

Todo list

Remove Color of Hyperref package (see Bakk.tex comments).	1
Check version of Polymer	1
Check version of W3C draft	1
Look up if there is any component collection available	1
Date on frontpage and signature	1
HTML Templates W3C Draft (18.März.2014) https://dvcs.w3.org/hg/webcomponents/raw-file/tip/spec/templates/index.html	12
richtiges Wort?	14
Information von: http://ejohn.org/blog/javascript-micro-templating	15
umschreiben	18
umschreiben	29
small self-written custom element	34
Nesting von Elements bzw. Reuse beispielhaft zeigen	34
advanced shadow dom ein wenig eingehen	34

Bachelorarbeit 2

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

StudentIn Georg Eschbacher, 1110601005
BetreuerIn Hubert Hölzl

Salzburg, am X. X 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.

Datum

Unterschrift

Kurzfassung

Vor- und Zuname: GEORG ESCHBACHER
Institution: FH Salzburg
Studiengang: Bachelor MultiMediaTechnology
Titel der Bachelorarbeit: Komponentenbasierte Softwarearchitektur und Softwareentwicklung: Ein Vergleich von
Begutachter: MSc HUBERT HÖLZL

Deutsche Zusammenfassung ...

... zwischen 150 und 300 Worte ...

Schlagwörter: Folgen, Bachelor, Wissenschaftliches Arbeiten

Abstract

English abstract ...

... between 150 and 300 words ...

Keywords: *a few descriptive keywords*

Inhaltsverzeichnis

1	Einführung	1
1.1	Motivation und Relevanz	1
1.2	Forschungsfrage	3
1.3	Struktur der Arbeit	3
2	Softwarearchitektur und Softwarekomponenten	3
2.1	Klassische Softwarekomponenten	4
2.2	Arten von Softwarekomponenten	5
2.3	Softwarearchitektur	7
2.3.1	Serviceorientierte Softwarearchitektur	9
2.3.2	Komponentenbasierte Softwarearchitektur und komponentenbasierte Softwareentwicklung	10
2.3.3	Unterschied eines Dienstes und einer Komponente	11
2.4	Konklusion	12
3	Web-Components	12
3.1	W3C Web-Components Standard	15
3.1.1	Templates	15
3.1.2	Decorators	16
3.1.3	Custom Elements	20
3.1.4	Shadow DOM	24
3.1.5	HTML Imports	28
3.1.6	Browser Unterstützung	31
3.2	Google Polymer	33
3.3	Konklusion	34
4	Web-Components Praxisbeispiel	34
4.1	Programmierung von Web-Components nach dem W3C Standard	34
4.2	Programmierung von Web-Components mit Hilfe von Google Polymer	34
4.3	Vergleich von den Programmierunterschieden zwischen W3C Standard und Google Polymer	34
5	Konklusion	34
5.1	Ausblick von Web-Components	34
5.2	Offene Fragen hinsichtlich der Entwicklung	34

1 Einführung

Remove Color of Hyperref package (see Bakk.tex comments).

Check version of Polymer

Check version of W3C draft

Look up if there is any component collection available

Date on frontpage and signature

1.1 Motivation und Relevanz

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

„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 zu erläutern, wird ein gängiger Tätigkeitsbereich eines Softwarearchitekten herangezogen: das Erstellen einer Liste von Komponenten, die in die gesamte 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 zum Ganzen machen (Szyperski, Gruntz und Murer 2002).

Komponentenentwicklung steht für die Herstellung von Komponenten, die eine spezielle Funktion in einer Softwarearchitektur übernehmen. Sämtliche Komponenten sollten immer für sich gekapselt und unabhängig von einander sein. Mehrere Komponenten werden mit Hilfe eines Verbindungsverfahrens zusammengeführt beziehungsweise verwendet.

Softwarekomponente haben ihren Ursprung im „Unterprogramm“. Ein „Unterprogramm“, oder auch „Subroutine“ genannt, ist ein Teil einer Software, die von anderen Programmen beziehungsweise Programmteilen aufgerufen werden kann. Eine Subroutine gilt als Ursprung der ersten Einheit für die Softwarewiederverwendung (Wheeler 1952). 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“ unser Denken: der Fokus kann auf neue Algorithmen und komplexere Themen gelegt werden. Des Weiteren entwickelten sich auch die Programmiersprachen weiter. Anspruchsvolle Subroutines waren ununterscheidbar von primitiven, atomaren Anweisungen (Szyperski, Gruntz und Murer 2002).

Komponentenbasierte Softwareentwicklung entwickelte sich in einer ununterbrochenen Linie von diesen frühen Anfängen. Moderne Komponenten, von denen die meisten webbasierte Services sind, sind viel größer und viel anspruchsvoller, als früher. Des Weiteren haben die neuen Komponenten eine dienstorientierte Architektur, was zu höherer Komplexität betreffend der Interaktionsmechanismen führt, als die damaligen Subroutinen (Andresen 2003).

In der gleichen Weise, wie die frühen Subroutinen die Programmierer vom Nachdenken über spezifische Details befreit haben, verschiebt sich durch komponentenbasierte Softwareentwicklung der Schwerpunkt von direkter Programmiersoftware zu komponierten Softwaresystemen, sprich komponentenbasierten Systemen. Auf der Grundlage, dass der Schwerpunkt auf der Integration von

Komponenten liegt, basiert die Annahme, dass es genügend Gemeinsamkeiten in großen Softwaresystemen gibt, um die Entwicklung von wiederverwendbaren Komponenten zu rechtfertigen und um dafür diese Gemeinsamkeiten ausnutzen zu können. Heute werden Komponente gesucht, die eine große Sammlung von Funktionen bereitstellen. Beispielsweise wird aus unternehmerischer Sicht nicht nach einem simplen Adressbuch, um die Daten beziehungsweise Kommunikation seiner Kunden zu sichern, sondern eine CRM-System¹ gesucht. Darüberhinaus sollte dieses System auch flexibel sein, d.h. es sollte mit anderen Komponenten erweiterbar sein. Die Auswirkungen, wenn die Kontrolle über die Zerlegung in die jeweiligen Komponenten vorhanden ist beziehungsweise wenn diese Kontrolle nicht vorhanden ist, werden in der Bachelorarbeit näher besprochen (Andresen 2003).

Komponentenbasierte Entwicklung dient der Verwaltung von Komplexität in einem System. Sie versucht diese Verwaltung zu erreichen, indem die Programmiererin und der Programmierer sich vollständig auf die Implementierung der Komponente fokussieren. Das Verknüpfen beziehungsweise Kombinieren von Komponenten sollte nicht mehr Aufgabe des Komponentenentwickler sein. Um diese Aufgabe zu lösen wird ein spezielles Framework oder externe Struktur der Komponente verwendet. Vorteile durch dieses Programmierparadigma sind somit einerseits die Zeitersparnisse und andererseits die erhöhte Qualität der Komponenten. Dadurch, dass bei der komponentenbasierten Entwicklung die Wiederverwendbarkeit von Komponenten im Vordergrund steht, gibt es verschiedene Anwendungsszenarien, die automatisch als Testszenarien dienen (Andresen 2003).

Im Zeitalter von Internet, Intranet und Extranet sind viele verschiedene Systeme und Komponenten miteinander zu verzahnen. Web-basierte Lösungen sollen auf Informationen eines Hostrechners zugreifen können, um z.B. mittels eines Unternehmensübergreifenden Extranets diversen Zwischenhändlern transparente Einblicke in die Lagerbestände eines Unternehmens zu ermöglichen. ERP²- und CRM-Systeme sollen eingebunden werden, um die Betreuung und den Service für Kunden zu optimieren. Aus diesen Gründen ist eine erweiterbare und flexible Architektur notwendig, die eine schnelle Reaktion auf neue Anforderungen ermöglicht.

Um diesen Anforderungen gerecht zu werden, gab es in den letzten Jahren eine unüberschaubare Anzahl an Frameworks beziehungsweise Bibliotheken, die die Entwickler unterstützten. Eine neue Technologie, die derzeit vom W3C versucht wird zu standardisieren, gehört zu den interessantesten neuen Webtechniken. Diese Technologie versucht eine Vielzahl von Funktionen der populärsten JavaScript-Komponentenframeworks aufzunehmen und nativ in den Browser zu portieren. Dadurch wird es möglich sein eigene Komponenten und Applikationen entwickeln zu können, welche die Vorteile dieser Technologie, die zuvor nur begrenzt durch Einbindung zahlreicher Bibliotheken beziehungsweise unter Verwendung zahlreicher Frameworks, nutzen.

Unter „Web-Components“ versteht man ein Komponentenmodell, das 2013 in einem Arbeitsentwurf des W3C veröffentlicht wurde. Es besteht aus fünf Teilen:

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, um so zuverlässigere Benutzerschnittstellen der Elemente garantieren zu können.

Imports definieren, wie Templates, Decorators und Custom Elements verpackt und als eine Resource geladen werden können.

