Webarchitekturen
- Einige Gedanken aus der Praxis -

Manfred Wolff, wolff@manfred-wolff.de

Software-Architekturen sind genau so fundamental, wie eine Architektur beim Hausbau. Während beim Hausbau niemand auf die Idee kommen würde den Architekten zu sparen, kommt dieses bei der Entwicklung von Software immer wieder vor. Der Artikel entwickelt einige grundsätzliche Gedanken, die bei der Herausbildung einer Architektur für webbasierte Anwendungen berücksichtigt werden sollten.

Einleitung

Ein Architekt entwirft den Plan für ein Haus. Dafür sammelt er sowohl die Ansprüche seiner Kunden auf (Anzahl der Kinderzimmer, Größe der Küche, Farbe der Wände etc.), als auch bestimme Umgebungsparameter wie z.B. Beschaffenheit des Bodens oder das Aussehen der umgebenen Häuser. Nachdem der Plan gezeichnet ist, endet allerdings die Arbeit des Architekten noch lange nicht. Er ist präsent auf der Baustelle für Fragen, Anmerkungen - manchmal muss er auch Details des Plans den Gegebenheiten anpassen.

Ohne Architektur funktioniert der Hausbau nicht - darüber braucht nicht diskutiert werden. Dennoch gibt es immer wieder Fälle, in denen versucht wird ein Softwaresystem ohne Architektur zu entwickeln. Hier findet man aber genau die gleichen Dinge vor wie beim Hausbau: Ansprüche des Kunden (bestimmte Funktionen des Programms), als auch bestimmte Umgebungsparameter, wie Anzahl der Nutzer des Systems etc.

Am Anfang steht die Anforderung

Software-Architekturen sind kurz gesagt dafür da eine Plattform bereit zu stellen, mit der fachliche, technische und nicht funktionale Anforderungen möglichst optimal umgesetzt werden können. Dabei stehen meistens die nicht-

funktionalen Anforderungen im Mittelpunkt.

Abbildung 1 zeigt die Grundlegenden Elemente einer Softwarearchitektur auf.

- Fachliche Anforderungen: Die fachlichen Anforderungen geben den Ausschlag für ein Projekt, es sei denn, das Projekt ist eine Frameworkentwicklung. Die fachlichen Anforderungen können die Architektur sehr stark beeinflussen, meistens sind es aber nicht funktionale Anforderungen, welche die Auswahl einer konkreten Architektur mehr beeinflussen.
- Nicht funktionale Anforderungen: Diese stehen orthogonal zum gesamten System. Hier werden Eigenschaften wie Verfügbarkeit, Wartbarkeit, Performance, Antwortzeiten etc. beschrieben. Nicht funktionale Anforderungen haben die Eigenschaft, dass sie teilweise in Konkurrenz zueinander stehen (z.B. Performance vs. Wartbarkeit). Teilweise werden nichtfunktionale Anforderungen durch funktionale Anforderungen gelöst (Security - Login), teilweise durch rein technische Lösungen (Ausfallsicherheit - Clusterlösung).

