ÜBUNGSZETTEL MIKROARCHITEKTUR

# BLUEJ-BUCH KLASSENENTWURF

Arbeite das Buchkapitel „Klassenentwurf“ (K8). Die Folien dazu sind online. Erledige alle Übungen aus dem Kapitel. Dokumentiere deine Erledigungen.

Übung 8.1:

* Was tut diese Anwendung?  
  lässt Spieler Gegend erkunden
* Welche Befehle akzeptiert das Spiel?  
  quit, help, go [east, west, north, south]
* Was bewirken die einzelnen Befehle?
  + quit: beendet das Spiel
  + help: zeigt Hilfsmenü
  + go [east, west, north, south]: bewegt Spieler in einen anderen Raum
* Wie viele Räume gibt es in der virtuellen Umgebung?
  + 5
* Zeichnen Sie eine Karte der gegebenen Räume

Haupteingang

Haupteingang

Cafe

Rechnerraum

Verwaltungs-Büro

Übung 8.2:

Spiel:

* Hauptklasse
* Erzeugt und initialisiert alle anderen Objekte (Räume, Parser)
* startet das Spiel
* wertet Befehle aus, die der Parser liefert
* sorgt für Ausführung

Raum:

* modelliert Räume
* Verbindung zu benachbarten Räumen

Parser:

* liest Benutzereingaben und wandelt sie in Befehle um
* liest Konsole ein
* liefert Befehl als Objekt zurück
* vergleicht Eingabe mit hinterlegten Befehlen

Befehl:

* Objekte halten Infos über eingegebene Befehle
* wandelt Zeichenkette in 2 Befehle um: go east
* Befehle werden überprüft
* bei unbekanntem Befehlswort: retun null

Befehlswörter:

* hält Aufzählung gültiger Befehlswörter
* erkennt gültige Eingaben

Rest in C:\Users\tobia\Desktop\ITKolleg\3\_4AKIF\POS1\LAND\BlueJ\_Projekte\ProjekteBlueJ\Kapitel08

Theorie ist klar

# BLUEJ-BUCH ENTWURF VON ANWENDUNGEN

Erledige das BlueJ-Buch-Kapitel „Entwurf von Anwendungen“ (K15). Die Folien dazu sind online. Erledige alle Übungen aus dem Kapitel. Dokumentiere deine Erledigungen.

Entwurf von Anwendungen

**15.1 Analyse und Entwurf**

**15.1.1 Die Verb/Substantiv-Methode**

Klassen und Objekte für einen Problembereich sowie ihre Verknüpfungen und Interaktionen identifizieren.

Substantiv = Klassen und Objekte

Verb = Methoden

**15.2 Klassenentwurf**

Klasse und Methoden identifizieren

**15.2.1 Entwurf von Klassenschnittstellen**

Methoden: private, protected, public?

**15.2.2 Entwurf von Benutzungsschnittstellen**

Interaktion mit Benutzer. Sollte unabhängig von Funktionalität sein. MVC (Model, View, Control)

**15.3 Dokumentation**

Vor Implementierung -> Schnittstellen dokumentieren

Klassen- und Methodenkommentare (Zweck beschreiben)

**15.4 Kooperation**

wohlstrukturierter objektorientierter Ansatz liefert gute Unterstützung für pair programming

ermöglicht Zerlegung der Aufgaben in lose gekoppelte Komponenten (Klassen)

Interaktion der Klassen über Schnittstellen

**15.5 Prototyping**

ermöglicht Testung von einzelnen Komponenten des Systems

Teile werden simuliert, um mit anderen Teilen zu experimentieren

z.B.: Suche liefert immer gleiches Ergebnis. Suche noch nicht implementiert, aber mit Ergebnis kann getestet werden

**15.6 Softwarewachstum**

**15.6.1 Wasserfallmodell**

Phasen:

* Analyse des Problems
* Entwurf der Software
* Implementierung der Softwarekomponenten
* Modultests
* Integrationstest
* Auslieferung des Systems an den Kunden

Wenn eine Phase fehlschlägt, muss möglicherweise eine Stufe zurückgegangen werden -> ist aber eigentlich nicht vorgesehen.

Etabliertestes Vorgehensmodell, hat aber viele Nachteile. Wichtigste Schwächen: Annahme, dass Entwickler die Funktionalität des zu erstellenden Systems von Anfang an komplett kennen und dass sich ein System nach der Auslieferung nicht mehr ändert.

**15.6.2 Iterative Vorgehensmodelle**

Einsatz von Prototypen und Einbeziehung des Kunden in den Entwicklungsprozess

Prototyp mit einfacher Funktionalität entwickeln -> Kunde vorstellen -> Verbesserungen und zusätzliche Funktionalität einbauen -> Kunde vorstellen

Anderer Ansatz: Software wird nicht entworfen -> sondern wächst

am Anfang kleines, sauber entworfenes System, das der Endbenutzer einsetzen kann -> nach und nach Funktionalität hinzufügen

**15.7 Der Einsatz von Entwurfsmustern**

**15.7.1 Struktur eines Musters**

üblicherweise in einheitlicher Form, das ein Minimum an Informationen garantiert. Enthält nicht nur Informationen über die Struktur einiger Klassen, sondern auch Probleme, die dieses Muster angeht, sowie konkurrierende Einflüsse, die für oder gegen den Einsatz des Musters sprechen.

