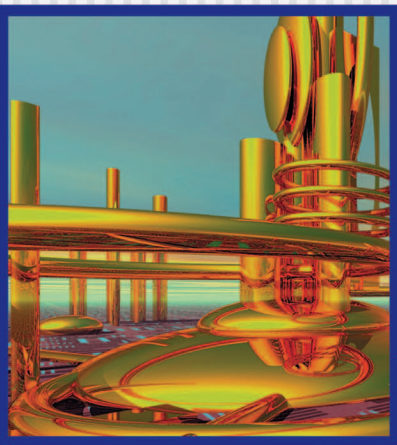
COMPANION WEBSITE
CWS
www.pearson-studium.de



informatik

Ulrike Hammerschall

Verteilte Systeme und Anwendungen

Architekturkonzepte, Standards und Middleware-Technologien

PEARSON
Studium

PEARSON Education

PEARSON Studium

Verteilte Systeme und Anwendungen

Architekturkonzepte, Standards und Middleware-Technologien

Ulrike Hammerschall

PEARSON Studium

ein Imprint von Pearson Education
München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Literatur

Dienste

The Open Group: Distributed TP: Reference Model; Berkshire, U.K., X/Open Company Ltd, Version 3, 1996.

The Open Group: Distributed TP: The XA Specification; Berkshire, U.K., X/Open Company Ltd, 1992.

ISO/IEC: International Standard ANSI/ISO/IEC 9075-1:1999 „Database Language SQL“.

Ian Gorton: Enterprise Transaction Processing Systems; Wokingham, Addison Wesley, 2000.

Gernot Starke: Effektive Software-Architekturen – ein praktischer Leitfaden; München. Hanser 2002.

Johannes Siedersleben: Moderne Software-Architektur; München, dpunkt Verlag, 2004.

George Coulouris, Jean Dollimore, Tim Kindberg: Distributed Systems, Concepts and Design. Wokingham: Addison Wesley, 3.Ausgabe 2001 (dt. Verteilte Systeme – Konzepte und Design; München, Pearson Studium, 3., überarbeitete Auflage 2002)

Andrew Tanenbaum, Marten van Steen: Distributed Systems; Englewood Cliffs, NJ: Prentice Hall 2003 (dt. Verteilte Systeme – Grundlagen und Paradigmen; München, Pearson Studium 2003)

Komponentenmodelle

Clemens Szyperski: Component Software, Beyond Object-Oriented Programming; Addison Wesley, Second Edition, 2002.

Volker Gruhn, Andreas Thiel: Komponentenmodelle – DCOM, JavaBeans, Enterprise JavaBeans, CORBA; Addison-Wesley, 2000.

Transaktionsmonitore

Jim Gray, Andreas Reuter: Transaction Processing, Concepts and Techniques; Burlington, MA, Elsevier, Morgan Kaufmann Publisher, 1992.

Philip A. Bernstein, Eric Newcomer: Principles of Transaction Processing; Burlington, MA, Elsevier, Morgan Kaufmann Publisher, 1997.

Entwurf verteilter Anwendungen

4

ÜBERBLICK

Eine Anwendung hat die Aufgabe, Anforderungen möglicher Anwender geeignet zu erfüllen. Anforderungen können funktionaler oder nichtfunktionaler Natur sein. Typische funktionale Anforderungen sind alle fachlichen Dienste, die ein Anwender nutzen möchte – wie beispielsweise: ein Buch in einem Internetshop kaufen, ein Ticket bei einem Online-Buchungssystem reservieren oder ein Navigationssystem nutzen.

Nichtfunktionale Anforderungen sind Anforderungen, die nicht direkt mit der Fachlichkeit zusammenhängen, dennoch für die Benutzung der Anwendung relevant sind. Nichtfunktionale Anforderungen machen beispielsweise Aussagen darüber, welche Verfügbarkeitsanforderungen für eine Anwendung einzuhalten sind, welche Antwortzeiten erwartet werden oder welche Sicherheitsanforderungen zu erfüllen sind.

Funktionale Anforderungen werden von der Anwendung realisiert. Die Umsetzung der nichtfunktionalen Anforderungen erfolgt in großen Teilen durch das verteilte System bzw. die verwendete Middleware. Einen nicht zu unterschätzenden Einfluss hat hier auch die Architektur der Anwendung.

