Graphische Nutzerschnittstellen

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

Inhalt

Internet-Adressen

- www.dofactory.com/Patterns/
- www.patterndepot.com
- www.javapractices.com
- http://www.se.uni-hannover.de/documents/kurz-und-gut/creational-patterns_tliro.pdf

Literatur

- Design Patterns
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Holub on Patterns: Learning Design Patterns by Looking at Code, 978-1590593882
- Pattern-Oriented Software Architecture
  Volume 1, ISBN 978-0471958697
- Pattern-Oriented Software Architecture
  Volume 2, ISBN: 978-0471606956

Software-Entwicklung

- Assembler
- Prozeduraler Ansatz (Daten und Methoden getrennt)
- Modulkonzept (Daten und Methoden zusammen)
- Objektorientierte Entwicklung
- Strukturierte Analyse / Design
- Objektorientierte Analyse / Design
- Jeder Lösungsansatz wird immer wieder neu entwickelt

Entwurf objektorientierter Software ist nicht leicht

- Was sind Objekte ?
- Was sind Attribute / Felder ?
- Definition der Schnittstellen (get/set)
- Bestimmen der Methoden
- Abhängigkeit zwischen Objekten
- Verallgemeinerung
- Wiederverwendbarkeit

Experte:

Experten wissen zu vermeiden, jedes Problem von Grund auf neu zu anzugehen.

- Bibliotheken
- Packages
- Templates
- OOP
- Schnittstellen

Sie verwenden Lösungen, die sie zuvor erfolgreich eingesetzt haben.

Definition von Muster mit Lösungsschemen

Der Begriff des Musters wurde vom Architekten Christopher Alexander 1977 wie folgt definiert:

„Jedes Muster beschreibt ein in unserer Umwelt beständiges, wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.“

Ein Muster stellt also eine bewährte Lösung nach Art einer Schablone für ein häufiges Probleme bereit. So gesehen sind bereits die länger existierenden Algorithmensammlungen wie z.B. „The Art of Computer Programming“ von Donald Knuth eine Katalogisierung von Mustern (vergl. [Gamma97, S.392]).

Die Beschreibung eines Musters enthält im allgemeinen folgende Abschnitte:

- Name: Im Idealfall Charakterisierung der Auswirkungen des Musters in einem Wort.
- Zweck: Was macht das Muster? Was ist das Grundprinzip, was ist sein Zweck? Welches Problem, in welchem Umfeld macht den Einsatz des Musters sinnvoll?
- Auch bekannt als
- Struktur: Beschreibung des eigentlichen Musters, textuell und Klassen-, eventuell ein Sequenzdiagramm.
- Konsequenzen: Vor- und Nachteile, die durch den Einsatz des Musters in einem Entwurf entstehen.
- Implementierung: Der Abschnitt präsentiert die Fallen, Tipps oder Techniken, die man kennen sollten, wenn man das Muster einsetzt.

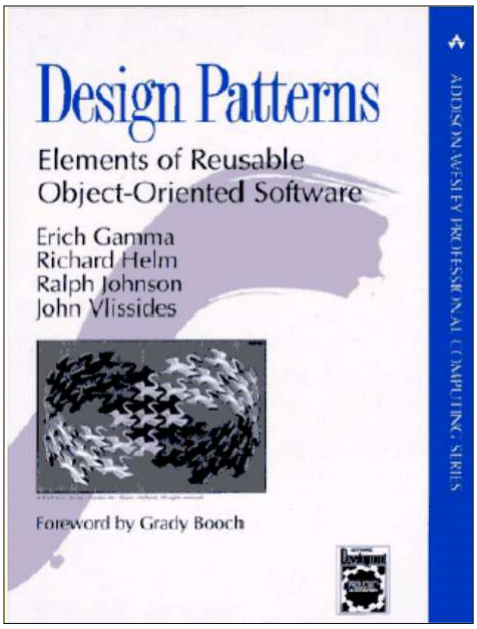
Klassifikation von Patterns

- Architektur Patterns
  - Beschreiben die grundlegende Struktur eines Softwaresystems
  - Client- Schicht, Server- Schicht und Daten- Schicht
- Design Patterns
  - Beschreiben auch Strukturen, allerdings auf einer niedrigeren Ebene als Architektur Patterns
  - Typischer Weise mit Klassen und deren Beziehung zueinander
  - Kann somit zur Ausgestaltung von Teilsystemen führen
- Idioms
  - Behandeln Detailprobleme die z.B. bei der Umsetzung von Design Pattern entstehen können
  - Implementierungsaspekte
  - Programmiersprachenspezifische Probleme (Sockets/Java)

Entwurfsmuster - Konsequenzen

- (+) Robustheit des Entwurfes
- (+) Höherer Wiederverwendungsgrad
- (+) Kommunikation
  - gemeinsames Vokabular
  - höhere Abstraktion
  - leichtere Dokumentation und Verständnis
- (-) höhere „Kosten“

Das Standardwerk



Entwurfsmuster: Gang of Four

		Aufgabe		
		Erzeugungsm.	**Strukturm.**	**Verhaltensmuster**
Gültigkeits-bereich	**klassenbasiert**	Fabrik	Adapter	Interpreter Schablonenmethode
	objektbasiert	Abstrakte Fabrik Erbauer Prototyp Singleton	Adapter Brücke Dekorierer Fassade Fliegengewicht Kompositum Proxy	Befehl Beobachter (MVC) Besucher Iterator Memento Strategie Vermittler Zustand Zuständigskette

Entwerfen von Systemen:

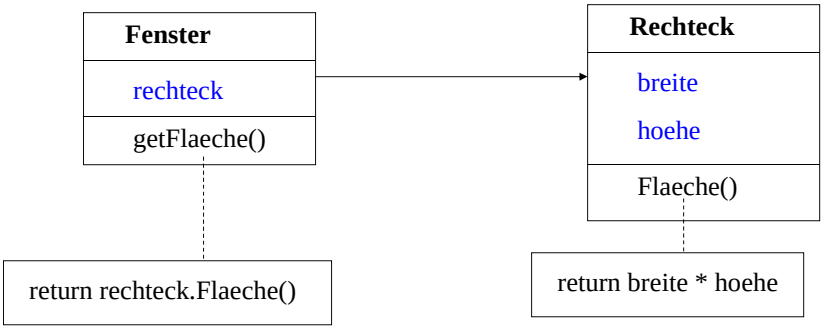
- Klassenvererbung
- Schnittstellenvererbung
- Delegation bzw.
- Komposition (Spezialfall der Delegation)
- Templates (Parametisierbare Typen)
- Klassenbibliotheken
- Frameworks

Verbung vs. Aggregation / Komposition

- Klassenvererbung nennt man auch White-Box-Wiederverwendung.
- Objektkomposition: Neue komplexe Funktionalität wird durch das Zusammenführen oder der Komposition erreicht (hat vs. ist)
- Die Komposition erwartet von Objekten, dass sie Schnittstellen der anderen Objekte respektieren.
- Es sind keine internen Details bekannt (Black-Box-Wiederverwendung).
- Klassenhierarchien bleiben klein.
- Es gibt eine größere Anzahl von Objekten. Das Verhalten hängt von ihren Beziehungen untereinander ab.
- Bei der Aggregation können die "Teile" des "Ganzen" auch einzeln existieren, bei der Komposition nur, wenn auch das "Ganze" existiert (z.B. UNummer mit Student).

Komposition

- Bei der Komposition sind zwei Objekte mit der Abarbeitung einer Anfrage beteiligt.
- Ein empfangenes Objekt delegiert Operationen an ein **Kompositionsobjekt** (Ähnlich einer Unterklassen).



Komposition:

- Der Hauptvorteil der Komposition besteht darin, dass es die Zusammensetzung von Verhalten zur Laufzeit vereinfacht.
- Das Fenster kann zur Laufzeit rechteckig oder kreisförmig sein.
- Nachteile:
  - Dynamische, hochgradig parametrische Software ist schwieriger zu verstehen.
  - Laufzeitineffzienten

Entwurfsmuster: **Klassifizierung**

1. **Aufgabe:** (was macht das Muster)

- **Erzeugungsmuster** betreffen den Prozess der Objekterzeugung.
- **Strukturmuster** befassen sich mit der Zusammensetzung von Klassen und Objekten.
- **Verhaltensmuster** charakterisieren die Art und Weise, in der Klassen und Objekte zusammenarbeiten und Zuständigkeiten aufteilen

Entwurfsmuster: **Klassifizierung**

2. Gültigkeitsbereich:

- **Klassenbasierte Muster** befassen sich mit Klassen und ihren Unterklassen. Beziehungen durch Vererbung (statisch).

- **Objektbasierte Muster** befassen sich mit Objektbeziehungen, die zur Laufzeit geändert werden können (dynamisch).

