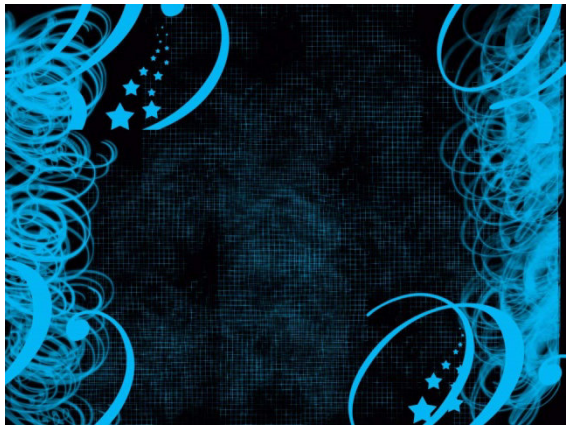
Objektorientierte und Funktionale Programmierung

SS 2014

6 Objektorientierte Entwurfsmuster



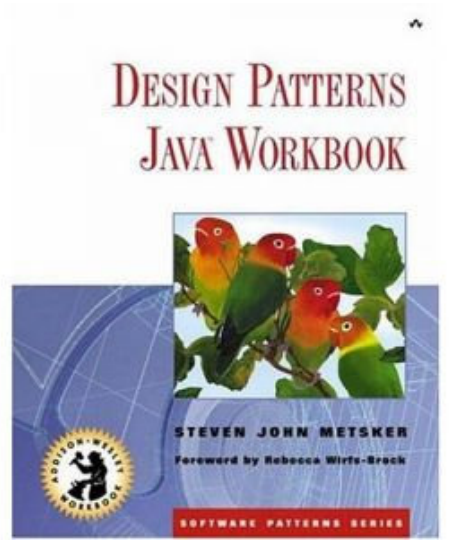
6 Objektorientierte Entwurfsmuster …

Lernziele

- Einige wichtige Entwurfsmuster kennen und verstehen
- Einsatzmöglichkeiten der Entwurfsmuster kennen
- Entwurfsmuster bei der Modellierung nutzen können
- Entwurfsmuster in OOD-Modellen erkennen können

Literatur

- [Ba05], Kap. 7.1, 7.2-7.4, 7.7, 7.8
- [GH+96], Kap. 1, 3.3, 3.4, 4.6, 5.2, 5.7
  - ggf. als Vertiefung



Inhalt

- Einführung
- *Factory Method (Fabrikmethode)*
- *Composite (Kompositum)*
- *Observer (Beobachter)*
- *Template Method (Schablonenmethode)*

6.1 Einführung

- **Entwurfsmuster (*design pattern*): *bewährte, generische*** Lösung für ein immer wiederkehrendes Entwurfsproblem
- Ein Muster besteht aus vier grundlegenden Elementen:
  - **Name des Musters**
  - **Problembeschreibung**
    - gibt an, wann das Muster sinnvoll anwendbar ist
  - **Lösungsbeschreibung**
    - kein konkreter Entwurf bzw. Implementierung
    - sondern abstrakte Beschreibung: Anordnung von Klassen bzw. Objekten
  - **Konsequenzen**
    - Vor- und Nachteile des Musters
    - zur Abwägung des Kosten-/Nutzenverhältnisses

6.1 Einführung ...

Klassifikation von Mustern

- **Klassenbasierte Muster**
  - behandeln Beziehungen zwischen Klassen
    - durch Generalisierungen ausgedrückt
    - zur Übersetzungszeit festgelegt
- **Objektbasierte Muster**
  - beschreiben Beziehungen zwischen Objekten
    - zur Laufzeit erzeugt bzw. veränderbar
  - nutzen Aggregation, Komposition, Delegation, z.T. auch Generalisierung

6.1 Einführung

Klassifikation von Mustern ...

- **Erzeugungsmuster**
  - machen System unabhängig davon, wie Objekte erzeugt, zusammengesetzt und repräsentiert werden
- **Strukturmuster**
  - setzen Klassen /Objekte zu größeren Strukturen zusammen
- **Verhaltensmuster**
  - Art und Weise der Interaktion zwischen Objekten / Klassen
    - Beschreibung komplexer Kontrollflüsse

