6. Konzipieren von Lösungen

6.1 Motivation

Für die Entwicklung von Kleinsoftware (typisch durch eine Person für den Eigengebrauch) ist kein systematischer Entwurf notwendig. Allenfalls werden Skizzen gemacht, im Übrigen wird direkt programmiert.

Größere Software dagegen braucht einen systematischen, strukturierten Aufbau, der nur durch ein sorgfältig erstelltes Lösungskonzept sichergestellt werden kann. Ohne ein solches Lösungskonzept ist es nicht oder nur mit großem Aufwand möglich

- die Lösung zu verstehen,
- die Entwicklungsarbeit auf mehrere Personen zu verteilen (Software-Entwicklung ist Teamarbeit!),
- die Lösung in vorhandene Software einzubetten (dazu müssen die Schnittstellen und Wechselwirkungen verstanden werden),
- die Lösung zu verteilen (auf mehrere Rechner, ggf. auch geographisch).

Ferner legt ein gutes Lösungskonzept den Grundstein für ein leicht pflegbares System. Es erleichtert das Auffinden zu ändernder Teile, ermöglicht lokale Änderungen unter Wahrung der ursprünglichen Strukturen und erlaubt, die Auswirkungen von Erweiterungen und Ergänzungen abzuschätzen und zu begrenzen.

In der Konzipierung begangene Fehler sind häufig gar nicht oder mit nur großem Aufwand reparierbar. Soll Software beschafft werden, so muss in der Konzipierung dafür gesorgt werden, dass Aufgabe und Lösung hinreichend zusammenpassen. Sorgfältiges Konzipieren ist daher für jede Art von professionell entwickelter Software wirtschaftlich.

6.2 Definitionen und Begriffe

Das *Konzipieren einer Lösung* ist die Erstellung und Dokumentation des *Architekturentwurfs* oder Grobentwurfs eines Systems. Dabei werden die wesentlichen Komponenten der Lösung und die Interaktionen zwischen diesen Komponenten festgelegt. Im Englischen spricht man von *Architectural Design*.

DEFINITION 6.1. *Architektur (architecture).* Die Organisationsstruktur eines Systems oder einer Komponente (IEEE 610.12).

DEFINITION 6.2. *Entwurf (design).* 1. Der Prozess des Definierens von Architektur, Komponenten, Schnittstellen und anderen Charakteristika eines Systems oder einer Komponente. 2. Das Ergebnis des Prozesses gemäß 1. (IEEE 610.12).

DEFINITION 6.3. *Lösungskonzept.* Das Dokument, welches das Konzept der Lösung, d.h. die Architektur der zu erstellenden Software dokumentiert. Synonyme: *Software-Architektur, Systemarchitektur, software architecture, system architecture.*

© 1998-2005 Martin Glinz. Alle Rechte vorbehalten. Reproduktion zum nicht kommerziellen Gebrauch mit Quellenangabe gestattet. Reproduktion - auch auszugsweise - zum kommerziellen Gebrauch nur mit schriftlicher Bewilligung des Verfassers gestattet.

Hinweis: Wenn von «System» die Rede ist, kann sowohl die zu erstellende Software als auch die technische bzw. organisatorische Umgebung, in welche die Software eingebettet ist, gemeint sein. Die Unterscheidung ergibt sich aus dem Kontext oder durch explizites Rückfragen. In diesem Kapitel meint «System» immer die zu erstellende Software, sofern nicht ausdrücklich etwas anderes gesagt wird.

6.3 Entwurfsprinzipien

6.3.1 Strukturen und Abstraktionen

Ein systematischer Aufbau ist die Grundlage jedes guten Entwurfs. Die Systematik besteht in der Wahl bzw. Verwendung geeigneter Strukturen und Abstraktionen. Die *Strukturen* gliedern das Lösungskonzept in *Komponenten* (Module, Prozesse, externe Akteure,...) und *Interaktionen* zwischen Komponenten (Aufrufe, Beziehungen, Nachrichten,...). Die *Abstraktionen* unterstützen das Verstehen des Konzepts durch systematische Vergröberung und Verfeinerung der Darstellung nach verschiedenen Kriterien. Sie ermöglichen dabei

- das Sichtbarmachen großer Zusammenhänge unter Weglassung der Details
- die Darstellung eines Details unter Weglassung/starker Vergröberung des Rests
- die Herstellung eines systematischen Zusammenhangs zwischen Übersichten und Detailsichten.

In der Konzipierung werden hauptsächlich vier Abstraktionsarten verwendet: Komposition, Benutzung, Generalisierung, Aspekte.

- Die hierarchische Zerlegung von komplexen Lösungen in gekapselte, in sich geschlossene Teillösungen (siehe Abschnitt 6.3.2) verwendet die *Kompositionsabstraktion*.
- Die Verwendung von Komponenten ohne Kenntnis ihres inneren Aufbaus (siehe Abschnitte 6.3.3, 6.6.2 und Bild 6.3) ist eine Anwendung der *Benutzungsabstraktion*.
- Die Bildung von Klassenhierarchien im objektorientierten Entwurf (vgl. Abschnitt 6.6.3), bei der jedes Objekt einer Unterklasse einen Spezialfall von Objekten aller ihrer Oberklassen darstellt, basiert auf der *Spezialisierungsabstraktion*.
- Die separate Darstellung von Querschnittsaufgaben (vgl. Abschnitt 6.3.7) ist ein Beispiel für die Verwendung der *Aspektabstraktion*.

6.3.2 Modularität

Die Modularisierung ist eine der Hauptaufgaben beim Konzipieren von Lösungen. Sie gliedert eine Lösung in kleinere, besser verstehbare und überschaubare Teillösungen. Module können rekursiv wieder in Module zerlegt werden, bis die Teillösungen klein genug sind. Modularisierung ist *das* Entwurfsprinzip schlechthin. Ohne eine gute Modularisierung ist die Komplexität großer Systeme nicht mehr beherrschbar.

DEFINITION 6.4. *Modul (module).* Eine benannte, klar abgegrenzte Komponente eines Systems.

Bei der Modularisierung kommt es wesentlich darauf an, dass nicht irgendwie zerlegt wird, sondern so, dass die Module in sich geschlossene Teillösungen mit möglichst wenigen, wohldefinierten Interaktionen zu anderen Modulen bilden. Wir verlangen von einer guten Modularisierung, dass sie folgendes leistet:

- Jeder Modul ist eine in sich geschlossene Einheit
- die Verwendung des Moduls erfordert keine Kenntnisse über seinen inneren Aufbau

- die Kommunikation eines Moduls mit seiner Umgebung erfolgt ausschließlich über eine klar definierte Schnittstelle
- Änderungen im Inneren eines Moduls, welche seine Schnittstelle unverändert lassen, haben keine Rückwirkungen auf das übrige System
- die Korrektheit des Moduls ist ohne Kenntnis seiner Einbettung ins Gesamtsystem nachprüfbar.

Die *Güte* einer Modularisierung lässt sich im Wesentlichen an folgenden drei Kriterien messen: den Maßen Kohäsion und Kopplung (Stevens, Myers und Constantine, 1974) sowie der geeigneten Berücksichtigung von Ressourcen (vgl. Abschnitt 6.3.6).

DEFINITION 6.5. *Kohäsion (cohesion).* Ein Maß für die Stärke des inneren Zusammenhangs eines Moduls. Je höher die Kohäsion ist, desto besser.

Module mit *zufälliger* Kohäsion (bunt zusammengewürfelte Teile) oder *zeitlicher* Kohäsion (zusammengefasst ist, was zeitlich miteinander auszuführen ist) sind softwaretechnisch schlecht und sollten vermieden werden. Anzustreben sind *funktionale* Kohäsion (der Modul realisiert eine in sich geschlossene Funktion) oder *objektbezogene* Kohäsion (der Modul enthält ein Datenobjekt und alle zu seiner Bearbeitung erforderlichen Operationen). Dazwischen gibt es verschiedene Kohäsionsstufen (*logisch, kommunikativ, sequentiell, prozedural*), die hier nicht näher diskutiert werden sollen. Für Details sei auf Stevens, Myers und Constantine (1974) sowie auf Page-Jones (1988) verwiesen.

DEFINITION 6.6. *Kopplung (coupling).* Ein Maß für die Abhängigkeit zwischen zwei Modulen. Die Modularisierung ist umso besser, je geringer die wechselseitige Kopplung zwischen den Modulen ist.

Inhaltskopplung (ein Modul verändert direkt lokale Daten oder Code eines anderen Moduls) ist unter allen Umständen zu vermeiden. *Globale* Kopplung (Daten, die von allen Modulen gelesen und verändert werden können) ist zu vermeiden oder allenfalls auf wenige Daten zu beschränken. *Datenbereichskopplung* (Module kommunizieren über Datenbereiche, die sie gemeinsam haben, oder sich gegenseitig zur Verfügung stellen) ist akzeptabel.

Anzustreben ist *Datenkopplung* (nur die tatsächlich benötigten Daten werden über wohldefinierte Schnittstellen übertragen).

Auch die konsequente Anwendung des Prinzips, dass jedes Datum einem Modul gehört und nur von diesem verändert werden darf, reduziert die Kopplung. Bei Bedarf stellt der Eigentümer eines Datums Operationen zur kontrollierten Manipulation des Datums durch Dritte bereit.

6.3.3 Das Geheimnisprinzip (Information Hiding): Das Fundamentalprinzip zur Gliederung komplexer Systeme

Das von Parnas (1972) entdeckte Geheimnisprinzip (information hiding) ist das Gliederungskriterium, welches gute Modularisierungen mit den oben genannten Eigenschaften liefert.

DEFINITION 6.7. *Geheimnisprinzip (information hiding).* Kriterium zur Gliederung eines Gebildes in Komponenten, so dass jede Komponente eine Leistung (oder eine Gruppe logisch eng zusammenhängender Leistungen) vollständig erbringt und zwar so, dass außerhalb der Komponente nur bekannt ist, *was* die Komponente leistet. *Wie* sie ihre Leistungen erbringt, wird nach außen *verborgen*.

Parnas hat das Geheimnisprinzip ursprünglich definiert als das Einkapseln von Entwurfsentscheidungen beim Modularisieren von Software mit dem Ziel, die *Realisierung* der Entwurfsentscheidungen vor den Modulverwendern zu *verbergen* und die Entwürfe dadurch leichter

änderbar zu machen. Die Definition 6.7 wurde bewusst allgemeiner gefasst, um deutlich zu machen, dass es sich hier um ein weit über den Software-Entwurf und die Informatik hinausreichendes, *allgemeines und fundamentales Abstraktionsprinzip zur Beherrschung komplexer Systeme* handelt.

Wir benutzen das Geheimnisprinzip tagtäglich zur Beherrschung der Komplexität unserer Umwelt; etwa indem wir Auto fahren, ohne wissen zu müssen, wie ein Verbrennungsmotor funktioniert oder ein elektrisches Gerät in Betrieb nehmen, ohne wissen zu müssen, wie der Strom in die Steckdose kommt.

Eine Bank ist aus der Sicht der Kunden ebenfalls eine klassisches nach dem Geheimnisprinzip arbeitendes System: Sie erbringt für ihre Kunden über eine Schnittstelle (Schalter, Bancomat) Leistungen (Geld entgegennehmen und sicher aufbewahren, Guthaben wieder auszahlen,...). Dabei wissen die Kunden nur, *was* die Bank leistet. *Wie* die Leistungen erbracht werden, zum Beispiel wie die Bank die Kontenführung organisiert oder wie sie die Kundeneinlagen so anlegt, dass sie verlangte Auszahlungen jederzeit leisten kann, bleibt nach außen verborgen. Das Geheimnisprinzip ist für beide Seiten wichtig: Die Kunden wollen ihr Konto benutzen, ohne wissen zu müssen, wie eine Bank funktioniert und ohne von bankinternen Organisationsänderungen betroffen zu sein. Die Bank will ihr Geschäft unabhängig von den Kunden organisieren, und sie will insbesondere Änderungen in der Realisierung ihrer Leistungen vornehmen können, ohne dies mit allen Kunden abstimmen zu müssen.