4.1 Softwarearchitektur

Wie ein Architekt die Grundstruktur eines Gebäudes plant, so plant der Softwarearchitekt die Grundstruktur einer Anwendung. Der Vergleich zum Gebäudearchitekten wurde an dieser Stelle nicht zufällig gewählt. Aufgaben und Probleme bei der Planung eines Gebäudes ähneln stark den Aufgaben und Problemen, denen sich ein Softwarearchitekt stellen muss. Dies geht soweit, dass die Disziplin der Softwarearchitektur eine Reihe von Konzepten und Ideen der Gebäudearchitekten in ihre Arbeit übernommen hat.

Die Struktur einer Anwendung wird, ebenfalls in Anlehnung an Gebäudearchitekturen, auch Softwarearchitektur genannt. Es gibt heute eine Reihe von Definitionen für Softwarearchitekturen. Bass, Clements und Kazman definieren in ihrem Buch „Software Architecture in Practice“ eine Softwarearchitektur wie folgt:

The software architecture of a program or computing system is the structure of structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Eine Softwarearchitektur beschreibt die Elemente einer Anwendung und ihre Beziehungen untereinander. Sie ist maßgeblich dafür verantwortlich, dass die Anwendung alle funktionalen und insbesondere alle nichtfunktionalen Anforderungen der zukünftigen Anwender erfüllt.

Nicht jede Architektur ist für jede Anwendung geeignet. Wurde eine falsche Architektur gewählt, eine die für die Anforderungen nicht „trägt“, kann dies beispielsweise dazu führen, dass die Anwendung zu langsam ist, dass sie fehlerhaft arbeitet, dass sie nicht den gewünschten Grad der Nebenläufigkeit aufbringt oder geforderte Verfügbarkeitsanforderungen nicht einhalten kann.

> ***Systemarchitektur*** Während Softwarearchitekturen ausschließlich die Architektur der Softwareanwendung betrachten, befasst sich eine Systemarchitektur mit Hardware- und Softwareanteilen des vollständigen Systems. Zur Systemarchitektur zählen sowohl die Architektur der verteilten Anwendung (Softwarearchitektur), wie auch Architektur des verteilten Systems.

4.1.1 Komponenten

Komponenten sind Strukturelemente einer Softwarearchitektur. Sie sind eine der tragenden Säulen komponentenbasierter Softwareentwicklung. Komponentenbasierte Entwicklung ist zwar von der Idee her nicht wirklich neu, hat sich aber vor allem in den letzten Jahren zu einer Art Hype entwickelt.

Umso erstaunlicher scheint es, dass in der Informatik bis heute kein einheitlicher Komponentenbegriff gefunden wurde. Das bedeutet nicht, dass es nicht reichlich Auswahl an Definitionen gäbe. So listet Clemens Szyperski in seinem Buch „Component Software“ gleich mehrere Seiten mit Definitionen zu Komponenten aus der Literatur auf – und liefert zusätzlich eine weitere Definition. Entscheidend aber ist, dass es keinen allgemein anerkannten Standard einer Komponente gibt. Es ist nicht Ziel dieses Buches, diese Lücke zu füllen; es wird auch an dieser Stelle keine Komponentendefinition eingeführt. Vielmehr wird aufgezeigt, warum Komponenten sinnvoll und notwendig sind und welche Eigenschaften sie (mindestens) erfüllen sollten.

Das Problem – Unstrukturierte Programmierung

Die Einführung von Komponenten in der Softwareentwicklung war eine Antwort auf eine Vielzahl von Problemen, die sich mit steigender Komplexität und steigendem Umfang der Anwendungen einstellten. Besonders ein Problem machte Softwareentwicklern und vor allem den Unternehmen zu schaffen: Anwendungen, die unstrukturiert entwickelt wurden, waren auf Dauer nicht wartbar.

Nach ihrer Entwicklung geht eine Anwendung in die Phase des Betriebs und der Wartung über. Im Rahmen der Wartung werden kleinere Änderungen durchgeführt, Fehler behoben oder Funktionalitäten an neue Anforderungen angepasst. In der Wartungsphase wird eine Anwendung häufig von neuen Entwicklern übernommen, die zum Teil erhebliche Schwierigkeiten haben, Struktur und Abläufe der Anwendungen zu verstehen. Die Folgen sind hohe Wartungskosten sowie in manchen Fällen eine kostspielige Neuentwicklung. Dies ist bei guter Strukturierung häufig nicht notwendig.