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

2. Enterprise Resource Planning

Dieses Komponentenmodell hat das Potenzial, die Entwicklung von Web-Applikationen enorm zu vereinfachen und zu beschleunigen. Mit Hilfe von „Custom Elements“ und „Imports“ lassen sich eigene, komplexe HTML-Elemente selbst bauen oder von anderen entwickelte Elemente in der eigenen Applikation oder Website nutzen. Beispielsweise könnte ein eigenes HTML-Element von einer einfachen Überschrift mit fest definiertem Aussehen, über einen Videoplayer, bis hin zu einer kompletten Applikation, alles sein. Vieles, was heute über Javascript-Bibliotheken abgewickelt wird, könnte künftig in Form einzelner Webkomponenten umgesetzt werden. Das verringert Abhängigkeiten und sorgt für mehr Flexibilität. Bis die dafür notwendigen Webstandards aber verabschiedet, in Browsern umgesetzt und diese bei ausreichend Nutzern installiert sind, wird aber noch einige Zeit vergehen. Google hat daher mit Polymer eine Bibliothek entwickelt, die die Nutzung von Webkomponenten schon heute ermöglicht und dazu je nach den im Browser vorhandenen Funktionen die fehlenden Teile ergänzt.

Die persönliche Motivation beziehungsweise Relevanz zu diesem Thema ist durch das praxisnahe Projekt gegeben. Hierbei wird der Fokus auf die Programmierung von zwei wiederverwertbaren Frontend-Oberflächen gelegt. Zum einen wird eine Diagramm-Komponente und zum anderen ein komplexes Menü entwickelt, dass für folgende Projekte weiterverwendet werden kann. Die Hauptanforderung ist die Kompatibilität zu so vielen Browsern wie möglich zu gewährleisten. Des Weiteren soll durch die einmalige Programmierung dieser Komponente und deren darauffolgende Wiederverwendung den Wartungsaufwand möglichst gering zu halten. Somit wird in dieser Arbeit geklärt, inwiefern die Entwicklung von Web-Components ohne Unterstützungen wie beispielsweise das Polymer Projekt bereits möglich ist. Des Weiteren 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 zur Zeit standardisierten Web-Components Technologie analysiert, sondern auch an Hand des von Google zur Verfügung gestellten Polyfills.

1.2 Forschungsfrage

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

1.3 Struktur der Arbeit

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. Des Weiteren 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 einer Abbildung 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, als auch die komponentenbasierte in Verbindung mit komponentenbasierter

Softwareentwicklung erklärt. Als Abschluss dieses Kapitel 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:“

- 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“ (Szyperski, Gruntz und Murer 2002). 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 es bedeutet, dass eine Komponente nicht partiell, sondern nur als Ganzes ausgeliefert wird. Ein Beispiel für diesen Kontext wäre, dass ein Dritter keinen Zugriff auf die involvierten Komponenten einer Software hat.

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 erfordert und was sie bietet.

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. Wenn Komponenten einen feststellbaren Zustand haben dürften, wäre es nicht möglich, zwei „gleiche“ Komponenten mit den gleichen Eigenschaften zu haben. Eine mögliche Ausnahme von dieser Regel sind Attribute, die nicht zur Funktionalität der Komponente beitragen. Ein Beispiel in diesem Kontext wäre die Seriennummer für die Buchhaltung. Dieser spezifische Ausschluss ermöglicht einen zulässigen technischen Einsatz eines Zustands, der kritisch für die Leistung sein könnte. Beispielsweise dafür sei der Cache. Eine Komponente kann einen Zustand mit der Absicht etwas zu cachen verwenden. Ein Cache ist ein Speicher, auf den man ohne Konsequenzen verzichten kann, jedoch möglicherweise reduzierte Leistung in Anspruch nimmt.

Wenn man eine Komponente mit Hilfe dieser Definition umsetzt, gilt sie als vollständig wiederverwertbar. Des Weiteren ergeben sich aus dieser Definition zwei Sichtweisen auf Komponenten, zum einen die Sichtweise des Verwenders der Komponente (siehe Unterabbildung 1a auf Seite 5) und zum anderen die Sichtweise des Entwicklers der Komponente (siehe Unterabbildung 1b auf Seite 5)). Der Verwender der Komponente kann keine Aussagen darüber treffen, auf welcher Basis die verwendete Komponente entwickelt wurde. Folglich kann die Implementierung als „Black-Box“

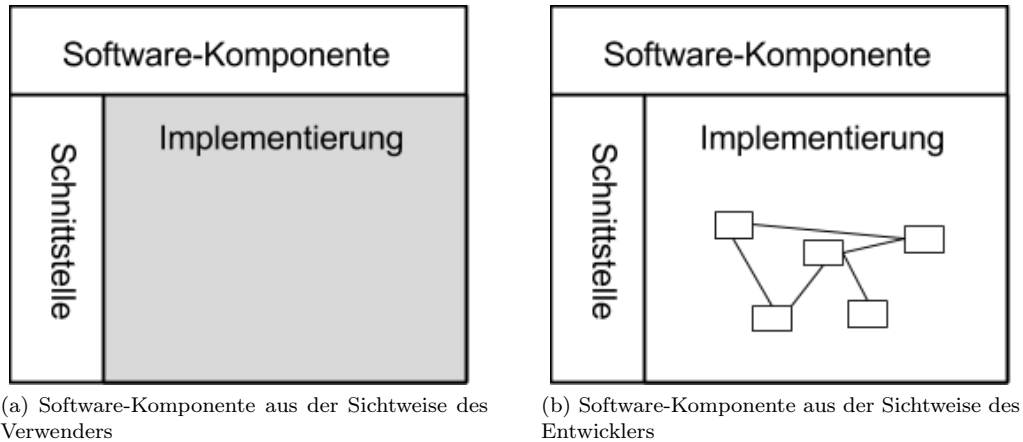


Abbildung 1: Software-Komponente aus unterschiedlichen Sichtweisen

für den Verwender gesehen werden. Der Entwickler hingegen hat vollständiges Wissen über den Aufbau und das Verhalten der Komponente.

In vielen aktuellen Ansätzen sind Komponenten eine schwerwiegende 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 in keinsten Weise den Gebrauch von Zuständen von Objekten fördert beziehungsweise zurückstufte (Szyperski, Gruntz und Murer 2002).

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 2 auf Seite 6 veranschaulicht dieses Architekturparadigma (Andresen 2003).

Auf Grundlage der in Abbildung 2 auf Seite 6 veranschaulichten Architektur können Komponenten diversen Schichten zugeteilt werden:

1. Komponenten der Präsentations-Schicht

Diese Komponenten stellen eine nach außen sichtbare Benutzerschnittstelle dar. Ein Beispiel dieser Schnittstelle ist eine GUI³-Komponente (Button, Menü, Slider, und vieles mehr...).

3. Graphical User Interface

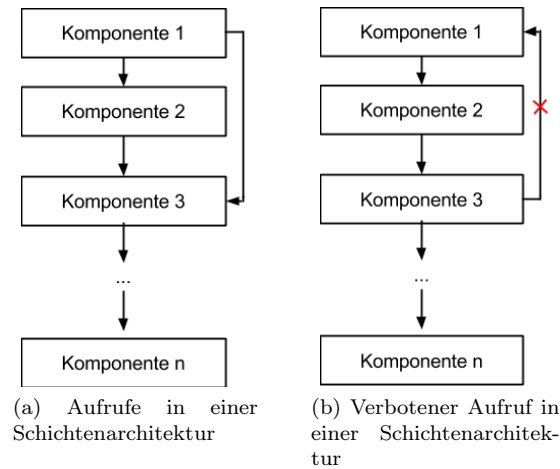


Abbildung 2: Beispiel einer Schichten-Architektur

2. Komponenten der Controlling-Schicht

Diese verarbeiten komplexe Ablauflogik und dienen als Vermittler zwischen Komponenten der Business- und Präsentations-Schicht. Beispielsweise 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, die die eigentliche Problemstellung löst. Oftmals bezeichnet man hier die sogenannte Entity-Komponente. Sie dient der Persistierung von Daten beziehungsweise bildet Entitäten auf Komponenten ab (Firma, Produkt, Kunden).

4. Komponenten der Integrations-Schicht

Sie dient 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 liefert den Datenzugriff für Komponenten der oben genannten Schichten. Bei Datenzugriffs-Komponenten werden die Besonderheiten einer Datenbank berücksichtigt.

Komponente 1 in Abbildung 2a auf Seite 6 würde hinsichtlich der genannten Einteilung die Präsentations-Schicht darstellen und somit 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 (Andresen 2003).

Es ist zu erwähnen, dass es mehrere Kopplungen von Schichten-Architekturen gibt. Die in Abbildung 2a auf Seite 6 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. Wenn man bedenkt, dass Architektur in verschiedenen Bereichen ein Thema ist und unterschiedliche Aspekte bei der Erstellung eines Systems umfasst, wird deutlich, warum eine allgemeingültige Definition schwer fällt (Vogel 2009). Zu Beginn wird die klassische Architektur als Ausgangspunkt verwendet. Eine mögliche Definition der klassischen Architektur bietet das „American Heritage Dictionary“⁴:

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

Wenn man diese Definition zugrunde legt, ist Architektur sowohl eine Kunst als auch eine Wissenschaft, 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. Generell entstehen Architekturen auf Grund von Anforderungen, wie beispielsweise der Wunsch nach einer Behausung und unter Verwendung von vorhandenen Mitteln wie zum Beispiel Werkzeugen. 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 (Vogel 2009).