Mindestbeschreibung des Musters:

* Name
* Beschreibung des Problems, das das Muster adressiert (Zweck, Motivation, Anwendbarkeit)
* Beschreibung der Lösung (Struktur, Teilnehmer, Partner)
* Erläuterung der Konsequenzen, mit Ergebnissen und Vor- und Nachteilen

**15.7.2 Dekorierer**

behandelt das Hinzufügen von Funktionalität zu einem bereits existierenden Objekt

Annahme: Objekt muss auf gleiche Methodenaufrufe reagieren wie Standardobjekt, aber zusätzliches oder geändertes Verhalten zeigen -> evtl. Erweiterung der Schnittstelle

Eine Möglichkeit: Vererbung -> Unterklasse überschreibt Methode oder weitere Methoden werden hinzugefügt -> statisch ☹

dynamische Lösung -> Dekorierer-Objekt: ist ein Objekt, das ein anderes Objekt umschließt und anstelle des umschlossenen Objekts verwendet werden kann (implementiert üblicherweise die gleiche Schnittstelle). Benutzer kommuniziert mit Dekorierer statt mit Original (ohne von dieser Ersetzung wissen zu müssen). Dekorierer-Objekt leitet Aufrufe an das umschlossene Objekt weiter, kann aber auch zusätzliche Aktionen durchführen.  
Beispiel: BufferedReader ist ein Dekorierer für Reader -> implementiert gleiche Schnittstelle und kann anstelle eines ungepufferten Readers verwendet werden, fügt aber zusätzliches Verhalten hinzu. Im Unterschied zur Vererbung kann ein Dekorierer auch für bereits existierende Objekte verwendet werden.

**15.7.3 Singleton**

Wenn von Klasse nur eine einzige Instanz erstellt werden soll -> Singleton garantiert dies und ermöglicht einheitlichen Zugriff auf diese Instanz.

Deklaration mit privatem Konstruktor -> kann nicht von außerhalb der Klasse aufgerufen werden

```java

public class Parser

{

private static Parser instanz = new Parser();

private static Parser gibInstanz(){

return instanz;

}

java```

In dieser Umsetzung des Musters:

* Konstruktor ist privat, kann also nur innerhalb der Klasse aufgerufen werden -> muss im statischen Teil der Klasse passieren, da sonst keine Instanzen existieren können.
* wird ein privates, statisches Datenfeld definiert und initialisiert, das eine einzige Instanz des Parsers enthält
* wird eine statische Methode gibInstanz() definiert, die den Zugriff auf die eine Instanz ermöglicht

**15.7.4 Fabrikmethode**

liefert Schnittstelle für die Erzeugung von Objekten, überlässt Suklassen die Entscheidung darüber, von welchen konkreten Klassen Instanzen erzeugt werden. Typischerweise erwartet ein Klient eine Superklasse oder ein Interface vom dynamischen Typ des tatsächlichen Objekts, während die Fabrikmethode Spezialisierungen liefert.

Beispiel: Collection

Collection hat iterator

```java

public void verarbeite(Colletion<Typ> c){

Iterator<Typ> it = c.iterator();

}

java```

Aus Sicht des Klienten gehen wir mit Objekten vom Typ Collection und Iterator um. Tatsächlich ist der (dynamische) Typ der Sammlung jedoch möglicherweise ArrayList, wodurch der Aufruf von iterator ein Objekt vom Typ ArrayListIterator zurückliefert. Oder HashSet und HashSetIterator …

Fabrikmethode wird in Subklasse spezialisiert, damit von dort spezialisierte Instanzen des „offiziellen“ Ergebnistyps geliefert werden.

**15.7.5 Beobachter**

Beobachter-Muster bietet eine Möglichkeit der Trennung zwischen Modell und Ansicht zu erreichen.

Beobachter-Muster definiert eine 1-zu-n-Beziehung, sodass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden -> lose Kopplung zwischen Beobachtern und beobachtetem Objekt.

Für das Beobachter-Muster werden 2 Typen benutzt: Observable (beobachtbar) und Observer (Beobachter). Die beobachtete Einheit erweitert die Klasse Observable und der Beobachter implementiert das Interface Observer.

Die Klasse Observer bietet Methoden an, mit denen sich Beobachter bei der beobachtbaren Einheit anmelden können. Sie garantiert, dass jedes Mal die Methode update() an jedem Beobachter aufgerufen wird, wenn die beobachtete Einheit die geerbte Methode notify() aufruft.

Kann auch angewendet werden, wenn der Zustand eines oder mehrerer Objekte von Zustand eines anderen Objekts abhängt.

# ENTWURFSMUSTER 1

Arbeite dich in das Thema Entwurfsmuster ein. Du sollst wissen, was Entwurfsmuster sind, wie sie beschrieben werden und wozu man sie benötigt.

Du bekommst dazu auch eine Einführung im Unterricht anhand des Strategy-Entwurfsmusters.

<https://de.wikipedia.org/wiki/Entwurfsmuster>

**Entwurfsmuster (design patterns)**

sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme sowohl in der Softwarearchitektur und -entwicklung. Stellen eine wiederverwendbare Vorlage zur Problemlösung dar, die in einem bestimmten Zusammenhang einsetzbar ist.

