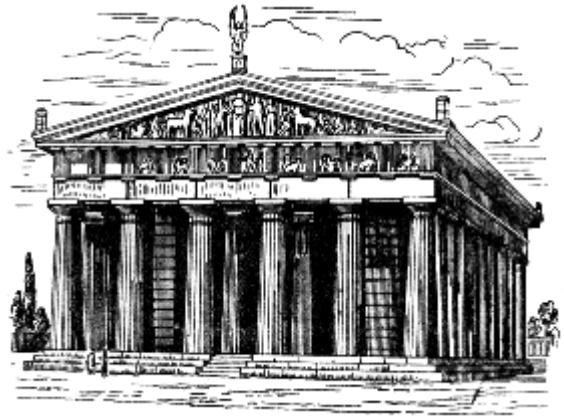
Softwarearchitektur

(Architektur: *αρχή* = Anfang, Ursprung + *tectum* = Haus, Dach)

6b. Ausprägungen und Wiederverwendung



Vorlesung Wintersemester 2005 / 2006

Technische Universität München
Institut für Informatik
Lehrstuhl von Prof. Dr. Manfred Broy

Dr. Klaus Bergner, Prof. Dr. Manfred Broy, Dr. Marc Sihling

Inhalt

- Umsetzung: Komponenteninfrastrukturen
  - Motivation : ein einfaches Beispiel
  - Leichtgewichtige Containerarchitekturen : Spring
  - Schwergewichtige Containerarchitekturen : EJB
- Wiederverwendung durch Muster
  - Designmuster
  - Architekturmuster
  - Analysemuster
- Referenzarchitekturen
- Produktlinienarchitekturen
- Zusammenfassung

Inhalt

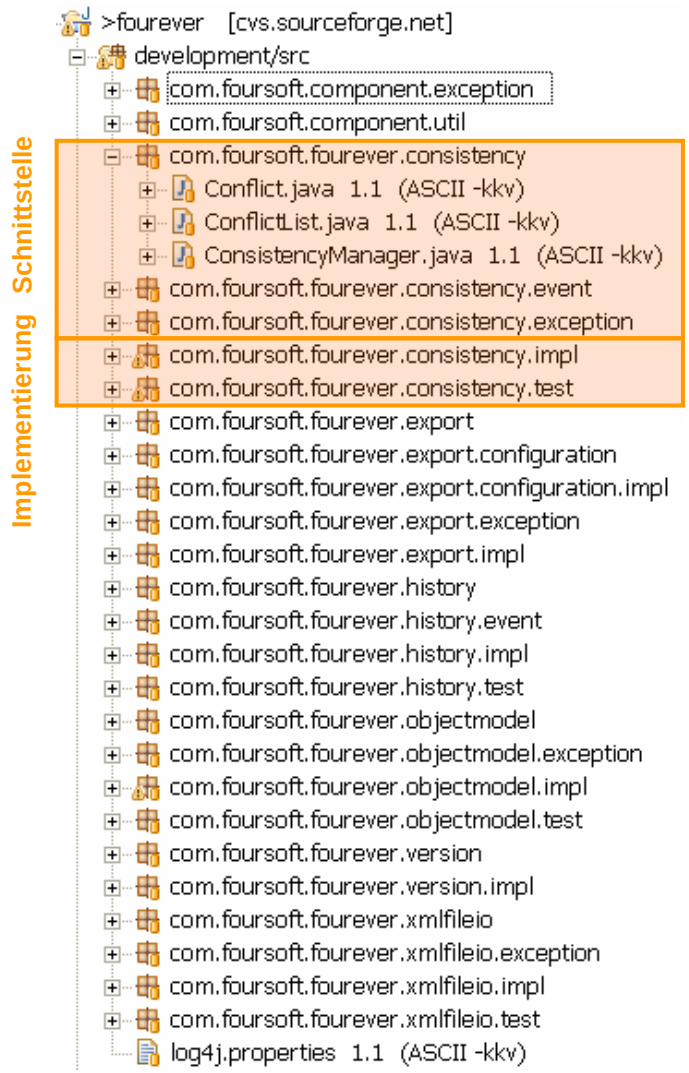
- Umsetzung: Komponenteninfrastrukturen
  - Motivation : ein einfaches Beispiel
  - Leichtgewichtige Containerarchitekturen : Spring
  - Schwergewichtige Containerarchitekturen : EJB
- Wiederverwendung durch Muster
  - Designmuster
  - Architekturmuster
  - Analysemuster
- Referenzarchitekturen
- Produktlinienarchitekturen
- Zusammenfassung

Komponenteninfrastruktur

- Ziel: die durch die Softwarearchitektur vorgegebenen Richtlinien werden durch die Infrastruktur forciert:
  - Komponenten kommunizieren ausschließlich über ihre Schnittstellen
  - Das „Ausspionieren“ der Implementierung einer Komponente ist technisch wirksam verhindert
  - Abhängigkeiten zur Laufzeit entsprechen den Designvorgaben
  - Verschaltung der Komponenten flexibel konfigurierbar
- Einfache Lösung: bereits mit „Hausmitteln“ können Komponenten auf objektorientierte Sprachen abgebildet werden
- Komfortable, aber aufwändige Lösung: professionelle, serverseitige Komponenteninfrastrukturen kümmern sich um Verwaltung („Life-Cycle“), Verschaltung und technische Belange von Komponenten

Einfache Lösung: Komponenten mit Spring und POJOs

- Grundlage sind einfache Java-Objekte (Plain Old Java Objects), deren externe Abhängigkeiten (Resourcen, etc.) vom Spring-Framework zugewiesen werden („dependency injection“)
- Die Schnittstelle einer Komponente ist durch ein oder mehrere Java-Schnittstellen gegeben, die sich untereinander sowie Schnittstellen abhängiger Komponenten kennen
- Jede Komponente hat einen Manager, der als Torwächter für die Kommunikation der Komponente zu ihrer Außenwelt und für die Kommunikation der Außenwelt mit der Komponente dient.



Überprüfung der Abhängigkeiten

- Die Einhaltung der Richtlinien, welche Abhängigkeiten zwischen den Komponenten bestehen dürfen, kann auf zwei Arten erfolgen:
  - **Statisch**, beispielsweise über die Verwendung der JUnit-Erweiterung JDepend (http://www.clarkware.com/software/JDepend.html)
    - Greift Komponente A wirklich nur auf die Schnittstellen von Komponente B zu?
    - Existieren keine zyklischen Abhängigkeiten (gemäß Architektur)?
    - Sind Abhängigkeiten stets von instabilen zu stabilen Komponenten gerichtet? (und damit: wie stabil ist eine Komponente?)
    - Wie „gut“ ist das aktuelle Design?
  - **Dynamisch**, beispielsweise durch
    - Mehrere Jar-Files (Schnittstellen und Implementierung getrennt) und zudem
    - Mehrere Class-Loader (ein Casting von der Schnittstelle auf ihre Implementierung ist technisch unterbunden).

Der Komponentenmanager am Beispiel

```
public interface XMLFileIOManager {
    public void setConfig(XMLFileIOConfig
          xmlconfig);
    Document createDocument(File xsdfile)
          throws IOException;
    Iterator getDocuments();
    Document openDocument(File xmlfile);
    Document getDocument(ObjectModel om);
    boolean closeDocument(Document d);

    void register(DocumentObserver o,
          int aspect);
    void unregister(DocumentObserver o);
}

public class XMLFileIOManagerImpl implements
    XMLFileIOManager {
    public static XMLFileIOManagerImpl
          createInstance(Log givenLog,
          ObjectModelManager omManager) {
        if (instance != null) {
          throw new ComponentInternalException(
           "XMLFileIOManager already
            initialized");
        }
        // code snipped here
    }
}
```

Selbst der Komponentenmanager ist anderen Komponenten nur über eine Schnittstelle bekannt.

Dependency Injection nach Initialisierung der Komponente

Methoden zur Initialisierung (Factory-Pattern) und Verwaltung von Schnittstellenobjekten

Klassische Abhängigkeiten am Framework vorbei. Ok, da hier keine Lifecycle-Verwaltung durch das Framework (z.B. Passivierung bei Speicherengpaß)

Dependency Injection zum Zeitpunkt der Erzeugung der Komponente

Manager als Singleton realisiert (nicht transparent für andere Komponenten)

Verschaltung der Komponenten

Konfigurationsdateien dieser Art definieren in einem XML-Format deklarativ die Verschaltung der diversen Komponenten (also den „Glue“)

Komponenten werden über Factories erzeugt (hier nun endlich die Komp.-implementierung)

Erst nach Berücksichtigung aller Abhängigkeiten entscheidet Spring über die Reihenfolge der Initialisierung aller Komponenten

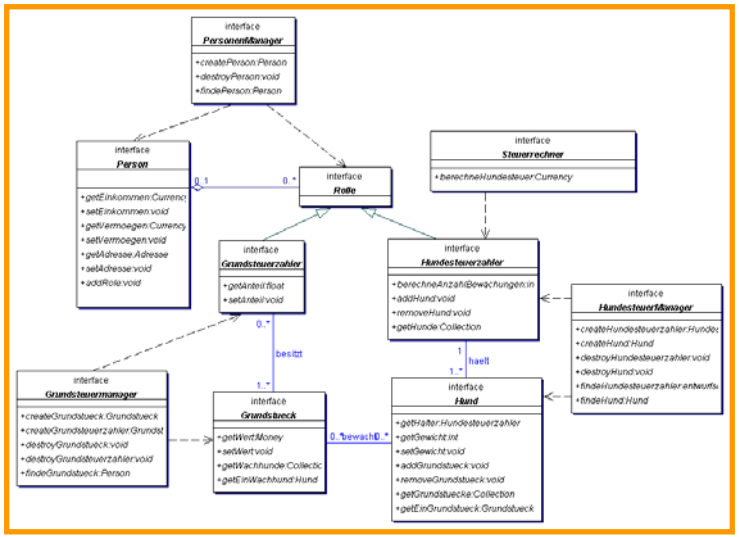
Spring stellt weiterhin einen „Application Context“ zur Verfügung, über den nach Komponenten gesucht werden kann („*dependency lookup*“)

```
<bean id="logfactory"
      class="org.apache.commons.logging.LogFactory"
      factory-method="getFactory">
</bean>
<bean id="objectmodelmanager">
    ...
</bean>
<bean id="xmlfileiolog" factory-bean="logfactory"
    factory-method="getInstance">
    <constructor-arg>
     <value>com.foursoft.fourever.xmlfileio</value>
    </constructor-arg>
</bean>
<bean id="xmlfileiomanager"
    class="com.foursoft.fourever.xmlfileio.impl.XML
           FileIOManagerImpl"
    factory-method="createInstance"
    destroy-method="destroy">
    <constructor-arg>
          <ref bean="xmlfileiolog"/>
    </constructor-arg>
    <constructor-arg>
          <ref bean="objectmodelmanager"/>
    </constructor-arg>
</bean>
```

Zwischenstand

- Mit einfachen Mitteln lässt sich in einer objektorientierten Sprache komponentenorientiert entwickeln
  - Strikte Trennung von Schnittstelle und Implementierung
  - Sehr direkte Umsetzung der fachlichen Sicht möglich
  - Nutzung von Spring zur Konfiguration und Verschaltung von Komponenten (Stichwort: „dependency injection“)
  - Nutzung von *Aspect-oriented Programming* für Logging und Tracing (nicht gezeigt)
  - Evtl. Generierung von abstrakten Basisklassen (nicht gezeigt)
- Jedoch: je mehr zusätzliche Technik in die Komponenten eingefügt werden soll, desto komplexer das Ergebnis.
- Zielsetzung: Technik als Dienstleistungen in die Komponenteninfrastruktur „auslagern“ und damit die Implementierung der Komponenten möglichst einfach halten.

Fachliche und technische Sichten



Persistenz

Transaktionsmanagement

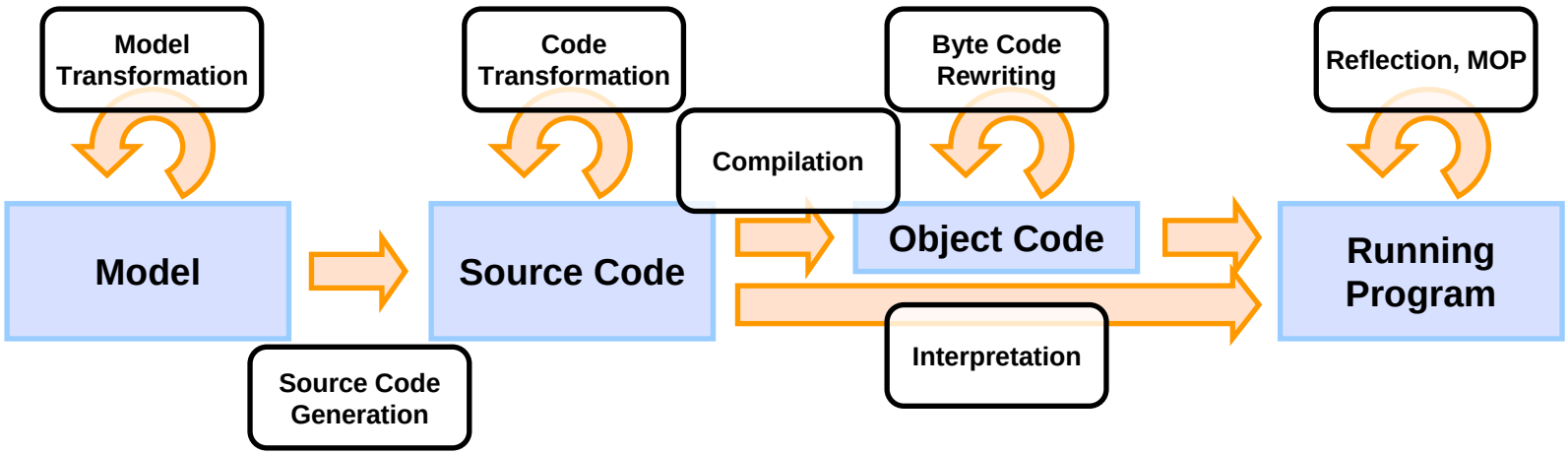
Lifecycle-Management

Security

…

- Die Realisierung erfordert das Zusammenführen der fachlichen mit den diversen technischen Sichten (meist „*cross-cutting concerns*“)
- Jede technische Sicht impliziert eine eigene Modellierung (z.B. technische Schnittstellen)
- Bei der Zusammenführung werden die technischen Sichten der fachlichen Sicht „aufgeprägt“: Komponenten müssen neben der fachlichen noch einige technische Schnittstellen umsetzen
- In manchen Fällen kann die fachliche Sicht nicht 1:1 übernommen werden – technische Gründe machen eine andere Modellierung notwendig
- Die Einbindung technischer Sichten kann umso einfacher automatisiert werden, je unabhängiger die Sicht von anderen Sichten ist (siehe AOP in der letzten Stunde) – Stichwort *Orthogonalität*

