Der kleine Software-Architekt

Teil 3, 24.10.2007

F.Riemenschneider

Zusammenfassung:
Die bisherigen Teile enthielten „nur“ Theorie insofern, dass keine konkreten Beispiele für eine Architektur herangezogen wurden. Das will dieser Teil mit einer ausführlichen Beschreibung eines echten Systemaufbaus ausgleichen. Der Leser erfährt, wie man von einem groben Schichtenentwurf auf Maven Module und Java Packages kommt und einige typische technische Architekturprobleme löst, um das Beispielsystem zum Leben zu erwecken.

Einführung

Die ersten beiden Teile dieser Serie sind recht trocken: sie haben wenig mit konkreter Technologie und existierenden Produkten zu tun. Wirklich für die Praxis lernen kann man aber nur am Beispiel. In diesem Artikel werde ich den Entwurf einer echten, implementierten technischen Architektur beschreiben, bei dem (im Software Sinne) handfeste Bestandteile verbaut sind.

Es ist mir wohl bewusst, dass trotz vieler nützlicher Ideen und Kniffe, die ich hier verwendet habe, die Halbwertzeit des Wissenswerts an manchen Stellen vielleicht ein halbes Jahr beträgt. Mit der sich stetig ändernden Werkzeug- und Produktlandschaft würde man einige Dinge in einem halben Jahr vielleicht ganz anders machen.

Es kommt mir daher viel stärker darauf an zu zeigen, dass die „schöne Theorie“ tatsächlich zum Einsatz kommt und dabei spürbaren Halt gibt. Es geht sogar noch weiter: in Produkten wie Spring oder Maven 2 stecken wertvolle Design-Prinzipien so selbstverständlich drin, dass – so hoffe ich – der praktische Einsatz bei manchem Entwickler automatisch zu einem besseren Design führt. Wobei besser hier heißt: weniger Abhängigkeiten, insgesamt eine geringe Kopplung zwischen den Softwareteilen und hohe Kohäsion innerhalb jedes Teils.

Die Architektur, die ich hier zeige, basiert auf Schnittstellen und wohlgeordneten Abhängigkeiten. Sie setzt Java Technologien wie Spring, Maven 2, JPA, JSF und Axis2 ein, liefert sich diesen aber nicht auf Gedeih und Verderb aus. Insbesondere wird die Geschäftslogik, die sich aus Präsentations-, Applikations-, Adapter- und Domänenlogik zusammensetzt, nicht mit den Programmteilen vermengt, die konkret auf Libraries wie Hibernate, JAXB oder Axis2 zugreifen. So wird insgesamt eine geringere Komplexität erreicht, denn – um mit den Worten von J.Siedersleben [Sie 04] zu sprechen – die verschiedenen Software Kategorien A und T bleiben sauber getrennt. Ein weiterer wichtiger Effekt ist, dass man bei sich ändernden Anforderungen oder unerwarteten Problemen leichter scharfgeschnittene Korrekturen durch Austausch einzelner Technologien oder Produkte durchführen kann. So wird ein Austausch nicht kostenlos, aber die Kosten sind wahrscheinlich geringer und in jedem Falle gut abschätzbar.

Wie geht’s hier weiter im Text? Am Anfang eines jeden Entwurfs stehen die konkreten Anforderungen an ein Software-System, dazu werde ich ein paar Sätze sagen.
Im Anschluss muss ich noch zwei Konzepte erläutern: erstens zur Trennung der Architekturarbeiten in Infrastrukturauswahl, technischer Architektur und Facharchitektur sowie zweitens zu den Schattierungen des Begriffs „Geschäftslogik“.
Dann werde ich ein wenig zur Software Entwicklungsumgebung (SEU) erklären, denn eine technische Architektur hat weitreichenden Einfluss auf eine SEU, z.B. – um nur das naheliegendste zu nennen – braucht man zur Java Entwicklung eine Java IDE und für SOAP Nachrichtenkommunikation ist die Verwendung von z.B. Soap-UI oder Tcpmon empfehlenswert. Und als letzten Punkt, bevor wir in die Architektur des Systems einsteigen, erkläre ich noch, was in einem SOA Projekt mit mehreren Teilsystemen deren Schnittstellen mit der Organisation von Maven Projekten im Konfigurationsmanagement zu tun haben.

Und dann wird’s konkret. Ich zeige den Grobentwurf für ein System, der sich an dem klassischen Stil einer Schichtenarchitektur anlehnt und wie sich daraus Maven Module ergeben. Anschließend folgt eine Abhandlung zu einer kleinen Auswahl von Querschnittsthemen, die sich durch das ganze System durchziehen. Wenn der Grobentwurf in Form von Modulen dann zusammen mit den Querschnittslösungen den Rahmen gibt, kann ich die einzelnen Schichten beleuchten, und zwar immer mit zwei Fragen: wie kommt die Technologie zum Einsatz, um die typischen Probleme zu lösen? Und wo wird die Geschäftslogik dann liegen? Daraus ergibt sich mithin auch das jeweilige Package Design und welche Elemente fachlich konkret zu implementieren und konfigurieren sind.
Und zum Schluss gebe ich dann noch ein paar Hinweise zur Gliederung der Spring-Konfiguration, damit sie beherrschbar bleibt.

Anforderungen und andere Einflussfaktoren

Die Autoren von [PBG 04] verwenden auf das Thema Einflussfaktoren ein kurzes aber lesenswertes Kapitel. Die Architektur eines Softwaresystems basiert im wesentlichen nicht auf den funktionalen Anforderungen sondern auf nicht-funktionalen Anforderungen z.B. bzgl. Performanz, Durchsatz, Änderbarkeit, Sicherheit und Zuverlässigkeit. Neben diesen nicht-funktionalen Anforderungen gibt es weitere Einflußfaktoren, z.B. wenn bestimmte Produkte oder Hersteller strategisch festgelegt sind, Fachentwickler mehr schlecht als recht für die Technologien qualifiziert sind oder ein besonders aggressiver Zeitplan verfolgt wird,

der die vollständige Lösung bestimmter Probleme vor dem ersten Release gar nicht erlaubt.
Auf solche Umstände gehe ich hier nicht explizit ein, doch setzt die Beispielarchitektur in diesem Artikel ein paar Annahmen voraus, die zwar für nicht wenige betriebliche Anwendungen zutreffend sein werden, aber daher noch lange nicht zwingend sind. Dies sind:

Die Anwendung wird von mehreren Benutzern gleichzeitig verwendet. Dies ist eine grundlegende Forderung, da sie uns dazu zwingt, über so Dinge wie Transaktionen, Zustandsverwaltung, Verteilung, Applikationsserver usw. nachzudenken.

Benutzerspezifischer Zustand wird in der Präsentationsschicht gehalten. Dies bedeutet für ein Web-UI die Nutzung der HTTP Session und für einen Rich Client auf Basis RCP oder Swing die Nutzung der entfernten Client JVM. Dadurch ist es unnötig, den Zustand zwischen Serveraufrufen z.B. in einer Stateful Session Bean oder der Datenbank zu halten. Langlaufende Geschäftsprozesse sollten gleichwohl durch Speicherung in der Datenbank unterstützt werden, denn diese Daten sollen sicherlich nicht verloren gehen, wenn die Session durch einen Ausfall z.B. des Clients zerstört wird.

Der zentrale Teil der Anwendung muss selbst nicht verteilt werden. Er wird auf einem Knoten, d.h. einer Instanz eines Applikationsservers entweder ganz oder gar nicht eingesetzt. Verteilung wird – wenn überhaupt – durch WebServices im Rahmen einer SOA zwischen verschiedenen Teilsystemen realisiert. Dadurch kann man auf Komponentenframeworks zur Verteilung, wie EJB etwa eines ist, verzichten und eine leichtgewichtige Servlet-Engine wie Tomcat einsetzen.

Die Speicherung der Daten erfolgt in einer relationalen Datenbank mit nicht trivialem Schema. Dies rechtfertigt den Einsatz eines OR-Mappers wie Hibernate. Anderenfalls würde ein Zugriff über JDBC möglicherweise ausreichen.

Die Benutzerschnittstelle wird in einem Web-Browser abgebildet. Daraus ergibt sich, dass man ein Web-Framework in einem Servlet-Container einsetzen sollte. Inbesondere setzt die Client-Server Kommunikation dann auf HTTP. Hätten wir es mit einem Rich Client zu tun, wäre RMI möglicherweise die erste Wahl, und daher würde statt einer Servlet-Engine eher ein EJB-ApplicationServer zum Einsatz kommen.

Die Integration zwischen Systemen erfolgt auf Basis von WebServices (SOAP/HTTP). Dies bedeutet zum einen, dass in der Integrationsschicht unserer Architektur WebServices erstellt werden und zum anderen, dass Aufrufe von Diensten in anderen Systemen über Adapter bewerkstelligt werden, die die aufgerufene Business Methode auf eine oder mehrere WebService Operations/Messages abbilden.

Man sieht, dass sich durch diese beispielhaften Annahmen sehr schnell die Entscheidungsspielräume bzgl. technischer Infrastruktur einengen. Wichtig ist, dass man diese Rahmenbedingungen formuliert und mit ihnen eine Infrastrukturauswahl begründet.
Das hilft auch dann, wenn sich Annahmen oder Vorgaben ändern oder als nicht tragfähig erweisen, und eine Korrektur der Entscheidungen erforderlich wird. Diese „Korrektur“ kann bedeuten, dass technische Infrastruktur oder Frameworks komplett ausgewechselt werden. Hat man dann bereits Geschäftslogik realisiert und diese stark mit Technologie gemischt, so wird das Auswechseln ein teurer Spaß. Die Trennung von Technologie und Geschäftslogik ist daher eine fundamentale Strategie, um den Schaden bei Eintritt solcher Risiken gering zu halten.

Die verschiedenen Kategorien einer Architektur

J. Siedersleben unterscheidet in [Sie 04] drei verschiedene Architekturen. An diese Unterscheidung lehnt sich folgende Begriffsbildung an:
Die *Architektur der technischen Infrastruktur* beschäftigt sich im wesentlichen mit der Auswahl und Festlegung der „großen“ Softwarekomponenten wie Datenbank, Applikationsserver, MessageBroker, ESB, BPEL-Engine, OR-Mapper, Komponenten-framework usw. sowie mit der Festlegung der Zielplattform i.S. Server-Hardware, Betriebs-system und Netzwerkinfrastruktur.
Die *technische Architektur* zeigt auf, wie diese Infrastruktur zusammenspielt und wie fachliche Usecases auf die verwendeten Technologien abgebildet werden. Dadurch stellt sie eine Schablone für das Design der häufig vorkommenden Usecases bereit. Mit dieser Schablone entdeckt ein Entwickler schnell die erforderlichen Elemente, in denen er die Geschäftslogik unterbringen muss.
Die *Facharchitektur* schließlich zerlegt die Fachlichkeit in kohäsive Komponenten und Module, so dass mithilfe der Designschablone

der technischen Architektur leicht sinnvolle Arbeitspakete abgeleitet werden können. Mangels umfangreichen Beispiels einer Fachdomäne werde ich hier wenig Konkretes zur Facharchitektur erklären.

