5.6 Entwurfsmuster

5.6.1 Entwurfsmuster – Grundidee (Pattern)

- beschreiben ein **häufig auftretendes Problem**
  - treten in spezieller **Entwurfssituation** (Analyse und Design) auf
- dokumentieren **bekannte** und **erprobte Lösungen** (**wiederverwendbare Struktur)**
  - Abstraktion: Lösung auf Ebene oberhalb einzelner Klassen
  - zielen auf nichtfunktionale Eigenschaften
    (Änderbarkeit, Wiederverwendbarkeit, Erweiterbarkeit)

Beschreibung von Entwurfsmustern

- erfolgt meist in strukturierter Form (es gibt aber kein Standardformat), z.B.
- **Kontext**
  - Beschreibung der (Entwurfs-)**Situation** in der ein Muster auftritt
- **Problem / Szenario**
  - Beschreibung der **im Kontext wirkenden Kräfte**
    - Anforderungen
    - Randbedingungen
    - gewünschte Eigenschaften – konkretes Bespielszenario
- **Lösung**
  - Beschreibung einer **Lösung**, die den Kräften Rechnung trägt
    - Klassenmodell mit Beziehungen

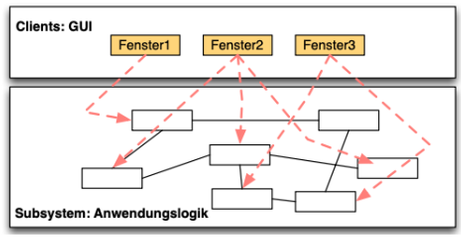
5.6.2 Fassade-Pattern

Motivation - Kontext

wie kann der Zugriff auf ein komplexes (Sub-)System vereinfacht werden?

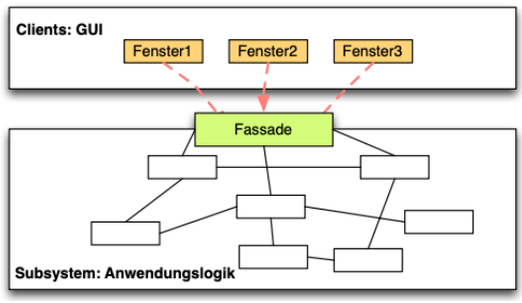
- **Komplexität** eines Subsystems soll **verborgen** werden **(Kapselung)**
  - Client kennt nicht innere Struktur des Subsystems
- Software-Schicht n weiß nur wenig über darunter liegende Schicht n-1s

Problem / Szenario



- Fensterklassen (GUI) nutzen sehr viele Klassen der Anwendungsschicht
  - **starke Kopplung**
  - **hoher Änderungsaufwand**

Lösung



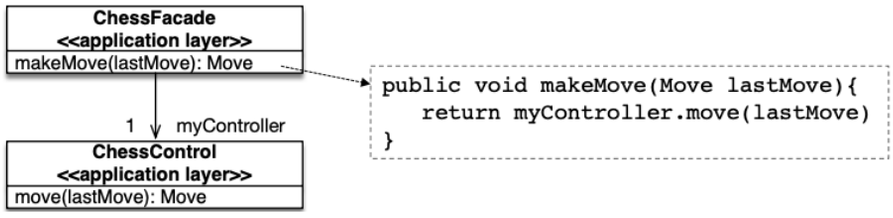
- Einführung einer/mehrerer **Fassade**-Klasse(n) (**Facade**)
  - Fassade kapselt das Subsystem:
    bietet **zentralen Zugriff** (Schnittstelle) auf das Subsystem
  - Fassade-Methoden enthalten keine Anwendungslogik, sondern delegieren Methodenaufrufe an das Subsystem
  - **Clients nutzen Subsystem ausschließlich über Fassade (abstract)**

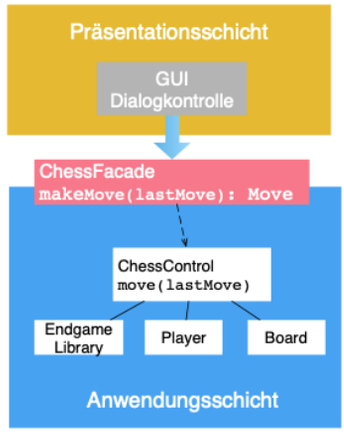
Vorteile:

- **geringe Abhängigkeit** zwischen Clients und Subsystemen (→ Entwurfsprinzip Lose Kopplung)
  - die Clients kennen nur die Fassade, aber nicht die innere Struktur des Subsystems
  - die Verwendung des Subsystems vereinfacht sich
  - das Subsystem kennt nicht seine Clients
- Robustheit / Wartbarkeit:
  - Änderungen der inneren Struktur des Subsystems schlagen nicht auf die Clients durch

Delegation (Forwarding)

- Fassaden arbeiten oft mit **Delegation**
  - **Methodenaufruf** an ein anderes Objekt **transparent weiterleiten**
- Beispiel
  - die Fassade-Methode **`makeMove()`** berechnet einen neuen Zug durch **Delegation** an Controller-Klasse **`ChessControl`**





Abgrenzung

Fassade versus Boundary-/ Control-Klassen des Unified Process

- **Fassade-Klasse**
  - ist ganz allgemein eine Klasse, die ein **Subsystem kapselt!**
  - **Fassaden sind allgemeiner als Boundary-Klasse, da diese auch technische Schnittstellen anbieten können**

- **Boundary-Klasse**
  - ist eine Klasse, die die fachliche Schnittstelle bereitstellt
  - **guter Entwurf**:
    i.d.R. werden die Boundary-Klassen als Fassade verwendet, die dann mittels Delegation Methoden von Control-Klassen aufrufen

- **Control-Klasse (siehe Kap. 4.2)**
  - ist eine Klasse, die keine Daten enthält, sondern **Abläufe realisiert**
  - **schlechter Entwurf**:
    manchmal wird eine Control-Klasse als Fassade verwendet

5.6.3 Observer-Pattern

Motivation

Kontext

- Änderungen in einer Komponente wirken sich auf andere SW- Komponenten aus
- ein (oder mehrere) Objekte soll(en) automatisch auf die Änderung eines anderen Objektes reagieren
  - **Konsistenz** zwischen Objekten soll gewährleistet werden
  - **lose Kopplung** soll aufrechterhalten werden
- Software-Schicht n-1 kennt darüber liegende Schicht n nicht

Problem

- typischer Anwendungsfall:
  - GUI-Fenster zeigen Daten des selben fachlichen Objekts an
  - Datenänderung in einem Fenster führt zu Aktualisierung aller anderen Fenster

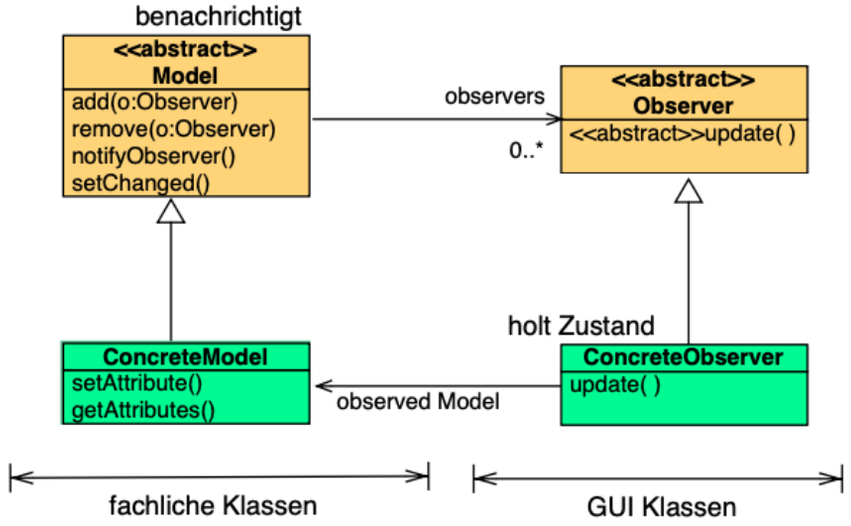
Observer-Pattern (GoF)