Verknüpfung durch Generierung



- Generator-Ansätze sind bewährt, um die Wiederverwendung übergreifender, technischer Aspekte zu ermöglichen. Hierbei sind außerdem Anpassungen im Hinblick auf die Zielplattform möglich.
- Bereits bekannt aus vergangenen Stunden
  - AOP: Aspect Oriented Programming
  - MDSD: Model Driven Software Development (konkret: MDA der OMG)
  - Generierung von Code-Templates für verteilte CORBA-Objekte

Verknüpfung durch Konfiguration

- Transaktionssteuerung bei einem Anwendungsserver
  - Component-Managed Transactions: Die fachliche Komponente erzeugt Transaktionen und ruft `begin`, `commit` und `abort` auf ihnen auf.
  - Container-Managed Transactions: Der Application Server bestimmt, wann eine Transaktion geöffnet und beendet wird.
- Beispiel: Ausschnitt aus Konfigurationsdatei für deklarative Transaktionsverwaltung im Spring Application Framework

```
<bean id=„trx_hundesteuermanager"
   class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
   <property name="transactionManager" ref="txManager"/>
   <property name="target" ref=„hundesteuermanager"/>
   <property name="transactionAttributes"><props>
      <prop key="create*">PROPAGATION_REQUIRED</prop>
      <prop key=„finde*">PROPAGATION_REQUIRED,readOnly</prop>
   </props></property>
</bean>
```

Spring-Mechanismus

Verwaltete Komponente

Transaktionsattribute für Operationen (propagation, isolation, read-only, timeout)

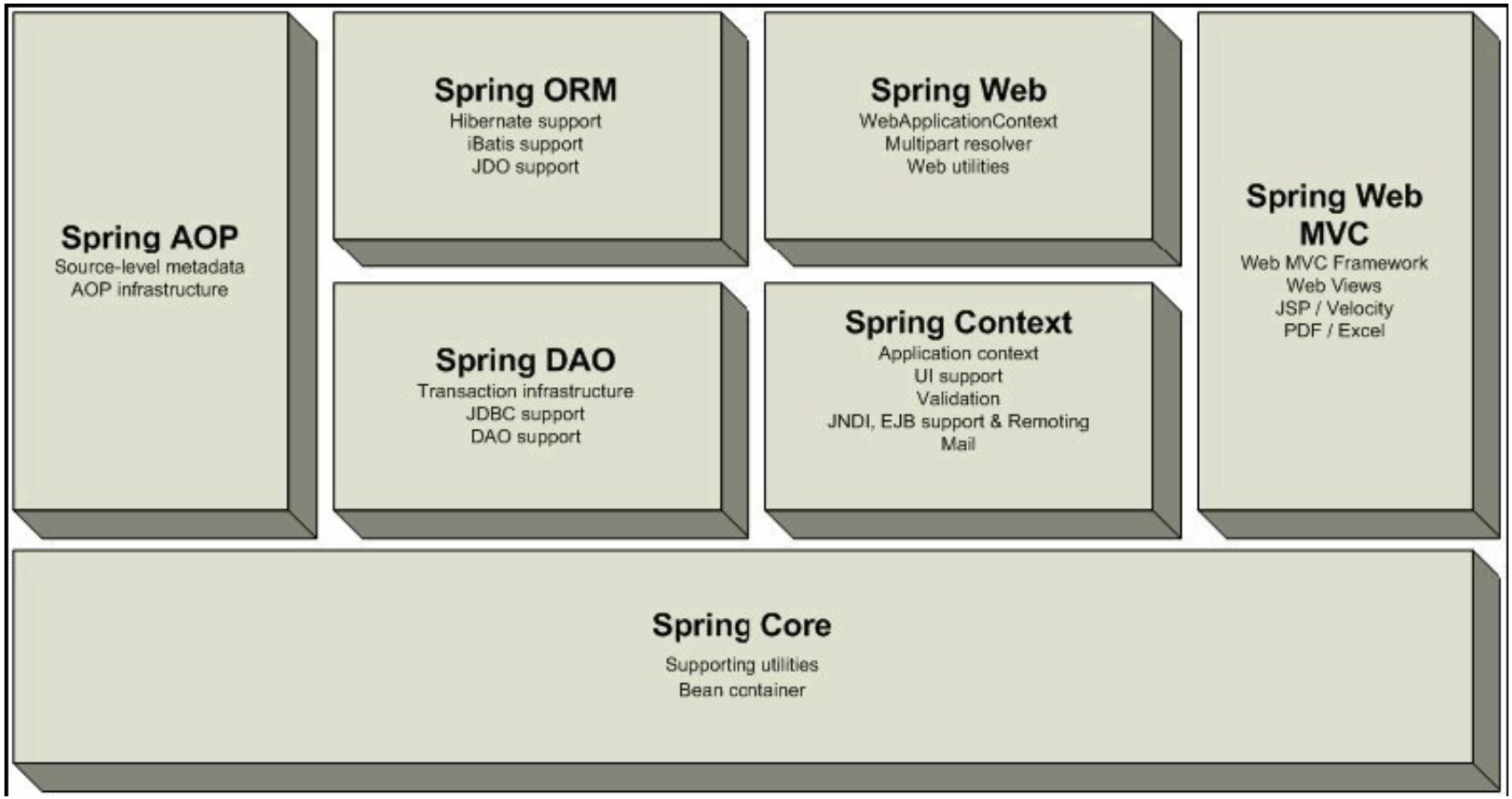
Containerarchitektur - Motivation

Aus der Enterprise Java Beans (EJB) Spezifikation 2.1 :

„*Enterprise JavaBeans will make it easy to write applications. Application developers will not have to understand low-level transaction and state management details; multi-threading; resource pooling; and other complex low-level APIs*.”

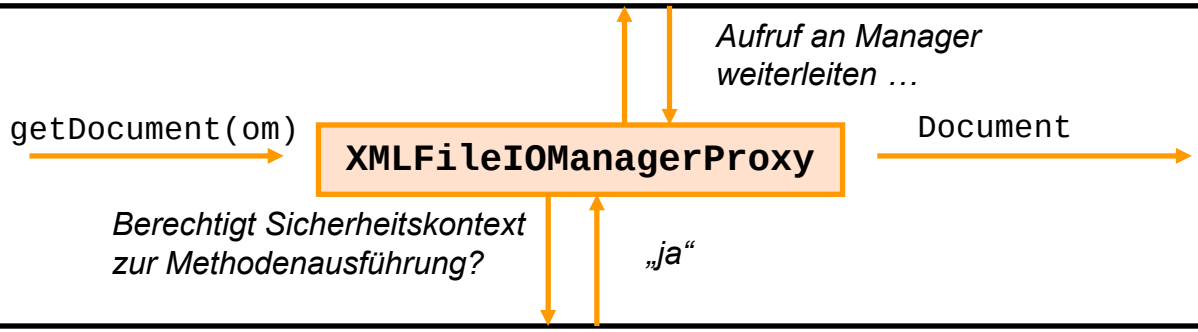
- Containerarchitekturen stellen eine Infrastruktur für die Verwaltung und Verschaltung von Komponenten einer Anwendung dar. Darüber hinaus bieten sie eine Reihe von Basisdiensten:
  - *Lifecycle Management*: Kontrolle des Lebenszyklus der Komponenten
  - *Lookup*: Lokalisierung von Komponenten
  - *Configuration*: Konfiguration und Parameterisierung von Komponenten
  - *Dependency Resolution*: Verwaltung der Abhängigkeiten zwischen den einzelnen Komponenten

Das Spring-Framework (1)
Struktur der Subframeworks



Das Spring-Framework (2)
AOP mit Spring

```
public class XMLFileIOManagerImpl {
  Document getDocument(ObjectModel om) throws UnauthorizedException {}
  // etc.
}
```



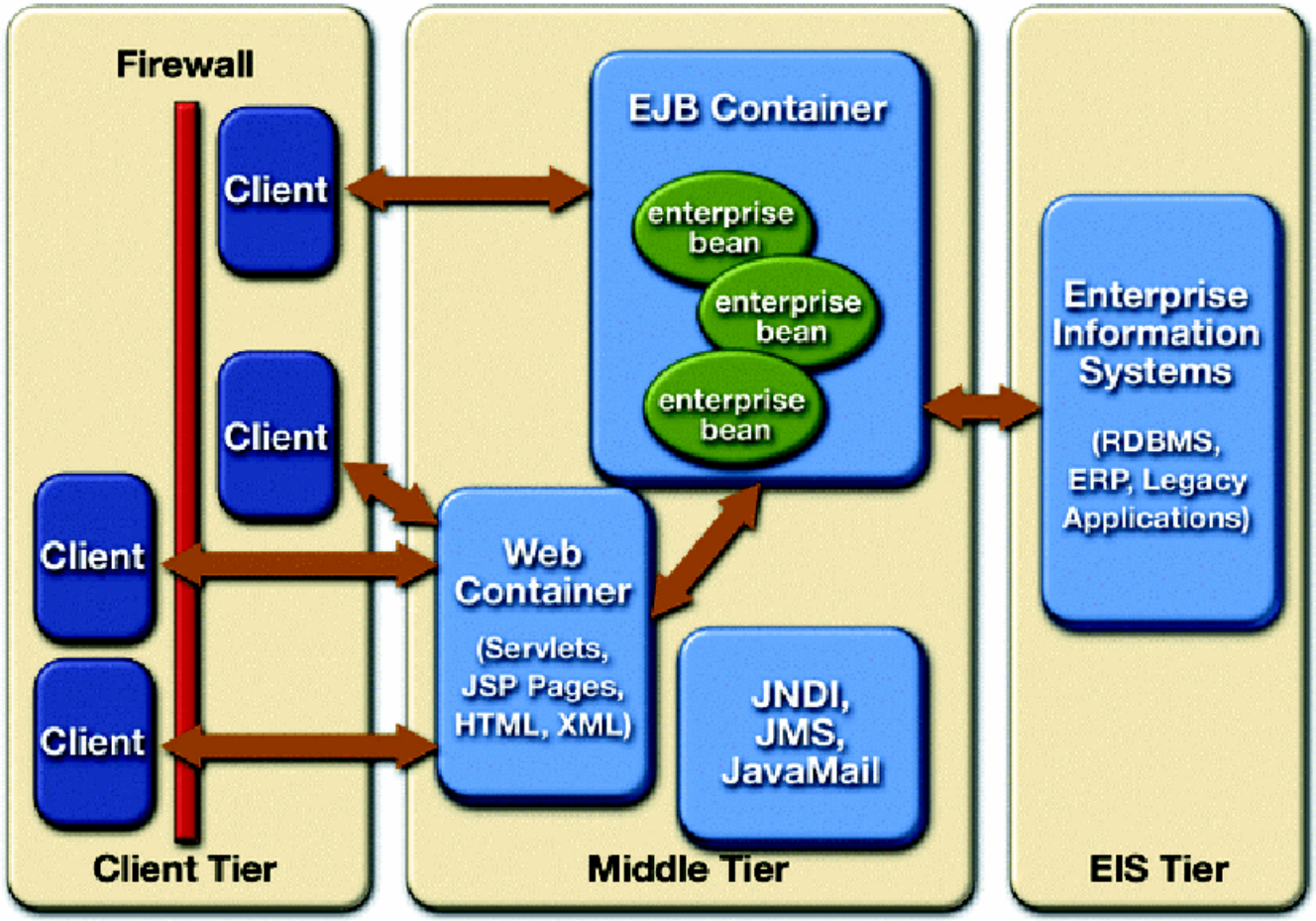

```
public class SecurityAdvice implements MethodInterceptor {
  public Object invoke(MethodInvocation invocation) throws Throwable {
    doSecurityCheck();
    return invocation.proceed();
  }
  protected void doSecurityCheck() throws UnauthorizedException {
    // custom implementation of this security aspect
  }
```

Spring Framework (3)
Zusammenfassung

“I’ve come to the conclusion that people forget about regular Java objects because they haven’t got a fancy name – so while preparing for a talk Rebecca Parsons, Josh Mackenzie and I gave them one: POJO (Plain Old Java Object). A POJO domain model is easier to put together, quick to build, can run and test outside of an EJB container, and isn’t dependent on EJB (maybe that’s why EJB vendors don’t encourage you to use them.” – *Martin Fowler*

- Spring realisiert eine Containerarchitektur basierend auf *leichtgewichtigen Komponenten* und dem Konzept der *dependency injection*.
- Spring ist eine Sammlung unterschiedlicher, bewährter Frameworks für bestimmte, dedizierte Aufgabenbereiche (Datenhaltung, Transaktionsmanagement, Verteilung, Logging, Web-Anwendungen)
- Für die Komponenten in Spring ist es irrelevant, ob sie innerhalb oder außerhalb eines Containers laufen. Daher vereinfachen sich Entwicklung, Testen, Portierung, etc...

