Softwaretechnik und Programmierparadigmen

VL09: Implementierung

Prof. Dr. Sabine Glesner
FG Programmierung eingebetteter Systeme
Technische Universität Berlin

Implementierung

- Funktionale und nicht-junktionale Anforderungen sind nun bekannt
- Entwurf mit abstrakten Modellen erstellt
- Wie werden die bisher ermittelten Anforderungen nun in ausführbare Programme überführt?

➢ Was für Anforderungen haben wir an die Implementierung?

➢ Welche Struktur soll das System haben?

➢ Wie kann ich die Anforderungen möglichst effizient umsetzen?

Übersicht

- Anforderungen an die Implementierung
- Architekturstile
- Design Patterns

Übersicht

- Anforderungen an die Implementierung
- Architekturstile
- Design Patterns

Anforderungen

typisches Vorgehen bei Projektbeginn:
Aufstellen einer Anforderungsliste

- Modularität -> Übersichtlichkeit
- Wiederverwendbarkeit
- Übersichtlichkeit
- Erweiterbarkeit
- Effizienz
- Separation of Concerns
- Wartbarkeit

weitere Anforderungen möglich, je nach Situation, teilweise Überschneidungen, teilweise nicht alles gleichzeitig erfüllbar

Während der Implementierung

- Versionsverwaltung — Zusammenarbeit im Team, Sicherheit
- Systemintegration/Deployment
- Problem-/Aufgabenverfolgung — auch Aufgabenverteilung im Team
- Dokumentation

Grundsätzliche Entscheidungen

- Wahl der Plattform
- Wahl der Programmiersprache(n)
- Wahl des GUI-Systems/Frontend
- Wahl der Persistierung/Datenhaltung

abhängig von bereits vorhandener Vorarbeit

Übersicht

- Anforderungen an die Implementierung
- Architekturstile
- Design Patterns

Architekturstile

- Abstrakte Beschreibung („Muster“) eines prinzipiellen Systemaufbaus
- Weitere Bezeichnungen: Architekturmuster, engl. architectural pattern
- Im Folgenden: Beispiele für anerkannte Architekturstile (es gibt noch mehr!)

Architekturstile - Überblick

Abgrenzung: Monolithische Systeme <-> System das Architekturstil hat

= System, das nicht weiter aufteilbar ist, kein erkennbarer Stil, nicht systematisch strukturiert (sinnvoll nur für sehr kleine Programme, zu Testzwecken)

Interaktive Systeme

- Model-View-Controller

Allgemeine Strukturierung (Chaos zu System)

- Layer-based architecture
- Repository-based architecture
- Pipes-and-Filter
- Event-based architecture

Verteilte Systeme

- Client-Server
- P2P
- SOA

Monolithische Systeme

- Keine systematische Strukturierung des Systems
- Abwesenheit von erprobtem Architekturstil

Vorteile:

- Winzige Anwendungen schneller entwickelt

Nachteile:

- Keine Modularität

➢keine Wiederverwendbarkeit von Teilkomponenten

➢schwer Erweiterbar

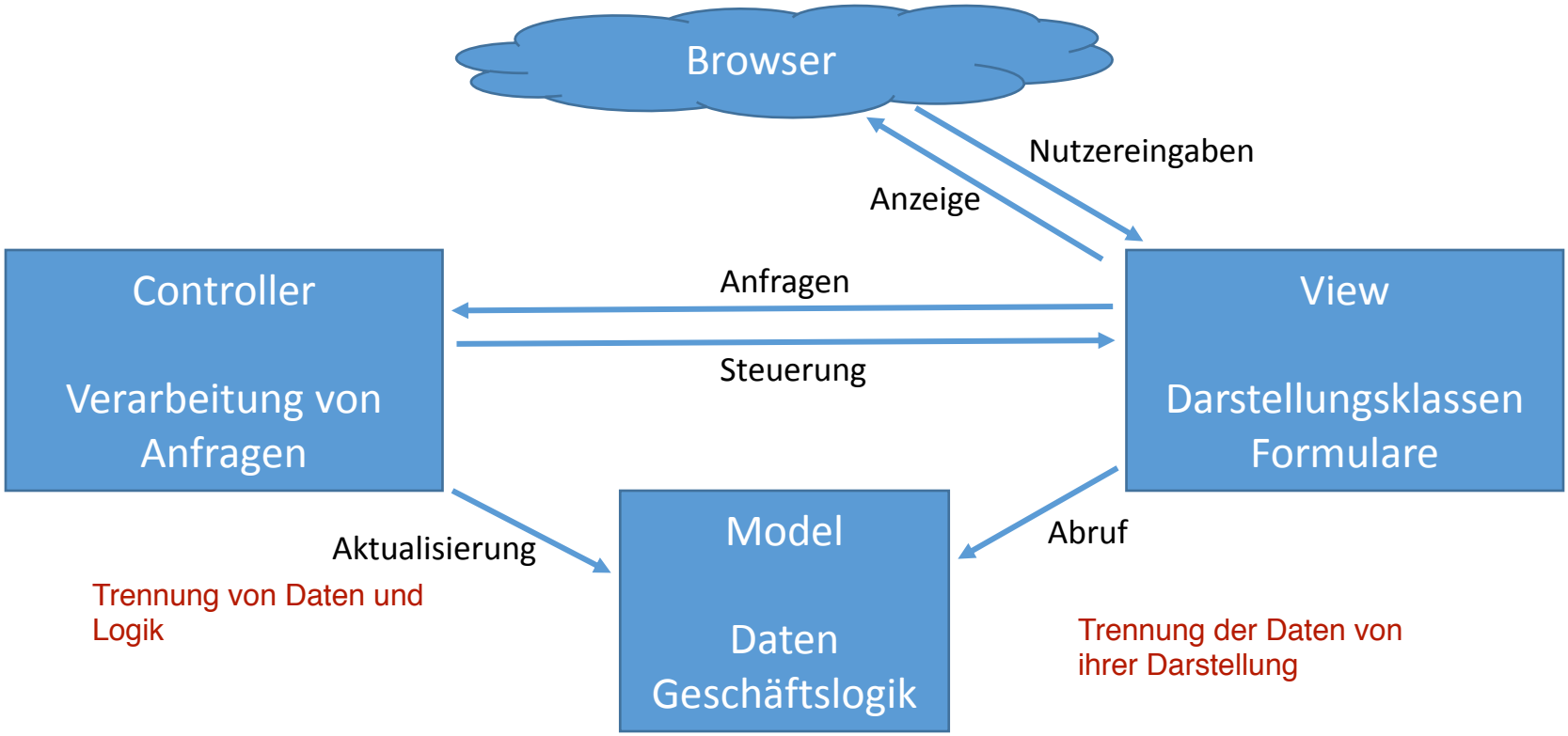
➢unübersichtlich

Model View Controller

- Trennung des Systems in drei Einheiten:
- Model
  - Enthält die darzustellenden Daten
  - Geschäftslogik innerhalb dieser Daten
  - Unabhängig von anderen Einheiten
- View (Präsentation)
  - Darstellung der Daten
  - Entgegennahme von Benutzerinteraktion
  - Kennt Modell und Control
  - Es kann mehrere Präsentationen für die Daten geben
- Controller (Steuerung)
  - Verwaltet die Präsentation
  - Wertet Benutzeranfragen aus und gibt sie ggf. an das Modell weiter

Model View Controller

- Beispiel einer Web-Anwendung



Model View Controller

- Sinnvoll, wenn es verschiedene Möglichkeiten der Präsentation und Interaktion mit dem System gibt
- Typisch in Systemen mit Fokus auf Benutzerschnittstellen (z.B. Web-basierte Anwendungen)
- Erleichtert spätere Erweiterungen, z.B. ganz neue Präsentationsarten (View)

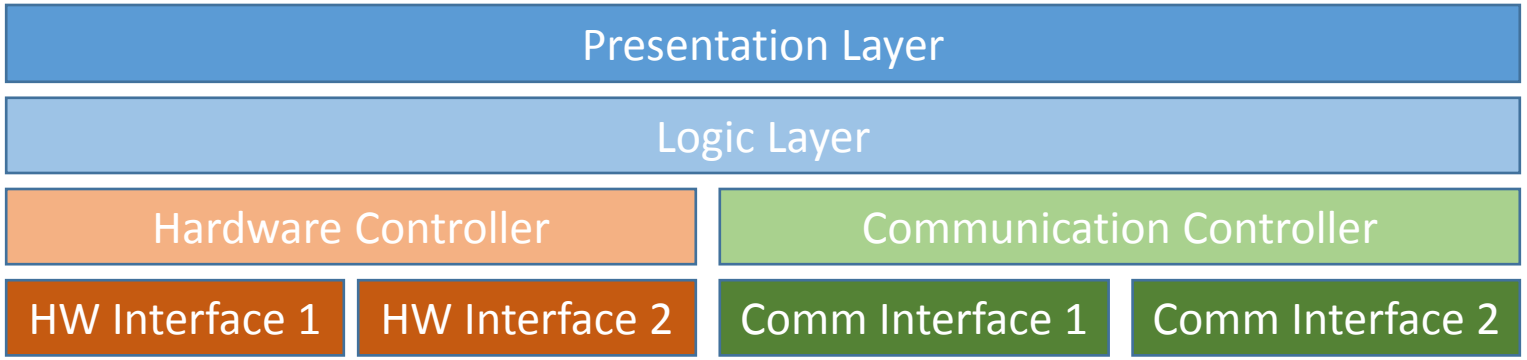
-> Erweiterbarkeit, Modularität, Separation auf Concerns
-> nicht unbedingt Effizienz (nicht im Vordergrund)

Layer-based Architecture

- Aufteilung in mehrere Abstraktionsschichten
- Trennung z.B. von technischen Details und Inhalten
- Schichten bieten darüber liegenden Schichten Dienste an
- Elementare Dienste in unterster Schicht

Layer-based Architecture

- Allgemeines Beispiel:



Layer-based Architecture

- Beispiel: TCP/IP-Referenzmodell (Wikipedia)
- Verschiedene Schichten und Protokolle der Internet-Kommunikation

OSI-Schicht	TCP/IP-Schicht	Beispiel
Anwendungen (7)	Anwendungen	HTTP, UDS, FTP, SMTP, POP, Telnet, OPC UA
Darstellung (6)		
Sitzung (5)		
		SOCKS
Transport (4)	Transport	TCP, UDP, SCTP
Vermittlung (3)	Internet	IP (IPv4, IPv6), ICMP (über IP)
Sicherung (2)	Netzzugang	Ethernet, Token Bus, Token Ring, FDDI, IPoAC
Bitübertragung (1)		

