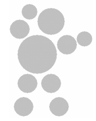


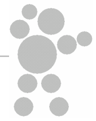
Der kleine Software-Architekt

Teil 2, 17.5.2007
Letzte Änderung: 3.10.07

F.Riemenschneider

mit Anregungen von
Marc Eilens und Markus Lauer

Zusammenfassung:
In dem zweiten Teil der Serie wird das nötige technische Wissen strukturiert, das ein Software-Architekt mitbringen muss, um sich dauerhaft in dieser Funktion zu behaupten. Zu diesem „Katalog" erhält der Leser passende Literaturtipps sowie einige Hinweise zum praktischen Einsatz des Wissens.



Einführung
Der erste Teil dieser Serie hat einen Einstieg in die Architektenlaufbahn aufgezeigt: indem ein Entwickler über die unmittelbaren Aufgaben hinaus positiv wirkt und über den „sauberen" Einsatz von Technologien reflektiert, signalisiert er Bereitschaft, in technischen Fragen Verantwortung zu übernehmen.
In diesem Teil der Serie versuche ich, einen Katalog für das technische Wissen aufzustellen, das ein Software-Architekt zumindest in Ausschnitten besitzen muss, um sich dauerhaft in einer solchen Position zu behaupten. Darüberhinaus gebe ich Hinweise, wie man sich das Wissen aneignen und es später einsetzen kann.

Wenn man Begriffe des technischen Wissens sammelt, dann fällt auf, dass die Stichworte, hinter denen jeweils „ein Brocken Wissen" steckt, klassifizierbar sind. Nun ist das Erfinden gedachter „Schubladen" ein ur-menschliches Mittel, um die komplexe Realität zu denken und in Sprache zu fassen, aber leider besitzt das Gebiet der Software-Architektur noch keine umfassende und präzise Begriffswelt wie etwa die Mathematik, Naturwissenschaften oder theoretische Teile der Informatik.
Das nachzuholen wird diesem Artikel natürlich nicht gelingen, aber so ganz komme ich ohne den Versuch einer solchen Taxonomie, die es mir erlaubt, den Katalog halbwegs systematisch zu gestalten, nicht aus.

Ich bin damit nicht allein: [PBG 04] verwenden in Abschnitt 8 als Sammelbegriff für das wachsende Wissen des Software-Architekten das Wort „Toolbox". Dieser umfasst aber neben dem technischen Wissen z.B. auch methodische Fähigkeiten, die in diesem Artikel zunächst nicht näher betrachtet werden.

Der folgende Abschnitt wird den Aufbau des Katalogs erläutern und begründen. Dann werden in weiteren Abschnitten, soweit das hier sinnvoll ist, die Katalogeinträge unter Angabe von Quellen gelistet. Anschließend findet der Leser einige Tipps, wie man das eigene Wissen erweitern und einsetzen kann.

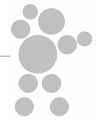
Aufbau des Katalogs
Das technisch orientierte Wissen in der Software-Architektur ist an einigen Stellen enorm schnelllebig, drei Jahre machen soviel aus wie in anderen Branchen ein Jahrzehnt. Daher wird jeder Katalog unvollständig und in der Sekunde des Schreibens schon veraltet sein. Aber es gibt Stellen, die eine deutlich höhere Halbwertzeit besitzen. Es macht daher für meine Ausführungen und die eigene Fortbildung Sinn, sich mehr auf diese zu konzentrieren. Daher ist die Hauptebene der Strukturierung auch an der Langlebigkeit des verbundenen Wissens ausgerichtet:

Ich erläutere im folgenden, was ich unter den o.g. Begriffen verstehe.

Entwurfsgrundsätze beantworten fundamentale Fragen zur Systemstrukturierung. Sie sind wie immerwährende Gesetze und können i.d.R. sowohl im Großen wie im Kleinen verwendet werden. Sie gelten z.T. schon seit mehr als dreißig Jahren, siehe z.B. [Par 72] für die Grundlagen zum „information hiding" und „design for change" oder [YC 79] für die Begriffe „high cohesion" und „low coupling".
Architekturstile sind eine Art Muster für den groben Systementwurf. Sie lassen sich an vielen Stellen auf Entwurfsgrundsätze zurückführen.

Die *Problemstellungen* sind die möglichen Themen, die bei der Neuentwicklung eines Systems abzuarbeiten sind. Bekannte *Lösungsmuster* zu einem Problem beschleunigen die Bearbeitung, da „das Rad nicht zweimal erfunden" werden muss. Das Wissen über die Probleme und anwendbaren Muster ist bezogen auf eine Klasse von Software-Systemen wie z.B. betriebliche Informationssysteme zwar stetig wachsend aber langlebig.
Neben den spezifischen Entwurfsmustern gibt es natürlich eine Reihe allgemeiner Entwurfsmuster, die im Klassiker [GoF] enthalten sind. Sie gehören schon zum Handwerkszeug eines Entwicklers, daher werden sie im folgenden nicht weiter betrachtet.