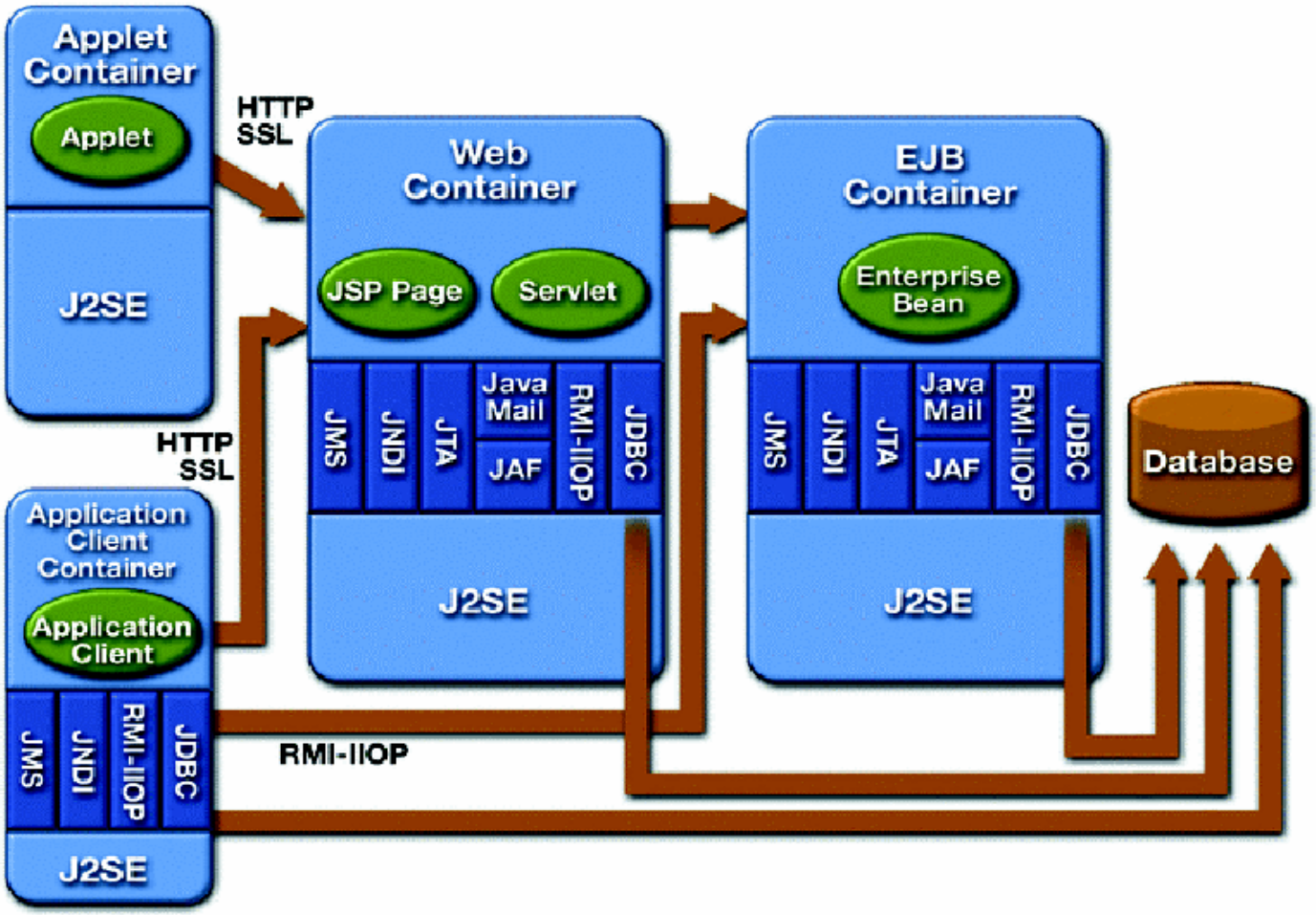
EJB (1)
Überblick der J2EE-Architektur



EJB (2)
Überblick

- J2EE ist eine Ansammlung von Java-Technologien bzw. Standards um das 3-/4-Schichtmodell zu unterstützen!
- EJB ist der Teil der Sammlung, der die Umsetzung der Anwendungsschicht betrifft.
- Sun liefert zum Teil Implementierungen oder definiert Standards, die von anderen Herstellern zu erfüllen sind
- EJB ist ein solcher Standard mit einer Referenzimplementierung von Sun (vgl. http://java.sun.com/j2ee)
- EJB ist ein serverseitiges Komponentenmodell und zudem transaktional, verteilt, skalierbar, sicher, portabel, integrierbar und netzwerkfähig
- In der J2EE-Architektur finden sich unterstützende Services
  - Naming Service (JNDI)
  - Transaction Service (JTA)
  - Messaging Service (JMS)
  - ...

EJB (3)
Fat Client und Thin Client



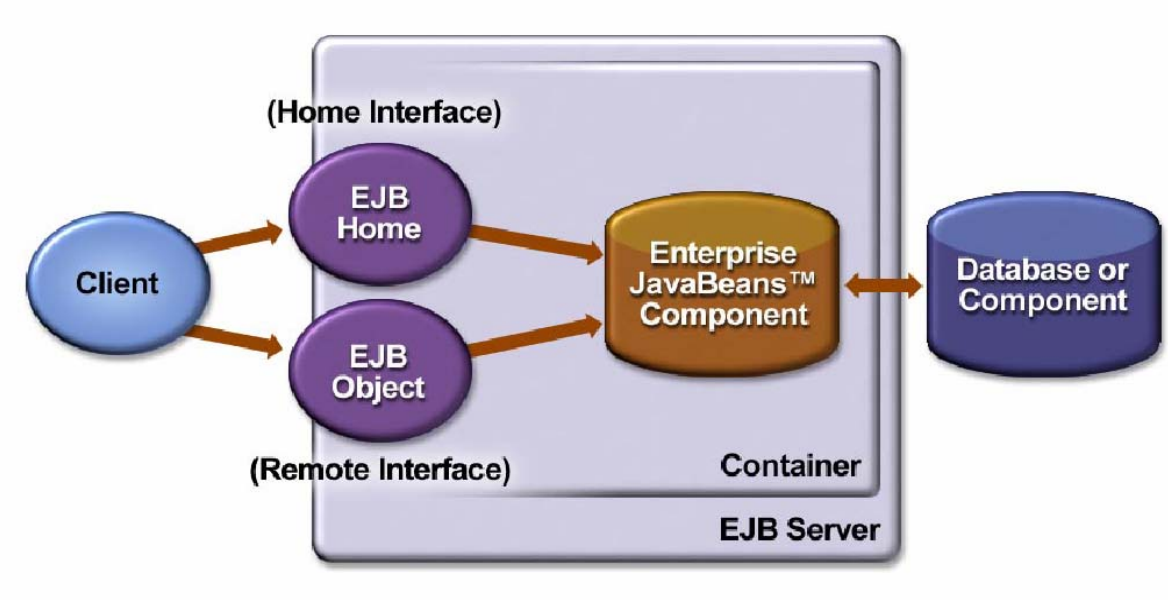
EJB (4)
Komponententypen

Enterprise Java Beans				
Entity Beans		Session Beans		Message-Driven Beans
Container-Managed Persistence	Bean-Managed Persistence	Stateful Session Beans	Stateless Session Beans	

- Entity Beans (meist fachliche Komponenten)
  - Zur Verwaltung persistenter Daten
  - Stehen mehreren Clients gleichzeitig zur Verfügung
  - Transaktionen werden immer vom Container gesteuert
  - Container-Managed Persistence
    - der EJB-Container übernimmt das Speichern der Entity Beans
    - die Spezifizierung der Attribute erfolgt im Descriptor
    - ab EJB 2.0 steuert der Container auch die Persistenz von Objektreferenzen
  - Bean-Managed Persistence
    - die Entity Bean sorgt (z.B. über JDBC) selbst für ihre Persistenz und die ihrer Objektreferenzen
- Session Beans: (meist fachliche Abläufe)
  - besitzen keine persistenten Daten und sind stets nur einem Client zugeordnet
  - Stateless: keinen eigenen Zustand, können beliebig vielen Clients nacheinander zur Verfügung gestellt werden, können in Pools ausgelagert werden
  - Stateful: besitzen interne Variablen mit der Lebensdauer der Session und können sich mit dem Client „unterhalten“, da ihr interner Zustand zwischen den Methodenaufrufen erhalten bleibt
- Message-Driven Beans (erst ab EJB 2.0)
  - zum Versenden asynchroner Nachrichten über die Java Message Service API

EJB (5)
Grundsätzlicher Aufbau

- Die **Bean-Klasse** implementiert die Funktionalität der Bean
- Das **Home-Interface** definiert eine Schnittstelle zum Erzeugen, Finden und löschen von EJBs durch Clients
- Das **Remote-Interface** stellt die „öffentlichen“ Methoden aus der Bean-Klasse den Klienten (für den Aufruf über RMI) zur Verfügung

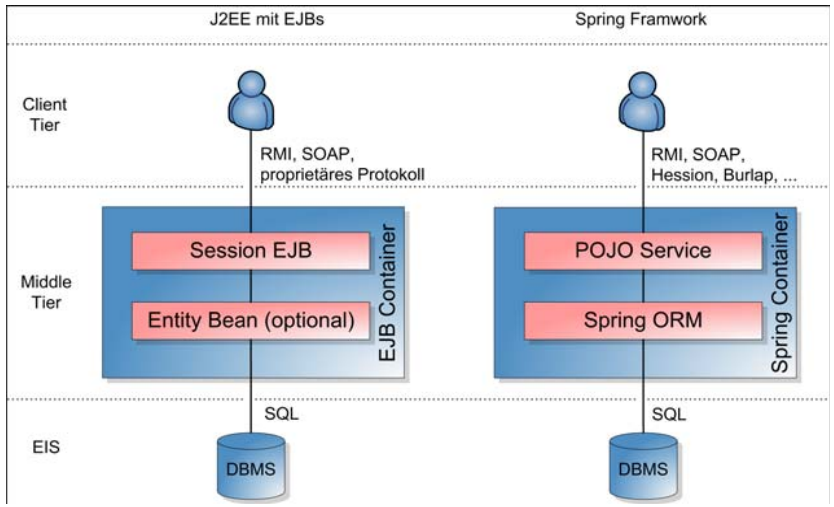


EJB (6)
XML-Deskriptor

- ermöglicht zentrale Administration der verwendeten Beans
  - konkurrierenden Zugriff (concurrency)
  - mehrere Threads in einer Entity Bean möglich
  - Persistenz und Transaktionen
  - Festlegung der Beansorte, Persistenz- und Transaktionstypen
  - bei Container-Managed Persistence Definition der Attribute
  - bei Container-Managed Transactions Festlegung der Transaktionsgrenzen
  - Namensvergabe und Paketverwaltung (naming)
  - Ortstransparenz aus Sicht der Beans
  - Sicherheitsmanagement (security)
  - binden von Security-Rollen an Beans zur Zugriffssteuerung der Beans untereinander

EJB (7)
Zusammenfassung

- EJB ist eine Containerarchitektur, die *schwergewichtige Komponenten* vorsieht
- Die Komponenten sind meist von mehreren Basisklassen abgeleitet und unterliegen strikten Anforderungen, beispielsweise hinsichtlich Threading und Vererbung
- Verhältnismäßig komplex: um eine Bean remote verfügbar zu machen, sind 4 Interfaces und eine Klasse notwendig
- Dafür die Möglichkeit, über „Session Facades“ die fachliche Logik zu kapseln
- Weitere Verbesserungen mit EJB 3.0



Inhalt

- Umsetzung: Komponenteninfrastrukturen
  - Motivation : ein einfaches Beispiel
  - Leichtgewichtige Containerarchitekturen : Spring
  - Schwergewichtige Containerarchitekturen : EJB
- Wiederverwendung durch Muster
  - Designmuster
  - Architekturmuster
  - Analysemuster
- Referenzarchitekturen
- Produktlinienarchitekturen
- Zusammenfassung

Muster – Wieso, weshalb, warum?

- Klassen sind sehr kleine Einheiten, deren Wiederverwendung steigert die Produktivität kaum.
- Datenbanken, Middleware und COTS-Komponenten ermöglichen Wiederverwendung auf der Ebene von ausführbaren, binären Komponenten.
- Frameworks und Bibliotheken unterstützen lediglich die Wiederverwendung von Programmcode.
- Aspect-oriented programming und Codegeneratoren unterstützen die Wiederverwendung von spezifischen Lösungsansätzen auf der Implementierungsebene.

