gernot **STARKE**

6. Auflage

EFFEKTIVE SOFTWARE-ARCHITEKTUREN

EIN PRAKTISCHER LEITFADEN

Ideal zur Vorbereitung auf die iSAQB®-Zertifizierung

Im Internet: Hintergrundinformationen, Ergänzungen, Beispiele, Checklisten

HANSER

innoQ

Starke

Effektive Softwarearchitekturen

Gernot Starke

Effektive Softwarearchitekturen

Ein praktischer Leitfaden

6., überarbeitete Auflage

HANSER

Der Autor:

Dr. Gernot Starke, Köln
innoQ Fellow
www.gernotstarke.de

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Manfred Sommer, München
Herstellung: Irene Weilhart
Umschlagdesign: Marc Müller-Bremer, *www.rebranding.de*, München
Umschlagrealisation: Stephan Rönigk
Layout: Manuela Treindl, Fürth
Gesamtherstellung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

Print-ISBN: 978-3-446-43614-5
E-Book-ISBN: 978-3-446-43653-4

Inhalt

Vorwort

Haben Sie jemals einen dummen Fehler zweimal begangen?
– Willkommen in der realen Welt.
Haben Sie diesen Fehler hundertmal hintereinander gemacht?
– Willkommen in der Software-Entwicklung.

Tom DeMarco,
in: „Warum ist Software so teuer?“

Wenn Sie sich für Baukunst interessieren, dann erkennen Sie sicherlich die „Handschrift“ berühmter Architekten wie Frank Lloyd Wright, Le Corbusier oder Mies van der Rohe immer wieder, egal wo auf der Welt Sie auf Bauwerke dieser Meister stoßen. Die Funktionalität des Guggenheim-Museums in New York oder des Opernhauses in Sydney, gepaart mit deren Schönheit und Ästhetik, sind unvergessliche Eindrücke. Das erwarten wir heute auch von unseren IT-Systemen: Funktionalität, gepaart mit Stil!

Seit mehr als zwanzig Jahren versuche ich, Systementwicklern die Kunst des Architekturentwurfs nahe zu bringen. Die Erfahrung hat mich gelehrt, dass man jede Person, die mit gesundem Menschenverstand ausgestattet ist, zu einem guten Systemanalytiker ausbilden kann. Softwarearchitekten auszubilden ist wesentlich schwieriger.

Früher waren viele unserer Systeme so einfach, dass einzelne Programmierer die Struktur leicht im Kopf behalten konnten. Heutzutage gehört mehr dazu, um die Struktur eines Systems zu beherrschen, die Auswirkungen von Technologie-Entscheidungen vorauszusehen und die Vielzahl von Hilfsmitteln wie Generatoren, Frameworks, Libraries und Entwicklungswerkzeugen kosteneffizient und zielführend einzusetzen.

Viele Jahre war ich davon überzeugt, dass nur Erfahrung in der Erstellung großer Systeme und selbst gemachte Fehler gute Architekten hervorbringen. Wir wussten einfach zu wenig über Wirkungen und Folgewirkungen von Design-Entscheidungen. In den letzten Jahren ist die Entwicklung von Architekturen mehr und mehr zur Ingenieursdisziplin herangereift.

Gernot Starke ist es gelungen, die Essenz dieser Disziplin auf den Punkt zu bringen. Die Tipps und Tricks, die er in diesem Buch zusammengetragen hat, vermitteln Ihnen eine Fülle von Praxiserfahrungen. Selbst wenn Sie zu den Veteranen der Branche gehören, werden Sie neben vielen Déjà-vu-Erlebnissen sicherlich noch die eine oder andere Perle entdecken. Wenn Sie gerade Ihre ersten Sporen als Architekt(in) verdienen, dann können Sie sich mit den Empfehlungen den einen oder anderen Holzweg ersparen.

Trotz aller Fortschritte in der IT bleibt Konstruktion und Ausgestaltung von Architekturen dauerhaft eine Domäne für kreative Gestaltungsarbeit von Menschen und Teams. Softwarearchitekt ist daher ein Beruf mit sicherer Zukunft!

Aachen, im Januar 2014

Peter Hruschka

Vorwort zur sechsten Auflage

Softwarearchitekur: Das ist die Königsdisziplin des Software-Engineering.

Prof. Ernst Denert
in [Siedersleben04]

Motiviert durch die Herausforderungen realer Projekte, habe ich in der sechsten Auflage dieses Buches große Teile gründlich renoviert. Die Architektursichten als Werkzeug der Kommunikation und Dokumentation orientiere ich an den von Peter Hruschka und mir gestalteten (und unter *www.arc42.de* frei verfügbaren) Vorschlägen und Begriffen.

Viele Teile habe ich um Hinweise zu Architektur in agilen Projekten ergänzt. Meiner Meinung nach ergänzen sich Agilität und Architektur wunderbar.

Diese Auflage enthält eine Zusammenfassung des aktuellen iSAQB-Lehrplans zum *Certified Professional for Software Architecture* (CPSA-F), mit Hinweisen zur gezielten Prüfungsvorbereitung.

Köln, Januar 2014

Gernot Starke

1 Einleitung

Wir bauen Software wie Kathedralen:
zuerst bauen wir – dann beten wir.

Gerhard Chroust

Bitte erlauben Sie mir, Sie mit einer etwas bösartigen kleinen Geschichte zur weiteren Lektüre dieses Buches zu motivieren.

Eine erfolgreiche Unternehmerin möchte sich ein Domizil errichten lassen. Enge Freunde raten ihr, ein Architekturbüro mit dem Entwurf zu betrauen und die Erstellung begleiten zu lassen. Nur so ließen sich die legendären Probleme beim Hausbau (ungeeignete Entwürfe, mangelnde Koordination, schlechte Ausführung, Pfusch bei Details, Kostenexplosion und Terminüberschreitung) vermeiden.

Um die für ihr Vorhaben geeigneten Architekten zu finden, beschließt sie, einigen namhaften Büros kleinere Testaufträge für Einfamilienhäuser zu erteilen. Natürlich verrät sie keinem der Kandidaten, dass diese Aufträge eigentlich Tests für das endgültige Unterfangen sind.

Nach einer entsprechenden Ausschreibung in einigen überregionalen Tageszeitungen trifft unsere Bauherrin folgende Vorauswahl:

- Wasserfall-Architektur KG, Spezialisten für Gebäude und Unterfangen aller Art
- V&V Architektur GmbH & Co. KG, Spezialisten für Regierungs-, Prunk- und Profanbauten
- Extremarchitekten AG

Alle Büros erhalten identische Vorgaben: Ihre Aufgabe besteht in Entwurf und Erstellung eines Einfamilienhauses (EFH). Weil unsere Unternehmerin jedoch sehr häufig, manchmal fast sprunghaft, ihre Wünsche und Anforderungen ändert, beschließt sie, die Flexibilität der Kandidaten auch in dieser Hinsicht zu testen.

Wasserfall-Architektur KG

Die Firma residiert im 35. Stock eines noblen Bürogebäudes. Dicke Teppiche und holzvertäfelte Wände zeugen vom veritablen Wohlstand der Firmeneigner.

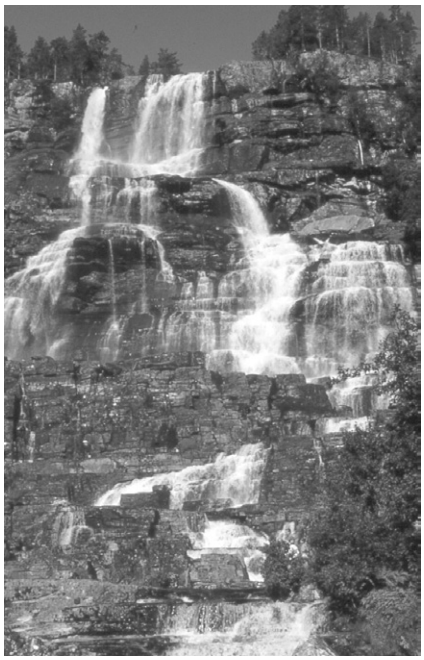


Foto von Wolfgang Korn

„Wir entwerfen auch komplexe technische Systeme“, erklärt ein graumelierter Mittfünfziger zur Bauherrin bei ihrem ersten Treffen. Sein Titel „Bürovorsteher“ prädestiniert ihn wohl für den Erstkontakt zu dem vermeintlich kleinen Fisch. Von ihm und einer deutlich jüngeren Assistentin wurde sie ausgiebig nach ihren Wünschen hinsichtlich des geplanten Hauses befragt.

Als sie die Frage nach den Türgriffen des Badezimmerschrankes im Obergeschoss nicht spontan beantworten kann, händigt man ihr ein Formblatt aus, das ausführlich ein Change-Management-Verfahren beschreibt.

Das Team der Wasserfall-Architektur KG legte nach wenigen Wochen einen überaus detaillierten Projektplan vor. Gantt-Charts, Work-Breakdown-Struktur, Meilensteine, alles dabei. Die nächsten Monate verbrachte das Team mit der Dokumentation der Anforderungsanalyse sowie dem Entwurf.

Pünktlich zum Ende dieser Phase erhielt die Unternehmerin einen Ordner (zweifach) mit fast 400 Seiten Beschreibung eines Hauses. Nicht ganz das von ihr Gewünschte, weil das Entwicklungsteam aus Effizienzgründen und um Zeit zu sparen einige (der Bauherrin nur wenig zusagende) Annahmen über die Größe mancher Räume und die Farbe einiger Tapeten getroffen hatte. Man habe zwar überall groben Sand als Bodenbelag geplant, könne das aber später erweitern. Mit etwas Zement und Wasser vermischt, stünden den Hausbewohnern später alle Möglichkeiten offen. Im Rahmen der hierbei erwarteten Änderungen habe das Team vorsorglich die Treppen als Rampe ohne Stufen geplant, um Arbeitern mit Schubkarren den Weg in die oberen Etagen zu erleichtern. Das Begehren unserer Unternehmerin, doch eine normale Treppe einzubauen, wurde dem Change-Management übergeben.

Die nun folgende Erstellungsphase (die Firma verwendete hierfür den Begriff „Implementierungsphase“) beendete das Team in 13 statt der geplanten 8 Monate. Die fünf Monate Zeitverzug seien durch widrige Umstände hervorgerufen, wie ein Firmensprecher auf Nachfrage erklärte. In Wirklichkeit hatte ein Junior-Planning-Consultant es versäumt, einen Zufahrtsweg für Baufahrzeuge zu planen – das bereits fertiggestellte Gartenhaus musste wieder abgerissen werden, um eine passende Baustraße anlegen zu können.

Ansonsten hatte das Implementierungsteam einige kleine Schwächen des Entwurfs optimiert. So hatte das Haus statt Treppe nun einen Lastenaufzug, weil sich die ursprünglich geplante Rampe für Schubkarren als zu steil erwies. Das Change-Management verkündete stolz, man habe bereits erste Schritte zur Anpassung des Sandbodens unternommen: Im ganzen Haus seien auf den Sand Teppiche gelegt worden. Leider hatte ein Mitglied des Wartungsteams über den Teppich dann, in sklavischer Befolgung der Planungsvorgaben, Zement und Wasser

aufgebracht und mit Hilfe ausgeklügelt brachialer Methoden zu einer rotgrauen zähen Paste vermischt. Man werde sich in der Wartungsphase darum kümmern, hieß es seitens der Firma.

Die zu diesem Zeitpunkt von den Wasserfall-Architekten ausgestellte Vorabrechnung belief sich auf das Doppelte der ursprünglich angebotenen Bausumme. Diese Kostensteigerung habe die Bauherrin durch ihre verspätet artikulierten Zusatzwünsche ausschließlich selbst zu verantworten.

V&V Architektur GmbH & Co. KG

Die V&V Architektur GmbH & Co. KG (nachfolgend kurz V&V) hatte sich in den vergangenen Jahren auf Regierungs-, Prunk- und Profanbauten spezialisiert. Mit dem unternehmenseigenen Verfahren, so wird versichert, könne man garantiert jedes Projekt abwickeln. Der von V&V ernannte Projektleiter überraschte unsere Unternehmerin in den ersten Projektwochen mit langen Fragebögen – ohne jeglichen Bezug zum geplanten Haus. Man müsse unbedingt zuerst das Tailoring des Vorgehensmodells durchführen, das Modell exakt dem geplanten Projekt anpassen. Am Ende dieser Phase erhielt sie, in zweifacher Ausfertigung, mehrere Hundert Seiten Dokumentation des geplanten Vorgehens.

Dass ihr Einfamilienhaus darin nicht erwähnt wurde, sei völlig normal, unterrichtete sie der Projektleiter. Erst jetzt, in der zweiten Phase, würde das konkrete Objekt geplant, spezifiziert, realisiert, qualitätsgesichert und konfigurationsverwaltet.

Foto von Ralf Harder

Der Auftraggeberin wurde zu diesem Zeitpunkt auch das „Direktorat EDV“ der Firma V&V vorgestellt. Nein, diese Abteilung befasste sich nicht mit Datenverarbeitung – die Abkürzung stand für *„Einhaltung Des Vorgehensmodells“*.

Nach einigen Monaten Projektlaufzeit stellte unsere Bauherrin im bereits teilweise fertiggestellten Haus störende signalrote Inschriften auf sämtlichen verbauten Teilen fest. Das sei urkundenechte Spezialtinte, die sich garantiert nicht durch Farbe oder Tapete verdecken ließe, erklärte V&V stolz. Für die Qualitätssicherung und das Konfigurationsmanagement seien diese Kennzeichen unbedingt notwendig. Ästhetische Einwände, solche auffälligen Markierungen nicht in Augenhöhe auf Fenster, Türen und Wänden anzubringen, verwarf die Projektleitung mit Hinweis auf Seite 354, Aktivität PL 3.42, Paragraph 9 Absatz 2 des Vorgehensmodells, in dem Größe, Format, Schrifttyp und Layout dieser Kennzeichen verbindlich definiert seien. Die Bauherrin hätte bereits beim Tailoring widersprechen müssen, nun sei es wirklich zu spät.

Extrem-Architekten AG

Die Extrem-Architekten laden unsere Unternehmerin zu Projektbeginn zu einem Planungsspiel ein. Jeden Raum ihres geplanten EFHs soll sie dabei der Wichtigkeit nach mit Gummi-

bärchen bewerten. Die immer nur paarweise auftretenden Architekten versprechen ihr eine erste funktionsfähige Version des Hauses nach nur 6 Wochen. Auf Planungsunterlagen würde man im Zuge der schnellen Entwicklung verzichten.

„Gummibär-Tango“ von Klaus Terjung

Zu Beginn der Arbeiten wurde das Team in einer Art Ritual auf die gemeinsame Vision des Hauses eingeschworen. Wie ein Mantra murmelten alle Teammitglieder ständig mit seltsam gutturaler Betonung die Silben „Einfa-Milien-Haus“, was sich nach einiger Zeit zu „Ei-Mi-Ha“ abschliff. Mehrere Außenstehende wollen gehört haben, das Team baue einen bewohnbaren Eimer. Sie stellten eine überdimensionale Tafel am Rande des Baugeländes auf. Jeder durfte darauf Verbesserungsvorschläge oder Änderungen eintragen. Dies gehöre zu einem Grundprinzip der Firma: „Kollektives geistiges Eigentum: Planung und Entwurf gehören allen“.

Nach exakt 6 Wochen laden die Extrem-Architekten die Unternehmerin zur Besichtigung der ersten funktionsfähigen Version ein. Wieder treten ihr zwei Architekten entgegen, jedoch erkennt sie nur einen davon aus dem Planungsspiel wieder. Der andere arbeitet jetzt bei den Gärtnern. Der ursprüngliche andere Gärtner hilft dem Elektriker, ein Heizungsbauer entwickelt dafür die Statik mit. Auf diese Wiese verbreite sich das Projektwissen im Team, erläutern beide Architekten eifrig.

Man präsentiert ihr einen Wohnwagen. Ihren Hinweis auf fehlende Küche, Keller und Dachgeschoss nehmen die Extrem-Architekten mit großem Interesse auf (ohne ihn jedoch schriftlich zu fixieren).

Weitere 6 Wochen später hat das Team eine riesige Grube als Keller ausgehoben und den Wohnwagen auf Holzbohlen provisorisch darüber befestigt. Das Kellerfundament haben ein Zimmermann und ein Statiker gegossen. Leider blieb der Beton zu flüssig. Geeignete Tests seien aber bereits entwickelt, dieser Fehler käme garantiert nie wieder vor.

Mehrere weitere 6-Wochen-Zyklen gehen ins Land. Bevor unsere Unternehmerin das Projekt (vorzeitig) für beendet erklärt, findet sie zwar die von ihr gewünschte Küche, leider jedoch im Keller. Ein Refactoring dieses Problems sei nicht effektiv, erklärte man ihr. Dafür habe man im Dach einen Teil der Wohnwagenküche verbaut, sodass insgesamt die Zahl der Küchen-Gummibären erreicht worden sei.

Das immer noch flüssige Kellerfundament hat eines der Teams bewogen, auf die Seitenwände des Hauses auf Dauer zu verzichten, um die Lüftung des Kellers sicherzustellen. Im Übrigen besitzt das Haus nur ein Geschoss, das aktuelle Statik-Team (bestehend aus Zimmermann und Gärtner) hat dafür die Garage in 3 Kinderzimmer unterteilt.

Weil das Team nach eigenen Aussagen auf die lästige und schwergewichtige Dokumentation verzichtet hatte, waren auch keine Aufzeichnungen der ursprünglichen Planung mehr erhalten.

Im Nachhinein beriefen sich alle Projektteams auf ihren Erfolg. Niemand hatte bemerkt, dass die Bauherrin keines der „implementierten“ Häuser wirklich akzeptierte.

Chaos nur am Bau?

Keineswegs! Ähnlichkeiten mit bekannten Vorgehensweisen bei der Softwareentwicklung sind ausdrücklich gewollt, denn nicht nur beim Hausbau herrscht Chaos.[1] Auch andere Ingenieurdisziplinen erleben *turbulente Situationen*, obwohl der Maschinenbau über mehr als 200 Jahre Erfahrung verfügt. In der Softwarebranche geht es mindestens ebenso schlimm zu.

Der regelmäßige Chaos-Report der Standish-Group zeigt eine seit Jahren gleichbleibende Tendenz: Über 30 % aller Software-Projekte werden (erfolglos) vorzeitig beendet, in über 50 % aller Software-Projekte kommt es zu drastischen Kosten- oder Terminüberschreitungen.[2]

1.1 Softwarearchitekten

Softwarearchitekten allein können diese Probleme nicht lösen. Auftraggeber mit klaren Zielvorstellungen und ein effektives und flexibles Projektmanagement sind wichtige Voraussetzungen für den Projekterfolg, ebenso ein motiviertes und sachkundiges Entwicklungsteam[3].

Architekten kommt in Software-Projekten eine besondere Rolle zu:

Softwarearchitekten bilden die Schnittstelle zwischen Analyse, Entwurf, Implementierung, Management und Betrieb von Software.

Architekten als Anwälte der Kunden

Diese verantwortungsvolle Schlüsselrolle bleibt in vielen Projekten oft unbesetzt oder wird nicht angemessen ausgefüllt. Architekten sollten als „Anwälte der Kunden“ arbeiten. Sie müssen sicherstellen, dass die Anforderungen der Kunden einerseits umsetzbar sind und andererseits auch umgesetzt werden.

Langfristigkeit

Softwarearchitekten denken langfristig – auf die gesamte Lebensdauer von IT-Systemen bezogen. Sie ermöglichen kurzfristige Änderungen, sichern gleichzeitig die Langlebigkeit und Nachhaltigkeit von Software.

Konzeptionelle Integrität

Softwarearchitekten verfolgen konzeptionelle Integrität: Die gesamte Konstruktion von Software sollte einem einheitlichen Stil folgen. Insbesondere sollten ähnliche Aufgabenstellungen in Systemen ähnlich gelöst werden. Dies erleichtert Verständnis und langfristige Weiterentwicklung.

Das Buch gibt aktiven und angehenden Softwarearchitekten praktische Ratschläge und Hilfsmittel, diese komplexe Aufgabe effektiver zu erfüllen. Es unterbreitet konkrete Vorschläge, wie Sie als Softwarearchitekt in der Praxis vorgehen sollten. Auch wenn Sie in anderen

---

[1] Viele Grüße nach Berlin. Ähnlichkeiten mit gescheiterten Bahnhof- oder Flughafenprojekten sind natürlich rein zufällig.

[2] Quelle: The Standish Group Chaos Report. Erhältlich unter *www.standishgroup.com*

[3] Eine gute Voraussetzung für Projekterfolg ist es, im Team die Eigenschaften **Kompetenz**, **Energie** und **Verantwortung** zu bündeln (danke an Dierk König für diese Formulierung.)

Funktionen in Software-Projekten arbeiten, kann dieses Buch Ihnen helfen. Sie werden verstehen, welche Bedeutung Architekturen besitzen und wo die Probleme beim Entwurf von Architekturen liegen.

1.2 Effektiv, agil und pragmatisch

Effektivität, Agilität und Pragmatismus prägen die Grundhaltung erfolgreicher Softwarearchitekten.

Agilität ist notwendig

Software wird in vielen Projekten immer noch als starres, unveränderliches Produkt betrachtet, obwohl Anwender und Auftraggeber laut nach hochgradig flexiblen Lösungen rufen. In der Praxis ähnelt die Softwareentwicklung leider oftmals eher dem Brückenbau: Eine Rheinbrücke bleibt auch in den kommenden Jahren eine Rheinbrücke. Weder verändert sich der Flusslauf, noch wird aus einer Eisenbahnbrücke eine Startbahn für Passagierflugzeuge. Für Software stellt sich die Lage ganz anders dar: Hier kann aus einem abteilungsinternen Informationssystem schnell eine Internet-E-Business-Lösung oder eine lokale Anwendung entstehen.

Langjährige Untersuchungen ergeben, dass sich 10 bis 25 % der Anforderungen an Software pro Jahr ändern (Quelle: Peter Hruschka). Management und Architekten von Software-Projekten müssen sich durch flexible und bedarfsgerechte Vorgehensweisen darauf einstellen. Das Schlüsselwort lautet „Agilität“.

Agilität und flexibles Vorgehen wird in Software-Projekten an vielen Stellen dringend benötigt:

- Requirements Manager müssen in Projekten flexibel mit Änderungen von Anforderungen umgehen.
- Softwarearchitekturen müssen stabile Grundgerüste bereitstellen, die die Umsetzung neuer und geänderter Anforderungen ermöglichen. In der heutigen Marktsituation müssen solche Änderungen schnell und effektiv erfolgen – oder sie sind wirkungslos.
- Projekt- und Produktmanager müssen in der Lage sein, während des Erstellung eines Systems flexibel auf neue Anforderungen, neue Technologien oder aktualisierte Produkte zu reagieren. Hier bietet agiles Vorgehen und risikobasiertes Projektmanagement viele Vorteile gegenüber den strikt am Vorgehensmodell orientierten konventionellen Methoden.
- Dokumentation muss sich an spezifischen Projektbedürfnissen orientieren statt an fix vorgegebenen Ergebnistypen. Inhalt ist in flexiblen Projekten wichtiger als Form.
- Agilität erfordert allerdings auch hohe Qualifikation und Professionalität der Beteiligten. Wenn Sie in einem agilen Projekt arbeiten, müssen Sie in allen Belangen mitdenken und Verantwortung übernehmen.

Insgesamt zählt in einem Projekt nur das Ergebnis. Selten kümmern sich Anwender oder Auftraggeber im Nachhinein um die Einhaltung starrer Vorgehensmodelle.

Softwarearchitekten müssen in ihrer Funktion als Schnittstelle zwischen den Projektbeteiligten diesen Ansatz der Agilität aufnehmen und in der Praxis umsetzen.

Handeln Sie agil!

Von Peter Hruschka

b-Agile

Agil heißt beweglich und flexibel sein. Mitdenken statt „Dienst nach Vorschrift" und Dogma.

Kein Vorgehensmodell passt für alle Arten von Projekten. Eine agile Vorgehensweise beurteilt das jeweilige Risiko der unterschiedlichen Aufgaben bei der Software-Entwicklung und wählt dann die geeigneten Maßnahmen. Folgende Schwerpunkte werden dabei gesetzt:

- Offen für Änderungen statt Festhalten an alten Plänen
- Eher Ergebnis-orientiert als Prozess-orientiert
- „Miteinander darüber reden" statt „gegeneinander schreiben"
- Eher Vertrauen als Kontrolle
- Bottom-up „Best Practices" austauschen und etablieren statt Top-down-Vorgaben diktieren
- Trotzdem heißt „Agilität" nicht, Anarchie zuzulassen:
- Agile Vorgehensmodelle haben Ergebnisse, nur unterscheiden sich diese für unterschiedliche Projekte in Anzahl, Tiefgang und Formalismus.
- Agile Entwicklung kennt Prozesse, nur lassen diese mehr Spielraum für Alternativen und Kreativität.
- Agile Methoden setzen auf Verantwortung; es werden nur notwendige Rollen besetzt.
- Agile Methoden basieren auf Feedback und Iterationen. Fordern und geben *(push und pull)* Sie Rückmeldung zu Ergebnissen – nur so können Teams und Ergebnisse besser werden.

Das Risiko entscheidet: Jeder Projektleiter, Systemanalytiker, Architekt, Tester und Entwickler überprüft ständig Risiken und entscheidet in seinem Umfeld über notwendige Maßnahmen, damit aus Risiken keine Probleme werden.

Dr. Peter Hruschka (hruschka@b-agile.de) ist unabhängiger Trainer und Methodenberater. Er ist Prinzipal der Atlantic Systems Guild, eines internationalen Think Tanks von Methodengurus, deren Bücher den State of the Art wesentlich mitgestaltet haben. (www.systemsguild.com).

Effektiv = Ziele erreichen

Weil das Begriffspaar „effektiv und effizient" immer wieder für Missverständnisse sorgt, möchte ich die Bedeutung beider Wörter hier kurz gegenüberstellen.

Eine Lexikon-Definition des Begriffes „Effizienz" lautet „Wirkungsgrad", also das Verhältnis von Aufwand zu Ertrag. Wenn Sie Auf-

Effizient = hoher Wirkungsgrad

gaben effizient erledigen, dann arbeiten Sie also mit hohem Wirkungsgrad. Sie investieren für den gewünschten Ertrag einen minimalen Aufwand. Spitzensportler etwa vermeiden in ihren Disziplinen überflüssige Bewegungen oder Aktionen, was in hochgradig effizientem Ausführen der jeweiligen Sportart resultiert.

Prägnant ausgedrückt, bedeutet das:

Effizient = Dinge richtig machen

Effektiv = zielorientiert

„Effektiv" bedeutet zielorientiert. Sie arbeiten effektiv, wenn Sie Dinge erledigen, die zur Erreichung Ihrer konkreten Ziele notwendig sind. Auch für diesen Begriff wieder eine prägnante Definition:

Effektiv = Die richtigen Dinge machen

Es ist viel wichtiger, die richtigen Dinge zu erledigen, als irgendwelche Dinge besonders effizient zu tun.

Softwarearchitekten müssen in hohem Maße effektiv arbeiten. Kunden und Auftraggeber bestimmen Ziele, Architekten müssen sicherstellen, dass diese Ziele auch erreicht werden.

Effektiv = agil und angemessen

Effektiv bedeutet auch, angemessen und bedarfsgerecht zu agieren. Auf die Entwicklung von Software angewandt, heißt das, sich permanent an den Bedürfnissen der Kunden und Auftraggeber zu orientieren (und nicht starr an den Buchstaben eines formalen Vorgehensmodells). Ich plädiere in diesem Buch für Agilität in diesem Sinne.

Effektive Softwarearchitektur bedeutet, das (für Kunden und Auftraggeber) richtige System konstruieren und bauen – mit technisch und organisatorisch angemessenen Mitteln.

Effektiv = pragmatisch

Projekterfolg wird grundsätzlich vom Auftraggeber beurteilt, nicht vom Architekten. Auftraggeber wollen in erster Linie ein produktives System erhalten und nicht die Einhaltung eines starren Vorgehensmodells erzwingen. Architekten müssen daher den Zweck des Systems im Auge behalten.

Pragmatisches Vorgehen bedeutet:

- Auf Dogmen und starre Vorschriften zu verzichten, wo sie nicht angemessen sind.
- Zielorientiert (= effektiv) Lösungen im Sinne der Kunden zu entwickeln.
- Auf Perfektionismus zu verzichten. 80 % des Ertrags erreicht man mit 20 % des Aufwands.

1.3 Wer sollte dieses Buch lesen?

Grundsätzlich können alle Stakeholder[4] von diesem Buch profitieren und erhalten Antworten auf zentrale Fragen.

- Projektmanager:
  - Warum sollen Sie für Architektur Geld ausgeben? Warum ist Softwarearchitektur wichtig?
- Projektleiter:
  - Was genau bewirkt Architektur im Projekt?
  - Welche Aufgaben erfüllen Softwarearchitekten?
  - Welche Bedeutung hat die Dokumentation von Architekturen („Was bedeuten all diese Symbole?“)?
  - Welche grundlegenden Lösungsansätze gibt es für Architekturen?
- Softwarearchitekten
  - Was sind die Methoden und Werkzeuge unserer Zunft?
  - Wie geht man beim Entwurf von Architekturen sinnvoll vor?
  - Welche praktisch erprobten Heuristiken und Ratschläge gibt es?
  - Wie meistert man die externen Einflussfaktoren, die den Entwurf von Architekturen erschweren?
  - Welche Sichten auf Architekturen benötigt man in der Praxis? Wie entwirft man diese Sichten?
- Software-Entwickler
  - Was bedeutet die Architektur für die Implementierung?
  - Welche allgemeinen Prinzipien von Softwarearchitektur und -entwurf sollten bei der Implementierung unbedingt befolgt werden?
- Jeder, der sich auf die Prüfung zum „Certified Professional for Software Architecture – Foundation Level“ (CPSA-F) des iSAQB e. V. vorbereiten möchte

[4] Im Verlaufe des Buches benutze ich den Begriff Stakeholder für Personen oder Organisationen, die bei der Erstellung des Systems mitwirken, sie beeinflussen oder am entwickelten System ein fachliches Interesse haben.

1.4 Wegweiser durch das Buch

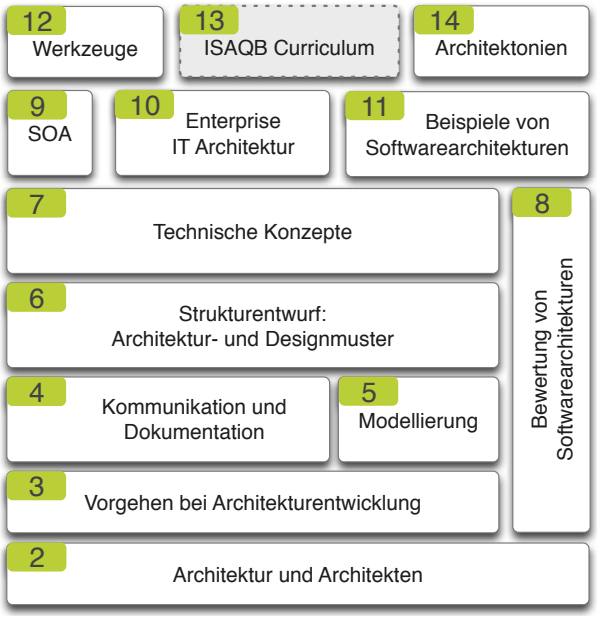


BILD 1.1
Wegweiser durch das Buch

Kapitel 2 erläutert die Begriffe „Architektur und Architekt". Es beantwortet die Fragen nach dem Was, Warum und Wer von Softwarearchitekturen.

Kapitel 3 legt den Grundstock für eine flexible und systematische Vorgehensweise zum Entwurf von Softwarearchitekturen. Es beschreibt die wesentlichen Aktivitäten bei der Architekturentwicklung, das Vorgehen „im Großen". Insbesondere erfahren Sie, wie Sie aus den Faktoren, die eine Architektur beeinflussen, angemessene Lösungsstrategien ableiten können.

arc42 Template

Kapitel 4 beschreibt, wie Sie mit praxisorientierten Sichten Ihre Softwarearchitekturen kommunizieren und dokumentieren. Jede Sicht beschreibt das System aus einer anderen Perspektive. Außerdem finden Sie hier einige Hinweise für gute Architekturdokumentation (nach arc42) sowie Strukturvorlagen für wichtige Dokumente.

Passend dazu fasst Kapitel 5 die Grundlagen der Modellierung für Softwarearchitekten zusammen. Sie finden u. a. einige Basiskonstrukte der UML, abgestimmt auf die Bedürfnisse von Softwarearchitekten.

Strukturierung von Systemen

Kapitel 6 widmet sich der Strukturierung von Systemen. Sie finden hier Architektur- und Entwurfsmuster sowie Entwurfsprinzipien, die Sie in unterschiedlichen Bereichen der Softwareentwicklung anwenden können. Es gibt Ihnen Werkzeuge an die Hand, um unerwünschte Abhängigkeiten in Ihren Entwürfen zu erkennen und aufzulösen.

(technische) Konzepte

Kapitel 7 enthält einen Katalog häufig benötigter (technischer) Konzepte. Hierzu zählen Persistenz (Datenspeicherung), Integration,

Verteilung, Kommunikation, Sicherheit, grafische Benutzeroberflächen, übergreifende Ablaufsteuerung (Workflow Management), Ausnahme- und Fehlerbehandlung sowie Management von Geschäftsregeln.

Kapitel 8 erklärt Ihnen die qualitative und Szenario-basierte Bewertung von Softwarearchitekturen.

In Kapitel 9 stelle ich Ihnen die Grundlagen Service-orientierter Architekturen (SOA) vor – sowohl aus geschäftlicher wie aus technischer Sicht.

Schließlich hebt Kapitel 10 Ihren Blick über den Tellerrand reiner Software-und Systemarchitekturen hinaus und führt Sie in die Unternehmens-IT-Architektur ein – neudeutsch Enterprise-IT-Architektur.

Beispiele von Architekturen

Kapitel 11 enthält Beispiele von Softwarearchitekturen, beschrieben nach der Strukturvorlage aus Kapitel 4.

Kapitel 12 stellt einige Kategorien von nützlichen Werkzeugen vor sowie mögliche Kriterien für deren Auswahl.

Vorbereitung auf CPSA-F

Kapitel 13 erläutert Ihnen, wie der standardisierte Lehrplan des *International Software Architecture Qualification Board* (ISAQB) inhaltlich mit diesem Buch zusammenhängt. Dieses Kapitel hilft Ihnen bei der Vorbereitung zur CPSA-F-Prüfung.[5]

In Kapitel 14 finden Sie eine etwas *andere* Zusammenfassung in Form eines Reiseberichts durch die IT-Welt.

Jedes Kapitel bietet kommentierte Literaturverweise, die Ihnen Hinweise und Empfehlungen für die Vertiefung des jeweiligen Themas geben.

Tipps, Ratschläge, Heuristiken und Regeln

Dieses Buch enthält viele praktische Tipps, Ratschläge, Heuristiken und Regeln. Damit Sie diese Regeln leichter auffinden können, sind sie typografisch durch graue Schattierung hervorgehoben.

1.5 Webseite zum Buch

Website: www.esabuch.de

Auf der Website *www.esabuch.de* finden Sie Informationen, die aus Platzgründen ins Internet weichen mussten. Dazu gehören aktuelle Literaturhinweise und Links sowie Hilfsmittel (Templates, Vorlagen) für Softwarearchitekten zum Download.

[5] CPSA-F: Certified Professional for Software Architecture

http://arc42.de

Sie können unter *www.arc42.de* den aktuellen Stand des praxisnahen und umfassend erprobten Architekturtemplates herunterladen – hilfreich für Entwurf, Entwicklung und Dokumentation von Softwarearchitekturen.

1.6 Weiterführende Literatur

[DeMarco+07] beschreiben (äußerst humorvoll und gnadenlos wahr) mehr als 80 hilfreiche „Verhaltensmuster“ aus IT-Projekten – die meiner Meinung nach alle ITler kennen sollten.

Für eilige Softwarearchitekten: In [Starke+11] haben Peter Hruschka und ich in Kurzform die wesentlichen Grundlagen von Softwarearchitektur und deren Dokumentation und Kommunikation zusammengefasst.

Da die Ursache der meisten Krankheiten in Software-Projekten in menschlichen statt technischen Problemen liegt, lege ich Ihnen eine eingehende Beschäftigung mit „Soft-Skills“ nahe – was besser durch Praxis als durch Literaturstudium funktioniert.

1.7 Danksagung

Danke an meine Traumfrau, Cheffe Uli, für Engelsgeduld, Motivation und tolle gemeinsame Zeit auf Bergen, Rädern, Yogamatten und Sofas. Du sorgst für mein glückliches Leben!

Danke an meine Diskussionspartner und Reviewer, dank deren Kommentare und Ergänzungen das Buch Form (und Inhalt) annehmen konnte: Khalid Dermoumi, Karl Eilebrecht, Phillip Ghadir, Wolfgang Keller (alias Mr. Review-Torture), Klaus Kiehne, Jürgen Krey, Christian (CK) Küster, Andreas Krüger, Alexander Nachtigall, Axel Noellchen, Michael Perlin, Robert Reimann, Lothar Piepmeyer, Peter Roßbach, Till Schulte-Coerne, Stefan Zörner.

Danke an Martin Bartonitz, Tobias Hahn, Wolfgang Korn, Stefan Tilkov und Oliver Wolf für ihre Beiträge.

Danke an das großartige Team der innoQ – ihr seid die Besten!

Ich habe ungemein von den Diskussionen und Arbeiten mit Peter Hruschka rund um arc42 profitiert. Peter hat mich ursprünglich motiviert, dieses Buch überhaupt zu schreiben.

Lynn und Per, für die wirklichen Prioritäten. Die Zeit mit euch ist immer zu kurz.

Zu guter Letzt vielen Dank an meine Kunden, dass ich in Ihren/euren Projekten so viel über Softwarearchitekturen erfahren und erleben durfte.

2 Architektur und Architekten

Architecture is about people.

Norman Foster

Fragen, die dieses Kapitel beantwortet:

- Was ist Softwarearchitektur?
- Warum ist Architektur wichtig?
- Welche Aufgaben erfüllen Softwarearchitekten?
- In welchem Kontext steht Architektur?
- Wie entstehen Architekturen?

Dieses Kapitel beantwortet die Frage „Was ist Softwarearchitektur?“. Es beschreibt Architekturen im Kontext der gesamten Softwareentwicklung. Eine zentrale Stellung kommt der vielseitigen Rolle des Softwarearchitekten zu.

Softwaresysteme werden zunehmend komplexer und umfangreicher. Damit nimmt die Bedeutung von Entwurf und Beschreibung grundlegender Systemstrukturen zu, während die Bedeutung der Auswahl von Programmiersprachen, Algorithmen und Datenstrukturen zurückgeht.

2.1 Was ist Softwarearchitektur?

Softwarearchitektur beschäftigt sich mit Abstraktion,
mit Zerlegung und Zusammenbau, mit Stil und Ästhetik.

[Kruchten95]

Von den vielen Definitionen der einschlägigen Literatur möchte ich Ihnen diejenige des IEEE-Standards[1] 1471 vorstellen:

Softwarearchitektur: Die grundsätzliche Organisation eines Systems, verkörpert durch dessen Komponenten, deren Beziehung zueinander und zur Umgebung sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten.

Dazu helfen einige Erläuterungen weiter:

Architektur enthält Strukturen

Strukturen, Zerlegung, Schnittstellen, Beziehungen

Die Architektur eines Softwaresystems besteht aus seinen Strukturen,[2] der Zerlegung in Komponenten,[3] deren Schnittstellen und Beziehungen untereinander.

Architektur definiert demnach die Bausteine eines Systems, beschreibt deren wesentliche (extern sichtbare) Merkmale und charakterisiert deren Beziehungen untereinander.

Damit enthält Architektur sowohl statische als auch dynamische Aspekte. Sie erfüllt sowohl die Aufgabe eines *Bauplans* als auch die eines *Ablaufplans* für Software.

Architektur beschreibt eine Lösung

Beschreibung einer Lösung

Die Architektur eines Systems beschreibt Strukturen und Konzepte eines Systems im Sinne von Konstruktionszeichnungen, Bauanleitungen und grundlegenden Entwurfsentscheidungen.

Für Softwaresysteme bestehen Architekturen aus (dokumentierten) Komponenten, Schnittstellen und übergreifenden Prinzipien. Erst durch eine konkrete Implementierung werden daraus reale Systeme.

Architektur basiert auf Entwurfsentscheidungen

Entwurfsentscheidungen

Die Architektur eines Softwaresystems resultiert aus einer Vielzahl von Entscheidungen. Einige davon bilden die Grundlage für den Entwurf der Komponenten, andere bestimmen über die Auswahl der eingesetzten Technologie.

---

1 http: *http://en.wikipedia.org/wiki/IEEE_1471*

2 Sie lesen richtig: Es sind mehrere unterschiedliche. Die weiter unten beschriebenen „Sichten“ beschreiben dasselbe System, aber unterschiedliche Strukturen. Im weiteren Verlauf des Buches benutze ich aus Gründen der Lesbarkeit trotzdem meistens den Singular.

3 Komponenten bezeichne ich nachfolgend meistens als Bausteine – und erkläre sie in Kapitel 4.4 genauer.

Seien Sie sich als Softwarearchitekt der Tatsache bewusst, dass Sie die Konsequenz mancher Ihrer Entscheidungen erst viel später bewerten können. So können Sie beispielsweise erst nach einigen Änderungen und Anpassungen eines Systems beurteilen, ob es die gewünschte Änderbarkeit oder Flexibilität auch wirklich erreicht hat.

Entscheidungen von besonderer Tragweite sollten Sie systematisch vorbereiten, treffen und dokumentieren – dazu gibt Ihnen Kapitel 3 weitere Hinweise.

Architektur unterstützt den Übergang von der Analyse zur Realisierung

Softwarearchitektur unterstützt den schwierigen Übergang von der Problemanalyse zur konkreten technischen Realisierung. Sie schlägt die Brücke zwischen der Fachdomäne und deren Umsetzung in Software, indem sie die geforderten Funktionen und Leistungsmerkmale auf eine Struktur von Softwarekomponenten und deren Beziehungen untereinander abbildet. Welchen Freiheitsgrad Sie als Architekt dabei haben, hängt von der Situation ab, in der Sie die Architektur entwickeln:

- Wenn Sie „auf der grünen Wiese“ entwickeln, sind Sie relativ frei in der Wahl von Komponenten und der Abbildung der Fachdomäne auf Komponenten.
- Wenn Sie ein Teilsystem innerhalb einer existierenden Systemlandschaft entwickeln, unterliegen Sie starken Restriktionen (die Sie teilweise erst ermitteln müssen).
- Wenn Sie nach einer vorgegebenen domänenspezifischen Softwarearchitektur eines von vielen ähnlichen Systemen entwickeln, reduziert sich die Architektur darauf, Fachlichkeit in eine bestehende Architektur zu gießen.

Architektur kann Systeme aus verschiedenen Sichten zeigen

Gebäudearchitekten erstellen eine Vielzahl unterschiedlicher Pläne und Sichten für Gebäude. Einige Beispiele dafür sind Grundriss, Statik, Elektro- und Heizungsplan, 3D-Sicht. Jede dieser einzelnen Sichten dokumentiert einzelne Aspekte des Gesamtsystems. Den verschiedenen Projektbeteiligten wird damit eine spezifische und ihren jeweiligen Belangen angemessene Sicht vermittelt. Jede Sicht ist für bestimmte Stakeholder nützlich.

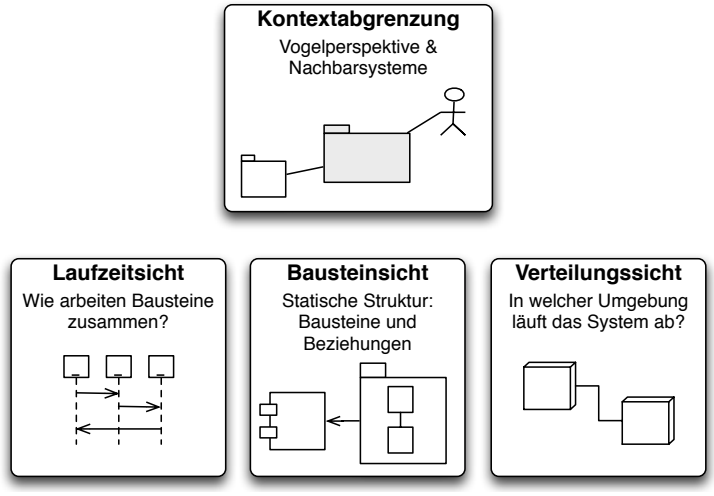


BILD 2.1
Vier Sichten auf Architektur

Unterschiedliche Sichten

Für Softwarearchitekturen gilt das analog: Sie sollten Architekturen mit Hilfe unterschiedlicher Sichten beschreiben. Die wichtigsten Sichten für Softwarearchitekturen sind die Bausteinsicht, die Laufzeitsicht sowie die Verteilungssicht, die eine Abbildung von Laufzeitbausteinen auf technische Infrastruktur (Hardware) beschreibt. Kapitel 4 beschäftigt sich ausführlich mit den Sichten, ihrer Notation und Entwicklung. In Bild 2.1 sehen Sie die Sichten im Überblick.

Architektur schafft Verständlichkeit

Architektur schafft Ordnung

Softwarearchitekturen machen Komplexität von Systemen beherrschbar und verständlich, indem sie (komplexe) Anforderungen in (geordnete) Strukturen und Konzepte übersetzen und diese übersichtlich dokumentieren.

Architektur ermöglicht Überblick

Für alle Projektbeteiligten dokumentieren sie angemessen und problembezogen die Struktur und das Zusammenwirken der einzelnen Komponenten. Architektur ermöglicht damit allen Beteiligten, die Lösung zu verstehen:

- Für das Management bilden sie die erste Möglichkeit der Verifikation, ob die Anforderungen erfüllbar sind und erfüllt werden.
- Für neue Projektmitarbeiter sind Architekturen der erste Schritt, sich mit der Struktur des Systems, seinem Entwurf und seinen Bestandteilen sowie den übergreifenden Konzepten vertraut zu machen.
- Architekturen ermöglichen Wartungsteams, die betroffenen Bestandteile leichter zu finden und die Folgen von Änderungen abzuschätzen (*impact analysis*).
- Für Betreiber von Softwaresystemen dokumentieren Softwarearchitekturen, welche Softwarekomponenten auf welchen physischen Systemteilen ablaufen. Sie stellen damit die Basis integrierter Systemarchitekturen dar, die neben Software auch Hardware und Organisationsstrukturen umfassen.

Architektur bildet den Rahmen für flexible Systeme

Architektur stellt Erweiterbarkeit sicher

Softwarearchitektur ist nach Tom DeMarco ein *„framework for change“*. Sie bildet also den Rahmen, der Flexibilität und Erweiterbarkeit von Software sicherstellt.

Softwarearchitekturen berücksichtigen dabei die Auswirkungen externer Einflussfaktoren organisatorischer und technischer Art auf das System. Diese Faktoren sind häufig Auslöser neuer Anforderungen an das System. Sie bestimmen, in welcher Hinsicht das System Flexibilität besitzen muss.

Architektur basiert auf Abstraktionen

Architektur lässt nicht benötigte Informationen gezielt weg

Im ersten Moment mag es Ihnen unglaublich vorkommen: Eine essenzielle Leistung von Architekten besteht darin, die für eine bestimmte Aufgabe nicht benötigten Informationen gezielt wegzulassen, zu abstrahieren. Diese Abstraktion besitzt eine Analogie in der Gebäudearchitektur: Eine Grundrisszeichnung enthält beispielsweise keine Informationen über Elektro- und Wasserleitungen. Hier wird bewusst Information gefiltert, um eine spezifische Darstellung der Architektur lesbar und verständlich zu halten.

Architektur schafft Qualität

Die Qualität eines Systems bezeichnet ganz allgemein die Summe seiner gewünschten Eigenschaften. Hierzu zählen Aspekte wie Performance, Verständlichkeit, Flexibilität, Robustheit, Sicherheit und natürlich auch Funktionalität.

Anforderungen an Qualitätseigenschaften, manchmal ungenau als *nichtfunktionale Anforderungen* (NFA) bezeichnet, stellen häufig die wirklich schwierigen Aufgaben für Softwarearchitekten dar. Einen bestimmten Algorithmus ohne jegliche NF-Anforderungen zu programmieren, ist erheblich leichter, als dieses Programm gleichzeitig auch noch verständlich, erweiterbar und performant für viele parallele Benutzer zu entwickeln.[4]

Die Architektur eines Systems bildet die Basis für dessen Qualität. Sie ist auch die Grundlage, um spezifische Qualitätseigenschaften bewerten zu können (Kapitel 8 erklärt Ihnen die Bewertung von Softwarearchitekturen).

In Abschnitt 3.7 (Lösungsstrategien entwickeln) finden Sie Tipps zur Erfüllung typischer Qualitätsanforderungen.

Architektur orientiert sich an Werten[5]

Orientiert an Stakeholder-Werten

Architekturentscheidungen sollen sich primär an den Wertmaßstäben der maßgeblichen Stakeholder orientieren. Falls time-to-market besonders wichtig ist, sollten Entscheidungen mit Fokus auf Entwicklungsproduktivität getroffen werden. Als Softwarearchitekt sollten Sie Ihr Handeln grundsätzlich an den Werten und Qualitätsmaßstäben der relevanten Stakeholder ausrichten – niemals primär an Ihren eigenen Zielen.

Architektur entsteht begleitend

Entsteht begleitend

Ich vertrete die Auffassung, dass Softwarearchitektur begleitend zur Entwicklung/Implementierung entstehen sollte. Bei jeglicher Änderung, Erweiterung oder Anpassung von Systemen müssen Sie irgendwelche Entwurfsentscheidungen treffen – also Architektur praktizieren.

Architektur ist Design, Design ist Architektur

„Architektur“ und „Design“ beschäftigen sich mit dem Entwurf von Systemen. Ich vertrete die Auffassung, dass die Grenze zwischen Softwarearchitektur und Software-Design fließend ist.

Damit rücken die Begriffe Design und Architektur sehr eng zusammen. Design oder Entwurf bezeichnet den Prozess der Erstellung der Architektur. Oft wird Architektur (mehr oder weniger intensiv) über die gesamte Lebendauer von Systemen praktiziert.

Manchmal gehört der Entwurf einer konkreten Klassenstruktur zur Aufgabe von Architekten, in anderen Fällen werden zusätzliche Designer diese Aufgabe lösen. Gehen Sie mit diesen Begriffen pragmatisch um, und suchen Sie nicht nach einer „formalen“ Definition.

---

[4] Das ist auch ein Grund, wieso sich Übungsbeispiele aus Programmierkursen fast nie in die Wirklichkeit übertragen lassen: Die (hohe) Komplexität nichtfunktionaler Anforderungen bleibt in solchen Beispielen in der Regel außen vor – genau **die** erschwert jedoch in der Praxis die Softwareentwicklung!

[5] Danke an Andreas Krüger für diese Formulierung.

Architektur kann sowohl Soll- wie Ist-Zustand beschreiben

Sie können die Architektur eines bestehenden Systems beschreiben, mit Strukturen, Konzepten und grundlegenden Architekturentscheidungen. Damit haben Sie den Ist-Zustand dokumentiert.

Soll- und Ist-Zustand

Andererseits können Sie Architektur zur Diskussion oder Planung zukünftiger Zustände („Soll-Zustand“) verwenden – indem Sie *mögliche* Strukturen, Konzepte oder Entscheidungen verwenden.

Diese beiden zeitlichen Dimensionen können während der Entwicklung von Systemen parallel zueinander existieren.

Der Smoketest für Softwarearchitekturen

Wenn Ihnen jemand die Darstellung einer Architektur vorlegt, stellen Sie folgende Fragen (die in einer guten Architekturdokumentation immer beantwortet sein sollten):

- Welche Grundsätze, Konzepte oder Entscheidungen liegen der Lösung zugrunde?
- Welche Verantwortlichkeiten (*responsibilities*) trägt jedes der Kästchen und Verbindungslinien in Diagrammen?
- Für jede Verbindungslinie: Warum existiert sie, welche Bedeutung hat sie? Was wird zu welchem Zeitpunkt und warum über diese Verbindungen transportiert?
- Wie erfüllen die gezeigten Bausteine die an das System gestellten Qualitätsanforderungen?

Schon mit diesen einfachen Fragen können Sie praktizierende Architekten in Projekten anregen, ihre Dokumente grundlegend zu verbessern.

2.1.1 Warum Softwarearchitektur?

Ich möchte den Nutzen und die Ziele von Softwarearchitektur nochmals zusammenfassen:

- Qualitätsanforderungen und -ziele erfüllen: Explizite konstruktive und technische Strategien, Maßnahmen, Taktiken oder Praktiken stellen die Erreichung der notwendigen Qualität sicher.
- Konzeptionelle Integrität: Ähnliche Aufgaben werden im System durchgängig ähnlich gelöst – Ausnahmen sind begründet.
- Verständlichkeit des Systems und seiner grundlegenden Entwurfsentscheidungen, Strukturen und Konzepte/Prinzipien: Angemessene Verwendung von Abstraktionen, Modellen, Dokumentation zur Erläuterung von Entscheidungen, Strukturen, Konzepten und Quellcode.
- Starker Fokus auf Langfristigkeit – daher Betonung von (langfristigen) Architekturzielen gegenüber (meist kurzfristigen) Projektzielen.
- Unterstützung des gesamten Lebenszyklus: Von der Problem- und Anforderungsanalyse über Konstruktion, Entwicklung, Implementierung bis zum praktischen Einsatz und Betrieb von Systemen, sowohl bei Neuentwicklung wie auch Wartung/Änderung von Systemen.
- Vereinfachung der Wiederverwendung von Systembestandteilen oder Konzepten.

2.2 Die Aufgaben von Softwarearchitekten

Das Leben von Softwarearchitekten besteht aus einer langen und schnellen Abfolge suboptimaler Entwurfsentscheidungen, die meist im Dunkeln getroffen werden.

[Kruchten2001]

Softwarearchitekten müssen deutlich mehr leisten, als „nur" Softwarearchitekturen zu entwerfen: sie sind „Anwälte der Kunden" und Berater für Manager und für das Realisierungsteam.

Architekten konstruieren und entwerfen

Softwarearchitekten (gemeinsam mit Entwicklungsteams) entwerfen und konstruieren alle Bestandteile, die für Entwicklung, Betrieb und Wartung eines Software-Systems notwendig sind:

- Bausteine:[6] Architekten und Entwickler konstruieren Systeme aus Bausteinen, die ihrerseits wieder komplexe Subsysteme sein können. Bausteine besitzen klare *Verantwortlichkeiten*.
- Schnittstellen: Über Schnittstellen kommuniziert ein System mit der Außenwelt. Schnittstellen bilden die Grundlage der„Verträge", auf deren Basis die Bausteine miteinander arbeiten (*design by contract*). Erst die Zusammenarbeit der Bausteine über Schnittstellen ermöglicht es dem System, seine Aufgaben zu erfüllen.
- Strukturen: Durch Bausteine und deren Zusammenwirken entwerfen Architekten sowohl statische als auch dynamische Strukturen.

Ich bin der Meinung, dass Architekten ihre Entwurfs-, Struktur- und Technologieentscheidungen am besten in einem (kleinen) Team treffen – nur in Ausnahmefällen alleine.

Architekten entscheiden

Dem bekannten Software- und Systemarchitekt Philippe Kruchten verdanken wir das Zitat zu Beginn dieses Abschnitts 2.2. Er bringt darin einige wesentliche Aspekte von Softwarearchitekten zum Ausdruck, die ich aus meiner persönlichen Erfahrung vollauf bestätigen kann:

- „*... schnelle Folge suboptimaler Entwurfsentscheidungen*": Architekten müssen viele Entscheidungen treffen: Welche Bausteine, welche Schnittstellen, welche Abläufe? Welche technischen Frameworks? Selbst implementieren, kaufen oder einen Mittelweg davon? Welches Teilteam entwickelt welche Komponenten? Wie sollen die Bausteine der Architektur heißen?[7]
- „*... die meist im Dunkeln getroffen werden*": Architekten wissen oftmals wenig über die langfristigen Konsequenzen ihrer Entscheidungen. Teilweise zeigt sich erst Monate (oder Jahre!) später, ob eine Architekturentscheidung vernünftig und angemessen war.

---

[6] Bausteine sind Komponenten, Subsysteme, Klassen, Funktionen, Module oder andere Abstraktionen von Quellcode. Siehe Abschnitt 4.4.

[7] Sinnvolle Benennung von „Dingen" trägt erheblich zur Qualität von Softwarearchitekturen und deren Implementierung bei. Vergeben Sie aussagekräftige Namen – für Schnittstellen, Subsysteme, Pakete, Komponenten, Klassen, Methoden oder sonstige Bezeichner! Benennen Sie um, wenn der Name eines Architekturbausteins nicht mehr seinen Zweck widerspiegelt. Geben Sie speziellen Bezeichnungen Vorrang vor allgemeinen.

Darüber hinaus arbeiten Softwarearchitekten oftmals mit innovativen technischen Frameworks, Betriebssystemen oder sonstigen Dingen, deren genaues Verhalten sie gar nicht kennen können. In solcher Unsicherheit hilft iteratives Vorgehen weiter (siehe Abschnitt 2.3).

Entscheidungen großer Tragweite sollten Sie angemessen dokumentieren. Am Anfang von Kapitel 3 finden Sie dazu einen Vorschlag.

Architekten garantieren die Erfüllung von Anforderungen

Architekten belegen Machbarkeit

Softwarearchitekten stellen die *Machbarkeit* und *Erfüllung* von Anforderungen sicher. Sie gewährleisten, dass Anforderungen einerseits erfüllbar sind und andererseits auch erfüllt werden. Dies bezieht sich sowohl auf die funktionalen Anforderungen (*required capabilities)* als auch auf die nichtfunktionalen Anforderungen und Randbedingungen (*required constraints*).

Architekten belegen die Machbarkeit des Systems durch Prototypen.

Nicht zuletzt sorgen Architekten auch dafür, dass Systeme mit angemessenen Kosten realisiert werden können!

Architekten beraten

Softwarearchitekten beraten andere Projektbeteiligte in architekturrelevanten Fragestellungen. Sie beraten:

- Management und Auftraggeber bei der Projektplanung und -organisation.
- Auftraggeber und Analyseteams hinsichtlich der Machbarkeit von Anforderungen. Dazu unterstützen sie bei der Bewertung von Kosten und Nutzen von Anforderungen. Sie klären die Auswirkungen von Anforderungen auf die Struktur, die Realisierung und den Betrieb von Systemen.
- Projektleiter beim Management (technischer) Risiken. Architekten sollten zusätzlich auch organisatorische Risiken kennen und berücksichtigen. Weiterhin helfen sie Projektleitern bei der Organisation und Steuerung des Implementierungsteams.
- Implementierungsteams bezüglich der Umsetzung der Architektur in Software. Dazu müssen Architekten das Team von der Architektur überzeugen und bei Bedarf ausbilden und unterstützen.
- Hardwarearchitekten und Betreiber des Systems hinsichtlich der Anforderungen, die das System an die zugrunde liegende Hardware stellt.
- Qualitätssicherung und Test hinsichtlich der Kritikalität und Testbarkeit von Systembestandteilen.

Architekten dokumentieren

Damit Projekte agil, flexibel und kurzfristig wandlungsfähig bleiben, müssen Architekten in hohem Maße angemessen und bedarfsgerecht arbeiten.

Angemessen dokumentieren

Im Wesentlichen kommt es auf angemessene Dokumentation und Kommunikation an. Sowohl Art als auch Umfang und Detaillierung müssen sich an den Bedürfnissen der jeweiligen Adressaten orientieren. Manchmal genügt eine kurze Skizze („auf der Rückseite eines gebrauchten Briefumschlags“), manchmal ein detailliertes UML-Diagramm, und in anderen Fällen treffen umfangreiche Dokumente die Bedürfnisse der Adressaten.

Kapitel 4 zeigt Ihnen, wie Sie bedarfsgerechte Dokumentationen von Architekturen erstellen, konkrete Beispiele dafür finden Sie in Kapitel 11.

Architekten sind Diplomaten und Akrobaten

Als Diplomaten schließen Architekten Kompromisse zwischen widersprüchlichen oder konkurrierenden Forderungen.

Als Akrobaten balancieren Architekten mit einer Vielzahl von Faktoren, die sich gegenseitig beeinflussen. Tabelle 2.1 zeigt einige solcher Faktoren (nach [Rechtin2000]), die im Wettbewerb miteinander stehen (und für Architekten die Arbeit interessant machen). Bild 2.2 zeigt eine Lösung in Abhängigkeit von der Stärke solcher Forderungen und Einflüsse.

Architektur enthält Kompromisse

TABELLE 2.1 Konkurrierende Faktoren, die Kompromisse und Balance benötigen

Form	⟷	Funktion
Strikte Kontrolle	⟷	Agile Entscheidungen
Kosten & Termine	⟷	Leistung und Qualität
Komplexität	⟷	Einfachheit bzw. Flexibilität
Technische Innovation	⟷	Etablierte Technologien
Top-down-Planung	⟷	Bottom-up-Realisierung
Kontinuierliche Verbesserung	⟷	Produktstabilität
Minimale Schnittstellen	⟷	Enge Integration

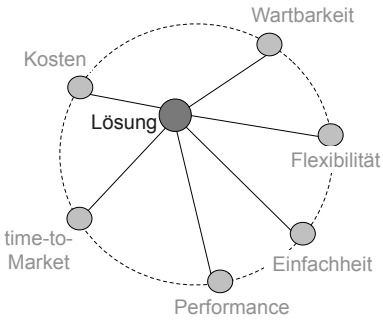


BILD 2.2
„Gummiband-Diagramm“: Lösung als Kompromiss

Architekten vereinfachen

Eine wichtige Regel für Architekten lautet: Vereinfache! Vereinfache! Vereinfache! Eine andere Lesart: Weniger ist (manchmal) mehr![8]

Entwurfsprinzipien: Kapitel 6

Einfache Strukturen sind leichter und günstiger realisierbar, einfacher verständlich, weniger fehleranfällig. Die zuverlässigste, preiswerteste und robusteste Komponente eines Systems ist diejenige, die erst gar nicht realisiert werden muss!

[8] nach [Rechtin2000], dem Kultbuch von System-Architekten.

Architekten kommunizieren

Architekten müssen den anderen Stakeholdern die Architektur vermitteln und sie von den Architekturentscheidungen überzeugen. Dazu gehört sowohl der Wille als auch die Fähigkeit, technische Sachverhalte für unterschiedliche Stakeholder angemessen aufzubereiten (siehe Kapitel 4), zu präsentieren und zu diskutieren.

Gute Softwarearchitekten zeichnen sich dadurch aus, dass sie ihre Ideen, Entwürfe und Entscheidungen aktiv an die übrigen Projektbeteiligten kommunizieren können.

- Architekten erläutern und argumentieren ihre Entscheidungen. Wenn nötig, verteidigen sie diese auch gegen Angriffe. Hierfür benötigen sie diplomatisches Geschick – davon wird im folgenden Abschnitt die Rede sein.
- Architekten überzeugen das Projektteam von Strukturen und Schnittstellen, indem sie Vor- und Nachteile transparent darstellen und unter den Rahmenbedingungen und Einflüssen des konkreten Projektes gegeneinander abwägen.
- Architekten präsentieren und vermarkten die Architektur, sodass möglichst alle Projektbeteiligten sie als DIE Architektur akzeptieren und in technischer Hinsicht an einem Strang ziehen.
- Sie unterrichten und coachen: treten als Berater, Trainer oder Lehrer auf, um das benötigte Know-how im Team zu verbreiten.

Für diese Aufgaben benötigen Softwarearchitekten starke kommunikative Fähigkeiten, sollten motivieren, präsentieren und argumentieren können. Fundierte technische Kenntnisse stellen lediglich eine *notwendige* Voraussetzung für diesen Teil der Architekturarbeit dar.

Architekten bewerten

Zielerreichung bewerten

Architekten müssen die Güte der Architekturen bewerten, um jederzeit den Grad der Zielerreichung zu kennen. Sie müssen wissen, ob und an welchen Stellen der Systeme nichtfunktionale Anforderungen (wie etwa Performance) riskant oder kritisch sind. Aus dieser objektiven Bewertung heraus können Architekten Maßnahmen zur Optimierung oder Risikominderung ableiten – in der Regel gemeinsam mit anderen Projektverantwortlichen.

Mehr zur Bewertung von Architekturen finden Sie in Kapitel 10.

Architekten explizieren

Architekten sollten wichtige Annahmen und Voraussetzungen explizit machen, um die möglichen Probleme *impliziter Annahmen* zu adressieren. Sie vermeiden dadurch mögliche Missverständnisse zwischen Beteiligten.

Architekten brauchen Mut

Unter Unsicherheit entscheiden

Am Ende eines erfolgreichen Projektes ist es einfach, Entwurfsentscheidungen zu kritisieren. Zu diesem Zeitpunkt wissen alle Beteiligten über Technologien, Produkte und auch Anforderungen genau Bescheid. Architekten mitten im Projektstress können solche Dinge teilweise nur vermuten. Architekten verfügen aus Zeitgründen oftmals nicht über genügend Informationen, um optimale Entscheidungen zu treffen. Damit das Projekt weiterlaufen kann, müssen

Architekten in solchen Fällen Mut zu (möglicherweise) suboptimalen Entscheidungen unter Unsicherheit aufbringen.

Beachten Sie den Unterschied zwischen Mut und Waghalsigkeit:

Mutig bedeutet, manche Risiken bewusst einzugehen, auch gegen den Willen anderer. Architekten benötigen Mut zu unbequemen Entscheidungen, zu potenziellen Konflikten mit anderen Projektbeteiligten, zu frühzeitigem Eingeständnis von früheren Fehlentscheidungen.

Waghalsigkeit hingegen nenne ich schnelle Entscheidungen ohne bewusste Risikoabwägung, ohne Beachtung von Konsequenzen oder ohne Prüfung möglicher Alternativen.

Architekten können die Schuld nicht auf diejenigen schieben, die die Anforderungen gestellt haben. Allerdings sollten Architekten die übrigen Projektbeteiligten möglichst frühzeitig auf mögliche Konsequenzen bestimmter (kritischer) Anforderungen hinweisen (und damit das Risikomanagement ihrer Projektleiter unterstützen!).

Die Werkzeuge von Architekten:

- Modelle, also vereinfachte Abbildungen und Abstraktionen der Wirklichkeit.
- System-Dokumentationen dienen Architekten als Grundlage effektiver Kommunikation mit anderen Projektbeteiligten. Eine Dokumentation enthält neben Modellen noch weitere Informationen. Siehe dazu Kapitel 4.
- Heuristiken (griechisch: *heuriskein*, einen Weg finden, führen): Erfahrungen, Regeln, Tipps. Dieses Buch enthält viele solcher Heuristiken und Tipps. Siehe dazu insbesondere Kapitel 6.
- Muster (*patterns*): einfach anwendbare Vorlagen oder Schablonen für elegante Lösungen zu spezifischen Entwurfsproblemen.[9] Durch Muster können Architekten Wiederverwendung auf hohem Niveau betreiben. Beispiele für Entwurfs- und Architekturmuster finden Sie in Kapitel 6.
- Zerlegung (Partitionierung): Architekten beherrschen Komplexität durch Zerlegung in Teilprobleme. Sie finden mehr über Zerlegung in Kapitel 6.
- Zusammensetzung (Aggregation): das Gegenstück zur Zerlegung. Architekten setzen Einzelteile zu Software-Systemen zusammen.
- Iteration: Software-Projekte brauchen iterative Prozesse und kurzfristiges Feedback. Daher sollten auch Architekten in Iterationen und Zyklen vorgehen. Anders ausgedrückt: Bei der Vielzahl von Einflussfaktoren und Anforderungen an Software ist es häufig schwierig, schon im „ersten Wurf" das Ziel exakt zu treffen.
- Compiler, Debugger und Prototypen: Diese Hilfsmittel benötigen Architekten, wenn es gilt, das Team bei der Implementierung zu unterstützen, die Machbarkeit von Implementierungsentscheidungen zu verifizieren oder technische Risiken zu überprüfen (*architect also implements*, nach [Coplien95]).

[9] In der Pattern Community sind auch Muster für organisatorische Probleme, Prozessprobleme sowie Implementierungsprobleme beschrieben. Siehe *http://www.hillside.net/~patterns*

Fazit: Darum kümmern sich Softwarearchitekten!

Sie haben in den letzten Abschnitten das Aufgabenspektrum von Softwarearchitekten kennengelernt. Bild 2.3 zeigt, worum sich Softwarearchitekten in IT-Projekten kümmern sollten.

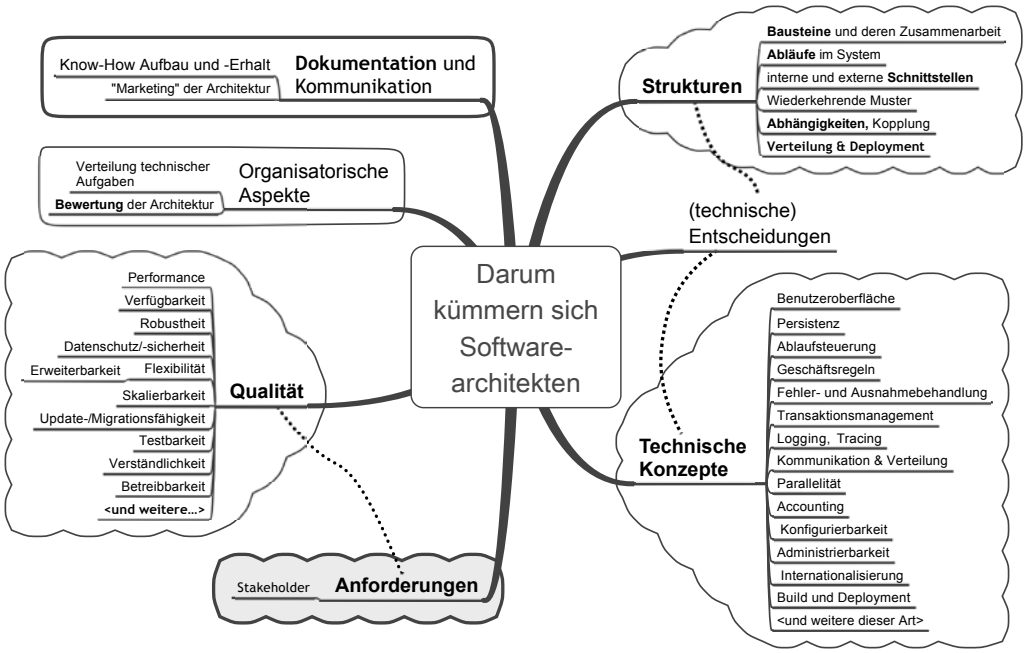


BILD 2.3 Zusammenfassung der Aufgaben von Softwarearchitekten

Der nächste Abschnitt gibt Ihnen einige Anhaltspunkte, wie Sie die Fülle dieser Aufgaben angehen können: iterativ und in kleinen Teams.

Kapitel 3 vertieft dann, wie Sie diese Aufgaben praktisch umsetzen können.

2.3 Wie entstehen Architekturen?

In Zyklen und Iterationen

Iterativ und inkrementell vorgehen

Die Anforderungen an Software, ihre Randbedingungen und Einflussfaktoren ändern sich. Am Ende eines Projektes sieht die tatsächliche Lösung immer etwas anders aus, als zu Projektbeginn geplant war. Die Entwicklung von Software-Systemen ähnelt der Verfolgung von beweglichen Zielen (*moving targets*). Bild 2.4 illustriert, wie sich Zwischenlösungen und das bewegliche Ziel im Projektverlauf durch Iterationen immer besser einander annähern.

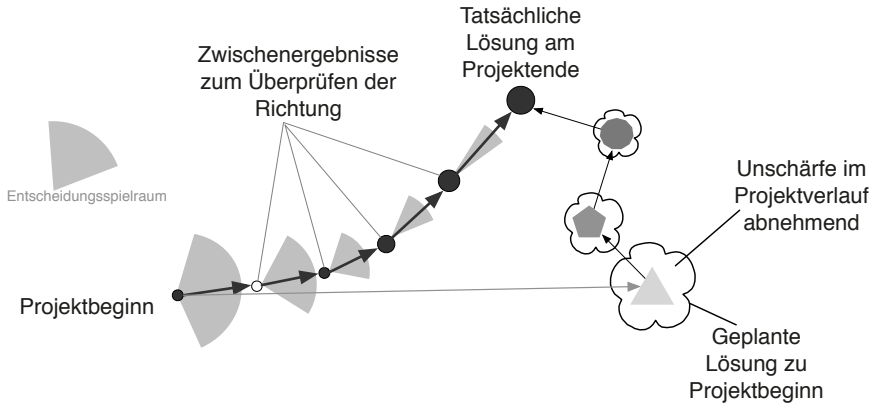


BILD 2.4 Verfolgung des „Moving Target" im iterativen Entwicklungsprozess (mit freundlicher Genehmigung von Klaus Eberhardt, *www.iteratec.de*)

Architecture Business Cycle

Gute Architekturen entstehen in Zyklen und Iterationen. Entwurfsentscheidungen können Anforderungen und damit organisatorische Abläufe beeinflussen (siehe Bild 2.5). [Bass+03] nennen diesen Zyklus den „*Architecture Business Cycle*".

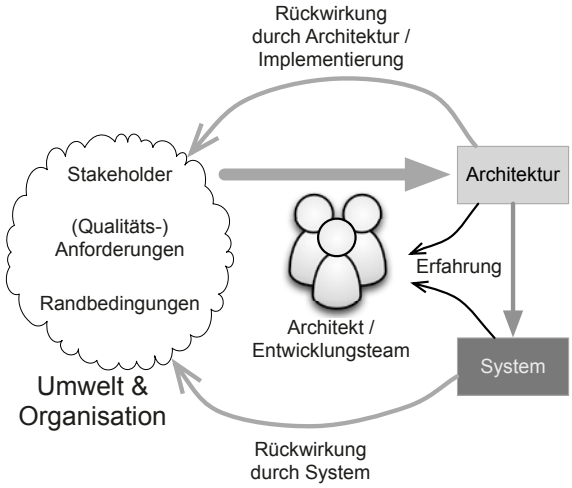


BILD 2.5
Rückwirkungen zwischen Architektur, Organisation und System (nach [Bass98])

Die Abbildung zeigt, dass sowohl die Architektur als auch das fertige System Rückwirkungen auf die Organisation beinhalten. Architekten gewinnen durch diese Rückkopplungen Erfahrung, indem sie die Auswirkungen der von ihnen gestalteten Systeme auf Organisationen erleben. Beachten Sie in diesem Zusammenhang auch die zeitlose Regel von Conway:[10]

[10] Er hat sie schon 1968 formuliert. Eine kurze Erläuterung von Open-Source-Guru Eric Raymond finden Sie unter *http://catb.org/~esr/jargon/html/C/Conways-Law.html*

Die Strukturen von Organisationen und der von ihnen entwickelten Architekturen sind isomorph.

Architekturen, Organisationen und Systeme beeinflussen sich gegenseitig.

In kleinen Teams

Gute Architekturen sind das Ergebnis eines kleinen Teams mit dem gemeinsamen Ziel, die Anforderungen des Kunden zu erfüllen.

Gemischte Erfahrung: Software und Fachdomäne

Zum Erfolg braucht dieses Team eine gute Mischung aus Erfahrung im Software-Engineering und in der jeweiligen Fachdomäne. Es benötigt Flexibilität hinsichtlich seiner internen Struktur und Arbeitsweise. Die einen werden vielleicht Architekten von Subsystemen, andere fachliche und organisatorische Berater des Kunden.

Wichtig ist jedoch ein klar identifizierter Teamleiter[11] mit folgenden Stärken:

- In Konflikten vermitteln und sie lösen helfen
- Mutige Entscheidungen treffen. „Mutig“ bedeutet hierbei „notfalls unter Unsicherheit “.
- Motivieren
- Kommunizieren
- Strukturieren

Vorgehen: Kapitel 3 und 6

Als Antwort auf die Frage nach dem „Wie“ von Softwarearchitekturen stelle ich Ihnen in Kapitel 3 und 6 flexible *methodische Werkzeuge* vor, mit denen Sie Architekturen entwerfen können.

Wie Architekturen *nicht* entstehen sollten

- Im Architekturkomitee, entkoppelt von der Realität echter Systeme.
- Im Elfenbeinturm, losgelöst von Kunden, Auftraggebern, der Projektleitung und dem Realisierungsteam.
- Nur auf bunten Marketing-Folien.[12]
- Als „Wir machen jetzt <Name-der-bevorzugten-Technologie>“. Die Datenbank ist nicht die Architektur, ebenso wenig das Netzwerk, der Transaktionsmonitor, REST, die 4-GL-Sprache oder ein beliebiger Standard. Diese Begriffe sind Teile oder Aspekte der Architektur, aber die Architektur kann nicht nur aus einem isolierten Aspekt bestehen [Kruchten2001].

[11] Sie finden in Projekten für diese Rolle manchmal andere Bezeichnungen, etwa: Chefarchitekt, Chief Technical Officer (CTO) oder Technischer Projektleiter.

[12] Diese Art von Architekturdokumentation, in der Regel stark abstrahiert, vereinfacht und für nicht-technische Projektbeteiligte ausgelegt, kann ein wichtiges Instrument zur Kommunikation der Architektur, beispielsweise an Manager oder Auftraggeber, sein.

2.4 In welchem Kontext steht Architektur?

Zentrale Rolle

In Softwareprojekten nimmt Architektur eine der zentralen Rollen ein. Sie fungiert als Brücke zwischen Analyse und Implementierung. Gleichzeitig dient die Architektur fast allen Projektbeteiligten als Leitbild oder Referenz.

Bild 2.6 zeigt, wie Softwarearchitektur mit anderen Entwicklungsaktivitäten zusammenhängt. In allen Fällen beeinflussen sich die Aktivitäten gegenseitig.

Das stellt ein weiteres Argument für die iterative Entwicklung dar. Durch Iterationen können Architekten die teilweise konkurrierenden Einflüsse und Anforderungen ausbalancieren.

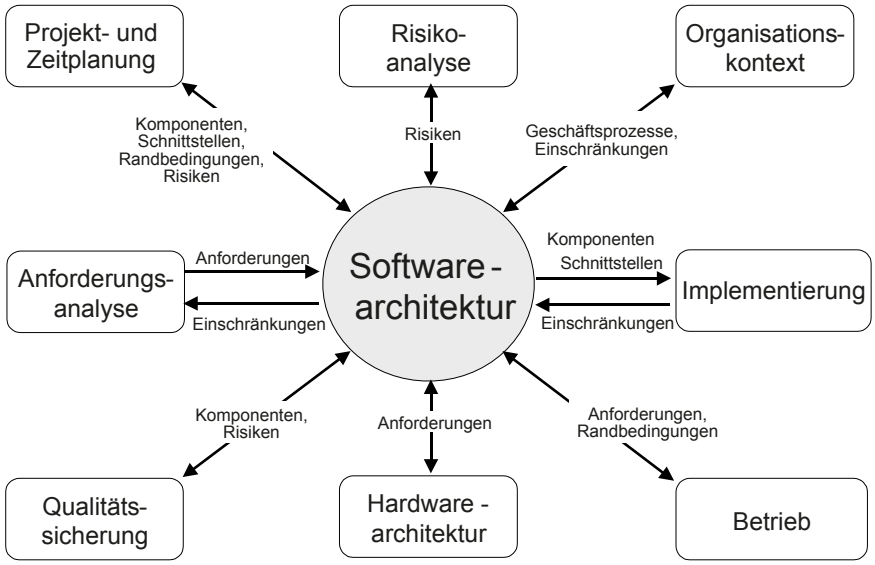


BILD 2.6 Architektur und andere Entwicklungsaufgaben

Architektur und Anforderungsanalyse

In der Anforderungsanalyse werden die fachlichen und technischen Anforderungen an das System formuliert. Dazu gehören:

- funktionale Anforderungen (Required Capabilities);
- nichtfunktionale (Qualitäts-)Anforderungen (Required Constraints).

Auf Basis dieser Anforderungen und der externen Einflussfaktoren entsteht die Softwarearchitektur. Architekten prüfen die genannten Anforderungen hinsichtlich ihrer Umsetzbarkeit und stimmen gegebenenfalls Änderungen mit den Analyseteams ab.

Architektur beeinflusst Anforderungen

Hierbei kommt es häufig vor, dass die Architektur weitreichende Rückwirkungen auf die Anforderungen hat. Architekten entscheiden häufig über die technische Machbarkeit fachlicher Anforderungen. Sie üben damit auch maßgeblichen Einfluss auf die detaillierte Projektplanung aus.

Ein Auftraggeber wird eventuell auf eine „teure“ Anforderung verzichten, um den Erfolg des Gesamtprojektes zu sichern oder ein vernünftiges Kosten/Nutzen-Verhältnis einzuhalten.

Es gehört zur Aufgabe des Architekten, die Anforderungen zu präzisieren, sie zu vervollständigen, ihre Auswirkungen zu prüfen und ihre Machbarkeit sicherzustellen!

> *Ein Architekt, der zu Beginn seiner Arbeit vollständige und konsistente Anforderungen benötigt, mag ein brillanter Entwickler sein – aber er ist kein Architekt.*
>
> *[Rechtin2000]*

Analyse = das Richtige entwickeln

Die Analysephase eines Projektes bestimmt oftmals dessen fachlichen Umfang (*scope*). Sie stellt sicher, dass das „richtige“ System entwickelt wird. Die Entwurfsphase stellt sicher, dass man das System „richtig“ entwickelt (nach [Larmann2001]).

Entwurf = richtig entwickeln

Der Scope eines Projektes ist die wichtigste Entscheidung überhaupt. Ein unklar definierter und bei den Entscheidern umstrittener Projektumfang gehört zu den größten Projektrisiken!

Architektur und Projektplanung

Aufgabenplanung

Großen Einfluss besitzt die Architektur auch auf die Projektplanung. Auf Basis der zu realisierenden Komponenten und Schnittstellen kann die Projektleiterin eine detaillierte und realistische *Work Breakdown Structure* und einen entsprechenden Projektplan erstellen.[13] Die Architektur bietet der Projektleitung die Möglichkeit, eine detaillierte Aufgabenplanung vorzunehmen, insbesondere für die Aktivitäten innerhalb der Implementierungsphase sowie die Hard- und Softwarebeschaffung für Entwicklungs-, Test- und Produktionsumgebung.

In agilen Projekten können Architekten dem Entwicklungsteam oder beispielsweise dem Scrum-Master und Product-Owner helfen, User-Stories, Features und deren Risiken und technische Konsequenzen zu durchdringen.

Architektur und Implementierung

Architekturen bilden den abstrakten Rahmen, der durch die Implementierung zum laufenden System wird. Architekten konstruieren (oft gemeinsam mit dem Entwicklungsteam) die zu implementierenden Bausteine, entwickeln Prototypen oder Musterlösungen als *Referenzarchitekturen*. Technische Details der Implementierung wiederum können die Architektur beeinflussen, indem sie Randbedingungen vorgeben.

Architekten wirken als Dienstleister des Entwicklungsteams, sie beraten in technischen oder strukturellen Fragen, helfen bei der Definition und Implementierung zentraler Bausteine und Schnittstellen. Oft erleichtern sie dem Team die ungeliebten Dokumentations- und Kommunikationsaufgaben. Außerdem treffen Architekten, in Abstimmung mit dem Team, wesentliche Architekturentscheidungen.

---

[13] Leider zeigt die Praxis oftmals ein anderes Bild: Projektleiter planen ohne Kenntnis von Architekturen. Dies führt im weiteren Projektverlauf zu Problemen, weil es zwei unterschiedliche Planungsstrukturen gibt: eine gemäß der Projektleitung und eine im Sinne der Architektur.

Architektur und Risikoanalyse

Risikomanagement

Architekten können (und sollten) zur Risikoanalyse und zum Risikomanagement von Projekten erheblich beitragen. Sie können technische und organisatorische Risiken erkennen und bewerten und darauf abgestimmte, angemessene Lösungsstrategien zusammen mit der Projektleitung entwickeln.

Arbeiten Sie als Architekt beim Risikomanagement eng mit Projektleitung, Kunden und Auftraggebern zusammen. Das gibt Ihnen die Möglichkeit, Risiken und passende Maßnahmen im Sinne von Lösungsstrategien aktiv zu steuern.

Mehr über Risikomanagement, Einflussfaktoren und Lösungsstrategien finden Sie in Kapitel 3.

Architektur und Organisationskontext

Organisatorische Faktoren: Kapitel 3

Softwaresysteme entstehen in einem organisatorischen Kontext, der ihre Entwicklung und damit verbunden auch die resultierende Architektur maßgeblich beeinflusst. Zu den organisatorischen Faktoren zählen etwa:

- Entwicklungsprozess
- Motivation und Erfahrung aller Beteiligten
- Termin- und Kostendruck

Manche Organisationen benutzen Vorgehensmodelle (wie z. B. das V-Modell [VMH97] oder den Rational Unified Process [Kruchten2001]), die den gesamten Prozess der Systementwicklung mehr oder minder detailliert beschreiben. Solche Vorgehensmodelle prägen durch ihre teilweise detaillierten Vorgaben die Erstellung von Architekturen.

Umgekehrt kann die Architektur auch organisatorische Abläufe bestimmen. Betriebswirtschaftliche Ablauf- oder Prozessoptimierung kann sich an Vorgaben oder Gegebenheiten von Systemarchitekturen orientieren.

Architektur und Betrieb

Die Architektur bestimmt die Art der Laufzeitkomponenten und beeinflusst damit den Betrieb eines Systems. Die für den Betrieb verantwortlichen Stakeholder müssen eine der Architektur entsprechende Laufzeitumgebung für das System bereitstellen. Diese umfasst beispielsweise Prozessoren, Speicher, Netzwerke, Betriebssysteme, Datenbanken und sonstige Software-Services.

(Software-)Architektur und Hardwarearchitektur

Bei der Erstellung komplexer Systeme müssen Software- und Hardwarearchitekten Hand in Hand arbeiten, weil sich beide gegenseitig beeinflussen. Insbesondere betrifft dies Qualitätsanforderungen, wie etwa Performanz, Hochverfügbarkeit oder Sicherheit.

Besonders kritisch stellt sich die Zusammenarbeit zwischen Software- und Hardwarearchitekten bei den so genannten eingebetteten Systemen (*Embedded Systems*) dar, weil dort essenzielle

Systemfunktionen sowohl von Soft- als auch von Hardware wahrgenommen werden können. Details dazu finden sich in [Hatley2000] oder [b-agile2002a].

Architektur und Qualitätssicherung

Auf Basis der Softwarearchitektur kann die Qualitätssicherung eines Projektes gemeinsam mit dem Architekturteam die gestellten Anforderungen hinsichtlich ihrer Test- und Prüfbarkeit bewerten.

Daneben sollte die Qualitätssicherung auf Basis der Architektur die Testplanung vornehmen. Die so genannten Black-Box-Tests können auch ohne Kenntnis der Architektur geplant werden, doch besitzen in vielen Fällen auch White-Box-Tests große Bedeutung für die Qualität von Systemen. Diese Tests beziehen sich explizit auf einzelne Systemkomponenten.

2.5 Weiterführende Literatur

[Bass+03] ist eine ausführliche und gründliche Einführung in das Thema Softwarearchitektur.

[Coplien95] beschreibt eine Vielzahl von „Organizational Patterns“, die für Architekten große Bedeutung besitzen. Dazu gehören unter anderem: „Architekt steuert das Produkt“, „Architekt implementiert“, „Entwurf zu zweit“, „Belohne Erfolg“ und einige andere. Diese Muster sind für Architekten wie für Projektleiter und Auftraggeber gleichermaßen wertvoll.

[Hatley2000] beschreibt einen Prozess zum systematischen Entwurf von Softwarearchitekturen, der sich besonders gut für *Real-Time* und *Embedded* Systeme eignet.

[Hofmeister2000] erläutert, wie Einflussfaktoren und Randbedingungen die Erstellung von Softwarearchitekturen prägen.

[Martin08] zeigt die Bedeutung guten Quellcodes auf. Architekten können daraus viel über Verständlichkeit und Dokumentation lernen.

[Rechtin2000] ist eine ausführliche und verständliche Einführung in „System-Architektur“. Die Autoren zeigen viele Gemeinsamkeiten zwischen Architekturen in unterschiedlichen Fachgebieten (etwa: Flugzeugbau, Gebäudearchitektur, Softwarearchitektur). Sie motivieren die Aufgabe der Architekten als „Anwälte der Kunden“. Eine Kernthese lautet: Architekten müssen auf Heuristiken (als kodifiziertes Erfahrungswissen) zurückgreifen. Das Buch enthält mehr als hundert solcher „Ratschläge“, die Sie auf jede Art von Architektur anwenden können.

[SEI2001] vergleicht verschiedene Definitionen des Begriffs „Softwarearchitektur“ miteinander. Der Artikel zeigt eindrucksvoll, wie unterschiedlich und vielseitig die Fachwelt diesen Begriff interpretiert.

In [Starke+09] erklären Peter Hruschka und ich Ihnen in Kurzform die Aufgaben von Softwarearchitekten.

3 Vorgehen bei der Architekturentwicklung

Erfahrung ist die härteste Lehrerin.
Sie gibt Dir zuerst den Test und anschließend den Unterricht.

Susan Ruth, 1993

Fragen, die dieses Kapitel beantwortet:

- Wie sollten Softwarearchitekten vorgehen?
- Was sind die typischen Aufgaben von Softwarearchitekten?
- Wie entwickeln Sie eine „erste Vorstellung" vom System?
- Wie finden Sie die relevanten Einflussfaktoren?
- Wie berücksichtigen Sie diese Einflüsse beim Entwurf von Architekturen?
- Wie erreichen Sie Qualität (nichtfunktionale Anforderungen)?
- Welche typischen Risiken drohen bei Softwarearchitekturen?
- Welche Lösungsstrategien adressieren diese Risiken?

Wie sollen Architekten vorgehen?

Die schlechte Nachricht zuerst: Es gibt kein deterministisches Verfahren, das in jedem Fall zu guten Softwarearchitekturen führt.[1] Sie werden sich praktisch immer iterativ-inkrementell an die Lösung heranarbeiten müssen.

Und jetzt die gute Nachricht: Bild 3.1 zeigt einige grundlegende Aktivitäten, die Ihnen beim effektiven Entwurf von Architekturen helfen.

Diese Aktivitäten beziehen sich sowohl auf die Neuentwicklung als auch auf die Änderung (= Wartung, Evolution, Sanierung) von Systemen. Auf einige Besonderheiten von „Änderungs- oder Wartungsfällen" gehe ich in Abschnitt 3.7 ein.

Konzepte und Ideen wieder verwenden

Sie sollten sich in jedem Fall über mögliche Wiederverwendung informieren: Wer hat eine ähnliche Aufgabe vor Ihnen gelöst, und wie? Sammeln Sie Lösungsideen, anstatt immer neu zu entwerfen.[2]

[1] Architekturen und Systeme würden dann automatisch generiert, und Sie hätten dieses Buch nicht gekauft.

[2] Bei der Wiederverwendung helfen Ihnen Architekturmuster und –stile (finden Sie in Kapitel 6) sowie technische Konzepte (Kapitel 7).

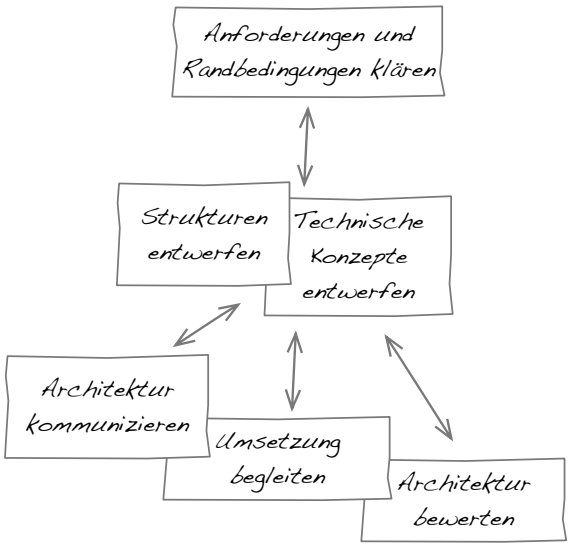


BILD 3.1
Systematischer Entwurf von Softwarearchitekturen

- Klären Sie Anforderungen mit den *maßgeblichen* Beteiligten. Im Idealfall liegen Ihnen dazu aktuelle, inhaltlich korrekte und präzise Beschreibungen aus dem *Requirements Engineering* vor. Siehe Abschnitt 3.2.
- Identifizieren Sie projektspezifische Randbedingungen und Einflussfaktoren (= Einschränkungen). Diese bilden *Leitplanken* Ihrer Entwurfsentscheidungen. Aus Anforderungen und Einflussfaktoren leiten Sie Risiken ab, die den Erfolg Ihrer Architekturarbeit bedrohen. Siehe Abschnitt 3.3.
- Nun treffen Sie (am besten in einem kleinen Team) Entwurfsentscheidungen über Strukturen und Konzepte. Entwickeln Sie Lösungsideen und -ansätze auf Basis der Anforderungen und Randbedingungen. Validieren Sie Entwürfe und Entscheidungen möglichst frühzeitig durch konkrete Implementierung. Hierzu finden Sie weitere Informationen in den Kapiteln 6 und 7.
  - Entwerfen Sie Strukturen: Welche Bausteine gibt es, wie arbeiten sie zusammen, wie und wo laufen sie ab? Diese Strukturentscheidungen folgen grundlegenden Entwurfsprinzipien, „*Best Practices*“ und Heuristiken. Außerdem sollten Sie Architekturstile und -muster, Referenzarchitekturen oder ähnliche Vorlagen zurate ziehen. Häufig beginnen Sie mit fachlichen Strukturen und erweitern diese dann sukzessive um technische Details. Genaueres lernen Sie in Kapitel 6 (Strukturentwurf) kennen.
  - Entwerfen Sie übergreifende (technische) Konzepte, wie Persistenz, Benutzeroberfläche, Verteilung, Sicherheit u. a. Das gesamte Kapitel 7 dieses Buches unterstützt Sie dabei. Das geschieht in der Regel parallel zum Entwurf der Strukturen – daher stehen diese beiden in Bild 3.1 zusammen.
- Kommunizieren Sie Ihre Softwarearchitekturen bedarfsgerecht und angemessen. In Kapitel 4 stelle ich Ihnen dazu einige Sichten vor, die spezifische Interessen der verschiedenen Projektbeteiligten berücksichtigen. Kommunizieren Sie sowohl schriftlich als auch mündlich!

- Während der Implementierung sollten Sie kontinuierlich Rückmeldungen Ihres Entwicklungsteams einholen. Überwachen Sie, ob Ihre Entscheidungen angemessen umgesetzt werden beziehungsweise die von Ihnen gewünschten Auswirkungen zeigen. Ein paar Tipps dazu finden Sie in Abschnitt 3.5.
- Schließlich sollten Sie sich regelmäßig fragen, ob Ihre Architektur den Anforderungen Ihrer Kunden genügt – sowohl in funktionaler wie in qualitativer Hinsicht. Kapitel 8 zeigt Ihnen den Weg zur systematischen Bewertung Ihrer Architekturen. Auch daraus leiten Sie Risiken ab – die Ihnen wiederum Anregungen für konkrete Entwurfsentscheidungen geben.

Architekturen entstehen iterativ und inkrementell

Architektur entsteht iterativ

Einige Aktivitäten aus Bild 3.1 können parallel stattfinden – in Ihrem Kopf oder auf verschiedene Personen verteilt. In jedem Fall klären Sie regelmäßig, welche Anforderungen und Einflussfaktoren sich geändert haben oder welche neuen Erkenntnisse das Entwicklungsteam gewonnen hat – daraufhin müssen Sie eventuell Ihre Entscheidungen überarbeiten und die Architektur Ihres Systems anpassen.

Gehen Sie inkrementell vor, d. h. treffen Sie neue, zusätzliche Entscheidungen auf der Grundlage früherer Erfahrungen und vorhandener Strukturen, Konzepte und Implementierung. Auf diese Weise kann Ihr System auf der Basis von Feedback und konkreter Erfahrung *wachsen*.

Ich möchte Ihnen das iterative Vorgehen bei der Architekturentwicklung noch aus einer weiteren Perspektive verdeutlichen (als ergänzende Erklärung zu Bild 3.1):

Architekturen müssen sowohl fachliche Anforderungen wie auch Qualitätsanforderungen erfüllen. Sie können sich als Architekt oder Entwicklungsteam diesem Problem aus unterschiedlichen Richtungen annähern:

- Top-down: vom Abstrakten zum Detaillierten
- Bottom-up: von Details oder feingranularen Teilen zum Abstrakten
- Outside-in: ausgehend von externen Schnittstellen das Innere des Systems gestalten

Bild 3.2 zeigt eine solche Perspektive auf iteratives Vorgehen bei der Architekturentwicklung:

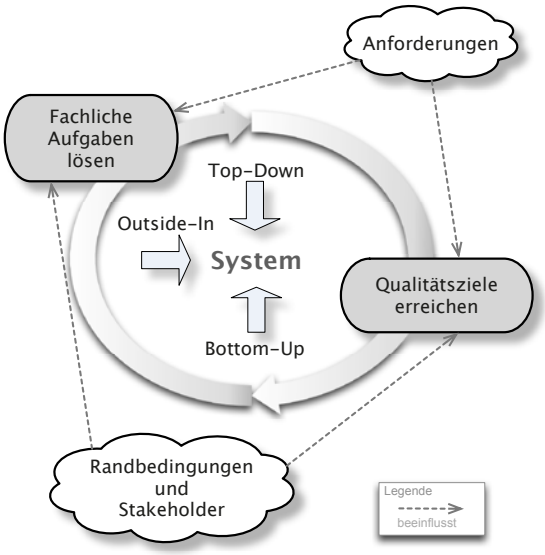


BILD 3.2
Fachliche und Qualitätsziele iterativ erreichen

Architekturen entstehen (meistens) im Team

Softwarearchitekten sollten meiner Ansicht nach mit (überschaubaren) Teams entscheiden, um einerseits das Wissen dieser Teams zu nutzen, andererseits die Akzeptanz von Entscheidungen frühzeitig zu sichern.

Verabschieden Sie sich von dem Gedanken, Softwarearchitekturen „ein für alle Mal“ zu entwickeln. Arbeiten Sie iterativ. Entwickeln Sie Ihre Entwürfe in Zyklen. Bleiben Sie agil, und nehmen Sie Rückkopplungen der Projektbeteiligten und der Organisation in Ihre Entwürfe auf. So entstehen Architekturen, die an den wirklichen Bedürfnissen und Anforderungen orientiert sind!

Bild 3.3 zeigt schematisch, dass sowohl Ihre Kunden wie auch andere Projektbeteiligte Anforderungen und Einflussfaktoren ändern. Aufgrund dieser Änderungen müssen Sie Ihren Architekturentwurf und Ihre Architekturentscheidungen iterativ den veränderten Gegebenheiten anpassen.

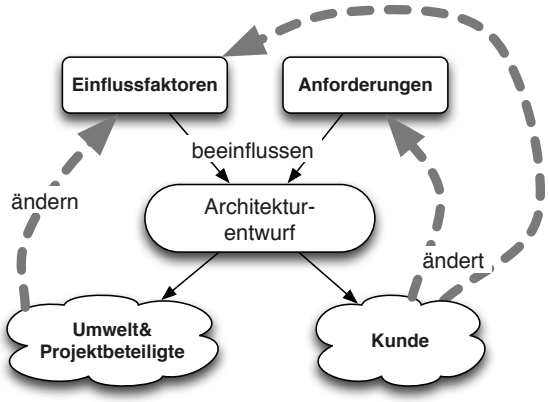


BILD 3.3
Iterativer Entwurf
aufgrund veränderter Anforderungen

Im vorigen Kapitel haben Sie unter dem Begriff „Architecture Business Cycle“ bereits eine andere Art der *Iteration* von Architekturen kennengelernt.

Treffen Sie Architektur- und Entwurfsentscheidungen bewusst, angemessen und systematisch

Machen Sie sich vor und während Ihrer Architekturentwicklung bewusst, welche architekturrelevanten Entscheidungen Sie oder andere Stakeholder treffen müssen. Stimmen Sie mit betroffenen Stakeholdern ab, ob und wann diese Entscheidung notwendig wird – es kann sowohl für schnelle wie auch für verzögerte Entscheidungen gute Gründe geben.

Entscheidungen dokumentieren

Wenn Sie eine Entscheidung für besonders wichtig oder bemerkenswert halten, dann sollten Sie diese dokumentieren: Halten Sie dazu den genauen Entscheidungsumfang („Was ist das Problem?“) sowie die Auslöser für diese Entscheidung („Warum müssen wir das entscheiden?“) fest, sowie

die Rahmenbedingungen, Ihre getroffenen Annahmen, Risiken und mögliche (verworfene) Alternativen.[3] In jedem Fall sollten Sie Ihre Entscheidung begründen.

Entwickeln Sie Mut zu Entscheidungen! Akzeptieren Sie, dass Sie so manche unsichere Entscheidung treffen müssen (und erinnern Sie sich bisweilen an das Zitat von Philippe Kruchten aus Abschnitt 2.2 ...).

3.1 Informationen sammeln

Noch bevor Sie eigene Lösungsideen entwickeln, sollten Sie recherchieren, wie andere Architekten vor Ihnen ähnliche Probleme gelöst haben:

Quellen für Wiederverwendung finden

- Beginnen Sie mit Ihrer eigenen Erfahrung – Ihre Kunden erwarten sowohl Domänenwissen als auch technisches Wissen von Ihnen.
- Prüfen Sie die Projekte innerhalb Ihrer Organisation, ob sich Ergebnisse wieder verwenden lassen. Seien Sie dabei pragmatisch!
- Suchen Sie im Internet nach Beschreibungen ähnlicher Systeme. Vielleicht können Sie passende Komponenten dazukaufen.
- Suchen Sie in der technischen Literatur, und durchforsten Sie die Entwurfsmuster auf Ihrem Gebiet.

In vielen Fällen bekommen Sie dadurch Beispiele und Muster und müssen nur noch kleine Teile völlig neu entwerfen.

Sie sollten ein tiefes Misstrauen gegenüber solchen Lösungen hegen, die sich in den Grundzügen nicht auf bekannte Dinge zurückführen lassen. Nur die wenigsten, die solche (angeblich innovativen) Lösungen präsentieren, sind wirklich geniale Erfinder – die meisten haben nur schlecht recherchiert.

[3] Stefan Zörner schlägt vor, bei wichtigen Entscheidungen auch die Namen der Entscheider festzuhalten.

3.2 Anforderungen klären

Als Grundlage Ihrer Entwurfstätigkeiten sollten Sie eine Vorstellung wichtiger Systemeigenschaften und -anforderungen besitzen und mindestens folgende Fragen über das System beantwortet haben:

- Was ist die Kernaufgabe des Systems (= funktionale Anforderungen)? Welches sind die wichtigsten Elemente der Fachdomäne?
- Um welche Kategorie von System handelt es sich?
- Welche Qualitätsanforderungen muss das System erfüllen (= Architekturziele)?
- Welche Personen oder Gruppen haben Interessen am System (= Stakeholder)?
- Was sind die (fachlichen und technischen) Nachbarsysteme (= externe Schnittstellen, Kontextabgrenzung)?

Die Antworten auf diese Fragen werden Ihnen (und dem Entwicklungsteam) eine Orientierung für erste Lösungsansätze geben.

3.2.1 Was ist die Kernaufgabe des Systems?

- Beschreiben Sie die Kernaufgabe und Verantwortlichkeit des Systems in zwei bis drei Sätzen. Formulieren Sie positiv, und benutzen Sie die Kernbegriffe der Fachdomäne.
- Fügen Sie die wichtigsten Begriffe oder Aspekte der Fachdomäne hinzu; wenige Begriffe (5–10) genügen häufig.
- Stimmen Sie diese Formulierung mit Kunden und Auftraggebern ab!

Beispiel: Das System unterstützt Call-Agents bei der Erfassung und Bearbeitung von Schadensmeldungen von Privatkunden.

Die wichtigsten Aspekte der Fachdomäne sind Vorfall, Schaden, Partner und Vertrag.

Eine solche kurze Formulierung erleichtert die Kommunikation über das System. Sie definiert für alle Beteiligten das wichtigste Ziel (= das System). Gleichzeitig schafft sie einen begrifflichen Kontext, an dem sich alle Beteiligten orientieren können.

Manchmal können Sie diese Aussagen aus der Anforderungsanalyse übernehmen. Andernfalls müssen Sie die Kernaufgabe mit Ihren Auftraggebern oder Kunden selbst formulieren.

3.2.2 Welche Kategorie von System?

Meist geben die Kernaufgaben eines Systems bereits Aufschluss, zu welcher der folgenden Kategorien es gehört:

- Interaktive Online-Systeme: Auch als operationale Systeme bezeichnet, arbeiten diese Systeme als Teil der normalen Geschäftsprozesse in Unternehmen. In den meisten Fällen enthalten sie Operationen auf Daten (Transaktionen, Einfüge-, Änderungs- und Löschoperationen), die vom Ablauf her in die Benutzeroberfläche eingebettet sind. Die Art der Transaktionen ist festgelegt. Die Systeme operieren auf möglichst aktuellen Datenbeständen. Sie erfordern ein hohes Maß an Systemverfügbarkeit und Performance.
- Mobile Systeme: Eine (moderne) Variante von Online-Systemen – durch die starke Verbreitung von Smartphones und Tablet-Computern sehr wichtig geworden. Starker Fokus auf Benutzeroberfläche und Benutzbarkeit, oftmals per Internet an Backend-Systeme angebunden.
- Entscheidungsunterstützungssysteme (*decision support system*): arbeiten oftmals auf Kopien der aktuellen Unternehmensdaten (*data warehouse*) und enthalten hauptsächlich lesende Datenzugriffe. Die Art der Anfragen an die Daten ist flexibel. Benutzer können neue Anfragen (*queries*) formulieren. Daher ist die Laufzeit von Anfragen im Vorfeld kaum abschätzbar. Solche Systeme tolerieren höhere Ausfallzeiten und geringere Performance.
- Hintergrundsysteme (Offline-Systeme, Batch-Systeme): dienen hauptsächlich der Datenmanipulation, oftmals zur Vor- oder Nachverarbeitung vorhandener Datenbestände; werden zur Interaktion mit anderen Systemen eingesetzt. In Kapitel 7 finden Sie mehr zum Thema Integration.
- Eingebettete Systeme (*embedded systems*): arbeiten eng verzahnt mit spezieller Hardware. Ein Beispiel für eingebettete Systeme sind Mobiltelefone (viele haben auch Echtzeitanforderungen).
- Systeme mit Echtzeitanforderungen (*real-time systems)*: Operationen werden innerhalb garantierter Zeiten fertiggestellt. Beispiel: Produktionssteuerung (Fließbänder mit festen Taktzeiten), Herzschrittmacher. Solche Systeme werde ich im Folgenden nicht mehr weiter ausführen – sie sind eine Wissenschaft für sich. ☹

Wenn das System zu mehreren der genannten Kategorien gehört, sollte die Architektur aus entsprechenden Teilsystemen bestehen, die jeweils zu einer Kategorie gehören.

3.2.3 Wesentliche Qualitätsanforderungen ermitteln

Qualität, laut Duden definiert als „Beschaffenheit, Güte, Wert“, stellt ein wichtiges Ziel für Softwarearchitekten dar. Bei Qualität handelt es sich allerdings um ein vielschichtiges Konzept, das mit einer Reihe gravierender Probleme behaftet ist:

- Qualität ist nur indirekt messbar: Es gibt kein absolutes Maß für die Qualität eines Produktes, höchstens für einzelne Eigenschaften (etwa: Zeit- oder Ressourceneffizienz).

- Qualität ist relativ: Verschiedene Stakeholder haben unterschiedliche Qualitätsbegriffe und -anforderungen.
  - Manager und Auftraggeber fordern oft Kosteneffizienz, Flexibilität und Wartbarkeit.
  - Endanwender fordern hohe Performance und einfache Benutzbarkeit.
  - Projektleiter fordern Parallelisierbarkeit der Implementierung und gute Testbarkeit.
  - Betreiber fordern Administrierbarkeit und Sicherheit.
- Qualität der Architektur korreliert nicht notwendigerweise mit der Qualität des Endproduktes: Gute Architekturen können schlecht implementiert sein und dadurch die Qualität des Gesamtsystems mindern. Aus hervorragendem Quellcode kann jedoch nicht auf die Qualität der Architektur geschlossen werden. Insgesamt gilt daher: Architektur ist für die Qualität eines Systems notwendig, aber nicht hinreichend.
- Die Erfüllung sämtlicher funktionaler Anforderungen lässt kaum Aussagen über die Erreichung der Qualitätsanforderungen zu. Betrachten Sie das Beispiel eines einfachen Sortierverfahrens: Die (triviale) Anforderung, eine Menge von Variablen gemäß einem vorgegebenen Sortierkriterium aufsteigend zu ordnen, ist eine beliebte Programmieraufgabe für Einsteiger. Nun denken Sie an einige zusätzliche nichtfunktionale Anforderungen, etwa:
  - Sortierung großer Datenmengen (Terabyte), die nicht mehr zeitgleich im Hauptspeicher gehalten werden können
  - Sortierung robust gegenüber unterschiedlichen Sortierkriterien (Umlaute, akzentuierte Zeichen, Phoneme, Ähnlichkeitsmaße und anderes)
  - Sortierung für viele parallele Benutzer
  - Sortierung unterbrechbar für lang laufende Sortiervorgänge
  - Erweiterbarkeit um weitere Algorithmen, beispielsweise für ressourcenintensive Vergleichsoperationen
  - Entwickelbarkeit im räumlich verteilten Team

Diese Qualitätsanforderungen können von naiven Implementierungen nicht erfüllt werden – dazu bedarf es grundlegender architektonischer Maßnahmen!

Viele Publikationen[4] über Softwarearchitektur ignorieren das Thema der Qualitätsanforderungen völlig. Es scheint fast, als fielen Verständlichkeit, Wartbarkeit und Performance von Systemen als Nebenprodukte ab, wenn eifrige Entwickler nur in ausreichender Menge Design- und Architekturmuster anwenden[5]. Das Gegenteil ist der Fall:

Qualitätsanforderungen müssen explizte Entwurfsziele sein. Treffen Sie Entscheidungen zur Erreichung solcher Ziele bewusst und frühzeitig. Qualität muss explizit konstruiert werden – sie entsteht nicht von selbst!

[4] Asche auf mein Haupt – in der ersten Auflage dieses Buches habe ich die gleiche Unterlassung begangen. Sie halten zum Glück eine neuere Auflage in der Hand. ☺

[5] Leider helfen selbst Ansätze wie „Clean-Code“ nur begrenzt: Auch lupenreiner „clean-code“ kann elementare Qualitätsanforderungen wie Datensicherheit, Benutzbarkeit oder Betreibbarkeit verletzen!

Typische Qualitätsanforderungen

Die Qualität von Software-Systemen wird immer bezogen auf einzelne Eigenschaften oder Merkmale. Beispiele für solche Merkmale sind Effizienz (Performance), Verfügbarkeit, Änderbarkeit und Verständlichkeit.

Es gibt eine ganze Reihe unterschiedlicher Definitionen von Qualitätsmodellen und Qualitätsmerkmalen. Alle definieren einige zentrale Qualitätseigenschaften (beispielsweise Zuverlässigkeit, Effizienz, Wartbarkeit, Portabilität etc.) und verfeinern diese Eigenschaften durch eine Hierarchie weiterer Merkmale.

Egal, welches dieser Modelle Sie verwenden: Achten Sie innerhalb Ihrer Systeme oder Projekte auf eine einheitliche Terminologie für Qualitätsanforderungen.

Tabelle 3.4 zeigt Ihnen die Qualitätsmerkmale gemäß DIN/ISO 9126.[6] Diese Norm enthält die wesentlichen Begriffe rund um Software-Qualität.

Falls Sie in dieser Tabelle (vergeblich) nach dem Stichwort „Performance“ suchen – im DIN-normierten Sprachgebrauch heißt es „Effizienz“.

TABELLE 3.1 Qualitätsmerkmale nach DIN/ISO 9126

Funktionalität	Vorhandensein von Funktionen mit festgelegten Eigenschaften; diese Funktionen erfüllen die definierten Anforderungen
▪ Angemessenheit	Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen, z. B. die benötigte Genauigkeit von berechneten Werten
▪ Richtigkeit	Eignung der Funktionen für spezifizierte Aufgaben, z. B. aufgabenorientierte Zusammensetzung von Funktionen aus Teilfunktionen
▪ Interoperabilität	Fähigkeit, mit vorgegebenen Systemen zusammenzuwirken. Hierunter fällt auch die Einbettung in die Betriebsinfrastruktur.
▪ Ordnungsmäßigkeit	Erfüllung von anwendungsspezifischen Normen, Vereinbarungen, gesetzlichen Bestimmungen und ähnlichen Vorschriften
▪ Sicherheit	Fähigkeit, unberechtigten Zugriff, sowohl versehentlich als auch vorsätzlich, auf Programme und Daten zu verhindern
Zuverlässigkeit	Fähigkeit der Software, ihr Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum zu bewahren
▪ Reife	Geringe Versagenshäufigkeit durch Fehlzustände
▪ Fehlertoleranz	Fähigkeit, ein spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle zu bewahren
▪ Wiederherstellbarkeit	Fähigkeit, bei einem Versagen das Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen

[6] Es gibt einige Nachfolgestandards (ISO 25000 – 25030), die jedoch in die Praxis bislang wenig Einzug gehalten haben.

TABELLE 3.1 (*Fortsetzung*) Qualitätsmerkmale nach DIN/ISO 9126

Benutzbarkeit	Aufwand, der zur Benutzung erforderlich ist, und individuelle Beurteilung der Benutzung durch eine festgelegte oder vorausgesetzte Benutzergruppe; hierunter fällt auch der Bereich Softwareergonomie.
▪ Verständlichkeit	Aufwand für den Benutzer, das Konzept und die Anwendung zu verstehen
▪ Erlernbarkeit	Aufwand für den Benutzer, die Anwendung zu erlernen (z. B. Bedienung, Ein-, Ausgabe)
▪ Bedienbarkeit	Aufwand für den Benutzer, die Anwendung zu bedienen
Effizienz	Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen
▪ Zeitverhalten	Antwort- und Verarbeitungszeiten sowie Durchsatz bei der Funktionsausführung
▪ Verbrauchsverhalten	Anzahl und Dauer der benötigten Betriebsmittel für die Erfüllung der Funktionen
Änderbarkeit	Aufwand, der zur Durchführung vorgegebener Änderungen notwendig ist. Änderungen: Korrekturen, Verbesserungen oder Anpassungen an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen
▪ Analysierbarkeit	Aufwand, um Mängel oder Ursachen von Versagen zu diagnostizieren oder um änderungsbedürftige Teile zu bestimmen
▪ Modifizierbarkeit	Aufwand zur Ausführung von Verbesserungen, zur Fehlerbeseitigung oder Anpassung an Umgebungsänderungen
▪ Stabilität	Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen
▪ Prüfbarkeit	Aufwand, der zur Prüfung der geänderten Software notwendig ist
Übertragbarkeit	Eignung der Software, von einer Umgebung in eine andere übertragen zu werden. Umgebung kann organisatorische Umgebung, Hardware- oder Software-Umgebung einschließen.
▪ Anpassbarkeit	Software an verschiedene festgelegte Umgebungen anpassen
▪ Installierbarkeit	Aufwand, der zum Installieren der Software in einer festgelegten Umgebung notwendig ist
▪ Konformität	Grad, in dem die Software Normen oder Vereinbarungen zur Übertragbarkeit erfüllt
▪ Austauschbarkeit	Möglichkeit, diese Software anstelle einer spezifizierten anderen in der Umgebung jener Software zu verwenden, sowie der dafür notwendige Aufwand

Qualitätsanforderungen über Szenarien konkretisieren

Jetzt kennen Sie zwar die wichtigsten Qualitätsmerkmale, benötigen aber noch ein Mittel, um diese Merkmale praxisgerecht für Ihre Projekte zu konkretisieren und zu definieren. Hierzu eignen sich Szenarien (nach [Bass+03]). Szenarien beschreiben, was beim Eintreffen eines Stimulus auf ein System in bestimmten Situationen geschieht. Sie charakterisieren damit das Zusammenspiel von Stakeholdern mit dem System. Szenarien konkretisieren Qualitätsanforderungen.

Ich möchte Ihnen die Anwendung von Szenarien zur Konkretisierung von Qualitätsanforderungen anhand einiger Beispiele[7] verdeutlichen.

- *Anwendungsszenarien (beschreiben das Verhalten des Systems bei seiner Nutzung):*
  - Die Antwort auf eine Angebotsanfrage muss Endbenutzern im Regelbetrieb in weniger als 5 Sekunden dargestellt werden. Im Betrieb unter Hochlast (Jahresendgeschäft) darf eine Antwort bis zu 15 Sekunden dauern; in diesem Fall ist vorher ein entsprechender Hinweis darzustellen.
  - Ein Benutzer ohne Vorkenntnisse muss bei der erstmaligen Verwendung des Systems innerhalb von 15 Minuten in der Lage sein, die gewünschte Funktionalität zu lokalisieren und zu verwenden.
  - Bei Eingabe illegaler oder fehlerhafter Daten in die Eingabefelder muss das System entsprechende spezifische Hinweistexte ausgeben und anschließend im Normalbetrieb weiterarbeiten.
- *Änderungsszenarien (beschreiben gewünschte Reaktionen bei Änderungen am System oder der Umgebung):*
  - Die Programmierung neuer Versicherungstarife muss in weniger als 30 Personentagen möglich sein.
  - Die Unterstützung einer neuen Browser- oder Client-JDK-Version muss in weniger als 30 Personentagen programmiert und getestet werden können.
  - Bei Ausfall einer CPU muss das Ersatzsystem (Hot-Spare) im Normalbetrieb innerhalb von 15 Minuten online sein.
  - Die Anbindung eines neuen CRM-Systems[8] muss innerhalb von 60 Tagen möglich sein.

Bestandteile von Szenarien[9]

Nach diesen Beispielen können Sie sicherlich etwas Methodik vertragen: Szenarien bestehen aus folgenden wesentlichen Teilen (in Klammern die Terminologie aus [Bass+03]:

- Auslöser (*stimulus*): beschreibt eine spezifische Zusammenarbeit des (auslösenden) Stakeholders mit dem System. Beispiele: Ein Benutzer ruft eine Funktion auf, ein Entwickler programmiert eine Erweiterung, ein Administrator installiert oder konfiguriert das System.

---

[7] Zahlreiche Beispiele von Szenarien für Qualitätsanforderungen finden Sie unter *https://bitbucket.org/arc42/quality-requirements*

[8] CRM (Customer-Relationship-Management, deutsch *Verwaltung von Kundenbeziehungen*) hat das Ziel, Kundenbeziehungen in einem Unternehmen zu organisieren und somit die Kundenzufriedenheit und -bindung sowie die Profitabilität zu erhöhen (nach: [Wikipedia]).

[9] Ausführliche Beispiele von Szenarien für Qualitätsanforderungen finden Sie unter *https://bitbucket.org/arc42/quality-requirements*

- Quelle des Auslösers (*source*): beschreibt, woher der Auslöser stammt. Beispiele: intern oder extern, Benutzer, Betreiber, Angreifer, Manager.
- Umgebung (*environment*): beschreibt den Zustand des Systems zum Zeitpunkt des Auslösers. Befindet sich das System unter Normal- oder Höchstlast? Ist die Datenbank verfügbar oder nicht? Sind Benutzer online oder nicht? Hier sollten Sie alle Bedingungen beschreiben, die für das Verständnis des Szenarios wichtig sind.
- Systembestandteil (*artifact*): beschreibt, welcher Bestandteil des Systems vom Auslöser betroffen ist. Beispiele: Gesamtsystem, Datenbank, Webserver.
- Antwort (*response*): beschreibt, wie das System durch seine Architektur auf den Auslöser reagiert. Wird die vom Benutzer aufgerufene Funktion ausgeführt? Wie lange benötigt der Entwickler zur Programmierung? Welche Systemteile sind von Installation/Konfiguration betroffen?
- Antwortmetrik (*response measure*): beschreibt, wie die Antwort gemessen oder bewertet werden kann. Beispiele: Ausfallzeit in Stunden, Korrektheit Ja/Nein, Änderungszeit in Personentagen, Reaktionszeit in Sekunden.

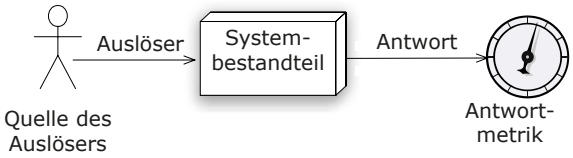


BILD 3.4
Schematische Darstellung von Szenarien (nach [Bass+03])

Qualitätsanforderungen explizit über Szenarien zu beschreiben, ist einfach und alleine deswegen nützlich!

Zusätzlich dienen diese Szenarien als wesentliche Grundlage der systematischen Architekturbewertung, die Sie in Kapitel 8 näher kennenlernen.

3.2.4 Relevante Stakeholder ermitteln

Stakeholder

Welche Rollen oder Personen (= Stakeholder) innerhalb oder außerhalb der Organisation haben ein Interesse am System? Beispiele sind Benutzer der Kernfunktionalität („Anwender"), Administratoren und Betreiber, Benutzer mit Sonderfunktionen (Genehmiger, Prüfer, Auditoren oder Ähnliche).

Welche Stakeholder stehen dem (neuen) System negativ gegenüber? Die jeweilige Einstellung der Stakeholder prägt die Art der Informationen, die sie Ihnen über das System geben werden.[10]

Insbesondere müssen Sie wissen, welche Erwartungshaltung die Stakeholder gegenüber dem Architektur- und Entwicklungsteam haben, welche Artefakte oder Dokumente Sie liefern und pflegen müssen.

[10] Softwaresysteme sind soziale Systeme. Als Architekt werden Sie daher häufig mit subjektiven Eindrücken statt Tatsachen konfrontiert, mit Meinungen statt Fakten, mit unrealistischen Erwartungen statt begründeter Anforderungen. Stellen Sie sich darauf ein!

Ich empfehle Ihnen, die Stakeholder Ihres Systems als Tabelle in die Architekturdokumentation aufzunehmen – wobei Sie folgende Informationen festhalten sollten:

TABELLE 3.2 Aufbau der Stakeholder-Tabelle

Rolle	Welche Aufgabe/Verantwortung tragen diese Stakeholder bezüglich des Systems?
Beschreibung	
Ziel/Intention	Welche konkreten Ziele verfolgen die Stakeholder bezüglich des Systems?
Erwartungshaltung	Welche konkreten Ergebnisse, Dokumente, Artefakte erwarten diese Stakeholder von Ihnen, der Architektur oder dem Entwicklungsteam?
Kontakt*	Sie benötigen für viele Ihrer Aufgaben Feedback der Stakeholder, daher sollten Sie deren Kontaktinformationen (E-Mail, Telefon) kennen.
Relevanz für Abnahme	Können diese Stakeholder die Abnahme oder den Einsatz des Systems erlauben, fördern oder behindern?

* Das mag formalistisch klingen, ist im Projektalltag jedoch praktisch zu wissen!

Beispiele für Stakeholder

Nachfolgend nenne ich Ihnen einige Beispiele für Stakeholder, die in meinen eigenen Projekten relevant waren (nicht alle auf einmal ...):

Management, Auftraggeber, Projekt-Steuerkreis, sonstige Projekt-Gremien, PMO, Projektmanager, Produktmanager, Fachbereich, Unternehmens-/Enterprise-Architekt, Architektur-Abteilung, Methoden-Abteilung, QS-Abteilung, IT-Strategie, (interner) Softwarearchitekt, Software-Designer, Entwickler, Tester, Konfigurationsmanager, Build-Manager, Release-Manager, Wartungsteam, externe Dienstleister, Zulieferer, Hardware-Designer, Rollout-Manager, Infrastruktur-Planer, Sicherheitsbeauftragter, Behörde, Aufsichtsgremium, Auditor, Mitbewerber/Konkurrent, Endanwender, Fachpresse, Fachadministrator, Systemadministrator, Operator, Hotline, Betriebsrat, Lieferant von Standardsoftware, verbundene Projekte, Normierungsgremium, Gesetzgeber ...

3.2.5 Fachlichen und technischen Kontext ermitteln

Sie sollten immer eine präzise Vorstellung der (fachlichen wie technischen) Umgebung eines Systems besitzen (= Kontext), d. h., welche Nachbarsysteme oder Benutzergruppen mit dem System interagieren.

Klären Sie möglichst frühzeitig:

- Welche Schnittstellen gibt es zu externen Systemen? Hierzu zählen:
  - Schnittstellen anderer genutzter Systeme
  - Schnittstellen, die für andere Systeme bereitgestellt werden

- Sind diese externen Schnittstellen stabil und zuverlässig oder volatil? Das bedeutet: Sind die Fremdsysteme bezüglich ihrer Schnittstellen stabil?
- Wie fehlertolerant muss das neue System gegenüber den fremden Systemen sein?
- Handelt es sich um funktionale Schnittstellen oder Datenschnittstellen? Können Sie über diese Schnittstellen Funktionen oder Methoden des anderen Systems aufrufen, oder funktioniert die Schnittstelle nur über den Austausch von Daten?
- Kommuniziert das System über die Schnittstellen synchron oder asynchron?
- Wie performant sind die anderen Systeme? Wie beeinflusst das Verhalten der Schnittstellen die Performance des Gesamtsystems?
- Gibt es die Möglichkeit, die Schnittstelle auf Seiten des anderen Systems zu ändern? Liegt der Quellcode der Schnittstelle vor?

- Dokumentieren Sie, welche Schnittstellen Sie bis jetzt identifiziert haben.
- Wenn das System Schnittstellen zu Fremdsystemen besitzt, sollten Sie über einen funktionalen Prototypen die technische Machbarkeit „end-to-end“ verifizieren. Insbesondere kritische oder riskante Schnittstellen sollten Sie möglichst frühzeitig auf diese Weise überprüfen.

Kontext als Diagramm – Erläuterung als Tabelle

Ich rate Ihnen, den Kontext Ihres Systems grafisch zu dokumentieren – und dieses Diagramm dann mit einer Tabelle zu erläutern (Bild 3.5).

Tabellarisch erklären Sie dazu die genauen Bezeichnungen aller Schnittstellen, die Richtung (ob sie aus der Sicht Ihres Systems benötigte (*required*) oder angebotene (*provided*) Schnittstellen sind) und die fachliche Bedeutung in Stichworten.

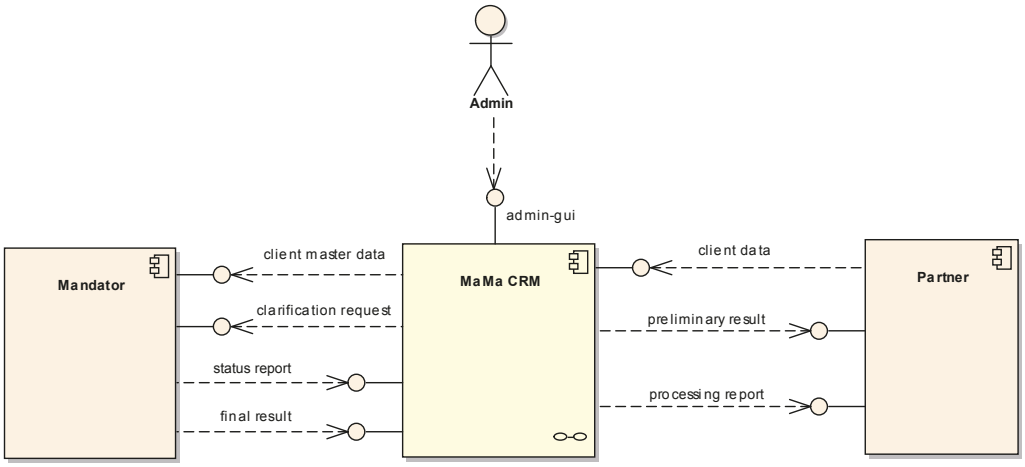


BILD 3.5 Beispiel einer Kontextabgrenzung

3.3 Einflussfaktoren und Randbedingungen ermitteln

Um als Softwarearchitekt anwendungs- und problembezogene Entwurfsentscheidungen zu treffen, müssen Sie die Faktoren kennen, die Ihre Architekturen beeinflussen oder einschränken werden (siehe dazu Bild 3.6). Diese Faktoren schränken Sie in Ihren Entscheidungen ein und besitzen daher einen prägenden Einfluss auf das Gesamtsystem.

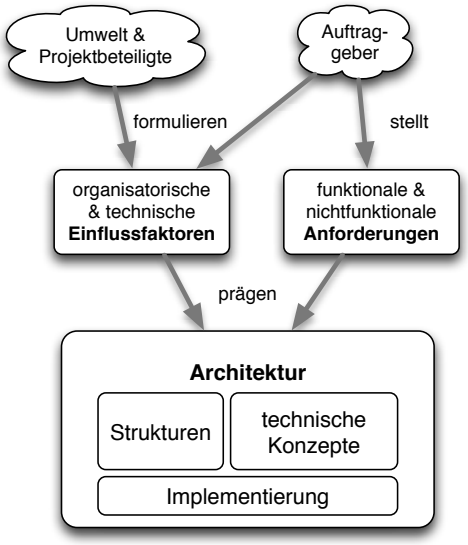


BILD 3.6
Anforderungen und Einflussfaktoren prägen den Architekturentwurf

Aus den Systemanforderungen sowie den Anforderungen der System-Umwelt ermitteln Sie architekturrelevante Einflussfaktoren. Im Folgenden stelle ich Ihnen einige typische Einflussfaktoren vor.

Randbedingungen und Einflussfaktoren fallen in folgende Kategorien:

- Organisatorische und politische Faktoren. Manche solcher Faktoren wirken auf ein bestimmtes System ein. Andere beeinflussen sämtliche Projekte innerhalb einer Organisation. [Rechtin2000] charakterisiert diese Faktoren als *facts of life*.
- Technische Faktoren. Sie prägen das technische Umfeld des Systems und seiner Entwicklung.

Meiner Erfahrung nach tendieren Softwarearchitekten dazu, die politischen und organisatorischen Faktoren zu unterschätzen und zu vernachlässigen. Das kann im Extremfall dazu führen, dass an sich lauffähige Systeme nicht zum Einsatz kommen.

Das hat mehrere Gründe: Erstens bestimmen die technischen Faktoren ein System viel offensichtlicher. Zweitens sind Softwarearchitekten in technischen Themen oftmals erfahrener als in organisatorischen oder gar politischen. Drittens herrscht in Software-Projekten manchmal ein technischer Zweckoptimismus, getreu dem Motto: Mit der neuen X-Technik und dem Y-Werkzeug haben wir unsere Probleme im Griff.

Wie [Rechtin2000] es treffend formuliert: *Es gibt keine rein technischen Probleme*. Sie werden schnell zu organisatorischen oder politischen Schwierigkeiten, und damit entgleiten sie auch der Kontrolle rein technisch orientierter Projektbeteiligter.

Die Einflussfaktoren und Randbindungen auf eine konkrete Softwarearchitektur sollten Sie in einer Tabelle (als Bestandteil der Architekturdokumentation) festhalten.

Organisatorische Faktoren

Diese Faktoren beziehen sich im weitesten Sinne auf die Umgebung, in der das System erstellt wird. Sie prägen das System indirekt. Hierzu zählen Aspekte wie Termin- und Budgetplanung, vorhandene Mitarbeiter, technische Ressourcen, Entwicklungsprozesse, Vorgaben bezüglich Werkzeuge und Ähnliches. Tabelle 3.3 gibt einen Überblick über typische organisatorische Faktoren.

Einige Tipps zur Identifikation von organisatorischen Faktoren und Risiken Ihrer Projekte:

- Denken Sie negativ. Murphys Regel besagt: Wenn etwas schiefgehen kann, wird es irgendwann schiefgehen. Und es geschieht immer im denkbar schlechtesten Moment.
- Politik und Organisation, nicht Technik, setzen die Grenzen dafür, was ein Projekt erreichen kann und darf. Bringen Sie die „politischen" Stakeholder auf Ihre Seite. Stellen Sie sicher, dass Sie die Intention der „Politiker" innerhalb Ihrer Organisation richtig verstanden haben.
- Eine fundamentale Gleichung lautet: „Geld = Politik" ([Rechtin2000]). Die Politik gibt Projekten die Kostenregeln vor. Diese Kostenregeln besitzen prägenden Einfluss auf Architekturen.
- Finden Sie heraus, welche Ziele die einzelnen Stakeholder mit dem System verfolgen. Oftmals werden in einem Projekt verschiedene Ziele verfolgt!
- Die besten technischen Lösungen sind nicht unbedingt die besten politischen Lösungen. Im Regelfall sind „politische" Stakeholder die Auftraggeber und Eigentümer von Projekten (nach [Rechtin2000]). Sie entscheiden meist nach anderen Kriterien als „technische" Stakeholder.
- Betrachten Sie systemübergreifende Prozesse oder Rollen innerhalb der Organisation. Beispiel: In vielen großen Unternehmen gibt es Vorgaben zur Datenmodellierung und zum Datenbankdesign (die von dedizierten Datenbankadministratoren geprüft und freigegeben werden müssen).
- Beziehen Sie andere Projekte innerhalb der Organisation in Ihre Betrachtung mit ein. Sie können aus deren Verlauf viel über Stärken und Schwächen der Organisation hinsichtlich Software-Erstellung lernen.
- Die Erfahrung der beteiligten Entscheider spielt meistens eine wichtige Rolle. So schränkt eine negative Erfahrung des Auftraggebers mit einer bestimmten Technologie Ihre Entwurfsalternativen möglicherweise ein.[11]
- Werfen Sie einen Blick auf das Risikomanagement Ihrer Projektleitung. Eventuell finden Sie dort neue und für die Architektur wichtige Faktoren.

[11] Edward Yourdon rät in seinem Buch „Death March: Surviving Mission Impossible Projects" dazu, bei einer nicht akzeptablen Fülle solcher Einschränkungen, das Projekt zu verlassen oder zu kündigen, um das eigene (berufliche) Überleben zu sichern.