Es gibt eine Vielzahl verschiedener Möglichkeiten, unstrukturierten und damit unwartbaren Code zu entwickeln. Zwei typische Fehler werden im Folgenden kurz vorgestellt.

Ein von Anfängern gern verübter Fehler ist die Entwicklung von so genanntem „Spaghetticode“. Der Kontrollfluss einer Anwendung windet sich über unstrukturierte Funktionsaufrufe quer durch den Code und ist nicht mehr nachvollziehbar.

> ***„Go To Statement Considered Harmful“*** In seinem legendären Artikel „Go To Statement Considered Harmful“ für die „Communications of the ACM“ erläutert Edsger Dijkstra bereits 1968, dass die Verwendung von Go-To-Statements in Programmen – eine damals sehr beliebte Programmiertechnik – mehr Schaden als Nutzen anrichten würde. Go-To-Statements ermöglichten beliebige Sprünge im Code und waren die Hauptverursacher unverständlicher, verschlungener Kontrollflüsse in Anwendungen. Schon wenige Jahre später waren Go-To-Statements weitgehend aus allen neueren Programmiersprachen verschwunden.

Durch Einführung objektorientierter Sprachen sollte dieses Problem behoben werden. Objektorientierung brachte jedoch nur einen Teilerfolg. Unter der Annahme „wir programmieren objektorientiert, unser Code ist gut strukturiert“ tappten viele Entwickler in eine andere Falle: ungewollte Abhängigkeiten zwischen Objekten, die „Code-Lasagne“.

Der Begriff „Code-Lasagne“ ist eine Anlehnung an den Begriff „Spaghetticode“. In diesem Fall ist jedes Objekt einer Anwendung direkt oder indirekt von jedem anderen Objekt im Code abhängig. Änderungen an einer Stelle können zu ungewollten Seiteneffekten an anderen Stellen im Code führen. Die Behebung von Fehlern wird dadurch äußerst kritisch.

Die Lösung – Komponenten

Generelles Problem bei unstrukturiertem Code sind die vielen inhaltlichen und technischen Abhängigkeiten, die unwillkürlich entstehen. Es existieren im Code keine eigenständigen, unabhängigen Bereiche mehr. Minimale Änderungen an der Funktionalität können sich ungewollt quer durch die gesamte Anwendung auswirken.

Komponenten wurden eingeführt, um Abhängigkeiten im Code zu beherrschen. Eine Komponente dient, wie eine Klasse, zur Strukturierung von Code, ist jedoch größer und kontrolliert an ihren Grenzen Beziehungen zu anderen Komponenten und zur Umgebung. Im Folgenden werden einige kennzeichnende Eigenschaften einer Komponente beschrieben:

- Eine Komponente kapselt Daten und Funktionen (bei prozeduralen Sprachen) oder Methoden (bei objektorientierten Sprachen).
- Der Zugriff auf eine Komponente erfolgt ausschließlich über ihre Schnittstellen. Eine Komponente implementiert eine oder mehrere Schnittstellen.
- Eine Komponente kann zur Laufzeit instanziiert werden. Instanzen von Komponenten werden auch Komponentenobjekte genannt.

Aufgabe des Softwarearchitekten ist es, im Rahmen des Architekturentwurfs Komponenten festzulegen und ihre Schnittstellen zu definieren. Zum Entwurf kann er sich an einigen generellen Regeln orientieren:

- Die Schnittstellen der Komponenten sollten so geplant werden, dass die Abhängigkeiten zwischen den Komponenten minimal bleiben. Das bedeutet: wenige Zugriffe zwischen Komponenten und möglichst keine Zyklen.
- Komponenten sollten so geplant werden, dass wahrscheinliche Änderungen sich nicht über die gesamte Anwendung auswirken, sondern möglichst lokal bleiben.
- Eine Komponente sollte so geplant werden, dass sie entweder technische oder fachliche Aspekte implementiert. Mischungen führen zu ungewollten Abhängigkeiten.
- Eine Komponente sollte so geplant werden, dass als Implementierung beispielsweise eine Open-Source Komponente oder auch eine kommerzielle Komponente verwendet werden kann.
- Eine Komponente sollte so geplant werden, dass Implementierungen der Komponenten für weitere Anwendungen wiederverwendbar sind. Dies gilt insbesondere für technische, in geringerem Maße auch für fachliche Komponenten.