- Muster als Instrumentarium zur Wiederverwendung von Spezifikationen, Design und Entwurfswissen

Ursprung des Musteransatzes

- Der Begriff „Pattern“ als Lösungsmuster, das Lösungsansätze für ähnliche bzw. sich wiederholende Designprobleme bietet, geht zurück auf den Architekten Christopher Alexander
- Beispiel Window Place [Ale77]:
  - Jeder liebt Aussichtsplätze in Hochhäusern, auf Bergen oder am Meer
  - Wenn man in einen Raum kommt der Fenster und Sitzplätze hat, die Sitzplätze aber nicht bei den Fenster sind, dann steht man vor folgendem Problem:
    - Man möchte sich hinsetzten, um es bequem zu haben
    - Man möchte möglichst in der Nähe des Lichts sein
  - Jeder Raum, in dem man sich länger aufhält sollte mindestens einen „Window Place“ haben – Sitzplätze vor dem Fenster

„Every pattern we define must be formulated in the form of a rule which establishes a relationship between a context, a system of forces which arises in that context and a configuration, which allows these forces to resolve themselves in that context.“

Muster sind Lösungsbeschreibungen

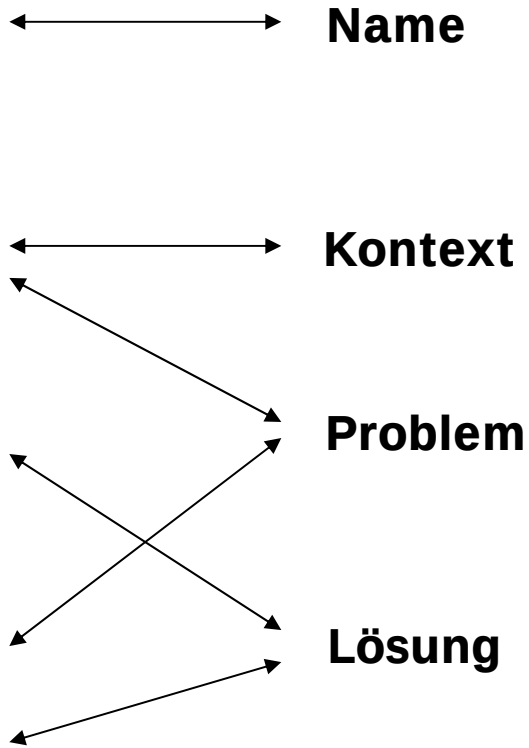
- Ein Designproblem beruht darauf, dass in gewissen Situationen (**Kontext**), sich widerstrebende Kräfte (**Problem**) entgegenstehen. Die **Lösung** besteht darin, mit den in der jeweiligen Designdisziplin zur Verfügung stehenden Werkzeugen die Situation so zu gestalten, dass diese Kräfte optimal ausgeglichen werden.
  - In der klassischen Architektur ist dies die räumliche Gestaltung von Gebäuden
  - In der Softwarearchitektur ist dies die Struktur der Komponenten und deren Beziehungen
- Ein Muster beschreibt eine Lösung für ein Designproblem
  - Muster sind konstruktiv, d.h. sie liefern Instruktionen zur Lösung einer Problemsituation
  - Muster sind abstrakt , d.h. sie bieten ein breites Anwendungsspektrum mit vielen verschiedenen Anwendungsfällen
  - Muster sind stabil, d.h. sie erfassen den stabilen Kern der Lösung
- Muster sind über die Zeit hinweg gültig und weitestgehend invariant gegenüber neuen Möglichkeiten

Mustersammlungen

- Musterkataloge sind eine einfache meist kategorisierte Sammlungen ähnlicher oder komplementärer Muster.
- Eine Mustersprache bildet eine Einheit, in der die zusammenhängenden Muster miteinander kooperieren, um ein gemeinsames Problem zu lösen.
- Mustersysteme umfassen eine Menge von Mustern auf verschiedenen Abstraktionsstufen. Die Beziehungen zwischen den Mustern beschreiben, wie sich durch die Kombination der Muster Lösungen komplexerer Probleme konstruieren lassen.
- In der Regel entwickelt sich aus einem Musterkatalog eine Mustersprache und eventuell dann ein Mustersystem.

Beschreibungsschema

- Bezeichnung und Überblick
  - Name
  - Kurzbeschreibung, Zweck
  - alternative Bezeichnungen
- Zweck
  - Motivation und Zielsetzung
  - Anwendbarkeit
- Lösung
  - graphische Beschreibung der Struktur
  - beteiligte Klassen
  - Interaktion
- Konsequenzen
  - Vor- und Nachteile
  - Implementierung und Beispiel-Code
  - Anwendungsbeispiele
  - Bezug zu anderen Mustern

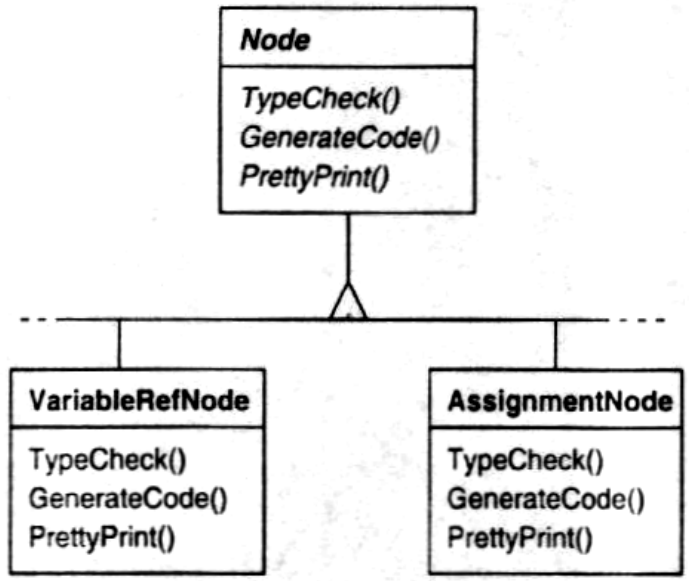


Beispiel für ein Design Pattern
das Visitor Muster (1)

- **Name:** Visitor
- **Zweck:** Führe auf jedem Objekt einer Struktur eine bestimmte Operation aus. Definiere neue Operationen, ohne die Klassen der Struktur zu ändern.
- **Motivation:** Operationen auf abstrakten Syntaxbäumen als Beispiel

  Ohne Visitor:

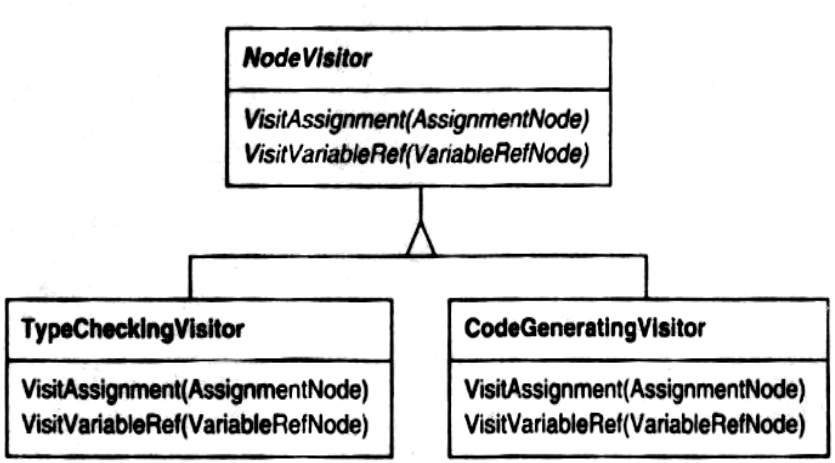
  - Knoten implementieren alle Operationen
  - beim Hinzufügen einer neuen Operation müssen alle Knoten geändert werden
  - Code für gleichartige Operationen ist über viele Klassen verstreut

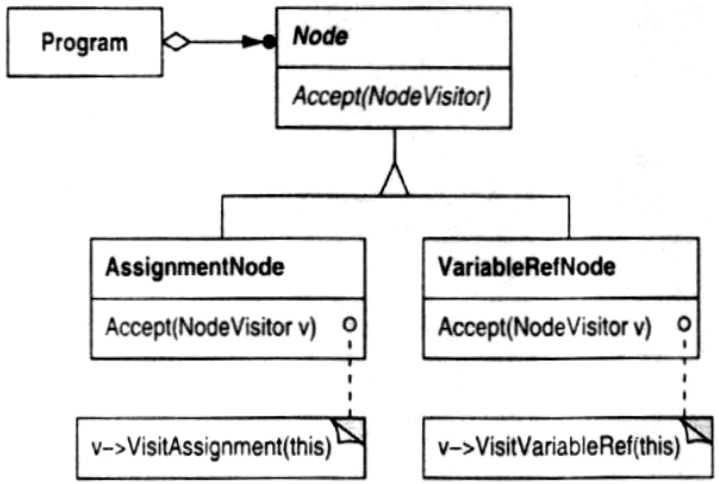


Beispiel für ein Design Pattern
das Visitor Muster (2)

Mit Visitor:

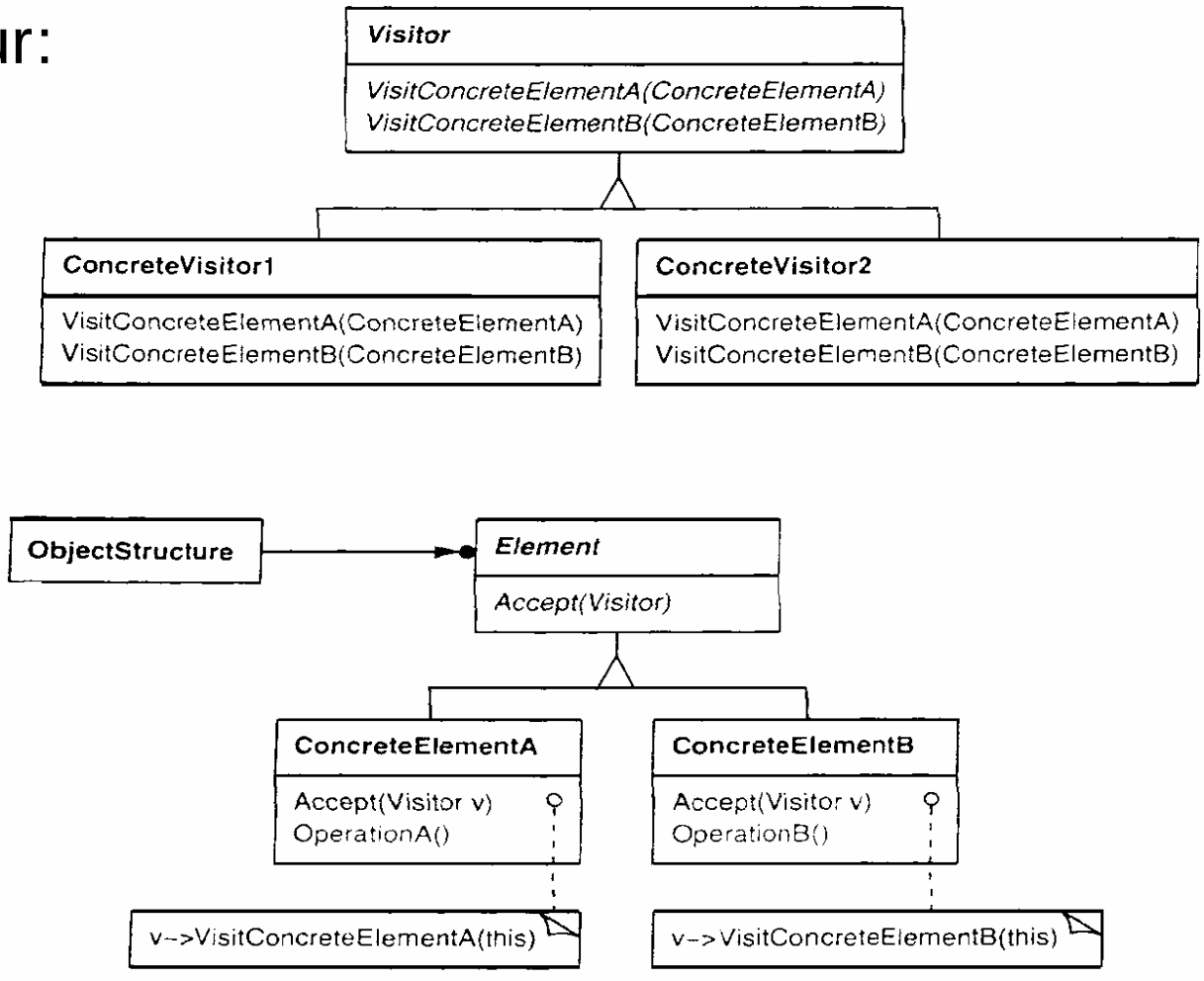
- Eine neue Visitor-Klasse für jede Operation als Unterklasse von NodeVisitor, enthält den Code für alle Typen von Knoten
- Knoten haben einen „Haken“ zum Andocken für den Visitor
- Neue Operationen können einfach durch Bilden einer neuen Klasse hinzugefügt werden
- Aber: Hinzufügen neuer Knotentypen ist schwierig!





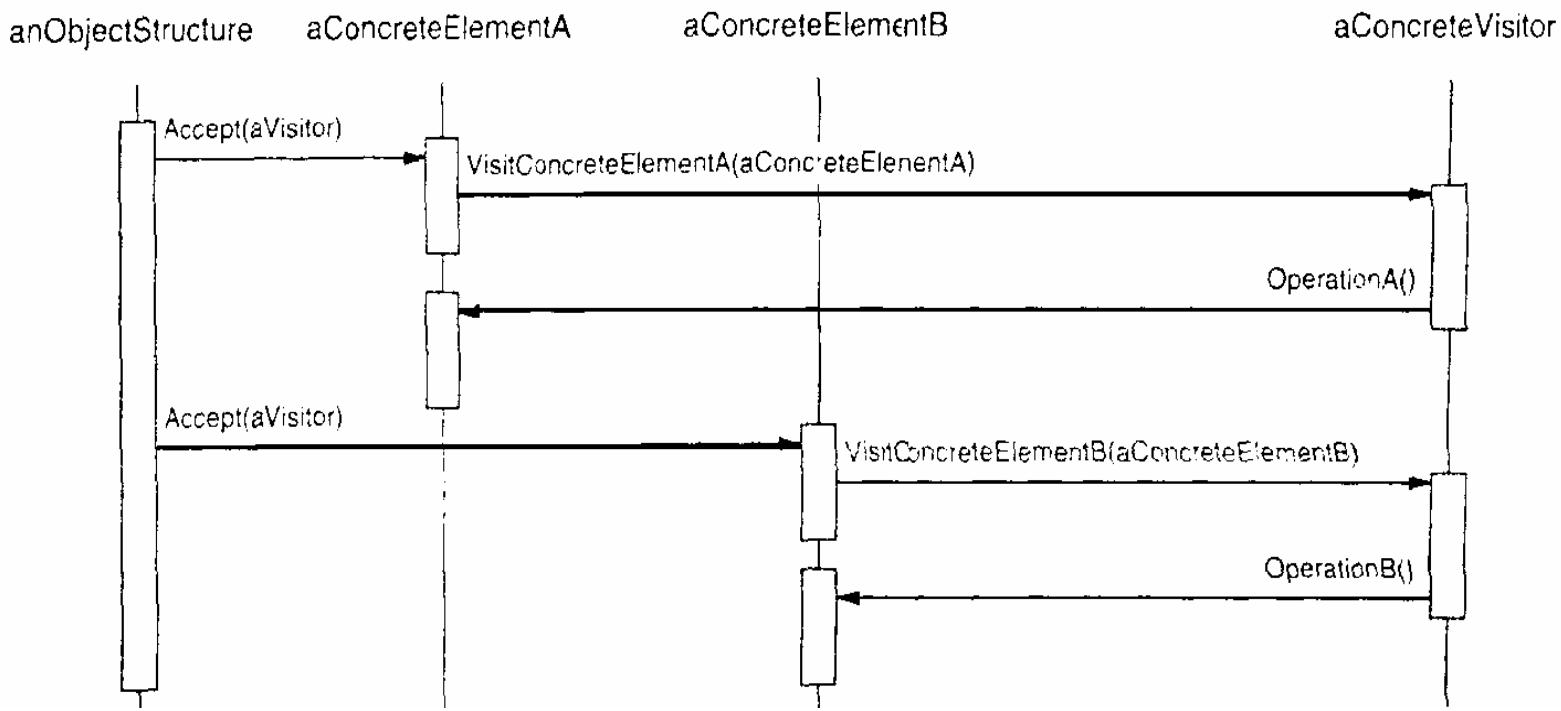
Beispiel für ein Design Pattern
das Visitor Muster (3)

