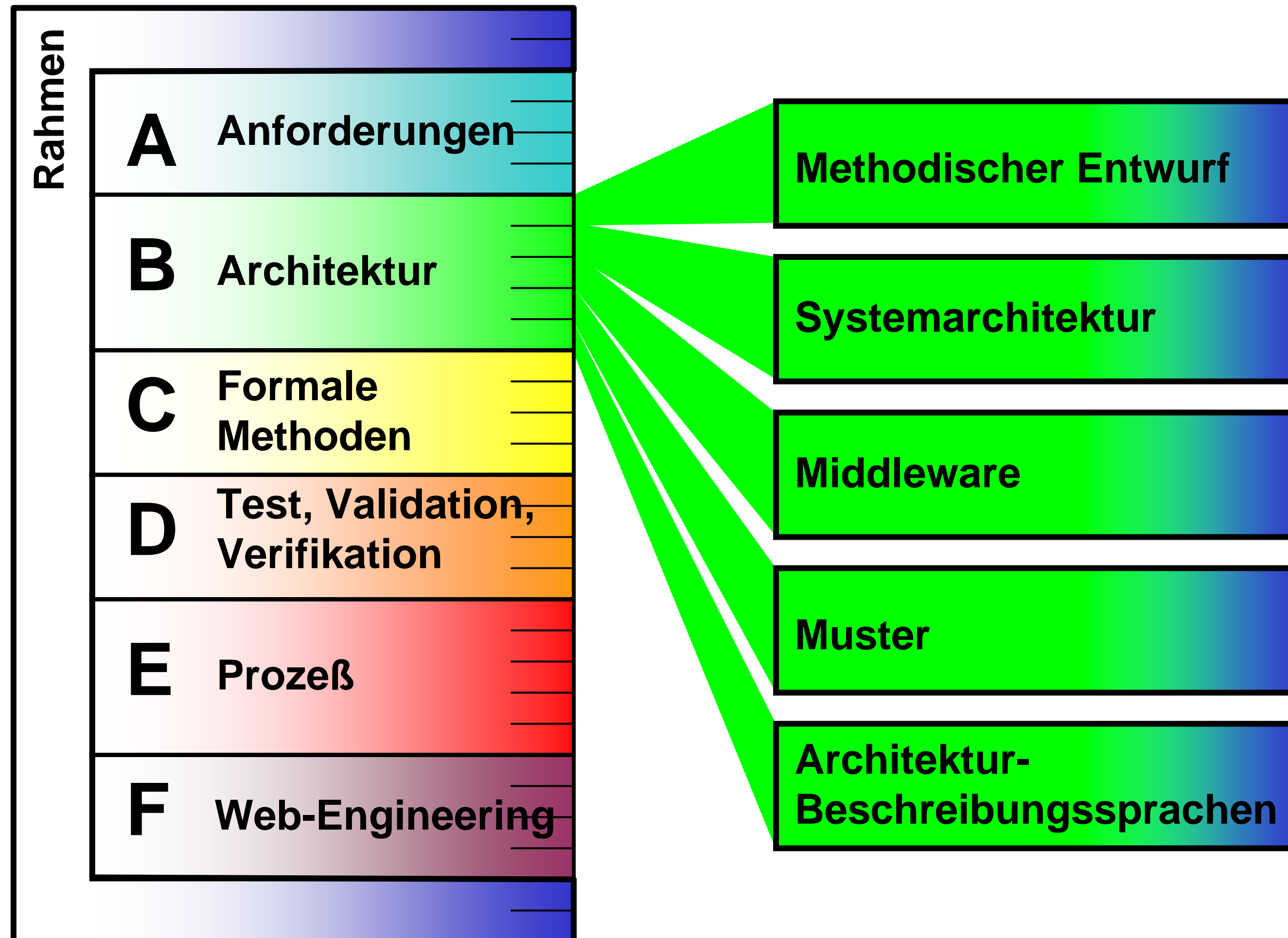
Methoden des Software-Engineering

Software Architektur

Martin Wirsing

Block B, Teil 5: Entwurfs- und Architekturmuster
WS 2006/07, LMU München

Gliederung Block B Teil 5: Muster



Ziele

- Entwurfs- und Architekturmuster kennen lernen
- Antimuster kennen lernen

Muster

Der Begriff **Muster** stammt aus der **Architektur** und wurde durch den Architekten Christopher Alexander geprägt:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

- In der Softwaretechnik sind Muster sind Wissens-„Komponenten“. Ein Muster ist eine Schablonen, die in vielen verschiedenen Situationen eingesetzt werden kann und beschreibt eine vorbildliche, bewährte Lösung für ein ausgewähltes Problem.

Entwurfsmuster: Historische Entwicklung

C. Alexander
*1936
Architekturprof. in Berkeley

- **1977 Christopher Alexander**: *A Pattern Language. Towns, Buildings, Construction* – eine Sammlung von Entwurfsmustern in der Architektur
- **1987 Kent Beck** und **Ward Cunningham**: Entwurfsmuster für graph. Benutzerschnittstellen in Smalltalk
- **1989-91 James Coplien**: Entwurfs-muster für C++ (Advanced C++ Idioms.
- **1991** Diss. **Erich Gamma** über Entwurfsmuster für Objekt-Orientierung
- **1995** Die „**Viererbande**" („Gang of Four", GoF) **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: „Design Patterns - Elements of Reusable Object-Oriented Software"
- **1996 Mary Shaw, David Garlan**: „Architectural styles" - Architekturmuster

Erich Gamma, Richard Helm, John Vlissides, Ralph Johnson

Mary Shaw
Prof. CMU

Entwurfsmuster

- **Entwurfsmuster** (engl. *design pattern*) sind *Regeln* **oder** *Richtlinien* (auch *Muster*), um häufig auftretende Probleme bei der Erstellung eines Programms zu lösen.
- Entwurfsmuster beziehen sich auf immer wiederkehrende Aufgabenstellungen, die beim Software-Design entstehen ***auf einer abstrakten Ebene*****.**
- Entstanden ist der Ausdruck „Entwurfsmuster" in der ***Architektur***, von wo er für die (objekt-orientierte) Softwareentwicklung übernommen wurde.
- Der Nutzen eines Entwurfsmusters liegt in der (programmiersprachen-unabhängigen) ***Identifikation (Konzeptualisierung) einer Klasse von Entwurfsproblemen*** und der ***Beschreibung und Diskussion*** *der Vor- und Nachteile* einer Lösung dafür.

Die Entwurfsmuster der „Viererbande"

Die 23 GoF Pattern sind in 3 Gruppen eingeteilt:

behandeln Objekterzeugung

Erzeugungsmuster:

- Abstract Factory (87)
- Builder (97)
- Factory Method (107)
- Prototype (117)
- Singleton (127)

Strukturmuster:

- Adapter (139)
- Bridge (151)
- Composite (163)
- Decorator (175)
- Facade (185)
- Flyweight (195)
- Proxy (207)

behandeln Komposition von Klassen und Objekten

behandeln Objektinteraktion und -Verantwortlichkeiten

Verhaltensmuster:

- Chain of Responsibility (223)
- Command (233)
- Interpreter (243)
- Iterator (257)
- Mediator (273)
- Memento (283)
- **Observer** (293)
- State (305)
- Strategy (315)
- Template Method (325)
- Visitor (331)

Elemente eines Musters

- Jedes Muster wird mit wenigstens vier wesentlichen Teilen beschrieben:
  - **Name**

    wird benutzt, um das Entwurfsproblem, seine Lösung und seine Folgen in einem oder zwei Worten zu beschreiben.
  - **Problem**

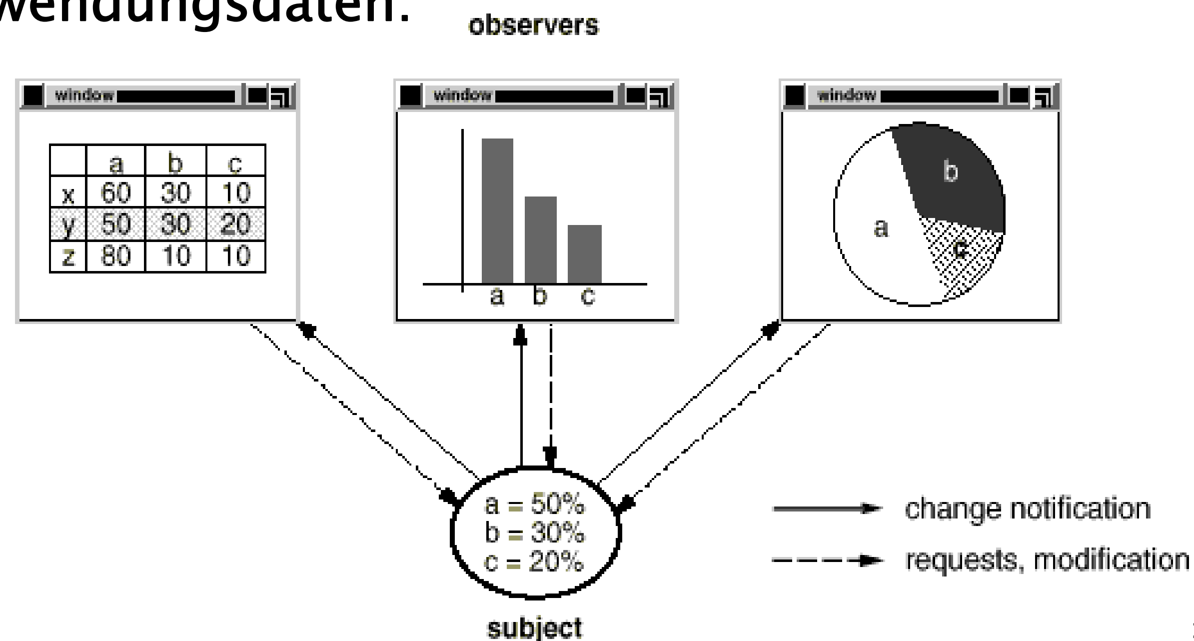
    beschreibt, wann ein Muster eingesetzt wird.
  - **Lösung**

    beschreibt die Teile, aus denen der Entwurf besteht, ihre Beziehungen, Zuständigkeiten und ihre Zusammenarbeit – kurz, die *Struktur, Teilnehmer* und *dynamisches Verhalten.*
  - **Folgen**

    sind die Ergebnisse sowie Vor- und Nachteile der Anwendung des Musters, etwa Performanz, Einflüsse auf *Flexibilität*, *Erweiterbarkeit*

