Architekturmuster

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Architekturmuster

Nachdem wir Muster auf der Ebene von *Klassen und Objekten* kennengelernt haben, beschäftigen wir uns nun mit typischen Mustern auf einer höheren Ebene: Muster, die die gesamte *Architektur* eines Systems beschreiben.

Wir betrachten die folgenden klassischen *Architekturmuster:*

- Model-View-Controller
- Layers
- Pipes and Filters
- Broker

. . . sowie einige *Anti-Muster*, die *nicht* auftreten sollten.







Model-View-Controller

Das *Model-View-Controller*-Muster ist eins der bekanntesten und verbreitetsten Muster für die Architektur *interaktiver* Systeme.







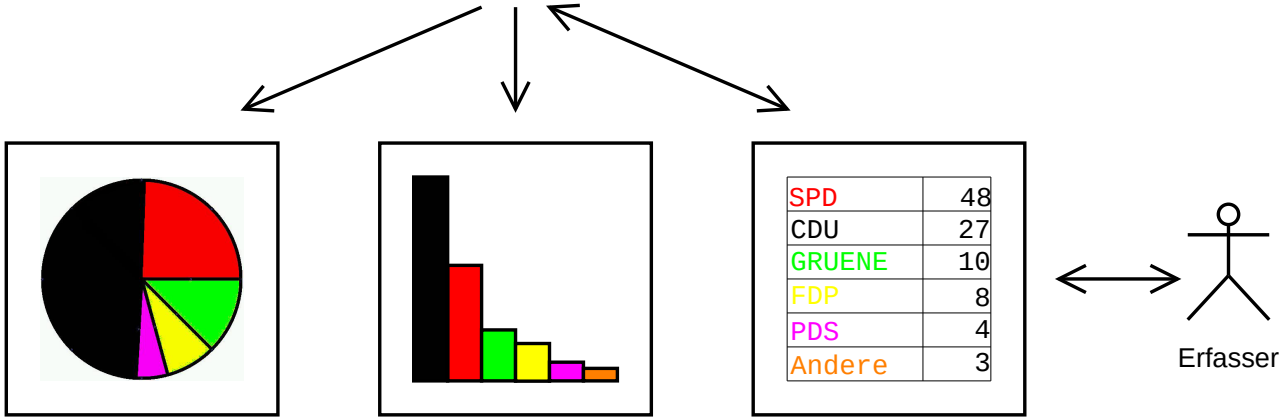
Beispiel: Wahlabend

Wir betrachten ein *Informationssystem für Wahlen*, das verschiedene *Sichten* auf Prognosen und Ergebnisse bietet.

CDU:	48%
SPD:	27%
GRUENE:	10%
FDP:	8%
PDS:	4%
Andere:	3%

Sonntagsfrage Bund
Infratest-dimap
2003-05-16

1.000 Befragte
von 2003-05-13
bis 2003-05-15



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:

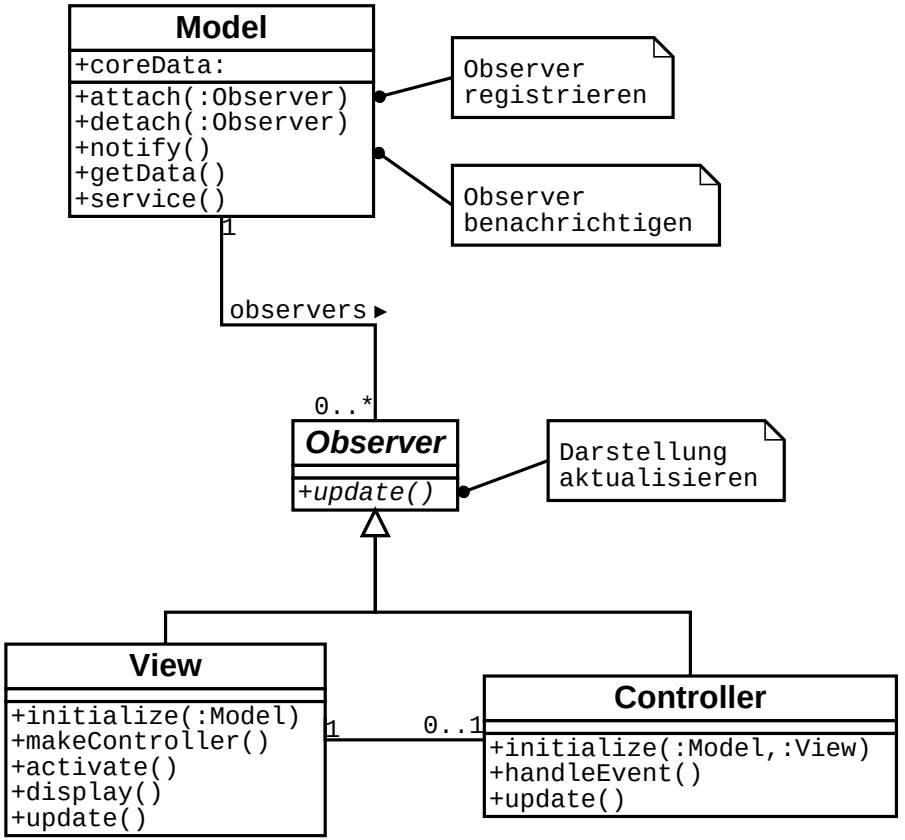
- Das *Modell* (Model) ist für die *Verarbeitung* zuständig,
- Die *Sicht* (View) kümmert sich um die *Ausgabe*
- Die *Kontrolle* (Controller) kümmert sich um die *Eingabe*.