- Struktur:



Visitor Muster (4)

- Interaktion:

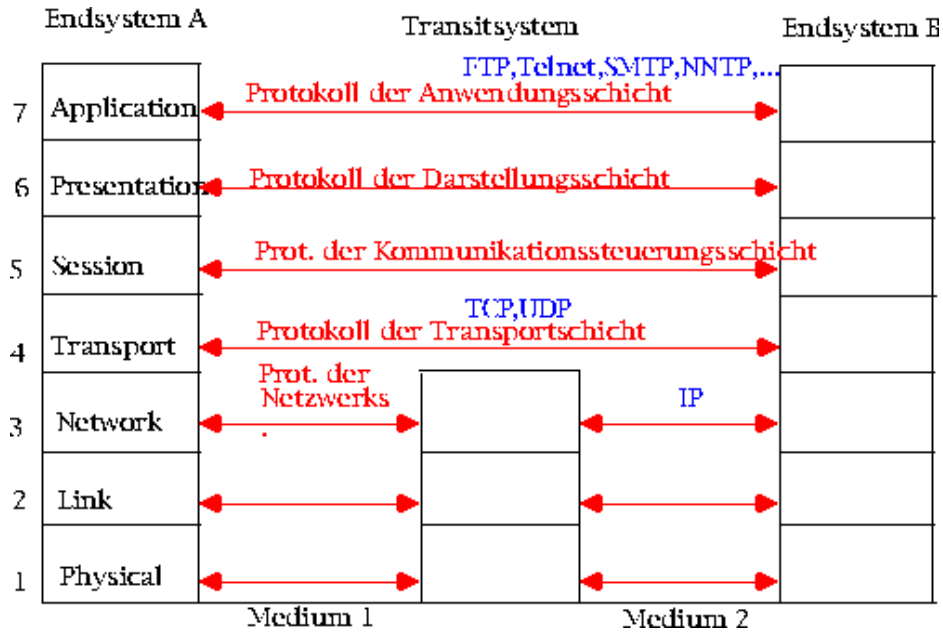


Architekturmuster

- Ein Designpattern beschreibt bewährte Lösungsansätze auf Klassenebene
- Ein Architekturmuster stellt ein Lösungsschema für ein Problem im Architekturentwurf dar. Es umfasst Strukturen und Verhalten, die durch Teildefinitionen von Komponenten, Klassen, Methoden, Vererbungs- und Komponentenbeziehungen sowie erklärenden Text beschrieben werden.
- Wiederverwendung von bewährten Architektur-Entwürfen
- Dokumentation von Entwurfsentscheidungen
- Effizientere Kommunikation durch ein Standardvokabular
- Dokumentiert im „Siemens-Buch“ [BMR+96]

Beispiel für Architekturmuster: Layers Muster (1)

- **Name:** Layers
- **Zweck:** Strukturiere Anwendungen, deren Funktionalität in Gruppen von Unterfunktionen zerfällt, wobei man die einzelnen Gruppen jeweils unterschiedlichen Abstraktionsniveaus zuordnen kann.
- **Beispiel:** Netzwerkprotokolle wie ISO/OSI, TCP/IP



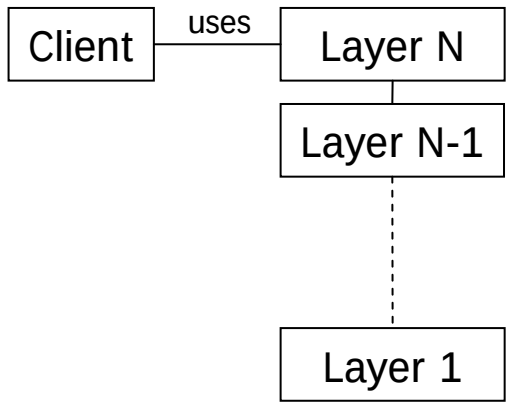
Beispiel für Architekturmuster: Layers Muster (2)

- **Kontext:** Ein großes System, das strukturiert werden muss.
- **Problem:**
  - Organisation von Funktionalität auf unterschiedlichen Abstraktionsebenen, von der Benutzungsoberfläche bis hin zu Gerätetreibern
  - Pflichtenheft spezifiziert allgemeine Anforderungen und konkrete Zielhardware, wobei die Abbildung der Anforderungen auf die Hardware nicht unmittelbar einsichtig ist
  - Kräfte, die ausbalanciert werden müssen:
    - Änderungen an Anforderungen und Quellcode sollen lokal bleiben
    - Schnittstellen sollen stabil bleiben (evtl. Verwendung von Standardschnittstellen)
    - Komponenten und Hardware/Software-Plattformen sollen austauschbar sein
    - Low-Level-Funktionalität soll in mehreren Systemen verwendet werden
    - Komponenten mit komplexer Funktionalität sollen weiter strukturiert werden
    - Möglichst wenige Komponenten sollen bei der Lösung einer Aufgabe involviert sein
    - Das System soll von einem Team von Programmierern mit klaren Verantwortlichkeiten erstellt werden
    - ...

Beispiel für Architekturmuster: Layers Muster (3)

- **Lösung:** Strukturiere das System in Schichten aus Komponenten, die auf der gleichen Abstraktionsebene angesiedelt sind. Komponenten einer höheren Schicht können zur Erledigung ihrer Aufgabe auf Komponenten der gleichen und der unmittelbar darunter liegenden Schicht zurückgreifen.
- **Struktur:**

Class Schicht J	**Collaboration** Schicht J-1
Responsibility • stellt Dienste für Schicht J+1 zur Verfügung • ruft Dienste von Schicht J-1 auf	



Beispiel für Architekturmuster: Layers Muster (4)

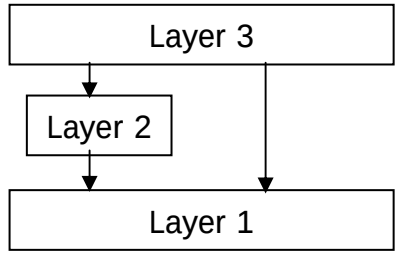
- **Dynamik:**
  - Szenario 1 (Top-Down): Requests von höheren Schichten werden durch alle Schichten bis zur untersten durchgereicht, eventuelle Replies kommen analog zurück
  - Szenario 2 (Bottom-Up): Notifications niederer Schichten werden von höheren Schichten analysiert, interpretiert und gegebenenfalls weiter nach oben durchgereicht
  - Szenario 3: wie 1, nur dass Requests nicht unbedingt bis ganz nach unten durchgereicht werden, sondern bereits von einer mittleren Schicht vollständig bearbeitet werden können
  - Szenario 4: wie 3, nur für Notifications
  - Szenario 5: Kombination von 1 bis 4 für zwei kommunizierende Protocol Stacks (ein Request eines Stacks wird zu einer Notification im anderen Stack und umgekehrt)

Beispiel für Architekturmuster: Layers Muster (5)

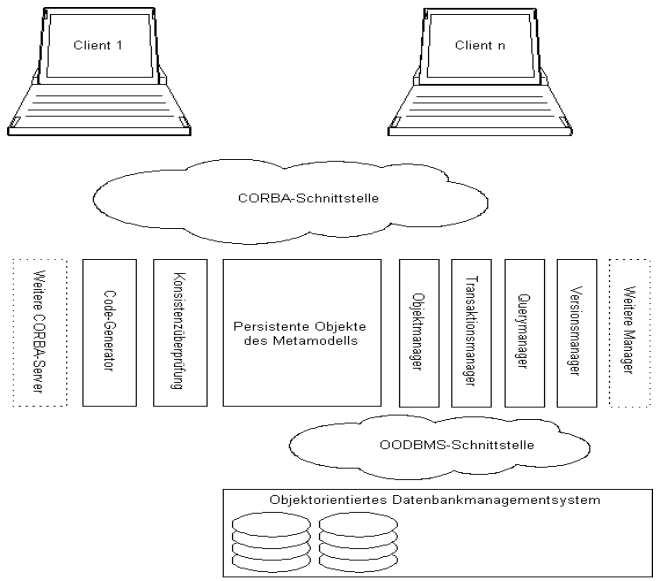
- **Vorgehen bei der Implementierung:**
  - Bestimme das Abstraktionskriterium
  - Bestimme die Anzahl der Abstraktionsebenen
  - Benenne die Schichten und ordne ihnen Aufgaben zu
  - Spezifiziere die Dienste der Schichten
  - Verfeinere die Aufteilung in Schichten
  - Spezifiziere die Schnittstelle der Schichten
  - Strukturiere die einzelnen Schichten
  - Spezifiziere die Kommunikation zwischen den einzelnen Schichten
  - Schirme die einzelnen Schichten gegeneinander ab
    - Implementierungsvarianten: Schnittstellen, Callbacks, Reactor-Entwurfsmuster
  - Entwerfe eine Strategie für die Fehlerbehandlung
    - Behandle Fehler möglichst auf unteren Abstraktionsebenen vollständig!
    - Reiche keine Low-Level-Fehlermeldungen nach oben durch!

Beispiel für Architekturmuster: Layers Muster (6)

- **Varianten:**
  - Abgeschwächte Schichtenarchitektur: Kommunikation nicht nur zwischen benachbarten Schichten
    Tradeoff: Performance vs. Wartbarkeit
  - ...



- **bekannte Anwendungsfälle:**
  - APIs
  - Virtual Machines
  - Informationssysteme mit Zwei- oder Dreischichtenarchitektur
  - Betriebssysteme



Beispiel für Architekturmuster: Layers Muster (7)

- **Vor- und Nachteile:**
  - \+ Wiederverwendung von Schichten
  - \+ Möglichkeit zur Standardisierung
  - \+ Minimierung von Code-Abhängigkeiten
  - \+ Austauschbarkeit der Implementierung von Schichten

  - – Änderungen müssen eventuell durch alle/viele Schichten durchgezogen werden
  - – Geringere Effizienz durch viele Schichtenübergänge
  - – Unnötige und doppelte Arbeit (Bsp: Fehlerkorrektur auf allen Schichten)
  - – Schwierigkeit, die „korrekte“ Anzahl von Schichten zu finden

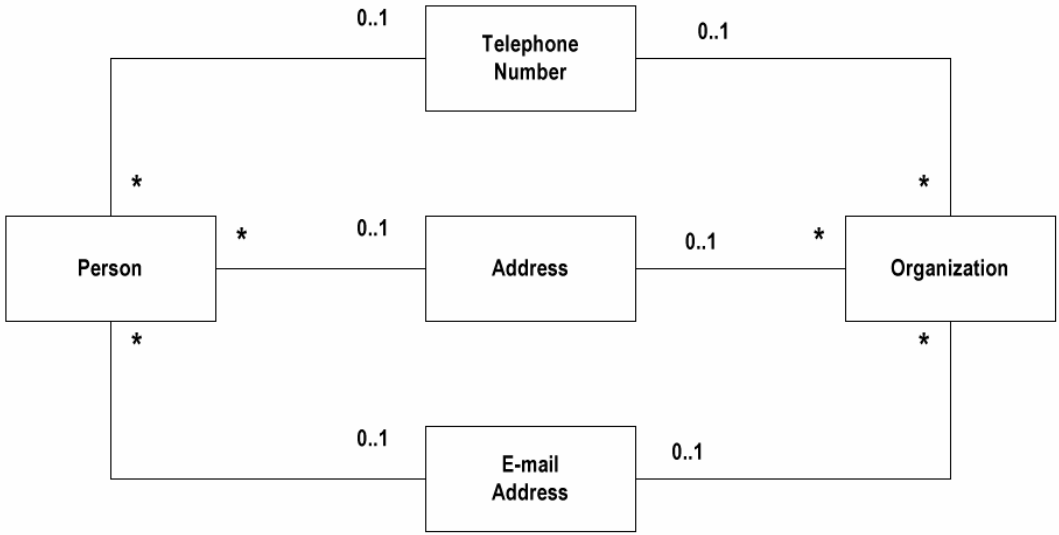
- **Beziehungen zu anderen Mustern**
  - Composite Message (Variante von Composite Muster für den Entwurf von Datenpaketen zwischen Schichten)
  - ...

