1. Einführung

Softwarearchitektur ist ein Teilgebiet des Software Engineering mit folgenden Punkten:

- Es soll Wissen über gut designte Architekturen, Frameworks und Architekturmuster vermitteln
- Es sollen Techniken, Methoden, Modell, Beschreibungssprachen und Tools entwickelt werden, die das Verständnis, die Analyse, das Design, die Konstruktion, die Beurteilung und die Entwicklung von Softwaresystemen verbessern.

Ein **Software System** ist ein technisches System oder Teil eines Systems, in dem Software eine dominante Rolle spielt. Beispiele:

- Schachcomputer – interaktives System
- Pascal Compiler – Input-, Output-System
- OS – interaktive Plattform zum Ausführen von Programmen
- WWW – Client-Server-System zum Dokumentenaustausch über das Internet
- Softwaresystem eines Airbags im Auto – embeded realtime system

Es gibt immer eine Plattform für Softwaresysteme oder sie sind ein Subsystem auf einem anderen System.

Beschreibungen: Softwaresystemen können verschiedene Strukturen haben. Das Problem ist, die Strukturen festzuhalten und zu beschreiben, so dass sie die Beziehungen widerspiegeln und dabei unnötige Redundanzen zu vermeiden.

- ein System, dass als einfache Applikation in einem Prozess läuft
- nonterminating systems – Plattformen für Anwendungen
- über Netzwerke weit verbreitete Systeme mit mehr oder weniger verknüpften Prozessen (Reservierungssystem oder www)

Die Struktur wird beschrieben durch das Zerlegen des Systems in Komponenten, dem definieren von Schnittstellen und dem Beschreiben der Beziehungen auf einer abstrakten Ebene.
Ein gutes Fundament einer Softwarearchitektur muss folgende Probleme lösen:

- Wie können Schnittstellen von Komponenten und deren Verbindungen modelliert und spezifiziert werden?

- Wie können Beschreibungen con verschiedenen Strukturen zu einem Framework zusammengefasst werden?
- Wie können die Beschreibungen mit formalen Methoden kombiniert werden, um die Systemeigenschaften zu kontrollieren?
- Wie kann man eine Effizienz- und Sicherheitsanalyse aufgrund der Beschreibung machen?

Bei einem installierten Softwaresystem befinden sich Daten und Binärdateien auf einer Festplatte. Bei der Ausführung werden Teile in den Arbeitsspeicher geladen. Das System hat verschiedene Teile:

- Conceptual Structure (Systemvoraussetzungen)
- Data flow (sends-data-to relation on system components)
- Control flow (becomes-active-after relation on system components)
- Hierachical structure (is-subsystem-of relation)
- Uses/call structure (Beziehungen zwischen Modulen, Klassen und Prozeduren)
- Process structure (Prozesse und Threads – synchronisiert mit, kann-nicht-laufen-ohne ...)
- Physical structure (beschreibt die Verteilung von Code und Daten auf der zugrunde liegenden Plattform)

Die **Softwarearchitektur eines Programms** oder Computersystems ist die Struktur, die alle Komponenten beinhaltet, die von außen sichtbaren Eigenschaften dieser Komponenten und die Beziehungen zwischen ihnen.
Im einzelnen:

- es wird beschrieben, wie die einzelnen Teile zusammenarbeiten
- es gibt keine EINE Architektur
- jede Software hat eine Architektur, denn sie kann in Komponenten zerlegt werden

Eine Softwarearchitektur ist die Beschreibung von Untersystemen und Komponenten eines Systems und deren Beziehungen zwischen einander.

Ein Softwaresystem ist **implementiert**, wenn es vollständig programmiert ist – Linux kann noch während der Installation konfiguriert werden.

Product-line-architecture – Spezielle Architektur einer Firma, die sich in allen Produkten widerspiegelt (Kosteneinsparung)

2. Software Systeme und Architektur

Softwaresysteme unterscheiden sich in Bezug auf ihre Plattform, die Systemvoraussetzungen und zum Beispiel ihren Nutzen.

Installierte oder „laufende“ (WWW) Softwaresysteme haben:

- eine Identität (wo ist das Programm installiert oder gestartet)
- eine Lebenszyklus (von der Installation bis zur Deinstallation)
- einen Status zu einer Zeit (welche Programmteile laufen, Status von Variablen)
- ein Verhalten

Schnittstellen des System bestehen zur Plattform (gestrichelt) und zur Systemumgebung (durchgezogen). Diese Schnittstellen haben unterschiedliche Eigenschaften.

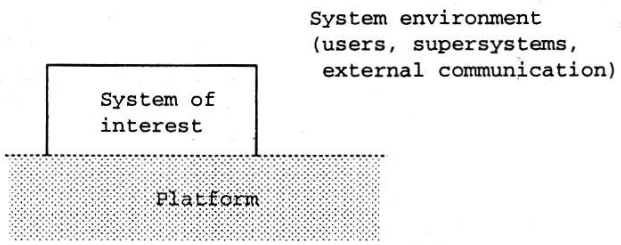

Figure 2.1: The two principle interface of a software system

Die **Plattform** ist ein Software- oder Hardwaresystem. Sie bietet Basisfunktionen, auf die das System aufbaut (z.B. für ein OS ist die Hardware die Plattform, für einen Browser ist das OS die Plattform).
Das Softwaresystem benutzt bzw. baut auf Teile der Plattform auf. Unterschiede zwischen Plattform:

- auf einer Plattform können verschiedene voneinander unabhängige Softwaresysteme laufen. Es ist nötig, die Plattform zu beschreiben.
- eine Beschreibung der Plattform und des Softwaresystems zeigen, die relevanten Teile des Systems.

Während der Laufzeit wird nur mit dem System gearbeitet. die Plattform wird nicht benötigt. Aber! Der Benutzer benötigt Zugriff zur Plattform, um das System zu starten.

Systemumgebung Ein System arbeit selten isoliert allein. Es gibt Interaktionen mit anderen Systemen oder mit dem Benutzer. Das ist die Umgebung, egal ob es technische Systeme oder Menschen sind.

Für die Kommunikation des Systems mit der Plattform muss nur die Implementierung umgesetzt werden. Für die Systemumgebung gibt es spezielle Schnittstellen z.B. GUI.

Einteilungen von Softwaresystemen:

- Wozu wird das System benutzt? (*Applikation*, wenn User mit dem Programm arbeiten; *embedded*, wenn es in ein System integriert ist; *Plattformsystem*, wenn es als Basis für andere Systeme dient) (in sich *geschlossene* Systeme – *offene* Systeme z.B. Browser können durch Plugins erweitert werden)
- Wie ist das Verhalten des Systems? (Batch-Prozess oder nonterminating; auch reaktiv oder interaktiv; Singleuser oder Multiuser-System)
- Weitere Möglichkeiten: rechensintensive oder Dialogprozess; das System läuft als ein Prozess oder mit mehreren Prozessen oder verteilt auf verschiedene Rechner

Als **Software** werden Programme und Daten bezeichnet. **Programme** sind Befehlsfolgen in einer formalen Sprache. **Daten** sind Informationen, die in Rechner gespeichert werden.
Gründe, warum Programme *und* Daten als Software bezeichnet werden:

- Programme können wie Daten behandelt und manipuliert werden und bestimmte Daten können als Programme interpretiert werden.
- Programme und Daten können vermischt sein. z.B. HTML-Seiten können Text und Java-Code enthalten.
- Der Umgang mit Dateien ob Daten oder Programme ist der gleiche.

Die interessantere Frage ist, was ist Software in einem Softwaresystem. Nicht nur der installierte Code, sondern alle benutzten Programme und Daten, die beider Ausführung benutzt werden, gehören zur Software.

Ein **Softwaresysteme** ist ein technisches System oder Teilsystem, bei dem Software im Mittelpunkt steht. Es sollte eindeutige Schnittstellen zur Plattform und zur Systemumgebung haben. Es muss für unterschiedliche Installationen auf verschiedenen Plattformen parametriert

sein, es beinhaltet verschiedene Funktionen, die entsprechend Benutzern bzw. Systemumgebung anpassen.
Softwaresysteme haben eine implizierte oder optionale Anzahl von Parametern (es kann auf verschiedenen Plattformen laufen oder Office kann mit oder ohne Rechtschreibprüfung installiert werden).
Ein Softwaresystem ist ausführbar, installiert bzw. eingerichtet, konfiguriert (vorbereitet für die Installation), programmiert (Teile für mögliche Parametrierungen sind implementiert) und designt (komplette Spezifikation inklusive Beschreibung der Architektur)

Beispiele für Software-Architekturen:

GCC - GNU Compiler Collection: Verschiedene Sprachen können kompiliert werden. Es werden Fehlermeldungen für den Benutzer ausgegeben und der Code kann für verschiedene Plattformen konfiguriert werden. GCC ist kein einfaches Softwaresystem, sondern eine Familie von Systemen.

Three Tier Architecture wird beim WebAssign benutzt. Studenten und Betreuer arbeiten mit diesem System. Es wird als User-System-Interface bezeichnet. Der Server ist die zentrale Recheneinheit. Die dritte Komponente ist das Datenbank-Management.

