HTBL Imst

AUFBAULEHRGANG FÜR INFORMATIK

Übungszettel – Mikroarchitektur

Name: Michael Bogensberger

Datum: 15.12.2021

1 Inhaltsverzeichnis

2 BlueJ — Klassenentwurf

Beim Klassenentwurf geht es darum, wie man gut wartbare und wiederverwendbare Klassen entwirft. Hier gibt es die Grundprinzipien Kopplung und Kohäsion.

2.1 Kopplung

Die Kopplung beschreibt den Grad der Abhängigkeit zwischen Klassen. Es wird beim Klassenentwurf eine möglichst geringe oder lose Kopplung angestrebt. Das heißt im Prinzip das eine Klasse möglichst unabhängig sein soll und über möglichst schmale Schnittstellen kommunizieren soll.

2.2 Kohäsion

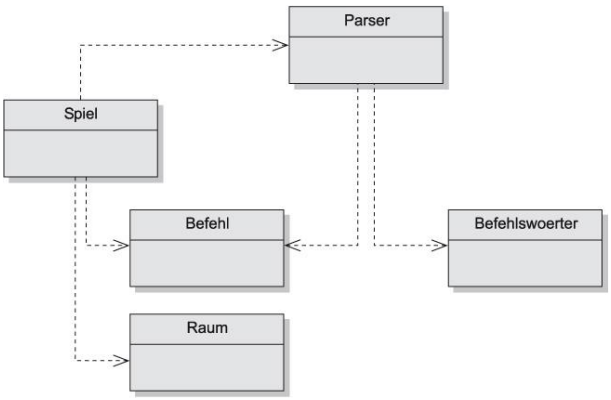
Kohäsion beschreibt, wie gut ein Programmteil eine logische Aufgabe abbildet. In einem Programm mit hoher Kohäsion ist genau jeder Programmteil bzw. Programmeinheit gut genau eine gut definierte Aufgabe verantwortlich. Eine Methode soll genau eine gut definierte Logik haben. Einen gut definierten Zweck sozusagen.

2.3 Welt von Zuul

Fügen Sie dem Spiel 'Welt von Zuul' zwei neue Bewegungsrichtungen hinzu:

- up
- down

Was müssen Sie hierzu ändern?



- Die Klasse `Befehlswörter` definiert die gültigen Befehle für das Spiel. Sie hält dazu ein Array von Zeichenketten mit den Befehlswörtern.
- Die Klasse `Parser` liest Eingabezeilen von der Konsole und versucht sie als Befehl zu interpretieren. Er erzeugt Objekte der Klasse `Befehl`, die die eingegebenen Befehle repräsentieren.
- Ein Objekt der Klasse `Befehl` repräsentiert einen Befehl, den der Benutzer eingegeben hat. Es bietet Methoden zum Test der Gültigkeit eines Befehls und zur Ausgabe der einzelnen Befehlswörter.
- Ein Objekt der Klasse `Raum` repräsentiert einen Ort im Spielgeschehen, die Ausgänge zu anderen Räumen haben können.
- Die Klasse Spiel ist die Hauptklasse des Spiels. Sie initialisiert das Spiel und liest dann die Befehle in einer Schleife ein und führt diese aus.

Abbildung 1 Welt von Zuul

Man müsste in den Klassen Spiel und Raum Änderungen vornehmen. Die neuen Richtungen müssten dann in der Klasse Raum bzw. bei den Ausgängen angepasst werden. Haben wir dass gemacht, müssen wir noch in der Klasse Spiel die Optionen für die neuen Richtungen anpassen.

3 BlueJ — Entwurf von Anwendungen

Es gibt vier zentrale Konzepte wenn es um den Entwurf von Anwendungen geht.

- Identifizieren von Klassen
- CRC-Karten
- Entwurf von Schnittstellen
- Entwurfsmuster

3.1 CRC-Karten

Die CRC-Karte (Class-Responsibility-Collaboration-Karte) dient als Hilfsmittel für objektorientiertes Entwerfen. CRC-Karten sind eine Möglichkeit, objektorientierte Programme in einem ersten Entwurf zu entwickeln. Sie dienen also als Hilfsmittel für objektorientiertes entwickeln. Es geht darum die Klasse, Verantwortlichkeiten und Kollaborationen darzustellen. Im oberen Bereich wird der Name der Klasse eingetragen. Im linken Bereich stehen die Zuständigkeiten der Klasse: "Was kann sie mit wem tun". Im rechten Bereich stehen mögliche Kollaborationspartner für Objekte dieser Art.

Abbildung 2 CRC-Karte - Aufbau

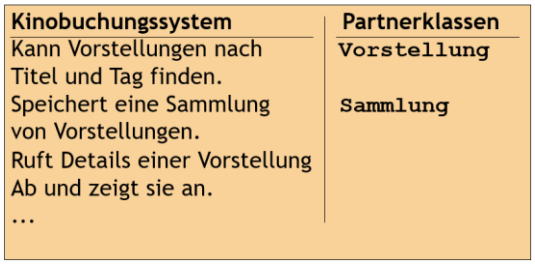


Abbildung 3 CRC-Karte - Beispiel Kino

4 Entwurfsmuster 1

Entwurfsmuster (englisch Design Patterns) sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme in der Softwarearchitektur und -entwicklung. Entwurfsmuster sind also nichts anderes als Vorlagen zur Problemlösung.

