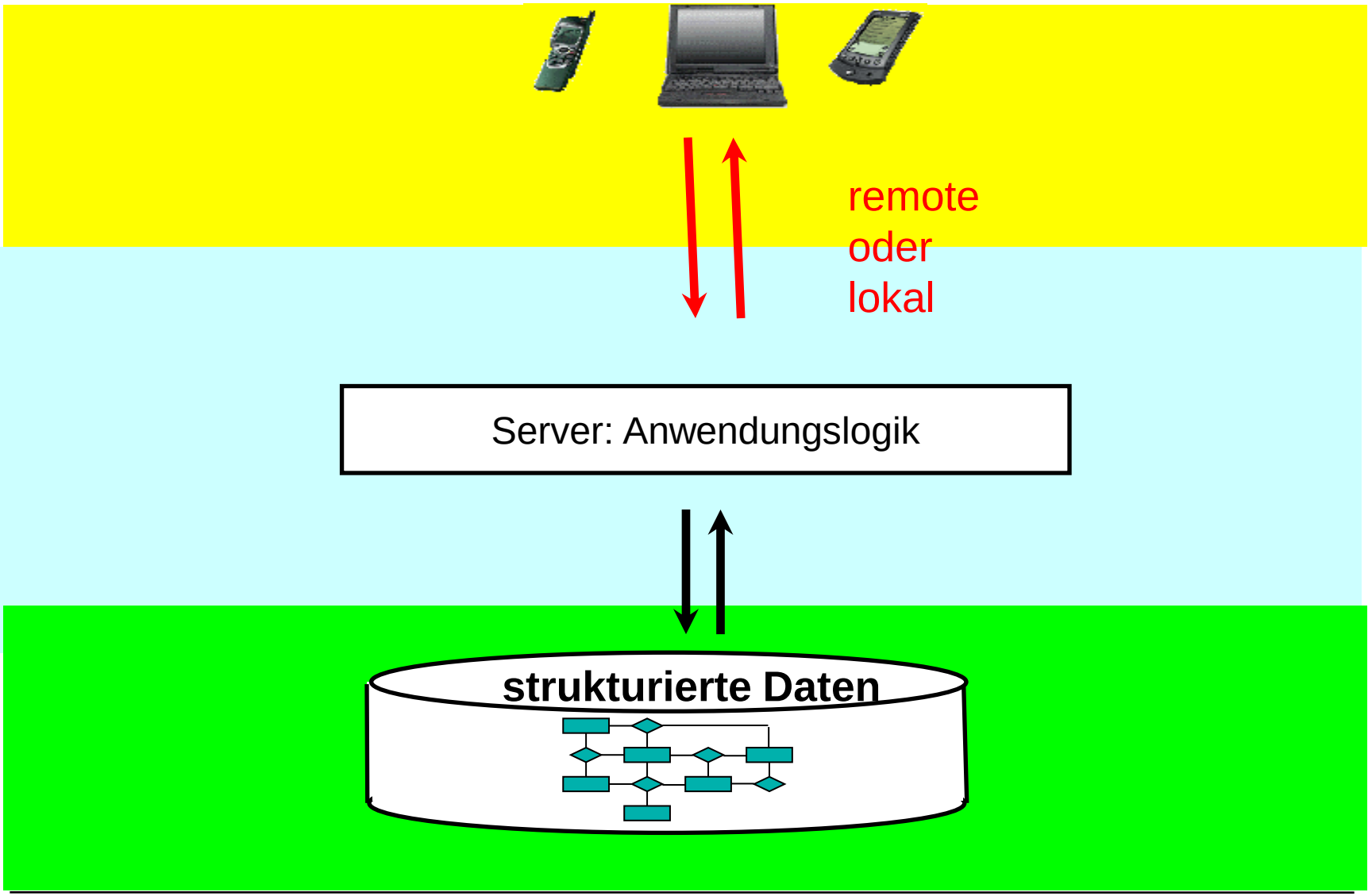
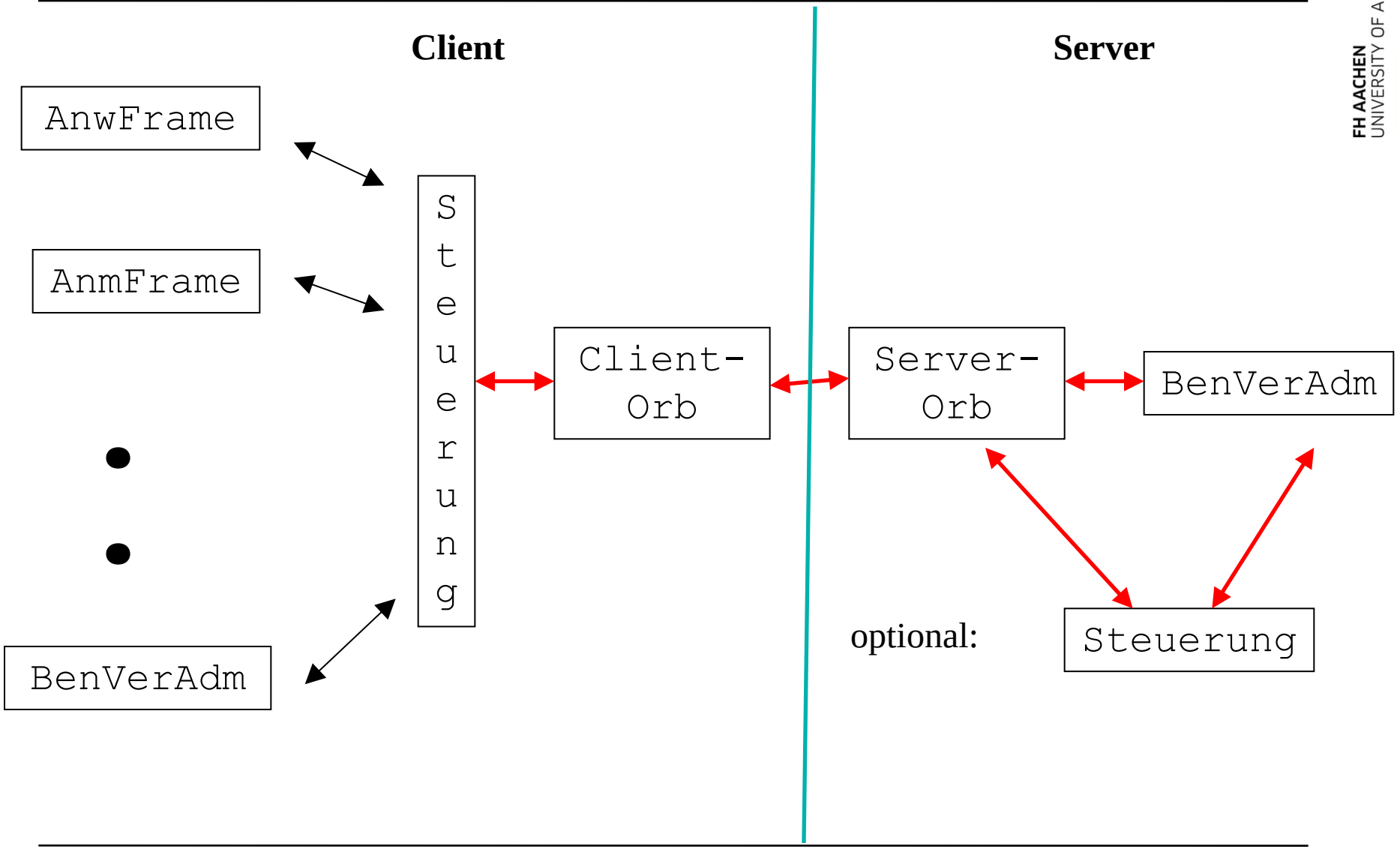
Architektur-Grobstruktur: Framework