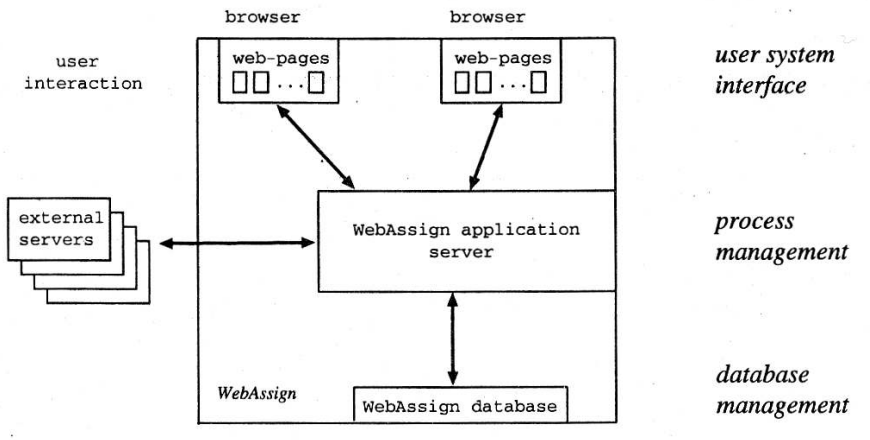

Figure 2.2: Conceptual architecture of the WebAssign system

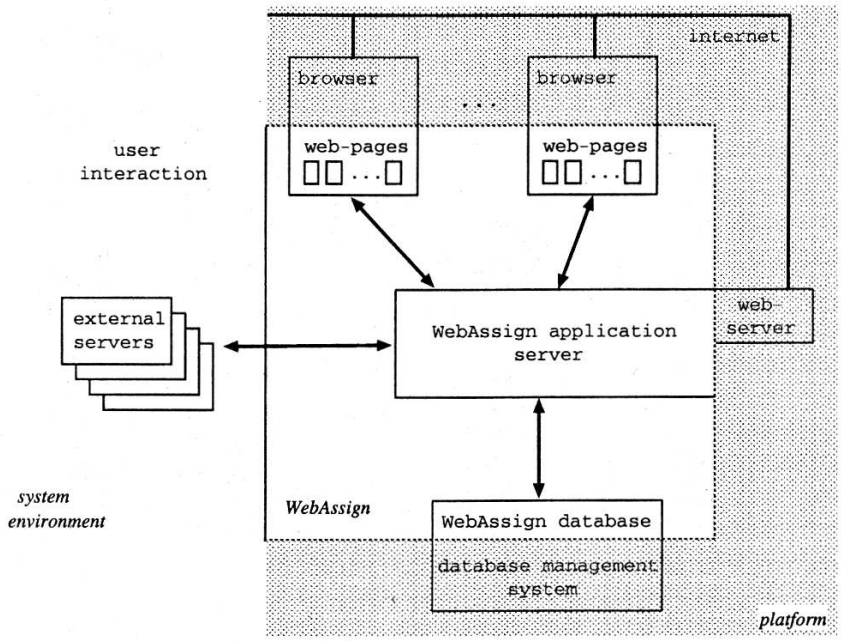

Figure 2.3: Conceptual architecture of the Web-Assign system with platform interface and external interface

3. Software Systeme und Architektur

Eine **Struktur** identifiziert wichtige Komponenten oder Bestandteile des Systems und erklärt ihre Beziehung und Interaktion.
Eine **Systemkomponente** ist ein Teil des laufenden Systems (ein Prozess) oder ein Teil der Software (ein Modul oder eine Prozedur).

Strukturen könne in Diagrammen durch verschienen Symbole und Verbindungen oder durch Texte (UML) beschriebnen werden.

- Welche Struktur soll zur Beschreibung einer Architektur gewählt werden?
- Wie ist die Beziehung zwischen den Strukturen? Ist sie konsistent?

Ein **View** zeigt bestimmte Eigenschaften einer Architektur oder die Architektur aus einem bestimmten Blickwinkel.
Es gibt verschiedene Ansichten, was ein View beschreiben soll. Z.B. Hofmeister, Nord und Soni: konzeptioneller View (beschreibt Hauptbestandteile und ihre Beziehung), Modul-View (funktionale Beschreibung und Layereinteilung), Excusion View (Ausführung – beschreibt die dynamische Struktur des Systems), Code-View (beschreibt, wie der Sourcecode organisiert ist)

Der **dynamic View** zeigt das Laufzeitverhalten. Von Bedeutung sind Prozesse, Threads, Objekte und Laufwerke die verbunden sind durch Procedure Calls, Protokolle oder eventbasierende Kommunikation. Relevant sind Ausführungszeit, Speicherbedarf und benötigte Bandbreite.

To illustrate the dynamic view of software systems, we consider the basic architecture of the World Wide Web. The WWW is a nonterminating system. At each point in time, it consists of a set of servers and a set of clients. Servers and clients are essentially processes. Servers run most of the time, have an address, wait for client requests, and can access a file system or database with Web-pages (a Web-page may contain text, graphics, and images; its structure and layout attributes are usually described in HTML). The address of the Web-page is composed of the address of the server and the name of the file containing the Web-page. The typical Web-client is a so-called browser. A browser allows one to load and display Web-pages. It is started and terminated during a user session. Web-server and -clients communicate over the Internet using the http-protocol. Figure 3.1 shows part of the WWW, namely three running browsers and two serverswith addresses www.bundestag.de and dict.leo.org.

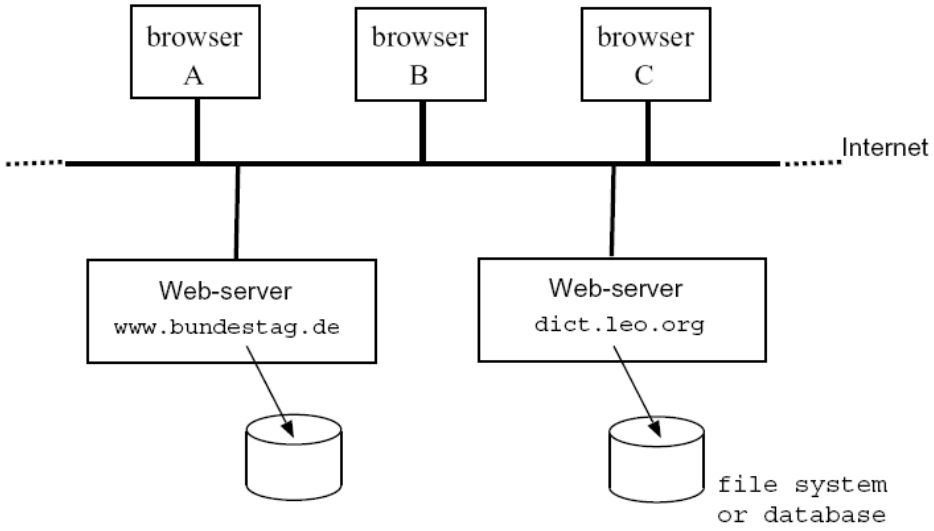

Figure 3.1: A tiny part of the WWW

Das Ausführen von Softwaresystemen basiert auf Programme und statische Daten mit Systemparametern. Der **statische View** zeigt die so genannten statischen Strukturen der Software.

Teile eines statischen View sind Programme, Bibliotheken, Module, Prozeduren, Klassen oder strukturierte Daten. Programme können in verschiedenen Programmier- oder

Scriptingsprachen vorkommen. Es gibt Sourcecode oder Binaries, verschiedene Versionen und verschiedene Konfigurationen für unterschiedliche Plattformen.
Eine einzelne Komponente eines statischen Views ist ein Software Package.

Es gibt parallelen zwischen dem dynamischen und statischen View.

Elements of dynamic structures	Elements of static structures
process	program
object	class
procedure incarnation	procedure
state	variables/files/execution points/...
protocol	??
event	?? (procedure)
??	module/package

Figure 3.2: Relationship between dynamic and static view

Dynamischer View eines Browsers (gestrichelte Linien für Eventbased-Communication):

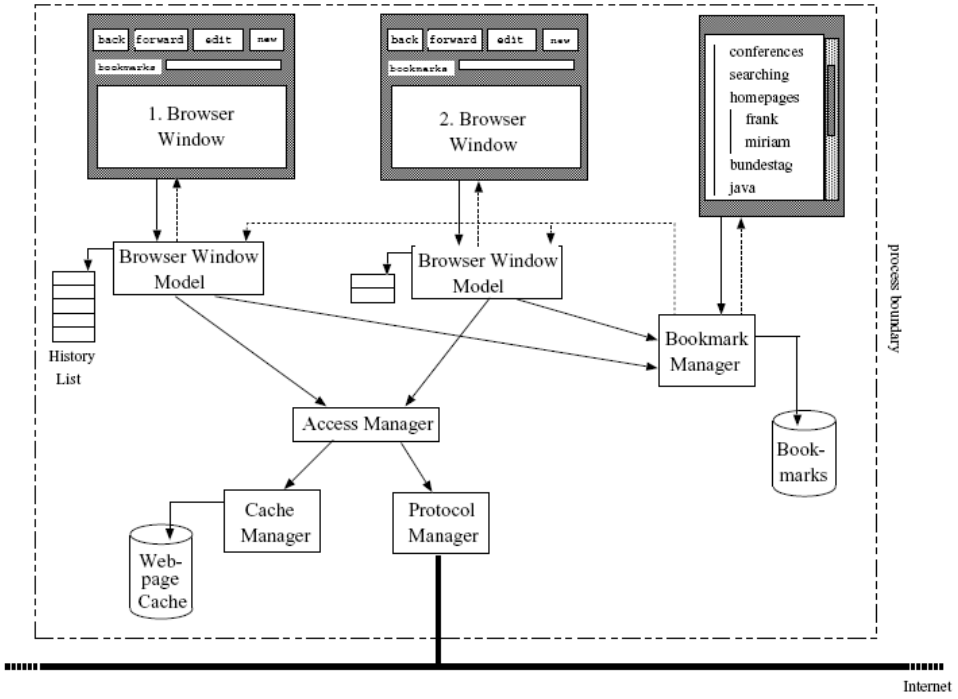

Figure 3.3: Conceptual structure of a browser

Dynamischer View eines Browsers:

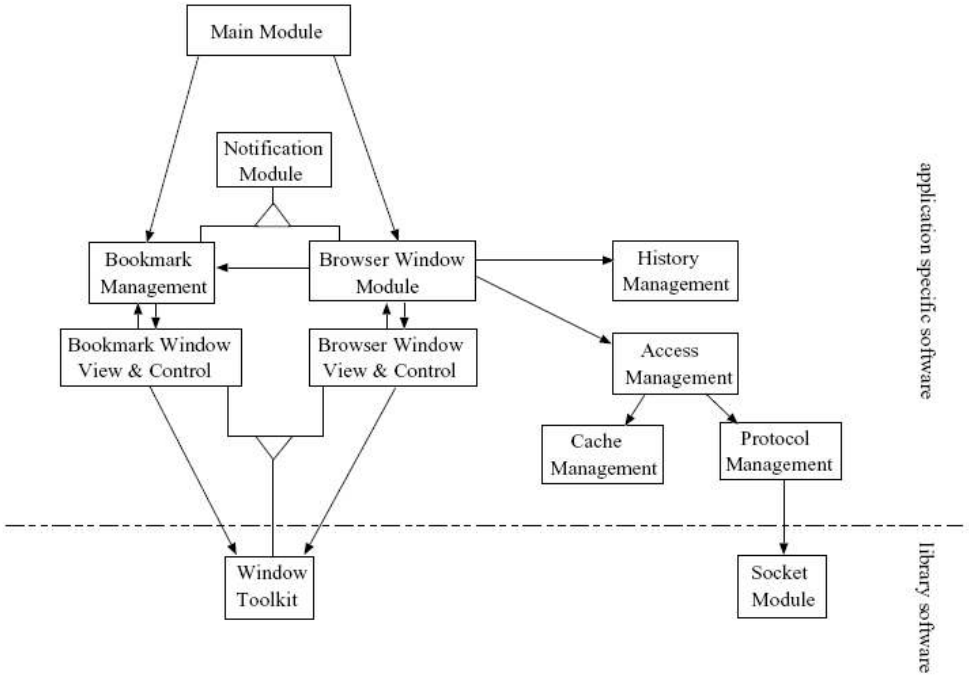

Figure 3.4: Module structure of a browser

Auf der Abstraktionsebene gibt es zwischen beiden Views vier Unterschiede:

- Die dynamische Struktur muss die Replikation von Komponenten festhalten.
- Was in der dynamischen Struktur als Verbindung erscheint, muss in der statischen Struktur durch Software dargestellt werden (Eventbased-Communicateion als notification module).
- Die statische Struktur benötigt Komponenten zum Systemstart und zur Initialisierung.
- Die statische Struktur zeigt die mehrfache Nutzung von Modulen durch das wieder verwenden der Software. ??

Beispiel Bankautomat:

use case diagramm:

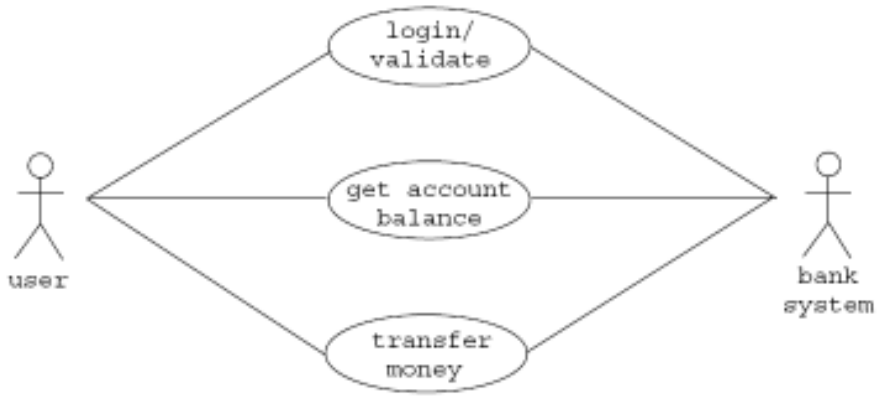

Figure 3.5: Basic tasks of the bank application

Es werden keine Beziehungen und kein Status dargestellt.

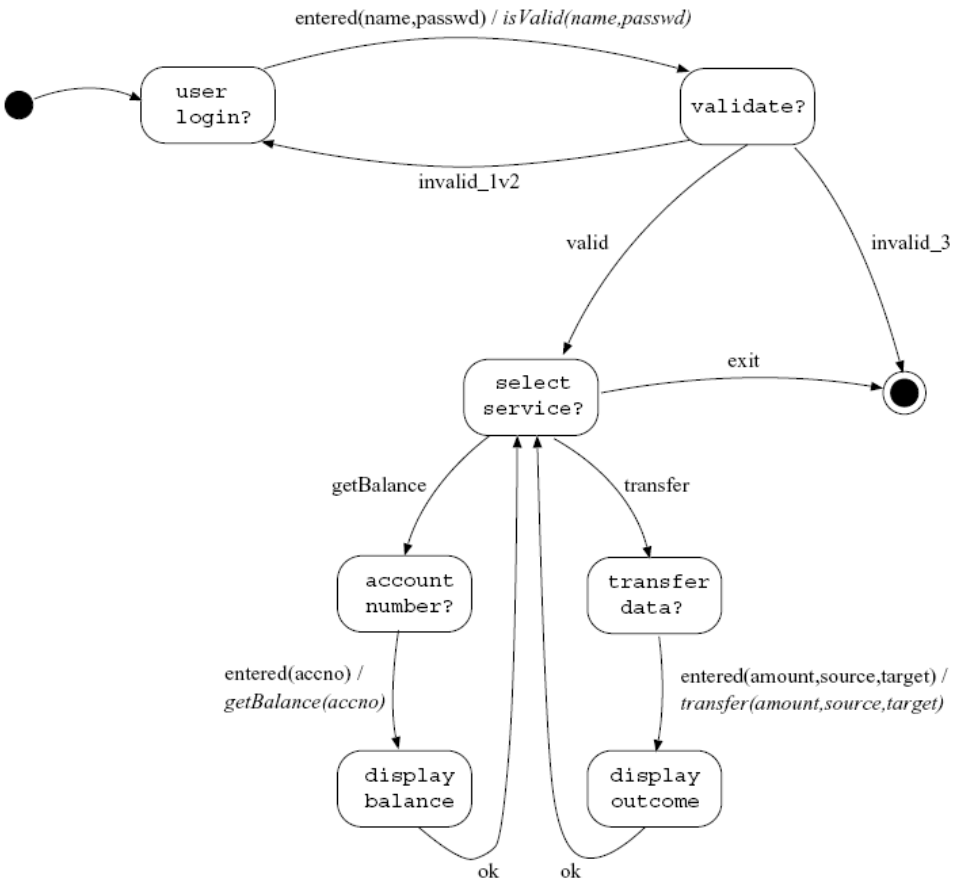

Figure 3.6: State transition system of the bank application

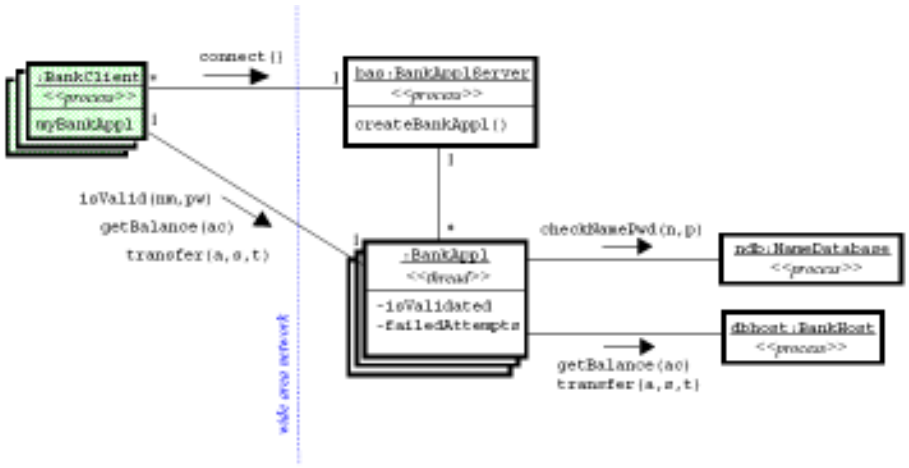

Figure 3.7: Process structure of the bank application

Die dynamische Struktur wird in verschiedenen Abstraktionsebene dargestellt und unter verschiedenen Perspektiven.

Statischer View eines Browsers:

Das Applikation-System benötigt zwei Clients. Einmal das auf der Eingabeseite des Benutzers und auf der Web-Seite als Plattform für den Webserver. Beide Clients sind durch Common Gateway Interfaces **CGI** verbunden.

Die Klassenstruktur zeigt die Implementierung von zwei Klassen.