Im Software-Entwurf sind die Komponenten Module, die Leistungen Teillösungen. Das WAS ist in der Modulschnittstelle beschrieben, das WIE sind die Entwurfsentscheidungen und ihre Realisierung durch Programme. Bei Software gibt es vier typische Arten von Entwurfsentscheidungen:

- wie eine Funktion realisiert ist, zum Beispiel die Kalkulation einer Versicherungsprämie
- wie ein Objekt aus dem Anwendungsbereich repräsentiert und realisiert ist, zum Beispiel ein Konto
- wie eine Datenstruktur aufgebaut ist und wie sie bearbeitet werden kann, zum Beispiel die Datenhaltung und die Suchfunktion für die Bereitstellung von «Gelben Seiten» im WWW
- wie Leistungen Dritter realisiert sind, zum Beispiel wie ein Datenbanksystem intern funktioniert.

6.3.4 Schnittstellen und Verträge

Schnittstellen (interfaces) beschreiben das Leistungsangebot eines Moduls. In Anwendung des Geheimnisprinzips soll die Beschreibung so weit wie möglich realisierungsneutral sein. In der Regel werden Leistungen in Form von Operationen und Datentypen bereitgestellt. Die Bereitstellung direkt zugreifbarer Daten wird nach Möglichkeit vermieden, da sie die Offenlegung der Datenrepräsentation erfordert. Voraussetzungen (preconditions) und Ergebniszusicherungen (assertions, postconditions) gestatten eine realisierungsneutrale Beschreibung der Operationen:

- Die *Voraussetzungen* geben an, welche Bedingungen erfüllt sein müssen, bevor die Operation aktiviert werden darf. Die Operation prüft die von ihr gemachten Voraussetzungen in der Regel nicht. Bei verletzten Voraussetzungen ist das Ergebnis der Operation undefiniert.
- Die *Ergebniszusicherungen* beschreiben die Ergebnisse der Operation (gelieferte Daten, Änderungen im Systemzustand)

Weitere Aussagen, sei es für eine Operation oder für den ganzen Modul, können mit Invarianten (invariants) und Verpflichtungen (obligations) gemacht werden:

- *Invarianten* sind Zusicherungen über Systemeigenschaften, die durch die Operation nicht verändert werden
- *Verpflichtungen* überbinden dem Verwender einer Operation Pflichten, die er erfüllen muss, um übergeordnete Systeminvarianten nicht zu verletzen.

Die Bedingungen und Zusicherungen beschreiben einen *Vertrag* zwischen dem Modul als Leistungserbringer und den Klienten, welche die Leistungen durch Verwendung von Operationen des Moduls nutzen: Unter der Bedingung, dass der Klient garantiert, dass die Voraussetzungen einer Operation des Moduls zum Zeitpunkt der Aktivierung erfüllt sind, garantiert der Modul, dass die Ergebniszusicherungen der Operation nach ihrem Abschluss erfüllt sind. Die konsequente Dokumentation aller Modulschnittstellen durch solche Verträge trägt erheblich zur Robustheit und Änderbarkeit der Software bei. Meyer (1988/1997 und 1992) hat dafür den Ausdruck «Design by Contract» geprägt.

6.3.5 Nebenläufigkeit

Dort wo es möglich ist, werden Informatikprobleme durch *sequentielle* Programme realisiert. Unter sequentiellen Programmen verstehen wir solche, die ein Problem durch eine sequentielle Folge von Programmschritten lösen. Zu jedem Zeitpunkt ist folglich höchstens ein Programmschritt in Bearbeitung. Die Bevorzugung sequentieller Programme hat vor allem zwei Gründe: Erstens sind sequentielle Programme auf klassischen Rechnern leicht realisierbar und zweitens sind sie leichter zu verstehen und zu entwerfen als Programme, in denen mehrere Dinge gleichzeitig parallel ablaufen.

Viele Probleme erfordern jedoch die gleichzeitige, koordinierte Bearbeitung mehrerer Aufgaben, sei es lokal an einem Ort oder geographisch verteilt auf mehrere Orte. Solche Probleme sind nicht oder nur sehr schlecht mit sequentiellen Programmen lösbar. Stattdessen wird die Lösung in diesem Fall auf mehrere *nebenläufige Prozesse* aufgeteilt.

DEFINITION 6.8. *Prozess (process).* Eine durch ein Programm gegebene Folge von Aktionen, die sich in Bearbeitung befindet.

Ein Prozess entsteht also zu dem Zeitpunkt, wo seine erste Aktion begonnen wird und er verschwindet nach Abschluss seiner letzten Aktion. In dieser Zeit ist er entweder aktiv (d.h. er arbeitet nach dem gegebenen Programm) oder er wartet auf benötigte Ressourcen oder er wartet auf einen anderen Prozess (zum Beispiel um mit ihm zu kommunizieren).

DEFINITION 6.9. *Nebenläufigkeit (concurrency).* Die parallele oder zeitlich verzahnte Bearbeitung mehrerer Aufgaben.

Während ihrer Arbeit müssen Prozesse in der Regel miteinander *kommunizieren*, d.h. sie müssen *Informationen austauschen* oder ihren Arbeitsfortschritt *synchronisieren*. Häufig wird auch beides miteinander kombiniert.

Die zur Synchronisation notwendigen Konzepte und Konstrukte werden vom Betriebssystem oder von einer Laufzeitumgebung für die verwendete Programmiersprache bereitgestellt. Interessierte Leserinnen und Leser seien auf die Literatur über Betriebssysteme verwiesen.

Informationsaustausch erfolgt entweder über das Senden und Empfangen von Nachrichten oder über Speicherbereiche, die beiden kommunizierenden Prozessen zugänglich sind. Nachrichtenaustausch ist etwas langsamer, dafür aber sicher und geographisch verteilbar. Kommunikation über gemeinsame Speicher ist schnell, muss dafür aber explizit gesichert werden und ist in der Regel nicht verteilbar.

Beispiel

Ein Informatiksystem soll Devisenhändler bei ihrer Arbeit unterstützen. Zu diesem Zweck sollen auf einer Arbeitsplatzstation folgende Komponenten verfügbar sein:

- ein Editor zur Bearbeitung und Anzeige der laufenden Kauf- und Verkaufsaufträge,

- ein Archivierer, der alle abgewickelten Transaktionen registriert und bei Bedarf anzeigt,
- ein Telefonbutler, der ein Telefonverzeichnis anzeigt, bei Anwahl eines Eintrags die zugehörige Person anruft sowie mehrere Linien gleichzeitig hält und makelt,
- ein Kursbeobachter, der die aktuellen Kurse und den Kursverlauf der gehandelten Währungen fortlaufend ermittelt und anzeigt.

Alle vier Komponenten sollen gleichzeitig arbeiten können und werden daher mit nebenläufigen Prozessen realisiert. Für gemeinsame Infrastrukturaufgaben (zum Beispiel Datenhaltung) werden weitere Prozesse definiert.

6.3.6 Berücksichtigung der Ressourcen

Eine (zumindest vorläufige) *Zuordnung von Ressourcen* zu den Komponenten der Software-Architektur ist bereits in der Konzipierung notwendig, weil sonst die technische Machbarkeit des Lösungskonzepts und die Erfüllbarkeit der gestellten Anforderungen (insbesondere der Leistungsanforderungen) nicht überprüft werden können. Ressourcen können sowohl logischer Art (z.B. Prozesse) als auch physischer Art (z.B. Prozessoren) sein. Folgende Zuordnungen müssen vorgenommen werden:

- Module zu Prozessen
- Prozesse zu Prozessoren
- Daten zu Speichern bzw. Medien
- Prozesskommunikation zu Kommunikationstechnologien und -medien.

Die wesentlichen Kriterien dabei sind (in dieser Reihenfolge): Erfüllung der Anforderungen, Klarheit der Struktur und Ressourceneffizienz. Angestrebt wird eine möglichst hohe Übereinstimmung zwischen der logischen und der physischen Struktur des Systems. Solange die Leistungsanforderungen erfüllt sind, werden dafür auch Ineffizienzen, z.B. ungleichmäßig ausgelastete Prozessoren, in Kauf genommen. Eine test- und änderungsfreundliche Software-Struktur ist besser und wirtschaftlicher als eine chaotische Struktur mit optimal ausgelasteten Ressourcen.

6.3.7 Aspektorientierung

Während man insgesamt versucht, die Lösung in voneinander möglichst unabhängige Teillösungen zu gliedern, gibt es Querschnittsaufgaben, die nur in einer Gesamtsicht befriedigend konzipiert werden können. Dabei wird die Darstellung auf den betreffenden Querschnittsaspekt fokussiert. Ein Gestaltungskonzept für die Benutzerschnittstelle zum Beispiel muss systemweit gelten. Es wäre schlecht, in verschiedenen Modulen unterschiedliche Benutzerschnittstellen zu haben. Für die folgenden Querschnittsaufgaben sollten aspektorientierte Teilkonzepte erstellt werden, sofern dieser Aspekt für das betreffende System relevant ist: Die Datenhaltung, insbesondere das konzeptionelle Datenbankschema bei Verwendung einer Datenbank, die Gestaltung der Benutzerschnittstelle, die Behandlung von Fehlern sowie die Gewährleistung von Sicherheit und Fehlertoleranz.

6.3.8 Nutzung von Vorhandenem

Wo immer möglich, werden Systeme nicht vollständig neu entwickelt, sondern es werden Teile oder gar das ganze System *beschafft* bzw. *wiederverwendet*. In der Konzipierung müssen die wesentlichen Beschaffungs- und Wiederverwendungsentscheidungen getroffen werden. Es ist daher eine zentrale Aufgabe der Konzipierung, zu untersuchen, welche Teile der Lösung mit welchen existierenden Komponenten realisiert werden können. Folgende Punkte sind immer zu untersuchen:

- Kann eine Lösung vollständig beschafft werden (Standardsoftware, konfigurierbare Bausteine)?
- Können abgeschlossene Teilsysteme (zum Beispiel ein Datenbanksystem) beschafft werden?
- Ist das System durch Einbettung in einen existierenden Software-Rahmen (framework) realisierbar?
- Können einzelne Komponenten (aus Programm- oder Klassenbibliotheken) genutzt werden?
- Können Architektur- und Entwurfsideen wiederverwendet werden (zum Beispiel durch Verwendung vorhandener Architekturmetaphern und -muster oder von Entwurfsmustern (design patterns, Gamma et al. 1995)?
- Kann/soll das Lösungskonzept so modifiziert werden, dass vorhandene Software wiederverwendet bzw. beschafft werden kann?

Für Beschaffungen und Wiederverwendung gelten die gleichen Abwägungen und Kostenüberlegungen wie für Entwurfsvarianten (vgl. Abschnitt 6.5.4). Bei den Kostenüberlegungen sind zwei spezifische Punkte zu beachten:
- Wie und in welchem Zeit- und Kostenrahmen können Kandidaten gesucht und evaluiert werden?
- Welche Parametrierungen, Konfigurierungen, Anpassungen und Ergänzungen sind notwendig und wieviel kosten diese?

6.3.9 Ästhetik

Wie in der Architektur von Gebäuden gibt es auch in der Software-Architektur eine Ästhetik. Diese äußert sich vor allem in
- der Wahl und konsequenten Verwendung eines Architekturstils (vgl. Abschnitt 6.6)
- klar erkennbaren, gestalteten Strukturen (dies im Gegensatz zu irgendwie Gewordenem oder Gewursteltem)
- einer der Struktur des Problems angemessenen Struktur der Architektur
- der Wahl der einfachsten und klarsten Lösung aus der Menge der möglichen Lösungen.