Feinstruktur der Architektur



Methoden-Spezifikationsstruktur

- **Schnittstelle** (Name, Parametertypen, Rückgabetyp)
- **Vorbedingung** (vor Ausführung)
- **Nachbedingung** (nach Ausführung)
- **Invariante** (während Ausführung)
- **Semantik** (Beschreibung der Aufgabe und Bedeutung)

- **für Argumente Spezifikation von:**
  - Typprüfungen
  - Wertprüfungen

Fakultätsfunktion

```
int fak(int n) {
  int out = 1;
  for (int i=1; i<=n; i++) {
    out *= i;
  }
  return(out);
}
```

Vorbedingung: **n>=0**

Nachbedingung: **out = n!**

Invariante: **out = i!**

Semantik: **Fakultätsfunktion**

oder **fak(0) = 1**

fak(n) = n*fak(n-1); n>0

Design-Regeln/-Heuristiken

- möglichst **kohärente Operationen**
  nur eine Aufgabe pro Operation
- **anstelle von Funktionsmodi separate Operationen**
  Entscheidung über Semantik über Signatur
- **keine Nebeneffekte**
  durch globale Variablen oder ähnliches
- **keine Semantikänderung** bei Überlagerung in Unterklassen
- möglichst **allgemeingültig** entwerfen
  im Hinblick auf Schnittstellen, nicht auf Implementierung
- Verallgemeinerungen **bleiben abstrakt**
- **Maximiere innere Bindung von Klassen**
  zusammengehörende Verantwortlichkeiten in Klasse bündeln
- **Minimiere äußere Bindung von Klassen**
  möglichst kleine Schnittstellen

Design-Regeln/-Heuristiken (Forts.)

- pro Operation höchstens **eine Seite Code**
  sonst besser Cobol verwenden
- **einheitliche und treffende Namen, Typen, Parameter**
  auch Reihenfolge der Parameter wichtig
- möglichst **wenige switch / if-Anweisungen**
  Indiz für imperatives Denken
- **Berücksichtigung von Extremwerten & Robustheit**
  Minimum, Maximum, nil
- **keine künstlichen Grenzen**
  dynamisches Verhalten implementieren
- **Rückgängigfunktionen, Fehlerbehandlung, Nutzerberechtigung, spez. Konfigurationen** berücksichtigen
- **unternehmensspez. & allg. Standards** berücksichtigen

Historie der Design Pattern

- **von Architektur abgeguckt**
- **dort wichtigste Literaturstelle:**

  C. Alexander: *A Pattern Language: Towns, Buildings, Constructions.* Harvard University Press, Cambridge, MA,1977.
- **beschreibt Lösungen für immer wiederkehrende Probleme**
- **ersten Arbeiten in Software-Entwicklung in erster Hälfte der 90er**
- **absolutes Standardwerk „Gang of Four“:**

Literatur

- **(GoF)** E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA: 1995
  in Deutsch: Entwurfsmuster

- B. Goldfedder: *The Joy of Patterns.* Addison-Wesley: 2002
  in Deutsch: Entwurfsmuster einsetzen

- spezielle Pattern: z.B. für Java, EJB, …

1. Beispiel

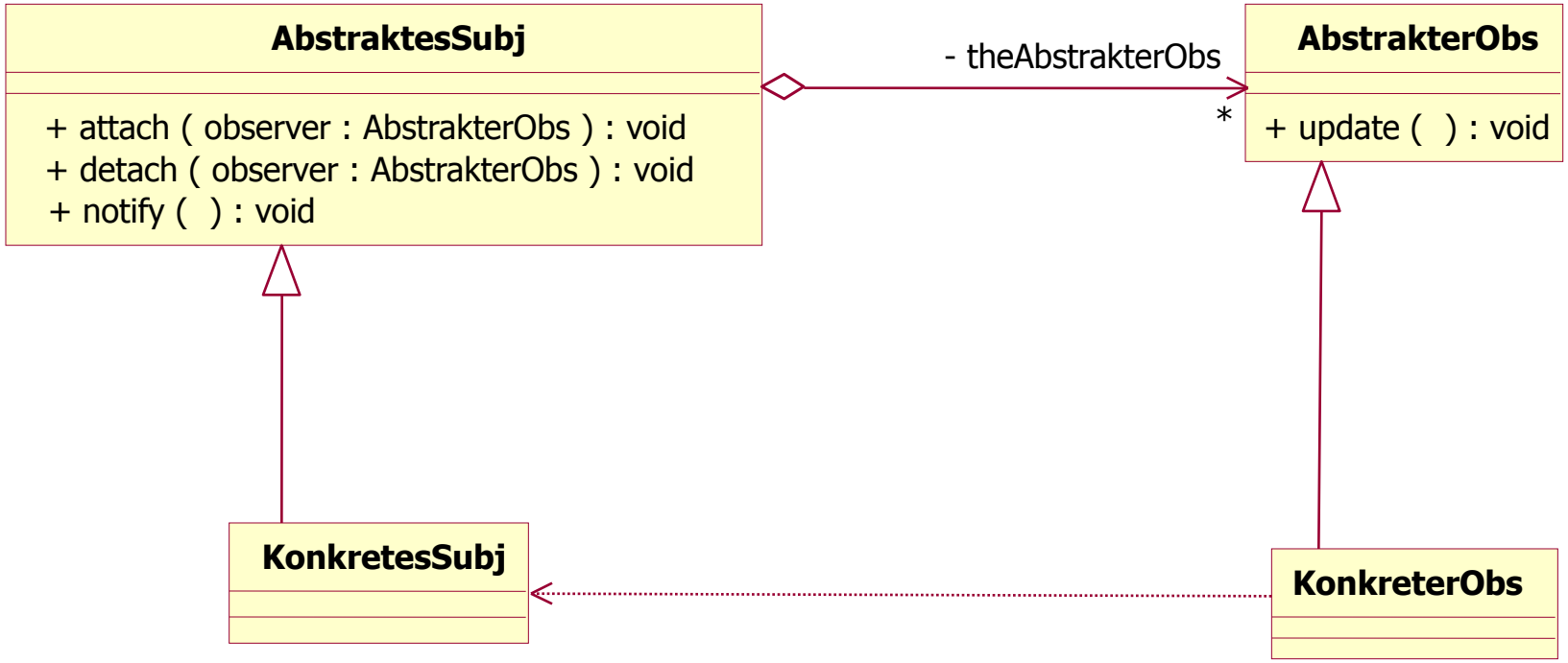
Aufgabe:

Es soll mehrere Darstellungen des Inhalts einer Excel-Tabelle oder einer Datenbank geben (z.B. Tabelle, Balkendiagramm, Kuchendiagramm, …)

Bei Änderung eines Tabelleneintrags soll die Änderung, falls relevant, in allen Darstellungen nachvollzogen werden.

Beschreiben Sie verbal ein Design zur Lösung dieser Aufgabenstellung!

Observer-Muster (Publisher-Subscriber)



Struktur Muster-Beschreibung (aus GoF)

- **Name**
  eindeutig & möglichst aussagekräftig
- **Zweck**
  Was macht Entwurfsmuster?
  Grundprinzip & Zweck?
  spezifische Fragestellung oder Entwurfsprobleme behandelt?
- **auch bekannt als**
  andere Namen für dieses Muster
- **Motivation**
  Bsp.-Szenario für Entwurfsproblem
  wie Muster Problem löst
- **Anwendbarkeit**
  in welchen Situationen und woran diese erkennbar

Struktur Muster-Beschreibung (Forts.)

- **Struktur**
  UML-Diagramme (in GoF noch OMT-Diagramme)

- **Teilnehmer**
  Beschreibung der beteiligten Klassen & deren Zuständigkeiten

- **Interaktionen zwischen Teilnehmern**

- **Konsequenzen**
  wie Ziele erreicht
  Vor- und Nachteile
  welche Ergebnisse
  welche Aspekte variabel

- **Implementierung**
  Fallen, Tipps und Techniken bei Implementierung
  sprachspezifische Aspekte und Impl.-Möglichkeiten

Struktur Muster-Beschreibung (Forts. II)

- **Beispielcode**
  zur Verdeutlichung seiner Umsetzung

- **Bekannte Verwendungen**
  mindestens 2 Beispiele aus unterschiedlichen Verwendungen

- **Verwandte Muster**
  Beziehungen zu anderen Mustern
  relevanten Unterschiede
  welche Muster zusammenverwendbar

Muster-Beschreibungen in anderen Büchern teilweise sehr verschieden

Observer-Muster aus GoF (Ausschnitt)

- **Zweck**
  Definiere eine 1-zu-n –Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und (automatisch) aktualisiert werden.

- **Anwendbarkeit**
  - Wenn Abstraktion zwei Aspekte besitzt, von denen der eine von dem anderen abhängt. Geteilte Kapselung ermöglicht, die Ob-jekte zu variieren und unabhängig wieder zu verwenden.
  - Wenn Änderung eines Objekts die Änderung anderer Objekte verlangt und man nicht weiß, wie viele geändert werden müssen.
  - Wenn Objekt andere benachrichtigen soll, ohne Annahmen zu treffen, wer diese anderen Objekte sind -> lose Kopplung

Observer-Muster aus GoF (Forts.)

- **Konsequenzen**
  - abstrakte Kopplung zwischen Subjekt und Beobachter
  - Unterstützung von Broadcasting-Kommunikation
- **Implementierung**
  - Abbildung von Subjekten auf ihre Beobachter: Merken der Beobachter in Liste, allerdings teuer (#Subjekte > #Beobachter)
  - Auslösen der Aktualisierung: Push/Pull?.
  - Fehlerhafte Referenzen auf gelöschte Subjekte
- **Bekannte Verwendungen**
  - Model-View-Controller: Model Subjekt, View Beobachter

