Diplomarbeit

Thema: Moderne Ansätze zur Oberflächengestaltung für hardwarenahe Programmierung

I Inhaltsverzeichnis

Seitenüberschlag:

1	1
2.1	2
2.2	3
3	3
4.1	2
4.2	3
4.3	3
4.4	1
5.1	1
5.2	3
5.3	1
5.4	1
6.1	3
6.2	2
6.3	5
6.4	25
6.5	1
7	1
8	1

Summe: 62

1 Motivation

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

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

2 Begriffsklärungen

Klärung Begriff „modern“

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

Klärung Begriff „hardwarenah“

[Erklärung]

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

Klärung Begriff „Prototyp“

[Erklärung]

- Unterschied Prototyp + Software

Klärung Begriff „graphische Benutzeroberfläche“

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

Klärung Begriff „Softwarearchitekturen“

3 Vorbetrachtungen zu Möglichkeiten der Programmierung von Benutzeroberflächen

3.1 Bisherige Architekturen für die Programmierung von Benutzeroberflächen

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

3.1.1 Formulare und Kontrollelemente

Diese Softwarearchitektur hat in der Literatur keinen spezifischen Namen, aber sie 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 dieser Architektur 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 populärer Ansatz, um dieses Problem zu lösen ist Datenbindung. Die Idee ist Änderungen in einem Datensatz sofort zu propagieren und so an allen nötigen Stellen zu ändern. Wenn die Übertragung abgeschlossen ist, wurde der Datensatz an allen Stellen geändert [2].

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

Interessant ist nun der Anwendungsfall. Sobald der Nutzer anfängt, den Namen in das Textfeld einzugeben wird durch die Datenbindung nicht nur die Variable für die GUI-Oberfläche, sondern gleichzeitig die Variable innerhalb des Programs geändert. Nach Bestätigung des Namens kann sowohl die interne Variable oder die GUI-Variable für

das Ausgabefeld geändert werden. Es spielt keine weitere Rolle, weil die Änderungen in jedem Fall durchgegeben werden.

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

Software-Frameworks implementieren dieses Verfahren intern häufig über das Beobachter-Muster [1]. Dies bedeutet, dass entsprechende Ereignisse ausgelöst werden, wenn sich die Variablen ändern. In den Ereignisbehandlungsroutinen muss die Änderung der jeweils redundanten Datensätzen vorgenommen werden. Die korrekte Verlinkung der jeweiligen Ereignisroutinen kann dabei komplex sein.

Ein weiterer Vorteil von Software-Frameworks ist, dass eine Wiederverwendung von einzelnen Kontrollelementen vereinfacht wird. Oft müssen jedoch Kontrollelemente ebenso applikationsspezifisch erstellt und individuell implementiert werden [1]. Dies stellt sogleich eine Herausforderung in der Betreibung dieser Architektur dar.

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

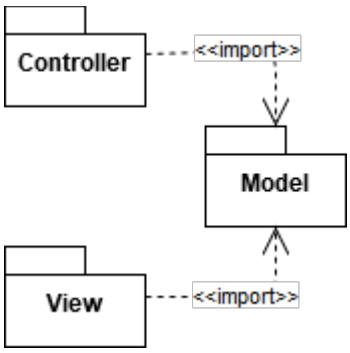
3.1.2 Model View Controller

Der Model View Controller (MVC, engl. für Modell-Präsentation-Steuerung) ist ein weiteres Architektur-Mustur für die Strukturierung von Software zur graphischen Oberflächengestaltung. Es ist mit eines der historisch ältesten Muster für die graphische Programmierung. Zurückblickend, waren in den 1970er Jahren GUI‘s keineswegs gängig. Der MVC ist mit einer der ersten Ansätze, Benutzeroberflächen in großem Umfang zu betreiben. Im Laufe der Zeit, wurden mehrfach Architektur-Muster als MVC bezeichnet, welche jedoch eher wenig mit der ursprünglichen Architektur gemein hatten [1]. Deswegen ist es umso wichtiger, dieses Muster explizit zu besprechen, um eine Abgrenzung von anderen Architektur-Mustern zu ermöglichen.