TABELLE 3.3 Typische organisatorische Einflussfaktoren

Faktor	Erläuterung
Organisation und Struktur	
Organisationsstruktur beim Auftraggeber	▪ Droht Änderung von Verantwortlichkeiten? ▪ Änderung von Ansprechpartnern
Organisationsstruktur des Projektteams	▪ mit/ohne Unterauftragnehmer ▪ Entscheidungsbefugnis der Projektleiterin
Entscheidungsträger	▪ Erfahrung mit ähnlichen Projekten ▪ Risiko-/Innovationsfreude
Bestehende Partnerschaften oder Kooperationen	▪ Hat die Organisation bestehende Kooperationen mit bestimmten Softwareherstellern? ▪ Solche Partnerschaften geben oftmals (unabhängig von Systemanforderungen) Produktentscheidungen vor.
Eigenentwicklung oder externe Vergabe	▪ Selbst entwickeln oder an externe Dienstleister vergeben?
Entwicklung als Produkt oder zur eigenen Nutzung?	Bedingt andere Prozesse bei Anforderungsanalyse und Entscheidungen. Im Fall der Produktentwicklung: ▪ Neues Produkt für neuen Markt? ▪ Verbessertes Produkt für bestehenden Markt? ▪ Vermarktung eines bestehenden (eigenen) Systems ▪ Entwicklung ausschließlich zur eigenen Nutzung?
Ressourcen (Budget, Zeit, Personal)	
Festpreis oder Zeit/Aufwand?	Festpreisprojekt oder Abrechnung nach Zeit und Aufwand?
Zeitplan	Wie flexibel ist der Zeitplan? Gibt es einen festen Endtermin? Welche Stakeholder bestimmen den Endtermin?
Zeitplan und Funktionsumfang*	Was ist höher priorisiert: der Termin oder der Funktionsumfang?
Release-Plan	Zu welchen Zeitpunkten soll welcher Funktionsumfang in Releases/Versionen zur Verfügung stehen?
Projektbudget	Fest oder variabel? In welcher Höhe verfügbar?
Budget für technische Ressourcen	Kauf oder Miete von Entwicklungswerkzeugen (Hardware und Software)?
Team	Anzahl der Mitarbeiter und deren Qualifikation, Motivation und kontinuierliche Verfügbarkeit

TABELLE 3.3 (*Fortsetzung*) Typische organisatorische Einflussfaktoren

Faktor	Erläuterung
Organisatorische Standards	
Vorgehensmodell	Vorgaben bezüglich Vorgehensmodell? Hierzu gehören auch interne Standards zu Modellierung, Dokumentation und Implementierung.
Qualitätsstandards	Fällt die Organisation oder das System in den Geltungsbereich von Qualitätsnormen (wie ISO 9000)?
Entwicklungs-werkzeuge	Vorgaben bezüglich der Entwicklungswerkzeuge (etwa: CASE-Tool, Datenbank, Integrierte Entwicklungsumgebung, Kommunikations-software, Middleware, Transaktionsmonitor)
Konfigurations- und Versionsverwaltung	Vorgaben bezüglich Prozessen und Werkzeugen
Testwerkzeuge und -prozesse	Vorgaben bezüglich Prozessen und Werkzeugen
Abnahme- und Freigabeprozesse	▪ Datenmodellierung und Datenbankdesign ▪ Benutzeroberflächen ▪ Geschäftsprozesse (Workflow) ▪ Nutzung externer Systeme (etwa: schreibender Zugriff bei externen Datenbanken)
Service Level Agreements	Gibt es Vorgaben oder Standards hinsichtlich Verfügbarkeiten oder einzuhaltender Service-Levels?
Juristische Faktoren	
Haftungsfragen	▪ Hat die Nutzung oder der Betrieb des Systems mögliche rechtliche Konsequenzen? ▪ Kann das System Auwirkungen auf Menschenleben oder Gesundheit haben? ▪ Kann das System Auswirkungen auf Funktionsfähigkeit externer Systeme oder Unternehmen haben?
Datenschutz	Speichert oder bearbeitet das System „schutzwürdige“ Daten?
Nachweispflichten	Bestehen für bestimmte Systemaspekte juristische Nachweis-pflichten?
Internationale Rechtsfragen	▪ Wird das System international eingesetzt? ▪ Gelten in anderen Ländern eventuell andere juristische Rahmen-bedingungen für den Einsatz (Beispiel: Nutzung von Verschlüsse-lungsverfahren)?

* Tom DeMarco beschreibt die Probleme mit Zeitplan und Funktionsumfang in seinem amüsanten und lehrreichen Roman über Projektmanagement („Der Termin“, 1998 bei Hanser erschienen).

Zu den organisatorischen Einflussfaktoren ein kleines Beispiel:

Beispiel: Die Informatik-Tochterfirma eines Konzerns erhielt den internen Auftrag, ein Internet-basiertes Informationssystem für Finanznachrichten und Geschäftsberichte zu entwickeln und zu vermarkten.

Das Entwicklungsteam schien anfangs frei von organisatorischen und technischen Einflüssen zu sein. Im Laufe der Entwicklung stellte sich heraus, dass die Firma mit Marktpartnern und Kunden projektübergreifende „Service Level Agreements" (SLAs) bezüglich der Verfügbarkeit von Software vertraglich vereinbart hatte. Als Konsequenz für die Architektur ergab sich die Notwendigkeit, eine (anfänglich nicht geplante) Überwachungskomponente zum verbindlichen Nachweis der Systemverfügbarkeit zu entwickeln. Dies machte eine kosten- und zeitintensive Änderung des gesamten Persistenzkonzeptes notwendig.

Der zuständigen Projektleiterin gelang es, mit den betroffenen Endkunden des Systems eine Änderung der Nachweispflicht und der SLAs zu vereinbaren und damit zumindest den Termindruck der notwendigen Änderungen zu entschärfen. ■

Technische Faktoren

Technische Faktoren mit Relevanz für die Softwarearchitektur betreffen einerseits die technische Infrastruktur, also die Ablaufumgebung des Systems. Andererseits umfassen sie auch technische Vorgaben für die Entwicklung, einzusetzende Fremdsoftware und vorhandene Systeme.

Einige Tipps bei der Suche nach technischen Faktoren:

- Analysieren Sie andere Projekte innerhalb Ihrer Organisation. Befragen Sie Architekten und Projektleiter solcher Projekte.
- Betrachten Sie andere Systeme innerhalb der Organisation. Wie sieht die technische Umgebung dieser Systeme aus? Wie werden diese Systeme betrieben?
- Betrachten Sie die vorhandene Infrastruktur hinsichtlich Hardware und Software.
- Das Qualitätsmanagement der Organisation kann Hinweise auf weitere Einflussfaktoren geben.
- Gibt es Methoden, Standards oder Vorlagen für Software-Projekte?
- Analog zum Tipp bei den organisatorischen Faktoren: Betrachten Sie systemübergreifende Abnahme- und Freigabeprozesse. Sie machen häufig Vorgaben zur Gestaltung von Datenmodellen, Datenbanken, Benutzeroberflächen, Geschäftsprozessen (Workflows), Sicherheit, Laufzeit- und Wartungsprozessen sowie der technischen Infrastruktur.

Tabelle 3.4 zeigt einige typische technische Einflussfaktoren.

TABELLE 3.4 Typische technische Einflussfaktoren

Faktor	Erläuterung
Hardware-Infrastruktur	Prozessoren, Speicher, Netzwerke, Firewalls und andere relevante Elemente der Hardware-Infrastruktur
Software-Infrastruktur	Betriebssysteme, Datenbanksysteme, Middleware, Kommunikationssysteme, Transaktionsmonitor, Webserver, Verzeichnisdienste
Systembetrieb	Batch- oder Online-Betrieb des Systems oder notwendiger externer Systeme?
Verfügbarkeit der Laufzeitumgebung	Rechenzentrum mit 7 x 24 h Betriebszeit? Gibt es Wartungs- oder Backupzeiten mit eingeschränkter Verfügbarkeit des Systems oder wichtiger Systemteile?
Grafische Oberfläche	Existieren Vorgaben hinsichtlich grafischer Oberfläche (*Style Guide*)?
Bibliotheken, Frameworks und Komponenten	Sollen bestimmte „Software-Fertigteile“ eingesetzt werden?
Programmiersprachen	Objektorientierte, strukturierte, deklarative oder Regelsprachen, kompilierte oder interpretierte Sprachen?
Referenzarchitekturen	Gibt es in der Organisation vergleichbare oder übertragbare Referenzprojekte?
Analyse- und Entwurfsmethoden	Objektorientierte oder strukturierte Methoden?
Datenstrukturen	Vorgaben für bestimmte Datenstrukturen, Schnittstellen zu bestehenden Datenbanken oder Dateien
Programmierschnittstellen	Schnittstellen zu bestehenden Programmen
Programmiervorgaben	Programmierkonventionen, fester Programmaufbau
Technische Kommunikation	Synchron oder asynchron, Protokolle

Das alles sollte die Systemanalyse geliefert haben

In einer idealen Situation hat die Systemanalyse, das Requirements Engineering oder Ihr Product-Owner bereits alle bis hier aufgeführten Informationen ermittelt und passend aufbereitet. Gute Systemanalytiker entwickeln ein fachliches Modell, stellen Anforderungen, Einflussfaktoren und Randbedingungen systematisch dar und beschreiben die geforderten Qualitäten des Systems in Form von Szenarien.

Nach meiner Erfahrung betreiben jedoch nur wenige Projekte eine so ausführliche und methodische Systemanalyse. In den übrigen Projekten müssen die Software- und Systemarchitekten Teile dieser Analyse nachholen, insbesondere die hier dargestellten Einflussfaktoren, Randbedingungen, Risiken und Qualitätsmerkmale ermitteln und beschreiben.

3.4 Entwerfen und kommunizieren

Bis hierhin haben Sie die Voraussetzungen geschaffen, um angemessen entwerfen (und implementieren) zu können. Fast der gesamte restliche Teil des Buches ist der Erklärung und Vertiefung dieser Aufgabe gewidmet (siehe Bild 3.7).

- Strukturen entwerfen: Hier geht es primär um den Aufbau des Systems aus Quellcode, die Struktur der Implementierung.
- Konzepte entwerfen: Hier geht es um die übergreifenden, meist technischen Belange.
- Sie sollten als verantwortungsbewusster Softwarearchitekt die Umsetzung (d. h. Implementierung und Test) des Systems im Detail begleiten. Dazu finden Sie einige Tipps am Ende dieses Abschnitts.
- Meistens arbeiten wir in Teams – dazu müssen Sie über Architektur, Entwürfe, Entscheidungen, Strukturen und Konzepte miteinander kommunizieren. Kapitel 4 gibt Ihnen hierzu einige Hinweise.
- Explizit zu bewerten, ob ein System und dessen Architektur die geltenden Anforderungen erfüllen kann, gehört ebenfalls zu Ihren Aufgaben. Kapitel 8 stellt einen systematischen Ansatz dazu vor.
- Last, but not least: Sie sollten einige generelle Regeln zur Lösung Ihrer Entwurfsprobleme verfügen – in Abschnitt 3.5 habe ich einige davon zusammengefasst.

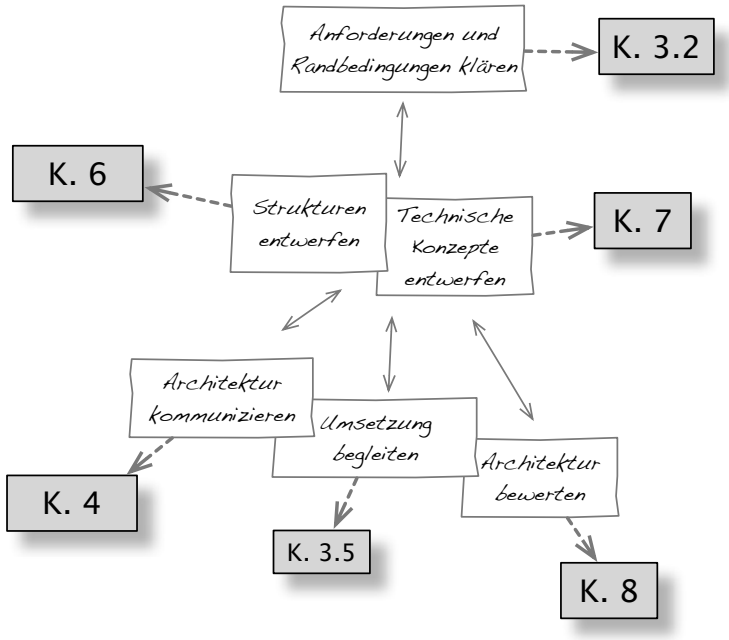


BILD 3.7 Hier finden Sie die weiteren Architektenaufgaben erklärt.

3.5 Umsetzung begleiten

Goldstücke suchen

Oftmals haben in meinen Projekten einzelne Entwickler hervorragende Verbesserungen gefunden und umgesetzt. Ihre Lösungen waren besser, einfacher, schneller oder günstiger als diejenigen des übrigen Teams oder der Architekturgruppe. Solche Lösungsansätze sollten Sie in jedem Fall in die Gesamtarchitektur übernehmen.[12] In anderen Industriezweigen nennt man dieses dauernde Suchen nach Verbesserungsmöglichkeiten „Kai-Zen" – ich nenne es gerne „Goldstücke suchen".

Bei der Entwicklung eines (Java-) Systems mit hohen Anforderungen an Flexibilität (genauer: Flexibilität von Datenim- und exporten) hatten wir diese Herausforderung durch (XML-basierte) Konfiguration gelöst. Hat gut funktioniert – allerdings war die Erstellung von Testfällen aufgrund der abstrusen Zahl spitzer Klammern eine wahre Strafarbeit für Entwickler und Tester ...

Knapp ein Dutzend Entwickler fand über lange Zeit täglich neue Gründe, keine Testfälle für diese Konfiguration zu schreiben. Die Menge unserer Testfälle blieb sträflich gering ...

Eines Tages hatte ein einzelner Entwickler plötzlich eine deutlich höhere Testrate. Auf Nachfragen stellt sich heraus, dass er unsere (bisher manuell erstellte) XML-Konfiguration sehr geschickt durch ein Sprachkonstrukt von Groovy* ergänzt – und damit den für eine komplexe Konfiguration nötigen Aufwand von bisher über 100 Zeilen XML-Konfiguration auf 10–20 Zeilen Groovy-Quellcode gesenkt hatte. Und das auch noch mit großartiger IDE-Unterstützung.

Ein echtes „Goldstück" für das gesamte Team, das diese Idee begeistert aufgenommen hat – und damit ein wesentliches Qualitätsziel des Systems deutlich verbessern konnte.

* Groovy ist eine dynamische Sprache auf der JVM, und hier ging es um die Groovy-MarkupBuilder (*http://groovy.codehaus.org*).

Missverständnisse aufdecken

Auf der anderen Seite könnten Entwickler auch Entwurfsentscheidungen oder Konzepte missverstehen oder schlecht umsetzen. In solchen Fällen sollten Sie als Architekt coachen und den Betroffenen die Entscheidungen oder Konzepte erläutern oder bei deren Implementierung unterstützen.

Quellcode lesen

In beiden Fällen müssen Sie in den Quellcode Ihrer Systeme schauen (= Code-Reviews), und die Ist-Implementierung mit dem gewünschten Soll vergleichen. Erst durch diese (Detail-) Arbeit können Sie wirklich erkennen, ob und wie Architekturentscheidungen letztlich umgesetzt werden.

[12] Eine Prüfung vorausgesetzt, sowie die Eignung für die jeweilige Problemstellung.

Peer-Reviews und Delegation

Gerade in größeren Teams kann eine einzelne Person nicht mehr den gesamten erstellten oder geänderten Quellcode überprüfen. Codeanalyse-Werkzeuge können helfen, aber in wesentlichen Fällen (kritische oder wichtige Stellen im System) sollten Menschen solche Reviews durchführen (nur sie können nach Gründen fragen und diese situativ einschätzen).

Oft genügt es auch, kleine Teile des Quellcodes zu inspizieren, um den Aufwand dieser Aufgabe in angemessenen Rahmen zu halten.

3.6 Lösungsstrategien entwickeln

Nach der Klärung von Anforderungen, Einflussfaktoren und Risiken, insbesondere der geforderten Qualitätsmerkmale und Projektrisiken, kennen Sie also die Schwierigkeiten, die beim Entwurf des Systems auf Sie zukommen. Nun gilt es, passende Lösungsstrategien und -alternativen zu entwickeln.

Diese Strategien entwickeln Sie parallel zum Entwurf von Sichten und Konzepten (zu denen Sie in den folgenden Kapiteln noch viel mehr lesen).

Ich möchte Ihnen in diesem Abschnitt Ratschläge und Anregungen zu einigen häufig vorkommenden Problembereichen von Softwaresystemen geben:

- Organisatorische Probleme: Mangel an Zeit, Budget oder Erfahrung
- Performance
- Flexibilität und Erweiterbarkeit

Sollten einige Ihrer spezifischen Probleme in eher technischen Bereichen angesiedelt sein, kann Ihnen sicherlich der Katalog typischer Konzepte (Kapitel 7) weiterhelfen.

3.6.1 Strategien gegen organisatorische Risiken

Meiner Erfahrung nach leiden die meisten Software-Projekte unter mindestens einem der folgenden (organisatorischen) Risiken:

- *Zu wenig Zeit:* Die Zeit bis zum Endtermin ist zu knapp bemessen, um die Anforderungen mit dem zugehörigen Projektteam zu erfüllen.
- *Zu wenig Budget:* Es mangelt dem Projekt an Geld. Aus Projektsicht notwendige Investitionen in Hardware, Software oder Wissen (in Form von Schulungen oder weiteren Mitarbeitern) unterbleiben.
- *Zu wenig Wissen und Erfahrung:* Es mangelt dem Projektteam an Wissen und Erfahrung in einigen Bereichen der Entwicklung. Beispielsweise wird ein neues, dem Team unbekanntes Entwicklungswerkzeug eingesetzt.

In seinem Buch „Death March“ ([Yourdon99]) gibt Edward Yourdon einige Ratschläge, wie Sie in schwierigen Projekten Ihr eigenes fachliches Überleben sichern. Falls die Situation Ihrer Ansicht nach völlig aussichtslos erscheint, sollten Sie dort nach Rettungsvorschlägen suchen.

Für die in der Praxis häufig vorkommenden „normal kritischen“ Situationen gebe ich Ihnen bezüglich der Softwarearchitektur folgende Ratschläge:

- Arbeiten Sie bei organisatorischen Problemen eng mit der Projektleitung oder dem Projektmanagement zusammen.
- Sie als Softwarearchitekt können wertvolle Hinweise zum Risikomanagement des Projektes liefern. Auf diese Weise können manche der Einflussfaktoren verhandelt werden (was Ihnen wiederum mehr Spielraum beim Entwurf der Architektur lässt).
- Sie können gemeinsam mit der Projektleitung alternative Modelle der Auslieferung oder Fertigstellung erarbeiten. Beispiel: Am vereinbarten Endtermin wird lediglich ein Teil der gewünschten Funktionalität geliefert.
- Sie können gemeinsam mit der Projektleitung die Auswirkung kritischer Anforderungen mit dem Auftraggeber diskutieren (und hier ebenfalls eine Entspannung der Situation erreichen).
- Falls sich die organisatorischen Probleme direkt auf technische Aspekte des Systems oder einzelne Bestandteile auswirken: Nutzen Sie diese Sachverhalte für Verhandlungen mit Projektleitung, Auftraggebern und Kunden.
- Prüfen Sie mit dem Auftraggeber, ob eine Revision der *Make-or-buy*-Entscheidung die Situation entschärfen kann. Gegebenenfalls kann auch der Zukauf einzelner Komponenten oder Systembestandteile helfen.
- Überarbeiten Sie mit dem Auftraggeber die Anforderungen an das System. Verhandeln Sie über andere Prioritäten der Anforderungen, um die Einschränkungen durch einige Einflussfaktoren zu lockern. Besonders kritisch und aufwändig sind (oftmals überzogene) Anforderungen an Verfügbarkeit und Performance.

Kooperation mit Projektleitung und Risikomanagement

Bedenken Sie bei organisatorischen Problemen einige wichtige Erfahrungen von Software-Projekten:

- Niemand arbeitet unter hohem Druck besonders produktiv. Und keiner denkt unter Druck schneller. Im Gegenteil: Hoher Druck erhöht die Fehlerquote und die Bereitschaft zu „schmutzigen Tricks“.
- Einem verspäteten Projekt mehr Mitarbeiter zu geben, verzögert das Projekt zusätzlich ([Brooks95]).
- Weitere Probleme tauchen von allein auf. Bauen Sie daher in alle Schätzungen Sicherheitszuschläge ein. Unterschätzen Sie sich lieber.
- Wenn die Politik nicht mitspielt, wird das System niemals laufen.

3.6.2 Strategien für hohe Performance

In meinen Projekten habe ich gelernt, dass praktisch jeder Auftraggeber eine hohe Performance[13] fordert. In vielen Fällen wird diese Forderung sehr allgemein formuliert („kurze Antwortzeiten").

Beispiel: Für eine Call-Center-Software forderte der Auftraggeber eine maximale Antwortzeit von einer Sekunde für sämtliche Operationen. Diese Anforderung wurde sehr hoch priorisiert und als formales Abnahmekriterium festgeschrieben. Gleichzeitig schrieb der Auftraggeber vor, sämtliche Daten des Systems mittels bereits vorhandener Mainframe-Systeme zu bearbeiten. Sekundäre oder redundante Datenspeicher zur Performancesteigerung waren nicht erlaubt.

Durch einen funktionalen Prototypen konnte der Architekt nachweisen, dass viele der Zugriffe auf diese Mainframe-Systeme bereits deutlich mehr als eine Sekunde benötigten.

Die Anforderung bezüglich der Antwortzeiten wurde daraufhin neu verhandelt. Der Auftraggeber verringerte die Priorität und formulierte sie wie folgt um:

„Die Antwortzeiten des Systems dürfen maximal eine Sekunde plus die kumulierten Antwortzeiten der beteiligten Mainframe-Systeme betragen."

Einige grundlegende Hinweise zur Bewältigung hoher Performanceanforderungen:

- Benutzen Sie einen Profiler, und führen Sie Lasttests durch. Lastkurven helfen in vielen Fällen, die Performance-Engpässe zu finden.
- Lösen Sie das Problem durch zusätzliche Hardware. Hardware ist (manchmal) preiswerter als Gehirnschmalz. Zusätzliche Prozessoren, mehr Speicher, schnellere Netzwerke oder Server können in manchen Fällen für ausreichende Performance sorgen. Einige Probleme bleiben mit diesem Ansatz jedoch erhalten:
  - Er erfordert teilweise erhebliche (Anfangs-)Investitionen.*
  - Sie müssen im Vorfeld den „Beweis" erbringen, dass die zusätzliche Hardware die Performance-Anforderungen erfüllt.
- Versuchen Sie, mit Ihren Auftraggebern die gewünschte Performance des Systems neu zu verhandeln.
- Verringern Sie die Kommunikation der Systemkomponenten. Fügen Sie Platzhalter oder Proxy-Klassen ein, um Kommunikation und Datentransfer zu sparen (Details finden Sie in Kapitel 5).
- Verringern Sie die Flexibilität des Systems. Verzichten Sie auf zusätzliche Abstraktionsschichten.

[13] Ich bleibe hier beim Begriff „Performance", obwohl die DIN/ISO Norm 9126 von „Effizienz" spricht.

- Verzichten Sie auf Verteilung. Verlagern Sie performancekritische Teile der Software zusammen auf einen Rechner oder Knoten, statt sie auf unterschiedliche Knoten zu verteilen.
- Führen Sie Redundanzen ein: Sie können Kopien kritischer Daten im Speicher halten, statt sie aus einer Datenbank zu lesen.
- Seien Sie pragmatisch: Wenn Performance wirklich wichtig ist, können Sie den Abstraktionsgrad der Programmiersprache verringern: Setzen Sie betriebssystemnahe Sprachen oder Assembler statt objektorientierter Sprachen ein.

* Oder Sie denken über „Cloud-Computing“ nach, bei dem Sie kaum eigene Hardware benötigen, sondern Rechenleistung über Internet mieten.

Beispiel: Eine Bank entwickelte ein System für Online-Überweisungen im Internet. Die Zielplattform waren „Personal Digital Assistants“ (PDA) in Verbindung mit Mobiltelefonen. Als Programmiersprache sollte Java eingesetzt werden. Die Sicherheit der Datenübertragung sollte durch den „Secure Socket Layer“ (SSL) erreicht werden. Die Performance war kritisch für die Akzeptanz und wurde daher hoch priorisiert.

Ein erster Prototyp, vollständig in Java implementiert, benötigte für den (komplexen) Anmeldevorgang (mit SSL-Handshake) am zentralen Server mehr als drei Minuten – eine nicht akzeptable Zeit.

Zusätzliche oder andere Hardware war in diesem Fall nicht möglich. Daher wurde die Architektur des Systems vereinfacht: Abstraktionsschichten innerhalb der Java-Anwendung entfielen. Zusätzlich entschloss sich der verantwortliche Softwarearchitekt, für die SSL-Implementierung Assembler statt Java einzusetzen. Mit Erfolg – die Anmeldung dauert mittlerweile nur noch etwa 4 Sekunden.

Der Nachteil: Das System hat dadurch massiv an Wartbarkeit und Verständlichkeit verloren. Die erwarteten Vorteile von Java hinsichtlich Portierbarkeit konnten nur eingeschränkt genutzt werden.

3.6.3 Strategien für Anpassbarkeit und Flexibilität

Flexibilität eines Systems umfasst Analysierbarkeit (änderungsbedürftige Teile identifizieren), Modifizierbarkeit (Anpassung) und Stabilität (keine unerwarteten Auswirkungen nach Änderungen). In praktisch allen Fällen kollidieren Anpassbarkeit und Flexibilität mit anderen Eigenschaften. Durch mehr Flexibilität:

- steigt Komplexität (und sinkt insofern die Verständlichkeit), weil Konfigurationsparameter und Indirektionen häufig über zusätzlichen Quellcode erkauft werden;
- sinkt Robustheit, weil sich beispielsweise unverträgliche Konfigurationseinstellungen nicht mehr zur Compile- oder Installationszeit überprüfen lassen;
- sinkt Performance, weil manche Entscheidungen erst zur Laufzeit getroffen werden können.

Beispiel: In einem unserer Projekte stand aufgrund der hohen erwarteten Lebensdauer des Systems (mehr als 15 Jahre!) die Flexibilität der Anwendung im Vordergrund aller Anforderungen. Sämtliche Details der eingesetzten Hard- und Softwareplattform sollten durch Architekturmittel gekapselt werden, CORBA war „politisch" verpönt. Direkte Methodenaufrufe zwischen verschiedenen Komponenten wurden vom Auftraggeber untersagt und ausschließlich über das Entwurfsmuster „Command" erlaubt. Die Softwarearchitekten konnten diese Anforderungen durch die Einführung mehrerer Abstraktionsschichten erreichen, die jeweils technische Details der darunter liegenden Schichten kapselten.

Gleichzeitig forderte der Auftraggeber allerdings eine sehr hohe Performance und kurze Antwortzeiten. Diese Forderung kollidiert mit der Schichtenbildung, die für die gewünschte Flexibilität notwendig war.

In diesem Fall zeigte ein ausführlicher technischer Prototyp die Unvereinbarkeit beider Anforderungen. Nach langen Verhandlungen willigte der Kunde ein, die (aus meiner Sicht übertriebenen) Forderungen hinsichtlich der Flexibilität zu lockern.

Wenn Flexibilität zu den Risiken (oder wichtigen Anforderungen) Ihres Systems gehört, dann können Ihnen folgende Ratschläge helfen:

Finden Sie gemeinsam mit Auftraggeber und Anwendern des Systems heraus, in welcher Hinsicht das System flexibel sein muss:

- Funktionalität (Hinzufügen neuer oder Modifizieren bestehender Funktionen)
- Datenstrukturen oder Datenmodell
- Eingesetzte Fremdsoftware (Datenbanksystem, GUI-Bibliothek, Middleware oder andere)
- Schnittstellen zu anderen Systemen (neue Schnittstellen, Modifikation bestehender Schnittstellen)
- Benutzerschnittstelle (veränderte Gestaltung oder Inhalte)
- Ziel- oder Betriebsplattform (Portabilität auf andere Betriebssysteme, Datenbanken oder Kommunikationssysteme)

Diese Informationen können den Bereich der notwendigen Flexibilität einschränken und Ihnen beim Entwurf des Systems mehr Freiheiten verschaffen. Sie können auf Basis dieser Informationen „Was-wäre-wenn"-Szenarien entwickeln, um verschiedene Architekturalternativen auf ihre Eignung zu prüfen.

- Sichern Sie die häufig geänderten Teile Ihres Systems durch automatische Tests (Unit- und Akzeptanztests) ab.
- Betreiben Sie möglichst ausgiebig „Information Hiding“.
- Verbergen Sie die internen Details einer Komponente vor anderen Komponenten.
- Führen Sie interne Abstraktionsschichten ein.
- Verringern Sie Abhängigkeiten zwischen Komponenten. Dabei kann Ihnen Kapitel 6 über Entwurfsprinzipien helfen.
- Halten Sie die Änderungen möglichst lokal: Die notwendigen oder erwarteten Änderungen sollten auf möglichst wenige Bausteine des Gesamtsystems beschränkt bleiben.
- Reduzieren Sie die Kopplung von Systembestandteilen untereinander:
  - Lassen Sie Bausteine immer über Schnittstellen miteinander kommunizieren. Verwenden Sie niemals Implementierungsdetails der benutzten Bausteine (das gewährleistet die Austauschbarkeit von Bausteinen).
  - Verwenden Sie Adapter, Fassaden oder Proxies (siehe Kapitel 6 und [GoF]), um Bausteine loser zu koppeln.
- Sorgen Sie für verständlichen Quellcode – der unterstützt auch den Umbau im Großen. In Robert Martins empfehlenswertem Buch *Clean Code* ([Martin08]) finden Sie mehr dazu.

Falls Sie auch Flexibilität zur Laufzeit Ihrer Systeme benötigen, können Ihnen die folgenden Vorschläge helfen. Ihr Tenor lautet: Entscheiden Sie möglichst spät.

- Verwenden Sie Konfigurationsparameter, um Installations- oder Laufzeitparameter möglichst lange flexibel zu halten.
- Lagern Sie Geschäftsregeln oder -prozesse in externe Repräsentationen aus (Stichwort: Process- oder Rule-Engines). Techniken wie BPEL, BPMN oder deklarative Regel-Engines unterstützen Sie dabei.
- Für objektorientierte Systeme:
  Verwenden Sie Polymorphismus*, um die Identifikation konkreter Typen erst zur Laufzeit vornehmen zu lassen (leider funktioniert das nur mit objektorientierten Programmiersprachen).

* Falls Sie diese Bezeichnung nicht mögen – Sie können alternativ „Runtime-Type-Identification“ sagen.

3.6.4 Strategien für hohe Verfügbarkeit

Sie können die Verfügbarkeit Ihrer Systeme durch folgende grundsätzliche Maßnahmen verbessern: Fehlererkennung (*fault detection*), Fehlerbehebung (*fault recovery*) und Fehlerverhütung (*fault prevention*). Zu allen dreien gebe ich Ihnen einige (kurze) Ratschläge, die Sie für Ihre konkreten Anforderungen spezialisieren müssen:

- Zur Fehlererkennung:
  Prüfen Sie frühzeitig auf Situationen, die zu Ausfällen oder Beeinträchtigungen führen können. Spendieren Sie beispielsweise Ihren Systemen eine robuste Ausnahmebehandlung, und/oder realisieren Sie Mechanismen, die periodisch (über *ping*- oder *echo*-Mechanismen) oder kontinuierlich (*heartbeat*) die Verfügbarkeit Ihrer Systeme prüfen.
- Zur Fehlerbehebung:
  - Lassen Sie mehrere redundante Bausteine über die Richtigkeit von Ergebnissen abstimmen (*voting*). Dies ist ein beliebtes, aber aufwändiges Mittel aus der Hochsicherheitstechnik.
  - Halten Sie Systembestandteile mehrfach redundant verfügbar (*hot spare*), und ersetzen Sie im Fehlerfall den defekten Systembestandteil durch ein solches Ersatzteil. Das gilt für Hardware wie auch für Software. Beispiel hierfür ist die Auslegung großer Rechenzentren, in denen die komplette Hard- und Software mehrfach und voneinander räumlich getrennt vorgehalten wird.*
- Zur Fehlerverhütung:
  - Verwenden Sie Transaktionen, indem Sie logisch zusammengehörige Operationen klammern. Falls innerhalb einer solchen Transaktion ein Fehler auftritt, setzen Sie das System durch ein Rollback auf den Zustand vor der Transaktion zurück.
  - Speichern Sie periodisch den gesamten Systemzustand in so genannten *Checkpoints*. Im Fehlerfall können Sie auf den letzten Checkpoint zurücksetzen.

* Für diese Strategie finden Sie auch den Begriff des Failover Cluster, wobei zwischen Hot-Standby- (Ersatzteil übernimmt ausschließlich im Fehlerfall) und Active-Active- (alle Teile laufen ständig mit, und im Fehlerfall wird der defekte Teil ausgeschaltet) Clustern differenziert wird. Eine gute Erläuterung dazu finden Sie in [Wikipedia].

Weiter mit Sichten und Dokumentation!

Schon mal als Vorwarnung: Im nächsten Kapitel stelle ich Ihnen die wichtigsten Sichten auf Softwarearchitekturen vor. Die müssen Sie kennen, bevor Sie mit dem Entwurf von Strukturen loslegen (dessen Prinzipien Sie in Kapitel 6 finden).

3.7 Wartung und Änderung von Software

Die Änderung bestehender Systeme ist wahrscheinlich die Standardaufgabe der meisten Softwareentwickler und -architekten. Nur in wenigen Fällen haben wir die Gelegenheit, auf der sogenannten grünen Wiese ein System völlig neu zu konstruieren, ohne Altlasten und ohne die gravierenden Einschränkungen bereits getroffener Architekturentscheidungen.

Grundsätzlich gelten alle Hinweise bezüglich des Vorgehens sowohl für die Neuentwicklung als auch für die Änderung bestehender Systeme. Allerdings erlebe ich in der Praxis immer wieder folgende Situation:

- Das bestehende System befindet sich seit mehreren Jahren im produktiven Einsatz.
- Das Entwicklungs- oder Wartungsteam bezeichnet den Quellcode als mittelmäßig bis schlecht – auf Basis des heutigen Wissens würden sie viele Dinge anders programmieren.
- Die Dokumentation ist lückenhaft, veraltet oder gar nicht vorhanden.
- Die meiste Zeit benötigt das Team zur Umgehung bekannter Schwächen oder das Kaschieren von Sünden der Vergangenheit.
- Erweiterungen oder Fehlerbehebung wird mit der Zeit immer schwerer.

Kurz gesagt – ein frustrierender Zustand mit wenig Aussicht auf Besserung, denn Budget für „gründlich aufräumen“ ist selten vorhanden. *Wegwerfen-und-neu-bauen* ist auch nur selten eine gangbare Option.

Die Frage lautet also: Wie können wir solch ein bestehendes System systematisch verbessern, ohne die Weiterentwicklung zu behindern und ohne erhebliche Mehraufwände zu produzieren?

Die Antwort gibt ausreichend Stoff für ein eigenes Buch – ich versuche eine Kurzfassung:

1. Sammeln Sie die Schwachstellen, Probleme und technischen Schulden des Systems in Form einer Tabelle oder Liste. Qualitative Bewertung (ATAM, siehe Kapitel 8), Codemetriken, detaillierte Analyse bekannter Fehler und Wartungsaufwände, Teamorganisation, Vorgehensweise – beleuchten und bewerten Sie alle technischen und organisatorischen Aspekte, derer Sie habhaft werden können.
2. Schätzen Sie nun zu jedem Eintrag folgende Aspekte:
   a) Kosten, die dieses Problem pro Monat oder pro Release verursacht
   b) Kosten, die eine Behebung dieses Problems ungefähr verursachen würde
   c) Risiken, die eine Behebung mit sich bringt.
3. Nun können Sie die bekannten Schwachstellen eindeutig priorisieren.

Jetzt beginnt der schwierigste Teil, nämlich, Ihr Management von der Notwendigkeit von Verbesserungen, Sanierungsmaßnahmen oder Refactorings zu überzeugen. Aber Sie sind ja dank obiger Vorgehensweise bestens vorbereitet: Ihre Manager verstehen „Geldeinheiten“ am besten – und Sie können jetzt (Punkt 2 sei Dank) gut argumentieren.

Gehen Sie in kleinen Schritten vor. Zeigen Sie an (kleinen) Beispielen, dass sich Ihre Verbesserungsmaßnahmen auch wirklich auszahlen.

Sie können ein Management niemals durch rein technische Argumentation („unser Code verletzt das Open-Closed-Prinzip“) von notwendigen Verbesserungsmaßnahmen überzeugen.

Sie müssen in jedem Fall technische Schulden auf Geld abbilden. Dazu müssen Sie sich mit Kosten, Aufwänden, Budgets oder ähnlich langweiligen Dingen beschäftigen. Aber das dient einem guten Zweck: Ihr System langfristig und nachhaltig zu verbessern.

3.8 Weiterführende Literatur

Die hier vorgestellten Ansätze, Einflussfaktoren zu gruppieren, zu priorisieren und mit spezifischen Strategien zu adressieren, werden in einigen anderen Disziplinen seit Langem erfolgreich angewandt, etwa im Risikomanagement von Projekten oder dem *System Engineering* (siehe [Rechtin2000]). [Bass+03] diskutieren Qualitätsmerkmale und Möglichkeiten zur Umsetzung als sogenannte *Architectural Tactics*.

[Hofmeister2000] beschreibt eine „globale Analyse“, die auch auf die Darstellung der Einflussfaktoren sowie die Entwicklung von Strategien abzielt.

[Martin08] zeigt den durchweg positiven Effekt von *Clean Code* auf: Verständlicher Quellcode hilft auch beim Verständnis von Architekturen.

[Rechtin2000] ist eine Fundgrube für Heuristiken und Vorgehensweisen beim Entwurf von Architekturen.

4 Kommunikation und Dokumentation von Architekturen

I am more and more convinced every day that communication is, in fact, what makes or breaks software projects.

Gojko Adzic (in: Bridging the Communication Gap)

Fragen, die dieses Kapitel beantwortet:

- Was müssen Architekten kommunizieren und dokumentieren?
- Was ist eine Sicht auf eine Architektur?
- Welche nützlichen Architektursichten gibt es?
- Wie entwerfen Sie die Sichten?
- Was gehört neben Sichten noch zur Architekturdokumentation?
- Welche Anforderungen muss eine Architekturdokumentation erfüllen?
- Was sollten Sie bei der Dokumentation von Architekturen beachten?

Softwarearchitekten müssen im Rahmen ihrer Kommunikations- und Dokumentationsaufgabe den übrigen Stakeholdern des Systems die Architektur erklären. Dazu gehören die Strukturen, Entscheidungen und Konzepte, deren Begründungen sowie Vor- und Nachteile. Manchmal müssen Softwarearchitekten all dies gegen Widerstände verteidigen oder die Architektur regelrecht vermarkten. Dazu bedarf es einer ausgewogenen Mischung aus Fingerspitzengefühl und Durchsetzungsvermögen, gepaart mit technischer und fachlicher Kompetenz. Aber Sie wussten ja bereits, dass es Softwarearchitekten manchmal schwer haben, oder?

In diesem Kapitel möchte ich Ihnen die die Anforderungen an praxisgerechte technische[1] Dokumentation vorstellen. Im zweiten Schritt präsentiere ich Ihnen einfache Grundregeln, mit deren Hilfe Sie bereits eine ganze Reihe dieser Anforderungen erfüllen können. Im Hauptteil des Kapitels lernen Sie das Konzept von Sichten und Perspektiven kennen. Danach wissen Sie, wie Sie angemessene[2] Architekturdokumentation für Ihre Systeme aufbauen können.

[1] Benutzer- oder Anwenderdokumentation gehorcht anderen Regeln, auf die ich hier nicht eingehe.
[2] Sparsam und nutzergerecht

4.1 Architekten müssen kommunizieren und dokumentieren

Architekten müssen kommunizieren

Softwarearchitekten treffen vielfältige Entwurfsentscheidungen, die von prägendem Einfluss auf die Arbeit anderer Projektbeteiligter sind. Um ein gemeinsames Verständnis der Architektur sicherzustellen, müssen Architekten die Entscheidungen (Strukturen, Konzepte, Begründungen, Modelle) *kommunizieren*. Bild 4.1 zeigt diese Aufgabe im Zusammenhang mit der Architekturentwicklung.

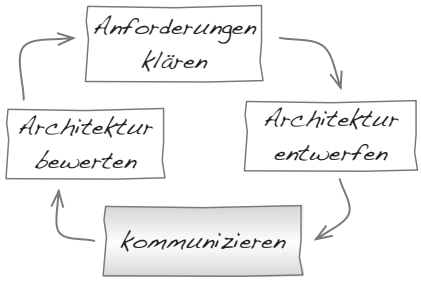


BILD 4.1
Kommunikationsaufgabe von Architekten

Nur durch Kommunizieren können Architekten ihre Entscheidungen motivieren, Entwürfe propagieren sowie die spezifische Architektur dem gesamten Team vermitteln!

Diese Kommunikation erfolgt in der Regel zunächst mündlich (durch gemeinsames Erarbeiten, Erklären, Vorstellen und Überzeugen) und danach schriftlich (durch Dokumente, Modelle oder andere Medien). Für die langfristige Verwendung über die Lebensdauer von IT-Systemen hinweg ist die schriftliche Kommunikation wesentlich – praktisch jedes System profitiert daher von einer methodisch und strukturell fundierten Dokumentation.

In der Praxis scheitern Projekte immer wieder, weil Softwarearchitekten ihrer Kommunikationsaufgabe nicht angemessen nachkommen. Stimmen Sie mit Ihren Projektbeteiligten den jeweiligen Kommunikations- und Dokumentationsbedarf ab:

- Fragen Sie Stakeholder im Vorfeld nach deren konkreten Bedürfnissen oder Vorstellungen von Dokumentation oder Erläuterungen.
- Holen Sie aktiv Rückmeldungen von Ihren Stakeholdern ein – das hilft Ihnen, Defizite in Ihrer Architekturkommunikation frühzeitig zu erkennen.
- Bleiben Sie offen für Verbesserungsvorschläge – aber zeigen Sie bei Bedarf auch Durchsetzungs- oder Durchhaltevermögen. Sie tragen die Verantwortung für technische Entscheidungen.
- Verwenden Sie Sichten zur getrennten Beschreibung der unterschiedlichen Strukturen.

- Trennen Sie diese Sichten von den übergreifenden technischen Konzepten (Kapitel 7 hält dazu viele Details für Sie bereit).
- Kommunizieren und dokumentieren Sie Top-down: Beginnen Sie mit *Vogelperspektiven* und fügen schrittweise Details hinzu.
- Benutzen Sie Vorlagen oder Templates für die Gliederung Ihrer Dokumentation (einen bewährten Vorschlag finden Sie in Abschnitt 4.9.1).

Darum sollten Sie Architekturen dokumentieren

Architektur verschafft Überblick

Der reine Quellcode kann in realen IT-Systemen aufgrund seines Umfangs sowie seines niedrigen Abstraktionsniveaus die Aufgabe der Dokumentation nicht vollständig übernehmen. Architekturen hingegen ermöglichen eine effektive (d. h. zielgerichtete) gesamthafte Kommunikation über ein IT-System – sie verschaffen Überblick.

Die Lebensdauer von IT-Systemen übersteigt in der Regel deutlich die ursprüngliche Erstellungszeit: Viele Systeme entstehen in Projekten von ungefähr einem Jahr Dauer und bleiben für fünf bis fünfzehn Jahre im Einsatz.[3] Verständliche, aktuelle und redundanzfreie Dokumentation ermöglicht es in solchen Situationen, auch über einen langen Zeitraum

- den Überblick zu wahren,
- auftretende Fehler und Probleme kurzfristig zu beseitigen,
- geänderte Anforderungen mit angemessenem Aufwand zu erfüllen und
- auf Änderungen im gesamten technischen Umfeld (Hardware, Betriebssysteme, Middleware, Fremdsysteme, Datenbanken etc.) zu reagieren.

Cirka 30–50 % der Wartungs- und Änderungskosten von Software werden benötigt, um bestehenden Code, Konzepte und Strukturen zu verstehen!*

Angemessene Dokumentation kann diesen Aufwand beträchtlich senken!

* Quelle: *http://blog.optimyth.com/2012/10/6-best-practices-to-save-in-software-maintenance*

In IT-Systemen signifikanter Größe korreliert der Aufwand für die oben genannten Aktivitäten direkt mit der Güte der Architekturdokumentation.

[3] In manchen Bereichen, wie etwa der Eisenbahntechnik, bleiben Softwaresysteme teilweise länger als 30 Jahre in Betrieb (bzw. die Hersteller müssen für diese Zeit die Wartung und Weiterentwicklung gewährleisten).

4.2 Effektive Architekturdokumentation

Für alle Arten von Architekturdokumentation gelten einige übergreifende Anforderungen und Regeln, die Sie bei der Erstellung solcher Dokumente berücksichtigen sollten. Ich stelle Ihnen zuerst die Anforderungen an Architekturdokumentation vor und leite daraus einige Grundlagen angemessener (= *guter*) Dokumentation ab.

4.2.1 Anforderungen an Architekturdokumentation

In den folgenden Abschnitten stelle ich Ihnen die wesentlichen Anforderungen an Architekturdokumentation vor.

- Die inhaltlichen Anforderungen an Dokumentation beschreiben, was die Dokumentation für Ihre Leserschaft leisten muss. Konkret stelle ich Ihnen einige typische Fragen vor, die Ihre Architekturdokumentation beantworten muss.
- Darüber hinaus sollte Dokumentation einige weitere Anforderungen erfüllen, die ich als „Qualitätsmerkmale von Dokumentation“ bezeichne.

Im Anschluss daran stelle ich Ihnen vor, wie Sie diese Anforderungen erfüllen können. Dazu lernen Sie eine praxistaugliche Struktur von Architekturdokumentation kennen.

Inhaltliche Anforderungen an Architekturdokumentation

Die Softwarearchitektur-Dokumentation (SWAD) muss den Stakeholdern eines IT-Systems eine Reihe wichtiger Auskünfte geben und ihre spezifischen Fragen zur Architektur und allen weiteren technischen Aspekten des Systems beantworten.

Folgende universelle Fragen muss die SWAD beantworten:

- Wie ist der Quellcode des System organisiert und strukturiert? Welche „größeren“ Implementierungseinheiten (Bausteine, Strukturelemente) gibt es, und welche Beziehungen gibt es dazwischen?
- Wie verhalten sich die Bausteine zur Laufzeit, und wie arbeiten sie zusammen?
- Wie fügt sich das System in seine Umgebung ein, insbesondere in seine technische Infrastruktur sowie die Projektorganisation?
- Welche grundlegenden Entscheidungen, Technologien oder Konzepte prägen die Lösung, deren Implementierung und Betrieb?

Die meisten Stakeholder haben daneben noch einige spezifische Fragen. Tabelle 4.1 zeigt einige wichtige im Überblick.

TABELLE 4.1 Typische Fragen an die Architekturdokumentation

Stakeholder	Fragen
Kunden und Auftraggeber	▪ Hat das Projektteam das Problem verstanden? ▪ Was ist der Kontext des Systems? Mit welchen Nachbarn arbeitet das System zusammen? ▪ Was sind die wesentlichen Entwurfsentscheidungen bzw. Lösungsstrategien? ▪ Passen die Strukturen und Konzepte der (geplanten) Lösung zum Problem? ▪ Ist diese Architektur für den geplanten Einsatz tragfähig hinsichtlich nichtfunktionaler Anforderungen wie Performance und Wartbarkeit?
Manager	▪ Was muss der Auftraggeber zuliefern, welche Teile müssen wir selbst entwickeln, welche Teile können wir zukaufen?
Fachexperten, Entwickler und Qualitätssicherer	▪ Welchen Teil des Systems entwerfe/realisiere/prüfe ich gerade, und wie ordnet sich dieser in das Gesamtsystem ein? ▪ Aus welchen Gründen wurden Entwurfsentscheidungen getroffen? ▪ An welchen Stellen muss das System flexibel und variabel sein?
Neue Mitarbeiter	▪ Was hat das Team bisher entworfen und entwickelt? ▪ Wie sieht das System aus der Vogelperspektive aus? Was ist die „Lösungsstrategie“ des Systems? ▪ Welche technischen Grundlagen (Implementierungstechnologien, Frameworks, Konzepte) sind relevant? ▪ Warum sieht der Code so aus? Welche Gründe haben dazu geführt? ▪ Wie sehen die technischen Konzepte (etwa: für Persistenz, Logging, Sicherheit o. Ä.) aus?
Administrator, Betreiber	▪ Wie müssen wir die Bausteine des Systems zur Laufzeit verteilen, damit alles korrekt funktioniert? ▪ Welche Regeln und Randbedingungen müssen wir bei der Installation des Systems, im laufenden Betrieb sowie beim Austausch von Systemteilen beachten? ▪ Wie muss die zugrunde liegende Hardware dimensioniert sein? ▪ Welche Abhängigkeiten von anderen Software-Systemen gibt es?

Sie entnehmen dieser (unvollständigen) Übersicht,[4] dass unterschiedliche Projektbeteiligte völlig unterschiedliche Fragestellungen hinsichtlich der Softwarearchitektur eines Systems haben.

Ihre Dokumentation sollte möglichst viele dieser Fragen beantworten können. Die weiter unten vorgestellten Architektursichten helfen dabei, die unterschiedlichen Informationsbedürfnisse verschiedener Lesergruppen angemessen zu erfüllen.

[4] Stellen Sie bitte in Ihren Projekten sicher, dass Sie sämtliche Fragen, die Ihre Stakeholder an die Architekturdokumentation haben, im Vorfeld kennen!

Qualitätsanforderungen an Architekturdokumentation

Alle Beteiligten stellen an die SWAD hohe Qualitätsansprüche.

- *Aktuell und korrekt:* Falls die Dokumentation Fehler enthält oder einen veralteten Stand widerspiegelt, wird sie niemand lesen wollen. Fehler in technischer Dokumentation sind nicht tolerierbar!
- *Zielgruppenadäquat:* Die Dokumentation muss auf die Bedürfnisse ihrer jeweiligen Leserschaft eingehen und deren jeweilige Fragen beantworten.
- *Verständlich:* Ihre Leser möchten „abgeholt" werden. Verwenden Sie geeignete Ausdrucksmittel, und bitten Sie Ihre Stakeholder, die Verständlichkeit zu bewerten, etwa in Schulnoten. Zur Verständlichkeit gehört auch Konsistenz, also die durchgängige und einheitliche Verwendung von Begriffen, Modellen und Strukturen zur Darstellung bestimmter Aspekte.
- *Wartbar:* Damit die Forderung nach Aktualität erfüllt werden kann, muss die SWAD leicht angepasst werden können. Dabei helfen Modularität und Kürze erheblich. Verständlichkeit steigt durch kontrollierte Redundanzen, die Wartbarkeit nimmt dadurch allerdings ab.
- *Kompakt:* Konzentrieren Sie sich auf die wesentlichen Fakten, die Ihre Leser kennen müssen. Arbeiten Sie aktiv und bewusst daran, Texte und Dokumente möglichst kurz und kompakt zu gestalten, Ihre Leser werden es Ihnen danken. Leider sind Kürze und Prägnanz in Dokumenten erheblich schwerer zu erreichen als übermäßige Länge. Kompakte Dokumentation ist in der Regel leichter wartbar.
- *Top-down*[5] *oder hierarchisch organisiert:* Sie sollten einen kurzen Überblick über das System bereitstellen sowie davon abgeleitete Verfeinerungen. In den folgenden Abschnitten finden Sie einige Vorschläge für geeignete Dokumentarten und Strukturen. Achten Sie immer darauf, trotz vieler Bäume den Wald noch erkennen zu können. Dieser Ratschlag bedeutet übrigens NICHT, dass Sie Entwurf und Entwicklung Top-down arbeiten sollen!
- *Kostengünstig:* Zumindest Ihre Manager werden lautstark danach rufen. Als Architekt sollten Sie auf diesen berechtigten Ruf hören. Dokumentieren Sie angemessen: so viel wie nötig, aber so wenig wie möglich.

Kurze, kompakte Dokumentation ist meistens besser als (übermäßig) ausführliche.

Verwenden Sie in Ihrer Dokumentation Abstraktionen, und reduzieren Sie systematisch Details. Verweisen Sie für Details auf Quellcode.

[5] Top-down ist lediglich die Richtung der Kommunikation – beim Konstruieren und Entwickeln können Sie ganz beliebig vorgehen.

4.2.2 Regeln für gute Architekturdokumentation

Nun kennen Sie die (Mindest-)Anforderungen an die Architekturdokumentation, benötigen aber noch geeignete Mittel und Wege, sie zu erfüllen. Leider ist es nicht damit getan, einfach „drauflos“ zu schreiben – Dokumentation benötigt Struktur! Ich habe genügend Projekte erlebt, in denen Hunderte, ja Tausende Seiten an (Detail-)Dokumentation produziert wurden, ohne dass die Verständlichkeit oder die Wartbarkeit des Gesamtsystems davon profitiert hätte. Häufig fehlte solchen Dokumentationen (die ihren Namen nicht verdienten) jegliche übergeordnete Struktur, sie blieben unverständlich und letztlich nutzlos.

Schaffen Sie Klarheit (um Missverständnisse zu vermeiden)

Sie müssen Syntax und Semantik einer Sprache kennen, um Sätze dieser Sprache zu verstehen. Andernfalls besteht das Risiko von Fehlinterpretationen und Missverständnissen.

Genauso verhält es sich mit Architekturbeschreibungen. Sie als Architekt und Ihre Leser, also die Projektbeteiligten, müssen von Syntax und Semantik der Architekturbeschreibung eine identische Vorstellung besitzen. Die in Abschnitt 4.9.1 beschriebene Strukturvorlage für Architekturbeschreibungen hilft Ihnen, Missverständnisse und Unklarheiten zu vermeiden. Sie gibt Ihnen eine Hilfestellung bezüglich Zweck, Form und Adressaten für die verschiedenen Sichten von Architekturbeschreibungen.

Ich habe in meiner Projektpraxis mehrfach Architekturbeschreibungen gesehen, die von Architekten, Auftraggebern, Projektleitern und Programmierern völlig unterschiedlich interpretiert wurden, zum Teil mit fatalen Folgen für die Projektziele. In Kapitel 2.1 haben Sie bereits ein (abschreckendes) Beispiel einer missverständlichen Architekturbeschreibung kennengelernt.

Einige Tipps für mehr Klarheit und weniger Missverständnisse:

- Schaffen Sie eine Gesamtstruktur Ihrer Dokumentation, und legen Sie diese Struktur in Form einer Navigationshilfe oder eines *Documentation Guide* offen (die Architektur-der-Architekturdokumentation).
- Kommunizieren Sie diese Struktur an alle Betroffenen, insbesondere die Leser und die Ersteller der Dokumentation.
- Stellen Sie sicher, dass alle Beteiligten die Syntax und Semantik der verwendeten Sprache oder Modellierungsmethoden kennen.
- Eine feste und in Ihrer Organisation etablierte Gliederung der Architekturdokumentation erleichtert deren Verständnis, sowohl für Leser als auch Autoren.

* Im Idealfall verfügt ein Projekt über eine Architektur der Gesamtdokumentation. Leider ist dieser Idealtypus in der Praxis nahezu ausgestorben.

Beherzigen Sie die Grundprinzipien von Dokumentation

Aufgrund ihrer zentralen Bedeutung fasse ich in diesem Abschnitt wesentliche Grundprinzipien für eine nützliche Dokumentation in Form einiger Anregungen zusammen.

Sie sollten diese Prinzipien bei der Erstellung von (Architektur-)Dokumenten beherzigen. Falls Sie als Reviewer solcher Dokumente tätig sind, können Sie die folgenden Ratschläge als Checkliste verwenden.

- Schreiben Sie aus der Sicht der Leser: Dokumente werden häufiger gelesen als geschrieben. Achten Sie darauf, dass Ihre Leser sie auch verstehen können. Sie sind als Autor Dienstleister am Leser. Verwenden Sie das Vokabular Ihrer jeweiligen Leserschaft. Erklären Sie zentrale und für das Verständnis wesentliche Begriffe in einem Glossar.
- Dokumentieren Sie das Warum: Beschreiben Sie die Gründe, die zu einem Entwurf geführt haben. Zu dieser Begründung gehört idealerweise auch die Beschreibung der verworfenen oder zurückgestellten Alternativen.
- Dokumentieren Sie Ihre Annahmen, Rahmenbedingungen und Voraussetzungen: Ihre Entscheidungen werden verständlicher, wenn Sie Annahmen und Voraussetzungen klar beschreiben. Bitte denken Sie daran: Für Sie als Autor sind möglicherweise ganz andere Dinge selbstverständlich als für Ihre Leser.
- Vermeiden Sie zu viele Wiederholungen (Don't repeat yourself, DRY-Prinzip), auch bekannt als Prinzip des Single-Point-of-Truth (SPOT-Prinzip). In der Praxis ist es manchmal nützlich, zur Verbesserung der Verständlichkeit kontrollierte Redundanz zuzulassen, um beispielsweise die Anzahl der Querverweise zu verringern. Beachten Sie, dass Redundanzen den Aufwand für die Pflege von Dokumenten stark erhöhen.
- Vermeiden Sie Überraschungen (Principle of least astonishment, POLA-Prinzip). Lösungen sollen möglichst geringe Verwunderung auslösen. In Dokumentationen sollten Sie alle möglichen Ursachen für Verwunderung zumindest ansprechen und erklären.
- Erklären Sie Ihre Notation (oder verwenden Sie Standards, die erklärt sind): Solange Sie die Bedeutung von Symbolen offen lassen, bleibt Spielraum für Interpretation, Mehrdeutigkeit und Missverständnis. Sie sollten es Ihren Lesern möglichst einfach machen, die Bedeutung von Symbolen und Diagrammen so zu verstehen, wie Sie es gemeint haben.
- Geben Sie jedem Diagramm eine Legende – eine Spezialisierung von „Erklären Sie Ihre Notation".
- Erklären Sie jedes Element in Diagrammen. Erklären Sie sämtliche Elemente (Bausteine und Beziehungen) innerhalb von (Architektur-)Diagrammen. (Siehe auch „Vermeiden Sie Mehrdeutigkeit".)

- (Möglichst) 7 +/– 2 Elemente pro Diagramm.* Eine Regel der kognitiven Psychologie besagt, dass der Mensch im Kurzzeitgedächtnis nur zwischen 5 und 9 Elemente speichern kann. Als Ausnahme gilt die „Architekturtapete“ in Abschnitt 4.3.
- Verwenden Sie standardisierte Strukturen. Etablieren Sie eine standardisierte Gliederung für die Architekturdokumentation. Stellen Sie sicher, dass Ihre Leser diese Struktur kennen. Das hilft Ihnen und Ihren Lesern ungemein, Informationen schnell zu finden.
- Kennzeichnen Sie offene Punkte. Falls Informationen noch nicht verfügbar sind, sollten Sie, anstatt Abschnitte oder Überschriften wegzulassen, solche Teile mit kurzen Kommentaren (etwa: <TBD> für to-be-done) kennzeichnen. (Siehe auch „Keine degenerierten Kapitel“.)
- Prüfen Sie Dokumente auf Zweckdienlichkeit. Führen Sie inhaltliche Reviews der Dokumente durch, die mehr als nur Formalien prüfen. Fragen Sie verschiedene Projektbeteiligte in unterschiedlichen Rollen nach deren inhaltlichem Urteil und Meinung. Lassen Sie Reviewer Schulnoten für die Verständlichkeit vergeben (und beseitigen Sie die Ursachen möglicher schlechter Noten!).**
- Keine degenerierten Kapitel.*** Lassen Sie keine Abschnitte oder Kapitel leer – kennzeichnen Sie absichtlich oder vorläufig leere Teile zumindest mit Stereotypen oder Formatierungen wie <<entfällt>>, <<to-be-done>> oder <<unzutreffend>>.

---

* Hier lasse ich Ausnahmen zu – wenn es 20 sinnvolle Bausteine ohne Struktur gibt, dann sollten Sie auch keine künstliche Struktur schaffen. In solchen Fällen hilft es gelegentlich, sich in manchen Diagrammen auf einen Teil der gesamten Bausteine zu beschränken (und zu erwähnen, dass es noch weitere gibt!).

** Nein, Sie sollen NICHT Ihre Reviewer beseitigen, sondern die schlecht verstandenen Teile der Dokumentation verständlich machen!

*** Der Begriff „degenerierte Kapitel“ spielt auf ein Anti-Pattern des objektorientierten Entwurfs an, bei dem Methoden einer Oberklasse in einer Unterklasse durch leere Implementierungen überschrieben werden. Im OO-Falle stellt dies eine krasse Verletzung des Liskov'schen Substitutionsprinzips dar.

Bitte beachten Sie auch die Ratschläge zur Modellierung in Kapitel 5.3.

4.3 Typische Architekturdokumente

In diesem Abschnitt stelle ich Ihnen einige häufig benötigte Arten von Dokumenten und deren mögliche Gliederung vor. Sie können diese als Vorlagen für Ihre eigenen Architekturdokumentationen verwenden.

Erstellen und pflegen Sie NUR Dokumente, die Sie und die Stakeholder des Systems wirklich benötigen.

Seien Sie bewusst sparsam!

Die folgende Liste stellt eine Obermenge dar, keinesfalls eine „Empfehlung-für-alle-Fälle“!!

Dafür benötigten Sie natürlich Werkzeuge – auf die gehe ich in Abschnitt 4.11 noch ein.

- Zentrale Architekturbeschreibung: die ausführliche Referenz. Enthält (möglichst) alle architekturrelevanten Informationen. Dazu gehören Architekturziele und Qualitätsanforderungen, Sichten, Entwurfsentscheidungen, verwendete Muster und Ähnliches. Einen Strukturvorschlag finden Sie in Abschnitt 4.9.1, ausführliche Beispiele echter Dokumentationen in Kapitel 11.
- Architekturüberblick: ein kompaktes Dokument (maximal 30 Seiten), das die Architektur des Systems motiviert und ihre wesentlichen Strukturen erläutert. Es beschreibt die Funktionsweise des Systems und begründet die wesentlichen Architekturentscheidungen. Wenn Sie keine Möglichkeit für eine ausführliche Architekturdokumentation haben, sollten Sie in jedem Fall einen solchen Architekturüberblick erstellen und pflegen. Seine Gliederung sollten Sie an den Vorschlag aus Abschnitt 4.9.1 anlehnen. Im Idealfall eine echte Teilmenge der „zentralen Architekturbeschreibung“.
- Dokumentationsübersicht: ein Verzeichnis sämtlicher architekturrelevanter Dokumente, sozusagen die Architektur der Architekturdokumentation.
- Übersichtspräsentation: eine Sammlung von Folien, mit der Sie in maximal einer Stunde die Architektur Ihres Systems fachlich und technisch motivieren und die Strukturen in den Grundzügen erläutern können.
- Architekturtapete: eine Mischung aus Baustein- und Verteilungssicht, die viele Architekturaspekte zusammen zeigt. Eine solche Tapete widerspricht dem „7 +/– 2“-Prinzip, kann allerdings manchmal eine gute Diskussionsbasis für Architekten und Entwickler sein.
- Handbuch zur Architekturdokumentation: Hier beschreiben Sie, wie die Architekturdokumentation in Ihrem Projekt funktioniert. Sie benennen Ihre Sichten mit zugehöriger Notation und erläutern die Struktur Ihrer Sichtenbeschreibungen. In vielen Fällen genügt das vierte Kapitel aus diesem Buch, die Informationen der Website [arc42] oder das kompakte [Starke+09].

- Technische Informationen zum Entwicklungsprozess: ein oder mehrere Dokumente mit den wichtigsten Informationen für Entwickler und Tester hinsichtlich Entwicklungsmethoden, Programmierung, Übersetzung/Bau sowie Start und Test des Systems. Ein solches Dokument können Sie *Programmers' Daily Reference Guide* nennen und sehr kurz halten – seine typische Leserschaft bevorzugt einstellige Seitenzahlen.

Einige Beispiele für entsprechende Architekturdokumentationen finden Sie in Kapitel 12.

Welches Werkzeug: Dokument, Modell oder Wiki?

Der Umfang der zentralen Architekturdokumentation übersteigt schnell die „Schmerzgrenze" von Textverarbeitungssystemen. Erstere lässt sich dann nur noch mit großer Mühe als einzelnes Dokument oder einzelne Datei pflegen.

Betrachten Sie den Gliederungsvorschlag aus Tabelle 4.5 als Rahmen, den Sie mit unterschiedlichen Werkzeugarten bearbeiten können:

- *Textverarbeitungssysteme*, beispielsweise Microsoft-Word™, OpenOffice™ oder ähnliche. Diese Medien haben den Vorteil der einfachen Benutzbarkeit, sind jedoch etwas *sperrig* zu pflegen.
- *(UML-)Modellierungswerkzeuge*. Hier bedarf es teilweise umfangreicher Konfiguration, bis die Reportgeneratoren der Werkzeuge die gewünschten Ergebnisse produzieren. Dafür sind Modelle meistens leichter wart- und aktualisierbar. Modelle besitzen Vorteile, wenn Sie Teile Ihrer Code-Artefakte generiere. Erläuternde Texte oder Tabellen sind mit solchen Werkzeugen häufig umständlich zu erstellen.
- *Wikis*: Kann ein Kompromiss aus Dokumenten und Modellen sein, sofern Ihr Wiki über eine ordentliche Export-Funktion verfügt. Vorsicht: Die Haltung gegenüber Wikis entwickelt sich schnell zu einer „Glaubensfrage" bei Managern und Entwicklern.

Ich persönlich mag die Kombination aus Wiki und Modellierungswerkzeug: Die textlastigen Teile (Erklärungen, Begründungen, Konzepte und jegliche Tabellen) stehen im Wiki, lediglich Diagramme pflege ich in einem UML-Werkzeug. Diese übernehme ich als jpg-Kopie ins Wiki (ja – das ist manueller Aufwand!)[6].

Mit jedem Release der Software generiere ich aus dem Wiki ein pdf, das ich zusammen mit dem Sourcecode des Systems unter Versionsverwaltung halte. Damit ist der Stand der Architekturdokumentation mit jeder „offziellen" Version der Software dauerhaft zugreifbar.

4.3.1 Zentrale Architekturbeschreibung

Die zentrale Architekturbeschreibung stellt den umfangreichsten und auch detailliertesten Teil der Architekturdokumentation dar. Sie heißt manchmal auch SWAD[7], Architektur-Gesamtdokument, Architektur-Referenz oder auch *Architecture-State-of-the-Art*. In Tabelle 4.2 finden Sie einen Gliederungsvorschlag, der auf dem (frei verfügbaren) Template [arc42] basiert.

[6] Oftmals komme ich mit einem Dutzend Diagramme aus – und die sind ziemlich grobgranular, d. h. brauchen nur selten geändert zu werden.

[7] SWAD = Software Architektur Dokumentation

TABELLE 4.2 Gliederungsvorschlag für die zentrale Architekturbeschreibung

Überschrift	Erläuterung
1. Einführung und Ziele	Die maßgeblichen Forderungen der Auftraggeber. In diesem Kapitel fassen Sie (kurz!) wichtige Punkte der Anforderungsdokumentation zusammen.
1.1 (Fachliche) Aufgabenstellung	Eine kompakte Zusammenfassung des fachlichen Umfelds. Erklärt den „Grund“ für das System. Stellen Sie die wichtigsten vom System bearbeiteten Geschäftsprozesse als Use-Cases oder User-Stories (Text und Diagramm) dar. Verweisen Sie für Details auf die (hoffentlich vorhandene) Anforderungsdokumentation.
1.2 Qualitätsziele	Beschreiben Sie hier Qualitätsanforderungen, deren Erfüllung oder Einhaltung den maßgeblichen Stakeholdern besonders wichtig ist. Sprechen Sie hier insbesondere architekturrelevante Themen wie Performance, Sicherheit, Änderbarkeit, Bedienbarkeit und Ähnliches an. Hierzu gehören auch Mengengerüste: Benennen und quantifizieren Sie wichtige Größen des Systems, etwa Datenaufkommen und Datenmengen, Dateigrößen, Anzahl Benutzer, Anzahl Geschäftsprozesse, Transfervolumen und andere. Beschränken Sie sich auf die wichtigsten (5 bis 10) Ziele.
1.3 Stakeholder	Eine Liste der wichtigsten Personen oder Organisationen im Umfeld des Systems, zusammen mit deren wesentlichen Zielen und Erwartungen an das System (beziehungsweise dessen Konstruktion und Entwicklung)
2. Einflussfaktoren und Randbedingungen	Führen Sie organisatorische und technische Randbedingungen auf, die Auswirkungen auf Architekturentscheidungen besitzen können. Führen Sie die wichtigsten Einschränkungen der Entwurfsfreiheit auf. Insbesondere gehören hier technische, organisatorische und juristische Randbedingungen sowie einzuhaltende Standards hinein.
3. Kontextabgrenzung	Sicht aus der „Vogelperspektive“, zeigt das Gesamtsystem als Blackbox und den Zusammenhang mit Nachbarsystemen, wichtigen Stakeholdern sowie der technischen Infrastruktur. Sie sollten sowohl den fachlichen als auch den technischen Kontext darstellen. Siehe Abschnitt 4.5.
4. Lösungsstrategie	Stellen Sie hier die Kernidee Ihrer Lösung dar: Was sind Ihre zentralen Lösungsansätze, Gestaltungskriterien oder Herangehensweisen? Dieses Kapitel sollten Sie kompakt halten – die ausführliche Darstellung können Sie auf die Bausteine beziehungsweise technischen Konzepte verschieben.

TABELLE 4.2 (*Fortsetzung*) Gliederungsvorschlag für die zentrale Architekturbeschreibung

Überschrift	Erläuterung
5. Bausteinsichten	Statische Zerlegung des Systems in Bausteine (Subsysteme, Module, Komponenten, Pakete, Klassen, Funktionen) und deren Zusammenhänge und Abhängigkeiten. Beginnt mit der Whitebox-Darstellung des Gesamtsystems, wird über abwechselnde Black- und Whitebox-Sichten schrittweise weiter verfeinert. Siehe Abschnitt 4.6 für die Strukturen der Bausteine und Abschnitt 4.9 für deren Schnittstellen.
5.1, 5.2, 5.3 ...	Verfeinerungsebenen 1, 2, 3 der Bausteinsicht.
6. Laufzeitsicht	Zeigen Sie das Zusammenspiel der Architekturbausteine in Laufzeitszenarien. Sie sollten hier mindestens die Erfolgsszenarien der zentralen Use-Cases beschreiben sowie weitere aus Ihrer Sicht wichtige Abläufe. Siehe Abschnitt 4.7.
7. Verteilungs- oder Infrastruktursicht	Diese Sichten zeigen, in welcher Umgebung das System abläuft, sowohl Hardware (Rechner, Netze etc.) als auch Software (Betriebssysteme, Datenbanken, Middleware etc.). Siehe Abschnitt 4.8.
8. (Übergreifende) Konzepte	Zu übergreifenden Konzepten zählen beispielsweise Persistenz, Ausnahme- und Fehlerbehandlung, Logging und Protokollierung, Transaktions- und Session-Behandlung, der Aufbau der grafischen Oberfläche, Ergonomie sowie Integration oder Verteilung des Systems. In Abschnitt 4.10 erfahren Sie, wie Sie Konzepte dokumentieren. In Kapitel 7 stelle ich Ihnen einige dieser Konzepte näher vor.
9. Entwurfsentscheidungen	Dokumentieren Sie hier die wichtigen *übergreifenden* Architekturentscheidungen und deren Gründe. Beschreiben Sie auch verworfene Alternativen. Entscheidungen bezüglich der Struktur einzelner Bausteine sollten Sie in den jeweiligen Whitebox-Darstellungen der Bausteinsicht beschreiben.
10. Qualitätsszenarien	Szenarien konkretisieren Qualitätsanforderungen und bilden eine wichtige Grundlage für Architekturentwicklung und -bewertung. Sie besitzen langfristigen Wert und bleiben meistens über längere Zeit stabil. Daher lohnt es sich, sie in die Architekturdokumentation aufzunehmen. Siehe Abschnitt 3.2.3 für Details dazu.
11. Risiken und technische Schulden	Risikomanagement ist im Allgemeinen die Aufgabe von Projektleitern und sollte auch von ihnen dokumentiert werden. Falls Ihr Projektleiter das jedoch vernachlässigt, haben Sie hier einen passenden Platz, um technische Risiken mit ihren möglichen Auswirkungen und Abhilfemaßnahmen zu dokumentieren. Legen Sie hier auch die technischen Schulden des Systems offen – das kann Ihnen bei der späteren Änderung helfen.
12. Glossar	Nur wenn nötig; ansonsten Verweis auf das Projektglossar.

4.3.2 Architekturüberblick

Der Architekturüberblick ist ein Textdokument, das die Inhalte der Übersichtspräsentation enthält, doch ohne weitere Erläuterung verständlich sein muss.

Im Architekturüberblick ist Kürze gefragt, nicht Vollständigkeit. Meine persönliche Zielvorstellung liegt bei diesen Überblicksdokumenten bei maximal 20 bis 30 Seiten – mehr mag ein gestresster Projektleiter oder termingeplagter Manager kaum lesen.

In jedem Fall enthält dieses Dokument die zentrale Lösungsstrategie, einige Sichten und erläutert daran die Funktionsweise der Architektur hinsichtlich der wichtigsten funktionalen und nichtfunktionalen Anforderungen.

Bei der Gliederung können Sie sich an der Übersichtspräsentation orientieren.

4.3.3 Dokumentationsübersicht

In der Regel werden Architekturdokumentationen aus mehreren Einzeldokumenten bestehen. Der Zusammenhang zwischen diesen Teilen wird in einer eigenständigen Dokumentationsübersicht (*Documentation Roadmap*) beschrieben.

- Welche Dokumente/Dokumentation gibt es (mit Namen und Ablageort)?
- Was ist deren Ziel und Inhalt (Abstract)?
- Welche Projektbeteiligten sollten was in welcher Reihenfolge lesen?
- Welche Abhängigkeiten bestehen zwischen den Dokumenten?

Ich habe in Projekten gute Erfahrungen damit gemacht, Dokumentationsübersichten auf der Einstiegsseite im Wiki zu beschreiben.

4.3.4 Übersichtspräsentation der Architektur

Im Laufe eines Projektes ergibt sich immer wieder die Notwendigkeit, die Architektur des Systems in Form einer Präsentation zu erläutern. Dazu sollten Sie eine Übersichtspräsentation erstellen, die Sie (möglicherweise in leichten Variationen) dann wieder verwenden können. Dafür schlage ich Ihnen folgende Gliederung vor:

- *Einführung:* Welches Problem löst das System? Verwenden Sie zwei bis drei Folien, nehmen Sie die allerwichtigsten Anforderungen an Architektur und technische Infrastruktur mit auf.
- *(Architektonischer) Lösungsansatz:* Beschreiben Sie auf ca. zwei Folien die Strategie sowie die Besonderheiten der Architektur. Benennen Sie grundlegende Architekturmuster oder -stile.
- *Kontext des Systems:* Zeigen Sie die Kontextabgrenzung mit den Außenschnittstellen sowie seinen Nachbarsystemen. Sie dürfen abstrahieren, d. h. Daten- oder Kontrollflüsse zu „größeren“ Einheiten zusammenfassen.

- *Strategische/grundlegende Lösungsentscheidungen:* Funktionsweise der Architektur hinsichtlich der wichtigsten Qualitätsmerkmale: Welche Maßnahmen haben Sie als Architekt ergriffen, um die geforderten Qualitätsmerkmale (etwa: Effizienz/Performance, Flexibilität, Skalierbarkeit, Sicherheit etc.) zu erreichen (zwei bis drei Folien)?
- *Architektursichten:* 5 bis 15 Folien zeigen jeweils eine Grafik mit Erläuterungen einiger wichtiger Architekturbausteine. Verweisen Sie auf die detaillierten Sichtenbeschreibungen. Hierin sollten mindestens ein bis zwei Bausteinsichten, eine Laufzeitsicht (für die wichtigsten Use-Cases des Systems) sowie eine Infrastruktursicht enthalten sein.
- *Übersicht über die Projektbeteiligten:* Ein bis zwei Folien stellen möglichst alle Projektbeteiligten sowie deren Ziele oder Intentionen und spezifische Informationsbedürfnisse dar.

4.3.5 Architekturtapete

In manchen Fällen kann eine Gesamtansicht vieler Architekturelemente auf einem einzigen Diagramm die Diskussion zwischen Architekten und Entwicklern vereinfachen. Eine solche Darstellung eignet sich in der Regel nicht als Einstieg oder Referenz, ermöglicht jedoch schnelle Sprünge von abstrakten zu detaillierten Aspekten der Architektur.

In meiner Berufspraxis habe ich so manches Mal die Nützlichkeit dieser „Tapeten“ erlebt:

- Sie beeindrucken damit in der Regel Ihre Manager.
- Sie können Architekturelemente (Bausteine, Knoten oder Schnittstellen) mit Statusinformationen überlagern, etwa hinsichtlich Fertigstellungsgrad, Risiko, Performance oder ähnlich wichtigen Managementinformationen.

Beachten Sie, dass die Erstellung riesiger Diagramme auch einen ungeheuren Aufwand bedeutet, ebenso wie deren Pflege. Erstellen Sie die Architekturtapete erst dann, wenn Stakeholder danach fragen.

Papier/Whiteboard und Stifte sind prima Medien für eine „große“ Übersicht dieser Art.

4.4 Sichten

Ich möchte das Konzept von Sichten mit einer Analogie aus dem Immobilienbau motivieren: Beim Entwurf und Bau von Immobilien entstehen für unterschiedliche Projektbeteiligte völlig unterschiedliche Pläne. Hier eine (unvollständige) Liste:

TABELLE 4.3 Sichten auf Immobilien

Plan/Sicht	Bedeutung	Format	Nutzer
Grund- und Aufriss	Lage und Beschaffenheit von Mauern, Maueröffnungen (Türen, Fenstern, Durchgängen), Böden, Decken	Normiert nach DIN	Architekt, Maurer, Käufer
Elektroplan	Lage von spannungsführenden Leitungen, Schaltern, Steckdosen, Verteilern, Sicherungen sowie sonstiger Elektroinstallation	Normiert nach DIN	Architekt, Käufer, Elektriker, Küchenbauer, Verwaltung (wegen Stromversorgung)
Heizungs-, Wasser- und Sanitärplan	Lage von Wasser- und Abwasserleitungen, Heizungsrohren sowie Gasleitungen	Normiert nach DIN	Architekt, Heizungs- und Sanitärinstallateur, Käufer, Küchenbauer, Verwaltung (wegen Abwasseranschluss)
3D-Modell	Dreidimensionale Darstellung des Gebäudes im Ganzen oder in Teilen	Beliebig, Bilder oder Filme („virtuelle Begehung“)	Käufer, Verkäufer
Raumplan	Zweidimensionale Darstellung von Zimmern und Einrichtung	Beliebig, angelehnt an DIN	Käufer, Innenarchitekt, Küchenbauer

Diese unterschiedlichen Pläne sind allesamt Modelle, d. h. Abstraktionen der Realität. Kein einzelner Plan gibt die gesamte Komplexität eines Hauses wieder, jeder Plan (jede Sicht) vernachlässigt gewisse Details: Jede Zielgruppe erhält genau die Pläne, die sie für ihre jeweilige Projektaufgabe benötigt!

Dieses Prinzip heißt Sichtenbildung. Viele Ingenieursdisziplinen arbeiten danach und verwenden dazu Normen, die die Struktur, Syntax und Semantik ihrer Pläne (Sichten) detailliert beschreiben.

Interpretieren Sie eine Sicht als eine spezifische Perspektive, aus der heraus Sie ein beliebiges System betrachten können.

4.4.1 Sichten in der Softwarearchitektur

- Eine einzelne Darstellung vermag die Vielschichtigkeit und Komplexität einer Architektur nicht auszudrücken. Betrachten Sie dazu Bild 4.2. Daher sollten Sie Architekturen grundsätzlich aus mehreren Sichten oder Perspektiven beschreiben.
- Sichten ermöglichen die Konzentration auf einzelne Aspekte des Gesamtsystems und reduzieren somit die Darstellungskomplexität.
- Die Beschreibung von Architekturen ist für viele Projektbeteiligte mit ganz unterschiedlichen Informationsbedürfnissen wichtig. Zum Beispiel benötigen Auftraggeber und Projektleiter andere Informationen aus der Architekturbeschreibung als Programmierer, Qualitätssicherer und Betreiber.

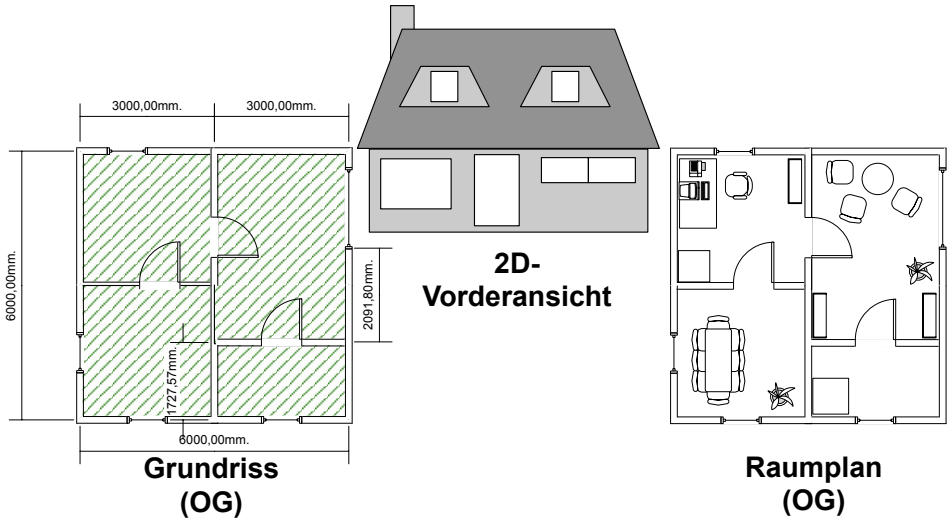


BILD 4.2 Drei Sichten auf eine Gebäudearchitektur

Stakeholder, Anforderungen und Dokumente

Stakeholder benutzen Dokumente, um Architekturen zu verstehen und darüber zu kommunizieren. Verschiedene Stakeholder benötigen meistens auch unterschiedliche Arten von Dokumenten. In Abschnitt 4.9 finden Sie Beispiele einiger typischer Architekturdokumente.

Eine Architekturdokumentation sollte sämtliche architekturrelevanten Belange der Stakeholder adressieren. Sie als Architekt stehen in der Pflicht, diese Stakeholder und deren Belange zu identifizieren und in der Architektur sowie ihrer Dokumentation zu berücksichtigen.

Sichten, Dokumente und Architekturdokumentation

Eine Sicht zeigt das System aus einer spezifischen Perspektive. Sie abstrahiert von Details, die für diese Perspektive nicht von Bedeutung sind. Sichten erlauben die Konzentration auf bestimmte Details oder bestimmte Aspekte.

Die Analogie aus der Gebäudearchitektur aus Bild 4.2 zeigt das Konzept von Sichten: Grundriss, Raumplan und 2D-Darstellung eines Gebäudes sind Sichten auf dessen Architektur, die jeweils einen bestimmten Aspekt hervorheben und dafür von anderen Details abstrahieren.

Dokumentation von Software- und Systemarchitekturen besteht ebenso aus mehreren Sichten, (meist) verteilt auf mehr als ein Dokument.

Die Diagramme oder textlichen Beschreibungen einer Sicht können auch unterschiedliche Abstraktionsebenen oder Detaillierungsstufen beschreiben. Damit können Sie verschiedene Teile einer einzigen Sicht beispielsweise für unterschiedliche Adressaten nutzen.

Sichten als Grundlage agiler Softwarearchitekturen

Architektursichten ermöglichen die praxisrelevante Beschreibung von Softwarearchitekturen. Die Sichten bleiben so flexibel, dass sie auch unterschiedlichen und heterogenen Projektanforderungen genügen. Damit bilden sie, neben den agilen Prozessen, eine wichtige Grundlage flexibler und anwendungsorientierter Softwareentwicklung.

Bei der Definition der Sichten standen *Agilität* und *Angemessenheit* im Vordergrund:

- Legen Sie Wert auf Flexibilität statt auf starre Muster. Setzen Sie Sichten bedarfsgerecht und effektiv ein, indem Sie die konkreten Informationsbedürfnisse der Stakeholder adressieren. Fragen Sie Ihre Stakeholder ausdrücklich, welche Aspekte des Systems sie für ihre jeweiligen Aufgaben benötigen.
- Und noch eine schwierige Regel: Verwenden Sie so wenig Formalismus wie möglich, aber so viel wie nötig.
- Orientieren Sie den Umfang der Dokumentation (von Sichten) am jeweiligen Risiko: Dokumentieren Sie Komponenten oder Systeme mit vielen Risikofaktoren ausführlich und detailliert. Bei Komponenten mit überschaubarem Risiko können Sie pragmatisch vorgehen und auf manche Details verzichten.

4.4.2 Vier Arten von Sichten

Bild 4.3 zeigt die vier wichtigsten Arten von Sichten, um Softwarearchitekturen effektiv und bedarfsgerecht zu beschreiben. Je nach Art Ihres Projektes können Sie diese Sichten verschieden detailliert darstellen oder unterschiedlich gewichten.

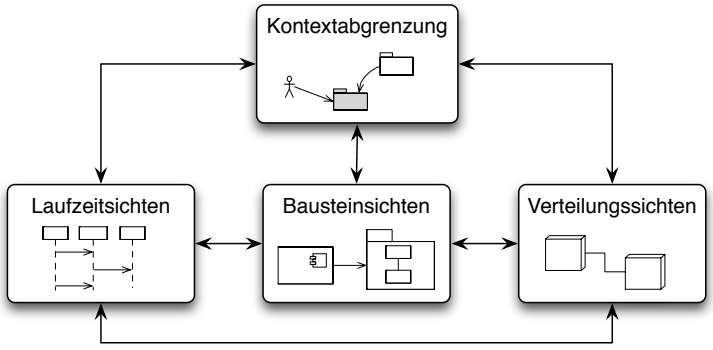


BILD 4.3 Vier Arten von Sichten

Die Pfeile zwischen den Sichten symbolisieren dabei mögliche Abhängigkeiten oder Wechselwirkungen.

Der Plural bei der Bezeichnung der Sichten drückt aus, dass Sie in der Regel mehr als eine Sicht dieser Art entwickeln werden:

- Sie verfeinern einzelne Teile der Sichten: Die interne Struktur von Elementen, die in einer (abstrakteren) Sicht als Black-Boxes stehen, wird in Verfeinerungen offengelegt und erläutert.

- Sie beschreiben unterschiedliche Architekturbausteine des gleichen Abstraktionsniveaus in verschiedenen Sichten der gleichen Art. Wenn Ihr System beispielsweise aus drei Subsystemen besteht, würden Sie zu jedem dieser Subsysteme eine eigene Baustein- und Laufzeitsicht entwickeln.

Die Sichten – ein Überblick

- *Kontextabgrenzungen* – Wie ist das System in seine Umgebung eingebettet? Kontextabgrenzungen zeigen das System als Blackbox in seinem Kontext aus einer Vogelperspektive. Hier zeigen Sie die Schnittstellen zu Nachbarsystemen, die Interaktionen mit wichtigen Stakeholdern sowie die wesentlichen Teile der umgebenden Infrastruktur.
- *Bausteinsichten* – Wie ist das System intern aufgebaut? Wie ist der Quellcode organisiert? Bausteinsichten zeigen die statischen Strukturen der Architekturbausteine des Systems, Subsysteme, Komponenten und deren Schnittstellen. Sie können (und sollten!) die Bausteinsichten Top-down beschreiben,[8] ausgehend von einer Kontextsicht. Die letzte mögliche Verfeinerungsstufe der (Baustein-)Zerlegung bildet der Quellcode. Bausteinsichten unterstützen Projektleiter und Auftraggeber bei der Projektüberwachung, dienen der Zuteilung von Arbeitspaketen (in Form von Architekturbausteinen) an Teams und Mitarbeiter und fungieren darüber hinaus als Referenz für Software-Entwickler.
- *Laufzeitsichten* – Wie läuft das System ab? Die Laufzeitsichten beschreiben, welche Bausteine des Systems zur Laufzeit existieren und wie sie zusammenwirken. Im Gegensatz zur statischen Betrachtungsweise bei den Bausteinsichten beschreiben Sie hier dynamische Strukturen.
- *Verteilungssichten (Infrastruktursichten)* – In welcher Umgebung läuft das System ab? Diese Sichten beschreiben die Hardwarekomponenten, auf denen das System abläuft. Sie dokumentieren Rechner, Prozessoren, Netztopologien und -protokolle sowie sonstige Bestandteile der physischen Systemumgebung. Die Infrastruktursicht zeigt das System aus Betreibersicht.

Gibt es noch weitere Sichten?

Neben den oben genannten Arten von Sichten benötigen manche Stakeholder Ihrer Projekte möglicherweise weitere Darstellungen bestimmter Systemaspekte.

Es spricht vieles dafür, Stakeholdern auch spezifische (oder exotische?!) Wünsche zu erfüllen, da zufriedene Stakeholder die Projektarbeit erheblich erleichtern, aber:

Mein Tipp:

Verzichten Sie möglichst auf weitere Sichten. Jede Sicht kostet Erstellungs- und Wartungsaufwand, der Sie eventuell von (wichtigeren) Architekturaufgaben abhält. Die grundlegenden Aspekte der Architektur- und Systementwicklung decken Sie in den meisten Fällen mit den vier Sichten (Kontextsicht, Bausteinsicht, Laufzeitsicht, Verteilungssicht) bereits ab.

[8] Entwickeln und bearbeiten sollten Sie die Bausteinsichten in einer für Sie passenden Reihenfolge – nicht unbedingt Top-down!

Beispiel (nach Peter Hruschka):

Beim Häuserbau könnten Kakteen- und Orchideenzüchter nach der Sonneneinstrahlung in einzelnen Räumen fragen und zum Wohle ihrer pflanzlichen Lieblinge einen gesonderten Plan wünschen. Wie groß ist Ihrer Erfahrung nach die Zahl derer, die beim Bau oder beim Kauf einer Immobilie diese „pflanzliche" Sicht als Entscheidungskriterium verwenden?

Falls Sie unbedingt eine neue Sicht beschreiben wollen (oder müssen): Definieren Sie vorher, welche Art von Elementen diese neuen Sichten enthalten und mit welcher Notation sie beschrieben werden sollen.

Wo sind Daten beschrieben?

Bei hochgradig datengetriebenen Anwendungen kann es nützlich sein, Daten- oder Informationsflüsse innerhalb des Systems explizit zu beschreiben. Dafür können Sie eigenständige Datensichten erstellen. Solche sind bei der Modellierung von Geschäftsprozessen nützlich und kommen deshalb bei der Entwicklung komplexer betrieblicher Systeme zum Einsatz, die aus vielen unterschiedlichen Softwaresystemen bestehen.

Ich erachte Datensichten nicht als reine Architektursichten, daher habe ich sie nicht in Bild 4.4 aufgenommen. Weil jedoch die Beschreibung von Datenstrukturen und -modellen gerade bei kommerziellen Informationssystemen oftmals große Bedeutung besitzt, finden Sie in Abschnitt 4.8 einige Tipps zu Datensichten.

Wo sind Schnittstellen dokumentiert?

Schnittstellen beschreiben Sie im Normalfall in den Bausteinsichten. Wenn Schnittstellen in Ihren Projekten eine herausragende Rolle spielen, dann extrahieren Sie Schnittstellenbeschreibungen in eigenständige Dokumente (und referenzieren in den Bausteinsichten darauf!).

In Abschnitt 4.7 lernen Sie, wie Sie Schnittstellen effektiv und verständlich beschreiben.

4.4.3 Entwurf der Sichten

Der Entwurfsprozess der Sichten wird von deren starken Wechselwirkungen und Abhängigkeiten geprägt. Softwarearchitekturen sollten daher iterativ entstehen, weil die Auswirkungen mancher Entwurfsentscheidungen erst über die Grenzen von Sichten hinweg spürbar werden.

Wechselwirkungen der Architektursichten

Der Entwurf einer bestimmten Architektursicht hat oftmals prägenden Einfluss auf andere Sichten. Änderungen einer Sicht ziehen Anpassungen anderer Sichten nach sich.

Betrachten Sie dazu nochmals Bild 4.3. Die Linien zeigen, wie sich die einzelnen Sichten gegenseitig beeinflussen. Nebenbedingungen oder Einschränkungen in einer der Sichten wirken sich auf die übrigen aus.

In welcher Reihenfolge entwerfen Sie die Sichten?

Letztlich spielt es kaum eine Rolle, mit welcher Architektursicht Sie beginnen. Im Laufe des Entwurfs der Softwarearchitektur werden Sie an allen Sichten nahezu parallel arbeiten oder häufig zwischen den Sichten wechseln.

Einige Tipps zum Vorgehen: Beginnen Sie den Entwurf der Architektur mit einer

- Bausteinsicht, wenn Sie:
  - bereits ähnliche Systeme entwickelt haben und eine genaue Vorstellung von benötigten Implementierungskomponenten oder technischen Konzepten besitzen;
  - ein bereits teilweise bestehendes System verändern müssen und damit Teile der Bausteinsicht vorgegeben sind.
- Laufzeitsicht, wenn Sie bereits eine erste Vorstellung wesentlicher Architekturbausteine besitzen und deren Verantwortlichkeit und Zusammenspiel klären wollen.
- Verteilungssicht, wenn Sie viele Randbedingungen und Vorgaben durch die technische Infrastruktur, das Rechenzentrum oder den Administrator des Systems bekommen.

Meiner Erfahrung nach beginnen die meisten Projekte mit einer Bausteinsicht und entwickeln parallel dazu erste Laufzeitszenarien.

Pragmatismus und Effektivität

Sie erhalten mit den Sichten ein mächtiges Werkzeug, das Sie in Projekten vielseitig unterstützen kann. Setzen Sie dieses Werkzeug angemessen und pragmatisch ein.

- Dokumentieren Sie nur so viel wie nötig. Identifizieren Sie konkrete Informationsbedürfnisse der Projektbeteiligten, dann können Sie Ihre Dokumentation darauf ausrichten.
- Anders ausgedrückt: Dokumentieren Sie sparsam. Abstrahieren Sie, beschränken Sie sich auf relevante und/oder überraschende Teile.
- Machen Sie die Architekturdokumentation allen Projektbeteiligten zugänglich.
- Stellen Sie sicher, dass alle Projektbeteiligten die Architektur verstehen. Kommunizieren Sie die Architektur und die zugehörigen Entwurfsentscheidungen. Denn: „Sie glauben nur so lange, dass Ihr Entwurf perfekt ist, bis Sie ihn jemand anderem gezeigt haben“ (nach [Rechtin2000]).
- Beachten Sie Rückmeldungen der Projektbeteiligten. Auch ein Entwurfs- oder Programmierfehler kann eine solche Rückmeldung sein – vielleicht hätten Sie anders dokumentieren sollen.

Wie viel Aufwand für welche Sicht?

Rechnen Sie damit, dass Sie 60 bis 80 % der Zeit, die Sie für den Entwurf der Architektursichten insgesamt benötigen, alleine für die Ausgestaltung der Bausteinsicht aufwenden. Der ausschlaggebende Grund hierfür: Die Bausteinsicht wird oftmals wesentlich detaillierter ausgeführt als die übrigen Sichten.

Dennoch sind die übrigen Sichten für die Softwarearchitektur und das Gelingen des gesamten Projektes wichtig! Lassen Sie sich von diesem relativ hohen Aufwand für die Bausteinsicht in keinem Fall dazu verleiten, die anderen Sichten zu ignorieren.

Spezielle Wechselwirkungen dokumentieren

Sie sollten in Ihrer Architekturdokumentation die Entwurfsentscheidungen dokumentieren, die besonderen Einfluss auf das System, dessen Erstellung und Wartung haben.

4.5 Kontextabgrenzung

Was zeigt die Kontextabgrenzung?

Kontext bedeutet Zusammenhang oder Umfeld. Die Kontextabgrenzung zeigt das Umfeld eines Systems sowie dessen Zusammenhang mit seiner Umwelt. In diesem Sinne ist die Kontextabgrenzung eine Vogelperspektive oder gesamthafte Übersicht.[9] Sie zeigt das System als Blackbox sowie dessen Verbindungen und Schnittstellen zur Umwelt. Es ist eine Sicht auf hoher Abstraktionsebene.

Fast alle Projektbeteiligten benutzen diese Vogelperspektive als Überblick oder Systemlandkarte. Sie erleichtert das Verständnis der übrigen Architektursichten.

4.5.1 Elemente der Kontextabgrenzung

Kontextabgrenzungen zeigen

- das System als Blackbox, d. h. in einer Sicht von außen;
- die Schnittstellen zur Außenwelt, zu Anwendern, Betreibern und Fremdsystemen, inklusive der über diese Schnittstellen transportierten Daten oder Ressourcen;
- die technische Systemumgebung, Prozessoren, Kommunikationskanäle.

Die Kontextabgrenzungen stellen also eine Abstraktion der übrigen Sichten dar, jeweils mit dem Fokus auf den Zusammenhang oder das Umfeld des Systems.

[9] In der Literatur finden Sie auch folgende Bezeichnungen: „Kontextsicht“ [iSAQB] oder „Conceptual Architecture View“ [Hofmeister2000].

4.5.2 Notation der Kontextabgrenzung

Da Sie in der Kontextabgrenzung abstrakte Darstellungen der Baustein-, und/oder Verteilungs-/Infrastruktursichten zeigen, können Sie auch deren jeweilige Notation verwenden:

- Schnittstellen zur Außenwelt zeigen Sie in Klassendiagrammen über Assoziationen zu anderen Systemen oder Akteuren.
- Die technische Systemumgebung zeigen Sie in Verteilungsdiagrammen.

4.5.3 Entwurf der Kontextabgrenzung

Im Idealfall erhalten Sie die Kontextabgrenzung als ein Ergebnis der Anforderungsanalyse – zumindest den fachlichen Teil davon.

Ich empfehle Ihnen, beim Entwurf der Kontextabgrenzung von einer fachlichen Sicht (Facharchitektur) auszugehen und auf dieser Basis eine Vogelperspektive des Systems zu entwerfen. Das minimiert den logischen Unterschied (*representational gap*) zwischen fachlichen und technischen Architekturmodellen.

Zeigen Sie in jedem Fall sämtliche[10] (!) Nachbarsysteme, ohne Ausnahme. Alle ein- und ausgehenden Daten und Ereignisse müssen in der Kontextabgrenzung zu erkennen sein.

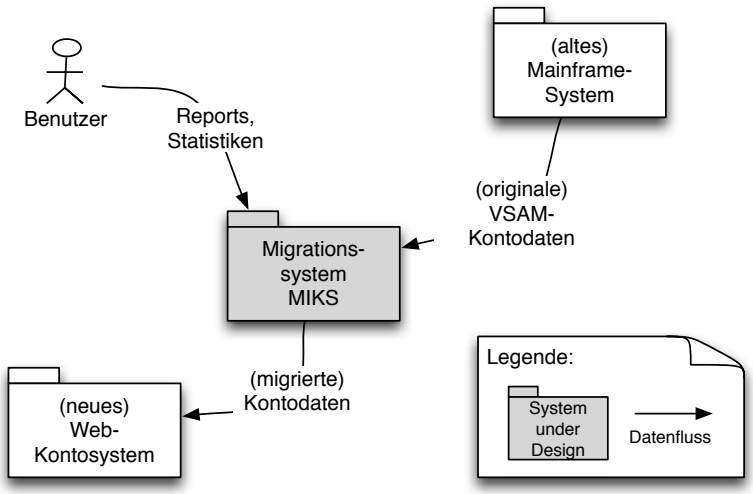


BILD 4.4 Beispiel: Diagramm einer Kontextabgrenzung

[10] Wobei Sie natürlich abstrahieren dürfen, d. h. beispielsweise mehrere ähnliche externe Schnittstellen zusammenfassen.

Verwenden Sie Kontextabgrenzungen zur Kommunikation

Kommunizieren Sie

Die Kontextabgrenzungen des Systems sowie die ersten Baustein- und Laufzeitsichten bieten Ihnen eine hervorragende Möglichkeit, mit anderen Projektbeteiligten über das System zu diskutieren. Nutzen Sie diese Gelegenheit!

- Erläutern Sie Auftraggebern oder Kunden, wie das System die Anforderungen erfüllen soll (und hören Sie auf deren Rückmeldungen).
- Beschreiben Sie der Projektleitung den Entwurf, und weisen Sie auf mögliche Risikofaktoren hin.
- Fragen Sie Entwickler nach deren Meinung. Ihr Entwurf befindet sich wahrscheinlich noch „weit weg von Technik“. Entwickler kennen häufig die Risiken und Tücken, von denen die Kontextabgrenzung abstrahiert!
- Sprechen Sie mit den künftigen Betreibern des Systems.
- Lassen Sie die Ergebnisse dieser Gespräche in die Entwürfe einfließen.

4.6 Bausteinsicht

Was sind Bausteine von Softwarearchitekturen?

Unter dem Begriff „Bausteine“ fasse ich sämtliche Software- oder Implementierungskomponenten zusammen. Bausteine repräsentieren (existierenden oder geplanten) Quellcode in verschiedenen Detaillierungsgraden. Dazu gehören Klassen, Prozeduren, Programme, Pakete, Komponenten oder Subsysteme. Bild 4.5 illustriert das in Form eines einfachen Metamodells.

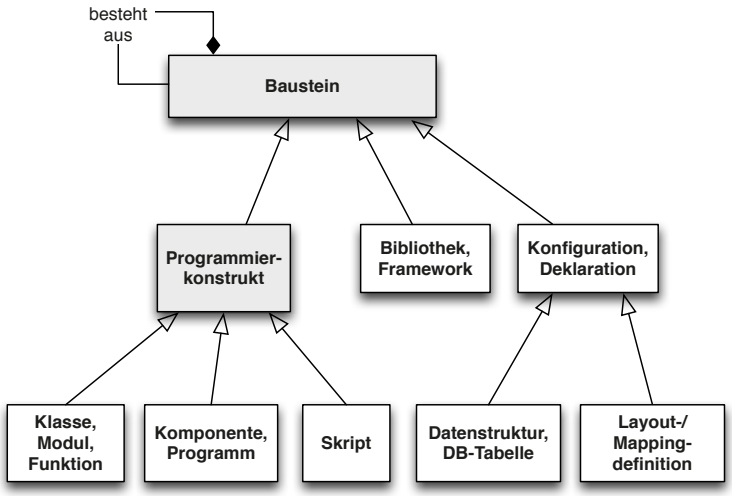


BILD 4.5 Metamodell von Bausteinen

Was zeigt die Bausteinsicht?

Die Bausteinsicht bildet die Aufgaben[11] des Systems auf Software-Bausteine oder -Komponenten ab. Diese Sicht macht Struktur und Zusammenhänge zwischen den Bausteinen der Architektur explizit. Bausteinsichten zeigen die Struktur des Quellcodes. In dieser Hinsicht entsprechen sie den konventionellen Implementierungsmodellen.

In der Bausteinsicht müssen Sie Funktionalitäten (Anwendungsfunktionalität, Kontrollfunktionalität sowie Kommunikationsaufgaben) und nichtfunktionale Anforderungen auf Architekturbausteine abbilden.[12]

Die Bausteinsicht beantwortet folgende Fragen:

- Aus welchen Komponenten, Paketen, Klassen, Subsystemen oder Partitionen besteht das System?
- Welche Abhängigkeiten bestehen zwischen diesen Bausteinen? Welcher Baustein muss welche Schnittstelle(n) implementieren?
- Welche Bausteine müssen Sie implementieren, konfigurieren oder kaufen, um die gewünschten Anforderungen zu erfüllen?

Ihre Adressaten sind alle an Entwurf, Erstellung und Test von Software beteiligten Projektmitarbeiter, bei Bedarf Auditoren oder externe Bewerter.

Bausteinsichten zeigen zwei verschiedene Arten von Abstraktion, nämlich Black- und Whiteboxes:

Blackboxes sind ausschließlich durch ihre externen Schnittstellen und ihre Funktionalität beschrieben. Sie folgen dem Geheimnisprinzip: ihr gesamtes Innenleben und somit ihre innere Komplexität bleiben verborgen.

- Whiteboxes sind *geöffnete* Blackboxes: sie zeigen deren innere Struktur und Arbeitsweise. Whiteboxes bestehen ihrerseits wiederum aus einer Anzahl von Blackboxes.

Den Zusammenhang zwischen Black- und Whitebox zeigt Bild 4.6.

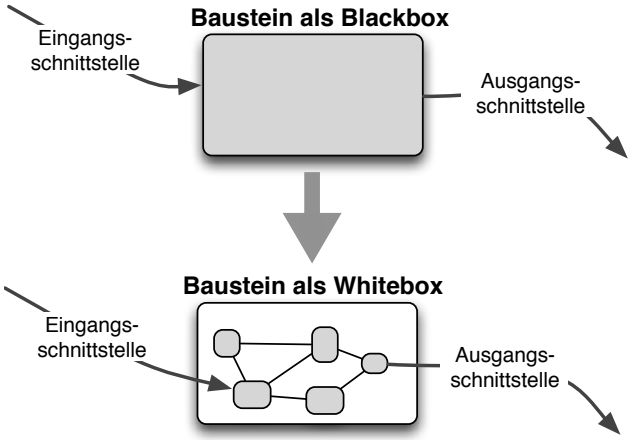


BILD 4.6
Zusammenhang
Black- und Whitebox

[11] Während der Entwicklung zeigt die Bausteinsicht die gewünschte Struktur der Bausteine, beim fertigen System die vorhandene.

[12] Daneben gibt es eine Reihe von Anforderungen, wie etwa Hochverfügbarkeit oder Clusterfähigkeit, die sich nicht ausschließlich durch systemeigene Bausteine umsetzen lassen, sondern Unterstützung durch die jeweilige Infrastruktur benötigen.

Verfeinern Sie Bausteinsichten Top-down

Sie können Blackboxes weiter verfeinern, indem Sie ihren „Inhalt“ als Whitebox zeigen. In Bausteinsichten zeigen Sie abwechselnd Blackbox- und Whitebox-Darstellungen. Hier stellen Sie Bausteine in Hierarchien oder Architekturebenen („Levels“) dar, wobei Sie den Detaillierungsgrad der Sichten schrittweise verfeinern.

Bild 4.7 zeigt diese Hierarchie von Verfeinerungen. Sie beginnt grundsätzlich mit der Kontextsicht, die das gesamte System als Blackbox darstellt. Die erste Verfeinerungsebene („Level 1“) stellt dann das Gesamtsystem als Whitebox dar.

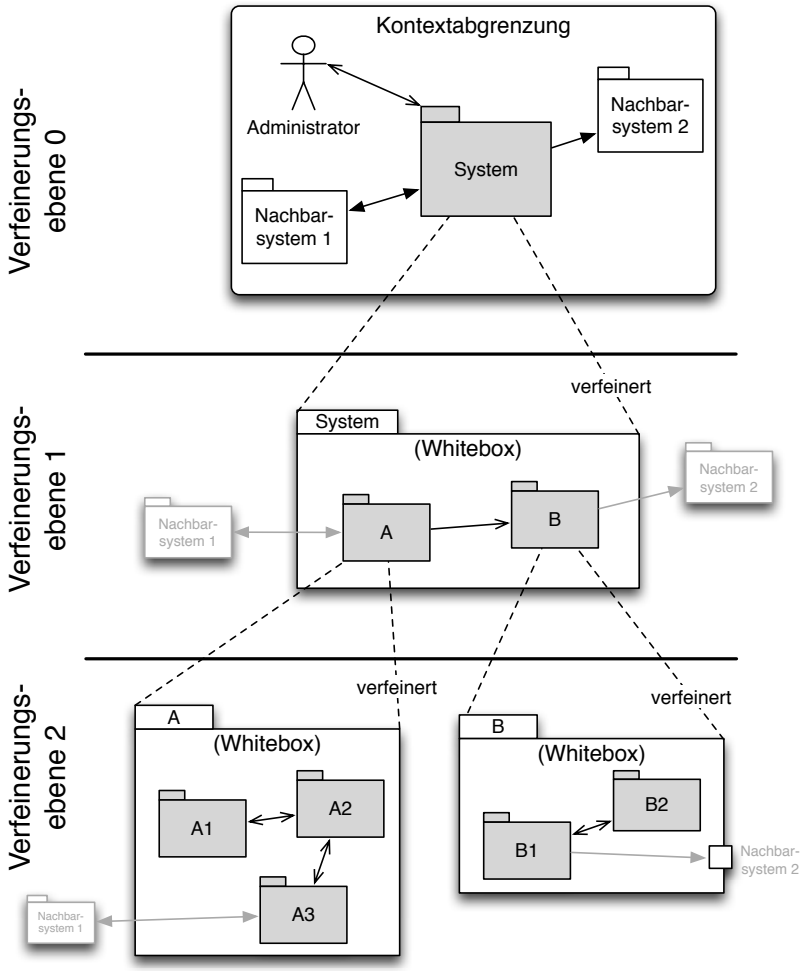


BILD 4.7 Hierarchie und Verfeinerung von Bausteinsichten

Nutzen Sie die Verfeinerung pragmatisch

Manchmal treten bei der schrittweisen Verfeinerung von Systemen Fälle auf, bei denen die oben gezeigte Art der Hierarchie schwierig oder umständlich ist. Wägen Sie in diesen Fällen Verständlichkeit, Wartbarkeit und Einfachheit ab. Im Zweifelsfall fragen Sie Ihre Leser!

Einige Beispiele solcher Situationen:

- Die Detaillierung einer Blackbox lässt sich durch Quellcode oder anderen Text leichter oder präziser beschreiben als durch Diagramme.
- Sie müssen mehrere Blackboxes gemeinsam verfeinern. Nutzen Sie in diesem Fall die Hierarchie im Sinne Ihrer Leser: Verfeinern Sie, was zusammengehört. In der folgenden Bild 4.8 sehen Sie die Blackboxes A und B aus Bild 4.7 gemeinsam in einer Whitebox.

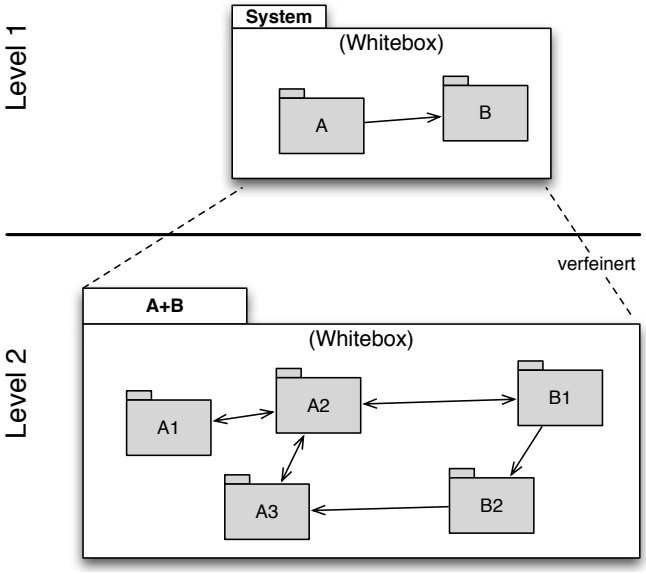


BILD 4.8
Gemeinsame Verfeinerung von Bausteinen

Verfeinern Sie redundanzfrei

Sie sollten bei der zunehmenden Detaillierung der Bausteinsicht auf wiederkehrende Muster oder Strukturen achten und die dadurch drohenden Redundanzen vermeiden: Dokumentieren Sie solche Muster nur einmal, und verweisen Sie bei weiterem Auftreten jeweils auf diese Musterdokumentation.

Ein bekanntes Beispiel dafür ist der Zusammenhang von Domänenobjekten mit ihrer Benutzeroberfläche: Innerhalb eines GUI-Frameworks wird diese Struktur immer gleich aussehen – daher sollte sie auch nur einmal beschrieben werden.

In diesem konkreten Fall bietet sich meiner Meinung nach eine Kombination aus Diagramm und Quellcode zur Dokumentation an. Alternativ beschreiben Sie solche Sachverhalte als technisches Konzept (siehe Abschnitt 4.10).

4.6.1 Elemente der Bausteinsicht

Verwenden Sie für Bausteine (Implementierungskomponenten) die folgenden Elemente (Symbole) der UML – unabhängig davon, ob Sie Black- oder Whiteboxes darstellen:

- Klassensymbole bezeichnen einzelne Bausteine. Das können Klassen einer objektorientierten Programmiersprache sein, aber auch alle anderen ausführbaren Software-Einheiten (Funktionen, Prozeduren, Programme).
- Komponenten bezeichnen ebenfalls einzelne Bausteine, bieten jedoch die Möglichkeit, die ein- und ausgehenden Schnittstellen genau zu beschreiben.
- Pakete stehen für Gruppen, Mengen oder Strukturen von Bausteinen. Diese Symbole zeigen, dass es sich um Abstraktionen handelt (die in der Architektur oder im detaillierten Entwurf weiter verfeinert werden).

Beschreiben Sie Verantwortlichkeiten und Schnittstellen von Blackboxes

Sie sollten zu jedem Blackbox-Baustein dessen spezifische Verantwortlichkeit oder Aufgabe in Kurzform dokumentieren.

Zusätzlich beschreiben Sie die Ein- und Ausgabeschnittstellen der Blackboxes. Schnittstellen können verschiedene Ausprägungen annehmen, etwa: Aufruf/Rückgabe (*call-return*), synchron-asynchron, *push/pull* oder Benachrichtigung (*event notification*). Sie sollten die Semantik dieser Beziehungen angemessen dokumentieren:

- Falls Sie eine bestimmte benötigte Semantik mit den Symbolen der UML nicht direkt ausdrücken können, benutzen Sie Notizen, oder verfeinern Sie die Darstellung durch zusätzliche Diagramme.
- Beschreiben Sie kritische Schnittstellen, etwa zu Nachbarsystemen, möglichst früh im Projekt und mit der gebotenen Genauigkeit für die Lesergruppe, so dass Überraschungen in der Zusammenarbeit mit Außenstehenden rechtzeitig entdeckt werden können.

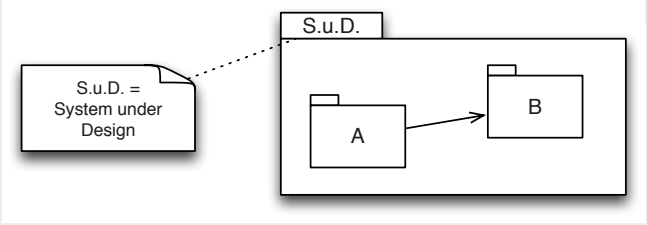
Weitere Hinweise und Vorschläge zur Dokumentation von Schnittstellen finden Sie in Abschnitt 4.9.

Ich empfehle Ihnen, zu Blackboxes den Zusammenhang zu den wichtigsten zugehörigen Code-Artefakten ausdrücklich zu beschreiben – ein Verweis auf die entsprechenden Dateien oder Pakete genügt. Damit dokumentieren Sie, welche Teile des Quellcodes diese Blackbox implementieren. Beschränken Sie sich dabei auf die *wesentlichen* Teile, und hüten Sie sich vor zu vielen Details – sonst wird die Pflege der Dokumentation zu teuer.

Beschreiben Sie die Struktur von Whiteboxes

Jede Whitebox-Beschreibung muss die innere Struktur eines Bausteins darstellen. Dabei helfen Diagramme und Modelle – ergänzt durch textuelle Erläuterung. Jede Whitebox-Beschreibung sollte die in Tabelle 4.4 beschriebenen Elemente umfassen.

TABELLE 4.4 Elemente von Whitebox-Beschreibungen

Überschrift	Inhalt
Übersichts-diagramm	Ein Diagramm (UML Paket- oder Klassendiagramm), das die innere Struktur dieser Whitebox zeigt. Ein Beispiel (Auszug aus Bild 4.7):  **BILD 4.9** Beispiel: Whitebox-Darstellung
Lokale Bausteine	Tabelle oder Liste der lokalen Blackbox-Bausteine. Deren interne Struktur können Sie in einer weiteren Verfeinerungsebene darstellen.
Lokale Beziehungen	Tabelle oder Liste der Abhängigkeiten und Beziehungen zwischen den lokalen Bausteinen.
Entwurfs-entscheidungen	Gründe, die zu dieser Struktur geführt haben (oder zur Ablehnung von Alternativen).

Gemeinsamkeiten mit anderen Systemen ausnutzen

Bevor Sie sich an den Entwurf der Bausteinsicht begeben, prüfen Sie, ob Sie Teile dieses Systems aus anderen Quellen wieder verwenden können. Suchen Sie in anderen Projekten, in der Literatur und bei Anbietern von Software-Komponenten nach Teilen, die Sie für Ihr aktuelles Projekt (wieder) verwenden können!

Suchen Sie dabei auch an den Stellen, die nicht unbedingt zum Kern Ihres konkreten Systems gehören, sondern allgemeine (und daher unauffällige) Basisdienste darstellen!

Selbst wenn diese Wiederverwendung nur kleine Teile des Systems betrifft, die Taktik des „Nicht-mehr-entwerfen-Müssens“ wirkt sich positiv aus!

4.6.2 Notation der Bausteinsicht

Zur Notation der Bausteinsicht stehen Ihnen verschiedene Diagrammarten der UML zur Verfügung. Meine Empfehlung: Zeigen Sie diese Sicht in jedem Fall durch Klassendiagramme, bei Bedarf unterstützt durch Komponenten- und Paketsymbole. UML-Klassen in diesen Diagrammen repräsentieren beliebige Bausteine oder Implementierungskomponenten.

UML-Diagramme: siehe Kapitel 5

Pakete symbolisieren Bausteine, die eine (komplexe) innere Struktur aufweisen, deren exakte Schnittstelle(n) Sie nicht genauer beschreiben wollen oder die keine physische Repräsentation zur Laufzeit besitzen. Sie stellen eine logische Kapselung oder Abstraktion dar.

UML-Komponenten sind Bausteine, für die Schnittstellenbildung und Kapselung wichtig sind. Zur Laufzeit wird es immer mindestens eine Instanz von Komponenten geben.

4.6.3 Entwurf der Bausteinsicht

Beim Entwurf der Bausteinsicht sind Sie als Architekt sozusagen beim Kern des eigentlichen Problems angelangt. Jetzt müssen Sie exakt beschreiben, wie das System (strukturell) aufgebaut ist und aus welchen Bausteinen es besteht.

Beginnen Sie[13] den Entwurf mit der Kontextabgrenzung, der *Vogelperspektive* der Implementierungsbausteine. Jetzt gilt es, kleinere Architekturelemente wie Sub- oder Teilsysteme, Komponenten oder sogar atomare Bestandteile zu finden. Mittlerweile sind Sie mit allerlei Rüstzeug gewappnet, diese Herausforderung zu meistern!

- Sie haben auf der Basis einer „Systemidee“ einen „ersten Wurf“ des Systems vor Augen. Wenn nicht: Unternehmen Sie einen (literarischen) Ausflug in Kapitel 3 (Vorgehen bei der Architekturentwicklung), und kehren Sie anschließend hierher zurück.
- Sie berücksichtigen Einflussfaktoren und Randbedingungen bei der Definition von Bausteinen. Insbesondere kennen Sie:
  - die technische Infrastruktur: Wo und wie muss das System ablaufen (Rechner, Betriebssysteme, Netzwerke)?
  - die technischen Einflussfaktoren (Programmiersprachen, Middleware, Datenbank, Transaktionsmonitor, Entwicklungsumgebung)
  - die organisatorischen Einflussfaktoren: Über welche Erfahrung verfügt das Team?
  - die Qualitätsanforderungen: Welche Performance und Zuverlässigkeit muss das System erreichen, welche Ansprüche bestehen hinsichtlich Änderbarkeit und so weiter? (Siehe dazu auch Kapitel 3, Vorgehen bei der Architekturentwicklung.)
- Sie können beim Entwurf der obersten Abstraktionsebene der Bausteinsichten von Elementen oder Begriffen der Fachdomäne starten.
- Sie können Architekturmuster oder Referenzarchitekturen einsetzen. Einige Hinweise dazu erhalten Sie in Kapitel 6 (Strukturentwurf).

Folgende Zerlegungsstrategien könnten Ihnen nützlich sein:

- Zerlegung nach fachlichen oder funktionalen Gesichtspunkten (= Domain-Driven Design, Zerlegung nach Anwendungsfällen)

[13] Ich finde es ideal, Entwürfe im Team zu diskutieren, um frühzeitig Feedback zu Entscheidungen zu bekommen.

- Zerlegung gemäß der Verteilungssicht, Hardware oder anderer technischer Infrastruktur (etwa: Bausteine für mobile Clients, Browser-Clients und/oder Server)
- Zerlegung gemäß organisatorischer Verantwortungen (z. B. analog der Struktur der Entwicklungsorganisation)
- Zerlegung gemäß vorhandenen Technologien, Frameworks, Programmiersprachen oder eingesetztenr Produkten.
- Zerlegung gemäß wichtiger Qualitätsanforderungen (etwa: Ein Baustein muss im Cluster laufen, während andere Bausteine das nicht müssen).

4.7 Laufzeitsicht

Alias: Ausführungssicht

Die Laufzeitsicht beschreibt, welche Bestandteile des Systems zur Laufzeit existieren und wie sie zusammenwirken. Dabei kommen wichtige Aspekte des Systembetriebs ins Spiel, die beispielsweise den Systemstart, die Laufzeitkonfiguration oder die Administration des Systems betreffen.

Primär dokumentiert die Laufzeitsicht, wie das System zur Laufzeit seine wesentlichen Aufgaben (= Funktionen, Use-Cases, Abläufe) ausführt.

Dokumentieren Sie das System zur Laufzeit

Neben der internen Struktur von Software-Systemen sollten Sie beschreiben, welche Komponenten zur Laufzeit existieren, wie diese Komponenten zusammengesetzt sind und wie sie ablaufen.

Beispiel: Die Fachlogik eines Systems im industriellen Bereich besteht aus einem Paket von nur 3 Klassen. Zur Laufzeit wird das System durch einen komplexen Konfigurationsprozess für jeden angeschlossenen Client als eigenständiger Betriebssystemprozess gestartet und durch einen Monitorprozess kontinuierlich überwacht.

Die Konfigurations- und Administrationskomponenten stellen einen großen Teil der Laufzeitsicht dar. Sie müssen auch in Bausteinsichten erscheinen. Möglicherweise (da es sich um vorgefertigte Komponenten handelt) werden in der Bausteinsicht nur ihre Schnittstellen beschrieben.

Dokumentieren Sie spezielle Aspekte der Laufzeit:

- Wie arbeiten die Systemkomponenten zur Laufzeit zusammen?
- Wie werden die wichtigsten Use-Cases durch die Architekturbausteine bearbeitet?
- Welche Instanzen von Architekturbausteinen gibt es zur Laufzeit, und wie werden diese gestartet, überwacht und beendet?

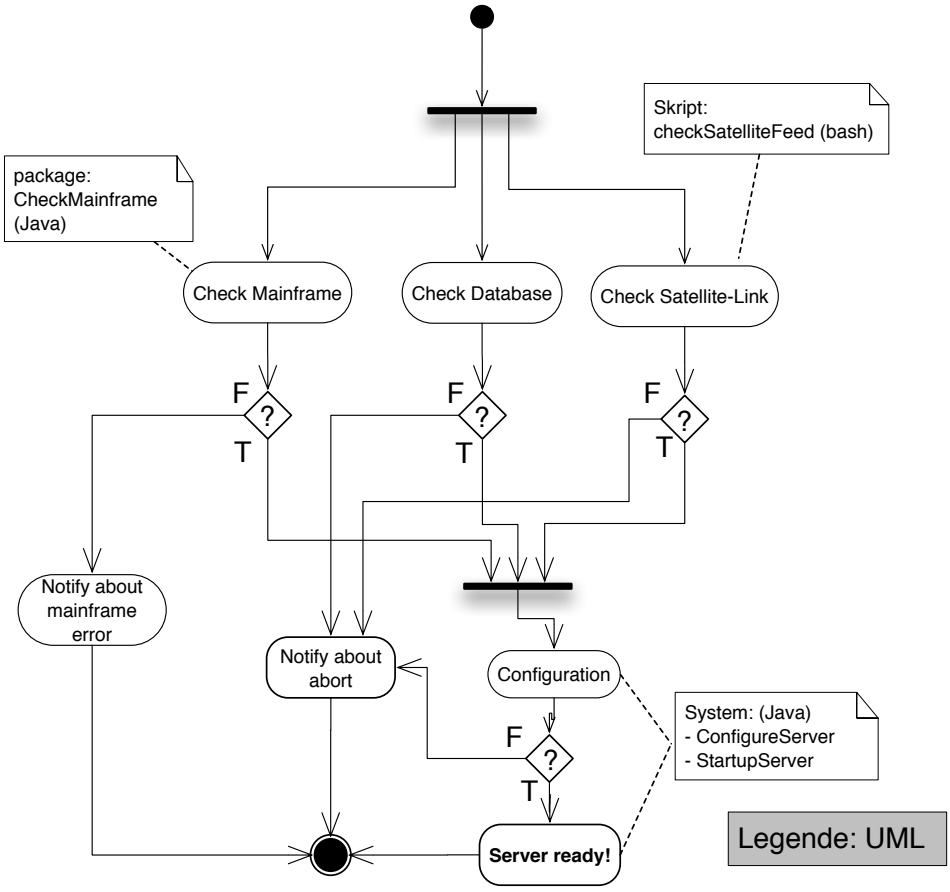


BILD 4.10 Beispiel einer Laufzeitsicht (Systemstart)

- Wie arbeiten Systemkomponenten mit externen und vorhandenen Komponenten zusammen?
- Wie startet das System (etwa: notwendige Startskripte, Abhängigkeiten von externen Subsystemen, Datenbanken, Kommunikationssystemen etc.)?

4.7.1 Elemente der Laufzeitsicht

Elemente der Laufzeitsicht sind ausführbare Einheiten sowie deren Beziehungen zueinander:

- Instanzen von Implementierungsbausteinen – alle diejenigen Architekturbausteine, von denen zur Laufzeit Exemplare erzeugt werden. Bitte beachten Sie, dass dies nicht unbedingt für alle Architekturbausteine gilt. Denken Sie an abstrakte Klassen oder logische Interfaces. Laufzeitelemente sind nur diejenigen, die als Einheit ausgeführt oder aufgerufen werden.
- Beziehungen zwischen den Laufzeitelementen – das können sowohl Daten- als auch Kontrollflüsse sein.

4.7.2 Notation der Laufzeitsicht

Dokumentieren Sie die Laufzeitsicht mit UML-Sequenz-, Aktivitäts- oder Kollaborations-/Kommunikationsdiagrammen. In Bild 4.11 zeigt ein Sequenzdiagramm das Zusammenwirken einiger Bausteine der Kundendaten-Migration MIKS.

Gerade bei den Sequenzdiagrammen sollten Sie die UML pragmatisch einsetzen: Anstelle der üblichen Objektsymbole können Sie bei Bedarf auch Komponenten-, Paket- oder andere Symbole verwenden. Dazu finden Sie ein Beispiel in Bild 4.11 (Komponente MigrationDatabase) und Bild 4.12.

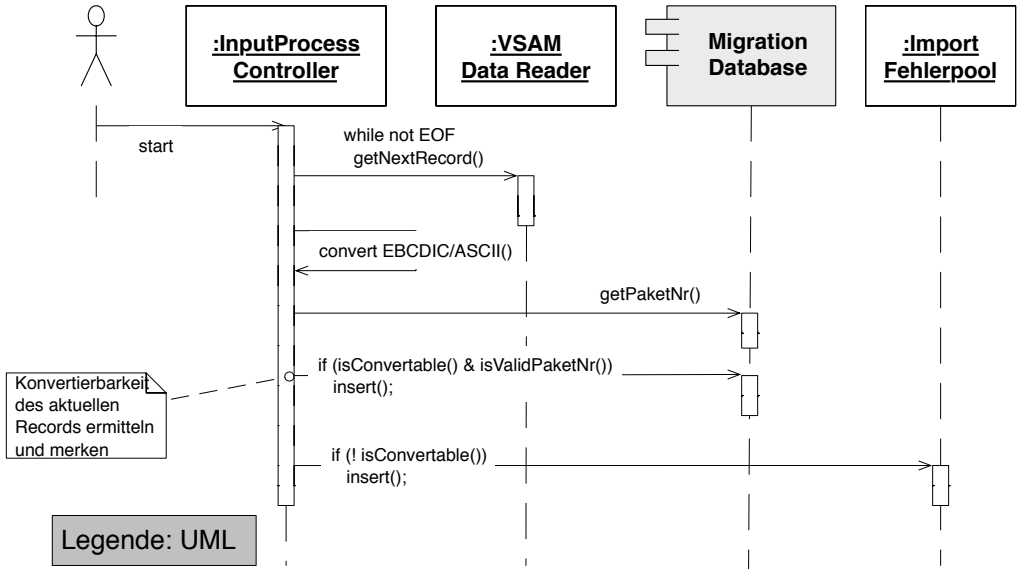


BILD 4.11 Beispiel: Laufzeitsicht MIKS Input-processing

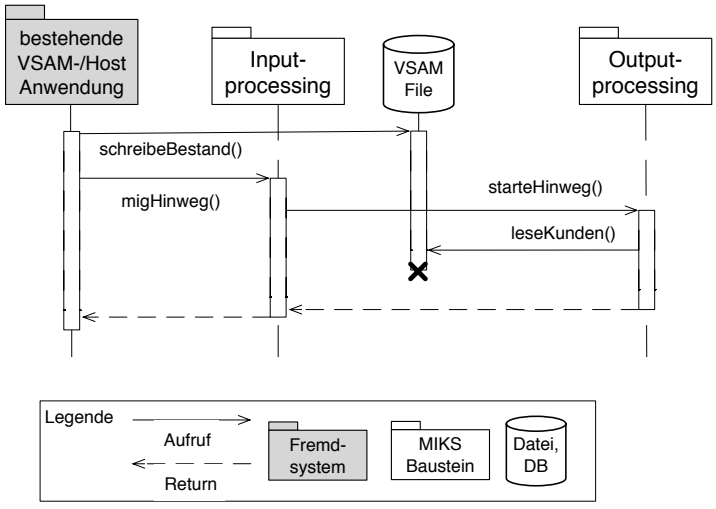


BILD 4.12 Beispiel: Laufzeitsicht Top-Level Use-Case MIKS

Eine Laufzeitsicht als Aktivitätsdiagramm zeigt Bild 4.10. Dort sehen Sie den Startvorgang eines Systems, bei dem zur Laufzeit vor der Konfiguration und dem Start des eigentlichen Systems die Verfügbarkeit einiger externer Subsysteme geprüft wird. Als UML-Notiz sind außerdem die Namen einiger beteiligter Skripte und Pakete genannt, die nicht Bestandteil des eigentlichen Systems sind.

Tipp:

Zur Beschreibung von Abläufen können Sie auch nummerierte Listen oder Quellcode verwenden, sofern er für Ihre Leser gut verständlich ist und – meine persönliche Regel – der entsprechende Text auf eine einzige Seite passt.

4.7.3 Entwurf der Laufzeitsicht

Beachten Sie bei der Dokumentation der Laufzeitsicht auch die Informationsbedürfnisse der Betreiber und Administratoren des Systems, nicht nur die von Managern und Entwicklern.

Elemente der Laufzeitsichten sind Instanzen der (statischen) Architekturbausteine, die Sie in den Bausteinsichten dokumentieren. Ein möglicher Weg zur Laufzeitsicht führt daher über die Bausteinsichten: Beschreiben Sie die Dynamik der statischen Bausteine, beginnend bei den wichtigsten Use-Cases des Gesamtsystems. Eventuell hilft Ihnen dabei auch die Kontextsicht, die Sie in Abschnitt 4.5 kennengelernt haben.

Ein anderer Weg besteht darin, die Laufzeitsicht auf Basis der Verteilungs-/Infrastruktursicht (siehe folgender Abschnitt) zu entwickeln.

4.8 Verteilungssicht

Alias: Infrastruktursicht

Technische Infrastruktur

Die Verteilungs- oder Infrastruktursicht) verfolgt zwei Intentionen:

1. Abbildung der technischen Ablaufumgebung des Systems in Form von Hardwarekomponenten (Prozessoren, Speicher, Netzwerke, Router und Firewalls);
2. Abbildung (= Deployment, Verteilung, Installation) Ihrer Softwarebausteine auf die Hardware.

Sie können in der Infrastruktursicht die Leistungsdaten und Parameter der beteiligten Elemente darstellen, wie etwa Speichergrößen, Kapazitäten oder Mengengerüste. Außerdem können Sie zusätzlich die Betriebssysteme oder andere externe Systeme aufnehmen.

Die Verteilungssicht ist von großer Bedeutung für die Betreiber des Systems, die Hardware-Architekten, das Entwicklungsteam sowie Management und Projektleitung.

4.8.1 Elemente der Verteilungssicht

In der Verteilungssicht beschreiben Sie folgende Elemente:

- Bestandteile der technischen Infrastruktur, so genannte Knoten. Dies sind entweder echte oder virtuelle Rechner oder Prozessoren, aber auch sonstige Hardware (Firewalls, Router, Speicher oder anderes).
- Laufzeitelemente, auch Laufzeitartefakte genannt (d. h. Instanzen von Bausteinen, siehe Abschnitt 4.6), die auf Knoten ablaufen (und dort installiert, neudeutsch *deployed*, werden).
- Kanäle sind Verbindungen zwischen Knoten (physische Kanäle) beziehungsweise zwischen Laufzeitelementen (logische Kanäle). Logische Kanäle müssen immer über physische Kanäle realisiert werden.

4.8.2 Notation der Verteilungssicht

Deployment Diagramme

Nutzen Sie für die Verteilungs-/Infrastruktursicht UML-Einsatzdiagramme (*deployment diagrams*). Sie können Knoten für beliebige technische Elemente verwenden, Komponenten- und Paketsymbole für die Laufzeitelemente (Software-Systeme) sowie UML-Beziehungen für die physischen Verbindungen zwischen den Knoten oder Laufzeitelementen.

Ein Beispiel finden Sie in Bild 4.13.

Das Mapping „Baustein-auf-Knoten“ ist tabellarisch eventuell einfacher als grafisch – möglicherweise haben Sie das ohnehin in Ihrem Build-Skript beschrieben.

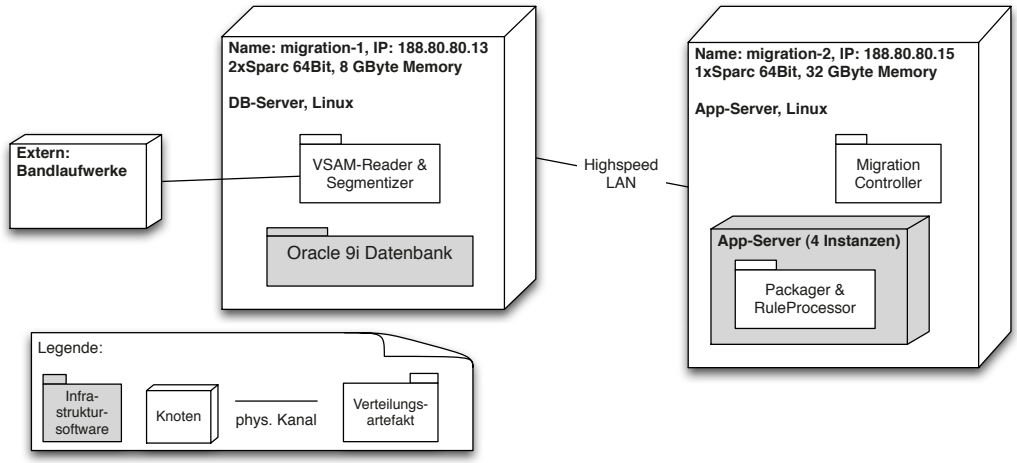


BILD 4.13 Beispiel für Verteilungssicht

Noch ein ergänzendes Wort zu den Kanälen: Für manche Stakeholder sind die logischen Kanäle wichtiger als die physischen oder umgekehrt. Erfragen Sie, welche Bedürfnisse Ihre Stakeholder für die Verteilungssicht genau haben. Für Entwickler sind oftmals die logischen interessanter, für Betreiber und Manager die physischen.

4.8.3 Entwurf der Verteilungssicht

- Ein Modell der Infrastruktursicht sollte eine Landkarte der beteiligten Hardware und der externen Systeme sein. Diese Information sollte sämtlichen Projektbeteiligten zur Verfügung stehen.
- Gleichen Sie die Infrastruktursicht mit den Mengengerüsten der beteiligten Daten ab. Genügen die verfügbare Hardware und die Kommunikationskanäle, oder gibt es potenzielle Engpässe?
- Sie können Mengengerüste und Datenvolumina mit der Infrastruktursicht überlagern, um potenzielle Engpässe zu erkennen.
- Falls Ihre Systeme in verteilten oder heterogenen Umgebungen ablaufen, sollten Sie vorhandene Kommunikationsmechanismen, Protokolle und Middleware in die Infrastruktursicht aufnehmen. Das ermöglicht einen Abgleich mit der Bausteinsicht.
- In heterogenen und verteilten Systemlandschaften sollten Sie eine detaillierte Sicht der technischen Infrastruktur erstellen. Diese sollte mindestens folgende Aspekte berücksichtigen: Netztopologie und Protokolle, real verfügbare Netzkapazitäten (maximale und mittlere Werte), Anzahl und Leistung der vorhandenen Prozessoren sowie die Speicherkapazitäten der beteiligten Rechner.
- Berücksichtigen Sie die realen Verfügbarkeiten der technischen Komponenten. Teilweise sind Komponenten durch nächtliche Datensicherungen, Batch-Läufe oder Wartungsarbeiten regelmäßig nicht verfügbar. Dokumentieren Sie diese Einschränkungen.
- Oftmals ist die „Verpackung“ von Implementierungsbausteinen in Deployment-Artefakte eine aufwendige oder komplexe Aufgabe, die hoffentlich Ihr Build-Prozess automatisiert. Bei Bedarf übernehmen Sie die Zuordnung von Bausteinen zu Deployment-Artefakten in Ihre Architekturdokumentation, oder Sie verweisen auf Ihre Build-Skripte.