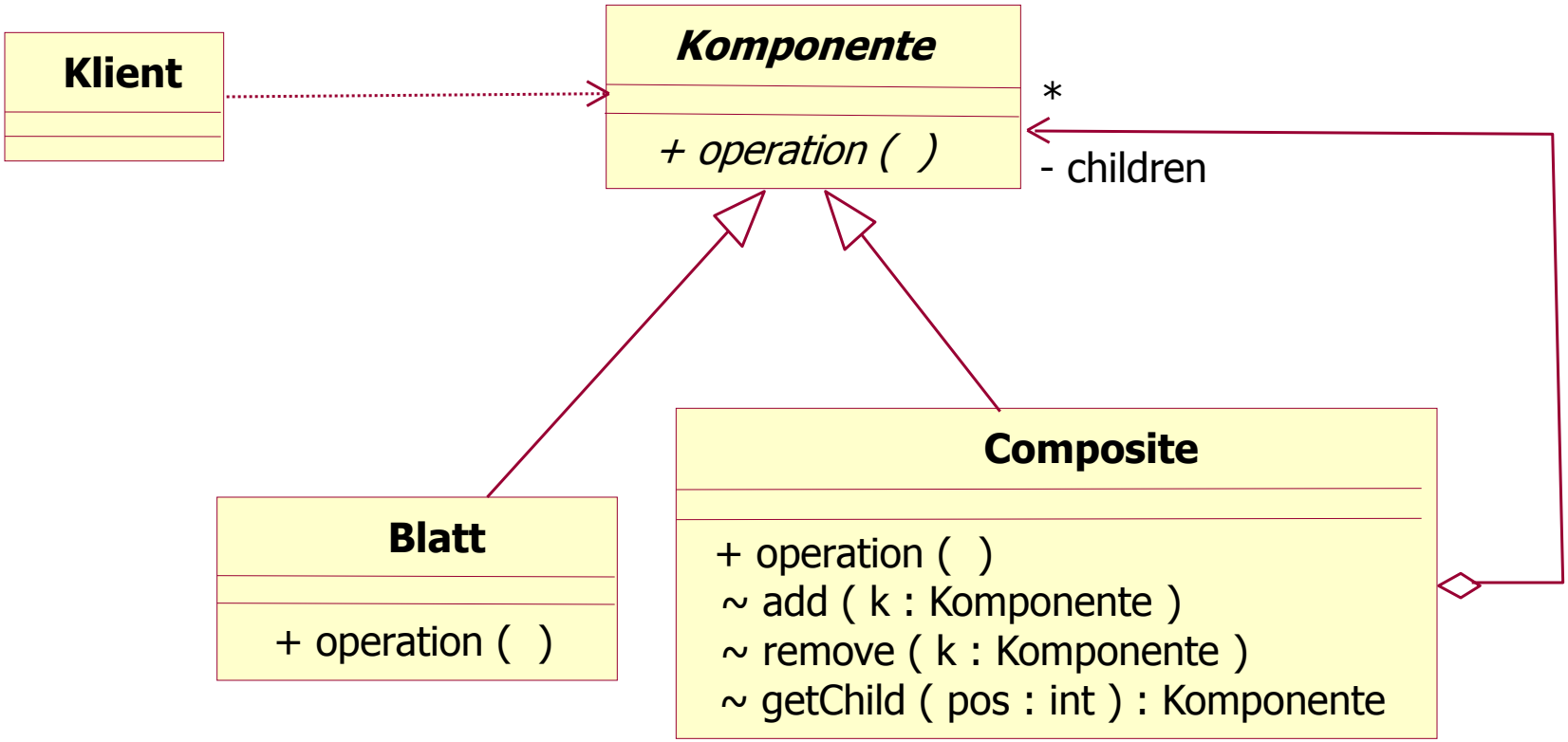
Datei/Verzeichnissystem

Aufgabenstellung:
Es soll ein Verzeichnissystem modelliert werden, bei dem die einzelnen Verzeichnisse aus Dateien oder wieder aus Verzeichnissen bestehen.

Für obiges System sollen Methoden zur Verfügung gestellt werden, die z.B. die Größe eines Verzeichnisses oder einer Datei liefern.

Der Client soll dabei nichts über die interne Struktur des Systems erfahren, insbesondere soll er beim Aufruf der Methoden nicht selbst unterscheiden müssen, ob es sich um eine Datei oder wieder um ein Verzeichnis handelt.