6.3.10 Qualität

Ein guter Entwurf ist:
- *effektiv*, das heißt er erfüllt die Vorgaben und löst das Problem des Auftraggebers,
- *wirtschaftlich*, das heißt gebrauchstauglich, kostengünstig und mehrfachverwendbar bzw. mehrfachverwendet,
- *softwaretechnisch gut*, das heißt leicht verständlich, robust, zuverlässig und änderungsfreundlich.

Die Sicherstellung dieser Qualitäten erfordert kontinuierliche Prüfmaßnahmen im Entwurfsprozess (vgl. Abschnitt 6.5.3, Prüfung).

6.4 Produkte

Das Ergebnis des Architekturentwurfs wird in einem Dokument niedergelegt, das wir *Lösungskonzept* oder auch *Software-Architektur* nennen. Das Lösungskonzept dokumentiert die Gliederung der Software in Prozesse, die Modularisierung, die Interaktionen zwischen den Prozessen und Modulen, die Zuordnung von Ressourcen sowie aspektbezogene Teilkonzepte für Querschnittsaufgaben wie Datenbankschema, Konzept der Mensch-Maschine-Kommunikation,

Fehlertoleranzkonzept, etc. Das Lösungskonzept dokumentiert auch die Einbettung in vorhandene Software sowie Beschaffungs- und Wiederverwendungsentscheide.

MUSTER 6.1. Möglicher Aufbau eines Lösungskonzepts

1. Einleitung
- **1.1 Überblick**
  Überblick über die gewählte Lösung
- **1.2 Ziele und Vorgaben**
  Beschreibung von Entwurfszielen und Vorgaben, die nicht in der Anforderungsspezifikation stehen
- **1.3 Einbettung und Abgrenzung**
  - Wo und wie ist das konzipierte System eingebettet
  - Wie und über welche Schnittstellen wird mit der Umwelt kommuniziert
- **1.4 Lösungsalternativen**
  Betrachtete, aber schließlich verworfene Lösungsalternativen: Kurze Skizze jeder Alternative, Grund für die Verwerfung

2. Struktur der Lösung
- **2.1 Übersicht**
  - Architekturstil, Metapher(n) und Architekturmuster, die der Architektur zugrunde liegen
  - Teilsysteme und ihre Aufgaben
- **2.2 Prozessstruktur**
  Prozesse und Kommunikation zwischen den Prozessen
- **2.3 Modulare Struktur**
  Module und ihre Zusammenhänge, bei objektorientiertem Entwurf Klassen- bzw. Objektmodelle
- **2.4 Entwurf der Module**
  - Beschreibung der Schnittstellen
  - ggf. Hinweise zur geplanten Implementierung
- **2.5 Physische Struktur**
  - Physische Gliederung der Software in Pakete, Komponenten, etc.
  - Ressourcenzuordnung

3. Aspektbezogene Teilkonzepte
Ein Unterkapitel je interessierendem Aspekt, zum Beispiel Datenhaltungskonzept, Mensch-Maschine-Kommunikationskonzept, Fehlerbehandlungskonzept, Fehlertoleranzkonzept, Sicherheitskonzept, etc.

4. Voraussetzungen und benötigte Hilfsmittel
- **4.1 Benötigte Software**
  Beschreibung der benötigten (fertigen) Software, welche für Entwicklung und/oder Betrieb des Systems zu beschaffen bzw. zu verwenden ist
- **4.2 Benötigte Hardware**
  Beschreibung der benötigten Hardware, welche für Entwicklung und/oder Betrieb des Systems zu beschaffen bzw. zu verwenden ist
- **4.3 Benötigtes Umfeld**
  - Charakterisierung der für den Betrieb des Systems erforderlichen organisatorischen und /oder technischen Strukturen und Abläufe

Quellennachweis

6.5 Der Entwurfsprozess

6.5.1 Einbettung

Die Erstellung des Architekturentwurfs ist der erste Schritt der Lösung eines Systems. Als Vorgabe muss eine Anforderungsspezifikation existieren. Je nach gewähltem Prozessmodell für den (gesamten) Entwicklungsprozess wird die Anforderungsspezifikation vollständig vor dem

Lösungskonzept erstellt oder die Erstellung erfolgt verzahnt, aber mit separater Dokumentation von Anforderungen einerseits und Lösung andererseits. Es sind sowohl hierarchische als auch zeitliche Verzahnung möglich. Bei der *hierarchischen Verzahnung* werden aus Globalanforderungen erste, grundsätzliche Lösungsentscheide abgeleitet. Diese induzieren Anforderungen an die Komponenten dieser Globallösung. Aus diesen Anforderungen werden wieder Lösungskonzepte abgeleitet, die wiederum zu Anforderungen an Teilsysteme dieser Lösungen führen können, usw. *Zeitliche Verzahnung* entsteht bei Verwendung von Wachstumsmodellen.

Das Lösungskonzept wiederum ist die Vorgabe für die Realisierung der Software, (d.h. den Detailentwurf, die Codierung und die Integration). Auch hier gibt es vor allem bei Wachstumsmodellen eine zeitliche Verzahnung von Architekturentwurf und Realisierung.

Der Architekturentwurf grenzt sich vom Detailentwurf ab, indem im Architekturentwurf nur die Komponenten, ihre Schnittstellen und ihre Interaktionen festgelegt werden, nicht aber die interne Realisierung der Komponenten durch Algorithmen und (verborgene) Datenstrukturen. Letzteres ist Aufgabe des Detailentwurfs.

6.5.2 Die Hauptaufgaben des Architekturentwurfs

Tabelle 6.1 fasst die Hauptaufgaben zusammen. Bild 6.1 zeigt ihre Zusammenhänge und Abhängigkeiten. Nachfolgend werden die einzelnen Aufgaben etwas näher charakterisiert. Das konkrete Vorgehen bei der Bearbeitung der Aufgaben ist stark von den verwendeten Entwurfstechniken und vom gewählten Prozessmodell abhängig.

TABELLE 6.1. Aufgaben des Architekturentwurfs

- **Aufgabe analysieren**
  - Anforderungen verstehen
  - Vorhandene bzw. beschaffbare Technologien und Mittel analysieren
- **Architektur modellieren und dokumentieren**
  - Grundlegende Systemarchitektur festlegen: Wahl geeigneter Architekturmuster und -metaphern, Festlegung des Architekturstils
  - Modularisieren
  - Nebenläufige Lösungen in Prozesse gliedern
  - Wiederverwendungs- und Beschaffungsentscheide treffen
  - Ressourcen zuordnen
  - Aspektbezogene Teilkonzepte für Querschnittsaufgaben erstellen
  - Lösungskonzept (als Dokument) erstellen
- **Lösungskonzept prüfen**
  - Anforderungen erfüllt?
  - Softwaretechnisch gut?
  - Wirtschaftlich?

6.5.3 Vorgehen beim Konzipieren

6.5.3.1 Aufgabenanalyse

Das Vorgehen bei der Aufgabenanalyse hängt sehr stark von der Art der vorhandenen Vorgaben ab. Teilweise können die gleichen Techniken wie bei der Gewinnung von Anforderungen (vgl. Kapitel 7) verwendet werden. Es geht darum, ein klares Verständnis zu erarbeiten, was verlangt wird und welche grundsätzlichen Lösungsmöglichkeiten in Frage kommen.

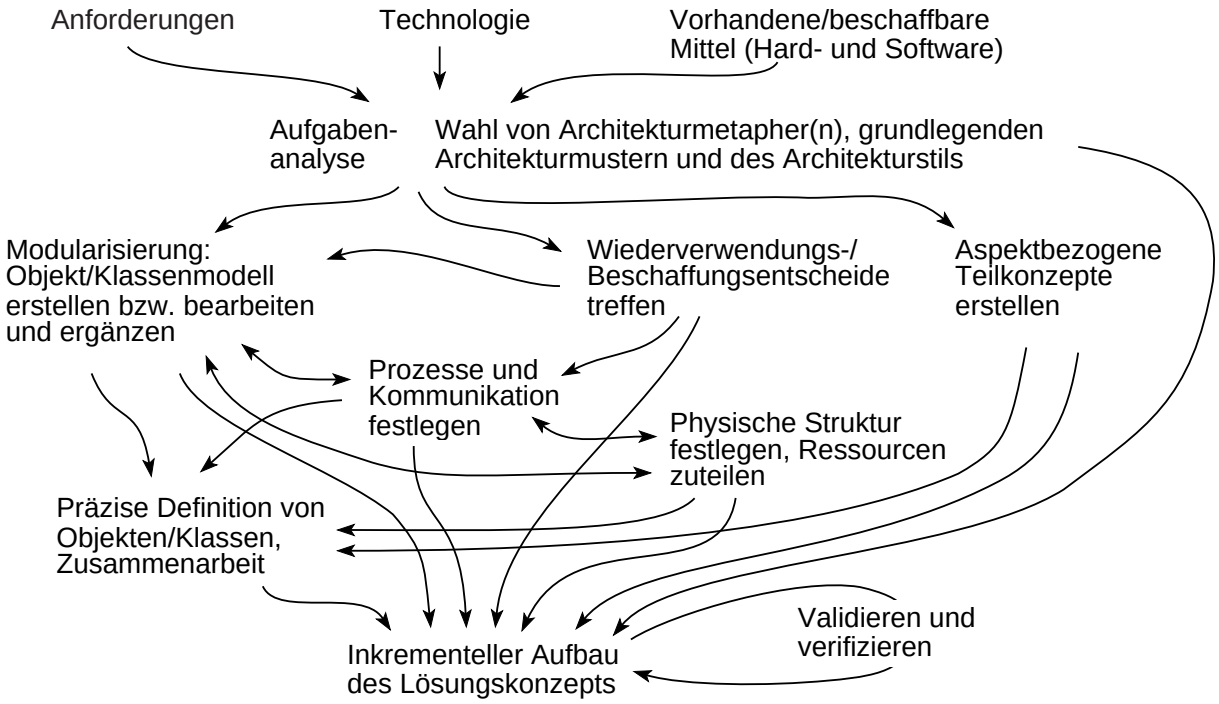


BILD 6.1. Zusammenhänge zwischen den Aufgaben des Architekturentwurfs (am Beispiel eines objektorientierten Entwurfs)

6.5.3.2 Modellierung und Dokumentation

Die Auswahl von Architekturmustern und Architekturmetaphern richtet sich nach dem gegebenen Problem. *Architekturmuster* sind vorgefertigte, parametrierbare Schablonen für die Gestaltung der Architektur eines Systems oder einer Komponente (vgl. Abschnitt 6.7). *Architekturmetaphern* sind Leitbilder für die Gestaltung von Architekturen. Durch Analogien zwischen der Metapher und der Architektur erleichtern sie das Verständnis der letzteren (vgl. Abschnitt 6.8).

Die Festlegung, mit welcher Art von Modulen und welcher Art von Modul-Kooperation gearbeitet werden soll, konstituiert zusammen mit der Auswahl passender Architekturmuster und/oder der Orientierung an einer Architekturmetapher den *Architekturstil* (vgl. Abschnitt 6.6). Dieser gibt der Architektur ein Gesicht und erleichtert das Verständnis. Der Stil wird so gewählt, dass die resultierende Architektur eine möglichst problemadäquate Struktur aufweist.

Die Art und die Technik der *Modularisierung* richten sich nach dem gewählten Architekturstil. Anzustreben sind Module mit hoher Kohäsion und geringer wechselseitiger Kopplung. Aus heutiger Sicht ist das Geheimnisprinzip ein zentrales Modularisierungskriterium, wenn der resultierende Entwurf leicht verständlich und gut änderbar bzw. erweiterbar sein soll.