Observer-Muster
Motivation

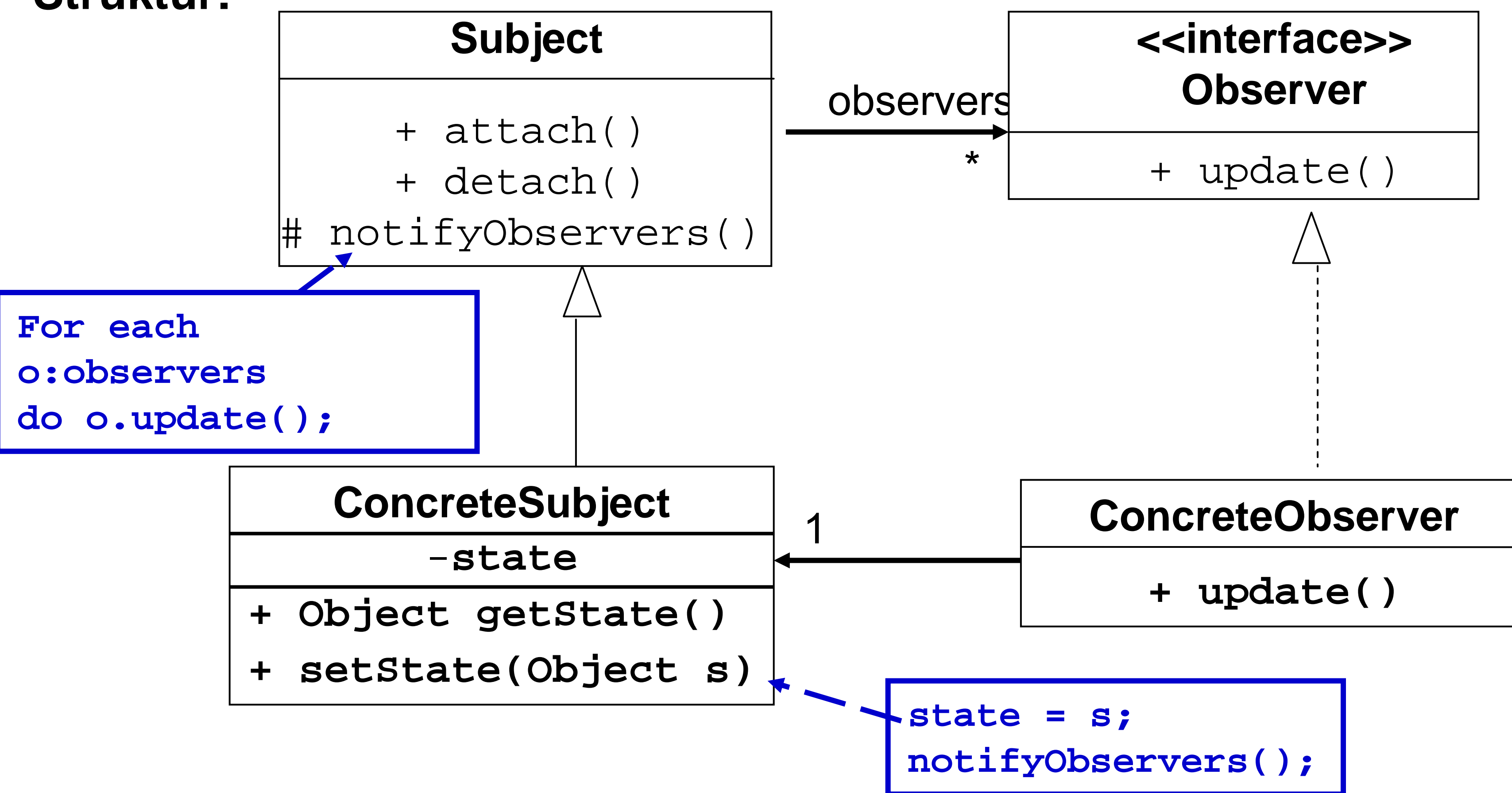
- Die Zustände von Objekten, die untereinander Beziehungen besitzen, müssen konsistent gehalten werden.
- Klassen sollen lose gekoppelt sein, um sie besser wiederverwenden zu können.
- Beispiel: GUI-Toolkit mit Trennung von Präsentation und zugrunde liegenden Anwendungsdaten:



Observer-Muster: Problem und Struktur

Problem *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.* GoF(293)

Struktur:



Observer-Muster: Teilnehmer

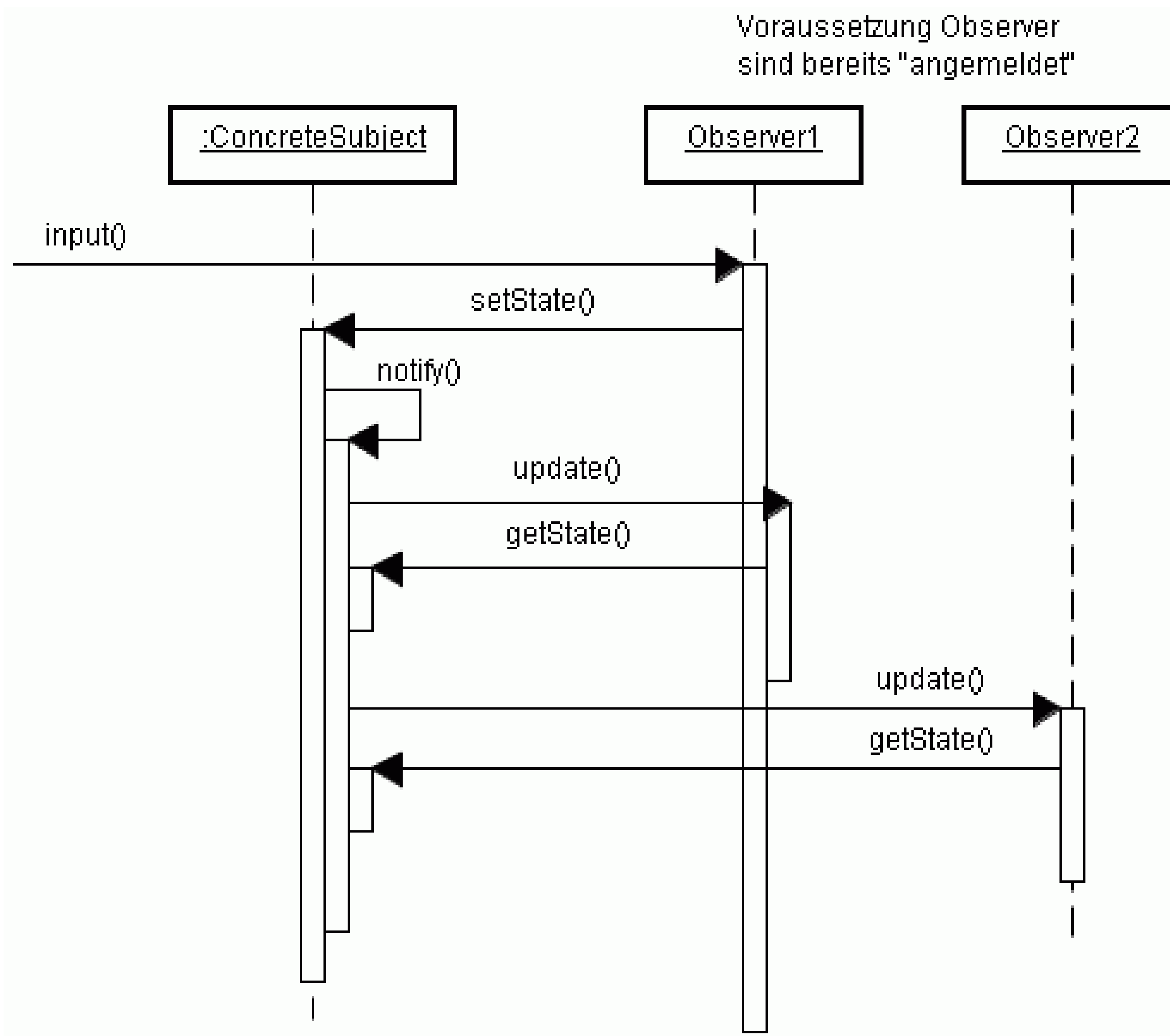
- Das **Beobachtermuster** (Observer) ermöglicht die Weitergabe von Änderungen eines Objekts an abhängige Objekte.
- **Teilnehmer**
  - **Subject (Beobachtbares Objekt, auch Publisher, also „Veröffentlicher“, genannt)**
    - Abstrakte Klasse zur Verwaltung einer beliebige Zahl von Observern
    - kennt Liste von Beobachtern, aber keine konkreten Beobachter
    - bietet Schnittstelle zur An- und Abmeldung von Beobachtern
    - bietet Schnittstelle zur Benachrichtigung von Beobachtern über Änderungen
  - **Observer (auch Subscriber, also „Abonnent“, genannt):**

    Interface für Objekte, die an Zustandsänderungen eines ConcreteSubject Objekts interessiert sind.
  - **ConcreteSubject (Konkretes, beobachtbares Objekt)**
    - Hält einen Zustand, an dem Observers Interesse haben, und benachrichtigt alle Beobachter bei Zustandsänderungen über deren Aktualisierungsschnittstelle
    - Schnittstelle zur Erfragung des aktuellen Zustands
  - **ConcreteObserver (Konkreter Beobachter):**

    Reagiert auf Zustandsänderungen der assoziierten Instanz von ConcreteSubject.

Observer-Muster: Dynamisches Verhalten

- Das ConcreteSubject benachrichtigt seine Observer bei jeder Zustandsänderung, die den Zustand der Observer inkosistent zu seinem eigenen machen könnte.
- Nachdem ein ConcreteObserver Objekt über eine Änderung im ConcreteSubject informiert wurde, kann ein ConcreteObserver Objekt weitere Informationen von ConcreteSubject holen.



Observer-Muster: Vor- und Nachteile

- **Vorteile:**
  - Subjekte und Beobachter können unabhängig variiert werden.
  - Subjekt und Beobachter sind auf abstrakte und minimale Art lose gekoppelt. Das beobachtete Objekt braucht keine Kenntnis über die Struktur seiner Beobachter zu besitzen sondern kennt diese nur über die Beobachter-Schnittstelle.
  - Ein abhängiges Objekt erhält die Änderungen automatisch.
- **Nachteile:**
  - Änderungen am Objekt führen bei großer Beobachteranzahl zu hohen Änderungskosten. Einerseits informiert das Subjekt jeden Beobachter, auch wenn dieser die Änderungsinformation nicht benötigt. Zusätzlich können die Änderungen weitere Änderungen nach sich ziehen und so einen unerwartet hohen Aufwand verursachen.
  - Ruft ein Beobachter während der Bearbeitung einer gemeldeten Änderung wiederum Änderungsmethoden des Subjektes auf, kann es zu Endlosschleifen kommen.
  - Der Mechanismus liefert keine Information darüber, was sich geändert hat. Die daraus resultierende Unabhängigkeit der Komponenten kann sich allerdings auch als Vorteil herausstellen.