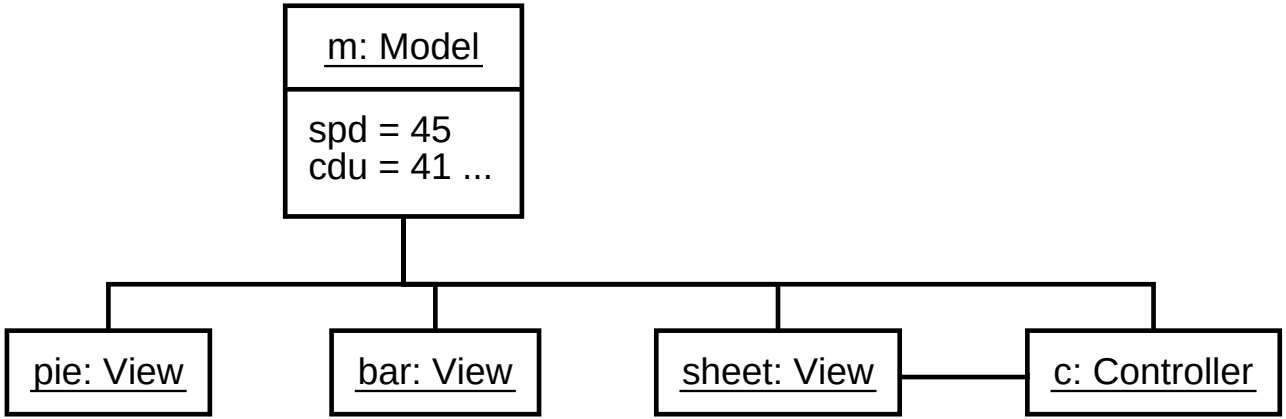


Struktur


Model
+coreData:
+attach(:Observer)
+detach(:Observer)
+notify()
+getData()
+service()
Observer registrieren
Observer benachrichtigen
1
observers
0..*
Observer
+update()
Darstellung aktualisieren
View
+initialize(:Model)
+makeController()
+activate()
+display()
+update()
1
0..1
Controller
+initialize(:Model,:View)
+handleEvent()
+update()

Struktur (2)

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.

Teilnehmer

Das Modell (model) verkapselt Kerndaten und Funktionalität.

Das Modell ist unabhängig von einer bestimmten Darstellung der Ausgabe oder einem bestimmten Verhalten der Eingabe.

Model	*Zusammenarbeit mit*
zuständig für	View, Controller
• Kernfunktionalität der Anwendung • Abhängige Sichten und Kontrollen registrieren • Registrierte Komponenten bei Datenänderung benachrichtigen	







Teilnehmer (2)

Die Sicht (view) zeigt dem Benutzer Informationen an. Es kann mehrere Sichten pro Modell geben.

View	*Zusammenarbeit mit*
zuständig für	Controller, Model
• Dem Anwender Information anzeigen • Ggf. zugeordnete Kontrolle erzeugen • Liest Daten vom Modell	





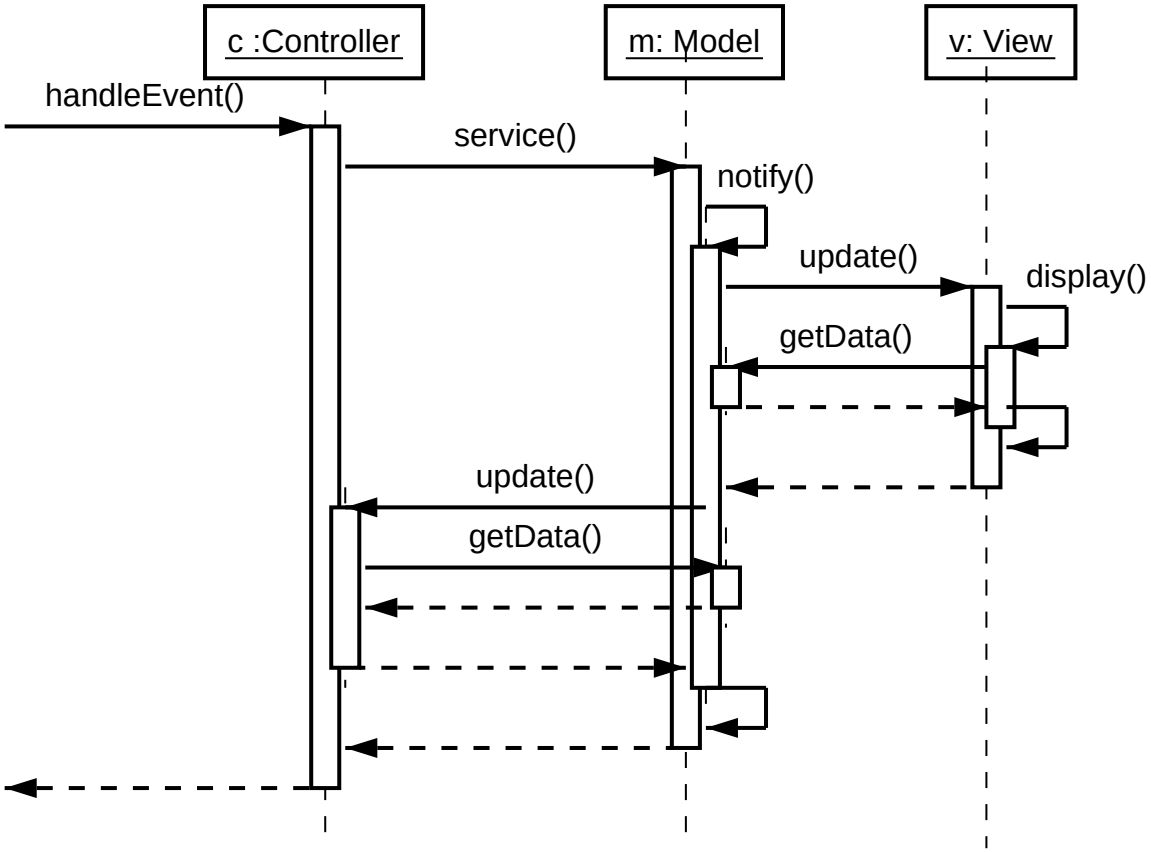


Teilnehmer (3)

