-Home-Anwendungen wird Software gezählt, welche lokal vor Ort bei Endnutzer auf den Geräten installiert ist, die der Steuerung und Automatisierung hauswirtschaftlicher Abläufe dienen. Dazu zählen unter anderem die Kontrolle der Heizung, der Fenster, der Türen sowie sonstiger Haushaltsgeräte, 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 Automaten an Supermarktkassen um eine Selbstbezahlung zu ermöglichen.

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, weil diese über eine sehr hohe Rechenkapazität verfügen und der Aspekt der hardwarenahen Programmierung dadurch an Bedeutung verliert.

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

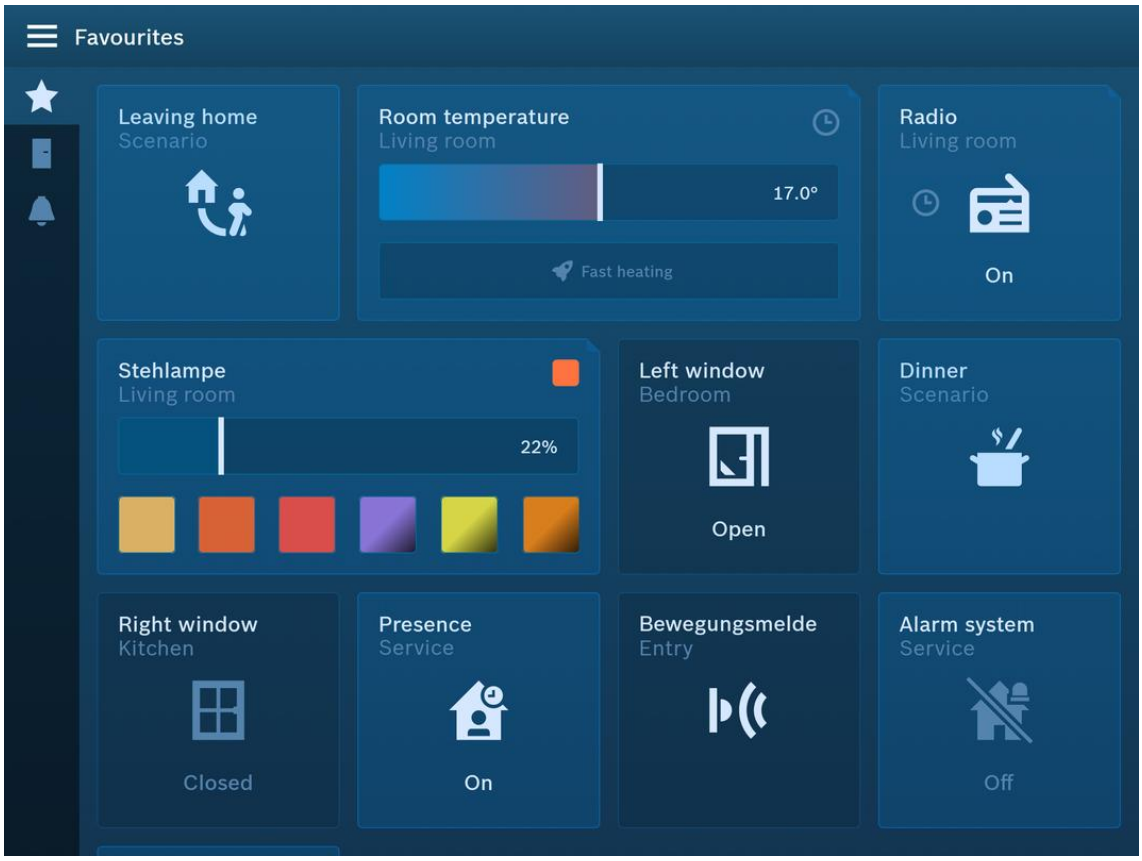

Abbildung 6: Screenshot der Android-Applikation "Bosch Smart Home" von der Robert Bosch GmbH

Die Anforderungen wurden anhand des Screenshots festgelegt.

6.1.1 Funktionale Anforderungen

/101/ Overlay:
Der Prototyp besteht aus einem Overlay. Innerhalb des Overlays müssen Titelleiste, Navigationsbar und Inhalt angeordnet werden.

/102/ Titelleiste:
Die Titelleiste enthält ein Textbanner. Das Textbanner zeigt den Titel des Inhalts.

/103/ Navigationsbar:
Die Navigationsbar muss aus einem Tab Favoriten und weiteren Tabs bestehen. Der Tab Favoriten ist bei Start der Software ausgewählt. Bei Klicken eines Tabs muss sich der Textbanner der 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, besteht der Inhalt aus einem Scroll-View, welcher ein Grid-Layout beinhaltet. In dem Grid-Layout müssen unterschiedlich große Kacheln angeordnet werden. Die Kacheln verfügen über Funktionalität um virtuell hauswirtschaftliche Abläufe zu steuern.

/201/ Kachel Leaving Home (zu dt. „Ausgehen“):
Die Kachel Leaving Home repräsentiert ein Szenario, welches über Klick der Kachel gestartet werden soll. Dazu muss bei Start des Szenarios eine entsprechende Meldung in der Konsole ausgegeben werden. Das Szenario soll nicht weiter implementiert werden.

/202/ Kachel Room Temperature (zu dt. „Raumtemperatur“):
Die Kachel Room Temperature muss über einen Schieberegler verfügen. Er muss über Funktionalität verfügen, mit welcher virtuell die gewünschte Raumtemperatur eingestellt werden kann. Die gewünschte Raumtemperatur muss Werte von 17 °C bis 29 °C annehmen können. Weiterhin muss die Kachel die aktuelle Raumtemperatur anzeigen. Die aktuelle Raumtemperatur soll sich linear an die gewünschte Raumtemperatur anpassen.

/203/ Kachel TV – Living Room (zu dt. „Fernseher - Wohnzimmer“)
Die Kachel TV – Living Room steuert einen Fernseher im Wohnzimmer. Dazu müssen die beiden Zustände „Fernseher – an“ und „Fernseher – aus“ angezeigt werden. Durch Klick auf die Kachel muss eine Änderung der Zustände erfolgen.

/204/ Kachel Lamp – Living Room (zu dt. „Lampe – Wohnzimmer“)
Die Lampe im Wohnzimmer muss über einen Schieberegler verfügen. Er muss Funktionalität anbieten für eine stufenlos einstellbare Dimmsteuerung. Dieser Schieberegler soll als Minimum den Wert 0.0% und als Maximum den Wert 100.0% annehmen. In dieser Kachel erfolgt eine Auswahl bezüglich der Lampenfarbe. Es sollen

mehrere Farben zur Auswahl stehen, welche durch Klick auf die jeweilige Farbe ausgewählt werden können. Die aktuelle Farbe muss visuell kenntlich gemacht werden.

/205/ Kachel Left Kitchen Window (zu dt. „Linkes Küchenfenster“)
Diese Kachel steuert den Zustand des linken Fensters in der Küche. Dazu müssen die beiden Zustände „Fenster – offen“ bzw. „Fenster – geschlossen“ angezeigt werden. Durch Klick auf die Kachel muss eine Änderung der Zustände erfolgen.

/206/ Kachel TV – Kitchen (zu dt. „Fernseher - Küche“)
Die Kachel TV – Kitchen steuert einen Fernseher in der Küche. Analog zu Anforderung /203/ müssen die beiden Zustände „Fernseher – an“ und „Fernseher – aus“ in gleicher Weise implementiert werden.

/207/ Kachel Alarm System (zu dt. „Sicherheitssystem“)
Die Kachel Alarm System steuert die Aktivität des Sicherheitssystems. Es müssen die beiden Zustände „Service – an“ und „Service – aus“ implementiert und angezeigt werden. Durch Klick auf die Kachel muss eine Änderung der Zustände erfolgen.

/208/ Kachel Dinner (zu dt. „Abendbrot“)
Die Kachel Dinner repräsentiert ein Szenario, unter welchem der Nutzer individuell Abläufe hinterlegen kann, welche in Bezug zum Abendbrot stehen. Beispielsweise fällt darunter die Zubereitung von abendlichen Speisen. Das Szenario wird durch Klick gestartet. Bei Start muss eine Meldung auf der Konsole ausgegeben werden. Das Szenario soll nicht weiter implementiert werden.

/209/ Kachel Add Favorites (zu dt. „Favoriten hinzufügen“)
Die Kachel Add Favorites repräsentiert Funktionalität um Favoriten hinzuzufügen. Dazu muss bei Klick eine Meldung auf der Konsole ausgegeben werden. Die Funktionalität soll nicht weiter implementiert werden.

/301/ Sprachen
Alle Anzeigetexte des Prototyps müssen in englischer Sprache sein.

6.1.2 Qualitätsanforderungen

/401/ Visuelle Aufbereitung:
Der Prototyp muss ansprechend visuell aufbereitet sein.

6.1.3 Rahmenbedingungen

6.1.3.1 Technisch/ Technologische Rahmenbedingungen

/501/ Statische Favoriten:
Die Favoriten dürfen für den Prototyp als statisch angenommen werden und müssen nicht dynamisch geladen werden.

/502/ Portabilität:
Die Portabilität der zugrunde liegenden Programmiersprache soll gewährleistet bleiben.

/503/ Hardwarenahe Programmierung:
Die Programmierung muss mit einer hardwarenahen Programmiersprache umgesetzt werden.

6.1.3.2 Rechtliche Rahmenbedingungen

/601/ Softwarerechte:
Die Software soll quelloffen sein und unter einer Open-Source-Lizenz zur Verfügung gestellt werden. Es ist zu beachten, dass ausschließlich Software-Bibliotheken genutzt werden sollen, welche ebenso diese Kriterien erfüllen.

/602/ Bildrechte:
Die Urheberrechte für Bilder müssen bei Verwendung berücksichtigt werden.

6.1.3.3 Organisatorische Rahmenbedingungen

/701/ Zeitrahmen:
Der Prototyp inklusive der Lösungskonzepte muss innerhalb der Diplomarbeit entwickelt werden. Deswegen ist der zeitliche Rahmen vom 18.07.2018 bis zum 18.12.2018 eingeschränkt.

6.2 Lösungskonzepte bezüglich der Technologie

In diesem Kapitel sollen die Lösungskonzepte bezüglich der genutzten Technologie für den Prototyp vorgestellt werden.

6.2.1 Verwendete Technologien und Software-Bibliotheken

Für den Prototyp soll auf verschiedene Technologien und Software-Bibliotheken zurückgegriffen werden, welche in diesem Kapitel vorgestellt sind.

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 Compiler- und Plattform-unabhängig und unter einer Open-Source-Lizenz verfügbar [60].

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

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 [61]. Sie ist verfügbar unter einer Open-Source-Lizenz.

Die Imps Engine hat als interne Abhängigkeiten Expat und SFMT, welche durch die Benutzung vererbt werden, aber keine weitere Verwendung im Quellcode des Prototyps finden. Auf die beiden Bibliotheken soll zum Verständnis trotzdem eingegangen werden. Expat ist eine C-Software-Bibliothek um XML-Dokumente zu verarbeiten [62]. SFMT (Abkürzung für „SIMD oriented Fast Mersenne Twister“, wobei SIMD für „Single Instruction Multiple Data“ steht) ist eine C-Software-Bibliothek für die Erzeugung von Pseudo-Zufallszahlen [63]. Dazu wird eine Variante des Mersenne Twister implementiert, welche von Mutsuo Saito und Makoto Matsumoto im Jahr 2006 vorgestellt wurde. Die Variante des Algorithmus verspricht eine bessere Performance im Vergleich zu ähnlichen Zufallsgeneratoren [64]. Da die Generierung von Zufallszahlen für die Diplomarbeit jedoch nicht relevant ist, wird auf den Algorithmus nicht tiefer eingegangen.

6.2.1.3 OpenCV

OpenCV (Open Source Computer Vision Library) ist unter einer Open-Source-Lizenz verfügbar [65]. Sie soll für das Laden von Bilddateien verwendet werden. Die Software-Bibliothek ist für die rechnergestützte Erkennung von Bildern optimiert, wodurch sie wesentlich umfangreicher ist und nur ein minimaler Teil des Umfangs genutzt werden

soll. Es besteht die Möglichkeit den Umfang des Prototyps zu reduzieren, indem das Laden der Bilddateien durch eine andere Software-Bibliothek unterstützt wird. Jeglicher Quellcode dieser Funktionalität, welcher ersetzt werden müsste, soll sich dabei ausgelagert unter „deps/opencv-glue“ befinden, siehe dazu die Anlagen.

6.2.1.4 OpenGL und GLEW

OpenGL ist ein Graphikkarten Standard für hohe graphische Qualität und Performance [66]. Er wird genutzt um graphische Elemente darzustellen. Um die Entwicklung zu vereinfachen wird auf Funktionalität der Imps Engine zurückgegriffen.

GLEW (OpenGL Extension Wrangler Library) ist eine Plattform-unabhängige Software-Bibliothek für das Laden von OpenGL-Erweiterungen, geschrieben in C/C++ und erhältlich unter einer Open-Source-Lizenz. Sie ist geeignet um zu bestimmen, welche Versionen von OpenGL unterstützt werden und welche Erweiterungen verfügbar sind [67]. Die Bestimmung wird zur Laufzeit durchgeführt. Für diesen Zweck soll GLEW genutzt werden.

6.2.1.5 CLion, MinGW, Windows 8.1

Für die Entwicklung soll die IDE CLion von JetBrains genutzt werden [68]. 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 soll MinGW verwendet werden [69], 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 werden soll, 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 keine syntaktische Unterstützung für OOP [70].