Die Festlegung der technischen Infrastruktur ist für dieses Beispiel recht einfach:

- Sun JDK 1.6
- Apache Tomcat 5.5
- HSQL-DB 1.8.0.5

An Schnittstellen und Frameworks setze ich zusätzlich ein:

- Apache Axis2 1.2
- JPA 1.0, realisiert durch Hibernate 3.2.0.ga
- JAXB 2.0 realisiert durch Apache Jaxme2 0.52
- JSF 1.1 realisiert durch Apache MyFaces 1.1.5
- Spring 2.0.6 für AOP und DI

Im weiteren Text werde ich mich vor allem auf die technische Architektur konzentrieren und in welcher Weise die Geschäftslogik durch sie „aufgenommen“ wird.

Der Begriff Geschäftslogik

… (engl. business logic) bezieht sich auf diejenigen Teile einer Anwendung, in denen das „Fachwissen“ enthalten ist. Es ist allgemein anerkannt, dass die Implementierung der Geschäftslogik weitgehend von technischer Infrastruktur oder technischen low-level APIs getrennt werden sollte. Aber der Begriff Geschäftslogik ist zu grob, um dem gemeinten Code einen eindeutigen Platz in einer Softwarearchitektur zuzuweisen. Man muss daher genauer werden und folgende Teile unterscheiden:

Domänenlogik ist die Gesamtheit aller unabhängig von einzelnen Anwendungen zum Geschäft gehörenden Daten, Algorithmen und Regeln. Z.B. wird eine Bank immer ein Konto und dazu passende, feststehende Berechnungen haben, egal welche Prozesse über Anwendungen abgebildet werden.

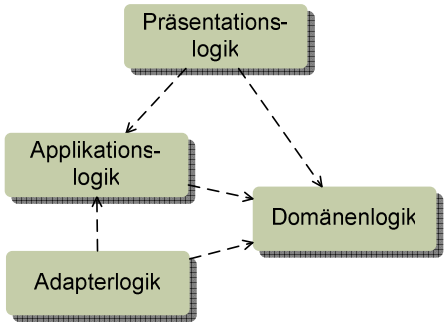
Applikationslogik ist die Gesamtheit aller prozessbedingten Daten, Algorithmen und Regeln. Anwendungssysteme bilden durch die Führung von Masken oder Protokolle in Schnittstellenvereinbarungen häufig definierte Schrittfolgen von Akteurstätigkeiten ab. Die Reihenfolge und Regeln hängen vom gewünschten Prozess und dem Akteur ab. Dabei meint „Akteur“ hier eine Benutzerrolle oder ein Fremdsystem.

Man kann in größeren Systemen häufig den Versuch beobachten, die Steuerung und die steuerungsrelevanten Daten in Workflow-Definitionen oder Regelsätze auszulagern und eine Workflow- bzw. Rule-Engine einzusetzen. Insbesondere auf Ebene einer Systemlandschaft möchte man die Interaktionen zwischen Systemen mittels zentraler Prozessbeschreibungen, die direkt aus Geschäftsprozessen hergeleitet werden, steuern. BPEL hat dort als Standard neue Bewegung hineingebracht. Dabei muss man sich aber vor Augen halten, dass Applikationslogik nicht ausschließlich Prozesslogik ist, und dass überdies Prozesse ganz unterschiedlichen Ebenen angehören können. Der feinstgranulare Prozess ist z.B. die Ausführung einer Java-Methode, der gröbstgranulare ist ein Geschäftsprozess. Jede Form von Prozess mit dem gleichen „Werkzeug“ angehen zu wollen, führt sicher zu problematischen Lösungen, weshalb man sich genau ansehen sollte, welche Ebene man mit welcher Technologie abbilden möchte.

Präsentationslogik ist die Gesamtheit aller Daten, Algorithmen und Regeln, die an der (Benutzer-)Schnittstelle verwendet werden, um den Anwender optimal zu unterstützen. Diese Logik wird häufig in die View, also in JSPs oder Fenster und Dialoge hineingewoben, was den Nachteil hat, dass sie nicht ohne die entsprechende Infrastruktur (d.h. GUI oder Servlet-Engine nebst Browser) testbar ist, und wenn, dann nur über spezielle Werkzeuge, deren Einsatz teuer ist. Da diese Logik aber häufig komplex ist, weil nur so eine hohe Bedientauglichkeit erreicht wird, sollte sie leicht automatisiert testbar sein. Es wäre also wünschenswert, die Präsentationslogik getrennt von der View-Technologie testen zu können.

Adapterlogik ist die Gesamtheit aller Transformations- und Abbildungsregeln sowie das implementierte Protokollwissen, um eine Integration eines Fremdsystems über eine vereinbarte Schnittstelle zu erreichen. Diese Logik stellt die fachliche Entkopplung sicher, soweit dies möglich ist. Damit soll vermieden werden, dass dem Fremdsystem eigentümliche „Spielregeln“ in das eigene System hineindiffundieren und damit eine schwer nachvollziehbare Kopplung entsteht.

Es gibt eine einzuhaltene Ordnung von Abhängigkeiten auf diesen Teilen:



Durch diese Ordnung versucht man, Einflüsse aus instabilen Bereichen eines Systems auf die Domänenlogik zu vermeiden. Es ist natürlich trotzdem möglich und teilweise unvermeidbar, dass Konzepte aus einem Fremdsystem oder der Benutzerschnittstelle in subtiler Weise in die Domänenlogik einsickern. Der Preis ist dann weniger langlebiger Code, d.h. der Bedarf für größere Änderungen wird früher entstehen, weil die Verständlichkeit und Erweiterbarkeit nicht mehr im erforderlichen Maß gegeben sind.

Nun, da wir etwas genauer sehen, was sich hinter dem stark strapazierten Begriff „Geschäftslogik“ verbirgt, können wir später jedem der vier Teile einen geeigneten Platz suchen.

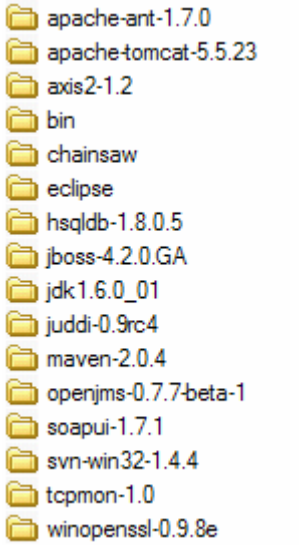
Eine Softwareentwicklungsumgebung
… hat die Aufgabe, ein Entwicklungsteam mit allen erforderlichen Werkzeugen zu versorgen, damit eine Software effizient und qualitätsgerecht für die Zielplattform erstellt werden kann.
Man kann zwischen zentralseitigen Komponenten einer SEU (z.B. Versionskontrollsystem, Wiki, Bugtracking-Datenbank, Continuous Integration / Build Server, Datenbankinstanzen, Testumgebungen usw.) und dezentralen Komponenten für den Entwicklungsarbeitsplatz (IDE, Modellierungswerkzeug, Quellcode-Generator, Test- und Analysetools, Ablaufumgebungen wie z.B. ein Applikationsserver usw.) unterscheiden. Hier wende ich mich nur der dezentralen Umgebung zu, der man sehr gut ein Default-Verzeichnislayout, das für alle Entwickler eines Projekts gleich ist, zugrunde legen kann. Sowas muss man nicht unbedingt einführen, aber bei einem größeren Projekt erspart man sich auf diese Weise viele Tage Arbeit, die sonst jeder Entwickler mit der Einrichtung seiner persönlichen Umgebung verbringen müsste.

Alles beginnt mit einem Wurzelverzeichnis und darin befinden sich dann:

- apps
- docs
- projects
- ssl
- workspaces

sowie ein Skript zum Setzen von Umgebungsvariablen (setenv).

Im apps Verzeichnis befinden sich alle Software-Werkzeuge und Produkte, so wie sie „aus der Tüte“ kommen und ggf. durch eigene Konfiguration angepasst. Dazu gehört z.B. eclipse, jdk, tomcat, maven, ant, hsqldb, openjms, soapui.



Mitten darunter befindet sich insbesondere auch das bin Verzeichnis, in dem Skripte abgelegt sind, die den Umgang mit der Umgebung erleichtern, z.B. das Starten und Stoppen von DB oder Tomcat. Eclipse selbst wird z.B. auch durch ein Skript gestartet, mit dem der Workspace, das JDK und noch Speichereinstellungen gesetzt werden. Durch zusätzliche Desktop-Links kann jeder Entwickler den Zugriff auf Eclipse und andere Werkzeuge der Umgebung beschleunigen.

Im docs Verzeichnis legt man allgemeine, nicht projektbezogene Referenzen und Anleitungen ab, die sich anderenfalls jeder Entwickler selbst aus dem Netz ziehen müsste. Dazu zählen z.B. die Spring Referenzdokumentation, verschiedene Java API Docs, W3C und OASIS Standards, eine Anleitung zu Maven 2 usw. Projektbezogene Spezifikationen, Anleitungen und Dokumentationen gehören natürlich zu den jeweiligen Maven-Projekten und Modulen unter Versionskontrolle.

Das projects Verzeichnis ist direkt nach Einrichtung der Umgebung natürlich leer. Hier liegen später die Wurzelverzeichnisse für die ausgecheckten Projekte.

Das ssl Verzeichnis enthält zu Testzwecken Zertifikate und Schlüssel für eine einfache PKI (public key infrastructure). Die dort abgelegten Keystores und Zertifikate können für Projekte und Testumgebungen verwendet werden.

Und im workspaces Verzeichnis befinden sich schließlich ein oder mehrere Eclipse Workspaces. Auch dieses ist frisch nach Installation nahezu leer.

Ein Skript (setenv), das im Wurzelverzeichnis der Umgebung liegt, setzt alle erforderlichen Umgebungsvariablen, damit Skripte aus apps/bin und den verschiedenen Werkzeugen leicht gestartet werden können und alles finden, was sie brauchen.