4.9 Dokumentation von Schnittstellen

Schnittstellen besitzen als Abgrenzungen zwischen den verschiedenen Bausteinen von Systemen besondere Bedeutung: Über Schnittstellen arbeiten diese zusammen und erzeugen letztlich den Mehrwert des Gesamtsystems.

Schnittstellen beinhalten in der Regel den Transfer von Daten oder Steuerungsinformationen. Manche Schnittstellen basieren auf Standardkonstrukten der eingesetzten Programmiersprachen (Methoden- oder Prozeduraufrufe, Remote-Procedure-Calls, Shared-Memory etc.). Andere Schnittstellen sind organisatorischer Art.

Viele Schnittstellen werden bereits als Bestandteile von Sichten dokumentiert, insbesondere in der Bausteinsicht. Häufig benötigen manche Stakeholder eigenständige Schnittstellendokumentationen – insbesondere für externe Schnittstellen.

Struktur von Schnittstellenbeschreibungen

Wie auch bei der Beschreibung von Sichten sollten Sie bei der Beschreibung von Schnittstellen eine einheitliche Gliederung verwenden.

Nachfolgend bezeichnet der Begriff *Ressource*[14] jeglichen Baustein einer Softwarearchitektur, der eine Interaktion über die betroffene Schnittstelle anbietet.

Ressourcen – ein überladener Begriff

Allgemein gesprochen ist eine Ressource ein Arbeitsmittel. In der Softwarearchitektur verwenden wir den Begriff in zwei leicht unterschiedlichen Ausprägungen:

- Die unspezifische Verwendung (die ich im Folgenden auch für die Schnittstellenbeschreibung benutze) subsumiert einen beliebigen Datenstrom, ohne weitere Einschränkung.
- Der REST-Architekturstil (representational state transfer) verwendet Ressourcen als Bestandteile von (Online-) Interaktionen und differenziert strikt zwischen Ressourcen und deren Repräsentationen.
  Siehe *http://www.w3.org/TR/webarch*

[14] Das Konzept der ressourcenbasierten Schnittstellenbeschreibungen stammt aus [Clements+03].

TABELLE 4.5 Struktur von Schnittstellenbeschreibungen

Überschrift	Inhalt
Identifikation	Genaue Bezeichnung und Version der Schnittstelle
Beispiele	Beispiele zur Benutzung dieser Schnittstelle; Quellcode, etwa in Form von Unit-Tests*

Bereitgestellte Ressourcen	Welche Ressourcen stellt dieses Element für Ihre Akteure (Benutzer, Aufrufer) bereit? Sie müssen hier mehrere Aspekte dokumentieren:
	▪ Syntax der Ressource, etwa die Signatur von Methoden, das API
	▪ Semantik der Ressource: Welche Auswirkungen hat ein Aufruf dieser Ressource? Hier gibt es mehrere Möglichkeiten:
	  ▪ Welche Events werden ausgelöst?
	  ▪ Welche Daten werden (wie) geändert?
	  ▪ Welche Zustände ändern sich?
	  ▪ sonstige von Menschen oder anderen Systemen wahrnehmbare Nebenwirkungen
	▪ Restriktionen bei der Benutzung der Ressource

Fehlerszenarien	Beschreiben Sie sowohl mögliche Fehlersituationen als auch deren Behandlung.
Qualitäts-eigenschaften	Welche Qualitätseigenschaften wie Verfügbarkeit, Performance, Sicherheit gelten für diese Schnittstelle? Die neudeutsche Bezeichnung dieses Teils der Schnittstellenbeschreibung lautet *Quality-of-Service* (QoS) *Requirements*.
Entwurfs-entscheidungen	Welche Gründe haben zum Entwurf dieser Schnittstelle geführt? Welche Alternativen gibt es, und warum wurden diese verworfen?

* Ein großartiges Vorbild ist die JOptSimple-Library, deren API hervorragend durch Unit-Tests beschrieben ist.

Schnittstellen in UML beschreiben

Bei der UML-Darstellung von Bausteinen haben Sie eine Reihe von Optionen (siehe Bild 4.14) für Schnittstellen – nachfolgend in der Reihenfolge der „Gründlichkeit“ und des damit verbundenen Aufwandes erklärt:

- Eine benannte und/oder gerichtete Verbindung zweier Bausteine sagt lediglich aus, dass es überhaupt eine Schnittstelle gibt.
- Ein Port-Symbol (kleines Viereck am Rande einer Komponente) gibt an, dass ein Baustein eine Schnittstelle besitzt – differenziert nicht zwischen Anbieten und Benötigen.
- Ein Baustein referenziert einen Port: Dieser Baustein (intern in Bezug auf die umgebende Komponente) realisiert oder benötigt diese Schnittstelle.
- Explizte Angabe von angebotener („Ball“) und benötigter („Socket“) Schnittstelle – mit (optional benannter) Beziehung dazwischen. Hierbei ist die Schnittstelle ein eigenständiges Modellierungselement mit (beliebig genauer) Beschreibung.
- Explizite Ball- und Socket-Notation in Kombination mit Ports: Hier können Sie bei Bedarf sogar die Schnittstellen noch Kommunikationswegen oder Technologien zuordnen.

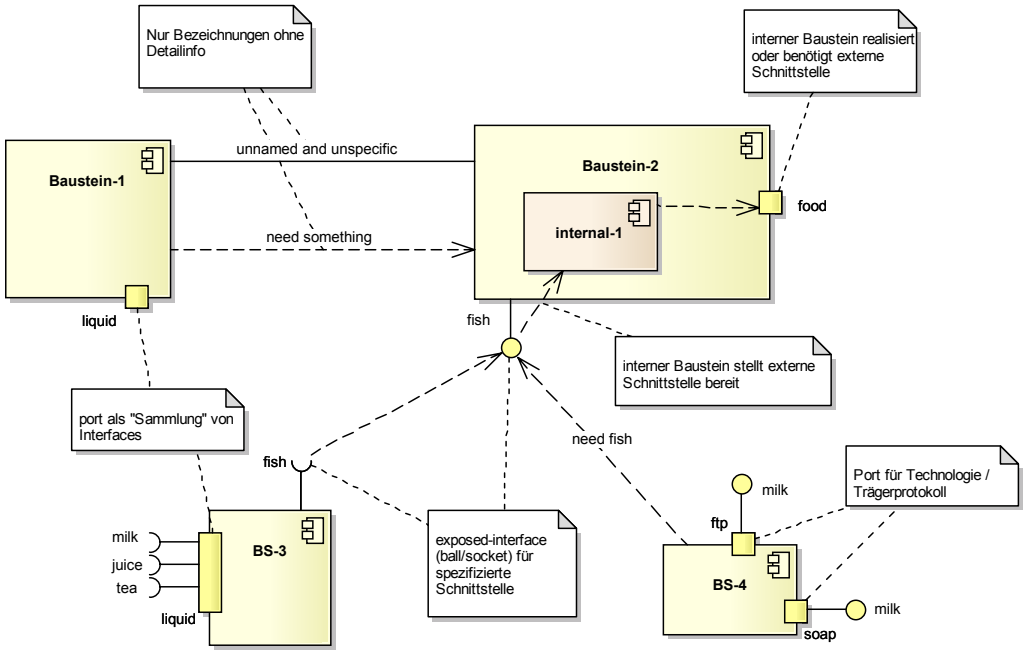


BILD 4.14 Optionen für Schnittstellen in UML

Einige Tipps zur Modellierung von Schnittstellen mit UML:

- Beginnen Sie sparsam mit der gerichteten und benannten Verbindung von Bausteinen. Nur wenn Sie mehr Details benötigen, führen Sie Ball-/Socket-Elemente ein.
- Falls mehrere Teams unabhängig voneinander an Bausteinen arbeiten, die eine gemeinsame Schnittstelle besitzen: Machen Sie diese Schnittstelle zu einem „first-order-citizen“ und investieren Sie in „exakte“ Darstellung. Verwenden Sie (mindestens) Ball-/Socket Notation.

Ergänzen Sie UML-Dokumentation an komplexen oder aufwendigen Schnittstellen immer um Beispielcode, der die Initialisierung und Benutzung der Schnittstelle genau aufzeigt.

4.10 Dokumentation technischer Konzepte

Technische Konzepte bilden einen Grundpfeiler der konzeptionellen Integrität: Sie zeigen die Lösung wiederkehrender Aufgaben und sorgen dafür, dass „gleiche Dinge gleich gelöst“[15] werden.

An dieser Stelle geht es um die Dokumentation, Beschreibung oder Darstellung solcher Konzepte. Dafür empfehle ich Ihnen folgende Gliederung (die Sie bei Bedarf auch leicht kürzen dürfen):

- **Aufgabenstellung, Anforderungen:** Skizzieren Sie die (wiederkehrende) Aufgabe möglichst genau, beschreiben Sie die Anforderungen an das Konzept (= die Lösung).
- Welche **Randbedingungen** und **Einschränkungen** muss das Konzept berücksichtigen?
- Lösungsansatz: Zeigen Sie Beispiele der Lösung, am besten mit Original-Quellcode („Referenzimplementierung“). Erklären Sie, falls nötig, zugehörige Strukturen und Abläufe.
- Geben Sie **Referenzen** an – denn gute Quellen zitieren ist weniger Arbeit, als dieselben Sachverhalte selbst zu dokumentieren.
- Weisen Sie ausdrücklich auf mögliche **Risiken** der Lösung hin, auf Kompromisse oder mögliche Nebenwirkungen.
- Schließlich können Sie **Alternativen** benennen und begründen, warum diese Alternativen nicht verwendet werden.

4.11 Werkzeuge zur Dokumentation

Egal, ob Sie visuell (= mit Diagrammen oder Bildern) oder verbal (= mit Worten) dokumentieren – für Erstellung, Pflege und Organisation sollten Sie Werkzeuge einsetzen.

Ich möchte Ihnen einige typische Kategorien vorstellen, anstatt für konkrete Vertreter (Schleich-)Werbung zu machen. Mehr Informationen finden Sie übrigens in [Zörner-12].

[15] Diese Formulierung stammt von Stefan Zörner aus [Zörner-12].

(UML) Modellierungswerkzeuge	
+ Effiziente Erstellung und Pflege von Diagrammen und Modellen + Gute Navigierbarkeit in Modellen	▪ schlecht geeignet für Text und Tabellen ▪ schlechte Integration in Code-Repositories ▪ oft unbrauchbare Reportgeneratoren
Textwerkzeuge	
+ universelle Verfügbarkeit + gute Integration von Text, Tabellen und Diagrammen	▪ schlechte Mehrbenutzerfähigkeit ▪ schwierig für große Dokumente ▪ keine Erstellung von Diagrammen
Wiki	
+ sehr gute Mehrbenutzerfähigkeit + gute Suche/Navigierbarkeit + (oft) Integration mit Issue-Tracker	▪ (kaum/keine) Offline-Fähigkeit ▪ Integration von Diagrammen (teilweise) aufwendig
Zeichenwerkzeuge	
+ optisch attraktive Resultate + hohe Flexibilität in Darstellung	▪ zeichnen statt modellieren ▪ (formal) exakte Modellierung nicht möglich

Mögliche Auswahlkriterien für Werkzeuge

Hier möchte ich Sie lediglich auf einige Ideen bringen, welche Kriterien für die Auswahl von Werkzeugen relevant sein könnten (bzw. in meinen Projekten einmal waren):

- Einfachheit der Benutzung: Erstellen, Ändern, Finden und Lesen, Volltextsuche, intuitive Benutzbarkeit ohne Handbuch
- Verfügbarkeit im Team: Nutzung ohne großen Aufwand, parallele Benutzung durch mehrere Personen gleichzeitig, für mehrere Betriebssysteme verfügbar
- Integration mit Code-Repository, Merge-Fähigkeit von Inhalten, Versionierung sowie Vergabe von Tags/Labels analog zum Quellcode
- Robustheit und Ausfallsicherheit, sichere Ablage sämtlicher Daten (und Metadaten)
- Rechte- und Rollenkonzept, Zugriffsschutz, Differenzierung von Benutzergruppen mit unterschiedlichen Schreib- und Leserechten auf bestimmte Teile der Dokumentation
- Automatisierbarkeit, Generierung benötigter Artefakte (etwa: pdf-Reports), Integration in Daily-Build, offene Datenformate
- Lizenz- und Betriebskosten, Lizenzmodell (shared- oder floating-Lizenz, Einzelplatz-Lizenz, Open-Source-Lizenz, Wartungs- oder Update-Kosten
- Offline-Fähigkeit: Nutzung auch ohne robuste Netzverbindung
- Review- oder Kommentierungsfähigkeit, Änderungs- oder Vorschlagsmodus
- Compliance zu Metamodell (etwa: UML-Metamodell)

Sie sehen schon – viele Faktoren könnten Ihre Werkzeugauswahl beeinflussen. Beginnen Sie „klein“ und lassen Sie niemals eine Dokumentation an Werkzeugen scheitern: Einfache Werkzeuge (Papier, Plain-Text) funktionieren auch!

4.12 TOGAF zur Architekturdokumentation

TOGAF („The Open Group Architecture Framework“) ist ein generisches und sehr flexibles Rahmenwerk (Framework) für die Entwicklung und das Management von Unternehmensarchitekturen. Es enthält eine Methode zur Architekturentwicklung („ADM“), die Aktivitäten und Prinzipien zur Entwicklung von Unternehmensarchitekturen umfasst. Außerdem definiert TOGAF folgende vier Typen von Architekturen:

- Business Architecture: definiert die Geschäftsstrategie, IT-Governance und die wesentlichen Geschäftsprozesse.
- Information (oder Data) Architecture: beschreibt die Struktur der logischen und physischen Unternehmensdaten sowie die Ressourcen zum Management dieser Daten.
- Application Architecture: stellt eine Vorlage zur Definition und zum Betrieb einzelner Anwendungssysteme bereit. Enthält das Mapping zwischen Anwendungen und den in der Business Architecture definierten Geschäftsprozessen.
- Technical Architecture: beschreibt die zum Betrieb der Anwendungen nötige Hard- und Softwareausstattung.

Der Entwicklungsprozess ADM adressiert grundsätzlich diese Architekturtypen, bedarf jedoch der organisationsspezifischen Anpassung. In Bild 4.15 sehen Sie einen Überblick (nach [OpenGroup 09]).

Wenn Sie sich jetzt immer noch nicht vorstellen können, wie Sie ein Stück Software mit TOGAF modellieren, stehen Sie nicht alleine da: Meiner Meinung nach eignet sich TOGAF aufgrund seiner größtenteils sehr abstrakten Konzepte kaum zur Modellierung konkreter Softwaresysteme, sondern spielt seine Stärken vielmehr bei der Entwicklung und Modellierung der Unternehmens-IT-Architekturen aus – von denen Sie in Kapitel 10 noch einiges lesen.

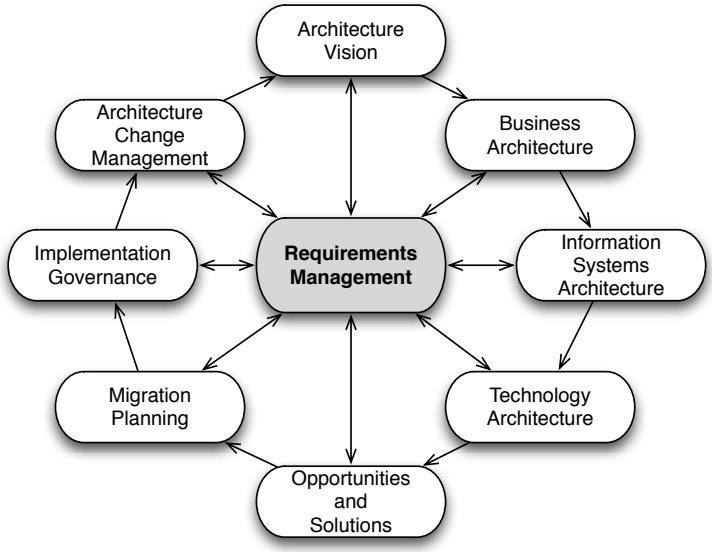


BILD 4.15
TOGAF Architecture Development Method

Als eindeutigen Pluspunkt sehe ich bei TOGAF eine generische, aber klare Begriffsbildung von Architekturen und (Enterprise-IT-)Architekturaufgaben.

Falls Sie jedoch für Ihr anstehendes oder laufendes Projekt eine methodische Unterstützung suchen, rate ich Ihnen von TOGAF ab: Die Aufwände zur Konfiguration von dessen Metamodell sowie die Umsetzung der zugehörigen Prozesse erfordert meines Erachtens mehr Zeit, als die meisten Entwicklungsprojekte aufbringen können – überlassen Sie das Ihrem IT-Vorstand oder Enterprise-IT-Architekt.

4.13 Weiterführende Literatur

[arc42] ist eine Sammlung von Architekturhilfsmitteln, Prozessbeschreibungen und Beispielen. Unter anderem finden Sie auf der Website ein Template für pragmatische Architekturdokumentation. arc42 ist für Software- und Systemarchitekten das, was [Volere] für Systemanalytiker und Requirements-Engineers ist.

[Clements+10] stellt einen mächtigen Ansatz zur Architekturdokumentation vor, der Möglichkeiten zur Definition eigener Sichten bietet. Leider geben die Autoren kaum Empfehlungen, welche Aspekte von Softwarearchitekturen in welcher Notation zu dokumentieren sind.

[Hargis+04] erläutern Grundprinzipien technischer Dokumentation. Geschrieben von erfahrenen technischen Redakteuren. Lesenswert, falls Sie viel dokumentieren müssen.

[Kruchten95] stellt ein Modell von fünf Sichten vor: Logische Sicht, Prozess-Sicht, Physische Sicht, Entwicklungssicht und Use-Case-Sicht. Der Ur-Vater von arc42 – leider hat Phillipe Kruchten die technischen Konzepte und Entwurfsentscheidungen in seiner Darstellung weggelassen.

[Zörner12] geht auf alle praktischen Belange der Dokumentation von Softwarearchitekturen ein. Wer Softwaresysteme dokumentieren muss, sollte dieses Buch kennen!

5 Modellierung für Softwarearchitekten

Mit Unterstützung von Peter Hruschka

Wenn die Sprache nicht stimmt,
ist das, was gesagt wird, nicht das, was gemeint ist.

Konfuzius

Fragen, die dieses Kapitel beantwortet:

- Was bedeutet „Modellierung“?
- Welche Möglichkeiten und Ausdrucksmittel gibt es für die Modellierung?
- Wie hilft UML bei der Modellierung?
- Welche Ausdrucksmittel stellt die UML für Softwarearchitekturen zur Verfügung?
- Welche Diagrammarten der UML sollten Sie für welche Architektursicht einsetzen?

5.1 Modelle als Arbeitsmittel

Modelle sind vereinfachte Darstellungen (Abstraktionen) der Realität. In komplexen Situationen oder Sachverhalten helfen Modelle, weil wir darin gezielt bestimmte Details der Realität aus- oder einblenden, um die Menge gezeigter Informationen zu kontrollieren.

In der Softwarearchitektur verwenden wir Modelle typischerweise in folgenden Situationen:

- Modellierung fachlicher Situationen (fachliche Klassen-, Prozess- oder Datenmodelle), um die inhaltliche Komplexität fachlicher Sachverhalte zu reduzieren und auf die für eine IT-Umsetzung relevanten Teile zu fokussieren.
- Modellierung statischer oder dynamischer Code-Strukturen (Paket-, Klassen- oder sonstiger Bausteindiagramme), um die Mengenkomplexität von Quellcode zu reduzieren und auch in großen Codebasen den Überblick zu behalten. Hierunter fällt auch die Modellierung von Interaktionen oder technischen Abläufen.

- Modellierung von Systemzusammenhängen (Komponenten-, Kontext- oder Verteilungsdiagramme), um Zusammenarbeit, Bezüge oder Abhängigkeiten zwischen Systemen (Hardware oder Software) zu zeigen.
- Modellierung als Basis für Codegenerierung: fachliche oder technische Situationen werden modellhaft dargestellt, um auf dieser Basis beispielsweise Quellcode, Datenstrukturen oder andere Artefakte per Generator zu erzeugen.
- Modellierung zur Vereinfachung, um andere Personen das Verständnis einer Situation zu erleichtern (= *Abholen von Stakeholdern*).

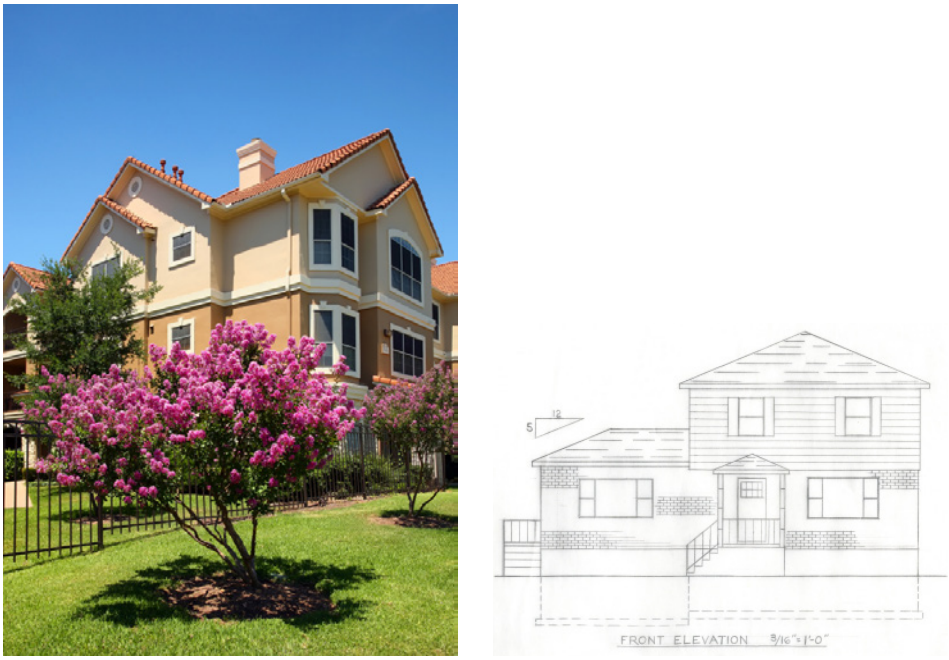


BILD 5.1 Realität und (ein mögliches) grafisches Modell[1]

Modelle unterstützen gezielte Sparsamkeit

Ich persönlich schätze Modelle, weil sie mir eine gezielte Sparsamkeit erlauben. Ich kann mich auf wesentliche Details einer aktuellen Situation konzentrieren und dabei genau diejenigen Informationen ver- und bearbeiten, die ich aktuell benötige.

Wenn Sie Modelle verwenden, müssen Sie zwei wesentliche Entscheidungen treffen:

- Welche Details lasse ich weg (= welche Teile der Realität sind Bestandteil meiner Modelle)?
- Wie stelle ich meine Modelle dar (grafisch oder textuell)?

Die erste dieser Fragen müssen Sie ganz allein beantworten (beziehungsweise erhalten in Kapitel 4 und Kapitel 5.2 einige Hinweise dazu) – für die zweite Frage habe ich im folgenden Abschnitt einige Anregungen für Sie gesammelt.

[1] Fotos lizenziert von *iStockPhoto.com*

5.1.1 Grafische oder textuelle Modellierung

Lange Zeit galten in der Informatik grafische Notationen als das Mittel der Wahl, etwa UML (siehe Kapitel 5.2), Entity-Relationship-Diagramme, Fluss- oder Aktivitätsdiagramme oder Petri-Netze. In den letzten Jahren haben textbasierte Modelle unter dem Stichwort „Domänenspezifische Sprachen“ (DSL, siehe [Völter-2013]) vermehrt an Bedeutung gewonnen.

Textuelle DSL als Mikro-Sprachen

Textbasierte Modelle finden sich beispielsweise in funktionalen Programmiersprachen oder Build-, Konfigurations- oder Installationsframeworks:

- Funktionale Sprachen wie Lisp, Haskell oder Clojure erleichtern es, ein Vokabular für eine bestimmte fachliche oder technische Situation als Funktionen (= funktionale Abstraktion) zu implementieren.
- Build-Systeme (wie ant[2] oder gradle[3]) stellen sprachliche Ausdrucksmittel für das Übersetzen, Bauen, Testen oder Verteilen von Sourcecode bereit.
- Installationsframeworks (wie homebrew[4] oder gvm[5]) besitzen „Vokabeln“ für das Installieren und De-Installieren bestimmter Software, deren Upgrade oder sogar für die Verwaltung mehrerer paralleler Versionen.

Textuelle Modelle von Software

Sie können beliebige Strukturen von Software (Komponente, Zusammenhänge, Abläufe) als Text beschreiben. Die Idee dieser textbasierten Beschreibung perfektioniert beispielsweise xtext[6], ein auf Basis des Eclipse-Ökosystems entstandenes Framework zur Entwicklung beliebiger Programmier- oder Domänensprachen.

Dieser Ansatz besitzt maximale Flexibilität. Allerdings müssen Sie vor dem eigentlichen Modellieren Ihres Sachverhaltes erst einmal Ihre Modellierungssprache definieren (oder Modellierung und Sprachentwicklung iterativ und agil verzahnen). In jedem Fall ist vor der eigentlichen Modellierung eine Vorarbeit notwendig, die Sie bei universellen Modellierungsmethoden oder -notationen nicht benötigen.

Textuelle Ansätze haben einige bestechende Vorteile:

- Text ist entwicklernah und wird mit denselben Werkzeugen (Editor, Versionsmanagement, diff-Tools) bearbeitet wie „normaler“ Quellcode.
- Spezifische DSL’s sind maximal flexibel – und können sehr exakt auf die jeweiligen Bedürfnisse abgestimmt und entwickelt werden.

Als starken Nachteil empfinde ich den oftmals hohen Initialaufwand und die technische Komplexität der DSL-Erzeugung. Insbesondere bei größeren Systemen fallen diese Aspekte auf lange Sicht kaum ins Gewicht.

---

2 Ant: Java-basiertes Buildsystem: *http://ant.apache.org*
3 gradle: Groovy-basiertes Buildsystem: *http://gradle.org*
4 Homebrew: Package-Installationsmanager für Mac-OS, *http://brew.sh*
5 Gvm: Installationsmanager für groovy-Frameworks, *http://gvmtool.net* (basiert auf dem Ruby-Pendant RVM)
6 *http://www.xtext.org*

Sowohl textbasierte wie auch grafische Ansätze habe ich in meiner Berufspraxis positiv wie negativ erlebt – daher möchte ich für keinen der beiden eine pauschale Empfehlung (oder Ablehnung) aussprechen.

- Klären Sie mit den beteiligten Personen die Anforderungen und Voraussetzungen, bevor Sie (oder Ihr Team) sich für eine Art der Modellierung entscheiden.
- Arbeiten Sie iterativ: Zeigen Sie anhand inhaltlich relevanter (Teil-) Aufgaben, dass und wie sich ein bestimmter Modellierungsansatz bei Ihnen verhält und „anfühlt".

Am Ende wird ein System niemals an der Wahl einer Modellierungsmethode scheitern, sondern höchstens daran, wie das Team Methoden und Werkzeuge der Praxis einsetzt.

5.2 UML 2 für Softwarearchitekten

In den folgenden Abschnitten möchten wir Ihnen pragmatische Vorschläge unterbreiten, wie Sie die Ausdrucksmittel von UML 2 zur Dokumentation von Softwarearchitekturen effektiv nutzen können. Das Kapitel baut auf dem Sichtenmodell aus dem vorigen Kapitel auf.

Warum sollten Sie UML nutzen?

Viele Ingenieurdisziplinen haben Standards zur besseren Kommunikation ihrer Ergebnisse geschaffen. Denken Sie nur an Gebäudearchitekten und ihre Baupläne, oder an Elektrotechniker und deren Schaltpläne. Auch Chemiker haben eine Formelsprache, um sich über Verbindungen auszutauschen. In der Software-Industrie gab es lange Zeit keine Standards zur Notation von Modellen.

Seit die Object Management Group (OMG) 1997 die Unified Modeling Language (UML) standardisiert hat, haben auch die Software-Ingenieure ein weltweit anerkanntes Mittel zur Dokumentation von Analyse- und Architekturmodellen. Seither wurde der Standard immer wieder in Details angepasst. Die praktisch relevanten Ausdrucksmittel sind stabil geblieben.

Die UML 2 stellt mehr als ein Dutzend unterschiedlicher Diagramm- und Darstellungsarten bereit, alle durch ein klares Metamodell präzise definiert. Das wirkt auf den ersten Blick sehr kompliziert und teilweise abschreckend – wie bei vielen mächtigen Werkzeugen.

Hier möchten wir Ihnen eine pragmatische Auswahl vorstellen, mit deren Hilfe Sie wesentliche Architektursichten effektiv modellieren und kommunizieren können.

UML ist als Standard für grafische Modelle industrieweit akzeptiert. Die Unterstützung durch eine Vielzahl praxiserprobter Modellierungswerkzeuge sichert diesen Status. Sie gehen daher für die nächsten Jahre kein Risiko ein, wenn Sie sich heute für UML als Notation entscheiden.

5.2.1 Die Diagrammarten der UML 2

Bevor wir Ihnen die Anwendung der UML 2 zur Darstellung von Softwarearchitekturen erläutern, möchten wir in Tabelle 5.1 einen Überblick über die verschiedenen Diagrammarten der UML 2 geben.

TABELLE 5.1 Diagrammarten der UML 2

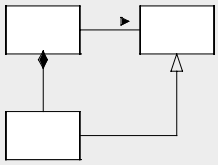
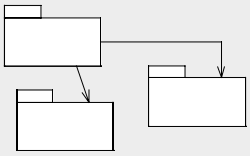
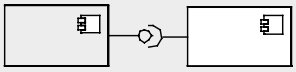
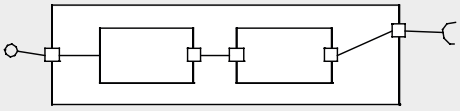
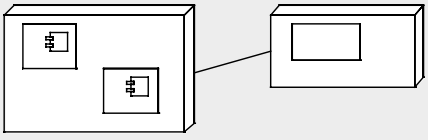
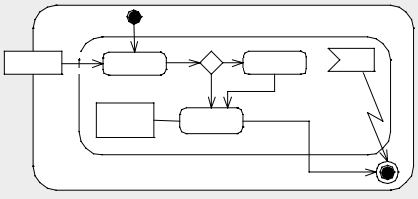
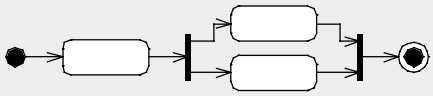
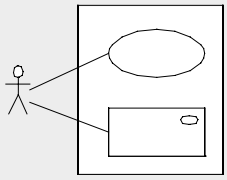
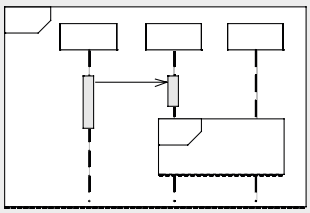
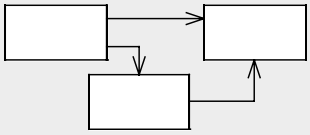
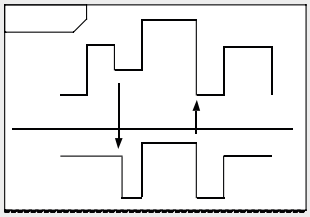
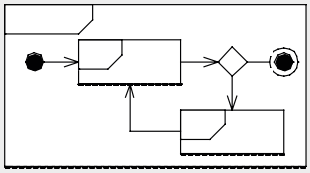
Diagrammart	Verwendung
Strukturdiagramme	
	**Klassendiagramm:** Zeigt die statische Struktur von Klassen und deren Beziehungen (Assoziationen, Aggregationen sowie Spezialisierungen/Generalisierungen).
	**Objektdiagramm:** Zeigt eine Struktur von Instanzen sowie deren Verbindungen; stellt insofern einen Schnappschuss des Klassendiagramms dar.
	**Paketdiagramm:** Gibt einen Überblick über die Zerlegung des Gesamtsystems in Pakete oder Teilsysteme. Enthält logische Zusammenfassungen von Systemteilen.
	**Komponentendiagramm:** Zeigt, wie Systembausteine zu Komponenten mit definierten Ein- und Ausgabeschnittstellen zusammengefasst werden.
	**Kompositionsstrukturdiagramm:** Zeigt den inneren Aufbau eines Systembausteins (Komponente, Klasse, Paket) sowie die Abbildung äußerer Schnittstellen auf innere.
	**Verteilungsdiagramm** (*Deployment-Diagram*): Zeigt die Infrastruktur und ihre Abhängigkeiten sowie die Verteilung der Laufzeitelemente auf die technische Infrastruktur (Hardware).
Verhaltensdiagramme	
	**Aktivitätsdiagramm:** Zeigt mögliche Abläufe innerhalb von Systembestandteilen (etwa: Klassen, Komponenten oder Use-Cases). Kann Algorithmen, Daten- und Kontrollflüsse sehr detailliert beschreiben.

TABELLE 5.1 (*Fortsetzung*) Diagrammarten der UML 2

Diagrammart	Verwendung
	**Zustandsdiagramm** (*State Diagram*): Zeigt die möglichen Zustände eines Systembestandteils sowie die erlaubten Zustandsübergänge an und verknüpft Aktivitäten mit diesen Zuständen und Übergängen.
	**Anwendungsfalldiagramm** (*Use-Case-Diagram*): Zeigt eine Übersicht über alle Prozesse, mit denen das System auf Wünsche von Akteuren (oder Nachbarsystemen) reagiert.
Interaktionsdiagramme	
	**Sequenzdiagramm:** Zeigt den Nachrichtenaustausch zwischen Instanzen von Systembestandteilen für einzelne Szenarien. Seit UML 2 schachtelbar.
	**Kommunikationsdiagramm:** In früheren UML Versionen Kollaborationsdiagramm genannt, zeigt die Zusammenarbeit zwischen Instanzen von Systembestandteilen.
	**Zeitverhaltensdiagramm** (*Timing-Diagram*): Zeigt Zeitabhängigkeiten zwischen Ereignissen beschreibt Zustandsänderungen in Abhängigkeit vom Zeitverlauf.
	**Interaktionsübersichtsdiagramm:** Zeigt das Zusammenspiel verschiedener Interaktionen und besteht in der Regel aus Referenzen auf andere Interaktionsdiagramme. Ein Art Aktivitätsdiagramm auf hoher Abstraktionsebene.

Einen Überblick über die UML 2-Notation finden Sie in [arc42] und [Rupp+12].

5.2.2 Die Bausteine von Architekturen

Die kleinsten Bausteine objektorientierter Softwarearchitekturen sind Klassen bzw. zur Laufzeit daraus instanziierte Objekte. UML stellt diese als Rechtecke dar, in die Sie neben dem Namen Attribute und Operationen eintragen können. Generell gilt: Ein einzelner Name deutet auf eine Klasse hin. Wann immer Sie im Namensfeld eine Unterstreichung und einen Doppelpunkt entdecken, handelt es sich um konkrete Objekte (Instanzen).

Sobald Ihre Systeme etwas umfangreicher werden, benötigen Sie neben Klassen noch Möglichkeiten zur Modellierung größerer Einheiten. Sie müssen aus kleineren Bausteinen größere Bausteine zusammenstellen. Zwei Beweggründe treiben Sie dazu: Sie wollen abstrahieren und, Sie wollen Sichtbarkeit und Zugriff regeln.

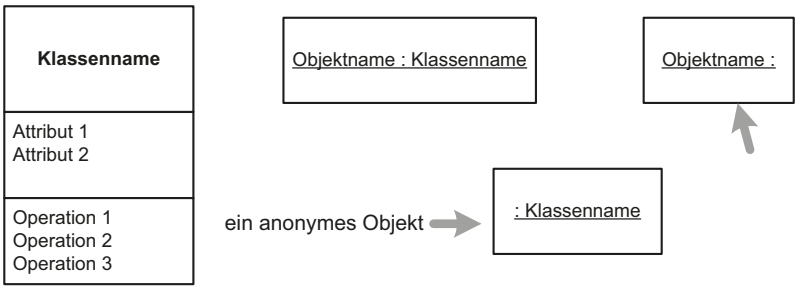


BILD 5.2 Klassen und Objekte in der UML

Dafür stellt die UML Pakete und Komponenten zur Verfügung. Etwas vereinfacht gelten folgende Regeln zur Unterscheidung:

1. Wenn Sie nur einen Namen für eine Menge von (kleineren und größeren) Bausteinen brauchen – also nur einen abstrakteren Begriff für eine Gruppe von z. B. acht kleineren Bausteinen –, dann sollten Sie Pakete verwenden.
2. Wenn Sie nicht nur abstrahieren wollen, sondern gleichzeitig auch Zugriffsschutz über sauber definierte Schnittstellen gewährleisten wollen, dann sind Komponenten das Ausdrucksmittel Ihrer Wahl.

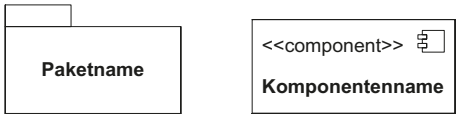


BILD 5.3
Größere Bausteine: Pakete und Komponenten

Vereinfacht ausgedrückt, sind Pakete nur Sammeltöpfe für kleine Bausteine. Komponenten hingegen sind eine Art von großen Klassen: sie können über eigene Attribute verfügen (aber wer will schon globale Variable?), können Beziehungen eingehen, haben nach außen sichtbare und nach innen versteckte Teile.

Pakete und Komponenten dürfen beliebig geschachtelt werden. Somit lassen sich auch sehr komplexe Systeme mit Tausenden von Klassen durch drei bis vier Abstraktionsebenen von Paketen oder Komponenten überschaubar gestalten.

5.2.3 Schnittstellen

Die Art der Schnittstellendarstellung unterscheidet sich für die unterschiedlichen Bausteine von Architekturen.

Die Schnittstellen der kleinsten funktionalen Bausteine, der Operationen, sind deren Ein- und Ausgabeparameter. Dafür lässt die UML die Syntax offen. Üblicherweise nutzt man ähnliche Schreibweisen wie in der Programmiersprache Ihrer Wahl.

Die Schnittstellen einer Klasse bilden im einfachsten Fall die Operationen, die die Klasse öffentlich („public“) macht (im Gegensatz zu den privaten Operationen, die von außen nicht aufgerufen werden können). Die UML verwendet das Zeichen „+“ vor dem Operationsnamen für public und „–“ für private (vgl. Bild 5.4).

Interessant wird die Darstellung der Schnittstellen von Komponenten. Bei diesen sind für uns als Architekt zwei Aspekte wichtig:

- Welche Services bietet die Komponente anderen Komponenten an (die Export-Schnittstelle, „provided interface“)?
- Was benötigt die Komponente von anderen (die Import-Schnittstelle, „required interface“)?

Die UML nutzt dafür entweder die Ball- und abgekürzte Socketnotation (vgl. Bild 5.4) oder verbindet die Komponente explizit mit Interfaceklassen. Betrachten Sie eine Interface-Klasse als eine degenerierte Klasse, die neben dem Namen (und der Kennzeichnung mit dem Stereotyp <<interface>>) nur die Signaturen von Operationen enthält, jedoch keine „Bodies“ und keine Attribute. Der Pfeil zu Schnittstellen, die der Baustein anbietet (d. h. implementiert), wird – nicht ganz zufällig – ebenfalls mit dem hohlen Dreieck notiert, wie die Generalisierung/Spezialisierung von Klassen (jedoch mit gestrichelter Linie). Die Komponente wird als Spezialisierung der Schnittstelle verstanden. Schnittstellen, die benötigt werden, sind über den gestrichelten Abhängigkeitspfeil mit dem Stereotyp <<use>> verbunden.

Die UML 2 hat noch ein weiteres Ausdrucksmittel für Schnittstellen eingeführt: Ports, dargestellt als kleines Rechteck im Rahmen der Komponente. Ein Port ist nur eine abkürzende Schreibweise für eine benannte Gruppe von beliebig vielen Export- und Importschnittstellen – und bringt somit keine konzeptionellen Neuigkeiten. Um zu wissen, was an dem Port

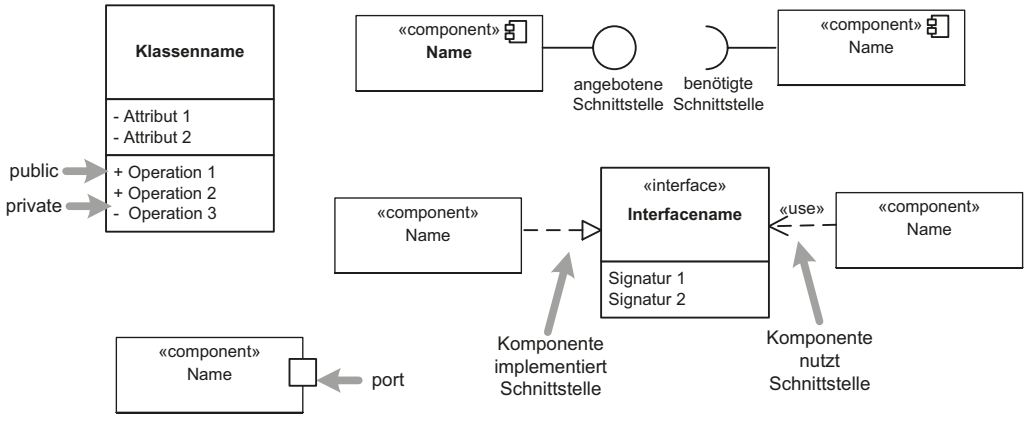


BILD 5.4 Verschiedene Arten der Schnittstellendarstellung

genau angeboten oder erwartet wird, müssen Sie diesen ausführlich beschreiben oder mit Ball- und Socketnotation explizit modellieren.

5.2.4 Die Bausteinsicht

Nach diesen Grundbegriffen zu Bausteinen und deren Schnittstellen können wir nun die UML-Modelle betrachten, mit denen Sie die Bausteinsicht darstellen können. Bild 5.5 zeigt ein UML-Klassendiagramm. Darin werden die Klassen statisch mit einfachen Assoziationen, Teile-Ganzes-Beziehungen („besteht aus“) oder Generalisierung/Spezialisierung („ist ein“) in Zusammenhang gebracht.

Der Zusammenhang zwischen den größeren Bausteinen wird in Form von Paket- oder Komponentendiagrammen hergestellt. Die wesentliche Beziehung zwischen diesen ist die einfache Abhängigkeit („dependency“), die durch Stereotypes wie <<access>>, <<import>> oder andere präzisiert werden kann. Je nach Reifegrad Ihrer Architekturfestlegungen können dabei direkte Abhängigkeiten oder Abhängigkeiten über Schnittstellen eingezeichnet werden.

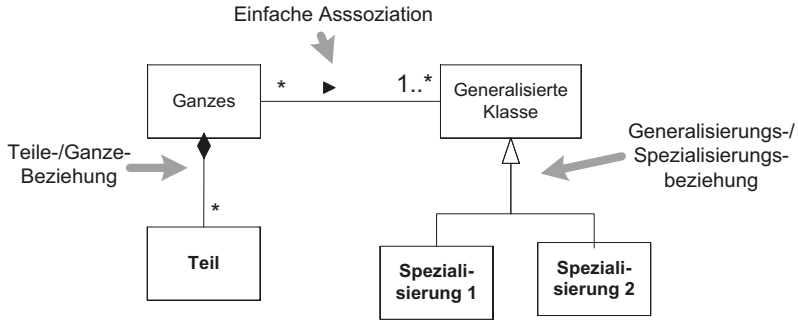


BILD 5.5 Klassendiagramm

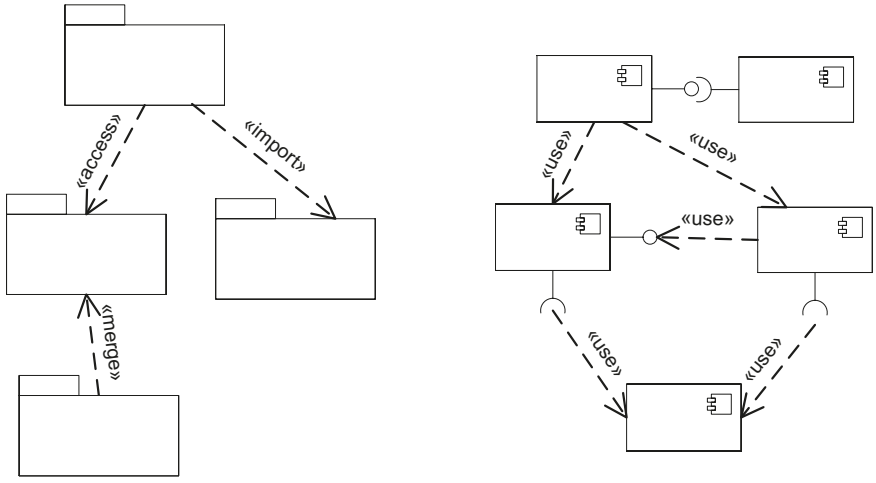


BILD 5.6 Paket- und Komponentendiagramm

Hier noch einige Tipps zum Erstellen von Klassen-, Paket- und Komponentendiagrammen.

- Denken Sie sorgfältig über gute Namen für Klassen und Komponenten nach.
- Nutzen Sie in Klassendiagrammen die Stärke der UML, Assoziationen mit Namen versehen zu können, Rollennamen anzugeben, die mögliche Navigationsrichtung vorzuschreiben und Multiplizitäten an beiden Enden festzulegen. Gepaart mit guten Namen werden die Diagramme semantisch aussagekräftiger und sparen das Nachschlagen in Hintergrunddokumenten.
- Nutzen Sie Stereotypes für verschiedene Arten von fachlichen und technischen Klassen und Komponenten, z. B. <<entity>> für datenorientierte, <<control>> für steuerungsorientierte Bausteine und <<viewY>> für Sichtenbildung.

Die bisher erwähnten Diagramme der Bausteinsicht stellen statische Aspekte in den Vordergrund. Zwei Diagramme für dynamische Aspekte ergänzen die Bausteinsicht: Aktivitätsdiagramme und Zustandsdiagramme. In beiden wird unabhängig von Laufzeitaspekten das allgemein gültige Verhalten von Bausteinen beschrieben. Wie Bild 5.7 zeigt, lassen sich in Aktivitätsdiagrammen mit Bahnen (engl. Swimlanes) einzelne Aktivitäten den Bausteinen zuordnen. In Zustandsdiagrammen modellieren Sie die für jede Instanziierung eines Bausteins prinzipiell vorhandenen Zustände und die Übergänge auslösenden Ereignisse. Die Operationen können dabei entweder den Übergängen oder den Zuständen zugeordnet werden.

- Beim Reengineering von Architekturen können Sie Aktivitätsdiagramme mit Bahnen benutzen, um aufzuzeigen, welche Funktionen derzeit welchem Baustein zugeordnet sind. Zeigen Sie so z. B. die Aufteilung zwischen Client und Server, zwischen verschiedenen Standorten oder Rechnern oder zwischen verschiedenen Abteilungen.
- Nutzen Sie Bahnen in Aktivitätsdiagrammen beim Neuentwurf gezielt, um Funktionen in die Verantwortung größerer Bausteine zu übergeben.
- Wenn es Ihnen schwerfällt, Funktionsreihenfolgen in Aktivitätsdiagrammen linear zu modellieren, dann verbergen sich dahinter oft Zustände. Nutzen Sie in diesem Fall Zustandsmodelle, um Funktionen in Abhängigkeit von externen, internen oder Zeitereignissen zu modellieren.
- Zustandsmodelle eignen sich hervorragend, um Lebenszyklen komplexer Klassen zu modellieren (von der Geburt nach Ausführung des Konstruktors über alle Ereignisse, die Objekten der Klassen danach zustoßen können, bis hin zur Zerstörung im Destruktor).
- Verwenden Sie Zustandsmodelle auch zur Modellierung des Verhaltens komplexer, asynchron operierender Subsysteme. Ein Beispiel dafür sind Echtzeitsysteme, die auf unabhängige äußere Ereignisse reagieren müssen; ein anderes Beispiel sind Benutzungsoberflächen, die kontextabhängig bei Benutzereingaben Menüs wechseln und Tastenbelegungen aktualisieren müssen.

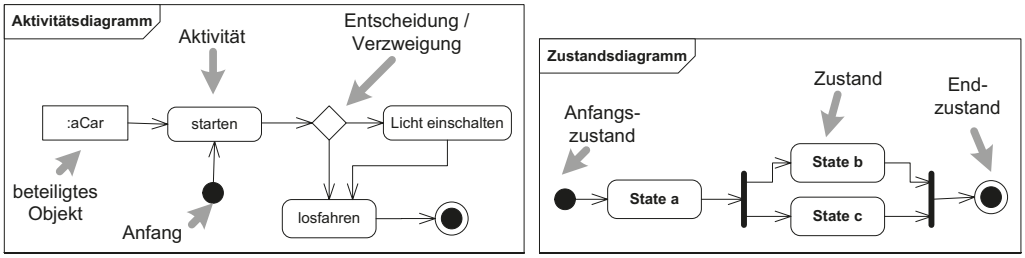


BILD 5.7 Aktivitäts- und Zustandsdiagramme

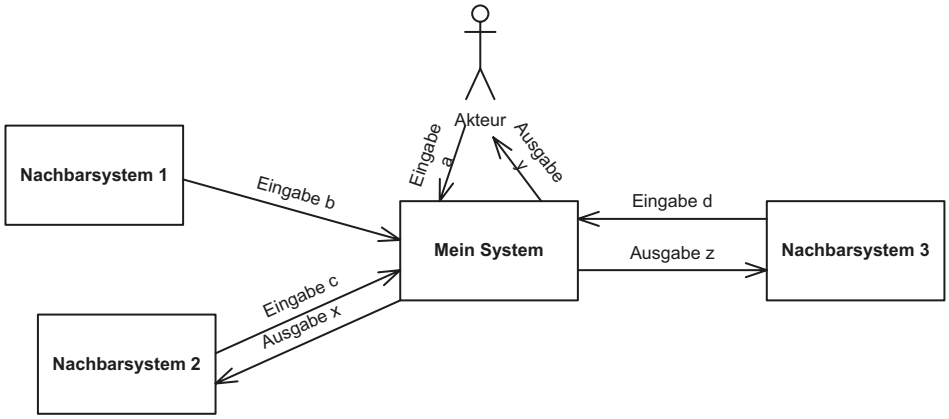


BILD 5.8 Klassendiagramm zur Darstellung des statischen Kontexts

Nutzen Sie Klassen- oder Komponentendiagramme auch für die Modellierung der statischen Kontextsicht. Zeichnen Sie ein Diagramm mit Ihrem System in der Mitte als Blackbox, und ordnen Sie rundherum alle Quellen an, von denen Ihr System Informationen erhält, und alle Senken, an die Ihr System Informationen liefert. Die UML hat dieses bewährte Kontextdiagramm nicht als eigenständiges Diagramm aufgenommen, doch erfüllt das Klassendiagramm diesen Zweck, wenn Sie sich die Freiheit nehmen, die Assoziationen zwischen System und Systemumgebung gerichtet zu zeichnen und sie mit den Ein- und Ausgabenamen zu beschriften (vgl. Bild 5.8).

5.2.5 Die Verteilungssicht

UML stellt eine Diagrammart zur Verfügung, um Infrastruktur unserer Softwarearchitektur zu modellieren und deren Bausteine dieser Infrastruktur zuzuordnen: das Verteilungsdiagramm (engl. deployment diagram).

Die Hauptelemente dieser Verteilungsdiagramme sind Knoten und Kanäle zwischen den Knoten. Stellen Sie sich Knoten als Standorte, als Cluster, als Rechner, als Chips oder als Devices vor – kurz: alles, was sich etwas merken (d. h. Daten persistent festhalten) und etwas tun (d. h. Operationen ausführen) kann. Die Kanäle, die die Knoten verbinden, repräsentieren die

physikalischen Übertragungswege, auf denen Informationen zwischen Knoten ausgetauscht werden können – also alles von Kabeln über Rohrpost, Rauchzeichen bis hin zu Bluetooth und anderen drahtlosen Verbindungen. Die UML behandelt diese Infrastruktur etwas stiefmütterlich. Für verteilte Systeme sind diese Diagramme jedoch unabdingbar.

Knoten sind das einzige dreidimensionale Symbol in der UML, um die Prozessoren und Devices besser ausdrücken zu können. Die Kanäle sind als normale Assoziationen (wie im Klassendiagramm) dargestellt. Knoten sind schachtelbar, um z. B. die Prozessoren eines Clusters oder die verschiedenen Platinen in einem großen Gehäuse darzustellen.

Eigenschaften von Knoten wie der Rechnertyp oder die Speichergröße können über Stereotypen (<<Security Server>>) oder Tagged Values ({memory = 40 Gbyte}) in den Diagrammen festgehalten werden.

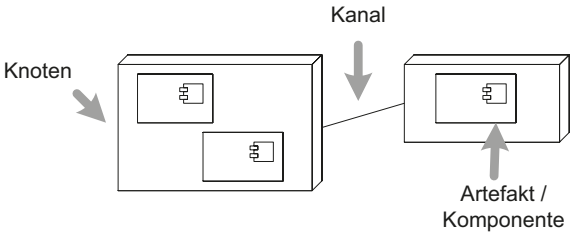


BILD 5.9
Knoten, Kanäle und verteilte Artefakte

- Modellieren Sie die Infrastruktur nur so weit, wie Sie es für die Diskussion der Verteilung der Software benötigen. Beschriften Sie auch die Kanäle sparsam, falls Sie z. B. Datenwege erläutern möchten.
- Ordnen Sie die Knoten im Bild so an, wie sie auch in der realen Welt angeordnet sind: bei geographisch verteilten Systemen z. B. Hamburg eher oben im Bild und Freiburg eher links unten; bei Prozessoren und Speichereinheiten in einem Gehäuse so, wie Sie diese Teile nach dem Aufschrauben des Gehäuses auch sehen können. Dies erleichtert die Wiedererkennung und verbessert die Lesbarkeit der Diagramme.

Der Hauptzweck der Verteilungsdiagramme ist die Zuordnung von Bausteinen zu Knoten.

Zeichnen Sie auch für die Verteilungssicht ein physikalisches Kontextdiagramm. Dazu setzen Sie den Knoten, auf dem Ihr System läuft, in die Mitte und ordnen – ähnlich wie beim logischen Kontextdiagramm – alle menschlichen Akteure und Nachbarprozessoren rundherum an. Legen Sie jetzt jedoch die physikalischen Schnittstellen fest, die zu den Nachbarsystemen vorhanden sind, z. B. Thin-Wire-Ethernet als Verbindung, das Druckerkabel, Bus-Systeme, Bluetooth etc. (vgl. Bild 5.10). Dieses Bild hilft Ihnen, die Abbildung von logischen Ein- und Ausgaben auf entsprechende Medien herzustellen.

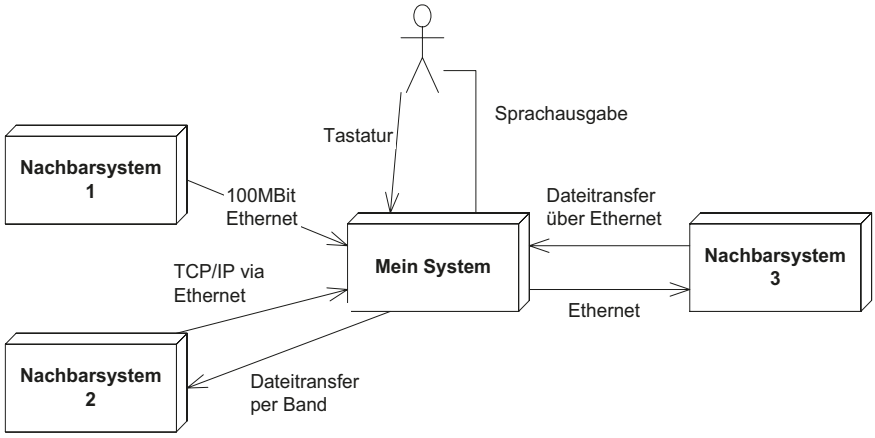


BILD 5.10 Physikalisches Kontextdiagramm: die Kanäle zu Ihrem System

5.2.6 Die Laufzeitsicht

Oftmals interessiert Sie nicht nur die prinzipielle Struktur der Bausteine Ihrer Architektur, sondern gezielte Aussagen, was zur Laufzeit Ihres Systems passieren soll. Auch dafür stellt die UML eine Menge von Diagrammen zur Verfügung. Betrachten wir zunächst die Elemente der Laufzeitsicht. Dabei handelt es sich immer um Instanzen von Bausteinen, die in der Bausteinsicht enthalten sind, also um Objekte zu den Klassen oder um instanziierte Komponenten.

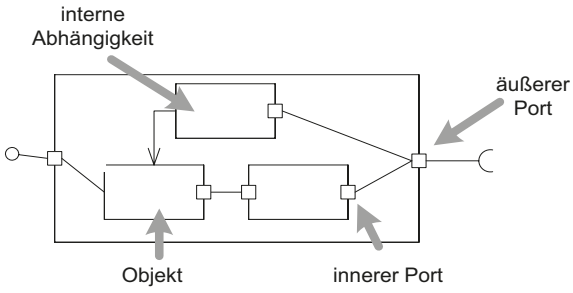
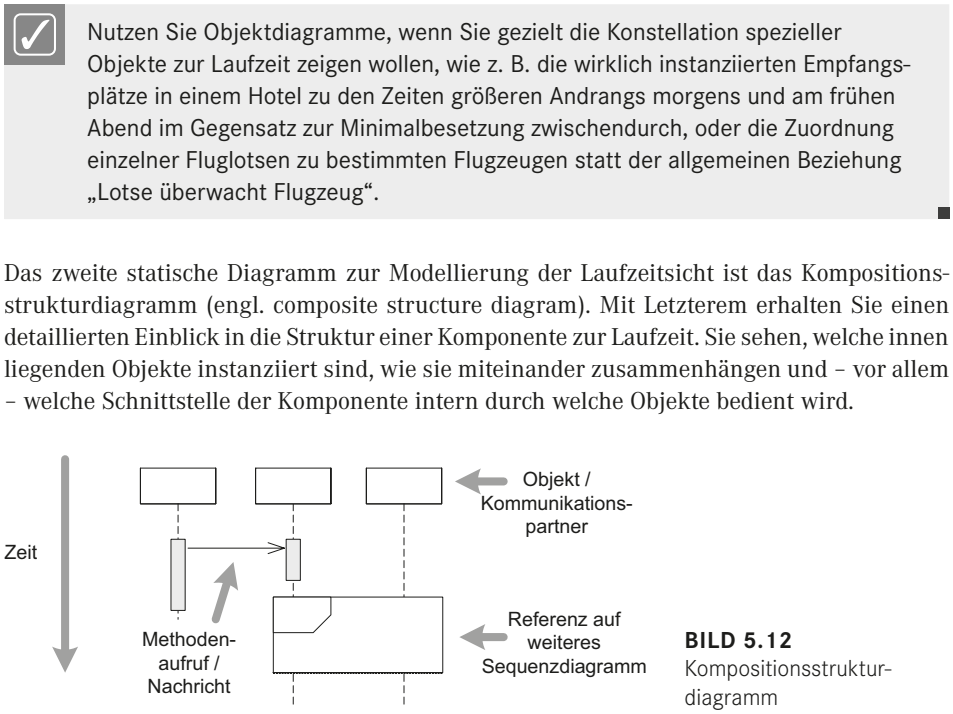


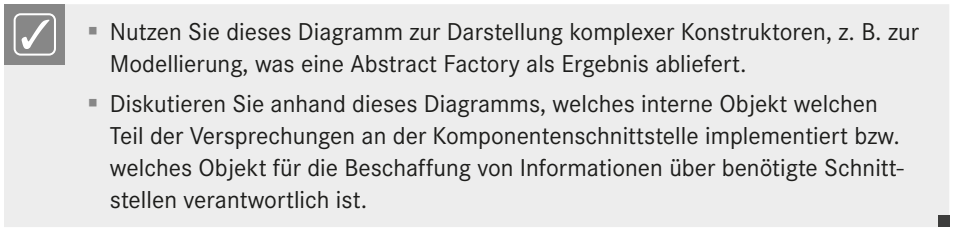
BILD 5.11
Objektdiagramm

Mit Objektdiagrammen können Sie Schnappschüsse der existierenden Laufzeitobjekte darstellen und auch instanziierte Beziehungen. Die UML bietet dabei die Möglichkeit, zwischen aktiven und passiven Objekten zu unterscheiden. Letztere werden, wie der Name schon ausdrückt, nicht selbst aktiv, sondern warten darauf, benutzt zu werden. Alle aktiven Bausteine (gekennzeichnet mit doppelter Seitenlinie in der Grafik) sind unabhängig voneinander selbstständig aktiv und führen Prozesse gemäß ihrer Spezifikation aus.



Das zweite statische Diagramm zur Modellierung der Laufzeitsicht ist das Kompositionsstrukturdiagramm (engl. composite structure diagram). Mit Letzterem erhalten Sie einen detaillierten Einblick in die Struktur einer Komponente zur Laufzeit. Sie sehen, welche innen liegenden Objekte instanziiert sind, wie sie miteinander zusammenhängen und – vor allem – welche Schnittstelle der Komponente intern durch welche Objekte bedient wird.

BILD 5.12 Kompositionsstrukturdiagramm



Mehr noch als die beiden bisher erwähnten statischen Modelle werden Sie zur Darstellung der Laufzeitsicht dynamische Modelle nutzen. Die UML verwendet für diese Diagramme den Sammelbegriff „Interaktionsdiagramme“, weil alle davon die Zusammenarbeit von Objekten zur Laufzeit beschreiben, d. h. den Austausch von Nachrichten, die gegenseitigen Aufrufe bzw. das Senden und Empfangen von Ereignissen.

Das populärste Diagramm ist das Sequenzdiagramm. Es stellt beispielhafte Abläufe (Szenarien) durch das System dar. Wie Bild 5.13 zeigt, sind die am Szenario beteiligten Objekte durch ein Rechteck mit daran hängender Lebenslinie gekennzeichnet. Von oben nach unten sehen Sie die Ereignisse und Nachrichten, die von einem Objekt zu einem anderen geschickt werden.

UML 2 hat die Sequenzdiagramme um zahllose Konstrukte erweitert, wie Abfragen, Schleifen, Parallelität, Schachtelung etc. Sie können auch die Art der Kommunikation durch unterschiedliche Pfeile grafisch darstellen (synchron mit und ohne Antwort, asynchrone Nachrichten etc.) Tasten Sie sich langsam an die Notationselemente heran, die Ihnen für die Kommunikation zwischen Systemteilen wichtig sind.

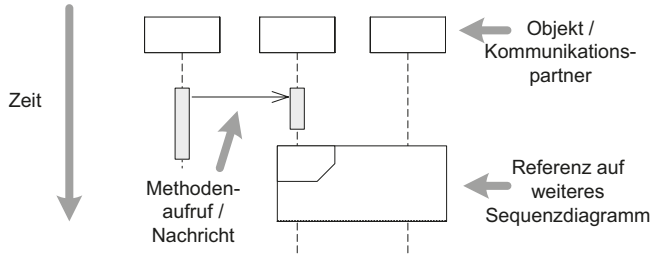


BILD 5.13
Sequenzdiagramm

- Nutzen Sie Sequenzdiagramme, um „Kandidatenobjekte“ zu finden, d. h. Bausteine, die für eine bestimmte Aufgabe die Verantwortung übernehmen können. So helfen Diagramme der Laufzeitsicht, die Elemente der Bausteinsicht zu identifizieren.
- Nutzen Sie Szenarien auch, um Ihre Architekturen zu überprüfen. Zeichnen Sie typische, offensichtliche Abläufe zwischen den Instanzen und
- Prüfen Sie, ob ihre Klassen und Komponenten entsprechende Operationen bereitstellen.
- Nutzen Sie Sequenzdiagramme auf verschiedenen Abstraktionsebenen. Die gröbste Abstraktion ist ein Laufzeitkontextdiagramm (siehe Bild 5.14). Sie können sich auch ausschließlich auf die Kommunikation zwischen Rechnern konzentrieren, indem Sie je ein Stellvertreterobjekt für jeden Rechner zeichnen und die Kommunikation zwischen diesen. Unterdrücken Sie dabei alle Nachrichten, die innerhalb eines Rechners ausgetauscht werden. Modellieren Sie dann auf einer nächstfeineren Ebene z. B. die Kommunikation zwischen Anwendungen, die auf einem Rechner laufen, und erst auf der feinsten Ebene die Kommunikation zwischen den primitiven Objekten innerhalb einer Komponente.

Auch in der Laufzeitsicht lohnt es sich, ein Kontextdiagramm zu zeichnen. Wiederum ist Ihr System dabei ein einziges Black-Box-Objekt. Sie zeigen in diesem Laufzeitkontextdiagramm beispielhafte Kommunikation des Gesamtsystems mit Akteuren und anderen Nachbarsystemen.

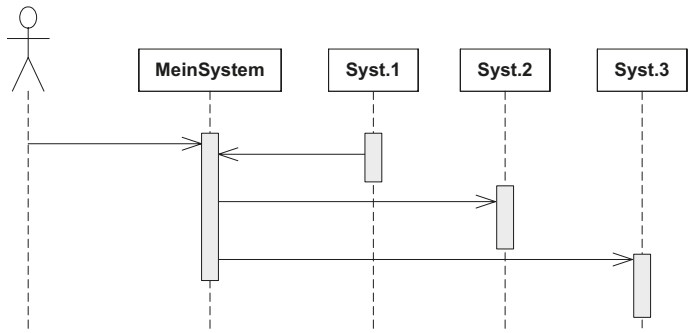


BILD 5.14
Laufzeitkontext-diagramm

Nutzen Sie diese Kontextsicht für die frühzeitige Klärung der heikelsten Schnittstellen Ihres Systems: der Schnittstellen nach außen!

Während Sequenzdiagramme die Abläufe „sequenziell“ von oben nach unten lesbar darstellen, können Sie mit Kommunikationsdiagrammen eher die Topologie Ihrer Klassen- und Komponentendiagramme aufgreifen. Stellen Sie sich ein Kommunikationsdiagramm als ein instanziiertes Klassendiagramm dar, in das Sie jetzt (natürlich nummeriert) die Folge von Nachrichten einzeichnen, die zwischen den Beteiligten ausgetauscht werden.

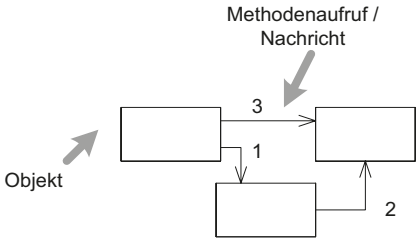


BILD 5.15
Kommunikationsdiagramm

- Nutzen Sie Kommunikationsdiagramme, wenn Sie schon Klassen und Komponenten gefunden haben und nun über deren beispielhafte Zusammenarbeit diskutieren wollen.
- Modellieren Sie ein Szenario für den Normalfall und dann z. B. Szenarien für heikle Sonderfälle. Testen Sie somit, ob die Klassen schon über geeignete Operationen verfügen, um den Gesamtablauf zu bewerkstelligen.

Da Interaktionen bei großen Systemen manchmal sehr komplex werden können, hat die UML 2 noch ein weiteres, sehr nützliches Diagramm bereitgestellt: das Interaktionsüberblicksdiagramm. Es bringt keine wesentlichen neuen Modellierungselemente ins Spiel. Stellen Sie sich das Diagramm als eine Art grafisches Inhaltsverzeichnis oder einen Übersichtsplan für viele einzelne Interaktionsdiagramme vor. Auf sehr hoher Ebene zeichnen Sie die Abhängigkeiten zwischen Interaktionsdiagrammen. Für genaueres Wissen zoomen Sie dann jeweils in eines der Diagramme hinein.

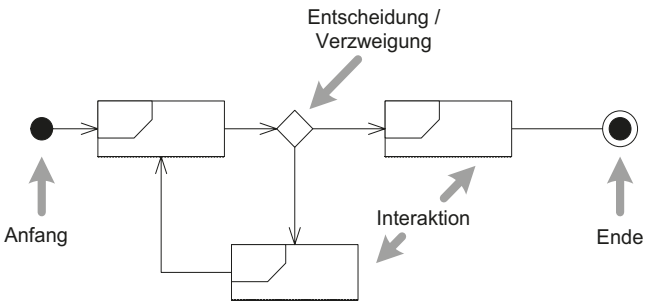


BILD 5.16
Interaktionsüberblicks-
diagramm

Mit den Ausdrucksmitteln aus Aktivitätsdiagrammen (Verzweigung, Parallelität, Verfeinerung, ...) können Sie in diesem Überblicksdiagramm alle möglichen Fragmente von Interaktionen kombinieren, die dann jeweils in eigenständigen Sequenz- oder Kommunikationsdiagrammen weiter ausgeführt sind.

- Nutzen Sie die Stärke von Szenarien. Während die Methoden der 80er-Jahre hauptsächlich auf die Bausteinsicht mit Strukturen und generalisierten Verhaltensmodellen Wert gelegt haben, bietet die UML jetzt zusätzlich die Aussagekraft der Beispiele an. Interaktionsdiagramme sind grundsätzlich beispielhafte Abläufe. Ein gutes Beispiel ist instruktiver als eine schlechte Abstraktion!
- Wählen Sie die Beispielabläufe gezielt aus (so wie Sie Testfälle auswählen): ein Beispiel für den Normalfall, einige Beispiele für Grenzfälle und seltene Ausnahmen, vielleicht auch ein Gegenbeispiel, um die Robustheit des Systems zu testen.

5.2.7 Darum UML

So hilfreich es auch ist, gemeinsam an einer Wandtafel zu stehen und Ideen über die Gestaltung von Architekturen durch informelle Diagramme auszutauschen – die Zusammenarbeit wird noch effektiver, wenn jeder bezüglich der dargestellten Kästen und Verbindungen das Gleiche versteht.

Die UML hat Kästchen und Striche für uns standardisiert. In den statischen Modellen lassen sich die Bausteine der Architektur auf verschiedenen Abstraktionsebenen (vom Gesamtsystem als Black Box über Subsysteme, Komponenten und Pakete bis zu den kleinsten Klassen) modellieren und miteinander in Beziehung setzen.

In den dynamischen Modellen zeigen Sie, wie Instanzen dieser Bausteine, d. h. Objekte, aktive Klassen, Prozesse und Threads, zur Laufzeit zusammenarbeiten und miteinander kommunizieren.

Verteilungsdiagramme erlauben das Mapping (von Artefakten) der Bausteine auf die vorhandene Infrastruktur.

Somit stellt Ihnen UML genügend grafische Hilfsmittel zur Verfügung, um alle Aspekte einer Softwarearchitektur transparent zu machen – und zwar in einer Weise, die weltweit verstanden wird.

5.2.8 Darum nicht UML

So hilfreich eine standardisierte Notation ist – die Mächtigkeit von UML kann auch abschreckend wirken. In Entwicklungsteams habe ich bezüglich UML oft heftige Akzeptanzprobleme erlebt, ausgelöst durch überzogenen Formalismus oder komplexe UML-Werkzeuge.

Fokussieren Sie auf Inhalten, bevor Sie viel Zeit in UML-Missionierung investieren. Auch in eher formlosen Notationen können Sie signifikante und relevante Inhalte vermitteln (= visuell kommunizieren). Zum Diskutieren und gemeinsamen Brainstormen eignen sich Skizzen am Whiteboard oft besser als formale Modelle. Bei Bedarf können Sie solche Skizzen zu einem späteren Zeitpunkt in (UML-)Werkzeuge überführen.

5.3 Tipps zur Modellierung

Let's keep the modeling baby but throw out the bureaucracy bathwater.

Scott Ambler,
in [Ambler-12]

- Modellieren-im-Team: Skizzieren Sie erste Versionen im Team am Whiteboard. Verzichten Sie (vorläufig) auf komplexe Grafik- oder UML-Werkzeuge. Solche Zusammenarbeit verbessert das gemeinsame Verständnis der jeweiligen Systemteile erheblich. Nennen Sie diese Teamarbeit bewusst „skizzieren", nicht „modellieren"!
- gerade-gut-genug: Vermeiden Sie das Streben nach Vollständigkeit Ihrer Modelle – meistens genügen Ausschnitte oder Teile des Systems für das Verständnis! [Ambler-12] nennt dies „Just Barely Good Enough".
- gerade-rechtzeitig: Verzögern Sie die Erstellung der ausgelieferten Dokumentation – es könnten sich ja noch Dinge ändern! [Ambler-12] nennt das „Document Late".
- Dokumentieren Sie kontinuierlich. Am Ende der letzten Iteration oder des Projektes können Sie sich nicht mehr an alle relevanten Dinge erinnern. In jeder Iteration oder Phase sollten Sie die dabei wichtigen Dinge dokumentieren. [Ambler-12] nennt das „Document Continuously".
- Diagramm-plus-Text: Ergänzen Sie grafische Modelle um kurze textuelle Erläuterungen. Gute technische Dokumentation kombiniert Bild mit Text!
- Manche Modellierungswerkzeuge bieten einen *Napkin-Style* an, bei dem Diagramme ähnlich wie handgemalt aussehen. Damit können Sie optisch sehr schön ausdrücken, dass das betreffende Modell einen Zwischenstand repräsentiert.
- Halten Sie Modelle möglichst redundanzfrei. Versuchen Sie, benötigte Dokumente aus einer einheitlichen Informations- oder Modellbasis zu generieren.

5.4 Weiterführende Literatur

[Rupp+12] ist das derzeit ausführlichste UML 2-Buch – nicht nur im deutschsprachigen Markt. Anhand vieler Beispiele aus alltäglichen Situationen präsentieren die Autoren die Diagrammarten.

[OMG_UML] ist die Originalquelle des UML 2-Standards: 4 Dokumente mit mehreren Hundert Seiten für Liebhaber staubtrockener Literatur (böse Zungen empfehlen diese Werke gegen Schlaflosigkeit).

[Ambler-12] Sehr pragmatisch finde ich die Hinweise von Scott Ambler zu gutem Modellierungsstil – zu finden unter *http://www.agilemodeling.com/style/*. Scott gibt sehr nützliche Hinweise, wie Sie die Mittel der UML besser, lesbarer und einfacher anwenden können.

6 Strukturentwurf, Architektur- und Designmuster

Der Schlüssel zur Maximierung der Wiederverwendung
liegt im Voraussehen von neuen Anforderungen ...
sowie im Entwurf Ihrer Systeme derart,
dass sie sich entsprechend entwickeln können.

[Gamma95] (Übersetzung Dirk Riehle)

Fragen, die dieses Kapitel beantwortet:

- Was sind Heuristiken?
- Wie können Sie Komplexität beim Entwurf behandeln?
- Welche Architekturmuster helfen beim „Entwurf im Großen“?
- Was sind Schichten, Layers und Tiers?
- Welche wichtigen Grundregeln gelten für den Entwurf von Software?
- Warum sind Abhängigkeiten ein gravierendes Problem?
- Wie minimieren oder verhindern Sie negative Folgen von Abhängigkeiten in Entwürfen?
- Welche Entwurfsmuster reduzieren Abhängigkeiten?
- Wie hängen Entwurf und Test zusammen?

In diesem Kapitel stelle ich Ihnen einige Ansätze zum Entwurf von Strukturen vor, zur Zerlegung eines Systems in Bausteine. Das ist die *klassische* Entwurfsaufgabe, zu der es keine algorithmische Lösung gibt. Wichtige Werkzeuge für angemessene (d. h. effektive) Entwürfe sind (und bleiben!) Erfahrung, Iterationen, Feedback und Geschmack, gepaart mit einer Portion Wiederverwendung.[1]

In den folgenden Abschnitten erfahren Sie einiges über Grundsätze der Zerlegung, über Architekturmuster und Heuristiken zum Entwurf. Daneben lernen Sie einige wichtige Entwurfsmuster und Entwurfsprinzipien kennen. Die Kombination dieser Ideen kommt Ihren Entwürfen zugute – leider aber ohne Erfolgsgarantie, denn Entwurf bleibt eine kreative Tätigkeit.

[1] Daneben werden Ihre Entwürfe primär von Technologie- und Produktvorgaben, organisatorischen Einschränkungen sowie den Qualitätsanforderungen an Systeme beeinflusst.

Aus Fehlern klug werden: Softwarearchitekten haben zu wenig Zeit

Erfolg kommt von Weisheit.
Weisheit kommt von Erfahrung.
Erfahrung kommt von Fehlern.

[Rechtin2000]

Dieses Zitat umschreibt die Erfahrung, dass man aus Fehlern hervorragend lernen kann. Leider sind nur wenige Auftraggeber bereit, die für Ihre Erfahrung als Softwarearchitekt notwendigen Fehler zu akzeptieren.

Heuristik ist Erfahrung

In dieser Situation helfen Heuristiken. Sie kodifizieren Erfahrungen anderer Architekten und Projekte, auch aus anderen Bereichen der Systemarchitektur.

Heuristiken sind nicht analytisch

Heuristiken, Abstraktionen von Erfahrung, sind nicht analytisch. Es sind Regeln zur Behandlung komplexer und wenig strukturierter (= schwieriger) Probleme (für die es meist beliebig viele Lösungsalternativen gibt). Heuristiken können helfen, Komplexität zu reduzieren.

Statt Heuristiken können Sie auch die Begriffe „Regel“, „Muster“ oder „Prinzip“ verwenden; im Grunde geht es immer um Ratschläge, die aus konkreten Situationen verallgemeinert oder abstrahiert wurden.

Heuristiken geben Orientierung, keine Garantie

Auf dem Weg in den sicheren Hafen (des fertigen Systems) bieten Heuristiken Orientierung im Sinne von Wegweisern, Straßenmarkierungen und Warnschildern. Aber Vorsicht: sie geben lediglich Hinweise und garantieren nichts. Es bleibt in Ihrer Verantwortung, die passenden Heuristiken für eine bestimmte Situation auszuwählen:

Die Kunst der Architektur liegt nicht in der Weisheit der Heuristiken, sondern in der Weisheit, a priori die passenden Heuristiken für das aktuelle Projekt auszuwählen.

[Rechtin2000]

Heuristiken effektiver Architektur

- Erfolg wird vom Kunden definiert, nicht vom Architekten.
- Beginnen Sie mit den riskanten oder schwierigen Teilen. Erstellen Sie möglichst frühzeitig Prototypen für diese Teile.
- Entwerfen Sie in (kleinen) Teams. Verifizieren Sie Entwürfe möglichst früh durch konkrete Implementierung. Arbeiten Sie Rückmeldungen (Feedback) in Entwürfe und Implementierung ein.
- Nehmen Sie nicht an, dass die ursprünglichen Anforderungen notwendigerweise die richtigen sind. Die endgültigen Anforderungen werden durch Implementierung und Architektur (mit-)geprägt! Hinterfragen Sie Anforderungen, weisen Sie auf Konsequenzen und Alternativen hin. Helfen Sie Auftraggebern und Projektleitung dabei, Aufwand und Nutzen der Anforderungen abzuwägen.

- Ein Modell ist keine Realität. Verifizieren Sie Ihre Modelle, möglichst gemeinsam mit Kunden und Auftraggebern. Wenn Sie glauben, Ihr Entwurf ist perfekt, dann haben Sie ihn noch niemandem gezeigt. Diskutieren Sie frühzeitig mit anderen Projektbeteiligten über Alternativen und Konsequenzen Ihrer Entwurfsentscheidungen.
- Entwerfen (und behalten) Sie Alternativen und Optionen so lange wie möglich in Ihren Entwürfen. Dokumentieren Sie Begründungen Ihrer Entwurfsentscheidungen und verworfene Optionen Sie werden sie brauchen. Nachvollziehbarkeit schafft Transparenz!

6.1 Von der Idee zur Struktur

Ich möchte Ihnen hier einige Techniken vorstellen, um die Komplexität eines Systems beherrschbar zu machen. Diese Heuristiken helfen Ihnen bei der Strukturierung von Systemen.

6.1.1 Komplexität beherrschen

Wenn Du es nicht in fünf Minuten erklären kannst,
hast Du es entweder selbst nicht verstanden, oder es funktioniert nicht.

[Rechtin2000]

Komplexität wird durch Zerlegung beherrschbar

Ein klassischer und systematischer Ansatz der Beherrschung von Komplexität lautet „Teile und herrsche“ (*divide et impera*). Zerlegen Sie Ihr Problem in immer kleinere Teile, bis diese Teilprobleme eine überschaubare Größe annehmen. Diesen Grundsatz der systematischen Vereinfachung können Sie für Softwarearchitekturen in verschiedenen Ausprägungen anwenden:

- „In Scheiben schneiden“: Zerlegen Sie ein System in (horizontale) Schichten. Jede Schicht stellt einige klar definierte Schnittstellen zur Verfügung und nutzt Dienste von darunter liegenden Schichten (siehe Abschnitt 6.2.3.3).
- Funktionale Zerlegung: Zerlegen Sie Ihr System in fachlich oder funktional motivierte Teile.
- In Abschnitt 6.2 finden Sie weitere Architekturstile, die Ihnen für die Zerlegung „im Großen“ Anregungen liefern können.

Bei der Vereinfachung durch Zerlegung sollten Sie ein zentrales Prinzip der Informatik beachten: die Kapselung (*information hiding*). Wenn Sie zerlegen, dann kapseln Sie Komplexität in Komponenten.

Betrachten Sie diese Komponenten als „black box“, und trennen Sie die Umgebung dieser Komponenten durch klar definierte Schnittstellen vom komplexen Innenleben.

Ohne Kapselung wird alles schwieriger

Wenn Sie die Kapselung missachten, entstehen ungewollte und schwierige Abhängigkeiten zwischen den einzelnen Systemteilen (Subsystemen, Komponenten), die insgesamt zu einer höheren Komplexität führen. Ohne Kapselung erschwert eine Zerlegung des Systems das Problem, statt es zu vereinfachen (denn: was bekannt ist, wird auch genutzt!).

6.1.2 Zerlegen – aber wie?

- Die wichtigsten Ratschläge für die Strukturierung Ihrer Systeme:
- Entwerfen Sie in Iterationen. Überprüfen Sie die Stärken und Schwächen eines Entwurfes anhand von Prototypen oder „technischen Durchstichen“. Bewerten Sie diese Versuche explizit, und überarbeiten Sie daraufhin Ihre Strukturen. Dieses iterative Vorgehen haben Sie in Kapitel 3 (Vorgehen beim Architekturentwurf) bereits kennengelernt – wenden Sie es beim Entwurf von Strukturen intensiv und nachhaltig an.
- Dokumentieren Sie die Entscheidungen, die zu einer bestimmten Struktur führen. Andere Projektbeteiligte werden sie künftig verstehen müssen.
- Orientieren Sie sich an den Qualitätsanforderungen für Ihr System. Leiten Sie Ihre wesentlichen Entwurfsentscheidungen aus diesen Q-Anforderungen her.
- Verwenden Sie Ihre Fachdomäne (= fachliche Funktionen und Daten) als Zerlegungskriterien.

Einige allgemeine Tipps zur Zerlegung von Systemen:

Eine Struktur soll geringe Kopplung und hohe Kohäsion besitzen.

Dieses Prinzip ist als *„Kopplung und Kohäsion“* schon 1979 von Larry Constantine und Ed Yourdon beschrieben worden. Kopplung ist ein Maß dafür, wie eng zwei Komponenten miteinander in Beziehung stehen oder voneinander abhängig sind. Kohäsion ist ein Maß, wie eng die Funktionen, die zwei Komponenten ausführen, miteinander in Beziehung stehen. Jedes Subsystem als Ergebnis einer Zerlegung sollte eine Menge von Konzepten oder Abstraktionen enthalten, die logisch oder inhaltlich eng zusammengehören (= hohe Kohäsion). Insbesondere soll die Kopplung von Komponenten innerhalb eines solchen Subsystems enger sein als zu Teilen anderer Subsysteme. Mehr zum Thema Kopplung finden Sie in Abschnitt 6.4.

- Wählen Sie bei der Zerlegung die Elemente so, dass sie möglichst unabhängig voneinander sind. Wenige Abhängigkeiten deuten häufig auf hohe Flexibilität und leichte Wartbarkeit von Systemen hin. Der ganze Abschnitt 6.4 ist diesem Thema gewidmet.
- Betrachten Sie Alternativen. Betrachten Sie das Problem von einem anderen Standpunkt aus. Überdenken Sie vermeintlich offensichtliche Ansätze.
- Eine Struktur sollte spätere Änderungen ermöglichen. Sie erreichen dies, wenn Ihre Entwürfe modular, verständlich und nachvollziehbar bleiben und Sie Verantwortlichkeiten trennen und kapseln (siehe Abschnitt 6.3.2).
- Komponenten sollen keine Annahmen über die Struktur anderer Komponenten machen.

6.1.3 Fachmodelle als Basis der Entwürfe

Beginnen Sie Ihren Entwurf grundsätzlich mit der Strukturierung der jeweiligen Fachdomäne. Dieser Ratschlag stellte bereits die Grundlage der (ehrwürdigen) *strukturierten Analyse* der frühen 80er-Jahre dar, gilt aber unverändert weiter. In den letzten Jahren hat Eric Evans dafür den Begriff *domain driven design* (DDD)[2] geprägt und die zugehörigen Konzepte präzisiert. Ich gebe Ihnen nachfolgend eine kurze Zusammenfassung, lege Ihnen zur Vertiefung weitere Literatur [Evans04, Vernon-13] ans Herz.

Modellieren Sie die Sprache der Fachdomäne

Die Basis jeder *guten* Software ist ein profundes Verständnis der jeweiligen Fachdomäne. Zu Beginn jeder Softwareentwicklung sollten Sie daher Ihr Domänenverständnis vertiefen. Modellieren Sie eine Abstraktion der Domäne, diskutieren Sie diese Abstraktion intensiv mit Fachexperten, aber auch mit Softwareentwicklern. Halten Sie dieses Domänenmodell frei von technischen Aspekten, beschränken Sie sich ausschließlich auf fachliche Themen!

Ein stabiles und akzeptiertes Domänenmodell bietet Ihnen (mindestens) folgende Vorteile:

- Über das Modell verbessern Sie die Kommunikation zwischen Fachexperten und Entwicklungsteam.
- Die Fachexperten können ihre Anforderungen präziser ausdrücken.
- Wenn Sie das Domänenmodell direkt in Software abbilden, können Sie es sehr leicht testen – ein unschätzbarer Vorteil.

Auf der Basis des gemeinsam abgestimmten Domänenmodells entsteht eine gemeinsame Sprache; [Evans04] nennt sie *ubiquitous language*. Das wird Ihr Projektjargon, Ihre domänenspezifische Sprache (deren Elemente Sie in das Projektglossar aufnehmen sollten).

[2] Nicht zu verwechseln mit dem ehrwürdigen Debugger-Frontend gleichen Namens, zu finden unter *http://www.gnu.org/software/ddd/*

Für das Domänenmodell verwenden Sie folgende Basisbausteine ([Evans04] nennt sie *Patterns*):

- Entities verfügen innerhalb der Domäne über eine unveränderliche Identität (einen Schlüssel) und einen klar definierten Lebenszyklus. Entities sind praktisch immer persistent. Sie stellen die Kernobjekte einer Fachdomäne dar.
- Value-Objects besitzen keine eigene Identität und beschreiben den Zustand anderer Objekte. Sie können aus anderen Value-Objekten bestehen, niemals aber aus Entitäten.
- Services stellen Abläufe oder Prozesse der Domäne dar, die nicht von Entities wahrgenommen werden. Es handelt sich dabei um Operationen, die in der Regel über keinen eigenen Zustand verfügen. Parameter und Ergebnisse dieser Operationen sind Domänenobjekte (Entities oder Value-Objects).

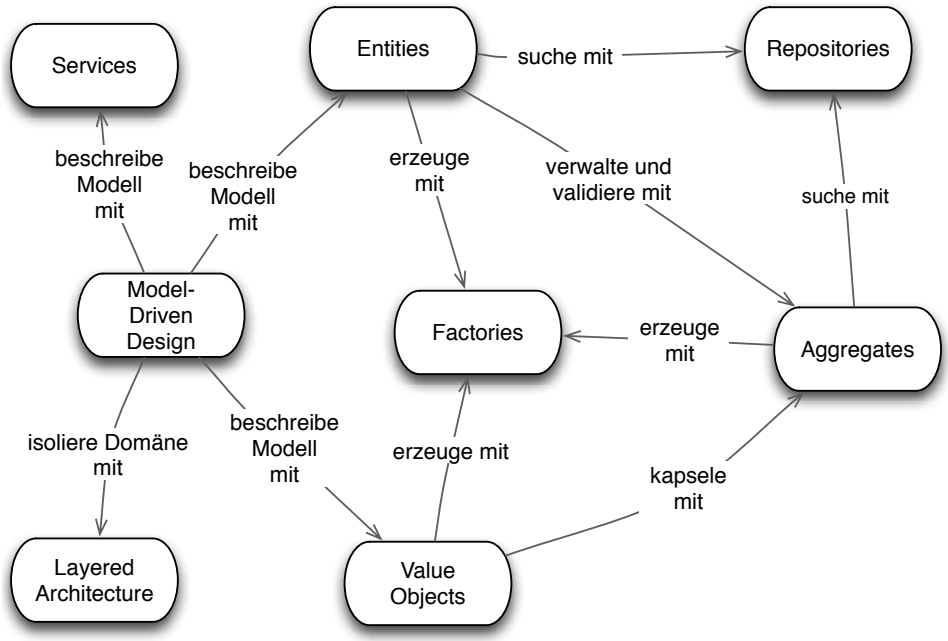


BILD 6.1 Muster des Domain Driven Design (nach [Abram+06] bzw. [Evans04])

Isolieren Sie die Domäne in der Architektur

Das Fachmodell, auch Analysemodell genannt, dient als Grundlage des weiteren Entwurfs. Damit auch langfristig die Struktur und Konzepte der Fachdomäne in der Architektur repräsentiert sind, sollten Sie die Domänenkonzepte (Entitäten, Services und Value-Objekte) in eine eigene Schicht der Architektur (*layered architecture*, siehe auch Abschnitt 6.2.1) verlagern.

DDD propagiert folgende Schichten:

User Interface (Presentation Layer)	Darstellung und Informationsanzeige, nimmt Eingaben und Kommandos von Benutzern entgegen.
Application Layer	Beschreibt oder koordiniert Geschäftsprozesse, delegiert an den Domain Layer oder den Infrastructure Layer.
Domain Layer (Fachmodell-Schicht)	Der Kern von DDD! Diese Schicht repräsentiert die Fachdomäne. Hier „lebt das Modell“ ([Evans04]) mit seinem aktuellen Zustand. Die Persistenz seiner Entitäten delegiert diese Schicht an den Infrastructure Layer.
Infrastructure Layer	Allgemeine technische Services wie beispielsweise Persistenz, Kommunikation mit anderen Systemen.

Verwalten Sie Domänenobjekte systematisch

- [Evans04] schlägt drei verschiedene Arten von „Verwaltungsobjekten“ vor, mit deren Hilfe Sie Ihre Domänenobjekte systematisch verwalten können:
- Aggregate: kapseln vernetzte (d. h. miteinander assoziierte) Domänenobjekte. Ein Aggregat hat grundsätzlich eine einzige Entität als Wurzelobjekt. Diese Wurzel ist der einzige „Einstiegspunkt“ in das Aggregat, sämtliche mit der Wurzel verbundene Domänenobjekte sind *lokal*. Objekte von außen dürfen nur Referenzen auf die Wurzelentität halten. Aggregate stellen die fachliche Konsistenz und Integrität ihrer enthaltenen Entitäten sicher.
- Factories: Entities und insbesondere Aggregate können komplexe Strukturen vernetzter Objekte bilden, die Sie nicht über triviale Konstruktoraufrufe erzeugen können oder wollen. Verwenden Sie Factories, um die Erzeugung von Aggregaten und Entitäten zu kapseln. Factory-Objekte arbeiten ausschließlich innerhalb der Domäne und haben keinen Zugriff auf den Infrastruktur-Layer.
- Repositories: kapseln die technischen Details der Infrastrukturschicht gegenüber den Domänenobjekten – sie besitzen die „Datenhoheit“. Dadurch bleibt das Domänenmodell auch in dieser Hinsicht „technologiefrei“. Repositories beschaffen beispielsweise Objektreferenzen von Entitäten, die aus Datenbanken gelesen werden müssen (siehe Abschnitt 6.2.3.3).

In Bild 6.2 (nach [Avram+06]) erkennen Sie das Zusammenwirken von Factory und Repository: Während die Factory Domainobjekte (Entities oder Aggregate) erzeugt, kümmert sich ein Repository um deren Übergabe an den Infrastructure Layer (im Beispiel repräsentiert durch die Datenbank). Somit stellen Repositories die Verbindung aus Fachdomäne und Infrastruktur da.

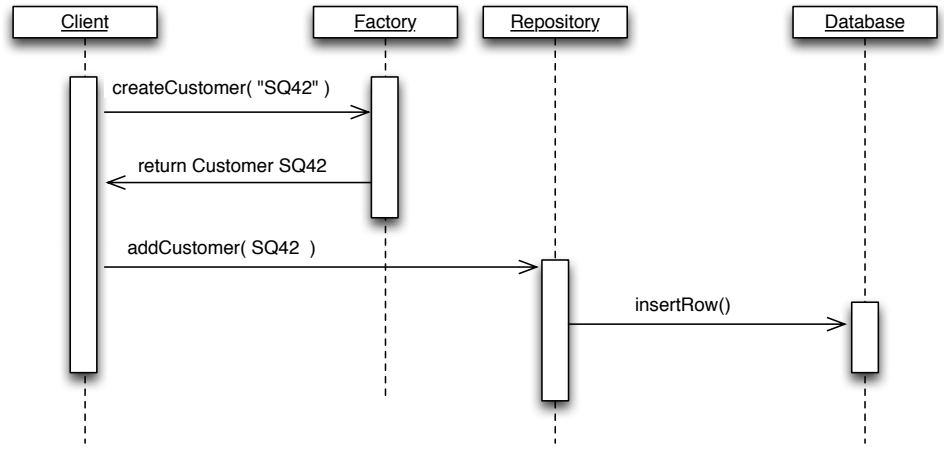


BILD 6.2 Zusammenwirken von Factory, Repository und Infrastruktur

6.1.4 Die Fachdomäne strukturieren

Für die Zerlegung der Fachdomäne möchte ich Ihnen einige Tipps geben:

- Bewahren Sie vor allem die konzeptionelle Integrität: Zerlegen Sie möglichst alle Teile nach ähnlichen Aspekten. Versuchen Sie, dieses „Konzept" der Zerlegung (oder Kapselung) im System konsistent anzuwenden.
- Kapseln oder gruppieren Sie Aspekte ähnlicher Änderungsrate. Gruppieren Sie Elemente, die in ähnlichen Intervallen modifiziert werden.* Insbesondere trennen Sie rein lesende von verändernden Operationen auf Daten!

* Dieser Ratschlag gilt auch für die Gestaltung grafischer Oberflächen!

Sie können die Fachdomäne nach zwei Gesichtspunkten strukturieren, wobei Sie auch beide Kriterien parallel anwenden können:

- nach Fachobjekten, den statischen oder datenorientierten Aspekten, oder
- nach fachlichen Abläufen, Anwendungsfällen oder Benutzertransaktionen.

Struktur nach Fachobjekten

Eine Zerlegung nach Fachobjekten (oder Fachklassen) entspricht im Wesentlichen der objektorientierten Zerlegung.

Wie auch bei der objektorientierten Modellierung kann es schwierig sein, die Verantwortlichkeit der eigentlichen Ablauflogik zu identifizieren.

Strukturieren Sie nach Fachobjekten, wenn

- Teile der Fachlogik in anderen Systemen wieder verwendet werden sollen;
- die Fachlogik komplex, umfangreich oder flexibel ist;
- Architekten und Entwickler das objektorientierte Paradigma beherrschen.

Struktur nach Abläufen oder Benutzertransaktionen

Ein Ablauf oder eine eine Benutzertransaktion umfasst eine Aktion, die ein Benutzer des Systems ausführen kann, inklusive sämtlicher systeminterner Operationen (etwa: Eingabedaten prüfen, Berechnungen ausführen, Ausgabedaten bereitstellen und anzeigen). Eine solche Benutzertransaktion kapselt einen systeminternen Ablauf oder einen Anwendungsfall.

Strukturieren Sie nach Abläufen oder Benutzertransaktionen, wenn:

- die Fachlogik im Wesentlichen aus Datenbeschaffung und einfachen Operationen auf diesen Daten besteht;
- die Fachlogik im Wesentlichen aus der Integration verschiedener Fremdsysteme besteht;
- sehr einfache oder sehr wenig Fachlogik vorliegt;
- Architekten oder Entwickler nur wenig mit Objektorientierung vertraut sind (und die Strukturierung nach Fachobjekten nicht beherrschen).

6.2 Architekturstile und -muster

Muster und Stil

Ein Architekturstil[3] fasst gemeinsame Merkmale von IT-Systemen zusammen, IT-Systeme eines bestimmten Stils bilden somit eine *Kategorie*. Jeder Stil enthält einige Regeln oder Muster zur Strukturierung eines Systems aus Elementen (Bausteinen) und Verbindungen (Beziehungen).

Architekturmuster können Ihnen bei der Zerlegung Ihres Systems in Bausteine und bei der Verteilung der Verantwortlichkeiten helfen. Sie bilden Vorlagen für Systemstrukturen. Ich verwende hier *Architekturmuster* als eine Konkretisierung oder Ausprägung von *Architekturstilen*.[4]

Reale Systeme: heterogen

Reale Systeme enthalten oft Merkmale mehrerer Architekturstile gleichzeitig, weil die Definitionen der Stile sich teilweise überschneiden. Das nennen wir dann *heterogene Architekturen*.[5]

[3] Der Begriff Architekturstil stammt aus [Shaw+96] – deren ursprüngliche Kategorisierung.

[4] Die genaue Abgrenzung zwischen Muster und Stil halte ich für irrelevant – beides sind nützliche Vorlagen.

[5] Siehe [Qiang+10] für eine ausführliche Darstellung – weitaus moderner und praxisnäher als das mittlerweile sehr alte Original [Shaw+96].

Im Vergleich zu den Entwurfsmustern („Design Patterns“, siehe Abschnitt 6.5 sowie [GoF]) beschreiben Architekturstile und Architekturmuster die Struktur von Systemen als Ganzes, weniger die Struktur einzelner Klassenverbünde oder Komponenten.

Ich zeige Ihnen Beispiele einiger praktisch bedeutsamer Architekturstile – die folgende Tabelle gibt einen ersten Überblick.

Diese Architekturstile bzw. Systemkategorien können Ihnen bei Entwurfsentscheidungen „im Großen“ nützlich sein. Daher sollten Sie für diese Stile folgende Parameter kennen:

- Arten der Bausteine, aus denen ein solches System besteht
- Arten der Verbindungen zwischen den Bausteinen
- Vor- und Nachteile dieses Architekturstils

Ich möchte Ihnen zuerst einige aus meiner Sicht wesentliche Architekturstile vorstellen und in Abschnitt 6.2.10 dann zusammenfassen, für welche Art von Anforderungen sich welcher Stil besonders eignet.

TABELLE 6.1 Architekturstile und Beispiele

Stil	Intention	Beispiele
Datenfluss-Systeme	besteht aus einer Folge (Sequenz) von Operationen auf Daten	Batch-Sequenziell (Abschnitt 6.2.1.1), Pipes-und-Filter (Abschnitt 6.2.1.2), Prozesssteuerung
Datenzentrische Systeme	gemeinsam verwendeter, zentraler Datenbestand	Abschnitt 6.2.2: Repository, Blackboard
Hierarchische Systeme	bestehen aus Bausteinen in unterschiedlichen Ebenen einer Hierarchie	Haupt- und Unterprogramm, Master-Server, Schichten (Abschnitt 6.2.3), Ports- und Adapter (Abschnitt 6.2.3.3), virtuelle Maschine
Verteilte Systeme	bestehen aus Speicher- und Verarbeitungsbausteinen, die über Kommunikationsnetze interagieren	Abschnitt 6.2.4: Client-Server, CQRS, Broker-Architekturen, Service-orientierte Architekturen, Peer-to-Peer
Ereignisbasierte Systeme	Besteht aus voneinander unabhängigen Bausteinen, die über Ereignisse (Events), kommunizieren, sich *implizit* aufrufen.	Abschnitt 6.2.5: Ungepufferte Kommunikation: Publisher-Subscriber Gepufferte Kommunikation: Message-Queue, Message-Service
Interaktions-orientierte Systeme	Systeme mit (grafischer) Bedienoberfläche	Abschnitt 6.2.6: Model-View-Controller, Model-View-Presenter, Model-View-View-Model (MVVM), Presentation-Abstraction-Control, Presentation-Model
Heterogene Systeme	Verwenden mehrere Architekturstile parallel	Siehe „Sonstige Architekturstile“, beispielsweise REST.

6.2.1 Datenfluss-Architekturstil

6.2.1.1 Architekturstil „Batch-sequenziell“

Batch-Systeme stammen aus den Zeiten von COBOL und Mainframes. Sie verarbeiten ihre Eingangsdaten in einer Folge von Transformationen, wobei diese meist strikt sequenziell ablaufen, d. h. ein Nachfolger startet erst, wenn sein Vorgänger sämtliche Eingangsdaten komplett verarbeitet hat. Diese Systeme schicken somit Daten in „Stapeln[6]“ von einem Verarbeitungsbaustein zum nächsten. Die Verbindungen zwischen diesen Bausteinen bilden oft Dateien, d. h. die Schnittstellen der einzelnen Datentransformationen sind Ein- und Ausgabedateien.

Ablaufsteuerung und Koordination muss durch externe Bausteine erfolgen, beispielsweise durch Skripte.

Trotz dieser einfachen Struktur und der fehlenden Interaktionsmöglichkeit im laufenden System laufen auch heute noch wesentliche Teile der (Offline-) Datenverarbeitung, etwa bei Banken und Versicherungen, nach diesem grundsätzlichen Muster – allerdings erweitert um Ablaufsteuerung, Fehler- und Ausnahmebehandlung sowie auf Basis von Datenbanken anstelle von Dateien.

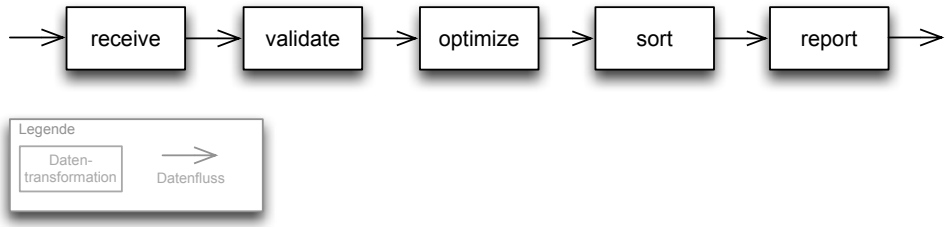


BILD 6.3 Beispiel für Architekturstil „Batch-sequenziell“

Kriterien für die Anwendung:

- Ergebnisse können durch eine (feste) Kette von Verarbeitungsschritten entstehen.
- Sämtliche Zwischenergebnisse können in ein einheitliches Format („Datei“) gebracht werden.
- Einzelne Verarbeitungsschritte können von mehreren oder allen Eingabeinformationen abhängen (Beispiele: Aggregierungen mehrerer oder aller Eingangsdaten, inhaltliche Bezüge von Eingangsdaten zueinander).

Vorteile:

- Einfache, funktional motivierte Aufteilung des Systems.
- (Meist) einfache und ähnliche Schnittstellen, z. B. Ein- und Ausgabedateien.

Nachteile:

- Ein externer Baustein muss die Steuerung des Systems übernehmen, ebenso die übergreifende Fehlerbehandlung.

[6] Daher der Name „batch“ (von engl. batch = Stapel).

Mögliche Erweiterungen:

- Parallelisierung einzelner Verarbeitungsschritte, um den Durchsatz zu erhöhen.
- Erweiterung einzelner Bausteine um weitere Ausgangsschnittstellen, um beispielsweise Verzweigungen (Fallunterscheidungen, Verarbeitungsvarianten) zu ermöglichen.
- Flexibilisierung der Schnittstellen zwischen den Bausteinen, Verwendung von Datenbanken oder Message-Queues in Ergänzung zu Dateien.

Implementierung

Das (Open-Source) Spring-Batch Framework[7] stellt eine robuste und flexible Basis für (Java-) Batch-Anwendungen bereit. Seine außergewöhnlich gute Dokumentation[8] enthält viele Ratschläge für Entwurf und Implementierung solcher Systeme.

6.2.1.2 Architekturstil Pipes und Filter

Kennen Sie das Pipe-Symbol der Unix-Kommandozeile? Es leitet die Ausgabe einzelner Unix-Kommandos an nachfolgende Programme weiter. Ein Beispiel:

```
ls –l | grep ".pl" | sort
```

Diese Befehlsfolge erzeugt zuerst (zeilenweise) eine Liste aller Dateien des aktuellen Verzeichnisses, übergibt das Resultat (durch das Symbol „|“, gesprochen „pipe“) an das grep-Programm zum Filtern aller Zeilen, in denen die Zeichenfolge „.pl“ vorkommt.

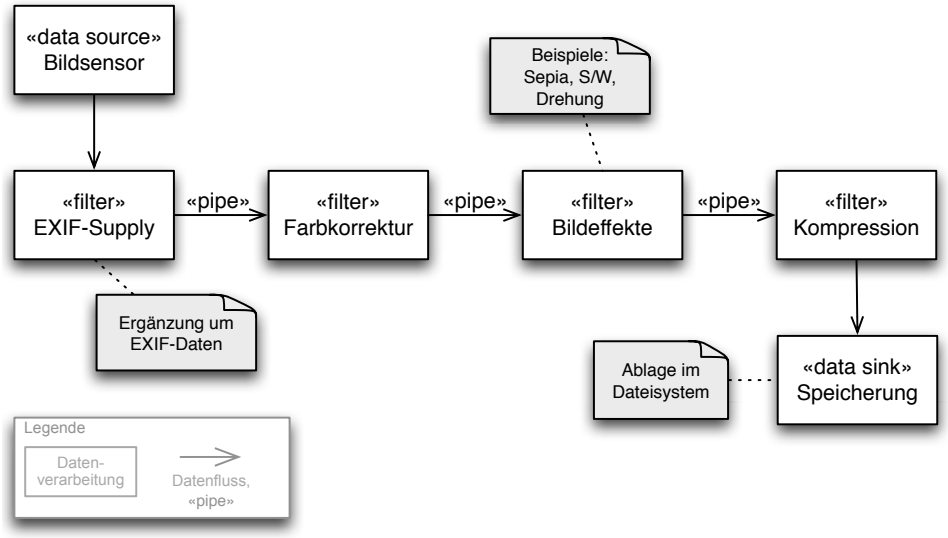


BILD 6.4 Pipe und Filter Struktur einer Digitalkamera (vereinfacht)

[7] *http://static.springsource.org/spring-batch/*
[8] *http://static.springsource.org/spring-batch/batch-processing-strategies.html*

Verallgemeinern Sie dieses Verfahren, gelangen Sie zum „Pipes-und-Filter“-Architekturmuster: eine Folge von Verarbeitungseinheiten (Filter), miteinander durch Datenkanäle (Pipes) verbunden. Am Beispiel einer Digitalkamera zeigt Bild 6.4 dieses Muster. Jeder Filter gibt seine Ergebnisse über eine Pipe an den nächsten Filter weiter – die Pipes dienen lediglich als Transportmittel für Daten.

Filter verarbeiten Daten

Filter verarbeiten

Filter sind aktive Verarbeitungs- oder Transformationsbausteine, die Eingabedaten über Eingangs-Pipe(s) erhalten, prozessieren und über Ausgangs-Pipe(s) weitergeben.

Filter haben in diesem Muster keinerlei direkte Abhängigkeiten zu anderen Filtern.

Filter können ihre Verarbeitungsergebnisse an eine oder mehrere Ausgangs-Pipes weiterreichen und damit Verzweigungen innerhalb der Datenflüsse realisieren.

Unterscheiden Sie zwischen aktiven und passiven Filtern:

- Aktive Filter holen sich aktiv ihre Eingangsdaten aus der Eingangspipe und stellen nach der vollständigen Verarbeitung ihre Ergebnisse aktiv der Ausgangspipe zur Verfügung (pull and push)
- Passive Filter warten, bis sie Eingangsdaten übergeben bekommen und bis ihre Ergebnisdaten abgeholt werden.

Die aktiv-/passiv-Eigenschaft von Filtern kann sich auch auf ihren Ein- bzw. Ausgang beschränken, beispielsweise *aktiv-Input, passive-Output*.

Pipes puffern und transportieren Daten

Pipes puffern und transportieren

Pipes sind Datenkanäle oder Transportmittel – sie können Daten entweder sofort oder auch zeitversetzt weitergeben. Manchmal fungieren sogar Datenbanken als Pipes (ein umfangreiches Beispiel in Kapitel 11 illustriert das). Pipes können Daten puffern, um unterschiedliche Verarbeitungsgeschwindigkeiten ihrer Ein- und Ausgangsfilter auszugleichen.

Im Pipe-und-Filter-System können sämtliche Bausteine gleichzeitig aktiv sein – ganz im Sinne paralleler Prozesse.

Beispiel: Kapitel 11

Kriterien für die Anwendung:

- Ergebnisse können durch eine (feste) Kette von Verarbeitungsschritten entstehen.
- Einzelne Datensätze können völlig unabhängig von anderen bearbeitet werden.
- Das Datenformat auf den Pipes ist stabil.

Vorteile:

- Einfache Struktur, leicht zu implementieren.
- Lose Kopplung beziehungsweise geringe Abhängigkeiten der Filter voneinander. Filter können i. d. R. einfach durch alternative Implementierungen ausgetauscht werden.

Nachteile:

- Fehlerbehandlung: Da Filter einander nicht kennen, ist die Behandlung von „Folgefehlern“ (ohne strukturelle Erweiterung am Muster) zumindest schwierig, teilweise kaum möglich.
- Konfiguration oder Initialisierung der gesamten Verarbeitungskette benötigt eine „zentrale Steuerung“. Verlagern Sie diese Aufgabe in die Filter, so erzeugen Sie Abhängigkeiten der Filter voneinander.
- Es gibt keinen „gemeinsamen Zustand“ von Filtern. Alle Verarbeitungsinformationen müssen mit den Daten übertragen (oder einer zentralen Steuerung überlassen) werden.
- Ungeeignet für interaktive Systeme, in denen Benutzer Eingriffsmöglichkeiten in die Verarbeitung benötigen.

Mögliche Erweiterungen:

- Pipes können entscheiden, an welche konkrete Instanz eines Filters sie ihre aktuellen Daten weitergeben („Wer hat gerade freie Kapazitäten?“).

Implementierung:

In der Realität kommen häufig Kombinationen aus Batch-sequenziell und Pipes-und-Filter vor. Beispiel:

1. Eine Eingangsdatei wird zuerst (Batch-sequenziell) vollständig gelesen, entkomprimiert und entschlüsselt. Diese drei Operationen müssen für alle Daten gemeinsam durchgeführt werden (genauer: für die gesamte Datei, d. h. für alle darin enthaltenen Datensätze).
2. Anschließend werden (Batch-sequenziell) sämtliche Datensätze kategorisiert (d. h. auf kleinere Datenstapel verteilt).
3. Jeder dieser einzelnen Stapel wird gemäß Pipe&Filter-Verarbeitung datensatzweise prozessiert.
4. Am Ende werden die Ergebnisse der Verarbeitung wieder (Batch-sequenziell) in einer einzelnen Datei gesammelt.

6.2.2 Datenzentrierter Architekturstil

Datenzentrierte Systeme enthalten einen zentralen (passiven) Datenspeicher, der von den übrigen Bausteinen des Systems verwendet wird.

6.2.2.1 Repository

Repository-Systeme gruppieren eine Reihe aktiver Bausteine (in [Quiang+10] *Agenten* genannt) um den passiven Datenspeicher. Die Agenten realisieren Verarbeitung und Ablaufsteuerung innerhalb des Systems.

Bild 6.5 zeigt die grundsätzliche Struktur von Repository-Systemen.

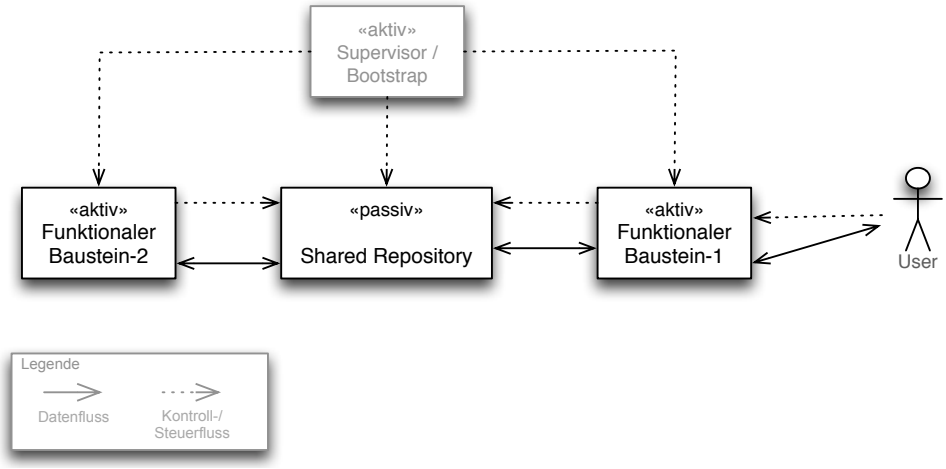


BILD 6.5 Bausteinstruktur (vereinfacht) eines Repository-Systems

Generell folgen sehr viele Informationssysteme dieser allgemeinen Struktur:

- Betriebliche Informationssysteme auf Basis von Datenbanken: Meist haben die „Funktionalen Bausteine“ hierbei eine komplexe innere Struktur.
- Versionsverwaltungssysteme für Quellcode (*Sourcecode-Repositories*), wie Subversion. Die modernen DVCS (*distributed version control systems*) wie Git oder Mercurial verteilen die Aufgaben des Repositories auf mehrere getrennte Instanzen und ermöglichen die Synchronisation dazwischen.
- Data-Warehouse oder Business-Intelligence-Systeme enthalten meist ein zentrales Repository, das über ETL-Mechanismen (*Extract-Transform-Load*) aus dem operativen Datenbestand befüllt wird.
- Die Registry von Microsoft Windows™
- Die Registraturen von Software-Installern (wie Linux-RPM)

6.2.2.2 Blackboard

> *The Blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.*
>
> Aus: [Buschmann+96, p71]

Blackboard[9] wurde im Bereich der künstlichen Intelligenz für Expertensysteme eingesetzt, etwa zur Sprach- oder Mustererkennung. [Buschmann+96] nennen drei wesentliche Bestandteile:

[9] siehe auch *http://en.wikipedia.org/wiki/Blackboard_system*

- eine oder mehrere Knowledge-Sources, die das Problem aus ihrer spezifischen Sicht untersuchen und Lösungsvorschläge an das Blackboard weitergeben;
- das zentrale „Blackboard“, das die Lösungsansätze oder -bestandteile der Knowledge-Sources verwaltet. Interpretieren Sie das Blackboard als die Sammelstelle für Zwischenergebnisse sowie endgültige Resultate der Verarbeitung;
- eine Kontrollkomponente, die das Blackboard beobachtet und bei Bedarf die Ausführung der Knowledge-Sources steuert.

Oft sind die Knowledge-Sources über einen publish/subscribe-Mechanismus an das Blackboard gekoppelt und erfahren somit, ob das Blackboard neue oder geänderte Informationen enthält.

Kriterien für die Anwendung:

- Falls Sie sich mit künstlicher Intelligenz beschäftigen, könnte Blackboard für Sie passen. Systeme mit externalisierten Geschäftsregeln[10] (siehe Kapitel 7.2) setzen im Grunde dieses Muster um: Das Blackboard ist der „Arbeitsspeicher“ der Regelmaschinen/Expertensysteme, in dem Regeln auf vorhandene Informationen angewendet werden.
- Komplexe Probleme ohne feste Lösungsverfahren, bei denen mehrere unabhängige Bausteine jeweils Teilaspekte einer möglichen Lösung berechnen können.
- Probleme, die sich nicht praktikabel über vollständige Suche oder Aufzählung lösen lassen.
- Probleme, bei denen partielle oder suboptimale Lösungen akzeptabel sind.

Vorteile:

- Hinzufügen weiterer Agenten (Wissens- oder Informationsverarbeiter) ist einfach.
- Parallelität: Alle beteiligten Agenten können parallel arbeiten.

Nachteile:

- Synchronisierung mehrerer Agenten kann aufwendig und schwierig sein.
- Aufgrund der bearbeiteten Probleme kann es schwierig sein, einen günstigen Zeitpunkt für das Ende der Berechnung zu finden.
- Test und Fehlersuche können schwierig sein.

6.2.3 Hierarchische Architekturstile

Systeme hierarchischer Architekturstile bestehen aus Elementen, die einander über- bzw. untergeordnet sind. Untergeordnete Bausteine bieten dabei ihren „Oberen“ feste Dienste an. Aufruf- oder Nutzungsbeziehungen verlaufen strikt in Richtung dieser Hierarchie.

Hierarchische Strukturen werden in der Praxis oftmals mit anderen Architekturstilen kombiniert.

[10] etwa mit Regel-Engines wie JBoss-Drools, Visual-Rules oder ILog-Rules.

Zu diesem Stil gehören die folgenden wichtigen Vertreter:

- Haupt-/Unterprogramm-Struktur, bekannt aus imperativen Programmiersprachen.
- Master-Slave, bei dem mehrere (redundante) Slave-Bausteine einem Master identische Dienste anbieten.
- Schichten (Layer), das in kommerziellen Informationssystemen wohl am meisten verbreitete Architekturmuster
- Ports- und Adapter, eine spezielle Ausprägung des Schichtenmodells
- Virtuelle Maschine, ebenfalls eine Spezialisierung des Schichtenmodells

Viele Betriebssysteme oder systemnahe Programme prägen eine klare Hierarchie von Bausteinen aus – siehe nachfolgende Abbildung.

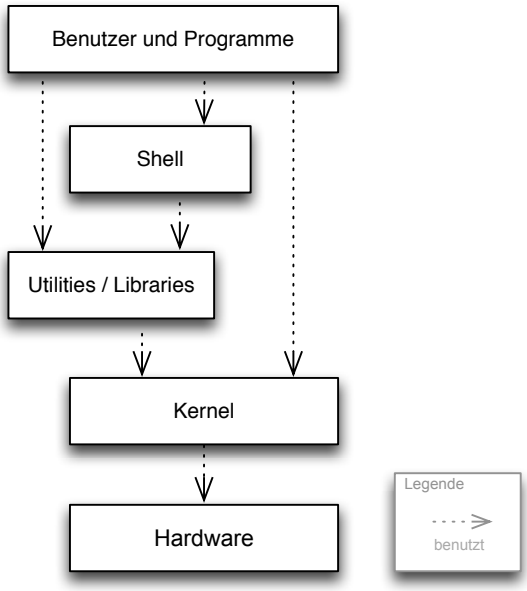


BILD 6.6
Hierarchischer Aufbau des Betriebssystems Unix (vereinfacht)

6.2.3.1 Master-Slave

Eine wichtige Ausprägung hierarchischer Architekturstile, *Master-Slave*, unterstützt die Qualitätsanforderungen Fehlertoleranz und Verfügbarkeit: Ein Master verwendet mehrere Slaves, (redundante) Anbieter desselben Dienstes.

Die Slaves können in objektorientierter Terminologie Instanzen derselben Klasse sein, aber auch unterschiedliche Implementierungen desselben Interfaces. Der Master könnte im letzteren Fall beispielsweise eine Anfrage an mehrere Slaves vergeben und das Gesamtergebnis durch einen Mehrheitsentscheid oder eine Mittelung bestimmen.

Grundsätzlich können sämtliche Slaves parallel arbeiten, bei Bedarf sogar auf jeweils unterschiedlicher Hardware.

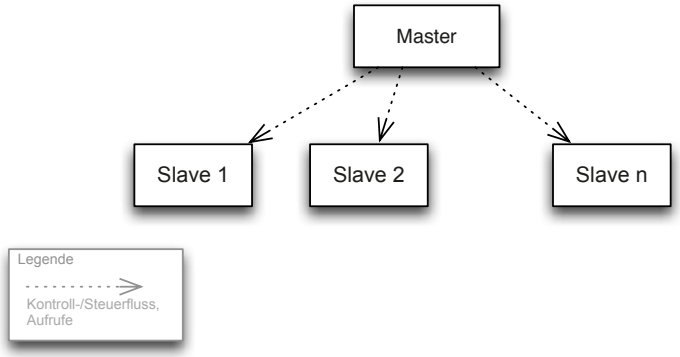


BILD 6.7
Master-Slave

Kriterien für die Anwendung:

- Aufgrund der möglichen Redundanz von Slaves eignet sich das Muster für Systeme mit hohen Anforderungen an Verfügbarkeit und Zuverlässigkeit.

6.2.3.2 Schichten (Layer)

Schichten, Layer

In Informations- oder Websystemen hat sich die Schichtenbildung (*layering*)[11] als ein klassisches Mittel zur Strukturierung etabliert. Eine Schicht bietet den darüber liegenden Schichten eine funktionale oder strukturelle Abstraktion bestimmter Dienste (*services*) an. Eine Schicht kapselt die Details ihrer Implementierung gegenüber der Außenwelt. Sie kann dabei ein beliebig komplexes Subsystem darstellen.

Komponenten innerhalb einer Schicht besitzen einen ähnlichen Abstraktionsgrad. So kapseln „niedrige“ Schichten oftmals technische oder sogar physikalische Aspekte von Systemen („Hardware- oder Infrastrukturschicht“).

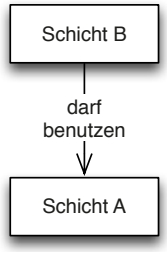


BILD 6.8
Schichten

Gegenüber anderen Zerlegungsmechanismen legt die Schichtenbildung einige wichtige Eigenschaften fest. So darf eine höhere Schicht („B“) Dienste einer darunter liegenden Schicht („A“) benutzen.

Die umgekehrte Nutzung (eine darunter liegende Schicht benutzt eine höhere) sollten Sie möglichst unterbinden. Sie schränkt die Unabhängigkeit einer Schicht von den darunter liegenden ein. Wenn die wechselseitige Nutzung von Komponenten notwendig ist, sollten sich diese Komponenten möglichst in ein- und derselben Schicht befinden.

[11] Layer (englisch): Schicht, Lage, Ebene

Schichten haben Vorteile ...

- Schichten sind voneinander unabhängig, sowohl bei der Erstellung als auch im Betrieb von Systemen.
- Sie können die Implementierung einer Schicht austauschen, sofern die neue Implementierung die gleichen Dienste anbietet.
- Schichtenbildung kann helfen, Abhängigkeiten zwischen Komponenten verschiedener Schichten zu reduzieren.
- Schichtenbildung ist ein leicht verständliches Strukturkonzept.

Ein Beispiel für (erfolgreiche) Schichtenbildung sind die ISO/OSI-Schichten für Kommunikation. So können Sie jederzeit einen eigenen ftp-Klienten auf Basis der TCP/IP-Schicht schreiben, ohne die Details darunter liegender Schichten zu kennen (siehe Abschnitt 7.4 (Kommunikation)).

... aber auch Nachteile

- Schichtenbildung kann die Performance eines Systems beeinträchtigen, weil Anfragen unter Umständen durch mehrere Schichten durchgereicht werden, bis sie anschließend bearbeitet werden. Abhilfe schafft das so genannte *Layer-Bridging*: Es erlaubt das „Überspringen“ von Zwischenstufen (siehe dazu Bild 6.9). Ein Aufruf der Schicht C an einen Service, den Schicht A bereitstellt, kann eine Schicht überspringen. Der Vorteil besserer Performance wird durch die zusätzliche Abhängigkeit (Schicht C hängt von Schicht A ab) erkauft.

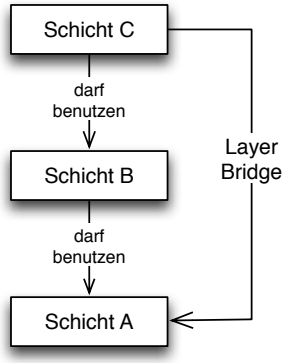


BILD 6.9
Layer Bridge

- Manche Änderungen eines Systems werden von Schichten schlecht unterstützt. Beispiel: Sie fügen einem System ein neues Datenfeld hinzu, das sowohl gespeichert als auch in der grafischen Benutzeroberfläche angezeigt werden soll. Das zieht Änderungen an allen beteiligten Schichten nach sich.

Ein (Standard-)Vorschlag für Schichten

Die folgenden Schichten bieten oftmals einen Startpunkt für eine weitere Zerlegung (siehe auch die Diskussion zu *Domain Driven Design* in Abschnitt 6.1.3):

- Präsentation (Benutzeroberfläche).
- (optional): Applikationsschicht, Koordination von Fachobjekten und Delegation an Fachdomäne und Infrastruktur.
- Fachdomäne (*Business Architecture*): Ein solches Subsystem ermöglicht seinen Entwicklern, sich auf die fachlichen Aspekte des Systems zu konzentrieren.
- Infrastruktur (*Technical Architecture*): Kapselt die Komplexität der technischen Infrastruktur gegenüber der Fachdomäne und der Präsentation. Die technische Infrastruktur können Sie bei Bedarf in weitere Komponenten oder Schichten zerlegen. Typische Untergliederungen hierfür sind:
  - Persistenz und Datenhaltung
  - Physikalische Infrastruktur (Schnittstellen zur Hardware)
  - Integration (von Fremdsystemen)
  - Sicherheit
  - Kommunikation und Verteilung

Client/Server

Interessant für die Praxis ist häufig die Aufteilung der Präsentation in einen Client- und einen Server-Teil. Der Client-Teil läuft dabei auf den Client-Rechnern ab, die Server-Präsentationsschicht auf dem Server.

- Vermeiden Sie es, Aspekte der Fachdomäne (fachliche Logik*) in die Präsentationsschicht zu verlagern. Das führt einerseits zu schwer wartbaren Systemen, andererseits reduziert es die Möglichkeit der Wiederverwendung innerhalb der Fachdomäne. Abhilfe bietet hier der Ports-und-Adapter-Architekturstil aus Abschnitt 6.2.3.3.
- Wenn Sie viele externe Ressourcen (Fremdsysteme) integrieren, kann eine Aufteilung der Infrastrukturschicht in Integrationsschicht und Ressourcenschicht übersichtlich sein.

* Völlig legitim oder erwünscht wäre jedoch, fachliche Validierungsregeln einmal zu implementieren und in unterschiedlichen Schichten (wieder) zu verwenden.

Nachteile der (Standard-)Schichten

In der Realität infiltriert[12] bei den oben genannten „Standard"-Schichten sehr häufig fachliche Logik die Applikations- und Präsentationsschicht. Das klingt zuerst wenig dramatisch, führt allerdings bei Systemen mit langer Lebensdauer und/oder hoher Änderungsrate zu einer Reihe gravierender Probleme:

[12] Den Begriff „infiltrieren" habe ich aus [Cockburn05] übernommen – den Sie (online) unbedingt lesen sollten.

- Die Testbarkeit der Fachlichkeit („Domänen-Logik “) sinkt – weil diese Logik jetzt über mehrere Schichten verteilt ist.
- Eine alternative Nutzung des Systems, beispielsweise über eine Batch-Schnittstelle oder ein neues grafisches Interface, wird stark erschwert, weil fachliche Logik für diese Schnittstelle erneut implementiert werden muss.
- Die Verständlichkeit des Gesamtsystems sinkt aufgrund der niedrigen Kohäsion der Bausteine innerhalb von Applikations- und Präsentationsschicht.

Aber damit nicht genug: Die Infrastrukturschicht, oft synonym für Datenbank, wird aus der darüber liegenden Fachdomäne aufgerufen: Damit dringt sehr leicht technisches Detail über die Persistenz in die Fachdomäne, die ja eigentlich komplett technikneutral sein sollte. Umgekehrt können wir eine Persistenz nicht implementieren, ohne Details der Fachdomäne zu kennen! Damit erhalten wir in jedem Fall eine Abhängigkeit in die falsche Richtung, *von Infrastruktur zur Fachdomäne*.

Aufgrund dieser praktischen Probleme hat Alistair Cockburn in 2005 eine bedeutsame Präzisierung der ursprünglichen Schichten-Idee vorgenommen ([Cockburn05]). Ursprünglich hat er sie als *hexagonale Architektur* veröffentlicht, mittlerweile auch bekannt[13] unter dem Namen *Ports-und-Adapter*.

6.2.3.3 Architekturstil „Ports und Adapter“

(In der Literatur auch bekannt als *Hexagonal Architecture*, von Robert Martin als *Clean-Architecture*[13] bezeichnet.)

Die jeweilige Fachdomäne sollte den Kern eines jeden Systems darstellen und keine (!) Abhängigkeiten von jeglicher Infrastruktur enthalten.

Die ursprüngliche Terminologie (von Alistair Cockburn) der *Ports* und *Adapter* finden Sie in Bild 6.10: Externe Systeme, Implementierungen von Use-Cases oder die Benutzerschnittstelle verwenden Ports der Fachdomäne. Die technische Infrastruktur implementiert diese Ports (im Sinne des Dependency-Inversion-Prinzips, siehe Abschnitt 6.4.4).

Bild 6.11, stark angelehnt an Robert Martins Clean-Architecture[14] illustriert dieselbe Idee in einem Schalenmodell: In einem inneren Kreis ist nichts (!) bekannt, was weiter außen definiert wird. Sourcecode-Abhängigkeiten zeigen immer (!) nach innen.

Die **Fachdomäne** kapselt fachliche Entitäten, Regeln, Objekte mit fachlichen Methoden, fachliche Funktionen oder fachliche Datenstrukturen. Innerhalb der Fachdomäne sollten Sie niemals Dinge ändern müssen, nur weil sich irgendwelche Details der Infrastruktur ändern.

Applikations-/Anwendungsregeln oder **Use-Cases** enthalten Regeln, Objekte oder Daten, die sich auf den Anwendungs- oder Systemzustand oder das Anwendungs- oder Systemverhalten beziehen. Sie implementieren die Anwendungsfälle des Systems – unter Zuhilfenahme der fachlichen Entitäten/Regeln der Fachdomäne.

---

[13] Die Ports-und-Adapter Ideen finden Sie beispielsweise in [Vernon13], [Freeman+10] sowie dem Apache-Isis-Framework.

[14] *http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html*

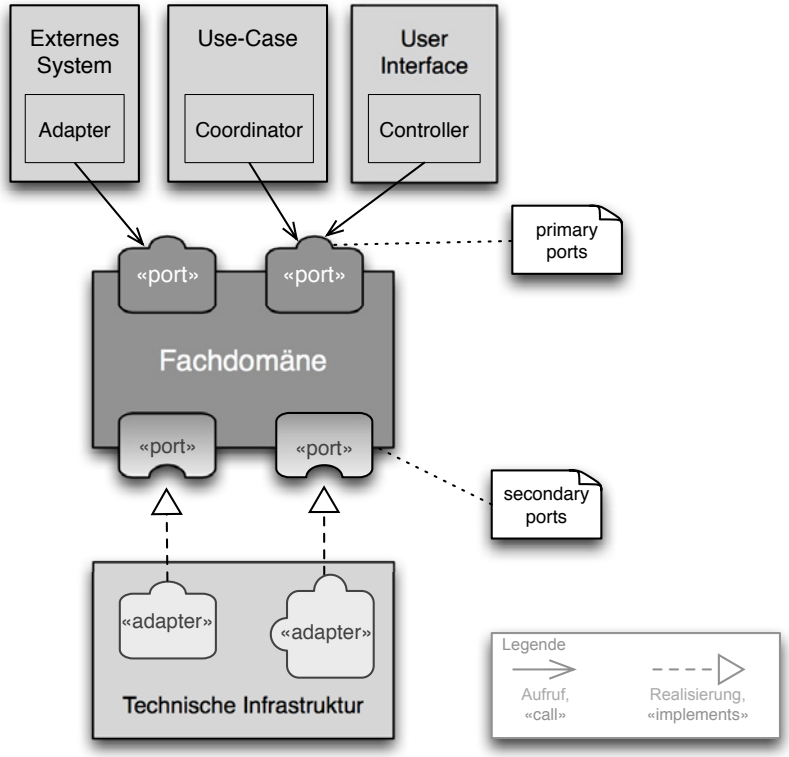


BILD 6.10 Ports und Adapter

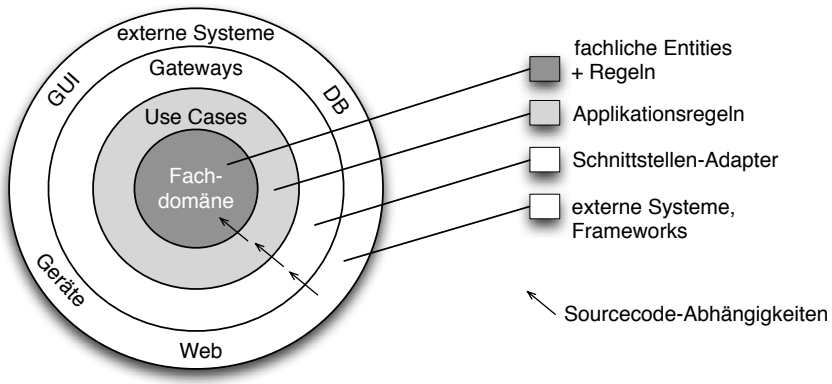


BILD 6.11 Schalen- oder Zwiebelmodell der *Clean-Architecture*

Die **Schnittstellen-Adapter** konvertieren Datenformate oder implementieren Schnittstellen der Fachdomäne und kapseln damit technische Details der konkreten Infrastruktur. Falls beispielsweise Daten in einer spezifischen NoSQL-Datenbank gespeichert werden müssen, so versteckt ein entsprechender Adapter sämtliche für diese Datenbank nötigen Details. Zu den Schnittstellen-Adaptern gehören auch Views, Controller oder Presenter einer grafischen Oberfläche.

Den äußersten Ring bilden **Frameworks** oder Softwareprodukte, die Sie im Rahmen einer Systementwicklung praktisch niemals selbst anpassen oder erweitern – außer mit sogenanntem Glue-Code.

Implementierung des (strikten) Schalenmodells

Zur Implementierung dieses Architekturstils bietet sich Dependency-Inversion an: Weiter außen liegende Bausteine implementieren Schnittstellen (Interfaces), die von inneren Bausteinen definiert werden. Die inneren Bausteine (beispielsweise Domain-Entities) benutzen nur die von ihnen selbst definierten Schnittstellen – die passende Implementierung wird zur Laufzeit bereitgestellt, beispielsweise durch Dependency-Injection oder Callbacks.

- Primäre Ports stellt die Fachdomäne als Methoden oder Funktionen zur direkten Verwendung (Aufruf) bereit (DomainAPI). In typischen OO-Sprachen werden sie als public-Methoden deklariert und implementiert.
- Sekundäre Ports hingegen sind lediglich Schnittstellen, zu denen andere Bausteine noch Implementierungen bereitstellen müssen. Insbesondere wird die konkrete technische Infrastruktur (etwa: Persistenz) solche sekundären Ports implementieren, aber auch Mock-Objekte für Unit- und Integrationstests.
- Die „adapter“[15] erfüllen die Aufgabe des gleichnamigen Entwurfsmusters – sie überführen die von Ports angeforderten Schnittstellen in die von der Infrastruktur bereitgestellten.