Bei der Konzipierung *nebenläufiger* Systeme müssen die *Prozesse bestimmt* und die *Module den Prozessen zugeordnet* werden. Dies kann wie folgt geschehen:

- Man bestimmt alle systemexternen Akteure, welche voneinander unabhängig und spontan (d.h. nicht aufgrund eines vorherigen Anstoßes durch das System) Informationen an das System senden und eine Systemreaktion erwarten.
- Für jeden dieser Akteure wird ein Prozess definiert. Alle Module, welche zur Erzeugung der geforderten Reaktion notwendig sind, werden dem Prozess zugeordnet.
- Dabei stellt sich meistens heraus, dass eine Reihe von Modulen in mehreren Prozessen benötigt wird. In dieser Situation werden zusätzliche Dienstleistungsprozesse definiert und die gemeinsam benötigten Module diesen Prozessen zugewiesen.

- Ist die Software auf mehrere Rechner verteilt, versucht man die Prozesse als Ganzes zu verteilen. Manchmal muss aber auch ein logischer Prozess auf mehrere physische Prozesse aufgeteilt werden.
- Weitere Prozesse können notwendig sein, wenn Aufgaben unterschiedlicher Dauer und Dringlichkeit zu erfüllen sind, wenn die Aufgaben, die ein Prozess zu erfüllen hat, zu umfangreich werden oder wenn Fehlertoleranzanforderungen zu erfüllen sind.

Die Kommunikationsbedürfnisse ergeben sich aus der Verteilung der Module auf die Prozesse.

Das Vorgehen bei *Wiederverwendung und Beschaffung* wird in Abschnitt 6.5.5 beschrieben.

Bei der *Ressourcenzuordnung* werden die Module in der Regel als Ganzes auf Prozesse verteilt. Je enger die Kopplung zwischen einer Gruppe von Modulen ist, desto eher sollten sie dem selben Prozess und dieser wiederum dem selben Rechner im System zugeordnet werden. Die Aufspaltung eines Moduls auf mehrere Prozesse oder gar mehrere Rechner erhöht den Kopplungsgrad des Systems und erschwert Fehlersuche und Modifikationen. Zu erfüllende Leistungsanforderungen (v.a. geforderte Reaktionszeiten und zu transportierende Datenvolumina) können aber zu Abweichungen von diesem Prinzip zwingen.

Die gleichen Überlegungen gelten für die Zuordnung von Prozessen und Daten. Bei der Verteilung auf mehrere Rechner ist immer der durch die Verteilung entstehende Kommunikationsbedarf, insbesondere die Menge der zu übertragenden Daten, zu berücksichtigen. Bei der Datenzuordnung muss dabei meistens zwischen redundanzfreier Speicherung mit erhöhtem Datenübertragungsbedarf einerseits und redundanter Speicherung mit geringem Übertragungsbedarf abgewogen werden.

Bei der Zuordnung der Prozesskommunikation muss vor allem darauf geachtet werden, dass die Leistungsanforderungen unter Nutzung der vorhandenen Technologie erreicht werden.

Dort, wo physische Randbedingungen oder Leistungsanforderungen dazu zwingen, eine von der logischen Struktur der Lösung abweichende physische Struktur zu wählen, muss neben der logischen auch die physische Struktur dokumentiert werden. Gleiches gilt, wenn die Software zu Liefer- oder Verwaltungszwecken in Pakete gegliedert wird. Das Lösungskonzept enthält dann verschiedene Sichten auf die Lösung (Kruchten 1995).

Die Erstellung *aspektbezogener Teilkonzepte* richtet sich nach den Regeln für den Entwurf des jeweiligen Aspekts, zum Beispiel den Regeln des Datenbankentwurfs für die Erstellung des Datenhaltungskonzepts. Auf eine nähere Ausführung wird hier verzichtet, da sie den Rahmen dieses Skripts sprengen würde.

Die Struktur des *Lösungskonzepts* (als Dokument) ist in Abschnitt 6.4. beschrieben. Wichtig ist, dass das Dokument nicht erst am Schluss der Entwurfstätigkeiten entsteht, sondern inkrementell während der gesamten Entwurfsarbeit.

6.5.3.3 Prüfung

Es ist zu *verifizieren*, dass das Konzept die in der Anforderungsspezifikation erfüllten Anforderungen erfüllt, dass es wirtschaftlich ist und softwaretechnisch gut. Kriterien für die softwaretechnische Güte sind zum Beispiel die Güte der Modularisierung (hohe Kohäsion, geringe Kopplung), die durchgängige Einhaltung des gewählten Entwurfsstils sowie Umfang und Verständlichkeit der Dokumentation.

Es ist sinnvoll, das Konzept zusätzlich auch zu *validieren*, d.h. unter Beteiligung der zukünftigen Benutzer festzustellen, ob es ihren Wünschen und Vorstellungen entspricht. Da das Konzept viel konkreter ist als die Anforderungsspezifikation, werden so möglicherweise noch Anforderungsfehler erkannt, bevor mit der Realisierung des Systems begonnen wird.

Das Mittel der Wahl zur Prüfung eines Lösungskonzepts ist das Review (vgl. Kapitel 9). Zur Prüfung kritischer Leistungsanforderungen sind evtl. Simulationen oder Labormuster (eine Prototypart, vgl. Kapitel 4) erforderlich. Auch Prototypen im engeren Sinn können zur Validierung herangezogen werden, zum Beispiel, um die Brauchbarkeit eines Benutzerschnittstellenkonzepts zu überprüfen.

Informelle Prüfungen finden kontinuierlich während der gesamten Konzipierungsarbeit statt. Vor Abschluss der Arbeit wird ein formelles Review durchgeführt und das Konzept entsprechend den erhobenen Befunden revidiert. Bei größeren Aufgaben werden Teillösungen schon vorab formellen Reviews unterzogen.

6.5.4 Variantenbehandlung

Es ist wesentlich, nicht stur einen einzigen Lösungspfad zu verfolgen, sondern stets den ganzen Raum möglicher Lösungen zu betrachten. Überall, wo sich mehrere Lösungsmöglichkeiten ergeben, werden die möglichen Lösungsvarianten soweit verfolgt, bis ein sinnvoller Entscheid für die beste Variante möglich ist. Das Lösungskonzept, welches das Resultat der Konzipierung dokumentiert, enthält keine Varianten mehr. Es dokumentiert lediglich wichtige verworfene Varianten kurz, um die getroffenen Entscheidungen nachvollziehbar zu machen.

Bei den Kosten dürfen nicht nur die unmittelbaren Entwicklungskosten einer Variante betrachtet werden. Vielmehr müssen auch die Auswirkungen auf die Betriebs- und Pflegekosten sowie mögliche Folgekosten durch Einflüsse auf andere Komponenten untersucht werden.

Für die Untersuchung von Lösungsvarianten darf jedoch nicht beliebig viel Aufwand getrieben werden. Die Kosteneinsparungen durch die Wahl einer besseren bzw. der besten Variante müssen stets im richtigen Verhältnis zu den Kosten für die Untersuchung der Varianten stehen. Als Faustregel gilt: Je größer bzw. teuerer der Untersuchungsgegenstand ist, desto aufwendiger darf die Untersuchung sein.

6.5.5 Vorgehen bei Beschaffung und Wiederverwendung

Die nachfolgend aufgeführten fünf Schritte mögen als grober Leitfaden für das Vorgehen bei *Beschaffungen* dienen. Der Beschaffungsprozess läuft parallel zur Spezifikation der Anforderungen und zur Konzipierung der Lösung. Der Aufwand, den man für die Beschaffung einer Komponente (bzw. eines Systems) treibt, hängt stark davon ab

- wie wichtig die Anwendung ist, für welche die Komponente beschafft wird
- wie teuer die beschaffte Komponente ist
- wie lange man die Komponente nutzen will.

Die Schritte 1 bis 3 sollten parallel zur Spezifikation der Anforderungen erfolgen. Die Schritte 4 und 5 (und evtl. ein Teil von 3) sind Teilaufgaben der Konzipierung.

1. **Anforderungen analysieren:** Wer seriös beschaffen will, muss als erstes die zu erfüllenden Anforderungen kennen.
2. **Hauptkriterien definieren:** Aus der Menge aller Anforderungen werden diejenigen herausgeschält, die kritisch und unverzichtbar sind (Faustregel: weniger als zehn, zum Beispiel drei bis fünf funktionale Anforderungen, zwei Leistungsanforderungen, der Maximalpreis und (bei reinen Software-Beschaffungen) die Hardware, auf der das System laufen muss).
3. **Marktübersicht verschaffen, Grobauswahl treffen:** Man muss herausfinden, was es für den gewünschten Problembereich überhaupt gibt. Quellen sind zum Beispiel Anzeigen,

Recherchen im WWW, Berichte und Tests in Zeitschriften, Händler sowie Kontakte zu anderen Unternehmen der gleichen Branche. Anhand der Kriterienliste wird eine Grobauswahl getroffen, bei der maximal noch drei bis fünf Kandidaten übrigbleiben sollten.

4. **Kandidatensysteme evaluieren:**
   - Über die in die engere Wahl gekommenen Komponenten bzw. Systeme beschafft man sich genaue Unterlagen und lässt sie sich ggf. vorführen.
   - Aussagen zu kritischen Punkten, die aus Werbematerial oder von Verkäufern stammen, sollten nachgeprüft werden.
   - Bei größeren und teureren Komponenten bzw. Systemen ist eine formale Evaluation sinnvoll. Dabei werden alle Anforderungen gewichtet, für jede Anforderung und jedes Kandidatensystem die Erfüllung geprüft und die Ergebnisse zusammengezählt. Der Kandidat mit den meisten Punkten erfüllt den Anforderungskatalog am besten.
   - Neben der formalen Bewertung sollten auch folgende Aspekte zur Beurteilung herangezogen werden
     - Vertrauenswürdigkeit des Herstellers/Händlers
     - Verfügbarkeit von Service
     - Erfahrungen mit anderen Produkten des gleichen Herstellers
     - Eigene Beschaffungspolitik (z.B. möglichst alles aus einer Hand beschaffen)
   - Für jeden Kandidaten ist außerdem zu prüfen, welcher (finanzielle und zeitliche) Aufwand für notwendige Parametrierungen oder Anpassungsentwicklungen erforderlich ist.
   - Sind Komponenten bzw. Systeme in einer Preisklasse über Fr. 10.000 zu beschaffen, empfiehlt sich die Miete einer Probeinstallation, welche eine intensive Erprobung unter realen Bedingungen ermöglicht. Dabei werden manchmal auch noch Probleme entdeckt, die bei der Formulierung der Anforderungen vergessen wurden.

5. **Entscheidung fällen und dokumentieren:** Aufgrund der Evaluation und evtl. Erprobung wird die Beschaffungsentscheidung gefällt. Die Gründe sollten kurz schriftlich festgehalten werden, damit die Entscheidung später noch nachvollziehbar ist.
   Erfüllt keiner der evaluierten Kandidaten die gestellten Anforderungen, so ist als erstes zu prüfen, ob der Anforderungskatalog zu streng ist und wo man ggf. Abstriche machen kann. Führt dieser Weg nicht zum Ziel, so ist eine Beschaffung nicht möglich. In diesem Fall muss eine Lösung selbst entwickelt werden.

Bei *Wiederverwendung* werden in der Regel nur Komponenten, keine ganzen Systeme betrachtet. Das Vorgehen ist grundsätzlich gleich. Im Schritt 3 tritt an die Stelle der Marktübersicht die unternehmensinterne Suche nach wiederverwendbaren Komponenten. Es ist zu beachten, dass auch bei wiederverwendeter Software meistens Anpassungs- und Parametrierungsarbeiten anfallen, die etwas kosten. Ferner ist für jede wiederverwendete Komponente die Frage der Pflegeverantwortung (Lieferant oder Verwender) und der dabei möglicherweise entstehenden Kosten für Pflegeverträge zu klären.

