Peter Zöller-Greer

Software-
Architekturen

Peter Zöller-Greer

Software - Architekturen

sind Baupläne für Software. Es gibt -wie im Baugewerbe- verschiedene (An-)Sichten für verschiedene Projektbeteiligte. Jede Ansicht beherbergt eigene Darstellungsbeschreibungen. Die wichtigsten werden im vorliegenden Werk behandelt.

In der gleichen Reihe erschienen:

Über 1 Jahr
fast ununterbrochen
No. 1 - Bestseller bei amazon.de
z.T. in 3 Kategorien
gleichzeitig!

Prof. Dr. rer. nat. Peter Zöller-Greer
ist Mathematiker und unterrichtet die Fächer Künstliche Intelligenz, Software-Engineering und Multi Media Systeme an der FH Frankfurt am Main-University of Applied Sciences.

9 783981 163933
Preis: 22,- Euro

ISBN 978-3-9811639-3-3

Die Reihe *Wissen & Praxis >kompakt<* nimmt sich komplexer Themen an und versucht diese so einfach wie möglich, beschränkt auf das Wesentliche, für Studium und Praxis gleichermaßen geeignet darzustellen.

Composia Verlag
www.composia.de

Software-Architekturen
Zöller-Greer

Composia Verlag

Software-Architekturen

Grundlagen und Anwendungen

Mit einer Einführung in
Architekturbeschreibungssprachen (ADLs)
UML, Object-Z, OCL, CORBA, IDL
Entwurfsmuster, Architektursichten
Architekturmuster, DDD, Architektur-Dokumentation
Komplexitätsprobleme, Standard-Architekturen
SOA, TOGAF, RM-ODP
Software Factories

Reihe *Wissen & Praxis >kompakt<*

Composia Verlag

Peter Zöller-Greer

Software Architekturen

Grundlagen und Anwendungen

Mit einer Einführung in
Architekturbeschreibungssprachen (ADLs)
UML, Object-Z, OCL, CORBA, IDL
Entwurfsmuster, Architektursichten
Architekturmuster, DDD, Architektur-Dokumentation
Komplexitätsprobleme, Standard-Architekturen
SOA, TOGAF, RM-ODP
Software Factories

Prof. Dr. Peter Zöller-Greer
studierte nach Abschluss einer Berufsausbildung als Physiklaborant (BASF AG Ludwigshafen/Rh.) von 1975-1981 Mathematik (Diplom) und Theoretische Physik an den Universitäten Siegen und Heidelberg. Er promovierte an der Universität Mannheim über eine approximationstheoretische Lösung eines Problems aus der Quantenmechanik und arbeitete zunächst als Systemanalytiker bei Brown Boveri Reaktor GmbH und danach als DV-Referent bei ABB Mannheim, bevor er 1989 die Geschäftsführung der Firma Composia GmbH in Mannheim übernahm. Bereits während dieser Tätigkeiten arbeitete er als freier Dozent an verschiedenen Hochschulen. Seit 1993 ist er Professor am Fachbereich Informatik und Ingenieurwissenschaften der FH Frankfurt am Main – University of Applied Sciences. Seine Lehr- und Arbeitsgebiete sind Künstliche Intelligenz, Software-Engineering und Multimedia-Systeme.

Bibliographische Information der Deutschen Bibliothek:
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet über http://dnb.ddb.de abrufbar.

3. Auflage

ISBN 978-3-9811639-3-3

Inhalt:

Vorwort		**4**
1	**Einführung**	**5**
2	**Architekturbeschreibungssprachen (ADL)**	**10**
3	**Entwurfsmuster**	**74**
4	**Architektur-Sichtenmodelle**	**96**
5	**Komplexitätsprobleme**	**110**
6	**Architektur-Muster und Standards**	**123**
7	**Dokumentation von Architekturen**	**143**
8	**Software-Factories**	**149**
Anhang	**Lösungen der Übungsaufgaben**	**152**
	Weiterführende Literatur	**164**
	Index	**165**

Vorwort

Es ist soweit: Nachdem die Disziplin „Software-Engineering“, welche bekanntlich mit ingenieurmäßigen Methoden versucht, Software zu entwickeln, sich etabliert hat, folgte der nächste logische Schritt: Nicht nur das „Feine“ muss geplant werden, auch das „Grobe“ (es sollte eigentlich umgekehrt sein). Und sowie der Bauingenieur sich an den Plan des Architekten hält, so soll auch in der Softwareentwicklung am Anfang zuerst ein Software-Architekt eine Software-Architektur entwickeln, welche alle Aspekte der Entwicklungsplanung umfasst. Mit diesem Plan können dann die einzelnen Projektbeteiligten (engl. „Stakeholder“) ihre Teilaufgaben lösen.

Bei dem vorliegenden Werk handelt es sich –wie bei allen Bücher dieser Reihe- um einen „Crash-Kurs“, der nicht den Anspruch erhebt, ein Lehrbuch zu sein; vielmehr soll es als Einführung für Anfänger oder als Überblick für Fortgeschrittene oder als Repetitorium für die Klausurvorbereitung dienen.

Es werden Grundkenntnisse im Umfang z.B. des Buches „Software Analyse und Design“ vom gleichen Autor aus der gleichen Reihe vorausgesetzt (vgl. hinteres Buch-Cover). Es wird auch dringend empfohlen, die Übungsaufgaben am Ende jedes Kapitels zu lösen; bei der Kompaktheit der Darstellung des Stoffes sind die Übungen wesentlicher Bestandteil des Lernprozesses und für das Gesamtverständnis unabdingbar.

Zu jedem dieser Kapitel gibt es eigene, z.T. sehr umfangreiche Lehrbücher, daher können die hier präsentierten Kapitel das jeweilige Gebiet natürlich nur einführend behandeln.

Ein besonderer Dank geht an Frau Mary Suriyakumar, B.Sc., die einige der grafischen Darstellungen entworfen und zur Verfügung gestellt hat. Ebenfalls Dank an Herrn Dr. Gernot Starke, der mir freundlicherweise auch einige seiner Grafiken aus dessen Buch „Effektive Software-Architekturen“ überlassen hat (entsprechend gekennzeichnet). Dank auch Herrn cand. Inf. Shahid Lodhi für die Durchsicht des Manuskripts.

Ich möchte an dieser Stelle auch meiner lieben Frau Diana für ihre unermüdliche Geduld mit mir danken. Dank auch unseren „Cratchits“, die mir einen neuen Blick auf das Wesentliche gaben.

Im Februar 2010

Peter Zöller-Greer

1. Einführung

Eine Architektur bezeichnet im Allgemeinen das Ergebnis eines planvollen Entwurfs und der Gestaltung z.B. eines Gebäudes. In der Informatik wird dieser Begriff mannigfaltiger verwendet, man spricht beispielsweise von Hardware-Architekturen und von Software-Architekturen.
Wie so oft im Software-Engineering sind jedoch Begriffe, gerade wenn es sich um ein neues Gebiet handelt, in der Literatur oft verschieden definiert. Wenn man auch bei diesen verschiedenen Definitionen erahnt, dass es sich dabei wohl um das gleiche handelt, so weichen die konkreten Beschreibungen dann doch häufig leicht voneinander ab. Das Fraunhofer-Institut für Software Engineering (IESE), welches im Rahmen eines vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Projekts „Quasi-Normen“ definiert hat, legt für den Begriff der Software-Architektur fest:

> Unter einer *Software-Architektur* versteht man eine Spezifikation über die Teile des zu entwickelnden Systems (Komponenten genannt), welche ihre Konnektoren sowie die Regeln für die Interaktion dieser Konnektoren mit den Systemteilen enthält.

Die hier verwendeten Begriffe (System, Konnektor, Interaktion) sind recht allgemein gehalten. In der Baubranche könnte man im weitesten Sinn bei den Komponenten z.B. vom Dach, dem Keller, den Stockwerken etc. sprechen, während die Konnektoren dort z.B. die Wasserleitungen, Stromleitungen und ähnliches bezeichnen können. Wir werden die Begriffe Komponenten und Konnektoren im Sinn der Software-Architektur später noch konkretisieren. Der Vorteil dieser Allgemeinheit ist immerhin, dass man sie auf alle Abstraktionsebenen und Sichten anwenden kann. Es handelt sich im weitesten Sinne bei einer Software-Architektur um die Zusammenfassung der am Projekt beteiligten Strukturen, d.h. aller Komponenten sowie deren Merkmale und Beziehungen.
Wie im richtigen Leben eines Bau-Architekten geht es hier also auch um die Errichtung eines „Bauplans“ und eines „Ablaufplans“.
Um dies besser zu verstehen, wenden wir uns nochmal der Analogie im Baubereich zu.
Wenn Kunden zu einem Architekten gehen, um sich ein Haus planen zu lassen, so offerieren sie zunächst dem Architekten ihre Wünsche nebst Umfang des Geldbeutels. Letzterer bestimmt bekanntlich entscheidend die Ausprägungen im geplanten Objekt. Zuerst wird das „Grobe“ beschrieben: Wo soll das Haus hin, wie sind die Grundstücksbergenzungen, ist das Grundstück bereits befriedet, welche Größe hat es (müssen z.B. bestimmte Abstände vom Nachbargrundstück gesetzlich eingehalten werden) etc.; zu diesem Zweck wird der Architekt in der

Regel auf einen Parzellenplan des Grundstücks zurückgreifen, der gewissermaßen aus der „Vogelperspektive“ den gewünschten Bauort nebst seinem Umfeld darstellt. Abbildung 1.1 zeigt ein Beispiel dafür.

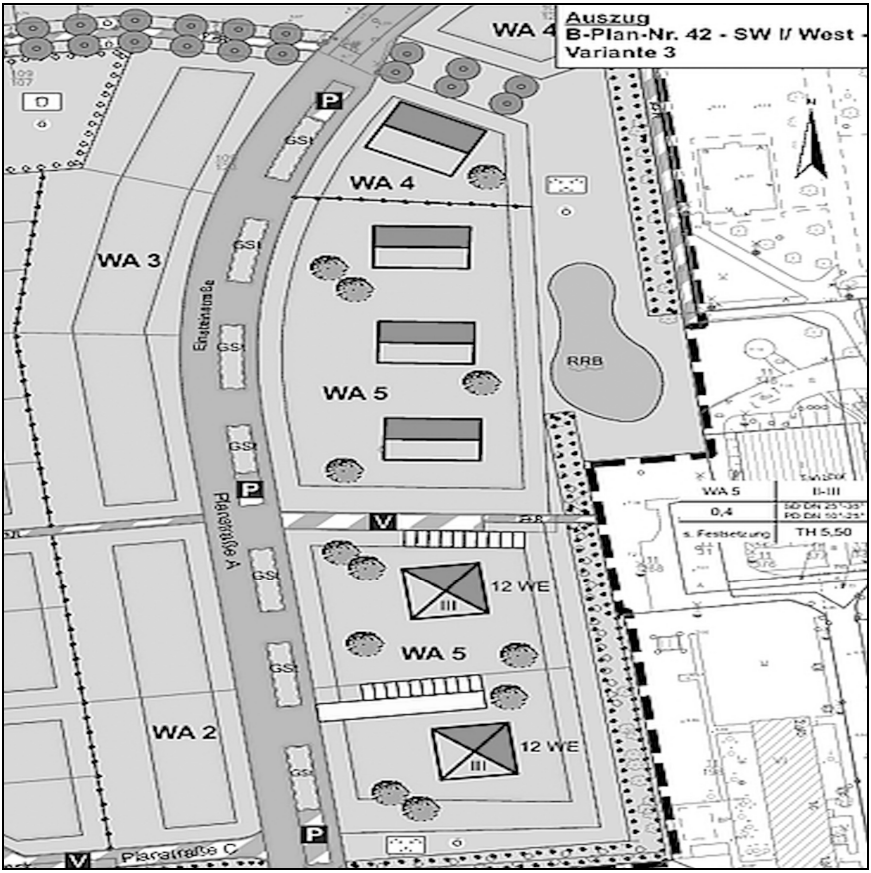


Abb. 1.1 Parzellenplan

Auch in der Software-Architektur gibt es dafür ein Analogon: Die sog. Kontext-Sicht. Wir werden dies im Kapitel „Architektursichten“ noch genauer ausführen, doch das kann man schon verraten: Die Kontextsicht im Rahmen der Software-Architektur gibt ebenfalls Auskunft über das „Umfeld“ der geplanten Software: vorhandene DV-Landschaften, umliegende Hard- und Software, Schnittstellen und so weiter.

Kennt der Architekt also den „Kontext“ des geplanten Bauvorhabens, so müssen dann Pläne her für die Ansichten des Hauses, für die Verlegung der Wasser- und Abflussrohre, für die elektrischen Leitungen etc.; es sind also mehrere Pläne für das gleiche Bauobjekt erforderlich, die verschiedene Ansichten des Objekts ermöglichen, und diese sind zum Teil auch nur für je eine bestimmte Zielgruppe erstellt (also z.B. die Wasserrohverläufe für die Sanitärfirma oder der Stromkabelleitungsplan für den Elektriker). In den Abbildungen 1.2 bis 1.4 sehen wir Beispiele dafür.

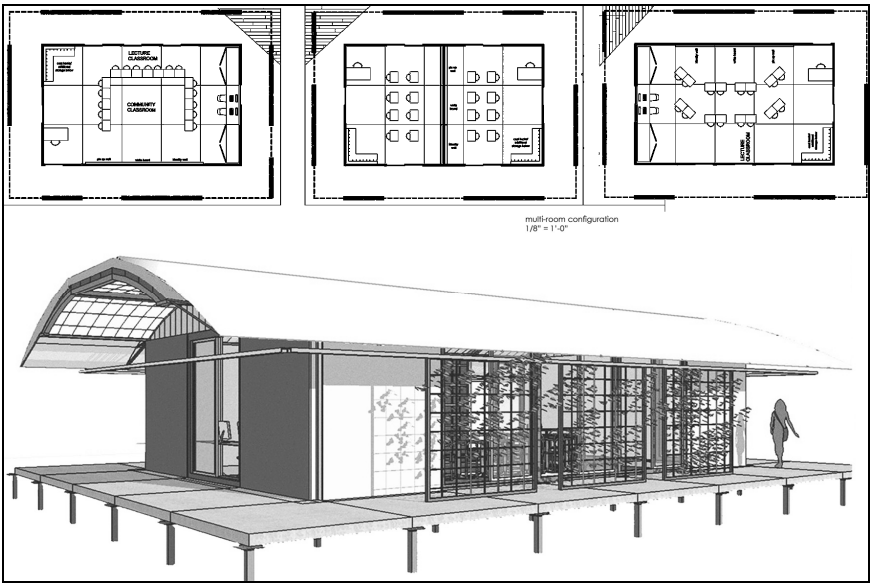


Abb. 1.2 Außenansicht und Draufsicht innen[1]

So wie es also verschiedene Ansichten für bestimmte Zielgruppen eines Bauvorhabens gibt, so wird im Software-Engineering dieses Analogon aufgegriffen, da man festgestellt hat, dass heutzutage reine Diagramme und/oder Beschreibungen zur Datenmodellierung oder zum Laufzeitverhalten bei weitem nicht mehr ausreichend sind. Insbesondere setzen solche Diagramme voraus, dass sie alle Stakeholder lesen können oder die Beschreibungen verstehen. In der heutigen Zeit sind aber die Verzahnungen und die Komplexität so groß geworden,

[1] Quelle: http://de.wikipedia.org/w/index.php?title=Datei:Sustainable_Portable_Classroom_-_The_Learning_Kit.jpg&filetimestamp=20090217091448

dass –ähnlich wie beim Bau- Spezialisten Einzelaufgaben vornehmen (beim Hausbau z.B. der Elektriker oder der Maurer oder der Innenausstatter).

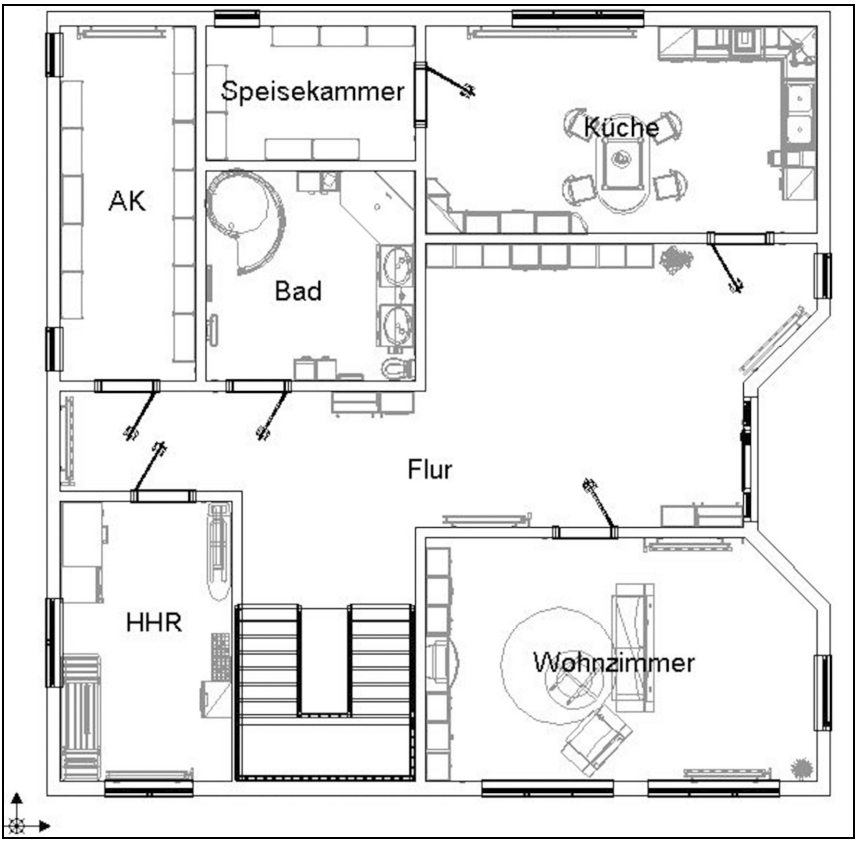


Abb. 1.3 Einrichtungsplan innen[2]

Daraus ergab sich die Notwendigkeit, auch im Software-Engineering „passende" Baupläne, zugeschnitten auf die jeweiligen Projektbeteiligten, einzuführen.
Und wie im Baugewerbe sind die jeweiligen Architektur-Sichten mit Eigenheiten angefüllt, die dann häufig in einer „normierten" Sprache -meistens nur für die entsprechende Zielgruppe verständlich- abgefasst sind. In Abb. 1.4 sehen wir z.B. einen Abwasserplan. In der Regel wird ein Elektriker damit wenig anzufangen wissen. Und er braucht es auch gar nicht. Für ihn sind eben nur die Verle-

[2] Quelle: http://de.wikipedia.org/w/index.php?title=Datei:GR-Erdgeschoss.jpg&filetimestamp=20051105204711

gungspläne der elektrischen Leitungen relevant, welche wiederum die Sanitärfirma wenig interessiert.

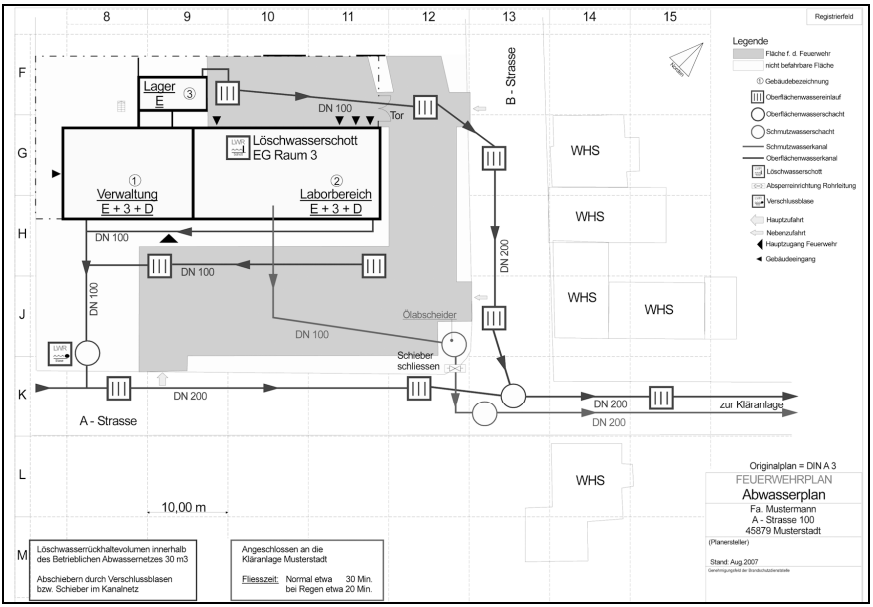


Abb. 1.4 Abwasserplan[3]

Ich denke, Sie haben jetzt eine Grundidee bekommen, was in einer Software-Architektur drinsteht, auch wenn die Details natürlich noch im Dunkel liegen: Es soll für ein Softwareprojekt ein „Satz“ von Ansichten erstellt werden, der für die jeweiligen Projektbeteiligten jeweils anders aussehen kann. Ein DV-Manager will eine andere Sicht auf das Projekt haben als der C#-Programmierer. Und diesem Umstand wird bei Software-Architekturen Rechnung getragen.
Wir werden später also vor allem ganz genau auf die Entwicklung und Dokumentation der jeweiligen Architektursichten eingehen. Doch bevor wir das können, müssen wir –wie der Elektriker oder der Sanitärbetrieb im Baugewerbe- die jeweilige „Fachsprache“ für jede Ansicht lernen. Im Software-Engineering nennt man das „Architekturbeschreibungssprachen“ (ADL = Architecture Description Languages). Im nächsten Kapitel werden wir einige solcher Sprache genauer unter die Lupe nehmen.

[3] Quelle: http://www.feuerwehr-limburg.de/Feuerwehrplaene/pdf/13%20Detailplan-Abwasser.pdf

2. Architekturbeschreibungssprachen (ADL)

Architecture Description Languages (ADL) sind, wie im vorherigen Kapitel schon angedeutet, in erster Linie dazu da, für bestimmte Zielgruppen den jeweiligen Ausschnitt oder auch die Gesamtheit des geplanten Softwareprojekts zu dokumentieren.
Wenn man will, kann man diese Sprachen in zwei ganz grobe Gruppen unterteilen: die erste Gruppe ist eine eher grafisch orientierte Spezifizierung wie sie z.B. durch UML-Diagramme oder ähnliches geleistet wird. Daneben kann man eine zweite Gruppe, nämlich die Gruppe der eher formalen Beschreibungssprachen unterscheiden, in denen z.B. ähnlich wie in Pseudo-Algorithmen bestimmte Sachverhalte beschrieben werden. Solche Beschreibungen können –so wie in konventionellen Programmiersprachen üblich- „sequentiell“ mittels if…then-Konstruktionen oder ähnlichem ausgestattet sein. Aber es sind auch prädikatenlogische Darstellungen denkbar, wie das z.B. in der Spezifikationssprache „Object-Z“ realisiert ist.
Nun würde es den Rahmen dieses Buches sprengen, auf alle verbreiteten ADLs im Einzelnen einzugehen, zumal diese sich häufig nur rein syntaktisch unterscheiden (so wie verschiedene Programmiersprachen untereinander auch). Wir wollen daher jeweils die zur Zeit verbreitesten Repräsentanten der beiden genannten Gruppen genauer untersuchen und andere, sich mehr oder weniger auch in Umlauf befindliche Sprachen zumindest kurz andeuten und deren Aufbau sowie Einsatzschwerpunkte betrachten.

UML als Architekturbeschreibungssprache

UML (Unified Modeling Language) stellt wohl die zur Zeit bekannteste Modellierungssprache dar. Sehr weit verbreitet sind Klassendiagramme, die hauptsächlich zur Darstellung (statischer) Datenmodelle eingesetzt werden.
Aber UML bietet wesentlich mehr und kann im Rahmen der Software-Architekturdokumentation in vielen Fällen ausreichend alle gegebenen Sachverhalte abbilden.
In Abb. 2.1 sehen wir einen Überblick über die einzelnen Diagrammtypen von UML. Wir werden nachfolgend jeden der 13 Diagrammtype genauer untersuchen und Beispiele dafür geben.

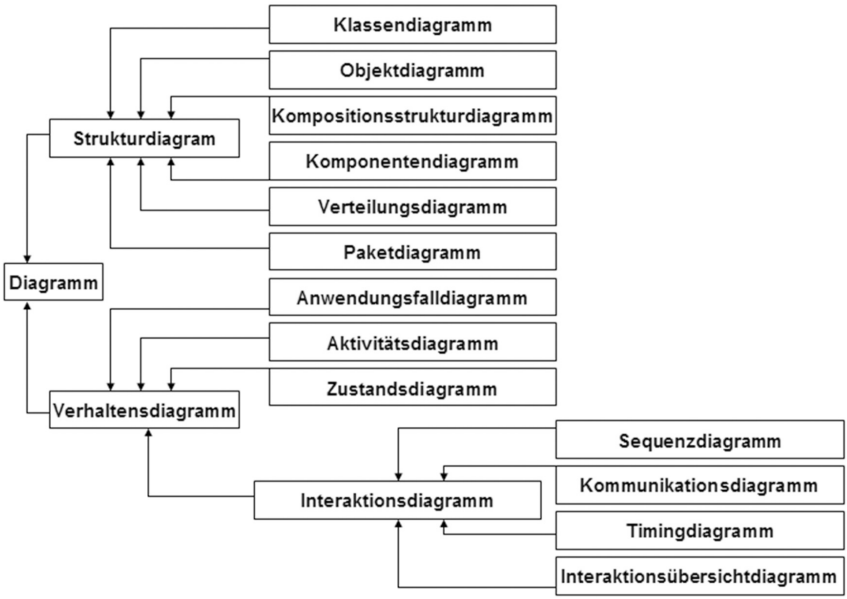


Abb. 2.1 UML-Diagrammtypen

Grob teilt man also UML-Diagramme in die zwei Klassen „Verhaltensdiagramme“ und „Strukturdiagramme“ ein. Welches Diagramm im Rahmen der jeweiligen Architekturbeschreibung gerade benutzt wird, hängt natürlich vom Zweck der Beschreibung ab. Allerdings können häufig bestimmte Sachverhalte durch verschiedene Diagrammtypen alternativ und gleich komplett beschrieben werden (z.B. kann manchmal ein Zustandsdiagramm alternativ zu einem Sequenzdiagramm eingesetzt werden, man braucht dann nicht unbedingt beides).
In der gängigen Literatur findet man folgende Definitionen für diese Diagrammtypen[4]:

Klassendiagramm: *Ein Klassendiagramm stellt eine Menge von Klassen, Interfaces und Kollaborationen sowie ihre Beziehungen dar. Klassendiagramme sind die bei der Modellierung softwareintensiver Systeme am häufigsten eingesetzten Diagramme. Mithilfe von Klassendiagrammen kann man die statische Entwurfs-*

[4] Vgl. z.B. Kecher, C., *UML 2.0, Das umfassende Handbuch*, 3. Auflage 2009; Erler, T., Ricken, M., *UML 2, Das bhv Taschenbuch*, 1. Auflage, verlage moderne industrie 2005; Booch G., Rumbaugh J., Jacobson I., *Das UML-Benutzerhandbuch*, Perason Education, 2006

sicht eines Systems darstellen. Klassendiagramme mit aktiven Klassen verwendet man zur Darstellung der statischen Prozesssicht eines Systems.
Objektdiagramm: *Ein Objektdiagramm zeigt eine Reihe von Objekten und deren Beziehungen. Objektdiagramme werden eingesetzt, um Datenstrukturen darzustellen; sie beschreiben statische Schnappschüsse der Instanzen der Klassendiagramme. Objektdiagramme bieten die statische Entwurfssicht oder die statische Prozesssicht eines Systems ebenso wie es Klassendiagramme tun, allerdings aus einer Perspektive von realen Klassen oder Klassenprototypen.*
Kompositionsstrukturdiagramm*: Ein Kompositionsstrukturdiagramm zeigt die internen Strukturen eines Classifiers. Classifiers können Klassen oder andere Systemelemente wie Zustände oder Aktivitäten sein. Dabei werden die Interaktionspunkte mit anderen Systemelementen herausgegriffen, die die Interaktion mit anderen Systemelementen bestimmen.*
Komponentendiagramm: *Ein Komponentendiagramm beschreibt eigenständige Softwareeinheiten und deren Zusammenspiel. Als Komponente bezeichnet man einen wiederverwendbaren Programmcode oder eine Hardware, welche eine abgegrenzte Aufgabe erfüllt. Komponenten sind von außen betrachtet Einheiten, die zur Anbindung anderer Komponenten Schnittstellen besitzen.*
Verteilungsdiagramm: *Ein Verteilungsdiagramm zeigt eine Menge von Knoten und deren Beziehungen. Verteilungsdiagramme sind dadurch mit Komponentendiagrammen verwandt, da ein Knoten üblicherweise eine oder mehrere Komponenten beinhaltet.*
Paketdiagramm: *Ein Paketdiagramm gibt einen Überblick über die Zerlegung des Gesamtsystems in Pakete oder Teilsysteme. Paketdiagramme enthalten logische Zusammenfassungen von Systemteilen.*
Anwendungsfalldiagramm: *Ein Anwendungsfalldiagramm (auch Use-Case- oder U-Case-Diagramm genannt) zeigt eine Menge von Anwendungsfällen und Akteuren (eine besonderen Form von Klassen) und deren Beziehungen. Man setzt Anwendungsfalldiagramme ein, um die statische Anwendungsfallsicht eines Systems darzustellen. Anwendungsfalldiagramme sind besonders für das Modellieren und Organisieren der Verhaltensweisen eines Systems von Bedeutung.*
Aktivitätsdiagramm: *Ein Aktivitätsdiagramm zeigt den schrittweisen Verlauf einer Berechnung. Eine Aktivität stellt dabei eine Reihe von Aktionen dar. Aktivitätsdiagramme sind besonders für die Modellierung der Funktionen eines Systems von Bedeutung. Aktivitätsdiagramme unterstreichen den Steuerungsfluss innerhalb der Ausführung eines Verhaltens.*
Zustandsdiagramm: *Ein Zustandsdiagramm stellt einen Zustandsautomaten dar, der aus Zuständen, Übergängen, Ereignissen und Aktivitäten besteht. Zustandsdiagramme werden eingesetzt, um die dynamische Sicht eines Systems zu modellieren. Sie sind vor allem für das Modellieren eines Interface, einer Klasse oder einer Kollaboration von Bedeutung. Bei Zustandsdiagrammen liegt der*

Schwerpunkt auf dem Verhalten eines Objekts nach einer Reihenfolge von Ereignissen. Dies ist für das Modellieren reaktiver Systeme von besonderer Bedeutung.

Sequenzdiagramm: *Ein Sequenzdiagramm ist ein Interaktionsdiagramm, das die zeitliche Abfolge von Nachrichten hervorhebt. Sequenzdiagramme setzt man ein, um die dynamische Sicht eines Systems darzustellen.*

Kommunikationsdiagramm: *Ein Kommunikationsdiagramm ist ein Interaktionsdiagramm, das die strukturelle Organisation der Dinge hervorhebt, die Nachrichten senden und empfangen. In einem Kommunikationsdiagramm sind Rollen, Verbinder (Connectoren) zwischen den Rollen und die Nachrichten dargestellt, die zwischen den die Rollen spielenden Instanzen ausgetauscht werden.*

Timingdiagramm: *Ein Timingdiagramm (auch Zeitverhaltensdiagramm genannt) ist ein Interaktionsdiagramm, welches die Zeitabhängigkeiten zwischen Ereignissen zeigt und die Zustandsänderungen in Abhängigkeit vom Zeitverlauf beschreibt.*

Interaktionsübersichtdiagramm: *Ein Interaktionsübersichtdiagramm ist ein Interaktionsdiagramm, welches das Zusammenspiel verschiedener Interaktionen aufzeigt und in der Regel aus Referenzen auf andere Interaktionsdiagramme besteht. Es ist eine Art Aktivitätsdiagramm auf hoher Abstraktionsebene.*

Wir wollen nun jedes dieser Diagrammtypen kurz anhand eines kleinen Beispiels betrachten.

Es sei dabei darauf hingewiesen, dass vor allem Klassendiagramme sehr verbreitet sind und der sichere Umgang damit nur anhand von vielen Beispielen eingeübt werden kann. Der Leser sollte, falls er hier zum ersten Mal mit Klassendiagrammen und deren Beziehungen untereinander (wie Assoziationen, Vererbungen, Aggregationen) konfrontiert wird, dies durch entsprechende Literatur dazu einüben[5].

Klassendiagramm und Objektdiagramm

Beginnen wir also mit dem Klassendiagramm. Klassen stellen bekanntlich eine Zusammenfassung zusammengehöriger Objekte dar. Die Zusammengehörigkeit wird dabei über Eigenschaften, sog. Integritätsbedingungen, definiert. Eine Schulklasse einer Grundschule besteht beispielsweise aus den Objekten „Schüler“, welche alle gemeinsame Eigenschaften besitzen: Sie sind (meistens) ungefähr gleich alt, gehören zum gleichen Einzugsgebiet einer Stadt, haben bereits die vorhergehende Klasse erfolgreich absolviert (Versetzung) und so weiter. Von einer gewissen Abstraktionsstufe aus gesehen sind diese „Objekte“ (bezüglich ihrer Integritätsbedingungen) sogar „ununterscheidbar“. Jedes Objekt stellt

[5] Vgl. z.B. Zöller-Greer, P.: *Software Analyse und Design*, Verlag Composia 2007

dabei einen Repräsentant der ganzen Klasse dar. Objekte werden auch Instanzen oder Exemplare einer Klasse genannt.
Klassen werden in UML durch Rechtecke dargestellt, die wieder in 3 Abteilungen unterteilt sind: Oben stehen der Klassenname, in der Mitte die Attribute und im unteren Teil die Methoden, mit denen auf die Objekte der Klasse zugegriffen werden kann. Außer dem Klassennamen sind alle anderen Einträge optional. Die Beziehungen zwischen Klassen werden durch Verbindungslinien ausgedrückt, welche auch beschriftet werden können („Rolle“ der Beziehung).
In Abb. 2.2 sehen wir ein Beispiel für so ein Klassendiagramm und in Abb. 2.3 für ein Objektdiagramm. Objektdiagramme sehen ähnlich wie Klassendiagramme aus, jedoch beziehen sich die Beschriftungen innerhalb der Rechtecke und der Verbindungslienen jetzt auf konkrete Objekte der Klassen.
An die Verbindungslinien werden bei dem Beziehungstyp „Assoziation“ noch die Kardinalitätsbeschränkungen dazu angegeben. Die wichtigsten sind:

*	für	keine oder viele
1..*	für	ein oder viele
0..1	für	keine oder ein
1	für	genau ein (diese 1 kann auch ganz weggelassen werden)
2..4	für	numerische Angaben, hier: 2, 3 oder 4.

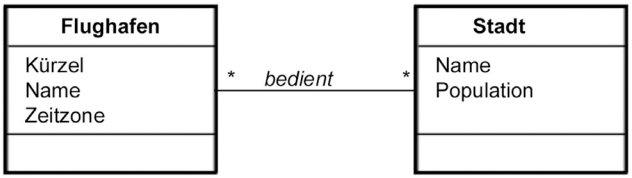


Abb. 2.2 Klassendiagamm mit Assoziation und Kardinalitätsbeschränkungen

Eine Assoziation wird als eine Gruppe von Links definiert, wobei ein Link eine Verbindung zwischen einander zugeordneten Objekten bezeichnet (bei 2 an einer Assoziation beteiligten Klassen werden dann auf Instanzenebene jeweils immer genau 2 Objekte einander zugeordnet). Auf Instanzenebene werden daher im Objektdiagramm die Kardinalitäten aller Links immer genau 1 betragen, und diese 1 wird in der Regel in den Diagrammen weggelassen.

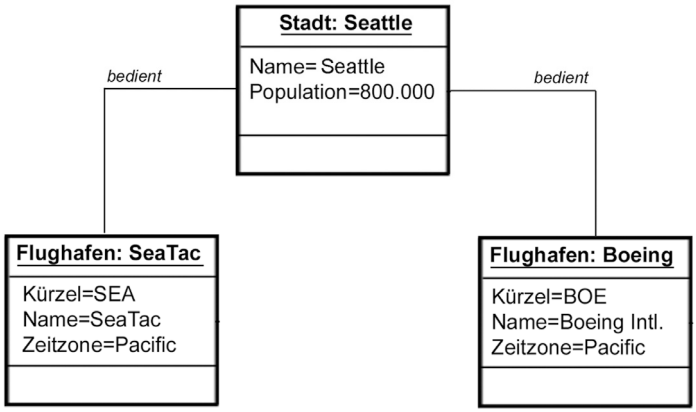


Abb. 2.3 Objektdiagramm mit Links

Manchmal besitzen Assoziationen eigene Attribute. Diese werden dann Linkattribute genannt, und die dazu gehörige Klasse als Assoziationsklasse bezeichnet (vgl. Abb. 2.4).

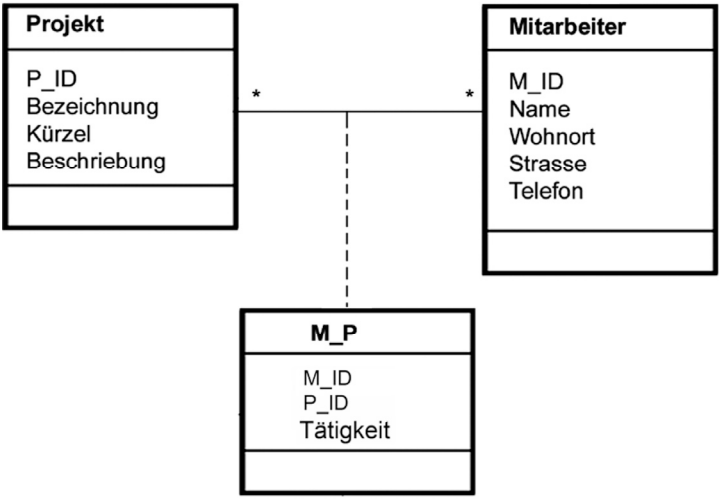


Abb. 2.4 Klassendiagamm mit Assoziationsklasse

Ich möchte an dieser Stelle einer weit verbreitenden Quelle für Verwirrung entgegen treten, und zwar in Bezug auf die Beschriftung der Kardinalitäten in Zusammenhang mit den Rollen. Betrachten wir zu diesem Zweck folgendes Beispiel: Eine Schallplattenfirma schließt einen Vertrag mit einem Künstler ab. In der Musikbranche ist es dabei üblich, dass ein Künstler immer exklusiv an eine Plattenfirma gebunden ist, während eine Plattenfirma viele Künstler unter Vertrag haben kann. In Abb. 2.5 sehen wir dazu ein mögliches Klassendiagramm:

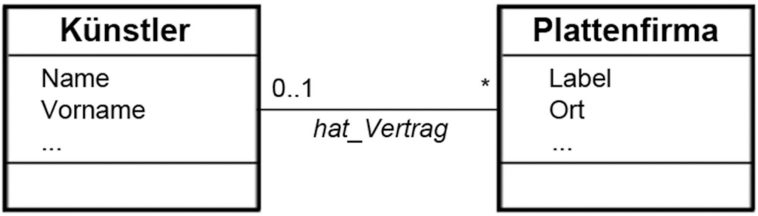


Abb. 2.5 Beispiel Plattenfirma, Sicht 1

In dieser Kardinalitätenschreibweise wird folgende Sicht vertreten:

Sicht 1 (Rolle):
Es wird *an* der Klasse aufgezählt, wie oft ein Exemplar dieser Klasse an einem Link mit einem Exemplar der gegenüberliegenden Klasse teilnehmen kann.

Nun gibt es aber verwirrenderweise eine zweite Sichtweise, die zwar denselben Sachverhalt zum Ausdruck bringt, aber eine andere Sicht auf die Assoziation darstellt (Abb. 2.6):

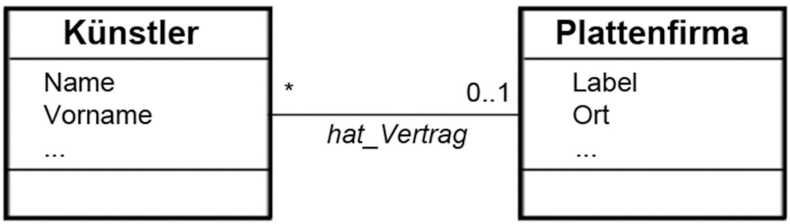


Abb. 2.6 Beispiel Plattenfirma, Sicht 2

Wie man sieht, sind jetzt die Kardinalitätsbeschränkungen gerade vertauscht. In dieser Sichtweise gilt:

Sicht 2 (inverse Rolle):
Es wird an der *gegenüberliegenden* Klasse aufgezählt, wie viele Links ein Exemplar der aktuellen Klasse mit der gegenüberliegenden Klasse eingehen kann.

Die Sicht der inversen Rolle hat sich interessanter Weise mehr verbreitet als die der Rolle. Auch wir werden, wenn nicht anders angegeben, in diesem Buch letztere Sichtweise vertreten. Um Mehrdeutigkeiten zu vermeiden kann es nicht schaden, wenn man bei der Erstellung von Klassendiagranmen z.B. mit Hilfe einer Legende hinschreibt, welche Sicht man gerade einnimmt.

Abschließend hierzu noch die Sicht auf eine mögliche konkrete Implementierung dieses Beispiels in einer relationalen Datenbank (Abb. 2.7):

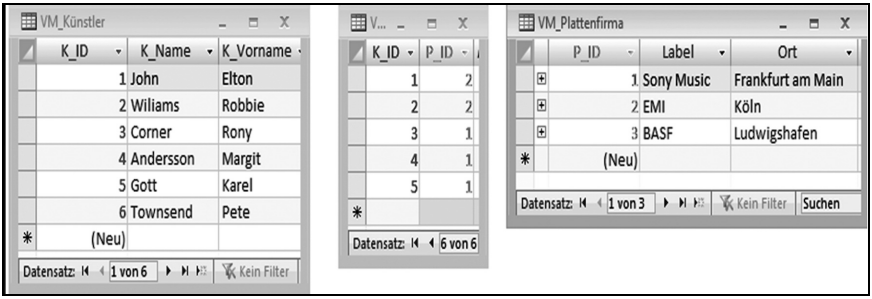


Abb. 2.7 Implementierungsbeispiel

Der Leser sollte sich die beiden Sichtweisen anhand der konkreten Implementierung nochmals verinnerlichen: Die Assoziation wurde hier anhand einer eigenen Tabelle realisiert, obwohl das hierfür nicht nötig gewesen wäre (man würde normalerweise den Primärschlüssel der Plattenfirma (*P_ID*) als Fremdschlüssel in einer zusätzlichen Spalte der Künstlertabelle unterbringen); eine eigene „Beziehungstabelle“ erzeugt man bekanntlich nur bei einer viele-zu-viele-Beziehung. Dennoch ist das so erlaubt (und hat sogar den Vorteil, dass bei Künstlern ohne Plattenvertrag kein Nullwert in der Fremdschlüsselspalte des Künstlers eingetragen ist, was die Datenbanktheoretiker besonders freuen dürfte) und soll hier zur Veranschaulichung der beiden Sichtweisen dienen.

Während Assoziationen „hat“-Beziehungen zwischen Klassen realisieren, stellen Spezialisierungen (je nach Sichtweise auch Generalisierungen oder Vererbungen genannt) Variationen einer Klasse im Sinne einer „ist-ein“-Beziehung dar. Die Verbindungslinien besitzen einen holen Pfeil am Ende der Basisklasse (auch

Oberklasse genannt); die Variationen werden auch abgeleitete Klassen, Subklassen oder Unterklassen genannt. Letztere ererben von der Basisklasse eventuelle Attribute, Methoden und Assoziationen (vgl. Abb. 2.8). Optional kann in der Nähe des Pfeils noch in geschweiften Klammern ein Eintrag vorgenommen werden, der folgendes bedeutet:

complete	heißt, dass die Basisklasse keine eigenen Instanzen enthält
incomplete	heißt, die Basisklasse kann eigene Instanzen haben
overlapping	heißt, gleiche Objekte können gleichzeitig in mehreren Unterklassen vorkommen
disjoint	heißt, ein Objekt einer Subklasse kann nicht noch in einer weiteren Subklasse vorkommen

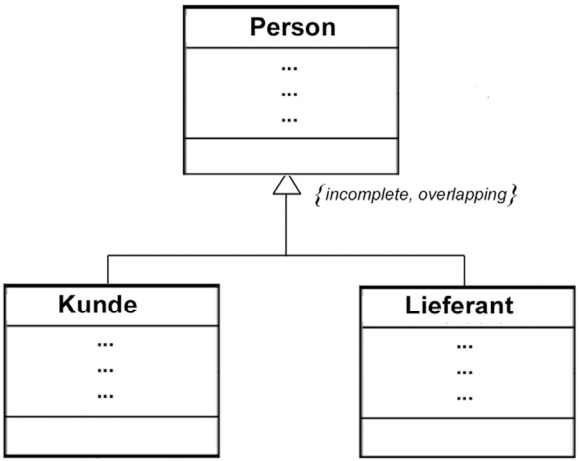


Abb. 2.8 Generalisierung (Spezialisierung, Vererbung)

Vererbungen können übrigens auch als Assoziationen dargestellt werden, in dem man zwischen der Basistabelle und jeder Subtabelle eine 1 zu 0..1 – Assoziation herstellt. „Philosophisch“ ist aber dann ein deutlicher Unterschied: Während z.B. eine Instanz von „Person“ mit der Variation „Kunde“ ein einziges Objekt liefert, entstehen bei der 1 zu 0..1 – Assoziationen 2 Objekte (die über einen Link in Beziehung zueinander stehen).

Im Rahmen der Klassendiagramme spielt ein Spezialfall der Assoziation eine große Rolle: die Aggregation. Aggregationen sind Assoziationen, von denen

man weiß, dass sie zusätzlich eine Gesamtheits-Teil-Beziehung darstellen (vgl. Abb. 2.9).

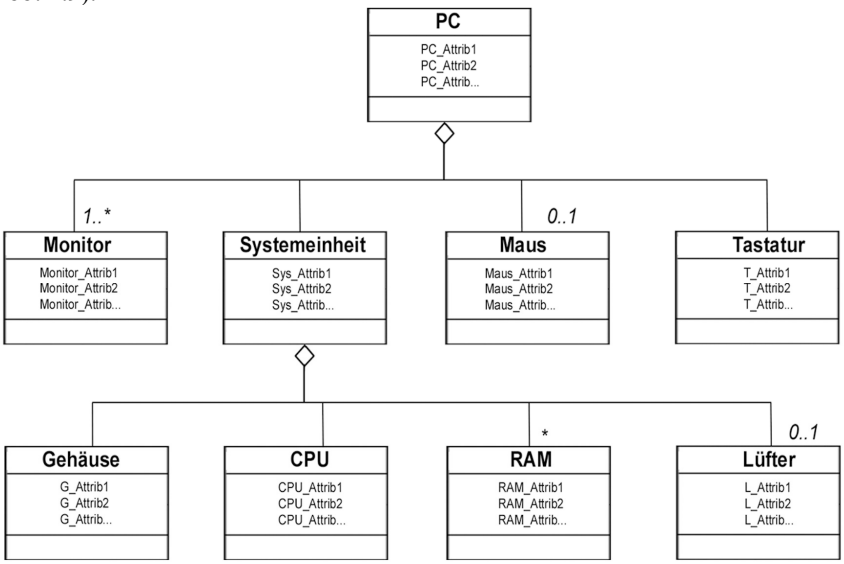


Abb. 2.9 Aggregationen

Aggregationen besitzen wieder selbst einen Spezialfall: Sogenannten Kompositionen. Dies sind Aggregationen, welche sich auf einen „lebensnotwendigen" Teil der Gesamtheit beziehen, der also nicht weggelassen werden kann. Bei Kompositionen wird die kleine Raute dann schwarz ausgefüllt.
Jetzt seien noch weitere wichtige Spezialfälle für Beziehungen unter Klassen aufgeführt.

Gerichtete Assoziation
Soll eine Assoziation immer nur in eine Richtung gehen (was speziell bei der Implementierung eine Rolle spielt), so kann man die Verbindungslinie mit einem offenen Pfeil versehen. In der späteren Implementierung drückt diese Richtung die Navigationsrichtung aus (vgl. Abb. 2.10).

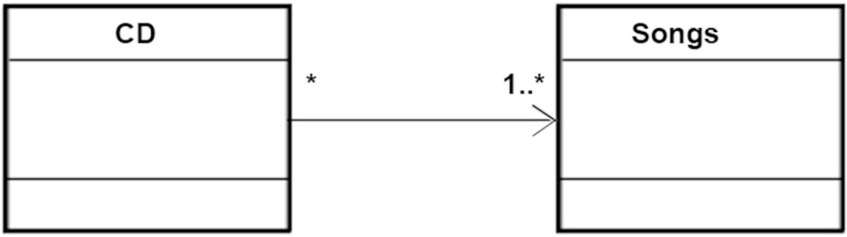


Abb. 2.10 Gerichtete Assoziation

In diesem Beispiel wird davon ausgegangen, dass in der späteren Implementierung des CD-Verwaltungsprogramms ein Song immer nur über die CD gesucht wird.

