Wahlpflichtfach Design Pattern

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- miwilhelm@hs-harz.de
- http://www.miwilhelm.de
- Raum 2.202
- Tel. 03943 / 659 338

Inhalt

Entwurfsmuster: „Abstrakte Factory“

- Fabrik-Entwurfsmuster dienen zur konsistenten Erzeugung voneinander abhängiger Klassen.
- Die Abstrakte Fabrik ist ein Entwurfsmuster der Softwareentwicklung und gehört zu der Kategorie der Erzeugungsmuster.
- Es definiert eine Schnittstelle zur Erzeugung einer Familie von Objekten.
- Die konkreten Klassen der zu instanziierenden Objekte werden **nicht näher** festgelegt.
- Einsatz des Fabrikmusters
  - Fabriken werden genutzt, um Software flexibel gegenüber Änderungen zu machen. Im nächsten Beispiel muss ein neuer Kämpfer nur (an einer Stelle) in die Fabrik integriert werden.

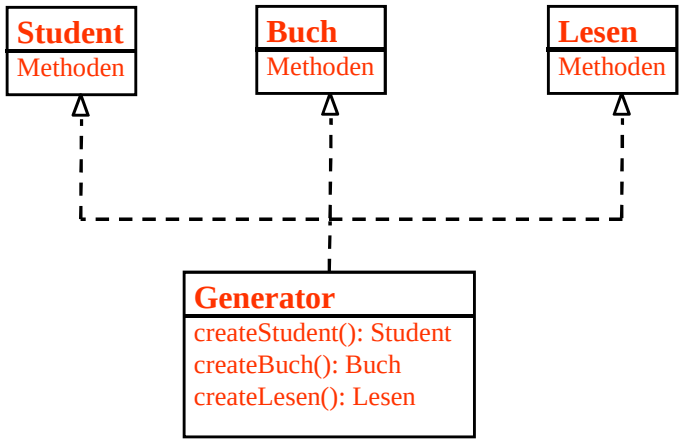
Einstieg-Beispiel

Student will Bücher lesen:

Person: Student
Objekt: Buch
Aktion: Buch lesen

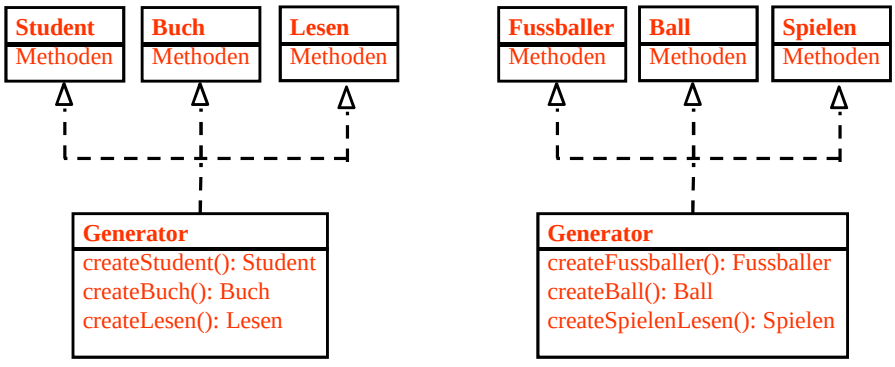
Benutzt wird ein Generator zum Erzeugen der drei Objekte:

Generator:
- createStudent()
- createBuch()
- createLesen()



Über den Generator, der Fabrik, kann der Anwender beliebige Elemente erstellen:
Über den Generator erstellt der Client seine konkreten „Spielweltelemente“

Mehrere Generatoren



Vorteile und Nachteile

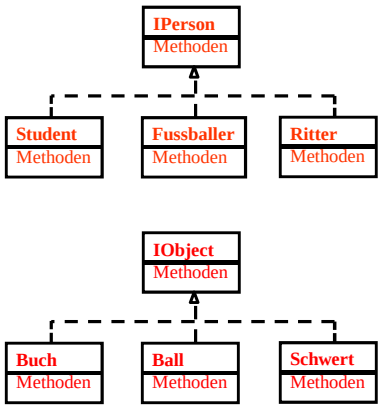
Inkonsistenz.
Der Programmierer kann nun die „Klassen“ beliebig miteinander kombinieren. Dies ist nicht vorgesehen, da Studenten nicht Fussball spielen können und Fussballer nicht lesen. Das "Framework" ermöglicht Inkonsistenz und damit Fehlfunktionen.