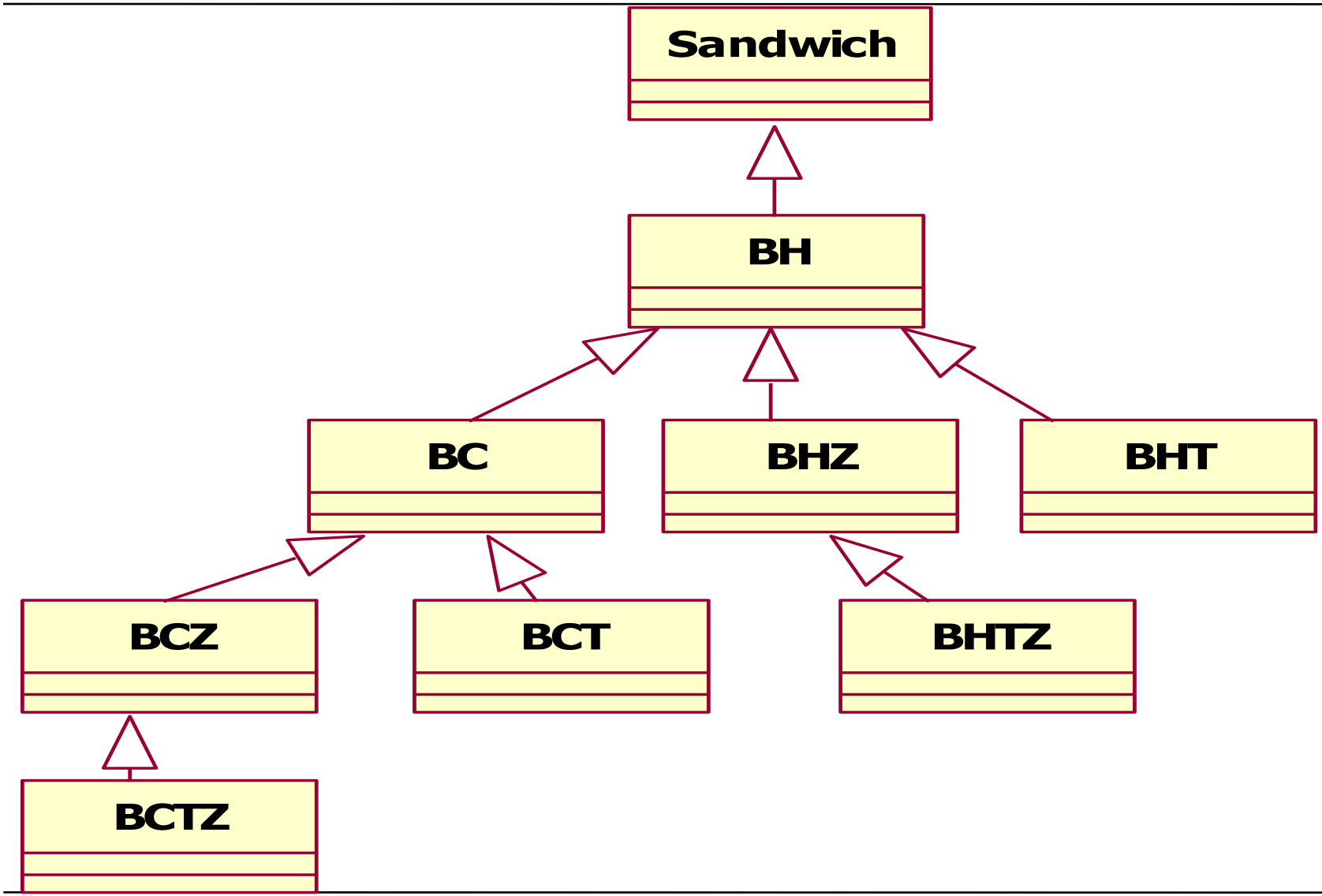
Composite-Muster-Struktur



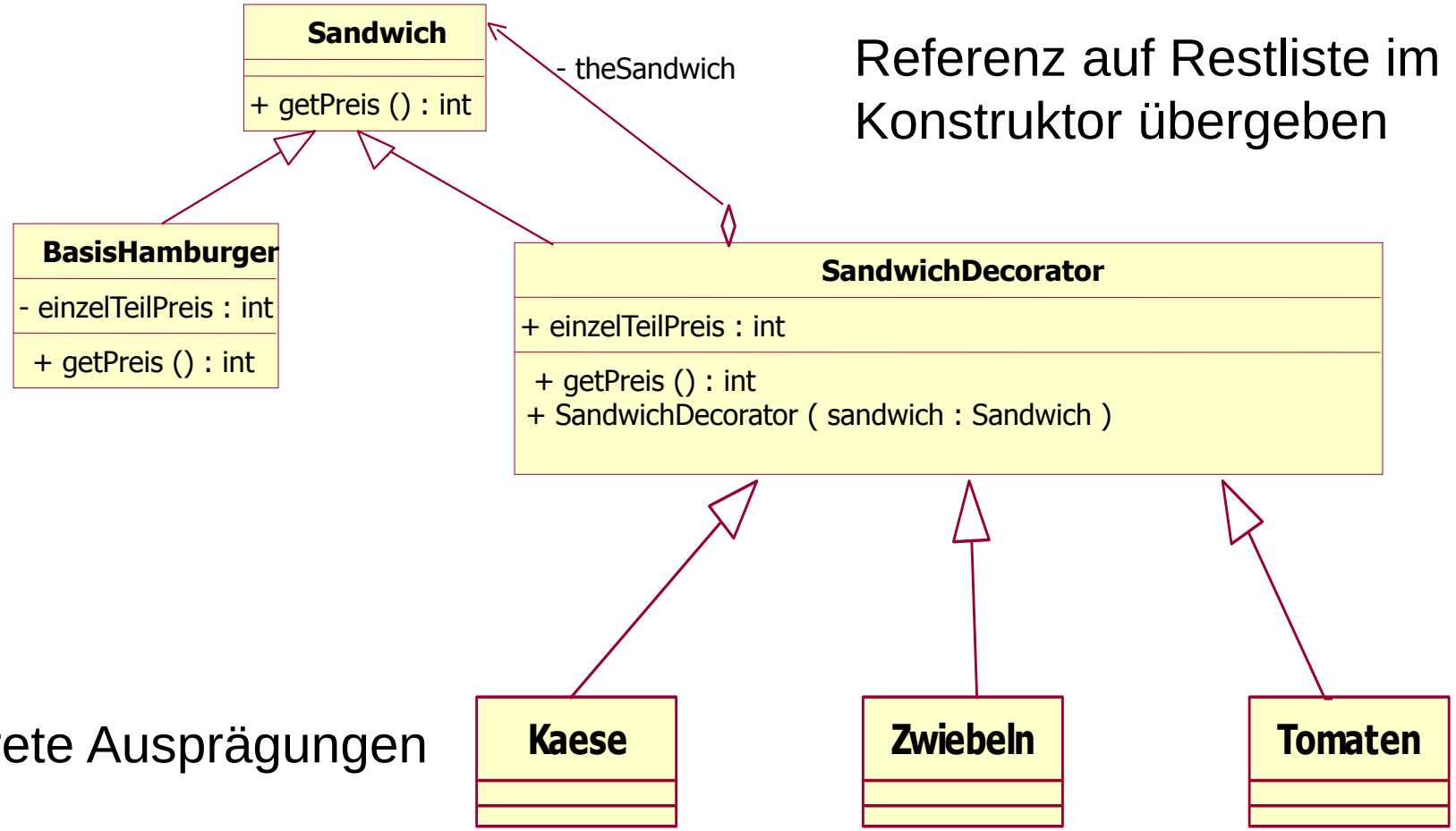
Susies BurgerShop

- **verkauft nur Burger, aber in allen Variationen:**
  - (BH) BasisHamburger
  - (BC) BasisCheeseburger
  - (BHZ) BasisHamburger mit Zwiebeln
  - (BCZ) BasisCheeseburger mit Zwiebeln
  - (BHT) BasisHamburger mit Tomaten
  - (BCT) BasisCheeseburger mit Tomaten
  - (BHZT) BasisHamburger mit Zwiebeln und Tomaten
  - (BCZT) BasisCheeseburger mit Zwiebeln und Tomaten
  - ...

Klassendiagr.: Käse, Zwiebeln, Tomaten



Decorator-Muster-Anwendung



Referenz auf Restliste im Konstruktor übergeben

konkrete Ausprägungen

Decorator-Muster allgemein

- **Darstellung einer Liste mit unterschiedlichen Einträgen**
- **Induktionsanfang: Listenende**