Abhängige Klassen
Eine Abhängigkeitsbeziehung in UML ist eine gerichtete Beziehung zwischen einem abhängigen (client) und einem unabhängigen Element (supplier). Beide Endpunkte können auch aus mehreren abhängigen bzw. unabhängigen Elementen bestehen. Zwischen Abhängigkeitsbeziehungen und Assoziationen gibt es einen entscheidenden Unterschied: von Abhängigkeitsbeziehungen können keine Instanzen angelegt werden. Graphisch wird eine Abhängigkeitsbeziehung als gestrichelte Linie mit einer offenen Pfeilspitze dargestellt. Die Pfeilspitze wird beim unabhängigen Element gezeichnet (siehe Abb. 2.11):

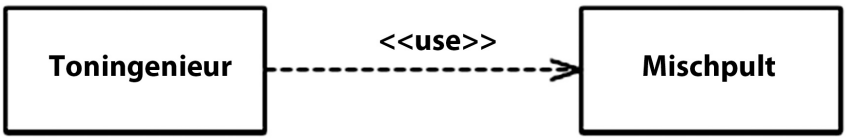


Abb. 2.11 Abhängige Klassen (hier: Toningenieur abhängig vom Mischpult)

Über der gestrichelten Linie können weitere Bezeichner stehen, welche die Art der Abhängigkeit angeben. Hier einige häufig verwendete Schlüsselwörter:

<table>
<tr><th>Schlüsselwort</th><th>Beschreibung</th></tr>
<tr><td><<use>></td><td>Das abhängige Element benutzt das unabhängige Element</td></tr>
<tr><td><<abstract>></td><td>Ein Element stellt eine Abstraktion des anderen Elements dar. Es gibt für die Abstraktion folgende Spezialfälle:
<table>
<tr><td><<derive>></td><td>weist darauf hin, dass das abhängige aus dem unabhängigen Element abgeleitet werden kann</td></tr>
<tr><td><<trace>></td><td>zeigt an, dass zwischen den beiden Modellelementen eine Abhängigkeitsbeziehung besteht, die bei Änderungen im unabhängigen Element zu berücksichtigen ist.</td></tr>
<tr><td><<refine>></td><td>weist darauf hin, dass ein Element ein anderes Element verfeinert, zum Beispiel indem es zusätzliche Details modelliert</td></tr>
</table></td></tr>
<tr><td><<include>></td><td>Die Inklusionsbeziehung bedeutet, dass ein Anwendungsfall das Verhalten eines anderen Anwendungsfalls miteinbezieht, d.h. dass der inkludierte Anwendungsfall aufgerufen wird, wenn der abhängige andere abläuft</td></tr>
<tr><td><<deploy>></td><td>Die Verteilungsbeziehung (engl. Deployment) modelliert eine Abhängigkeit zwischen einem Artefakt und einem Knoten. Sie zeigt an, dass der Artefakt auf den Knoten ausgeliefert und auf geeignete Art und Weise installiert wird</td></tr>
</table>

Realisierungsbeziehung
Die Realisierungsbeziehung (Realization) und die Schnittstellenrealisierungsbeziehung (InterfaceRealization) sind zwei weitere Ausprägungen der Abstraktionsbeziehung. Hierbei steht das unabhängige Element für eine Spezifikation und das abhängige für eine mögliche Realisierung davon. Oft modelliert diese Beziehung eine Klasse, die eine Schnittstelle implementiert:

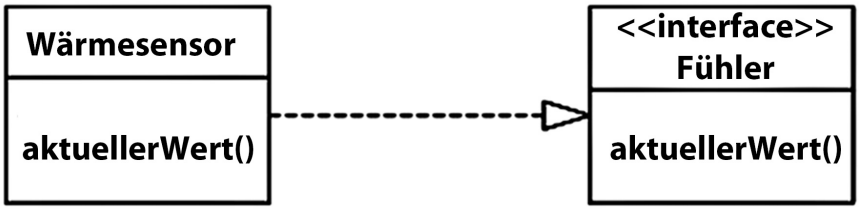


Abb. 2.12 Beispiel einer Realisierungsbeziehung

Stereotypen
In Abb. 2.12 sehen wir über dem Klassennamen „Fühler“ das Wort <<interface>>. Obwohl auch hier die spitzen Doppelklammern benutzt werden, handelt es sich hier nicht um die Beschreibung einer Abhängigkeit, sondern um einen sog. „Stereotypen“. Stereotypen stehen *innerhalb* der Klasse und sind ausgeschrieben oder können durch Symbole (oder beides) dem Klassennamen zugefügt werden. In den meisten Fällen sind die Stereotypen selbsterklärend. So bezeichnet z.B. <<component>> eine Komponetenklasse, <<entity>> eine Entitätsklasse, <<boundary>> eine Schnittstellenklasse, die das Verhalten und Aussehen von Benutzerschnittstellen kapselt, <<control>> eine Steuerklasse, die den Ablauf zwischen Anwendungsklassen kontrolliert und so weiter.
Damit seien die Klassen- und Objektdiagramme zunächst abgehandelt, sodass wir uns dem nächsten UML-Diagrammtyp zuwenden können.

Kompositionsstrukturdiagramm
In einem Kompositionsstrukturdiagramm kann man erkennen, welche innenliegenden Objekte gerade instanziiert sind und wie sie zusammenspielen. Und man sieht, welche Schnittstellen der jeweiligen Komponenten durch welche Objekte bedient werden.
Schnittstellen kann man in UML durch drei Möglichkeiten darstellen. Die Allgemeinste besteht aus einem einfachen kleinen Rechteck am Rand einer Klasse (vgl. Abb. 2.13):

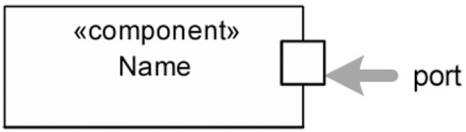


Abb. 2.13 Schnittstelle in allg. Port-Notation

Wir sehen in Abb. 2.13, dass hier der Stereotyp <<component>> benutzt wird. Komponenten sind bestimmte Klassentypen (meistens welche mit Schnittstellen). Wir gehen später im Rahmen der Komponentendiagramme hierauf noch genauer ein.

Eine zweite Möglichkeit, Schnittstellen darzustellen, sehen wir in Abb. 2.14. Dort wird zwischen den beiden Komponenten Datenbank und Oberfläche die sog. „Socket-Notation“ benutzt. Der Vollkreis bezeichnet die anbietende Schnittstelle (provided Interface), während der Halbkreis die benötigte Schnittstelle (required Interface) darstellt.

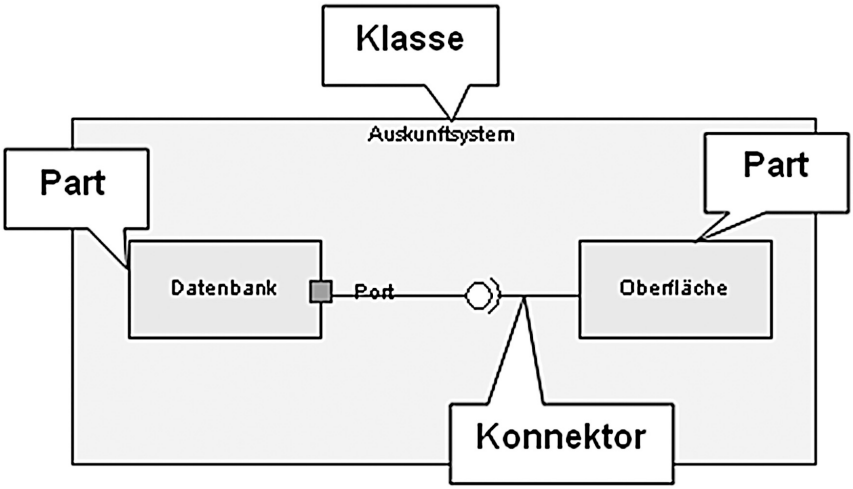


Abb. 2.14 Schnittstelle in Socket-Notation (mit Port)

Schließlich kann man Schnittstellen auch wie in Abb. 2.15 durch eine eigene Interface-Klasse repräsentieren.

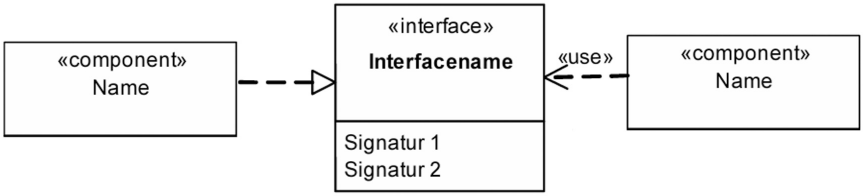


Abb. 2.15 Schnittstelle mit Interface-Klasse

Die Komponente im linken Teil des Bildes realisiert ihre Schnittstelle durch die Klasse „Interfacename“. Die rechte Komponente „benutzt“ dann diese Realisierung (die Pfeile sind also im Sinne von Abb. 2.11 und 2.12 benutzt).
Im Zusammenhang mit Kompositionsstrukturdiagrammen ist noch der Begriff des Kollaborationstyps wichtig. Ein Kollaborationstyp ist ein abstraktes Modellelement, das eine Sicht auf kooperierende Modellelemente darstellt. Er ist als solches nicht instanziierbar, sondern wird durch die einzelnen Ausprägungen der in ihm enthaltenen Modellelemente in ihrer jeweiligen Rolle repräsentiert. Kollaborationstypen beschreiben die Funktionsweise eines Systems. Sie stellen die zur Erfüllung einer gemeinsamen Aufgabe notwendige Zusammenarbeit zwi-

schen verschiedenen Einheiten dar. Abb. 2.16 zeigt hierfür die übliche Darstellung.

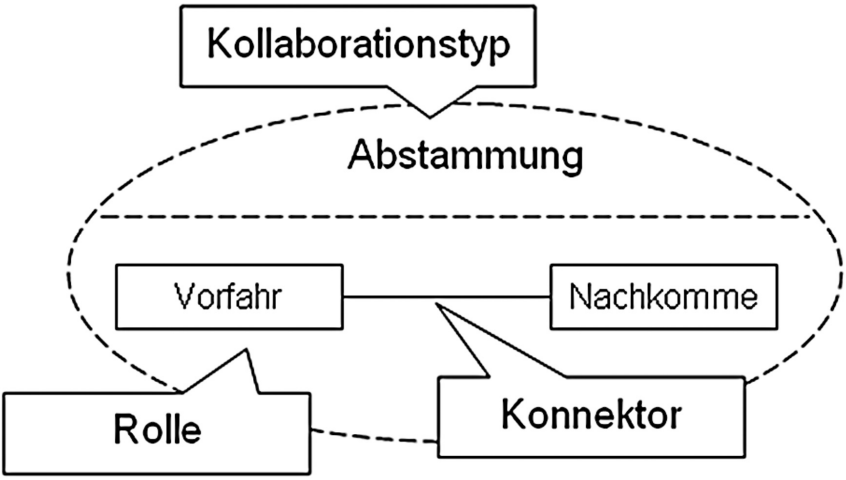


Abb. 2.16 Kollaborationen

Komponentendiagramm

Komponenten sind, wie bereits definiert, abgrenzbare Hard- oder Software-Einheiten und können eine Art „Megaklassen" darstellen. Sie können selbst eigene Attribute besitzen (z.B. im Sinne globaler Variablen) und Beziehungen eingehen. Auch können sie beliebig geschachtelt werden, so dass selbst sehr komplexe Systeme mit Tausenden von Klassen durch einige wenige Abstraktionsebenen mittels Komponenten überschaubar gemacht werden können.

Abb. 2.17 liefert ein Beispiel für ein Komponentendiagramm. Bei der Gelegenheit sei auf die besagten Komponenten-Icons hingewiesen, welche rechts neben den Stereotypen untergebracht sind. Die Komponenten repräsentieren ein komplettes System bzw. einzelne Architekturbausteine. Bei der Betrachtung des Systemkontextes werden Komponenten sowohl zur Darstellung für das zu erstellende System als auch für die Nachbarsysteme verwendet. Die Ports definieren dabei einen namentlichen Interaktionspunkt einer Komponente, der bereitgestellte und benötigte Schnittstellen zugeordnet werden können. Eine Komponente kann mehrere Ports mit gleichen Schnittstellen besitzen, d.h. die gleiche Operation kann über verschiedene Ports aufgerufen werden und wird somit intern von der Komponente unterschiedlich behandelt.

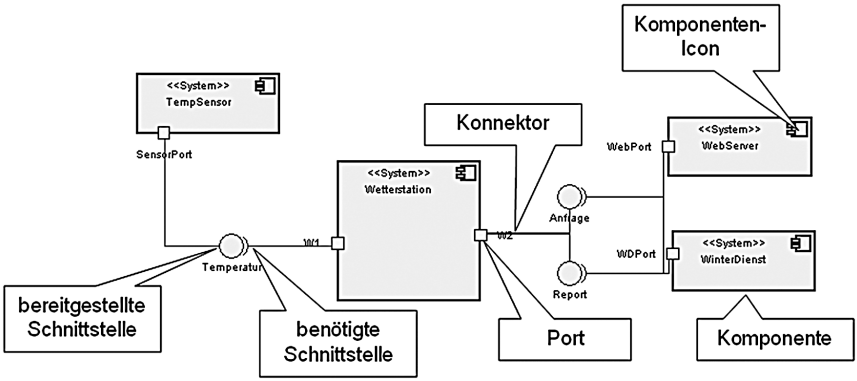


Abb. 2.17 Komponentendiagramm

Paketdiagramm
Unter einem Paket verstehen wir einfach die Zusammenfassung von Teilen eines Diagramms zu einer „Black Box“ (Abb. 2.18).

Abb. 2.18 Paket

Ein Paket kann also eine Gruppe von Klassendiagrammen oder Komponentendiagrammen oder sogar andere Paketdiagramme enthalten.

In Abb. 2.19 sehen wir u.a., dass Abhängigkeiten zwischen Paketen durch gestrichelte Linien dargestellt werden können.

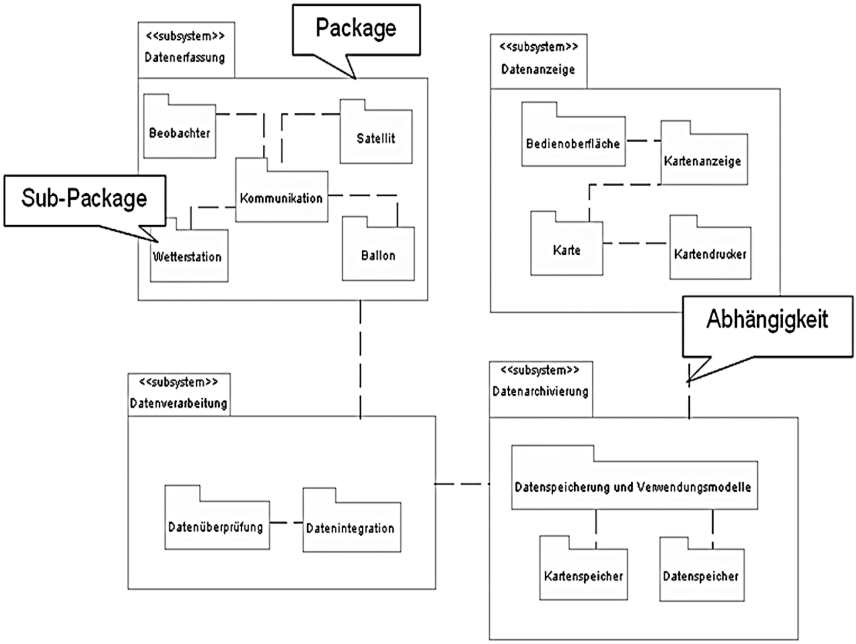


Abb. 2.19 Paketdiagramm

Verteilungsdiagramm

Beim Verteilungsdiagramm (engl. deployment diagram) wird die (überwiegend hardwarebezogene) Infrastruktur dadurch beschrieben, dass in Form sog. „Knoten" die Standorte, Cluster, Rechner, Chips oder sonstige Geräte, die etwas speichern oder „tun" können, dargestellt werden. Verbunden werden diese Knoten mit sog. „Kanälen", welche (physikalische) Übertragungswege repräsentieren, die Informationen zwischen den Knoten austauschen (siehe Abb. 2.20).

Auch hier werden die Abhängigkeiten durch gestrichelte Linien (hier mit Abhängigkeitspfeil, der in Richtung der unabhängigen Komponente weist) hervorgehoben.

Die durchgezogene Linie stellt eine Verbindung (z.B. einen Link) dar.

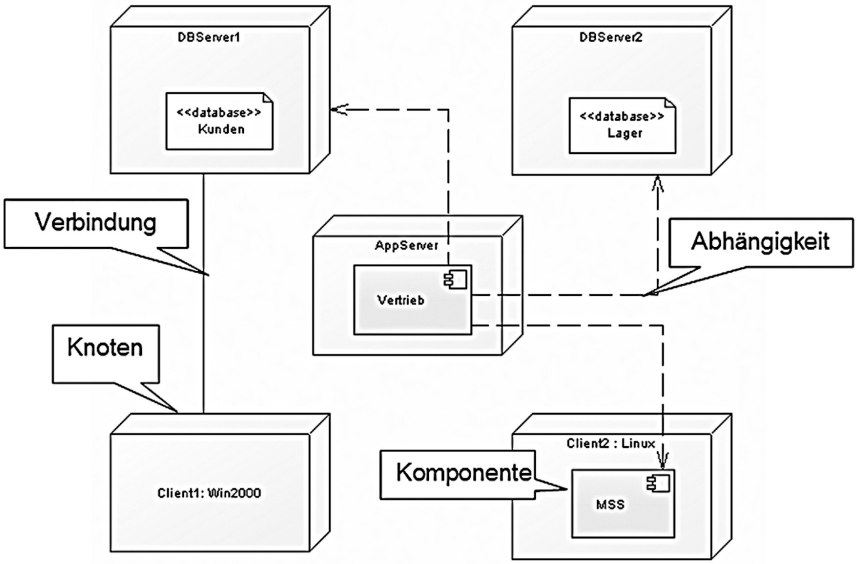


Abb. 2.20 Verteilungsdiagramm

Anwendungsfalldiagramm (Use-Case-Diagramm)
U-Case-Diagramme bestehen im Wesentlichen aus 4 Komponenten: Strichmännchen (sog. Aktuere), welche beteiligte Personen repräsentieren, Ellipsen, welche Aktionen anzeigen, Verbindungslinien, welche die beiden Letztgenannten in Verbindung bringen, sowie erläuternde Beschriftungen. Im Beispiel auf Abb. 2.21 sehen wir einen Ausschnitt der Verhältnisse für die Vertrags- und Abrechnungsbedingungen bei einer Schallplattenfirma: Ein Produzent nimmt einen Künstler (Interpreten/Performer/Artist) unter Vertrag. Der Produzent nimmt in einem Tonstudio einen Song mit dem Interpreten auf, produziert also ein „Master-Tape“ und macht sich damit dann auf die Suche nach einem Tonträgerhersteller, also der eigentlichen Plattenfirma (Record Company). Hat er diese gefunden, so wird jetzt zwischen dem Produzenten und dem Tonträgerhersteller ein weiterer Vertrag abgeschlossen, der sog. Lizenzvertrag (License Contract). Die Plattenfirma lässt daraufhin den Tonträger mit dem Song vervielfältigen und veröffentlicht diesen, in dem sie einen Distributor mit dem Vertrieb der Tonträger beauftragt. In regelmäßigem Turnus rechnet der Distributor mit der Plattenfirma über die verkauften Tonträger ab, d.h. er zahlt nach Abzug seiner Provision einen bestimmten Betrag (Payoff1) an die Plattenfirma. Diese wiederum zahlt an den Produzenten die im Lizenzvertrag vereinbarte Provision

(Payoff2). Der Produzent schließlich zahlt daraufhin an den Künstler die im Künstlervertrag vereinbarte Beteiligung (Payoff3).

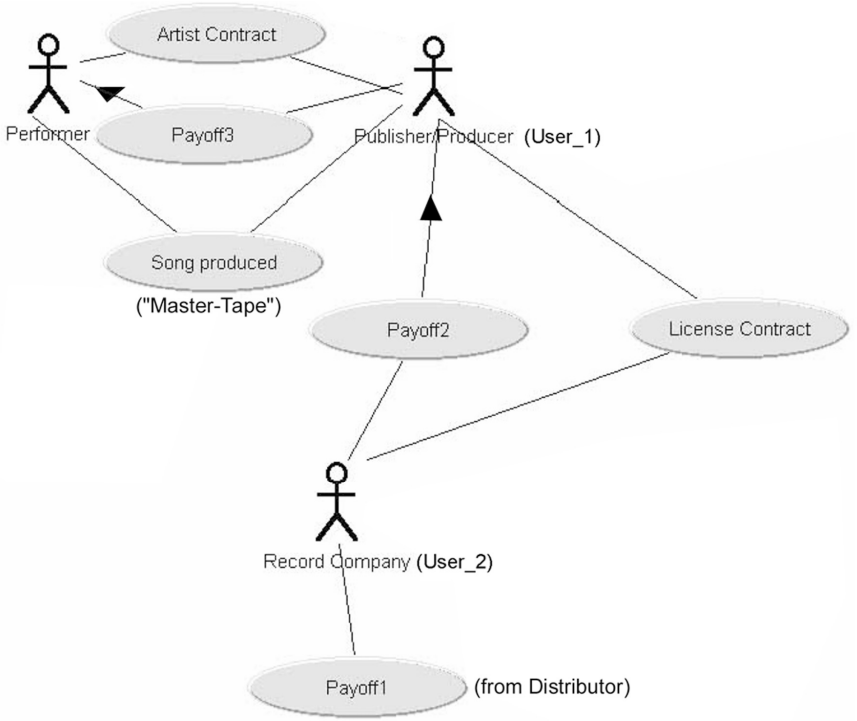


Abb. 2.21 Anwendungsfalldiagramm

Aktivitätsdiagramm

Aktivitätsdiagramme ähneln in gewisser Weise den Zustandsdiagrammen (diese werden weiter unten besprochen). Das kommt daher, dass beide Diagramme eng miteinander verknüpft sind: Die Aktivitäten lösen verschiedene Zustände des Systems aus. Der Startknoten beschreibt den Anfang des Ablaufs und der Endknoten beendet die gesamte Aktivität. Es können mehrere Endknoten für Aktivitäten existieren. Ein Verzweigungsknoten spaltet eine Kante in mehrere Alternativen auf. Durch den Verzweigungsknoten können Bedingungen festgelegt werden. Eine Aktion steht für die Beschreibung eines Verhaltens oder die Bearbeitung von Daten einer Aktion, die innerhalb einer Aktivität nicht weiter zerlegt werden. Eine Aktivität besteht aus einer Folge von Aktionen und weiteren Elementen. Der Aktivitätsfluss findet zwischen zwei Aktionen, oder einer Aktion

und einem Kontrollelement, statt. Swimlanes schließlich beschreiben die Verantwortungsbereiche für die Aktivitäten. Jede Aktivität gehört zu genau einem Verantwortlichkeitsbereich. Die Reihenfolge der Verantwortlichkeitsbereiche spielt keine Rolle. Die Beziehungen können zwischen den einzelnen Bereichen hervorgehoben werden. In Abb. 2.22 sehen wir ein Beispiel dafür.

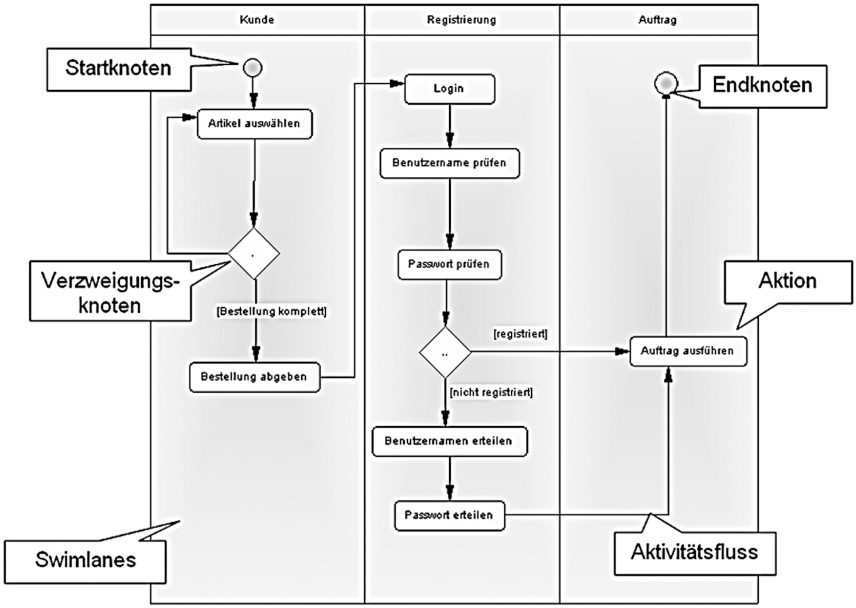


Abb. 2.22 Aktivitätsdiagramm

Zustandsdiagramm

Zustandsdiagramme geben, wie das Wort schon sagt, Auskunft über den Zustand eines Systems. Dabei werden die zu Zustandsänderungen führenden Aktionen angegeben. Es entsteht also –ähnlich wie beim Aktivitätsdiagramm- eine Art „Workflow“ oder Flussdiagramm derjenigen Vorgänge, die an den Zustandsänderungen beteiligt sind. Im Gegensatz zum später noch zu besprechenden Sequenzdiagramm wird hier der Fokus auf die Zustandsänderungen gelegt.
In Abb. 2.23 sehen wir das Beispiel eines Telefoniervorgangs.

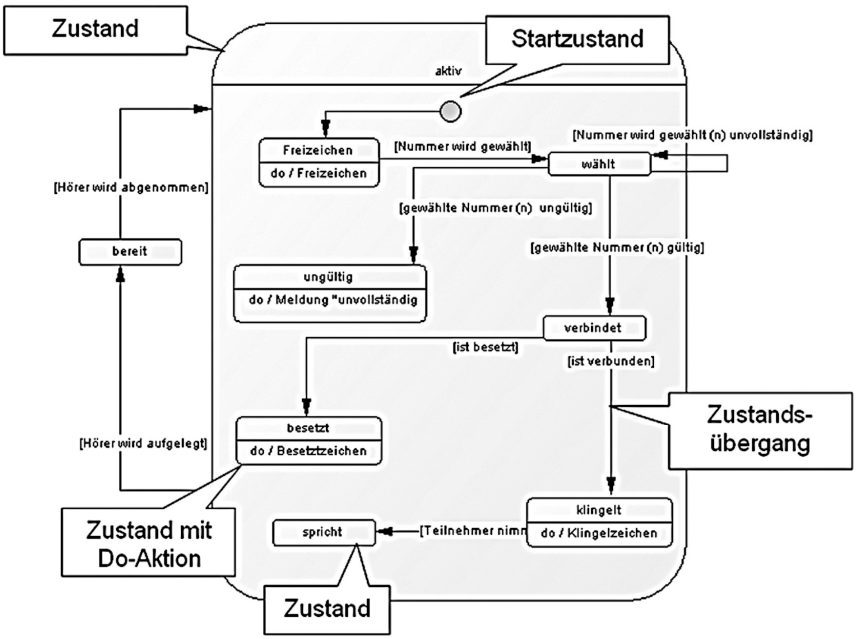


Abb. 2.23 Zustandsdiagramm für Telefonieren

Sequenzdiagramm

Sequenzdiagramme werden gerne zum zeitdynamischen Beschreiben von Vorgängen eingesetzt. Insbesondere der zeitliche Ablauf mit den dafür notwendigen Reihenfolgen von Aktionen kann daraus sehr gut abgelesen werden. Gerade zur Beschreibung wichtiger Szenarien ist diese Diagrammart sehr gut geeignet.

Im Einzelnen besteht ein Sequenzdiagramm aus folgenden Komponenten:

Die Aktivitäten der Objekte äußern sich in der Ausführung von Methoden, die durch Nachrichten eines Objekts an ein anderes aktiviert werden.

Lebenslinien repräsentieren die Objekte, die zwischen einer Interaktion Nachrichten austauschen.

Durch eine Kontrollstruktur des alt-Operators können alternative Abläufe, die durch Bedingungen versehen sind, zusammengefasst werden. Es gibt auch andere Kontrollstrukturen, wie ref, assert, break, consider, loop, opt etc.

Eine Nachricht ist die Kommunikationseinheit, die bei der Kommunikation zwischen zwei Objekten ausgetauscht wird. Es gibt drei Arten von Nachrichten: synchrone Nachrichten, asynchrone Nachrichten und Antworten.

In Abb. 2.24 sehen wir ein Beispiel für ein Sequenzdiagramm:

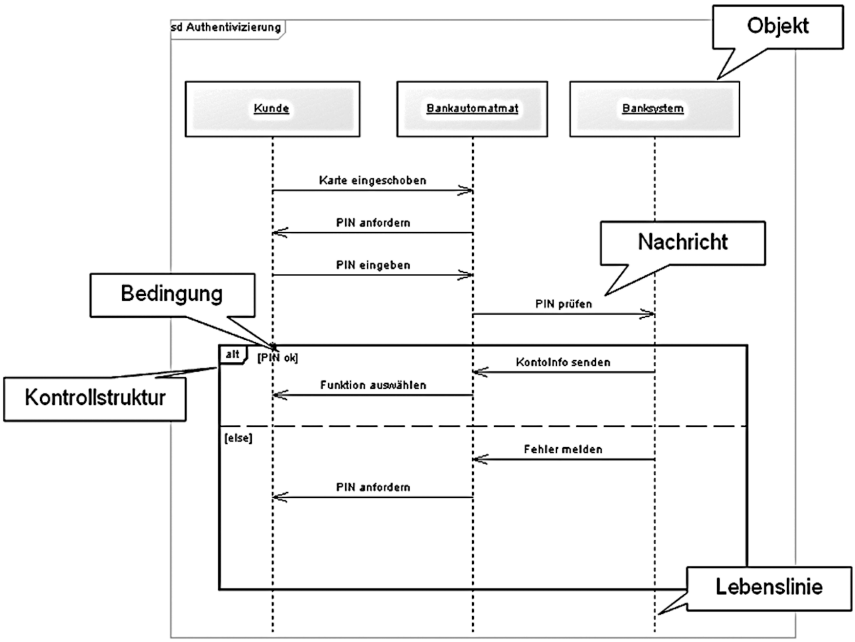


Abb. 2.24 Sequenzdiagramm für Geldautomat

Kommunikationsdiagramm

Man kann sich Kommunikationsdiagramme als instanziierte Klassendiagramme vorstellen, in die eine Folge von Nachrichten, welche von den Beteiligten ausgetauscht werden, eingezeichnet wird. Die Reihenfolge der Nachrichten kann durch eine Nummer ausgedrückt werden. Durch diese Sequenznummer werden die Reihenfolge und die Schachtelung der Verbindungen durch Methoden angegeben. Dann kann man –ähnlich wie bei den Use-Case-Diagrammen- ggf. noch Akteure einzeichnen, wenn das hilfreich erscheint.

In Abb. 2.25 sehen wir ein Beispiel eines Kommunikationsdiagramms für eine Banküberweisung.

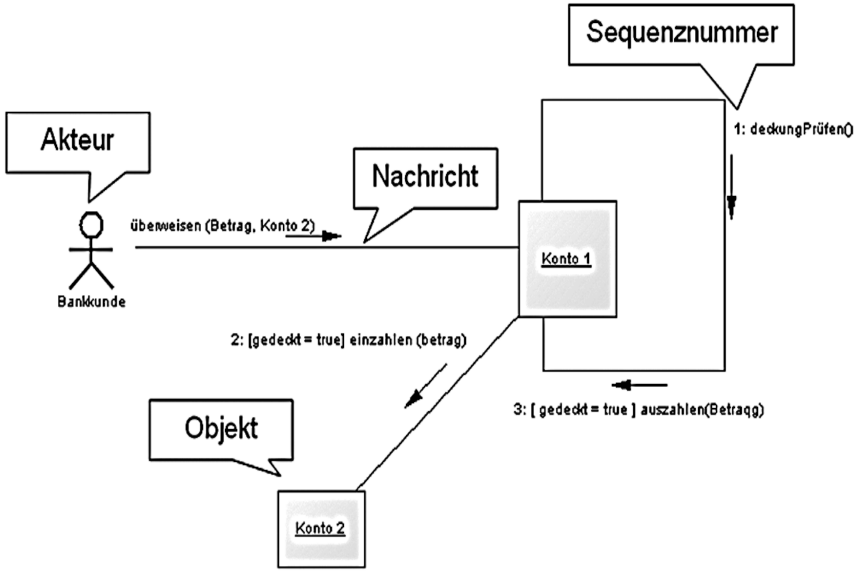


Abb. 2.25 Kommunikationsdiagramm für Geldüberweisung

Timingdiagramm

Wie das Wort schon sagt, werden hier reine Zeitverläufe zur Darstellung gebracht. Die zwei wichtigsten Elemente des Timing-Diagramms sind die Lebenslinien und die Nachrichten:

Lebenslinien zeigen Bedingungsänderungen von Objekten. Es werden Zustandsänderungen aufgrund von Ereignissen im Zeitablauf dokumentiert.

Nachrichten bewirken im Rahmen der Interaktion zwischen Interaktionspartnern Zustandsänderungen

In Abb. 2.26 sehen wir das Beispiel einer Interaktion eines Automobils mit einem Garagentor:

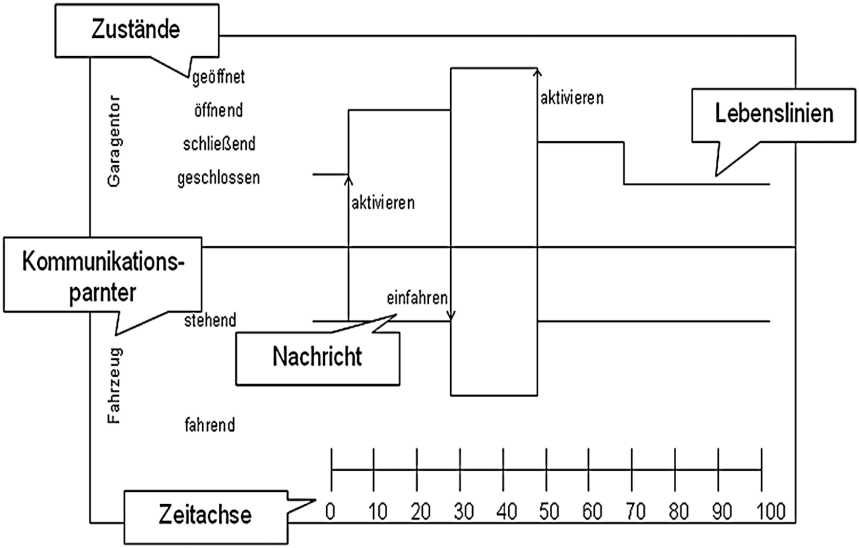


Abb. 2.26 Timing-Diagramm für eine automatisierte Garageneinfahrt

Interaktionsübersichtsdiagramm

Mit diesem Diagrammtyp (der manchmal auch Interaktionsüberblicksdiagramm genannt wird) ist es möglich, eine Art „Meta-Sicht" auf mehrere miteinander in Beziehung stehende Sequenzdiagramme oder andere Interaktionsübersichtsdiagramme zu erhalten. Die einzelnen Bestandteile des Interaktionsübersichtsdiagramms sind:

Interaktionen: Sie beschreiben ein komplettes Interaktionsdiagramm:

Start- und Endpunkt: Diese beschreiben den Einstiegs- und Ausstiegspunkt des Ablaufs.

Entscheidungsknoten: Er beschreibt ein Element zur Ablaufsteuerung der Interaktionsübersicht.

Transition: Das ist ein definierter Übergang zwischen den einzelnen Interaktionsdiagrammen und wird jeweils bei Beendigung einer Interaktion ausgeführt.

Interaktionsreferenz: beschreibt eine Referenz auf ein anderes Interaktionsdiagramm oder ein Sequenzdiagramm.

In nachfolgender Abbildung (Abb. 2.27) sehen wir wieder als Beispiel den Bankautomat. Wie man sieht, wird dort auf zwei andere Sequenzdiagramme „Geld abheben" und „Kontostand abfragen" referenziert.

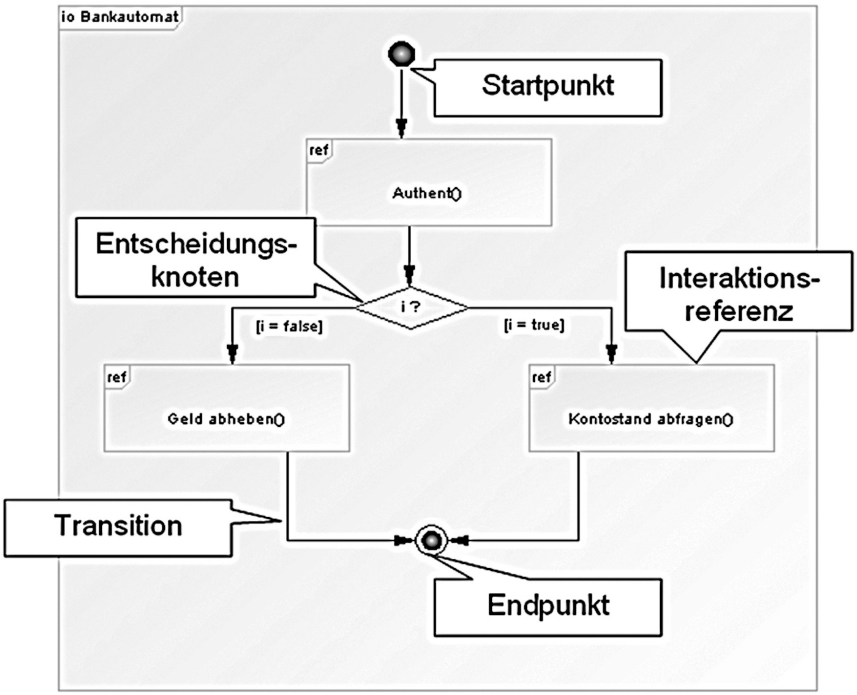


Abb. 2.27 Interaktionsübersichtsdiagramm für einen Bankautomat

Nachdem wir nun die „klassischen“ UML-Diagramme als Architekturbeschreibungssprache kennen lernten, wenden wir uns einem Repräsentanten der formalen ADL zu: Object-Z.

Object-Z als Architekturbeschreibungssprache

In den 80er Jahren des letzten Jahrhunderts wurde de formale Architekturbeschreibungssprache „Z“ eingeführt. Daraus hat sich dann später Object-Z entwickelt. Object-Z erweitert einfach nur die Syntax von Z auf objektorientierte Strukturen. Aus diesem Grund müssen wir uns zunächst mit der Syntax von Z näher beschäftigen.
Der Name Z wurde zu Ehren des Mathematikers Ernst Friedrich Zermolo (1871-1953) gewählt, welcher sich ausgiebig mit axiomatischer Mengenlehre beschäftigte. Die Z-Notation selbst wurde ab 1977 von der Universität Oxford entwi-

ckelt. Um in diese Sprache einzuführen, folgen wir einem „klassischen“ Beispiel, das erstmals von J.M. Spivey 1988 vorgestellt wurde[6].
Was zunächst etwas ungewöhnlich anmutet, sich aber später als sehr fruchtbar erweisen wird ist die Tatsache, dass die Beschreibungssprache Z auf den Axiomen der Mengenlehre sowie auf der Prädikatenlogik (1. Stufe) basiert. Es gibt dabei zwei grundsätzliche Arten von Schemata, mit denen sie arbeitet: Zustands-Schema und Operations-Schema. Das Zustands-Schema beschreibt den Zustandsraum des Problems, während das Operations-Schema Veränderungen oder Abfragen erlaubt.
Ein Schema in Z besteht wiederum aus 3 Teilen: dem Schema-Namen, dem Deklarationsteil und dem Prädikatenteil. Das Ganze ist wie folgt grafisch aufgebaut (Abb. 2.28):

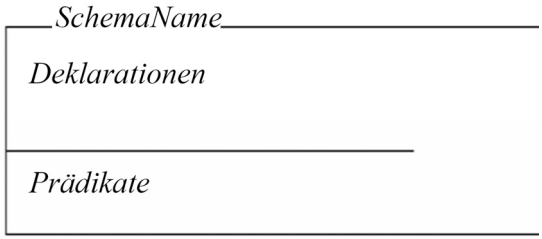


Abb. 2.28 Z-Schema

Im Deklarationsteil werden Variablen und/oder Domains (Mengen) definiert, während im Prädikatenteil die Regeln und Beziehungen für die deklarierten Variablen und/oder Mengen angegeben werden. Prädikate sind bekanntlich Funktionen oder Relationen, deren Rückgabewerte nur *wahr* oder *falsch* sein können, und selbstverständlich wird im Prädikatenteil unterstellt, dass die geforderten Regeln *wahr* sein müssen.
Wie gesagt, wir wollen dies am konkreten Beispiel demonstrieren. Dazu sei ein zu entwickelndes computergestütztes „Birthday-Book“ betrachtet; ein Notizbuch also, in dem die Geburtstage von Freunden und Bekannten festgehalten sind.
Die Daten sollen gesucht und angelegt werden können. Dann soll es eine „Reminder-Funktion“ geben, die einen an die Geburtstage erinnert.

Eine Möglichkeit dies zu realisieren, könnte so aussehen (Abb. 2.29):

[6] Vgl. Spivey, J.N.: *An introduction to Z and formal specifications* in: Software Engineering Journal, Januar 1988, S. 40ff

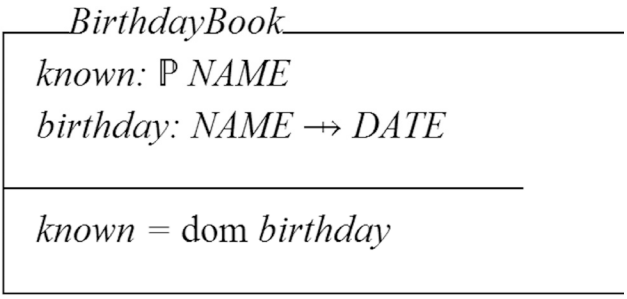


Abb. 2.29 Zustands-Schema für BirthdayBook

Im Einzelnen bedeuten:

known	Die Menge der abgespeicherten Namen
NAME	Der (abstrakte) Datentyp *NAME* (kann z.B. *char(30)* sein)
:	Der Doppelpunkt heißt hier: „…ist aus…“
ℙ	Deutet an, dass es sich hier um eine Menge handelt (also ℙ *Name* bedeutet: „Menge der Namen“)
birthday	Eine Funktion, welche zu einem Namen das Geburtsdatum zurückliefert
DATE	Datentyp DATUM
⇸	Zuordnung für Mengen
dom	Wertebereich (Domain); hier: der Urbildbereich der Abbildungsfunktion *birthday*, d.h. die Menge aller Namen, welche auf die Funktion *birthday* anwendbar sind)

Im Zustands-Schema sind alle Angaben allgemeingültig, also **invariant**. Dies spielt bei mathematischen Beweisen eine wichtige Rolle.
Verbal ausgedrückt teilt uns dieses Schema also folgendes mit: Es gibt ein Zustandsschema mit Namen *BirthdayBook*. Dort wird zunächst eine Menge *knwon* deklariert, von der wir wissen, dass sie aus der Menge aller Namen ist. Dann wird eine (Mengen-)Funktion *birthday* deklariert, von der wir wissen, dass sie zur Menge aller Namen die Menge aller „passenden“ Geburtsdaten zurückliefert.

Im Prädikatenteil werden jetzt die beiden Mengen zueinander in Beziehung gesetzt: die Menge *known* setzt sich gerade aus den Namen zusammen, für die man das Geburtsdatum kennt.
Eines fällt jetzt schon auf: Aufgrund der angesprochenen Invarianz lässt sich die Menge *known* vollständig aus der Menge *birthday* herleiten. Dies macht *known* eigentlich überflüssig. Dass dies trotzdem hier eingeführt wird, hat etwas mit der Übersichtlichkeit zu tun. Und für die spätere praktische Implementierung. Hier noch ein Beispiel für die Mengen:
known = {Karl, Schorsch, Hugo}
birthday = {Karl → 25 Jan, Schorch → 13 Feb, Hugo → 29 März}
Geläufiger ist die in die Datenverarbeitung eine Schreibweise als Tupel-Menge:
birthday = {(Karl, 25_Jan), (Schorch, 13_Feb), (Hugo, 29_März)}
Stattdessen kann man auch einfach eine Tabelle *birthday* mit den zwei Spalten „Name“ und „Geburtstag“ anlagen.
In der vorliegenden Form des Zustandsschemas wurden übrigens keine Formate oder ähnliches deklariert. Dies ist natürlich möglich, doch es kann auch „allgemein“ geblieben werden, was gewisse Vorteile hat, speziell am Anfang einer Entwicklungsphase.
Kommen wir nun zu den Operations-Schemata. Eine grundlegende Operation ist natürlich das Anlegen von neuen Personen nebst deren Geburtstag. In Z würde man das z.B. so machen (Abb. 2.30):

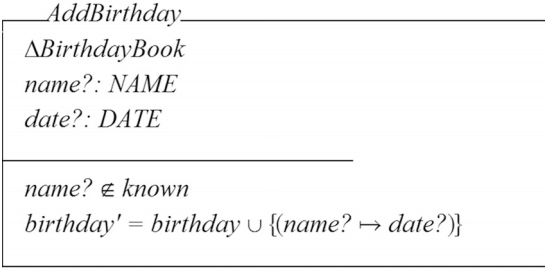


Abb. 2.30 Operations-Schema für BirthdayBook

Der Name des Schemas ist also *AddBirthday*. Das Delta-Symbol (Δ) vor *BirthdayBook* im Deklarationsteil soll andeuten, dass hier eine Änderung des Zustands dieses Schemas stattfindet. Solche Operations-Schemata sind per Definition zustandsändernd (da jede Operation ja gerade dafür vorgesehen ist). Dann folgen zwei Variablen, nämlich *name* und *date* (die auch z.B. *x* und *y* heißen könnten). Das Fragezeichen dahinter bedeutet in der Z-Konvention, dass es sich hier um eine Eingabevariable handelt. Dahinter steht wieder die Domain, der sie zugeordnet ist.

Im Prädikatenteil werden nun –wie gehabt- die Variablen durch Regeln in Beziehung zueinander gesetzt. Zunächst sehen wir, dass der eingegebene Name (noch) nicht Element der bereits vorhandenen Namen sein muss (wobei hier auch noch nicht spezifiziert ist, was passiert, wenn dem so wäre; darum kümmern wir uns später). Dann wird eine Wertzuweisung vorgenommen: eine neue Variable *brithday'* wird definiert, wobei das Hochkomma in Z bedeutet, dass diese Variable den Wert der rechten Seite neu zugewiesen bekommt. Es ist anzumerken, dass die Variable *birthday* eine Menge bezeichnet! Das erklärt, warum auf der rechten Seite auch das Vereinigungssymbol aus der Mengenlehre steht: Die „neue" Menge *brithday'* ist die Vereinigung aus der „alten" Menge *birthday* und der Zuordnung des (neuen) eingegebenen Namens nebst dessen Geburtsdatum.
Es sei erwähnt, dass (wie immer in der Prädikatenlogik) alle Regeln im Pädikatenteil erfüllt sein müssen, damit die Operation erfolgreich ausgeführt wird. So gesehen sind die einzelnen Teilprädikate „ver-undet" (also konjungiert). Ist weiter nichts spezifiziert, so würde also hier in dem Fall, dass der eingegebene Name schon vorhanden wäre, schlicht nichts passieren (der Pädikatenteil scheitert, da die erste Bedingung schon scheitert und damit die gesamte Konjunktion).
An der Stelle eine Zwischenbemerkung: Nachdem ein neuer Name nebst Geburtsdatum angelegt wurde, ist zu erwarten, dass nun die Menge *known* auch um den neuen Namen erweitert wurde, d.h. dass im Prinzip gilt:

$$known' = known \cup \{name?\}$$

Was gefühlsmäßig einleuchtet, kann auch mathematisch bewiesen werden. Es gilt nämlich:

$known' = \text{dom } birthday'$	(Invarianz hinterher)

und damit

$= \text{dom}(\text{birthday} \cup \{name? \rightarrow date?\}$	(Spezifikation von AddBirthday)
$= \text{dom birthday} \cup \text{dom } \{name? \rightarrow date?\}$	(Eigenschaft von dom)
$= \text{dom birthday} \cup \{name?\}$	(Eigenschaft von dom)
$= known \cup \{name?\}$	(Invarianz vorher)

Diese „Spielereien" sind unter Umständen von entscheidender Bedeutung: Wir haben bereits darauf hingewiesen, dass es Situationen geben kann, wo man nicht alles durchtesten kann, aber dennoch gewisses Programmverhalten garantiert werden muss. Da hilft dann „nur" der mathematische Beweis „vorab", der das gewünschte Verhalten bestätigen kann. Eine wichtige Eigenschaft, die in dem

kleinen Beweis ausgenutzt wurde, ist die Invarianz. Darauf sollte immer ein besonderes Augenmerk geworfen werden.
Weiter ausgenutzt wurden allgemeine Eigenschaften von Domains, wie z.B.
$\mathrm{dom}\,(f \cup g) = (\mathrm{dom}\, f) \cup (\mathrm{dom}\, g)$ und
$\mathrm{dom}\,\{a \rightarrow b\} = \{a\}$

Da sich viele Regeln der Z-Notation an die Prädikatenlogik und die Mengenlehre halten, stehen einem hier natürlich sofort die gesamte Palette an mathematischen Beziehungen zur Verfügung.
Kommen wir nun zur zweiten Möglichkeit der Operations-Schemata: Die „klassische“ Abfrage.
Angenommen, wir möchten nun den Geburtstag einer Person feststellen. In Z würde ein mögliches Schema so aussehen (Abb. 2.31):

FindBirthday

$\Xi BirthdayBook$
$name?: NAME$
$date!: DATE$

$name? \in known$
$date! = birthday(name?)$

Abb. 2.31 Operations-Schema für Abfrage

Das Schema *FindBirthday* hat im Deklarationsteil als erstes den Buchstaben Ξ (das große griechische „xi“) vor dem Schema-Namen *BirthdayBook* (auf den sich dieses Operations-Schema bezieht) stehen. Dies bedeutet in der Z-Konvention, dass hier eine Abfrage getätigt wird. Dieses Zeichen deutet auch an, dass damit der Zustand des Original-Schemas (also von *BirthdayBook*) diesmal nicht verändert wird (was man bei Abfragen ja erwarten würde).
Dann sehen wir ein Ausrufungszeichen hinter der Variablen *date*. In der Z-Notation wird das Ausrufungszeichen für Ausgabevariable verwendet. Somit könnte man das Schema aus Abb. 2.31 mit Worten wie folgt beschreiben:
Das Operations-Schema *FindBirthday* bezieht sich auf das Zustands-Schema *BirthdayBook*, dessen Zustand hier nicht verändert wird. Im Deklarationsteil werden zwei Variable definiert: die Eingabevariable *name* vom Datentyp *NAME* und die Ausgabevariable *date* vom Datentyp *DATE*.