- Wenn Zustand von ConcreteModel verändert wird durch setter-Methoden, dann wird notifyObserver() ausgeführt. NotifyObserver() geht durch die Liste aller Observer und führt update() aus von den ConcreteObserver, wo die getter-Methoden der ConcreteModel aufgerufen werden

Lösung

– Observer-Pattern (Beobachter-Pattern) realisiert Benachrichtigungs-Mechanismus

– Objekt (Model) benachrichtigt andere Objekte (Observer) sobald es sich geändert hat



Verantwortlichkeit

Model (fachliches Objekt, Subjekt)

- ist **beobachtbar** (observable)
- kennt seine **Beobachter** (observers)
- bietet eine Schnittstelle zum An- und Abmelden von Beobachtern **`(add()/remove())`**
- **`notifyObservers()`** ruft für alle Observer deren **`update()`**–Methode auf

Concrete Model (fachliches Objekt)

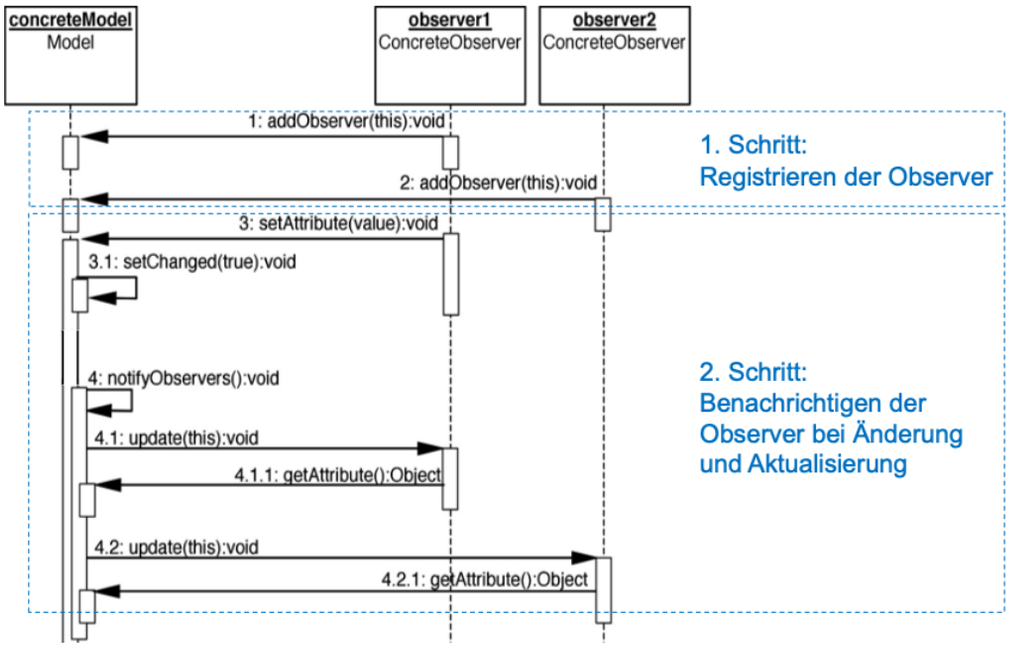
- speichert konkrete Daten in seinen Attributen
- benachrichtigt seine Beobachter, wenn es sich geändert hat, in dem es **`notifyObservers()`** zu geeignetem Zeitpunkt aufruft

Observer

- stellt eine Aktualisierungsschnittstelle zur Verfügung: Methode **`update()`**

ConcreteObserver (z.B. GUI-Klasse)

- ist **Beobachter** fachlicher Objekte (ConcreteModel)
- verwaltet eine Referenz auf das beobachtete fachliche Objekt (**`observedModel`**)
- implementiert die Aktualisierungsschnittstelle(=**`update()`**)
  - wird von beobachteten Objekten (Model) aufgerufen
  - holt sich die Daten aus dem Model, um Model-Änderungen zu berücksichtigen