Unter *Technologien* verstehe ich hier Konzepte, die bei der Lösung der gegebenen Probleme helfen können. Dieser Bereich, dem ich z.B. EJB, WS-Standards, AJAX und



dergleichen zuordne, ist schon recht kurzlebig: in fünf Jahren kann die Welt völlig anders aussehen.

Produkte und *Werkzeuge* sind konkrete Technologie-Implementierungen, die man kaufen, kostenlos herunterladen oder selbst herstellen kann. Man kann sie danach einteilen, ob sie nur zur Entwicklungszeit relevant sind oder auch im lauffähigen System eine Rolle spielen. Beispiel wäre ein Build-Werkzeug wie Ant (i.d.R. nur zur Entwicklung nötig) bzw. ein Application Server wie JBoss (läuft auch, wenn das System im Betrieb ist).
Um meine Beschreibungen etwas schlanker zu halten, kennzeichne ich diese Einteilung über die Begriffe Werkzeug bzw. Produkt.
Den Begriff „Werkzeug" verwende ich für Produkte, die nur zur Entwicklungszeit relevant sind. Produkte, die auch zur Laufzeit eine Rolle im System spielen, gehören schlicht der Klasse „Produkt" an.
Die extreme Kurzlebigkeit des Wissens über Werkzeuge und Produkte entsteht aus dem Innovationsdrang der IT-Branche und der Konkurrenz unter den Herstellern.

Die Begriffsbedeutungen für Lösungsmuster, Technologie, Produkt und Werkzeug sollen zum besseren Verständnis noch in einem kurzen Beispiel erläutert werden.
Ein komplexes fachliches Datenmodell zusammen mit der Unternehmensvorgabe, eine relationale Datenbank einzusetzen, sind zwei Randbedingungen, die der Architekt bei der Problemstellung Persistenz berücksichtigen muss. Ein heute häufig gewähltes Lösungsmuster schlägt den Einsatz eines OR-Mappers vor, um einen großen Teil der verbundenen Probleme in den Griff zu kriegen. Dabei bieten sich verschiedene Technologien an, z.B. EJB Entity Beans, JDO oder ein Produkt, das keine Standard-Schnittstelle implementiert. Um konkret zu werden, muss anschließend aber auch festgelegt werden, welches Produkt zum Einsatz kommen soll, d.h. welche Technologie-Implementierung verwendet wird.

Obwohl im Beispiel so niedergeschrieben, ist die Lösungsfindung keine feste Schrittfolge wie z.B. „Lösungsmuster wählen", „Technologie wählen", „Produkt und/oder Werkzeug wählen", obwohl ein derart stringentes Vorgehen wünschenswert ist. Man denke nur an Technologien, die zwar in Form einer Spezifikation vielversprechend aussehen, zu denen es aber keine ausgereiften Produkte gibt oder keine, die vom Projekt bezahlbar sind. Dann muss man vom Einsatz der Technologie absehen. Oder anders herum kommt es vor, dass Produkte aus „politischen Gründen" gesetzt sind, dann sind damit schon Technologien und mitunter auch Lösungsmuster festgelegt, die verbaut werden müssen. Die Aufgabe des Software Architekten besteht – ganz gleich unter welchen Bedingungen – immer darin, den langfristig wirtschaftlichen Einsatz von Software-Lösungen zu ermöglichen.

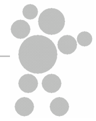
Entwurfsgrundsätze und Architekturstile
Diese beiden Wissensgebiete stellen das Fundament von jeder Entwicklungsarbeit dar.
Entwurfsgrundsätze sind Leitlinien, deren Einhaltung immer anzustreben ist bzw. deren Nichteinhaltung die Ursache für bestehende oder zu erwartende Probleme sein kann.
Um nützlich zu sein, müssen diese Leitlinien konkret sein. Der Grad ihrer Einhaltung sollte sich messen lassen, d.h. idealerweise gibt es Metriken und Messwerkzeuge, die die Überprüfung erleichtern oder sogar automatisieren.
Die meines Erachtens wichtigsten sind:

1. Design for Change.
2. High Cohesion.
3. Low Coupling.

Einen guten Überblick erhält man mit [PoSA 1], eine lebensnahe Einführung mit vielen Beispielen und darüberhinaus Metriken bietet [Mar 03].

Architekturstile hingegen sind die Ausgangspunkte, mit denen ein Gesamtentwurf eines Systemteils, eines Systems oder einer ganzen Systemlandschaft begonnen wird. Ein konkreter Architekturstil fördert automatisch die Einhaltung von Entwurfsgrundsätzen.
So gibt es z.B. in einer Mehrschichtarchitektur per definitionem klare Zuständigkeiten (geförderter Grundsatz: high cohesion) und eine Ordnung von Abhängigkeiten (geförderter Grundsatz: low coupling). In einer SOA werden Dienstschnittstellen nach fachlichen Gesichtspunkten entworfen (geförderter Grundsatz: high cohesion), der Aufrufer kennt keine Implementierung (geförderter Grundsatz: information hiding / design for change), und diensterbringende Systeme oder Komponenten werden in fachliche Domänen gruppiert (geförderter Grundsatz: high cohesion). Und um auch in einer SOA wieder Ordnung in das Abhängigkeitsgeflecht zu bringen, kann man Dienste zusätzlich Schichten zuordnen (geförderter Grundsatz: low coupling).