Keine Allgemeingültigkeit.
Der Programmierer kann keinen **allgemeingültigen** Code schreiben. Hat er Code für das „Lesen“ geschrieben und will nun die Fussballer programmieren, so muss sein Code komplett umschreiben (ein Wartungsalptraum).

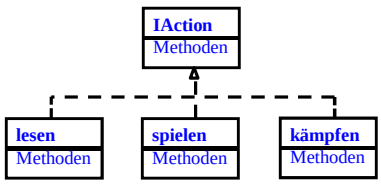
Keine Erweiterbarkeit.
Ein Hauptgrund liegt in der Verwendung von konkreten Klassen.
Dies führt zu einer **engen Kopplung**.
Eine Factory, die **konkrete Klassen** zurückgibt, ist sinnlos.
Sie verfehlt den Zweck der Factorys:
Flexibler Austausch von Implementierungen.

Abhilfe: Interfaces

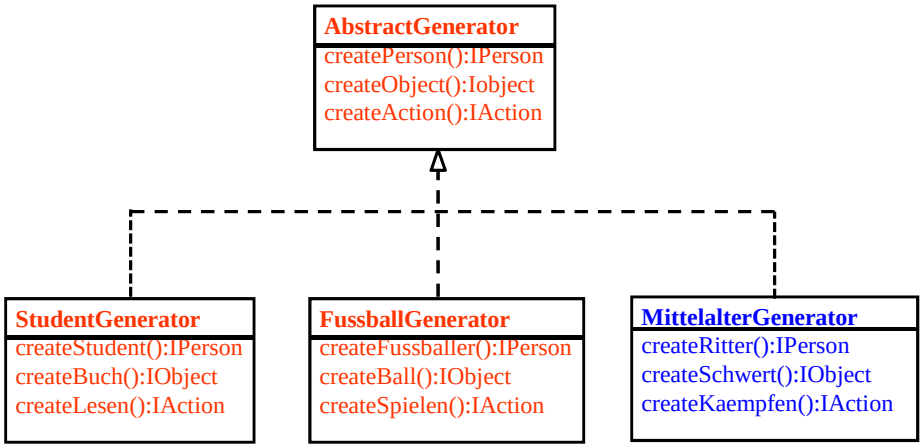
Eigenschaften
- Die Interfaces bilden ein abstraktes Schema für die einzelnen Objekte
- Trotzdem kann man die „Instanzen“ mischen (ArrayList)



Abhilfe:
- Definition eines abstrakten Generators
- Einbau von drei Generatoren



Interface: AbstractGenerator



Klassen und Interfaces

MainArena.java	JFrame-Hauptprogramm
Arena.java	Eigentliche Action-Datei
Fighter.java	Implementiert einen Kämpfer
Ort.java	Implementiert einen Ort
Generator.java	Abstrakter Generator
Gen_AngryBirds.java	Generator für Angry Birds
Gen_CatsDogs.java	Generator für Cats and Dogs
IFighter.java	Schnittstelle für den Kämpfer
IOrt.java	Schnittstelle für einen Ort

Klassen

```
public abstract class Generator {
  public abstract  IFighter  getAngreifer();
  public abstract  IFighter  getVerteidiger();
  public abstract  IOrt      getOrt();
}

public class Fighter implements  IFighter {
  String name;
  public Fighter(String name) {
    this.name=name;
  }
  public String getName() {
    return name;
  }
}

public interface IFighter {
    public String getName();
}
```


