Technische Hochschule Ingolstadt

Fakultät für Elektrotechnik und Informatik

Zukunft in Bewegung

Entwurfsmuster

Software Engineering

Prof. Dr. Bernd Hafenrichter

Motivation

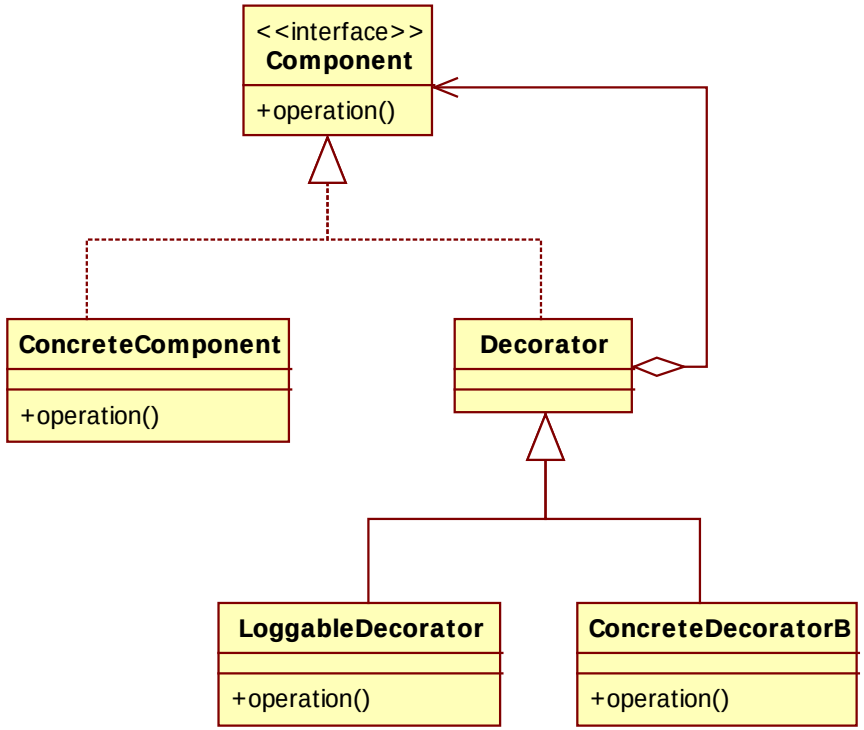
- Objekte existieren nicht isoliert
- Objekte werden kombiniert und zu größeren Einheiten zusammengefasst
- Strukturmuster befassen sich mit der Komposition von Klassen und Objekte, um größerer Strukturen zu bilden.
- z.B.: Baumstrukturen, …

Decorator

- **Zweck**:
  - Erweitere ein Objekt dynamisch um Zuständigkeiten
  - Dekoriere bieten eine flexible Alternative zu Unterklassenbildung
- **Beispiel:**
  - Erweitere die Elemente eine graphischen Oberfläche um die Möglichkeit des Scrollings
  - Füge eine Protokollierung zu einer Komponente hinzu
  - Erweitere eine Business-Schicht um eine Zugriffskontrolle

Decorator

- **Lösung:**



Decorator

Component:

- Deklariert die zur Verfügung stehende Schnittstelle

ConcreteComponent:

- Definiert ein Objekt welches die geforderte Schnittstelle implementiert

Decorator:

- Hält eine Referenz auf das zu dekorierende Objekt
- Hat die gleiche Schnittstelle wie die zu dekorierende Komponente

ConcreteDecorator

- fügt der dekorierten Komponente zusätzliche Funktionalität hinzu
- Überschreibt die die Angebotenen Methoden
- Delegiert Aufrufe an das dekorierte Objekt

Decorator

Bewertung:

Vorteile:

