

*Bachelorarbeit 2*

**Komponentenbasierte Softwarearchitektur und Softwareentwicklung:  
Ein Vergleich von Web-Components und Google Polymer**

Student Georg Eschbacher, 1110601005  
Betreuer Hubert Hölzl, MSc

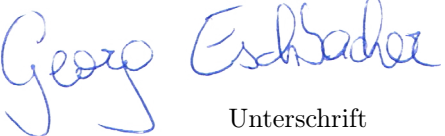
Salzburg, am 05. Mai 2014

## Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst, und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Weiters versichere ich hiermit, dass ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission weder im In- noch im Ausland vorgelegt und auch nicht veröffentlicht.

Salzburg, am 05. Mai 2014  
Datum

  
Unterschrift

## Kurzfassung

Vor- und Zuname:	Georg Eschbacher
Institution:	FH Salzburg
Studiengang:	Bachelor MultiMediaTechnology
Titel der Bachelorarbeit:	Komponentenbasierte Softwarearchitektur und Softwareentwicklung: Ein Vergleich von Web-Components und Google Polymer
Begutachter:	Hubert Hölzl, MSc

Im Rahmen dieser Arbeit werden die Gemeinsamkeiten beziehungsweise Unterschiede zwischen Web-Components und Konzepten der Softwarearchitektur sowie Softwareentwicklung analysiert. Mit Hilfe von Web-Components wird die Entwicklung von Komponenten im Web standardisiert. Bis dato wurden Komponenten immer mit Hilfe diverser Frameworks und Bibliotheken entwickelt. Komponenten, die mit zwei unterschiedlichen Frameworks beziehungsweise Bibliotheken entwickelt wurden, waren nicht interkompatibel. Um Komponenten verwenden zu können, die auf unterschiedlichen Frameworks und Bibliotheken basieren, mussten sämtliche Frameworks und Bibliotheken als Abhängigkeiten integriert werden, was wiederum zu einigen Interferenzen führte. Durch die korrekte Verwendung von Web-Components wird eine Komponente vollständig gekapselt und kann durch eine definierte Schnittstelle verwendet werden. Somit können einerseits keine Interferenzen zu anderen Komponenten entstehen und andererseits die Komponenten untereinander interoperieren. Inwiefern die Entwicklung von Komponenten mit Hilfe von Web-Components Konzepte der Softwarearchitektur zulassen, wird in dieser Arbeit geklärt. Web-Components werden speziell mit den Konzepten der serviceorientierten Softwarearchitektur, der komponentenbasierten Softwarearchitektur und der komponentenbasierten Softwareentwicklung verglichen. Dadurch, dass Web-Components ein sehr junger Standard ist (2013), wird diese Technologie seitens der Browser noch nicht vollständig unterstützt. Das Projekt „Polymer“ bietet Polyfills für sämtliche Funktionalitäten von Web-Components an. Somit werden in dieser Arbeit die bereits genannten Konzepte sowohl mit Web-Components, als auch Polymer verglichen. Auch wird Polymer als Ganzes mit Web-Components hinsichtlich sämtlicher Blickwinkel (bereitgestellte Funktionalität, Unterstützung, etc.) verglichen.

**Schlagwörter:** Softwarearchitektur, Softwarekomponenten, serviceorientierte Softwarearchitektur, komponentenbasierte Softwarearchitektur, komponentenbasierte Softwareentwicklung, Web-Components, Polymer

## Abstract

The goal of this thesis is to analyse similarities and distinctions between Web-Components and concepts of software-architecture and software-development. Web-Components standardize the development of components on the web. Yet, components were always developed with the help of various frameworks and libraries. Components based on two different frameworks or libraries were not compatible with each other. To use components, which are based on different frameworks or libraries, all used frameworks or libraries had to be integrated as dependency which in turn led to some interferences. The correct use of Web-Components guarantees that a developed component is fully encapsulated and can be used through a defined interface. Therefore there are no interferences with other components and so components are able to interoperate with each other. This thesis clarifies how far the development of components developed with Web-Components allow concepts of software architecture. Specifically Web-Components are compared with the concepts of service-oriented software-architecture, component-based software-architecture and component-based software development. Web-Components is a very young standard (2013) and thereby this technology on behalf of the browsers is not fully supported. The project „Polymer“ offers polyfills for all functionalities of Web-Components. With that said, the above mentioned concepts are compared with both Web-Components and polymer. Furthermore polymer as a whole is compared with Web-Components in respect of any perspective (supplied functionality, support, etc.).

**Keywords:** Software-Architecture, Software-Components, serviceoriented Software-Architecture, Component-based Software-Architecture, Component-based Software-Engineering, Web-Components, Polymer

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Relevanz . . . . .	1
1.2	Motivation . . . . .	2
1.3	Forschungsfrage . . . . .	3
1.4	Struktur der Arbeit . . . . .	3
<b>2</b>	<b>Softwarearchitektur und Softwarekomponenten</b>	<b>5</b>
2.1	Klassische Softwarekomponenten . . . . .	5
2.2	Arten von Softwarekomponenten . . . . .	7
2.3	Softwarearchitektur . . . . .	8
2.3.1	Serviceorientierte Softwarearchitektur . . . . .	11
2.3.2	Komponentenbasierte Softwarearchitektur und komponentenbasierte Softwareentwicklung . . . . .	12
2.3.3	Unterschied eines Dienstes und einer Komponente . . . . .	13
2.4	Konklusio . . . . .	13
<b>3</b>	<b>Web-Components</b>	<b>16</b>
3.1	Templates . . . . .	18
3.2	Decorators . . . . .	20
3.3	Custom Elements . . . . .	24
3.3.1	Registrierung neuer Elemente . . . . .	24
3.3.2	Erweiterung bereits vorhandener Elemente . . . . .	25
3.3.3	Erweiterung von Type Extension Custom Elements . . . . .	25
3.3.4	Erweiterung von Elementen . . . . .	26
3.3.5	Unresolved Elements . . . . .	26
3.3.6	Instanziierung von Custom Elements . . . . .	26
3.3.7	Instanziierung von Type-Extension-Elements . . . . .	27
3.3.8	Hinzufügen von Eigenschaften und Methoden zu einem Element . . . . .	27
3.3.9	Lebenszyklus-Callback Methoden . . . . .	27
3.4	Shadow DOM . . . . .	28
3.4.1	Trennung von Inhalt und Darstellung . . . . .	29
3.5	HTML Imports . . . . .	32
3.5.1	Ressourcen bündeln . . . . .	33
3.5.2	Load/Error Event-Handling . . . . .	33
3.5.3	Benutzung des Inhalts eines Imports . . . . .	34
3.5.4	JavaScript in einem HTML-Import . . . . .	35
3.5.5	HTML-Importe im Zusammenhang mit Templates . . . . .	35
3.5.6	HTML-Importe im Zusammenhang mit Custom-Elements . . . . .	36

3.5.7	Abhängigkeits-Management und Sub-Importe . . . . .	36
3.6	Browser Unterstützung . . . . .	36
3.7	Konklusio . . . . .	38
<b>4</b>	<b>Google Polymer</b>	<b>39</b>
4.1	Polymer Architektur . . . . .	39
4.2	Polyfilled Templates . . . . .	40
4.3	Polyfilled Custom-Elements . . . . .	41
4.4	Polyfilled Shadow-DOM . . . . .	42
4.5	Polyfilled HTML-Imports . . . . .	42
4.6	Besonderheiten von Polymer . . . . .	42
4.7	Unterstützung . . . . .	43
4.7.1	Unterstützung seitens der Browser . . . . .	43
4.7.2	Unterstützung seitens der Entwicklung . . . . .	44
4.8	Konklusio . . . . .	44
<b>5</b>	<b>Vergleich zwischen Web-Components und Google Polymer</b>	<b>46</b>
5.1	Vergleich hinsichtlich verschiedener Arten von Softwarekomponenten . . . . .	46
5.2	Vergleich hinsichtlich klassischer Softwarearchitektur . . . . .	47
5.3	Vergleich hinsichtlich serviceorientierter Softwarearchitektur . . . . .	48
5.4	Vergleich hinsichtlich komponentenbasierter Softwarearchitektur . . . . .	48
5.5	Vergleich hinsichtlich komponentenbasierter Softwareentwicklung . . . . .	48
5.6	Vergleich der zur Verfügung gestellten Funktionen . . . . .	49
5.7	Vergleich der Unterstützung . . . . .	52
<b>6</b>	<b>Web-Components Praxisbeispiel</b>	<b>53</b>
6.1	Programmierung von Web-Components nach dem W3C Standard . . . . .	53
6.2	Programmierung von Web-Components mit Hilfe von Google Polymer . . . . .	56
<b>7</b>	<b>Konklusio</b>	<b>61</b>
7.1	Ausblick von Web-Components . . . . .	62
7.2	Offene Fragen hinsichtlich der Entwicklung . . . . .	63

## Abkürzungsverzeichnis

<b>ECOOP</b>	European Conference on Object-Oriented Programming
<b>CRM-System</b>	Customer-Relationship-Management System
<b>ERP-System</b>	Enterprise-Resource-Planning System
<b>GUI</b>	Graphical User Interface
<b>IT</b>	Information technology
<b>SOA</b>	serviceorientierte Architektur
<b>XML</b>	Extensible Markup Language
<b>CORS</b>	Cross-Origin-Resource-Sharing
<b>SOP</b>	Same-Origin-Policy
<b>FOUC</b>	Flash of unstyled Content
<b>UI</b>	User-Interface

# 1 Einleitung

## 1.1 Relevanz

Zu Beginn muss geklärt werden, wie eine Softwarekomponente im Allgemeinen definiert ist. 1996 wurde die Softwarekomponente bei der European Conference on Object-Oriented Programming (ECOOP) folgendermaßen definiert (siehe Szyperski, Gruntz und Murer 2002, S. 35-47):

„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“

Um dies näher erläutern zu können, wird ein gängiger Tätigkeitsbereich eines Softwarearchitekten herangezogen: das Erstellen einer Liste von Komponenten, die in die gesamte Software-Architektur problemlos eingefügt werden kann. Diese Liste gibt dem Entwicklungsteam vor, welche Komponenten das Softwaresystem zum Schluss umfassen wird. In einfachen Worten zusammengefasst sind Softwarekomponenten die Teile, die eine Software definieren (siehe Szyperski, Gruntz und Murer 2002, S. 35-47).

Komponentenentwicklung bezeichnet die Herstellung von Komponenten, die eine spezielle Funktion in einer Softwarearchitektur übernehmen. Sämtliche Komponenten sollten immer für sich gekapselt und unabhängig voneinander sein. Dies garantiert die Wiederverwendbarkeit von bereits entwickelten Komponenten. Mehrere Komponenten werden mit Hilfe eines Verbindungsverfahrens zusammengeführt, beziehungsweise verwendet.

Softwarekomponenten haben ihren Ursprung im „Unterprogramm“. Ein „Unterprogramm“, oder auch „Subroutine“ genannt, ist der Teil einer Software, der von anderen Programmen beziehungsweise Programmteilen aufgerufen werden kann. Eine Subroutine gilt als Ursprung der ersten Einheit für die Softwarewiederverwendung (siehe Wheeler 1952).

Programmiererinnen und Programmierer entdeckten, dass sie sich auf die Funktionalität von zuvor geschriebenen Codesegmenten berufen können, ohne sich beispielsweise um ihre Implementierung kümmern zu müssen. Neben der Zeitersparnis, die dadurch entstand, erweiterte diese „Technik“ die Denkweisen: Der Fokus beim Entwickeln kann auf neue Algorithmen und komplexere Themen gelegt werden (siehe Szyperski, Gruntz und Murer 2002, S. 3-12).

In der gleichen Weise, wie die frühen Subroutinen die Programmierenden und Programmierer vom Nachdenken über spezifische Details befreit haben, verschiebt sich durch komponentenbasierte Softwareentwicklung der Schwerpunkt von direkter Programmiersoftware zu komponierten Softwaresystemen, also komponentenbasierten Systemen. Das heißt, dass sich der Schwerpunkt in Richtung Integration von Komponenten verlagert hat. Darauf basiert die Annahme, dass es genügend Gemeinsamkeiten in großen Softwaresystemen gibt, um die Entwicklung von wiederverwendbaren Komponenten zu rechtfertigen. Diese Komponenten können dann auf Grund ihrer Gemeinsamkeiten für weitere Systeme genutzt werden. Derzeit werden Komponenten gesucht, die eine große Sammlung von Funktionen bereitstellen. Für Unternehmen werden beispielsweise nicht nur simple Adressbücher, sondern ganze CRM-Systeme<sup>1</sup> gesucht, um die Daten beziehungsweise Kommunikationen seiner Kunden zu sichern. Darüber hinaus sollten diese Systeme auch flexibel sein, d.h. es sollte mit anderen Komponenten erweitert werden können (siehe Andresen 2003, S. 17-25).

Komponentenbasierte Entwicklung dient des Weiteren der Verwaltung von Komplexität innerhalb eines Systems. Sie versucht die Komplexität gering zu halten, indem die Programmiererin und der Programmierer sich vollständig auf die Implementierung der Komponenten fokussieren können. Das Verknüpfen beziehungsweise Kombinieren von Komponenten sollte demnach nicht

<sup>1</sup>Ein „Customer Relationship Management System“ bezeichnet ein System, das zur Kundenpflege beziehungsweise Kundenkommunikation dient.



mehr Aufgabe der Entwicklerin und des Entwicklers sein. Um diese Aufgabe zu lösen wird ein Architektur-Framework<sup>2</sup> eingesetzt, das Aktivitäten zur Identifikation, Spezifikation, Realisierung und Verteilung von Komponenten beschreibt. Vorteile durch dieses Programmierparadigma sind somit einerseits die Zeitersparnisse und andererseits die erhöhte Qualität der Komponenten (siehe Andresen 2003, S. 1-3). Standardmäßig werden in einem Softwaresystem Annahmen über den Kontext impliziert, in dem das System funktioniert. Die komponentenbasierte Softwareentwicklung verlangt,

„[...] , dass all diese Annahmen explizit definiert werden, damit das System in verschiedenen Kontexten wiederverwendet werden kann (siehe Andresen 2003).“

Durch diese explizite Definition von Annahmen gibt es verschiedene Anwendungsszenarien, die automatisch als Testszenarien der Komponenten dienen.

Im Zeitalter von Internet, Intranet und Extranet sind viele verschiedene Systeme und Komponenten miteinander zu verbinden. Aus den folgenden Gründen ist eine erweiterbare und flexible Architektur notwendig, die eine schnelle Reaktion auf neue Anforderungen ermöglicht: Web-basierte Lösungen sollen auf Informationen eines Servers zugreifen können, um z.B. mittels eines Extranets diversen Händlern transparente Einblicke in die Lagerbestände eines Unternehmens zu ermöglichen. ERP<sup>3</sup>- und CRM-Systeme sollen eingebunden werden, um die Betreuung und den Service für Kunden optimieren zu können (siehe Andresen 2003, S. 39-42).

Um diesen Anforderungen gerecht zu werden, gab es in den letzten Jahren eine unüberschaubare Anzahl an Frameworks beziehungsweise Bibliotheken, die die Entwicklung von Komponenten vereinfachten. Das Problem hierbei war, dass die Komponenten, die mit Hilfe der Frameworks beziehungsweise Bibliotheken entstanden sind, nicht mit anderen Frameworks beziehungsweise Bibliotheken verknüpft werden konnten. Eine Standardisierung, wie Komponenten im Web-Bereich aussehen müssen beziehungsweise wie die Schnittstellen definiert sein müssen, um Wiederverwendbarkeit garantieren zu können, gab es bis dato nicht. Eine neue Technologie, die zurzeit vom W3C standardisiert wird, gehört zu den interessantesten Webtechniken, da sie eine Standardisierung für Komponenten im Web-Bereich bieten. Diese Technologie versucht eine Vielzahl von Funktionen der populärsten JavaScript-Komponentenframeworks aufzunehmen und nativ in den Browser zu portieren. Dadurch wird es möglich, benutzerdefinierte Komponenten und Applikationen entwickeln zu können, ohne dabei zahlreiche andere Bibliotheken einbinden zu müssen. Weiters wird durch die Standardisierung die Wiederverwendbarkeit beziehungsweise Interoperabilität von Komponenten gewährleistet.

## 1.2 Motivation

Die persönliche Motivation des Autors zu diesem Thema entstand grundsätzlich in der Zeit eines Praktikums des Autors. Hier war es erforderlich, Frontend Komponenten für verschiedene Web-Applikationen zu entwickeln. Grundanforderung der Komponenten war, dass sie wiederverwendet werden können. Hauptproblem bei der Entwicklung der Komponenten war, dass die Web-Applikationen auf unterschiedlichen Frameworks basierten und sämtliche Komponenten immer an den „Standards“ der Frameworks angepasst werden mussten.

Web-Components bietet als erste Technologie einen allgemeinen Standard für Komponenten im Web-Bereich, wodurch sie für den Autor sehr interessant ist. Dadurch, dass Web-Components jedoch noch unter schlechter Browser-Unterstützung leiden, wird in dieser Arbeit auch der Polyfill Polymer von Google näher analysiert. Polymer versucht jegliche Funktionen von Web-Components

<sup>2</sup>Ein Komponenten-Architektur-Framework basiert auf einer Reihe von Plattform Entscheidungen, einer Reihe von Komponenten-Frameworks und ein Interoperabilitätsdesign der Komponenten-Frameworks (siehe Szyperski, Gruntz und Murer 2002, 419-422).

<sup>3</sup>Enterprise Resource Planning

im Browser zu emulieren, um sie somit für sämtliche „Evergreen“-Browser zur Verfügung stellen zu können.

Ein weiterer Punkt der persönlichen Motivation des Autors ist die grundsätzliche Änderung beziehungsweise Erweiterung einiger Konzepte der Web-Entwicklung durch Web-Components. Durch die vollständige Kapselung von Komponenten können keine Abhängigkeitsprobleme untereinander mehr entstehen. Beispielsweise hierfür sind unterschiedliche jQuery-Versionen in den Komponenten. Auch ist das Erstellen von Custom-Elements eine große Änderung. Das Markup von Komponenten kann zukünftig mit Hilfe eines benutzerdefinierten Tags gerendert werden. Dies hilft vor allem bei Komponenten, die sehr darstellungsabhängig sind.

Auf Grund des bereits zuvor genannten Hauptproblems bei der Entwicklung von Komponenten wird auch ein Praxisprojekt im Zuge der Bachelorarbeit erstellt. Hierbei wird der Fokus auf die Programmierung von zwei wiederverwertbaren Frontend-Oberflächen gelegt. Zum einen werden eine Diagramm-Komponente und zum anderen ein komplexes Menü entwickelt, das für spätere Projekte weiterverwendet werden kann. Die Hauptanforderung ist einerseits die Kompatibilität zu so vielen Browsern wie möglich zu gewährleisten und andererseits die Interoperabilität mit anderen Komponenten sicherzustellen. Des Weiteren soll durch die einmalige Programmierung dieser Komponente und deren darauffolgende Wiederverwendung der Wartungsaufwand möglichst gering gehalten werden. Somit wird in dieser Arbeit geklärt, inwiefern die Entwicklung von Web-Components ohne Unterstützungen wie beispielsweise das Polymer Projekt bereits möglich ist. Weiterhin soll geklärt werden, welche Aspekte der klassischen Softwarearchitektur bei der Entwicklung von Web-Components aufgegriffen werden und inwiefern es mit komponentenbasierter Softwareentwicklung beziehungsweise komponentenbasierter Softwarearchitektur vereinbar ist. Folgend werden diese Aspekte nicht nur an Hand der zurzeit standardisierten Technologie namens Web-Components analysiert, sondern auch an Hand des von Google zur Verfügung gestellten Polyfills namens Polymer.

### 1.3 Forschungsfrage

In dieser Arbeit soll folgende Forschungsfrage geklärt werden:

Welche Aspekte greifen Web Components aus der komponentenbasierten Softwarearchitektur auf und welche Vor- und Nachteile bietet dabei das Google Polymer Projekt.

### 1.4 Struktur der Arbeit

Die Arbeit beginnt mit einer allgemeinen Einführung zu den Begriffen „Softwarearchitektur“ und „Softwarekomponenten“ (siehe Kapitel 2.1 auf Seite 5). Nach dieser Erklärung werden die verschiedenen Arten von Softwarekomponenten aufgezeigt und näher erläutert (siehe Kapitel 2.2 auf Seite 7). Folglich wird der bereits erklärte Begriff der Softwarearchitektur (siehe Kapitel 2.3 auf Seite 8) hinsichtlich zwei Teilbereiche genauer analysiert: einerseits die serviceorientierte Softwarearchitektur (siehe Kapitel 2.3.1 auf Seite 11) und andererseits die komponentenbasierte Softwarearchitektur (siehe Kapitel 2.3.2 auf Seite 12). Als Abschluss des Kapitels wird der feine Unterschied zwischen einem Dienst und einer Komponente näher erläutert (siehe Kapitel 2.3.3 auf Seite 13).

Das darauffolgende Kapitel bietet zu Beginn einen Überblick über Web-Components (siehe Kapitel 3 auf Seite 16). Weiters wird die Spezifikation dieser Technologien an Hand einiger Beispiele näher erläutert (beginnend bei Kapitel 3.1 auf Seite 18 ff.). Auf Grund der mangelnden Browser-Unterstützung von Web-Components (siehe Kapitel 3.6 auf Seite 36) wird folglich Google-Polymer vorgestellt. Dieses Projekt soll sämtliche „Evergreen“-Browser unterstützen (siehe Kapitel 4.7 auf Seite 43) und somit die Entwicklung von Web-Components-Technologien bereits ermöglichen. Der Polyfill wird in Kapitel 4 auf Seite 39 genauer erklärt.

In Kapitel 5 auf Seite 46 wird ein detaillierter Vergleich von Web-Components und Polymer erstellt. Dieser Vergleich beinhaltet Aspekte beziehungsweise Relationen zur klassischen Softwareentwicklung (siehe Kapitel 5.2 auf Seite 47), serviceorientierten Softwarearchitektur (siehe Kapitel 5.3 auf Seite 48), komponentenbasierten Softwarearchitektur (siehe Kapitel 5.4 auf Seite 48) und komponentenbasierten Softwareentwicklung (siehe Kapitel 5.5 auf Seite 48).

Als Praxisprojekt der Arbeit wurden zwei Komponenten definiert: eine Diagramm-Komponente und eine Menü-Komponente. Beide Komponenten werden zum einen mit Hilfe von nativen Technologien und zum anderen mit Hilfe von Google-Polymer umgesetzt (siehe Kapitel 6.1 auf Seite 53 und Kapitel 6.2 auf Seite 56).

Die bis zu diesem Zeitpunkt definierten Begriffe und Erkenntnisse werden im Abschlusskapitel der Arbeit nochmals aufgegriffen und kurz beschrieben (siehe Kapitel 7 auf Seite 61). Abschließend wird der Ausblick sowie noch offene Fragen dieser Arbeit geklärt.

## 2 Softwarearchitektur und Softwarekomponenten

Das Kapitel 2 verhilft der Leserin und dem Leser den Begriff „Softwarearchitektur“ zu verstehen. Demnach sollen Vorteile von der Verwendung von Softwarearchitektur genannt werden. Auch ist zu erwähnen, dass Softwarearchitektur auch Nachteile bringt, die jedoch nicht in dieser Arbeit behandelt werden. Weiterhin werden wichtige Begriffe wie serviceorientierte Architektur beziehungsweise komponentenbasierte Softwarearchitektur verbunden mit komponentenbasierter Softwareentwicklung näher erläutert.

Zu Beginn wird geklärt, wie eine klassische Softwarekomponente definiert ist. Daraufhin werden diverse Sichtweisen einer Komponente an Hand von Abbildung 1 auf Seite 6 gezeigt. Folglich werden auf Basis der erklärten Definition mehrere Arten von Komponenten aufgelistet. Hierbei wird bereits der Begriff Softwarearchitektur genannt, der im darauffolgenden Kapitel beschrieben wird. Nach einer kurzen, allgemeinen Definition dieses Begriffs, erfolgt die Überleitung und Verbindung von Architektur und Software. Es ist zu erwähnen, dass in diesem Kapitel nur Architektur behandelt wird, die sich über die Erstellung, Auslieferung und den Betrieb von Software jeglicher Art erstreckt. Folglich gibt es Berührungspunkte zu anderen Architektur-Arten wie zum Beispiel der Daten- oder Sicherheitsarchitektur, die jedoch nicht in dieser Arbeit behandelt werden. Danach werden sowohl die serviceorientierte Architektur, als auch die komponentenbasierte Architektur in Verbindung mit komponentenbasierter Softwareentwicklung erklärt. Als Abschluss dieses Kapitels wird der Unterschied zwischen einem Service und einer Komponente an Hand von mehreren Punkten beschrieben.

### 2.1 Klassische Softwarekomponenten

„The characteristic properties of a component are that it: (siehe Szyperski, Gruntz und Murer 2002, S. 35-38)“

- is a unit of independent deployment
- is a unit of third-party composition
- has no (externally) observable state

Dies ist die Definition einer Komponente von Clemens Szyperski aus dem Buch „Component software: Beyond object-oriented programming“ (siehe Szyperski, Gruntz und Murer 2002, S. 35-38). Diese Definition bedarf hinsichtlich dieser Arbeit weiterer Erläuterung:

#### **A component is a unit of independent deployment**

Dieser Punkt der Definition besitzt eine softwaretechnische Implikation. Damit eine Komponente „independent deployable“ also unabhängig auslieferbar ist, muss sie auch so konzipiert beziehungsweise entwickelt werden. Sämtliche Funktionen der Komponente müssen vollständig unabhängig von der Verwendungsumgebung und von anderen Komponenten sein. Des Weiteren muss der Begriff „independent deployable“ als Ganzes betrachtet werden, denn dies bedeutet, dass eine Komponente nicht partiell, sondern nur als Ganzes ausgeliefert wird.

#### **A component is a unit of third-party composition**

Hier wird der Begriff „composable“ dahingehend verstanden, dass Komponenten zusammensetzbar sein sollen. In diesem Kontext bedeutet dies, dass eine Applikation aus mehreren Komponenten bestehen kann. Des Weiteren soll auch mit einer Komponente interagiert werden können, was einer klar definierten Schnittstelle bedarf. Nur mit Hilfe einer solchen Schnittstelle kann garantiert werden, dass die Komponente einerseits vollständig gekapselt von anderen Komponenten ist und andererseits mit der Umgebung interagieren kann. Dies erfordert demnach eine klare Spezifikation, was die Komponente einerseits erfordert und andererseits bereitstellt.

