SE Besprechung

Nachtrag zu Übung 3

SE, 17.11.09

Dustin Wüest

Übung 3, Aufgabe 2 – Nachtrag

- Mars-Roboter
  - → Gute, Vollständige Dokumentation wichtig!
    (siehe auch Ariane 5 Fallstudie)
  - → Zusammenarbeit mit Kunde **nicht nur** bei Agilen SE möglich
  - → Welche Teillieferungen? Roboter funktioniert nicht mit der Hälfte

Agile SE als Prozesswahl eher problematisch…

SE Besprechung

Übung 4 – Architektur, Modulentwurf

SE, 17.11.09

Dustin Wüest

Übungsabgaben

- Email Betreff

  SE EX HS09
- Zip-File

  Ex2_NachnameA_NachnameB_NachnameC.zip
- pdf-File

  Ex2_NachnameA_NachnameB_NachnameC.pdf
- Im pdf File

  Vollständige Namen + Matrikelnummern

- Bei Nichteinhalten ist Punktabzug möglich

Architekturstile & Entwurfsmuster

Architekturstil
definiert die strukturelle Organisation eines Systems
(globaler als Entwurfsmuster)

Framework / Bibliothek
übernimmt bestimmte Aufgaben (domänenspezifisch, z.B. ein Mathematik-Framework in C++, kann runtergeladen und verwendet werden)

Entwurfsmuster (Design Pattern)
adressiert ein bestimmtes Problem (domänenunabhängig), welches immer wieder auftauchen kann (muss man selber implementieren)

Aufgabe 2.1 – Architekturstile

Was sind die strukturellen Teile des Systems?

Wie kommunizieren diese Teile miteinander?

- Begründen der Auswahl → Brücke zwischen Text in der Aufgabe und dem gewählten Stil
- Komponenten, Konnektoren, **Einschränkungen**

Komponenten des Architekturstils ≠ System-Komponenten

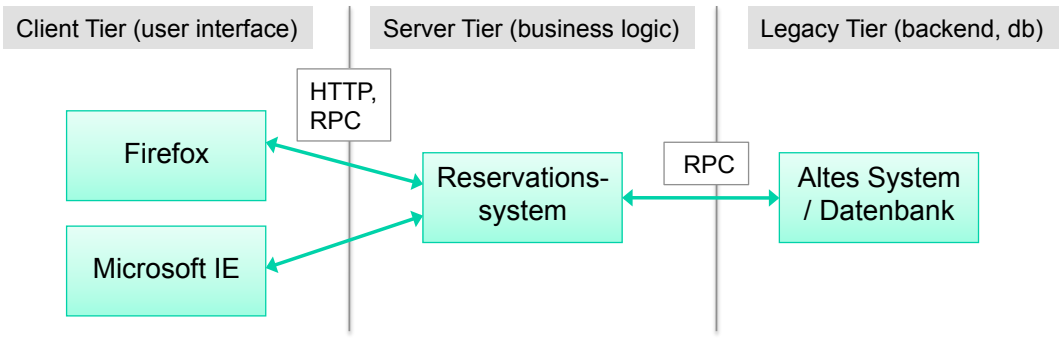
Aufgabe 2.1.A – Infrastruktur

- Layered Architecture **?**
  - Aufteilung nach Funktionalität, *oft hierarchisch (Abstraktion nimmt in höheren Schichten zu)*
  - Geheimnisprinzip: Spezifikation der Schnittstellen zwischen Schichten *(bzw. der Leistungen der jeweils unteren Schicht)*; einzelne Schichten können ausgetauscht werden, solange das Interface gleich bleibt
  - Sagt nichts über die physische Aufteilung / Aufteilung in mehrere Prozesse

Aufgabe 2.1.A – Infrastruktur

3-Tier Client/Server

- Mehrere Prozesse / verschiedene Speicherbereiche (*hier*: auf verschiedenen Computern)



Aufgabe 2.1.A – Infrastruktur

3-Tier Client/Server

Komponenten: unabhängig entwickelte Objekte / Programme bieten Operationen / Dienste an