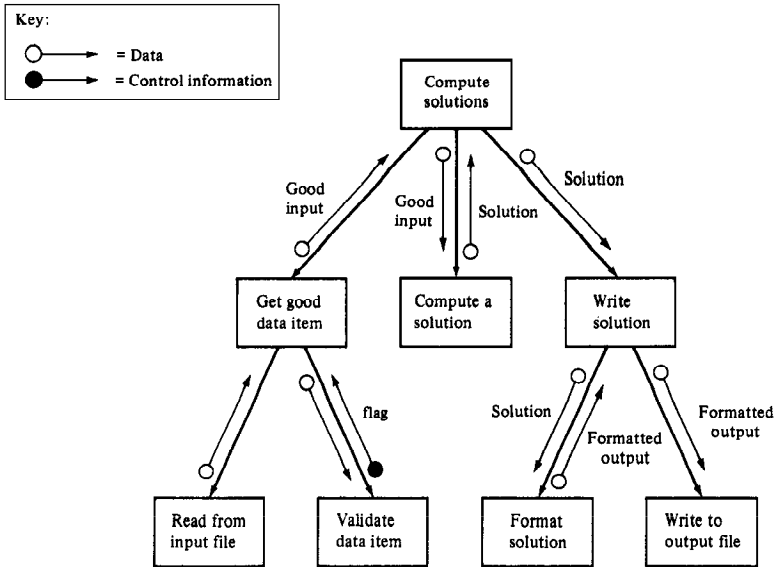
6.6 Ausgewählte Architekturstile

Eine Architektur ist im wesentlichen durch die Art der verwendeten Module, die Art(en) der Kooperation zwischen den Modulen und die Art der Modularisierung bestimmt. Sind diese drei Arten aufeinander abgestimmt, entsteht ein kohärenter *Architekturstil*. In den folgenden Abschnitten werden einige ausgewählte Architekturstile skizziert.

6.6.1 Funktionsorientierte Architektur (Structured Design)

Structured Design (Stevens, Myers, Constantine 1974; Page-Jones 1988) ist der klassische Vertreter einer *funktionsorientierten* Architektur. Die Grundidee ist, dass jeder Modul des Systems *eine Funktion* berechnet. Zur Realisierung einer Funktion können andere, einfachere Funktionen aufgerufen werden. Auf diese Art und Weise entsteht eine Hierarchie von Funktionen, wobei unten die elementaren und weiter oben die komplexeren Funktionen stehen. Die Funktion an der Spitze der Hierarchie ist das Hauptprogramm. Der Datenaustausch zwischen den Modulen erfolgt entweder bei Aufruf und Rückkehr durch Parameter oder durch Zugriff auf explizit modellierte gemeinsame Speicherbereiche. Bei den Parametern wird teilweise noch zwischen echten Daten und Steuerinformation unterschieden. Häufig werden die Funktionen zusätzlich in Eingabe-, Verarbeitungs- und Ausgabefunktionen gegliedert und die Hierarchie entsprechend strukturiert (Bild 6.2).

Eine funktionsorientierte Modularisierung kann den in Abschnitt 6.3.2 genannten Kriterien nur teilweise genügen. Alle Aufgaben, bei denen eine Gruppe von Funktionen einen gemeinsamen Status verwaltet (zum Beispiel Treiber, Datenstrukturen oder intern kooperierende Anwendungsfunktionen), lassen sich nicht geeignet behandeln. Solche Funktionsgruppen müssten nämlich nach den Regeln von Abschnitt 6.3.2 zusammen mit den Statusdaten zu einem Modul zusammengefasst werden, was jedoch mit Modulen, die nur *eine* Funktion enthalten (und damit nur einen möglichen Aufruf haben) nicht möglich ist. Diese ungenügenden Abstraktionsmöglichkeiten erschweren das Verständnis und die Änderbarkeit großer funktionsorientierter Entwürfe massiv.



Quelle: Fairley (1985)

BILD 6.2. Funktionsorientiert gegliedertes System

6.6.2 Datenorientierte Architektur (Entwurf mit Datenabstraktionen)

Im Zentrum der datenorientierten Architektur steht eine Modularisierung nach dem Geheimnisprinzip (vgl. 6.3.3). Wie bereits dort erwähnt, gibt es dabei vier typische Arten von Entwurfsentscheidungen, die zu Modulen führen:

(a) wie eine Funktion realisiert ist,
(b) wie ein Objekt aus dem Anwendungsbereich repräsentiert und realisiert ist,
(c) wie eine Datenstruktur aufgebaut ist und wie sie bearbeitet werden kann,
(d) wie Leistungen Dritter realisiert sind.

Die Fälle (b), (c) und (d) benötigen einen Abstraktionsmechanismus, welcher eine Datenstruktur und die auf den Daten dieser Struktur möglichen Operationen zu einer Einheit zusammenfasst. Eine solche Abstraktion wird *Datenabstraktion* genannt. *Abstrakte Datentypen* sind das bekannteste Mittel zum Aufbau von Datenabstraktionen. Der im Kapitel 7.5.4 spezifizierte Stack ist ein typisches Beispiel für einen abstrakten Datentyp.

Bei der datenorientierten Architektur besteht ein System aus einer Menge von Datenabstraktionen, die in ähnlicher Weise wie bei einer funktionsorientierten Architektur aufeinander aufbauen. Es entsteht so eine *Benutzungshierarchie*: Die Datenabstraktionen benutzen in ihren Implementierungen Operationen, die von tieferliegenden Datenabstraktionen angeboten werden. In der Regel versucht man, die Datenabstraktionen in Schichten zu gliedern (Anwendung der Virtuelle Maschinen-Metapher, vgl. 6.8.2) Bild 6.3 zeigt als Beispiel die Architektur der Software für einen Bancomaten. Man beachte, dass ein Pfeil in diesem Bild nicht wie in Bild 6.2 den Aufruf einer einzelnen Funktion, sondern die Benutzung von Leistungen bezeichnet. Die Zusammenarbeit zwischen Leistungsanbietern und Leistungsverwendern kann durch Verträge spezifiziert werden (Design by Contract).

Nach diesem Stil entworfene Systeme sind aufgrund der Eigenschaften des Geheimnisprinzips leicht änderbar und vor allem im Großen gut verstehbar. Ein datenorientierter Entwurf führt damit genau auf Modularisierungen, wie sie gemäß Abschnitt 6.3.2 erwünscht sind.

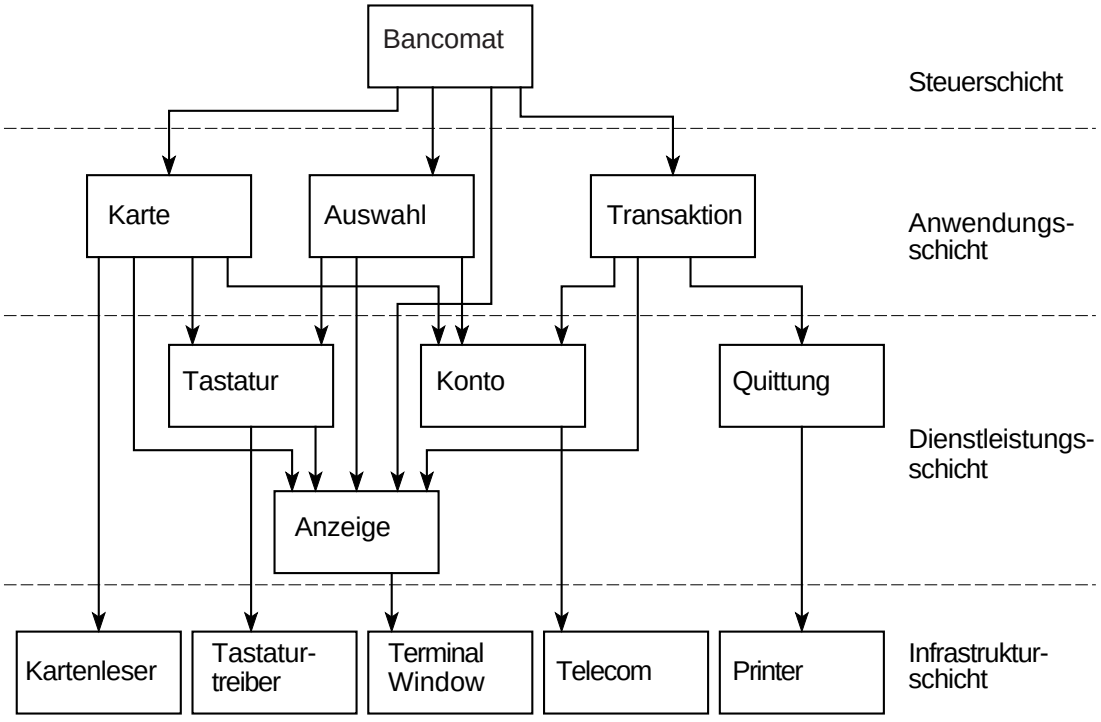


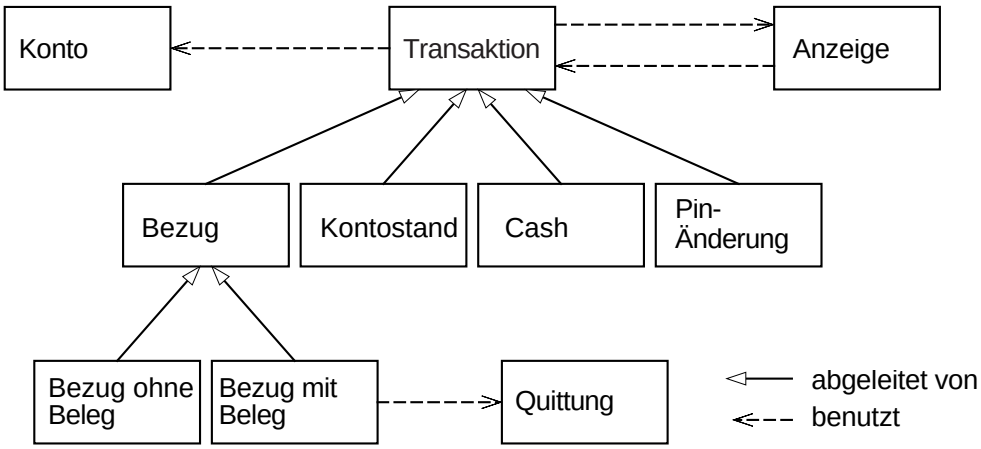
BILD 6.3. Benutzungshierarchie und Schichtung im Entwurf mit Datenabstraktionen

6.6.3 Objektorientierte Architektur

Im objektorientierten Entwurf wird ein System als eine Menge kooperierender Objekte aufgefasst. Wie eine Datenabstraktion besteht ein Objekt aus Daten und allen darauf möglichen Operationen. Objekte bzw. Klassen (d.h. Objekttypen) bilden die Einheiten der Modularisierung. Jedes Objekt repräsentiert ein Objekt aus dem Anwendungsbereich (zum Beispiel einen Kunden oder einen Artikel) oder ein in sich geschlossenes Informatik-Element, das für die Problemlösung benötigt wird (zum Beispiel ein Dialogfenster oder einen Schaltknopf in der Benutzerschnittstelle). Das Verbergen des Objektinneren nach dem Geheimnisprinzip ist teilweise

möglich und wird angestrebt. Das Prinzip der Vererbung (s.u.) lässt jedoch ein konsequentes Information Hiding nicht zu.