**A component has no (externally observable) state**

Eine Komponente sollte keinen externen „observable“ (feststellbaren) Zustand haben. Die Originalkomponente darf nicht von Kopien ihrer selbst unterschiedlich sein. Dies garantiert, dass sämtliche Kopien einer Komponente nach außen hin „gleich“ sind. Eine mögliche Ausnahme dieser Regel wäre jedoch ein Attribut der Komponente, das nicht zu ihrer Funktionalität beiträgt. Ein Beispiel dafür wäre ein Attribut für die Zwischenspeicherung von Daten. Dieses Attribut hat keinerlei Auswirkung auf die Funktionalität der Komponente selbst, jedoch Auswirkung auf die Performanz dieser Komponente. Attribute dieser Art sind erlaubt.

Wenn eine Komponente mit Hilfe dieser Definition umgesetzt wird, gilt sie als vollständig wiederverwertbar. Zugleich ergeben sich aus dieser Definition zwei Sichtweisen auf Komponenten. Zum einen die Sichtweise der Verwenderin beziehungsweise des Verwenders der Komponente (siehe Unterabbildung 1a auf Seite 6) und zum anderen die Sichtweise der Entwicklerin beziehungsweise des Entwicklers der Komponente (siehe Unterabbildung 1b auf Seite 6). Die Benutzerin und der Benutzer der Komponente können keine Aussagen darüber treffen, auf welcher Basis die verwendete Komponente entwickelt wurde. Folglich kann die Implementierung als „Black-Box“ für die Verwenderin beziehungsweise den Verwender gesehen werden. Die Entwicklerin beziehungsweise der Entwickler hingegen hat vollständiges Wissen über den Aufbau und das Verhalten der Komponente.

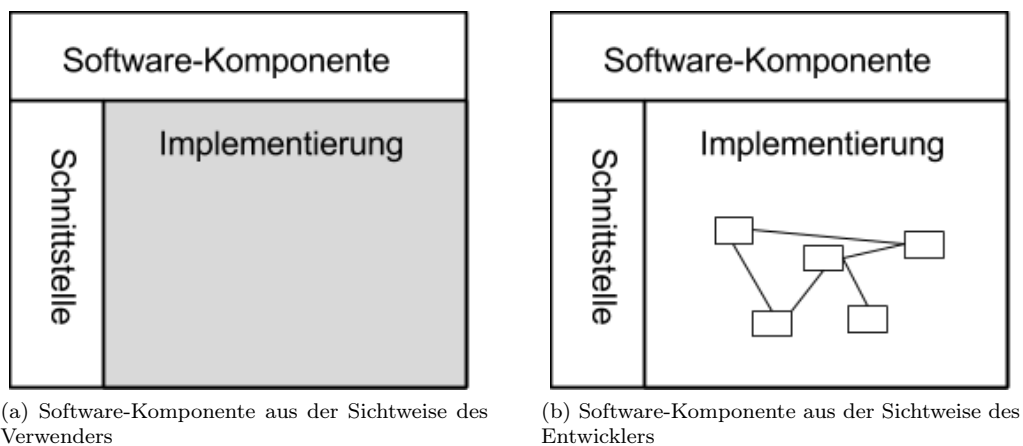


Abbildung 1: Software-Komponente aus unterschiedlichen Sichtweisen

In vielen aktuellen Ansätzen sind Komponenten eine große Einheit mit genau einer Instanz in einem System. Beispielsweise könnte ein Datenbankserver eine Komponente darstellen. Oftmals wird der Datenbankserver im Zusammenhang mit der Datenbank als Modul mit einem feststellbaren Zustand angesehen. Dahingehend ist der Datenbankserver ein Beispiel für eine Komponente und die Datenbank ein Beispiel für das Objekt, das von der Komponente verwaltet wird. Es ist wichtig zwischen dem Komponentenkonzept und dem Objektkonzept zu differenzieren, da das Komponentenkonzept keinen Gebrauch von Zuständen von Objekten fördert (siehe Szyperski, Gruntz und Murer 2002, S. 35-38).

Eine Softwarearchitektur ist die zentrale Grundlage einer skalierbaren Softwaretechnologie und ist für komponentenbasierte Systeme von größter Bedeutung. Nur da, wo eine Gesamtarchitektur mit Wartbarkeit definiert ist, finden Komponenten die Grundlage, die sie benötigen.

## 2.2 Arten von Softwarekomponenten

Verschiedene Arten von Komponenten können entsprechend ihren Aufgabenbereichen klassifiziert werden. Eine übersichtliche Art der Zuordnung von Aufgabenbereichen zu Komponenten kann auf der Basis der Trennung von Zuständigkeiten erfolgen, zum Beispiel auf der Basis einer Schichten-Architektur. Eine Schichten-Architektur dient der Trennung von Zuständigkeiten und einer losen Kopplung der Komponenten. Sie unterteilt ein Software-System in mehrere horizontale Schichten, wobei das Abstraktionsniveau der einzelnen Schichten von oben nach unten zunimmt. Eine jede Schicht bietet der unter ihr liegenden Schicht Schnittstellen an, über die diese auf sie zugreifen kann. Abbildung 2a auf Seite 7 veranschaulicht eine valide Form dieses Architekturparadigma. Abbildung 2b auf Seite 7 hingegen visualisiert eine nicht valide Schichtenarchitektur, denn Komponente 3 benutzt die Schnittstelle von Komponente 1. Somit verstößt dieses Beispiel gegen den Punkt, dass jede Schicht der unter ihr liegenden Schicht ihre Schnittstellen anbietet. (siehe Andresen 2003, S. 17-25).

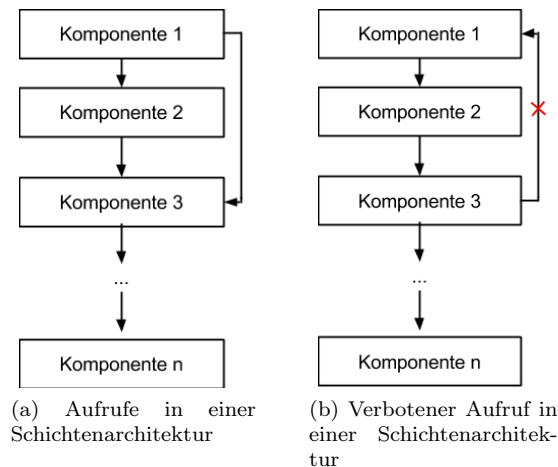


Abbildung 2: Beispiel einer Schichten-Architektur

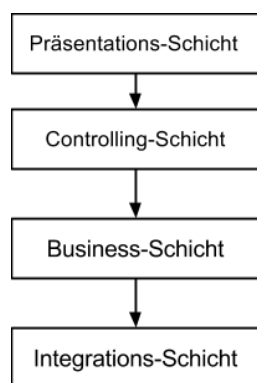


Abbildung 3: Zuteilung der Komponenten

Auf Grundlage der in Abbildung 2 auf Seite 7 veranschaulichten Architektur können Komponenten diversen Schichten zugeteilt werden, wie in Abbildung 3 auf Seite 7 zu sehen ist:

**1. Komponenten der Präsentations-Schicht**

Diese Komponenten stellen eine nach außen sichtbare Benutzerschnittstelle dar. Ein Beispiel dieser Schnittstelle ist eine GUI<sup>4</sup>-Komponente (Button, Menü, Slider, und vieles mehr...).

**2. Komponenten der Controlling-Schicht**

Diese verarbeiten komplexe Ablauflogik und dienen als Vermittler zwischen Komponenten der Business- und Präsentations-Schicht. Ein Beispiel hierfür wäre ein Workflow-Controller. Er koordiniert die Interaktion mit einer oder mehreren Businesskomponenten. Diese Komponente kann zum Beispiel den Ablauf eines Geschäftsprozesses umsetzen.

**3. Komponenten der Business-Schicht**

Sie bilden die Geschäftslogik im Sinne autonomer Businesskonzepte ab. Geschäftslogik in diesem Kontext bedeutet, dass die Komponente die Logik besitzt, welche die eigentliche Problemstellung löst. Sie dient beispielsweise der Persistierung von Daten beziehungsweise bildet Entitäten auf Komponenten ab (Firma, Produkt, Kunden).

**4. Komponenten der Integrations-Schicht**

Sie dienen der Anbindung an Alt-Systeme, Fremd-Systeme und Datenspeicher. Dies könnten beispielsweise Connector-Komponenten oder Datenzugriffs-Komponenten sein. Connector-Komponenten dienen der Integration eines Fremd-Systems und Datenzugriffs-Komponenten liefern den Datenzugriff für Komponenten der oben genannten Schichten. Bei Datenzugriffs-Komponenten werden zum Beispiel die Besonderheiten einer Datenbank berücksichtigt.

Komponente 1 in Abbildung 3 auf Seite 7 würde hinsichtlich der genannten Einteilung die Präsentations-Schicht darstellen und somit der Nutzerin und dem Nutzer zur Interaktion und bereitstellen von Informationen dienen. Weiters würde Komponente 2 der genannten Abbildung die Controlling-Schicht repräsentieren. Die Präsentations-Schicht kann die vom Nutzer eingegebenen Daten der Controlling-Schicht weiterleiten. Diese würde sich mit Hilfe der zugrunde liegenden Business-Schicht (Komponente 3) um die Generierung von Antworten bezüglich der Nutzereingaben kümmern. Komponente 3, in diesem Beispiel die Business-Schicht, würde Aktivitäten zur Abwicklung eines Geschäftsprozesses ausführen oder Komponenten der Integrations-Schicht aktivieren. Die Komponenten können indirekt vom Nutzer über die Controlling-Schicht, von Komponenten der Integrations-Schicht oder aber von anderen Systemen aktiviert werden. Komponente 4 beziehungsweise die Integrations-Schicht ist für die Anbindung bestehender Systeme, Datenbanken und für die Nutzung spezifischer Middleware zuständig (siehe Andresen 2003, S. 22-25).

Es ist zu erwähnen, dass es mehrere Kopplungen von Schichten-Architekturen gibt. Die in Abbildung 2a auf Seite 7 verwendete Kopplung wird auch als „lockere Schichten-Architektur“ bezeichnet. Zusätzlich zu dieser Kopplung gibt es noch eine „reine“, „stärker gelockerte“ und „vollständig gelockerte“ Schichten-Architektur. Diese Kopplungen werden nicht näher in dieser Arbeit erläutert.

## 2.3 Softwarearchitektur

Architektur ist nicht ausschließlich eine technologische Angelegenheit, sondern beinhaltet zahlreiche soziale und organisatorische Gesichtspunkte, die den Erfolg einer Architektur und damit eines gesamten Projekts erheblich beeinflussen können.

Dadurch, dass Architektur in verschiedenen Bereichen relevant ist und unterschiedliche Aspekte bei der Erstellung eines Systems umfasst, fällt eine allgemeingültige Definition schwer (siehe Vogel 2009, S. 8-11).

Zu Beginn wird die klassische Architektur als Ausgangspunkt verwendet. Eine mögliche Definition der klassischen Architektur bietet das „American Heritage Dictionary<sup>5</sup>“:

<sup>4</sup>Graphical User Interface

<sup>5</sup>Siehe American Heritage Online-Dictionary

Architecture is:

1. The art and science of designing and erecting buildings.
2. A style and method of design and construction
3. Orderly arrangement of parts

Diese Definition legt zu Grunde, dass Architektur sowohl eine Kunst als auch eine Wissenschaft ist, die sich sowohl mit dem Entwerfen, als auch mit dem Bauen von Bauwerken beschäftigt. Sie konzentriert sich nicht nur auf die Planung, sondern erstreckt sich bis hin zu der Realisierung eines Bauwerks. Ferner ist ein Schlüsselergebnis der Architekturtätigkeit das Arrangieren von Teilen des Bauwerks. Laut dieser Definition ist Architektur hiermit nicht nur die Struktur eines Bauwerks, sondern auch die Art und Weise, an etwas heranzugehen. Grundlegend basieren Architekturen auf gewissen Anforderungen und Werkzeugen, mit denen die Anforderungen umgesetzt werden. Ein Beispiel in diesem Kontext wäre der Wunsch nach einer Behausung, welcher unter Verwendung von vorhandenen Mitteln (Werkzeugen) realisiert werden kann.

Historisch basiert der eigentliche Entwurf auf dem Prinzip von Versuch und Irrtum. Erst durch die gewonnenen Architektur-Erfahrungen, welche mündlich oder schriftlich weitergegeben wurden, entwickelten sich Architekturstile. Folglich basiert Architektur auf Konzepten beziehungsweise Methoden, die sich in der Vergangenheit bewährt haben (siehe Vogel 2009, S. 41-68).

Zum Begriff „Architektur“ in der IT existieren im Gegensatz zur klassischen Architektur unzählige Definitionen<sup>6</sup>. Daran zeigt sich, dass es eine Herausforderung darstellt eine Definition zu finden, die allgemein anerkannt wird (siehe Shaw und Garlan 1996).

Softwarearchitektur erstreckt sich von der Analyse des Problembereichs eines Systems bis hin zu seiner Realisierung. Sie bewegt sich nicht auf der Abstraktionsebene fein-granularer Strukturen wie Klassen oder Algorithmen, sondern vielmehr auf der Ebene von Systemen, also grob-granularer Strukturen. Oftmals werden bei Projekten keine Aufwände im Zusammenhang mit Architektur bezahlt, was dazu führt, dass es im späteren Verlauf der Entwicklung zu vermeidbaren höheren finanziellen Kosten auf Grund eines erhöhten Wartungsaufwands kommen kann (siehe Vogel 2009, S. 8-11).

### Symptome mangelhafter Softwarearchitektur

Fatalerweise zeigen sich die Folgen einer mangelhaften Architektur in der IT nicht selten erst mit erheblicher Verzögerung. Dies bedeutet, dass ernste Probleme eventuell erst auftreten, wenn ein System zum ersten Mal produktiv eingesetzt wird. Eine Architektur, die ungeplant entstanden ist, sich also unbewusst im Laufe der Zeit entwickelt hat, führt zu erheblichen Problemen während der Erstellung, der Auslieferung und dem Betrieb eines Systems. Folgende Symptome können potentiell auf eine mangelhafte Architektur hindeuten (siehe Vogel 2009, S. 6-8):

- Fehlender Gesamtüberblick
- Komplexität ufer aus und ist nicht mehr beherrschbar
- Planbarkeit ist erschwert
- Risikofaktoren frühzeitig erkennen ist kaum möglich
- Wiederverwendung von Wissen und Systembausteinen ist erschwert
- Wartbarkeit ist erschwert
- Integration verläuft nicht reibungslos
- Performance ist unzureichend

<sup>6</sup>Das Software Engineering Institute (SEI) der Carnegie-Mellon Universität der Vereinigten Staaten von Amerika hat in der Fachliteratur über 50 verschiedene Definitionen für den Begriff „Softwarearchitektur“ ausgemacht (Software Architecture Definitions).



- Architektur-Dokumentation ist unzureichend
- Funktionalität beziehungsweise Quelltext ist redundant
- Systembausteine besitzen zahlreiche unnötige Abhängigkeiten untereinander
- Entwicklungszyklen sind sehr lang

### **Folgen mangelhafter Softwarearchitektur**

Die Folgen einer mangelhaften Softwarearchitektur wurden dem Buch „Software-Architektur: Grundlagen - Konzepte - Praxis“ von Oliver Vogel entnommen (siehe Vogel 2009, 6-8):

- Schnittstellen, die schwer zu verwenden beziehungsweise zu warten sind weil sie einen zu großen Umfang besitzen.
- Quelltext, der an zahlreichen Stellen im System angepasst werden muss, wenn Systembausteine, wie beispielsweise Datenbank oder Betriebssystem, geändert werden.
- Klassen, die sehr viele ganz unterschiedliche Verantwortlichkeiten abdecken und deshalb nur schwer wiederzuverwenden sind („Monster“-Klassen).
- Fachklassen, deren Implementierungsdetails im gesamten System bekannt sind.

### **Vorteile von Architektur**

Unabhängig davon, welche Art von System entwickelt wird, legt eine Architektur immer die Fundamente fest. Die Details, die für das zu entwickelnde System notwendig sind, werden nicht von einer Architektur festgelegt (siehe Buschmann 1996) nach (siehe Vogel 2009, S. 6-8). Folglich wird Architektur von den Fundamenten definiert, ohne auf deren interne Details einzugehen. Folgende Fragen im Hinblick auf ein System werden durch eine Architektur beantwortet:

- Auf welche Anforderungen sind Strukturierung und Entscheidungen zurückzuführen?
- Welches sind die wesentlichen logischen und physikalischen Systembausteine?
- Wie stehen die Systembausteine in Beziehung zueinander?
- Welche Verantwortlichkeiten haben die Systembausteine?
- Wie sind die Systembausteine gruppiert beziehungsweise geschichtet?
- Was sind die Kriterien, nach denen das System in Bausteine aufgeteilt wird?

Architektur beinhaltet demnach alle fundamentalen Festlegungen und Vereinbarungen, die zwar durch die fachlichen Anforderungen angestoßen worden sind, sie aber nicht direkt umsetzt.

Ein wichtiges Charakteristikum von Architektur ist die Handbarkeit und Überschaubarkeit von Komplexität. Sie zeigt nur die wesentlichen Aspekte eines Systems, ohne zu sehr in die Details zu gehen. So ermöglicht Architektur in relativ kurzer Zeit einen Überblick über ein System zu erlangen.

Die Festlegung, was genau die Fundamente und was die Details eines Systems sind, ist subjektiv beziehungsweise kontextabhängig. Gemeint sind in jedem Fall die Dinge, welche sich später nicht ohne weiteres ändern lassen. Dabei handelt es sich um Strukturen und Entscheidungen, welche für die Entwicklung eines Systems im weiteren Verlauf eine maßgebliche Rolle spielen (siehe Fowler 2005). Beispiele hierfür sind die Festlegung, wie Systembausteine ihre Daten untereinander austauschen oder die Auswahl der Komponentenplattform<sup>7</sup>. Derartige architekturrelevante Festlegungen wirken sich systemweit aus im Unterschied zu architekturirrelevanten Festlegungen (wie beispielsweise eine bestimmte Implementierung einer Funktion), die nur lokale Auswirkungen auf ein System haben (siehe Bredemeyer und Malan 2004).

<sup>7</sup>Beispiele für Komponentenplattformen sind JEE, .NET, Adobe AIR und viele mehr...

### 2.3.1 Serviceorientierte Softwarearchitektur

Mit Hilfe von SOAs (serviceorientierte Softwarearchitektur) können verteilte Systeme entwickelt werden. Hierbei können die Systemkomponenten eigenständige Dienste darstellen. Das System selbst kann auf geographisch verteilten Rechnern laufen. Die standardisierten XML-basierten Protokolle wurden dafür entwickelt, um die Dienstkommunikation und den Informationsaustausch unter diesen Diensten zu unterstützen. Folglich sind Dienste sowohl Plattform- als auch sprachunabhängig implementiert. Software-Systeme können durch Kompositionen lokaler und externer Dienste, welche nahtlos miteinander interagieren, aufgebaut werden (siehe Sommerville 2011, S. 509-514). Abbildung 4 auf Seite 11 visualisiert, wie Dienste verwendet werden. Eine Dienstanbieterin beziehungsweise Dienstanbieter veröffentlicht einen speziellen Dienst in einer Dienstverwaltung. Dieser Dienst kann wiederum von Dienstnehmerinnen und Dienstnehmern über die Dienstverwaltung gefunden werden. Wenn ein spezieller Dienst von einer Dienstnehmerin oder einem Dienstnehmer verwendet wird, entsteht eine Verbindung zwischen Dienstanbieterin oder Dienstanbieter und Dienstnehmerin oder Dienstnehmer.

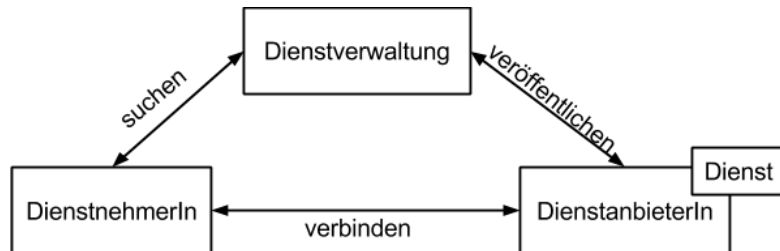


Abbildung 4: Serviceorientierte Architektur

Eine Vielzahl von Unternehmen, darunter auch Microsoft, haben solche Dienstverwaltungen anfangs des 21. Jahrhunderts eingerichtet. Sie waren lediglich für die Suche von externen Services errichtet worden. Durch die rasche Entwicklung der Suchmaschinenteknologie vereinfachten diese die Benutzung von Suchmaschinen stark. Sie galten rasch als die bevorzugte Variante um externe Services zu suchen und somit haben sich Dienstverwaltungen als redundant herausgestellt (siehe Sommerville 2011, S. 511).

Applikationen basierend auf Diensten zu entwickeln erlaubt es Unternehmen und anderen Organisationen zu kooperieren, indem sie die Dienste des jeweils anderen benutzen. Dies bedeutet, dass die Geschäftslogik, die mit Hilfe eines Dienstes erreichbar ist, auch anderen Unternehmen über diesen Dienst zur Verfügung steht. Systeme, die einen umfangreichen Informationsaustausch über deren geographische Grenzen haben, können mit Hilfe von Services automatisiert werden. Ein Beispiel in diesem Kontext wäre ein Lieferketten-System, das in gewissen Zeitabständen Artikel über ein anderes System kauft. Die Bestellung des Artikels wird über den speziellen Dienst des anderen Unternehmens abgewickelt (siehe Sommerville 2011, S. 512).

Grundsätzlich sind SOAs lose gekoppelt, wobei die Dienstbindungen sich während der Laufzeit noch ändern können. Dies bedeutet, dass eine andere, jedoch äquivalente Version des Dienstes zu unterschiedlichen Zeiten ausgeführt werden kann. Einige Systeme werden ausschließlich mit Web-Diensten gebaut, andere jedoch mischen Web-Dienste mit lokal entwickelten Komponenten (siehe Sommerville 2011, S. 512).

### 2.3.2 Komponentenbasierte Softwarearchitektur und komponentenbasierte Softwareentwicklung

Eine Komponente ist ein Softwareobjekt, welches gekapselt ist und mit anderen Komponenten durch eine klar definierte Schnittstelle interagieren kann (siehe Kapitel 2.1 auf Seite 5). Das Ziel der komponentenbasierten Softwareentwicklung ist die Steigerung der Produktivität, Qualität und „time-to market“<sup>8</sup>. Diese Steigerung soll mit dem Einsatz durch Standardkomponenten und Produktionsautomatisierungen erfolgen. Ein wichtiger Paradigmenwechsel bei dieser Entwicklungsmethode ist, dass Systeme auf Basis von Standardkomponenten aufbauen sollen, anstatt bereits funktionierende Komponenten neu zu entwickeln. Folglich ist die komponentenbasierte Softwareentwicklung eng mit dem Begriff der komponentenbasierten Softwarearchitektur verbunden. Die Architektur dieser Entwicklungsmethode bewegt sich nicht auf der Abstraktionsebene fein-granularer Strukturen wie Klassen oder Algorithmen, sondern vielmehr auf der Ebene von Systemen, also grob-granularer Strukturen (siehe Sommerville 2011, S. 452-468).

Folglich muss die Definition der komponentenbasierten Softwarearchitektur hinsichtlich des bereits definierten Begriffs der Softwarearchitektur (siehe Kapitel 2.3 auf Seite 8) wie folgt erweitert werden. Softwarearchitektur erstreckt sich von der Analyse des Problembereichs eines Systems bis hin zu seiner Realisierung. Sie ist sowohl die Identifikation, als auch die Dokumentation der einerseits statischen Struktur und andererseits der dynamischen Interaktion eines Software Systems, welches sich aus Komponenten und Systemen zusammensetzt. Dabei werden sowohl die Eigenschaften der Komponenten und Systeme als auch ihre Abhängigkeiten und Kommunikationsarten mittels spezifischer Sichten beschrieben und modelliert (siehe Andresen 2003, S. 43).

Weiters erstreckt sich die komponentenbasierte Softwareentwicklung von der Definition, bis hin zur Implementierung, sowie Integrierung beziehungsweise Zusammenstellung von lose gekoppelten, unabhängigen Komponenten in Systemen. Die Grundlagen einer solchen Entwicklungstechnik sind folgende (siehe Sommerville 2011, S. 452-468):

- Unabhängige Komponenten, die nur über klar definierte Schnittstellen erreichbar sind. Es muss eine klare Abgrenzung zwischen der Schnittstelle und der eigentlichen Implementierung der Komponente geben (siehe Abbildung 1a auf Seite 6).
- Komponentenstandards, die die Integration von Komponenten erleichtern. Diese Standards werden an Hand eines Komponentenmodells<sup>9</sup> dargestellt. Mit Hilfe dieses Modells werden Standards wie beispielsweise die Kommunikation zwischen Komponenten, oder die Struktur der Schnittstelle festgelegt.
- Middleware, die Softwareunterstützung für Komponentenintegration bereitstellt. Middleware für Komponentenunterstützung könnte beispielsweise Ressourcenallokation, Transaktionsmanagement, Sicherheit oder Parallelität sein.
- Ein Entwicklungsprozess, der komponentenbasierte Softwareentwicklung mit komponentenbasierter Softwarearchitektur verbindet. Die Architektur benötigt einen Prozess, der es zulässt, dass sich die Anforderungen an das System entwickeln können, abhängig von der Funktionalität der zur Verfügung stehenden Komponenten.

An Hand der genannten Grundlagen der komponentenbasierten Softwareentwicklung werden die Eckpfeiler der komponentenbasierten Architektur aufgebaut (siehe Szyperski, Gruntz und Murer 2002, 35-47).

- Interaktionen zwischen Komponenten und deren Umfeld sind geregelt

<sup>8</sup>Unter „time-to market“ wird die Dauer von der Produktentwicklung bis zur Platzierung des Produkts am Markt verstanden

<sup>9</sup>Beispiele für Komponentenmodelle sind Enterprise Java Beans, Cross Platform Component Object Model, Distributed Component Object Model, und viele mehr.