Es ist festzuhalten, dass OOP nicht an eine Syntax gebunden ist, sondern ein Programmierparadigma darstellt, nach welchem Daten und zugehöriger Code bzw. Funktionen gekapselt werden [71]. Die Objekte kommunizieren über Funktionen miteinander.

Deswegen ist es grundsätzlich kein Problem, OOP in C durchzuführen. Es ist eine Tatsache, dass populäre Bibliotheken in C, beispielsweise die Softwarebiblitothek „POSIX Threads“ [72], einen objektorientierten Aufbau verfolgen. Um das Beispiel weiter auszuführen, sei auf die zugehörige Spezifikation hingewiesen [73]. In der Spezifikation wird deutlich, dass Funktionen zu einem bestimmten Datentyp, wie beispielsweise *pthread_mutex_t*, über die Namensgebung, in diesem Falle *pthread_mutex**, gruppiert und auf diese Weise an den Datentyp gekapselt werden. Diese Bibliothek ist an dieser Stelle von einem objektorientierten Aufbau geprägt.

Das Paradigma OOP bietet jedoch nicht nur die Kapselung von Daten und Funktionen, sondern auch weiterführende Konzepte und Muster, beispielsweise Vererbung oder das Beobachter-Muster. Um diese Funktionalität bereitzustellen wird die Imps Engine verwendet. Über eine Reihe von Annotationen in den Kommentaren können reguläre C-Strukturen mit Objektfunktionalität aufgewertet werden [74]. Dies wird für die Implementierung des Prototyps genutzt.

Die Klassen sind dabei grundlegend als C-Strukturen aufgebaut. Anhand von Konstruktor-Methoden wird erkannt, dass es sich bei einer C-Struktur um eine Klasse handelt. Es wird entsprechende objektorientierte Funktionalität für die Klasse über die Imps Engine generiert. Konstruktor-Methoden kennzeichnen sich durch die Namensgebung. Sie beginnen mit *init** oder *make** und erhalten als ersten Parameter einen Zeiger auf ein Objekt der Klasse.

Klassen werden dabei häufig, jedoch nichts zwangsweise, über Klassenattribute instanziiert. Klassenattribute werden genutzt um sie der Konstruktor-Funktion zu übergeben. Sie enthalten dabei selber Attribute, welche vor dem Konstruktoraufruf relativ frei definiert werden können. Relativ frei deswegen, weil die Klassenattribute selber ein Objekt sind, welches nur über implementierte Funktionen verändert werden darf. Dieses Muster hat eine Ähnlichkeit zu dem Builder-Muster (zu dt. „Bauer-Muster“) in Java [75] mit dem Unterschied, dass eine Instanziierung der ursprünglichen Klasse nicht innerhalb der Klassenattribute erfolgt, sondern diese dem Konstruktor übergeben werden.

Es sei an dieser Stelle angemerkt, dass optionale Attribute für Klassen über das Muster der Verwendung von Klassenattributen implementiert werden können. Dazu nimmt eine Instanz der Klassenattribute für jedes optionale Attribut bei deren Erstellung einen Standardwert an, welcher optional über die verfügbaren Funktionen verändert werden kann. Dieser Mechanismus soll für die Auszeichnungssprache genutzt werden, um eine optionale Klassenkonfigurierung zu unterstützen.

Es soll im folgenden auf die verfügbaren Annotationen eingegangen werden. Die verfügbaren Annotation lassen sich dabei je nach Verwendungsart in vier Bereiche einteilen:

- Annotationen für Klassenattribute
- Annotationen für Strukturen
- Annotationen für Funktionen
- Annotationen für Dateien

Jeder Bereich besitzt eine unterschiedliche Reichweite. Annotationen für Klassenattribute haben nur Gültigkeit für ein Attribute innerhalb einer C-Struktur. Annotation für Strukturen, Funktionen und Dateien beziehen sich auf Strukturen, Funktionen bzw. Dateien in der Programmiersprache C.

Annotationen für Attribute generieren Funktionen für den Lese- und Schreibzugriff der jeweiligen Attribute. Laut dem OOP-Paradigma sind interne Objektvariablen nur über Funktionen zugreifbar. Dies wird über den Mechanismus vereinfacht, da nicht jede häufig vorkommende Funktion, wie ein einfacher Lese- und Schreibzugriff, implementiert werden muss. Außerdem kann dem ersten Attribut einer Klasse die Annotation *extend* vergeben werden. Diese Annotation ist nur zulässig, falls es sich bei dem ersten Attribut um eine Klasse handelt. In diesen Fall wird eine Vererbung der Klasse durchgeführt [74]. Dazu wird für jede nicht statische Funktion der Superklasse eine Funktion der Subklasse generiert, welche diese aufruft. Eine Funktion ist in diesem Kontext statisch, wenn sie als ersten Parameter nicht einen Zeiger auf sich selbst übergeben bekommt. Konstruktor- und Destruktor-Funktionen werden dabei nicht übernommen.

Das Beobachter-Muster wird über Delegates (zu dt. „Delegierte“) implementiert. Ein Delegate ist dabei eine C-Struktur und umfasst einen Zeiger zu einem korrespondierenden Objekt, sowie beliebig viele Funktionszeiger, welche auf Funktionen zeigen, die auf diesem Objekt ausgeführt werden können. Das korrespondierende Objekt wird der jeweiligen Funktion übergeben. Es wird so ermöglicht, dass das Objekt, sowie die Funktionszeiger beliebig austauschbar sind, wodurch ein ähnlicher Abstraktionsmechanismus erzeugt wird, wie dies in anderen Programmiersprachen, beispielsweise Java, über die Definition eines Interfaces (zu dt. „Schnittstelle“) möglich ist.

Delegates können als Struktur-Annotation *event* erhalten. Dies bewirkt, dass eine zugehörige Event-Klasse (zu dt. „Ereignis-Klasse“) zu dem Delegate erstellt wird. Die Ereignis-Klasse umfasst dabei eine Liste von Delegates, welche über Funktionalität ergänzt und dezimiert werden kann. Es ist über die Ereignis-Klasse möglich, mehrere Delegates gleichzeitig anzusprechen. Dazu werden die jeweiligen Delegates in der Liste angesprochen. Dieses Muster ist interessant für Ereignisse, an denen mehrere Objekt interessiert sind, zum Beispiel ein *onClick*-Ereignis eines Buttons. Es existiert außerdem eine Annotation *register* für Klassenattribute. Diese Annotation steht mit den Ereignissen in Verbindung und erlaubt es die Registrierung von Delegates für ein

Ereignis Klassen-extern durchzuführen. Dazu wird entsprechende Funktionalität generiert. In unserem Beispiel könnte angenommen werden, dass der Button ein *onClick*-Ereignis besitzt, welches über die Annotation *register* verfügt. In diesem Fall kann jedes Objekt, welches den Button kennt, sich für die *onClick*-Methode des Buttons registrieren. Der Button würde dann lediglich intern mit dem Ereignis-Objekt arbeiten, ohne dass es die dazu zugehörigen Objekte kennt und könnte die entsprechenden Ereignisse werfen.

Als letztes Muster soll auf Namensräume (zu engl. „Namespaces“) eingegangen werden. In der Programmiersprache C existiert laut Spezifikation keine Unterstützung für Namensräume, weswegen Funktionsnamen Programm-weit eindeutig sein müssen [70]. Um dies zu gewährleisten soll jede Funktion den Klassennamen beinhalten. Dabei steht der Klassenname zuerst und soll mittels Unterstrich ‚_‘ von dem Funktionsnamen getrennt werden um eine Eindeutigkeit von Funktionen über mehrere Klassen hinweg zu gewährleisten.

Da OOP in der Programmiersprache C nicht primärer Forschungsgegenstand ist, sondern lediglich für die Lösungskonzepte Anwendung findet und diese in dem Kapitel ausreichend erklärt wurden, soll nicht näher auf die Realisierung von OOP in C eingegangen werden.

6.2.3 Speicherverwaltung über Allocation Counting

Um eine Speicherverwaltung zu ermöglichen wurde auf einen Garbage Collector verzichtet und stattdessen ein Speicherverwaltungskonzept genutzt, welches als Allocation Counting (zu dt. „Zählen der Allokierungen“) genannt wird. Dieses Konzept ist in der Imps Engine implementiert und wird innerhalb des Prototyps angewandt.

Das Speicherverwaltungskonzept sieht vor zu jedem Objekt eine ganze Zahl zu hinterlegen, welche die Anzahl an Verwendungen mitzählt, im folgenden bezeichnet als Allocation Counter (zu dt. „Allokierungszähler“). Falls ein Objekt ein anderes Objekt verwendet, muss dieses den Allocation Counter um eins erhöhen. Falls das Objekt nicht mehr benötigt wird, muss der Allocation Counter wieder um eins vermindert werden. Nach der Erstellung eines Objekts wird der Allocation Counter auf eins gesetzt. Falls er null beträgt, wird das Objekt zerstört und darf nicht weiter verwendet werden.

Um Entwicklerkomfort zu ermöglichen existiert ein Autorelease-Pool. Dem Autorelease-Pool können Objekte übergeben werden. Es existiert für jeden Thread (sinngemäß zu dt. „Programmablauf“) ein eigener Autorelease-Pool, welcher geleert werden kann. Bei Leerung eines Autorelease-Pools wird der Allocation Counter von jedem darin enthaltenen Objekt um eins vermindert. In der Implementierung können Objekte dem Autorelease-Pool übergeben werden, welche nicht mehr verwaltet werden müssen und in zukünftigen Routinen nicht mehr benötigt werden. Wird ein Objekt dem Autorelease-Pool übergeben, jedoch von einem anderen Objekt noch benötigt, kann

dieses den Allocation Counter um eins erhöhen. Der Allocation Counter des Objekts wird durch den Autorelease-Pool zwar um eins vermindert, das Objekt bleibt aber dennoch erhalten.

Der Vorteil dieser Speicherverwaltung liegt im Vergleich zum Garbage Collector in der Performance. Während der Speicherfreigabe von nicht mehr benötigten Objekten können alle Programmabläufe fortgeführt werden. Die Nachteile im Vergleich zum Garbage Collector bestehen darin, dass Objekte manuell freigegeben werden müssen. Es muss der Allocation Counter vermindert werden, wobei der Autorelease-Pool an dieser Stelle Komfortabilität schafft. Eine weiterer Nachteil besteht darin, dass eine gegenseitige Referenzierung von Objekten eine Zerstörung verhindert. Zur Veranschaulichung dieses Falls sei gegeben, dass ein Objekt *A* ein Objekt *B* verwendet und das Objekt *B* wiederum Objekt *A*. Jedes Objekt hat den Allocation Counter des anderen Objekts um eins erhöht. Wenn *A* und *B* nicht weiter benötigt werden und nicht referenziert sind, bleiben sie dennoch bestehen, weil der Allocation Counter stets größer gleich eins ist.

Das Problem der Doppelreferenzierung kann effektiv durch eine auf Objektebene hierarchisch geordnete Softwarearchitektur umgangen werden. Weiterhin kann mittels geeigneter Debugging-Funktionalität der Zustand aller Objekte überwacht werden, wodurch eine gegenseitige Referenzierung erkannt werden kann. Letzteres muss jedoch die Entwicklungsumgebung unterstützen.

6.2.4 Verwendung einer Auszeichnungssprache

Es soll für die graphische Oberflächengestaltung eine Auszeichnungssprache angewendet werden. Als Grundlage dient die Auszeichnungssprache YAML.

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

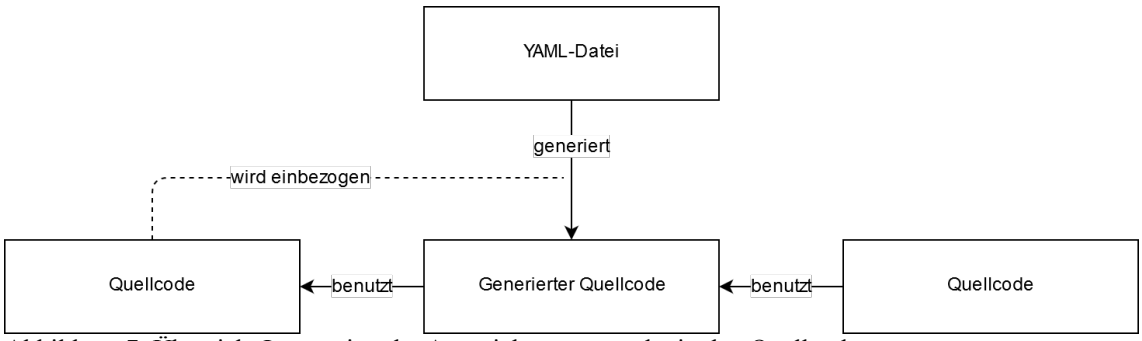

Abbildung 7: Übersicht Integration der Auszeichnungssprache in den Quellcode

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 Abbildung 7 visualisiert. Weiterhin ist festzuhalten, dass in den Schritt der Generierung zugrunde liegender Programmcode mit einbezogen wird. Dies bedeutet, dass die Funktionalität, welche bereits im Quellcode implementiert wurde, Auswirkungen auf das Generierungs-Verhalten hat.