Vorgehen:

- Finden passender Objekte
- Bestimmen von Objektgranularität
- Spezifizieren von Objektschnittstellen
- Spezifizieren von Objektimplementierungen
- Wiederverwendungsmechanismen anwenden
- Strukturen der Laufzeit- und Übersetzungszeit aufeinander beziehen
- Veränderungen in Entwürfen vorhersehen

1. Beispiel: Singleton

Problem:

- **Sicherstellen, daß nur eine Instanz existiert.**
- **Allgemeiner Zugriff auf diese Instanz muss sichergestellt werden.**

Singleton, Java

```
public class Singleton {

int value = 0;

public Singleton(int value){
        this.value = value;
}


static public void main(String[] args) {
        Singleton a = new Singleton(1);
        Singleton b = new Singleton(2);
        Singleton c = new Singleton(3);
 }

}
```

Singleton, Java

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

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


Singleton, Java

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

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

Singleton, Java

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

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

       return instance;
     }
}
```


Singleton, Java, Double-Checked

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

  public static Singleton getInstance(){
       if (instance == null) {
         synchronized(Singleton.class) {
            if (instance == null) {              // ???
               instance = new Singleton();
    }
          }
       }
       return instance;
     }
}
```

Singleton, Java, Double-Checked

```
public class Singleton {
    private static Singleton instance = null;
    private boolean okay=false;
    protected Singleton(){
    }
   public static Singleton getInstance(){
        if (instance == null  || !okay) {
          synchronized(Singleton.class) {
            if (instance == null) {              // ???
              instance = new Singleton();
              okay=true;
            }
          }
        }
        return instance;
      }
}
```


Singleton, Java, Double-Checked

```
public class Singleton {
    private static Singleton instance = null;
    private static boolean okay=false;
    protected Singleton(){
    }
   public static Singleton getInstance(){
        if (!okay) {
          synchronized(Singleton.class) {
            if (instance == null) {              // ???
              instance = new Singleton();
              okay=true;
            }
          }
        }
        return instance;
      }
}
```

Singleton, Java, Double-Checked

- Das könnte doch funktionieren.
- Das Speichermodell der Java-Plattform kann zu Problemen führen.
- Die zweite Bedingung if (instance == null) kann nämlich zu true evaluieren, ohne dass „new Singleton()“ aufgerufen, und das instanziierte Objekt dem Klassenattribut instance zugewiesen wurde!
- Um das exemplarisch zu verdeutlichen, betrachtet man den folgenden Pseudo-Bytecode für den Befehl:
  - instance = new DoubleCheckedLockingSingleton() an:

Quelle:
http://www.theserverside.de/singleton-pattern-in-java/

Singleton, Java, Double-Checked

```
01.    // Speicher alloziieren
02.    ptrMemory = allocateMemory()
03.    // Den Speicher dem Klassenattribut zuweisen, ab hier gilt: instance != null
04.    assignMemory(instance, ptrMemory)
05.    // Den Konstruktor aufrufen, das Objekt ist dann ab hier korrekt instanziiert
06.    callDoubleCheckedLockingSingletonConstructor(instance)
```

- Zwischen der 4. und der 6. Zeile könnte die Ausführung des Threads durch die Java Virtual Machine unterbrochen, und die Ausführung eines zweiten Threads vorgezogen werden.
- Die zweite Bedingung if (instance == null) würde damit zu true evaluieren, ohne dass bisher der Konstruktor (Zeile 6) aufgerufen worden wäre. Das double-checked locking ist damit nicht sicher.
- Konstrukte wie diese sind in JIT-Compilern nicht unüblich und da man nicht immer voraussagen kann, wo der Code läuft sollte man das double-checked locking vermeiden, um auf der sicheren Seite zu stehen.

Quelle:
http://www.theserverside.de/singleton-pattern-in-java/

Singleton, Java: Korrekte Lösung

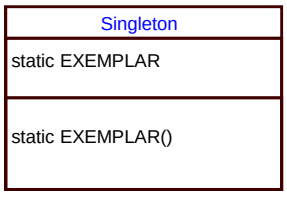
```
public class Singleton {
    private static Singleton instance = new Singleton();
    protected Singleton(){
    }

   public static Singleton getInstance(){
        return instance;
     }
}
```

Threadsicher ?

Singleton: UML

Problem:

- **Sicherstellen, daß nur eine Instanz existiert.**
- **Allgemeiner Zugriff auf diese Instanz muss sichergestellt werden.**



Singleton

Zweck:

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

Motivation:

- Eindeutiger Druckerspooler, ein Dateisystem, ein Fenstersystem
- Globale Variable ermöglicht Zugriff auf das Objekt, eindeutig?
- Einzigartigkeit in der Klasse abgefangen

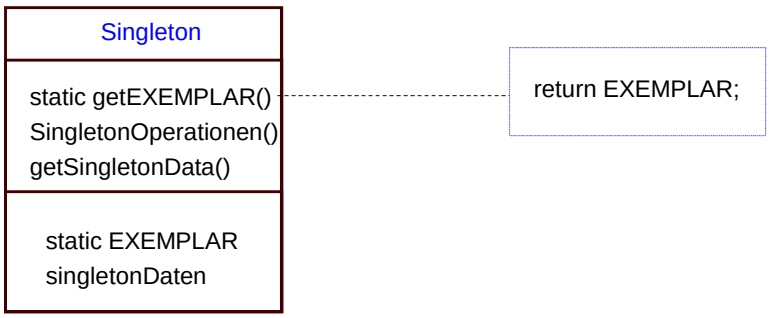
Anwendbarkeit:

Verwenden Sie das Singletonmuster, wenn

- es genau ein Exemplar einer Klasse geben muss und der Zugriff eindeutig sein soll.
- das einzige Exemplar durch Bildung von Unterklassen erweiterbar sein soll. Anwender sollen die Erweiterungen ohne Modifikation weiter verwenden.

Singleton mit UML-Notation

Struktur:



Singleton

Teilnehmer:
- definiert eine Exemplaroperation, die es Klienten ermöglicht, auf sein einziges Exemplar zuzugreifen.
- ist potenziell für die Erzeugung seines einzigen Exemplars zuständig.

Interaktion:
- Klienten greifen ausschließlich durch die getExemplar-Operation der Singletonklasse zu.

Konsequenzen / Vorteile:
- *Zugriffskontrolle auf das Exemplar.* Aus der Kapselung des Konstruktors folgt die strikte Kontrolle.
- *Kleinerer Namensraum.* Das Singletonmuster ist eine Verbesserung gegenüber globalen Variablen

Singleton

Konsequenzen / Vorteile:
- *Verfeinerung von Operationen und Repräsentation*. Die Klasse kann abgeleitet und spezialisiert werden. Die Konfiguration zur Laufzeit ist möglich.
- *Variable Anzahl von Exemplaren*. Änderung der Methode getExemplar() ist einem Konstruktor stellt den Normalzustand ohne weitere Codeänderung her.

Implementierung:
- *Problem Ableitung neuer Klassen.*
- Damit können mehrere Instanzen existieren.
- Problem Serialize

Singleton mit enums, Java

```
enum Singleton {
    INSTANCE;
  public int i=33;
  private Singleton() {
    this(0);
  }
  private Singleton(int i) {
    this.i = i;
  }
}  // Singleton6
```

- Der Vorteil dieser Variante ist, dass sie selbst aufwändigen Manipulationsversuchen durch Serialisierung und/oder Introspektion um eine Mehrfachinstanziierung zu ermöglichen widersteht.
- Es ist die derzeit beste Methode, Singletons zu implementieren.
- Der Nachteil ist jedoch, dass diese Methode erst ab Java Version 1.5 funktioniert.

Singleton mit enums, Java

```
static public void main(String[] args) {
    System.out.println(" ");

    Singleton  a  = Singleton.INSTANCE;
    System.out.println("a: "+ a.getValue()  );
    a.setValue(2222);
    System.out.println(" ");

    Singleton  b = Singleton.INSTANCE;
    b.setValue(1111);
    System.out.println("a: "+ a.getValue()  );
    System.out.println("b: "+ b.getValue()  );
}
```

Singleton, C#: Beispiel SingletonA

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Singleton1 {

   sealed class Singleton {
      private static Singleton instance = new Singleton();
      protected Singleton() {
      }

      public static Singleton getInstance() {
         return instance;
      }

   }
}
```


Simple Singleton in C#

```
sealed class Singleton {

    private Singleton() {
    }

   public static readonly Singleton Instance = new Singleton();

}
```

Singleton, C#: Beispiel SingletonB

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Singleton1 {

   sealed class Singleton {
      private static Singleton instance = new Singleton();
      protected Singleton() {
      }

          // Eigenschafts-Notation          public int Std {get; private set;}
       public static readonly SingletonB Instance = new SingletonB();

   }
}
```


Double-Check Lock in C#

```
sealed class Singleton {
   private static volatile Singleton _instance = null;

   protected Singleton() {
   }

   public static Singleton Instance() {
      if (_instance == null) {
         lock (typeof(Singleton)) {
            if (_instance == null) {
               _instance = new Singleton();
            }
         }
      }
      return _instance;
   }
}
```

Double-Check Lock in C#

```
sealed class Singleton {
   private static volatile Singleton _instance = null;
   private bool okay=false;

   protected Singleton() {
   }

   public static Singleton Instance() {
      if (!okay)    // _instance == null) {
         lock (typeof(Singleton)) {
            if (_instance == null) {
               _instance = new Singleton();
               okay=true;
            }
         }
      }
      return _instance;
   }
}
```


Singleton mit Lazy Creation

- In den vorherigen Lösungen werden die Instanzen immer sofort beim Starten erzeugt.
- Bei größeren Klassen bzw. bei Klassen, deren Instanz man nicht immer benötigt, wäre eine spätere Instanziierung sinnvoller.
- Diese Technik heißt „lazy creation“
- In Java wäre das das Beispiel „Singleton mit Double-Checked“

Singleton mit Lazy Creation in C#: Beispiel SingletonC

```
class SingletonC {

    protected SingletonC() {    }


   class SingletonCreator {
     static SingletonCreator() {   }

        // internal besser als private
      internal static readonly SingletonC instance = new SingletonC();
    }

     public static SingletonC Instance {
        get { return SingletonCreator.instance; }
     }

  }
```


volatile vs. synchronized

- Bei Multithreading-Anwendungen müssen Variablen gegenüber dem gleichzeigen Schreiben gesichert werden.
- Dazu existieren die beiden Techniken „volatile“ und „synchronized“.
- **synchronized:**
  - Mit „synchronized“ sichert man den Zugriff auf eine Methode Thread-sicher ab.
  - Die Änderung wird aber nicht automatisch weitergeleitet.

- **volatile:**
  - Mithilfe des Schlüsselworts volatile lässt sich der Zugriff auf eine Variable synchronisieren. Ist eine Variable als volatile deklariert, muss die JVM sicherstellen, dass alle zugreifenden Threads ihre Kopien aktualisieren, sobald die Variable geändert wird.
  - Eine Sperre à la „lock“ ist nicht vorgesehen.
  - Aber nun wird beim jedem Zugriff ein Broadcast gesendet.
  - Das volatile Increment bzw. Decrement ist nicht threadsicher.

Volatile und Increment resp. Decrement

Seit Java 5 helfen hier die Klassen des Pakets java.util.concurrent.atomic. Diese besitzen die geeigneten Methoden.

Klassen:
- AtomicBoolean
- AtomicInteger
- AtomicLong

Felder:
- AtomicIntegerArray
- AtomicLongArray.