Die Kontrolle (controller) verarbeitet Eingaben und ruft passende Dienste der zugeordeten Sicht oder des Modells auf. Jede Kontrolle ist einer Sicht zugeordnet; es kann mehrere Kontrollen pro Modell geben.

Controller	*Zusammenarbeit mit*
zuständig für • Benutzereingaben annehmen • Eingaben auf Dienstanforderungen abbilden (Anzeigedienste der Sicht oder Dienste des Modells)	View, Model

Dynamisches Verhalten



Folgen des Model-View-Controller-Musters

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: GUI-Bibliotheken, Smalltalk, Microsoft Foundation Classes







Schichten und Abstraktionen: Layers

Das *Layers*-Muster trennt eine Architektur in verschiedene *Schichten*, von denen jede eine Unteraufgabe auf einer bestimmten Abstraktionsebene realisiert.







Beispiel: ISO/OSI-Referenzmodell

Netzwerk-Protokolle sind wahrscheinlich die bekanntesten Beispiele für geschichtete Architekturen.

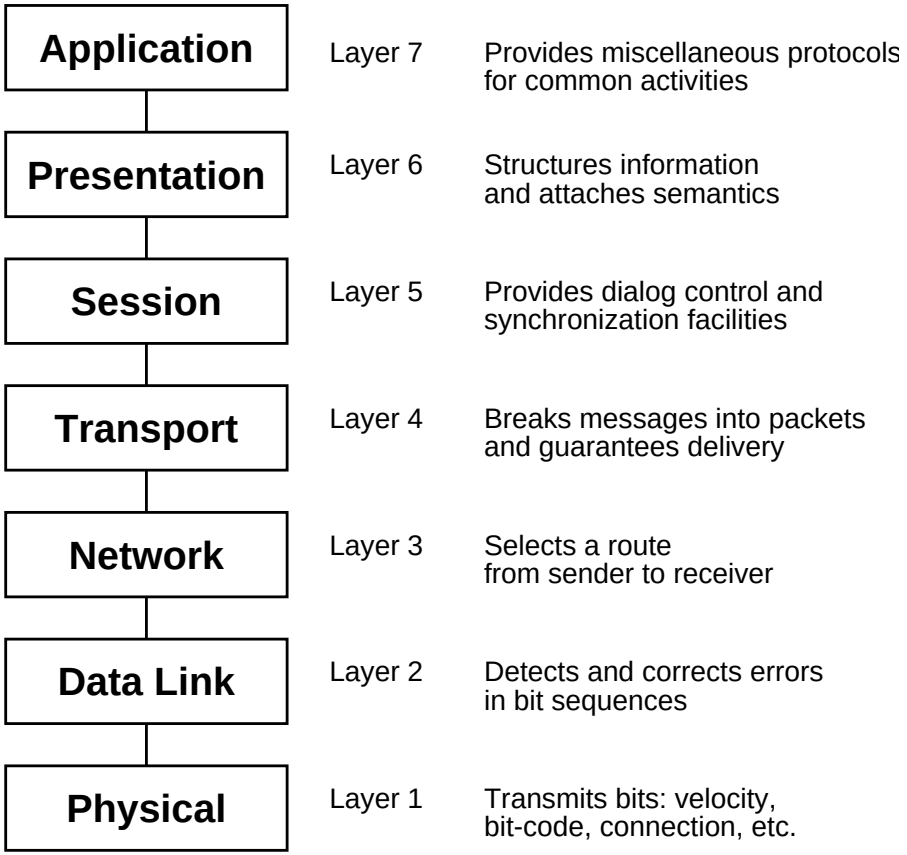
Das *ISO/OSI-Referenzmodell* teilt Netzwerk-Protokolle in 7 Schichten auf, von denen jede Schicht für eine bestimmte Aufgabe zuständig ist:







Beispiel: ISO/OSI-Referenzmodell (2)



Problem

Aufgabe: Ein System bauen, 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.







Problem (2)

Dabei sollen folgende *Ziele* berücksichtigt werden:

- Änderungen am Quellcode sollten möglichst wenige Ebenen betreffen
- Schnittstellen sollten stabil (und möglicherweise standardisiert) sein
- Teile (= Ebenen) sollten austauschbar sein
- Jede Ebene soll separat realisierbar sein



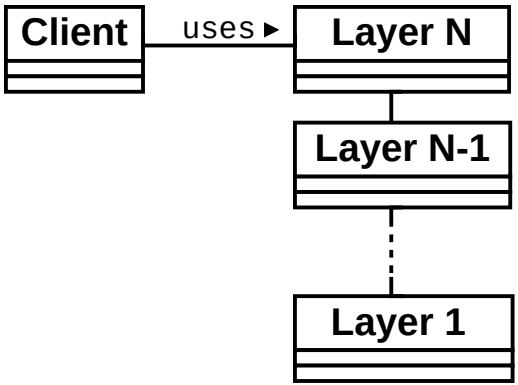




Lösung

Das *Layers*-Muster gliedert ein System in zahlreiche *Schichten*.