Dieser Umstand lässt sich 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 besteht im Großteil aus dem Wissen über die Art und Weise, wie ein Objekt dieser Klasse instanziiert werden kann sowie welche Attribute diese Klasse besitzt. Die Attribute können dabei zwingend benötigt werden oder optional sein. Auf diese Information wird für die Generierung des Quellcodes zugegriffen, damit die Objekte durch den generierten Quellcode korrekt erstellt werden können.

Die Brillianz in diesem Lösungskonzept steckt in der Tatsache, dass anhand des objektorientierten Aufbaus der Klassen die benötigten Informationen automatisch aus dem Quellcode gewonnen werden können. 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 ebenso keine Dopplung der Spezifizierung der genutzten Header-Dateien statt, weil sie lediglich in dem Build-Tool hinterlegt werden müssen, welches für die Programmerstellung genutzt werden soll. Ein starker Vorteil dieser Lösung im Vergleich zu analysierten technologischen Lösungen besteht darin, dass falls eine Klasse für die Auszeichnungssprache genutzt werden soll, kein zusätzlicher Anpassungsaufwand notwendig ist. Die Art und Weise, wie diese Klasse über programmiertechnischen Weg erstellt wird, soll identisch für den generierten Quellcode übernommen werden. Dies setzt einen genormten Aufbau von Klassen voraus, welcher durch die Nutzung der Imps Engine gegeben ist.

6.2.5 Auflösung der gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode

Innerhalb eines YAML-Dokuments 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 bisherigen analysierten Lösungen, beispielsweise XAML oder QML, erfolgt eine gegenseitige Referenzierung zwischen Auszeichnungssprache und Quellcode, siehe Abbildung 8: Darstellung der ungünstigen gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode.

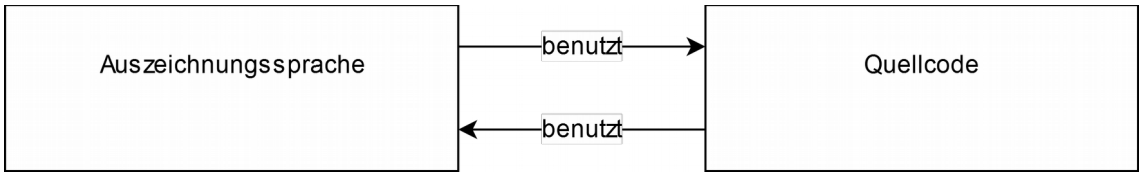

Abbildung 8: Darstellung der ungünstigen gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode

Aus softwaretechnischer Sicht ist dies gefährlich, weil es jede Datei aus der Auszeichnungssprache fest an mindestens eine Datei aus dem Quellcode bindet. Beide Bestandteile sind daher nicht getrennt voneinander wiederverwendbar, sondern immer nur zusammen als Block. Dieses Problem 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 oder es muss die nötige Abstraktion im Quellcode geschaffen werden.

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

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

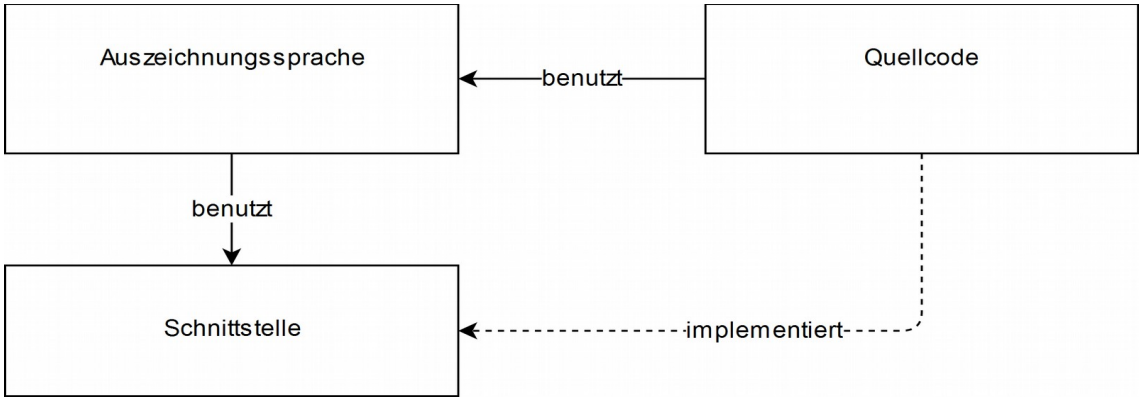

Abbildung 9: Auflösung der gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode durch Schnittstelle

Die Grundidee besteht in der Einführung einer Schnittstelle um eine gegenseitige Referenzierung auszuhebeln. Dabei kennt der Quellcode sowohl Schnittstelle als auch Auszeichnungssprache und implementiert die Schnittstelle, in diesem Fall die *onClick*-Methode. Die Auszeichnungssprache muss entsprechend der Logik lediglich auf die Schnittstelle zurückgreifen und benötigt kein Wissen über den dahinterliegenden Quellcode.

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

Der Vorteil besteht darin, dass die Auszeichnungssprache unabhängig vom Quellcode wiederverwendet werden kann. Softwaretechnisch ist dies klarer getrennt und dadurch besser testbar.

6.2.6 Spezifikation der Auszeichnungssprache

Es wurde bereits im Kapitel 6.2.4 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 6.2.4 soll festgehalten werden, dass die Objekterstellung mithilfe der Auszeichnungssprache vollständig rekursiv möglich sein soll. Dies bedeutet, dass Objekte beliebig tief ineinander geschachtelt werden können.

Die Grammatik wird wie folgt festgelegt und näher erklärt. Nicht rekursiv geschriebene Wörter bezeichnen Grammatikelemente. Rekursiv geschriebene Wörter definieren das jeweilige Grammatikelement. Der Zuweisungsoperator ‚:=‘ ist 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, dass abhängig davon, in welchem Kontext der Parser sich befindet, sind verschiedene grammatikalische Ausdrücke (GA) zulässig und werden unterschiedlich interpretiert. Der Kontext wird anhand der vorhandenen Klassen und Funktionalitäten im Quellcode 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 eines regulären Ausdrucks erklärt.

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

Ein GA Element ist im Vergleich zu anderen grammatikalischen Ausdrücken relativ abstrakt. Ein Element kann ein Objekt, ein Array, eine Struktur oder ein konstanter Ausdruck sein. In welche der vier Kategorien ein Element fällt, wird aus dem aktuellen Kontext bestimmt. 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 auf gleiche Weise wie die Instanziierung der initialen Klasse beschrieben unter dem GA Dokument. Klassen haben notwendige und optionale Attribute. Diese werden über beliebig viele Schlüssel-Werte-Paare angegeben. Die Paare verweisen auf 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 aufgrund der Komplexität in Unterkapiteln erklärt werden.

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 eine feste Länge für einen Datentyp. Aufgrund dieser Tatsache können nur Objekte von einem Typ instanziiert werden. Um diese Einschränkung zu realisieren 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. Das Array müsste durch Zeiger auf den Datentyp *void* (zu deutsch: „Leere“) definiert werden. Die Zeiger könnten dann über Casts (zu deutsch: „Besetzungen“) 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 permanente Abstraktionsschicht einführt, welche nicht umgangen werden kann.

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

Ein GA Array-Objekt hat einen reduzierten Umfang des GA Objekt. Im Vergleich zum Objekt, ist dieser GA ausschließlich durch eine YAML-Map auf grammatikalische Ausdrücke Element definiert. Diese müssen ähnlich dem GA Objekt den notwendigen und optionalen Klassenattributen entsprechen. Funktionalität, wie die Typisierung kann aufgrund der beschränkten Datentyp-Länge nicht verwendet werden. 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 lautet die Bezeichnung dieses Datentyps „Plain old Java Object“ (POJO, zu deutsch: „Blankes, altes Java Objekt“). In der Programmiersprache C sind dies übliche C-Structs ohne Objektfunktionalität aus dem C-Standard [70]. 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 zwei-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 zwei-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 für Strukturen eine zu hohe Komplexität dar.

Die zweite Möglichkeit ist die Nutzung eines YAML-Arrays. Dabei muss der YAML-Array genau der Anzahl der Attribute der Struktur entsprechen. Jedes Array Element wird an dieser Stelle chronologisch genau einem Attribut zugeordnet. Die ersten beiden Möglichkeiten verwenden 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 in der Grammatikspezifikation ermöglicht worden.

Die dritte Möglichkeit besteht in der Erstellung der Struktur aus einem GA Konstante. Dabei wird lediglich auf eine Konstante aus dem Quellcode verwiesen, welche genutzt werden soll.

Egal über welche der drei 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 dem GA Element hängt es von dem Kontext ab, welcher GA gewählt wird. Beispielsweise wird für das aktuelle Element eine Integer-Variable verlangt, stehen ein GA Nummerischer-Ausdruck und ein GA Konstante zur Auswahl. Zeichenkette und Boolescher-Ausdruck kommen dafür nicht in Frage. Diese beiden genannten grammatikalischen Ausdrücke

unterscheiden sich in ihrer Syntax eindeutig voneinander, sodass ein deterministisches Verhalten garantiert werden kann. Festzuhalten ist, dass für einen GA Konstanter-Ausdruck ausschließlich YAML-Skalar-Werte erlaubt sind. Eine YAML-Map oder ein YAML-Array sind ungültig und resultieren in einem Fehler.

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“, zu dt. „Zeichen“), daher ein String in der Programmiersprache 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 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 erforderlich sind, die eine Zahl repräsentieren. Für folgende Datentypen trifft dies 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 GA 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+)*“. Diese sind im Konkreten eine Addition „+“, eine Subtraktion „-“, eine Multiplikation „*“ oder eine Division „/“. 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 [70]. 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 bedingt eingesetzt werden. Dieser Datentyp ist 8-bit groß. Die restlichen 7 Bits bleiben ungenutzt. 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. Eine Konstante kann vielseitig eingesetzt werden: für Objekte, Strukturen und Interne Datentypen der Programmiersprache C. Die Funktionsweise besteht immer darin, 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 jedoch auch Möglichkeiten Variablen zu verändern, welche als konstant definiert wurden. Dies kann für Verwirrung bei Softwareentwicklern sorgen, weswegen eine Nutzung dieser Möglichkeit für die Diplomarbeit ausgeschlossen wurde und nicht weiter in Betracht gezogen wird.

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 eine Zeichenkette „Hintergrundfarbe“ ebenso für eine Konstante mit dem Namen „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, wenn ein GA Name vorliegt, welcher eine Referenz auf eine Variable abbildet, keine Konflikte zu anderen grammatikalischen Ausdrücken existieren. Sollte auf eine Konstante verwiesen werden, welche im Quellcode nicht hinterlegt ist, resultiert dies in einem Fehler.

Zusätzlich verfügt der GA Konstante noch über eine Besonderheit. Konstanten, welche öffentlich zugreifbar sind, sind genau wie Funktionen über einen Namensraum, dem Klassennamen, gekennzeichnet, z. Bsp. *IAColor_red*. Dabei steht der Klassenname *IAColor* für den Namensraum und *red* (zu deutsch: rot) für den Variablennamen, getrennt mittels Unterstrich ‚_‘. Wenn die Konstante in der gleichen Datei hinterlegt wurde, wie die Klasse, 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.

6.2.7 Referenzierung der erstellten Objekten der Auszeichnungssprache im Quellcode

Die Referenzierung von Objekten soll im Quellcode möglich sein. Dabei ist zunächst festzuhalten, dass der Zweck der Auszeichnungssprache darin besteht eine Objekterstellung durchzuführen. Pro YAML-Datei wird ein Wurzel-Objekt erzeugt. Dieses Wurzel-Objekt kann über weitere Objekte und Strukturen verfügen, welche nach einer Grammatikspezifikation rekursiv ineinander geschachtelt werden können, siehe dazu Kapitel 6.2.6 Spezifikation der Auszeichnungssprache.

Es soll auf sämtliche Objekte der Zugriff ermöglicht werden, inklusive dem Wurzel-Objekt und verschachtelten Objekten. 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 bezüglich nicht ausreichendem Arbeitsspeichers auftreten können, doch dieser wird bereits von der Imps Engine über eine Methode *onMemoryWarning* (zu deutsch: „Eintreten einer Speicherwarnung“) abgefangen. Deswegen kann der Rückgabewert problemlos und komfortabel für das Wurzel-Objekt genutzt werden.

Eine größere Herausforderung stellen verschachtelte Objekte dar. 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 6.2.6 Spezifikation der Auszeichnungssprache. Der zugehörige YAML-Map-Wert gibt dabei den Referenznamen an, über welchen das Objekt referenzierbar sein soll. Anschließend kann optional dem Objekterstellungsalgorithmus für jede Referenz ein Zeiger übergeben werden. Anhand des Zeigers kann das Objekt an der gewünschten Stelle abgelegt werden. Da diese Stelle im Quellcode bekannt ist, kann auf das Objekt zugegriffen werden, nachdem der Objekterstellungs-Algorithmus beendet ist.

Es ist anzumerken, dass dieser Mechanismus lediglich für ein Objekt pro Referenznamen funktioniert, weil ansonsten der jeweilige Objektzeiger überschrieben werden würde. Hinzu kommt, dass YAML-Maps unsortiert sind, weswegen die Reihenfolge, in welcher Objekte des gleichen Referenznamens erstellt werden, zufällig sein kann. Dadurch kann bei einer multiplen Referenzierung keine eindeutige Aussage getroffen werden, welches Objekt zuletzt geschrieben wird. Um dieses Problem zu umgehen wurde 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. Für die Diplomarbeit besteht an dieser Stelle keine Notwendigkeit.