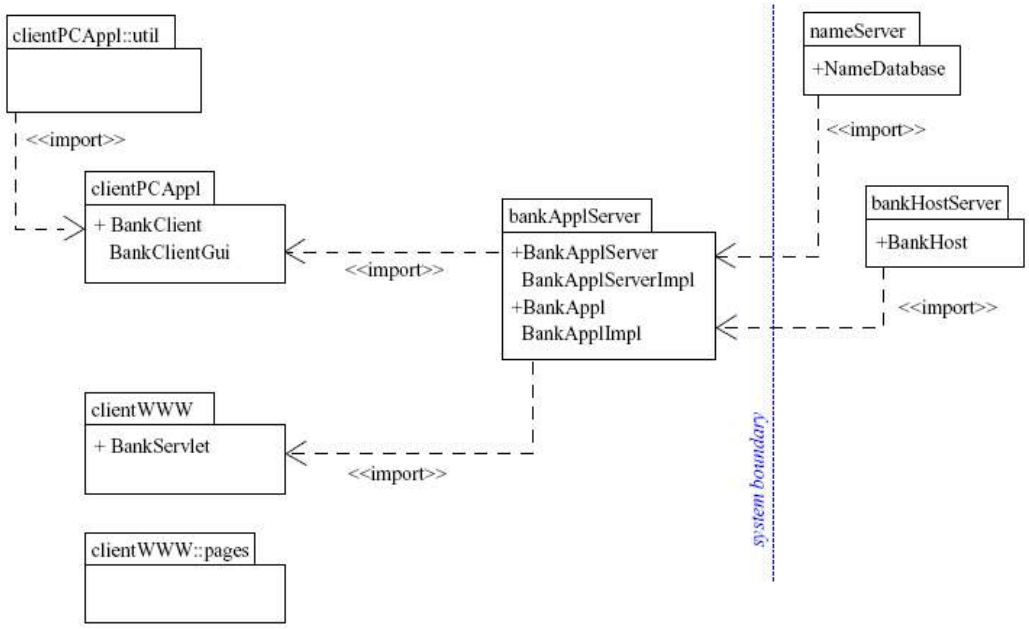

Figure 3.8: Package structure of the bank application

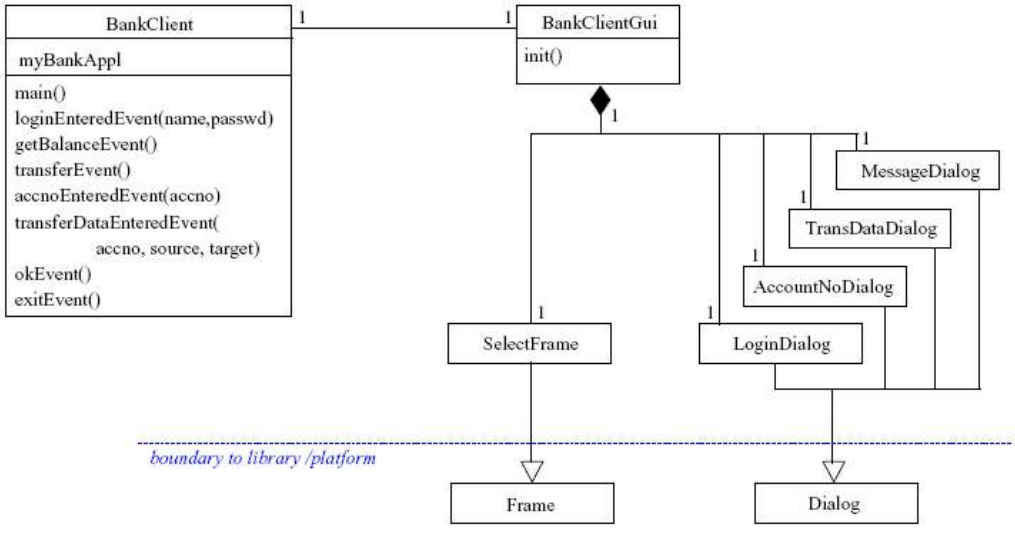

Figure 3.9: Class structure of one of the bank clients

Es gibt keine klaren Grenzen zwischen dynamischen und statischen View. Ein mischen von beiden kann ein System kompakter und aussagekräftiger beschreiben.

Aspekte eines Softwaresystems ist die Beschreibung des System als ganzes. Es gibt funktionale und nicht-funktionale Aspekte.
Funktionale Aspekte sind das korrekte Verhalten des Systems (z.B. Deadlockfreiheit)
Nicht-funktionale Aspekte sind: Performance, Sicherheit (Zugang nur für autorisierte Benutzer), Verfügbarkeit, Zuverlässigkeit, Erweiterbarkeit, Wiederverwendbarkeit (auf neuen Plattformen), Portierbarkeit, leicht zu erlernen.
Auch hier ist die Grenze nicht scharf. Im Grenzbereich befinden sich Fehlerbehandlung, Performanz, Garbagecollction.

Verfügbarkeit. Ein Prozess ist in der Zeit T r% verfügbar mit r/100

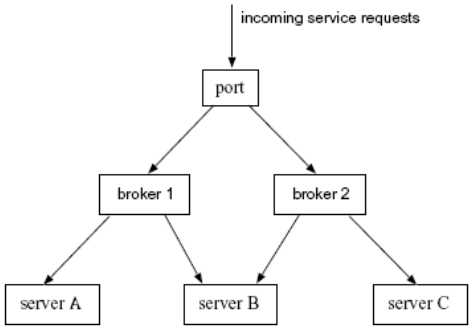

Figure 3.10: A simple system with a port, brokers, and servers

Based on the availability of the components, we compute the availability of the service of interest. We assume that the port is always available, that brokers are 70% and servers are 80% available and that the availability of a process is independent of the other processes. It follows that no broker is available 9% of the time, broker 1 alone is available 21%, broker 2 alone is available 21%, and both brokers are available 49%. As the service of interest is only provided by server A and B, it is not available at 16%:

$$9\% + 21\% * 4\% + 21\% * 20\% + 49\% * 4\% = 16\%$$

4. Architekturmuster

Architekturmuster zeigen die Struktur eines dynamischen Views. Die Muster können verschiedene Aspekte oder Eigenschaften darstellen. Sie sind nicht fest und können in unterschiedlichen Instanzen der Muster sein.
Die Komponenten der Muster sind Teile der Architektur. Sie können aktive oder passive Komponenten sein. Es gibt folgende Arten von Komponenten:

- Computational Component (führt spezielle Berechnungen aus, ist nicht persistent)
- Memory Component (speichert die Daten und ist für den Zugriff verantwortlich)

- Sensor Component (mist physikalische Größen z.B. Temperatur und speichert sie in Variablen)
- Prozess-Contol (Kontolliert physikalischen Prozess oder Variablen)

Darstellung von Mustern: ausgefüllte Pfeile für synchrone Kommunikation, offene Pfeile für asynchrone Kommunikation, gestrichelte Linien für getriggerte Aktionen.

Layerarchitektur. Jeder Layer stellt Services für die übergeordneten Layer zur Verfügung und benutzt Services der untergeordneten Layer. Ein Layer kommuniziert nur mit einem Layer direkt unter ihm. Die Systemumgebung hat nur Zugriff auf den obersten Layer, alle anderen sind versteckt.

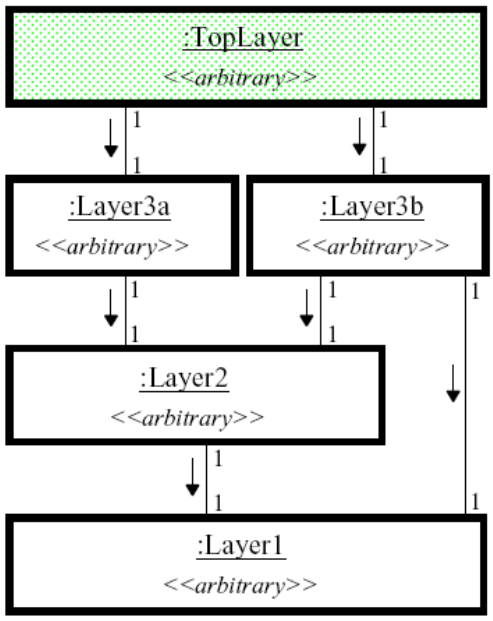

Figure 4.2: A layered system

Beispiele:

- OS mit folgenden Layern: Hardware, Systemkern, Systemservices (Calls), Applicationlayer
- Netzwerksysteme mit Protokollstacks auf jeder Seite der Verbindung

Layerd Systems können eine sehr stark abstrahierte Darstellung haben. Dadurch können sehr komplexe Systeme schrittweise dargestellt werden.
Sie unterstütz die Erweiterung, da sie nur mit benachbarten Layern kommunizieren.
Sie unterstützen Wiederverwendung, da sie klare Schnittstellen haben.

Die **repositorie Architektur** hat eine zentrale Speicherkomponente (repository), die von mehreren Clients benutzt wird.