Im Prädikatenteil wird gefordert, dass die Eingabevariable *name* dem System bereits bekannt sein muss (da sie als Element von *known* gefordert wird). Der Ausgabevariablen *date* wird der Rückgabewert der Funktion *birthday*, welche auf den eingegebenen Namen angewendet wird, zugewiesen.
Es sei hier angemerkt, dass im Z-Standard die Klammer bei einer Funktion auch weggelassen werden kann. So ist also in den Schemata sowohl

brithday(n)

als auch

birthday n

als Schreibweise zulässig.
Nun kommen wir zum eigentlich Sinn des zu entwickelnden Programms: Wir möchten an einem gegebenen Tag an alle Leute erinnert werden, die an diesem Tag Geburtstag haben, denn diesen Leuten wollen wir Geburtstagskarten (engl. *cards*) zumailen.
Zu diesem Zweck betrachten wir folgendes Schema (Abb. 2.32):

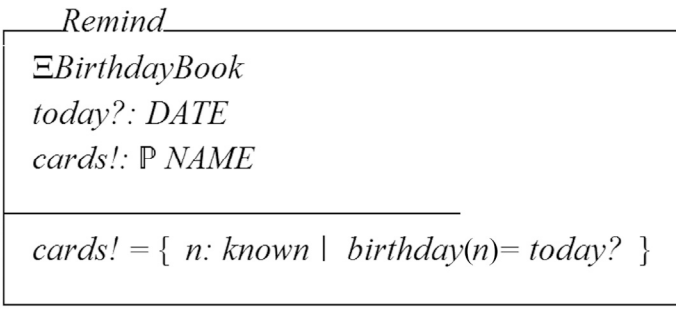


Abb. 2.32 Operations-Schema für Erinnerung an Geburtstage

Auch hier eine verbale Beschreibung des Sachverhalts:
Das Schema *Remind* bezieht sich zustandsbelassend auf das Schema *BirthdayBook* mit der Eingabevariable *today* und der Ausgabe**menge** *cards*. Die Ausgabemenge *cards* besteht aus denjenigen Elementen der Menge *known*, welche die Eigenschaft besitzen, dass der Rückgabewert der Funktion *birthday* jedes Elements (*n*, das sind die Namen) mit dem Datum *today* übereinstimmt. Also gerade das, was wir brauchen.
Die für manche vielleicht etwas ungewohnte Schreibweise der Mengeneigenschaft bedeutet eigentlich in konventioneller Schreibweise lediglich folgendes:

$$y \in \{x : S \mid beliebige\_Eigenschaft\_von(x)\}$$

$$\Leftrightarrow$$

$$y \in S \wedge beliebige\_Eigenschaft\_von(y) = wahr$$

Damit ist unsere Spezifikation fast schon fertig. Was fehlt ist nur noch eine Angabe des Startzustands (engl. initial state). Dieser könnte z.B. so aussehen (Abb. 2.33):

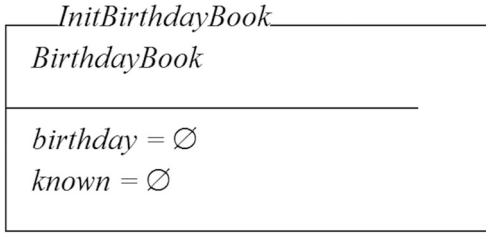


Abb. 2.33 Zustands-Schema für die Initialisierung

Dieses Schema bedarf wohl keiner Erläuterung. Es wird also mit einem „leeren" System begonnen.
Grundsätzlich haben wir hier schon eine ganze Menge Information spezifiziert. Das Grundverhalten des Systems ist beschrieben und es waren auch schon kleine math. Beweise möglich. Was mangelt, ist aber noch eine gewisse Benutzerfreundlichkeit. Wie wir sahen, passiert z.B. in dem Fall, dass man eine neue Person anlegen will, die aber schon im System vorhanden ist, einfach gar nichts. Hier wäre es natürlich wünschenswert, wenn das System irgendeine Rückmeldung wie z.B. „Name bereits vorhanden" oder ähnliches meldet.
Ein Vorteil von Z ist, dass man bereits vorhandene Schemata mit neuen logisch „kombinieren" kann. So braucht man also nicht die vorhandenen unbedingt abzuändern oder gar neu anzufangen. Diese Tatsache können wir ausnutzen, um geeignete Erfolgs- oder Misserfolgsmeldungen zu generieren. Betrachten wir folgendes Zustands-Schema (Abb. 2.34):

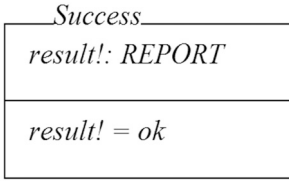


Abb. 2.34 Zustands-Schema für Erfolg

Es wird hier ein Datentyp *REPORT* vorausgesetzt (der z.B. als „Menge der Message-Boxen auf den Bildschirm" definiert sein könnte) und eine Ausgabevariable *result* definiert, welche aus dieser Menge ist. Im Pädikatenteil wird dieser Ausgabevariablen der Wert „ok" zugewiesen.

Für unsere Spezifikation könnte man jetzt einfach folgende Regel dazuschreiben:

$$AddBirthday \wedge Success$$

Damit haben wir schon mal den Fall abgehandelt, dass das Hinzufügen eines neuen Namens (und des Geburtsdatums) erfolgreich war (zur Erinnerung: eine Konjunktion ist nur wahr, wenn alle beteiligten Terme wahr sind; *Success* ist per Definition immer wahr, und *AddBirthday* nur, wenn das Hinzufügen geklappt hat). Was aber noch fehlt, sind die Fehlerfälle. Da es hier bei uns aber nur zwei Fälle gibt, nämlich den, wo der Name noch nicht im System ist (und dann ja erfolgreich hinzugefügt wird und der eben gerade abgehandelt wurde) oder den, wo der Name bereits vorhanden ist. Für letzteren spezifizieren wir jetzt das passende Schema (Abb. 2.35):

AlreadyKnown

$\Xi BirthdayBook$
$name?: NAME$
$result!: REPORT$

$name? \in known$
$result! = already\_known$

Abb. 2.35 Zustands-Schema für Fehlermeldung

Diesmal wird im Schema *AlreadyKnown* also im Pädikatenteil „geprüft“, ob der eingegebene Name bereits bekannt ist. Dann wird der Variablen *result* der Text *already_known* zugewiesen und diese ausgegeben. Damit kann man nun eine „robuste“ Version von *AddBirthday* mittels folgender Regel formulieren:

$$RAddBirthday \mathrel{\widehat{=}} (AddBirthday \wedge Success) \vee AlreadyKnown$$

Natürlich hätte man das Schema *RAddBirthday* auch von vorn herein mit den gewünschten Fällen spezifizieren können. Ein solches Schema könnte wie folgt aussehen (Abb. 2.36):

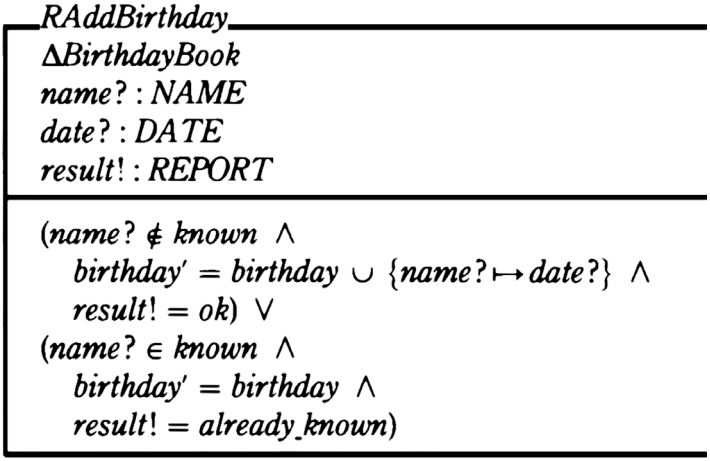


Abb. 2.36 Spezifikation für ein „robustes“ *AddBirthday*-Schema

Wie man hier sieht, sind im Sinne der Prädikatenlogik die beiden Fälle, dass der Name (noch) nicht im System ist bzw. dass er bereits dem System bekannt ist, im Prädikatenteil disjungiert. Jeder Disjunktionsterm besteht seinerseits aus geeigneten Konjunktionstermen, welche in ihrer Ausprägung gerade denen in den einzelnen Schemata von vorhin entsprechen. Aus der Prädikatenlogik kennen Sie dies, dort hat man solche Terme (Disjunktion von Konjunktionen) als DF (disjunktive Formen, genauer: unvollständige disjunktive Normalformen, DNF) bezeichnet.
Das Entsprechende kann man nun auch mit dem Schema *FindBirthday* machen. Zu diesem Zweck definieren wir ein Schema *NotKnown* (Abb. 2.37):

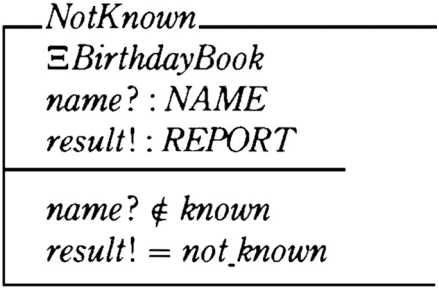


Abb. 2.37 Schema *NotKnown*

Damit kann man nun ebenfalls eine robuste Version von *FindBirthday* formulieren:

$$RFindBirthday \mathrel{\widehat{=}} (FindBirthday \wedge Success) \vee NotKnown$$

Das Schema *Remind* kann nie scheitern, denn es ist „nie“ falsch. Daher genügt es hier, für eine robuste Version davon einfach zu fordern:

$$RRemind \mathrel{\widehat{=}} Remind \vee Success$$

Nun waren wir mit unserer Spezifikation relativ abstrakt. Für eine konkrete Implementierung muss man natürlich etwas detaillierter spezifizieren. Dies wollen wir nun für unser Beispiel tun, denn dadurch lernen wir weitere Elemente von Z kennen.

Unsere bisherigen Datentypen waren nicht konkretisiert. Will man eine implementierungsnähere Spezifikation erstellen, so bedarf es hier detaillierterer Beschreibungen. Angenommen, wir benutzten Arrays für unsere Namen und Geburtsdaten in der Form

$$names\text{: array[1..] of } NAME$$
$$dates\text{: array[1..] of } DATE$$

Mathematisch modelliert man sowas durch Funktionen der Menge $\mathbb{N}_1$ der natürlichen Zahlen in die Mengen *NAME* bzw. *DATE*:

$$names: \mathbb{N}_1 \rightarrow NAME$$
$$dates: \mathbb{N}_1 \rightarrow DATE$$

Damit entspricht *names[i]* des Arrays schlicht *names(i)* der Funktion. Eine Wertzuweisung der Form

$$names[i]:=v$$

würde man in Z schreiben als:

$$names' = names \oplus \{ i \mapsto v \}$$

Die rechte Seite bedeutet hier eine Funktion, welche *names'* überall die gleichen Werte wie *names* zuweist, außer für *names(i)*, dort steht jetzt stattdessen der Wert von *v*.

Allgemein gilt in Z:

$f \oplus \{x \mapsto y\}$	bedeutet: das Ergebnis stimmt mit *f* überein außer für diejenigen *x*, die auf *y* abgebildet werden; dort wird *f* durch diese *y* ersetzt.
$f \oplus g$	bedeutet: das Ergebnis stimmt mit *f* überein außer im Wertebereich vom *g*, dort wird f durch *g* ersetzt.

Auch den Zustandsraum des Programms beschreiben wir als Schema mit Z. Für praktische Zwecke führen wir hier noch zusätzlich eine Variable *hwm* ein (soll die Abkürzung für „Hochwassermarke“ bedeuten). Diese Variable stellt die aktuelle Anzahl belegter Array-Elemente dar (Abb. 2.38):

*BirthdayBook*1

$$names : \mathbb{N}_1 \rightarrow NAME$$
$$dates : \mathbb{N}_1 \rightarrow DATE$$
$$hwm : \mathbb{N}$$

$$\forall\, i, j : 1 \,..\, hwm \bullet i \neq j \Rightarrow names(i) \neq names(j)$$

Abb. 2.38 Schema *BirthdayBook1*

Die Menge $\mathbb{N}$ ist hier als die Menge der natürlichen Zahlen einschließlich der Null zu verstehen.
Im Pädikatenteil ist das Zeichen „•“ zu lesen ist als „…gilt…“. Damit liest sich dieser Teil also so: Für alle *i* und *j* von 1 bis *hwn* gilt: Wenn *i* ungleich *j* ist, dann ist auch *names(i)* ungleich *names(j)*.
Dieses Schema sichert, dass es keine Wiederholungen von Einträgen in *names(1)* bis *names(hwm)* gibt. Diese Forderung macht deshalb Sinn, da die Funktionion *birthday(n)* ja eine mathematische Funktion in dem Sinne darstellt, dass es zu jedem Urbild genau ein Bild gibt. Es kann zwar zu einem Bild (also einem Geburtsdatum) mehrere Namen geben, aber jeder Name hat nur ein Geburtsdatum. Jetzt muss noch sichergestellt werden, dass auch jeder Name mit einem Geburtsdatum verbunden ist (also dass keine „Lücke“ in der Funktion *birthday* entsteht). Dies erledigen wir mit dem Schema Abs (steht für Absolute Relation):

Abs

BirthdayBook
*BirthdayBook*1

$$known = \{ i : 1 \,..\, hwm \bullet names(i) \}$$
$$\forall\, i : 1 \,..\, hwm \bullet birthday(names(i)) = dates(i)$$

Abb. 2.39 Schema *BirthdayBook1*

In Abb. 2.39 sehen wir also, dass im Deklarationsteil von *Abs* die beiden Schemata *BirthdayBook* und *BirthdayBook1* angezogen werden. Im Pädikatenteil werden nun zwei Konzepte miteinander verbunden: zum einen die Variablen des abstrakten Zustands –*known* und *birthday*- und welche des konkreten Zustands: *names*, *dates* und *hwm*.

Die erste Regel im Prädikatenteil besagt, dass die Menge *known* nur aus den Namen bestehen darf, die irgendwo in *names(1),...,names(hwm)* vorkommen. Dies kann man mathematisch auch so ausdrücken:

$$n \in known \Leftrightarrow (\exists i : 1..hwm \bullet n = names(i))$$

(Verbal ausgedrückt: *n* ist Element der Menge *known* genau dann, wenn es ein *i* aus *1..hwm* gibt mit der Eigenschaft, dass *n* gleich *names(i)* ist).
Das zweite Prädikat im Prädikatenteil fordert lediglich, dass der Geburtstag von *names(i)* dem korrespondierenden Element *dates(i)* aus dem Array *dates* entspricht.
Es ist in diesen Schema-Definitionen, wie wir sie vorgenommen haben, durchaus möglich, dass mehrere konkrete Zustände den gleichen abstrakten Zustand repräsentieren können: Die Reihenfolge von Namen und Geburtstagen spielt bisher keine Rolle, solange einfach ein Name mit dem richtigen Geburtsdatum vorhanden ist. Weil die Reihenfolge nicht dazu benutzt wird, herauszufinden, welcher abstrakte Zustand durch einen konkreten repräsentiert ist, repräsentieren z.B. zwei konkrete Zustände, die sich nur dadurch unterscheiden, dass sie zwar die gleichen Namen und Geburtsdaten, doch in verschiedenen Reihgenfolge, enthalten, den gleichen abstrakten Zustand. Dies ist aber praktisch meistens nicht weiter von Bedeutung, denn überflüssige Information zu vermeiden ist in der Regel mit unverhältnismäßig großem Aufwand verbunden. Da ist es einfacher, ab und zu „Säuberungsläufe“ durchzuführen, die logische Doubletten dieser Art von Zeit zu Zeit aus den Datenbeständen rausschmeißt.
Auf der anderen Seite entspricht in unserem Beispiel jeder konkrete Zustand aber genau einem abstrakten. Es ist zwar technisch möglich, dass ein konkreter Zustand mehreren abstrakten zuordenbar ist, doch wenn das der Fall ist, sollte man sein Design überprüfen, denn so etwas sollte nicht vorkommen (z.B. aus Gründen wünschenswerter Kapselung).
Wenden wir uns als nächstes der konkreten Implementierungsspezifikation für das Hinzufügen von Einträgen in unser *BirthdayBook* zu (Abb. 2.40).
Nachdem im Prädikatenteil zuerst gewährleistet wird, dass der neue Name noch nicht existiert, wird, um einen neuen Geburtstag anzulegen, zunächst die Variable *hwm* um 1 erhöht. Danach werden Name nebst Geburtsdatum nach bereits besprochener Manier hinzugefügt.
Wir wollen jetzt nochmal kurz „philosophisch“ werden: Das eben angegebene Schema beschreibt eine Operation, welche die gleichen Ein- und Ausgaben hat wie das abstraktere Schema *AddBirthay*. Es operiert jedoch auf einem konkreten anstatt einem abstrakten Zustand.

AddBirthday1

$\Delta BirthdayBook1$
$name? : NAME$
$date? : DATE$

$\forall i : 1..hwm \bullet name? \neq names(i)$
$hwm' = hwm + 1$
$names' = names \oplus \{hwm' \mapsto name?\}$
$dates' = dates \oplus \{hwm' \mapsto date?\}$

Abb. 2.40 Schema *AddBirthday1*

Außerdem stellt das Schema *AddBirthday1* eine korrekte Implementierung von *AddBirthday* dar, und zwar aus folgenden Gründen:

1. Immer, wenn *AddBirthday* gültig in einem abstrakten Zustand ist, dann ist die Implementierung *AddBirthday1* gültig in jedem entsprechenden konkreten Zustand
2. Der Endzustand, der aus *AddBirthday1* resultiert, repräsentiert einen abstrakten Zustand, den *AddBirthday* produzieren könnte

Beweis dieser Behauptungen:

Zu 1.
Die Operation *AddBirthday* ist genau dann gültig, wenn seine Vorbedingung $name? \notin known$ erfüllt ist. In diesem Fall teilt uns das Prädikat von *Abs*

$$known=\{i:1..hwm \bullet names(i)\}$$

mit, dass *names?* kein Element von *names(i)* sein kann:

$$\forall\ i:1..hwm \bullet name? \neq names(i)$$

Das ist aber gerade die Voraussetzung von *AddBirthday1*.

Zu 2.
Siehe Übungsaufgabe Nummer 5 am Ende dieses Kapitels ☺

Für die zweite abstrakte Operation, *FindBirthday*, kann man nun auch ein Implementations-Schema als konkreten Zustand angeben (Abb. 2.41):

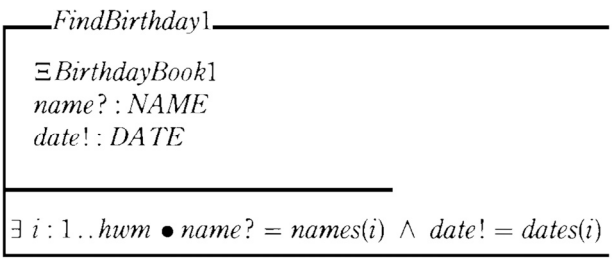


Abb. 2.41 Schema *FindBirthday1*

Object-Z

Wir haben bereits erwähnt, dass Object-Z eine Erweiterung von Z darstellt. Diese Z-Erweiterung stellt im Prinzip lediglich noch eine zusätzliche Nomenklatur für Klassen zur Verfügung. Dabei verschmelzen zum Teil die einzelnen Z-Schemata miteinander.

Auch hier wollen wir zunächst wieder ein Beispiel heranziehen[7]. Dafür spezifizieren wir den Zustand einer Queue (FirstIn/FirstOut) in „konventionellem" Z. Danach werden wir die entsprechende Object-Z Klasse spezifizieren um die Unterschiede zu demonstrieren.

Die Queue sei durch eine Variable *items* gegeben, welche die Elemente der Queue spezifiziert. Dann gibt es eine Variable *count*, welche die Gesamtanzahl an Elementen, die jemals in der Queue waren, darstellt (*count* ist also nicht der aktuelle Stand der Anzahl der sich in der Queue gerade befindlichen Elemente, sondern die Gesamtanzahl aller Elemente, die sich seit der Queue-Initialisierung insgesamt darin befunden haben; man könnte das als eine Art „Peak-Meter" auffassen).

Wir beschreiben zunächst die Queue durch folgendes Zustands-Schema und lernen bei der Gelegenheit wieder auch einiges Neue bezüglich der normalen Z-Notation (Abb. 2.42):

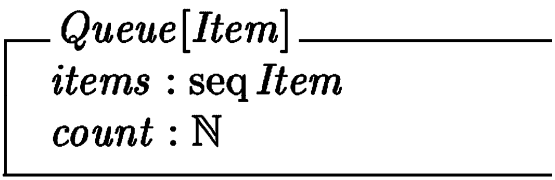


Abb. 2.42 Zustands-Schema Queue[*Item*]

[7] In Anlehnung an G. Smith, *The Object-Z Specification Language*, Kluver Academic Publishers, 2000

Wie man Abb. 2.42 sieht, kann man hinter den Schema-Namen auch den Namen einer Domain angeben, auf die sich das aktuelle Schema bezieht. Das ist dann sinnvoll, wenn man mehrere Zustands-Schemata zum gleichen Sachverhalt, aber bezogen auf jeweils andere Domains spezifizieren möchte (man muss das nicht, aber man kann es).
Dann sehen wir einen „neuen" Datentyp: seq. Dieser steht für „Sequenz" und meint, dass die Items durch eine mathematische Folge dargestellt werden. Der Zähler ist aus dem Wertebereich der natürlichen Zahlen (einschließlich der Null).
Abb. 2.43 zeigt die Initialisierung der Queue:

$$
\begin{array}{|l}
\textit{QueueInit}[\textit{Item}] \\
\textit{Queue}[\textit{Item}] \\
\hline
\textit{items} = \langle\,\rangle \\
\textit{count} = 0 \\
\hline
\end{array}
$$

Abb. 2.43 Initialisierungszustand der Queue[*Item*]

In Z werden Folgen durch spitze Klammern gekennzeichnet. Damit stellt ⟨ ⟩ eine leere Folge dar, und der Zähler beginnt bei 0.
Nun kann man auch Folgen verändern. Die Hinzunahme eines Elements ist nachfolgend dargestellt (Abb. 2.44a):

$$
\begin{array}{|l}
\textit{Join}[\textit{Item}] \\
\Delta \textit{Queue}[\textit{Item}] \\
\textit{item}? : \textit{Item} \\
\hline
\textit{items}' = \textit{items} \frown \langle \textit{item}? \rangle \\
\textit{count}' = \textit{count} + 1 \\
\hline
\end{array}
$$

Abb. 2.44a Hinzunahme eines Elementes zur Queue[*Item*]

Hier lernen wir wieder ein neues Z-Symbol kennen: Die Vereinigung zweier Folgen wird durch das Zeichen ⌢ ausgedrückt. Dieses Zeichen soll stilisiert zum Ausdruck bringen, dass hier zwei Folgen „aneinandergehängt" werden, wobei

die Folge ⟨*items?*⟩ nur aus einem, nämlich dem neuen Element besteht. Das letzte hinzugekommene Element *item?* wird also (hinten) an die Folge drangehängt. Und der Zähler wird entsprechend um eins hochgezählt.

Hier schließlich noch der Sachverhalt, dass ein Element die Queue verlässt. Da es sich hier um das FirstIn/FirstOut-Prinzip handelt, wird diesmal das „erste" anstehende Element der Queue ausgegeben, wobei der Zähler unverändert bleibt (Abb. 2.44b):

$$
\begin{array}{|l}
\hline
Leave[Item] \\
\Delta Queue[Item] \\
item! : Item \\
\hline
items = \langle item! \rangle \frown items' \\
count' = count \\
\hline
\end{array}
$$

Abb. 2.44b Ausgabe eines Elementes der Queue[*Item*]

Es mutet vielleicht etwas fremdartige an, dass jetzt, obwohl doch hier etwas aus der Folge rausgenommen wird, wieder das Zeichen ⌢ dazu verwendet wird. Das sollte man hier aber einfach so interpretieren wie man in der Mathematik z.B. eine negative Zahl „addiert": 5 + (-3) = 2. Also obwohl man addiert, kommt weniger heraus (da die addierte Zahl negativ war). Analog wird das Element *item!* ja ausgegeben und hat damit so etwas wie den Charakter eines negativen Elements. Deswegen ist die Folge hinterher kleiner als vorher, da mit einer „herauszunehmenden" Teilfolge vereinigt, nämlich mit ⟨*item!*⟩.

Z selbst bietet zunächst keine formalen Ausdrucksmittel für Klassen und Schnittstellen. Oft wird sich gerade für letzteres mit Hilfs-Operations-Schemata beholfen, die nicht selbst Bestandteil der Operations-Schemata sind. In Object-Z wird der Begriff der Schnittstelle präzisiert, in dem man in dem Schema „Klasse" gleich zu Beginn eine sog. Sichtbarkeitsliste anbringt. In dieser werden die Zustandsvariablen von etwaigen Schnittstellen repräsentiert. Diese Sichtbarkeitsliste wird durch einen stilisierten aufrechten Pfeil, der „nach außen" zeigt, symbolisiert (Abb. 2.45):

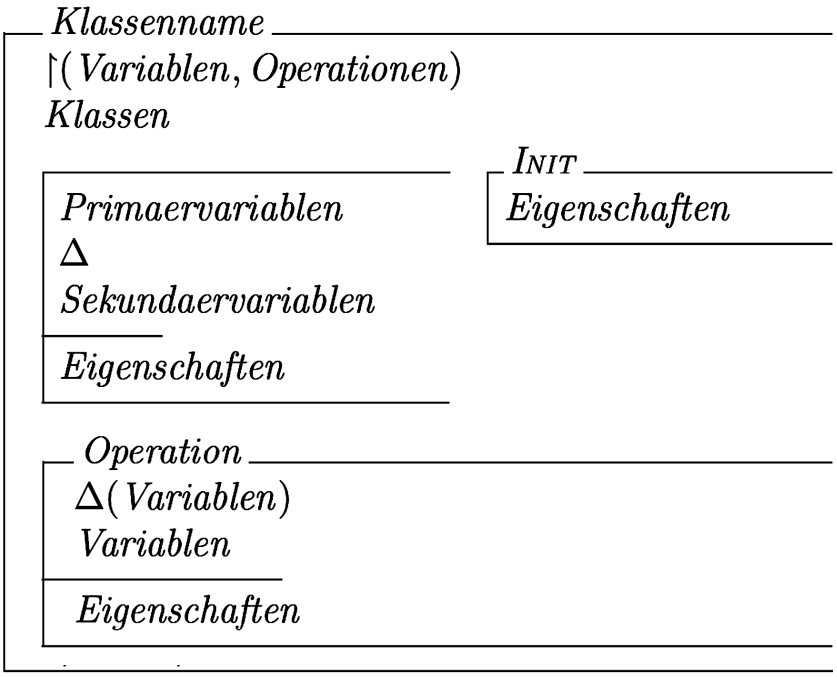


Abb. 2.45 Spezifikation einer Klasse in Object-Z

Wir wollen uns das jetzt mal am konkreten Beispiel der eben spezifizierten Queue anschauen (Abb. 2.46). Die „Schnittstelle“ der Queue umfasst hier die Zustandsvariable *count*, das initiale Zustands-Schema *INIT* sowie die Operationen *Join* und *Leave*. Im Gegensatz dazu repräsentiert die Zustandsvariable *items* Information, auf die nur „von innen“, also innerhalb des Objekts zugegriffen werden kann. In diesem Sinn liegt hier also eine Kapselung vor.
Außerdem ist diese Klasse generisch mit dem „formalen“ generischen Parameter *Item*. Er gibt den Typ des Items in der Queue an. Der Sichtbarkeitsbereich ist also hier die komplette Klasse. Deswegen braucht man hier auch nicht –wie sonst in der Standard-Z-Notation üblich und in unserer konventionellen Spezifikation zuvor auch geschehen- die formalen generischen Parameter in jeder Schemadefinition wieder neu einfügen. Abb. 2.46 zeigt uns aber noch mehr.
So führt die Sichtbarkeitsliste in der Regel alle Konstanten, Zustandsvariablen, initiale Zustands-Schemata und Operationen einer Klasse explizit auf. Man nennt diesen Teil auch manchmal den *features*-Teil der Klasse. Nun kann es vorkommen, dass alle *features* bereits Teil der Schnittstelle der Klasse sind, und dann ist die Sichtbarkeitsliste nicht unbedingt nötig. Min anderen Worten: Wenn

die Sichtbarkeitsliste fehlt, heißt dass, dass alle *features* direkt bereits in der Klasse selbst sichtbar sind.

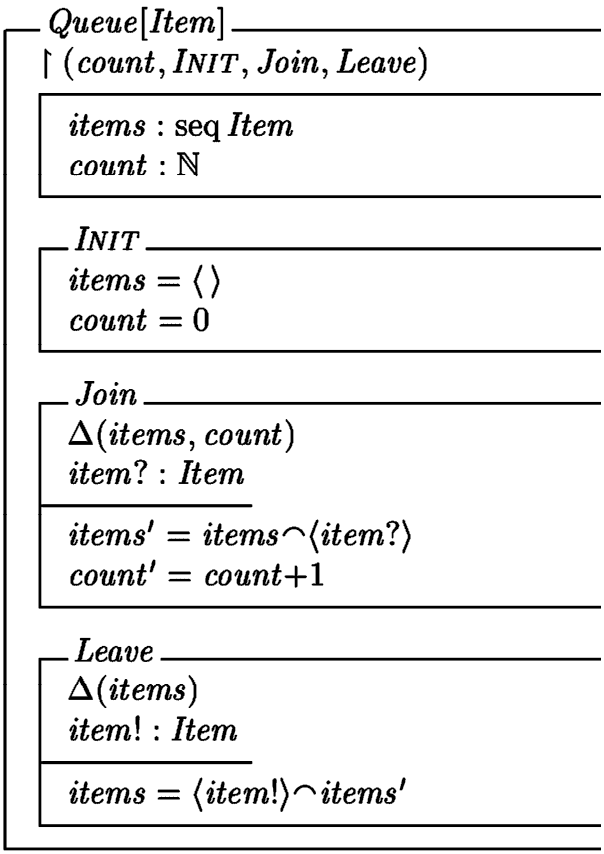


Abb. 2.46 Object-Z-Schema für das Beispiel einer Queue

In Standard-Z werden die Rollen von Schemata, wie zum Beispiel Zustandsschema, initialer Zustand oder Operation eines Systems nicht durch formale, sondern nur durch informelle Konventionen festgelegt. In Object-Z dagegen wird die Rolle eines jeden Schemas einer Klasse jetzt formal durch seinen Namen (header) angegeben.
Das erste Schema der Klasse *Queue[Item]* ist das Zustandsschema. Die Rolle als Zustandsschema wird dadurch angezeigt, dass es als einziges keinen Namen führt.

Das zweite Schema der Klasse ist das initiale Zustandsschema, was durch den Namen ***INIT*** angezeigt ist**.** Dieser Name ist ein reserviertes Wort, das zu keinem anderen Zweck in einer Spezifikation verwendet werden darf. Weil das initiale Zustandsschema nur auf ein Zustandsschema Bezug nehmen kann, wird das, was mit ihm in der Klasse eingekapselt ist, implizit in das initiale Zustandsschema mit aufgenommen und braucht daher nicht nochmal explizit einbezogen werden. Mit anderen Worten: das initiale Zustandsschema von *Queue[Item]* bezieht sich auf die Zustandsvariablen *items* und *count,* obwohl diese nicht explizit in einer Deklaration enthalten gewesen sind. Es ist sogar so, dass ein initiales Zustandsschema nie Deklarationen enthält. Es modelliert einfach die initialen Bedingungen, und diese Bedingungen sind vollständig durch seine Prädikate spezifiziert.
Alle Schemata einer Klasse, außer denen, die als Zustands- und initiales Zustandsschema ausgezeichnet sind, sind also Operationen. Das Zustand-Schema ist damit selbst implizit Teil jeder Operation. Darüberhinaus gilt das Zustandsschema in seiner "gestrichenen" Form auch als Teil jeder Operation. Das bedeutet, dass die Operationen der Klasse *Queue[Item]* auf die Variablen *items, count, items'* und *count'* Bezug nehmen können.
Eine Klassenoperation erweitert das Standard-Z-Schema, indem es eine Δ-Liste hinzufügt. Die Δ-Liste ist eine Liste von Zustandsvariablen, die von der Operation verändert werden können. Das heißt, alle Zustandsvariablen, die nicht in der Δ-Liste enthalten sind, bleiben wie sie sind. Die Operation *Join* der Klasse *Queue[Item]* verändert beide Zustandsvariablen *items* und *count,* während die Operation *Leave* nur *items* verändert. Daher ist es nicht nötig, das Prädikat *count'* **=** *count* in *Leave* zu integrieren, wie es in der Z-Spezifikation nötig gewesen ist.
Wie das Beispiel zeigt, kann die Klassennotation von Object-Z schon sehr grundlegende Z-Spezifikationen vereinfachen. Der eigentliche Vorteil von Klassen wird aber erst richtig deutlich, wenn sie zur Spezifikation von Systemen interagierender Objekte verwendet werden. Wir wollen abschließend noch ein weiteres Beispiel einer Object-Z-Spezifikation angeben. Dabei lernen wir auch wieder ein paar weitere neue Dinge bezüglich der Notation.
Sie haben bestimmt schon einmal „Tic Tac Toe" gespielt. Wenn nicht, hier kurz die Regeln:
Auf einer 3x3-Tabellen-Anordnung (vgl. Abb. 2.47) können zwei Spieler spielen. Einem Spieler ist das Kreuz, dem anderen der Kreis zugeordnet. Wer beginnt, wird ausgelost. Dann fängt z.B. der Kreis-Spieler an, irgendwo in die leere Tabelle seinen Kreis reinzusetzen. Als nächstes kommt der andere Spieler und malt ein Kreuz ein. Sinn und Zweck des Spiels ist es, als erster eine Reihe

oder Spalte oder Diagonale gleicher Zeichen (also Kreuze oder Kreise) zu erzeugen. Wem das zuerst gelingt, hat gewonnen. Jeder Spieler versucht also, dieses Ziel zu erreichen und versucht gleichzeitig zu verhindern, dass der andere es vor ihm schafft.

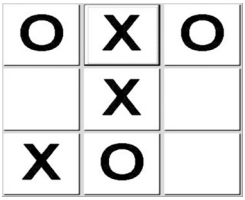


Abb. 2.47 Tic Tac Toe

So simpel wie das ganze aussieht: Es gibt immerhin 255.168 verschiedene Spielverläufe, von denen 131.184 mit einem Sieg des beginnenden Spielers enden, 77.904 mit einem Sieg des zweiten Spielers und 46.080 mit einem Unentschieden.
Wir wollen nun dieses Spiel in Object-Z spezifizieren[8]. Dazu betrachten wir (das zugegebenermaßen nicht besonders schöne) Design aus Abb. 2.48:

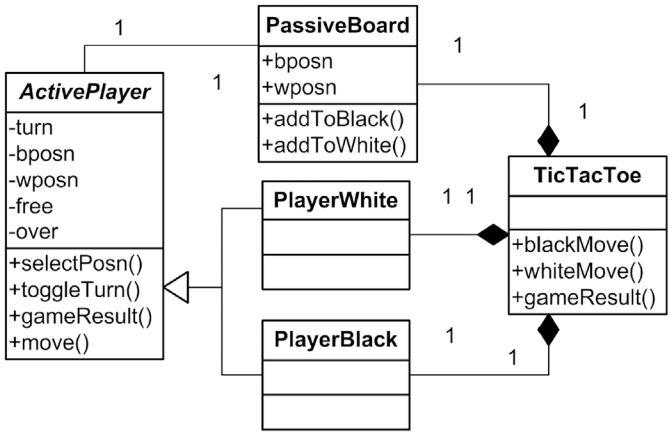


Abb. 2.48 Tic Tac Toe –Design als UML-Diagramm

Die beiden Spieler werden hier als *PlayerBlack* und *PlayerWhite* bezeichnet.
Zuerst spezifizieren wir die benutzten Konstanten und Datentypen (Abb. 2.49).

[8] Wir folgen einem Beispiel aus: Fischer, Ch., preprint des European Research Center for Information Systems, Münster

In Z (und natürlich in Object-Z) kann man das auch „jenseits“ der Schemata machen, gewissermaßen als „global Definition“, wobei man dafür auch eine „Pseudo-Schema“-Form wählen kann (z.B. um die Domain von den Eigenschaften abzugrenzen):

Typen:

$$Posn ::= 0 \,..\, 8$$
$$Result ::= black\_wins \mid white\_wins \mid draw$$

Konstanten:

$$inLine : \mathbb{P}\, Posn \rightarrow \mathbb{B}$$
$$\forall\, ps : \mathbb{P}\, Posn \bullet inLine(ps) \Leftrightarrow$$
$$\exists\, s : \{\{0,1,2\},\{3,4,5\},\{6,7,8\},\{0,3,6\},$$
$$\{1,4,7\},\{2,5,8\},\{0,4,8\},\{2,4,6\}\} \bullet s \subseteq ps$$

Abb. 2.49 Tic Tac Toe Konstanten und Typen

In Abb. 2.49 werden als die Funktion *inLine* sowie deren Domain definiert.
Als nächstes schauen wir uns eine mögliche Spezifikation der Klasse *PassiveBoard* an (Abb. 2.50):

PassiveBoard

$$bposn, wposn : \mathbb{P}\, Posn$$

INIT

$$bposn = \varnothing \wedge wposn = \varnothing$$

addToBlack

$$\Delta(bposn)$$
$$p? : Posn$$
$$bposn' = bposn \cup \{p?\}$$

addToWhite

$$\Delta(wposn)$$
$$p? : Posn$$
$$wposn' = wposn \cup \{p?\}$$

Abb. 2.50 Tic Tac Toe Klasse *PassiveBoard*

Wie zu sehen ist, sind die Methoden der Klasse *ActiveBoard* jeweils als Operations-Schemata spezifiziert.
Etwas komplexer wird die Spezifikation von *ActivePlayer* (Abb. 2.51):

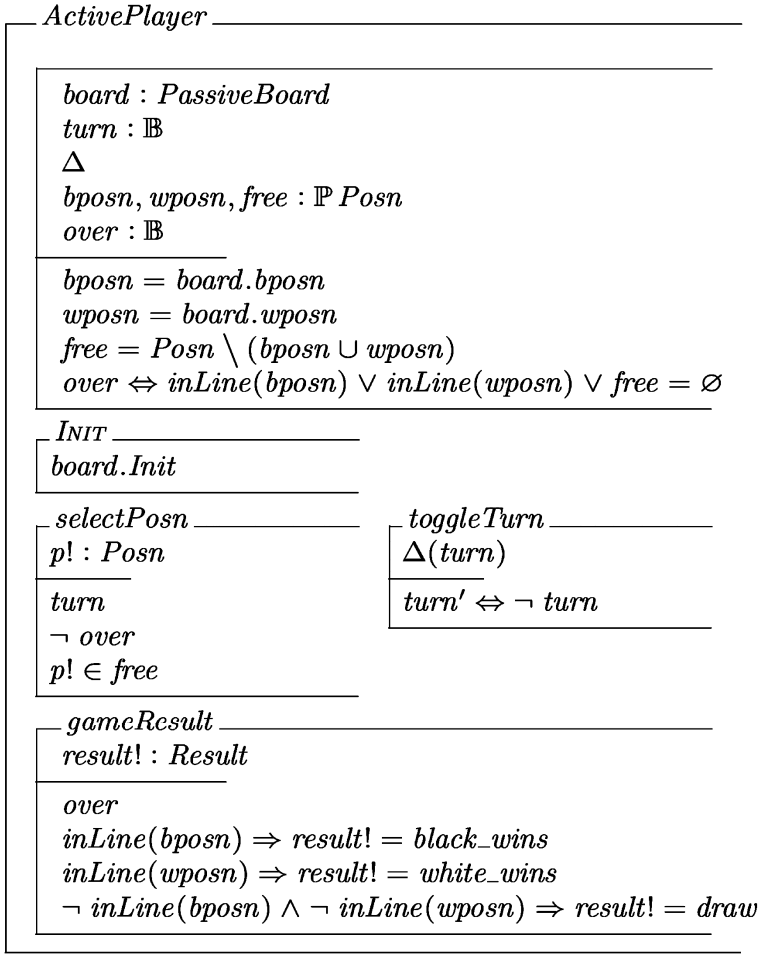


Abb. 2.51 Tic Tac Toe Klasse *ActivePlayer*

Last not Least muss noch die Klasse *TicTacToe* spezifiziert werden (Abb. 2.52):

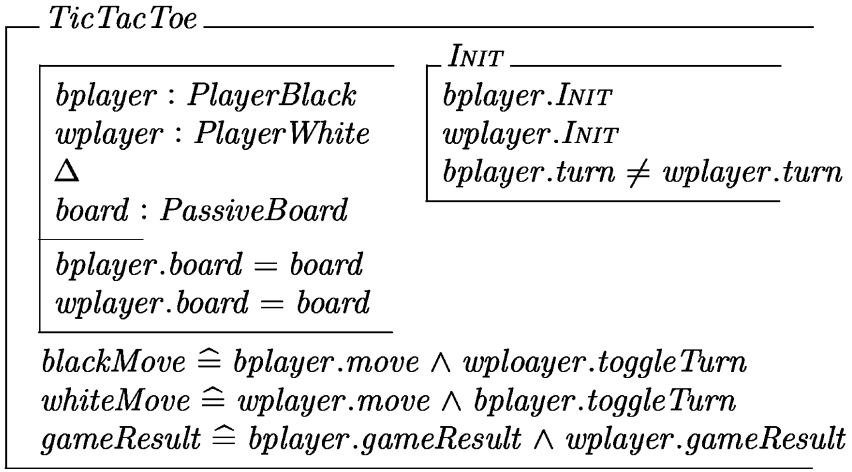


Abb. 2.52 Tic Tac Toe Klasse *TicTacToe*

Wir sehen in Abb. 2.52 einen Punkt z.B. in *bplayer.move*, was die aus objektorientierten Sprachen bekannte Methode (hier: *move*) bezeichnet, die auf das davor stehende Objekt (hier: *bplayer*) zugreift.

Auch in diesem Beispiel kann man –im Gegensatz zu einer reinen UML-Spezifikation- Korrektheitsbeweise einer Implementierung bzw. Konsistenzbeweise einer Spezifikation durchführen. Auch hierfür ein kleines Beispiel[9].

Wir wollen als Sicherheitsbeweis der Spezifikation zeigen, dass folgende Aussage richtig ist: „Wenn das Spiel beendet ist, so gibt es genau einen Gewinner oder das Spiel ist unentschieden“.

Wir gehen dabei so vor, dass wir zunächst die Objekte um notwendige Hilfsvariable erweitern (Abb. 2.53), dann das Beweisziel formalisieren und schließlich den Beweis formal zeigen.

[9] Ibid.

Abkürzungen definieren:

$$b == inLine(bposn)$$
$$w == inLine(wposn)$$

Hilfsvariablen in der Klasse $TicTacToe$ **definieren:**

$$bposn, wposn, free : \mathbb{P}\, Posn$$
$$over, drawn : \mathbb{B}$$

$$bposn = board.bposn$$
$$wposn = board.wposn$$
$$free = Posn \setminus (bposn \cup wposn)$$
$$over \Leftrightarrow inLine(bposn) \lor inLine(wposn) \lor free = \varnothing$$
$$drawn \Leftrightarrow over \land \neg\, inLine(bposn) \land \neg\, inLine(wposn)$$

Abb. 2.53 Hilfsvariablen definieren

Damit kann man das Beweisziel wie folgt formalisieren:
Im Kontext der Klasse *TicTacToe* gilt stets die Behauptung

$$over \Rightarrow (drawn \land \neg b \land \neg w) \lor (\neg drawn \land b \land \neg w) \lor (\neg drawn \land \neg b \land w)$$

Beweis:
Hilfssatz: Im Kontext der Klasse *TicTacToe* ist die Behauptung $\neg b \land \neg w$ stets wahr.
Beweis des Hilfssatzes: siehe Übungsaufgabe Nummer 7 am Ende dieses Kapitels.
Jetzt gilt: Wenn $\neg over$, dann ist die Behauptung wahr.
Wenn *over*, dann lässt sich die Behauptung auf ihre Konsequenz reduzieren und unter Anwendung des Distributivgesetzes in folgende Form bringen:

$$(drawn \land \neg b \land \neg w) \lor (\neg drawn \land ((b \land \neg w) \lor (\neg b \land w)))$$

Da *over* wahr ist, gilt nach der Definition $drawn \Leftrightarrow over \land \neg b \land \neg w$:

$$drawn \Leftrightarrow \neg b \land \neg w$$

Wenn *over* gilt, dann gilt für den Fall, dass *drawn* wahr ist: $\neg b \land \neg w$
Für den Fall $\neg drawn$ muss folgender Ausdruck wahr sein:

$$
\begin{aligned}
&(b \wedge \neg\, w) \vee (\neg\, b \wedge w) \\
&= (b \wedge \neg\, w) \vee (\neg\, b \wedge w) \vee (\neg\, b \wedge \neg\, w) \\
&= (b \wedge \neg\, w) \vee (\neg\, b \wedge (w \vee \neg\, w)) \\
&= (b \wedge \neg\, w) \vee \neg\, b \\
&= (b \vee \neg\, b) \wedge (\neg\, w \vee \neg\, b) \\
&= true
\end{aligned}
$$

Damit ist die Behauptung gezeigt.

Wir haben in diesem Abschnitt die Spezifikationssprachen Z und Object-Z etwas genauer untersucht, wobei wir bei weitem nicht den vollen Sprachumfang besprochen haben. Zudem gibt es noch eine Reihe anderer Sprachen dieser Art wie z.B. Larch und Vienna Development Method (VDM). Doch diese sind im Prinzip alle ähnlich zu Z aufgebaut.
Wir wollen uns nun noch einigen weiteren Architekturbeschreibungssprachen (wenn auch nicht mit der gleichen Ausführlichkeit) zuwenden.

Rapide[10]

Rapide hat seinen Ursprung an der Stanford University und ist eine der bekanntesten Architecture Description Languages. Diese ADL hat ihren Schwerpunkt im Prototyping großer, verteilter (Echtzeit-) Systeme.
Rapide besteht aus mehreren Teilsprachen. Zu ihrem Kern gehören eine Sprache zur Beschreibung von Komponenten-Schnittstellen (die *Type Language*) und eine Sprache zur Beschreibung von Kommunikationsmustern (die so genannte *Pattern Language*).
Komponenten werden über ihre Schnittstellen beschrieben. Die Schnittstellen definieren sowohl die exportierte Funktionalität, wie auch die benötigte Funktionalität. Abhängigkeiten zwischen Komponenten werden damit explizit gemacht. Die Schnittstellen enthalten synchrone Funktionen (*Functions*) und asynchrone Funktionen (*Actions*).
Systemabläufe werden als partiell geordnete Menge von Ereignissen beschreiben, den *Posets*. Mit der Pattern Language wird das Systemverhalten über diese Ereignismengen spezifiziert.
Analysen des Systemverhaltens werden über dessen Simulation vorgenommen. Das Verhalten wird unter verschiedenen eingehenden Ereignissen beobachtet. Rapide untersucht die Ereignisse, die vom System erzeugt werden und die Konformität zu den in der Architektur festgelegten Constraints (Einschränkungen).

[10] Das benutzte Beispiel basiert auf Ausschnitten der Dokumentationen des VSEK Projekts des Fraunhofer Instituts für Software Engineering (IESE)

Der folgende Ausschnitt spezifiziert die beiden Interface-Typen *Publisher* und *Subscriber*, beide beschreiben einen Teil eines Publisher/Subscriber-Entwurfsmusters:

```
type Subscriber is interface
requires function retrieve_info() return Data;
action in notify_subscr;

behavior
notify_subscr => retrieve_info();

constraint
match (notify_subscr > retrieve_info());
end;

type Publisher is interface
provides function retrieve_info() return Data;
action out notify;
end;
```

Der Typ *Subscriber* benötigt die Funktion *retrieve_info*, mit der er erforderliche Informationen erhält. Damit wird seine importierte Schnittstelle beschrieben. Der Typ *Publisher* exportiert die Funktion *retrieve_info*. Asynchrone Kommunikation findet über die Aktion *notify_subscr* statt. Auch hier wird exportieren (*out*) und importieren (*in*) modelliert.
Das Verhalten des Typs wird im Abschnitt *behavior* über Transitionsregeln dargestellt. Eine Regel gibt an, wie sich der Zustand als Reaktion auf Ereignisse ändert. Im Beispiel wird *retrieve_info* als Reaktion auf *notify_subscr* aufgerufen. Konsistenzbedingungen werden im Abschnitt *constraint* definiert. Im Beispiel wird festgelegt, dass *retrieve_info* erst aufgerufen werden darf, wenn *notify_subscr* davor stattgefunden hat.
Architekturen werden mit der Architecture Language aus den Typen (Interfaces) zusammengesetzt. Die *Provides*- werden mit den *Requires-Funktionen* verbunden, Out-Actions werden mit In-Actions verknüpft.
Das nachfolgende Beispiel zeigt, wie aus *Publisher* und *Subscriber* eine Architektur aufgebaut wird:

```
architecture PublishSubscribe is
P: Publisher;
S: Subscriber;

connect
S.retrieve_info to P.retrieve_info;
P.notify to S.notify_subscr;
end PublishSubscribe;
```

Große Architekturen setzen sich aus Sub-Architekturen zusammen. Eine Architektur kann dazu einem Interface zugeordnet werden.

OCL

OCL steht für Object Constraint Language (OCL). OCL erlaubt es, Ausdrücke und Bedingungen auf objektorientierte Modelle und andere Modell-Artefakte zu spezifizieren. OCL stellt eine Erweiterung des UML-Standards dar und wurde ebenfalls von der OMG eingeführt[11]. Es existieren in OCL Ausdrücke für folgende Situationen:

- Zur Spezifizierung der Initialisierungswerte von Attributen und Assoziations-Enden
- Zur Spezifikation von Ableitungsregeln für Attribute und Assoziations-Enden
- Zur Spezifikation des Inhalts einer Operation
- Zur Spezifikation einer Instanz in einem dynamischen Diagramm (z.B. Sequenzdiagramm)
- Zur Spezifikation der Bedingungen in einem dynamischen Diagramm
- Zur Spezifikation aktueller Parameterwerte in einem dynamischen Diagramm

Daneben gibt es verschiedene Sorten von sog. „Constraints“ (dt. Beschränkungen):

- **Invariants** müssen zu jeder Zeit für eine Instanz oder Assoziation gelten.
- **Preconditions/Postconditions** müssen zu dem Zeitpunkt gelten, an dem die Ausführung der zugehörigen Operation beginnt/endet.
- **Initial & derived Values** stellen Bedingungen für Ausgangs- und abgeleitete Werte dar
- **Definitionen:** es können Attribute und Operationen definiert werden, die nicht im Modell enthalten sind.
- **Body Definition** von Operationen mit isQuery = true.
- **Guards** müssen gelten, wenn ein Zustandsübergang beginnt.

Diese Constraints sind immer mit einem Kontext verbunden. Dieser ist ein Modell „Entity“, wie z. B. eine Klasse, ein Typ, ein Interface oder eine Komponente. Man unterscheidet den Kontexttyp und die Kontextinstanz. Auf letztere beziehen sich die Angaben eines Constraints.

Abb. 2.54 zeigt dafür ein Beispiel:

[11] Siehe http://www.omg.org/docs/formal/06-05-01.pdf

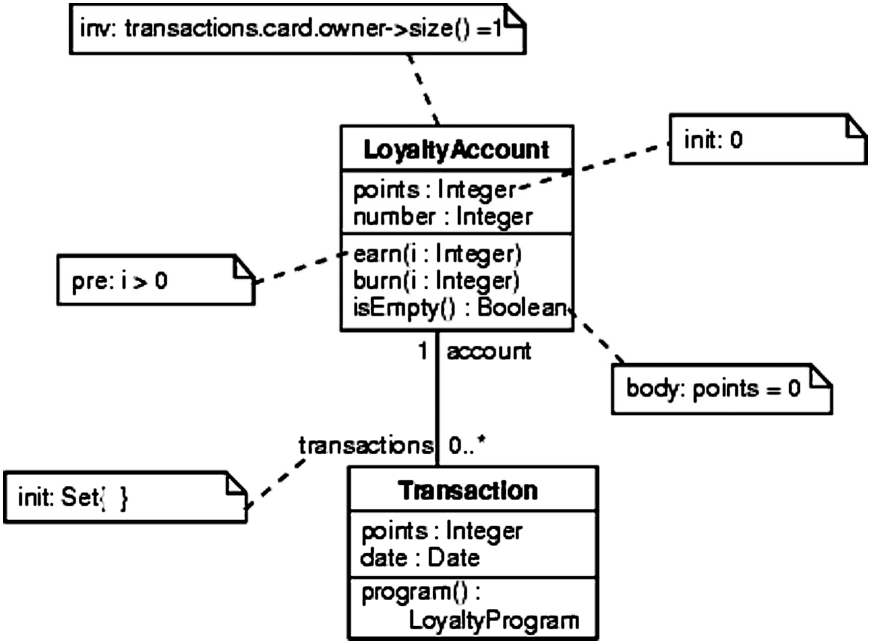


Abb. 2.54 Constraints und Kontext

Der Kontexttyp aller Ausdrücke in Abb. 2.54 ist die Klasse *LoyaltyAccount*. Die Vorbedingung (*pre: i>0*) hat als Kontext die Operation *earn*. Wenn diese evaluiert wird, dann entspricht die kontextuelle Instanz der von *LoyaltyAccount* (für welche die Operation aufgerufen wurde). Der initial value (*init: 0*) hat als Kontext das Attribut *points*. Die kontextuelle Instanz hier ist dann die Instanz von *LoyaltyAccount*, welche neu erzeugt wird.