Observer-Muster: Bekannte Verwendungen

- **Bekannte Verwendungen**
  - Smalltalk Model/View/Controller
    (Model = Subject, View = Observer) [siehe später]
  - ET++ (Erich Gamma)
  - User Interface Toolkits: InterViews, Andrew Toolkit, Unidraw
    java.util-Klassen Observer, Observable
  - JavaBeans-Framework (PropertyChangeListener, PropertyChangeSupport)

Weitere Entwurfsmuster, Architekturmuster und Antimuster

- **Weitere Entwurfsmuster**
  siehe Vorlesung „Objekt-orientierte Software-Entwicklung

- **Architekturmuster** beschreiben Konzepte für die gesamte *Architektur* eines Systems.

- **Wichtige Beispiele** sind
  - Model-View-Controller
  - Layers
  - Pipes and Filters
  - Broker

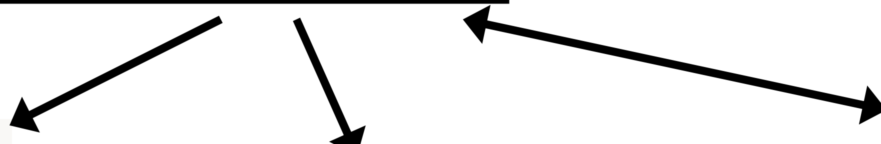
- **Anti-Muster** beschreiben unerwünschte Situationen, die **nicht** auftreten sollten.

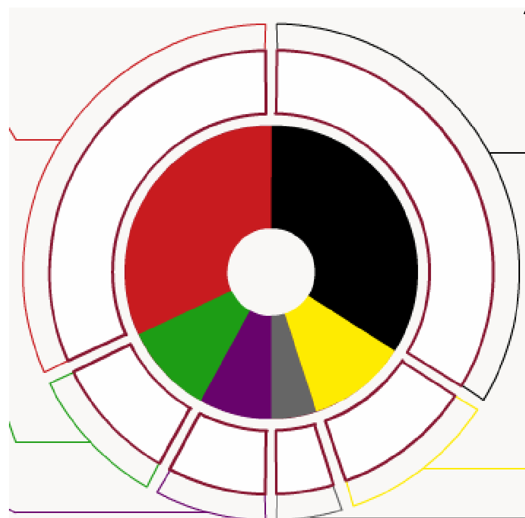
Model-View-Controller

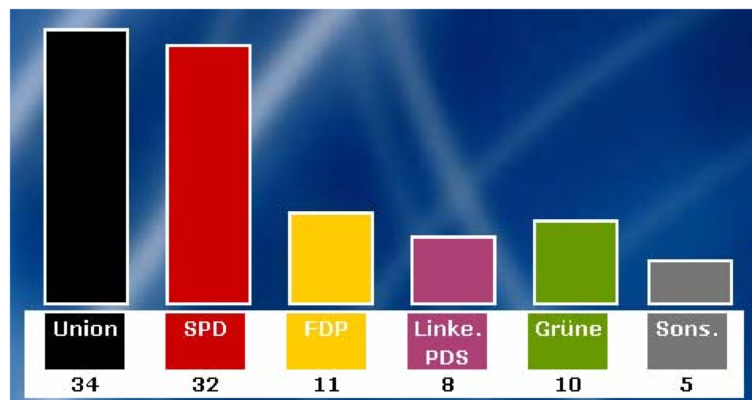
Beispiel: Informationssystem für Wahlen, das verschiedene Sichten auf Prognosen und Ergebnisse bietet.

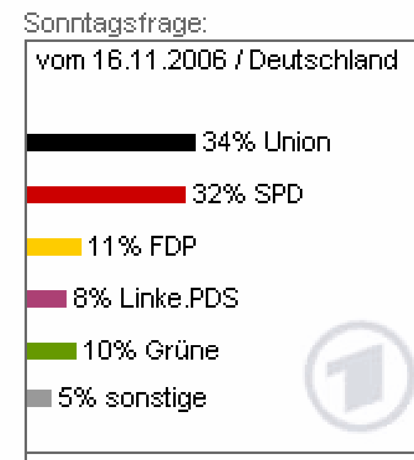
CDU:	**34%**
SPD:	**32%**
FDP:	**11%**
GRUENE:	**10%**
Links-PDS:	**09%**
Sonstige:	**03%**

Sonntagsfrage Bund
Infratest-dimap
2006-11-16
1.000 Befragte
von 2006-11-14
bis 2006-11-15









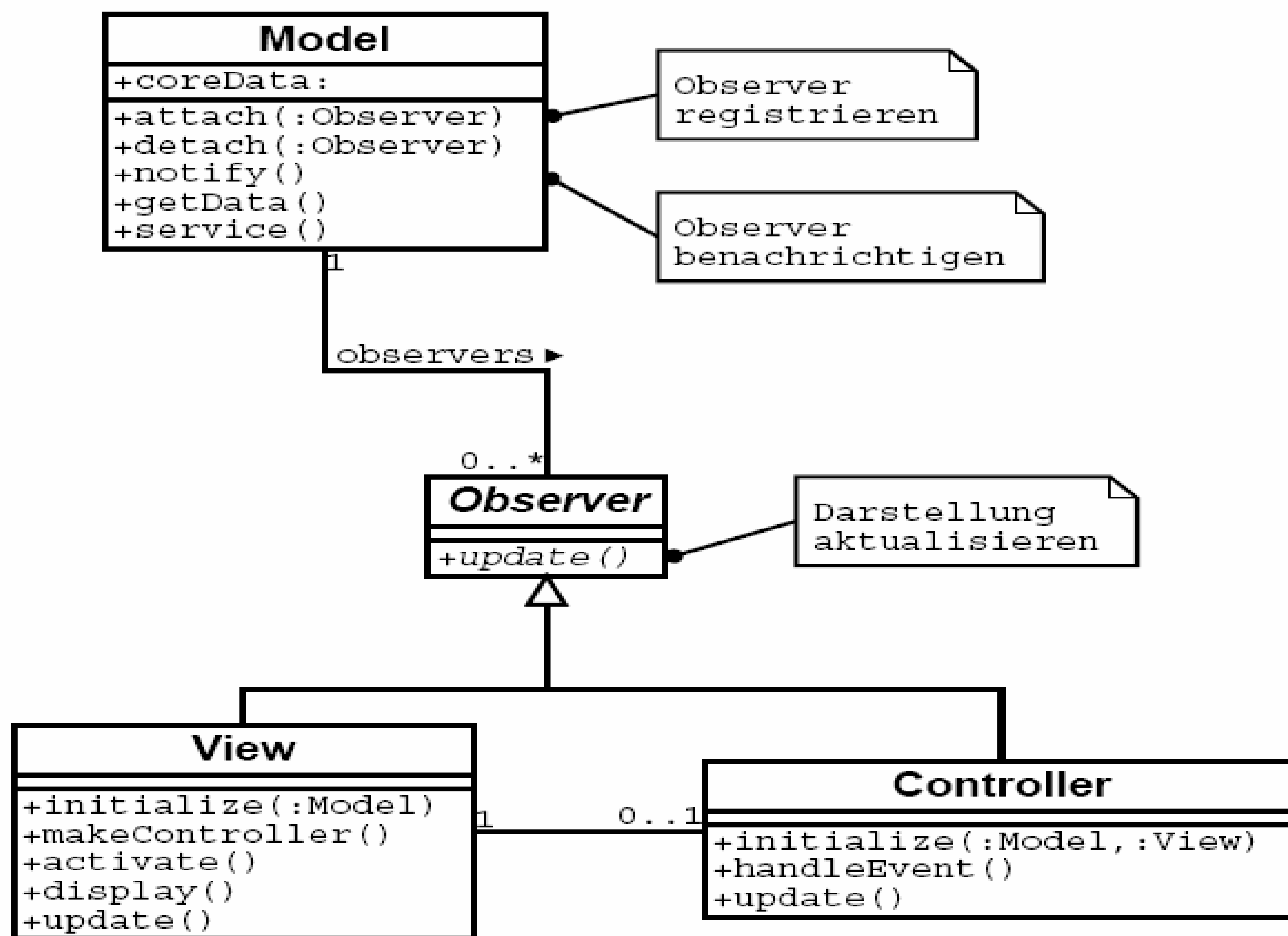
Model-View-Controller

- Das *Model-View-Controller*-Muster ist eines der bekanntesten und verbreitetesten Muster für die Architektur **interaktiver** Systeme.
- **Problem**

  Benutzerschnittstellen sind besonders häufig von Änderungen betroffen.
  - Wie kann ich dieselbe Information auf verschiedene Weise darstellen?
  - Wie kann ich sicherstellen, dass Änderungen an den Daten sofort in allen Darstellungen sichtbar werden?
  - Wie kann ich die Benutzerschnittstelle ändern (womöglich zur Laufzeit)?
  - Wie kann ich verschiedene Benutzerschnittstellen unterstützen, ohne den Kern der Anwendung zu verändern?
- **Lösung**