Eine solche Umgebung ist natürlich auch Bestandteil des Konfigurationsmanagement. Deshalb muss sie zwar nicht in einem Versionskontrollsystem abgelegt werden, aber die zentrale Versionierung und Archivierung ist wichtig, um auch Jahre später frühe Projektreleases zusammen mit der passenden Werkzeuglandschaft reaktivieren zu können. Denn was nutzt Quellcode von vor fünf Jahren, den niemand mehr zusammenbauen kann, weil die Werkzeuge fehlen?

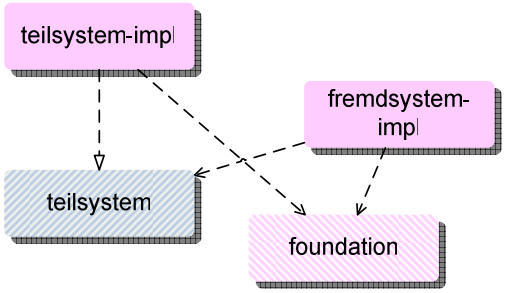
Noch ein paar Sätze zu Maven. Maven ist für den Java Build im Vergleich zu Ant was funktionale Programmiersprachen im Vergleich zu imperativen Sprachen sind. In Maven werden Projekte und Module zentral in einer Datei pom.xml (POM=project object model) beschrieben, daraus ergeben sich die Buildschritte für Maven automatisch. In Ant hingegen wird jeder Buildschritt explizit "programmiert". Letzteres ist insbesondere in Projekten mit mehreren Teams und Subsystemen höchst anfällig für unnötige Doppelarbeit, divergierende Subsystemverzeichnisstrukturen und mithin unterschiedliche Buildverfahren.
Maven erzwingt in sanfter Weise eine Gleichartigkeit durch "convention over configuration", d.h. um ein vom Maven-Default abweichendes Verhalten zu erhalten, ist zusätzlicher Konfigurationsaufwand erforderlich. Je näher ein Softwaresystem am Maven-Default ist, desto kleiner bleibt also der Konfigurationsaufwand.
Darüberhinaus besitzt Maven sehr wirksame Konzepte zur Projekt-Modularisierung und zur Verwaltung von Systemabhängigkeiten zwischen organisationseigenen Entwicklungsergebnissen und insbesondere 3rd-Party Libraries organisationsfremder Hersteller.
Ein zentrales Element der Abhängigkeitsverwaltung sind sogenannte Repositories, die Libraries (i.a. JARs) enthalten. Das öffentliche Default-Repository ist http://repo1.maven.org/maven2, daneben gibt es noch alternative Orte, von denen ibiblio.org wohl der bekannteste ist. Repositories sorgen dafür, dass Libraries nicht mehr direkt im Projekt abgelegt werden, stattdessen verweisen die POM-Dateien nur noch auf Libraries. Maven sorgt im Rahmen eines Build dafür, dass die benötigten Libraries (und übrigens auch Maven spezifische Plugins) aus einem zentralen Repository heruntergeladen und in einem lokalen, user-spezifischen Repository-Cache unter HOME/.m2 abgelegt werden. Maven lässt sich gut in Eclipse und andere IDEs integrieren.

Insgesamt bietet Maven auf viele Fragen zur technischen Organisation einer komplexen Mehrteam Entwicklung weitreichende Antworten. Der Einsatz von Maven sollte daher beim Start einer größeren Neuentwicklung immer in Betracht gezogen werden.

SOA Schnittstellen und die Organisation von Maven Projekten im Konfigurationsmanagement

Wer sich aufmerksam mit den Regeln zum Package Design in [Mar 03] auseinandersetzt, wird feststellen, dass die Einhaltung der Prinzipien zu Instabilität und Abstraktion immer dazu führt, dass stabile Schnittstellen von instabilen Implementierungen getrennt werden. Dieses Muster findet sich schon auf Ebene von Java Interfaces vs. Classes, nach [Mar 03] ebenso auf Package-Ebene und konsequenterweise auch auf Ebene unternehmensweiter Systemlandschaften in einer SOA wieder. Ein Verdienst des SOA Stils ist es, auf dieser Ebene eine große Bedeutung auf Schnittstellen zu legen. Im Falle von SOAP WebServices als „Infrastruktur“ der SOA sind es WSDL-Dateien, in denen die Aufrufsyntax von Diensten und via XML-Schema das Format der auszutauschenden Daten spezifiziert wird. Die Semantik muss heute immer noch in zusätzlichen Dokumenten beschrieben werden. Schnittstellen werden damit als „eigenständige Lebewesen“ wahrgenommen: sie erhalten Versionsnummern und man diskutiert die Inhalte von WSDL und XML-Schema statt in Implementierungsdetails abzugleiten.
Damit kommt ihnen logischerweise aber auch eine eigenständige Bedeutung im Konfigurationsmanagement zu. Schnittstellen werden daher am besten unter Versionskontrolle als ebenbürtige Maven Projekte behandelt, die im wesentlichen die WSDL, das XML-Schema, ein Dokument zur Service-Semantik und möglicherweise noch

daraus generierte Java-Klassen enthalten. Nutzende Maven Projekte deklarieren im POM eine Abhängigkeit genauso wie das implementierende Projekt, aus dem beim Build letztlich die deployfähigen WebServices herausfallen. Dadurch entsteht wieder genau das gesunde Abhängigkeitsmuster, in dem instabile Implementierungen auf stabile Schnittstellen zeigen:



Wir finden mithilfe dieser Trennung der Schnittstellen von ihren Implementierungen wenigstens zwei Klassen von Maven Projekten: WebService Schnittstellenprojekte (teilsystem) und Implementierungsprojekte, wobei wir letztere noch weiter zerlegen können in Projekte zwecks Bereitstellung querschnittlicher technischer Lösungen (foundation) und Projekte, die konkret deployfähige Einheiten wie WARs oder AARs erzeugen (*-impl).

Vom Architekturstil bis zu Maven Modulen

Soweit zu noch erklärungswürdigen Konzepten und der Struktur der Entwicklungsumgebung. Jetzt geht's zum Grobentwurf der Implementierung eines einzelnen Systems. Wir starten mit folgender Aufteilung im Stile einer *Schichtenarchitektur* in

- Präsentationsschicht
- Integrationsschicht
- Fachkern
- Datenhaltung

Für die weiteren Erläuterungen brauchen wir zwei Systemnamen. Das *teilsystem* ist das System, dessen Architektur wir hier betrachten. Das *fremdsystem* ist ein Stellvertreter für alle möglichen anderen Systeme, mit denen das teilsystem kommuniziert.

Um die Maven Module des teilsystems zu finden, die wir hier brauchen, muss man sich zuerst überlegen, welche Build-Ergebnisse letztlich erzeugt werden müssen. In diesem Fall sind es zwei: ein WebArchive (WAR) für die Web-UI und ein AxisArchive (AAR) für die Bereitstellung der Dienste über SOAP/HTTP. Zunächst ergeben sich daraus zwei Module:

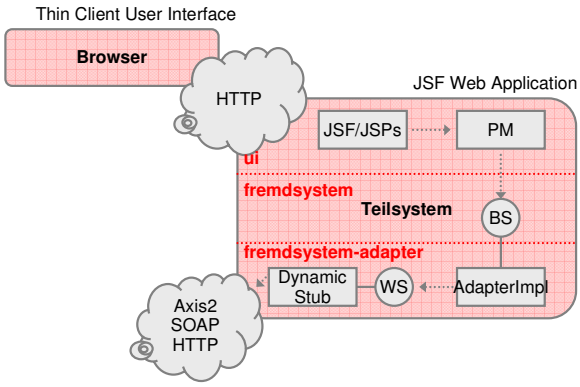
- teilsystem-ui und
- teilsystem-ws.

Betrachten wir jedes einzelne Modul genauer: ws stellt die Domänenlogik über WebServices bereit, aber die Implementierung sollte keinesfalls nur durch WebServices hindurch testbar sein. Es empfiehlt sich vielmehr, ein eigenständiges Modul für den Fachkern zu erfinden:

- teilsystem-core

Dieses enthält dann die Java Business Interfaces, die unabhängig von der Bereitstellung über WebServices sind, sowie die Java Geschäftsobjekte und Service-Implementierungen als POJOs und die Datenzugriffsschicht, die die Abbildung auf die DB verantwortet.
Wird aus dem Fachkern heraus ein Aufruf eines Fremdsystems erforderlich, so ist ein Adapter nötig. Dieser implementiert eine Java Schnittstelle, die aus Sicht des teilsystem-Fachkerns technisch wie ein Teil der eigenen Geschäftslogik aussieht, tatsächlich aber einen WebService Aufruf durchführt. Das führe ich weiter unten etwas genauer aus.

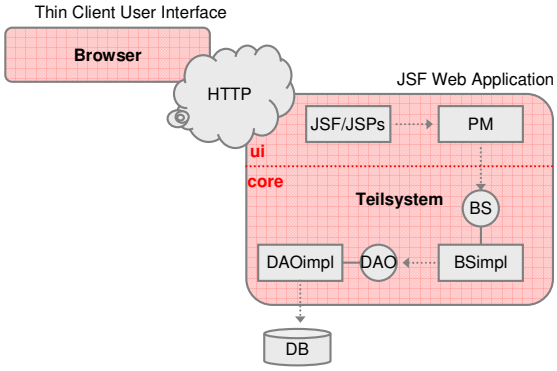
Das Web-UI im Modul ui kann man in zwei Geschmacksrichtungen beim Build behandeln: Erstens kann man das Web-UI als sehr dünne Web-Schicht betrachten, die, sobald sie die Ausführung von Domänenlogik anstoßen muss, selbst WebServices aufruft. In diesem Fall wird das Modul core nicht in das WAR eingebaut. Dieser Ansatz ist dem SOA Gedanken sehr nah, kostet aber natürlich eine zusätzliche Indirektion, denn es kommt ein WebService Adapter zwischen die Oberfläche und die verwendete Geschäftslogik. Alle Aufrufe von der dünnen UI aus gehen also als XML-Nachrichten über das Netzwerk.



Die UI selbst muss über einen Adapter die Geschäftslogik aufrufen.

Zweitens kann man den Fachkern im core Modul per Build in das WAR einbauen. Die saubere Trennung bleibt zwar erhalten, denn der Fachkern ist ja in einem eigenen Modul, aber es entsteht der Nachteil, dass bei einer

Änderung des Fachkerns aus Sicht des Betriebs immer zwei Systeme neu deployt werden müssen: die Web-UI und die WebService-Anwendung. Wird das vergessen, so entstehen u.U. gefährliche Inkonsistenzen, weil zwei verschiedene Versionen des gleichen Fachkerns produktiv sind.