- Die Rollen von Komponenten sind definiert
- Schnittstellen von Komponenten sind standardisiert
- Aspekte der Benutzeroberflächen für Endbenutzer und Assembler sind geregelt

Diese Eckpfeiler verdeutlichen, wie eng die komponentenbasierte Softwareentwicklung in Verbindung mit komponentenbasierter Softwarearchitektur steht.

### 2.3.3 Unterschied eines Dienstes und einer Komponente

Dienste und Komponenten haben offensichtlich viele Gemeinsamkeiten. Beide sind wiederverwendbare Elemente und es gibt nur wenige Unterschiede.

Dienste sind eine Entwicklung von Softwarekomponenten, bei denen das Komponentenmodell eine Reihe von Standards verbunden mit Webdiensten darstellt. Folglich können Dienste im Unterschied zu Komponenten wie folgt definiert werden: Ein Webdienst ist ein Dienst, der über standardisierte XML- und Internet-Protokolle erreicht werden kann. Ein wichtiger Unterschied zwischen einem Dienst und einer Komponente ist, dass ein Dienst möglichst unabhängig und lose gekoppelt ist, während eine Komponente explizite Abhängigkeiten zu ihrem Kontext hat. Das bedeutet, dass Webdienste immer in der gleichen Weise arbeiten sollen, unabhängig von ihrer Einsatzumgebung. Die Schnittstelle eines Dienstes ist eine „Offers“-Schnittstelle, die den Zugriff auf die Dienstfunktionalität ermöglicht. Dienste streben einen unabhängigen Einsatz in unterschiedlichen Kontexten an. Daher haben sie nicht eine „Requires“-Schnittstelle, wie sie Komponenten haben. Komponenten sind meist auf zumindest eine „Grundkomponente“, wie beispielsweise die des Core-Systems, angewiesen.

Ein Dienst wird wie folgt benutzt: Eine Anwendung definiert, welchen Dienst sie benötigt. Dafür werden die Anforderungen an den Dienst in einer Nachricht zu dem Dienst gesendet. Der Empfangsdienst analysiert die Nachricht, führt den angeforderten Dienst durch und sendet, nach erfolgreichem Abschluss, eine Antwort als Nachricht zurück. Die Anwendung analysiert daraufhin die Antwort auf die gewünschten Informationen.

Ein Dienst muss somit nicht lokal zur Verfügung stehen und kann als externe Datenquelle angesehen werden. Eine Komponente hingegen muss lokal verfügbar sein. Komponenten können Dienste als Schnittstelle bereitstellen, die von außen erreichbar sind. Weiters können externe Dienste für gewisse Funktionalitäten auf lokale Komponenten des externen Dienstsystems zugreifen.

Dienste können zur Auslagerung gewisser Funktionalitäten verwendet werden. Durch die Auslagerung können Applikationen um ein vielfaches kleiner gemacht werden. Ein Beispiel dafür ist die Auslagerung des „Exception handling“<sup>10</sup>. Dies ist vor allem dann ein Vorteil, wenn die Applikation in Geräte eingebettet wird, bei denen es nicht möglich ist, Fehler aufzuzeichnen und nachzuvollziehen.

## 2.4 Konklusio

In diesem Kapitel wurde der Begriff der klassischen Softwarekomponente erläutert. Eine klassische Softwarekomponente wird von drei Punkte charakterisiert:

1. A component is a unit of independent deployment
2. A component is a unit of third-party composition
3. A component has no (externally) observable state

<sup>10</sup>„Exception-Handling“ ist die Fehlerbehandlung zur Laufzeit, welche verhindert, dass das Programm abstürzt und nach einem Fehler noch benutzbar ist.

Folglich wird auch der Begriff der Softwarearchitektur beschrieben, der jedoch in der IT keine eindeutige Definition besitzt. In dieser Arbeit wird der Begriff wie folgt definiert:

Softwarearchitektur erstreckt sich von der Analyse des Problembereichs eines Systems bis hin zu seiner Realisierung.

Mit Hilfe dieser Definition werden die Symptome und Folgen mangelhafter Softwarearchitektur, sowie die Vorteile von Architektur analysiert.

Darüber hinaus wird mit dem bereits definierten Begriff der Softwarearchitektur zwei spezielle Aspekte genauer beschrieben: die serviceorientierte Softwarearchitektur und die komponentenbasierte Softwarearchitektur mit der komponentenbasierten Softwareentwicklung.

Die Applikation, die mit serviceorientierter Architektur entwickelt wurde, kann auf Systemkomponenten basieren, die eigenständige Dienste darstellen. Dienste sind gegenüber von Komponenten grober granular und können Komponenten beinhalten. Auf Grund der Verwendung standardisierter Protokolle für die Kommunikation zwischen Diensten sind sie sowohl plattform- als auch sprachunabhängig implementiert. Services sind im Idealfall idempotent. Dies bedeutet, dass Services immer zu den gleichen Ergebnissen führen, unabhängig wie oft sie mit den gleichen Daten wiederholt werden.

Komponentenbasierte Softwareentwicklung ist der Prozess, der das Design und die Konstruktion von Systemen unter Verwendung von wiederverwendbaren Softwarekomponenten hervorhebt. Sie verschiebt den Schwerpunkt von der eigentlichen Entwicklung von Software zur Komposition von Software-Systemen. Diese Art der Softwareentwicklung erstreckt sich von der Standardisierung betrieblicher und technischer Aufgaben, über die allgemeine Entwicklung der Software beziehungsweise Komponente, bis hin zur Komposition mehrerer Komponenten.

Komponentenbasierte Softwarearchitektur ist ein Teilgebiet der komponentenbasierten Softwareentwicklung. Sie befindet sich im Bereich der Entwicklung und Komposition von Komponenten beziehungsweise Software. Die Architektur erstreckt sich von dem Bauplan eines Systems, über die Konstruktionsregeln für die Erstellung dieses Bauplans, bis hin zur Spezifikation des Aufbaus der Teilsysteme und Softwareschichten. Somit versucht sie die Beziehungen und Kommunikationen der Komponenten unter allen relevanten Blickwinkeln zu spezifizieren. Somit setzen komponentenbasierte Softwarearchitekturen das Konzept der „Separation of Concerns“ sehr stark um.

„Durch die Laufzeitumgebung werden technische und funktionale Belange getrennt und in verschiedene Komponenten gekapselt. Die Trennung dieser Belange ermöglicht, dass sie unabhängig voneinander weiterentwickelt und in verschiedenen Systemen wiederverwendet werden können (siehe Vogel 2009, S. 161-164).“

Da die serviceorientierte Architektur auf Diensten aufbaut und diese viele Ähnlichkeiten zu Komponenten aufweisen, veranschaulicht Tabelle 1 auf Seite 15 die Gemeinsamkeiten beziehungsweise Unterschiede dieser.

Die in diesem Kapitel und in der Konklusion nochmals zusammengefassten Begriffe dienen in Kapitel 7 auf Seite 61 als Grundbasis für die Beantwortung der Forschungsfrage.

Service	Komponente
Kompositionen sind nicht möglich	Kompositionen sind möglich
verfügen über definierte Schnittstellen	
sind im Idealfall idempotent	
vollständig unabhängig verwendbar	explizite Abhängigkeiten zu ihrem Kontext
Verfügen über klare Abgrenzungen zwischen Schnittstellen und Implementierungen	
Kann extern und lokal zur Verfügung gestellt werden	Muss lokal verfügbar sein

Tabelle 1: Unterschied zwischen Diensten und Komponenten

## 3 Web-Components

Unter „Web-Components“ wird ein Komponentenmodell verstanden, das 2013 in einem Working-Draft des W3C veröffentlicht wurde. Dieser Entwurf beinhaltet fünf Konzepte:

1. HTML-Templates
2. Decorators
3. Custom Elements
4. Shadow-DOM
5. HTML-Imports

Um Web-Components besser verstehen zu können, wird in diesem Kapitel zu Beginn eine kurze Übersicht über die Entstehung von diversen Web-Bibliotheken gezeigt. Sämtliche Bibliotheken dienen der benutzerdefinierten Erstellung von Komponenten, oder stellen selbst eine Reihe von benutzbaren Komponenten bereit. Eine nennbare Schwäche hierbei ist, dass sämtliche Komponenten nicht interoperabel sind, ohne die Basisbibliotheken zu inkludieren.

Die aufgelisteten Bibliotheken wurden an Hand der Popularität auf Github ausgewählt. Neben den genannten Bibliotheken gibt es jedoch eine Vielzahl anderer Projekte, die für einen kurzen und minimalen Überblick nicht genannt werden.

**2005:** Veröffentlichung von Dojo Toolkit<sup>11</sup> mit der innovativen Idee von Widgets. Mit wenig Code konnten Entwickler komplexe Elemente, wie beispielsweise einen Graph oder eine Dialog-Box in ihrer Website hinzufügen (siehe Forbes u. a. 2005).

**2006:** jQuery<sup>12</sup> stellt Entwicklern die Funktion zur Verfügung Plugins zu entwickeln, die später wiederverwendet werden können (siehe Osmani u. a. 2006).

**2008:** Veröffentlichung von jQuery-UI<sup>13</sup>, was vordefinierte Widgets und Effekte mit sich bringt (siehe Osmani u. a. 2008).

**2009:** Erstveröffentlichung von AngularJS<sup>14</sup>, ein Framework mit Direktiven (siehe Williams u. a. 2009).

**2011:** Erstveröffentlichung von React<sup>15</sup>. Diese Bibliothek gibt den Entwicklern die Fähigkeit, das User Interface ihrer Website zu bauen, ohne dabei auf andere Frameworks, die auf der Seite benutzt werden, achten zu müssen (siehe Chiu u. a. 2011).

**2013:** Veröffentlichung der Spezifikation von Web-Components (siehe Glazkov und Cooney 2013)

Wie bereits erwähnt, ist Web-Components ein Komponentenmodell, das aus fünf Konzepten besteht. Folglich werden die Konzepte kurz erläutert:

**HTML Templates** beinhalten Markups, die vorerst inaktiv ist, aber bei späterer Verwendung aktiviert werden können. Auch kann das definierte Markup in einem Template vervielfältigt werden und dient somit der Wiederverwendbarkeit (ausführliche Erklärung siehe Kapitel 3.1 auf Seite 18).

<sup>11</sup>Mehr Information zu Dojo Toolkit auf <http://dojotoolkit.org/>

<sup>12</sup>Mehr Information zu jQuery auf <http://jquery.com/>

<sup>13</sup>Mehr Information zu jQuery UI auf <http://jqueryui.com/>

<sup>14</sup>Mehr Information zu AngularJS auf <http://angularjs.org/>

<sup>15</sup>Mehr Information zu Facebook React auf <http://facebook.github.io/react/>

**Decorators** verwenden CSS-Selektoren basierend auf den Templates, um visuelle beziehungsweise verhaltensbezogene Änderungen am Dokument vorzunehmen (ausführliche Erklärung siehe Kapitel 3.2 auf Seite 20).

**Custom Elements** können neue Elemente definieren, oder bereits bestehende Elemente erweitern (ausführliche Erklärung siehe Kapitel 3.3 auf Seite 24).

**Shadow DOM** erlaubt es eine DOM-Unterstruktur vollständig zu kapseln. Sämtliche Elemente in dieser Unterstruktur sind von außen nicht erreichbar. Somit werden zuverlässigere Benutzerschnittstellen der Elemente garantiert, da keine Interferenzen in den Unterstrukturen entstehen können (ausführliche Erklärung siehe Kapitel 3.4 auf Seite 28).

**HTML Imports** definieren, wie Templates, Decorators und Custom Elements verpackt und als eine Ressource geladen werden können. (ausführliche Erklärung siehe Kapitel 3.5 auf Seite 32).

Beispielsweise könnte ein HTML-Element von einer einfachen Überschrift mit fest definiertem Aussehen, über einen Videoplayer, bis hin zu einer kompletten Applikation, alles darstellen. Vieles, was derzeit über JavaScript-Bibliotheken abgewickelt wird, könnte künftig in Form einzelner Webkomponenten umgesetzt werden. Dies verringert Abhängigkeiten und sorgt für mehr Flexibilität.

Obwohl „Web-Components“ für viele Entwicklerinnen und Entwickler noch kein Begriff ist, wird es bereits von diversen Browsern verwendet. Beispiele hierfür sind der „Datepicker“, oder das `<video>`-Element. Abbildung 5 auf Seite 17 zeigt die Datepicker-Komponente und Abbildung 6 auf Seite 18 zeigt den dazugehörigen Quellcode. An Hand dieses Codes ist zu sehen, dass sämtliche Kontrollbuttons des Datepickers „versteckt“ sind, sprich im Shadow-DOM liegen.

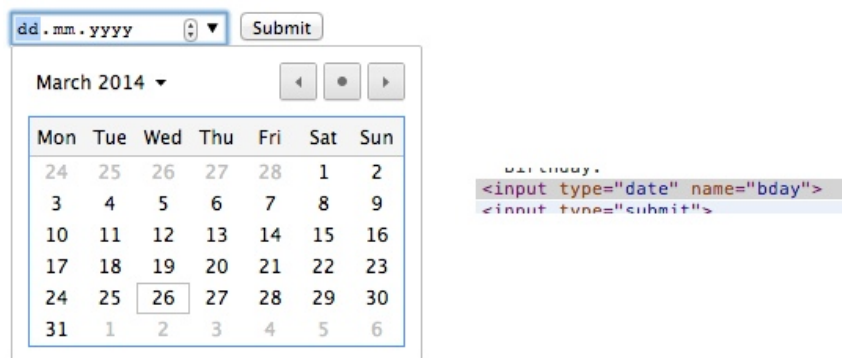


Abbildung 5: Beispiel von Web-Components im Browser an Hand eines Datepickers

### Warum Web-Components?

Javascript Widgets und Plugins sind fragmentiert, da diese auf unterschiedlichen Bibliotheken und Frameworks basieren und diese oftmals nicht interoperabel sind. Web-Components versuchen einen Standard in Widgets und Plugins zu bringen. Das Problem der nicht miteinander funktionierenden Plugins wird bei Web-Components mit Kapselung gelöst. Durch die Lösung dieses Problems ist die Wiederverwendbarkeit von Komponenten garantiert, da es sämtliche Interferenzen zwischen Plugins löst. Web-Components können des Weiteren viel mehr als nur UI-Komponenten sein.





Abbildung 6: Beispiel von Web-Components im Browser an Hand eines Datepickers

### Unterstützung von Web-Components

Zurzeit ist die Hauptproblematik von Web-Components die mangelhafte Browser-Unterstützung. Kein einziger Browser unterstützt diesen Standard zu 100% (nähere Erläuterung siehe Kapitel 3.6 auf Seite 36). Es gibt bereits mehrere Möglichkeiten beziehungsweise Polyfills<sup>16</sup>, um dennoch Web-Components nutzen zu können. Beispiele hierfür sind:

- Polyfill-Webcomponents<sup>17</sup>
- Polymer-Project<sup>18</sup>
- X-tags<sup>19</sup>

Dadurch, dass die Spezifikation von Web-Components Kapselung vorschreibt, sind sämtliche Polyfills interkompatibel.

Diese Arbeit beschränkt sich hauptsächlich auf die Entwicklung von Web-Components mit Hilfe des Standards beziehungsweise der Polyfill-Bibliothek Polymer.

Durch die Benutzung des Polyfills „Polymer“ funktionieren Web-Component in allen „Evergreen“-Browser<sup>20</sup> sowie dem Internet-Explorer 10 und neueren Versionen. Folglich funktionieren sie auch auf mobilen Endgeräten, wo iOS6+, Chrome Mobile, Firefox Mobile, oder Android 4.4+ vorhanden ist.

## 3.1 Templates

Das folgende Kapitel basiert ausschließlich auf der Spezifikation von Templates des W3C (siehe Dimitri Glazkov 2013) und auf dem Artikel „HTML’s new Template Tag“ (siehe Bidelman 2013c).

Laut W3C sind Templates

„a method for declaring inert DOM subtreess in HTML and manipulating them to instantiate document fragments with identical contents.“

<sup>16</sup>Ein Polyfill ist ein Browser-Fallback, um Funktionen, die in modernen Browsern verfügbar sind, auch in alten Browsern verfügbar zu machen.

<sup>17</sup>Mehr Information zu Polyfill-Webcomponents unter <http://github.com/timoxley/polyfill-webcomponents>

<sup>18</sup>Mehr Information zu Polymer unter <http://www.polymer-project.org/>

<sup>19</sup>Mehr Information zu X-tags unter <http://x-tags.org/>

<sup>20</sup>Ein „Evergreen“-Browser ist ein Web-Browser, der sich automatisch beim Start updatet.

Somit sind Templates eine Methode um inaktive DOM-Unterstruktur in HTML deklarieren und manipulieren zu können. Dadurch ist es möglich, sämtliche identische Dokumentfragmente mit identischem Inhalt instanziiieren zu können, ohne dabei identische Dokumentfragmente kopieren zu müssen.

In Web-Applikationen wird oft die gleiche Unterstruktur von Elementen wiederverwendet, mit dem passenden Inhalt gefüllt und zum Dokument hinzugefügt. Ein Beispiel in diesem Kontext wäre eine Liste von Artikel, die mit mehreren `<li>`-Tags in das Dokument eingefügt werden. Des Weiteren kann jeder `<li>`-Tag weitere Elemente, wie beispielsweise einen Link, ein Bild, einen Paragraphen, etc., enthalten. Derzeit bot HTML keine native Möglichkeit an, eine solche Aufgabenstellung zu lösen.

Folgend wird eine Liste von Autos mit Hilfe eines Templates erstellt:

```
1 <template id="carTemplate">
2   <li>
3     <span class="carBrand"></span>
4     <span class="carName"></span>
5   </li>
6 </template>
```

Code-Beispiel 1: Web-Components Template-Standard

Ein Template, wie das aus Code-Beispiel 1 auf Seite 19, kann sowohl im `<head>`- als auch im `<body>` definiert werden. Das Template, inklusive Subtree, ist inaktiv. Wenn sich ein `<img>`-Tag mit einer validen Quelle in diesem Template befinden würde, würde der Browser dieses Bild nicht laden. Darüber hinaus ist es nicht möglich ein Element des Templates via JavaScript zu selektieren, wie in Code-Beispiel 2 auf Seite 19 gezeigt wird.

```
1 document.querySelectorAll('.carBrand').length; // length ist 0
```

Code-Beispiel 2: Beispiel-Selektor eines Elements in einem Template, das nicht aktiven DOM ist

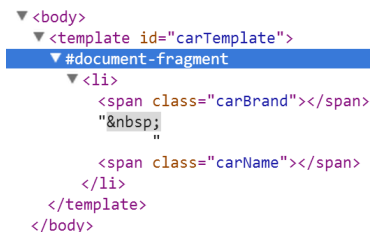


Abbildung 7: Visualisierung des DOM eines inaktiven Templates

In Abbildung 7 auf Seite 19 wird gezeigt, dass das Template ein Dokument-Fragment ist. Dies bedeutet, dass es ein eigenständiges Dokument ist und unabhängig vom ursprünglichen Dokument existiert. Folglich bedeutet dies, dass sämtliche `<script>`, `<form>`, `<img>`, -Tags etc. nicht verwendet werden können.

```
1 var template = document.getElementById('carTemplate');
2 template.content.querySelector('.carBrand').length; // length ist 1
3
4 var car = template.content.cloneNode(true);
5 car.querySelector('.carBrand').innerHTML = "Seat";
6 car.querySelector('.carName').innerHTML = "Ibiza";
7
8 document.getElementById("carList").appendChild(car);
```

Code-Beispiel 3: Verwendung des Templates 1 auf Seite 19

Code-Beispiel 3 auf Seite 19 basiert auf dem in Code-Beispiel 1 auf Seite 19 definierten Template. Zu Beginn wird sich in Zeile 1 des Code-Beispiels 3 eine Referenz zum bereits definierten Template in die Variable `template` gespeichert. Daraufhin wird der gesamte Knoten in Zeile 4 mit Hilfe einer `deep-copy` geklont und folglich mit Daten befüllt. Damit das mit Daten befüllte Listenelement auch sichtbar wird, wird es in Zeile 8 in das aktive DOM eingefügt.

## 3.2 Decorators

Das folgende Kapitel basiert ausschließlich auf der Einführung zu Web-Components aus dem W3C (siehe Glazkov und Cooney 2013) und auf dem Artikel „Decorators - NextGen Markup pt.2“ (siehe *Decorators - nextgen markup pt.2* 2013).

Decorators sind Elemente, die nach dem Decorator-Pattern benannt sind. Zurzeit gibt es keinerlei Unterstützung seitens der Browser zu diesem Konzept, somit wird in dieser Arbeit das vom W3C definierte Konzept nur theoretisch erläutert. Um jedoch dieses Konzept vollständig verstehen zu können, wird zuerst das genannte Pattern kurz beschrieben.

Grundsätzlich gehört das Decorator-Pattern zu den Struktur-Pattern der Softwareentwicklung. Das Pattern dient als Alternative zu Vererbung. Unter der Annahme, dass ein Objekt einen Rahmen erhalten soll, kann dies mit zwei Konzepten gelöst werden: Einerseits durch Vererbung oder andererseits durch Komposition.

Bei Vererbung würde der Rahmen von einer anderen Klasse geerbt werden. Somit besitzen jedoch alle Sub-Klassen den Rahmen. Diese Methode ist sehr inflexibel, da der Rahmen statisch durch Vererbung erstellt wird. Eine Endbenutzerin beziehungsweise ein Endbenutzer hätte keine Kontrolle darüber, wie eine Komponente mit einem Rahmen versehen werden kann.

Ein flexibler Ansatz wäre, dass die Komponente von einem Objekt umgeben wird, dass einen Rahmen beinhaltet. Das umgebene Objekt wird auch „Decorator“ genannt. Folgend wird dieser Ansatz an Hand von Abbildung 8 auf Seite 20 näher beschrieben.

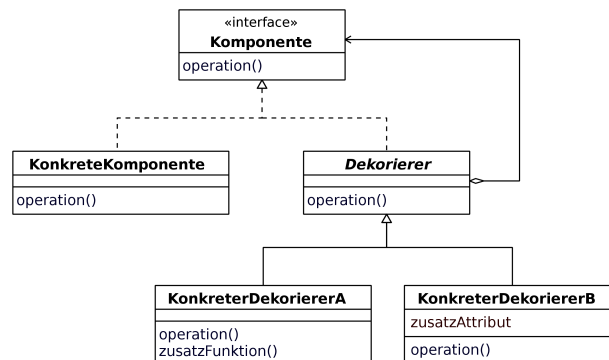


Abbildung 8: Decorator-Pattern UML (siehe Gamma 1995, S. 196-200)

Der Decorator wird vor das zu dekorierende Objekt gelegt. Die abstrakte Komponente definiert öffentliche Schnittstellen für das zu dekorierende Objekt. Die konkrete Komponente definiert ein Objekt, das dekoriert werden kann. Der abstrakte Decorator beinhaltet eine Referenz auf eine konkrete Komponente und bietet dieselbe Schnittstelle wie die abstrakte Komponente an. Der konkrete Decorator definiert und implementiert eine oder mehrere spezielle Decorators. Somit können Komponenten zur Laufzeit um zusätzliche Funktionalitäten und Darstellungen erweitert werden (siehe Gamma 1995, S. 196-200).

Um Decorators mit Hilfe des W3C-Konzepts näher erklären zu können, wird in diesem Kapitel folgendes Beispiel verwendet. Es gibt eine Liste von Autos, wobei jedes Auto eine Modellbezeichnung, eine Marke, ein Bild, sowie eine Kurzbeschreibung hat. In Bezug auf das Decorator-Pattern

(siehe Abbildung 8 auf Seite 20) wäre ein Auto eine konkrete Komponente. Das Markup eines Autos würde wie folgt aussehen:

```

1 <li class="car-item">
2   
3   <h3 class="car-model">Seat Ibiza</h3>
4   <p class="car-description">The SEAT Ibiza is a supermini car manufactured by the
      Spanish automaker SEAT. It is SEAT's best-selling car and perhaps the
      most popular model in the company's range.</p>
5 </li>

```

Code-Beispiel 4: Web-Components Decorators - Markup eines Autos

Folgend wird angenommen, dass die Funktionalität eines Autos erweitert wird: Ein Auto-Element bietet die Funktionalität, dass das Auto sichtbar, unsichtbar beziehungsweise geschlossen werden kann. Das bereits vorhandene Markup von Code-Beispiel 4 auf Seite 21 wird somit wie folgt erweitert:

```

1 <li class="car-item">
2   <section class="window-frame">
3     <header>
4       <a class="frame-toggle" href="#">Min/Max</a>
5       <a class="frame-close" href="#">Close</a>
6     </header>
7     
8     <h3 class="car-model">Seat Ibiza</h3>
9     <p class="car-description">The SEAT Ibiza is a supermini car manufactured by
      the Spanish automaker SEAT. It is SEAT's best-selling car and
      perhaps the most popular model in the company's range.</p>
10  </section>
11 </li>

```

Code-Beispiel 5: Web-Components Decorators - Markup eines Autos mit Rahmen

Code-Beispiel 5 auf Seite 21 würde somit das Markup eines Autos und zwei Buttons beinhalten: einen zum Umschalten zwischen sichtbar und unsichtbar und einen um das Bild komplett zu schließen. Wird das gezeigte Beispiel mit den bisherigen standardisierten Möglichkeiten umgesetzt, wird der Quellcode schnell relativ groß.

Decorators würden in diesem Beispiel bereits helfen. Es wäre möglich, spezielle Elemente im DOM mit mehr Markup, Gestaltung und zusätzlicher Funktionalität zu versehen. Essentiell hierbei ist, dass die zusätzliche Funktionalität nur für eine gewünschte Menge an Elementen erweitert werden kann. Hinsichtlich des bereits erklärten Decorator-Pattern wäre nachstehendes Code-Beispiel ein konkreter Decorator (siehe Abbildung 8 auf Seite 20). Die gewünschte Funktionserweiterung des Autos wird mit Hilfe eines Decorators umgesetzt. Das Markup würde somit wie folgt aussehen:

```

1 <decorator id="frame-decorator">
2   <template>
3     <section id="window-frame">
4       <header>
5         <a id="toggle" href="#">Min/Max</a>
6         <a id="close" href="#">Close</a>
7       </header>
8       <content></content>
9     </section>
10  </template>
11 </decorator>

```

Code-Beispiel 6: Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen)

Decorators werden grundsätzlich mit `<template>`-Elementen eingesetzt (mehr zu Templates in Kapitel 3.1 auf Seite 18). Des Weiteren wird in Zeile 8 des Code-Beispiels 6 ein `<content>`-Element verwendet. Dies ist zwingend notwendig, da in dieser Stelle der Inhalt des zu dekorierenden Elements eingefügt wird. In diesem `<content>`-Element wird somit die Referenz der Komponente, die

der Decorator beinhaltet, eingefügt. Auch ist zu erwähnen, dass in diesem Beispiel nur IDs verwendet werden. Dies dient der Visualisierung, dass IDs innerhalb eines `<decorator>`-Elements gekapselt sind. Sie werden nie im DOM erscheinen beziehungsweise verfügbar sein. `document.getElementById("window-frame")` wird keine Elemente zurückgeben, weder vor noch nach der Anwendung des `<decorator>`-Elements.