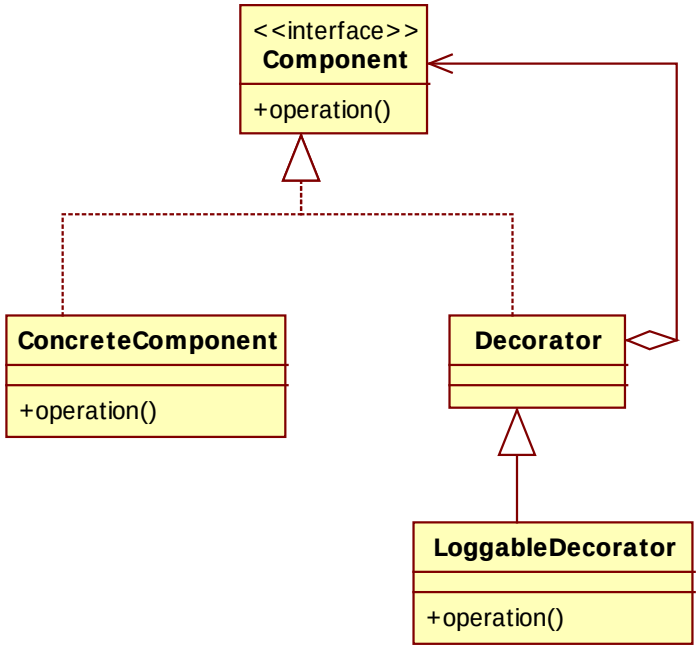
- mehrere Dekorierer können hintereinandergeschaltet werden können
- die Dekorierer können zur Laufzeit und sogar nach der Instanzierung ausgetauscht werden.
- Die zu dekorierende Klasse ist nicht unbedingt festgelegt (wohl aber deren Schnittstelle).
- lange und unübersichtliche Vererbungshierarchien können vermieden werden

Nachteil:

- Da eine dekorierte Komponente nicht identisch mit der Komponente selbst ist (als Objekt), muss man beim Testen auf Objekt-Identität vorsichtig sein.
- Ein Vergleich kann falsch ausgehen, obwohl dieselbe Komponente gemeint ist

Decorator

- **Beispiel: Protokollierbare Methodenaufrufe**



Decorator

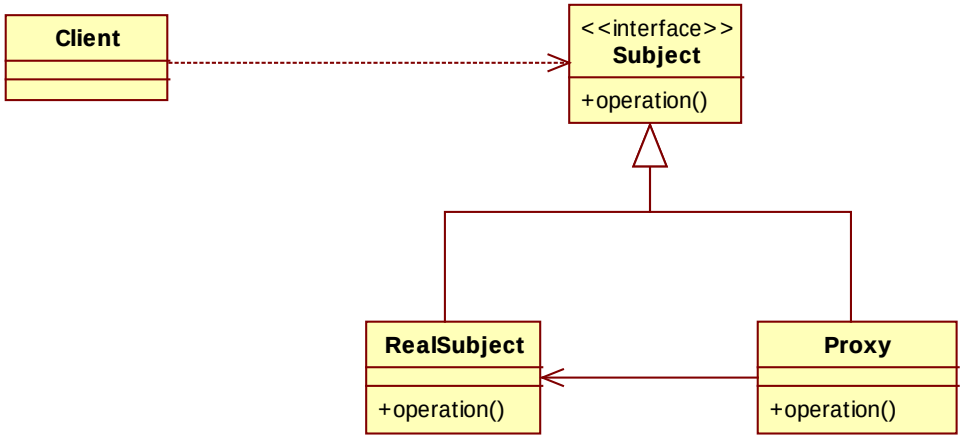
- **Verwendung (allgemein):**
  - Die Instanz eines Dekorierers wird vor die zu dekorierende Klasse geschaltet.
  - Der Dekorierer hat die gleiche Schnittstelle wie die zu dekorierende Klasse.
  - Aufrufe an den Dekorierer werden dann verändert oder unverändert weitergeleitet (Delegation), oder sie werden komplett in Eigenregie verarbeitet.
  - Der Dekorierer ist dabei „unsichtbar“, da der Aufrufende gar nicht mitbekommt, dass ein Dekorierer vorgeschaltet ist.