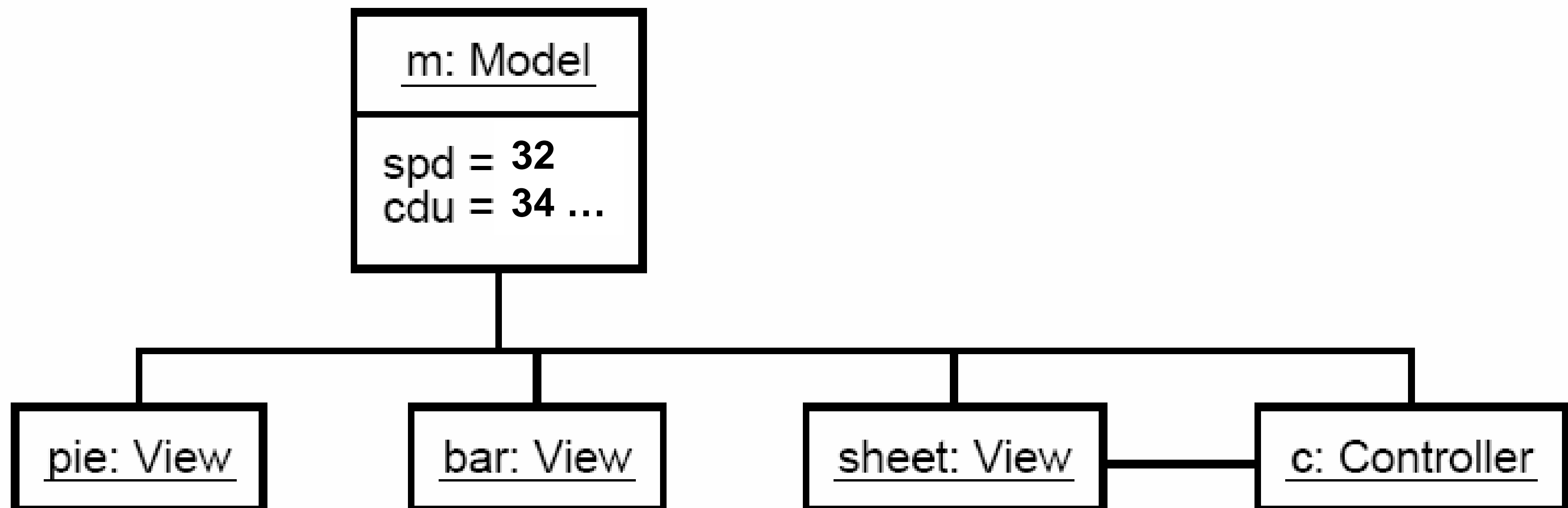
  Das *Model-View-Controller*-Muster trennt eine Anwendung in drei Teile:
  - Das *Modell* **(Model)** ist für die *Verarbeitung* zuständig,
  - Die *Sicht* **(View)** kümmert sich um die *Ausgabe*
  - Die *Kontrolle* **(Controller)** kümmert sich um die *Eingabe*.

Model-View-Controller: Struktur



Model-View-Controller: Struktur

- Bei jedem Modell können sich mehrere *Beobachter* (= Sichten und Kontrollen) *registrieren*.

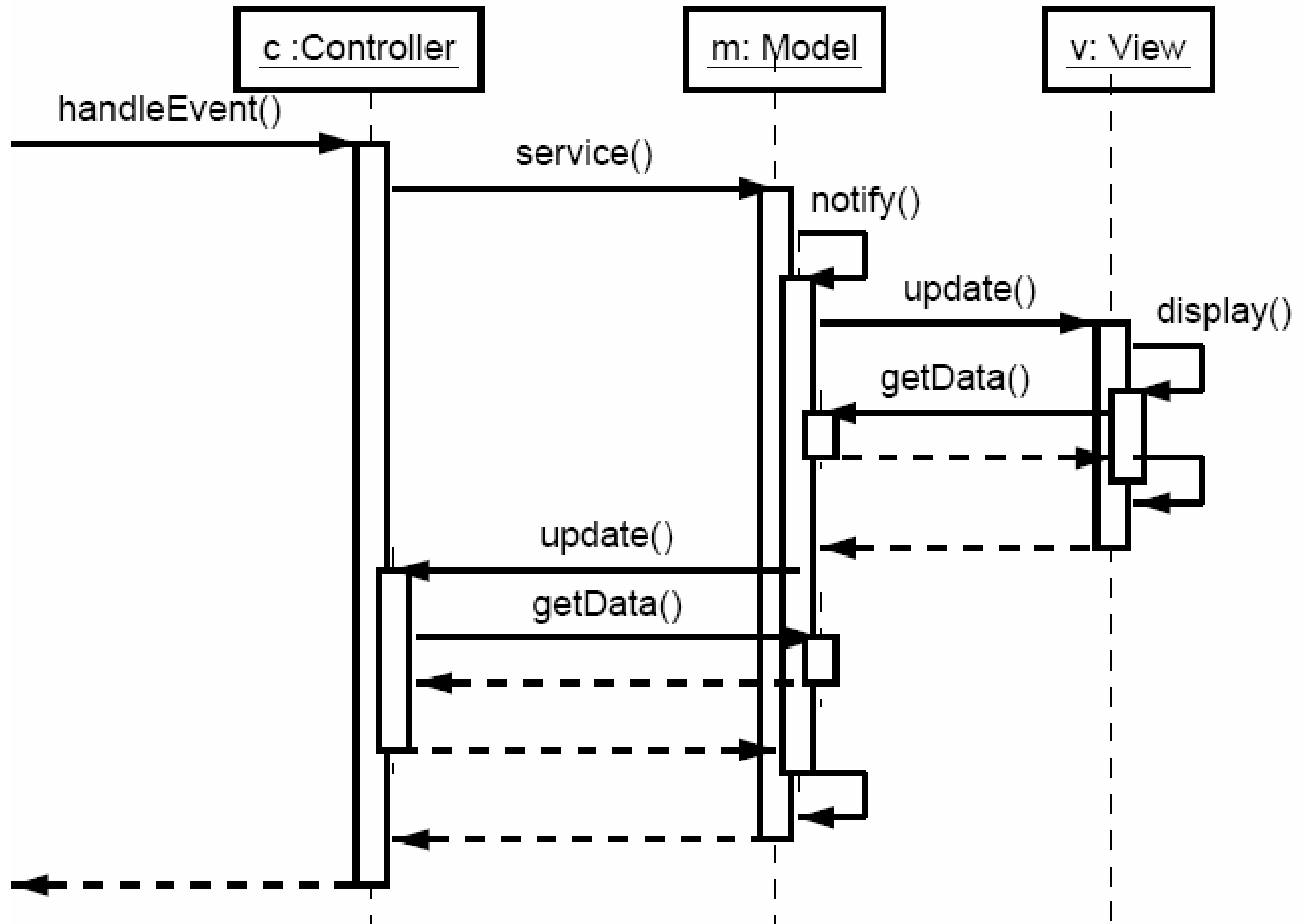


- Bei jeder Änderung des Modell-Zustands werden die registrierten Beobachter *benachrichtigt;* sie bringen sich dann auf den neuesten Stand.

Model-View-Controller

- Das **Modell (model)** verkapselt Kerndaten und Funktionalität. Das Modell ist unabhängig von einer bestimmten Darstellung der Ausgabe oder einem bestimmten Verhalten der Eingabe.
- Die **Sicht (view)** zeigt dem Benutzer Informationen an. Es kann mehrere Sichten pro Modell geben.
- Die **Kontrolle (controller)** verarbeitet Eingaben und ruft passende Dienste der zugeordneten Sicht oder des Modells auf. Jede Kontrolle ist einer Sicht zugeordnet; es kann mehrere Kontrollen pro Modell geben.

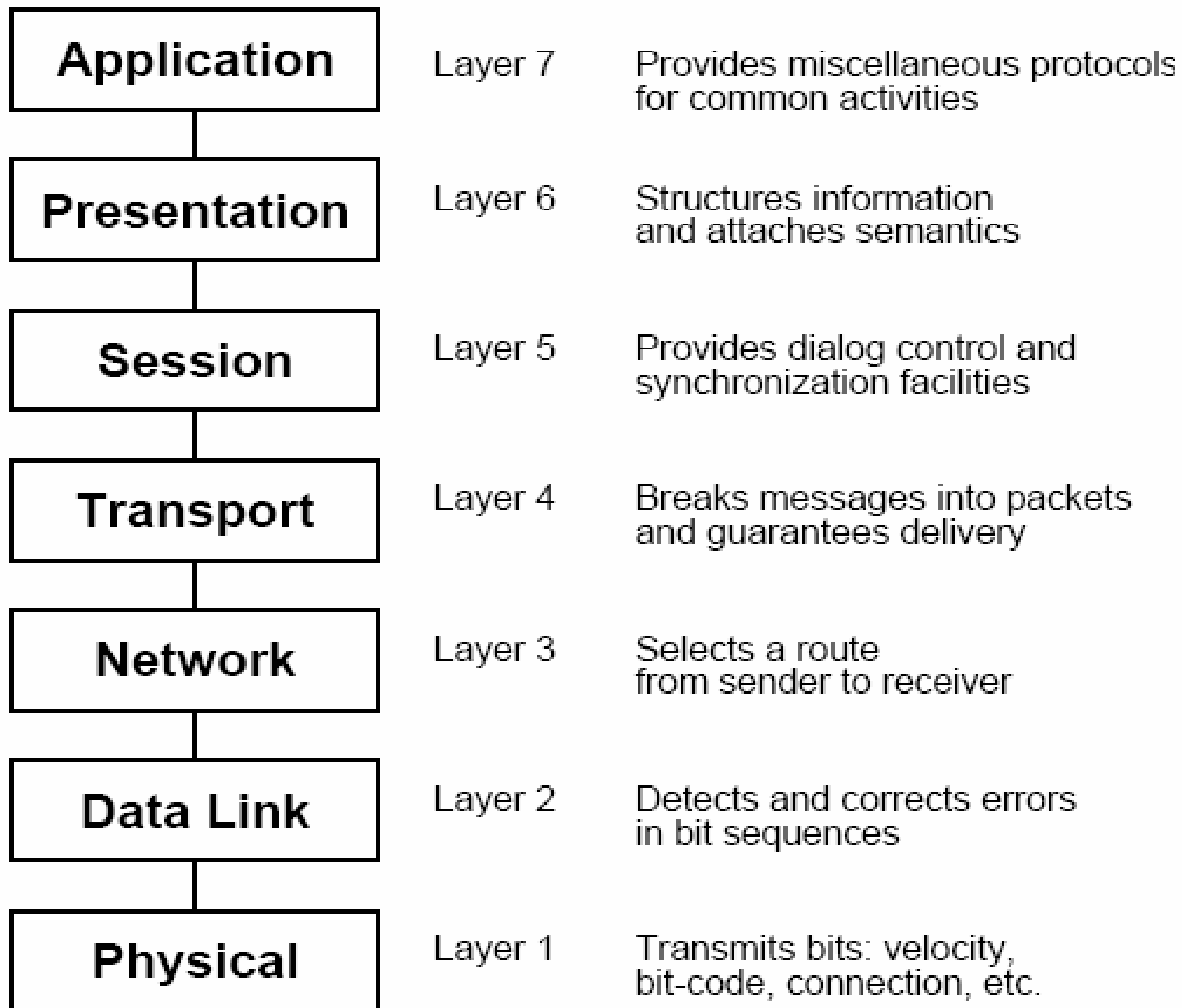
Model-View-Controller: Dynamisches Verhalten



Model-View-Controller

- **Vorteile**
  - Mehrere Sichten desselben Modells
  - Synchrone Sichten
  - "Ansteckbare" Sichten und Kontrollen
- **Nachteile**
  - Erhöhte Komplexität
  - Starke Kopplung zwischen Modell und Sicht
  - Starke Kopplung zwischen Modell und Kontrollen (kann mit
  - Command-Muster umgangen werden)
- **Bekannte Einsatzgebiete**: Viele Anwendungssysteme (-> SWEP), GUI-Bibliotheken, Smalltalk, Microsoft Foundation Classes