Die Objekte können auf zwei verschiedene Arten kooperieren. Erstens durch *Benutzung*: Ein Objekt benutzt in seiner Implementierung Operationen, die von anderen Objekten angeboten werden. Im Gegensatz zum Entwurf mit Datenabstraktionen wird keine Benutzungshierarchie angestrebt. Zweitens durch *Vererbung*: Von jeder Klasse können *Unterklassen* abgeleitet werden, welche alle Eigenschaften (Datenstrukturen und Operationen) von der Originalklasse übernehmen. Man sagt auch, eine Klasse *vererbt* ihre Merkmale an ihre Unterklassen. Die Möglichkeit der Vererbung macht die Entwürfe außerordentlich flexibel und eröffnet neue Möglichkeiten der Wiederverwendung. Der Entwurf und die Realisierung einer Unterklasse erfordern in der Regel jedoch eine zumindest teilweise Kenntnis der inneren Struktur der Klasse, von der abgeleitet wird. Information Hiding ist daher zwischen Ober- und Unterklassen nur begrenzt möglich.



Erläuterung: Modelliert ist ein Ausschnitt aus einem Bancomat-System. Die möglichen Transaktionen sind als Unterklassen von TRANSAKTION modelliert. Dies erlaubt die flexible Erweiterung der Software um weitere Transaktionsarten (zum Beispiel Überweisung tätigen). TRANSAKTION benutzt KONTO für Kontostandsabfragen und -mutationen. BEZUG MIT BELEG benutzt QUITTUNG zum Erstellen von Belegen. Die Interaktion zwischen TRANSAKTION und ANZEIGE ist nach dem Beobachtermuster (vgl. Bild 6.7) konstruiert: TRANSAKTION benachrichtigt ANZEIGE jedesmal, wenn innerhalb von TRANSAKTION oder von einer ihrer Unterklassen eine Veränderung stattgefunden hat, welche die Anzeige beeinflusst. ANZEIGE holt sich daraufhin die benötigte Information ab und zeigt sie an.

BILD 6.4. Benutzung und Vererbung in einer objektorientierten Architektur

Durch die Verwendung von zwei verschiedenen Kooperationsmechanismen (Benutzung und Vererbung) ist der objektorientierte Entwurf methodisch anspruchsvoller als traditionelle Entwurfsverfahren. Bei richtiger Anwendung ist das resultierende Lösungsmodell von hoher Qualität, indem es einerseits ein geradliniges Abbild des Modells der Aufgabenstellung darstellt und andererseits änderbar und erweiterbar ist.

Unsachgemäßer und undisziplinierter Gebrauch der Objektorientierung kann jedoch auch zu ausgesprochen schlechten Entwürfen führen. Labyrinthische Benutzungsgeflechte quer über alle Abstraktionsebenen und die Verwendung der Vererbung als Steinbruch (irgendwie Passendes wird durch Ableitung von Unterklassen übernommen, Unpassendes wird entfernt oder umdefiniert) führen zu nebenwirkungsreichen, schwer verstehbaren Systemen, deren Pflege extrem aufwendig werden kann.

6.6.4 Prozessorientierte Architektur

Nebenläufige Systeme bestehen aus einer Menge unabhängig arbeitender, untereinander kooperierender Akteure, die typisch als Prozesse realisiert sind. Die Prozesse kooperieren durch

Austausch von Nachrichten oder durch Zugriff auf gemeinsame Speicherbereiche. Die Prozesse können auf einem Rechner ablaufen oder auf verschiedene Rechner verteilt sein.

In prozessorientierten Architekturen sind die Prozesse die Module der obersten Stufe. Jeder Prozess ist typisch ein *sequentiell* ablaufender Systemteil, der seinerseits modularisiert ist, beispielsweise in objektorientiertem Stil.

Neben der Prozessstruktur und den Architekturen der einzelnen Prozesse muss auch eine Kommunikationsarchitektur entworfen werden. Heute erleichtert man sich diese Aufgabe in der Regel durch die Verwendung käuflicher Kommunikationssoftware, welche den Prozessen komfortable Kommunikationsschnittstellen zur Verfügung stellt. Die Prozesskommunikation wird so von einer Dienstleistungsschicht übernommen, deren Architektur und Realisierung vor den Verwendern verborgen bleibt.

Bild 6.5 zeigt als Beispiel einer prozessorientierten Architektur die Erfassung, Bereitstellung und Archivierung von Wechselkursdaten in einer Bank oder einem Handelshaus.

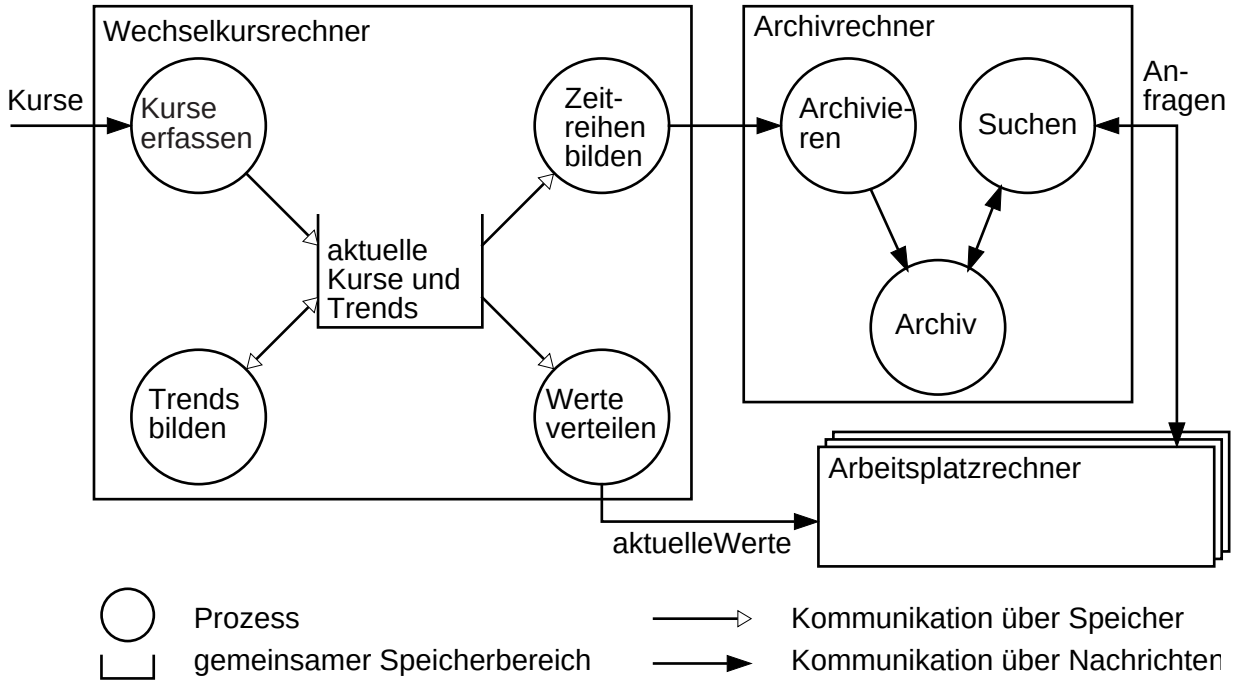


BILD 6.5. Beispiel einer prozessorientierten Architektur

6.6.5 Komponentenorientierte Architektur

Die komponentenorientierte Architektur ist eine Weiterentwicklung der objektorientierten Architektur, insbesondere im Hinblick auf Verteilung und Wiederverwendung. Es zeigt sich nämlich, dass Klassen als Einheiten für die geographische Verteilung eines Systems und als Einheiten käuflicher, wiederverwendbarer Software oft zu klein sind.

In einer komponentenorientierten Architektur versteht man unter einer *Komponente* eine Menge zusammengehöriger Objekte bzw. zusammengehöriger Klassen, die von ihrer Umgebung abgekapselt und nur über eine oder mehrere Schnittstelle(n) der Komponente zugänglich sind. Komponenten in diesem Sinn sind nach dem Geheimnisprinzip gebildete Module auf einer Stufe oberhalb von Objekten und Klassen, die zudem geographisch verteilt sein können.

Die Komponenten kommunizieren miteinander über einen *Makler* (broker): Einerseits geben sie nach außen eine Schnittstelle bekannt, über die sie angesprochen werden können. Andererseits greifen sie mit Hilfe des Maklers auf Objekte anderer Komponenten zu, indem sie in deren Schnittstelle definierte Operationen aufrufen (Bild 6.6). Die Komponenten sind stark voneinan-

der entkoppelt. Sie brauchen weder die Art der Realisierung der Kommunikation durch den Makler, noch die geographische Lokalisierung der Partnerkomponenten noch deren Implementierung zu kennen. Wenn die Schnittstellen in einer eigenen Schnittstellenbeschreibungssprache (interface definition language, IDL) beschrieben sind, können die Komponenten sogar in unterschiedlichen Programmiersprachen realisiert sein.

Ein typischer Vertreter einer komponentenorientierten Architektur ist die sogenannte *Client/Server-Architektur*. Eine Client/Server-Architektur besteht aus einer Menge von Klienten (Clients), d.h. Anwendungsprogrammen, welche auf Arbeitsplatzstationen laufen, einer Menge von Lieferanten (Servers), die Daten bereitstellen (Datenbanksysteme, Dateisysteme) und einer Kommunikationsschiene, welche alle Komponenten miteinander verbindet. Erweitert man die Kommunikationsschiene zu einer Zwischenschicht, welche Kommunikations-, Koordinations- und Übersetzungsdienste sowie ausgewählte Teile der Anwendung realisiert, so spricht man von einer *Middleware-Architektur*. (Tresch 1996).

Komponentenorientierte Architekturen werden manchmal auch als *Architektur mit verteilten Objekten* bezeichnet.

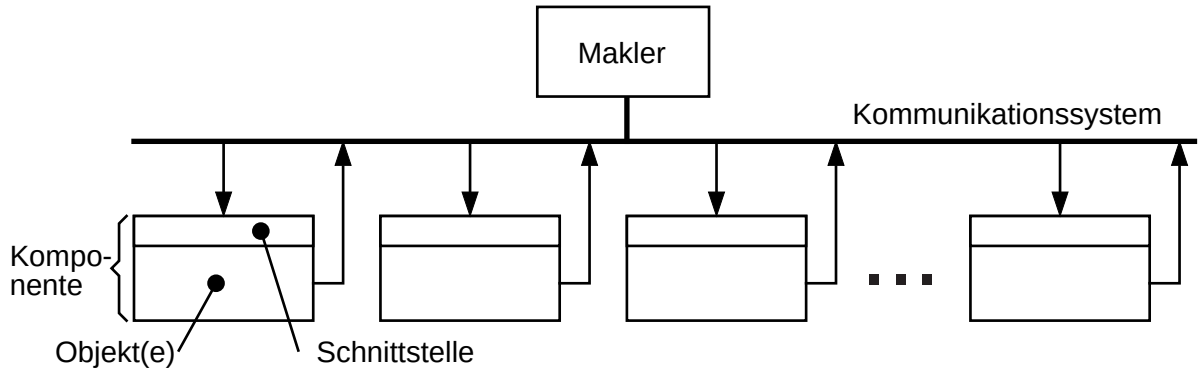


BILD 6.6. Beispiel einer komponentenorientierten Architektur

6.7 Wiederverwendung von Architekturen und Entwurfswissen

6.7.1 Entwurfsmuster

Entwurfsmuster (Gamma et al. 1995) stellen bewährte, vorgefertigte Lösungsschablonen für wiederkehrende Entwurfsprobleme bereit und ermöglichen damit die Wiederverwendung von Entwurfswissen. Sie schaffen ferner eine begriffliche Basis und Terminologie für die Kommunikation über solche Probleme.