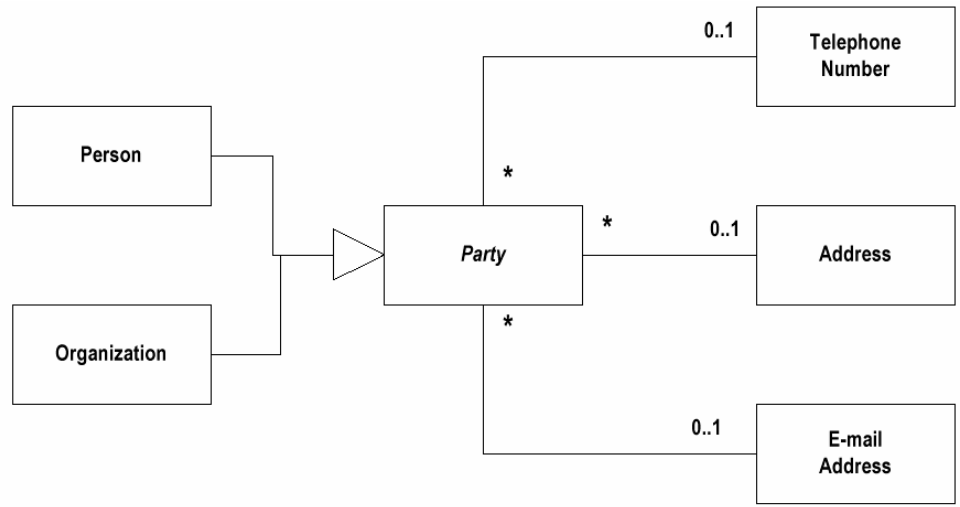
Analysemuster: Übersicht

- Analysemuster entstammen einer Idee von Fowler [Fow97]
  - Zeigen hohes Potential für frühe Phasen des Entwurfs
  - Dokumentieren wertvolles, internes Wissen
  - Kontinuierliche Weiterentwicklung des internen Wissens
  - Hohe Wiederverwendung von Branchenwissen
  - Über die Standardisierung erreicht man Interoperabilität und Flexibilität
- Fowler‘s Analysemuster sind keine vollständigen Muster
  - Kein einheitliches Dokumentationsschemata
  - Keine klare Struktur; es fehlt ein Musterkatalog
  - Nur Dokumentation von „Nähkästchen-Wissen“ in Form von Beispielen

Beispiel für Analysemuster: Organisation und Verantwortlichkeit (1)

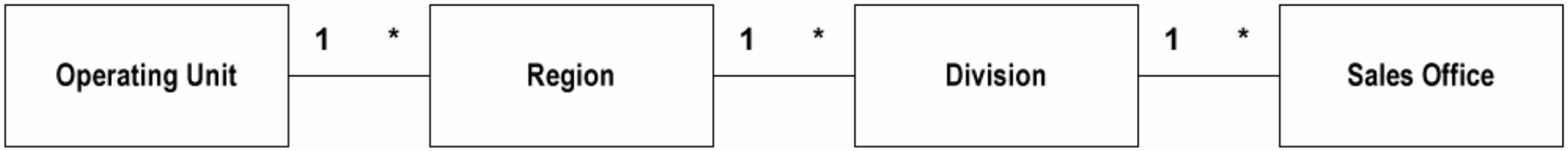
- Naive Modellierung einer Adressverwaltung
- Problem: Finde geeignete Abstraktion für Organisation und Person
- Lösung: Party als gemeinsame Abstraktion von Organisation und Person

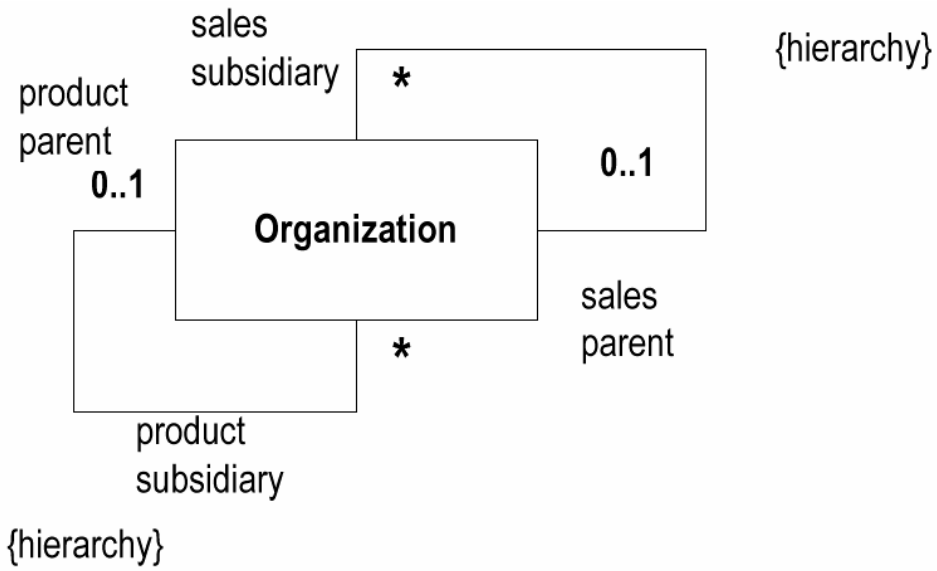




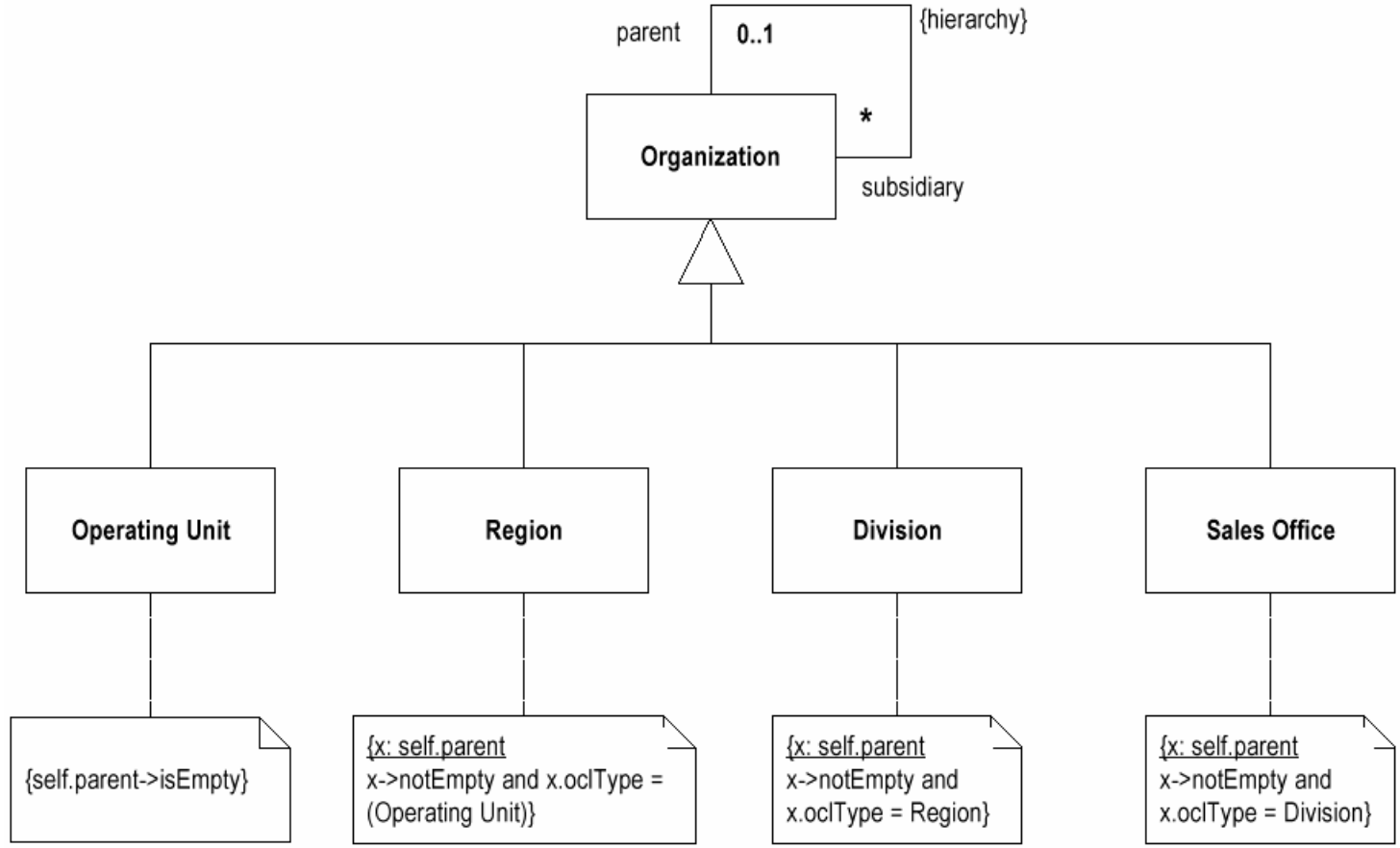
Beispiel für Analysemuster: Organisation und Verantwortlichkeit (2)

- Naive Modellierungen von Organisationshierarchie und -struktur





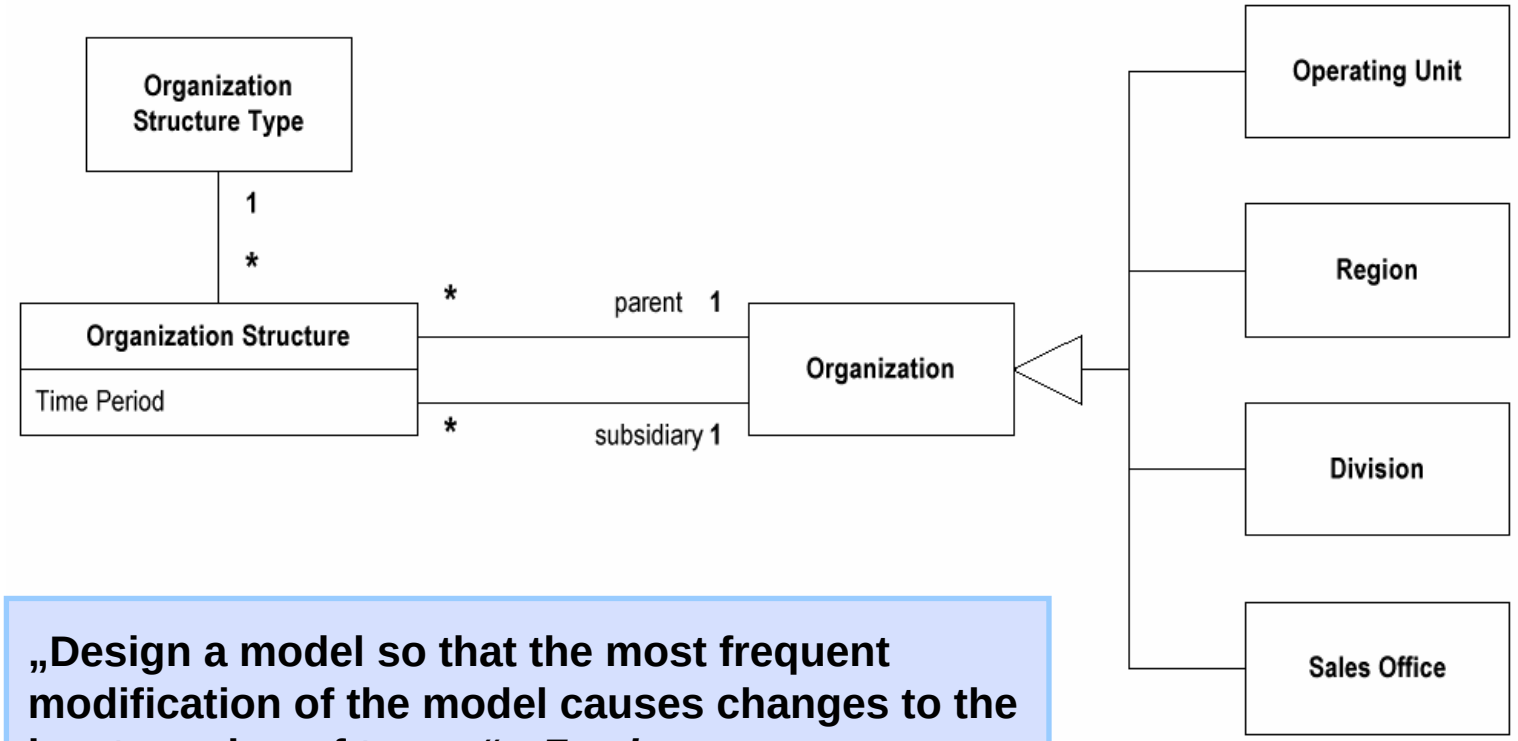
Beispiel für Analysemuster: Organisation und Verantwortlichkeit (3)



- Problem: Struktur ist inflexibel und nicht wieder verwendbar

Beispiel für Analysemuster: Organisation und Verantwortlichkeit (4)

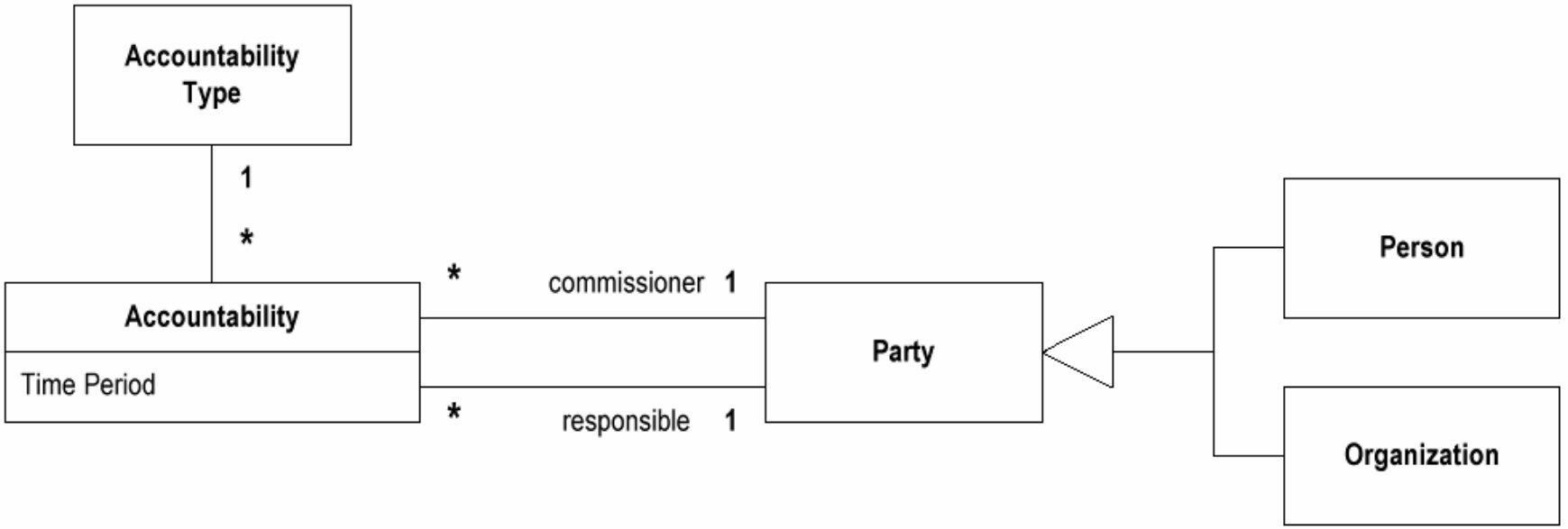
- Lösung: Verwendung einer einzigen, aber im Modell typisierbaren Assoziation