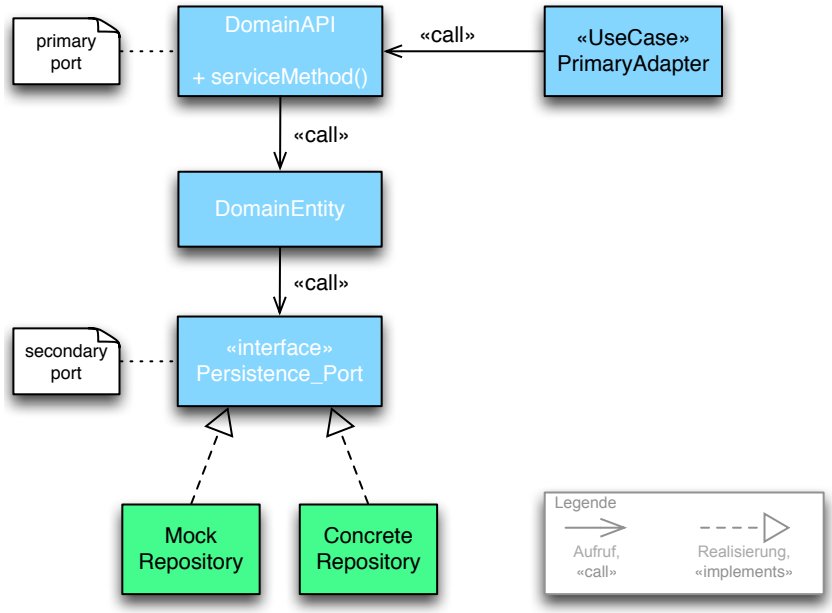


BILD 6.12 Implementierung der *Clean-Architecture*

[15] Cockburn unterscheidet genau genommen noch zwischen primären und sekundären Adaptern: Die primären rufen primäre Ports auf, die sekundären Adapter implementieren sekundäre Ports. Ich vereinfache hier – und nenne nur die „sekundären“ Adapter.

Kriterien für die Anwendung:

- langlebige fachliche Strukturen, sowohl Daten wie auch Prozesse. In kommerziellen Informationssystemen ist das bei der Mehrheit der Systeme gegeben.
- Notwendigkeit hoher bis sehr hoher Testabdeckung der Fachdomäne. Durch die Ports können Infrastrukturteile (etwa: Datenbank, Verzeichnisdienste) oder externe Schnittstellen hervorragend durch Mock-Implementierungen ersetzt werden – was die Testbarkeit signifikant steigert.

Vorteile:

- Implementierung fachlicher Teile komplett frei von Einflüssen der Infrastruktur.
- Hohe Wartbarkeit und einfache Änderbarkeit des gesamten Systems durch hohe Modularisierung.
- Leichte Austauschbarkeit der Infrastruktur, weil der fachliche Kern des Systems technologieunabhängig ist.

Nachteile:

- Die strikte Einhaltung der Ports- und Adapterstruktur erfordert eine saubere, konsistente Modellierung bzw. Implementierung der Fachdomäne. In bestehenden Systemen fehlt diese klare Trennung meistens.

6.2.4 Architekturstile verteilter Systeme

Ein verteiltes System zeichnet sich durch Zusammenarbeit von Verarbeitungs- und Speicherbausteinen über Kommunikationsnetze hinweg aus. Sowohl Prozesse als auch Benutzer können damit räumlich verteilt auf gemeinsamen oder individuellen Daten arbeiten.

Die Bausteine verteilter Systeme können über unterschiedliche Mechanismen miteinander kommunizieren: einerseits durch direkten und synchronen Aufruf entfernter Bausteine (remote-procedure-call, remote method invocation), andererseits durch indirekten und asynchronen Austausch von Nachrichten oder Events (siehe Abschnitt 6.2.5).

Die wesentliche Ausprägungen verteilter Architekturen sind:

- Client-Server
- Broker
- Service-orientierte Architekturen
- Peer-to-Peer

Manche der in Abschnitt 6.2.3 erläuterten hierarchischen Systeme (Layer, Ports-und-Adapter) werden in der Praxis als verteilte Systeme implementiert, d. h. die Kommunikation der Schichten untereinander findet über Netzwerke statt.

6.2.4.1 Client-Server

Client-Server-Systeme bestehen aus mindestens zwei unabhängigen, miteinander kommunizierenden Prozessen:

- Clients benötigen bestimmte Dienste oder Services, die sie bei einem ihnen bekannten Server erfragen.
- Server stellen Clients bestimmte Dienste bereit. Auf entsprechende Anfragen hin erfüllen sie diese Aufgaben und schicken über das jeweilige Netzwerk die Ergebnisse an den aufrufenden Client zurück.

Die Kommunikation zwischen Client und Server erfolgt synchron – ansonsten klassifiziert man diese Systeme als ereignis- oder nachrichtenbasiert (siehe Abschnitt 6.2.5).

Den Vorteilen der hohen Flexibilität und der klaren Trennung der Verantwortlichkeiten dieses Architekturmodells stehen Nachteile hinsichtlich Sicherheit, höherem Entwicklungsaufwand und vielfältigen Fehlerquellen gegenüber. Dennoch hat sich dieses Modell in vielen Facetten durchgesetzt – insbesondere in den Schichten (siehe 6.2.3.2 sowie 6.2.3.3).

6.2.4.2 Command-Query-Responsibility-Segregation

Von Oliver Wolf

Command-Query-Responsibility-Segregation (CQRS) ist ein Architekturstil, der aktuell ziemlich in Mode ist und einige der als gegeben angenommenen Muster der Softwarearchitektur ziemlich auf den Kopf stellt. Dabei ist die Idee, die CQRS zugrunde liegt, schon alt: Schon in den 1980er-Jahren hat Bertrand Meyer, einer der Väter des Design-by-Contract-Paradigmas und OO-Guru, vorgeschlagen, Methoden[16] danach zu klassifizieren, ob sie den Zustand von Daten verändern (Commands) oder Daten lediglich lesen, ohne dabei Seiteneffekte zu verursachen (Queries).

Diese Unterscheidung ist unter anderem im Hinblick auf parallele Verarbeitung sinnvoll, denn Queries lassen sich beliebig parallelisiert ausführen, ohne sich gegenseitig zu beeinflussen. Außerdem sind Queries idempotent: Im Fehlerfall können sie problemlos erneut ausgeführt werden und liefern stets das gleiche Ergebnis (sofern nicht zwischenzeitlich durch ein Command Daten modifziert wurden).

Bei CQRS wird das Prinzip von CQS nun nicht nur auf eine einzelne Klasse, sondern auf ein System als Ganzes übertragen – „CQS in the large“, gewissermaßen. Ein erster Schritt dabei ist es, zunächst einmal äußere Schnittstellen wie Web-Service-Interfaces, die klassisch meistens eine Kombination von Queries und Commands enthalten (getCustomer(), updateCustomer(), ...), aufzuteilen in Command- und Query-Schnittstellen. Diese getrennten Schnittstellen können dann technisch auch getrennt realisiert werden – im Extremfall in verschiedenen Programmiersprachen, auf physisch getrennten Systemen oder sogar in getrennten Netzen. Besondere Vorteile bringt eine solche Trennung oft dann, wenn zur Laufzeit des Systems einer verhältnismäßig geringen Menge von (eher komplexen) Commands eine große Anzahl schnell abzuarbeitender Queries gegenübersteht. In einem solchen Fall könnten die Queries zum Beispiel von vielen leichtgewichtigen Query-Prozessoren in der Cloud bearbeitet werden, während einige wenige leistungsfähige Rechner im Rechenzentrum die Commands übernehmen. Um zusätzlich Rechenaufwand bei der Verarbeitung von Queries einzusparen, ist es oft sinnvoll, sich außerdem von der Vorstellung eines gemeinsamen Datenbankschemas für Commands und Queries zu lösen und ein spezielles Query-Schema mit bereits voraggregierten oder denormalisierten Daten aufzubauen.

[16] Im Sinne der Objektorientierung, d. h. Methoden von Klassen und Objekten.

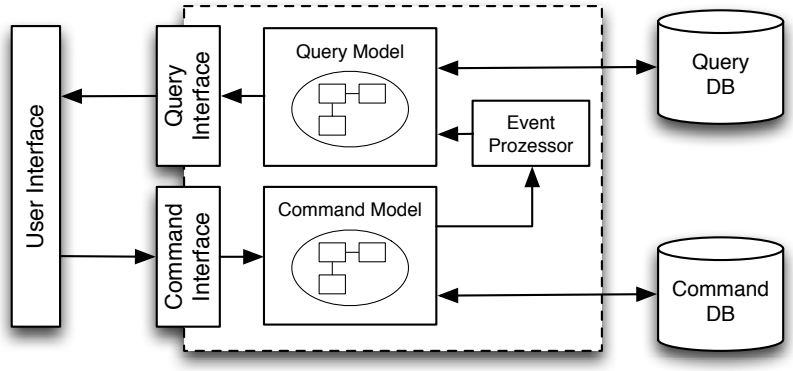


BILD 6.13 CQRS, Command Query Responsibility Segregation

Die Konsistenz der Daten zwischen Query- und Command-Komponenten kann im einfachsten Fall natürlich mit einer gemeinsam genutzten Datenbank (und gegebenenfalls einem Abgleich der Schemas, z. B. über Trigger o. Ä.) erreicht werden. Aber auch das ist in vielen Fällen nicht zwingend notwendig, denn oft sind in realen Anwendungssystemen die Anforderungen an die Datenkonsistenz gar nicht so strikt, wie man auf den ersten Blick vermutet. Typisches Beispiel: Ein Sachbearbeiter ruft in seiner Anwendung die Maske mit den Kundenstammdaten auf, um eine Adresse zu ändern. Nun klingelt das Telefon, und ein Kollege verwickelt ihn in ein Gespräch. Als er endlich anfängt, die neue Adresse einzutragen, sind die Daten auf seinem Bildschirm bereits 10 Minuten alt! In einem solchen System muss bei Schreiboperationen ohnehin immer mit der Möglichkeit von Inkonsistenzen gerechnet und mit Mechanismen wie optimistischem Locking Abhilfe dagegen geschaffen werden. Dementsprechend ist es hier oft auch ausreichend, wenn die Ergebnisse von Commands erst leicht verzögert wirksam und für Queries sichtbar werden („Eventual Consistency"). Für ein CQRS-System hat das eine wichtige Konsequenz – es bedeutet nämlich, dass man durchaus auch völlig getrennte und technologisch unterschiedliche Datenbanken für Commands und Queries verwenden kann. Zur Kommunikation von durch Commands vorgenommenen Datenänderungen dienen dann Events, welche durch einen geeigneten Mechanismus, wie z. B. ein Message-Queuing-System, vom Command- zum Query-Teil übertragen werden. Die damit mögliche Optimierung der Query-Verarbeitung, z. B. durch Verwendung hochskalierbarer NoSQL-Datenbanken, macht häufig den zusätzlichen Aufwand und die höhere Komplexität mehr als wett.

Ein letzter Schritt, der sich noch stärker von traditionellen Anwendungsarchitekturen entfernt, ist das sogenannte Event Sourcing. Hierbei wird praktisch komplett auf die persistente Speicherung von Zuständen fachlicher Entitäten verzichtet. Stattdessen werden die von Commands erzeugten Events, die von der Erzeugung bis zur eventuellen Löschung einer Entität anfallen, mit Zeitstempeln versehen gespeichert. Der aktuelle Zustand kann dann jederzeit durch Wiedereinspielen der Events ab „Stunde null" (in der Regel transient) reproduziert werden. Ein großer Vorteil dabei: Auch der (historische) Zustand zu jedem anderen Zeitpunkt lässt sich so wiederherstellen. Dem gegenüber steht der hohe Speicherbedarf für die Events sowie der Rechenaufwand für die Eventverarbeitung beim Systemanlauf oder beim erstmaligen Zugriff auf eine Entität. Letzterer kann durch regelmäßige Erzeugung von Snapshots (die man wiederum als Events auffassen kann) stark reduziert werden.

Oliver Wolf (oliver.wolf@innoq.com) arbeitet als Principal Consultant bei innoQ. Nach langjähriger Erfahrung in verschiedenen Unternehmen als Lead Architect, Technischer Produktmanager, Entwicklungsleiter und Berater für IT-Sicherheit gilt sein aktuelles Interesse der Architektur hochskalierbarer, verteilter Systeme sowie der engen Verzahnung von Entwicklung und Betrieb („DevOps“).

6.2.4.3 Broker

Die klassische CORBA (Common Object Request Broker Architecture)-Architektur hat in den letzten Jahren meiner Einschätzung nach stark an Bedeutung verloren. Trotz ihrer konzeptionellen Klarheit und der potenziell hohen Flexibilität haben sich CORBA-Systeme in der Praxis kaum durchgesetzt. Ich glaube, die Ideen der Ortstransparenz (*location transparency*) und Unabhängigkeit von Programmiersprache und Netzwerktopologie wurden bei CORBA durch „Design-by-Committee“[17] gründlich sabotiert.

Dennoch möchte ich (und sei es nur aus historischen Gründen, denn mir gefällt die Idee von CORBA ...) die zentralen Begriffe dieses Architekturstils vorstellen – siehe Bild 6.14):

- *Client*: ein Baustein, der einen bestimmten Dienst verwenden möchte, jedoch nur dessen Schnittstellendefinition kennt. Die konkrete Implementierung sowie der Ausführungsort dieses Dienstes sind dem Client unbekannt und egal.
- Ein *Service* kann einen bestimmten Dienst für interessierte Clients erbringen. Technische Details und Ausführungsort der Clients sind dem Service unbekannt und egal.
- Ein Broker koordiniert Aufrufe eines Clients mit angebotenen Diensten eines Service. Er sorgt für die Übergabe aller notwendigen Parameter sowie die Rückgabe der Ergebnisse. Zu seinen Aufgaben gehört es, die konkreten Implementierungen/Ausführungsorte von Services zu finden.
- Stubs (clientseitige Proxies) vermitteln zwischen Client und Broker. Stubs verbergen alle technischen Details der Remote-Kommunikation, beispielsweise Serialisierung und Marshalling/Unmarshalling. Im Falle von CORBA übersetzen sie in das Internet-Inter-Orb-Protocoll, IIOP.
- Skeletons (serverseitige Proxies) vermitteln zwischen Broker und Service. Wie Stubs kapseln sie Details der technischen Kommunikation. In CORBA übersetzen sie IIOP zurück in die Technologie des jeweiligen Service.

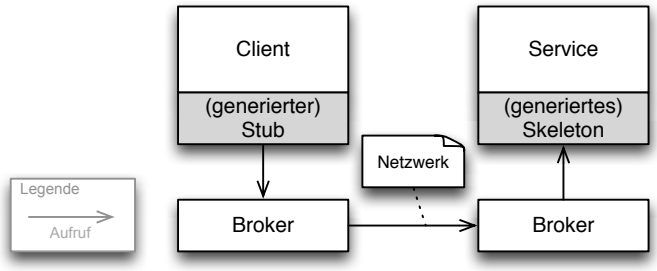


BILD 6.14
(CORBA) Broker Architekturstil

[17] „A camel looks like a horse designed by commitee“, erstmals in der Zeitschrift VOGUE im Jahr 1958 beschrieben.

- Bridges sind (optionale) Verbindungen zwischen unterschiedlichen Brokern, die technische Details deren Implementierung überwinden.[18] Sie würden beispielsweise zwischen Brokern unterschiedlicher Anbieter übersetzen oder zwischen .NET-Remoting, Java- und CORBA-Brokern.

Der Grundansatz der Broker, das Vermitteln von Nachrichten oder Aufrufen zwischen Aufrufern und Anbietern, wird in der Praxis oftmals von nachrichten- oder ereignisbasierter Middleware wahrgenommen.

In CORBA-Broker-Architekturen werden die Schnittstellen von Services mit Hilfe einer Interface-Definition-Language (IDL) beschrieben. Aus dieser IDL generiert ein Broker-spezifischer Compiler dann Stubs und Skeletons, die dann jeweils mit Client und Service zusammenarbeiten. Grundsätzlich können mit diesem Ansatz Clients und Services in unterschiedlichen Programmiersprachen entwickelt werden – die notwendigen (technischen) Konvertierungen zwischen Daten- und Objektformaten übernehmen die Stubs und Skeletons.

6.2.4.4 Peer-to-Peer

Peer-to-Peer-Architekturen bestehen aus gleichberechtigten und über Netzwerke verbundenen Komponenten (Peers), die im Netz gleichzeitig die Rolle von Clients und Servern wahrnehmen beziehungsweise sich *Ressourcen* teilen. Ressourcen in diesem Sinne sind etwa CPU-Zeit, Speicherplatz, aber auch Dateien.

Ein wichtiger Einsatzzweck von P2P-Netzen ist die ausfallsichere Datenverteilung,[19] bekannt durch Netzwerke wie Emule oder Bittorrent. Heute werden auch digitale Telefonate oder Instant-Messaging-Daten über P2P-Netze geleitet.

P2P-Architekturen zeichnen sich durch hohe Ausfallsicherheit aus, weil es keinen *Flaschenhals* für Verfügbarkeit gibt (keinen *single point of failure*): Andererseits ist das Auffinden und Erkennen von Peers in großen Netzen schwierig, ebenso die Fehler- und Ausnahmebehandlung (was geschieht beispielsweise, wenn ein einzelner Peer seine ihm zugewiesene Aufgabe falsch löst?).

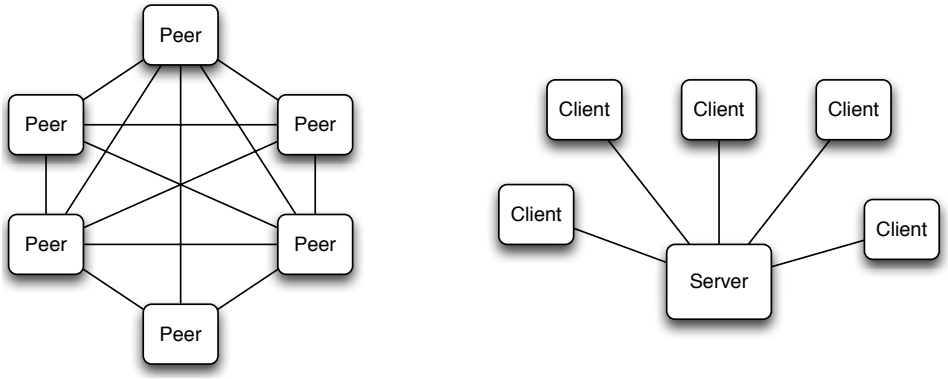


BILD 6.15 Peer-to-Peer versus Client/Server

[18] Diese wundersame Brückenbildung habe ich in der Praxis niemals in realen Systemen erlebt – das scheiterte immer schon an kleinen Beispielen ☹. Aber die Idee klingt super, finde ich ...

[19] So werden beispielsweise umfangreiche Linux-Distributionen häufig durch P2P-Netze weitergegeben.

Für *kleinere* Architekturen besitzt P2P praktisch keine Bedeutung, sondern kommt primär im Internet zum Einsatz.

Kriterien für die Anwendung:

- Teilnehmer (Peers) können vorhandene Ressourcen (Speicher, Prozessor, Netzbandbreite) teilen – einzelne Teilnehmer müssen daher keine beliebig großen Mengen davon auf Vorrat halten.
- Anbieter umfangreicher Downloads verfügen nicht über ausreichende Serverkapazitäten (und können keine der bekannten Download-Plattformen verwenden).
- Hohe bis höchste Ausfallsicherheit ist gefordert.
- Es besteht keine besondere Anforderung an die Schutzwürdigkeit der verarbeiteten Daten (aufgrund der praktisch nicht kontrollierbaren Laufzeitstruktur wäre deren Sicherheit kaum zu gewährleisten).

Vorteile:

- Potenziell höchste Ausfallsicherheit durch hohe Redundanz – sofern eine ausreichend große Zahl von Peers vorhanden ist.
- Peers können sich selbstständig finden – zentrale Administration wird grundsätzlich nicht benötigt. Dieser Vorteil führt jedoch zu einigen Nachteilen – siehe unten.

Nachteile:

- Mangels zentraler Administration könnten Daten durch manipulierte Peers verfälscht oder gelöscht werden. Das lässt sich durch (aufwendige) kryptografische Verfahren erschweren.
- Verteilte Kommunikation in Rechnernetzen erfordert Beschäftigung mit den möglichen Fehlerquellen – insbesondere technischen Details der Rechner-Rechner Kommunikation.

6.2.5 Ereignisbasierte Systeme – Event Systems

Systeme auf Basis von Ereignissen (Events) bestehen grundsätzlich aus zwei Arten von Bausteinen: Ereignisquellen und -senken (*Event-Sources* und *Event-Sinks*). Quellen und Senken kommunizieren in diesem Stil über Ereignisse anstatt über im Quellcode verankerte Abhängigkeiten miteinander. Die Kommunikationsbeziehung arbeitet unidirektional von der Ereignisquelle zur Ereignissenke – diese Beziehungen werden in der Regel erst zur Laufzeit hergestellt. Weil so keine direkten Aufrufe implementiert werden, heißt dieser Stil auch *implicit invocation style*.

6.2.5.1 Ungepufferte Event-Kommunikation

In der *broadcast*-Variante ereignisbasierter Systeme publiziert eine Event-Source Ereignisse auf einem lokalen Netzwerk. Alle verbundenen Empfänger prüfen jedes Ereignis, ob es für den jeweiligen Empfänger interessant ist: Wenn ja, wird das Ereignis bearbeitet, wenn nein, wird es ignoriert.[20]

[20] [Hohpe+04] nennt das „publish-subscribe with reactive filtering“

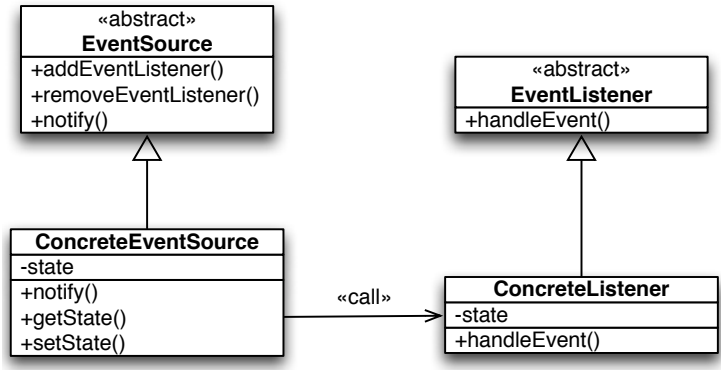


BILD 6.16
Bausteine ungepufferter Event-Kommunikation

Alternativ können sich Empfänger von Ereignissen (*Subscriber*) bei den Sendern (*Publisher*) registrieren. Sender übermitteln Ereignisse in diesem Fall direkt an Subscriber.

6.2.5.2 Message- oder Event-Queue-Architekturen

Die direkte Verbindung zwischen Sendern und Empfängern von Events wird im Architekturstil der „Message- oder Event-Queue-Systeme“ um Puffer (Warteschlangen) ergänzt (siehe Bild 6.17).

Ein Erzeuger von Nachrichten schickt diese an die Queue, aus der Empfänger sie entweder synchron oder asynchron abholen. Diese Struktur wird manchmal als fire-and-forget bezeichnet, weil Erzeuger keine Bestätigung über Erhalt oder Verarbeitung ihrer Nachrichten erhalten. Insbesondere können Erzeuger keine Annahmen darüber treffen, ob ihre Nachrichten überhaupt bearbeitet werden, beziehungsweise in welcher Reihenfolge.

Sind diese Informationen für die Erzeuger wichtig, oder müssen die Empfänger gar explizite Antworten auf ihre jeweiligen Nachrichten geben, so bedarf das Message-Queue-Modell einiger Erweiterungen, etwa über explizte Konversationen oder Korrelations-Ids.[21] Damit können auch mehrere Nachrichten miteinander in Zusammenhang gebracht werden, sodass der ursprüngliche Erzeuger alle Reaktionen auf eine Nachricht nachvollziehen kann.

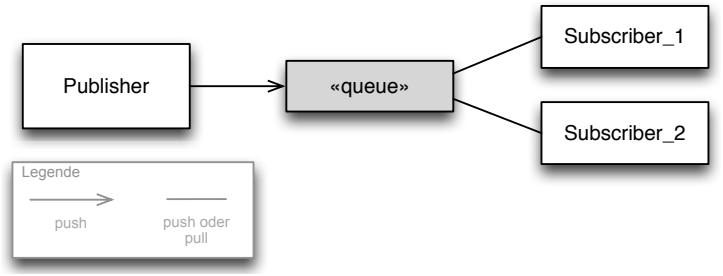


BILD 6.17
Message-Queue: Puffer für Nachrichten und Events

[21] Ich bemühe schon wieder [Hohpe+04]: Er bezeichnet diese Techniken als Conversations und Correlations.

6.2.5.3 Message-Service-Architekturen

Wir können die Aufgaben der Warteschlange des vorigen Abschnitts noch um allgemeine Message-Services erweitern – beispielsweise um folgende:

- Routing von Nachrichten an bestimmte Empfänger, etwa *content-based routing*.
- Ändern der Reihenfolge von Nachrichten – solche mit höherer Priorität werden bevorzugt zugestellt.
- Vorverarbeitung von Nachrichten – etwa Archivierung, Verschlüsselung, Protokollierung, statistische oder sonstige Auswertung.
- Zusammenfassung mehrerer Nachrichten oder Aufteilung von Nachrichten
- Überwachung hinsichtlich bestimmter Qualitätseigenschaften – etwa: Werden die Nachrichten vom Empfänger in der gewünschten Zeit verarbeitet? Ist ein bestimmter Empfänger überlastet und muss eine zusätzliche Instanz gestartet werden?
- Zuverlässige Speicherung von Nachrichten, bis die jeweiligen Empfänger die vollständige Verarbeitung bestätigt haben (*reliable messaging*).

Willkommen in der vielseitigen Welt der *Enterprise Application Integration*, die im Wesentlichen auf dieser Art der entkoppelten Kommunikation basiert. Gregor Hohpe hat mit [Hohpe+04] einen Klassiker zu diesem Thema geschaffen, mit vielen auch heute noch gültigen Patterns.

Kriterien für die Anwendung:

- Erzeuger (Sender) von Nachrichten benötigt für die weitere Verarbeitung keine synchrone Antwort (fire-and-forget).
- Integrationsszenarien: Es müssen mehrere unterschiedliche Systeme zusammenarbeiten.

Sender erzeugt Nachrichten erheblich schneller, als sie Empfänger verarbeiten können. Eine Message-Queue als Puffer kann hier ausgleichen.

Vorteile:

- Sender und Empfänger von Nachrichten können in völlig unterschiedlichen Technologien und Programmiersprachen erstellt werden.
- Message-Queues, insbesondere solche mit zuverlässiger Zustellung (*reliable messaging*), können die Verfügbarkeit und Robustheit von Systemen erheblich steigern. Aus diesem Grund werden MQ-Systeme insbesondere im Bereich Finanz- und Kontodaten häufig eingesetzt.

Nachteile:

- Message-Queue (ob kommerziell oder Open-Source) sind in sich komplexe Systeme mit teilweise hohem Einführungs- und Administrationsaufwand.

Asynchrone und nachrichtenbasierte Programmierung ist signifikant aufwendiger als einfacher call-and-return-Stil. Fehlersuche in asynchronen Systemen kann aufwendig sein.

6.2.6 Interaktionsorientierte Systeme

Viele Informationssysteme, sowohl im Web wie auch auf Desktop- oder Mobilcomputern, basieren auf Interaktion mit ihren (menschlichen) Benutzern. In diesem Abschnitt stelle ich Ihnen zwei der bekanntesten Strukturmuster vor, die solche Mensch-Maschine-Interaktion adressieren.

6.2.6.1 Model-View-Controller

Teile ein interaktives System mit (grafischer) Bedienoberfläche in Bestandteile auf, die über einen Notifizierungsmechanismus (*change propagation*) ihre gegenseitige Konsistenz sichern:

- Das *Model* enthält die (fachlichen) Daten und Funktionen des Systems – und ist völlig unabhängig von der konkreten UI-Technologie. [Buschmann+07] nennt es den *funktionalen Kern* des Systems.
- Benutzer interagieren über *Controller* mit dem System. *Controller* geben Benutzereingaben zur weiteren Bearbeitung an *Model* oder *View* weiter.
- *View* sorgt für die Anzeige der (Model-)Daten auf der grafischen Oberfläche.

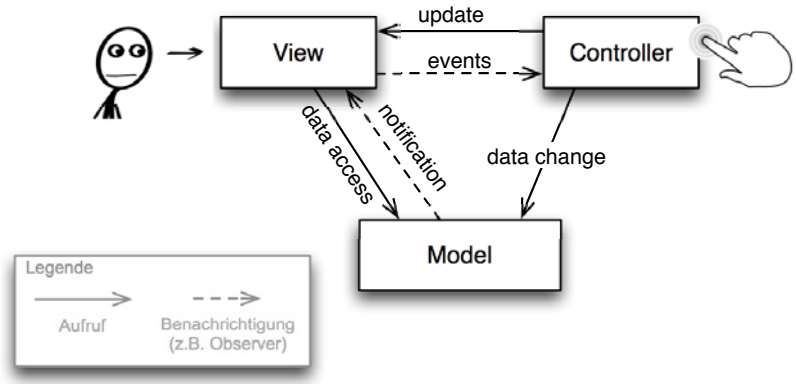


BILD 6.18 Model-View-Controller (Struktur)

Varianten:

Bei der Nutzung interaktiver Anwendungen im Web stellt ein Browser die View dar. Aufgrund des eingesetzten HTTP-Protokolls können Model und Controller jedoch die klassischen Notifizierungsmechanismen, beispielsweise über das Observer-Pattern, nicht verwenden. In diesem Fall wird der Controller entweder in einen Client- und Server-Teil zerlegt[22] bzw. die gesamte Applikation in zwei eigenständige MVC-Triaden. Alternativ bietet sich das Presentation-Model an – siehe den folgenden Abschnitt 6.2.6.2.

Umsetzung in der Praxis:

Egal, in welchem technischen Umfeld Sie Systeme implementieren, ob Web oder Desktop: Wahrscheinlich werden Sie die Details des MVC-Musters durch Ihre konkrete UI-Technologie

[22] Siehe beispielsweise *http://welcome.totheinter.net/2008/08/05/mvc-pattern-for-the-web/*

vorgegeben bekommen. Ob Swing, JavaFX, ASP.NET, iOS, Android oder die vielen Web-UI-Frameworks – praktisch alle implementierten Varianten von MVC und stellen Ihnen als Entwickler dann eine MVC-Infrastruktur bereit.[23] Sie müssen sich dann nur noch ins gemachte Nest setzen ☺ oder mit den Restriktionen dieser Frameworks leben.

Siehe auch Abschnitt 6.2.3.3 (Ports- und Adapter) bezüglich der strikten Kapselung des (Domänen-)Models.

Vorteile:

- Klare Trennung von Verantwortlichkeiten.
- Erlaubt mehrere Views pro Model – durchaus auch in unterschiedlichen Technologien.
- Reichhaltige Auswahl an passenden Frameworks.

Nachteile:

- Implementierung und insbesondere Debugging von Applikationen mit Notifikationen kann aufwendig sein (verwenden Sie daher ausgereifte MVC-Frameworks!).
- Notifikationen von Modellen mit mehreren Controller-View-Paaren kann zeitaufwendig sein, sodass Systeme „langsam“ wirken können. Insbesondere dann, wenn auch einzelne grafische Elemente (Widgets) als eigenständige MVC-Triaden implementiert sind.

6.2.6.2 Presentation-Model

Wie auch das MVC-Muster trennt das *Presentation-Model* die reine Ansicht (View) von der Behandlung von Benutzereingaben (Controller) und den dargestellten Daten (Model). Zwischen dem View-Controller Duo und dem eigentlichen Model entkoppelt das Presentation-Model in zwei Richtungen (siehe Bild 6.19):

1. Presentation-Model repräsentiert Daten und Verhalten eines grafischenViews, ohne Bezug auf eine konkrete UI-Implementierungstechnologie zu nehmen. Beispielsweise könnte es die folgenden Zustände einer CheckBox als Boolesche Werte enthalten:
   - aktiv oder inaktiv (enabled/disabled)
   - ein- oder ausgeschaltet
   - sichtbar oder unsichtbar
2. Presentation-Model übernimmt die Benachrichtigungen des Views über Datenänderungen. Es koordiniert sich mit dem eigentlichen (Domain-)Model.

Das Presentation-Model stellt eine Abstraktion einer grafischen Darstellung dar, ohne selbst Abhängigkeiten von konkreten UI-Technologien zu haben. Es enthält sämtliche Zustandsinformationen, die sich durch Benutzeraktionen ändern könnten.

Es trifft sämtliche Entscheidungen, welche Information wie angezeigt wird – was die Implementierung des Views selbst stark vereinfacht! Das wiederum hat zur Folge, dass automatisierte Tests im Wesentlichen auf das Presentation-Model konzentriert werden können (und die unsäglich schwierigen UI-Tests sich somit drastisch vereinfachen lassen!).

[23] Ausnahmen bestätigen diese Regel – beispielsweise setzt der Open-Dolphin Framework auf das Presentation-Model Muster (siehe Abschnitt 6.2.6.2).

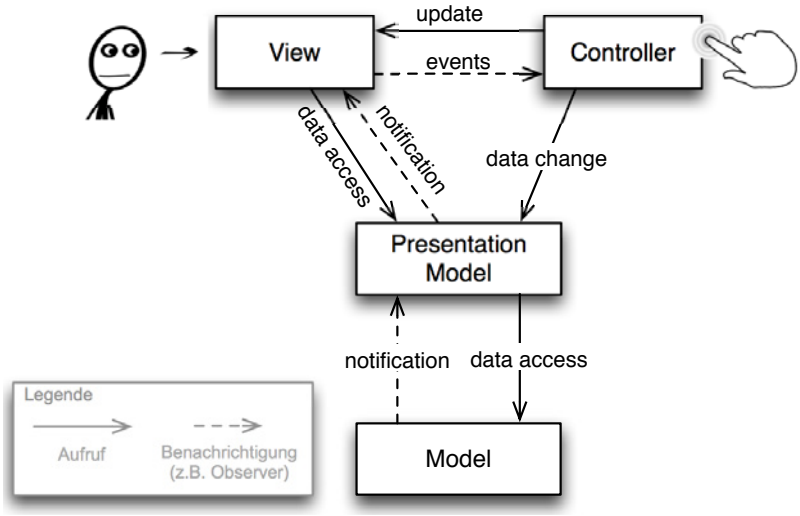


BILD 6.19 Presentation Model (Struktur)

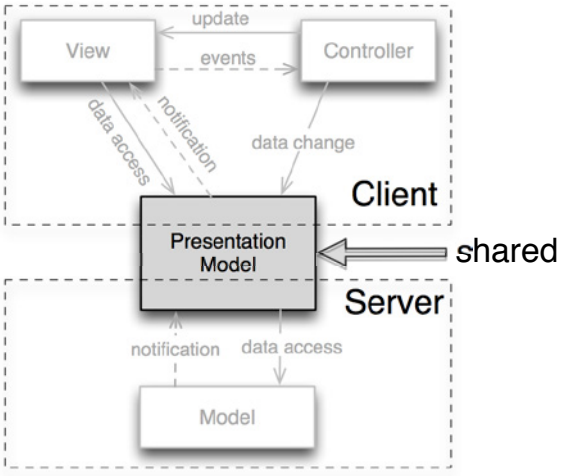


BILD 6.20
Presentation Model, geteilt zwischen Client und Server

Das Presentation Model kommt primär in verteilten Client/Server-Systemen zum Einsatz und wird dabei zwischen Client und Server geteilt (siehe Bild 6.20). Auf der Client-Seite ist es über Notifikationen mit dem View verbunden – diese Abhängigkeit sollte durch einen Framework bereitgestellt werden. Die in den Abbildungen gezeigte Notifikation zwischen Model und Presentation-Model ist optional.

Kriterien für die Anwendung:

- Falls Sie das Verhalten Ihrer grafischen Oberfläche ausgiebig automatisiert testen wollen, kann Ihnen das Presentation-Model helfen: Da es den gesamten Zustand der Oberfläche hält und koordiniert, genügt es oftmals, das Presentation-Model zu testen.

Sie möchten eine „Rich Internet Application“ entwickeln, die aus Masken mit vielfältigen Benutzereingaben sowie möglichst unmittelbaren Validierungen besteht. Statt den View-Code mit diesen Aufgaben zu belasten, können Sie das Presentation-Model gut dafür verwenden.

Vorteile:

- Das eigentliche Model kann ein reines Domänen-Model bleiben (POJO oder POCO)[24] – ist somit einfacher und i. d. R. leichter testbar.
- Der View, die eigentliche Darstellung, bleibt frei von fachlicher Logik oder Prüfungen, weil diese komplett im *Presentation-Model* liegen.

Nachteile:

- Sie benötigen einen Synchronisations- oder Notifikationsmechanismus zwischen *Presentation-Model* und View (den Sie allerdings bei interaktiven grafischen Systemen praktisch immer benötigen).

Quellen und weitere Informationen:

- Der quelloffene Java UI-Framework openDolphin (*http://open-dolphin.org*) basiert auf dem *Presentation-Model*-Muster – er adressiert grafische Benutzerschnittstellen für Unternehmensanwendungen.
- Martin Fowler hat dieses Architekturmuster unter *http://martinfowler.com/eaaDev/PresentationModel.html* beschrieben. Er bezieht sich dabei auf Vorlagen von VisualWorks-Smalltalk.

Andere Muster für interaktive Systeme

Martin Fowler erläutert viele weitere Muster für interaktive Systeme[25]:

Supervising Controller (*Presenter*)	Wie bei MVC erfolgt auch hier eine Dreiteilung in Daten (Model), Anzeige (View) und Eingabeverarbeitung/Koordination (Presenter). Im Gegensatz zu MVC hat der View keine Beziehung zum Presenter. Benutzereingaben werden von UI-Elementen (Widgets) unmittelbar zur Verarbeitung an den Presenter delegiert.*
Passive View	Im Gegensatz zu MVC ist der View-Teil hier völlig passiv, d. h. kümmert sich nicht um die Aktualisierung von angezeigten Daten aus dem Model. Diese Aufgabe obliegt hier dem Controller – der damit zum wesentlichen Gegenstand von Tests wird –, weil im View praktisch „nichts Schlimmes“ passieren kann.

* Diese Struktur ist auch als „Model-View-Presenter“ bekannt, siehe z. B. *http://de.slideshare.net/esug/twisting-thetriad*

[24] Plain Old {Java | C#} Object
[25] Auf *http://martinfowler.com/eaaDev* beziehungsweise [Fowler02]

6.2.7 Weitere Architekturstile und -muster

Eine Vielzahl weiterer Architekturmuster könnte Ihnen sicherlich in manchen Fällen die Entwurfstätigkeit erleichtern. Die Bücher „Pattern Languages of Program Design“ (PloP) enthalten dazu viele Anregungen, insbesondere die „Pattern-Language for Distributed Computing“ [Buschmann+07]. Außerdem bietet Martin Fowlers „Patterns of Enterprise Application Architecture“ [Fowler02] einen großen Fundus an Strukturvorschlägen.

REST Architektur

Von Stefan Tilkov

In seiner berühmten Dissertation hat der Apache-Mitgründer und HTTP-Miterfinder Roy Fielding diesen Stil als die „Architektur des Web“ bezeichnet und dazu im Nachhinein die Konzepte beschrieben, die Grundlage der Standardisierung der Webprotokolle (vor allem HTTP und URI) waren.[26] Rein formal ist damit das Web als Ganzes die einzig relevante Instanz dieses Stils! In der Praxis interessiert man sich für diese Unterscheidung jedoch herzlich wenig und benutzt die Bezeichnung „RESTful“ für konkrete Architekturen, die Webtechnologien so benutzen, wie es die Prinzipien des Architekturstils vorschreiben.

REST steht für *representational state transfer* und wird durch folgende Regeln definiert:

- REST impliziert eine Client/Server-Architektur. Der Server ist dabei eine Webanwendung; der Client kann entweder ein Webbrowser sein oder eine andere Art *user agent*, sei es ein generischer oder ein von Ihnen entwickeltes Programm.
- Der Server stellt seine Fähigkeiten nach außen nicht als eine Menge von Diensten und/oder Operationen, sondern als *Ressourcen* zur Verfügung, die über einen einheitlichen, standardisierten Mechanismus (eine URI) identifiziert werden. Dies bedeutet, dass in einem REST-System nicht nur Ihre Anwendungen oder Ihre Dienste eine URI bekommen, sondern alle fachlichen Konzepte, die Ihnen wichtig sind.[27]
- Die Kommunikation des Clients mit den Ressourcen erfolgt mit Hilfe von Standard-Methoden (im Web im Wesentlichen GET, PUT, POST, DELETE) und nicht mit anwendungsspezifischen Operationen. Anders formuliert: Sie benutzen das Standard-Applikationsprotokoll HTTP und erfinden nicht Ihr eigenes.
- Auf Basis der standardisierten Semantik der Operationen können zwischen Client und Server intelligente, generische Zwischenstationen (*intermediaries*) geschaltet werden, die nützliche Dinge tun. Bestes Beispiel dafür sind *Caches*, die anhand von Metadaten, die als Teil von HTTP-Antworten vom Server zurückgeliefert werden, eine Vielzahl von Requests vermeiden können.
- Die Kommunikation erfolgt mit Hilfe von *Repräsentationen* der Ressourcen. Davon kann es für jede Ressource mehr als eine geben, z. B. eine JSON-, eine XML- und eine HTML-Variante.

---

[26] Mit dieser Fußnote errege ich wahrscheinlich die Missgunst aller RESTafarians: Ich halte REST primär für einen Kommunikationsstil – und eine Untermenge der hierarchischen oder verteilten Systeme. Trotz dieser kleinen begrifflichen Meinungsverschiedenheit: REST ist für moderne, interaktive Web-Anwendungen zentral – und jeder Webentwickler sollte seine Grundzüge kennen!

[27] Woher auch rührt, dass Ansätze wie SOAP/WSDL/WS-*-Webservices oder Frameworks wie JSF von REST-Verfechtern nur als Negativbeispiel eingesetzt werden

- In einer „echten“ REST-Architektur wird darüber hinaus der Kontrollfluss über *Hypermedia* gesteuert. Dahinter verbirgt sich trotz des hochtrabenden Namens nicht mehr als ein aus interaktiven HTML-Anwendungen bekanntes Konzept: Der Server teilt dem Client in der Antwort auf seine Anfragen jeweils über *Hypermedia-Controls* (Links, Formulare usw.) mit, welche Interaktionen als Nächstes möglich sind. Der ideale REST-Client benötigt daher nur eine einzige URI als Einstieg und erfährt von allen anderen Ressourcen nur dynamisch.

Ganz ähnlich, wie eine relationale Datenbank dann am besten funktioniert, wenn Sie sie so benutzen, wie es sich die Erfinder der relationalen Algebra gedacht haben, können Webtechnologien ihre Stärken am besten ausspielen, wenn sie REST-Prinzipien folgen. Vor einigen Jahren war REST noch ein relativ exotischer Ansatz einiger Außenseiter, hat sich in den letzten Jahren aber zu einer ernst zu nehmenden Alternative zu SOAP/WSDL-basierten Architekturen, im Webumfeld gar zur Standardlösung entwickelt.

Varianten:

- Wie die Formulierung „echte REST-Architektur“ weiter oben schon andeutet, gibt es Varianten, die nicht alle Regeln befolgen. In den meisten populären REST-Schnittstellen wird der Hypermedia-Aspekt ignoriert (es gibt einige Reifegradmodelle, die auf diese Unterscheidung näher eingehen; vom Erfinder des Konzeptes kommen jedoch klare Worte: „REST-APIs must be hypertext-driven“). In solchen Systemen listet die Dokumentation eine Menge an URI-Mustern mit Parametern auf; es ist Aufgabe des Aufrufers, diese mit Werten zu füllen und dann entsprechend der Dokumentation mit den korrekten HTTP-Verben aufzurufen.
- REST wird in der Regel als Architekturstil für Services betrachtet, ist aber mindestens genauso sinnvoll für Webanwendungen, bei denen der Aufrufer ein von einem Menschen bedienter Webbrowser ist. Dazu wird die Oberfläche moderner Webanwendungen entweder vollständig in JavaScript realisiert und HTML auf dem Client erzeugt (sog. Single Page Applications, SPAs) oder ein klassischerer Ansatz gewählt, bei dem HTML auf dem Server erzeugt und JavaScript optional und nur für die Verbesserung der Ergonomie verwendet wird (Resource-oriented Client Architecture, ROCA). In beiden Varianten kann der Server nach REST-Prinzipien realisiert werden.

Kriterien für die Anwendung:

- Die Interoperabilität, also die Möglichkeit, aus bekannten und unbekannten Ökosystemen auf ein REST-System zuzugreifen, ist größer als bei praktisch jedem anderen Ansatz, schließlich ist nichts weiter verbreitet als Webtechnologien. Dass Sie ein Web-API nach REST-Prinzipien umsetzen sollten, versteht sich fast von selbst.
- REST-Architekturen sind auch im unternehmensinternen Integrationsumfeld besonders gut geeignet, wenn Sie sich nicht von einzelnen Herstellern oder Produkten abhängig machen wollen.

Vorteile:

- Sie können die weit verbreiteten und ausgereiften Werkzeuge und Bibliotheken, die im Web zum Einsatz kommen, optimal ausnutzen.
- Durch die statuslose Kommunikation skalieren REST-Systeme besonders gut.

- Die Kopplung zwischen den Kommunikationspartnern in einem REST-Setup ist besonders gering, die Möglichkeiten einer Evolution von Schnittstellen sind in besonderem Maße auf Langlebigkeit ausgelegt.

Nachteile:

- Wie immer sind lose Kopplung, Standardisierung und Skalierbarkeit nicht umsonst. So müssen Sie sich auch beim Einsatz von RESTful HTTP bewusst sein, dass Sie Nachrichten über das Netz senden und keine lokalen Methoden aufrufen; und auch die Indirektion über Hypermedia kostet etwas.

Quellen und weitere Informationen:

Die wesentliche Grundlage für REST legt Roy Fielding in seiner Dissertation[28]. Ansonsten stellt [Tilkov-11] den Zusammenhang zwischen REST, HTTP und deren praktischer Anwendung dar.

Falls Sie zu HTTP noch mehr wissen möchten – dessen Spezifikation sagt alles: RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, *http://www.ietf.org/rfc/rfc2616.txt*

*Stefan Tilkov (*stefan.tilkov@innoq.com*) ist Geschäftsführer und Principal Consultant der innoQ Deutschland GmbH, wo er sich vorwiegend mit der strategischen Beratung im Umfeld von Softwarearchitekturen beschäftigt. Er ist Autor des Buchs „REST und HTTP" (dpunkt Verlag), Autor zahlreicher Fachartikel und häufiger Sprecher auf internationalen Konferenzen.*

6.3 Heuristiken zum Entwurf

Nach dem Ausflug zu den Architekturmustern möchte ich nun zu grundlegenden Entwurfsregeln oder -heuristiken kommen. Sämtliche der hier vorgestellten Entwurfsprinzipien sind seit Langem dokumentiert. Ich stelle bei der Analyse bestehender Systeme jedoch immer wieder mit Schrecken fest, dass sie in der Praxis oft sträflich vernachlässigt werden.

6.3.1 Das So-einfach-wie-möglich-Prinzip

KISS: Keep it simple

Einfachheit oder Schlichtheit sind wünschenswerte Eigenschaften eines jeglichen Softwareentwurfs. (Es sei denn, Sie möchten verhindern, dass andere Menschen diesen Entwurf verstehen.)

Entwerfen Sie nach dem Prinzip „So einfach wie möglich". Fragen Sie bei jeder, aber wirklich jeder Entwurfsentscheidung immer wieder: Wie geht es einfacher? Dieser Ratschlag gilt für alle Ebenen des Entwurfs, für Geschäftsmodelle, Architekturen und Klassendiagramme.

[28] Fielding, Roy Thomas: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000, *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*

Schlichtheit und Einfachheit erleichtern all jenen, die Ihre Entwürfe später lesen, das Verständnis. Sie vermeiden auf diese Weise, dass potenzielle Probleme durch überflüssige Komplexität verdeckt werden. Einfachheit hilft, Probleme bereits im Vorfeld zu erkennen.

Welche Komponenten Ihres Systems werden garantiert die wenigsten Fehler enthalten? Diejenigen, die Sie durch sinnvolle Vereinfachung weglassen konnten.

Angemessen komplex entwerfen

Das So-einfach-wie-möglich-Prinzip hängt eng mit dem Begriff der Angemessenheit zusammen:

Setzen Sie Komplexität in dem Maße ein, wie es Ihnen für den Sachverhalt angemessen erscheint. Im Zweifelsfall wählen Sie die weniger komplexe Alternative.

Für „Angemessenheit" kann ich Ihnen keine Faustregel oder Metrik angeben. Dieses Kapitel stellt einige Prinzipien vor, die Ihnen bei der Entscheidung für angemessene Entwurfsalternativen helfen können.

Ich habe in Projekten Architekturdiagramme gesehen, die auf tapetengroßen Ausdrucken mehr als hundert Module gleichzeitig darstellten. Mit solchen Ungetümen können Sie bestenfalls Ihren Chef beeindrucken und schlimmstenfalls neue Mitarbeiter abschrecken. Als Entwurfsdokumentation für Projekte eignen sie sich mit Sicherheit nicht.

Eine Faustregel der kognitiven Psychologie (die sich seit vielen Jahren auch bei Systemanalytikern etabliert hat) besagt, dass Diagramme zwischen fünf und neun Komponenten enthalten sollen. Daran müssen Sie sich nicht sklavisch halten. Sie sollten aber darauf achten, eine angemessene (schon wieder dieser schwammige, ungenaue Begriff!) Komplexität in Entwürfen zu erreichen.

Einfachheit und Agilität hängen eng miteinander zusammen

Einfache und schlichte Strukturen sind leichter verständlich und dadurch leichter veränderbar. Abhängigkeiten werden leichter erkannt und lassen sich leichter beseitigen.

In agilen Entwicklungsprozessen werden Einfachheit und Schlichtheit zu einer grundlegenden Maxime erhoben. Agile Prozesse funktionieren hochgradig iterativ. In kurzen Intervallen werden die aktuellen Kundenbedürfnisse mit den Vorgaben der Projekte abgeglichen. Dadurch entstehen Systeme, die an den aktuellen Bedürfnissen orientiert sind (und nicht an den Bedürfnissen von vor drei Jahren, als das Projekt gestartet wurde!).

Agile Prozesse versuchen, die Softwareentwicklung durch bedarfsgerechte, angemessene Maßnahmen bei allen Entwicklungsaktivitäten zu verbessern. Sie priorisieren Einfachheit gegenüber Funktionsvielfalt.

6.3.2 Entwerfen Sie nach Verantwortlichkeiten

- Eine wichtige Regel des effektiven Entwurfs besteht darin, sich jeweils auf spezifische Verantwortlichkeiten, Zuständigkeiten oder Sachverhalte zu konzentrieren.
- Unterteilen Sie komplexe Sachverhalte in mehrere einfache, die jeweils einen einzelnen Aspekt enthalten oder modellieren. Setzen Sie gegebenenfalls Hierarchien oder zusätzliche Abstraktionen ein.
- Nennen Sie Verantwortlichkeiten beim Namen! Beschreiben Sie sie in kurzen Sätzen.

Verantwortlichkeit sollten Sie auf allen Ebenen des Entwurfs beachten, für einzelne Klassen in objektorientierten Entwürfen genauso wie für ganze Systeme und Subsysteme.

Verantwortlichkeiten gibt es für „Wissen“ (*knowing*) und „Handeln“ (*doing*):

- *Wissen:* Eine Komponente ist für eine bestimmte Menge an Information verantwortlich.
- *Handeln:* Eine Komponente ist für bestimmte Aktivitäten verantwortlich (etwa: Steuern, Kontrollieren, Berechnen, Erzeugen).

Wenn Sie Verantwortlichkeiten an Komponenten delegieren, müssen Sie die Auswirkungen dieser Verantwortung berücksichtigen. Insbesondere haben Verantwortlichkeiten große Auswirkungen auf die Kopplung und Kohäsion von Komponenten untereinander. Diese Begriffe diskutiere ich in Abschnitt 6.4, in dem es um Abhängigkeiten von Bausteinen geht.

Trennung von Technik und Fachlichkeit

Besonders wichtig ist die Trennung technischer und anwendungsspezifischer (d. h. fachlicher) Teile des Systems. Halten Sie diese beiden Aspekte in Ihren Entwürfen strikt getrennt, um später sowohl Technik als auch Fachlichkeit getrennt voneinander weiterentwickeln zu können.

Die strikte Trennung dieser unterschiedlichen Aspekte des Systems erfüllt die Forderungen nach geringer Kopplung und hoher Kohäsion (siehe Abschnitte 6.4.1 ff.). Sie erleichtert die Arbeitsaufteilung, weil sich Teams dadurch auf bestimmte Aspekte konzentrieren und spezialisieren können.

Modularität

Die Modularität eines Systems besagt, ob und wie es in eine Menge jeweils in sich geschlossener Bausteine (Module) zerlegt wurde. Jedes Modul sollte bestimmte Verantwortlichkeiten kapseln, die über wohldefinierte Schnittstellen zugänglich sind. Die Bausteine eines modularen Systems sollten Black-Boxes sein, deren Innenleben möglichst verborgen bleibt: es genügt, die Schnittstelle von Bausteinen zu kennen.

Modularität hat eine Reihe bestechender Vorteile: Modulare Bausteine

- können getrennt voneinander entwickelt,
- unabhängig voneinander geändert und
- ohne Nebenwirkungen durch andere Bausteine mit identischer Schnittstelle ausgetauscht werden.

6.3.3 Konzentrieren Sie sich auf Schnittstellen

Das Ganze ist mehr als die Summe seiner Teile. Dieses „Mehr“ resultiert aus den gegenseitigen Beziehungen zwischen den Teilen des Systems. Diese Beziehungen werden durch Schnittstellen ermöglicht.

Von technischen Details abgesehen, stecken die wichtigsten Aspekte für Architekten in den Schnittstellen und Beziehungen zwischen den Komponenten ([Rechtin2000]). Das hat mehrere Gründe:

- Aus diesen Beziehungen resultiert, wie oben gesagt, der Mehrwert des Systems gegenüber den Einzelkomponenten. Die Beziehungen ermöglichen die einzigartige Funktion des Systems. Keine Komponente kann (auf Systemebene) die gewünschte Funktionalität allein bieten.
- Die einzelnen Komponenten oder Teilsysteme kommunizieren und kooperieren über Schnittstellen miteinander. Diese Zusammenarbeit bildet die Basis des Gesamtsystems.
- Die Spezialisten für einzelne Teilsysteme konzentrieren sich auf ihre (lokalen) Probleme. Sie betrachten die übrigen Systeme oder Komponenten als „Peripherie“. Sie als Architekt halten die Fäden des Systems zusammen – eben über die Schnittstellen.
- Über Schnittstellen findet auch die Kommunikation mit der Außenwelt statt – ohne die jedes System nutzlos bleibt.

6.3.4 Berücksichtigen Sie Fehler

Sie sollten bei Entwürfen daran denken, dass sich Fehler praktisch nicht vermeiden lassen. Dies gilt gleichermaßen für Analyse-, Entwurfs-, Implementierungs- und Bedienungsfehler. Einige Tipps dazu:

- Sorgen Sie dafür, dass Fehler leicht gefunden werden können. Streben Sie in Entwürfen nach Einfachheit, Schlichtheit und Verständlichkeit.
- Halten Sie Auswirkungen von Fehlern möglichst lokal. Vermeiden Sie Störungen des Gesamtsystems nach dem Eintritt von Fehlersituationen.
- Streben Sie Robustheit an. Falsche Eingaben (garbage-in) dürfen niemals zu falschen Ausgaben (garbage-out) führen.
- Sorgen Sie dafür, dass alle beteiligten Entwickler diese Tipps beherzigen. Robuste Entwürfe sind ohne robusten Quellcode nutzlos.

6.4 Optimieren von Abhängigkeiten

Erst Beziehungen zwischen Bausteinen oder Komponenten ermöglichen deren Zusammenarbeit, bilden also die Grundsubstanz kooperierender Strukturen von Software-Bausteinen. Andererseits bedeuten Beziehungen auch *Abhängigkeiten* zwischen Bausteinen. Diese Abhängigkeiten können jedoch auch zu einem gravierenden Problem führen:

Änderungen an einem Baustein führen zu Konsequenzen an anderen Stellen: Ändern Sie beispielsweise eine Schnittstelle, die ein anderer Baustein verwendet, so müssen Sie den nutzenden Baustein möglicherweise ebenfalls ändern.

Beziehungen zwischen Bausteinen implizieren Abhängigkeit dieser Bausteine voneinander – manchmal auch Kopplung genannt. Weiter vorne, in Abschnitt 6.1.2, habe ich Ihnen geraten, die Kopplung zwischen Bausteinen möglichst lose zu gestalten – nun möchte ich diesen Ratschlag etwas präzisieren. Aber lassen Sie mich zuerst mal ein generelles Problem von Software schildern, die über längere Zeit geändert (oder „gewartet“) wird.

Managen Sie Abhängigkeiten! Beobachten (= messen) Sie Abhängigkeiten in Ihren Systemen durch automatisierte Codeanalyse, beobachten Sie Veränderungen über die Zeit.

Bewerten Sie kontinuierlich die Notwendigkeit von Abhängigkeiten.

Struktur von Software degeneriert mit der Zeit

Viele Abhängigkeiten lassen Design verfaulen

Architekten und Designer haben zuerst eine saubere und flexible Struktur für ein Softwaresystem entworfen. Auch in der ersten Version der Software ist diese Struktur erkennbar. Dann jedoch kommt die Software zum praktischen Einsatz, und es geschieht das Unvermeidliche: Anforderungen ändern sich. Die Software wird angepasst, verändert, erweitert, gepflegt. Manchmal übersteht die ursprüngliche Struktur diese Änderungen. In den meisten Fällen entsteht über die Zeit jedoch ein schwer verständlicher, unwartbarer Moloch ohne erkennbare Struktur. Änderungen sind schwierig und führen zu Problemen an völlig anderen Stellen. Das System wird labil, Erweiterungen werden immer riskanter. Robert C. Martin bezeichnet in [Martin2000] diesen Zustand als „verfaultes Design“.

Architektur muss Rahmen für Änderungen sein

Diese verbreitete Entwicklung resultiert daraus, dass der ursprüngliche Entwurf nicht berücksichtigt hat, dass später neue und geänderte Anforderungen hinzukommen können. Die Architektur war eben kein „Rahmenwerk für zukünftige Änderungen“, wie der Methodenguru Tom DeMarco es von guten Architekturen fordert.

Symptome von verfaultem Design

Es sind im Wesentlichen drei Eigenschaften von Software, die auf ein verfaultes Design hinweisen:

- *Starrheit:* Selbst einfache Änderungen an der Software sind schwierig und bedingen Modifikationen in einer Vielzahl abhängiger Komponenten.
- *Zerbrechlichkeit:* Änderungen an einer Stelle des Programms führen zu Fehlern an ganz anderen Stellen.
- *Schlechte Wiederverwendbarkeit:* Selbst innerhalb eines Projektes können Komponenten kaum wieder verwendet werden, weil sie zu viele Abhängigkeiten von anderen Programmteilen enthalten.

Entwurf muss Wandel ermöglichen

Die Ursache dieser (in der Praxis verbreiteten Probleme) liegt in den ständigen Änderungen der Anforderungen an Software begründet. Alle Stakeholder wissen, dass sich Anforderungen ändern!

> *Wenn unsere Entwürfe aufgrund der permanent wechselnden Anforderungen versagen, so ist das der Fehler unserer Entwürfe!*
> *Wir müssen einen Weg finden, unsere Entwürfe unverwüstlich gegen solche Änderungen zu machen und sie vor Verfaulen zu schützen.*
>
> [Martin2000, p. 4]

Abhängigkeiten verwalten

Unsere Herausforderung liegt darin, die gegenseitigen Abhängigkeiten aller Bestandteile von Software (Subsysteme, Komponenten, Pakete, Klassen, Funktionen) in einer vernünftigen Weise zu verwalten. Manche Abhängigkeiten zwischen Modulen lassen sich nicht vermeiden oder sind sogar erwünscht (etwa: einer anderen Klasse muss eine bestimmte Nachricht geschickt werden, eine bestimmte Funktion eines anderen Subsystems muss aufgerufen werden).

Wir müssen in Entwürfen die Abhängigkeiten zwischen Komponenten so konstruieren, dass künftige Änderungen keine neuen Abhängigkeiten erzeugen.

Dabei gilt es, einige fundamentale Prinzipien zu berücksichtigen, die ich Ihnen in den nachfolgenden Abschnitten vorstellen möchte. Diese Prinzipien beziehen sich vordergründig auf den Entwurf objektorientierter Strukturen. Sie sollten diese Grundsätze jedoch auch bei anderen Betrachtungseinheiten beherzigen, weil sie Ihre Entwürfe erheblich stabiler gegenüber künftigen Änderungen machen. Auf der Ebene von Softwarearchitekturen helfen diese Prinzipien, die Wartbarkeit und Erweiterbarkeit Ihrer Systeme zu verbessern.

Als zusätzliches Hilfsmittel, mit dem Sie Abhängigkeiten in Ihren Entwürfen und Architekturen systematisch reduzieren können, haben sich einige Entwurfsmuster[29] (*design patterns*) erwiesen. Einige davon lernen Sie in Abschnitt 6.5 kennen.

[29] Entwurfsmuster sind größtenteils für objektorientierte Systeme publiziert. Das Verständnis dieser Muster übt jedoch unserer Erfahrung nach einen positiven Einfluss auf beliebige Softwaresysteme aus (siehe insbesondere [Gamma95] und [Buschmann+96]).

Arten der Kopplung

Die Kopplung von Bausteinen ist ein Maß für die zwischen diesen Bausteinen bestehenden Abhängigkeiten. Eine geringe Kopplung erfüllt die Forderung nach Einfachheit (siehe Abschnitt 6.3.1) und ist eine wichtige Voraussetzung für die Flexibilität und die Verständlichkeit von Entwürfen. Hohe Kopplung bedeutet gleichzeitig hohe Komplexität und Starrheit der Entwürfe.

Falls Sie die Forderung nach minimalen Abhängigkeiten wörtlich nehmen und zwischen Ihren Bausteinen überhaupt keine Abhängigkeiten bestehen, verhindern Sie damit deren Zusammenarbeit ... Das wäre ja auch nicht im Sinne der Erfinder – gefragt ist also ein *gesundes* Maß an Abhängigkeit oder Kopplung.

Folgende Arten der Kopplung werden Ihnen in der Praxis oft begegnen:

- Kopplung durch Aufruf: Ein Baustein benutzt direkt einen anderen Baustein, ruft beispielsweise eine Funktion oder Methode des benutzten Bausteins auf.
- Kopplung durch Erzeugung: Ein Baustein erzeugt einen anderen (oder eine Instanz einer Klasse von anderen Bausteinen). Diese Art der Abhängigkeit können Sie durch Fabrik-Muster (*Factory-Pattern*, siehe [GoF] und [Eilebrecht+06]) verringern, in modernen Systemen auch durch *Dependency Injection*[30] (siehe auch Abschnitt 6.4.8).
- Kopplung durch Daten oder Datenstrukturen: Das kann beispielsweise ein Parameter eines Funktions- oder Methodenaufrufs sein, oder aber eine Datenbankstruktur. In serviceorientierten Architekturen trifft diese Art der Kopplung auch auf sogenannte Dokumentstrukturen zu.
- Kopplung über Hardware oder Laufzeitumgebung trifft zu, wenn Bausteine beispielsweise im gleichen Adressraum oder innerhalb der gleichen virtuellen Maschine oder des gleichen Netzwerksegmentes ablaufen müssen.
- Kopplung über die Zeit: Falls für Aktivitäten verschiedener Bausteine eine notwendige Reihenfolge besteht, beispielsweise eine Aktion eines Bausteins immer zeitlich *vor* einer anderen Aktion eines anderen Bausteins erfolgen muss.

6.4.1 Streben Sie nach loser Kopplung

To loosely couple systems, don't connect them.

David Orchard, BEA-Systems

Die oben geforderte Wandlungsfähigkeit unserer Entwürfe beruht in hohem Maße darauf, wie eng die Bausteine eines Systems voneinander abhängen. Wenn es Ihnen gelingt, die Kopplung einzelner Bausteine lose zu gestalten, können Sie diese Bausteine unabhängig voneinander variieren – Ihr Gesamtsystem bleibt wandlungsfähig und flexibel.

[30] Perfektioniert beispielsweise in Frameworks wie Spring (*www.springframework.org*). Eine einführende Motivation und Erklärung hat Martin Fowler online beschrieben: *http://martinfowler.com/articles/injection.html*

Sie sollten Abhängigkeiten zwischen Bausteinen bewusst und vorsichtig einsetzen. Erinnern Sie sich stets daran, dass ein Übermaß an Abhängigkeiten die Flexibilität Ihres Systems drastisch reduziert und zu *verfaultem Design* führen kann.

Kopplung von Systemen können Sie ausgezeichnet auf Quellcode-Ebene kontrollieren. Viele Werkzeuge* können entsprechende Metriken ermitteln. Setzen Sie solche Metriken ein, und beobachten Sie kontinuierlich die Kopplung Ihrer Systembausteine. Falls die Kopplung während der Entwicklung signifikant ansteigt, sollten Sie durch Zerlegung und Umstrukturierung gegensteuern.

* Suchen Sie im Internet nach „Dependency Analyzer".

6.4.2 Hohe Kohäsion

Die Kohäsion, auf Deutsch „Zusammenhangskraft", eines Architekturbausteins gibt an, wie sehr dessen Elemente inhaltlich zusammengehören. Es ist sinnvoll, in Entwürfen nach hoher Kohäsion, das heißt nach hohem Zusammenhang von Bausteinen zu streben: Zusammen, was zusammengehört.

Möglichst hohe Kohäsion

In Kapitel 3 bzw. Abschnitt 6.1.3 hatte ich Sie bereits aufgefordert, Verantwortlichkeiten, insbesondere fachliche und technische Aspekte, in Entwürfen zu trennen. Das war ein Vorgriff auf die Forderung nach hoher Kohäsion: Fachliche und technische Aspekte besitzen in der Regel kaum inhaltliche Zusammenhänge und gehören daher getrennt. Perfekt modelliert wird das im Ports-und-Adapter-Architekturmuster aus Abschnitt 6.2.3.3 (siehe insbesondere Bild 6.11).

6.4.3 Offen für Erweiterungen, geschlossen für Änderungen

Komponenten sollen offen für Erweiterung sein, aber geschlossen für Änderungen (Offen-Geschlossen-Prinzip, OGP)*

* Bei der Benennung dieses Prinzips hat Robert C. Martin Pate gestanden.

OGP ist Ziel jeder Architektur!

Komponenten sollten grundsätzlich so beschaffen sein, dass künftige Erweiterungen ohne Änderungen des Quellcodes möglich werden. Egal, ob Sie Klassen, Pakete oder andere Strukturen entwerfen: Wenn Komponenten dem Offen/Geschlossen-Prinzip (OGP) genügen, können Sie neue und zusätzliche Eigenschaften hinzufügen, ohne bestehenden Code modifizieren zu müssen. OGP stellt sicher, dass künftige Erweiterungen ohne Nebenwirkungen funktionieren, weil sich Änderungen nicht im bestehenden und funktionierenden Code fortpflanzen können!

Wenn Sie funktionierenden Code ohne Änderung beibehalten, werden alle Komponenten, die diesen Code nutzen, auch weiterhin so arbeiten wie bisher!

Betrachten Sie das folgende (schlechte) Codefragment:

```
...
void draw(Form f) {
     if (f.type == circle)
          drawCircle( f );
     else if (f.type == square)
          drawSquare( f );
...
}
```

Hier soll in einer Methode `draw()` eine geometrische Form gezeichnet werden. Je nach deren Typ (Kreis oder Quadrat) wird über eine geschachtelte Abfrage die eigentliche Zeichenmethode aufgerufen.

Falls dieses Programm um weitere Formen, etwa Ellipse oder Dreieck, erweitert werden soll, müsste jedes Mal die oben gezeigte `draw()`-Methode entsprechend erweitert und im Quelltext angepasst werden.

Bild 6.21 zeigt eine sinnvolle Modellierung dieses Sachverhalts. Das Zeichnen geometrischer Objekte wurde in einer Schnittstelle (*interface*) modelliert. Die geometrischen Objekte selbst implementieren diese Schnittstelle. Das aufrufende System ist nur von der Schnittstelle abhängig und kennt die Klassen der beteiligten Objekte nicht mehr.

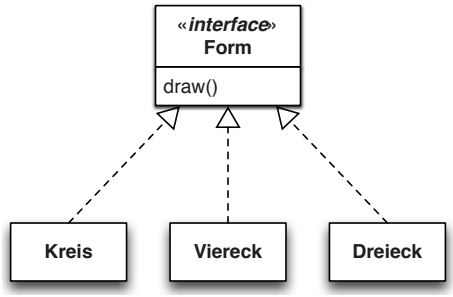


BILD 6.21
(Besseres) Polymorphes Modell geometrischer Formen, genügt dem OGP

Polymorphismus

Diese Art der Modellierung nutzt eine der grundlegenden Eigenschaften[31] objektorientierter Sprachen, den Polymorphismus. Der Begriff stammt aus dem Griechischen und bedeutet *Vielgestaltigkeit*. Kurz gesagt, entkoppelt Polymorphismus die Schnittstelle von der Implementierung, das „Was“ vom „Wie“.

In den meisten objektorientierten Sprachen funktioniert Polymorphismus auf Basis von dynamischem Binden. Hierbei wird die Zuordnung zwischen aufrufendem und aufgerufenem Objekt erst zur Laufzeit getroffen. Prozedurale Sprachen legen diese Zuordnung bereits zur Übersetzungszeit statisch fest.

[31] Die anderen beiden zentralen Eigenschaften sind Datenabstraktion und Vererbung.

OGP für nicht-objektorientierte Systeme

OGP verbessert Entwürfe

Das Sprachmittel der Polymorphie steht in der Regel nur bei objektorientierten Programmiersprachen zur Verfügung. Dennoch können Sie in allen Fällen vom OGP profitieren. Selbst wenn nur die Modelle Ihrer Systeme dem Offen-Geschlossen-Prinzip genügen und die Implementierung ohne den Komfort von Polymorphismus auskommen muss – die konzeptionelle Auflösung von Abhängigkeiten verbessert die Qualität der Entwürfe.

6.4.4 Abhängigkeit nur von Abstraktionen

Bevorzugen Sie Abhängigkeiten von Abstraktionen, nicht von konkreten Implementierungen.

Die Abhängigkeit von Abstraktionen[32] anstelle von konkreten Implementierungen ist der Schlüssel zu flexiblen und erweiterbaren Architekturen.[33]

Klassische Entwürfe für prozedurale Sprachen zeigen eine sehr charakteristische Struktur von Abhängigkeiten. Wie Bild 6.21 illustriert, beginnen diese Abhängigkeiten meist an einer zentralen Stelle, etwa dem Hauptprogramm oder dem zentralen Modul einer Anwendung. Die Module auf einer so hohen Ebene implementieren oft abstrakte Prozesse. Sie sind aber direkt von der Implementierung der konkreten Einheiten, nämlich den einzelnen Funktionen, abhängig.

Abhängigkeit schafft Probleme

Diese Abhängigkeit schafft in der Praxis große Probleme. Sie verhindert nämlich, dass die Implementierung konkreter Funktionen geändert werden kann, ohne Auswirkungen auf das Gesamtsystem zu haben.

Es tritt genau das Gegenteil davon ein, was Sie sich für flexible Architekturen wünschen. Möchten Sie beispielsweise die proprietäre Verwaltung der Zugriffsrechte eines Systems durch eine Standardkomponente[34] ersetzen, stellt die Abhängigkeitsstruktur von Bild 6.22 ein großes Problem dar.

Ihre Entwürfe werden viel flexibler und wartbarer, wenn Sie möglichst nur Abhängigkeiten von Abstraktionen zulassen. Betrachten Sie Bild 6.23. Dort hängt der zentrale Geschäftsprozess ausschließlich von abstrakten Klassen oder Schnittstellen ab. Konkrete Implementierungen hängen durch die Vererbungsbeziehung ebenfalls von Schnittstellen oder abstrakten Klassen ab. Auf hoher Ebene gibt es dabei keine Abhängigkeit von konkreten Implementierungen.

Abhängigkeitsstrukturen dieser Art erlauben es, Implementierungskomponenten austauschbar zu halten. Sie sind deshalb die Basis wichtiger Komponentenmodelle oder Dependency-Injection Frameworks (Spring & Co.)

---

[32] In der Literatur, etwa [Martin2000], wird dieses Prinzip oft „Prinzip der Umkehrung von Abhängigkeiten“ (*dependency inversion principle*) genannt.

[33] Im Extremfall können solche Abhängigkeiten auch zu schwer verständlichen Indirektionen führen – und beispielsweise Debugging von Systemen deutlich erschweren.☹

[34] Wie etwa das standardisierte LDAP (lightweight directory access protocol) oder das bei Mainframes verbreitete System RACF.

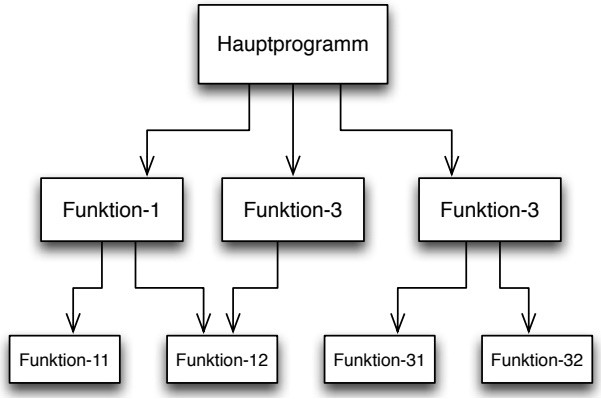


BILD 6.22
(Schlechte) Abhängigkeiten in prozeduralen Systemen

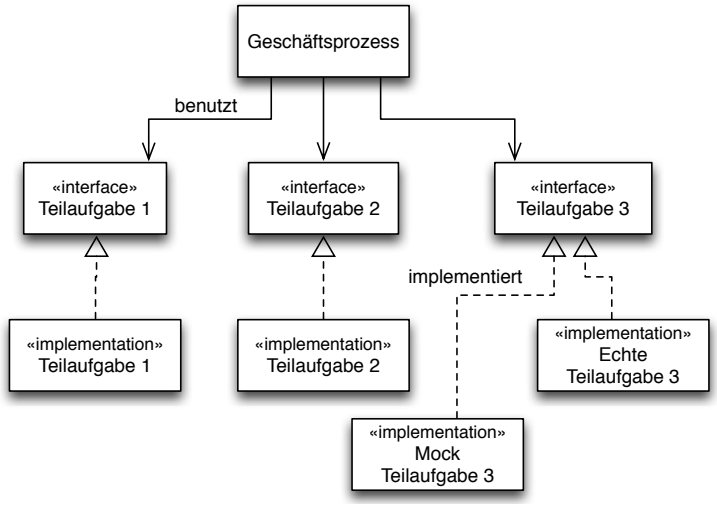


BILD 6.23
Ideale Abhängigkeitsstruktur

6.4.5 Abtrennung von Schnittstellen

Mehrere spezifische Schnittstellen sind besser als eine große Universalschnittstelle.

Wenn eine Komponente von mehreren anderen Komponenten eines Systems benutzt wird, dann entwerfen Sie mehrere nutzerspezifische Schnittstellen.

Modellieren Sie die verschiedenen Verantwortungsbereiche einzelner Komponenten in jeweils einer Schnittstelle.

Spezifische Schnittstellen

Jede dieser spezifischen Schnittstellen modelliert ausschließlich Aufgaben einer Komponente. So vermeiden Sie umfangreiche Universalschnittstellen. Das hält die einzelnen Schnittstellen einfacher und verringert somit Komplexität.

Was geschieht, wenn Sie dieses Prinzip nicht anwenden, sehen Sie in Bild 6.24. Eine Klasse wird von zwei Komponenten benutzt (im Bild als Client A und Client B bezeichnet). Wird eine der ausschließlich von Client A genutzten Methoden geändert, muss auch der Code von Client B neu übersetzt, getestet und ausgeliefert werden.

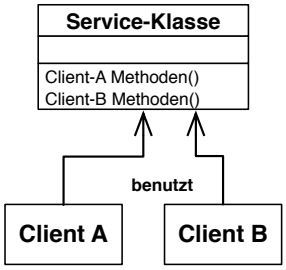


BILD 6.24
(Schlechte) Universalschnittstelle

In Bild 6.25 sehen Sie eine bessere Modellierung dieses Systems. Die spezifischen Schnittstellen für Client A und Client B sind explizit als solche modelliert. Eine Änderung in den spezifischen Methoden für Client A zieht keine Änderung an Client B nach sich. Insgesamt ist das System flexibler und wartungsärmer geworden.

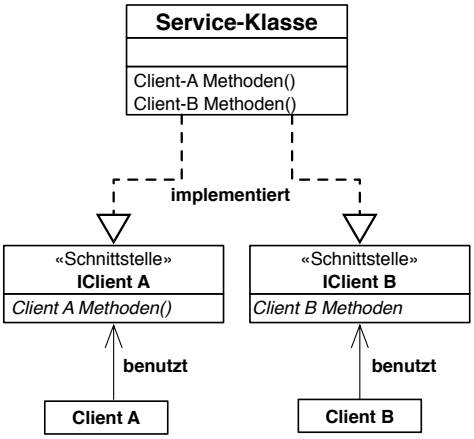


BILD 6.25
(Bessere) Separierte Schnittstellen

Aus einem etwas anderen Blickwinkel betrachtet: Wählen Sie Schnittstellen mit einem (einzigen) semantischen Zusammenhang, beispielsweise „sortierbar". Klassen können solche einfachen Schnittstellen leichter implementieren als große Schnittstellenmonolithen wie etwa „sortierDruckUndSpeicherbar".

Schnittstellen angemessen abtrennen

Die Einführung neuer Schnittstellen zur Entkopplung von Benutzern und Anbietern verbessert in vielen Fällen die Erweiterbarkeit und Flexibilität von Entwürfen. Gerade beim Entwurf von Klassenstrukturen kann die Abtrennung von Schnittstellen allerdings schnell unübersichtlich werden, weil die Anzahl der Schnittstellen stark wächst.

Beachten Sie daher das Prinzip der Schlichtheit: Wenn Ihre Entwürfe nach der Abtrennung von Schnittstellen unübersichtlich werden, dann

- gruppieren oder kategorisieren Sie Klassen und
- modellieren Sie Schnittstellen für jede Gruppe oder Kategorie.

Falls einzelne Methoden Ihrer Service-Klasse von mehreren Klassen genutzt werden, können Sie diese Methoden problemlos in mehrere Schnittstellen aufnehmen.

6.4.6 Zyklische Abhängigkeiten vermeiden

Vermeiden Sie zyklische Abhängigkeiten (wie der Vampir den Knoblauch!).

Zyklische Abhängigkeiten sind für alle Arten von Strukturen problematisch, weil sie die Wartbarkeit und Änderbarkeit von Systemen erschweren. Bild 6.26 zeigt eine solche Struktur. Drei Module hängen zyklisch voneinander ab. Stellen Sie sich nun vor, Sie hätten die Implementierung von Modul A verbessert. Bevor Sie eine neue Version des Moduls freigeben können, müssen Sie nicht nur Modul A testen, sondern aufgrund der zyklischen Abhängigkeiten auch noch die Module B und C.

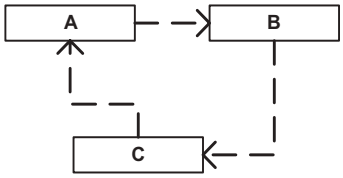


BILD 6.26
Zyklische Abhängigkeit

Praktisch werden Sie zyklische Abhängigkeiten nicht immer vermeiden können. Ihnen stehen jedoch einige Mittel zur Abhilfe zur Verfügung:

- Erzeugen Sie ein neues Modul oder Paket, je nachdem, welche Abstraktionsstufe eines Entwurfs Sie bearbeiten. Es enthält diejenigen Bestandteile aus A, die von C genutzt werden.
- Wenden Sie das Prinzip der Abhängigkeit von Schnittstellen an, wie in Bild 6.27 gezeigt: Modellieren Sie eine Schnittstelle für die von C genutzten Bestandteile des Moduls A, und platzieren Sie diese in Modul C. Dadurch ersetzen Sie die benutzt-Beziehung zwischen den Modulen durch eine Vererbungsbeziehung.

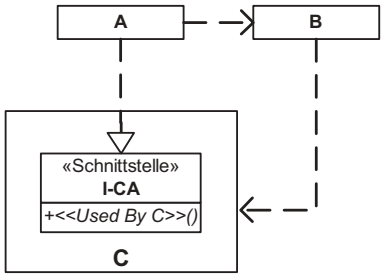


BILD 6.27
Auflösung einer zyklischen Abhängigkeit

6.4.7 Liskov-Substitutionsprinzip (LSP)

Klassen sollen in jedem Fall durch ihre Unterklassen ersetzbar sein.
(Wenn Sie schon Vererbung einsetzen, dann auch richtig!)

Das Substitutionsprinzip wurde von Barbara Liskov in [Liskov81] beschrieben.[35] Es besagt, dass jede Basisklasse immer durch ihre abgeleiteten Klassen (Unterklassen) ersetzbar sein soll. Eine Methode, die ein Objekt vom Typ der Basisklasse erwartet, soll auch korrekt funktionieren, wenn ein Objekt der Unterklasse U übergeben wird.

So selbstverständlich dies klingt – die Technik der Vererbung erlaubt auch anderes Verhalten.

Betrachten Sie das Beispiel von Kreis und Ellipse (nach [Martin2000]): Die Schulweisheit, wonach ein Kreis lediglich eine spezielle Ellipse darstellt, könnte dazu verleiten, die Vererbungsbeziehung wie in Bild 6.28 zu modellieren. Rein technisch ist solche Vererbung compilierbar, doch birgt die Implementierung versteckte Risiken.

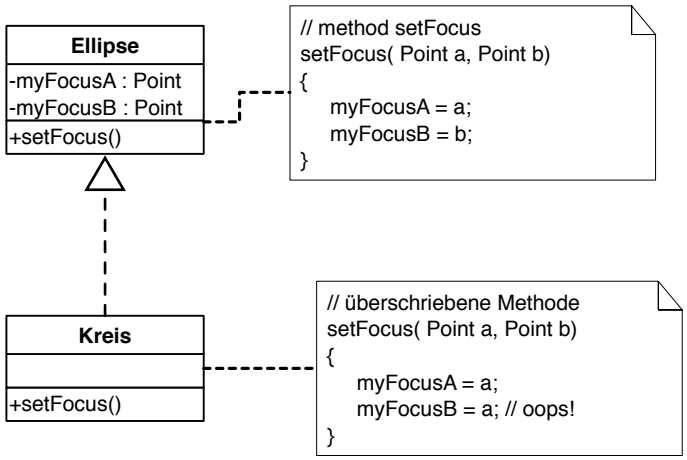


BILD 6.28
(Falsche) Vererbung von Kreis und Ellipse

[35] [Martin2000] bringt es mit dem „Design-by-Contract“ von Bertrand Meyer in Verbindung.

So besitzt eine Ellipse zwei Brennpunkte, ein Kreis jedoch nur einen, nämlich seinen Mittelpunkt. Die Implementierung der `setFocus()`-Methode der Ellipse wird in der Unterklasse Kreis überschrieben. Die Klasse Kreis stellt damit sicher, dass ihre Objekte nur einen einzigen Mittelpunkt haben. Das ist mathematisch korrekt, schafft jedoch ein Problem:

Ein Benutzer einer Ellipse darf erwarten, dass folgendes Stück Code erfolgreich funktioniert:

```
void f( Ellipse e) {
     Point p1 = new Point( 1,1);
     Point p2 = new Point( 2,2);
     e.setFocus( p1, p2 );
     assert ( e.getFocusB() == p2); // oh je!
}
```

Die Funktion setzt zwei Brennpunkte einer Ellipse und prüft anschließend, ob diese korrekt übernommen wurden.

Für Ellipsen funktioniert obiger Code problemlos. Sie können der Funktion jedoch auch eine Instanz von Kreis übergeben. In diesem Fall arbeitet die obige Methode jedoch fehlerhaft: Die überschriebene `setFocus()`-Methode der Klasse Kreis verhält sich anders als erwartet!

Substituierbarkeit gewährleistet Einhaltung von Verträgen

Wenn Sie das Substitutionsprinzip durchgängig berücksichtigen, erfüllen sämtliche Unterklassen die impliziten „Verträge“ ihrer Oberklassen. Überschriebene Methoden besitzen keine stärkeren Vorbedingungen als die der Oberklasse und keine schwächeren Nachbedingungen. Robert Martin formuliert es prägnant:

Unterklassen sollen nicht mehr erwarten und nicht weniger liefern als ihre Oberklassen.

6.4.8 Dependency Injection (DI)

Bei der Nutzung von Schnittstellen im objektorientierten Entwurf besteht immer das Problem, zu einer (abstrakten) Schnittstelle zur Laufzeit eine (konkrete) Instanz zu beschaffen: Wer übernimmt die Verantwortlichkeit, den Lebenszyklus der genutzten Instanzen zu verwalten? Wer entscheidet, welche konkrete Klasse zur Laufzeit letztlich instanziiert werden soll? Bild 6.29 zeigt diesen Fall: Ein Client der Klasse A modelliert eine benutzt-Beziehung zu einer Schnittstelle ISrv. Zur Laufzeit muss eine konkrete Instanz von ISrv erzeugt werden, auf die A zugreifen kann. Wer soll das bewerkstelligen? Wenn A dafür selbst verantwortlich ist, so besteht eine zu enge Kopplung zwischen A und dem Typ der konkreten Klasse (Srv).

Das „Entkopplungsmuster“ *Dependency Injection* stellt dafür einen eigenständigen Baustein bereit (den *Dependency Injector, DI-Container* oder *Assembler*), der zur Laufzeit über die oben genannten Fragen entscheidet. Der Assembler übergibt die Referenzen auf konkrete Instanzen an die abhängigen Objekte. Im Beispiel (siehe Bild 6.29) erzeugt er in Schritt 1 eine Instanz (namens srv) und übergibt diese im zweiten Schritt an den Client A.

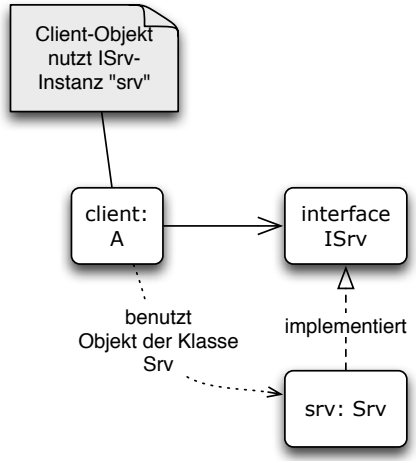


BILD 6.29
Woher kommt die Instanz bei Entkopplung über Schnittstellen?

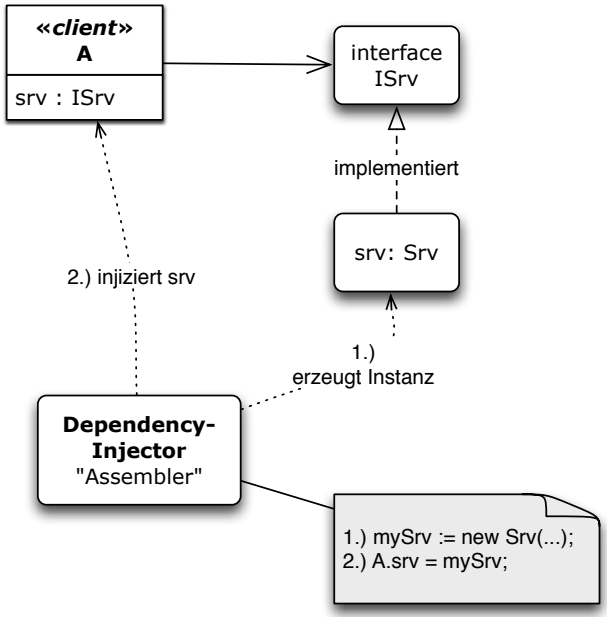


BILD 6.30
Dependency Injection

Martin Fowler diskutiert in seinem Einführungsartikel[36] die sogenannte „Setter-Injection", bei der Abhängigkeiten über set-Methoden injiziert werden, und die „Constructor Injection", bei der die notwendigen Instanzen bereits beim Konstruktoraufruf mitgegeben werden. Viele Frameworks (wie Spring[37] für Java und .NET, PicoContainer, Guice oder ObjectBuilder) implementieren dieses Prinzip und ermöglichen es somit, Abhängigkeiten „per Konfiguration" zu verwalten.

[36] *http://martinfowler.com/articles/injection.html*
[37] *www.springframework.org* und *www.springframework.net*

Sollten Ihre Entwürfe unter einer großen Zahl von Abhängigkeiten (im Quellcode) leiden, sollten Sie den Einsatz eines Dependency-Injection-Frameworks in Erwägung ziehen. Allerdings kurieren diese Frameworks nur Abhängigkeiten auf Codeebene. Abhängigkeiten „im Großen“, zwischen Paketen und Subsystemen, müssen Sie weiterhin selbst verwalten!

6.5 Entwurfsmuster

Gang-Of-Four

Entwurfsmuster beschreiben einfache und elegante Lösungen für häufige Entwurfsprobleme. Die in Software-Kreisen mittlerweile berühmte „Gang-of-Four“[38] (GoF) legte mit [Gamma95] den Grundstein zu diesem interessanten und in der Praxis sehr wichtigen Gebiet des Software-Entwurfs. Ich habe hier einige Muster zusammengestellt, die auch über den Entwurf von Klassenstrukturen hinaus Bedeutung besitzen. Insbesondere können sie Ihnen dabei helfen, Abhängigkeiten in Entwürfen zu reduzieren.

6.5.1 Entwurf mit Mustern

Muster vereinfachen Kommunikation

Entwurfsmuster helfen Ihnen als Softwarearchitekt sowohl bei der eigentlichen Entwurfstätigkeit als auch bei der Kommunikation mit Designern und Entwicklern.

Weil viele wichtige Muster von ihren Autoren sehr präzise dokumentiert wurden, vermeiden sie Missverständnisse bei der Diskussion von Entwürfen. Mit ihrer Hilfe lässt sich ein gemeinsames Verständnis von Entwürfen auf hohen Abstraktionsebenen leichter herstellen. Die prägnanten Bezeichnungen von Mustern erlauben es, auch komplexe Strukturen in vergleichsweise wenigen Worten[39] präzise zu beschreiben. Das erleichtert es Teams, die „Vision“ eines Entwurfs zu entwickeln und prägnant zu beschreiben.

Muster können Ihnen helfen, sich bei der Dokumentation Ihrer Entwürfe auf das Wesentliche zu konzentrieren, nämlich auf die Besonderheiten Ihrer konkreten Aufgabe.

Abschließend aber eine Mahnung zur Vorsicht: Setzen Sie Muster bedarfsgerecht ein. Muster machen Ihre Entwürfe oft komplexer, indem sie zusätzliche Abstraktionen oder Komponenten einführen. Schlichtheit und Verständlichkeit können jedoch wichtiger sein als hohe Flexibilität. Wählen Sie im Zweifelsfall die schlichte Alternative (und dokumentieren Sie diese Entwurfsentscheidung).

[38] Erich Gamma, Rich Helm, Ralph Johnson, John Vlissides

[39] Erich Gamma und Kent Beck haben das mit [Gamma99] brillant vorgeführt. Die beiden beschreiben auf wenigen Seiten ein komplexes Testframework. Sie benutzen Entwurfsmuster, um die Konzepte dieser Architektur erstaunlich kompakt, aber dennoch präzise und verständlich darzustellen.

6.5.2 Adapter

[Ein Adapter] passt die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster lässt Klassen zusammenarbeiten, die ansonsten dazu nicht in der Lage wären.

[Gamma95]

Wenn Sie DIES brauchen, aber DAS haben, kann Ihnen das Adapter-Muster helfen. Genauer gesagt: Ein Adapter soll die Schnittstelle eines existierenden Moduls ändern.

Setzen Sie dieses Muster ein, wenn Sie ein existierendes Modul verwenden wollen, dessen Schnittstelle nicht mit der von Ihnen benötigten Schnittstelle übereinstimmt.

Bild 6.31 zeigt das Adapter-Muster. Sie erkennen darin eine vorhandene Klasse, die DAS anbietet, und einen Klienten, der DIES braucht.

Adapter unterstützt Kapselung

Das Adapter-Muster benötigen Sie auch, wenn Sie bestehende Anwendungssysteme kapseln. Somit ist es ein wichtiges Werkzeug für die Integration und das Umwickeln (*Wrapping*) von Altsystemen (siehe Abschnitt 7.3 bzw. 7.4).

[Gamma95] beschreibt das Muster im Detail und schlägt verschiedene Erweiterungen vor, etwa den Zwei-Wege-Adapter oder den steckbaren Adapter.

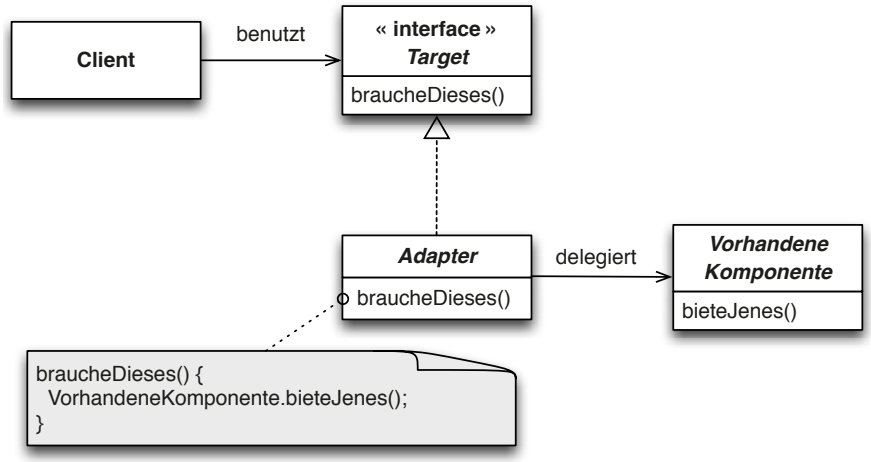


BILD 6.31 Entwurfsmuster „Adapter“

6.5.3 Beobachter (Observer)

Ein Observer definiert eine kontrollierte Abhängigkeit zwischen Objekten, sodass die Änderung eines Objektes die Benachrichtigung und Aktualisierung aller abhängigen Objekte auslöst.

Verwenden Sie Observer, wenn eine Komponente in der Lage sein soll, andere Komponenten zu benachrichtigen, ohne zu wissen,

- wer die anderen Komponenten sind;
- wie viele Komponenten geändert werden müssen (nach [Gamma95]).

Beobachter reagiert auf Zustandsänderung

Es kommt vor, dass eine Komponente (= ein Beobachter) auf eine Zustandsänderung einer anderen Komponente (= das konkrete Subjekt) reagieren soll, ohne dass das Subjekt den Beobachter kennt. Ein Subjekt kann eine beliebige Zahl von Beobachtern besitzen. Verändert das Subjekt seinen Zustand, so werden alle diese Beobachter benachrichtigt. Als Reaktion darauf synchronisiert sich jeder Beobachter mit dem Subjekt, indem er dessen Zustand erfragt.

In Bild 6.32 verwaltet das Subjekt seine Beobachter. Es bietet ihnen eine Schnittstelle zum dynamischen An- und Abmelden. Konkrete Beobachter kennen ihre beobachteten (konkreten) Subjekte.

Es ist interessant, das dynamische Verhalten des Beobachtermusters zu betrachten (siehe Bild 6.33).

Das Beobachtermuster ist die Grundlage des Model-View-Controller-Musters, das häufig bei Entwurf und Implementierung von Benutzeroberflächen eingesetzt wird (siehe Abschnitt 6.2.6.1).

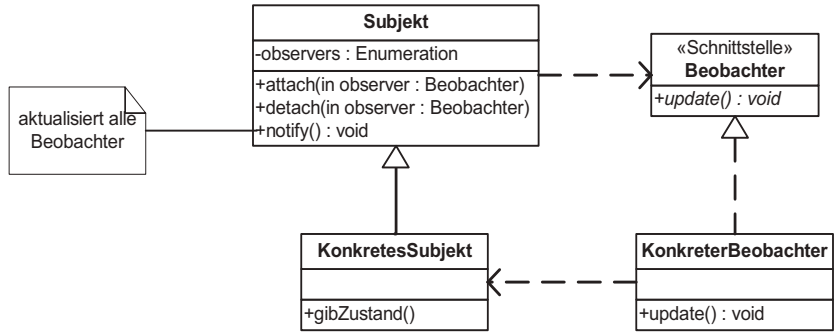


BILD 6.32 Entwurfsmuster „Beobachter“

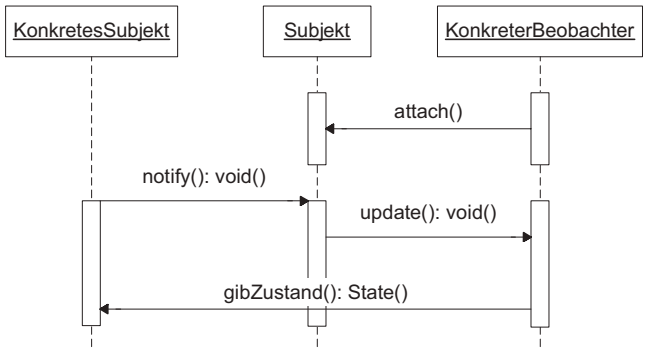


BILD 6.33 Dynamisches Verhalten des Beobachtermusters

6.5.4 Dekorierer (Decorator)

Ein Dekorierer erweitert eine Komponente dynamisch um Funktionalität.

Verwenden Sie das Dekorierermuster, um einer Komponente dynamisch und transparent Funktionalität hinzuzufügen, ohne die Komponente selbst zu erweitern (nach [Gamma95]).

Im Gegensatz zum Adapter (Abschnitt 6.5.2) verändert ein Dekorierer Funktionalität, nicht aber die Schnittstelle einer Komponente. Ein Adapter liefert hingegen eine neue Schnittstelle.

Wie der Adapter besitzt auch das Dekorierer-Muster Bedeutung für die Integration, Kapselung und Weiterbenutzung bestehender Anwendungssysteme.

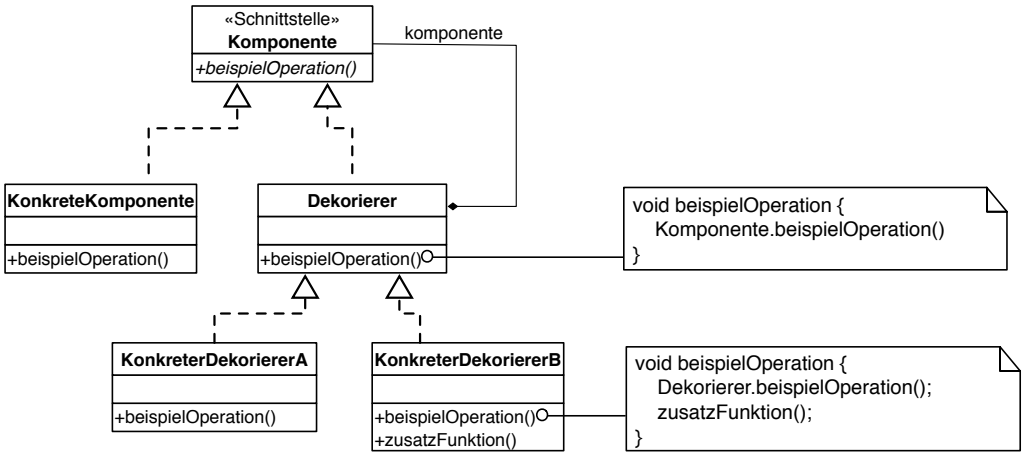


BILD 6.34 Entwurfsmuster „Dekorierer“

6.5.5 Stellvertreter (Proxy)

Ein Stellvertreter (Proxy) stellt einen Platzhalter für eine andere Komponente (Subjekt) dar und kontrolliert den Zugang zum echten Subjekt.

Die Schnittstelle des Stellvertreters ist identisch mit der Schnittstelle des echten Subjektes, sodass der Stellvertreter als Ersatz des Subjektes auftreten kann.

Sie können das Stellvertreter-Muster einsetzen, wenn Sie den Zugriff auf das echte Subjekt kontrollieren wollen. Der Proxy kann das echte Subjekt erzeugen und löschen.

Darüber hinaus kann er weitere Verantwortlichkeiten besitzen:

- Entfernte Stellvertreter (*remote proxies*) sind verantwortlich für die Abwicklung von Anfragen an das entfernte Subjekt (das möglicherweise auf einem anderen Rechner existiert).
- Virtuelle Stellvertreter (*virtual proxies*) können Informationen des echten Subjektes enthalten, um den Zugriff auf das echte Subjekt zu verzögern. Dies ist nützlich, wenn der Zugriff auf das echte Subjekt langsam oder teuer ist, wenn dazu etwa Datenbankoperationen notwendig werden.
- Schützende Stellvertreter (*protection proxies*) prüfen, ob Zugreifer auf das echte Subjekt über die notwendigen Rechte verfügen.

Bild 6.35 zeigt die Struktur des Stellvertreter-Musters. Zur Laufzeit leitet ein Proxy Anfragen erst im Bedarfsfall an das echte Subjekt weiter.

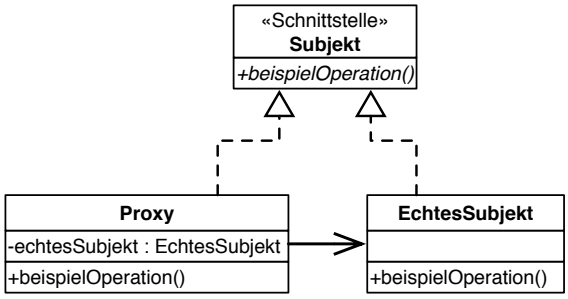


BILD 6.35
Entwurfsmuster „Stellvertreter (Proxy)“

6.5.6 Fassade

Eine Fassade „bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems“.

[Gamma95]

Fassade reduziert Abhängigkeiten

Das Entwurfsmuster Fassade bietet eine weitere Möglichkeit, Abhängigkeiten zwischen unterschiedlichen Systemkomponenten zu reduzieren. Betrachten Sie dazu Bild 6.36. Es illustriert, wie die Fassade die internen Komponenten des Subsystems nach außen verbirgt.

Verwenden Sie das Fassade-Muster, wenn Sie

- zu einem komplexen Subsystem eine einfache Schnittstelle anbieten wollen;
- Klienten eine vereinfachte Sicht auf das Subsystem anbieten wollen (Klienten brauchen nicht hinter die Fassade zu schauen!);
- ein System in Schichten aufteilen wollen und einen (definierten) Eintrittspunkt in jede Schicht benötigen.

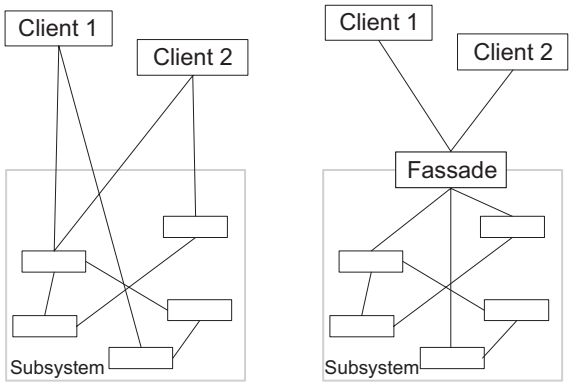


BILD 6.36
Entwurfsmuster „Fassade“

Eine Fassade kennt die internen Details des Subsystems. Sie muss wissen, welche der internen Komponenten für eine bestimmte Anfrage oder einen Aufruf zuständig ist. Sie delegiert Anfragen von Clients an die jeweils zuständigen Subsystemkomponenten (nach [Gamma95]).

Eine Spezialisierung des Fassade-Musters ist die Wrapper-Fassade, die sich auf die Kapselung von Daten und Funktionen nicht-objektorientierter Programmierschnittstellen bezieht. Siehe [Schmidt2000].

6.5.7 Zustand (State)

Das Zustandsmuster „ermöglicht einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hat.“

[Gamma95]

Es kommt häufig vor, dass Komponenten abhängig von ihrem Zustand ein spezielles Verhalten zeigen sollen. In der UML modellieren Sie das in Zustands- oder Zustandsübergangsdiagrammen. In einem bestimmten Zustand darf eine Komponente dann nur bestimmte Operationen ausführen.

Das Zustandsmuster löst dieses Problem über Polymorphismus: Die in einem bestimmten Zustand möglichen Operationen werden durch eine (konkrete) Unterklasse implementiert, die bei Zustandsübergängen bei Bedarf ausgetauscht werden. Zur Laufzeit wird die jeweilige Operation vom (abstrakten) Zustand (in Bild 6.37 heißt er ItsState) über Polymorphismus durch den jeweilig instanziierten konkreten Zustand realisiert.

Eine leichte Variation dieses Musters, zusammen mit dem zugehörigen Zustandsdiagramm, finden Sie in Bild 6.38. Die konkreten Zustandsklassen implementieren sämtliche für ihren eigenen Zustand benötigten Operationen.

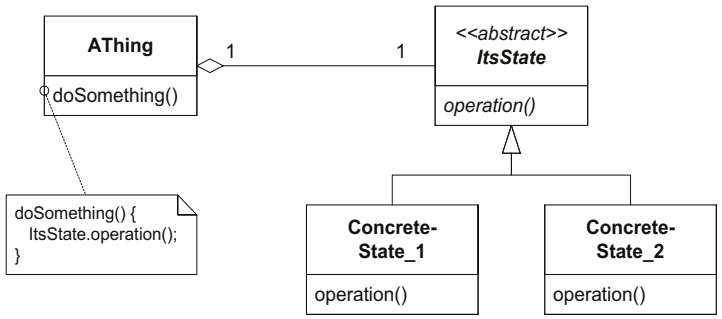


BILD 6.37 Entwurfsmuster „Zustand“ (nach [Gamma95])

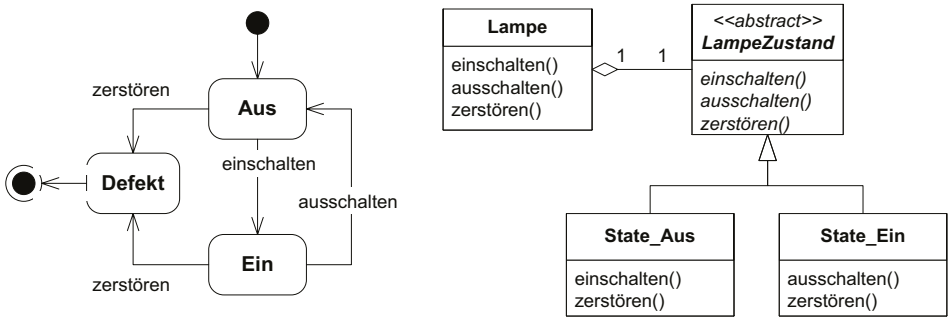


BILD 6.38 Beispiel für Zustandsmuster einer Lampe

6.6 Entwurf, Test, Qualitätssicherung

Bewertung: Kapitel 8

Die Themen Test und Qualitätssicherung werden in manchen Projekten mit den funktionalen Tests der Implementierung gleichgesetzt. Für Softwarearchitekten sollten Tests und Qualitätssicherung jedoch wesentlich früher beginnen. Architekturentscheidungen und Sichten können hinsichtlich der Erfüllung der Systemanforderungen qualitativ bewertet werden. Details dazu finden Sie in Kapitel 8 sowie in [Bass+03] und [Clements+02].

Daneben sollten Sie als Architekt die Testbarkeit des fertig implementierten Systems im Auge behalten. Sie sollte von Anfang an zu Ihren wichtigen Kriterien für den Systementwurf gehören. Hierbei zeichnet sich insbesondere die Trennung fachlicher und technischer Bestandteile (wie beim *Domain Driven Design*) als besonders einfach testbar aus.

Zusätzlich helfen automatisierte Tests, den Entwurf der eigenen Bausteine und Schnittstellen auf Einfachheit und Verständlichkeit hin zu prüfen: Fällt die Implementierung eines Unit-Tests sehr schwer, ist möglicherweise die Schnittstelle des zu testenden Bausteins schlecht!

Einige wichtige Ratschläge zu diesem Thema:

- Die Anzahl der nach einem Test *nicht entdeckten* Fehler ist proportional zur Anzahl der in diesem Test *entdeckten* Fehler. Anders formuliert: Fehler clustern: Wenn Sie in einer Komponenten bereits viele Fehler gefunden haben, werden Sie dort noch weitere Fehler finden. Konzentrieren Sie sich beim Test auf diese (kritischen) Komponenten!
- Die Kosten der Fehlerbehebung steigen exponentiell mit der Anzahl der Entwicklungsphasen, die seit dem Eintritt/Entstehen des Fehlers in das System vergangen sind. Entwickeln Sie daher Prototypen, um Fehler möglichst früh zu finden!
- Implementieren Sie (auch in Ihrer Rolle als Architekt!) automatisierte Testfälle mit den Testframeworks Ihrer Wahl. Das gibt Ihnen sehr kurzfristig Feedback zu Entwurfsentscheidungen.
- Prüfen Sie Codequalität im Rahmen Ihrer Continuous-Integration oder Daily-Builds. Falls Sie so etwas noch nicht haben, wird's höchste Zeit ☺

6.7 Weiterführende Literatur

Obwohl das Thema Softwareentwurf zu den wichtigsten Fundamenten der Informationstechnik zählt, gibt es nur relativ wenig Literatur, die systematisch die Grundlagen dieses Gebietes darstellt.

Zu Grundlagen des Entwurfs

GRASP

[Larman2001] beschreibt die praktische Anwendung von UML und Mustern. Er orientiert sich stark am Begriff der „Verantwortlichkeit" (*responsibility*). Larman definiert dazu einige Muster (*General Responsibility Assignment Patterns*, GRASP) und illustriert an vielen Beispielen deren Anwendung. Mein persönlicher Favorit zu objektorientiertem Entwurf.

[Martin96] beschreibt die Grundlagen des objektorientierten Entwurfs. Seine Beispiele illustriert Robert Martin an der Sprache C++, die Entwurfsprinzipien gelten jedoch für sämtliche objektorientierten Sprachen, teilweise sind sie sogar unabhängig von der Objektorientierung.

[Martin2000] stellt neben einigen der auch hier beschriebenen Entwurfsprinzipien noch weitere vor, etwa einige Prinzipien zur Aufteilung von Klassen auf Pakete.

Domain Driven Design

[Evans04] sowie [Avram+06] (Letzteres auch online verfügbar) stellen *Domain Driven Design* im Detail vor. Mein Tipp für ernsthafte DDD-Aspiranten: [Vernon-13], allerdings benötigen Sie eine Menge Zeit für dieses monumentale Stück Weisheit.

[Siedersleben04] erläutert den Entwurf von Komponenten anhand der Trennung in technische und anwendungsbezogene Teile. Seine Darstellung des Themas ist sowohl gründlich als auch praxisbezogen.

Zu speziellen Aspekten des Entwurfs

[Fowler99] erläutert die Methode des Refactoring. Sie stellt einen systematischen Umbau von Programmen dar und ermöglicht es, den Entwurf und die Implementierung bestehender Systeme zu verbessern.

[Johnson+05] stellt die praktische Anwendung von *Dependency Injection* mit dem Spring-Framework vor – eine alte, aber lesenswerte Darstellung pragmatischen Software-Entwurfs.

Zu Entwurfsmustern

Eine generelle Empfehlung zu Entwurfsmustern sind die Beiträge zu den Konferenzen über Entwurfsmuster (*Pattern Languages of Programs*, PloP, ChiliPloP) und deren europäische Schwesterkonferenz, EuroPloP. Die meisten Beiträge sind online verfügbar, *http://www.hillside.net* ist die Heimat der Pattern-Gemeinde im Internet .

[Gamma95] ist Pflichtlektüre für alle, die (objektorientierten) Software-Entwurf betreiben.

[Eilebrecht+13] gibt einen kompakten Überblick über Muster.

Zu Architekturmustern und Referenzarchitekturen

[Buschmann+96] benutzen einige der grundlegenden Muster von [Gamma95] als Grundlage erweiterter oder umfangreicherer Architektur-Muster. So beschreiben sie ausführlich Schichten (*Layer*), Pipes-and-Filter, Model-View-Controller sowie den Befehlsverarbeiter (*Command Processor*).

[Fowler02] nähert sich Muster von der pragmatischen Seite her, indem er Lösungsmuster für typische Problemstellungen kommerzieller Informationssysteme aufzeigt. Der Titel des Buches (*Patterns for Enterprise Application Architecture*) ist meiner Meinung nach jedoch irreführend: Im Fokus des Buches liegen eindeutig Entwurfs- und Programmiermuster – Architekturmuster im Stil von [Buschmann+96] suchen Sie dort vergeblich. Allerdings ist seine Systematik unmittelbar anwendbar.

[Buschmann+07] ist der vierte Band der POSA-Serie (Pattern Oriented Software Architecture). Sehr lesenswerte Zusammenfassung des Vorgehens beim Entwurf, dem systematischen Übergang von *mud to structure*.

Zu Test und Qualitätssicherung

Test-Bibel: [Binder200]

Eine Fülle von Büchern beschäftigt sich mit dem Testen von Software-Systemen, für unterschiedliche Teststufen (System-, Integrations- oder Komponententest) und verschiedene Programmiersprachen. Als Softwarearchitekt sollten Sie meiner Meinung Testfälle für Ihr System schreiben – passende Frameworks gibt es heute in praktisch jeder Programmiersprache. Lernen Sie, einen der Frameworks für Behavior- Driven-Development oder Acceptance-Testing zu nutzen, beispielsweise Cucumber (cucumber.info) oder Spockframework (spockframework.org).

Brian Marick gehört zu den Koryphäen auf dem Gebiet „Testen“ und sein Artikel [Marick97] über *Classic Testing Mistakes* zu den „Augenöffnern“!

7 Technische Konzepte und typische Architekturaspekte

We shape our buildings: thereafter they shape us.

Winston Churchill

Fragen, die dieses Kapitel beantwortet:

- Was ist Entwurf-durch-Routine, und was bedeutet Entwurf-durch-Innovation?
- Was sind Strukturen und technische Konzepte?
- Wie behandelt man folgende Themen in Software-Entwürfen:
  - Persistenz
  - Geschäftsregeln
  - Integration, Verteilung und Kommunikation
  - Graphische Oberflächen
  - Workflow-Management
  - Sicherheit
  - Protokollierung, Logging
  - Ausnahme- und Fehlerbehandlung

Routine und Innovation im Software Engineering

Der Ausdruck „Software Engineering“ entstand 1969 im Zuge eines von der NATO veranstalteten Workshops ([NATO69]). Seither schmückt sich unsere Disziplin mit den Lorbeeren des „*ingenieurmäßigen* Vorgehens“, wobei die Praxis sich doch erheblich von anderen Ingenieurdisziplinen unterscheidet. So ist es im Maschinenbau, der Elektrotechnik und der Gebäudearchitektur durchaus üblich, bei Neuentwicklungen auf bereits vorhandene Lösungen zurückzugreifen. Bei der Konstruktion einer Flugzeugbremse werden die beteiligten Ingenieure auf dokumentierte Erkenntnisse und Lösungen zurückgreifen und diese gegebenenfalls modifizieren. Kaum ein Maschinenbauingenieur käme beispielsweise auf die Idee, die für eine Bremse notwendigen Hydraulikpumpen, die Bremsscheiben und die Steuerelektronik von Grund auf neu zu entwerfen.

In Softwareprojekten sieht die Realität gänzlich anders aus: Dort entwickeln Software-Architekten oftmals sämtliche für ein System benötigten Bestandteile neu, obwohl große Teile davon in anderen, ähnlichen Systemen bereits vorhanden sind.

Entwurf-durch-Routine

Die klassischen Ingenieurdisziplinen verlassen sich im Gegensatz zur Informatik darauf, dass ihr Wissen in einer für Praktiker anwendbaren Form kodifiziert wird. Sie praktizieren damit eine Form des Entwurfs, den [Shaw+96] als *Entwurf-durch-Routine* bezeichnet. Diese Art des Entwurfs ist charakterisiert durch die Wiederverwendung großer Teile früherer Lösungen. Mit dieser Art des Entwurfs können Ingenieure die weitaus meisten praktischen Probleme lösen!

Entwurf-durch-Innovation

Im Gegensatz dazu erfordert der *Entwurf-durch-Innovation* die Entwicklung neuartiger Lösungen für bislang unbekannte Aufgabenstellungen. Diese innovative Arbeit mag spektakulärer oder auch interessanter sein, in der Praxis tritt sie jedoch höchst selten auf. Im Bereich der praktischen Informatik entstehen die meisten Softwaresysteme in bekanntem Kontext. Dennoch arbeiten Informatiker häufig nach dem Verfahren *Entwurf-durch-Innovation*. Sie verzichten dadurch auf Wiederverwendung und erhöhen gleichzeitig das Projektrisiko.

- Entwerfen Sie möglichst „durch-Routine“. Verwenden Sie erprobte Konzepte und Ansätze weiter. Pflegen Sie in Ihrer Organisation ein Wertesystem, das Wiederverwendung höher bewertet als Neuerfindung.
- Zahlreiche solche Muster finden Sie in der Literatur (etwa: [Gamma95], [Buschmann96], [Larman2001], [Fowler02]).
- Muster zu vielen Aspekten der Software-Entwicklung finden Sie im Internet unter den Tagungsberichten der Pattern-Konferenzen. Ein guter Startpunkt ist die Website *www.hillside.net/~patterns*

Der nachfolgende Katalog von Architekturaspekten unterstützt den *Entwurf-durch-Routine* und hilft Ihnen, Software ingenieurmäßig zu entwerfen, indem er Lösungsansätze für häufig wiederkehrende Architekturfragen aufführt.

Darum technische Konzepte und Architekturaspekte

In diesem Buch fasse ich unter der Überschrift *Technische Konzepte und Architekturaspekte* einige der übergreifenden und querschnittlichen *technischen* Themen zusammen, die Softwarearchitekten bei der Entwicklung von IT-Systemen häufig und intensiv beschäftigen. Zu solchen Konzepten gehören übergreifende oder querschnittliche Themen, die meistens mehrere Bausteine des Systems betreffen. Diese Konzepte prägen die Bausteinstrukturen oder deren Implementierung nachhaltig. Sie stellen oftmals zentrale technische Entscheidungen dar. Ziel dieses Kapitels ist es, Ihnen den Kontext dieser Konzepte oder Architekturaspekte und erste Lösungsansätze aufzuzeigen und Sie dann auf die einschlägige und vertiefende Literatur zu verweisen. Diese Lösungsansätze haben den Charakter grober Schablonen, die Sie für den konkreten Einsatz modifizieren oder anpassen müssen.

Dennoch lohnt sich die Anwendung solcher Vorlagen in der Praxis, weil Sie als Architekt mit Hilfe solcher Vorlagen beim Entwurf von Softwarearchitekturen drei wichtige Vorteile nutzen können:

- Sie sparen Zeit, weil Muster und Bausteine Ihnen Teile Ihrer Modellierungs- und Entwurfsarbeit erleichtern.

Zeit sparen

- Sie vermindern Unsicherheit, weil die Beschreibung der Muster und Bausteine Sie auf mögliche Risiken oder Problemzonen der jeweiligen Bereiche hinweist.

Risiken mindern

- Sie vermindern das Projektrisiko, weil es sich bei den Mustern um praktisch erprobte Vorlagen handelt und Sie konkrete Tipps für deren Einsatz bekommen.

Das Buch deckt aus Platzgründen nur einige solcher technischen Konzepte ab. Zur Vertiefung finden Sie kommentierte Literaturhinweise.

Technische Konzepte beinhalten konkrete Technologien

Oft manifestieren sich in technischen Konzepten die Details der jeweiligen Implementierungstechnologie oder entsprechender Frameworks. Die technischen Konzepte beeinflussen die Bausteinstrukturen von Systemen – und umgekehrt.

In diesen Konzepten steckt technisches Wissen, und meistens können Sie solche Konzepte nur auf Basis fundierter technischer Erfahrung erstellen. Hier ist es besonders nützlich, auf die Erfahrung der Entwicklungsteams zurückzugreifen.

Welche Arten technischer Konzepte oder Architekturaspekte gibt es?

Zunächst möchte ich Ihnen eine Vorstellung von der thematischen Breite solcher Architekturaspekte vermitteln. In den weiteren Abschnitten des Kapitels finden Sie zu einigen von ihnen Details.[1]

Diese Liste ist für Sie möglicherweise unvollständig! Stellen Sie für Ihre Projekte oder Domäne eine ähnliche Tabelle als Checkliste bereit, damit Sie beim Entwurf Ihrer Architekturen an alle für Sie wesentlichen Aspekte denken.

[1] Wobei ich rein subjektiv diejenigen Aspekte ausgewählt habe, die mir in den letzten Jahren in Projekten als besonders wichtig erschienen.

TABELLE 7.1 Architekturaspekte (nach einer Zusammenstellung aus [arc42])

Architekturaspekt	Bedeutung und Herausforderung
Persistenz (siehe Abschnitt 7.1)	Datenspeicherung, insbesondere von Objekt- und Klassenstrukturen auf Tabellen, Entkopplung von Fachdomäne und (Datenbank-)Infrastruktur
Geschäftsregeln (siehe Abschnitt 7.2)	Behandlung domänenspezifischer Zusammenhänge und Regeln als eigenständige („externalisierte“) Einheiten
Integration (siehe Abschnitt 7.3)	Einbindung bestehender Systeme in einen neuen oder veränderten Kontext
Verteilung (siehe Abschnitt 7.4)	Verteilung von Systembestandteilen auf getrennte Ablaufumgebungen
Kommunikation (siehe Abschnitt 7.5)	Übertragung von Daten zwischen Systemkomponenten, innerhalb und außerhalb von Prozess- oder Adressräumen
Grafische Oberflächen (siehe Abschnitt 7.6)	Steuerung von Navigation und Zustandswechseln in grafischen Benutzeroberflächen, Verarbeitung von Benutzereingaben und Ereignissen
Übergreifende Ablaufsteuerung, Workflowmanagement (siehe Abschnitt 7.7)	Steuerung systemübergreifender Abläufe, Koordination verschiedener Softwaresysteme
Sicherheit (siehe Abschnitt 7.8)	Methoden zur Gewährleistung von Datenschutz und -sicherheit, Verhinderung von Datenmissbrauch
Logging, Protokollierung, Tracing (siehe Abschnitt 7.9)	Sammeln von Informationen über den Programmzustand während der Laufzeit
Ausnahme- und Fehlerbehandlung (siehe Abschnitt 7.10)	Abweichungen von „erwarteten“ Situationen, Fehlverhalten oder Defekte von Systemteilen, unerwartetes Verhalten von Benutzern oder anderen Systemen
Batchverarbeitung	(Stapelverarbeitung): Datensatzweise Verarbeitung großer Datenbestände ohne Interaktion mit Benutzern
Transaktionsbehandlung	Transaktionen sind nicht teilbare Arbeitsabläufe, die in sich komplett abgeschlossen sind. Transaktionen müssen immer komplett und vollständig abgearbeitet oder aber „zurückgerollt“ werden. Solche Transaktionen müssen die ACID-Forderungen erfüllen: Atomicity, Consistency, Isolation und Durability.
Konfigurierbarkeit	Anpassung des Systems an besondere Umwelt- oder Umfeldbedingungen durch Änderungen von Installations-, Start- oder Laufzeitparametern
Zustands-, Session- oder Sitzungsbehandlung	Verwaltung von Client-und Server-Zuständen, Zuordnung von Clients zu spezifischen Sessions („Sitzungen“), insbesondere im verteilten Client/Server- und Web-Umfeld wichtig
Plausibilisierung und Validierung von Eingabe- oder Eingangsdaten	Wo und wann sollen Datenprüfungen stattfinden, um einen angemessenen Kompromiss zwischen Wartbarkeit („zentralisierte Prüfung“) und Laufzeit („Prüfung bei Dateneingang oder -eingabe“) zu finden?

7.1 Persistenz

Persistenz: Beständigkeit, Dauerhaftigkeit. Daten aus dem (flüchtigen) Hauptspeicher auf ein beständiges Medium (und wieder zurück) bringen.

7.1.1 Motivation

Daten müssen dauerhaft gespeichert werden

Viele Daten, die Software-Systeme bearbeiten, müssen dauerhaft auf einem Speichermedium gespeichert oder von solchen Medien gelesen werden:

- Flüchtige Speichermedien (Hauptspeicher oder Cache) sind technisch nicht für dauerhafte Speicherung ausgelegt. Daten gehen verloren, wenn die entsprechende Hardware ausgeschaltet oder heruntergefahren wird.
- Die Menge der von kommerziellen Software-Systemen bearbeiteten Daten übersteigt üblicherweise die Kapazität des Hauptspeichers.
- Auf Festplatten, optischen Speichermedien oder Bändern sind oftmals große Mengen von Unternehmensdaten vorhanden, die eine beträchtliche Investition darstellen.
- Einfache Speicherung bzw. Serialisierung von Daten in Dateien genügt in der Regel nur für sehr kleine Datenmengen und *single-user* Anwendungen.[2]

Persistenz ist ein technisch bedingtes Thema und trägt nichts zur eigentlichen Fachlichkeit eines Systems bei. Dennoch müssen Sie sich als Architekt mit dem Thema auseinandersetzen, denn ein erheblicher Teil aller Software-Systeme benötigt einen effizienten Zugriff auf persistent gespeicherte Daten. Hierzu gehören praktisch sämtliche kommerziellen und viele technischen Systeme. Eingebettete Systeme und Software für mobile Geräte gehorchen jedoch oft anderen Regeln hinsichtlich ihrer Datenverwaltung – was ich hier nicht weiter vertiefe.

Verschiedene Speichermodelle

Aufgrund der vielseitigen Anforderungen an performante, zuverlässige und plattformübergreifende Speicherung von Daten entstanden neben dem einfachen „Speichern in Dateien" verschiedene Arten von Datenbanksystemen[3] als Speichermodelle.

---

[2] Hier zeigt das pickle-Modul aus der Python-Standardbibliothek ein vorbildlich einfaches und verständliches Vorgehen auf. Falls Python wider Erwarten für Ihr Problem nicht ausreichen sollte, könnten Sie mit dem mächtigen Google-Protocol-Buffer an fast jedes Serialisierungs-Ziel gelangen: „Protocol Buffers are a way of encoding structured data in an efficient yet extensible format. Google uses Protocol Buffers for almost all of its internal RPC protocols and file formats." (*http://code.google.com/p/protobuf/*)

[3] Der korrekte Terminus lautet Datenbank-Management-Systeme (DBMS) – aber so genau nehme ich das meist nicht.

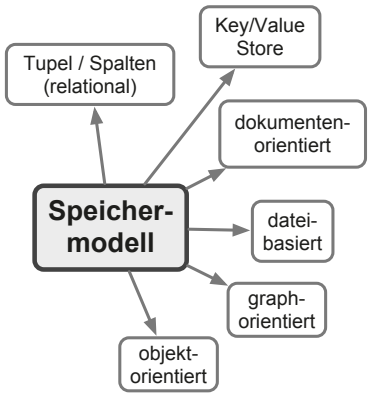


BILD 7.1
Speichermodelle

Typen von DBMS

Eine Datenbank bietet Mittel, um Daten zu persistieren und wieder in den Hauptspeicher zu laden. Es gibt verschiedene Paradigmen von Datenbanken (chronologisch sortiert):

- Netzwerkdatenbanken
- Hierarchische Datenbanken
- Relationale Datenbanken (Speichermodell Tupel/Spalten). Hierin werden Daten in Tabellen gespeichert, die über Schlüsselbeziehungen aufeinander referenzieren können. Relationale Datenbanken stoßen bei Clusterung, Replikation beziehungsweise automatischer Partitionierung von Datenbeständen an ihre Grenzen.[4] Viele Vertreter dieser Zunft glänzen durch höchste Zuverlässigkeit.
- Objektorientierte Datenbanken. Sie können Objekte und Objektgraphen direkt speichern und wiederherstellen. Trotz einfacher Programmiermodelle, hoher Robustheit und Performance sind diese Systeme in der Praxis nur wenig verbreitet.
- NoSQL Datenbanken[5] (alternative Speichermodelle beispielsweise Key/Value, Column-Stores, Graphen oder dokumentenorientiert). Hinter diesem Begriff verbirgt sich eine nicht klar abgegrenzte Menge von Datenbanksystemen, die meistens auf die Nutzung von SQL als Abfragesprache verzichten. Der Markt für diese Systeme ist sehr heterogen, viele sind relativ jung, viele Open Source oder zumindest frei verfügbar.

Bisher größte Bedeutung: Relationale DBMS

Die weitaus größte Bedeutung am Markt haben bisher relationale DBMS gewonnen. Netzwerk- und hierarchische Datenbanken kommen in neu entwickelten Systemen praktisch nicht mehr zum Einsatz, doch liegen immer noch große Mengen Unternehmensdaten auf Mainframe-Systemen in dieser Art von Datenbanken. Objektorientierte oder Graph-Datenbanken besitzen aktuell (Stand 2013) nur geringen Marktanteil.

NoSQL auf dem Vormarsch

Die sogenannten NoSQL-Systeme gewinnen im Web-Umfeld, insbesondere bei Anwendungen mit hohen Last-, Skalierungs- oder Performanceanforderungen, immer mehr an Bedeutung.

---

[4] Eine Ausnahme bilden RDBMS, die auf spezialisierter Hardware laufen. Dabei sind Hard- und Software auf Skalierung, Datenpartitionierung und Verteilung von Grund auf ausgelegt. Ein Nachteil sind die interessanten Listenpreise solcher Systeme ...

[5] NoSQL steht für Not-Only-SQL. Siehe [Edlich+11].

Allerdings können einfache Anwendungen auch mit dem Speichermodell „Datei“ auskommen, bei dem Daten durch Serialisierung in Dateien direkt auf dem Dateisystem abgelegt werden.

Verwenden Sie **relationale DMBS** für Anwendungen aus folgenden Bereichen:

- Anwendungen mit vielen Datensätzen gleicher (tabellarischer) Struktur (festes Schema). Änderungen an dieser Struktur sind selten.
- Betriebliche Informationssysteme, in denen zahlreiche, aber relativ einfache Operationen auf Daten ausgeführt werden.
- Anwendungen, die mit Fremdsystemen auf Basis relationaler DBMS arbeiten.
- Anwendungen, die vorhandene Mainframe-Legacy-Systeme integrieren. RDBMS passen oft recht gut zu den vorhandenen hierarchischen oder Netzwerk-Datenbanken solcher Systeme.
- Anwendungen in konservativem, organisatorischem Umfeld. RDBMS sind ausgereift und weit verbreitet, praktisches Wissen und Erfahrung problemlos verfügbar. Ihr Einsatzrisiko ist gering.
- Anwendungen, die auf „kurzen“ Transaktionen basieren. Relationale Datenbanken eignen sich nicht für lange Transaktionen, bei denen Objekte über Stunden oder Tage gesperrt werden müssen.

Verwenden Sie **objekt- oder graphenorientierte oder andere nicht-relationale DBMS** für Anwendungen aus folgenden Bereichen:

- Multimedia-Systeme, die komplexe Datenstrukturen wie Grafiken, Audio oder Video bearbeiten.
- Anwendungen aus den Bereichen: Computer-Aided-Design (CAD), Computer-Aided-Manufacturing (CAM) oder Computer-Aided Software Engineering (CASE).
- Anwendungen, die in hohem Maße von Vererbungshierarchien und benutzerdefinierten Datentypen abhängen (objektorientierte DB).
- Anwendungen, die komplexe Graphen speichern oder verarbeiten müssen (graphorientierte DB).

Sie sollten **NoSQL Datenbanken** für Anwendungen aus folgenden Bereichen in Erwägung ziehen:

- Anwendungen mit sehr hohen Benutzerzahlen und/oder sehr großem Datenvolumen. Beispiel: Große Webanwendungen, Börsensysteme mit hohen Transaktionsvolumina.
- Anwendungen mit sehr hohen Anforderungen an Skalierbarkeit beziehungsweise Verteilung auf sehr viele Knoten.
- Anwendungen, bei denen Sie durch Prototypen den Einsatz neuartiger Technologien wie NoSQL verifizieren können (d. h., bei denen Sie ausreichend Zeit für solche Prototypen zur Verfügung haben!).

Mischformen der Speicherung (= polyglott persistence) bilden eine mögliche Option – wobei durch zusätzliche Komponenten (hier: verschiedene DBMS) die Komplexität eines Systems steigt.

Einfache Ansätze führen zu Problemen

Im einfachsten Fall, dargestellt in der linken Hälfte von Bild 7.2, kümmert sich eine fachliche Komponente eines Systems selbst um die notwendige Speicherung ihrer Objekte. Dabei enthalten Klassen aus der Fachdomäne den Programmcode für die jeweiligen Datenbankoperationen, etwa als SQL-Befehle.

Dies führt zu sehr starker (und manchmal fataler) Abhängigkeit fachlicher Komponenten vom Persistenzmechanismus. Änderungen an der Datenbank erfordern dann meistens Änderungen am Quelltext der fachlichen Komponenten. Beispielsweise führt die Anpassung des Datenmodells (etwa aus Performancegründen) zu notwendigen Änderungen am Fachmodell. Testen des Systems funktioniert nur gegen die echte Datenbank, eine Mock-Datenbank ist ohne Änderung im Quellcode nicht möglich.

Selbst eine einfache Änderung, wie die Umbenennung einer Datenbanktabelle, lässt sich nicht ohne Änderung am Quellcode der Fachklasse durchführen.

Deutlich flexibler ist die Variante rechts im Bild 7.2: Eine Schnittstelle entkoppelt die fachlichen Bausteine von der Persistenz. Repository-Bausteine können diese Schnittstelle implementieren. Diese Struktur entkoppelt die fachlichen Bausteine komplett von der zugrunde liegenden Technologie (siehe Kapitel 6.2.3.3).

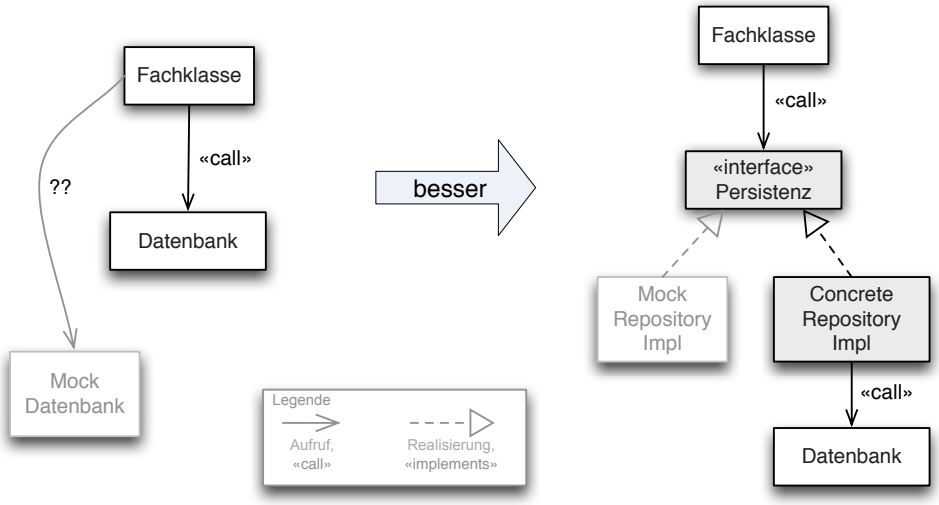


BILD 7.2 Einfache Ansätze für Persistenz

7.1.2 Einflussfaktoren und Entscheidungskriterien

Es gibt eine ganze Reihe von Einflussfaktoren und Entscheidungskriterien, die Sie beim Entwurf und der Implementierung von Persistenzkomponenten beachten sollten (siehe Bild 7.3).