6.2.8 Erzeugung von Klassen durch Subklassen

Wie in dem Kapitel 6.2.6 Spezifikation der Auszeichnungssprache dargestellt wurde, kann der GA Objekt einen optionalen YAML-Map-Schlüssel *__type__* beinhalten.

Es ist festzuhalten, dass wenn der optionale Schlüssel nicht gesetzt wird, der Objekttyp anhand des dahinterliegenden Quellcodes bestimmt wird. Daher erwartet beispielsweise eine Klasse *Werkstatt* im Konstruktor ein Objekt von der Klasse *Auto* mit dem Parameternamen *einAuto*, so wird für den YAML-Map-Wert zugehörig zu dem YAML-Map-Schlüssel *einAuto* ein Objekt von der Klasse *Auto* instanziiert. Es sollte allerdings für die *Werkstatt* keine Rolle spielen, ob es sich wirklich um ein Objekt der Klasse *Auto* handelt oder stattdessen um ein Objekt einer Klasse, welches von *Auto* erbt und dadurch sämtliche Funktionalität, welche für die *Werkstatt* benötigt wird mit enthält. Das Vorhandensein der Funktionalität ist durch die Vererbung garantiert. Ein Beispiel einer solchen Subklasse könnte eine Klasse *Sportwagen* sein. Soll nun in einer YAML-Datei ein Objekt der Klasse *Werkstatt* angelegt werden mit einer Instanz der Klasse *Sportwagen* für den Parameter *einAuto*, anstelle von *Auto*, so kann dies durch Aufnahme eines zusätzlichen Wertepaares *__type__* als YAML-Map-Schlüssel und *Sportwagen* als YAML-Map-Wert erfolgen.

Bei diesem Verfahren ist anzumerken, dass lediglich eine Verwendung von Subklassen legitim ist. Eine Verwendung von Superklassen würde bedeuten, dass die Funktionalität der Klasse nicht in vollem Umfang garantiert werden kann. Dieser Fall ist somit unzulässig.

Es fand ebenso die Überlegung statt Schnittstellen für Objekte einzuführen hinter denen sich eine beliebige Klasse verbergen kann, welche die Schnittstelle implementiert. Diese Funktionalität wurde aus Gründen der Komplexität verworfen. Graphische Benutzeroberflächen sind häufig hierarchisch aufgebaut, beispielsweise HTML. Schnittstellen würden die Hierarchie nur unnötig verkomplizieren, zumal mit den YAML-Dateien Personen, welche keine weiterführende Programmierkenntnisse besitzen, arbeiten sollen. Falls in der Praxis konkrete Anwendungsfälle für Schnittstellen ermittelt werden können, ist diese Entscheidung zu überdenken. Für diese Diplomarbeit stellte das Nichtvorhandensein von allgemeinen Schnittstellen zu Klassen keine Probleme dar.

6.2.9 Objekterstellungsreihenfolge

Die Objekterstellungsreihenfolge wurde so festgelegt, dass Objekte und Strukturen von innen nach außen erstellt werden müssen. Dieser Umstand liegt darin begründet, dass äußere Objekte und Strukturen teilweise innere Objekte und Strukturen benötigen, weil sie als Parameter für deren Erstellung benutzt werden. Beispielsweise ist die logische Reihenfolge beim Konstruieren eines Buttons zuerst den Text oder das Hintergrundbild anzulegen, welches der Button nutzt und erst anschließend den Button zu erstellen.

Es ist programmiertechnisch auch möglich zuerst die äußeren Objekte anzulegen, anschließend die inneren und in einem letzten Schritt die entsprechenden Referenzierungen zu setzen. Dabei würden die äußeren Objekte nach dem Erstellen der Inneren verändert werden. Beispielsweise ist es bei unterschiedlichen Layouts üblich, dass Funktionalität angeboten wird, um neue Layout-Elemente hinzuzufügen und wegzunehmen. Hierbei spielt das Muster der Unveränderbarkeit (zu engl. „Immutable Pattern“) eine Rolle. Im strengen Sinn verlangt es, dass Objekte komplett losgelöst von inneren Zuständen und somit unveränderlich sind. Dieses Muster verspricht sowohl eine Vereinfachung des Quellcodes, als auch eine bessere Testbarkeit der Software, weil keine Zustände in den Objekten vorliegen. Für die implementierte Lösung wurde dieses Muster, solange es praktikabel durchführbar ist, berücksichtigt. Deswegen wurde an dieser Stelle die Einschränkung getroffen innerhalb der YAML-Dateien keine Objektveränderung durchzuführen. Es sollen lediglich Objekte erstellt werden. Durch diese Einschränkung ist der Ansatz, Objekte von innen nach außen zu erstellen zwingend notwendig.

Es sei angemerkt, dass durch die Kollektion von Objekten, insbesondere in YAML-Maps und YAML-Arrays, eine parallele Erstellung der Objekte möglich ist. Für den Prototyp soll jedoch lediglich eine serielle Erstellung erfolgen, weil die Thematik der parallelen Algorithmierung komplex ist und die Umsetzung einen neuen Schwerpunkt schafft. Deswegen soll in dieser Diplomarbeit diese Möglichkeit der Effizienzsteigerung nicht weiter berücksichtigt werden.

6.2.10 Objekterstellungs-Injektion

Die Objekterstellungs-Injektion ist eine Funktionalität, welche in die Lösung integriert werden soll. Sie soll erlauben Klassenparameter vor der eigentlichen Erstellung zu setzen bzw. zu injizieren. Es soll so ermöglicht werden, dass nicht nur innerhalb des Quellcodes Parameter für Klassen gesetzt werden können, sondern dies auch von außerhalb geschehen kann. Eine gegenseitige Referenzierung wird dadurch nicht eingeführt. Demzufolge widerspricht dieses Lösungskonzept nicht dem Lösungsansatz beschrieben unter dem Kapitel 6.2.5 Auflösung der gegenseitigen Referenzierung zwischen Auszeichnungssprache und Quellcode.

Die Umsetzung erfolgt über Funktionszeiger und dem YAML-Map-Schlüssel *__ref__*. Dabei wird *__ref__* bereits zur Referenzierung des erstellten Objektes von außerhalb genutzt, siehe dazu Kapitel 6.2.7 Referenzierung der erstellten Objekten der Auszeichnungssprache im Quellcode. Es sollen lediglich Objekte, welche über den Schlüssel *__ref__* und einen eindeutigen Referenznamen verfügen, injizierbar sein. Über den Referenznamen soll es möglich sein, eine Injizier-Funktion als Funktionszeiger zu setzen. Die Injizier-Funktion hat dabei zwei Parameter: Der erste ist eine Referenz auf das Objekt, welches den Objekterstellungsalgorithmus aufruft. So kann die Injektion dynamisch innerhalb des aufrufenden Objektes durchgeführt werden. Der zweite Parameter ist eine Referenz auf die Attribute des zu erstellenden Objektes,

welche in der YAML-Datei definiert wurden. Es sei angemerkt, dass das Objekt zum Zeitpunkt des Injektions-Aufrufs noch nicht erstellt worden ist. Es werden lediglich die Attribute übergeben, welche im Quellcode entsprechend gesetzt werden können um die Injektion durchzuführen. Wenn keine Funktion für ein injizierbares Objekt übergeben wird, werden die Attribute unverändert für die Klassen-Instanziierung genutzt.

Die Funktionalität ist für bestimmte naheliegende Anwendungsfälle unkomfortabel, weil keine Referenzierung von mehreren Objekten möglich ist. Beispielsweise müsste zu jedem Textelement einzeln die Injektions-Routine als Funktionszeiger übergeben werden, wenn der Font zu allen Textelementen einer YAML-Datei über den Mechanismus gesetzt werden soll. Für diese Art von Anwendungsfällen ist die Injektions-Funktionalität deswegen sehr ungeeignet. Stattdessen ist die Eignung zu prüfen um eine *onClick*-Methode eines Buttons zu setzen. Diese Methode kann vor der Erstellung eines Buttons injiziert werden und somit muss die Methode als Programmfunktionalität nicht innerhalb der Auszeichnungssprache gesetzt werden. Dies geht einher mit der Bedingung der deklarativen Eigenschaft von generellen Auszeichnungssprachen, dargelegt unter Kapitel 4.2.1 Extensible Markup Language (XML).

6.2.11 Ressource-Provider Konzept

Das Konzept des Ressource-Provider soll den Zugriff auf dynamische Ressourcen gewährleisten. Es gibt bereits ein Lösungskonzept um den Zugriff auf statische Ressourcen zu ermöglichen, siehe dazu Kapitel 6.2.6 Spezifikation der Auszeichnungssprache unter dem GA Konstante. Der Unterschied zwischen statischen und dynamischen Ressourcen besteht darin, dass statische Ressourcen bereits zu Programmstart angelegt werden und im Laufe der Applikation in der Regel relativ fest sind und selten bis gar nicht verändert werden. Es gibt jedoch Anwendungsfälle in denen es notwendig ist Ressourcen zur Laufzeit zu laden und abhängig vom Programmablauf zu erstellen und freizugeben. Anlehnend dazu existiert das Software-Muster des Lazy Loadings (zu dt. „faules Laden“), welches ein möglichst spätes Laden von Ressourcen empfiehlt. Bei strikter Anwendung des Software-Musters bedeutet dies, dass ein Objekt oder eine Ressource, welche von einem Softwareprogramm benötigt wird, erst zum spätest-möglichen Zeitpunkt angelegt wird. Der Vorteil darin besteht, dass möglichst wenig Arbeitsspeicher beansprucht wird. Es kann jedoch auch hilfreich sein Ressourcen zu bestimmten Zeiten vorzuladen um Stockungen in der Benutzeroberfläche zu vermeiden. Konzepte, welche für statische Ressourcen ausgelegt sind, beispielsweise die Unterstützung von Konstanten, unterstützen diese Vorgehensweise ungenügend.

Es existieren zudem deutlich naheliegendere Anwendungsfälle. Jede etwas umfangreichere Applikation mit einer Benutzerschnittstelle verwaltet Bilder im Programmcode, welche angezeigt werden sollen. Dabei sind Softwarebibliotheken für die Programmierung von Grafikkarten häufig als Zustandsmaschine (zu engl. „State

Machine“) implementiert, beispielsweise auch OpenGL. Dies bedeutet, dass keine Objektstruktur, sondern Datenstrukturen existieren. Im Fall von OpenGL werden beispielsweise Bilder über die Datenstruktur Textur verwaltet. Die Texturen werden global angelegt und sind über eine Schnittstelle nutz- und ansprechbar. Die Texturen können während des Programmablaufs beliebig erzeugt und zerstört werden. In der Regel bauen Software-Technologien für die Oberflächenentwicklung auf diesen Zustandsmaschinen und Datenstrukturen auf und setzen eine für die Entwicklung komfortable Objektstruktur auf.

Im Sinne der hardwarenahen Programmierung soll dieser Zwiespalt möglichst ressourcenschonend aufgehoben werden. Dazu werden die Datenstrukturen in Objekten verwaltet. Über das Konzept des Ressource-Providers erfolgt der direkte Zugriff auf die jeweiligen Objekte in den YAML-Dateien.

Das Konzept muss über die Einführung einer neuen Annotation für C-Funktionen mit der Bezeichnung *resourceProvider* implementiert werden. Annotationen sind erklärt unter Kapitel 6.2.2 Objektorientierte Programmierung in C. Die Annotation soll die zugehörige Funktion zu einem Ressource-Provider aufwerten. Diese Funktion muss als Rückgabewert ein Objekt ausgeben und kann Funktionsparameter beinhalten, welche über die YAML-Dateien definiert werden müssen. Die Definition der Funktionsparameter soll in den YAML-Dateien syntaktisch auf gleiche Weise erfolgen, wie es für die Instanziierung von Klassenattributen vorgenommen wird. Schlussfolgernd bedeutet dies, dass Attribute für Klassen und Parameter für eine zum Ressource-Provider aufgewerteten Funktion in einer YAML-Datei auf gleiche Weise erstellt werden. Der Unterschied besteht ausschließlich darin, dass erstellte Objekte zum einen genutzt werden um eine Klasse zu instanziieren und zum anderen eine als Ressource-Provider annotierte Funktion aufzurufen. Da in beiden Fällen das Ergebnis ein Objekt ist, im ersten Fall ein neu erstelltes, im zweiten Fall ein von der Funktion zurückgegebenes, kann eine nahtlose Integration des Ressource-Providers in der YAML-Datei erfolgen.

Eine zum Ressource-Provider aufgewertete Funktion muss dabei ein Objekt nicht zwangsläufig neu erstellen, sondern kann im Falle von Bildern beispielsweise anhand einer als Funktionsparameter übergebenen Zeichenkette lediglich zurück geben. Dies setzt voraus, dass das Bild bereits erstellt wurde. Die Referenzierung eines Ressource-Providers innerhalb einer YAML-Datei ist unter Kapitel 6.2.6 Spezifikation der Auszeichnungssprache unter dem GA Objekt beschrieben. Der Ressource-Provider ist über einen Namen identifizierbar, welcher in Klammern hinter der Annotation im Quellcode hinterlegt werden muss.

Es sei an dieser Stelle angemerkt, dass die Funktionalität für die Bereitstellung der Funktionsparameter eines Ressource-Providers im Vergleich zu Klassenattributen eingeschränkt ist. Es sind keine optionalen Parameter und Arrays möglich. Der Grund liegt darin, dass Konstruktoren von Klassen über eine eigene Attribut-Klasse optionale Parameter und Arrays implementieren. Diese Funktionalität dem Konzept eines Ressource-Provider hinzuzufügen, würde ihn wesentlich komplexer werden lassen. Es

soll deswegen im Rahmen der Diplomarbeit darauf verzichtet werden. Es konnte kein praktischer Anwendungsfall für die zusätzliche Funktionalität von Arrays und optionale Parametern für dieses Lösungskonzept ermittelt werden.