Weiterhin ist es möglich, sämtliche Elemente eines Decorators zu gestalten. In Code-Beispiel 7 auf Seite 22 werden die beiden Buttons mit `float: right;` gestaltet. Um die floats der Elemente wieder zu löschen, wird das `<header>`-Element mit der CSS-Klasse `clearfix` erweitert.

```

1 <decorator id="frame-decorator">
2   <template>
3     <section id="window-frame">
4       <style scoped>
5         #toggle float: right;
6         #close float: right;
7       </style>
8       <header class="clearfix">
9         <a id="toggle" href="#">Min/Max</a>
10        <a id="close" href="#">Close</a>
11      </header>
12      <content></content>
13    </section>
14  </template>
15 </decorator>

```

Code-Beispiel 7: Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen) inklusive Style

Es ist zu beachten, dass sämtliche Gestaltungen außerhalb des `<style scoped>`-Elements unter Verwendung der richtigen Klassennamen immer noch angewendet werden.

Um nun das bereits vorhandene Markup mit Funktionalität versehen zu können, muss zuerst noch ein wichtiger Punkt bezüglich Events erläutert werden. Bei Decorators gibt es keine „normalen“ Events. Das Hinzufügen beziehungsweise Entfernen eines Decorators würde das Event, wenn es bereits auf ein Element gebunden war, löschen. Decorators bieten nun die Möglichkeit einen Event-Controller zu erstellen, um mit Hilfe von diesen Events verwalten zu können.

```

1 <decorator id="frame-decorator">
2   <script>
3     this.listen({
4       selector: "#toggle", type: "click",
5       handler: function (event) {
6         // do the toggle button logic here
7       }
8     });
9     this.listen({
10      selector: "#close", type: "click",
11      handler: function (event) {
12        // do the close button logic here
13      }
14    });
15  </script>
16  <template>
17    <section id="window-frame">
18      <style scoped>
19        #toggle {float: right;}
20        #close {float: right;}
21      </style>
22      <header class="clearfix">
23        <a id="toggle" href="#">Min/Max</a>
24        <a id="close" href="#">Close</a>
25      </header>

```

```

26         <content></content>
27     </section>
28 </template>
29 </decorator>

```

Code-Beispiel 8: Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen) inklusive Style und Funktionalität

Der in Code-Beispiel 8 auf Seite 8 gezeigte Decorator kann somit verwendet werden, um die gewünschte Funktionalität (das Bild des Autos soll sichtbar, unsichtbar beziehungsweise geschlossen werden können) bereitstellen zu können, ohne dabei für jedes `<li>`-Element extra Markup hinzufügen zu müssen. Schlussendlich würde das Beispiel für ein Auto wie folgt aussehen:

```

1  <decorator id="frame-decorator">
2      <script>
3          this.listen({
4              selector:"#toggle", type:"click",
5              handler: function (event) {
6                  // do the toggle button logic here
7              }
8          });
9          this.listen({
10             selector:"#close", type:"click",
11             handler: function (event) {
12                 // do the close button logic here
13             }
14         });
15     </script>
16     <template>
17         <section id="window-frame">
18             <style scoped>
19                 #toggle {float: right;}
20                 #close {float: right;}
21             </style>
22             <header class="clearfix">
23                 <a id="toggle" href="#">Min/Max</a>
24                 <a id="close" href="#">Close</a>
25             </header>
26             <content></content>
27         </section>
28     </template>
29 </decorator>
30
31 <li class="car-item">
32     
33     <h3 class="car-model">Seat Ibiza</h3>
34     <p class="car-description">The SEAT Ibiza is a supermini car manufactured by the
        Spanish automaker SEAT. It is SEAT's best-selling car and perhaps the
        most popular model in the company's range.</p>
35 </li>

```

Code-Beispiel 9: Web-Components Decorators - Markup eines Autos mit Decorator

```

1  .car-item {
2      decorator: url(#frame-decorator);
3  }

```

Code-Beispiel 10: Web-Components Decorators - CSS für die Verwendung von Decorators

Unter der Verwendung des in Code-Beispiel 10 auf Seite 23 gezeigten CSS-Attributs wird der Decorator für das in Code-Beispiel 9 auf Seite 23 gezeigte Markup verwendet. Mit Hilfe des `decorator` : Attribut in CSS kann die URL beziehungsweise die Id des definierten Decorators angegeben werden. Das HTML-Element, das die CSS-Klasse mit dem gegebenen `decorator`-Attribut besitzt, wird mit dem im `<decorator id='frame-decorator'>` definierten Markup versehen. In Bezug zu dem

Decorator-Pattern wäre dies der Vorgang, wo der Decorator vor die Komponente, sprich in unserem Fall vor ein Auto, gesetzt wird. Weiters kann dieser Decorator vor spezielle Elemente gesetzt werden. Somit müssen nicht alle Autos die Funktionserweiterung erhalten.

### 3.3 Custom Elements

Das folgende Kapitel basiert ausschließlich auf der Spezifikation von Custom Elements des W3C (siehe Glazkov 2013a) und auf dem Artikel „Custom Elements“ (siehe Bidelman 2013a).

Custom Elements sind im Gegensatz zu bereits bestehenden DOM-Elementen eine neuer Typ von Elementen. Sie können von der Entwicklerin beziehungsweise dem Entwickler beliebig definiert werden und müssen nur wenige Vorschriften einhalten, damit die entwickelten Elemente als valide gelten. Im Gegensatz zu Decorators (siehe Kapitel 3.2 auf Seite 20), welche zustandslos und kurzlebig sind, können Custom Elements Funktionalität kapseln und eine Schnittstelle zur Verwendung bereitstellen.

Web Components würden ohne die Funktionalitäten von Custom-Elements nicht existieren. Sie stellen folgendes bereit:

1. Definition neuer HTML- beziehungsweise DOM-Elemente
2. Erstellung von Elementen, die die Funktionalität von bereits bestehenden Elementen erweitern
3. Logische Bündelung von benutzerdefinierten Funktionalitäten in nur einem Tag
4. Erweiterung von Schnittstellen von bereits vorhandenen DOM-Elementen

#### 3.3.1 Registrierung neuer Elemente

Custom Elements können mit Hilfe von der Methode `document.registerElement()` erstellt werden:

```
1 var myCar = document.registerElement('my-car');
2 document.body.appendChild(new myCar());
```

Code-Beispiel 11: Registrierung eines Custom-Elements

Das erste Argument der `document.registerElement()` Methode ist der Name des neuen Elements. Dieser Name muss einen Bindestrich enthalten. Diese Einschränkung erlaubt den Parser die Differenzierung zwischen benutzerdefinierten und regulären Elementen. Darüber hinaus gewährleistet dies auch eine Aufwärtskompatibilität, wenn neue Tags in HTML aufgenommen werden. Beispielsweise wären `<my-car>` oder `<my-element>` valide Elemente, wobei `<myCar>` oder `<myElement>` nicht valide wären. Die vorher genannte Methode hat ein zweites, optionales Argument, was ein Objekt wäre, dass das Prototyp-Objekt des Elements definiert (siehe Code-Beispiel 12 auf Seite 24). Dies wäre der Platz um dem Element öffentliche Methoden oder Eigenschaften zu geben. Standardmäßig erben sämtliche Custom-Elemente von `HTMLElement`. Somit wäre Code-Beispiel 11 auf Seite 24 das Gleiche wie Code-Beispiel 12 auf Seite 24.

```
1 var myCar = document.registerElement('my-car', {
2   prototype: Object.create(HTMLElement.prototype)
3 });
4 document.body.appendChild(new myCar());
```

Code-Beispiel 12: Registrierung eines Custom-Elements mit gegebenem Prototyp-Objekt

Ein Aufruf der `document.registerElement('my-car')`-Methode teilt dem Browser mit, dass ein Neues Element mit dem Namen „my-car“ registriert wurde. Folglich wird ein Konstruktor zurückgegeben, der zur Instanziierung neuer Elemente verwendet werden kann.

Standardmäßig wird der Konstruktor im globalen Window-Objekt abgelegt. Falls dies nicht erwünscht ist, kann ein Namespace dafür festgelegt werden. Code-Beispiel 13 auf Seite 25 veranschaulicht die Registrierung eines Elements in einem Namespace.

```
1 var myApp = {}
2 myApp.myCar = document.registerElement('my-car', {
3   prototype: Object.create(HTMLElement.prototype)
4 });
5 document.body.appendChild(new myApp.myCar());
```

Code-Beispiel 13: Registrierung eines Custom-Elements mit gegebenem Prototyp-Objekt und Namespace

### 3.3.2 Erweiterung bereits vorhandener Elemente

Durch Custom-Elements wird es möglich bereits vorhandene oder selbsterstellte Elemente zu erweitern. Um ein Element erweitern zu können, muss der `registerElement()`-Methode der Name und das Prototyp-Objekt des Elements, das erweitert werden soll, übergeben werden. Wenn beispielsweise `<element-a>` `<element-b>` erweitern möchte, so muss `<element-a>` bei der Registrierung das Prototyp-Objekt von `<element-b>` angegeben werden. In Code-Beispiel 14 auf Seite 25 wird die Funktionalität eines `<button>` erweitert.

```
1 var MegaButton = document.registerElement('mega-button', {
2   prototype: Object.create(HTMLButtonElement.prototype),
3   extends: 'button'
4 });
```

Code-Beispiel 14: Erweiterung von Elementen

Wenn ein Custom-Element von einem nativen Element erbt wird dies auch „type extension custom Element“ bezeichnet. Diese Elemente erben von einer speziellen Version des `HTMLElement`. Sozusagen „ist Element A ein Element B“. Der bereits definierte `MegaButton` (siehe Code-Beispiel 14 auf Seite 25) wird in Code-Beispiel 15 auf Seite 25 verwendet.

```
1 <button is="mega-button">
```

Code-Beispiel 15: Verwendung von Type-Extensions

### 3.3.3 Erweiterung von Type Extension Custom Elements

Um ein `<my-car-extended>`-Element zu erstellen, das `<my-car>` erbt, muss bei der Registrierung des erweiterten Elements das Prototyp-Objekt des gewünschten „Basis“-Elements angegeben werden. Code-Beispiel 16 auf Seite 25 verdeutlicht dies.

```
1 var MyCarProto = Object.create(HTMLElement.prototype);
2 var MyCarExtended = document.registerElement('my-car-extended', {
3   prototype: MyCarProto,
4   extends: 'my-car'
5 });
```

Code-Beispiel 16: Verwendung von Type-Extensions



### 3.3.4 Erweiterung von Elementen

„The HTMLUnknownElement interface must be used for HTML elements that are not defined by this specification (Bidelman 2013a).“

Dies bedeutet, dass sämtliche nicht valide deklarierten Elemente funktionieren und von `HTMLUnknownElement` erben. Valide deklarierte Elemente erben von `HTMLElement`. Wenn Browser die Methode `document.registerElement()` nicht bereitstellen, werden folglich auch Custom Elemente nicht unterstützt. Folglich werden sämtliche Custom-Elements, auch wenn sie valide deklariert sind, von `HTMLUnknownElement` erben, da Custom-Elements im Allgemeinen nicht unterstützt werden.

### 3.3.5 Unresolved Elements

Auch wenn Custom-elements in einem Skript registriert werden müssen, können sie bereits vor der Ausführung dieses Skripts deklarieren oder erstellt werden. Zum Beispiel ist es möglich `<my-car>` zu deklarieren, obwohl `document.registerElement('my-car')` später erst aufgerufen wird.

Bevor Elemente zu ihrer gewünschten Definition upgegradet werden, werden sie als „unresolved“-Elements bezeichnet. Dies sind HTML-Elemente, die einen validen benutzerdefinierten Namen haben, jedoch noch nicht registriert wurden. Tabelle 2 auf Seite 26 zeigt den Unterschied zwischen unresolved und unknown Elementen.

Name	Erbt von	Beispiel
Unresolved Element	HTMLElement	<code>&lt;my-car&gt;</code> , <code>&lt;my-wheel&gt;</code> , <code>&lt;my-element&gt;</code>
Unknown Element	HTMLUnknownElement	<code>&lt;myCar&gt;</code> , <code>&lt;my_wheel&gt;</code>

Tabelle 2: Unterschied zwischen ungelösten und unbekannten Elementen

### 3.3.6 Instanziierung von Custom Elements

Wie bei allen anderen Standardelementen können Custom-Elemente in HTML deklariert, oder im DOM mit Hilfe von JavaScript erstellt werden.

#### 1. HTML-Deklaration

```
1 <my-car></my-car>
```

Code-Beispiel 17: Instanziierung eines Custom-Elements mit Hilfe von HTML-Deklaration

#### 2. Erstellung im DOM mit Hilfe von JavaScript

```
1 var myCar = document.createElement('my-car');
2 myCar.addEventListener('click', function(e) {
3   alert('Thanks!');
4 });
```

Code-Beispiel 18: Instanziierung eines Custom-Elements mit Hilfe von JavaScript

#### 3. Erstellung mit Hilfe des `new`-Operator

```
1 var myCar = new MyCar();
2 document.body.appendChild(myCar);
```

Code-Beispiel 19: Instanziierung eines Custom-Elements mit Hilfe von JavaScript

### 3.3.7 Instanziierung von Type-Extension-Elements

Die Instanziierung von Type-Extension-Custom-Elements ist ähnlich zu der Instanziierung Custom-Elements.

#### 1. HTML-Deklaration

```
1 <button is="mega-button">
```

Code-Beispiel 20: Instanziierung eines type extension custom Elements mit Hilfe von HTML-Deklaration

#### 2. Erstellung im DOM mit Hilfe von JavaScript

```
1 var megaButton = document.createElement('button', 'mega-button');
2 // megaButton instanceof MegaButton === true
```

Code-Beispiel 21: Instanziierung eines type extension custom Elements mit Hilfe von JavaScript

#### 3. Erstellung mit Hilfe des new-Operator

```
1 var megaButton = new MegaButton();
2 document.body.appendChild(megaButton);
```

Code-Beispiel 22: Instanziierung eines type extension custom Elements mit Hilfe von JavaScript

### 3.3.8 Hinzufügen von Eigenschaften und Methoden zu einem Element

Custom-Elements werden erst durch maßgeschneiderte Funktionalität des Elements mächtig. Öffentliche Schnittstellen des Elements können mit Hilfe von Eigenschaften und Methoden erstellt werden. Folgend wird das Element `<x-foo>` registriert, welches eine read-only Eigenschaft namens `bar` hat und eine `foo()`-Methode bereitstellt.

```
1 var XFoo = document.registerElement('x-foo', {
2   prototype: Object.create(HTMLElement.prototype, {
3     bar: {
4       get: function() { return 5; }
5     },
6     foo: {
7       value: function() {
8         alert('foo() called');
9       }
10    }
11  })
12 });
```

Code-Beispiel 23: Beispiel eines Elements `<x-foo>` mit einer lesbaren Eigenschaft und einer öffentlichen Methode

Es gibt eine Vielzahl von verschiedenen Möglichkeiten, wie ein Prototyp-Objekt erstellt werden kann. In dieser Arbeit wird ausschließlich die zuvor gezeigte Methode verwendet.

### 3.3.9 Lebenszyklus-Callback Methoden

Elemente können spezielle Methoden definieren, die zu einer speziellen Zeit ihres Lebenszyklus automatisch aufgerufen werden. Diese Methoden werden Lebenszyklus-Callback Methoden genannt und jede Methode hat einen bestimmten Namen und Zweck, der in der folgenden Tabelle (Tabelle 3 auf Seite 28) genauer erläutert wird:

Callback-Name	Aufgerufen, wenn
<code>createdCallback</code>	eine Instanz des Elements erstellt wurde
<code>attachedCallback</code>	eine Instanz in das Dokument eingefügt wurde
<code>detachedCallback</code>	eine Instanz vom Dokument entfernt wurde
<code>attributeChangedCallback(attrName, oldVal, newCal)</code>	eine Eigenschaft hinzugefügt, upgedated, oder entfernt wurde

Tabelle 3: Lebenszyklus-Callback Methoden

Sämtliche Lebenszyklus-Callback Methoden sind optional. Sie können beispielsweise dazu verwendet werden, um im `createdCallback()` eine Verbindung zu einer Datenbank herzustellen. Wenn das Element entfernt wird, wird die dafür erstellte Verbindung im `detachedCallback()` geschlossen.

Folgend wird angenommen, dass im `createdCallback()` diverse Elemente angelegt werden. Diese Elemente können diverse Events bearbeiten. Weiters beinhaltet das Custom-Element ein Objekt `createdElements={}` wo sämtliche angelegte Elemente vom `createdCallback()` gespeichert werden. Wenn im `detachedCallback()` das `createdElements`-Objekt nicht ordnungsgemäß gelöscht wird beziehungsweise die Referenzen der erstellten Elemente nicht auf `null` gesetzt werden, kann dies zu einem Memory-Leak führen. Unter Nicht-Beachtung des `detachedCallback()` wird der Garbage-Collector des Browsers die angelegten Elemente im `createdCallback()` nicht entfernen können.

### 3.4 Shadow DOM

Das folgende Kapitel basiert ausschließlich auf der Spezifikation von Shadow-DOM des W3C (siehe Glazkov 2013b) und auf dem Artikel „Shadow DOM 101“ (siehe Dominic Cooney, Dimitri Glazkov 2013).

Mit Hilfe von Shadow-DOM können Elemente mit einer neuen Art von Knoten verbunden werden. Diese neue Art von Knoten wird auch „Shadow-Root“ genannt. Ein Element, das einer Shadow-Root zugeordnet ist, wird auch „Shadow-Host“ bezeichnet. Anstatt den Inhalt eines Shadow-Hosts zu rendern, wird immer der des Shadow-Roots gerendert.

```

1 <button>Hello, world!</button>
2 <script>
3   var host = document.querySelector('button');
4   var root = host.createShadowRoot();
5   root.textContent = 'Hello, shadow DOM!';
6 </script>
```

Code-Beispiel 24: Shadow-Root Beispiel eines Buttons

Code-Beispiel 24 rendert zuerst das in Abbildung 9a auf Seite 29 gezeigte Ergebnis. Danach wird mit Hilfe von JavaScript und Shadow-DOM das Element, wie in Abbildung 9b auf Seite 29 zu sehen ist, verändert.

Wenn das `<button>`-Element nach seinem Inhalt mittels der `textContent`-Eigenschaft abgefragt wird, wird das Resultat nicht `"Hello, shadow DOM!"` zurückgeben, sondern `"Hello, world!"`, da die DOM-Unterstruktur unter der Shadow-Root vollständig gekapselt ist.

Es ist zu erwähnen, dass dies ein sehr schlechtes Beispiel für Suchmaschinen, Browser-Extension, Screen-Readers etc. ist, da sämtlicher Inhalt des Shadow-DOMs nicht für diese erreichbar ist.

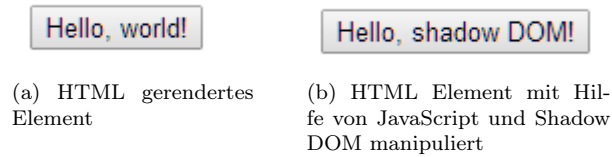


Abbildung 9: Beispiel einer Shadow-Root Node

Shadow-DOM ist nur für semantisch bedeutungsloses Markup, das benötigt wird um eine Webkomponente zu erstellen, gedacht.

### 3.4.1 Trennung von Inhalt und Darstellung

Code-Beispiel 25 auf Seite 29 wird als Ausgangsbasis dieses Beispiels genommen (siehe Dominic Cooney, Dimitri Glazkov 2013). Abbildung 10 auf Seite 30 zeigt diese Grundbasis, um mit darauffolgenden Schritten sämtlichen Inhalt von der Darstellung zu trennen. Dies garantiert, dass der tatsächliche Inhalt für Suchmaschinen, Screen-Readers, Browser-Extensions etc. erreichbar und die Darstellung für die Endbenutzerin beziehungsweise den Endbenutzer „unsichtbar“ ist.

```

1  <style>
2  .outer {
3    border: 2px solid brown;
4    border-radius: 1em;
5    background: red;
6    font-size: 20pt;
7    width: 12em;
8    height: 7em;
9    text-align: center;
10 }
11 .boilerplate {
12   color: white;
13   font-family: sans-serif;
14   padding: 0.5em;
15 }
16 .name {
17   color: black;
18   background: white;
19   font-family: "Marker Felt", cursive;
20   font-size: 45pt;
21   padding-top: 0.2em;
22 }
23 </style>
24 <div class="outer">
25   <div class="boilerplate">
26     Hi! My name is
27   </div>
28   <div class="name">
29     Bob
30   </div>
31 </div>

```

Code-Beispiel 25: Namensschild ohne Shadow-DOM - Ausgangsbasis um Inhalt von Darstellung zu trennen

Dadurch, dass dem DOM-Tree Kapselung fehlt, ist die gesamte Struktur des Namensschildes im Dokument sichtbar. Wenn beispielsweise externe Elemente auf der Webseite dieselben Klassennamen verwenden, würden diverse CSS-Klassen überschrieben werden.



Abbildung 10: Ausgangsbeispiel von der Trennung von Darstellung und Inhalt bei Shadow-DOM

### 1. Verstecken von Darstellungsdetails

Semantisch gibt es nur zwei wichtige Informationen bei diesem Beispiel:

- (a) Es ist ein Namensschild
- (b) Der Name ist „Bob“.

Daraus wird das Markup erstellt, das semantisch näher bei der gewünschten Information ist (siehe Code-Beispiel 26).

```
1 <div id="nameTag">Bob</div>
```

Code-Beispiel 26: Darstellung des Markups mit der gewünschten Information ohne Darstellung

Des Weiteren wird sämtlicher Code, der der Darstellung dient, in ein `<template>`-Element gepackt (siehe Code-Beispiel 27). Dies ist notwendig um sämtliche Darstellungsdetails von dem eigentlichen Inhalt trennen zu können.

```
1 <div id="nameTag">Bob</div>
2 <template id="nameTagTemplate">
3   <style>
4     .outer {
5       border: 2px solid brown;
6       border-radius: 1em;
7       background: red;
8       font-size: 20pt;
9       width: 12em;
10      height: 7em;
11      text-align: center;
12    }
13    .boilerplate {
14      color: white;
15      font-family: sans-serif;
16      padding: 0.5em;
17    }
18    .name {
19      color: black;
20      background: white;
21      font-family: "Marker Felt", cursive;
22      font-size: 45pt;
23      padding-top: 0.2em;
24    }
25  </style>
26  <div class="outer">
27    <div class="boilerplate">
28      Hi! My name is
```

```

29     </div>
30     <div class="name">
31         Bob
32     </div>
33 </div>
34 </template>

```

Code-Beispiel 27: Darstellung des Markups mit der gewünschten Information mit Hilfe von einem Template

Zu diesem Zeitpunkt wird „Bob“ das einzige sein, das gerendert wird. Das Template enthält sämtlichen Code der Darstellung und muss nun beispielsweise mit JavaScript hinzugefügt werden. In Code-Beispiel 28 auf Seite 31 wird zuerst eine Shadow-Root am Element `<div id=nameTag></div>` erstellt. Danach wird nach dem Template gesucht und der Inhalt dieses Templates an die Shadow-Root angefügt.

```

1 <script>
2   var shadow = document.querySelector('#nameTag').createShadowRoot();
3   var template = document.querySelector('#nameTagTemplate');
4   shadow.appendChild(template.content);
5 </script>

```

Code-Beispiel 28: Hinzufügen des Inhalts eines Templates in eine Shadow-Root

Da nun eine Shadow-Root mit Markup vorhanden ist, wird das Namensschild wieder korrekt angezeigt. Wenn das Element mit Hilfe der Entwickler-Tools im Browser inspiziert wird, wird nur die gewünschte Information ohne Darstellungselemente angezeigt. Dies zeigt, dass durch die Verwendung von Shadow-DOM sämtliche Darstellungsdetails im Shadow-DOM gekapselt wurden und von außen nicht erreichbar sind.

## 2. Trennung von Inhalt und Darstellung

Mit Hilfe von Code-Beispiel 27 und 28 werden sämtliche Darstellungsdetails versteckt, jedoch wurde der Inhalt noch nicht mit der Darstellung getrennt. Wenn beispielsweise der Name des Namensschildes ausgetauscht werden müsste, müsste dies an zwei Stellen gemacht werden: Einerseits an der Stelle im Template und andererseits an der Stelle des `<div id=nameTag></div>`-Elements.

Um den tatsächlichen Inhalt von sämtlichen Darstellungen zu trennen, muss eine Komposition von Elementen benutzt werden. Das Namensschild setzt sich einerseits aus dem roten Hintergrund mit den „Hi! My name is“-Text zusammen und andererseits aus dem Namen der Person.

Als Programmiererin beziehungsweise Programmierer einer Komponente kann entschieden werden, wie die Komposition des erstellten Elements funktionieren soll. Mit Hilfe des `<content>`-Elements können Kompositionen erstellt werden. Dieses Element erstellt einen „Insertion-Point“ in der Darstellung und sucht Inhalte aus dem Shadow-Host, die an dieser Stelle angezeigt werden sollten.

```

1 <div id="nameTag">Bob</div>
2 <template id="nameTagTemplate">
3   <style>
4     .outer {
5       border: 2px solid brown;
6       border-radius: 1em;
7       background: red;
8       font-size: 20pt;
9       width: 12em;
10      height: 7em;
11      text-align: center;
12    }
13    .boilerplate {
14      color: white;

```

```

15     font-family: sans-serif;
16     padding: 0.5em;
17 }
18 .name {
19     color: black;
20     background: white;
21     font-family: "Marker Felt", cursive;
22     font-size: 45pt;
23     padding-top: 0.2em;
24 }
25 </style>
26 <div class="outer">
27     <div class="boilerplate">
28         Hi! My name is
29     </div>
30     <div class="name">
31         <content></content>
32     </div>
33 </div>
34 </template>

```

Code-Beispiel 29: Erweiterung des Code-Beispiels 27 mit dem `<content>`-Element

In Code-Beispiel 29 auf Seite 31 wird das Namensschild mit dem vom Shadow-Host projizierten Inhalt in das `<content>`-Element gerendert. Dies vereinfacht die Struktur des Dokuments, da der Name nur noch an einer Stelle vorhanden ist. Müsste nun der Name aktualisiert werden, wäre das mit folgender Methode möglich: `document.querySelector('#nameTag').textContent = 'Shellie';`.