Jede Schicht schützt die unteren Schichten vor direktem Zugriff durch höhere Schichten.



Teilnehmer

Schicht j	*Zusammenarbeit mit*
zuständig für • Stellt Dienste bereit für Schicht $j+1$. • Delegiert Aufgaben an Schicht j–1.	Schicht j–1

Dynamisches Verhalten

Es gibt verschiedene weitverbreitete 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.

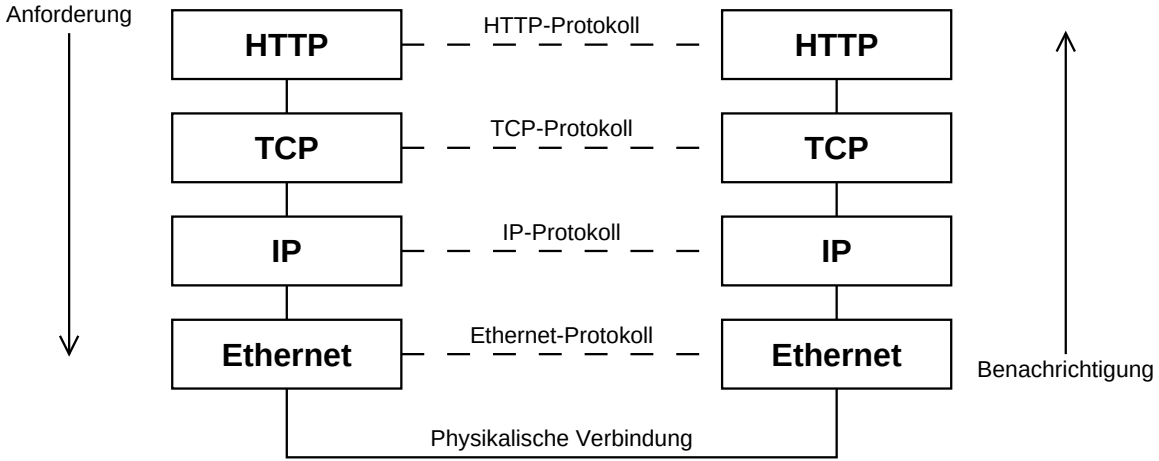
Ggf. werden die Ergebnisse der unteren Schichten wieder nach oben weitergeleitet, bis das letzte Ergebnis an den Benutzer zurückgegeben wird.

Bottom-Up Anforderung Hier empfängt die unterste Schicht ein Signal, das an die oberen Schichten weitergeleitet wird; schließlich benachrichtigt die oberste Schicht den Benutzer.

Dynamisches Verhalten (2)

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:



Folgen des Layers-Musters

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







Einsatzgebiete

Bekannte Einsatzgebiete:

- Application Programmer Interfaces (APIs)
- Datenbanken
- Betriebssysteme
- Kommunikation. . .







Datenstrom verarbeiten: Pipes and Filters

Das *Pipes and Filters*-Muster bietet eine Struktur für Systeme, die einen *Datenstrom* verarbeiten:

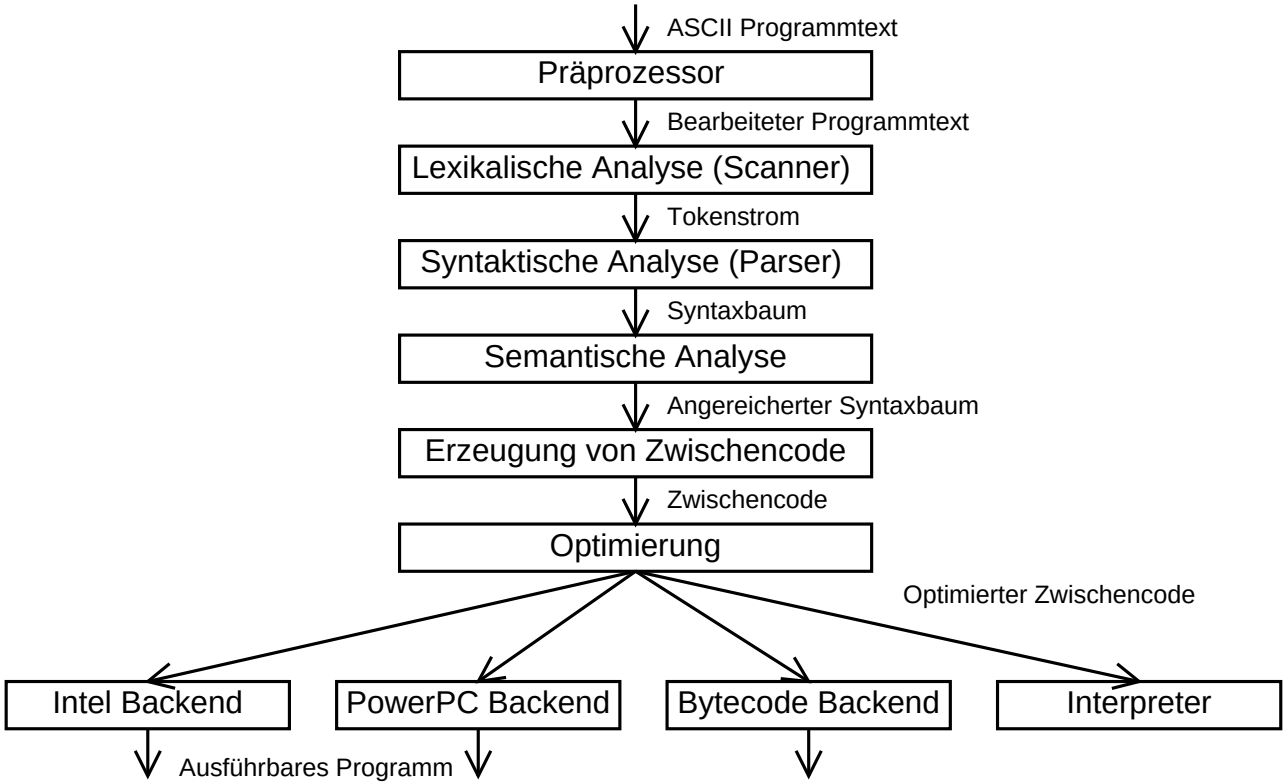
- Jede Verarbeitungsstufe wird durch eine *Filter*-Komponente realisiert
- *Pipe*s (Kanäle) leiten die Daten von Filter zu Filter