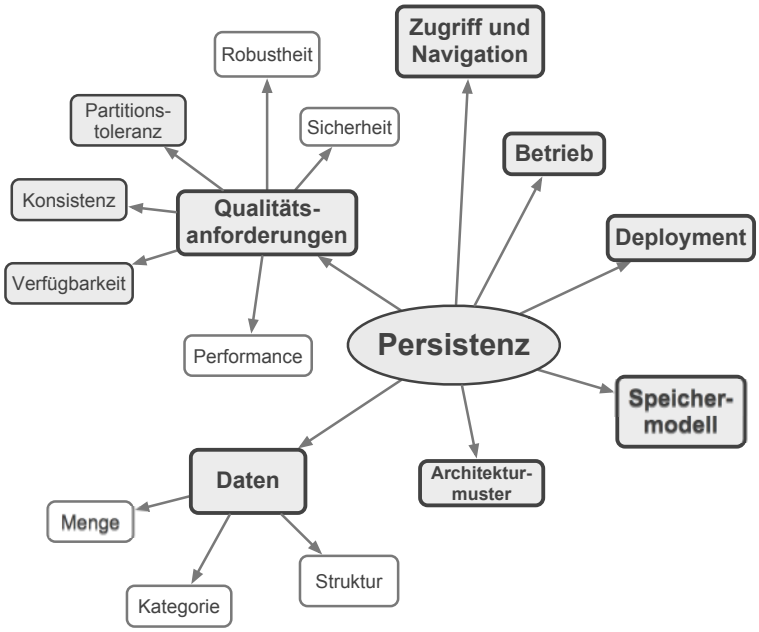


BILD 7.3 Überblick über Faktoren, die Einfluss auf die Persistenz haben

Neben Struktur, Kategorie und Menge der zu speichernden Daten spielen Qualitätsanforderungen, notwendige Zugriffe und Navigation, Betrieb, Deployment sowie die oben bereits erwähnten unterschiedlichen Speichermodelle eine große Rolle beim Entwurf der Persistenz. Auch aufgrund der aktuellen Entwicklungen im NoSQL-Bereich (siehe [Edlich+11]) sollte die Antwort auf die Frage „Worin speichern wir unsere Daten?“ heutzutage nicht mehr automatisch „in einer relationalen Datenbank“ lauten, sondern differenzierter untersucht werden.[6]

7.1.2.1 Art der zu speichernden Daten

Die Art der zu speichernden Daten prägt den Entwurf ihrer Persistenz maßgeblich. Hier sollten Sie drei wesentliche Fragenkomplexe klären:

- Besitzen die Daten eine feste Struktur („schemabehaftet“), oder sind sie unstrukturiert? Kann die Struktur über die Zeit variieren, oder bleibt sie lange Zeit fest? Relationale Datenbanken sind für feste tabellarische (relationale) Strukturen optimiert, können jedoch mit schwach strukturierten Daten variabler Struktur relativ schlecht umgehen, bzw. Sie müssten variable Datenstrukturen bereits im (festen!) Tabellenschema vorsehen.
- Welche Mengen oder Volumina von Daten müssen Sie speichern? Wenn diese Menge den verfügbaren Speicherplatz einzelner Rechnersysteme überschreitet, müssen Sie über Partitionierung Ihrer Daten nachdenken – wobei manche NoSQL-Systeme Sie hervorragend unterstützen und die meisten relationalen Datenbanken Ihnen eher Kopfschmerzen verursachen werden.

[6] Für unternehmensintern genutzte Anwendungen wird sicherlich die Antwort auch heute noch meistens „relationale Speicherung“ lauten; allerdings entstehen viele aktuelle Softwaresysteme im Kontext „Internet“ – und dabei existieren weitere Optionen, die in puncto Verfügbarkeit, Skalierbarkeit und Betriebskosten hervorragend abschneiden.

- Müssen Sie Daten und deren Strukturinformation versionieren? Müssen Sie eventuell sogar mit unterschiedlichen Versionen von Daten zur gleichen Zeit umgehen können? Manche NoSQL-Datenbanken bringen dafür Features wie *Multi-Version-Concurrency-Control* mit (etwa Apache's CouchDB). Das erlaubt Ihnen, auftretende Konflikte zwischen unterschiedlichen Versionen zum großen Teil automatisiert zu bearbeiten. In relationalen Strukturen müssen Sie das in der Regel selbst modellieren und implementieren.

7.1.2.2 Konsistenz und Verfügbarkeit (ACID, BASE oder CAP)

Die Verarbeitung von Daten in relationalen Datenbanken erfüllt in der Regel die ACID[7]-Eigenschaften, d. h. sämtliche Verarbeitungsschritte sind atomar, konsistent, isoliert und dauerhaft. Dies stellt die Verlässlichkeit von Transaktionen sicher.

Gerade im Web-Umfeld benötigen Anwendungen diese strikten Garantien nicht unbedingt. Blog-Kommentare oder Ihre Friends-Liste bei Facebook dürfen, zumindest für kurze Zeit, leicht inkonsistent sein. Diese gegenüber ACID etwas gelockerten Anforderungen werden in den BASE-Kriterien zusammengefasst: *basically available, soft state, eventually consistent*.

Hierzu schreiben [Edlich+11]:

> *„Bei BASE dreht sich im Gegensatz zu ACID alles um Verfügbarkeit. Konsistenz wird der Verfügbarkeit bei BASE untergeordnet. Wo ACID einen pessimistischen Ansatz bei der Konsistenz verfolgt, ist BASE ein optimistischer Ansatz, bei dem Konsistenz als ein Übergangsprozess zu sehen ist und kein fester Zustand nach einer Transaktion. Daraus entsteht ein völlig neuartiges Verständnis von Konsistenz: Eventually Consistency. Systeme, die nach BASE arbeiten, erreichen auch irgendwann den Status der Konsistenz, die Betonung liegt dabei aber auf irgendwann. Infolgedessen wird Konsistenz erst nach einem Zeitfenster der Inkonsistenz erreicht und nicht unmittelbar nach jeder Transaktion."*

CAP-Theorem: Neue Qualitätsanforderungen an Datenbanksysteme

In diesem Zusammenhang steht auch die Erkenntnis, dass in verteilten Systemen die Merkmale Konsistenz, Verfügbarkeit und Partitionstoleranz nicht vollständig vereinbar sind.

CAP-Theorem

Unter dem Titel CAP-Theorem (consistency, availability, partitiontolerance) hat Eric Brewer ([Brewer00]) diese Erkenntnis publiziert. Sie können jeweils zwei dieser drei Eigenschaften sicher erreichen, müssen dann jedoch Abstriche bei der dritten Eigenschaft hinnehmen. Siehe auch Bild 7.5.

Nun sind Konsistenz und Verfügbarkeit verständliche Begriffe – was hingegen bedeutet Partitionstoleranz? Dieser Begriff wurde erst durch die horizontale Skalierung von (NoSQL)-Datenbanken bedeutsam.

Stellen Sie sich vor, Ihr Datenbestand ist auf mehrere Knoten (= Partitionen) verteilt, die sich gegenseitig selbstständig aktualisieren (d. h. über Änderungen (insert, update) informieren). Diese Situation zeigt Bild 7.4. Wann meldet die Datenbank eine erfolgreiche Schreiboperation: Genügt es, die Daten in einer der Partitionen zu speichern (Fall 1), müssen es mehrere sein (Fall 2) oder müssen *alle* Partitionen erfolgreich speichern?

[7] Ausführliche Diskussionen von ACID finden Sie in der Standardliteratur zu Datenbanksystemen. [Edlich+11] enthält eine Gegenüberstellung von ACID und BASE.

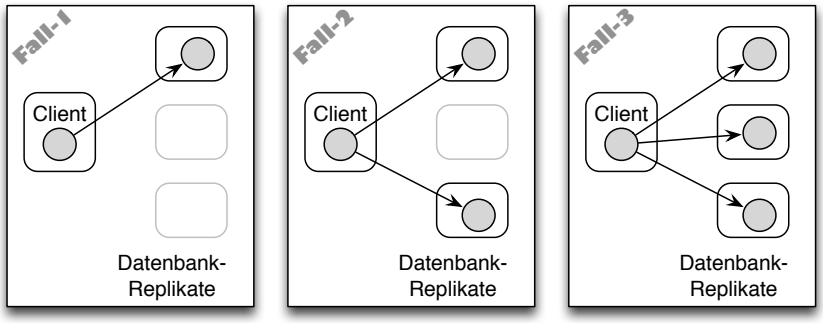


BILD 7.4 Partitionierung, oder: Wann ist eine Schreiboperation erfolgreich?

Partitionstoleranz gibt an, wie tolerant ein System gegenüber dem Ausfall einzelner Partitionen (bzw. der Verbindung zwischen einzelnen Partitionen) ist.

In Fall 1 könnte es zu kurzfristiger Inkonsistenz kommen: Aktualisiert ein Client Daten auf dem Knoten K1, während die Verbindung zwischen K1 und dem Rest des Systems unterbrochen ist, werden andere Clients bei Anfragen nach diesen Daten gegen die anderen Knoten des Systems andere (d. h. inkonsistente) Antworten erhalten.

Die ausführliche Diskussion dieses Thema füllt Bände[8] – der Einfachheit halber empfehle ich Ihnen den anschaulichen grafischen Überblick von Nathan Hurst („Visual Guide to NoSQL“)[9] – in Kurzform in Bild 7.5:

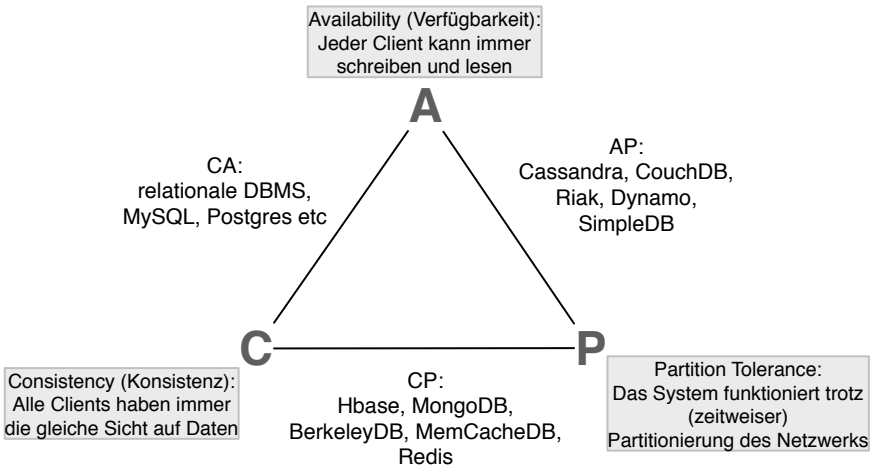


BILD 7.5 CAP-Eigenschaften einiger NoSQL-Datenbanken[8]

[8] Ich empfehle den Originalautor Eric Brewer, der eine aktualisierte Einschätzung des CAP-Theorems („12 Years later“) gibt: *http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed*

[9] Online in seinem Blog: *http://blog.nahurst.com/visual-guide-to-nosql-systems*

7.1.2.3 Zugriff und Navigation

Bezüglich der Zugriffe auf Ihre Daten und Navigation zwischen diesen sollten Sie prüfen, ob Sie charakteristische Muster für diese Zugriffe erkennen können. Beispielsweise müssen Sie bei den meisten Speichermodellen parallele Schreibzugriffe auf Daten über Sperren verhindern.

Sollten Sie allerdings fast ausschließlich lesende Zugriffe auf Ihre Daten erwarten, so können Sie diese durch Replikation, Parallelisierung, Caching und Indizierung signifikant beschleunigen – was allerdings möglicherweise Ihre Schreibperformance deutlich einschränkt.

Benötigen Sie die Möglichkeit, deklarativ und flexibel ständig neue Abfragen auf Ihre Daten zu formulieren? Dafür wurde SQL geschaffen, das Sie ans relationale Modell bindet. Feste Abfragen können Sie mittels Map-Reduce-Verfahren a priori ausrechnen lassen, was erhebliche Geschwindigkeitsvorteile mit sich bringen kann, allerdings zurzeit primär von NoSQL-Systemen unterstützt wird.

7.1.2.4 Deployment und Betrieb

Ob Ihr System auf einem leistungsfähigen Server-Cluster abläuft, einem Desktop oder Notebook, auf kleinen Smartphones oder sogar auf embedded-Devices – für die Ausgestaltung Ihrer Persistenz machen diese unterschiedlichen Ablaufumgebungen riesige Unterschiede. Ich empfehle Ihnen realitätsnahe Last- und Performancetests durch Prototypen, um die Leistungsfähigkeit Ihrer Ziel-Hardware möglichst frühzeitig zu prüfen.

Bei kommerziellen DBMS sollten Sie die Lizenzkosten für die konkrete Zielhardware, insbesondere bezüglich der Anzahl der Prozessoren, Prozessorkerne, Server-Instanzen oder Betriebsarten ins Kalkül ziehen.

Betriebliche Aspekte

Zur Laufzeit Ihrer Systeme kümmern sich in vielen Fällen dedizierte Personen (in der Rolle von Administratoren oder Betreibern) um geregelte, zuverlässige Abläufe. Diese Stakeholder fordern von Datenbanken folgende Eigenschaften:

- *Konfiguration:* Lässt sich die Datenbank an die Laufzeitumgebung oder an geänderte Anforderungen per Konfiguration anpassen?
- *Monitoring:* Lassen sich Laufzeitparameter und -verhalten automatisch überwachen?
- *Backup:* Über welche Möglichkeiten der Datensicherung verfügt die Datenbank? Ist eine Sicherung im laufenden Betrieb möglich? Lassen sich Zugriffsrechte auf Datensicherungen vergeben und Sicherungen oder bestimmte Teile davon sich zuverlässig wieder einspielen?
- *Portabilität:* Lässt sich die Datenbank mit anderen Betriebssystemen, virtuellen Maschinen oder Netzwerkkonfigurationen zuverlässig betreiben?

In der Realität könnten Ihre Betreiber auch nach Möglichkeiten zum Tuning fragen, zur Anbindung an vorhandene Rechte-Management-Systeme oder gar Migration. Ich gehe auf diese (schweren) Fragen hier nicht weiter ein.

7.1.3 Lösungsmuster

7.1.3.1 Persistenzschicht

Entkoppeln Sie in Ihren Architekturen den fachlichen Kern Ihrer Systeme von der gewählten (technischen!) Persistenz. Dazu stelle ich Ihnen hier die *Persistenzschicht* vor, die gut zum Modell der Anwendungsschichten (siehe Kapitel 6) passt.

Die hier vorgestellte Persistenzschicht gehört zur Infrastrukturschicht. Ob Sie die Persistenz als getrennte Anwendungsschicht modellieren oder als integralen Bestandteil der Infrastruktur, bleibt Ihnen überlassen.

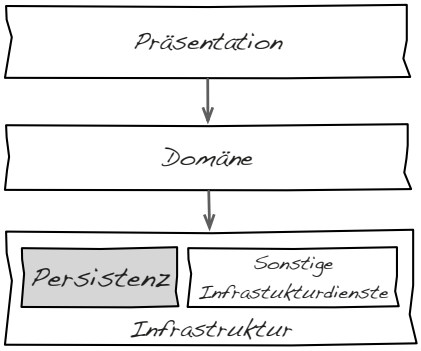


BILD 7.6
Persistenzschicht und Infrastrukturschicht

Wie funktionieren Persistenzschichten?

Eine Persistenzschicht kapselt die Details der Datenspeicherung gegenüber den übrigen Anwendungsteilen. Sie können Persistenzschichten unterschiedlich flexibel und aufwendig realisieren:

- Sie implementieren eine eigene Klasse, um die Persistenzaufgaben zu kapseln. Der folgende Abschnitt („Datenklassen“) diskutiert einige Vor- und Nachteile dieses Ansatzes.
- Sie verwenden ein Framework, wie beispielsweise Hibernate. Das bringt Ihnen etwas höhere Komplexität, allerdings auch ein erhebliches Maß an Flexibilität und Robustheit. Für relationale Datenbanken stellt dies eine weit verbreitete Standardlösung dar, die viele Risiken von Persistenz mindert. Allerdings orientieren sich viele Persistenz-Frameworks heute noch an relationalen Datenbanken. Für NoSQL-Speichermodelle müssen Sie unter Umständen selbst Lösungen entwickeln.

Datenklassen: Die einfachste Persistenzschicht

Datenklassen

Active Data Objects (ADO)

Wenn Sie die Persistenz Ihrer Fachklassen in dedizierten Datenklassen kapseln, erreichen Sie eine bessere Kapselung von Fachdomäne und Infrastruktur. Diesen Ansatz illustriert Bild 7.7. Er wird unter anderem in dem von Microsoft favorisierten Ansatz der „ActiveX Data Objects“ (ADO) verfolgt.

Änderungen an der Datenbank oder dem zugrunde liegenden Datenmodell bewirken dabei „nur“ Änderungen an den Datenklassen, nicht aber an den Fachklassen. Eine vollständige Trennung von Anwendungslogik und Infrastruktur erreicht dieser Ansatz nicht – Ihre Entwickler müssen immer noch die technischen Grundlagen der Datenbank beherrschen.

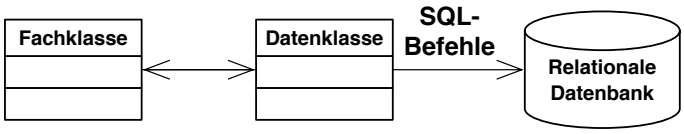


BILD 7.7
Persistenzschicht mit Datenklassen

Schwierig ist dieser Ansatz auch dann, wenn Sie vernetzte Objektstrukturen speichern wollen, und nicht nur „einzelne" Objekte. Die Abhängigkeit zwischen Klassenstrukturen und Tabellenstrukturen ist sehr hoch, was Änderung und Optimierung erschwert.

Verwenden Sie diesen Ansatz in folgenden Fällen:

- Sie benötigen keine flexible und hochgradig wartbare Persistenzschicht, und Ihre Anwendung verfügt nur über wenige Fachklassen.
- Sie persistieren nur einzelne Objekte, keine vernetzten Objektstrukturen. Die Vererbungstiefe ist klein.
- Endtermin und Entwicklungskosten werden in Ihrem Projekt deutlich höher bewertet als Flexibilität und Wartbarkeit. Höhere Änderungs- und Anpassungsaufwände werden von Kunden und Auftraggebern akzeptiert.
- Die zu speichernden Klassen des fachlichen Modells sind Datencontainer, haben viele Attribute und wenige Methoden. Ihre Abbildung auf relationale Strukturen ist sehr einfach und keinen Änderungen unterworfen.
- Es gibt in Ihrer Organisation keine wiederverwendbare Persistenzschicht und kein (kommerzielles) Persistenz-Framework.

Anforderungen an Persistenzschichten

Wenn Sie eine Persistenzschicht oder eine Persistenzkomponente entwerfen, sollten Sie folgende Aspekte als mögliche Anforderungen beachten:[10]

- Einfache Kapselung des Persistenz-Mechanismus. Fachliche Klassen benutzen lediglich Methoden wie save(), read() oder delete(), alle weiteren Dinge verbirgt die Persistenzschicht.
- Vollständige Kapselung der Persistenz. Dabei werden auch die oben erwähnten Methoden vor den Fachobjekten versteckt: read() verbirgt sich in einer Factory, save() erledigt ein Transaktionsmechanismus (alles, was „dirty" (geändert) ist, wird gespeichert), delete() bearbeitet ein Destruktor.
- Unterstützung unterschiedlicher Persistenzmechanismen oder Speichermodelle: Datenbanksysteme, Dateien oder auch Legacy-Systeme.
- Unterstützung von Vererbungshierarchien und Polymorphismus.
- Unterstützung von Transaktionen.

[10] Nach [Ambler2000] und [Keller98]. In der Realität habe ich noch viele weitere Anforderungen erlebt: Locking- und Caching-Strategien, spezielle Anfragesprachen, Connection-Pooling, spezielle Assoziationen zwischen persistenten Objekten, kaskadierende Operationen, Replikation, Backup, Benutzer- und Rollenkonzepte, Verschlüsselung ... Bei dieser Menge an Anforderungen denken Sie hoffentlich noch mal über das Selbst-Entwickeln nach ...

- Unterstützung von Lazy-Loading: Es werden immer nur diejenigen Teile von Objektgraphen aus der Datenbank in den Speicher geladen, die man aktuell benötigt. Hilfreich wäre auch das Pendant – Eager-Loading: Laden auf Vorrat.
- Bearbeitung mehrerer Objekte auf einmal, analog dem Cursor-Konzept von SQL.
- Automatische Erzeugung eindeutiger Object-Identifier (OIDs).
- Unterstützung mehrerer Verbindungen (connections) parallel. Dies ist bereits für das Kopieren von Objekten von einer Datenbank in eine andere notwendig.
- Eine programmatische Schnittstelle (= API) für die Erstellung und Ausführung von Abfragen (Queries).
- Generierung von Migrationsskripten bei Änderungen des Datenmodells.
- Toleranz gegenüber Maßnahmen zum Performance-Tuning von Datenbanken. Solche Optimierungen kommen recht häufig vor und sollten ohne Änderungen am Quellcode von Anwendungssystemen funktionieren.

Einige Fragen und Antworten zu Persistenz

von Wolfgang Keller

- *Zu welchem Zeitpunkt in einem (objektorientierten) Projekt sollte man die relationale Datenbank entwerfen?*
  Entwerfen Sie die Tabellen auf Basis des Objektmodells, nachdem Sie einen Architektur-Prototyp implementiert haben.
- *Was ist eine gute Struktur für ein Persistenz-Subsystem?*
  Sie sollten zwei Subsysteme für die Persistenz vorsehen, die eine Schichtenstruktur bilden. Die obere Schicht (Objektschicht, *object layer*) kapselt die Konzepte der Objektorientierung, während die untere Schicht (*storage layer*) eine Schnittstelle zu Ihren physikalischen Speichermedien, Datenbanken oder Dateien bildet.
- *Wie verbindet man eine Persistenzschicht mit einem transaktionsbasierten Mainframe-Datenserver?*
  Benutzen Sie einen Agenten oder Vermittler, der Aufrufe an den Mainframe bündelt (*request bundling*) und sie als Block unter der Kontrolle des (Mainframe) Transaktionsmonitors ausführt.
- *Wie repräsentiert man die Individualität eines Objekts in einer relationalen Datenbank?*
  Erzeugen Sie synthetische Schlüssel (*object identifier*), die jedes Objekt von seiner Geburt bis zu seiner Zerstörung begleiten. Begraben Sie diesen Schlüssel mit dem Objekt.
- *Wie stellt man die Identität eines Objekts sicher (und behält sie)?*
  Erzeugen Sie einen Objekt-Cache für jeden Client-Prozess der Datenbank. Dieser Cache basiert auf einem Container (Array, Vektor, Set oder Ähnliche), der Object-Identifier auf Objekte (repräsentiert durch Pointer oder Proxies) abbildet. Der Object Cache enthält Zeiger auf die instantiierten Objekte, die aus der Datenbank gelesen wurden.

- *Wie verhindert man, dass alle „verwandten" (assoziierten) Objekte geladen werden, wenn man ein Objekt bearbeitet, das viele Beziehungen zu anderen Objekten hat?*
  Benutzen Sie Proxy-Objekte, die nur den Object-Identifier (OID) enthalten, sowie einen Zeiger. Dieser Zeiger kann auch NULL sein – das Objekt wird erst dann geladen, wenn es tatsächlich benötigt wird.
- *Wie handhabt man Transaktionen?*
  Machen Sie aus einer Transaktion ein Objekt mit Methoden begin, commit, rollback, retry (TransactionObject, siehe [Keller97]).
- *Wie füllt man Auswahllisten, List-Boxen oder Combo-Boxen?*
  Erzeugen Sie passende Datenbank-Views. Diese sollten nur die notwendigen Attribute plus einen Primärschlüssel enthalten (*Narrow Views*).
- *Wie kann man das Füllen von Auswahllisten oder Combo-Boxen noch weiter beschleunigen (und den Transfer nicht benötigter Daten vermeiden)?*
  Laden Sie diese Daten in Blöcken mit vertretbarer Zugriffs- oder Ladezeit. Als Daumenregel empfehlen wir etwa 30–50 Datensätze oder das Doppelte der Zeilenzahl der Auswahlliste (*Short Views*).
- *Wie kann man den performanten Zugriff auf große Datenmengen durch eine objektrelationale Persistenzschicht gewährleisten?*
  Benutzen Sie „stored procedures" oder eine spezialisierte Zugriffskomponente ausschließlich für diesen Zweck. Diese sollte exakt die benötigten Daten auf einmal laden (*Cluster Read*).
- *Wie verkürzt man die Wartezeit, wenn große Datenmengen in eine Datenbank geschrieben werden sollen?*
- Sammeln Sie die Schreiboperationen innerhalb der Persistenzschicht, und schicken Sie diese en bloc an die Datenbank (*bundled write*).
- Speichern Sie Daten in einem lokalen Puffer, und geben Sie die Kontrolle an das System zurück. Starten Sie einen parallelen Thread, der diesen Puffer an die Datenbank übermittelt (*store for forward*).
- Schreiben Sie die Daten in eine (eventuell transaktionsgesicherte) Datei, und laden Sie diese Datei zu einem späteren Zeitpunkt in die Datenbank (*flat file write*).
- Arbeiten Sie asynchron, etwa durch eine Warteschlange (*message queue*). Kommerzielle Message-Queue-Produkte unterstützen dabei auch Transaktionen.

Diese Fragen und Antworten repräsentieren einen kleinen Teil der Entwurfsmuster für objektrelationale Persistenzschichten, die [Keller98] und [Keller97] ausführlich vorstellen. Bevor Sie selbst eine Persistenzschicht implementieren, sollten Sie diese Entwurfsmuster gründlich lesen.

Wolfgang Keller (Email: wk@objectarchitects.de) arbeitet als selbstständiger Projektmanager und Berater für Unternehmensarchitekturen. Davor war er u. a. Entwicklungsmanager bei der Generali Vienna Group, an der Entwicklung mehrerer großer OO-Systeme beteiligt, hat Zugriffsschichten selbst implementiert und zugekaufte verwendet. Als Hobby betreibt er eine Website, unter anderem mit Patterns zum Thema Persistenz (www.objectarchitects.de).

7.1.3.2 DAO: Eine Miniatur-Persistenzschicht

Es gibt ein bedeutendes Entwurfsmuster, das Sie im Zusammenhang mit Persistenz kennen sollten, das Data Access Object (DAO). Es stellt eine Art Miniatur-Persistenzschicht dar und entkoppelt die eigentliche Datenspeicherung von der jeweiligen Anwendung. DAOs arbeiten meist mit sogenannten Data-Transfer-Objects (DTO) oder Transferobjekten zusammen. Sie erhöhen die Flexibilität von Systemen hinsichtlich Details der Persistenzschicht und vereinfachen dadurch beispielsweise Versionswechsel zugrunde liegender Datenbanken oder ähnliche Änderungen. In Bild 7.8 und 7.9 sehen Sie die statische Struktur sowie den schematischen Ablauf.

In Java-Systemen, insbesondere ab JEE6, machen die Annotationen für Persistenz den Einsatz des DAO-Patterns in den meisten Fällen obsolet.[11]

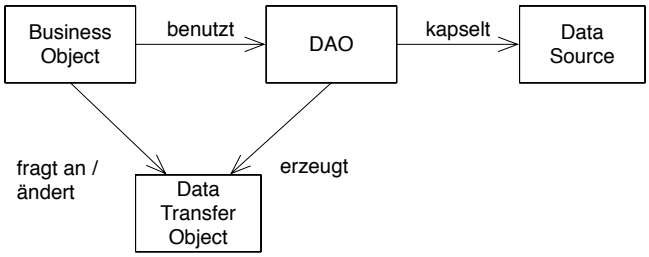


BILD 7.8
Struktur des DAO-Musters

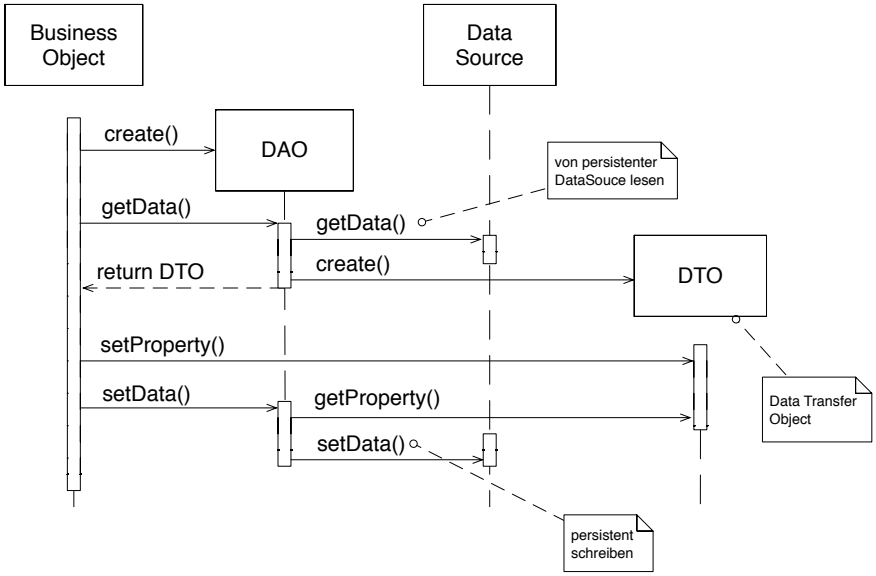


BILD 7.9 Schematischer Ablauf bei Nutzung des DAO-Musters

[11] Der Java- und Tomcat-Guru Peter Roßbach nennt das „POJO schlägt DAO“.

7.1.4 Bekannte Risiken und Probleme

Risiko durch Abhängigkeit von technischen Details des DBMS

Praktisch alle Anbieter von Datenbanksystemen (relationale wie auch nicht-relationale) entwickeln ihre Produkte kontinuierlich weiter. Als Architekt müssen Sie daher bei Versionswechsel Ihrer Datenbank mit Änderungen am Quellcode Ihrer Systeme rechnen, es sei denn, eine Persistenzschicht (siehe folgender Abschnitt) kapselt solche Änderungen.

Es ist grundsätzlich möglich, während der Lebens- und Nutzungsdauer eines Systems den zugrunde liegenden Persistenzmechanismus auszutauschen, etwa zwecks Performancesteigerung oder aufgrund geänderter Einsatzszenarien der betroffenen Systeme. Falls Ihre Systeme modular aufgebaut sind und die Persistenz sauber kapseln, sollten solche Operationen mit überschaubarem Risiko möglich sein. In allen anderen Fällen sollten Sie solche Änderungen als „Operation am offenen Herzen“ betrachten und entsprechende Risikovorsorge treffen.

Seiteneffekte, wenn Geschäftslogik in der Datenbank abläuft

Insbesondere relationale Datenbanken verfügen meist über sogenannte *Trigger*: Operationen, die beim Eintreten bestimmter Bedingungen direkt von der Datenbank ausgelöst werden.

Solche Trigger können beispielsweise Aufräumoperationen wie „kaskadierendes Löschen“ implementieren, aber auch fachliche Logik („Wenn das Attribut Rechnungssumme > 100, dann erhöhe die Rabattstufe um eins“).

Sie ahnen sicherlich die Risiken dieser Trigger: Sie sind nicht Bestandteil des eigentlichen Sourcecodes Ihres Systems, operieren jedoch auf dessen Daten. Aus Sicht Ihres Systems (sei es Cobol, Java oder C#) realisieren Trigger *Seiteneffekte* – ein riskantes Unterfangen.

Ähnliches könnte Ihnen mit Stored-Procedures oder sogar (modifizierenden) Views passieren.

Ich habe Situationen erlebt, in denen Dutzende von Triggern auf der Datenbank eine wilde Mischung technischer und fachlicher Aufgaben realisierten, zwar hochperformant, aber weitgehend undokumentiert – parallel zur eigentlichen Anwendung.

Die Wartbarkeit und Erweiterbarkeit dieses Systems wurde durch diese Trigger erheblich negativ beeinflusst. Über die langjährige Lebensdauer dieses Systems bekamen die Trigger nach und nach den Status von „Einflussfaktoren und Randbedingungen“ anstelle von Lösungsmechanismen.

Setzen Sie Datenbank-Trigger daher mit Vorsicht ein.

- Vermeiden Sie es, fachliche Logik oder Regeln durch Trigger zu implementieren.
- Verwenden Sie Trigger nur, wenn Ihre normale Programmiersprache keine angemessene Möglichkeit für die betreffenden Operationen bietet.
- Dokumentieren Sie Trigger – nehmen Sie insbesondere in Ihren „normalen“ Quellcode Verweise auf die Trigger auf!

Kooperation mit Mainframe-Systemen

Manche Organisationen speichern große Teile ihrer Unternehmensdaten auf Mainframes (Großrechnern, Host-Systeme). Zugriffe und Speicherung dieser Daten erfolgen dann über Mainframe-Programme und deren häufig Cobol-basierte Schnittstellen.

Plausibilitäten

Viele dieser Mainframe-Programme enthalten fachliche Logik (Plausibilitätsprüfungen), die das Speichern und Lesen von Daten beeinflussen. Manchmal ist es schwierig, die bestehende Logik mit der Fachlichkeit des neuen Systems zu koordinieren. Insgesamt bedeutet die Interaktion mit solchen Systemen eine zusätzliche Komplexität durch technische und fachliche Abhängigkeiten.

Strukturbrüche

Impedance mismatch

In vielen Fällen kommt es bei der Entwicklung von Software-Systemen zu einem Strukturbruch (dem sogenannten impedance mismatch) zwischen Programmiersprache und Datenbanksystem. Der typische Fall dieses Problems: Objektorientierte Programme speichern Daten in relationalen Datenbanken. Objektstrukturen im Hauptspeicher müssen in Tabellenstrukturen im Datenbanksystem übersetzt werden.[12] Weder syntaktisch noch semantisch ist diese Übersetzung eindeutig – ein weites und schwieriges Feld für Architekten!

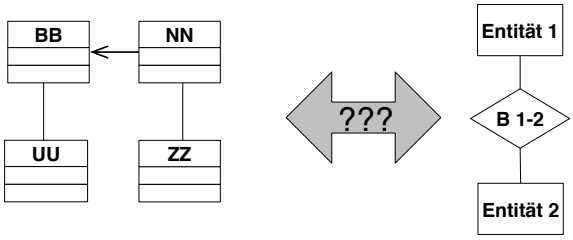


BILD 7.10
Strukturbruch zwischen objektorientierten und relationalen Modellen

7.1.5 Weitere Themen zu Persistenz

Vererbung Aggregation Assoziation

Zur Behandlung der Persistenz von Systemen gehören nach meiner Erfahrung neben dem Entwurf der Persistenzschicht noch folgende Aspekte:

- Abbildung von Objektstrukturen auf Relationen (*OR-Mapping*)
- Entscheidung zwischen Eigenentwicklung oder Kauf (*Make-or-Buy*)

Abbildung von Objektstrukturen auf Relationen

OR-Mapping

Die Abbildung von Objekt- und Klassenstrukturen (Objekt-relationales Mapping, OR-Mapping) auf Relationen löst den Strukturbruch zwischen diesen beiden Paradigmen auf (siehe Bild 7.11). Für die Basiskonstrukte objektorientierter Sprachen (Vererbung, Aggregation, komplexe Typen und Assoziationen) legt ein solches Mapping fest, wie die Übersetzung in relationale Strukturen erfolgt.

[12] Leider lösen moderne NoSQL-Datenbanken dieses Mapping-Problem nicht automatisch; vielmehr gibt es inzwischen diverse neue Kategorien von Mappings: Objekte-auf-Dokumente, Objekte-auf-Key-Value-Repräsentation sowie beliebige Mischungen davon. Trotz NoSQL ist die Persistenzwelt also immer noch schwierig. ☺

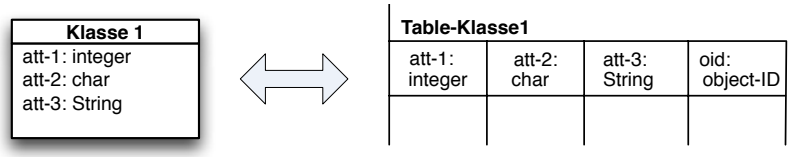


BILD 7.11 OR-Mapping für einfache Attribute

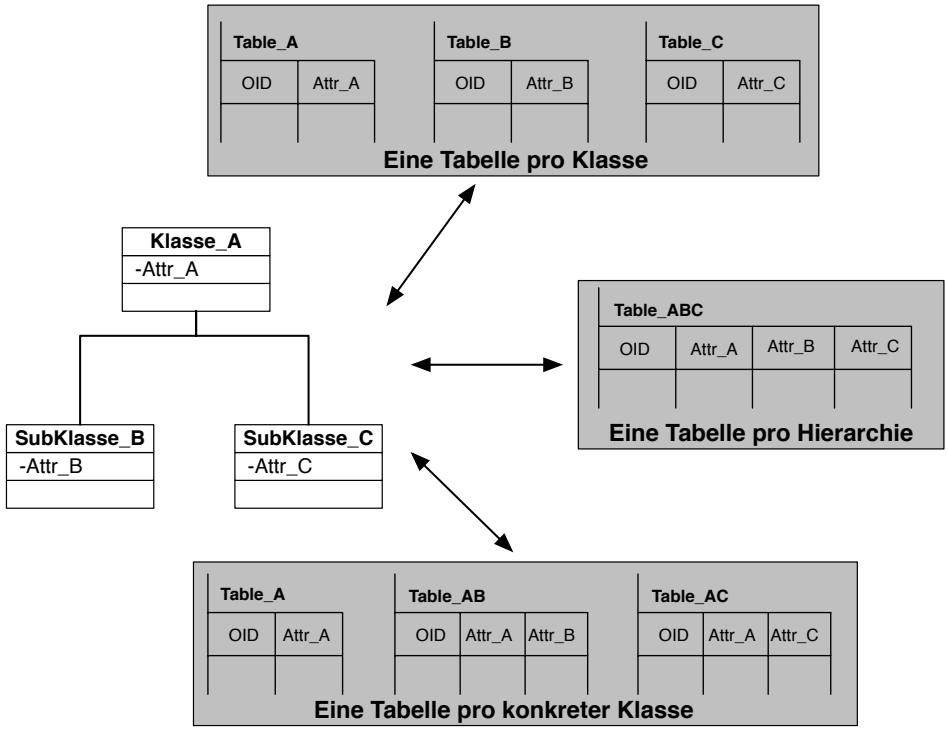


BILD 7.12 Abbildung von Vererbungshierarchien auf Tabellen

Grundsätzlich wird eine Klasse auf eine Tabelle abgebildet, indem Attribute von primitiven Datentypen der Klasse zu Spalten der Tabelle werden (siehe Bild 7.12).

Jedes Objekt benötigt eine eindeutige Kennzeichnung (object identifier, OID, Objektschlüssel), weil Objekte trotz identischer Attributwerte unterschiedliche Identität haben können. Diese Identifikation ist bei relationalen Tupeln nicht automatisch gegeben. In Bild 7.12 ist der OID als zusätzliches Attribut in der Tabelle dargestellt.

- Komplexe Attribute und Aggregationen müssen durch Zerlegungsalgorithmen in „flache“ Strukturen überführt werden. Dies gilt auch für die in objektorientierten Systemen häufig eingesetzten *Collections* (etwa: Vector, HashTable, Dictionary, Tree, Set etc.) sowie andere rekursive Strukturen. Hierzu gibt es verschiedene Ansätze:
  - Denormalisierung: Alle Attribute der aggregierten Klasse werden in die Tabelle der aggregierenden Klasse aufgenommen.

  - Aggregation über Fremdschlüsselbeziehung: Zwischen den beteiligten Tabellen wird über den OID eine Fremdschlüsselbeziehung aufgebaut.
  - Benutzung einer Überlauftabelle: Die ersten k Objekte einer 1:n-Assoziation oder einer Aggregation werden mit in die Tabelle zu einer Klasse aufgenommen, alle weiteren in der Überlauftabelle gespeichert.
- Assoziationen werden zu Fremdschlüsselbeziehungen (*foreign key relations*), wobei der Objektschlüssel (OID) des referenzierten Objektes zum Fremdschlüsselattribut wird.
- Zur Abbildung von Vererbung gibt es unterschiedliche Strategien (siehe Bild 7.12):
  - Eine Tabelle pro Klasse der Vererbungshierarchie: Die Attributwerte eines Objektes aus einer abgeleiteten Klasse befinden sich dabei verteilt auf alle Tabellen der Klassen, die vor ihm in der Hierarchie stehen. Objekte werden durch Join-Operationen über alle betroffenen Tabellen gelesen.
  - Eine Tabelle pro Hierarchie. Diese Tabelle (in Bild 7.12 „Table_ABC“) vereinigt die Attribute aller Klassen einer Klassenhierarchie in einer Tabelle. Hierbei benötigt man kein Join mehr zum Zusammensetzen eines Objektes aus einer abgeleiteten Klasse.
  - Eine Tabelle pro konkreter Klasse. Es gibt keine Tabelle für abstrakte Basisklassen. Für jede konkrete Klasse existiert eine Tabelle, in der ihre Attribute und die Attribute aller in der Hierarchie vor ihr stehenden Klassen aufgenommen werden.

Zur Entscheidung für eine konkrete Strategie des Mappings müssen Sie einige Aspekte berücksichtigen:

- Die Struktur Ihrer Objekte zur Laufzeit. Wie häufig sind polymorphe Objekte? Wie umfangreich werden Aggregationen im Durchschnitt? Wie tief sind die Vererbungshierarchien?
- Benötigen Sie den Zugriff auf Objekte über ihre jeweiligen Oberklassen? Dann brauchen Sie Unterstützung für Polymorphismus. Bevor Sie den mühevoll implementieren, erwägen Sie, auf Implementierungsvererbung ganz zu verzichten und stattdessen nur Protokollvererbung zu benutzen.
- Die Struktur der Anfragen (queries). Wie häufig resultieren Anfragen in (zeitaufwendigen) Joins auf der Datenbank? In welchem Zahlenverhältnis stehen die CRUD-Operationen zueinander (gibt es etwa mehr lesende oder mehr schreibende Operationen)?
- Welche Anfragen lassen sich durch ein alternatives Mapping optimieren?
- Wie unterscheiden sich Schreib- und Leseverhalten des Mappings?
- Wie speicheraufwendig ist ein bestimmtes Mapping?
- Wie gewichten Kunden und Auftraggeber Speicherplatz und Performance gegeneinander? Manche Mapping-Strategien nutzen Speicherplatz optimal aus und sind dafür bei vielen Operationen langsamer.

Mapping-Strategie

In der Realität werden Sie für Ihr konkretes Objektmodell oft eine Mischung aus den unterschiedlichen Strategien wählen, die sich an den konkreten Bedürfnissen orientiert. Tabelle 7.2 (in Anlehnung an [Keller97]) zeigt einige Vor- und Nachteile verschiedener Mapping-Strategien auf.

Eine flexible Persistenzschicht sollte in jedem Fall unterschiedliche Mapping-Strategien unterstützen.

TABELLE 7.2 Vergleich der Mapping-Strategien von Vererbungsbeziehungen

	Performance			Platzbedarf	Änderbarkeit
	Schreiben	Lesen	Polymorphes Lesen		
Eine Tabelle pro Klasse	–	–	–o	+	+
Eine Tabelle pro Vererbungspfad	+	+	–	+	–
Eine Tabelle pro konkreter Klasse	+o	+o	+	–	+

Legende: + = Gut, - = Schlecht, * = irrelevant, o = neutral

Weitere Patterns zur Persistenz

In seinem aus meiner Sicht sehr hilfreichen Buch „Patterns of Enterprise Application Architecture“ ([Fowler]) beschreibt Martin Fowler eine Reihe weiterer Lösungsmuster, die für den Entwurf von Persistenzkonzepten nützlich sein können. Eine ausführliche Darstellung würde den Rahmen dieses Buches sprengen, doch sind zumindest Kurzbeschreibungen seiner Patterns online[13] verfügbar.

Beachten Sie insbesondere die folgenden Themen:

- Data Source Architectural Patterns, etwa: Table und Row Data Gateways, Active Record sowie Data Mapper
- objektrelationale Verhaltensmuster, etwa: Lazy Load
- objektrelationale Strukturmuster, etwa: Identity Field, Foreign Key Mapping, Single, Class oder Concrete Table Inheritance

Eigenentwicklung oder Kauf (*Make-or-Buy*)?

Eine leistungsfähige und flexible Persistenzschicht zu entwickeln ist schwierig und aufwendig. Sie ist abhängig von den zugrunde liegenden Datenbanken. Versions- oder Herstellerwechsel ziehen Änderungen der Persistenz-Komponente nach sich. Diese Aspekte erschweren die Erstellung und Pflege einer eigenen Persistenzschicht. Für relationale Datenspeicherung rate ich Ihnen vom Eigenbau fast grundsätzlich ab. Für Datenspeicherung in NoSQL-Systemen verbreiten sich zurzeit (Stand 2013) erste Systeme.

Sie sollten in einem funktionalen Prototyp die Leistungsmerkmale verfügbarer Persistenzschichten mit den Anforderungen Ihres Systems abgleichen (und dabei alle beteiligten Legacy-Systeme berücksichtigen). Erst auf dieser Grundlage sollten Sie eine Make-or-Buy-Entscheidung treffen.

Da im Bereich „Persistenz“ viele Frameworks quelloffen sind, können Sie durchaus diese Frage erweitern auf „make, modify or buy?“

[13] *http://martinfowler.com/eaaCatalog/*

7.1.6 Zusammenhang zu anderen Aspekten

Sicherheit

Sie müssen gewährleisten, dass die Sicherheit der bearbeiteten Daten auch über die Persistenzschicht hinweg gewährleistet bleibt. Außerdem dürfen auch Protokolle oder Logs keinerlei sicherheitsrelevante Informationen enthalten.

Beispiel: In einem System wurden sensible Personendaten bearbeitet und gespeichert. Das System selbst wurde durch ausgeklügelte technische und organisatorische Mechanismen (Smartcards, 4-Augen-Prinzip) vor Missbrauch geschützt. Die Persistenzschicht erlaubte durch ein Benutzer-/Rollenkonzept nur autorisierten Benutzern den Zugriff auf diese sensiblen Daten. Die zugrunde liegende Datenbank verfügte ebenfalls über einen adäquaten Zugriffsschutz.

Leider wurde die tägliche Sicherung des gesamten Datenbestandes als „Dump" auf ein Band geschrieben. Mit einfachen Kommandos des Betriebssystems ließen sich sämtliche Daten von diesem Band auch außerhalb des Systems wieder lesen. Alle Sicherheitsmechanismen des Systems, der Persistenzschicht und der Datenbank wurden dadurch wirkungslos.

Das Problem entdeckte übrigens ein externer Review; es wurde durch eine Anpassung des Sicherungskonzeptes wirksam gelöst. ■

Wenn Sie mit sicherheitskritischen Daten arbeiten, prüfen Sie sämtliche Kopien dieser Daten auf ihre Sicherheit. Eine hochgradig gesicherte Persistenzschicht genügt nicht, wenn gleichzeitig Daten im (ungesicherten) Dateisystem abgelegt werden. ■

Transaktionen

Alles-oder-Nichts

Software-Systeme lösen oftmals mehrere logisch zusammengehörige (Datenbank-) Operationen aus. Diese müssen oft dem „Alles-oder-Nichts"-Prinzip genügen: Entweder sämtliche Operationen sind erfolgreich, oder keine einzige darf ausgeführt werden. Solche Transaktionen dienen dazu, die Konsistenz und fachliche Korrektheit eines Datenbestands zu sichern. Persistenz und Transaktionen hängen von daher eng miteinander zusammen.

Protokollierung und Monitoring

Protokolle helfen bei Optimierung

Die Abbildung von Objekten auf Tabellen (OR-Mapping) können Sie durch entsprechende Protokollierung optimieren. Lassen Sie beispielsweise die SQL-Anweisungen der Persistenzschicht über einige Zeit protokollieren. Dann können Sie prüfen, ob sich etwaige Join-Operationen durch kontrollierte Einführung von Redundanzen (Stichwort: Denormalisierung) vermeiden lassen.

Caching

Caching optimiert Zugriffszeiten

Der Zugriff auf persistente Speichermedien ist hinsichtlich Zeit- und Ressourcenaufwand meistens mehrere Größenordnungen aufwendiger (d. h. teurer) als Zugriffe auf Objekte im Speicher. Durch Zwischenlagern oder Vorhalten von Objekten im Speicher können Sie Zugriffszeit sparen, allerdings auf Kosten eines signifikant höheren Verwaltungsaufwands. Kritische Fragestellungen beim Caching sind beispielsweise:

- Nach welcher Strategie nehmen Sie Daten in den Cache-Speicher auf und wie/wann wieder heraus?
- Halten Sie die Daten zum Lesezugriff im Cache, oder wollen Sie auch schreibende Zugriffe beschleunigen? Bei Systemabstürzen besteht dann das Risiko, dass Inkonsistenzen zwischen dem persistenten Speichermedium und dem (vor einem Absturz noch nicht gesicherten) Cache auftreten.
- Wie groß dimensionieren Sie den Cache für welche Art von Objekten?

Generieren der Persistenz mit MDA oder MDSD

MDSD kann Persistenz vereinfachen

MDA oder MDSD-Generatoren können Ihnen eine Menge Aufwand bei der Entwicklung Ihrer Persistenz abnehmen. Beispielsweise können Sie Entitäten in einem fachlichen Modell über Stereotypen („Entity“) kennzeichnen und den Generator und dessen Templates die notwendigen Codeartefakte (Klassen, SQL-Skripte, Konfigurationsdateien etc.) erzeugen lassen.

Schwieriger gestalten sich bei diesem Vorgehen *Releasewechsel* Ihrer Systeme, weil die Generatoren in der Regel die bestehenden Datenbestände nicht berücksichtigen, d. h. keine *Migrationsskripte* generieren. Diese müssen Sie auf Basis der bestehenden und neuen Tabellen- oder anderer Speicherstrukturen selbst entwickeln.

7.1.7 Praktische Vertiefung

Damit Sie sich selbst einen praktischen Überblick verfügbarer Werkzeuge zur Speicherung von Daten verschaffen können, empfehle ich Ihnen die folgenden Einstiegspunkte (mit jeweils steigender Komplexität beziehungsweise Leistungsfähigkeit). Experimentieren Sie selbst[14] – das macht Spaß und hat einen großen Lerneffekt:

- Speichern Sie beliebige Datenstrukturen und Objekte aus Python (oder Jython) mit Hilfe des Standardmoduls „pickle“. Dabei lernen Sie eine einfache, aber effektive Art der Serialisierung ins Dateisystem kennen. Für Fehler- und Ausnahmebehandlung müssen Sie selbst sorgen – dafür gibt’s praktisch keinen technischen Overhead.
- Experimentieren Sie mit einer embedded-Datenbank. Das könnte beispielsweise die (relationale) H2[15] sein, oder auch die objektorientierte db4o[16]. Hier können Sie schon mit wenigen Codezeilen Objekte in einer *echten* Datenbank speichern.

[14] bzw.: Lassen Sie sich von dem vielseitigen Buch „Seven Databases in Seven Weeks“ [Redmond+12] inspirieren.
[15] Siehe *www.h2database.com*, die kleine aber feine Open-Source-Datenbank.
[16] *www.db4o.com*

- Nun vertiefen Sie sich in die Java Persistence API (JPA). Darin hat (zumindest für die Java-Welt) die Speicherung von Daten eine immense Vereinfachung erfahren.
- Eine weitere interessante Möglichkeit der Speicherung und Suche von Daten bietet das GORM (grails object relational mapping) des Groovy-basierten Web-Framework Grails. Natürlich können Sie auch das Ruby-Pendant dazu, Rails, für diesen Versuch nutzen.
- Das CoreData Modul aus Apples Betriebssystem MacOS® und iOS® leistet bei der Verwaltung und Speicherung von Daten (zumindest auf diesen Plattformen) gute Dienste.[17]
- Als Nächstes steht eine der *ausgewachsenen* relationalen Datenbanksysteme an. Ob MySQL oder PostgreSQL, spielt dabei kaum eine Rolle – selbst von den bekannten kommerziellen Datenbanksystemen[18] bekommen Sie in der Regel kostenfreie Testversionen. Auf diese Systeme können Sie aus praktisch allen gängigen Programmiersprachen zugreifen, SQL lernen und die Probleme des objektrelationalen Mappings am eigenen Code erfahren. In jedem Fall sollten Sie zugehörige Management-Werkzeuge in Ihre Experimente einbeziehen.
- Nun erarbeiten Sie sich Kenntnisse (mindestens) in einer der modernen NoSQL-Datenbanken, vielleicht CouchDB, MongoDB oder RIAK[19]. Alle drei sind Open Source und ermöglichen hochgradig skalierbare Speicherstrukturen. Sie formulieren Ihre Abfragen mit Hilfe von map/reduce-Funktionen und können in typischer Web-Manier mit REST und http auf Ihre Daten zugreifen. Beide Vertreter geben Ihnen einen groben Einblick, wie wirklich große Systeme speichertechnisch *ticken* können. Zum tieferen Verständnis konsultieren Sie [Edlich+11], der viele Aspekte rund um NoSQL anschaulich und vertieft erklärt.
- Als Nächstes lernen Sie Speicherkonzepte der *Cloud* kennen. Dafür empfehle ich Ihnen den S3-Service von Amazon®. Eine erste Übersicht gibt *http://www.hongkiat.com/blog/amazon-s3-the-beginners-guide/*.
- Falls Sie noch mehr Interesse an Datenspeicherung haben, könnten Sie mit den Konzepten von DataWarehouse- oder Business-Intelligence-Systemen fortfahren. Diese OLAP (online analytical processing)-Systeme dienen primär der flexiblen Analyse und Auswertung von Daten.

7.1.8 Weiterführende Literatur

[Edlich+11] führen in die Welt der NoSQL-Datenbanken ein. Sie beschreiben deren konzeptionelle Grundlagen (wie map/reduce oder das CAP-Theorem) sowie einige typische Vertreter. Leicht zu lesen – jedoch entwickelt sich die NoSQL-Welt schneller weiter, als Verlage Papier bedrucken können. Aus diesem Grund betreibt der Hauptautor die lesenswerte NoSQL-Website (*http://nosql-database.org/*).

[Fowler02] erläutert diverse praxisgerechte Muster zur Persistenz, unterlegt mit kurzen Codebeispielen. Ein wichtiger Lese- und Nachschlagetipp für Persistenz, diese Patterns sind wirklich brauchbar!

---

[17] Sofern Sie sich auf ObjectiveC einlassen wollen und einen Mac zum Ausprobieren griffbereit haben. Letzteres empfehle ich Ihnen ohnehin.

[18] Beispielsweise die Express-Edition (XE) von Oracle®.

[19] Unter *http://vimeo.com/11240885* sehen Sie, wie Sie in weniger als 60 Minuten einen 3-Node RIAK-Cluster aufsetzen können – das hat einen sehr hohen Geek-Faktor.

[Keller97] beschreibt Strategien zur Abbildung von Objekten auf Tabellen und diskutiert ausführlich deren Vor- und Nachteile.

[Redmond+12] stellt sieben unterschiedliche Datenbanksysteme vor – und erklärt sehr praxis- und codenah deren Programmierung. Mir hat's außerordentlich gut gefallen.

Herzlichen Dank an Jürgen Krey, Lothar Piepmeyer, Peter Roßbach, Khalid Dermoumi, Michael Perlin, Robert Reimann, Till Schulte-Coerne und Stefan Tilkov für die hilfreichen Beiträge und Anregungen zur Persistenz. Dieses Thema hätte ein eigenes Buch verdient.

7.2 Geschäftsregeln

Geschäftsregeln: Domänenspezifische Kausalzusammenhänge oder bedingte Handlungsanweisungen.

7.2.1 Motivation

In der Praxis existieren viele Zusammenhänge der Form „Wenn eine Bedingung X erfüllt ist, dann handle auf folgende Art und Weise: ...".

Geschäftsregeln

Ich nenne Zusammenhänge dieser Art nachfolgend „Geschäftsregeln" oder auch Fachlogik – obwohl sie manchmal sehr technischer Art sein können.[20]

Statt einer förmlichen Definition möchte ich Ihnen einige Beispiele solcher Regeln aufzeigen:

- (Versicherungsbranche) Schadenfälle mit einem geschätzten Gesamtschaden von über 100 000 Euro werden nur in der Zentrale reguliert.
- (Versicherungsbranche) Gehört ein an einem Schadenfall direkt Beteiligter zur oberen Managementebene des Konzerns, so darf der Schaden nur von Sachbearbeitern der Vertraulichkeitsstufe V1 bearbeitet werden.
- (Finanzbehörde) Hat der Steuerpflichtige seine Termine zur Vorauszahlung mehr als zwei Mal erst nach Erinnerung wahrgenommen, so wird ihm 24 Monate lang kein Aufschub gewährt.

In vielen Branchen gelten Regeln dieser oder ähnlicher Art. Viele Aktionen oder Abläufe hängen von bestimmten Bedingungen oder Zuständen ab – und genau diese legen die unternehmens- oder domänenspezifische Fachlichkeit fest. Eine Entität „Kunde" allein mit ihren Attributen macht eben noch keine Fachlichkeit aus – erst die Kombination aus Daten und (durch Regeln gesteuertem) Verhalten macht die „Musik".

[20] Denken Sie etwa an Regeln für das Routing von Nachrichten innerhalb eines Systems – das manchmal von Auslastungs- oder Verfügbarkeitskriterien abhängt und nicht von fachlichen Bedingungen.

Oft stecken zu viele Geschäftsregeln im Code

Meine These zu Geschäftsregeln: (Zu) viele solcher Regeln stecken verborgen und fest zementiert in Form verschachtelter if-then-else-Konstrukte tief in schwer änderbarem Quellcode. Nur wenige fachliche Abläufe und Regeln können wir in Anwendungssystemen so einfach lesen und verstehen wie die oben genannten Beispiele. Nur mit großen Schwierigkeiten können wir in bestehenden Systemen neue Geschäftsregeln implementieren oder an aktuelle Gegebenheiten anpassen.

Leichte Anpassbarkeit stellt für viele Softwarearchitekturen jedoch ein wesentliches Entwurfsziel dar: Wir benötigen Entwurfs- oder Architekturmittel, um Geschäftsregeln einfach ändern zu können.

Verteilte Geschäftsregeln erschweren Änderungen

Woraus resultiert diese schwere Änderbarkeit? Wir haben doch Fachlichkeit von Technik getrennt, Entwurfsprinzipien angewendet, brav dokumentiert und sind methodisch vorgegangen. Unsere Architektur enthält einen *business layer* mit fachlich motivierten Domänenklassen.

Grundsätzlich gut – die Trennung von Fachlichkeit und Technik schafft Überblick und führt zu verständlichen Strukturen. Leider schleichen sich in die geschäftlichen Bausteine (Klassen, Methoden, Funktionen) jedoch immer wieder Teile ein, die Änderungen erschweren:

Unabhängig von der jeweiligen Implementierungstechnologie stehen fachliche Objekte in solchen Architekturen dabei in gegenseitigen Abhängigkeitsbeziehungen. Fachliche Methoden referenzieren jeweils andere (fachliche) Objekte, um (fachliche) Abläufe zu zementieren, pardon, zu implementieren. Noch einmal: Fachliche Abläufe, ein Bestandteil der Fachlogik, entstehen aus dem aktiven Zusammenspiel einzelner Fachobjekte.

Beim Entwurf solcher Zusammenarbeit müssen fachliche Abhängigkeiten vorab feststehen. Zwischen den beteiligten Klassen entstehen statische Abhängigkeiten. Änderungen an der Fachlogik bedürfen hierbei der Anpassung von Quellcode, immer verbunden mit seinem erneuten Build- und Deploy-Zyklus der gesamten Applikation.[21] Zu diesen Abhängigkeiten kommt das Navigationsproblem: Ein Fachobjekt muss möglicherweise vor der eigentlichen fachlichen Tätigkeit durch komplexe Objektgraphen zu anderen Objekten navigieren. So entstehen aus vermeintlich einfachen fachlichen Operationen häufig umfangreiche (und gar nicht mehr einfache) Codefragmente. Die Navigation zwischen Objekten hat nichts mit Fachlichkeit zu tun – sie basiert auf der Architektur und dem Design des jeweiligen Objekt- und Klassengeflechtes. Damit bewerte ich sie als reine Infrastruktur- oder Technikaufgabe, die Wartbarkeit und Verständlichkeit von Quellcode negativ beeinflusst.

Das Sequenzdiagramm in Bild 7.13 zeigt diese Situation. Die Fachlogik (d. h. die Geschäftsregeln) ist auf drei Objekte verteilt. Beachten Sie insbesondere die Abfragen „if x“ und „if y“ im Sequenzdiagramm: Sollten sich diese Bedingungen ändern, folgt daraus zwangsläufig eine Änderung von Quellcode sowie ein erneutes Deployment der beteiligten Komponenten.

[21] Daran ändert auch *Dependency Injection* kaum etwas: Sie können damit Abhängigkeiten reduzieren, aber praktisch nicht aus Ihren fachlichen Abläufen eliminieren.

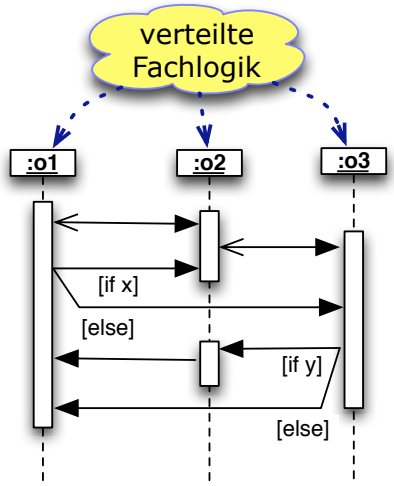


BILD 7.13
Fachobjekte mit verteilter Ablauflogik

Externalisierte Regeln zur Rettung

Alternativ können Sie einem Regelinterpreter (auch Regelmaschine, *rule-engine* genannt) zur Laufzeit die Ausführung der Fachlogik überlassen. Dazu beschreiben Sie Geschäftsregeln als eigenständige Einheiten (*first order citizens*) in einer Regelsprache.

Die grundsätzliche Vorgehensweise zeigt Bild 7.14. Ihr eigenes System übergibt die verfügbaren Geschäftsobjekte (o1, o2, o3) an die Regelmaschine. Die wiederum führt auf den Geschäftsobjekten die benötigten Methoden aus.

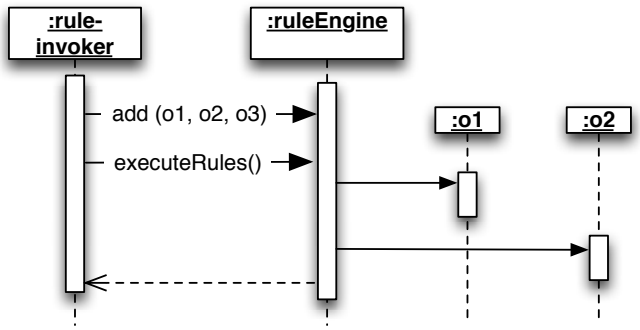


BILD 7.14
Regelmaschine steuert Ablauflogik

Regeln in diesem Kontext bestehen aus Wenn-dann-Sätzen. Im Wenn-Teil (auch condition part oder „Left-Hand-Side“ bzw. LHS genannt) stehen eine Menge von Bedingungen. Der Dann-Teil der Regeln (auch „Right-Hand-Side“ bzw. RHS) beschreibt, was zu tun ist, falls diese Bedingungen erfüllt sind:

```
Rule "MySimpleRule"
   when <conditions>   //Left-Hand-Side, LHS
   then <consequence>  //Right-Hand-Side, RHS
end
```

Durch Regeln können wir Fachlogik an zentraler Stelle (beispielsweise in eigenen Dateien) beschreiben. Sie drücken Fachlichkeit deklarativ aus, im Gegensatz zum imperativen Programmcode. Regeln beschreiben, was wir erreichen möchten. Das zugehörige Wie erledigt die Regelmaschine für uns. Dieses Prinzip steckt übrigens auch hinter SQL und regulären Ausdrücken, zwei erfolgreichen Beispielen deklarativer Problemlösung.

7.2.2 Funktionsweise von Regelmaschinen

Regelbasierte Ansätze dieser Art reichen bis in die achtziger Jahre des letzten Jahrhunderts zurück – in Informatik-Zeitrechnung also fast bis in die Steinzeit. Damals setzten hauptsächlich Wissenschaftler solche Regelsysteme ein, weil sie sich nur schwer in normale Systeme integrieren ließen. Im Rahmen der künstlichen Intelligenz begannen sie seinerzeit mit der Entwicklung der sogenannten Expertensysteme, meist auf Basis von Regelsystemen.

Die Regelmaschine arbeitet auf einem sogenannten Working-Memory, das die sogenannten Fakten sowie den aktuellen Bearbeitungszustand der Regeln enthält. Bei der Auswertung von Regeln geht die Regelmaschine in der Abfolge „Match“, „Select“, „Act“ vor:

- In der **Match**-Phase versucht die Regelmaschine, die Bedingungsteile sämtlicher Regeln mit den Fakten aus dem Working-Memory zu erfüllen. Sie testet dabei sämtliche möglichen Kombinationen von Fakten. Eine erfüllbare Regel mitsamt den dafür notwendigen Fakten wird in den Conflict-Set aufgenommen.
- Während der **Select**-Phase ermittelt die Regelmaschine für sämtliche erfüllbaren Regeln (also den gesamten Conflict-Set) eine Ausführungsreihenfolge, den Schedule. Bei den meisten Regelmaschinen erhöht beispielsweise die Komplexität der LHS ihre Priorität, teilweise spielen sogar Zeitstempel der beteiligten Fakten eine Rolle.
- In der **Act**-Phase schließlich führt die Regelmaschine die rechten Regelseiten der Regeln aus dem Conflict-Set in der vorher bestimmten Reihenfolge aus. Nun kann’s spannend werden: Die RHS einer Regel kann dazu führen, dass die Regelmaschine mit einer neuen Match-Phase wieder starten muss: RHS können nämlich die Fakten im Working-Memory ändern.

Diesen Zyklus implementieren moderne Regelmaschinen durch hochgradig optimierte Algorithmen (wie RETE[22], LEAPS), deren Details ich Ihnen hier erspare.

Bild 7.15 zeigt die Grundkonstrukte eines regelbasierten Systems: Ihr eigenes System verwendet, wie bisher auch, Fach- oder Geschäftsobjekte. Als sogenannte Fakten (*Facts*) stellt es dem Working-Memory der Regelmaschine Geschäftsobjekte zur Verfügung. Die Regelmaschine wertet eine Menge von Regeln aus, die zur Laufzeit aus Dateien oder Datenbanken gelesen werden.

Diesen Ablauf finden Sie schematisch in Bild 7.16 als Sequenzdiagramm dargestellt.[23] Ihr System initialisiert die Regelmaschine mit der Regeldatei „rules.txt“, erzeugt Fachobjekte und übergibt diese an die Regelmaschine zur Verwendung im Working-Memory. Anschließend wird die Regelauswertung (select-match-act-Zyklus) gestartet, die möglicherweise zur Änderung der Fachobjekte (hier: Objekt f) führt.

---

[22] Eine gute Erklärung finden Sie auf Wikipedia: *http://en.wikipedia.org/wiki/Rete_algorithm*

[23] In den Details unterscheiden sich die verfügbaren Regelmaschinen. Meine Darstellung orientiert sich an JBoss-Drools, einer in der Java-Welt häufig eingesetzten Open-Source-Regelmaschine. Siehe [Drools].

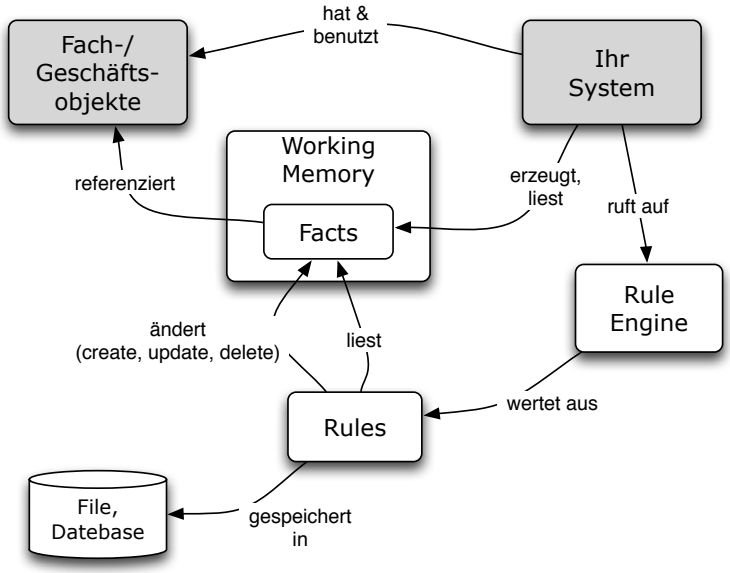


BILD 7.15 Grundkonstrukte eines regelbasierten Systems

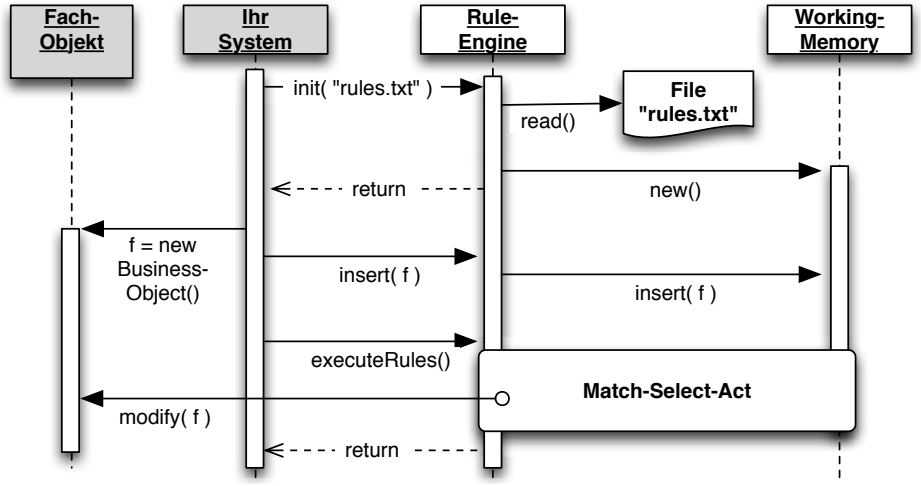


BILD 7.16 Schematischer Ablauf der Regelverarbeitung

Änderungen an fachlichen Regeln werden in diesem Szenario in der Regeldatei „rules.txt“ gepflegt.

Bei dieser Variante der Zusammenarbeit haben die Fachobjekte untereinander deutlich weniger Abhängigkeiten. Das kann zu besserer Änderbarkeit führen. Obendrein lassen sich Regeln häufig in (fast) natürlicher Sprache formulieren:

```
When (Person.Age < 18)
Then Person.setCreditAllowed( false );
```

Falls Sie sich jetzt fragen, ob so etwas in der Praxis wirklich funktioniert, habe ich einige Beispiele für (erfolgreiche) Anwendungen von Regelsystemen parat, was so viel bedeutet wie „Ja, es funktioniert an vielen Stellen“:

- Content-based Routing von Nachrichten in Service-orientierten Architekturen
- Tarif- oder Preisberechnung in Versicherungen
- Steuerung von Call-Center-Abläufen
- Steuerung der Datensatzverarbeitung von Batch-Prozessen

7.2.3 Kriterien pro & kontra Regelmaschinen

Ich möchte Ihnen einige Kriterien vorstellen, die für oder gegen den Einsatz von Regelsystemen sprechen:

- Müssen Sie innerhalb Ihrer Anwendungen Entscheidungen auf der Basis vieler bedingter Verzweigungen (z. B. verschachtelte if-then-else-Konstrukte) treffen? Dann passt eine Regelmaschine sicherlich gut zu Ihrem Problem. Können Sie Ihre Entscheidungen jedoch durch Datenbankabfragen oder rein algorithmische Berechnungen treffen, verwenden Sie besser keine.
- Wenn sich Ihre Geschäftsregeln nur selten ändern, können Sie statt Regeln ebenso konventionellen Programmcode schreiben und sich das Risiko und die Aufwände von Regelmaschinen ersparen.
- Versuchen Sie mal, einige Ihrer Geschäftsregeln in „Wenn-dann“-Form aufzuschreiben. Lassen Sie sich ruhig von einem Nicht-IT’ler dabei unterstützen. Wenn Ihnen nach einer halben Stunde immer noch keine fachlich sinnvollen Regeln eingefallen sind, spricht das gegen eine Regelmaschine. Lassen Sie sich aber vorher mal von Beispielregeln einer Regelmaschine inspirieren. Manchmal braucht das Umdenken von klassischen Programmiersprachen zu Regelsprachen einige Zeit.
- Wenn Ihre Entscheidungsprozesse einfach sind, d. h. Sie immer nur ein oder zwei if-Statements zur endgültigen Entscheidung benötigen, ist eine Regelmaschine wahrscheinlich nicht angemessen.
- Falls es bei Ihnen auf Millisekunden ankommt, weil Sie höchste Anforderungen an Performance stellen, vergessen Sie Regelmaschinen (obwohl die in den letzten Jahren wirklich flott geworden sind). Gleiches gilt für Echtzeitanforderungen (also hard-real-time constraints). Zumindest sind mir keine Regelmaschinen bekannt, die Laufzeiten im Echtzeitsinne garantieren können.
- Haben Sie Druck im Projekt? Stehen Sie kurz vor Fertigstellung? Neue Technologien stellen ein zusätzliches Risiko dar.
- Bedenken Sie, dass Einarbeitung in die Methode und Technik sowie Erstellung der Regeln hohen Aufwand benötigen. Falls Sie den Return-on-Investment für eine Regelmaschine in weniger als 12 Monaten erreichen müssen, wird’s riskant.
- Wenn Sie eine System- oder Softwarearchitektur entwerfen, bei der Fachlogik klar von der Technik getrennt sein soll, dann können Regelsysteme Sie dabei unterstützen.

- Regelmaschinen sind serverseitige Komponenten. Falls Sie Teile der Geschäftsregeln auf Clients (Browser, Smartphone, grafische Oberfläche) implementieren müssen: Erstens Vorsicht vor der entstehenden Redundanz, zweitens sind Regelmaschinen hierfür nicht gut geeignet.
- Wenn Sie architektonisch sauber arbeiten können, ihr System beispielsweise genau gemäß der Ports-und-Adapter-Architektur (= Clean-Architecture, siehe Kapitel 6.2.3.3) implementieren – können Sie auf eine Regelmaschine verzichten.

7.2.4 Mögliche Probleme

Wechsel des Programmier-Paradigmas

Vielen Entwicklern imperativer Programmiersprachen (Java, C++, C#, VB und Co.) fällt es sehr schwer, auf das deklarative Paradigma regelbasierter Sprachen umzuschwenken. Sie sollten Regelsprachen niemals dazu missbrauchen, „normale" imperative Programme zu schreiben.

Dieser Unterschied kann zu Widerständen seitens der Entwickler führen, zu Problemen bei der Fehlersuche oder zu schlechten Regelwerken.

Sie sollten ausreichend Zeit für die Einführung eines neuen Programmierparadigmas einplanen und möglichst mit kleineren Regelwerken beginnen. Hat Ihr Team erst mal „Lunte gerochen", lösen sich diese Probleme leicht auf.

Hohe Flexibilität birgt Risiken

Regeln in regelbasierten Systemen besitzen den gleichen Stellenwert wie sonstiger Quellcode. Sie müssen getestet werden und beeinflussen den Ablauf der Programme. Aus rein fachlicher Sicht bringen Regeln jedoch eine Menge zusätzliche Flexibilität. Regeln können Sie ohne die Umstände starrer Releasezyklen ändern. Diese Flexibilität hat ihren Preis in Form von höherem Risiko: Die einfache Änderbarkeit birgt das Risiko, dass Änderungen *einfach so* geschehen: Mit dem Texteditor im Produktivsystem flugs einige Regeln geändert – und schon ist es um die Stabilität des Systems geschehen (weil nichts mehr läuft).

Regeln wollen verwaltet werden

Geschäftsregeln einer Regelmaschine sollten Sie genauso behandeln wie anderen Quellcode: Modularisiert beschreiben, in Versionsverwaltung halten und bei allen Änderungen automatisiert testen.

Falls Sie sehr viele Regeln beschreiben und pflegen müssen, benötigen Sie Unterstützung für Modularisierung und Refactoring der Regeln.

Falls Ihre Regeln zur Laufzeit (!) geändert werden können, wird Versionsmanagement schwierig bis unmöglich. Bedenken Sie: Wenn Anwender oder Administratoren die fachlichen Regeln ändern dürfen, können diese Stakeholder auf diese Weise gravierende Fehler ihres Systems auslösen.

7.2.5 Weiterführende Literatur

Das Open-Source-Regelsystem JBoss-Drools kann mit einer ausgezeichneten Dokumentation aufwarten (siehe *http://jboss.org/drools*) – auch zu grundlegenden Regel-Themen. Seine Regelsprache können Sie sogar anwendungsspezifisch anpassen. Für Java-Projekte empfehle ich dieses System als Einstieg.

Ansonsten tummeln sich auf dem Markt sehr viele kommerzielle Regelsysteme, teilweise mit umfangreicher Unterstützung für Entwicklung und Betrieb (sogenannte *business rules management systems*).

7.3 Integration

Integration: Einbindung bestehender Systeme (in einen neuen Kontext). Auch bekannt als (Legacy) Wrapper, Gateway, Enterprise Application Integration (EAI).

7.3.1 Motivation

Integration von Legacy-Systemen

Legacy

Viele Software-Systeme entstehen in einem Umfeld bestehender Anwendungen und Daten. Insbesondere kommerzielle Informationssysteme werden in der Regel in einem solchen Kontext von Legacy[24]-Systemen entwickelt. Der Entwurf von Software in diesem Umfeld hat dann häufig einen der folgenden Schwerpunkte:[25]

- Erweiterung oder Ergänzung bestehender Software (zur Anpassung existierender Geschäftsprozesse).
- Ablösung von Altsystemen (weil diese nicht mehr wirtschaftlich wartbar oder anpassbar sind).
- Entwicklung neuer Software mit Schnittstellen zu Altsystemen (Eintritt in neue Geschäftsfelder mit Nutzung von Teilen der existierenden Software).
- Bestehende Daten müssen beim Entwurf neuer Systeme berücksichtigt werden.
- Notwendige Kooperation von Altsystemen durch Reorganisation oder Fusion von Unternehmen. In diesem Fall müssen unterschiedliche Systeme kooperieren, die ähnlichen Geschäftsprozessen dienen.

[24] Legacy (englisch): Altlast, Hinterlassenschaft. Hier in der Bedeutung von „Altsystem“ zu verstehen.

[25] Achten Sie auch auf Betrieb und Wartung der neuen Systeme. Sie können diese Aufgaben als Architekt zwar zunächst an das Projektmanagement delegieren, aber die technischen Aspekte werden garantiert wieder bei Ihnen landen!

Integration kommerzieller Komponenten

COTS

Das Problem der Integration tritt auch beim Entwurf von Systemen auf, die in hohem Maße auf kommerzielle Komponenten oder Frameworks zurückgreifen. Diese „Fertigprodukte“ (*Commercial-Off-The-Shelf*-Komponenten, COTS) können aus unterschiedlichen Bereichen stammen:

- Technische Komponenten, beispielsweise Bausteine grafischer Oberflächen, Persistenz-Frameworks oder Komponenten zur Datenverschlüsselung;
- Fachliche Komponenten, beispielsweise Rechnungswesen, Lager- oder Personalwirtschaft.

Frontend-Integration

Integration kann sich natürlich auch auf rein visuelle Darstellung innerhalb einer (grafischen) Oberfläche beziehen:

- Sie integrieren über Links/Hypermedia die Ausgabe anderer Programme. Beispiele hierfür sind Portalserver oder die sogenannten Mashups (= Kombination bestehender Inhalte). Typisches Beispiel: Integration von Werbung innerhalb von Webseiten.
- Sie integrieren Teile der Benutzerschnittstelle anderer Programme. Was dann für Endbenutzer wie ein einziges Programm aussieht, ist in Wirklichkeit eine Sammlung (Frontend-integrierter) Einzelteile. In Web-Technologien ist diese (grafische) Integration verbreitet, etwa durch Portalserver.

Integration bedeutet Wiederverwendung

Die Einbindung bestehender Systeme oder Komponenten bedeutet Wiederverwendung. Wiederverwendung kann in folgenden Aspekten helfen:

- Kosten und Zeit sparen: Wiederverwendete (d. h. zu integrierende) Komponenten müssen nicht neu entworfen, entwickelt und getestet werden.
- Risiko mindern: Neuentwicklung von Komponenten ist grundsätzlich mit Risiken behaftet. Durch den Einsatz bereits bestehender Software können Sie das gesamte Projektrisiko reduzieren.

Schnittstellen und Verantwortlichkeiten sind wichtig

Für die Legacy-Integration, besteht eine wesentliche Herausforderung in der Zuordnung klarer Verantwortlichkeiten und dem Entwurf effektiver Schnittstellen. Bei der Integration von COTS-Komponenten hingegen muss man die darin vorhandenen Verantwortlichkeiten und Schnittstellen verstehen und angemessen einsetzen (weil es in COTS-Komponenten meist nichts mehr zu entwerfen gibt).

Minimal invasiv

Dabei ist es ein wichtiges Entwurfsziel, in bestehende Systeme möglichst *minimal invasiv*[26] einzugreifen. Die Integration sollte aus Gründen der Risiko- und Kostenminimierung möglichst ohne Veränderung oder Anpassung der bestehenden Systeme vorgenommen werden.

[26] Der Begriff stammt ursprünglich aus der Medizin.

Beachten Sie bei der Integration kommerzieller Frameworks, dass diese häufig signifikante Teile der Softwarearchitektur fest vorgeben. Das kann in den Fällen hinderlich sein, in denen die Kontrolle des Gesamtsystems nur über das Framework und nicht mehr unter der Verantwortung des „eigentlichen" Systems erfolgen kann (*Don't call us, we call you!*).*

* Dies ist in der Framework-Gemeinde als „Hollywood-Prinzip" bekannt.

7.3.2 Typische Probleme

Folgende Probleme könnten Ihnen beim Entwurf und bei der Implementierung integrierter Systeme das Leben erschweren:

- Monolithisch implementierte Legacy-Systeme: Es gibt keine (erkennbare) Trennung zwischen Applikationslogik, Datenhaltung und Benutzeroberfläche. Dies tritt oftmals bei sehr alten Mainframe-Batch-Programmen auf.
- Technisch optimierte Legacy-Systeme: In längst vergangenen Zeiten wurden Software-Systeme oftmals unter Speicherplatz- oder Performance-Gesichtspunkten optimiert. Solche Optimierungen gingen (und gehen) meist zu Lasten der Wartbarkeit[27] oder Integrierbarkeit.
- Der Quellcode der Systeme liegt nicht mehr vor und ist nicht dokumentiert. Dieses Phänomen geht, streng nach Murphys Regel, häufig mit den beiden vorigen Fällen einher.
- Gerade auf Mainframe-Systemen enthalten die Datenzugriffsprogramme häufig „Plausibilitätsprüfungen"[28]. Diese Prüfungen sind eine Form impliziter Fachlogik. Oft entstehen durch diese Plausibilitäten starke Abhängigkeiten von Komponenten untereinander, die eine Integration der betroffenen Legacy-Systeme erschweren.
- Die Fehler- und Ausnahmebehandlung der Legacy-Systeme basiert häufig auf numerischen Rückgabewerten. Teilweise werden technische und fachliche Fehlersituationen gemeinsam behandelt, was eine aussagekräftige Fehlerbehandlung erschwert.
- Technische Aspekte der Kommunikation zwischen dem neuen System und den Altsystemen, etwa unterschiedliche Protokolle, aufwendige Datenkonvertierung, fehlerträchtiger Datentransport, Fehlerbehandlung oder programmiersprachabhängige Aspekte. Siehe Abschnitt 7.5 (Kommunikation).
- Durch die Integration bestehender Systeme entstehen oftmals (und teilweise ungewollt!) verteilte Systeme. Wenn etwa das neue System als Client/Server-System auf neuen Hardware- und Betriebssystemen realisiert wird und auf bestehende Mainframe-Anwendungen zugreift, so ist das Gesamtsystem verteilt! Bitte beachten Sie daher den Abschnitt 7.4 (Verteilung).

[27] Können Sie sich noch an die Jahr-2000-Probleme erinnern? Die wurden wesentlich durch technische Optimierungen (2-stellige Jahreszahl) ausgelöst.

[28] Im typischen Mainframe-Jargon oft „Plausis" genannt.

- Performance: Wenn die Performance bereits existierender Systeme schlecht ist, stellt ihre Integration in ein neues System ein technisches Projektrisiko dar.[29] Siehe dazu auch Abschnitt 7.4 (Verteilung).
- Sicherheit der integrierten Systeme. Insbesondere Mainframe-basierte Legacy-Anwendungen arbeiten mit völlig anderen Sicherheitsmechanismen als moderne Client/Server-Systeme. Hierdurch entstehen Risiken.
- Software-Entwickler haben manchmal die negative Eigenschaft, eigenen Code gegenüber wiederverwendetem zu bevorzugen (und damit auf mögliche Vorteile integrierter Systeme zu verzichten).
- Durch unterschiedlich geprägte Entwicklungsteams (beispielsweise Cobol-Mainframe- und Java-Client-Entwickler) kann es zu gravierenden Kommunikationsproblemen kommen, weil den Teams eine gemeinsame „Sprache“ fehlt.

7.3.3 Lösungskonzepte

Integration kann auf ganz verschiedenen Wegen erfolgen (nach [Hohpe+03]):

- Dateitransfer: Eine Applikation schreibt in eine Datei, die eine andere Applikation zu einem späteren Zeitpunkt lesen kann.
- Gemeinsame Datenbank (*Shared Database*): Mehrere Systeme verwenden eine gemeinsame Datenbank.
- Funktions- oder Methodenaufrufe (Synchrone Applikationsintegration, *Remote Procedure Call*, RPC): Ein System stellt nach außen aufrufbare Funktionen oder Methoden bereit, die über sogenannte Remote Procedure Calls synchron aufgerufen werden können.
- Asynchrone Kopplung (Messaging): Ein System schickt eine Nachricht an einen gemeinsamen oder öffentlichen Nachrichtenkanal. Andere Anwendungen können diese Nachricht von diesem Kanal zu einem späteren Zeitpunkt lesen.

Dateitransfer

Dateitransfer ist die wohl einfachste Art der Integration, unabhängig von Architekturen, Programmiersprachen oder Betriebssystemen der beteiligten Systeme. Beide Systeme müssen sich auf ein gemeinsames Dateiformat, Dateipfad und -namen sowie organisatorische Details (wann und wohin wird die Datei geschrieben, wann wird sie von wem wieder gelöscht?) einigen.

Dem Vorteil der Einfachheit und universellen Verfügbarkeit von Dateien stehen einige Nachteile gegenüber:

- Oftmals hoher Aufwand zur Konvertierung des gemeinsamen Dateiformats in das jeweils benötigte Format. XML-basierte Formate bieten sich heute als Austauschformat an, können allerdings von vielen Mainframe-Anwendungen nicht problemlos geschrieben oder gelesen werden.

[29] Oftmals können bestehende Systeme nicht beeinflusst oder optimiert werden, weil etwa der Quellcode nicht vorliegt, die Änderungen mit hohem Risiko verbunden wären, die eingesetzten Rechner voll ausgelastet sind oder Ähnliches.

- Geringe Performance von Dateioperationen.
- Sicherheitsmechanismen gelten lediglich für die gesamte Datei, es gibt keine feingranulare Zugriffskontrolle auf einzelne Teile. Das Zielsystem kann die Authentizität der Daten nicht ohne Weiteres sicherstellen.
- Das Quellsystem hat keine Information darüber, ob, wann und mit welchem Ergebnis das Zielsystem die Datei verarbeitet hat.

Gemeinsame Datenbank

Alle beteiligten Systeme müssen sich auf eine gemeinsame Datenbank mit einem einheitlichen Schema einigen. Hinsichtlich Performance, Zeitnähe von Updates, Konsistenz und Sicherheit ist diese Lösung deutlich leistungsfähiger als der Dateitransfer. Durch leistungsfähige Datenbanksysteme ist die Nutzung dieses Konzeptes aus unterschiedlichen Programmiersprachen und Betriebssystemen (fast) problemlos möglich. Doch auch dieser Ansatz kann einige Nachteile besitzen:

- In der Praxis ist es manchmal (überraschend) schwierig, ein für mehrere beteiligte Systeme geeignetes Datenbankschema zu entwerfen.
- Martin Fowler gibt in [Hohpe+03] zu bedenken, dass insbesondere die Integration von Standardprodukten in dieses Lösungskonzept schwierig ist, weil solche Produkte in der Regel mit unveränderlichen Datenbankschemata geliefert werden.
- Falls viele beteiligte Systeme parallel die gemeinsame Datenbank lesen und schreiben, kann diese zum Engpass hinsichtlich Performance werden. Die Ansätze der verteilten-und-replizierten Datenbanken birgt organisatorische und technische Risiken.
- Enge Kopplung zwischen den beteiligten Systemen: Änderungen an den grundlegenden Datenstrukturen bedingen Änderungen an allen beteiligten Anwendungen.

Remote Procedure Call

Falls Sie Systeme über Funktionen anstatt über deren Daten integrieren möchten, stehen Ihnen die *Remote Procedure Calls* als Lösungsansatz zur Verfügung. Konzeptionell bedeutet RPC den Aufruf einer Funktion auf einem (möglicherweise entfernt, das heißt in einem anderen Prozessraum) ablaufenden Programm. Bild 7.17 zeigt das Vorgehen schematisch. Dort finden Sie zwei zentrale Begriffe, Stubs und Skeletons: Diese Komponenten wickeln für die beteiligten Systeme die Kommunikation mit dem entfernten Partnersystem ab.

Jedes beteiligte System verantwortet seine Daten selbst, was eine gute Kapselung bedeutet.

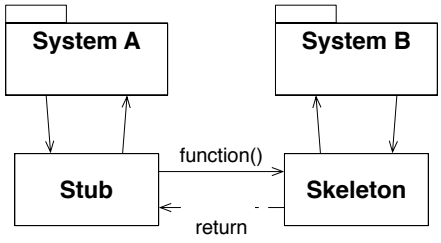


BILD 7.17
Schematische Darstellung von Remote Procedure Calls

Dieses Konzept wird durch Technologien wie CORBA (siehe unten), COM, .NET-Remoting, Java-RMI und SOAP umgesetzt. Es besticht durch hohe Flexibilität und direkte Wiederverwendbarkeit bestehender Systeme. RPCs sind die Grundlage von Client/Server-Systemen. In den folgenden Abschnitten finden Sie weitere Muster und Hinweise zur Integration über RPC.

Zu den möglichen Nachteilen von *Remote Procedure Calls* gehören folgende:

- Entfernte Funktionsaufrufe sind in Bezug auf Zeit- und Ressourcenbedarf um mehrere Größenordnungen teurer als lokale Aufrufe. In der Praxis führt das teilweise zu gravierenden Performanceproblemen.
- Es bedarf einer komplexen technischen Infrastruktur (Middleware), um entfernte Funktionsaufrufe zu ermöglichen. Technologien wie CORBA und andere sind zwar stabil, performant und in der Praxis bewährt, erhöhen jedoch die Komplexität von Systemen (und Projekten) ganz erheblich.
- Durch die komplexe Hardware- und Software-Infrastruktur von gekoppelten Systemen gefährden viele potenzielle Fehlerquellen den ordnungsgemäßen Gesamtablauf.
- Enge Kopplung zwischen den beteiligten Systemen:
  - Mögliche Blockierung: Wenn das aufgerufene System nicht oder nur sehr langsam antwortet, kann es zu Blockierung im Aufrufer kommen, weil der im Regelfall wartet, bis die Antwort vorliegt.
  - Eine Schnittstellenänderung im aufgerufenen System hat möglicherweise Konsequenzen im Aufrufer. Dadurch besteht das Risiko, dass Änderungen nicht-lokale Auswirkungen nach sich ziehen. In der schematischen Abbildung müssen Sie bei Änderungen am System B stets auch System A testen!

Messaging

Bei der nachrichtenbasierten Integration, kurz Messaging, senden sich die beteiligten Systeme Datenpakete, die in sogenannte Nachrichten verpackt werden. Eine von den Applikationen getrennte Kommunikationssoftware (*Message Oriented Middleware*, MOM) übermittelt die Nachrichten über Kanäle (Channels oder Queues). Kanäle speichern Nachrichten so lange, bis sie von den Empfängern verarbeitet wurden.

Nachrichten im Messaging-Konzept bestehen meist aus zwei Teilen:

- Im *Message-Header* befinden sich Meta-Informationen über die Nachricht, etwa der Name von Sender und Empfänger. Message-Header werden meist von der Messaging-Middleware genutzt.
- Im *Message-Body* befinden sich die eigentlichen Nutzdaten. Diese können vom Empfänger beliebig interpretiert werden, als Daten oder auch als Aufforderung, einen bestimmten Funktionsaufruf auszuführen.

Bild 7.18 zeigt den schematischen Verlauf dieser Kommunikation (Erläuterung in Anlehnung an [Hohpe+03]):

1. *create:* Der Sender erzeugt die Nachricht und füllt sie mit Daten. Wenn nötig, wird der Nachricht adressiert, d. h. mit einer Empfängeradresse versehen.
2. *send:* Der Sender fügt die Nachricht einem Kanal hinzu und übergibt sie damit der Kontrolle des Messaging-Systems.

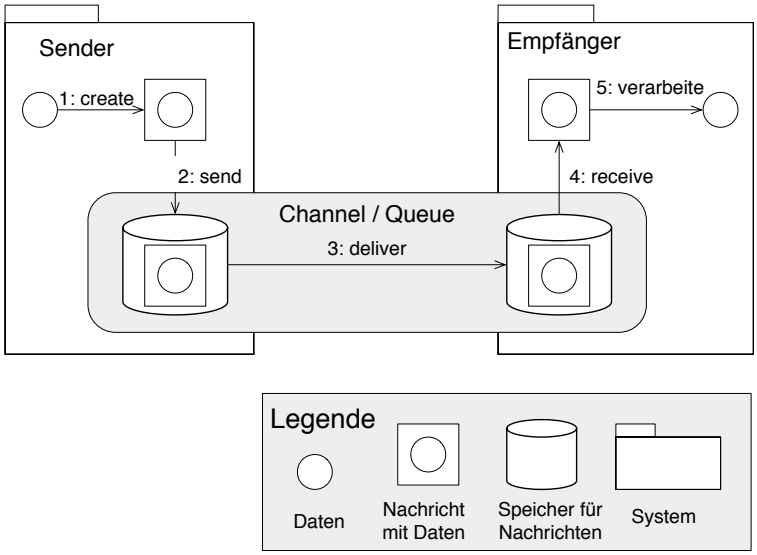


BILD 7.18 Schematische Darstellung von Messaging (nach [Hohpe+03])

3. *deliver:* Das Messaging-System leitet die Nachricht vom Computer des Senders an den des Empfängers weiter und stellt sie dort dem Empfänger zur Verfügung.
4. *receive:* Der Empfänger liest die Nachricht vom Kanal und übernimmt sie damit vom Messaging-System.
5. *verarbeite:* Der Empfänger extrahiert die Daten aus der Nachricht und verarbeitet sie.

In diesem Ablauf erkennen Sie zwei wichtige Konzepte:

- Senden-und-vergessen (*send-and-forget, fire-and-forget*): Dem Sender wird die Zustellung der Nachricht vom Messaging-System garantiert. Er kann die Nachricht nach dem Senden getrost vergessen. Dies ist der wesentliche Unterschied zu synchroner Kommunikation: Der Sender muss nicht auf die Verarbeitung der Nachricht warten!
- Speichern-und-weiterleiten (*store-and-forward*): Wenn der Sender die Nachricht dem Kanal hinzufügt, speichert das Messaging-System diese Nachricht auf dem Computer des Senders. Zu einem geeigneten Zeitpunkt wird sie dann auf den Computer des Empfängers weitergeleitet und dort ebenfalls gespeichert.

Bei Integration durch Messaging müssen sich die Anwendungen auf den Kanal sowie auf das Format der Nachricht einigen. Das Messaging-System übernimmt sämtliche technischen Details der Kommunikation. Es garantiert unverfälschte Übertragung sowie Zustellung der Nachricht. Kommerzielle MOM-Produkte bieten zusätzliche Funktionen, wie etwa Rückruf und Neuordnen von Nachrichten, Vergabe von Prioritäten, Versenden an mehrere Empfänger und weitere.

Die asynchrone Integration durch Messaging besitzt unter anderem folgende Vorteile:

- Kommunikation über Systemgrenzen hinweg, unabhängig von Programmiersprachen und Betriebssystemen (solange auf den beteiligten Systemen das Messaging-System verfügbar ist!).

- Zuverlässigkeit (*guaranteed delivery*) durch das zwischenzeitliche Speichern von Nachrichten.
- Vermeidet Blockierung beim Sender und Überlastung der Empfänger.
- Auch Anwendungen ohne permanente Netzverbindung können durch Messaging integriert werden: Sie übermitteln oder empfangen ihre Nachrichten, wenn sie gerade online sind – in der übrigen Zeit können sie unabhängig von Netzverbindungen arbeiten.

Nun, nach so vielen Vorteilen dürfen Sie mit Recht auch einige Nachteile erwarten:

- Komplexes Programmiermodell. Das konventionelle Modell von Funktions- oder Methodenaufrufen funktioniert für Messaging nicht mehr.
- Das Konzept garantiert keine spezifische Reihenfolge von Nachrichten.
- Falls geschäftliche Anforderungen synchrone oder zumindest kurzfristige Antworten erfordern, können Sie das Konzept nicht ohne Weiteres anwenden. Ihre Kunden wollen beispielsweise eine Preisauskunft sofort haben, und nicht nach einer beliebigen Zeitspanne.
- Abhängigkeit von Messaging-Produkten, neudeutsch *vendor lock-in*.

Viele große Informationssysteme, bei denen Zuverlässigkeit und Entkopplung wichtige Architekturziele sind, basieren auf Messaging. Hierzu gehören beispielsweise die zentralen Buchungssysteme von Banken, Abrechnungssysteme der Telekommunikation sowie große Buchungs- und E-Commerce-Systeme.

Ich werde hier nicht weiter auf die Aspekte asynchroner Integrationslösungen eingehen. Eine umfassende und praxisnahe Abhandlung des Themas finden Sie in [Hohpe+03].

7.3.4 Entwurfsmuster zur Integration

Wenn Sie als Architekt häufig mit Integration zu tun haben, sollten Sie den Abschnitt über Entwurfsmuster in Kapitel 6 gründlich lesen. Folgende Entwurfsmuster gehören bei der Integration zum wichtigen Entwurfswerkzeug:

- *Adapter:* Passt die Schnittstelle einer Komponente an eine von ihren Klienten erwartete Schnittstelle an. Lässt ansonsten inkompatible Komponenten zusammenarbeiten.
- *Brücke:* Entkoppelt eine Abstraktion von ihrer Implementierung.
- *Fassade:* Eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems.
- *Proxy:* Ein vorgelagertes Stellvertreterobjekt.
- *Vermittler (Mediator):* Definiert ein Objekt, welches das Zusammenspiel mehrerer anderer Objekte in sich kapselt.

Manchmal finden Sie auch die Begriffe *Wrapper* oder *Gateway* für Integrationsbausteine.

7.3.5 Konsequenzen und Risiken

Integration oder Migration?

Ablösung von Altsystemen

Migration

Keep the data – toss the code

Reengineering

In manchen Fällen dient die Integration in Wirklichkeit der Ablösung eines Altsystems. Das neue System basiert in diesem Fall auf einer neuen technischen Basis, geht in seiner Funktion aber nicht über das alte System hinaus. In diesem Falle sollten Sie prüfen, ob Sie anstelle der Integration eine (schrittweise) Migration vornehmen. Ein Leitsatz lautet: *Keep the data – toss the code*. Wenn Ihre Integrationsbemühungen also in Reengineering von Programmcode ausarten, sollten Sie ernsthaft darüber nachdenken, ob eine Ablösung durch ein neues System mit einmaliger Datenmigration nicht billiger ist, als den Code eines Altsystems zu restrukturieren. Migrationen kann man dann auch schrittweise vornehmen.

Bei einer schrittweisen Migration erhalten Sie kurzfristige Rückmeldungen über Erfolg, Akzeptanz oder Probleme. Dieses Vorgehen stellt eine spezielle Ausprägung des iterativen Vorgehens dar. Das Projektrisiko sinkt in diesem Fall erheblich. Weitere Informationen zur Migration finden Sie in [Keller2000], zu Reengineering in [Demeyer+02].

Ist Integration angemessen oder zu riskant?

Integration birgt organisatorische und technische Risiken. Sie müssen diese Risiken gegenüber den erwarteten Vorteilen der Wiederverwendung abwägen. In folgenden Fällen droht besonders hohes Risiko. Sie sollten dann mit Ihren Kunden oder Auftraggebern über Alternativen oder Abhilfen diskutieren.

- Ist der Geschäftswert der existierenden Systeme größer als diese Risiken? Wenn ja, stellt die Integration den geeigneten Weg dar.
- Im Fall besonders alter oder unsauber programmierter Legacy-Anwendungen kann es einfacher und risikoärmer sein, nur die bestehenden Daten zu erhalten und die funktionalen Teile des Systems „zu entsorgen“.
- Sie benötigen für ein Integrationsprojekt mindestens einen „Kenner“ des zu integrierenden Systems im Team, sowohl der fachlichen als auch der technischen Aspekte! Falls Ihnen diese Ressource oder dieses Wissen nicht zur Verfügung steht, ist ein Scheitern der Integration wahrscheinlich!
- Falls das bestehende System auf Techniken basiert, die im aktuellen Projekt nicht eingesetzt werden (Programmiersprache, Betriebssystem, Datenbank, Transaktionsmonitor oder Ähnliches), benötigen Sie auch zu diesen Themen Know-how.

So offensichtlich die beiden letzten Forderungen auch klingen, so häufig werden sie in der Praxis missachtet. Der Grund ist oftmals, dass die Kenner der alten Systeme mit deren Betrieb und Pflege bereits mehr als vollständig ausgelastet sind. Die für eine Integration so wichtigen Experten haben dann schlichtweg keine Zeit für andere Projekte.

- Gibt es in Ihrer Organisation Erfahrung mit dem Wrapping von Systemen? Sind entsprechende Werkzeuge vorhanden, oder müssen sie eingekauft oder entwickelt werden?
- Sind die vorhandenen Systeme stabil, oder befinden sie sich in der Entwicklung? Auch in diesem Fall kann das Risiko einer Integration deren Geschäftswert möglicherweise übersteigen. Eine Lösung besteht darin, gemeinsam mit Auftraggebern und Projektleitung ein iteratives Vorgehen zu definieren, bei dem die Schnittstellen während der einzelnen Iterationsphasen garantiert stabil bleiben.

Problemfall: Transaktionen

Als problematisch bei der Integration bestehender Systeme erweist sich manchmal die Implementierung systemübergreifender Transaktionen.

Falls das alte und das neue System ihre jeweiligen Transaktionen mit unterschiedlichen Transaktionsmonitoren realisieren, müssen Sie als Architekt deren Zusammenarbeit sicherstellen, etwa über einen funktionalen Prototypen.

Eine mögliche Vereinfachung besteht darin, echte (Datenbank-)Transaktionen jeweils auf ein System zu beschränken.

Datenschutz und Systemverfügbarkeit

Eine Kette ist so stark wie das schwächste Glied. Diese Regel gilt auch für integrierte Systeme. Sie werden das merken, wenn Sie als Architekt eines integrierten Systems eine bestimmte Systemeigenschaft garantieren müssen. Häufig übergreifende Anforderungen sind etwa:

- Verfügbarkeit des Gesamtsystems, Wiederanlaufzeit, erlaubte Wartungszeiten;
- Einhaltung von gesetzlichen Bestimmungen (etwa: Datenschutzgesetz);
- Einhaltung allgemeiner Service Level Agreements.

Solche Garantien fallen schwer, wenn Sie als Architekt die entsprechenden Eigenschaften der integrierten (Legacy-) Systeme nicht entsprechend beurteilen oder überprüfen können. Unser Ratschlag:

Prüfen Sie, ob und welche Anforderungen hinsichtlich Verfügbarkeit, Wiederanlaufzeit, maximaler Ausfallzeit und Datenschutz bestehen. Prüfen Sie die zu integrierenden Systeme hinsichtlich dieser Anforderungen.

7.3.6 Zusammenhang mit anderen Aspekten

Kommunikation und Verteilung

Kommunikation und Verteilung sind mit der Integration eng verwandt. Kommunikation stellt die technischen Grundlagen zur Integration bereit. Verteilung ist das Gegenstück zur Integration. Gerade die (möglichen) Probleme bei der Verteilung von Komponenten gelten auch für die Integration. Siehe Abschnitt 7.4 (Verteilung) und Abschnitt 7.5 (Kommunikation).

Sicherheit

Bei Legacy-Systemen auf Basis von Mainframes kommen häufig Sicherheitsmechanismen und entsprechende Produkte zum Einsatz, die auf anderen Plattformen nicht verfügbar sind. Beachten Sie beispielsweise folgenden Aspekt:

Die Benutzerverwaltung auf Mainframe-Systemen unterscheidet sich manchmal von der im Client/Server-Bereich. Wenn Sie unterschiedliche Mainframe-Systeme integrieren müssen, kann es an dieser Stelle zu Problemen kommen. Nehmen Sie diesen Aspekt als Einfluss- oder Risikofaktor in Ihre Planung auf!

Technische Benutzer

Es kann hilfreich sein, sogenannte „technische Benutzer“ einzuführen. Das sind Benutzerkennungen, denen keine reale Person entspricht und deren Kennwörter nicht ablaufen. Diese technischen Benutzer können nützlich sein, wenn Teile Ihrer Anwendungen auch ohne angemeldete Benutzer aktiv sein müssen.

Beispiel: Für das Call-Center einer Versicherung wurde eine dezentrale Anwendung entworfen, die eine Mainframe-basierte Adressverwaltung integrierte. Einige Basisdienste, beispielsweise die Prüfung von Postleitzahlen, sollten als Systemdienste unabhängig von einzelnen Benutzern betrieben werden. Der Mainframe beharrte jedoch darauf, die Liste der gültigen Postleitzahlen nur nach erfolgreicher Benutzerautorisierung „herauszugeben“.

Zur Abhilfe wurde ein technischer Benutzer eingeführt, der den Start sämtlicher Basisdienste ermöglichte. ■

Administration

Die Administration integrierter Systeme wirft eine Reihe von Problemen auf:

- Überwachung: Die Überwachung hinsichtlich Betriebsbereitschaft kann auf verteilten Systemen komplex sein. Es gibt entsprechende Werkzeuge zur Systemverwaltung, die hier unterstützen.
- Systemstart: Der Start („das Hochfahren“) eines verteilten Systems kann bestimmte Konstellationen der beteiligten Komponenten erfordern. Klären (und dokumentieren!) Sie exakt, welche Schritte zum Start des Gesamtsystems erforderlich sind (und in welchen Zuständen sich alle beteiligten Subsysteme befinden müssen).
- Einhaltung von Service Level Agreements: Wie bereits oben erwähnt, stellen „garantierte“ Eigenschaften des Gesamtsystems (etwa: Verfügbarkeit und Datenschutz) Probleme dar.

Für die Administration integrierter Systeme ergibt sich die zusätzliche Schwierigkeit, die Einhaltung allgemeiner Service Level Agreements *nachzuweisen*. Stellen Sie sich beispielsweise vor, mehrere beteiligte Subsysteme schreiben jeweils ihre eigenen (und proprietären) Protokolldateien (Logfiles).

7.3.7 Weiterführende Literatur

[Hohpe+03] enthält über 60 verschiedene Muster zu Entwurf und Implementierung (asynchroner) Integrationslösungen, sogenannter Messaging Solutions. Die Autoren führen sehr verständlich in die vielfältigen Aspekte von Integrationslösungen ein und behandeln sowohl Grundlagen als auch (sehr) fortgeschrittene Themen.

[Keller02] diskutiert Enterprise Application Integration (EAI) von einem sehr praktischen Standpunkt aus. Falls EAI in Ihrer Organisation ein Thema ist, sollten Sie dieses Buch einerseits selbst lesen und andererseits den verantwortlichen Managern schenken.

7.4 Verteilung

Verteilung: Entwurf von Software-Systemen, deren Bestandteile auf unterschiedlichen und eventuell physikalisch getrennten Rechnersystemen ablaufen.

7.4.1 Motivation

- Bei Client/Server-Systemen arbeiten Clients und Server in Form eines verteilten Systems zusammen.
- Verteilung von Komponenten kann nützlich sein, wenn diese Komponenten spezielle Eigenschaften der zugrunde liegenden entfernten Hard- oder Softwareplattform ausnutzen.
- Verteilte Systeme können durch Integration bestehender (Legacy-)Systeme in neue Software entstehen. Befinden sich das neue System und das bestehende System auf unterschiedlichen Hardware-Plattformen, so liegt eine Form der Verteilung vor.
- Der Wunsch nach hoher Ausfallsicherheit oder dynamische Lastverteilung auf mehrere getrennte Systeme kann die Verteilung von Software-Komponenten bedeuten.
- Die Verbreitung von Smartphones und Tablet-Computern als intelligente Clients erlaubt, verteilte Systeme auch in Massenmärkten zu nutzen.

7.4.2 Typische Probleme

Folgende typische Probleme treten bei verteilten Systemen auf:

- Entfernte Methoden- oder Funktionsaufrufe erfordern einen erheblich höheren Overhead als lokale Aufrufe.
- Komplexe Basismechanismen verteilter Objekte: Ein verteiltes System muss eine Reihe von Basismechanismen zur Bearbeitung „entfernter" Objekte bereitstellen. Dazu gehören:
  1. Objekterzeugung und Objektzerstörung.
  2. Lokalisieren (Suchen, Finden, Identifizieren) entfernter Objekte, etwa über einen Namensdienst.
  3. Aufruf entfernter Objekte (Methoden, Prozeduren).
  4. Behandlung von Objektreferenzen (*object handles, object references*).
  5. Verteilte Garbage-Collection.

Diese Mechanismen sollten im Idealfall auch über Programmiersprachengrenzen hinweg funktionieren.

- Datenkonvertierung und -transport: Ein verteiltes System muss die Integrität von Objekten über Komponentengrenzen hinweg sicherstellen. Das Marshalling verpackt Daten zum Transport über „den Draht", ein korrespondierendes Unmarshalling entpackt die übertragenen Daten wieder. Hierzu gehören Konvertierungen von Zeichensätzen (EBCDIC, ASCII, UniCode), Datentypen und Protokollen.
- Fehlerbehandlung über Systemgrenzen hinweg sowie die Differenzierung zwischen fachlichen Fehlern und (technischen) Kommunikationsfehlern.

7.4.3 Lösungskonzept

Techniken für Verteilung

Bekannte technische Ansätze zur Implementierung verteilter Systeme sind folgende:

- Kommunikation über http als Applikationsprotokoll mit REST (siehe Kapitel 6); heute (Stand 2013) gerade in Web-basierten Systemen das Mittel der Wahl.
- Web-Services auf Basis des *„Simple Object Access Protocol"* (SOAP)[30].
- Enterprise Java Beans (EJB), jedoch nur im Java-Umfeld.
- CICS und MQ-Series,[31] beide im kommerziellen Umfeld.
- Der offene Standard CORBA, entwickelt und standardisiert durch die Object Management Group.[32] CORBA-Objekte sind in hohem Maße unabhängig von Programmiersprache, Betriebssystem und Ablaufumgebung. Es gibt sowohl kommerzielle als auch frei verfügbare Implementierung von CORBA. Der Standard definiert viele wichtige Dienste wie Naming, Persistence, Transaction, Messaging und andere. In der Praxis kaum noch von Bedeutung.

[30] Details zu SOAP und Web-Services finden Sie unter *www.w3.org/TR/SOAP*.
[31] CICS und MQ-Series sind Produkte (und Warenzeichen) der Firma IBM.
[32] *www.omg.org*

- COM,[33] bekannt und verbreitet hauptsächlich in Windows™-Umgebungen. Eingeschränkte Verfügbarkeit auf anderen Betriebssystemen.

Ich rechne auch andere kommerzielle Middleware-Produkte zu den Techniken, die Verteilung ermöglichen. Hierzu gehören beispielsweise Transaktionsmonitore, die verteilte Verarbeitung ermöglichen sowie das weite Feld der EAI-Werkzeuge (*Enterprise Application Integration*).

Beschreibung von Schnittstellen

Ein wichtiges Hilfsmittel für alle Aspekte der Verteilung ist die exakte Beschreibung der Schnittstellen zwischen den Komponenten. SOAP, CORBA und COM enthalten dafür jeweils eigene Beschreibungssprachen (*interface definition languages*). In REST-Systemen können Sie die Syntax und Semantik der übermittelten Ressourcen exakt beschreiben.

Tipps zur Verteilung (Verteilen – aber richtig!)

- Können Sie Verteilung vermeiden, indem Sie sich auf eine einzelne Plattform beschränken? Das senkt das Entwicklungsrisiko und vereinfacht den Betrieb des Systems. Verteilung heißt immer höhere Komplexität und mehr. Fehlerquellen.
- Verteilen Sie so, dass Verarbeitungsprozesse „nahe“ bei den betroffenen Daten ablaufen. Das kann Netzverkehr reduzieren und die Performance verbessern.
- Entwerfen Sie möglichst stabile Schnittstellen. Aufgrund der Abhängigkeiten zwischen Client und Server ist dieser Aspekt wichtiger als bei nicht-verteilten Systemen.
- Reduzieren Sie die Menge der zu übertragenden Daten. Versuchen Sie auf die Übertragung komplexer Objektstrukturen zu verzichten. Führen Sie stattdessen möglichst schlanke „*transfer objects*“* ein. Solche Transfer-Objects repräsentieren einen Extrakt aus einem Objekt (oder einer Objektstruktur). Mit ihrer Hilfe reduzieren Sie die Anzahl entfernter Funktions- oder Methodenaufrufe.
- Fassen Sie zusammengehörige Funktions- oder Methodenaufrufe in Prozessobjekten zusammen. Das ist eine Ausprägung des Fassade-Entwurfsmusters (siehe Kapitel 6).
- Erzeugen Sie Objekte so spät wie möglich, nämlich erst dann, wenn sie wirklich benötigt werden. Lassen Sie verteilte Komponenten nicht „auf Vorrat“ arbeiten. Weichen Sie von dieser Regel nur bei Objekten ab, die das System auf jeden Fall benötigt (etwa: DB-Verbindungen oder ähnliche Hilfsobjekte; Objekte, deren Erzeugung viel Zeit in Anspruch nimmt).

[33] Die Bezeichnung COM subsumiert hier COM, COM+ und DCOM.

7.4.4 Konsequenzen und Risiken

Die Verteilung von Komponenten besitzt viele Vorteile hinsichtlich Flexibilität, Skalierbarkeit und Lastbalancierung. Jedoch stehen diesen Vorteilen auch eine Reihe potenzieller Risiken gegenüber:

- Der Betrieb verteilter Systeme umfasst mehrere potenzielle Fehlerquellen. Sämtliche Kommunikationskanäle (Netzwerke) müssen für den fehlerfreien Betrieb eines verteilten Systems zur Verfügung stehen. Auch kleine Probleme mit Netzwerken (Aus- oder Überlastung, kurze Aussetzer) können eine Fehlfunktion des Gesamtsystems bewirken.
- Entwicklung und Test verteilter Systeme sind schwieriger als bei zentralen Systemen. Stellen Sie sicher, dass Ihre Entwicklungsumgebung beispielsweise über Mechanismen zum Remote-Debugging verfügen. Fügen Sie Komponenten zur Fehler- und Ausnahmebehandlung ein, die auch Kommunikations- oder Verfügbarkeitsprobleme behandeln.
- Ungünstige Verteilung von Komponenten kann aufgrund extensiver Kommunikation der Komponenten untereinander zu Performance-Problemen führen.
- Verteilte Client/Server-Systeme benötigen manchmal einen komplexen Prozess der Software-Verteilung, -Installation und Versionsverwaltung.

7.4.5 Zusammenhang mit anderen Aspekten

Verteilung, Integration und Kommunikation hängen eng miteinander zusammen. Siehe daher auch die Abschnitte 7.3 und 7.5.

7.4.6 Weiterführende Literatur

Die OMG bietet auf ihrer Website, *www.omg.org*, detaillierte Informationen zu CORBA.

[Tilkov-11] ist ein guter Wegweiser durch Konzepte, Herausforderungen und Lösungen Web-basierter Architekturen mit REST und HTTP. Leicht lesbar und praxisorientiert.

7.5 Kommunikation

Kommunikation: Übertragung von Daten zwischen Systemkomponenten. Bezieht sich auf Kommunikation innerhalb eines Prozesses oder Adressraumes, zwischen unterschiedlichen Prozessen oder auch zwischen unterschiedlichen Rechnersystemen.

Ziel dieses Abschnitts ist es, einige für den Entwurf von Systemen wichtige Grundbegriffe der Kommunikation vorzustellen. Ich habe mögliche Lösungskonzepte in die Vorstellung der Begriffe und Konzepte eingearbeitet.

Die Aspekte der physikalischen Übertragung kommen hier nicht zur Sprache. Sie finden viele Grundlagen der Datenkommunikation beispielsweise in [Tanenbaum89].

7.5.1 Motivation

Beim Entwurf verteilter oder integrierter Systeme (siehe Abschnitte 7.3 Integration oder 7.4 Verteilung) müssen Komponenten miteinander kommunizieren. Eine solche Kommunikation kann im einfachen Fall aus dem Austausch von Daten oder Dateien bestehen (*file transfer*), es kann sich aber auch um den Aufruf von entfernten Funktionen, Methoden oder Programmen handeln (*remote procedure call*).

Eine physikalische Verbindung zweier Rechnersysteme allein genügt noch nicht, um diese Aufgaben zu erfüllen.

7.5.2 Entscheidungsalternativen

Es gibt eine ganze Reihe von Entscheidungen, die Sie hinsichtlich der Kommunikation zwischen Komponenten treffen können (und müssen):

- Findet Kommunikation synchron oder asynchron statt?
- In welchem Stil kommunizieren Komponenten miteinander? Mögliche Varianten sind: Remote Procedure Call, Publish/Subscribe oder Broadcast.
- Kommunizieren die beteiligten Komponenten direkt oder indirekt (über Gateways, Broker oder andere Middleware) miteinander?
- Welche Protokolle (etwa: TCP/IP, HTTP, IIOP, SOAP) kommen zum Einsatz?
- Wie werden unterschiedliche technische Gegebenheiten der beteiligten Kommunikationspartner (etwa: Zeichensatz, Zahlendarstellung, Wortgröße) behandelt?

7.5.3 Grundbegriffe der Kommunikation

Ich beschreibe Ihnen in diesem Abschnitt die architekturrelevanten Fragen und Aspekte der Kommunikation. Der Fokus liegt dabei auf den Anforderungen kommerzieller Systeme.

Synchron und asynchron

Synchron

Bei der synchronen Kommunikation wartet (blockiert) der Aufrufer, bis die aufgerufene Komponente ein Ergebnis (oder eine Statusinformation) zurückliefert. Synchrone Kommunikation ist der Normalfall bei der Softwareentwicklung. Alle gängigen Programmiersprachen nutzen fast ausschließlich synchrone Mechanismen zur Kommunikation von Programmkonstrukten untereinander.

Asynchron

Asynchrone Kommunikation hingegen basiert auf dem Prinzip „versenden und vergessen“. Eine Nachricht wird verschickt, aber der Aufrufer arbeitet sofort weiter, ohne auf ein Ergebnis oder eine Zustandsinformation zu warten (Einweg-Kommunikation[34]). Ein bekanntes Beispiel asynchroner Kommunikation ist die elektronische Post.

Asynchrone Kommunikation besitzt für Softwarearchitekturen hohe Bedeutung, weil über Schnittstellen zu externen Systemen häufig asynchron kommuniziert wird.

Erwägen Sie in folgenden Fällen den Einsatz asynchroner Kommunikation:

- Die Verbindung zwischen Client und Server ist nicht permanent. Falls Clients „offline“ sind, bleibt nur die asynchrone Kommunikation.
- Die beteiligten Systeme sind nur teilweise oder zu unterschiedlichen Zeiten verfügbar.
- Sie erwarten spezielle Anfragen oder eine hohe Transaktionslast, die Ihre Server aus Performance- oder Verfügbarkeitsgründen nicht synchron verarbeiten können. Beispiel: Datenbankoperationen, deren Laufzeit für Benutzer nicht akzeptabel ist (und deren Resultate sie nicht unbedingt sofort benötigen, etwa Änderungen einer großen Zahl von Datensätzen).
- Das System löst Aktionen aus, die manueller Nacharbeit bedürfen (etwa: Freigaben, Bestätigungen, Korrekturen oder Klassifikationen). Dieser Fall tritt häufig bei den Workflow-Systemen auf, siehe Abschnitt 7.9 (Workflow). Konkrete Beispiele aus unseren Projekten sind: Bestätigung von Bestellungen, Kreditfreigaben, Rechnungsprüfung oder Freigaben von Lieferungen.
- Sie wollen Nachrichten über einen Zeitraum sammeln und erst später verarbeiten. Dieses Vorgehen ist nützlich, wenn das System beispielsweise die Reihenfolge oder Priorität von Nachrichten erst nach dem eigentlichen Erzeugen der Nachricht festlegen kann.

Möglichkeiten asynchroner Kommunikation

Die Realisierung asynchroner Kommunikation erfordert in jedem Fall einen Speicher für die zu verarbeitenden Nachrichten. Dieser kann elektronisch sein (etwa Message Queues, Dateien oder E-Mail), aber auch aus Papier bestehen.

Message Queues

Asynchrone Kommunikation kann über Nachrichtenpuffer (*Message Queues, MQ*) ablaufen. Diese Variante ist die Domäne der sogenannten „Messaging-Systeme“, die als flexible Verwalter solcher Nachrichtenpuffer auftreten.

[34] Das zugehörige Konstrukt der CORBA Interface Definition Language heißt treffend „oneway“.

Sie können etwa die Übermittlung der Nachricht garantieren (*guaranteed delivery*) und Nachrichten persistent zwischenspeichern, um Systemausfälle zu überbrücken.

Beispiel: Großunternehmen benutzen für ihre Transaktionen (Buchungen, Bestellungen oder Ähnliches) oft Messaging-Systeme und asynchrone Kommunikation. Transaktionen werden über eine bestimmte Periode (etwa: einen Werktag) hinweg gesammelt und zu bestimmten Zeiten von der Warteschlange *en bloc* verarbeitet.*

* Das kann auch in Batch-Systemen geschehen.

„Richtige" Messaging-Systeme (kommerziell wie Open Source) haben noch andere Vorteile. Sie können Nachrichten beispielsweise (im Nachhinein) umsortieren, neu priorisieren, gruppieren oder auf unterschiedliche Zielsysteme verteilen (etwa zur Lastbalancierung). Hilfreich ist auch die Möglichkeit der „garantierten" Zustellung (= *robust* oder *guaranteed delivery*).

Andere Varianten asynchroner Kommunikation, etwa Dateien oder E-Mails, sind technisch einfach und weniger aufwendig, hinsichtlich Flexibilität allerdings den Messaging-Systemen unterlegen.

Beachten Sie, dass Kommunikation mit Medienbrüchen ebenfalls asynchron ist.

Stile der Kommunikation

Remote Procedure Call (RPC)

Remote Procedure Call (RPC) arbeitet genau wie ein lokaler Prozedur- oder Methodenaufruf, die aufgerufene Komponente kann sich jedoch auf einem entfernten Rechnersystem befinden. Client und Server sind in RPC-Szenarien häufig (über Festlegungen im Quellcode) eng gekoppelt. Grundsätzlich muss der RPC einige Schwierigkeiten bewältigen:

- Wie werden entfernte Prozeduren lokalisiert und gestartet? Was geschieht, wenn mehrere Clients gleichzeitig dieselbe Prozedur benötigen?
- Wie werden Parameter zwischen Client und Server ausgetauscht?
- Wie werden Fehlersituationen behandelt? Da Client und Server unabhängig voneinander fehlschlagen können, muss ein RPC-basiertes System alle möglichen Kombinationen von Fehlersituationen behandeln.

Publish-Subscribe

Publish – Subscribe: In diesem Szenario meldet ein Client durch eine Subscription an einer bestimmten Art von Nachrichten sein Interesse an. Ausführlich erläutert, finden Sie das Publish-Subscribe-Muster in [Buschmann96]. Client und Server sind in diesem Fall nur lose gekoppelt. Clients müssen von Servern nur die Schnittstelle für Subscribe-Operationen kennen. Der Mechanismus wird im Observer-Entwurfsmuster sowie im Model-View-Controller-Paradigma ausgenutzt.

Broadcast

Bei *Broadcast*-Kommunikation ist die Kopplung zwischen Client und Server noch lockerer. Server übermitteln Nachrichten ohne Angabe eines konkreten Empfängers oder „an alle". Verwenden Sie Broadcast in folgenden Fällen:

- Wenn das System prüfen soll, welche Services oder Clients gerade für Anfragen zur Verfügung stehen („Hey, wer kann mir gerade mal helfen?“).
- Wenn das System von verschiedenen Services „Angebote“ einholen soll.
- Wenn Sie dynamische und spontane Interaktion zwischen Komponenten benötigen.
- Wenn das System den Namen des Empfängers nicht kennt.

Direkte und indirekte Kommunikation

Bei der direkten Kommunikation kennt der Client seinen Server. Das bedeutet für den Software-Entwurf eine Abhängigkeit dieser beiden Komponenten voneinander. Schon Änderungen der Server-Adresse ziehen Änderungen an allen beteiligten Clients nach sich.

Bei der indirekten Kommunikation sind Clients von ihren Servern entkoppelt. Services werden über Vermittler[35] aufgerufen. Dadurch wird es möglich, Services flexibler zu betreiben. Eine Änderung am Service bedeutet höchstens eine Änderung beim Vermittler oder Gateway, die beteiligten Clients bleiben unverändert. Wichtige Elemente der indirekten Kommunikation sind Namensdienst (*naming*) sowie Broker, Message-Systeme oder Gateways.

Sie sollten indirekte Kommunikation einsetzen, wenn:

- Kommunikation über System- oder Rechnergrenzen notwendig wird;
- technische Rahmenbedingungen von Clients oder Servern sich häufig ändern (etwa: Standorte, Netzwerkparameter oder Service-Bezeichner);
- an das Gesamtsystem hohe Anforderungen hinsichtlich Flexibilität gestellt werden;
- Abhängigkeiten zwischen Client und Server nicht erwünscht sind.

In Kapitel 6 (Strukturentwurf) finden Sie weitere Ratschläge, Abhängigkeiten innerhalb Ihrer Entwürfe zu verringern oder zu vermeiden.

Protokolle

- TCP/IP (*Transmission Control Protocoll/Internet Protocoll*): Das Basisprotokoll im Internet. Es ermöglicht die Kommunikation zwischen jeweils zwei Teilnehmern (*peers*) über die sogenannten Sockets. Diese bestehen aus IP-Adresse und einer Portnummer.
- HTTP (*Hyper-Text Transmission Protocoll*), basiert auf TCP/IP. Es ist das Standardprotokoll bei der Kommunikation im Internet (häufig: zwischen Webserver und Browser). Es ist zustandslos, was bei Internet-basierten Client/Server-Systemen das Problem der Verwaltung des Anwendungszustands aufwirft. Siehe [Tilkov-11].
- SOAP: Ein Protokoll, das *Remote Procedure Calls* in XML beschreibt und (meistens via HTTP) überträgt.
- IIOP (*Internet Inter-ORB Protocoll*): Das Standard-Protokoll für die Kommunikation von CORBA-Objekten untereinander. IIOP ermöglicht auch die Kommunikation zwischen Object-Request-Brokern verschiedener Hersteller.

[35] Hierzu gehört jegliche Art von Middleware oder Broker.

Das OSI 7-Schichten Modell

Das Referenzmodell für Datenkommunikation ist das OSI-7-Schichten-Modell, dargestellt in Bild 7.19.

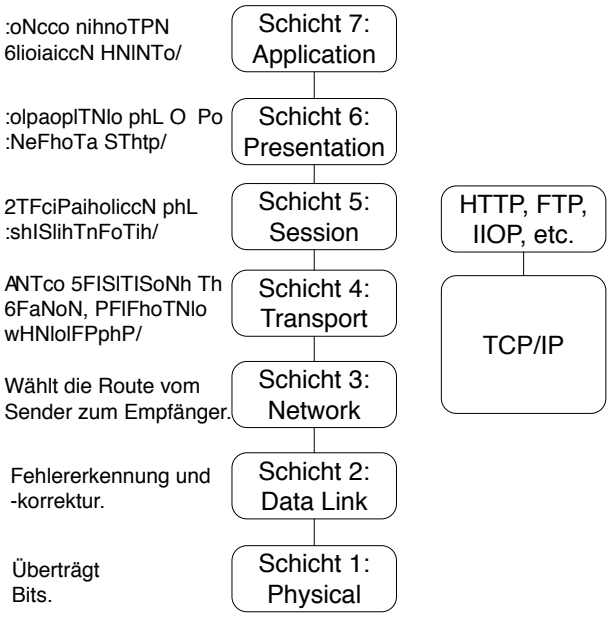


BILD 7.19
OSI-Referenzmodell und TCP/IP

Parallel dazu zeigt diese Abbildung das TCP/IP-Protokoll in Relation zum OSI-Modell. Es umfasst nahezu[36] die Leistungen der Netzwerk- und Transportschichten des OSI-Modells. Oberhalb von TCP/IP befinden sich die bekannten Protokolle HTTP, FTP und IIOP.

Weitere Informationen zum OSI-Modell, zu Protokollstacks und anderen Feinheiten der Kommunikation finden Sie in [Tanenbaum89].

7.5.4 Weiterführende Literatur

Die Grundlagen sowie ausführliche Details zum Thema Integration und Kommunikation finden Sie im Standardwerk [Hohpe+03] – besonders empfehlenswert!

[Tanenbaum89] ist (immer noch) ein Standardwerk für Details zur Datenkommunikation.

[Tilkov-11] erklärt Ihnen auf angenehme Art und Weise alles, was Sie über REST wissen sollten.

[36] Falls Sie es ganz genau wissen möchten: Der Netzwerk-Guru Andrew Tanenbaum erklärt es in [Tanenbaum89] sehr ausführlich! Für Softwarearchitekturen sind die Unterschiede nahezu bedeutungslos.

„Beam me up, Scotty“

Heureka, es ist vollbracht! Eines der letzten Geheimnisse unserer Zeit ist gelöst! Entdecker rund um den Globus, im Internet und möglicherweise sogar in einigen obskuren Ecken des Universums gierten seit langer Zeit danach, das Geheimnis eines unscheinbar kurzen Satzes zu ergründen: „Beam me up, Scotty!“

Benutzt in gefährlichen oder hoffnungslosen Situationen, in schweren Krisen oder in ängstlicher Agonie: Dieser Satz aktiviert den berühmten Raumschiff-Enterprise-Beam-Transporter. Die Star Trek Crew benutzte ihn lange Jahre, um Materie oder Leute von Punkt A nach Punkt B zu transportieren, wobei entweder A oder B das Raumschiff Enterprise selbst war.

Über die wahre Natur dieses Beam-Transporters wurde viel spekuliert, aber Tatsachen kamen nicht ans Licht. Die spektakulär simple Antwort wurde von einer Horde von Java-Entwicklern gefunden, die sich mitten in einem der heute so verbreiteten E-Commerce-Projekte befanden.

Der MAMI (*manager of middleware*) dieses Projektes, früher praktisch nur für ihre Sammlung von Science-Fiction-Videos bekannt, kam beim Morgenyoga der plötzliche Gedanke „Wir beamen es zum Kunden, statt horrende Transportkosten zu bezahlen.“ Sie kritzelte diese Idee auf die Rückseite eines gebrauchten Briefumschlags, bevor sie sich ihren wichtigen Administrivia (siehe [DeMarco98]) zuwandte.

Rein zufällig wurde dieser Briefumschlag von einer übereifrigen Java-Entwicklerin aufgefunden, die sich auf dem Schreibtisch von MAMI nach einer sinnvollen Aufgabe umsah. Zu jung, um Spock, Scotty und den Beam-Transporter zu kennen, fing sie kurzerhand an zu programmieren.

Sie nahm die Objekte, die ihr Boss auf dem zerfledderten Umschlag beschrieb, und serialisierte sie in einen XML-Wrapper. Dann, in einem dieser seltenen Anflüge schierer Genialität, schickte sie dies als HTTP POST Request über den Draht, wobei sie gleichzeitig das Problem der störenden Firewalls löste (obgleich dies eine stetige Einnahmequelle für ganze Legionen hochbezahlter IT-Berater ist, aber dieser Gedanke kam ihr in diesem Moment nicht!).

Fertig. Sie hatte es geschafft! Der Beam-Transporter war endlich Realität geworden, oder zumindest virtuelle Realität!

Zu schön, um wahr zu sein? SOAP, das (früher sogenannte)„*Simple Object Access Protocol*“, bewegt praktisch beliebige Objekte von Punkt A nach Punkt B – und darum dreht sich beim Beam-Transporter schließlich alles. SOAP mag nicht die perfekte Star-Trek „Beam-me-up“-Lösung sein, aber es ist zumindest ein Schritt in die richtige Richtung: Es erlaubt Prozedur-Aufrufe und Datenaustausch zwischen HTTP-basierten Clients und Servern, getreu dem Prinzip:

„Nimm die am weitesten verbreiteten und effektivsten Techniken (XML and HTTP), um ein Mindestmaß an gemeinsamem Verständnis in der heutigen heterogenen IT-Infrastruktur zu erreichen!“

(Nachdruck der JBiz-Kolumne aus dem JavaSpektrum März 2001, mit freundlicher Genehmigung des SIGS-Datacom Verlags.)

7.6 Grafische Oberflächen(GUI)

7.6.1 Motivation

Benutzer sind anspruchsvoll, benötigen ergonomische, reaktive und angemessene Bedienoberflächen (je nach Fachlichkeit, Art und Vorkenntnissen der Benutzer, technischen Gegebenheiten)

7.6.2 Einflussfaktoren und Entscheidungskriterien

Aus meiner Sicht prägen im Wesentlichen drei Aspekte die Umsetzung einer Benutzeroberfäche:

- Welche Arten von Benutzern verwenden das System, über welche Kenntnisse und Fähigkeiten verfügen sie?
- Welche Arten von Interaktionen soll das System ermöglichen oder unterstützen?
- Welche Plattformen und Technologien soll das System unterstützen?

Art der Benutzer

Welche Personen sollen das System benutzen: Können Sie technische und/oder fachliche Kenntnisse voraussetzen? Sollen neue und erfahrene Benutzer identisch behandelt werden oder unterschiedliche Benutzerschnittstellen bekommen?

Handelt es sich um eine geschlossene oder offene Benutzergruppe? Geschlossene Gruppen finden sich primär bei Inhouse-Anwendungen, offene Benutzergruppen beispielsweise bei Web-Anwendungen oder COTS[37]-Systemen.

Soll das System verschiedene Rollen von Benutzern unterstützen, mit jeweils unterschiedlichen Rechten und Interaktionsmöglichkeiten?

Gemeinsam ist menschlichen Benutzern eine Menge typischer Verhaltensmuster ([Tidwell-2011] nennt sie *behavioral patterns*). Ich möchte sie in Form einer Tabelle kurz vorstellen – sie stellen meiner Meinung nach eine gute Grundlage für Anforderungen an Benutzeroberflächen dar.

[37] Commercial Off-The-Shelf: Software, die als Standardprodukt ohne Individualisierung verkauft wird. Beispiel: Office-Produkte oder Computerspiele.

TABELLE 7.3 Was sich viele Benutzer von Benutzeroberflächen wünschen (behavioral patterns, nach [Tidwell-2011])

Verhalten	Erklärung
Sicheres Erforschen	„Lass mich das System erkunden, ohne dass ich in Schwierigkeiten gerate.“
Unmittelbare Belohnung	„Ich möchte jetzt sofort etwas schaffen, nicht irgendwann später.“
Gerade gut genug	„Das ist gut genug. Ich möchte nicht mehr Zeit investieren, um zu lernen, wie es noch besser klappen könnte.“
Mittendrin ändern	„Ich habe mir mittendrin etwas anderes überlegt.“
Aufgeschobene Entscheidung	„Ich möchte das jetzt nicht beantworten – lass mich einfach fertig werden.“
Schrittweise Verbesserung	„Lass mich das noch mal ändern, das sieht nicht gut aus. Und noch mal. Jetzt ist es besser.“
Gewöhnung	„Dieser Befehl funktioniert überall – warum nicht hier?“*
Räumliches Gedächtnis	„Gerade war der Button noch an dieser Stelle – wo ist er jetzt hin?“
Vorausblickendes Gedächtnis	„Ich lege das hier hin, um mich daran zu erinnern, es später fertig zu machen.“

* Ein prima Beispiel: In Emacs bewirkt die Tastenfolge „Ctrl-A → Ctrl-X → Ctrl-S“ eine Cursorbewegung und danach das Speichern der Datei. In Microsoft-Word bedeutet die gleiche Tastenfolge „gesamten Text markieren, gesamten Text ausschneiden (= löschen), die (jetzt leere) Datei speichern. Ooops ☺

Art der Interaktionen und GUI-Idiome

Welche Aktionen oder Interaktionen soll die Benutzerschnittstelle unterstützen:

- Reine Darstellung von Informationen, beispielsweise Status- oder Ergebnisanzeige – ohne direkte Interaktion mit Benutzern.
- Darstellung und Benutzung von Datenobjekten, ohne oder stark eingeschränkte Möglichkeiten zu deren Manipulation. Beispielsweise Media-Player, Auswahl und Anzeige von Aktienkursen, Produktauswahl in Web-Shops.
- Erzeugung und Manipulation von Datenobjekten, beispielsweise masken- oder formularbasierte Anwendungen, E-Mail- oder Kalenderprogramme, Texteditoren, Zeichenprogramme.
- Immersive Systeme, die ihren Benutzern durch optische, akustische (und teilweise auch taktile) Effekte „andere Welten“ suggerieren. Beispiele hierfür sind primär Spiele, etwa Autorennen, Flugsimulatoren und Actionspiele.[38]

Jenifer Tidwell stellt in ihrem großartigen Buch [Tidwell-2011] eine Vielzahl solcher Idiome vor – für alle nützlich, die sich mit GUI-Design beschäftigen.