**Arten von Entwurfsmustern**

* Erzeugungsmuster (Creational Patterns)  
  Dienen der Erzeugung von Objekten. Entkoppeln die Konstruktoren eines Objekts von seiner Repräsentation. Objekterzeugung wird gekapselt und ausgelagert, um den Kontext der Objekterzeugung unabhängig von der konkreten Implementierung halten.   
  „Programmiere auf die Schnittstelle, nicht auf die Implementierung!“
* Strukturmuster (Structural Patterns)  
  Erleichtern den Entwurf von Software durch vorgefertigte Schablonen für Beziehungen zwischen Klassen.
* Verhaltensmuster (Behavioral Patterns)  
  Modellieren komplexes Verhalten der Software und erhöhen damit die Flexibilität der Software hinsichtlich ihres Verhaltens
* Muster für objektrelationale Abbildung  
  Dienen der Ablage und dem Zugriff von Objekten und deren Beziehungen in einer relationalen Datenbank.
* Nachrichtenübermittlungsmuster (Messaging Patterns)  
  Für die Erstellung, die Weiterleitung und die Umwandlung von Daten in Form von Nachrichten, inklusive der dazu benötigten Nachrichtenkanäle und Nachrichtenendpunkte, sowie Komponenten für die Systemverwaltung

**Anforderungen und Nutzen:**

Ein gutes Muster sollte:

* ein oder mehrere Probleme lösen
* ein erprobtes Konzept bieten
* auf realen Designs basieren
* über das rein Offensichtliche hinausgehen
* den Benutzer in den Entwurfsprozess einbinden
* Beziehungen aufzeigen, die tiefergehende Strukturen und Mechanismen eines Systems umfassen

Entwurfsmuster beinhalten in der Regel Referenzen auf andere Muster. Mithilfe dieser ist es möglich, Mustersprachen zu entwickeln.

Der primäre Nutzen eines Entwurfsmuster liegt in der Beschreibung einer Lösung für eine bestimmte Klasse von Entwurfsproblemen. Weiterer Nutzen ergibt sich aus der Tatsache, dass jedes Muster einen Namen hat. Dies vereinfacht die Diskussion unter Entwicklern, da man abstrakt über eine Struktur sprechen kann. So sind etwa Software-Entwurfsmuster zunächst einmal unabhängig von der konkreten Programmiersprache.

**Nachteile**

Der erfolgreiche Einsatz von Entwurfsmustern in der Vergangenheit kann dazu verleiten, die Entwurfsmuster als Wunderwaffe und Garant für gutes Design anzusehen. Unerfahrene Entwickler können geneigt sein, möglichst viele bekannte Muster zu verwenden, und dabei übersehen, dass in ihrem Fall vielleicht eine elegantere Lösung ohne den Einsatz von Mustern möglich wäre. Entwurfsmuster garantieren nicht, dass der Entwurf gut ist.

**Beispiel für Muster:**

1. Erzeugungsmuster (creational patterns)
   * **Erbauer (Builder)**
   * Fabrikmethode (Factory method)
   * Multiton
   * Prototyp
2. Strukturmuster (structural design patterns)
   * Container
   * **Dekoriere (Decorator)**
   * Fassade (Facade)
   * Fliegengewicht (Flyweight)
   * Kompositum (Composite oder Whole-Part)
   * Stellvertreter (Proxy)
3. Verhaltensmuster (behavioral design patterns)
   * Interceptor
   * Interpreter
   * Kommando (Command)
   * Schablonenmethode (Template method)
   * **Strategie (Strategy)**
   * Vermittler (Mediator)
   * Zustand (State)
   * Zuständigkeitskette (Chain of responsibility)
4. Muster für objektrelationale Abbildung
   * Datentransferobjekt (Data transfer object)
   * Active Record
   * Unit of Work
   * Lazy Loading
   * Command-Query-Responsibility-Segregation
5. Nachrichtenübermittlungsmuster (Messaging Patterns)
   * Message
   * Document Message
   * Splitter
   * Aggregator
   * Message Store
   * Smart Proxy
6. Weitere Muster
   * Extension Interface
   * Fluent Interface
   * Inversion of Control
   * Locks
   * Repository
   * Value Object
   * Registry

<https://de.wikipedia.org/wiki/Strategie_(Entwurfsmuster)>

**Strategy-Entwurfsmuster:**

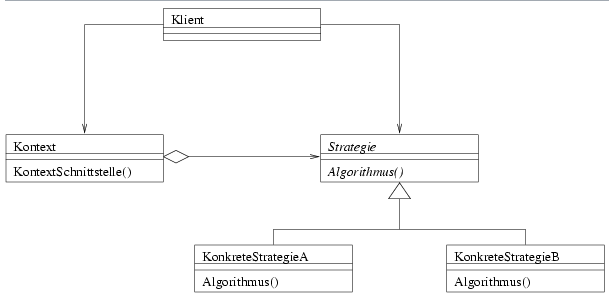
gehört zur Kategorie der Verhaltensmuster (behavioral design patterns). Strategie definiert eine Familie austauschbarer Algorithmen.

Verwendung:

ähnlich wie Klassenbibliothek. Handelt sich jedoch nicht um externe Programmteile, sondern um integrale Bestandteile des eigentlichen Programms, die deshalb als eigene Objekte definiert wurden, damit sie durch andere Algorithmen ausgetauscht werden können.