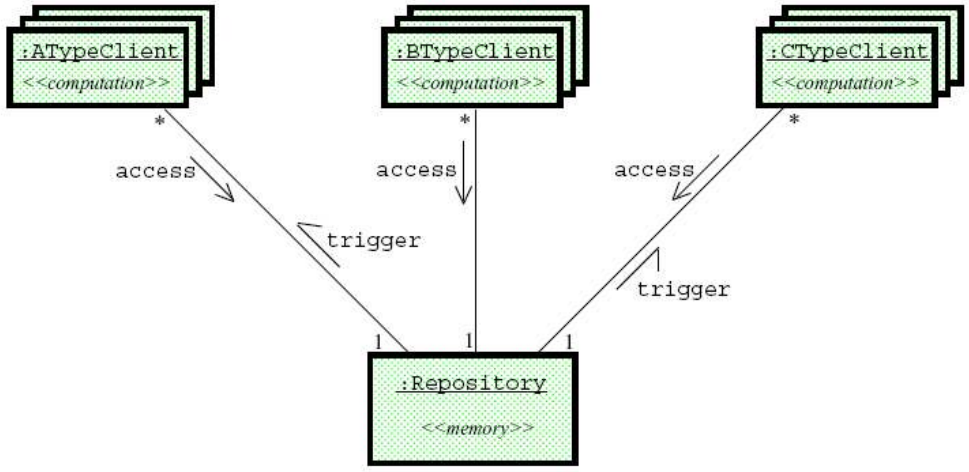

Figure 4.3: A repository system

Ein Datenbanksystem hat eine Repositoryarchitektur. Die Clients können Daten auf dem Server Lesen und Schreiben.

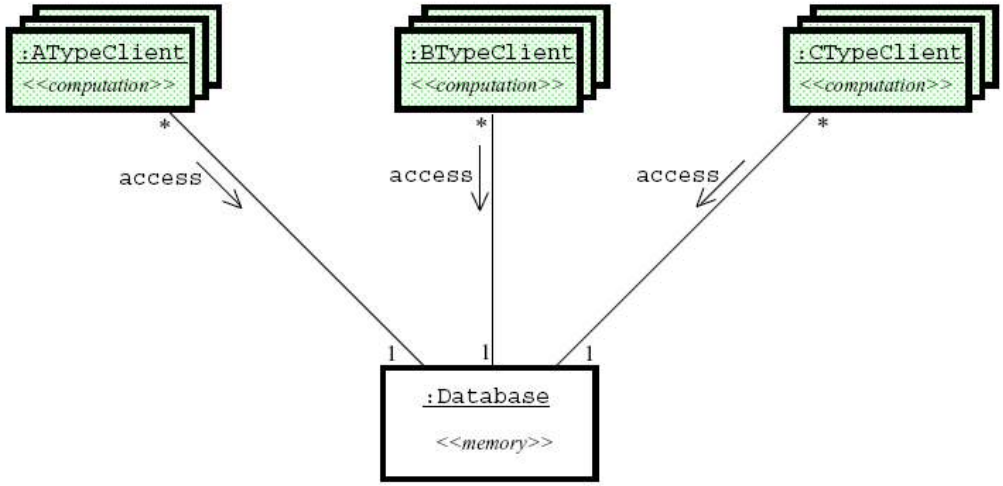

Figure 4.4: A database system

Dem **Compound-Object-Muster** liegt die objektorientierte Programmierung zu Grunde. Eine Komponente kann Aktivitäten triggern, Berechnungen ausführen oder einen Teil des Systemstatus festhalten. Eine Komponente besteht aus mehreren kleinen Komponenten, die gekapselt sind. Sie verständigen sich durch Nachrichten. Jede Architektur mit definierten Schnittstellen und gekapselten Daten kann in einem Compound-Object-Muster dargestellt werden.

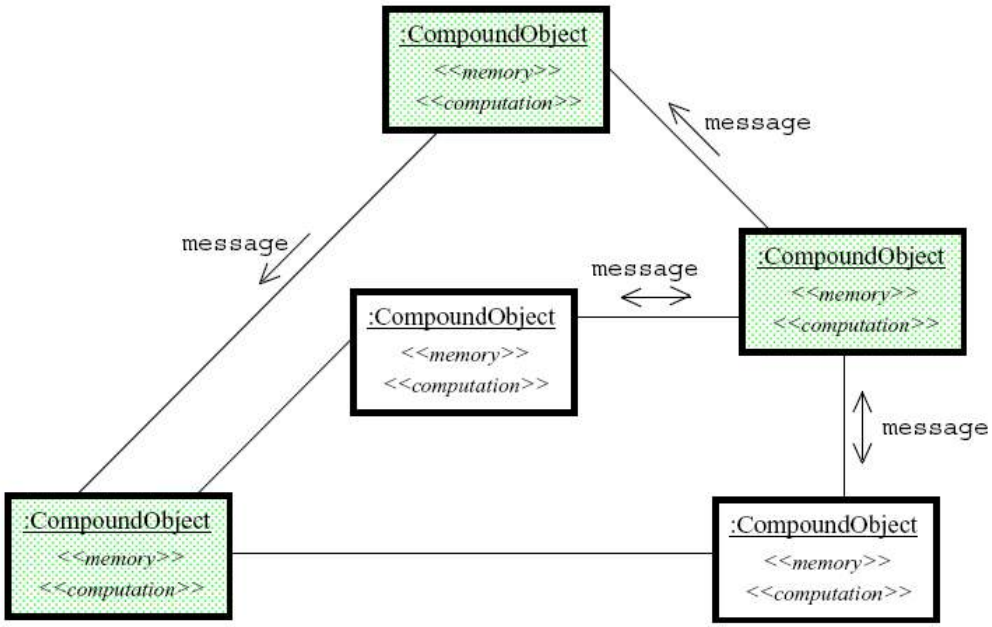

Figure 4.6: A system of compound objects

Pipes und Filter. Bei diesem Muster gibt es zwei unterschiedliche Komponenten. Die Pipe ist die inaktive Datenquelle. Filter sind die aktiven Objekte. Sie lesen Daten von den Inputpipes und schreiben sie in die Outputpipes.

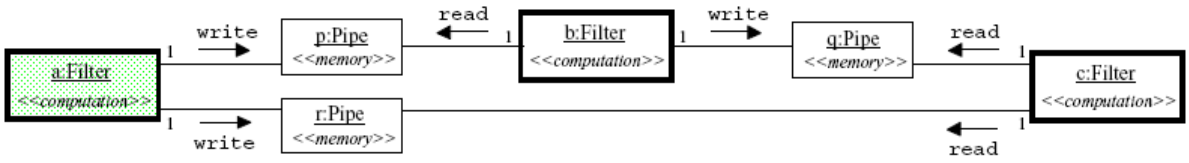

Figure 4.7: System of pipes and filters

Bei Unix gibt es Pipelines, die zwischen den Prozessen Daten austauschen.

Die bisher genannten Muster beschreiben die Organisation von Softwaresystemen. Prozess-Control-Muster entsprechen einem integrierten System. Es werden bestimmte Variablen (Prozessvariablen) überwacht (Benutzereingaben) und entsprechende Aktionen ausgelöst. Es gibt folgende Variablen: Inputvariablen, Kontrollvariablen und manipulierbare Variablen (durch sie wird ein Prozess gesteuert).

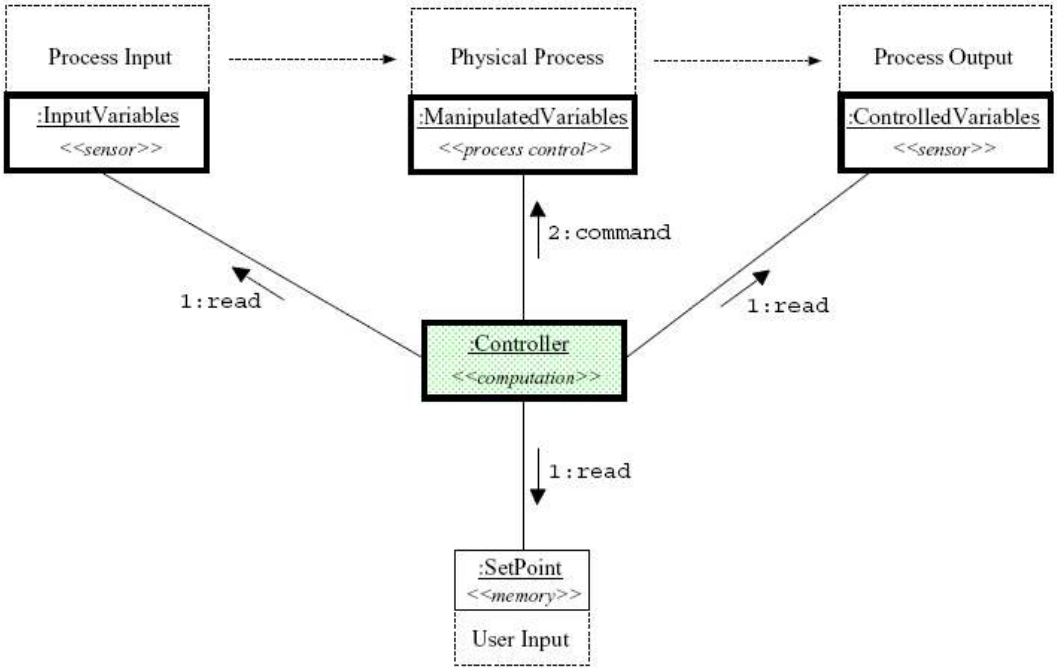

Figure 4.8: Process-control pattern

5. Architekturmuster

Programm-Framework ist eine wichtige Technik bei der Wiederverwendung von Software. Das Framework ist in verschiedenen Sprachen geschrieben. Elemente des Frameworks sind Prozeduren, Module, Klassen, Interfaces oder einfache Programmelemente.