4-Schichten-Modell

Layer-based Architecture

- Strikte Schichtenarchitektur: Schichten dürfen nicht übersprungen werden
- Vorteile:
  - Abstraktion von Details der einzelnen Schichten
  - Modularität: Einfacher Austausch von Schichten möglich
- Nachteile:
  - Trennung kann Performance-Nachteile bringen
  - Anfragen/Antworten müssen teilw. über mehrere Schichten hinweg weitergeleitet werden

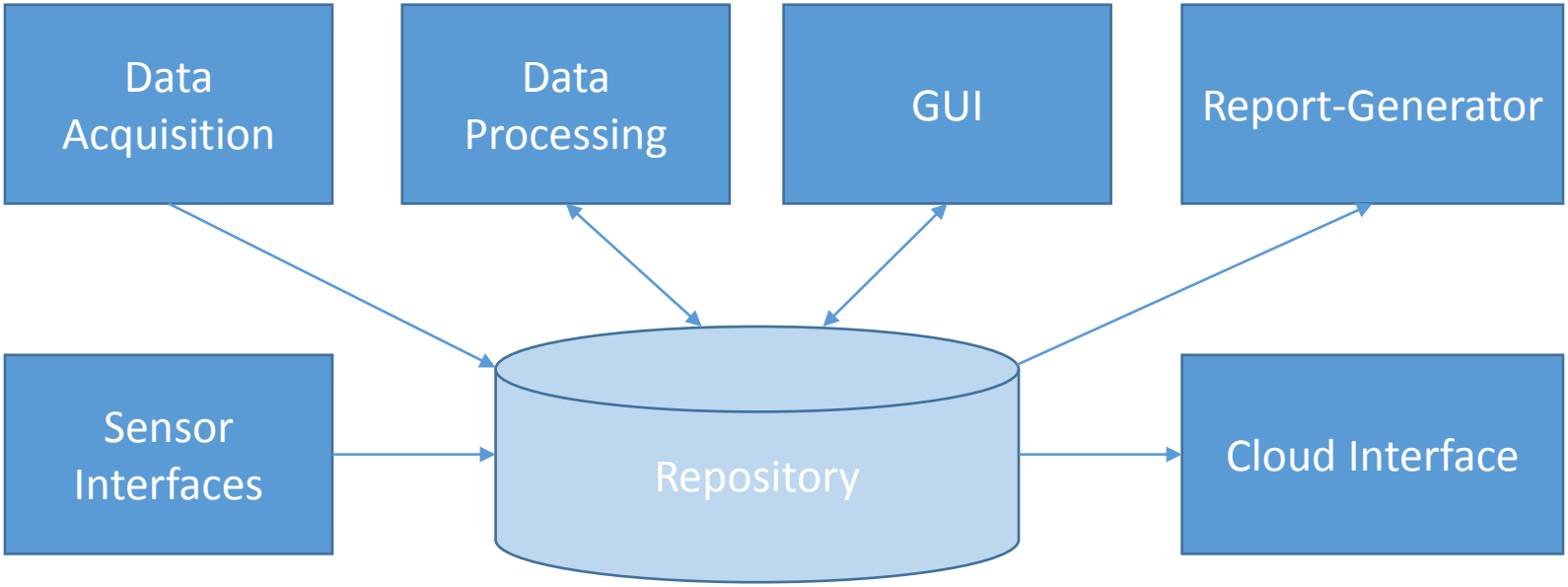
Repository-based Architecture

- Daten des Systems in einem zentralen Datenspeicher (z.B. Datenbank)
- Funktionseinheiten sind nur über die gemeinsamen Daten verbunden
- Koordination von schreibenden/lesenden Komponenten im Repository (z.B: Locks)
- Sinnvoll für Systeme mit großen Datenmengen, die lange gespeichert werden sollen

Systeme können sowohl Repository-based Architecture als auch MVC verwenden!

Repository-based Architecture

- Beispiel: Verwaltung von Wetterdaten



Repository-based Architecture

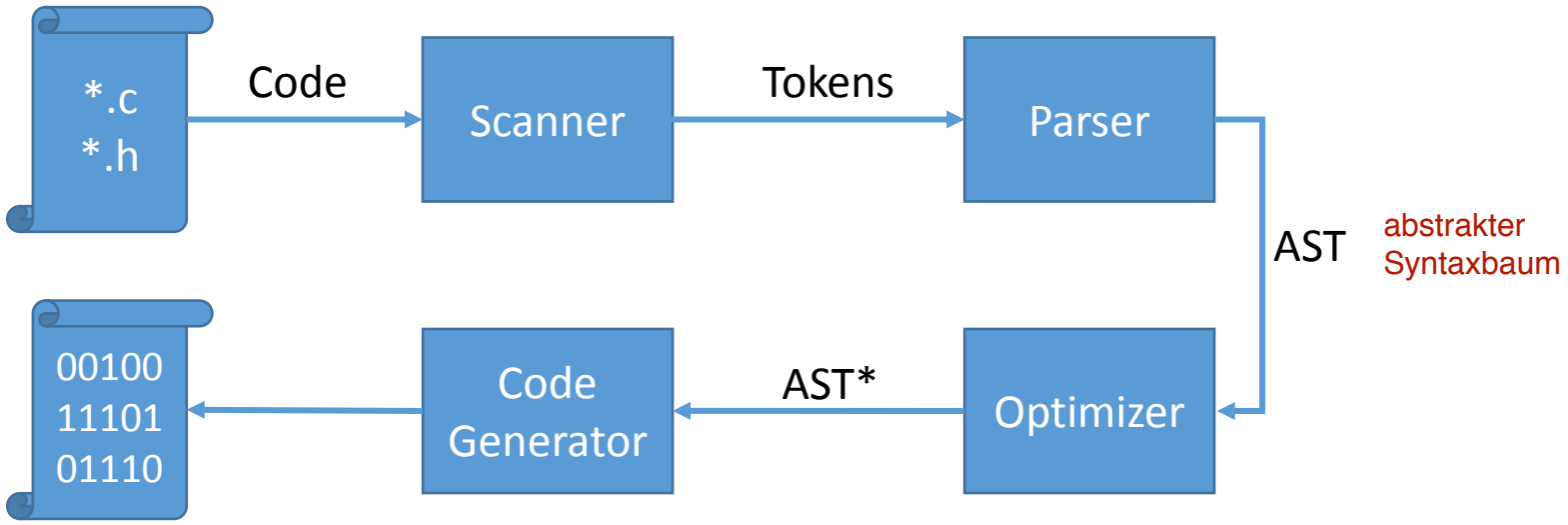
- Vorteile:
  - Unabhängige Komponenten
  - Wenig Schnittstellen, modular
  - Konsistente, zentrale Datenhaltung
- Nachteile:
  - Kommunikation zwischen Komponenten über die Daten teilweise ineffizient
  - Repository ist „single point of failure“

Pipes-and-Filter Architecture

- Arbeitsschritte nur durch Daten verknüpft (fließen wie durch ein Rohr von einer Komponente zur nächsten)
- Typisch für Systeme in denen Daten schrittweise verändert (weiterverarbeitet) werden
- Bearbeitung nacheinander und parallel möglich

Pipes-and-Filter Architecture

- Beispiel: einfacher Compiler



Pipes-and-Filter Architecture

- Vorteile:
  - System bildet Geschäftsprozesse direkt ab
  - Übersichtlich
  - Modular, leicht erweiterbar
- Nachteile
  - Daten müssen von jeder Komponente erneut aufbereitet werden

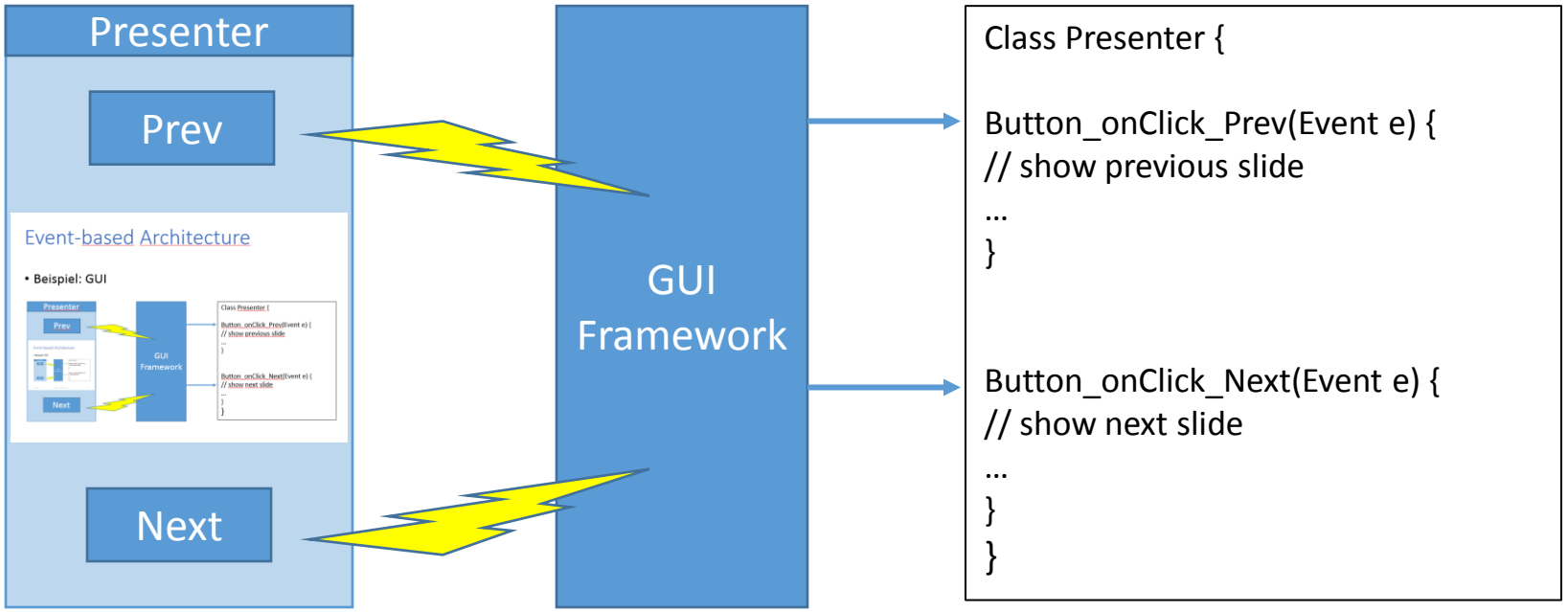
Event-based Architecture

- Komponenten sind unabhängig von einander
- Warten auf Ereignisse (Events) zur Steuerung ihrer Aktivitäten (consumer, sink)
- Oder lösen Ereignisse aus (producer, agent)
- Zentrale Komponente (event channel) verteilt die Ereignisse

oft in eingebetteten Systemen verwendet, da eventbasiert

Event-based Architecture

- Beispiel: GUI-Verwaltung



Event-based Architecture

Vorteile:

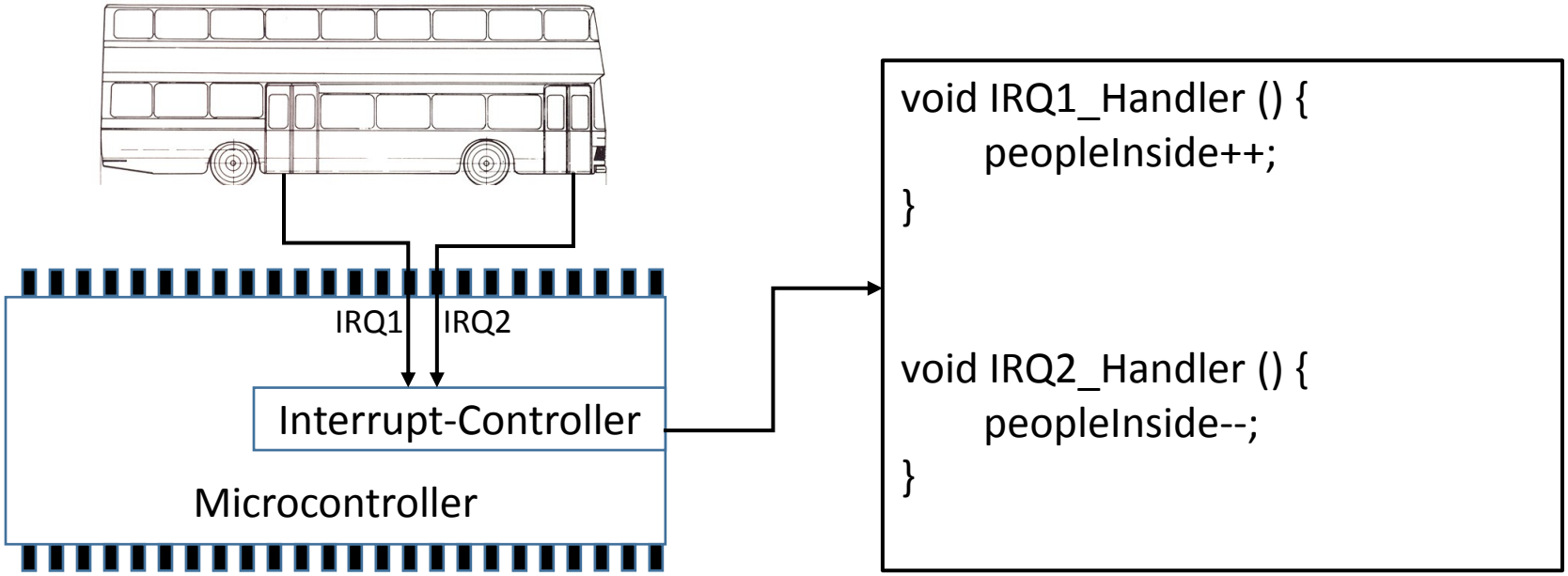
- Reaktion auf Ereignisse kann unmittelbar erfolgen (Kein Polling nötig)
- Geeignet für asynchrone und unvorhersehbare Umgebungen
- Entkopplung von Komponenten

Nachteile:

- Erhöhte Anforderungen an Synchronisation
- Verhalten schwerer vorhersagbar

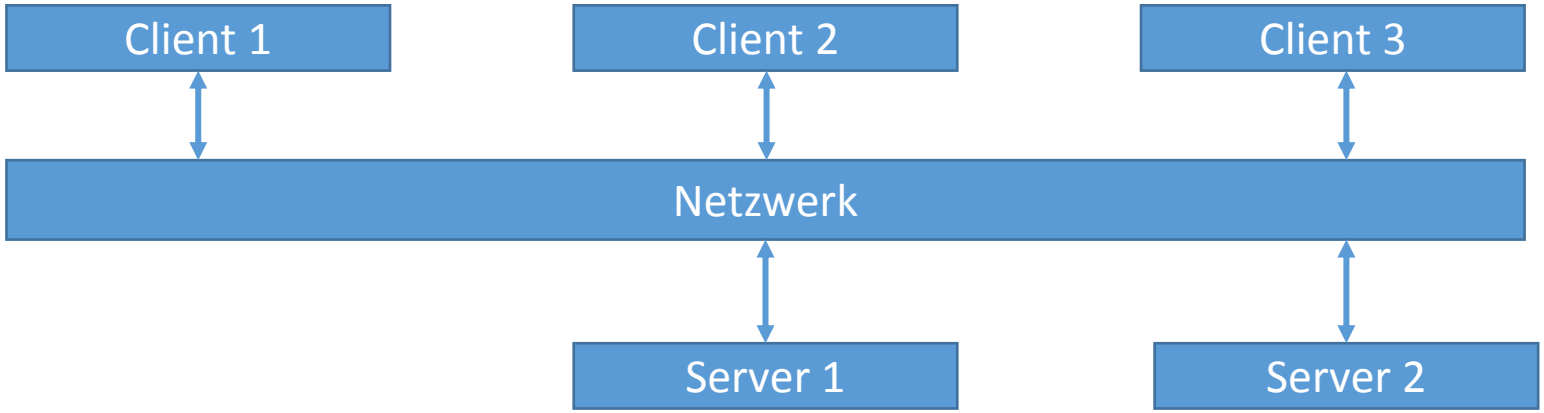
Interrupt-based Architecture

- „Low-level“ Event-based architecture
- Beispiel: Personenzähler im Bus



Client-Server Architecture

- Architekturstil für verteilte Systeme
- Jede Systemfunktion wird als Dienst auf einem zentralen Server angeboten
- Clients können diese Funktion in Anspruch nehmen

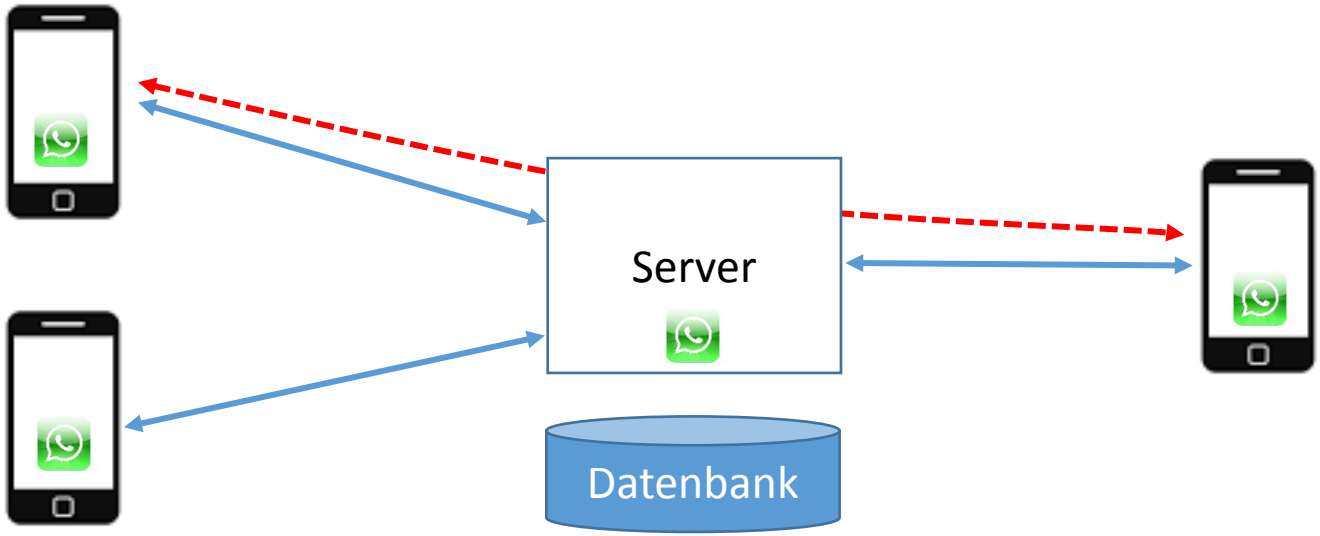


Client-Server Architecture

- Keine direkte Kommunikation zwischen Clients

➢Interaktion zwischen Clients über Server möglich

- Beispiel: Instant Messaging



Client-Server Architecture

Vorteile:

- Funktionen stehen zentral zur Verfügung, müssen nicht mehrfach implementiert werden
- Einfache Verwaltung gemeinsam genutzter Ressourcen

Nachteile:

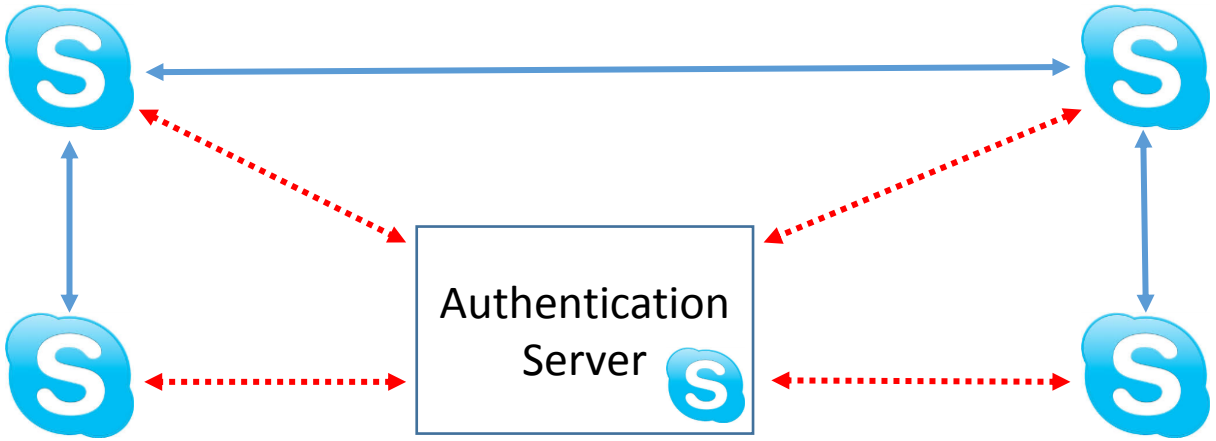
- Single point of failure
- Zusätzliche Kommunikation
- Ungleiche Lastverteilung / schlechte Ressourcennutzung

Peer-to-Peer Architektur

- Teilnehmende Komponenten sind gleichberechtigt
- Jede kann Funktionen bereitstellen und nutzen
- Meist macht ein zentraler Server die Teilnehmer bekannt

