

1

Was ist Architektur?

*„Architecture should be as simple as possible, but not simpler.“
Frei nach Albert Einstein*

1.1 Maschinen und Architekturen

Architekturen strukturieren Anwendungen: Wie auch das Prinzip „Divide And Conquer“ helfen Architekturen bei der Komplexitätsbewältigung und senken so die Kosten für die Pflege und manchmal auch für die Entwicklung von Anwendungen. Und doch ist diese Struktur in der Praxis nicht von Dauer, denn der Feind Nr. 1 jeder Architektur ist der Compiler. Arbeitet man in Java, hat man noch Glück: Im Bytecode bleibt die Struktur der Software nahezu unverändert erhalten. Aber schon bei der nächsten Optimierung, z. B. durch den HotSpot-Compiler, geht diese Struktur unwiederbringlich verloren. Aus den eleganten Ideen, Strukturen und Mustern entsteht lediglich Maschinencode.

Paradoxerweise wirken sich gute Architekturen mit ihren Indirektionen, Kapselungen und Entkoppelungen negativ auf die Wartbarkeit des Maschinencodes aus. Und ein „Urzeitentwickler“ könnte mit dem auf diese Weise entstandenen Maschinencode nur wenig anfangen. Denn für ihn wären die vielen sinnlosen Prozessorzyklen, welche aus den Architektur- oder Designpatterns entstanden sind, kaum fassbar. Dennoch spielt dieses Verfahren in der Praxis kaum noch eine Rolle: Immer weniger „Urzeitentwickler“ sind in der Lage, die Maschinenprogrammierung von modernen Prozessoren direkt durchzuführen. Die Wartbarkeit des Maschinencodes wird daher in der Regel zu einem völlig irrelevanten Argument.

Auch C-Entwickler mussten anfänglich ziemlich lange „kämpfen“, bis die höchst ineffiziente, viel zu abstrakte C-Sprache die direkte Assemblerprogrammierung abgelöst hat. Das gleiche Problem beschäftigte die C++-Entwickler: Objekte? Würde das nicht unnötigen Mehraufwand bedeuten? Ließe sich das Gleiche nicht mit der funktionalen Programmierung erreichen, die zudem den Vorteil einer deutlich höheren Performance und eines wesentlich geringeren Ressourcenverbrauchs hätte? Ähnlich turbulent verlief auch die Einführung von Java und .NET: Bei diesen Sprachen werden sogar die echten CPUs gekapselt und die Anwendungen für eine nicht-existierende Hardware kompiliert. Und diese virtuelle Maschine interpretiert zunächst den künstlichen Maschinencode und führt erst anschließend echten Maschinencode aus. In Anbetracht dieser Tatsache hätte jeder pflichtbewusste „Urzeitentwickler“ wahrscheinlich sofort seinen Job gekündigt ...

Aus Maschinensicht sind Unternehmensarchitekturen noch deutlich ineffizienter: Denn diese beschreiben die Strukturierung von Anwendungen, welche bereits in der virtuellen Hardware ablaufen. Aus Sicht der Maschine (CPU) handelt es sich daher bei höheren Sprachen und insbesondere bei Architekturen um eine unfassbare Verschwendung von Ressourcen. Da aber Hardware immer weniger kostet, nimmt man diese Laufzeitnach-

teile gerne in Kauf. Schließlich wird kreative Software immer noch von Menschen gebaut. Mit der steigenden Komplexität von Anwendungen nimmt die Strukturierung von Software an Bedeutung zu. Nur wenige Entwickler sind in der Lage, Maschinencode zu schreiben. Man findet schon mehr Assemblerentwickler, etwas mehr C-Entwickler, vielmehr Java/.NET-Entwickler. Nun stellt sich die Frage, wie viel Struktur bei der Verwendung von modernen Programmiersprachen noch notwendig ist – nach jedem Compiler bzw. Optimierungsvorgang wird die schöne Struktur sowieso in eine Bitfolge von 0 und 1 umgewandelt ...

1.2 Warum über Architekturen nachdenken?

Im Gegensatz zum Bauwesen lassen sich Anwendungen oft schneller ohne die Berücksichtigung von Architekturen entwickeln. Man muss nur motivierte Entwickler engagieren: Entwickler, die noch Spaß an der Entwicklung von Software haben, werden sich bald und ohne unnötige Formalismen selbst organisieren, effiziente Teams bilden, und sich erstaunlich schnell in das Projekt und die Technologie einarbeiten. Bei komplexen Aufgaben werden mehrere Entwickler an einem PC zusammenarbeiten – das ganze Vorgehen wird sehr agil.

Die Entwicklungskosten lassen sich noch weiter (aber nicht mehr so drastisch) senken, wenn Sie folgende Punkte berücksichtigen:

- Nicht nachdenken – loshacken!
- Die Anzahl der Subsysteme, Komponenten, Packages und Klassen minimieren.
- Auf Objekte verzichten – statische Methoden bevorzugen.
- Systematische Fehlerbehebung kostet Zeit und somit auch Geld – „catch throwable“ funktioniert ebenfalls zuverlässig. Auch muss man nicht soviel schreiben.
- Die Dokumentation auf keinen Fall selbst verfassen (ist nicht mehr zeitgemäß), dafür gibt es auch interessante Reverse-Engineering UML-Tools.
- Copy & Paste bevorzugen – bevor man über andere Arten der Wiederverwendung zu lange nachdenken muss
- Unnötig ist die Trennung von Zuständigkeiten bzw. der Fachlichkeit von der Technik – ohne Trennung funktioniert’s auch gut!
- Sämtliche Muster, Vorgehensweisen und Idiome auf jeden Fall ignorieren – das Lesen von Büchern, Whitepapers und Artikeln dauert einfach zu lange. Diese Zeit geht dem Projekt unwiederbringlich verloren.

Die hier genannten „Verbesserungsvorschläge“ wurden tatsächlich in nicht so offensichtlicher Form in Projekten befolgt – zumindest hat der Quellcode danach ausgesehen. Nichtsdestotrotz kann diese Art Software zu entwickeln mit viel Glück auch sehr günstig sein. Aber im Gegenzug fallen die Folgekosten für die Produktionseinführung, den laufenden Betrieb und die Wartung deutlich dramatischer aus. Dennoch wird die Unternehmung aus der Sicht des Projektleiters als erfolgreich abgeschlossen beurteilt werden. Auch wenn das Projekt bereits einige Monate später auf Grund von hohen Wartungskosten beendet wird... Dies ist im Übrigen ein „Trend“ der letzten Jahre: Die meisten Projekte

gehen ohne Probleme in die Produktion. Interessant ist nur, ob sie auch die ersten Erweiterungen und Korrekturen überstehen.

1.2.1 Baukunst vs. Softwareschmiede

Die Entwicklung von Software wird oft mit der Herstellung von Produkten aus anderen Industriesparten verglichen. Jedoch zeichnet sich Software durch andere Eigenschaften aus, die sich nicht so ohne weiteres mit anderen (Kauf-)Produkten vergleichen lassen: Denn hierbei handelt es sich um ein virtuelles, veränderbares Produkt. Im Gegensatz dazu sind Gebäude erheblich konkreter und bei weitem nicht so erweiterbar wie Software. Auch sind die Wartungskosten eines Hauses deutlich günstiger im Vergleich zu der Anfangsinvestition. In der Softwareentwicklung übersteigen hingegen die Wartungs- und Betriebskosten wesentlich die Aufwände für die Entwicklung. Dennoch findet man immer wieder diesen Art Vergleich: Software-Architekturen werden mit eher konkreten Gebäudearchitekturen oder CAD-Entwürfen gleichgesetzt und Manager wundern sich folglich, warum Software nach der Installation so teuer ist bzw. sich die Folgekosten so schlecht planen lassen.

Auch gibt es einen großen Unterschied zwischen einem typischen Bauherrn und einer Fachabteilung – Bauherren wissen oft ganz genau, was sie wollen – und Sie wissen auch ganz genau, dass nachträgliche Änderungen viel Geld kosten. Dies fließt vom Anfang an in die Kalkulation mit ein: Man entscheidet sich zwischen einem Fertig- und einem „Individual“-Haus. Und weiß, dass die Umsetzung von Änderungen, Erweiterungen und Sonderwünschen bei Fertighäusern letztendlich deutlich teurer zu Buche schlagen als bei einem Individualhaus. Dieses Bewusstsein fehlt oft (noch) bei dem Software-Auftraggebern: Spezifikationen und Pflichtenhefte verlieren oft schon während der Entstehungszeit ihre Gültigkeit. Und um dieses Problem zu umgehen, werden die Anforderungen so schwammig formuliert, dass die Entwickler/Architekten sämtliche Freiheiten genießen, aber dadurch auch keine vertragliche Sicherheit haben. Dieser Effekt lässt sich noch verstärken durch möglichst umfangreiche, oder am besten noch generierte Spezifikationen.

Softwarearchitekturen lassen sich eben nicht mit Architekturen von anderen Bereichen vergleichen. Auch die Motivation für eine Softwarearchitektur ist eine andere. Während man einen Bauleiter bereits für den Bau von Gebäuden benötigt, stört ein Softwarearchitekt eher die Entwicklung. Gute, d. h. motivierte Entwickler kommen auch ohne abstrakte Architekturen und die oft technologieneutralen Präsentationen oder Entwürfe eines Architekten aus. Softwarearchitekturen und somit auch -architekten werden primär für die Minimierung der Gesamtkosten und nicht für die Beschleunigung der Entwicklung benötigt. Somit leben die Architekten gefährlich, zumindest in einem ständigen Konflikt mit dem Projektleiter. Projektleiter sind vorwiegend an der Einhaltung der Termine und somit der Verkürzung der Entwicklungsphase, die Architekten dagegen auch an der Wartbarkeit der Anwendung interessiert.

Der Hauptunterschied zwischen den Architekten der Bau- und Softwarebranche ist jedoch offensichtlich: Während ein Architekt aus der Baubranche tatsächlich ein virtuelles, und manchmal ein eher konkretes Modell des Gebäudes erstellt und dem Kunden präsentiert, ist der Softwarearchitekt nur am Rande an den Äußerlichkeiten der Anwendung interessiert – für ihn zählen vielmehr die inneren Werte. Somit ist die Arbeit eines Softwarearchitekten für den Auftraggeber bzw. Kunden schwer greifbar und somit auch oft verdächtig.

1.2.2 Wartbare Software

„Wartbare Software“ – was ist mit diesem Begriff eigentlich gemeint? Diese Frage lässt sich schnell beantworten: Software ist dann wartbar, wenn der Aufwand für die Pflege und Weiterentwicklung möglichst gering ist. Daran schließt sich eine weitere interessante Frage an: Lässt sich Wartbarkeit messen? Die Antwort: Eine direkte Messung ist kaum möglich, vielmehr kann man aus anderen messbaren Einheiten auf die Qualität der Software schließen. Denn aus dem Grad der Koppelung, Schichtung und Einhaltung der Namenskonventionen lässt sich indirekt die Wartbarkeit ableiten. Soweit die Theorie. Die Praxis gestaltet sich dann oft schwieriger, als man denkt. Ein Beispiel: Beim Review einer Web-Anwendung im Rahmen einer automatisierten Qualitätskontrolle hat es sich herausgestellt, dass die Koppelung zwischen den Modulen sehr gering ist – worauf man kurzfristig sogar stolz war. Nach einem „manuellen“ Review der Module kam eine andere Tatsache ans Licht – bedingt dadurch, dass die Software von C-Programmierern entwickelt wurde: Es waren nur wenige, aber dafür mächtige, monolithische Klassen vorhanden, die nur wenig miteinander kommunizierten. Folglich waren die Kohäsion sehr hoch, die Koppelung sehr lose und die Software letztlich unwartbar.