Zur Abgrenzung: *Frameworks*

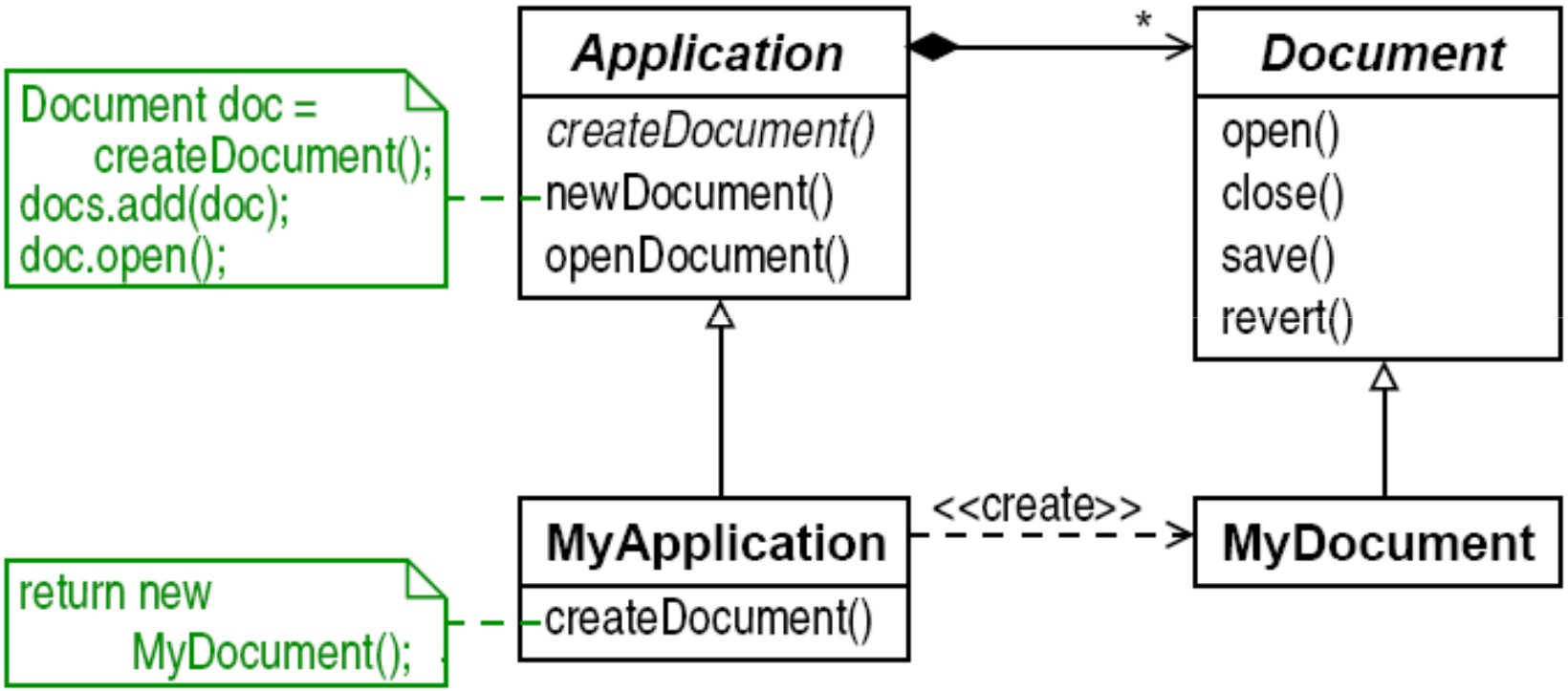
- Menge von Klassen, die einen wiederverwendbaren Entwurf für einen Anwendungsbereich implementiert
- Besteht aus konkreten und abstrakten Klassen
  - Anpassung des *Frameworks an ein konkretes Problem durch* Implementierung von Unterklassen der abstrakten Klassen
  - Methoden dieser Klassen werden vom *Framework aufgerufen*
    - Hollywood-Prinzip: "*Don't call us, we'll call you*"
- Unterschiede zwischen *Frameworks und Entwurfsmustern:*
  - Entwurfsmuster sind abstrakter (kein Code)
  - Entwurfsmuster sind kleiner (ein *Framework enthält oft* mehrere Muster)
  - Entwurfsmuster sind weniger spezialisiert

6.2 *Factory Method* (Fabrikmethode)

- Klassenbasiertes Erzeugungsmuster
- **Zweck:**
  - definiere in einer Klasse eine Schnittstelle zum Erzeugen von Objekten
  - lasse aber Unterklassen entscheiden, von welcher Klasse diese Objekte sind
- **Motivation**: z.B. *Framework* für Anwendungen, die mehrere Dokumente gleichzeitig anzeigen können
  - *Framework* verwendet abstrakte Klasse für die Dokumente
  - muß Dokumente erzeugen können, kennt aber nur deren abstrakte Oberklasse
  - Lösung: Erzeugung der Objekte durch eine **Fabrikmethode**, die von einer Unterklasse überschrieben wird

6.2 *Factory Method* (Fabrikmethode) …

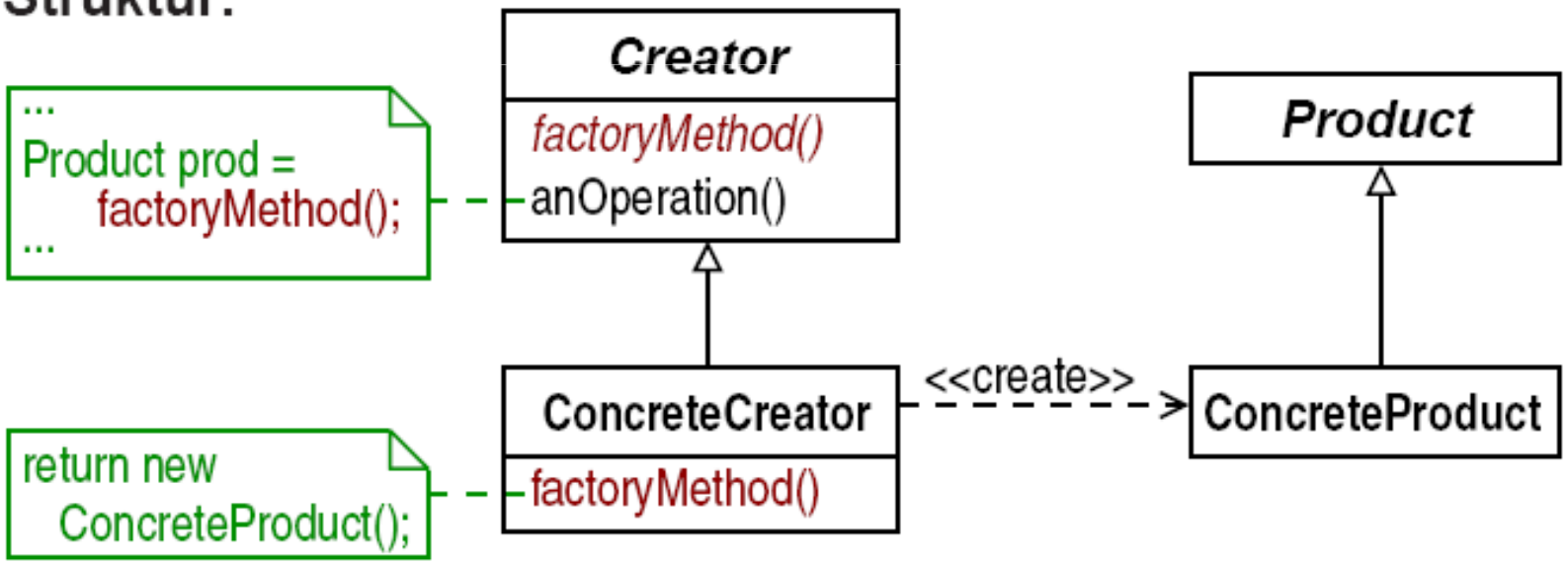
- **Motivation**...: UML-Diagramm des Beispiels