Schichten und Abstraktionen: Layers



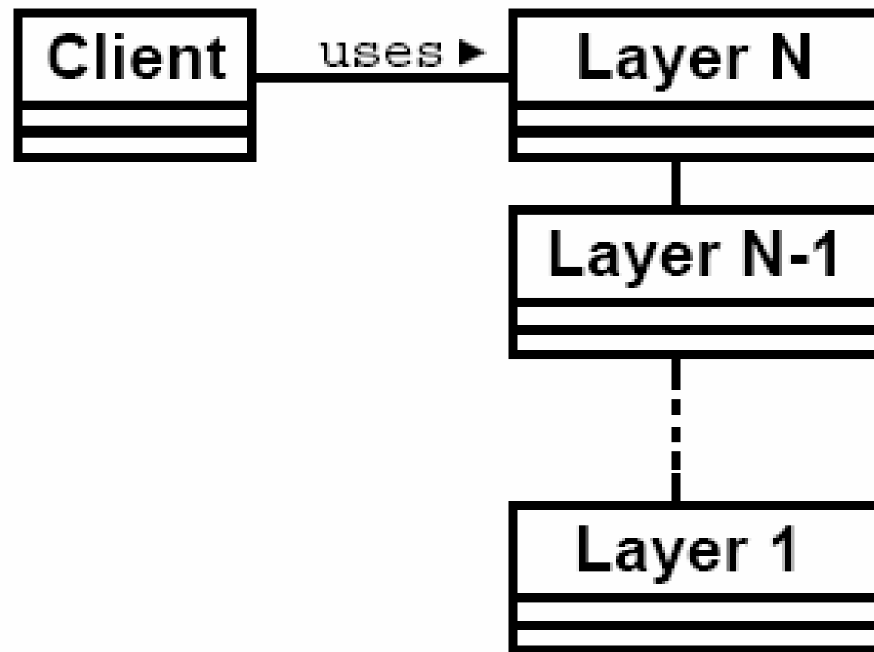
Schichten und Abstraktionen: Layers

- Das **Layers-Muster** trennt eine Architektur in verschiedene Schichten, von denen jede eine Unteraufgabe auf einer bestimmten Abstraktionsebene realisiert.
- **Problem** Ein System konstruieren, das
  - Aktivitäten auf niederer Ebene wie Hardware-Ansteuerung, Sensoren, Bitverarbeitung sowie
  - Aktivitäten auf hoher Ebene wie Planung, Strategien und Anwenderfunktionalität vereinigt, wobei die Aktivitäten auf hoher Ebene durch Aktivitäten der niederen Ebenen realisiert werden.
- Dabei sind folgende Ziele zu berücksichtigen:
  - Änderungen am Quellcode sollen möglichst wenige Ebenen betreffen
  - Schnittstellen sollten stabil (und möglicherweise standardisiert) sein
  - Teile (= Ebenen) sollten austauschbar sein
  - Jede Ebene soll separat realisierbar sein

Schichten und Abstraktionen: Layers

Lösung

- Das *Layers*-Muster gliedert ein System in zahlreiche *Schichten*.
- Jede Schicht schützt die unteren Schichten vor direktem Zugriff durch höhere Schichten.
  - Insbesondere stellt jede Schicht *j* Dienste bereit für Schicht *j+1* und delegiert Aufgaben an Schicht *j-1*.



Layers: Dynamisches Verhalten

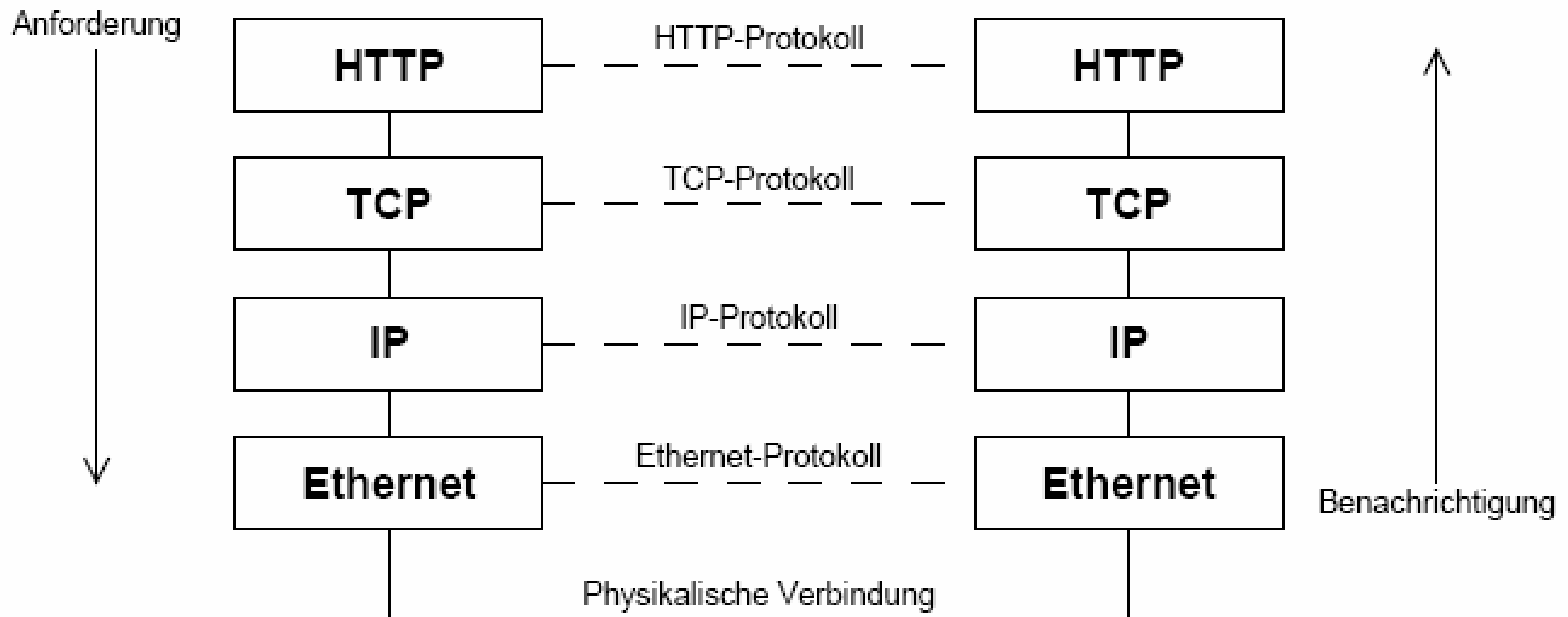
Zwei vorherrschende Szenarien:

- **Top-Down Anforderung**
  - Eine Anforderung des Benutzers wird von der obersten Schicht entgegengenommen; diese resultiert in Anforderungen der unteren Schichten bis hinunter auf die unterste Ebene.
  - Ggf. werden die Ergebnisse der unteren Schichten wieder nach oben weitergeleitet, bis das letzte Ergebnis an den Benutzer zurückgegeben wird.
- **Bottom-Up Anforderung**
  - Hier empfängt die unterste Schicht ein Signal, das an die oberen Schichten weitergeleitet wird;
  - schließlich benachrichtigt die oberste Schicht den Benutzer.

Layers: Dynamisches Verhalten

- **Protokollstack**

  In diesem Szenario kommunizieren zwei *n*-Schichten-Stacks miteinander.
  - Eine Anforderung wandert durch den ersten Stack hinunter, wird übertragen und schließlich als Signal vom zweiten Stack empfangen.
  - Jede Schicht verwaltet dabei ihr eigenes Protokoll.
- **Beispiel – TCP/IP-Stack**