In der Praxis lässt sich die Wartbarkeit auch mit anderen, einfacheren Indikatoren bestimmen: nach dem Aufwand, der für die Einarbeitung von neuen Entwicklern erforderlich ist. Je klarer, ordentlicher, durchgängiger und offensichtlicher die Struktur der Software, desto weniger Zeit benötigt ein Entwickler, bis er einen eigenen produktiven Einsatz leisten kann. Anwendungen werden von der „Entwicklung“ oft an den Betreiber übergeben. Somit wird Software von unterschiedlichen Personenkreisen entwickelt und gepflegt. Insbesondere in diesem Fall zahlen sich Mehraufwände für die Strukturierung und mögliche Refactorings ziemlich schnell aus.

1.3 Die Theorie

Nun wissen wir, was die Architekturdiziplin bringt, wir wissen aber immer noch nicht, was Architektur eigentlich ist. Die Vergleiche mit CAD-Zeichnungen und Bauplänen bringen hier leider nur wirkliche Architekturschüler weiter. Für die Praxis der Softwareentwicklung hingegen sind solche Vergleiche völlig nutzlos.

1.3.1 Definitionen

Dieser Abschnitt stellt relevante Fachbegriffe vor. Auf den nächsten Seiten werden Sie sich mit den zentralen Begriffen zu diesem Thema beschäftigen.

Architektur

Die meisten Definitionen in der Fachliteratur bezeichnen mit „Architektur“ zunächst die innere Struktur eines Systems. Dabei geht es insbesondere um das Zusammenspiel von internen Modulen. Sowohl die Struktur als auch das Zusammenspiel solcher Module und die Schnittstellen sind im Zusammenhang mit dem Architekturbegriff von Bedeutung – und so könnte man die Erläuterung folgendermaßen formulieren: „Unter Architektur versteht man die Regeln für das Zusammenspiel von zusammengehörigen (kohäsiven) Einheiten in einem gegebenen Kontext.“

Kontext

Fachliche Zusammenhänge lassen sich mit der Hilfe von Komponenten oder Objekten abbilden. In Software wird aber die Realität nicht vollständig abgebildet, sondern jeweils nur die interessanten, funktionalen Aspekte – also ein Ausschnitt aus der realen Welt. Der fachliche Kontext hilft bei der Konzentration auf das Wesentliche und so können die für das Projekt interessanten Eigenschaften eines Sachverhalts hervorgehoben und die zweitrangigen Aspekte weggelassen werden. Beim Kontext handelt es sich also um eine Ansammlung von Filterkriterien oder eine eingeschränkte Sicht auf die Fachlichkeit des Systems. Bei der Modellierung eines Online-Shops sind beispielsweise Verfügbarkeit, Preise, Rabatte, Zahlungsweisen und Beschreibungen wichtiger als die Beschaffenheit, die Materialien, die Gewichte oder die technischen Merkmale der angebotenen Produkte. Ein Buch kann hier z. B. auf ein abstraktes Produkt mit den Attributen: Preis, Autor, Anzahl der Seiten, Beschreibungen und eine Liste von Bewertungen reduziert werden. Das gleiche, fachliche Objekt könnte jedoch bei der Entwicklung von Verlagssoftware über völlig andere Eigenschaften verfügen. Dann wären unter Umständen die Art des Covers, die Anzahl der Farben, die Art des Papiers, das Layout oder die Vergütung der Autoren wichtiger. Diese Ausführungen verdeutlichen, auf welche Weise der fachliche Kontext zur Bildung von kohäsiven Einheiten eingesetzt wird.

Kohäsion (Cohesion)

Die Kohäsion bestimmt die Ähnlichkeit bzw. Zusammengehörigkeit von Elementen. Die Kohäsion ist hoch, wenn die Elemente jeweils ähnliche Aufgaben erfüllen bzw. für Ähnliches zuständig sind. In der Architektur spielt eher die „fachliche“ Kohäsion eine übergeordnete Rolle. Modellierungselemente werden daher nach fachlichen und nicht nach technischen Aspekten gruppiert.

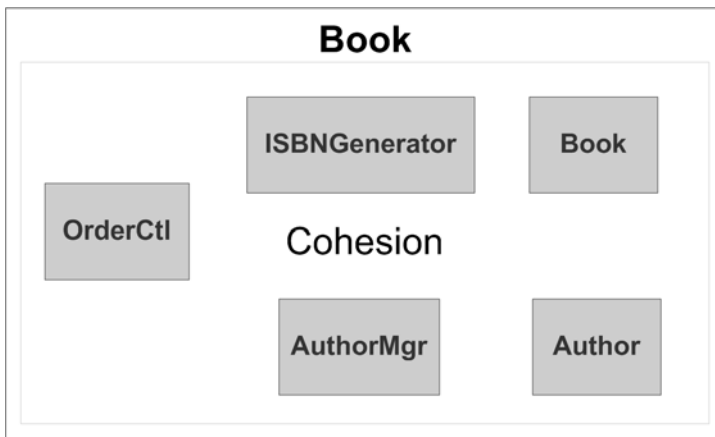


Abbildung 1.1: Gruppierung von zusammengehörigen Elementen

Die Gruppierung von Anwendungen nach ihrer Verteilung ist suboptimal, da sich das Deployment potentiell jederzeit ändern kann. So wurden in einem Projekt beispielsweise die Klassen einer serverseitigen Anwendung in „Client“- und „Server“-Packages partitioniert. Die fachlichen Zuständigkeiten wurden dabei aber nicht berücksichtigt. Das

Problem: Validierungen und Plausibilitätsüberprüfungen hatten sich aus Performancegründen anschließend vom Server in den Client verschoben. Konsequenterweise wäre es nun notwendig geworden, die Klassen ebenfalls vom Package „Server“ in den „Client“ zu befördern. Doch wie so oft stand hier der Wille für das Werk: Der Projektleiter hatte zwar Refactoring-Ressourcen und das entsprechende Zeitfenster nach der erfolgreichen Produktionseinführung zugesagt – doch wurde dies nie realisiert. Denn die „Never Touch The Running System“-Devise ist oft mächtiger als der anfängliche gute Vorsatz des Projektleiters.

Bei der Bildung von kohäsiven Einheiten muss auf jeden Fall der fachliche Kontext mit berücksichtigt werden. So wäre bei der Entwicklung von Online-Shops eine gemeinsame Ablage von Büchern, DVDs oder Elektronikartikeln in einem Namensraum (z. B. Package) durchaus denkbar, da hier ein „verkaufbares“ Produkt im Vordergrund steht. In einem Verlag dagegen würde man sogar Bücher von Fachartikeln trennen, da sich die Fachlichkeit (der Workflow der Publikation, z. B. die Verrechnung, die Erscheinungsdaten oder auch die Verwaltung der Abonnements) der beiden Objekte deutlich voneinander unterscheidet.

Koppelung

Die Kohäsion vereinfacht das Auffinden von bereits existierenden Elementen, da diese nach fachlichen Aspekten gruppiert wurden. Die konsequente Gruppierung der Objekte unter der Berücksichtigung ihrer Kohäsion ergibt einen Baum, dessen Knoten durch Anwendungen, Subsysteme, Schichten, Komponenten und Packages repräsentiert werden. Die Blätter werden dargestellt durch Klassen bzw. Schnittstellen. Die Kohäsion selbst sagt jedoch noch nichts über die Realisierung von Abhängigkeiten zwischen zusammengehörigen Elementen aus.

Die Koppelung ist der Grad der Abhängigkeit zwischen unabhängigen Elementen. Die Abhängigkeit wird durch Assoziationen, UML-Abhängigkeiten und Kompositionen (also Referenzen), aber auch durch logische Abhängigkeiten (Datenabhängigkeiten) realisiert.

Bei den Elementen, die gekoppelt werden, kann es sich um Attribute, Methoden, Klassen, Packages, Subsysteme und Anwendungen handeln. Jedoch: Bei der Architekturbildung spielt lediglich die gröbere Koppelung (ab der Klassenebene) eine Rolle. Lose gekoppelte Elemente sind besser wartbar als monolithische Systeme. Aber dieser Vorteil ist gleichzeitig ein wesentlicher Nachteil: Die lose Koppelung wird oft mit dem „late binding“ realisiert, und dieses schaltet die Typüberprüfungen des Compilers aus. In Java kann diese Anforderung mit Reflection, also z. B. `Class.forName("fully_qualified_name").newInstance()`, erfüllt werden. Bei einer Änderung der entkoppelten Elemente muss die Anwendung dann nicht mehr neu kompiliert werden. Der Nachteil dieser Lösung ist allerdings, dass Sie öfter mit Laufzeitfehlern rechnen müssen, welche sich negativ auf die Robustheit der Anwendung auswirken.

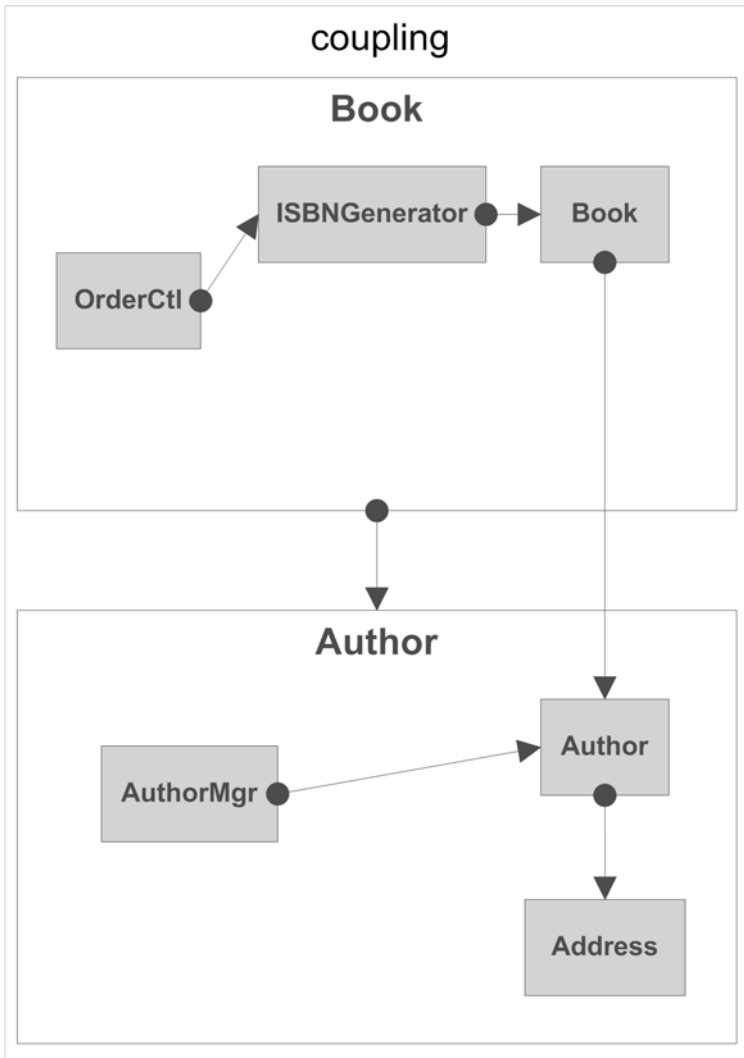


Abbildung 1.2: Die Koppelung zwischen kohäsiven Einheiten

Lose Koppelung kann auch mit der Hilfe von „Message Oriented Middleware“ (MOM)-Produkten erreicht werden. Die so entkoppelten Elemente kommunizieren zunächst lediglich asynchron miteinander. Doch diese Möglichkeit hat einen entscheidenden Nachteil: Neben der Typsicherheit verliert man hier auch noch den Komfort des synchronen Aufrufes. Der Einsatz von MOM-Lösungen ist daher nur bei asynchronen, fachlichen Vorgängen sinnvoll. Insbesondere quasi-asynchrone Verbindungen, d. h. bidirektionale, synchrone und queue-basierte Verbindungen sind deutlich komplexer als ein Remote Procedure Call (RPC)-Aufruf. Insbesondere die Fehlerbehandlung, d. h. die Behandlung von „Poisoned Messages“, Reaktion auf Timeouts, robuste Konvertierung der abstrakten in konkrete Nachrichten ist nicht trivial.