HTTP, RPC — Konnektoren: Internetprotokolle (z.B. HTTP), RPC

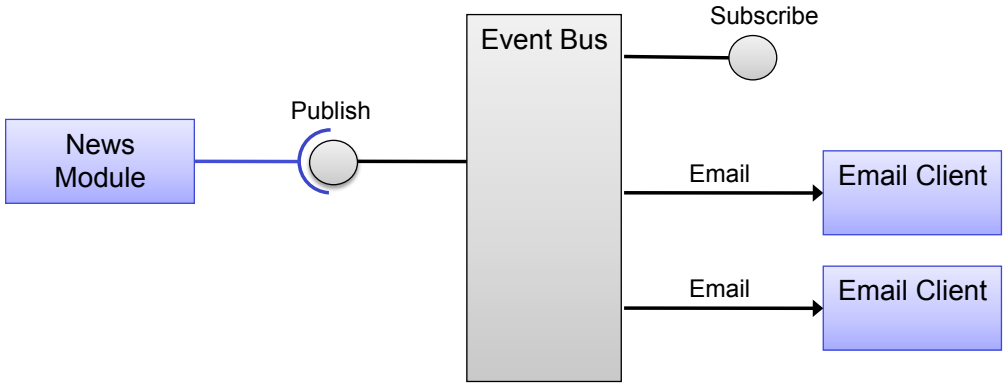
- (Konfigurationen: Persistente oder temporäre Verbindungen)
- Einschränkungen: „Point-to-Point“ Kommunikation

− Single Point of Failure

\+ Trennung der Aufgabenbereiche

Aufgabe 2.1.B – Theater News

Event-based System / Publish-Subscribe



Aufgabe 2.1.B – Theater News

Event-based System / Publish-Subscribe

Komponenten: Publishers / Subscribers

Konnektoren: Infrastruktur zur Übertragung von Events und zur Registrierung

Einschränkungen: Kommunikation nur über den Event Bus

\+ Komponenten sind lose gekoppelt

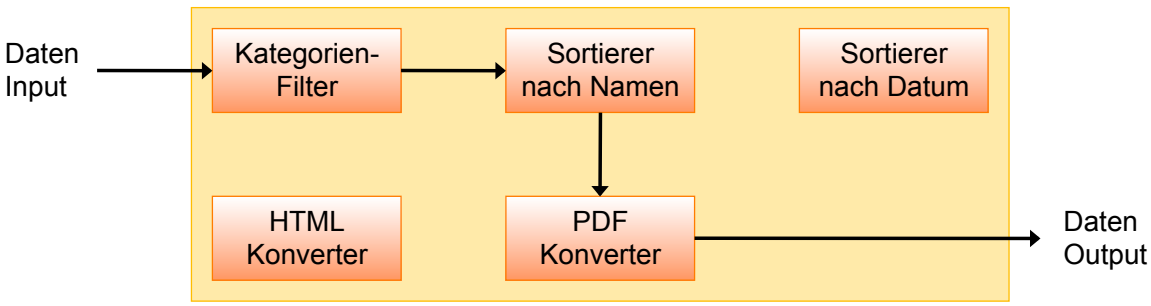
− Keine Garantie, dass Publisher eine Antwort bekommt

− Beliebige Reihenfolge der Antworten

...