  Bsp.: BasisHamburger
  Allgemein: KonkreteKomponente

- **Induktionsschluss: Listenverlängerung**

  Bsp.: SandwichDecorator
  Allgemein: Dekorierer

- **Vereinigung der beiden Fälle**

  Bsp.: Sandwich
  Allgemein: Komponente

- **Wichtig: Aufbau durch Konstruktor, entspricht Vorhängen**

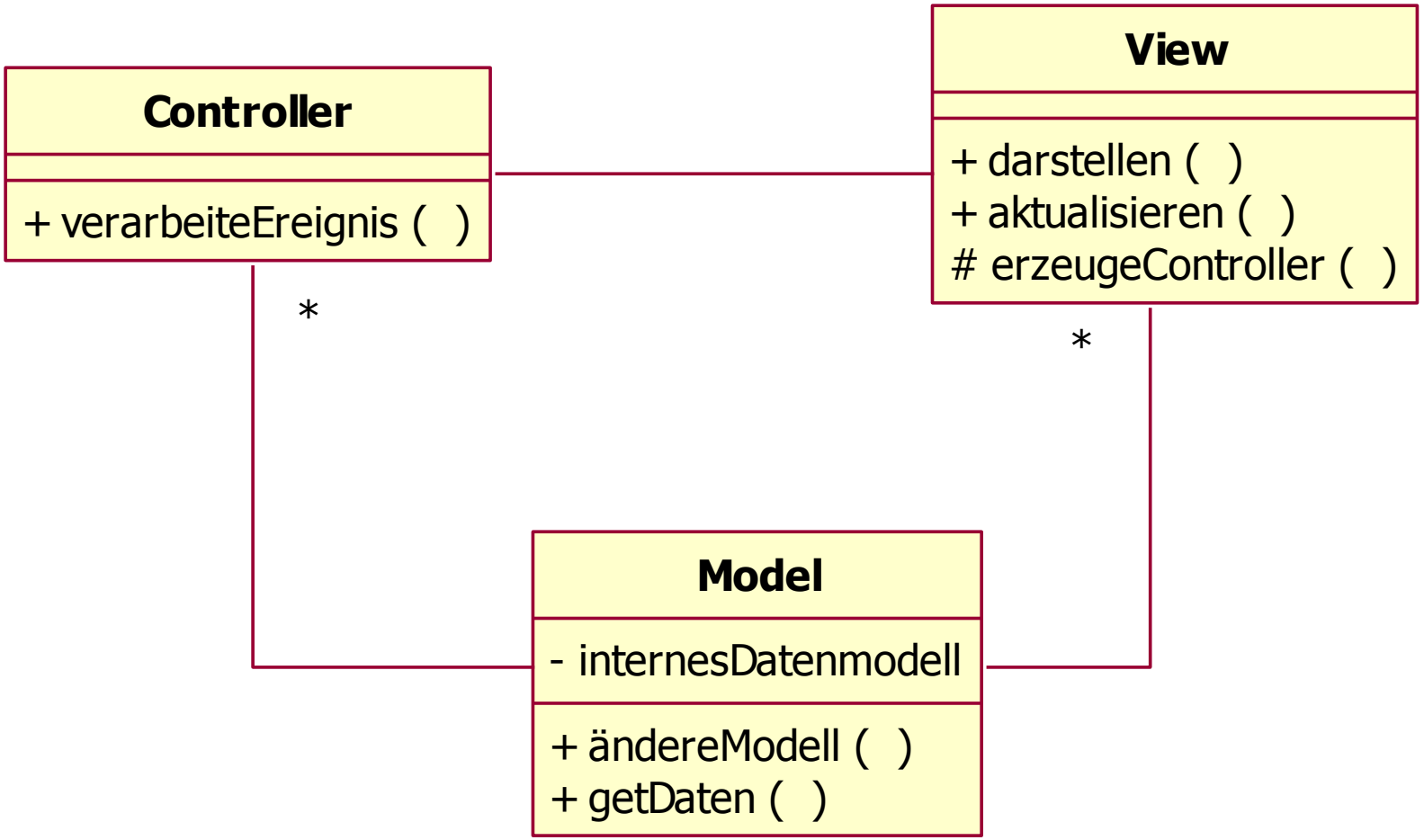
MVC-Architektur

- **Model**
  - kapselt die wesentlichen Daten der Anwendung
  - stellt Methoden zur Berechnung und Verarbeitung bereit
  - ermöglicht Zugriff auf darzustellende Daten

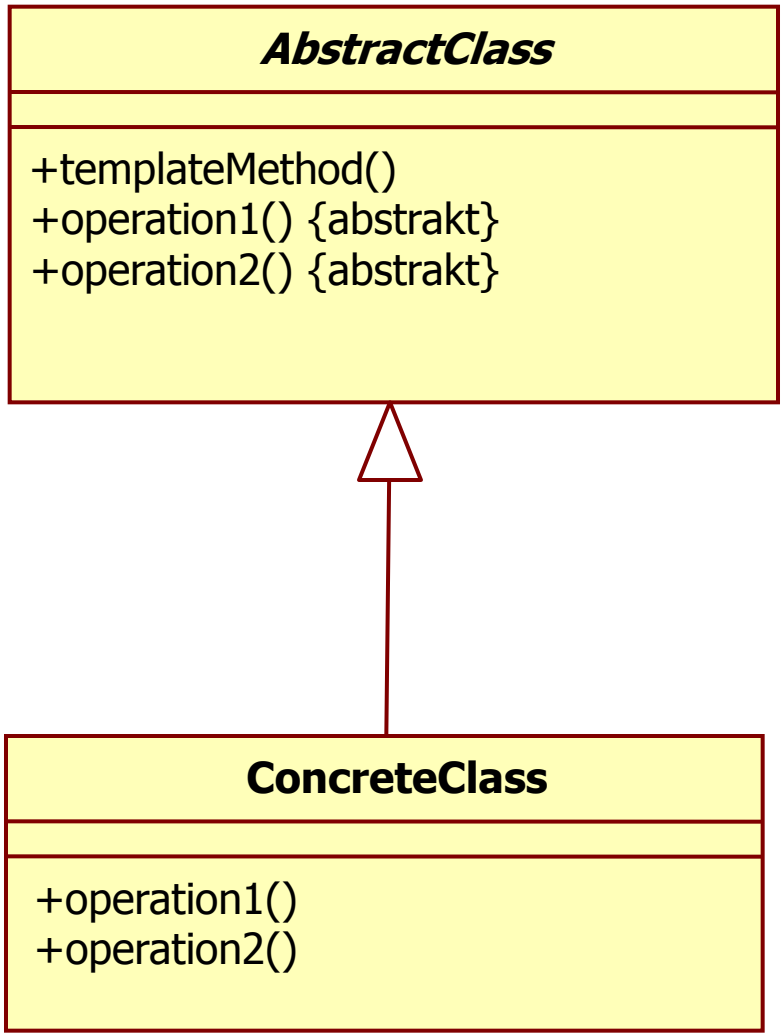
- **View**
  - präsentieren dem Benutzer Informationen
  - aktualisieren sich bei Veränderung dieser Informationen
  - erzeugen eine passende Controller-Komponente

- **Controller**
  - nehmen Benutzereingaben auf und behandeln diese
  - leiten Aufträge an Model weiter
  - veranlassen Views, sich zu aktualisieren

MVC-Pattern


Controller
+ verarbeiteEreignis ( )
View
+ darstellen ( )
+ aktualisieren ( )
# erzeugeController ( )
*
*
Model
- internesDatenmodell
+ ändereModell ( )
+ getDaten ( )

