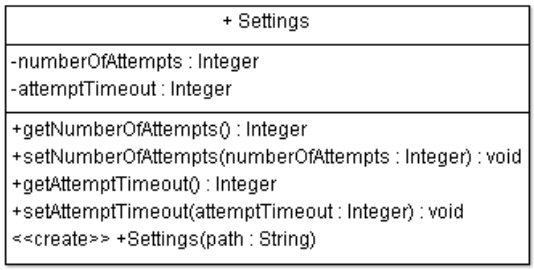
Informatik: Singleton und Observer Pattern

- Verstehen wie in Java das Singleton Pattern implementiert wird
- Vor- und Nachteile dieses Patterns erkennen
- Verstehen wann das Factory methode Pattern sinnvoll eingesetzt werden soll
- Das wichtige Observer Pattern verstehen und anwenden können
- Verstehen welche Möglichkeiten Java bietet das Observable Pattern zu verwenden (Klasse `Observable` und Interface `Observer`)

Einleitendes Beispiel Singleton: Einstellungen

Über eine Klasse sollen anwendungsweit verwendbare Einstellungswerte zur Verfügung gestellt werden die in Datei abgelegt worden sind

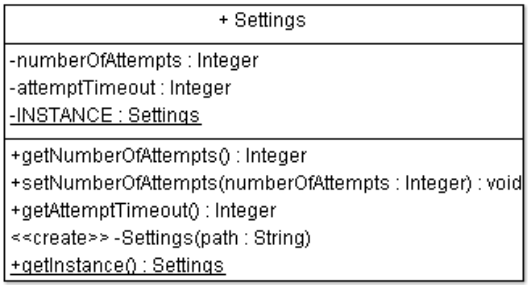
Schlechte Lösung



An verschiedensten Stellen (Clients) im Code wird `Settings`-Objekt wiederholt instanziiert und liefert inkonsistente Werte

Bessere Lösung

Das Settings-Objekt soll nur einmal instanziiert und allen Clients zur Verfügung gestellt werden. Alle Clients arbeiten mit ein und demselben Objekt




```
public class Settings
{
  private static final String PATH = "settings.cfg";
  private static final Settings INSTANCE = new Settings(PATH);
  private Integer numberOfAttempts = -1;
  private Integer AttemptTimeout = -1;
  private Settings(String path) {
    // Schreibt Einstellungen aus Datei in die Membervariablen
    …
  }
  public static Settings getInstance() {
    return INSTANCE;
  }
  // Getters und Setters für numberOfAttempts und AttemptTimeout
  …
}
```

- Konstruktor ist privat
- Klasse übernimmt Kontrolle über das Anlegen des Objektes
- Sofortiges Laden *Eager Loading* vs. Anlegen bei Bedarf *Lazy Loading*
- Über Klassenmethode `getInstance()` wird Objekt zur Verfügung gestellt
- Geeignet für Manager-Klassen (z.B. `EntityManager` bei DB-Zugriff)
- Evtl. Synchronisierungsproblem, wenn mehrere Clients nebenläufig auf Objekt zugreifen!!! (Beim Eager Loading kein Sync.problem)

☹ Instanz ist überall verfügbar ➔ schwer nachvollziehbar wer sie verwendet

☹ Instanz ist immer dieselbe ➔ nicht auf Variationen ausgelegt

☹ Nachträgliche Änderung äußerst schwierig

Singleton

- *"Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit."*

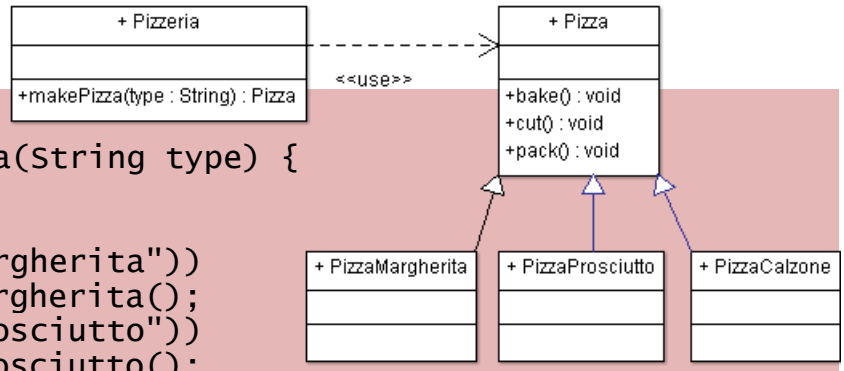
[1]