Der Fachkern wird direkt „unter“ die UI gebaut.

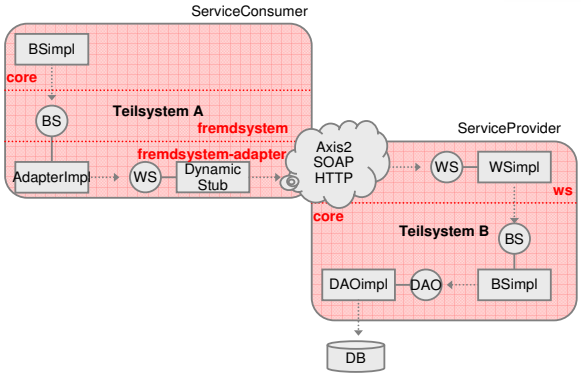
Um zu einer Entscheidung zu kommen, ist hier eine Abwägung zwischen erhöhtem Managementaufwand während des Betriebs und erhöhten Entwicklungskosten sowie verminderter UI-Performanz erforderlich.

Wie auch immer: im ersten Fall (WAR ohne core) sind Adapter erforderlich, denn die WebService Anwendung, die den Fachkern kapselt, stellt sich wie ein Fremdsystem gegenüber der Web-UI dar. Und im zweiten Fall (WAR enthält core) können die Klassen aus der UI-Schicht direkt auf Java Business Interfaces des Fachkerns zugreifen.

Bisher habe ich drei Module (ui, ws und core) hergeleitet und von fremdsystem Adaptern gesprochen. Wie werden letztere nun in Module organisiert?
Ganz einfach: zu jedem fremdsystem, das aus dem teilsystem angesprochen wird, gibt es ein Modul

- fremdsystem-adapter.

Dieses tritt gegenüber dem jeweiligen fremdsystem als Service Consumer auf und implementiert in dem teilsystem, in dem es sich befindet, dort liegende Java Business Interfaces.

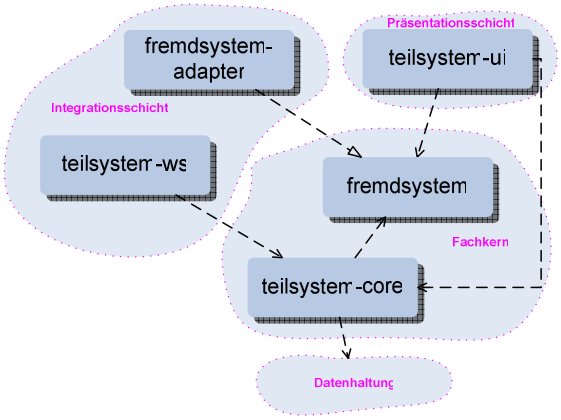


Adapter zu anderen Systemen rufen z.B. andere WebServices auf.

Eine Frage bleibt aber noch: wo genau liegen diese Java Business Interfaces? Bisher haben wir als Kandidaten das core-Modul benannt. Im Falle einer WebService-Anwendung, die i.W. aus ws und core besteht, macht das ja auch Sinn. In einer dünnen Web-UI ist core aber nicht vorhanden, dort gibt es nur ui und die benötigten fremdsystem-adapter. Man könnte jetzt eine Abhängigkeit von ui auf die Adapter Module beschließen oder – anders herum – die Adapter die infrage kommenden Java Business Interfaces, die in ui liegen, implementieren lassen. Wie man’s auch dreht, gut ist beides nicht, da Implementierungen untereinander keine Abhängigkeiten besitzen sollten. Die Lösung ist ein weiteres Modul

- fremdsystem,

das nur die aus dem Fremdsystem genutzten Java Business Interfaces wie eine Facade enthält. Die vollständige Maven Modulstruktur eines Maven Implementierungsprojekts sieht dann so aus:



Module erwecken die Schichtenarchitektur zum Leben.

Um nun im Rahmen der gefundenen Module Pakete zu definieren, in die wir den Fachcode gliedern, benötigen wir eine Idee von den technischen Komponenten, die unsere technische Architektur zur Abbildung von Fachlichkeit anbietet. D.h. man braucht bereits

das Wissen zu den üblicherweise zu lösenden Problemen, deren Lösungsmustern und der zur Realisierung eingesetzten Technologien und Produkten, um ein tragfähiges Package Design zu definieren.

In den folgenden Abschnitten wird jedes Modul einzeln behandelt, so dass neben den nötigen technischen Lösungen je Modul auch ein Package Design entsteht, in das sich der jeweils passende Teil der Geschäftslogik einsortiert. Dabei wird großer Wert darauf gelegt, dass die jeweilige Geschäftslogik möglichst nicht durch die Nutzung technischer low-level APIs „verschmutzt" wird.

Vorweg behandle ich allerdings noch die …

Lösung einiger typischer querschnittlicher Probleme

Nach einer knappen Erläuterung zu AOP nehme ich mir aus der Liste der möglichen Probleme folgende heraus:

- Protokollierung
- Transaktionsbehandlung
- Instrumentierung
- Konfiguration

Aspekt-orientierte Programmierung

Denkt man heute über die Lösung querschnittlicher Probleme (engl. cross-cutting concerns) nach, dann sollte man Aspekt-orientierte Programmierung (AOP) in Betracht ziehen.

Ein *Aspekt* eines Software-Systems ist eine querschnittliche Lösung wie Tracing, Autorisierungsprüfungen, Transaktionsbehandlung, Monitoring und ähnliches. Die Implementierungen der Lösungen werden ohne AOP meist über die gesamte Anwendung verteilt. Dadurch werden Änderungen dieser Aspekte teuer, da sie sich der starken Verteilung entsprechend „flächendeckend" auf den Code auswirken. Mit AOP hingegen werden Aspekte einmal programmiert und dann deklarativ auf bestimmte Stellen im System (Einstiegspunkte, engl. Joinpoints) angewendet.

„Spring AOP" ist die Spring Implementierung für aspektorientierte Programmierung. In Spring gibt es zur AOP drei Elemente: *Pointcut*, *Advice* und *Advisor*. Ein Pointcut definiert eine Untermenge von Einstiegspunkten aus den möglichen Joinpoints (=die Gesamtmenge aller Einstiegspunkte des Systems). Ein Advice ist eine Java-Klasse, von der eine bestimmte Methode aufgerufen wird, wenn ein Einstiegspunkt bei der Ausführung erreicht ist. Ein Advisor kombiniert einen Pointcut mit einem Advice, so dass konkret für ausgewählte Joinpoints das Anwendungsverhalten bestimmt ist. Mit anderen Worten: ein Advisor implementiert einen Aspekt.

Pointcuts werden in der Spring Konfiguration des Systems definiert, der Advice wird in Java geschrieben und die Zusammenführung von Pointcut und Advice wird dann wieder in der Konfiguration deklariert.

In den nun folgenden Absätzen werde ich – wo angebracht – AOP als Mittel zur Umsetzung jeweils erwähnen.

Protokollierung

… gliedert sich üblicherweise in zwei Teilaspekte: Logging und Tracing. Logging meint dabei das Schreiben von Meldungen, die während des Betriebs der Anwendung gewünscht sind. Darunter gehören sicherlich auch Fehlermeldungen und Warnungen. Daneben können aber auch Zeitmessungen oder sicherheitsrelevante Ereignisse (An- und Abmeldung, Ausführung sensibler Operationen) mitgeschrieben werden.
Tracing dagegen dient der Programmablaufverfolgung meist zwecks Fehleranalyse. Traces können sehr umfangreich sein, weshalb Tracing im Betrieb üblicherweise deaktiviert wird.
Wir verwenden als Log-Framework Log4J, gekapselt durch die Schnittstelle Apache Commons Logging.
Zum Tracing wäre es denkbar, mittels AOP z.B. alle Methoden-Entries und -Exits mitzuschreiben, allerdings ist der Wert derart erzeugter Meldungen m.E. nicht sehr hoch.
Daher wird Logging und auch Tracing i.a. ohne AOP explizit programmiert. Dabei wird Tracing von Logging durch die Loglevel TRACE und DEBUG unterchieden, ab INFO heißt es also „Logging".
Es gibt spezielle Logger, hier z.B. einen Sicherheitslogger und einen Performanzlogger, die durch zentral deklarierte symbolische Konstanten im Java Code angesprochen werden können. I.d.R. jedoch benutzt jede Klasse einen Logger, der ihren vollqualifizierten Namen trägt. Da Log4J eine Logger-Hierarchie mittels Punkt-Notation in Loggernamen aufbaut, kann man Logausgaben genau entlang des Package-Designs steuern. Ist das sauber entworfen, so hat man für gewöhnlich schon alles, was man braucht.
Insbesondere werden Logger nicht durch Spring injiziert, sondern statisch in jeder Klasse deklariert. DI ist hier meiner Ansicht nach unsinnig, da sowieso jede Klasse ihren eigenen Logger braucht.

Transaktionsbehandlung

Für praktisch jede betriebliche Anwendung spielen Transaktionen eine zentrale Rolle, weil eigentlich immer Mehrbenutzerbetrieb möglich sein soll, und Transaktionen eben das Mittel sind, die negativen Auswirkungen gleichzeitigen Zugriffs auf gemeinsame Ressourcen (Daten) zu vermeiden.
Ich fange jetzt nicht im „Urschleim“ bei den ACID-Eigenschaften an, sondern wende mich direkt den konkreten Entscheidungen zu, die man in einem Java-System treffen muss.

Die möglicherweise an Transaktionen beteiligten Systeme wie MessageBroker, EIS oder Datenbanken werden im Transaktionssprachgebrauch einheitlich als ResourceManager bezeichnet. Die erste Frage, die man sich stellen muss, ist: Hat ein teilsystem nur zu einem oder zu mehreren ResourceManagern eine Verbindung? Ist letzteres der Fall, muss man sich mit der Konsistenzwahrung über die beteiligten ResourceManager hinweg auseinandersetzen. Eine vergleichsweise einfache Lösung wären in diesem Fall noch verteilte Transaktionen mithilfe von JTA und einem Two-Phase-Commit, wenn man das seltene Glück hat, dass alle ResourceManager XA-fähig sind. Ohne XA ist Kreativität gefragt, da man i.d.R. anhand der fachlichen Szenarien untersuchen muss, welche Probleme auftreten können, und wie man jeweils das Schlimmste verhindern kann.