Für den Prototyp muss mindestens ein Ressource-Provider für die dynamische Verwaltung von Bildern implementiert werden.

6.2.12 Schritte für die Kompilierung des Prototyps

In den vorherigen Unterkapiteln 6.2.1 bis 6.2.11 wurden die verschiedenen Lösungskonzepte vorgestellt. In diesem Kapitel können diese nun zusammengeführt werden. Es soll gezeigt werden, wie die Lösungstechnologie auf Ebene der Kompilierung umgesetzt wird. Die Kompilierung ist dabei der Prozess aus Quellcode ein ausführbares Programm zu erstellen. Folgende drei Schritte müssen durchgeführt werden:

- Generierung der Funktionalität für die Objektorientierung
- Generierung der Objekt-Erstellungsalgorithmen aus YAML
- Reguläre Kompilierung des C-Quellcodes mit generiertem Code

Jeder Schritt wird unter Bezugnahme zu den einzelnen Lösungskonzepten in einem eigenen Unterkapitel dargestellt.

Es sei vorausgesetzt, dass alle notwendigen Quellcodedateien für die Kompilierung und damit für die Erstellung des Programms korrekt über CMake angegeben und miteinander verlinkt sind, inklusive aller YAML-Dateien.

6.2.12.1 Generierung der Funktionalität für Objektorientierung

Als ersten Schritt der Kompilierung erfolgt die Generierung der Funktionalität für die Objektorientierung. Dazu wird generisch Funktionalität erzeugt, welche eine OOP in der Programmiersprache C syntaktisch ermöglicht, siehe dazu 6.2.2 Objektorientierte Programmierung in C. Der generierte Quellcode wird in den vorhandenen Quellcode eingebunden.

6.2.12.2 Generierung der Objekt-Erstellungsalgorithmen aus YAML

Nachdem die Generierung der Funktionalität für die Objektorientierung erfolgt ist, wird aus hinterlegten YAML-Dateien Quellcode generiert. Die YAML-Dateien sind dabei in CMake verlinkt. Dazu wird die Objekthierarchie aus YAML entsprechend den

vorgestellten Lösungskonzepten, siehe Kapitel 6.2.4 bis 6.2.11, unter Hinzunahme der Klassen aus dem Quellcode generiert.

Der generierte Quellcode wird in den vorhandenen Quellcode eingebunden.

6.2.12.3 Reguläre Kompilierung des C-Quellcodes mit generiertem Code

In einem letzten Schritt findet die eigentliche Kompilierung des C-Quellcodes statt. Dazu wird der geschriebene Quellcode zusammen mit verwendeten Softwarebibliotheken, siehe dazu Kapitel 6.2.1 Verwendete Technologien und Software-Bibliotheken und mit dem generierten Code nach Maschinencode kompiliert. Auf diese Weise wird ein Programm erzeugt, welches nativ und hardwarenah lauffähig ist.

6.3 Lösungskonzepte bezüglich des Prototyps

Es sollen in diesem Kapitel und den dazugehörigen Unterkapiteln die Lösungskonzepte bezüglich des Prototyps dargestellt werden. Diese unterscheiden sich zu den vorgestellten Lösungskonzepten in Kapitel 6.2 Lösungskonzepte bezüglich der Technologie insofern, dass sie weniger Allgemeingültigkeit besitzen und in Bezug zu dem Prototyp stehen.

6.3.1 Grober Softwareentwurf mittels MVP-Konzept

Der Prototyp soll das MVP-Konzept nutzen, siehe dazu Kapitel 3.1.3 Model-View-Presenter (MVP). Spezifisch soll die erklärte Variante des Passive View genutzt werden. Die Begründung besteht darin, dass dieses Softwarekonzept die Programmierung von strukturiertem und testbarem Quellcode erlaubt sowie eine hardwarenahe portable Lösung entstehen lässt, welche das MVP-Konzept umsetzt. Für diese Kombination der Anforderungen schnitten bisherige Technologien laut Vergleich, siehe dazu Kapitel 5.5 Auswertung, noch verbesserungswürdig ab. In diesem Bereich soll eine Steigerung angestrebt werden.

Es soll auf den konkreten Aufbau der einzelnen Bestandteile des Prototyps Flower eingegangen werden. Ein Diagramm zur Darstellung der groben Softwarearchitektur resultiert aus dem Softwarekonzept MVP, siehe dazu Abbildung 4: Zusammenspiel zwischen Model, View und Presenter. Deswegen wurde auf eine Darstellung im Fließtext verzichtet. Die Abbildung befindet sich stattdessen in den Anlangen wieder unter der Bezeichnung „Abb. Grobe Softwarearchitektur Flower, Packetdiagramm“.

Es sei angemerkt, dass außerhalb der Bestandteile des Models, der Views und des Presenters eine Applikations-Umgebung existiert, welche Funktionalität zum Starten und Beenden der Applikation implementiert. Die Umgebung soll Model, View und Presenter erzeugen und diese entsprechend miteinander verlinken.

Abgesehen von der zu implementierenden Software kommen Softwarebibliotheken zum Einsatz, siehe dazu Kapitel 6.2.1 Verwendete Technologien und Software-Bibliotheken. Auf diese soll an entsprechender Stelle innerhalb des Quellcodes zurückgegriffen werden.

6.3.2 Aufbau des Models

Das Model bildet den fiktiven Zustand des Hauses ab. Dazu sind zu den funktionalen Anforderungen entsprechende Variablenwerte in dem Model zu hinterlegen. Diese Werte lassen sich der Tabelle 3: Übersicht der Variablen des Models entnehmen.

Tabelle 3: Übersicht der Variablen des Models

Datentyp	Variablenname	Beschreibung
float	*currentRoomTemperature*	Repräsentiert die aktuelle Raumtemperatur
float	*targetRoomTemperature*	Repräsentiert die eingestellte Raumtemperatur
int	*livingRoomLampColorIndex*	Enthält den Index der aktuell ausgewählten Lampenfarbe für das Wohnzimmer
bool	*isTvLivingRoomOn*	Stellt den Zustand des Fernsehers im Wohnzimmer dar (1 – an, 0 – aus)
bool	*isTvKitchenOn*	Stellt den Zustand des Fernsehers in der Küche dar (1 – an, 0 – aus)
bool	*isLeftKitchenWindowOpen*	Stellt den Zustand des linken Küchenfensters dar (1 – offen, 0 – geschlossen)
bool	*isAlarmSystemOn*	Stellt den Zustand des Alarm-Systems dar (1 – an, 0 – aus)

Die Variablen sind innerhalb des Models in einer Klasse *HomeState* zu implementieren. Über Funktionalität sollen sie als Schnittstelle nach außen lese- und schreibbar sein. Änderungen an den Variablenwerte sind auf der Konsole auszugeben. Für die Variable *currentRoomTemperature* ist außerdem ein Ereignis zu implementieren, welches ausgeführt wird, sobald sich dieser Wert ändert.

6.3.3 Aufbau der Views

Da der Prototyp hauptsächlich aus einer Ansicht besteht, soll er aus einer Klasse *MainView* modelliert werden, welche sämtliche Logik der Anzeige abdeckt. Es sind für einzelne Bestandteile, beispielsweise die quadratischen Kacheln, Klassen anzulegen, welche von dem *MainView* benutzt werden um eine Wiederverwendbarkeit innerhalb des Views zu gewährleisten.

Die Objekterstellung der Elemente der graphischen Benutzeroberfläche ist über YAML-Dateien zu realisieren. Dazu sind folgende drei YAML-Dateien innerhalb des Views zu erstellen:

- OverlayLayout.yaml
- ContentFavorites.yaml
- SmallTileLayout.yaml

Die Datei *OverlayLayout.yaml* verfolgt den Zweck die Titelleiste, die Navigationsbar und einen Platzhalter für den Inhalt zu erstellen und anzuordnen, siehe Anforderung /101/, /102/ sowie /103/. Die Datei *ContentFavorites.yaml* soll einen Scroll-View mit Grid-Layout erstellen, siehe Anforderung /104/. Dazu sind alle benötigten Kacheln in dem Grid-Layout zu erstellen, siehe Anforderungen /201/ bis /209/. Da kleine quadratische Kacheln relativ häufig vorkommen, ist als Abstraktion dafür eine eigene Klasse zu implementieren, welche alle quadratischen Kacheln modelliert. Die Objekte der Klasse sollen über die Datei *SmallTileLayout.yaml* erstellt werden. In der Datei *ContentFavorites.yaml* ist diese Klasse wiederzuverwenden.

Die Klasse *MainView* muss die definierte Objekterstellung in *OverlayLayout.yaml* und *ContentFavorites.yaml* ausführen und die Objekte entsprechend miteinander verlinken, sodass die Funktionalität der Navigationsbar implementiert ist und der Inhalt Favoriten an entsprechender Stelle angezeigt wird.

Es sei angemerkt, dass laut der MVP-Variante des Passive View möglichst keine Applikationslogik in dem View enthalten sein darf. Der Programmcode soll so strukturiert sein, dass er möglichst transparent alle Benutzereingaben weitergibt und selber die Benutzereingaben nicht unnötig weiterverarbeitet.

6.3.4 Aufbau des Presenters

Der Presenter übernimmt die Aufgabe aus dem MVP-Softwarekonzept als Vermittler zwischen View und Model zu agieren. Dazu erhält er bei Instanziierung ein Objekt der Klasse *MainView* bzw. *HomeState*. Als Mittelsmann geht er auf Änderungen oder geworfene Ereignisse im *MainView* bzw. *HomeState* ein und führt entsprechende Funktionalität aus.

6.3.5 Aufbau der Umgebungsklassen

Es gibt weitere Klassen, welche zur Softwarearchitektur des Prototyps gehören, jedoch nicht in dem MVP-Softwarekonzept enthalten sind. Diese lassen sich verallgemeinert als Umgebungsklassen bezeichnen. Die Umgebungsklassen enthalten Funktionalität, um den Prototyp zu starten. Dazu gehören die Initialisierung der genutzten Softwarebibliotheken, die Erstellung des Applikationsfensters und das Laden der benötigten Bilder sowie das Bereitstellen der Schriftarten. Die Bilder und Schriftarten sollen dabei in einer eigenen Klasse *Resources* (zu dt. „Ressourcen“) nutzbar gemacht werden. Damit diese innerhalb der YAML-Dateien dynamisch referenziert werden können, ist in der Klasse das Konzept des Ressource-Provider, siehe Kapitel 6.2.11, anzuwenden.

Des Weiteren muss die Umgebungsfunktionalität je ein Objekt der Klassen *MainView*, *HomeState* und *Presenter* anlegen, verwalten und korrekt miteinander verlinken. Dies schließt ein, dass der *MainView* graphisch anzuzeigen und der *HomeState* pro Zeitintervall zu updaten ist, damit eine Anpassung der aktuellen Raumtemperatur zur eingestellten Raumtemperatur Klassen-intern vorgenommen werden kann, siehe dazu Anforderung /202/.

6.3.6 Änderungsweitergabe erläutert anhand der Raumtemperatur

Um das Zusammenspiel zwischen *HomeState*, *Presenter* und *MainView* im Prototyp zu erläutern, insbesondere die Weitergabe von Veränderungen, soll als Beispiel die Raumtemperatur gewählt werden. Dazu sei folgendes Klassendiagramm gegeben, siehe Abbildung 10: Klassendiagramm zur Veranschaulichung der Raumtemperaturänderung.

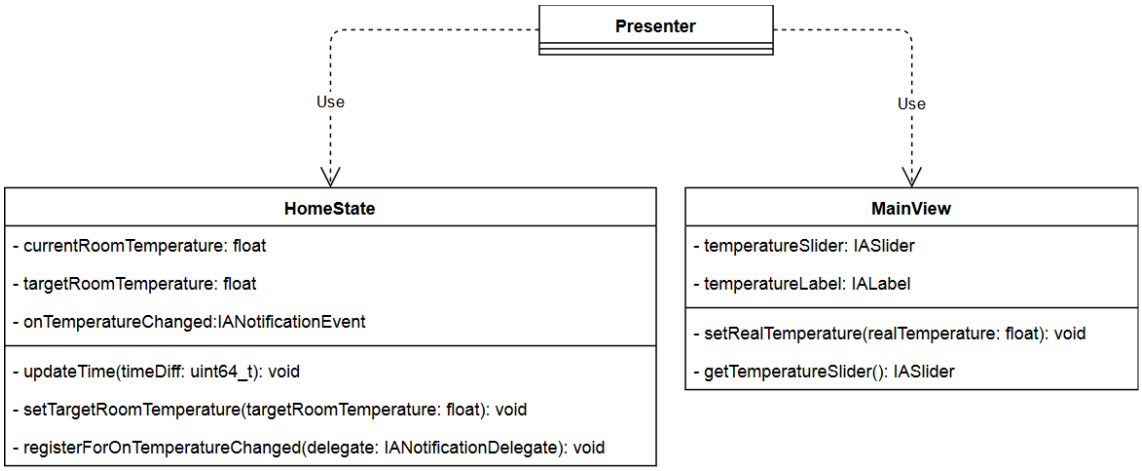

Abbildung 10: Klassendiagramm zur Veranschaulichung der Raumtemperaturänderung

Es sei angemerkt, dass dieses Klassendiagramm nicht vollständig in Bezug auf Funktionen und Variablen der jeweiligen Klassen ist, jedoch die notwendige Information für ein Verständnis des Beispiels enthält.