Bewertung Observer-Pattern

Vorteile

- lose Kopplung
  - Beobachter (Fenster) und fachliche Klassen (Datenquellen) sind **entkoppelt**
- Model- und Beobachter-Klassen können zu unterschiedlichen Abstraktionsschichten im System gehören, das Schichtenmodell bleibt intakt
- Erweiterbarkeit
  - Beobachter können problemlos hinzugefügt und entfernt werden
- automatische **Synchronisation** von Beobachtern

Nachteile

- zusätzliche **Komplexität**
- wann und wie oft müssen bestimmte Observer benachrichtigt werden? (ggf. hohe **Anzahl von Aktualisierungen** / Datenzugriffen)

- Sollte nicht über mehrere physische System hinweg gestreut werden. Nur innerhalb logischen Schichten innerhalb eines Systems.
- Verschiedene Benachrichtigungsmöglichkeiten. Der Observer oder das Datenobjekt

Durch Beobachter (Observer)

Vorteil

- Mehrere Benachrichtigungen können zusammengefasst werden

Nachteil

- Man darf die Benachrichtigung nicht vergessen

Durch Datenobjekt (Client)

Vorteil

- Es wird nicht vergessen

Nachteil

- Es können sehr viele Benachrichtigungen entstehen

5.6.5 Kompositum-Pattern

Verantwortlichkeit - Kompositum (Composite) Kontext

- **rekursive Hierarchien** von Objekten implementieren
  - primitive (atomare) Objekte
  - und Container: enthalten atomare Objekte und wiederum Container
- **Objekte und Container sollen gleichbehandelt werden, ohne ihren Typ bestimmen zu müssen**

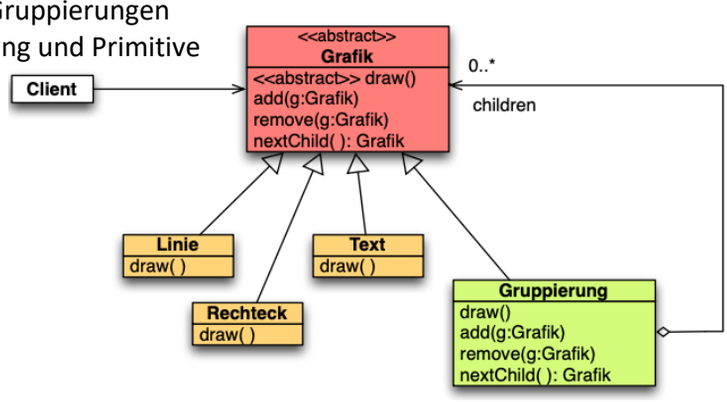
Problem / Szenario

- in einem Grafik-Editor kann man
  - Grafik-Primitive (Linien, Kreise, Rechtecke, ...) gruppieren
  - hierarchische Strukturen bilden, z.B.
    - Gruppierungen enthalten Grafik-Primitive
    - und ggf. wiederum Gruppierungen
  - Editor (Client) behandelt Gruppierungen und Primitive gleich (verschieben, kopieren, ...)

Beispiel – Lösung für Grafikeditor

- Vorteil: einfach erweiterbar, Client kennt nicht atomare Objekte und muss nicht zwischen Grafik und Gruppierung unterscheiden
- Container-Methoden müssen in Grafik leer sein, damit die atomaren Objekte diese nicht aufrufen können

Grafik:	**abstrakte Oberklasse** für Primitive und Gruppierungen (enthält leere Methoden für add(), nextChild() etc.)
Grafik-Primitive:	Linie, Rechteck, Text, ...
Gruppierung:	Container für Primitive und (rekursiv!) Gruppierungen
Client:	unterscheidet nicht zwischen Gruppierung und Primitive