Hier indes haben wir das Glück, dass wir je teilsystem nur von einer Datenbank ausgehen, was vieles einfacher macht. Es gibt im Enterprise Java Umfeld grundsätzlich drei Wege:

- Transaktionen explizit und verbindungsbezogen nutzen.
- Transaktionen explizit mit dem JTA API programmieren.
- Deklarativ gesetzte Container Managed Transactions (CMT) nutzen.

Ich kann die Diskussion hier kurz halten: ich bevorzuge eindeutig den letztgenannten Weg, weil er weniger Codeverschmutzung verursacht, Transaktionskontexte bequem an weitere Methoden propagiert werden können, und dies alles deutlich schneller und fehlerfreier geht als von Hand programmiert.

In einem EJB-Container liegen die Transaktionsgrenzen CMT genau beim Aufruf einer Bean-Methode und werden deklarativ per Deployment-Deskriptor oder Annotation eingestellt. Spring unterstützt so etwas ebenfalls und bietet dabei die Möglichkeit, Transaktionsgrenzen zentral mittels AOP auf definierte Pointcuts zu legen. Damit ist es ein leichtes, die Grenzen auf alle Methoden der Klassen in bsimpl zu legen oder stattdessen z.B. die Methoden der WS-Implementierungen zu verwenden, ohne einzelne Klassen oder Deployment-Deskriptoren ändern zu müssen. Die eigentliche Konfiguration des Verhaltens beim Aufruf einer Methode passiert in einem AOP Advice, in dem der Methodenname (oder Wildcards) mit Attributwerten für IsolationLevel, Propagation, Read-Only, TimeOut und die Rollback-Auslöser assoziiert wird.

Wie findet man in einem teilsystem die Transaktionsgrenzen? In einem EJB-System ist es einfach: die Bean-Methoden stellen selbstverständlich auch die Transaktionsstart und –endepunkte dar. In einem System mit Spring haben wir die Wahl, wo wir die Grenzen ziehen wollen. Die Frage, die man beantworten muss, ist: welche Einheiten stellen aus fachlicher Sicht „units of work“ dar, für die die ACID-Eigenschaften gelten sollen?
In einem teilsystem sind es einerseits die Benutzeraktionen, die als Einheit gelten sollen, und andererseits die WebService-Operationen. Bei einer dünne Web-UI, die keinen Fachkern und auch keine direkte Verbindung zu einem ResourceManager besitzt, stößt man hier auf eine „Unschärfe“. Die Web-UI ruft bei einer einzelnen Benutzeraktion u.U. mehrere WebService-Operationen von schlimmstenfalls verschiedenen Fremdsystemen auf. Das, was aus Benutzersicht als Einheit wirkt, wird technisch in verschiedenen unabhängigen Transaktionen verarbeitet und kann bei vereinzelten Fehlschlägen zu einem fachlich inkonsistenten Zustand führen, der freilich nur bei Betrachtung der Daten mehrerer Teilsysteme gleichzeitig auffällt. Was sich hier auftut, ist wieder das Konsistenzwahrungsproblem, diesmal allerdings auf Ebene von Teilsystemen und nicht innerhalb eines Teilsystems auf Ebene verschiedener ResourceManager. Und in diesem Fall ist bei heutigem Stand entweder Kreativtät gefragt oder der Versuch, Technologien wie WS-AtomicTransaction und WS-Coordination zu nutzen. Eine einfache Standardlösung gibt es heute nicht. Daher ist es im Fall der dünnen Web-UI also sinnlos, die Action-Methoden, über die eine Benutzeraktion ausgelöst wird, als Transaktionsgrenze zu wählen. Damit bleiben hier nur die Methoden der WebService-Implementierungen, die den WS-Operationen entsprechen, übrig.

Der IsolationLevel steuert, wie stark eine Transaktion von anderen abgegrenzt werden soll. Je stärker die Abgrenzung und damit Konsistenz, desto weniger performant und skalierbar wird die Abarbeitung. Man hat die

Wahl zwischen vier Stufen (in der Reihenfolge niedrige zu hohe Isolation):

- ReadUncommitted
- ReadCommitted
- RepeatableRead
- Serializable

Auch hier will ich nicht in eine längliche Diskussion gehen: die erste und letzte Stufe fallen i.d.R. raus, weil die negativen Auswirkungen bzgl. Konsistenz bzw. Parallelität zu gravierend sind. In der Praxis sollte man per default ReadCommitted verwenden, denn fachlich ist RepeatableRead selten erforderlich und wird darüberhinaus z.B. von einigen Datenbanken gar nicht unterstützt.

Die Propagation regelt, unter welchen Umständen tatsächlich eine neue Transaktion begonnen und durch ein Commit geschlossen wird. Die möglichen Werte sind:

- Required
- Mandatory
- RequiresNew
- Supports
- NotSupported
- Never

Da eine verständliche Beschreibung der Bedeutungen den Rahmen dieses Artikels sprengen würde, verweise ich hier auf die EJB 3-Spezifikation, Abschnitt 13.6.2 [EJB 3.0]. Ein pragmatischer Ansatz ist, immer Required zu wählen, und im Falle, dass man andere Eigenschaften benötigt, einen der anderen Werte zu verwenden.

Zu guter Letzt noch das Thema „langlaufende Transaktionen“. Über solche muss man sich dann Gedanken machen, wenn eine aus Anwendersicht zusammenhängende Tätigkeit, für die dementsprechend alle oder auch nur Teile der ACID-Eigenschaften gelten sollen, viele Minuten, Stunden oder gar Tage in Anspruch nimmt. Es ist nicht ratsam, solche Langläufer über technische Transaktionen abzuwickeln, da dies mit großer Wahrscheinlichkeit die Parallelität der Abarbeitung und mithin die Skalierbarkeit sehr negativ beeinflusst. In dem hier betrachteten teilsystem werden daher die mit der langlaufenden Tätigkeit verbundenen Daten transient in der HTTP-Session gehalten und erst bei Abschluss über entsprechende Dienste in die Datenbank geschrieben.

Instrumentierung

… ist grundsätzlich ein Klassiker für den Einsatz von AOP. Hier ist mit Instrumentierung das Anbringen von Zeitmesstrecken gemeint, um die Verweildauern in den verschiedenen Schichten des Systems festzustellen. Dafür kann man JAMon einsetzen und einen kleinen Advice implementieren, den man dann z.B. auf alle Methoden der daoimpl-Klassen anwendet. Dadurch sammelt man Daten wie Aufrufhäufigkeit und verbrauchte Zeit in sogenannten Monitoren. Die Frage, wie die Inhalte der Monitore verfügbar gemacht werden, muss natürlich auch beantwortet werden. Es stellt keine gute Lösung dar, direkt im Advice in eine Logdatei zu schreiben, denn das würde das Zeitverhalten des Systems völlig verzerren. Der einfachste vernünftige Weg besteht hier darin, mit Spring einen TimerTask einzurichten und einen speziellen Logger zu verwenden. Der TimerTask wird z.B. alle 30 Sekunden angesprochen, und liest dann die Monitore aus und erzeugt die Ausgabe. Auf diese Weise sind die zwei Belange „Zeiten sammeln“ und „Zeiten protokollieren“ sinnvoll voneinander uns insbesondere vom Fachcode getrennt.

Konfiguration

In Systemen, die Spring verwenden, fällt es besonders auf, jedoch gibt es genau genommen schon immer verschiedene Bedeutungen hinter der Bezeichnung „Konfiguration“. Aus Benutzersicht bedeutet Konfiguration sicher etwas ganz anderes als aus Entwicklungssicht, obwohl z.B. Deployment-Deskriptoren oder die web.xml auch als Konfiguration bezeichnet werden, und ihr Inhalt zur Entwicklungszeit geschrieben und später – außer durch die Entwicklung – nicht mehr geändert wird. Die wichtige Frage bei Konfiguration ist also immer: Wer sollte was wann ändern können?

Die Akteure sind dabei: Entwicklung, Installation/Betrieb und Benutzer. Beim Betrieb kann man zusätzlich unterscheiden, ob die Einstellungen vor Deployment, beim deployten aber heruntergefahrenen System oder zur Laufzeit des Systems vorgenommen werden können und auch wirksam werden.

Zur Entwicklungszeit besitzen wir mit Spring, der services.xml, web.xml, jsf-config.xml usw. alle Stellschrauben, die man sich wünschen kann. Mithilfe von Springs PropertyPlaceholder kann man darüberhinaus innerhalb der Spring-Konfiguration sehr gut Properties einlesen und damit Einfluss auf die Spring-Konfiguration nehmen. Das funktioniert also auch im Betrieb mit dem heruntergefahrenen System, da Spring die Konfiguration nur zum Start liest und auswertet. Log4J ist immerhin in der Lage, System-Properties auszuwerten, und im Tomcat kann man im Kontext überwachbare Dateien eintragen, bei deren Änderung automatisch der ganze Webkontext neu gestartet wird.

Möchte man zur Laufzeit ohne Neustart Änderungen durch Properties oder XML-Konfiguration durchführen können, die kurze

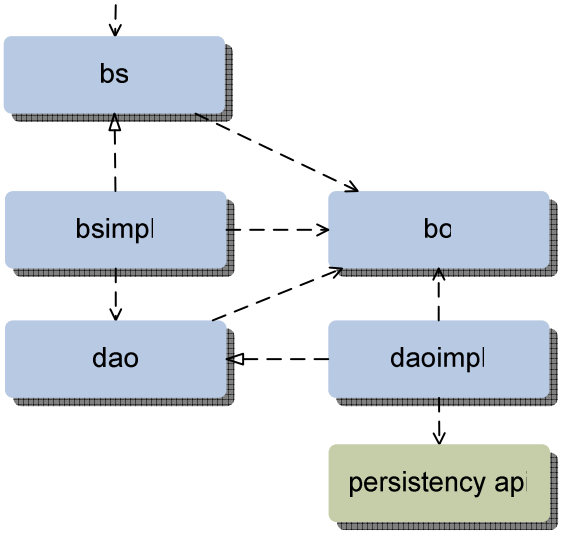
Zeit später wirksam werden, dann benötigt man einen TimerTask, der die Datei überwacht und bei festgestellter Änderung eine Rekonfiguration anstößt. Das ist mit Spring leicht zu realisieren. Ein ganz anderer Weg für die Anpassung von Parametern dagegen ist die Verwendung von JMX, auf den ich hier nicht weiter eingehen möchte.
Die Administration von Fachkonstanten wird i.d.R. durch Anwender mit besonderen Rollen durchgeführt. Als Ablageort für solche Konfigurationsparameter würde ich – wenn möglich – immer eine Datenbanktabelle vorziehen. Je nach Häufigkeit und Komfortanspruch kann die Aktualisierung durch ein SQL-Tool, einen Uploader oder sogar durch dafür vorgesehene Masken erfolgen.
Will schlussendlich der Benutzer seine persönlichen Einstellungen ändern, so wird er dazu sicherlich eigene Masken vorfinden. Die Werte landen dann aber üblicherweise auch in der Datenbank, denn hier wird man sich serverseitig nicht auf den Eiertanz mit benutzerbezogenen Property-Dateien einlassen. Konfiguration von Benutzereinstellungen und Fachkonstanten ist also ein handfester Bestandteil eines Fachkonzepts und benötigt entsprechende Usecase Beschreibungen.