In dem Diagramm lassen sich die drei Grundbestandteile des MVP-Konzeptes erkennen. Den *HomeState* als Model, den *Presenter* als Presenter und den *MainView* als View. Die Bestandteile wurden bereits einzeln beschrieben, siehe dazu Kapitel 6.3.2 bis 6.3.4.

Es sei darauf hingewiesen, dass *HomeState* und *MainView* den *Presenter* nicht kennen. Der *Presenter* kann auf Änderungen der Klasse *HomeState* eingehen, indem er sich für das Ereignis *onTemperatureChanged* registriert. Nutzereingaben kann er verarbeiten, indem er sich auf das entsprechende Ereignis des Objekts *temperatureSlider* registriert. Die Klasse *IASlider* wirft dabei ein Ereignis bei einer Änderung des Schiebereglers durch den Nutzer. Dieser Aspekt ist in dem Klassendiagramm nicht dargestellt.

Eine Änderung der Raumtemperatur ist in zwei Abläufe zu unterteilen. Einerseits muss die Änderung der eingestellten Raumtemperatur betrachtet werden und andererseits die Änderung der realen Raumtemperatur. Es soll zunächst die Änderung der eingestellten Temperatur betrachtet werden, siehe Abbildung 11: Sequenzdiagramm zur Änderung der eingestellten Raumtemperatur.

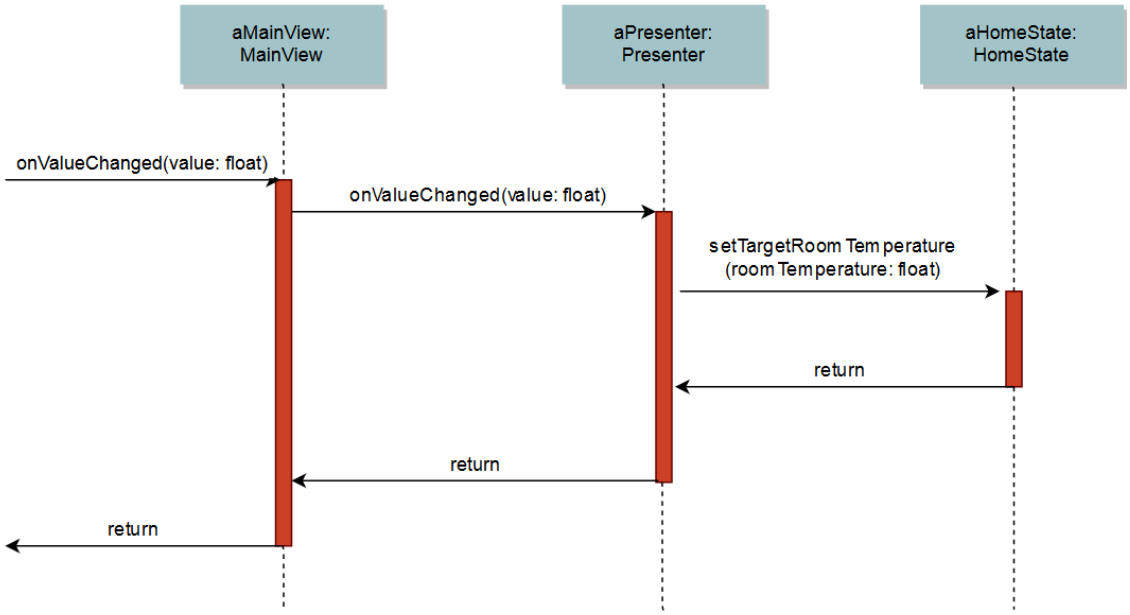

Abbildung 11: Sequenzdiagramm zur Änderung der eingestellten Raumtemperatur

In der oben stehenden Abbildung sind die Objekte *aMainView*, *aPresenter* und *aHomeState* Instanzen der Klassen *MainView*, *Presenter* bzw. *HomeState*. Der Ablauf zur Änderung der eingestellten Raumtemperatur wird von dem Nutzer eingeleitet. Dazu setzt der Nutzer einen neuen Wert mithilfe des Schiebereglers für die Raumtemperatur. Der neue Wert wird über ein *onValueChanged*-Ereignis an das Objekt *aMainView*

weitergeleitet. Das Ereignis wird von dort aus an *aPresenter* weitergeleitet. Dies setzt voraus, dass der *Presenter* sich im Vorfeld für das entsprechende Ereignis registriert hat. Der *Presenter* führt nun die eigentliche Verarbeitung der Nutzereingabe durch, indem er *aHomeState* über die entsprechende Funktionalität updatet. Im Anschluss werden die Prozessabläufe beendet und der weitere Programmfluss ausgeführt. Die Änderung der eingestellten Raumtemperatur wird auf diese Weise an das Model übergeben.

Der Ablauf zur Veränderung der realen Raumtemperatur und damit die simulative Anpassung an die eingestellte Raumtemperatur ist dargestellt in der Abbildung 12: Sequenzdiagramm zur Änderung der realen Raumtemperatur.

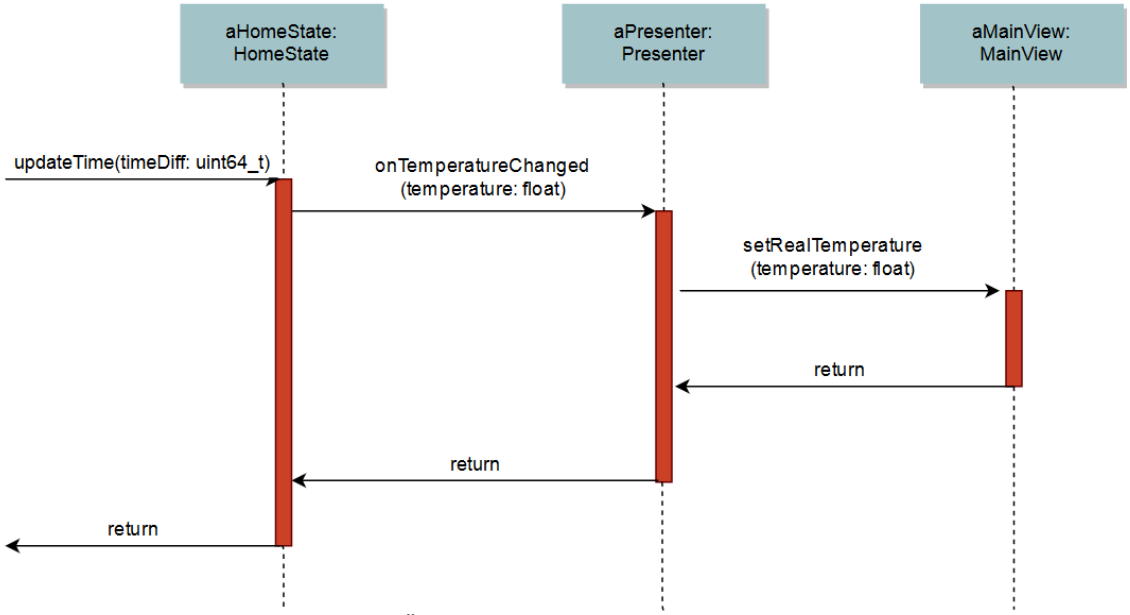

Abbildung 12: Sequenzdiagramm zur Änderung der realen Raumtemperatur

Die Veränderung der realen Raumtemperatur wird von der Applikationsumgebung angestoßen. Dazu wird in regelmäßigen Abständen die Funktion *updateTime* auf das Objekt *aHomeState* ausgeführt. Die Funktion übergibt die Zeitdifferenz zum letzten Funktionsaufruf. Innerhalb des Models in der Klasse *HomeState* wird nun abgekapselt von anderen Programmteilen die Anwendungslogik ausgeführt und die reale Raumtemperatur linear an die eingestellte Raumtemperatur angepasst. Falls es dabei zu einer Veränderung gekommen ist, wird der *Presenter* über ein *onTemperatureChanged*-Ereignis über diese Veränderung informiert, siehe dazu oben stehende Abbildung. Der *Presenter* verarbeitet anschließend das Ereignis und kann den *MainView* über die Funktion *setRealTemperature* anpassen. Der *MainView* kann im Anschluss die Veränderung des *HomeState* anzeigen. Abschließend werden die Prozessabläufe jeweils über ein *return* (zu dt. „Rückgabe“) beendet und der Programmfluss kann fortgesetzt werden.

Die Veränderung von anderen Zuständen in dem *HomeState* und *MainView* ist auf gleiche Funktionsweise umgesetzt worden, sodass auf weiterführende Erklärung verzichtet wird.

6.4 Implementation

In diesem Teil soll auf gewonnene Erkenntnisse während der Implementierung eingegangen werden. Die Implementierung ist erfolgreich durchgeführt worden. Die Anforderungen aus Kapitel 6.1 und die Lösungskonzepte aus den Kapiteln 6.2 und 6.3 konnten umgesetzt werden. Es folgt ein Screenshot des implementierten Prototyps Flower, siehe Abbildung 13: Screenshot Flower, Grafiken erstellt durch Martin Krautschick. Sämtliche Grafiken sind durch Martin Krautschick erstellt worden [12].

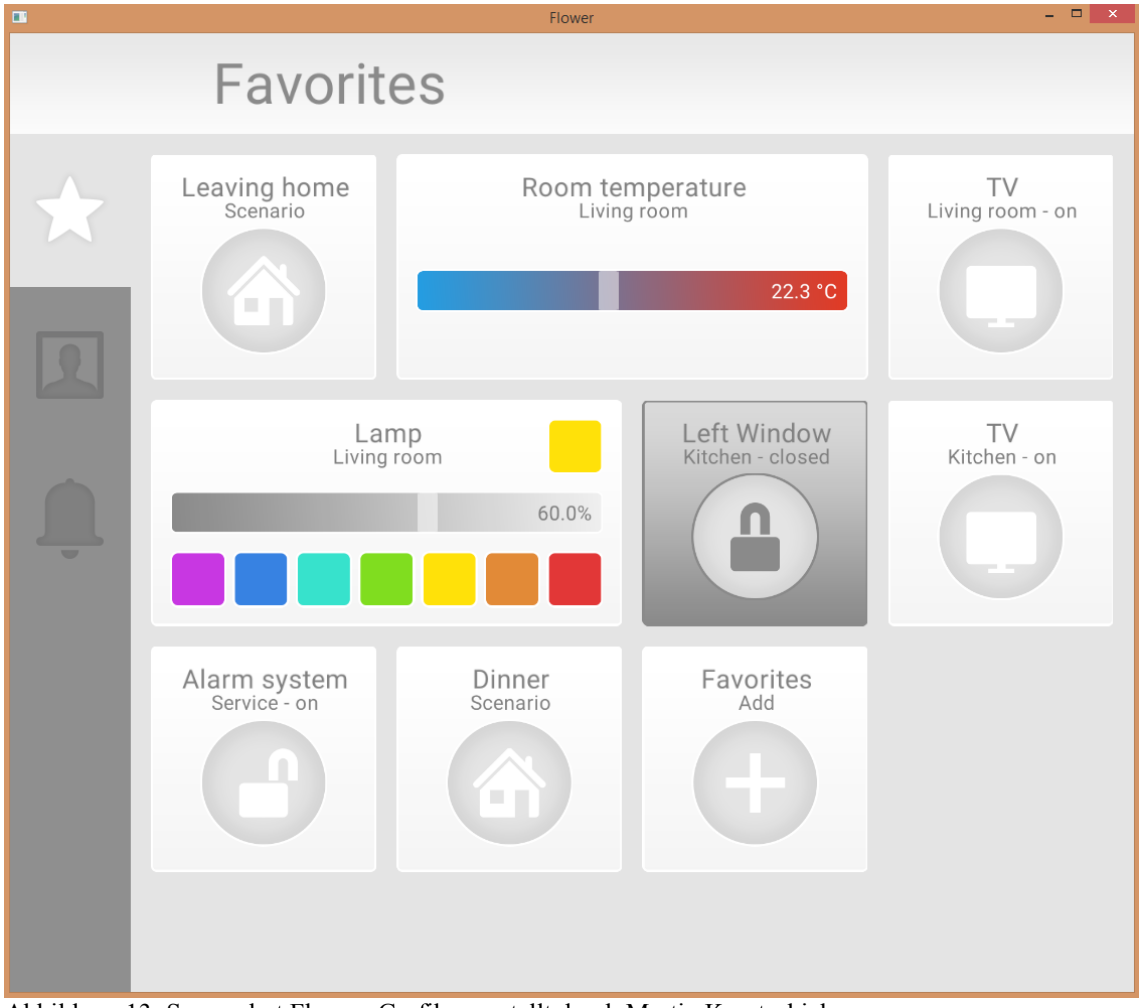

Abbildung 13: Screenshot Flower, Grafiken erstellt durch Martin Krautschick

6.4.1 Eignung OOP in Verbindung mit Allocation Counting

Über die OOP konnte die Implementierung zielführend durchgeführt werden. Oberflächen-Elemente fügen sich problemlos in die objektorientierte Programmierung ein. Die daraus resultierenden Objekthierarchien sind nachvollziehbar.