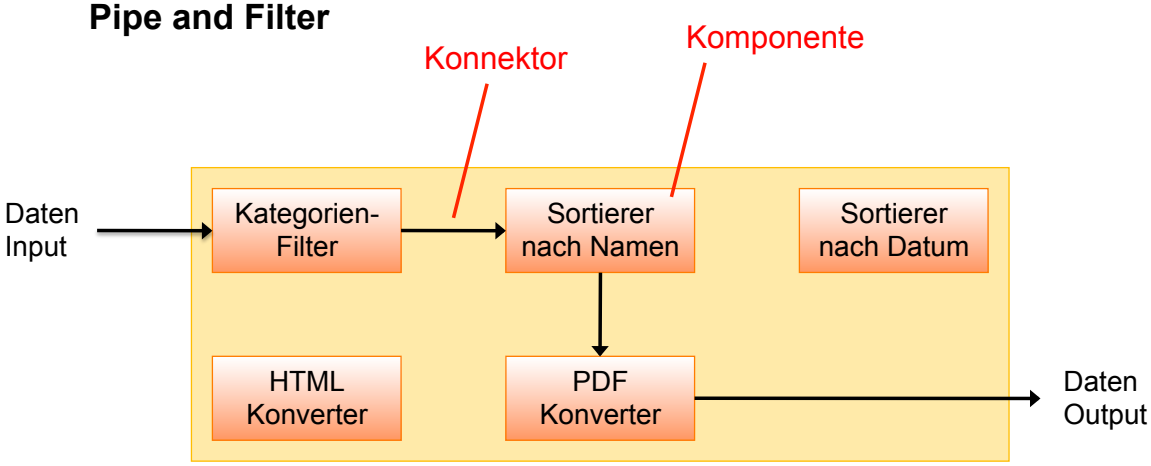
Aufgabe 2.1.C – Vorstellungsinfos

Pipe and Filter



Aufgabe 2.1.C – Vorstellungsinfos

Pipe and Filter



Aufgabe 2.1.C – Vorstellungsinfos

Pipe and Filter

Komponenten: Filter (Filter, Sortierer, Konverter)

Konnektoren: Pipes

Einschränkungen: Filter teilen keine Zustände, Filter kennen ihre Nachbarn nicht

\+ Wiederverwendbarkeit (hinzufügen, umsortieren, auswechseln von Filtern)

− Stapelverarbeitung (Batch Processing)

− Rechnungsaufwand (Overhead, z.B. muss jeder Filter die Datei erneut parsen)

Aufgabe 2.1.C – Vorstellungsinfos

Blackboard? → Nein

Motivation für Blackboard: mehrere Agenten arbeiten an Teillösungen, und benutzen dazu einen gemeinsamen Datenpool (lesen + schreiben)

Die Agenten reagieren auf Veränderungen im Blackboard, d.h. das Blackboard ist der Initiator / hat die Kontrolle

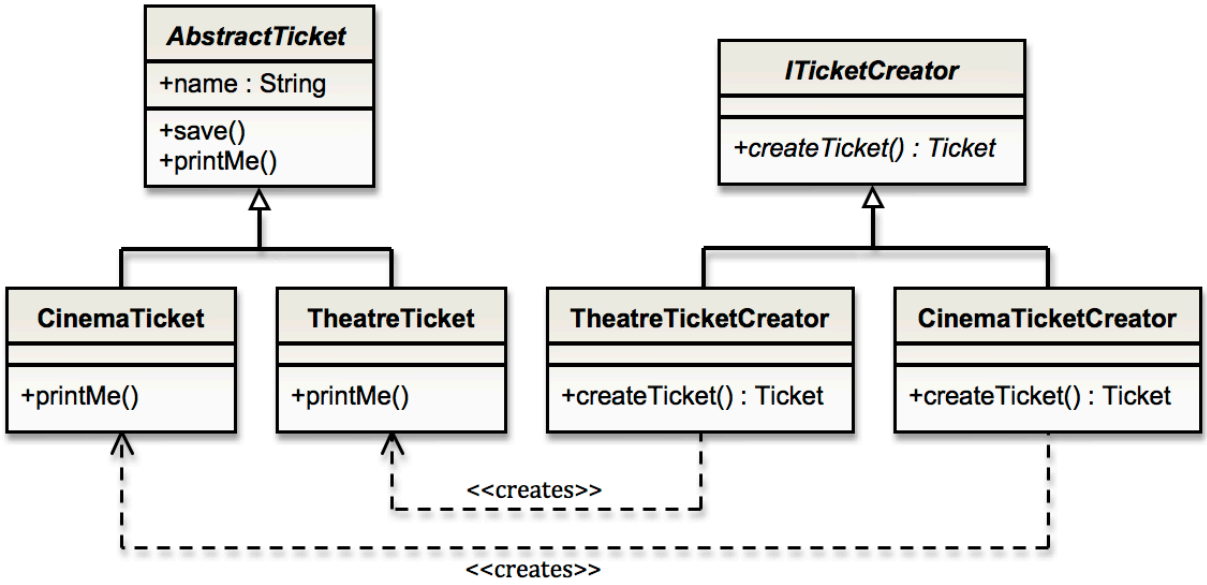
Für Problemlösungs-Prozesse

Blackboard-Architektur trifft allenfalls auf den ersten Satz der Aufgabe zu, ignoriert aber den Rest

