LEHR- UND FORSCHUNGSEINHEIT
PROGRAMMIERUNG &
SOFTWARETECHNIK

Softwaretechnik 2015/2016

PST Lehrstuhl

Prof. Dr. Matthias Hölzl

Joschka Rinke

AGENDA

Übung 8:
03.12.2015

- **Fragen**
- **Besprechung Blatt07**

Aufgabe 1 – Design Patterns

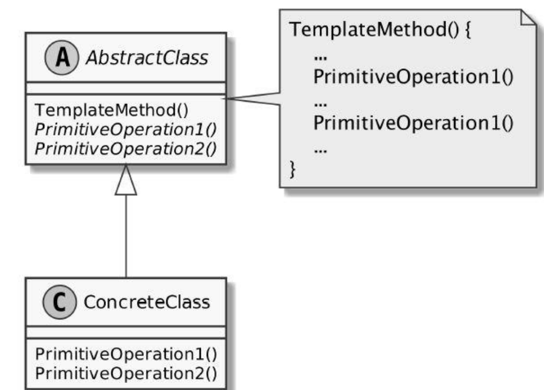
Ein Entwurfsmuster wird beschrieben durch:

- **Name**
- **Beschreibung der Problemklasse**
- **Anwendungsbeispiel**
- **Lösung**
- **Konsequenzen (Nutzen-/Kostenanalyse)**

Aufgabe 1 – Design Patterns

Template Method Pattern

- **Name: Template Method Pattern**
- **Beschreibung:**
  Das Verhaltensmuster der Schablonenmethode definiert das **Skelett eines Algorithmus** in einer Operation und **delegiert einzelne Schritte an Unterklassen**.
  Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.
- **Beispiel:** …
- **Anwendung:**
  Durch die Schablonenmethode werden die invarianten Teile des Algorithmus genau einmal festgelegt.
  Es wird den Unterklassen überlassen, das variierende Verhalten zu implementieren.
  Vermeidung von redundantem Code.
- **Struktur und beteiligte Klassen (Lösung):**
  Abstract Class: legt abstrakte Operationen fest die konkrete Subklassen implemtieren müssen
  Concrete Class: implementiert die abstrakten Operationen der Oberklasse
  Die konkrete Subklasse erbt von der abstrakten Oberklasse
- **Konsequenzen:**
  Template Methods sind eine wichtige Technik für die Wiederverwendung von Code.
  Sie bieten eine wechselseitige Kontrollstruktur.
  Außerdem ist es wichtig, dass klar gekennzeichnet ist, welche Operationen überschrieben werden müssen und welche nicht.



Aufgabe 1 – Design Patterns

Composite Pattern

- **Name: Composite Pattern**
- **Beschreibung:**

  Das Strukturmuster Kompositum wird verwendet um **Teil-Ganzes-Hierarchien** zu repräsentieren.
  Dazu werden Objekte zu **Baumstrukturen** zusammengefasst.
  Die Verwendung des Kompositummusters ermöglicht es sowohl einzelne Objekte,
  als auch ihre Kompositionen einheitlich zu behandeln.
- **Beispiel:** …
- **Anwendung:**

  Implementierung von Teil-Ganzes-Hierarchien.
  Verbergen der Unterschiede zwischen einzelnen und zusammengesetzten Objekten
- **Struktur und beteiligte Klassen (Lösung):**

  Component Class: ist Basisklasse und definiert gemeinsames Verhalten
  Composite Class: enthält Komponenten (Component Class) oder auch Blätter als Kindobjekte
  Leaf Class: repräsentiert ein Einzelobjekt und hat keine Kindobjekte
- **Konsequenzen:**

  Composite Patterns ermöglichen eine einheitliche Behandlung von Objekten und deren Kompositionen
  Es ist sehr leicht erweiterbar um neue Blatt- und Container-Klassen.
  Zu allgemeine Entwürfe erschweren es Kompositionen auf bestimmte Klassen zu beschränken,
  dadurch kann Typsicherheit zur Laufzeit nicht garantiert werden und weitere Typprüfungen werden
  evtl. notwendig.
- **Bekannte Verwendungen, Verwandte Pattern, …**

Aufgabe 1 – Design Patterns

Welche Vorteile ergeben sich durch Verwendung von Design Patterns?

- **Reduzierung der Komplexität**
- **Wiederverwendung erprobter Lösungen**
- **Erhöhte Performance**
- **Aufbrechen von Abhängigkeiten**

Aufgabe 1 – Design Patterns

Welche Nachteile entstehen möglicherweise durch Design Patterns?

- **evtl. erhöhte Komplexität**
- **evtl. kennen Entwickler ein Pattern nicht, das führt zu Unstimmigkeiten**
- **evtl. fehlende Robustheit im Programm**
- **evtl. erhöhte Fehleranfälligkeit**

Aufgabe 1 – Design Patterns

Vorteile

- **Wiederverwendung erprobter Lösungen**
- **Abstrakte Dokumentation von Entwürfen**
- **Entwickler kennen Patterns, verwenden gemeinsames Vokabular**

Nachteile

- **evtl. erhöhte Komplexität**
- **evtl. weniger statische Informationen (Informationen, die gleichbleibend für alle Anwendungsfälle gelten)**
- **evtl. geringere Performance**

Aufgabe 1 – Design Patterns

Klassifizierung in:

- **Creational Patterns:** **Erzeugung von Objekten**
- **Structural Patterns:** **strukturelle Komposition von Klassen oder Objekten**
- **Behavioral Patterns:** **Interaktion von Objekten und Verteilung von Verantwortlichkeiten**