core: Der Fachkern und persistente Geschäftsobjekte mit JPA

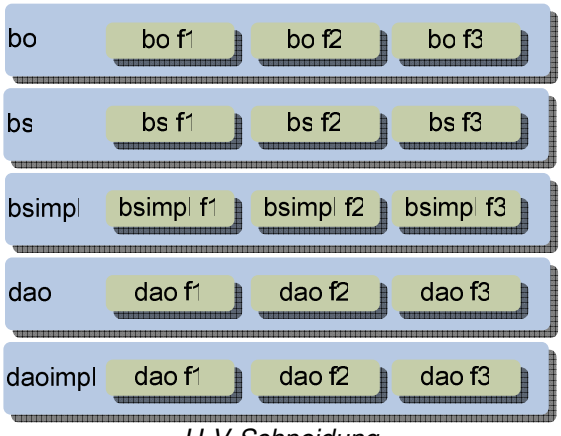
Der Fachkern im teilsystem-core besitzt ein Paket *bs*, in dem die Java Business Interfaces liegen, die die Business Services beschreiben, die der Fachkern bietet. Der Zugriff auf Funktionalität erfolgt nur über diese Schnittstellen.
Implementiert werden sie von Klassen im Paket *bsimpl*. Diese greifen zum Lesen und Schreiben von persistenten Objekten auf DataAccessObjects (DAOs) zu, deren Schnittstellen in *dao* liegen. Die passenden Implementierungen befinden sich in *daoimpl*. Dies ist die konkrete Datenzugriffsschicht, die die Verwendung der Technologie JPA weitgehend kapselt. Das führe ich unten etwas genauer aus.
Alle aus den vorgenannten Paketen verwendeten Geschäftsobjekte liegen im Paket *bo*. Sie sind z.T. persistent und enthalten die mit ihnen untrennbar verbundene Domänenlogik. Die Service-Implementierungen in bsimpl stellen hingegen die Facade und Feinsteuerung dar, um einheitlich auf Daten, Berechnungen und Prüfungen zuzugreifen.
Wir können sehr leicht die „Building Blocks“ eines Model Driven Design nach [Eva 03] den Paketen zuordnen:

- Entities, ValueObjects und Aggregates: bo
- Factories, Specifications: bo
- Repositories: dao+daoimpl
- Services: bo oder bsimpl

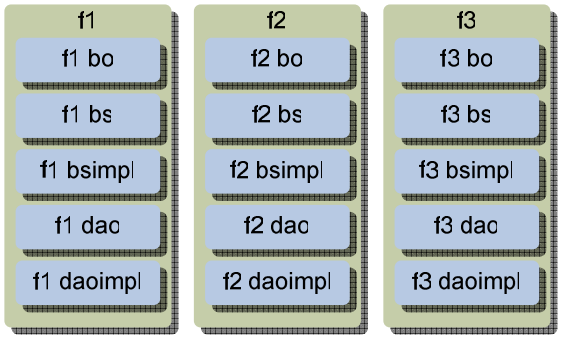
Damit haben wir auch schon alle Elemente für die Entwicklung eines Fachkerns benannt. Der dazu passende Abhängigkeitsgraph der Java Pakete sieht dann so aus:



In einem größeren System wird der Fachkern mit diesen fünf Paketen kaum auskommen. Hier muss man sich gut überlegen, wie man schneiden möchte: erst nach technischen Verantwortlichkeiten und dann nach fachlicher Zusammengehörigkeit oder umgekehrt. Die Gliederung nach technischen Verantwortlichkeiten bezeichnet man auch als *horizontale Schneidung*, die Gliederung nach fachlicher Zusammengehörigkeit als *vertikale Schneidung*. Zwecks Veranschaulichung seien f1, f2 und f3 die Bezeichnungen dreier Fachkomponenten. Die Kombinationen der zwei Schneidungen sehen dann so aus:



H-V-Schneidung



V-H-Schneidung

Wie entscheidet man sich nun? Wenn die Trennung zwischen den Fachkomponenten f1, f2 und f3 nicht durch alle Schichten hinweg sauber ist oder sogar je nach Schicht etwas anders aussieht, dann ist das ein starkes Indiz für eine mangelne Kohäsion von f1, f2 und f3, daher bietet sich die H-V-Schneidung an. Stellt man aber eine starke Kohäsion fest, so dass sich eine V-H-Schneidung leicht durchsetzen lässt, dann muss man sich im Rahmen einer SOA noch gut überlegen, statt den Fachkern zu gliedern verschiedene Teilsysteme einzuführen, die über die SOA Infrastruktur (ESB, MessageBroker, WebServices) verbunden werden. Das hieße dann ebenfalls einen V-H-Schnitt zu setzen, diesen aber nicht erst im Modul core sondern bereits auf Ebene des Gesamtsystems vorzusehen. Diese Entscheidung zu treffen ist im übrigen eine wichtige Aufgabe der Facharchitektur, die in diesem Artikel mangels Fachlichkeit leider etwas kurz kommt.

Zwecks Trennung technischer Infrastruktur von Fachlichkeit muss man sich nur auf den Datenzugriff konzentrieren. Es sollten also ausschließlich die Klassen in daoimpl Gebrauch von den technischen Schnittstellen zum Speichern von Daten machen. Würde man JDBC verwenden, so gäbe es also nur in daoimpl Abhängigkeiten zu java.sql.*.
Wir setzen hier aber JPA in Verbindung mit Annotationen ein, die in den persistenten Geschäftsobjekten angebracht werden, um die Abbildung von Attributen auf Tabellenspalten festzulegen. Das ist zunächst sehr elegant, heißt aber auch, dass wir die eigentlich technologiefreien Geschäftsobjekte doch mittels Annotationen abhängig von javax.persistence.* machen. Dürfen wir das?

Um die Frage zu beantworten, muss man sich vor Augen führen, warum ein gesundes Design, die Kontrolle von Abhängigkeiten und die Trennung Technologie von Fachlichkeit wichtig sind. Es ist letztlich die Wartbarkeit des Systems, die davon stark profitiert. Da in der Wartung im Durchschnitt zwei Drittel der Entwicklungskosten anfallen, die im Laufe des Softwarelebens zusammenkommen, handeln wir damit wirtschaftlich sinnvoll. Wartung vereinfachen heißt etwas konkreter: Verständlichkeit verbessern, Testbarkeit durch Testtreiber unterstützen, Änderungen möglichst lokal halten usw. Wie steht's damit bei Einsatz von JPA mit Annotationen?
Zunächst mal zur Alternative: die Definition des Mappings kann auch durch externe XML-Dateien erfolgen, die natürlich ebenso geschrieben und gewartet werden wollen. Dies ist – zumindest ohne MDSD, also händisch – teurer, als Annotationen zu verwenden. Annotationen „verschmutzen" den fachlichen Javacode nur geringfügig, d.h. die Verständlichkeit leidet nicht sonderlich, die Testbarkeit bleibt erhalten und der Austausch einer relationalen Datenhaltung gegen einen anderen Persistenzmechanismus ist eher unwahrscheinlich, so dass wir nicht befürchten müssen, mit einem Schlage alle Geschäftsobjekte „reinigen" zu müssen. Daher kann man diese Abhängigkeit zu JPA hinnehmen.

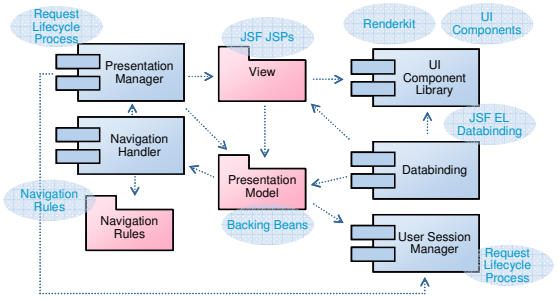
Verwendet man Spring als JPA Container, dann erhält man in daoimpl Klassen, die mit Transaktionen, Connection-Handling und dergleichen technischer Details nichts mehr zu tun haben. Die API, um Objekte zu finden, zu laden oder zu speichern ist derart schlank, dass man zunächst die Sinnhaftigkeit der dao-Pakete bezweifeln kann. Wozu dienen sie eigentlich noch?
Es gibt zwei Gründe, die weiterhin für die isolierte Datenzugriffsschicht sprechen:
Erstens können DAOs gleichsam Repositories (vgl. dazu [Eva 03]) gut die Entscheidung darüber treffen, welche Objektnetze mit einem Methodenaufruf geladen oder geschrieben werden. Auch evtl. nötiges Zusammensetzen von Objektnetzen, das durch mehrere Queries vorbereitet werden muss, kann in konsistenter Weise in dieser Schicht passieren.
Zweitens würde man JPA-Zugriffe ohne dezidierte DAOs tatsächlich auch in die Business Service Implementierungen verstreuen. Ändert sich das API irgendwann, so muss der gesamte Fachkern geprüft werden.
Wenn in den DAOs kein Transaktionshandling mehr auftritt, wo findet es denn dann statt? Diese Frage ist als querschnittliche Lösung bereits beantwortet: deklarativ über AOP auf den Methoden der Java Business Interfaces.

Ein sehr wichtiges Designziel bei nicht-trivialer Domänenlogik ist die einfache Testbarkeit, die erfordert, dass die Klassen im Fachkern möglichst keine Annahmen über ihre technische Umgebung machen dürfen. Dadurch sind sie leicht zu isolieren und vor

allem ohne sperrige Infrastruktur wie Applikationsserver und Datenbank testbar.
Das hier gezeigte Design erreicht dies und ist damit ein sehr guter Ausgangspunkt für Fachkerne.