Beispiel: Compiler



Problem

Aufgabe: Ein System bauen, das einen Strom von Eingabedaten verarbeiten oder umwandeln soll. Das System soll oder kann nicht als monolithischer Block gebaut werden.

- Das System soll *erweitert* werden können, indem einzelne Teile ausgetauscht oder neu kombiniert werden – womöglich sogar durch den Anwender.
- Kleine Verarbeitungsstufen können leichter wiederverwendet werden.
- Verarbeitungsstufen, die nicht aufeinander folgen, teilen keine Information (und sind somit entkoppelt).
- Es gibt verschiedene Eingabequellen (z.B. Sensoren)
- Explizites Speichern von Zwischenergebnissen ist fehlerträchtig (insbesondere, wenn es Anwendern überlassen wird).
- Parallele Verarbeitung soll zukünftig möglich sein.

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 *Pipe*s verbunden. Die Folge von Verarbeitungsstufen heißt *Pipeline.*







Teilnehmer

Filter

Filter	*Zusammenarbeit mit*
zuständig für	Pipe
• Holt Eingabedaten. • Wendet eine Funktion auf seine Eingabedaten an. • Liefert Ausgabedaten.	

Teilnehmer (2)

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.

Natürlich sind auch *Kombinationen* dieser Grundprinzipien moglich.







Teilnehmer (3)

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.







Teilnehmer (3)

Pipe Eine Pipe verbindet Filter miteinander; sie verbindet auch die Datenquelle mit dem ersten Filter und den letzten Filter mit der Datensenke.

Pipe	*Zusammenarbeit mit*
zuständig für	Datenquelle, Filter, Datensenke
• Übermittelt Daten. • Puffert Daten. • Synchronisiert aktive Nachbarn.	

Verbindet eine Pipe zwei aktive Komponenten, sichert sie die *Synchronisation* der Filter.

Teilnehmer (4)

Datenquelle, Datensenke Diese Komponenten sind die *Endstücke* der Pipeline und somit die Verbindung zur Aussenwelt.

Datenquelle/-senke	*Zusammenarbeit mit*
zuständig für • Übermittelt Daten an/aus Pipeline.	Pipe

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.