Information Hiding Modules Die Idee der Komponente in der Softwareentwicklung ist nicht wirklich neu. Bereits Anfang der 70er Jahre führte Parnas seine „Information Hiding Modules" ein, die Vorläufer der heutigen Softwarekomponenten. Die Idee war einfach: Ein Modul ist wie eine Komponente Strukturelement einer Anwendung. Zur Identifikation von Modulen gibt Parnas eine Regel an: Ein Modul enthält immer ein Geheimnis, welches es vor der Außenwelt schützt. Über seine Schnittstelle legt es fest, welche Informationen über die Modulgrenze hinaus gegeben werden und welche im Besitz des Moduls verbleiben.

Geheimnis eines Moduls ist beispielsweise die Implementierung eines Suchalgorithmus oder die Berechnung von eindeutigen Identifikatoren. Der Anwender kennt weder den Algorithmus noch das Vorgehen zur Berechnung. Er erhält lediglich auf Anfrage das Suchergebnis bzw. einen fertigen Identifikator. Ziel des Information Hiding ist die Austauschbarkeit und damit die Wiederverwendung der Module. Solange die Schnittstelle gleich bleibt, ist es einerlei, welcher Algorithmus sie tatsächlich implementiert. Module sind reine Entwurfselemente, sie spielen im Gegensatz zu Komponenten zur Laufzeit keine Rolle.

4.1.2 Architekturentwurf im Entwicklungsprozess

Die Entwicklung einer Anwendung erfolgt im Rahmen eines Vorgehensmodells. Ein Vorgehensmodell legt die Abläufe und die Ergebnisse fest, die im Rahmen der Entwicklung zu erstellen sind. Es gibt heute eine Vielzahl von Vorgehensmodellen, wie beispielsweise das Wasserfallmodell oder das Spiralmodell, auf die an dieser Stelle jedoch nicht weiter eingegangen wird. Unabhängig davon, welches Vorgehensmodell gewählt wird, ist vor der Entwicklung einer Anwendung grundsätzlich ihre Architektur zu planen. Der Architekturentwurf fällt im Wesentlichen in die Designphase eines Vorgehensmodells. Wurden alle Anforderungen an die zu entwickelnde Anwendung gesammelt und dokumentiert, kann die Struktur der Anwendung entworfen werden. *Abbildung 4.1* zeigt die Einbettung des Architekturentwurfs in den Entwicklungsprozess.

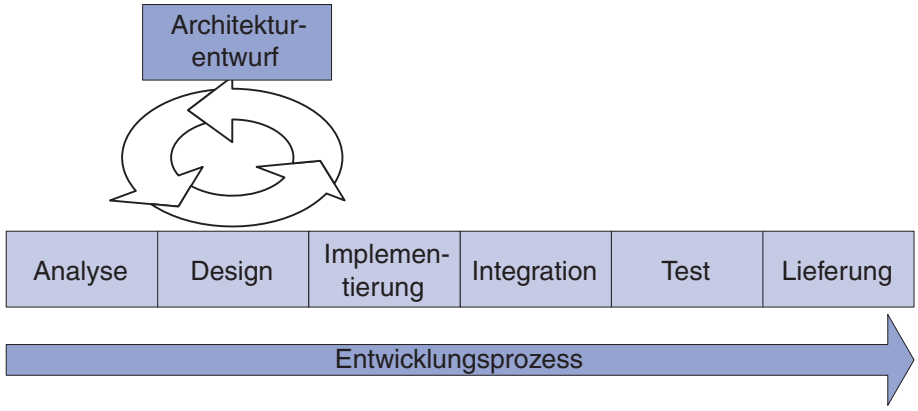


Abbildung 4.1: Architekturentwurf im Entwicklungsprozess

Der Architekturentwurf wird in einem iterativen Prozess durchgeführt. Anhand der Anforderungen wird ein erster Grobentwurf erstellt und seine Tragfähigkeit geprüft. Iterativ wird der Entwurf verfeinert, bis die Architektur feststeht und die Anwendung nach den Vorgaben implementiert werden kann.