Zum Begriff „Architektur“ in der IT existieren im Gegensatz zur klassischen Architektur unzählige Definitionen⁵. Daran zeigt sich, dass es eine Herausforderung darstellt, eine Definition zu finden, die allgemein anerkannt wird (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 (Vogel 2009).

Symptome mangelhafter Softwarearchitektur

Fatalerweise zeigen sich die Folgen einer mangelhaften Architektur in der IT nicht selten erst mit erheblicher Verzögerung, das heißt, ernste Probleme treten eventuell erst auf, wenn ein System zum ersten Mal produktiv eingesetzt wird oder wenn es bereits im Einsatz ist und für

4. Siehe American Heritage Online-Dictionary

5. 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

neue Anforderungen angepasst werden muss. 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 (Vogel 2009):

- Fehlender Gesamtüberblick
- Komplexität ufert 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
- Performanz ist miserabel
- Architektur-Dokumentation ist unzureichend
- Funktionalität beziehungsweise Quelltext ist redundant
- Systembausteine besitzen zahlreiche unnötige Abhängigkeiten untereinander
- Entwicklungszyklen sind sehr lang

Folgen mangelhafter Softwarearchitektur

Diese sind folgende (Vogel 2009):

- 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 ("MonsterKlassen").
- Fachklassen, deren Implementierungsdetails im gesamten System bekannt sind.

Vorteile von Architektur

Unabhängig davon, welche Art von System entwickelt wird, legt eine Architektur ausgehend von den Anforderungen an das System immer die Fundamente und damit die tragenden Säulen, jedoch nicht die Details für das zu entwickelnde System fest ((Buschmann 1996) nach (Vogel 2009)). Architektur handelt also von den Fundamenten, 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 Festlegungen und Kriterien, nach denen das System in Bausteine aufgeteilt wird?

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

Ein wichtiges Charakteristikum von Architektur ist, dass sie Komplexität überschaubar und handhabbar macht, indem sie nur die wesentlichen Aspekte eines Systems zeigt, ohne zu sehr in die Details zu gehen, und es so ermöglicht, 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 (Fowler 2005). Beispiele hierfür sind die Festlegung, wie Systembausteine ihre Daten untereinander austauschen oder die Auswahl der Komponentenplattform⁶. Derartige architekturelevante 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 (Bredemeyer und Malan 2004).

2.3.1 Serviceorientierte Softwarearchitektur

In diesem Subkapitel wird Begriff „serviceorientierte Softwarearchitektur“ mit „SOA“ abgekürzt.

Mit Hilfe von SOAs können verteilte Systeme, wo die Systemkomponente eigenständige Dienste darstellen und das System selbst auf geographisch verteilten Rechnern läuft, entwickelt werden. 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 (Sommerville 2011).

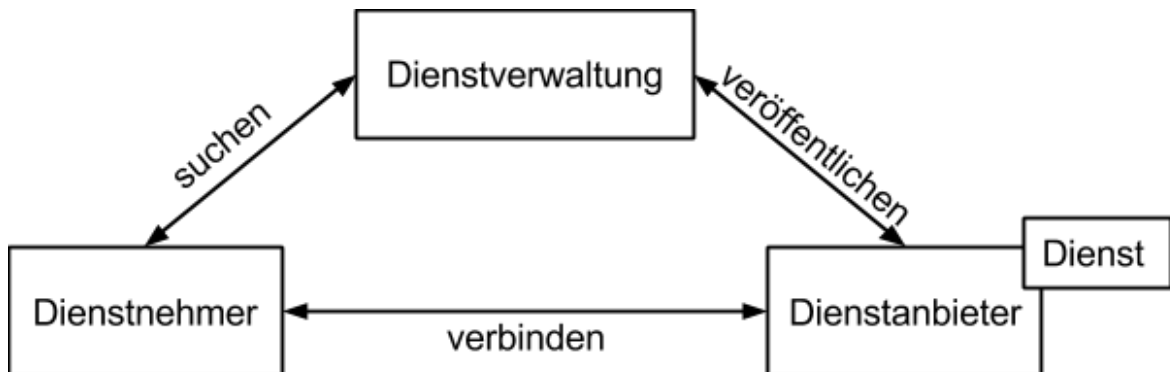


Abbildung 3: Serviceorientierte Architektur

Eine Vielzahl von Unternehmen, darunter auch Microsoft, haben solche Dienstverwaltungen anfangs des 21. Jahrhunderts eingerichtet, die sich bereits als redundant herausgestellt haben und bereits von der Suchmaschinenteknologie ersetzt wurden.

Wenn eine Applikation mit Hilfe von Diensten gebaut wird, fördert dies zugleich den Nutzen, dass andere Unternehmen auch auf diese Dienste zugreifen können. Dies kann natürlich auch in anderer Hinsicht gesehen werden, dass in seiner eigenen Applikation Dienste eines anderen Unternehmens verwendet werden können. Dies hat den Vorteil, dass Systeme, die einen hohen Datenaustausch zwischen zwei Unternehmen haben und über deren geographischen Grenzen hinausgehen, automatisiert werden können. Beispielsweise für dieses Konzept wäre eine Lieferkette zwischen zwei Unternehmen, wo ein Unternehmen Artikel über einen Dienst des anderen Unternehmens kauft.

6. Beispiele für Komponentenplattformen sind JEE, .NET, Adobe AIR und viele mehr...

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.

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 4). Das Ziel der komponentenbasierten Softwareentwicklung ist die Steigerung der Produktivität, Qualität und „time-to market“⁷. Diese Steigerung soll mit dem Einsatz von einerseits Standardkomponenten und andererseits Produktionsautomatisierungen erfolgen. Ein wichtiger Paradigmenwechsel bei dieser Entwicklungsmethode ist, dass man Systeme auf Basis von Standardkomponenten bauen soll, anstatt das „Rad immer wieder neu zu erfinden“. 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 (Sommerville 2011).

Folglich muss die Definition der komponentenbasierten Softwarearchitektur hinsichtlich des bereits definierten Begriffs der Softwarearchitektur (siehe Kapitel 2.3 auf Seite 7) 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 (Andresen 2003).

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

- 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 5).
- Komponentenstandards, die die Integration von Komponenten erleichtern. Diese Standards werden an Hand eines Komponentenmodells⁸ 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.

7. Unter „time-to market“ versteht man die Dauer von der Produktentwicklung bis zur Platzierung des Produkts am Markt

8. Beispiele für Komponentenmodelle sind Enterprise Java Beans, Cross Platform Component Object Model, Distributed Component Object Model, und viele mehr.

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

- Interaktionen zwischen Komponenten und deren Umfeld sind geregelt
- 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.

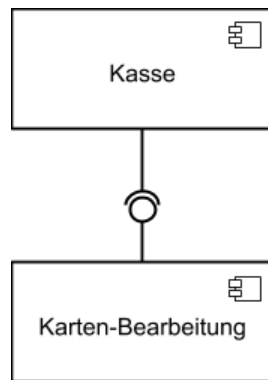


Abbildung 4: Beispiel für die Interaktion zwischen zwei Komponenten, ausgedrückt in UML

2.3.3 Unterschied eines Dienstes und einer Komponente

Dienste und Komponenten haben offensichtlich viele Gemeinsamkeiten. Beide sind wiederverwendbare Elemente, jedoch gibt es einige, wichtige Unterschiede zwischen Diensten und Komponenten (Sommerville 2011).

- Dienste können von jedem Dienstanbieter innerhalb oder außerhalb einer Organisation offeriert werden. Wie bereits vorher erläutert, ist es möglich, eine Applikation ausschließlich auf externen Diensten basierend zu erstellen.
- Der Dienstanbieter veröffentlicht allgemeine Informationen über den Dienst, sodass es autorisierten Benutzern möglich ist, diese zu nutzen. Es gibt keinerlei Verhandlungen zwischen dem Dienstanbieter und dem Dienstnehmer damit der Dienst benutzt werden kann.
- Applikationen können die Bindung von Diensten bis zur Auslieferung dieser verzögern.
- Opportunistische Entwicklungen von neuen Diensten ist möglich. Das heißt, dass ein Dienstanbieter einen neuen Dienst erkennen kann, indem bereits vorhandene Dienste auf neue, innovative Weise verknüpft werden.
- Dienstnehmer können für Dienste gemäß ihrer Verwendung bezahlen, anstatt einer Provision. Folglich ist es möglich anstatt eine Komponente zu kaufen, die nur selten genutzt wird, einen Dienst zu benutzen, der nur dann bezahlt wird, wenn er in Verwendung ist.
- Applikationen können um ein vielfaches kleiner gemacht werden, wenn man beispielsweise das „Exception handling“ als externen Dienst implementiert. Dies ist vor allem dann ein Vorteil, wenn die Applikation in andere Geräte eingebettet wird.

- Applikationen können reaktiv sein und sich an ihre Umgebung durch Bindung an verschiedene Dienste (je nach Umgebung) anpassen.

Dienste sind eine natürliche 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. Das bedeutet, dass sie immer in der gleichen Weise arbeiten sollen, unabhängig von ihrer Einsatzumgebung. Die Schnittstelle eines Dienstes ist eine „bietet“-Schnittstelle, die den Zugriff auf die Dienstfunktionalität ermöglicht. Dienste streben einen unabhängigen Einsatz in unterschiedlichen Kontexten an. Daher haben sie nicht eine „erfordert“-Schnittstelle, wie Komponenten sie haben. Komponenten sind meist immer auf zumindest eine „Grundkomponente“, wie beispielsweise die Komponente des Core-Systems, angewiesen.