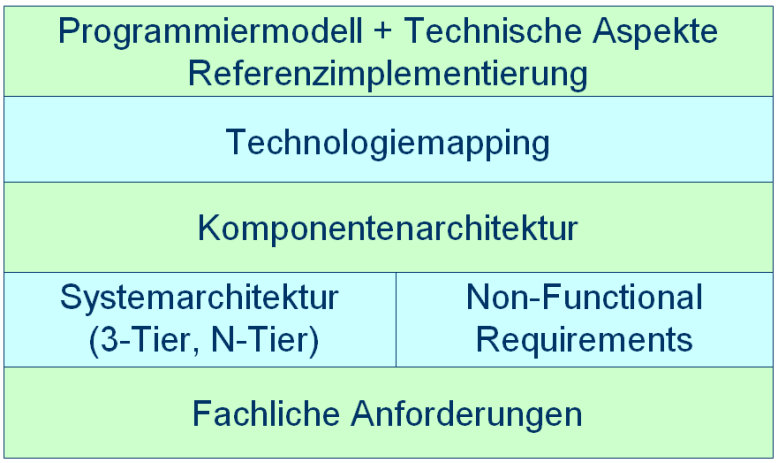

Abbildung 1: Bausteine einer Software Architektur

- Systemarchitektur: Aus den erst genannten Punkten ergibt sich eine Systemarchitektur. Es werden die Schichten (Layer) identifiziert, die für dieses System notwendig sind. Oft kommen klassische 3-Tier Architekturen zum Einsatz (Präsentation, Geschäftslogik, Datenspeicherung).
- Komponentenarchitektur: Neben der fachlichen Identifikation von Komponenten (sowohl fachliche als auch technische Komponenten), muss an

diesem Punkt festgelegt werden wie sich Komponenten definieren. Dazu gehören Dinge wie Lebenszyklus, Erzeugung etc.

- Technologiestack: Erst an diesem Punkt muss sich der Architekt Gedanken über das konkrete Mapping von Architekturbausteinen zur einer Technologie machen. Die Anforderung könnte z.B. Logging sein, die konkrete Technologie im Java Open Source Umfeld Log4J.

- Programmiermodell und Referenzimplementierung: Der letzte Schritt in der Entwicklung einer Architektur für ein konkretes Projekt ist die Erstellung eines Programmiermodells und einer Referenzimplementierung. Oft wird das Programmiermodell von einer konkreten Technologie vorgegeben. Zu diesem Zeitpunkt muss sich auch Gedanken über die Produktions- sowie über die Testumgebung gemacht werden.

Aller Anfang ist schwer

Die Frage ist: Wann beginnt die Erstellung einer Software Architektur, wann endet sie? Ich habe Erfahrungen in der Entwicklung von Architekturen von kleinen Projekten (vier Entwickler) bis sehr großen Projekten (größer 200 Entwickler). Meine Erfahrung ist:

1. Mit der Architektur sollte begonnen werden, bevor die Entwicklung einsetzt, damit die Architektur etwas Vorsprung hat.

2. Die Architekten müssen den gesamten Entwicklungszyklus zur Verfügung stehen. Sie dienen als Katalysator der Architektur in den Entwicklungsteams.

Von daher würde ich folgenden Ablauf empfehlen:

- Kick off der Architektur: Ideal ist, wenn es zu jedem Entwicklungsteam einen konkreten Ansprechpartner gäbe, der die Architektur kennt und im Team etabliert. Sinn haben auch übergeordnete Architektur-Boards zu etablieren, die projektübergreifend agieren und so eine Unternehmens-Architektur vereinheitlichen können. Das Kick off sollte an einem neutralen Ort stattfinden (Tagungshotel), damit frei von der Tagesarbeit gearbeitet werden kann.

- Das erste Papier sollte kurz die Grundzüge der Architektur beschreiben, ohne auf technische Details, Plattformwahl etc. einzugehen; auch wenn die konkrete technologische Plattform schon festzustehen scheint. Im Mittelpunkt des „ersten Papiers“ sollten vor allen Ideen für die Umsetzung der funktionalen- und nichtfunktionalen Anforderungen stehen, sowie eine erste Idee einer Systemarchitektur.

- Aufbauend auf diesem Papier wird verfeinert: Hier stehen vor allem die Komponentenarchitektur sowie das Technologiemapping auf dem Plan. Das Technologiemapping beschreibt, wie Teile einer Systemarchitektur in konkrete Frameworks und Technologien übersetzt werden. In Abbildung 2: Blueprint einer modernen Architektur für webbasierte Programme, ist eine solche Systemarchitektur skizziert, das Technologiemapping könnte wie in Tabelle 1 aussehen:

Systemarchitektur	**Mapping**
Servlet Container	Tomcat 5.5
Rendering	Velocity
Leightweighted Container	Spring
O/R Mapping	Hibernate

Tabelle 1: Beispiel eines Technologiemappings

- Last but not least muss das Architekturteam noch den konkreten Transfer in ein Programmiermodell bzw. einer Referenzimplementierung leisten. Hier sind dann auch Festlegungen für die Produktionsumgebung (Server, Datenbank, Skalierung) sowie für die Testumgebung zu treffen.