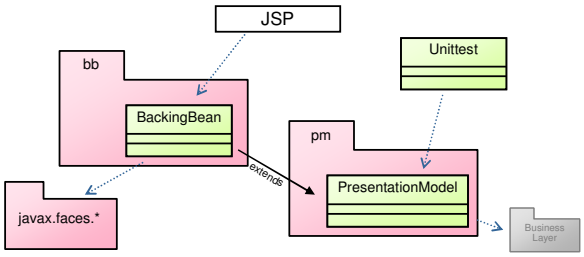
ui: Die Präsentationsschicht auf Basis von JSF

Viele der klassischen Probleme eines Web-Frontends werden durch JavaServer Faces recht passabel gelöst:
Session-spezifischer Zustand wird in *ManagedBeans* verwaltet, die mit Spring DI vergleichbar in der JSF Konfiguration eingetragen werden.
Die Präsentationslogik kann leicht in sogenannte *BackingBeans* ausgelagert werden. Das sind eigentlich ManagedBeans, die ihren Sondernamen daher haben, dass sie direkte Gegenstücke zu JSF-JSPs sind.
UIComponents, die in JSF-JSPs durch Tags deklariert werden und die eigentlichen UI-Widgets darstellen, können durch Ausdrücke in *JSF-EL* an BackingBean Attribute gebunden werden.
Die Transformationen, die zwischen Attributen der BackingBeans und HTML-Feldern notwendig sind, können durch *Converter* definiert werden.
Die Prüfungen, die erforderlich sind, bevor Aktionen ausgeführt werden, lassen sich durch *Validators* deklarieren.
Die Erzeugung von HTML (=rendering) wird durch *RenderKits* von dem Kern der UIComponents getrennt, so dass theoretisch auch anderen Zielkanäle als HTML+JS-fähige Browser möglich sind.
Die Seitennavigation wird mithilfe von *NavigationRules* konfiguriert und dadurch aus dem Javacode herausgezogen.
Die Aktionen, die bei einem Post-Back von Formulardaten vom Browser an den Server ausgeführt werden, werden über *Events* und *Listener* eingesteuert.
Internationalisierung wird durch die Einbindung von *ResourceBundles* ermöglicht.
Und dass JSF das Post-Redirect-Get Muster schlicht ignoriert, ist zwar weniger gelungen, aber durch den konfigurierbaren *RequestLifecycleProcess* und einen besonderen *LifecycleListener* kann man das Problem auch halbwegs elegant lösen.



Elemente einer UI Architektur und die JSF Konzepte

JSF bringt also praktisch alle wesentlichen Elemente einer UI Architektur von sich aus mit. Damit sind die Elemente der technischen Architektur, die eine Maske fachlich zum Leben erwecken, vorerst schnell gefunden: für jede Webseite gibt es eine JSF-JSP, die das Layout und die UI-Components umfasst, sowie eine passende BackingBean. Die BackingBean repräsentiert die Daten und die Präsentationslogik der Seite. Sie kann direkt Java Business Services aufrufen, die – man vergleiche die Diskussion beim Modulentwurf – direkt die Funktionalität bereitstellen oder durch Adapter implementiert sind, die ein fremdsystem aufrufen.
Da wir in dem System Spring einsetzen, sollten die BackingBeans die Business Service-Implementierungen injiziert bekommen, was sich durch einen Spring eigenen DelegatingVariableResolver, der in der JSF-Konfiguration eingetragen wird, rasch ermöglichen lässt.
Soweit, so gut. Bei der Verwendung der BackingBeans stellt man aber schnell fest: so ganz frei von JSF-Infrastruktur lassen sich BackingBeans gar nicht halten. Tatsächlich ist man z.B. für Listen gezwungen, die Daten der Geschäftsobjekte in spezielle JSF-Objekte zu überführen. Durch eine leichte Erweiterung des Entwurfs kann man dort aber Abhilfe schaffen. Die fachlichen Daten und Methoden der BackingBeans werden in sogenannte PresentationModels ausgelagert, von denen die BackingBeans erben. Auf diese Weise kann man die echte Präsentationslogik leicht durch Unittests testen, während die wenigen erforderlichen JSF-spezifischen Transformationen in der Unterklasse sind. Diese wird dann auch in die JSF-Konfiguration eingetragen, so dass die JSF-JSPs auf die um JSF-Besonderheiten erweiterten Backing-Beans gebunden werden.



Package Design der UI mit Trennung von BackingBeans und PresentationModels

Die Präsentationslogik hat also ihren Platz gefunden, doch wo steckt nun eigentlich die Applikationslogik? Die Ausführung einer Action-Methode eines PresentationModel stellt einen Teil der Applikationslogik dar, da hier gesteuert wird, welche Java Business Services aufgerufen werden. Dabei liefert die Action-Methode am Ende immer einen *Outcome*-Wert. Die Steuerung des Seitenflusses wird in JSF durch Regeln konfiguriert, die auf solche *Outcome*-Werte reagieren. Der prozessbezogene Zustand, auch Prozesskontext genannt, kann entweder transient in ManagedBeans gehalten werden oder – wenn langlaufende fachliche Transaktionen auch bei Abstürzen überleben müssen – mithilfe zusätzlicher Business Services, die letztlich die Datenbank verwenden. Man sieht also, dass die Applikationslogik ein wenig über ManagedBeans, die NavigationRules und die Action-Methoden in den PresentationModels verstreut ist.
Soll zusätzlich ein „prozessbewusstes“ fremdsystem verwendet werden, das Prozesskontextverwaltung und –steuerung übernimmt, dann dürfen die JSF NavigationRules nur eine untergeordnete Rolle spielen. In diesem Fall muss man das Zusammenspiel zwischen Seitennavigation und Prozesssteuerung sehr sorgfältig regeln, denn sonst sind fachliche Abläufe extrem schwer nachzuvollziehen und Änderungen werden dementsprechend mühsam.

Am Ende dieser Erläuterungen können wir dem Entwickler nun die Elemente, über die die Geschäftslogik der Präsentationsschicht verteilt wird, nennen:

- Layout und UI-Components einer Seite werden in JSF-JSPs definiert.
- Der Pageflow wird in Form von NavigationRules in der JSF-Konfiguration definiert.
- PresentationModels mit Fachattributen und Action-Methoden sind POJOs im Paket pm.
- BackingBeans erben von PresentationModels und liegen im Paket bb.
- Spezielle Validatoren und Converter können in zusätzlichen dafür bestimmten Paketen abgelegt werden.

ws+adapter: Die Integrationsschicht mit WebServices und Axis2

Für die Integration mit Fremdsystemen über gegenseitig bereitgestellte Dienste muss man immer zwei Fragen beantworten:

- Wie stellt das teilsystem seine Dienste bereit?
- Wie werden die Dienste eines fremdsystems vom teilsystem aus in Anspruch genommen?

Die Dienste Philosophie ist nicht beschränkt auf WebServices oder EJB/RMI sondern lässt sich auch auf Basis eines Messaging Systems „leben“. Integriert man hingegen zwei oder mehr Systeme über FTP oder gemeinsame Nutzung von DB-Tabellen, sollte man statt über Dienste eher in Kategorien von Datenaustausch nachdenken, also: wer stellt wann etwas bereit, und wer holt wann etwas ab?

Hier geht es aber konkret um WebServices über HTTP und – noch konkreter – um die Beantwortung der o.g. Fragen mithilfe von Axis2. Axis2 ist ein SOAP-Framework, das zahlreiche WS-Standards unterstützt. Die Datenübermittlung geschieht mithin auf Basis von XML. Dadurch erhalten wir eine dritte Frage:

- Wie werden Objekte in XML serialisiert bzw. aus XML deserialisiert?

Dies ist indes schnell beantwortet: es gibt heute zahlreiche bequem zu nutzende Java-XML-Databinding Frameworks, die mittels Generierung von Java Klassen aus XML-Schema oder unter Zuhilfenahme von Mapping-Dateien die Java-Objekt-zu-XML-und zurück-Transformation realisieren. Wir setzen hier ein Framework ein, das JAXB, eine Java Standard Schnittstelle in diesem Bereich, implementiert: Apache JaxMe2.
Zur Nutzung dieses Frameworks müssen wir zwei Dinge tun. Erstens aus XML-Schema Java-Klassen generieren, und zweitens an geeigneter Stelle API Aufrufe zur Serialisierung oder Deserialisierung im teilsystem unterbringen. Die Generierung der Java-Klassen erfolgt durch ein geeignetes Maven2 Plugin in der Build-Phase „generate-sources“, und die API Aufrufe werden gekapselt, so dass zur (De-)Serialisierung jeweils nur noch eine Zeile Code nötig ist.

Warum generiert man eigentlich nicht umgekehrt aus Java-Klassen das XML-

Schema (und auch WSDL)? Dieser *Code-First-Ansatz* scheint zwar schneller und einfacher zu funktionieren, trägt uns aber nicht sonderlich weit, denn das würde bedeuten, dass die eigentlichen WS-Schnittstellen der Teilsysteme auf ihrer Implementierung basieren. I.d.R. jedoch stimmen wir zwischen verschiedenen Systemen immer erst die Schnittstellen ab, bevor die Implementierung startet. Richtig ist also: WSDL und XML-Schema sind zuerst da und maßgeblich. Letzteres bezeichnet man als den *Contract-First-Ansatz*, der in allen ernst gemeinten Systemintegrationen der sinnvollere ist.

Es bleiben nach der Klärung der Frage zum Java-XML-Databinding zwei technische Aufgaben: WebServices mit Axis2 bereitstellen und WebServices anderer fremdsysteme aufrufen.

Bei der Bewältigung der Aufgaben gibt es zusätzlich zwei Randbedingungen:

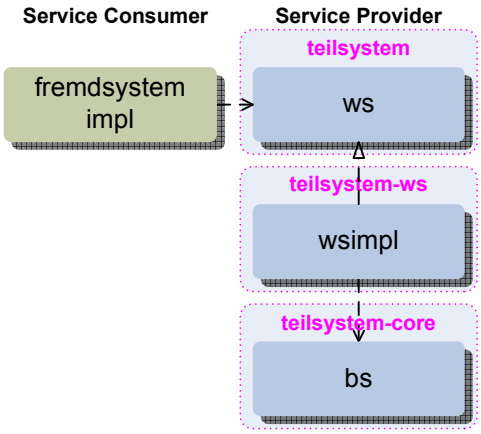
- Die Adapterlogik soll wenig oder gar nichts mit Axis2 oder XML zu tun haben.
- Die Adapterlogik soll über Spring konfiguriert und von Axis2 aus gefunden werden können.

WebService bereitstellen

Zunächst muss man wissen, wie Axis2 Service-Implementierungen grundsätzlich bereitstellt und einen eingehenden WebService Aufruf verarbeitet. Axis2 bietet eine Web-Applikation, die im Tomcat deployt wird. Unter WEB-INF/services erwartet Axis2 sogenannte Service-Archives (AARs). Diese enthalten in META-INF

- services.xml,
- WSDL-Dateien und
- XML-Schema Dateien.