Proxy

- **Zweck**:
  - Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjektes
- **Beispiel:**
  - Lokaler Stub bei RMI
  - Partielles laden von Objekten

Proxy

- **Lösung:**



Proxy

Subject:

- Deklariert eine gemeinsame Schnittstelle von RealesSubjekt und Stellvertreter
- D.h. der Stellvertreter kann überall dort eingesetzt werden wo auch das RealeSubjekt engesetzt werden kann

Proxy:

- Verwaltet eine Referez auf das reale Objekt
- Kontrolliert den Zugriff auf das reale Objekt

RealSubject:

- Definiert das reale Objekt welches durch den Proxy geschützt werden soll

Proxy

Bewertung:

Vorteile:

- Zugriff auf kostspielige Objekte ist für den Benutzer transparent

Nachteil:

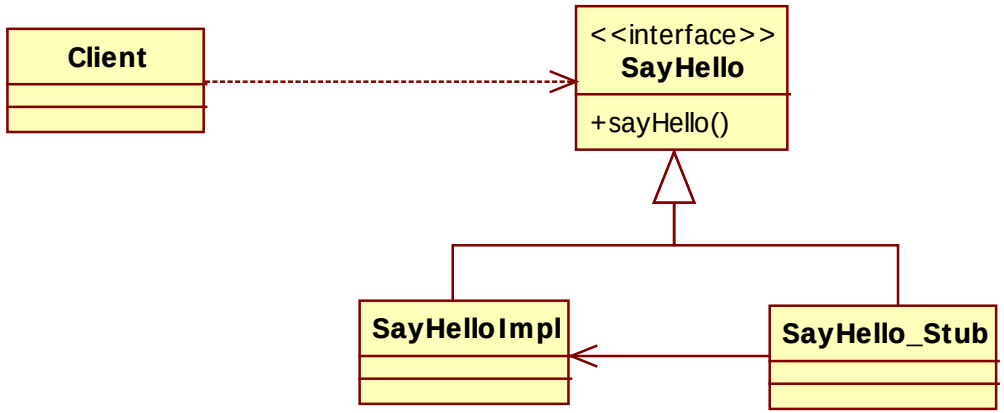
- Da ein Proxy nicht identisch mit der Komponente selbst ist (als Objekt), muss man beim Testen auf Objekt-Identität vorsichtig sein.
- Ein Vergleich kann falsch ausgehen, obwohl dieselbe Komponente gemeint ist

Proxy

- **Verwendung (allgemein):**
  - Remote-Proxy/Stub für verteilte Objekte
  - Virtueller Stellvertreter: Verzögerung "teurer" Operationen auf den Zeitpunkt des tatsächlichen Bedarfs.
  - Schutzproxy: Sicherstellen von Zugriffsrechten auf ein Objekt .