In Java wird mittels Design Pattern das Zusammenspiel von Klassen, Interfaces, Objekten und Methoden mit dem Ziel beschrieben, vordefinierte Lösungen für konkrete Programmierprobleme anzubieten.

4.1 Beispiel - Singleton Design Pattern

Ein Beispiel für ein Design Pattern wäre das Singleton Design Pattern. Beim Singleton Pattern handelt es sich in Java um genau eine Klasse. Diese darf nur ein einziges Mal istanziert werden. Während der Programmlaufzeit existiert nur ein einziges Objekt.

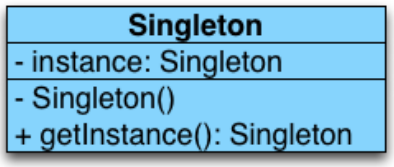


Abbildung 5 Singleton Design Pattern - Klassendiagramm

```
public class Singleton
{
  private static Singelton instance = null;

  private Singleton() {}

  public static Singleton getInstance()
  {
    if (instance == null)
    {
      instance = new Singleton();
    }

    return instance;
  }
}
```

Abbildung 4 Singleton Design Pattern - Code Beispiel

Die Aufgabe vom Singleton Design Pattern ist es also zu verhindern, dass von einer Klasse mehr als ein Objekt erstellt wird. Das wird dadurch erreicht, dass das gewünschte Objekt in einer Klasse selbst erzeugt wird und dann als statische Instanz abgerufen wird.

5 Entwurfsmuster 2

Nun ist gefragt, sich in die folgenden Design Patterns einzuarbeiten. In diesem Abschnitt ist also eine kurze Dokumentation zu den jeweiligen Design Patterns zu finden. Jene sind:

- Chain of Responsibility (Zuständigkeitskette)
- Template Hook (Template method pattern, Schablonenmethode)
- Decorator
- Builder
- Adapter
- Observer

5.1 Zuständigkeitskette (Chain of Responsibility)

Die Zuständigkeitskette gehört zu den Verhaltensmustern und wird für Algorithmen verwendet. Die Zuständigkeitskette dient als Entkopplung von Sender und Empfänger. Sie dient also dazu die Kopplung von Objekten zu verringern. Das wird dadurch erreicht, dass sie versucht eine Aufgabe in einer Kette von Objekten zu bearbeiten.

5.1.1 Beispiel Handler

Ich als Client rufe meine Handler auf zb Handler_1 und übergebe mittels HandlerRequest irgendein Objekt das bearbeitet werden soll. Wenn der Handler diesen Request bearbeiten kann, dann bearbeitet er ihn auch. Kann er ihn nicht bearbeiten dann gibt er es an zb dem Handler_2 weiter.

Jeder Handler enthält keine oder eine Instanz eines weiteren Handlers.

Die Chain of Responsibility wird verwendet wenn ich zb ein Objekt habe das bearbeitet werden muss, ich aber nicht genau weiß wer dafür zuständig ist.

Ein Hilfreiches Video ist hier zu finden: [Chain of Responsibility Video]

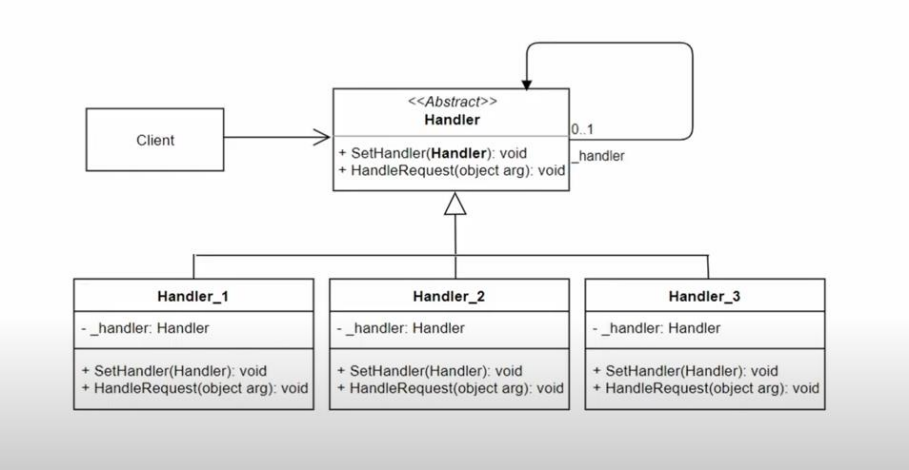


Abbildung 6 Chain of Responsibility - Beispiel Handler

Um das Konzept der Chain of Responsibility noch einmal etwas anders zu veranschaulichen finden Sie hier ein Sequenz Diagramm zur Chain of Responibility.