Ein Framework sieht wie eine Sammlung von Klassenbibliotheken aus. Aber nicht jede Sammlung von Bibliotheken ist ein Framework. Die Klassen eines Frameworks haben die gleichen Core-Funktionen, sie arbeiten geschlossen zusammen.
Ein Framework basiert oft aus ein bestimmtes Muster. GUIs sind oft mit dem Model-Viewer-Controller-Muster erstellt.
Frameworks haben zwei Nachteile: die einzelnen Komponenten sind sichtbar (Whitebox) und sie sind an die Mechanismen und Limits der zugrunde liegenden Programmiersprache gebunden.

Bei den **Whiteboxes** greifen die Programmierer auf interne Strukturen zu. Damit ist die Weiterentwicklung dieser Klassen schwierig.
Es müssen klare Schnittstellen geschaffen werden (**Graybox**).

Das Gegenstück ist die **Blackbox**. Sie ist vollständig gekapselt. Der Benutzer weiß nichts über die implementierten Komponenten.

Architektureigenschaften von GUIs: Sie sollen eine benutzerfreundliche Interaktion mit den System ermöglichen.

- die Kontrolle der Applikation soll ermöglicht werden
- Daten sollen eingeben werden können
- und der Status soll grafisch dargestellt werden.

MVC-Muster (Modell-View-Controller) bei GUIs
Modell – die zugrunde liegende Applikation
View – repräsentiert die Grafik des GUI
Controller – steuert die Interaktion zwischen Benutzer und Modell
MVC-Muster sind eventbasierent.

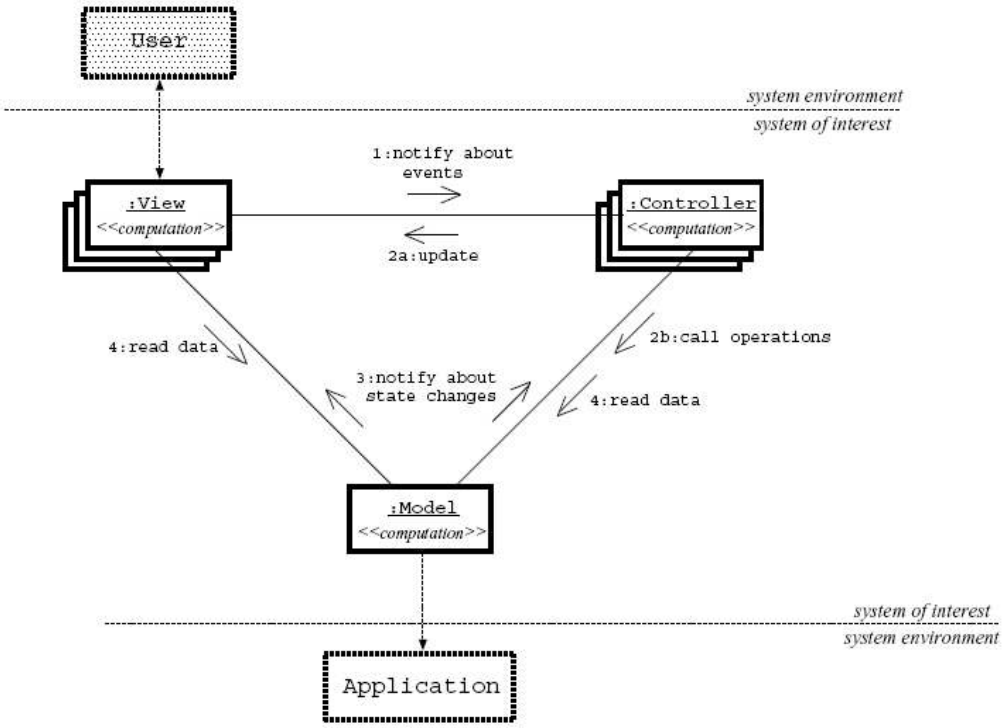

Figure 5.1: MVC-pattern

AWT – Abstract Window Toolkit für Java:

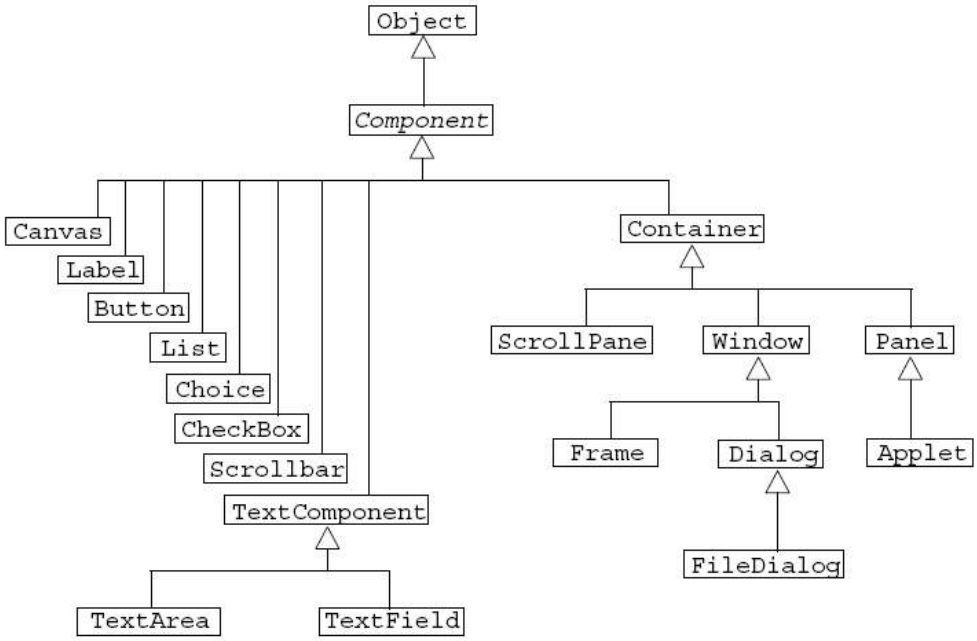

Figure 5.2: Component class hierarchy of the AWT

In summary, a program framework like the AWT can fix some architectural decisions (e.g. the base architecture for the construction of views or the event-handling mechanism) and leave other aspects of architectural design open (e.g. how to separate views and application kernel). Whereas classical GUI toolkits are fairly weak w.r.t. higher-level architectural support, several attempts have been made to design frameworks that support more powerful architectural patterns, in particular patterns for whole application systems. An example is the ET++ application framework that is described in the companion paper of this course unit. In such frameworks, the architectural reuse can achieve the same importance as the reuse of code.

6. Architektur von Komponentensoftware

Das Compound-Object-Muster ist die Grundlage für das Framework. Die Komponenten sind die Basis des Frameworks. Sie sind oft Blackboxen mit fest vorgegebenen Schnittstellen.

Die Komponenten des Frameworks müssen in der Software zusammengebracht werden. Dafür gibt es verschiedene Methoden:

- static linking (die Komponenten werden vor der Programmausführung verknüpft)
- dynamic linking (die Komponenten werden während der Laufzeit verknüpft)

- Connecting (die Komponenten laufen als Teile unterschiedlicher Prozesse, es gibt eine spezielle Objektreferenz, die die Kommunikation per Remote ermöglicht)

Diese Methoden können auch kombiniert werden.

Ein **Framework** besteht aus Regeln und Schnittstellen. **Komponenten** bestehen aus Programmen und Daten. Sie sind in einer bestimmten Sprache geschrieben.
Frameworks können unabhängig von der Plattform sein und bestimmte Muster unterstützen (EJB – Three-Tier-Architektur).

Microsoft's Component Object Model (COM). COM ist für verschiednen Betriebssysteme verfügbar. Es gibt Schnittstellen zu weiteren Frameworks z.B. OLE. Durch das Framework werden Systeme im Binärcode erzeugt, d.h. die Komponenten können in verschiedenen Sprachen geschrieben sein.

Die Komponenten heißen im COM Klassen. sie bestehen aus COM-Objekten. Das COM-Objekt stellt seine Services über verschiedene Schnittstellen zur Verfügung. Diese Schnittstellen haben Namen. Eine Konvention sagt, dass sie immer mit „I" beginnen.

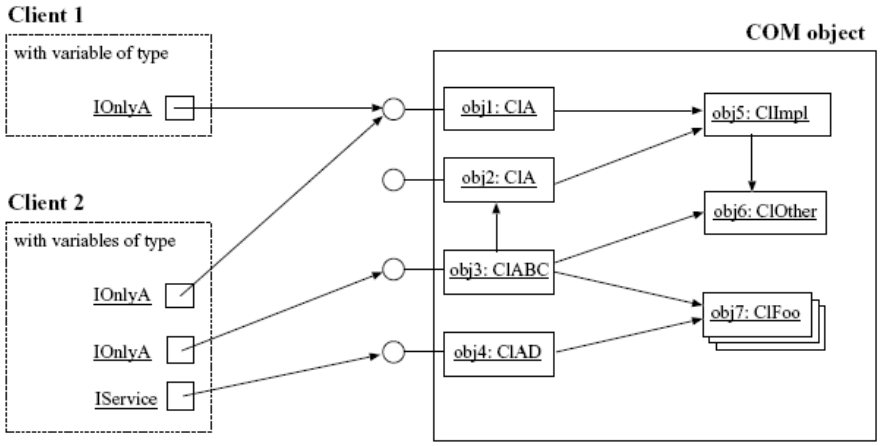

Figure 6.1: COM object as a collection of pure objects

Für die Kommunikation gibt es Interface Identifier (IID) – 128bit Nummern. Es gibt Server zum Erzeugen von COM-Objekten (in-process-server, lokal-server, remote-server).

Enterprise JavaBeans (EJB). Es wird die TT-Architektur unterstützt. Man unterscheidet zwischen Session Beans und Entity Beans. Sie haben immer ein Home- und Remote-Interface.