[38] Wie etwa die berüchtigten Ego-Shooter und Kampfspiele (z. B. World of Warcraft), aber auch die Klassiker um Myst *http://mystonline.com*

Plattformen und verfügbare Technologien

Benutzeroberflächen müssen sich oftmals an den vorhandenen technischen Möglichkeiten der vorhandenen Zielplattform (Hardware und Software) orientieren. Sie würden persönlich eine ausgefeilte, grafisch anspruchsvolle Oberfläche implementieren, sind aber durch das Laufzeitsystem der Zielplattform an sehr einfache grafische Elemente gebunden ...

TABELLE 7.4 Kurzübersicht einiger Plattformen und UI-Technologien

Web-Clients (primär Browser)	▪ HTML, insbesondere HTML-5 mit CSS und JavaScript (bzw. die vielen Frameworks*, die HTML aus Ausgabeformat generieren) ▪ Proprietäre Formate, etwa Adobe-Flash
Rich-Clients (Arbeitsplatzrechner mit grafischen Displays, Tastatur und Maus/Trackpad o. Ä.)	▪ Microsoft Windows (WPF, WinForms) ▪ Java (JavaFX, Swing, Eclipse-RCP) ▪ Plattformneutral: QT**, GTK+ (Gimp-Toolkit***) oder die wxWidgets.
Mobile Clients (Smartphones, Tablets o. Ä.)	▪ Android ▪ iOS

* Rails (Ruby), Grails/Griffon (Groovy), Zend (Php), Django (Python), Ring (Clojure) u. v. a. m.
** Ein Klassiker der UI-Entwicklung: *http://qt-project.org/*
*** *http://www.gtk.org/*

Im Bereich der Frontend/GUI-Entwicklung haben insbesondere Web-Entwickler die Qual der Wahl: Eine Vielzahl von Frameworks bringen auch entscheidungsfreudige Zeitgenossen an die Grenzen ihrer Belastbarkeit.

7.6.3 GUI-relevante Architekturmuster

Sie sollten beim Entwurf von interaktiven Systemen sicherlich die klassischen Architekturmuster aus Kapitel 6.2.8 kennen – allen voran Model-View-Controller oder Presentation-Model.

Nach meiner praktischen Erfahrung geben in der Realität allerdings die konkreten technischen (UI-)Frameworks große Teile der diesbezüglichen Architektur- und Implementierungsentscheidungen bereits vor. Viele dieser UI-Frameworks (beispielsweise Windows-WPF oder WinForms, Java-Swing, JavaFX, Java-ServerPages, Ruby-on-Rails, Grails, iOS, Zend, Django oder Griffon) implementieren ihre eigene Interpretation von MVC oder einem ähnlichen Muster. Die Frameworks geben Ihnen vor, wie sie Ihren UI-Code zu strukturieren haben, wie UI-relevante Dinge im Quellcode heißen und wie diese sich verhalten.

Falls Sie also UI-intensive Anwendungen konstruieren, entwickeln oder begleiten müssen, sollten Sie die eingesetzten UI-Frameworks verstehen – das ist wahrscheinlich wichtiger als die konzeptionellen Grundlagen von MVC und Co ...

Im Zweifel implementieren Sie zwei oder drei kleine Prototypen auf Basis unterschiedlicher UI-Frameworks und entscheiden erst danach, welches Framework Sie auf welche Weise in Ihr System aufnehmen.

7.6.4 Struktur und Ergonomie von Benutzeroberflächen

Ich möchte erst gar nicht versuchen, Sie in der Kürze dieses Buches zu einem auch nur halbwegs ordentlichen UI-Designer zu machen – dafür ist das Thema zu umfangreich und zu komplex. Ich möchte Sie aber für einige Themen sensibilisieren, die Sie bei der ganzheitlichen Gestaltung von (grafischen) Oberflächen berücksichtigen sollten.

Struktur im Großen (UI-Metapher)

Von einer sehr hohen Warte aus betrachtet, hat fast jedes System eine Vielzahl unterschiedlicher Aufgaben zu erledigen. Auf einer so hohen Ebene entscheiden Sie als Architekt (oder UI-Designer) die UI-Metapher Ihres Systems. Dazu einige Vorschläge:

- Schreibtisch-Metapher: Benutzer bestimmen selbst, welche Objekte in ihrer unmittelbaren Umgebung (Schreibtisch) sichtbar bzw. zu bearbeiten sind.
- Assistenten-Metapher (= Fabrik- oder Fließband-Metapher): Das System leitet Benutzer durch eine feste Folge von Arbeitsschritten. Beispiel: Installations- oder Konfigurationsassistenten.
- Werkbank-Metapher (= Werkzeug-Material-Metapher): Fachliche Aufgabenstellungen und entsprechende Lösungsansätze zur Bearbeitung einzelner Informationsobjekte stehen an der Oberfläche bereit. Beispiel hierfür: Grafische Editoren oder Bildbearbeitungsprogramme.
- Formular-Metapher: Ein Formular (= Maske) enthält meist textuell dargestellte Daten, die teilweise auch erzeugt oder manipuliert werden können.

Struktur im Kleinen

Innerhalb der gesamten Benutzeroberfläche werden Sie meist kleinere Teile bereitstellen, die ihrerseits wieder spezialisierte Aufgaben lösen. Schon wieder möchte ich [Tidwell-2011] zitieren – die vier mögliche Ausprägungen dieser kleineren Teile (= Seiten) vorstellt:

1. Zeige eine einzelne Information oder ein Ding an.
2. Zeige eine Liste von Informationen oder Dingen an.
3. Stelle Werkzeuge zum Erzeugen einer Information oder eines Dinges an.
4. Stelle Mittel zur Bearbeitung einer Aufgabe bereit (= Manipulation von Daten).

Navigation innerhalb von Benutzeroberflächen

Haben Sie die Benutzeroberfläche gemäß ihrer Teilaufgaben modular gestaltet, so müssen die Benutzer durch diese einzelnen Teile navigieren. Dafür haben sich unterschiedliche Möglichkeiten etabliert:

- Menüs und Menübäume: Über Menü (meist am oberen Rand des Bildschirms) können Benutzer Aufgaben oder Daten auswählen und werden durch das System dann an die entsprechende Stelle im UI geleitet.
- Navigationsleisten oder -elemente: Hervorgehobene Elemente der Benutzeroberfläche erlauben die Navigation an bestimmte Stellen der Oberfläche.
- Tastaturkürzel (= Hot-Keys), die sofort die gewünschten Dinge darstellen bzw. Aktionen ausführen.

Benutzer sollten grundsätzlich sehen können, an welcher Stelle des Systems sie sich gerade befinden („Schritt 3 von 6“) oder in welchem Kontext sie arbeiten. Hierzu bieten sich etwa benannte Fenstertitel oder die sogenannten breadcrumbs (= Brotkrümelnavigation) an.

Arten von Benutzereingaben

Neben der bekannten Tastatur- und Mausbedienung grafischer Oberflächen sind in den letzten Jahren weitere Technologien zur Entgegennahme von Benutzereingaben entwickelt worden:

- Kombinierte und spezialisierte Eingabegeräte, beispielsweise Game-Controller, Joysticks, Grafiktablets.
- Berührungsempfindliche Displays ermöglichen Interaktionen durch Berührung und Gesten direkt auf dem Bildschirm (= touch-Display). Weit verbreitet haben sich diese Systeme primär über SmartPhones und Tablet-Computer.
- Dreidimensionale Sensoren, die Bewegungen im Raum erkennen können. Zuerst haben die Hersteller von Spielekonsolen[39] erste Versionen dieser Sensoren praxisreif gemacht, die Arm- und Handbewegungen im Bereich einiger Zentimeter differenzieren können. Mittlerweile differenzieren optische Sensoren[40] in einem Erkennungsbereich von ca. 50 cm Radius im dreidimensionalen Raum im Bereich von Millimetern. Damit werden feinmotorische Interaktionen möglich.

Natürlich sind analoge Bedienkonzepte (Dreh- und Schieberegler, Taster, Schalter) heute auch durch Softwaresysteme nutzbar. In hohem Maße werden beispielsweise komplexe Maschinen heute durch Kombinationen verschiedener Eingabekanäle bedient.

7.6.5 Bekannte Risiken und Probleme

Verbindung von UI und Daten (Data Binding)

Nehmen wir das Beispiel eines einfachen Informationssystems, das Daten aus einer Datenbank verarbeitet und an einer Benutzeroberfläche anzeigt. Betrachten Sie dazu Bild 7.20. Die Datenbank enthält ein Datum in Zustand (1), das von einem Server gelesen wird – Zustand (2). Zwei Clients stellen dieses Datum nun unabhängig voneinander dar (3) und (4).

Jeder dieser Teile (1) bis (4) könnte dieses Datum modifizieren – durch DB-Trigger, externe Zugriffe auf die Datenbank, fachliche Ereignisse auf Serverseite oder auch durch die verschiedenen Benutzer.

Die Darstellung zeigt einen klassischen Fall verschiedener Repräsentationen desselben Datums. In der Softwarearchitektur, insbesondere bei der Konstruktion grafischer Oberflächen, benötigen wir Mechanismen, um diese unterschiedlichen Repräsentationen miteinander in Beziehung setzen zu können. Die Änderung einer der Repräsentationen sollte von einem Laufzeitsystem, etwa durch Events oder Notifikationen, den übrigen Beteiligten mitgeteilt werden. Diese können dann selbstständig entscheiden, welche Aktionen sie ergreifen (etwa: Daten neu lesen, Daten verwerfen, Daten als ungültig markieren o. Ä.).

---

[39] Etwa: Microsoft© Kinect: *http://www.microsoft.com/en-us/kinectforwindows/*

[40] Ein Beispiel ist der LeapMotion-Sensor, der Bewegungen von Hand- und Fingersegmenten im Bereich von Millimetern erkennen kann: *http://leapmotion.com*

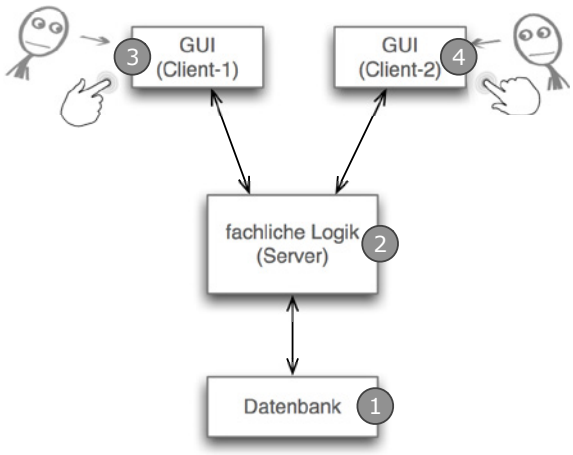


BILD 7.20
Notwendigkeit von Data-Binding

Insbesondere sollte ein Data-Binding-Mechanismus folgende Aufgaben für Sie lösen:

- Zirkuläre Abhängigkeiten von Repräsentationen erkennen und die damit möglichen unendlichen Event-Schleifen auflösen können.
- Transitive Abhängigkeiten von Repräsentationen erkennen und entsprechende Events auslösen – etwa aus dem Ursprungsdatum berechnete Informationen dann ebenfalls als geändert markieren.
- Senden und Verarbeiten von Events an Bedingungen knüpfen oder nur selektiv bestimmte Events „abonnieren“ können.

Client/Server-Verteilung

Oftmals läuft die Benutzeroberfläche eines Systems (= Client) getrennt von der Backend-Verarbeitung (= Server). Mit dieser Trennung gehen eine Reihe möglicher Schwierigkeiten einher:

- Remote-Kommunikation zwischen Client und Server, bedeutet fast immer einen höheren Implementierungsaufwand, beispielsweise durch Serialisierung/Deserialisierung, Remote-Call-Infrastruktur oder die Behandlung möglicher netzwerkbedingter Fehler.
- Netzwerke zwischen Clients und Servern können, wie generell in verteilten Systemen, mit zeitverzögerter Übertragung (Latenz) zu kämpfen haben. Für zeitkritische Aufgaben an Benutzeroberflächen können solche Verzögerungen hinderlich wirken.
- Auf Client und Server können unterschiedliche Betriebssysteme, Laufzeitumgebungen oder Implementierungstechnologien eingesetzt werden. Dies ist etwa bei Web-Anwendungen der Normalfall – wo auf (modernen) Clients oft Browser & JavaScript-Runtimes verwendet werden, auf der Serverseite aber ganz andere Technologien vorherrschen.
- Fachliche Logik auf Client oder Server implementieren: Implementierung auf dem Client macht diese Prüfungen für Benutzer gefühlt performanter, möglicherweise muss diese Logik dann redundant für verschiedene Client-Typen implementiert werden und wäre schwerer wartbar. Die Implementierung im Server wäre universell verwendbar und zentralisiert.
- Synchronisation von Aktionen mehrerer paralleler Benutzer: Falls mehrere Benutzer parallel dieselben Objekte oder Daten bearbeiten, müssen Sie Aspekte wie Konsistenz, Transaktionalität, Rollback oder Kompensation beachten.

Internationalisierung

International eingesetzte Systeme müssen neben verschiedenen Sprachen (= Locales) und Zeichensätzen auch mögliche weitere Aspekte berücksichtigen:

- Formatierung von Daten, etwa: Uhrzeit, Datum, Telefonnummern
- Syntaktische Unterschiede in ähnlichen Daten, etwa: Postleitzahl
- Schreibrichtungen: im westlichen Sprachraum links-nach-rechts, in arabischen Ländern rechts-nach-links, in manchen asiatischen Ländern oben-nach-unten.
- Länderübergreifende Uhrzeitregelungen:[41] Welche Zeitzone soll eine internationale Anwendung als Basis verwenden? Falls mehrere Zeitzonen möglich sind, gibt es im System keine „Normalzeit", sondern nur benutzerrelative Zeiten. Ach ja – da waren noch Sommer- und Winterzeit.
- Kulturelle Unterschiede, insbesondere in Bild- und Zeichensprache, sowie Farbwahl. Schwarz gilt in vielen westlichen Kulturen als Farbe der Trauer, andere Ländern verwenden dafür Weiß.
- Bezeichnung von Personen: Ein- oder zwei Vornamen, mit oder ohne Suffix („Jr."), die Vornamen vor den Nachnamen oder dahinter, ist ein Vorname zwingend (in vielen afrikanischen Kulturen gibt es beispielsweise keine Vornamen!), bleibt die Anrede nach einer Hochzeit identisch, oder ändert sich ... hier drohen Ihnen eine Menge möglicher „Fettnäpfchen".

Die Optik: Farben, Fonts, Fläche, Anordnung

Die Festlegung optischer und gestalterischer Details erscheint uns IT'lern manchmal als simple Kleinigkeit – in der Realität verbirgt sich dahinter die hohe Kunst des Designs.

Im Interesse Ihrer zukünftigen Benutzer – lassen Sie einen erfahrenen, ausgebildeten (Interaktions-)Designer die optischen Details Ihrer Benutzeroberfläche entscheiden. Ich habe oft und intensiv unter Oberflächen gelitten, die von mir und anderen „IT-Geeks" entworfen wurden – fast immer mit dem Ergebnis mangelhafter Akzeptanz und entsetzter Benutzer. ☹

Ich empfehle Ihnen einen Blick in das „Non-Designers Design-Book" [Williams-2008], um die Vielfalt rund um dieses Thema zu erleben und verständliche und hilfreiche Ratschläge aufzugreifen.

7.6.6 Zusammenhang zu anderen Aspekten

In den Architekturmustern und -stilen aus Kapitel 6 finden Sie eine Reihe von Lösungsvorschlägen zur Konstruktion von Systemen mit Benutzeroberflächen – siehe insbesondere Kapitel 6.2.8 über die interaktionsorientierten Systeme.

[41] Wer mal versucht hat, in einem der verbreiteten Kalender-Programme (ziemlich egal, in welchem) einen internationalen Termin über mehrere Zeitzonen zu organisieren, weiß, wovon ich hier spreche.

7.7 Workflow-Management: Ablaufsteuerung im Großen

von Martin Bartonitz

Ablaufsteuerung im Sinne von Workflow-Management bezieht sich auf systemübergreifende Aspekte: Steuerung und Koordination zwischen verschiedenen Software-Systemen.

Workflow ist für Softwarearchitektur aus folgenden Gründen von Bedeutung:

- Es betrifft die Schnittstellen zwischen verschiedenen Systemen.
- Es beeinflusst möglicherweise die Dienste oder Schnittstellen, die ein System „nach außen" bereitstellt.
- Zur Abwicklung komplexer Geschäftsprozesse sind oftmals mehrere Software-Systeme notwendig.

Synonym zu *Workflow-Management* verwenden wir den Begriff *Business Process Management* – abgekürzt BPM. Während Workflow-Management sich vordringlich auf die technische Abwicklung von Aktivitätsketten fokussierte, hat BPM den Anspruch einer ganzheitlichen Sicht auf die Geschäftsprozesse. Neben der verwendeten Technik werden auch Methoden zur kontinuierlichen Prozessoptimierung betrachtet.

Einen großen Einfluss auf eine Reihe ergänzender Aspekte des BPM hatten mehrere neue Standards. Anfang des Jahrtausends erblickten die Web-Services das Licht der weiten Welt. In großen Prozessumgebungen müssen diverse Web-Services „orchestriert" werden. Dazu wurde der Standard BPEL (= Business Process Execution Language) durch die marktbestimmenden Unternehmen unter dem Dach der OASIS spezifiziert und in ihren BPM-Suiten umgesetzt.

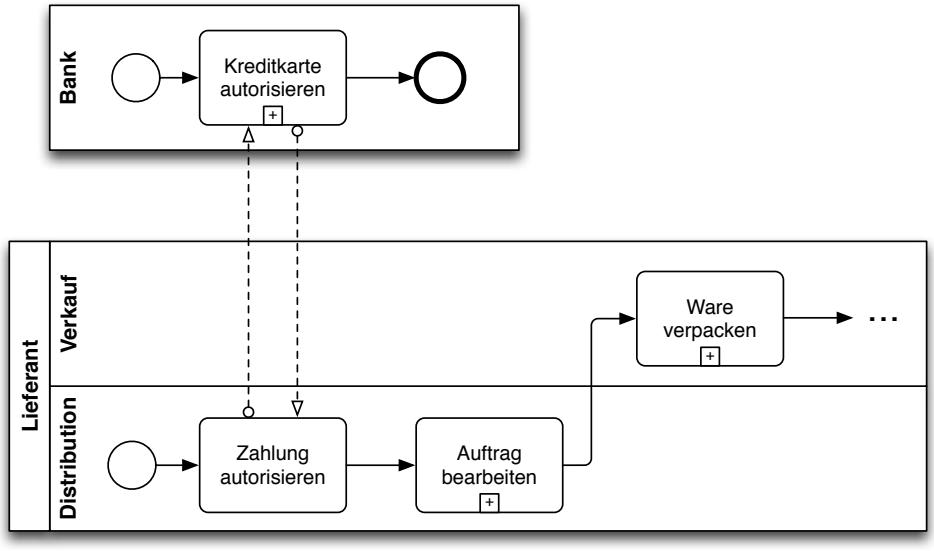


BILD 7.21 BPMN-Beispiel

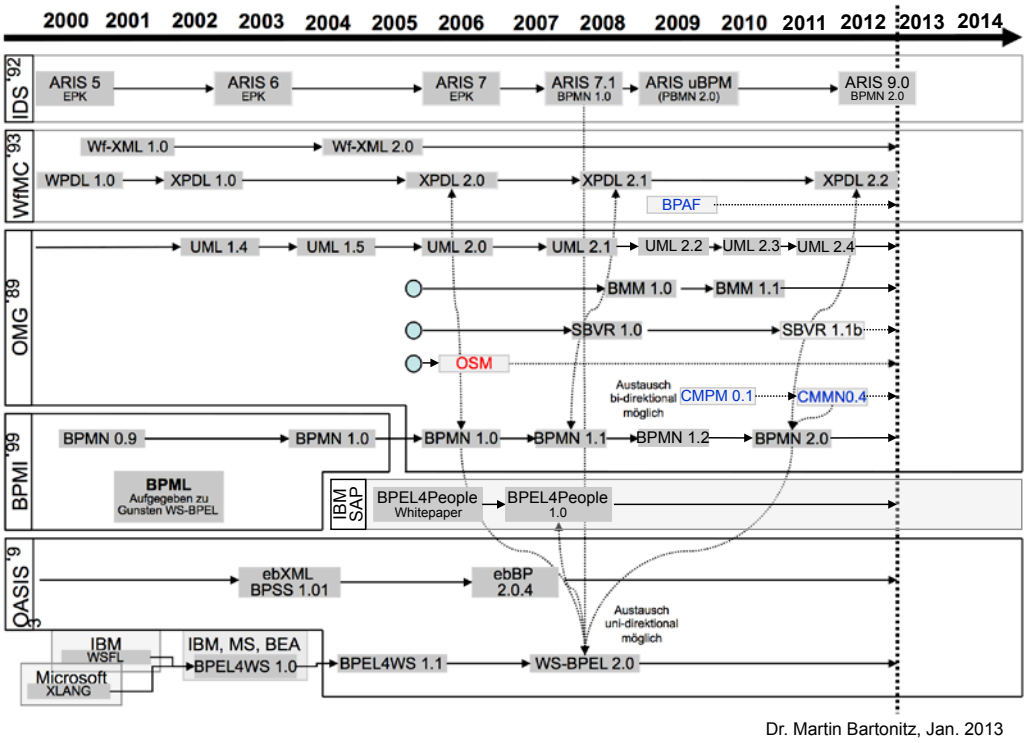


BILD 7.22 Entwicklung von Standards im Geschäftsprozessmanagement

Während BPEL die Aufgabenketten ohne Beteiligung von Menschen steuerte, werden diese in der XPDL = XML Process Definition Language der Workflow Management Coalition (WfMC) berücksichtigt (human centric anstelle von machine centric). Die XPDL war auch geplant als Austauschformat zwischen unterschiedlichen Anwendungen wie Modellierungs- und Simulations-, aber auch Workflow-Engines unterschiedlicher Hersteller. Eine ernst zu nehmende Nutzungsverbreitung haben jedoch weder XPDL noch BPEL erlangt.

Es gibt seit 2010 einen hoffnungsvollen Kandidaten, der die gewünschte Interoperabilität noch erlangen könnte. Dabei handelt es sich um die BPMN = Business Process Model and Notation. Die BPMN diente in ihrer ersten Version allein der grafischen Dokumentation von strukturierten Aufgabenketten. Mit Version 2.0 kam eine eigene Persistierung der Modelle, die zuvor auf Basis von XPDL von einer Reihe von Herstellern unterstützt wurde.

Erste Projekterfahrungen mit der Modellierung auf Basis von BPMN haben gezeigt, dass die Dokumentationssicht der Anwender nach der Umsetzung in die Workflow-Umgebung doch deutlich anders aussah. Die Anzahl der Aktivitäten ist meist auf die Hälfte zusammengeschrumpft. Das passiert dann, wenn Modellierer Geschäftsprozesse in BPMN erfassen, ohne das technische Zielsystem im Detail zu kennen. Workflow-Systeme haben ihre Eigenheiten. Sie haben sich über die Zeit in diversen Kundenprojekten entwickelt, sprich sie folgten den dortigen weiteren Anforderungen. Wird nun ein schon vorhandenes Modell an den Workflow-Engineer übergeben, so wird er sich doppelt bemühen. Einerseits wird er die Aufgabenketten den Möglichkeiten der Engine anpassen. Aber noch viel wichtiger: er wird auf

Ergonomie achten und die Aufgaben möglichst zusammenfassen. Damit verlagern sich Teile der dokumentierten Aufgaben im BPMN Modell in die von dem Workflow Client aufgerufenen Anwendungen und unterliegen damit nicht mehr der Workflow-Engine.

Die Empfehlung des Autors lautet:

1. Sollen Prozesse ganzheitlich und in aller Tiefe dokumentiert werden, dann nutze professionelle Werkzeuge dazu, und übertrage diese inklusive der entsprechenden Transformation manuell in den Editor des Workflow Engine.
2. Sollen Prozesse nur ausgeführt werden, dann nutze den Editor der Workflow Engine und lebe mit den Restriktionen der Dokumentation.

7.7.1 Zweck der Ablaufsteuerung

Geschäftsprozesse repräsentieren Arbeitsabläufe in Unternehmen, durch die unternehmerischer Mehrwert entsteht. Geschäftsprozesse sind häufig Vorgänge, die auf Basis einiger Stammdaten zugehörige Vorgangsdaten bearbeiten. Typische Stammdaten sind Adressen oder Produktdaten, Beispiele für Vorgangsdaten sind Angebote, Aufträge oder Rechnungen.

Geschäftsprozesse werden in vielen Unternehmen auch heute noch zu einem erheblichen Teil auf Papier- oder Aktenbasis abgewickelt. Kommt eine Akte zu einem Sachbearbeiter, führt er einige spezifische Arbeitsschritte aus (ändert einige Daten, verschickt eine Rechnung oder Ähnliches) und leitet die Akte an andere Bearbeiter (etwa zur Genehmigung, zur Ansicht oder für weitere Aktionen) weiter. In zunehmendem Maße werden solche Akten (oder Vorgänge) in Unternehmen zwecks Weiterbearbeitung per Mail an andere Sachbearbeiter oder Abteilungen geschickt. Dieses Vorgehen ist umständlich, schwer nachvollziehbar und kann nicht kontrolliert werden. Ziel von Workflow-Systemen ist die (möglichst nahtlose) Einbindung solcher Abläufe in Software-Systeme.

Die Übergänge zwischen verschiedenen Systemen sollen transparent und ohne Medienbrüche erfolgen. Daher steht das Workflow-Management in enger Beziehung zu Integration und Kommunikation (siehe die Abschnitte 7.3, Integration, und 7.5, Kommunikation).

Bild 7.23 verdeutlicht, wie die Aufbau- und Ablauforganisation in Unternehmen auf den Entwurf von Software-Systemen einwirken. Workflow-Systeme sollen dabei koordinierend und steuernd auf den Ablauf anderer Systeme einwirken. Sie sollen:

- den richtigen Personen (in der Aufbauorganisation)
- zur richtigen Zeit (innerhalb der Ablauforganisation)
- mit den richtigen Werkzeugen (Software-Systemen, Applikationen)
- die richtigen Daten bereitstellen.

Zusätzlich verwalten sie Versions- und Statusinformation über die betroffenen Geschäftsvorfälle und weisen anhand von Gruppen-, Rollen- oder Verantwortlichkeitsprofilen Vorgänge flexibel zu (unter Berücksichtigung von Benutzerrechten, Vertretungs- oder sonstigen Geschäftsregeln).

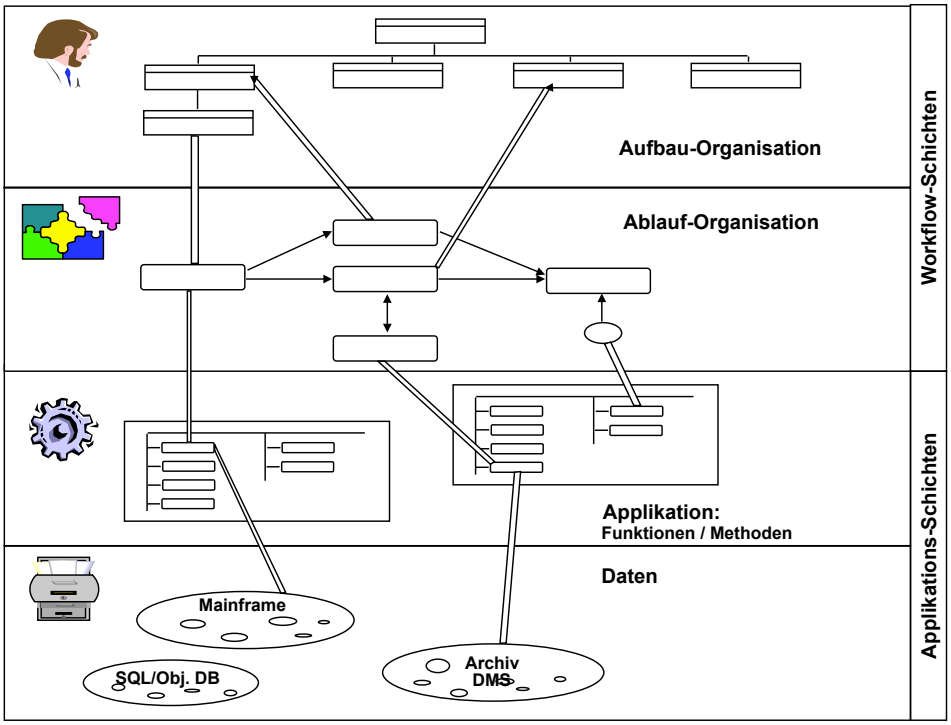


BILD 7.23 Von der Aufbauorganisation zum Software-System

Definition von Workflow-Management

Die Workflow Management Coalition[42] (WfMC) definiert Workflow-Management wie folgt:

> Workflow is concerned with the automation of procedures where documents, information or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal (WfMC Workflow Reference Model 1995, Chapter 2).
>
> Workflow-Management ist die Automatisierung von Geschäftsprozessen oder Vorgängen, in denen Dokumente, Informationen oder Arbeitsaufträge unter Berücksichtigung von Regeln oder definierten Verfahren zur Erreichung allgemeiner Geschäftsziele von einem Teilnehmer zum nächsten gereicht werden.

Diese Definition stellt Anwender in den Mittelpunkt und lässt die beteiligten Systeme unspezifiziert.

[42] *www.wfmc.org*

7.7.2 Lösungsansätze

Wollen Sie in der Architektur eines Software-Systems eine flexible Ablaufsteuerung (im Sinne der oben genannten Definition von Workflow-Management) vorsehen, so haben Sie drei Alternativen:

1. Ein Zustandsautomat mit Transitionstabellen steuert das System. Dieser Zustandsautomat ist Bestandteil des Systems selbst (*embedded workflow*).
2. Die Steuerung erfolgt durch ein externes Workflow-Management-System (WMS) oder synonym Business Process Management System (BPMS).
3. Vorgänge werden manuell per E-Mail oder Groupware weitergeleitet.

Im ersten Fall werden die Zustände der Applikation sowie die möglichen Übergänge zwischen ihnen in einer Tabelle hinterlegt und können im Falle neuer Anforderungen darin zentral administriert werden. Zustände beziehen sich hierbei auch auf Bearbeitungszustände von Vorgängen oder Dokumenten. Beispiele: „Kreditantrag genehmigt“, „Kreditantrag vorläufig freigegeben“, „Kreditantrag endgültig freigegeben“.

Im zweiten Fall wird anhand der Anforderungen des Projektes und der Organisation eine passende Standardsoftware ausgesucht. Streng genommen ist das Workflow-Management-System (WMS) selbst ein Zustandsautomat, allerdings mit einer Reihe weiterer Aufgaben. Standard-WMS haben den Vorteil, dass wie weiter unten beschrieben eine Vielzahl von fertigen Funktionen mitgeliefert werden. Oft verwenden kommerzielle Systeme eine proprietäre Benutzerverwaltung, was den Betriebs- und Verwaltungsaufwand erhöht. Die meisten Systeme bieten allerdings eine Synchronisation mit Verzeichnisdiensten per LDAP.

Im dritten Fall arbeiten Software-Systeme lose gekoppelt. Es besteht das Risiko von Medienbrüchen oder Prozessunterbrechungen. Im Falle hochgradig komplexer oder kreativer Prozesse („individuelle Vorgänge“), deren Bearbeitung stark von Heuristiken geprägt ist, bietet sich diese Alternative an.

Entscheidungskriterien

Für die Überlegung, welche dieser drei Alternativen der Ablaufsteuerung für Sie in Frage kommt, finden Sie in Tabelle 7.5 einige einfache Entscheidungskriterien.

TABELLE 7.5 Heuristik zur Entscheidung der Ablaufsteuerung

Kriterium	WMS	Groupware	Embedded
Häufigkeit	repetitiv	individuell	selten
Prozessstruktur (Regeln)	stark	schwach	stark/schwach
Benutzerrollen	viele	wenige	wenige
Transparenz	zwingend	nicht zwingend	nicht zwingend
Terminverfolgung	wichtig	unwichtig	unwichtig
Werkzeugnutzung	generell	teilweise	kaum
Dokumente (Input/Output)	bekannt	teilweise bekannt	unbekannt
Externe Schnittstellen	klar definiert, automatisierbar	wenig externe Kommunikation	keine

Definition von Workflow-Management-Systemen

Die Definition eines Workflow-Management-Systems (WMS) durch die WfMC hat einen technischen Fokus:

A Workflow Management System is one which provides procedural automation of a business process by management of the sequence of work activities and the invocation of appropriate human and/or IT resources associated with the various activity steps. (WfMC – Workflow Reference Model, Chapter 2)

Ein Workflow-Management-System ist ein System, das Prozesse definiert, erzeugt und steuert durch den Einsatz von Software, die auf einer oder mehreren Workflow Engines läuft, die Prozessdefinitionen interpretieren, mit Anwendern interagiert und, wo gefordert, Software-Systeme aufruft.

Typische Funktionen von WMS

Workflow-Management-Systeme enthalten einige typische Funktionsgruppen:

- Definitions- oder Modellierungswerkzeuge für die Aufbauorganisation (die Benutzerverwaltung soll sicherstellen, dass rollen- oder personenbezogene Rechte auch systemübergreifend korrekt eingehalten werden) und die Ablauforganisation (Geschäftsprozesse). Die Systeme stellen hierfür (grafische) Werkzeuge zur Verfügung, mit deren Hilfe die entsprechenden Ablaufparameter zu den beteiligten (externen) Software-Systemen hinterlegt werden können.
- Sowohl der Client als auch der Server müssen in der Lage sein, an vorgesehenen Stellen im Geschäftsprozess beteiligte Fremdsysteme (Applikationen) aufzurufen. In diesem Fall werden notwendige Parameter mit der Applikation ausgetauscht. Siehe hierzu den Abschnitt 7.3 (Integration).
- WMS benötigen eine Administrationskomponente, mit der notwendige Systemparameter gepflegt werden können (etwa der Arbeitstagekalender).
- Auf Basis einer Protokollierungskomponente können Berichte über den Verlauf der Prozesse erstellt werden. Hier sei der Hinweis erlaubt, dass in Deutschland der Betriebsrat oder Personalrat großen Einfluss auf die Ausgestaltung der entsprechenden Prozesse und auf die zu protokollierenden Daten besitzt. Involvieren Sie diese Organisationseinheiten deshalb frühzeitig.

Praxisrelevante Zusatzfunktionen

Workflow-Management-Systeme müssen mehrere andere Systeme steuern. Diese Steuerung umfasst eine Reihe von Zusatzaufgaben, die spezifisch für die Bearbeitung von Geschäftsprozessen ist:

- Integration mit einem Postkorb: Ziel ist es, die Nutzung von Workflow- und E-Mail-Systemen möglichst einheitlich im Sinne der Benutzerakzeptanz zu gestalten. Daher bieten einige WMS-Plugins für die wichtigsten Mail-Systeme an, über die sie Worklist-Daten direkt in diesen Applikationen präsentieren und Workflow-Funktionen zur Ausführung anbieten.

- Wiedervorlage: Ein Auftrag wird mit einem Termin versehen und zum definierten Datum wieder im eigenen Postkorb oder in dem einer Gruppe vorgelegt.
- Termin: Termine können für jede einzelne Aufgabe absolut oder relativ berechnet werden. Terminfestlegungen sollen eine Kontrolle des Durchflusses ermöglichen. Wird der Termin überschritten, werden diverse Mahnungen fällig oder andere Eskalationswege automatisch initiiert.
- Monitor: Diese Funktion führt Buch über den Status sowie die Historie der Prozesse. Sie ist damit in der Lage, Terminüberschreitungen, Laufzeitüberschreitungen und Ähnliches zu erkennen und zu dokumentieren. Neue Begriffe am Markt sind hier Business Activity Monitoring, Complex Event Processing und Business Performance Management. Das System soll mehr und mehr sich anbahnende Probleme, die durch Ressourcen-Engpässe verursacht werden, frühzeitig erkennen, Warnungen erhalten und damit proaktiv gegensteuern können.
- Ablage: Ein Prozess ohne elektronische Dokumente ist kaum noch denkbar. Diese Dokumente müssen abgelegt, oder die Referenz auf die Ablage muss mitgeführt werden.

Aufbau von Workflow-Management-Systemen

WMS sind im Normalfall als Client/Server-Systeme implementiert. Bild 7.24 verdeutlicht, dass sowohl der Workflow Client als auch der Server (die sogenannte *Engine*) mit vorhandenen Applikationen zwecks Steuerung kommunizieren.

Der Workflow Client (Worklist Handler) bietet dem Benutzer eine Liste der Arbeitsaufträge (Activity), die durch ihn zu erledigen sind. Initiiert der Benutzer eine der Aufgaben, so steuert der Client anhand der hinterlegten Parameter die jeweils nutzbare Anwendung an. Abschließend wird der Geschäftsfall in seinem Status weitergeschaltet, d. h. zur nächsten Aktivität geleitet. Können mehr als eine Aktivität erreicht werden, kann die Entscheidung entweder durch den Benutzer oder vordefiniert durch die Workflow Engine erfolgen.

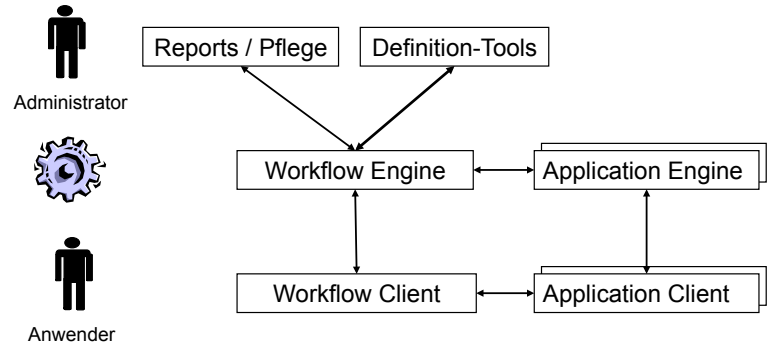


BILD 7.24 (Reduziertes) WfMC-Workflow-Referenz-Modell

7.7.3 Integration von Workflow-Systemen

Die Integration eines Workflow-Management-Systems mit anderen Systemen kann auf zweierlei Arten erfolgen:

- Ein Workflow-Management-System integriert die externen Systeme (siehe oben).
- Externe Systeme integrieren die Workflow-Funktionen: Arbeiten Benutzer immer nur mit einer Applikation, kann die Integration so gestaltet werden, dass über eine Programmierschnittstelle des Workflow-Management-Systems die Worklist von der Engine gelesen und in der Applikation angezeigt wird. Hierbei verschmelzen die beiden in Bild 7.24 dargestellten Clients zu einer gemeinsamen Anwendung.

Falls Sie ein System entwerfen, das mit einem WMS kooperieren soll, sollten Sie diese Aspekte mit Ihren Auftraggebern klären. Diese Entscheidung besitzt großen Einfluss auf die Gestaltung der Ablaufsteuerung innerhalb Ihres Systems, obwohl das WMS primär die übergreifende Steuerung behandelt. ■

Ein anderer Aspekt der Integration, der zunehmend wichtiger wird, ist der Austausch mit anderen Prozessmodellierungs- oder Simulationstools. Viele große Firmen setzen zur Dokumentation ihrer Geschäftsprozesse grafische Tools[43] ein und möchten diese Modelle im WMS wieder verwenden. Für einen solchen Import externer Modelle in das Workflow-Management-System könnte zukünftig der Standard BPMN 2.0 in Frage kommen (siehe die Einleitung zum Workflow-Management). Anschließend werden die Modelle den Eigenschaften der Workflow-Engine angepasst. Dabei erfahren die Modelle meist eine Metamorphose: es werden nur etwa die Hälfte der vorher dokumentierten Aktivitäten übrig bleiben. Zudem werden die Aktivitäten mit weiteren spezifischen Steuerungsinformationen angereichert. Damit weichen die beiden Modelle stark ab, und ein erneutes Importieren ergibt keinen Sinn mehr.

Der Autor geht davon aus, dass ein manuelles Übertragen sinnvoller ist, weil dieser Prozess der Metamorphose dann viel bewusster und mit besserer Qualität durch den Workflow-Engineer im Zusammenspiel mit dem Anwender/Prozessberater erfolgen wird.

7.7.4 Mächtigkeit von WMS

WMS und BPMS müssen sich seit einigen Jahren daran messen lassen, in welchem Umfang sie die Workflow Patterns von Wil van der Aalst [VDAalst2002] umsetzen. Diese Muster beschreiben verschiedene Kombinationsmöglichkeiten bei der Abarbeitung von Aufgabenketten sowie Gestaltungsmöglichkeiten für den Datenfluss, der durchaus unterschiedliche Wege einschlagen kann.

Auf der Website[44] zu [VDAalst2002] sind die Patterns zum besseren Verständnis grafisch animiert. Eine tabellarische Aufstellung beschreibt eine Reihe von Systemen bezüglich ihrer Umsetzung der Patterns. Ebenso sind eine Reihe von Prozessbeschreibungssprachen, darunter auch die schon erwähnten XPDL und BPEL, hinsichtlich ihrer Pattern-Vollständigkeit beurteilt worden.

[43] Visio™, ARIS™ oder andere Zeichenwerkzeuge.
[44] *www.workflowpatterns.com*, eine Initiative der Universitäten Eindhoven und Queensland.

7.7.5 Weiterführende Literatur

[VDAalst2002] ist eine Übersicht über Workflow-Pattern.

[Bartonitz2005] gibt einen Überblick über die wichtigsten Standards inklusive der historischen Entwicklung rund um Workflow-Management, fortgesetzt in [Bartonitz2006, 2010].

Martin Bartonitz (martin@bartonitz.net) wandte sich 1992 von der Prozesssteuerung in der Welt der Physik der Vorgangssteuerung in der Bürowelt zu. Erste Erfahrungen damit machte er beim Hersteller des Workflow-Management-Systems COSA Workflow als Projektleiter, Trainer und Produktmanager. Zwischen 1998 und 2003 stellt er sein Wissen auf den Gebieten Methoden und Werkzeuge im Workflow-Umfeld als Berater zur Verfügung. Seit 2004 arbeitet er als Produktmanager in den Bereichen Workflow- und Dokumenten.Management, zuerst bei der Firma SAPERION AG, aktuell bei der OPTIMAL SYSTEMS GmbH.

7.8 Sicherheit

von Wolfgang Korn und Tobias Hahn

„There's no such thing as perfect security."

– Bruce Schneier

In der modernen Gesellschaft finden sich Computer überall. Egal ob im Auto, an der Supermarktkasse oder im eigenen Smartphone – die Informationstechnologie hat inzwischen alle Bereiche des täglichen Lebens erreicht. Daten werden kontinuierlich erfasst, verarbeitet und zwischen verschiedenen Systemen übertragen. Die Sicherheit der Daten und Systeme zu gewährleisten, ist eine Herausforderung, die bereits in der Konzeptionsphase beginnt und in allen Phasen der Softwareentwicklung beachtet werden muss.

In diesem Abschnitt werden Sie mit den grundlegenden Konzepten der IT-Sicherheit vertraut gemacht. Anhand von ausgewählten Beispielen wird verdeutlicht, wie sich einzelne Herausforderungen in der Praxis lösen lassen.

7.8.1 Motivation – Was ist IT-Sicherheit?

Das BSI definiert IT-Sicherheit als den „Schutz von Informationen". Die persönlichen Daten von Privatpersonen, die Kundendaten einer Bank oder die Firmengeheimnisse eines Unternehmens sind schützenswerte Daten und sollten nur für berechtigte Personen oder Systeme zugänglich sein. Erhöht man allerdings den Schutz der Daten, z. B. indem man die Zugriffsmöglichkeiten einschränkt, hat das normalerweise Auswirkungen auf andere Eigenschaften des Systems, wie zum Beispiel auf die Benutzbarkeit und somit auch auf die Produktivität.

Professor Eugene Spafford, bekannt geworden unter anderem durch die Analyse des ersten Computerwurms:

> „The only true secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards – and even then I have my doubts."

Besonders wichtig wird dieser Punkt durch die Reaktion eines normalen Nutzers auf Sicherheitsmechanismen, die ihn bei der Arbeit einschränken. Muss der Nutzer selbst einen zusätzlichen Aufwand leisten, um die Sicherheit zu erhöhen, z. B., indem er verschlüsselte Mails verschickt, wird er in den meisten Fällen darauf verzichten und den für ihn einfacheren, jedoch unsichereren Weg wählen. Selbst in den Zeiten von PRISM werden die meisten E-Mails immer noch unverschlüsselt verschickt, obwohl es mit PGP bereits seit Langem eine inzwischen auch relativ einfach zu bedienende Möglichkeit zur Verschlüsselung von E-Mails gibt. Eine von Telepolis durchgeführte Umfrage unter deutschen Journalisten zeigte, dass die meisten nicht in der Lage sind oder kein Interesse daran haben verschlüsselte E-Mails zu verschicken oder zu empfangen. Eine hohe Akzeptanz von IT-Sicherheit durch den Nutzer wird nur erreicht, indem die Schutzmechanismen möglichst transparent umgesetzt werden.

7.8.2 Sicherheitsziele

Die grundlegenden Konzepte der IT-Sicherheit sind Vertraulichkeit, Integrität und Verfügbarkeit, die je nach Anwendungsgebiet unterschiedlich wichtig bewertet werden.

Vertraulichkeit

Vertraulichkeit = lesbar nur für Zielgruppe

Vertraulichkeit bedeutet, dass Daten lediglich von denjenigen Personen oder Systemen gelesen werden können, für die diese Daten bestimmt sind. Insbesondere bei Systemen, die über Rechnernetze oder das Internet miteinander kommunizieren, ist bei der Klartextkommunikation keine Vertraulichkeit gegeben. Daten passieren auf ihrem Weg durch solche Netze verschiedene Stationen, die in der Regel von Fremden administriert werden. Da die Sicherheit dieser Stationen weder eingeschätzt noch beeinflusst werden kann, sollte hier immer vom schlimmsten Fall ausgegangen werden.

Denken Sie pessimistisch: Sehen Sie Klartextkommunikation über Rechnernetze grundsätzlich als unsicher an! Firmeneigene interne Netze stellen hier keine Ausnahme dar!

Integrität

Integrität = Manipulationen entdecken

Integrität bedeutet, dass Daten nicht unberechtigt in ihrem Inhalt manipuliert werden.[45] Diese Forderung gilt während der Kommunikation, aber auch wenn sich Daten gerade in der Datenbank ausruhen.[46]

[45] Das Format der Daten kann sich durch die technischen Gegebenheiten der Kommunikation durchaus ändern, etwa durch Protokoll- oder Zeichensatzkonvertierungen.

[46] Oder sich mit Freunden im Container des Application-Servers treffen.

Vor der Veränderung der Daten während der Kommunikation kann ein Software-System grundsätzlich nicht geschützt werden. Es ist jedoch möglich, solche Veränderungen oder Manipulationen systematisch aufzudecken. Man spricht dabei von der Gewährleistung der Datenintegrität. Verfahren, die hierfür eingesetzt werden können, sind Hash-Funktionen und digitale Signaturen.

Authentifizierung und Autorisierung

Authentiztät = Nachweis der Identität

Um Informationen und Daten kontrolliert verfügbar zu machen, werden die Konzepte der Authentifizierung und Autorisierung verwendet. Bei der Authentifizierung wird nachgewiesen, dass der Nutzer, der auf Daten zugreifen will, auch wirklich der ist, der er vorgibt zu sein. Dieser Nachweis kann auf verschiedenen Wegen erfolgen, z. B. durch Passwörter (etwas, das der Nutzer kennt), durch Smartcards oder kryptographische Tokens (etwas, das der Nutzer besitzt) oder durch biometrische Merkmale, wie einen Fingerabdruck (etwas, das der Nutzer ist). Zur Erhöhung des Sicherheits-Niveaus können auch verschiedene Methoden kombiniert werden, man spricht dann von Zwei- oder generell Mehr-Faktor-Authentifizierung. Ein bekanntes Beispiel hierfür ist Online-Banking: Nach der Anmeldung mit Benutzername und Passwort werden zusätzlich TANs verwendet, die sich im Besitz des Nutzers befinden. Dieses Verfahren ist weit verbreitet und wird von großen Teilen der Nutzer akzeptiert.

Der Vorgang der Authentifizierung selbst kann sich sowohl auf Personen (etwa bei der Anmeldung an ein System) als auch auf Software-Systeme beziehen. In letzterem Fall weist beispielsweise ein Web-Server gegenüber dem Benutzer seine Identität nach.

Autorisierung = Berechtigung

Unmittelbar verknüpft mit der Authentisierung ist die Autorisierung. Im Rahmen der Autorisierung werden Benutzern oder Systemen nach erfolgreicher Authenfizierung ihrer Identität entsprechende Berechtigungen eingeräumt. Anhand der zugewiesenen Berechtigungen wird dann überprüft, ob auf die angeforderten Ressourcen oder Daten zugegriffen werden darf. Die zugewiesenen Rechte sollten dabei möglichst dem *„Principle of least privilege“* folgen: Der Nutzer darf nur auf Ressourcen zugreifen, die er unbedingt für die Erfüllung seiner Aufgabe benötigt.

Verfügbarkeit

Denial-of-Service

Neben den genannten Zielen der Sicherheit von Datenkommunikation gibt es nach dem Modell l des Bundesamtes für Sicherheit in der Informationstechnik noch das Ziel der „Verfügbarkeit“. Dies bedeutet, dass der Zugriff auf Daten und Programme für berechtigte Nutzer jederzeit möglich sein muss. Ein Beispiel für den zumindest temporären Verlust von Verfügbarkeit sind die sogenannten *„Denial-of-Service“*-Angriffe, bei denen z. B. ein Webserver durch eine sehr große Anzahl von gefälschten Anfragen überlastet wird. Der Webserver kann dann keine weiteren Anfragen mehr bearbeiten, die Verfügbarkeit ist nicht mehr gegeben.

Elektronische Kommunikation muss gesichert werden

Ungesicherte elektronische Kommunikation besitzt nicht einmal das Sicherheitsniveau einer Postkarte: Während bei einer Postkarte in der Regel lediglich die Vertraulichkeit verletzt ist, erreicht die ungesicherte elektronische Kommunikation keines der drei Sicherheitsziele.

Veränderungen und falsche Absender können bei einer (handschriftlichen) Postkarte leicht erkannt werden, während dies bei elektronischen Daten nicht ohne weitere Hilfsmittel möglich ist.

Anhand einer kurzen Geschichte soll hier verdeutlicht werden, welche Auswirkungen die Verletzung der einzelnen Sicherheitsziele haben kann:

... Wie schon in den vergangenen Jahren verliefen auch die diesjährigen Gehaltsgespräche nicht zu Mallory's Zufriedenheit. „Das muss sich jetzt ändern!", dachte sich Mallory, seines Zeichens Systemverantwortlicher der Firma Insecure Communications Inc. Bereits im vergangenen Jahr hatte Mallory beim routinemäßigen Schnüffeln in den Mails seiner Kollegen festgestellt, dass sein Chef Bob nach Abschluss der Gehaltsverhandlung die Ergebnisse per Mail der zuständigen Personalreferentin Alice mitteilte. „Gut, dass Bob ein so ausgeprägtes Sicherheitsbewusstsein besitzt. Leichter kann er es mir nicht machen.", sagte sich Mallory und machte sich auch gleich ans Werk. Er selbst betreute den bei Insecure Communications eingesetzten Mail-Server, was ihm sein Vorhaben deutlich erleichterte. Das „Korrigieren" seiner Gehaltserhöhung war insofern eine Kleinigkeit für ihn. Die kleine Änderung würde Alice sicher nicht bemerken – wie auch. Nebenbei verschaffte sich Mallory auch gleich einen Überblick über die Ergebnisse seiner Kollegen. Dabei stieß er auf Cora Victim, seine Kollegin in der Systemadministration. Ihr gutes Abschneiden in der diesjährigen Gehaltsrunde erhöhte seinen Neid auf die begabte Kollegin. „Wenn ich schon mal dabei bin, könnte ich auch gleich etwas gegen die ständige Bevorzugung von Cora tun. Eine kurze Mail von Cora an Alice, in der sie sich über ihre nicht angemessene Gehaltserhöhung beschwert, sollte erst mal reichen, um sie ein wenig in Misskredit zu bringen." ...

7.8.3 Lösungskonzepte

Verschlüsselung
Plaintext
Ciphertext

Vertraulichkeit, Integrität und Authentizität lassen sich durch die Verwendung von kryptographischen Verfahren und Algorithmen auf einem hohen Sicherheitsniveau erreichen.

Eine der elementaren Anwendungen von Kryptographie ist die Verschlüsselung (siehe Bild 7.25): Der zu übermittelnde Klartext (*Plaintext*) wird vom Absender durch einen Algorithmus in eine verschlüsselte Form (*Ciphertext*, von engl. cipher:

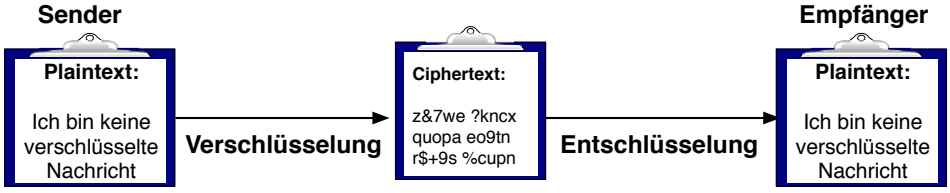


BILD 7.25 Prinzip der Verschlüsselung

Verschlüsselung) überführt. Dieser Ciphertext wird zum Empfänger übertragen und dort wieder in den Klartext entschlüsselt. Wird bei gleichem Klartext ein unterschiedlicher Schlüssel verwendet, ergibt sich ein unterschiedlicher Geheimtext.

Grundlagen kryptographischer Verfahren

Die Sicherheit von modernen kryptographische Verfahren, die zur Gewährleistung der oben genannten Ziele eingesetzt werden, hängt von der Geheimhaltung der dabei verwendeten Schlüsseln ab. Der jeweils eingesetzte Algorithmus sollte allgemein bekannt sein.[47]

In die Transformation von Plaintext in Ciphertext geht eine Variable ein, der sogenannte Schlüssel. Der Schlüssel bestimmt, wie die Transformation der Eingabedaten erfolgt. Die Sicherheit dieser Verfahren beruht darauf, dass die benötigten Schlüssel nur den berechtigten Teilnehmern zugänglich sind und nicht erraten oder ausprobiert werden können. Mit Kenntnis des Schlüssels kann jedermann Nachrichten verschlüsseln und verschlüsselte Nachrichten lesen. Ohne Kenntnis des Schlüssels sollte es nicht möglich sein, vom Klar- auf den Geheimtext (und umgekehrt) zu schließen.

Für die Auswahl der Verfahren sollte man folgende Regeln berücksichtigen:

Verwenden Sie **ausschließlich** Verfahren, deren Sicherheit nur auf der Geheimhaltung der verwendeten Schlüssel basieren. Dieser wichtige Grundsatz wurde bereits 1883 formuliert und ist als *Kerckhoffs'sches Prinzip* bekannt. Algorithmen, deren „Sicherheit" in der Unkenntnis des Verfahrens besteht (*Security by Obscurity*), haben zwangsläufig das Problem, dass keine zuverlässigen Erkenntnisse über ihre Sicherheit vorliegen.

Eigene Implementierungen bekannter Algorithmen sind selten sinnvoll. Für nahezu alle Verfahren existieren kommerzielle oder freie (teilweise Open-Source-)Produkte. Auf Grund der breiteren Anwendung der Produkte werden Sicherheitsmängel schneller bekannt und behoben, als dies bei Eigenentwicklungen der Fall ist.

Verschlüsselungsverfahren

Für die Verschlüsselung gibt es symmetrische und asymmetrische Verfahren:

- Bei symmetrischen Verfahren kommt für Verschlüsselung und Entschlüsselung der gleiche Schlüssel zum Einsatz. Um verschlüsselt miteinander kommunizieren zu können, müssen alle Teilnehmer den geheimen Schlüssel kennen. Da die Sicherheit des Verfahrens von der Geheimhaltung des Schlüssels abhängt, muss ein sicherer Austausch der Schlüssel zwischen allen beteiligten Partnern stattfinden. Bei Kommunikation mit mehr als zwei Beteiligten ist auch die Anzahl der benötigten Schlüssel (n*(n-1)/2) kaum handhabbar. Ein symmetrisches Verfahren kann als sicher angesehen werden, wenn es keinen Angriff gibt, der wesentlich schneller als das komplette Durchsuchen des Schlüsselraums ist (*Brute-Force Angriff*). Der bekannteste Vertreter dieser Verfahren ist AES.

[47] Die Nutzung bekannter Algorithmen bietet den Vorteil, dass diese von unabhängigen Personen geprüft und auf ihre Sicherheit hin getestet werden können. Erkannte Sicherheitsmängel werden publiziert und die Algorithmen oder Implementierungen entsprechend angepasst.

- Bei asymmetrischen Verfahren wird ein Schlüsselpaar verwendet. Einer der beiden Schlüssel wird nur zur Verschlüsselung verwendet und ist öffentlich zugänglich. Die Entschlüsselung ist nur mit dem zweiten, geheimen Schlüssel möglich. Diese Verfahren werden als *Public-Key-Verfahren* bezeichnet. Wichtigster Vertreter der asymmetrischen Verfahren ist der RSA-Algorithmus. Ein Verfahren, das ausschließlich zur Signatur genutzt werden kann, ist DSA. Der Nachteil dieser Verfahren besteht in ihrer Ausführungsgeschwindigkeit: Sie sind etwa 1000-fach langsamer als ihre symmetrischen Kollegen. Aus diesem Grund kommen in der Praxis normalerweise Kombinationen beider Verfahren, sogenannte *Hybrid-Verfahren*, zum Einsatz. Die Sicherheit der asymmetrischen Verfahren hängt bei den meisten Verfahren von einem zugrunde liegenden mathematischen Problem ab, welches ohne die Kenntnis von Zusatzinformationen (in Form des geheimen Schlüssels) nicht gelöst werden kann.

Hybride Verfahren

Hybride Verfahren (siehe Bild 7.26) machen sich die Vorteile der symmetrischen (Geschwindigkeit) und der asymmetrischen Verfahren (Schlüsselverteilung) zunutze. Zunächst wird ein zufälliger symmetrischer Sitzungsschlüssel generiert. Mit diesem werden die Nutzdaten verschlüsselt (Punkt 1 in Bild 7.26). Anschließend wird der Sitzungsschlüssel mit dem öffentlichen Schlüssel des Empfängers asymmetrisch verschlüsselt (Punkt 2). Jetzt können die verschlüsselten Nutzdaten und der verschlüsselte Sitzungsschlüssel gefahrlos transportiert werden. Auf Seiten des Empfängers wird zunächst der Sitzungsschlüssel wieder (asymmetrisch) entschlüsselt (Punkt 3). Mit dem Sitzungsschlüssel können die Nutzdaten wieder symmetrisch entschlüsselt werden (Punkt 4).

Dieses Verfahren wird auch im Internet beim Aufruf verschlüsselter Webseiten über das https-Protokoll verwendet. Das vom Server übertragene Zertifikat enthält dessen öffentlichen Schlüssel, mit dem der Client dann den von ihm gewählten symmetrischen Schlüssel verschlüsseln und an den Server schicken kann.

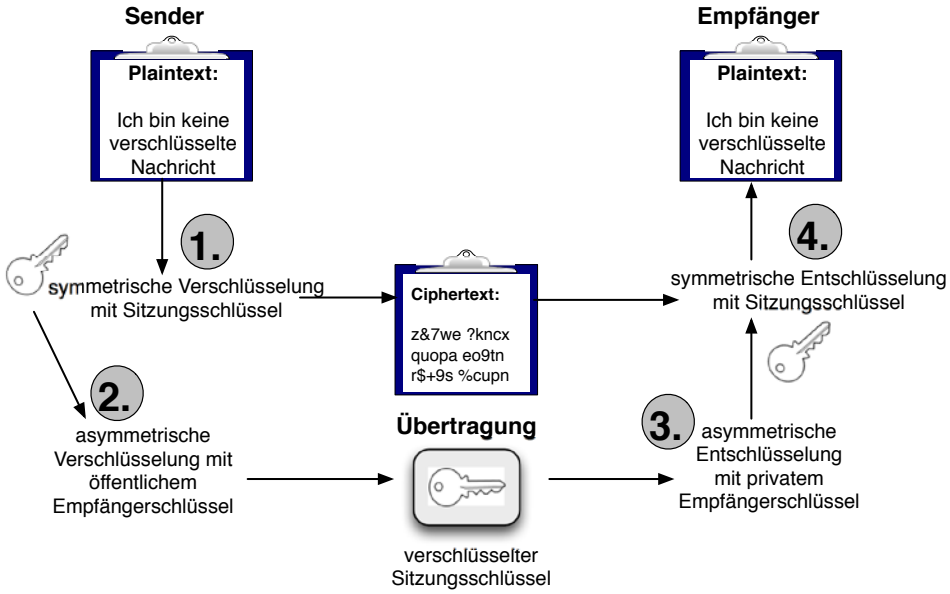


BILD 7.26 Hybride Verschlüsselungsverfahren

Integrität und Authentizität

Digitale Signatur

Integrität und Authentizität werden mit Hilfe digitaler Signaturen gewährleistet, die auf asymmetrischen Verschlüsselungsverfahren basieren. Dabei ist die Bedeutung von öffentlichem und geheimem Schlüssel vertauscht. Das Verfahren zur Erzeugung einer digitalen Signatur entnehmen Sie Bild 7.27.

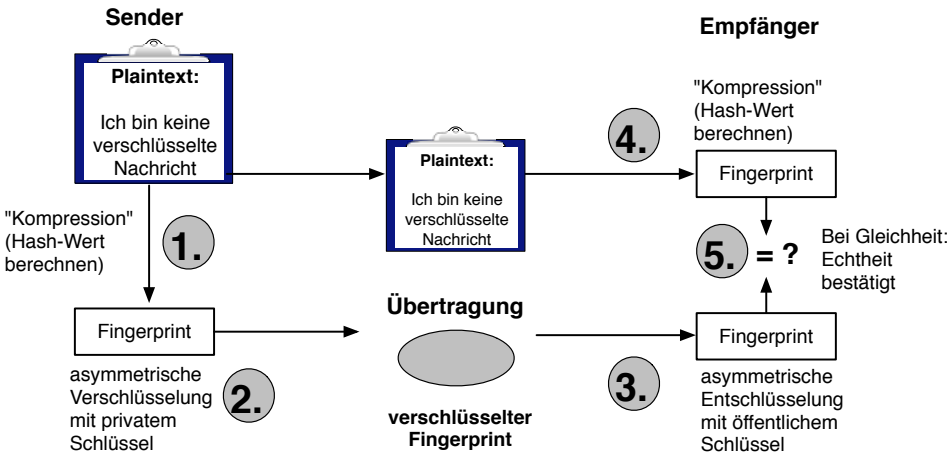


BILD 7.27 Digitale Signatur

Hashfunktion
Fingerprint
Message Digest

Zunächst wird mit Hilfe einer kryptographischen Hashfunktion (etwa: MD5, SHA-1) eine Prüfsumme (*Fingerprint, Message Digest*) der zu signierenden Nachricht errechnet. Daran anschließend wird diese Prüfsumme mit dem privaten Schlüssel verschlüsselt. Die verschlüsselte Prüfsumme wird dann zusammen mit der Nachricht an den Empfänger übertragen. Der Empfänger entschlüsselt die Prüfsumme mit dem öffentlichen Schlüssel des Absenders. Gelingt dies, ist die Herkunft der Nachricht gesichert. Im nächsten Schritt wird die Prüfsumme der Nachricht errechnet und mit der der Nachricht beigefügten Prüfsumme verglichen. Ergeben sich Abweichungen, wurde die Nachricht während des Transports verändert.

Authentisierung durch Challenge-Response

Authentisierungen auf Basis kryptographischer Verfahren funktionieren ähnlich wie die digitale Signatur. Im ersten Schritt stellt das System, an dem die Authentisierung erfolgen soll, eine zufällige Datenfolge bereit. Diese wird von dem System, das sich authentisieren will, mit seinem privaten Schlüssel verschlüsselt und zurück übertragen. Durch Entschlüsselung dieser Daten mit dem öffentlichen Schlüssel und Vergleich mit den Originaldaten kann überprüft werden, ob die Authentisierung erfolgreich war oder nicht. Dieses Verfahren wird auch als *Challenge-Response-Verfahren* bezeichnet.

Die beschriebenen Verfahren setzen die nachprüfbare Zuordnung von öffentlichen Schlüsseln zu Personen oder Systemen voraus.[48]

[48] Falls diese Zuordnung nicht sauber gelöst ist, kann es zu den gefährlichen „*man in the middle*"-Angriffen kommen, bei denen ein Angreifer beiden Kommunikationspartnern eine falsche Identität vortäuscht und Nachrichten verfälscht oder missbraucht.

Ein Beispiel aus der Praxis: OAuth

OAuth steht für Open Authorization und ist ein offenes, auf Standards der IETF aufbauendes Protokoll zur sicheren API-Autorisierung verschiedener Applikationen. Durch die Verwendung von OAuth können Nutzer einer dritten Partei Zugriff auf ihre Web-Ressourcen erlauben, ohne ihre Credentials (Benutzername und Passwort) weitergeben zu müssen. Die Zugangsdaten und die Nutzeridentität bleiben der Client-Anwendung gegenüber verborgen. Die hierbei verwendeten Access Tokens ermöglichen (nach Autorisierung durch den Nutzer) einer Applikation im Namen eines Nutzers den Zugriff auf eine Web-Ressource. In den Tokens sind die vom Nutzer erlaubten Berechtigungen enthalten, die entsprechend dem „*Principle of least privilege*“ vergeben werden sollten.

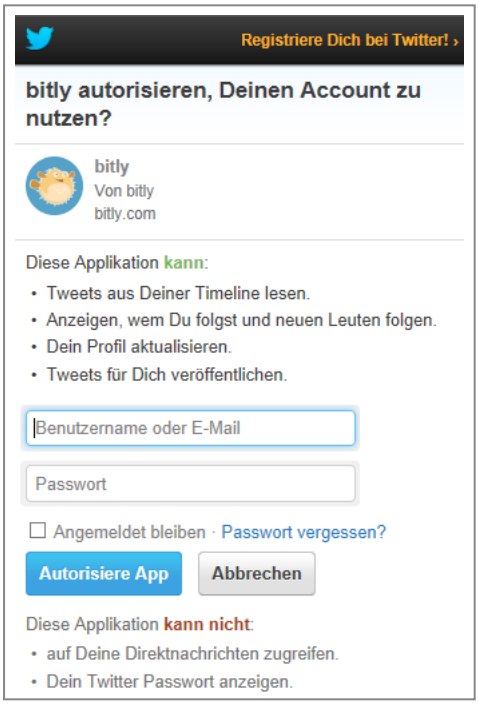


BILD 7.28
Autorisierungsdialog für bitly und twitter

OAuth bietet Unterstützung für mehrere Anwendungsfälle, die in der englischen Literatur oft als „*flows*“ bezeichnet werden. Beispielhaft wird hier das Webserver-Szenario dargestellt. Die daran beteiligten Parteien werden in drei unterschiedliche Rollen eingeteilt:

- Der Nutzer des Dienstes.
- Ein Client, z. B. eine Web- oder eine Mobil-Anwendung. Als Beispiel wird hier der Kurz-URL-Dienst *bitly* verwendet.
- Ein Server, auf dem die Ressourcen des Nutzers liegen. Unser Beispiel: *twitter*.

Will jetzt der Nutzer dem Client Zugriff auf den Server geben, werden die folgenden Schritte durchlaufen:

1. Der Nutzer öffnet die bitly Webseite und möchte bitly erlauben, auf seinen twitter-Account zuzugreifen.

2. bitly startet den OAuth-Protokollablauf, der Nutzer wird zu twitter umgeleitet, dort werden ihm die von bitly angeforderten Berechtigungen angezeigt. Stimmt der Nutzer zu, kann er, indem er sein twitter-Passwort eingibt, den Zugriff für bitly erlauben.
3. Nach erfolgreicher Autorisierung erhält bitly ein Access Token für den twitter Account des Nutzers.
4. bitly kann das Token jetzt verwenden, um im Namen des Nutzers auf dessen twitter-Account zuzugreifen.

Der bei diesem Beispiel verdeutlichte Anwendungsfall ist nur einer von vielen die mit OAuth realisiert werden können. Durch die Verwendung der anderen Szenarios lässt sich das OAuth Protokoll auch in vielen weiteren Situationen sinnvoll einsetzen. Es wird inzwischen von vielen großen Seiten verwendet, um den Zugriff per API zu kontrollieren. Hierzu gehören unter anderem twitter, Facebook, LinkedIn und XING. Google bietet Entwicklern mit dem „OAuth 2.0 Playground“ eine Möglichkeit, das bei ihren Diensten verwendete Protokoll kennenzulernen.

Zertifikate: Zuordnung von Schlüsseln an Personen oder Systeme

Zertifikate = Bindung von Schlüsseln an Personen

Die Bindung von Schlüsseln an Personen oder Systeme wird durch *Zertifikate* erreicht. Diese enthalten Informationen zum Inhaber sowie dessen öffentlichen Schlüssel. Um ein Fälschen dieser Zertifikate zu verhindern, werden sie vom Herausgeber digital unterzeichnet. Voraussetzung für die Vertrauenswürdigkeit von Zertifikaten ist daher die Vertrauenswürdigkeit des Herausgebers. Die Rolle dieser Herausgeber wird durch Zertifizierungsstellen (*trust center*) übernommen. In Deutschland unterliegen diese Zertifizierungsstellen als Voraussetzung für die Rechtsverbindlichkeit der Zertifikate einer strengen behördlichen Kontrolle.

Werden Zertifikate lediglich zur Sicherung der internen Kommunikation Ihres Unternehmens benötigt, können Sie eine eigene Zertifizierungsstelle aufbauen. Dazu einige kritische Aspekte als Entscheidungshilfe:

- Ist der Betrieb einer eigenen Zertifizierungsstelle und die Verwendung eigener Zertifikate juristisch abgesichert? Wenn Sie durch die Verwendung der Zertifikate Rechtsverbindlichkeit erreichen wollen, unterliegen Sie dem Signaturgesetz.
- Zum Betrieb einer eigenen Zertifizierungsstelle benötigen Sie qualifiziertes Personal.
- Die technische und organisatorische Infrastruktur eines Trust-Centers muss sicher und verfügbar betrieben werden. Wenn Sie beispielsweise die sichere Verwahrung Ihrer Zertifikate nicht gewährleisten können, sollten Sie auf den Betrieb eines eigenen Trust-Centers verzichten.
- Der Betrieb eines eigenen Trust-Centers verursacht durch administrative und organisatorische Prozesse erhebliche Kosten. Stellen Sie diese den Kosten für eingekaufte Dienstleistung gegenüber.
- Sowohl fachliche als auch technische Gründe können für den Betrieb einer eigenen Zertifizierungsstelle sprechen. Ist beispielsweise die Hinterlegung von Schlüsseln gefordert, lässt sich dies mit einer akkreditierten Zertifizierungsstelle nicht realisieren.

Eine Frage der Ebene

Bei der Implementierung von Sicherheitsmaßnahmen sollten Sie zuerst festlegen, auf welcher Ebene des OSI-Schichtenmodells die Realisierung erfolgt:

- Netzwerk-Ebene (OSI-Schicht 3). Eine Implementierung auf dieser Ebene hat keinen Einfluss auf ein zu entwickelndes System. Die Sicherheits-Protokolle dieser Ebene werden häufig für die Realisierung sogenannter *Virtual Private Networks* (*VPN*) genutzt. Innerhalb dieser Protokolle (etwa *IPSEC*) erfolgen Verschlüsselung, Sicherstellung der Datenintegrität und Authentisierung der Endsysteme. Die Mechanismen werden in diesem Fall auf den gesamten Datenverkehr zwischen den beteiligten Endsystemen angewandt. Für Software-Systeme ist die Verwendung der Sicherheitsmechanismen auf der Netzwerk-Ebene damit völlig transparent. Eine selektive Anwendung auf einzelne Anwendungen oberhalb der Netzwerk-Ebene ist nicht möglich. Den Zeitpunkt für Entschlüsselung und Prüfung der Datenintegrität können Anwendungen ebenfalls nicht bestimmen. Übertragene Daten werden unmittelbar nach dem Eingang beim Empfänger entschlüsselt und geprüft.
- Transport-Ebene (OSI-Schicht 4). Auch für diese Variante existieren standardisierte Protokolle wie *Secure Socket Layer (SSL)* oder *Transport Layer Security (TLS)*. Bei Einsatz dieser Protokolle kann jede Anwendung eigenständig über deren Verwendung entscheiden. Die Mechanismen beziehen nicht mehr das gesamte Endsystem ein, wie bei der Sicherung auf Netzwerk-Ebene. Generell werden aber alle über eine solche Transportstrecke übertragenen Daten gesichert. Die Anwendung hat keinen Einfluss auf den Zeitpunkt von Entschlüsselung und Integritätsprüfung. Beides erfolgt unmittelbar nachdem die Daten den Transportkanal verlassen haben. Für eine zu realisierende Anwendung bedeutet die Verwendung solcher Mechanismen nur einen geringen Zusatzaufwand. Lediglich beim Aufbau neuer Verbindungen ergeben sich Änderungen, beispielsweise durch die Behandlung von Zertifikaten. Die Abwicklung der eigentlichen Kommunikation unterscheidet sich nicht von der ungesicherten Kommunikation.
- Anwendungs-Ebene (OSI-Schicht 7). Diese Art der Implementierung gibt dem System maximale Kontrolle über die eingesetzten Sicherheitsmechanismen. Es bedeutet allerdings auch, dass die Verwendung der Mechanismen in einem System explizit programmiert werden muss. Transparenz oder Austauschbarkeit bezüglich der Verwendung von Sicherheitsmechanismen geht somit verloren. Klassisches Beispiel für dieses Verfahren ist die Verschlüsselung von E-Mails gemäß *S/MIME*.

In der Praxis hängt die Entscheidung für ein spezifisches Verfahren von den funktionalen und organisatorischen Anforderungen an das zu realisierende Software-System ab. Einige Faustregeln:

- Für Anwendungen im Kontext des Signaturgesetzes ist eine Sicherung auf Transport-Ebene nicht ausreichend, da einige gesetzliche Forderungen (etwa die Nachvollziehbarkeit einer Signaturprüfung) nicht erfüllt sind.
- Wenn Sie Daten auch nach dem Transport sichern wollen, müssen Sie Sicherungsmaßnahmen auf Applikations-Ebene einsetzen.
- Genügt Ihnen die Sicherung der Daten während der Übertragung, sollten Sie Sicherheitsmaßnahmen auf Transport-Ebene wählen.

- Müssen Sie Sicherheitsmaßnahmen in bestehende Anwendungen integrieren, können Sie dies auf der Transport-Ebene mit geringerem Aufwand erreichen.

7.8.4 Security Engineering mit Patterns

Die hier vorgestellten kryptographischen Verfahren können, wenn sie richtig eingesetzt werden, einen wichtigen Teil zur Entwicklung von sicheren Systemen beitragen. Es sind allerdings keine „Zaubermittel“, die einzusetzenden Verfahren müssen sorgfältig ausgewählt und umgesetzt werden. Selbst bei dem weltweit eingesetzten AES-Verfahren gibt es Betriebsmodi, die als unsicher angesehen werden. Da sich die meisten Software-Entwickler oder -Architekten nur selten mit Sicherheit beschäftigen, fehlt ihnen das nötige Spezialwissen.

Eine Lösung hierfür bieten die 1997 von Yoder und Baralow vorgestellten und von Schumacher und Rödig weiterentwickelten „Security Patterns“ ([Schumacher06]). Analog zu den aus anderen Bereichen bekannten Mustern wird auch hier das Wissen von Experten gesammelt und strukturiert wiedergegeben. Ein Muster beschreibt dabei ein in einem bestimmten Kontext immer wieder auftretendes Problem und bietet eine Lösung unter Einbeziehung aller zu beachtenden Aspekte. Die von Schumacher et.al. eingeführten Security Patterns bestehen aus den folgenden Elementen:

- Kontext: Anhand eines Beispiel-Szenarios werden die Umstände beschrieben, unter denen das zu lösende Problem auftreten kann. Zusätzlich wird, sofern relevant, auf bereits angewendete Security Patterns verwiesen.
- Problem: Unter Einbeziehung aller funktionalen und nicht-funktionalen Anforderungen an das System wird hier das Problem beschrieben, welches durch die Anwendung des Patterns gelöst werden soll.
- Lösung: Die Lösung des vorgestellten Problems im Rahmen des Kontexts unter Einbeziehung aller Anforderungen. Dabei kann die Lösung auf unterschiedlichen Ebenen umgesetzt werden und beinhaltet gegebenenfalls Gegenmaßnahmen, um die Auswirkungen bestimmter Angriffe oder Bedrohungen zu minimieren.
- Auswirkungen: Hier werden die Vor- und Nachteile der vorgestellten Lösung diskutiert.

Im Folgenden wird anhand eines Beispiels versucht, dem Leser die relevanten Punkte zu verdeutlichen. Das hier ausgewählte Muster ist nur eines von vielen, die in [Schumacher06] ausführlich beschrieben werden.

Security Patterns in der Praxis: Das „Security Session“ Pattern

- Kontext: In einem Mehrbenutzer-System müssen die einzelnen Komponenten in der Lage sein, die mit einzelnen Benutzern assoziierten Berechtigungen zu überprüfen.
- Problem: In komplexen Systemen kann der Nutzer an vielen verschiedenen Stellen unterschiedliche Aktionen durchführen. Die Berechtigungs-Prüfung für jede einzelne Aktion durch Eingabe von Benutzername und Passwort ist nicht praktikabel, eine Speicherung und Weitergabe der Credentials durch die einzelnen Systeme darf nicht erfolgen.

- Lösung: Es wird ein Session-Objekt eingeführt, das alle für die Berechtigungsprüfungen relevanten Benutzerdaten enthält. Jede vom Benutzer durchgeführte Aktion wird anhand des in der Anfrage enthaltenen Session-Objekts überprüft. Ein bekanntes Beispiel aus dem Bereich der Web-Anwendungen sind Cookies.
- Auswirkungen:
  - Die Bindung von Session-Objekten an Benutzer ermöglicht es, die gleichzeitige Nutzung von Credentials zu erkennen, was auf einen Kompromiss des Benutzeraccounts hindeuten kann.
  - Die Session-Objekte sind ein attratkives Ziel für Angreifer und müssen daher auch geschützt werden, z. B. durch Verschlüsselung oder Signaturen.

7.8.5 Weiterführende Literatur

Microsoft (Online: *https://www.microsoft.com/security/sdl/resources/publications.aspx*) beschreibt seinen eigenen „Security Development Lifecycle “: Für die einzelnen Phasen der Softwareentwicklung werden Security Practices vorgestellt, mit denen Sicherheitslücken verhindert, entdeckt und behoben werden können:

Das *Open Web Application Security Project* (*https://www.owasp.org*) bietet einen aktuellen Überblick über die relevanten Sicherheitsaspekte im Bereich von Web-Anwendungen.

[Schneier96] ist das Standardwerk zum Thema Kryptographie.

[Schmeh98] stellt das Thema Sicherheit und Kryptographie fundiert dar und ist dabei auch unterhaltsam zu lesen. Die mathematische Darstellung von Algorithmen wurde auf ein sehr erträgliches Maß beschränkt.

[Schumacher06] erläutert die Konzepte der Security Patterns und enthält eine ausführliche Liste von in unterschiedlichen Situationen anwendbaren Mustern.

[SigG/SigV] enthalten die rechtlichen Grundlagen zur Verwendung digitaler Signaturen in Deutschland.

[SSH] ist eine kurze Einführung in Kryptographie.

*Wolfgang Korn (*wolfgang.korn@gmx.de*) hat mehrere Jahre als Berater für IT-Sicherheit und Softwarearchitekturen gearbeitet. Zur Zeit arbeitet er als Softwarearchitekt für die blueCarat AG in Köln. Seine Mission besteht darin, seine Familie von den Vorzügen mechanischer Klaviere zu überzeugen und für seine Kunden sichere Architekturen zu entwickeln.*

Tobias Hahn arbeitet als Wissenschaftler am Fraunhofer-Institut für sichere Informationstechnologie in Darmstadt. Hier beschäftigt er sich, sowohl im Rahmen von Forschungsprojekten als auch von Industrieaufträgen, mit verschiedenen Aspekten der IT-Sicherheit.

7.9 Protokollierung

Protokollierung: Sammeln von Informationen über den Programmzustand während des Programmablaufs.

Es gibt zwei Ausprägungen der Protokollierung, das „Logging“ und das „Tracing“. Bei beiden werden Funktions- oder Methodenaufrufe in das Programm aufgenommen, die zur Laufzeit über den Status des Programms Auskunft geben.

In der Praxis gibt es zwischen Logging und Tracing allerdings sehr wohl Unterschiede:

- Logging kann fachliche oder technische Protokollierung sein, oder eine beliebige Kombination von beidem.
  - Fachliche Protokolle werden gewöhnlich anwenderspezifisch aufbereitet und übersetzt. Sie dienen Endbenutzern, Administratoren oder Betreibern von Softwaresystemen und liefern Informationen über die vom Programm abgewickelten Geschäftsprozesse.
  - Technische Protokolle sind Informationen für Betreiber oder Entwickler. Sie dienen der Fehlersuche sowie der Systemoptimierung.
- Tracing soll *Debugging*-Information für Entwickler oder Supportmitarbeiter liefern. Es dient primär zur Fehlersuche und -analyse.

7.9.1 Typische Probleme

Hohe Abhängigkeit von der Protokollierung

Ein Problem einfacher *ad-hoc*-Lösungen der Protokollierung ist die starke Abhängigkeit nahezu sämtlicher Komponenten von der Protokollierung. In fast allen anderen Komponenten wird von der Protokollierung explizit Gebrauch gemacht, indem Aufrufe der Form

```
writeLogEntry( message, priority, category );
```

enthalten sind. Der Empfänger solcher Nachrichten (oder Funktionsaufrufe) muss daher in jeder Komponente bekannt sein. Spätere Erweiterungen der Protokollierung, insbesondere ihrer Schnittstelle, ist mit Quellcode-Anpassungen an vielen Stellen verbunden.

Von daher ist es im Rahmen der präventiven Qualitätssicherung angebracht, vorhandene Bibliotheken zur Protokollierung heranzuziehen, die flexible Anpassung an erweiterte Nutzungsbedürfnisse ohne Code-Änderungen erlauben. Hierzu gehören die Konfiguration zur Laufzeit (Ein- und Ausschalten der Protokollierung, Umlenkung auf andere Ausgabekanäle) und die Definition neuer Kategorien, Prioritäten oder Ausgabekanäle.

Hohe Flexibilität gefordert

Die für die Protokollierung verantwortliche Softwarekomponente, nachfolgend kurz *Logger* genannt, benötigt Flexibilität in folgenden Aspekten:

- Welche Ereignisse sollen protokolliert werden? Hierzu werden Ereignisse in Kategorien eingeteilt. Der Logger entscheidet zur Laufzeit darüber, ob ein auftretendes Ereignis protokolliert wird oder nicht.
- Wie wichtig oder gravierend sind Ereignisse einer bestimmten Kategorie? Dazu können Kategorien mit Prioritäten versehen werden, etwa `DEBUG`, `INFO`, `WARNING`, `ERROR`, und `FATAL`.
- Wohin sollen Ereignisse protokolliert werden? Je nach Anforderung können dazu beliebige Ausgabekanäle sinnvoll oder notwendig sein: Neben den konventionellen Logdateien können Ereignisse an Betriebssystemdämonen[49] oder Warteschlangen geschickt werden, oder sogar über Mobilfunk (SMS) oder E-Mail direkt an die verantwortlichen Personen.
- Wie sollen Ereignisse einer bestimmten Kategorie formatiert werden? Für bestimmte Ereigniskategorien mag eine Übersetzung in die Landessprache der Endanwender sinnvoll sein. Benötigt das System die Nachrichten als Text, HTML, XML oder in anderen Formaten?

Protokollierung in heterogenen Systemen

Protokollierung über Programmiersprachen- und Betriebssystemgrenzen ist komplex. Es gibt einige technische Probleme (etwa: Netzbelastung, Parameterübergabe zwischen unterschiedlichen Programmiersprachen).

Bei getrennter Protokollierung auf Clients und Server ist die Konsolidierung von Client- und Serverprotokollen schwierig.

7.9.2 Lösungskonzept

Die Apache Foundation hat den *Logging Framework for Java* (log4j) entwickelt, eine freie Bibliothek zur Protokollierung in Java.[50] Die Konzepte dieser Bibliothek erfüllen die oben genannten Anforderungen an Logging und Tracing. Die Konzepte dieses Ansatzes sind leicht auf andere Sprachen übertragbar.

[49] Etwa den Unix `syslogd` oder entsprechende Mechanismen unter anderen Betriebssystemen.
[50] Mittlerweile gibt es auch Portierungen für C++ und Python.

Einige Tipps zur Protokollierung:

- Nutzen Sie möglichst ein bestehendes Framework. Dieses Rad müssen Sie nicht neu erfinden!
- Protokollieren Sie sämtliche Interaktionen der Benutzeroberfläche, weil damit der „Gang“ des Benutzers durch die Applikation nachvollziehbar wird.
- Protokollieren Sie zusätzlich alle Interaktionen zwischen Subsystemen oder Komponenten.
- Nutzen Sie die Protokollierung zur Fehlersuche und -analyse. Durch Protokollaufrufe instrumentierter Code erleichtert die Fehlersuche.
- Beachten Sie die entstehende Netzbelastung, wenn Sie in verteilten Systemen Protokolle schreiben. Lösungsansatz: Getrennte Protokolle auf den einzelnen Plattformen. Nachteilig hierbei ist, dass der Zugriff auf die Client-Protokolle eventuell (rein organisatorisch) schwer sein kann, etwa wenn die Clients im Internet verteilt sind.
- Wenn die kontinuierliche Fortschreibung einer Protokolldatei aus Platz- oder Performancegründen nicht möglich ist, können Sie Ringpuffer als Alternative einsetzen. Dabei steht eine feste Anzahl von Einträgen im Protokoll zur Verfügung, die zyklisch überschrieben werden. Siehe dazu auch [Marick2000].

7.9.3 Zusammenhang mit anderen Aspekten

- Sicherheit: Die korrekte Behandlung sensibler Daten in Protokolldateien muss gewährleistet werden.
- Verteilung: Auf welchem physischen Knoten läuft der Logger? Gibt es eine oder mehrere Instanzen?
- Kommunikation: Logging in verteilten Systemen erzeugt Netzlast.

7.9.4 Weiterführende Literatur

Einige Open-Source-Projekte erstellen Bibliotheken für die Protokollierung. Zu den bekannten gehört der log4j Framework des Apache-Projektes, zu finden unter *www.apache.org*. Einen Wrapper zur Kapselung unterschiedlicher Logging-Frameworks bietet Apache im Commons-Projekt. Ab Version 1.4 enthält Java eine Standard-API für Logging.

[Gupta2003] diskutiert Logging mit Java und log4j ausführlich, mit vielen Benutzungshinweisen.

[Marick2000] beschreibt einige Muster von Logging mit Ringpuffern, die sich zum dezentralen Einsatz eignen, also etwa auf Arbeitsplatzrechnern mit eingeschränktem Speicherplatz.

7.10 Ausnahme- und Fehlerbehandlung

Errors are serious problems.

Tim McCune, aus „Exception Handling Antipatterns“

7.10.1 Motivation

Kennen Sie Murphys Regel? Es geht immer etwas schief. Und das grundsätzlich im unpassenden Augenblick. Reifen platzen meistens im strömenden Regen, und Sie bekommen genau dann Zahnschmerzen, wenn Ihre Lieblingszahnärztin gerade ihren wohlverdienten Urlaub genießt.

Murphys Regel

Software stürzt bevorzugt während wichtiger Präsentationen ab – und dann begräbt sie gleich noch diverse andere Programme mit im virtuellen Nirwana.[51]

Demgegenüber erwarten Anwender und Administratoren zu Recht, dass Software in Fehler- und Ausnahmesituationen:

- vorhersehbar und angemessen reagiert und Anwender nicht im „Regen“ stehen lässt;
- möglichst keine Auswirkungen auf andere Programme hat, insbesondere BSOD[52] und ähnliche Effekte vermeidet;
- die Diagnose des Fehlers erleichtert.

Ausnahme- und Fehlerbehandlung dreht sich um folgende Teilaufgaben:

- Ausnahmen und Fehler erkennen. In diesem Fall sollten Sie die folgenden drei Fragen[53] beantworten können:
  1. Was ging schief?
  2. Wo ist es passiert?
  3. Warum ging es schief?
- Auf Fehler angemessen reagieren. Bei Bedarf Benutzer oder Administratoren benachrichtigen und passende korrektive Maßnahmen ergreifen.
- Fehlersituationen vorzeitig erkennen und präventiv reagieren. Für *fehlertolerante* Systeme wichtig – beispielsweise den freien Speicherplatz kontinuierlich messen und bei Speicherknappheit gründlich aufräumen.

[51] Kenner von Wahrscheinlichkeitsrechnung und Stochastik mögen schmunzeln. Ich zumindest kenne Murphy zur Genüge.

[52] Blue Screen Of Death. Frühere Versionen eines bekannten (und mittlerweile sehr robusten) Betriebssystems zeigten BSOD im Falle fataler Fehler.

[53] Nach Jim Cushing, *http://today.java.net/pub/a/today/2003/12/04/exceptions.html*

Fehlerbehandlung ist kein (reines) Programmierproblem

Immer wieder stoße ich in Projekten auf die Meinung „Fehlerbehandlung übernimmt doch schon die Programmiersprache“ – zumindest bei solchen Entwicklern, deren favorisierte Programmiersprache ein Exception-Konzept mitbringt.

Aus meiner Sicht weit gefehlt: Vor der Realisierung irgendeiner Ausnahme- oder Fehlerbehandlung steht der grundlegende und strukturelle Entwurf eines entsprechenden Konzeptes, das folgende Fragen klärt:

- Welche Ausnahmen und Fehlerkategorien gibt es überhaupt?
  - Welche externen Ereignisse können zu Ausnahmen oder Fehlern im Programm führen? Beispiele: Falsche oder fehlende Eingabedaten, fehlerhaft bediente Eingabeschnittstellen, Fehler in der genutzten Infrastruktur. Wie erkennen Sie solche Situationen?
  - Welche internen Zustände gelten als Ausnahme oder Fehler? Wie erkennen Sie diese?
- Auf welche dieser Kategorien soll Ihr System in welcher Form reagieren?
  - Gibt es Ausnahmen oder Fehler, die das System selbstständig oder mit Nachfragen korrigieren muss?
  - Welche Fehlersituationen muss das Programm an welche Adressaten melden? Diese Frage steht im Zusammenhang mit dem Aspekt „Protokollierung und Logging“ (siehe Abschnitt 7.11).
- Wie können Sie die Behandlung von Ausnahmen und Fehlern angemessen (d. h. mit einem zur Domäne passenden Verhältnis aus Testaufwand und Testabdeckung) testen? Dazu müssen Sie Ausnahmen und Fehler reproduzierbar herstellen, was oft ein erhebliches Problem darstellt.

Allerdings läuft es nach diesen konzeptionellen Aufgaben auf eine gründliche und einheitliche Programmierung hinaus – was Programmiersprachen mit Exception-Konstrukten erheblich erleichtern.

Fehler sind Abweichungen von Annahmen

Ein *Fehler* liegt vor, wenn ein Unterschied zwischen einem erwarteten und einem konkret vorliegenden Ergebnis oder Zustand entsteht. Beispiele:

- Ihr System erwartet an einer Eingabeschnittstelle XML, erhält aber pdf.
- Klainer Fähler. Sie erwarten (berechtigterweise) die korrekte Schreibweise.
- Ihr System erwartet Eingabedaten über eine Web-Service-Schnittstelle, erhält dort aber keine (sondern Sie erhalten per Post eine CD).
- Ihr System sucht aus einer Datenbank Bestelldaten anhand von Kundennummern heraus. Während der Suche fährt der Operator die Datenbank herunter.
- Ihr Webserver erhält zu viele Anfragen und kann nicht mehr schnell genug antworten.
- Ein Baustein Ihres Systems empfängt ein Ereignis, auf das er nicht vorbereitet ist (etwa: ein Interrupt-Signal).
- Ein Baustein Ihres Systems kommt während der Verarbeitung in einen inkonsistenten Zustand, führt eine Division-durch-Null aus, arbeitet auf einem nicht initialisierten Objekt, überschreitet die maximale Puffergröße, erzeugt einen Speicherüberlauf oder benimmt sich auf andere Weise daneben.

Sie sehen – die Fantasie des Fehlerteufels kennt kaum Grenzen.

Als *Ausnahme* bezeichne ich eine Situation, in der ein Programmbaustein aufgrund eines *Fehlers* seinen „normalen“ Ablauf nicht fortführen kann.[54]

Architekten kümmern sich um Laufzeitfehler

Uns Softwarearchitekten interessieren hauptsächlich Fehler, die zur Laufzeit von Programmen auftreten. Compiler tragen zur Vermeidung mancher Fehler bei. Leider verschwimmt jedoch in manchen Fällen die Grenze zwischen Compile- und Laufzeit:

- In interpretierten Sprachen findet eventuell keine Syntaxprüfung zur Compilezeit statt (etwa in Shellskripten, Ruby oder Perl).
- Manche (compilierten) Systeme enthalten Bestandteile, die erst zur Laufzeit interpretiert werden, beispielsweise Metadaten, Konfigurationsdaten oder interpretierte Geschäftsregeln. Das sind beliebte Mittel, um die Flexibilität von Systemen zu erhöhen und gleichzeitig die vom Compiler gewährte Sicherheit zu reduzieren!

Sie erkennen jetzt, dass wir als Softwarearchitekten auch *falch gechribene* Worte ins Kalkül unserer Fehlerbehandlung aufnehmen müssen, zumindest dann, wenn uns die Compiler mal wieder im Stich lassen. Die Theorie „Der Compiler wird’s schon finden“ ist eben nur eine Theorie.

7.10.2 Fehlerkategorien schaffen Klarheit

Meine vordringliche Empfehlung bezüglich Ausnahme- und Fehlerbehandlung lautet, die Arten oder Kategorien der möglichen Fehler möglichst umfassend zu ermitteln. Im Idealfall können Sie dazu auf die Vorarbeit Ihrer Systemanalytiker zurückgreifen, in der Realität bleibt diese Arbeit komplett an Ihnen hängen. Anschließend definieren Sie, wie Ihr System mit diesen Kategorien umgehen soll, und erstellen daraus konkrete Vorgaben zur Implementierung der Fehlerbehandlung.

Ich schlage Ihnen mehrere Kategorisierungen vor, ohne Anspruch auf Vollständigkeit. Mithilfe dieser Kategorien können Sie einfacher beschreiben, ob und wie Ihr System in solchen Fällen reagieren soll.

Sie werden merken, dass ein einzelner Fehler durchaus zu mehreren Kategorien gehören kann.

Fachliche und technische Fehler

Fachliche Fehler resultieren aus Problemen in den fachlichen Abläufen oder entsprechenden Daten von Systemen. Beispiel: „Kein Vertrag zu diesem Kunden vorhanden.“ Solche Fehler können Sie mit genügend Verständnis der Fachdomäne recht gut charakterisieren. Insbesondere sollte Ihr System zu *erwarteten* fachlichen Fehlern eine für Anwender plausible Abhilfe anbieten.

[54] In manchen (Java-)Programmen habe ich Exceptions gefunden, die völlig normale („fachliche“) Ereignisse abbildeten. Ich persönlich halte diese Interpretation von Ausnahmen für schlecht, weil damit der „Fehlercharakter“ von Ausnahmen verschwimmt.

Demgegenüber resultieren technische Fehler oft aus Problemen der zugrunde liegenden Infrastruktur („Datenbankverbindung verloren“) oder reinen Programmierfehlern („Division durch Null“ oder NPE, Null Point Exception). Sie können jedoch auch in fachlichen Abläufen oder Bausteinen auftreten.

Technische Fehler sollten Sie nicht in fachlichen Bausteinen behandeln – das bläht deren Code unnötig auf. Dennoch müssen Sie auf technische Fehler in fachlichen Bausteinen reagieren!

Syntax-, Format- oder Protokollfehler

Syntaxfehler: parsen

Daten oder übermittelte Dokumente können syntaktische Fehler aufweisen und für Ihr System dann unbrauchbar sein (etwa wenn ein lieferndes System die Syntax der übertragenen Auftragsnummern ohne Absprache geändert hat ...). Syntaxfehler kann Ihr System durch *Parsen* feststellen.

Formatfehler: interpretieren

Trotz korrekter Syntax können Daten für Ihr System unverständlich sein, beispielsweise den vereinbarten Wertebereich überschreiten oder ungültige Werte für Datenbankschlüssel darstellen. Solche Formatfehler (oder semantische Fehler) kann Ihr System nur durch *Interpretieren* diagnostizieren. Sie führen oft zu semantischen Problemen, sofern sie nicht frühzeitig erkannt werden. In diese Kategorie fallen Verletzungen von Kann/Muss-Bedingungen oder die sogenannten „Plausibilisierungsfehler“ an Benutzeroberflächen.

Gegen Syntax- und Formatfehler können Sie sich durch klar definierte Schnittstellenverträge und entsprechende Prüfungen (assertions) an Ihren Eingangsschnittstellen wappnen. Zusätzlich können Unit-Tests überprüfen, ob Ihre Bausteine selbst solche Fehler verursachen.

Nehmen wir jetzt an, das datenliefernde Fremdsystem hat syntaktisch korrekte Daten in den richtigen Formaten geliefert, leider jedoch zum falschen Zeitpunkt an den falschen Ort oder über einen falschen Kanal. Dann haben wir es mit einem *Protokollfehler* zu tun. Einer der Partner hat das vereinbarte *Protokoll* einer Schnittstelle verletzt und dadurch einen Fehler verursacht. Siehe dazu auch „Exakte Verträge“ in Abschnitt 7.12.3.

Meiner Erfahrung nach wird die Kategorie „Protokollfehler“ häufig vergessen! Sie fallen häufig erst im operativen Betrieb auf und lassen sich oftmals nur manuell erkennen und reparieren.

Korrigierbare und unkorrigierbare Fehler

Eine fehlerhafte manuelle Eingabe kann Ihr System mit einer entsprechenden Meldung beim Benutzer reklamieren und eine Korrektur einfordern. Diese Kategorie von Fehlern nenne ich *korrigierbar*.

Bei Schnittstellen von und zu Fremdsystemen ist eine Korrektur oftmals schwieriger, weil sie nicht interaktiv eingefordert werden kann (es sei denn, die entsprechende Schnittstelle sieht eine solche Korrektur vor!). Viele technische Fehler sind unkorrigierbar: Ist eine notwendige Datenbankverbindung nicht vorhanden, kann Ihr System nicht speichern – eben unkorrigierbar.[55]

Solche unkorrigierbaren Fehler kann Ihr System nicht selbstständig beheben. Sie werden zu (fachlichen) *Clearing*-Fällen oder aber sorgen für den Abbruch der jeweiligen Aktion oder eines Prozesses. Möglicherweise muss Ihr System im Falle unkorrigierbarer Fehler jedoch noch „Aufräumarbeiten“ in Form von Rollbacks oder Kompensationen durchführen, um das Gesamtsystem wieder in einen konsistenten Zustand zu bringen.

Tipps zur Ermittlung von Fehlerkategorien

- Stimmen Sie Ihre Ausnahme- und Fehlerkategorien mit erfahrenen Testern oder Ihrer Qualitätssicherung ab. Den QS-Experten fallen garantiert noch diverse Fehler ein, an die Sie (oder Ihre Entwickler) bisher nicht gedacht haben.
- Achten Sie insbesondere auf Schnittstellen: An Stellen, wo Daten Systemgrenzen überschreiten, entstehen oftmals *semantische Lücken*. Sie dürfen auch *Problemzonen* dazu sagen.
- Besonders kritisch sind die Schnittstellen zu Fremdsystemen, also die externen Schnittstellen Ihres Systems.

7.10.3 Muster zur Fehlerbehandlung

Exakte Verträge mit externen Systemen

Exakte Absprachen

Treffen Sie mit externen Systemen möglichst exakte Absprachen hinsichtlich der auszutauschenden Daten. Beachten Sie dabei neben der reinen Syntax und Semantik der ausgetauschten Daten auch Formate (Zeichen- und Zahlencodierung, Schlüsselwerte etc.), Muss-/Kann-Belegungen, Wertebereiche, Ausnahme- oder Sonderfälle, Grenzfälle (Nullwerte, leere Dateien, überlange Zeichenketten etc.), Übertragungskanäle und -protokolle, Häufigkeit des Datenaustausches, eventuelle Berechtigungs- oder Sicherheitsaspekte, Transaktions- und Zustandsbehandlung, eventuell notwendige Transformation, Synchronität, Statusverfolgung und andere. Kann das liefernde System bereits gelieferte Daten nachträglich für ungültig erklären und durch eine neue Version ersetzen?

Sie sehen – solche Details können Schnittstellenverträge deutlich aufblähen, meiner Erfahrung nach ist jedoch nur eine solche systemübergreifende Fehlerbehandlung realistisch.

[55] Aber: Mit entsprechendem Aufwand könnten Sie auch solche Fehler korrigieren – sofern das für Ihre Auftraggeber und Stakeholder wichtig genug ist.

Mein persönlicher Tipp: Skizzieren Sie jede (wirklich jede einzelne) externe Schnittstelle aus Laufzeitsicht mit ihrem kompletten *Protokoll*: Wer stößt wie eine Übertragung vom/ins System an? Gibt es nur Nettodaten oder auch eine Quittung? Zeichnen Sie ein paar Sequenzdiagramme für den Sonnenschein- und einige Fehlerfälle.

Verstecke technische Fehler vor Anwendern

Meldungen zu technischen Fehlern sind meist nur für Administratoren oder Betreiber relevant und fast nie für Endanwender von Softwaresystemen.[56]

Stellen Sie zur Behandlung technischer Fehler eine einheitliche Schnittstelle bereit, die folgende Dinge leistet:

- Technische Fehler gehören praktisch immer in das zentrale Logfile, wo sie den Administratoren oder Entwicklern bei der Fehlerdiagnose helfen. Hier sollten Sie sämtliche relevanten Details protokollieren.
- Parallel dazu zeigen Sie Endanwendern eine benutzerfreundliche und allgemein verständliche Meldung. Erläutern Sie, dass etwas Gravierendes geschehen ist, das aber nichts mit den Aktionen der Anwender zu tun hat. Fügen Sie dieser Meldung eine eindeutige Fehlerkennung hinzu, auf die sich Anwender gegenüber Administratoren berufen können.
- Sofern angemessen, sollte eine automatisierte Meldung an den Support (Administrator, Helpdesk, Call-Center oder Ähnliches) versandt werden.

Optimiere Diagnostizierbarkeit

Sie haben sicherlich schon erlebt, dass die Suche nach einem Fehler tagelang dauerte und dessen Behebung nur wenige Minuten. Daher sollte ein übergreifendes Entwurfs- und Implementierungsziel sein, Fehler und deren Ursachen möglichst einfach und schnell diagnostizieren zu können.

Dafür benötigen Sie einerseits ein einheitliches Konzept zur Ausnahme- und Fehlerbehandlung, andererseits eine Menge Disziplin und Energie bei dessen Umsetzung und Überprüfung!

Prüfen Sie möglichst viele dieser Vorgaben automatisiert durch Codeanalyse.

Siehe auch die Hinweise zu „Original- und Folgefehler“ im nachfolgenden Abschnitt.

7.10.4 Mögliche Probleme

Verteilte Verarbeitung

Falls ein Fehler in einem entfernten („remote“) Baustein auftritt, müssen Sie weitere Informationen zu diesem Fehler eventuell erst über ein Netzwerk übermitteln. Das kann in manchen Fällen zu lange dauern (oder aufgrund technischer Fehler nicht funktionieren). Dann werden die Protokollierung oder das Logging solcher Fehler problematisch. [Longshaw+04] gibt den (guten!) Rat: „Log at Distribution Boundary“ (Protokolliere den Fehler dort, wo er auftritt).

[56] Schlimmer noch: Da Endanwender solche Meldungen häufig nicht verstehen, sinkt möglicherweise ihr Vertrauen in das System.

Leite über Verteilungsgrenzen lediglich allgemeine Fehlermeldungen weiter, und vermeide es, umfangreiche Fehlerprotokolle quer durch verteilte Systeme zu senden.

Dieses Problem taucht bereits bei einfachen Client/Server-Systemen auf. Dort müssen manche Meldungen den Anwender an der Benutzeroberfläche (also im Client) erreichen, andere das zentrale Logfile und manche beide Stellen.

Original- und Folgefehler schwer unterscheidbar

Klingt trivial: Unterscheiden Sie zwischen der Ursache (dem Originalfehler) und seinen Folgefehlern. Nur so behalten Sie den Überblick darüber, in welchen Bausteinen wirkliche Fehler auftreten (und können dann auch das wahre Problem nachhaltig beheben).

Leider spielt uns die Praxis in diesem Falle schon wieder übel mit: Oftmals versteckt sich das ursprüngliche Problem, insbesondere im Falle *unerwarteter* Fehler. Es kann sehr schwierig sein, Originalfehler automatisch zu ermitteln – da hilft oft nur Erfahrung, Geduld und Forensik („Spurensuche").

In Sprachen mit Exception-Handling tritt dieses Problem oft auf, wenn Exceptions nach ihrem Auftreten „verschluckt" oder „umgewandelt" werden. Vermeiden Sie leere Catch-Klauseln oder häufige Umwandlung von Exception-Typen innerhalb des Quellcodes.[57]

Ein Tipp aus [Longshaw+04]: Geben Sie jedem Fehler ein eindeutiges Fehlerkennzeichen (*unique ID*). Das kann in verteilten Logfiles die Ursachenforschung stark vereinfachen.

7.10.5 Zusammenhang mit anderen Aspekten

Protokollierung und Logging

- Geben Sie als Softwarearchitekt dem Entwicklungsteam klare Vorgaben hinsichtlich des Loggings von Ausnahmen und Fehlern: Alle Fehler sollten in einem Logfile landen, möglicherweise sogar in mehreren (eine ausführliche Meldung in einem lokalen Logfile, eine verkürzte Version in einem zentralen). Es muss nach einem Fehler immer möglich sein, Ursache, Ort und Auswirkung von Fehlern exakt zu bestimmen! *Log at distribution boundaries*.
- Ihre Fehlerkategorien können Entwicklern als Anhaltspunkte für Logging dienen.
- Falls Ihre bevorzugte oder vorgeschriebene Programmiersprache keine Exceptions kennt oder Laufzeitfehler nur schwer erkennen kann: Eventuell können Sie zumindest Teile Ihrer Systeme in Sprachen wie C# oder Java erstellen, die mit technischen und Laufzeitfehlern gut umzugehen wissen (sprich: ausgereifte Exception-Konzepte bereitstellen!).
- Sowohl Logging wie auch Exception-Handling können Sie mit aspektorientierter Entwicklung (*aspect-oriented programming*, AOP) zentralisieren. Mehr dazu in [Laddad03] oder [AspectJ].

[57] Java-Projekten empfehle ich dafür Tools wie PMD (pmd.sourceforge.net), CheckStyle (checkstyle.sourceforge.net) oder ähnliche Codeanalyse-Werkzeuge, um solche typischen Probleme bereits im Build zu identifizieren!

Internationalisierung

Falls Ihre Software in mehreren Ländern zum Einsatz kommt, müssen Sie Meldungen über Ausnahmen und Fehler internationalisieren, d. h. die entsprechenden Meldungstexte in den adäquaten Ländereinstellungen des jeweiligen Benutzers ausgeben. Siehe Abschnitt 7.8.

7.10.6 Weiterführende Literatur

Es sieht ziemlich düster aus mit Literatur zu diesem (wichtigen!) Thema. Sie finden mit etwas Mühe diverse wissenschaftliche Traktakte teilweise mit einem mathematisch-statistischen oder auch historischen Fokus (etwa Sammlungen bekannter Software-Fehler, wie z. B. *http://www5.in.tum.de/~huckle/bugse.html*).

Sehr nützlich fand ich die Pattern-Sammlung von Andy Longshaw und Eoin Woods ([Longshaw+04]). Darin zeigen die Autoren sieben zentrale Muster zur Behandlung von Fehlern auf.

Zum Thema Ausnahmebehandlung auf Programmierebene (Exception-Handling) hat Tim McCune einige lesenswerte Antipatterns zusammengefasst, online unter: *http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html*

Einige Hinweise zum „ordentlichen“ Exception-Handling (nicht nur) für Java zeigt *http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html*. Gunjan Doshi erläutert Kriterien, nach denen Sie eine Schnittstelle mit Exceptions entwerfen sollten.

8 Bewertung von Softwarearchitekturen

To measure is to know.

James Clerk Maxwell[1]

Fragen, die dieses Kapitel beantwortet:

- Warum sollten Sie Architekturen und Code bewerten?
- Was können Sie in der IT überhaupt bewerten?
- Was unterscheidet qualitative von quantitativer Bewertung?
- Wie gehen Sie bei der Bewertung von Architekturen vor?
- Welche Stakeholder sollten bei der Bewertung mitwirken?
- Wie setzen Sie Softwaremetriken sinnvoll ein?

Bewerten Sie schon, oder raten Sie noch?

Bewertung unterstützt Steuerbarkeit

„*You cannot control what you cannot measure*“, sagt Tom DeMarco treffend. Positiv formuliert, bedeutet diese Aussage für IT-Projekte und Systementwicklung: Wenn Sie kontrolliert auf Ihre Projektziele zusteuern möchten, müssen Sie regelmäßig Ihre Zielerreichung bewerten, um bei Bedarf steuernd eingreifen und korrigierende Maßnahmen einleiten zu können.

Während der Systementwicklung sollten Sie regelmäßig die Frage „Gut genug?“ stellen und beantworten. In Kapitel 3 haben Sie die wesentlichen Aufgaben in der Architekturentwicklung kennengelernt – darin kommt die (regelmäßige) Bewertung vor.

Bild 8.1 zeigt im Überblick, dass Bewertung auf alle übrigen Architektenaufgaben starken Einfluss nehmen kann. Auf Basis systematischer Bewertung können Sie als Softwarearchitekt schlichtweg Ihre eigene Arbeit besser planen und Ihre Ergebnisse verbessern!

In der Praxis finden Bewertungen (unter dem Titel „Audit“ oder „Review“) oftmals in späten Entwicklungsphasen statt, wenn das betroffene System bereits im produktiven Einsatz läuft. Viel zu selten investieren laufende Projekte in die Bewertung – obwohl das den meisten Projekten/Systemen guttäte.

[1] *http://de.wikipedia.org/wiki/James_Clerk_Maxwell*: Formulierte die Maxwellgleichungen, Grundlagen der Berechnung elektromagnetischer Wellen.

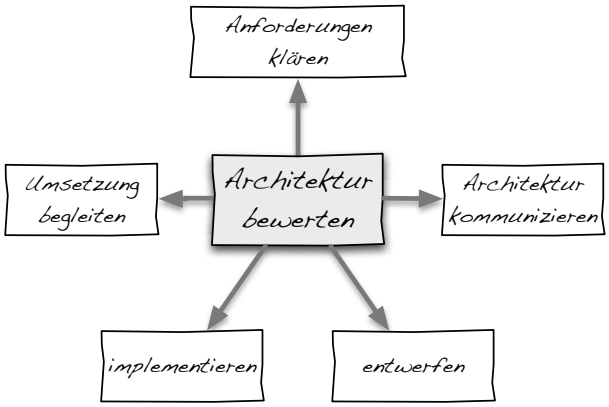


BILD 8.1
Architekturbewertung als relevante Aufgabe

Was Sie in der IT bewerten können – und wie

Sie können in Software-Projekten zwei Arten von „Dingen“ bewerten:

- Prozesse, wie etwa Entwicklungs- oder Betriebsprozesse: Hierbei können Sie organisatorische Aspekte betrachten oder aber den Einsatz von Ressourcen bewerten. Leider können Sie auf Basis der Prozesse kaum Aussagen über die Qualität der entwickelten Systeme treffen.[2] Dieses Thema werde ich hier nicht weiter vertiefen.
- Artefakte, wie beispielsweise Anforderungen, Architekturen, Quellcode und andere Dokumente. Einige Arten dieser Artefakte (wie beispielsweise Quellcode) können Sie quantitativ, das heißt in Zahlen, bewerten. Andere (wie etwa die Softwarearchitektur) entzieht sich der rein zahlenmäßigen Bewertung. Diese Gruppe von Artefakten bewerten Sie qualitativ, also ihrer Beschaffenheit oder Güte nach. Zur letzten Gruppe gehören Softwarearchitekturen, deren Bewertung *qualitativ* erfolgt.

Zunächst möchte ich Ihnen erläutern, welche Ergebnisse Sie bei Bewertungen grundsätzlich erreichen können. Anschließend lernen Sie (in Abschnitt 8.1) qualitative Architekturbewertung kennen, angelehnt an die verbreitete ATAM-Methode.

Schließlich finden Sie in Abschnitt 8.2 einige Grundlagen quantitativer Bewertung und Metriken.

Bewertung als Soll-Ist-Vergleich

Betrachten Sie qualitative Bewertung als einen Soll-Ist-Vergleich:

- Das Soll beschreibt die Qualitätsanforderungen beziehungsweise Architekturziele. Es charakterisiert die gewünschten Eigenschaften eines Systems.
- Sie vergleichen gegen das (aktuelle oder geplante) Ist-System, möglicherweise auf Basis von Dokumentation, Plänen oder Modellen. Dabei gehen Sie freingranular vor, d. h. Sie vergleichen einzelne Anforderungen des „Soll“ mit ihren Entsprechungen im „Ist“.

[2] Sie können jedoch durch gezielte Reflexionen oder Projekt-Retrospektiven signifikante Verbesserungen erreichen, die zumindest mit hoher Wahrscheinlichkeit positive Auswirkungen auf die Qualität von Systemen haben.

Lassen Sie mich ein fachfremdes Beispiel heranziehen – siehe die nachfolgende Tabelle: Ich möchte qualitativ meinen morgendlichen Grüntee bewerten:

TABELLE 8.1 Beispiel für Soll-Ist-Vergleich

Soll	Ist	Ergebnis des Soll-Ist-Vergleichs
Hellgrüne Farbe	Hellbraune Farbe	Anforderung nicht erfüllt.
Typischer Sencha-Geschmack	Geschmack nach Kirsche und Mandel	Gewünschter Geschmack nicht gegeben, dafür aber Preis-Anforderung besonders gut erfüllt.
Grasiger Geruch	Geruchlos	Problem: Tee erfüllt Geruchsanforderung nicht.
Ungezuckert		Wegen des Kirscharomas nicht prüfbar.
Preis < 9 €/100 g	5 €/100 g	Anforderung (über-) erfüllt.
Serviert in dünnwandiger Tasse	Wandstärke der Tasse < 1,5 mm	Anforderung voll erfüllt, ok.

Bei dieser Art Vergleich können Sie grundsätzlich drei verschiedene Ergebnisse erzielen (siehe auch Bild 8.2):

1. Soll stimmt mit Ist überein – die Architektur bzw. deren Umsetzung ist bezüglich dieser spezifischen Eigenschaft ok.
2. Soll stimmt in mehreren Qualitätsanforderungen nur teilweise mit Ist überein – es wurden Kompromisse geschlossen (einzelne Merkmale wurden verbessert auf Kosten anderer).
3. Einzelne Qualitätsanforderungen können im Ist nicht erreicht werden (Risiko) oder werden nicht erreicht (Problem, eingetretenes Risiko).

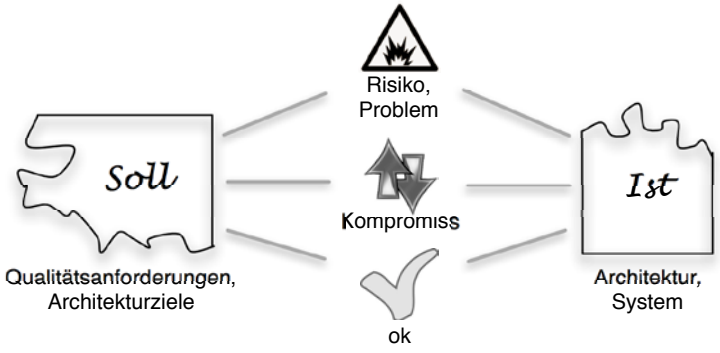


BILD 8.2 Bewertung als Soll-Ist-Vergleich

Für qualitative Bewertung ist die Granularität der Anforderungen entscheidend: Sie benötigen einen detaillierten und möglichst konkreten Katalog an Qualitätsanforderungen und Architekturzielen!

Als Vorbild könnten Sie die bekannten Produktbewertungen der „Stiftung Warentest“ heranziehen, die oftmals sehr detaillierte Kriterienkataloge (=Anforderungen) als Basis ihrer Bewertungen verwenden.

8.1 Qualitative Architekturbewertung

Bewertung von Qualitätsmerkmalen

Bewertung von Softwarearchitekturen liefert qualitative Aussagen: Sie bewerten Architekturen danach, inwieweit sie die Erreichung bestimmter Qualitätsanforderungen ermöglichen oder behindern.

Beispielsweise treffen Sie Aussagen darüber, in welchem Maß das jeweilige System den Anforderungen an Wartbarkeit, Flexibilität, Performance oder Sicherheit genügt.

Dabei liefert eine Architekturbewertung kein absolutes Maß, d. h., sie bewertet nicht die „Güte-über-alles", sondern nur im Hinblick auf spezifische Kriterien. Architekturbewertung hilft, Risiken zu identifizieren, die sich möglicherweise aus problematischen Entwurfsentscheidungen ergeben.

Bewertungsgrundlagen für Softwarearchitekturen

In Kapitel 3 (Vorgehen bei der Architekturentwicklung) haben Sie wichtige Qualitätsmerkmale von Software-Systemen kennengelernt. Ich habe Ihnen gezeigt, wie Sie mit Hilfe von Szenarien diese Qualitätsmerkmale beschreiben und operationalisieren können.

Mit Hilfe von Szenarien können Sie spezifische Messgrößen für Ihre Systeme definieren oder, noch besser, von den maßgeblichen Stakeholdern definieren lassen.

Qualität ist spezifisch für Stakeholder

Dabei gilt es zu ermitteln, welche spezifischen Kriterien die Architektur zu erfüllen hat. Anhand dieser Kriterien können Sie dann die „Güte" oder „Eignung" der Architektur ermitteln.

Eine Architektur ist in diesem Sinne geeignet, wenn sie folgende Kriterien erfüllt:

- Das System (das mit dieser Architektur entwickelt wird oder wurde) erfüllt sowohl seine funktionalen als auch nichtfunktionalen Anforderungen (Qualitätskriterien).
- Insbesondere erfüllt das System die spezifischen Anforderungen an Flexibilität, Änderbarkeit und Performanceverhalten.
- Das System kann mit den zur Verfügung stehenden Ressourcen realisiert werden. Sowohl das Budget wie auch der Zeitplan, das Team, die beteiligten Fremd- und Legacy-Systeme, die vorhandene Basistechnologie und Infrastruktur sowie die gesamte Organisationsstruktur beeinflussen die Umsetzbarkeit einer Architektur.

Zur Bewertung einer Architektur ist es daher von entscheidender Bedeutung, die spezifischen Qualitätskriterien des betroffenen Systems zu kennen. In der Praxis werden diese Kriterien oder Qualitätsziele oft erst bei einer Architekturbewertung expliziert.

Bereits in Kapitel 3 habe ich Ihnen empfohlen, die Qualitätsanforderungen an Ihre Systeme explizit durch Szenarien zu dokumentieren. Diese Szenarien bilden eine wertvolle Orientierungshilfe für weitere Entwicklungsschritte und eine ausgezeichnete Basis für die Architekturbewertung!

Qualitative Bewertung nach ATAM

Ich möchte Ihnen im Folgenden ein methodisches Vorgehen zur Architekturbewertung vorstellen, orientiert an der ATAM[3]-Methode von [Bass+03] und [Clements+03]. Bild 8.3 gibt einen Überblick.

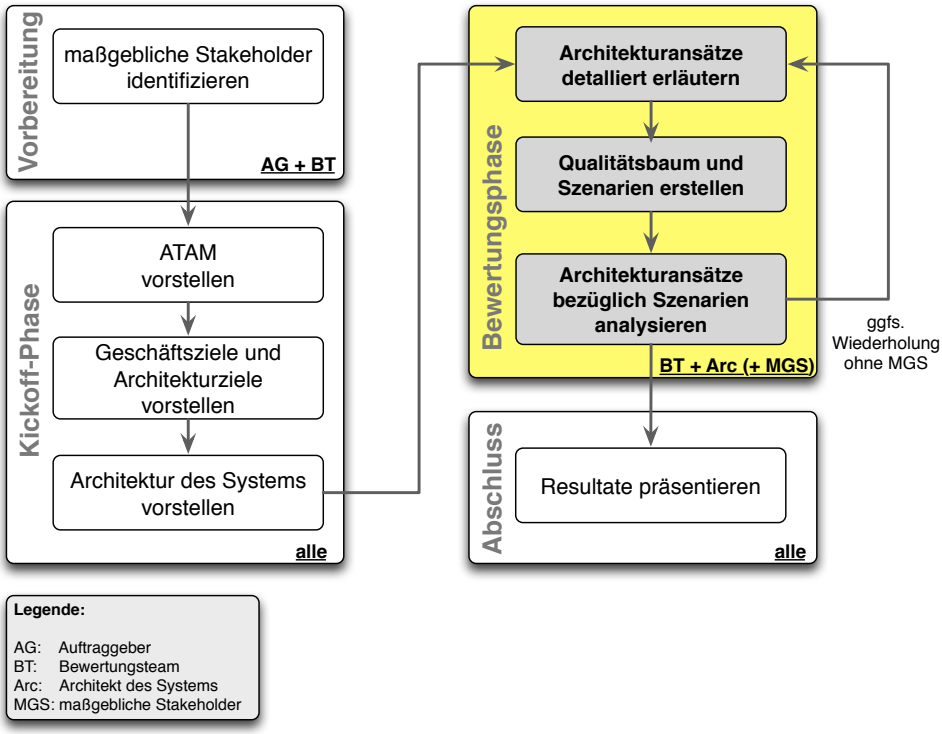


BILD 8.3 Vorgehen bei der Architekturbewertung nach ATAM

Der fundamentale Schritt bei der Architekturbewertung ist die Definition der von den maßgeblichen Stakeholdern geforderten Qualitätsmerkmale in möglichst konkreter Form. Als Hilfsmittel dazu verwenden Sie Qualitätsbäume und Szenarien (siehe auch Abschnitt 3.6.2). Ein Beispiel sehen Sie in Bild 8.4.

Maßgebliche Stakeholder identifizieren

Da die konkreten Ziele Ihrer Architekturen ausschließlich durch die spezifischen Ziele Ihrer Stakeholder vorgegeben werden, muss der Auftraggeber oder Kunde im ersten Schritt jeder Architekturbewertung die hierfür entscheidenden („maßgeblichen“) Stakeholder identifizieren. In der Regel geht eine Architekturbewertung von wenigen Stakeholdern aus, häufig aus dem Management oder der Projektleitung.

[3] Architecture Tradeoff Analysis Method

Stimmen Sie mit Ihren Auftraggebern gemeinsam ab, welche Stakeholder bei der Festlegung der Bewertungsmaße mitarbeiten sollen.* Denken Sie – neben Ihren Auftraggebern oder Kunden – in jedem Fall an Endanwender und Betreiber.

* Ich nenne diese spezielle Gruppe von Stakeholdern gerne die *maßgeblichen* Stakeholder.

Bewertungsmethode (ATAM) vorstellen

Bevor Sie mit der Definition der Bewertungsziele beginnen, sollten Sie den maßgeblichen Stakeholdern die Bewertungsmethode vorstellen. Insbesondere müssen Sie die Bedeutung nachvollziehbarer und konkreter Architektur- und Qualitätsziele klarstellen.

Verdeutlichen Sie den Beteiligten, dass es bei der qualitativen Architekturbewertung um die Identifikation von Risiken und Nicht-Risiken sowie um mögliche Maßnahmen geht, nicht um die Ermittlung (quantitativer) *Noten*.

Geschäftsziele vorstellen

Im Idealfall stellt der Auftraggeber die geschäftlichen (Qualitäts-)Ziele des Systems vor, das zur Bewertung ansteht. Dabei sollte der gesamte geschäftliche Kontext zur Sprache kommen, die Gründe für die Entwicklung des Systems sowie dessen Einordnung in die fachlichen Unternehmensprozesse.

Falls diese Ziele in Ihren Projekten bereits in den Anforderungsdokumenten aufgeführt sind, lassen Sie dennoch die *maßgeblichen Stakeholder* zu Wort kommen: Bei der Bewertung geht es um deren aktuelle Sichtweise. Außerdem sind die Zielformulierungen aus den Anforderungsdokumenten von Systemanalytikern gründlich gefiltert worden.

Ich teile die Erfahrung von [Clements+03], die von *Aha-Erlebnissen* bei der Vorstellung der aktuellen Ziele berichten: Die unterschiedlichen Teilnehmer von Bewertungsworkshops sitzen häufig das erste Mal gemeinsam an einem Tisch und lernen aus erster Hand die Zielvorstellungen anderer Stakeholder kennen. Dabei kommt es zu wertvollen „Ach sooo war das gemeint“-Erlebnissen.

Architektur vorstellen

Anschließend sollte der Architekt des Systems die Architektur vorstellen. Dazu eignet sich die Gliederung der Architekturpräsentation, die Sie in Kapitel 4 kennengelernt haben.

Es sollte in jedem Fall der komplette Kontext des Systems ersichtlich werden, das heißt sämtliche Nachbarsysteme. Außerdem gehören zu einer solchen Vorstellung die Bausteine der oberen Abstraktionsebenen sowie Laufzeitsichten einiger wichtiger Use-Cases oder Änderungsszenarien.

Architekturansätze erläutern

Welche Maßnahmen innerhalb der Architektur oder der Implementierung wurden zur Erreichung der wesentlichen (Qualitäts-)Anforderungen ergriffen? Wie wurden, strukturell oder konzeptionell, die wesentlichen Probleme oder Herausforderungen gelöst? Was sind die wesentlichen, prägenden Architekturentscheidungen?

Architekt oder Entwickler des Systems sollten diese Fragen beantworten, als Ergänzung zur vorangegangenen Überblicksvorstellung.

Einen spezifischen Qualitätsbaum erstellen

Lassen Sie Ihre Stakeholder im Anschluss an die Vorstellung der Geschäftsziele sowie der Architektur im Rahmen eines kreativen Brainstormings die wesentlichen geforderten Qualitätsziele erarbeiten. Ordnen Sie diese anschließend in (informeller) Baum-Form an: Die allgemeineren Merkmale stehen weiter links (oder oben im Baum), die spezielleren Anforderungen weiter rechts (oder unten).

Mindmaps können bei der Erstellung und Dokumentation von Qualitätsbäumen gute Dienste leisten. Sie sind leicht verständlich und insbesondere für ein *Brainstorming* von Qualitätsmerkmalen geeignet.

Ein Beispiel eines solchen Qualitätsbaumes finden Sie in Bild 8.4 und in Bild 8.5. Dort sind als globale Qualitätsziele Performance, Erweiterbarkeit und Verfügbarkeit eingetragen und durch Latenz, Durchsatz etc. verfeinert.

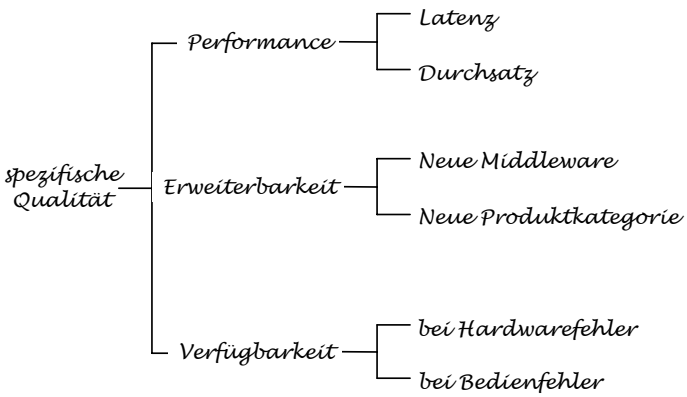


BILD 8.4
Beispiel eines Qualitätsbaumes

Entwickeln Sie Ihre Qualitätsbäume in kleinen und streng fokussierten Workshops, im Stile von Brainstorming-Sitzungen. Lassen Sie die Teilnehmer zu Beginn Architektur- und Qualitätsziele in ihren eigenen Worten beschreiben, ohne sie unmittelbar einzuordnen oder zu bewerten.

Falls Ihre Stakeholder etwas Starthilfe benötigen, können Sie die Architektur- und Qualitätsziele sowie Mengengerüste aus Anforderungsdokumenten in diese Diskussion einbringen. Ich finde es jedoch produktiver, die Teilnehmer einer solchen Diskussion möglichst wenig durch bestehende Dokumente zu belasten.

Anschließend strukturieren Sie gemeinsam mit den Teilnehmern die Beiträge in Form des Qualitätsbaumes. Dabei können Sie die Wurzel des Baumes auch anders benennen, etwa: „*Nützlichkeit*“ oder „*wichtigste Architekturziele*“.

Damit haben Sie den ersten Schritt, die Beschreibung der Bewertungsziele, bereits erledigt.

Qualitätsmerkmale durch Szenarien verfeinern

Szenarien: Abschnitt 3.6.2

Jetzt verfeinern Sie gemeinsam mit den maßgeblichen Stakeholdern die Qualitätsanforderungen und Architekturziele des Systems durch Szenarien. Wichtig ist dabei, die Formulierung der Szenarien möglichst konkret und operationalisiert zu halten. In Bild 8.5 sehen Sie einen um zwei Szenarien angereicherten Qualitätsbaum.

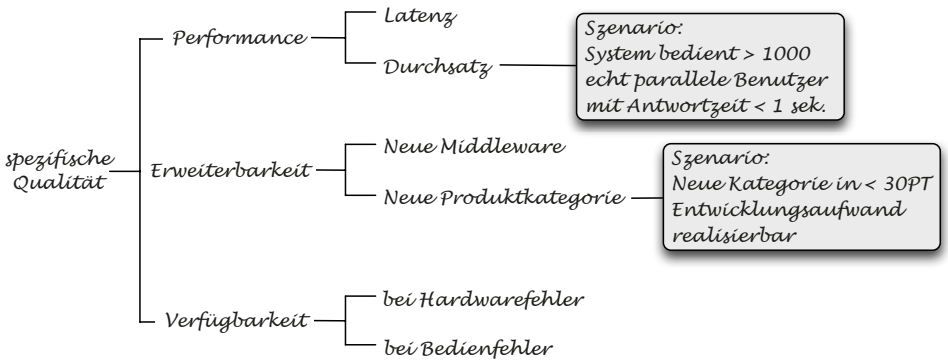


BILD 8.5 Qualitätsbaum mit Szenarien

Oftmals fällt es den Projektbeteiligten leichter, in konkreten Szenarien zu denken als in abstrakten Qualitätsmerkmalen. Beginnen Sie in solchen Fällen ruhig mit einem Brainstorming von Szenarien, und ordnen Sie die zugehörigen Qualitätsanforderungen erst im Nachgang als Baum oder Mindmap.

Mehr als 50 ausführlich formulierte praktische Beispiele zu Szenarien finden Sie im freien arc42-Subprojekt „quality-requirements“ unter:
https://bitbucket.org/arc42/quality-requirements

Bereits für mittelgroße IT-Systeme können Sie in solchen Brainstorming-Sitzungen durchaus 30 bis 50 verschiedene Szenarien finden. Im Normalfall wird Ihnen für die detaillierte Bewertung derart vieler Szenarien jedoch keine Zeit bleiben, sodass Sie sich auf einige wenige beschränken müssen. Dabei helfen Prioritäten.

Lassen Sie Ihre maßgeblichen Stakeholder die gefundenen Szenarien nach ihrem jeweiligen geschäftlichen Nutzen oder Wert priorisieren. Ich bevorzuge dabei eine kleine Skala, etwa A = wichtig, B = mittel, C = weniger wichtig.

Machen Sie deutlich, dass es bei den Prioritäten lediglich um die Reihenfolge bei der Bewertung geht! Auch ein für die Bewertung mit C priorisiertes Szenario wird das Entwicklungsteam liefern – sofern irgendwie möglich!

Architektur hinsichtlich der Szenarien analysieren

Mit den (priorisierten) Szenarien für die aktuellen Qualitätsanforderungen und Architekturziele besitzen Sie nun einen Bewertungsmaßstab für Ihre konkrete Architektur. Jetzt folgt die eigentliche Bewertung, die Sie in der Regel in einer kleinen Gruppe gemeinsam mit den Architekten des Systems durchführen. Weitere Stakeholder sind dabei im Normalfall nicht anwesend.[4]

Gehen Sie bei dieser Kernaufgabe gemäß den Prioritäten der Szenarien vor, beginnend mit den wichtigen und schwierigen! Lassen Sie sich in Form eines *Walkthrough* von Architekten oder Entwicklern erläutern, wie die Bausteine des Systems zur Erreichung dieses Szenarios zusammenspielen oder welche Entwurfsentscheidungen das jeweilige Szenario unterstützen.

Als Bewerter oder Auditor einer Architektur untersuchen Sie die zentralen Architekturziele und Qualitätsanforderungen auf Basis der jeweils definierten Szenarien gemeinsam mit dem Architekten des Systems. Lassen Sie sich die zugehörigen Architekturentscheidungen und -ansätze erläutern, und beantworten Sie die folgenden Fragen:

- Welche Architekturentscheidungen wurden zur Erreichung eines Szenarios getroffen?
- Welcher Architekturansatz unterstützt die Erreichung des Szenarios?
- Welche Kompromisse ging man mit dieser Entscheidung ein?
- Welche anderen Qualitätsmerkmale oder Architekturziele beeinflusst diese Entscheidung darüber hinaus?
- Welche Risiken erwachsen aus dieser Entscheidung oder aus diesem Ansatz?
- Welche Risiken gibt es für die Erreichung des jeweiligen Szenarios und der damit verbundenen Qualitätsanforderungen?
- Welche Analysen, Untersuchungen oder Prototypen stützen diese Entscheidung?

Leider muss ich Ihnen an dieser Stelle eine schlechte Nachricht überbringen: Es gibt keinen Algorithmus, der für Sie die Zielerreichung einer Architektur hinsichtlich der Szenarien bestimmt. Ihre Erfahrung (oder die des Bewertungsteams) ist gefragt – auch Ihre subjektive Einschätzung.

Ergebnisse: Risiken + Kompromisse

Als Resultate erhalten Sie einen Überblick über die Güte der Architektur hinsichtlich konkreter Szenarien, und damit auch hinsichtlich der spezifischen Architektur- und Qualitätsziele. Sie finden heraus,

- welche **Risiken** bezüglich der Erreichung oder Umsetzung wesentlicher Qualitätsanforderungen oder Architekturziele existieren;
- welche Szenarien auf jeden Fall (d. h. risikolos) erreicht werden; ([Clements+03] spricht hier von *Non-Risks*);
- welche **Kompromisse** bei Entwurf und Entwicklung eingegangen wurden (beispielsweise wenn zugunsten hoher Performance die Komplexität gesteigert und die Änderbarkeit reduziert wurden).