[1] Nach Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley. 1. Auflage 1996 (***Gang of Four***)

Einleitendes Beispiel Fabrikmethode: Pizzeria

Eine Pizzeria bietet verschiedene Pizzas an die gebacken, zerteilt und verpackt ausgeliefert werden

Schlechte Lösung

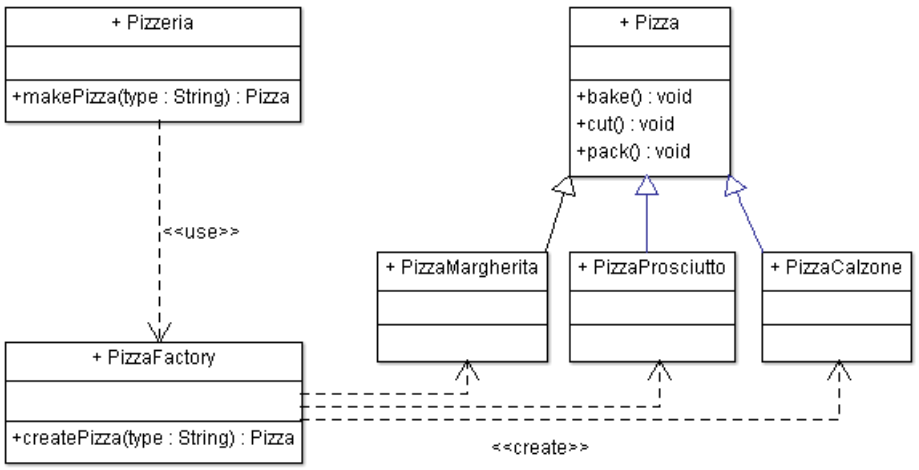



```
public class Pizzeria {
  public Pizza makePizza(String type) {
    Pizza ret = null;
    // Instanziierung
    if (type.equals("Margherita"))
      ret = new PizzaMargherita();
    if (type.equals("Prosciutto"))
      ret = new PizzaProsciutto();
    if (type.equals("Calzone"))
      ret = new PizzaCalzone();
    // Verarbeitung
    ret.bake();
    ret.cut();
    ret.pack();
    return ret;
  }
}
```

- ☹ Instanziierung und Verarbeitung werden zusammen durchgeführt
- ☹ Geringe Wiederverwendbarkeit: Durch Ableitung kann keine Pizzeria erstellt werden die andere Pizzas anbietet, den Verarbeitungscode aber gleich verwenden möchte

Mögliche Lösung

Auslagern der Instanziierung in eine eigene Klasse, Pizzeria muss nicht geändert werden, wenn Angebot angepasst wird



```
public class Pizzeria {
  private PizzaFactory pizzaFactory = new PizzaFactory();
  public Pizza makePizza(String type) {
    // Instanziierung
    Pizza ret = pizzaFactory.createPizza(type);
    // Verarbeitung
    ret.bake();
    ret.cut();
    ret.pack();
    return ret;
  }
}
```

☺ Trennung der Objekterstellung von Objektverarbeitung

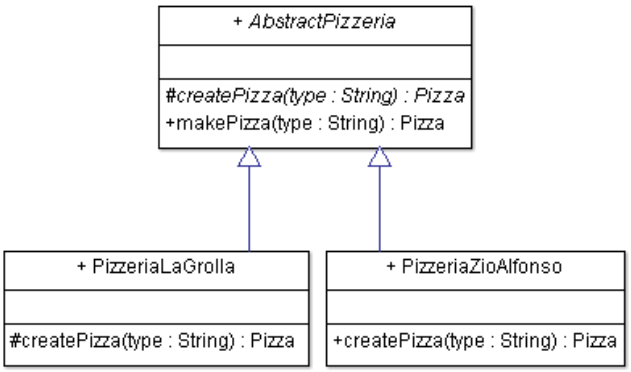
☺ `PizzaFactory` ist wiederverwendbar insbesondere, wenn sie abstrakt ist. Dadurch können die Angebote einzelner Pizzerias erstellt werden: `LaGrollaFactory`, `ZioAlfonsFactory`

PROBLEM: Anhand der Factory können Objekte erstellt werden, die unverarbeitet bleiben

Lösung Fabrikmethode (Factory Method Pattern)

Die als `abstract` definierte Factory-Methode sorgt in abgeleiteter Klasse für das Anlegen der Objekte.