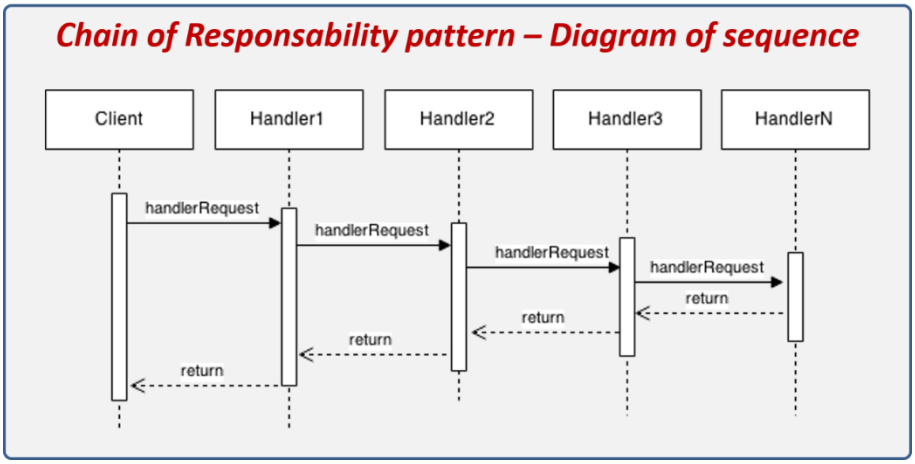


Abbildung 7 Chain of Responsibility - Sequenz Diagramm

5.2 Schablonenmethode (Template Hook)

Bei der Schablonenmethode (Template Hook) ist der grundsätzliche Ablauf eines Algorithmus bereits fest vorgegeben. Jedoch können sich einzelne Schritte bzw. Details ändern. Die Grundsätzliche Struktur bleibt jedoch unverändert.

Das Template Hook kommt bei zb folgenden Anwendungen zum Einsatz:

- Sortieralgorithmen
  Grundsätzlich ist der Ablauf immer gleich. Wenn ich zb nach Kundennummer oder Geburtsdatum sortieren möchte, ändern sich hier jedoch ein paar Details.
- Daten über Netzwerk versenden
  Grundsätzlich ist der Ablauf hier immer:
  1) Verbindung aufbauen
  2) Daten senden
  3) Verbindung schließen

  Wenn ich zb eine http Verbindung habe und eine SMTP Verbindung habe sind die Schritte bzw. Reihenfolge wie Aufbau, senden und abbauen immer gleich. Das aufbauen ist zb unterschiedlich.

Hier wieder ein Video zum Template Hook zu finden: Template Hook Video
Zudem ist hier ein hilfreiches Guide zu finden: TH-Guide

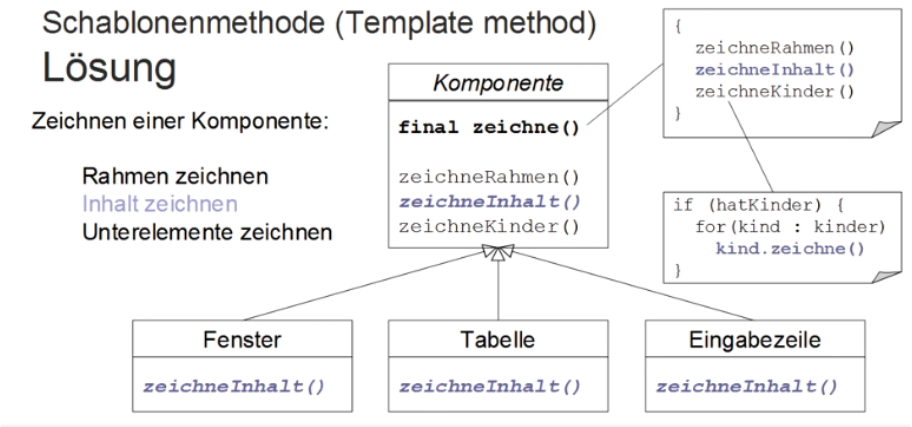


Abbildung 8 Template Hook - Beispiel zeichnen

5.3 Decorator Pattern

Das Decorator Pattern ist ein Entwurfsmuster zur flexiblen bzw. alternativen Unterklassenbildung, um eine Klasse um zusätzliche Funktionalität zu erweitern. Es dient also im Prinzip dazu, Klassen unkompliziert zu erweitern.

Angenommen wir haben hier zb. drei GUI-Fenster. Wir wollen jedoch, dass in einem davon ein Bild gezeichnet wird. Wie gehen wir nun vor. Zunächst gibt es die Klasse Decorator. Diese erbt von der Component Klasse. Dadurch haben wir beide Funktionen foo() und bar() zur Verfügung. Wir geben nun im Konstruktor von Decorator eine Instanz also ein Component Objekt mit. Der Decorator hält also eine Component. Der Decorator ruft im Prinzip nur foo() oder bar() vom Component auf. Hier hat sich also noch nichts verändert. Nun erstellen wir eine Unterklasse vom Decorator. Diese nennen wir nun EchterDecorator. Hier drinnen überschreiben wir nun die foo() Methode. Hier können wir also neuen Code hinzufügen. Wir können hier ja zb zuerst comp.foo() aufrufen und danach zusätzliche Funktionalität einbauen.