Peer-to-Peer Architektur

- Beispiel: Skype (in den Anfängen)
- Jeder Client meldet sich beim Server nur zur Authentifizierung und zum Abrufen des Status der Kontakte



Peer-to-Peer Architektur

Vorteile:

- Effiziente Kommunikation (keine Umwege)
- Gleichmäßige Last/Ressourcenverteilung

Nachteile

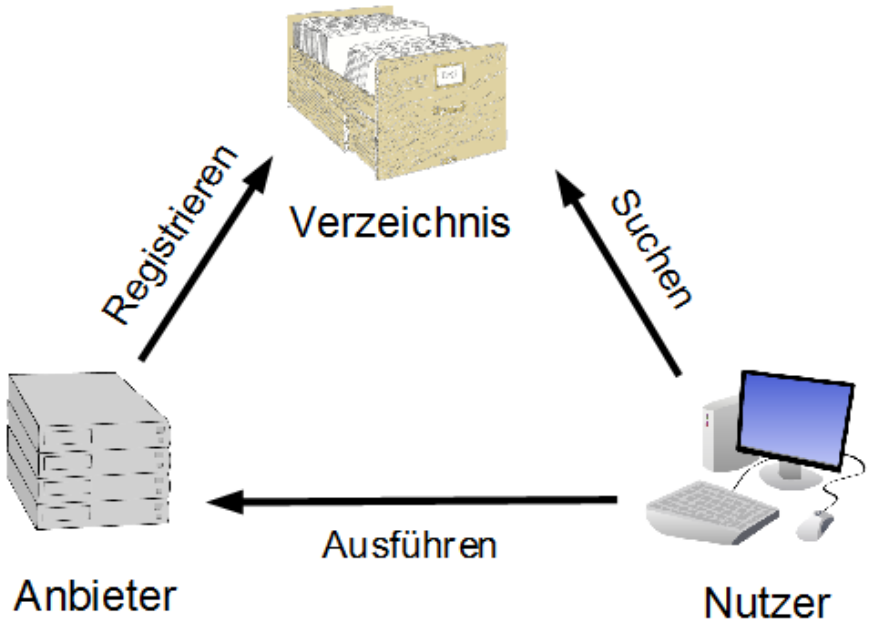
- Gemeinsam genutzte Ressourcen schwierig zu synchronisieren
- Funktionen mehrfach implementiert (Komponenten komplexer)

Service Oriented Architecture

- Kurz: SOA
- System besteht aus verteilten, unabhängigen Komponenten
- Komponenten bieten Funktionalitäten als Services an
- Komponenten agieren als Black-Boxes
- Globale Registry für Service-Anbieter und Services
- Komplexe Komponenten können wieder andere durch Services verwenden
- Gemeinsames Protokoll für alle Services (Plattformunabhängig)
- Erlauben loose coupling (dynamische Verbindung im Betrieb)

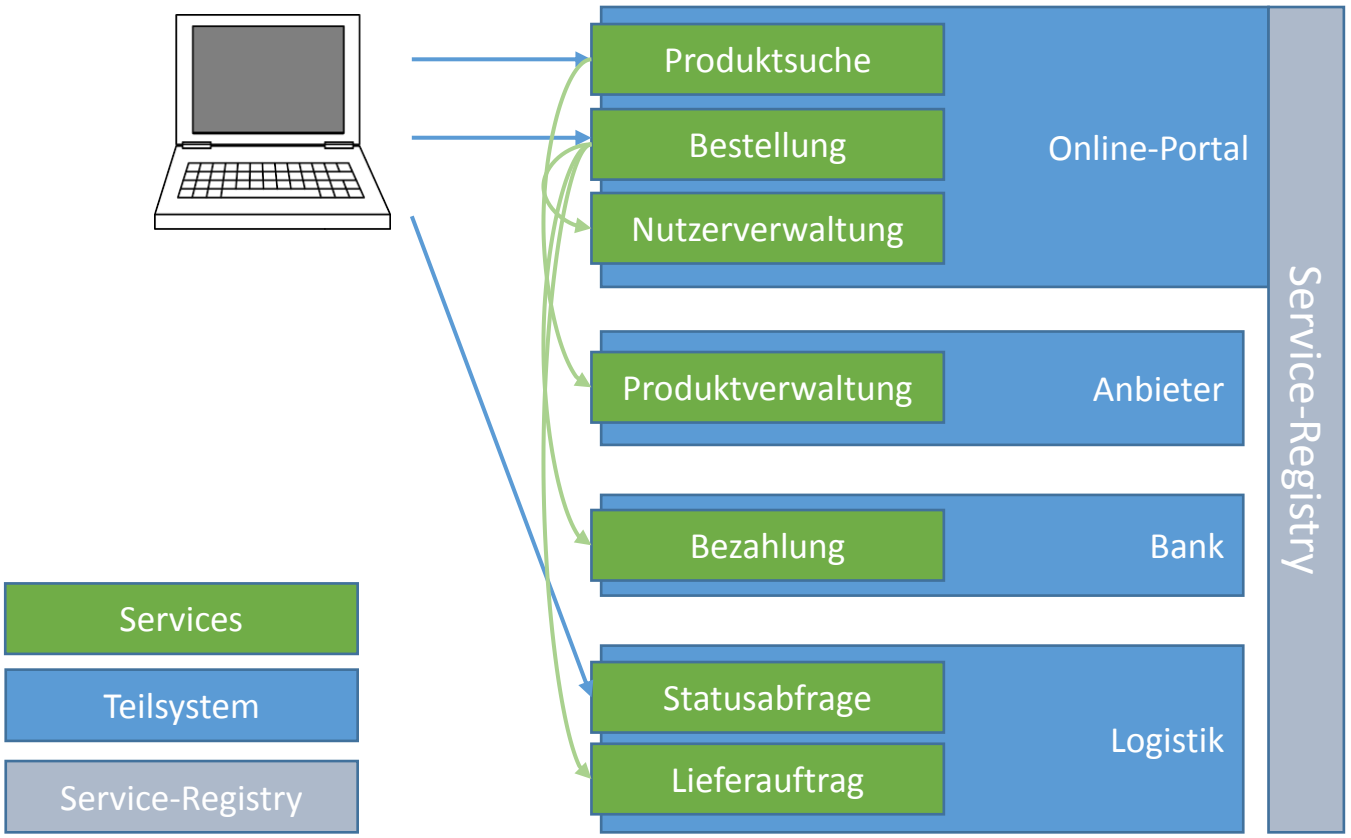
Service Oriented Architecture

- Beispiel für Implementierung: Webservices
- Beispiele für Protokoll: SOAP, REST
- Allgemeine Struktur:



Service Oriented Architecture

- Beispiel: Web-Shop



Service Oriented Architecture

Vorteile:

- Services/Funktionalitäten austauschbar
- Erweiterung durch simple Registrierung neuer Services
- Einheitliche/standardisierte Protokolle für Schnittstellen

Nachteile:

- Komplexe technische Infrastruktur
- Registry ist „single point of failure“

Kombination von Architekturstilen

- Architektur spielt für die Implementierung wesentliche Rolle
  - Grundsätzlich sollte die Software eine klare Struktur haben
  - Verwendung bewährter Architekturstile empfohlen
  - Funktionales Verhalten möglicherweise mit mehr Architekturstilen abbildbar als nicht-funktionale Anforderungen
- In größeren Systemen können mehrere Architekturstile gemischt auftreten
- Einzelne Komponenten verteilter Systeme können wiederum jeweils eigene Architektur haben

Übersicht

- Anforderungen an die Implementierung
- Architekturstile
- Design Patterns

Design Patterns

- Deutsch: Entwurfsmuster
- Generische Lösung für wiederkehrendes Entwurfsproblem
- Erfahrungen mit erfolgreichen Lösungsansätzen übertragbar machen
- Architekturstile sind Design Patterns im größeren Rahmen (ganzes System)
- Überschneidungen möglich (z.B. strittig ob MVC Architekturstil oder Design Pattern)

Design Patterns

- 1995: Populäre Veröffentlichung einer Sammlung:
  - Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software.
  - Bekannt als “Gang of Four/GoF”

Auf OOP-Bereich bezogen!

- Elemente
  - Name des Musters
  - Problembeschreibung
  - Lösungsbeschreibung
  - (Konsequenzen)
- Nachfolgend wurden weitere Patterns von anderen Autoren veröffentlicht

Design Patterns

Nach „Design Patterns“ von der „Gang of Four“

Erzeugungsmuster	Strukturmuster	Verhaltensmuster
factory method	bridge	template method
abstract factory	decorator	observer
singleton	façade	visitor
builder	flyweight	iterator
prototype	composite	command
	proxy	memento
		strategy
		mediator
		state
		chain of responsibility

Erzeugungsmuster – Beispiel 1

Name: Factory Method (Virtual Constructor)

Problem:

- Unser Programm verwendet abstrakte Typen
- Es kann (und will) nicht vorhersagen welche konkreten Typen verwendet werden und wie sie erzeugt werden
- Konkrete Anwendung soll entscheiden welche Typen erzeugt werden und wie das geschehen soll

Lösung:

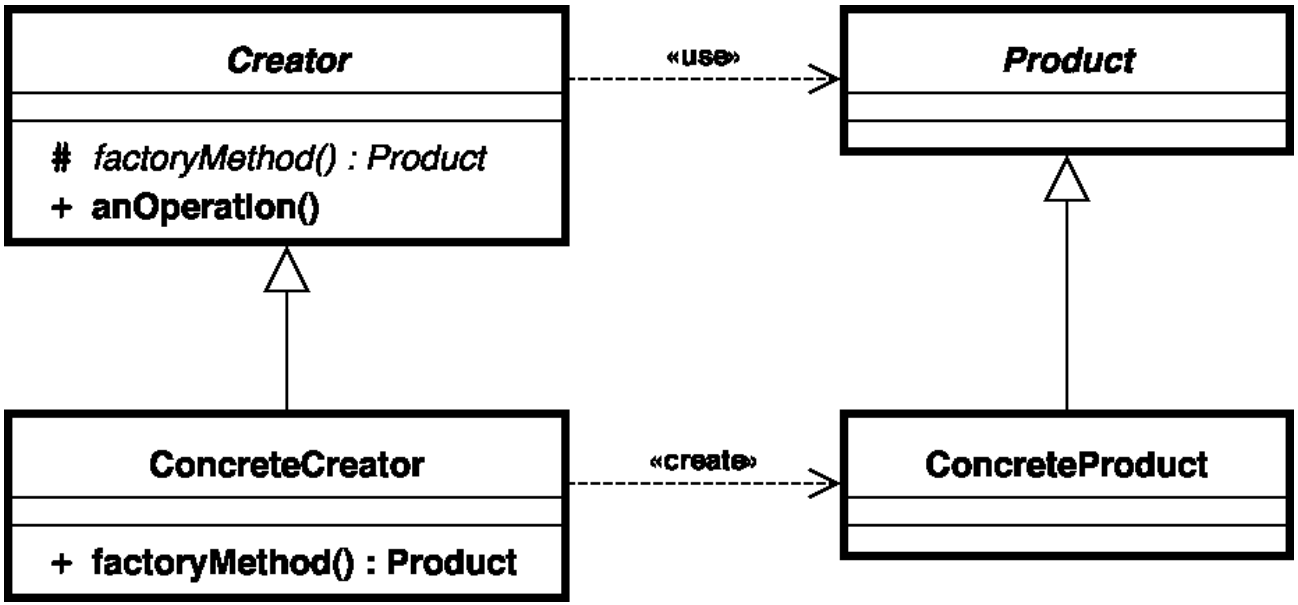
- Für die Erzeugung der Typen wird eine abstrakte Methode definiert
- Subklassen entscheiden welche Klassen instanziiert werden indem sie auch diese abstrakte Methode implementieren