[4] Ziehen Sie andere Beteiligte hinzu, wenn es während der Bewertung Rückfragen zu den Szenarien (= Qualitätsanforderungen) gibt!

Krönender Abschluss: Maßnahmen definieren

Offiziell gehört es nicht mehr zu ATAM – aber zu jeder realen Bewertung: die Formulierung von Maßnahmen, Strategien oder Taktiken gegen die gefundenen Risiken.

Das ist ganz normale Architektur- und Entwicklungsarbeit – nur können Sie jetzt auf Basis sehr spezifischer Anforderungen (= Szenarien + erkannte Risiken) konstruktive Entscheidungen treffen.

Positive Nebenwirkungen von Architekturbewertungen

Neben den wertvollen Aussagen hinsichtlich Zielerreichung und Risiken bringen qualitative Architekturbewertungen oftmals eine Reihe positiver Nebeneffekte hervor.[5]

- Die maßgeblichen Stakeholder diskutieren über ihre eigenen (Qualitäts-)Ziele.[6] Wesentliche Anforderungen werden dadurch präzisiert – sehr zur Freude der Entwicklungsprojekte.
- Zentrale Architekturentscheidungen werden offengelegt und hinsichtlich ihrer Risiken und eingegangenen Kompromisse transparent.
- Dokumentation wird besser, weil die Bewertung einzelner Szenarien ohne Dokumentation häufig kaum durchführbar ist.

Schließlich kommt es vor, dass maßgebliche Stakeholder sich des hohen Risikos bewusst werden, das durch einige ihrer Anforderungen und Qualitätsziele entsteht.

Bewerten Sie die Architektur so früh wie möglich. Die Ergebnisse qualitativer Architekturbewertung sind für Projekte und Systeme meistens von langfristigem Wert. Investition in Architekturbewertung lohnt sich.*

* Zumal auch kurze Bewertungsworkshops von wenigen Stunden Dauer schon kritische Risiken in Architekturen aufdecken können – erheblich schneller als umfangreiche Code- oder Schwachstellenanalysen.

[5] Obwohl Seiten- und Nebeneffekte im aktuellen Trend „funktionaler Programmierung“ ja als böse gelten – bei qualitativer Bewertung sind sie ausdrücklich erwünscht. ☺

[6] Diese Diskussion der Stakeholder hätte bereits in der Analysephase stattfinden sollen. Leider besteht in der Realität hinsichtlich Architekturzielen und Qualitätsanforderungen meistens weder Klarheit noch Einigkeit.

8.2 Quantitative Bewertung durch Metriken

Es gibt eine ganze Reihe quantitativer Metriken, die Sie für Ihre Projekte und Ihren Quellcode ermitteln können. Einige Beispiele:

- Für Anforderungen: Anzahl der geänderten Anforderungen pro Zeiteinheit.
- Für Quellcode: Abhängigkeitsmaße (Kopplung), Anzahl der Codezeilen, Anzahl der Kommentare in Relation zur Anzahl der Programmzeilen, Anzahl statischer Methoden, Komplexität (der möglichen Ablaufpfade, *cyclomatic complexity*), Anzahl der Methoden pro Klasse, Vererbungstiefe und einige mehr.
- Für Tests und Testfälle: Anzahl der Testfälle, Anzahl der Testfälle pro Klasse/Paket, Anzahl der Testfälle pro Anforderung, Testabdeckung.[7]
- Für ganz oder teilweise fertige Systeme: Performanceeigenschaften wie Ressourcenverbrauch oder benötigte Zeit für die Verarbeitung bestimmter Funktionen oder Anwendungsfälle.
- Für Fehler: Mittlere Zeit bis zur Behebung eines Fehlers, Anzahl der gefundenen Fehler pro Paket/Subsystem.
- Für Prozesse: Anzahl der implementierten/getesteten Features pro Zeiteinheit, Anzahl der neuen Codezeilen pro Zeiteinheit, Zeit für Meetings in Relation zur gesamten Arbeitszeit, Verhältnis der geschätzten zu den benötigten Arbeitstagen (pro Artefakt), Verhältnis von Manager zu Entwickler zu Tester.

Sie sehen, eine ganze Menge von Eigenschaften können Sie quantitativ charakterisieren.

Dennoch: Metriken als Unterstützung der Entwicklung haben sich nach meiner Erfahrung in der Praxis bisher kaum durchgesetzt. Insbesondere messen viele Unternehmen nur punktuell, anstatt Messungen kontinuierlich mit der Entwicklung zu verzahnen.

Beispiel: In einem mittelgroßen Client/Server-Projekt gab der Kunde einem Forschungsinstitut den Auftrag, die etwa 1000 entwickelten und bereits produktiv (und zur Zufriedenheit des Kunden) eingesetzten Java-Klassen durch Metriken zu analysieren. Ziel dieser Untersuchung sollte sein, spätere Wartungsaufwände besser abschätzen zu können.

Im ersten Bericht der Forscher wurde die Qualität der Software heftig kritisiert, weil verschiedene Metriken starke Abweichungen vom wissenschaftlichen Ideal zeigten.

Kunde, Architekt, Entwickler und Projektleiter waren entsetzt – die Software funktionierte einwandfrei, dennoch wurde die Qualität bemängelt. Damit war die Akzeptanz von Software-Metriken im Entwicklungsteam auf ein Allzeit-Tief gesunken.

Übrigens stellte sich später heraus, dass ein übereifriger Forscher neben den Klassen der Anwendung den gesamten GUI-Framework sowie die zugekaufte Middleware mit analysiert hatte und die bemängelten Metrik-Defekte ausschließlich in den Schnittstellen zu diesen Frameworks lagen.

[7] Testabdeckung ist eine verbreitete Metrik, jedoch auch eine sehr kritische. [Binder2000] und [Marick97] beschreiben einige der damit verbundenen Probleme im Detail.

Metriken über Zeit beobachten

Einige Ratschläge zum Einsatz von Metriken, insbesondere Code-Metriken:

- Metriken können gute Hinweise für strukturelle Veränderungen von Software geben – über die Funktionsfähigkeit und Qualität zur Laufzeit sagen Codemetriken hingegen nichts aus. Setzen Sie Metriken daher kontinuierlich ein, und beobachten Sie deren Veränderungen über die Zeit.
- Metriken benötigen einen fachlichen und technischen Kontext, um vergleichbar zu sein. Sammeln Sie Daten aus Vergleichsprojekten.
- Metriken können bei unvorsichtiger Anwendung wichtige strukturelle Aussagen über Entwürfe im Zahlendschungel verbergen. Daher: Minimieren Sie die Anzahl der Metriken, und konzentrieren Sie Analysen auf überschaubare Ausschnitte. Weniger ist oft mehr!
- Insbesondere sollten Sie Abhängigkeiten im Quellcode durch ein Werkzeug bei jedem Build* messen und überwachen.

* Sie haben doch hoffentlich einen Daily Build etabliert, inklusive täglicher Durchführung sämtlicher Unit-Tests, oder? Solche Werkzeuge (wie Jenkins, Hudson, Gradle oder Maven) können Metriken mit jedem Build berechnen.

Systeme nur auf Basis von Quellcodemetriken zu bewerten, ist riskant, weil grundlegende *strukturelle* Schwachstellen dadurch möglicherweise überhaupt nicht aufgedeckt werden. So kann es vorkommen, dass Sie qualitative Defizite erst spät im Entwicklungsprozess entdecken.[8] Möglicherweise notwendige Änderungen der Architektur sind dann in der Regel teuer und aufwendig. Daher möchte ich mich auf den folgenden Seiten auf die qualitative Bewertung von Architekturen konzentrieren.[9]

Messungen am laufenden System liefern nur Indikatoren

Metriken können jedoch zur Systemlaufzeit (etwa: Performancewerte, Ressourcenverbrauch oder auch Fehlerzahlen) wertvolle Indikatoren für Probleme liefern. In jedem Fall müssen Sie auf Basis solcher Messwerte noch *Ursachenforschung* betreiben!

Metriken können Ihnen zeigen, ob Sie Ihren Code gut geschrieben haben. Ob Sie allerdings den richtigen Code haben, können Sie nur durch qualitative Analysen herausfinden.

Beginnen Sie bereits in frühen Entwicklungsphasen mit der Erfassung von Laufzeitmetriken, insbesondere Performancedaten (etwa: Durchlaufzeiten wichtiger Anwendungsfälle und Ressourcenverbrauch). Ermitteln Sie diese Werte automatisiert, und achten Sie auf Trends.

[8] Iterative Entwicklungsprozesse mit frühzeitigem Feedback vermeiden dieses Problem.

[9] Ich rate Ihnen aber, in Entwicklungsprojekten Codemetriken zur Steigerung der Qualität einzusetzen. Sie sollten in jedem Fall Komplexitäts- und Abhängigkeitsmaße auf Codeebene messen und überwachen – viele Entwicklungsumgebungen unterstützen Sie dabei.

Bewerten Sie insbesondere Architekturen

Von den vielen Artefakten, die innerhalb von Software-Projekten entstehen, eignen sich Architekturen meiner Meinung nach besonders für die Bewertung: Architekturentscheidungen haben oftmals große Tragweite, sowohl für die zu entwickelnden Systeme als auch für die Projekte und Organisationen, die diese Systeme realisieren, benutzen und betreiben.

Viele dieser Entwurfsentscheidungen müssen Teams oder Architekten unter Unsicherheit oder (vagen) Annahmen treffen – das birgt Risiken! Architekturbewertung hilft, diese Risikofaktoren frühzeitig und spezifisch zu identifizieren.

8.3 Werkzeuge zur Bewertung

Einige Aspekte der Qualität von Systemen können Sie anhand automatisierbarer Untersuchungen des Quellcodes beobachten. Mittlerweile existieren eine Reihe freier (teilweise quelloffener) Werkzeuge, die auch sprachübergreifende Codequalität messen und analysieren können.

Als Beispiel möchte ich Ihnen die SonarQube® Plattform[10] skizzieren: Frei nutzbar verfügt sie über Plug-ins für alle erdenklichen Programmiersprachen (Java- und C#, C/C++, Groovy, Scala, Erlang, Drools, PL/SQL, Cobol u. v. m). SonarQube kann:

- Untersuchungsergebnisse in einer Datenbank speichern, und die Veränderungen über die Zeit in einer sogenannten „Time Machine“ sichtbar machen. Ein großartiges Instrument, um in Verbesserungs- oder Refactoring-Projekten Erfolge darzustellen!
- doppelten und ungenutzten Code identifizieren;
- Einhaltung von Coding-Styles prüfen;
- Komplexität im Code messen;
- durch seine Client/Server-Architektur in viele Build-Systeme integriert werden;
- relativ einfach über seine offene Plug-in-Schnittstelle um spezielle Prüfungen erweitert werden.

Es gibt alternative Werkzeuge (z. B. PMD, FindBugs, CheckStyle, Squale, Structure101, SotoGraph) – allerdings schützen Sie diese Werkzeuge natürlich nicht davor, den falschen Code richtig zu entwickeln. Grüne Ampeln bei Werkzeugen bedeuten daher nicht unbedingt glückliche Benutzer oder Stakeholder. Dennoch: Meiner Ansicht nach sollten diese Werkzeuge zum Handwerkszeug praktizierender Softwarearchitekten gehören.

[10] *http://sonarqube.org*.

8.4 Weiterführende Literatur

[Bass+03] und [Clements+03] beschreiben die ATAM-Methode (*Architecture Tradeoff Analysis Method*) ausführlich. Meiner Erfahrung nach ist diese Methode jedoch durch die wiederholte Bewertung der Architekturansätze in vielen Fällen zu aufwendig in der Anwendung. Trotzdem legt sie die Grundlage für den Ansatz der Szenario-basierten Bewertung.

[Kruchten+02] fassen den State of the Art bezüglich der Architekturbewertung zusammen; viele Hinweise, wie eine Bewertung praktisch ablaufen kann.

[Henderson-Sellers96]: ausführliche Darstellung objektorientierter Metriken.

[Lorenz94]: praktische Einführung in Metriken für objektorientierte Systeme.

9 Service-orientierte Architektur (SOA)

Mit kräftiger Unterstützung von Stefan Tilkov

SOA is a Lifestyle.

Ann Thomas Manes, in [SOAX 07]

Fragen, die dieses Kapitel beantwortet:

- Warum gibt es SOA?
- Was ist SOA?
- Wie funktionieren Services?
- Was gehört (noch) zu SOA?
- Was hat SOA mit Softwarearchitektur zu tun?

Warum gibt es SOA?

SOA ist ein Business-Thema

Als Motivation für SOA skizzieren wir die Situation vieler Unternehmen hinsichtlich (externer) Einflussfaktoren und (interner) Informationsverarbeitung: Unternehmen existieren, um Mehrwert zu erwirtschaften, und zwar durch wertschöpfende Geschäftsprozesse. Dabei wirken im Wesentlichen vier Einflussfaktoren auf die Unternehmen und deren Geschäftsprozesse:

1. Die Ansprüche von Kunden und Partnern steigen: Der Druck globaler Märkte auf Unternehmen nimmt zu. Um weiterhin profitabel zu bleiben, müssen Unternehmen ihre Geschäftsprozesse häufig an wechselnde Einflüsse von Märkten und Kunden anpassen. Flexibilisierung und Modularisierung von Geschäftsprozessen wird zur Grundlage unternehmerischen Handelns.
2. Viele Geschäftsprozesse funktionieren ausschließlich mit IT-Unterstützung. Ohne Informatik sind viele Unternehmen nicht handlungs- beziehungsweise lebensfähig. Denken Sie beispielsweise an die gesamte Finanz- und Telekommunikationsbranche. Leider hemmt die IT oftmals die (so dringend benötigte) Flexibilität und Agilität von Unternehmen, weil sie sich nicht schnell oder flexibel genug an geänderte Prozesse adaptieren kann.
3. Bestehende IT-Systeme sind häufig Applikationsmonolithen, unflexible und über lange Jahre gewachsene starre Anwendungssilos. Die zugehörigen IT-Organisationen sind als Fürstentümer organisiert und orientieren sich weniger an der Wertschöpfung des Gesamtunternehmens als an eigener Macht und Einfluss.

4. IT-Budgets werden oftmals für die Entwicklung oder Modernisierung einzelner Anwendungen vergeben, ohne direkten Bezug zu deren aktuellen oder künftigen Wertschöpfung. Dadurch entstehen die sogenannten „Millionengräber". Budgets sollten sich an der Wertschöpfung der IT-unterstützten Services und Geschäftsprozesse orientieren.

Für Unternehmen folgt daraus die Notwendigkeit, die Abstimmung zwischen Geschäft (*Business*) und IT zu verbessern: In Zukunft muss die IT hochflexible Geschäftsprozesse effektiv und schnell unterstützen. Dazu bedarf es der Abkehr von Anwendungsmonolithen und der Schaffung flexibler Software-Einheiten – eben Services. Aus solchen Services müssen Unternehmen neue Geschäftsprozesse in kurzer Zeit entwerfen und in Betrieb nehmen können.

Genau deshalb (und nur deshalb!) brauchen Unternehmen SOA: Es geht um unternehmerische Flexibilität, um die Fähigkeit, auch unter veränderlichen Marktbedingungen als Unternehmen wertschöpfende Geschäftsprozesse anbieten zu können. SOA wird ausschließlich durch diese geschäftliche Argumentation getrieben – Technik ist nur Mittel zum Zweck.

Diese beiden Perspektiven von SOA (geschäftlich/organisatorisch und technisch) führen unserer Erfahrung nach häufig zu Missverständnissen bei der Kommunikation zwischen den beteiligten Personen: Die Fachleute unterstellen den IT'lern ein mangelndes SOA-Verständnis, umgekehrt beschweren sich die IT-Experten über die SOA-unkundigen Fachleute. Unserer Meinung nach gehören zu SOA beide Perspektiven – keine kann alleine funktionieren.

9.1 Was ist SOA?

SOA muss in Unternehmen übergreifend wirken, sowohl auf Geschäftsprozesse als auch auf IT. Daher haben sehr unterschiedliche Beteiligte mit SOA zu tun und jeweils ihre spezifische Auffassung von „Was ist SOA?". Einige Beispiele (in Anlehnung an [Lublinsky 07]) finden Sie in Tabelle 9.1.

Verstehen Sie SOA als Synthese dieser Meinungen:

- Eine Service-orientierte Architektur (SOA) ist eine unternehmensweite IT-Architektur mit Services (Diensten) als zentralem Konzept.
- Services realisieren oder unterstützen Geschäftsfunktionen.
- Services sind lose gekoppelt.
- Services sind selbstständig betriebene, verwaltete und gepflegte Softwareeinheiten, die ihre Funktion über eine implementierungsunabhängige Schnittstelle kapseln.
- Zu jeder Schnittstelle gibt es einen Service-Vertrag, der die funktionalen und nichtfunktionalen Merkmale (Metadaten) der Schnittstelle beschreibt. Zu jeder Schnittstelle kann es mehrere, voneinander unabhängige, Implementierungen geben.

TABELLE 9.1 SOA-Definitionen aus verschiedenen Perspektiven

Perspektive (Stakeholder)	SOA ist ...
Manager	eine Menge von IT-Assets („Fähigkeiten"), aus denen wir Lösungen für unsere Kunden und Partner aufbauen können.
Enterprise-IT-Architekt	Architekturprinzipien und -muster für übergreifende Modularisierung, Kapselung, lose Kopplung, Wiederverwendbarkeit, Komponierbarkeit, Trennung von Verantwortlichkeiten.
Projektleiter	ein Ansatz für parallele Entwicklung mehrerer Projekte oder Aufgaben.
Entwickler, Programmierer	ein Programmiermodell, basierend auf Standards wie Web-Services, REST, XML, HTTP.
Administrator, Operator	ein Betriebsmodell für Software, bei dem wir nur noch Teile unserer Anwendungen (Services) selbst betreiben und viele Services von Partnern nutzen.
Strategieberater, Software-Hersteller	ein unklar definiertes Hype-Thema, das in den Ohren unserer Kunden vielversprechend klingt. Ein gigantischer Beratungs- und Werkzeugmarkt voller Chancen und Risiken.

- Die Nutzung (und Wiederverwendung) von Services geschieht über (entfernte) Aufrufe („Remote Invocation").
- SOA setzt so weit wie möglich auf offene Standards, wie XML, SOAP[1], WSDL und andere.
- Services sind bereits von ihrem Entwurf her *integrierbar*.[2]

In den folgenden Abschnitten wollen wir diese Definition etwas erläutern. Zur Einordnung stellt Bild 9.1 (auf der nächsten Seite) den Begriff „Service" in den Kontext von Unternehmen, Geschäftsprozess und Implementierung.

Services als zentrales Konzept

In SOA übernehmen Services in hohem Maße die Rolle bisheriger „Anwendungen": Unternehmen planen, entwerfen, entwickeln, testen und betreiben Services, nicht Anwendungen. Outsourcing erfolgt auf Basis von Services, nicht von Hardware oder Anwendungen. Abteilungen verantworten Services, keine Anwendungen.

Services realisieren Geschäftsfunktionen

Services realisieren Geschäftsfunktionen, die für ihre Nutzer einen geschäftlichen Mehrwert darstellen. Unter solchen Geschäftsfunktionen verstehen wir Aktivitäten, die ein Unternehmen direkt oder indirekt in seiner Wertschöpfung unterstützen – sei es bei der Fertigung eines Produktes oder dem Erbringen einer Dienstleistung. Beispiele für Services sind Bestellannahme, Kostenkalkulation, Angebotserstellung, Produktionsplanung, Vertragsmanagement, Kundenverwaltung, Abrechnung, Mahnwesen.

---

[1] SOAP: *http://www.w3.org/TR/SOAP*, WSDL: *http://www.w3.org/TR/WSDL*

[2] Dieses Versprechen ist in der Praxis oft schwer einzulösen, weil Service-Anbieter und Service-Nutzer sich nur schwer auf eine gemeinsame Semantik von Daten und Dokumenten einigen können.

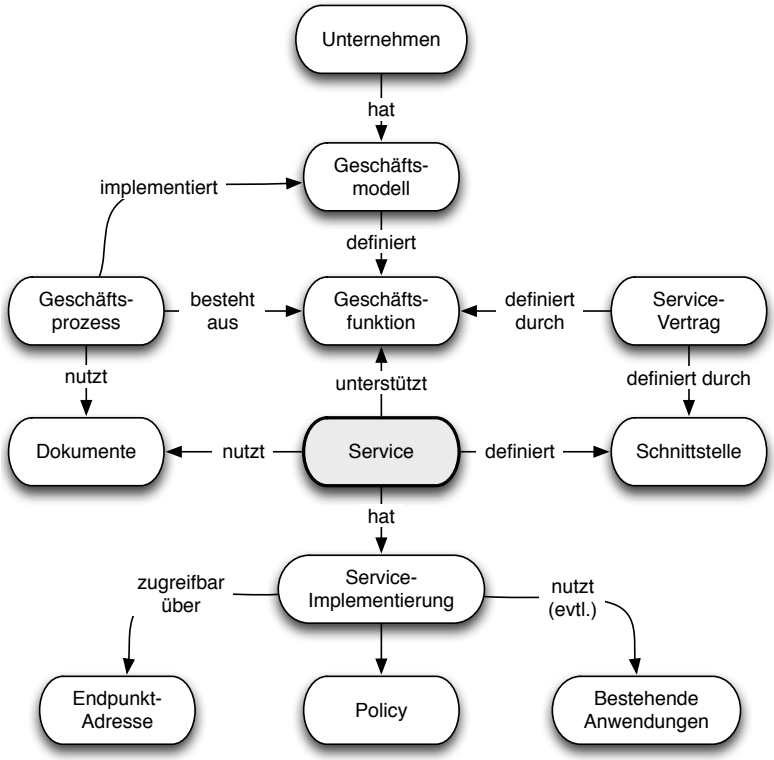


BILD 9.1 Services im Kontext von Unternehmen und Implementierung

Warum beschränken wir Services auf Geschäftsfunktionen? Es gibt doch auch technische Dienste, beispielsweise „Datensicherung". Solche technischen Dienste können sinnvolle und wichtige Funktionen erbringen, tragen jedoch nicht direkt zur Wertschöpfung eines Unternehmens bei. Eine aus wirtschaftlicher Sicht sinnvolle SOA hat geschäftliche Flexibilität (business agility) zum Ziel und orientiert sich an wertschöpfenden Services.

Lose Kopplung

Der Begriff „Kopplung" bezeichnet den Grad der Abhängigkeiten zwischen zwei oder mehr „Dingen". Je mehr Abhängigkeiten zwischen Services bestehen, desto enger sind sie aneinander gekoppelt. Kopplung bezieht sich dabei auf alle Arten der Abhängigkeiten von Services, beispielsweise:

- Zeitliche Abhängigkeit: Ein Service kommuniziert synchron mit einem anderen, d. h. beide Services müssen zeitgleich in Betrieb sein.
- Örtliche Abhängigkeit: Ein Service ruft einen anderen Service unter einer bestimmten Adresse auf – der aufgerufene Service darf diese Adresse nicht ändern.
- Struktur- oder Implementierungsabhängigkeit: Die Implementierung eines Service verwendet die Implementierung eines anderen Service direkt, anstatt über die „offizielle" Schnittstelle zu kommunizieren. Somit ist die Implementierung des aufgerufenen Service nicht mehr austauschbar.

- Datenabhängigkeit: Ein Service nutzt eine (interne) Datenstruktur eines anderen Service, interpretiert beispielsweise die ersten acht Stellen eines Datenbankschlüssels als Artikelnummer. Somit kann die interne Repräsentation dieser Daten nicht mehr geändert werden.

Lose Kopplung von Services bringt eine hohe Unabhängigkeit und damit Flexibilität einzelner Services mit sich. Services in einer SOA sollten lose gekoppelt sein; mit anderen Worten: nur so eng miteinander verflochten, wie unbedingt notwendig.

Das Idealziel – lose Kopplung in allen Dimensionen – ist dabei weder immer erreichbar noch immer wünschenswert. Zeitliche Entkopplung beispielsweise erfordert asynchrone Kommunikation, und die ist schwieriger zu implementieren als synchrone.

Einen weiteren Aspekt möchten wir hervorheben, nämlich das dynamische Binden: Ein Service-Nutzer („Service-Consumer“ oder „Dienstnutzer“) ermittelt die Adresse seines Service-Anbieters (auch „Service-Provider“ oder „Dienstanbieter“ genannt) in der Regel erst zur Laufzeit. Hierbei unterstützen ihn spezialisierte Verzeichnisdienste („Registries“).

Schnittstelle und Implementierung

Jeder Service stellt für seine Nutzer eine fest definierte Schnittstelle bereit. Intern, das heißt nach außen unsichtbar, besitzt jeder Service eine Implementierung, die letztendlich für die eigentliche Arbeit des Service verantwortlich zeichnet. Die Implementierung eines Service besteht wiederum aus Geschäftslogik und zugehörigen Daten.

Services bieten ihre Dienste ausschließlich über diese Schnittstellen an. Service-Nutzer sollen und dürfen keinerlei Annahmen über das Innenleben (die Implementierung) von genutzten Services treffen. Dieses Geheimnisprinzip stellt die Unabhängigkeit und lose Kopplung einzelner Services sicher.

Service-orientierte Architekturen setzen dieses Konzept durchgängig ein: Sämtliche Services (sowohl Geschäfts- als auch technische Services) bieten klar definierte Schnittstellen an, für die es eine oder mehrere Implementierungen geben kann. Die Nutzer eines Service besitzen ausschließlich eine Abhängigkeit zur Schnittstelle des genutzten Service.

Service-Vertrag und Metadaten

In einer SOA gibt es eine Vielzahl von Metadaten, d. h. Daten über Daten. Dazu zählen funktionale und nichtfunktionale Aspekte, beispielsweise

- Beschreibungen der Service-Schnittstellen, der möglichen Operationen und der an ihnen ausgetauschten Informationen;
- Sicherheitsanforderungen und Berechtigungen;
- Performanceanforderungen;
- organisatorische Zuordnungen;
- die Adresse, unter der ein Service aufrufbar ist (auch „Endpunkt-Adresse“ genannt).

Es gibt zwei Arten solcher Metadaten: funktionale und nichtfunktionale. Zu den funktionalen Metadaten zählen die Schnittstellendefinitionen und die Definition der ausgetauschten Informationsobjekte. Nichtfunktionale Metadaten beschreiben Aspekte wie Sicherheitsrestriktionen, Transaktionseigenschaften oder den Zuverlässigkeitsgrad der Nachrichtenzustellung. Nichtfunktionale Eigenschaften eines Service heißen auch Policies. Schnittstelle und Policy zusammen bilden den Service-Vertrag.

Ein Service kann durchaus mit identischer Schnittstelle, aber mit verschiedenen Policies (d. h. in unterschiedlichen nichtfunktionalen Ausprägungen) angeboten werden. Betrachten Sie als Beispiel die Kursabfrage von Wertpapieren:

- Eine Policy (die kostenfreie „Community Edition“ dieses Service) liefert die um 15 Minuten gegenüber der Börse verzögerten Kursdaten.
- Eine zweite Policy (die kostenpflichtige „Premium Edition“) liefert über die identische Schnittstelle die Kursdaten in Echtzeit.

Wiederverwendung durch Aufruf

Übliche Komponenten- oder Objektmodelle unterscheiden sich von Services durch ein grundlegend unterschiedliches Programmiermodell:

- In der Objektorientierung können von einer Klasse viele Instanzen existieren. Diese unterscheiden sich durch ihren internen Zustand, der unmittelbar einem bestimmten Ausführungskontext entspricht. Der Lebenszyklus von Objekten wird durch dessen Benutzer gesteuert: Benötigen Sie ein Objekt, erzeugen Sie sich eine Instanz. Wenn Sie fertig sind, lassen Sie diese Instanz wieder entfernen.
- Der Lebenszyklus von Services in einer SOA hingegen ist an keinen bestimmten Service-Nutzer gebunden. Services existieren in ihrer eigenen Betriebsumgebung, unabhängig von ihren Nutzern.
- Wiederverwendung von Objekten oder Komponenten geschieht durch Einbettung in den Ausführungskontext der Nutzer.
- Services laufen dort ab, wo ihr Betreiber (oder Provider) sie ausführen lässt – in der Regel über Rechnergrenzen getrennt von den Service-Nutzern. Daraus folgt die Notwendigkeit zur systemübergreifenden (remote) Kommunikation, was wiederum signifikante Performance-Nachteile gegenüber herkömmlicher Programmierung bedeutet.

(Selbstbeschreibende) Dokumente

Service-Aufrufe übergeben ihre Daten in Form von *Dokumenten*, also strukturierten *Datenbehältern*. Meist verwenden Services dazu XML-Dokumente, deren Aufbau durch spezifische XML-Schemata vom Service-Provider vorgegeben wird. Selbstbeschreibende Formate schützen dabei vor Fehlern, sowohl bei der Interpretation der Dokumente als auch bei fachlichen Erweiterungen. Vergleichen Sie beispielsweise den folgenden Methodenaufruf

```
createInvoice( 20070905, 42, 84, "K-SQ-42")
```

mit dem Dokument(fragment):

```
<invoice>
         <date>2007-09-05</date>
         <productid>42</productid>
         <quantity>84</quantity>
         <customerid>K-SQ-42</customerid>
</invoice>
```

Zum Verständnis des Methodenaufrufs müssen Sie die Signatur dieser Methode kennen, da der Methodenaufruf nicht selbstbeschreibend ist. Die Übergabe zusätzlicher Informationen (beispielsweise Adressdaten zur Rechnung) erfordert Codeänderungen in der betroffenen Klasse.

Sie können hingegen dem Dokument durchaus einige neue Daten hinzufügen, ohne dass die betroffenen Services geändert werden müssen: Ein Service-Provider kann, beispielsweise mit XPath, aus dem Dokument genau die für ihn notwendigen Informationen extrahieren. Zusätzliche Daten stören nicht.

Falls Sie jetzt an Performance denken: Korrekt – die in selbstbeschreibenden Dokumenten enthaltene Informationsredundanz kostet Laufzeit, verursacht durch Parsing-Aufwände und zusätzlichen Datentransfer. Andererseits gewinnen Sie eine erhebliche Flexibilität, der in SOA große Bedeutung zukommt.

9.2 So funktionieren Services

Wir möchten Ihnen nun erklären, wie Services einer SOA technisch funktionieren. Die Unterscheidung zwischen Service-Anbieter (Provider) und Service-Nutzer (Consumer) haben wir oben bereits erörtert – sie wird jetzt bedeutsam.

Die Zusammenarbeit zwischen Consumer und Provider benötigt eine ganze Menge Vorbereitung:

- Der Provider definiert seine funktionale Schnittstelle in einer maschinenlesbaren Form (etwa: WSDL, Web-Service Description Language). Diese Schnittstelle wird Bestandteil des Service-Vertrags.
- Ein Provider implementiert diese Schnittstelle und fügt die nichtfunktionalen Eigenschaften dieser Implementierung (ihre „Policies“) als Metadaten dem Servicevertrag hinzu.
- Der Provider stellt die Implementierung des Service in einer Laufzeitumgebung zur Verfügung. Dazu legt er die Adresse (Endpunkt-Adresse) fest.
- Schließlich publiziert der Provider all diese Informationen in einem Service-Verzeichnis. Nun können Service-Consumer auf diese Informationen zugreifen.
- Ein Service-Consumer sucht (und findet) im Service-Verzeichnis die Definition der Provider-Schnittstelle.
- Unter Nutzung der Provider-Schnittstelle entwickelt der Consumer einen eigenen Service, der den Provider „verwendet“, d. h. zur Laufzeit bestimmte Operationen der Provider-Schnittstelle aufruft.

Dieses Zusammenspiel zeigt Bild 9.2.

Es gibt viele technische Varianten, mit denen Sie diese Art der Zusammenarbeit implementieren können. Einige davon (aber nicht die einzigen!) sind Web-Services, REST, CORBA oder nachrichtenbasierte Systeme.

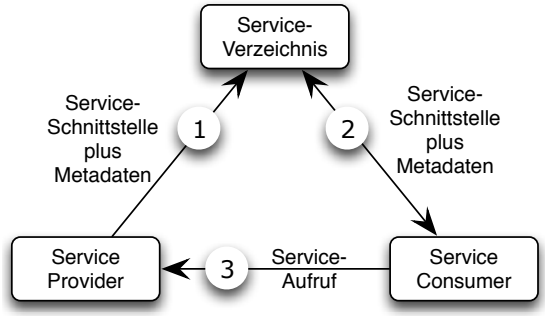


BILD 9.2
Das Service-Dreieck:
Consumer, Provider, Verzeichnis

9.3 Was gehört (noch) zu SOA?

Services bilden den Kern von SOA – jedoch gehören noch einige weitere Bestandteile und Konzepte dazu, ohne die im wahrsten Sinne „nichts läuft“. Einige möchten wir Ihnen nachfolgend vorstellen.

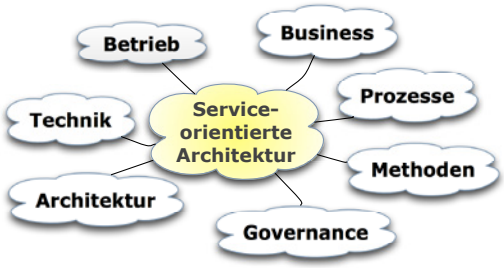


BILD 9.3
Was gehört noch zu SOA

Architekturstile

Sie können SOA mit unterschiedlichen Architekturstilen aufbauen:

- schnittstellenorientiert: Ähnlich dem konventionellen Remote Procedure Call (RPC) definieren Sie für sämtliche Funktionen Ihrer Services eigene Operationen. Meist folgen diese Operationen dem (synchronen) Kommunikationsmuster *Request-Response*, gelegentlich sogar mit dem Verzicht auf Rückgabewerte (*one-way operations*). Die Operationen erhalten in ihren Signaturen Parameter.
- nachrichtenorientiert: Bei einer nachrichtenorientierten SOA stehen die ausgetauschten Informationen im Vordergrund. In der Regel spricht man hier von Dokumenten, die durch Nachrichten übertragen werden – anders formuliert: Nachrichten enthalten als Nutzdaten (*Payload*) ein Dokument und zusätzlich Adressinformationen und andere Metadaten. Dieser Architekturstil kommt häufig bei asynchron gekoppelten Systemen zum Einsatz, im Zusammenhang mit entsprechender Kommunikationsinfrastruktur (*message-oriented middleware*, MOM).[3]

[3] Eine geniale Einführung in diesen Architekturstil gibt [Hohpe+03] als „Integrationsmuster“.

- ressourcenorientiert: Eine Alternative für Service-Orientierung im großen Stil ist REST, der ressourcenorientierte Architekturstil. Die Zusammenarbeit zwischen Services erfolgt hierbei über eine einheitliche und minimalistische Schnittstelle, die aus vier Operationen besteht:
  - GET zum Lesen einer Ressource
  - PUT zum Verändern einer Ressource
  - DELETE zum Löschen
  - POST zum Erzeugen bzw. zur sonstigen Verarbeitung einer Ressource

Anwendungs-Frontends

Auch in einer SOA bedienen Menschen ganz normale Programme auf ganz normalen Client-Rechnern. Das sind die Anwendungen und Programme eines Unternehmens, die Geschäftsprozesse auslösen und deren Ergebnisse verarbeiten. Anwendungs-Frontends können eine echte Benutzeroberfläche haben, aber auch Batchsysteme sein. Solche Anwendungs-Frontends nutzen in ihrer Implementierung vorhandene Service-Provider und treten selbst als Service-Consumer auf.

XML & Co.

Markup-Sprachen auf der Basis von XML bilden zurzeit die *lingua franca* von SOA, obwohl bereits Alternativen wie YAML oder JSON bereitstehen.

Als Softwarearchitekt in SOA-Projekten sollten Sie über ein grundlegendes Verständnis von XML-Techniken verfügen. Dazu zähle ich die folgenden:

- XML, Attribute, Entitäten und „Content“, Wohldefiniert- und Wohlgeformtheit, XML-Namensräume.
- XML-Schema zur Beschreibung konkreter XML-Sprachen und deren Syntax.
- XPAth zur Navigation und Selektion innerhalb von XML-Dokumenten.

Ein verbreitetes Format zum Nachrichtenaustausch innerhalb einer SOA stellt SOAP dar. Das Gegenstück zur Beschreibung von Services heißt WSDL (Web-Service Description Language).

Ablaufsteuerung und Koordination von Services

Services sollen koordiniert ablaufen und durch ihr Zusammenwirken die Unternehmensprozesse unterstützen. Diese Koordination kann entweder durch eine zentrale Steuerung (im Sinne eines Dirigenten, der die Musiker eines Orchesters koordiniert) erfolgen oder aber durch föderierte Intelligenz (im Sinne eines Balletts, in dem sich Tänzerinnen und Tänzer eigenständig koordinieren).

Beide Varianten, in der SOA-Terminologie *Orchestrierung* beziehungsweise *Choreographie* genannt, kommen in SOA zur Ablaufsteuerung von Services zum Einsatz – die zugehörigen Standards heißen BPEL[4] und WS-CDL[5].

[4] BPEL: *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*
[5] WS-CDL: *http://www.w3.org/TR/ws-cdl-10/*

Technische Infrastruktur für SOA

Eingangs haben wir bereits Registries als Verzeichnisse von Service-Metadaten erwähnt. Solche Registries übernehmen zur Laufzeit die Vermittlung von Anfragen zwischen Service-Consumern und Service-Providern. Vorher müssen die Service-Provider eine solche Registry mit den notwendigen Metadaten über den jeweiligen Service befüllen. Eine Registry kann darüber hinaus interessante Informationen über das Nutzungsverhalten von Services sammeln, die Einhaltung von Serviceverträgen überprüfen oder zwischen unterschiedlichen Policies von Consumern und Providern vermitteln.

Andererseits bilden Registries auch die Grundlage von Service-Wiederverwendung: Zur Entwicklungszeit können sich Service-Designer und -Entwickler über vorhandene Service-Verträge informieren. Manche Verzeichnisse bieten außerdem die Möglichkeit, sämtliche weiteren Dokumente über Services zu verwalten, etwa Anforderungsdokumente, Testberichte oder Ähnliches. Diese Werkzeugkategorie heißt dann „Service-Repository“ und könnte grundsätzlich von einer Registry getrennt ablaufen.

Die dritte Kategorie von Werkzeugen, die wir ansprechen möchten, ist der Enterprise-Service-Bus (ESB). Ein ESB vernetzt sämtliche technischen Beteiligten einer SOA miteinander. Möchte ein Service-Nutzer mit einem Service-Provider in Kontakt treten, so übernimmt der ESB die gesamten Details der Kommunikation – inklusive notwendiger Zusatzleistungen. Auf den ersten Blick ähnelt dieser Ansatz dem Request-Broker aus CORBA, verfügt jedoch über einige zusätzliche Facetten:

- Technische Verbindung aller Komponenten, inklusive der notwendigen Netzwerk- und Protokolldetails.
- Ein Service-Bus muss Brücken zwischen heterogenen Technologien schlagen: Service-Consumer sind möglicherweise in anderen Programmiersprachen entwickelt als Service-Provider. Der ESB abstrahiert zusätzlich von Betriebssystemen und Middleware-Protokollen.
- Kapselung verschiedener Kommunikationskonzepte: Der ESB ermöglicht beispielsweise die Kommunikation synchroner und asynchroner Komponenten miteinander, oder aber die Übersetzung zwischen verschiedenen Protokollen.
- Bereitstellung technischer Dienste: In einer „echten“ SOA müssen neben den eigentlichen (wertschöpfenden) Services noch technische Dienste vorhanden sein, wie etwa Logging, Autorisierung oder Nachrichtentransformation. Diese werden über den Service-Bus angeboten (oder durch ihn vermittelt.).

Governance und Management

So wie Unternehmen früher Dutzende von Anwendungen (Programmen, Systemen) eingesetzt haben, können in einer SOA schnell viele Hundert verschiedene Services entstehen. Den Überblick zu behalten, deren Weiterentwicklung (oder Abschaltung) gezielt zu steuern, Qualitätsvorgaben zu überprüfen und die Services ggfs. auch extern anzubieten – all das bedarf einer Management- oder Steueraufgabe: SOA Governance.

Für Softwarearchitekten sicherlich ein Randthema, für eine erfolgreiche SOA-Einführung und einen SOA-Betrieb aus meiner Sicht ganz wesentlich.

9.4 SOA und Softwarearchitektur

Am Anfang dieses Kapitels schrieben wir, dass SOA ein Business-Thema ist. Hier nochmals zur Wiederholung:

> ... Es geht um unternehmerische Flexibilität, um die Fähigkeit, auch unter veränderlichen Marktbedingungen als Unternehmen wertschöpfende Geschäftsprozesse anbieten zu können. ... Technik ist nur Mittel zum Zweck.

Insofern schätzen wir den Zusammenhang von SOA, Softwarearchitektur und anderen Disziplinen wie folgt ein: Vornehmlich gehört SOA in den organisatorischen Bereich der Unternehmen.

- Die Annäherung von IT und Business muss organisatorisch vollbracht werden – hier haben Softwarearchitekten keinen nennenswerten Einfluss.
- Anschließend gilt es, die IT-Gesamtarchitektur der Unternehmen Service-orientiert zu organisieren. Hierzu erhalten Softwarearchitekten im Idealfalle passende Vorgaben von Enterprise-Architekten – im schlimmsten Fall darf ein (armer) Softwarearchitekt eines kleinen Entwicklungsprojektes die Weichen der Gesamt-SOA stellen (eine undankbare Aufgabe, weil kurzfristige Projektziele häufig von langfristigen Unternehmens- und SOA-Zielen abweichen).
- Danach sollten Unternehmen eine wirkungsvolle und durchgängige „Regierung“ für die Gesamt-IT sowie ihre SOA etablieren – neudeutsch als „Governance“ bekannt.

Deshalb können Softwarearchitekten nur in sehr geringem Maße die Wege einer übergreifenden SOA bestimmen. Andererseits bestimmen technische Entwurfsentscheidungen darüber, ob Service-Implementierungen den Erwartungen ihrer Nutzer gerecht werden – und solche Entscheidungen treffen Softwarearchitekten.

Um es mathematisch zu formulieren: Für den Erfolg von SOA sind Softwarearchitekten eine notwendige, aber nicht hinreichende Bedingung.

9.5 Weiterführende Literatur

[SOAX 07] sammelt Grundlagen- und Erfahrungsberichte vieler Experten zu SOA und verwandten Themen. Deckt auf fast 900 Seiten sämtliche Bereiche von SOA ab, organisatorische wie technische.

[Lublinsky 07] beschreibt SOA als Architekturstil, kurz und prägnant. Lesenswert, enthält viele Referenzen. Gut finde ich seinen Ansatz einer *Pattern-Language* für SOA.

[Melzer 07] erläutert die grundlegenden Aspekte, die hinter SOA und Web-Services stecken.

Anmerkung: Teile dieses Kapitels entstanden auf Basis des „Einmaleins der SOA“ von Stefan Tilkov und Gernot Starke aus [SOAX 07].

10 Enterprise-IT-Architektur

Enterprise architecture is the organizing logic for business processes and IT infrastructure.

[Ross+06]

Fragen, die dieses Kapitel beantwortet:

- Auf welchen Ebenen arbeiten Architekten?
- Was bedeutet Enterprise-IT-Architektur?
- Welche Aufgaben sollten Enterprise-IT-Architekten wahrnehmen?

Architekten arbeiten auf verschiedenen Ebenen

Bisher handelte dieses Buch immer von Softwarearchitektur, also den Strukturen einzelner Softwaresysteme. Nun lernen Sie einige weitere Arten von Architekturen kennen, deren Ergebnisse und Entwürfe von prägendem Einfluss auf Softwaresysteme sind.

Die wesentlichen Architekturebenen von Unternehmen finden Sie als Architekturpyramide in Bild 10.1 (in Anlehnung an [Dern09]) dargestellt. Diese Pyramide zeigt nach oben hin zunehmend abstrakte Architekturebenen:

- Die Spitze der Architekturpyramide bildet die Strategie – die Ziele der Organisation – sowie mögliche Wege, um sie zu erreichen, und deren Grenzen. Verfeinert durch die Business-Architektur klärt die Strategie unter anderem die Zusammenarbeit zwischen Business und IT. Auf Strategien gehe ich nicht weiter ein, sondern verweise interessierte Leserinnen auf [Keller12].
- Die Business-Architektur definiert die geschäftlichen Prozesse und betriebswirtschaftlichen Funktionen einer Organisation. Sie verwendet den Begriff „Informationen“ in wirtschaftlichem oder rein fachlichem Sinn, ohne direkten Bezug zu Softwaresystemen. Diese Prozesse haben jeweils einen spezifischen Informationsbedarf.
- Die nächste Ebene der Pyramide, die Informationsarchitektur, definiert die Struktur und Zusammenarbeit aller IT-Systeme einer Organisation (auch IS-Portfolio[1] oder *Anwendungslandschaft*) genannt. Sie stellt die wesentlichen Informationsflüsse und Schnittstellen zwischen sämtlichen Softwaresystemen einer Organisation dar. Die Informationsarchitektur stillt den Informationsbedarf der Geschäftsprozesse. Auf dieser Ebene tummeln sich die Enterprise-IT-Architekten, von denen in diesem Kapitel die Rede sein wird.

[1] Informationssystem-Portfolio, nach [Dern09].

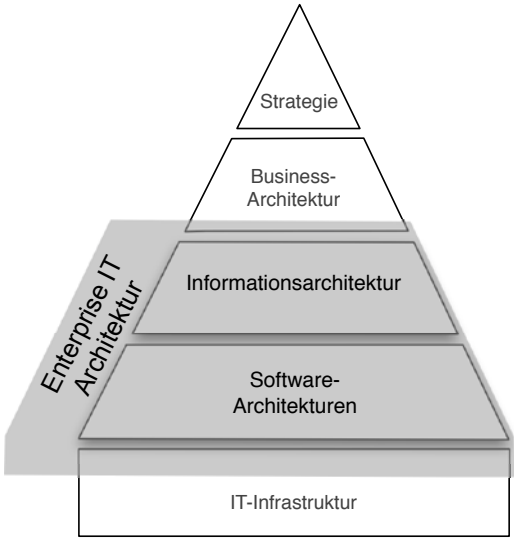


BILD 10.1
Architekturpyramide
(in Anlehnung an [Dern 09])

- Unterhalb der Informationsarchitektur liegt die Ebene der Softwarearchitekturen. Sie bildet das Kerngebiet dieses Buches und beschäftigt sich mit Entwurf und Implementierung einzelner Softwaresysteme. Hierzu gehören für die IT-Systeme einer Organisation jeweils die unterschiedlichen Sichten (Bausteine, Abläufe, Verteilung).[2] Der Gesamtkontext der einzelnen Softwarearchitekturen entstammt den darüber liegenden Ebenen der Informations- sowie Business-Architektur.
- Den Fuß der Pyramide bildet die IT-Infrastruktur. Deren *Architektur* dreht sich um Hardware, Netztopologie, Betriebssysteme, Datenbanksysteme und Middleware. Entscheidungen oder Randbedingungen dieser Infrastruktur betreffen oftmals unmittelbar die darüber liegende Ebene der Softwarearchitekturen. Die „Bausteine" der IT-Infrastruktur sind so konkret, dass Sie sie sogar anfassen können ...

10.1 Wozu Architekturebenen?

Inwiefern hilft die oben dargestellte Pyramide unserem Verständnis von Architekturebenen? Auch wenn einzelne Softwareprojekte komplex und schwierig bleiben, klärt sie aus meiner Sicht einige wesentliche Eckpunkte:

- Im Umfeld mittlerer und großer Organisationen oder Unternehmen geht es heute insbesondere um Komplexitätsmanagement großer Anwendungslandschaften. Dabei hilft insbesondere die Ebene „Informationsarchitektur" als Vermittler zwischen der Business- und der IT-Sicht.

[2] Von Gernot Dern stammt die Anregung, diese Ebene *Solution Architectures* zu nennen, um ihren Charakter genauer zu beschreiben: Sie stellen *Lösungen* für den Informationsbedarf der oberen Ebenen der Architekturpyramide dar.

- Es gibt Strukturen oberhalb einzelner Softwaresysteme. Diese Informations- oder Business-Architekturen geben wesentliche Einflussfaktoren und Randbedingungen (siehe hierzu Kapitel 3.3) vor, die den Entwürfen einzelner Systeme oft sehr konkrete und enge Grenzen setzen.
  - Durch diese Strukturen im Großen können Unternehmen die finanziellen und personellen Aufwände für ihre Informatik-Unterstützung beeinflussen.
  - Weiterhin können Unternehmen diese Strukturen kontrolliert vereinfachen und insofern die Gesamtkomplexität ihrer IT reduzieren.
- Einzelne Softwaresysteme bilden die Bausteine der höheren Abstraktionsebene „Informationsarchitektur". Dort gelten ähnliche Strukturierungs- oder Entwurfsprinzipien wie bei einzelnen Softwaresystemen: Schnittstellen sollten klar definiert und Abhängigkeiten bewusst entschieden werden. Leider mangelt es vielen Organisationen an dieser „Struktur im Großen" – mit teilweise dramatischen Konsequenzen hinsichtlich der Gesamtkosten der IT. Auch hierauf gehe ich in diesem Buch nicht weiter ein.
- Die Strategie eines Unternehmens steht in einem klar definierten Zusammenhang mit Geschäftsprozessen und somit auch mit der IT – zumindest sollte dies der Fall sein.

Erst in den letzten Jahren haben Unternehmen begonnen, die Strukturen oberhalb einzelner Systeme (Informations- und Business-Architektur) wirklich ernst zu nehmen. Fortschrittliche Organisationen, insbesondere solche mit vielen IT-Systemen, schufen die Positionen oder Teams für Enterprise-IT-Architektur.

Bei der Definition von Aufgaben dieser Gruppe von Architekten hilft die Pyramide aus Bild 10.1 deutlich weiter.

10.2 Aufgaben von Enterprise-Architekten

Über Rolle und Aufgaben von Enterprise-Architekten gibt es sowohl in der Praxis als auch in der Literatur sehr unterschiedliche Ansichten. Einen guten Einstieg mit klarer Begriffsbildung finden Sie in den empfehlenswerten Büchern [Dern09] und [Keller12]. Ich möchte Ihnen hier einige der Kernaufgaben vorstellen, ohne dabei detailliert auf die Schnittstellen zur Businessarchitektur oder Softwarearchitektur genauer einzugehen.

10.2.1 Management der Infrastrukturkosten

Die IT-Infrastruktur weist ein erhebliches Potenzial zur Reduktion von IT-Ausgaben (sprich: Geld!) auf. Deswegen nenne ich sie als erste Aufgabe, obwohl sie sich methodisch kaum fassen lässt. Von der Server-Konsolidierung über Lizenzmanagement bis hin zum Scorecarding (Vergleich mit den Infrastrukturkosten anderer Unternehmen) reicht die Bandbreite möglicher Tätigkeiten.

10.2.2 Management des IS-Portfolios

Enterprise-IT-Architekten kümmern sich um das Informationssystem-Portfolio ihrer Organisation (auch Anwendungslandschaft oder IS-Portfolio genannt).

Der Begriff IS-Portfolio bezeichnet die Gesamtheit aller Softwaresysteme einer Organisation. In der Realität können das mehrere Hundert bis mehrere Tausend[3] verschiedene Programme sein – mit all ihren Schnittstellen, gegenseitigen Abhängigkeiten, Releasezyklen und Technologien. Bei größeren Unternehmen (Beispiel: Finanzdienstleister, Versicherungen, Automobilhersteller) erfordern Unterhalt und Management dieses IS-Portfolios ein jährliches Budget von mehreren Hundert Millionen Euro, bei einigen sogar bis in den Milliardenbereich. Sie ahnen richtig: Eine anspruchsvolle Aufgabe – denn sie verbindet fachliche mit technischen und organisatorischen Herausforderungen.

Aber eins nach dem anderen – lassen Sie mich mit den *Anforderungen* an das IS-Portfolio beginnen.

Welche Anforderungen bestimmen das IS-Portfolio?

Nach welchen Anforderungen und Kriterien entwickeln und planen Enterprise-IT-Architekten das IS-Portfolio ihrer Organisationen? Woher stammen ihre Anforderungen, was sind ihre Einflussfaktoren und Randbedingungen?[4]

In erster Linie bestimmen die geschäftlichen Anforderungen der Business-Architektur das IS-Portfolio (werfen Sie einen Blick auf die Pyramide aus Bild 10.1). Dazu gehören neue Dienstleistungen des Unternehmens, neue Geschäftsprozesse oder Veränderungen des geschäftlichen Umfeldes.

Außerdem bestimmen technische oder strategische Überlegungen das IS-Portfolio. Einige Beispiele:

- Die künftige Ausrichtung an Open-Source-Middleware oder Datenbanken.
- Die Konzentration des IT-Betriebs auf genau zwei unterstützte Betriebssysteme statt der bisherigen vier. Hierbei gilt es beispielsweise, passende Migrationswege für die betroffenen Systeme oder Projekte zu finden.
- Die Fusion mit einem anderen Unternehmen und die damit verbundene Konsolidierung der gesamten IT.

In den letzten Jahren haben zunehmend auch gesetzliche (regulatorische) Rahmenbedingungen für Anpassungen im IS-Portfolio gesorgt – so etwa die gesetzlichen Nachweispflichten oder die Vorratsdatenspeicherung.[5]

Enterprise-IT-Architekten können nicht wie Softwarearchitekten auf fertige Pflichtenhefte oder Anforderungsanalysen zurückgreifen, sondern müssen die Anforderungen an das IS-Portfolio kontinuierlich selbst weiterentwickeln – wobei ihnen Vorstände, Geschäftsführer, RZ-Leiter und andere wichtige Personen kräftig „dreinreden“ werden.

---

[3] Einige deutsche Automobilhersteller verfügen über mehr als 3000 (dreitausend) verschiedene eigenentwickelte (Server- oder Mainframe-)IT-Systeme.

[4] Diese Begriffe haben Sie im Zusammenhang mit Softwarearchitektur in Abschnitt 3.3 kennengelernt.

[5] Beispiele: SOX, Basel-II, Solvency-II, ITIL/Cobit.

Was gehört zur Planung eines IS-Portfolios?

Wie Softwarearchitekten die Abhängigkeiten ihrer „Bausteine“ (Pakete, Komponenten, Klassen, Module) unter Beachtung von Entwurfsprinzipien und konkreten Anforderungen bewusst entwerfen, so sollten Enterprise-IT-Architekten auf ihrer (höheren) Abstraktionsebene einer ähnlichen Systematik folgen:

- Sie entwerfen Schnittstellen für andere Softwaresysteme.
- Sie planen die Reihenfolge und Gruppierung von Entwicklungs- oder Wartungsprojekten. Damit bringen sie Aktivitäten einzelner Entwicklungsprojekte in eine zeitliche Ordnung und setzen Prioritäten für einzelne Entwicklungsaufgaben. Als Ergebnis entstehen oftmals langfristige *Architekturpläne*, ähnlich dem Beispiel aus Bild 10.2.
- Sie geben technische Vorgaben an Entwicklungsprojekte oder -abteilungen; in Bild 10.2 beispielhaft etwa die Verwendung des MDA-Generators AndroMDA 4.0 ab Mitte Q1-2008 oder der Einsatz von MySQL 5 ab ca. Ende Q2-2008.
- Sie unterscheiden zwischen Projekten zur Anwendungs- und Infrastrukturentwicklung.
- Sie bewerten das IS-Portfolio und definieren Optimierungsmaßnahmen.

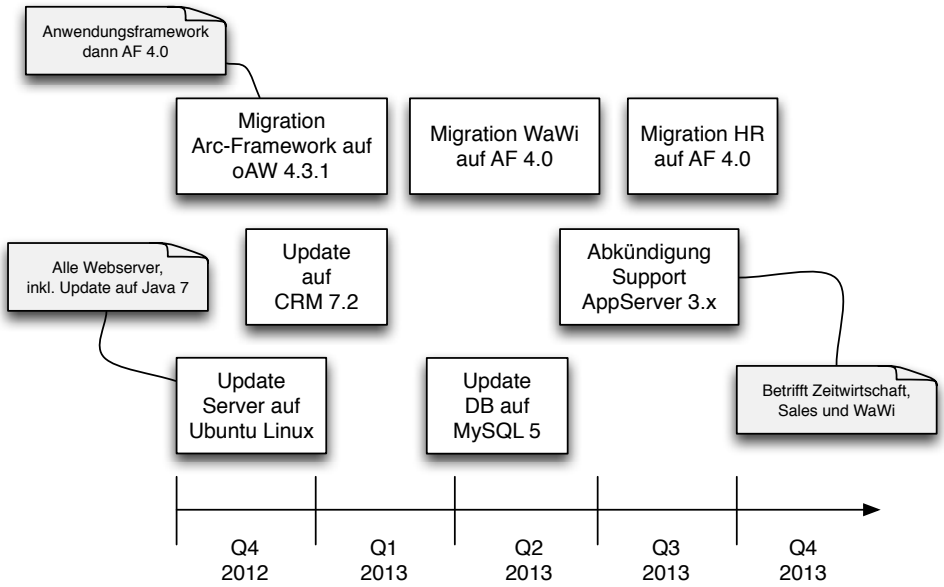


BILD 10.2 Beispiel eines Architekturplans aus dem IS-Portfoliomanagement

10.2.3 Definition von Referenzarchitekturen

Zur Vereinheitlichung und Homogenisierung der gesamten Anwendungslandschaft eines Unternehmens helfen *Referenzarchitekturen* (auch *Blueprints* genannt). Sie geben vor, nach welchen Maßgaben einzelne Systeme zu strukturieren sind, welche Technologien und Produkte eingesetzt werden sollten und wie der Betrieb dieser Anwendungen stattfinden soll.

Projekte können sich an solchen Referenzarchitekturen orientieren und erheblichen Entwurfs- beziehungsweise Evaluierungsaufwand ersparen. Referenzarchitekturen bilden die Grundlage für unternehmensinterne Wiederverwendung von Softwarebausteinen.

Was ist eine Referenzarchitektur?

Lassen Sie mich ein wenig genauer erklären, was eine Referenzarchitektur ausmacht. Dazu verwende ich die Definition aus [Dern09]:

> *Eine Referenzarchitektur ist eine IT-Architektur, die standardisierend für die IT-Architekturen einer Gruppe von Informationssystemen wirkt.*

Dazu gehören folgende Bestandteile:

- Die Definition des Einsatzbereiches dieser Referenzarchitektur: Für welche Art von Systemen gelten ihre Vorgaben – und in welchen Fällen nicht?
- Die (abstrakten) Strukturen, insbesondere Baustein- und Laufzeitstruktur. Sie klären, welche grundsätzlichen Bausteine in diesem Typus von Anwendungen vorhanden sein müssen, welche Schnittstellen sie haben und wie sie zur Laufzeit zusammenwirken.
- Technische Vorgaben, wie etwa die Definition der erlaubten Hardware, Datenbanken, Middleware oder Betriebssysteme.
- Optional können auch Vorgaben zu Entwicklungsprozessen zur Referenzarchitektur gehören, wenn Sie beispielsweise bestimmte Teile Ihrer Web-Entwicklung grundsätzlich als Outsourcing-Aufträge vergeben.

- Einige meiner Kunden haben gute Erfahrung damit gemacht, lediglich *technische* Aspekte wie Persistenz oder Web-Oberflächen als Referenzarchitektur zu beschreiben. Das hilft insbesondere im Zusammenspiel mit MDA (siehe Kapitel 9) weiter, weil über einen Generator dann die entsprechenden Teile der Referenzarchitektur automatisch erzeugt werden können.

Die Referenzarchitekturen der Enterprise-Architekten geben nach [Dern09] in der Regel keine konkreten Komponenten zur Wiederverwendung vor. Sie stellen keine unmittelbar einsetzbaren *Frameworks* oder *Bibliotheken* bereit.[6]

Wie entstehen Referenzarchitekturen?

Eine erste Antwort dazu: Referenzarchitekturen entstehen genau wie gute Entwurfsmuster, niemals nur auf dem Papier. Sie sind aus positiven Erfahrungen abgeleitet und abstrahieren von „guten“ Systemen.

[6] Hingegen verwenden viele Softwarearchitekten den Begriff *Referenzarchitektur* im Sinne der Vorgabe konkreter Komponenten oder Implementierungen.

- Untersuchen Sie innerhalb Ihrer Organisation systematisch und umfassend, welche Projekte oder Lösungsansätze besonders gut funktioniert haben oder besonders kostengünstig arbeiten.
- Misstrauen Sie „theoretischen" Referenzarchitekturen oder Blaupausen aus sekundären Quellen. Systeme müssen in Ihrer Organisation funktionieren, und das können Sie nur unter den dortigen spezifischen Bedingungen nachweisen.
- Misstrauen Sie insbesondere den „pauschalen Modetrends" der Informatik: Mal sollte CORBA alles verbessern, mal half Objektorientierung aus dem Schlamassel, dann EAI, gefolgt von SOA. All diese Ansätze haben ihre Stärken – sind jedoch als „pauschale" Referenzarchitektur nach meiner Erfahrung kontraproduktiv. Letztlich hilft nur ein systematischer Entwurf spezifischer Strukturen.*

* Wie ich ihn in diesem Buch ja durchgängig propagiere (aber das ist Ihnen sicherlich schon aufgefallen!).

Wählen Sie funktionierende operative Systeme als Vorbilder, und leiten Sie daraus Referenzarchitekturen ab. Möglicherweise gibt es mehrere Referenzarchitekturen, beispielsweise eine für Web-Systeme, eine zweite für interne Backend-Systeme.

Und nun die zweite (traurige) Antwort: Referenzarchitekturen zu entwickeln, ist schwierig. Der Grund: Die Erstellung wiederverwendbarer Komponenten oder Strukturen ist schwer – und meiner Erfahrung nach mindestens zwei- bis dreimal aufwendiger als die Entwicklung einer Einmal-Lösung.[7] Lassen Sie sich nicht entmutigen – langfristig lohnt sich dieser Aufwand, sowohl für die Entwicklung als auch für den Betrieb neuer Systeme.

10.2.4 Weitere Aufgaben

Um für Enterprise-IT-Architekten keine Langeweile aufkommen zu lassen, gebe ich Ihnen noch einen kleinen Ausblick auf einige zusätzliche Aufgaben (Details finden Sie in [Dern09] sowie [Keller12]):

- Mitwirkung bei der Formulierung und Weiterentwicklung der IT-Strategie.
- Entwicklung der Infrastruktur und des Systembetriebs.
- Kartografie und Kommunikation der Informationsarchitektur, d. h. modellhafte Darstellung sämtlicher Softwaresysteme, Schnittstellen sowie deren Abbildung auf Geschäftsprozesse.
- Umsetzung von IT-Governance. Falls Sie diesen Begriff hier zum ersten Mal lesen, schauen Sie zur Vertiefung mal auf [Cobit]. Es geht um die Regierung der Informatik im Unternehmen, letztlich den Nachweis, dass IT-Budgets bestmöglich zum Wohle des Unternehmens eingesetzt werden.
- Management von IT-Budgets und Investitionen.

[7] Andererseits habe ich niemals behauptet, Architektur sei einfach. ☺

Tipps für Enterprise-Architekten

- Homogenisierung und Standardisierung schaffen Übersichtlichkeit und reduzieren Komplexität – das sollten Ihre wesentlichen Ziele sein, zumal Ihre Betriebs- und Entwicklungskosten dadurch mittelfristig deutlich sinken werden. Vorsicht: Rechnen Sie mit starkem Widerstand bei der Einführung und Durchsetzung von Standards – es drohen Glaubenskriege und Philosophenstreit. Daher benötigen Sie neben Überzeugungsfähigkeit auch die „Insignien der Macht“*, um Standards durchzusetzen.
- Zu viel Standardisierung kostet unverhältnismäßig hohen Planungs- und Kontrollaufwand. Lassen Sie ruhig Abweichungen zu, solange Sie wissen, warum und zu welchem Preis. Kein Mensch möchte in einem Reinraum leben – ein wenig Unordnung ist gemütlich ...
- Kartografieren Sie Ihre gesamte IT-Landschaft. Dazu setzen Sie dieselben Sichten und Diagrammarten ein wie für einzelne Softwaresysteme (siehe Abschnitt 4.4 ff.). Der einzige Unterschied liegt im Abstraktionsgrad der einzelnen Bausteine. Hüten Sie sich vor völlig anderen Diagrammarten oder Werkzeugen – ansonsten drohen Mehraufwand und Redundanz.
- Das primäre Ziel von Enterprise-IT-Architekten besteht in der Reduktion von Komplexität. Manche Manager wünschen sich von ihnen hauptsächlich Kostensenkung, teilweise durch „Abschalten“ bestehender Systeme. Als Enterprise-IT-Architekt haben Sie aber bereits hervorragende Arbeit geleistet, wenn die Komplexität Ihrer Anwendungslandschaft nicht weiter ausufert.**

* Die Komplexität von Anwendungslandschaften steigt so lange, bis Sie einen signifikanten Aufwand in deren Reduktion investieren (sprich: Enterprise-IT-Architektur betreiben). Komplexität sinkt niemals von alleine – insbesondere nie durch vorschnelle Management-Entscheidungen (aber das dürfen Sie Ihren Managern in dieser Form nicht sagen ...).

** Mit diesen Insignien meine ich beispielsweise den Rückhalt der Linienmanager, des IT-Vorstandes, des Rechenzentrumsleiters und der übrigen wesentlichen Manager Ihrer IT-Welt.

Fazit

Unternehmen setzen oft zahlreiche unterschiedliche Softwaresysteme ein – diese „IT-Landschaft“ benötigt Planung und Struktur. Wie sich Softwarearchitekten um Struktur und Qualität einzelner Softwaresysteme kümmern, so Enterprise-IT-Architekten um die Struktur und Qualität der gesamten IT-Landschaft von Unternehmen. Sie sorgen für eine möglichst reibungslose Unterstützung von Geschäftsprozessen durch Softwaresysteme – neudeutsch *Business-IT-Alignment*.

Ihre Rolle verspricht einen hohen Nutzen für Unternehmen, sowohl finanziell als auch bezüglich der gesamten Effizienz.

Viele etablierte Grundsätze des Software-Engineering und der Softwarearchitektur gelten auch auf dieser hohen Abstraktionsebene – allerdings gehört neben einem fundierten Verständnis von Software auch ein ebenso guter Einblick in die fachlichen Prozesse und Unternehmensabläufe zu den Voraussetzungen erfolgreicher Enterprise-IT-Architekten.

10.3 Weiterführende Literatur

[Dern09] beschreibt das Vorgehen beim IT-Architekturmanagement. Der Autor demonstriert anhand des Beispiels einer (hypothetischen) Versicherung Aufgaben und mögliche Ergebnisse des Architekturmanagements.

[Keller12] stellt die IT-Unternehmensarchitektur aus dem Blickwinkel der verantwortlichen Architekten und Manager dar. Das Buch fokussiert auf methodische Aspekte, mit zahlreichen praktischen Hinweisen zu wichtigen Themen wie IT-Strategie, Governance, Anwendungsportfolios, Budgets sowie der Einführung von IT-Architektur in Unternehmen.

[Ross+06] stammt aus der Feder einiger renommierter Hochschullehrer. Sie erläutern, wie und warum eine IT-Strategie letztlich zu einer verbesserten *Performance* des gesamten Unternehmens führt; richtet sich an IT-Topmanager, enthält Ratschläge zur Verbesserung des ROI für IT-Investitionen sowie zur Senkung der IT-Kosten und der IT-Risiken. Staubtrocken und sehr abstrakt.

Das SEBIS-Projekt der TU München beschäftigt sich mit der Kartografie von Unternehmensarchitekturen. Abschnitt 4.8 aus [Keller12] enthält eine Einführung.

Danke an Gernot Dern und Wolfgang Keller für den gründlichen Review dieses Kapitels.

11 Beispiele von Softwarearchitekturen

Dieses Kapitel dokumentiert die Architekturen zweier realer Softwaresysteme gemäß den methodischen und strukturellen Ansätzen aus Kapitel 4. Es orientiert sich dabei an der arc42-Strukturvorlage, die ich Ihnen in Abschnitt 4.9.1 vorgestellt habe.

Anmerkung zur Nummerierung: Hier habe ich die „normale" Überschriftenzählung dieses Buches in den Unterkapiteln außer Kraft gesetzt und nummeriere die Abschnitte exakt so, wie es in einer *echten* Architekturdokumentation der Fall wäre.

- In Abschnitt 11.1 zeige ich Ihnen ein System zur Datenmigration im Finanzbereich. Sie erleben einen auf Performance optimierten Datenfluss-Architekturstil, ähnlich einer Pipes-und-Filter-Architektur.
- Abschnitt 11.2 dokumentiert eine Produktfamilie aus dem Umfeld der *Customer Relationship Management*-(CRM)-Systeme.

Beide Systeme gibt es wirklich. Für die Darstellung hier im Buch habe ich von der Realität stellenweise stark abstrahiert und viele Details aus didaktischen Gründen vereinfacht.

11.1 Beispiel: Datenmigration im Finanzwesen

1 Einführung und Ziele

Zweck des Systems

Dieses Dokument beschreibt die Softwarearchitektur des M&M[1]-Systems zur Migration von ca. 20 Millionen Personen- und Kontodaten der Firma Fies Teuer AG, einer Organisation aus der Finanzdienstleistung.[2]

Der Auftraggeber von M&M betreibt seit etwa 1970 einige Mainframe-Anwendungen (Cobol, VSAM) zur Pflege von Personen-, Konto- und Bankdaten. Diese Systeme werden zurzeit durch eine homogene Java-Anwendung abgelöst, die durch ein anderes, parallel zu M&M laufendes Projekt entwickelt wird.

Fies und Teuer AG hat in Deutschland ca. 20 Millionen Kunden und pflegt für diese Kunden insgesamt mehr als 50 Millionen Konten. Kunden können natürliche oder juristische Personen sein, teilweise auch andere Organisationen (Verbände, Vereine etc.).

Konten enthalten neben reinen Buchungsinformationen auch statistische oder sonstige finanzbezogene Informationen zu den Kunden oder mit ihnen assoziierten Personen (z. B. Ehepartner) oder Organisationen (z. B. von ihnen geführte Unternehmen).

Die inhaltliche Bedeutung dieser Konto- und Buchungsinformationen sowie die darin enthaltenen Attribute haben über die Betriebszeit der bisherigen Anwendung stark gewechselt. Zur Umstellung auf das neue konsolidierte Objektmodell hat ein Team von ca. 20 Fachexperten eine Menge von mehreren Hundert fachlichen Regeln aufgestellt, nach denen die Migration ausgeführt werden muss.

Sämtliche bestehenden Daten aus dem bisherigen Format (VSAM-Dateien, EBCDIC-Codierung) müssen in dieses Java-Objektmodell migriert werden. Diese Migration ist Aufgabe des hier beschriebenen Systems.

Ausgangssituation der bestehenden Daten

- Früher waren die Daten anhand der „Konten“ organisiert.
  - Für jegliche Operationen mussten Sachbearbeiter zuerst die betroffenen Konten identifizieren, erst danach konnten fachliche Operationen (Buchungen, Auskünfte) für die betroffenen Personen ausgeführt werden.
  - Im Zuge der steigenden Kundenorientierung der Fies und Teuer AG sollen die Daten künftig anhand von Personen organisiert werden.
- Die Ausgangsdaten liegen in Form verschiedener Dateien auf Bändern vor, deren Einträge nach fachlichen Kriterien einander zugeordnet werden müssen.

---

[1] M & M steht für **M**igration von **M**assendaten.

[2] Wie Sie leicht erraten können, habe ich den Namen dieser Organisation bewusst verändert ...

- Früher geltende Schlüssel- und Kennzahlensysteme müssen in neue Schlüssel überführt werden. Beispiel: Die alte Tarifklasse „T13“ wird zur neuen Tarifklasse „Jährliche Zahlung ohne Skonto“.

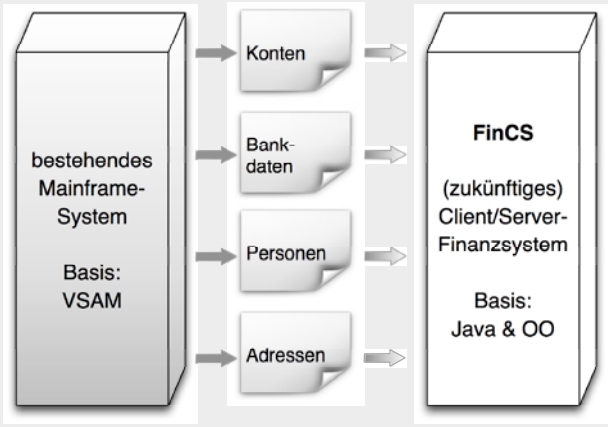


BILD 11.1
Zweck des Systems: Migration bestehender Daten

Leserkreis

- Alle in Abschnitt 1.3 dieser Dokumentation genannten Stakeholder von M&M.
- Software-Entwickler, Architekten und technische Projektleiter, die Beispiele für eine Architekturdokumentation (auf Basis des arc42-Templates) suchen.
- Mitarbeiter von IT-Projekten, die sich das Leben in der Softwarearchitektur vereinfachen, indem sie auf ein bewährtes Template zur Dokumentation zurückgreifen.

1.1 Fachliche Aufgabenstellung

Die wichtigsten funktionalen Anforderungen:

- Bestehende Kunden-, Konto- und Adressdaten sollen von den bisherigen VSAM[3]-basierten Mainframe[4]-Programmen aus dem EBCDIC-Format in das Objektmodell von FINCS, einer in Entwicklung befindlichen Java-Anwendung, migriert werden.
- Die Ausgangsdaten liegen in Form verschiedener Dateien oder „Bänder“ vor, deren Einträge nach fachlichen Kriterien einander zugeordnet werden müssen.
- Früher waren die Daten nach *Konten* organisiert, im neuen System sind *Personen* der Bezugspunkt.
- Teile der früheren Schlüssel- oder Kennzahlensysteme sollen in neue Schlüssel überführt werden.

---

[3] VSAM = Virtual Storage Access Method; eine Zugriffsmethode für von IBM-Großrechnern verwendete Dateien. VSAM-Dateien bestehen aus einem Metadatenkatalog sowie mindestens einer physischen Datei. Mehr zu VSAM in den Literaturhinweisen am Ende dieses Kapitels. Übrigens gibt es das Gerücht „VSAM ist grausam“.

[4] Eine aktuelle Einführung in OS/390 finden Sie unter *http://www.informatik.uni-leipzig.de/cs/esvorles/index.html*

1.2 Qualitätsziele

Die primären Qualitätsziele von M&M lauten:

- Effizienz (Performance): Migration von ca. 20 Millionen Personen- und Kontodaten innerhalb von maximal 24 Stunden.
- Korrektheit: Die Migration muss revisionssicher und juristisch einwandfrei erfolgen. Hierzu sind geeignete Maßnahmen zur Fehlervermeidung und -erkennung nötig.

Nicht-Ziele

Was M&M nicht leisten soll:

- Änderbarkeit oder Flexibilität der fachlichen Transformationsregeln – die Migration ist einmalig.
- Es bestehen keine besonderen Anforderungen an Sicherheit – M&M wird nur ein einziges Mal produktiv betrieben, und das innerhalb eines gesicherten Rechenzentrums.

1.3 Stakeholder

Rolle	Beschreibung	Ziel
Management der Firma Fies und Teuer AG	Projektmanagement, Lenkungsgremium	Reibungslose und fehlerfreie Migration. Keine negative Presse.
Endkunden der Firma	20 Millionen Kunden. Treten im Projektverlauf nicht in Erscheinung, sind nur indirekt beteiligt.	Korrekte Migration ihrer Daten.
Revision oder Buchprüfung	Interne Revisionsabteilung, die (inhaltliche und juristische) Korrektheit der Migration durch Stichproben validieren wird.	Einfacher Zugang zu Stichproben. Korrekte Migration.
Projekt FINCS	Das Entwicklungs- und Architekturteam des künftigen Client/Server Finanzsystems.	Flexibilität beim geplanten Objekt- und Datenmodell.
Boulevard-presse	Legt besonderes Augenmerk auf Firma Fies und Teuer AG. Fehlverhalten wird gnadenlos publiziert.	Migrationsfehler, „Whistleblower“ im Projekt finden.

2 Einflussfaktoren und Randbedingungen

2.1 Technische Einflussfaktoren und Randbedingungen

Randbedingung	Erläuterung
Hardware-Infrastruktur	IBM Mainframe als Plattform des Altsystems, Sun Solaris Cluster für das Zielsystem
Software-Infrastruktur	▪ Sun Solaris als Betriebssystem der Zielumgebung ▪ Oracle als neue Datenbank ▪ JEE-kompatibler Applikationsserver als mögliche Betriebsumgebung
Ausgangsdaten in EBCDIC	Ausgangsdaten liegen in EBCDIC-Kodierung auf vier getrennten Bändern vor. Eine Vorsortierung durch die Fies und Teuer AG ist nicht möglich.
Systembetrieb	Batch
Grafische Oberfläche	keine, Bedienung kann per Konsole erfolgen
Programmiersprachen	Aufgrund des vorhandenen Know-hows des Entwicklungsteams soll in Java programmiert werden.
Analyse- und Entwurfsmethoden	objektorientiertes Vorgehen
Datenstrukturen	Objektmodell der Zielumgebung bekannt (Bestandteil eines anderen Projektes)

2.2 Organisatorische Einflussfaktoren

Die Fies und Teuer AG ist als juristisch penibler Auftraggeber bekannt. Das Management versteht rein gar nichts von IT (und gerüchteweise auch nichts von Geld, aber das ist eine andere Geschichte).

Die verworrenen Eigentumsverhältnisse der Fies und Teuer AG lassen komplizierte Entscheidungsstrukturen befürchten.

Fies und Teuer AG hat langfristige Lieferverträge mit diversen IT-Dienstleistern abgeschlossen, die eine marktorientierte und freie Auswahl eventuell benötigter externer Mitarbeiter methodisch verhindern.

Eine hochgradig effektive externe Qualitätssicherung fordert eine umfangreiche Dokumentation (und prüft diese sogar auf inhaltliche Korrektheit – in der IT-Branche eher unüblich, doch ist das ebenfalls eine andere Geschichte).

3 Kontextabgrenzung

Fachlicher Kontext

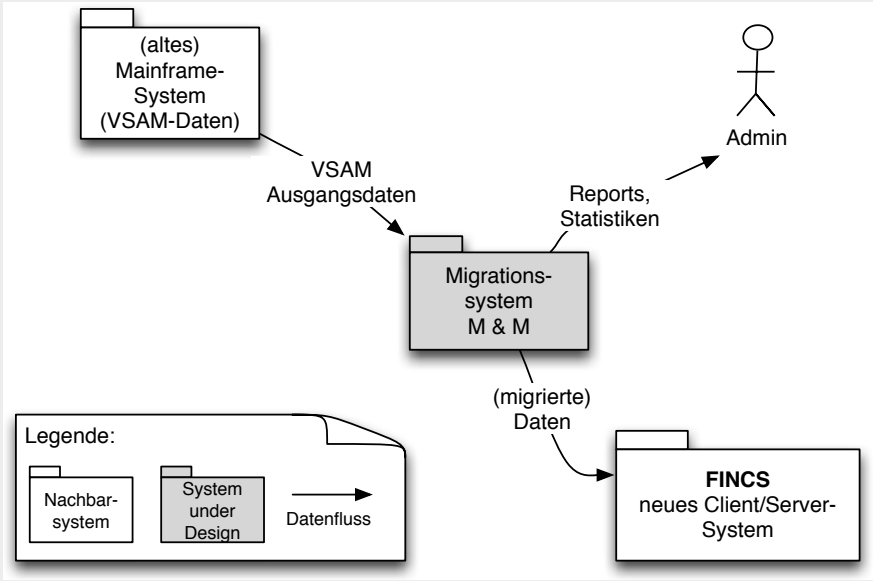


BILD 11.2 Fachlicher Kontext

Kurzbeschreibung der externen Schnittstellen

Schnittstelle	Beschreibung	Technologie
VSAM-Daten	Vier Arten von Daten (Personendaten, Adressdaten, Bankdaten, Kontodaten)	Bänder, LTO-1, siehe Abschnitt 3.2.1
Migrierte Daten	Gemäß des vom FINCS-Projekt erstellten Objektmodells	Übergabe des Objektgraphen über eine Session-Bean
Reports, Statistiken	Während der Migration: Kontinuierliche Ausgabe der Anzahl migrierter Personen. Nach der Migration: Ausgabe der Anzahl von Fehlersätzen	Konsole

Technischer oder Verteilungskontext

Die Migration wird auf zwei getrennten Servern ablaufen: ein Migrationsserver zur eigentlichen Datenmigration und Ausführung der technischen und fachlichen Transformationen, ein zweiter als Datenbankserver.

Das neue System (FINCS) wird direkt auf die von der Migration erstellte Datenbank zugreifen. Der Migrationsserver muss daher das künftige Klassen- und Tabellenmodell kennen.

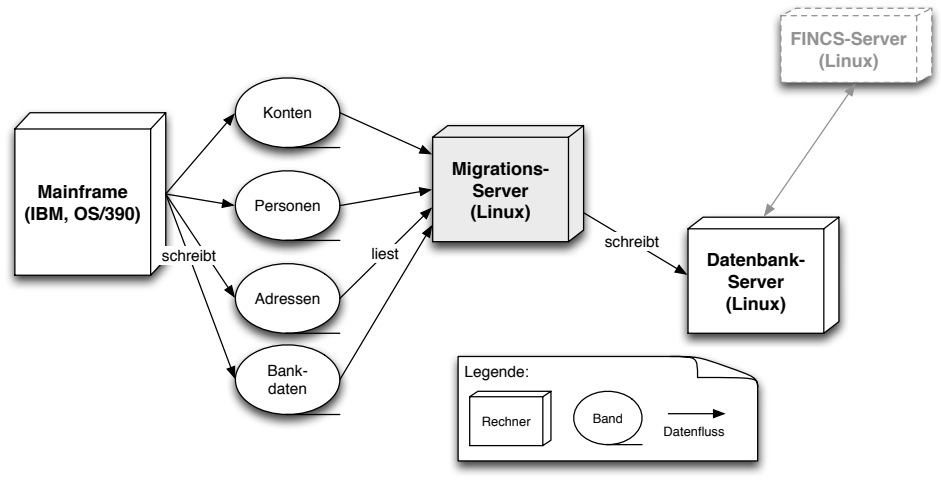


BILD 11.3 Technischer-/Verteilungskontext

Übersicht der Eingangsdaten(-schnittstellen)

Sämtliche Eingangsdaten werden auf jeweils einzelnen LTO-1-Bändern pro Datenart geliefert. Jedes dieser Bänder enthält bis zu 100 Gigabyte an Daten – die genaue Datenmenge kann aufgrund technischer Beschränkungen der Fies und Teuer AG vorab nicht spezifiziert werden.

Eingangsdaten	Beschreibung
Kontodaten	Trotz der Bezeichnung enthält dieses Band die Buchungs-/Bewegungsdaten für sämtliche Konten. Variables Satzformat mit 5–25 Feldern, gemäß FT-Satzart 43.
Personendaten	Stammdaten der von der Fies und Teuer AG betreuten Personen und Organisationen (Unternehmen, Vereine, Stiftungen, Behörden). Variables Satzformat mit 15–50 Feldern, gemäß FT-Satzart 27.
Adressdaten	Adress- und Anschriftsdaten, inklusive Informationen über Empfangsberechtigte, Orts- und Straßenangaben, Angaben über Haupt- und Nebenadressen (Zentralen, Filialen) für Organisationen und Unternehmen. Variables Satzformat mit 5–40 Feldern, gemäß FT-Satzart 33. Mehrere Sätze pro Person möglich.
Bankdaten	Daten externer Banken (Referenz- und Gegenkonten der Personen und Organisationen). Festes Satzformat, aber mehrere Sätze pro Person möglich.

4 Lösungsstrategie

Migration im Batch mit folgenden Kernkonzepten:

- Datenfluss-Architekturstil mit Pipe-und-Filter-Elementen, mit relationaler Datenbank als Pipe (Qualitätsziel: Performance).
- Parallelisierung der rechenintensiven fachlichen Migration (Qualitätsziel: Performance).
- Fehlersensoren in sämtlichen Verarbeitungsschritten, Sammlung sämtlicher erkannter Fehler in der Fehlerdatenbank (Qualitätsziel: Korrektheit).

5 Bausteinsicht

Ausgehend von einer Menge von VSAM-Dateien (geliefert als Bänder), konvertiert („migriert“) die M&M-Anwendung sämtliche Datensätze in das neue Objektmodell.

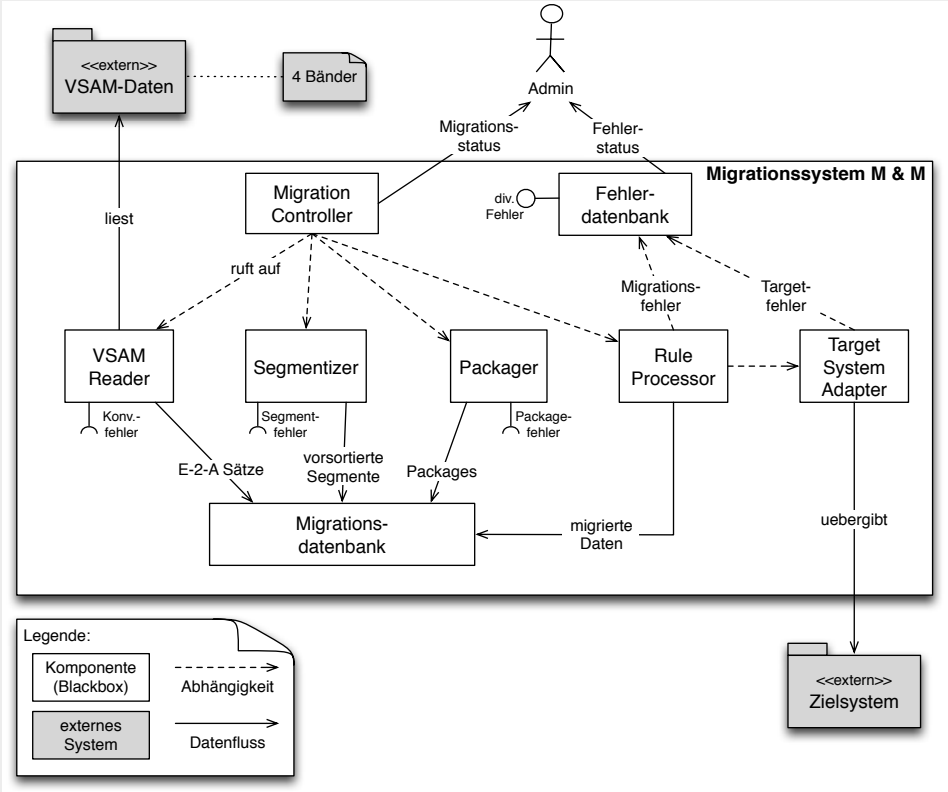


BILD 11.4 Whitebox-Darstellung des Migrationssystems, Level-1

5.1 M&M Bausteinsicht Level 1

Grundlegende Strukturentscheidung: Datenfluss mit Pipes-und-Filters

Zur Lösung des M&M-Migrationsproblems wenden wir ein modifiziertes Pipes-und-Filter-Architekturmuster an, ausführlich beschrieben im Buch „Pattern-Oriented Software Architecture“ von Buschmann et al. ([Buschmann+96]).

Ein wesentliches Entscheidungskriterium für diese Architektur war die Möglichkeit, Teile dieser Bausteine im Bedarfsfall (Performance-Bedarf) zu parallelisieren.

Entgegen dem klassischen Pipe-und-Filter-Muster arbeitet der Eingangsbaustein (VSAM Reader) seine Eingangsdaten erst vollständig ab, bevor der nächste Filter aufgerufen wird.

Baustein	Bedeutung
Migrations-Controller	Koordiniert die übrigen Bausteine und meldet den aktuellen Migrationsstatus an den Admin. Es handelt sich um eine einzelne Java-Klasse mit ausführlicher Ausnahme- und Fehlerbehandlung sowie einer Schnittstelle für Reporting und Auswertungen. Keine weitere Doku.
VSAM-Reader	Führt Formatkonvertierung von EBCDIC nach ASCII/UNICODE durch. Details siehe Abschnitt 5.1.1.
Segmentizer	Vorbereitung der Parallelisierung des „Rule Processors“: Zuordnung der einzelnen Datensätze der unterschiedlichen Datenquellen (Personen-, Konto-, Bank- und Adressdaten) zu zusammengehörigen Datengruppen oder Segmenten. Elemente eines Segments können unabhängig von Inhalten anderer Segmente bearbeitet werden. Notwendig ist die Aufteilung der Eingangsdaten auf mindestens 3–5 verschiedene Segmente. Es geht um eine möglichst *schnelle* Einteilung in Segmente, nicht um Optimierung oder Balancierung der Segmente. Daher liegt das Hauptaugenmerk auf Performance. Keine weitere Doku.
Rule-Processor	Erzeugt aus den formatkonvertierten Alt-Daten einen Objektgraphen gemäß den vom Neusystem FINCS erwarteten Klassenstrukturen. Siehe Abschnitt 5.1.2.
Migrations-Datenbank	Speichert die konvertierten und segmentierten Migrationsdaten zwischen. Dient im Sinne des Pipe&Filter-Architekturmusters als Pipe, d. h. enthält keinerlei eigene Logik. Die Migrationsdatenbank ist eine Menge von Tabellen einer relationalen Datenbank (Person, Konto, Bank, Adresse plus einige Schlüsseltabellen).
Target-System-Adapter	Anbindung an das Zielsystem FINCS. Siehe Abschnitt 5.1.3.

5.1.1 VSAM Reader

- *Zweck/Verantwortlichkeit:* Konvertiert Datensätze aus VSAM-Format (EBCDIC-Codierung) in ein unter Unix direkt verarbeitbares ASCII- oder UNICODE-Format. Gleichzeitig werden die Bestandteile der VSAM-Daten (teilweise besitzen einzelne Bits- oder Bitgruppen in der EBCDIC-Darstellung besondere Bedeutung) in einzeln identifizierbare Datenelemente aufgelöst. Der VSAM Reader führt keinerlei fachliche Prüfungen durch, sondern lediglich Formatkonvertierungen.
- *Schnittstelle(n):* Eingabe von VSAM-Dateien. Für jede Dateiart (Personen, Kontodaten, Adressdaten, Bankdaten) existieren jeweils einzelne Reader-Komponenten, die den Satzaufbau ihrer jeweiligen Eingabedateien kennen.
- *Variabilität:* keine, weil das Gesamtsystem nur einmalig ablaufen soll
- *Offene Punkte:* Für eine spätere Nutzung bei anderen Unternehmen könnten die einzelnen Konverter „pluggable“ gemacht werden. Dies ist zurzeit nicht vorgesehen.

5.1.2 Rule Processor (und Packager)

Zweck/Verantwortlichkeit: erzeugt einen Objektgraphen mit der jeweiligen Person (natürliche oder nichtnatürliche Person) als Wurzelknoten sowie allen zugehörigen fachlichen Objekten als Zweigen (Adressen, Bankverbindungen, Konten, Buchungen).

Der Rule Processor führt damit den eigentlichen fachlichen Migrationsschritt aus der alten (datenartorientierten) Darstellung in die neue (personenorientierte) Darstellung durch. Eventuell sind hierzu mehrere Iterationen über die Regelbasis notwendig.

Beispiele (sind Bestandteil der Anforderungsspezifikation; hier zur Illustration des notwendigen Vorgehens erneut dargestellt):

- Bei der Migration einer verheirateten Frau müssen vorher der Ehepartner sowie gegebenenfalls sämtliche früheren (geschiedenen, verschwundenen oder gestorbenen) Ehepartner migriert werden.
- Bei der Migration von Minderjährigen müssen vorher die natürlichen Eltern, Vormunde und Pflegeeltern migriert werden.
- Bei der Migration von Firmen müssen vorher sämtliche Vorgängerunternehmen, aus denen die aktuelle Firma hervorgegangen ist, migriert werden. Gleiches gilt analog für Vereine oder Verbände.

Damit diese fachlich zusammengehörigen Daten oder Datengruppen durch parallel ablaufende Prozesse migriert werden können, müssen in einem Vorverarbeitungsschritt sämtliche zusammengehörigen Daten als „Paket“ zusammengestellt werden. Diese Aufgabe übernimmt der Packager.

5.1.3 Target System-Adapter

Zweck/Verantwortlichkeit: übernimmt den vom Rule Processor erzeugten Objektgraphen in das Objektmodell des (neuen) Zielsystems. Hierfür sind eventuell kleinere Modifikationen notwendig, je nach endgültig entschiedener Laufzeitumgebung auch nur die abschließende Persistierung.

Nach erfolgreicher Übernahme in das neue System (Zielsystem) liefert der Target System-Adapter eine Meldung zurück – im Fehlerfall wird der gesamte Objektgraph zusammen mit den Fehlermeldungen des Zielsystems in die Fehlerdatenbank geschrieben.

5.2 Bausteinsicht Level 2

Dieser Abschnitt verfeinert die Blackbox-Bausteine der (im vorhergehenden Abschnitt beschriebenen) Ebene 1. Aufgrund des Beispielcharakters dieser Dokumentation sind nur wenige Bausteine detailliert dargestellt.

Anmerkung: Auch in einer echten Dokumentation können Sie sich auf die besonders wichtigen Komponenten konzentrieren. Geben Sie jedoch für sämtliche Komponenten eine Begründung, warum Sie auf eine detaillierte Darstellung verzichten.

5.2.1 VSAM-Reader Whitebox

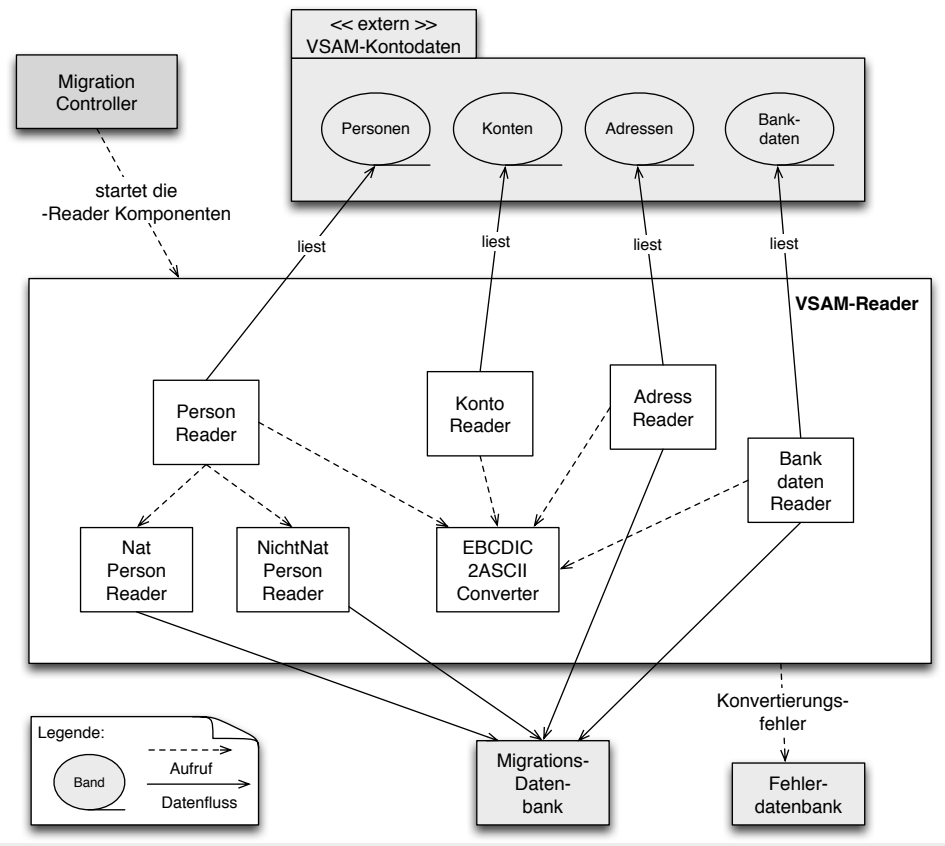


BILD 11.5 Interner Aufbau des VSAM-Readers

Baustein	Bedeutung
EBCDIC2ASCII Converter	Konvertiert eine Zeichenkette von EBCDIC nach ASCII. Als einziger Baustein des Systems in ANSI-C implementiert. Behandelt nur Zeichenketten bis zur Länge von max. 256 Zeichen. Siehe: *http://en.wikipedia.org/wiki/EBCDIC*
Reader-Komponenten	Die Reader-Komponenten lesen jeweils eine bestimmte Datenart aus „ihrem“ Eingabeband. Sie überführen sämtliche darin enthaltenen Daten in eigenständige Attribute. Die einzelnen Reader kapseln die Syntax der „alten“ VSAM-Repräsentation sowie die technischen Details der Datencodierung. Die einzelnen Reader schreiben ihre Ergebnisse in die Migrationsdatenbank, im Fehlerfall rufen sie die Fehlerdatenbank.

5.2.2 Rule Processor Whitebox

Der Rule Processor führt die eigentliche fachliche Migration von Datensätzen durch. Er kombiniert die unterschiedlichen Kategorien von Eingangsdaten (Personen, Konten und Buchungen, Adressen, Bankverbindungen) miteinander und bearbeitet alle fachlichen Sonderfälle. Insbesondere besorgt der Rule Processor die benötigte Reorganisation der Daten, sodass künftig die Personen als Einstiegspunkte in die Navigation dienen, statt, wie früher, die Konten.

Strukturrelevante Entwurfsentscheidungen

- Kein kommerzieller Regelinterpreter: Nach einigen Versuchen mit kommerziellen Regelmaschinen entschied sich das Projektteam gegen den Einsatz einer solchen Komponente. Letztendlich war der Grund, dass die Umsetzung der fachlichen Regeln in eine Regelsprache keine wesentliche Vereinfachung gegenüber der Programmierung in Java gebracht hätte.

  Anmerkung: Aus heutiger Sicht (also posthum ...) würde ich diese Entscheidung nochmals überdenken und insbesondere Open-Source-Regelframeworks[5] auf ihre Tauglichkeit hin überprüfen.
- Keine explizite Regelsprache: Der Ansatz, die Regeln von den Fachexperten formulieren und nur durch einen Interpreter ausführen zu lassen, scheiterte an der Komplexität des gesamten Datenumfelds. Die effiziente und performante Navigation innerhalb des jeweils benötigten Objektraumes war mit einer Regelsprache nicht zu bewerkstelligen. Siehe dazu die Anmerkung zum Regelinterpreter.

[5] Wie beispielsweise JBoss-Drools.

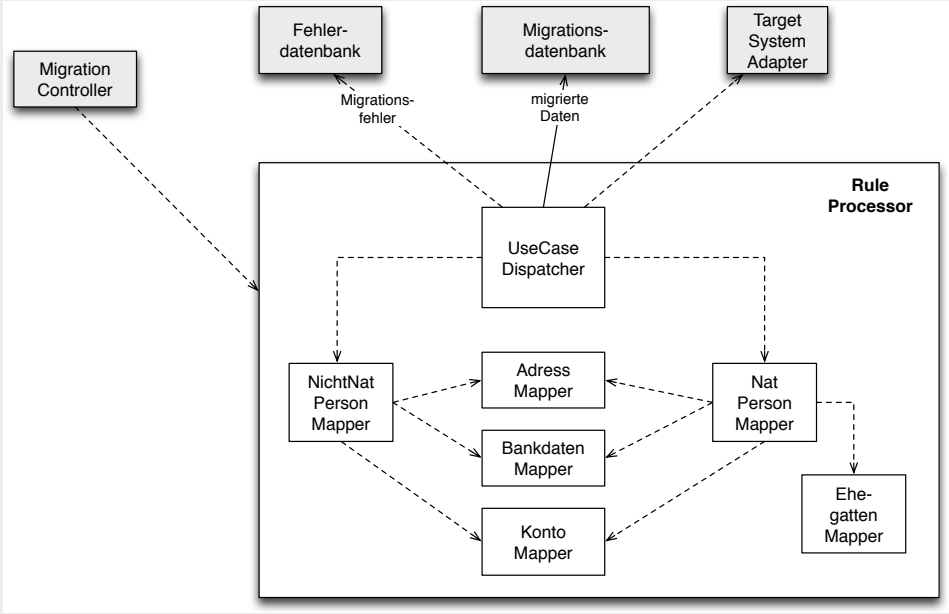


BILD 11.6 Interner Aufbau des Rule Processors

Baustein	Bedeutung
UseCase-Dispatcher	Ruft die eigentlichen fachlichen Migrationskomponenten (Mapper) auf. Siehe Abschnitt 5.2.2.1.
Person-Mapper (NichtNat sowie Nat)	Erzeugt aus den Eingabedaten eine natürliche (NatPerson) oder Nicht-Natürliche.* (NichtNat-)Person im neuen Objektmodell und füllt die Adressdaten, Bankdaten, Konto- und Buchungsdaten entsprechend auf. Handelt es sich um einen Verheirateten, wird die zugehörige Ehefrau (sowie eventuell Ehefrauen aus früheren Ehen) passend migriert. Der Person-Mapper benutzt die anderen Mapper.

* Im Sprachgebrauch dieses Systems sind nicht-natürliche Personen sämtliche Organisationen, beispielsweise Firmen, Vereine, Verbände, Genossenschaften u. Ä.

Anmerkung: Aufgrund des Beispielcharakters dieser Dokumentation sind nachfolgend nur der UseCaseDispatcher sowie der PersonMapper ansatzweise beschrieben. In einer echten Dokumentation sollten Sie alle Bausteine so weit wie nötig detaillieren.

5.2.2.1 UseCase-Dispatcher

Zweck/Verantwortlichkeit: Der UseCase-Dispatcher entscheidet aufgrund der Datenkonstellation, welche der fachlichen Mapper-Komponenten angesprochen werden muss. Diese Entscheidung ist nicht trivial und nicht durch die Mapper selbst zu treffen. Teilweise müssen neben den aktuellen Daten weitere Sätze gelesen werden, um die Entscheidung abschließend treffen zu können.

Beispiele:

- Einpersonengesellschaften als nichtnatürliche Personen können teilweise durch den Personen-Mapper abgebildet werden, abhängig vom Gründungsdatum der Gesellschaft und dem Verlauf der Firmengeschichte (muss immer eine Einpersonengesellschaft gewesen sein und darf nicht aus einer Mehrpersonengesellschaft hervorgegangen sein).
- Natürliche Personen, die in der Vergangenheit Mitinhaber von Mehrpersonengesellschaften gewesen sind und darüber hinaus heute noch als Unternehmer tätig sind, müssen in Abhängigkeit von der Rechtsform der früheren Gesellschaft sowohl als natürliche als auch als nichtnatürliche Person geführt werden. Die Konto- und Buchungsinformationen sind entsprechend aufzuteilen.

<<Weitere Architekturbausteine werden in diesem Beispiel nicht dokumentiert>>

6 Laufzeitsicht

In der ersten Phase lesen die einzelnen Reader des VSAM-Readers alle angelieferten Dateien („Bänder“) und befüllen die Migrationsdatenbank. Danach teilt der Segmentizer die Daten in parallel bearbeitbare Segmente auf.

Der RuleProcessor lässt durch einen Packager (in Bild 11.7 durch den Aufruf „getPackage“ gekapselt) alle für die Migration einer Person oder eines Kontos notwendigen Daten aus der Migrationsdatenbank selektieren und führt anschließend die fachlichen Regeln auf diesem Datenpaket („Package“) aus.

Der TargetSystemAdapter schreibt die migrierten Daten in die neue Datenbank.

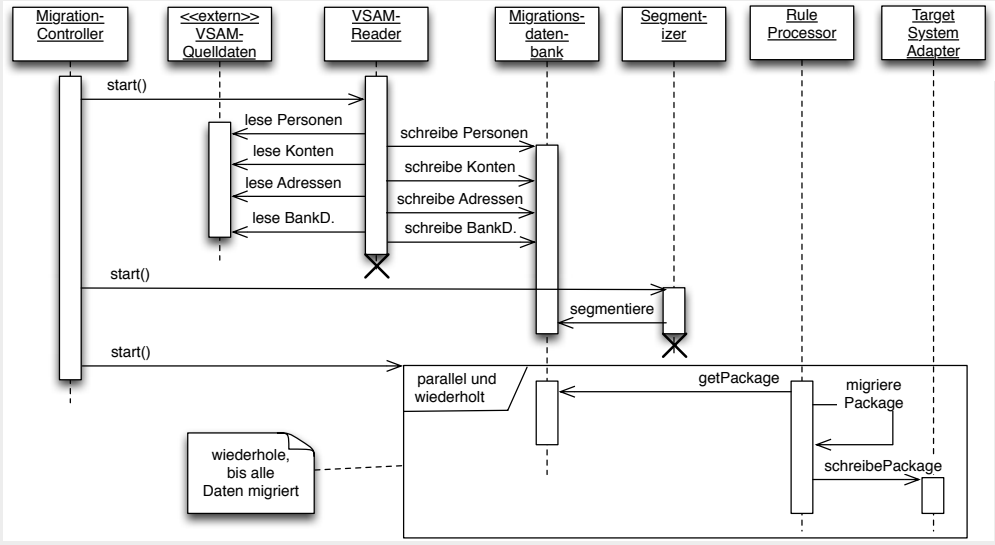


BILD 11.7 Laufzeitsicht der M&M-Migration

7 Verteilungssicht

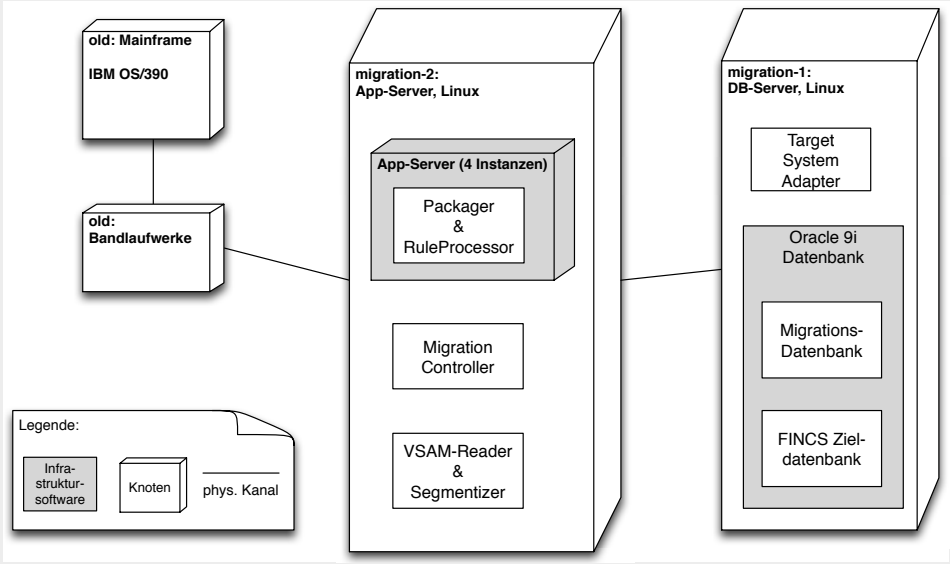


BILD 11.8 Verteilungssicht: Migration auf zwei getrennten Servern

Die gesamte Migration findet auf einem eigenständigen Linux-Server („migration-2“) statt, auf dem in einem EJB-Container mehrere parallele Instanzen der eigentlichen Migrationskomponenten (Packer & RuleProcessor) ablaufen.

Über einen schnellen Bus ist dieser Server mit dem DB-Server („migration-1“) verbunden, auf dem der VSAM-Reader und der Segmentizer die initiale Befüllung der Migrationsdatenbank vornehmen.

Anmerkung: In einer echten Architekturdokumentation sollten Sie hier weitere Leistungsdaten der Hardware (Rechner, Netze, Bus-Systeme etc.) beschreiben.

8 Typische Strukturen und Muster

<<Entfällt im Beispiel>>

9 Technische Konzepte

9.1 Persistenz

Die Datenspeicherung im neuen System übernimmt ein Java Persistenz-Framework (Hibernate). Das Mapping der Objektstruktur auf die relationale Datenbank wird vom Framework komplett übernommen.

Für die eigentlichen Migrationskomponenten wird Persistenz damit nahezu transparent: Ist ein Objektgraph im Speicher aufgebaut, kann er auf einmal gespeichert werden.

Persistenz der Zwischenformate und Fehlerfälle

Die Eingangsdaten in sequenziellem VSAM-Format werden über SQL direkt in die Migrationsdatenbank geschrieben. Ebenso gelangen sämtliche Fehlersätze per SQL in die Fehlerdatenbank.

9.2 Ablaufsteuerung

Die Koordination und Ablaufsteuerung aller Migrationsprozesse wird durch einen zentralen Baustein (MigrationController) durchgeführt.

Die gesamte Verarbeitung erfolgt satzweise. Die Personen-Datei wird satzweise verarbeitet, sämtliche zugehörigen Kontoinformationen werden diesen Sätzen zugeordnet und die resultierenden Datenpakete anschließend gemeinsam migriert.

Aus Kostengründen hat der Auftraggeber eine kommerzielleWorkflow- oder Prozess-Engine abgelehnt.

9.3 Ausnahme- und Fehlerbehandlung

Aufgrund der hohen Korrektheitsanforderungen der Auftraggeber werden sämtliche Ausnahmen beziehungsweise nicht migrierbaren Datensätze in einer Fehlertabelle gespeichert.

Sämtliche Fehlersätze müssen nach der Migration von Sachbearbeitern manuell migriert werden, wofür lediglich ca. 200 Personentage Aufwand zur Verfügung stehen. Jeden Tag kann ein Sachbearbeiter im Mittel 25 Personen migrieren, daher dürfen in der Fehlertabelle höchstens 5000 (= 200 * 25) Datensätze enthalten sein.

9.4 Transaktionsbehandlung

Die einzelnen Migrationsschritte speichern ihren Zustand jeweils satzweise ab. Dadurch ist zu jedem Zeitpunkt ein Wiederaufsetzen durch Neustart der gesamten Anwendung möglich.

9.5 Geschäftsregel und Validierung

Aus Kosten- und Lizenzgründen hat der Auftraggeber die Verwendung einer kommerziellen Regelmaschine (wie ILOG oder VisualRules o. Ä.) a priori ausgeschlossen.

Die Geschäftsregeln (hier: Migrationsregeln) wurden vollständig in Java codiert.

9.6 Kommunikation und Integration

Die Integration mit dem Quellsystem erfolgt über den Austausch von VSAM-Bändern.

Integration mit dem Zielsystem über Java Session-Beans und den Java Application Server. Der Objektgraph der migrierten Daten wird als POJO-Parameter übermittelt (*plain old java object*).

10 Entwurfsentscheidungen

Entscheidung	Datum & Entscheider	Begründung, Konsequenz, Alternativen
Beschreibung der Migrationslogik in Java, NICHT in einer vereinfachten natürlichen Sprache	PL	Nach einigen Versuchen, unter anderem mit eigens formulierten Regelsprachen (mit Lex und Yacc) befand das Entwicklungsteam, dass eine Umsetzung fachlicher Regeln in eine solche Regelsprache keine wesentliche Vereinfachung gegenüber der Programmierung in Java gebracht hätte.
Kein kommerzieller Regelinterpreter	PL	Nach einigen Versuchen mit kommerziellen Regelmaschinen entschied sich das Projektteam gegen den Einsatz einer solchen Komponente. Siehe Entscheidung zu Regelsprache. *Anmerkung:* Aus heutiger Sicht (also posthum ...) würde ich diese Entscheidung überdenken und insbesondere Open-Source-Regelframeworks wie JBoss-Drools auf ihre Tauglichkeit hin überprüfen.
Pipes-und-Filter-Architektur	Architekt, Entwickler	Die Parallelisierbarkeit der eigentlichen Migrationsregeln (im RuleProcessor) wird hierdurch vereinfacht. Es wird keinerlei GUI benötigt.
Dual-Server	Auftraggeber	Aufgrund der hohen Performance-Anforderungen werden Datenbank und Migrationslogik auf zwei getrennten Servermaschinen ausgeführt. Als DB-Server kommt die später vom neuen Kontosystem genutzte Maschine zum Einsatz.
Die Verarbeitung der fachlichen Regeln durch den Rule Processor (siehe Abschnitt 5.1.6 bzw. 5.2.2) wird durch mehrere parallele Prozesse (innerhalb des J2EE Application Servers) durchgeführt.	PL, Kunde	Anforderung seitens des Kunden. Eine rein sequenzielle Bearbeitung aller vorhandenen Personen wäre in der verfügbaren Laufzeit von 24 Stunden nicht möglich gewesen.

11 Qualitätsszenarien

<Entfällt im Beispiel>

12 Risiken

<Entfällt im Beispiel>

13 Glossar und Referenzen

Das Glossar entfällt im Beispiel.

Mehr zu VSAM im frei verfügbaren IBM-Redbook:
http://www.redbooks.ibm.com/abstracts/sg246105.html

Anmerkungen zum System

Einige Worte zur Klärung: Dieses Projekt hat wirklich stattgefunden, und das M&M-System ist wirklich entwickelt worden. Der Auftraggeber, die Fies und Teuer AG, hieß in Wirklichkeit natürlich anders, auch ist der fachliche Inhalt ein wenig vereinfacht worden.

Das echte Migrationsprojekt fand um 2002/2003 irgendwo in Deutschland statt. Über die Laufzeit von gut zwölf Monaten analysierten mehr als zehn Fachexperten die Logik und Geschäftsregeln der bestehenden Mainframe-Systeme und erstellten auf dieser Basis das fachliche Migrationskonzept. Hierbei galt es, die besonderen Anforderungen an Nachweispflichten zu berücksichtigen, ebenso die über die Zeit veränderten gesetzlichen Rahmenbedingungen. Verzahnt damit erstellte ein Team von mehr als zehn Entwicklern und Architekten die passende Software.

Einige der Stolpersteine und Herausforderungen des „echten“ M&M-Projektes:

- Die Datenqualität der Ausgangsdaten schwankte stark: So musste das Projekt die ursprünglich eingeführte Unterscheidung zwischen „Muss“ und „Optional“ für Datenfelder stark einschränken, weil 2–3 % der mehr als 20 Millionen Personendatensätze nicht alle (heute benötigten!) Muss-Attribute enthielten.
- Logische oder fachliche Attribute waren durch extreme Speicherplatzoptimierungen stark verklausuliert und immens schwer identifizierbar. Beispiel: „Wenn es sich um einen Fall des Typs 42 handelt, dann gibt Bit 4 (im EBCDIC-Format) des Datenrecords an, ob es sich um eine Hin- oder Rückbuchung handelt.“
- Die Logik der bestehenden Programme war teilweise älter als 30 Jahre – und für manche der Altprogramme gab es keine Ansprechpartner mehr.

11.2 Beispiel: Kampagnenmanagement im CRM

1 Einführung und Ziele

Zweck des Systems

Dieses Dokument beschreibt die Softwarearchitektur der MaMa[6]-Plattform zur Steuerung von Prozessabläufen zwischen Anbietern, Dienstleistern und Endkunden in Massenmarktsegmenten. MaMa ist technische Basis für Produkte, die für spezifische Anwendungsbereiche angepasst oder konfiguriert werden müssen – sozusagen die Mutter für konkrete Systeme.

Auftraggeber und gleichzeitig Betreiber von MaMa ist ein innovatives Rechenzentrum (IRZ), das im Rahmen von Geschäftsprozess-Outsourcing den Betrieb von Software als Service für seine Mandanten anbietet.[7] IRZ wird konkrete Ausprägungen von MaMa betreiben.

Folgende Beispiele illustrieren die Einsatzmöglichkeiten von MaMa:

- Telekommunikationsunternehmen (= Mandanten) bieten ihren Kunden neue Tarife an. Kunden können diese Angebote über verschiedene Kanäle (Brief, Fax, Telefon) annehmen oder Fragen dazu stellen.
- Handelsunternehmen (= Mandanten) senden spezifische Aktionen (Werbung oder Marketing, etwa: Preisausschreiben oder zielgruppenspezifische Sonderangebote) an ausgewählte Teile ihrer Kunden. Diese können wiederum über unterschiedliche Kanäle auf diese Aktionen reagieren.
- Versicherungsunternehmen (= Mandanten) senden an ihre Versicherten Vorschläge zur Änderung oder Anpassung bestimmter Verträge. Die Versicherten können über verschiedene Kanäle (Brief, E-Mail, Fax, Telefon oder persönliche Gespräche in Geschäftsstellen) auf diese Vorschläge reagieren.

Etwas verallgemeinert, sehen Sie diese Aufgaben in Bild 11.9.

Definitionen der Begriffe Mandant, Partner und Marktteilnehmer finden Sie in Kapitel 13 (Glossar).

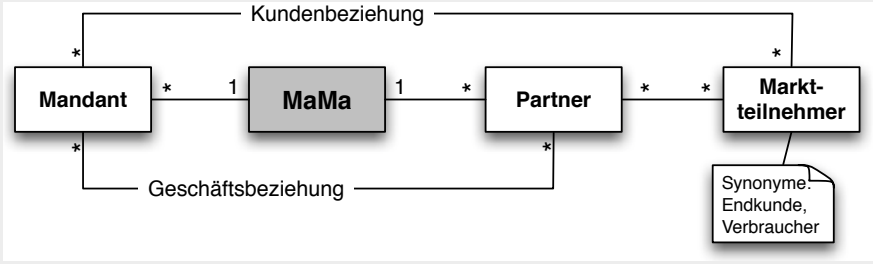


BILD 11.9 Zweck des Systems: Koordination von Prozessabläufen zwischen Mandanten, Partnern und Marktteilnehmern im Massenmarkt

[6] MaMa steht für **Ma**ssen**Ma**rkt.

[7] Die Rahmendaten und den Namen dieses Projektes habe ich erfunden, das hier beschriebene System existiert jedoch wirklich.

Leserkreis

- Alle in Abschnitt 1.3 dieser Dokumentation genannten Stakeholder von MaMa.
- Software-Entwickler, Architekten und technische Projektleiter, die Beispiele für eine Architekturdokumentation (auf Basis des arc42-Templates) suchen.
- Mitarbeiter von IT-Projekten, die sich das Leben in der Softwarearchitektur vereinfachen, indem sie auf ein bewährtes Template zur Dokumentation zurückgreifen.

1.1 Fachliche Aufgabenstellung

Die wichtigsten funktionalen Anforderungen sind:

- MaMa unterstützt die Ablaufsteuerung bei CRM-Kampagnen, die das Rechenzentrum IRZ für seine Auftraggeber (genannt „Mandanten“ oder „Anbieter“) im Rahmen des *Business Process Outsourcing* abwickelt. Folgende Ausprägungen muss MaMa in der ersten Version unterstützen:
  - Tarif- und Vertragsänderungen für Versicherungsgesellschaften, Telekommunikations- und Internetanbieter sowie Energieversorger. Ein Beispiel dazu finden Sie in Abschnitt 1.1.1 weiter unten.
  - Rückfragen zu Stammdaten für Versicherungsgesellschaften oder die Gebühreneinzugszentralen von Rundfunk- und Fernsehgesellschaften.
  - Abfrage zusätzlicher Stamm- oder Personendaten für Versicherungsunternehmen, die im Rahmen von Gesetzesänderungen notwendig werden.
- MaMa ist Basis einer Produktfamilie und muss daher möglichst ohne Programmänderungen für unterschiedliche Ein- und Ausgangsschnittstellen konfigurierbar sein. Siehe dazu Abschnitt 1.1.2.

Anmerkung: Aufgrund der hohen fachlichen Komplexität dieses Systems möchte ich die fachlichen Anforderungen anhand eines Beispiels näher erläutern. In einer *echten* Projektsituation sollte die Anforderungsdokumentation das leisten –die Ihnen in diesem Beispiel leider nicht zur Verfügung steht.

1.1.1 Einsatz von MaMa für Vertrags- und Tarifänderungen bei Telekommunikationsunternehmen

Der Mobilfunkanbieter MoF[8] schlägt seinen Kunden (synonym: Marktteilnehmern) neue Tarife vor: Manche der bestehenden Tarife oder Tarifkombinationen sind überaltert und technisch nicht mehr aktuell. Andere Tarife sind nicht mehr marktgerecht und können gegen leistungsfähigere oder kostengünstigere ausgetauscht werden. Da solche Änderungen jedoch in die Vertragsverhältnisse zwischen MoF und seinen jeweilig betroffenen Kunden eingreifen, ist zwingend die schriftliche Zustimmung der Kunden erforderlich.

[8] Wie Sie richtig vermuten, habe ich den Namen dieses Unternehmens frei erfunden.

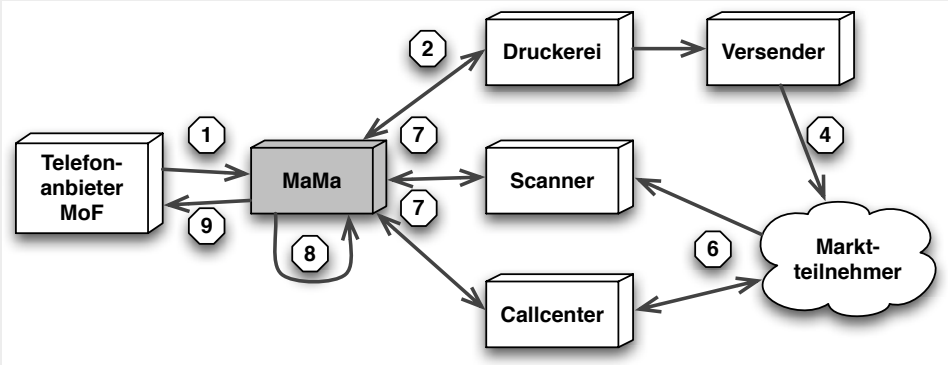


BILD 11.10 Ablauf einer Kampagne für Vertrags- und Tarifänderungen

MoF möchte im Rahmen einer Kampagne alle betroffenen Kunden schriftlich über die Angebote dieser neuen Tarifmöglichkeiten informieren. Die Kunden können schriftlich, per Fax oder in Filialen eine für sie passende Tarifkombination auswählen und damit formell eine Vertragsänderung beauftragen. Konkret läuft diese Kampagne wie folgt ab:

1. MoF selektiert in seinen internen Systemen die betroffenen Kunden und exportiert deren Stamm-, Tarif- und Vertragsdaten an MaMa.
2. MaMa übergibt die relevanten Teile dieser Daten an eine Druckerei, die personalisierte Anschreiben für sämtliche betroffenen Kunden erstellt und nach Regionen vorsortiert an einen Briefdienstleister liefert.
3. MaMa benachrichtigt die weiteren Partner über diese Kampagne und übergibt die jeweils benötigten Daten. Hierzu gehören:
   - Ein Scan-Dienstleister, der Rücksendungen entgegennehmen und digitalisieren wird.
   - Ein Telefondienstleister[9], der alle telefonischen Rückmeldungen oder Rückfragen bearbeitet.
   - Sämtliche Filialen beziehungsweise Vertriebspartner von MoF, damit auch persönliche Rückmeldungen der Kunden dort korrekt bearbeitet bzw. weitergeleitet werden.
4. Der Briefdienstleister transportiert die gedruckten Sendungen an die Kunden und übermittelt elektronisch die aufgetretenen Zustellprobleme (verzogen mit Nachsendeantrag, unbekannt verzogen, unbekannt, Annahme verweigert).
5. Die Kunden können auf verschiedene Arten auf dieses Angebot reagieren:
   - Kunde füllt das beigefügte Antwortschreiben vollständig und korrekt aus.
   - Kunde füllt das beigefügte Antwortschreiben unvollständig oder fehlerhaft aus.
   - Kunde hat Rückfragen.
   - Kunde reagiert nicht.

[9] („Denglisch“:) Callcenter

6. In allen Fällen kann der Kunde folgende Arten der Rückmeldung wählen, die letztlich alle durch MaMa koordiniert und bearbeitet werden sollen:
   - Briefsendung an eine (kampagnenspezifische) Adresse.
   - Rückmeldung per Fax.
   - Rückmeldung per E-Mail oder Web-Formular.
   - Persönliche Rücksprache in einer Filiale von MoF.
   - Telefonische Rückmeldung an ein Call-Center.
7. Die jeweiligen Partner übermitteln periodisch oder kontinuierlich alle ihnen vorliegenden Rückmeldungen an MaMa.
8. MaMa ermittelt aus den vorliegenden Antwortdaten der verschiedenen Partner täglich (bei Bedarf auch häufiger) die passenden Folgeaktionen:
   - Es liegen sämtliche Daten inklusive der rechtsgültigen Unterschrift vor, die MoF für eine Tarif- oder Vertragsänderung benötigt. In diesem Fall werden diese Daten an MoF übermittelt.
   - Teile der benötigten Daten fehlen noch und müssen durch passende Rückfragen beim Kunden ermittelt werden.
   - Eine fehlende Unterschrift muss schriftlich über ein erneutes Anschreiben eingeholt werden.
   - Fehlende Tarifinformationen können durch ein Call-Center telefonisch erfragt werden. Wenn der Kunde dreimal nicht erreicht werden konnte, wird er erneut angeschrieben.
9. MoF ändert aufgrund der von MaMa gelieferten Ergebnisdaten die Verträge und Tarife der jeweiligen Kunden.

Wichtig für den Mobilfunkanbieter MoF ist, dass er nur mit MaMa Daten austauschen muss, um diese Kampagne durchzuführen. Sämtliche unterstützenden Prozesse erledigt MaMa durch Kooperation mit passenden Partnern!

1.1.2 Konfiguration einer Kampagne

MaMa ist die Basis einer Produktfamilie, d. h. nur in einer konkreten Ausprägung („Konfiguration“) ablauffähig. Dies bedeutet, dass vor der Durchführung einer Kampagne verschiedene Festlegungen getroffen werden müssen. Die folgende Tabelle gibt einen Überblick über diese Konfiguration, orientiert am Beispiel der MoF-Kampagne aus dem vorigen Abschnitt.

Schritt innerhalb der Kampagne	Beispiel (aus Mobilfunk-Domäne)
Mögliche Aktionen der Kampagne definieren (Input- und Output-Aktionen sowie MaMa-interne Aktionen*).	Output-Aktionen: BriefDrucken InfoAnCallCenter InfoAnScandienstleister ErgebnisseAnMandant Input-Aktionen: StammdatenInput ScandatenInput CallCenterInput FilialenInput Interne Aktionen: <keine>
Marktteilnehmerdaten sowie zugehörige Kampagnendaten (i. d. R. Vertrags-, Tarif- oder Angebotsdaten) als Erweiterung des MaMa-Basismodells in UML modellieren.	Attribute der Marktteilnehmer: Bestehender Tarif Liste möglicher neuer Tarife Vertragsnummer Ablaufdatum aktueller Vertrag Laufzeit des neuen Vertrags
Abläufe inklusive Plausibilitätsprüfungen als UML-Aktivitätsdiagramm definieren.	<entfällt im Beispiel>
Für alle Input- und Output-Aktionen die jeweiligen Schnittstellen zu Partnern konfigurieren: Datenformate und Protokolle festlegen	<entfällt im Beispiel>
Metadaten der Kampagne definieren und festlegen	z. B. Start- und Endtermine, Stichtage für Aktivitäten, Adresse für Rücksendungen, Telefonnummern für Rückrufe bei Kunden, Zugangsdaten für Auftraggeber und Partner für ftp-Server (zum Up-/Download von Daten)
Kampagne durchführen	Wird für das Beispiel in Abschnitt 1.1.1 genauer beschrieben.

* Die Bedeutung von „Aktionen“ wird im Abschnitt 8.1 über Ablaufsteuerung näher erklärt.

1.2 Qualitätsziele

Die primären Qualitätsziele von MaMa lauten:

1. Flexibilität: Kampagnen sollen ohne Programmierung und ausschließlich durch Konfiguration aufgesetzt werden können. Das betrifft:
   - die spezifischen Datenstrukturen einer Kampagne müssen innerhalb von 2 h vollständig konfiguriert werden können;
   - die ein- und ausgehenden Schnittstellen zu den Partnern (Übertragungsprotokolle, Aufbau der Datensätze, Text- oder Binärschnittstellen, CSV-, Fix- oder XML-Formate, einmalige oder periodische Übertragung, Push- oder Pull-Aufrufe der Schnittstelle sowie die Art der Authentifizierung, Autorisierung, Datenverschlüsselung und -kompression) sollen innerhalb von 8 Arbeitsstunden vollständig konfiguriert werden können.
   - die notwendigen Abläufe der Kampagne inklusive ihrer Plausibilitätsprüfungen.
   - Die Im- und Exportkonfiguration sowie die individuellen Prozessabläufe sollen innerhalb eines Arbeitstages vollständig konfiguriert werden können.
2. Datenschutz und -sicherheit müssen beispielsweise für Mandanten aus dem Gesundheits- oder Versicherungswesen auditierbar sein. Bei Bedarf müssen daher sämtliche Datenänderungen einer Kampagne revisionssicher protokolliert werden können.
3. Performance: Kampagnen können praktisch beliebig umfangreich werden. Für einige Mandanten sollen beispielsweise mehrere 10 Millionen Endkunden innerhalb einer Kampagne bearbeitet werden. Weitere Mengengerüste:
   - Gesamtgröße aller Attribute einzelner Kundendatensätze innerhalb einer Kampagne zwischen 1 kByte und 15 Mbyte (für Kampagnen mit umfangreichen Bild- oder Scandaten).
   - Anzahl möglicher Partnerunternehmen: unter 100
   - Anzahl von Mandanten/Anbietern: unter 100
   - Anzahl gleichzeitiger Kampagnen: unter 100
4. Mandantentrennung: Die Daten der jeweiligen Mandanten einer Kampagne müssen vollständig voneinander getrennt sein. Es darf unter keinen Umständen passieren, dass beispielsweise die Kundendaten mehrerer Versicherungen miteinander verwechselt werden.

Nicht-Ziele

Was MaMa nicht leisten soll:

- MaMa soll exklusiv vom Auftraggeber IRZ betrieben werden. Es soll kein *Produkt* werden.
- CRM: MaMa soll keine gebräuchlichen Customer-Relationship-Management-Systeme ersetzen.

1.3 Stakeholder

Name/Rolle	Ziel/Berührungspunkt	Notwendige Beteiligung
Management des Auftraggebers IRZ	Fordert flexible und leistungsfähige Basis für ihre zukünftigen Geschäftsfelder.	Periodische Abstimmung über aktuelle und geänderte Anforderungen, Lösungsansätze
Softwareentwickler, die MaMa mitentwickeln	Müssen den inneren Aufbau von MaMa gestalten und implementieren.	Aktive Beteiligung an technischen Entscheidungen
Partner	Schnittstellen zwischen MaMa und Partnern	Partner müssen Daten liefern bzw. empfangen und haben großes Interesse an flexiblen Schnittstellen.
Eclipse-RCP Projekt (Hersteller des eingesetzten UI-Frameworks)	MaMa verwendet Eclipse-RCP zur Umsetzung der grafischen Programmteile.	Eclipse ändert die Programmierschnittstellen häufig und kaum vorhersehbar – was Auswirkungen auf die Anpassung der GUI besitzt.
Normungsgremien	MaMa wird in gesetzlich normierten Bereichen (z. B. Krankenversicherung) eingesetzt.	Normungsgremien (im Beispiel nicht namentlich genannt) geben rechtliche Randbedingungen vor.

Zu MaMa gibt es in der Realität eine Reihe weiterer technischer Dokumente, insbesondere die Schnittstellendokumentation, mit deren Hilfe Partnerunternehmen (wie Druck-, Scan- oder Callcenter-Dienstleister) eine Anbindung an MaMa realisieren können (obwohl in der Praxis *deren* Schnittstellen meist feststehen und die Betreiber von MaMa die Anbindung konfigurieren!).

2 Einflussfaktoren und Randbedingungen

2.1 Technische Einflussfaktoren

Randbedingung	Erläuterung
Software-Infrastruktur	Linux als Betriebssystem der Ablaufumgebung MySQL beziehungsweise bei Bedarf Oracle als Datenbank Java 6 oder 7 als Software-Plattform im Betrieb verfügbar
Systembetrieb	Batch, soweit möglich
Grafische Oberfläche	Eclipse RCP zur Konfiguration von Kampagnen und Schnittstellen
Programmiersprachen und -frameworks	Java, Hibernate, MDSD-Generator zur Erzeugung von Teilen des Quellcodes
Modellierungs-werkzeuge	UML-2-kompatibles Modellierungswerkzeug zur Modellierung von kampagnenspezifischen Datenstrukturen und -abläufen
Technische Kommunikation	Muss zu Mandanten beziehungsweise Partnern über ftp, sftp, http, https möglich sein, mit und ohne VPN. Anbindung an Messaging-Strukturen (wie Tibco oder MQ-Series) ist nicht vorgesehen.

2.2 Organisatorische Einflussfaktoren

IRZ ist ein agiler Auftraggeber, der innerhalb der Softwareentwicklung iterative Prozesse bevorzugt.

Allerdings legt IRZ großen Wert auf langlebige Systeme und fordert daher aussagekräftige, wartungsfreundliche technische Dokumentation (von der Sie gerade einen Auszug in den Händen halten …).

3 Kontextabgrenzung

3.1 Allgemeiner fachlicher (logischer) Kontext

In der logischen Kontextabgrenzung von Bild 11.11 erkennen Sie, dass MaMa nur mit dem Mandanten beziehungsweise mehreren Partnern kommuniziert. Für jede Kampagne werden spezifische Partner ausgewählt – ein Beispiel finden Sie in Abschnitt 3.2.

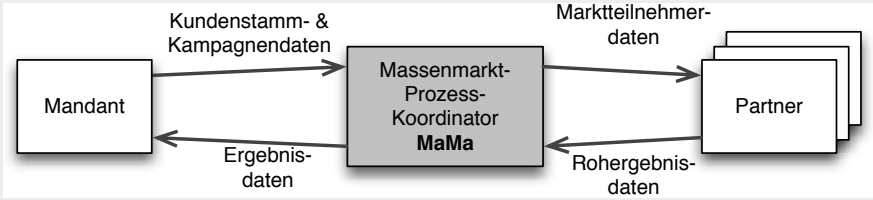


BILD 11.11 Allgemeiner fachlicher (logischer) Kontext

Kurzbeschreibung der externen Schnittstellen

Schnittstelle/Nachbarsystem	Ausgetauschte Daten (Datenformate, Medien)
Kundenstamm- und Kampagnendaten (von Mandanten)	Mandanten übermitteln Stammdaten ihrer Kunden sowie Daten/Metadaten der Kampagnen an MaMa.
Ergebnisdaten (an Mandanten)	MaMa übergibt Ergebnisse, Zwischenergebnisse oder Rückfragen (aufgrund von Fehlern oder Ausnahmen) an Mandant.
Marktteilnehmerdaten (an Partner)	MaMa übergibt Marktteilnehmerdaten an Partner.
Rohergebnisdaten (von Partnern)	Partner übergeben ihre jeweiligen Zwischenergebnisse an MaMa.

Partner sind für MaMa externe Dienstleistungserbringer, beispielsweise:

- Druckereien, zum Druck von Briefsendungen oder ähnlichen Schriftstücken;
- Briefversender oder Distributoren, um Drucksendungen zu Endkunden zu transportieren;
- Scandienstleister, zur Umwandlung von Rücksendungen der Endkunden in Datenformate, inklusive automatische Texterkennung (OCR);
- Call-Center und Telko-Dienstleister, zur telefonischen Ansprache von Endkunden, Faxversand oder Bearbeitung von Anrufen der Endkunden;
- Webhoster und E-Mail-Provider für jegliche Online-Dienste, die im Rahmen von Kampagnen erbracht werden müssen.

3.2 Spezielle Kontextabgrenzung der Mobilfunk-Kampagne

In Bild 11.10 haben Sie bereits die Kontextabgrenzung einer konkreten Instanz von MaMa für Mobilfunk-Unternehmen gesehen. Die konkrete Ausprägung für die Telco-Domäne sehen Sie in Bild 11.12.