Ein Dienst definiert, was er von einem anderen Dienst braucht. Dafür werden die Anforderungen des Dienstes 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. Dieser Dienst analysiert daraufhin die Antwort auf die gewünschten Informationen. Im Gegensatz zu Komponenten benutzen Dienste keine „remote procedure“ oder Methodenaufrufe um die Funktionalität des anderen Dienst erreichen zu können (Sommerville 2011).

2.4 Konklusion

3 Web-Components

HTML Templates W3C Draft (18.März.2014) <https://dvcs.w3.org/hg/webcomponents/raw-file/tip/spec/templates/index.html>

Um Web-Components besser verstehen zu können, wird in diesem Kapitel zu Beginn eine kurze Übersicht über die Geschichte von Web-Bibliotheken gezeigt.

- 2005** Veröffentlichung von Dojo Toolkit⁹ mit der innovativen Idee von Widgets. Mit ein paar Zeilen Code konnten Entwickler komplexe Elemente, wie beispielsweise einen Graph oder eine Dialog-Box in ihrer Website hinzufügen.
- 2006** jQuery¹⁰ stellt Entwicklern die Funktion zur Verfügung Plugins zu entwickeln, die später wiederverwendet werden können.
- 2008** Veröffentlichung von jQuery UI¹¹, was vordefinierte Widgets und Effekte mit sich bringt.
- 2009** Erstveröffentlichung von AngularJS¹², ein Framework mit Direktiven.
- 2011** Erstveröffentlichung von React¹³. 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
- 2013** Veröffentlichung des Entwurfs von Web-Components, jedoch mit schlechter Browser Unterstützung

9. Mehr Information zu Dojo Toolkit unter <http://dojotoolkit.org/>

10. Mehr Information zu jQuery unter <http://jquery.com/>

11. Mehr Information zu jQuery UI unter <http://jqueryui.com/>

12. Mehr Information zu AngularJS unter <http://angularjs.org/>

13. Mehr Information zu Facebook React unter <http://facebook.github.io/react/>

Mit der Veröffentlichung von Dojo Toolkit sagen Entwickler die Vorteile von wiederverwendbaren Modulen. Wenn man zurzeit Plugins auf einer Website erwähnt, denken die meisten Entwickler von jQuery Plugins, da sie beinahe überall Verwendung finden und ein großes Spektrum von Funktionen bieten. Mit den Veröffentlichungen von AngularJS und React wurde gezeigt, in welche Richtung sich Web-Anwendungen bewegen. Sie zeigen, dass es nicht nur um visuelle Elemente geht, sondern auch um Elemente, die eine komplexe Logik besitzen.

Ähnlich zu HTML5 ist Web-Components ein Sammelbegriff für mehrere Features:

Shadow DOM (ausführliche Erklärung siehe Kapitel 3.1.4 auf Seite 24) erlaubt es das DOM und CSS zu kapseln

HTML Templates (ausführliche Erklärung siehe Kapitel 3.1.1 auf Seite 15) sind ein Weg, um den DOM zu klonen und somit den Klon wiederzuverwenden

Custom Elements (ausführliche Erklärung siehe Kapitel 3.1.3 auf Seite 20) können einerseits neue Elemente definieren, oder bereits bestehende Elemente erweitern. Dies bedeutet, dass ein Entwickler beispielsweise den HTML `<input>`-Tag dahingehend erweitern kann, dass dieser nur das Format von Kreditkartennummern unterstützt. Ein Beispiel für die Definition eines neuen Elements wäre ein Element, dass sämtliche Felder, die für die Bezahlung mit einer Kreditkarte notwendig sind, bereitstellt.

HTML Imports (ausführliche Erklärung siehe Kapitel 3.1.5 auf Seite 28) sind dazu da, um externe HTML-Dateien in die bestehende Website zu integrieren, ohne dabei den Code kopieren zu müssen. Sie können beispielsweise dazu verwendet werden, um Web-Components in eine Website zu integrieren.

Decorators (ausführliche Erklärung siehe Kapitel 3.1.2 auf Seite 16) sind Elemente, die nach dem „Decorator-pattern“ benannt sind. Durch dieses Pattern ist es möglich Elemente um zusätzliche Funktionalitäten zur Laufzeit erweitern zu können.

Obwohl „Web-Components“ für viele Entwickler noch kein Begriff ist, wird es bereits vom Browser automatisch verwendet. Beispiele dafür sind der Datepicker oder das `<video>`-Element. Abbildung 5 auf Seite 13 zeigt die Datepicker-Komponente und Abbildung 6 auf Seite 14 zeigt den dazugehörigen Source-Code. Dieser Code zeigt, dass sämtliche Kontrollbuttons des Datepickers vor dem Entwickler „versteckt“, also im Shadow DOM liegen.

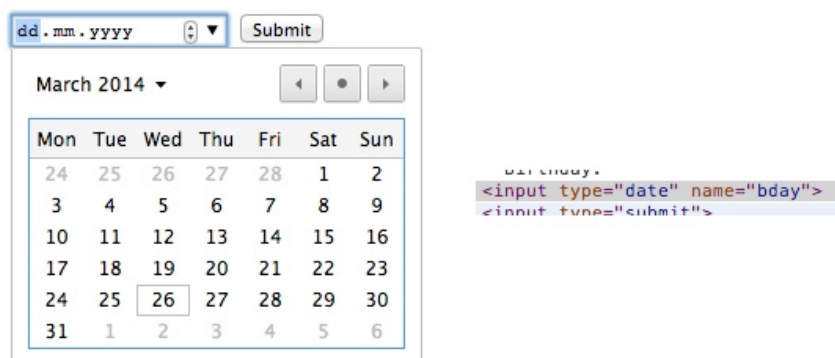


Abbildung 5: Beispiel von Web-Components im Browser an Hand von dem Datepicker

Warum Web-Components?

Javascript Widgets und Plugins sind fragmentiert, weil sie auf diversen unterschiedlichen Bibliotheken und Frameworks basieren, die möglicherweise nicht miteinander funktionieren. Web-Components versuchen einen gewissen Standard in Widgets und Plugins zu bringen. Das Problem



Abbildung 6: Beispiel von Web-Components im Browser an Hand von dem Datepicker

der nicht miteinander funktionierenden Plugins versucht Web-Components mit Kapselung zu lösen. 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. Eine Bibliothek könnte bereits eine Komponente darstellen, die eine gewisse Funktionalität bereitstellt.

richtiges
Wort?

Unterstützung von Web-Components

Zur Zeit ist die Hauptproblem von Web-Components die mangelhafte Browserunterstützung. Kein einziger Browser unterstützt diesen Standard zu 100%. Es gibt bereits mehrere Möglichkeiten beziehungsweise Polyfills¹⁴, um dennoch Web-Components nutzen zu können. Beispiele hierfür sind:

- Polyfill-Webcomponents¹⁵
- Polymer-Project¹⁶
- X-tags¹⁷

Obwohl es mehrere Polyfills bezüglich Web-Components gibt, ist es egal, auf welcher Basis man seine Web-Components programmiert, denn durch die Standardisierung ist die Interkompatibilität der einzelnen Komponenten gegeben. Diese Arbeit beschränkt sich hauptsächlich auf die Entwicklung von Web-Components mit Hilfe des Standards beziehungsweise der Polyfill-Bibliothek Polymer. Durch die Benutzung von dem Polyfill Polymer funktionieren Web-Component in allen „evergreen“-Browsern¹⁸ und Internet-Explorer 10 und neuer. Des Weiteren funktionieren sie dadurch auf mobilen Endgeräten, wo iOS6+, Chrome Mobile, Firefox Mobile, oder Android 4.4 oder höher vorhanden ist. Auch mit Hilfe von Polyfill bieten sowohl der Internet-Explorer 9 und niedriger, als auch Android-Browser 4.3 oder niedriger keine Unterstützung für Web-Components. Dies bedeutet, dass Web-Components zur Zeit noch für die Verwendung im Web bereit sind, außer man hat als Zielgruppe ausschließlich eine Plattform, die unterstützt wird.

Web-Components Alternativen Die folgenden Alternativen von Web-Components benutzen ähnliche Pattern um die gewünschte DOM-Abstrahierung zu erreichen.

14. Ein Polyfill ist ein Browser-Fallback, um Funktionen, die in modernen Browsern verfügbar sind, auch in alten Browsern verfügbar zu machen.

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

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

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

18. Ein „Evergreen“-Browser ist ein Web-Browser, der sich automatisch beim Start updatet.

React benutzt seinen eigenen „Virtual DOM“ und versucht in keinster Hinsicht Web-Components zu simulieren. Folglich ist die Browser-Unterstützung von React besser, als jene der Web-Components. Ab Internet-Explorer 8 werden sämtliche Browser vollständig unterstützt. Zur Zeit wird diese Technologie in Facebooks und Instagrams Kommentarsystemen eingesetzt.

AngularJS besitzt diverse Interferenzen zu Web-Components, jedoch versucht auch diese Technologie nicht Web-Components zu simulieren, um bessere Browser-Unterstützung zu bieten (Internet Explorer 8+). Die genaue Unterscheidung bezüglich AngularJS-Direktiven und Web-Components werden in Kapitel 3.2 auf Seite 33 erklärt.