Erzeugungsmuster – Beispiel 1

Anwendung:

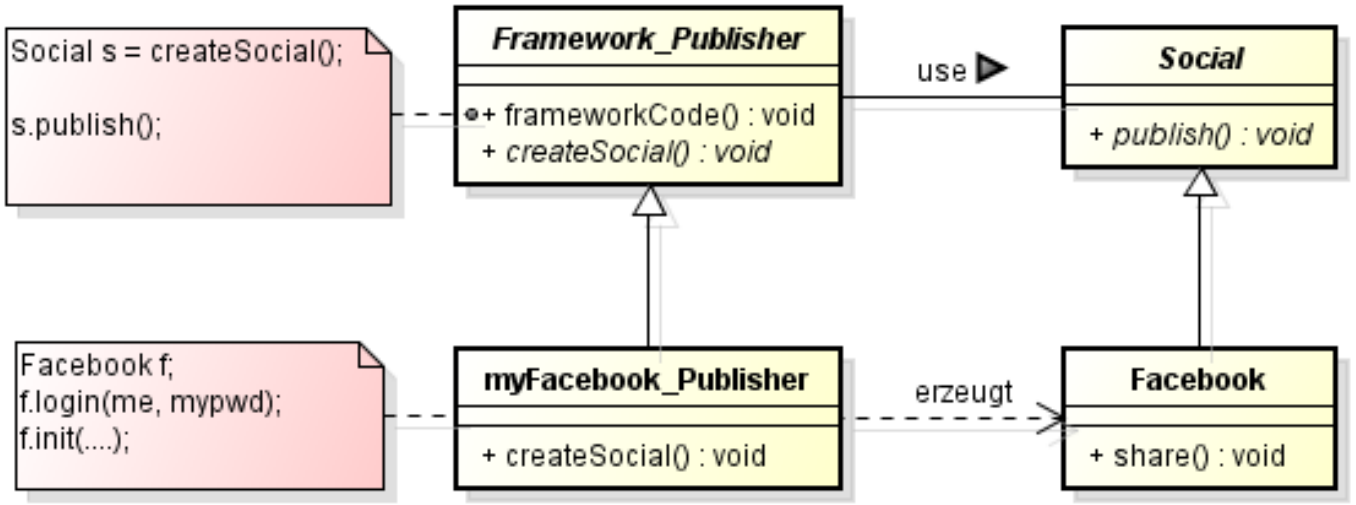
- Frameworks und Klassenbibliotheken

Struktur:



Erzeugungsmuster – Beispiel 1

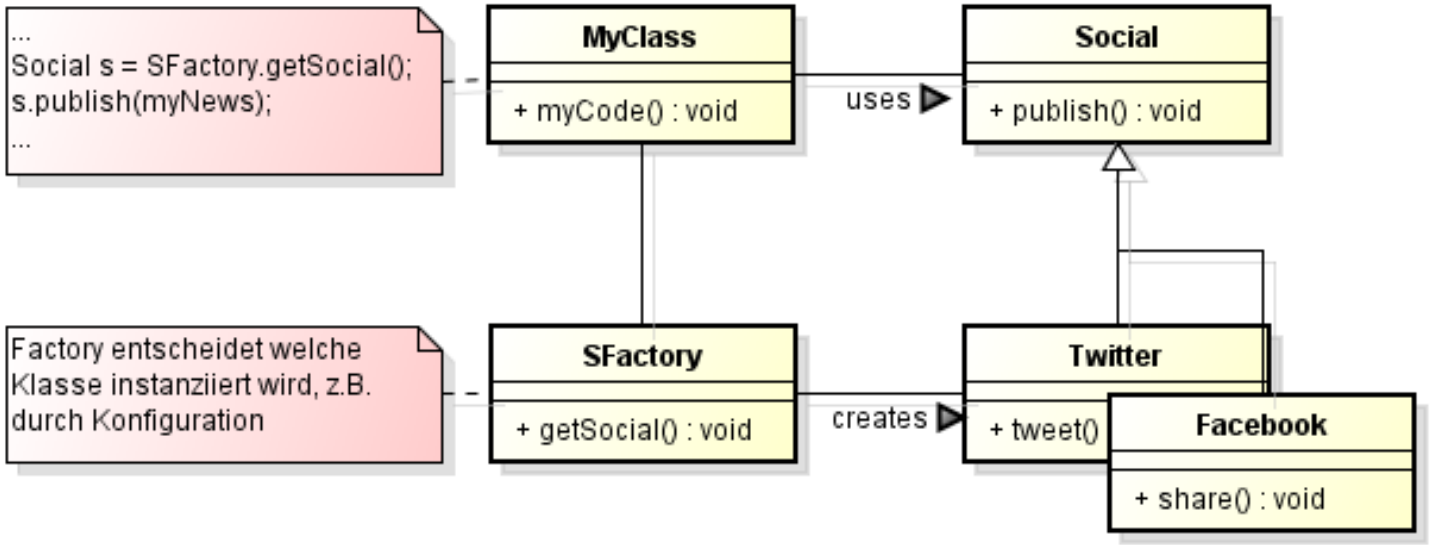
- Beispiel Fabrikmethode
- Zwei Abstrakte Klassen im Framework werden von unserem Programm implementiert



Erzeugungsmuster – Beispiel 1

Abgrenzung: Beispiel für einfache Factory-Klasse

- Hier überlässt man die Erstellung einer konkreten Factory
- Weniger flexibel als Factory Method, wo auch die Funktion zur Erstellung im Framework abstrakt ist



Erzeugungsmuster – Beispiel 2

Name: Singleton

Problem:

- Für manche Klassen ist nur eine Instanz sinnvoll
- Beispiel: Dateisystem
- Sicherstellen, dass nur eine Instanz erzeugt werden kann
- Globale/statische Variable für die Instanz reicht nicht

Lösung:

- Die Klasse muss selbst dafür sorgen, dass sie (oder Subklassen) nur einmal instanziiert wird

Erzeugungsmuster – Beispiel 2

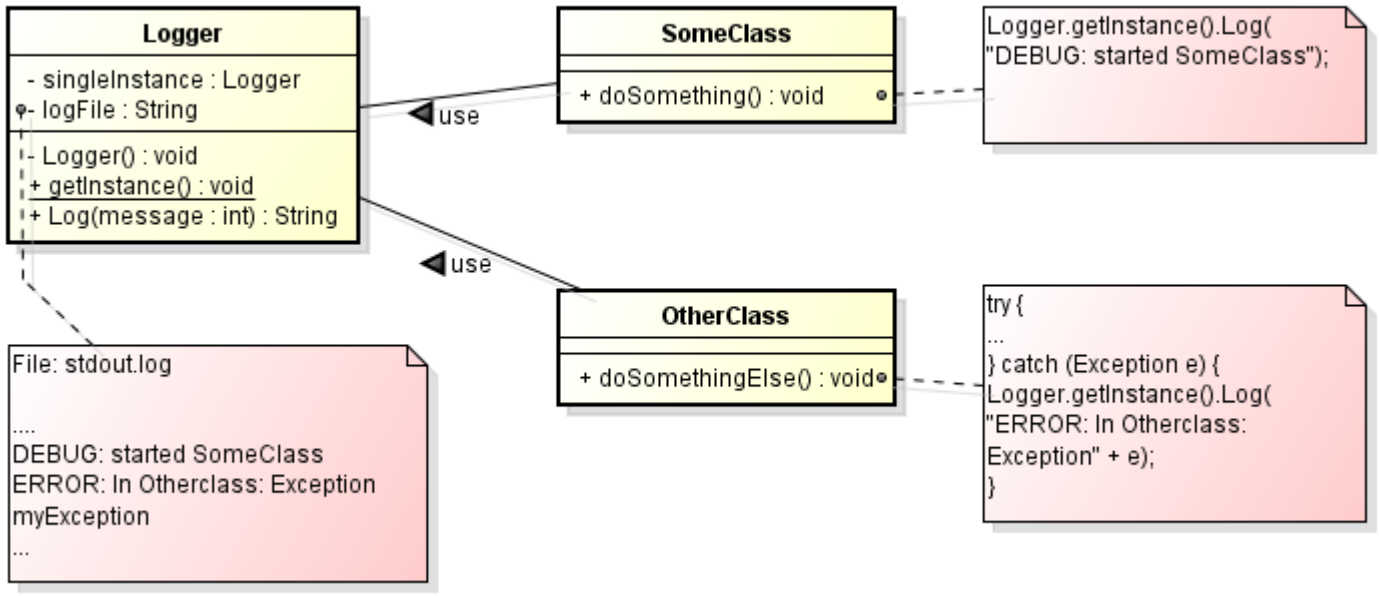
- Konstruktor nicht von außen erreichbar
- Instanziierung in einer Methode versteckt
- Erstellt die Instanz beim ersten Aufruf (lazy initialization)

```
public class mySingleton
{
        private static mySingleton instance = null; // static singleton instance

        private mySingleton () { }  // private constructor

        public static synchronized mySingleton getInstance()
        {
                if (instance == null) {
                        instance = new mySingleton();
                }
                return instance;
        }
}
```

Erzeugungsmuster – Beispiel 2

Beispiel Logging:

- Logger schreibt Applikationsweit in die gleiche Datei -> Singleton



Strukturmuster – Beispiel 1

Operationen auf Listen, Bäumen = Menge von Daten, die gleichartig behandelt werden

Name : Kompositum (composite)

Problem:

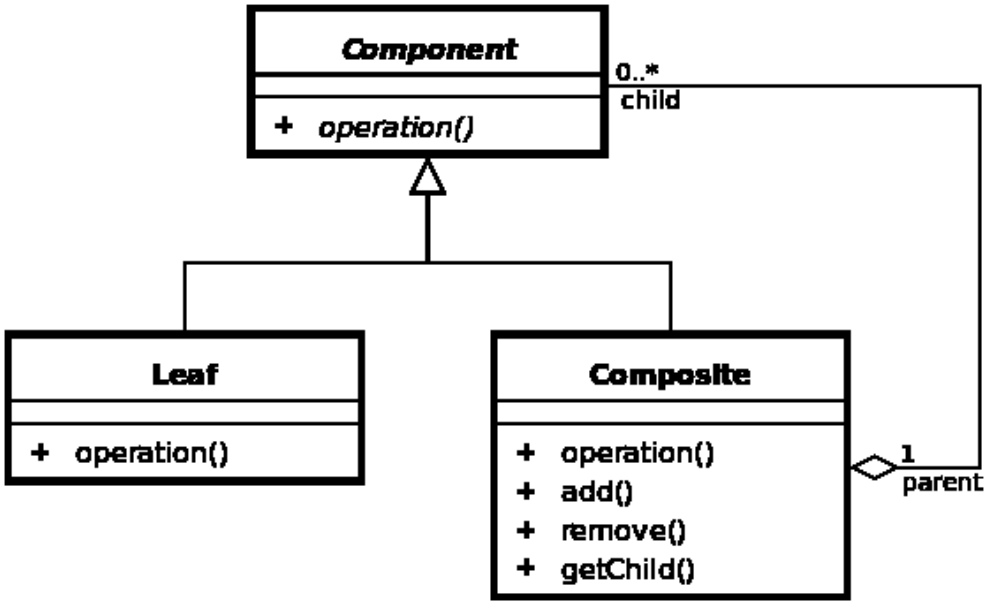
- Daten sind hierarchisch organisiert (baumförmig)
- Programm führt auf allen Knoten gleichartige Operation aus (Baum-Traversion)

Lösung:

- Composite definiert Hierarchien die aus komplexen Objekten (composites) und einfachen Objekten besteht
- Für das Programm transparent, was für ein Objekt behandelt wird (gemeinsame abstrakte Operation für alle Knoten)

Strukturmuster – Beispiel 1

Struktur:



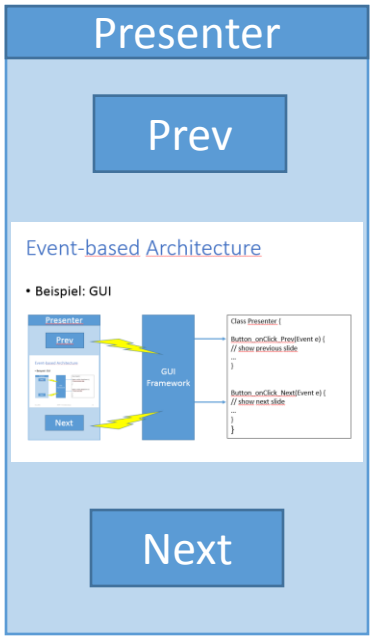
Vorteile:

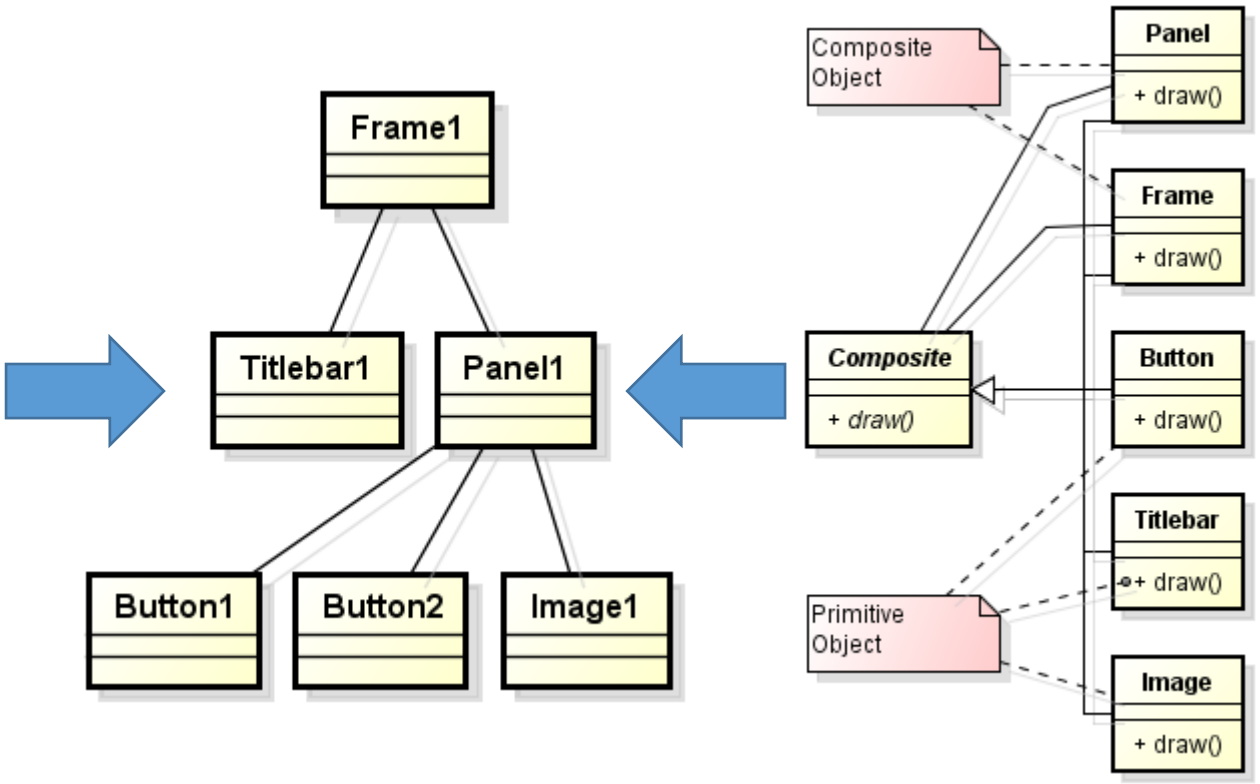
- Vereinfacht den Client-Code
- Neue Komponenten können leicht hinzugefügt werden

Strukturmuster – Beispiel 1

Beispiel: GUI

draw() Funktion ist auf jedem Element realisiert, unter Umständen aber auf jedem Element unterschiedlich





Strukturmuster – Beispiel 2

Name: Fassade (*façade)*

Fassade ist einfache, saubere Schnittstelle nach Außen (fasst Funktionalitäten zusammen und abstrahiert von der Komplexität des Systems)

Problem:

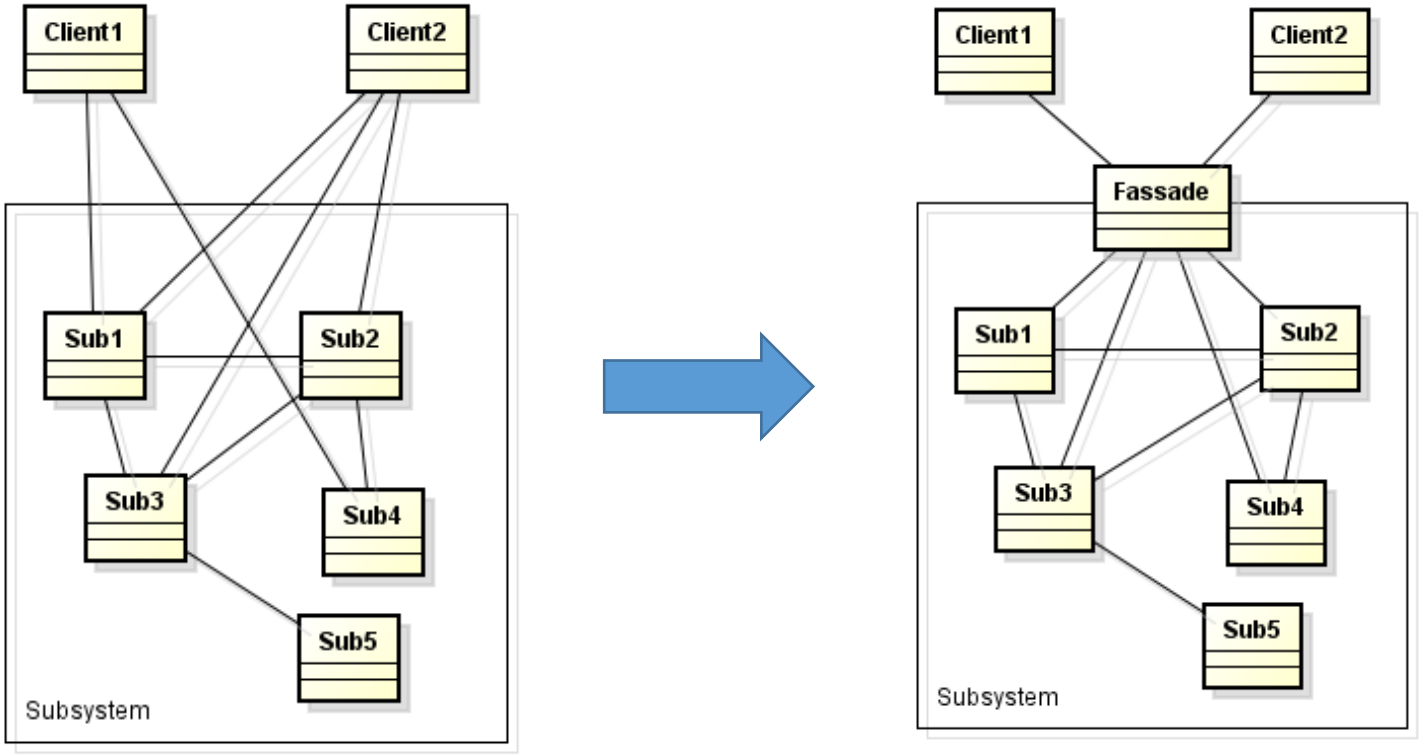
- Clients greifen auf komplexes Subsystem zu
- Verwendete Klassen bieten mehr Funktionen als nötig, Abhängigkeiten/Kommunikation zu komplex

Lösung:

- „Fassade“ bietet vereinfachte Schnittstelle nach außen
- Fasst Kommunikation/Funktionalität an einer Stelle zusammen
- Abstrahiert von Komplexität des Subsystems