„Design a model so that the most frequent modification of the model causes changes to the least number of types.“ - *Fowler*

Beispiel für Analysemuster: Organisation und Verantwortlichkeit (5)

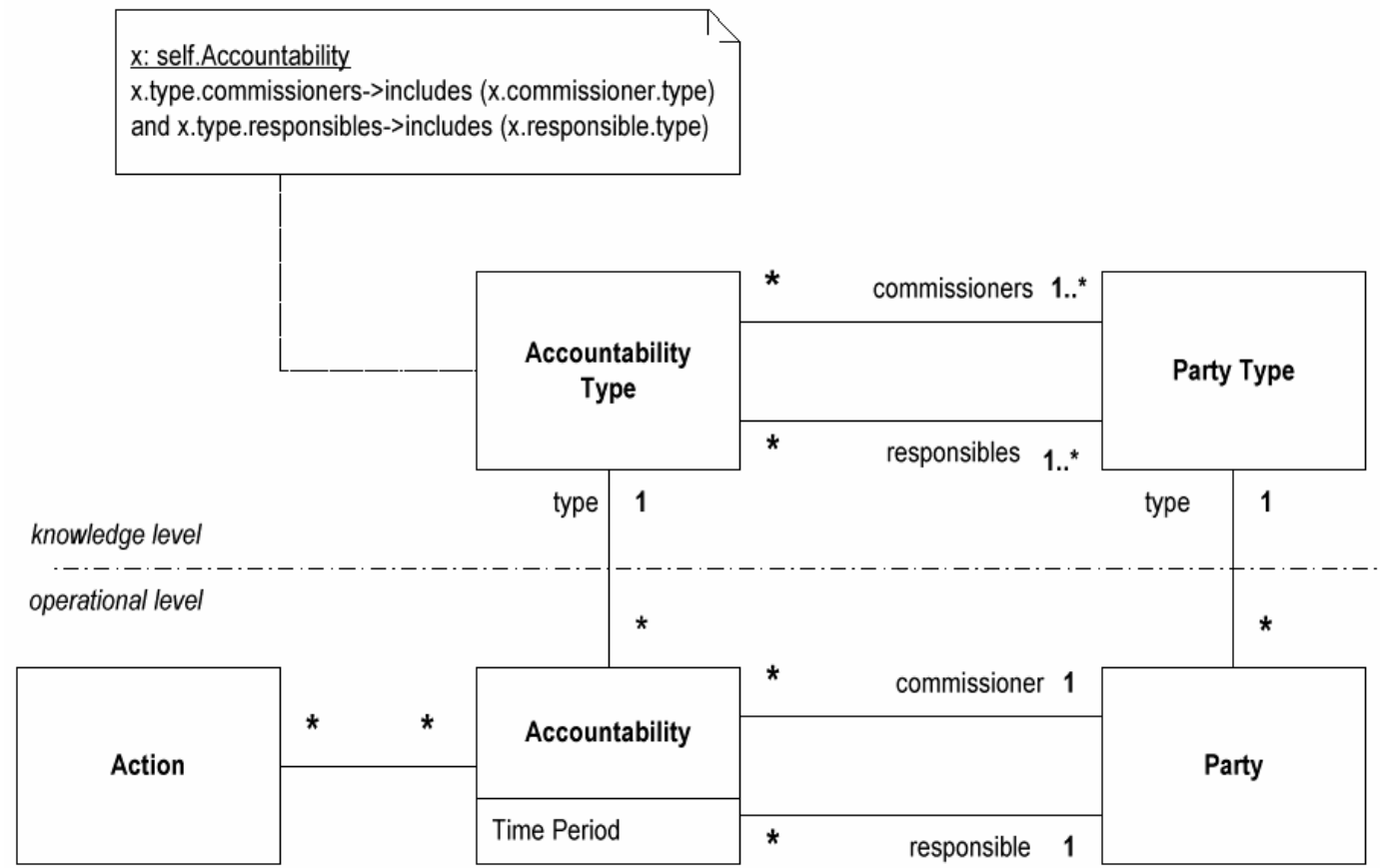
- Erweiterung: Typisierbare Assoziation über die gemeinsame Abstraktion von Person und Organisation



„Whenever defining features for a type, that has a supertype, consider whether placing the features on the supertype makes sense" - *Fowler*

Beispiel für Analysemuster: Organisation und Verantwortlichkeit (6)

- Erweiterung: Gültige Konfiguration der Assoziationen zw. Parties und Accountabilities im Knowledge Level ablegen.



Nutzen von Mustern

- Entwickler
  - Hilfe bei Entwurfsentscheidungen
  - Nutzung von erprobtem Wissen erfahrener Entwickler
  - Code-Beispiele können den Einstieg erleichtern und als Ausgangsbasis für eigene Lösungen dienen
  - Vertrautheit und leichter Einstieg durch standardisierte Beschreibung
  - Hilfe beim Weitergeben und Festhalten von eigenem Wissen
- Team
  - Bildung einer einheitlichen Sprache
  - Dokumentationsstandard
  - Wissenstransfer an neue Mitarbeiter
- Unternehmen
  - Standardisierte Dokumentation des Firmenwissens
  - Wiederverwendung erprobter Lösungen
  - Einheitliche Architektur der Firmensysteme

Problemfelder

- Aufwand für Einarbeitung und Auswahl von Mustern kann hoch sein
  - Komplexe Mustersprachen mit starken Querbezügen zwischen vielen Mustern
  - Organisation, Kategorisierung und Einordnung von Mustern
  - Derzeit kein einheitliches Schema für die Beschreibung von Mustern
- Anwendung von Mustern erfordert Erfahrung beim Entwurf von Softwaresystemen
  - Ausbalancieren der Kräfte eines Musters
  - Auswahl und Kombination von Mustern
  - Einsatz von Mustern aus unterschiedlichen Katalogen
- Erkennen von Mustern im Code eines Systems ohne Dokumentation mühsam
- Kaum Ansätze zur Werkzeugunterstützung
- Erarbeitung von hilfreichen Mustern schwierig und aufwendig
  - „Übermusterisierung“ (Was ist eigentlich kein Muster?)
  - „Einzelmuster“ (Muster, die nur einmal auftreten)
  - Engagement von erfahrenen „Schäfern“ beim „Ausbrüten“ von Mustern nötig

Inhalt

- Umsetzung: Komponenteninfrastrukturen
  - Motivation : ein einfaches Beispiel
  - Leichtgewichtige Containerarchitekturen : Spring
  - Schwergewichtige Containerarchitekturen : EJB
- Wiederverwendung durch Muster
  - Designmuster
  - Architekturmuster
  - Analysemuster
- Referenzarchitekturen
- Produktlinienarchitekturen
- Zusammenfassung

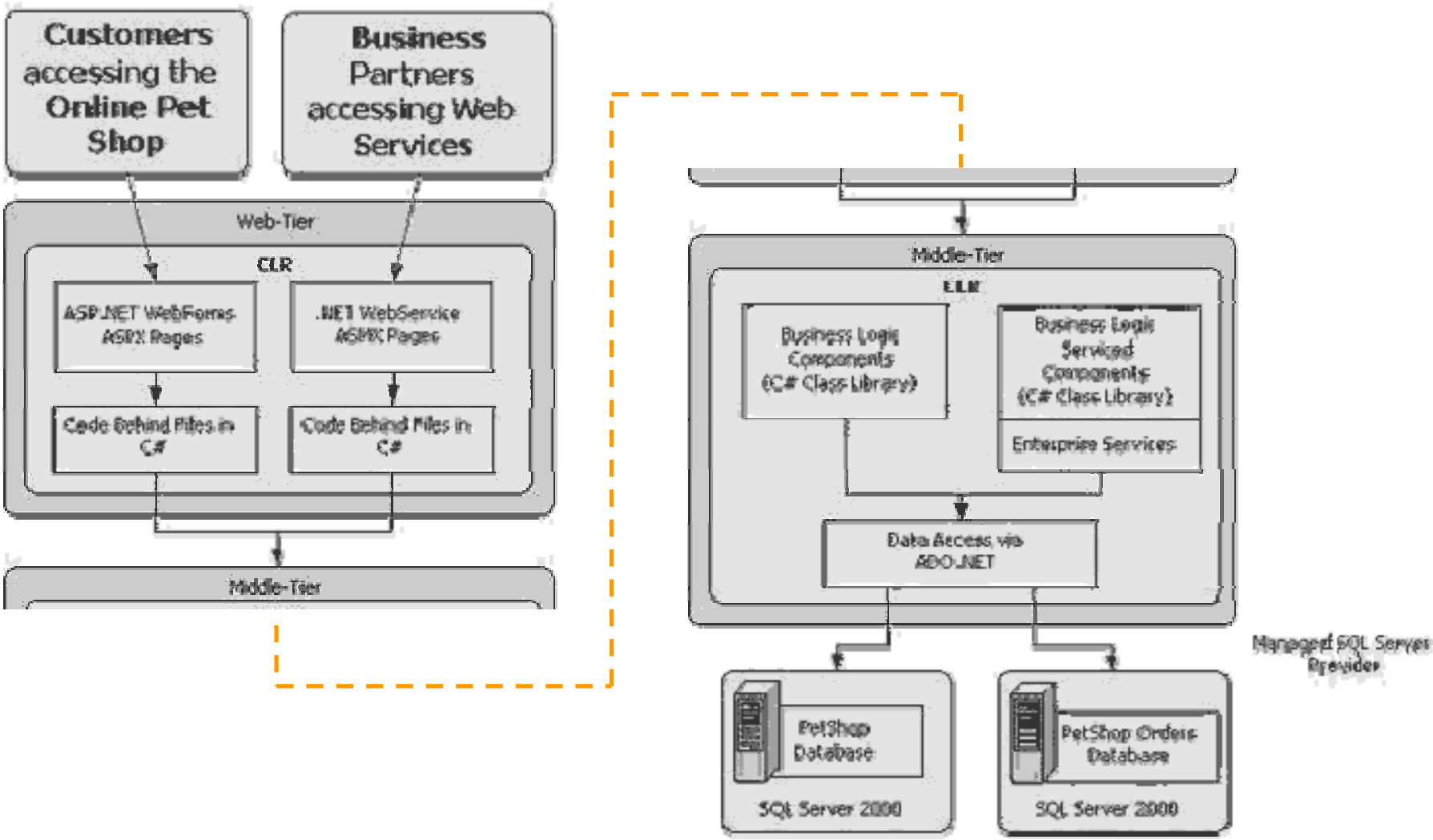
Was sind Referenzarchitekturen?

- Referenzarchitekturen beschreiben eine Vorlage (Blueprint) für die Software- oder / und Systemarchitektur von Softwaresystemen.
- Referenzarchitekturen legen insbesondere fest:
  - Funktionale Aufteilung des Systems unter technischen Gesichtspunkten
  - Technische Trägersysteme, die Rahmen für Applikationsentwicklung darstellen (Middleware, Datenbanken, Standardschnittstellen, etc.)
  - Prozesskommunikationsgrenzen und Ausführungsort von Systemteilen
  - Verwendete Systemsoftware, Netzwerke und Hardware
- Für die Entwicklung von Softwaresystemen wird dann diese Architektur „kopiert“ und gegebenenfalls modifiziert

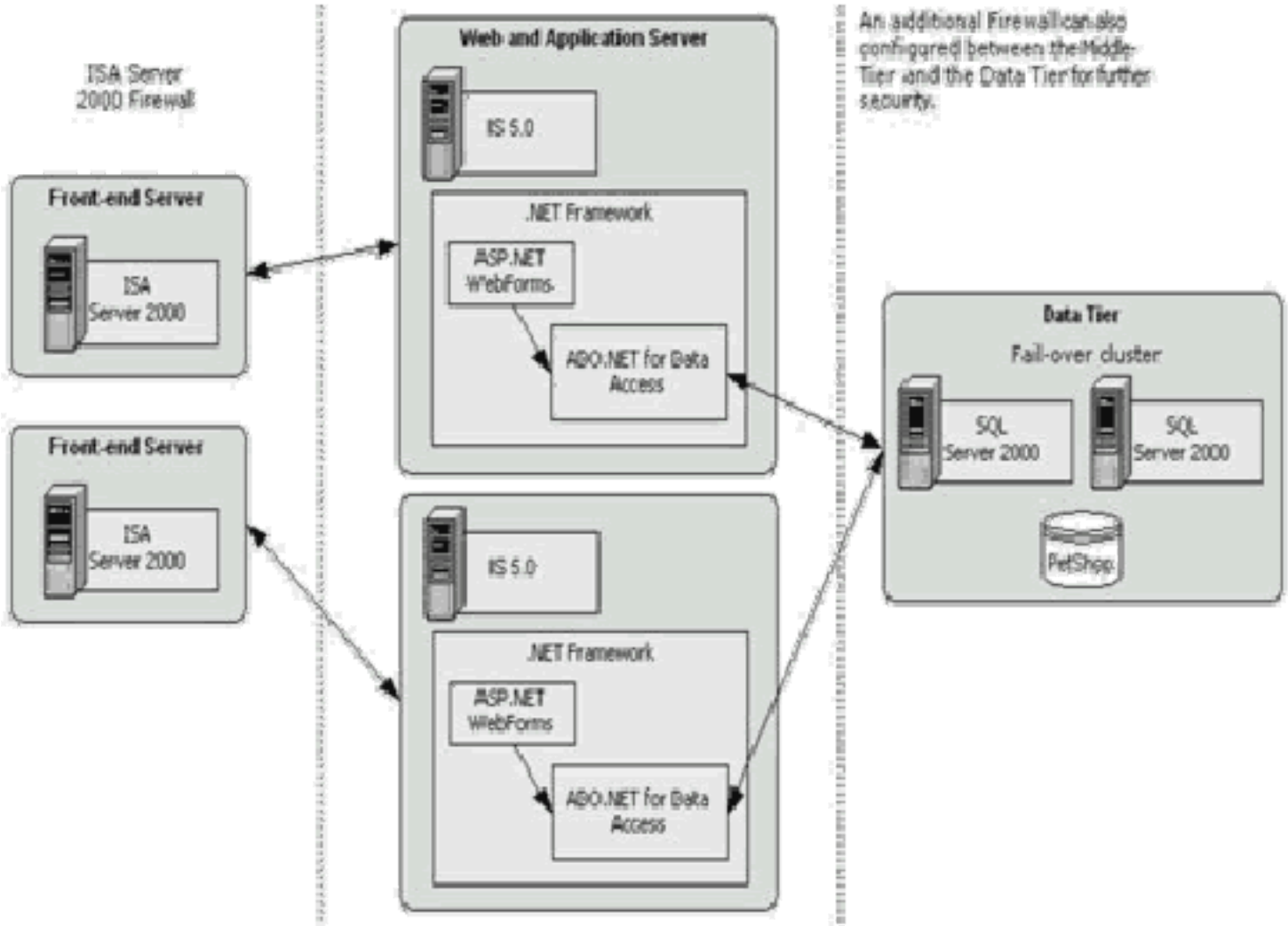
Vorteile und Nachteile von Referenzarchitekturen