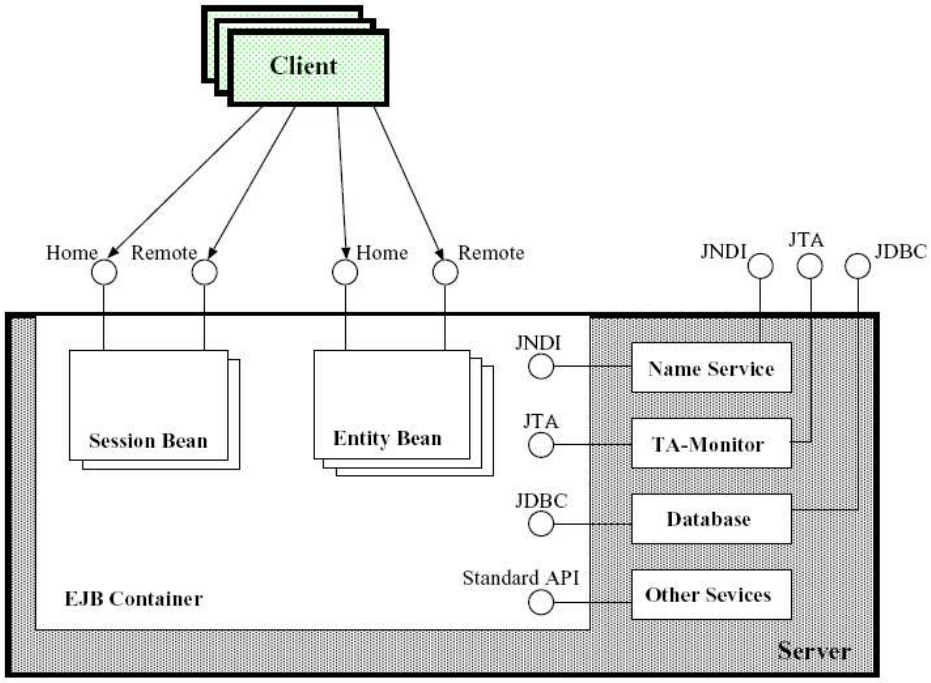

Figure 6.2: The principal architecture of an EJB based system

EJB sind Application-Assembler-Constructs einer serverseitigen Anwendung. Es wird ein EJB-Container erzeugt.

Die Beans sind Teile des EJB Framework. Es gibt bean instancen, die zu einem Laufzeit-Objekt zeigen und bean components, die ihre Implementierung referenzieren.

Entity Beans repräsentieren Daten-Objekte, die vom System benutzt werden. Der Status wird in einer Datenbank gespeichert.
Session Beans werden benutzt, um Tasks oder Prozesse darzustellen. Ein Session Bean steht für einen Service des Clients.

Eine Komponente besteht aus vier oder fünf Teilen:

- Remote Interface – enthält die Methoden, die Verbindung zum Client herstellen
- Home Interface – definiert den Lebenszyklus der Komponente, diese Interface wird benutzt um Instanzen zu erzeugen
- Primery Key Class – behört zu jedem Bean, wird als Referenz beim Remote Interface benutzt
- Bean Class – beinhaltet die Implementierung des Home- und Remote-Interfaces

- Deployment Descriptor – Verwaltet Informationen, die in anderen Teilen nicht gespeichert werden. Hierüber können verschiedene Beans zu einer Einheit verbunden werden. Geschrieben im XML-Format

```
public void changeBalance(float amount) throws RemoteException
  accountBalance += amount;
}

// the methods of interface javax.ejb.EntityBean
public void setEntityContext(EntityContext ctx)
                             throws RemoteException {
  theContext = ctx;
}

public void unsetEntityContext() throws RemoteException {
  theContext =  null;
}

public void ejbRemove()
          throws RemoteException, RemoveException {}
public void ejbActivate()  throws RemoteException {}
public void ejbPassivate() throws RemoteException {}
public void ejbLoad()      throws RemoteException {}
public void ejbStore()     throws RemoteException {}
```

Figure 6.3: The implementation of bean class BankAccountBean

Die Grundlage der EJB Komponentenstruktur ist der EJB-Container. Eine Bean-Instanze kann es nur in einem Container geben. Er ist die Laufzeitumgebung der Beans und stellt alle Services zur Verfügung.

Managing Beans – die Laufzeit der Instanzen muss vom Container verwaltet werden.
persistencd, transactions, security

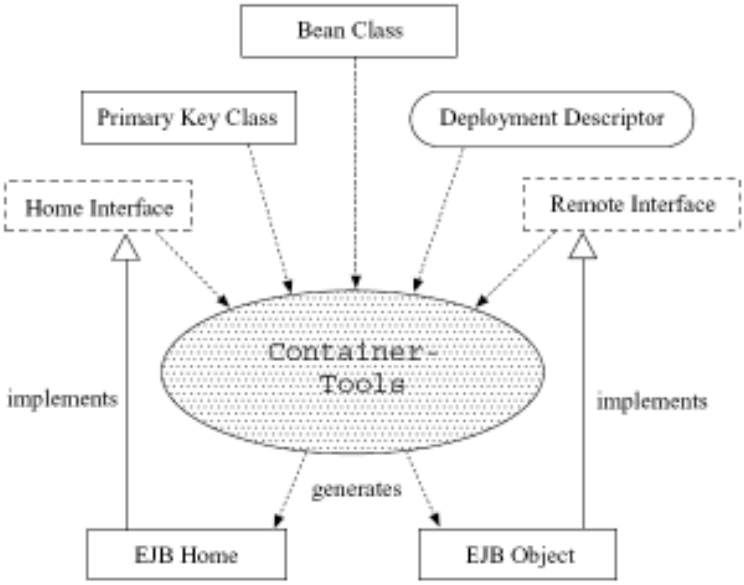

Figure 6.6: Generation of EJB home and remote class

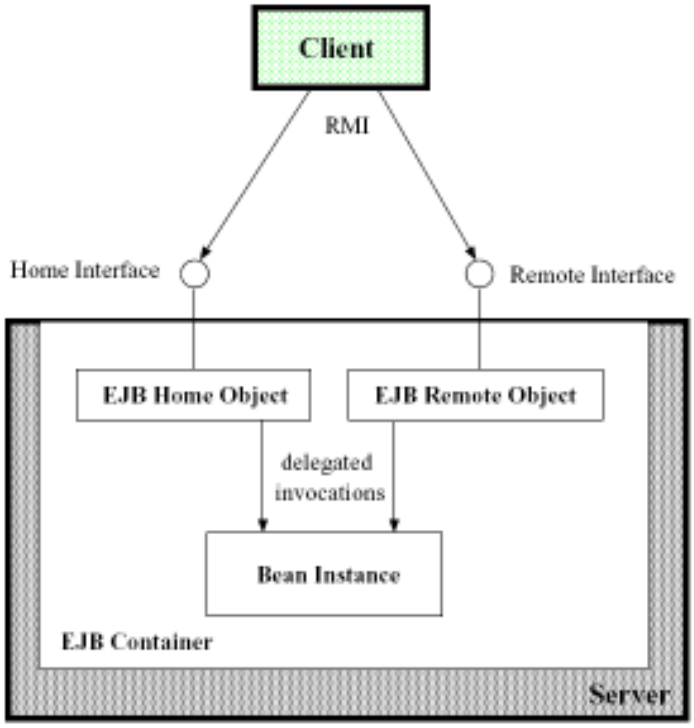

Figure 6.7: Bean access via EJB home and remote object

7. Techniken der Architektur

Das Ergebnis der Design-Phase ist die Architektur der Software. Sie muss

- den richtigen Level an Abstraktion haben
- Standardisiert sein, damit sich Benutzer über sie unterhalten können
- und auf bestimmte Techniken basieren, damit die Konsistenz geprüft werden kann, sie Visualisiert werden kann und Code generiert werden kann.

Es gibt verschiedenen Möglichkeiten Architekturen zu beschreiben (darzustellen) – ADL – Architecture Description Languages.

Die **Unified Mdeling Language (UML)** bietet Notationen für objektorientierte Analyse und Design. Es gibt verschiedene Darstellungen mit UML.
Für statische Views gibt es Class Diagrams, Implementation Diagrams (component oder deployment).
Und für dynamische Views: Use Case Diagrams, Statechart Diagrams, Activity Diagrams, Interaction Diagram, Sequence Diagram.