Verwendung von Strategien bietet sich an, wenn

* viele verwandte Klassen sich nur in ihrem Verhalten unterscheiden
* unterschiedliche (austauschbare) Varianten eines Algorithmus benötigt werden
* Daten innerhalb eines Algorithmus von Klienten verborgen werden sollen
* verschiedene Verhaltensweisen innerhalb eines Klasse fest integriert sind (meist über Mehrfachverzweigungen), aber
  + die verwendeten Algorithmen wiederverwendet werden soll bzw.
  + die Klasse flexibler gestaltet werden soll



Die Klasse Strategie definiert nur eine Schnittstelle (Interface) für alle unterstützten Algorithmen. Die Implementierung der eigentlichen Algorithmen finden sich erst in den Ableitungen wieder (konkrete Strategie).

Der Kontext hält eine Variable der Schnittstelle Strategie, die mit einer Referenz auf das gewünschte Strategieobjekt belegt ist. Auf diese Weise wird der konkrete Algorithmus über die Schnittstelle eingebunden und kann bei Bedarf selbst zur Laufzeit noch dynamisch gegen eine andere Implementierung ausgetauscht werden.

Vorteile:

* Es wird eine Familie von Algorithmen definiert
* Es wird die Auswahl aus verschiedenen Implementierungen ermöglicht und dadurch erhöhen sich die Flexibilität und die Wiederverwendbarkeit
* Es können Mehrfachverzweigungen vermieden werden und dies erhöht die Übersicht der Codes
* Strategien bieten eine Alternative zur Unterklassenbildung der Kontexte

Nachteile:

* Klienten müssen die unterschiedlichen Strategien kennen, um zwischen ihnen auswählen und den Kontext initialisieren zu können
* Gegenüber der Implementierung der Algorithmen im Kontext entsteht hier ein zusätzlicher Kommunikationsaufwand zwischen Strategie und Kontext

Beispiel:

* Steuerberechnungsprogramm, das die Berechnung von Steuersätzen möglichst in Strategie-Objekte auslagern sollte, um einfach länderabhängig konfigurierbar zu sein
* Speicherung eines Dokuments oder einer Grafik in verschiedenen Dateiformaten

# ENTWURFSMUSTER 2

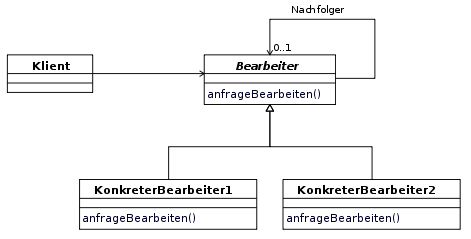
Arbeite dich in folgende Entwurfsmuster ein:

**Chain of Responsibility (Zuständigkeitskette)**

gehört zur Kategorie der Verhaltensmuster und wird für Algorithmen verwendet. Dient der Entkopplung des Auslösers einer Anfrage mit seinem Empfänger.

Verwendung:

Mehrere Objekte werden hintereinander geschaltet (miteinander verkettet), um gemeinsam eine eigehende Anfrage bearbeiten zu können. Diese Anfrage wird an der Kette entlang geleitet, bis eines der Objekte die Anfrage beantworten kann. Der Klient, von dem die Anfrage ausgeht, hat dabei keine Kenntnis darüber, von welchem Objekt die Anfrage beantwortet wird.



Akteure:

* Bearbeiter, der ein Interface für die Anfrage definiert
* konkreter Bearbeiter, der alle Anfragen bearbeitet, für die er selbst zuständig ist und alle anderen Anfragen an das nächste Kettenglied (den nächsten Bearbeiter) weiterleitet
* Klient, der die Anfrage an irgendeinem konkreten Bearbeiter initiiert

Vor und Nachteile:

Vorteil: Klient muss den zuständigen Bearbeiter nicht kennen. Selbst die Kettenglieder müssen nur ihren direkten Nachfolger und nicht den Gesamt-Aufbau der Kette kennen -> geringe Kopplung. Außerdem kann die Zuständigkeit von Objekten für bestimmte Anfragen verändert werden, ohne dass potenzielle Klienten davon in Kenntnis gesetzt werden müssen.

Nachteil: Es gibt keine Garantie, dass die Anfrage tatsächlich bearbeitet wird. Wenn das letzte Glied der Kette eine Anfrage erhält, für die es ebenfalls nicht zuständig ist, wird die Anfrage nach obigem Muster verworfen. Dies muss durch eine entsprechende Fehlerbehandlung abgefangen werden.

Es muss sichergestellt werden, dass jeder Bearbeiter in der Kette nur einmal vorkommt, sonst entstehen Kreise und das Programm bleibt in einer Endlosschleife hängen.

[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\01\_Chain\_of\_responsibility](01_Chain_of_responsibility)

**TemplateHook (Schablonenmethode)**

gehört zur Kategorie der Verhaltensmuster. Teilschritte eines Algorithmus können variabel gehalten werden.

Funktionsweise:

Skelett eines Algorithmus wird in einer abstrakten Klasse definiert. Konkrete Ausformung der einzelnen Schritte wird an Unterklassen delegiert. Dadurch besteht die Möglichkeit, einzelne Schritte des Algorithmus zu verändern oder zu überschreiben, ohne dass die zu Grunde liegende Struktur des Algorithmus modifiziert werden muss.

Zusätzlich können in der Schablonenmethode an bestimmten Stellen Hook-Operationen aufgerufen werden, deren Standardimplementierung in der abstrakten Klasse nichts tut. Auf diese Weise kann man an vordefinierten Stellen im Algorithmus zusätzliche Funktionalität einfügen.