6.2 *Factory Method* (Fabrikmethode) …

- **Anwendbarkeit:**
  - wenn eine Klasse die von ihr zu erzeugenden Objekte nicht im voraus kennen kann
  - wenn Unterklassen einer Klasse festlegen sollen, welche Objekte erzeugt werden

Struktur:



6.2 *Factory Method* (Fabrikmethode) …

- **Struktur...:**
  - Product definiert die Schnittstelle der Objekte, die die Fabrikmethode erzeugt
  - ConcreteProduct implementiert diese Schnittstelle
  - Creator deklariert (und nutzt) die abstrakte Fabrikmethode
  - ConcreteCreator implementiert sie so, daß Objekte der Klasse ConcreteProduct zurückgegeben werden
- **Interaktionen:**
  - Creator verlässt sich darauf, daß Unterklassen die Fabrikmethode korrekt implementieren
- **Konsequenzen:**
  - das Muster verhindert, daß ein *Framework* anwendungsspezifische Klassen kennen muß

6.2 *Factory Method* (Fabrikmethode) …

- **Beispiel Abstrakter Hersteller**

```
package de.theserverside.designpatterns.factorymethod;

import java.util.ArrayList;
import java.util.List;

/**
* Stellt im UML-Klassendiagramm "Creator" dar, von der konkrete
* Klassen abgeleitet werden, die auch instanziiert werden können.
*/
public abstract class AbstrakterHersteller {
   protected List<AbstraktesFahrzeug> fahrzeuge =
      new ArrayList<AbstraktesFahrzeug>();

   /**
    * Delegiert die Instanziierung der konkreten Fahrzeuge an
    * implementierende Unterklassen.
    **/
```

6.2 *Factory Method* (Fabrikmethode) …

- **Beispiel**

```
public AbstrakterHersteller() {
    this.erzeugeFahrzeuge();
}

public List<AbstraktesFahrzeug> getFahrzeuge() {
    return fahrzeuge;
}

/**
 * Muss von einer Methode überschrieben werden, die konkrete
 * Fahrzeuge instanziiert. Dies ist das Herzstück des Factory
 * Method Pattern
 *
 */
protected abstract void erzeugeFahrzeuge();
```

6.2 *Factory Method* (Fabrikmethode) …

- **Beispiel BMW**

```
package de.theserverside.designpatterns.factorymethod;

/**
* Stellt im UML-Klassendiagramm "ConcreteCreator" dar, die die
* konkreten Klassen (ConcreteProduct) instanziiert.
*/
public class BMW extends AbstrakterHersteller {
    /**
     * Implementiert die abstrakte Methode aus der Oberklasse
     * und erzeugt konkrete Fahrzeugobjekte
     */
    protected void erzeugeFahrzeuge() {
        fahrzeuge.add(new Z4(231));
    }
}
```

6.2 *Factory Method* (Fabrikmethode) …

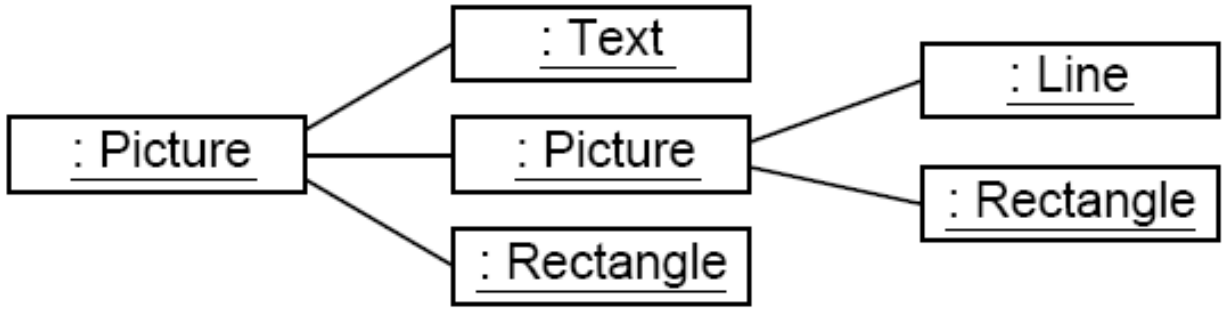
- **Beispiel BMW**


```
package de.theserverside.designpatterns.factorymethod;

/**
* Stellt im UML-Klassendiagramm "ConcreteProduct" dar,
die von
* der Factory Methode instanziiert wird.
*/
public class Z4 extends AbstraktesFahrzeug {
    public Z4(int kw) {
        super("BMW", "Z4", kw);
    }
}
```