4.2 Entwurfsprobleme

Entwurfsprobleme, denen sich ein Softwarearchitekt stellen muss, können vielfältiger Natur sein. Im Folgenden werden beispielhaft einige Entwurfsprobleme vorgestellt, die insbesondere bei interaktiven, datenzentrierten verteilten Anwendungen auftreten.

Entwurf der Anwendungslogik

Die Anwendungslogik als zentrale Einheit einer Anwendung muss geeignet strukturiert werden, so dass alle funktionalen und nichtfunktionalen Anforderungen erfüllt werden können. Strukturierung bedeutet die geeignete Aufteilung der Software in Komponenten, so dass Abhängigkeiten gering und spätere Änderungen lokal bleiben. Jede Komponente erfüllt innerhalb des Anwendungskerns ihre individuelle Aufgabe. Aufgabe des Architekten ist es, einen geeigneten Komponentenschnitt zu finden.

Entwurf der Benutzerschnittstelle

Anwender interagieren über die Benutzerschnittstelle mit der Anwendung. Die Benutzerschnittstelle ist sozusagen die Visitenkarte der Anwendung und kann die Akzeptanz der Anwender entscheidend mitbeeinflussen. Der Entwurf der Benutzerschnittstelle umfasst zum Einen Überlegungen zum graphischen Design der Schnittstelle (Fenstergestaltung, „Look and Feel“), zum Anderen aber auch zur Dialogführung: Wie wird der Anwender durch die Dialoge geführt, welche Hilfestellungen stehen ihm zur Verfügung und wie werden ihm Fehler oder Ausfälle der Anwendung mitgeteilt? Ebenfalls Teil des Entwurfs ist die Prüfung der Benutzereingaben: Anwender können ungewollt oder auch gewollt falsche Eingaben machen. Durch Prüfungen der Eingaben direkt an der Benutzerschnittstelle werden bereits die gröbsten Fehler herausgefiltert. Eine weitere Prüfung erfolgt in der Anwendungslogik.

Entwurf der Client-Server-Schnittstelle

Der Entwurf an der Client-Server-Schnittstelle betrifft die Frage nach dem zu verwendenden Programmiermodell auf Ebene der Anwendung. Zur Auswahl stehen das objektorientierte Modell, das prozedurale Modell und das nachrichtenorientierte Modell. Jedes der Modelle bringt seine Vor- und Nachteile mit sich.

Das prozedurale Modell betrachtet den Server als Diensterbringer. Dienste sind grob-granulare Funktionsaufrufe, die mehrere Aktionen gebündelt anbieten. Schnittstellen, die nach dem prozeduralen Modell entworfen sind, werden auch dienstorientiert bezeichnet. An der Client-Server-Schnittstelle kann die Verwendung einer dienstorientierten Schnittstelle zu einer drastischen Minimierung der Aufrufe führen. Webservices sind typische Vertreter des dienstorientierten Modells.

Das objektorientierte Modell geht davon aus, dass nicht zwischen lokalen und entfernten Objekten unterschieden wird. Der objektorientierte Entwurf der Client-Server-Schnittstelle berücksichtigt keine Verteilungsaspekte. Vorteil des Modells ist, dass es bei objektorientierten Anwendungen an Rechnergrenzen keinen Paradigmenbruch gibt. Objektorientierte Kommunikation kann jedoch auf Grund vieler kleiner Methodenaufrufe über das Netz rasch zu Performance-Einbrüchen führen. *Abbildung 4.2* zeigt an einem Beispiel die Unterschiede zwischen objektorientierter und dienstorientierter Schnittstelle.