Die Anfänge dieses Musters sind stark mit Programmiersprache Smalltalk verzweigt, welche zugleich als eine der ersten objektorientierten Programmiersprachen gilt [3]. Sie gehen in die 80er Jahre zurück [1]. Deswegen wird im folgenden Verlauf der Diplomarbeit der Begriff MVC auf das Architektur-Muster, welches Smalltalk verwendete eingegrenzt.

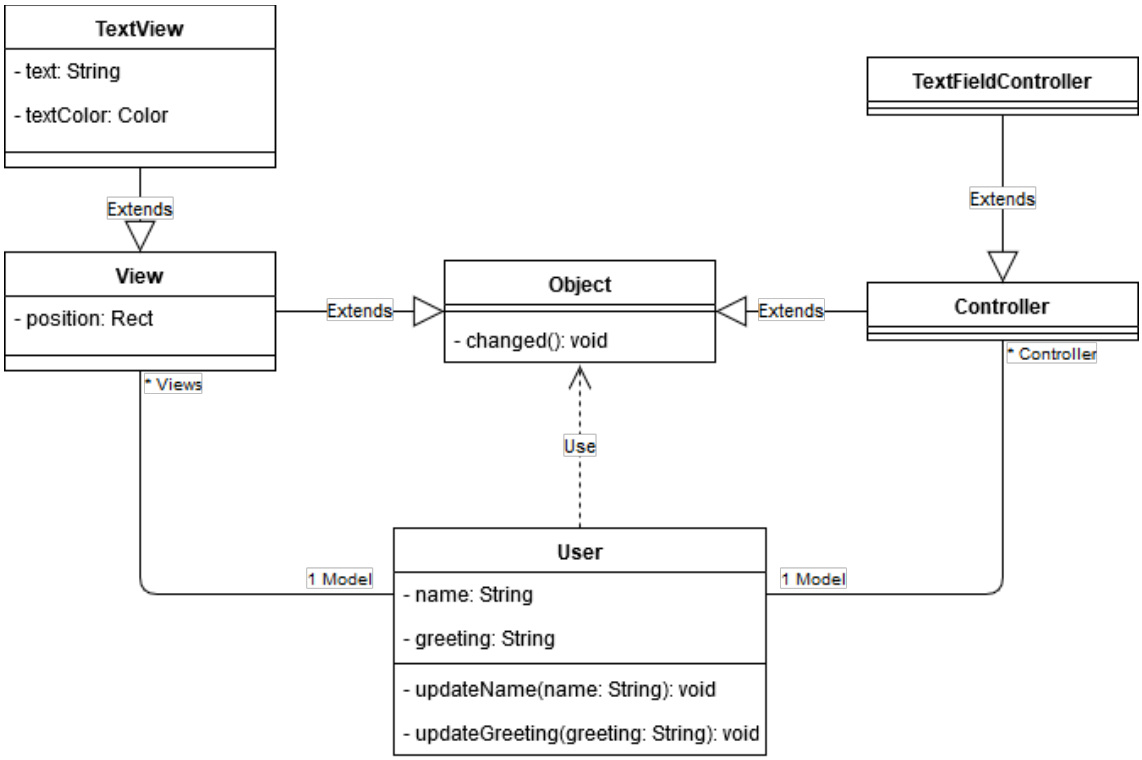
Eine der wichtigsten Errungenschaften von MVC ist die Separierung von graphischen Elementen und Domain-Objekten. Letztere sollten in der Lage sein, komplett losgelöst von jeglichem graphischen Inhalt zu funktionieren. Sie bilden unsere Sicht auf die Welt ab. Sie müssen in der Lage sein, mehrere Präsentation-Schichten zu bedienen. In der Unix-Welt wird dies stark genutzt, da nahezu jede Applikation sowohl über GUI, als auch über Kommandozeile ansteuerbar ist. Dies ist nach aktuellem Stand immer noch der Fall.