Das Namensschild wird automatisch nach Zuweisung eines neuen Namens aktualisiert, da der Inhalt vom Namensschild in das `<content>`-Element projiziert wird. Somit wurde die Trennung von Inhalt und Darstellung erreicht.

### 3.5 HTML Imports

Das folgende Kapitel basiert ausschließlich auf der Spezifikation von HTML Imports des W3C (siehe Glazkov und Morrita 2014) und auf dem Artikel „HTML Imports - #include for the Web“ (siehe Bidelman 2013b).

Es gibt eine Vielzahl von Möglichkeiten, wie diverse Ressourcen geladen werden können. Für JavaScript gibt es beispielsweise den `<script src='>`-Tag, für CSS gibt es den `<link rel='stylesheet'>`-Tag. Weiters gibt es eigene Tags für Bilder, Video, Audio, etc. Die meisten Web-Inhalte haben einen einfachen, deklarativen Weg, um sie zu laden. HTML hingegen besitzt keinen standardisierten Weg. Derzeit gibt es folgende Möglichkeiten HTML-Dokumente zu laden:

1. `<iframe>`
2. AJAX
3. `<script type='text/html'>`

Keine dieser Methoden ist ein standardisierter Weg, um externes HTML zu laden.

HTML-Imports bieten einen standardisierten Weg, wie ein externes HTML-Dokument in ein lokales HTML-Dokument geladen werden kann. Ein HTML-Import ist des Weiteren nicht auf Markup limitiert, sondern kann auch CSS, JavaScript, etc. beinhalten. Code-Beispiel 30 auf Seite 32 zeigt, wie ein lokales HTML-Dokument geladen wird.

```

1 <head>
2   <link rel="import" href="path/to/imports/car.html" />
3 </head>

```

Code-Beispiel 30: Laden eines lokalen HTML-Dokuments

Die URL eines Imports wird auch „Import-Stelle“ genannt. Um Inhalt von einer externen Domain laden zu können, muss die Import-Stelle CORS<sup>21</sup> aktiviert haben. CORS ist ein Mechanismus, um Browsern Cross-Origin-Requests zu ermöglichen. Zugriffe dieser Art sind normalerweise durch die SOP<sup>22</sup> untersagt. SOP wiederum untersagt JavaScript, ActionScript und Cascading-Style-Sheets, auf Objekte zuzugreifen, deren Speicherort nicht der Origin entspricht. CORS ist somit ein Kompromiss zugunsten größerer Flexibilität im Internet unter Berücksichtigung möglichst hoher Sicherheitsmaßnahmen. Code-Beispiel 31 auf Seite 33 zeigt, wie ein externes HTML-Dokument geladen wird.

```

1 <head>
2   <!-- http://example.com has CORS enabled. -->
3   <link rel="import" href="http://example.com/car.html">
4 </head>

```

Code-Beispiel 31: Laden eines externen HTML-Dokuments

Der Netzwerk-Stack des Browsers entfernt sämtliche Duplikate bezüglich Requests von derselben URL. Dies bedeutet, dass Importe, die dieselbe URL haben, nur einmal geladen werden.

### 3.5.1 Ressourcen bündeln

Importe stellen Konventionen bereit um HTML, CSS, JavaScript etc. bündeln zu können, damit sie als eine Datei lieferbar sind. Dies ist eine sehr wesentliche Eigenschaft von HTML-Importe. Durch die Funktionen, die HTML-Importe bereitstellen, ist es möglich, eine Web-Applikation in einzelne, logische Segmente aufzuteilen und der Endbenutzerin beziehungsweise dem Endbenutzer dennoch nur eine URL geben zu müssen.

Ein sehr gutes Beispiel für einen sinnvollen Einsatz von HTML-Importe wäre Bootstrap<sup>23</sup>. Bootstrap ist ein Frontend-Framework, um responsive mobile-first Projekte zu entwickeln. Es besteht aus individuellen CSS- und JavaScript-Dateien, sowie Fonts. Darüber hinaus benötigt es für die bereitgestellten Funktionen jQuery. Code-Beispiel 32 auf Seite 33 zeigt, wie Bootstrap auf mehrere Dokumente aufgeteilt und geladen werden kann. Um sämtliche Dateien von Bootstrap verwenden zu können muss somit nur noch eine Datei namens `bootstrap.html` geladen werden.

```

1 <!-- main.html -->
2 <head>
3   <link rel="import" href="bootstrap.html">
4 </head>
5
6 <!-- bootstrap.html -->
7 <link rel="stylesheet" href="bootstrap.css">
8 <link rel="stylesheet" href="fonts.css">
9 <script src="jquery.js"></script>
10 <script src="bootstrap.js"></script>
11 <script src="bootstrap-tooltip.js"></script>
12 <script src="bootstrap-dropdown.js"></script>
13 ...

```

Code-Beispiel 32: Inkludierung von Bootstrap mit Hilfe von einem HTML-Import

### 3.5.2 Load/Error Event-Handling

Das `<link>`-Element feuert ein „Load“-Event, wenn ein HTML-Import erfolgreich geladen wurde und ein „Error“-Event, wenn dies nicht der Fall war. Code-Beispiel 33 auf Seite 34 zeigt ein Beispiel von Error-Handling bei HTML-Importe.

<sup>21</sup>Cross-Origin Resource Sharing

<sup>22</sup>Same-Origin-Policy

<sup>23</sup>Mehr Information zu Bootstrap unter <http://getbootstrap.com/>



```

1 <head>
2   <script>
3     function handleLoad(e) {
4       console.log('Loaded import: ' + e.target.href);
5     }
6     function handleError(e) {
7       console.log('Error loading import: ' + e.target.href);
8     }
9   </script>
10
11   <link rel="import" href="car.html" onload="handleLoad(event)" onerror="
12     handleError(event)">
13 </head>

```

Code-Beispiel 33: Error-Handling bei HTML-Importe

Ein wichtiger Punkt bei dem in Code-Beispiel 33 gezeigten Beispiel ist, dass die Funktionen vor dem Import definiert wurden. Der Browser versucht einen HTML-Import zu laden, wenn er dem Tag begegnet. Wenn zu diesem Zeitpunkt die Funktionen noch nicht existieren, würden Fehler in der Konsole ausgegeben werden, da die Funktionsnamen noch `undefined` sind.

### 3.5.3 Benutzung des Inhalts eines Imports

Wenn ein HTML-Import benutzt wird, bedeutet dies nicht, dass an der Stelle, wo der Import-Befehl geschrieben wird, der Inhalt des Imports platziert wird. Vielmehr bedeutet es, dass der Browser das zu importierende Dokument analysiert und lädt, um es für weitere Verwendung bereit stellen zu können. Wenn der Inhalt eines Imports erreicht werden will, da er beispielsweise gewisse `<template>`-Elemente beinhaltet, muss dafür JavaScript verwendet werden. Code-Beispiel 34 auf Seite 34 holt sich den Inhalt eines HTML-Imports. Die importierte Datei (`warnings.html`) beinhaltet diverses gestaltete Markup, was in der Hauptseite (`main.html`) verwendet wird. Des Weiteren wird nur ein spezieller Teil des Imports verwendet, nämlich das `<div>`-Element mit der Klasse `warning`. Der restliche Inhalt des importierten HTML-Dokuments bleibt inaktiv und wird nicht vom Browser gerendert.

```

1 <!-- warnings.html -->
2 <div class="warning">
3   <style scoped>
4     h3 {
5       color: red;
6     }
7   </style>
8   <h3>Warning!</h3>
9   <p>This page is under construction</p>
10 </div>
11
12 <div class="outdated">
13   <h3>Heads up!</h3>
14   <p>This content may be out of date</p>
15 </div>
16
17
18 <!-- main.html -->
19 <head>
20   <link rel="import" href="warnings.html">
21 </head>
22 <body>
23   <script>
24     var link = document.querySelector('link[rel="import"]');
25     var content = link.import;
26
27     // Grab specific DOM from warning.html's document.
28     var el = content.querySelector('.warning');
29

```

```

30     document.body.appendChild(el.cloneNode(true));
31   </script>
32 </body>

```

Code-Beispiel 34: Klonen des Inhalts eines HTML-Imports

### 3.5.4 JavaScript in einem HTML-Import

Importe befinden sich nicht im Hauptdokument. Im zu importierenden Dokument kann das DOM vom Hauptdokument und sein eigenes DOM erreicht werden. Somit ist es möglich im importierenden Dokument das DOM des Hauptdokuments zu erweitern und zu manipulieren. Code-Beispiel 35 auf Seite 35 zeigt, wie das zu importierende Dokument eines seiner Stylesheets selbst im Hauptdokument hinzufügt. Wichtig hierbei ist, dass das zu importierende Dokument einerseits eine Referenz zum eigenen Dokument und andererseits eine Referenz zum Hauptdokument beinhaltet.

```

1 <link rel="stylesheet" href="http://www.example.com/styles.css">
2 <link rel="stylesheet" href="http://www.example.com/styles2.css">
3 <script>
4   // importDoc references this import's document
5   var importDoc = document.currentScript.ownerDocument;
6
7   // mainDoc references the main document (the page that's importing us)
8   var mainDoc = document;
9
10  // Grab the first stylesheet from this import, clone it,
11  // and append it to the importing document.
12  var styles = importDoc.querySelector('link[rel="stylesheet"]');
13  mainDoc.head.appendChild(styles.cloneNode(true));
14 </script>

```

Code-Beispiel 35: JavaScript im HTML-Import, um Inhalt automatisch im Hauptdokument hinzuzufügen

Ein Skript in einem Import kann entweder Code direkt ausführen, oder Funktionen bereitstellen, die dem importierenden Dokument zur Verfügung stehen. Grundregeln für JavaScript in einem HTML-Import sind folgende:

- Skripte im Import werden im Kontext des importierenden Dokuments aufgerufen. Das bedeutet, dass `window.document` im Import-Dokument eine Referenz zum Dokument ist, dass die Datei importiert. Dies hat zur Folge, dass sämtliche Funktionen, die im Import-Dokument definiert werden zum `window`-Objekt hinzugefügt werden. Folglich ist es nicht erforderlich `<script>`-Blöcke im Hauptdokument hinzuzufügen, da sie ausgeführt werden. Wenn Dateien von Dritten importiert werden könnte hier ein Sicherheitsrisiko vorhanden sein.
- Importe blocken den Browser nicht beim Parsen des Hauptdokuments. Dennoch werden Skripte in den Importen der Reihe nach verarbeitet. Ein Vorteil, der entsteht, wenn HTML-Importe im `<head>` definiert werden, ist, dass der Browser sofort beginnt die Inhalte der Importe zu parsen. Skripte im Hauptdokument hingegen blockieren den Browser beim parsen. Demnach sollten sämtliche Skripte im Hauptdokument am Ende der Datei geladen beziehungsweise durchgeführt werden.

### 3.5.5 HTML-Importe im Zusammenhang mit Templates

Ein sehr großer Vorteil, wenn HTML-Importe und Templates zusammen verwendet werden ist, dass Skripte innerhalb eines Templates nicht beim Laden des zu importierenden Dokuments ausgeführt werden, sondern erst, wenn das Template aktiv wird, sprich dem DOM des Hauptdokuments hinzugefügt wird.

### 3.5.6 HTML-Importe im Zusammenhang mit Custom-Elements

Wenn diese beiden Technologien vereint werden, muss sich die Benutzerin und der Benutzer, der beispielsweise ein externes Custom-Element mit Hilfe eines HTML-Imports lädt, nicht um die Registrierung des Elements kümmern, da es bereits im zu importierenden Dokument registriert werden kann. Folglich muss im zu importierenden Dokument die Registrierung des Custom-Elements am Hauptdokument erfolgen. Kapitel 3.3.1 auf Seite 24 beschreibt, wie Custom-Elements registriert werden können.

### 3.5.7 Abhängigkeits-Management und Sub-Importe

Sub-Importe sind vor allem dann von Vorteil, wenn eine Komponente wiederverwendbar oder erweitert werden soll. Beispielsweise kann jQuery als eine Komponente angesehen und als HTML-Import definiert werden. Wenn mehrere Custom-Elements mit Hilfe von jQuery entwickelt und externen Personen zur Verfügung gestellt werden, werden die Abhängigkeiten für sämtliche Elemente automatisch geladen. Unter der Annahme, dass drei Custom-Elemente geladen werden, wobei jedes einzelne Element an sich jQuery als Abhängigkeit mittels HTML-Import geladen hat, wird jQuery trotzdem nur ein einziges mal im Hauptdokument geladen.

Es ist auch möglich, die entwickelten Elemente mit Hilfe von `bower`<sup>24</sup> zur Verfügung zu stellen. Wenn Elemente mittels `bower` installiert werden, werden sämtliche definierte Abhängigkeiten auch installiert.

## 3.6 Browser Unterstützung

Die nachstehende Tabelle 4 zeigt die Unterstützung der Browser bezüglich Web-Components. Grün bedeutet, dass die Technologie in dem jeweiligen Browser als stabil gewertet wird und verwendet werden kann. Gelb bedeutet, dass die Technologie vom jeweiligen Browser in Arbeit ist, noch Bugs auftreten, oder mittels eines Flags erreichbar ist. Rot bedeutet, dass es keine Information bezüglich der Technologie in dem jeweiligen Browser gibt und somit auch nicht unterstützt wird.

#### Chrome (Version: 34.0.1847.131)

Was Web-Components betrifft ist Googles Chrome Vorreiter bei der Implementierung der Spezifikation. Drei Technologien von Web-Components werden bereits als stabil in Chrome bezeichnet. Auch HTML-Importe sind bereits vorhanden, jedoch ist es vor einer Nutzung erforderlich, sie zu aktivieren.

#### Opera (Version: 20.0.1387.91)

Dadurch, dass Opera seit einiger Zeit auf die Basis von Chromium (Blink) gewechselt hat, wird von Opera der gleiche Weg wie von Google erwartet. Zurzeit gibt es keine kennbaren Unterschiede was die Implementierung der Spezifikation bezüglich den beiden Browsern angeht.

#### Firefox (Version: 28.0)

Auch Mozilla versucht mit Firefox den Standard schnellstmöglich umzusetzen, jedoch gibt es einige Bugs diesbezüglich. Nur Templates werden bis dato als stabil angesehen und Custom-Elements, sowie Shadow-DOM sind nur mittels einer Flag erreichbar, beziehungsweise weisen noch Fehler auf.

#### Safari (Version: 7.0.3)

Obwohl viele Funktionen von Web-Components in Webkit implementiert wurden, wurden sie nie in Safari verwendet beziehungsweise zur Verfügung gestellt. Mit der Abzweigung des

<sup>24</sup>Mehr Information zu Bower auf <http://bower.io/>

Chromium-Port von Safari wurde begonnen sämtliche Web-Components Funktionen aus dem Browser zu entfernen.

#### Internet Explorer (Version: 11.0.9600.16659)

Microsoft gibt kein öffentliches Statement bezüglich ihren Entwicklungsplänen ab und somit ist es nicht klar, inwiefern sie die Technologien von Web-Components implementieren werden. Der vor kurzem veröffentlichte Internet Explorer 11 besitzt keine Schnittstellen für Web-Components.





















	Chrome	Opera	Firefox	Safari	IE
Templates					
HTML-Importe					
Custom Elements					
Shadow DOM					

Tabelle 4: Browser Unterstützung von Web-Components

Code-Beispiel 36 auf Seite 37 zeigt, wie die Funktionen beziehungsweise einzelnen Technologien auf Verfügbarkeit im Browser getestet werden können. Das Ergebnis kann in der Konsole des benutzten Browsers eingesehen werden.

```

1 function supportsTemplate() {
2   return 'content' in document.createElement('template');
3 }
4 function supportsCustomElements() {
5   return 'registerElement' in document;
6 }
7 function supportsImports() {
8   return 'import' in document.createElement('link');
9 }
10 function supportsShadowDom(){
11   return typeof document.createElement('div').createShadowRoot === 'function';
12 }
13
14 (function() {
15   supportsTemplate()? console.log("Templates are supported!") : console.error("
    Templates are not supported!")
16   supportsCustomElements()? console.log("Custom elements are supported!") : console
    .error("Custom elements are not supported!")
17   supportsImports()? console.log("HTML-Imports are supported!") : console.error("
    HTML-Imports are not supported!")
18   supportsShadowDom()? console.log("Shadow-DOMs are supported!") : console.error("
    Shadow-DOMs are not supported!")
19 })();

```

Code-Beispiel 36: Feature-Detection für Web-Components

### 3.7 Konklusio

In diesem Kapitel wurde das Komponentenmodell namens „Web-Components“ näher veranschaulicht. Dieses Komponentenmodell legt einen Standard fest, wie Komponenten im Web interoperabel entwickelt werden können. Um dies erreichen zu können, werden die fünf Konzepte von Web-Components kurz beschrieben:

**HTML-Templates** beinhalten Markup, das vorerst inaktiv ist, aber bei späterer Verwendung aktiviert werden kann.

**Decorators** verwenden CSS-Selektoren basierend auf den Templates, um visuelle beziehungsweise verhaltensbezogene Änderungen am Dokument vorzunehmen.

**Custom Elements** ermöglichen eigene Elemente mit neuen Tag-Namen und neuen Skript-Schnittstellen zu definieren.

**Shadow DOM** erlaubt es eine DOM-Unterstruktur vollständig zu kapseln. Dies ermöglicht die Interoperabilität der Komponenten, da sich keine Interferenzen mehr bilden können.

**HTML-Imports** definieren, wie Templates, Decorators und Custom Elements verpackt und als eine Ressource geladen werden können.

Wenn diese fünf Konzepte von Web-Components miteinander verwendet werden, ist die Wiederverwendbarkeit und Interkompatibilität einer entwickelten Komponente gegeben. Es muss jedoch bedacht werden, dass sämtliches Markup im Shadow-DOM nicht von Suchmaschinen, Screen-Readern oder dergleichen erkannt wird.

Auch die derzeitige Browser-Unterstützung von Web-Components muss berücksichtigt werden, da bis dato kein einziger Browser die Technologien zu 100% nativ unterstützt. Um dieses Problem jedoch umgehen zu können beziehungsweise Web-Components trotzdem benutzen zu können, wird in dieser Arbeit Google Polymer als Polyfill verwendet. Dadurch ist die Unterstützung sämtlicher „Evergreen“-Browser und Internet-Explorer 10 (und neuer) gewährleistet.

## 4 Google Polymer

In allgemeinen Worten beschrieben ist Google-Polymer ein Framework, das sämtliche Funktionen von Web-Components mit Hilfe eines Polyfills zur Verfügung stellt. Ein Polyfill ist ein Browser-Fallback, um Funktionen, die in modernen Browsern verfügbar sind, auch in alten Browsern verfügbar zu machen. Polymer verfolgt den Ansatz, dass alles was entwickelt wird, eine Web-Komponente ist und sich mit der Entwicklung des Webs weiterentwickelt.

In diesem Kapitel wird folgend zuerst die Architektur hinter dem Polymer-Framework erklärt. Danach werden die einzelnen Polyfills von Polymer näher erläutert und spezielle Eigenschaften des Frameworks kurz beschrieben. Polymer stellt zurzeit für vier der fünf Spezifikationen von Web-Components einen Polyfill bereit. Kapitel 4.2, 4.3, 4.4 und 4.5 folgen der Annahme, dass die Polyfills nicht miteinander verwendet werden. Wenn sämtliche Polyfills, die Polymer bereitstellt, verwendet werden möchten, stellt Polymer zwei Dateien zur Verfügung. Welche Dateien das sind und wie die Polyfills dann verwendet werden wird in Kapitel 6.2 auf Seite 56 an Hand von zwei entwickelten Komponenten beschrieben.

Abschließend wird die Unterstützung von Polymer seitens der Entwickler und seitens der Browser noch näher erläutert.

### 4.1 Polymer Architektur

Die nachstehende Abbildung 11 auf Seite 39 zeigt die Architektur von Polymer.

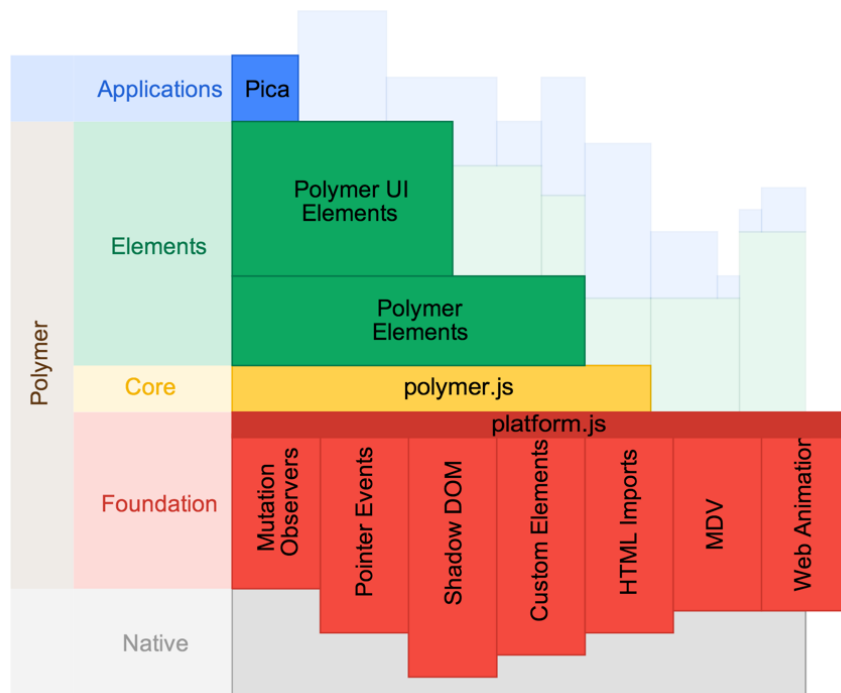


Abbildung 11: Polymers Architektur

**Die rote Schicht** visualisiert die Polyfills, die Polymer zur Verfügung stellt. Ein Polyfill ist ein Browser-Fallback, um Funktionen, die in modernen Browsern verfügbar sind, auch in alten Browsern verfügbar zu machen. Diese erlauben die Benutzung von Web-Components. Wichtig hierbei ist, dass die Größe dieser Polyfill-Bibliotheken mit der Weiterentwicklung

der Browser abnimmt. Dies bedeutet, dass je mehr Funktionalität von der Spezifikation in den Browsern implementiert ist, desto kleiner sind die Polyfill-Bibliotheken. Der Idealfall für Polymer wäre, dass sämtliche Zusatzbibliotheken, die die nativen Browser-Funktionen emulieren, nicht mehr benötigt werden.

**Die gelbe Schicht** stellt die Meinung von Google dar, wie die spezifizierten Browser-Schnittstellen zu Web-Components zusammen verwendet werden sollen. Zusätzlich zu den spezifizierten Technologien werden des Weiteren Funktionalitäten wie „data-bindings“, „change watcher“, „public properties“, etc. bereitgestellt.

**Die grüne Schicht** repräsentiert eine umfassende Reihe von Interface-Komponenten. Diese entwickeln sich ständig weiter und basieren auf der gelben, sowie roten Schicht.

## 4.2 Polyfilled Templates

Die Dateien zu dem Polyfill für Templates sind auf <http://github.com/Polymer/TemplateBinding> verfügbar.

Polyfilled Templates funktionieren mit der Inklusion des Skripts `load.js`. Dieses Skript stellt neben dem Polyfill der Spezifikation mehrere Besonderheiten bereits, wie beispielsweise „Template bindings“, „Template repeats“, „Template ifs“ und „Template refs“. All diese Attribute dienen der Bindung von Daten mit einem `<template>`-Element. Folgend wird kurz auf jedes Konzept erläutert.

**Template bindings** binden ein Objekt an ein Template. Dies wird meist verwendet, um zu garantieren, dass nur eine Instanz des Objekts für ein Template gültig ist. Code-Beispiel 37 auf Seite 40 zeigt ein minimales Beispiel zu Template-bindings.

```
1 <template bind="{ { singleton } }">
2   Creates a single instance with given bindings when singleton model data is
   provided.
3 </template>
```

Code-Beispiel 37: Polymer Template Bind

**Template repeat** erlaubt es über ein Objekt beziehungsweise Array zu iterieren und das Template mit den jeweiligen Daten des Objekts beziehungsweise Arrays zu rendern. Code-Beispiel 38 auf Seite 40 zeigt ein minimales Beispiel zu Template-repeat.

```
1 <template repeat="{ { user in users } }">
2   {{user.name}}
3 </template>
```

Code-Beispiel 38: Polymer Template Repeat

**Template if** dient dazu, um das Template mit einer „ConditionalValue“ zu versehen. Somit wird das Template nur verwendet, wenn die „ConditionalValue true“ ist. Code-Beispiel 39 auf Seite 40 zeigt ein minimales Beispiel zu Template-if.

```
1 <template if="{ { conditionalValue } }">
2   Binds if and only if conditionalValue is truthy.
3 </template>
```

Code-Beispiel 39: Polymer Template If

**Template ref** können dazu verwendet werden, um auf den Inhalt anderer Templates zu referenzieren. Code-Beispiel 40 auf Seite 41 zeigt ein minimales Beispiel zu Template-ref.

```

1 <template id="myTemplate">
2   Used by any template which refers to this one by the ref attribute
3 </template>
4
5 <template bind ref="myTemplate">
6   When creating an instance, the content of this template will be ignored, and
   the content of #myTemplate is used instead.
7 </template>

```

Code-Beispiel 40: Polymer Template Ref

Sämtliche bereitgestellten Funktionen des polyfilled Templates können miteinander verbunden und verwendet werden.

### 4.3 Polyfilled Custom-Elements

Die Dateien zu dem Polyfill für Custom-Elements sind auf <http://github.com/polymer/CustomElements> verfügbar.