Invarianten von Attributen
Der einfachste Constraint ist die Invariante eines Attributes. Angenommen, unser Modell enthält eine Klasse *Customer* mit einem Attribut *age*. Dann wird durch nachfolgenden Constraint das Alter einschränkt:

```
context Customer inv:
 age >= 18
```

Invarianten von Assoziationen
Häufig muss man auch Beschränkungen auf assoziierte Objekte deklarieren. Angenommen, unser Modell enthält eine Klasse *Customer* mit einer Assoziation zur Klasse *Salesperson*, wobei die Kardinalitätsbeschränkung der Rolle *salesrep* gleich 1 ist. Dann schränkt nachfolgende Bedingung den Wert eines Attributes mit Namen *knowledgelevel* der assoziierten Instanz von *Salesperson* folgendermaßen ein:

```
context Customer inv:
 salesrep.knowledgelevel >= 5
```

Objektsammlungen
Hat man Kardinalitätsbeschränkungen größer als 1 (was häufig der Fall ist), so resultiert die Instanziierung einer Klasse in eine ganze Kollektion von Objekten der Assoziierten Klasse. Constraints können nun entweder auf diese Kollektion selbst (z.B. durch Einschränkung deren Größe) oder direkt auf die Elemente selbst angesetzt werden. Angenommen, in unserem Modell hätte die Kardinalitätsbeschränkung 1..* auf der Seite der *Customer* (wieder bezogen auf *Salesperson*), so könnte man die Beziehung durch folgenden Constraint weiter einschränken:

```
context Salesperson inv:
 clients->size()<=100 and clients->forAll(c: Customer|c.age >= 40)
```

Vor- und Nachbedingungen
Hier können die Parameter der Operationen benutzt warden. Es gibt dazu ein reserviertes Schlüsselwort mit Namen *result*, welches den Rückgabewert der Operation bezeichnet. Dieses Schlüsselwort ist allerdings nur für Nachbedingungen benutzbar. Als Beispiel sei eine Operation *sell* unserem Modell zur Klasse *Salesperson* hinzugefügt.

```
context Salesperson::sell( item: Thing ): Real
 pre: self.sellableItems->includes( item )
 post: not self.sellableItems->includes(item) and result=item.price
```

Ableitungsregeln
Modelle spezifizieren häufig abgeleitete Attribute und Assoziationen. Ein abgeleitetes Attribut steht nie für sich alleine. Sein Wert muss immer aus einem anderen (Basis-)Wert aus dem Modell hergeleitet werden können. Dies kann in OCL durch sogenannte Ableitungsregeln ausgedrückt werden. In nachfolgendem Beispiel sei der Wert des abgeleiteten Elements *usedServices* definiert als alle Dienste, welche eine Transaktion auf ein Konto generiert haben:

```
context LoyaltyAccount::usedServices : Set(Services)
derive: transactions.service->asSet()
```

Initial Values
Die Initialwerte eines Attributs lassen sich ebenfalls festlegen. In nachfolgendem Beispiel seien die Initialwerte des Attributs *points* gleich 0 und für das Assoziationsende *transactions* die leere Menge:

```
context LoyaltyAccount::points : Integer
 init: 0
 context LoyaltyAccount::transactions : Set(Transaction)
 init: Set{}
```

Während eine Ableitungsregel eine Invariante darstellt, sind Initialwerte nur bei der Generierung mit ihrem entsprechenden Wert belegt.

Diese Beispiele sollten genügen, um ein Gefühl für die Aussagekraft von OCL zu bekommen.

ACME[12]

Zum Austausch von Architekturbeschreibungen haben Garlan, Monroe und Wile diese Sprache vorgeschlagen. Sie erfasst statische Informationen über die Kommunikation und deren Struktur. Verhaltensbeschreibungen sollen über andere ADLs ergänzt werden. In ACME kann man Komponenten und Konnektoren Attribute zuzuordnen. Eine Beschreibung des Verhaltens kann z.B. durch die Werte von Attributen gegeben sein.
ACME bietet eine konkrete Syntax an. Nachfolgendes Beispiel soll einen Eindruck von der Sprache ACME vermitteln. Der Quelltext definiert die Komponenten *bestellverwaltung* und *kundenverwaltung* mit entsprechenden Ports und einen (mit Rückgabewert arbeitenden) Konnektor *direkterAufruf*. Die beiden Komponenten werden mithilfe der Attachments über den Konnektor verbunden.

```
System bringdienst = {
Component bestellverwaltung = { Port sucheKundenDaten }
Component kundenverwaltung = { Port kundenInformationen }
...
Connector direkterAufruf = { Roles {caller, callee} }
Attachments : {
bestellverwaltung.sucheKundenDaten to direkterAufruf.caller;
kundenverwaltung.kundenInformationen to direkterAufruf.callee }
}
```

[12] Quelle: Beneken, G.-H.: *Logische Architekturen - Eine Theorie der Strukturen und ihre Anwendung in Dokumentation und Projektmanagement*, Dissertation, TU München, 2008

ACME konzentriert sich auf die Laufzeitarchitektur. Es betrachtet die Implementierung des ITSystems nicht explizit. Für die Entwicklung betrieblicher Informationssysteme werden jedoch beide Darstellungen benötigt.

CORBA - IDL

Die CORBA Interface Definition Language (IDL) ermöglicht die Technologieneutrale und rein deklarative Beschreibung von Schnittstellen. Aufgrund ihrer Unabhängigkeit von Programmiersprachen eignet sie sich gut für die Beschreibung von Schnittstellen zwischen Bausteinen von Software-Architekturen. Nachfolgend zeige ich Ihnen einige wichtige Merkmale der IDL.
Schnittstellen (interface) definieren Operationen (Methoden, Funktionen) und beschreiben die Parameter sowie Rückgabewerte und das Aufrufverhalten dieser Operationen. Parameter besitzen einen Modus (in, out, inout), der die Richtung der Datenübergabe definiert, und einen Typ, der den Wertebereich oder Datentyp der Ergebnisse definiert. Die IDL ermöglicht die Deklaration eigener Datentypen, unterstützt Vererbung, exakte Spezifikation der Ausnahmebehandlung von Operationen sowie synchrone und asynchrone Operationen. Hier ein Beispiel für die IDL-Beschreibung einer Schnittstelle:

```
module Geldinstitut {
     interface Konto {
           exception KeineAbhebungMoeglich {string Grund};
                void hebeGeldAb(
                    in float Betrag;
                    out float NeuerKontostand;)
                     raises KeineAbhebungMoeglich); ...
}}
```

Zur Beschreibung asynchroner Operationen eignet sich das IDL-Konstrukt *oneway*. Diese Beschreibung ist allerdings rein deklarativ und sagt nichts über Details der Implementierung aus. Wenn man weitere Eigenschaften der Kommunikation oder der Schnittstelle beschreiben möchte, so kann man in IDL Kommentare oder auch UML-Diagramme verwenden. Notizen sowie die bereits besprochene Ball/Socket-Notation stehen zur Verfügung. Ansonsten gibt es für die Realisierung asynchroner Kommunikation viele weitere technische Möglichkeiten (z.B. Message Queues, Dateien oder E-Mail).
In Abb. 2.55 sehen wir den grundsätzlichen Aufbau einer IDL-Beschreibung:

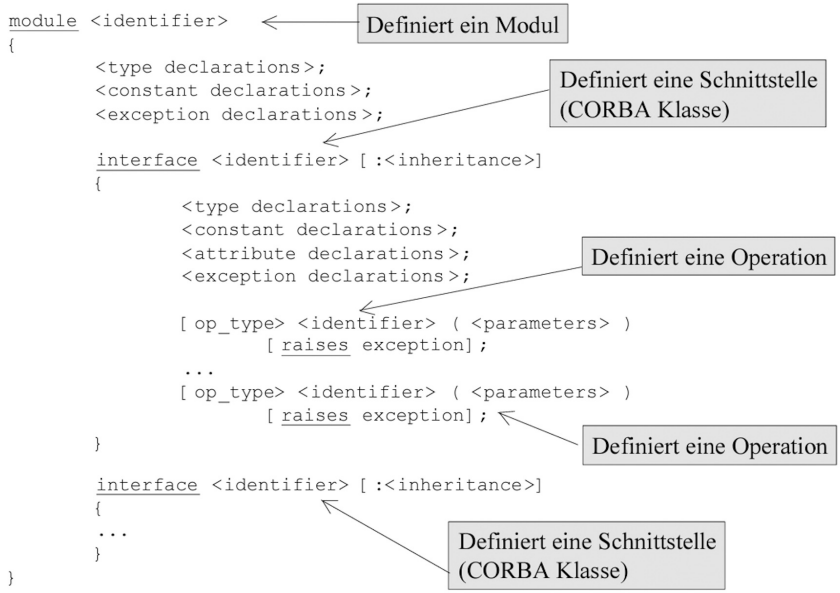


Abb. 2.55 CORBA-IDL[13]

xADL 2.0[14]

xADL 2.0 wurde von Daschofy, van der Hoek und Taylor vorgeschlagen und basiert auf XML. Es ist aufgrund seines modularen Aufbaus erweiterbar. Wie ACME bietet xADL 2.0 keine eigene Verhaltensbeschreibung an. Die Autoren wollen über die in xADL 2.0 vorgesehenen Erweiterungsmechanismen entsprechende Verhaltensbeschreibungen ergänzen (lassen).

xADL 2.0 besteht aus der „Kernsprache“ xArch32, welche grundlegende Elemente wie Komponenten und Konnektoren enthält. Es erlaubt die hierarchische Strukturierung einer Architekturbeschreibung.

Die Syntax von xArch und xADL 2.0 ist durch mehrere XML-Schemata bestimmt. Die standardisierten Erweiterungsmechanismen der XML-Schemata bilden die syntaktische Grundlage für den Erweiterungsmechanismus von xADL 2.0.

Verschiedene XML-Schemata bauen xArch zu xADL 2.0 aus: Über xArch werden Laufzeitarchitekturen spezifiziert, Beschreibungselemente sind Instanzen von Komponenten und Konnektoren. Auf xArch baut ein Schema zur Darstel-

[13] Quelle: Starke, G.: *Effektive Software-Architekturen*, Hanser, 2008, S. 102

[14] Quelle: Beneken, G.-H.: *Logische Architekturen - Eine Theorie der Strukturen und ihre Anwendung in Dokumentation und Projektmanagement*, Dissertation, TU München, 2008

lung der Architektur zur Entwurfszeit auf. Der unten dargestellte xADL 2.0 Quelltext zeigt das Beispiel einer Darstellung eines *Bringdienstes* zur Entwurfszeit. Er soll einen Eindruck von xADL 2.0 Beschreibungen vermitteln: Komponenten, Konnektoren und Interfaces (Ports, Rollen) werden über XML-Tags dargestellt, und die Schnittstellen der Komponenten und Konnektoren sind über Link-Elemente miteinander verbunden.

```
<xArch>
<archStructure>
<component id="bestellverwaltung">
       <interface id="bestellverwaltung.sucheKundenDaten">
       <direction>out</direction>
       </interface>
</component>

<component id="kundenverwaltung">
       <interface id="kundenverwaltung.kundenInformationen">
       <direction>in</direction>
       </interface>
</component>

<connector id="direkterAufruf">
       <interface id="direkterAufruf.caller">
       <direction>in</direction>
       </interface>
       <interface id="direkterAufruf.callee">
       <direction>out</direction>
       </interface>
</connector>

<link id="link1">
       <point> <anchor href="#bestellverwaltung.sucheKundenDaten"/>
       </point>
       <point> <anchor href="#direkterAufruf.caller"/>
       </point>
</link>

<link id="link2">
       <point>
       <anchor href="#kundenverwaltung.kundenInformationen"/>
       </point>
       <point> <anchor href="#direkterAufruf.callee"/>
       </point>
</link>
</archStructure>
</xArch>
```

xADL 2.0 bietet darüber hinaus ein XML-Schema zur Unterstützung des Konfigurationsmanagements an, außerdem gibt es XML-Schemata mit denen optionale Komponenten sowie verschiedene Varianten beschrieben werden können.

Diese Schemata wurden für die Entwicklung von Produktlinien-Architekturen vorgesehen. Für die Generierung von Quelltexten sind ebenfalls XML-Schemata vorhanden. Aus einer xADL 2.0- Beschreibung können damit Quelltexte in einer Programmiersprache wie Java erzeugt werden.

FODA (Czarnecki-Eisenecker)-Notation

Man kann sich streiten darüber, ob es sich bei FODA um eine Architekturbeschreibungssprache handelt, doch meiner Meinung nach passt diese Notation durchaus in dieses Kapitel.
In Hinblick auf die später noch näher zu erläuternden Software-Factories kann es Sinn machen, die Features von Software eigens etwas detaillierter zu beschreiben. Dies kann z.B. UML nicht leisten. Eine gängige Methode dafür ist die sog. FODA-Notation (FODA= Feature Oriented Domain Analysis), welche mittlerweile mehrfach erweitert wurde (Abb.2.56):

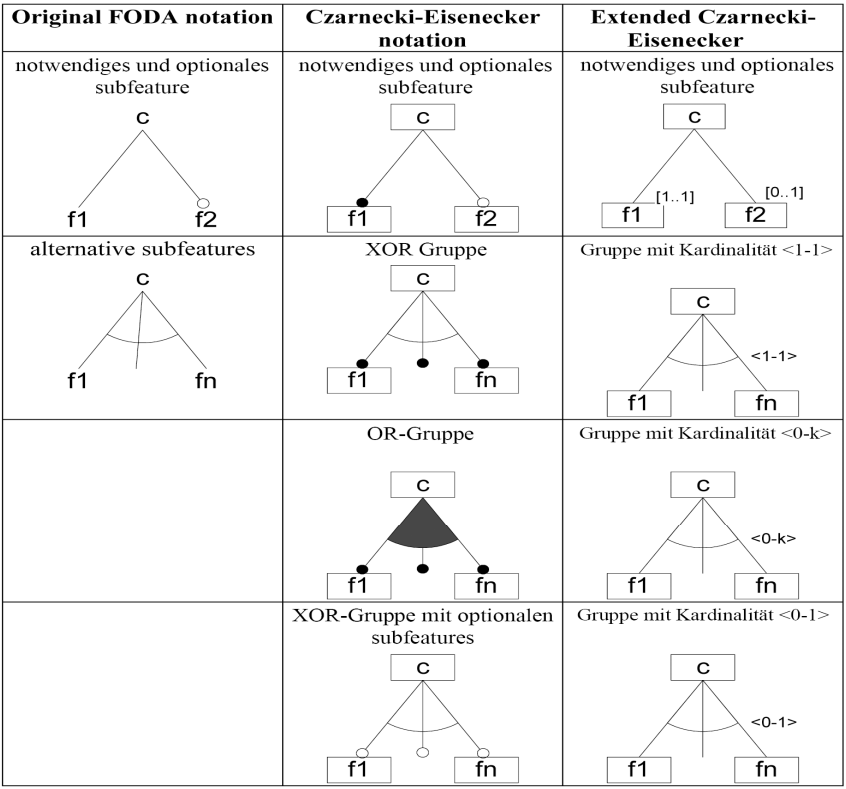


Abb. 2.56 Feature-Notationen

In Abb. 2.57 sehen wir die beispielhafte Feature-Spezifikation für das Sicherheitsprofil einer Software:

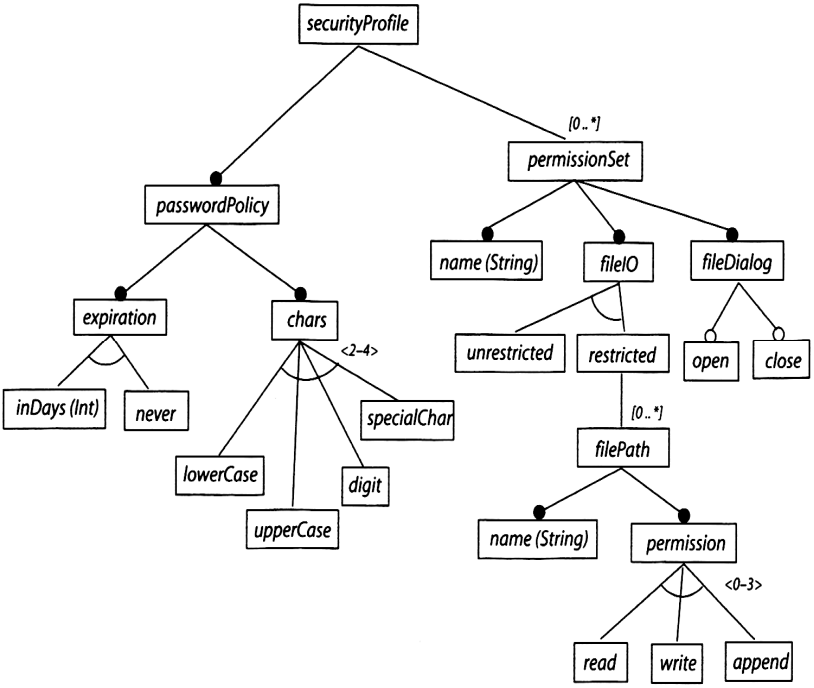


Abb. 2.57 Feature-Modell für ein Sicherheitsprofil[15]

Feature-Modelle eignen sich besonders gut z.B. zur Abgrenzung von Problemdomains.

Sonstige ADLs

Natürlich existieren neben den genannten noch viele weitere ADLs, die in diesem Rahmen nicht behandelt werden können. Nachfolgende Tabelle gibt Ihnen einen Überblick über weitere Information zu hier besprochenen bzw. weiteren ADLs:

[15] Quelle: J. Greenfield et. all, *Software-Factories*, Wiley Publishing 2004

ADL	Organisation	Federführung	URL
AADL	Society for Automotive Engineers (SAE)	Bruce Lewis	http://www.aadl.info/
ACME	Carnegie Mellon University	David Garlan	http://www.cs.cmu.edu/~acme
AESOP	Carnegie Mellon University	David Garlan	http://www.cs.cmu.edu/Web/Groups/able/able.html
CODE	University of Texas at Austin	J. C. Browne	http://www.cs.utexas.edu/users/code
ControlH & MetaH	Honeywell Technology Center	Steve Vestal	http://www.htc.honeywell.com/projects/dssa/dssa_tools.html
Demeter	Northeastern University	Karl Lieberherr	http://www.ccs.neu.edu/home/lieber/demeter.html
FR	Ohio State University	B. Chandrasekaran	http://www.cis.ohio-state.edu/lair/Projects/ARPA/arpa.html
Gestalt	Siemens Corporate Research, Inc.	Bob Schwanke	http://www.sei.cmu.edu/architecture/IWSSD8.Gestalt.paper.html
Modechart	University of Texas at Austin	Al Mok	http://www.cs.utexas.edu/users/mok/RTS/
Rapide	Stanford University	David Luckham	http://pavg.stanford.edu/rapide/
RESOLVE	Ohio State University	Bruce Weide	http://www.cis.ohio-state.edu/~weide
UniCon	Carnegie Mellon University	Mary Shaw	http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/UniCon/
Wright	Carnegie Mellon Univ.	David Garlan	http://www.cs.cmu.edu/Web/Groups/able/able.html
UML	Rational Software Corporation		http://www.rational.com/uml/index.jtmpl

In nachfolgender Abbildung[16] sehen Sie die Einsatzgebiete teilweise dieser und weiterer ADLs:

[16] Quelle: Medvidovic, N. und Taylor, N.R., *A Classification and Comparison Framework for Software Architecture Description Languages*, preprint (S.9, 12), Center for Systems and Software Engineering, University of Southern California, USA, 1999

ADL	ACME	Aesop	C2	Darwin	MetaH	Rapide	SADL	UniCon	Weaves	Wright
Focus	Architectural interchange, predominantly at the structural level	Specification of architectures in specific styles	Architectures of highly-distributed, evolvable, and dynamic systems	Architectures of highly-distributed systems whose dynamism is guided by strict formal underpinnings	Architectures in the guidance, navigation, and control (GN&C) domain	Modeling and simulation of the dynamic behavior described by an architecture	Formal refinement of architectures across levels of detail	Glue code generation for interconnecting existing components using common interaction protocols	Data-flow architectures, characterized by high-volume of data and real-time requirements on its processing	Modeling and analysis (specifically, deadlock analysis) of the dynamic behavior of concurrent systems

Features / *ADL*	Characteristics	Interface	Types	Semantics	Constraints	Evolution	Non-Functional Properties
ACME	*Component*; implementation independent	interface points are *ports*	extensible type system; parameterization enabled with templates	no support; can use other ADLs' semantic models in property lists	via interfaces only	structural subtyping via the *extends* feature	allows any attribute in property lists, but does not operate on them
Aesop	*Component*; implementation independent	interface points are *input* and *output ports*	extensible type system	(optional) style-specific languages for specifying semantics	via interfaces and semantics; stylistic invariants	behavior-preserving subtyping	allows association of arbitrary text with components
C2	*Component*; implementation independent	interface exported through top and bottom *ports*; interface elements are *provided* and *required*	extensible type system	component invariants and operation pre- and postconditions in 1st order logic	via interfaces and semantics; stylistic invariants	heterogeneous subtyping	none
Darwin	*Component*; implementation independent;	interface points are *services* (*provided* and *required*)	extensible type system; supports parameterization	π-calculus	via interfaces and semantics	none	none
MetaH	*Process*; implementation constraining	interface points are *ports*	Predefined, enumerated set of types	ControlH for modeling algorithms in the GN&C domain; implementation semantics via paths	via interfaces and semantics; modes; non-functional attributes	none	attributes needed for real-time schedulability, reliability, and security analysis
Rapide	*Interface*; implementation independent	interface points are *constituents* (*provides*, *requires*, *action*, and *service*)	extensible type system; contains a types sublanguage; supports parameterization	partially ordered event sets (posets)	via interfaces and semantics; algebraic constraints on component state; pattern constraints on event posets	inheritance (structural subtyping)	none
SADL	*Component*; implementation independent;	interface points are input and output *ports* (*iports* and *oports*)	extensible type system; allows parameterization of component signatures	none	via interfaces; stylistic invariants	subtyping by constraining supertypes; refinement via pattern maps	requires component modification
UniCon	*Component*; implementation constraining	interface points are *players*	predefined, enumerated set of types	event traces in property lists	via interfaces and semantics; attributes; restrictions on players that can be provided by component types	none	attributes for schedulability analysis
Weaves	*Tool fragments*; implementation constraining	interface points are *read* and *write ports*; interface elements are *objects*	extensible type system; types are component *sockets*	partial ordering over input and output objects	via interface and semantics	none	allows association of arbitrary, uninterpreted annotations with components
Wright	*Component*; implementation independent;	interface points are *ports; port interaction semantics specified in CSP*	extensible type system; parameterizable number of ports and computation	not the focus; allowed in CSP	protocols of interaction for each port in CSP; stylistic invariants	via different parameter instantiations	none

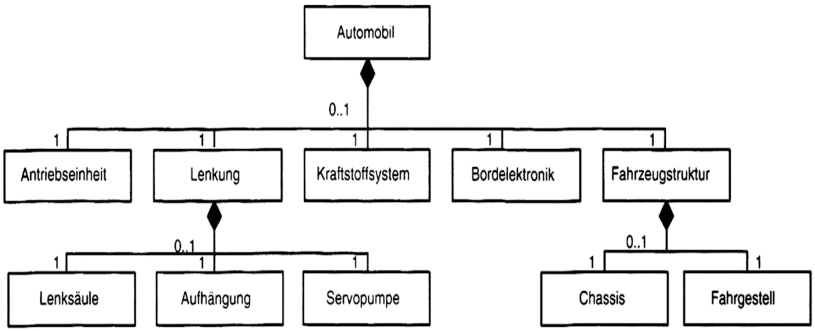
Abb. 2.58 Überblick Einsatz ADLs

Wir haben nun einige wichtige ADLs betrachtet. Diese werden wir später für die Erstellung von Architekturen brauchen. Bevor wir aber soweit sind, muss noch ein weiteres wichtiges Thema abgehandelt werden: Der Einsatz von Entwurfsmustern.

Übungsaufgaben:

1. Es sei der Sachverhalt, dass ein Gitarrenverstärker zum Spielen von Rock'n'Roll benötigt wird, gegeben. An den Gitarrenverstärker sind eine Gitarre und eine Stromzufuhr anschließbar. Erstellen Sie Schnittstellendiagramme der folgenden Form:
   a) Kompositionsstrukturdiagramm (ohne äußere Schnittstellen)
   b) Ein Komponentendiagramm für den Gitarrenverstärker
   c) Ein Schnittstellendiagramm mit Realisierungsbeziehungen und Abhängigkeiten

2. Es sei folgende Automobilklasse gegeben:



   Erstellen Sie daraus ein
   a) Kompositionsstrukturdiagramm als „Black-Box" mit den äußeren Schnittstellen Kraftstoff, Zündung, Lenkung, Abgase und Mobiltelefon
   b) Erstellen Sie aus der Black-Box von a) ein Kompositionsstrukturdiagramm als White-Box

3. Ein Backofen sei in einem Verteildiagramm (Deployment-Diagram) nebst einem sich darin befindlichen Gänsebraten (Stereotyp „Fleisch") gegeben. Eine Backanleitung für den Gänsebraten (Stereotyp „deployment spec") sieht vor, dass die Backzeit bei einer Temperator von 180 Grad bei Unter- und Oberhitze 170 Minuten beträgt. Stellen Sie dies in einem einfachen UML-Diagramm dar.

4. Wir hatten im Abschnitt über die Spezifikationssprache Z ein „robustes“ Z-Schema zur Fehlermeldung für den Fall, dass ein Geburtstagskind schon bekannt war, erstellt. Dies geschah dadurch, dass wir ein Operationsschema *AlreadyKnown* (Abb. 2.35) erstellten und dann das robuste Schema definierten durch

$$RAddBirthday \triangleq (AddBirthday \wedge Success) \vee AlreadyKnown$$

Erstellen Sie analog ein robustes Schema *RFindBirthday* (vgl. Abb. 2.31) für den Fall, dass ein Geburtstag für einen Namen gesucht wird, der dem System nicht bekannt ist.

5. Beweisen Sie mittels Z die Behauptung: Der Endzustand, der im Beispiel aus *AddBirthday1* resultiert, repräsentiert einen abstrakten Zustand, den *AddBirthday* produzieren könnte

6. Geben Sie jeweils ein Implementierungs-Schema in Z für *AbsCards*, *Remind1* und *InitBirthdayBook1* gemäß den Spezifikationen der abstrakten Zustände der entsprechenden Schemata von Seite 47 an!

7. Zeigen Sie, dass im Kontext der Klasse *TicTacToe* (vgl. Seite 58) die Behauptung $\neg b \vee \neg w$ stets wahr ist.

8. In Abschnitt über FODA hatten wir in Abb. 2.57 ein Feature-Modell für ein Sicherheitsprofil aufgezeigt. Ergänzen Sie dieses Modell um folgendes:

   a) Passwörter sollen alle 30 Tage ablaufen

   b) Passwörter müssen 3 oder 4 verschiedene Zeichenarten verwenden

   c) Ein weiteres „Permission Set“ ist einzubauen, was die I/O-Rechte und Datendialogzugriffsrechte über das Internet einschränkt.

3. Entwurfsmuster

Wenn eine Architektin ein Haus plant, so wird sie normalerweise nicht jedes Element des Bauplans immer wieder neu entwerfen; sie wird, sofern vorhanden, auf bereits bewährte und andernorts schon ebenfalls so benutzte Komponenten zurückgreifen. Dachterrassen zum Beispiel gibt es in Hülle und Fülle. Möchte ein Kunde sowas haben, wird die Architektin zuerst in ihrer Kartei nachschauen, ob sie nicht eine bereits geplante und berechnete Dachterrasse eines anderen Gebäudes verwenden kann. Manchmal kann sowas direkt übernommen werden, manchmal muss eine Vorlage mehr oder weniger abgeändert und den aktuellen Bedürfnissen angepasst werden.
Gleiches gilt im Software-Engineering. Dort werden sog. Entwurfsmuster eingesetzt, welche ähnliche Anforderung an eine Projektspezifikation bezüglich der Entwurfslösung zu formalisieren versuchen.
Ein Entwurfsmuster beschreibt eine „generische“ Lösung für ein mehr oder weniger häufig auftretendes Entwurfsproblem. Es ist damit im Prinzip eine wieder verwendbare Vorlage zur Problemlösung im DV-Entwurf. Entwurfsmuster eignen sich z.B. dafür, Wissen erfahrener Entwickler über die Struktur und das Zusammenspiel von Bausteinen und Klassen innerhalb von Softwaresystemen zu formalisieren, zu dokumentieren und weiterzugeben.
Der Einfluss von Entwurfsmustern ist häufig eher lokal begrenzt, d.h. nur innerhalb von einzelnen Bausteinen vorhanden. Entwurfsmuster können angewendet werden, ohne dass die Gesamtarchitektur davon betroffen ist. Sie eignen sich deshalb besonders für den Einsatz durch einzelne Entwickler einzelner Bausteine. Entwurfsmuster werden nach einem einheitlichen Schema dokumentiert. Hierdurch sind sie leichter anwendbar, und auch der Vergleich zwischen scheinbar ähnlichen Entwurfsmustern fällt leichter. Eine Möglichkeit sieht folgende Komponenten vor[17]:

Mustername und Klassifizierung
Der Musternamen vermittelt den wesentlichen Gehalt des Musters. Ein guter Name ist wichtig, da er Teil des Entwurfsvokabulars ist. Muster werden hinsichtlich zweier Kriterien klassifiziert: Aufgabe und Gültigkeitsbereich. Die Aufgabe gibt wieder, was das Muster macht. Muster können entweder eine *erzeugende*, eine *strukturorientierte* oder eine *verhaltensorientierte* Aufgabe haben. Der Gültigkeitsbereich legt fest, ob sich ein Muster primär auf Klassen oder auf Objekte bezieht.

[17] vgl. E. Gamma et. all: *Design Pattern,* Addison-Wesley, 1996

Zweck
Der Zweckabschnitt beantwortet die folgenden Fragen: Was macht das Entwurfsmuster? Welche spezifischen Fragestellungen oder Probleme im Entwurf behandelt es?

Motivation
Der Motivationsabschnitt besteht aus einem Szenario, welches ein Entwurfsproblem schildert und beschreibt, wie die Klassen- und Objektstrukturen des Musters das Problem lösen. Das Szenario hilft, die folgenden abstrakteren Beschreibungen des Musters leichter zu verstehen.

Anwendbarkeit
Der Anwendbarkeitsabschnitt beschreibt, in welchen Situationen das Entwurfsmuster angewendet werden kann. Es benennt die Problemsituationen, in denen das Muster helfen kann und woran diese Situationen erkannt werden können.

Struktur
Das Strukturdiagramm besteht aus einer grafischen Repräsentation der Klassen im Entwurfsmuster. Hierzu werden in erster Linie UML-Interaktionsdiagramme verwendet, um Abfolgen von Operationsaufrufen zwischen Objekten zu veranschaulichen.

Teilnehmer
Der Teilnehmerabschnitt beschreibt die am Entwurfsmuster beteiligten Klassen und Objekte sowie ihre Zuständigkeiten.

Interaktionen
Der Interaktionsabschnitt beschreibt, wie die Teilnehmer zur Erfüllung der gemeinsamen Aufgabe zusammenarbeiten.

Konsequenzen
Der Konsequenzabschnitt diskutiert, wie das Muster seine Ziele zu erreichen versucht, welche Vor- und Nachteile sich durch seine Anwendung ergeben, was für Ergebnisse zu erwarten sind und welche Aspekte der Systemstruktur voneinander unabhängig variiert werden können.

Implementierung
Der Implementierungsabschnitt präsentiert Fallen, Tipps und Techniken, deren Kenntnis für die Implementierung des Musters hilfreich sind. Darüber hinaus werden ggf. sprachspezifische Aspekte und Implementierungsmöglichkeiten benannt.

Beispielcode
Der Beispielcode diskutiert Codefragmente, die veranschaulichen, wie das Entwurfsmuster beispielsweise in C++ implementiert werden kann.

Bekannte Verwendungen
Dieser Abschnitt benennt Beispiele für das Muster, die in realen Systemen zu finden sind. Dabei werden mindestens zwei Beispiele aus unterschiedlichen Anwendungsbereichen aufgeführt.

Verwandte Muster
Der letzte Abschnitt der Musterbeschreibung setzt das Muster in Bezug zu anderen Entwurfsmustern, diskutiert die relevanten Unterschiede und erläutert, mit welchen Mustern das jeweilige Entwurfsmuster zusammen verwendet werden kann.

Generell sollte die Dokumentation des Entwurfsmusters ausreichende Informationen über das Problem, welches das Muster behandelt, über den Kontext der Anwendung und über die vorgeschlagene Lösung enthalten. Entwurfsmuster lassen sich nach verschiedenen Aufgaben und Gültigkeitsbereichen klassifizieren. Eine Möglichkeit wäre[18]:

Entwurfsmuster		***Aufgabe***		
		Erzeugungsmuster	***Strukturmuster***	***Verhaltensmuster***
Gültigkeitsbereich	***Klassenbasiert***	Fabrikmethode	Adapter	Interpreter Schablonenmethode
	Objektbasiert	Abstrakte Fabrik Erbauer Prototyp Singleton	Adapter Brücke Dekorierer Fassade Fliegengewicht Kompositum Proxy	Befehl Beobachter Besucher Iterator Momento Strategie Vermittler Zustand Zuständigkeitskette

Eine wichtige Aufgabe des Analytikers ist es jetzt, herauszufinden, welches der obigen Entwurfsmuster in einem konkreten Fall benutzt werden soll. Dabei ist es nicht so, dass man einfach nur eine „Checkliste“ durcharbeiten braucht und am Schluss kommt der gewünschte Mustertyp heraus. Es ist vielmehr so, dass man oft das gleiche Problem mit verschiedenen Mustertypen lösen könnte, nur eignen sich manchmal eben bestimmte Mustertypen für bestimmte Problemfelder bes-

[18] ibid.

ser als andere. Hinzu kommt auch, dass in der Industrie häufig aus „traditionellen Gründen“ mit bestimmten Mustertypen gearbeitet wird, relativ unabhängig vom Problemtyp, da mit diesen Mustern schon viel Erfahrung vorhanden ist.
Um den Einsatzbereich der jeweiligen Muster etwas näher zu spezifizieren, seien nachfolgend die gebräuchlichsten Typen charakterisiert. Dabei wird nach den Aufgabentypen unterschieden.

Klassenbasiert:

Fabrikmethode (FactoryMethod, Virtual Constructor)
Diese Methode dient dem Erzeugen von Objekten, von denen nur die abstrakte Basisklasse bekannt ist. Eine (bis dahin abstrakte) Anwendung benötigt ein (bis dahin abstraktes) Objekt, also etwas, das sie nicht direkt erzeugen kann (da abstrakt), obwohl sie muss. Die Anwendung benutzt dafür dann eine Methode, die ein solches Objekt erzeugt. Die spätere nicht-abstrakte Anwendung überschreibt die Methode und erzeugt das konkrete Objekt (Abb. 3.1).

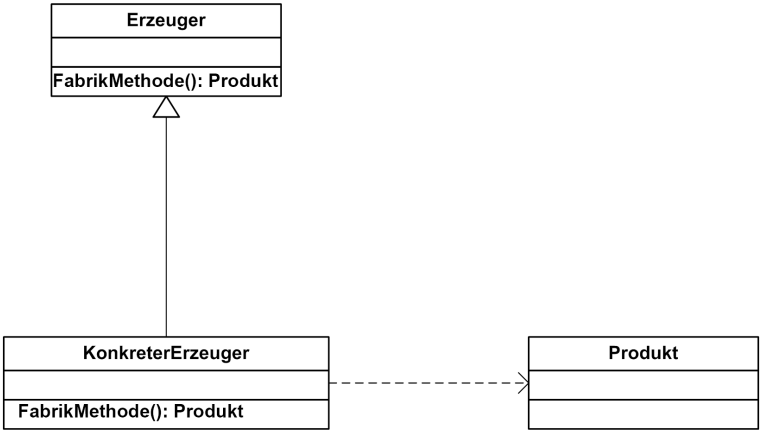


Abb. 3.1 Fabrikmuster

Objektbasiert:

Abstrakte Fabrik (Abstract Factory, ToolKit)
Hier sind die Objekte und deren Erzeugung vor dem Klient verborgen. Fabrik und Objekte werden dadurch austauschbar.
Der Klient soll gewisse Objekte benutzen, aber ihre Klassen nicht kennen. Um sie zu erzeugen, benutzt er eine Fabrik, welche eine Schnittstelle bietet, um die

(dahinter verborgenen) Objekte zu erzeugen. Durch Austausch der Fabrik können andere Objekte erzeugt werden (Abb. 3.2).

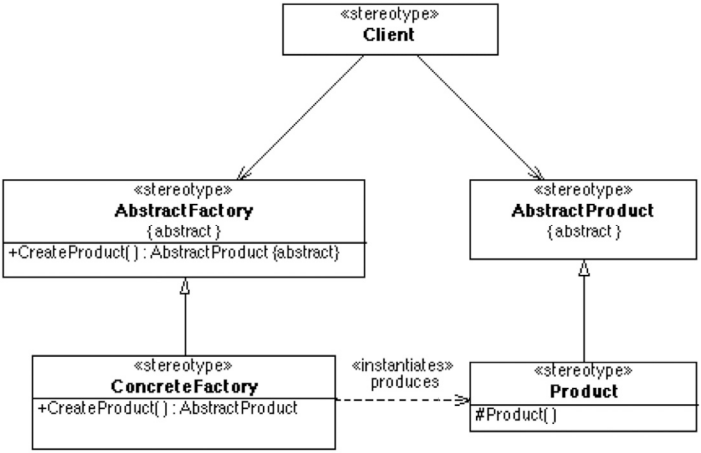


Abb. 3.2 Abstrakte Fabrik

Erbauer (Builder)

Beim Erbauermuster trennt man die Berechnungen (=Konstruktion, Director) von der Präsentation (=Builder), wodurch bei gleicher Berechnung leicht unterschiedliche Präsentationen gewählt werden können. Abb. 3.3 zeigt das zugehörige UML-Diagramm dieses Musters.

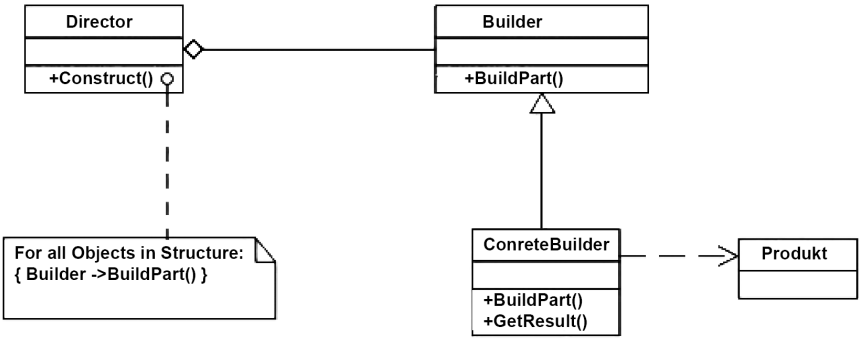


Abb. 3.3 Erbauer-Muster

Prototyp (Prototype)
Hier kreiert ein Erzeuger Objekte (z.B. mit der Clone-Funktion) eines übergebenen Objekts (Prototypen). Der Erzeuger soll also Objekte erzeugen, von denen er nur eine Basisklasse kennt (d.h. die konkrete Klasse ist unbekannt). Eine Clone-Funktion in dieser Basisklasse (Prototyp) erzeugt damit eine Kopie des Objekts (Abb. 3.4). Jeder Erzeuger erhält bei Initialisierung ein fertiges Objekt von der Klasse, von der er Objekte erzeugen soll (die Benutzung der Clone-Funktion ermöglicht dies auch ohne die Klasse zu kennen).

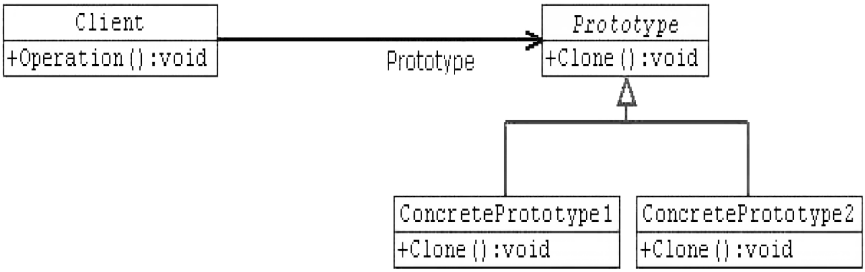


Abb. 3.4 Prototyp

Singleton
Bei diesem Muster wird garantiert, dass nur ein Objekt von diesem Typ erzeugt wird (Abb.3.5). Dazu muss der Konstruktor privat sein, ebenso muss es eine private Klassenvariable (durch ein Minuszeichen davor gekennzeichnet) des eigenen Typs geben und eine öffentliche Klassenmethode für den Zugriff (und gegebenenfalls für die Erzeugung).

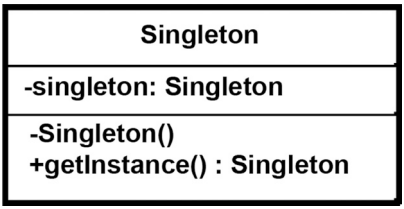


Abb. 3.5 Singleton

Klassenbasiert:

Adapter (Wrapper)
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. Damit kann man mit dem Adapter die Schnittstelle eines existierenden Moduls ändern. Das Adapter-Muster wird eingesetzt, wenn man ein existierendes Modul verwenden möchte, dessen Schnittstelle nicht mit der benötigten Schnittstelle übereinstimmt (Abb. 3.6).

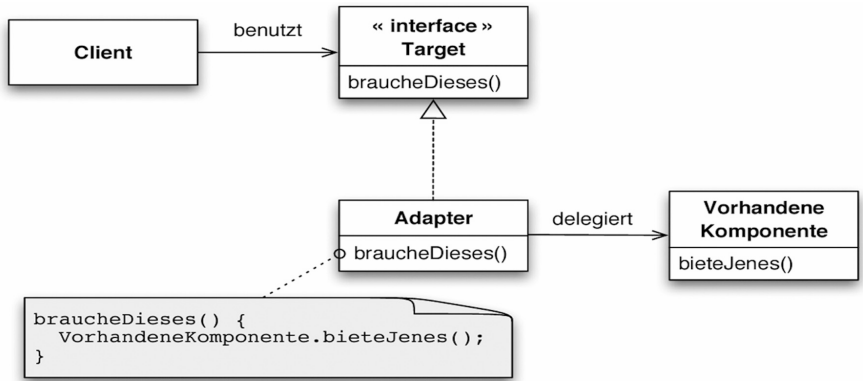


Abb. 3.6 Adapter[19]

Objektbasiert:

Adapter (Wrapper)
Wie bei „Klassenbasiert“

Brücke (Bridge, Handle, Body)
Dieses Muster trennt Abstraktion und Implementierung durch „Abstraktionshierarchie“ und „Implementierungshierarchie“.
Wenn eine Hierarchie aufgrund unterschiedlicher Implementierung auf jeder Ebene aufgespalten werden muss, macht es Sinn, Abstraktion und Implementierung zu trennen, um so eine Abstraktionshierarchie und eine Implementierungshierarchie zu erzeugen (Abb. 3.7).

[19] Quelle: Starke, G.: *Effektive Software-Architekturen*, Hanser, 2008

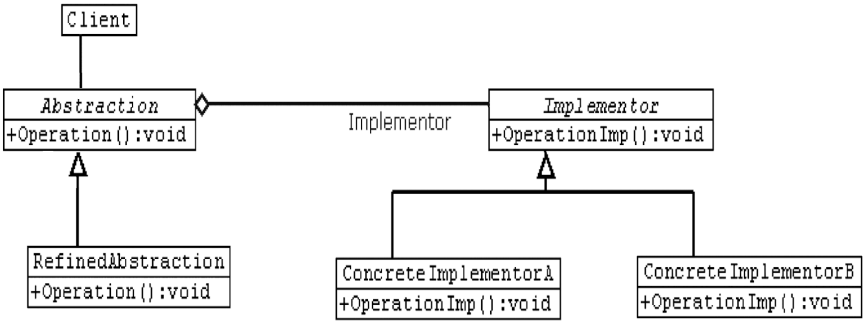


Abb. 3.7 Brücke

Dekorierer (Decorator, Wrapper)

Der Dekorierer erweitert eine Komponente um Funktionalität, und zwar dynamisch. Möchte man also einer Komponente dynamisch und transparent mehr Funktionalität zuweisen, ohne dass dabei die Komponente selbst erweitert wird, so bietet sich dieses Entwurfsmuster an. Man beachte: Während das Entwurfsmuster „Adapter" lediglich eine Schnittstelle verändert, so verändert der Dekorierer die Funktionalität einer Komponente (nicht aber die Schnittstelle). Dies ist für die Integration, Weiterbenutzung und auch Kapselung von Bedeutung (vgl. Abb. 3.8).

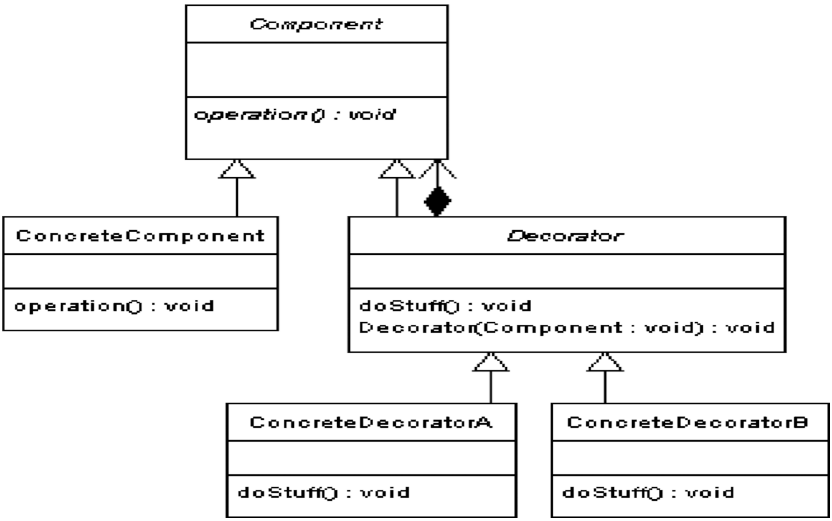


Abb. 3.8 Dekorierer

Fassade (Facade):
Das Entwurfsmuster Fassade liefert eine einheitliche Schnittstelle gegenüber einer Menge von Schnittstellen (eines oder mehrere Subsysteme). Dadurch lassen sich Abhängigkeiten zwischen unterschiedlichen Systemkomponenten reduzieren. Die Fassade muss demnach die internen Details der Subsysteme kennen. Sie muss also z.B. wissen, welche der internen Komponenten für eine bestimmte Aufgabe zuständig sind und delegiert dann entsprechende Anfragen an die dafür zuständigen Subsystemkomponenten weiter (Abb. 3.9).

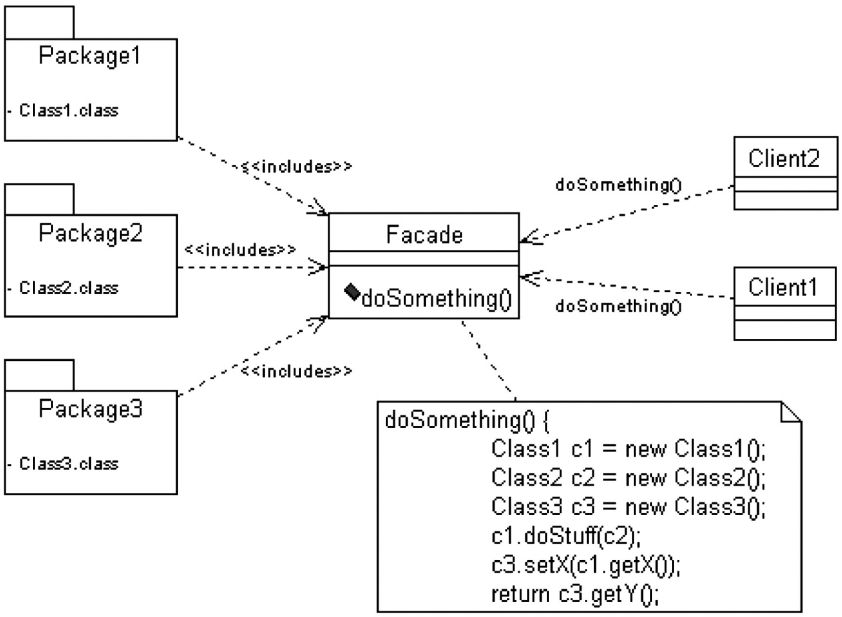


Abb. 3.9 Fassade

Abb. 3.9 zeigt rechts zwei Klienten, die auf (verschiedene) Subsysteme der links stehenden Pakete zugreifen möchten. Die Klasse „Facade“ verbirgt die Subsystemkomponenten nach außen und entscheidet, welcher Zugriff genau wohin geht. Dieses Entwurfsmuster bietet sich also insbesondere immer dann an, wenn man zu einem komplexen Subsystem eine einfache Schnittstelle anbieten möchte. Außerdem liefert sie Klienten eine vereinfachte Sicht auf das Subsystem.

Fliegengewicht (Flyweight)
Dieses Muster erlaubt es, kontextabhängige Information aus einem Objekt zu entfernen und extern zu verwalten, um so die Anzahl benötigter Klassen zu

minimieren. Dies wird benutzt, um zu viele Objekten einer Klasse zu reduzieren. Wenn es möglich ist, aus der Klasse kontextabhängige Informationen (extrinsisch) herauszunehmen und von einem Kontextobjekt verwalten zu lassen, so enthält die Klasse nur noch kontextunabhängige (intrinsische) Informationen.
Dann können Objekte, die sich bisher nur im Kontext unterschieden haben, durch ein einzelnes Objekt repräsentiert werden. Statt mit vielen verschiedenen Objekten wird nun mit Referenzen auf gemeinsam genutzte Objekte gearbeitet. Einer Methode wird beim Aufruf die kontextabhängige Information mitgegeben (Abb. 3.10).