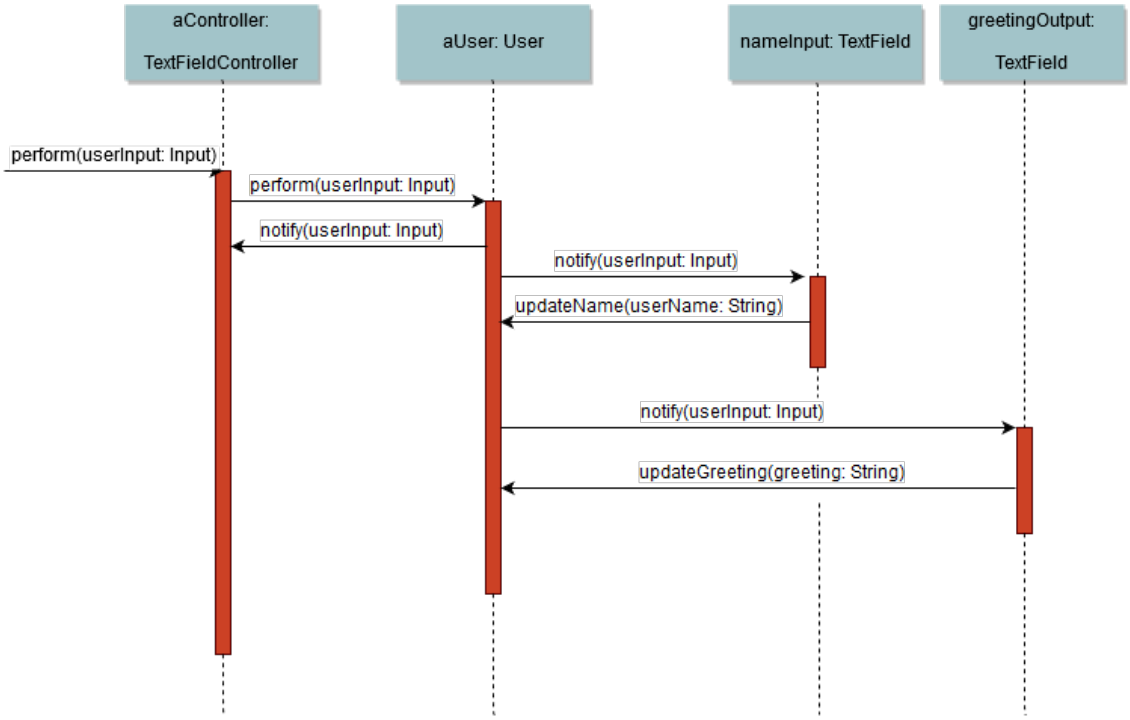
Um das Architektur-Muster im Groben zu beschreiben, werden Domain-spezifische-Objekte innerhalb des Musters als „Model“ bezeichnet. Die graphischen Elemente bilden den „View“. Die Aufgabe des „Controller“ ist es, Benutzereingaben zu verarbeiten. Dabei kennt im strengen Sinn das Model weder Controller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



1. Abbildung: Zusammenspiel Model View Controller

Das Model kann dabei als Abbildung der realen Welt gesehen werden. Anders als in dem Formulare und Kontrollelemente-Architektur-Muster geht dieses Model nicht von Datensätzen aus, die manipuliert werden sollen, sondern von Objekten, mit welchen man in der Anwendung agiert. Als Beispiel soll der gleiche Anwendungsfall dienen, welche unter Formulare und Kontrollelemente ausgewählt wurde. Wenn man die Begrüßungsanwendung über die MVC-Architektur umsetzen würde, so könnte das Model aus einem Objekt bestehen, welche den Namen beinhaltet.