Ein Tutorial zum Decorator Pattern: Decorator Pattern Video
Ein Guide zum Decorator Pattern: Decorator Pattern Guide

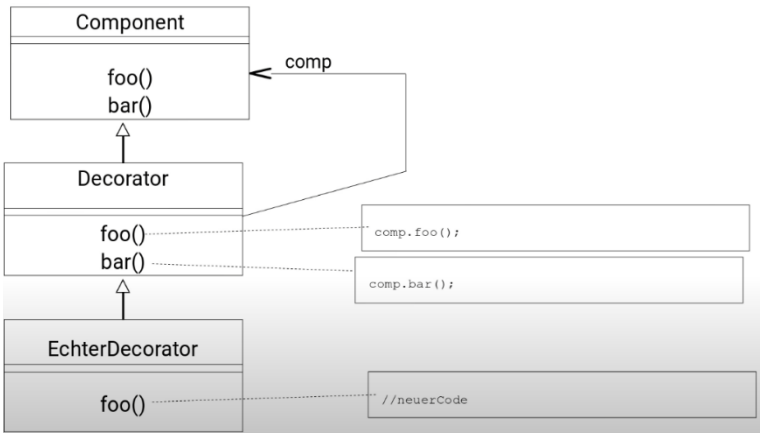


Abbildung 9 Decorator Pattern Beispiel

5.4 Builder Pattern

Das Builder Pattern verbessert sowohl die Sicherheit bei der Konstruktion als auch die Lesbarkeit des Programmcodes. Das Ziel des Erbauer-Entwurfsmusters ist, dass ein Objekt nicht mit den bekannten Konstruktoren erstellt wird, sondern mittels einer Hilfsklasse.

Im Builder Design Pattern werden vier Akteure unterschieden:

- **Direktor**: Dieser Akteur konstruiert das komplexe Objekt, indem er die Schnittstelle des Erbauers verwendet. Er kennt die Anforderungen an die Arbeitsreihenfolge des Erbauers. Beim Direktor wird die Konstruktion eines Objekts vom Kunden („Auftraggeber“) entkoppelt.
- **Erbauer**: Hält eine Schnittstelle zur Erzeugung der Bestandteile eines komplexen Objekts (bzw. Produkts) bereit.
- **Konkreter Erbauer**: Dieser Akteur erzeugt die Teile des komplexen Objekts und definiert (und verwaltet) die Repräsentation des Objekts und hält eine Schnittstelle zur Ausgabe des Objekts vor.
- **Produkt**: Das Ergebnis der „Tätigkeit“ des Builder Patterns, also das zu konstruierende komplexe Objekt.

Beim Direktor geschieht der entscheidende Vorgang des Erbauer-Musters, die Trennung der Herstellung eines Objekts/Produkts vom Auftraggeber.

Link zum Builder Pattern: [Builder Pattern Guide]

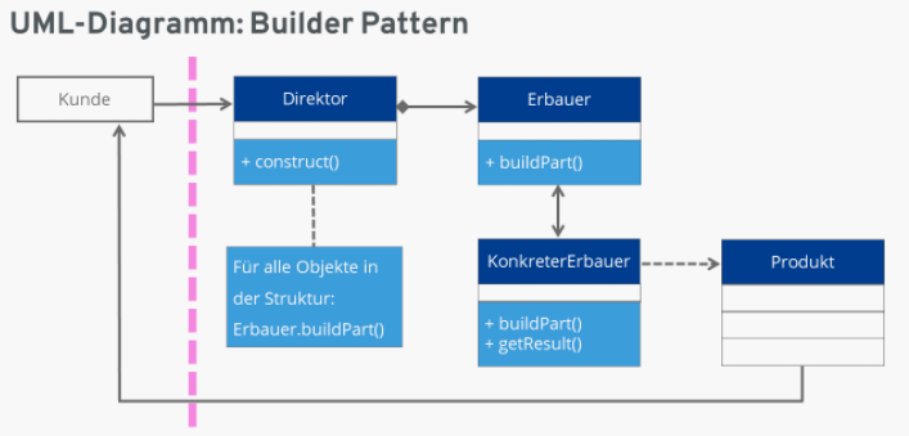


Abbildung 10 Builder Pattern Beispiel

5.5 Adapter Pattern

Das Adapter Pattern ist da, um zwei nicht kompatible Klassen miteinander kompatibel zu machen. Dies erreicht man indem man die beiden einfach zur einen Adapter verknüpft.

Der Client verwendet also das Target Interface. Er ruft also irgendwann die Methode foo() auf. Wo diese jedoch nun definiert ist, ist für den Client unwichtig. Der Adapter erbt nun vom Target Interface. Der Adapter implementiert nun natürlich die Methode foo(). Der Client will nun aber baz() aufrufen. Der Adapter hält sich nun eine Instanz der Klasse "Adaptierter". In der Methode foo() steht nun adaptee.baz(). Dadurch ist es nun möglich die baz() Methode aufzurufen.