3.1 W3C Web-Components Standard

3.1.1 Templates

Das folgende Kapitel basiert ausschließlich auf der Einführung zu Web-Components aus dem W3C (Glazkov und Cooney, Juni 2013). Laut W3C sind Templates

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

Somit sind Templates eine Methode um inaktive DOM-Subtrees im HTML zu deklarieren und manipulieren, um so sämtliche identische Dokumentfragmente mit identischem Inhalt zu instanzieren.

In Web-Applikationen muss man oft den gleichen Subtree von Elementen öfters benutzen, mit den passenden Inhalt füllen und es zum Maintree hinzufügen. Man hat zum Beispiel eine Liste von Artikel, die man mit mehreren `<i>`-Tags in das Dokument einfügen will. Des Weiteren kann jeder `<i>`-Tag weitere Elemente, wie beispielsweise einen Link, ein Bild, einen Paragraphen, etc., enthalten. Bis jetzt bot HTML keine native Möglichkeit an, eine solche Aufgabenstellung zu lösen. Mehrere Beispiele, wie Entwickler diese Aufgabe lösten, sind:

1. Versteckte Elemente

Dies galt unter den Entwicklern als die einfachste, aber auch als die ineffizienteste Methode. Man gab sein Markup irgendwo in das DOM (meist mit der CSS-Eigenschaft `display: none;` am Parent-Element des Markups) und wann immer es gebraucht wurde, wurde es geklont, mit diversen Daten gefüllt und schlussendlich in das DOM eingefügt. Dies beinhaltet diverse Nachteile. Sämtliche Ressourcen, die im Markup verwendet wurden, sind bei Seitenaufruf geladen worden. Die Gesamtperformanz des Browsers wurde durch die vielzählige DOM-Traversierung beeinträchtigt.

2. String-Templates

Man versuchte die Probleme der Methode mit „versteckten Elementen“ zu beseitigen, indem man in einem `<script>`-Tag ein Template mit einem String definiert. Dadurch, dass diese Methode hinsichtlich Laufzeit-Parsen von `innerHTML` geht, können XSS-Attacks ermöglicht werden. Folglich wird ein Beispiel gezeigt, wo ein Template mit Hilfe eines Strings in einem `<script>`-Tag definiert wird:

```
1 <script type="text/html" id="test_Showcase">
2 </script>
```

Listing 1: String-Template

Folgend wird an Hand eines Beispiels, wo eine Liste von Autos benutzt wird, der neue Standard zu Templates erläutert werden:

Information
von:
<http://ejohn.org/micro-templating>

```

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

```

Listing 2: Web-Components Template-Standard

Ein Template, wie das aus der Listing 2 auf Seite 16, kann sowohl im `<head>`- als auch im `<body>`-Tag definiert werden. Das Template, inklusive Subtree, ist inaktiv. Dies bedeutet, dass wenn sich ein ``-Tag mit einer validen Quelle in diesem Template befinden würde, würde der Browser dieses Bild nicht laden. Des Weiteren ist es nicht möglich via JavaScript ein Element des Templates zu selektieren.

```

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

```

Listing 3: 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 16 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>`, ``-Tags etc. nicht verwendet werden.

```

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);

```

Listing 4: Verwendung des Templates 2 auf Seite 16

Listing 4 auf Seite 16 basiert auf dem in Listing 2 auf Seite 16 definierten Template. Zuerst wird sich in Zeile 1 der Listing 4 das vorher definierte Template in die Variable `template` geholt. Daraufhin wird der gesamte Knoten in Zeile 4 mit Hilfe einer `deep-copy` geklont und des Weiteren mit Daten befüllt. Damit das mit Daten befüllte Listenelement auch sichtbar wird, wird es in Zeile 8 in das aktive DOM eingefügt.

3.1.2 Decorators

Das folgende Kapitel basiert ausschließlich auf der Einführung zu Web-Components aus dem W3C (Glazkov und Cooney, Juni 2013).

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

Grundsätzlich gehört das Decorator-Pattern zu den Struktur-Pattern der Softwareentwicklung. Das Pattern ist eine flexible Alternative zur Unterklassenbildung, um eine Klasse zur Laufzeit um zusätzliche Funktionalitäten erweitern zu können. Hinsichtlich der Verwendung des Patterns und dem Dekorator-Konzept bringt dies kleine Vorteile mit sich. Sie bestehen beispielsweise darin, dass mehrere Dekorierer hintereinandergeschaltet werden können. Weiters ist die Laufzeitbeeinflussung der Dekorierer als Vorteil zu sehen, da sie ausgetauscht werden können. Folglich kann Funktionalität eines individuellen Objekts zur Laufzeit erweitert werden, ohne dabei die Funktionalität von anderen Objekten der selben Klasse zu ändern.

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. 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>

```

Listing 5: Web-Components Decorators - Markup eines Autos

Unter der Annahme, dass man das Bild des Autos um die Funktionalität erweitern will, sodass man es sichtbar machen, unsichtbar machen beziehungsweise schließen kann, würde man das bereits vorhandene Markup aus Listing 5 auf Seite 17 wie folgt erweitern:

```

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>

```

Listing 6: Web-Components Decorators - Markup eines Autos mit Rahmen

Listing 6 auf Seite 17 würde somit das Markup eines Autos beinhalten, wobei es zwei Buttons gibt: einen zum Umschalten zwischen sichtbar und unsichtbar und einen um das Bild komplett zu löschen. Wenn man es mit den bisherigen standardisierten Möglichkeiten umsetzt, wird der Quellcode schnell aufgebläht.

Decorators würden in diesem Beispiel bereits helfen. Man könnte definieren, dass spezielle Elemente im DOM mit mehr Markup, Style und zusätzlicher Funktionalität versehen werden. Essentiell hierbei ist, dass es möglich ist die Basisfunktionalität nur für eine gewünschte Menge an Elementen erweitern zu können. Wenn man die erweiterte Funktionalität des Auto-Beispiels mit Hilfe von Decorators umsetzen würde, würde es 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>

```

Listing 7: Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen)

Dieses Beispiel bedarf näherer Erklärung. Decorators werden grundsätzlich mit `<template>`-Elementen eingesetzt (mehr zu Templates in Kapitel 3.1.1 auf Seite 15). Des Weiteren wird in Zeile 8 der Listing 7 ein `<content>`-Element verwendet. Dies ist zwingend notwendig, da in dieser Stelle der Inhalt des zu dekorierenden Elements eingefügt wird. Auch ist zu erwähnen, dass in diesem Beispiel nur `ids` verwendet werden, was nicht zu empfehlen ist. Jedoch sollte es zur Visualisierung dienen, 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 Listing 8 auf Seite 18 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>

```

Listing 8: 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 etwas erläutert werden. Bei Decorators gibt es keine „normalen“ Events, wie man es gewöhnt ist. Das Hinzufügen beziehungsweise Entfernen eines Decorators würde das Event, wenn es auf ein Element gebunden war, löschen. Anstatt normalen Events hat man mit Hilfe von Decorators die Möglichkeit einen Event-Controller zu erstellen, um mittels diesen Events verwalten zu können.

umschreiben

```

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>

```

Listing 9: Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen) inklusive Style und Funktionalität

Der in Listing 9 auf Seite 9 gezeigte Decorator kann somit verwendet werden, um die gewünschte Funktionalität (das Bild des Autos soll sichtbar, unsichtbar beziehungsweise gelöscht werden können) bereitstellen zu können, ohne dabei für jedes ``-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
35         Spanish automaker SEAT. It is SEAT's best-selling car and perhaps the
36         most popular model in the company's range.</p>
37 </li>

```

Listing 10: Web-Components Decorators - Markup eines Autos mit Decorator

```

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

```

Listing 11: Web-Components Decorators - CSS für die Verwendung von Decorators

Unter der Verwendung des in Listing 11 auf Seite 19 gezeigten CSS-Attributs wird der Decorator für das in Listing 10 auf Seite 19 verwendet.

3.1.3 Custom Elements

Custom Elemente sind ein neue Typen von bisher bestehenden DOM-Elementen. Sie können vom Autor beliebig definiert werden und müssen nur wenige Vorschriften einhalten. Im Gegensatz zu Decorators (siehe Kapitel 3.1.2 auf Seite 16), welche zustandslos und kurzlebig sind, können Custom Elements den Zustand kapseln und eine Schnittstelle zur Verwendung bereitstellen. Tabelle 1 zeigt die Schlüsselunterschiede zwischen den beiden Konzepten.

	Decorators	Custom Elements
Lebensdauer	kurzlebig, wenn ein passender CSS-Selector vorhanden ist	stabil, angepasst an die Lebensdauer des Elements
dynamisches hinzufügen, entfernen	Ja, auf Basis des CSS-Selectors	Nein; einmalig (bei der Erstellung des Elements)
In einem Skript erreichbar	Nein; transparent zum DOM und kein Hinzufügen einer Schnittstelle möglich	Ja, im DOM erreichbar und eventuell vorhandene Schnittstelle
Zustand	zustandsloser Ansatz	zustandsorientiertes DOM-Objekt
Verhalten	Simulation durch Änderung des Decorators	Veränderung durch Scripts und Events

Tabelle 1: Schlüsselunterschiede zwischen Decorators und Custom Elements

Web Components würden ohne die Funktionalitäten von benutzerdefinierten Elementen nicht existieren:

1. neue HTML- beziehungsweise DOM-Elemente definieren
2. Elemente erstellen, die die Funktionalität von bereits bestehenden Elementen erweitern
3. Logische Bündelung von benutzerdefinierten Funktionalitäten in nur einen Tag.
4. Schnittstellen von bereits vorhandenen DOM-Elementen erweitern.