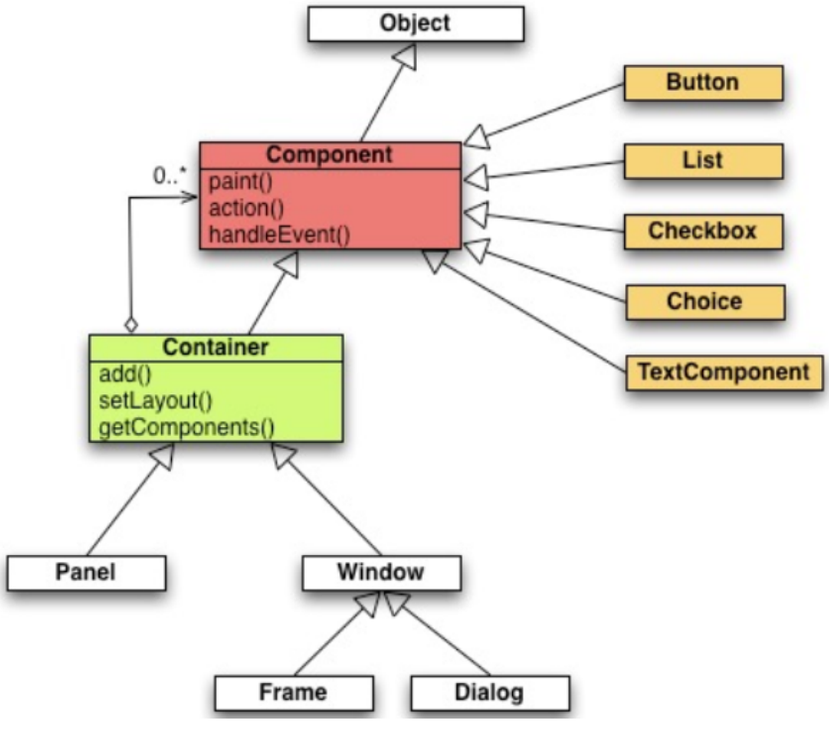


- Gruppierung ruft draw() der childrens auf

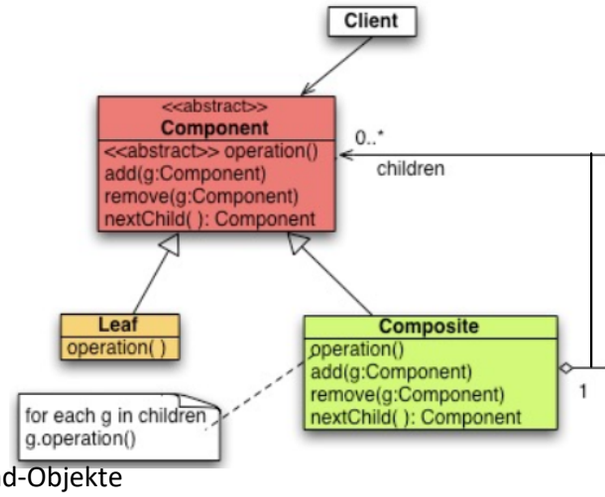
```
class Gruppierung extends Grafik{
    ArrayList<Grafik> children;
    public void draw() {
        for (int i = 0; i < children.size(); i++)
            children.get(i).draw();
  }
}
```

```
class Linie extends Grafik{
    public void draw () { // Zeichnen einer Linie }
}
```

- Abweichung vom Standard-Pattern: Container-Methoden nicht in Component enthalten



Kompositum Lösung (GoF)



Component (Grafik)

- deklariert **Schnittstelle für Unterklassen**
  - für die Verwaltung von Kind-Objekten
  - implementiert ggf. leere **`add`**- und **`remove`**-Methoden
  - implementiert ggf. Default-Verhalten für **`operation()`**

Leaf (Linie, ...)

- implementiert **Verhalten für atomare Objekte: operation()**
  - keine Container-Funktionen

Composite (Gruppierung)

- **Container für Component-Objekte**
  - implementiert die Container-Methoden
  - delegiert die **`operation`**-Methode: Iteration über alle Kind-Objekte

Bewertung Vorteile:

- Implementierung des Client vereinfacht sich:
  - alle Elemente werden gleich behandelt
  - im Beispiel: Aufruf von draw() für alle Grafik-Objekte (= atomare und Gruppierungen)
- Erweiterbarkeit:
  - neue Elemente (Kreise, Dreiecke,...) können einfach eingebunden werden
  - Composite-Klasse kennt nur Component-Objekte

Nachteile:

- die abstrakte Klasse Component (Grafik) bietet auch Container-Methoden (add(), nextChild()) für die atomaren Objekte (Kreis, Linie, Text,...) an