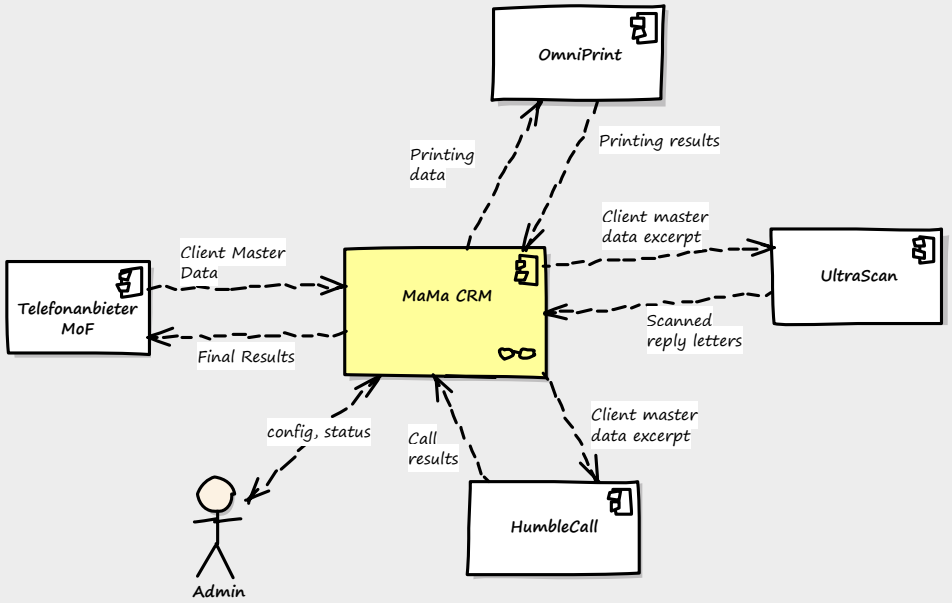


BILD 11.12 Fachlicher Kontext der Telco-Domäne

3.3 Verteilungskontext: MaMa als Basis einer Produktfamilie

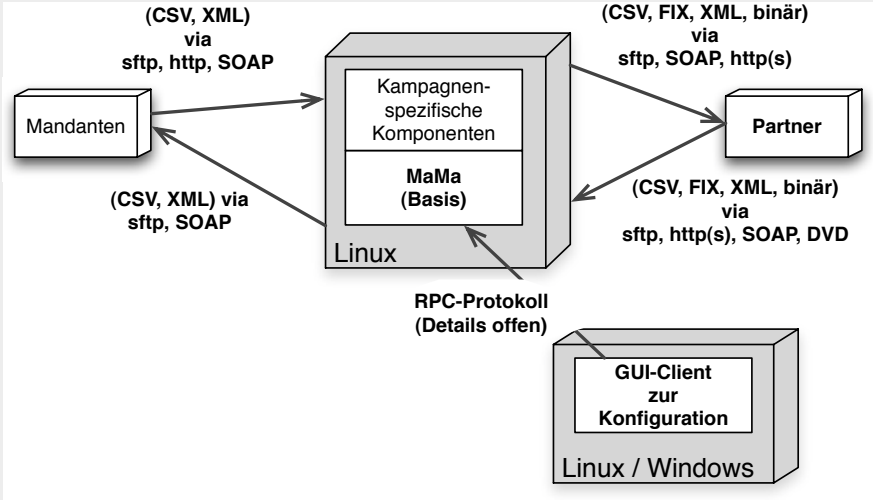


BILD 11.13 Allgemeiner Verteilungskontext von MaMa mit (schematischen) externen Schnittstellen

MaMa stellt die Basis einer Produktfamilie dar. Jede Kampagne wird auf einer (in der Regel virtuell) dedizierten Linux-Installation betrieben. In jeder Kampagne gibt es vier Kategorien externer Schnittstellen von und zu MaMa:

Kurzbeschreibung der externen Schnittstellen

Schnittstelle/Nachbarsystem	Technologie/Protokoll
Kundenstamm- und Kampagnendaten (von Mandanten)	CSV- oder XML-Formate als Dateien per sftp- oder https-Upload. Alternativ: Einlieferung per Webservice oder DVD.
Ergebnisdaten (an Mandanten)	CSV- oder XML-Formate als Dateien per sftp. Alternativ: Ablieferung per Webservice.
Marktteilnehmerdaten (an Partner)	Format je nach Partner und Kampagne. MaMa muss CSV, Fix-Length, XML sowie Webservices unterstützen.
Rohergebnisdaten (von Partnern)	Format je nach Partner, s. o.

Fachliche Abläufe finden Sie in der Laufzeitsicht in Abschnitt 6.1 dieser Dokumentation.

4 Lösungsstrategie

Die Struktur von MaMa basiert in hohem Maße auf dem technischen Konzept zur konfigurierbaren Ablaufsteuerung, das Sie in Kapitel 8.1 dieser Dokumentation finden. Ebenfalls wichtig für das Verständnis ist das Konzept der Produktfamilie, erläutert in Kapitel 8.2 (zusammen mit der Persistenz und Generierung). Schließlich erlaubt MaMa die Ein- und Ausgabedaten pro Kampagne und Mandant zu konfigurieren.

Diese Konzepte stellen die Qualitätsziele „Flexibilität“ und „Performance“ sicher.

5 Bausteinsicht

Voraussetzung für das Verständnis der statischen Struktur von MaMa ist das Konzept zur Ablaufsteuerung (siehe Kapitel 8.1 dieser Dokumentation).

5.1 MaMa-Bausteinsicht Level 1

Sämtliche Abhängigkeiten innerhalb von Bild 11.14 sind *benutzt*-Beziehungen innerhalb des MaMa-Quellcodes.

Strukturiert wurde MaMa auf dieser Ebene primär anhand funktionaler Aspekte: Jede der Level-1-Blackboxen verantwortet unmittelbar einen Teil der MaMa-Gesamtfunktionalität. Einzig die Bausteine `CampaignDataManagement` und `OperationsMonitoring` haben rein unterstützende Aufgaben (für die Datenspeicherung respektive die Überwachung zur Laufzeit).

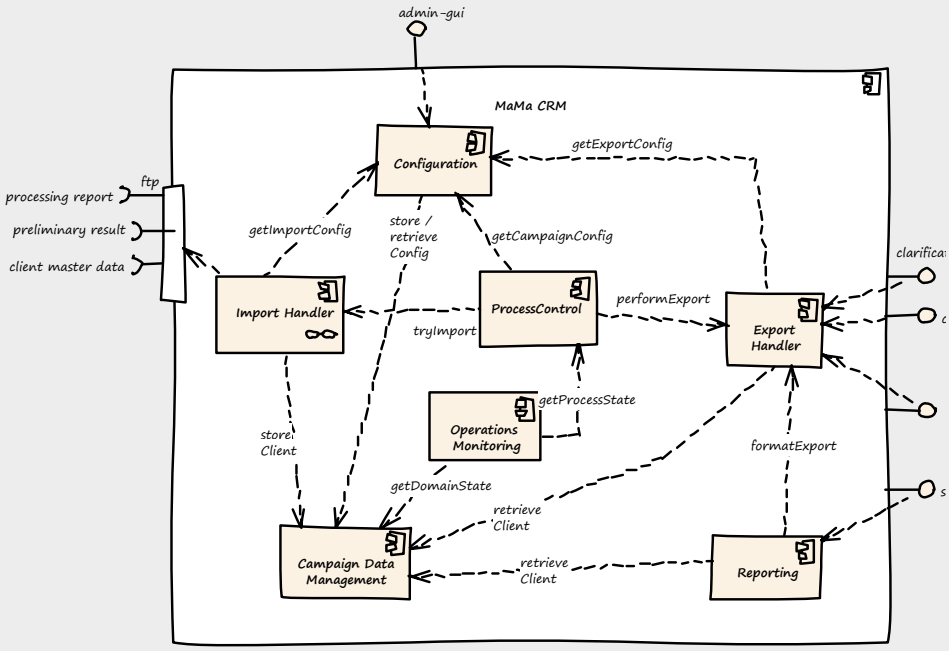


BILD 11.14 Whitebox-Darstellung von MaMa, Level 1

Das Architekturziel der hohen Flexibilität gegenüber Datenstrukturen und Ein-/Ausgangsschnittstellen wurde an die Bausteine `Configuration`, `InputHandler` und `ExportHandler` delegiert.

Das Architekturziel der hohen Performance verantworten primär die Bausteine Input und Output, unterstützt durch CampaignDataManagement.

Die folgenden Abschnitte beschreiben die in der Abbildung dargestellten Bausteine, strukturiert nach dem Blackbox-Template.

Level-1-Bausteine von „MaMa“

Baustein	Beschreibung
Import bzw. Export Handler	Liest/schreibt Daten über die Außenschnittstellen, d. h. an Partner und Mandanten. Hierzu gehören insbesondere Teile von IClient-Instanzen und Clarification Requests.
Campaign Data Management	Speichert sämtliche Kampagnendaten, insbesondere die Domänenentitäten (Instanzen von IClient), zusätzlich Konfigurationsdaten.
ProcessControl	Steuert die Abläufe/Prozesse der Kampagne, verantwortet Ausführung kampagnenspezifischer Geschäftsregeln.
Configuration	Hält Konfiguration aller Import- und Exportformate, Im- und Export Filter.
Operations Monitoring	Beobachtet und berichtet den Zustand sämtlicher Import- und Exportprozesse sowie relevanter Teile der Datenbank.
Reporting	Fasst den Zustand der Kampagnen für die Mandanten bzw. die Betreiber des Systems zusammen, insbesondere die aufgelaufenen Kosten für die externen Partner (z. B. Druckerei).

5.1.1 ImportHandler

- *Zweck/Verantwortlichkeit:* Nimmt über Außenschnittstellen alle Arten von Daten von Mandanten und Partnern entgegen.
- *Schnittstelle(n):* Eingang: Kundenstamm-, Kampagnen-, Kampagnen-Meta- und Rohergebnisdaten (CSV, XML, binär). Nutzt Methoden/Funktionen von CampaignDataManagement.
- *Variabilität:* Eingangskanäle und -formate konfigurierbar (CSV, FIX, AFP, PDF, XML über (S) FTP, SOAP, https-upload, ssh-remote-copy, DVD-Eingang, MQ-Series, TIBCO)
- *Ablageort/Datei:* Package `de.arc42.example.mama.importHandler`, insbesondere die Klasse `InputManager`
- *Offene Punkte:* Zurzeit unterstützt dieser Baustein keine PGP-verschlüsselten Eingangsdaten, kann bei Webservice-Einlieferung keine digitalen Signaturen prüfen und besitzt keine Anbindung an MQSeries und TIBCO.

▸▸ **Die Verfeinerung als Whitebox finden Sie in Abschnitt 5.2.1.**

5.1.2 Campaign Process Control

- *Zweck/Verantwortlichkeit:* Steuert die Abläufe innerhalb von Kampagnen, verantwortet die Ausführung und Prüfung der kampagnenspezifischen Geschäftsregeln.
- *Schnittstelle(n):* Lesezugriff auf CampaignDataManagement, Notify-Mechanismus an Output.
- *Variabilität:* Die konkreten Regeln für Plausibilisierung sowie Ablaufsteuerung einer Kampagne werden bei der Einrichtung der Kampagne konfiguriert.
- *Ablageort/Datei:* Package: `de.arc42.example.mama.campaigncontrol`

▸▸ **Die Verfeinerung als Whitebox finden Sie in Abschnitt 5.2.2**.

5.1.3 Campaign Data Management

Dieser Baustein wird vollständig aus dem spezialisierten Domänenmodell generiert.
Details siehe Abschnitt 8.2.

- *Zweck/Verantwortlichkeit:* Verwaltet sämtliche Kampagnendaten und Metadaten.
  Stellt Verwaltungsoperationen zum Lesen, Schreiben und Löschen all dieser Daten bereit.
- *Schnittstelle(n):* Erhält Daten von Input, Configuration sowie Campaign ProcessControl.
  Liefert Daten für praktisch alle anderen Bausteine.
- *Variabilität:* Variabel gegenüber spezifischen Datenstrukturen von Kampagnen. Die konkreten Datenzugriffsklassen, DB-Tabellen etc. werden spezifisch pro Kampagne aus einem Modell generiert.
- *Ablageort/Datei:* Package: `de.arc42.example.mama.campaigndata`
- *Offene Punkte:* Bisher nur wenige Möglichkeiten zum Performance-Tuning der Datenbank.

5.1.4 Configuration

- *Zweck/Verantwortlichkeit:* Grafische Oberfläche zur Konfiguration von Mandanten, Kampagnen, Schnittstellen und Geschäftsregeln.
- *Schnittstelle(n):* Speichert Konfigurationsdaten in `CampaignDataManagement`.
- *Ablageort/Datei:* Package: `de.arc42.example.mama.configuration`
- *Offene Punkte:* Teile der Konfiguration einer Kampagne erfolgen zur Zeit über ein UML-Modell. Die vom Auftraggeber gewünschte „*zentrale Stelle zur Konfiguration sämtlicher Bestandteile einer Kampagne*“ ist somit noch nicht gegeben.

5.1.5 Export Handler

- *Zweck/Verantwortlichkeit:* Liefert über die Außenschnittstellen Ergebnisdaten (an Mandanten) sowie Marktteilnehmerdaten (an Partner).
- *Schnittstelle(n):* Ergebnisdaten (CSV, XML) sowie Marktteilnehmer-, Kampagnendaten (CSV, FIX, XML, PDF). Nutzt Methoden/Funktionen von CampaignDataManagement.
- *Variabilität:* Ausgangskanäle und -formate konfigurierbar (CSV, FIX, XML über (S)FTP, SOAP, MQ-Series und TIBCO)
- *Ablageort/Datei:* Package `de.arc42.example.mama.output`, insbesondere die Klasse `OutputManager`
- *Offene Punkte:* Zurzeit kann dieser Baustein keine Ausgangsdaten mit PGP verschlüsseln, keine digitalen Signaturen erzeugen und besitzt keine Anbindung an MQSeries und TIBCO.

5.1.6 Reporting sowie Operations Monitoring

<Werden im Beispiel nicht weiter betrachtet.>

5.2 MaMa-Bausteinsicht Level 2

Dieser Abschnitt detailliert die Blackbox-Bausteine der Ebene 1. Aufgrund des Beispielcharakters dieser Dokumentation zeige ich hier nur Auszüge der gesamten Dokumentation.

5.2.1 Whiteboxsicht Baustein „Import Handler“, Level 2

Die Struktur des `ImportHandler` wurde aufgrund funktionaler Aspekte gewählt: Die enthaltenen Bausteine leisten jeweils einen Schritt in der grundsätzlichen Verarbeitungskette von Eingabedaten (Empfangen, Archivieren, Filtern, Validieren und In-Java-Objekte-Übersetzen). Die Ablaufsteuerung zwischen diesen Bausteinen obliegt dem Receiver – eine deutliche Schwäche im Entwurf dieser Whitebox. Details dazu in Abschnitt 5.3.1.

Dieser Baustein verantwortet das Architekturziel der „flexiblen Eingangsformate und Übertragungsprotokolle“ für jede Kampagne.

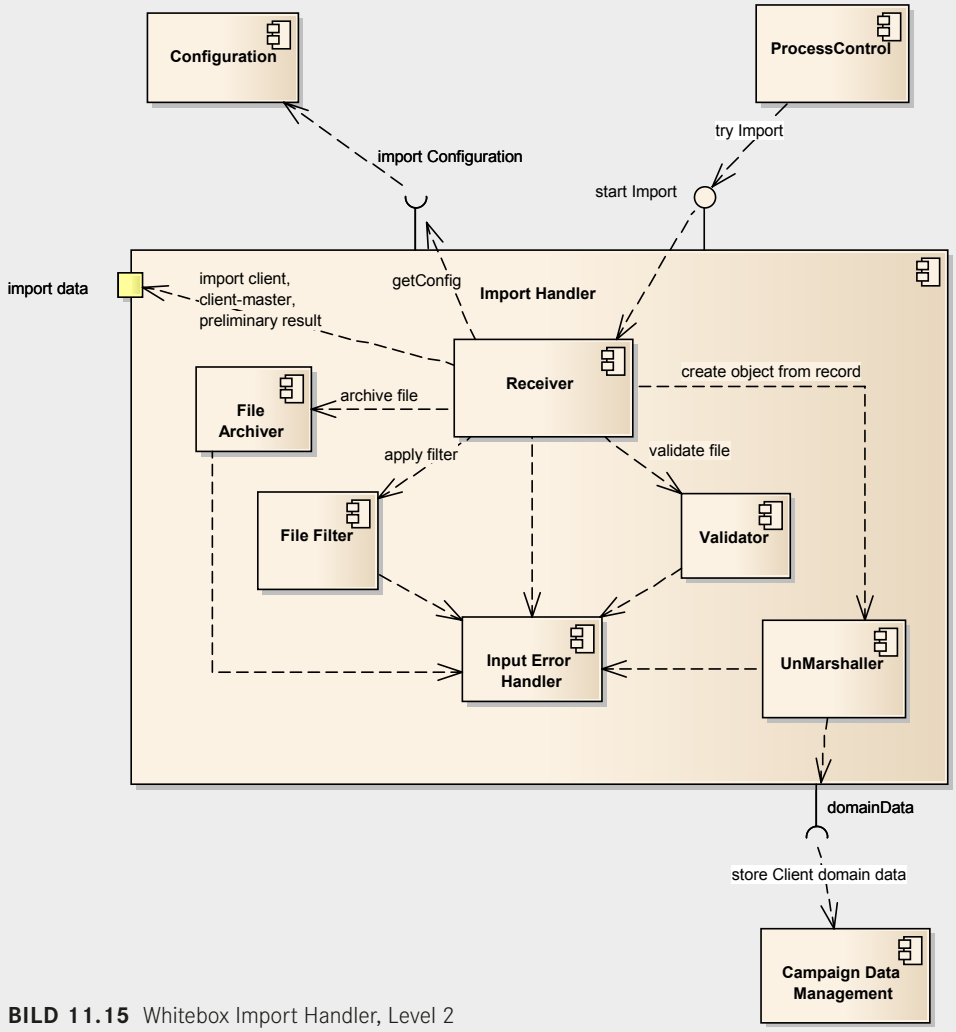


BILD 11.15 Whitebox Import Handler, Level 2

Baustein	Beschreibung
Receiver	Führt den eigentlichen Empfang der Daten durch und koordiniert die übrigen Aufgaben beim Import.
Validator	Prüft eine Datei auf Einhaltung ihrer Formatkonvention (beispielsweise: korrekte Anzahl Felder in CSV-Datei),
File Filter	Führt dateiweite Konvertierungen durch, bislang Entschlüsselung und Dekomprimierung.
File Archiver	Speichert empfangene Dateien und Daten in einem nicht löschbaren Archiv, sofern für die Kampagne notwendig.
Input Error Handler	Berichtet sämtliche beim Import und den nachgelagerten Konvertierungs- und Prüfaufgaben aufgetretene Fehler.
UnMarshaller	Der Magier: wandelt Zeichenketten in gültige IClient-Instanzen um, d. h. in Domain-Entities.

5.2.1.1 Blackbox „Receiver“

- *Zweck/Verantwortlichkeit:* Führt den physischen Daten- oder Dateiempfang durch (wird implementiert durch beispielsweise FileReceiver, FTPReceiver, SOAPReceiver und andere). Koordiniert die übrigen Aufgaben der Input-Verarbeitung.
- *Schnittstelle(n):* Nimmt über die Außenschnittstellen Kundenstamm- & Kampagnendaten von Mandanten entgegen sowie Rohergebnisdaten von den Partnern. Interne Schnittstellen zu allen anderen Bausteinen von Input.
- *Variabilität:* Konfigurierbar für unterschiedliche Eingabekanäle und -protokolle (FTP, http-Upload, DVD-Einlieferung).
- *Ablageort/Datei:* Package `de.arc42.example.mama.receiver`, insbesondere das Interface `IReceiver`
- *Offene Punkte:* Zurzeit keine Schnittstellen zu MQ-Series oder TIBCO.

`Receiver` übernimmt zu viele Aufgaben – neben dem Datenempfang auch noch die Koordination anderer Bausteine. Diese schlechte funktionale Kohäsion führt zu schwer verständlichem Quellcode und zu hohen Wartungsrisiken. Die Koordinationsaufgaben sollten daher in einen eigenen Baustein verlagert werden; siehe Abschnitt 11.1 (Risiken).

▸▸ Die Verfeinerung als Whitebox finden Sie in Abschnitt 5.3.1.

5.2.1.2 Blackbox „Validator“

- *Zweck/Verantwortlichkeit:* Überprüft, ob die einzelnen Bestandteile (Datei, Datensätze, Satzteile, Records o. Ä.) der Datei den vereinbarten Kriterien für diese Kampagne/Partner/Mandanten genügen.
- *Schnittstelle(n):* Erhält von `Receiver` die zu validierenden Daten in Blöcken (Dateien oder einzelne Dateinsätze), liest die Validierungsregeln aus CampaignDataManagement.
- *Variabilität:* Syntaxregeln können pro Kampagne individuell festgelegt werden.
- *Ablageort/Datei:* Package `de.arc42.example.mama.validator`
- *Offene Punkte:* Zurzeit werden Validierungsfehler nur in Logfiles abgelegt und nicht korrekt an `InputErrorHandler` gemeldet.

Dieser Baustein basiert auf dem [Validator] Framework in Kombination mit einer MaMa-spezifischen Beschreibungssprache. Validiert sowohl Dateien wie einzelne Datensätze – was aus Vereinfachungsgründen künftig auf zwei getrennte Bausteine verteilt werden könnte. Aus heutiger Sicht nicht riskant.

Anmerkung: Diese Information geht über eine reine Blackbox-Beschreibung etwas hinaus. Den Aufwand einer zusätzlichen Whitebox-Beschreibung halte ich jedoch für unangemessen …

5.2.1.3 Blackbox „FileFilter“

Anmerkung: Hier mal eine alternative Darstellung des Blackbox-Templates, als Tabelle:

Zweck/Verantworung	Führt sämtliche notwendigen dateiweiten Konvertierungen durch, beispielsweise Dekompression, Entschlüsselung, Auswertung digitaler Signaturen oder Ähnliche.
Schnittstellen	Erhält von `Receiver` die zu bearbeitenden Dateien und eine Liste der auszuführenden Operationen.
Ablageort	Package `de.arc42.example.mama.filefilter`
Offene Punkte	Zurzeit kann `FileFilter` keine PGP-verschlüsselten Dateien lesen, keine digitalen Signaturen auswerten und keine verschlüsselten ZIP-Dateien entschlüsseln.

5.2.1.4 Blackbox „FileArchiver“

- *Zweck/Verantwortlichkeit:* speichert die vom `Receiver` empfangenen Dateien in einem von MaMa nicht löschbaren Archiv, sofern für die Kampagne notwendig.
- *Schnittstelle(n):* enthält vom `Receiver` die zu archivierenden Dateien.
- *Ablageort/Datei:* Package `de.arc42.example.mama.archiver`, insbesondere die Klasse `FileArchiver`

Der FileArchiver ist eine Fassade zum frei verfügbaren [HoneyComb], einer von Sun Microsystems entwickelten Speicherlösung für unveränderliche Daten.

5.2.1.5 Blackbox „InputErrorHandler“

- *Zweck/Verantwortlichkeit:* Sammelt sämtliche während des Input-Vorgangs auftretenden Fehler und -meldungen zur späteren Bearbeitung.
- *Schnittstelle(n):* Erhält Fehlermeldungen von allen Input-Bausteinen, mit Ausnahme von `Validator`.
- *Ablageort/Datei:* Package `de.arc42.example.mama.inputerror`
- *Offene Punkte:* Zur Zeit keine Integration mit `Validator`.

Die fehlende Integration von `InputErrorhandler` mit dem `Validator` ist eine signifikante Schwäche in Entwurf und Implementierung von MaMa. Siehe Abschnitt 11.1 (Risiken).

5.2.1.6 Blackbox „UnMarshaller“

- *Zweck/Verantwortlichkeit:* Erzeugt aus den Dateibestandteilen (Records o. Ä.) wieder Objekte des Domain-Models und speichert diese ggfs. im `CampaignDataManagement`.
- *Schnittstelle(n):* Erhält vom `Receiver` Listen von Strings, aus denen `UnMarshaller` echte MaMa-Objekte erstellt.
- *Ablageort/Datei:* `de.arc42.example.mama.unmarshaller`

5.2.2 Whitebox Campaign Process Control, Level 2

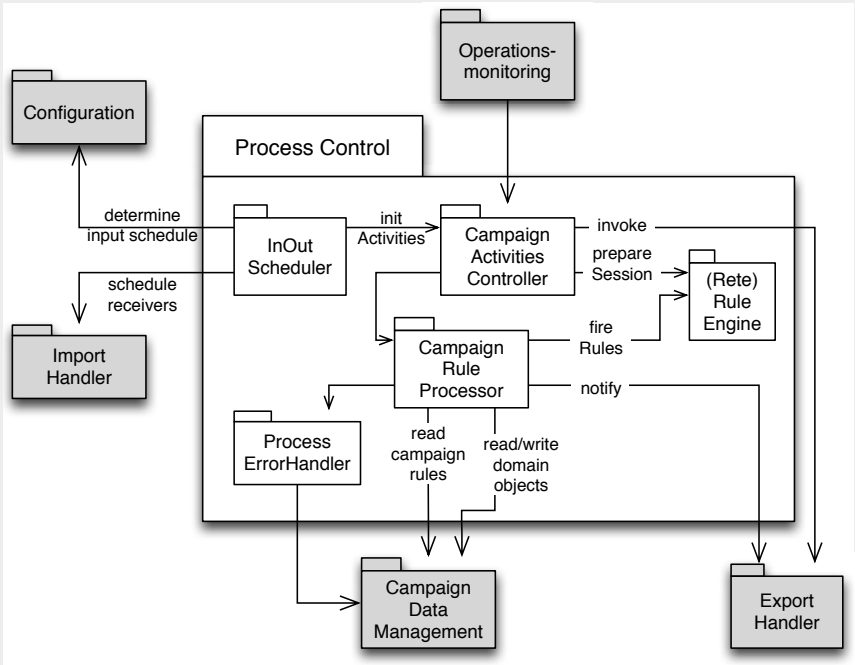


BILD 11.16 CampaignProcessControl, Level 2

Blackbox-Bausteine von CampaignProcessControl, Level 2

Blackbox	Bedeutung
InOutScheduler	Steuert den Aufruf der für eine Kampagne jeweils benötigten Input- und Output-Aktivitäten (abhängig von Mandanten, Partnern sowie den jeweils vorhandenen Daten).
CampaignActivitiesController	Steuert die Aktivitäten einer Kampagne (genauer: Output- und interne Aktivitäten). Implementiert dazu den in Abschnitt 8.1 beschriebenen Algorithmus.
CampaignRuleProcessor	Bestimmt aus den für eine Kampagne (bzw. einzelne Marktteilnehmer) vorhandenen Daten die nächsten (Output-)Aktivitäten. Basiert auf kampagnenspezifischer Prozess- und Regeldefinition – nutzt eine Regelmaschine; siehe Abschnitt 8.3.
ProcessErrorHandler	Behandelt sämtliche Fehler von Kampagnendaten (Stammdaten, Marktteilnehmerdaten, Rohergebnisdaten) und löst eine Output-Aktion mit den jeweils betroffenen Partnern oder Mandanten aus („Clearing“).
(Rete)RuleEngine	Open-Source-Regelmaschine zur Auswertung der kampagnenspezifischen Ablauf- und Plausibilitätsregeln; siehe [Drools].

Für weitere Details siehe Abschnitt 8.3 (Konzept zu Geschäftsregeln).

5.3 MaMa Bausteinsicht Level 3

Dieser Abschnitt detailliert exemplarisch einen Blackbox-Baustein der Ebene 3.

5.3.1 Whiteboxsicht Baustein „Receiver“, Level 3

Der Baustein „Receiver“ nimmt die Daten entgegen, die an den Außenschnittstellen von MaMa physisch angeliefert werden. Gleichzeitig koordiniert Receiver die übrigen Aufgaben der Input-Verarbeitung. Diese Vermischung von Verantwortlichkeiten ist riskant und führt in der Entwicklung zu hohen Änderungsaufwänden bei der Einführung neuer Daten- oder Dateiformate. Siehe die Anmerkungen in Abschnitt 5.2.1.1 (Blackbox `Receiver`) sowie Kapitel 11 (Risiken).

Anmerkung: Die nachfolgende Whiteboxsicht beschreibt die Zielvorstellung, die nach dem Refactoring von Receiver erreicht werden soll.

Funktional werden in Bild 11.17 die Aufgaben von „Datenempfang“ (div. Listener sowie ServiceToFileConverter) getrennt von der Koordination der weiteren Verarbeitung (FileProcessor, FileSplitter, RecordProcessor). Die letztgenannten Bausteine könnten künftig in einen eigenständigen Level-2-Baustein (Vorschlag: InputController) ausgelagert werden.

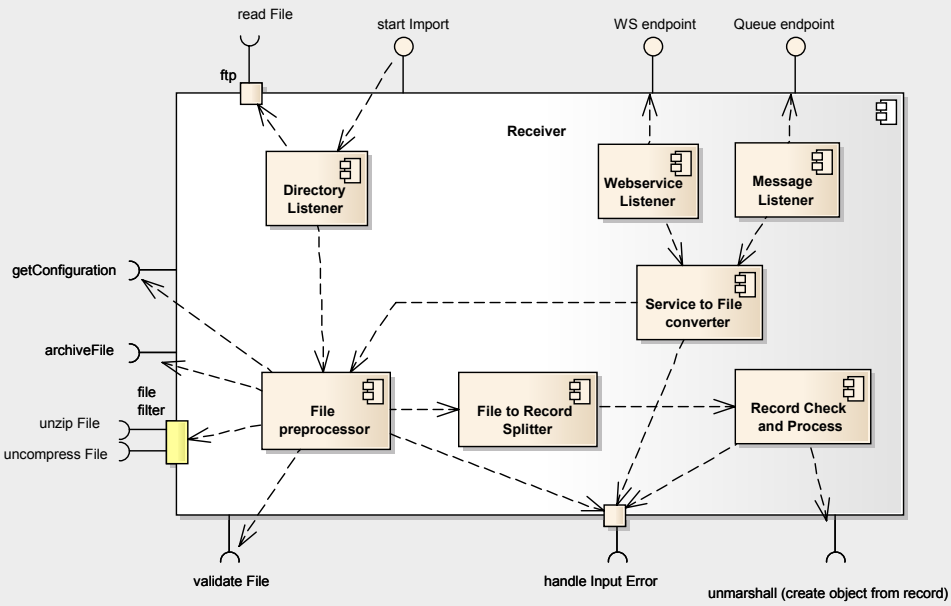


BILD 11.17 Whitebox Receiver, Level 3 (Zielstruktur, zurzeit nicht realisiert)

Blackbox-Bausteine von Receiver, Level 3

Blackbox	Bedeutung
DirectoryListener	Überprüft periodisch ein vorgegebenes Verzeichnis im Dateibaum auf die Existenz neuer Dateien eines kampagnen- oder aktivitätsspezifischen Namensmusters. Wird eine Datei gefunden, so übergibt der Listener diese an den FileProcessor. Für jede Input-Aktivität der Kampagne wird eine eigene Instanz dieses Bausteins als Thread (respektive Prozess) gestartet.
WebServiceListener	Empfängt Daten über Web-Services. Die zugehörige WSDL-Beschreibung inklusive URL und Service-Namen wird in der Konfiguration der Kampagne festgelegt. Im Falle des fehlerfreien Empfangs einer Nachricht wird diese an ServiceToFileConverter übergeben.
MessageListener	Empfängt Nachrichten von TIBCO beziehungsweise MQ-Series und übergibt diese an ServiceToFileConverter. Zurzeit nicht implementiert.
ServiceToFileConverter	Konvertiert Eingabedatenströme (Messages, Web-Service-Nachrichten) in Dateien, zur Weiterverarbeitung durch FileProcessor.
File PreProcessor	Koordiniert die Verarbeitung von Eingabedateien. Nach Bearbeitung Weitergabe an File-toRecord-Splitter.
File to Record Splitter	Zerlegt eine Datei mit Hilfe eines Parsers in einzelne Datensätze: Bei CSV-Dateien sind das einzelne Zeilen, bei XML-Dateien ist die Zerlegung abhängig vom XML-Schema. Fix-Format-Dateien werden auf Basis ihrer Record-Definition zerlegt. Übergibt eine Liste von Strings an Record-Check-and-Process.
Record Check and Process	Verarbeitet einzelne Datensätze.

6 Laufzeitsicht

Anmerkung: Als Alternative zu Sequenzdiagrammen beschreibe ich einige der folgenden Szenarien in Textform. Sie sollten im Einzelfall entscheiden, ob Ihre Leser besser mit einem Diagramm oder mit Text arbeiten können. In manchen Fällen können Sie bei Laufzeitszenarien übrigens auch Quellcodefragmente in die Architekturdokumentation übernehmen!

6.1 Szenario: Schematischer Input von Daten

Dieses Szenario beschreibt schematisch, wie MaMa die über die Außenschnittstellen (Kundenstamm-, Kampagnen- oder Rohergebnisdaten) (als Datei) gelieferten Daten entgegennimmt. Dieses Szenario gilt nicht für die Datenlieferung mittels Webservices! Eine Verfeinerung für den Import einer CSV-Datei finden Sie in Abschnitt 6.2.

1. Mandant oder Partner liefert Datei (per sftp-Upload) in vorab definiertes Verzeichnis.
2. Ein `DirectoryListener` (aus dem Baustein `Receiver`) erkennt die Einlieferung und startet den Baustein `FileProcessor` (der startet `FileFilter`, `Validator` und `FileArchiver`).
3. `FileProcessor` importiert die gelieferten Daten und ruft für jeden (syntaktisch korrekten) Datensatz `CampaignRuleProcessor` auf. Für fehlerhafte Datensätze wird eine Fehlermeldung erzeugt und ein Clearing beim Mandant oder Partner beauftragt.
4. `CampaignRuleProcessor` ermittelt für den aktuellen Datensatz die notwendigen Folgeaktivitäten.

6.2 Szenario: Import einer CSV-Datei

Voraussetzungen:

- Die betroffene Kampagne ist funktionsfähig konfiguriert.
- Mandant und Partner verfügen über entsprechende Rechte, um Dateien bei MaMa einliefern zu können. Anmerkung: Datenlieferung per Webservice ist nicht Bestandteil dieses Szenarios.

Ablauf:

1. Mandant oder Partner liefert Datei (per sftp-Upload) in vorab definiertes Verzeichnis.
2. MaMa liest aus `CampaignDataManagement` die Metadaten für diesen Mandanten (u. a. Filter- und Prüfregeln für Dateien, Syntaxregeln für Datensätze).
3. `Receiver` übergibt Datei mit Statusinfo (Datum, Datei-Hash) an `FileArchiver` – der sie in das Permanent-Archiv kopiert.
4. `FileProcessor` prüft Metadaten der Datei; nicht ok: Clearingfall.
5. `FileFilter` wendet alle konfigurierten Filter (decompress, decrypt etc.) an; nicht ok: Clearingfall.
6. `FileSplitter` zerlegt in Einzelsätze.
7. `RecordProcessor` verarbeitet Datei satzweise und koordiniert `CampaignRuleProcessor` und `UnMarshaller`:
   - 7.1 `UnMarshaller` lässt übergebene Daten durch Validator syntaktisch validieren; nicht ok: Clearingfall.
   - 7.2 Falls Daten ok, sucht anhand des Primärschlüssels im Repository nach bereits vorhandener Instanz dieses Datensatzes.

  - 7.2.a: Nicht vorhanden: legt neues Objekt (persistent) an.
  - 7.2.b: Vorhanden: Update mit gerade gelieferten Daten (offen: Historisierung).
- 7.3 CampaignRuleProcessor wendet Geschäftsregeln der Kampagne auf aktuellen Datensatz an; primär werden dabei die notwendigen Folgeaktivitäten ermittelt (siehe Abschnitt 8.1 – Ablaufsteuerung).

8. Erzeuge Statusmeldung für Mandanten und übergebe diese via „Output“.

Ergebnisse dieses Szenarios:

- Die in der Datei gelieferten korrekten Datensätze sind in den MaMa-Datenbestand der betroffenen Kampagne übernommen.
- Sämtliche fehlerhaften Daten werden als Clearingfälle zurück an den Lieferanten der Datei gemeldet.

6.3 Szenario: Import einer CSV-Datei (alternative Darstellung)

Zum direkten Vergleich finden Sie nachfolgend das oben geschilderte Szenario als Sequenzdiagramm.

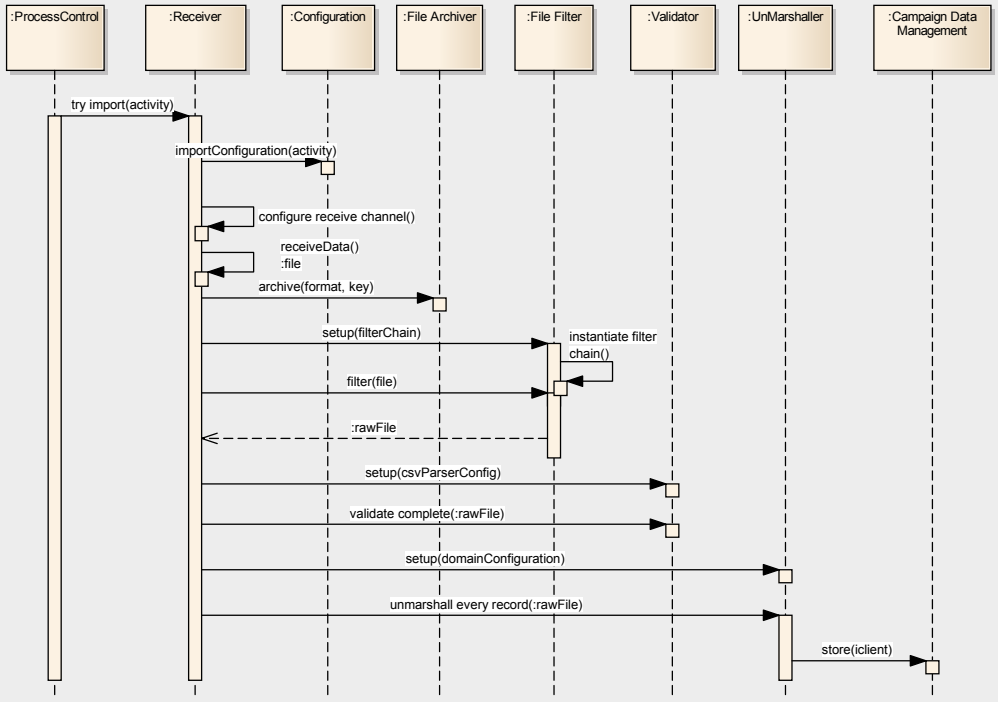


BILD 11.18 Laufzeitszenario: Import einer CSV-Datei mit Filterung

7 Verteilungssicht

Die Verteilungssicht von MaMa ist nahezu trivial, da die meisten Implementierungsbausteine der Bausteinsicht in ein einzelnes Java-Klassenarchiv (jar) zusammengefasst werden.

Die graphischen Teile von MaMa (Configuration, OperationsMonitoring) bilden dabei Ausnahmen.

Verteilungsartefakte für MaMa

Verteilungsartefakt	Baustein (aus Bausteinsicht)
MaMa<CampaignName>	Alle, bis auf OperationsMonitoring.
MaMaOperationsMonitor	OperationsMonitoring, CampaignDataManagement.
MaMaConfigurator	Configuration, CampaignDataManagement, CampaignProcessControl

Anmerkung: Auf die technischen Leistungsdaten der zugrunde liegenden Hardware habe ich in diesem Beispiel verzichtet.

8 Querschnittliche Konzepte

8.1 Ablaufsteuerung

Das für MaMa zentrale Konzept der konfigurierbaren Ablaufsteuerung basiert wesentlich auf dem Begriff der *Aktivität*, jeweils bezogen auf einzelne Marktteilnehmer. MaMa kennt Output-, Input- sowie interne Aktivitäten:

- Output-Aktivitäten übergeben Teile der Daten über Marktteilnehmer (Beispiel: Name, Adresse, Vertrags- oder Tarifdaten) an einen Partner, damit dieser Partner eine für die Kampagne notwendige Aktion durchführen kann. Beispiele für solche Aktionen:
  - Einen Brief an die jeweiligen Marktteilnehmer schreiben (Partner: Druckerei).
  - Die Marktteilnehmer per Telefon kontaktieren (Partner: Call Center).
  - Auf den Besuch oder Anruf der Marktteilnehmer vorbereitet sein (Partner: Call Center sowie Filialen).

  Output-Aktivitäten haben immer einen definierten Adressaten. Als Konsequenz mancher Output-Aktivitäten liefern die betreffenden Partner zu einem späteren Zeitpunkt Rohergebnisdaten an MaMa zurück, die über Input-Aktivitäten eingelesen werden.
- Interne Aktivitäten verändern den Zustand oder die Daten von Marktteilnehmern, ohne dabei einen externen Partner zu benutzen. Hierzu gehören Datentransformationen, Berechnungen, Aggregation mehrerer Datensätze (beispielsweise um Eltern und Kinder zu Familien zusammenzuführen), Löschungen und Statusänderungen.
- Input-Aktivitäten nehmen Daten von den Außenschnittstellen entgegen.

Schematisch verlaufen solche Aktivitäten nach folgendem Grundmuster:

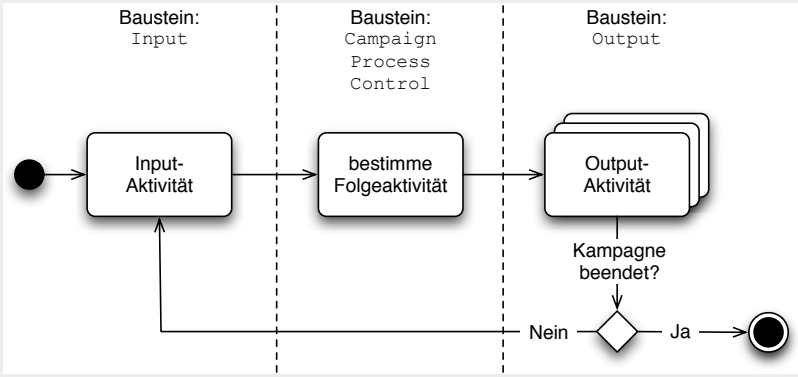


BILD 11.19 Ablaufsteuerung von MaMa – Abfolge von Aktivitäten (schematisch)

Während jeder Input-Aktivität bestimmt MaMa die notwendigen Folgeaktivitäten auf Basis der Geschäftsregeln der Kampagne sowie der Daten jedes Marktteilnehmers.

Lassen Sie mich diesen Ablauf anhand des bereits bekannten Beispiels „Vertrags-/Tarifänderungen eines Telekommunikationsunternehmens“ kurz erläutern:

- In einer Input-Aktivität nimmt MaMa vom Mandanten MoF die Vertragsdaten eines (hypothetischen) Kunden „Billy Beispiel“ entgegen (Tarif „T-2000“).
- Als Folgeaktiväten bestimmt MaMa die Aktiväten „DruckeBrief“, „InfoAnFilialen“ sowie „InfoAnCallCenter“.
  - „DruckeBrief“ bedeutet, dass für diesen Kunden ein Schreiben zu drucken und zu versenden ist, mit dem er auf einen möglichen Tarifwechsel in den „Tarif Minipreis“ hingewiesen wird. Diesen Tarifwechsel muss der Marktteilnehmer durch seine Unterschrift bestätigen, die Tarifdetails auf einem mitgeschickten Formblatt auswählen.
  - „InfoAnFilialen“ bedeutet, dass eine Kopie des erstellten Briefs an die Filialen von MoF übermittelt wird.
  - „InfoAnCallCenter“ bedeutet, dass dem Call-Center eine Kopie des erstellten Anschreibens übermittelt wird. Falls sich der Marktteilnehmer bei Rückfragen nun telefonisch an das Call-Center wendet, sind dort seine Vertrags- und Tarifdaten bekannt.
- Nun führt MaMa die Output-Aktivität „DruckeBrief“ aus und übermittelt die betroffenen Daten an den Druckdienstleister.

Die technische Lösung von MaMa basiert auf einer Liste von Folgeaktivitäten, die jedem Marktteilnehmer zugeordnet ist – den entsprechenden Auszug aus dem fachlichen Modell entnehmen Sie Bild 11.20.

Im Objektdiagramm aus Bild 11.21 erkennen Sie, wie ein Marktteilnehmer über eine Input-Aktivität seinen „Vertrag“ erhält, aus dem MaMa dann anhand der kampagnenspezifischen Geschäftsregeln die Folgeaktivitäten *DruckeBrief*, *InfoAnFilialen* und *InfoAnCallCenter* ermittelt.

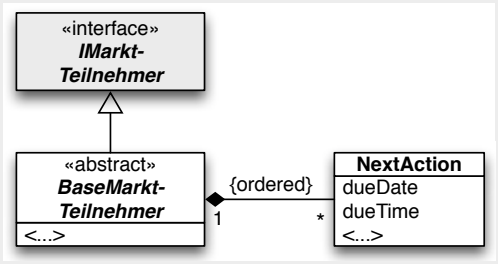


BILD 11.20
Fachliches Modell von Marktteilnehmer und Folgeaktivitäten

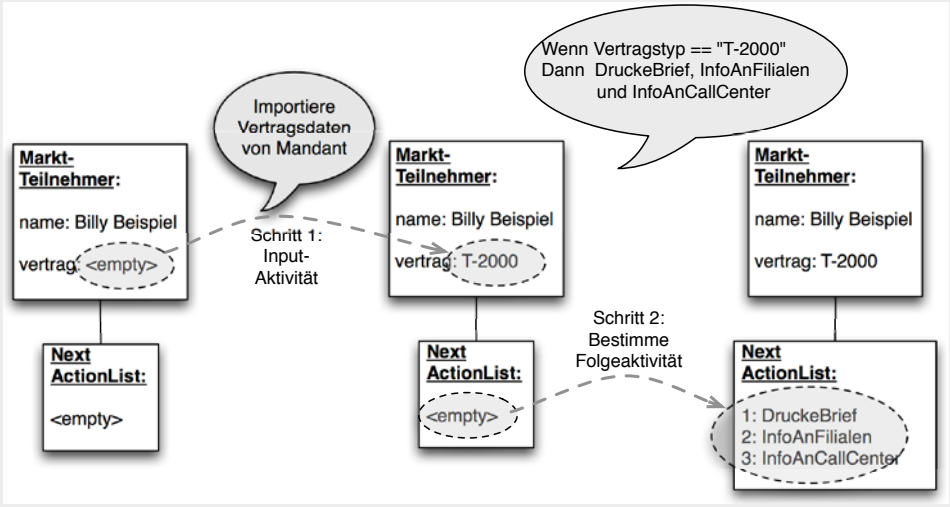


BILD 11.21 Beispiel für Ablaufsteuerung von Aktionen

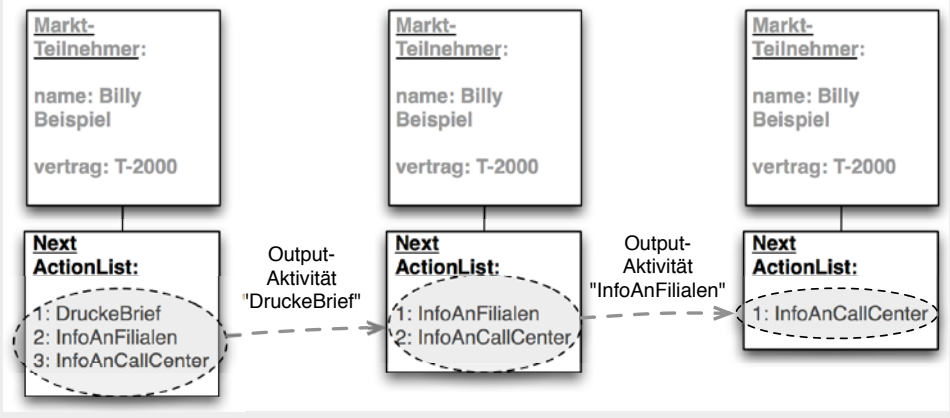


BILD 11.22 Beispiel: Update der NextActionList nach Output-Aktivitäten

MaMa wird nun die für diese Kampagne möglichen Aktionen durchführen und dabei nach jedem Schritt die Liste der möglichen Folgeaktivitäten von „Billy Beispiel“ anpassen:

Allgemein folgt MaMa einem einfachen Schema bei der Bearbeitung von (Output- und internen) Aktivtäten:[10]

1. Ermittle, welche Output-Aktivitäten in dieser Kampagne möglich sind.
2. Für all diese Aktivitäten:
   - 2.1 Ermittle diejenigen Marktteilnehmer, deren NextActionList die aktuelle Aktivität enthält.
   - 2.2 Führe die Aktivität für diesen Marktteilnehmer aus. Bei Output-Aktivitäten werden dabei Daten (Attribute) dieses Marktteilnehmers zur Ausgabe an einer Außenschnittstelle gesammelt. Bei internen Aktivitäten werden Aggregationen, Berechnungen, Transformationen oder Löschungen von Daten durchgeführt.
   - 2.3 Lösche die aktuelle Aktivität aus der NextActionList dieses Marktteilnehmers.
   - 2.4 Wende die Geschäfts-/Plausibilitätsregeln der Kampagne auf den aktuellen Marktteilnehmer an (siehe hierzu das Konzept zu Geschäftsregeln in Abschnitt 9.3).

Dieser Algorithmus wird von `CampaignActivitiesController` im Paket CampaignProcess-Control implementiert. Er wird zeitgesteuert oder periodisch ausgelöst vom InOutScheduler. Siehe dazu Abschnitt 5.2.2.

8.2 Produktfamilie, Persistenz und Generierung

MaMa bildet die Basis für eine Produktfamilie, von der für konkrete Kampagnen jeweils spezifische Ausprägungen instanziiert werden. Dies geschieht durch folgende Schritte:

1. Erweiterung des MaMa-Domänenmodells (einen Auszug davon finden Sie in Bild 11.23).
2. Generierung des kampagnenspezifischen Quellcodes, bestehend aus den spezialisierten Domänenklassen, Persistenzklassen (Zugriffs-, Transferklassen, DB-Tabellen u. Ä.). In diesem Schritt werden insbesondere der komplette Level-1-Baustein `Campaign Data Management` (siehe Bild 11.14) sowie die notwendigen DB-Skripte generiert!

[10] Falls Sie das lieber in Pseudocode lesen:

```
// ermittle, welche Aktivitäten in dieser Kampagne möglich sind
Iterator aktionen = getListOfOutputActivities( kampagne );
// führe alle Aktivitäten aus
while (aktionen.hasNext()) {
      currentAction = aktionen.next()
      // selektiere diejenigen Marktteilnehmer, deren
      // NextActionList den Eintrag currentAction enthält
         teilnehmer = getActionableTeilnehmer( currentAction )
      // führe currentAction aus
         performAction( currentAction, teilnehmer)
         applyRules( kampagne, teilnehmer )
}
```

3. Kampagnenspezifische Konfiguration:
   - 3.1 Festlegung der möglichen Input- und Output-Aktionen (d. h. Füllen der CampaignActionTypes)
   - 3.2 Definition der konkreten Syntax aller ein- und ausgehenden Daten (CSV-Formate, XML-Schemata etc.)
   - 3.3 Definition der Ein-/Ausgabeprotokolle, inklusive Metadaten, für Auftraggeber sowie sämtliche Partner
   - 3.4 Definition der kampagnenspezifischen Geschäftsregeln

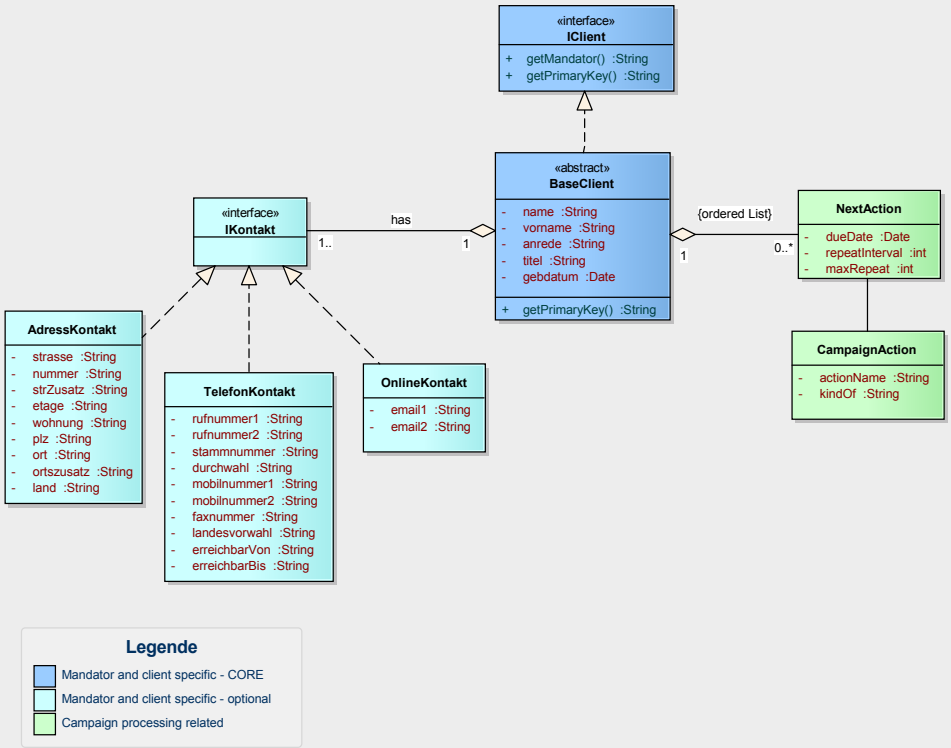


BILD 11.23 Domänenmodell als Basis der MaMa-Produktfamilie (MaMa-Core)

Danach kann die Kampagne gestartet werden.

Die Konfiguration aus Schritt 3 geschieht zum Teil über eine Konfigurationsoberfläche (siehe Baustein „Configuration“ in Abschnitt 5.1.4).

Beispiel eines angepassten MaMa-Domänenmodells (hier am Beispiel einer hypothetischen Versicherung):

Beachten Sie in Bild 11.24 insbesondere die neu hinzugekommenen Entitäten MorbidInsurance, die von der Core-Entität BaseClient ableitet und die kampagnen- bzw. mandantenspezifischen Attribute hinzufügt.

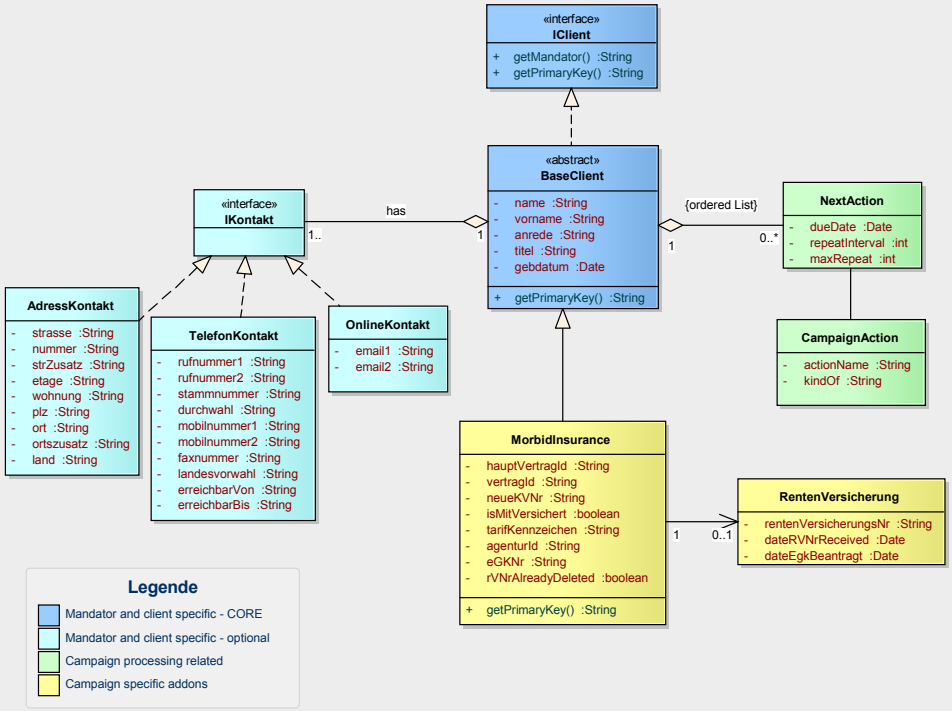


BILD 11.24 Konkretes MaMa-Domänenmodell, abgeleitet aus MaMa-Core

8.3 Geschäftsregeln

Der Auftraggeber hatte ursprünglich eine grafische Modellierung von Geschäftsregeln gefordert, was zurzeit jedoch nur einige kommerzielle Regelmaschinen in praxistauglicher Form leisten. Da seitens des Auftraggebers eine Open-Source-Lösung favorisiert wurde ([Drools]), beschreibt MaMa die Regeln einer Kampagne in textueller Syntax.

Regeln haben die Form „Wenn <X> dann <Y>“, wobei X und Y beliebige Ausdrücke in Java sein können. Mehr zur Regelsprache unter [Drools] bzw. im Java-Paket `de.arc42.example.mama.campaignruleprocessor`.

Die Bausteine `CampaignRuleProcessor` sowie `CampaignActivitiesController` kapseln sämtliche Details der Regelmaschine gemäß dem folgenden Konzept eines *Rule Engine Wrappers*):

Ziel dieses Konstruktes ist einerseits hohe Performance durch Wiederverwendung vorbereiteter „Working Memory Sessions“ (bei denen dann pro bearbeitetem Datensatz nur noch die jeweiligen Fakten über die Methode `addFact()` eingefügt werden müssen und sämtliche Regeln aus der Datei CampaignRules.drl bereits geladen sind), andererseits die potenzielle Austauschbarkeit der Regelmaschine (was durch die Wahl der Drools-Regelsprache sehr unwahrscheinlich ist).

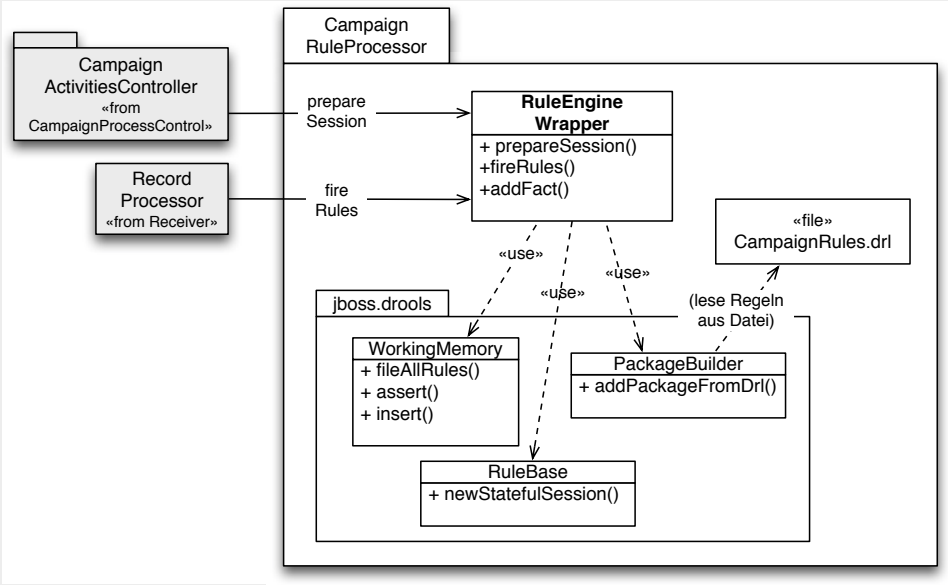


BILD 11.25 Rule Engine Wrapper zur Kapselung der Regelmaschine

8.4 Ausnahme- und Fehlerbehandlung

Für MaMa relevante Ausnahmen und Fehler betreffen hauptsächlich folgende Kategorien von Fehlern, die alle an Eingabeschnittstellen auftreten:

- Fehlerhaft übertragene Daten (z. B. falsche Prüfsummen)
- Syntaktisch falsche Daten (z. B. fehlende Attribute oder falsche Datentypen)
- Nicht zuzuordnende Daten (fehlerhafte Primärschlüssel)
- Verletzungen kampagnenspezifischer Konsistenzbedingungen

MaMa selbst kann diese Fehler nicht korrigieren, sondern lediglich dem jeweiligen Sender von Daten mitteilen.

Die Erkennung dieser Fehler delegiert MaMa an verschiedene Bausteine – die wichtigsten davon sind `ProcessErrorHandler` (aus CampaignProcessControl) und `InputErrorHandler` (aus Input).

9 Entwurfsentscheidungen

9.1 Kein CRM-Werkzeug

- *Entscheidung:* MaMa verwendet kein CRM-Werkzeug als technische Grundlage.
- *Begründung:* Kommerzielle CRM-Werkzeuge erfüllen die funktionalen Anforderungen von MaMa nicht.

9.2 Kein ETL-Werkzeug

- *Entscheidung:* MaMa verwendet kein ETL-Werkzeug zum Import von Daten oder Dateien.
- *Begründung:* Zu hohe Lizenz- und Betriebskosten kommerzieller ETL-Werkzeuge.
- *Offen:* Die Open-Source-Suite [Pentaho] enthält eine leistungsfähige ETL-Komponente namens Kettle. Diese sollte möglichst zeitnah auf ihre Verwendbarkeit in MaMa geprüft werden, bevor zusätzliche Aufwände bei der Weiterentwicklung der Validatoren entstehen.

10 Qualitätsszenarien

<Entfällt im Beispiel>

11 Risiken

ID	Problem/Risiko	Auswirkung	Beschreibung
R1	Input-Kette ist selbst entwickelt.	Überhöhte Aufwände bei Implementierung	Eigenentwicklung der gesamten Input-Kette erscheint zurzeit sehr aufwendig, könnte durch Einsatz eines Open-Source-ETL-Tools [Pentaho] vereinfacht werden. Siehe Abschnitt 9.2.
R2	Hohe Komplexität beim Reporting von fehlerhaften Input-Dateien	Hohe Entwicklungs- und Betriebsaufwände, schlechte Verständlichkeit des Input-Packages	Schlechte Fehlerbehandlung bei Validierung von Eingabedaten. Der Validator behandelt Fehler (z. B. Syntaxfehler in Eingabedateien) zurzeit völlig anders als sämtliche anderen Input-Bausteine. Das führt zu Mehraufwand und erhöhter Komplexität beim Reporting und Clearing.
R3	Receiver-Baustein funktional überlagert	Hohe Wartungsrisiken bei Änderungen an Receiver	Siehe Abschnitt 5.2.1.1. Receiver übernimmt zurzeit zu viele Aufgaben und sollte in zwei Funktionsblöcke zerlegt werden (`Receiver` und `InputController`).

12 Glossar und Referenzen

Begriff	Bedeutung
CRM	Customer Relationship Management. Siehe de.wikipedia.org/wiki/Customer_Relationship_Management.
Endkunde	Synonym: Kunde, Marktteilnehmer, Verbraucher, Einzelperson, Familie oder sonstige Personengruppe, die in einem Vertrags- oder Kundenverhältnis mit einem Mandanten steht.
ETL	Extract-Transform-Load. Aus dem Data-Warehousing bekannte Familie von Werkzeugen zum Import von Daten. Siehe de.wikipedia.org/wiki/ETL-Prozess.
Mandant	Synonym: Anbieter. Anbieter von Produkten oder Dienstleistungen im Massenmarkt. Für Mandanten wickelt MaMa die (virtuelle) Kundenbeziehung im Rahmen einzelner Kampagnen ab.
Kampagne	Zeitlich befristete Aktion eines Mandanten, in der ein definierter Waren- oder Informationsaustausch zwischen Mandant und ausgewählten Endkunden oder Endkundengruppen stattfindet. Im Sinne von MaMa finden Kampagnen mit dem Ziel statt, Änderungen an bestehenden Vertragsverhältnissen herbeizuführen (z. B. andere Tarife, Vertragslaufzeiten, Produkt- oder Leistungskombinationen oder Ähnliches).
Partner	(Synonym: Dienstleistungspartner) Organisation oder Unternehmen, das Dienstleistungen innerhalb von Kampagnen übernimmt. Partner sind über elektronische Schnittstellen in die Kampagnen eingebunden. Typische Partner sind Druck-, Scan- und Telefondienstleister.
Rohergebnisdaten	Daten, die von Partnern an MaMa geliefert und dort aggregiert, umgerechnet oder angereichert werden, um die Ergebnisdaten einer Kampagne zu ermitteln.

Referenzen

Quelle	Beschreibung
[Drools]	Jboss Drools Rule Engine: Online: *www.jboss.org/drools/*
[HoneyComb]	Sun Microsystems: OpenSolaris HoneyComb Fixed Content Storage, frei verfügbares Speicher-/Archivsystem für x86-Systeme. Online: *www.opensolaris.org/os/project/honeycomb/*
[Olivieri]	Ricardo Olivieri: *www.ibm.com/developerworks/java/library/j-drools/*
[Pentaho]	Pentaho Open Source Business Intelligence. Online: *www.pentaho.com/*
[Pohl+05]	K. Pohl, G.Böckle, F. v. d. Linden: Software Product Line Engineering. Springer 2005. Eine fundierte Einführung in Produktfamilien, deren Modellierung und Entwicklung.
[Validator]	Apache Commons Validator Framework: Online: *commons.apache.org/validator/*

Anmerkungen zum Projekt

Das System MaMa durfte ich in der Zeit ab 2004 irgendwo in Deutschland wirklich (mit) entwerfen und seine Umsetzung begleiten. In der Realität waren die umgesetzten Geschäftsprozesse leicht umfangreicher als hier dargestellt. Die Namen des Systems und der Auftraggeber habe ich absichtlich verändert.

Auch in der Realität bildet MaMa eine Familie komplexer Geschäftsprozesse ab – eine so genannte Produktfamilie. Es setzt innerhalb dieser Prozesse gewisse Ähnlichkeiten voraus, die für viele CRM-Kampagnen[11] erfahrungsgemäß gegeben sind – beispielsweise die in Abschnitt 9.1, Bild 11.17 dargestellte Abfolge von Input- und Output-Aktivitäten.

Das System befindet sich bei seinen zufriedenen Kunden und Mandanten seit 2005 im produktiven Einsatz.

[11] Die CRM-Experten unter den Lesern mögen mir die starke Vereinfachung des Themas verzeihen. Mir ist bewusst, dass sich hinter „CRM“ noch weit mehr verbirgt, als ich in diesem Beispiel dargestellt habe.

12 Werkzeuge für Softwarearchitekten

Wer als Werkzeug nur einen Hammer hat,
sieht in jedem Problem einen Nagel!

Paul Watzlawik

Fragen, die dieses Kapitel beantwortet:

- Welche Kategorien von Werkzeugen können helfen?
- Nach welchen Kriterien könnte ich solche Werkzeuge auswählen?

Eine kleine Warnung vorweg: Sie werden in diesem Kapitel nichts über konkrete Werkzeuge lernen, keine Bedien- oder Einsatzhinweise finden, keine Tipps, wie Sie mit Tool X Ihr Problem Y lösen.

Kategorien von Werkzeugen

Vielmehr möchte ich versuchen, etwas Systematik in die Vielfalt der Werkzeuglandschaft zu bringen – und Ihnen das Spektrum aller möglicherweise hilfreichen Werkzeugkategorien vorstellen. Dazu erfahren Sie ein wenig über mögliche Auswahlkriterien.

12.1 Kategorien von Werkzeugen

Werkzeuge zum Anforderungsmanagement

Anforderungsmanagement sollte über den gesamten Lebenszyklus von Systemen praktiziert werden – in Form von Lastenheften oder Backlogs, mit Use-Cases, User-Stories oder anderen Formen von funktionalen und nichtfunktionalen Anforderungen. Es geht hierbei um Erhebung, Erfassung, Analyse, Dokumentation und Management von Anforderungen.

Diese Werkzeuge sollten textuelle und/oder grafische Darstellung ermöglichen, Redundanzen reduzieren helfen sowie die Rückverfolgbarkeit zwischen Anforderungen und Architektur/Code erlauben.

Herausforderungen sind insbesondere Mehrbenutzerfähigkeit, Zusammenführen/Merge von Modellen sowie die Unterstützung von Versions- und Konfigurationswerkzeugen.

Modellierungswerkzeuge

Modellierungswerkzeuge können fachliche und technische Modelle von Software sowie Anforderungs- und Problemdomänen darstellen. Sie helfen, (meist grafische) Modelle zu erstellen und zu pflegen.

Diese Kategorie von Werkzeuge unterstützt ab der Ebene von Geschäftsprozessen bis hin zur Modellierung von Klassen, Detailabläufen oder Zustandsübergängen beliebig abstrakte oder detaillierte Darstellung statischer und dynamischer Abstraktionen.

Herausforderungen dieser Werkzeugkategorie sind Mehrbenutzerfähigkeit, Vergleich und Analyse von Modellen, Trennung von Modellobjekten und deren grafische Darstellung in Diagrammen, explizite Metamodelle sowie ein Kompromiss aus Standard-Kompatibilität (etwa zu UML) und Pragmatik (etwa: Freiform-Diagramme). Hilfreich können Reverse-Engineering-Fähigkeiten sein sowie die Generierung von Dokumenten aus Modellen.

Werkzeuge zur Generierung (von Code oder sonstigen Artefakten)

Generierungswerkzeuge können ausgehend von abstrakten Beschreibungen (grafischen oder textbasierten Modellen in standardisierten oder domänenspezifischen Sprachen) beliebige Artefakte generieren. Primär finden sie Anwendung zur Erzeugung von Quellcode, Datenbanktabellen oder -skripten, Konfigurationsdateien, Lexern oder Parsern, XML-Schemata oder in Form von Webservice-Unterstützung. Interessant für Entwickler ist die Möglichkeit der programmatischen Nutzung (über Programmierschnittstellen oder die Integration in Build-Prozesse).

Herausforderung dieser Werkzeuge ist die häufig geringe Akzeptanz bei Entwicklern, die aufgrund des hohen Leistungsumfangs lernintensive Bedienung, die Flexibilität der Generierung beziehungsweise Transformationen sowie die Unterstützung frei definierbarer oder veränderbarer Metamodelle.

Werkzeuge zum Code-Management/Implementierungswerkzeuge

Diese Tools unterstützen Sie bei der Erstellung, Verarbeitung, Modifikation, Prüfung, Übersetzung und Ausführung von Quellcode. Primär zähle ich jede Art von Entwicklungsumgebung hierzu, aber auch (syntaxbewusste) Editoren, Refactoring-Tools, Debugger und alle Hilfsmittel rund um die Versionierung (siehe dazu auch das Thema Build- und Konfigurationsmanagement).

Die Unterstützung von Merging, Branching sowie garantiert verlustfreie Speicherung sämtlicher Versionen aller kontrollierten Artefakte für eine Vielzahl von Benutzern zählt zu den Grundforderungen. Für die Akzeptanz sind Verbreitung, Verständlichkeit sowie Anpassbarkeit an spezielle Aufgaben/Anforderungen hilfreich.

Werkzeuge zur statischen Codeanalyse

Diese Tools prüfen statische Eigenschaften von Quellcode – beispielsweise Abhängigkeiten (Kopplung), Komplexität, Kohäsion, Größe, Testabdeckung oder Einhaltung von Programmierkonventionen.

Eine große Herausforderung dieser Kategorie liegt im heute verbreiteten *polyglott programming*, d. h. der Unterstützung unterschiedlicher Programmiersprachen. Weiterhin hat die

starke Verbreitung von Dependency-Injection Frameworks die direkte Analyse von Abhängigkeiten deutlich erschwert – weil Abhängigkeiten in solchen Frameworks dynamisch per Konfiguration aufgebaut werden können.

Werkzeuge zur Laufzeitanalyse von Software

Laufzeit- oder dynamische Analyse betrachtet Softwaresysteme während ihrer Ausführung. Wesentliche Arten von Resultaten sind Laufzeit- und Geschwindigkeitsmessungen, Zeitmessungen bestimmter Systemteile (absolut oder relativ), Messung von Speicher- oder Ressourcennutzung, Kommunikationsverhalten sowie statistische Auswertungen.

Die große Herausforderung besteht darin, durch die Messungen das Laufzeit- oder Speicherverhalten des Systems möglichst wenig zu beeinflussen sowie die Ergebnisse zielgruppengerecht aufzubereiten. In verteilten Systemen gestaltet sich die Korrelation der auf unterschiedlichen Knoten gemessenen Resultate beliebig schwierig. ☹

Werkzeuge zum Build- und Konfigurationsmanagement

Diese Werkzeuge erlauben die teilweise oder vollständige Automatisierung von Übersetzungs-, Paketierungs-, Prüf-, Test- oder Generierungsaufgaben von Software. Zu ihnen zählen Continuous-Integration-Systeme (die beispielsweise nach Commits im Versionsmanagement die benötigten Teile des Build-Prozesses gezielt anstoßen).

Schwierig für diese Werkzeuge ist der Kompromiss aus deklarativer und imperativer Darstellung von Build-Aufgaben, die Behandlung direkter oder transitiv abhängiger Artefakte und Bibliotheken sowie die Erreichung möglichst hoher Automatisierungsgrade.

Management-Aufgaben wie Inventarisierung sowie Wiederherstellung beliebiger Konfigurationen (inklusive Betriebssystem, benötigte Basissoftware und Datenbestände, Dokumentation sowie der eigentlichen Systeme) sind in manchen Fällen notwendig.

Werkzeuge zum Test von Software

Insbesondere automatisierte Tests geben ein frühes Feedback zu Strukturen, Konzepten, Schnittstellen und Implementierung bereits während der Konstruktion und Entwicklung. Unit-, Integrations- und Akzeptanztests erleichtern oft die Kommunikation mit Testern und auch Fachabteilungen. Laufzeit-, Last- und Performancetests während der Entwicklung erlauben Prognosen über das Zeit- und Ressourcenverhalten des fertigen Systems. Penetrations- und Sicherheitstests adressieren die Sicherheit der Systeme und beteiligten Daten.

Zu den Herausforderungen zählen die Verwaltung von Testfällen, Testdaten sowie Testergebnissen, die Unterstützung für den Test verteilter Systeme, der Test grafischer Oberflächen sowie lange laufender fachlicher Abläufe. Es hat sich als gute Praxis herausgestellt, Fremdsysteme über sogenannte Mocks zu entkoppeln, die zu Testzwecken das Verhalten dieser Fremdsysteme simulieren.

Ich halte die testorientierte Entwicklung (in Form von Test-Driven oder Acceptance-Test-Driven Development) für ein sehr hilfreiches Entwurfsinstrument, das die Qualität detaillierter Entwurfs- und Implementierungsentscheidungen beträchtlich steigern kann.

Werkzeuge* wie Spockframework, Cucumber, Fitnesse oder andere sollten zur Standardausstattung praktizierender Softwarearchitekten gehören.

* Siehe etwa *http://spockframework.org*, *http://cukes.info*, *http://fitnesse.org* oder [Freeman+10].

Werkzeuge zur Dokumentation

Werkzeuge zur Dokumentation sollen uns bei der langfristigen Kommunikation von Entscheidungen, Strukturen, Konzepten sowie sonstiger relevanter Sachverhalte unterstützen.

Zu dieser Kategorie gehören text- und grafikbasierte Werkzeuge zur Erstellung, Pflege oder Generierung von Dokumenten.

Für alle Dokumentationswerkzeuge auf der Basis binärer Formate stellen Teamfähigkeit (parallele Bearbeitung) und Zusammenführung (merge) Herausforderungen dar. Rein textbasierte Tools haben dafür Probleme mit formalen oder gestalterischen Vorgaben (z. B. Corporate Layout, Corporate Design).

12.2 Typische Auswahlkriterien

Im realen Leben müssen Sie selbst die Menge der für Sie und Ihre Stakeholder relevanten Auswahlkriterien für Werkzeuge ermitteln – aber ein paar typische Vertreter beziehungsweise Oberbegriffe möchte ich Ihnen dazu mitgeben.

Technische Kriterien

Arbeitsergebnisse (Artefakte, Dokumente, Modelle) sollten in jedem Fall versioniert gespeichert werden können, d. h. fast alle Werkzeuge benötigen eine Integration mit Subversion, git und Co.

Da wir die meiste Software in Teams entwickeln, müssen oftmals mehrere Benutzer parallel mit Werkzeugen arbeiten – d. h. die sie sollten arbeitsteiliges Vorgehen unterstützen. Mehrere Bearbeiter sollten parallel agieren können, ohne sich gegenseitig zu behindern.

Organisatorische Kriterien

Lizenzmodalitätenkosten spielen bei der Auswahl von Werkzeugen oftmals eine bedeutende Rolle. Beachten Sie, dass eine gesamthafte Kostenbetrachtung von Werkzeugen neben den Anschaffungskosten auch die Kosten der Einführung, Konfiguration sowie eventuelle regelmäßige Wartungs- und Betriebskosten berücksichtigen sollte.

Für manche Produkte benötigen Sie professionelle Unterstützung – dann sollten Sie die Verfügbarkeit dieser Unterstützungsleistung sowie deren Kosten ebenfalls in Ihre Entscheidungskriterien aufnehmen.

Weiterhin spielen gerade im Open-Source-Umfeld die Lizenzbedingungen von Produkten oder Frameworks eine Rolle: Klären Sie, welche Nutzungs- und Vertriebsrechte Ihnen diese Lizenzen einräumen. Manche sind in Dual-Lizenz-Modellen erstellt, bei denen die kommerzielle Nutzung anderen Einschränkungen unterworfen ist als die private Nutzung.

Hüten Sie sich vor juristischen Fallstricken – und ziehen Sie im Zweifel einen in IT- und Lizenzrecht versierten Experten zurate.

Schließlich möchte ich Akzeptanz (im Entwicklungsteam) als ein wesentliches Kriterium für die Auswahl von Werkzeugen nennen. Nicht jedes Werkzeug passt zu jedem Team – und ohne Akzeptanz sind manche Werkzeuge zum Scheitern verurteilt. Akzeptanz können Sie durch gezieltes Coaching und Begleitung steigern – aber nur in seltenen Fällen gegen die Überzeugung eines Teams erzwingen.

13 iSAQB Curriculum

Fragen, die dieses Kapitel beantwortet:

- Was ist das International Software Architecture Qualification Board?
- Was nützt ein standardisiertes Curriculum für Softwarearchitekten?
- Welche Gebiete umfasst der iSAQB-Lehrplan?
- Wo können Sie die Lerninhalte des iSAQB-Lehrplans nachlesen?
- Welche Teile des iSAQB-Lehrplans deckt dieses Buch ab?

Viele Missverständnisse um Begriffe der Softwarearchitektur

Softwarearchitektur ist eine relativ junge Disziplin, über deren genauen Umfang und Ausgestaltung in der IT-Branche trotz vieler Publikationen immer noch unterschiedliche Meinungen kursieren. Die Aufgaben und Verantwortungsbereiche von Softwarearchitekten werden sehr unterschiedlich definiert und in Entwicklungsprojekten oftmals neu verhandelt.

Für andere Disziplinen wie Projektmanagement, Requirements Engineering oder Testen gibt es weitgehenden Konsens, was deren Arbeitsbereiche betrifft. Für sie bieten unabhängige Organisationen[1] Lehrpläne an, die klar beschreiben, welche Kenntnisse und Fähigkeiten eine entsprechende Ausbildung vermitteln muss.

Anfang 2008 haben sich verschiedene Hochschullehrer und Praktiker zum „International Software Architecture Qualification Board" (ein eingetragener Verein) zusammengeschlossen, um einen standardisierten Rahmen für die Ausbildung von Softwarearchitekten zu definieren:

> *„Die Grundlagenausbildung[2] für Softwarearchitekten vermittelt das notwendige Wissen und Fähigkeiten, um für kleine und mittlere Systeme ausgehend von einer hinreichend detailliert beschriebenen Anforderungsspezifikation eine dem Problem angemessene Softwarearchitektur zu entwerfen und zu dokumentieren, die dann als Implementierungsgrundlage bzw. -vorlage genutzt werden kann. Teilnehmer dieser Ausbildung erhalten das Rüstzeug, um problembezogene Entwurfsentscheidungen auf der Basis ihrer vorab erworbenen Praxiserfahrung zu treffen."* [ISAQB]

[1] Für Tester: ISTQB (*www.istqb.org*); für Requirements Engineering: *www.certified-re.de*; für Projektmanagement gibt es mehrere solcher Organisationen mit leicht unterschiedlichem Fokus, etwa das PMI (*www.pmi.org*).

[2] Im Sprachgebrauch des iSAQB heißt die Grundlagenausbildung „*Foundation Level*" – in Abgrenzung zum „*Advanced Level*" der fortgeschrittenen Architekten.

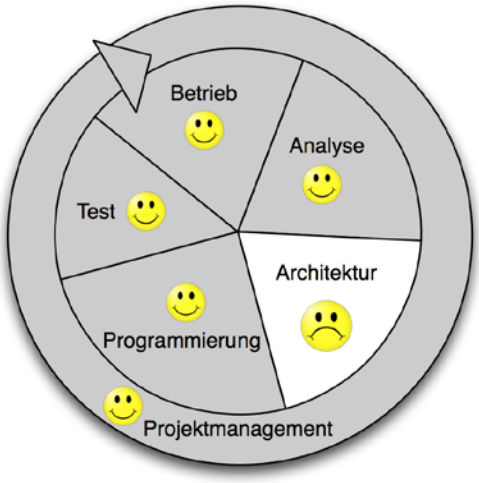


BILD 13.1
Die Rolle und der Ausbildungsgang von Softwarearchitekten sind unterspezifiziert.

13.1 Standardisierte Lehrpläne für Softwarearchitekten

Entstanden aus Hochschule und Praxis

Als Grundlage für die Ausbildung von Softwarearchitekten hat der iSAQB in Zusammenarbeit mit Hochschulen und Praktikern umfassende Lehrpläne entwickelt, die das Berufsbild von Softwarearchitekten beschreiben. Seit 2009 liegt ein solcher für die Grundlagenausbildung (*foundation level*) vor und wurde Ende 2013 deutlich überarbeitet. Ebenfalls seit 2013 gibt es eine Reihe von Vertiefungen für Fortgeschrittene (*advanced level*).

13.1.1 Grundlagenausbildung und Zertifizierung *Foundation-Level*

Insbesondere vom Foundation-Level-Lehrplan profitieren sowohl Softwarearchitekten wie auch Unternehmen, da er die eingangs geschilderten Missverständnisse bezüglich Rolle und Aufgabe von Softwarearchitekten *unabhängig* klärt (Wer darf sich als Softwarearchitekt bezeichnen? Was leisten Softwarearchitekten für Projekte? Welche Aufgaben sollten sie wahrnehmen?).

Auf Basis dieses Foundation-Level-Lehrplans können Softwarearchitekten die Prüfung und Zertifizierung zum *Certified Professional for Software-Architecture, Foundation Level* (CPSA-F) ablegen.

Gemäß dem Foundation-Level-Lehrplan sollen Softwarearchitekten folgende Inhalte vermittelt werden:

- Begriff und Bedeutung von Softwarearchitektur,
- Aufgaben und Verantwortung von Softwarearchitekten,
- Rolle von Softwarearchitekten in Projekten sowie
- Methoden und Techniken zur Entwicklung von Softwarearchitekturen.

Teilnehmer sollen die Fähigkeit erlernen, mit anderen Projektbeteiligten aus den Bereichen Anforderungs- und Projektmanagement, Entwicklung und Qualitätssicherung wesentliche Softwarearchitektur-Entscheidungen abzustimmen und Softwarearchitekturen auf der Basis von Sichten, Architekturmustern und technischen Konzepten zu dokumentieren und zu kommunizieren. Sie sollen die wesentlichen Schritte beim Entwurf von Softwarearchitekturen verstehen und nach entsprechender Ausbildung für kleine und mittlere Systeme selbstständig durchführen können.

Der standardisierte Lehrplan definiert die Rolle von Softwarearchitekten und präzisiert deren Aufgaben und Verantwortung in IT-Projekten. Damit hilft er allen Projektbeteiligten bei der Klärung der gegenseitigen Erwartungshaltung.

13.1.2 Fortgeschrittene Aus- und Weiterbildung (*Advanced-Level*)

Fortgeschrittene Architekten, die Verantwortung für mittlere bis große Systeme übernehmen können, haben je nach Branche, Technologie oder System sehr unterschiedliche Schwerpunkte. Der iSAQB e. V. trägt dieser Vielfalt Rechnung, indem die Ausbildung zu Advanced-Level-Architekten stark individualisierbar angelegt ist. Sie können aus einer Vielzahl von Ausbildungsmodulen die jeweils thematisch passenden zusammenstellen, um das spezifisch benötigte Kompetenzprofil aufzubauen.

Der iSAQB e. V. fordert jedoch, dass Sie dabei folgende drei Kompetenzbereiche abdecken:

- Methodische Kompetenzen: Systematisches Vorgehen bei Architekturaufgaben, unabhängig von Technologien
- Technische Kompetenz: Kenntnis und Anwendung von Technologien zur Lösung von Entwurfsaufgaben
- Kommunikative Kompetenz: Fähigkeiten zur produktiven Zusammenarbeit mit unterschiedlichen Stakeholdern, Kommunikation, Präsentation, Argumentation, Moderation

Vor einer Advanced-Level-Zertifizierung müssen Sie in allen diesen Kompetenzbereichen Aus- oder Weiterbildung im Form sogenannter *Credit-Points* nachweisen. Bereits früher absolvierte Schulungen oder Zertifizierungen können Sie sich auf Antrag beim iSAQB e. V. ganz oder teilweise anerkennen lassen.

Weitere Voraussetzungen[3] für eine *Advanced-Level*-Zertifizierung ist die erfolgreiche CPSA-F Prüfung, mindestens dreijährige Berufserfahrung an mindestens zwei unterschiedlichen IT-Systemen sowie die erfolgreiche Bearbeitung einer umfangreichen Hausarbeit.

[3] Die exakte Liste der Voraussetzungen finden Sie unter [isaqb].

13.2 Können, Wissen und Verstehen

Der iSAQB-Lehrplan differenziert zwischen drei Ebenen von Fähigkeiten:

1. *können:* Diese Aufgaben oder Tätigkeiten müssen Softwarearchitekten selbstständig ausführen können – situativ abhängig von ihrer Praxiserfahrung und den konkreten Projektanforderungen entweder komplett eigenverantwortlich oder auch unterstützt durch weitere Personen. Diese Teile des Lehrplans sind für die iSAQB-Zertifizierung relevant und somit Bestandteil entsprechender Prüfungen. In den Lernzielen mit R1 gekennzeichnet.
2. *verstehen:* Diese Lernziele können Bestandteil von Zertifizierungsprüfungen sein; Softwarearchitekten müssen die Aufgaben oder Tätigkeiten nicht unbedingt selbst ausführen können. Diese Teile des Lehrplans können geprüft werden. In den Lernzielen mit R2 gekennzeichnet.
3. *kennen (wissen):* Begriffe, Konzepte, Methoden oder Praktiken können das Verständnis für Architekturen oder Architekturentscheidungen unterstützen. Diese Inhalte können in Schulungen bei Bedarf unterrichtet werden, gehören aber in keinem Fall zu Prüfungen des *Foundation Levels*. In den Lernzielen mit R3 gekennzeichnet.

Für die *Können*-Inhalte fordert der iSAQB (ganz in meinem Sinne) ausführliche Übungen in entsprechenden Schulungen. Angehende Softwarearchitekten sollten die Aufgaben selbstständig planen und ausführen können und anschließend in der Lage sein, die erarbeiteten Ergebnisse objektiv zu bewerten.

13.3 Voraussetzungen und Abgrenzungen

Voraussetzung: praktische Erfahrung

Entsprechend der oben genannten Zielsetzung setzt der iSAQB-Lehrplan *Erfahrung* in der Softwareentwicklung voraus. Insbesondere gehören folgende Inhalte nicht zum Lehrplan (wohl aber zum notwendigen Können von Softwarearchitekten!):

- Mehr als 18 Monate praktische Erfahrung in der Softwareentwicklung, erworben anhand unterschiedlicher Projekte oder Systeme
- Kenntnisse und praktische Erfahrung in mindestens einer höheren Programmiersprache
- Grundlagen von Modellierung und Abstraktion
- Grundlagen von UML (Klassen-, Paket-, Komponenten- und Sequenzdiagramme) und deren Abbildung auf Quellcode
- Praktische Erfahrung in technischer Dokumentation, insbesondere der Dokumentation von Quellcode, von Systementwürfen oder technischen Konzepten.

Da der iSAQB primär auf methodische Fähigkeiten und Wissen fokussiert, gehören konkrete Implementierungstechnologien oder spezielle Werkzeuge explizit *nicht* zum standardisierten Lerninhalt, ebenso wenig wie UML, Systemanalyse oder Projektmanagement.

13.4 Struktur des iSAQB-Foundation-Level-Lehrplans

Bild 13.2 zeigt die Struktur des iSAQB-Lehrplans (Foundation Level) mit ihrer vorgesehenen zeitlichen Aufteilung.

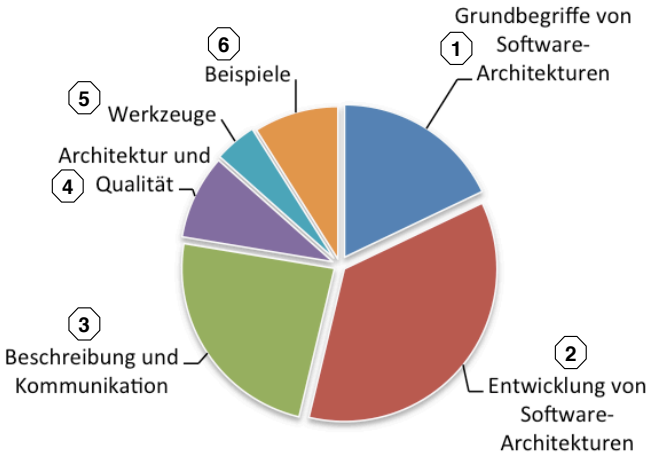


BILD 13.2 Struktur und zeitliche Aufteilung des iSAQB-Lehrplans für Softwarearchitekten

In den nachfolgenden Abschnitten stelle ich Ihnen die Lernziele dieser Kapitel vor und gebe Ihnen jeweils Hinweise, wo Sie im Buch Erklärungen dazu finden.

In den Lernzielen gebe ich zu Ihrer Hilfe noch die Prüfungsrelevanz mit an (R1: wird geprüft, R2: kann geprüft werden, R3: wird nicht geprüft).

I. Grundbegriffe von Softwarearchitekturen

LZ-1-1	Definitionen von Softwarearchitektur: Gemeinsamkeiten benennen (Bausteine, Komponenten, Schnittstellen, Beziehungen, Strukturen, Konzepte, Prinzipien)	R1	Abschnitt 2.1
LZ-1-2	Nutzen und Ziele von Softwarearchitektur: Fokus auf Langlebigkeit und Qualitätszielen	R1	Abschnitt 2.1.1
LZ-1-3	Softwarearchitektur im Software-Lebenszyklus	R2	Abschnitte 2.3, 2.4
LZ-1-4	Aufgaben von Softwarearchitekten	R1	Abschnitt 2.2, Kapitel 3
LZ-1-5	Beziehung zu anderen Stakeholdern	R1	Abschnitt 2.4
LZ-1-6	Zusammenhang zu Vorgehen bei Entwicklung	R1	Kapitel 3
LZ-1-7	Architektur- und Projektziele	R1	Abschnitt 2.1.1
LZ-1-8	Explizite und implizite Aussagen	R1	Abschnitt 2.2
LZ-1-9	Zusammenhang zu Unternehmensarchitektur	R3	Kapitel 10
LZ-1-10	Typen softwareintensiver Systeme: Informationssysteme, eingebettete oder Echtzeitsysteme, Decision-Support-Systeme, mobile oder Web-Systeme.	R2	Kapitel 3.2.2

R1: wird geprüft, R2: kann geprüft werden, R3: wird nicht geprüft

II. Entwicklung von Softwarearchitekturen

In diesem Abschnitt verwenden Sie die Begriffe und Konzepte der voranstehenden Teile, um in Übungen selbstständig Architekturentwürfe zu erstellen und zu kommunizieren.

LZ-2-1	Vorgehen zur Architekturentwicklung	R1	Kapitel 3
LZ-2-2	Architekturen entwerfen	R1	Kapitel 3
LZ-2-3	Einflussfaktoren erheben und einordnen	R1	Abschnitte 3.1, 3.2, 3.3
LZ-2-4	Technische Konzepte auswählen und erarbeiten	R1	Abschnitt 3.6, Kapitel 7
LZ-2-5	Wichtige Architekturstile und -muster	R1-3	Abschnitt 6.2
LZ-2-6	Entwurfsprinzipien erläutern und anwenden	R1	Abschnitt 6.3
LZ-2-7	Abhängigkeiten und Kopplung von Bausteinen	R1	Abschnitte 6.4, 6.1.2
LZ-2-8	Bausteine/Strukturelemente entwerfen	R1	Kapitel 6
LZ-2-9	Schnittstellen entwerfen	R1	Abschnitte 6.3.3
LZ-2-10	Entwurfsmuster verstehen und anwenden	R2	Abschnitt 6.5

R1: wird geprüft, R2: kann geprüft werden, R3: wird nicht geprüft

Weitere Quellen dazu: [Buschmann+07], [Eilebrecht+13], [Evans04], [Fowler02], [Gamma95].

III. Beschreibung und Kommunikation von Softwarearchitekturen

Dieser Abschnitt führt einige Begriffe und Konzepte ein, die Sie zur Entwicklung von Softwarearchitekturen benötigen. Er hängt daher sehr eng mit dem vorhergehenden Kapitel zusammen, das Entwurf und Entwicklung von Architekturen behandelt.

LZ-3-1	Qualitätsmerkmale technischer Dokumentation	R1	Abschnitte 4.2.1, 4.2.2
LZ-3-2	Softwarearchitekturen stakeholdergerecht beschreiben und kommunizieren	R1	Kapitel 4
LZ-3-3	Notations-/Modellierungsmittel zur Beschreibung von Softwarearchitekturen	R2	Abschnitte 4.3, Kapitel 5
LZ-3-4	Architektursichten erläutern und anwenden	R1	Abschnitte 4.4 – 4.8, Kapitel 5
LZ-3-5	Kontextabgrenzung erläutern und anwenden	R1	Abschnitt 4.5
LZ-3-6	Querschnittliche Architekturkonzepte erläutern und anwenden	R1	Abschnitt 4.10, ergänzend: Kapitel 7
LZ-3-7	Schnittstellen beschreiben	R1	Abschnitt 4.9
LZ-3-8	Architekturentscheidungen erläutern und dokumentieren	R2	Kapitel 3, Kapitel 4
LZ-3-9	Dokumentation als schriftliche Kommunikation	R2	Kapitel 4
LZ-3-10	Weitere Hilfsmittel und Werkzeuge zur Dokumentation	R3	Abschnitt 4.11

R1: wird geprüft, R2: kann geprüft werden, R3: wird nicht geprüft

Quellen dazu: Kapitel 4 und 5 sowie 11 (für Beispiele), [arc42], [Clements+03], [Hargis+04], [Zörner-12].

IV. Softwarearchitekturen und Qualität

Das zentrale Thema dieses Kapitels ist Qualität: Qualitätsanforderungen, ausgedrückt durch hierarchische Qualitätsmodelle (beispielsweise DIN/ISO 9126 oder DIN/ISO 25010), ausgedrückt durch Qualitätsmerkmale, Szenarien und Qualitätsbäume. Da sich Qualitätsmerkmale gegenseitig beeinflussen, müssen Sie bei Entwurf und Entwicklung häufig Kompromisse bei der Umsetzung von Qualitätsmerkmalen eingehen. Qualitative Architekturbewertung von Risiken hinsichtlich der Erreichung von Qualitätsmerkmalen bildet einen weiteren Schwerpunkt.

LZ-4-1	Qualitätsmodelle und Qualitätsmerkmale	R1	Abschnitte 3.2.3, 8.1
LZ-4-2	Qualitätsanforderungen an Software und -Architekturen	R1	Abschnitt 3.2.3, Abschnitt 8.1
LZ-4-3	Softwarearchitekturen qualitativ bewerten	R2	Abschnitt 8.1
LZ-4-4	Softwarearchitekturen quantitativ bewerten	R2	Abschnitt 8.2
LZ-4-5	Qualitätsziele und -anforderungen erreichen	R2	Abschnitt 3.6, (Kapitel 7)

R1: wird geprüft, R2: kann geprüft werden, R3: wird nicht geprüft

V. Werkzeuge für Softwarearchitekten

In diesem Teil lernen Sie einige Kategorien möglicher Werkzeuge kennen, die Softwarearchitekten bei der Erfüllung ihrer Aufgaben nützen können. Sie erarbeiten dazu (mögliche) Entscheidungskriterien für die Auswahl dieser Werkzeuge.

LZ-5-1	Wichtige Werkzeugkategorien benennen	R1	Kapitel 12, Abschnitt 4.11
LZ-5-2	Werkzeuge bedarfsgerecht auswählen	R2	Kapitel 12

R1: wird geprüft, R2: kann geprüft werden, R3: wird nicht geprüft

VI. Beispiele von Software-Achitekturen

Sie lernen hier praktische Beispielarchitekturen sowie den Bezug von Anforderungen zur Lösung kennen sowie, (zumindest ansatzweise) die technische Umsetzung der Beispiele. Diese Inhalte sind nicht prüfungsrelevant.

LZ-6-1	Bezug von Anforderungen zu Lösung	R3	Kapitel 11
LZ-6-2	Technische Umsetzung einer Lösung nachvollziehen	R3	Kapitel 11

R1: wird geprüft, R2: kann geprüft werden, R3: wird nicht geprüft

Zwei Beispiele praktischer Softwarearchitekturen finden Sie in Kapitel 11.

13.5 Zertifizierung nach dem iSAQB-Lehrplan

Sie haben die Möglichkeit, sich bei verschiedenen unabhängigen Anbietern[4] durch eine Prüfung gemäß dem iSAQB-Lehrplan zertifizieren zu lassen. Die Prüfungsanbieter müssen dabei standardisierte, vom iSAQB erarbeitete Prüfungsfragen verwenden – die sich strikt am CPSA-F-Lehrplan orientieren.

Die Prüfung selbst findet nach einem Multiple-Choice-Verfahren statt. Das Prüfungsergebnis ist somit objektiv und frei von Interpretation und Ermessen.

Ich bin persönlich der Meinung, dass Sie mit einer solchen Prüfung zwar *Wissen* nachweisen, Ihre Qualifikation als Softwarearchitekt jedoch nur sehr bedingt unter Beweis stellen können. Das wird Ihnen erst durch (erfolgreiche) praktische Arbeit an konkreten Architekturen gelingen.

[4] Stand Dezember 2013 führen ISQI (*www.isqi.org*) sowie Future Network e. V. (*www.future-network.at*) Zertifizierungsprüfungen durch.

14 Nachwort: Architektonien

In diesem Kapitel finden Sie eine überarbeitete Fassung meines Artikels „Von Analytistan nach Architektonien“, im Original erschienen in der Zeitschrift iX Ausgabe 8/2006. Veröffentlichung mit freundlicher Genehmigung des Heise Zeitschriftenverlages.

In Kürze

Das kleine Architektonien grenzt unmittelbar ans bekannte Analytistan mit seiner prächtigen Hauptstadt Anforderungshausen. Zwischen beiden Lagern herrscht ... ja, wie soll man es nur ausdrücken ... eine Art gespannter Zustand: Während in Anforderungshausen essenzielle Wünsche in den Köpfen der Bewohner wabern, müssen die Architektonier die wirkliche Implementierung vorbereiten und überwachen. Sie stehen unter dem Einfluss herrschsüchtiger Managier und penibler Qualitessen und müssen gleichzeitig die kritischen Stimmen der zahlreichen Prograländer aushalten.

Sich in diesem Spannungsfeld langfristig zu behaupten, erfordert sowohl Vielseitigkeit als auch Kunstfertigkeit. In den langen Jahren ihrer kargen Existenz haben die Architektonier dafür ihr architektonisches Manifest geschaffen, aus dem ich Ihnen einige Auszüge vorstellen möchte.

Folgen Sie mir vorab auf eine kurze Reise durch die IT-Welt, auf der ich Ihnen die wesentlichen Stationen vorstellen möchte.

14.1 In sechs Stationen um die (IT-)Welt

Wie Sie sicher wissen, gehört die IT-Welt ganz und gar den Kundeniern – einer größtenteils unerforschten, unberechenbaren und ziemlich vielseitigen Gattung *homo dubiosis* mit einem Einschlag von *homo semisapiens*. Aus unerfindlichen Gründen beschränken die Kundenier den direkten Kontakt mit der schönen IT-Welt auf drei Länder, nämlich Analytistan, Betriebien und Kostenia. Und werden einmal Bewohner der übrigen IT-Länder nach Kostenia eingeladen (oder entführt!), geschieht das entweder, um sie dort ins Gefängnis zu werfen oder ihnen eine andere Staatsbürgerschaft zu verleihen.

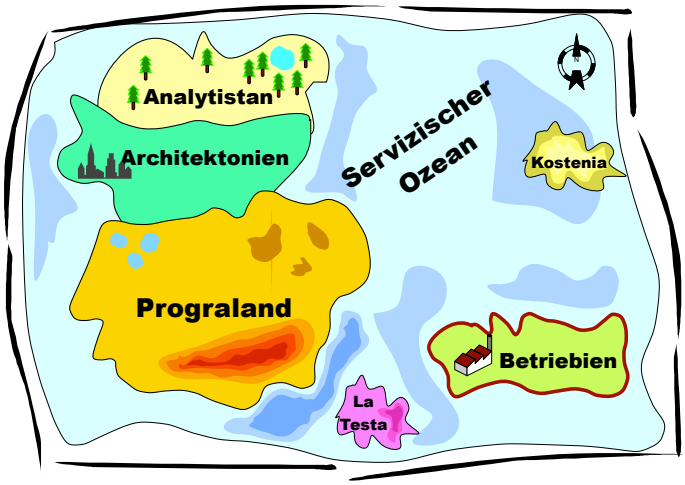


BILD 14.1
Landkarte der IT

Analytistan

Dies ist die Heimat der liebenswürdigen und stets freundlichen Analytistaner, auch genannt Requies (gesprochen: Rekkies). Diese Spezies spricht die gleiche Sprache wie die bereits erwähnten Kundenier. Requies verfügen über kommunikative und hypnotische Fähigkeiten, mit denen sie sogar unbewusste Wünsche der Kundenier erkennen können. Analytistaner fürchten sich vor nichts mehr als vor Entscheidungen – niemals würden sie eigenständig eine Auswahl zwischen möglichen Alternativen treffen. Stets befragen sie in solchen Fällen die ihnen wohlgesonnenen Kundeniern (die allerdings, seien wir ehrlich, häufig die Antwort schuldig bleiben).

Requies gehen zu Beginn von IT-Projekten häufig bei den betroffenen Kundeniern ein und aus. Dies führt jedoch auf beiden Seiten zu gewissen Abnutzungserscheinungen, sodass die intensive Kommunikation oftmals bereits nach kurzer Zeit wieder eingestellt wird.

Requies lieben das geschriebene Wort über alles und pflegen das Landesmotto *„Papier ist geduldig“*. Wo ihre Nachbarn aus Architektonien oder die technophilen Programmisten zu präzisen Werkzeugen wie Java oder Ruby greifen, bleiben Requies grundsätzlich bei redundanzbehafteter, ungenauer, aber dafür ästhetischer, natürlicher Sprache.

Bemerkenswert an den Requies ist auch ihre fast schon legendäre Unbeliebtheit: Außer bei Kundeniern dürfen sie sich praktisch nirgendwo auf dem IT-Globus blicken lassen – obwohl viele von ihnen ziemlich nett sind, ehrlich!

Architektonien

Architektonier lieben die Dunkelheit – zumindest müssen sie dort arbeiten. Und falls es bei der Arbeit doch mal hell wird, herrscht garantiert dichter Nebel. Mit 99%iger Wahrscheinlichkeit hat ein Architektonier bei der Arbeit keine klare Sicht, weder auf die zu lösenden Aufgaben noch auf die vorhandene Technologie.

Macht aber nichts – Architektonier haben im Lauf ihrer fast 25-jährigen Evolution großen Mut entwickelt. Und den brauchen sie auch, nämlich um ihrer wichtigsten Aufgabe nachzukommen: Entscheidungen zu treffen. Ihre zweite Aufgabe lautet, über ihre Arbeit zum

Volk zu sprechen. Architektonier besitzen eine gehörige Portion Sendungsbewusstsein und Kommunikationstalent.

Zwischen Analytistan und Architektonien verläuft eine Grenze, die der ehemaligen innerdeutschen Grenze ähnlich ist: Architektonier können sie dank ihres legendären Mutes problemlos in beide Richtungen passieren, für Requies bleibt sie jedoch in beide Richtungen versperrt.

Prograland

Die Heimat von Geeks, Nerds und weiteren technophilen Vielseitern. Alle Neugeborenen werden in Prograland traditionsgemäß in lauwarme Bytesuppe gelegt. Das prägt, gründlich und dauerhaft. Prograland straft sämtliche biologischen Thesen Lügen: Trotz 98,8 % maskuliner Bevölkerung nimmt die Einwohnerzahl von Prograland ständig zu. Irgendwie muss eine Art Raum-Zeit-Kanal frische Geeks in dieses vielseitige, farbenfrohe und wortkarge Land einschleusen. Aber das soll hier keine Rolle spielen.

Prograland besteht aus einer Vielzahl von Provinzen, zwischen denen teilweise offene Konflikte herrschen. So befeinden sich seit Jahren die Hochebenen von *IX mit den Niederungen von MS*, oder C* neidet J* seine Erfolge.

Insgesamt sind die Prograländer ziemlich verspielt und bleiben gerne unter sich. Requies sind nicht geduldet, Geld und Zeit spielen keine Rolle (es sei denn, es handelt sich um Millisekunden – die zählen. Personentage aber nicht).

Obwohl ich erklärter Sympathisant von Prograländern bin – sie sind (und bleiben) die Urheber vieler dramatischer IT-Probleme. Wer sonst schreibt `#define private public` in die zentrale Header-Datei und umgeht damit das Geheimnisprinzip? Wer sonst produziert Stack-Overflows, Memory-Leaks und Buffer-Overflows (die dann andere Geeks wiederum für gute oder böse Zwecke ausnutzen können)? Aber eigentlich sind sie sehr nett und umgänglich.

La Testa

Von Prograland durch ein wildes Meer und gewaltige Untiefen getrennt liegt die kleine und wilde Insel La Testa. Weitgehend unberührt und unerforscht, ist sie die Heimat der Testanier (auch „Q“ genannt), vor denen sich die Prograländer sehr fürchten. Das sorgt für gute nachbarschaftliche Beziehungen. Kein echter Testanier begibt sich freiwillig nach Prograland. Weder Geld noch gutes Zureden können einen echten Testanier dazu bewegen, seine elaborierten Worte in den, wie die Q ihn nennen, *technokratischen Sumpf* zu expedieren. Manchen Architektoniern sagt man nach, sie hätten erfolgreich zwischen Q und Geeks vermittelt – aber das können genauso gut Gerüchte sein.

In den letzten Jahren bahnt sich in dieser brisanten Situation übrigens Besserung an – nachdem ein Eidgenosse gemeinsam mit einem eXtremen ein paar Zeilen ultracoolen Java-Code geschrieben und unter der Bezeichnung „JUnit“ veröffentlicht hat.

Architektonier und Kundenier sind auf La Testa gern gesehene Gäste. Angeblich sollen gerade Architektonier auf La Testa einer Extremsportart frönen: dem Bug-Fishing. Aber auch das könnte ein Gerücht sein.

Betriebien

Betriebien ist das einzige komplett industrialisierte Land in der IT-Welt. Es ist umgeben von einem Bauwerk beeindruckender Größe – der betriebischen Mauer. Einem Gerücht nach soll es das einzige Bauwerk sein, das aus dem Weltall mit bloßem Auge sichtbar ist. Falls jemand ins Weltall kommt, sollte er dieses Gerücht in der Blogosphäre bestätigen. Ansonsten bekommt kaum ein Einwohner der übrigen IT-Welt jemals die vermeintliche Schönheit von Betriebien zu Gesicht.

Gelegentlich hört oder liest man, dass Betriebien DAS Land der IT-Welt sei, in dem „Prozesse" wirklich funktionieren. Übrigens das einzige, garantiert!

Kostenia

Über diese Insel mag ich hier nichts weiter schreiben. Es ist die Heimat der Managissimos – der Regenten der IT-Welt. Sie sorgen für Geld und Zeit in IT-Projekten. Wenn sie genug daran gedacht haben, sinnieren sie über Ressourcen. Und ab und zu organisieren sie auch die Zusammenarbeit von Bewohnern der verschiedenen Länder in der IT-Welt.

14.2 Ratschläge aus dem architektonischen Manifest

Bis hierhin haben Sie einige Details über die seltsame IT-Welt gelesen. Im Folgenden möchte ich Ihnen einige Teile des architektonischen Manifests erläutern. Sie sollen Ihnen als Tipps für erfolgreiche Reisen durch dieses Habitat dienen. Die Ratschläge dieses Manifests richten sich an (Software- und IT-)Architekten, können in vielen Fällen aber auch für andere Mitspieler in IT-Projekten positive Wirkung zeigen. Unter anderem verbessert das Verständnis dieser architektonischen Erfolgsmuster die Völkerverständigung in IT-Land – was die Chance auf erfolgreiche Projekte deutlich verbessern hilft.

Handeln Sie aktiv!

Es gibt nichts geschenkt, und niemand (keiner, nirgendwer, nobody) wird Ihnen freiwillig und ohne Aufforderung Geld, Mitarbeiter, Kompetenz oder Mitspracherechte geben, wenn Sie nicht AKTIV danach fragen. Als Architekt müssen Sie entscheiden, mitreden, vermarkten, argumentieren und möglicherweise investieren – und dazu benötigen Sie Ressourcen und Kompetenzen. Handeln Sie aktiv – und holen Sie sich alles, was Sie für Ihre produktive Architekturarbeit benötigen! Warten Sie nicht darauf, dass Ihnen der Weihnachtsmann oder das Christkind oder der Osterhase solche Dinge schenkt – das passiert nur im Märchen (und die IT-Welt ist ziemlich märchenfrei ...)

Arbeiten Sie iterativ

Iterativ vorgehen bedeutet, Ergebnisse regelmäßig zu bewerten, um schrittweise Verbesserungen einpflegen zu können.

Alle Bewohner der IT-Welt profitieren von Feedback, Rückmeldungen und Ratschlägen. Diskutieren Sie mit Ihren KollegInnen über Ihre Arbeitsergebnisse, Ihre Entwürfe und Strukturvorschläge. Dieser Ratschlag gilt ganz besonders für Architekturentwürfe und deren Dokumentation (obwohl auch alle anderen Arten von Artefakten von iterativer Verbesserung profitieren!).

Durch iteratives Vorgehen können Sie die Qualität Ihrer Arbeitsergebnisse deutlich steigern und gleichzeitig das Risiko von Fehlern oder Unzulänglichkeiten drastisch senken. Diesem Argument öffnen sich auch hartgesottene und kostenzentrierte Managissimos!

Entwickeln Sie eine Null-Rhesus-Negativ-Mentalität

Die Blutgruppe „Null-Rhesus-Negativ“ ist mit sämtlichen anderen Blutgruppen kompatibel. Architektonier müssen mit sämtlichen Beteiligten von IT-Projekten „*kommunikationskompatibel*“ sein. Diese Eigenschaft ermöglicht Ihnen, mit allen anderen Stakeholdern konstruktiv über architekturrelevante Sachverhalte zu kommunizieren.

Sie werden jetzt vielleicht einwenden, dass doch *Managissimos* diese Kommunikation mit anderen Stakeholdern erledigen. Aber das zweifle ich stark an, denn Managissimos können auf *technischem* Niveau oftmals nur unzureichend argumentieren, weil sie beispielsweise der Sprachen von Prograland oder Betriebien kaum mächtig sind.

Üben Sie sich also in Kommunikation. Lernen und üben Sie zu präsentieren, zu argumentieren, zu diskutieren. Trainieren Sie Ihre Kommunikationsfähigkeiten – das steigert neben Ihrer Beliebtheit in Projekten ganz eindeutig Ihre Erfolgsaussichten.

Seien Sie mutig

In Analytistan scheint immer die Sonne, und alle Einwohner sind gut gelaunt. Architektonier hingegen leben die meiste Zeit des Jahres im Nebel und arbeiten praktisch grundsätzlich im Dunkeln.

Requies schreiben auf, was Kundenier ihnen sagen (oder was die Analytiker glauben, was die Kunden gemeint haben könnten). Architektonier müssen (bequeme oder unbequeme) Entscheidungen treffen, die den Erfolg von IT-Projekten stark beeinflussen können. Mut hilft Ihnen dabei als Charaktereigenschaft in vielen Situationen Ihres Architektenlebens weiter. Ich möchte die Empfehlung zu Mut hier in mehrere Richtungen konkretisieren:

- Mut zu suboptimalen Entscheidungen
- Mut zur Lücke
- Mut zum Widerspruch

Als Software- oder IT-Architekt befinden Sie sich ständig in Entscheidungssituationen: Neben Strukturentscheidungen gilt es vielerlei technisch orientierte Aspekte zu entscheiden, die über die Grenzen einzelner Systembausteine hinweg Auswirkungen haben. Ihre Entscheidungen haben also prägenden Einfluss auf das System und seine weitere Entwicklung. Sie beeinflussen das Team, den Entwicklungsprozess und möglicherweise sogar die Anforderungen der Kunden.

Nach Abschluss des Projektes werden Sie auf eine Vielzahl von Entscheidungen zurückblicken und jetzt ganz genau erkennen, welche davon besonders gut und welche suboptimal waren. Zu jedem früheren Zeitpunkt (noch einmal: zu *jedem* früheren Zeitpunkt) bleibt Ihnen diese Bewertung verwehrt – es könnten sich ja noch wichtige Einflussfaktoren, Randbedingungen oder Anforderungen ändern ...

Als Fazit bleibt: Sie können prinzipiell erst am Ende wissen, ob eine bestimmte Entscheidung gut oder schlecht war.

Darum brauchen Sie den Mut, suboptimale, aber notwendige Entscheidungen auch in unklaren Situationen zu treffen. Das Streben nach Perfektion kostet immens viel Zeit und Aufwand – meistens genügen 80 %-Lösungen vollauf.

Structure follows Constraints

Überall lesen Sie den für Designer und Architekten angeblich so wichtigen Leitsatz *Form follows Function*. „Die Form, die Gestaltung von Dingen soll sich aus ihrer Funktion ableiten" (zitiert nach [wikipedia]). Das mag für Gebrauchsgegenstände und Gebäude eine recht nützliche Herangehensweise sein – für Architektonier jedoch ist dieser Ratschlag, entschuldigen Sie meine überdeutliche Ausdrucksweise, kompletter Unfug. Für sie geht es nämlich primär um Strukturen, Konzepte und Qualitäten – Funktionen werden oftmals sogar zu Nebensächlichkeiten. Der Grund dafür liegt darin, dass die Entscheidungsfreiheit der Architektonier primär von Einschränkungen und Randbedingungen beeinflusst wird, also von leidigen Aspekten wie beispielsweise Performance, Sicherheit, Wartbarkeit, Implementierbarkeit oder Verständlichkeit.

Architekturen sind Strukturen von Bausteinen (und deren Schnittstellen und Interaktionen), zusammen mit übergreifenden Konzepten. In erster Linie prägen Qualitätsanforderungen diese Strukturen und Konzepte, erst in zweiter Linie funktionale Anforderungen.

Schauen Sie als Architekt grundsätzlich zuerst auf die Randbedingungen und Qualitätsanforderungen, denn damit haben Sie in den meisten Fällen schon die wesentlichen Probleme des konkreten Systems im Fokus.

Akzeptieren Sie unvollständige Anforderungen

Streben nach vollständigen und korrekten Anforderungen führt entweder zu Halluzinationen oder ins Paradies. Diese Möglichkeiten sind sehr ungleich verteilt.

Akzeptieren Sie daher ohne Ärger oder Angst unvollständige Anforderungen. Als Architektonier müssen Sie sowieso Requies-Arbeit leisten, ob Sie wollen oder nicht. Architekten müssen Einflussfaktoren und Randbedingungen klären – egal, ob die Analytiker sehr gute oder gar keine Vorarbeit für Sie geleistet haben. Sie müssen zu den Anforderungen passende Architekturentscheidungen treffen und dafür in den Anforderungen mögliche Widersprüche oder Lücken erkennen, hinterfragen und sie bei Bedarf im Sinne der Kundenier interpretieren. Hierfür benötigen Sie Mut.

Akzeptieren Sie, dass Ihre Kundenier Ihnen ungenaue Anforderungen stellen. Sie meinen es nicht böse. Akzeptieren Sie insbesondere, dass Kundenier Ihnen ungenaue Qualitätsanforderungen stellen – auch das geschieht nicht mit böser Absicht, sondern, weil es sehr schwierig ist, Dinge wie Flexibilität, Wartbarkeit, Verständlichkeit oder Integrationsfähigkeit zu operationalisieren. Als Architektonier müssen Sie das operationalisieren, was die Requies eigentlich bereits hätten erledigen sollen.

Architektur beeinflusst Anforderungen

Bereits Fragmente von Architekturen beeinflussen in der Regel das Problem oder die Wünsche der Kunden. Der Einfluss ist umso größer, je weniger die Architektur darauf vorbereitet ist.

Sobald ein Kundenier einen Teil eines laufenden Systems zu sehen oder zu spüren bekommt, wird sich seine Meinung zu seinen Anforderungen ändern – sie wird konkreter, erhält andere Prioritäten oder wird um bisher ungekannte Facetten erweitert. Kent Beck (ein sehr berühmter Bürger von Prograland, der zeitweise zum Volksheld in La Testa wurde, dafür aber in Kostenia Einreiseverbot erhielt ...) untertitelte eines seiner lesenswerten Bücher mit „Embrace Change".

Erfragen Sie aktiv von Ihren Kundeniern aktuelle Anforderungen. Zeigen Sie ihnen regelmäßig nach Iterationsende den Stand der laufenden Software, und hören Sie sorgfältig auf die Rückmeldung. Das ist eine entscheidende Feedbackschleife für erfolgreiche Projekte!

Prüfen Sie Anforderungen

Architektonien kontrolliert den Zugang von Waren und Dienstleistungen ins Land sehr streng, insbesondere solche aus Analytistan. Keine virtuelle oder reale Entität darf ohne Review die Grenze passieren.

Und auf keinen Fall dürfen Anforderungen nach Architektonien gelangen, die nicht vor dem Grenzübertritt ein domänenkundiger Architektonier gelesen, verstanden und für gut genug befunden hat.

Manchmal kommen Kundenier auf Ideen, die für Requies wie auch andere Kundenier sehr verlockend klingen, so als müsse die neue Software diese Anforderungen unbedingt erfüllen. Kundige Architektonier sehen solchen Ideen an, dass der Preis für ihre Erfüllung unangemessen hoch liegt – und können die Kundenier darauf hinweisen. (Requies können das nicht, weil sie die Architektur nicht kennen.) Dazu schreiben die Requie-Gurus Suzanne und James Robertson: „Anforderungen, die Sie sich nicht leisten können, sind keine Anforderungen."

Architektonier sollen also aus ihrer – meist technischen – Sicht die Angemessenheit von Anforderungen prüfen. Falls gewisse Wünsche der Kundenier auffällig schwer zu realisieren sind, extrem lange dauern oder wunderähnlicher Anstrengungen bedürfen, müssen Architektonier sich zu Wort melden: Zuerst argumentieren sie eindringlich und für Kundenier verständlich die Konsequenzen solcher Anforderungen oder Ideen. Sollten die Kundenier auf ihrer Idee beharren, so rechnen Architektonier die Kosten aus – Hilfe leisten hierbei die Managissioms auf Kostenia – und dokumentieren die Entscheidung der Kundenier aufs Genaueste.

Ach ja – wahrscheinlich benötigen Sie all Ihren Mut, um sich der geballten Macht von Kundeniern und Requies in den Weg zu stellen. Und möglicherweise werden diese es Ihnen nicht einmal danken. Als Architektonier können Sie allerdings deutlich ruhiger schlafen, wenn Sie (mal wieder) Ihre Kundenier oder Ihre Projekte vor unangemessenen Anforderungen bewahrt haben.

Ihre Weiterreise

In diesem Nachwort durfte ich Ihnen ansatzweise meine Sicht der IT-Welt präsentieren. Setzen Sie diese Reise nun auf eigene Faust fort. Erinnern Sie sich der Ratschläge des architektonischen Manifestes, falls Sie mal wieder nach Architektonien kommen (oder Bewohner dieses interessanten Landes auf Ihrer Reise treffen). Auf einigen meiner Reisen durfte ich

Manifeste oder Traditionen der übrigen Länder kennen (und schätzen) lernen, und die von mir hoch geschätzten Kollegen [Coplien] haben einen fantastischen Reiseführer geschaffen, in dem sogar das hier unerwähnt gebliebene, sagenumwobene Organisatorien eine Rolle spielen soll. Ich wünsche Ihnen viel Erfolg – und freue mich auf Ihre Reiseberichte!

15 Literatur

[Ambler-12] Scott Ambler, *www.agilemodeling.com*. Ratschläge für schlanke und agile Modellierung.

[arc42] *Hruschka, P., Starke, G.:* ARC42 – Ressourcen für Software-Architekten. *www.arc42.de* bzw. *arc42.org*

[AspectJ] *www.eclipse.org/aspectj*

[Avram+06] *Avram, A., Marinescu, F.:* Domain Driven Design Quickly. PDF-Version online: *http://infoq.com/books/domain-driven-design-quickly*

[Bartonitz10] *Bartonitz, M.:* BPM, GPM, BAM, BPMN, BPEL, XPDL, EABPM, CMPM. BPM-Netzwerk.de, online: *http://goo.gl/3YgSId*

[Bartonitz06] *Bartonitz, M.:* Wachsen die BPM- und Workflow-Lager zusammen? *http://www.bpm-netzwerk.de/articles/66*

[Bass+03] *Bass, L., Clements, P., Kazman, R.:* Software Architecture in Practice. 2. Auflage, Addison-Wesley, 2003.

[Binder2000] *Binder, R.:* Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, 2000.

[Bosch2000] *Bosch, J.:* Design & Use of Software Architectures. Addison-Wesley, 2000.

[Brewer00] Eric Brewer, CAP-Theorem: *http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf*

[Brooks95] *Brooks, F.:* The Mythical Man-Month, Anniversary Edition. Addison-Wesley, 1995.

[Buschmann+96] *Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.:* A System of Patterns. Pattern-Oriented Software Architecture. Wiley, 1996.

[Buschmann+07] *Buschmann, F., Henney, K., Schmidt, D.:* Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing. Volume 4 der POSA-Serie. Wiley, 2007.

[Clements+02] *Clements, P., Kazman, R., Klein, M.:* Evaluating Software Architectures – Methods and Case Studies. Addison-Wesley, 2002.

[Clements+10] *Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J. et al:* Documenting Software Architectures – Views and Beyond. 2. Auflage, Addison-Wesley, 2010.

[Cockburn05] Alistair Cockburn, Hexagonal Architecture. Online: *http://alistair.cockburn.us/Hexagonal+architecture* (abgerufen: Mai 2013)

[Coplien] *Coplien, J.:* Organizational Patterns. Prentice Hall, 2004.

[DeMarco98] *DeMarco, T.:* Der Termin – Ein Roman über Projektmanagement. Hanser, 1998.

[DeMarco+07] *DeMarco, T., Hruschka, P., Lister, T., McMenamin, St., Robertson, J., Robertson, S.:* Adrenalin Junkies & Formular Zombies. Hanser, 2007.

[Demeyer+02] *Demeyer, S., et. al.:* Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002.

[Dern09] *Dern, G.:* Management von IT-Architekturen. Informationssysteme im Fokus von Architekturplanung und -entwicklung. Vieweg, Edition CIO, 3. Auflage, 2009.

[Edlich+11] *Edlich, S., Friedland, A., Hampe, J., Brauer, B.:* NoSQL. Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. 2. Auflage, Hanser, 2011.

[Eilebrecht+13] *Eilebrecht, K., Starke, G.:* Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung. 4. Auflage, Springer, 2013. Siehe auch *http://patterns-kompakt.de*

[Evans04] *Evans, E.:* Domain Driven Design. Addison-Wesley, 2004. Zugehörige Website: *www.domaindrivendesign.org*

[Fowler99] *Fowler, M.:* Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[Fowler02] *Fowler, M.:* Patterns for Enterprise Application Architecture. Addison-Wesley, 2002.

[Freeman+10] *Freeman, S., Pryce, N.:* Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, 2010.

[Gamma95] *Gamma, E., Helm, R., Johnson, R., Vlissides, J.:* Design Patterns. Addison-Wesley, 1995.

[Gamma99] *Gamma, E., Beck, K.:* JUnit: A Cook's Tour. JavaReport, Mai 1999. Verfügbar im Internet: *http://www.junit.org*

[Gupta2003] *Gupta, S.:* Logging in Java with the JDK 1.4 Logging API and Apache log4j. Apress.

[Hargis+04] *Hargis, G., Caray, M., Hernandez, A.:* Technical Quality Technical Information: A Handbook for Writers and Editors. Prentice Hall, 2004.

[Hatley2000] *Hatley, D., Hruschka, P., Phirbai, I.:* Process for System Architecture and Requirements Engineering. Dorset House, 2000.

[Henderson96] *Henderson-Sellers, B.:* Object-Oriented Metrics: Measures of Complexity. Prentice Hall, UK, 1996.

[Hofmeister2000] *Hofmeister, C., Nord, R., Soni, D.:* Applied Software Architecture. Addison-Wesley, 2000.

[Hohpe+03] *Hohpe, G., Woolf, B.:* Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2003.

[Hunt+03] *Hunt, A., Thomas, D.:* Der Pragmatische Programmierer. Hanser, 2003.

[IEEE2000] IEEE Architecture Working Group: IEEE Recommended Practice for Architectural Description, Standard P1471, *www.pithecanthropus.com/~awg*

[ISAQB] International Software Architecture Qualification Board e. V., *www.isaqb.org*

[Keller2000] *Keller, W.:* The Bridge to the New Town – A Legacy System Migration Pattern. EuroPloP 2000. *http://www.objectarchitects.de/ObjectArchitects/papers/WhitePapers/ZippedPapers/pacman03.pdf*

[Keller96] *Keller, W., Coldewey, J.:* Relational Database Access Layers. PloP 1996. *http://www.objectarchitects.de/ObjectArchitects/papers/Published/ZippedPapers/plop_relzs05.pdf*

[Keller97] *Keller, W.:* Mapping Objects to Tables – A Pattern Language. Beitrag zur EuroPloP Konferenz 1997, *http://www.sdm.de/g/arcus*

[Keller98] *Keller, W.:* Object/Relational Access Layers – A Roadmap, Missing Links and More Patterns. EuroPloP 1998. *www.objectarchitects.de/ObjectArchitects/papers/Published/ZippedPapers/or06_proceedings.pdf*

[Keller02] *Keller, W.:* Enterprise Application Integration: Erfahrungen aus der Praxis. dpunkt, 2002.

[Keller12] *Keller, W.:* IT-Unternehmensarchitektur: Von der Geschäftsstrategie zur optimalen IT-Unterstützung. 2., erweiterte Auflage, dpunkt, 2012.

[Kruchten2001] *Kruchten, P.:* The Top Ten Misconceptions about Software Architecture. *www.therationaledge.com/content/apr_01/m_misconceptions_pk.html*.

[Kruchten95] *Kruchten, P.:* Architectural Blueprints – The „4+1" View Model of Architecture. IEEE Software November 1995; 12(6), p. 42–50.

[Kruchten+02] *Kruchten, P., et al.:* Software Architecture Review and Assessment (SARA) Report. Online verfügbar unter: *philippekruchten.com/architecture/SARAv1.pdf*

[Laddad03] *Laddad, R.:* AspectJ in Action, Practical Aspect-Oriented Programming. Manning, 2003.

[Larman2001] *Larman, C.:* Applying UML and Patterns: An Approach to Object-Oriented Analysis and Design. 2. Auflage. Prentice Hall, 2001.

[Longshaw+04] *Longshaw, A., Woods, E.:* Patterns for Generation, Handling and Management of Errors. Konferenzbeitrag zur OT2004. *http://www.blueskyline.com/ErrorPatterns/ErrorPatternsPaper.pdf*

[Lorenz94] *Lorenz, M., Kidd, J.:* Object-Oriented Software Metrics. Object-Oriented Series. Prentice Hall, 1994.

[Lublinsky 07] *Lublinsky, B.:* Defining SOA as an architectural style. IBM DeveloperWorks. *http://www-128.ibm.com/developerworks/architecture/library/ar-soastyle/*, Januar 2007

[Marick2000] *Marick, B.:* Using Ring Buffer Logging to Help Find Bugs. PloP 2000. *http://visibleworkings.com/trace/Documentation/ring-buffer.pdf*

[Marick97] *Marick, B.:* Classic Testing Mistakes. 1997. Online unter *http://www.testing.com/writings/classic/mistakes.pdf*.

[Martin08] *Martin, R.:* Clean Code: A Handbook of Agile Software Craftmanship. Addison-Wesley, 2008.

[Martin02] *Martin, R.:* Agile Software Development: Principles, Patterns and Practices. Addison-Wesley, 2002.

[Martin2000] *Martin, R.:* Design Principles and Design Patterns. *www.objectmentor.com*.

[Melzer07] *Melzer, I.:* Service-orientierte Architekturen mit Web Services: Konzepte, Standards, Praxis. Spektrum Akademischer Verlag, 2. Auflage, 2007.

[NATO69] *NATO Conference Software Engineering:* Bericht einer Konferenz, 1969. Gesponsert vom NATO Science Committee. Quelle nach [Shaw96a]

[OMG_UML] *Object Management Group:* Unified Modeling Language. *www.omg.org/uml*

[Open Group 09] *The Open Group:* TOGAF, Version 9. Online: *www.opengroup.org/architecture/togaf9-doc/arch/*

[Parnas72] *Parnas, D. L.:* On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 1972; 15: p. 1053–1058.

[Quiang+10] *Quiang, K., Fu, X. et al.:* Software Architecture and Design Illuminated. Jones and Bartlett Publishers, 2010.

[Rechtin2000] *Rechtin. E., Maier, M.:* The Art of Systems Architecture. CRC Press, 2000.

[Redmond+12] *Redmond, E., Wilson, J.:* Seven Databases in Seven Weeks. A Practical Guide to Databases and the NoSQL Movement. Pragmatic Programmers, 2012.

[Robertson 12] *Robertson, S. u. J.:* Mastering the Requirements Process: Gettig Requirements Right. 3. Auflage, Addison-Wesley, 2012.

[Ross+06] *Ross, J. W., Weill, P., Robertson, D.:* Enterprise Architecture as Strategie. Harvard Business School Press, 2006.

[Rupp+09] *Rupp, C., Sophist Group:* Requirements-Engineering und -Management. 5. Auflage, Hanser, 2009.

[Rupp+12] *Rupp, C., Queins, S., Sophisten:* UML 2 glasklar. Praxiswissen für die UML-Modellierung. Hanser, 4. Auflage, 2012.

[Schmeh98] *Schmeh, K.:* Safer Net. dpunkt, 1998.

[Schmidt2000] *Schmidt, D., Stal, M., Rohnert, H., Buschmann, F.:* Pattern-Oriented Software-Architecture – Patterns for Concurrent and Networked Objects. Vol. 2. Wiley, 2000.

[Schneier96] *Schneier, B.:* Applied Cryptography. John Wiley, 1996.

[SEI2001] *Software Engineering Institute (SEI):* How Do You Define Software Architecture? *http://www.sei.cmu.edu/architecture/definitions.html*

[Shaw+96] *Shaw, M., Garlan, D.:* Software Architecture: Perspectives on an Emerging Discipline. Upper Saddle River, Prentice Hall, New Jersey 1996.

[Siedersleben04] *Siedersleben, J.:* Moderne Softwarearchitektur. Umsichtig planen, robust bauen mit Quasar. dpunkt, 2004.

[SigG/SigV] Regulierungsbehörde für Telekommunikation und Post, *www.regtp.de*

[SOAX 07] *Starke, G., Tilkov, S. (Hg.):* SOA Expertenwissen. dpunkt, 2007.

[Starke+11] *Starke, G., Hruschka, P.:* Software-Architektur kompakt. 2. Auflage, Springer/Elsevier, 2011.

[SSH] Introduction to Cryptography. *http://www.ssh.com/tech/crypto/intro.cfm*

[Tanenbaum89] *Tanenbaum, A.:* Computer Networks. Prentice Hall, 1989.

[Tidwell-2011] *Tidwell, J.:* Designing Interfaces. O'Reilly, 2011.

[Tilkov-11] *Tilkov, S.:* REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien. 2. Auflage, dpunkt, 2011.

[VDAalst-2002] van der Aalst, Will: Workflow Patterns. Website: *http://workflowpatterns.com*

[Vernon13] *Vaughn, V.:* Implementing Domain-Driven-Design. Addison-Wesley, 2013.

[Völter-2013] Markus Völter, DSL Engineering. Online: *http://dslbook.squarespace.com/*

[Volere] Volere – Schablonen für Requirements Engineering.
Online unter *www.volere.co.uk* (englisch) oder *www.volere.de* (deutsch).

[Williams-2008] *Williams, R.:* The Non-Designers Design Book. PeachPit Press, 2008.

[Zörner12] *Zörner, S.:* Softwarearchitekturen dokumentieren und kommunizieren – Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirksam festhalten. Hanser, 2012.

Stichwortverzeichnis

EFFEKTIVE SOFTWARE-ARCHITEKTUREN //

- **Aktueller Überblick und methodische Einführung**
- **Direkt umsetzbare Tipps für praktizierende Softwarearchitekten**
- **Ideal zur Vorbereitung auf die Zertifizierung zum »Certified Professional for Software Architecture™« (Foundation Level) des iSAQB**
- **Die 6. Auflage enthält eine umfangreiche Darstellung von Architekturstilen und -mustern, ausführliche technische Konzepte, NoSQL-Datenbanken sowie aktualisierte und erweiterte Beispielarchitekturen.**
- **Detaillierte Beispiele zum Einsatz von arc42**

Softwarearchitekten müssen komplexe fachliche und technische Anforderungen an IT-Systeme umsetzen und diese Systeme durch nachvollziehbare Strukturen flexibel und erweiterbar gestalten.

Dieser Praxisleitfaden zeigt Ihnen, wie Sie Softwarearchitekturen effektiv und systematisch entwickeln können. Gernot Starke unterstützt Sie mit praktischen Tipps, Architekturmustern und seinen Erfahrungen.

Sie finden Antworten auf zentrale Fragen:

- Welche Aufgaben haben Softwarearchitekten?
- Wie gehen Sie beim Entwurf vor?
- Wie kommunizieren und dokumentieren Sie Softwarearchitekturen?
- Wie helfen Architekturstile und -muster?
- Wie bewerten Sie Softwarearchitekturen?
- Wie behandeln Sie Persistenz, grafische Benutzeroberflächen, Geschäftsregeln, Integration, Verteilung, Sicherheit, Fehlerbehandlung, Workflow-Management und sonstige technische Konzepte?
- Was müssen Softwarearchitekten über NoSQL, Domain-Driven-Design und arc42 wissen?
- Welche Aufgaben nehmen Enterprise-IT-Architekten wahr?

Dr. Gernot **STARKE** stellt sich seit vielen Jahren der Herausforderung, die Architektur großer Systeme effektiv zu gestalten. Zu seinen Kunden zählen mittlere und große Unternehmen aus den Branchen Finanzdienstleistung, Logistik, Handel, Telekommunikation und dem öffentlichen Bereich. Er ist Mitinitiator und -betreiber von arc42, Mitgründer des iSAQB e.V. sowie Fellow der innoQ.

@gernotstarke

AUS DEM INHALT //

- **Vorgehen bei der Architekturentwicklung**
- **Architekturmuster und -stile**
- **Technische Konzepte**
- **SOA und Enterprise-IT-Architektur**
- **Architekturbewertung**
- **Dokumentation von Architekturen**
- **Modellierung für Softwarearchitekten**
- **Werkzeuge für Softwarearchitekten**
- **Beispiele realer Softwarearchitekturen**
- **iSAQB Curriculum**

HANSER

€ 44,99 [D] | € 46,30 [A]
ISBN 978-3-446-43614-5

9 783446 436145

www.hanser-fachbuch.de/computer