DEFINITION 6.10. *Entwurfsmuster (design pattern)*. Eine spezielle Komponente, die eine allgemeine, parametrierbare Lösung für ein typisches Entwurfsproblem bereitstellt.

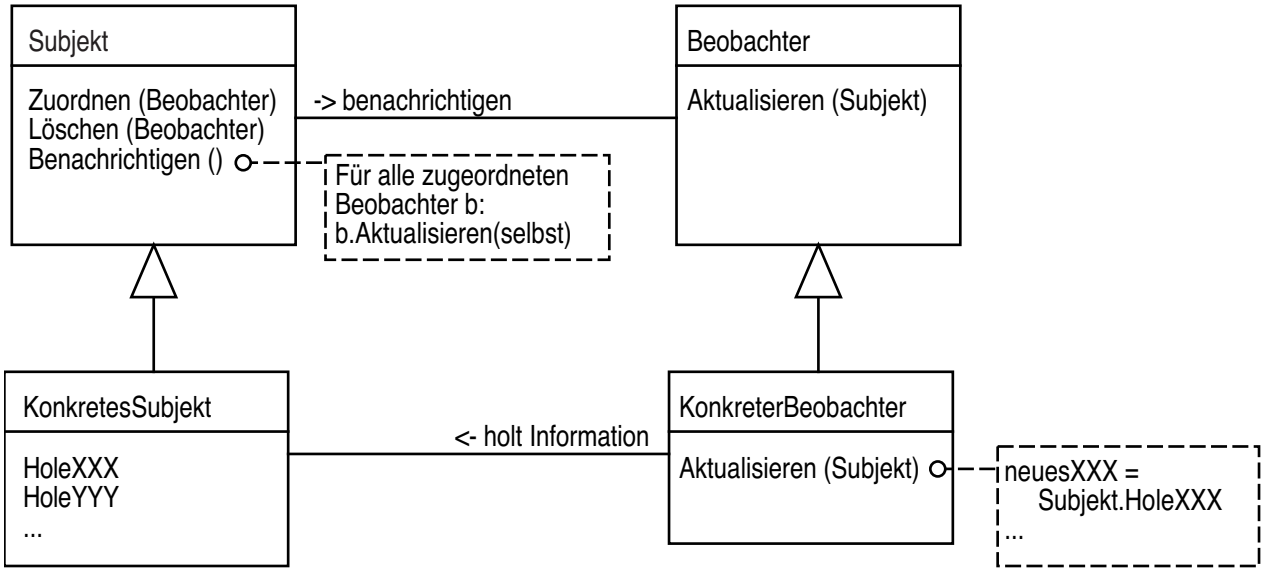
Grundsätzlich gibt es eine Fülle von Mustern, sowohl allgemeiner Art als auch für spezifische Anwendungsbereiche. *Musterkataloge* erschließen das Wissen über Muster und machen die Muster wiederverwendbar.

Entwerfen mit Mustern setzt voraus, dass die Entwerfenden einen Grundschatz an Mustern kennen. Bei der Modularisierung müssen Anwendungssituationen für Muster erkannt und die entsprechenden Muster verwendet werden.

Bild 6.9 zeigt das *Beobachtermuster* (observer pattern, Gamma et al. 1995) als Beispiel. Die Intention des Beobachtermusters ist, dass Objekte Daten bei einem Informationsanbieter (Sub-

jekt) abonnieren können. Bei jeder Änderung der abonnierten Daten werden die Abonnenten (Beobachter) automatisch über die Änderung informiert bzw. werden ihnen die geänderten Daten zugestellt. Dieses Muster hat zwei typische Anwendungen:

- Entkopplung voneinander abhängiger Objekte durch Ersetzung direkter Kommunikation über Aufrufe durch eine indirekte über Benachrichtigung
- Trennung von Informationsbereitsteller und einer Menge von Informationsverarbeitern bzw. -darstellern.



1 ZUORDNEN fügt ein Beobachterobjekt in die Liste der zu benachrichtigenden Beobachter ein.

2 Nach jeder Veränderung ruft ein KONKRETESSUBJEKT seine (geerbte) Operation BENACHRICHTIGEN auf. Diese iteriert die Liste der Beobachter und schickt jedem Beobachter die Botschaft AKTUALISIEREN. Das benachrichtigende Subjekt wird als Parameter mitgegeben.

3 Jeder benachrichtigte Beobachter reagiert, indem er beim benachrichtigenden Subjekt mit den Botschaften HOLEXXX, HOLEYYY etc. die ihn interessierenden Informationen abruft.

Varianten:

- Der Botschaft AKTUALISIEREN können weitere Informationen (z.B. die Art des eingetretenen Ereignisses) als Parameter mitgegeben werden.
- Der Botschaft AKTUALISIEREN können die veränderten Daten gleich mitgegeben werden, wodurch der Abruf durch HOLEXXX entfällt. Dieses Bringprinzip ist effizient, koppelt aber Gegenstand und Beobachter stärker als das mit HOLEXXX realisierte Holprinzip.

BILD 6.7. Das Beobachtermuster

6.7.2 Rahmen

Rahmen (frameworks) stellen vorgefertigte Lösungsgerüste für bestimmte Problemklassen bereit und gestatten damit die Wiederverwendung ganzer Architekturen.

DEFINITION 6.11. *Rahmen (framework).* Eine Menge kooperierender Module, die das Grundgerüst für die Lösung einer bestimmten Klasse von Problemen bilden.

Konkrete Lösungen entstehen durch Ergänzung oder Spezialisierung des Rahmens durch problemspezifische Module. Der Umfang eines Rahmens kann stark variieren:

- Enge Rahmen lösen ein spezifisches Teilproblem, zum Beispiel die Realisierung einer Benutzerschnittstelle, die Verwaltung von Dateien mit Vergabe und Prüfung von Zugriffsrechten oder die Grundfunktionalität eines Editors.

- Umfassende Rahmen bilden das Lösungsgerüst für ein vollständiges System, zum Beispiel ein Rahmen für das Schaltergeschäft in einer Bank.

Manchmal werden in einer Architektur mehrere Rahmen gleichzeitig verwendet. Dabei muss der Vorteil der Wiederverwendung allerdings sorgfältig abgewogen werden gegen die Nachteile möglicher Architekturkonflikte zwischen den verschiedenen Rahmen. Die Einbettung unterschiedlicher Architekturprinzipien der verschiedenen Rahmen in eine gemeinsame, kohärente Gesamtarchitektur kann sehr schwierig bis sogar unmöglich sein.

6.7.3 Architekturmuster

Analog zu Entwurfsmustern sind Architekturmuster vorgefertigte Strukturen für typische Architekturprobleme.

Definition 6.12. *Architekturmuster*. Eine vorgefertigte, parametrierbare Schablone für die Gestaltung der Architektur eines Systems oder einer Komponente.

Es gibt eine Fülle von Architekturmustern. Man kann unterscheiden zwischen Strukturmustern, Steuermustern und Modularisierungs-/Entkopplungsmustern. Nachfolgend werden einige ausgewählte Architekturmuster beschrieben.

6.7.3.1 Strukturmuster: Das Matrixmuster

Das Matrixmuster (Bild 6.8) gliedert ein System in je eine Menge von Datenmodulen und von Funktionsmodulen, wobei jede Funktion auf jedes Datum zugreifen kann. Die Funktionsmodule enthalten keine permanenten Daten. Typische Anwendungen dieses Musters findet man in der klassischen Architektur datenbankbasierter Systeme.

6.7.3.2 Steuermuster

Das EVA (Eingabe-Verarbeitung-Ausgabe)-Muster

Ein Steuermodul steuert nacheinander (in Sequenz oder iterativ) Eingabe-, Verarbeitungs- und Ausgabemodule an. Dies ist das klassische Steuermuster für funktionsorientierte sequentielle Programme (vgl. Bild 6.2).

Das Hauptschleifenmuster

Ein Prozess misst, regelt und steuert, indem er in einer Endlosschleife zyklisch alle Datenquellen (Sensoren, etc.) abfragt und alle Datensenken (Anzeigen, Aktuatoren...) mit aktualisierten Werten versorgt. Dieses Muster wird wegen seiner Robustheit gerne für die Steuerung sicherheitskritischer Anwendungen verwendet.

Das Hollywood-Muster (“Don’t call us, we call you”)

Ein Ereignisverwalter registriert alle Eingabeereignisse und ruft die zugehörigen Dienste auf. Das Anwendungsprogramm enthält kein Hauptprogramm mehr. Dieses Muster liegt den meisten käuflichen Anwendungsrahmen zugrunde.

6.7.3.3 Modularisierungs-/Entkopplungsmuster: Das Model-View-Controller (MVC)-Muster

Ein System wird in drei Teile gegliedert: Das Modell (“Model”) enthält die Anwendungslogik und das Modell des Anwendungsbereichs, die Sicht (“View”) realisiert die äußere, sichtbare Repräsentation des Systems und die Steuerung (“Controller”) behandelt alle Benutzereinga-

ben (Krasner und Pope, 1988). Mit diesem Muster werden die Anwendungslogik, die Repräsentation der Anwendung gegenüber der Außenwelt und die Steuerung der Eingaben voneinander entkoppelt. Das MVC-Muster ist ein zentrales Muster im Entwurf objektorientierter Systeme. Es kann auch im Kleinen als Entwurfsmuster verwendet werden (Gamma et al. 1995).

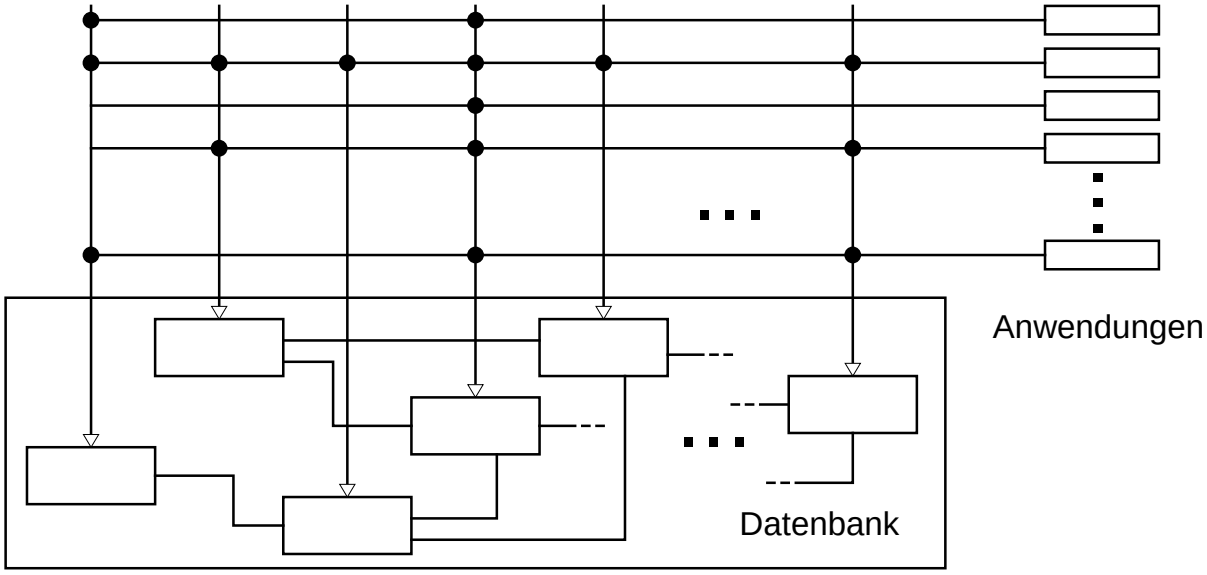


BILD 6.8. Architektur eines datenbankbasierten Anwendungssystems nach dem Matrixmuster

6.8 Architekturmetaphern

Eine *Metapher* ist ein sprachlicher Ausdruck, bei dem ein Wort aus seinem Bedeutungszusammenhang in einen anderen übertragen, *als Bild verwendet wird*.