Strukturmuster – Beispiel 2

- Die Fassade kann auch als Controller-Klasse für die Clients aufgefasst werden



Strukturmuster – Beispiel 3

Name: Stellvertreter (proxy)

Problem:

- Ein Zugriff/Verbindung zu einem Objekt kann durch einen Pointer nicht ausreichend dargestellt werden
- Zugriffsoperationen sind komplexer oder nur Teilmengen der Zugriffsmöglichkeiten sollen erlaubt sein

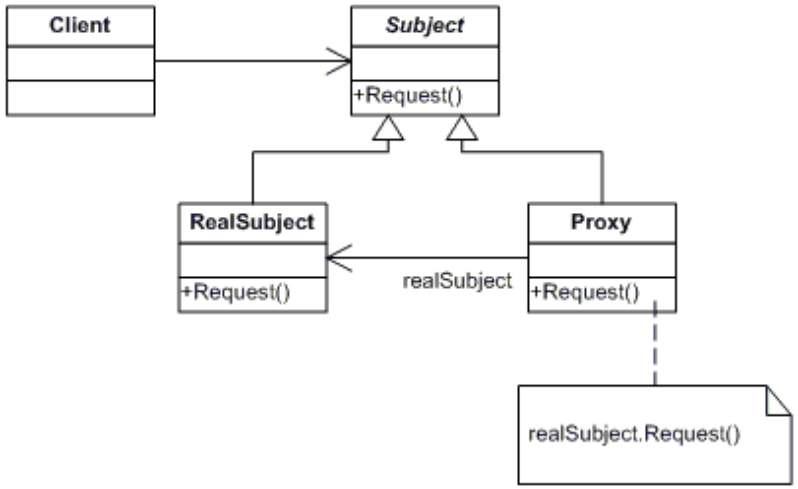
Lösung:

- Proxy-Klasse ersetzt die tatsächliche Klasse an der Stelle der Verwendung
- Kapselt Zugriffe und implementiert zusätzliche Funktionalität

Dynamic Dispatch: am besten passende Methode wird zur Laufzeit ausgesucht (aus gleichnamigen Methoden in Klassen und Unterklassen z.B.)

Strukturmuster – Beispiel 3

- Struktur:



- Proxy und „echte“ Klasse erben von abstrakten Typ
- Proxy reicht Abfragen weiter und fügt eigene Funktionalität hinzu

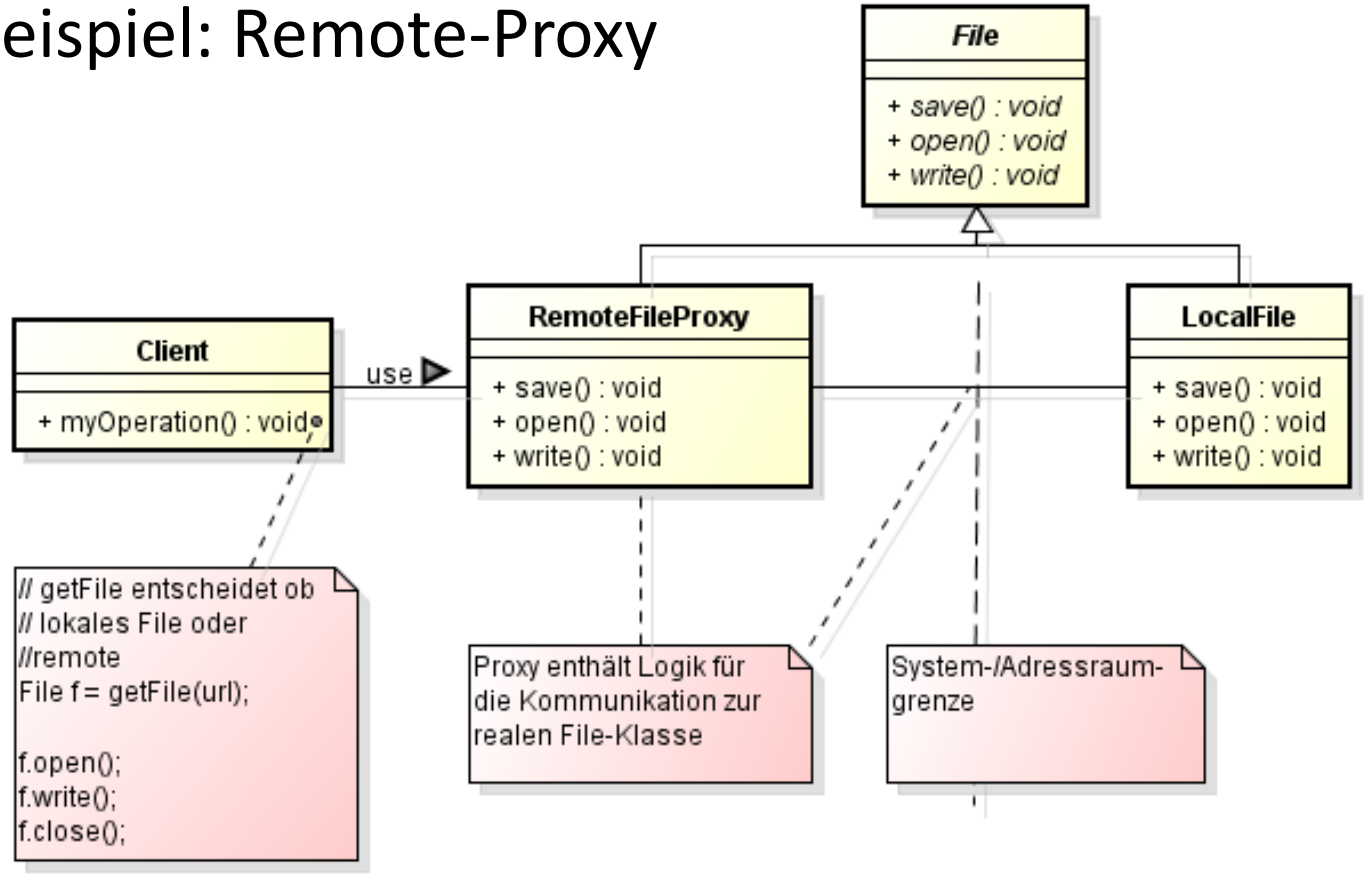
Strukturmuster – Beispiel 3

Anwendung:

- Remote Proxy: Lokale Repräsentation eines Remote-Objekts (andere Bezeichnung: Botschafter)
- Virtual Proxy: Die Erzeugung des Objektes ist aufwendig aber nicht immer notwendig. Proxy erstellt das Objekt erst bei Bedarf
- Protection Proxy: Bietet eingeschränkten Zugriff auf Objekte die größeren Schutz benötigen
- Smart Reference: Führt zusätzliche Aktionen beim Zugriff aus (z.B. Zugriffszähler)

Strukturmuster – Beispiel 3

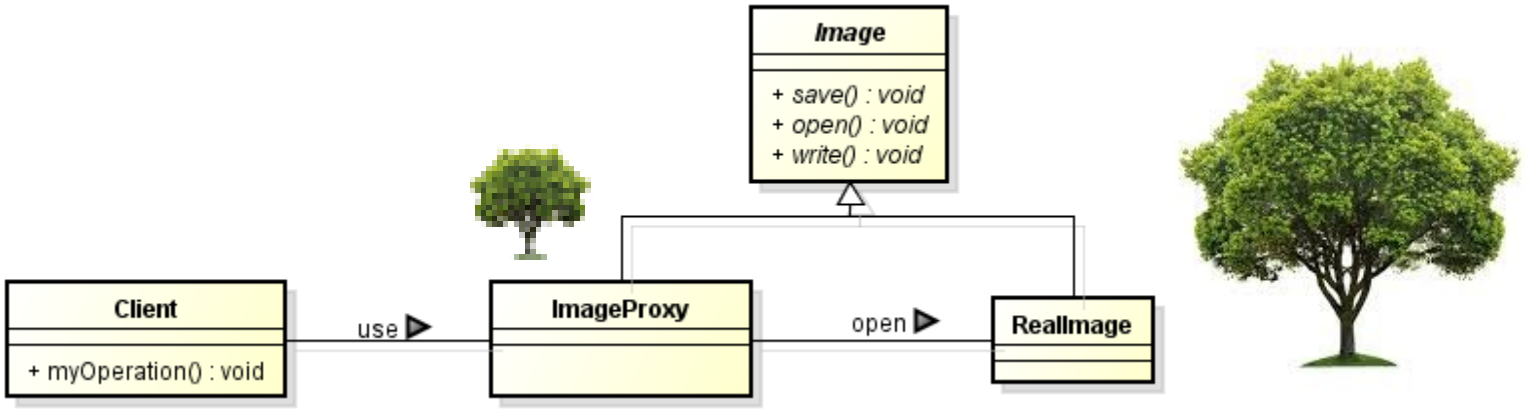
- Beispiel: Remote-Proxy



Strukturmuster – Beispiel 3

Beispiel: Virtual Proxy

- Ganzes Bild laden ist aufwendig
- Proxy stellt Thumbnail zur Verfügung
- Lädt echtes Bild nur wenn nötig



Verhaltensmuster – Beispiel 1

Name: Beobachter (observer)

Problem:

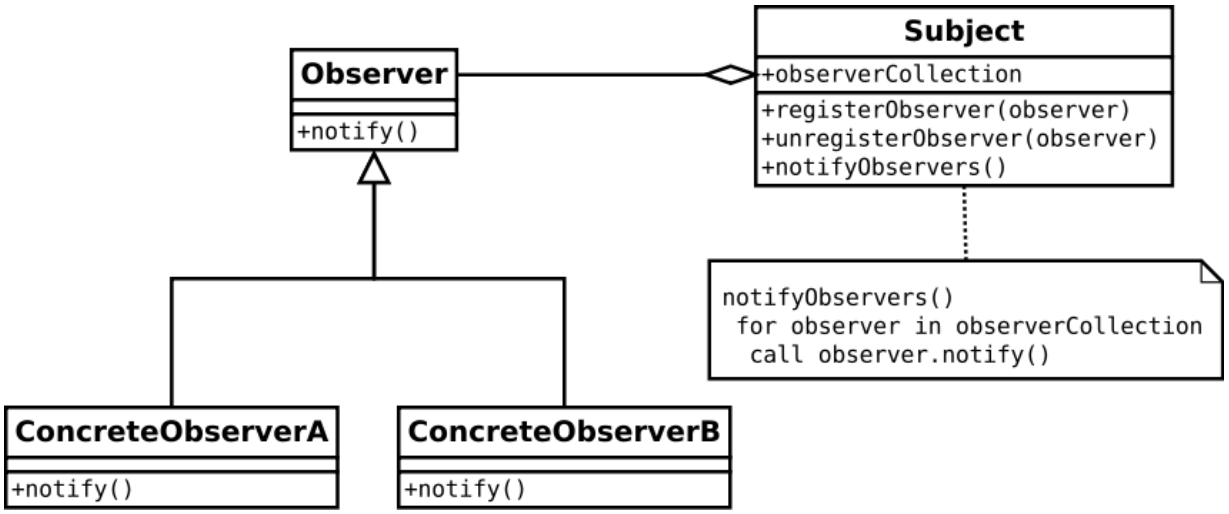
- Mehrere Objekte sollen unmittelbar informiert werden wenn sich eines ändert (Beobachtung)
- Das beobachtete Objekt (Subjekt) kann nicht vorhersehen welche Beobachter es gibt
- Beobachter können wechseln