TextView
- text: String
- textColor: Color
Extends
View
- position: Rect
Extends
Object
- changed(): void
Extends
Controller
TextFieldController
Extends
* Views
* Controller
Use
User
- name: String
- greeting: String
- updateName(name: String): void
- updateGreeting(greeting: String): void
1 Model
1 Model


aController: TextFieldController
aUser: User
nameInput: TextField
greetingOutput: TextField
perform(userInput: Input)
perform(userInput: Input)
notify(userInput: Input)
notify(userInput: Input)
updateName(userName: String)
notify(userInput: Input)
updateGreeting(greeting: String)

[4]

3.1.3 MVP

3.1.4 MVVM

[1]

3.2 Arbeitsabläufe für die Programmierung von Benutzeroberflächen

Programmierer programmiert, Designer entwürft.

3.3 Moderne Anforderungen an die Programmierung von Benutzeroberflächen

3.4 Anforderungen durch hardwarenahe Programmierung

Rechenleistung

→ Betrachtungen zum Prozessor

→ häufig 8- oder 16-bit, dieser Fall wird aber nicht betrachtet

→ Abgrenzung zu Microcontrollern an dieser Stelle

Memory

→ Eingeschränkt

Entwicklungskosten vs. Produktionskosten

→ Trade-Off zwischen Entwicklungskosten und Produktionskosten erklären

→ Erklären, dass über diese Ansätze lediglich Entwicklungskosten gesenkt werden sollen und dadurch minimal-geringe Mehrkosten in der Produktion in Kauf genommen werden → hängt von Anzahl an ausgerollten Geräten ab

Lebensdauer

→ ...

Ausfallsicherheit

→ ...

[Garbage Collection sorgt für unvorhergesehenes Zeitverhalten → nicht benutzen]

[Möglichst selten Speicher-Allokalisierung durchführen]

[Evtl. Ausflug in Fragmentierung von heaps geben → ganz zum schluss schreiben, evtl. weglassen]

[5, S. 5]

4 Bewertung von bisherigen Technologien zur Eignung für hardwarenahe Benutzeroberflächenerstellung

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

4.1 Bewertungskriterien

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

4.2 Anforderungen für die Programmierung von graphischen Benutzeroberflächen

→ Erweiterbarkeit mit neuen Elementtypen

→ Wiederverwendbarkeit von erstellten Views in anderen Views

→ Widerspruch zwischen Performance zur Laufzeit und Entwicklerkomfort → Lösen über interaktiven Modus mgl.?

→ Keinen Quellcode für View-Elemente schreiben (die keine weitere Funktionalität beinhalten)

→ Einbinden von bestehenden Objekten und Variablen möglich

→ Möglichst viele Programmierparadigmen unterstützen ↔ keine Framework-Strukturen vorgeben, um dynamische Programmierung zu ermöglichen

- Langlebigkeit des Quellcodes?

- Vendor Locking [6]

4.3 Anforderungen für hardwarenahe Programmierung

- Overhead durch dynamische Allokalisierung von Memory (besonders in Bezug auf Objektorientierung) → Objektstruktur möglichst flach halten

- Keine VM (daher kein java)

- Konflikt zwischen Objektorientierung und Graphikkartenbibliothek, welche als Statemachine arbeitet

- Kein Garbage-Collector

4.4 Ausgewählte Technologien

[Zu jeder Möglichkeit min. einen prominenten Vertreter auswählen, Technologien kurz vorstellen]

4.5 Vergleich

[Punkte vergeben, jede Punktevergabe kurz begründen]

[inklusive kurzes Fazit]

5 Auszeichnungssprachen

5.1 Definition und Funktion von Auszeichnungssprachen

Eine Auszeichnungssprache (engl. „Markup Language“) ist eine Computersprache, welche ein Dokument um syntaktische Elemente erweitert, um eine maschinelle Verarbeitung zu ermöglichen [7].

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

Markup Languages 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 [8]. Dabei hat jede Markup Language eigene Syntax-Elemente. Häufig kann zwischen einzelnen Syntax-Elementen unformatierter Fließtext stehen.

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

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

Diese Liste hat keinen Anspruch auf Vollständigkeit.

Weil Markup Languages für die Erstellung von Oberflächen eine starke Verbreitung gefunden haben, werden Sie in dieser Arbeit näher betrachtet.