Ein Architekturstil ist ein Lösungsmuster, aber eben von sehr grundsätzlicher Bedeutung. Ein Architekt muss für ein System (oder einen



Systemteil) anhand von Einflussfaktoren einen geeigneten Stil auswählen. Daher muss er auch einen Überblick über die bekannten Architekturstile für Software besitzen.
Die wichtigsten sind

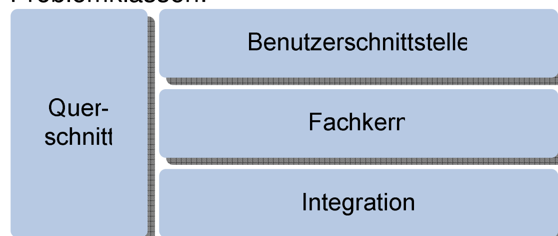
- Layers
- Model-View-Controller
- Broker
- Service-oriented architecture

Problemstellungen und Lösungsmuster
Auch die Problemstellungen und ihre Lösungsmuster lassen sich wieder klassifizieren. Und das geschieht wie folgt:
Betriebliche Informationssysteme sind fast immer in nur drei logische Teile zu zerlegen: die *Schittstellen zum Benutzer*, die *Integration von Fremdsystemen* und der *fachliche Kern* des Systems. Jeder dieser Teile besitzt seine spezifischen Herausforderungen, repräsentiert also eine eigene Problemklasse. Insbesondere umfasst der Aufbau jedes Teils in Anlehnung an [Sie 04] wieder drei Aspekte: die technische Infrastruktur in Form einsetzbarer Produkte, die technische Architektur sowie die Implementierung von Fachlichkeit.

Neben den drei Problemklassen gibt es zusätzlich noch die Klasse *querschnittlicher Probleme*, welche sich weitgehend durch das ganze System ziehen.

Nun könnte man einwenden, dass diese nach Mehrschicht-Architektur klingende Zerlegung nicht auf „moderne" SOA Landschaften passt und daher schon an Relevanz verliert. Das ist aber nicht richtig: Eine SOA ist eine Software-Architektur, die sich auf die geordnete Verbindung einer Menge von Systemen über fachlich motivierte Dienste konzentriert. Die konkreten Probleme jedoch, die in den Einzelsystemen und in der SOA Landschaft insgesamt auftreten, sind dieselben, die auch nach der obigen Zerlegung klassifiziert werden können. Sie treten quasi nur „anders geschnitten" auf.

Hier sind also die vier genannten Problemklassen:



- *Querschnitt* (foundation): Probleme, die sich an vielen Stellen des Systems auswirken.
- *Fachkern* (domain): Probleme, die den eigentlichen fachlichen Kern des Systems betreffen.
- *Integration* (integration): Probleme, die sich bei der Anwendungsintegration und Anbindung von Fremdsystemen ergeben.
- *Benutzerschnittstelle* (user interface): Probleme, die sich aus den Schnittstellen zu menschlichen Benutzern ergeben, d.h. alle Arten menschenlesbarer Ausgabe und von Menschen erfassbarer Eingabe betreffen.

Innerhalb jeder Problemklasse kann man nun die Themen auflisten, die bei betrieblichen Informationssystemen üblicherweise zu bearbeiten sind. Und hier lohnt es sich für den werdenden Software-Architekten besonders, Zeit und Mühe in das Erlernen zu investieren. Ich versuche dazu im folgenden, zu jedem Schlagwort eine Umschreibung der Probleme in Form von Fragen anzugeben.

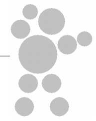
Querschnittsthemen
Verteilung. Ist eine Verteilung des Systems erforderlich? Welche Bestandteile des Systems werden wo deployt? Welche technische Kommunikationsinfrastruktur wird verwendet? Wie werden daraus resultierende Probleme bzgl. Sicherheit, Performanz, Protokollierung, Transaktionssicherheit usw. begrenzt oder gelöst?

Datenintegrität / Transaktionen. Wie stellt man sicher, dass Daten bei Mehrbenutzeraktivität und über verschiedene Systeme hinweg konsistent verarbeitet werden? Wie geht man mit langlaufenden (fachlichen) Transaktionen um?

Internationalisierung (I18N). Was ist vorzusehen, damit eine Anwendung in verschiedenen Ländern mit unterschiedlichen Sprachen, Einheiten für Währung und Maßzahlen sowie Zahlenformaten eingesetzt werden kann?

Sicherheit. Wie stellt das System sicher, dass keine Attacken auf oder Mißbrauch von Daten und Funktionen möglich sind? Dazu gehören auch so prominente Themen wie Authentifizierung und Authorisierung.