Reflection an Attribute und Referenzen:
- AtomicIntegerFieldUpdater<T>
- AtomicLongFieldUpdater<T>
- AtomicReference
- AtomicReferenceArray<E>
- AtomicReferenceFieldUpdater<T,V>
- AtomicMarkableReference<V>
- AtomicStampedReference<V>

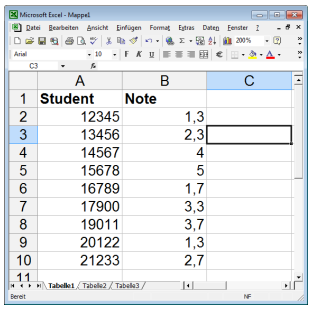
Volatile und Increment resp. Decrement

```
public class Id  {

   private static final AtomicLong id = new AtomicLong();

   private Id() {
    }

   public long next() {
     return id.getAndIncrement();
   }

}
```

Quelle: com/tutego/insel/thread/atomic/Id.java

MDI-Fenster:

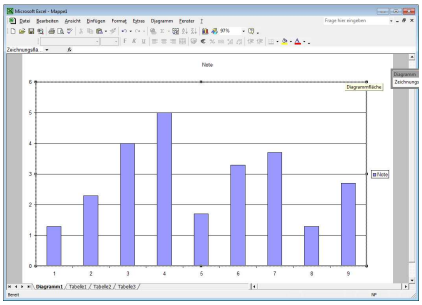
Daten in einer Tabelle

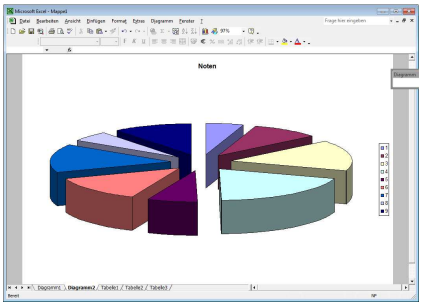


	A	B	C
1	Student	Note	
2	12345	1,3	
3	13456	2,3	
4	14567	4	
5	15678	5	
6	16789	1,7	
7	17900	3,3	
8	19011	3,7	
9	20122	1,3	
10	21233	2,7	
11			

Problem:
Datenaktualisierung

Daten in einer Grafik





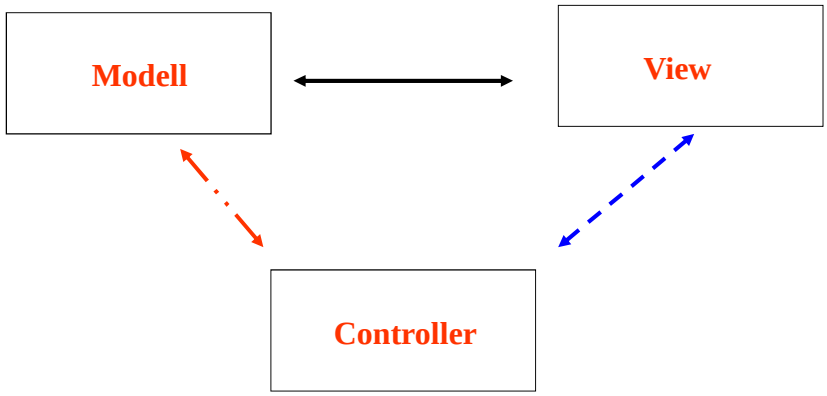
2. Beispiel: Observer Pattern

Problem:

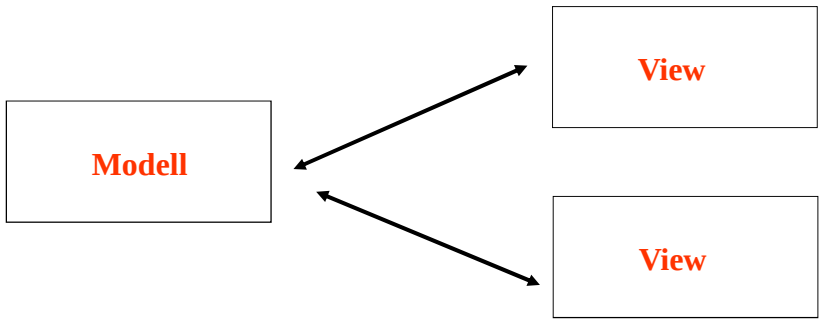
Mehrere Objekte einer Gruppe müssen sich benachrichtigen, wenn einige Attribute geändert wurden. Die Viewer sind nicht „bekannt“.

Smalltalk:	Model-View-Controller
MFC	Dokument-View-Architektur
Java:	Model-View-Architektur

Beispiel: Observer Pattern MVC



Beispiel: Observer Pattern MVC



Beispiel: Observer Pattern MVC

- Das Model kann man sich als das Datenmodell (Datei, Datenbank) vorstellen, der den aktuellen Zustand und das Verhalten des gesamten Systems repräsentiert (Datenhaltung, Anwendung)
- Der View hat die Aufgabe die Daten des Models auf irgend eine Art darzustellen. Dabei hat der View nur die Aufgabe, die Daten darzustellen (Visualisierung, Darstellung)
- Der Controller setzt die eingehenden Anforderungen (z.B. Eingaben von der Tastatur oder Maus) in Methoden um, die das Model dazu veranlassen, die Daten entsprechen zu verändern (Datenmanipulation, Steuerung)

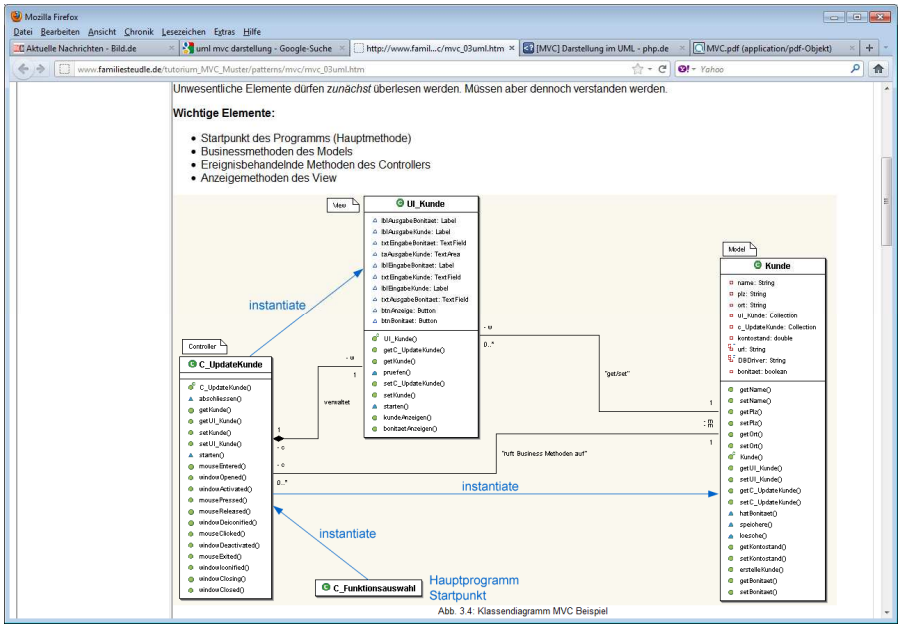
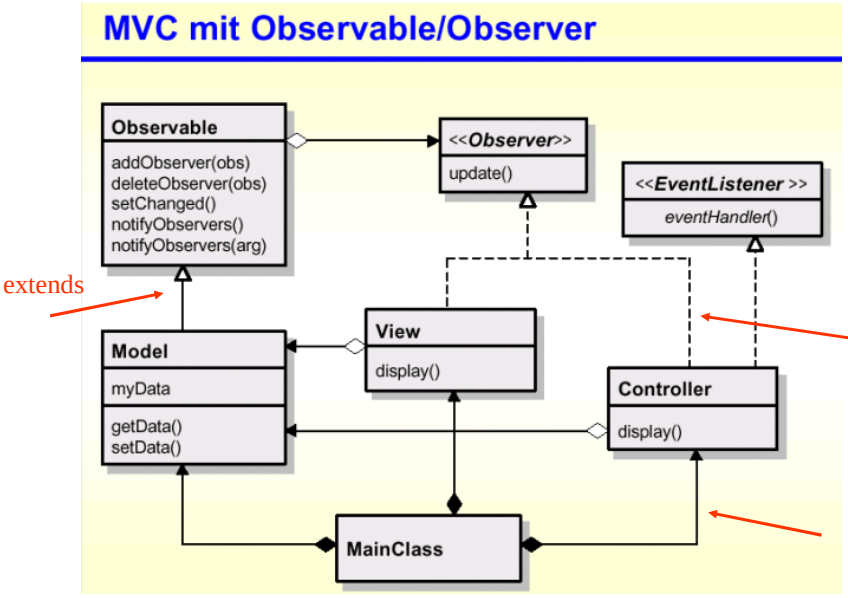
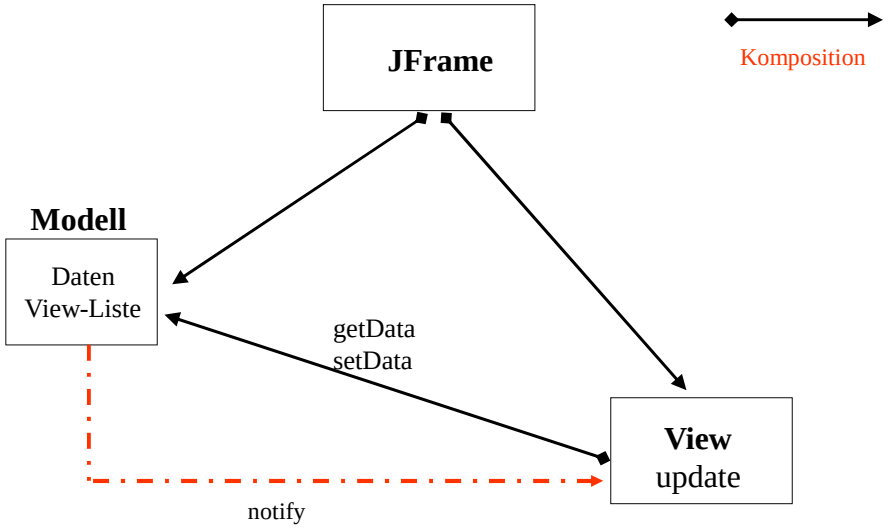