Template Method → Behavioral Pattern

Aufgabe 1 – Design Patterns

Ein weiteres Behavioral Pattern ist:

- **Singleton Pattern**
- **Composite Pattern**
- **Observer Pattern**

Aufgabe 1 – Design Patterns

Beispiel Pattern für die Kategorisierung

- **Creational Patterns: Singleton, Abstract Factory, …**
- **Structural Patterns: Composite, …**
- **Behavioral Patterns: Template Method, Observer, …**

Aufgabe 2 – Singleton Pattern

Singelton Design Pattern erster Ansatz:

- **Konstruktor *protected* oder *private***

  Nachteil *protected* → im gleichen Package sichtbar

  Nachteil *private* → keine Subklassenerzeugung mehr möglich

  → falls *private* gewollt ist, dann Klasse als final deklarieren

- **getInstance()-Methode zum Erzeugen einer Instanz**

```
…
if(instance == null)
    instance = new Singleton();
return instance;
```

Probleme:

- **Je nach Sichtbarkeit können evtl. doch mehrere Instanzen erzeugt werden.**
- **Thread safety? → Synchronize getInstance() → teuer ☹**
- **Testen kann sehr schnell aufwändig werden…**

→ gesamte Code kommt auf die Vorlesungsseite

Aufgabe 2 – Singleton Pattern

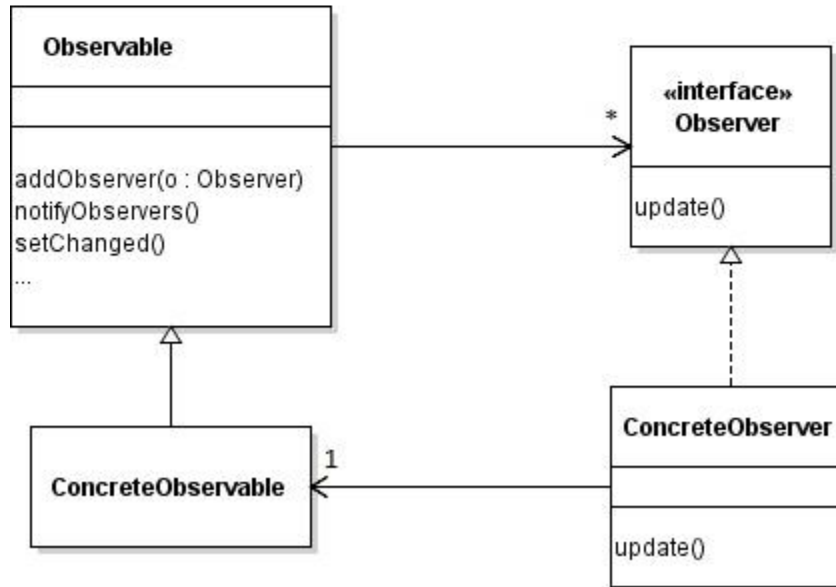
Definition Enum Types:

- **Enums sind spezielle Datentypen → Aufzählungstypen**
- **Enums werden umgesetzt als eine bestimmte Art normaler Klassen**
  → **Sie haben alle Eigenschaften, die normale Klassen haben**
- **Enums haben außerdem Konstanten und zusätzliche Methoden**
- **Enums werden bei der ersten Verwendung einmalig instantiiert**
  → **z.B. EnumClass.INSTANCE**
- **Enums garantieren einmalige Instantiierung**

→ **einelementige Enums sind gut geeignet um Singletons zu implementieren**

LEHR- UND FORSCHUNGSEINHEIT
PROGRAMMIERUNG &
SOFTWARETECHNIK

Aufgabe 3 – Observer Pattern



Was muss also konkret wie implementiert werden?

- **Kino muss Observable erweitern**
- **Kino muss Observer implementieren**
- **Frame muss Observable erweitern**
- **Frame muss Observer implementieren**

Aufgabe 3 – Observer Pattern

Observable

Observer
«interface»

Kino

...

+ incrementIndex()
+ addEmployee(emp : String)
+ getIndex() : int
+ getShifts() : Shift

Frame

...

+ initFrame()
+ getFirstPanel() : FirstPanel
+ getSecondPanel() : SecondPanel
+ changeView(p : JPanel)
+ update()
...

JFrame

FirstPanel

...

+ setTextArea(text : String)
+ setLabel(text : String)

JPanel

SecondPanel

...

+ setTextArea(text : String)
+ setLabel(text : String)

Aufgabe 3 – Observer Pattern

Kino erweitert Observable, damit können Observer registriert werden und selbige über Änderungen informiert werden.

Die Klasse Kino erbt also insbesondere folgende Methoden von Observable:

- **addObserver(Obsserver o) → Fügt einen Observer zum Observer-Set des Objekts hinzu**
- **notifyObservers() → Benachrichtigt alle Observer falls sich das Objekt geändert hat**
- **setChanged() → Markiert dieses Objekt als geändert**

Aufgabe 3 – Observer Pattern

Was sollte hier noch hinzugefügt werden?

```
public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable(){
                @Override
                public void run(){
                        Kino kino = new Kino();
                        Frame frame = new KinoFrame((KinoModel)kino);
                        kino.addObserver(frame);
                        frame.initGUI();
                }
        });
}
```

Aufgabe 3 – Observer Pattern

Kino:

```
public class Kino extends Observable {
    …
    incrementIndex(){
        …
        setChanged();
        notifyObservers();
    }
}
```

Frame:

```
public class Frame implements Observer {
    ….
    update(Observable o, Object arg){
        …
    }
}
```