Die Speicherverwaltung über Allocation Counting hat sich ebenso bewährt. Die über YAML-Dateien erstellten Objekthierarchien werden elegant durch das Allocation Counting unterstützt. Der Grund liegt darin, dass jedes Objekt jeweils den Allocation Counter der benötigten Objekte aus der nächsten Hierarchieebene erhöhen kann und somit diese Objekte nicht gelöscht werden. In der Software muss dann in Bezug auf die Speicherverwaltung der YAML-Dateien lediglich das Wurzelobjekt verwaltet werden. Bei Freigabe wird der Allokation Counter von den Objekten aus den unterliegenden Hierarchieebenen vermindert und diese Objekte werden ebenso freigegeben. Dieser Aspekt ermöglicht eine einfache Speicherverwaltung.

6.4.2 Layouts

Um die Anforderungen und Lösungskonzepte umzusetzen, wurde eine Vielzahl von Layouts erstellt. Die Layouts dienen dabei jeweils dem Zweck, GUI-Elemente anzuordnen. Auf in der Praxis bewährte Layouts im Rahmen des Prototyps soll näher eingegangen werden. Jedes Layout erbt von einer Basisklasse *Layout*, welche Funktionalität wie das Anzeigen eines Hintergrunds implementiert.

6.4.2.1 Grid-Layout

Ein Grid-Layout ist dafür konzeptioniert worden, um Elemente in einem zwei-dimensionalen Raster anzuordnen. Das Layout wurde für den Inhalt der Favoriten verwendet.

6.4.2.2 Flow-Layout

Ein Flow-Layout zeigt Elemente in einer Leiste an, die horizontal oder vertikal sein kann. Es eignet sich, um Elemente innerhalb dieser Leiste zu positionieren und anzuordnen. Für die Navigationsbar wurde dieses Layout verwendet.

6.4.2.3 Scroll-Layout

Ein Scroll-Layout zeigt den Inhalt in einem scrollbaren Fenster an. Dazu kann die Länge des Inhalts spezifiziert werden um die Scroll-Länge zu steuern. Der Inhalt der Favoriten befindet sich in einem Scroll-Layout, damit er stets vollständig einsehbar ist, sollte der Bildschirm nicht ausreichen.

6.4.2.4 Card-Layout

Ein Card-Layout oder Stack-Layout setzt verschiedene Inhalte direkt übereinander. Es ist daher möglich, einzelne Inhalte relativ frei anzuordnen und die Reihenfolge der Überlagerungen zu bestimmen. Für den Prototyp wurde dieses Layout in erster Linie als Platzhalter in der Datei *OverlayLayout.yaml* für die Favoriten und in der Datei *ContentFavorites.yaml* als Platzhalter für die aktuelle Farbe der Lampe im Wohnzimmer gewählt.

6.4.3 Auszeichnungssprache in YAML

Die Implementation der Grammatik der Auszeichnungssprache in YAML hat funktioniert. Die Kompilierung in C-Code konnte über Perl-Skripte problemlos umgesetzt werden.

Es sei an dieser Stelle darauf hingewiesen, dass laut der Grammatik der Auszeichnungssprache, Objekte innerhalb eines YAML-Arrays im Vergleich zu YAML-Map-Werten eingeschränkt konfiguriert werden können. Für die Gründe sei auf das GA Array hingewiesen. Bei der Implementierung des Prototyps sind durch die Einschränkung keine negativen Auswirkungen entstanden. Dabei ist anzumerken, dass das GA Array ausschließlich für die Konfigurierung von Elementen für Layouts verwendet wurde. Jedes Element eines Layouts ist innerhalb eines YAML-Arrays identisch und verfügt über ein Attribut *content*, welches auf den Inhalt des Elements verweist. Das Element kann dadurch zusätzlich konfiguriert werden, indem beispielsweise die Position oder Länge des Elements abhängig vom Layout definiert werden kann.

Es ist ebenso anzumerken, dass die Einschränkung der Objekte eines YAML-Arrays durch Zeiger auf *void* optional umgangen werden kann. Dazu muss der YAML-Array auf Elemente verweisen, welche über Attribute verfügen, die als Zeiger auf *void* definiert sind. Die Attribute können über die entsprechenden Attributsnamen referenziert werden. Hinter dem Zeiger auf *void* kann sich nun ein beliebiges Objekt verbergen, wodurch die Einschränkung aufgehoben wird und in der YAML-Datei der volle Funktionsumfang zur Objekterstellung genutzt werden kann. Die Erkenntnis der optionalen Aufhebung wurde in der Implementierung gewonnen und ist in Ergänzung

zu der Erklärung bezüglich Zeiger auf *void* unter dem GA Array zu sehen, siehe dazu Kapitel 6.2.6 Spezifikation der Auszeichnungssprache.

6.4.4 Konstanten

Das Konzept der Konstanten ist in dem View enthalten. Innerhalb des Views gibt es je eine Datei *Colors.h* und *Constants.h* in welcher Konstanten für Farben bzw. für die Positionierung von GUI-Elementen hinterlegt sind. In den YAML-Dateien werden die Konstanten referenziert.

6.4.5 Ressource-Provider-Konzept

Das Ressource-Provider-Konzept wurde sehr erfolgreich umgesetzt. Grafiken und Fonts werden in der Datei *Resources.h* über mehrere Ressource-Provider global zur Verfügung gestellt und in einzelnen YAML-Dateien genutzt.

6.4.6 Objekterstellungs-Injektion

Das Lösungskonzept der Objekt-Injektion, siehe Kapitel 6.2.10 Objekterstellungs-Injektion, hat sich für die Implementierung nicht bewährt. Der Grund liegt in der Umständlichkeit, welche sich in der Praxis als Hürde erwies. Um beispielsweise mittels der Objekt-Injektion eine *onClick*-Methode eines Buttons zu setzen müssen zwei Funktionen implementiert werden, die Injektions-Funktion und die *onClick*-Methode. Da in der Programmiersprache C jede Funktion explizit implementiert werden muss, führte dies während der Implementierung des Prototyps schnell zu zahlreichem unnötigen Injektions-Quellcode. Das Problem wurde verstärkt, weil die Injektions-Funktion in der Regel nur die *onClick*-Methode gesetzt hatte, sodass die Relevanz der Injektions-Funktion in Frage gestellt wurde.

Als Alternative wurden Ereignisse eingesetzt. Die *onClick*-Methode wurde nicht zur Erstellung des Button gesetzt sondern nach Abschluss der Objekterstellung. Daraus resultierte, dass keine Injektions-Funktion mehr benötigt wurde und die *onClick*-Methode jeweils über einen Funktionsaufruf des Buttons gesetzt wird. Dieser Ansatz hat sich im Gegenteil zu dem Lösungskonzept der Injektion bewährt. Es konnte auf diese Weise eine gegenseitige Referenzierung zwischen Auszeichnungssprache und Quellcode effektiv über je einen Funktionsaufruf umgangen werden.

Das Lösungskonzept der Objekterstellungs-Injektion ist in den Skripten zur Generierung des Quellcodes aus den YAML-Dateien noch enthalten, jedoch wurde es aufgrund der genannten Gründe in dem Prototyp durch Ereignisse ersetzt.

6.4.7 Eleganter Aufbau der Hierarchie in YAML

Während der Implementierung der YAML-Dateien ist aufgefallen, dass tiefere Hierarchieebenen für Verwirrung sorgen können. Dieser Aspekt scheint zunächst logisch, jedoch wird er durch die Tatsache verstärkt, dass YAML-Maps ausschließlich an ihrer Einrückung erkannt werden. Dies bedeutet, dass es schwer nachvollziehbar sein kann, wenn tiefe Verzweigungen beendet werden und in einer höheren Ebene eine YAML-Map fortgeführt wird. An dieser Stelle müssen die Anzahl der Einrückungen verglichen werden, damit festgestellt werden kann, zu welchem Objekt die weiteren YAML-Schlüssel-Werte-Paare gehören.

Dieses Problem kann jedoch über einen geschickten Aufbau der Hierarchie in den YAML-Dateien umgangen werden. Die Lösung besteht darin, die YAML-Schlüssel-Werte-Paare geschickt zu sortieren, sodass hierarchischen Verzweigungen weitestgehend keine YAML-Schlüssel-Werte-Paare folgen müssen. Im Umkehrschluss bedeutet dies, dass YAML-Schlüssel-Werte-Paare, welche in keiner weiteren Verzweigung resultieren, zu Beginn aufgeführt werden. Dieses Prinzip wurde in den YAML-Dateien angewandt.

6.4.8 Kompilierung und Ausführung des Prototyps

Um den Prototyp auszuführen muss die Datei *Flower.exe* im Verzeichnis *bin* ausgeführt werden.

Um den Prototyp zu kompilieren muss CMake ausgeführt werden. CMake muss als Parameter „-DIA_PLATFORM=windows“ erhalten, damit die Imps Engine für die Windows Platform konfiguriert wird. Anschließend wird das Projekt in dem über CMake spezifizierten Build-System erstellt, beispielsweise MinGW. Über das Build-System wird ein ausführbares Programm erstellt. Die Datei *Flower.exe* wurde auf diese Weise erstellt.

6.5 Validierung und Auswertung

In diesem Kapitel wird eine Validierung und Auswertung der entstandenen Lösungstechnologie vorgenommen. Es ist festzuhalten, dass die erarbeiteten Lösungskonzepte erfolgreich umgesetzt und durch die prototypische Entwicklung validiert werden konnten. Die entstandene technologische Lösung wird in die Bewertungstabelle des Vergleichs, siehe Kapitel 5.4, aufgenommen. Die Einordnung der Technologie erfolgt in Tabelle 4: Einordnung der neu entwickelten Technologie in die Bewertungsübersicht der Technologien zur Implementierung von graphischen Benutzeroberflächen.

Tabelle 4: Einordnung der neu entwickelten Technologie in die Bewertungsübersicht der Technologien zur Implementierung von graphischen Benutzeroberflächen

Kriterium	JavaFX	WPF	React Native	Qt	Flower
Eignung für F&K	2	2	1	3	2
Eignung für MVC	3	3	3	3	3
Eignung für MVP	2	2	3	2	3
Eignung für MVVM	3	3	2	3	2
Nähe zur Hardware	1	2	1	3	3
Portabilität	2	2	2	2	3
Gesamt-punktzahl	13	14	12	16	16

Wie in der oben stehenden Tabelle zu erkennen ist, konnte die Technologie jeweils drei Punkte in den Kriterien Portabilität, Nähe zur Hardware und Eignung für MVP erzielen. Damit konnte das Ziel erreicht werden für diese Kombination eine Verbesserung bisheriger Technologien zu realisieren.

Es ist festzuhalten, dass eine Gesamtpunktzahl von 16 Punkten gleich dem Ergebnis der Technologie Qt ist. In einzelnen Bereichen, wie die Eignung für F&K und MVVM wurde eine Verschlechterung zu bisherigen Technologien festgestellt werden. Diese beiden Fakten lassen schlussfolgern, dass Verbesserungen nur in einzelnen Kriterien erzielt werden konnten. Das Fazit des Vergleichs aus Kapitel 5.5, dass keine universelle Technologieempfehlung gegeben werden kann und eine Technologiewahl stets in Abhängigkeit zu dem Anwendungsfall getroffen werden muss, bleibt bestehen.

Ein Aspekt, welcher in weiterführender Forschung untersucht werden kann, ist die Interoperabilität zwischen bisherigen Technologien und Flower. Flower hat in dem Bereich der Portabilität drei Punkte erzielt, weil sie unabhängig von Plattform und Technologie genutzt werden kann und keine Tools enthält, dessen Verwendung von der Technologie zwangsweise benötigt wird. Es wäre deswegen interessant Fälle zu untersuchen Flower mit anderen Technologien zu kombinieren, beispielsweise JavaScript. Der Vorteil einer solchen hybriden Lösung wäre, dass View-Elemente in der Programmiersprache JavaScript implementiert und in einem Browser-Fenster angezeigt werden könnten. Die Applikationslogik könnte hardwarenah in C und die Views in YAML erstellt werden. Auf diese oder ähnliche Weise könnte versucht werden durch hybride Ansätze die einzelnen Vorteile der Lösungen miteinander zu kombinieren. Diese hybriden Ansätze weiter zu untersuchen würde jedoch den Rahmen der Diplomarbeit sprengen.

7 Abschließende Bemerkungen

Eine Wahl einer Technologie zum Start eines Projektes zu treffen ist nie leicht. Unterschiedliche Faktoren spielen eine Rolle und die Anforderungen können sich häufig im Laufe des Projektes ändern, sodass sich eine ursprünglich gute Technologiewahl als ungünstig entwickeln kann.

In der Diplomarbeit wurde gezeigt, dass in bisherigen Technologien zur hardwarenahen, graphischen Benutzeroberflächenprogrammierung Verbesserungspotential besteht. Es wurden mehrere innovative und neuartige Lösungskonzepte aufbauend auf dem aktuellen Stand der Forschung entworfen und implementiert. Die praktische Eignung konnte für viele der entwickelten Konzepte nachgewiesen werden, jedoch blieb der erhoffte Nutzen durch ein Konzept in der Praxis aus. Durch die neu entwickelte Technologie konnte eine Verbesserung in Teilbereichen erzielt und bestätigt werden. Es wurde allerdings auch eine Verschlechterung in anderen Teilbereichen festgestellt. Die Zukunft wird zeigen, ob die Technologie in weiteren Projekten Anwendung findet und die Vorteile die Nachteile überwinden.

