Entwurfsmuster

Bewährte "OOP" Entwürfe für wiederkehrende Entwurfsprobleme, entwickelt von GoF.

Wiederverwendung

Wiederverwendung von Quellcode/Klassen:

- Copy&Paste (schlecht → DRY)
- Vererbung (oft schlecht)
- Aggregation und Komposition (gut → FCoI)

Wiederverwendung von Objekten:

- Thread in **Executor**-Pool
- DB-Connection in Connection-Pool

Wiederverwendung von Komponenten:

- Logging-Komponente
- Java EE-Beans
- Corba-Komponenten, etc.

> Wiederverwendung von Objekten, Quellcode/Klassen oder Komponenten ist ratsam, bringt aber grosse Herausforderungen mit sich. Eine gute Alternative dazu ist die **Wiederverwendung von Entwurfsmuster**

Klassifikation von Entwurfsmustern

- **Erzeugungsmuster**:
  Abstrahieren Erzeugung von Objekten (Typ, Zeitpunkt, Art) → "Fabrik"-Konzept, z.B. Singleton
- **Strukturmuster**:
  Fassen Objekte/Klassen zu grösseren/veränderten Strukturen zusammen (Adapter, Brücke, etc.) → Verbindung unterschiedlicher Strukturen, z.B. Fassade
- **Verhaltensmuster**:
  Beschreiben Interaktion zwischen Objekten (Kontrollflüsse, Zuständigkeit) → z.B. Beobachter, Interpreter, Strategie, etc.

Sekundäre Unterteilung:

- **Klassenmuster**: Beziehungen bereits zur Kompilierzeit fest
- **Objektmuster**: Beziehungen zur Laufzeit veränderbar

Singleton

> Erzeugungsmuster (objektorientiert)

Nur eine Instanz wird erzeugt mit Zugriffspunkt → Client verwendet `public class Singleton {...}`

Eigenschaften:

- statisches Attribut für Objektinstanz
- statische Methode für Zugriff auf Objekt
- privater Konstruktor

→ führt zu starker Kopplung, deswegen **mit Vorsicht einsetzen**

Fassade

> Strukturmuster (objektorientiert)

Stellt eine einheitliche, zusammengefasste Schnittstelle zu einer Menge von Schnittstellen eines Subsystems zur Verfügung → Client verwendet Fassade **Bestellsystem** als Überklasse von **Lager**, **Kunde** und **Bestellung**

Eigenschaften:

- Kennt Subklassen und ihre Funktion
- Delegiert Anfragen an Subklassen
- Subsystemklassen implementieren Funktion (kennen Fassade nicht)

Motivation für Einsatz:

- Vereinfachte Anwendung eines Subsystems
- weniger Abhängigkeiten

→ Gefahr: reiner Durchlauferhitzer, deshalb darauf achten, dass Fassade nur **delegiert**

Strategie

> Verhaltensmuster (objektorientiert)

Definiere Familie von Algorithmen, kapsle diese und mach sie austauschbar. Somit ist es möglich Algorithmus unabhängig von Client zu variieren.

Aufbau:

- **Strategie**,
  stellt Schnittstelle zur Verfügung und wird von Client verwendet
- **Konkrete Strategie(n)**,
  implementieren Algorithmus (evtl. greifen auf Kontext zu)
- **Kontext**,
  besitzt Referenz auf konkrete Strategie und wird mit Strategie konfiguriert (evtl. Datenschnittstelle für Strategien)

Motivation für Einsatz:

- wenn Bedarf nach unterschiedlichem Verhalten viele Bedingungsanweisungen zur Folge hätte

→ wird leicht unterschätzt, kann schon bei sehr kleinen Methoden lohnen. Damit lassen sich grosse und hässliche **switch**-Statements wunderbar eliminieren.

Beobachter

> Verhaltensmuster (objektorientiert)

Definiert Abhängigkeit zwischen einem Subjekt dessen Zustand ändern kann, und einer Menge von Beobachtern die darüber informiert werden sollen.

→ Subjekt benachrichtigt Beobachter, konkreter Beobachter beobachtet konkretes Subjekt

Aufbau:

- **Subjekt**,
  kennt Beobachter (`0..n`) und bietet Schnittstelle zur An-/Abmeldung
- **Beobachter**,
  definiert Benachrichtigungsschnittstelle und empfängt Aktualisierungen/Ereignisse
- **Konkretes Subjekt/Konkreter Beobachter**,
  konkrete Typen

Motivation für Einsatz:

- Typisch für MVC
- wenn nur lose Kopplung der Zuhörer bestehen soll
- wenn Anzahl vorhandener Zuhörer nicht interessiert

→ Bei Java Event-/Listener-Modell verwenden, da besseres Design (Vererbung entfällt)

Adapter

> Strukturmuster (objekt- **oder** klassenorientiert)

Anpassen der Schnittstelle einer Klasse an die vom Client erwartete (Ziel-)Schnittstelle (Wrapper).

Motivation für Einsatz:

- Wiederverwendung von existierenden Klassen, deren Schnittstelle unpassend ist
- um möglichst allgemeine Schnittstelle zu implementieren (Anpassung durch Adapter)

→ verwendet adaptierte Klasse/Objekt und spezialisiert/implementiert Zielschnittstelle