Tutorial zum Adapter Pattern: [Adapter Pattern Video]

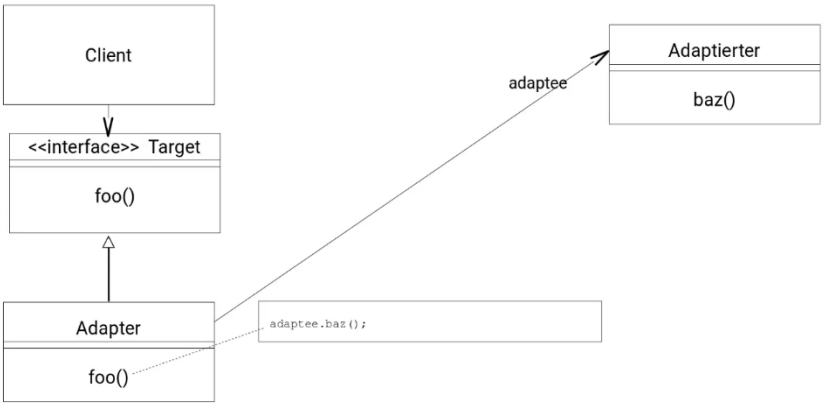


Abbildung 11 Adapter Pattern Beispiel

5.6 Observer Pattern

Das Observer Pattern arbeitet mit zwei Typen von Akteuren:
Es gibt das zu beobachtende Objekt. Dem gegenüber gibt es das beobachtende Objekt. Dieses soll über Änderungen in Kenntnis gesetzt werden.

Ein Beispiel: Es gibt einen Pinguin und es gibt Forscher die den Pinguin beobachten. Wenn die Forschen nun also immer auf dem Laufenden bleiben wollen, müssen sie immer wieder nach dem Pinguin schauen. Das Prinzip des Observer Patterns ist es, dass der Pinguin von sich aus Bescheid gibt wenn sich was ändert.

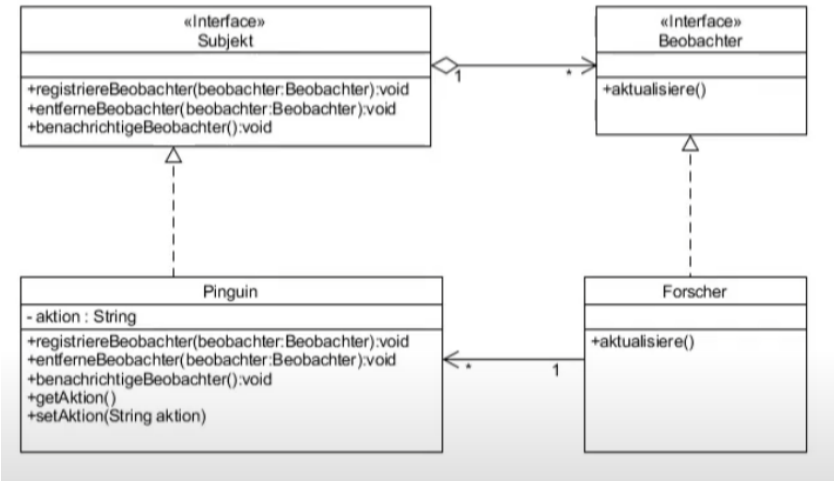


Abbildung 12 Observer Pattern Beispiel

Beim Observer Pattern gibt es zwei Grundsätze:

- Push: Der Beobachter holt sich die Information, sobald er eine Nachricht erhält, dass sich etwas geändert hat.
- Pull: Die Änderung wird direkt mitgeteilt.

Video zum Observer Pattern: [Observer Pattern Video]

6 Währungsrechner (Chain of Responsibility)

Erstellen Sie eine Verantwortlichkeitskette mit mindestens zwei Währungsrechnern (z.B. EUR2YEN, EURO2Dollar). Neben dem Standardverhalten einer Chain of Responsibility soll es möglich sein, neue Währungsrechner in die Kette aufzunehmen bzw. bestehende aus der Kette zu löschen (jeweils am Ende der Kette).
Link zum Code: GitHub Projekt
Das zur Aufgabe passende Klassendiagramm:

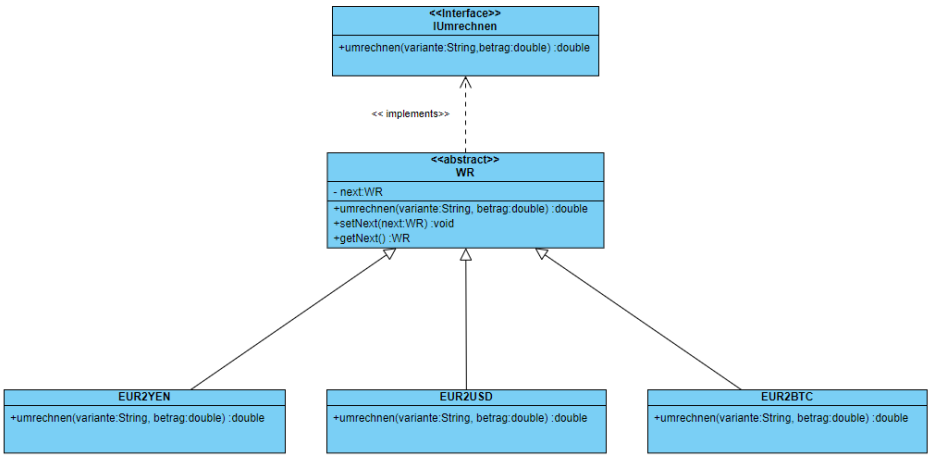


Abbildung 13 Währungsrechner - COR - Klassendiagramm

Das zur Aufgabe passende Sequenzdiagramm:

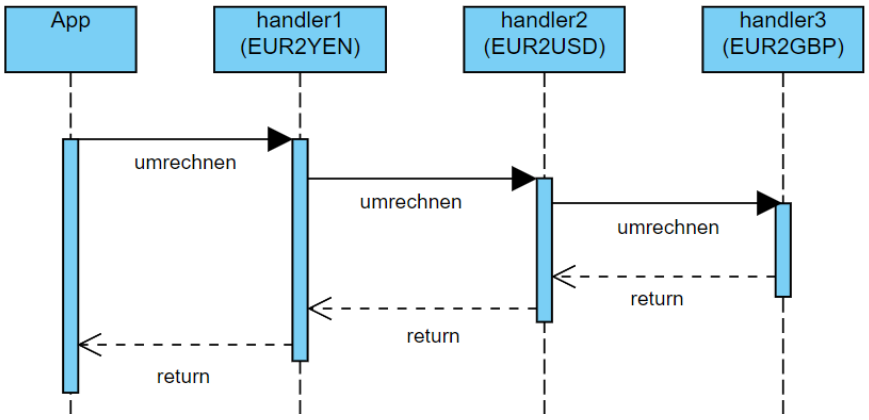


Abbildung 14 Währungsrechner - COR - Sequenzdiagramm

7 Währungsrechner (Template Hook)

Vermeiden Sie Codeduplikate in der Methode umrechnen() in den resultierenden Unterklassen. Platzieren Sie den Code dazu soweit wie möglich und sinnvoll in der abstrakten Klasse WR und delegieren Sie ausschließlich das eigentliche umrechnen per Template-Hook-Muster an die Unterklassen.

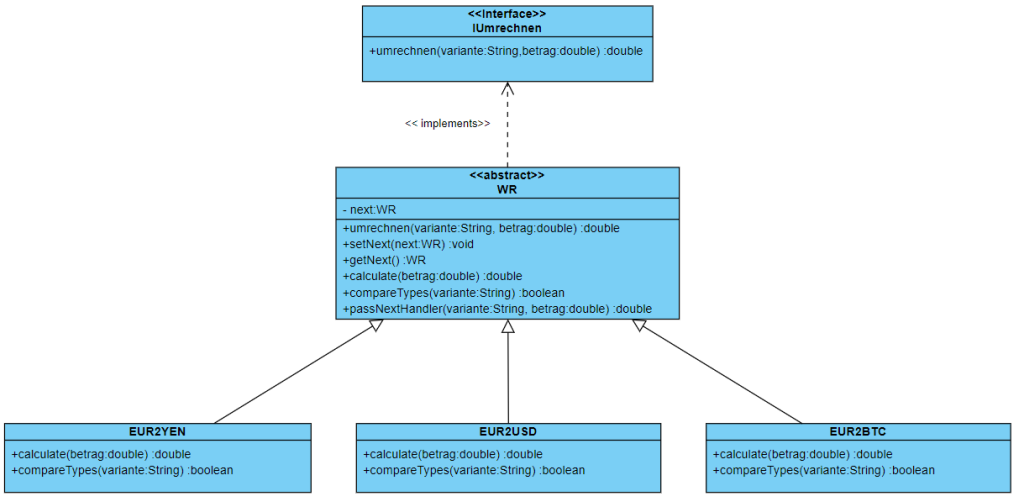


Abbildung 15 Währungsrechner - Template Hook - Klassendiagramm

8 Währungsrechner (Decorator)

Dekorieren Sie ihre Währungsrechner-Kette mit folgenden Dekoratoren:

a) Belegung eines Umrechnungsvorganges mit Gebühren (z.B. 0,5 % des Umrechnungsbetrages)
b) Belegung eines Umrechnungsvorganges für Umrechnungen von Euro nach Währung X (nicht in die andere Richtung) mit fixen Gebühren von 5 Euro.

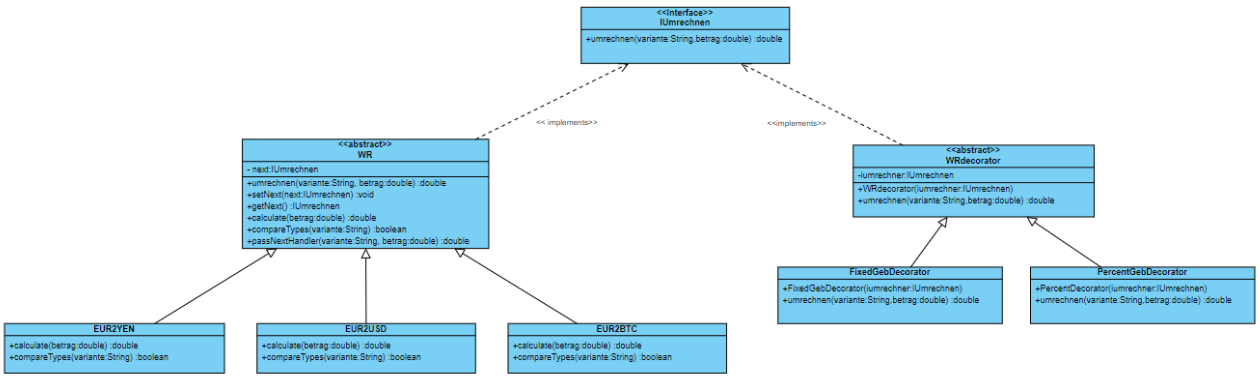


Abbildung 16 Währungsrechner - Decorator - Klassendiagramm

9 Währungsrechner (Builder)

Implementieren Sie das Builder-Pattern für eine Objektinstanziierung einer auf Wunsch auch dekorierten Währungsrechner-Kette.