Innerhalb der services.xml wird zu einer Operation eines Service ein vollqualifizierter Klassenname spezifiziert. Bei einem eingehenden Aufruf verarbeitet Axis2 die SOAP-Nachricht über mehrere Phasen, bis dann am Ende eine Instanz der spezifizierten Klasse erzeugt und aufgerufen wird. Die Axis2 Entwickler haben vorgesehen, dass hier direkt die Fachklasse, die letztlich Geschäftslogik enthält, angegeben wird. Es hindert uns jedoch niemand daran, stattdessen einen MessageReceiver anzugeben, der passend zum Servicename eine Spring-Bean aus dem ApplicationContext fischt und entsprechend der angestoßenen Operation via Reflection eine Methode dieser Bean aufruft. Zwischendrin erledigt der MessageReceiver noch das erforderliche Deserialisieren des Request-Payloads, das als Inputparameter der Methode verwendet wird, und das Serialisieren des Returnwerts der Methode, das als Response-Payload des WS-Aufrufs verwendet wird.
Durch diesen Weg wird insbesondere die Generierung von Service-Klassen im Rahmen des Build überflüssig.
Die Spring-Bean, die auf diese Weise als WS-Implementierung verwendet wird, kann ihrerseits das ggf. erforderliche Mapping zwischen Java-Objekten, die dem Nachrichteninhalt entsprechen, und den echten Geschäftsobjekten im bo-Paket des Fachkerns durchführen und Business-Services aus dem bs-Paket aufrufen. Die von der WS-Implementierung verwendeten Business-Services werden mittels Spring in die WS-Implementierung injiziert.
In der folgenden Abbildung sind die Packages und die sie umgebenen Module wiedergegeben. Man beachte, dass das Paket wsimpl die WebService Implementierungen passend zu den Java Interfaces im Paket ws enthält, das sich in einem anderen Maven-Projekt, nämlich dem Schnittstellenprojekt eines Teilsystems, befindet.



Package Design für WS Bereitstellung

WebService aufrufen

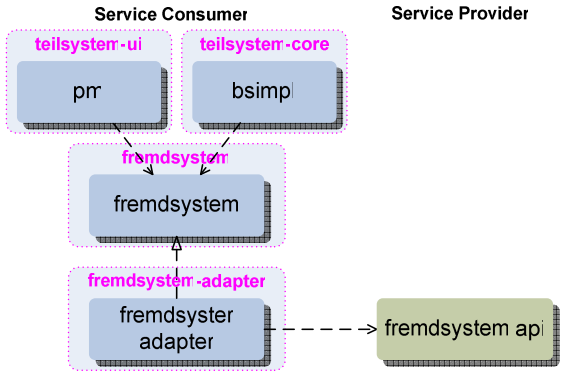
Die API, die Axis2 einem Service Consumer anbietet, ist nicht kompliziert zu verwenden. Dennoch möchten wir den Aufruf möglichst frei von Axis2 und XML spezifischen Code halten, zumal beim Aufruf immer der gleiche Prozess abläuft: Java-Objekte nach XML serialisieren, SOAP-Envelope aus Header und Payload zusammensetzen, Aufruf an Endpoint-Reference durchführen, Response zerlegen und Payload zu Java-Objekten deserialisieren.

Wir erreichen eine starke Vereinfachung für den Aufrufer durch den Einsatz von Dynamic Proxies. Zu jedem Service einer WSDL gibt es im Schnittstellen-Projekt ein passendes Java-Interface. Dieses nutzen wir, um einen Dynamic Proxy zu erzeugen. Das ist ein anonymes Java-Objekt, das aus Sicht des

Aufrufers genau das Java-Interface implementiert, das bei Erzeugung des Proxy spezifiziert wurde. Der dabei mitzugebende Invocation Handler wird später bei jedem Methodenaufruf auf dem Proxy verwendet und führt in unserem Fall genau die oben aufgelisteten Schritte aus, um einen WebService aufzurufen.

Mit der Verwendung von Factory-Methoden in der Spring-Konfiguration kann zu einem entfernten WebService so sehr leicht ein passender Proxy erzeugt werden. Auch dafür ist also keine Code-Generierung erforderlich.

Die folgende Abbildung zeigt das Package Design für einen Adapter zu einem Fremdsystem. Aus Sicht der Aufrufer gibt es nur Java Business Interfaces. Der Adapter aber implementiert diese und greift auf die Schnittstelle des fremdsystems zu.



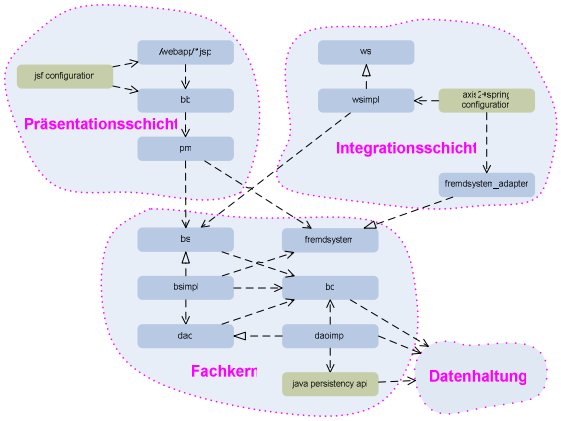
Package Design eines Adapters zu einem fremdsystem

Wo steckt nun die Adapterlogik?
Sie ist natürlich in der Integrationsschicht angesiedelt und dort aber in zwei Teile zerlegt, was sich ja bereits in der Modulbildung niederschlägt.
Zum einen steckt sie in der Implementierung eines WebService, die für Aufrufe durch fremdsysteme bereitgestellt wird, im Modul teilsystem-ws. Da der Aufruf letztlich an eine Spring-Bean durchgeleitet wird, kann dieser Teil der Adapterlogik in einer gewöhnlichen Java-Klasse im Paket wsimpl ohne jede Kenntnis von Axis2, SOAP, HTTP oder XML untergebracht werden. Sie implementiert ein Interface aus dem Paket ws, das in dem Schnittstellenprojekt des Teilsystems untergebracht ist. Die Aufgabe dieser Java-Klasse ist, das nachrichtenartige Objektnetz zu verwenden, um Dienste im Fachkern aufzurufen und ggf. wieder eine Nachricht in Form eines Objektnetzes zurückzuliefern.

Der andere Teil Adapterlogik sitzt ebenfalls in einer gewöhnlichen Java-Klasse, die ein fremdsystem Java Business Interface, wie es sich aus Sicht des aufrufenden teilsystems gestaltet, implementiert. Diese Adapterimplementierung wandelt die übergebenen Geschäftsobjekte in ein nachrichtenartiges Objektnetz und verwendet (ohne es zu merken, weil dieser von Spring injiziert wird) den Dynamic-Proxy, der über den Axis2 ServiceClient die eigentliche SOAP-Kommunikation regelt.

Damit ist das Mapping zwischen Geschäftsobjekten und Nachrichten sowie evtl. nötiges Protokollwissen in POJOs untergebracht, die nicht ahnen, in welchem technologischen Kontext sie verwendet werden.

Schichten einer Anwendung mit Spring definieren
In der folgenden Abbildung sieht man alle Packages, die hier den Schichten zugeordnet wurden.



Packages des teilsystems

Eine dazugehörige Spring-Konfiguration wird sehr schnell unübersichtlich und schlecht wartbar, wenn man versucht, alle erforderlichen Instanzen in einer XML-Konfigurationsdatei unterzubringen.
Zudem möchte man sich die Freiheit erhalten, ein teilsystem zu Testzwecken mit Mock-Objekten zusammenzusetzen, z.B. wenn ein Team die UI und ein anderes den Fachkern nebst Fremdsystem-Adaptern baut. Das UI Team sollte in der Lage sein, die Masken mit Testdaten zu füllen, indem sie Dummy-Implementierungen der Java Business Interfaces in den Paketen bs oder fremdsystem nutzen.
Zum Glück können auch Spring Konfigurationen modular aufgebaut sein. Die einzelnen Konfigurationen orientieren sich dann an den Modulen, für die sie gelten.

Folgende Spring-Dateien sind in unserer Architektur sinnvoll:

- Modul teilsystem-core: spring-core-beans.xml für Business Services, DAOs, Transaktionsgrenzen.
- Modul fremdsystem-adapter: spring-fremdsystem-adapter-beans.xml z.B. für Client-Stubs zum Aufruf fremder WebServices.
- Modul teilsystem-ws: spring-ws-beans.xml für die WebService Implementierungen.
- Modul teilsystem-ui: spring-ui-beans.xml, alternativ ui-config.xml, wenn die Faces-Konfiguration ausreicht.

Die Zusammenführung der einzelnen Dateien geschieht dann beim Aufbau des Spring ApplicationContext, bei dem eine Menge von XML-Konfigurationen angegeben werden. Die Benennung der einzelnen Dateien folgt einem einheitlichen Muster, grundsätzlich besitzt jedes Implementierungsmodul die Datei in src/main/resources, so dass im Zielarchiv (JAR, WAR oder AAR) stets im Classpath bereitsteht.

Zusammenfassung und Ausblick

Die vorgestellte technische Architektur auf Basis von open-source Java Produkten ist die Grundlage einer Lösung für eine ganze Klasse von Java-Server Anwendungen.

Bei der Herleitung sind wir recht systematisch von wichtigen Rahmenbedingungen und dem Architekturstil der Schichtentrennung auf Module und letztlich Java Packages gekommen. Wir haben exemplarisch Lösungsansätze zu querschnittlichen aber auch zu schichtenspezifischen Problemen gesehen. Die verschiedenen Anteile von Geschäftslogik haben ihren Platz gefunden, wobei wir erfolgreich die Verwendung technischer low-level APIs auf wenige Stellen begrenzt haben.

Die Überlegungen, die hinter den Entwurfsentscheidungen stecken, sind langlebig. Die konkreten Entscheidungen können sich bei anderen Rahmenbedingungen und neuen technischen Möglichkeiten aber ändern. Gleichwohl lohnt es sich, das hier gezeigte Modul- und Package Design wie eine Transparentfolie auf andere ähnliche in Entstehung befindliche Systeme aufzulegen und sich zu fragen, welche Abweichungen wodurch begründet werden können.

Die technische Architektur, die hier skizziert wird, ist keine Papierware, sondern so auch prototypisch implementiert worden. In Kürze werde ich auch den passenden Quellcode dazu auf meiner Webseite bereitstellen.

Literatur

[EJB 3.0] http://java.sun.com/products/ejb/docs.html.

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

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

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

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

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