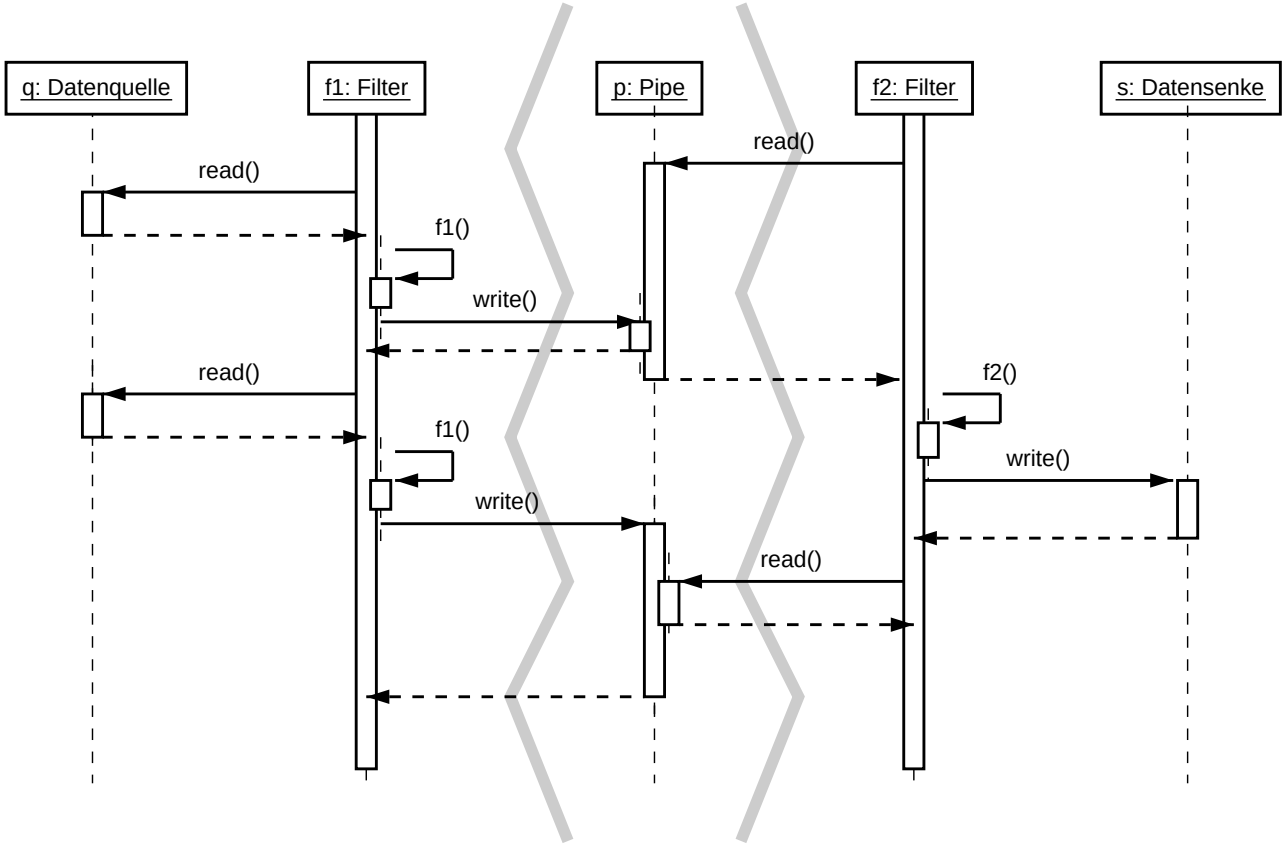
Dynamisches Verhalten

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








q: Datenquelle
f1: Filter
p: Pipe
f2: Filter
s: Datensenke
read()
read()
f1()
write()
read()
f2()
f1()
write()
write()
read()

Folgen des Pipes and Filters-Musters

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







Folgen des Pipes and Filters-Musters (2)

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: UNIX (Pionier)







Vermitteln von Ressourcen: Broker

Beispiel: Rechenzeit verteilen

Moderne Arbeitsplatzrechner verbringen einen Großteil ihrer Rechenzeit damit, auf Eingaben des Benutzers zu warten.

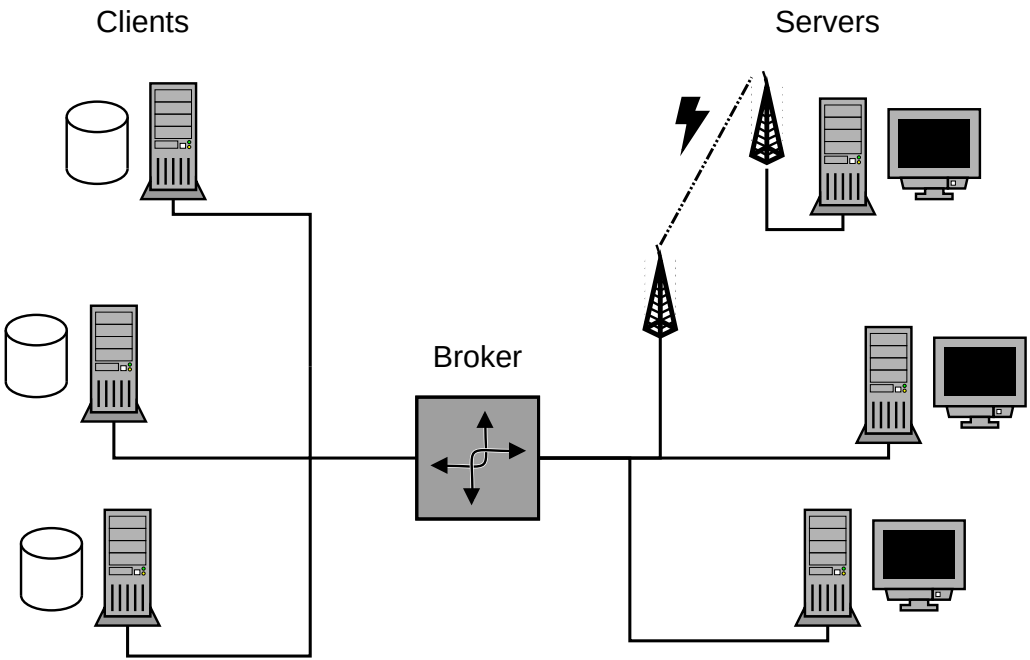
Sie möchten ein System bauen, in dem diese Rechenzeit anderen Anwendern zur Verfügung gestellt werden kann – etwa, um gemeinsam rechenintensive Probleme zu lösen.







Vermitteln von Ressourcen: Broker (2)



Vermitteln von Ressourcen: Broker (3)

Ihr System soll

- zwischen Anbietern (*Servern*) und Anwendern (*Clients*) von Rechenzeit *vermitteln*
- *transparent* arbeiten: Anwender müssen nicht wissen, wo ihre Berechnungen ausgeführt werden
- *dynamisch* arbeiten: Zur Laufzeit können jederzeit Anbieter und Anwender hinzukommen oder wegfallen.







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 Dienste-Anbietern (*Server*) und Dienste-Anwendern (*clients*). Dienste-Anbieter melden sich beim Broker an und machen ihre Dienste für Anwender verfügbar. Anwender senden Dienste-Anforderungen an den Broker, der sie an einen geeigneten Anbieter weiterleitet.