- ☺ Reduzierung der Entwicklungs- und Produktionskosten
  - ☺ Wiederverwendung von Entwurfswissen
  - ☺ Geringere Lizenzkosten durch beispielsweise einheitliches Datenbanksystem
  - ☺ Geringere Wartungs- und Einarbeitungskosten da einheitliche Umgebung
- ☺ Erhöhung der anwendungsübergreifenden Interoperabilität
- ☺ Anwendungsübergreifende Wiederverwendung von Komponenten
- 😐 Abhängigkeit von Technologie- oder Lösungsanbieter
  - Technologie- und Lösungsanbieter ☺
  - Anwender ☹
- ☹ Referenzarchitekturen beschränken sich meist auf technische Aspekte

Beispiel: Microsoft .NET Referenzarchitektur für eCommerce Systeme (1)



Beispiel: Microsoft .NET Referenzarchitektur für eCommerce Systeme (2)



Inhalt

- Umsetzung: Komponenteninfrastrukturen
  - Motivation : ein einfaches Beispiel
  - Leichtgewichtige Containerarchitekturen : Spring
  - Schwergewichtige Containerarchitekturen : EJB
- Wiederverwendung durch Muster
  - Designmuster
  - Architekturmuster
  - Analysemuster
- Referenzarchitekturen
- Produktlinienarchitekturen
- Zusammenfassung

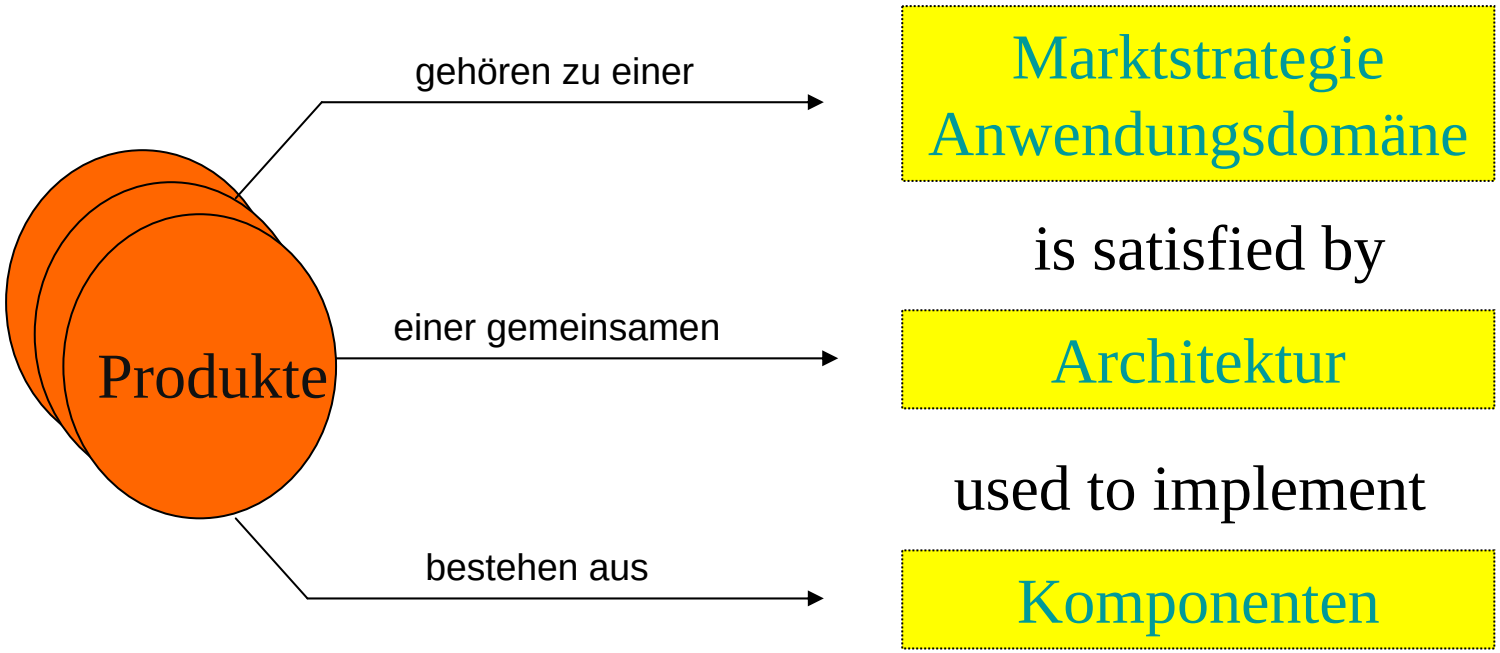
Produktlinienarchitekturen – Wieso, weshalb, warum?

- Referenzarchitekturen beschränken sich hauptsächlich darauf die technische Aspekte der Software- und Systemarchitektur von Anwendungen festzulegen.
- Bei der Produktentwicklung steht aber die Funktionalität im Vordergrund nicht die Technik!
  - Produktzyklen sind relativ kurz (1-3 Jahre)
  - Technikzyklen im Produkt sind relativ lang (5-15 Jahre)
- Strategische Erfolgsfaktoren im „heißen“ Produktmarkt
  - Kosten
  - Qualität
  - Time-to-market
  - Diversifizierung
- Der Produktlinienansatz versucht den klassischen Entwicklungs-zyklus zu ändern und Produkte aus vorgefertigten, konfigurierbaren Komponenten zusammen zu stellen.

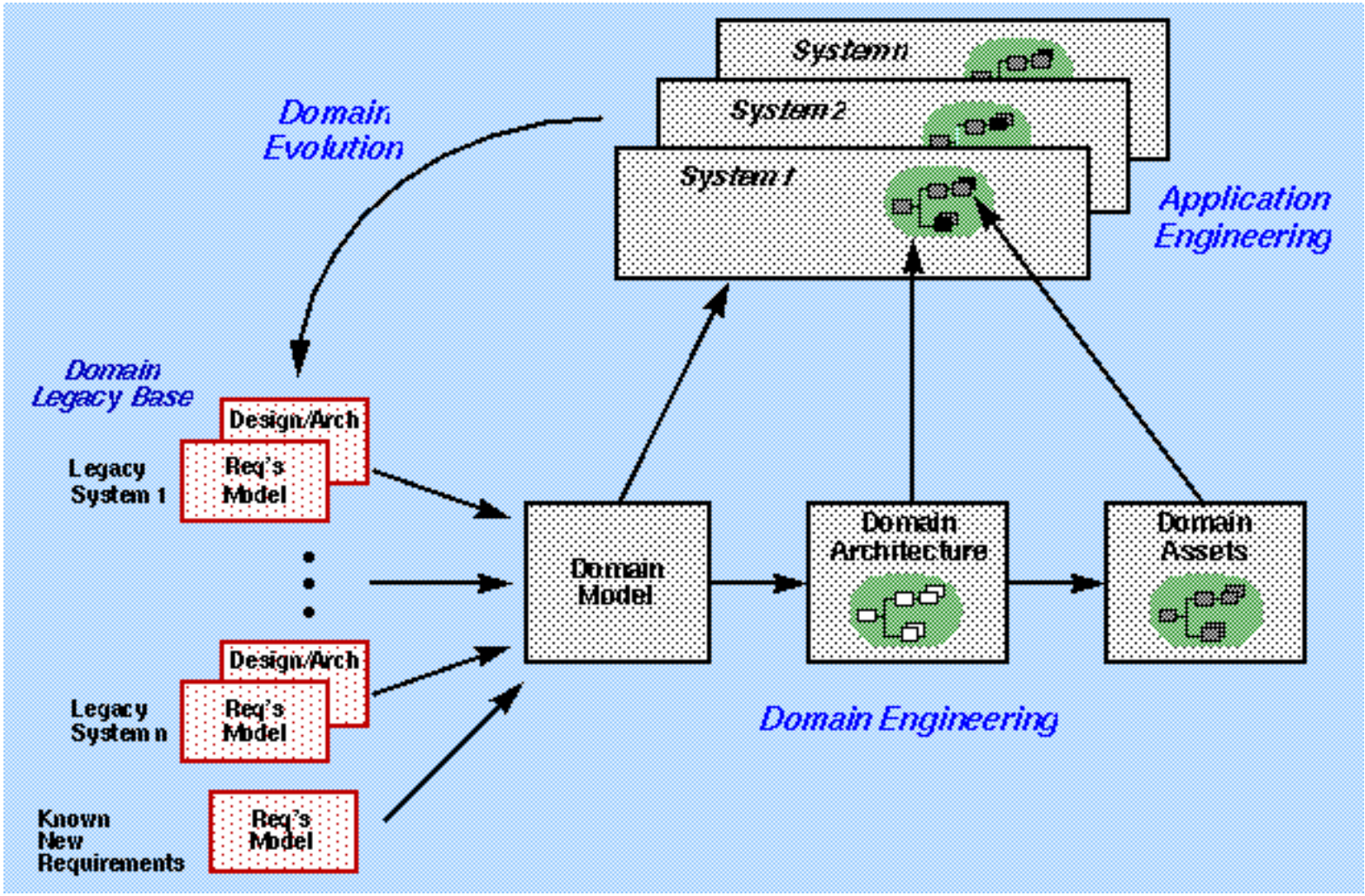
Produktlinienansatz (1)

- Produktlinienarchitekturen (Product Line Architecture) beschreiben von einer Gruppe von ähnlichen Systemen
  - Architektureigenschaften
  - Typische Komponenten
  - Beziehungen zwischen den Komponenten
- Produktlinien (Product Lines) sind eine Gruppe von Produkten, die eine gemeinsame, organisierte Menge von Fähigkeiten besitzen, um die Anforderungen des Marktes, des Benutzers oder der Anwendung zu erfüllen.
- Eine Domäne (Domain) ist ein Bereich mit einem spezifischen Anwendungswissen; eine spezielle Anwendungsexpertise.

Produktlinienansatz (2)



Produktlinienansatz (3)



Produktlinienansatz (4)

- Der Produktlinienansatz ist
  - Domänenspezifisch
  - Prozessgetrieben
  - Architekturzentriert
- Somit ermöglicht der Produktlinienansatz die Wiederverwendung verschiedener Softwareentwicklungsartefakte für die Erstellung neuer, ähnlicher Systeme, wie zum Beispiel
  - Anforderungen
  - Entwürfe und Design
  - Programmcode
  - Testfälle
- Vorsicht: Sehr schwer beherrschbar; nur zu empfehlen, wenn die Anwendungsdomäne vollständig verstanden wurde!

Inhalt

- Umsetzung: Komponenteninfrastrukturen
  - Motivation : ein einfaches Beispiel
  - Leichtgewichtige Containerarchitekturen : Spring
  - Schwergewichtige Containerarchitekturen : EJB
- Wiederverwendung durch Muster
  - Designmuster
  - Architekturmuster
  - Analysemuster
- Referenzarchitekturen
- Produktlinienarchitekturen
- Zusammenfassung

Zusammenfassung

- Die pragmatische Wiederverwendung von Softwarearchitekturen in Form von Design, Implementierung und Erfahrung ist essentielle Voraussetzung für die effektive Softwareentwicklung
- (Wieder-)Verwendung von existierenden Implementierungen wird bereits gelebt, zum Beispiel Datenbanken, Middleware, Bibliotheken, Frameworks
- (Wieder-)Verwendung von Entwurfswissen durch Muster gehört zu dem Handwerkszeug eines guten Architekten
- Referenzarchitekturen geben den technischen Rahmen für Anwendungssysteme vor
- Der Produktlinienansatz bietet einen ganzheitlichen Wiederverwendungs- bzw. Entwicklungsansatz für ähnliche Softwaresysteme

Literaturhinweise

- [Ale77] Christopher Alexander. A Pattern Language. Oxford University Press. 1977.
- [Ale79] Christopher Alexander. The Timeless Way of Building. Oxford University Press. 1979.
- [GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. 1995.
- [BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal. Pattern-Oriented Software Architecture , Volume 1: A System of Patterns. John Wiley & Sons 1996.
- [SSR+00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Net-worked Objects. John Wiley & Sons. 2000.
- [Fow97] Martin Fowler. Analysis Patterns, Reusable Object Models. Addision-Wesley. 1997.
- [CN01] Paul Clements, Linda Northrop. Software Product Lines: Practices and Patterns. Addison Wesley Publishing Company. 2001