Registrierung neuer Elemente

Benutzerdefinierte Elemente können mit Hilfe von `document.registerElement()` erstellt werden:

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

Listing 12: Registrierung eines Custom-Elements

Das erste Argument der `document.registerElement()` Methode ist der Name des neuen Elements. Dieser Name muss ein 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 Prototype-Objekt des Elements definieren würde. Dies wäre der Platz um seinen eigenen Element öffentliche Methoden oder Eigenschaften zu geben. Standardmäßig erben sämtliche benutzerdefinierte Elemente von `HTMLElement`. Somit wäre Listing 12 auf Seite 20 das Gleiche wie Listing 13 auf Seite 21.

```

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

```

Listing 13: Registrierung eines Custom-Elements mit gegebenem Prototype-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 Constructor returned, den man zur Instanziierung neuer Elemente verwenden kann.

Standardmäßig wird der Konstruktor im globalen Window-Objekt abgelegt. Falls dies nicht erwünscht ist, kann man auch einen Namespace dafür festlegen. Listing 14 auf Seite 21 veranschaulicht dies.

```

1 (
2   var myApp = {}
3   myApp.myCar = document.registerElement('my-car', {
4     prototype: Object.create(HTMLElement.prototype)
5   });
6 )
7 document.body.appendChild(new myApp.myCar());

```

Listing 14: Registrierung eines Custom-Elements mit gegebenem Prototype-Objekt und Namespace

Bereits vorhandene Elemente erweitern

Durch benutzerdefinierte Elemente wird es möglich bereits vorhandene oder selbsterstellte Elemente zu erweitern. Um ein Element erweitern zu können, muss man der `registerElement()`-Methode den Namen und das Prototype-Objekt des Elements angeben, das man erweitern möchte. Wenn beispielsweise `<element-a>` `<element-b>` erweitern möchte, so muss `<element-a>` bei der Registrierung das Prototype-Objekt von `<element-b>` angegeben werden. In Listing 15 auf Seite 21 wird die Funktionalität eines `<button>` erweitert.

```

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

```

Listing 15: Erweiterung von Elementen

Wenn ein benutzerdefiniertes Element von einem nativen Element erbt, nennt man es auch „typerweitertes, benutzerdefiniertes Element“ (type extension custom element). Sie erben von einer speziellen Version des `HTMLElement`. Sozusagen „ist element A ein element B“. Verwendet wird eine Typerweiterung wie folgt:

```

1 <button is="mega-button">

```

Listing 16: Verwendung einer Typerweiterung

Bereits vorhandene, benutzerdefinierte Elemente erweitern

Um ein `<my-car-extended>`-Element zu erstellen, dass vom `<my-car>`-Element erbt, muss man bei der Registrierung des erweiterten Elements das Prototype-Objekt des gewünschten „Basis“-Element angeben. Listing 17 auf Seite 21 verdeutlicht diest.

```

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

```

Listing 17: Verwendung einer Typerweiterung

Wie Elemente erweitert werden

„The HTMLUnknownElement interface must be used for HTML elements that are not defined by this specification.“

Dies bedeutet, dass sämtliche nicht valide deklarierten Elemente funktionieren, jedoch nicht vom standardmäßigen `HTMLElement` erben, sondern von `HTMLUnknownElement`. Wenn Browser die Methode `document.registerElement()` nicht bereitstellen, werden folglich auch Custom Elemente nicht unterstützt. Wenn dennoch Elemente mit Hilfe der genannten Methode versucht werden zu erstellen und keine Unterstützung des Browsers vorliegt, wird auch das erstellte Element von `HTMLUnknownElement` erben.

Ungelöste Elemente

Da benutzerdefinierte Elemente in einem Skript registriert werden, können sie dennoch deklariert beziehungsweise erstellt werden. Beispielsweise kann man `<my-car>` deklarieren, obwohl `document.registerElement('my-car')` erst viel später aufgerufen wird.

Bevor Elemente zu ihrer gewünschten Definition upgegradet werden, werden sie als ungelöste Elemente bezeichnet. Dies sind HTML-Elemente, die einen validen benutzerdefinierten Namen haben, jedoch noch nicht registriert sind. Tabelle 2 auf Seite 22 zeigt den Unterschied zwischen ungelösten Elementen und unbekannten Elementen.

Name	Erbt von	Beispiel
Ungelöstes Element	HTMLElement	<code><my-car></code> , <code><my-wheel></code> , <code><my-element></code>
Unbekanntes Element	HTMLUnknownElement	<code><myCar></code> , <code><my_wheel></code>

Tabelle 2: Unterschied zwischen ungelösten und unbekannten Elementen

Instanziierung benutzerdefinierter Elemente

Die geläufigen Techniken, um ein Element zu erstellen, gelten auch für benutzerdefinierte Elemente. Wie bei allen anderen Standardelementen kann man benutzerdefinierte in HTML deklarieren, oder im DOM mit Hilfe von JavaScript erstellen.

1. HTML-Deklaration

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

Listing 18: Instanziierung eines benutzerdefinierten 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 });
```

Listing 19: Instanziierung eines benutzerdefinierten Elements mit Hilfe von JavaScript

3. Erstellung mit Hilfe des `new`-Operator

```

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

```

Listing 20: Instanziierung eines benutzerdefinierten Elements mit Hilfe von JavaScript

Instanziierung von Typweiterungs-Elementen Instanziierung von typerweiterungs benutzerdefinierten Elementen ist auffallend ähnlich zu der Instanziierung von benutzerdefinierten Elementen.

1. HTML-Deklaration

```

1  <button is="mega-button">

```

Listing 21: Instanziierung eines typerweiterten, benutzerdefinierten 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

```

Listing 22: Instanziierung eines typerweiterten, benutzerdefinierten Elements mit Hilfe von JavaScript

3. Erstellung mit Hilfe des new-Operator

```

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

```

Listing 23: Instanziierung eines typerweiterten, benutzerdefinierten Elements mit Hilfe von JavaScript

Hinzufügen von Eigenschaften und Methoden zu einem Element

Benutzerdefinierte Elemente werden erst durch maßgeschneiderte Funktionalität des Elements mächtig. Man kann eine öffentliche Schnittstelle mit Hilfe von Eigenschaften und Methoden für sein Element erstellen. Folgend wird das Element `<x-foo>` registriert, welche 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 });

```

Listing 24: Beispiel eines Elements `<x-foo>` mit einer lesbaren Eigenschaft und einer öffentlichen Methode

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

Lebenszyklus-Callback Methoden

Elemente können spezielle Methoden definieren, die zu einer speziellen Zeit ihres Lebenszyklus 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 24) genauer erläutert wird:

Callback-Name	Aufgerufen, wenn
createdCallback	eine Instanz des Elements erstellt wurde
attachedCallback	eine Instanz in das Dokument eingefügt wurde
detachedCallback	eine Instanz vom Dokument entfernt wurde
attributeChangedCallback(attrName, oldVal, newVal)	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 und wenn das Element entfernt wird die dafür erstellte Verbindung im `detachedCallback()` schließen. Weiters können diese Methoden zur Konfiguration von Event-Listener verwendet werden.

3.1.4 Shadow DOM

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, dass einer Shadow-Root zugeordnet ist, wird auch als „Shadow-Host“ bezeichnet. Anstatt den Inhalt eines Shadow-Hosts zu rendern, wird der des Shadow-Root gerendered.

```

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

Listing 25: Shadow-Root Beispiel eines Buttons

Listing 25 rendered zuerst das in Abbildung 8a auf Seite 24 gezeigte Ergebnis. Danach wird mit Hilfe von JavaScript und ShadowDOM das Element, wie in Abbildung 8b auf Seite 24 zu sehen ist, verändert.

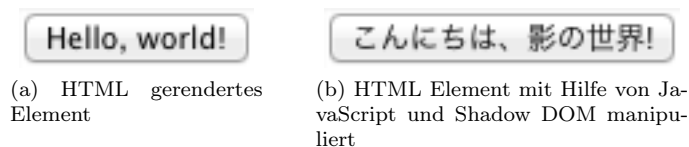


Abbildung 8: Beispiel einer Shadow-Root Node

Darüber hinaus wird das Resultat, wenn man den `<button>` nach seinem Inhalt mit Hilfe der `textContent` Eigenschaft abfragt nicht

1 @、影世界@!、

zurückgegeben, sondern "Hello, world!", weil die DOM-Unterstruktur unter der Shadow-Root vollständig gekapselt ist.

Es ist zu erwähnen, dass dies ein sehr schlechtes Beispiel ist, da sämtlicher Inhalt im Shadow-DOM nicht für Suchmaschinen, Erweiterungen, Screen readers, etc. erreichbar ist. Shadow-DOM ist nur für semantisch bedeutungsloses Markup, das benötigt wird um eine Webkomponente zu erstellen.