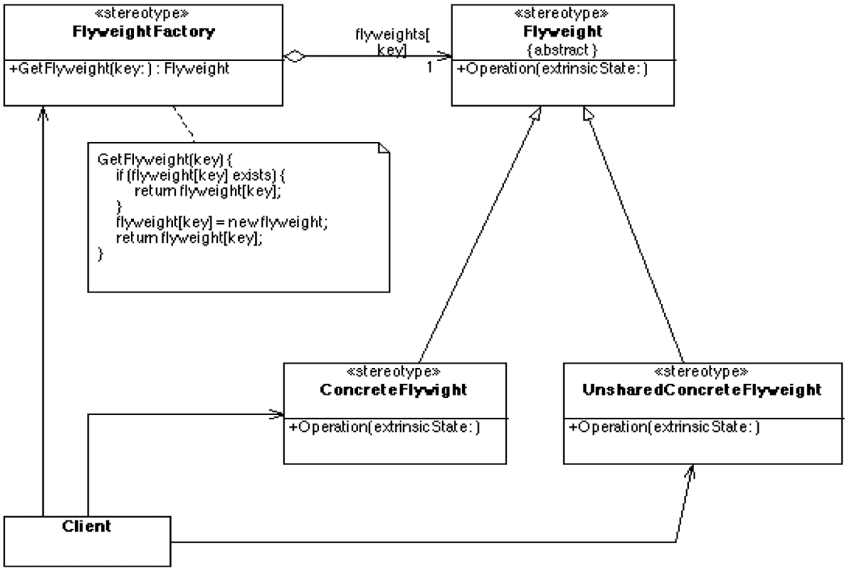


Abb. 3.10 Fliegengewicht

Kompositum (Composite)
Dieses Muster bezeichnet eine Gruppierung von Objekten, wobei die Gruppe selbst wieder Objekt gleicher Art ist. Einzelne Objekte sollen also so gruppiert werden, dass die Gruppe sich verhält wie ein einzelnes Objekt und damit wieder mit anderen Objekten/Gruppen gruppiert werden kann. Für die Objekte existiert eine Basisklasse, und von dieser wird das Kompositum abgeleitet, wodurch es die gleiche Schnittstelle wie die Objekte hat und sich entsprechend verhält. Hinzu kommt aber, dass das Kompositum Objekte dieser Basisklasse aufnehmen

kann. Ein Methodenaufruf wird von dem Kompositum an alle aggregierten Objekte weitergereicht (z.B. für die Gruppierung von Grafikelementen o.ä.).

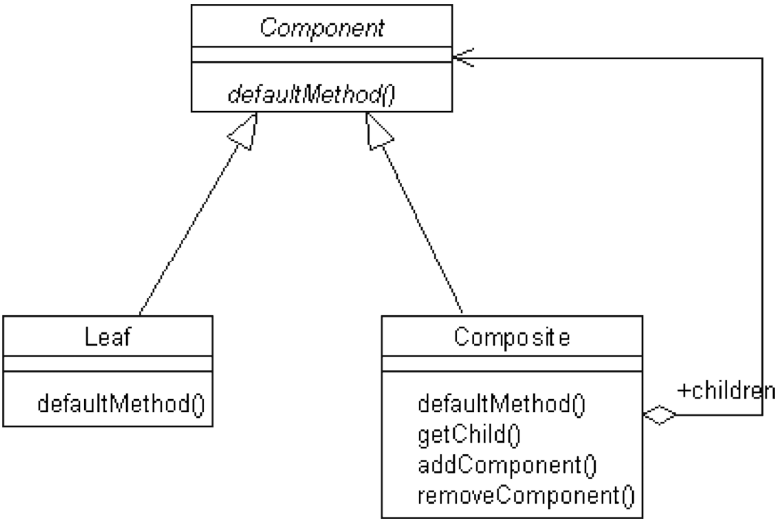


Abb. 3.11 Kompositum

In Abb. 3.11 sehen wir die Basisklasse „Component“, deren „Blatt“ (Leaf) alle Methoden von „Component“ implementiert. Die Klasse „Composite“ schließlich implementiert Methoden zur Manipulation der „Children“. Die Methoden werden gewöhnlich an diese Children delegiert.

Proxy (Surrogat)
Ein Proxy (auch „Stellvertreter“ genannt) stellt einen Platzhalter für eine andere Komponente (wir nennen sie „Subjekt“) dar. Sie kontrolliert den Zugang zum sog. „echten Subjekt“. Die Schnittstelle des Proxy ist dabei identisch mit der Schnittstelle des echten Subjekts. Damit kann der Proxy als Ersatz des Subjekts dienen. Man setzt dieses Entwurfsmuster meistens ein, wenn man den Zugriff auf das echte Subjekt kontrollieren will (Abb. 3.12). Der Proxy kann das echte Subjekt erzeugen und löschen. Man unterscheidet dabei:

Remote Proxy (entfernte Stellvertreter)
Hier werden die Anfragen an entfernte Subjekte (z.B. auf einem anderen Rechner) kontrolliert.

Virtual Proxy (virtueller Stellvertreter)
Will man den Zugriff auf das echte Subjekt verzögern, so kann man dieses Muster einsetzten. Es enthält Informationen des echten Subjekts. So ein verzögerter Zugriff macht z.B. Sinn, wenn der Zugriff auf das echte Subjekt langsam, aufwändig oder teuer ist.

Protection Proxy (Schützender Stellvertreter)
Damit lässt sich prüfen, ob der auf das echte Objekt Zugreifende über ausreichende Rechte verfügt.

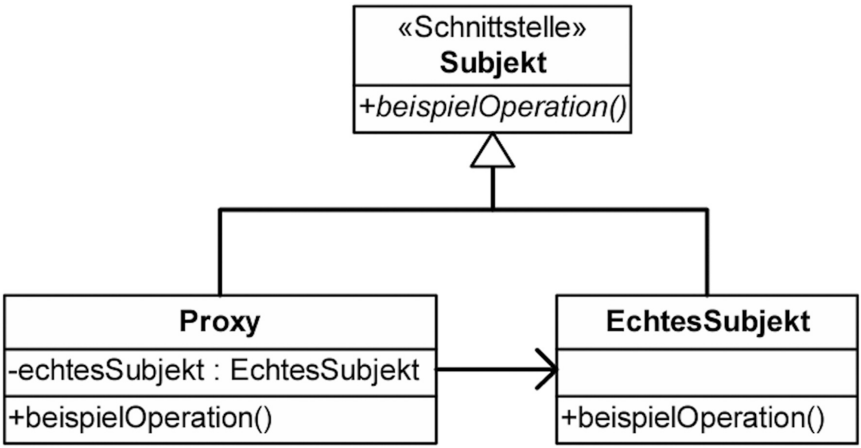


Abb. 3.12 Proxy[20]

Klassenbasiert:

Interpreter
Hier wird die Überführung einer Grammatik in Objekte zur Konstruktion eines Interpreters realisiert. Folgt eine Sprache einer Grammatik, kann diese durch Objekte dargestellt werden (Terminal/Nichtterminal-Objekte). Diese verhelfen wiederum zum Bau eines Interpreters, der Sätze der Sprache interpretieren kann. Dadurch kann dieses Muster z.B. eingesetzt werden, um eine spezielle Programmier- oder Abfragesprache (wie SQL) zur schnellen Problemlösung zu implementieren (siehe Abb. 3.13).

[20] Quelle: ibid.

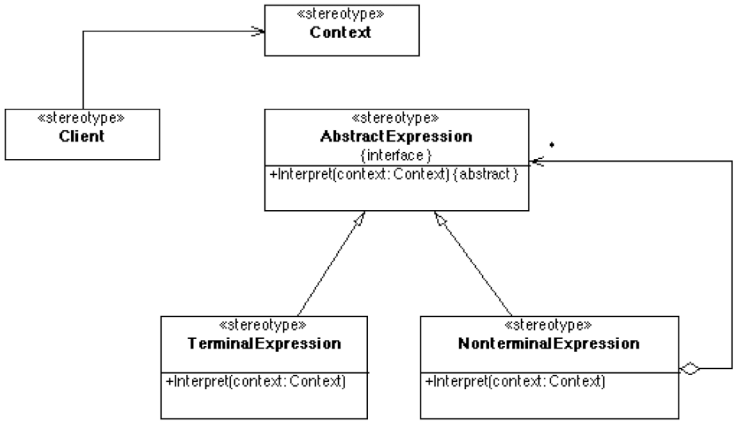


Abb. 3.13 Interpreter

Schablonenmethode (Template Method)

Dieses Muster beschreibt aus Primitivmethoden zusammengesetzte Skelett-Algorithmen, wobei die Primitiven durch Nachfolgeklassen implementiert werden. Eine Schablonenmethode stellt einen Algorithmus dar, von dem nur das Skelett existiert. Dieses wird aus primitiven (und vor allem abstrakten) Methoden zusammengesetzt. Damit ist der grobe Fahrplan des Algorithmus festgelegt, die abstrakten Methoden werden durch die Unterklassen implementiert (Abb. 3.14).

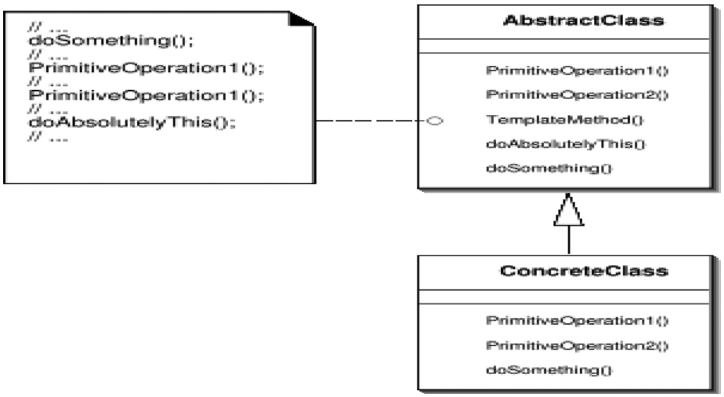


Abb. 3.14 Schablonenmethode

Anders gesagt: Wenn die Algorithmen der Unterklassen strukturell gleich sind und sich nur in spezifischen Implementierungen unterscheiden, so kann man diese Struktur mittels abstrakter Methoden auch schon in der Basisklasse festlegen.

Objektbasiert:

Befehl (Command, Action, Transaction)
Befehle werden als Objekte gekapselt, die dann „herumgereicht“ werden können. Der Aufrufer kennt mitunter den Empfänger nicht, aber er kennt die Basisklasse Befehl mit der Methode Execute. Der konkrete (abgeleitete) Befehl kennt dann den Empfänger und ruft dessen Methode (Action) auf. Parameter, Stapelung, Makrobefehle, Logbuch und Rückgängig wird dadurch möglich (Abb. 3.15).

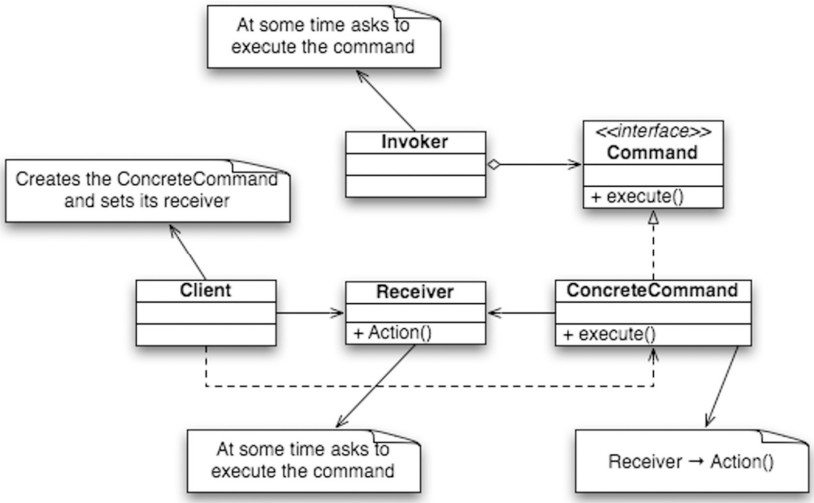


Abb. 3.15 Befehlsmuster[21]

Eine Drucker-Bibliothek enthält beispielsweise eine Klasse “Print Job”. Ein Anwender wird nun typischerweise ein neues Print-Objekt erzeugen, seine Eigenschaften festlegen und schließlich eine Methode aufrufen, welche das Objekt zum Drucker schickt.

[21] Quelle: http://en.wikipedia.org/wiki/File:Command_Design_Pattern_Class_Diagram.png

Beobachter (Observer):
Ein Observer definiert eine kontrollierte Abhängigkeit zwischen Objekten, sodass die Änderung eines Objektes die Benachrichtigung und Aktualisierung aller abhängigen Objekte auslöst. Der Observer wird also verwendet, wenn eine Komponente anderen Komponenten eine Nachricht zukommen lassen will, ohne zu wissen, wer die anderen Komponenten sind oder wie viele Komponenten geändert werden sollen (Abb. 3.16).

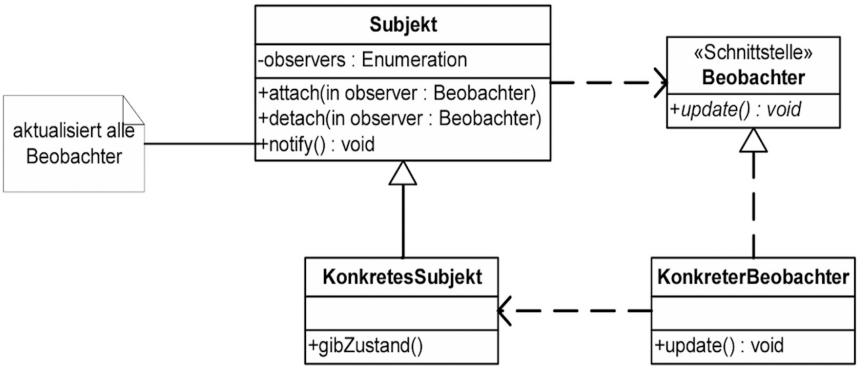


Abb. 3.16 Beobachtermuster[22]

Das heißt, wenn eine Komponente (=ein Beobachter) auf eine Zustandsänderung einer anderen Komponente reagieren soll, ohne dass diese den Beobachter kennt, dann ist dieses Entwurfsmuster angebracht.

Es können mehrere Beobachter ein „Subjekt" (=die Komponente, die beobachtet wird) observieren. Ändert das Subjekt seinen Zustand, werden alle Beobachter davon benachrichtigt. Der neue Zustand des Subjekts wird erfragt und alle Beobachter „synchronisieren" sich mit dem Subjekt.

Das Beobachter-Muster basiert übrigens auf dem Model-View-Controller-Prinzip (MVC), welches im Rahmen der Software-Ergonomie eine wichtige Rolle spielt.

[22] Quelle: Starke, G.: *Effektive Software-Architekturen*, Hanser, 2008

Besucher (Visitor)
In diesem Entwurfsmuster werden Algorithmen aus ihren Klassen gelöst und in Algorithmenklassen zusammengefasst. In einer Hierarchie mit vielen Klassen verteilt sich ein Algorithmus (in Form von Methoden) auf jede einzelne Klasse (Abb. 3.17).

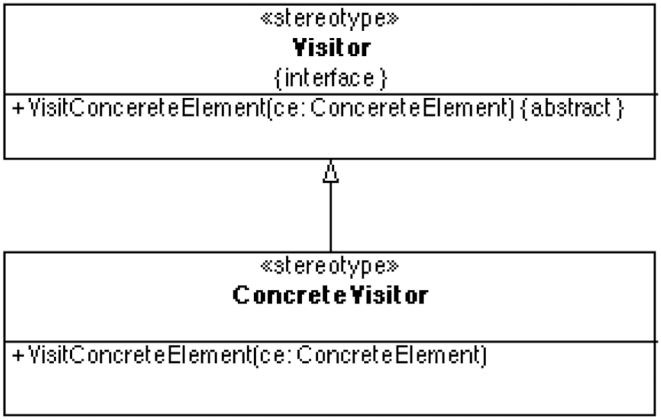


Abb. 3.17 Besuchermuster

Bei vielen Algorithmen wäre es schöner, wenn die Klassenhierarchie zunächst unabhängig von den Algorithmen ist und ein Visitor, was einem Algorithmus-Objekt entspricht, die einzelnen Objekte traversiert und dann den Algorithmus entsprechend dem Objekt ausführt. Dadurch sind (objektspezifische) Algorithmen nicht mehr auf die Objekte verteilt, sondern beim Algorithmus-Objekt zentralisiert. Die Objekte müssen den Visitor akzeptieren und sich selbst an den Algorithmus übergeben.

In Abb. 3.18 sehen wir, wie sich das dann am konkreten Objekt manifestiert: Eine Objektstruktur besteht aus verschiedenen Elementen, von denen konkrete Elemente abgeleitet sind. Diese „akzeptieren“ den Besucher und führen die entsprechenden Operationen aus.

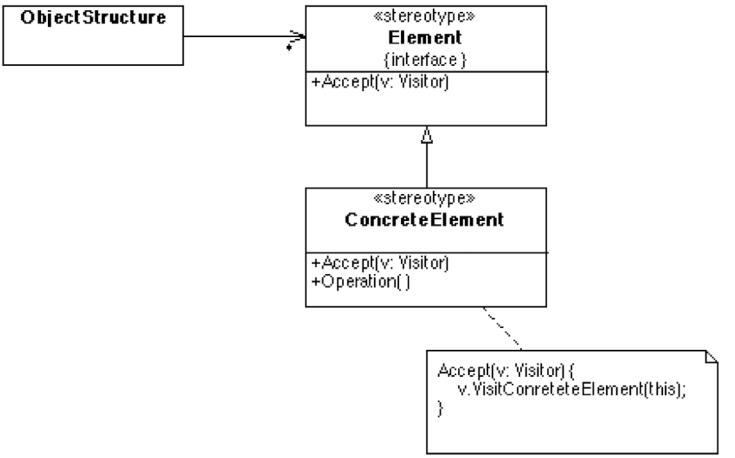


Abb. 3.18 Anwendung des Besuchermusters auf eine Objektstruktur

Iterator (Cursor)

In diesem Muster geschieht eine Trennung von Listen-Datenstruktur und Element-Traversion/Iteration mittels Iteratorobjekten. Man trennt die Listenstruktur von der Iteration derselben, indem man ein eigenes Iterator-Objekt entwirft, welches für die Traversierung zuständig ist (Abb. 3.19).

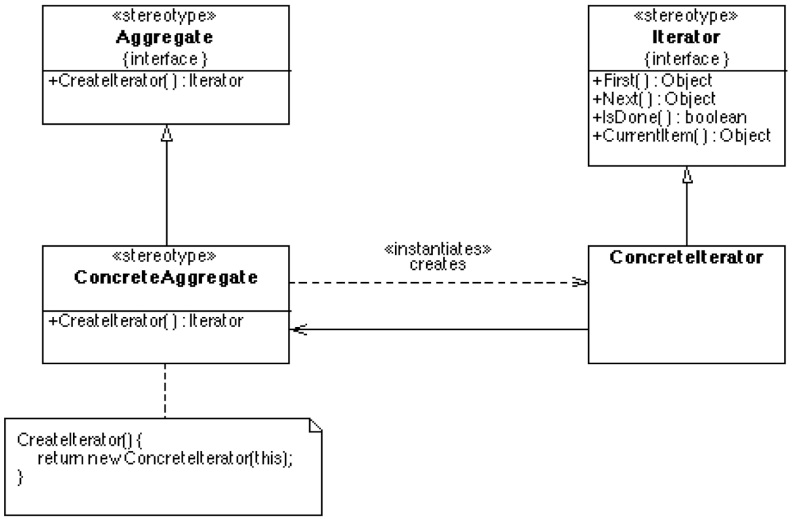


Abb. 3.19 Iterator

Dadurch können auf einer Liste verschiedene Iterationen stattfinden, Iterationen mit Filter etc. sind möglich, unabhängig von der inneren Struktur. Die Liste erzeugt einen Iterator mittels einer Fabrikmethode. Dadurch erzeugt die konkrete Liste ihren konkreten Iterator. Iteratoren werden also dazu benutzt, sequentiellen Zugriff auf die Elemente eines aggregierten Objekts zu ermöglichen, ohne seine dahinter liegende „Repräsentation“ offenzulegen. Damit kapselt der Iterator die interne Struktur bzw. das Aussehen seiner internen Iteration.

Memento (Token)
Diese Muster gewährleistet die „Innerer-Zustand“-Speicherung für Undo-Operationen ohne Aufhebung der Kapselung. Ein Objekt (Urheber) soll seinen Zustand so speichern, dass ein eventuelles Undo möglich ist. Dies geschieht, indem der Klient ein Memento von dem Objekt anfordert, indem das Objekt seinen Zustand kodiert. Durch Rückgabe des Mementos an das Objekt kann dieses sich in den alten Zustand zurückversetzen. Andere Objekte können das Memento nicht interpretieren, die Kapselung bleibt bewahrt (Abb. 3.20).

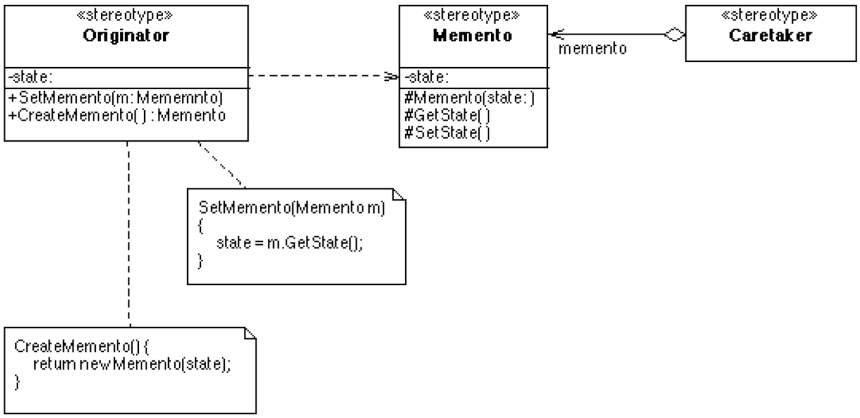


Abb. 3.20 Memento

Strategie (Strategy, Policy)
Das Strategiemuster garantiert eine Trennung von Algorithmen und Objekten durch austauschbare „Algorithmen-Objekte“. Man entkoppelt Algorithmen vom Kontext-Objekt und erzeugt eigenständige Algorithmen-Objekte. Die Algorithmen-Objekte benutzen die gleiche Schnittstelle, wenn sie die gleiche Aufgabe bearbeiten (Abb. 3.21).
Dem Kontext-Objekt kann ein beliebiger Algorithmus (Auswahl durch den Klienten) zugeordnet werden zur Lösung der Aufgabe.

Algorithmenverbesserung/-hinzunahme ist so ohne die Änderung des Kontext-Objektes möglich.

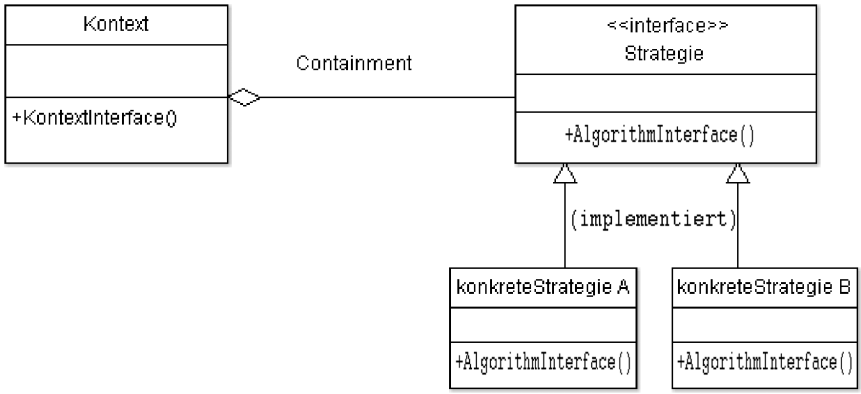


Abb. 3.21 Strategiemuster

Vermittler (Mediator)
Bei diesem Muster werden Objektabhängigkeiten nicht durch die Objekte, sondern durch Vermittler festgelegt. Wenn sich ein Verhalten auf mehrere Objekte verteilt, so können Abhängigkeiten zwischen den Objekten entstehen. Dadurch erfordern Änderungen die Reaktionen anderer Objekte (Abb. 3.22).

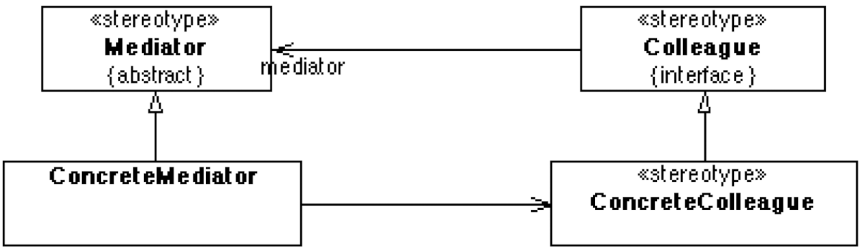


Abb. 3.22 Vermittler

Um nicht die Objekte/Klassen alle verknüpfen zu müssen, wird ein Vermittler zwischengeschaltet, der die Reaktion auf Veränderungen steuert. Alle Objekte sind nur mit dem Vermittler verknüpft, der Vermittler kennt alle beteiligten Objekte und steuert ihr abhängiges Verhalten.

Zustand (State)
Das Zustandsmuster ermöglicht einem Objekt sein Verhalten zu verändern, falls sich sein (interner) Zustand ändert. Nach außen wirkt das dann so, als ob das Objekt seine Klasse gewechselt hat. Das kommt meistens dann vor, wenn eine Komponente in einem bestimmten Zustand nur bestimmte Operationen ausführen darf (Abb. 3.23).
Ein klassisches Beispiel ist z.B. ein Geldautomat. Er darf Geld nur dann rausrücken, wenn ein bestimmter Zustand erreicht ist. In der Regel beschreibt man das mit Zustandsdiagrammen. Das Zustandsmuster kann nun über einen „Polymorphismus" die für einen bestimmten Zustand zulässigen Operationen durch eine konkrete Unterklasse implementieren. Letztere kann dann bei Zustandsänderungen ausgetauscht werden. Zur Laufzeit wird die zuständige Operation vom abstrakten Zustand über einen Polymorphismus durch den jeweiligen instanziierten konkreten Zustand realisiert.

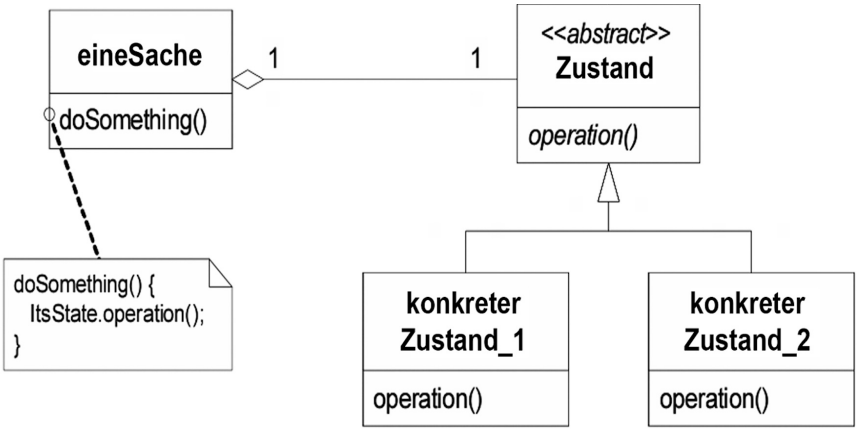


Abb. 3.23 Zustandsmuster[23]

Zuständigkeitskette (Chain of Responsibility)
Anfragen werden in diesem Entwurfsmuster entlang einer Kette an ihre potentiellen Empfänger weitergereicht (Abb.3.24). Sendet ein Objekt eine Anfrage, wo der Empfänger noch nicht bekannt ist bzw. die Anfrage an mehrere Objekte gerichtet ist, so kann man die potentiellen Empfänger entlang einer Kette testen, bis ein Objekt auf die Anfrage reagiert. Dabei kennt jedes Objekt einen Nachfolger bzgl. der Zuständigkeit. Baumstrukturen etc. sind möglich.

[23] Ibid.

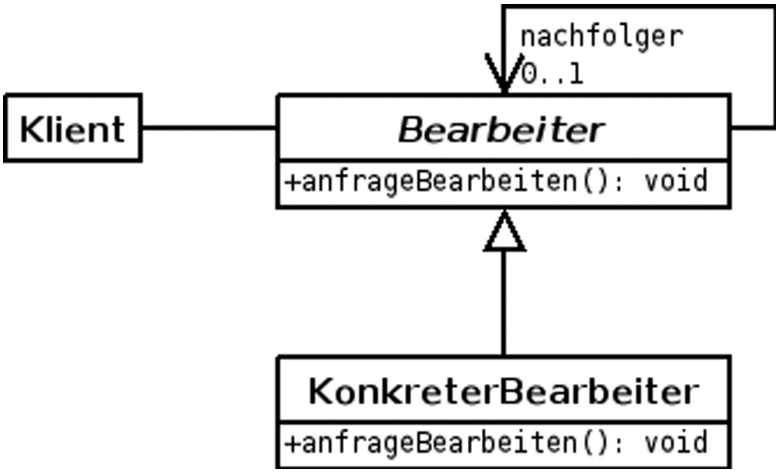


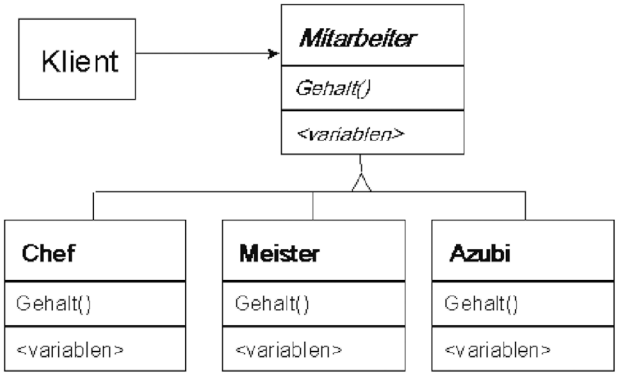
Abb. 3.24 Zuständigkeitskette

Übungsaufgaben:

1. Das Beobachtermuster beschreibt eigentlich dynamisches Verhalten. Daher kann man es auch als Sequenzdiagramm darstellen. Geben Sie eine solche Darstellung an.

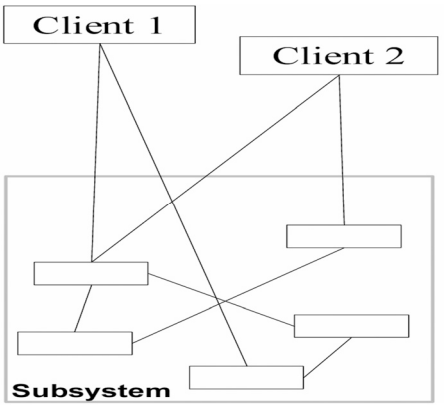
2. Beschreiben Sie das Einschalten, Ausschalten und Zerstören einer Lampe mittels:
   a) eines Zustandsdiagramms
   b) des Entwurfsmusters „Zustand“

3. Angenommen, in einem Programm zur Mitarbeiterverwaltung gibt es drei Arten von Mitarbeitern, die in drei Klassen implementiert sind. Die Struktur der Objektvariablen ist gleich, nur der Algorithmus zur Gehaltsberechnung ist verschieden. Ein „Klient“ stellt dabei die Anfrage zur Gehaltsermittlung. Nachfolgend ein mögliches UML-Diagramm hierfür:



Eine genauere Analyse des Problems legt nahe, hierfür das objektbasierte Verhaltensmuster „Strategie“ zu benutzen. Dieses Muster wird nämlich vorzugsweise dann benutzt, wenn sich die meisten Klassen nur in ihrem Verhalten unterscheiden. Geben Sie eine entwurfsmusterfreundliche Darstellung an, wo die Objekte geeignet gekapselt sind!

4. Es seien zwei Clients gegeben, die auf diverse Komponenten eines Subsystems zugreifen:



Wie würde das System aussehen, wenn Sie das Entwurfsmuster „Fassade“ einsetzten?

4. Architektur-Sichtenmodelle

In Kapitel 1 hatten wir angedeutet, dass es –wie im Baubereich- verschiedene Ansichten auf das zu entwickelnde System geben kann. Insbesondere hatten wir darauf hingewiesen, dass jede Sicht für einen bestimmten Kreis der Projektbeteiligten („Stakeholder“) besondere Relevanz besitzt. Das ist ebenfalls mit dem Baubereich vergleichbar, wo z.B. ein Verlegeplan der Abflussrohre für die Leute vom Hoch- und Tiefbau interessant ist, aber nicht für den Elektrofachbetrieb. Letzter benötigt eigene Pläne („Sichten“) für seine Arbeit. Und so wie jeder Fachbetrieb „seine“ Sprache in den entsprechenden Plänen wiederfindet, so haben wir uns in Kapitel 2 mit den Architekturbeschreibungssprachen des Software-Engineerings beschäftigt; dieses wurde zum Teil bereits für Kapitel 3 umgesetzt, wo wir Entwurfsmuster mittels UML beschrieben haben.
Seine volle Wirkung entfaltet das ganze aber erst jetzt, wo wir lernen wollen, wie man die verschiedenen Baupläne einer Software erstellt. Zur Erinnerung: Ein Architekt im Baugewerbe muss selbst auch *alle* „Fachsprachen“ beherrschen: Er oder sie muss wissen, wie man einen Verlegeplan für Abflussrohre zeichnet, wie man den Plan für die Elektroinstallateure erstellt, wie man die Ansichten für die Maurer und den Innenarchitekt malt etc.; daher muss der Software-Architekt ein vergleichbares Wissen für die Erstellung der nachfolgend genauer betrachteten (An-)Sichten betreffs der Software-Architektur haben. Er muss zumindest die Pläne in der „richtigen“ Fachsprache erstellen können, so dass die jeweiligen Stakeholder diese Pläne dann konkret im Projekt umsetzten können.
Nun ist es leider wie sooft im Software-Engineering aber so, dass die Nomenklatur der Software-Baupläne nicht einheitlich ist. Und das gilt nicht nur für Architekturbeschreibungssprachen, wie wir in Kapitel 2 schon sahen, sondern auch für die abstrakteren Sichtenmodelle, welche mit diesen Sprachen erzeugt werden. Wobei der UML-Standard als ADL wohl nicht mehr zu verdrängen ist und schon einiges abdecken kann. Trotzdem bleibt uns nichts anderes übrig, als die zur Zeit bekanntesten Varianten der Sichtenmodelle vorzustellen und eines davon repräsentativ für alle anderen genauer unter die Lupe zu nehmen. Wir beginnen also zuerst mit einem Überblick und werden dann das letzte der vorgestellten Sichtenmodelle ausführlich behandeln.

4+1-Sichtenmodell nach Kruchten:

Dieses Modell besteht, wie der Name schon sagt, aus 5 Sichten, wobei die 5. Sicht bezüglich ihrer „Perspektive“ etwas aus dem Rahmen fällt, weshalb die Autoren deshalb wohl das Ganze nicht 5-Sichtenmodell, sondern 4+1-Sichtenmodell nannten. Die Sichten im Einzelnen:

Logical View
Hier geschieht die Darstellung der Schlüsselabstraktion des Systems und deren Beziehungen sowie die Identifikation der systemweiten Mechanismen und Designelemente. Eine starke Fokussierung der Sicht ist durch die Anwendungsdomäne und weniger durch die technische Realisierung gegeben.

Process View
Hier wird der Zuordnung von Elementen aus der logischen Sicht zu Kontrollflüssen dargestellt. Die Fokussierung dieser Sicht ist auf Parallelität, Verteilung, Systemintegration und Fehlertoleranz gerichtet.

Deployment View
Hierbei handelt es sich um die Darstellung der Anordnungen von Softwareeinheiten in einer verteilten Umgebung.

Physical View
Diese Sicht beinhaltet die Zuordnung der Elemente aus der logischen Sicht und der Verteilungssicht zu den Ausführungseinheiten.

„Orthogonal" dazu beschreibt Kruchten die Use-Case-Sicht (daher auch die Bezeichnung 4+1):

Use Case View
Dies ist die Beschreibung einer Menge von Anwendungsfällen, die durch das Zusammenspiel der vier anderen Sichten realisiert werden.
Dieses Modell ist übrigens Bestandteil des Rational Unified Process (RUP).

Siemens-Sichten-Modell nach Hofmeister, Nord und Soni

Dieses Modell kennt ebenfalls 4 Sichten:

Conceptual Architecture View
Diese Sicht enthält die wesentlichen Bestandteile des Systems. Es wird die Umsetzung der Anforderungen auf die Architektur beschrieben. Das System wird in konzeptuelle Komponenten zerlegt, deren Verbindungen definiert werden.

Module Architecture View
Bei dieser Sicht liegt der Schwerpunkt auf der technologischen Umsetzung der in der konzeptuellen Architektursicht beschriebenen Konzepte. Daher spielt hier die Zuordnung von konzeptuellen Komponenten und deren Verbindungen zu Subsystemen bzw. Modulen eine wichtige Rolle. Es sind Schichtenbildung und die Beachtung der vorhandenen Softwareplattform wichtige Themen.

Execution Architecture View
Hier wird vor allem die Laufzeitumgebung beschrieben. Die Zuordnung der vorhandenen Komponenten zu Elementen des Laufzeitsystems (z.B. Prozessen) und die Zuordnung der logischen Ressourcen zu den physikalischen Ressourcen (z.B. Hardware) steht dabei im Vordergrund.

Code Architecture View
Hier geht es um die Umsetzung und Verteilung von ausführbaren Einheiten, d.h. um die Abbildung von Modulen auf Quellcode und der Erzeugung ausführbarer Einheiten aus Quellcode.

4-Sichten-Modell nach Starke

Wir setzen im Folgenden unseren Schwerpunkt auf eine Darstellung nach Dr. Gernot Starke[24]. Er beschreibt ein 4-Sichten-Modell, welches sich in vielen praktischen Softwareprojekten bewährt hat. Außerdem sind hier Agilität und Angemessenheit in den Vordergrund gestellt. Starke beschriebt diese 4 Sichten wie folgt:

Kontextsichten – Wie ist das System in seine Umgebung eingebettet?

Kontextsichten zeigen das System als Blackbox in seinem Kontext aus einer Vogelperspektive. Hier zeigt man die Schnittstellen zu Nachbarsystemen, die Interaktionen mit wichtigen Stakeholdern sowie die wesentlichen Teile der umgebenden Infrastruktur.

Bausteinsichten – Wie ist das System intern aufgebaut?

Bausteinsichten zeigen die statischen Strukturen der Architekturbausteine des Systems, Subsysteme, Komponenten und deren Schnittstellen. Man sollte die Bausteinsichten top-down entwickeln, 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.

[24] Starke, G.: *Effektive Software-Architekturen*, Hanser, 2008, Seite 82ff

Laufzeitsichten – Wie läuft das System ab?

Die Laufzeitsichten beschreiben, welche Bausteine des Systems zur Laufzeit existieren und wie sie zusammenwirken. Im Gegensatz zu der 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.

Wie man sieht, benutzt Starke den Plural für die einzelnen Sichten. Er begründet das damit, dass es in Realität tatsächlich auch mehrere Elemente in jeder Sicht gibt. Es gibt als z.B. nicht nur eine Laufzeitsicht, sondern unter Umständen deren viele.

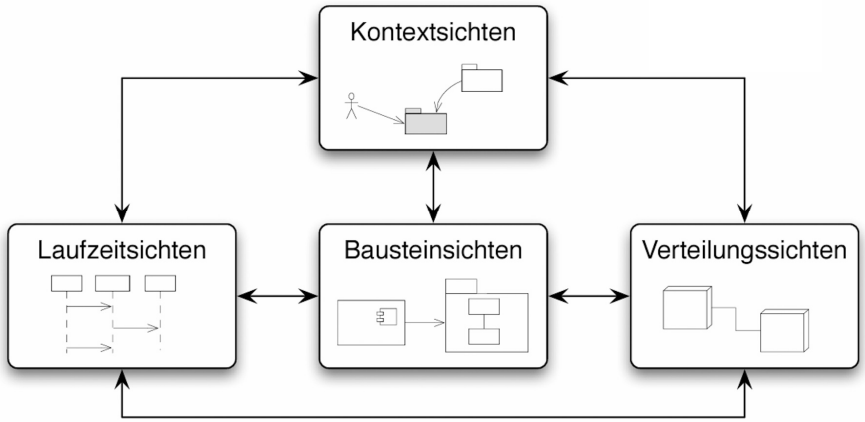


Abb. 4.1 4-Sichtenmodell nach Starke[25]

Die Pfeile zwischen den Sichten stellen mögliche Abhängigkeiten oder Wechselwirkungen dar. Während des Projekts verfeinert man häufig einzelne Teile der Sichten. Oft beschreibt man während des Projektverlaufs auch unterschiedliche Architekturbausteine des gleichen Abstraktionsniveaus in verschiedenen

[25] Quelle: ibid., S. 83

Sichten der gleichen Art. Besteht ein System z.B. aus drei Subsystemen, dann werden zu jedem dieser Subsysteme eine eigene Baustein- und Laufzeitsicht entwickelt.
Manchmal wünschen bestimmte Stakeholder ihre „eigene“ Sicht. Sowas kann (notfalls) gemacht werden, wenn die 4 angegebenen Sichten dafür nicht geeignet sind. Dann kann man zusätzlich „neue“ Sichten definieren. Wenn bei einem Projekt z.B. extensiv Datenflüsse eine wichtige Rolle spielen, so kann man dafür eine eigene Datensicht erfinden, obwohl man im Normalfall Datenmodellierungen nicht als eigene Sicht zu einer Architektur ansehen würde. Starke hat diese auch nicht extra in seinem Modell untergerbacht hat. Wenn man keine eigene Sicht dafür erzeugt, würde man diese Dinge in der Bausteinsicht unterbringen (es gibt dann eben einen Baustein „Datenflüsse“ oder „Datenmodellierung“).
Die Reihenfolge der Erzeugung der Sichten hängt von den jeweiligen Gegebenheiten ab. Starke empfiehlt zu beginnen mit der

- Bausteinsicht, wenn bereits ähnliche Systeme entwickelt wurden und eine genaue Vorstellung von benötigten Implementierungskomponenten vorliegt oder ein bereits teilweise bestehendes System verändert werden muss und damit Teile der Bausteinsicht vorgegeben sind.
- Laufzeitsicht, wenn bereits eine erste Vorstellung wesentlicher Architekturbausteine vorhanden und deren Verantwortlichkeit und Zusammenspiel zu klären ist.
- Verteilungssicht, wenn viele Randbedingungen und Vorgaben durch die technische Infrastruktur, das Rechenzentrum oder den Administrator des Systems vorliegen.

Die Kontextsicht stellt in gewissem Sinn eine Generalisierung der anderen 3 Sichten unter Berücksichtigung des Environments dar. Man kann damit beginnen, wenn man sozusagen die „Vogelperspektive“ des Projekts zuerst dokumentieren will. Beim Häuserbau wäre das vergleichbar mit der Tatsache, dass man z.B. mit dem Entwurf des Grundstückes, abhängig von der gewünschten oder vorhandenen Grünbepflanzung und dem umgebenden Wald, den Straßen etc. beginnt und dann das zukünftige Haus als ein Element dieses Entwurfs einbringt. Danach können dann die Pläne für das eigentliche Wohnhaus gemacht werden.
Nachfolgend betrachten wir nun diese 4 Sichten etwas ausführlicher.

Kontextsicht
Diese Sicht entspricht im weitesten Sinn dem „Conceptual Architecture View“ der Siemens-Sicht bzw. dem „Logical View“ des 4+1-Modell nach Kruchen.
Kontextsichten stellen dar:

- das **System als Black-Box**, 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 wichtigsten **Anwendungsfälle (Use Cases)** des gesamten Systems
- die **technische Systemumgebung**, Prozessoren, Kommunikationskanäle etc.

Die Kontextsichten „abstrahieren" die übrigen Sichten und zeigen, wie das System in seine „Umgebung" eingebettet ist. Dem vergleichbar ist im Baugewerbe der Parzellenplan, wie wir ihn in Abb. 1.1 kennen gelernt hatten.
Was die Notation betrifft, so verwendet man, da in der Kontextsicht die abstrakten Darstellungen der Baustein-, Laufzeit- und Verteilungs- und Infrastruktursichten vorhanden sind, deren jeweilige Notation, d.h. beispielsweise für die Darstellung großer Systemstrukturen bieten sich Klassendiagramme an, ggf. mit Paketen und Komponenten. Die Schnittstellen zur Außenwelt lassen sich durch Klassendiagrammen über Assoziationen zu anderen Systemen oder Akteuren darstellen. Und für Anwendungsfälle oder Abläufe bieten sich dynamische UML-Diagramme (Sequenz-, Kommunikations- oder Aktivitätsdiagramme) bzw. Verteilungsdiagramme für die technische Systemumgebung an.
Weil man den „reprensentional gap" zwischen fachlichen und technischen Architekturmodellen minimieren will, geht man besser von einer fachlichen Sicht für den Entwurf der Kontextsicht aus und versucht auf dieser Basis eine Vogelperspektive des Systems zu entwerfen.
Starke empfiehlt, wirklich alle Nachbarsysteme (ohne Ausnahme!) darzustellen. Alle ein- und ausgehenden Daten müssen in der Kontextsicht zu ersehen sein (vgl. Abb. 4.2).
Wenn die Kontextsicht einmal erstellt ist, muss eine intensive Kommunikation mit den beteiligten Stakeholdern erfolgen. Dies sollte natürlich sowieso während des gesamten Projekts gemacht werden, aber gerade am Anfang, wo wichtige Grundentscheidungen getroffen werden, ist eine permanente Kommunikation und ein Feedback der Auftraggeber sehr wichtig. Nur so lassen sich von Anfang an durchgängige Dokumentation und Systementwicklung in Richtung dessen, was am Schluss auch heraus kommen soll, einigermaßen sicherstellen. Oft scheitern Softwareprojekte daran, dass die Vorstellung zwischen den Auftraggebern, den Planern und den Leuten, die das Programm schließlich implementieren, „auseinander läuft". Die führt zu unnötig hohen Kosten und Verzögerungen oder gar – wie angedeutet- zum Scheitern des kompletten Vorhabens.

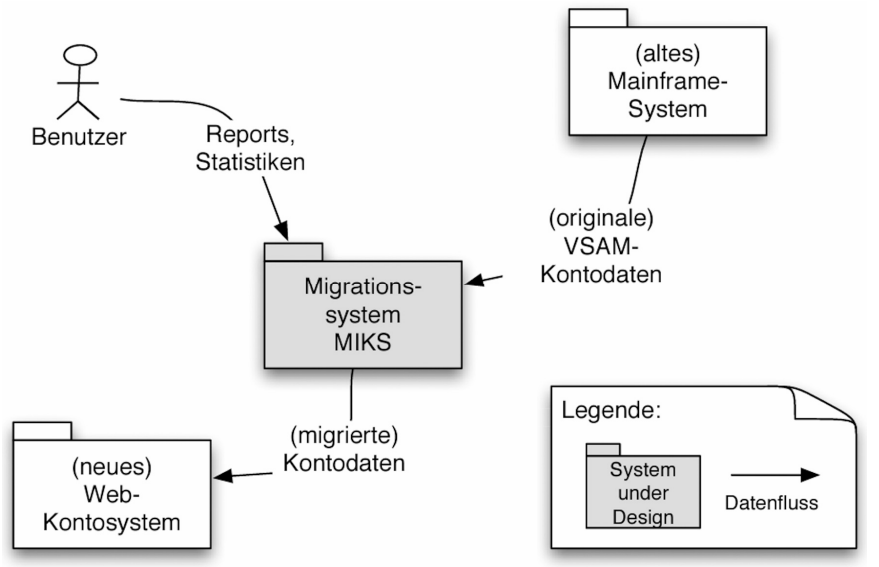


Abb. 4.2 Beispiel Kontextsicht[26]

Bausteinsicht

Für die tatsächliche spätere Realisierung ist die Bausteinsicht von entscheidender Bedeutung. Der Begriff „Baustein“ umfasst nämlich sämtliche Software- und Implementierungskomponenten und stellt so eine Abstraktion vom Quellcode dar (oder genauer: der Quellcode ist eine Spezialisierung dieser Bausteine). Die Bausteinsicht bildet daher die (gewünschten) Funktionalitäten des Systems wie Anwendungsfunktionalität, Kontrollfunktionalität oder Kommunikation auf Software-Bausteine und –Komponenten ab. Vieles, was man aus konventionellen Implementierungsmodellen kennt, findet sich hier wieder. Im Rahmen von UML als Architekturbeschreibungssprache (ADL) zieht man „alle Register“ und benutzt z.B. Pakete, Klassen und Subsysteme und zeigt deren Abhängigkeiten auf. Bezieht man sich auf das klassische Wasserfallmodell, so sind hier vor allem die Phasen DV-Entwurf, Implementierung und Test (3.-5. Phase) untergebracht. Hinzu kommen vom Management ggf. noch Arbeits- und Aktivitätspläne. Gerne benutzt man aber auch hier Diagramme, die verschiedene Detaillierungstiefen darstellen. So kommen häufig Blackbox-Darstellung vor (zur Erinnerung: Eine Blackbox wird nur durch ihre Funktionalität und ihre externen Schnittstellen beschrieben). Aber irgendwann müssen diese Blackboxes „geöffnet“ werden, und so entstehen Whitebox-Darstellungen. Es ist hier also häufig

[26] Quelle: ibid., S. 88

eine Darstellung über mehrere „Levels“ gefragt, die Schritt für Schritt verfeinert werden:

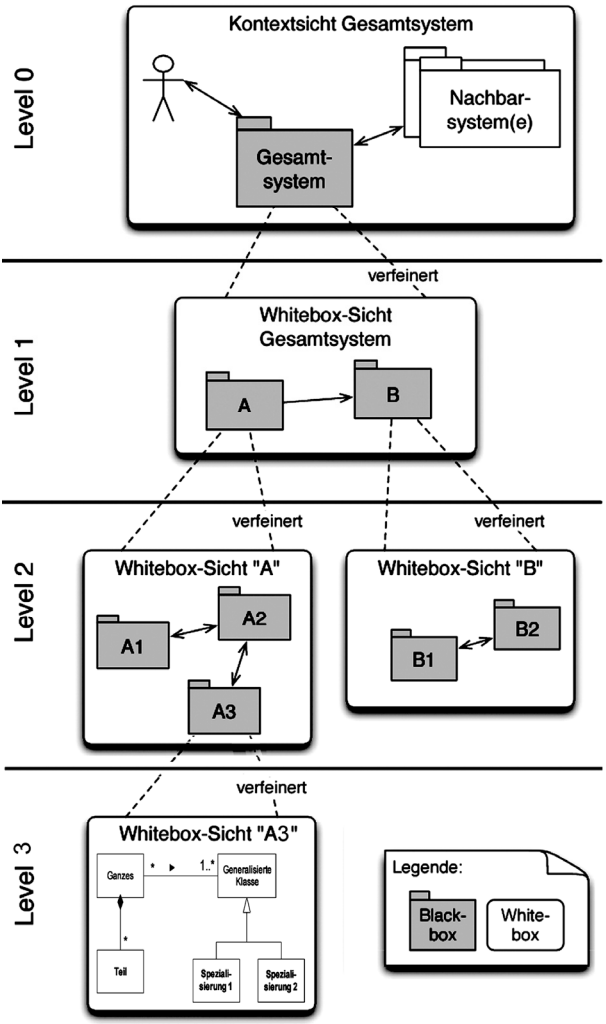


Abb. 4.3 Beispiel: Blackboxes und Whiteboxes[27]

[27] In Anlehnung an ibid., S. 90

In Abb. 4.3 sehen wir eine Hierarchie von Verfeinerungen. Sie beginnt grundsätzlich mit der allg. Kontextsicht auf „höchstem“ Abstraktionsniveau (hier Level 0 genannt). Dann folgen die Verfeinerungsstufen; dies geht so lange, bis der jeweilige Stakeholder konkret damit etwas anfangen kann. In der Darstellung hat sich eingebürgert, dass (in UML als ADL) Pakete für Gruppen, Mengen oder Strukturen von Bausteinen stehen. Komponenten als einzelne Bausteine werden dazu benutzt, ein- und ausgehende Schnittstellen möglichst exakt zu beschreiben, während schließlich Klassensymbole diejenigen Bausteine darstellen, welche die Software-Einheiten oder üblichen Klassen einer objektorientierten Programmiersprache darstellen.
Es ist hier sehr hilfreich, wenn zu jedem Blackbox-Baustein auch dessen spezifische Verantwortlichkeit sowie die Aufgabe in Kurzform beschrieben werden. Beim objektorientierten Entwurf haben sich hier die sog. CRC-Karten (Class-Responsibility-Collaborators) bewährt.
Schnittstellen sollten u.a. folgendes beschrieben: Call-Return (Aufruf/Rückgabe), synchron-asynchron, push/pull oder event notification (Benachrichtigung). Wichtig ist hierbei, dass die Semantik angemessen dokumentiert ist. Wenn UML dazu nicht ausreicht, dann benutzen Sie ggf. zusätzliche Notizen und/oder Pseudo-Code; natürlich stehen auch weitere ADLs zur Verfügung, z.B. Corba-IDL o.ä. (siehe Kapitel 2). Auch sollten insbesondere kritische Schnittstellen möglichst frühzeitig so genau wie möglich beschrieben werden. Es ist oftmals sogar sinnvoll, Code-Artefakte zu benutzen. Wenn immer möglich, sollte Variabilität genutzt werden, um z.B. bestimmte Bausteine auch an anderer Stelle wiederbenutzen zu können. Allerdings sollte man das nicht übertreiben: Je variabler ein Baustein wird, desto komplexer seine Beschreibung. Hier sollte also ein ausgewogenes Verhältnis gefunden werden. Und hier gilt, wie in vielen anderen Bereichen des Software-Engineerings auch: Wiederverwendung bereits vorliegender Bausteine, wenn möglich. Was andere schon entwickelt und getestet haben, ist i.d.R. stabiler als neu entworfenes.