Definition 6.13. *Architekturmetapher*. Leitbild für die Gestaltung einer Architektur.

Eine *Architekturmetapher* ist ein Modell, das eine Architektur über analoge, vertraute Bilder erschließt. Die Metapher ermöglicht ein besseres Verständnis der Systemstruktur und ist das Leitbild für die Gestaltung der Architektur. In den folgenden Abschnitten werden einige Architekturmetaphern kurz vorgestellt.

6.8.1 Die WAM (Werkzeug-Automat-Material)-Metapher

Das System besteht aus *Materialien* (fachliche Arbeitsgegenstände der Anwendung), die von den Benutzern mit passenden *Werkzeugen* interaktiv bearbeitet werden. Vollständig automatisierbare Routineaufgaben werden von *Automaten* erledigt (Züllighoven, 1998).

- Werkzeuge sind gegenüber den Materialien aktiv, indem sie Materialien bearbeiten. Sie werden von Menschen benutzt bzw. bedient. Diesen gegenüber verhalten sie sich passiv oder assistierend; sie überlassen die Kontrolle immer den bedienenden Menschen.
- Materialien sind passiv und/oder speichernd. Sie werden bearbeitet und können sowohl Arbeitsgegenstand als auch Arbeitsergebnis sein.
- Automaten sind aktiv. Sie erledigen Aufgaben vollautomatisch und ohne menschliches Zutun.

Die WAM-Metapher eignet sich insbesondere für die Architektur von Assistenzsystemen, d.h. Systemen, welche kreative menschliche Arbeit (zum Beispiel Beratungs- oder Entwurfstätigkeiten) unterstützen.

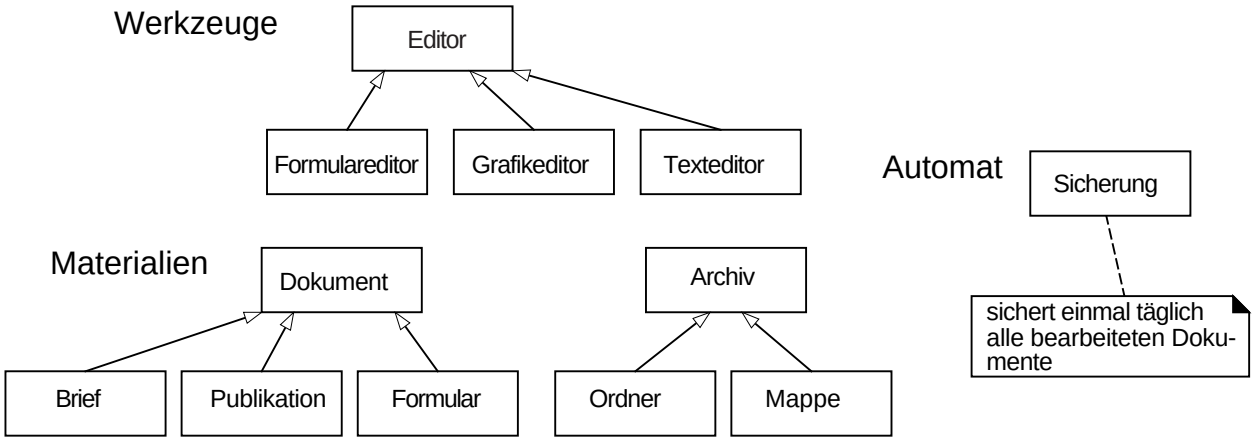


BILD 6.9. Die WAM (Werkzeug-Automat-Material)-Metapher

6.8.2 Weitere Architekturmetaphern

- **Die Organisationshierarchie.** Die Architektur wird analog zur Organisationshierarchie in einem Unternehmen strukturiert, mit Delegation von Aufgaben und Verantwortung von oben nach unten und Berichten von unten nach oben.
- **Die virtuellen Maschinen.** Die Architektur ist als eine Menge aufeinander aufbauender Schichten von *virtuellen*, d.h. künstlichen Maschinen aufgebaut. Die Maschinen jeder Schicht bieten Leistungen für die darüberliegende Schicht an und benutzen dazu die Maschinen der darunterliegenden Schicht. Kommunikationsarchitekturen sind meistens nach dieser Metapher konstruiert.
- **Das Steckersystem.** Die Architektur wird in Analogie zu technischen Steckersystemen aufgebaut. In einen Grundrahmen, welcher typisch Kommunikations-, Dialog, und Datenverwaltungsdienste anbietet, werden Anwendungselemente „eingesteckt“. Anwendungsrahmen (vgl. 6.7.2) sind oft nach dieser Metapher konstruiert (Bild 6.10).

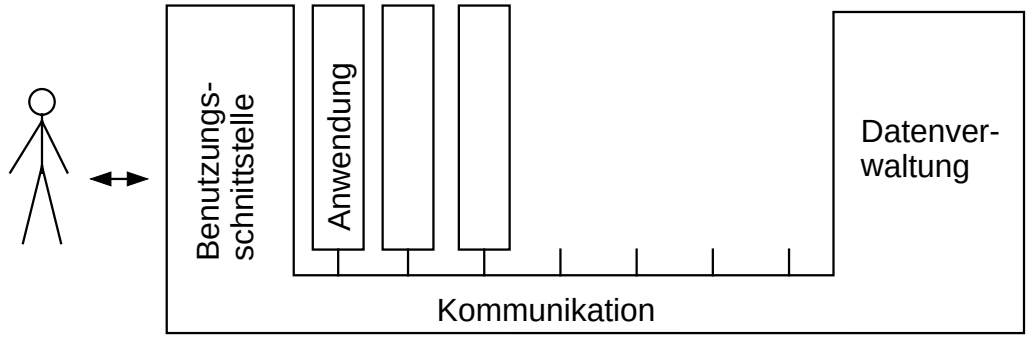


BILD 6.10. Die Steckersystem-Metapher

Aufgaben

6.1 Welche Vorteile hat eine Modularisierung nach dem Geheimnisprinzip (a) für die Verwender eines Moduls, (b) für die Modulersteller/für das Pflegepersonal?

6.2 Gegeben sei folgende Modularisierung der Steuerung einer Abfüllanlage für Flüssigkeiten:
Modul Tank: Steuerung des Einlassventils, Behandlung der Sensoren für vollen und leeren Tank, Meldung des aktuellen Tankzustands an Leitstand
Modul Abfüllventil: Steuerung des Auslassventils am Tank
Modul: Abfüllung: Steuerung des Fließbands mit den abzufüllenden Behältern, Behandlung des Waage-Sensors (misst die abgefüllte Menge)
Randbedingungen: 1. Bei leerem Tank darf das Auslassventil nicht geöffnet sein. 2. Während der Nachfüllung des Tanks (Einlassventil offen) muss das Auslassventil geschlossen sein.
Beurteilen Sie diese Modularisierung, insbesondere im Hinblick auf Information Hiding und Zusammenarbeit.

Ergänzende und vertiefende Literatur

Fairley (1985) sowie Ghezzi, Jazayeri und Mandrioli (1991) beschreiben die Prinzipien guten Entwurfs.

McDermid (1991) enthält eine Übersicht über klassische (nicht-objektorientierte) Entwurfsverfahren.

Das Tutorium von Freeman und Wasserman (1983) gibt anhand von Originalartikeln einen Überblick über verschiedene klassische Entwurfstechniken. Unter anderem ist auch Parnas (1972) abgedruckt.

Meyer (1997) bzw. (1988) ist *das* klassische Lehrbuch über objektorientierten Entwurf. Weitere lesenswerte Bücher über objektorientierten Entwurf sind Booch (1994), Wirfs-Brock, Wilkerson, Wiener (1990) und Züllighoven (1998).

Shaw und Garlan (1996) geben einen Überblick über Software-Architektur und gehen insbesondere auf verschiedene Architekturstile und auf Architekturbeschreibungssprachen ein. Garlan und Shaw (1993) ist eine Kurzfassung, die sich hauptsächlich mit Architekturstilen beschäftigt. Bass, Clements und Kazman (1998) ist ebenfalls ein Lehrbuch über Software-Architektur.

Gamma, Helm, Johnson und Vlissides (1995) führen in das Konzept der Entwurfsmuster ein und beschreiben 23 häufige und typische Muster im Detail.

Dijkstra (1968) beschreibt als erster eine Architektur, die aus mehreren aufeinander aufbauenden Schichten besteht.

Parnas (1972) beschreibt als erster das Geheimnisprinzip (Information Hiding).

Züllighoven (1998) beschreibt ein in langjähriger Praxis erprobtes Vorgehen für die Konstruktion objektorientierter Systeme.

Zitierte Literatur

Bass, L., P. Clements, R. Kazman (2003). *Software Architecture in Practice*. 2nd edition Reading, Mass.: Addison-Wesley.

Booch, G. (1994). *Object Oriented Analysis and Design with Applications*. Second Edition. Redwood City, Ca.: Benjamin/Cummings.

Dijkstra E.W. (1968). The Structure of the THE multiprogramming System. *Communications of the ACM* **11**, 5 (May 1968). 341-346.

Fairley, R.E. (1985). *Software Engineering Concepts*. New York, etc.: McGraw-Hill.

Freeman, P., A. I. Wasserman (eds.) (1983). *Tutorial on Software Design Techniques*. IEEE Computer Society Press, Order Number 514.

Gamma, E., R. Helm, R. Johnson, J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass. etc.: Assison-Wesley.

Garlan, D. M. Shaw (1993). An Introduction to Software Architecture. In V. Ambriola and G. Tortora (eds.) *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, Singapore. 1-39 [Auch erschienen als technische Bericht von CMU bzw. SEI: CMU-CS-94-166, CMU/SEI-94-TR-21]

Ghezzi, C., M. Jazayeri, D. Mandrioli (1991). *Fundamentals of Software Engineering*. Englewood Cliffs: Prentice-Hall.

IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. IEEE Computer Society Press.

Krasner, G.E., S.T. Pope (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* **1**, 3. 26-49.

Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software* **12**, 6 (Nov 1995). 42-50.

McDermid, J. (1991). *Software Engineer's Reference Book*. Oxford: Butterworth-Heinemann. ca. 1200 p.

Meyer, B. (1988). *Object-Oriented Software Construction.* Englewood Cliffs, N.J.: Prentice Hall.
[auf Deutsch: *Objektorientierte Softwareentwicklung*, München: Hanser, 1990]

Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer* **25**, 10 (Oct. 1992). 40-51.

Meyer, B. (1997). *Object-Oriented Software Construction.* Englewood Cliffs, N.J.: Prentice Hall. [Neuauflage von Meyer (1988)]

Page-Jones, M. (1988). *The Practical Guide to Structured Systems Design*. Englewood Cliffs, N.J.: Prentice Hall.

Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* **15**, 12 (Dec. 1972). 1053-1058.

Shaw, M., D. Garlan (1996). Software Architecture: Perspectives on an Emerging Discipline. Englewood Cliffs, N.J.: Prentice Hall. 242 p.

Stevens, W.G., G. Myers, L. Constantine (1974). Structured Design. *IBM Systems Journal* **13**, 2. 115-139.

Tresch, M. (1996). Middleware: Schlüsseltechnologie zur Entwicklung verteilter Informationssysteme. *Informatik-Spektrum* **19**, 5 (Okt 1996). 249-256.

Wirfs-Brock, R., B. Wilkerson, L. Wiener (1990). *Designing Object-Oriented Software*. Englewood Cliffs, N.J.: Prentice Hall.
[auf Deutsch: *Objektorientiertes Software Design*, München: Hanser, 1993]

Züllighoven, H. (1998). *Das objektorientierte Konstruktionshandbuch*. Heidelberg: dpunkt Verlag.