Abb. 3.4: Klassendiagramm MVC Beispiel

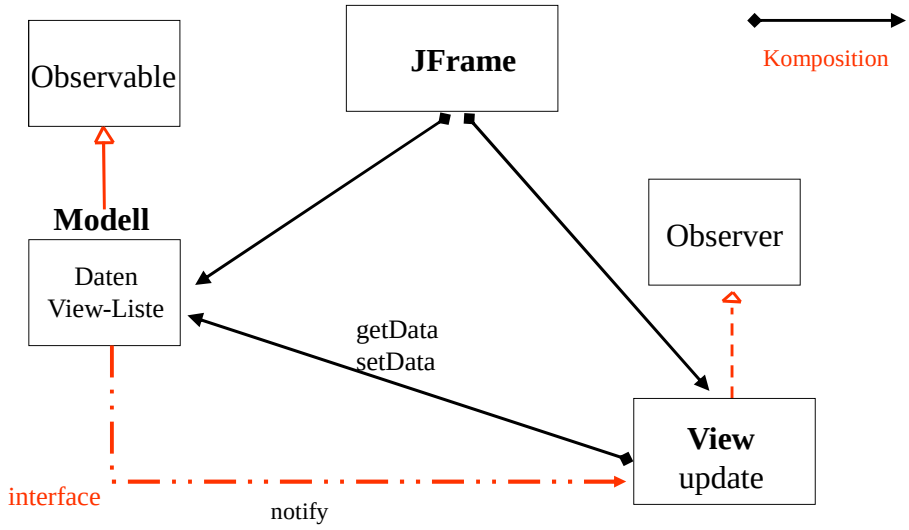
MVC mit Observable/Observer



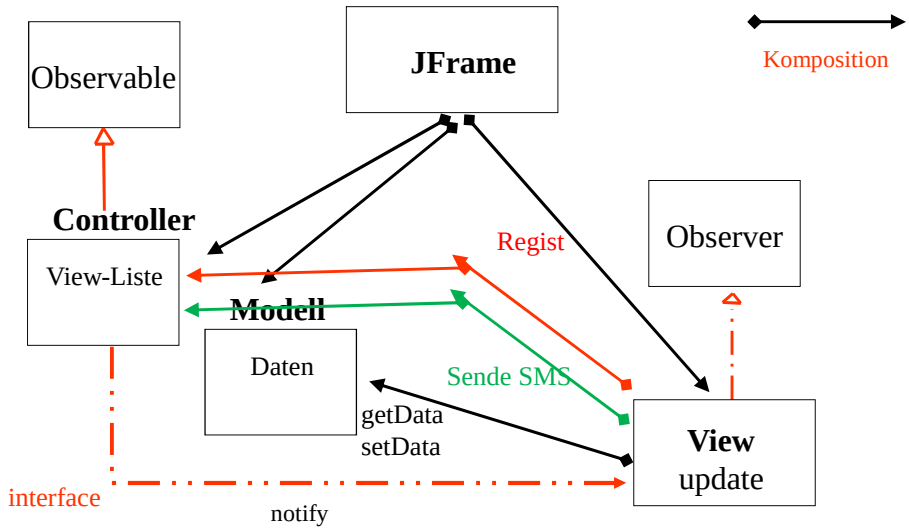
Observer Pattern (MVC ohne Controller)



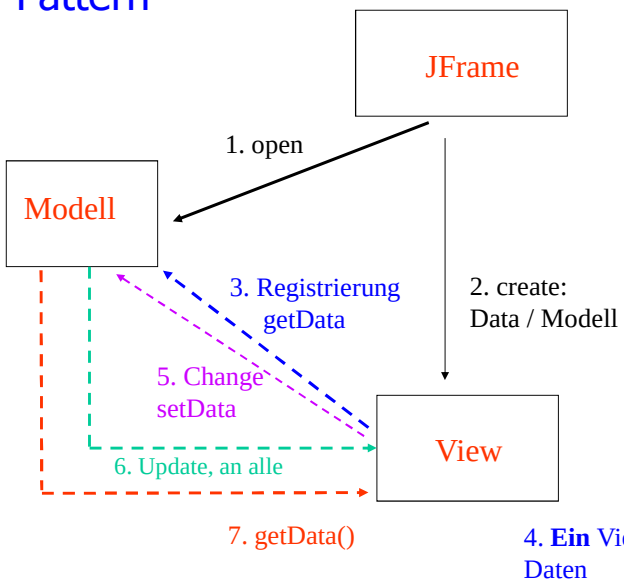
Observer Pattern (MVC ohne Controller)



Observer Pattern (MVC ohne Controller)



Observer Pattern



Bemerkungen zum Observer Pattern

- Entkopplung
  - Subjekt kennt nur Observer-Interface, keine konkreten Observer
  - update = dynamisch gebunden => upcalls
- Falls Observer von mehreren Subjekten abhängen sollen:
  - update(Observable o)
- Falls Art der Änderung übergeben werden soll: push vs pull
  - notifyObservers(Object arg)
  - update(Observable o, Object arg)
- Wann wird notify aufgerufen?
  - Zustandsänderungsfunktion => i.a. viele updates
  - Verzögert & explizit: changed-Flag: setChanged / clearChanged
    - * notifyObservers wird nur aktiv wenn isChanged() == true
    - * notifyObservers ruft clearChanged auf

Bemerkungen zum Observer Pattern

- Kausalität der Änderungen
  - Verbieten von Änderungen während update
  - Queueing-Mechanismus
- Multithreading
  - Während der Meldung „notify“ kann add/remove des Observers aufgerufen werden. Die Listenerliste wird geändert (InvalidStateException bei Iterator)
    * synchronized(this){
      for(int i=0; i<observers.size(); i++){…..
    * Vector v;
      synchronized(this){v = (Vector)observers.clone();}
      for(int i=0; i<v.size(); i++){…
- Interfaces vs. Classes

Java-Beispiel: Main-Frame

```
public class MVC extends JFrame {
   private Modell modell;

 public MVC() {
   modell = new Modell();
   modell.setName("Paul");
}

 void MnNew_actionPerformed(ActionEvent e) {
   MyInternalFrame1 frame = new MyInternalFrame1(modell);

   frame.setBounds( 10, 10, 250, 50);
   _desktopPane.add(frame);
  }

}
```

Java-Beispiel: Anlegen eines „Datenmodells“

```
class Modell  extends java.util.Observable {
  String sName= "Freya";

  public String getName() {
    return sName;
  }

  public void setName(String Name) {
    this.sName = Name;
  }

  public void DataChangedFromViewer(String s) {
    System.out.println("s in myData: "+s);
    this.setName(s);
    super.setChanged();
    super.notifyObservers();
  }
} // Modell
```


Java-Beispiel: Clientfenster

```
class MyInternalFrame extends JInternalFrame implements Observer {
 private Modell modell;

 public MyInternalFrame(Observable DatenModell) {

   modell = (Modell) DatenModell;
   DatenModell.addObserver(this);  // Registrierung
   edit.setText(    modell.getName()     );
}
```

Java-Beispiel: Clientfenster2

```
public void setGUI()  {
  edit.addActionListener(new java.awt.event.ActionListener() {
     //  Aufgerufen wenn Return-Taste
    public void actionPerformed(ActionEvent e) {
      edit_Change(e);
    }
  });
 }  // setGUI

 public void update(Observable o, Object arg) {
  System.out.println("Update");
  edit.setText( modell.getName() );
 }

 void edit_Change(ActionEvent e) {
  System.out.println("geändert");
  modell.DataChangedFromViewer( edit.getText()  );
 }
```


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.

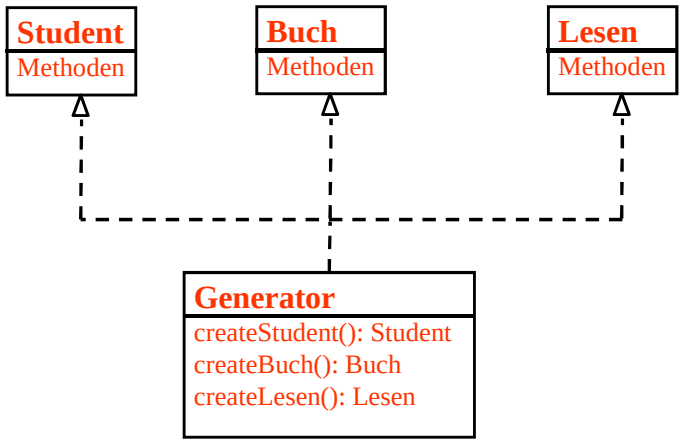
Einstieg-Beispiel

Student will Bücher lesen:

Person: Student
Objekt:Buch
Aktion: Buch lesen

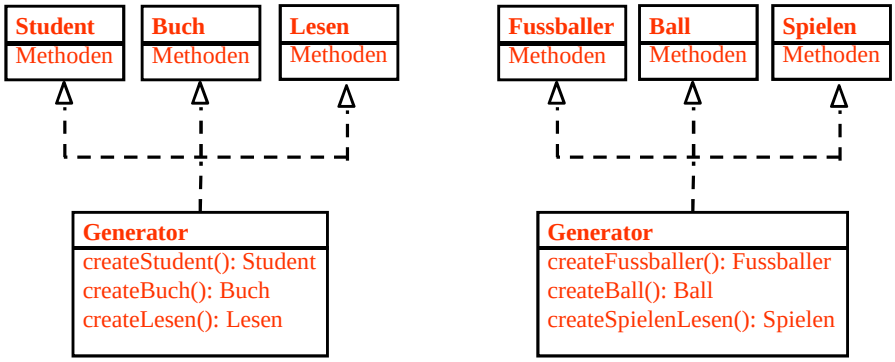
Benutzt wird ein Generator zum Erzeugen der drei Objekte:

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



Eigenschaften

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.