Trennung von Inhalt und Darstellung

Listing 26 auf Seite 25 wird als Ausgangsbasis dieses Beispiels genommen. Abbildung 9 auf Seite 26 zeigt diese Grundbasis, um mit darauffolgenden Schritten sämtlichen Inhalt von der Darstellung zu trennen.

```

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>

```

Listing 26: Namensschild ohne Shadow-DOM - Ausgangsbasis um Inhalt von Darstellung zu trennen

Dadurch, dass dem DOM-Baum Kapselung fehlt, ist die gesamte Struktur des Namensschildes im Dokument sichtbar. Wenn beispielsweise andere Elemente auf der Webseite dieselben Klassennamen verwenden würden, würde das Aussehen von dem nicht zum gewünschten Ergebnis führen.

1. Verstecken von Darstellungsdetails

Semantisch gibt es nur zwei wichtige Informationen bei diesem Beispiel:

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

Daraus wird das Markup erstellt, das semantisch näher bei der gewünschten Information ist.



Abbildung 9: Ausgangsbeispiel von der Trennung von Darstellung und Inhalt bei Shadow-DOM

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

Listing 27: Darstellung des Markups mit der gewünschten Information ohne Darstellung

Des Weiteren wird sämtlicher Code, der der Darstellung dient, in ein `<template>`-Element gepackt.

```
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>
```

Listing 28: Darstellung des Markups mit der gewünschten Information mit Hilfe von einem Template

Zu diesem Zeitpunkt wird „Bob“ das einzige sein, das gerendered wird. Da sämtliche Code, der der Darstellung dient, in ein `<template>` gepackt wurde, muss man dies beispielsweise mit JavaScript hinzufügen. In Listing 29 auf Seite 27 wird zuerst eine Shadow-Root am Element

`<div id=nameTag></div>` erstellt. Danach wird nach dem Template gesucht und der Inhalt 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>
```

Listing 29: Hinzufügen des Inhalts eines Templates in eine Shadow-Root

Da man nun eine Shadow-Root gesetzt und mit Markup versehen hat wird das Namensschild wieder gerenderd. Wenn das Element inspiziert wird, sieht man nur die gewünschte Information, ohne Darstellungselementen.

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 Listing 28 und 29 werden sämtliche Darstellungsdetails versteckt, jedoch wurde der Inhalt noch nicht mit der Darstellung getrennt. Wenn man beispielsweise den Namen des Namensschildes austauschen müsste, müsste man es an zwei Stellen amchen. 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 dem Namen der Person.

Als Autor einer Komponente entscheidet man, wie die Komposition des erstellten Elements funktionieren soll, mit Hilfe des `<content>`-Elements. Dieses Element erstellt einen Einsatzpunkt in der Darstellung und sucht Inhalte aus dem Shadow-Host, die an dieser Stelle dargestellt 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>

```

Listing 30: Erweiterung der Listing 28 mit dem `<content>`-Element

In Listing 30 auf Seite 27 wird das Namensschild mit dem vom Shadow-Host projizierten Inhalt in das `<content>`-Element gerendered. Dies vereinfacht die Struktur des Dokuments, da der Name nur noch an einer Stelle vorhanden ist. Wenn man nun den Namen aktualisieren möchte, kann man das mit folgender Methode tun: `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.

3.1.5 HTML Imports

Es gibt eine Vielzahl von Möglichkeiten, wie man diverse Ressourcen lädt. 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 sie zu laden. HTML hingegen besitzt keinen standardisierten Weg. Zurzeit gibt es folgende Weg um HTML zu laden:

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

Jede dieser Methoden bringt seine Vor- und Nachteile mit sich und keine dieser Methoden ist eine standardisierte Weise, wie man externes HTML lädt.

HTML-Imports bieten einen standardisierten Weg, wie man ein HTML-Dokument in ein anderes HTML-Dokument lädt. Ein HTML-Import ist des Weiteren nicht auf Markup limitiert, sondern es kann auch CSS, JavaScript, etc. beinhalten. Listing 31 auf Seite 28 zeigt, wie man ein lokales HTML-Dokument lädt.

```

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

```

Listing 31: Laden eines lokalen HTML-Dokuments

Die URL eines Imports nennt man auch „Import-Stelle“. Um Inhalt von einer anderen Domain zu laden, muss die Import-Stelle CORS¹⁹ aktiviert sein. Listing 32 auf Seite 28 zeigt, wie man ein externes HTML-Dokument lädt.

```

1 <head>
2   <!-- Resources on other origins must be CORS-enabled. -->
3   <link rel="import" href="http://example.com/car.html">
4 </head>

```

Listing 32: Laden eines externen HTML-Dokuments

19. Cross-Origin Resource Sharing

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

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 und mächtige Eigenschaft von HTML-Importe. Durch die Funktionen, die HTML-Importe bereitstellen, ist es möglich, eine Web-Applikation in einzelne, logische Segmente aufzuteilen und dem Endbenutzer dennoch nur eine URL geben zu müssen.

Ein sehr gutes Beispiel für einen sinnvollen Einsatz von HTML-Importe wäre Bootstrap²⁰. Bootstrap besteht aus individuellen CSS- und JavaScript-Dateien, sowie Fonts. Darüber hinaus benötigt es für die bereitgestellten Plugins JQuery. Listing ?? auf Seite ?? zeigt, wie man Bootstrap auf mehrere Dokumente aufteilen und laden könnte.

```

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 ...

```

Listing 33: Inkludierung von Bootstrap mit Hilfe von einem HTML-Import

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 ist. Listing 34 auf Seite 29 zeigt ein Beispiel von Error-Handling bei HTML-Importe.

```

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>

```

Listing 34: Error-Handling bei HTML-Importe

Ein wichtiger Punkt bei dem in Listing 34 gezeigten Beispiel ist, dass die Funktionen vor dem Import definiert wurden. Der Browser versucht einen HTML-Import dann 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.

Benutzung des Inhalts eines Imports

Wenn man einen HTML-Import benutzt, 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 es lädt, um es dann für weitere Verwendung bereit zu haben. Wenn man den Inhalt eines Imports erreichen will, da es beispielsweise

umschreiben

20. Mehr Information zu Dojo Toolkit unter <http://getbootstrap.com/>

gewisse `<template>`-Elemente beinhaltet, muss man dafür JavaScript verwenden. Listing ??uf Seite ?? holt sich den Inhalt eines HTML-Imports. Die importierte Datei (`warnings.html`) beinhaltet diverses gestaltete Markup, was in der Hauptseite (`main.html`) verwendet werden sollte. 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 gerendered.

```

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>

```

Listing 35: Klonen des Inhalts eines HTML-Imports

JavaScript in einem zu importierendem Dokument

Importe befinden sich nicht im Hauptdokument. Sie können als Satelliten zum Hauptdokument gesehen werden. Im zu importierenden Dokument kann der DOM vom Hauptdokument und sein eigenes DOM erreicht werden. Listing 36 auf Seite 30 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>

```

Listing 36: 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. Weiters ist es nicht erforderlich `<script>`-Blöcke im Hauptdokument hinzuzufügen, da sie ausgeführt werden.
- Importe blocken den Browser nicht beim Parsen des Hauptdokuments. Dennoch werden Skripte in den Importen der Reihe nach verarbeitet.

HTML-Importe im Zusammenhang mit Templates

Ein sehr großer Vorteil, wenn diese beiden Unterpunkte von Web-Components zusammen verwendet werden ist, dass Skripte innerhalb eines Templates nicht beim Laden des zu importierenden Dokuments ausgeführt werden, sondern erst dann, wenn das Template aktiv wird, sprich dem DOM des Hauptdokuments hinzugefügt wird.

HTML-Importe im Zusammenhang mit benutzerdefinierten Elementen

Wenn man diese beiden Technologien vereint, muss sich der Benutzer, der beispielsweise ein fremdes, benutzerdefiniertes Element mit Hilfe eines HTML-Imports lädt, nicht um die Registrierung des Elements kümmern, da es bereits im zu importierenden Dokument gemacht werden kann.

Abhängigkeits-Management und Sub-Importe

Sub-Importe sind vor allem dann vom Vorteil, wenn eine Komponente wiederverwendbar oder erweitert werden soll. Beispielsweise kann man JQuery als eine Komponente ansehen und sie als HTML-Import definieren. Wenn man mehrere, benutzerdefinierte Elemente mit Hilfe von JQuery entwickelt und sie anderen zur Verfügung stellt, werden die Abhängigkeiten für sämtliche Elemente automatisch geladen. Nimmt man an, das man drei benutzerdefinierte Elemente lädt, wobei jedes einzelne Element an sich JQuery als Abhängigkeit mittels HTML-Import geladen hat, wird JQuery trotzdem nur ein einziges Mal im Hauptdokument geladen.

3.1.6 Browser Unterstützung

Die nachstehende Tabelle 4 zeigt die Unterstützung der Browser bezüglich Web-Components. Sie wurde zuletzt am 13 März 2014 aktualisiert und ist somit aktuell. Sämtliche Browser aus der Tabelle besitzen die aktuellste Version. Grün bedeutet, dass die Technologie in dem jeweiligen Browser als stabil gewertet wird und verwendet werden kann. Gelb bedeutet, dass es vom jeweiligen Browser in Arbeit ist, noch Bugs auftreten, oder mittels einer Flag erreichbar ist. Rot bedeutet, dass es keine Information bezüglich der Technologie in dem jeweiligen Browser gibt.

Chrome

Was Web-Components betrifft ist Googles Chrome der goldene Standard. Sie haben sich an die Spitze gesetzt, was die Umsetzung der Spezifikation angeht. Drei Technologien von Web-Components werden bereits als stabil in Chrome bezeichnet. Auch HTML-Importe sind bereits vorhanden, jedoch um diese benutzen zu können, ist es erforderlich, eine Flag im Browser zu aktivieren.

Opera

Dadurch, dass Opera seit einiger Zeit auf die Basis von Chromium (Blink) gewechselt hat,

wird von Opera der gleiche Weg wie von Google erwartet. Zur Zeit gibt es keine kennbaren Unterschiede was die Implementierung der Spezifikation angeht bezüglich den beiden Browsern.

Firefox

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

Safari

Obwohl viele Funktionen von Web-Components in Webkit implementiert wurden, wurden sie nie in Safari verwendet beziehungsweise zur Verfügung gestellt. Mit der Abzweigung des Chromium-Port von Safari wurde begonnen sämtliche Web-Components Funktionen aus dem Browser zu entfernen.

Internet Explorer

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 scheint keine der Schnittstellen für Web-Components zu beinhalten.





















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

Tabelle 4: Browser Unterstützung von Web-Components

Listing 37 auf Seite 32 zeigt, wie man die Funktionen beziehungsweise einzelnen Technologien auf Verfügbarkeit im Browser testen kann. 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("
16     Templates are not supported!")
17   supportsCustomElements()? console.log("Custom elements are supported!") : console
18     .error("Custom elements are not supported!")
19   supportsImports()? console.log("HTML-Imports are supported!") : console.error("
20     HTML-Imports are not supported!")
21   supportsShadowDom()? console.log("Shadow-DOMs are supported!") : console.error("
22     Shadow-DOMs are not supported!")
23 })();