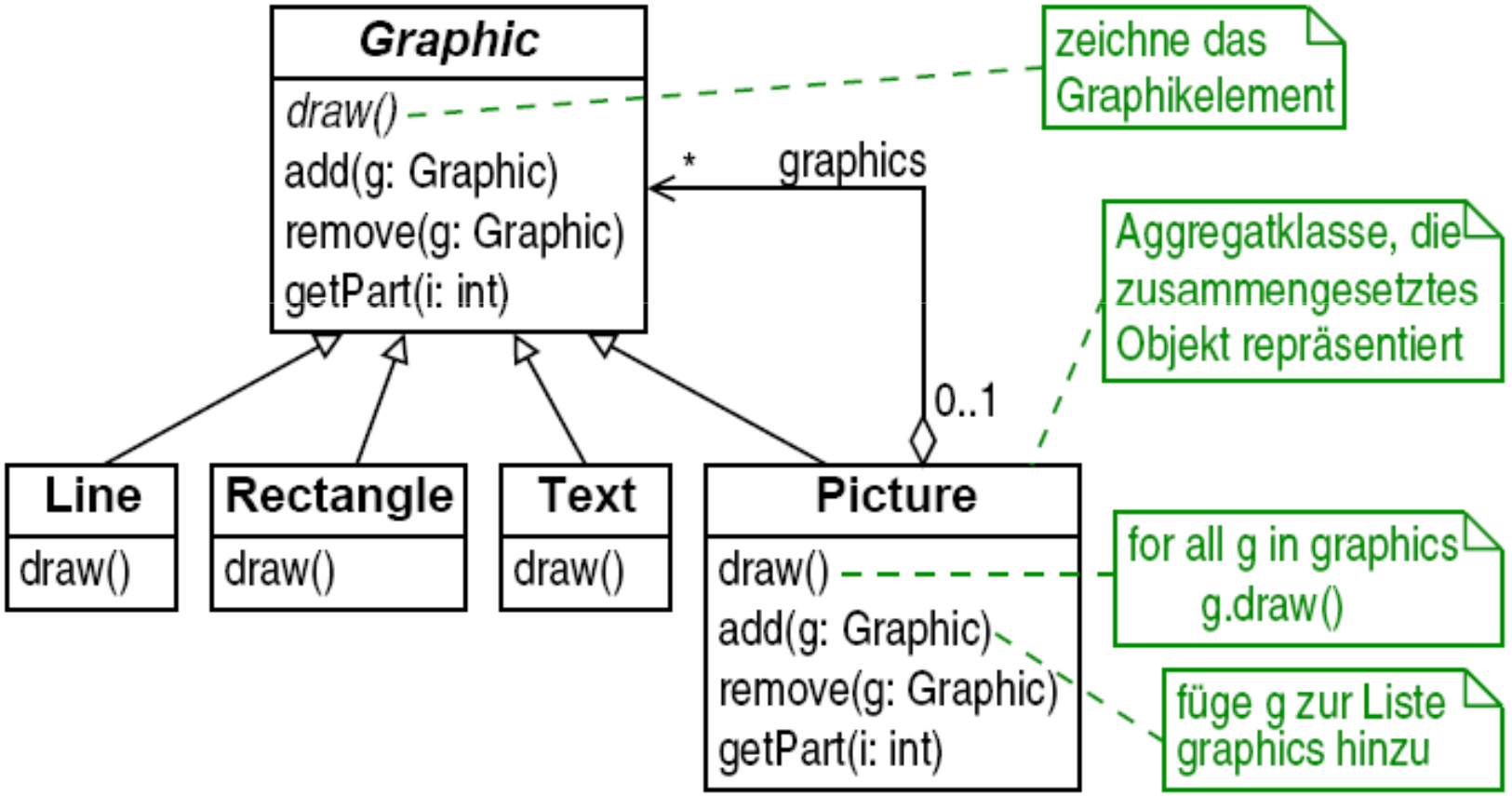
6.3 *Composite* (Kompositum)

- Objektbasiertes Strukturmuster
- **Zweck:**
  - setzt Objekte zu Baumstrukturen zusammen, um Teil/Ganzes-Hierarchien darzustellen
  - ermöglicht es, einzelne Objekte (Teile) und Baum von Objekten (Ganzes) gleich zu behandeln
- **Motivation: z.B. graphische Elemente in einem Graphikeditor**
  - einzelne Elemente werden zu Graphiken zusammengesetzt, die wiederum Teil von komplexeren Graphiken sein können:



6.3 *Composite* (Kompositum) …

- **Motivation** ...: UML-Diagramm des Beispiels

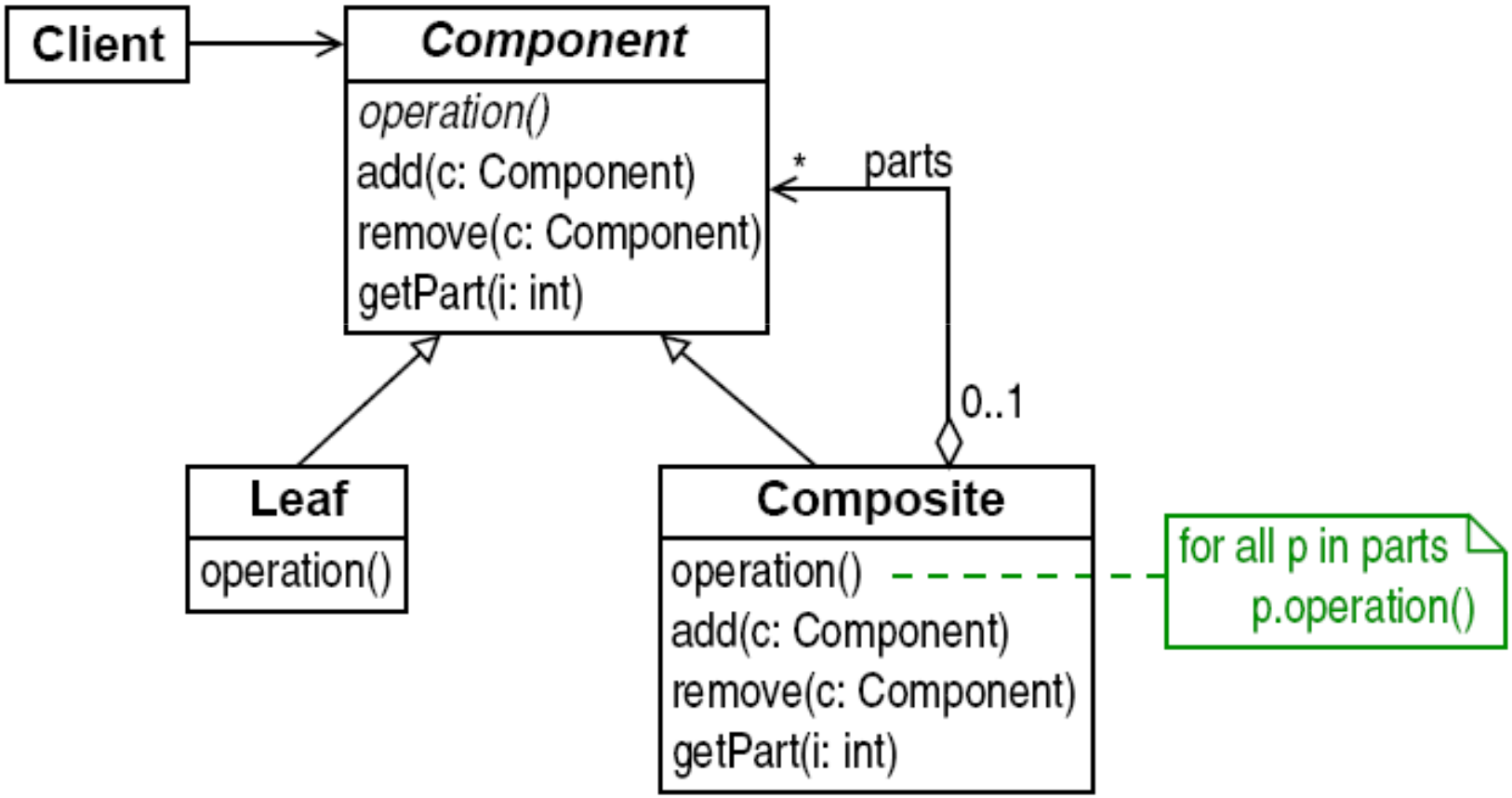


6.3 *Composite* (Kompositum) …

- **Anwendbarkeit:**
  - wenn Teil/Ganzes-Hierarchien dargestellt werden müssen
  - wenn Klienten keinen Unterschied zwischen elementaren und zusammengesetzten Objekten wahrnehmen sollen
    - d.h. sie sollen alle Objekte gleich behandeln können
- **Struktur:**
  - Component deklariert gemeinsame Schnittstelle für alle Objekte, incl. Verwaltung von Teilobjekten
  - Leaf repräsentiert elementare Objekte
  - Composite definiert Verhalten von zusammengesetzten Objekten
    - speichert Teilobjekte
    - reicht Methodenaufrufe ggf. an alle Teilobjekte weiter

6.3 *Composite* (Kompositum) …

- **Struktur...:**

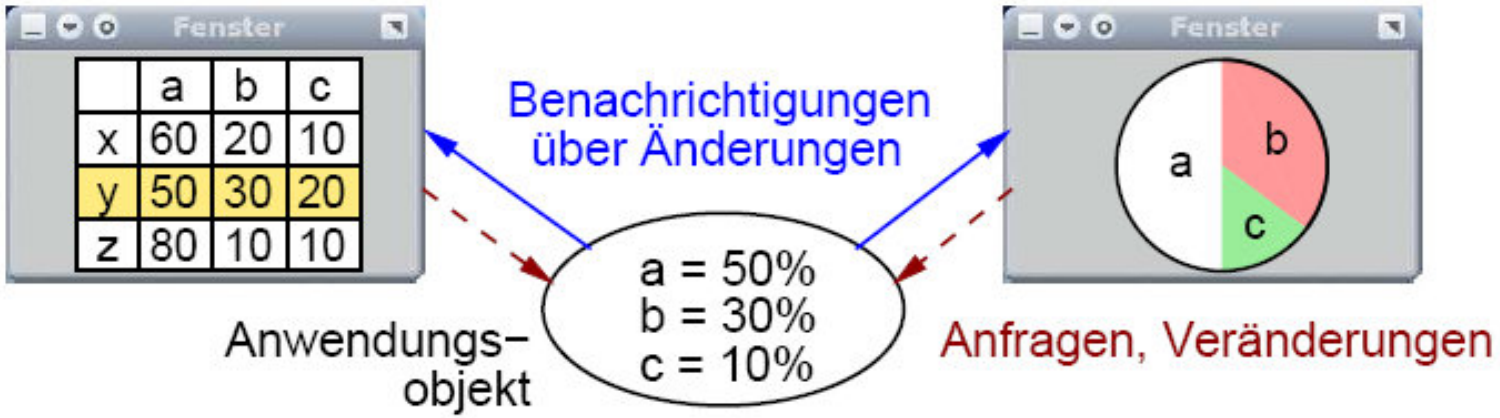


6.3 *Composite* (Kompositum) …

- **Interaktionen:**
  - Klienten verwenden nur die Schnittstelle Component
  - elementare Objekte verarbeiten Botschaften direkt
  - zusammengesetztes Objekt gibt Botschaften an seine Teilobjekte weiter
    - ggf. mit Vor- und/oder Nachbearbeitung
- **Konsequenzen:**
  - vereinfacht Klienten, da einfache und zusammengesetzte Objekte gleich behandelt werden
  - vereinfacht es, neue Arten von Komponenten einzufügen
  - erschwert es, mögliche Komponenten des Kompositums einzuschränken (Prüfung zur Laufzeit nötig)
- Anmerkung: vgl. z.B. JComponent, JPanel und JButton in Swing

6.4 *Observer* (Beobachter)