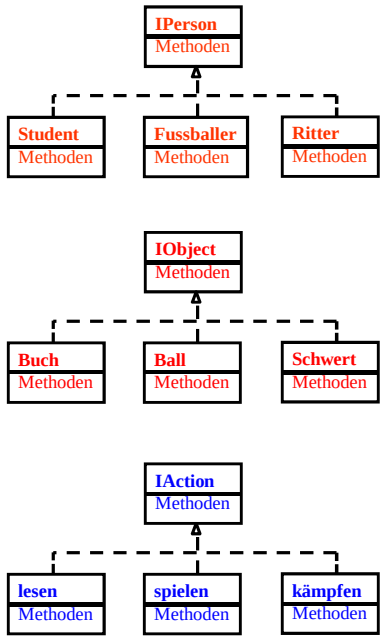
Keine Allgemeingültigkeit.
Der Programmierer kann keinen **allgemeingültigen** Code schreiben. Hat er Code für das „Lesen“ geschrieben und will nun die Fussballer programmieren, so muss sein Code komplett umschreiben (ein Wartungsalptraum).

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.

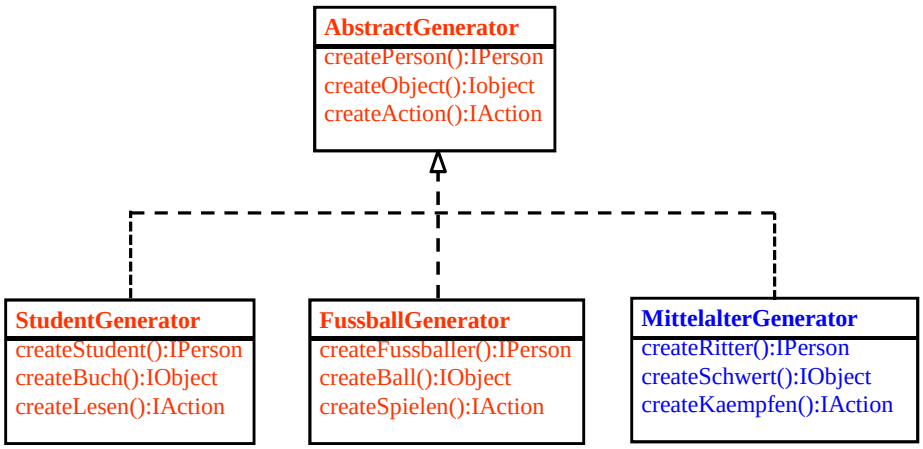
Abhilfe: Interfaces

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

Abhilfe:
Definition eines abstrakten Generators
Einbau von drei Generatoren



Interface: AbstractGenerator



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.

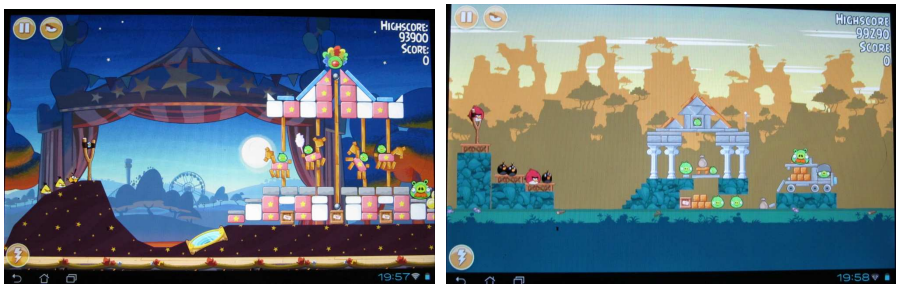
Konsistenz.

Die zusammengefassten Generatoren stellen sicher, dass nur Objekte erstellt werden, die auch zueinander passen und miteinander funktionieren.

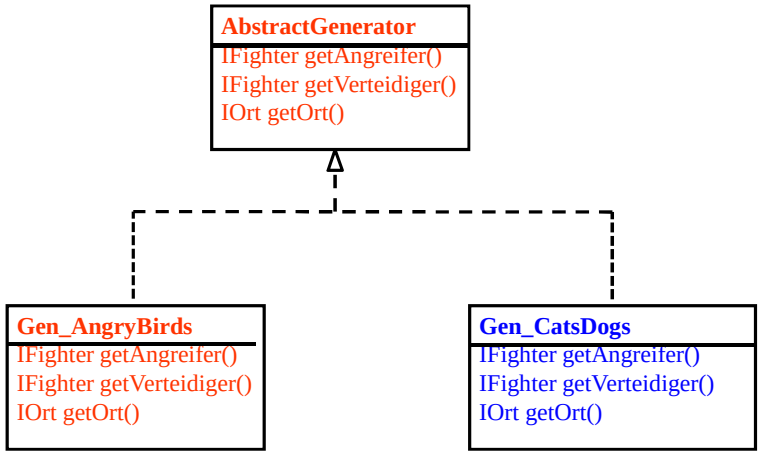
Mit einem Singleton kann man nun einen neue Fabrik erstellen.

Diese Fabrik kapselt dann die drei Fabriken. Damit kann ein Anwender nicht zwei Generatoren erzeugen.

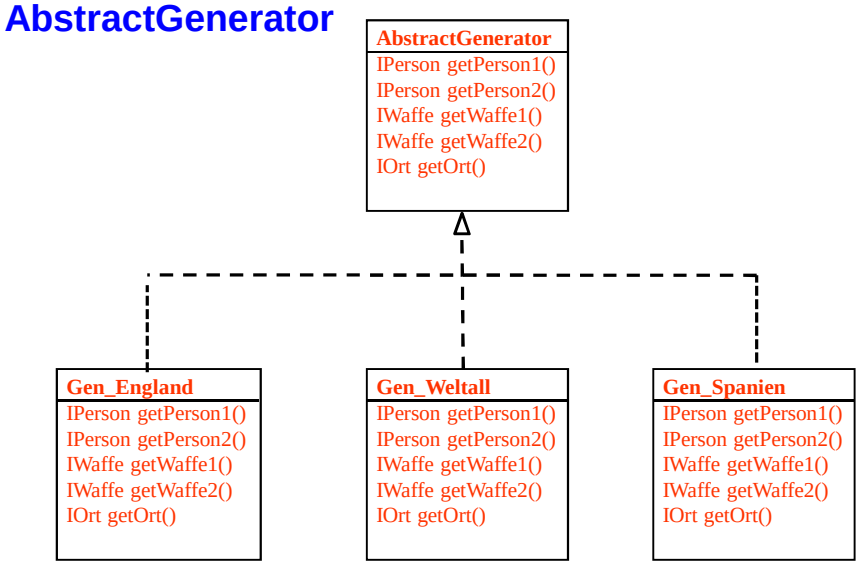
Anwendung: Angry Birds



AbstractGenerator



AbstractGenerator



Entwurfsmuster Dekorator

- **Absicht**
  - Stattet ein Objekt dynamisch mit zusätzlichen Eigenschaften aus. Ein Dekorator bietet eine flexible Alternative zu Subklassen.
- **Motivation**
  - Des Öfteren möchte man ein Objekt, und nicht eine ganze Klasse, um einige Eigenschaften erweitern. So sollte es eine grafische Benutzeroberfläche gestatten einem Element individuelle Eigenschaften, wie Rollbalken oder einen Rahmen hinzuzufügen.
  - Dies könnte mittels Vererbung geschehen: Dies platziert etwa einen Rahmen um jede Instanz der entsprechenden Subklasse. Dies ist nicht sehr flexibel, da hier die Wahl des Rahmens statisch vorgenommen wird, d.h. ein Klient kann nicht beeinflussen wie und wann eine Komponente mit einem Rahmen versehen wird.

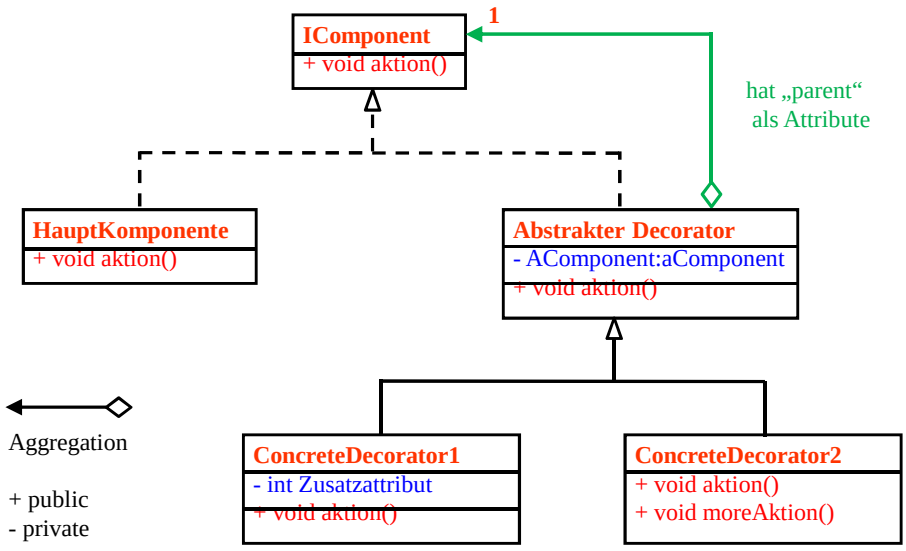
Entwurfsmuster Dekorator

- **Motivation (Forts.)**
  - Eine flexiblere Lösung ist die Komponente in ein anderes Objekt einzubetten. Dieses neue Objekt fügt den Rahmen hinzu. Ein derartiges umschließendes Objekt heisst Dekorator. Der Dekorator stellt die Schnittstelle der ursprünglichen Komponente zur Verfügung, daher ist seine Präsenz für die Klienten der Komponente nicht sichtbar.
  - Der Dekorator leitet Anforderungen an die Komponente weiter und führt optional zusätzliche Aufgaben aus (mehr Arbeit).
  - Dekoratoren einer Komponente können beliebig tief verschachtelt werden. So kann einer Textkomponente ein Rollbalken und dem neuen Objekt ein Rahmen hinzugefügt werden.

Entwurfsmuster Dekorator

- **Anwendbarkeit**
  - Dynamisches und transparentes Hinzufügen von Verantwortlichkeiten zu einem Objekt ohne andere Objekte zu beeinflussen.
  - Eine zu große Erweiterung durch Subklassen ist inpraktikabel, da diese zeitintensiv sind.
  - Eine Klassendefinition kann auch nicht zur Verfügung stehen oder nicht möglich sein (Klasse ist als final deklariert).