Aufgabe 2.2.A – Entwurfsmuster

- **Erzeugungsmuster (Creational Patterns)**
  Verlagern der Objekt-Erzeugung
  (z.B. Singleton, Abstract Factory)
- **Strukturmuster (Structural Patterns)**
  Komposition von Klassen / Objekten
  (z.B. Adapter, Decorator)
- **Verhaltensmuster (Behavioral Patterns)**
  Interaktion von Klassen / Objekten
  (z.B. Strategy, Observer)

Aufgabe 2.2.B – Mehrere Tickettypen



Aufgabe 2.2.B – Mehrere Tickettypen

- **Abstract Factory**

- Die Rollen der am Entwurfsmuster beteiligten Klassen

ITicketCreator:	Abstrakte Fabrik
Theatre-/Cinema-TicketCreator:	Konkrete Fabriken
AbstractTicket:	Abstraktes Produkt
Theatre-/Cinema-Ticket:	Konkrete Produkte

Aufgabe 2.2.B – Mehrere Tickettypen

- **Implementierung des Musters ‘Abstract Factory‘**

Ticket

CinemaTicket

TheatreTicket

ITicketCreator

CinemaTicketCreator

TheatreTicketCreator

TicketApp1

Aufgabe 2.2.B – Mehrere Tickettypen

- **Implementierung des Musters ‘Abstract Factory‘**

Ticket

CinemaTicket — Keine Änderungen nötig

TheatreTicket

ITicketCreator

CinemaTicketCreator

TheatreTicketCreator

TicketApp1

Aufgabe 2.2.B – Mehrere Tickettypen

- **Implementierung des Musters ‘Abstract Factory‘**

Ticket
CinemaTicket
TheatreTicket
ITicketCreator
CinemaTicketCreator
TheatreTicketCreator
TicketApp1

```
/**
 * Creates tickets.
 *
 * @author      Dustin Wueest
 * @history     21.10.09 DW first version
 * @version     21.10.09 1.0
 * @responsibilities creates tickets
 */
public interface ITicketCreator {

    public abstract Ticket createTicket();
}
```


Aufgabe 2.2.B – Mehrere Tickettypen

- **Imple**

Ticket
CinemaT
TheatreT
ITicketCr
CinemaT
TheatreT
TicketAp

```
/**
 * Creates tickets for cinema screenings.
 *
 * @author      Dustin Wueest
 * @history     21.10.09 DW first version
 * @version     21.10.09 1.0
 * @responsibilities creates cinema tickets
 */
public class CinemaTicketCreator implements ITicketCreator {

    public Ticket createTicket() {
        return new CinemaTicket();
    }

}
```

Aufgabe 2.2.B – Mehrere Tickettypen

- **Implementierung des Musters 'Abstract Factory'**

Ticket

CinemaT

TheatreT

ITicketCr

CinemaT

TheatreT

TicketAp

```
/**
 * Creates tickets for theatre shows.
 *
 * @author      Dustin Wueest
 * @history     21.10.09 DW first version
 * @version     21.10.09 1.0
 * @responsibilities creates theatre tickets
 */
public class TheatreTicketCreator implements ITicketCreator {

    public Ticket createTicket() {
        return new TheatreTicket();
    }

}
```



```
/**
 * Test application, creates and prints tickets for the theatre and cinema.
 *
 * @author      Dustin Wueest
 * @history     21.10.09 DW first version
 * @version     21.10.09 1.0
 * @responsibilities creates and prints tickets
 */
public class TicketApp2 {

    private static ITicketCreator creator = new TheatreTicketCreator();

    public static void main(String[] args) {
        TicketApp2 app = new TicketApp2();
        app.methodOne();
    }

    public void methodOne() {
    Ticket ti = creator.createTicket();
    ti.printMe();
    }

    public void methodTwo( String name ) {
    Ticket ti = creator.createTicket();
    ti.setPerson( name );
    ti.save();
    }
```

```java
/**
 * Test application, creates and prints tickets for the theatre and cinema.
 *
 * @author      Dustin Wueest
 * @history     21.10.09 DW first version
 * @version     21.10.09 1.0
 * @responsibilities creates and prints tickets
 */
public class TicketApp2 {

    private static ITicketCreator creator = new TheatreTicketCreator();

    public static void main(String[] args) {
        TicketApp2 app = new TicketApp2();
        app.methodOne();
    }

    public void methodOne() {
    Ticket ti = creator.createTicket();
    ti.printMe();
    }

    public void methodTwo( String name ) {
    Ticket ti = creator.createTicket();
    ti.setPerson( name );
    ti.save();
    }
```