Auch polyfilled Custom-Elements benötigen einen Bindestrich im Namen, um als valide zu gelten. Um den Polyfill direkt verwenden zu können, muss das Skript namens `custom-elements.min.js` inkludiert werden. Der Polyfill handhabt sämtliche Registrierungen von Custom-Elements asynchron. Jedoch werden sämtliche Registrierungen erst getätigt, wenn das Event namens `DOMContentLoaded` gefeuert wurde. Der Polyfill für Custom-Elements fügt zwei weitere Lebenszyklus-Callback Methoden zu den vier Methoden der Spezifikation hinzu. Tabelle 5 auf Seite 41 erläutert diese neuen Callback-Methoden.

Callback-Name - Polymer	Aufgerufen, wenn
created	eine Instanz des Elements erstellt wurde
ready	das Custom-Elements vollständig aufbereitet wurde
attached	eine Instanz in das Dokument eingefügt wurde
domReady	die Child-Elemente (Light DOM) erstellt wurden
detached	eine Instanz vom Dokument entfernt wurde
attributeChanged (attrName, oldVal, newCal)	eine Eigenschaft hinzugefügt, upgedated, oder entfernt wurde

Tabelle 5: Lebenszyklus-Callback Methoden bei Polymer

Die eigentliche Erstellung und Verwendung von Custom-Elements unterscheidet sich nicht gegenüber derer von nativen Web-Components und kann somit in Kapitel 3.3 auf Seite 24 nachgelesen werden.



## 4.4 Polyfilled Shadow-DOM

Die Dateien zu dem Polyfill für Shadow-DOM sind auf <http://github.com/Polymer/ShadowDOM> verfügbar.

Der Polyfill zu Shadow-DOM ist mit Hilfe von Wrappern implementiert worden. Ein Wrapper umhüllt einen nativen DOM-Knoten in einem Wrapper-Knoten. Dieser Wrapper-Knoten sieht und funktioniert beinahe identisch zu der nativen Implementation von Shadow-DOM. Diese Art des Polyfills besitzt jedoch einige Limitationen wie beispielsweise die vollständige Kapselung durch einen Wrapper. Sämtliche Inhalte in einem polyfilled Shadow-DOM können von Screen-Readern, Suchmaschinen, Browser-Extensions etc. erreicht werden. Auch ist das Prototyp-Objekt des Wrapper-Knoten ein anderes, als das Prototyp-Objekt der nativen Shadow-Root.

## 4.5 Polyfilled HTML-Imports

Die Dateien zu dem Polyfill für HTML-Imports sind auf <http://github.com/Polymer/HTMLImports> verfügbar.

Polyfilled HTML-Imports funktionieren mit der Inklusion des Scripts `html-imports.min.js`. Nachdem dieses Skript inkludiert wurde, können HTML-Importe verwendet werden, obwohl sie vom Browser eventuell nicht unterstützt werden. Für die Benutzung von Inhalten der geladenen Importe, wird ein eigenes Event namens `HTMLImportsLoaded` gefeuert. Dieses Element wird gefeuert, wenn alle HTML-Importe geladen wurden und zur weiteren Verwendung bereitstehen.

Bei nativen HTML-Importen wird das Parsen des Hauptdokuments von Skriptblöcken im Hauptdokument geblockt. Bei nativen HTML-Importen wird dies gemacht, um sicherzustellen, dass die importierten Dateien geladen und sämtliche Custom-Elements upgegradet wurden und somit im Hauptdokument verfügbar sind. Polyfilled HTML-Importe beinhalten dieses Verhalten nicht. Ersatzweise wird wieder ein Event namens `WebComponentsReady` gefeuert, das das native Verhalten „emulieren“ soll.

Auch ist zu erwähnen, dass bei nativen HTML-Importen die Referenz zum Hauptdokument mit Hilfe von `document.currentScript.ownerDocument` erreicht werden kann. Unter Benutzung des Polyfills ist das Hauptdokument mittels `document._currentScript.ownerDocument` erreichbar.

## 4.6 Besonderheiten von Polymer

Zusätzlich zu den Web-Components stellt Polymer noch zwei weitere Konzepte bereit:

### 1. Pointer-Events

Mouse- und Touch-Events sind zwei fundamental unterschiedliche Events in Browsers. Aktuelle Plattformen, die Touch-Events implementiert haben, haben auch Mouse-Events implementiert, um die Abwärtskompatibilität gewährleisten zu können. Jedoch werden nur Mouse-Events nur bedingt gefeuert und besitzen auch eine andere Semantik. Die nachstehende Liste zeigt einige Beispiele für die bedingte Feuerung von Mouse-Events:

- Mouse-Events werden nur gefeuert, wenn die Touch-Sequenz endet.
- Mouse-Events werden nicht von Elementen ohne Click-Event-Handler gefeuert.
- Click-Events werden nicht gefeuert, wenn der Inhalt einer Seite sich durch ein Mouse-Move-Event oder Mouse-Over-Event ändert.
- Click-Events werden mit einer Verzögerung von 300ms nach Ende einer Touch-Sequenz gefeuert.

Auch werden Touch-Events nur zu dem Element gesendet, dass das Touch-Start-Event bekommt. Dies ist grundsätzlich unterschiedlich zu Mouse-Events. Mouse-Events werden auf das Element, dass sich unter der Maus befindet, gesendet. Um hier eine Gemeinsamkeit erreichen zu können, werden Touch-Events mit Hilfe von `document.elementFromPoint` neu kalibriert.

Um solche Inkompatibilitäten vorzubeugen, stellt Polymer Pointer-Events bereit.

## 2. Web-Animations

Web-Animations ist eine neue Spezifikation für animierten Inhalt im Internet. Es wird als W3C Spezifikation als Teil der CSS- und SVG-Working-Group entwickelt. Diese Spezifikation thematisiert die Mängel der bereits vorhandenen Spezifikationen zu CSS- und SVG-Animationen. Web-Animations sollen des Weiteren die zugrunde liegenden Implementierungen von CSS-Transitions, CSS-Animations und SVG-Animations ersetzen. Dies würde dabei helfen, den Code für Unterstützung von Animationen im Web gering zu halten und die verschiedenen Animations-Spezifikationen interoperabel zu machen.

Polymer bietet neben den bereits erläuterten Polyfills und Besonderheiten auch die Möglichkeit vordefinierte Elemente zu verwenden. Ein Beispiel für ein vordefiniertes Element wäre das `<polymer-ajax>`-Element. Es versucht einen Standard für Entwicklerinnen und Entwickler bereitzustellen, um Ajax-Requests zu erstellen beziehungsweise abzuwickeln. Dieses Element ist ähnlich zu folgender Funktion von jQuery: `$.ajax()`<sup>25</sup>. Der Unterschied zwischen den beiden Möglichkeiten einen Ajax-Request abzuwickeln ist, dass die `$.ajax()`-Methode Abhängigkeiten besitzt, wohingegen die `<polymer-ajax>`-Methode nur pures JavaScript benutzt und somit nur Polymer als Abhängigkeit besitzt.

Wie Custom-Elements mittels Polymer entwickelt werden können und welche Coding-Conventions dieses Framework beinhaltet, wird in Kapitel 6.2 auf Seite 56 genauer erläutert.

## 4.7 Unterstützung

In diesem Kapitel wird folgend einerseits die Unterstützung seitens der Browser und andererseits die Unterstützung seitens der Entwickler zu Polymer näher betrachtet.

### 4.7.1 Unterstützung seitens der Browser

Polymer und seine Polyfills zielen darauf ab, in sämtlichen „Evergreen“-Browsern zu funktionieren. Tabelle 6 auf Seite 44 zeigt die Kompatibilität zu diversen Browsern. Ein grünes Kästchen bedeutet, dass die Funktionalität im jeweiligen Browser vorhanden ist. Rot hingegen bedeutet, dass es keine Unterstützung dieser Funktionalität gibt. Die folgende Liste beschreibt die Browser und ihre Versionen hinsichtlich der Funktionalitäten.

#### Chrome Android, Chrome, Canary, Firefox

Unter diesen Browsern funktionieren die Polyfills ohne bekannte Probleme. Folgend werden die Versionen der Browser angegeben, bei denen es getestet wurde:

**Chrome Android** Version 34.0.1847.114

**Chrome** Version: 34.0.1847.131

**Canary** Version 36.0.1964.2

**Firefox** Version 29.0

<sup>25</sup>Mehr Information zur jQuery.ajax-Funktion unter <http://api.jquery.com/jQuery.ajax/>

**Internet Explorer, Safari, Mobile Safari**

Diese Browser unterstützen die polyfilled Templates nicht. Die restlichen Polyfills funktionieren jedoch auch in diesen Browsern problemlos. Folgend werden die Versionen der Browser angegeben, bei denen es getestet wurde:

**Internet Explorer** Version 11.0.9600.16659

**Safari** Version: 7.0.3

**Mobile Safari** Version 7.0











































	Chrome Android	Chrome	Canary	Firefox	IE	Safari	Mobile Safari
Templates							
HTML- Importe							
Custom- Elements							
Shadow- DOM							
Pointer- Events							
Web- Animations							

Tabelle 6: Browser Unterstützung von Polymer

#### 4.7.2 Unterstützung seitens der Entwicklung

Google-Polymer wurde im Februar 2013 erstellt. Seit der Erstellung dieses Projekts wurde mindestens jedes Monat eine neue Version von Polymer veröffentlicht. Mit zunehmenden Publikationen von Polymer auf Events wie beispielsweise der Google-I/O<sup>26</sup>, steigt auch die Motivation der Entwicklerinnen und Entwickler von Polymer. Die Team hinter Polymer versucht sämtliche Fragen zu Polymer auf Stack-Overflow<sup>27</sup>, Google-Groups<sup>28</sup>, Reddit und eine Vielzahl anderer Foren zu beantworten. Somit besitzt dieses Projekt eine sehr gute Unterstützung von Seiten der Entwickler.

## 4.8 Konklusio

In diesem Kapitel wurde das Framework namens „Polymer“ näher veranschaulicht. Dieses Framework bietet Polyfills für vier der fünf Konzepte von „Web-Components“. Neben den Polyfills für Web-Components werden weitere Funktionalitäten wie beispielsweise „Pointer-Events“, „Web-Animations“ und eine Vielzahl vordefinierter Elemente zur Verfügung gestellt.

<sup>26</sup>Mehr Informationen zur Google I/O auf <https://www.google.com/events/io>

<sup>27</sup>Mehr Informationen zu Stack-Overflow auf <http://stackoverflow.com/>

<sup>28</sup>Mehr Informationen zu Google-Groups auf <http://groups.google.com/>

Die Unterstützung von Polymer hinsichtlich der Browser ist sehr gut. Beinahe 100% der Browser unterstützen sämtliche Polyfills von Polymer. Auch ist die Unterstützung seitens der Entwicklerinnen und Entwickler nennenswert. Das Polymer-Team versucht sämtliche Fragen von Benutzerinnen und Benutzer beantworten zu können und ist somit auch auf einer Vielzahl von Foren präsent.

Wichtig zu nennen ist jedoch, dass sämtliche Polyfills sowohl einzeln, als auch gemeinsam verwendet werden können. Wenn alle vier Polyfills gemeinsam verwendet werden, müssen nicht alle Polyfills extra inkludiert werden. Die Entwicklung von Komponenten mit Hilfe aller Polyfills wird in Kapitel 6.2 auf Seite 56 näher erläutert.

## 5 Vergleich zwischen Web-Components und Google Polymer

Dieses Kapitel dient der Beantwortung eines Teils der Forschungsfrage. Es wird ein umfangreicher Vergleich zwischen Web-Components und Google-Polymer erstellt. Dieser Vergleich beinhaltet mehrere Aspekte, wie beispielsweise den Aspekt hinsichtlich komponentenbasierter Softwareentwicklung beziehungsweise komponentenbasierter Softwarearchitektur. Weiters werden auch grundlegende Unterschiede der beiden Technologien aufgelistet und gegenübergestellt. Dies dient dem Verständnis diverser wichtiger Unterschiede, die bei der Verwendung der jeweiligen Technologie berücksichtigt werden müssen. Abschließend wird der Vergleich hinsichtlich der Unterstützung seitens der Browser und die Unterstützung seitens der Entwicklung gezogen.

### 5.1 Vergleich hinsichtlich verschiedener Arten von Softwarekomponenten

In Bezug auf verschiedene Arten von Komponenten, können auch Web-Components entsprechend ihren Aufgabenbereichen klassifiziert werden. Diese Klassifizierung kann auf Basis einer Schichtenarchitektur erfolgen. Diese Architektur dient der Trennung von Zuständigkeiten und einer losen Kopplung der Komponenten. Wie bereits in Kapitel 2.2 auf Seite 7 erklärt wurde, können Komponenten diversen Schichten zugeteilt werden. Folgend werden die Schichten in Relation mit Web-Components gesetzt.

#### Komponenten der Präsentations-Schicht

Theoretisch sind dies Komponenten, die eine nach außen sichtbare Benutzerschnittstelle bereitstellen. Hinsichtlich Web-Components würde diese Schicht aus Komponenten bestehen, die mit Hilfe von Custom-Elements, Decorators, Templates und Shadow-DOM umgesetzt wurde. Durch die Verbindung dieser Konzepte kann einerseits Wiederverwendbarkeit und andererseits vollständige Kapselung erreicht werden. Weiters können bei Custom-Elements die nach außen sichtbaren Benutzerschnittstellen definiert werden. Templates, Decorators und Shadow-DOM garantieren lediglich die Kapselung und Wiederverwendbarkeit gewisser Darstellungen.

#### Komponenten der Controlling-Schicht

Komponenten dieser Schicht dienen der Verarbeitung komplexer Ablauflogiken. Somit können sie auch als Kommunikations-Komponenten angesehen werden, die zwischen Komponenten der Business- und Präsentations-Schicht vermitteln. Komponenten, die mit dem Konzept der Custom-Elements und dem Konzept des Shadow-DOMs umgesetzt werden, können dieser Schicht angehören. Hierbei wird keinerlei Markup für die Komponente benötigt. Es wird lediglich eine benutzerdefinierte Komponente erstellt, die mit Komponenten von anderen Schichten kommunizieren kann. Durch die Verwendung von Shadow-DOM wird sämtlichen Interferenzen, die mit der Kommunikations-Komponente entstehen könnten, vorgebeugt.

#### Komponenten der Business-Schicht

Komponenten dieser Schicht lösen die eigentlichen Problemstellungen. In Bezug auf Web-Components werden für die Entwicklung von Komponenten, die dieser Schicht angehören, nur zwei Konzepte benötigt: Custom-Elements und Shadow-DOM. Custom-Elements garantieren gewisse Schnittstellen, wie Komponenten dieser Schicht verwendet werden können und Shadow-DOM dient der Vorbeugung von Interferenzen.

#### Komponenten der Integrations-Schicht

Diese Komponenten dienen der Anbindung an Alt-Systeme, Fremd-Systeme und Datenspeicher. Ähnlich den Komponenten der Business-Schicht und Controlling-Schicht, können auch Komponenten der Integrations-Schicht mit zwei Konzepten von Web-Components umgesetzt werden: Custom-Elements und Shadow-DOM.

Polymer bietet neben der Möglichkeit Custom-Elements zu erstellen auch die Möglichkeit, bereits vordefinierte Elemente zu verwenden. Diese Elemente können in zwei Kategorien unterschieden werden: Core-Elements und UI-Elements. Elemente, die der Kategorie „Core“ angehören, können der Controlling-, Business- und Integrations-Schicht zugeordnet werden. Elemente der „UI“-Kategorie, gehören lediglich der Präsentations-Schicht an, wie bereits der Name verrät.

## 5.2 Vergleich hinsichtlich klassischer Softwarearchitektur

Klassische Softwarearchitektur erstreckt sich von der Analyse des Problembereichs eines Systems bis hin zu seiner Realisierung. Sie bewegt sich nicht auf der Abstraktionsebene fein-granularer Strukturen wie Klassen oder Algorithmen, sondern vielmehr auf Ebene von Systemen. Hinsichtlich dieser Definition von Softwarearchitektur weisen Web-Components nur bedingt Gemeinsamkeiten auf. In Bezug auf die Analyse des Problembereichs bietet diese Technologie keine Hilfestellungen. Jedoch bei der Realisierung eines Systems bietet Web-Components bei Custom-Elements einige Lebenszyklen, die äußerst wichtig in einem System sind. Diese Lebenszyklen gewährleisten die Überschaubarkeit von Komponenten.

Wenn jedoch die Hilfestellungen von Web-Components bezüglich Softwarearchitektur nicht berücksichtigt werden, können diverse Mängel in einer Softwarearchitektur entstehen. Die nachstehende Liste nennt Symptome mangelhafter Softwarearchitektur, die jedoch mit der richtigen Verwendung von Web-Components nicht entstehen können.

### **Komplexität ufer aus und ist nicht mehr beherrschbar**

Wenn Konzepte wie beispielsweise Shadow-DOM von Web-Components nicht wie vorgesehen verwendet werden, kann die Komplexität einer Komponente schnell ausufern. Wenn die vollständige Kapselung einer Komponente nicht mehr gegeben ist, könnten einige Interferenzen auftreten, die möglicherweise zu fehlerhaften Ereignissen führen.

### **Wiederverwendung von Wissen und Systembausteinen ist erschwert**

Wenn sämtliche Konzepte von Web-Components gemeinsam verwendet werden, ist die Wiederverwendbarkeit und Interoperabilität von Komponenten gegeben.

### **Integration verläuft nicht reibungslos**

Symptome dieser Art können bei Web-Components mit dem Konzept des Shadow-DOMs behandelt werden. Durch die richtige Verwendung dieses Konzepts ist eine vollständige Kapselung gegeben. Somit können neue Komponenten problemlos integriert werden, ohne Interferenzen zu bereits bestehenden Komponenten zu verursachen.

Für sämtliche anderen Symptome mangelhafter Softwarearchitektur (siehe Kapitel 2.3 auf Seite 8) bieten Web-Components keine Hilfestellungen.

Die folgende Liste beschreibt die Folgen einer mangelhaften Softwarearchitektur in Relation mit Web-Components. Hierbei bieten diese Technologien Vorteile, um einige dieser Folgen verhindern zu können.

- Schnittstellen, die schwer zu verwenden beziehungsweise zu warten sind weil sie einen zu großen Umfang besitzen.

Komponenten, die mit Hilfe von Web-Components umgesetzt werden, können bei falscher Anforderungsanalyse einen zu großen Umfang besitzen. Somit obliegt es der Entwicklerin beziehungsweise dem Entwickler der Komponente diesen Punkt zu berücksichtigen.

- Quelltext, der an zahlreichen Stellen im System angepasst werden muss, wenn Systembausteine, wie beispielsweise Datenbank oder Betriebssystem, geändert werden.

Web-Components erlauben die Komposition von Komponenten. Unter richtiger Verwendung sämtlicher Konzepte von Web-Components, können keine redundanten Quelltexte entstehen. Konzepte wie Templates und Decorators verhindern redundante Quelltexte zu entwickeln.

- Klassen, die sehr viele ganz unterschiedliche Verantwortlichkeiten abdecken und deshalb nur schwer wiederzuverwenden sind („Monster“-Klassen).

Web-Components verhindert nicht die Entwicklung von „Monster“-Klassen. Es obliegt vollständig der Programmiererin beziehungsweise dem Programmierer, dass sämtliche Verantwortlichkeiten auf Komponenten aufgeteilt werden.

- Fachklassen, deren Implementierungsdetails im gesamten System bekannt sind.

Komponenten, die das Konzept von Shadow-DOM und Custom-Elements richtig verwenden, geben ihre Implementierungsdetails nicht im gesamten System preis.

### 5.3 Vergleich hinsichtlich serviceorientierter Softwarearchitektur

Sowohl native Web-Components, als auch polymer bieten keine Vorteile hinsichtlich der Erstellung serviceorientierter Architekturen. Diese Art der Architektur dient der Entwicklung von verteilten Systemen. Hierbei können Systemkomponenten eigenständige Dienste darstellen. Das System selbst kann auf geographisch verteilten Rechnern laufen. Die standardisierten Kommunikationsprotokolle wurden für den Informationsaustausch zwischen Diensten entwickelt. Software-Systeme können durch Komposition lokaler und externer Dienste aufgebaut werden. Diese Dienste wiederum können auf lokal vorhandene Komponenten zurückgreifen. Weder Web-Components, noch Polymer bieten Hilfestellungen hinsichtlich der Kommunikation mit Diensten.

### 5.4 Vergleich hinsichtlich komponentenbasierter Softwarearchitektur

Auch in Bezug auf die komponentenbasierte Softwarearchitektur bietet Polymer mehr Hilfestellungen als native Web-Components. Diese Form der Softwarearchitektur versucht sowohl die Eigenschaften der verwendeten Komponenten und Systeme, als auch ihre Abhängigkeiten und Kommunikationsarten für das zu entwickelnde System zu definieren. Um Abhängigkeiten in einem System zu lösen verwenden native Web-Components und Polymer die Technologie von HTML-Imports. Hierbei gibt es keinerlei Unterschiede bei der Verwendung.

Polymer versucht jedoch im Gegensatz zu Web-Components einige Kommunikationsarten in einem System zu regeln. Ein Beispiel in diesem Kontext wäre das `<polymer-ajax>`-Element, welches versucht, die Verwendung von `XMLHttpRequests` zu standardisieren. Des Weiteren gibt Polymer gewisse „Coding-Conventions“ vor, um die Interoperabilität zwischen Komponenten garantieren zu können. Auch sollen die Interferenzen durch die Einhaltung dieser Richtlinien gering gehalten werden. Folglich bietet somit Polymer Hilfestellungen bei „Bau“ eines Systems beziehungsweise Teilsystems.

### 5.5 Vergleich hinsichtlich komponentenbasierter Softwareentwicklung

Dadurch, dass Google-Polymer den Ansatz verfolgt, dass „alles eine Komponente ist“ (Glazkov u. a.), bietet es hinsichtlich komponentenbasierter Softwareentwicklung Vorteile gegenüber nativen Web-Components. Das zugrunde liegende Paradigma der komponentenbasierten Softwareentwicklung ist, dass ein Software-System auf Basis von Standardkomponenten aufbauen soll, anstatt bereits funktionierende Komponenten neu zu entwickeln. In diesem Kontext stellt Polymer eine

Reihe von vordefinierten Komponenten zur Verfügung, was ein klarer Vorteil gegenüber nativen Web-Components ist. Diese Komponenten können als Standardkomponenten angesehen werden, mit der Möglichkeit, sie beliebig erweitern zu können. Auch versuchen diese Standardkomponenten gewisse Funktionen zu standardisieren, wie beispielsweise die Abhandlung eines Ajax-Calls. Native Web-Components hingegen bieten derzeit keine vordefinierten Komponenten an. Somit bietet Polymer Vorteile sowohl bei der Standardisierung betrieblicher und technischer Aufgaben, als auch bei der Entwicklung allgemeiner Komponenten durch Kompositionen von vordefinierten Komponenten.

## 5.6 Vergleich der zur Verfügung gestellten Funktionen

Tabelle 8 auf Seite 50 stellt sämtliche zur Verfügung gestellten Funktionen von Web-Components und Google-Polymer gegenüber. Diese Tabelle baut auf die bereits beschriebenen Konzepte von Kapitel 3 auf Seite 16 und Kapitel 4 auf Seite 39.

Callback-Name - Spezifikation	Callback-Name - Polymer	Aufgerufen, wenn
createdCallback	created	eine Instanz des Elements erstellt wurde
-	ready	das Custom-Elements vollständig aufbereitet wurde
attachedCallback	attached	eine Instanz in das Dokument eingefügt wurde
-	domReady	die Child-Elemente (Light DOM) erstellt wurden
detachedCallback	detached	eine Instanz vom Dokument entfernt wurde
attributeChangedCallback (attrName, oldVal, newCal)	attributeChanged (attrName, oldVal, newCal)	eine Eigenschaft hinzugefügt, upgedated, oder entfernt wurde

Tabelle 7: Lebenszyklus-Callback Methoden bei Polymer



	Web-Components	Polymer
Templates	Verwendung von Templates mit Hilfe des <template>-Elements	
Template-Bidings	-	✓
Template-Conditionals	-	✓
Template-References	-	✓
Template-Repeats	-	✓
Decorators	Nur in der Spezifikation vorhanden	-
Custom-Elements	<code>document.registerElement('my-element');</code>	<ul style="list-style-type: none"> <li>• <code>document.registerElement('my-element');</code>; unter Verwendung des einzelnen Polyfills</li> <li>• <code>Polymer('my-element');</code>; unter Verwendung des gesamten Frameworks</li> </ul>
	vier Lebenszyklus-Callback-Methoden	sechs Lebenszyklus-Callback-Methoden
Shadow-DOM	<code>document.querySelector('div').createShadowRoot();</code>	<ul style="list-style-type: none"> <li>• <code>document.querySelector('div').createShadowRoot();</code> unter Verwendung des einzelnen Polyfills</li> <li>• Unter Verwendung des gesamten Frameworks ist sämtliches Markup innerhalb des eines Custom-Elements im Shadow-DOM.</li> </ul>
	Vollständige Kapselung	Limitierte Kapselung
	Inhalt des Shadow-DOMs ist nicht von Screen-Readern, Suchmaschinen und dergleichen einsehbar	Inhalt des Shadow-DOMs ist von Screen-Readern, Suchmaschinen und dergleichen einsehbar
HTML-Imports	Verwendung von HTML-Imports mit Hilfe der Relation <code>&lt;link rel='import' href='src.html'/&gt;</code> innerhalb eines Link-Tags	
	<code>document.currentScript.ownerDocument</code>	<code>document._currentScript.ownerDocument</code>
Vordefinierte-Elemente	-	✓
Pointer-Events	-	✓
Web-Animations	-	✓

Tabelle 8: Vergleich der zur Verfügung gestellten Funktionen von Web-Components und Polymer

Folgend werden die gegenübergestellten Funktionen von Tabelle 8 auf Seite 50 genauer erläutert:

### Templates

Templates werden von beiden Technologien sehr ähnlich gehandhabt. Wenn Templates von nativen Web-Components mit dem einzelnen Polyfill-Template von Polymer verglichen werden, besteht hier hinsichtlich der Funktionsweise kein Unterschied. Der Polyfill hingegen stellt zusätzlich zur Spezifikation einige Besonderheiten wie beispielsweise Template-Bindings, Template-Conditionals, etc. bereit.