Es empfiehlt sich beim Entwurf der Bausteinsicht folgendes zu berücksichtigen:

- technische Infrastruktur (Rechner, Betriebssysteme, Netzwerke)
- technische Einflussfaktoren (Programmiersprachen, Datenbank, Entwicklungsumgebung etc.)
- organisatorische Einflussfaktoren (Teams, Erfahrungen, Management)
- Qualitätsanforderungen (Performance, Zuverlässigkeit, Änderbarkeit etc.)
- Architekturmustereinsatz prüfen (Referenzarchitekturen)

Im Prinzip sind das also alles Dinge, die in einem „anständigen“ Pflichtenheft bereits beschrieben sein müssten.

Laufzeitsicht (Ausführungssicht)
In der Laufzeitsicht wird beschrieben, welche Bestandteile des Systems zur Laufzeit existieren und wie sie zusammenarbeiten. Dazu gehört auch, wie sich Laufzeitkomponenten aus Instanzen von Implementierungsbausteinen zusammensetzen. Wenn es sich um Embedded Systems und/oder Echtzeitsysteme handelt, so kommt noch eine Spezifikation der Prozesse, Tests und Threads hinzu.

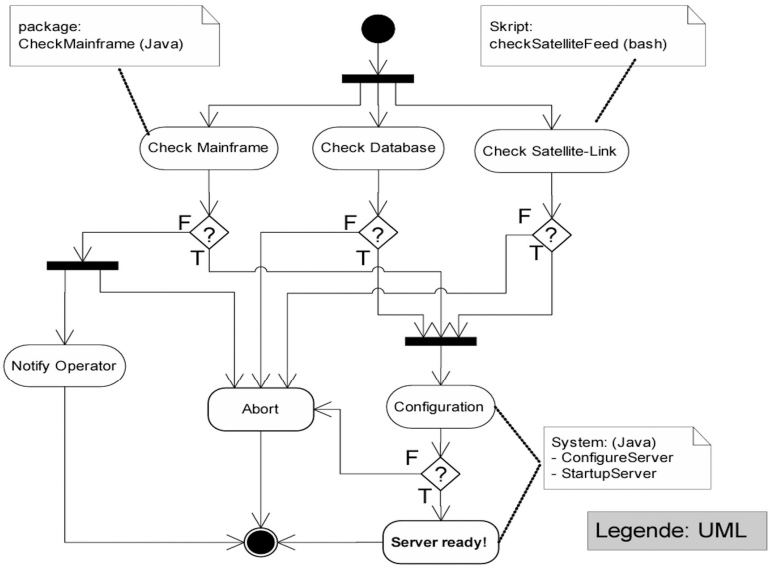


Abb. 4.4 Beispiel: Laufzeitsicht eines Systemstarts[28]

Es sind des Weiteren folgende Aspekte zu beschrieben:

- Zusammenarbeit des Systemkomponenten
- Bearbeitung von Use-Cases durch die Architekturbausteine
- Existenz, Start, Überwachung und Terminierung von Instanzen der Architekturbausteine zur Laufzeit

[28] Quelle: ibid.; S. 95

- Zusammenarbeit von Systemkomponenten mit externen Komponenten
- Systemstart (z.B. notwendige Skripte, Subsystemabhängigkeiten, Datenbanken, Kommunikationssysteme etc.)

Elemente der Laufzeitsicht sind also alle ausführbaren Einheiten sowie deren Beziehungen. Während der Laufzeit werden ja Instanzen der (statischen) Implementierungsbausteine erzeugt (das gilt natürlich nicht für abstrakte Klassen und logische Schnittstellen). Die Beziehungen sind durch Daten- und/oder Kontrollflüsse gegeben. Zur Notation dieser Dinge sind in UML Sequenz-, Aktivitäts- und/oder Kommunikationsdiagramme (früher: Kollaborationsdiagramme) geeignet (siehe Abb. 4.5).

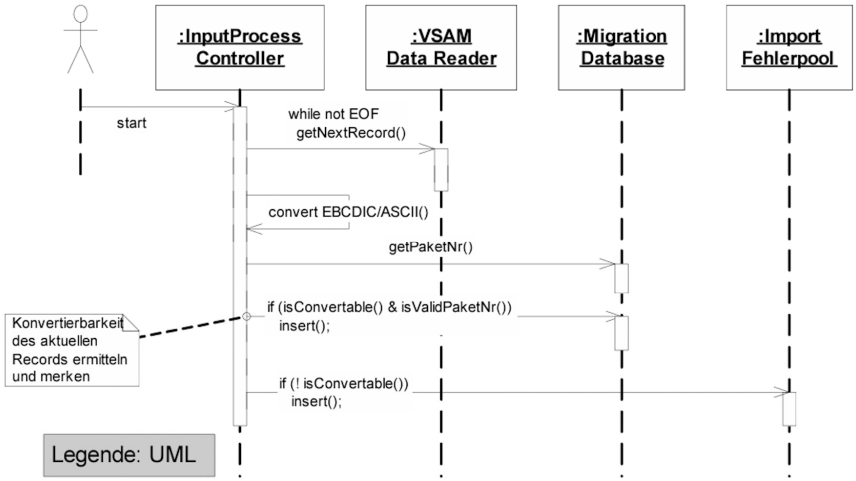


Abb. 4.5 Beschreibung eines Input-Processing[29]

Es ist nun durchaus Gang und Gäbe, hier die UML-Notation etwas „aufzubohren“. So kann man z.B. in den Sequenzdiagrammen auch Paketsymbole etc. verwenden (siehe Abb. 4.6):

[29] Quelle: ibid., S. 96

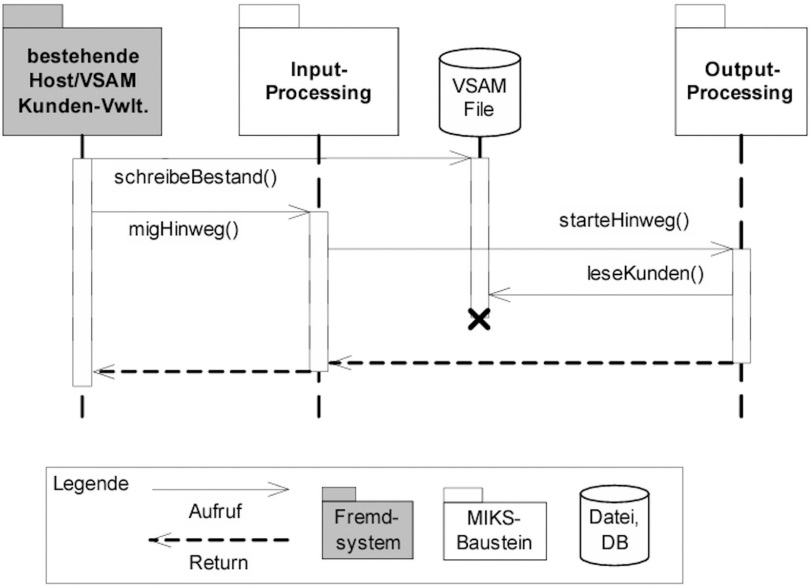


Abb. 4.6 Laufzeitsicht-Beschreibung mit Paketen[30]

Manchmal gibt es hier Überlappungen mit der nachfolgenden Sicht (Verteilungssicht), denn auch hier spielt Laufzeitverhalten eine Rolle. Der Software-Architekt sollte hier entscheiden, wo er welche Dinge unterbringt (auch in beiden Sichten möglich).

Verteilungssicht (Infrastruktursicht)
Die Ablaufumgebung eines Systems (Hardwarekomponenten wie Prozessoren, Speicher, Netzwerke, Firewalls etc.) nebst deren Protokolle werden in dieser Sicht spezifiziert. Dazu gehören auch Leistungsdaten (Speichergrößen, Mengengerüste) und sonstige Parameter.
Wie schon erwähnt, sind auch Laufzeitelemente hier darstellbar. Diese Laufzeitelemente werden zusammen mit sonstigen Bestandteilen der technischen Infrastruktur in der Verteilungssicht als sog. Knoten dargestellt (wobei man sagt, dass die Laufzeitelemente auf den Knoten ablaufen). Die Verbindung zwischen den Knoten nennt man physikalische Kanäle, während die Verbindung zwischen den auf den Knoten laufenden Laufzeitelementen logische Kanäle genannt werden.

[30] Quelle: ibid., S. 97

Grafisch dargestellt werden die Elemente der Verteilungssicht in UML durch Deployment-Diagramme (Einsatz-Diagramme). Komponenten- und Paketsymbole werden dann für die Laufzeitelemente benutzt (das sind dann hier die Software-Systeme). Starke unterscheidet hierbei innerhalb der Darstellung (entgegnen der UML-Norm, was aber hier sehr sinnvoll ist) grafisch die logischen und physikalischen Kanäle (siehe Abb. 4.7).

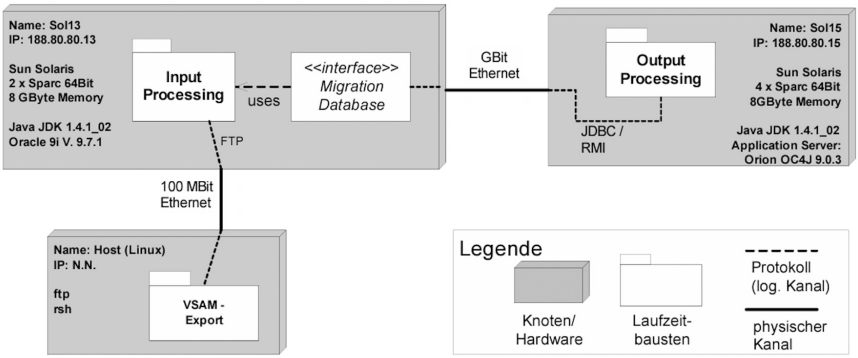


Abb. 4.7 Beispiel: Verteilungssicht[31]

Die Verteilungssicht kann im weitesten Sinne als eine „Landkarte“ der beteiligten Systeme aufgefasst werden. Es sollte dabei auf evtl. Engpässe aufgrund des Mengengerüsts geachtet werden. Bei heterogenen oder verteilten Systemen macht es Sinn, alle vorhandenen Kommunikationsmechanismen (Protokolle, Middleware, Netzwerke etc.) aufzunehmen. Auch auf Verfügbarkeiten der Komponenten sollte ein Augenmerk gerichtet werden.

Wir haben nun einige Architektursichten kennen gelernt, es gibt davon natürlich noch mehr und ich verweise hier auf die Literaturliste am Ende des Buches. Das Prinzip jedoch ist allen gemeinsam: Mit einer oder mehreren Architekturbeschreibungssprachen werden verschiedene Aspekte des geplanten Software-Projekts beschrieben. Diese Beschreibungen werden in den genannten Architektursichten geeignet untergebracht, also gewissermaßen „kategorisiert“. Wir hatten hierfür das 4-Sichten-Modell nach Starke repräsentativ etwas genauer betrachtet, wobei Starke selbst sein Modell ausführlich in der genannten Literatur beschreibt.

[31] Quelle: ibid., S. 98

Übungsaufgaben:

1 Es sei folgende Kundenanforderung gegeben: Es ist ein Migrationssystem zu entwickeln, welches die (originalen) VSAM-Daten eines alten Mainframe-Systems in ein neues Client/Server-System überführen soll. Der die Migration später ausführende Benutzer soll durch Reports und Statistiken über den Verlauf der Migration informiert werden. Entwickeln Sie aus dieser Information ein Kontextsicht-Diagramm.

2 In einem Finanzamt stehen zwei Server. Ein Programm „datenbank.exe“ greift auf Daten beider Server zu. Auf Server „s1“ befinden sich die zwei Applikationen „Hundesteuerverwaltung.app“ und „Grundsteuerveraltung.app“, während auf Server „s2“ die Applikation „Personalverwaltung.app“ vorliegt. Hundesteuerverwaltung.app sowie Grundstücksverwaltung.app und Personalverwaltung.app verarbeiten Information für Database.exe, wobei Information aus Hundesteuerverwaltung.app auch für Grundstücksverwaltung.app verwendet wird. Letzere beide Applikationen liefern auch Information für Personalverwaltung.app. Entwerfen Sie eine geeignete Verteilungssicht.

5. Komplexitätsprobleme

Wir haben bisher einige Architekturbeschreibungssprachen kennengelernt und wissen, wie und welche Architektursichten damit beschrieben werden können. Das Ziel dieses Unterfanges ist es, eine Architekturdokumentation zu entwerfen, die einerseits für alle Stakeholder die „passenden" Sichten enthält und mit der andererseits das geplante Projekt konkret in Software umgesetzt werden kann. Gerade bei Letzterem ist allerdings entscheidend, dass komplexe Zusammenhänge in weniger komplexe aufgelöst werden und dann jeweils „für sich" einer bewältigbaren Lösung zugeführt werden können.
Obwohl die Möglichkeiten, Komplexität zu erzeugen, so mannigfaltig wie die zu lösenden Probleme sind, kann man dennoch gewisse Grundsätze und Problemfelder spezifizieren und systematisch entflechten. Es würde den Rahmen dieses Buches sprengen, hier auf dieses große Fachgebiet der Komplexitätstheorie einzugehen; wir wollen statt dessen einige einfache Grundsätze und Grundprobleme nebst deren Lösungen in diesem Zusammenhang betrachten.
Wir beginnen damit, dass wir uns dem Komplexitätsbegriff in der Datenverarbeitung nähern.

Komplexität von Algorithmen
Wir wollen ausnahmsweise hier die Vorgehensweise „Bottom Up" zur Erfassung des Komplexitätsbegriffs benutzten. Der Grund hierfür ist, dass dem Leser vermutlich die Begriffswelt der Algorithmen bekannt ist. Dort kann man Komplexität recht scharf definieren und so ein Gefühl dafür bekommen, was man unter dem Begriff eigentlich versteht. Später verallgemeinern wir dann auf ein weniger scharfes Komplexitätsmaß.

Zunächst definieren wir den Begriff eines Algorithmus selbst:

Def. (*Algorithmus)*
Ein Algorithmus ist eine Abbildung von einer Menge *E* von Eingaben in eine Menge *A* von Ausgaben. Er ist eine systematische Methode zur Lösung eines Problems.

Anders ausgedrückt: Ein Algorithmus ist eine Rechenvorschrift, der Eingaben aus der Input-Menge E in Ausgaben, die zusammen die Outputmenge A darstellen, umwandelt. So eine Rechenvorschrift kann natürlich fehlerhaft sein, und daher fordert man (zumindest theoretisch), dass ein Algorithmus korrekt sein muss.

Def. (*semantische Korrektheit)*
Ein Algorithmus ist semantisch korrekt, wenn er folgende Bedingungen erfüllt:
(i) jeder Input $e \in E$, der eine festgelegte Eingabebedingung α erfüllt, wird auf eine Ausgabe $a \in A$ abgebildet, die eine vorgegebene Ausgabebedingung β erfüllt.
(ii) Der Algorithmus ist endlich

Betrachten wir hierfür folgendes Beispiel:

```
Function intdiv(x As Integer, y As Integer)
 Dim q As Integer
 Dim r As Integer
 q = 0
 r = x
 Do While y <= r
    q = q + 1
    r = r - y
 Loop
 Debug.Print "Quotient= ", q, "Rest= ", r
End Function
```

Interessant an diesem Beispiel ist, dass eine Division der beiden ganzen Zahlen x und y durchgeführt wird, ohne dass tatsächlich im Programm eine wirkliche Division ausgeführt wird. Es wird mit den Grundrechenarten Addition und Subtraktion ausgekommen. Die Frage, ob dieser Algorithmus nun semantisch korrekt ist, lässt sich so nicht beantworten, da semantische Korrektheit immer nur zusammen mit den Eingabebedingungen α den Ausgabebedingungen β Sinn macht. Daher betrachten wir folgende Bedingungen:

a) $x \geq 0 \wedge y > 0 \wedge x = q \bullet y + r \wedge 0 \leq r < y$
b) $x \geq 0 \wedge y \geq 0 \wedge x = q \bullet y + r \wedge 0 \leq r < y$
c) $x \geq 0 \wedge y > 0 \wedge x = q \bullet y + r \wedge 0 < r < y$

Zunächst ist zu identifizieren, welches die Eingabe- und welches die Ausgabebedingungen sind. Dies ist relativ einfach: An den Eingabebedingungen sind nur die Eingabevariablen beteiligt, während immer dann, wenn Ausgabevariable (mit) betroffen sind, es sich um die Ausgabebedingungen handelt.

Fall a)
Hier ist die Eingabebedingung: α: $x \geq 0 \wedge y > 0$
und die Ausgabebedingung: β: $x = q \bullet y + r \wedge 0 \leq r < y$

Dies entspricht dem, was man bei der Division positiver Zahlen mathematisch erwarten würde. *y* darf nicht Null sein, denn die Division durch Null ist nicht definiert. Wenn q dann den Quotienten und r den Rest darstellt, dann gibt Bedingung β genau die Bedingungen wider, die alle Eingabe- und Ausgabevariablen erfüllen müssen. Auch der Algorithmus macht hier keine Probleme und erfüllt die Anforderungen der semantischen Korrektheit.

Fall b)
Hier ist die Eingabebedingung: α: $x \geq 0 \wedge y \geq 0$
und die Ausgabebedingung: β: $x = q \bullet y + r \wedge 0 \leq r < y$
Man sieht, dass sich nun die Eingabebedingung geändert hat. *y=0* ist jetzt zugelassen. Mathematisch ist dies nicht zulässig, aber da im Algorithmus nur addiert und subtrahiert wird, könnte man zunächst vermuten, dass das nichts macht. Bei genauerer Betrachtung stellt man jedoch fest, dass die *do-while*-Schleife dann zu einer Endlosschleife wird. Betrachtet man z.B. die Werte *x=5* und *y=0*. Beim Eintritt in die Schleife besitzt *r* den Wert *5* und *y* den Wert *0*. Innerhalb der Schliefe wird dann wegen *r=r-y* der Wert von *r* aber nicht reduziert, da ja *y=0* ist. Damit bleibt die Schleifenbedingung, nämlich *y<=r*, hier also konkret: *0<=5*, immer gleich. Somit wurde eine Endlosschleife erzeugt. Daher ist der Algorithmus nicht mehr endlich, und damit nicht mehr semantisch korrekt. Lösen kann man das Problem auf zwei Arten: Entweder man ändert die Eingabebedingung ab oder man lässt die Eingabebedingung und ändert den Algorithmus. So könnte man z.B. eine Abfrage einbauen, ob *y=0* ist und in diesem Fall nicht in die Schleife verzweigen, sondern eine Meldung ausgeben und das Programm abbrechen. Damit wäre der Algorithmus wieder semantisch korrekt.

Fall c)
Hier ist die Eingabebedingung: α: $x \geq 0 \wedge y > 0$
und die Ausgabebedingung: β: $x = q \bullet y + r \wedge 0 < r < y$
Hier wurde nun die Ausgabebedingung gegenüber dem Fall a) abgeändert. Es ist jetzt nur zulässig, dass der Rest ungleich Null ist. Wenn man jedoch als Beispiel *x=8* und *y=4* wählt, was gemäß den Eingabebedingungen zulässig ist, dann endet der Algorithmus ganz normal und gibt die Zahlen *q=2* und *r=0* aus. Damit verstößt er jedoch gegen die Ausgabebedingung β und ist damit nicht semantisch korrekt. Auch hier kann man wieder entweder die Ausgabebedingung oder den Algorithmus abändern um ihn semantisch korrekt zu machen (z.B. kann man im Algorithmus abfragen, ob *r=0* ist und in diesem Fall die Ausgabe von *q* und *r* verhindern und stattdessen eine entsprechende Meldung ausgeben).
Bevor wir den Begriff der Komplexität näher definieren, soll zuerst an einem Beispiel diese Problematik genauer betrachtet werden. Hierzu untersuchen wir das Problem eines Handlungsreisenden (Travelling Salesman Problem, TSP).

Ein Handlungsreisender soll n Städte besuchen und dabei den kürzesten Weg wählen, so dass die Fahrtkosten minimal bleiben. Um dieses Problem exakt zu lösen, müsste man alle $½(n-1)!$ Möglichkeiten durchprobieren. Ein Algorithmus, der dieses Programm auf einem Computer lösen soll, könnte daher für jede dieser Möglichkeiten die Weglänge errechnen und die kürzeste ausgeben. Dieser relativ einfach Algorithmus besitzt in dem noch genau zu definierenden Sinn allerdings eine extreme hohe algorithmische Komplexität. Schon für 30 Städte würde die Rechenzeit selbst auf dem schnellsten Computer der Welt noch das Alter des Universums übersteigen. Dies zeigt deutlich, wie wichtig es ist, insbesondere für Realzeitanwendungen, die Komplexität, welche direkten Einfluss auf die Rechenzeit hat, zu verkleinern. Um dafür ein Maß zu finden, betrachten wir die sogenannten elementaren Operationen eines Algorithmus. Diese sind:

- Wertzuweisungen
- +,-,*,:,<,=,>,and, or, not etc.
- I/O-Statements
- Stop/End/Calls

Damit lässt sich nun der Begriff der totalen Kostenfunktion definieren.

Def. (*totale Kostenfunktion*)
Sei $K:E \rightarrow A$ ein Algorithmus, der eine Eingabemenge E auf eine Ausgabemenge A semantisch korrekt abbildet. Sei $e \in E$ eine Eingabe, auf die K angewendet wird. Weiter sei $t:e \rightarrow \boldsymbol{N}$ eine Abbildung der Eingabe e in die Menge der natürlichen Zahlen $\boldsymbol{N}$, welche die Anzahl der elementaren Operationen, die bei der Anwendung von K ausgeführt werden, ermittelt. $t(e)$ heißt dann die totale Kostenfunktion von K bezüglich e.

Die totale Kostenfunktion stellt also ein Maß dar, wie viele Operationen ein Programm durchführt, was ja direkt mit der Rechenzeit korrespondiert. In den Übungen am Ende des Kapitels wird dazu ein Beispiel untersucht.

Def. (*Ordnung einer Funktion)*
Es seien zwei Funktionen f und g auf einer Menge $X \subseteq \boldsymbol{R}$ gegeben, wobei $\boldsymbol{R}$ die Menge der reellen Zahlen darstellt. Dann heißt:

(i) f ist **höchstens** von der Ordnung g, $f=O(g)$, falls es eine Konstante $c>0$ gibt, so dass $|f(x)| \leq c \cdot |g(x)|$ für alle $x>x_0 \in X$.

(ii) f ist **mindestens** von der Ordnung g, $f=\Omega(g)$, falls es eine Konstante $c>0$ gibt, so dass $|f(x)| \geq c \cdot |g(x)|$ für alle $x>x_0 \in X$.

(iii) f ist genau von der Ordnung g, $f=\Theta(g)$, falls $f=O(g)$ und $f=\Omega(g)$.

Die Ordnung einer Funktion gibt ihr asymptotisches Verhalten im Vergleich zu einer Funktion g an. Man erkennt daran, dass eigentlich nicht die exakte Anzahl an ausgeführten elementaren Operationen für die Kostenfunktion erforderlich ist, sondern dass es bereits ausreicht, die Größenordnung zu kennen.

Def. (*Größe der Eingabe)*
Die Größe $|e|$ einer Eingabe $e \in E$ ist eine Zahl oder ein Tupel, welche(s) ein sinnvolles Maß für die Eingabe bezüglich seiner Verarbeitung durch einen Algorithmus darstellt. Die Größe der Eingabe wird auch Inputgröße genannt und ist eine Funktion von den die Eingabe festlegenden Parametern, also $|e|=g(e)$.

Betrachten wir z.B. eine Matrixmultiplikation zweier Matrizen A und B, wobei A eine n*m-Matrix und B eine m*k-Matrix sei. Dann könnte eine Eingabe sein: $e= (n,m,k,A,B)$. Für die Inputgröße könnten sinnvoll sein:

$g(e)=|e|=(n,m,k)$ oder auch
$g(e)=|e|=max(m,n,k)$ bzw.
$g(e)=|e|=m \bullet n \bullet k$

Welche der Inputgrößen man wählt, hängt von der Art des Algorithmus ab. Benutzt man z.B. einen Algorithmus, der hintereinander die übliche Zeilen-mal-Spalten-Multiplikation durchführt, so erscheint als Maß das Produkt mnk sinnvoll, während z.B. bei der Nutzung eines Parallelprozessors gleichzeitig alle Zeilen und Spalten der Matrizen verarbeitet werden können, so dass hier $max(m,n,k)$ oder (m,n,k) ein sinnvolles Maß darstellt.
Ist die Größe des Inputs einmal definiert, so kann man damit schließlich den Begriff der Komplexität eines Algorithmus genauer fassen.

Def. (*Zeitkomplexität eines Algorithmus*)
Sei E die Menge der Eingaben und $g(e)$ eine Inputgröße. Die Funktion $T(g):=sup\{t(e) \mid e \in E \wedge |e|=g(e)\}$ heißt Zeitkomplexität oder Worse-Case-Kostenfunktion des Algorithmus.

Bei der Zeitkomplexität handelt es sich also um die ungünstigste aller Möglichkeiten für einen Algorithmus über alle möglichen Eingaben. Die Ordnung der Kostenfunktion stellt gleichzeitig auch die Ordnung der Zeitkomplexität dar.
Der Vollständigkeit halber soll hier noch die Definition der Raumkomplexität angegeben werden. Sie stellt ein Maß für den durch einen Algorithmus maximal belegten Speicherplatz in einem Rechner dar. Allerdings spielt die Raumkom-

plexität heutzutage wegen der geringen Kosten der Speichermedien kaum noch eine Rolle.

Def. (*Raumkomplexität eines Algorithmus*)
Sei E die Menge der Eingaben, $g(e)$ eine Inputgröße und $s(e)$ die maximale Speicherbelegung einer Eingabe $e \in E$. Die Funktion $S(g):=sup\{s(e) \mid e \in E \wedge |e|=g(e)\}$ heißt Raumkomplexität des Algorithmus.

Der hier eingeführte Komplexitätsbegriff wurde schwerpunktmäßig auf Algorithmen bezogen. Da alles, was in der EDV formalisiert wird, letztlich algorithmisch beschreibbar ist, hat man damit zumindest prinzipiell eine Methode, Komplexität zu quantifizieren.
Aus verschiedenen Gründen ist jedoch die Anwendung dieser Komplexitätsbegriffe beispielsweise auf den Entwurf statischer Klassen im Rahmen von UML kaum praktikabel. Will man Komplexitäten im abstrakteren Vorfeld bestimmten, so sollte man ein pauschaleres (und damit allerdings auch nicht so scharfes) Komplexitätsmaß verwenden.

Software-Kategorien
Ein erster Schritt zur Bewältigung von Komplexität im allgemeineren Sinne ist die Bildung von (weniger komplexen) Kategorien.
Reussner und Hasselbring schlagen eine Unterteilung in die Software-Kategorien „0-Software“, „A-Software“, „T-Software“ und „R-Software“ vor[32], wobei Kombinationen möglich sind. Es wird dabei die „Separation of Concern“ (Trennung von Zuständigkeiten) als Kriterium benutzt. Bevor wir diese Kategorien im Einzelnen erklären, wollen wir zunächst ein paar allgemeine Dinge zu Software-Kategorien sagen.
Um eine saubere Aufteilung in Software-Kategorien vornehmen zu können, müssen diese bestimmte Bedingungen einhalten:

- Kategorien sind **teilgeordnet** (d.h. jede Kategorie kann eine oder mehrere andere Kategorien verfeinern)
- Kategorien sind in **zyklenfreien Graphen** anordenbar (die Wurzelkategorie wird Kategorie 0 genannt – das allgemeine Grundwissen, das überall vorhanden ist und dessen Verwendung keine unerwünschten Abhängigkeiten erzeugt)

[32] in R. Reussner u. W. Hasselbring: *Handbuch der Software-Architektur*, dpunkt.verlag 2006, Seite 92ff

- Kategorien sind **rein**, wenn es im Kategoriegraphen genau einen Weg von dieser Kategorie zur Wurzelkategorie gibt; sonst heißt die Kategorie **unrein**

Reine Kategorien entstehen damit durch stückweise Verfeinerung der Kategorie 0; unreine Kategorien vermengen zwei oder mehr Kategorien.
Jede Komponente bzw. Schnittstelle einer Architektur gehört immer zu einer oder mehreren Kategorien, wobei das Ziel ist, dass davon möglichst wenige Kategoriezugehörigkeiten vorhanden sind. Diese Kategorien sollten im Kategoriengraph auch möglichst weit „unten" stehen: Je komplizierter oder spezieller ein Thema ist, desto weniger Software sollte sich damit befassen („Be as simple as possible").
Diese Forderungen reichen bereits aus, um ein einfaches Komplexitätsmaß herzuleiten. Wie setzten dazu voraus, dass jede einfache Komponente genau eine Kategorie (rein oder unrein) besitzt (was im Prinzip immer erzwungen werden kann). Hierzu definieren wir:

- Die Komplexität einer reinen Kategorie ist die Anzahl ihrer Vorfahren im Kategoriegraphen
- Die Komplexität einer unreinen Kategorie ist gleich der Summer der Komplexitäten der Kategorien, die sie vermengt
- Die Komplexität einer einfachen Komponente ist gleich der Komplexität der (einzigen) Kategorie, der sie zugehört
- Die Komplexität einer zusammengesetzten Komponente ist gleich dem gewichteten Mittel der Komplexitäten ihrer Subkomponenten

Offenbar hat die Kategorie 0 auch die Komplexität 0. Dieser Idealfall ist aber in der Realität höchst selten möglich.
Wir wollen dazu ein Beispiel betrachten[33]:
Es sei eine Anwendung zu schreiben, welche mit geeigneter Grafik die Möglichkeit des Skatspielens simuliert. Bekannt braucht es dazu 3 Spieler. Die Aufgabe der zu entwickelnden Software besteht darin, ein System zu erzeugen, mit dessen Hilfe drei reale oder virtuelle Spieler Skat spielen. Dabei liegt die Anzahl n der realen Spieler zwischen 0 und 3. Bei n = 3 ist das System nichts weiter als ein intelligenter Spieltisch, der den Spielverlauf protokolliert und gespielte Spiele speichert; mit n = 0 kann man Spiele simulieren, etwa um verschiedene Strategien zu vergleichen. Welche Software-Kategorien gibt es hier? Ein großer Teil der Software weiß, dass es sich um ein Kartenspiel mit einer festen Anzahl von Personen handelt, die der Reihe nach drankommen. Da gibt es zunächst eine Kategorie *Kartenspiel*, die für Skat, Rommee, Schafskopf und vergleichbare Spiele gilt. Sie wird spezialisiert durch die Kategorie *Skat*, in der die Regeln des

[33] In Abwandlung von ibid., S. 90

Skatspiels bekannt sind. Die Skatregeln werden sich so schnell nicht ändern, aber es sind viele verschiedene Strategien denkbar, und deshalb ist das eine eigene Kategorie. *Skatstrategie* ist eine Verfeinerung von *Skat*, denn eine Skatstrategie kann man nur in Kenntnis der Skatregeln formulieren. Nun zur Visualisierung: Dort gibt es zunächst die Kategorie *Kartenspiel- GUI*, wo man unabhängig von der GUI-Bibliothek festlegt, wie ein Kartenspiel am Bildschirm aussieht: in der Mitte die ausgespielten Karten, an den drei Seiten die Blätter der Spieler; jeder Spieler sieht nur sein eigenes Blatt, die der anderen Spieler sind verdeckt (vgl. Abb. 5.1).

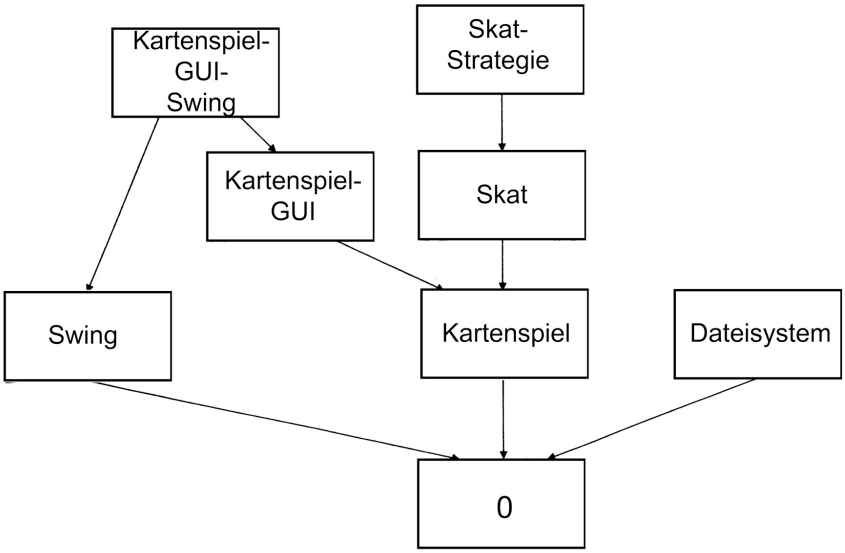


Abb. 5.1 Skat-Kategorien

Es liegt nahe, die verwendete GUI-Bibliothek (im Beispiel *Swing*) um einige Darstellungsmöglichkeiten zu erweitern, z.B. um die Spielkarte und das aus einer oder mehreren Karten bestehende Blatt. So erhalten wir eine weitere Kategorie, nämlich „*Kartenspiel-GUI-Swing*“, wo wir zwar Swing und Kartenspiele kennen, aber weder Skat noch Rommé. Wenn die stattgefundenen Spiele gespeichert werden, dann brauchen wir als weitere Kategorie noch *Dateisystem*; dort ist die Software, die Daten abspeichert und wieder einliest. So erhalten wir Kategorien aus der Anwendung (Skat, Skatstrategie), aus der Technik (Dateisystem) und eine Kombination (Kartenspiel-GUI-Swing). Grundlage jeder Kategorie ist die Kategorie 0, zu der alles gehört, was ohne weiteres im gesamten System zur Verfügung steht – z.B. in Java sind das normalerweise die Pakete java.lang und java.util. Kategorien helfen bei der Suche nach Komponenten, in-

dem man sich die Kategorien der Reihe nach anschaut und überlegt, welche Komponenten und welche Schnittstellen man in der jeweiligen Kategorie unterbringen könnte.
Nun zurück zu der Einteilung von Software in die Kategorien 0, A, T und R. Software der Kategorie 0 haben wir schon besprochen.
A-Software ist Software, die zu Kategorie „Anwendung" gehört. Sie ist unabhängig von der benutzen Technik. Viele Business-Informationssysteme zählen hierzu.
T-Software ist Software, die zur Kategorie „Technik" zu zählen ist. Diese Software ist unabhängig von der Anwendung, wohl aber von der benutzten Technik. Typischerweise sind hier APIs zu nennen.
R-Software ist Software, die zur Kategorie „Repräsentation" gehört. Darunter versteht man Programme, die lediglich Daten transformieren und dabei keiner „innerer" Ablauflogik folgen. Solche Software ist stereotyp und lässt sich leicht generieren. Beispiele hierfür sind Format-Konverter und Datenbank-Mapper.

Feature-Delokalisierung
Eine weitere Methode, Komplexität in den Griff zu bekommen, ist die Methode der Feature-Delokalisierung[34]. Frühe objektorientierte Methoden nahmen fälschlicherweise an, dass jedes Objekt eines Problems durch ein Objekt in der Lösung repräsentiert werden kann. Das heißt man nahm an, dass sich Problemfeatures direkt Lösungsfeatures zuordnen ließen. Jedoch ist es meistens so, dass die Problemfeatures verstreut und mit mehreren Lösungsfeatures vermischt sind.
Um beispielsweise die Implementierung einer Komponente „Produkt" in einem Software-System zu ändern, müssen alle damit verwobenen Komponenten geändert werden, was auch die Implementierungen von Modulen wie z.B. „Bestellung" und „Kunde" betrifft. Dieses Phänomen wird als *Feature-Delokalisierung* bezeichnet und ist eine Hauptquelle für externe Komplexität. Sie tritt auf, wenn mehrere Lösungsfeatures wie Web Server Pages, Disributed Component und Database Connectivity ein einzelnes Problemfeature wie „Produkt" implementieren, oder wenn ein einzelnes Lösungsfeature wie WebServerPages mehrere Problemfeatures wie *Produkt*, *Bestellung* und *Kunde* implementiert. Problemfeatures werden üblicherweise durch Anforderungen definiert, Lösungsfeatures dagegen durch Implementierungen.
In den **Anforderungen** sind Problemfeatures am stärksten lokalisiert, Lösungsfeatures dagegen am stärksten delokalisiert. So wird z.B. eine Bestellung durch eine einzelne Analyseklasse definiert, Web Server Pages zusammen mit Security, Distributed Component und Database Connectivity dagegen durch mehrere Analyseklassen.
In **Implementierungen** sind umgekehrt Lösungsfeatures am stärksten lokali-

[34] Wir folgend hier J. Greenfield et. all, *Software-Factories*, Wiley Publishing 2004, Seite 49f

siert, die Problemfeatures dagegen am stärksten delokalisiert. Beispielsweise wird WebServerPages durch eine einzelne Baugruppe definiert, dagegen ist *Bestellung* zusammen mit *Kunde* und *Produkt* über mehreren Baugruppen verstreut.
Die Delokalisierung von Lösungsfeatures kann man oft bereits an den Anforderungen ablesen, während man die Delokalisierung von Problemfeatures häufig an der Beschreibung von Implementierungen ablesen kann. Die Implementierung für die Komponenten eines Bestellsystems könnte beispielsweise Webseiten für *Produkt*, *Bestellung* und *Kunde*, verteilte *Produkt*-, *Bestellung*- und *Kunden*-Komponenten sowie eine SQL-DDL-Datei enthalten, welche die *Produkt*-, *Bestellung*- und *Kunden*-Tabellen beschreibt. Wenn natürlich das Problem nicht sauber definiert ist oder von den Anforderungen nicht ausreichend modelliert wird, können sogar die Problemfeatures in den Anforderungen schlecht lokalisiert sein. Analoges gilt für die Lösung: Bei einem schlechten Lösungsentwurf können die Lösungsfeatures sogar in der Implementierung schlecht lokalisiert sein.
Feature-Delokalisierung verursacht zwei wohlbekannte Programmierprobleme: das Traceability-Problem und das Rekonstruktionsproblem.
Das **Traceability-Problem** (Problem der Nachvollziehbarkeit) tritt auf, wenn die Lösungsfeatures, die ein Problemfeature implementieren, nicht leicht identifiziert werden können. Dadurch wird es schwierig, alle Dinge zu identifizieren, die in der Implementierung geändert werden müssen, und zu gewährleisten, dass alle Änderungen konsistent vorgenommen werden, wenn sich die Anforderungen ändern.
Das **Rekonstruktionsproblem** tritt auf, wenn Entwurfsinformationen verloren gehen und nicht leicht rekonstruiert werden können, wodurch Wartung und Verbesserungen erschwert werden. Lösungen werden gemeinhin anhand von Entwurfsentscheidungen konzipiert, die auf bestimmten Überlegungen basieren. Diese Informationen werden normalerweise während der Entwicklung gut verstanden, aber wenn die Entwicklung beendet ist, werden sie selten anderswo als im Gedächtnis der Beteiligten aufbewahrt. Menschen sind mobil und vergessen im Laufe der Zeit Einzelheiten, so dass die Informationen leicht verloren gehen. Es ist dann ziemlich schwierig, schlecht dokumentierte Software zuversichtlich zu modifizieren, und es kommt ziemlich häufig vor, dass ein Feature, lange nachdem es nicht mehr benötigt wurde, beibehalten wurde, weil niemand wusste, warum es hinzugefügt worden war oder ob es sicher entfernt werden konnte.

Kapselung und Domain Driven Design[35]
Wie schon erwähnt, stellt die Zerlegung eines komplexen Systems in mehrere, weniger komplexe Systeme eine bewährte Methode zur Bewältigung von Kom-

[35] Wir folgen hier wieder Starke, G.: *Effektive Software-Architekturen*, Hanser, 2008, Seite 143ff

plexität dar. Letztendlich ist das Ziel, eine verständliche Software-Architektur zur erhalten, immer durch die Lösung „Kapselung“ erreichbar. Doch wie kapseln? Dazu gibt es oft viele Möglichkeiten: Software-Kategorien und Feature-Delokalisierung sind gute Ansatzpunkte hierzu und auch die Fachdomänen wie die genannten Kategorien A und T (bzw. R). Dabei ist es sinnvoll, in der Sprache der jeweiligen Fachdomäne zu modellieren. Auf der Basis eines gemeinsam abgestimmten Domänenmodells ergibt sich auch eine gemeinsame Sprache, die manchmal „ubiquitous language“ genannt wird. Das Ganze nennt man auch Domain Driven Design (DDD). Diese Entwurfsmethode ist durch folgendes charakterisiert:

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 nicht über einen eigenen Zustand verfügen. Parameter und Ergebnisse dieser Operationen sind Domänenobjekte (Entities oder Value-Objects).

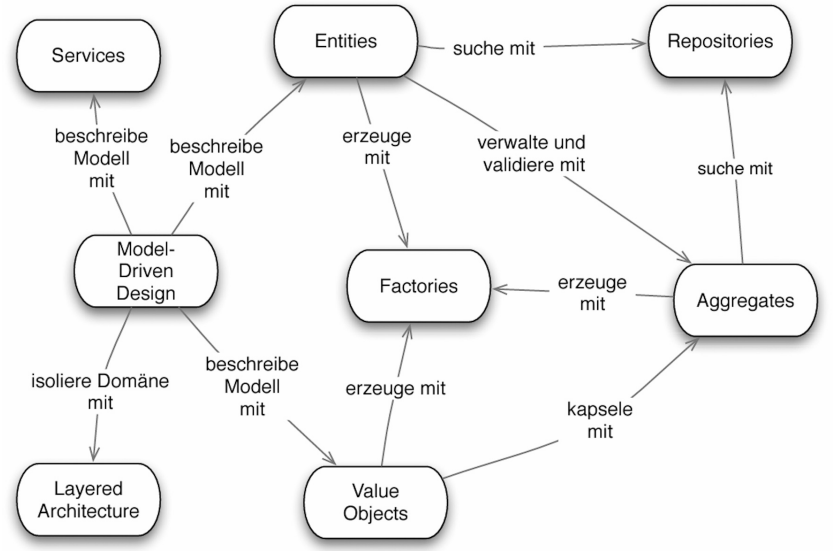


Abb. 5.2 Domain Driven Design[36]

[36] Quelle: ibid., S. 142

Es werden dabei folgende Schichten propagiert[37]:

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“ 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.

Abb. 5.3 Domain Driven Design-Schichten

Die Begriffe „Aggregate“, „Factories“ und „Repositories“ sind dabei wir folgt beschrieben:

Aggregate:
Sie 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 enthalten.
Factories:
Entities und insbesondere Aggregate können komplexe Strukturen vernetzter Objekte bilden, die Sie nicht über triviale Konstruktoraufrufe erzeugen können oder wollen. Factories werden verwendet, 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:
Alle Arten von Objekten (sowohl aus dem Domain Layer wie auch dem Application Layer) benötigen eine Möglichkeit, die Objektreferenzen anderer Objekte zu erhalten. Repositories kapseln die technischen Details der Infrastrukturschicht gegenüber den Domänenobjekten. Dadurch bleibt das Domänenmodell auch in dieser Hinsicht „technologiefrei“. Repositories beschaffen beispielsweise Objektreferenzen von Entitäten, die aus Datenbanken gelesen werden müssen.

[37] Quelle: ibid., S. 143

Übungsaufgaben:

1. Sei $K:E \rightarrow A$ ein Algorithmus, der eine Eingabemenge E auf eine Ausgabemenge A semantisch korrekt abbildet. Sei $e \in E$ eine Eingabe, auf die K angewendet wird. Weiter sei $t:e \rightarrow \mathbf{N}$ eine Abbildung der Eingabe e in die Menge der natürlichen Zahlen $\mathbf{N}$, welche die Anzahl der elementaren Operationen, die bei der Anwendung von K ausgeführt werden, ermittelt. *t(e)* bezeichne die totale Kostenfunktion von K bezüglich e. Bestimmen Sie die totale Kostenfunktion *t(n)* des folgenden Algorithmus für die Fälle
   a) n ist Primzahl:
   b) n ist Quadrat einer Primzahl
   c) n ist gerade:

```
Function PrimeCheck(n As Integer)
Dim p As Integer
Dim b As Boolean
p = 2
b = True
Do While p * p <= n And b
    If n Mod p = 0 Then b = False
    p = p + 1
Loop
PrimeCheck = b
```

2. Geben Sie mind. zwei Beispiele für Softwaren der Kategorie 0

3. Geben Sie ein Sequenzdiagramm an, welches das Zusammenwirken von Factory und Repository im Domain Driven Design an folgendem Beispiel widerspiegelt: Ein Client kreiere einen Customer mit Namen SQ42 und wird von der Factory darüber informiert. Der neue Customer wird in das Repository aufgenommen und danach in die Datenbank.

4. In Abb. 5.1 wurden die Software-Kategorien eines virtuellen Skatspiels dargestellt. Geben Sie Komplexität jeder Kategorie an!

6. Architektur-Muster und Standards

In diesem Abschnitt wollen wir uns den Möglichkeiten zuwenden, wie Architekturen im Allgemeinen klassifiziert und strategisch eingesetzt werden können.
Es geht also darum, Strategien zu entwickeln, wie man die verfügbaren Komponenten der Architektur geeignet klassifiziert und beschreibt, so dass diese im „Architekturmanagement“ in einer geeigneten Architekturdokumentation beschrieben werden können. Während Entwurfsmuster –wie das Wort schon sagt- Teile das Systementwurfs darstellen, sind Architekturmuster weiter gefasst. Sie beschreiben die Struktur von Systemen als Ganzes. Manchmal wird ein Architekturmuster auch Architekturstil genannt.

Schichten und Tiers
Die Einteilung von Architekturmustern in Schichten (engl. Layers) wurde dem bekannten OSI-Schichtenmodell der ISO, welches insbesondere Kommunikationsaspekte hervorhebt, nachempfunden. Jede Schicht kann man sich als „virtuelle Maschine“ vorstellen, welche der darüber liegenden Schicht bestimmte Dienste anbietet. Schichten sind damit „Black-Boxes“, da sie die Details ihrer eigentlichen Implementierung kapseln. Analog kapseln die Komponenten einer Schicht auch die darunter liegenden Schichten in dem Sinn, dass z.B. technische oder physikalische Aspekte verborgen sind. Wie bei einer Zwiebel verbergen also obere Schichten die unteren. Schichten können auch übersprungen werden, wenn eine obere Schicht einen Dienst aus einer weiter unten liegenden benötigt. So etwas nennt man dann „Layer Bridge“ (vgl. Abb. 6.1).

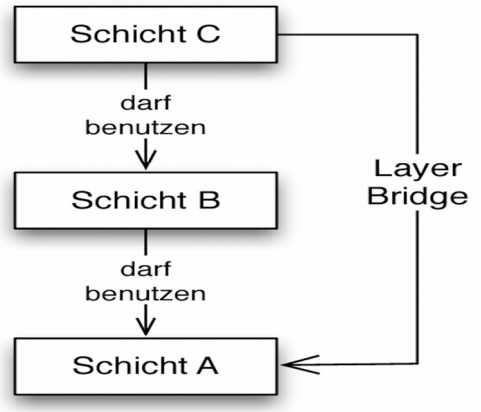


Abb. 6.1 Layer Bridge[38]

[38] Quelle: Starke, G.: *Effektive Software-Architekturen*, Hanser, 2008, Seite 147

Sinnvoll ist auf jeden Fall, in der Praxis wenigstens folgende Schichten zu haben:

1. Präsentationsschicht
2. Applikationsschicht
3. Business Architecture (Fachdomäne)
4. Technical Architecture (Infrastruktur)

Die 4. Schicht unterteilt man manchmal weiter z.B. in Persistenz und Datenhaltung, Schnittstellen zu Hardware, Integration von Fremdsystemen, Sicherheit, Kommunikation und Verteilung.
Außer den Schichten gibt es auch noch den Begriff der sogenannten „Tiers". Man redet manchmal z.B. von „n-Tier-Architekturen". Eigentlich unterscheiden sich Tiers und Schichten im Wesentlichen nur durch 2 Aspekte: Während Schichten in der Regel Information nur in eine Richtung (z.B. von Oben nach unten) abrufen (es gibt Ausnahmen!), ist bei Tiers der Informationsfluss in beide Richtungen üblich. Dann werden Tiers meistens nur für Software-Komponenten benutzt. Ein Beispiel sehen wir in Abb. 6.2.