Proxy

- **Beispiel: RMI-Sub**

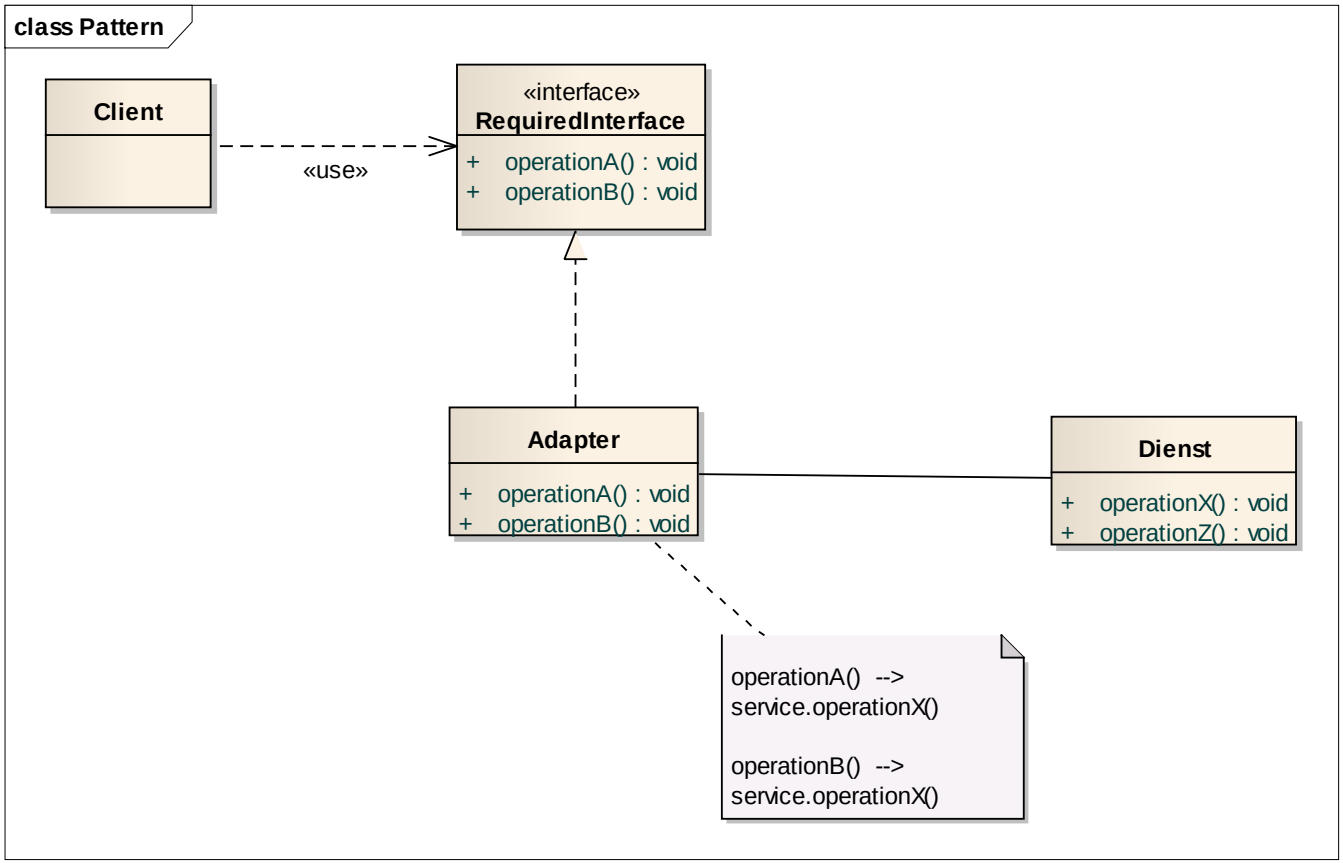


Adapter

- **Zweck**:
  - Ein Objekte bietet eine Schnittstelle/Methoden an welche aber für den Client nicht passend ist.
  - Biete eine adaptierte Implementierung an welche die geforderte Schnittstelle implementiert und auf das ursprüngliche Objekt umsetzt.
- **Beispiel:**
  - Lokaler Stub bei RMI
  - Partielles laden von Objekten

Adapter

- **Lösung:**



Adapter

RequiredInterface:

- Definiert das Interface welche von dem Client erwartet wird

Adapter:

- Übernimmt die Anpassung des Interfaces an die Implementierungsklasse
- Die Schnittstelle „RequiredInterface“ wird auf die Schnittstelle der Klasse Dienst angepasst

Dienst:

- Bietet implementierte Operationen an welche nicht mit der geforderten Schnittstelle kompatibel sind

Adapter

Bewertung:

Vorteile:

- Kommunikation zwischen unabhängigen Software-Komponenten
- Leichter Austausch von Klassen (neuer Adapter)
- Adapter können individuell für die geforderte Lösung angepasst werden

Nachteil:

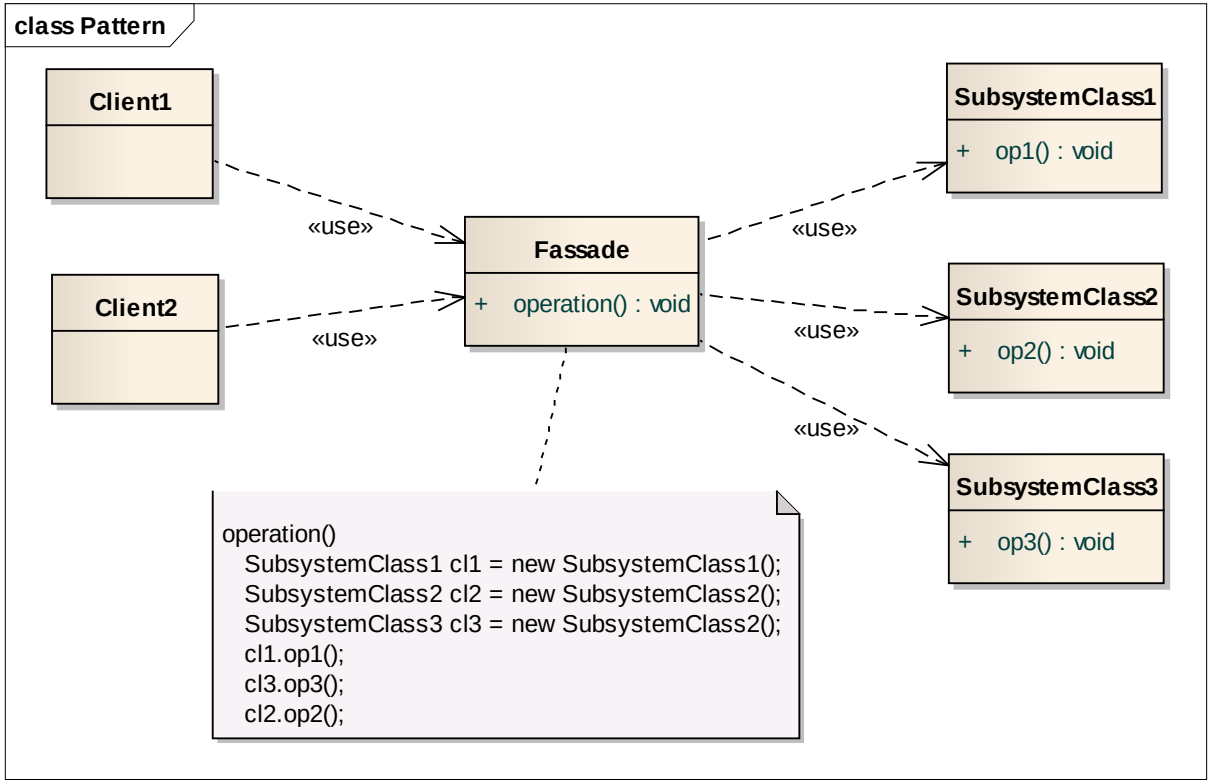
- Schlechte Wiederverwendbarkeit das ein Adapter spezielle für die vorhandene Dienst-Klasse und Interface angepasst wird
- Indirekter Funktionsaufruf durch Delegation könnte zu Performanceverlusten führen.

Fassade

- **Zweck**:
  - Biete eine einheitliche und vereinfachte Schnittstelle zu einer Menge von Schnittstellen oder Klassen eines Subsystems an
  - Enthält eine Subsystem viele technische Klassen welche selten von außen verwendet werden sollte ein Fassade verwendet werden. Die stellt eine vereinfachte Schnittstelle zur Verfügung.