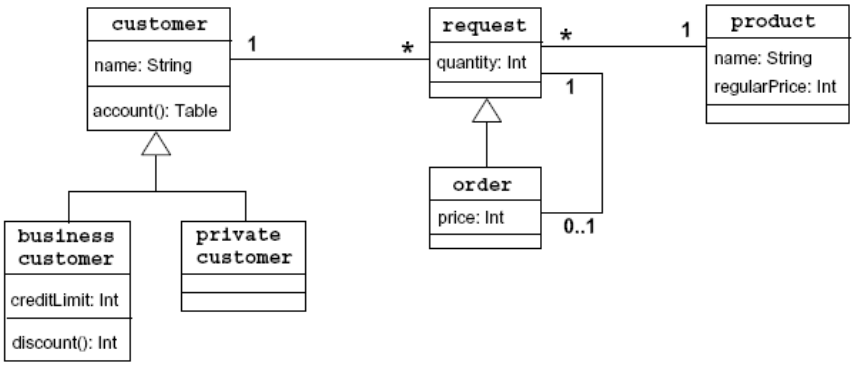

Figure 7.1: A class diagram of the order processing system

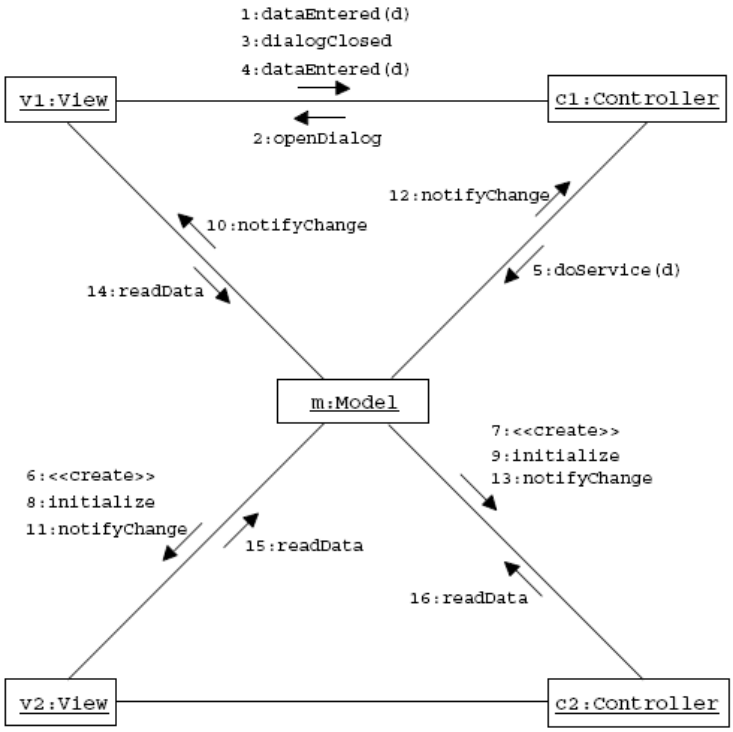

Figure 7.2: A collaboration diagram describing an MVC-scenario

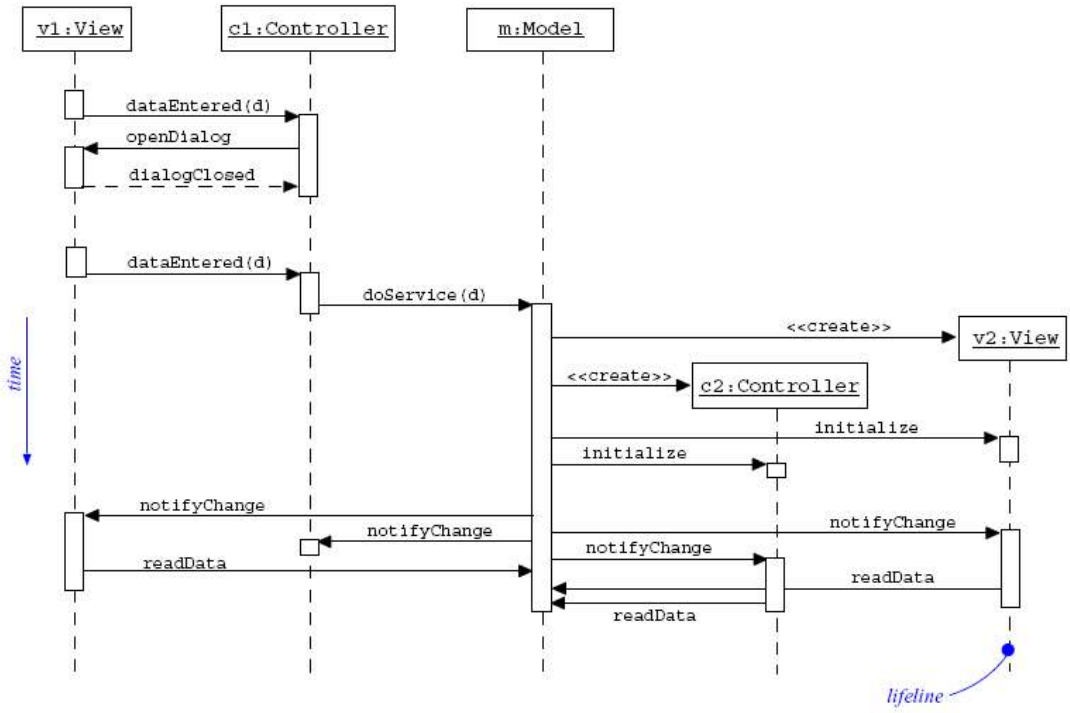

Figure 7.3: A sequence diagram describing an MVC-scenario

Das gleiche System wird unterschiedlich dargestellt.

Architectural Frameworks sollen:

- eine Sammlung von Sprachen zur Beschreibung der de Architektur mit einer guten Semantik sein, die verschiedenen Notationen (grafisch, textorientiert) hat.
- Sie muss für jede Sprache eine Sprache eine Sammlung von Styles, Regel und Anweisungen haben, aus denen man die Befehle bildet.
- es muss ein Definition geben, was ein Element einer Architektur-Beschreibung ist.

Das **TAF Architectural Framework** stellt Softwaresysteme durch gekapselte Komponenten dar. Dafür wird eine grafische Notation verwendet.

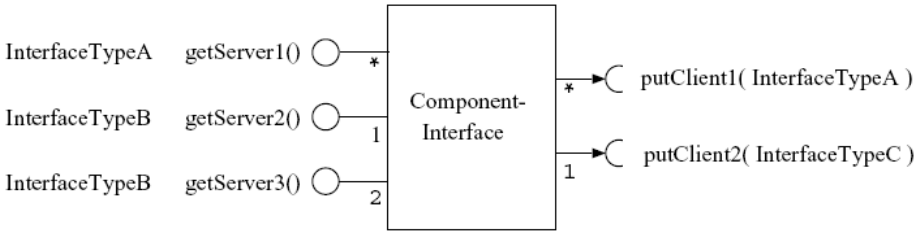

Figure 7.4: The graphical representation of an TAF-component

Für die Beschreibung gibt es folgende Regeln:

- Zur Beschreibung dient GNOTAF (grafical notation of TAF). Die Komponenten können ähnlich wie das Java-Modell beschrieben werden.
- Topologie: Es gibt eine Anzahl von Instanzen mit Namen und einer Beschreibung für die Verknüpfung.

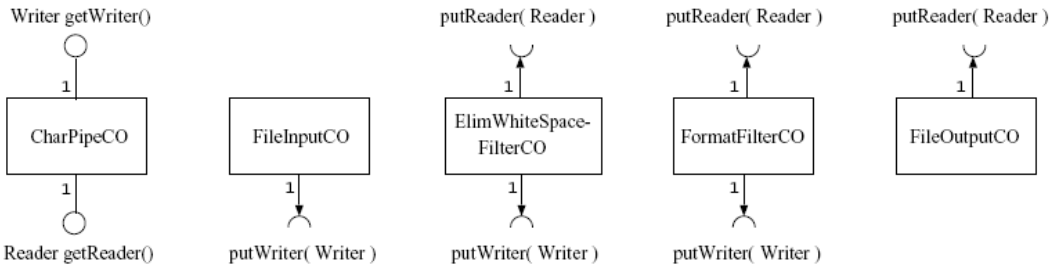

Figure 7.5: The interfaces of formatter architecture

8. Entwickeln einer Architektur

Die Erstellung einer Design zählt zur Entwicklungsphase. Auf dessen Grundlage wird das Softwaresystem entwickelt.
Durch das Design sollen

- die Schwächen der Anforderungen der Software erkannt werden
- Subsysteme definiert werden (eine Zerlegung)
- funktionale und nicht-funktionale Anforderungen gezeigt werden
- die Wiederverwendung vorhandener Module unterstützt werden

Ziel des Software Engineering ist die Entwicklung, Installation und Wartung von Software zu strukturieren, systematisieren und teilweise automatisieren.
Functional Design basieren auf einem statischen View, die technische Struktur ist vorrangig
Object-Oriented Design baut auf objektorientierte Muster auf.
Das **extreme Programming** integriert die Anforderungsanalyse, das Design, die Implementierung und das Testen.

Regeln für die Architektur:

1. The architecture should feature well-defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concerns. Each module should have a well-defined interface that encapsulates or ”hides” changeable aspects (such

as implementation strategies and data-structure choices) from other software that uses its facilities.

2. The modules should reflect a separation of concerns that allows their respective development teams to work largely independently of each other.
3. The information-hiding modules should include those that encapsulate idiosyncrasies of the computing infrastructure, thus insulating the bulk of the software from change should the platform change.
4. The architecture should never depend upon a particular version of a commercial product or tool. If it depends upon a particular commercial product, it should be structured such that changing to a different product is straight-forward and inexpensive.
5. Modules that produce data should be separate from modules that consume data. This tends to increase modifiability because changes are often confined to either the production or consumption side of data. If new data is added, both sides will have to change, but the separation allows for a staged (incremental) upgrade.
6. For parallel-processing systems, the architecture should feature welldefined processes or tasks that do not necessarily mirror the module structure. That is, processesmay thread throughmore than onemodule; a module may include procedures that are invoked as part of more than one process.
7. Every task or process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
8. The architecture should feature a small number of simple interaction patterns. That is, the system should do the same things in the same way throughout. This will aid in understandability, reduce development time, increase reliability, and enhance modifiability. It will also show conceptual integrity in the architecture, which while not measurable, leads to smooth development.