Abstract Factory vs. Factory Method

- **Factory Method**

  Subklassen sollen entscheiden, welches Produkt erzeugt wird.

  z.B.: TicketApp1 hat eine abstrakte Methode *createTicket()*.

  CinemaTicketApp extends TicketApp1, und implementiert die Methode *createTicket()*.

  Motivation:

  – Eine Klasse kennt nicht die konkrete Klasse der Objekte, welche sie erstellen soll.

  – Die Subklassen sollen die zu erstellenden Objekte spezifizieren

Abstract Factory vs. Factory Method

- **Abstract Factory**

  Eine separate Klasse (Factory) ist für die Objekterzeugung zuständig. Die Factory Klasse ist abstrakt.

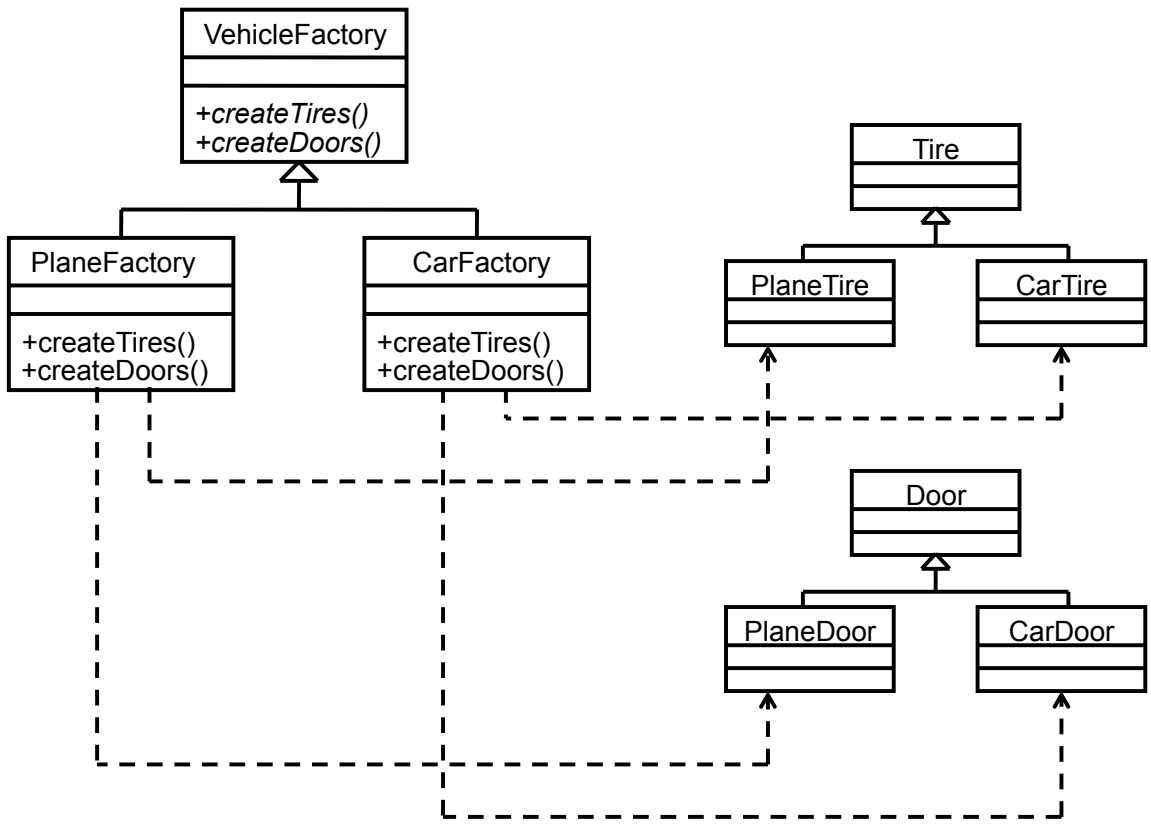
  Die konkreten Produktklassen werden in Subklassen der Factory erzeugt (→ Factory besitzt eine Factory Method)