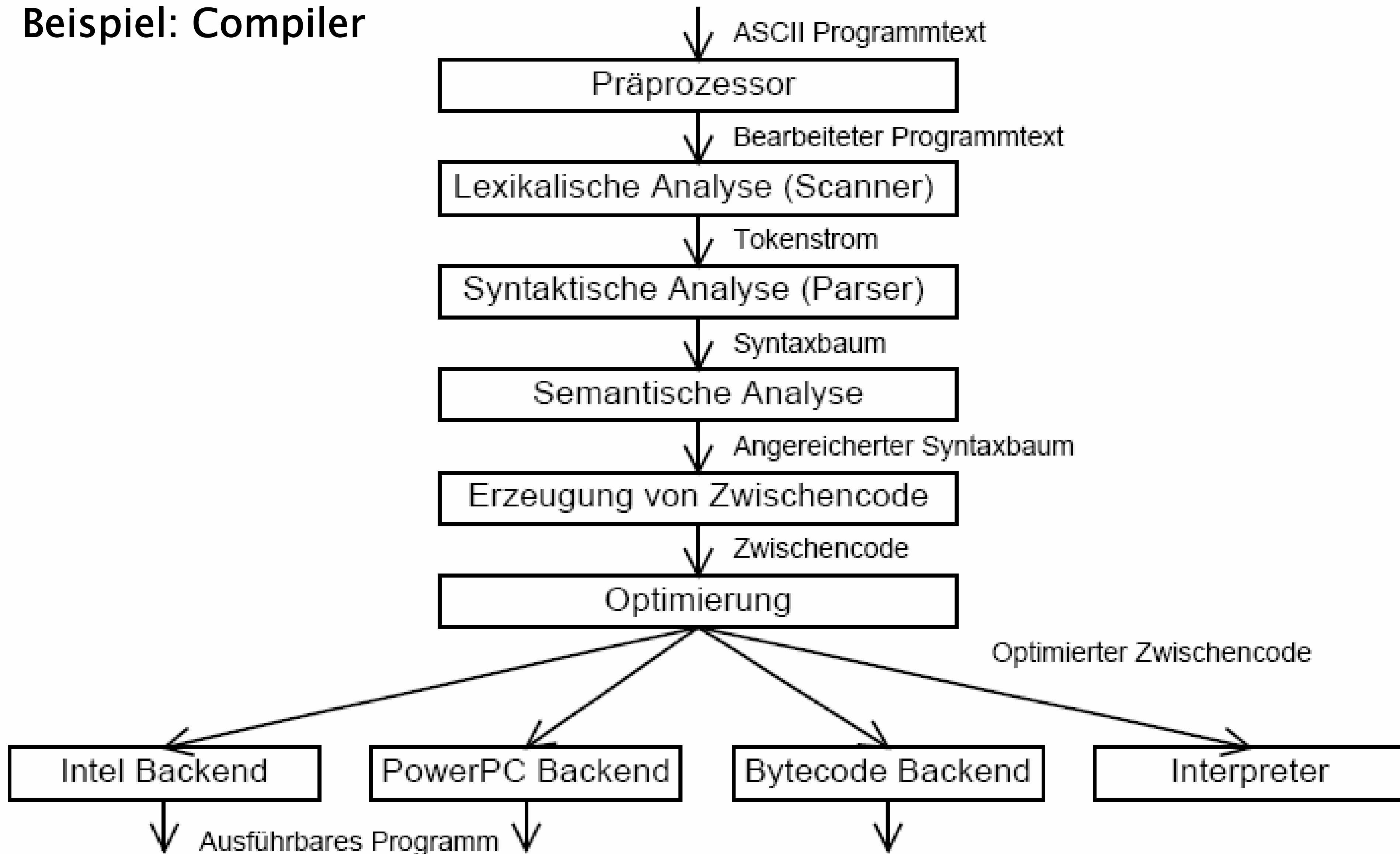


Layers

- **Vorteile**
  - Wiederverwendung und Austauschbarkeit von Schichten
  - Unterstützung von Standards
  - Einkapselung von Abhängigkeiten
- **Nachteile**
  - Geringere Effizienz
  - Mehrfache Arbeit (z.B. Fehlerkorrektur)
  - Schwierigkeit, die richtige Anzahl Schichten zu bestimmen
- **Bekannte Einsatzgebiete**:
  - Application Programmer Interfaces (APIs)
  - Datenbanken
  - Betriebssysteme
  - Kommunikation. . .

Verarbeitung von Datenströmen: Pipes and Filters

Beispiel: Compiler



Verarbeitung von Datenströmen: Pipes and Filters

- Das **Pipes and Filters-Muster** bietet eine Struktur für Systeme, die einen Datenstrom verarbeiten:
  - Jede Verarbeitungsstufe wird durch eine Filter-Komponente realisiert
  - Pipes (Kanäle) leiten die Daten von Filter zu Filter
- **Problem**

  Ein System konstruieren, das einen **Strom von Eingabedaten verarbeiten** oder umwandeln soll, wobei
  - das System **nicht als monolithischer Block** gebaut werden soll
  - **Austausch oder Neukombimation von Teilen** möglich sein soll,
  - nicht aufeinander folgende Verarbeitungsstufen **entkoppelt** sein sollen
  - **parallele Verarbeitung möglich** sein soll.
- **Lösung**

  Das Pipes and Filters-Muster teilt die Aufgaben des Systems in **mehrere Verarbeitungsstufen**, die **durch den Datenfluss** durch das System **verbunden** sind:
  - Die Ausgabe einer Stufe ist die Eingabe der nächsten Stufe.
  - Jede Verarbeitungsstufe wird durch einen **Filter** realisiert. Ein Filter kann Daten inkrementell verarbeiten und liefern – er kann also mit der Ausgabe beginnen, noch bevor er die Eingabe komplett eingelesen hat. Dies ist eine wichtige Voraussetzung für paralleles Arbeiten.
  - Die Eingabe des Systems ist eine **Datenquelle**, die Ausgabe des Systems eine **Datensenke.** Datenquelle, Filter und Datensenke sind durch **Pipes** verbunden. Die Folge von Verarbeitungsstufen heißt **Pipeline.**

Pipes and Filters: Teilnehmer

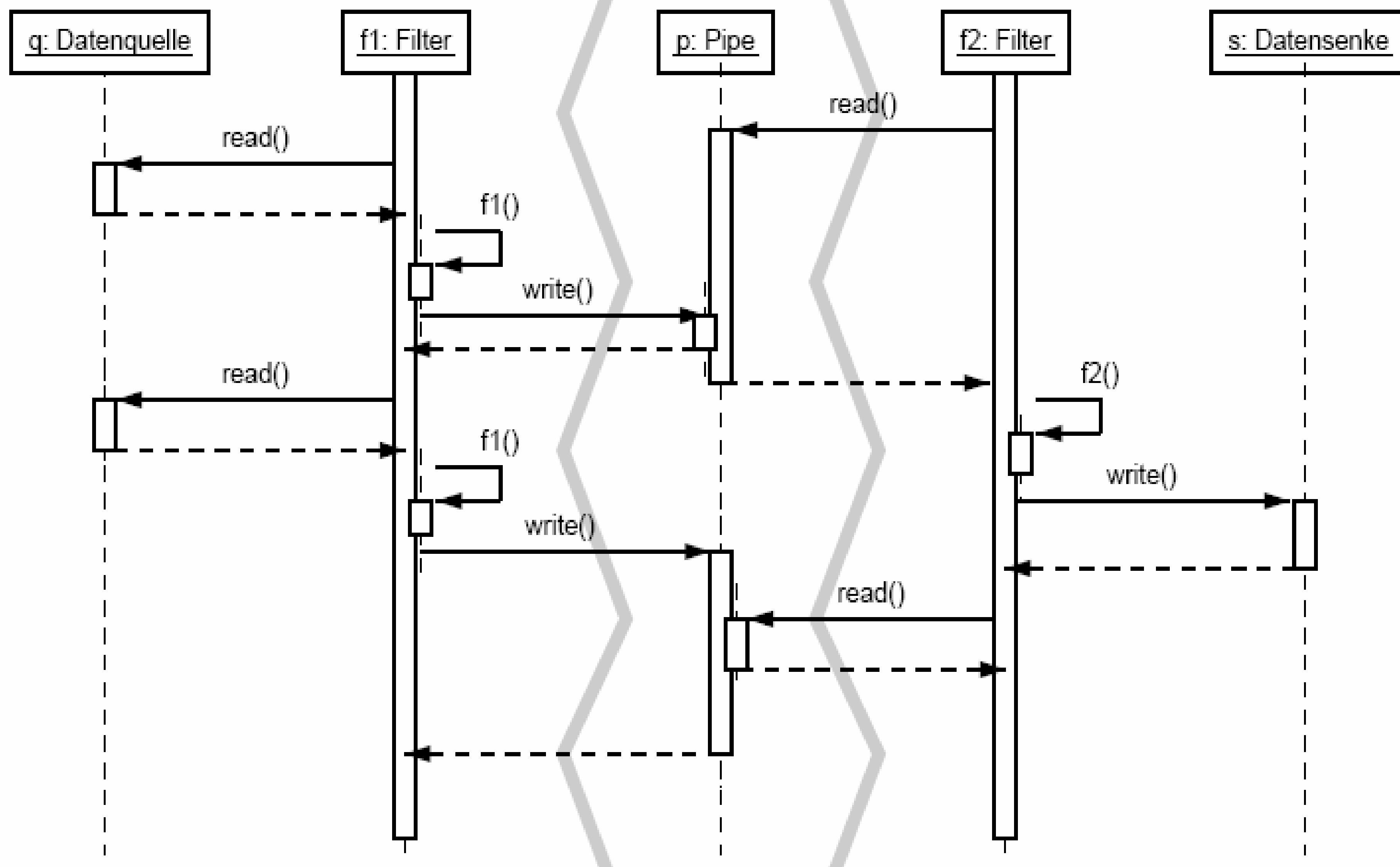
- **Filter**
  - Holt Eingabedaten, wendet eine Funktion auf seine Eingabedaten an und liefert die Ausgabedaten.
  - Ein Filter kann auf dreierlei Weise mit den Daten umgehen:
    - Er kann die Daten **anreichern**, indem er weitere Informationen berechnet und hinzufügt,
    - Er kann die Daten **verfeinern**, indem er Information konzentriert oder extrahiert
    - Er kann die Daten **verändern**, indem er sie in eine andere Darstellung überführt.
  - Ein Filter kann auf verschiedene Weise *aktiv* werden:
    - Die folgende Pipeline holt Daten aus dem Filter
    - Die vorhergehende Pipeline schickt Daten in den Filter
    - Meistens ist der Filter jedoch *selbst aktiv* – er holt Daten aus der vorhergehenden Pipeline und schickt Daten in die folgende Pipeline.

Pipes and Filters: Teilnehmer

- **Pipe**
  - Eine Pipe verbindet Filter miteinander; sie verbindet auch die Datenquelle mit dem ersten Filter und den letzten Filter mit der Datensenke.
  - Eine Pipe übermittelt Daten, puffert Daten und synchronisiert aktive Nachbarn.
- **Datenquelle, Datensenke**
  - Diese Komponenten sind die *Endstücke* der Pipeline und somit die Verbindung zur Außenwelt. Sie übermitteln Daten an/aus Pipeline.
  - Eine Datenquelle kann entweder *aktiv* sein (dann reicht sie von sich aus Daten in die Pipeline) oder *passiv* (dann wartet sie, bis der nächste Filter Daten anfordert).
  - Analog kann die Datensenke aktiv Daten anfordern oder passiv auf Daten warten.

Pipes and Filters: Teilnehmer

Zwei *aktive* Filter sind durch eine Pipe verbunden; beide Filter arbeiten parallel.