Fassade

- **Lösung:**



Fassade

Fassade:

- Bildet eine einheitliche, vereinfachte Schnittstelle auf die Klassen eines Subsystems ab

SubsystemClass:

- Implementierungsklasse eines Subsystems welches technische Details enthält

Fassade

Bewertung:

Vorteile:

- Die Verwendung von Subsystemen wird vereinfacht
- Die Subsystemklassen sind unabhängig von der Fassade
- Lose Kopplung zwischen Subsystem und Client

Nachteil:

- Funktionsumfang eines Subsystems wird eingeschränkt
- Zusätzlicher, indirekter Funktionsaufruf (overhead)
- Äderungen des Subsystems bewirken Änderung an der Fassade

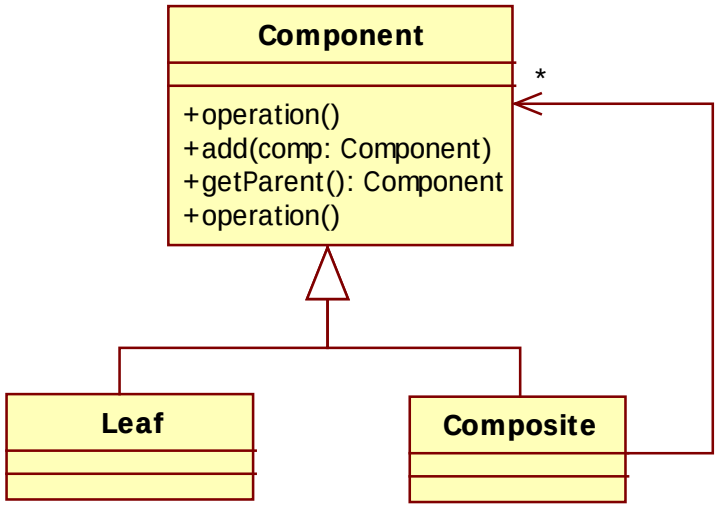
Kompositum

- **Zweck**:
  - Füge Objekte zu Baumstrukturen zusammen um Teile-Ganzes-Hierarchien zu repräsentieren
  - Einheitliche Verwendung von einzelnen als auch von zusammengesetzen Objekten

- **Beispiel:**
  - Java-Swing

Kompositum

- **Lösung:**



Kompositum

Component:

- deklariert die Schnittstelle für Objete in der zusammengefügten Struktur
- implementiert (falls angegeben) ein Default-Verhalten
- deklariert eine Schnittstelle zum Zugriff auf/zur Verwaltung von Kindobjekten
- optional eine Schnittstelle zum Zugriff auf die Elternobjekte

Leaf:

- repräsentiert Objekte die keine Kinder haben (Blattobjekte)
- Definiert verhalten für die primitiven Objekte

Composite:

- definiert Verhalten für Komponenten mit Kindern
- Speichert Kindobjektekomponenten/Implementiert kindobjekt-bzeogene Operationen

Kompositum

Bewertung:

Vorteile:

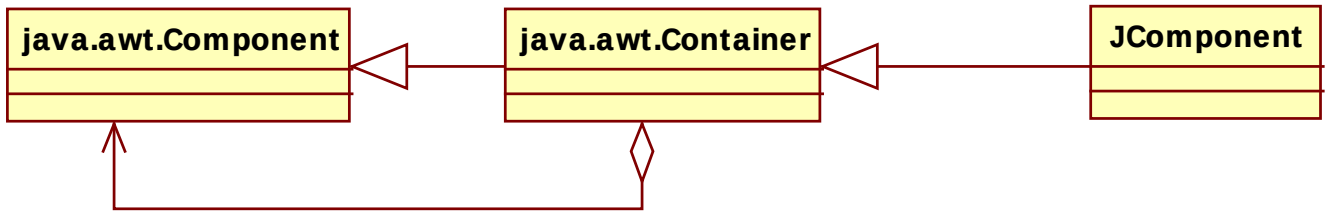
- einheitliche Behandlung von Primitiven und Kompositionen
- leichte Erweiterbarkeit um neue Blatt- oder Container-Klassen

Nachteil:

- Ein zu allgemeiner Entwurf erschwert es, Kompositionen auf bestimmte Klassen (und damit zumeist Typen) zu beschränken.

Kompositum

- **Beispiel: Java-Swing**



Kompositum

- **Verwendung (allgemein):**
  - Implementierung von Teil-Ganzes-Hierarchien
  - Verbergen der Unterschiede zwischen einzelnen und zusammengesetzten Objekten