Zu den Aufgaben des Brokers gehört es,

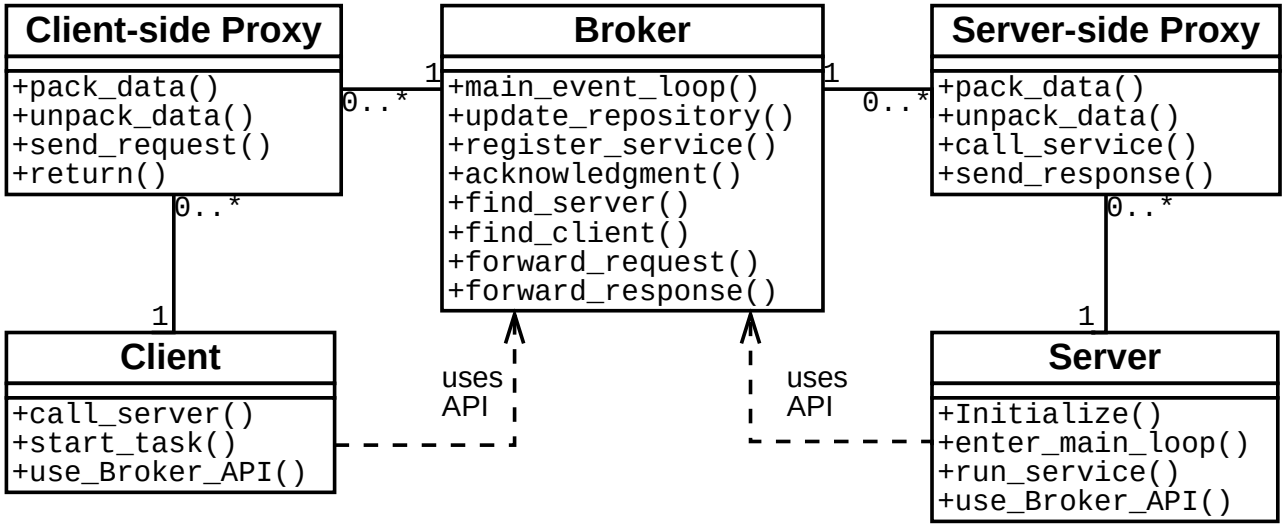
- den passenden Anbieter zu finden
- Anfragen des Anwenders an den passenden Anbieter weiterzuleiten
- die Antwort des Anbieters an den Anwender zurückzusenden







Struktur


Client-side Proxy
+pack_data()
+unpack_data()
+send_request()
+return()
Broker
+main_event_loop()
+update_repository()
+register_service()
+acknowledgment()
+find_server()
+find_client()
+forward_request()
+forward_response()
Server-side Proxy
+pack_data()
+unpack_data()
+call_service()
+send_response()
1
0..*
1
0..*
0..*
1
0..*
1
Client
+call_server()
+start_task()
+use_Broker_API()
uses
API
uses
API
Server
+Initialize()
+enter_main_loop()
+run_service()
+use_Broker_API()

Teilnehmer

Server Der Server bietet Dienste an.

Server	*Zusammenarbeit mit*
zuständig für	Server-side Proxy, Broker
• Implementiert Dienste • Meldet sich beim lokalen Broker an • Kommuniziert mit Client durch Server-side Proxy	

Teilnehmer (2)

Client Der Client macht die Dienste dem Benutzer zugänglich.

Client	*Zusammenarbeit mit*
zuständig für	Client-side Proxy, Broker
• Realisiert Funktionalität für Benutzer • Sendet Anfragen an Server mittels Client-side Proxy	

Teilnehmer (3)

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).

Client-side Proxy	*Zusammenarbeit mit*
zuständig für • Kapselt system-spezifische Funktionalität ein • Vermittelt zwischen Client und Broker	Client, Broker

Teilnehmer (4)

Server-side Proxy

zuständig für

- Fordert Dienste des Servers an
- Kapselt system-spezifische Funktionalität ein
- Vermittelt zwischen Server und Broker