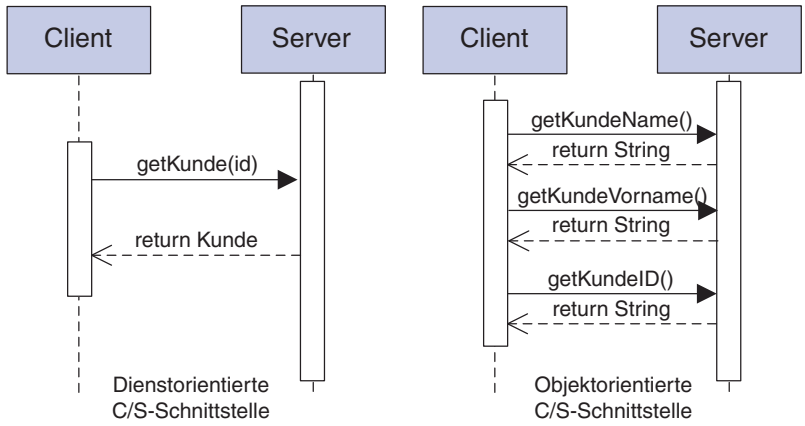


Abbildung 4.2: Vergleich dienstorientierte und objektorientierte Schnittstelle

Das nachrichtenorientierte Modell sieht asynchrone Kommunikation an der Client-Server-Schnittstelle vor. Dieser Fall tritt eher selten auf. Zur Realisierung können die verschiedenen Modelle asynchroner Kommunikation, wie sie in Kapitel 6 beschrieben sind, verwendet werden. Vorteil asynchroner Kommunikation ist die Entkopplung von Client und Server. Der Client ist während der Bearbeitung seines Aufrufs am Server nicht blockiert.

Neben dem Programmiermodell der Schnittstelle ist im Entwurf zu klären, in welcher Form Daten über die Schnittstelle transportiert werden. Im dienstorientierten Modell werden zur Minimierung der Aufrufe häufig Datencontainer wie Vektoren oder zusammengesetzte Objekte verwendet.

Entwurf der Sitzungsverwaltung

Der Entwurf der Sitzungsverwaltung beschäftigt sich mit der Frage, wo und wie die Sitzungsdaten der Anwender geeignet verwaltet werden. Zur Auswahl stehen eine Reihe von Möglichkeiten: Bei Webanwendungen kann eine Sitzungsverwaltung beispielsweise am Browser, am Webserver oder am Anwendungsserver erfolgen, teilweise auch in der Datenbank.

Eine Entwurfsentscheidung kann sich stark auf die Skalierbarkeit und damit auf die Performance einer Anwendung auswirken, sowohl im positiven, wie auch im negativen Sinn. Ein schlechter Entwurf führt zu hohem Speicherverbrauch und im schlimmsten Fall zum Ausfall der Anwendung bei steigender Belastung. Ein guter Entwurf hat zum Ziel, dass eine Anwendung weitgehend unabhängig von der Zahl der Anwendersitzungen ist und skalierbar bleibt.

Entwurf der Anwendungsprozesse (Workflow)

Ein Anwendungsprozess oder Workflow beschreibt einen mehr oder weniger komplexen Ablauf innerhalb einer Anwendung. Im einfachsten Fall handelt es sich um einen einfachen Aufruf. Komplexe Prozesse ziehen sich über mehrere Methodenaufrufe hinweg, die Daten müssen gegebenenfalls in Sitzungen zwischengespeichert werden. Der Prozess selbst braucht Informationen darüber, in welchem Zustand er sich gerade befindet. Der Entwurf von Prozessen umfasst die Überlegung, wie Prozesse in Code umgesetzt werden, wie sie mit der Sitzungsverwaltung zusammenarbeiten und wo die Zustandsdaten zu den Prozessen gespeichert werden.

Entwurf der Datenhaltung

Für datenzentrierte Anwendungen ist ein Konzept für die Datenhaltung zu entwerfen. Das umfasst den Entwurf des Datenbankmodells, den Entwurf des Datenmodells und die Abbildung zwischen den beiden Modellen. Unterstützung erfährt der Architekt dabei durch Persistenzdienstes. Insbesondere übernehmen Persistenzdienst weitgehend die Abbildung zwischen den Modellen.

4.3 Architekturkonzepte

Architekturkonzepte beschreiben Lösungen für häufig auftretende Entwurfsprobleme. Der Softwarearchitekt prüft von Fall zu Fall, ob zu einem aufgetretenen Entwurfsproblem bereits ein Architekturkonzept existiert, und wendet dieses an. Architekturkonzepte sind so allgemeingültig beschrieben, dass sie in vielen ähnlichen Situationen eingesetzt werden können. Ziel ist die Wiederverwendung von Konzepten. Ein Architekt sollte von Erfahrungen anderer Architekten profitieren und bekannte Fehler vermeiden. Im Folgenden werden drei Arten an wiederverwendbaren Architekturkonzepten vorgestellt: Heuristiken, Muster und Musterarchitekturen.