Klassen

```
public interface IOrt {
  public String getName();

}


public class Ort  implements IOrt{
    String name;

  public Ort(String name) {
    this.name=name;
  }

  public String getName() {
    return name;
  }

}
```

Klassen

```
public class Gen_CatsDogs   extends Generator {

  @Override
  public IFighter getAngreifer() {
    return new Fighter("Cats");
  }

  @Override
  public IFighter getVerteidiger() {
    return new Fighter("Dogs");
  }

  @Override
  public IOrt getOrt() {
    return new Ort("Innenstadt");
  }

}
```


Klassen

```
public class  Gen_AngryBirds  extends Generator {


  @Override
  public IFighter getAngreifer() {
    return new Fighter("Birds");
  }

  @Override
  public IFighter getVerteidiger() {
    return new Fighter("Pigs");
  }

  @Override
  public IOrt getOrt() {
    return new Ort("Wald und Sumpf, AngryBirds");
  }

}
```

```
public class Arena  extends JFrame implements ActionListener{

 JTextArea editor = new JTextArea();
 JButton bnFight = new JButton("Kämpfen");

 IFighter vert;
 IFighter angrf;
 IOrt ort;

  public Arena(Generator gen) {
    vert = gen.getVerteidiger();
    angrf = gen.getAngreifer();
    ort=gen.getOrt();

   editor.setText("Arena:");
   editor.append("\nPerson1: "+vert.getName());
   editor.append("\nPerson2: "+angrf.getName());
   editor.append("\nOrt: "+ort.getName());

   bnFight.addActionListener(this);
   setVisible(true);
  }
```

```
  private void bnFight_click() {
      editor.append("\n\nKampf:"+
      angrf.getName()+"  vs. "+
      vert.getName()+"\n");
  }
```


Eigenschaften

- Code des Interfaces „AbstractGenerator“ und die jeweilige Implementierungen ist unabhängig.
- Welches konkrete Objektfamilie durch den Generator zurückgeliefert wird, entscheidet allein die Implementierung.
- Was ändert sich für den Programmierer?
  Er ist von der realen Implementierung entkoppelt.
  Er benutzt die Schnittstellen-Methoden.
  Der Programmierer kann nun **allgemeingültigen** Code schreiben.

Eigenschaften

Abstrakte Fabriken erlauben eine Flexibilität und Allgemeingültigkeit durch Entkopplung des Programmieres von konkreten Implementierungen.

Der Typ der Welt, echter Generator, wird an einer zentralen Stelle festgelegt. Danach kann man die abtrakten Methoden benutzen.

Der Programmierer weiß nicht, dass er in Wirklichkeit mit Ritter, Fussballer, Studenten arbeitet. Das ermöglicht das denkbar einfache Austauschen des Generators und damit der gesamten Objektfamilie.

Erweiterbarkeit und Wartbarkeit.

Damit können schnell neue Welten ins System integriert werden. Dazu muss lediglich der AbstractGenerator implementiert werden
Alles ohne bestehenden Code zu brechen.

.

Eigenschaften

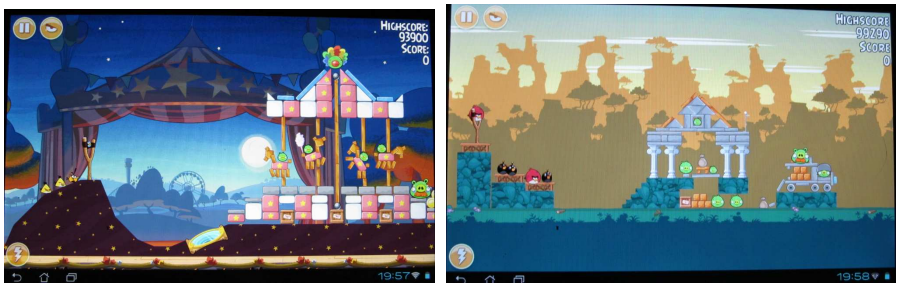
Wartbarkeit.

