Todo list

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 angegeben 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:	Vorname NACHNAME
Institution:	FH Salzburg
Studiengang:	Bachelor MultiMediaTechnology
Titel der Bachelorarbeit:	Die Bachelorarbeit und ihre Folgen
Begutachter:	Titel Vorname Nachname

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

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 Verbindungsverfahren 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-System[1] 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. ERP[2]- 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 Ressource 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

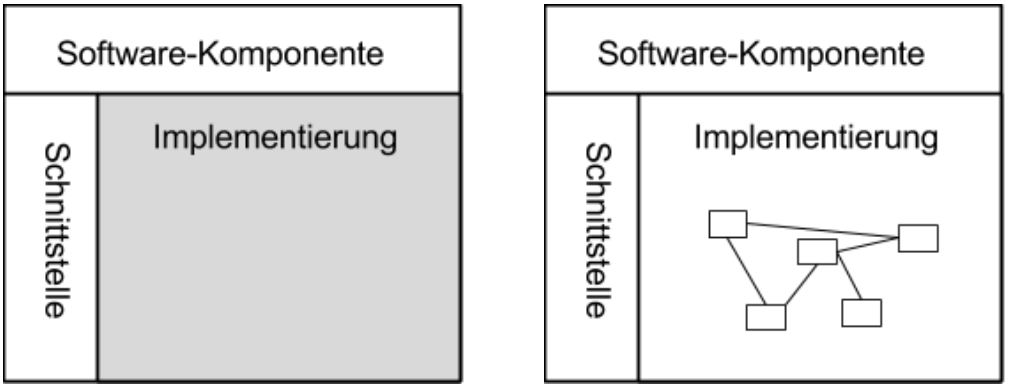
Dies ist die Definition einer Komponente von Clemens Szyperski aus dem Buch „Component software: Beyond object-oriented programming“(Szyperski, Gruntz und Murer 2002). Diese Definition bedarf hinsichtlich dieser Arbeit weiterer Erläuterung:

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 die verwendete Komponente entwickelt wurde. Folglich kann die Implementierung als „Black-Box“



(a) Software-Komponente aus der Sichtweise des Verwenders

(b) Software-Komponente aus der Sichtweise des Entwicklers

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 keinster weise den Gebrauch von Zuständen von Objekten fördert beziehungsweise zurückstuft (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

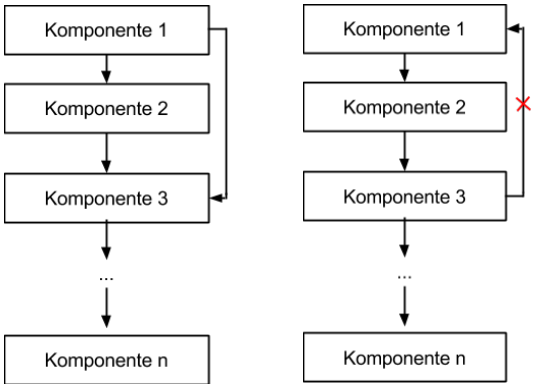
Verschiedene Arten von Komponenten können entsprechend ihren Aufgabenbereichen klassifiziert werden. Eine übersichtliche Art der Zuordnung von Aufgabenbereichen zu Komponenten kann auf der Basis der Trennung von Zuständigkeiten erfolgen, zum Beispiel auf der Basis einer Schichten-Architektur. Eine Schichten-Architektur 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 GUI[3]-Komponente (Button, Menü, Slider, und vieles mehr...).

3. Graphical User Interface



(a) Aufrufe in einer Schichtenarchitektur

(b) Verbotener Aufruf in einer Schichtenarchitektur

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.Beispielhaft 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 Perstistierung 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 Definitionen[5]. 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 verschieene Definitionen für den Begriff „Softwararchitektur“ 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 der 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 Komponentenplattform[6]. Derartige architekturrelevante Festlegungen wirken sich systemweit aus im Unterschied zu architekturirrelevanten Festlegungen (wie beispielsweise eine bestimmte Implementierung einer Funktion), die nur lokale Auswirkungen auf ein System haben (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).

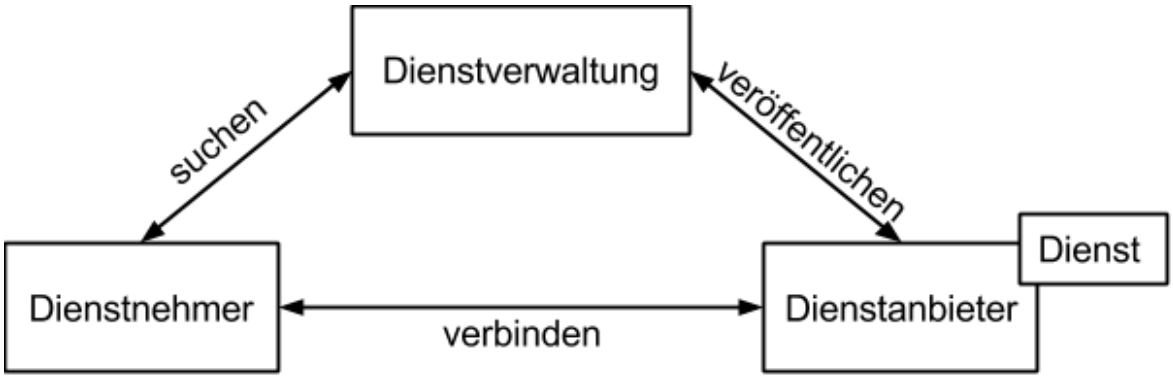


Abbildung 3: Serviceorientierte Architektur

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 Suchmaschinentechnologie 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 Systeme, 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 Integrierung 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 Komponentenmodells[8] 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 Softwareentwicklug 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.