Sie ist `protected` so dass sie nicht separat aufgerufen werden kann




```
public abstract class AbstractPizzeria
{
  protected abstract Pizza createPizza(String type);
  public Pizza makePizza(String type) {
    // Instanziierung
    Pizza ret = createPizza(type);
    // Verarbeitung
    ret.bake();
    ret.cut();
    ret.pack();
    return ret;
  }
}
```

```
public class PizzeriaLaGrolla extends AbstractPizzeria
{
```

```
  protected Pizza createPizza(String type) {
    Pizza ret = null;
    if (type.equals("Margherita"))
      ret = new PizzaMargherita();
    else if (type.equals("Prosciutto"))
      ret = new PizzaProsciutto();
    return ret;
  }
}
```

Factory Method

- "Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren."

[2]

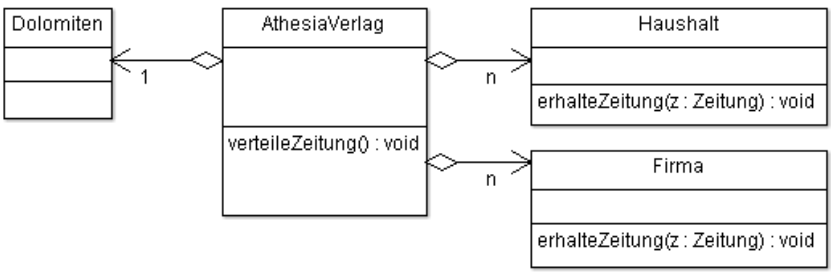
Unsere Klassenbezeichnung	Allgemeine Klassenbezeichnung
`AbstractPizzeria`	`Creator`
`PizzeriaLaGrolla`	`ConcreteCreator`
`Pizza` (evtl. `abstract`)	`Product`
`PizzaMargherita`	`ConcreteProduct`

Einleitendes Beispiel: Observer Pattern

Der Athesia-Verlag will seine Tageszeitung Dolomiten an Haushalte und Firmen verteilen. Bei jeder neuen Zeitung soll eine Auslieferung erfolgen.

Schlechte Lösung

Verlag ruft in `verteileZeitung()` für jeden Empfänger `erhalteZeitung()` auf um Zeitung zu übermitteln

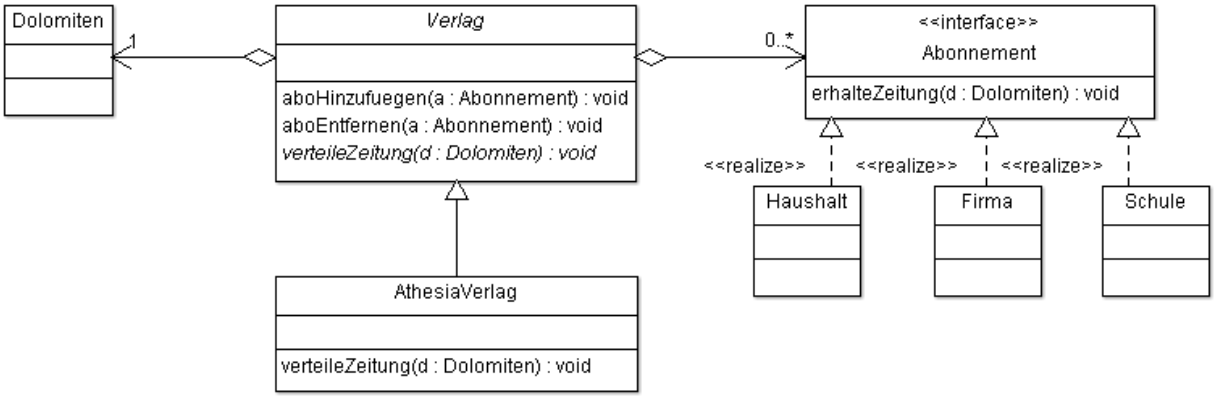


[2] Nach Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley. 1. Auflage 1996 (*Gang of Four*)

☹ Kommt neuer Empfängertyp (z.B. Schule) dazu, muss Verlag angepasst werden.