4.3.1 Heuristiken

Heuristiken sind Regeln und Konzepte, die sich in der Praxis bewährt haben und als generelle Richtlinien für Softwarearchitekten gelten. Ein Softwarearchitekt erwirbt sich im Laufe der Zeit einen reichen Erfahrungsschatz, welche Entwurfsentscheidungen sich für welche Entwurfsprobleme bewährt haben. Diese Erfahrungen werden aus konkreten Entwurfssituationen heraus verallgemeinert und anderen Architekten zugänglich gemacht.

Eine Reihe von Heuristiken ist heute bereits Allgemeingut und wird intuitiv eingesetzt, ohne dass dies vielen Softwarearchitekten tatsächlich bewusst ist. Bekanntes Beispiel einer Heuristik ist der Architekturgrundsatz „Separation of Concerns“ (Trennung der Zuständigkeiten), der von Dijkstra in seinem Essay „On the role of scientific thought“ bereits 1974 eingeführt wurde. Der Grundsatz besagt, dass Software so strukturiert sein sollte, dass Zuständigkeiten innerhalb der Anwendung wohl separiert sind und eindeutig Teilsystemen oder Komponenten zugeordnet werden können.

Ebenfalls als Beispiel einer Heuristik können die „Information Hiding Modules“ von Parnas (siehe Kasten) gesehen werden. Heuristiken sind bis heute die wichtigsten Entscheidungshilfen für den Architekturentwurf, auch wenn es zum Teil schwierig ist, sie in einen allgemeinen formalen Rahmen zu bringen. Häufig ist immer noch die mündliche Weitergabe von Erfahrungen die effektivste Form einer Heuristik.

4.3.2 Muster und Musterarchitekturen

Muster (Pattern) beschreiben allgemeine Lösungen zu häufig wiederkehrenden Problemen des Architekturentwurfs. Im Gegensatz zu Heuristiken bleiben sie jedoch nicht allgemein, sondern beschreiben konkret Problem und Lösungsansatz, wenn möglich mit Implementierungsvorschlag.

Ursprünglich ein Konzept der Gebäudearchitektur, finden Muster heute in immer mehr Bereichen der Softwareentwicklung ihre Verwendung. Beispielsweise können Entwicklungs- oder Entwurfsprozesse mit Hilfe von Prozessmustern beschrieben werden, fachliche Anforderungen und Probleme mit Hilfe von Analysemustern. Für den Entwurf von Softwarearchitekturen haben sich eine Reihe von Design- und Architekturmustern etabliert. Ein Muster beschreibt hier fokussiert eine Lösung für ein spezifisches Entwurfsproblem. Es wird niemals erfunden, sondern immer durch Beobachtungen in der Praxis identifiziert. Die Dokumentation eines Musters orientiert sich an vier Kernaspekten:

Kontext: Es wird die Entwurfssituation beschrieben, in der das Entwurfsproblem auftritt, sowie eine Entscheidungshilfe für den Einsatz des Musters gegeben.

Problem: Das Entwurfsproblem, welches durch den Einsatz des Musters gelöst werden soll, wird detailliert dargestellt.

Lösung: Es wird eine Lösung für das Entwurfsproblem vorgestellt. Die Lösung beschreibt detailliert, welche Elemente beteiligt sind und wie ihre Beziehungen untereinander sind. Die Beschreibung kann um zusätzliche Informationen wie Beispiellösungen oder auch mögliche Varianten ergänzt werden.

Konsequenzen: Es werden die Konsequenzen (positiv und negativ) beschrieben, die der Einsatz des Musters mit sich bringen kann. Beispielsweise kann der Einsatz eines Musters einen Performancegewinn bedeuten, jedoch ungünstige Abhängigkeiten im Code erzeugen.

Wie *Abbildung 4.3* zeigt, lassen sich Muster auf unterschiedlichen Abstraktionsebenen im Architekturentwurf einordnen.