Beispiel: Ein OutputStream implementiert eine konkrete Methode zum Schreiben eines Byte-Arrays. Diese Methode benutzt eine Methode zum Schreiben eines einzelnen Bytes, um das ganze Array nach und nach zu schreiben. Die Methode für das einzelne Byte ist jedoch noch abstrakt, da ein OutputStream selbst noch nicht spezifisch ist. Klassen wie FileOutputStream können diese Methode implementieren. Sie erben dann eine bereits implementierte Methode zum Schreiben eines Byte-Arrays.

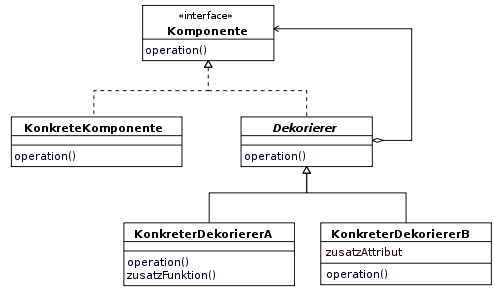
[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\02\_Template](02_Template)

**Decorator (Dekorierer)**

gehört zur Kategorie der Strukturmuster. Das Muster ist eine flexible Alternative zur Unterklassenbildung, um eine Klasse um zusätzliche Funktionalitäten zu erweitern.

Verwendung:

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.



Akteure:

* abstrakte Komponente: definiert öffentliche Schnittstelle für die zu dekorierenden Objekte
* konkrete Komponente: definiert Objekte, die dekoriert werden können
* abstrakter Dekorierer: hält Referenz auf abstrakte Komponente und bietet dieselbe Schnittstelle wie die abstrakte Komponente
* konkreter Dekorierer: definiert und implementiert eine oder mehrere spezielle Dekorationen

Vor- und Nachteile:

Vorteile: mehrere Dekorierer können hintereinander geschaltet werden. Dekorierer können zur Laufzeit und sogar nach der Instanziierung ausgetauscht werden. Die zu dekorierende Klasse ist nicht unbedingt festgelegt (wohl aber deren Schnittstelle). Zudem können lange und unübersichtliche Vererbungshierarchien 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 ausgegeben werden, obwohl dieselbe Komponente gemeint ist. Zudem müssen bei der Verwendung von dekorierten Komponenten die Nachrichten vom Dekorierer an das dekorierte Objekt weitergeleitet werden.

Beispiele:

Stream-Klassen in Java-Bibliothek: Man dekoriert das konkrete Stream-Objekt mit Objekten, die neue Eigenschaften zu dem Stream hinzufügen und sich weiter dekorieren lassen. Dabei können Decorator-Objekte verwendet werden, die neue Statusinformationen hinzufügen oder solche, die neue Schnittstellen zur Verfügung stellen.

[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\03\_Decorator](file:///C:\Users\tobia\Desktop\ITKolleg\5_6AKIF\FSE\LAND\FSE_LAND\06_Mikroarchitektur\03_Decorator)

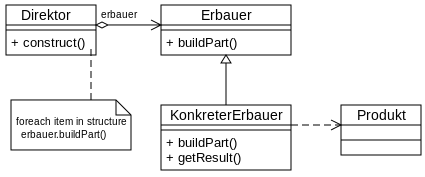
**Builder**

gehört zur Kategorie der Erzeugungsmuster. trennt die Konstruktion komplexer Objekte von deren Repräsentationen, wodurch dieselben Konstruktionsprozesse verwendet werden können.

Verwendung:

Einsatz bietet sich an, wenn:

* zu einem komplexen Objekt unterschiedliche Repräsentationen existieren sollen,
* die Konstruktion eines komplexen Objekts unabhängig von der Erzeugung der Bestandteile sein soll oder
* der Konstruktionsablauf einen internen Zustand erfordert, der vor einem Klienten verborgen werden soll



Akteure:

* Erbauer: spezifiziert eine abstrakte Schnittstelle zur Erzeugung der Teile des komplexen Objektes. erzeugt Teile des komplexen Objekts durch Implementierung der Schnittstelle. definiert und verwaltet die von ihm erzeugte Repräsentation. bietet Schnittstelle zum Auslesen des Produkts
* Direktor: konstruiert ein komplexes Objekt unter Verwendung der Schnittstelle des Erbauers. arbeitet eng mit Erbauer zusammen: Er weiß, welche Baureihenfolge der Erbauer verträgt oder benötigt. entkoppelt somit den Konstruktionsablauf vom Klienten.
* Produkt: repräsentiert das zu konstruierende komplexe Objekt

Vor- und Nachteile:

Vorteile: Implementierung der Konstruktion und der Repräsentation werden isoliert. Die Erbauer verstecken ihre interne Repräsentation vor dem Direktor. Neue Repräsentationen lassen sich leicht durch neue konkrete Erbauerklassen einfügen. Der Konstruktionsprozess wird an einer dedizierten Stelle (im Direktor) gesteuert; spätere Änderungen - etwa ein Mehrphasen-Konstruktionsprozess statt einer Einphasen-Konstruktion - lassen sich ohne Änderung der Klienten realisieren.

Nachteile: Es besteht eine enge Kopplung zwischen Produkt, konkretem Erbauer und den am Konstruktionsprozess beteiligten Klassen.