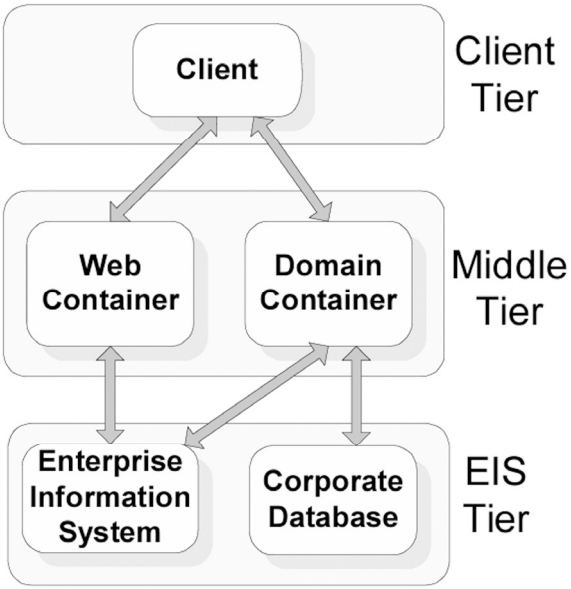


Abb. 6.2 Beispiel eines 3-Tier-Systems für das Web[39]

[39] Quelle: ibid., S. 148f

Filter und Pipes
Bei zeitdynamischer Betrachtungsweise bestimmter Ereignisfolgen kann es erforderlich sein, dass abhängig von vorgegebenen Bedingungen eine „Kanalisierung“ bzw. eine Filterung erfolgen muss.
Ein einfaches Beispiel dazu findet man in der „Umleitung“ der Bildschirmausgabe in eine Datei: Wenn man im DOS-Fenster den Befehl

```
C:\Dir *.exe
```

eingibt, so werden bekanntlich alle Dateien im aktuellen Verzeichnis mit der Endung „exe“ auf den Bildschirm gebracht. Möchte man diese Information in eine Textdatei schreiben, so kann folgendes eingeben:

```
C:\Dir *.exe > a.txt
```

Danach existiert eine Datei a.txt mit dem Inhalt, der zuvor auf dem Bildschirm zu sehen war.
Dieses Beispiel zeigt zweierlei: Einerseits wurde eine Filterbedingung (*.exe) angegeben und andererseits wurde das Ergebnis in eine Textdatei umgeleitet. Letzteres könnte man im weitesten Sinne als eine „Pipe“ bezeichnen. Eine Verallgemeinerung dieses Sachverhalts führt zum sog. „Pipe & Filter“-Architekturmuster. Starke demonstriert dies recht gut am Beispiel einer Digitalkamera[40] (siehe Abb. 6.3):

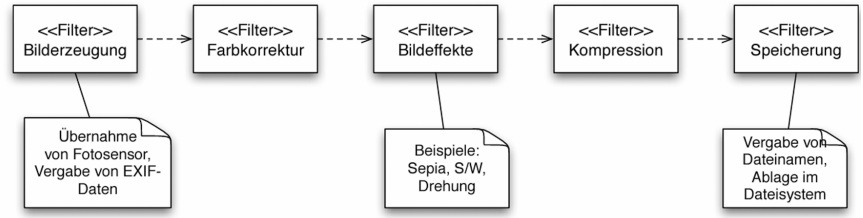


Abb. 6.3 Pipes & Filter am Beispiel einer Digitalkamera

In Abb. 6.3 ist zu sehen, dass jeder Filter (hier kann der Begriff wörtlich genommen werden) sein Ergebnis aktiv an den nächsten Filter weiterreicht, und zwar jeweils über eine Pipe (die gestrichelten Linien symbolisieren dies). Man sieht auch, dass Pipes nicht als eigene Komponenten auftreten.

Es gibt nun verschiedene Möglichkeiten, wie Filter und Pipes zusammenarbeiten:

- jeder Filter beendet seine Aufgabe komplett, bevor sein Ergebnis aktiv an eine Pipe übergeben wird, welche es dann zum nächsten Filter leitet.

[40] Ibid., S. 149

- Die Filter beenden nicht komplett ihre Aufgabe, sondern leiten schon Teilergebnisse über die Pipes an den nächsten Filter. Dies ermöglicht ein Parallelprocessing und steigert so u.U. die Verarbeitungsgeschwindigkeit erheblich.
- Die Pipe erfragt das nächste Ergebnis bei ihrem Eingangsfilter und übergibt es an den Ausgangsfilter.
- Eine zentrale Steuerung beaufsichtigt und koordiniert dabei das gesamte Zusammenwirken. In Abb. 6.4 kann man dies wieder am Beispiel der Digitalkamera nachverfolgen.

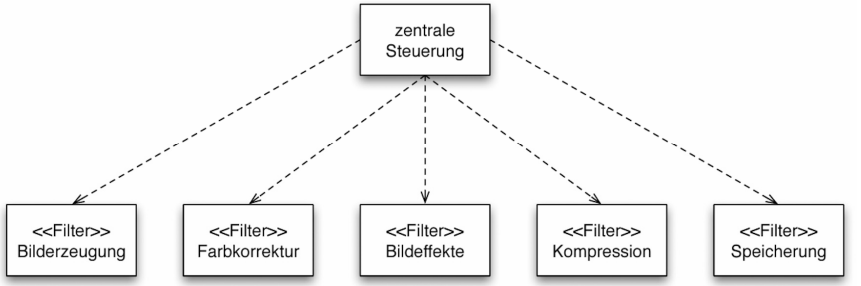


Abb. 6.4 Pipes & Filter mit Zentraler Steuerung am Beispiel einer Digitalkamera[41]

Wir sehen also, dass Pipes eigentlich nichts anderes als Datenkanäle sind. Man kann sie auch als Puffer zwischen den Filtern einsetzten, falls eine zeitversetzte Weitergabe angemessen ist. Man kann sogar ganze Datenbanken in dem Sinn als Pipes auffassen. Pipes stellen damit eine Art Mechanismus zur Entkopplung für Filter dar. „Intelligente“ Pipes entscheiden zum Beispiel, an welche Instanz eines Filters sie ihre aktuellen Daten weitergeben. Dadurch können Pipes also auch kapseln.
Pipes findet man aber auch in Software, z.B. in den einzelnen Phasen eines Compilers. Dort kann man Lexer, Parser und Maschinencode-Generator als Filter auffassen, welche ihre Ergebnisse jeweils durch Pipes weiterreichen.
Nachteilig bei der Verwendung dieses Architekturmusters kann sein, dass sich die Filter nicht „kennen“, was bei der Fehlerbehandlung problematisch werden kann: Folgefehler können oft nicht erkannt bzw. behoben werden. Da Filter keinen gemeinsamen Zustand kennen, wird dies noch erschwert.
Auch die gesamte Konfiguration der Verarbeitungskette gestaltet sich u.U. als schwierig. Bei komplexen Gebilden sollte man da auf die zentrale Steuerung vornehmlich zurückgreifen.

[41] Quelle: ibid.

Blackboard-Achitekturmuster

Blackboard-Systeme gehen aus einem Ansatz der Künstlichen Intelligenz (KI) hervor. Im Unterschied zu klassischen KI-Methoden (Vorwärts- und Rückwärtsschließen) wird hier das sog. „Opportunistische Schließen“ zur angewendet[42]. Bei jedem Schritt einer Problemlösung wird aufgrund der aktuellen Situation entschieden, wie weiter verfahren wird. In diesem Sinne sind Blackboard-Systeme nicht deterministisch, weil sie für unterschiedliche Eingabedaten ganz unterschiedliche Lösungsstrategien verfolgen können. Oft werden daher Blackboard-Architekturen dort eingesetzt, wo ein deterministischer Lösungsansatz für ein Problem nicht existiert oder nicht bekannt ist. Im letzteren Fall dient der Blackboard-Ansatz auch oft dazu, eine solche deterministische Lösungsstrategie zu finden. Der Begriff Blackboard-Architektur wurde zum ersten Mal 1962 von Allen Newell vorgestellt. Der Begriff leitet sich von der Metapher einer Kreidetafel (engl. blackboard) ab, um die eine Reihe von Experten sitzt, die gemeinsam ein Problem lösen müssen, indem sie die Tafel als Medium zur Kommunikation benutzen. Ein „Software-Blackboard-System“ besteht analog aus so einem Blackboard, d.h. einer gemeinsam zugänglichen Datenstruktur, die das Problem und vorhandene Teillösungen repräsentiert, sowie aus diversen Experten, auch Wissensquellen (knowledge sources) genannt. Jede dieser Wissensquellen hat Detailwissen zu einem bestimmten Bereich. Zur Steuerung der Experten existiert eine Kontrolle, die die Experten der Reihe nach befragt und so den Zugriff auf das Blackboard regelt. Der Inhalt des Blackboards ist meistens strukturiert, wobei eine beliebige Methode zur Wissensrepräsentation (z.B. Frames oder Semantische Netzwerke) verwendet werden kann. Voraussetzung dabei ist allerdings, dass alle beteiligten Module des Systems diese Methode der Repräsentation kennen. Blackboard-Architekturen wurden Anfang der 70er Jahre zum ersten Mal im „Hearsay-Projekt“ zur Spracherkennung eingesetzt.

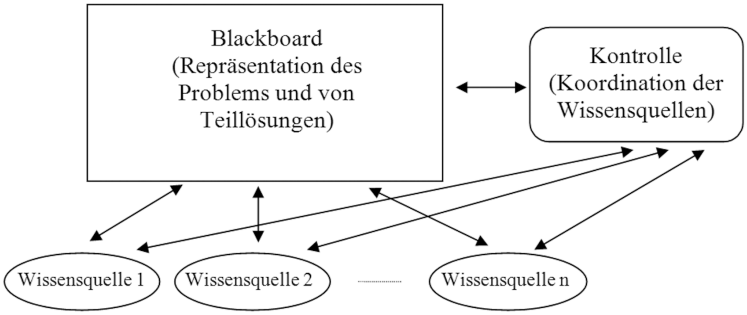


Abb. 6.5 Blackboard-Architekturmuster

[42]http://www.cis.uni-muenchen.de/projects/Irgroup/sauter99_full.doc

Peer-to-Peer-Architekturmuster

Dieses wichtige Architekturmuster spielt im Zusammenhang mit Net-Applikationen eine große Rolle (Napster, Freenet etc.). Eine ausführliche Besprechung dieses Musters findet sich z.B. im "Handbuch der Software-Architektur[43]". Dort wird das Peer-to Peer-Muster wie folgt definiert:

Peer-to-Peer (P2P) ist ein Kommunikationsmodell gleichberechtigter Einheiten (Peers), die direkt oder indirekt miteinander kommunizieren.

Die Gleichstellung von Peers wird erreicht, indem ein Peer sowohl Client- als auch Server-Funktionalität übernehmen kann. Man bezeichnet so ein Peer deshalb auch als Servent (abgeleitet von *Serv*er und Cli*ent*). Meistens ist ein Peer einfach nur eine Software-Applikation, die als Servent agiert. Damit ist eine Peer-to-Peer-Applikation ein Softwareprogramm, welches die Funktionalität eines einzelnen Peers implementiert. Es sind aber auch Umsetzungen in Hardware denkbar.

Peer-to-Peer-Netzwerke realisieren die Verbindungen zwischen den einzelnen Peers. Die Peers werden damit im Sinne des Peer-to-Peer-Kommunikationsmodells vernetzt. In Anlehnung an die Graphentheorie werden Peers häufig auch als Knoten bezeichnet. Peer-to-Peer-Netzwerke sind meist virtuelle Netzwerke, die auf einem physikalischen Netzwerk implementiert sind. Wenn Peers direkt über das Netzwerk verbunden sind, werden sie auch „Nachbarn" genannt. Wie bei allen Netzwerken braucht man auch hier Netzwerkprotokoll, Peer-to-Peer-Protokoll genannt. Dieses besteht aus der Menge der Vereinbarungen, nach denen die Kommunikation in einem Peer-to-Peer-Netzwerk erfolgt. Das Peer-to-Peer-Netzwerk wird von Peer-to-Peer-Systemen zur Umsetzung ihrer Anforderungen genutzt. Unter einem Peer-to-Peer-System versteht man dabei ein verteiltes System, dem das Peer-to-Peer-Kommunikationsmodell zugrunde liegt. Eine Peer-to-Peer-Architektur beschreibt nun die Architektur des Peer-to-Peer-Systems und umfasst nicht nur die Architektur einer einzelnen Anwendungen, sondern auch Aspekte der Netzwerktopologie.

Nun lassen sich Peer-to-Peer-Architekturen unter verschiedenen Aspekten klassifizieren. Das Peer-to-Peer-Kommunikationsmodell kann durch verschiedene Architekturen umgesetzt werden. Solche Architekturen werden in erster Linie danach klassifiziert, in wie weit sie die Idee des eigentlichen Modells realisieren. „Reine" Peer-to-Peer-Architekturen gelten als „strenge Umsetzung", da dabei alle Knoten als Servents auftreten. Mit solchen Ansätzen können viele Beschränkungen von Client-Server-Architekturen aufgehoben werden, z.B. durch Verbesserung der Skalierbarkeit. Im Allgemeinen kommt es aber zu einer höheren Komplexität bei der Umsetzung bestimmter Funktionalitäten, wie z.B. der Sicherheit. Es steht nämlich dann kein zentraler Server für die Authentifizierung und Autorisation zur Verfügung, so dass sensitive Daten verteilt gehalten wer-

[43] R. Reussner u. W. Hasselbring: *Handbuch der Software-Architektur*, dpunkt.verlag 2006, Seite 445ff

den müssen. Außerdem ist die Einhaltung bestimmter nicht-funktionaler Eigenschaften schwierig. Zum Beispiel erfordert die Suche nach Ressourcen in reinen Peer-to-Peer-Architekturen ein höheres Nachrichtenaufkommen als in Client-Server-Architekturen. Dort muss evtl. nur ein einzelner Server befragt werden, während in der reinen Peer-to-Peer-Architektur meist mehrere oder sogar alle Peers befragt werden müssen. Die Performanz kann dadurch im Vergleich zu Client-Server-Systemen beeinträchtigt werden. Derartige Probleme werden in hybriden Architekturen dadurch umgangen, dass Peer-to-Peer- und Client-Server- Konzepte gemischt werden. Betrachten wir die Alternativen etwas genauer.

Reine Peer-to-Peer-Architekturen (engl. pure Peer-to-Peer architectures) setzen das Peer-to-Peer-Kommunikationsmodell streng um und verzichten dementsprechend komplett auf zentrale Server. Alle Peers treten als Servents auf, um Dienste auf viele Knoten verteilen zu können, so dass solche Architekturen als dezentral gelten. Reine Peer-to-Peer-Architekturen lassen sich danach unterscheiden, ob sie strukturierte oder unstrukturierte Netzwerktopologien benutzen. In strukturierten Peer-to-Peer-Architekturen unterliegt die Netzwerktopologie bestimmten Beschränkungen. Verschiedene Ausprägungen sind denkbar, wie z.B. baumartige Strukturen. Abb. 6.6 zeigt einen Überblick über gängige Peer-to-Peer-Topologien:

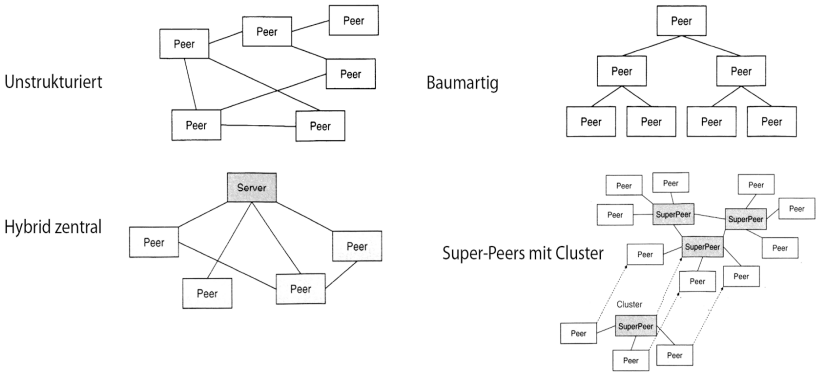


Abb. 6.6 Peer-to-Peer-Topologien

Es gibt noch eine ganze Reihe weiterer Architekturmuster, welche aufzuzählen den Rahmen dieses Buches sprengen würde. Es sei an dieser Stelle auf die weiterführende Literatur verwiesen.

Nach den Architekturmustern wollen wir uns einigen Standard-Architekturen zuwenden, die sich zur Zeit jedenfalls gewisser Popularität erfreuen.

SOA

SOA ist die Abkürzung für Service Orientierte Architektur[44]. Es handelt sich dabei um ein Architekturkonzept, welches sich im Wesentlichen aus Diensten zusammensetzt. Dabei ist die Architektur eines Softwaresystems durch eine Konfiguration von Komponenten und Konnektoren beschreibbar. Eine bestimmte Konfiguration bildet dabei eine konkrete Architektur. Werden diese Vorstellungen auf das Konzept der SOA überführt, so entsprechen die Komponenten einer (dienstorientierten) Architektur den Diensten, die Konnektoren den verschiedenen möglichen Interaktionen zwischen Diensten. Das Konzept der Service-Oriented Architecture sieht im Wesentlichen drei Komponenten vor (siehe Abbildung 6.7):

- Service Providers
- Service Registries
- Service Requestors

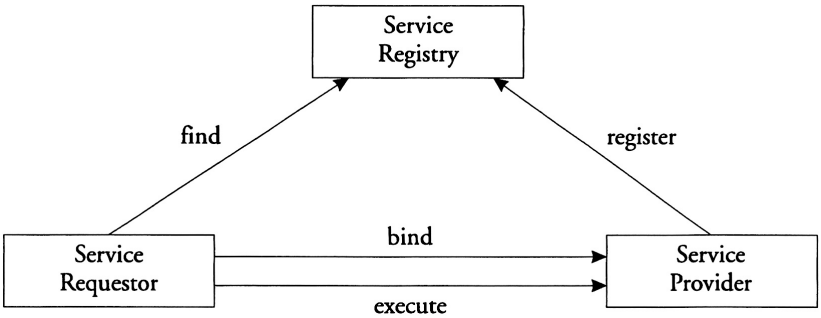


Abb. 6.7 SOA

Die im SOA-Konzept vorhandenen *Basisinteraktionen* entsprechen auf der Konzeptebene den Konnektoren einer Service-orientierten Architektur:

- Registrierung eines Dienstes (*register*, RegisterConnector)
- Suchen und Finden eines Dienstes (*find*, FindConnector)
- Herstellen der Bindung zu einem Dienst (*bind*, BindConnector)
- Stellen einer Anfrage an einen Dienst (*execute*, ExecuteConnector)

[44] Wir folgen Zöller-Greer, P.: *Software Anaylse und Design*, Verlag Composia, 2007, S. 20ff

Eine der wichtigsten Eigenschaften von SOAs ist, dass sie weitgehend transparent sind. In SOAs sind immer die Realisierungen von Diensten strikt von ihren Beschreibungen getrennt ist. Konkret bedeutet dies, dass zu jedem Dienst eine separate Schnittstelle (service interface) existiert, die den Dienst eindeutig beschreiben kann. Unter Zuhilfenahme von plattformunabhängigen Beschreibungstechniken wie der Web Service Description Language (WSDL), kann somit nicht nur der Dienst, sondern sogar die zur Realisierung verwendete Plattform abstrahiert werden. Die Realisierung der einzelnen Dienste ist somit transparent und das Gesamtsystem heterogen und interoperabel.
Es besteht beispielsweise die Möglichkeit, Dienste in Java, *CORBA* oder Microsoft .NET zu realisieren. Sofern zur Beschreibung WSDL (oder etwas Äquivalentes) verwendet wird, ist es möglich, die Dienste auf anderen Basismaschinen zu nutzen. Dies zieht sich von der Nutzung eines Java-Web Services in einer .NET-basierten Anwendung bis hin zu komplexen Einsatzszenarien im Bereich der *EAI (Enterprise Application Integration)*.
Wie wir in Abb. 6.7 sahen, besteht die Service-Oriented Architecture aus den drei Komponenten Service Provider, Service Registries und Service Requestoren. Diese Komponenten werden nachfolgend genauer beschrieben.

Service Provider
Service Provider entsprechen den Dienstleistern. Ein Dienstleister kann entweder eine Softwarekomponente oder ein Dienst-anbietendes Unternehmen sein. Dies hängt hauptsächlich von der aktuellen Sicht auf die Architektur ab. Ein Service Provider stellt damit eine Funktionalität oder eine Ressource und eine passende Beschreibung zur Verfügung. Diese Beschreibung entspricht einem Vertrag. Dieser beschreibt die Interaktion mit dem Dienst (Schnittstellenbeschreibung) und kann weitere Informationen enthalten, die nicht an die konkrete Funktionalität gebunden sind (sog. „nichtfunktionale Aspekte“).

Service Registry
Die Registry stellt eine zentrale Registrierung für die angebotenen Dienste dar, während die *Service Registry* eine dezentrale Registrierung beschreibt. Service Provider registrieren ihre Dienste bei einer Registrierung mithilfe eines Vertrags (service interface). Eine Registrierung kann dabei unterschiedlich ausgelegt sein: es gibt z.B. verzeichnisartige Registrierungen wie UDDI, welche gerade im Bereich der Web Services weit verbreitet sind (UDDI steht für Universal Description, Discovery and Integration; eine Organisation, die Webdienste anbieten möchte, kann eine UDDI –Registrierung, auch Business Registration genant, erstellen, welche aus einem XML-Dokument besteht, dessen Format in einem von UDDI definierten Schema festgelegt ist). Eine solche Registrierung ist passiv, sie kann nur Aufträge von Service Providern entgegennehmen (registrieren,

updaten, löschen eines Dienstes) oder Anfragen von potenziellen Clients (Suche nach Dienstangebot) entgegennehmen.
Aktive Registrierungen stellen demgegenüber eine Möglichkeit zur Verfügung, weitere komplexe Aufgaben wahrnehmen können. Zu nennen sind hier Konzepte, wie sie beispielsweise in ODP (Open Distributed Processing) mithilfe von Tradern beschrieben werden. Weitere Aufgaben der Registrierungen sind Kategorisierung und Katalogisierung sowie Priorisierung und Ranking.

Service Requestor/Client
Service Requestoren sind mit Clients üblicher Client/Server-Architekturen vergleichbar. Sie kennen nur Registrierungen, an die sie Suchanfragen nach bestimmten Dienstleistungen stellen können. Die Dienstvermittlung erfolgt dann auf der Basis der registrierten Verträge. Analog zu Client/Server-Architekturen ist die Diensterbringung für den Client transparent und hinter einer Schnittstelle versteckt.

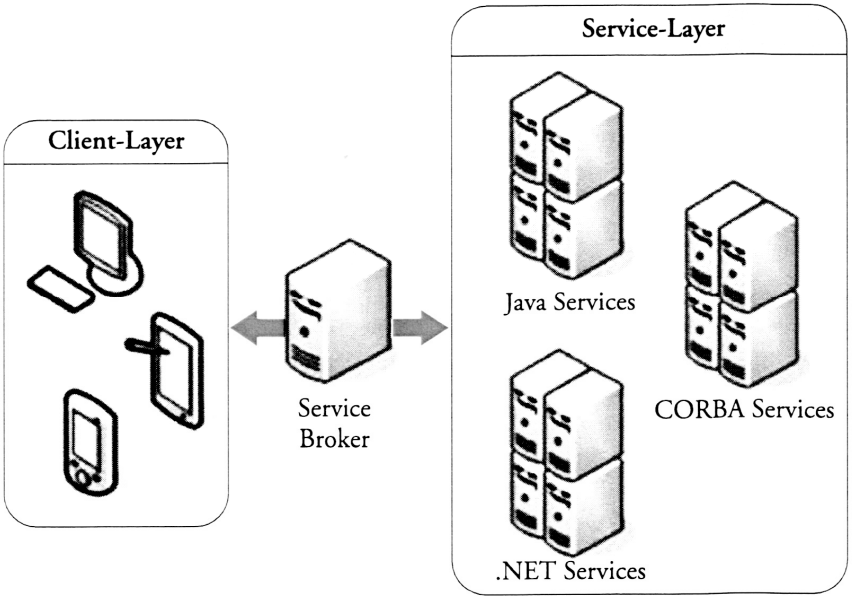


Abb. 6.8 Servicenutzung unter SOA

Die Komponenten einer SOA können nun verwendet werden, um eine dienstorientierte Architektur in drei Teilsysteme zu gruppieren:

1. Dienstnutzung (Client Layer)
2. Dienstvermittlung (Service Broker)
3. Diensterbringung (Service Layer)

Gezeigt ist hier ein heterogenes Szenario, das einige Besonderheiten der SOA verdeutlicht. Dienste können nahezu beliebig realisiert sein (rechter Teil der Abbildung). Sie müssen lediglich der Anforderung genügen, ihre Funktionalität über eine normierte Schnittstelle, z.B. eine WSDL-Schnittstelle, anzubieten.
Eine Registry (Service Broker, Mitte der Abbildung 3.2) kann administrative Aufgaben wahrnehmen (Katalogisierung, Kategorisierung) und, weitaus wichtiger, die Funktion eines Vermittlers ausüben. Der Service Broker ermöglicht sowohl Plattform-, als auch Ortstransparenz. Diese „Indirektionsstufe" ist gerade für den Dienstnutzer interessant, da sie es ihm ermöglicht, beliebige Dienste an beliebigen Orten zu nutzen. SOAs werden deshalb meistens als lose gekoppelte Architekturen betrachtet.
SOA erfährt derzeit gerade durch den vermehrten Einsatz von Internetapplikationen eine immer größere Relevanz.

TOGAF
TOGAF[45] steht für „The Open Group Architecture Forum" und hat seit 1994 eine sogenannte „Architecture Developement Method" (ADM) entwickelt, welche sich zu einem gewissen Standard vor allem im Hinblick auf die „Enterprise Architectures" entwickelt hat. Unter einer Enterprise versteht TOG ein Kollektion von Unternehmen, die gemeinsame Ziele verfolgen. Bei der Benutzung von TOGAF wird die Unternehmensarchitektur in vier Domänen modelliert: Geschäftsarchitektur, Anwendungsarchitektur, Datenarchitektur und Technologiearchitektur. Genauer:

Geschäftsarchitektur
Die Geschäftsarchitektur umfasst die Aufbauorganisation, die Strategie und die Geschäftsprozesse des Unternehmens. Die Geschäftsprozessmodellierung liefert dann die Geschäftsprozessarchitektur.

Informations- und Datenarchitektur
Hier werden die relevanten Daten nebst ihren Beziehungen, welche für die Geschäftsprozesse benötigt werden, identifiziert und beschrieben. Das Ganze soll dabei stabil, vollständig, konsistent und für alle Beteiligten verständlich sein. Dabei repräsentiert die Informationsarchitektur –wie der Name schon sagt- Informationen, Informationsgruppen und deren Informationsbedürfnisse. Der

[45] http://www.opengroup.org/togaf/

Begriff Informationsgruppe fasst dabei die verschiedenen Rollen zusammen, die denselben Informationsbedarf haben.

Anwendungsarchitektur
Hier werden Anwendungen verwaltet, welche für die Ausführung von Geschäftsprozessen nötig sind. Außer der Bestandsführung aller Anwendungen werden auch die Beziehungen und Schnittstellen zwischen den Anwendungen betrachtet. Kategorisiert werden die Anwendungen gemäß ihrer fachlichen Funktionalität sowie der davon verarbeiteten Informationen. Während die Kategorien selbst relativ stabil sind, können die konkreten Anwendungen, die innerhalb der Kategorien zum Einsatz kommen, beliebig ersetzt werden.

Technologiearchitektur
Die Technologiearchitektur beschreibt den Aufbau sowie den Betrieb der IT-Infrastruktur. Damit kann festgelegt werden, auf welcher Grundlage Anwendungen beschafft, integriert und betrieben werden können.

ADM wird im Prinzip zyklisch durchlaufen. Durch die zyklische Ausführung dieses Prozesses wird die Architektur auch fortgeschrieben.
Der Prozess selbst besteht aus acht Phasen:

- In der Phase A (Architekturvision) werden die Ziele und die Beteiligten für die Aktualisierung der Unternehmensarchitektur festgelegt.
- In den Phasen B-D werden für die Geschäfts-, Anwendungs-, Informations-/Daten- und die Technologiearchitektur der Ist- und der Soll-Zustand beschrieben. Die entscheidenden Unterschiede werden herausgearbeitet.
- In Phase E werden die Vorhaben festgelegt, welche die Transformation des IST-Zustands zum Soll-Zustand durchführen.
- Phase F beschreibt die übergreifende Zusammenarbeit der einzelnen Vorhaben
- Phase G überwacht Phase F.
- In Phase H werden die Anforderungen sowie die externe Einflüsse gesammelt; sie dienen dann als Grundlage für den nächsten Durchlauf der ADM.

Abb. 6.9 zeigt einen Überblick über die zyklische ADM-Struktur:

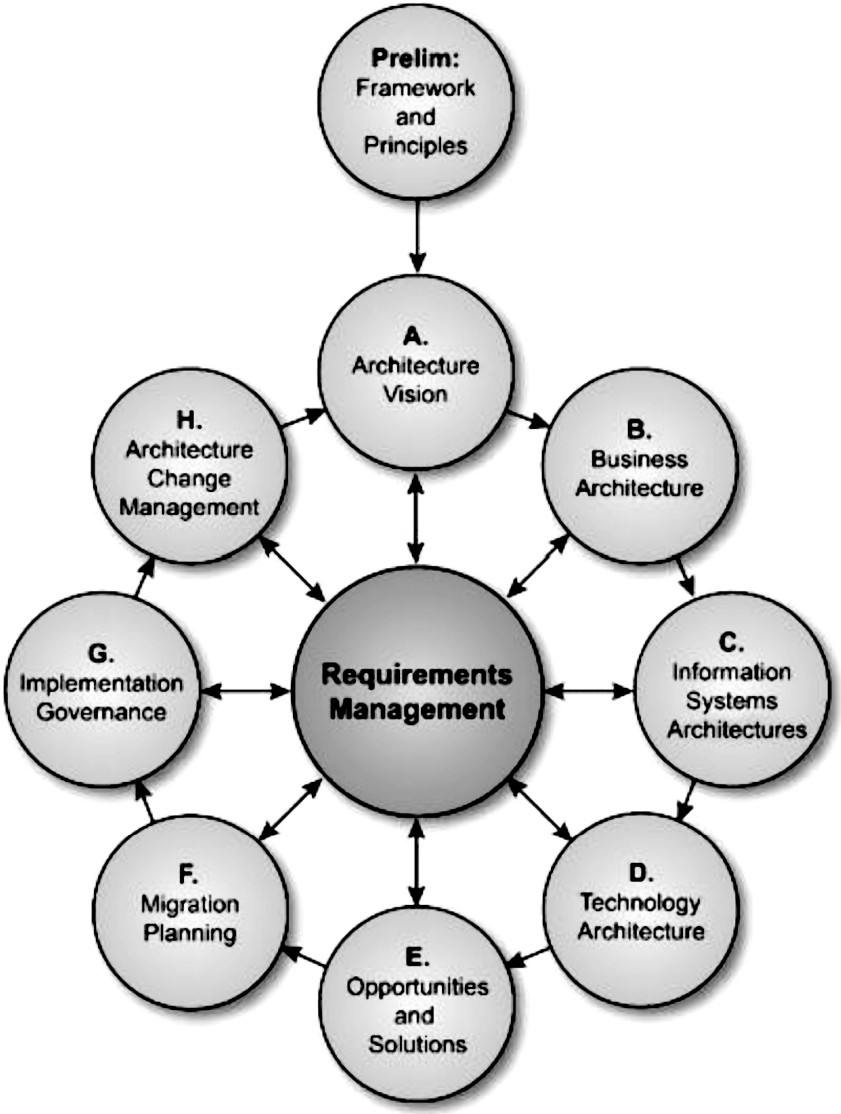


Abb. 6.9 ADM-Zyklen[46]

[46] Quelle: ibid.

MDA

MDA steht für „Modell Driven Architecture“ und wurde von der OMG entwickelt. Allerdings handelt es sich hier gar nicht um eine Software-Architektur im Sinne unserer Definition, sondern allenfalls um eine Anwendung davon. Daher wird an dieser Stelle nur relativ kurz darauf eingegangen, um wenigstens die wichtigsten damit verbundenen Begriffe zu kennen[47].

MDA ist eine Strategie zur modellgetriebenen und generativen (d.h. automatisch generierten) Soft- und Hardwareentwicklung. Durch automatische Generierung von Quellcode aus den Modellen wird der Automatisierungsgrad der Entwicklung erhöht (und somit Fehlerquellen minimiert). MDA etabliert sich zunehmend als ein Standard, der Software schneller, effizienter, kostengünstiger und qualitativ hochwertiger zu erstellen vermag.

Der Kern der MDA besteht in der Trennung von fachlichen und technischen Teilen eines Softwaresystems in Form von plattformunabhängigen Modellen (PIMs) und plattformspezifischen Modellen (PSMs), die beide unabhängig voneinander wieder verwendet werden können.

In den PIMs wird das fachliche Wissen (Fachlogik) des Softwaresystems/Anwendung, wie z. B. Geschäftsprozesse oder Fachverfahren, technologieunabhängig erfasst und modelliert. Im Gegensatz dazu wird in den plattformspezifischen Modellen die Implementierungstechnologie definiert, d. h. die technischen Aspekte bezogen auf eine konkrete Plattform. Hierbei kommen häufig Frameworks zum Einsatz. Die explizite Trennung der Fachlogik von der Implementierungstechnologie erlaubt eine leichtere Wiederverwendung der Fachlogik, z. B. im Rahmen einer Migration auf eine neue Plattform.

Ziel der MDA ist unter anderem, die Lücke zwischen Modellen (bzw. Modellierern) und Quelltext (bzw. Programmierern) zu schließen. Im Allgemeinen bewirkt ja jede Änderung im Entwicklungsprozess auch Änderungen auf den einzelnen Abstraktionsebenen. Grundsätzlich gilt die Faustregel, dass je höher der Abstraktionsgrad, desto invarianter ist er gegenüber Änderungen tieferliegender Ebenen. Zudem sollen sich die abstrakteren Modelle (PIM) durch Mapping (zumindest teilweise) automatisch in weniger abstrakte Modelle (PSM) überführen lassen. Dies scheint mit MDA in scheinbare Nähe gerückt. Eine Standardisierung in diesem Bereich verspricht stabilere Softwaren und kürzere, besser kalkulierbare Entwicklungszeiten.

Konzepte und Basistechnologien

Die OMG veröffentlicht in ihrem MDA-Guide[48] die grundlegenden Konzepte und Technologien. Wir folgenden im Wesentlichen diesen Ausführungen.

Es gibt 4 wichtige Konzepte in MDA:

[47] Ausführlicher siehe z.B. Zöller-Greer, P.: *Software Analyse und Design*, Verlag Composia, 2007

[48] siehe http://www.omg.org/docs/omg/03-06-01.pdf

Computation Independent Model (CIM)
Das CIM repräsentiert die Geschäfts- oder Domänensicht des zu entwickelnden Softwaresystems und ist der Modelltyp mit der höchsten Abstraktionsebene.

Platform Independent Model (PIM)
Mit Hilfe des PIMs wird die Struktur und das Verhalten des zu entwickelnden Softwaresystems spezifiziert. Dies erfolgt unabhängig von den technischen Aspekten (Software- und Hardwareplattformen) der Implementierung.

Platform Specific Model (PSM)
Das PSM erweitert das PIM um die konkreten technischen Details, die für die Implementierung des Softwaresystems in der jeweiligen Zielplattform notwendig sind.

Modell-Mapping
Mit Hilfe so genannter Mappings (Abbildungen) wird das PIM gegenüber einer konkreten Zielplattform basierend auf den Plattformprofilen in ein entsprechendes PSM übersetzt.

MDA unterstützt folgende Basistechnologien:

UML
Die Unified Modeling Language (UML, vgl. Kapitel 5) ist integraler Bestandteil der MDA-Konzepte. In der aktuellen Version UML 2.0 wird durch die Erweiterung hinsichtlich Metamodellierung und Profilmechanismus der MDA-Ansatz unterstützt. Die UML dient der (graphischen) Beschreibung von Modellen, die je nach Modellart z. B. die Struktur oder das Verhalten festlegen. Hierzu werden (bzw. wurden bereits) diverse Erweiterungen in Form von Plattformprofilen (wie z.B. UML/Realtime für Echtzeitanwendungen) konzipiert, um der UML immer mehr Plattformen zu erschließen.

UML-Profil
UML-Profile sind der Standardmechanismus zur Erweiterung des Sprachumfangs der UML, um sie an spezifische Einsatzbedingungen (z. B. fachliche oder technische Domänen) anzupassen.

MOF
Die Meta Object Facility (MOF) ist ein Standard der OMG zur Definition von Metamodell-Sprachen. Die MOF ist auf der Meta-Metamodellebene angesiedelt

und definiert den Aufbau der Metamodellebene auf der z. B. UML und CWM liegen.

XMI
eXtensible Markup Language Metadata Interchange (XMI) ist ein XML-basiertes Austauschformat für UML Modelle, was die Interoperabilität von Modellen zwischen verschiedenen Werkzeugen unterschiedlicher Hersteller ermöglicht. So lassen sich mittels der MDA die Modelle zwischen verschiedenen Werkzeugen und Werkzeugkategorien unterschiedlicher Hersteller austauschen.

CWM
Das Common Warehouse Metamodel (CWM) ist ein von der OMG entwickeltes und standardisiertes Referenzmodell für den formalen und herstellerunabhängigen Zugriff und Austausch von Metadaten in der Domäne Data Warehousing.

Die OMG legt in ihrem grundlegenden MDA-Guide viel Wert auf das Mapping von PIM nach PSM. Dies ist in der Tat auch der wichtigste Aspekt der ganzen Angelegenheit: Hat man nämlich z.B. ein UML-Klassendiagramm (innerhalb des PIM), so ist das Problem gerade, daraus „automatisch“ eine lauffähige Software-Anwendung zu erzeugen (welche alle Angaben des konkreten PSM benötigt).

Es muss also möglich sein, dass für ein vorhandenes PIM als „Input“ ein konkretes PSM als „Output“ geliefert werden kann, wobei natürlich die plattfomkonkreten Parameter (Betriebssystem, Hardware etc.) ebenfalls mitgegeben werden müssen. Dies soll nachfolgend näher spezifiziert werden.

Mapping PIM → PSM

Grundsätzlich unterscheidet die OMG verschiedene Formen des Mappings. So gibt es das „Type Mapping“, welches ein Mapping zwischen Metamodellen (wie MOFs) ermöglicht, und dem „Instance Mapping“, wo konkrete Elemente eines PIM in konkrete Elemente eines PSM umgesetzt werden. Über sogenannte „Marks“ (Markierungen) werden diese Elemente ausgewählt. Marks können allgemein beschrieben und wiederverwendet werden. Beispielsweise kann die Mark „Entity“ ein PSM-Element sein, was allgemein Klassen aus dem PIM zugeordnet wird. Die Marks werden also dazu benutzt, geeignete Information für das Mapping zur Verfügung zu stellen, wie nachfolgende Abbildung zeigt.

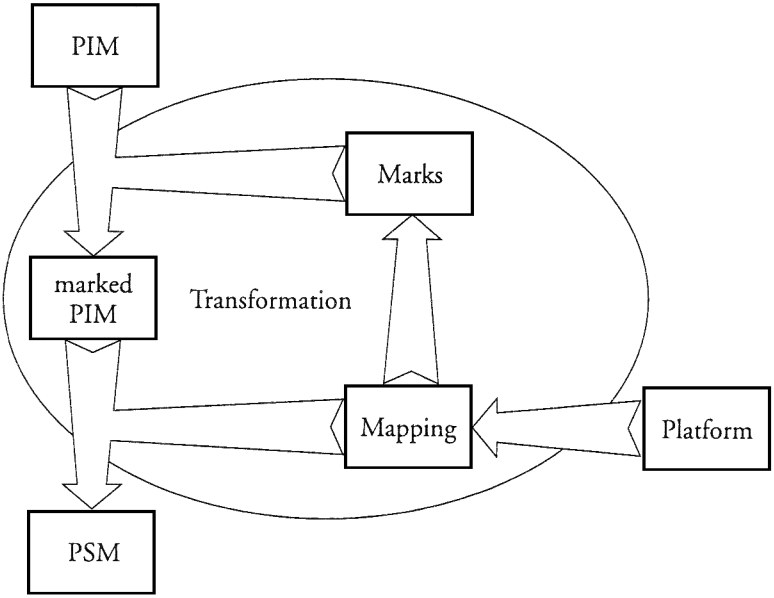


Abb. 6.10 Mapping mit Marks

Wie man in Abb. 6.10 erkennt, fließen also die Marks in die Mapping-Beschreibung ein, sodass das PIM zu einem „Marked PIM“ wird. Zusammen mit der Plattform-Information kann daraus das PSM erzeugt werden (und daraus schließlich der Code für die Anwendung). Nun sind Transformationsprozesse nicht nur auf diese Ebene beschränkt. Man kann weiter abstrahieren und eine „Meta-Ebene“ einführen, wo die jeweiligen Entsprechungen zu finden sind.

RM-ODP

RM-ODP steht für Reference Model of Open Distributed Processing. Es handelt sich dabei um eine ISO-Norm, die ein Metamodell zur Beschreibung von Informationssystemen zum Gegenstand hat. RM-ODP legt damit die Konzepte fest, die von ODP Specification Languages beschrieben werden. Der Architektur-Standard wurde erstmals von der ISO in ISO/IEC 10746-3 | ITU-T Rec. X.903 beschrieben und enthält die Spezifikation der notwendigen Eigenschaften, welche einen Prozess als “offen” standardisieren. Dabei handelt es sich um Einschränkungen, welche erfüllt sein müssen, um „ODP-konform“ zu sein. Die Semantiken dafür werden in ISO/IEC 10746-4 | ITU-T Rec. X.904 festgelegt.
Es gibt fünf generische Sichten auf das System (vgl. Abb. 6.11):

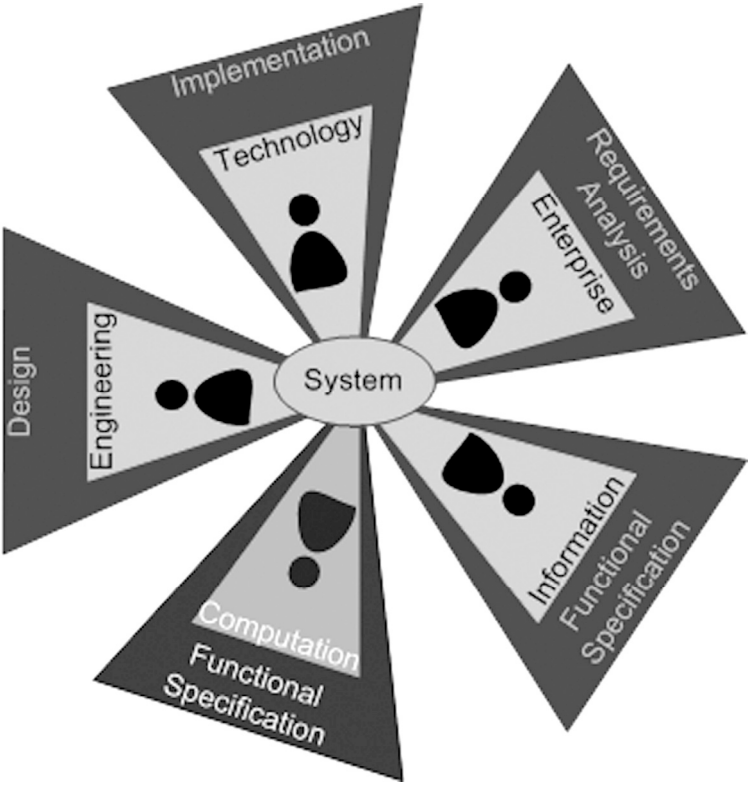


Abb. 6.11 RM-ODP-Sichten

Enterprise Viewpoint: definiert den Anwendungsbereich sowie das Ziel bzw. die Menge der auszuführenden Regeln des ODP-Systems.

Informational Viewpoint: spezifiziert die Semantik der Informationen und Informationsverarbeitung.

Computational Viewpoint: zerlegt die Funktionalität eines ODP-Systems in interagierende Komponenten und verfeinert so die Spezifikation.

Engineering Viewpoint: spezifiziert eine Infrastruktur, die die Verteilung unterstützt.

Technology Viewpoint: definiert die Auswahl von Technologien für ein ODP-System und beschreibt die Implementation und Informationen zum Testen des Systems.

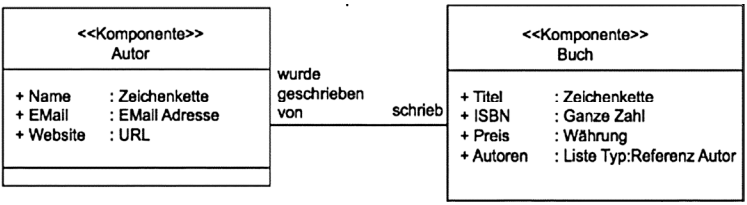
Dieses Modell wird überwiegend im Verwaltungsbereich, z.B. für eGovernment, eingesetzt. Außerdem dient es für Festlegungen z.B. im Bereich von Datenbeschreibungen wie XML und XSD, Middleware-Technologien wie J2EE und .NET, Sicherheitsstandards wie ISIS-MTT und Austauschformate für diverse Dateitypen wie Texte, Bilder, Audio- und Videodateien.

Übungsaufgaben:

1. Es gibt ein Architekturmuster mit Namen „Broker“. Es beinhaltet sechs Komponenten: Server, clients, brokers, bridges,client-side proxies und server-side proxies. Server stellen über Schnittstellen ihre Dienste über die flexible IDL (Interface Definition Language) oder einen Binärstandard zur Verfügung. Entweder sie enthalten spezielle Funktionalitäten für eine Anwendung, oder aber allgemein gehaltene Dienste für mehrere Anwendungsgebiete. Clients sind Anwendungen, die Dienste von mindestens einem Server aufrufen. Nachdem die Operation ausgeführt wurde, erhalten sie das Ergebnis über den Broker. Server können ebenfalls als Clients agieren und heben so eine feste Rollenzuweisung zugunsten von dynamischen Strukturen auf. Da die Clients nicht den Standort des angesprochenen Servers kennen müssen, ergibt sich der Vorteil, Komponenten zur Laufzeit auszutauschen oder neue Dienste hinzuzufügen. Die „Vermittler“ bilden die Schnittstelle zwischen Client und Server. Falls der richtige Server allerdings nicht von dem Broker direkt angesteuert werden kann, muss er mit weiteren Brokern kommunizieren, bei denen der gesuchte Server angemeldet ist. Dies geschieht über so genannte „bridges“. Zwischen Server/Client und Broker besteht bei der indirekten Kommunikation eine Verbindung über die Vermittlungskomponente proxy. Dies dient zum Beispiel der Internalisierung von Daten (Client) bzw. dem Aufruf der relevanten Dienste (serverseitig) und berücksichtigt so jeweils die implementierungsspezifischen Details von Client und Server. Zunächst fordert die Client Anwendung den Dienst eines lokalen Servers an. Die Nachricht wird vom proxy an den Broker weitergeleitet, welcher in seinem Verzeichnis nach dem richtigen Server sucht, um die Dienstanforderung dann entsprechend weiterzuleiten. Vom server-side proxy wird der geforderte Dienst dann gestartet und das Ergebnis über den gleichen Weg wieder zurück bis zur Client Anwendung geleitet.

Stellen Sie den hier geschilderten Sachverhalt betreffs des Architekturmusters „Broker“ mit Hilfe eines Diagramms zur Unterbringung in der Laufzeitsicht dar!

2. Im Rahmen von MDA hatten wird die Transformationen vom PIM nach PSM besprochen. Es sei auf PIM-Ebene folgendes gegeben:



Diese Abbildung stelle ein (gekürztes) Autor/Buch-Modell dar, wobei gegenüber den Plattformen der eingesetzten Programmiersprache und Middleware abstrahiert wurde. Erstellen Sie ein entsprechendes UML-Diagramm auf PSM-Ebene für eine Sprache Ihrer Wahl (z.B. Java Beans o.ä.).

7. Dokumentation von Architekturen

Wenn man eine Software-Architektur ermittelt, dann muss dies natürlich geeignet dokumentiert werden. Sinnvollerweise geschieht dies während der gesamten Entwurfsphase, doch wenn die Architektur dann endlich „steht", sollte man sie in einer Dokumentation festschreiben, die auch Jahre nach deren Erstellung noch verstanden werden kann. Er reicht also nicht, einfach die während der Erstellungsphase gesammelten Werke in einem oder mehreren Aktenordnern abzuheften. Es muss am Ende eine Dokumentation erstellt werden, die in einer möglichst standardisierten Form dem Leser zugänglich gemacht werden kann.
Die Dokumentation, die während des Entwicklungsprozesses getätigt wird (manchmal auch SWAD=Software Architektur Dokumentation genannt) ist, wie erwähnt, zunächst nicht statisch, sondern einer gewissen Evolution unterlegen. Wir haben zwar die „Einzelteile" einer Architekturbeschreibung kennen gelernt (wie Sichtenmodelle, Spezifikationssprachen etc.), doch dies alles muss ja geeignet zusammen gebracht werden, sodass daraus eine sinnvolle Architektur-Dokumentation entsteht. Zudem sind Aspekte wie Sicherheit, Persistenz, Ergonomie etc. zu berücksichtigen. Auch muss man sich Gedanken machen zu Tests, Evaluierungen (wie Prototyping, Simulation, Verifikation) und so weiter. Dazu wollen wir uns einige Gedanken machen und so den Inhalt einer grundständigen Architektur-Dokumentation entwickeln. Der Erfinder des 4-Sichten-Modells nach Starke hat dazu sehr gute Vorschläge gemacht, denen wir hier folgen[49].

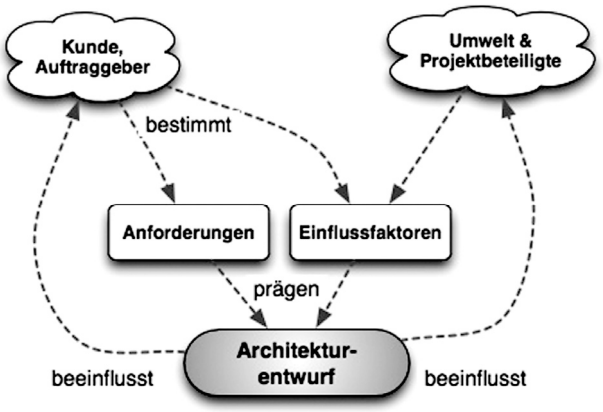


Abb. 7.1 Iterative Entwicklungsschritte[50]

[49] Starke, G.: *Effektive Software-Architekturen*, Hanser, 2008, Seite 106ff
[50] Quelle: http://www.arc42.de/prozess/process.html

Grundsätzlich sollte der gesamte Prozess einer Architekturentwicklung den gleichen Methoden und Ansätzen genügen, die Ihnen schon durch das allgemeine Projektmanagement (speziell im Software-Engineering) bekannt sind. Dies lässt sich auf Architekturentwicklungen teilweise übertragen (Abb. 7.1).

Eine Architektur lässt sich nicht in „einem Schritt“ entwickeln, sondern entsteht vielmehr „iterativ“. Ähnlich wie im Spiralmodell bei der Softwareentwicklung sind also die jeweiligen Schritte unter Umständen mehrmals zu durchlaufen.