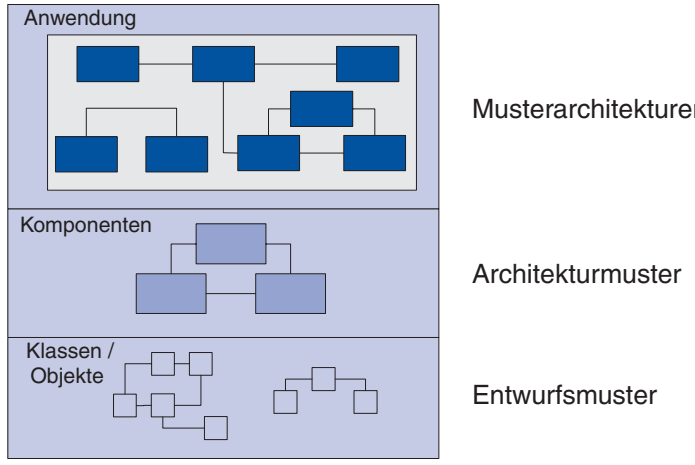


Abbildung 4.3: Abstraktionsstufen von Mustern

Entwurfsmuster

Entwurfsmuster beschreiben Lösungen für Entwurfsprobleme auf der Ebene von Klassen und Modulen. Sie geben den Softwareentwicklern Hilfestellung bei der Implementierung einer Anwendung. Für den Architekturentwurf sind Entwurfsmuster jedoch – trotz ihres Namens – zu fein-granular.

Architekturmuster

Architekturmuster gehen eine Abstraktionsstufe höher. Sie beschreiben strukturelle Prinzipien einer Softwarearchitektur auf der Ebene von Komponenten und Subsystemen. Die Verwendung von Schichten ist ein Beispiel eines Architekturmusters.

Musterarchitekturen

Auf der obersten Hierarchieebene der Muster stehen Musterarchitekturen, häufig auch Referenzarchitekturen genannt. Musterarchitekturen beschreiben nicht so sehr Entwurfslösungen für Teilprobleme einer Softwarearchitektur, sondern konzentrieren sich auf den Entwurf vollständiger Anwendungen. Im Gegensatz zu Mustern ist der Bereich der Musterarchitekturen bisher noch nicht sehr stark entwickelt. Es gibt heute einige unternehmensspezifische Musterarchitekturen sowie eine verschwindende Anzahl allgemeingültiger Musterarchitekturen. Erschwerend kommt hinzu, dass bis heute nicht wirklich festgelegt wurde, was zum Umfang einer Musterarchitektur tatsächlich gehört und was nicht.

Grundsätzlich steckt hinter einer Musterarchitektur die Idee, dass Anwendungen in Gruppen von Anwendungstypen, so genannte Domänen, eingeteilt werden können. Anwendungen, die in einer Domäne sind, bringen ähnliche Entwurfsprobleme mit sich, für die allgemeine Lösungen zu finden sind. Die Lösungskonzepte werden in einer Musterarchitektur zusammengefasst. Musterarchitekturen können technologiespezifisch oder technologieunabhängig formuliert werden.

Da Inhalt und Umfang nicht festgelegt sind, kann eine Musterarchitektur sehr unterschiedlich aussehen. Sie umfasst mindestens Vorgaben zu allgemeinen Entwurfskonzepten und Entwurfsrichtlinien, die sich zum Teil auf Architektur- und Entwurfsmuster abstützen. Zusätzlich werden häufig fertige Frameworks, Komponenten und Bibliotheken angeboten, die eine Musterarchitektur-konforme Entwicklung unterstützen.

In Abschnitt 4.5 wird beispielhaft eine technologieunabhängige Musterarchitektur beschrieben, die sich vor allem auf die Domäne der verteilten betrieblichen Informationssysteme konzentriert. Eine technologiespezifische Musterarchitektur wird in Kapitel 9 vorgestellt.

4.4 Musterkataloge

Muster werden, wie erwähnt, nicht erfunden, sie werden in konkreten Projekten identifiziert und aufbereitet. Häufig verwendete Muster werden in Musterkatalogen zusammengefasst und veröffentlicht. Dort werden sie einem stetigen Überarbeitungsprozess unterzogen. Neue Muster werden hinzugefügt, existierende werden überarbeitet. Muster, die sich als ungeeignet oder überflüssig erwiesen haben, werden gegebenenfalls. wieder aus dem Katalog entfernt.

Copyright