[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\04\_Builder](04_Builder)

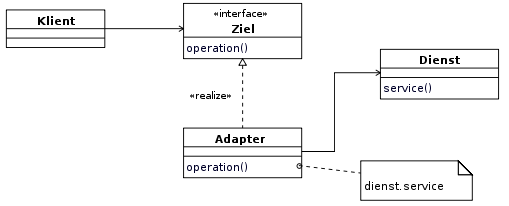
**Adapter (Hüllenklasse oder Wrapper)**

gehört zur Kategorie der Strukturmuster. dient zur Übersetzung einer Schnittstelle in eine andere. Dadurch wird die Kommunikation von Klassen mit zueinander inkompatiblen Schnittstellen ermöglicht.

Verwendung:

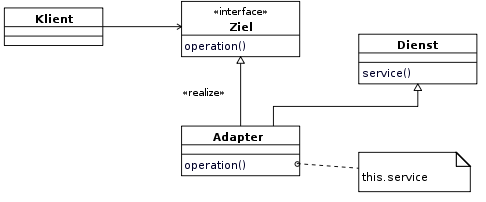
wenn eine existierende Klasse verwendet werden soll, deren Schnittstelle nicht der benötigten Schnittstelle entspricht (oft bei Klassen, die zur Wiederverwendung konzipiert wurden). Werkzeugsammlungen oder Klassenbibliotheken stellen ihre Dienste durch klar definierte Schnittstellen zur Verfügung, die in der Regel nicht geändert werden sollen oder können. Adapter wird auch bei der Erstellung wiederverwendbarer Klassen benutzt, wenn diese mit unabhängigen oder nichtvorhersehbaren Klassen zusammenarbeiten sollen.

Adapter mit Delegation (Objektadapter)



Adapter hat eine Assoziation zu der zu adaptierenden Klasse und leitet die Anfragen per Delegation weiter. Vorteil: Adapter und der dahinterliegende Dienst können ausgetauscht werden. Dafür muss aber die gesamte Schnittstelle implementiert werden, auch wenn nur ein Teil der Implementierung angepasst werden soll.

Adapter mit Vererbung (Klassenadapter)



wird mit Hilfe von Mehrfachvererbung realisiert. Adapter erbt die Implementierung der zu adaptierenden Klasse und zusätzlich die zu implementierende Schnittstelle. Aufruf über Selbstdelegation.

Akteure:

* Dienst bietet wiederzuverwendende Dienstleistungen mit fest definierter Schnittstelle an.
* Klient nutzt Dienste über inkompatible Schnittstellen und greift dabei auf adaptierte Schnittstellen zurück.
* Ziel definiert die Schnittstelle, die der Klient benutzen kann.
* Adapter adaptiert Schnittstelle des Dienstes auf die Schnittstelle zum Klienten.

Vor- und Nachteile:

Vorteil: kann sich genau einer Zielklasse anpassen und kann dadurch nur das Verhalten der Zielklasse überschreiben. Objektadapter kann auch Unterklassen mit anpassen.

Nachteil: Adapter adaptiert die Schnittstelle des Dienstes auf die Schnittstelle zum Klienten.

[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\05\_Adapter](05_Adapter)

**Observer (Beobachter)**

gehört zur Kategorie der Verhaltensmuster. dient der Weitergabe von Änderungen an einem Objekt an von diesem Objekt abhängige Strukturen.

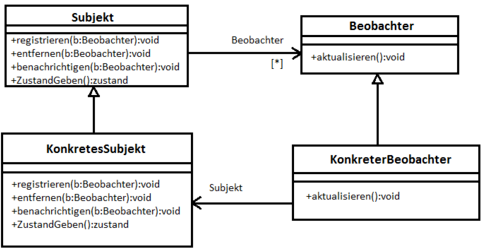
Verwendung:

Allgemeine Anwendungssituationen: wenn eine Abstraktion mehrere Aspekte hat, die von einem anderen Aspekt derselben Abstraktion abhängen, wo Änderung eines Objekts Änderungen an anderen Objekten nach sich zieht oder ein Objekt andere Objekte benachrichtigen soll, ohne diese im Detail zu kennen.

Anwendungsbeispiel: Eine oder auch mehrere Komponenten stellen den Zustand eines Objekts grafisch dar. Sie kennen die gesamte Schnittstelle dieses Objektes. Ändert sich der Zustand des Objektes, müssen die darstellenden Komponenten darüber informiert werden. Andererseits soll das Objekt aber von den Komponenten unabhängig bleiben, ihre Schnittstelle also nicht kennen. Beispiel: Messerergebnisse werden gleichzeitig in einem Balkendiagramm, einem Liniendiagramm und einer Tabelle dargestellt. Messwerte ändern sich ständig. Komponenten der Diagramme sollen diese Änderungen permanent darstellen, das gemessene Objekt soll dabei aber keine Kenntnis von der Struktur dieser Komponente besitzen.