Webanwendungen haben von Natur aus bestimmte Eigenschaften, die zu der hier konkret vorgestellten Architektur führen:

- Eine Webanwendung wird durch einen Browser bedient. Das Prinzip solcher Anwendungen ist, dass eine Anfrage an den Server gestartet wird und dieser durch eine Antwort (eine geränderte HTML-Seite oder eine XML-Seite) beantwortet wird. Prinzipiell kann das User Interface auch jeder andere Client sein (Rich-Client, Thin-Client), er muss nur das entsprechende Protokoll bedienen.

- Außer bestimmten Sitzungsparametern, die normalerweise entweder in der HTTP-Session oder in einer Session-Bean abgelegt werden, ist der gesamte Prozess statuslos. Damit können alle Geschäftskomponenten

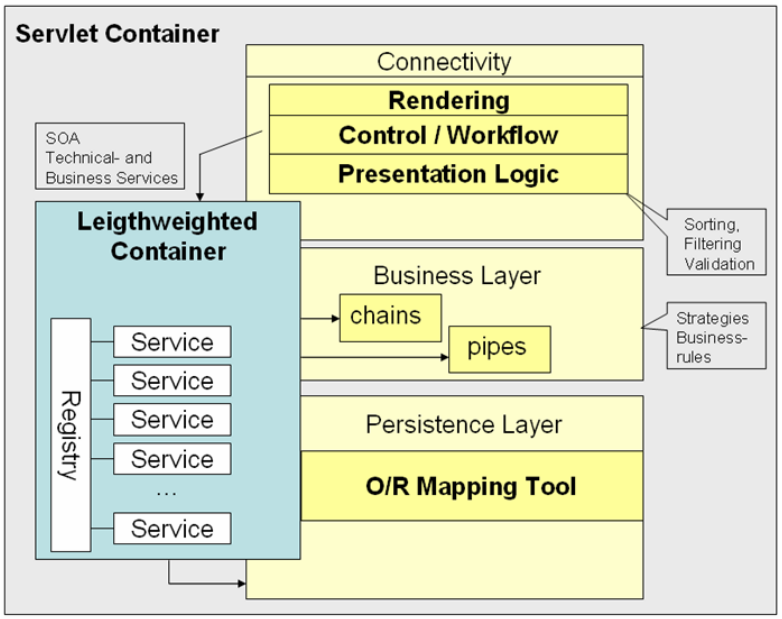

Abbildung 2: Blueprint einer modernen Architektur für webbasierte Programme

auch statuslos ausgelegt sein, so dass sie entweder als Singletons oder als gepoolte Objekte vorliegen können. Damit der Lebenszyklus dieser Komponenten variabel gestaltet werden kann, empfiehlt es sich, einen Container zu benutzen. Dabei muss es nur in seltenen Fällen ein EJB-Containers sein. Leichtgewichtete Container wie Hivemind, Pico-Container oder Spring sind für Webanwendungen ideal geeignet.

- Viel wichtiger als eine klare Layerstruktur (Präsentation, Geschäftslogik, Datenspeicherung) ist es eine klare Komponentenstruktur zu erarbeiten. Ich persönlich sehe kein Problem darin, dass ein Präsentationsobjekt (z.B. eine Struts Action) direkt mit einer Persistenzkomponente interagiert, wenn es nur darum geht, Daten zu beschaffen. Das Zwischenschalten einer Geschäftskomponente erscheint mir da zu akademisch.

Damit sind wir auch schon beim Thema: Der Komponentenarchitektur.