Man kann damit auch sehr leicht Änderungen an bestehenden Welten durchführen. Statt Ritter gibt es nun Degenkämpfer. Es entsteht kein Änderungsaufwand am Client, dank allgemeingültigen, auf Abstraktion gestützten Code.

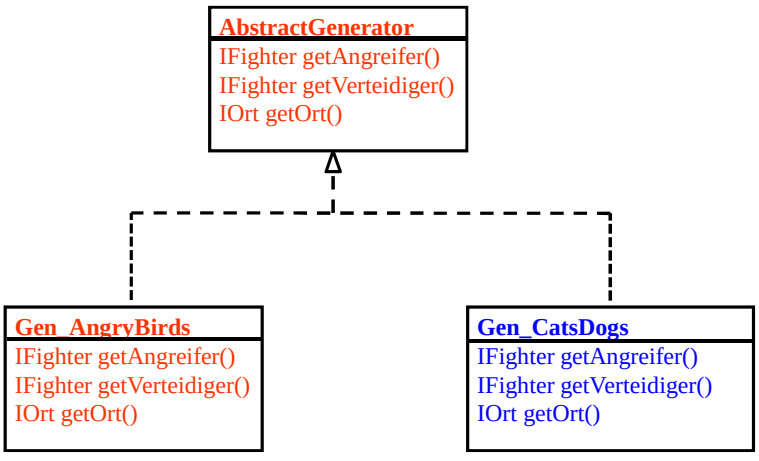
Konsistenz.

Die zusammengefassten Generatoren stellen sicher, dass nur Objekte erstellt werden, die auch zueinander passen und miteinander funktionieren.
Mit einem Singleton kann man nun einen neue Fabrik erstellen.
Diese Fabrik kapselt dann die drei Fabriken. Damit kann ein Anwender nicht zwei Generatoren erzeugen.

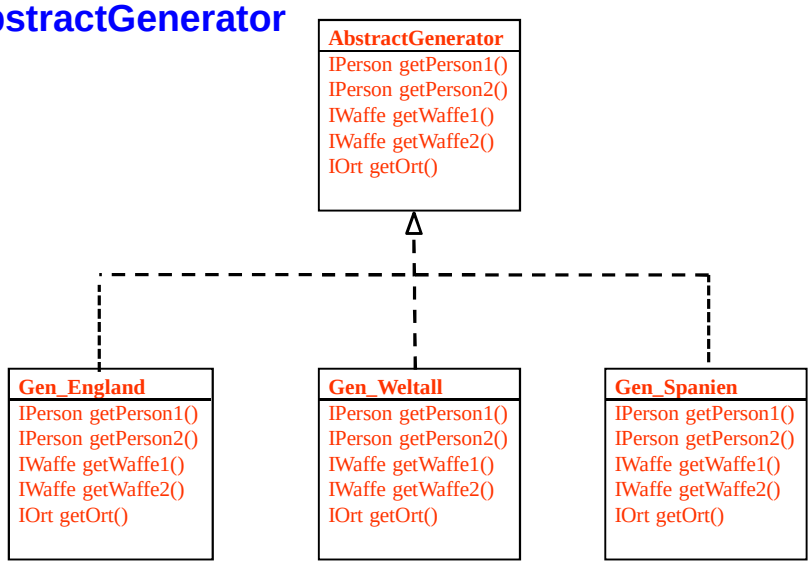
Anwendungsbeispiel: Angry Birds



AbstractGenerator



AbstractGenerator



Klassen und Interfaces des Vorlesungsbeispiels

MainLabor.java
Arena.java

Generator.java
Gen_England.java
Gen_Spanien.java
Gen_Starwars.java

IOrt.java
IPerson.java
IWaffe.java

Pers_England.java
Pers_Spanien.java
Pers_Weltall.java

Pers_England.java
Pers_Spanien.java
Pers_Weltall.java

Lichtschwert.java
Ball.java
PfeilBogen.java
Schwert_Schild.java

Rasen.java
Wald.java
Weltall.java