Obwohl monolithische Systeme schlecht wartbar sind, vergrößert die „vorsätzliche“ Entkoppelung von kohäsiven Elementen zunächst ebenfalls die Komplexität des Systems und die notwendigen Aufwände für die Sicherung der Robustheit. Die Folge: Eine unbegründete Entkoppelung von Elementen wirkt sich ebenso negativ auf die Wartbarkeit des Systems aus. So sollte man auf jeden Fall – wo immer möglich und sinnvoll – die Typsicherheit des Compilers bevorzugen. Denn in der Regel sind Compilerfehler immer den Laufzeitfehlern vorzuziehen.

Zwar lassen sich bei den meisten serverseitigen Anwendungen die Layer nahezu beliebig voneinander entkoppeln. Und mit einigen Patterns wie Command, Adapter und Idiomen wie z. B. Value-Object kann man den Compiler überlisten, d. h. der Compilervorgang bleibt auch bei der Erweiterung/Änderung bzw. Entfernung der Funktionalität oder Implementierung kompatibel. Aber das geht nur, wenn die Parameter, Rückgabewerte und auch Exceptions allgemein genug sind. Die Zuständigkeit für die Sicherstellung der Kompatibilität der Typen und Signaturen der Services überträgt sich dann in diesem Fall vom Compiler direkt auf den Entwickler. Zwar kapselt die Einführung von speziellen Transfer-Objekten für die Entkoppelung der Persistenz die lokalen Änderungen der Persistenz. Aber bei den meisten Change-Requests sind alle Schichten betroffen, sodass bei einer funktionalen Erweiterung die Persistenz das Transfer-Objekt, die Mapping-Logik, die Schnittstellen der Layer und der Client einzeln angepasst werden müssen. Die Konsequenz für den Entwickler: Eine „gute“ Kapselung der Persistenz lohnt sich nur dann, wenn die Änderungen der Persistenz sich nicht auf die anderen Schichten auswirken.

Package

Ein Package ist eine Organisationseinheit und gleichzeitig auch ein Namensraum. Das Package verhält sich ähnlich wie ein Verzeichnis eines Filesystems. Es kann für die Gruppierung von Klassen, aber auch von Dokumentationen, Diagrammen und anderen Artefakten verwendet werden. In der Architektur spielt ein Package eine wichtige Rolle, da dadurch Elemente nach beliebigen Aspekten gruppiert werden können. Packages eignen sich hervorragend für die Sichten- und auch Schichtenbildung komplexer Anwendungen. Hier können Referenzen auf bereits vorhandene Elemente abgelegt werden. Somit lassen sich mit Hilfe von Packages auch technische und nicht nur fachliche Aspekte repräsentieren und dokumentieren.

Fachliche Komponente

Eine fachliche Komponente ist ein Package mit zusätzlicher semantischer Bedeutung. Im Gegensatz zu einem Package gruppiert eine fachliche Komponente kohäsive Elemente nach fachlichen Kriterien. Dabei kapselt sie ihr Innenleben stärker als ein Package. Die Fachlichkeit einer Komponente wird somit lediglich durch definierte und explizit veröffentlichte Schnittstellen verwendbar. Auch die Abhängigkeiten der fachlichen Komponente zu anderen Komponenten und Services werden durch explizite Schnittstellen manifestiert. Eine Komponente hat daher also „Nutzer“ (Required)- und „Bieter“ (Provided)-Schnittstellen.

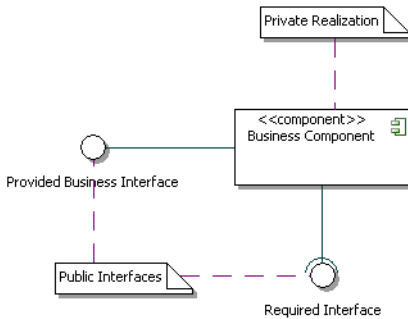


Abbildung 1.3: Die Schnittstellen einer fachlichen Komponente

Sowohl die Kohäsion des Innenlebens einer Komponente als auch die Qualität der Schnittstellen haben direkte Auswirkungen auf die Wartbarkeit der Anwendung und die Qualität der Architektur. Die Außenschnittstellen einer Komponente sollten dabei für den Benutzer (einen besonderen Stakeholder – den Akteur) einen echten Mehrwert darstellen. Und sie sollten immer aus der Sicht des Akteurs entworfen werden. Besonders wichtig dabei sind die Namensgebung der Schnittstelle, ihre Methoden, Parameter und Rückgabewerte. Auch darf nicht vernachlässigt werden, dass das Auftreten von möglichen Fehlern erfasst werden muss. Diese sollten aus der Sicht des Aufrufers modelliert werden und insbesondere für den Stakeholder verständlich sein.

Da Exceptions die Signatur der Schnittstellen-Methoden beeinflussen, sollte auch hier die Prämisse „Trennung der Fachlichkeit von der Technik“ gelten. Fachliche Exceptions werden aus den funktionalen Anforderungen abgeleitet, technische Exceptions ergeben sich bereits auf Grund der Technologiewahl und des Deployments der Komponente. Laufzeitfehler und schwerwiegende Systemfehler existieren nur, weil der Compiler das Auftreten von Fehlern wie z. B. einer NullPointerException nicht feststellen und somit durch Compilerfehler unterbinden kann.

Ein fachlich stark kohäsives Package mit expliziten, am besten vertraglich definierten, aber eher losen Abhängigkeiten zu anderen Einheiten kann als fachliche Komponente definiert werden. Die Schnittstellen dieser fachlichen Komponenten sind eher grobgranular und auf die Anwendungsfälle abbildbar. Da die Schnittstellen von fachlichen Komponenten Anwendungsfälle repräsentieren, müssen die Aktionen (Methoden) der Außenschnittstellen die Komponente jeweils in einem konsistenten Zustand „verlassen“. Diese Anforderung wird oft durch unabhängige, technische Transaktionen gelöst. Die Fachlichkeit der Komponenten wird dabei von den technischen Aspekten getrennt. Somit wird sie in einer definierten Laufzeitumgebung – einem Container – ausgeführt. In der Praxis werden Komponenten in einen so genannten Applikationsserver, z. B. einem EJB-Container oder .NET-Server, deployed. Dieser übernimmt die Querschnittsaufgaben wie das Logging, das Caching oder die Verteilung – ein Container kann dabei mit einem „Komponentenbetriebssystem“ verglichen werden.

Dieser Vorgang birgt eine neue Anforderung: Da ein Container in einem anderen Adressraum abläuft als die Anwendung, müssen die Schnittstellen von fachlichen Komponenten remotefähig sein. Auch die Parameter und die Rückgabewerte müssen demzufolge zwischen unterschiedlichen Adressräumen – oft auch Maschinen – übertragen werden.

Vision

Aufgrund von feinen Spezifikationen ist es oft sehr schwer zu bestimmen, was tatsächlich zu tun ist. Stellen Sie sich die folgende Spezifikation für den Bau eines Motors vor:

- Drehmoment: 500 NM bei 2000 Umdrehungen/Minute,
- Leistung: 400 PS,
- Verbrauch: 8-10 l/100 km bei 90 km/h,
- 8 Zylinder,
- Direkteinspritzer,

sowie weitere technische Daten wie Gewicht, Durchmesser der Zylinderbohrung, die zu verwendenden Materialien, Aussehen und Farbe. Diese Spezifikation wäre für ein Team von Spezialisten bestimmt. Es wäre interessant zu sehen, ob der Kunde mit dem Ergebnis des Projekts zufrieden wäre.

Zunächst ist aus den Vorgaben nicht unbedingt ersichtlich, ob ein Lastwagen- oder ein Sportwagenmotor zu bauen ist. Es wäre doch viel effizienter, zunächst die Eckdaten – die Vision – an die Ingenieure zu übergeben. Beispiel: „Bauen Sie mir einen Sportwagenmotor. Für mich ist eine möglichst starke Beschleunigung bei geringem Verbrauch sehr wichtig“. Mit dieser Information kann man in den ersten Iterationen viel mehr anfangen als mit zu detaillierten Spezifikationen. Nach der Erstellung des Prototypen oder einer Simulation können die Daten weiter verfeinert werden.

Es scheint, als ob in der Praxis die Fachabteilungen und Auftraggeber das Schreiben von Spezifikationen lieben würden. Oft bekommt man hunderte, manchmal tausende von Seiten übergeben. Leider wird aber oft nur das dokumentiert, was für die Entwickler ohnehin offensichtlich ist. Die kritischen und interessanten Eigenschaften eines Systems werden oft gar nicht erst erfasst, dafür wird aber akribisch das z. B. kaum veränderbare Verhalten von Oberflächen dokumentiert.

Die Aufgabe der Vision ist, dieser Tendenz entgegen zu wirken. Die Vision definiert mit so knappem Umfang wie nur möglich die Essenz und den Grund für die Existenz des Systems. Nach dem Lesen der wenigen Seiten des Visionsdokuments (sollten nicht mehr als 5 Seiten sein), sollte es jedem Architekten klar sein, WARUM das System gebaut wird, was die „Killerfeatures“ sind und warum überhaupt der Auftrag für die Entwicklung erteilt wurde. Für die Bildung der Architektur ist manchmal ein gutes Vision-Dokument wertvoller, als umfangreiche Dokumentationen des gewünschten Verhaltens. Man könnte die Amazon.com-Anwendung mit tausenden Seiten beschreiben. Die Beschreibung des Maskenflusses, die Anordnung der Schaltflächen, der feingranularen Use Cases wie Login, Logout, Eingabe der Adresse usw. kann beliebig aufwändig sein. Nach der Lektüre eines solchen Dokuments würde man aber immer noch nicht wissen, dass amazon.com primär für den Verkauf von Büchern erstellt wurde. Diese Information sollte jedoch in der Vision nicht fehlen.

Falls die Vision fehlt, sollte dieses Dokument auch nachträglich erstellt werden. Für die Klärung der offenen Fragen reicht oft ein kurzes, informelles Treffen während der Mittagspause mit dem Auftraggeber ... oder natürlich auch tausende von formalen und offiziellen Meetings mit Protokollen, Präsentationen, LOPs, TODOs, Refactoring der Microsoft Project Charts und allem was einfach dazugehört.

Standard

In der IT werden Technologien, Vorgehensweisen und Patterns als „Standard“ definiert, die man sich einfach nicht wegdenken kann. So sind Ant, JUnit, XDoclet zwar keine offiziellen Standards, dennoch sehr verbreitet und deswegen durchaus als Standard zu verstehen. Nicht nur Infrastruktur-Frameworks, sondern auch Architekturen, Patterns und Vorgehensweisen können sich stark verbreiten. GoF-Patterns, J2EE-Patterns von Sun, InversionOfControl (IoC), Tomcat oder Spring sind einfach Tatsachen, die man unbedingt als Architekt berücksichtigen sollte.