Komponentenarchitektur

Komponente möchte ich wie folgt definieren:

- Eine Komponente ist eine fachliche- oder technische abgeschlossene Einheit. In einer Komponente sind Objekte zusammengefasst, deren Zusammenhang besonders hoch ist. Typische Komponenten können sein Rollensystem, Benutzerverwaltung, Reisekostenabrechnung im fachlichen Bereich und Drucken, Sicherheit etc. im technischen Bereich.

- Komponenten sollen keine bidirektionalen Abhängigkeiten besitzen. Im Idealfall kann eine fachliche Komponente alle seine Anfragen äus sich heraus"beantworten. In der Praxis wird es Komponenten geben, die häufig von anderen Komponenten referenziert werden (z.B. eine Konfigurations Komponente), und einige die vollständig autark arbeiten können.

- Komponenten haben wohldefinierte Schnittstellen. Im Idealfall bestehen die Schnittstellen der Komponente nur aus Interfaces. Eine Komponente selbst ist im Idealfall auch nur ein Interface, welches verschieden implementiert werden kann. Die Schnittstelle sollte keine Annahme über deren Implementierung machen müssen. Das bedeutet konkret:

  - Die Schnittstelle soll keine Typen enthalten, die eine bestimmte Technik erfordern (plan old java object = POJO).
  - Die Schnittstelle sollte nur Exceptions "werfen", die in der Komponente selbst definiert sind.

Die Komponentenstruktur sollte sich grundsätzlich in der Packagestruktur wiederfinden.

What about SOA

Aus meiner Sicht ist SOA (service orientated architecture) fachlich eine Spezialisierung einer guten Komponentenarchitektur. Wenn eine Komponente als Dienst aufgefasst wird, ist die Spezialisierung zu SOA ganz einfach: SOA ist eine Komponentenarchitektur, bei der jede Komponente einen Dienst bereitstellt. Wesentlich dabei ist, dass der Dienst gefunden wird, dazu dienen Registrierdienste oder Service Provider. Das wichtigste bei der SOA-Diskussion erscheint mir, dass SOA-Dienste für den Anwender völlig transparent sind. Dies wird erreicht, in dem die Realisierung des Dienstes sehr strikt von seiner Beschreibung getrennt ist. Dieses wird oft mit Hilfe einer plattform-unabhängigen Beschreibungssprache realisiert (WSDL).

Architektur konkret

Fachliche- und technische Funktionen in werden in Komponenten oder Diensten zusammengefasst. Als Service-Provider wird ein leichtgewichteter Container genutzt. Dieser verfügt über eine Registrierung, in dem sich die Komponenten (Dienste) registrieren und über eine Schnittstelle, mit deren Hilfe die Dienste gefunden werden (Service Broker). Struts ist ein Präsentationsframework. Die Businesslogik sollte in Anwendungs-Containern entwickelt werden. Leichtgewichtete Container wie Spring oder Hivemind bieten eine ideale Ergänzung zu Struts.

Als Client für diese Komponenten (Dienste) werden Action-Klassen in der Präsentationsschicht vorgehalten. Alle Zustandsinformationen sollten in einem Session-Memory abgelegt werden. Dieses Memory sollte plattformunabhängig entwickelt werden, damit unabhängig von einem konkreten Technologiemapping immer das gleiche Programmiermodell umgesetzt werden kann. Hier helfen Kontexte.

Nicht alle Äspekteëiner Anwendung können in Komponenten oder Diensten gekapselt werden. Eine gute, komponentenorientierte Architektur besitzt Eigenschaften, wie "responsibility driven design"(Design getrieben durch Verantwortlichkeiten) und sog. ßeperation of concerts"(Kapselung von Angelegenheiten). In der Praxis gibt es bestimmte Funktionen, die verteilt verantwortet werden müssen, sog. crosscutting concerns. Die Aspektorientierte Programmierung hilft diese verteilten Funktionen wie z.B. das Logging zu bündeln. Aus meiner Sicht gehört eine AOP Framework heutzutage zu einer guten Architektur. AOP wird deshalb zu Recht als "the next big thing after OO"bezeichnet.