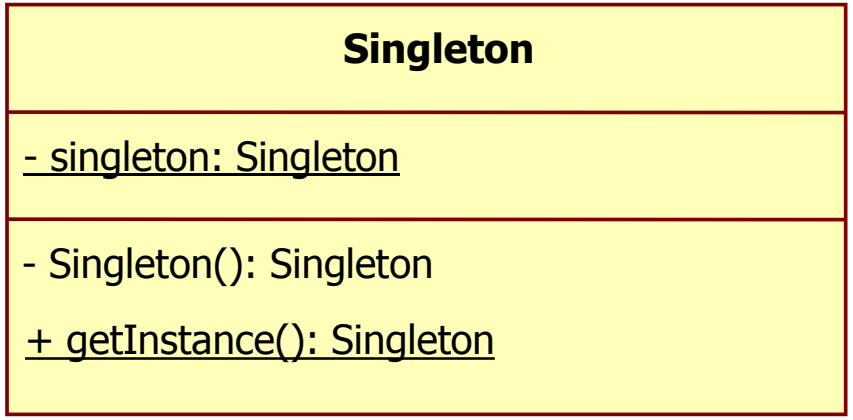
Template Method (Framework)



Verarbeitung der abstrakten Methoden 1 & 2 in templateMethod()

Implementierung der Methoden 1 & 2 und Erben von templateMethod()

Singleton

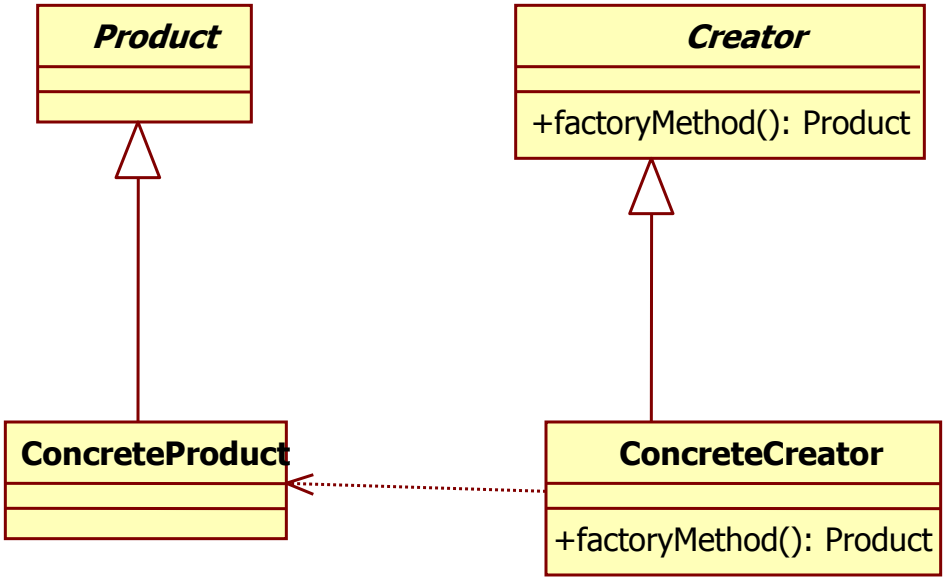


- einziges Objekt als privates statisches Attribut halten
- Konstruktor privat
- Zugriff auf Referenz durch statischen getter

Vorteile der Singleton-Verwendung:

- Sicherstellen, dass nur 1 Objekt verfügbar, z.B.: Drucker-Spooler
- Kontrolle und Steuerung der Erzeugung von Objekten der Klasse

Factory Method

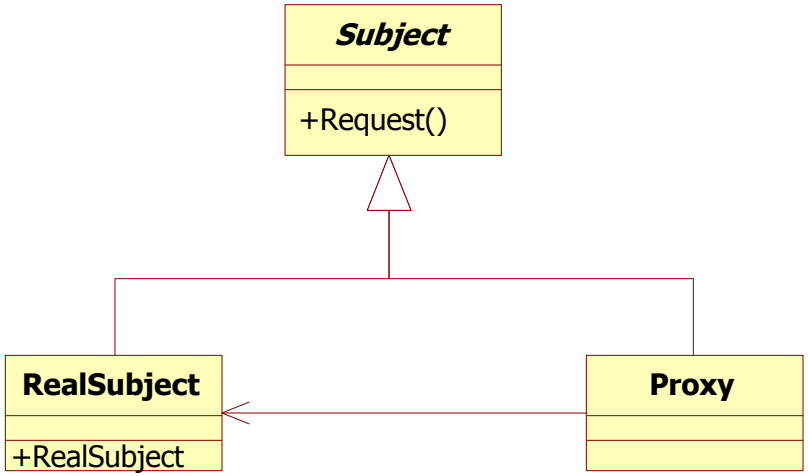


- Ersetze Konstruktoren der Unterklassen in Oberklasse durch factoryMethod()
- Überlagere factoryMethod() in Unterklassen durch Aufruf des Konstruktors zur Erzeugung von ConcreteProduct

Vorteile der Factory-Method-Verwendung:

- Zugriff auf Konstruktoren von Unterklassen, die noch nicht verfügbar
- Verarbeitung von Objekten von Unterklassen, die noch nicht verfügbar

Proxy-Pattern

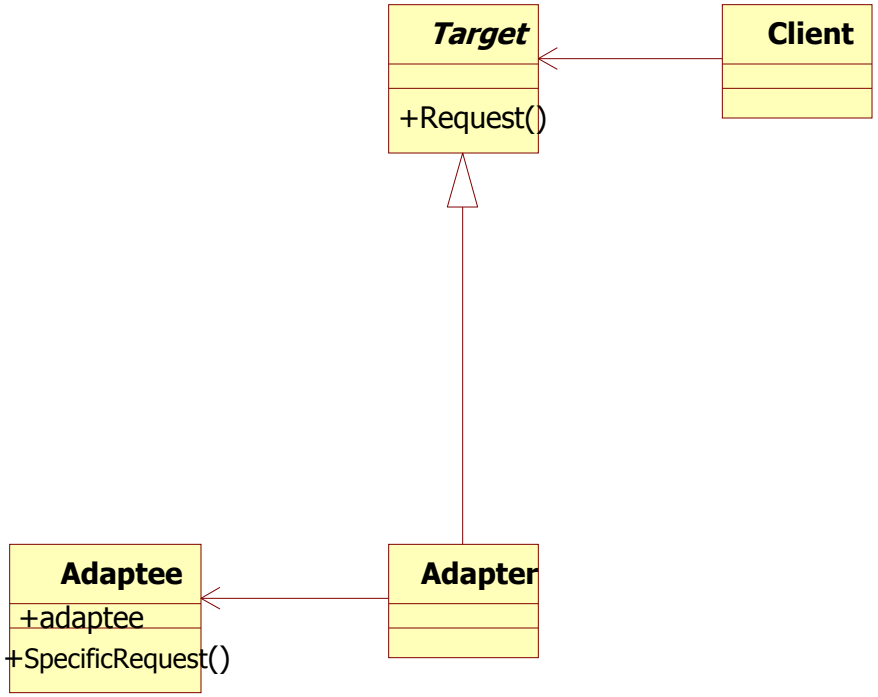


- Proxy implementiert gleiche Schnittstelle wie RealSubject
- Proxy hat Zugriff auf RealSubject-Objekt
- Client hat nur Zugriff auf Proxy
- Bsp.: Client-Orb in OOS