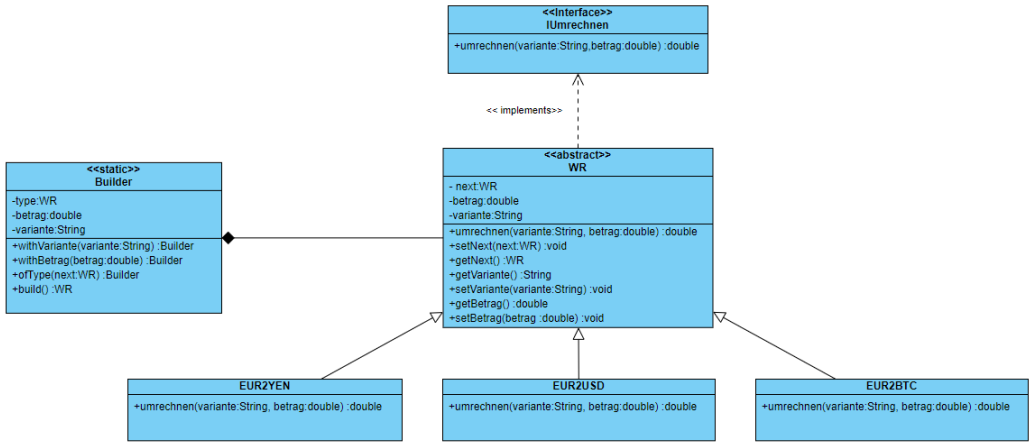


Abbildung 17 Währungsrechner - Builder - Klassendiagramm

10 Währungsrechner (Adapter)

Externe Anwendungen benötigen eine Implementierung der Schnittstelle ISammelumrechnung (siehe Angabe), um Sammelumrechnungen durchführen zu können. Stellen Sie einen Adapter bereit, der Sammelumrechnungen in der geforderten Form (siehe Methodensignatur) zur Verfügung stellt und dazu die Funktionalität der (dekorierten) Währungsrechner-Kette verwendet.

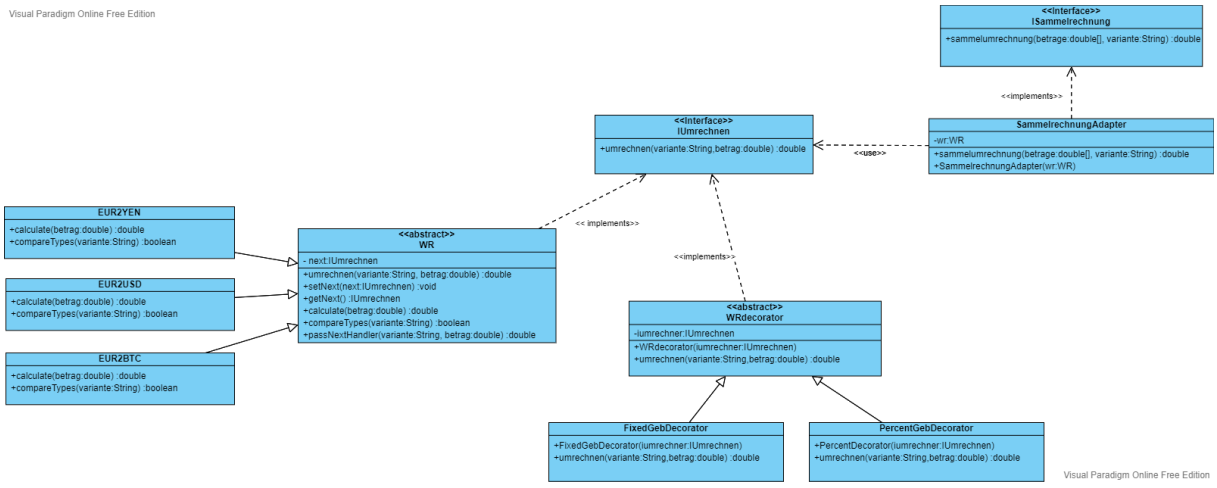


Abbildung 18 Währungsrechner - Adapter - Klassendiagramm

11 Währungsrechner (Observer)

Am Währungsrechner sollen sich mehrere Observer registrieren können. Jedes Mal wenn eine Umrechnung stattfindet sollen alle Observer benachrichtigt werden. Alle Informationen der Umrechnungen (Ausgangsbetrag, Ausgangswährung, Zielwährung, Zielbetrag) sollen mit der Benachrichtigung versendet werden. Beispielhaft sollen zwei Observer implementiert werden:

1) Atom-Feed-Observer: Erzeugt einen Atom-Feed mit allen Umrechnungsinformationen und Zeitstempel (verwende dazu: https://mvnrepository.com/artifact/rome/rome)
2) Log-Observer: Erzeugt eine Log-Text-Datei mit allen Umrechnungsinformationen und Zeitstempel.