Fehlerbehandlung. Wie reagiert das System auf unvorhersehbare Ereignisse (z.B. Hardware-Ausfall)? Wie wird mit erwartbaren fachlichen Fehlern umgegangen?



Protokollierung. Was muss das System tun, damit sein Zustand und seine Aktivitäten entlang der Zeitachse nachvollziehbar sind? Wo und was wird geloggt? Wie sehen Log-Meldungen aus? Wie wird zwischen betriebsrelevanten (Log-) und entwicklungsrelevanten (Trace-) Meldungen unterschieden? Wenn Logs verteilt sind, wie können sie zusammengeführt werden?

Konfiguration. Welche Merkmale des Systems müssen zur Laufzeit oder mithilfe eines Neustarts verändert werden? Wie kann diese Veränderung durch den Betrieb oder Anwender herbeigeführt werden?

Zeitsteuerung. Wie können Aufgaben zeitgesteuert und/oder wiederholt ausgeführt werden? Wie werden ggf. Zeitpläne verwaltet?

Betriebsunterstützung. Was muss das System mitbringen, damit Mitarbeiter des Betriebs seinen Zustand einsehen und bewerten können, Pflegemaßnahmen durchführen und Fehlersituationen erkennen und bereinigen können? Wie sieht der Wiederanlauf nach einem Unfall aus? Wie wird das Update auf neuere Versionen unterstützt?

Management Reporting. Wie werden Kennzahlen und Statistiken, die der Kostenträger zwecks wirtschaftlicher Bewertung benötigt, gesammelt bzw. erzeugt?

Wartung durch Entwicklung. Wie wird der Einsatz und die Pflege von automatisierten Tests realisiert? Wie sieht eine Instrumentierung aus, damit das System Kennzahlen über Performance und Ressourcenverbrauch sammelt?

Integration

Prozesse / Nutzerkollaboration. Werden Geschäftsprozesse (GP) systemübergreifend definiert und technisch abgebildet? Wie ist das Zusammenspiel mit den ggf. vorhandenen GP-Implementierungen der beteiligten Systeme? Wie wird ggf. mit Zustand in der Schnittstelle verfahren?

Systemkopplung. Wie sollen zwei oder mehr Systeme grundsätzlich miteinander kooperieren? Asynchron oder synchron? Nur „fire and forget“ oder „request and response“? Soll dokumentenorientiert, im Funktionsaufruf-Stil oder gar durch Nutzung gemeinsamer DB-Tabellen kommuniziert werden? Wie wird bei asynchroner Kommunikation ggf. die Korrelation zwischen Anfrage und Antwort hergestellt? Welche transaktionalen Eigenschaften werden benötigt und wie werden sie sichergestellt?

Impedance Mismatch.
Wie sehen die grundsätzlichen Unterschiede zwischen den Datenrepräsentationen der beteiligten Systeme aus? Handelt es sich um objekt- oder tabellenorientierte Darstellungen?
Wie werden z.B. Objektreferenzen bzw. Fremdschlüssel behandelt? Sind Datennetze Bäume oder gerichtete Graphen? Gibt es Polymorphie?
Gibt es ein gemeinsames Nachrichten- und Datenmodell, das in einer Spezifikation der Schnittstelle festgelegt wird?

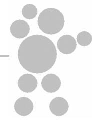
Datentransport. Wie werden Daten zwischen den beteiligten Systemen ausgetauscht? Über ein Messaging System, über RPC-Mechanismen (RMI, IIOP, RFC, …), per FTP oder über gemeinsam genutzte Datenbanktabellen? Wie werden die beteiligten Systeme / Dienste ggf. an einer Kommunikationsinfrastruktur registriert und aufgefunden?

Datentransformation. In welchem Umfang, wie und wo sollen fachliche oder syntaktische Transformationen von ausgetauschten Geschäftsdaten stattfinden? Wie und womit werden Mappings zwischen den Daten des Fachkerns der Anwendung und dem Zielformat durchgeführt?

Fachkern

Persistenz. Wie hält das System dauerhaft die Daten vor, die es führend oder als Replik von anderen Systemen pflegt und verwendet? (Dauerhaft heißt dabei, dass die Daten auch nach einem Neustart des Systems wieder verfügbar sind.) Wie greift der Code, der Geschäftslogik implementiert, auf persistente Daten zu? Wie werden Datenmodelle, die sehr umfangreich sind, gepflegt und letztlich in Code/Schema überführt? Nach welchen Kriterien werden große Modelle unterteilt? Wie wird die Migration sichergestellt? Wie wird ggf. eine Historisierung von Geschäftsdaten ermöglicht?

Geschäftsregeln / Validierung. Wie werden Regeln, die sich aus der Fachdomäne und der konkreten Anwendung ergeben, explizit implementiert, so dass sie leicht gefunden und geändert werden können? Wie und wann werden Prüfregeln auf Benutzereingaben oder Daten von Fremdsystemen angewendet und wie gelangen die Prüfergebnisse dann zum jeweiligen Verursacher (Benutzer, Fremdsystem, Clearingstelle)?