Standards sollten immer einer Eigenentwicklung vorgezogen werden. Denn die Wahrscheinlichkeit, dass ein Entwickler GoF-Patterns kennt, ist deutlich höher als bei einem in-house entwickeltem Muster. Außerdem sind Standards besser dokumentiert. Zudem ist die Wahrscheinlichkeit größer, dass es Angebote für die entsprechenden Schulungen gibt. Populäre Technologien arbeiten zudem aufgrund ihrer hohen Verbreitung fehlerfreier als individuelle, exotische Lösungen. Ein weiteres Argument für den Einsatz von Standards: Diese wirken sich immer positiv auf die Wartbarkeit einer Anwendung aus – auch wenn die Frage Standard vs. Eigenentwicklung in der Realisierungsphase eines Projekts noch keine wichtige Rolle spielt.

Neben den bereits erwähnten Quasi-Standards existieren auch noch „echte“ Standards. Für die Bildung einer Softwarearchitektur wird diese rein theoretische Unterscheidung jedoch keine Rolle spielen: Ausschlaggebend für die Akzeptanz durch die Entwickler und für eine kostengünstige Wartung der Entwicklung ist allein das Kriterium, dass der verwendete Standard möglichst weit verbreitet, populär und anerkannt sein muss.

Pattern

Ein Pattern ist eine fertige Lösung für ein wiederkehrendes Problem – und dieses hat in der Regel eine allgemeine Bedeutung, damit sich die Formulierung des Patterns überhaupt lohnt. Im Rahmen der Software-Architektur existieren verschiedene Arten von Patterns, wobei die Design-Patterns aus dem GoF-Buch zu den populärsten Vertretern zählen (und auch die Basis für eine Reihe von Derivaten bilden). Patterns sind keineswegs „Rocket Science“ – im Gegenteil. Nach einiger Projekterfahrung würde man meist auch selbst auf die Lösung eines Problems kommen. Allerdings findet jeder Entwickler naturgemäß eine etwas andere Variante. Diese unterscheidet sich durch eine andere Benennung, durch die Struktur und nicht selten auch durch Details in der Funktionsweise. Zwar machen solche individuellen Lösungen die Architekturen oft interessanter und „peppiger“. Ihr entscheidender Nachteil aber ist, dass sie für andere Entwickler nicht immer nachvollziehbar und somit auch schwerer wartbar sind.

Patterns sollten daher beim Entwurf einer Architektur nicht fehlen – denn durch ihre allgemeine Bekanntheit lässt sich der Umfang der Dokumentation minimieren. Allerdings schränkt diese Arbeitsweise den schöpferischen und kreativen Geist eines Entwicklers etwas ein, sodass dieses Vorgehen immer wieder auf Widerstände in Projekten stößt. Eine weitere wichtige Aufgabe der Architektur besteht auch darin, nicht möglichst viele Patterns vorzugeben, sondern die Vielfalt auf ein praktikables, vernünftiges Maß zu begrenzen. Denn die meisten Projekte kommen mit erstaunlich wenig Mustern aus. Je weniger, desto besser, ist hier die Devise – ohne jedoch die Patterns in Eigenforschung neu erfinden zu müssen.

Abstraktion

Der Begriff der Abstraktion lässt sich einfach definieren: „die Fähigkeit des Architekten, zum richtigen Zeitpunkt unnötige Details kontextabhängig zu vergessen oder wichtige Aspekte hervorzuheben“. Der kritische Punkt bei dieser Angelegenheit: die Identifizierung des richtigen Zeitpunktes, der unnötigen Details oder der wichtigen Eigenschaften und des Kontextes. Aber ohne diese Anwendung der Abstraktion wird die Architektur einer Anwendung nicht „sauber“, da man sich immer wieder mit unnötigen Details beschäftigen muss.

Die Abstraktion lässt sich in mehrere Ebenen unterteilen: Auf der Ebene der Fachlichkeit findet man lediglich Geschäftsaktivitäten, fachliche Zustände und Fachklassen. In der nächsten Stufe würde man bereits entweder Außenschnittstellen der Komponenten oder Objekte sehen. Bei einer weiteren Detaillierung kann man die Persistenz, die Controller und die Aktivitäten erkennen. Wenn man noch tiefer eintaucht, werden weitere Details wie z. B. Connection-Pools, Value-Objekte oder OR-Mapper sichtbar. Grundsätzlich gilt: Je besser die Architektur und das Design einer Entwicklung konzipiert wurden, desto offensichtlicher treten die einzelnen Details hervor.

Ein Architekt sollte auf jeden Fall in der Lage sein, den Abstraktionsgrad abhängig von der aktuellen Projektphase beliebig zu wählen. So hat vielleicht eine Fachklasse in UML plötzlich eine ganz andere Bedeutung, weil man sich die offensichtliche Technik jederzeit „dazudenken“ und auf diese Weise die Architektur verifizieren kann. Verfügt ein Architekt über dieses „geistige“ Auge, können Transaktionseinstellungen, Facaden, Controller, Verteilung, Persistenzklassen, Value-Objekte, Infrastrukturframeworks und die Interaktion mit anderen fachlichen Komponenten visualisiert werden – und man kann sich viele unnötige Iterationen sparen. Die Model Driven Architecture (MDA) beginnt also im Kopf des Architekten – und im Gegensatz zu gängigen Transformatoren sollte der Trafo im Kopf des Architekten das Reverse-Engineering unterstützen.

Die Abstraktion ist auch der erste Schritt für die Entstehung von Schichten – die eigentlich nichts anderes sind als Abstraktionsstufen mit fest definierten Zuständigkeiten.

Kapselung

Abstraktion und Kapselung teilen sich einige Ideen. Mit Hilfe der Kapselung werden unnötige, oft technische Details vor der Außenwelt versteckt – und nachdem diese unsichtbar bleiben, können sie von Außenstehenden auch nicht verwendet werden. Aus diesem Grund lassen sich gekapselte Elemente jederzeit verändern, erweitern oder sogar entfernen, ohne die Nutzer (Clients) der veröffentlichten Dienste ändern zu müssen. Wie die meisten Architekturprinzipien kann man insbesondere die Kapselung rekursiv anwenden: So gibt es in einer Klasse private und public Methoden – und private Methoden sind für andere Klassen nicht sichtbar. Außerdem gibt es die so genannten Friend-Sichtbarkeiten, welche nur innerhalb eines Namensraums bzw. Packages gelten. Hier findet man gekapselte Klassen, welche außerhalb dieses Packages nicht sichtbar sind. Auf einer noch höheren Ebene befinden sich Komponenten. Und diese haben ein sehr ausgeprägtes Kapselungsbewusstsein, denn nur über die vorgesehenen Schnittstellen lassen sich die fachlichen Dienste einer Komponente überhaupt nutzen. Darüber hinaus wird die Kapselung auch von Subsystemen eingesetzt. Diese gruppieren kohäsive Komponenten und können auch explizite Schnittstellen anbieten. Dabei sollten Subsysteme

immer voneinander unabhängig sein, denn Anwendungen und Anwendungsverbände verfügen oft über keine oder nur dedizierte Enterprise Application Integration (EAI)-Schnittstellen für den Zugriff auf die benötigte Fachlichkeit (EAI-Schnittstellen arbeiten meistens asynchron und eher grobgranular).

Zuständigkeit

Jedem Element einer Anwendung sollten genau festgelegte Zuständigkeiten zugewiesen werden. Dabei sollte das Element für möglichst wenige Aufgaben vorgesehen werden, diese aber im Gegenzug besonders gut und zuverlässig erfüllen. Auch sind mächtige Elemente wie z. B. Klassen mit sehr vielen Methoden deutlich komplexer als einfachere, miteinander kommunizierende Klassen mit klar definierten Zuständigkeiten. Allerdings wird dies nicht immer so gesehen. Insbesondere OO-Anfänger haben oft mit vielen, feinen Klassen ein Problem. Diese Entwickler sehen in der Verteilung der Zuständigkeiten auf feinere, kommunizierende Klassen einen Verlust an Übersichtlichkeit und Einfachheit im System. Für diesen Entwicklerkreis ist es einfacher, wenige, aber dafür mächtigere Klassen zu pflegen, anstatt verwobene Objektgeflechte verstehen zu müssen.

Gibt es innerhalb eines Projekt-Teams diese Meinung, sollte dies vom Chef-Architekten nicht ignoriert werden: Der Autor dieses Buches spricht hier aus Erfahrung. Noch in den Zeiten der „New-Economy“ hatte ich mir eine aus meiner Sicht saubere Architektur, ausgedacht. Doch für meinen Auftraggeber – dieser entpuppte sich als echter „Host-Entwickler“ – war Java zumindest damals verdächtig und seiner Meinung nach völlig überflüssig. Die Folge: Das UML-Modell wurde ohne ein Review abgelehnt. Die Begründung: Die schlechte Wartbarkeit und Entwickelbarkeit. Ferner wurde UML als proprietär angesehen – man wunderte sich, warum keine Wolken-Diagramme direkt in Word gezeichnet werden. Nach einigen Meeting-Iterationen schließlich die Einigung (auf die ich mich einlassen musste): Verzicht auf alle „überflüssigen“ Klassen, Methoden und insbesondere auf alle Interfaces und Verteilen der gesamten Geschäftslogik auf möglichst wenige Elemente. Als kleine Rache wurde ein (Anti-)Pattern namens „Funnel“ eingeführt (= Trichter). Der Name war Programm: Das ursprüngliche „schöne“ und einfache Design wurde in wenige Klassen „gepresst“ – wie bei einem Trichter eben. Ich hatte diese Namen ohne eine Erklärung in Meetings, PowerPoints usw. eingeführt (z. B. Enterprise Funnel Architecture kurz EFA). Im Anschluss wurde die „Funnel“-Architektur immer populärer – und der Auftraggeber ahnt wohl bis heute nicht, dass es sich hierbei um ein „Anti-Pattern“ handelte. Übrigens: Diese Architektur hat nur wenige Monate überlebt – nach einigen Iterationen hat man die Klassen doch zerschlagen.

Nach einer gewissen Zeit konnten sich die meisten „Host-Entwickler“ doch mit Java anfreunden und kamen nun mit dem eigenen monolithischen Design immer schlechter zurecht. Doch gibt es auch gegenteilige Beispiele. Insbesondere junge Architekten lieben Patterns und UML-Werkzeuge. Diese Mischung ist für Projekte gefährlich: Hier entstehen hunderte von gut dokumentierten Klassen – doch ist es schwer, den Überblick zu behalten und am Ende kann oft keiner mehr sagen, wofür die einzelnen Klassen eigentlich zuständig sind. Es ist schwer vorhersehbar, bei welcher dieser Extremstrategien mehr Kosten für das Projekt entstehen.

Die Vision kann auch als eine essentielle Zuständigkeit der Anwendung aufgefasst werden. Nur sollte man die Zuständigkeiten weiter auf Subsysteme, Komponenten, Schnitt-

stellen, Klassen, Methoden und Attribute übertragen. Werden die einzelnen Zuständigkeiten übertragen, sollte zur Kontrolle immer die Frage gestellt werden: „Warum ist das Element überhaupt da?“ Die Antwort sollte möglichst klar und eindeutig ausfallen. Denn je länger die Antwort, desto verdächtiger ist betreffende Element. Sind die einzelnen Zuständigkeiten übertragen, ist ihre Dokumentation essentiell wichtig für die Wartbarkeit und Erweiterbarkeit der Anwendung und hat absoluten Vorrang vor der Dokumentation des Ist-Zustandes. Ziel ist also immer zu versuchen, das „Warum?“ anstelle des Ist-Zustands zu dokumentieren.