Vorteile der Proxy-Verwendung:

- dient auf Client-Seite als Ersatz für Server-Objekt (remote)
- kann Real-Subject erst bei Bedarf erzeugen (virtual)
- dient als Schutz für Real-Subject (protection)

Adapter-Pattern



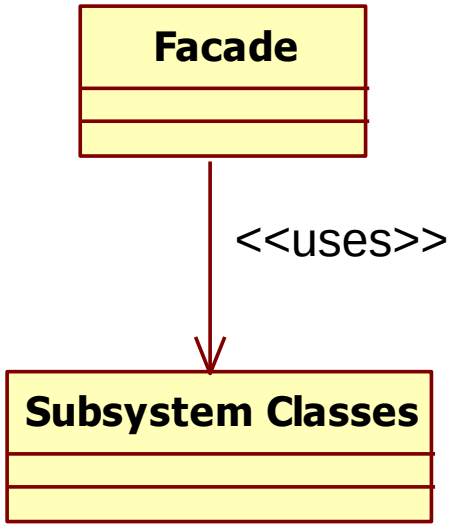
- Funktionalität vorhanden aber Schnittstelle passt nicht zum Client-Aufruf

->setze Adapter dazwischen, der Schnittstelle anpasst und vorhandene Funktionalität nutzt

Vorteile der Adapter-Verwendung:

- Wiederverwendbarkeit
- Kontrolle des Zugriffs

Facade-Pattern



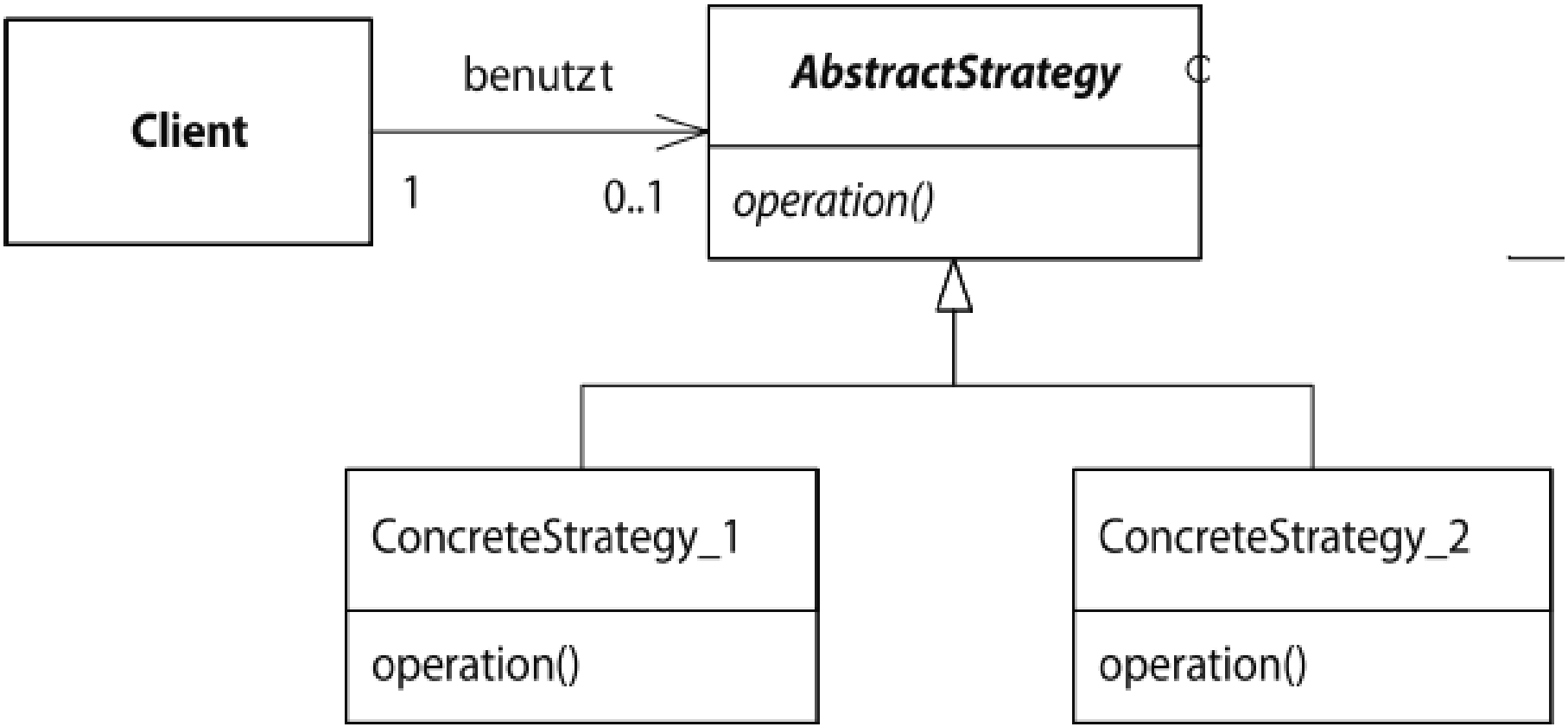
- Funktionalität wild auf mehrere Klassen verteilt.
- Client soll ohne Kenntnis der wilden Struktur auskommen

->Zugriff durch Client über neu definierte Facade

Vorteile der Facade-Verwendung:

- muss wilde Struktur des Systems nicht preisgeben
- Verwendung in verschiedenen Kontexten
- kontrollierter Zugriff

Strategy-Pattern (Diagramm)



Strategy-Pattern

- kapselt Algorithmus in Klasse
- soll unabhängig von Clients austauschbar sein

Bsp.:

  - Sortieralgorithmensystem
  - Feiertage in international nutzbarem Kalender

Vorteile der Strategy-Verwendung:

- Client nur von Abstraktion abhängig, nicht von Implementierung
- zusätzliche Implementierung hinzufügbar
- Flexibilität bei der Auswahl der Algorithmen