Prozesse / Nutzerkollaboration. Wie kann das System die explizite Abbildung von Geschäftsprozessen oder Ausschnitten aus diesen auf eine oder mehrere Anwendungen erleichtern, so dass später preiswert Änderungen möglich sind? Wie wird die Zusammenarbeit mehrerer Anwender an gleichen geschäftlichen Daten innerhalb von Prozessen unterstützt? Wie kann Bearbeitungsfortschritt sichtbar und meßbar gemacht werden?

Fachkonstanten / Aufzählungen. Wie bildet das System die möglicherweise zahlreichen Konstanten mit fachlicher Bedeutung ab, so dass auch über eine Systemevolution hinweg Einträge hinzugefügt oder geändert werden können? Wann sind Fachkonstanten erforderlich, wann sind sie durch Konfiguration oder polymorphes Verhalten vermeidbar? Degeneralisierung?

Massendatenverarbeitung. Wie kann das System performant und zuverlässig viele Datensätze in einem Lauf verarbeiten? Wird alles in einer Transaktion oder in vielen einzelnen Transaktionen erledigt? Was passiert mit Datensätzen, bei deren Verarbeitung Fehler auftreten?

Benutzerschnittstelle

Terminal, d.h. GUI und/oder Browser. Über welche Bedienungsschnittstellen können die verschiedenen Gruppen von Anwendern das System benutzen? Wie gelangen die Benutzeroberfläche und Updates von ihr zum Anwender? Wie trennt man Präsentation von der Präsentationslogik? Wo liegt der Schnitt zur Anwendungslogik oder Domänenlogik? Wie sieht diese Schnittstelle aus? Wie und wo wird der Zustand aus Benutzersicht gepflegt (z.B. Window-Manager bzw. Pageflow-Engine)? Mit welchen Werkzeugen wird der Entwurf der Oberfläche erleichtert? Gibt es Werkzeuge zum Rapid-Prototyping? Wie wird der Übergang von Usability-Prototypen zur implementierten Präsentationschicht gestaltet? Wie stellt das System die möglichst frühe fachliche Prüfung von Eingaben sicher? Wie wird das Databinding, d.h. der Übergang von Attributinhalten aus Geschäftsobjekten zu Elementen der Oberfläche, realisiert?

Drucken. In welchem Format werden Druckerzeugnisse abgelegt? Wie werden Druckerzeugnisse produziert, d.h. wie geschieht die Zusammenführung von Templates und Inhalten aus Geschäftsobjekten? Welche Werkzeuge können dies erleichtern? Wie werden Druckerzeugnisse zum Drucker transportiert? Wie wird zentrales Drucken gesteuert und überwacht? Wie ist das Zusammenspiel von Geschäftsprozessen mit in ihrem Rahmen durchgeführten Ausdrucken?

Offline Client. Folgende Fragen werden relevant, wenn ein mobiler Client offline, d.h. ohne Verbindung zum Server, arbeitsfähig sein soll. Wie kann der Client auf Geschäftslogik zurückgreifen? Wie werden Bewegungsdaten lokal zwischengespeichert? Wie werden Bestandsdaten auf den Client transportiert? Wie wird mit Synchronisationskonflikten umgegangen? Wie reagiert das System auf Dienstaufrufe, die nur online verfügbar sind?

Sonstige Ein-/Ausgabegeräte. Die Fragen, die sich hier ergeben, hängen sehr stark von den Fähigkeiten und Einschränkungen der individuellen Geräteklassen ab. Daher fällt es mir schwer, repräsentative Fragen zur Problemcharaktersierung zu formulieren.

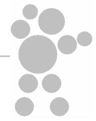
Daten Im- und Export. Wie verarbeitet das System Anfragen nach einem Massendatenexport? Wie wird ein Massendatenimport verarbeitet? Wie erfolgt eine Fortschrittsrückmeldung?

Technologien

Technologien sind im Gegensatz zu Produkten und Werkzeugen, die eingesetzt werden, abstrakt. Man verwendet sie ja nicht direkt, sondern immer eine taugliche Implementierung. Wir nutzen in Diskussionen trotzdem häufig diese Abstraktionen, um die Wahl eines Produkts offenzulassen. Außerdem kann ein Produkt gleich eine ganze Reihe von Technologien implementieren, so dass durch Nennung des Produktnamens nicht klar ist, in welcher Rolle dieser verwendet oder welche funktionale Komponente des Produkts gemeint ist.

Technologien helfen so als konzeptuelle Bausteine, Lösungsmuster umzusetzen. Für die Zeitsteuerung z.B. verwendet man üblicherweise eine Komponente „Scheduler“. In der konkreten Lösungsfindung definieren wir, was wir von einem solchen Scheduler erwarten. Dieser kann letztlich als externes Produkt eingekauft werden oder ist in einfacher Form in einem Framework wie Spring integriert. Unsere Anforderungen helfen uns, die richtige Wahl zu treffen.