### Decorators

Derzeit sind Decorators nur als Konzept des W3C vorhanden. Die Implementation der Spezifikation beziehungsweise des Konzepts ist von den Browser-Herstellern abhängig. Polymer stellt keine Informationen bereit, ob sie dieses Konzept aufnehmen werden beziehungsweise auch als Polyfill bereitstellen werden.

### Custom-Elements

Bei Custom-Elements gibt es zwischen nativen Web-Components und Polymer einige Gemeinsamkeiten. Unter Verwendung des einzelnen Polyfills funktionieren Custom-Elements wie native Web-Components. Ein Vorteil, der jedoch durch den Polyfill entsteht, ist, dass zwei zusätzliche Lebenszyklus-Callback-Methoden eingeführt werden. Tabelle 7 auf Seite 49 erläutert die Gemeinsamkeiten beziehungsweise die neuen Lebenszyklus-Callback-Methoden.

Wenn jedoch nicht der einzelne Polyfill das ganze Framework von Polymer verwendet wird, gibt es kleine syntaktische Unterschiede zur nativen Implementation. Ein Element wird nicht mehr mit Hilfe der `document.registerElement()`-Methode, sondern mit Hilfe des `Polymer()`-Methode registriert.

### Shadow-DOM

Auch bei Shadow-DOM ist der Vergleich von nativen Web-Components und Polymer abhängig davon, ob der einzelne Polyfill zu Shadow-DOM, oder das gesamte Polymer-Framework verwendet wird. Unter Verwendung des einzelnen Polyfills besteht kein Unterschied hinsichtlich der Benutzung von Shadow-DOM. Wird jedoch das gesamte Framework benutzt, wird Shadow-DOM automatisch in einem Custom-Element verwendet.

Auch wenn die Verwendung von Shadow-DOM kaum im Vergleich zur nativen Methode abweicht, gibt es dennoch große Unterschiede hinsichtlich der Implementation. Während die native Implementation von Shadow-DOM vollständige Kapselung gewährleistet, kann dies nicht mit einem Polyfill garantiert werden. Somit garantiert der Polyfill nur eine limitierte Kapselung. Objekte, die mit dem polyfilled Shadow-DOM entwickelt werden, sind vom eigentlichen DOM gekapselt, jedoch nicht ihr Inhalt. Sämtlicher Inhalt eines polyfilled Shadow-DOM ist für Suchmaschinen, Screen-Readers, Browser-Extensions und dergleichen erreichbar. Dies hat sowohl Vor- als auch Nachteile. Ein Vorteil wäre, dass die Trennung von Inhalt und Darstellung, wie in Kapitel 3.4.1 auf Seite 29 besprochen wird, nicht durchgeführt werden muss. Ein klarer Nachteil ist, dass keine vollständige Kapselung möglich ist und so Interferenzen nie vollständig ausgeschlossen werden können.

### HTML-Imports

HTML-Imports funktionieren sowohl nativ, als auch mit dem Polyfill mit Hilfe der `rel='import'`-Relation innerhalb eines Link-Tags. Es gibt nur minimale Unterschiede bei der Verwendung von nativen HTML-Imports und polyfilled HTML-Imports. Wird beispielsweise versucht das Dokument der Datei zu bekommen, das eine externe HTML-Datei importiert, ist im Polyfill dies mit Hilfe von `document._currentScript.ownerDocument` möglich. Bei nativen HTML-Imports kann ohne den Unterstrich auf das „Owner-Dokument“ referenziert werden (`document.currentScript.ownerDocument`).

Auch gibt es Unterschiede, wenn im Hauptdokument der Inhalt von geladenen HTML-Dateien verwendet wird. Bei nativen Importen ist hierbei nichts zu beachten. Bei polyfilled

Importen jedoch auf ein Event namens „HTMLImportsLoaded“ gewartet werden, um den Inhalt von Importen weiter verarbeiten zu können.

### **Vordefinierte-Elemente**

Polymer stellt einige vordefinierte Elemente bereit. Diese Elemente können beliebig verwendet und erweitert werden.

### **Pointer-Events**

Auch wenn Pointer-Events nicht Teil der Spezifikation von Web-Components sind, sind sie dennoch sehr hilfreich. Durch die Verwendung dieser Events wird die Entwicklung von Applikationen, die mehrere Eingabemöglichkeiten (Maus, Touch, Stift, etc.), verwenden, um einiges erleichtert.

### **Web-Animations**

Web-Animations gehören nicht zur Spezifikation von Web-Components. Dennoch bietet Polymer die Funktionalität dieser Spezifikation an. Web-Animations thematisieren die Mängel der bereits vorhandenen Spezifikationen von CSS- und SVG-Animationen und sollen demnach die zugrunde liegenden Implementierungen von CSS-Transitions, CSS-Animations und SVG-Animations ersetzen.

## **5.7 Vergleich der Unterstützung**

Wie bereits in Kapitel 3.6 auf Seite 36 beschrieben, gibt es derzeit keinen Browser, der native Web-Components zu 100% unterstützt. Polymer hingegen wird bereits von einigen Browsern vollständig unterstützt (siehe Kapitel 4.7 auf Seite 43).

Werden die Teams, die hinter den jeweiligen Technologien stehen, verglichen, besitzt auch das Team-Polymer einige Vorteile. Durch die Präsenz in einer großen Anzahl von Foren ist die Unterstützung seitens der Entwicklung sehr groß. Die Teams, die die Spezifikationen nativ implementieren, sind sehr wenig präsent.

## 6 Web-Components Praxisbeispiel

In diesem Kapitel wird das Praxisprojekt, das für diese Arbeit entstanden ist, beschrieben. Der gesamte Quellcode ist auf [http://github.com/gbeschbacher/bachelorarbeit\\_2/](http://github.com/gbeschbacher/bachelorarbeit_2/) im Unterordner `praxisprojekt` erreichbar. Es ist von Vorteil, wenn zu Beginn die Datei „FeatureDetection.html“ aufgerufen und das Ergebnis angesehen wird. Diese Datei zeigt die Unterstützung sämtlicher Funktionen von Web-Components in der Browser-Konsole an. Ist die Unterstützung im verwendeten Browser nicht gegeben, können die Beispiele in den Unterordnern `diagram` und `menu` nicht angesehen werden. Diese beiden Beispiele bauen auf nativen Schnittstellen von Web-Components auf. Genauer zur Entwicklung mit Hilfe von nativen Schnittstellen ist in Kapitel 6.1 auf Seite 53 beschrieben.

Trotz vielleicht fehlender Unterstützung von nativen Funktionen des Browsers, kann ein Teil des Praxisprojekts ausgeführt und angesehen werden. Googles Polyfill namens Polymer garantiert die Unterstützung von „Evergreen“-Browsern. Genauer zur Installierung und Verwendung von Polymer ist in Kapitel 6.2 auf Seite 56 beschrieben.

Generell wurden zwei verschiedene Web-Komponenten einerseits mittels nativen Schnittstellen des Browsers und andererseits mit Polymer als Polyfill entwickelt.

Die erste entwickelte Komponente ist ein dynamisches Diagramm. Mit Hilfe von „data-attributes“ können diverse optionale Einstellungen für diese Komponente festgelegt werden, wie beispielsweise das Intervall, in dem sich das Diagramm aktualisiert. Weiters kann die maximale Anzahl an sichtbaren Datenpunkte des Diagramms angegeben werden. Die Daten die im Diagramm gezeigt sind werden einfachheitshalber berechnet. Jeder Datenpunkt ist abhängig von seinem vorherigen und unterscheidet sich nur in einem vordefinierten Bereich. Die Komponente könnte ohne weiteres dahingehend ausgebaut werden, dass die Daten von einer externen Datenquelle entgegengenommen und visualisiert werden. Für die Visualisierung beziehungsweise der Erstellung des Diagramms wurde die frei verfügbare Bibliothek `canvas.js`<sup>29</sup> benutzt.

Die zweite entwickelte Komponente ist eine „Multi-Level“-Menü Komponente, die auf dem von Co-drops zur Verfügung gestellten Bibliotheken basiert<sup>30</sup>. Diese Komponente rendert eine vordefinierte Menüstruktur, die sowohl Desktop- als auch Mobile-tauglich ist. Zurzeit kann diese Komponente nur visuell angepasst werden, jedoch nicht dynamisch durch beispielsweise weitere Menüpunkte erweitert werden.

Durch die Entwicklung der beiden Komponenten wird festgestellt, inwiefern bereits komponentenbasierte Softwarearchitektur-Konzepte beziehungsweise komponentenbasierte Softwareentwicklung miteinbezogen werden. Diese Analyse wird im darauffolgenden Kapitel erstellt und dient als Fundament für die Beantwortung der Forschungsfrage.

### 6.1 Programmierung von Web-Components nach dem W3C Standard

Dieses Subkapitel baut auf den bereits beschriebenen Grundlagen von Web-Components auf (siehe Kapitel 3 auf Seite 16).

Um einige der nativen Schnittstellen für Web-Components in Chrome verwenden zu können, müssen zuerst zwei Flags aktiviert werden. Auf `chrome://flags/` muss die Flag `activate HTML-Imports` und `enable experimental Web Platform features` aktiviert werden. Dies ist zum einen für HTML-Imports und zum anderen für die Verwendung von Custom-Elements notwendig. Folglich muss in den Entwickler-Tools von Chrome Shadow-DOM aktiviert werden. Die Entwickler-Tools sind mit Hilfe des Shortcuts `ctrl+shift+j` erreichbar und in den Einstellungen dieses Tools (rechts oben an Hand eines Zahnrads zu erkennen) kann die Option `show Shadow-DOM` aktivieren werden.

<sup>29</sup>Mehr Informationen zu `canvas.js` auf <http://canvasjs.com/>

<sup>30</sup>Mehr Informationen zu den Bibliotheken auf [github](https://github.com)

Ein weiterer wichtiger Punkt um das Praxisprojekt testen zu können ist, dass es unter `http://localhost/bachelorarbeit_2/praxisprojekt` erreichbar sein muss. Am Beispiel eines „LAMP“ unter Linux wäre das gesamte Github-Projekt unter `/var/www/bachelorarbeit_2` abgelegt. Diese Struktur ist erforderlich, um CORS sicherzustellen. CORS muss wiederum aktiviert sein, um HTML-Importe korrekt verwenden zu können (siehe Kapitel 3.5 auf Seite 32).

Um sicherzustellen, dass sämtliche Funktionen von Web-Components verfügbar sind, kann die HTML-Datei `FeatureDetection.html` ausgeführt und das Resultat angesehen werden.

Folgend wird zuerst die Diagramm-Komponente beschrieben und daraufhin die Menü-Komponente.

## Diagramm-Komponente

Im Hauptordner der Diagramm-Komponente befinden sich 3 HTML-Dateien. Diese Dateien unterscheiden sich nur gering voneinander. Sie repräsentieren verschiedene Sichtweisen, wie die Technologien von Web-Components zusammen verwendet werden können. Folgend wird jede Datei näher erläutert.

## 1. main.html-Datei

Diese Datei stellt zwei Diagramm-Komponenten dar, die voneinander unabhängig sind. Mit Hilfe der `document-register()`-Methode wurde das Element `<chart-live>` registriert und mittels der verfügbaren Lebenszyklus-Methoden wird das Element erstellt. In der `createdCallback`-Methode werden sämtliche Initialisierungswerte, die für das Diagramm notwendig sind, festgelegt. Zeile 10 zeigt die Erstellung des eigentlichen Diagramms mit Hilfe von `canvasJS`. Im `attachedCallback` wird das Intervall für die Aktualisierung des Diagramms gesetzt und aufgerufen. In der `updateChart`-Methode werden, wie bereits in der Einleitung geklärt, die Daten für das Diagramm einfachheitshalber berechnet. Code-Beispiel 41 auf Seite 54 ist dem aktuellen Quellcode entnommen und zeigt einen Ausschnitt der entwickelten Methoden, welche in der `main.html`-Datei zu sehen sind.

```

1 document.registerElement('chart-live', {
2     prototype: Object.create(HTMLElement.prototype, {
3         createdCallback: {
4             value: function(){
5                 this.chartId = this.children[0].id;
6                 this.xVal = 0;
7                 this.yVal= 100;
8                 this.updateInterval = 20;
9                 this.dataLength = 500;
10                this.dps = [];
11                this.chart = new CanvasJS.Chart(this.chartId, {
12                    title:{text: "Live Chart"},
13                    data:[{
14                        type: "line",
15                        dataPoints: this.dps
16                    }]
17                });
18            }
19        },
20        attachedCallback:{
21            value:function(){
22                this.updateChart(this.dataLength);
23                var that = this;
24                setInterval(function(){that.updateChart()}, that.updateInterval);
25            }
26        },
27        detachedCallback:{value:function(){}},
28        attributeChangedCallback:{value:function(attrName, oldVal, newVal){}},
29        updateChart:{
30            value: function(count){
31                count = count || 1
32                for(var i = 0; i < count; i++){

```

```

33         this.yVal = this.yVal + Math.round(5+Math.random()*(-5-5));
34         this.dps.push({
35             x: this.xVal,
36             y: this.yVal
37         });
38         this.xVal++;
39     };
40     if(this.dps.length > this.dataLength){
41         this.dps.shift();
42     }
43     this.chart.render();
44 }
45 }
46 })
47 });

```

Code-Beispiel 41: main.html

## 2. main2.html-Datei

Diese Datei unterscheidet sich nur vom Grundkonzept der bereits erklärten `main.html`-Datei. Die gesamte Komponente wurde ausgelagert und wird nur noch über das in Code-Beispiel 44 auf Seite 55 gezeigte JavaScript verwendet. Das Diagramm, welches in der Datei `permChart.html` definiert wird, wird mit einem HTML-Import geladen. Die `permChart.html`-Datei unterscheidet sich von den Diagramm-Funktionen nicht zum Diagramm in Beispiel 41 auf Seite 54. Der grundlegende Unterschied ist der Aufbau der Komponente. Code-Beispiel 42 auf Seite 55 zeigt, dass bei diesem Beispiel bereits ein `<template>`- und `<content>`-Element verwendet wird. Diese beiden Elemente sind wichtig, um das Shadow-DOM in diesem Fall richtig benutzen zu können. Code-Beispiel 43 auf Seite 55 stellt dies genauer dar. Nur statische Elemente des Diagramms befinden sich im Shadow-DOM, da sich das Diagramm über die externe Bibliothek `canvasJS` aktualisiert und somit noch von außen erreichbar sein muss.

```

1 <template id="chart">
2   <chart-live>
3     <div id="tryThis" style="height: 300px; width: 100%; ">
4       <content></content>
5     </div>
6   </chart-live>
7 </template>

```

Code-Beispiel 42: permChart.html-Aufbau

```

1 var template = currentDoc.querySelector("#chart");
2 var shadow = document.querySelector('#tryThis');
3 shadow.createShadowRoot().appendChild(template.content);

```

Code-Beispiel 43: permChart.html-Verwendung

```

1 var permChartContent = document.querySelector('link[rel="import"]');
2 var template = permChartContent.import.querySelector('template');
3 document.body.appendChild(document.importNode(template.content, true));

```

Code-Beispiel 44: main2.html-Verwendung von der in `permChart.html` definierten Komponente

## 3. main3.html-Datei

Dieses Beispiel ist sehr ähnlich zu Beispiel 1 aus Datei `main.html`. Der Unterschied liegt darin, dass die Datei `main3.html` die Diagramm-Komponente vollständig importiert und dieser Import sämtliche Notwendigkeiten, wie die Registrierung des Custom-Elements `<chart-live>`, übernimmt. Somit kann das Element verwendet werden, ohne selbst ein Element registrieren zu müssen.

### Menü-Komponente

Die Menü-Komponente ist sehr einfach und ähnlich zur Datei `main3.html` der Diagramm-Komponente aufgebaut. In der Hauptdatei der Menükomponente wird lediglich die `<push-menu>`-Komponente per HTML-Import geladen. Die Registrierung und Darstellung, sprich Markup des Elements wird in eine externe Datei (`pushMenu.html`) ausgelagert, um so die Wiederverwertbarkeit garantieren zu können, ohne Code duplizieren zu müssen.

Die `pushMenu`-Datei beginnt mit einem Import der benötigten CSS- und JavaScript-Dateien von Codrops (siehe Code-Beispiel 45 auf Seite 56). Daraufhin folgt das HTML-Markup des Menüs, wie in Code-Beispiel 46 auf Seite 56 zu sehen ist. Es wird aus Platzgründen nur der Anfang des Templates gezeigt. Die grundlegende Initialisierung des Menüs liegt ausschließlich in der `attachedCallback`-Lebenszyklus-Methode, wie in Code-Beispiel 47 auf Seite 56 zu sehen ist.

Ein wichtiger Punkt bei dieser Komponente, was die Wiederverwertbarkeit anbelangt, ist, dass sie derzeit nur statisch entwickelt wurde. Dies bedeutet, dass sie keinerlei Anpassungsmöglichkeit hinsichtlich der Menüstruktur für andere Projekte bietet.

```
1 <link rel="import" href="misc.html" />
```

Code-Beispiel 45: `pushMenu.html` - Import der Bibliotheken von Codrops

```
1 <template id="mainMenu">
2 <div class="container">
3   <div class="mp-pusher" id="mp-pusher"><!-- pusher -->
```

Code-Beispiel 46: `pushMenu.html` - Beginn des Templates

```
1   attachedCallback:{
2     value:function(){
3       new mlPushMenu(document.getElementById('mp-menu'), document.
4         getElementById('trigger'));
5     },
6   },
```

Code-Beispiel 47: `pushMenu.html` - `attachedCallback`-Methode des Menüs

## 6.2 Programmierung von Web-Components mit Hilfe von Google Polymer

Dieses Subkapitel baut auf den bereits vorher beschriebenen Grundlagen von Google Polymer auf (siehe Kapitel 4 auf Seite 39). Weiters wurden sämtliche Informationen, die in diesem Kapitel über Polymer genannt werden, von der Dokumentation zu Polymer (<http://www.polymer-project.org/>) entnommen (siehe Glazkov u. a.).

Die Verwendung von Google Polymer ist einfacher, als die von nativen Web-Components. Der empfohlene Weg um Polymer korrekt zu installieren ist „bower“<sup>31</sup>. Dadurch, dass im Unterordner `praxisprojekt/polymer` bereits eine `bower.json`-Datei vorhanden ist, muss lediglich `bower install` in der Konsole im korrekten Ordner ausgeführt werden, um Polymer zu installieren. In der `bower.json`-Datei sind bereits sämtliche Konfigurationen vorhanden, um die Version von Polymer (0.2.2) zu bekommen, mit der das Praxisprojekt entwickelt wurde.

Folgend wird zuerst ein Beispiel mit einer vordefinierten Komponente von Polymer gezeigt und daraufhin folgt eine nähere Erläuterung der beiden selbst entwickelten Komponenten.

<sup>31</sup>Bower kann über npm mit Hilfe von `npm install -g bower` installiert werden. Mehr Informationen zu Bower auf <http://bower.io/>

### Vordefinierte Elemente in Polymer

Wie bereits in Kapitel 4 auf Seite 39 kurz erwähnt, gibt es bei Polymer einige vordefinierte Elemente, die den Entwicklerinnen und Entwicklern einiges erleichtern sollen beziehungsweise einen gewissen Standard in die Entwicklung bringen soll.

Um Polymer auf einer Seite benutzen zu können, muss zuerst die Polyfill-Unterstützung geladen werden. Dies entspricht der gelben Schicht der bereits geklärten Abbildung 11 auf Seite 39. Diese Datei soll vor sämtlicher DOM-Manipulation geladen werden, um garantieren zu können, dass sämtliche gewünschte Funktionen zur Verfügung stehen. Im nächsten Schritt kann bereits das gewünschte Element in der Hauptdatei geladen und danach deklariert werden. In Code-Beispiel 48 auf Seite 57 werden die bereits genannten Schritte gezeigt und darüber hinaus wird die vom `<polymer-ajax>`-Element zur Verfügung gestellte Schnittstelle benutzt.

```

1 <head>
2   <script src="bower_components/platform/platform.js"></script>
3   <link rel="import" href="bower_components/polymer-ajax/polymer-ajax.html">
4 </head>
5 <body>
6   <polymer-ajax url="http://example.com/json" handleAs="json"></polymer-ajax>
7   <script>
8     window.addEventListener('polymer-ready', function(e) {
9       var ajax = document.querySelector('polymer-ajax');
10
11       ajax.addEventListener('polymer-response', function(e) {
12         console.log(this.response);
13       });
14
15       ajax.go(); // Call its API methods.
16     });
17   </script>
18 </body>

```

Code-Beispiel 48: Einsatz einer Polymer-Komponente

Elemente können jegliche Art von Attributen übergeben bekommen. Valide Attribute zu definieren obliegt der Autorin beziehungsweise dem Autor der Komponente. Bei vordefinierten Elementen können die erwarteten Typen für jedes Attribut in der Element-Referenz<sup>32</sup> nachgeschlagen werden.

Polymer bietet weitere Unterstützungen, die sehr hilfreich bei der Visualisierung beziehungsweise Gestaltung von Elementen sind. Beispielsweise gibt es das `unresolved`-Attribut, um somit „FOUC“<sup>33</sup> vorzubeugen. Es gibt eine Vielzahl von CSS-Selektoren, die dabei helfen Polymer-Elemente besser erreichen und gestalten zu können. Diese Selektoren werden jedoch nicht in dieser Arbeit verwendet.

### Diagramm-Komponente

Wie bereits bei der vordefinierten Polymer-Komponente wird auch für die Diagramm-Komponente zuerst das Skript der Plattform geladen und im Anschluss darauf die Komponente. Dies reicht bereits in der Hauptdatei aus, um die Komponente verwenden zu können. In der Datei `polymer-chart.html` wird zuerst die Basis von Polymer namens `polymer.html` geladen. Es sind keine weiteren Dateien notwendig, um ein grundlegendes Element erstellen zu können. Für die Diagramm-Komponente hingegen muss, wie bei der nativen Implementation der Komponente, `canvasjs.html` geladen werden. Polymer erleichtert die Syntax ein klein wenig, was die Registrierung des Elements betrifft. Grundlegend funktioniert die Registrierung eines Elements mit Hilfe des Polymer-Objekts, wie Code-Beispiel 49 auf Seite 58 veranschaulicht. Ein weiterer, wichtiger Punkt dieses Code-Beispiels ist, dass die bereits genannten sechs Lebenszyklus-Methoden vorhanden sind.

<sup>32</sup>Mehr Information zu den Element-Referenzen auf <http://www.polymer-project.org/docs/elements/>

<sup>33</sup>Flash of unstyled Content



Dadurch, dass bei der Diagramm-Komponente sämtliches Markup durch die Bibliothek `canvas.js` erstellt wird, muss sichergestellt werden, dass das Element von außen erreichbar ist. Standardmäßig wird Markup für eine Polymer-Komponente wie es in Code-Beispiel 50 auf Seite 58 zu sehen ist erstellt. Dieses Beispiel ist direkt auf die Diagramm-Komponente angepasst. Der Name der Komponente muss, gleich wie in der Spezifikation, immer einen Bindestrich beinhalten, um valide zu sein. Die `attributes=""` des Elements lassen bereits erkennen, welche Eigenschaften die Benutzerin beziehungsweise der Benutzer der Komponente von außen verändern kann. Wie diese Eigenschaften von außen gesetzt verwendet werden können, zeigt Code-Beispiel 51 auf Seite 58. Ein weiteres, wichtiges Merkmal bei Polymer-Komponenten ist, dass standardmäßig sämtliches Markup, das in der Komponenten-Datei (hier `polymer-chart.html`) definiert ist, im Shadow-DOM liegt. Daher wird auch im Fall dieser Komponente das `<content>`-Element verwendet, damit das dynamisch erzeugte Markup von `canvas.js` hier gerendert wird.

Sämtliche Funktionen und Variablen, um die Komponente steuern beziehungsweise erstellen zu können bleibt gleich wie bei der nativen Implementation. Code-Beispiel 52 auf Seite 58 zeigt den vollständigen Code der Komponente. Hierbei muss beachtet werden, dass `this` in dem verwendeten Scope immer eine Referenz zum Polymer-Element repräsentiert.

```

1 Polymer('chart-live',{
2   created: function() {},
3   ready: function() {},
4   attached: function() {},
5   domReady: function() {},
6   detached: function() {},
7   attributeChanged: function(attrName, oldVal, newVal) {},
8 });

```

Code-Beispiel 49: Registrierung einer Polymer-Komponente

```

1 <polymer-element name="chart-live" attributes="chartId chartClass dataLength
   updateInterval">
2   <template>
3     <content></content>
4   </template>
5 </polymer-element>

```

Code-Beispiel 50: Markup in einer Polymer-Komponente

```

1 <chart-live chartId="tempChart" updateInterval=1000 dataLength=2000 chartClass="
   chartLive2"></chart-live>

```

Code-Beispiel 51: Markup einer verwendeten Polymer-Komponente

```

1 <polymer-element name="chart-live" attributes="chartId chartClass dataLength
   updateInterval">
2   <template>
3     <content></content>
4   </template>
5   <script>
6     Polymer('chart-live',{
7       created: function() {},
8       ready: function(){
9         _canvasDiv = document.createElement('div');
10        _canvasDiv.className = this.chartClass = this.chartClass || "chartLive";
11        _canvasDiv.id = this.chartId = this.chartId || "chartLive"
12
13        this.updateInterval = this.updateInterval || 20;
14        this.dataLength = this.dataLength || 500;
15        this.appendChild(_canvasDiv);
16
17        this._xVal = 0;
18        this._yVal = 100;
19        this._dps = [];
20        this._chart = new CanvasJS.Chart(this.chartId, {

```

```

21         title:{text: this.chartId},
22         data:[{
23             type: "line",
24             dataPoints: this._dps
25         }]
26     });
27 },
28     attached: function(){
29         this.updateChart(this.dataLength);
30         var that = this;
31         setInterval(function(){
32             that.updateChart();
33         }, that.updateInterval);
34     },
35     detached: function(){},
36     attributeChanged: function(attrName, oldVal, newVal){},
37     updateChart: function(count){
38         count = count || 1;
39         for(var j=0; j<count; j++){
40             this._yVal = this._yVal + Math.round(5+Math.random()*(-5-5));
41             this._dps.push({
42                 x: this._xVal,
43                 y: this._yVal
44             });
45             this._xVal++;
46         };
47         if(this._dps.length > this.dataLength){
48             this._dps.shift();
49         }
50         this._chart.render();
51     }
52
53     });
54 </script>
55 </polymer-element>

```

Code-Beispiel 52: polymer-chart.html

## Menü-Komponente

Von der Hauptdatei `index.html` ist die Menü-Komponente sehr ähnlich zur Diagramm-Komponente. Es wird lediglich das „Herz“ von Polymer namens `platform.js` geladen und darauffolgend das Custom-Element. Im `body` wird folglich das Element deklariert. Weiters werden drei globale Funktionen definiert, welche für das Menü notwendig sind.