☹ Es besteht keine Möglichkeit Abonnenten dynamisch hinzuzufügen bzw. zu entfernen

Bessere Lösung



☺ Alle Abonnenten implementieren gemeinsame Schnittstelle `Abonnement`

☺ Unterschiedlichste Objekte können Abonnenten werden

☺ Alle Abonnenten können in einer einfachen Liste geführt werden

☺ Bei `verteileZeitung()` muss einfach nur über Liste iteriert werden

☺ Abonnenten können sich beim Verlag dynamisch an- bzw. abmelden

☺ Administrationsmethoden (`aboHinzufuegen()`, `aboEntfernen()`) sind ausgelagert

Observer

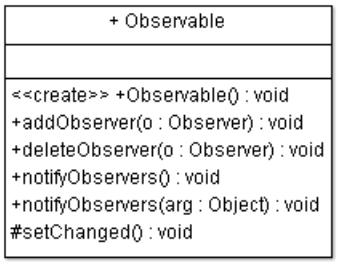
- "Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, das alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden."

[3]

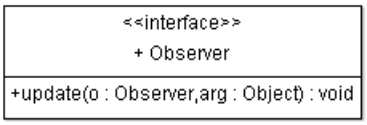
Unsere Klassenbezeichnung	Allgemeine Klassenbezeichnung
`AthesiaVerlag`	`Subject` oder `Observable`
`Abonnent`	`Observer` oder `Dependents`
`verteileZeitung()`	Methode mit der Benachrichtigung aller Observer ausgelöst wird
`erhalteZeitung()`	Aktualisierungsmethode (oder Updatemethode, Benachrichtigungsmethode) der Observer

Realisierung in Java

Klasse



Interface[4]



ACHTUNG

`setChanged()` muss unbedingt vor `notifyObservers()` aufgerufen werden damit letztere Methode ihre Observer benachrichtigt

---

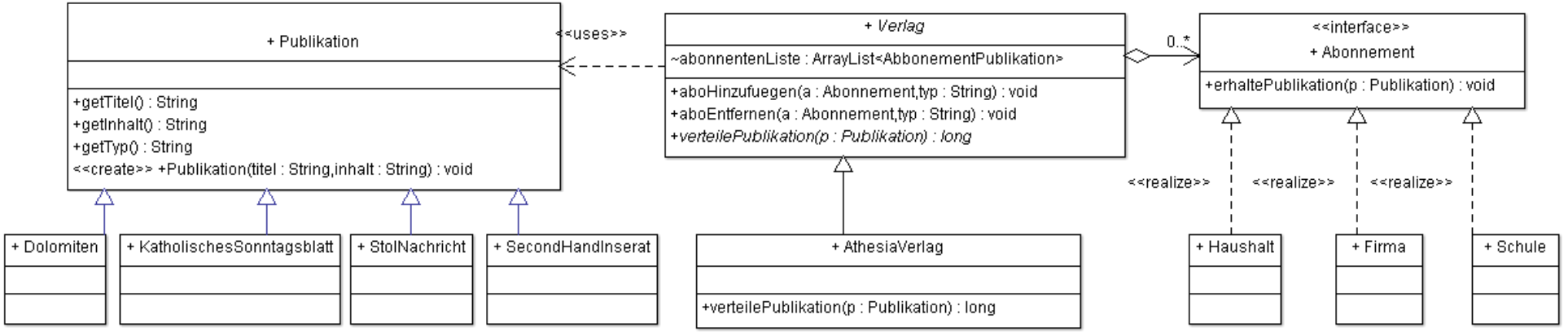
[3] Nach Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software". Addison-Wesley. 1. Auflage 1996 (***Gang of Four***)

[4] Ab Java 9 als *deprecated* gekennzeichnet, weil `Observable` nur als Klasse verwendbar und in der abgeleiteten Klasse die eigentlich threadsicheren Methoden überschrieben werden könnten

AUFGABE

Wie würden Sie obiges Klassendiagramm anpassen damit ein Verlag unterschiedlichste Zeitungen (z.B. Dolomiten, Katholisches Sonntagsblatt, usw.) und sogar Nachrichten (z.B. Stol.it, Second-Hand, usw.) anbieten kann?

LÖSUNG



- `getTyp()` liefert `Dolomiten`, `KatholischesSonntagsblatt`, `StollNachricht`, `SecondHandInserat` zurück
- `erhaltePublikation()` gibt Typ des Abonnenten (`Haushalt`, `Firma`, `Schule`) und Titel der Publikation aus (z.B. `Schule Schule 1 erhält StolNachricht 1`)
- `proteced` Klasse `AbonnementPublikation` enthält Abonnement und Typ
- `verteilePublikation()` liefert die Anzahl der verteilten Publikationen zurück