```

Listing 37: Feature-Detection für Web-Components

3.2 Google Polymer

In allgemeinen Worten abgefasst ist Google-Polymer ein Framework, das sämtliche Funktionen von Web-Components zur Verfügung stellt. Polymer verfolgt den Ansatz, dass alles was entwickelt wird, ist eine Web-Komponente und diese entwickelt sich mit dem Web mit. Abbildung 10 auf Seite 33 zeigt die Architektur von Polymer.

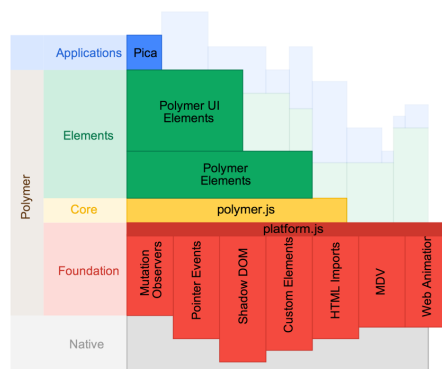


Abbildung 10: Polymers Architektur

Die rote Schicht visualisiert die Polyfills, die Polymer zur Verfügung stellt. 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 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 gebraucht 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“, „öffentliche Eigenschaften“, etc.

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

Ein Beispiel für ein vordefiniertes Element der Google-Polymer Bibliothek wäre das `<polymer-ajax>`-Element. Es erscheint in erster Linie als nicht sehr nützlich, jedoch versucht es, einen Standard

für Entwickler bereitzustellen, um Ajax-requests zu erstellen beziehungsweise abzuwickeln. Dieses Element ist ähnlich zu folgender Funktion: `$.ajax()`²¹. Der Unterschied zwischen den beiden Möglichkeiten, einen Ajax-Request abzuwickeln, ist, dass die `$.ajax()`-Methode Abhängigkeiten besitzt, wohingegen die `<polymer-ajax>`-Methode vollkommen unabhängig ist.

small self-written custom element

Nesting von Elements bzw. Reuse beispielhaft zeigen

3.3 Konklusion

4 Web-Components Praxisbeispiel

advanced shadow dom ein wenig eingehen

4.1 Programmierung von Web-Components nach dem W3C Standard

4.2 Programmierung von Web-Components mit Hilfe von Google Polymer

4.3 Vergleich von den Programmierunterschieden zwischen W3C Standard und Google Polymer

5 Konklusion

5.1 Ausblick von Web-Components

5.2 Offene Fragen hinsichtlich der Entwicklung

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

Abbildungsverzeichnis

1	Software-Komponente aus unterschiedlichen Sichtweisen	5
2	Beispiel einer Schichten-Architektur	6
3	Serviceorientierte Architektur	9
4	Beispiel für die Interaktion zwischen zwei Komponenten, ausgedrückt in UML . .	11
5	Beispiel von Web-Components im Browser an Hand von dem Datepicker, Urldate: 04.2014 https://s3.amazonaws.com/infinum.web.production/repository_items/files/000/000/238/original/datepick	13
6	Beispiel von Web-Components im Browser an Hand von dem Datepicker, Urldate: 04.2014 https://s3.amazonaws.com/infinum.web.production/repository_items/files/000/000/236/original/datepick	14
7	Visualisierung des DOM eines inaktiven Templates, Urldate: 04.2014 http://www.prevent-default.com/wp-content/uploads/2013/04/document-fragment-300x132.png	16
8	Beispiel einer Shadow-Root Node	24
9	Ausgangsbeispiel von der Trennung von Darstellung und Inhalt bei Shadow-DOM	26
10	Polymers Architektur, Urldate: 04.2014 http://i.stack.imgur.com/Ksn6s.png	33

Listings

1	String-Template	15
2	Web-Components Template-Standard	16
3	Beispiel-Selektor eines Elements in einem Template, das nicht aktiven DOM ist . .	16
4	Verwendung des Templates 2 auf Seite 16	16
5	Web-Components Decorators - Markup eines Autos	17
6	Web-Components Decorators - Markup eines Autos mit Rahmen	17
7	Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen)	17
8	Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen) inklusive Style	18
9	Web-Components Decorators - Markup der zusätzlichen Funktionalität (Rahmen) inklusive Style und Funktionalität	18
10	Web-Components Decorators - Markup eines Autos mit Decorator	19
11	Web-Components Decorators - CSS für die Verwendung von Decorators	19
12	Registrierung eines Custom-Elements	20
13	Registrierung eines Custom-Elements mit gegebenen Prototype-Objekt	21
14	Registrierung eines Custom-Elements mit gegebenen Prototype-Objekt und Name- space	21
15	Erweiterung von Elementen	21
16	Verwendung einer Typerweiterung	21
17	Verwendung einer Typerweiterung	21
18	Instanziierung eines benutzerdefinierten Elements mit Hilfe von HTML-Deklaration	22
19	Instanziierung eines benutzerdefinierten Elements mit Hilfe von JavaScript	22
20	Instanziierung eines benutzerdefinierten Elements mit Hilfe von JavaScript	23
21	Instanziierung eines typerweiterten, benutzerdefinierten Elements mit Hilfe von HTML-Deklaration	23
22	Instanziierung eines typerweiterten, benutzerdefinierten Elements mit Hilfe von Ja- vaScript	23
23	Instanziierung eines typerweiterten, benutzerdefinierten Elements mit Hilfe von Ja- vaScript	23
24	Beispiel eines Elements <code><x-foo></code> mit einer lesbaren Eigenschaft und einer öffentli- chen Methode	23
25	Shadow-Root Beispiel eines Buttons	24
26	Namensschild ohne Shadow-DOM - Ausgangsbasis um Inhalt von Darstellung zu trennen	25
27	Darstellung des Markups mit der gewünschten Information ohne Darstellung . . .	26
28	Darstellung des Markups mit der gewünschten Information mit Hilfe von einem Template	26
29	Hinzufügen des Inhalts eines Templates in eine Shadow-Root	27
30	Erweiterung der Listing 28 mit dem <code><content></code> -Element	27

31	Laden eines lokalen HTML-Dokuments	28
32	Laden eines externen HTML-Dokuments	28
33	Inkludierung von Bootstrap mit Hilfe von einem HTML-Import	29
34	Error-Handling bei HTML-Importe	29
35	Klonen des Inhalts eines HTML-Imports	30
36	JavaScript im HTML-Import, um Inhalt automatisch im Hauptdokument hinzuzu- fügen	30
37	Feature-Detection für Web-Components	32

Tabellenverzeichnis

1	Schlüsselunterschiede zwischen Decorators und Custom Elements	20
2	Unterschied zwischen ungelösten und unbekannten Elementen	22
3	Lebenszyklus-Callback Methoden	24
4	Browser Unterstützung von Web-Components (Stand 13.03.2014) http://jonrimmer.github.io/are-we-components	32

Literatur

- Andresen, Andreas. 2003. *Komponentenbasierte softwareentwicklung: mit mda, uml und xml*. München und Wien: Hanser. ISBN: 3446222820.
- 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.
- Fowler, Martin. 2005. *The new methodology*. <http://martinfowler.com/articles/newMethodology.html>. [Online, 30.03.2014].
- Glazkov, Dimitri, und Dominic Cooney. Juni 2013. *Introduction to web components*. <http://www.w3.org/TR/components-intro/>. [Online, 12.04.2014].
- 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.
- 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.