Struktur des Entwurfsmuster Dekorator



Entwurfsmuster Dekorator

- **Teilnehmer**
  - IComponent
    - definiert die Schnittstelle für Objekte die dynamisch erweitert werden können.
  - ConcreteComponent
    - definiert ein Objekt das für zusätzliche Eigenschaften verantwortlich ist.
  - Decorator
    - unterhält eine **Referenz** auf eine IComponent und definiert eine IComponent entsprechende Schnittstelle.
  - ConcreteDecorator 1/2
    - fügt IComponent neue Eigenschaften hinzu

Dekorator

- **Beispiel**

```
interface IComponent {
  public void doStuff();
}
class ConcreteComponent implements IComponent {
  public void doStuff() { }
  public void doMoreStuff() { }
}
abstract class Decorator implements IComponent {
  private Component comp;
  public Decorator (Component c) { comp = c ; }
  public void doStuff() { comp.doStuff(); }
}

public class ConcreteDecorator extends Decorator {
  public void doEvenMoreStuff() { }
}
```

Dekorator

- **Zusammenarbeit**
  - Ein Dekorator leitet Anforderungen an seine IComponent weiter.
  - Zusätzlich kann er andere Operationen vor oder nach einer solchen Weiterleitung ausführen.

Entwurfsmuster Dekorator

- **Konsequenzen: Vorteile**
  - Grössere Flexibilität als Vererbung. Das Dekorator-Muster stellt eine flexiblere Methode einem Objekt zusätzliche Eigenschaften zu geben als (Mehrfach-)Vererbung.
  - Mittels Dekoratoren können Eigenschaften einfach **zur Laufzeit** hinzugefügt und auch wieder zurückgenommen werden.
  - Im Gegensatz erfordert Vererbung die Erzeugung einer neuen Klasse für jede neue unabhängige Eigenschaft. Dies führt zu vielen Klassen und entsprechender Komplexität.
  - Durch verschiedene Dekoratoren für eine Komponente können mehrere Eigenschaften nach Bedarf hinzugefügt werden. Zudem erlauben es Dekoratoren, im Gegensatz zur Vererbung, einfach eine Eigenschaft mehrmals hinzuzufügen.

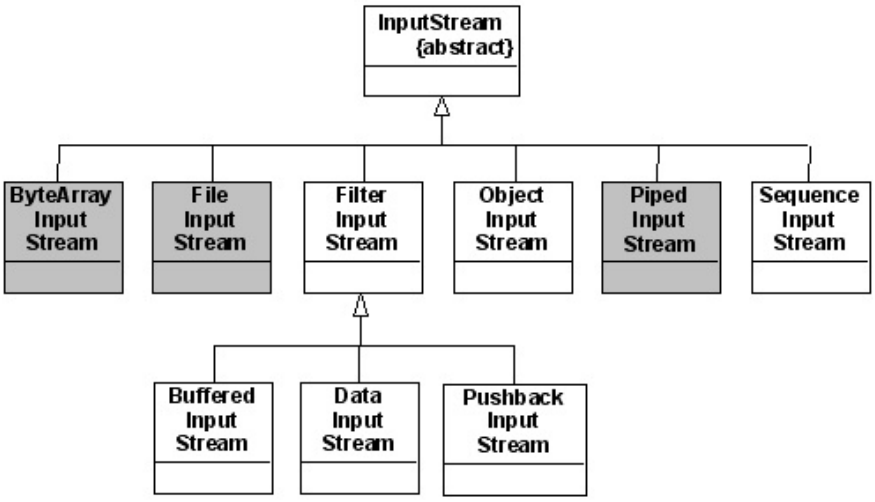
Entwurfsmuster Dekorator

- **Konsequenzen: Vorteile (Forts.)**
  - Vermeidet mit Eigenschaften überladene Klassen (fat classes) nahe der Hierarchiewurzel. **Statt alle möglichen Eigenschaften in einer komplexen, anpassbaren Klasse vorherzusehen**, wird eine einfache Klasse definiert und diese inkrementell mit Dekoratoren erweitert.
  - Dies erlaubt Funktionalität mit einfachen Einzelbausteinen zu erzeugen. Daher entstehen einer Anwendung keine Kosten für ungenutzte Eigenschaften.
  - Dekoratoren erlauben eine einfache und unabhängige Erweiterung einer Klasse, während neue Subklassen dazu neigen für die neue Eigenschaft unwichtige Einzelheiten offen zu legen.

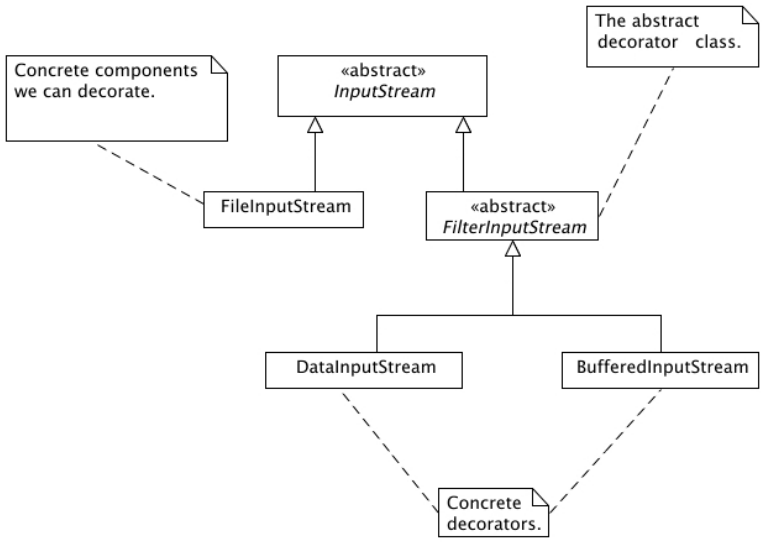
Entwurfsmuster Dekorator

- **Konsequenzen: Nachteile**
  - Ein Dekorator und seine Component sind nicht identisch. Ein Dekorator agiert als transparente Hülle. Vom Objektstandpunkt ist die dekorierte Komponente nicht identisch mit der Komponente. Daher sollten Abfragen nach Objekt-Identität bei der Benutzung von Dekorator vermieden werden.
  - Objektinflation. Ein Dekorator-basiertes Design resultiert oft in Systemen mit vielen, nahezu gleich aussehenden Objekten. Solche Systeme sind für Eingeweihte leicht an unterschiedliche Erfordernisse anzupassen, für andere aber schwer zu lernen und von Fehlern zu befreien.: fout etc.

Java I/O als Beispiel für einen Dekorator



Beispiel eines Dekorator-Pattern mit I/O



Dekoratorbeispiel Pizza (1)

Abbilden einer Pizza-Bestellung

Attribute:
int radius;

Zutaten:
Käse
Salami
Huhn
Ei

Preis:
Bestimmen des Gesamtpreises

Dekoratorbeispiel Pizza (2)

```
class Pizza {
  public int radius;
  public boolean kaese=false;
  public boolean salami=false;
  public boolean chicken=false;

  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Radius: "+radius
        +"\n  Kase: "+kaese
        +"\n  Salami: "+salami
        +"\n  Hühnchen: "+chicken+"\n\n";
  }

} // Pizza
```

Aufruf:

```
Pizza p1 = new Pizza(20);
p1.kaese=true;
p1.ei=true;
System.out.println("p1: "+p1);

 Pizza p2 = new Pizza(32);
 p2.kaese=true;
 p2.chicken=true;
 p2.salami=true;
 System.out.println("p2: "+p2);
```

Eigenschaft, aber keine Aktionen !!

viele Abfragen, bei der Bestimmung des Gesamtpreises

pizza1.java

Dekoratorbeispiel Pizza (3): Ableiten von Pizza

```
class Pizza {
  public int radius;
  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Pizza: Radius: "+radius+"\n";
  }
}  // Pizza

class PizzaKaese extends Pizza {
  public PizzaKaese(int r) {
    super(r);
  }
  public String toString()  {
    return "PizzaKaese: "+super.toString()+"\n";
  }
}  // PizzaKaese
```

```
class PizzaSalami extends Pizza {
  public PizzaSalami(int r) {
    super(r);
  }
  public String toString()  {
    return "PizzaSalami: "+super.toString()+"\n";
  }
}  // PizzaSalami
```

jetzt Aktionen !!

Aber man darf nur eine Zutat auswählen

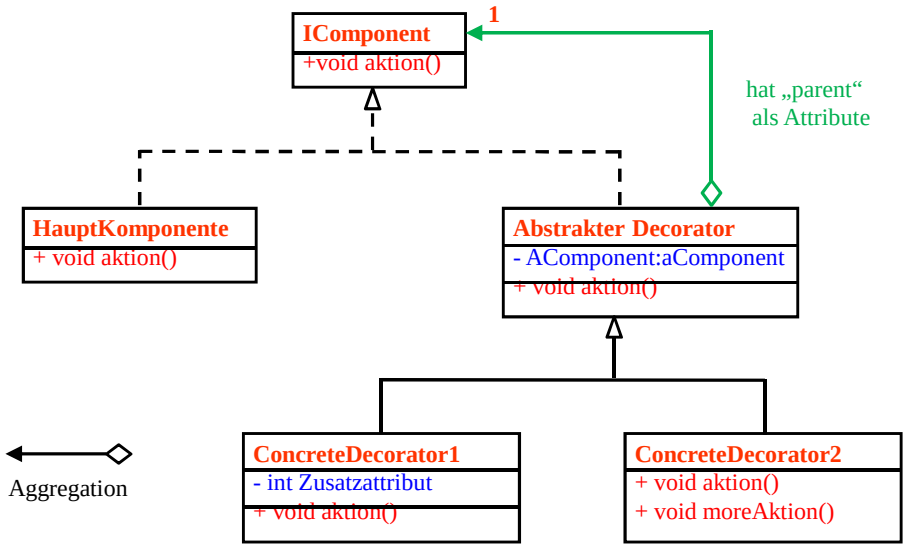
Dekoratorbeispiel Pizza (4): Verschachtelte Ableitung

```
class Pizza {
  public int radius;
  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Pizza: Radius: "+radius+"\n";
  }
}  // Pizza

class PizzaKaese extends Pizza {
  public PizzaKaese(int r) {
    super(r);
  }
  public String toString()  {
    return "PizzaKaese: "+super.toString()+"\n";
  }
}  // PizzaKaese
```

```
class PizzaSalami extends PizzaKaese {
  public PizzaSalami(int r) {
    super(r);
  }
  public String toString()  {
    return "PizzaSalami: "+super.toString()+"\n";
  }
}  // PizzaSalami
```

jetzt Aktionen !!
Man darf mehrere eine Zutaten auswählen
Alle sind an oder aus
Bool‘sche Variable