Geschwätzigkeit (Chattiness)

Den meisten Entwicklern ist das folgende Phänomen gut bekannt – und dient zur Vorstellung dieses Begriffs: In Standard-Meetings wird nach dem Peer-To-Peer-Schema miteinander kommuniziert. In moderierten Team-Meetings kann auch eine Hub-And-Spoke- bzw. Broker-Kommunikation stattfinden (es wird hauptsächlich mit dem Projektleiter und selten direkt kommuniziert). Und auf eine gewisse Art und Weise sind die meisten Meetings auch produktiv – es entstehen unzählige Protokolle, Refactorings der Projektpläne, Diagramme und ToDo-Listen. Aber am Ende stellt sich die Frage, ob alle Beteiligten nach einem „Standard-Meeting“ tatsächlich mehr Wissen also vorher haben bzw. ob die aufgewendete Zeit sich bei der konkreten Arbeit am Projekt tatsächlich auszahlt.

Eine grundsätzliche Frage gilt es daher zu beantworten: „Wie viel Kommunikation ist erforderlich, um das angepeilte Ziel zu erreichen.“ Auf das Thema Softwarearchitektur übertragen: Eine Anwendung ist dann geschwätzig, wenn Objekte viel miteinander kommunizieren müssen, um eine funktionale Anforderung zu erfüllen. (In verteilten Anwendungen sollte man bei der Beurteilung zusätzlich differenzieren, ob es sich um eine lokale oder entfernte Kommunikation handelt.) Je höher das „Chattiness“, desto schlechter die Performance und wahrscheinlich auch die Erweiterbarkeit, Robustheit und Wartbarkeit der Anwendung. Eine übermäßige „Remote“-Geschwätzigkeit von Objekten kann sogar das Scheitern eines Projekts verursachen.

Es gibt ein prominentes Beispiel für ein hohes Chattiness, welches anfänglich auch als Vorteil (höhere Skalierbarkeit) verkauft wurde: Die Remote-Schnittstellen der Entity Beans sind sehr geschwätzig. Für die Durchführung eines vollständigen Updates mussten potentiell sehr viele Methodenaufrufe erfolgen (die Suche mit einem Singleton-Finder und ein Aufruf eines Setters pro Attribut der Entität). Die Vision der transparenten Verteilung von persistenten Objekten auf unterschiedliche Adressräume hat in der Praxis daher noch nie funktioniert. Dieser Ansatz wird voraussichtlich auch künftig scheitern – es sei denn man schafft es, die Latenzzeiten von Hard- und Software gegen Null zu optimieren. Geschwätzige Architekturen sind oft auch „spröde“. Das hat leider nicht nur Auswirkungen auf die Performance, sondern auch auf die Wartbarkeit des Systems.

Sprödigkeit (Brittleness)

Die Geschwätzigkeit von Teilnehmern der Meetings ist im Vergleich zu der Geschwätzigkeit von Objekten bis zu einem gewissen Punkt überhaupt nicht spröde. Man merkt oft gar nicht, dass einige Mitglieder in der Runde fehlen. Natürlich, wenn nur der Projektleiter bzw. Manager anwesend sind, funktioniert das Meeting nicht mehr. Im Gegensatz zu einem Team-Meeting entstehen durch geschwätzige Anwendungen bzw. Komponen-

ten „harte“ Abhängigkeiten zwischen den betroffenen Klassen. Zunächst stellt das kein Problem dar. Doch spätestens bei dem Versuch, eine Klasse zu verändern, sind potentiell alle Kommunikationspartner von einer lokalen Änderung betroffen. Spröde Anwendungen lassen sich nur schwer pflegen, da sich sogar durch kleine Erweiterungen größere Refactorings ergeben können – und dann hat der Spruch „Never Touch A Running System“ uneingeschränkte Gültigkeit.

Die Sprödigkeit kann also als Maß der Koppelung zwischen Einheiten in einem Namensraum definiert werden. Dabei kann es sich um Methoden innerhalb einer Klasse, um Klassen innerhalb eines Packages bzw. einer fachlichen Komponente, um die direkten Abhängigkeiten zwischen Komponenten oder sogar um direkte Abhängigkeiten zwischen eigentlich unabhängigen Anwendungen handeln. Insbesondere der letzte Fall wird oft als Verkaufsargument der EAI-Toolhersteller benutzt, da die direkte Koppelung von Anwendungen hohe Integrations- und Pflegekosten verursacht. Die Sprödigkeit einer Architektur bzw. einer Anwendung lässt sich nicht immer durch die technische Entkoppelung der beteiligten Einheiten erreichen. Auch asynchrone, lose gekoppelte Anwendungen können spröde sein, da man bei diesem Vorgehen die semantische Koppelung vernachlässigt. Für die Wartung können manchmal direkte Koppelungen günstiger ausfallen, da hier bereits der Compiler in der Lage ist, die Kompatibilität von beteiligten Kommunikationspartnern zu überprüfen. In sehr lose gekoppelten Systemen wird eine derartige Inkompatibilität als Laufzeitfehler manifestiert, welcher nicht immer einfach zu finden ist ...

Auf jeden Fall ist es immer eine gute Idee, sich ein stabiles System als Ziel zu setzen. Die meisten Systeme und Architekturen sind in den ersten Iterationen erstaunlich stabil. Doch je näher die Dead-Line rückt und je öfter sich die Anforderungen ändern, desto weniger Zeit bleibt für die Refactorings. Die Kunst des Architekten besteht also darin, die Sprödigkeit kontinuierlich für die gesamte Dauer des Projekts (und auch bei jeder Erweiterung) zu überwachen und gezielte Refactoring-Maßnahmen durchzuführen (z. B. Einführung von Facaden, Controllern, Schnittstellen, Dependency Injection oder Messagings).

1.3.2 Der Architekturbaum

Der Baum - die Vorgabe

Bei der Planung von Anwendungen lohnt sich ein genauer Blick auf das betreffende Unternehmen. Denn bei diesen handelt es sich um hierarchische Organisationen, die aus einer zentralen Einheit bestehen sollten, welche über eine definierte Zuständigkeit verfügt. Diese Einheit wird weiter gegliedert – und die Anwendungen selbst entstehen in der Regel erst auf Abteilungsebene. Dabei werden diese nicht immer mit rationalen Mitteln „getrieben“. Da die Abteilungen oft miteinander konkurrieren („Kostenstellen“), lässt die abteilungsübergreifende Kommunikation bzw. der „Konzerngeist“ oft zu wünschen übrig.

In so einem Umfeld entstehen nicht selten Anwendungen mit überlappender Funktionalität. Die Entwicklung, Wartung, und Weiterentwicklung von Anwendungen erfolgt (eigentlich immer) so unkoordiniert, dass eine effiziente Wiederverwendung von bereits

bestehender Funktionalität in anderen Anwendungen eine echte Herausforderung ist. Die Folge: Diese Redundanzen treiben die IT-Gesamtkosten des Unternehmens in die Höhe, obwohl die Kalkulation auf den einzelnen Abteilungsebenen immer noch plausibel erscheint. In so einem Umfeld entstehen „Real World“-Architekturen: Und diese Realität der IT muss der Architekt berücksichtigen, ansonsten wird die Stabilität von Applikations-schnittstellen überschätzt.

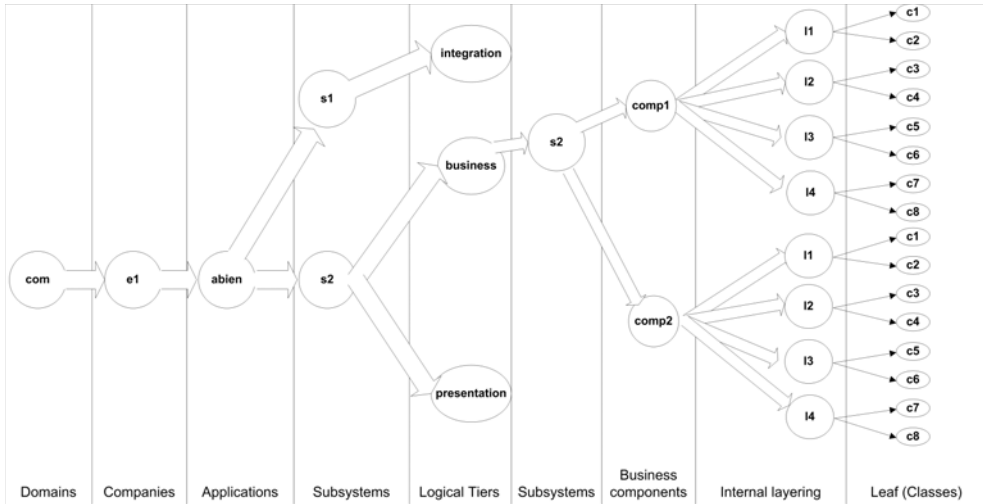


Abbildung 1.4: Der Architekturbaum

Diese hierarchische Struktur spiegelt sich auch in der Software wider. Da Anwendungen oft vollständig von einer einzelnen Abteilung realisiert werden, lässt sich die Unternehmensstruktur bis zur Abteilungsebene auch auf die Software übertragen. Dabei helfen UML-Packages, welche sich glücklicherweise auf Java-Packages oder C#-Namespaces direkt abbilden lassen.

Bei der Strukturierung ist es allerdings wichtig zu beachten, dass nicht die Abteilungen selbst, sondern vielmehr deren Verantwortlichkeiten berücksichtigt werden. Der Grund: Unternehmen werden ziemlich oft refactored (es heißt offiziell: umstrukturiert) – und diese Ereignisse sind oft schwer vorhersehbar. Dabei können sich die Namen von Abteilungen jederzeit ändern, sodass die Unternehmensstruktur selbst als höchst instabil eingestuft wird. Die Gründe für diese Entscheidungen des Managements bzw. die Auswirkungen für die Unternehmensstruktur selbst sind für einen Entwickler oder sogar Architekten (dieser ist etwas näher am Management) nur selten nachvollziehbar. Was allerdings meistens stabil bleibt, sind die Verantwortlichkeiten – auf diese kann man die Struktur der Software abbilden. Man ist daher schlecht beraten, für die Strukturierung der Software die Namen der Abteilungen zu verwenden, und manchmal ist der Name des Unternehmens bereits instabil genug. So hat sich beispielsweise während eines einjährigen Projekts eine New-Economy Firma drei Mal umbenannt und schließlich den Konkurs angemeldet ... wenigstens war so die letzte Umbenennung der Java-Packages nicht mehr notwendig.

Die Struktur der Software ist bereits bis zu dem Namen der Abteilung fest vorgegeben (z. B. com.abien.reasearch). Aber auch bei dem Entwurf der weiteren Schichten kann man die eigene Kreativität nur bedingt ausspielen. Denn nach dem Abteilungsnamen folgt meist der Anwendungs- oder Projektname. Auch die nächste Schicht strukturiert grob die Zuständigkeiten einer Anwendung mit logischen Schichten – und steht schon meistens fest: *presentation*, *business* und *integration* passen tatsächlich für die meisten Anwendungen sehr gut. In der „Presentation“ wird die Oberfläche der Anwendung abgelegt, in dem Business-Layer findet man die Geschäftslogik wieder und in die Integration-Schicht werden Adaptern/Integratoren gepackt. Diese systematische Trennung erleichtert eine saubere Schichtung der Komponenten. Nur in objektorientierten „Fat-Clients“ ist die strikte Trennung von Geschäftslogik und Präsentation nicht immer sinnvoll, da man hier auf die Idee kommen könnte, Geschäftsobjekte automatisch „rendern“ zu lassen. Damit könnte man schnell und elegant Anwendungen entwickeln. Leider bedeutet „automatisch“ immer auch „einheitlich“, worauf sich zwar sehr viele Auftraggeber sehr schnell einlassen, aber den guten Vorsatz nur wenige Iterationen lang durchhalten.