- Objektbasiertes Verhaltensmuster
- **Zweck:**
  - bei Änderung eines Objekts werden alle davon abhängigen Objekte benachrichtigt
  - diese können dann ihren Zustand aktualisieren
- **Motivation:** z.B. verschiedene graphische Darstellungen der Daten eines Objekts in einem GUI

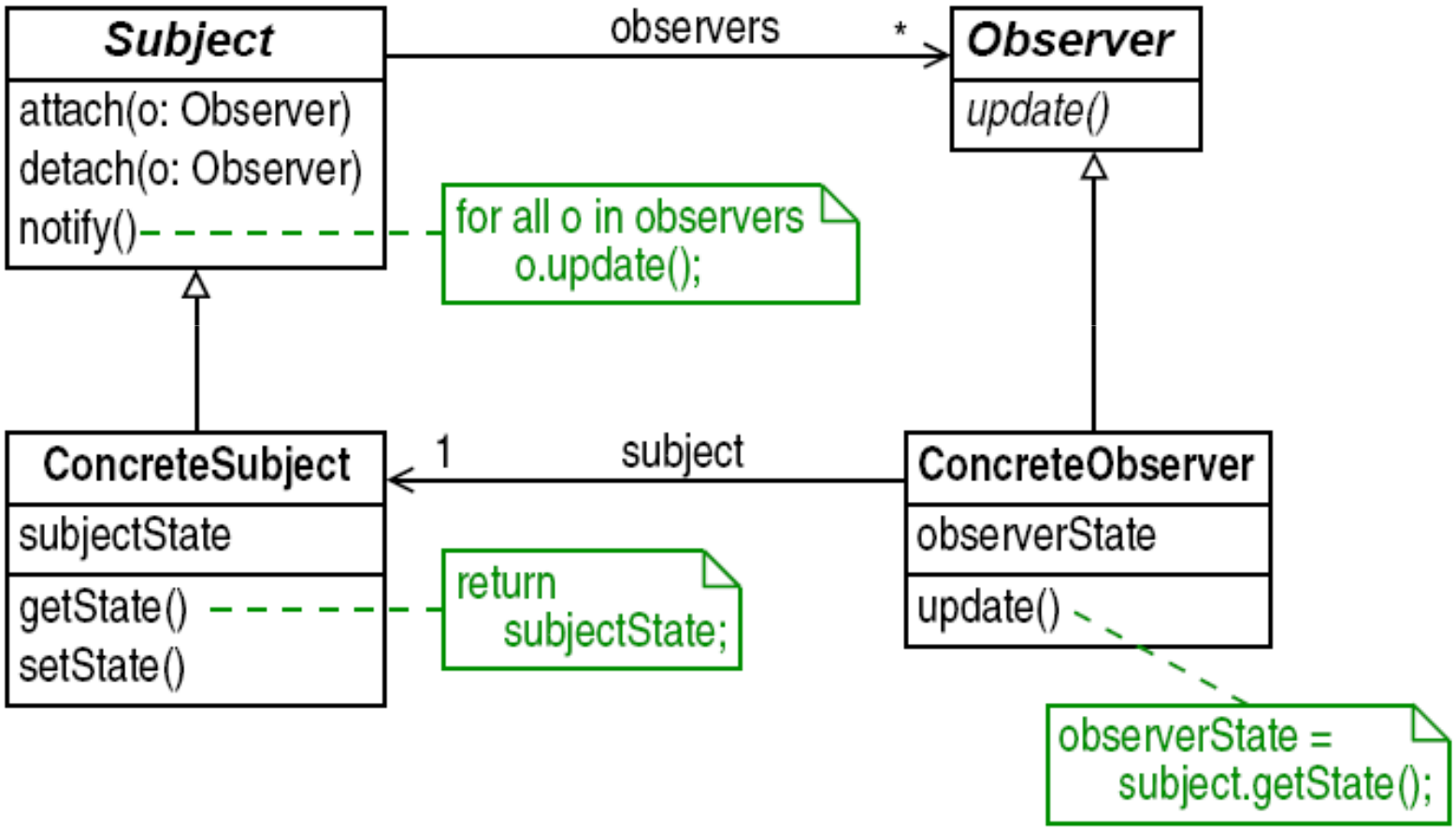


6.4 *Observer* (Beobachter) …

- **Anwendbarkeit:**
  - eine Abstraktion besitzt zwei Aspekte, die wechselseitig voneinander abhängen: Kapselung in unterschiedliche Objekte
  - Änderung eines Objekts⇒ Änderung anderer Objekte
    - unbekannt, wie viele Objekte geändert werden müssen
  - Objekt soll andere Objekte benachrichtigen; lose Kopplung
- **Struktur:**
  - Subject kennt beliebige Anzahl von Beobachtern
  - Observer: Schnittstelle für alle konkreten Beobachter
  - ConcreteSubject speichert Daten, die für konkrete Beobachter relevant sind
  - ConcreteObserver kennen das konkrete Subjekt, merken sich Zustand, der mit dem des Subjekts konsistent sein soll