  z.B.: TicketApp1 benutzt zur Objekterstellung eine “Factory-Klasse“, welche eine Methode *createTicket()* besitzt (wie in der Aufgabe).

  Motivation:

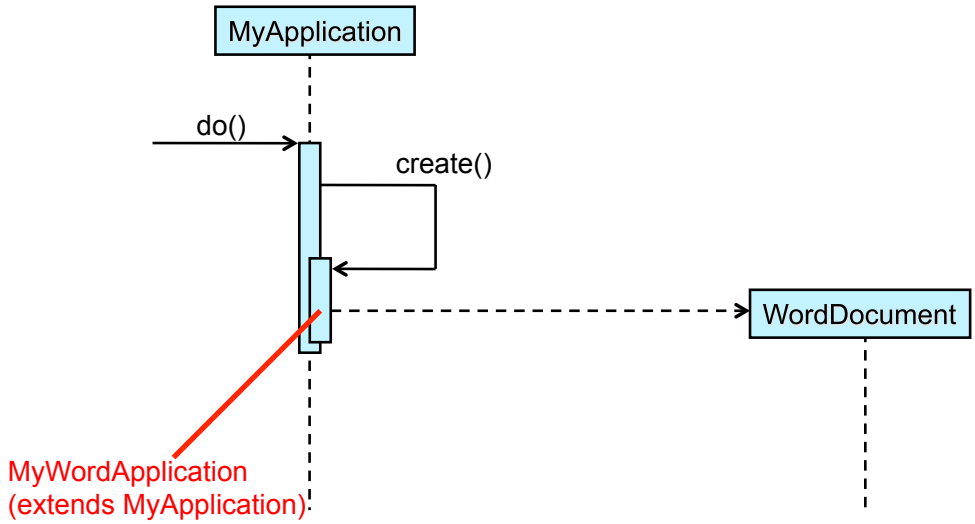
  – System soll unabhängig davon sein, wie die Produkte erstellt und repräsentiert werden