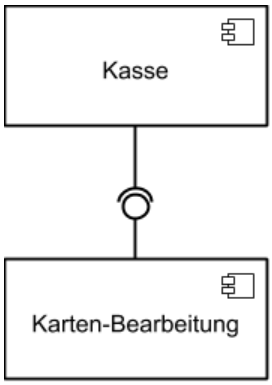


Abbildung 4: Beispiel für die Interaktion zwischen zwei Komponenten, ausgedrückt in UML

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 Toolkit[9] 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 jQuery[10] stellt Entwicklern die Funktion zur Verfügung Plugins zu entwickeln, die später wiederverwendet werden können.

2008 Veröffentlichung von jQuery UI[11], was vordefinierte Widgets und Effekte mit sich bringt.

2009 Erstveröffentlichung von AngularJS[12], ein Framework mit Direktiven.

2011 Erstveröffentlichung von React[13]. 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 3.2.4 auf Seite 17) erlaubt es das DOM und CSS zu kapseln

HTML Templates (ausführliche Erklärung siehe Kapitel 3.2.1 auf Seite 15) sind ein Weg, um den DOM zu klonen und somit den Klon wiederzuverwenden

Custom Elements (ausführliche Erklärung siehe Kapitel 3.2.3 auf Seite 17) 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 3.2.5 auf Seite 17) 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 3.2.2 auf Seite 17) sind Elemente, die nach dem „Decorator programming 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>`-Elemen. 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.

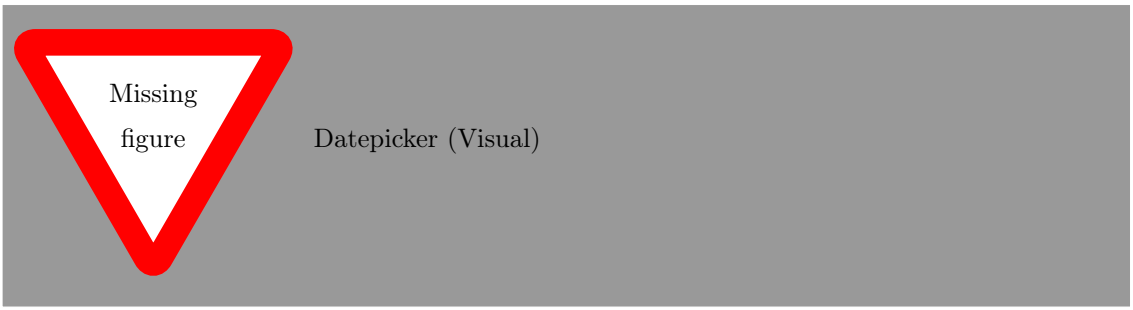


Abbildung 5: Beispiel einer Schichtenarchitektur

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 der nicht miteinander funktionierenden Plugins versucht Web-Components mit Kapselung zu lösen.

Abbildung 6: Beispiel einer Schichtenarchitektur

Durch die Lösung dieses Problems ist die Wiederverwendbarkeit von Komponenten garantiert, da es sämtliche Interferenzenzwischen 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 Polyfills[14], um dennoch Web-Components nutzen zu können. Beispiele hierfür sind:

- Polyfill-Webcomponents[15]
- Polymer-Project[16]
- X-tags[17]

Obwohl es mehrere Polyfills bezüglich Web-Componnts 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"-Browsern[18] 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 Patterns um die gewünschte DOM-Abstrahierung zu erreichen.

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.

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.

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 3.3 auf Seite 17 erklärt.

3.1 Relevanz von Web-Components hinsichtlich der Forschungsfrage

3.2 W3C Web-Components Standard

3.2.1 Templates

Laut W3C sind Templates

referenz zu W3C adden

> „a method for declaring inert DOM subtress 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 `<li>` -Tags in das Dokument einfügen will. Des Weiteren kann jeder `<li>`-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:

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

```
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:

```
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 15, kann sowohl im `<head>`- als auch im `<body>`-Tag definiert werden. Das Template, inklusive Subtree, ist inaktiv. Dies bedeutet, dass wenn sich ein `<img>`-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>`, `<img>`-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 15

Listing 4 auf Seite 16 basiert auf dem in Listing 2 auf Seite 15 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.

3.2.2 Decorators

3.2.3 Custom Elements

3.2.4 Shadow DOM

3.2.5 HTML Imports

3.2.6 Browser Unterstützung

3.3 Google Polymer

Unterscheidung zwischen predefined Elements, self-written custom Elements und Benutzung von Elementen

Nesting von Elements bzw. Reuse beispielhaft zeigen

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()`[19]. 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.

3.4 Konklusion

4 Web-Components Praxisbeispiel

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

19. Mehr Information zur jQuery.ajax-Funktion unter http://api.jquery.com/jQuery.ajax/

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

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].

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`.