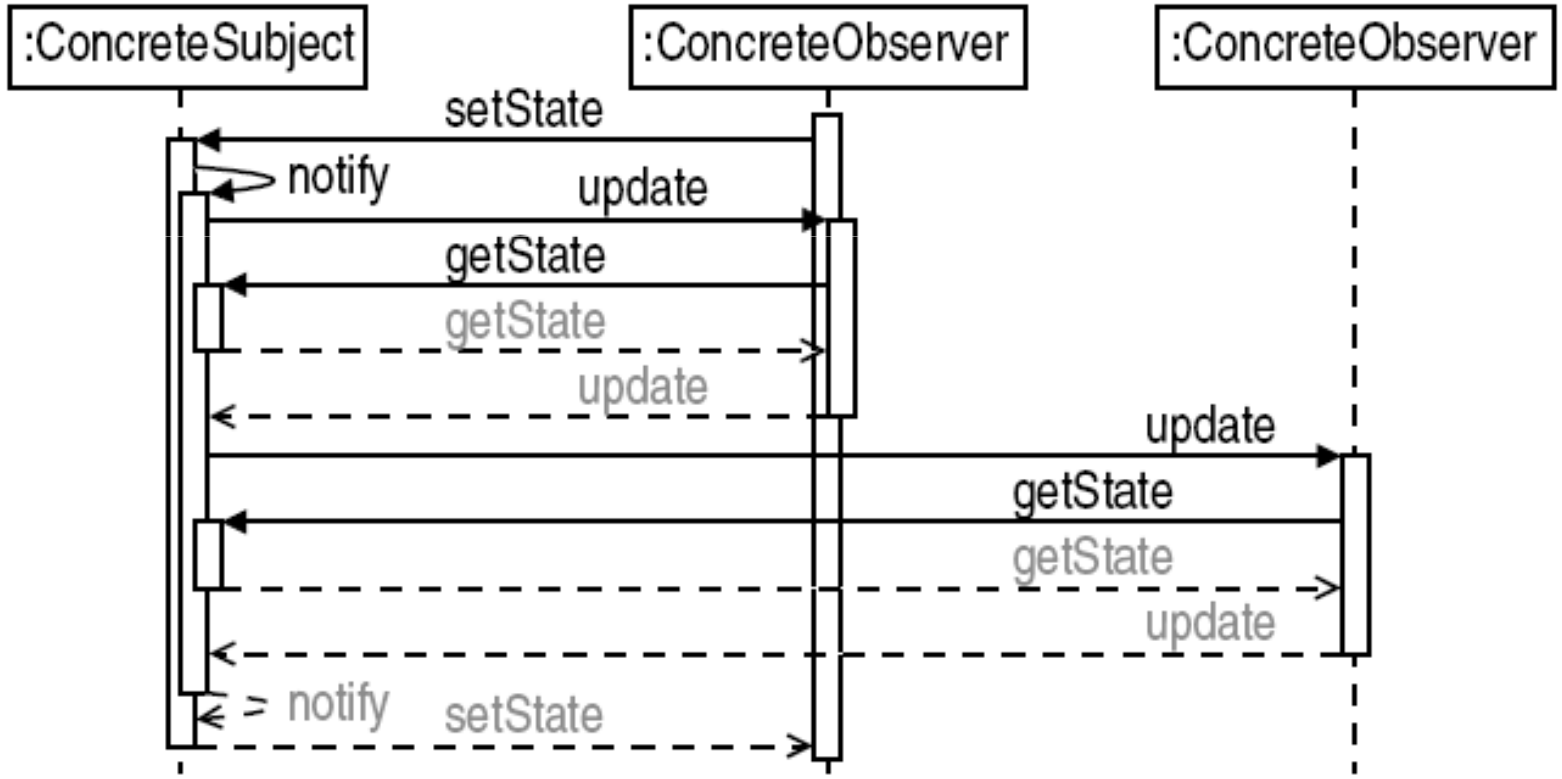
6.4 *Observer* (Beobachter) …

- **Struktur:**



6.4 *Observer* (Beobachter) …

- **Interaktionen:**
- konkretes Subjekt benachrichtigt bei Änderung alle Beobachter, die daraufhin ihren Zustand aktualisieren



6.4 *Observer* (Beobachter) …

- **Konsequenzen:**
  - Code von Subjekten und Beobachtern ist unabhängig
  - Subjekte und Beobachter einzeln wiederverwendbar
  - neue Beobachter ohne Änderung des Subjekts hinzufügbar
  - unerwartete Kaskaden von Benachrichtigungen möglich
- Anmerkungen:
  - vgl. Ereignisbeobachter in Swing bzw. AWT (=> **5.4.5)**
  - das *Observer -Muster ist auch Grundlage der Anbindung des* Fachkonzepts an ein GUI
  - *Observer -Muster wird im MVC- (Model / View/Controller)-* Paradigma verwendet
    - *Model ˆ= Subjekt, View zur Datenansicht ˆ= Beobachter,* zusätzlich *Controller zur Veränderung der Daten*