  – System soll mit einer Familie von Produkten konfiguriert werden



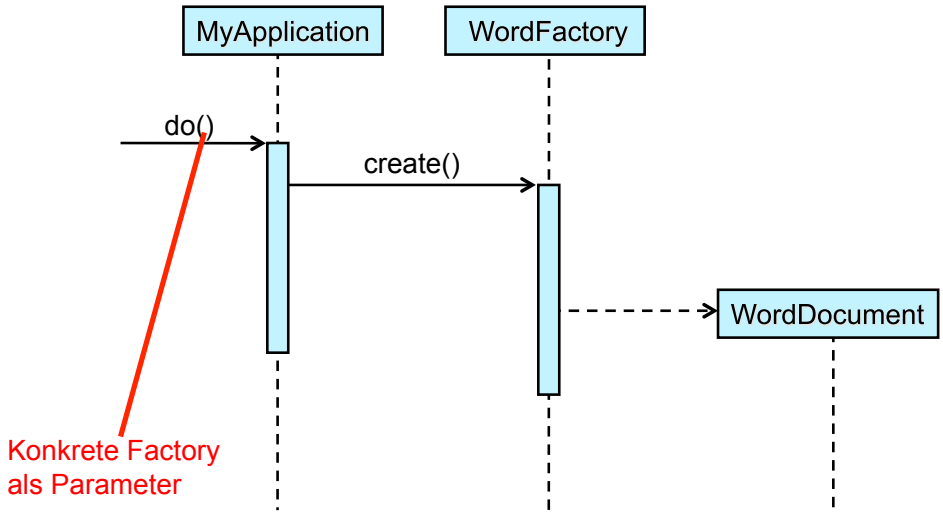
Abstract Factory vs. Factory Method

Factory Method



Abstract Factory vs. Factory Method

Abstract Factory



Aufgabe 2.2.C – Einsamer Creator

Singleton, es kann nur eine Instanz von ObjectCreator geben

Kann analog für Cinema-/Theatre-TicketCreator implementiert werden

```
public class ObjectCreator {

    private static ObjectCreator instance;

    //The only Constructor of this class.
    private ObjectCreator() {
    }

    public static ObjectCreator getInstance() {
        if( ObjectCreator.instance == null ) {
            ObjectCreator.instance = new ObjectCreator();
        }
        return ObjectCreator.instance;
    }
}
```


Aufgabe 2.2.C – Einsamer Creator

Singleton, es kann nur eine Instanz von ObjectCreator geben

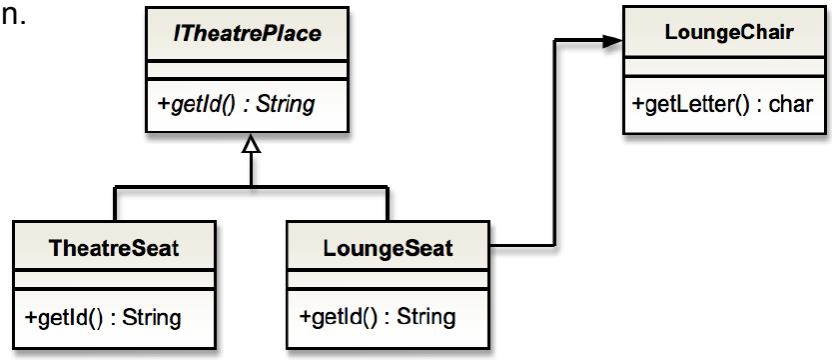
Kann analog für Cinema-/Theatre-TicketCreator implementiert werden

```
public class ObjectCreator {

    private static ObjectCreator instance;

    //The only Constructor of this class.
    private ObjectCreator() {
    }

    public static ObjectCreator getInstance() {
        if( ObjectCreator.instance == null ) {
            ObjectCreator.instance = new ObjectCreator();
        }
        return ObjectCreator.instance;
    }
}
```

Aufgabe 2.2.D – Inkompatible Sitze

Adapter

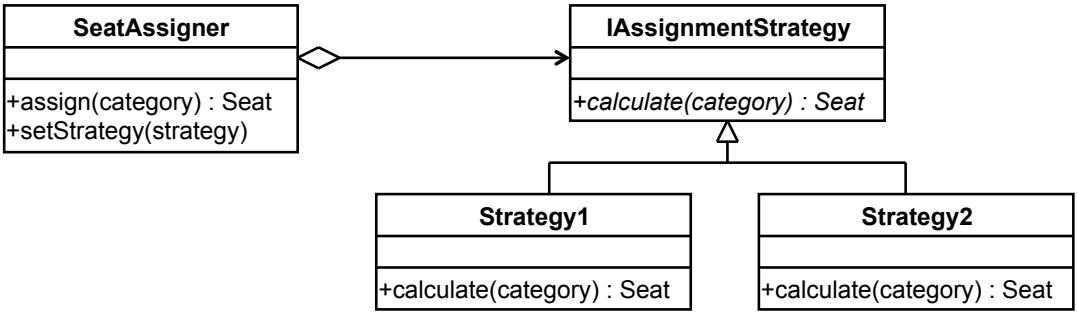
LoungeSeat besitzt ein LoungeChair Attribut. Ein LoungeChair wird dem LoungeSeat bei der Erzeugung als Parameter übergeben. In der getId() Methode (oder im Konstruktor) wird die getLetter() Methode aufgerufen, der retournierte CHAR wird in einen STRING umgewandelt und zurückgegeben.



Aufgabe 2.2.E – Autom. Zuordnung

Strategy

Konkrete Algos, Interface davon, Eine Klasse („Kontext“), die ein Attribut „Algorithmus“ hat; und von aussen (von einer anderen Klasse) kann einer der bestimmten Algos übergeben werden



Übung 4 Resultate