Das Custom-Element unterscheidet sich sehr von der Diagramm-Komponente. Es beginnt mit dem Laden externer Skripte. Folgend wird das Markup des Menüs festgelegt. Code-Beispiel 53 auf Seite 60 zeigt den Beginn des Elements, wo bereits ein wichtiges Merkmal zu sehen ist. Der `<style>`-Tag innerhalb des Templates lädt mit Hilfe mehrerer `CSS@import` -Befehle diverse Dateien. Wichtig hierbei ist, dass Polymer dies erkennt und sämtliche Gestaltungsdateien in diesem `<style>`-Tag einbettet. Dies hat zur Folge, dass sämtliche Gestaltungen in diesem Element gekapselt sind und somit keinerlei Problem darstellen, wenn beispielsweise eine zweite Containerklasse in der Hauptdatei vorhanden ist. Somit können keine Interferenzen von CSS auftreten. Das Markup der Menü-Komponente unterscheidet sich nur minimal zum Markup der nativen Implementation, deswegen wird es nicht näher besprochen.

Code-Beispiel 54 auf Seite 60 zeigt einen weiteren wichtigen Inhalt von Polymer. Es wird zuerst einer „Immediate Function“ aufgerufen. Dies hat zur Folge, dass sämtliche Funktionen innerhalb dieser „Immediate Function“ nach außen hin gekapselt sind. Die Registrierung des Polymer-Elements funktioniert innerhalb dieser Funktion gleich. An Hand von persönlichen Coding-Conventions kann bereits eine Aussage über öffentliche und private Methoden getroffen werden. Sämtliche Methoden, die mit einem `_` als Funktionsname beginnen, sind nicht öffentlich. Auch ist es ersichtlich, dass sämtliche Methoden innerhalb des Polymer-Objekts von außen erreichbar sind.

```

1 <polymer-element name="mega-menu" attributes="">
2   <template>
3     <style>
4       @import "http://localhost/bachelorarbeit_2/praxisprojekt/polymer/menu/css/
        normalize.css";
5       @import "http://localhost/bachelorarbeit_2/praxisprojekt/polymer/menu/css/
        demo.css";
6       @import "http://localhost/bachelorarbeit_2/praxisprojekt/polymer/menu/css/
        icons.css";
7       @import "http://localhost/bachelorarbeit_2/praxisprojekt/polymer/menu/css/
        component.css";
8       .scroller{
9         overflow-y: hidden;
10      }
11    </style>
12
13    <div class="container">

```

Code-Beispiel 53: polymer-menu.html

```

1 (function(){
2   Polymer('mega-menu', {
3     options : {
4       type : 'overlap', // overlap || cover
5       levelSpacing : 40,
6       backClass : 'mp-back'
7     },
8     created: function() {},
9     ready: function() {},
10    attached: function(){
11      this.el = this.$["mp-menu"];
12      this.trigger = this.$.trigger;
13      this.support = Modernizr.csstransforms3d;
14      if(this.support) {
15        _init(this);
16      }
17    },

```

Code-Beispiel 54: polymer-menu.html

## 7 Konklusio

Wie Komponenten im Web entwickelt werden können beziehungsweise welche Anforderungen sie gerecht werden müssen, wurde bis dato nicht standardisiert. Somit gab es mehrere Möglichkeiten, wie Plugins, Widgets, etc. entwickelt wurden. Jedoch waren die Interoperabilitäten zwischen den Komponenten meist nicht gegeben, da es keine standardisierte Kommunikationssprache zwischen den Komponenten gab. Dieses Problem wird durch die Verwendung von Web-Components gelöst. Web-Components ist ein Sammelbegriff von fünf Konzepten. Wenn sämtliche Konzepte zusammen verwendet werden, ist die Interoperabilität und Wiederverwendbarkeit von Web-Komponenten gegeben. Um dies näher erläutern zu können werden folgend die Konzepte von Web-Components mit der Definition einer klassischen Softwarekomponente gegenübergestellt:

### 1. A component is a unit of independent deployment

Hinsichtlich Web-Components wird dieser Punkt hauptsächlich durch HTML-Importe und Shadow-DOM realisiert. Damit eine Komponente „independent deployable“ sprich unabhängig auslieferbar ist, muss sie auch so konzipiert und entwickelt sein. Shadow-DOM erlaubt es, sämtliche Unterstrukturen eines Elements vollständig zu kapseln, sodass es keine Interferenzen zu anderen Komponenten geben kann. Um sie noch auszuliefern zu können, werden HTML-Importe verwendet. Diese erlauben es, eine Komponente mit all ihren Abhängigkeiten als eine Datei auszuliefern.

### 2. A component is a unit of third-party composition

Komponenten, die mit den Konzepten von Custom-Elements und Shadow-DOM von Web-Components umgesetzt wurden, sind zusammensetzbar. Durch die gemeinsame Verwendung dieser beiden Technologien können Autorinnen und Autoren von Komponenten klar definierte Schnittstellen entwickeln und Komponenten vollständig kapseln. Dies bedeutet, dass die entwickelten Komponenten nur über die zur Verfügung gestellten Schnittstellen verwendet werden können. Durch die Gegebenheit dieser klar definierten Grenze zwischen Innerem und Äußerem können keine Interferenzen mit anderen Komponenten entstehen. Somit ist auch die Interaktion zwischen dem System und der Komponente über die definierten Schnittstellen gegeben.

### 3. A component has no (externally observable) state

Die Definition besagt, dass eine Originalkomponente nicht von Kopien ihrer selbst unterschiedlich sein darf. Web-Components bieten für diesen Punkt der Definition keinerlei Hilfestellungen und somit obliegt es der Autorin beziehungsweise dem Autor der Komponente, dies zu berücksichtigen. Auch wenn externe Zustände erlaubt wären, die nicht zur Funktionalität der Komponente an sich beitragen, sind Web-Components in diesem Kontext nicht behilflich.

Wenn die Konzepte von Web-Components und Polymer hinsichtlich klassischer Softwarearchitektur in Verbindung gebracht werden, entstehen Überschneidungen der Grundkonzepte. Somit werden die Konzepte bereits von klassischer Softwarearchitektur beeinflusst und nehmen auf die Konzepte dieser Rücksicht (siehe Kapitel 5.2 auf Seite 47).

Folgend werden nochmals kurz die Resultate der Vergleiche von Web-Components und Polymer mit diversen Aspekten der Softwarearchitektur beschrieben:

### Vergleich hinsichtlich komponentenbasierter Softwarearchitektur

Durch die Bereitstellung von vordefinierten Elementen bietet Polymer mehr Vorteile hinsichtlich komponentenbasierter Softwarearchitektur als native Web-Components.

### Vergleich hinsichtlich serviceorientierter Softwarearchitektur

In Bezug auf die serviceorientierte Softwarearchitektur bieten weder native Web-Components, noch Polymer Hilfestellungen. Mit Kompositionen von Systemkomponenten, die bei SOA

Dienste darstellen, können Software-Systeme aufgebaut werden. Dienste wiederum können auf lokal vorhandene Komponenten zurückgreifen.

Werden Web-Components und Polymer mit dem Konzept der komponentenbasierten Softwareentwicklung in Relation gesetzt, ergibt sich, dass Polymer mehr Vorteile hinsichtlich dieser Entwicklungsmethode bietet, als native Web-Components. Durch das Paradigma, das Polymer verfolgt („Alles ist eine Komponente und entwickelt sich mit dem Web mit “), entsteht ein Vorteil gegenüber nativen Web-Components der folgend erklärt wird: Auf Grund dieses Paradigmas stellt Polymer eine Vielzahl von vordefinierten Elementen zur Verfügung, die beliebig verwendet und erweitert werden können. Durch diese bereitgestellten Elemente verfolgt Polymer noch ein weiteres wichtiges Konzept von komponentenbasierter Softwareentwicklung: Systeme sollen auf Basis von Standardkomponenten aufbauen, anstatt bereits funktionierende Komponenten neu zu entwickeln.

Im Hinblick auf die zur Verfügung gestellten Funktionen von nativen Web-Components und Polymer ist Polymer im Vorteil. Folgend werden die Vorteile von Polymer gegenüber Web-Components kurz erläutert:

- Zusätzlich zu normalen Templates stellt Polymer eine Vielzahl von Template-Besonderheiten wie beispielsweise Template-Bindings bereit. Diese Besonderheiten vereinfachen beziehungsweise erweitern die Funktionalitäten von Templates um einiges.
- Durch die zwei neuen Lebenszyklus-Callback-Methoden von Custom-Elements bei Polymer ist es möglich, besser auf spezielle Ereignisse zu reagieren.
- Durch die limitierte Kapselung von Shadow-DOM sind sämtliche Inhalte im Shadow-DOM von Suchmaschinen, Screen-Readern, Browser-Extensions, etc. erreichbar. Dies kann sowohl ein Vor-, als auch Nachteil sein. Somit obliegt die Entscheidung hierbei bei der Leserin beziehungsweise dem Leser des Autors.
- Die Bereitstellung von vordefinierten Elementen bringt einige Vorteile hinsichtlich komponentenbasierter Softwarearchitektur und komponentenbasierter Softwareentwicklung mit sich.
- Die Unterstützung der Spezifikationen von Web-Animations und Pointer-Events, sowie die Bereitstellung von Polyfills dieser Konzepte, vereinfachen die Entwicklung von Cross-Browser-Applikationen um einiges.

Ein weiterer wichtiger Vorteil von Polymer gegenüber Web-Components ergibt sich, wenn die Unterstützung hinsichtlich der Entwicklung und hinsichtlich der Browser verglichen wird. Polymer wird bereits von einigen Browsern zu 100% unterstützt. Auch ist die Unterstützung seitens der Entwicklerinnen und Entwickler von Polymer sehr gut. Durch die Aktivität und Präsenz der Entwicklerinnen und Entwickler von Polymer in einigen Foren steigt die Attraktivität des Frameworks.

## 7.1 Ausblick von Web-Components

Die nahe Zukunft von Web-Components ist sehr von der Entwicklung der Browser abhängig. Solange kein einziger Browser Web-Components zu 100% unterstützt, wird die Akzeptanz dieser Technologien unter den Entwicklerinnen und Entwicklern gering sein. Auch Polyfills helfen hier nur bedingt, da sie nur „Evergreen“-Browser und Internet Explorer 10+ unterstützen. Somit muss noch einige Zeit vergehen, bis die breite Masse die dafür notwendigen Browser installiert hat. Auch ist derzeit noch kein Projekt in Production-Environment bekannt, das auf die Technologien von Web-Components aufbaut.

Hinsichtlich eines langfristigen Ausblicks können Web-Components jedoch die Entwicklung von Komponenten im Web stark verändern. Mit der Akzeptanz der Entwicklerinnen und Entwickler

und der Unterstützung der Browser werden Komponenten auf eine neue Art erstellbar sein. Es ist wahrscheinlich, dass JavaScript-Bibliotheken und Komponentenframeworks an Beliebtheit verlieren beziehungsweise auf Technologien von Web-Components umsteigen werden. Darüber hinaus wird es wahrscheinlich eine Plattform geben, die eine Reihe von Komponenten bereitstellen wird.

## 7.2 Offene Fragen hinsichtlich der Entwicklung

In Kapitel 2.3 auf Seite 8 wird der theoretische Unterschied zwischen serviceorientierter Softwarearchitektur und komponentenbasierter Softwarearchitektur kurz beschrieben. Hinsichtlich der Entwicklung von Web-Components könnte man erforschen, inwiefern sie als Service einsetzbar sind beziehungsweise welche Anforderungen dafür benötigt werden.

In Kapitel 3 auf Seite 16 werden lediglich die Konzepte von Web-Components und deren Verwendung näher beschrieben. Jedes Subkapitel könnte dahingehend erweitert werden, dass die dahinterstehenden Browser-Funktionalitäten näher erläutert werden.

Auch werden bei der Entwicklung der Komponenten in Kapitel 6 auf Seite 53 nur zwei Aspekte betrachtet: native Web-Components und Google-Polymer. Hier könnten mehrere Frameworks beziehungsweise Bibliotheken miteinander verglichen werden. Einerseits könnten sämtliche Polyfills von Web-Components und andererseits sämtliche Bibliotheken, die der Entwicklung von Komponenten dienen, genauer betrachtet und gegenübergestellt werden.

Die Konzepte von Web-Animations und Pointer-Events wurden in Kapitel 4.6 auf Seite 42 nur theoretisch besprochen. Diese beiden Spezifikationen könnten genauer analysiert und hinsichtlich Browser-Kompatibilität getestet werden.

## Literatur

- Andresen, Andreas. 2003. *Komponentenbasierte softwareentwicklung: mit mda, uml und xml*. München und Wien: Hanser. ISBN: 3446222820.
- Bidelman, Eric. 2013a. *Custom elements*. <http://www.html5rocks.com/en/tutorials/webcomponents/customelements/>. [Online, 12.04.2014].
- . 2013b. *Html imports - #include for the web*. <http://www.html5rocks.com/en/tutorials/webcomponents/imports/>. [Online, 12.04.2014].
- . 2013c. *Html's new template tag*. <http://www.html5rocks.com/en/tutorials/webcomponents/template>. [Online, 12.04.2014].
- Bredemeyer, Dana, und Ruth Malan. 2004. *Software architecture action guide*. <http://www.ruthmalan.com>. [Online, 30.03.2014].
- Buschmann, Frank. 1996. *Pattern-oriented software architecture*. Chichester u. a.: Wiley. ISBN: 0471958697.
- Chiu, Aaron, Oleksandr Kuvshynov, Alex Feinberg, Alex Himel, Ali Parr, Alok Menghrajani, Amjad Masat u. a. 2011. *Facebook-React*. <https://github.com/facebook/react>. [Online, 19.04.2014; Github-Stars: 5698].
2013. *Decorators - nextgen markup pt.2*. <http://www.prevent-default.com/decorators-nextgen-markup-pt-2/>. [Online, 12.04.2014].
- Dimitri Glazkov, Tony Ross, Rafael Weinstein. 2013. *Html templates*. <http://www.w3.org/TR/2013/WD-html-templates-20130214/>. [Online, 12.04.2014].
- Dominic Cooney, Dimitri Glazkov. 2013. *Introduction to web components*. <http://www.w3.org/TR/components-intro/>.
- Forbes, Bryan, Christophe Jolif, Colin Snover, Dylan Schiemann, Ed Chatelain, James Thomas, Kenneth Franqueiro u. a. 2005. *Dojo*. <https://github.com/dojo>. [Online, 19.04.2014; Github-Stars: 1316].
- Fowler, Martin. 2005. *The new methodology*. <http://martinfowler.com/articles/newMethodology.html>. [Online, 30.03.2014].
- Gamma, Erich. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading und Mass: Addison-Wesley. ISBN: 978-0201633610.
- Glazkov, Dimitri. 2013a. *Custom elements*. <http://www.w3.org/TR/2013/WD-custom-elements-20131024/>. [Online, 20.04.2014].
- . 2013b. *Shadow dom*. <http://www.w3.org/TR/2013/WD-shadow-dom-20130514/>. [Online, 20.04.2014].
- Glazkov, Dimitri, und Dominic Cooney. 2013. *Introduction to web components*. <http://www.w3.org/TR/components-intro/#decorator-section>. [Online, 20.04.2014].
- Glazkov, Dimitri, und Hajime Morrita. 2014. *Html imports*. <http://www.w3.org/TR/2014/WD-html-imports-20140311/>. [Online, 20.04.2014].
- Glazkov, Dimitri, Erik Arvidsson, Daniel Freedman, Steve Orvell, Scott Miles, Frankie Fu, Eric Bidelman u. a. *Polymer project*. <http://www.polymer-project.org/>. [Online, 19.04.2014].
- Osmani, Addy, Anne-Gaelle Colom, Adam Sontag, Alexander Schmitz, Aulvi, Bob Holt, Juan-Pablo Buritica u. a. 2006. *JQuery*. <https://github.com/jquery/jquery>. [Online, 19.04.2014; Github-Stars: 30391].

- Osmani, Addy, Anne-Gaelle Colom, Adam Sontag, Alexander Schmitz, Aulvi, Bob Holt, Juan-Pablo Buritica u. a. 2008. *JQuery ui*. <https://github.com/jquery/jquery-ui>. [Online, 19.04.2014; Github-Stars: 8266].
- Shaw, Mary, und David Garlan. 1996. *Software architecture: perspectives on an emerging discipline*. Upper Saddle River und N.J: Prentice Hall. ISBN: 0131829572.
- Sommerville, Ian. 2011. *Software engineering*. 9th ed. Boston: Pearson. ISBN: 9780137053469.
- Szyperski, Clemens, Dominik Gruntz und Stephan Murer. 2002. *Component software: beyond object-oriented programming*. 2nd ed. New York und London: ACM / Addison-Wesley. ISBN: 0201745720, [http://www.sei.cmu.edu/productlines/frame\\_report/comp\\_dev.htm](http://www.sei.cmu.edu/productlines/frame_report/comp_dev.htm).
- Vogel, Oliver. 2009. *Software-architektur: Grundlagen - Konzepte - Praxis*. 2. Aufl. Heidelberg: Spektrum, Akad. Verl. ISBN: 9783827419330.
- Wheeler, David. 1952. *The use of sub-routines in programmes*. ACM. doi:10.1145/609784.609816.
- Williams, Ashley, Brad Green, Brian Ford, Caitlin Potter, Chirayu Krishnappa, Paul Rohde, Rob Eisenberg u. a. 2009. *Angular js*. <https://github.com/angular/angular.js>. [Online, 19.04.2014; Github-Stars: 22891].



## Abbildungsverzeichnis

1	Software-Komponente aus unterschiedlichen Sichtweisen . . . . .	6
2	Beispiel einer Schichten-Architektur, Urldate: 04.2014 <a href="http://www.oop-uml.de/drei-schichten-architektur.php">http://www.oop-uml.de/drei-schichten-architektur.php</a> . . . . .	7
3	Zuteilung der Komponenten . . . . .	7
4	Serviceorientierte Architektur, Urldate: 04.2014 <a href="http://www.w3.org/2003/Talks/0521-hh-wsa/slide5-0.html">http://www.w3.org/2003/Talks/0521-hh-wsa/slide5-0.html</a> . . . . .	11
5	Beispiel von Web-Components im Browser an Hand eines Datepickers, Urldate: 04.2014 <a href="https://s3.amazonaws.com/infinum.web.production/repository_items/files/000/000/238/original/datepicker.jpg">https://s3.amazonaws.com/infinum.web.production/repository_items/files/000/000/238/original/datepicker.jpg</a> . . . . .	17
6	Beispiel von Web-Components im Browser an Hand eines Datepickers, Urldate: 04.2014 <a href="https://s3.amazonaws.com/infinum.web.production/repository_items/files/000/000/236/original/datepicker_shadow_dom.jpg">https://s3.amazonaws.com/infinum.web.production/repository_items/files/000/000/236/original/datepicker_shadow_dom.jpg</a> . . . . .	18
7	Visualisierung des DOM eines inaktiven Templates, Urldate: 04.2014 <a href="http://www.prevent-default.com/wp-content/uploads/2013/04/document-fragment-300x132.png">http://www.prevent-default.com/wp-content/uploads/2013/04/document-fragment-300x132.png</a> . . . . .	19
8	Decorator-Pattern UML . . . . .	20
9	Beispiel einer Shadow-Root Node . . . . .	29
10	Ausgangsbeispiel von der Trennung von Darstellung und Inhalt bei Shadow-DOM (Dominic Cooney, Dimitri Glazkov 2013) . . . . .	30
11	Polymers Architektur, Urldate: 04.2014 <a href="http://www.polymer-project.org/images/architecture-diagram.svg">http://www.polymer-project.org/images/architecture-diagram.svg</a> . . . . .	39

## Tabellenverzeichnis

1	Unterschiede zwischen Diensten und Komponenten . . . . .	15
2	Unterschied zwischen ungelösten und unbekannten Elementen (Bidelman 2013a) .	26
3	Lebenszyklus-Callback Methoden (Bidelman 2013a) . . . . .	28
4	Browser Unterstützung von Web-Components Úrldate: 13.03.2014 <a href="http://jonrimmer.github.io/are-we-componentized-yet/">http://jonrimmer.github.io/are-we-componentized-yet/</a> . . . . .	37
5	Lebenszyklus-Callback Methoden bei Polymer . . . . .	41
6	Browser Unterstützung von Polymer, URLdate: 13.03.2014 <a href="http://www.polymer-project.org/resources/compatibility.html">http://www.polymer-project.org/resources/compatibility.html</a> . . . . .	44
7	Lebenszyklus-Callback Methoden bei Polymer . . . . .	49
8	Vergleich der zur Verfügung gestellten Funktionen von Web-Components und Polymer . . . . .	50

## Code-Beispiele

1	Web-Components Template-Standard . . . . .	19
2	Beispiel-Selektor eines Elements in einem Template, das nicht aktiven DOM ist . .	19
3	Verwendung des Templates 1 auf Seite 19 . . . . .	19
4	Web-Components Decorators - Markup eines Autos ( <i>Decorators - nextgen markup pt.2</i> 2013) . . . . .	21
5	Web-Components Decorators - Markup eines Autos mit Rahmen ( <i>Decorators - nextgen markup pt.2</i> 2013) . . . . .	21
6	Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen) ( <i>Decorators - nextgen markup pt.2</i> 2013) . . . . .	21
7	Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen) inklusive Style ( <i>Decorators - nextgen markup pt.2</i> 2013) . . . . .	22
8	Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen) inklusive Style und Funktionalität ( <i>Decorators - nextgen markup pt.2</i> 2013) . . . .	22
9	Web-Components Decorators - Markup eines Autos mit Decorator ( <i>Decorators - nextgen markup pt.2</i> 2013) . . . . .	23
10	Web-Components Decorators - CSS für die Verwendung von Decorators ( <i>Decorators - nextgen markup pt.2</i> 2013) . . . . .	23
11	Registrierung eines Custom-Elements (Bidelman 2013a). . . . .	24
12	Registrierung eines Custom-Elements mit gegebenen Prototype-Objekt (Bidelman 2013a) . . . . .	24
13	Registrierung eines Custom-Elements mit gegebenen Prototype-Objekt und Namespace (Bidelman 2013a) . . . . .	25
14	Erweiterung von Elementen (Bidelman 2013a) . . . . .	25
15	Verwendung von Type-Extensions (Bidelman 2013a) . . . . .	25
16	Verwendung von Type-Extensions (Bidelman 2013a) . . . . .	25
17	Instanziierung eines Custom-Elements mit einer HTML-Deklaration (Bidelman 2013a) .	26
18	Instanziierung eines Custom-Elements mit Hilfe von JavaScript (Bidelman 2013a) .	26
19	Instanziierung eines Custom-Elements mit Hilfe von JavaScript (Bidelman 2013a) .	26
20	Instanziierung eines type extension custom Elements mit Hilfe von HTML-Deklaration (Bidelman 2013a) . . . . .	27
21	Instanziierung eines type extension custom Elements mit Hilfe von JavaScript (Bidelman 2013a) . . . . .	27
22	Instanziierung eines type extension custom Elements mit Hilfe von JavaScript (Bidelman 2013a) . . . . .	27
23	Beispiel eines Elements <code>&lt;x-foo&gt;</code> mit einer lesbaren Eigenschaft und einer öffentlichen Methode (Bidelman 2013a) . . . . .	27
24	Shadow-Root Beispiel eines Buttons (Dominic Cooney, Dimitri Glazkov 2013) . . .	28
25	Namensschild ohne Shadow-DOM - Ausgangsbasis um Inhalt von Darstellung zu trennen . . . . .	29
26	Darstellung des Markups mit der gewünschten Information ohne Darstellung (Dominic Cooney, Dimitri Glazkov 2013) . . . . .	30

27	Darstellung des Markups mit der gewünschten Information mit Hilfe von einem Template (Dominic Cooney, Dimitri Glazkov 2013) . . . . .	30
28	Hinzufügen des Inhalts eines Templates in eine Shadow-Root (Dominic Cooney, Dimitri Glazkov 2013) . . . . .	31
29	Erweiterung des Code-Beispiels 27 mit dem <content>-Element (Dominic Cooney, Dimitri Glazkov 2013) . . . . .	31
30	Laden eines lokalen HTML-Dokuments (Bidelman 2013b) . . . . .	32
31	Laden eines externen HTML-Dokuments (Bidelman 2013b) . . . . .	33
32	Inkludierung von Bootstrap mit Hilfe von einem HTML-Import (Bidelman 2013b) . . . . .	33
33	Error-Handling bei HTML-Importe (Bidelman 2013b) . . . . .	34
34	Klonen des Inhalts eines HTML-Imports (Bidelman 2013b) . . . . .	34
35	JavaScript im HTML-Import, um Inhalt automatisch im Hauptdokument hinzuzufügen (Bidelman 2013b) . . . . .	35
36	Feature-Detection für Web-Components . . . . .	37
37	Polymer Template Bind (Glazkov u. a.). . . . .	40
38	Polymer Template Repeat (Glazkov u. a.). . . . .	40
39	Polymer Template If (Glazkov u. a.). . . . .	40
40	Polymer Template Ref (Glazkov u. a.). . . . .	41
41	main.html . . . . .	54
42	permChart.html-Aufbau . . . . .	55
43	permChart.html-Verwendung . . . . .	55
44	main2.html-Verwendung von der in permChart.html definierten Komponente . . . . .	55
45	pushMenu.html - Import der Bibliotheken von Codrops . . . . .	56
46	pushMenu.html - Beginn des Templates . . . . .	56
47	pushMenu.html - attachedCallback-Methode des Menüs . . . . .	56
48	Einsatz einer Polymer-Komponente . . . . .	57
49	Registrierung einer Polymer-Komponente . . . . .	58
50	Markup in einer Polymer-Komponente . . . . .	58
51	Markup einer verwendeten Polymer-Komponente . . . . .	58
52	polymer-chart.html . . . . .	58
53	polymer-menu.html . . . . .	60
54	polymer-menu.html . . . . .	60