Lösung:

- Subjekt stellt Möglichkeit bereit, sich anzumelden (publish)
- Beobachter melden sich beim Subjekt an (subscribe)
- Subjekt aktualisiert angemeldete Beobachter bei Änderung

Verhaltensmuster – Beispiel 1

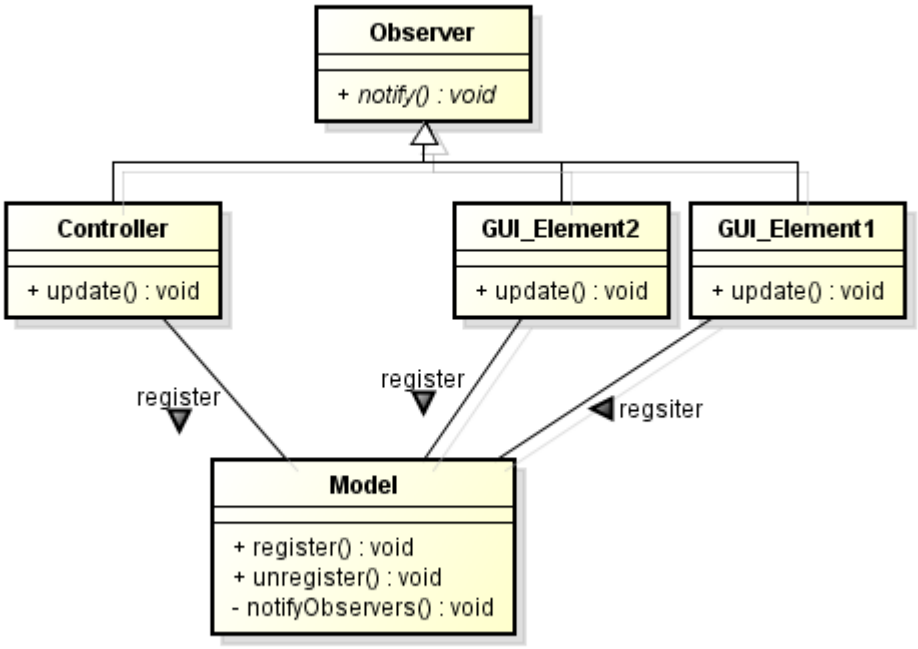
Struktur:



- Die Beobachter erweitern die abstrakte Klasse Observer
- Das Subjekt führt die Aktualisierung mit „notify“ durch

Verhaltensmuster – Beispiel 1

Beispiel: GUI mit MVC



- Model ist hier Subjekt
- Alle GUI-Elemente mit Inhalten aus dem Model sind Observer
- Auch der Controller kann Observer sein
- Grund: View und Controller sollten unmittelbar über Änderungen informiert werden
- Für Model nicht klar, welche GUI-Elemente es gibt

Verhaltensmuster – Beispiel 2

Name: Kommando (command)

Problem:

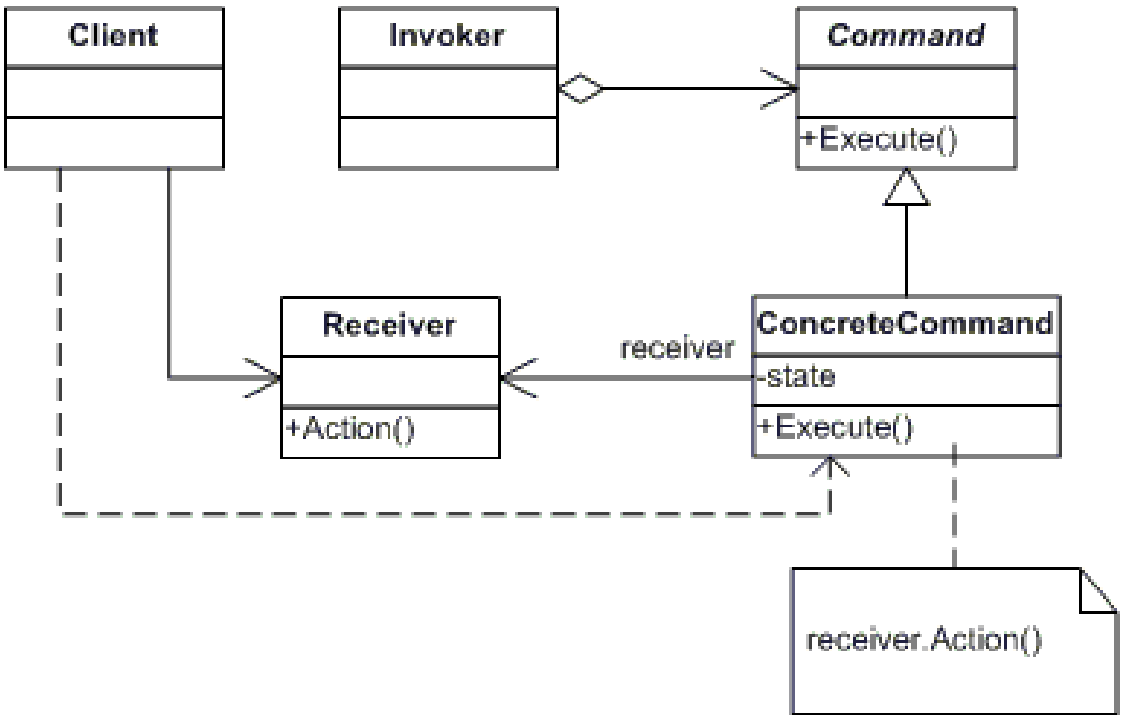
- Eine Anweisung soll nicht nur ausgeführt sondern auch verwaltet werden
- Beispiele:
  - Verzögerung einer Ausführung durch Warteschlangen
  - Parametrierung von Objekten (Clients) mit Anforderungen
  - Aufzeichnung von Anforderungen

Lösung:

- Command: Interface für die Ausführung von Operationen
- Client erstellt Command statt eine Operation direkt zu starten

Verhaltensmuster – Beispiel 2

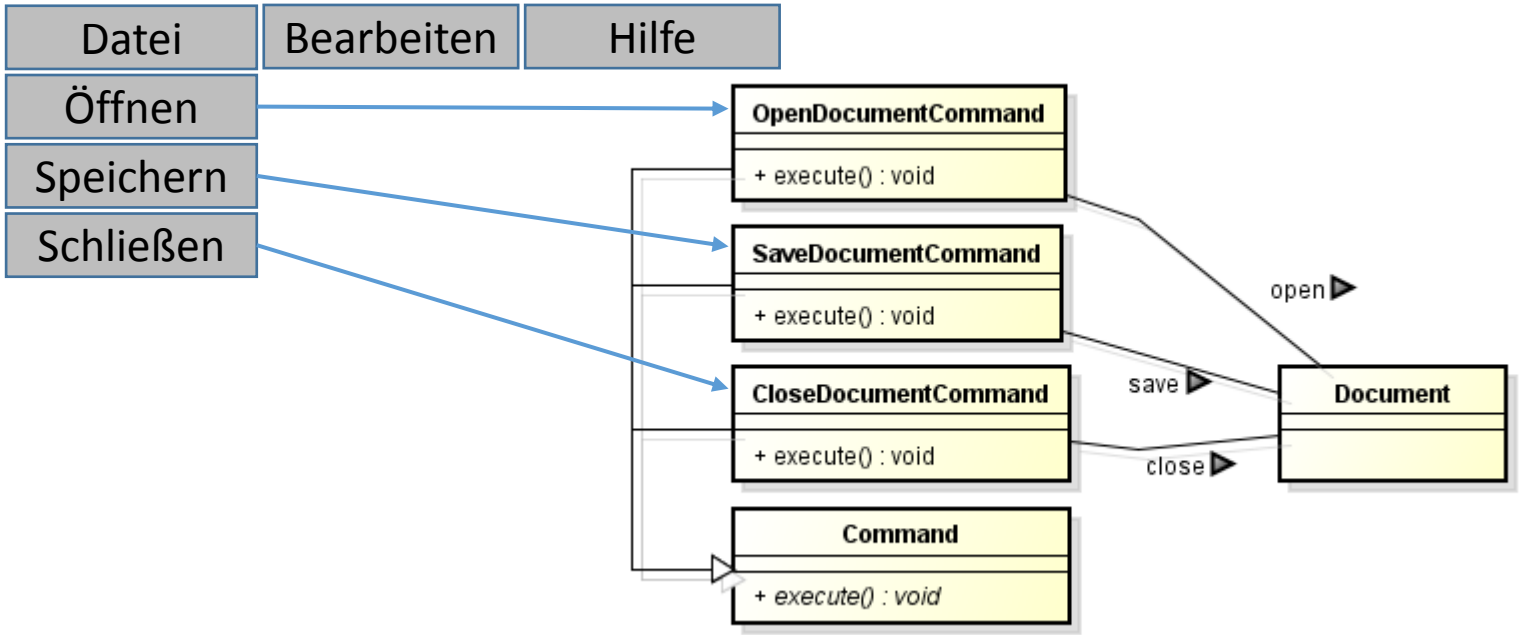
Struktur:



- Die Implementierung ConcreteCommand bildet die Verbindung zwischen Anforderung und Empfänger

Verhaltensmuster – Beispiel 2

- Beispiel: Flexible Menüstruktur
- Kommandos werden im GUI-Framework für Menübuttons konfiguriert



Zusammenfassung

- Implementierung stellt eigene Anforderungen an das Modell
- Durch Vielfalt der Probleme/Lösungsmöglichkeiten sind Vorgaben an Implementierung schwierig
- „Gute Erfahrungen“ in Mustern beschrieben
- Muster für Gesamtstruktur: Architekturstile
- Muster für bestimmte (Detail-)Probleme: Design Patterns

Abgrenzung schwierig, gehen oft ineinander über oder in Architekturstile

Quellen

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns
- Ian Somerville: Software Engineering, 9. Auflage
- Heide Balzert: Lehrbuch der Objektmodellierung, 2. Auflage