Nach dieser groben Trennung werden den einzelnen „Tier“ noch weiter gegliedert. Die Struktur der Presentation wird oft durch Frameworks (Struts, JavaServer Faces, Rich Client Platform) oder Patterns (z. B. Model View Controller) vorgegeben. Auch hier sollte man die Hauptzuständigkeiten hervorheben und das störende „Rauschen“ der technischen Details herausfiltern. Die Struktur dieser Tier ist weitgehend von der individuellen Anwendung abhängig. So können Masken für die Stammdatenpflege weitgehend zur Laufzeit aus den Metadaten der Geschäftsobjekte erzeugt werden. Dagegen müssen aufwändige Oberflächen für Expert- bzw. problemlösende Systeme oft noch „von Hand“ entwickelt werden. Das Potential für eine Wiederverwendung ist in diesem Fall sehr gering, es lassen sich auch schwer übergreifende Gemeinsamkeiten finden. Wenn dem so ist, sollte man wieder die Hauptzuständigkeiten wie z. B. Model, View, Controller berücksichtigen.

Die Business-Tier lässt sich deutlich durchgängiger organisieren, da (leider) die meisten Anwendungen sehr datengetrieben sind – oft beschäftigen sich 80 % der Geschäftslogik oder sogar mehr mit trivialen Create Read Update Delete (CRUD)-Anwendungsfällen. Die fachlichen Hauptverantwortlichkeiten der Anwendung werden dabei auf Subsysteme abgebildet. Ein Subsystem ist kohäsiv und besteht aus möglichst unabhängigen, aber dennoch kohäsiven (im Vergleich zu Komponenten aus anderen Subsystemen) fachlichen Komponenten.

Schließlich lässt sich auch das Innenleben einer Komponente durchgängig strukturieren: Man kann hier sogar eine projektübergreifende Struktur festlegen. Jede Komponente besteht ja aus einem Koordinator, einer Reihe von fachlichen aber feingranularen Geschäftsobjekten und gegebenenfalls persistenten Entitäten. Ferner verfügt eine fachliche Komponente über möglichst hart definierte Anforderungen an die Außenwelt (mit z. B. Required- bzw. Nutzer-Schnittstellen) und öffentlichen Schnittstellen, welche anderen Komponenten angeboten werden.

Eine harte Definition der Struktur der Integrationsschicht lohnt sich nur für EAI-lastige Projekte. In Standardprojekten könnte man sogar diesen Tier in die Struktur einer Komponente mit aufnehmen.

Die bisherigen Ausführungen haben ein fiktives, ideales Modell von einer Architektur entwickelt: eine hierarchische Struktur mit klar definierten Zuständigkeiten und einer klaren Abhängigkeitsrichtung. Mit diesem Ansatz würden Sie jedes Review mit Bravur bestehen, ferner wäre diese Architektur in der Praxis auch nobelpreisverdächtig, da jeder Knoten ohne eine Anpassung des Nachbarn ausgetauscht werden könnte ... nur leider ist diese Struktur zwar richtig (und bisher nahezu perfekt), jedoch völlig unrealistisch.

In der Praxis kommunizieren auch die Knoten und nicht nur die Blätter miteinander. Ohne diese unschöne Komplikation lassen sich leider keine funktionierenden Anwendungen entwickeln. Persistente Objekte von unterschiedlichen Komponenten müssen miteinander kommunizieren – oft sind auch „harte“ und direkte Referenzen wie z. B. bei der Container Managed Relation oder Methoden für die Verwaltung von Beziehungen in Hibernate, NHibernate oder Java Data Objects (JDO) erforderlich. Leider wirkt sich die Abhängigkeit der Blätter (Klassen) auch auf die gegenseitige Abhängigkeit der Komponenten, Subsysteme oder Anwendungen aus. Dies ist davon abhängig, ob sich die beteiligten Klassen in einer kohäsiven oder unabhängigen fachlichen Komponente befinden.

Die Realisierung der Abhängigkeit

Erst jetzt fängt die echte Arbeit an, denn es müssen auf jeder Ebene Abhängigkeiten erlaubt oder verboten werden. Ferner muss für jede Schicht die technische Realisierung dieser Abhängigkeit definiert werden. Nun wird es interessant: Dürfen Anwendungen überhaupt miteinander kommunizieren? Wenn ja, in welcher Form? Macht es einen Unterschied, ob Anwendungen einer Abteilung oder unterschiedlicher Abteilungen miteinander kommunizieren? Wie kommunizieren Komponenten unterschiedlicher Subsysteme miteinander? Wie sieht die Kommunikation von Komponenten genau aus?

Auch diese Fragen lassen sich manchmal anwendungsübergreifend beantworten. Dabei muss aber eine Reihe von Einflussfaktoren berücksichtigt werden wie z. B. nichtfunktionale Anforderungen, Randbedingungen, Releasezyklen der Software, Abhängigkeiten zwischen den Abteilungen, eingesetzte Programmiersprachen, die Art der Fachlichkeit und somit synchrone bzw. asynchrone Kommunikation.

Die Architektur sollte hier Anforderungen an die Realisierung der Abhängigkeiten stellen. Im Rahmen der Architektur wird also zunächst die „Vision“ der Anwendungsstruktur erarbeitet. Diese kann dann später in der Designphase verfeinert werden.

Tabelle 1.1 fasst einige Beispiele für die Umsetzung der Koppelung der unterschiedlichen Organisationseinheiten zusammen.

Schicht	Art der Koppelung	Begründung
B2B	Möglichst lose (SOAP, XML-RPC, XML ggf. Messaging), möglichst synchron. Keine binären Abhängigkeiten.	Kein Einfluss auf die Beschaffenheit der Schnittstellen des externen Partners. Kein Einfluss auf den Releasezyklus der Software Die Anwendungen können potentiell in einer anderen Programmiersprache entwickelt werden. Die B2B-Anwendungsfälle sind jedoch synchron, sodass eine synchrone Anbindung voraussichtlich einfacher zu realisieren sein wird.
Abteilung	Engere Koppelung ist zu prüfen (z. B. RMI, CORBA), möglichst synchron. Binäre Abhängigkeit denkbar.	Es existiert eine Unternehmensrichtlinie für die verwendete Technologie (Java, J2EE oder .NET). Eine gemeinsame Releasestrategie der Abteilungen wäre denkbar. Direkte Koppelungen sind einfacher realisierbar. Die Technologie der Kommunikationspartner ist bereits bekannt.
Subsystem	Direkte Abhängigkeiten sind zu vermeiden, aber erlaubt. Das Verhältnis der Kohäsion zu Kopplung sollte möglichst groß sein. Die Kommunikation darf jedoch nur über dedizierte Kanäle (Schnittstellen) erfolgen.	Alle Subsysteme werden in einer Programmiersprache implementiert (dies wäre beim Einsatz von .NET zu verifizieren). Alle Subsysteme werden in einen Container deployed. Es werden immer alle Subsysteme gleichzeitig getestet und installiert.
Tier	Die Abhängigkeiten zwischen den Tieren werden nach dem „Strict Layering“-Prinzip geregelt, d. h. der Presentation-Tier darf auf den Business-Tier zugreifen, der Business-Tier nur auf den Integration-Tier. Bidirektionale Abhängigkeiten sind nicht erlaubt. Bei der Realisierung der Abhängigkeiten ist auf jeden Fall die Kapselung der Tiere zu berücksichtigen. Direkte, binäre, „in-process“-Koppelung ist erlaubt.	Alle Tiere werden auf einem Hardwareknoten ablaufen (in J2EE wird der Web- und der EJB-Container auf einer Maschine ablaufen). Alle Tiere entstehen im Rahmen des Projekts. Bei einer Erweiterung der Geschäftslogik sind potentiell alle Tiere beteiligt, sodass eine unnötige Entkoppelung keine Vorteile mit sich bringt. Alle Tiere werden in einer Programmiersprache bzw. Plattform umgesetzt.
„Business“-Subsystem	Gruppiert zusammengehörige Komponenten. Die Kommunikation der „Business“-Subsysteme sollte im Verhältnis zu der internen Kommunikation der Komponenten eher sporadisch sein. Direkte Aufrufe der Komponenten der Business Subsysteme sind erlaubt.	„Business“-Subsysteme dienen der Gruppierung von kohäsiven Komponenten. Für die Wiederverwendung von bereits vorhandener Geschäftslogik ist die Kommunikation von Komponenten aus unterschiedlichen Subsystemen essentiell wichtig. Der Business-Tier wird immer vollständig entwickelt, getestet und installiert.

Schicht	Art der Koppelung	Begründung
Komponente	Dürfen direkt miteinander kommunizieren, wobei die Kapselung immer berücksichtigt werden muss. Es ist auch insbesondere auf „Maximal Cohesion, Minimal Coupling“ zu achten. Die Kommunikation zwischen den Komponenten findet ausschließlich über explizit ausgezeichnete, öffentliche Schnittstellen statt.	Alle Komponenten werden im Rahmen des Projekts entwickelt. Die Kommunikation zwischen den Komponenten findet immer innerhalb einer Transaktion und somit auch „in-process“ statt. Komponenten eines Subsystems werden in einem Adressraum ablaufen. Alle Komponenten werden in einem definierten Build-Prozess (Maven, Ant, NAnt) gebaut und getestet. Bei der Umsetzung von neuen Anforderungen und Bug-Fixes wird die gesamte Anwendung neu installiert.
Layering der Komponenten	Die internen Elemente einer Komponente eines größeren Layers dürfen nur auf die öffentlichen Schnittstellen einer Komponente eines feineren Layers zugreifen.	Die Kapselung der Komponenten ist unbedingt zu berücksichtigen, somit sind die Komponenten deutlich besser testbar und austauschbar. Die Komponenten der feineren Schichten sind hoch wieder verwendbar und sollten so auch in einem anderen Kontext wieder verwendet werden können. Die Komponenten des größeren Layers verfügen über mächtigere Schnittstellen und so ist ihre Wiederverwendung in einem anderen Kontext unwahrscheinlich.
Internes Layering	Die Abhängigkeiten von Klassen innerhalb einer Komponente werden klar definiert. Innerhalb einer Komponente wird ausschließlich „in process“ d. h. lokal kommuniziert. Es wird das „strict layering“ Prinzip befolgt. Bidirektional Beziehungen zwischen den Klassen unterschiedlicher Layer sind nicht erlaubt. Bidirektional Beziehungen von Klassen innerhalb eines Layers sind nicht erwünscht.	Die interne Struktur von fachlichen Komponenten ist oft stabil genug, um diese für alle Komponenten einer Anwendung oder sogar anwendungsübergreifend bzw. unternehmensweit zu definieren. Durchgängige innere Struktur erhöht die Wartbarkeit, da es ausreicht, die technische, innere Struktur einer Komponente zu verstehen. Auf ein fest definiertes Innenleben einer Komponente lassen sich einfachere Aspekte wie z. B. Transaktionen, Security, Zustandsverwaltung oder Audits übertragen. Zirkuläre Abhängigkeiten zwischen Klassen sollte man vermeiden, da sich dadurch eine Reihe von Problemen (Endlosschleifen, Serialisierungsprobleme, Reentrancy-Probleme) vermeiden lassen. Zirkuläre bzw. bidirektionale Beziehungen sollten immer begründet werden.

Tabelle 1.1: Definition der Anforderungen an die Schichten

1.3.3 Die Begleiterscheinungen

Die Erarbeitung einer Architektur ist kein Selbstzweck oder dient der Beschäftigung des Architekten bzw. ausschließlich der Nachdokumentation der Ergebnisse – sondern sie ist vielmehr für die Beschreibung einer abstrakten Systemsicht notwendig, welche für mehrere Personenkreise bestimmt ist. Diese Personenkreise nennt man Rollen oder Betrachter (Stakeholder). Dabei handelt es sich um fiktive oder reelle Personen, welche an ein-

zelen Aspekten der Architektur interessiert sind. Es kann es sich aber auch um ein Profil von mehreren Personengruppen handeln, für die die die gemeinsamen Aspekten des Systems von Bedeutung sind.