Struktur des Entwurfsmuster Dekorator



Dekoratorbeispiel Pizza (5): Interface IPizza

```
private void   bnTest1_click() {
    Pizza p1 = new Pizza(20);
    System.out.println("p1: "+p1.backen());
    System.out.println("  Preis: "+p1.getPreis() );

    PizzaKaese p2 = new PizzaKaese(p1);
    System.out.println("p2: "+p2.backen());
    System.out.println("  Preis: "+p2.getPreis() );
```

```
interface IPizzabacken {
  public String backen();
  public double getPreis();
}
```

Ergebnis:

```
p1: Pizza: Radius: 20
  Preis: 6.78

p2: Pizza: Radius: 20
  PizzaKaese:
  Preis: 8.120000000000001
```


pizza4.java

Dekoratorbeispiel Pizza (5): Interface IPizza

Test1:

p1: Pizza: Radius: 20
Preis: 6,78

p2: Pizza: Radius: 20
PizzaKaese:
Preis: 8,12

p3: Pizza: Radius: 20
PizzaKaese:
PizzaSalami:
Preis: 10,78

p4: Pizza: Radius: 20
PizzaKaese:
PizzaSalami:
PizzaChicken:
Preis: 13,23

Preise:

Pizza:	6,78;
Nudeln:	7.80;
Käse:	+ 1.50;
Salami:	+ 2.50;
Schinken	+ 2.75;
Tomaten:	+ 0.56;

```
import java.text.*;
DecimalFormat df;
df  =   new DecimalFormat  ( ",##0.00" );
Syso("   Preis: "+df.format(b3.getPreis()) + " Euro");
```

df

Dekoratorbeispiel Pizza (6): Interface IPizza

```
class Pizza implements IPizzabacken {
  public int radius;
  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Pizza: Radius: "+radius+"\n";
  }
  public String backen() {
   return toString();
  }
  public double getPreis() {
    return 6.78;
  }
}  // Pizza
```

```
class PizzaKaese implements IPizzabacken {
  private IPizzabacken p;
  public PizzaKaese(IPizzabacken p) {
    this.p = p;
  }
  public String toString()  {
    return "PizzaKaese: "+"\n";
  }
  public String backen() {
   return p.backen()+"   "+toString();
  }
  public double getPreis() {
    return 1.34+p.getPreis();
  }
}  // PizzaKaese
```


pizza4.java

Dekoratorbeispiel Pizza (7): IPizza, Beilage

```
interface IPizzahaus {
  public String backen();
  public double getPreis();
}



// vereinfacht die Erstellung der Beilagen
abstract class Beilage implements IPizzahaus {
    protected IPizzahaus p;

    public Beilage(IPizzahaus p) {
        this.p=p;
    }
}
```


Dekoratorbeispiel Pizza (8): IPizza, Beilage

```
// Hauptgerichte
class Pizza implements IPizzahaus {
  public int radius;
  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Pizza: Radius: "+radius+"\n";
  }
  public String backen() {
   return toString();
  }
  public double getPreis() {
    return 4.80;
  }
}  // Pizza
```

```
// Hauptgerichte
class Nudeln implements IPizzahaus {
  public double menge;
  public Nudeln(double menge) {
    this.menge = menge;
  }
  public String toString()  {
    return "Nudeln: Menge: "+menge+"\n";
  }
  public String backen() {
   return toString();
  }
  public double getPreis() {
    return 7.80;
  }
}  // Nudeln
```

Dekoratorbeispiel Pizza (9): IPizza, Beilage

```
abstract class Beilage implements IPizzahaus {
   protected IPizzahaus p;

   public Beilage(IPizzahaus p) {
      this.p=p;
   }
}
```

```
class Kaese extends Beilage  {
  public Kaese(IPizzahaus p) {
    super(p);
  }
  public String toString()  {
    return "Zusatz Kaese: "+"\n";
  }
  public String backen() {
   return p.backen()+"  "+toString();
  }
  public double getPreis() {
    return p.getPreis() + 1.50;
  }
} // Kaese
```

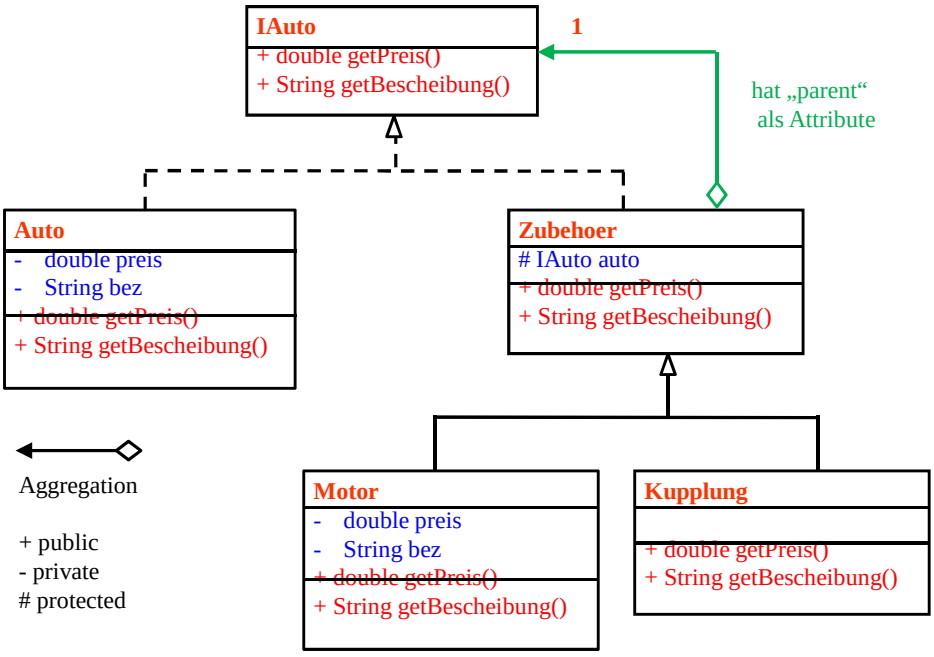

Dekoratorbeispiel Pizza (5): Interface IPizza

```
Test1:
mit 1. Beilage:        Pizza: Radius: 15
 --- Zusatz Kaese:
    fuer 6,30 Euro
Test2:
mit 1. Beilage: Pizza: Radius: 18
 --- Zusatz Kaese:
mit 2. Beilage: Pizza: Radius: 18
 --- Zusatz Kaese:
  Zusatz Salami:
    fuer 9,05 Euro
mit 3. Beilage: Pizza: Radius: 18
 --- Zusatz Kaese:
  Zusatz Salami:
  Zusatz Salami:
    fuer 9,61 Euro
```

Preise:

Pizza:	4.80;
Nudeln:	7.80;
Käse:	+ 1.50;
Salami:	+ 2.50;
Schinken	+ 2.75;
Tomaten:	+ 0.56;

```
import java.text.*;
DecimalFormat df;
df =  new DecimalFormat  ( ",##0.00" );
Syso("  Preis: "+df.format(b3.getPreis()) + " Euro");
```



Dekoratorbeispiel (Stream Decorator Implementierung)

```
import java.io.*;  // eigene Klasse
public class CharUpperInputStream extends FilterInputStream {

        public CharUpperInputStream(InputStream is) {
          super(is);
        }

        public int read() throws IOException {
          int c = super.read();
          return Character.toUpperCase(c);
        }
}
```

Dekorator Beispiel

```
import java.io.*;
public class CharUpperInputStream {
        private void read(String sFilename) throws IOException {
          InputStream is = new FileInputStream (sFilename);
          InputStream bis = new BufferedInputStream (is);
          InputStream cuis = new CharUpperInputStream( bis );
          int c ;
          while (( c = cuis.read() ) >= 0) {
                System.out.print((char) c);
          }
        }
}
```


Iterator

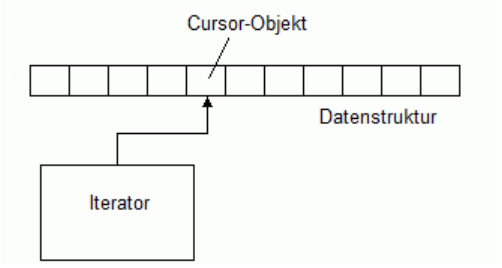
- **Absicht**
  - Oft enthält eine Klasse mehrere Elemente desselben Typs (eine Aggregation). Ein Iterator erlaubt es auf diese Elemente sequentiell zuzugreifen ohne Kenntnis darüber zu haben, wie die Elemente genau in der Klasse gespeichert sind.
- **Alias**
  - Cursor in Datenbanken (next, prev, first, last)

Iterator

- **Motivation**
  - Die wesentliche Idee ist die Traversierung der Elemente eines Aggregates vom Aggregat selbst zu separieren.
  - Die Traversierung wird in einem Iteratorobjekt implementiert.
  - Der Iterator stellt eine **allgemeine Schnittstelle** zum Zugriff auf die Elemente des Aggregats zur Verfügung.

Iterator

- Ein Iterator ist ein Objekt, das eine bestimmte Datenstruktur sequenziell durchläuft.
- Mit jedem Aufruf seiner Methode ***next*** liefert der Iterator jeweils das nächstfolgende Element (bezeichnet als das Cursor-Objekt).
- Mithilfe der Methode ***hasNext*** stellt der Iterator fest, ob noch weitere Elemente vorhanden sind.