Technologien entstehen häufig aus Produkten. Ein Hersteller beginnt, andere springen auf den Zug auf, weil es einen breiten Markt zu geben scheint. Und um dem Kind einen Namen zu geben, wird letztlich eine griffige



Bezeichnung mit einem einprägsamen Akronym gefunden.
Manchmal ist aber auch die Forschung zuerst da. In ihrem Rahmen können neue Konzepte lange diskutiert und erprobt werden, bis ein erste industrietaugliche Implementierung entsteht.

Technologien können mehr oder minder gut spezifiziert sein und werden nicht selten auch durch APIs mit hoffentlich präzise spezifiziertem Verhalten repräsentiert. Das ist im Java-Umfeld ausgeprägt anzutreffen. Zu weit verbreiteten Technologien sind auch Musterkataloge zu finden wie z.B. [J2EE] und online http://java.sun.com/blueprints.

Die Liste der nennbaren Technologien ist lang und wächst ständig. Daher beschränke ich mich auf wenige, die häufig im Umfeld betrieblicher Informationssysteme anzutreffen sind:

Grundlegende Applikationsinfrastrukturen
Application-Server
Web-Server / Servlet Container
Portal-Server
Load-Balancer
Verteilungsinfrastrukturen wie EJB, CORBA, DCOM oder SOAP
Komponenten-Frameworks
Workflow- / Rules-Engine
OR-Mapper
(R)DBMS

Integrationstechnologien
Web-Service Standards
Messaging System
ESB/JBI
SCA/SDO
BPEL-Engine

Technologien für Benutzerschnittstellen
Web-Framework
GUI-Framework
Print-Management-System
Content-Management-System
Rendering (d.h. aus Layout und Daten les- und/oder druckbare Ergebnisse produzieren).

Technologien für Querschnittthemen
Directory-Service
Aspekt-orientierte Programmierung
Logging-Framework
Scheduler

Werkzeuge für den Entwicklungsprozess
Build-Werkzeug
Generierungswerkzeug
Modellierungswerkzeug
Unittest-Framework
Metriken-Werkzeug

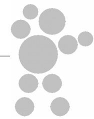
Geeignete Literatur und i.d.R. auch taugliche freie Implementierungen für Prototypen oder sogar Produktion gibt es inzwischen zu jeder relevanten Technologie.

Produkte und Werkzeuge
An dieser Stelle könnte ich versuchen, eine seitenlange Liste von Namen, Abkürzungen und Versionsnummern zusammenstellen, die nicht vollständig und schnell veraltet wäre. Da dies müßig ist, unterlasse ich das.
Stattdessen verweise ich hier auf http://java-source.net, eine Link-Sammlung zu Open-Source Produkten und Werkzeugen für die Java-Welt.

Um einen allgemeinen Überblick zu bekommen und zu behalten, empfehle ich, regelmäßig einschlägige Magazine (auf Papier und online) zu „scannen". Darüberhinaus sind Konferenzen und Messen ideale Orte, um geballte Informationen zu sammeln. Und – last but not least – sollte man bei neuen Namen und Akronymen, die einem in Gesprächen mit Kollegen begegnen, recherchieren, was es ist und ob sich eine tiefere Beschäftigung damit lohnt.

Das eigene Wissen erweitern
An der Menge von Problemklassen und Technologien erkennt man schnell, dass es aussichtslos ist, sich all das mögliche Wissen „in einem Rutsch" aneignen zu wollen. Also wo beginnen?
Eine gesunde Basis halte ich für unumgänglich, und die besteht aus praktisch verwertbaren Entwurfsgrundsätzen [Mar 03] und den Architekturstilen in [PoSA 1] sowie für SOAs den Artikel unter http://www.xml.com/pub/a/ws/2003/09/30/soa.html. In [Mar 03] finde ich besonders die Regeln und Metriken zum Package-Design äußerst nützlich. Diese sind übrigens auch im Web unter http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf verfügbar.
Wer ein wenig Historie mag, dem sei auch [Par 72] oder das ganze Paket in [HW 01] empfohlen. Der Aspekt „Trennung der Fachlichkeit von technischer Architektur" wird gut in [Sie 04] behandelt. Das Spring-Framework (www.springframework.org) liefert passend dazu mit dem Modul „Spring-Core" eine ideale Infrastruktur für einen kopplungsarmen Systemaufbau.