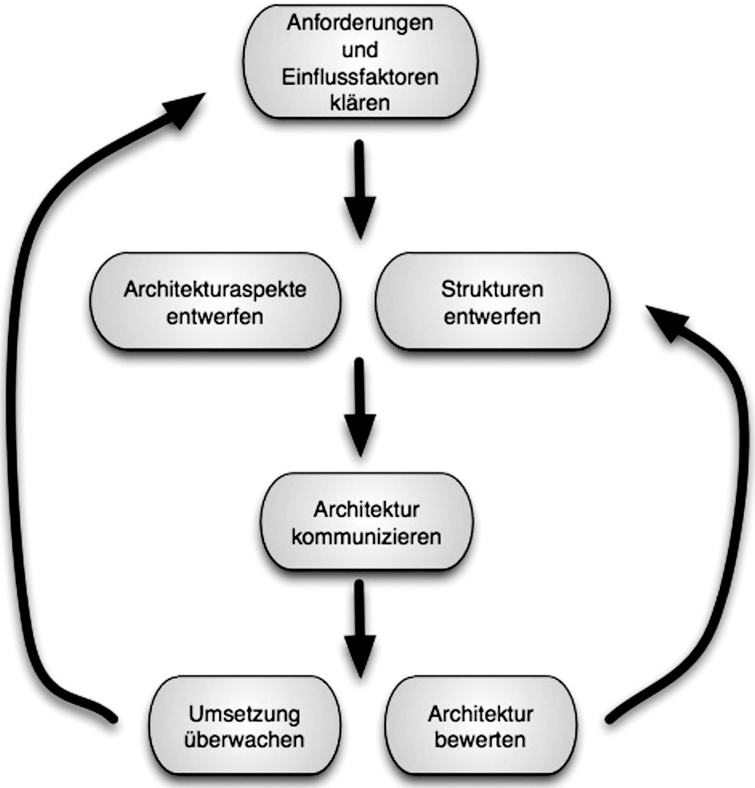


Abb. 7.2 Aktivitäten bei der Architekturentwicklung[51]

Abb. 7.2 zeigt die jeweiligen Aktivitäten, welche bei einer Architekturentwicklung zu leisten sind. Starke beschriebt diese Aktivitäten so[52]:

[51] Quelle: ibid.
[52] Quelle: ibid.

1. Anforderungen und Einflussfaktoren klären

1.1 Kontext abgrenzen
Das System in Zusammenhang seiner Nachbarn beschreiben:

- logischen Kontext ermitteln (bzw. aus Requirements- Spezifikation übernehmen)
- physischen Kontext ermitteln (physische Formate von Schnittstellen und Übertragungsmedien)

1.2 Requirements verstehen bzw. vervollständigen
Diese Tätigkeit kann entfallen, wenn gut geschriebene und qualitätsgesicherte Anforderungen vorliegen. Ist dies nicht der Fall, so sollten Software-Architekten insbesondere die nicht-funktionalen Anforderungen nochmals genau hinterfragen, denn sie bilden wesentliche Einflussfaktoren für die Architektur.

1.3 Architekturtreiber und Randbedingungen ermitteln
Insbesondere sollten Sie:

- Ziele für die Architektur festlegen
- vorhandene Stakeholderlisten um architekturrelevante Stakeholder ergänzen, bzw. Stakeholder ermitteln
- Fachliche Aufgabenstellung (aus der Requirements- Spezifikation) abstrahieren oder neu erstellen

2. Strukturen entwerfen
Trennen Sie besonders die fachliche Architektur von technischen Aspekten. Verwenden Sie gezielt Szenarien, um beispielhaft ausgesuchte Qualitätseigenschaften kritisch zu überprüfen.

3. Aspekte konzipieren

4. Architektur kommunizieren
Mit Sichten und Aspekten versuchen Sie, eine möglichst vollständige Dokumentation der Software- Architekt zu erstellen. Nicht jeder im Projekt will und muss das alles lesen, verstehen und kommentieren. Deshalb gehören zu den Detailaufgaben dieser Aktivität folgende, immer wieder durchzuführende Schritte:

4.1 Wünsche und Bedürfnisse der Stakeholder identifizieren.
Prüfen Sie regelmäßig die Stakeholderliste, ob sie vollständig ist, wer davon über welche Teile der Architektur informiert werder muss und in welcher Form dies am besten geschieht.

4.2 Architektur kommunizieren

Zu jedem beliebigen Zeitpunkt sollten Sie die vorliegenden Erkenntnisse und Entscheidungen stakeholdergerecht zusammenfassen und präsentieren.
Extrahieren Sie aus der Gesamtheit der Architekturinformationen die Teile in der Form, wie sie für die jeweiligen Stakeholder wichtig sind in der geeigneten Form (Präsentationen, Dokumente, Übersichtslisten, ...). Achten Sie jedoch darauf, dass diese Extrakte kein Eigenleben entwickeln und aufwendig gepflegt und versioniert werden müssen. Holen Sie das Feedback der Stakeholder ein, um es entweder in die zentralen Sichten und Aspekte einbringen zu können oder die Ziele, die Randbedinungen oder den Kontext anzupassen.

5. Umsetzung der Architektur überwachen
Wenn eine Heerschaar von Designern und Programmierern die Architektur mit detailliertem Leben füllt, Source Code schreibt oder generiert, ist die Arbeit des Architekten noch lange nicht erledigt. Die Rolle des Architekten erfordert die ständige Überwachung der vorgegebenen Strukturen, sowie die zeitgerechte Einarbeitung berechtigter Änderungswünsche - d.h. die Anpassung der Architektur an neue Randbedingungen oder Erkenntnisse im Projekt. Zu diesen Aufgaben gehören regelmäßige Gespräche mit allen Personen im Projekt, die auf der Architektur aufsetzen, also hauptsächlich mit Designern und Programmierern, aber auch mit Qualitätsmanagement, insbesondere den Testern, und vielen anderen. Von diesen Gruppen erhält der Architekt Feedback bezüglich der "Tragfähigkeit" und "Brauchbarkeit" der Architektur. Dieses Wissen fließt in die ständige Weitergestaltung der Architektur ein, d.h. in Änderungen von Strukturen oder Aspekten (siehe oben). Ebenso gehört die frühzeitige Abstimmung mit Migrations- und Inbetriebnahmeteams sowie Produktion und Betrieb zu den Überwachungsaufgaben.

6. Architektur bewerten
Nutzen Sie gezielt ausgewählte Szenarien und Prototypen für die Architekturbewertung.
Die Ergebnisse dieser Tätigkeit (offene Punkte, identifizierte Risiken, ...) sagen Ihnen, was sie weiter vorgehen sollen. Oftmals sind nur lokale Änderungen und Anpassungen an den Sichten und Aspekten vorzunehmen, manchmal müssen Sie den Regelkreis jedoch größer gestalten und aufgrund der Bewertungsergebnisse auch Ihre Ausgangssituation, die Ziele, die Randbedingungen, ... korrigieren.

Was die Dokumentation dieser Aktivitäten bzw. deren Ergebnisse betrifft, so kann folgender Gliederungsvorschlag evtl. eine Hilfestellung leisten[53]:

[53] Quelle: ibid.

Überschrift	Inhalt
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 vom System bearbeiteten Geschäftsprozesse als Use-Cases (Text und Diagramm) dar.
1.2 Architekturziele	Beschreiben Sie weitere wesentliche (d.h. für die Architektur relevante) funktionale und nichtfunktionale Anforderungen, deren Erfüllung oder Einhaltung den maßgeblichen Stakeholdern besonders wichtig ist. Hier sollten insbesondere architekturrelevante Themen wie Performance, Sicherheit, Wartbarkeit und Ähnliches angesprochen werden. 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 (5–10) Ziele
1.3 Stakeholder	Eine Liste der wichtigsten Personen oder Organisationen im Umfeld des Systems.
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. Kontextsicht	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.
4. 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.
4.1, 4.2, 4.3,…	Verfeinerungsebenen 1, 2, 3,… der Bausteinsicht.
4.x Muster und/oder spezielle Abhängigkeiten	Manchmal reicht die hierarchische Zerlegung der Bausteine nicht aus, um den Überblick über die Strukturen des Systems zu behalten. Wenn beispielsweise bestimmte Strukturmuster mehrfach auftreten, können Sie diese hier dokumentieren. Genau wie bei den übrigen Bausteinsichten verwenden Sie die statischen UML-Diagramme plus die zugehörigen Erläuterungen (analog der Black- und WhiteboxDokumentation).
5. Laufzeitsichten	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.
6. Verteilungs- oder Infrastruktursicht	Diese Sichten zeigen, in welcher Umgebung das System abläuft, sowohl Hardware (Rechner, Netze etc.) als auch Software (Betriebssysteme, Datenbanken, Middleware etc.).
7. Entwurfsentscheidungen	Diese Sichten zeigen, in welcher Umgebung das System abläuft, sowohl Hardware (Rechner, Netze etc.) als auch Software (Betriebssysteme, Datenbanken, Middleware etc.).
8. Szenarien zur Architekturbewertung	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 operationalisieren Qualitätsmerkmale und bilden eine wichtige Grundlage für Architekturbewertung. Sie besitzen langfristigen Wert und bleiben meistens über längere Zeit stabil. Daher lohnt es sich, sie in die Architekturdokumentation aufzunehmen.

Überschrift	Inhalt
9. Querschnittliche Architekturaspekte	Zu übergreifenden Aspekten 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. „Füllen“ Sie diese Abschnitte, wenn es keine einzelnen Bausteine gibt, die diese Aspekte abdecken.
10. Projektaspekte	Unter dieser Überschrift sammeln Sie organisatorische und projektrelevante Faktoren, die Einfluss auf die Architektur besitzen. Es scheint ungewöhnlich, in der Architekturdokumentation auch Informationen aus dem Projektmanagement abzulegen – häufig gibt es dafür aber keinen anderen Ort, an dem Architekten diese wiederfinden würden.
10.1 Change Requests	Die Architektur größerer Systeme durchläuft während ihres „Lebens“ vielfältige Änderungen. Die wichtigen ChangeRequests sollten Sie an dieser Stelle festhalten, damit strukturelle oder sonstige architektonische Änderungen nachvollziehbar werden. Am einfachsten ist es, wenn Sie hier Querverweise zur entsprechenden Anforderungsdokumentation pflegen. Manchmal können Sie diesen Teil der Dokumentation auch durch Release Notes abdecken.
10.2 Technische Risiken	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.
10.3 Offene Punkte	Offene Punkte, bekannte Schwachstellen, ungeklärte Sachverhalte oder aufgeschobene Entscheidungen.
10.4 Erwartete Änderungen	Falls Änderungen an organisatorischen Randbedingungen oder Einflussfaktoren maßgeblichen Einfluss auf die Architektur haben, sollten Sie diese hier beschreiben (oder auch die dann notwendigen Maßnahmen).
11. Glossar	Nur wenn nötig; ansonsten Verweis auf das Projektglossar.

Übungsaufgaben:

1. Entwickeln Sie eine beispielhafte Architekturdokumentation für einen fiktiven oder realen Fall Ihrer Wahl!

8. Software Factories

Die „großen“ Softwarehäuser wie Microsoft oder IBM und SAP stehen vor dem Problem, dass sie das Produkt „Software“ im großen Stil oft in verschiedenen Variationen herstellen. Betrachten wir z.B. die Office-Produktlinie von Microsoft. Da gibt es mehrere Produkte wie Excel, Word etc.; und diese werden ständig gepatched und weiterentwickelt. Solche Software-Fabriken haben natürlich organisatorisch vergleichbares zu leisten wie eine Fabrik, die z.B. Automobile herstellt. Aus diesem Grund hat sich der Begriff Software Factories etabliert.
Das ganze A und O einer Software-Factory liegt also darin, eine funktionsfähige „Produktlinie“ im Sinne der organisierten und logistisch ausgereiften Herstellung des Produkts zu schaffen.

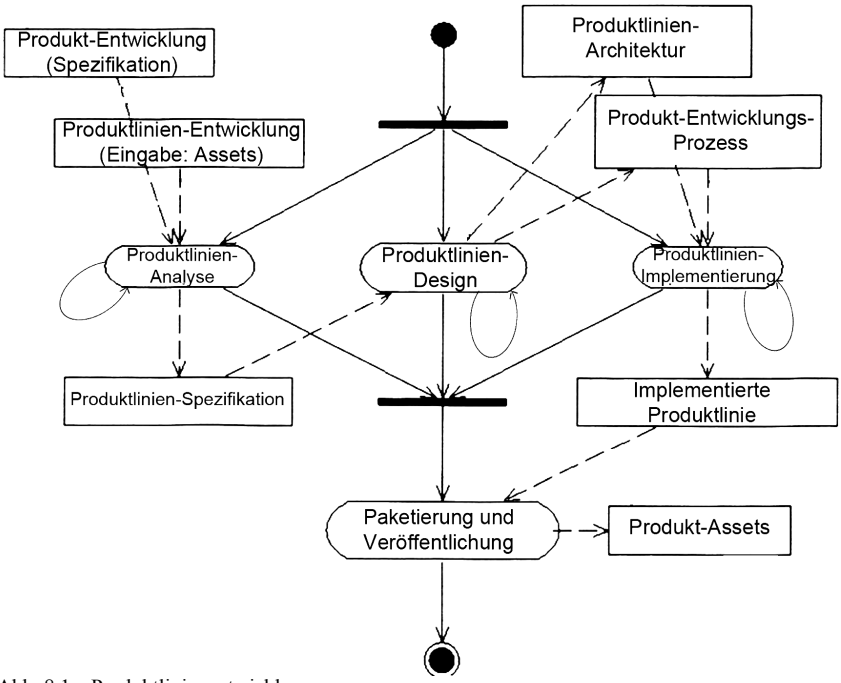


Abb. 8.1 Produktlinienentwicklung

Man sieht im Diagramm von Abb. 8.1, dass die drei Punkte „Produktlinien-Analyse“, „Produktlinien-Entwurf“ sowie „Produktlinien-Implementierung“ eine zentrale Rolle spielen. Diese Aktivitäten werden zyklisch durchlaufen. Input der Produktlinien-Analyse sind die Spezifikation des Produkts und die

„Produktlinie-Assets“, also die Gegebenheiten der Produktlinie. Als Ergebnis der Produktlinienanalyse erhält man eine Produktlinien-Spezifikation, welche in die Aktivität "Produktlinien-Design" Eingang findet. Auch diese Aktivität wird ggf. zyklisch durchlaufen. Heraus kommt eine Produktlinien-Architektur, welche ihrerseits den Produkt-Entwicklungsprozess beschreibt. Diese beiden Spezifikationen sind der Input der eigentlichen Produktlinien-Implementierung, welche auch zyklisch durchlaufen werden kann. Ergebnis dieser Implementierung ist dann die eigentliche Produktlinie, welche schließlich paketiert und veröffentlicht werden kann. Daraus können weitere Produkt-Assets entstehen.
Bei der Gelegenheit kann man auch feststellen, dass die für die Produktproduktion zuständige Architektur einem zyklischen Verfeinerungsprozess unterliegt (Abb. 8.2):

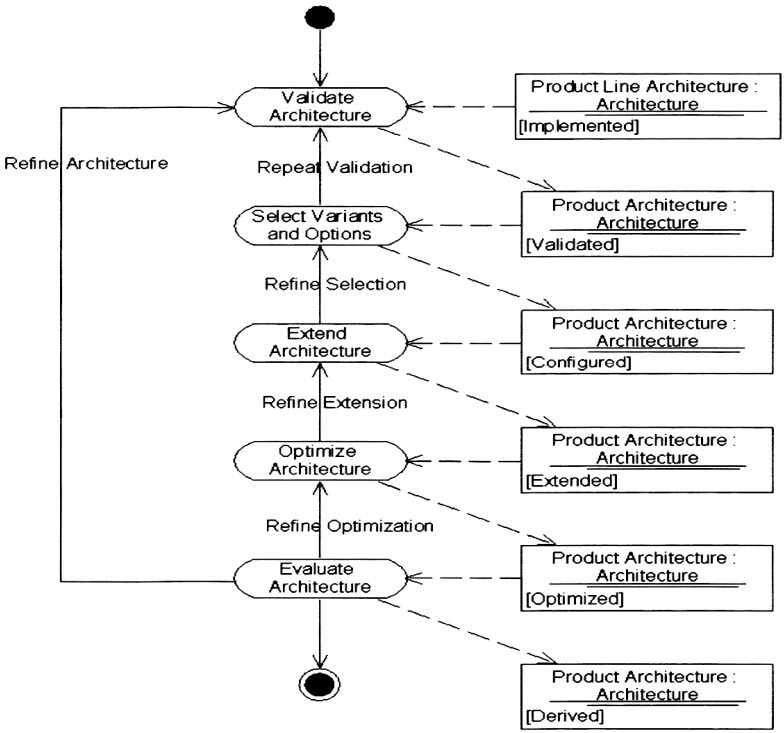


Abb. 8.2 Produktproduktions-Architektur[54]

[54] nach J. Greenfield et. all, *Software-Factories*, Wiley Publishing 2004,S. 274ff

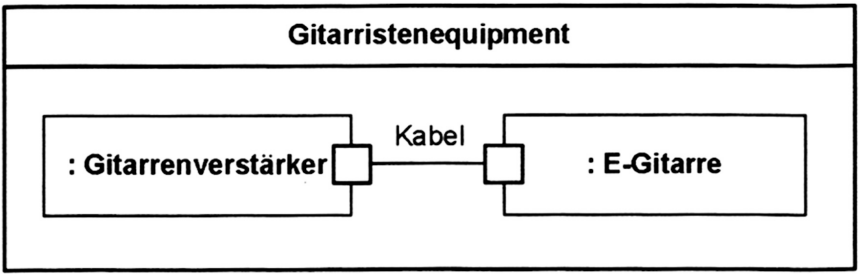
Übungsaufgaben:

1. In Abb. 8.1 wurde u.a. eine Produktlinienimplementierung dargestellt. Erstellen Sie ein sinnvolles UML-Diagramm zu den sog. „Implementierungs-Assets“: Hier sollen Komponenten zur Verfügung gestellt werden, die zur Produktion der Familienmitglieder benötigt werden. Die Komponenten werden von Lieferanten gekauft und/oder innerbetrieblich entwickelt. Sie können aus vorhandenen Produkten abgeleitet oder extern entwickelt werden. Zu jeder Komponente kann es Kontrakte, Design- und Benutzerdokumentationen, Testartefakte, Werkzeuge und Beschreibungen der Mikroprozesse geben, um sie während der Produktentwicklung anpassen und anwenden zu können. Diese Produktions-Assets können von dieser Aktivität auch paketiert, gespeichert, versioniert, gewartet und verbessert werden. Variabilitätspunkte der Produktlinienarchitektur werden durch Assemblierung und Konfiguration von Komponenten implementiert. Feinkörnige Variabilitätspunkte können durch Assoziation, Aggregation, Aufruf, Vererbung, Delegation, Parametrisierung und Einschließung implementiert werden. Mittelkörnige Variabilitätspunkte können durch Adaption, Dekoration, Mediation, Interzeption, Visitation und Observation implementiert werden. Grobkörnige Variabilitätspunkte können durch Assembly implementiert werden. Viele dieser Mechanismen werden durch das Design oder architektonische Patterns unterstützt.

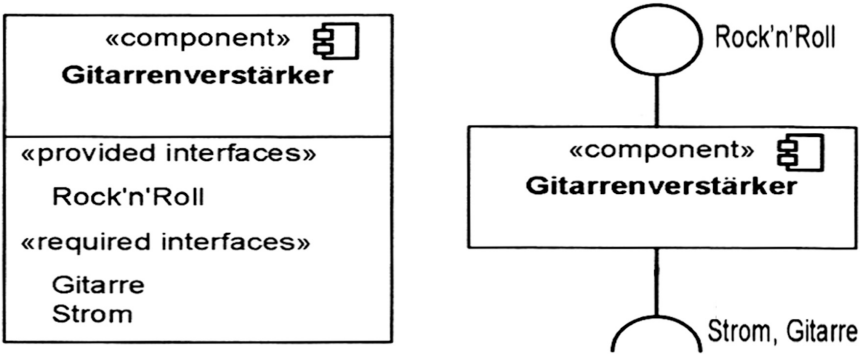
Lösung der Übungsaufgaben

Kapitel 2:

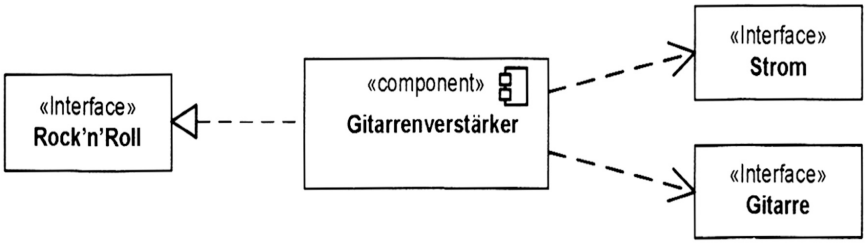
1. a)



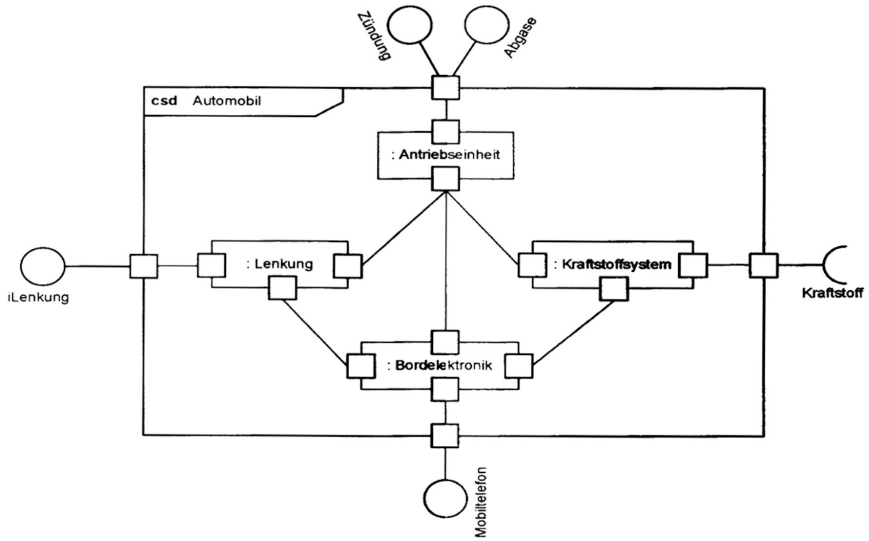
b)



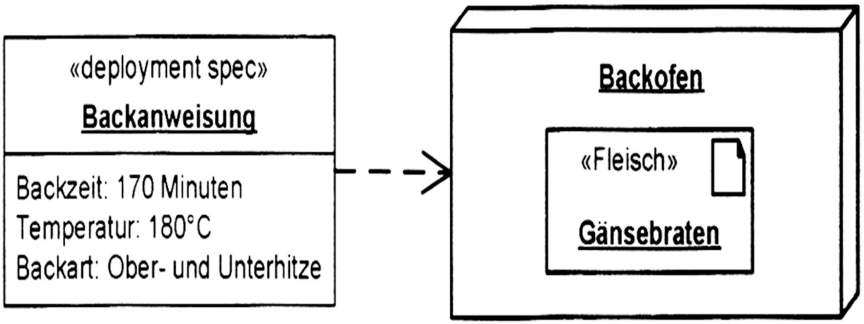
2. a)



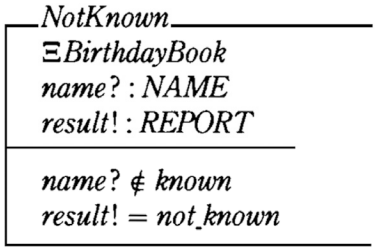
b)


Zündung
Abgase
csd Automobil
: Antriebseinheit
: Lenkung
: Kraftstoffsystem
: Bordelektronik
iLenkung
Kraftstoff
Mobiltelefon

3.


«deployment spec»
Backanweisung
Backzeit: 170 Minuten
Temperatur: 180°C
Backart: Ober- und Unterhitze
Backofen
«Fleisch»
Gänsebraten

4.



$RFindBirthday \mathrel{\widehat=} (FindBirthday \land Success) \lor NotKnown$

5.

Die Domains dieser zwei Funktionen sind schon mal gleich, denn:

dom *birthday´* = *knwon´* (Invarianz hinterher)

= *{i:1..hwm´ • names´(i)}* (aus *Abs*)

= *{i:1..hwm • names´(i)}* ∪ *{names´(hwm´)}*
(weil *hwm´=hwm+1*)

= *{i:1..hwm • names(i)}* ∪ *{name?}*
(weil *names´=names* ⊕ *{hwm´↦name?}*)

= *knwon* ∪ *{name?}* (aus *Abs*)

= dom *birthday* ∪ *{name?}* (Invarianz vorher)

Nun gibt es keine Veränderung in irgend einem Teil des Arrays, die nicht schon vor der Benutzung dieser Operation in Gebrauch war, sodass für alle *i* aus *1...hwm* gilt:

$$names´(i) = names(i) \land dates´(i) = dates(i)$$

Für jedes *i* aus diesem Bereich gilt somit

birthday´(names´(i)) = *dates´(i)* (aus *Abs*)

= *dates(i)* (*dates* bleibt unverändert)

= *birthday(names(i))* (aus *Abs*)

Für den neuen Namen, welcher mit dem Index $hwm'=hwm+1$ gespeichert wird, gilt dann entsprechend

$$birthday'(names'(hwm')) = dates'(hwm') \quad \text{(aus } Abs\text{)}$$
$$= date?$$

Damit sind aber auch die beiden Funktionen

$$birthday'$$

und

$$birthday \cup \{name? \mapsto date?\}$$

gleich, was heißt, dass sich der abstrakte Zustand vor und nach der Operation sich genau so verhält wie von *AddBirthday* vorgegeben.

6.

AbsCards

$$cards : \mathbb{P}\ NAME$$
$$cardlist : \mathbb{N}_1 \rightarrow NAME$$
$$ncards : \mathbb{N}$$

$$cards = \{ i : 1 \,..\, ncards \bullet cardlist(i) \}$$

Remind1

$$\Xi BirthdayBook1$$
$$today? : DATE$$
$$cardlist! : \mathbb{N}_1 \rightarrow NAME$$
$$ncards! : \mathbb{N}$$

$$\{ i : 1 \,..\, ncards! \bullet cardlist!(i) \}$$
$$= \{ j : 1 \,..\, hwm \mid dates(j) = today? \bullet names(j) \}$$

InitBirthdayBook1

$$BirthdayBook1$$

$$hwm = 0$$

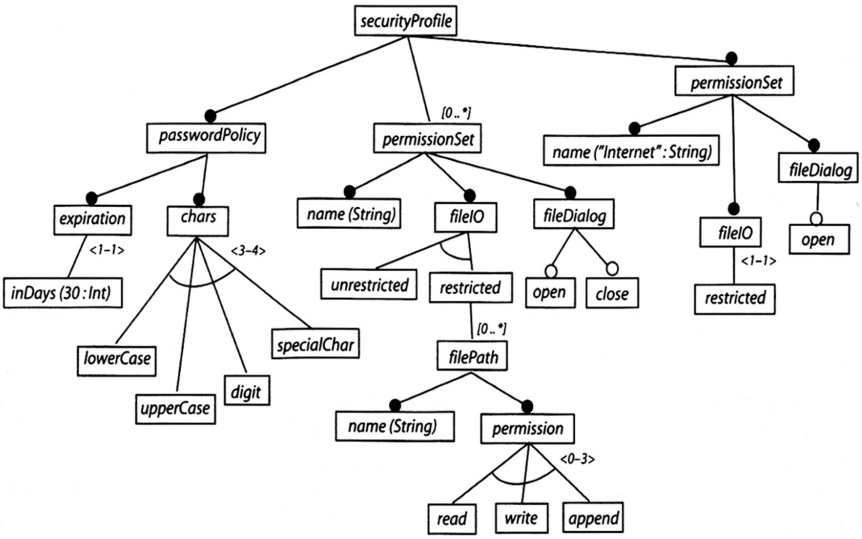
7. Wir beweisen die die Behauptung $\neg b \vee \neg w$ durch Induktion.

Induktionsanfang
Bei der Initialisierung ist $bposn = wposn = \varnothing$.
Da $inLine(\varnothing) = false$, ist der Induktionsanfang bewiesen.

Induktionsschritt

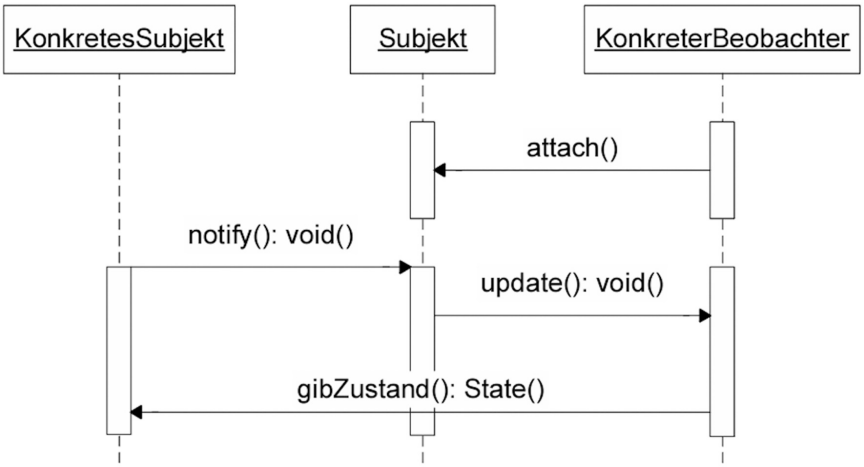
1. Die Operation $gameResult$ ändert weder $bposn$ noch $wposn$.
2. Die Operation $blackMove$ ist nur dann anwendbar, wenn die Voraussetzung $\neg\ over$ gilt. (Operation $selectPosn$ in Klasse $ActivePlayer$).
   Aus der Definition von $over$ ($over \Leftrightarrow w \vee b \vee free = \varnothing$) folgt daher, dass $\neg\ w$ gilt.
   Die Methode $blackMove$ verändert $wposn$ nicht. Daher bleibt $\neg\ w$ wahr. Somit ist auch Satz 1 wahr.
3. Für die Operation $whiteMove$ gilt (aus Gründen der Symmetrie) dasselbe wie für $blackMove$.

8.

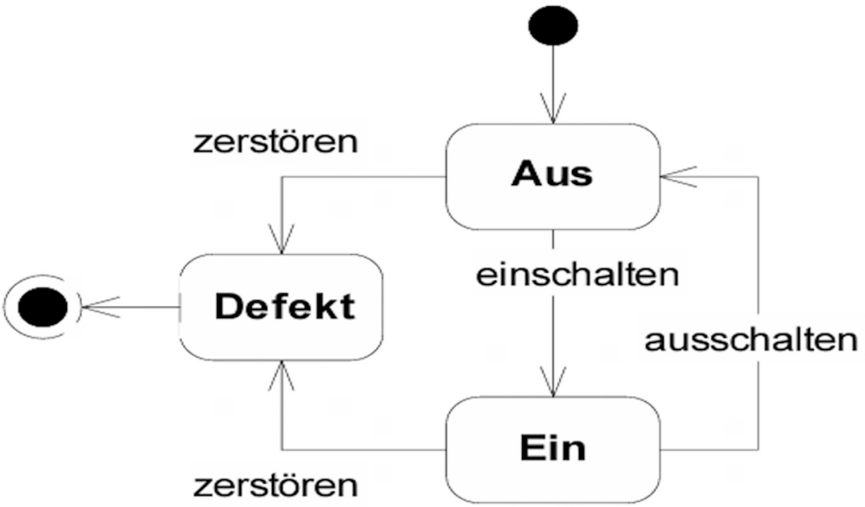


Kapitel 3:

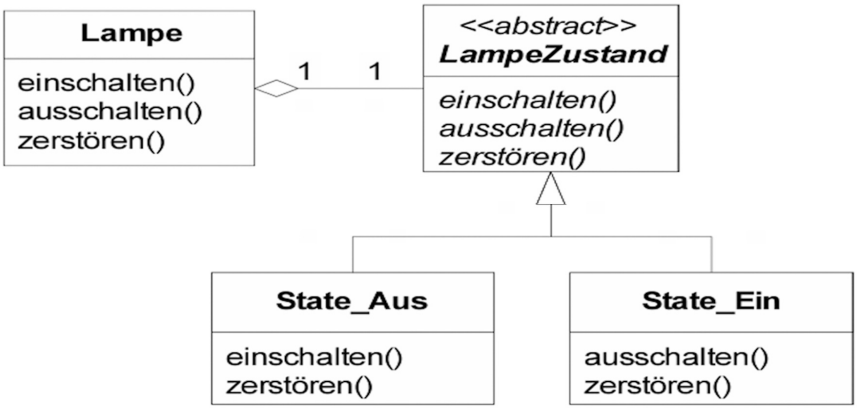
1. Sequenzdiagramm des Beobachtermusters:



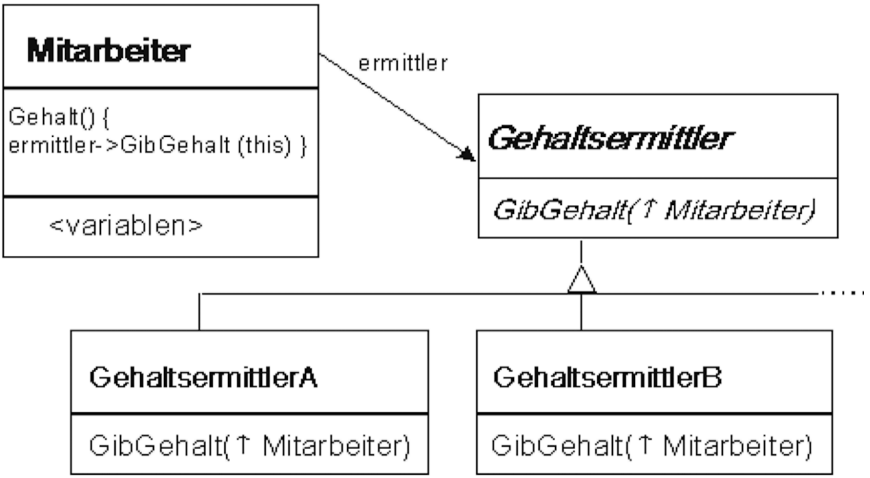
2. a) Zustandsdiagramm:



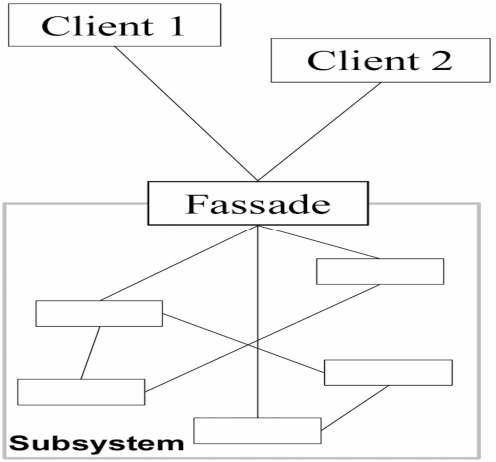
b) Entwurfsmuster „Zustand“:



3. „Entwurfsmusterfreundliche“ Darstellung:

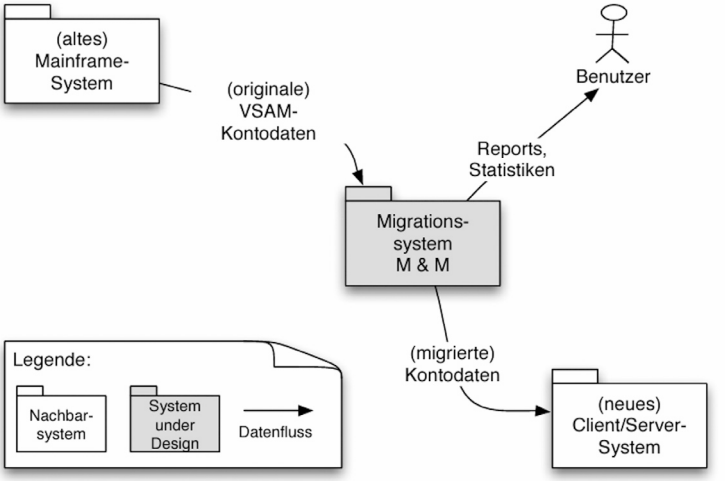


4.

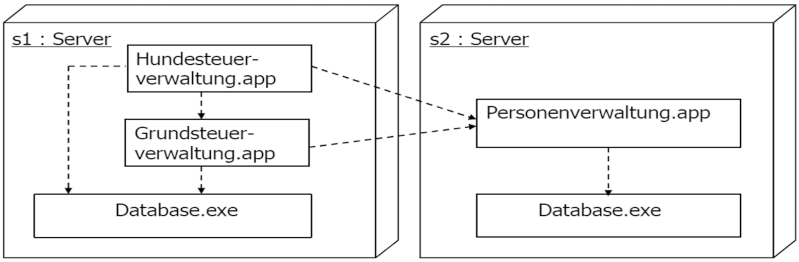

Client 1
Client 2
Fassade
Subsystem

Kapitel 4:

1.


(altes)
Mainframe-
System
(originale)
VSAM-
Kontodaten
Benutzer
Reports,
Statistiken
Migrations-
system
M & M
(migrierte)
Kontodaten
(neues)
Client/Server-
System
Legende:
Nachbar-
system
System
under
Design
Datenfluss

2.



Kapitel 5:

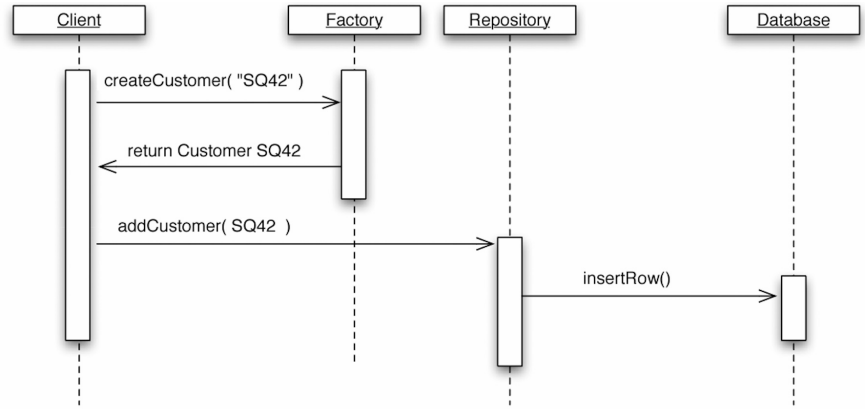
1. Dieses Programm stellt fest, ob eine eingegebene Zahl (n) eine Primzahl ist oder nicht. Zunächst bestimmen wir die Anzahl der elementaren Operationen. Die beiden Wertzuweisungen am Anfang ($p=2$ und $b=true$) stellen zwei elementare Operationen dar. In der *Do-While*-Zeile stellen $*, \leq, \wedge$ je eine elementare Operation dar, das sind also drei.
Diese Zeile wird maximal $int(\sqrt{n})$-mal ausgeführt, so dass diese Zeile insgesamt höchstens $3int(\sqrt{n})$-mal ausgeführt wird.
Im Innern der Schleife findet sich je für $n \bmod p, =0, p=$ und $p+1$ eine elementare Anweisung, das sind insgesamt vier.
Nun wird ins Innere der Schleife $[int(\sqrt{n})-1]$-mal verzweigt, so dass dies dann $4[int(\sqrt{n})-1]$ elementare Operationen sind.
Wird die IF-Anweisung bejaht, so kommt noch die eine Wertzuweisung $b=False$ hinzu.
Damit erhält man für das Innere der Schleife maximal $4[int(\sqrt{n})-1]+1$ elementare Operationen.
Die letzte Wertzuweisung $PrimeCheck=b$ stellt noch eine weitere elementare Operation dar, so dass wir damit im maximal an elementaren Operationen für den Algorithmus erhalten:
$2 + 3int(\sqrt{n}) + 4[int(\sqrt{n})-1]+1+1 = 7int(\sqrt{n})$.
Damit ergibt sich für unsere totale Kostenfunktion: $t(n) \leq 7int(\sqrt{n})$.
Insbesondere ergibt sich für
   a) n ist Primzahl: $t(n) = 7int(\sqrt{n})-1$
   b) n ist Quadrat einer Primzahl: $t(n) = 7int(\sqrt{n})$
   c) n ist gerade: $t(n) = 14$.

2. Zu Software der Kategorie 0 könnte man folgendes zählen:

   (a) Bei Simulationsproblemen braucht man immer eine virtuelle Zeit. Dazu baut man eine „Zeitmaschine“; diese hat die Kategorie 0

   (b) Viele Systeme (z.B. Word) besitzen einen „undo/redo“-Mechanismus; diese Softwarekomponente kann man auch zur Kategorie 0 zählen

3.

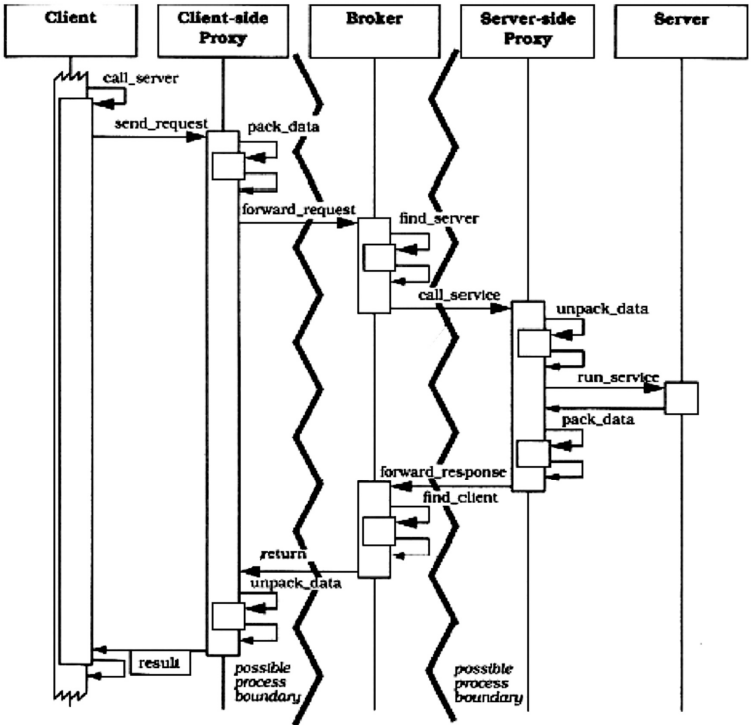


4. Die jeweiligen Komplexitäten sind:

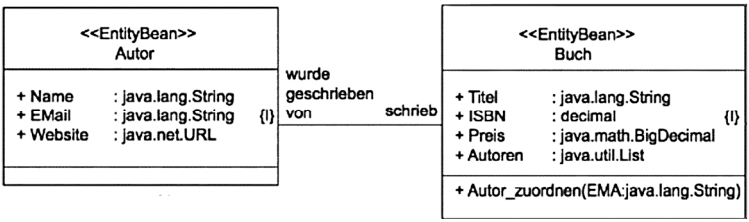
Wurzelkategorie	0
Swing	1
Kartenspiel	1
Dateisystem	1
Kartenspeil-GUI	2
Skat	2
Kartenspiel-GUI-Swing	5
Skatstrategie	3

Kapitel 6:

1.



2. Das UML-Diagramm aus der Aufgabenstellung ist (normalerweise) einem weitaus geringerem Wandel unterlegen als das in der unteren Abbildung, wo das PIM aus der ursprünglichen Abbildung plattformabhängig - über entsprechende Annotationen - an die Plattformen Java bzgl. der Sprache und EJB (Enterprise Java Beans) als Middleware gebunden wurde:



Kapitel 7:

1. Siehe z.B.:

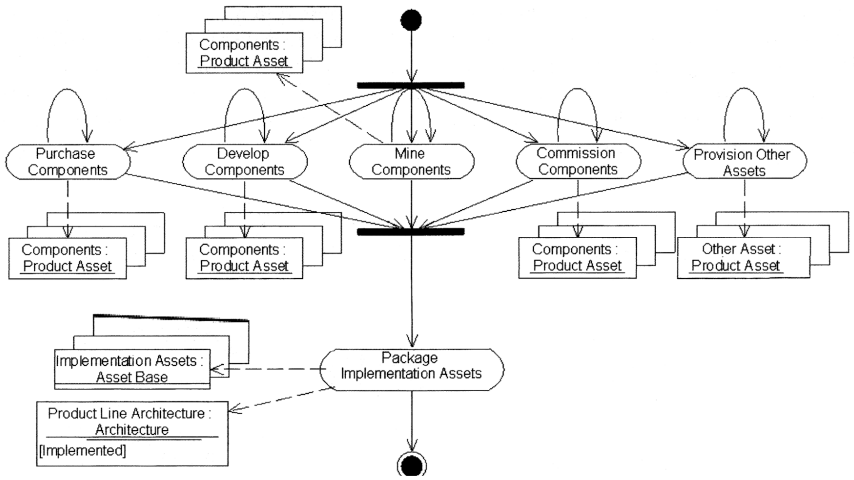
http://files.getdropbox.com/u/45486/arc42-downloads/Arc42-Documentation-HowTo-V3.pdf

oder allgemeiner: http://www.arc42.de/download/downloads.html

oder konkretes Beispiel in Starke, G.: *Effektive Software-Architekturen*, Hanser, 2008, Seite 345ff

Kapitel 8:

1.



Weiterführende Literatur:

G. Starke: *Effektive Software-Architekturen*, Hanser, 2009

E. Gamma et. all: *Design Pattern,* Addison-Wesley, 1996

R. Reussner u. W. Hasselbring: *Handbuch der Software-Architektur*, dpunkt.verlag 2006

J. Greenfield et. all, *Software-Factories*, Wiley Publishing 2004

G. Smith, *The Object-Z Specification Language*, Kluver Academic Publishers, 2000

Index:

4+1-Sichtenmodell 96
4-Sichten-Modell nach Starke 98

abgeleitete Klassen 18
Abhängige Klassen 20
Abstract Factory 77
Abstrakte Fabrik 76, 77
ACME 64
Action 87
Adapter 76, 80
ADL 9, 10, 102
ADM 133
Aggregationen 18, 19, 20
Aktivitätsdiagramm 12, 28
Algorithmus 110
Anwendungsfalldiagramm 12, 27
Architekturbeschreibungssprache 9, 10, 102
Architektur-Muster 123
Architektur-Standards 123
Architekturstil 123
Assoziation 14
Assoziationsklasse 15
Ausführungssicht 105

Basisklassen 17
Bausteinsicht 98. 102
Befehl 76, 87
Beobachter 76, 88
Besucher 76, 89
Blackboard-Achitekturmuster 127
Blackbox-Darstellung 102
Body 80
Bridge 80
Brücke 76, 80
Builder 78

Chain of Responsibility 93
Classifier 12
Code Architecture View 98
Command 87
Composite 83
Conceptual Architecture View 97, 100
CORBA - IDL 65, 69, 104
Cursor 90
CWM 138
Czarnecki-Eisenecker-Notation 68

Dekorierer 76, 81
Deployment View 97
Deployment-Diagramme 108
Dokumentation von Architekturen 143
Domain Driven Design 119, 120

Einsatz-Diagramme 108
elementare Operationen 113
Enterprise Architectures 133
Entitätsklasse 22
Entwurfsmuster 74
Erbauer 76, 78
Ergonomie 143
Evaluierungen 143
Execution Architecture View 98
Exemplare 14

Fabrikmethode 76, 77
Facade 82
FactoryMethod 77
Fassade 76, 82
Feature-Delokalisierung 118
features-Teil 51
Filter 125
Fliegengewicht 76, 82
Flyweight 82
FODA-Notation 68

Generalisierung 17, 18
Gerichtete Assoziation 19
Größe der Eingabe 114

Handle 80

IDL 65
Infrastruktursichten 99, 107
Inputgrößen 114
Instanzen 14
Integritätsbedingungen 13
Interaktionsüberblicksdiagramm 33
Interaktionsübersichtsdiagramm 13, 33
Interpreter 76, 85
inverse Rolle 17
ISO/IEC 10746x 139
Iterator 76, 90

Kapselung 119
Kardinalitäten 14
Kardinalitätsbeschränkungen 14
Klassen 13
Klassendiagamm 14

Klassendiagramm 11
Knoten 107
Kollaborationsdiagramme 106
Kollaborationstyp 23
Kommunikationsdiagramm 13, 31
Komplexitätsprobleme 110
Komponenten 5, 12, 24
Komponentendiagramm 12, 23
Komponetenklasse 22
Komposition 19
Kompositionsstrukturdiagramm 12, 22
Kompositum 76, 83
Konnektoren 5
Kontextsicht 98, 100
Kostenfunktion 113
Kruchten-Sichtenmodell 96

Larch 59
Laufzeitsicht 99, 105
Layer Bridge 123
Layers 123
Linkattribute 15
Links 14, 15
Logical View 97,100

MDA 136
Mediator 92
Memento 91
Module Architecture View 97
MOF 137
Momento 76

n-Tier-Architekturen 124

Oberklassen 18
Object-Z 34, 48
Objektdiagramm 12, 15
Objekte 13
Observer 88
OCL 61
oneway 65
Operations-Schema 35
Opportunistische Schließen 127
Ordnung einer Funktion 113
Paketdiagramm 12, 25

Peer-to-Peer-Architektur 128
Peer-to-Peer-Architekturmuster 128
Peer-to-Peer-Netzwerk 128
Perr-to-Peer-Protokoll 128
Perr-to-Peer-System 128
Persistenz 143
Physical View 97
Pipes 125
Policy 91
Port-Notation 22, 23
Posets 59
Problem eines Handlungsreisenden 112
Process View 97
Produktlinien 149
Protection Proxy 85
Prototyp 76, 79
Prototyping 143
Proxy 76, 84

Rapide 59
Raumkomplexität eines Algorithmus 115
Realisierungsbeziehung 21
Rechenvorschrift 110
Referenzarchitekturen 104
Rekonstruktionsproblem 119
Remote Proxy 84
RM-ODP 139
Rolle 14, 16

Schablonenmethode 76, 86
Schichten 123
Schnittstellen 22
Schnittstellenklasse 22
Schnittstellenrealisierungsbeziehung 21
semantische Korrektheit 111
Sequenzdiagramm 13, 30
Servent 128
Sicherheit 143
Sichtbarkeitsliste 51
Sichtenmodelle 96
Siemens-Sichten 97
Simulation 143
Singleton 76, 79
SOA 130
Socket-Notation 22
Software Factories 149
Software-Architektur 5
Software-Kategorien 115
Spezialisierung 17, 18
Stakeholder 96
State 93
Stereotypen 22
Steuerklasse 22
Strategie 76, 91
Strategy 91
Subklassen 18
Surrogat 84

Template Method 86
Tests 143
Tiers 123, 124
Timingdiagramm 13, 32
TOGAF 133
Token 91
ToolKit 77
totale Kostenfunktion 113
Traceability-Problem 119
Transaction 87
Travelling Salesman Problem 112

ubiquitous language 120
U-Case-Diagramm 12
UML 10, 137
Unterklassen 18
Use-Case 12
Use-Case-Diagramm 27
Use-Case-Sicht 97

VDM 59
Vererbung 17, 18
Verifikation 143
Vermittler 76, 92
Verteilungsdiagramm 12, 26
Verteilungssicht 99, 107
Vienna Development Method 59
Virtual Proxy 85
Visitor 89

Whitebox-Darstellung 102
Worse-Case-Kostenfunktion 114
Wrapper 80, 81

xADL 2.0 66
XMI 138

Z 34
Zeitkomplexität eines Algorithmus 114
Zeitverhaltensdiagramm 13
Z-Notation 34
Zustand 76
Zuständigkeitskette 76, 93
Zustandsdiagramm 12, 29
Zustandsmuster 93
Zustands-Schema 35