Neben dem Wissen, welches sich in dieser Diplomarbeit befindet, sollen zwei bedeutsame Erkenntnisse abschließend hervorgehoben werden. Dies ist zum ersten die Erkenntnis eine Technologiewahl stets unter Betrachtung des Anwendungsfalls zu treffen und zum zweiten, dass die gegenseitige Referenzierung von Auszeichnungssprache und Quellcode softwaretechnisch ungünstig ist, weil sie der deklarativen Eigenschaft von generellen Auszeichnungssprachen widerspricht. Trotzdem wird die deklarative Eigenschaft in fast jeder betrachteten Technologie aufgebrochen. Die gewünschte Einfachheit in der Praxis wird an dieser Stelle einem sauberen Softwaredesign vorgezogen.

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] „Basic UNIX commands“. [Online]. Verfügbar unter: http://mally.stanford.edu/~sr/computing/basic-unix.html. [Zugegriffen: 14-Dez-2018].

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

[6] „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].

[7] „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].

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

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

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

[11] „PNG (Portable Network Graphics) Home Site“. [Online]. Verfügbar unter: http://www.libpng.org/pub/png/. [Zugegriffen: 14-Dez-2018].

[12] K. Merker, „Die für den Prototyp Flower erstellten Bilder wurden angefertigt durch Martin Krautschick in Anlehnung an die Applikation ‚Bosch Smart Home‘ von der Robert Bosch GmbH.“ .

[13] „Top 10 IT Entwicklung Trends in 2018 | ASFAST-EDV“. [Online]. Verfügbar unter: http://asfast-edv.de/blog/top-10-it-entwicklung-trends-in-2018/. [Zugegriffen: 16-Dez-2018].

[14] „Agile Methodology“. [Online]. Verfügbar unter: http://agilemethodology.org/. [Zugegriffen: 16-Dez-2018].

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

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

[17] „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].
[18] M. Barr, *Programming Embedded Systems in C and C++*. O’Reilly Media, Inc., 1999.
[19] „7. Memory : Stack vs Heap“. [Online]. Verfügbar unter: https://www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html. [Zugegriffen: 14-Dez-2018].
[20] „Heap-Fragmentierung – Mikrocontroller.net“. [Online]. Verfügbar unter: https://www.mikrocontroller.net/articles/Heap-Fragmentierung. [Zugegriffen: 22-Nov-2018].
[21] „Markup Language Definition“. [Online]. Verfügbar unter: https://techterms.com/definition/markup_language. [Zugegriffen: 04-Okt-2018].
[22] „Auszeichnungssprache :: ML (markup language) :: ITWissen.info“. [Online]. Verfügbar unter: https://www.itwissen.info/Auszeichnungssprache-markup-language-ML.html. [Zugegriffen: 04-Okt-2018].
[23] „JSON“. [Online]. Verfügbar unter: https://www.json.org/. [Zugegriffen: 14-Dez-2018].
[24] „About W3C“. [Online]. Verfügbar unter: https://www.w3.org/Consortium/. [Zugegriffen: 24-Nov-2018].
[25] „Extensible Markup Language (XML) 1.0 (Fifth Edition)“. [Online]. Verfügbar unter: https://www.w3.org/TR/REC-xml/. [Zugegriffen: 24-Nov-2018].
[26] 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].
[27] „XML Tutorial“. [Online]. Verfügbar unter: https://www.w3schools.com/xml/. [Zugegriffen: 24-Nov-2018].
[28] K. Schild und M. Mochol, „Aufbau von XML- Dokumenten“, S. 69, 2006.
[29] 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].
[30]. „NET | Free. Cross-platform. Open Source.“, *Microsoft*. [Online]. Verfügbar unter: https://www.microsoft.com/net. [Zugegriffen: 24-Nov-2018].
[31] 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].
[32] 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].
[33] 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].
[34] 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].

[35] „Introduction to FXML | JavaFX 11“. [Online]. Verfügbar unter: https://openjfx.io/javadoc/11/javafx.fxml/javafx/fxml/doc-files/introduction_to_fxml.html. [Zugegriffen: 28-Nov-2018].
[36] „Getting Started with JavaFX: Using FXML to Create a User Interface | JavaFX 2 Tutorials and Documentation“. [Online]. Verfügbar unter: https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm. [Zugegriffen: 28-Nov-2018].
[37] „HTML 5.2“. [Online]. Verfügbar unter: https://www.w3.org/TR/html52/index.html#contents. [Zugegriffen: 24-Nov-2018].
[38] „HTML and XHTML“. [Online]. Verfügbar unter: https://www.w3schools.com/html/html_xhtml.asp. [Zugegriffen: 24-Nov-2018].
[39] „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].
[40] „QML Applications | Qt 5.11“. [Online]. Verfügbar unter: https://doc.qt.io/qt-5.11/qmlapplications.html. [Zugegriffen: 24-Nov-2018].
[41] „Property Binding | Qt QML 5.11“. [Online]. Verfügbar unter: http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html. [Zugegriffen: 28-Nov-2018].
[42] T. Q. Company, „Legal | Licensing - Qt“. [Online]. Verfügbar unter: https://www.qt.io/licensing/. [Zugegriffen: 28-Nov-2018].
[43] „YAML Ain’t Markup Language (YAML™) Version 1.2“. [Online]. Verfügbar unter: http://yaml.org/spec/1.2/spec.html. [Zugegriffen: 12-Nov-2018].
[44] „Kubernetes Documentation“. [Online]. Verfügbar unter: https://kubernetes.io/docs/home/. [Zugegriffen: 24-Nov-2018].
[45] „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].
[46] „XML, JSON und YAML im Vergleich“, *predic8.com*. [Online]. Verfügbar unter: http://www.predic8.de/xml-json-yaml.htm. [Zugegriffen: 12-Nov-2018].
[47] asb3993, „Part 1 – Understanding the Xamarin Mobile Platform - Xamarin“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/understanding-the-xamarin-mobile-platform. [Zugegriffen: 28-Nov-2018].
[48] „What is Vendor Lock-In? - Definition from Techopedia“, *Techopedia.com*. [Online]. Verfügbar unter: https://www.techopedia.com/definition/26802/vendor-lock-in. [Zugegriffen: 15-Dez-2018].
[49] „About | TIOBE - The Software Quality Company“. [Online]. Verfügbar unter: https://tiobe.com/company/about/. [Zugegriffen: 28-Nov-2018].
[50] gewarren, „Introduction to WPF - Visual Studio“. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/visualstudio/designers/introduction-to-wpf. [Zugegriffen: 28-Nov-2018].
[51] „Main - OpenJFX - OpenJDK Wiki“. [Online]. Verfügbar unter: https://wiki.openjdk.java.net/display/OpenJFX/Main. [Zugegriffen: 28-Nov-2018].
[52] „JavaFX Developer Home“. [Online]. Verfügbar unter: https://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html. [Zugegriffen: 15-Dez-2018].

[53] „React Native · A framework for building native apps using React“. [Online]. Verfügbar unter: https://facebook.github.io/react-native/index.html. [Zugegriffen: 28-Nov-2018].
[54] „React – A JavaScript library for building user interfaces“. [Online]. Verfügbar unter: https://reactjs.org/index.html. [Zugegriffen: 28-Nov-2018].
[55] B. Evkoski, „React Native: What it is and how it works“, *We Talk IT*, 12-Juni-2017. .
[56] T. Q. Company, „Complete software development framework | Qt“. [Online]. Verfügbar unter: https://www.qt.io/what-is-qt/. [Zugegriffen: 28-Nov-2018].
[57] T. Q. Company, „Libraries & APIs, Tools and IDE | Qt“. [Online]. Verfügbar unter: https://www.qt.io/qt-features-libraries-apis-tools-and-ide/. [Zugegriffen: 28-Nov-2018].
[58] „Interacting with QML Objects from C++ | Qt QML 5.11“. [Online]. Verfügbar unter: http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html. [Zugegriffen: 28-Nov-2018].
[59] „Presenting Data in a Table View | Qt SQL 5.11“. [Online]. Verfügbar unter: http://doc.qt.io/qt-5/sql-presenting.html. [Zugegriffen: 28-Nov-2018].
[60] „CMake“. [Online]. Verfügbar unter: https://cmake.org/. [Zugegriffen: 16-Dez-2018].
[61] „impsapps/impsengine“. [Online]. Verfügbar unter: https://github.com/impsapps/impsengine. [Zugegriffen: 16-Dez-2018].
[62] „libexpat/libexpat: Expat library: Fast streaming XML parser written in C; in the process of migrating from SourceForge to GitHub“. [Online]. Verfügbar unter: https://github.com/libexpat/libexpat. [Zugegriffen: 16-Dez-2018].
[63] „MersenneTwister-Lab/SFMT: SIMD-oriented Fast Mersenne Twister“. [Online]. Verfügbar unter: https://github.com/MersenneTwister-Lab/SFMT. [Zugegriffen: 16-Dez-2018].
[64] „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].
[65] „OpenCV library“. [Online]. Verfügbar unter: https://opencv.org/. [Zugegriffen: 12-Nov-2018].
[66] „OpenGL - The Industry Standard for High Performance Graphics“. [Online]. Verfügbar unter: https://www.opengl.org/. [Zugegriffen: 12-Nov-2018].
[67] „GLEW: The OpenGL Extension Wrangler Library“. [Online]. Verfügbar unter: http://glew.sourceforge.net/. [Zugegriffen: 12-Nov-2018].
[68] „CLion: A Cross-Platform IDE for C and C++ by JetBrains“, *JetBrains*. [Online]. Verfügbar unter: https://www.jetbrains.com/clion/. [Zugegriffen: 12-Nov-2018].
[69] „MinGW | Minimalist GNU for Windows“. [Online]. Verfügbar unter: http://www.mingw.org/. [Zugegriffen: 12-Nov-2018].
[70] „C99 Standard PDF Entwurf“. [Online]. Verfügbar unter: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n897.pdf. [Zugegriffen: 12-Nov-2018].
[71] „Was ist OOP?“ [Online]. Verfügbar unter: https://www.dev-insider.de/was-ist-oop-a-677737/. [Zugegriffen: 13-Nov-2018].

[72] „pthreads(7) - Linux manual page“. [Online]. Verfügbar unter: http://man7.org/linux/man-pages/man7/pthreads.7.html. [Zugegriffen: 13-Nov-2018].
[73] „<pthread.h>“. [Online]. Verfügbar unter: http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html. [Zugegriffen: 13-Nov-2018].
[74] „ImpsEngine: Annotations documentation“. [Online]. Verfügbar unter: https://impsapps.github.io/impsengine-doc/d9/d3b/md_docs__annotations-_documentation.html. [Zugegriffen: 13-Nov-2018].
[75] „Design Patterns: The Builder Pattern - DZone Java“, *dzone.com*. [Online]. Verfügbar unter: https://dzone.com/articles/design-patterns-the-builder-pattern. [Zugegriffen: 05-Dez-2018].

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

F&K	Formulare und Kontrollelemente
FXML	FX Markup Language
GLEW	OpenGL Extension Wrangler Library
GUI	Graphical User Interface
JSON	JavaScript Object Notation
HTML	Hypertext Markup Language
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
OOP	Objekt-oriented Programming
OpenCV	Open Source Computer Vision Library
PI	Processing Instruction
QML	Qt Modeling Language
SFMT	SIMD oriented Fast Mersenne Twister
SGML	Standard Generalized Markup Language
SIMD	Single Instruction Multiple Data
WPF	Windows Presentation Foundation
WWW	World Wide Web
XAML	Extensible Application Markup Language
XML	Extensible Markup Language
YAML	YAML Ain‘t Markup Language

Anlagenverzeichnis

Anlage	Speicherpfad	Zusätzliche Hinweise
Grafiken	*assets/*	Die Grafiken wurden erstellt durch Martin Krautschick [12].
Ausführbares Programm	*bin/*	Das Verzeichnis enthält benötigte dynamische Bibliotheken.
Quellcode des Prototyps Flower	*c/*	Die erstellten YAML-Dateien befinden sich unter *c/shared/view/yaml/* .
Verwendete Softwarebibliotheken	*deps/*	Enthält die Softwarebibliotheken GLEW und OpenCV. Ebenso ist die Plattform-spezifische Funktionalität zum Laden und Verwalten von Bildern für die Windows Plattform abgelegt unter dem Pfad *deps/opencv-glue/* .
Diplomarbeitsdokument	*docs/*	Unter dem Pfad *docs/Abbildungen/* befinden sich alle Abbildungen der Diplomarbeit. Das Dokument der Diplomarbeit befindet sich unter *docs/* .
Schriftarten	*fonts/*	Eine Lizenz zu der verwendeten Schriftart befindet sich unter dem Pfad *fonts/Roboto/*.
Generierte Quellcodedateien	*generated/*	Enthält generierte Quellcodedateien für Objektfunktionalität. Besonders relevant sind *OverlayLayout.h*, *ContentFavorites.h* und *SmallTileLayout.h*, weil diese Quellcodedateien aus den entsprechenden YAML-Dateien generiert wurden.
Softwarebibliothek Imps Engine	*impsengine/*	Unter dem Pfad *impsengine/scripts/perl/generate-generics/* befinden sich die erstellten Perl-Scripte für die Generierung des Quellcodes aus YAML. Die Scripte werden über CMake angestoßen.
Quellen	*quellen/*	Enthält alle im Quellenverzeichnis aufgeführten Quellen.

Anlage	Speicherort	Zusätzliche Hinweise
CMake Build-Skript	*CMakeLists.txt*	Konfiguriert die Kompilierung des Prototyps Flower.