> *Eigne Dir die Entwurfsgrundsätze und das Wissen über Architekturstile an. Übe sie bewusst durch Bewertung eigener und fremder Lösungen ein.*



Darüberhinaus empfehle ich gleichermaßen ein Querlesen und die Spezialisierung in einzelnen Themen der Problemklassen. Durch das Querlesen, also das Lernen, was es grundsätzlich gibt, ohne sich die Details aneignen zu wollen, erwirbt man die Fähigkeit, Architekturprobleme als solche zu erkennen und festzustellen, wo man nachlesen oder wen man fragen kann. Hier ist [PoEAA] auf jeden Fall ein guter Anfang, da dort viele Klassiker-Probleme in Informationssystemen besprochen werden.
Durch die Spezialisierung qualifiziert man sich für konkrete Jobs in einem Projekt. Dabei heißt Spezialisierung im einzelnen:

1. die Probleme einer Problemklasse zu verstehen,
2. die Lösungsmuster und einsetzbaren Technologien zu kennen,
3. die Produkte, die häufig zum Einsatz kommen, kennenzulernen, und letztlich
4. beispielhaft Lösungen zu implementieren, um nicht beim theoretischen Wissen zu bleiben.

Beschäftige Dich mit einzelnen Themen aus den vier Problemklassen intensiv, um Dich zu spezialisieren und Dein Profil zu schärfen. Lies dazu, diskutiere und probiere Vorschläge aus. Untersuche implementierte Lösungen, um zu lernen, wie andere denken.

Im folgenden möchte ich noch einige auf die Problemklassen bezogene Literaturtipps loswerden:

Querschnittsthemen
Das Thema Verteilung wird J2EE bezogen in [Joh 02] im Kapitel 1 behandelt, ansonsten etwas technologieneutraler in [PoEAA], Kapitel 7.
Das Thema Transaktionen wird überblicksweise auch in [PoEAA], Kapitel 5, behandelt. Dort sind auch weitere Literaturhinweise zu finden. [Sie 04] widmet sich in Abschnitt 9 u.a. auch diesem Thema.
Zum Einstieg in die Probleme rund um Internationalisierung und Lokalisierung kann ich http://developers.sun.com/dev/gadc/technicalpublications/articles/archi18n.html empfehlen.
Fehlerbehandlung wird gut in [Sie 04], Kapitel 5, behandelt, darüberhinaus enthält [Joh 02], Kapitel 4, eine erfrischende Abhandlung im J2EE Kontext.
Für die übrigen Themen sind mir momentan keine befriedigenden Quellen bekannt.

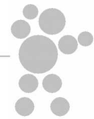
Integration
Zum Thema Messaging bietet [EIP] einen gelungenen Überblick. Heute darf man bei Integrationsaufgaben den Architekturstil SOA nicht mehr vernachlässigen. Einen Einstieg kann man sich (noch) mit [KBS 04] verschaffen. Der Bereich Integration ist in den letzten Jahren stark in Bewegung bekommen, so dass man sich am besten durch Online-Artikel und Spezifikationen z.B. über www.oasis-open.org und andere Konsortien einen aktuellen Überblick verschafft.

Fachkern
Das lesenswerteste, was mir bislang untergekommen ist, findet sich in [Eva 03]. Einen Auszug daraus kann man unter http://www.domaindrivendesign.org/books/PatternSummariesUnderCreativeCommons.doc finden.
Das Thema Persistenz mit einem OR-Mapper wird recht gut und praktisch in [BK 04] beschrieben.

Benutzerschnittstelle
In diesem Umfeld wird als Muster häufig MVC (Model-View-Controller) genannt [PoSA 1]. Obwohl in ihm die Grundstruktur heutiger Benutzerschnittstellen beschrieben ist, reicht ein Verweis auf MVC (oder MVC Model 2 für Web-Anwendungen) kaum aus, um eine gutes Design zu implementieren.
Leider habe ich bislang keine einzelne umfassende Quelle gefunden, in der eine halbwegs vollständige und praktische Abhandlung zum Aufbau von GUIs oder Web-Frontends nachzulesen wäre. Daher nenne ich folgend die wenigen Stellen, an denen man Wissensfragmente findet.

Für GUIs finde ich im Moment als Einstieg http://www.martinfowler.com/eaaDev/uiArchs.html sehr nützlich. Um sich das dort erwähnte Databinding mal konkret anzusehen, kann man http://www.jgoodies.com/download/libraries/binding/binding-1_4_0.zip ausprobieren.
Zu konkreten Lösungen z.B. zum Window-Management und die Dialogzustandskontrolle sind mir keine schriftlichen Quellen bekannt. Immerhin einen Überblick über die Architektur einer GUI und den Schichtenaufbau erhält man in [Sie 04], Kapitel 10.

Einen Überblick zum Design eines Web-Frontends findet man in [Joh 02] in den Kapiteln 12 und 13. Auch in [PoEAA] werden einige der typischen Muster in Kapitel 14 besprochen. Im Gegensatz zur GUI gibt es für Web-Frontends einige Frameworks, deren Studium sich lohnt, z.B. Struts oder Tapestry



von Apache, OpenSymphony WebWork, und Spring-MVC.

Der Einsatz des Wissens

In der praktischen Architekturarbeit gilt es, zu spezifischen Problemen des zu bauenden Systems Lösungen zu finden, um dann Technologien und geeignete Produkte auszuwählen. Dies ist gerade bei seltener auftretenden Problemen kein linearer und theoretischer Prozess, sondern hat viel mit Versuch und Irrtum zu tun, da der Teufel im Detail liegt. Dies ist einer der Gründe, warum Architekten auch implementieren können müssen, und warum – meines Erachtens – ein guter Architekt einige Zeit als guter Entwickler gearbeitet haben muss.