Lösung: Das beobachtete Objekt bietet einen Mechanismus, um Beobachter an- und abzumelden und diese über Änderungen zu informieren. Es kennt alle seine Beobachter nur über die (überschaubare) Schnittstelle Beobachter. Änderungen werden völlig unspezifisch zwischen dem beobachteten Objekt und jedem angemeldeten Beobachter ausgetauscht. Dieses braucht also die weitere Struktur dieser Komponenten nicht zu kennen. Die Beobachter implementieren ihrerseits eine (spezifische) Methode, um auf die Änderung zu reagieren. Es gibt drei Arten:

* Push Notification: Jedes Mal, wenn sich das beobachtete Objekt ändert, werden alle Beobachter benachrichtigt. Es werden jedoch keine Daten mitgeschickt, weshalb diese Form immer die gleiche Beobachter-Schnittstelle hat. Die Beobachter müssen nach Eintreffen der Nachricht Daten abholen.
* Push-Update Notification: Jedes Mal, wenn sich das beobachtete Objekt ändert, werden alle Beobachter benachrichtigt. Zusätzlich leitet das beobachtete Objekt die Update-Daten, die die Änderungen beschreiben, an die Beobachter weiter.
* Pull Notification: Der Beobachter fragt ständig nach dem Zustand des beobachteten Objekts nach.



Akteure:

* Subjekt (beobachtbares Objekt) hat eine Liste von Beobachtern, ohne deren konkrete Typen zu kennen. Es bietet eine Schnittstelle zur An- und Abmeldung von Beobachtern und eine Schnittstelle zur Benachrichtigung von Beobachtern über Änderungen an.
* konkretes Subjekt: speichert den relevanten Zustand und benachrichtigt alle Beobachter bei Zustandsänderungen über deren Aktualisierungsschnittstelle unter Verwendung der Abfrageschnittstelle. Es verfügt über eine Schnittstelle zur Erfragung des aktuellen Zustands.
* Beobachter: definieren eine Aktualisierungsschnittstelle.
* konkrete Beobachter: verwalten die Referenz auf ein konkretes Subjekt, dessen Zustand sie beobachten und speichern und dessen Zustand konsistent ist. Sie implementieren eine Aktualisierungsschnittstelle unter Verwendung der Abfrageschnittstelle des konkreten Subjekts.

Vor- und Nachteile:

Vorteile: Subjekte und Beobachter können unabhängig variiert werden. Subjekt und Beobachter sind auf abstrakte und minimale Art lose gekoppelt. Das beobachtete Objekt braucht keine Kenntnis über die Struktur seiner Beobachter zu besitzen, sondern kennt diese nur über die Beobachter-Schnittstelle. Ein abhängiges Objekt erhält Änderungen automatisch. Es werden auch Multicasts unterstützt.

Nachteile: Änderungen am Subjekt führen bei großer Beobachteranzahl zu hohen Änderungskosten. Außerdem informiert das Subjekt jeden Beobachter, auch wenn dieser die Änderungsinformation nicht benötigt. Änderungen können weitere Änderungen nach sich ziehen. Mechanismus liefert keine Information darüber, was sich geändert hat. Wenn Beobachter während der Bearbeitung einer gemeldeten Änderung wiederum Änderungsmethoden des Subjekts aufruft, kann es zu Endlosschleifen kommen.

[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\06\_Observer](06_Observer)

Einarbeiten heißt:

* Muster lernen
* Muster anhand einer minimalen Aufgabenstellung selbst implementieren
* Alles dokumentieren.

# AUSGANGSPUNKT FÜR DIE FOLGENDEN AUFGABEN

Die im Laufe der folgenden Aufgaben zu erstellende Software bildet einen einfachen Währungsrechner nach, der unter Anwendung verschiedener Entwurfsmuster implementiert werden muss. Die abstrakte Klasse WR sowie die Interfaces IUmrechnen und ISammelumrechnung sind als Ausgangspunkt für die Arbeit sind im Anhang gegeben.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

**Hinweis**: Wenn du die folgenden Entwurfsmuster anhand einer zur Währungsrechner-Thematik alternativen Domäne implementieren möchtest, dann ist das natürlich in Absprache mit der LV-Leitung möglich.

# 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).

[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\07\_Waehrungsrechner\01\_Chain\_of\_responsibility](07_Waehrungsrechner/01_Chain_of_responsibility)

# 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.

[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\07\_Waehrungsrechner\02\_Template](07_Waehrungsrechner/02_Template)

# DECORATOR

Dekorieren Sie ihre Währungsrechner-Kette mit folgenden Dekoratoren:

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

[C:\Users\tobia\Desktop\ITKolleg\5\_6AKIF\FSE\LAND\FSE\_LAND\06\_Mikroarchitektur\07\_Waehrungsrechner\03\_Decorator](07_Waehrungsrechner/03_Decorator)

# BUILDER

Implementieren Sie das Builder-Pattern (<https://dzone.com/articles/design-patterns-the-builder-pattern>) für eine Objektinstanziierung einer auf Wunsch auch dekorierten Währungsrechner-Kette.

**PASST IT!**

# 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.

# 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

# SOLID UND SONSTIGE DESIGNPRINZIPIEN

Lerne im Detail die die SOLID-Prinzipien:

<https://dev.to/ham8821/solid-principles-to-start-with-object-oriented-programming-1e49>

<http://openbook.rheinwerk-verlag.de/oo/oo_03_prinzipien_000.htm#Xxx999144>

<https://en.wikipedia.org/wiki/SOLID>

<https://www.youtube.com/watch?v=TMuno5RZNeE>

<https://www.youtube.com/playlist?list=PL4CE9F710017EA77A>

Lerne mindestens 3 **weitere Prinzipien** des objektorientierten Entwurfs kennen (DRY, YAGNI, KISS etc. siehe Moodle-Kurs <https://moodle.tsn.at/course/view.php?id=24763#section-10>).