Stakeholder

Der Stakeholder beschreibt eine Rolle, welche direkt oder indirekt an der Anwendung bzw. dem System interessiert ist. Dabei sind Stakeholder kaum an allen Aspekten der Architektur eines Systems interessiert: Daher bringt die Bildung einer Sicht auf das Architekturmodell oder das Repository Vorteile und damit die einhergehenden Kommunikationsaufwände verringern sich.

„Unsichtbare“ Stakeholder sind in Projekten am risikoreichsten. Stakeholder (oder Abteilungen), welche an dem Freigabeprozess bei der Produktionseinführung zwar beteiligt sind, sind bei der Anwendungsentwicklung nicht unbedingt offensichtlich und oft unbekannt. Sie sind eine Gefahr für den Erfolg des Projektes, denn sie „stören“ den Projektablauf zunächst nicht. Die Rechtsabteilung, der Betrieb oder Usability-Experten sind Stakeholder, welche zuweilen erst recht spät erkannt werden – diese haben aber auf den erfolgreichen Abschluss des Projekts einen großen Einfluss. Leider können sich die Anforderungen der Stakeholder an das System auch überschneiden bzw. miteinander konkurrieren. So war beispielsweise in einem Webprojekt der Auftraggeber an der Auswertung des Benutzerverhaltens interessiert, wogegen die Rechtsabteilung daraufhin die Produktionseinführung verweigerte und die Verschlüsselung der Logfiles forderte.

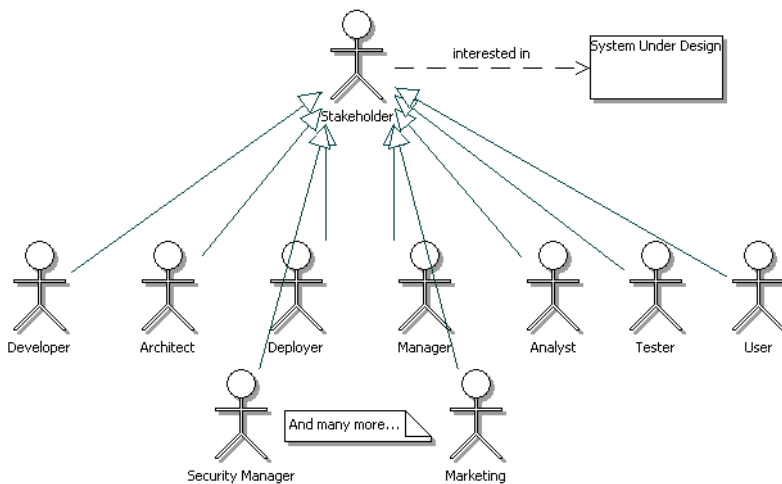


Abbildung 1.5: Die direkt und indirekt beteiligten Rollen

Interessanterweise sind nur ein kleiner Teil der Stakeholder Akteure, also Endbenutzer, die tatsächlich mit dem System arbeiten. Auch diese Benutzergruppe ist wiederum nur an einem kleinen Ausschnitt des Repositories interessiert: an der Benutzerdokumentation und den Schulungsunterlagen.

Modell

Die Definition der Architektur ist nur ein Modell, das in einem Repository abgelegt wird. Dabei ist dieses Modell nur theoretisch vollständig. In einem perfekten Repository würde man neben UML-Modellen auch Textbeschreibungen wie z. B. Word-Dateien ablegen können. Die Folge eines solchen vollständigen Modells wäre aber, dass es sehr schnell sehr komplex und unübersichtlich würde und daher die Verwendung von Stakeholder-spezifischen Sichten in der Praxis notwendig machen.

Welche Art von Modell letztendlich realisiert wird, ist von dem Kontext des Projekts und natürlich den Stakeholdern abhängig. So sind formale Architekturbeschreibungen wie z. B. UML nur dann sinnvoll, wenn diese von den Stakeholdern verstanden und auch akzeptiert werden. Oft hat beispielsweise eine kurze und exakte Textbeschreibung mehr Informationsgehalt als ein umfangreiches, aber „schwammiges“ bzw. schwer verständliches UML-Modell. Bei der Entscheidung über das Modell sollte man daher immer daran denken, dass das Ziel heißt, eine Architektur für die Stakeholder zu entwickeln und dass das Modell nicht der Evaluierung von mächtigen UML-Tools oder der Präsentation der Fähigkeiten des Architekten dient.

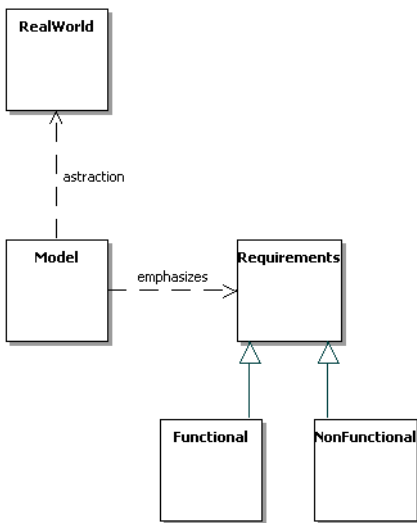


Abbildung 1.6: Die Erfassung der Fachlichkeit und der Randbedingungen

Das erstellte Modell hält die Realität fest, unter denen das Projekt steht. Die Hauptaufgabe des Modells ist jedoch nicht nur das exakte Erfassen mit allen Einzelheiten, sondern vor allem die Abstraktion von unnötigen oder irrelevanten Details und die Hervorhebung der wesentlichen Aspekte des Systems.

Kontext

Ein weiterer Faktor spielt bei der Architektur eine große Rolle: der Kontext. Denn das Projekt selbst ist ebenfalls Teil einer bereits bestehenden Umgebung oder eines Umfelds. Unter dem Begriff Kontext versteht man die Skills der Entwickler, die verfügbaren Ent-

wicklungswerkzeuge, die strategischen Entscheidungen bezüglich der Laufzeitplattformen (z. B. J2EE oder .NET) oder der verwendeten Datenbanken. Den Kontext sollte man auf jeden Fall bei der Arbeit berücksichtigen, da dadurch die Art der Entstehung der Architektur stark beeinflusst werden kann. Denn die Architektur setzt bestimmte Anforderungen an das Design und die Entwicklung voraus – und es für den erfolgreichen Abschluss eines Projekts ist es nicht ratsam, unrealistische Anforderungen an das Design oder die Entwicklung zu stellen.

Insbesondere können Aspekte wie Transaktionalität, Deployment oder Verteilung der Layer durch den Kontext des Projekts beeinflusst werden. So erlauben J2EE oder .NET einen viel natürlicheren Umgang mit Transaktionen oder mit der Verteilung von Objekten, als dies bei einer selbst entwickelten Persistenzschicht oder bei Frameworks der Fall ist. Daneben beeinflussen auch die Entwickler selbst und die Auftraggeber sowohl die Form der Beschreibung als auch die Architektur selbst: Für typische Datenbankentwickler sind beispielsweise Fat-Clients deutlich einfacher zu entwickeln als dreischichtige Architekturen mit asynchronen Ansätzen wie z. B. Java Messaging Service.

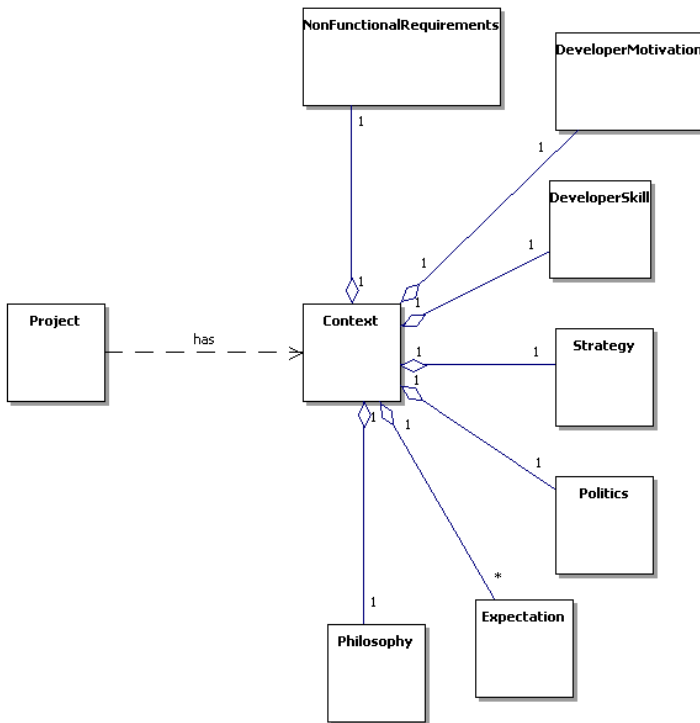


Abbildung 1.7: Das Umfeld des Projekts

Auch der Einsatz von Generatoren oder Transformatoren ist nur dann sinnvoll, wenn die Entwickler bereit sind, solche Werkzeuge auch einzusetzen. Viele Projekte sind schon sub-optimal gelaufen, weil die beteiligten Entwickler sich mit den betreffenden Werkzeugen oder Ansätzen nicht auseinandersetzen wollten und diese regelrecht sabotiert wurden.

Zwar entscheidet der Kontext alleine noch nicht über die Architektur – aber es handelt sich hier um einen wichtigen Einflussfaktor. Denn in der Praxis beeinflusst der Projektkontext ziemlich stark die Effizienz der Entwickler. Allerdings lassen sich nicht alle einzelnen Faktoren des Kontexts formal erfassen. So ist die Motivation der Entwickler manchmal viel wichtiger als ihre genauen fachlichen Qualifikationen oder ihre Kenntnisse über eine spezielle Plattform oder ein Framework. Insbesondere in anspruchsvollen Projekten ist die Arbeit mit wenigen, aber dafür hoch motivierten Entwicklern in der Regel Erfolg versprechender als mit vielen hoch qualifizierten (belegt mit vielen Zertifikaten und Trainingsteilnahmen) „Dienst-Nach-Vorschrift“-Teammitgliedern. Denn mit einem kleinen, aber gut eingespielten Team lässt sich nahezu alles erreichen. Viele bekannte Applikationsserver wurden von Zwei-Personen-Teams initial entwickelt. An anderen Produkten hatten nach formalen Wasserfallmodellen hunderte Personen gearbeitet – ihre Qualität war und ist aber nicht immer besser. Allerdings lässt sich die Motivation der Beteiligten schwer messen und ist schon allein aus menschlichen Gründen schwer dokumentierbar.

Neben der Motivation spielen auch andere „unfassbare“ Faktoren eine große Rolle. So fordert manchmal der Auftraggeber in kürzester Zeit Unmögliches oder ein Kunde nimmt direkten Einfluss auf die Beschaffenheit oder die technische Realisierung des Systems, ohne die notwendigen Kenntnisse mitzubringen. Aber auch diese Information ist wertvoll und sollte unbedingt berücksichtigt werden – auch wenn solche Feststellungen nur schwer dokumentierbar sind.