Während der „Forschungsarbeit“ der Lösungsfindung orientiert sich ein Architekt an den Entwurfsgrundsätzen. Solange die werdende Lösung diesen gerecht wird, wird auch die Qualität stimmen. Die bekannten Lösungsmuster beschleunigen die Arbeit, da man „das Rad nicht neu erfindet“. Zudem erleichtern sie die Verständigung bei der Diskussion von Lösungsansätzen mit Kollegen. Der Architekt tut gut daran, die Wortführer und „key player“ im Team zu finden, und seine Sorgen und Ideen noch während der Lösungsfindung zu teilen, um späte Konfrontationen zu vermeiden.

Steht die Lösung erstmal, so muss nach erfolgreicher Verprobung auch die Dokumentation stattfinden, in der erläutert wird, wie die Lösung aussieht und warum sie gerade so aussieht, welche Alternativen es gibt und warum sie nicht gewählt wurden. Dadurch können andere die Arbeit des Architekten nachvollziehen und ggf. bewerten oder korrigieren. Ein solches Review ist ein wichtiger Prüfstein, der verhindern soll, dass unausgereifte Lösungen „in die Breite“ getragen werden.

Während der Implementierung muss der Architekt, der den Entwurf „verbrochen“ hat, nicht selten unterstützen oder anleiten. Dies ist der Moment, in dem soziale Kompetenzen sehr wichtig werden, denn er muss den schmalen Grat zwischen Führung und Diensterbringung finden. Konsens ist wichtig, ein qualitativ hochwertiger Entwurf darf diesem aber nicht geopfert werden.
Werden jedoch Mängel am Entwurf sichtbar, so sollten alle Beteiligten dies als Chance begreifen, den Entwurf und damit das werdende System zu verbessern.
Auch in diesem Teil der Arbeit muss der Architekt sachlich sattelfest argumentieren können, wobei ihm sein technisches Wissen in all seinen Ebenen nützt.

Zusammenfassung und Ausblick

Dieser Teil der Serie hat das technische Wissen, das ein Software-Architekt mindestens ausschnittsweise verinnerlicht haben sollte, in klarer Struktur benannt. Diese Struktur bietet Orientierung, um den eigenen Standpunkt festzustellen und konkrete Lernziele auszumachen. Wo es möglich ist, sind Angaben relevanter Literatur zu finden.

Im nächsten Teil geht es darum, eine komplette technische Architektur beispielhaft anhand von Rahmenbedingungen, einer technischen Infrastruktur und dem Architekturstil „Layers“ herzuleiten. Dann wird das, was in diesem Teil größtenteils nur aufgezählt und angedeutet wurde, praktisch verwertet.

Literatur

[BK 04] Bauer, King, “*Hibernate in Action*”, Manning, 2004.

[EIP] Hohpe, Woolf, “*Enterprise Integration Patterns*”, Addison-Wesley, 2003.

[Eva 03] E.Evans, “*Domain-Driven Design*”, Addison-Wesley, 2003.

[Fow 00] M.Fowler, „*Refactoring*“, Addison-Wesley, 2000.

[GoF] Gamma, Vlissides, Helm, Johnson, “*Design Patterns*”, Addison-Wesley, 1994.

[HW 01] Hoffmann, Weiss, “*Software Fundamentals*”, Addison-Wesley, 2001.

[J2EE] D.Alur et al., “*Core J2EE Patterns. Best Practices and Design Strategies*”, Prentice Hall, 2003.

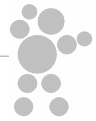
[Joh 02] R.Johnson, “J2EE Design and Development”, Wrox Press, 2002.

[KBS 04] Krafzig, Banke, Slama, „*Enterprise SOA. Service Oriented Architecture Best Practices*“, Prentice Hall, 2004.

[Lar 02] Craig Larmann, “*Applying UML and Patterns”*, Prentice-Hall, 2002.

[Mar 03] R.C.Martin, “*Agile Software Development*”, Prentice-Hall, 2003.

[Par 72] D.L.Parnas, “*On the criteria to be used in decomposing systems into modules*”, Comm. ACM 15(12), 1972, enthalten in [HW 01].



[PBG 04] Posch, Birken, Gerdom, „*Basiswissen Softwarearchitektur*“, dpunkt Verlag, 2004.

[PoEAA] Fowler, Rice, Foemmel, “*Patterns of Enterprise Application Architecture*”, Addison-Wesley, 2002.

[PoSA 1] Buschmann et al., “*A System of patterns. Pattern oriented Software Architecture 1*”, Wiley&Sons, 1996.

[Sie 04] J.Siedersleben, „*Moderne Softwarearchitektur*“, dpunkt Verlag, 2004.

[YC 79] E.Yourdon, L.Constantine, „*Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*”, Prentice Hall, 1979.