Hier zusammengefasst meine Maxime für webbasierte Programme:

1. Kapsele die Clientlogik in Action-Objekte. Clientlogik besteht im Wesentlichen aus den Aufgaben: Validierung, Sortierung, Filterung. Die gesamte Clientlogik sollte sprachenunabhängig gestaltet werden. Dazu dienen Java Ressource-Bundles.

2. Benutze Kontexte für die Kommunikation zwischen den Action Objekten und zur Transformation von Informationen zwischen zwei Anfragen. Damit bleiben die Action-Objekte unabhängig von einer bestimmten Technologie.

3. Kapsele die Serverlogik in Komponenten oder Diensten. Dabei sollten fertige Lösungen für Container (EJB, Leichtgewichtete Container) genutzt werden, um den Lebenszyklus der Komponenten zu regeln.

Was hier noch fehlt ist der Übergang zur Persistenz. Die meisten Webanwendungen kommen ohne Speicherung von Daten nicht aus. Auch gilt das Prinzip der Transparanz: Anwendungen sollten nicht wissen, wie Daten konkret gespeichert werden. Sie sollten mit Datenzugriffsobjekten arbeiten (DAO), welche die konkrete Technik der Datenspeicherung verdecken.

Architektur ist Overhead

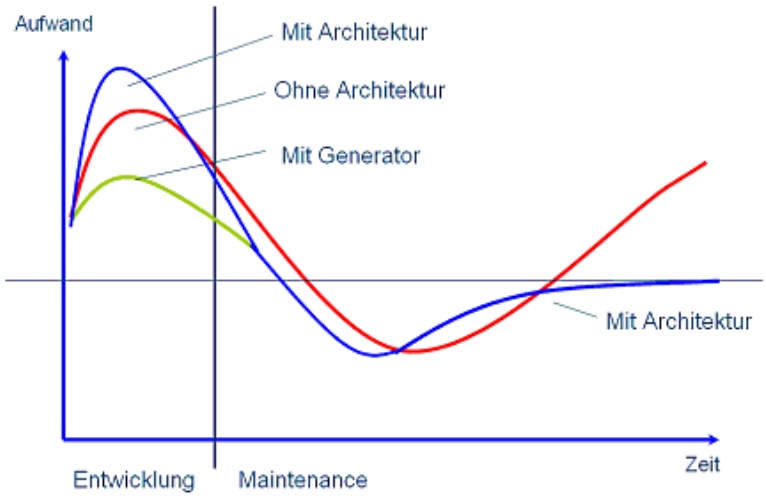

Abbildung 3: Kostenentwicklung mit Einfluss von Architektur und Generierung

In der Einleitung habe ich geschrieben, dass niemand ernsthaft ein Haus bauen würde bevor der Architekt seine Arbeit erledigt hat. Dennoch ist gerade der LAMP-Ansatz ein zwar erfolgreicher aber dennoch oft architekturloser. Wie trifft sich das? Architektur kostet Geld, Architektur kann auch Performance kosten. Das ist unbestritten. Wie in Abbildung 3: Kostenentwicklung mit Einfluss von Architektur und Generierung zu sehen ist, kostet Architektur tatsächlich in der Entwicklung Geld. Aber wie die rote Kurve zeigt, werden ohne eine Architektur die Wartungskosten nicht im Zaum gehalten.

Große Projekte verschlingen ihr Geld in der Wartung, denn über 90% des Software-Lebenszyklus verbringt ein Softwaresystem in der Wartung. Um hier die Aufwände in den Griff zu bekommen ist eine Architektur unabdinglich. Was die Kurve aber auch zeigt: Wenn mit generativen Techniken gearbeitet wird (wie z.B. dem openArchitectureware Framework), können auch die Entwicklungskosten deutlich gesenkt werden.