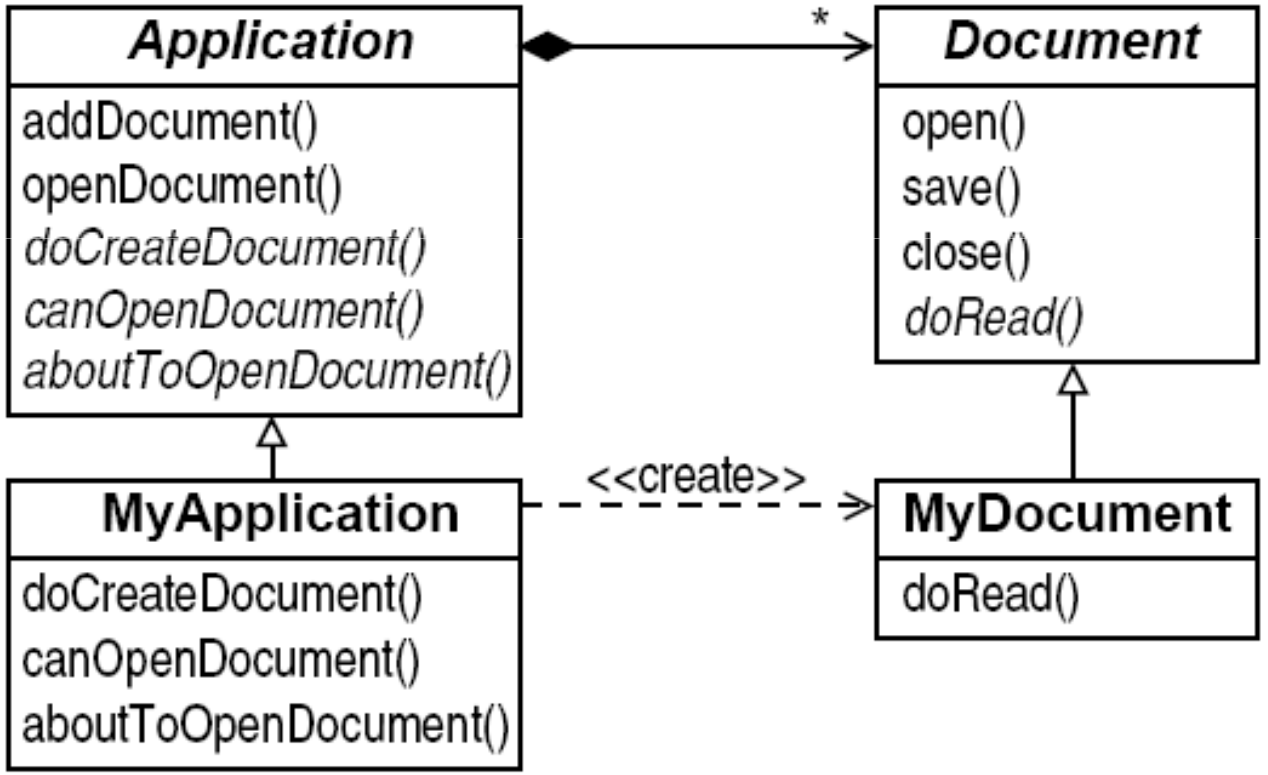
6.5 *Template Method* (Schablonenmethode)

- Klassenbasiertes Verhaltensmuster
- **Zweck:**
  - definiert in einer Operation das Skelett eines Algorithmus
  - delegiert einzelne Teilschritte an Unterklassen
- **Motivation**: z.B. Framework für Dokumenten-Anwendung
  - öffnen des Dokuments durch folgenden Algorithmus:

```
public void openDocument(String name) {
  if (!canOpenDocument(name)) return;
  Document doc = doCreateDocument(); // Fabrikmethode
  if (doc != null) {
    addDocument(doc);
    aboutToOpenDocument(doc); // abstrakte Methode
    doc.open();
    doc.doRead();             // abstrakte Methode
  }}
```

6.5 *Template Method* (Schablonenmethode)

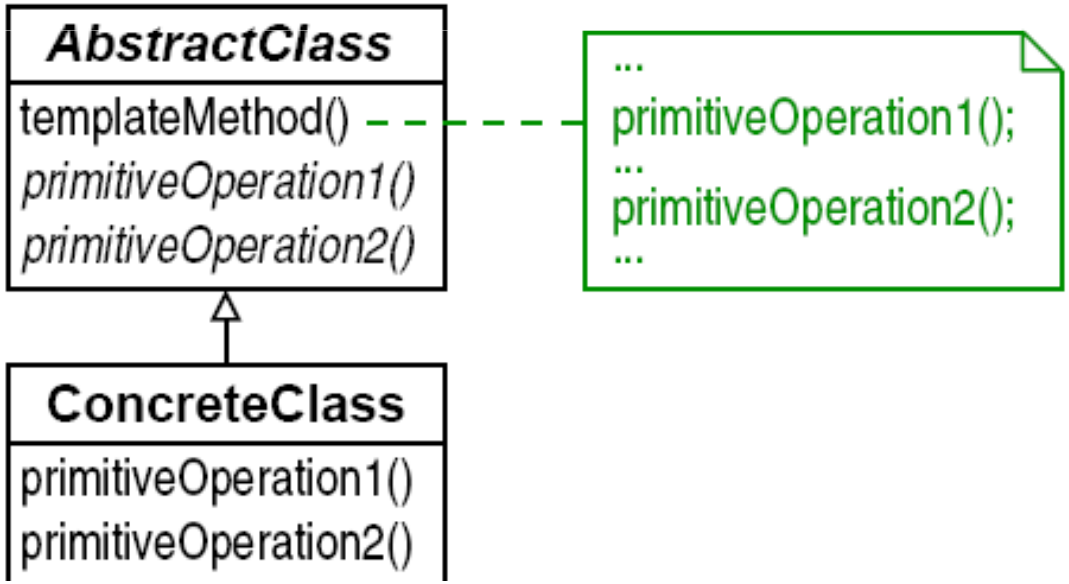
- **Motivation...:**
- openDocument() ist eine **Schablonenmethode**
- Klassendiagramm:



6.5 *Template Method* (Schablonenmethode)

- **Anwendbarkeit:**
  - um invariante Teile eines Algorithmus nur einmal zu codieren und variierende Teile in Unterklassen zu implementieren
  - wenn gemeinsames Verhalten von Klassen in einer Oberklasse realisiert werden soll, um Code-Duplikation zu vermeiden
  - um Erweiterungen durch Unterklassen zu kontrollieren

Struktur:



6.5 *Template Method* (Schablonenmethode)

- **Struktur...:**
  - AbstractClass definiert abstrakte primitive Operationen, implementiert die Schablonenmethode (Skelett des Algorithmus)
  - ConcreteClass implementiert die primitiven Operationen
- **Interaktionen:**
  - ConcreteClass stützt sich darauf ab, daß AbstractClass die invarianten Teile des Algorithmus vorgibt
- **Konsequenzen:**
  - grundlegende Technik zur Wiederverwendung von Code
  - invertierter Kontrollfluß (”*Don’t call us, we’ll call you“)*
  - AbstractClass kann auch leer implementierte primitive Operationen (Einschubmethoden) vorsehen
    - kontrollierte Erweiterbarkeit durch die Unterklassen