Nicht selten verlangen die Auftraggeber eine Funktionsgarantie für ein technologisch anspruchsvolles System, obwohl die Anforderungen noch nicht klar definiert sind. So hatten wir noch zu Zeiten des Internet Explorer 2 und des Netscape Navigator 2 und 3 eine Servlet-basierende eCommerce-Anwendung gebaut (es war eine Online-Shop-Lösung). Das System funktionierte bereits, wir konnten auch die Anforderungen bezüglich der Skalierbarkeit, Performance und Funktionalität erfüllen. Auch mehrere große Firmen hatten sich bereits für diese Lösung interessiert. Allerdings: Die Browser Netscape und Internet Explorer waren auch bei einfachen HTML-Seiten nicht zueinander kompatibel. Und durch die Verwendung von JavaScript wurde die Kompatibilitätsproblematik der Browser weiter verschärft. Eines Tages stellte der Auftraggeber die folgende Frage: „Wie lange würden Sie für die Implementierung eines interaktiven Beraters brauchen? Er soll Sarah Porta heißen und die Marketingabteilung arbeitet bereits an dem Aussehen dieses Avatars, sodass diese Aufgabe in kürzester Zeit bewältigt werden sollte. Oh ... ganz vergessen, Sie sollten lediglich HTML für die Implementierung verwenden, denn es gibt momentan zu viele Probleme mit JavaScript.“

Zwar war der Auftraggeber über die Problematik aufgeklärt, dass sich mit HTML kaum animierte Figuren nach den Vorgaben der Fachabteilung entwickeln lassen. Das stellte aber noch kein Problem dar, viel mehr Sorgen bereitete die künstliche Intelligenz der Beraterin. „Sarah Porta“ sollte ja abhängig von dem Inhalt des Warenkorbs entsprechende Produkte vorschlagen. Der Auftraggeber antwortete auf diese Bedenken: „Für die Animation habe ich noch keine Lösung, aber ich habe bereits einige Firmen kontaktiert, welche in diesem Bereich bereits funktionierende Lösungen haben. Die Lösung des KI-Problems ist sehr einfach: Wir lassen die Avatare einfach von unserem Call Center fernsteuern – wir haben hier genügend Mitarbeiter im Support.“

Dieses Gespräch bedeutete von meiner Seite das Ende der Zusammenarbeit, den weiteren Fortgang des Projektes habe ich fortan aus sicherer Entfernung beobachtet: Die Reali-

sierung einer Thin-Client-Anforderung wurde von den anderen Firmen sofort verworfen – allerdings wurde das Projekt noch erstaunlich lange künstlich am Leben gehalten. Dabei wurde ein wenig mit Applets, dann mit Flash experimentiert. Leider kam es nie zu der Fernsteuerung der Avatars, denn die Idee des Poolings von Sachbearbeitern in Call Centern war doch auf gewisse Art und Weise interessant. Auch könnte man anhand dieses Beispiels das Verhalten von Stateless Session Beans wirklich gut erklären.

Sicht

Die Sicht (oder auch die View) ist eine hierarchische Ansammlung von Filterkriterien, welche die wichtigen Aspekte des Modells hervorheben oder unwesentliche Merkmale unterdrücken. Somit verfügt eine Sicht über ähnliche Merkmale wie die Abstraktion. Allerdings wird eine Sicht auf das Modell stakeholder-spezifisch definiert, wobei die Abstraktionen von unwesentlichen fachlichen Merkmalen während der Entstehung der Architektur stakeholder-übergreifend sind.

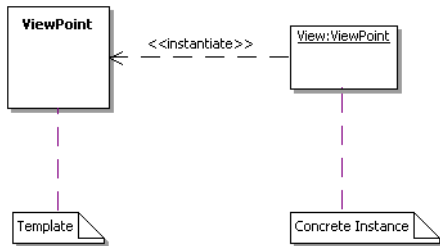


Abbildung 1.8: Die View - die Instanz eines Viewpoints

Eine Sicht kann tatsächlich das Modell visualisieren. Genauso gut kann sie aber auch interessante Dokumente oder Präsentationen für einen Stakeholder zusammenfassen.

Viewpoint (die Schablone einer Sicht)

Bei einer Sicht handelt es sich bereits um die Anwendung der Filterkriterien auf das Modell. Somit ist eine Sicht bereits die „Instanz“ oder ein konkretes Beispiel – dabei ist es interessant zu wissen, wie man Sichten bildet.

Eine Sicht wird mit Hilfe eines Viewpoints definiert. Das ist nichts anders als eine Anleitung für den Aufbau von Sichten. Dabei kann es sich hier sowohl um Word-, UML- oder andere Projekttemplates als auch um ein formales Metamodell handeln. Wenn man bei Sichten von Instanzen spricht, könnte man einen Viewpoint mit einer Klasse vergleichen. Und obwohl die bisherige Definition recht theoretisch klingt, sind Viewpoints sehr praxisnah. Beispielsweise könnte ein Viewpoint für den Stakeholder die zu verwendenden Diagramme und Beschreibungen definieren. Dabei handelt es sich lediglich um eine Einschränkung, welche die Beschreibung des Modells noch weiter schärft oder sogar formalisiert.

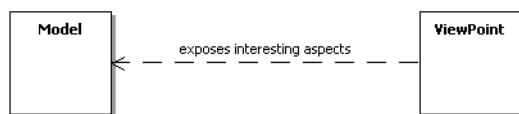


Abbildung 1.9: Die Darstellung von interessanten Aspekten eines Modells

Insbesondere die zu verwendenden Stereotypen, Modellelemente oder Darstellungsformen sollten an dieser Stelle möglichst knapp, aber genau beschrieben werden. Zusätzlich sollte eine beispielhafte Instanziierung – eine Sicht also – immer mitgeliefert werden.

Das (Meta-)Meta-Architekturmodell

Die bisher beschriebenen Artefakte stehen in gegenseitigen Beziehungen zueinander. Wenn man jedes bisher definierte Konzept als eine Klasse darstellen würde, könnte man die Beschreibung der Architektur in einem Klassendiagramm visualisieren. Dieses Klassendiagramm ist für alle Architekturen gültig.

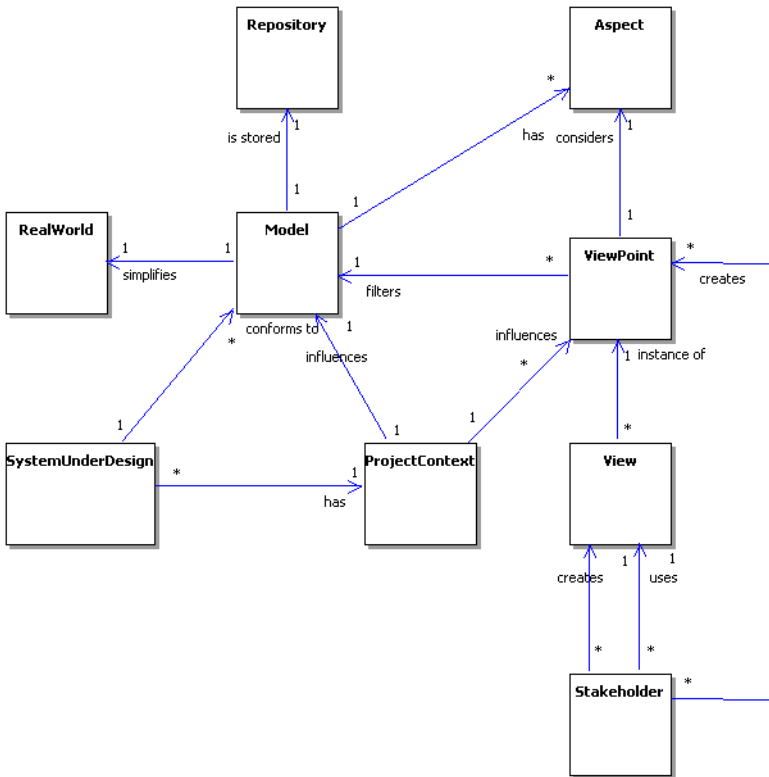


Abbildung 1.10: Die Struktur einer „Meta“-Meta-Architektur

Dabei wird ein Ausschnitt der realen Welt mit der Hilfe eines Modells festgehalten. Das Ziel dieser Aktivität ist die Vereinfachung des Sachverhalts und die Definition des Projekt-„Scopes“. Das Modell wird in einem Repository abgelegt. Dieses Repository sollte versionierbar sein und im Idealfall über ein API verfügen. Dieses könnte man dann für die Generierung des Quellcodes in der Entwicklungsphase verwenden.

Die Art der Modellierung wird durch den Projektkontext beeinflusst. Abhängig vom Kontext kann es sich bei dem Modell sowohl um ein einfaches Verzeichnis handeln, in dem Textdokumente abgelegt werden, als auch um ein UML-Modell. Auch der Informationsgehalt und die Granularität der Beschreibungen sind vom Kontext abhängig. In agilen Projekten (z. B. XP) wird die Dokumentation eher schlank ausfallen. In hardwarenahen Projekten dagegen kann sogar auf Pseudocode-Niveau dokumentiert werden. Von den Fähigkeiten der Auftraggeber und der Entwickler ist auch die Art der Erfassung der Anforderungen abhängig. Obwohl mit State-Of-The-Art-Vorgehensweisen wie z. B. UML sogar eine formalisierte Erfassung der Abläufe, Daten und Zustände möglich ist, müssen die beteiligten Stakeholder auch in der Lage sein, solche Modelle mit Hilfe von UML zu erstellen und zu verstehen. Darüber hinaus sollten auch alle Beteiligten bereit sein, UML zu diesem Zweck zu verwenden. Oft gehen durch die Unsicherheit der Stakeholder bei der Erstellung der Modelle viel Zeit und Geld verloren: So werden oft triviale und offensichtliche Sachverhalte penibel genau modelliert, die Kernaspekte dagegen vernachlässigt. Solche Modelle sind für die Entwickler völlig wertlos – eine pragmatische und schlanke Word-Dokumentation bringt an dieser Stelle oft mehr als hunderte schwammige Diagramme. Falls der Auftraggeber bereit ist, sich aktiv an dem Entwicklungsprozess zu beteiligen, können in diesem Fall viele Fragen interaktiv beantwortet und lediglich die Beschlüsse dokumentiert werden.

Das Modell hält alle Aspekte des Systems fest. Naturgemäß haben die beteiligten Stakeholder eher spezielle Interessen: So ist beispielsweise der Deployer lediglich an den Ports (z. B. 1521 für den Thin-JDBC-Treiber), der technischen Infrastruktur, den Deployment-Units wie z. B. EARS, WARs, EJB-JARs, RARs oder DLLs bzw. SOs interessiert. Die QA-Abteilung ist dagegen mehr an den Außenschnittstellen der Komponenten und gegebenenfalls an der Beschreibung und dem Verhalten der Oberflächen gelegen. Die Datenbankexperten wiederum haben eher das persistente Objektmodell und mit dem Mapping auf die Datenbanktabellen im Blick.

Diese „Filterfunktionalität“ übernehmen hierbei die Viewpoints: Viewpoints bestimmen die Regeln für die Zusammenstellung von interessanten Aspekten des Modells für die Stakeholder. In der Folge erhält dann jeder Stakeholder genau diejenigen Informationen, die er benötigt. Dabei sind Viewpoints nicht nur für UML-Modelle verwendbar. Ähnliche Funktionalität kann man auch in einfachen Textdokumenten mit Gliederung oder Verlinkung erreichen. Übertragen auf die Textdokumentation sind Viewpoints mit Dokumentvorlagen vergleichbar. Die Viewpoints selbst werden von speziellen Stakeholdern wie z. B. dem Architekten oder dem Projektleiter erstellt.

Eine Sicht ist eine konkrete Anwendung des Viewpoints. Stakeholder sind unter Umständen mehr an den Sichten interessiert – dabei können eine einzelne oder mehrere Sichten für einen Stakeholder von Bedeutung sein. So sollten beispielsweise den Architekten nicht nur die groben Zusammenhänge, sondern auch die konkrete Realisierung der Vorgaben oder z. B. Deploymentdiagramme interessieren.