Nutzung eines Iterators

- Java for-each-Schleife
  - Java-Klassen mit dem Interface **Iterable** können direkt in for-each Schleifen benutzt werden
  - In Array bzw. ArrayList automatisch vorhanden

Deklaration:

```
class A<T> implements Iterable<T> {
  ...
}
```

Anwendung:

```
A<String> a = new A<String>();
. . . .
for (String c:a) {
  System.out.println(c);
}
```


Wie kommen wir dahin

- Testbeispiel

```
String[] str = {"1", "2", "3"};
for (String s : str) {
  System.out.println(s);
}
```

- Beim Kompilieren wird aus dieser Schleife folgender Code:

```
String args1[] = {"1", "2", "3"};
String args2[] = args1;
int i = args2.length;
for(int j = 0; j < i; j++) {
  String s = args2[j];
  System.out.println(s);
}
```

Iterator

- Anwendbarkeit
  - Das Iterator-Muster kann benutzt werden, um auf die Elemente eines aus gleichartigen Objekten zusammengesetzten Aggregats zuzugreifen, ohne den inneren Aufbau des Aggregats zu kennen.
  - Methode „next“ ist eine Black-Box
  - um mehrere unabhängige Läufe über die Elemente eines Aggregats zu ermöglichen.
  - um eine gemeinsame Schnittstelle zum Durchlaufen unterschiedlicher Aggregate zur Verfügung zu stellen. Dies entspricht polymorpher Iteration.

Teilnehmer des Entwurfsmuster

- Iterator
  - definiert die Schnittstelle für Zugriff auf und Traversierung der Elemente
- KonkreterIterator
  - implementiert die Iterator-Schnittstelle. Unterhält Informationen über die letzte Position innerhalb der Aggregatelemente.
- Aggregat
  - definiert eine Schnittstelle zur Erzeugung eines Iteratorobjekts
- KonkretesAggregat
  - implementiert die Erzeugung eines Iterators und gibt eine instanziierte Referenz auf einen KonkreterIterator zurück.

Iteratoren

- Zusammenarbeit
  - Ein KonkreterIterator kennt das aktuelle Objekt im Aggregat und kann das nächste in der Traversierung folgende Element berechnen.
- Konsequenzen
  - Komplexe Aggregate können in mehr als einer Weise traversiert werden (z.B. vorwärts, rückwärts).
  - Iteratoren machen es einfach den Traversierungsalgorithmus zu ändern.
  - Iteratoren vereinfachen die Aggregatschnittstelle, da das Aggregat keine Iterationsschnittstelle unterstützen muss.
  - Ein Iterator kennt seinen Iterationszustand. Daher kann mehr als eine Iteration auf einem Aggregat gleichzeitig durchgeführt werden.

Was kann man nun iterieren

- In for-each Schleifen kann alles gestellt werden, was Iterable implementiert.
- Iterable ist ein generisches Interface und erwartet, dass die Methode iterator überschrieben wird, welche einen ebenfalls generischen Iterator zurück gibt.
- Wie so ein Iterator aussieht wird später erläutert. Zuerst programmieren wir eine Klasse MyIterable, die Iterable implementiert, und in der Methode iterator vorerst null zurück gibt.

```
import java.util.Iterator;
public class MyIterable<T> implements Iterable<T> {
  public Iterator<T> iterator() {
    return null;
  }
}
```

Nutzung der iterierbaren Objekte

Wir können nun ein Objekt dieser Klasse in einer for-each-Schleife verwenden.

```
MyIterable<String> myit = new MyIterable<String>();
for (String s : myit) {
  System.out.println(s);
}
```

Selbstverständlich endet die Ausführung dieses Codes in einer NullPointer-Exception, da wir keinen Iterator zurückgeben, sondern lediglich null. Was macht der Compiler aus diesem Code:

```
MyIterable myiterable = new MyIterable();
String s;
for(Iterator iterator = myiterable.iterator(); iterator.hasNext();
System.out.println(s))
  s = (String)iterator.next();
```


Was ist nun der Iterator

- Ein Iterator ist hier eine Art Aufzählung oder Auflistung von Daten und besitzt die Methoden ***hasNext***, ***next*** und ***remove***.
- ***hasNext*** gibt einen boolean zurück, ob noch weitere Elemente in dieser Auflistung vorhanden sind.
- Mit ***next*** wird das nächste Element abgefragt.
- ***remove*** entfernt das aktuelle Element aus dem Iterator.

```
import java.util.Iterator;
public class SOList<Type> implements Iterable<Type> {
   private Type[] arrayList;
   private int currentSize;
   public SOList(Type[] newArray) {
      this.arrayList = newArray;
      this.currentSize = arrayList.length;
   }
```



```
   public Iterator<Type> iterator() {
      Iterator<Type> it = new Iterator<Type>() {
         private int currentIndex = 0;

         public boolean hasNext() {
            return currentIndex < currentSize && arrayList[currentIndex] != null;
         }

         public Type next() {
            return arrayList[currentIndex++];
         }

         public void remove() {
            // TODO Auto-generated method stub
         }
      };
      return it;
   }
}
```

Einfacher Array-Iterator

```
import java.util.Iterator;

public class MyIterator<E> implements Iterator<E> {
  private int position = -1;
  private E[] arr = null;

 public void setData(E[] elements) {
  this.arr = elements;
 }
 public boolean hasNext() {
  return this.position + 1 < this.arr.length;
 }
 public E next() {
   return
this.arr[++this.position];
  }

 public void remove() {
//nicht implementiert
  }
 }
```


Einbindung des Iterators

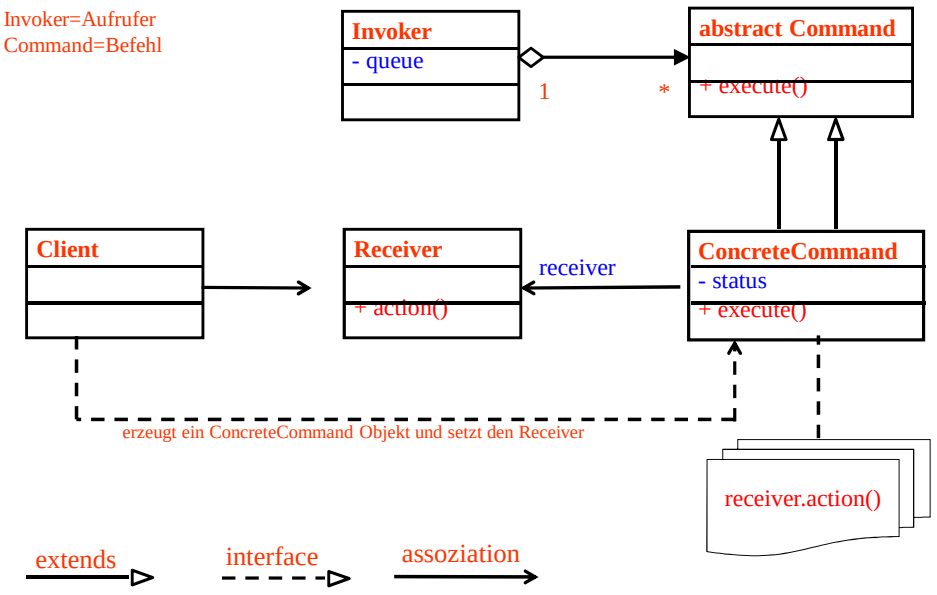
```
import java.util.Iterator;
 public class MyIterable<T> implements Iterable<T> {
   private T[] elements = null;

 public void setElements(T[] elements) {
   this.elements = elements;
 }
 public Iterator<T> iterator() {
   MyIterator<T> iter = new MyIterator<T>();
   iter.setData(this.elements);
   return iter;
 }
}
MyIterable<String> myit = new MyIterable<String>();
myit.setElements(new String[] {"1", "2", "3"});
for (String s : myit) {
 System.out.println(s);
}
```

Diese Klasse kann nun über die for-each-Schleife verarbeitet werden

Befehls-Muster

- **Motivation**
  - Manchmal sollen Objekte Anforderungen an andere Objekte senden, ohne Näheres über diese Anforderungen zu wissen. Zum Beispiel gibt es in Benutzeroberflächen Knöpfe und Menüpunkte welche entsprechend ihrer Bedienung durch den Benutzer eine Aktion initiieren sollen. Dabei ist die eigentlich ausgeführte Aktion für diese Bedienelemente belanglos.
- **Anwendbarkeit**
  - Das Befehl-Muster kann in folgenden Fällen angewandt werden:
    - Parametrisierung von Objekten mit auszuführenden Aktionen, wie etwa bei Menüpunkten. In imperativen Programmiersprachen wird dieses Vorgehen callback genannt.
    - Eine Anforderung soll zu einer anderen Zeit ausgeführt werden als sie spezifiziert wird.
    - Befehle werden in eine Warteschlange gestellt
    - ActionListener

Struktur des Commands/Befehls-Entwurfsmusters



Befehls-Entwurfsmuster

- **Teilnehmer**
  - Command
    - deklariert eine Schnittstelle zur Ausführung einer Operation
  - ConcreteCommand
    - definiert eine Verbindung zwischen dem Befehlsempfängerobjekt Receiver und einer Aktion.
    - implementiert execute durch Ausführung der entsprechenden Operation(en) des Befehlsempfängers.
  - Client
    - erzeugt ein ConcreteCommand Objekt und setzt dessen Befehlsempfänger (Receiver).
  - Invoker
    - fordert von Command die Ausführung einer Anfrage.
  - Receiver
    - Kann die angeforderten Operationen ausführen. Jede Klasse kann als Empfänger (Receiver) dienen.

Befehls-Entwurfsmuster

- **Zusammenarbeit**
  - Der Client erzeugt ein ConcreteCommand-Objekt und spezifiziert das Receiver-Objekt.
  - Ein Invoker-Objekt speichert das ConcreteCommand-Objekt.
  - Der Invoker führt durch den Aufruf von execute eine Anforderung aus.
  - Das ConcreteCommand-Objekt führt Operationen auf dem Receiver aus.

Befehls-Entwurfsmuster

- **Konsequenzen**
  - Befehl entkoppelt das Objekt welches eine Operation aufruft von dem Objekt welches die Operation ausführt.
  - Befehl ist ein echtes Objekt. Es kann wie jedes andere Objekt erweitert und manipuliert werden.
  - Mehrere Befehle können in einem Befehl zusammengefasst werden. (Makros)
  - Es ist einfach neue Befehle hinzuzufügen, da keine bestehende Klasse verändert werden muss.