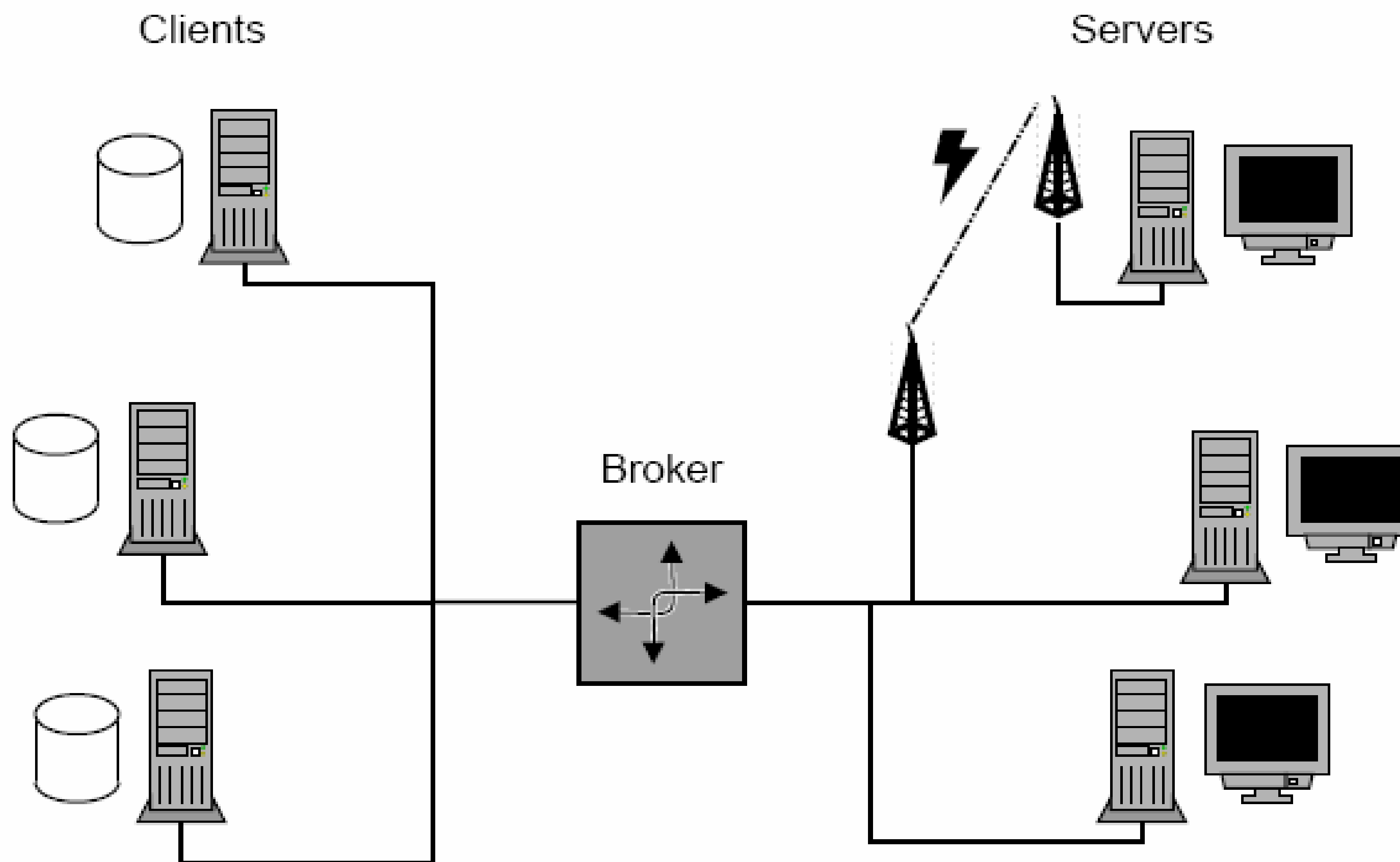


Pipes and Filters

- **Vorteile**
  - Kein Speichern von Zwischenergebnissen (z.B. in Dateien) notwendig
  - Flexibilität durch Austauschen von Filtern
  - Rekombination von Filtern (z.B. in UNIX)
  - Filter können als Prototypen erstellt werden
  - Parallel-Verarbeitung möglich
- **Nachteile**
  - Gemeinsamer Zustand (z.B. Symboltabelle in Compilern) ist teuer und unflexibel.
  - Effizienzsteigerung durch Parallelisierung oft nicht möglich (z.B. da Filter aufeinander warten oder nur ein Prozessor arbeitet)
  - Overhead durch Datentransformation (z.B. UNIX: Alle Daten müssen in/aus Text konvertiert werden)
  - Fehlerbehandlung ist schwer zu realisieren
- **Bekannte Einsatzgebiete**
  - Compiler, UNIX

Vermitteln von Ressourcen: Broker

- **Beispiel: Rechenzeit verteilen:** Systeme, bei denen Rechenzeit anderen Anwendern zur Verfügung gestellt werden kann.

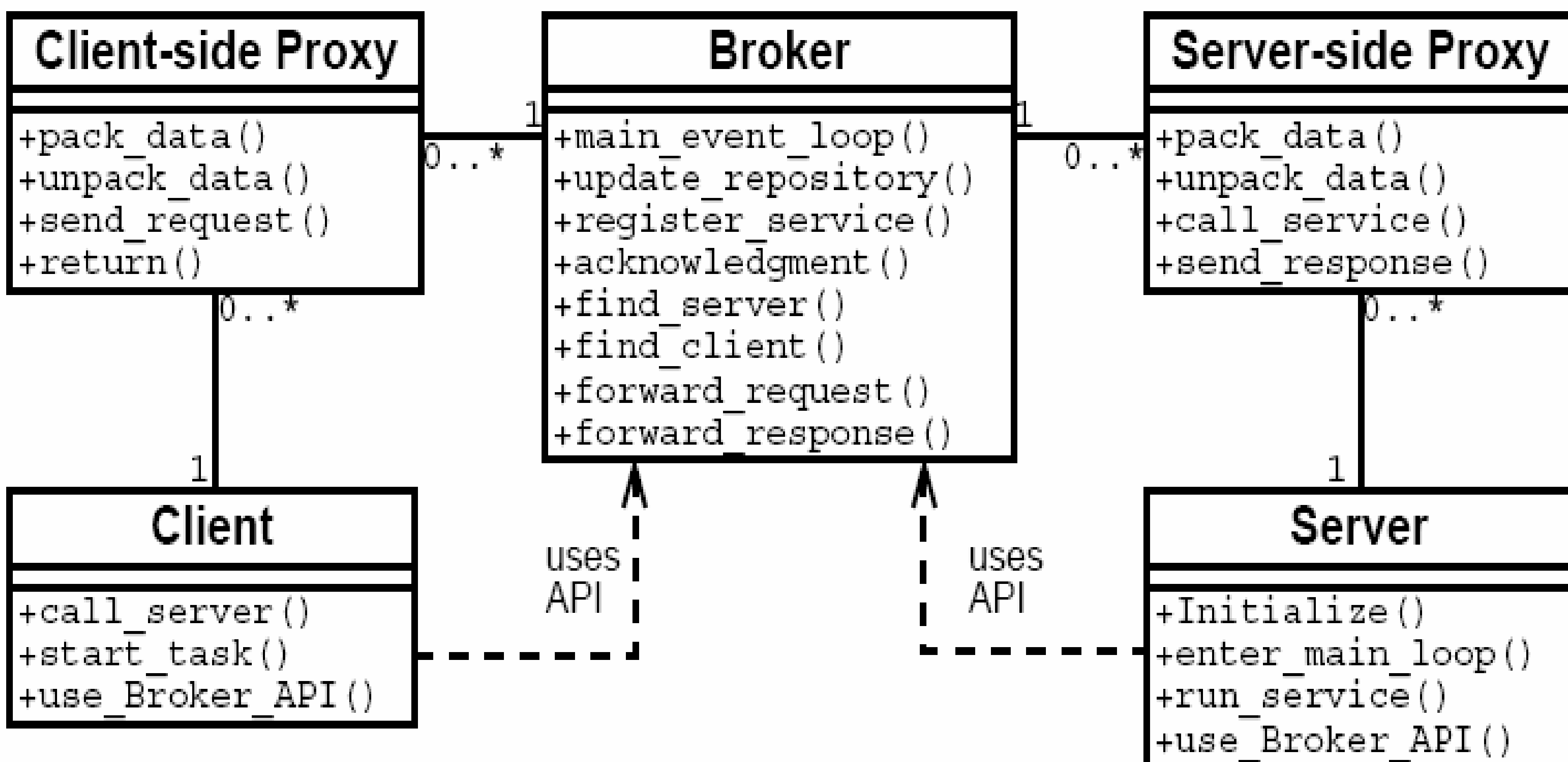


Broker

- **Problem**
  - Ein komplexes Software-System soll als **Menge von entkoppelten, zusammenarbeitenden Komponenten** realisiert werden (und nicht als eine monolithische Anwendung).
  - Die Komponenten sollen **verteilt** sein; dies soll jedoch möglichst **transparent** gestaltet werden.
  - **Zur Laufzeit** können **neue Komponenten hinzukommen** oder wegfallen.
- **Lösung**

  Ein **Broker (Vermittler)** vermittelt zwischen Dienstanbietern (Server) und Dienstanwendern (Clients).
  - **Dienstanbieter** melden sich beim Broker an und machen ihre Dienste für
  - Anwender verfügbar.
  - **Anwender** senden Dienstanforderungen an den Broker, der sie an einen geeigneten Anbieter weiterleitet.
  - Zu den **Aufgaben des Brokers** gehört es,
    - den passenden Anbieter zu finden
    - Anfragen des Anwenders an den passenden Anbieter weiterzuleiten
    - die Antwort des Anbieters an den Anwender zurückzusenden