5.2 Vorstellung ausgewählter Auszeichnungssprachen

5.2.1 XML

5.2.2 XAML

5.2.3 JSON

5.2.4 YAML

5.3 Problematik der impliziten Typisierung bei Auszeichnungssprachen

[Kurz auf die Problematik der Typisierung eingehen → Implizite Typvergabe bei embedded Programming häufig nicht gewollt]

6 Prototypische Entwicklung

6.1 Anforderungen an den Prototyp „XXX“

Der Prototyp soll grundsätzliche Anwendungsfälle für das Lösungskonzept abdecken. Dazu muss zunächst betrachtet werden, für welche Anwendungen eine hardwarenahe Oberflächengestaltung in der Praxis benötigt wird. Folgende Anwendungsfälle konnten ermittelt werden:

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

Smart-Home-Anwendungen

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

Für den Prototyp sind Smart-Home Anwendungen interessant, welche über eine graphische Oberfläche mit dem Endnutzer interagieren.

Anwendungen für Automaten jeglicher Art

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

Anwendungen für Bürogeräte

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

Mobile Anwendungen

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

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

6.1.1 Funktionale Anforderungen

- Implementation eines Taschenrechners?

- Login-Page?

- Kontaktliste?

201: Resource Provider

Es muss Funktionalität implementiert werden, welche

6.1.2 Qualitätsanforderungen

301:

6.1.3 Rahmenbedingungen

6.1.3.1 Technisch/ Technologische Rahmenbedingungen

401: Portabilität

Die Portabilität der zugrunde liegenden Programmiersprache soll gewährleistet bleiben.

402: Hardwarenahe Programmierung

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

6.1.3.2 Rechtliche Rahmenbedingungen

6.1.3.3 Organisatorische Rahmenbedingungen

6.2 Lösungskonzepte

Grober Softwareentwurf (Schritte zum kompilieren eines Projektes)

Vorstellung der Konzepte:

6.2.1 Syntax der Auszeichnungssprache

```
document := yaml_key['__generate__'] : object_type, (yaml_key :  element)*
object_type := \w[\w\d]*
element := (object | array | expression)
object := (yaml_key['__type__'] : object_type)?,  (yaml_key['__ref__'] :
object_ref_name)?,
                    (yaml_key : element)*
object_ref_name := \w[\w\d]*
array := yaml_array[array_object]
array_object := (yaml_key : element)*
```

expression := struct | string | constant_expression | reference
struct := (yaml_key : expression)* | yaml_array[expression] | expression
string := \w[\w\d]*
constant_expression := \d[\d\s\+\-*\/]*
reference := \w[\w\d]*

- Resource-Provider Konzept

- Objekterstellungs-Injektion

- Referenzierung von Objekten in der Markup Language von außerhalb

- Typüberladung

6.3 Implementation

6.3.1 Objektorientierung in C

6.3.2 Ausgewählte Basisklassen

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

6.4 Validierung

[Überprüfen, ob Rahmenbedingungen erfüllt worden sind]

7 Open-Source-Aktivitäten und Abschließende Bemerkungen des Projektes

8 Quellenverzeichnis

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

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

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

[4] „Organizing Presentation Logic“, *martinfowler.com*. [Online]. Verfügbar unter: https://martinfowler.com/eaaDev/OrganizingPresentations.html. [Zugegriffen: 14-Sep-2018].

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

[6] „Trends in der modernen Software-Entwicklung“. [Online]. Verfügbar unter: https://www.microsoft.com/germany/techwiese/know-how/trends-in-der-modernen-software-entwicklung.aspx. [Zugegriffen: 14-Sep-2018].

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

[8] „Auszeichnungssprache :: ML (markup language) :: ITWissen.info“. [Online]. Verfügbar unter: https://www.itwissen.info/Auszeichnungssprache-markup-language-ML.html. [Zugegriffen: 04-Okt-2018].

II Abbildungsverzeichnis

III Tabellenverzeichnis

IV Abkürzungsverzeichnis

GUI