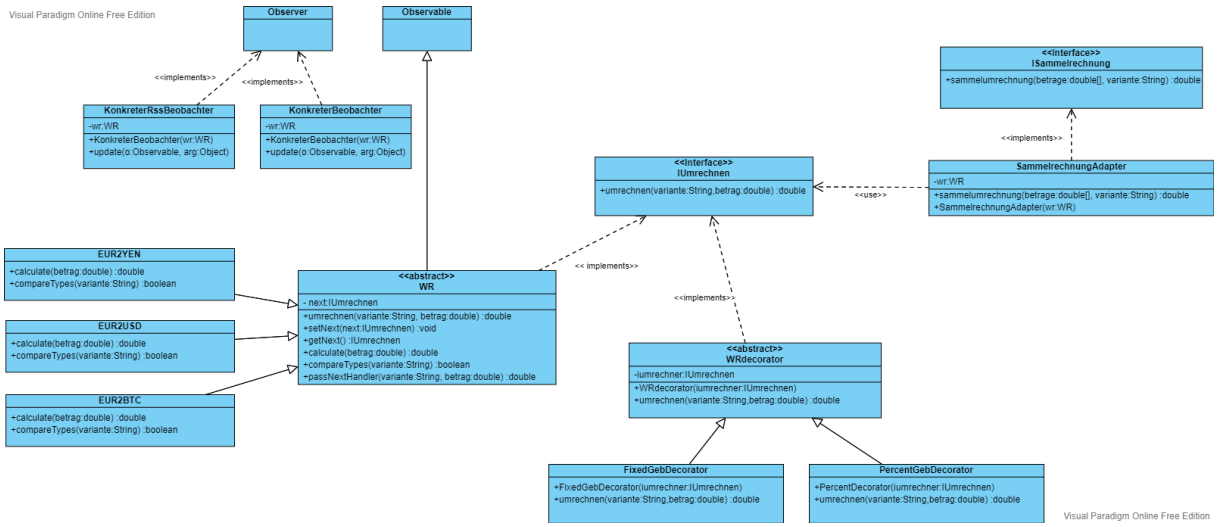


Abbildung 19 Währungsrechner - Observer - Klassendiagramm

12 Projektcode (GitHub)

Alle Implementierungen zu den Design Patterns sind unter folgendem Link im GitHub zu finden:
Design Pattern Repository

13 SOLID Prinzip

Beim SOLID-Prinzip handelt es sich um Designvorgaben für Softwareentwicklung. Ziel ist es, dass Software leicht modifizierbar und leicht nachvollziehbar gestaltet wird. Ein ganz gut verständlicher Guide zum SOLID Prinzip ist hier zu finden: SOLID-Guide.

- SRP: Single-Responsibility-Prinzip
  Besagt, dass jede Klasse nur eine einzige Verantwortung haben soll.

- OCP: Open-Closed-Prinzip
  Besagt, dass Software leicht erweiterbar sein soll, ohne dabei das Verhalten jener Software zu ändern. Ein Beispiel dafür wäre die Vererbung. Sie verändert die Oberklasse nicht, fügt jedoch Funktionalität hinzu.

- LSP: Liskov'sche Substitutions-Prinzip
  Prinzipiell ist hiermit gemeint, dass bei der Vererbung die Unterklasse jederzeit alle Eigenschaften der Oberklasse enthalten muss, so dass diese von der Oberklasse verwendet werden können. Die Unterklasse darf keine Änderungen der Funktionalitäten der Oberklassen enthalten. Die Oberklasse darf jedoch durch neue Funktionalitäten erweitert werden.

- ISP: Interface-Segregation-Prinzip
  Dient dazu, dass User nicht dazu gezwungen werden, Teile von Schnittstellen zu implementieren, die sie nicht benötigen. Dies soll dazu führen, dass Schnittstellen nicht zu groß werden und das sie nicht auf einen speziellen Nutzen schrumpfen.

- DIP: Dependency-Inversion-Prinzip
  Das Dependency-Inversion-Prinzip besagt, dass Klassen auf einem höheren Abstraktionslevel nicht von Klassen auf einem niedrigen Abstraktionslevel abhängig sein sollen. Ein guter Guide zum DIP-Prinzip: DIP-Guide

14 Weitere Prinzipien

Unter diesem Abschnitt sind weitere Designprinzipien zu finden.

14.1 KISS Prinzip

Steht für Keep It Simple, Stupid. Es geht darum möglichst einfache Lösungen zu finden.

14.2 YAGNI Prinzip

Steht für You Aint Gonna Need It. Mit der gleichen Intensität, mit der man hinterfragt, ob man die einfachste Lösung gewählt hat, sollte man auch fragen, ob man ein gegebenes Softwarefeature überhaupt benötigt.

14.3 DRY Prinzip

Steht für Dont repeat yourself. Im Prinzip geht es darum, dass Redundanz vermieden werden soll.

15 Abbildungsverzeichnis