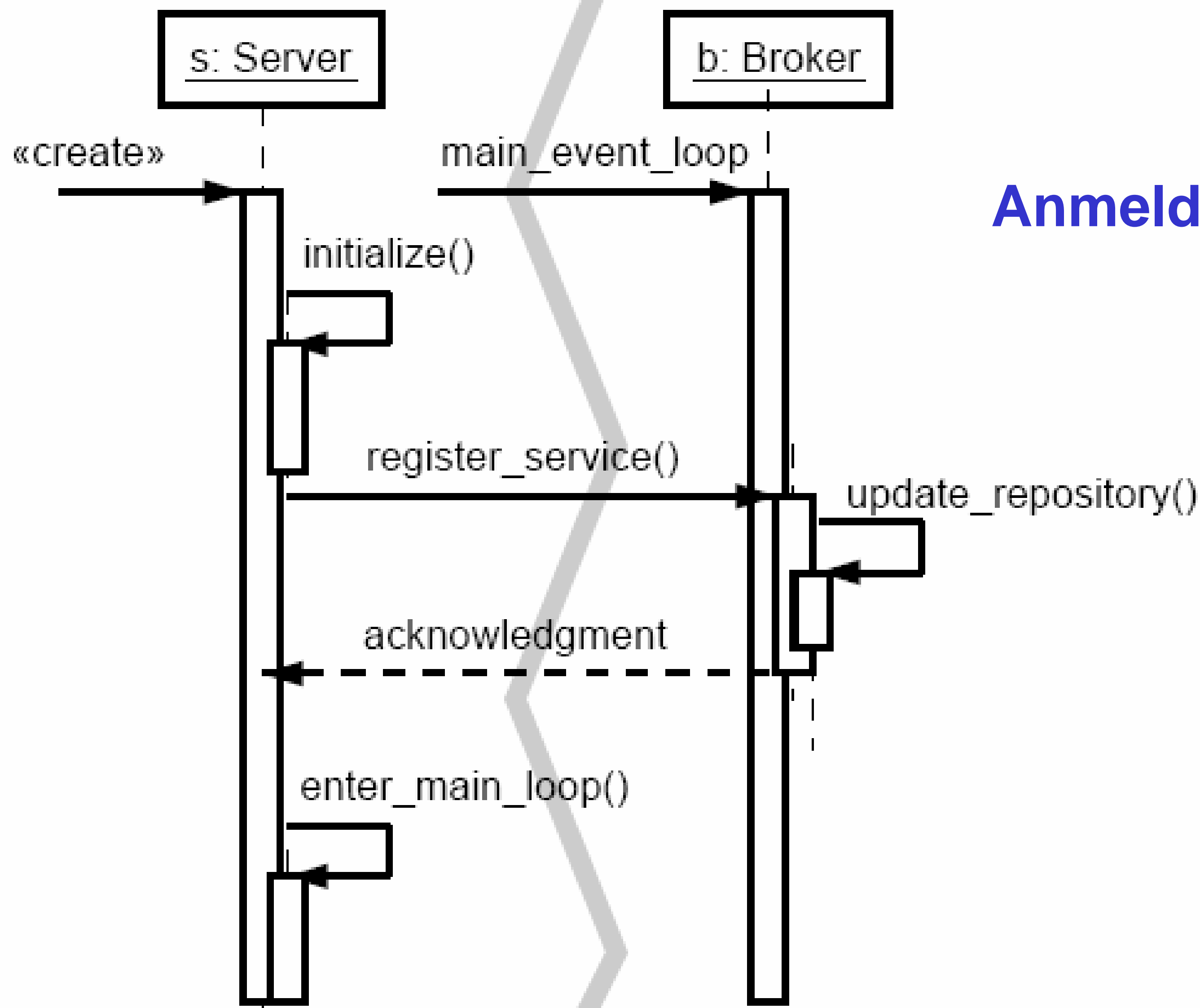
Broker: Struktur



Broker: Teilnehmer

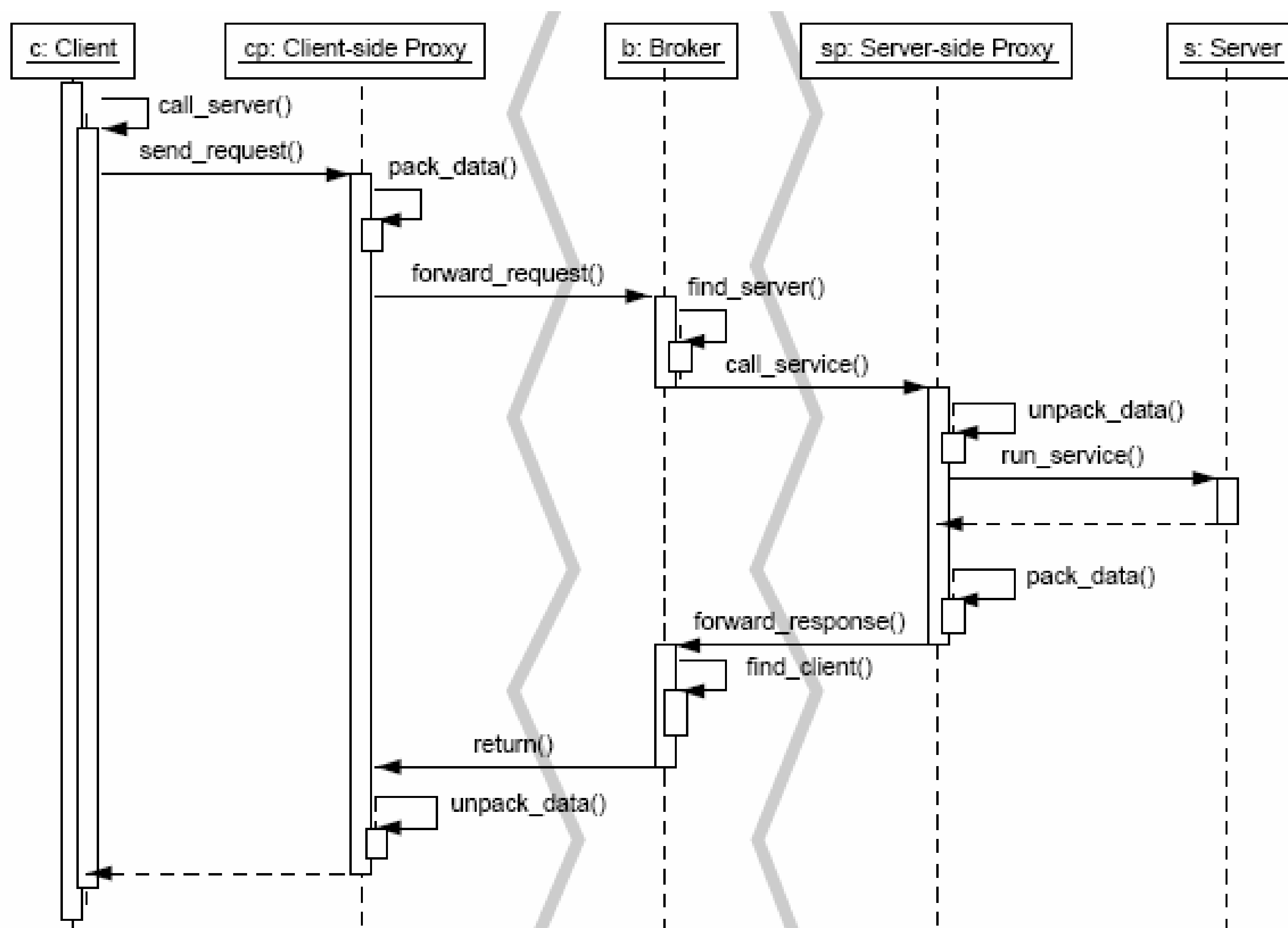
- **Server**
  - Der **Server bietet Dienste an**, indem er
    - Dienste implementiert,
    - sich beim lokalen Broker anmeldet und
    - mit dem Client über einen Server-side Proxy kommuniziert
- **Client**
  - Der **Client macht die Dienste dem Benutzer zugänglich**, indem er
    - die Funktionalität für die Benutzer realisiert und
    - Anfragen an Server mittels Client-side Proxy sendet.
- **Proxies**
  - Ein **Proxy vermittelt zwischen Client (bzw. Server) und dem Broker.**
  - Insbesondere **kapselt** der Proxy die **Kommunikation** zum und vom Broker ein (einschließlich Netzzugriffen und Aufbereiten von Daten).

Broker: Dynamisches Verhalten



Anmelden eines Servers

Broker: Anfrage über Broker



Broker

- **Vorteile**
  - Transparente verteilte Dienste
  - Änderbarkeit und Erweiterbarkeit von Komponenten
  - Interoperabilität zwischen verschiedenen Systemen
  - Wiederverwendbarkeit von Diensten
- **Nachteile**
  - Effizienz beschränkt durch Indirektion und Kommunikation
  - Test und Fehlersuche im Gesamt-System sind schwierig
  - (andererseits kann eine neue Anwendung auf bewährten Diensten aufsetzen)

- **Bekannte Einsatzgebiete**
  - Common Object Request Broker Architecture (CORBA)
  - Microsoft .NET, Suns J2EE
  - WWW

Anti-Muster

Treten die folgenden Muster in der Software-Entwicklung auf, liegen schwerwiegende Mängel in Entwurf oder Programmierung vor:

- **The Blob.**
  - Ein Objekt (”Blob“) enthält den Großteil der Verantwortlichkeiten, während die meisten anderen Objekte nur elementare Daten speichern oder elementare Dienste anbieten.
  - **Lösung**: Code neu strukturieren (Refactoring!).
- **The Golden Hammer.**
  - Ein bekanntes Verfahren (”Golden Hammer“) wird auf alle möglichen Probleme angewandt: *Mit einem Hammer sieht jedes Problem wie ein Nagel aus.*
  - **Lösung**: Ausbildung verbessern.
- **Alien Spiders (Außerirdische Spinnen)**:
  - Ein Design, das den Namen nicht verdient. Sehr gesprächige, kommunikative Objekte, die sich alle gegenseitig kennen. Überhaupt keine Nutzung von Entwurfsmustern. Bei n Objekten gibt es n*(n-1)/2 Kommunikationspaare.
  - Lösung: Ausbildung verbessern, Muster einsetzen.

Anti-Muster

- **Cut-and-Paste Programming.**
  - Code wird an zahlreichen Stellen wiederverwendet, indem er kopiert und verändert wird: Verteile und beherrsche nicht„. Dies sorgt für ein Wartungsproblem,
  - **Lösung**: Black-Box-Wiederverwendung, Ausfaktorisieren von Gemeinsamkeiten.
- **Spaghetti Code.**
  - Der Code ist weitgehend unstrukturiert; keine Objektorientierung oder Modularisierung; undurchsichtiger Kontrollfluss.
  - **Lösung**:
    - Vorbeugen – Erst entwerfen, dann codieren.
    - Existierenden Spaghetti-Code neu strukturieren (Refactoring!)
- **Mushroom Management.**
  - Entwickler werden systematisch von Endanwendern ferngehalten.
  - **Lösung**: Kontakte verbessern.

Zusammenfassung

- Ein **Muster** ist eine Schablone, die in vielen verschiedenen Situationen eingesetzt werden kann und beschreibt eine vorbildliche, bewährte Lösung für ein ausgewähltes Problem.
- Der **Nutzen** eines Musters liegt in der (programmiersprachen-unabhängigen) **Identifikation (Konzeptualisierung) einer Klasse von Software-Entwicklungsproblemen und der Beschreibung und Diskussion der Vor- und Nachteile einer Lösung** dafür.
- **Entwurfsmuster** (engl. design pattern) sind **Regeln** oder **Richtlinien** (auch Muster), um häufig auftretende Probleme bei der Erstellung eines Programms zu lösen.
- Die „Viererbande" (Gamma, Helm, Vlissides und Johnson) unterscheidet drei Arten von **Entwurfsmustern**: **Erzeugungs-, Struktur- und Verhaltensmuster.** Z.B. ist Observer ein Verhaltensmuster.
- **Architekturmuster** (Garlan, Shaw) beschreiben Konzepte für die gesamte *Architektur* eines Systems. Wichtige Beispiele sind **Model-View-Controller, Layers, Pipes and Filters, Broker.**
- **Anti-Muster** beschreiben unerwünschte Situationen, die **nicht** auftreten sollten.

Literatur

- Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksfahl-King, Shlomo Angel: *Eine Muster-Sprache. Städte, Gebäude, Konstruktion.* Löcker, Wien 1995
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern-orientierte Softwarearchitektur. Ein Pattern-System.* Addison-Wesley-Longman, Bonn 1998
- James O. Coplien: *Advanced C++ Programming Styles and Idioms.* Addison Wesley, 1991
- Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides: Design Patterns. Addison Wesley, 1995
- Mary Shaw und David Garlan: Software Architecture - Perspectives on an Emerging Discipline. Prentice Hall, 1996