Wenn du alles im Detail gelernt hast, **weise an verschiedenen Stellen deiner Implementierungen** aus den vorhergehenden Aufgaben **die Realisierung oder die Verletzung** der SOLID-Prinzipien bzw. der übrigen Designprinzipien (DRY, YAGNI, KISS etc. siehe Moodle-Kurs) nach.

**SOLID**

* Single-responsibility principle: Jede Klasse sollte nur eine Zuständigkeit haben.
* Open-closed principle: Entitäten sollten offen für Erweiterung, aber geschlossen für Veränderungen sein.
* Liskov substitution principle: Vererbte Klassen sollen pro Methode nur das machen was erwartet wird (class Tier{ gibLaut()}, class Hund { gibLaut() return „Wuff“})
* Interface segregation principle: viele kleine Interfaces sind besser als ein großes
* Dependency inversion principle: Abhängigkeit von Abstraktion, nicht von konkreter Implementierung.

**KISS**

Keep it simple (, | and) stupid

Für ein Problem soll eine möglichst einfache Lösung gefunden werden.

ähnelt der Aussage von Ockhams Rasiermesser: Wenn es mehrere Erklärungen für einen bestimmten Sachverhalt gibt, dann ist diejenige Erklärung zu bevorzugen, die am einfachsten ist, also mit den wenigsten Annahmen und Variablen auskommt.

**DRY**

Don’t repeat yourself

Redundanz soll vermieden oder zumindest reduziert werden.

Keine Code-Duplikate -> Änderungen müssen nur an einer Stelle vorgenommen werden

**YAGNI**

You Aren’t Gonna Need It

Bezeichnet das Prinzip des Extreme Programming, das besagt, dass in einem Programm erst dann Funktionalität implementiert werden sollte, wenn klar ist, dass diese Funktionalität tatsächlich gebraucht wird.

Programme werden oft durch zusätzlichen oder allgemeinen Code auf mögliche künftige Änderungsanforderungen vorbereitet -> Überlegung: Änderungen sind später aufwändiger umzusetzen. Stattdessen ergeben sich häufig Anforderungen, die bei der ursprünglichen Entwicklung nicht vorhergesehen wurden und deren Umsetzung durch den Code, der eigentlich Änderungen erleichtern sollte, nicht unterstützt und oft sogar behindert wird.

YAGNI führt also in erster Linie zu schlankerem, einfacherem Code und ermöglicht damit, später kommende Anforderungen kostengünstig umzusetzen.

**Dependency Injection**

Abhängigkeitsinjektion

Reglementiert die Abhängigkeiten eines Objektes zur Laufzeit. Benötigt ein Objekt beispielsweise bei seiner Initialisierung ein anderes Objekt, ist diese Abhängigkeit an einem zentralen Ort hinterlegt - es wird also nicht vom initialisierten Objekt selbst erzeugt.

Ermöglicht Single-Responsibility-Prinzip: Verantwortlichkeit für den Aufbau des Abhängigkeitsnetzes zwischen den Objekten eines Programmes aus den einzelnen Klassen in eine zentrale Komponente zu überführen.

Dependency Injection überträgt die Verantwortung für das Erzeugen und die Verknüpfung von Objekten an eine eigenständige Komponente, wie beispielsweise ein extern konfigurierbares Framework. Dadurch wird der Code des Objektes unabhängiger von seiner Umgebung.

**Gesetz von Demeter**

Objekte sollen nur mit wenigen Objekten in ihrer unmittelbaren Umgebung kommunizieren -> Kopplung verringern und Wartbarkeit erhöhen

**Composition over inheritance**

Klassen sollten polymorphes Verhalten und Code-Wiederverwendung durch deren Zusammensetzung (beinhalten Instanzen anderer Klassen, die die gewünschte Funktionalität implementieren) erreichen, nicht durch Vererbung.

Interfaces beinhalten die Methoden, die jede ableitende Klasse implementieren muss. Interfaces ermöglichen polymorphes Verhalten.

**Self-documenting Code**

Namenskonventionen folgen, dadurch muss nicht immer die Struktur hinter der Methode/Klasse/Interface bekannt sein.

Ziele:

* Code ist einfacher zu lesen und zu verstehen
* Aufwand, um das System zu warten oder zu erweitern wird minimiert
* User und Entwickler müssen nicht auf weitere Dokumentationen des Codes zurückgreifen

**Design by contract**

Ziel ist das reibungslose Zusammenspiel einzelner Programmmodule durch die Definition formaler Verträge zur Verwendung von Schnittstellen, die über deren statische Definition hinausgehen.

Vertrag besteht beispielsweise über die Verwendung einer Method

* Vorbedingungen: Aufrufer muss diese einhalten
* Nachbedingungen: Aufgerufenes muss diese einhalten
* Invarianten: über alle Instanzen einer Klasse hinweg geltende Grundannahme

Invarianten sind logische Aussagen, die für alle Instanzen einer Klasse über den gesamten Objektlebenszyklus hinweg gelten.

**Information Hiding (Datenkapselung)**

Verbergen von Daten oder Informationen vor dem Zugriff von außen. Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt stattdessen über definierte Schnittstellen.

Zentrales Modell ist der abstrakte Datentyp, in dem Daten in einer Datenstruktur zusammengefasst sind, auf die nur festgelegte Zugriffsfunktionen zugegriffen werden kann.

anderes Beispiel: Verbergen von Daten innerhalb von Gültigkeitsbereichen.

public (+), private (-), protected (#), package (~)