Zusammenarbeit mit

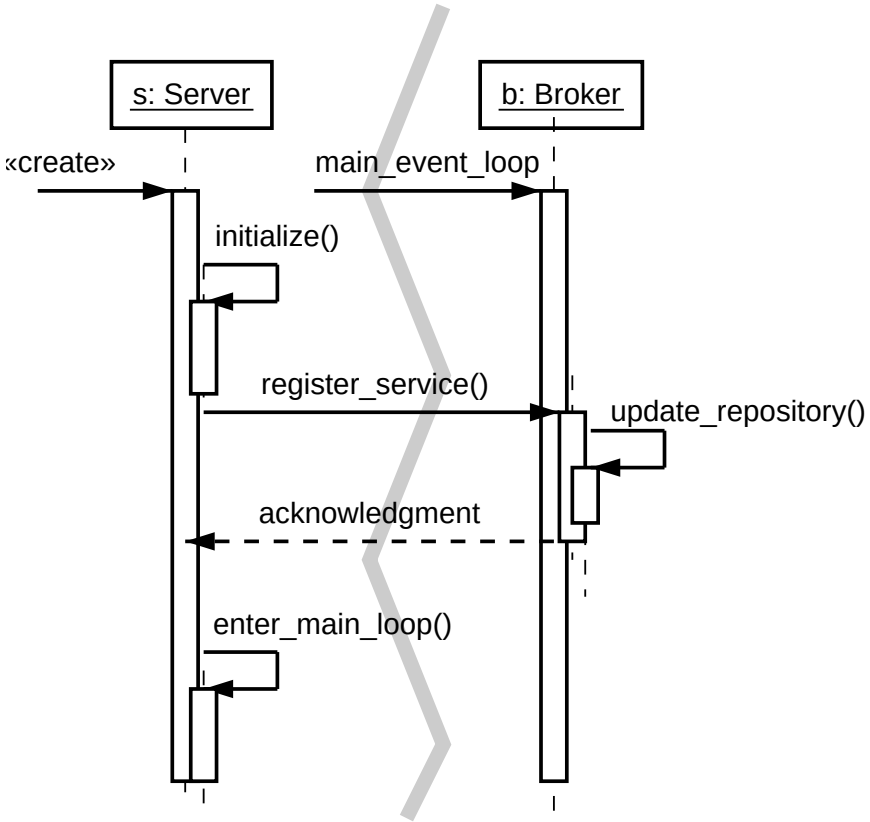
Server, Broker



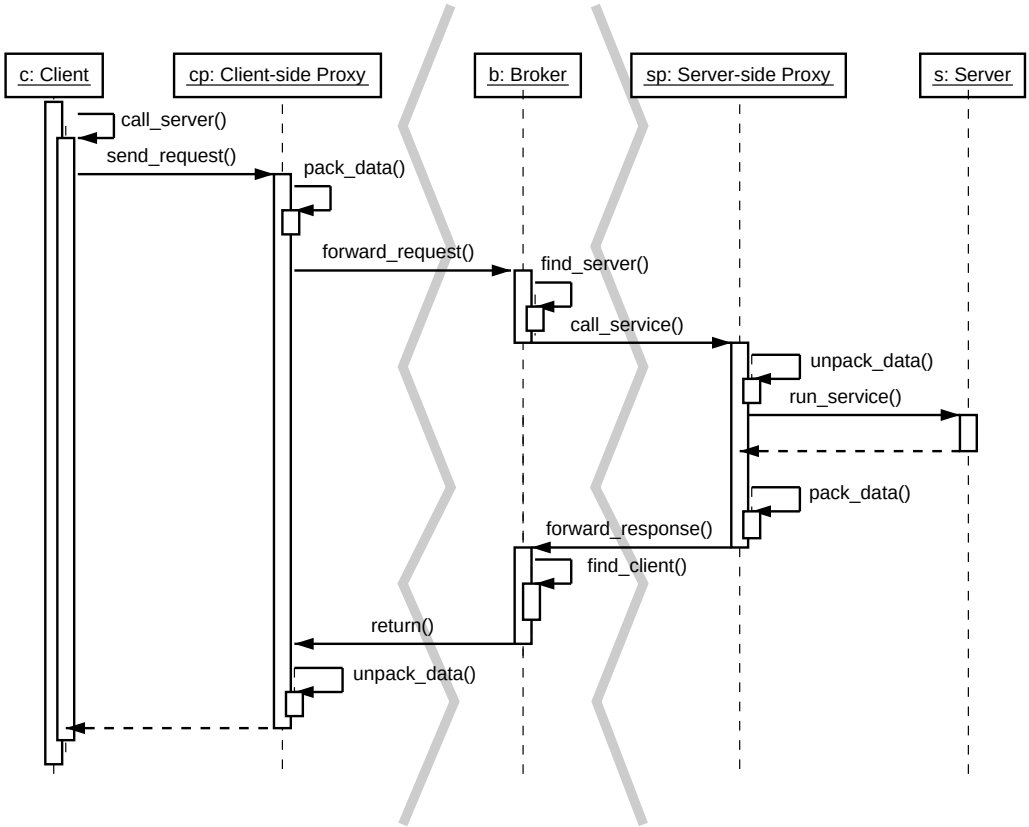




Anmelden eines Servers



Anfrage über Broker



Folgen des Broker-Musters

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)







Einsatzgebiete

Bekannte Einsatzgebiete:

- *Common Object Request Broker Architecture* (CORBA)
- Microsoft .NET, Suns J2EE
- . . . und natürlich das WWW, das größte Broker-System der Welt.







Anti-Muster

Treten die folgenden Muster in der Software-Entwicklung auf, ist Alarm angesagt.







Anti-Muster: Programmierung

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.

Cut-and-Paste Programming. Code wird an zahlreichen Stellen wiederverwendet, indem er kopiert und verändert wird. Dies sorgt für ein Wartungsproblem.

Lösung: Black-Box-Wiederverwendung, Ausfaktorisieren von Gemeinsamkeiten.

Anti-Muster: Programmierung (2)

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.

Anti-Muster: Architektur

Vendor Lock-In. Ein System ist weitgehend abhängig von einer proprietären Architektur oder proprietären Datenformaten.

Lösung: Portabilität erhöhen, Abstraktionen einführen.

Design by Committee. Das typische Anti-Muster von Standardisierungsgremien, die dazu neigen, es jedem Teilnehmer recht zu machen und übermäßig komplexe und ambivalente Entwürfe abzuliefern („Ein Kamel ist ein Pferd, das von einem Komitee entworfen wurde“). Bekannte Beispiele: SQL und CORBA.

Lösung: Gruppendynamik und Treffen verbessern (→ Teamarbeit).

Anti-Muster: Architektur (2)

Reinvent the Wheel. Da es an Wissen über vorhandene Produkte und Lösungen fehlt (auch innerhalb einer Firma), wird das Rad stets neu erfunden – erhöhte Entwicklungskosten und Terminprobleme.

Lösung: Wissensmanagement verbessern.







Weitere Anti-Muster

- *Lava Flow* (schnell wechselnder Entwurf),
- *Boat Anchor* (Komponente ohne erkennbaren Nutzen),
- *Dead End* (eingekaufte Komponente, die nicht mehr unterstützt wird),
- *Swiss Army Knife* (Komponente, die vorgibt, alles tun zu können). . .