Entwurfsmuster

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

Wichtig: Klausuranmeldung

- **2. Teilklausur am 22.7.2006 !**
- **Klausuranmeldung ist für Klausurteilnahme erforderlich!**
- Die Anmeldung zur Klausur ist jetzt über UniWorx möglich.
- Bitte bis zum 19.7., 18 Uhr, anmelden.

Ziele

- Den Nutzen von Entwurfsmustern verstehen lernen
- Das Strategiemuster wiederholen
- Fabrik- und Beobachtermuster als wichtige Beispiele von Entwurfsmustern kennen und anwenden lernen

Entwurfsmuster

- **Entwurfsmuster** (engl. *design pattern*) sind *Regeln* oder *Richtlinien* (auch *Muster*), um häufig auftretende Probleme bei der Erstellung eines Programms zu lösen.
- Entwurfsmuster beziehen sich auf immer wiederkehrende Aufgaben-stellungen, die beim Software-Design entstehen *auf einer abstrakten Ebene*.
- Entstanden ist der Ausdruck „Entwurfsmuster“ in der *Architektur*, von wo er für die (objekt-orientierte) Softwareentwicklung übernommen wurde.
- Der Nutzen eines Entwurfsmusters liegt in der (programmiersprachen-unabhängigen) *Identifikation (Konzeptualisierung) einer Klasse von Entwurfsproblemen* und der *Beschreibung und Diskussion der Vor- und Nachteile* einer Lösung dafür.

Entwurfsmuster: Historische Entwicklung

C. Alexander *1936 Architekturprof. in Berkeley

- **1977 Christopher Alexander**: *A Pattern Language. Towns, Buildings, Construction* – eine Sammlung von Entwurfsmustern in der Architektur
- **1987 Kent Beck** und **Ward Cunningham**: Entwurfsmuster für graph. Benutzerschnittstellen in Smalltalk
- **1989-91 James Coplien**: Entwurfsmuster für C++ (Advanced C++ Idioms.
- **1991** Diss. **Erich Gamma** über Entwurfsmuster für Objekt-Orientierung
- **1995** Die „**Viererbande**“ („Gang of Four“, GoF) **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: „ Design Patterns - Elements of Reusable Object-Oriented Software“

Erich Gamma, Richard Helm, John Vlissides, Ralph Johnson

Die Entwurfsmuster der „Viererbande“

Die 23 GoF Pattern sind in 3 Gruppen eingeteilt:

behandeln Objekterzeugung

behandeln Objektinteraktion und -Verantwortlichkeiten

Erzeugungsmuster:
- **Abstract Factory** (87)
- Builder (97)
- **Factory Method** (107)
- Prototype (117)
- Singleton (127)

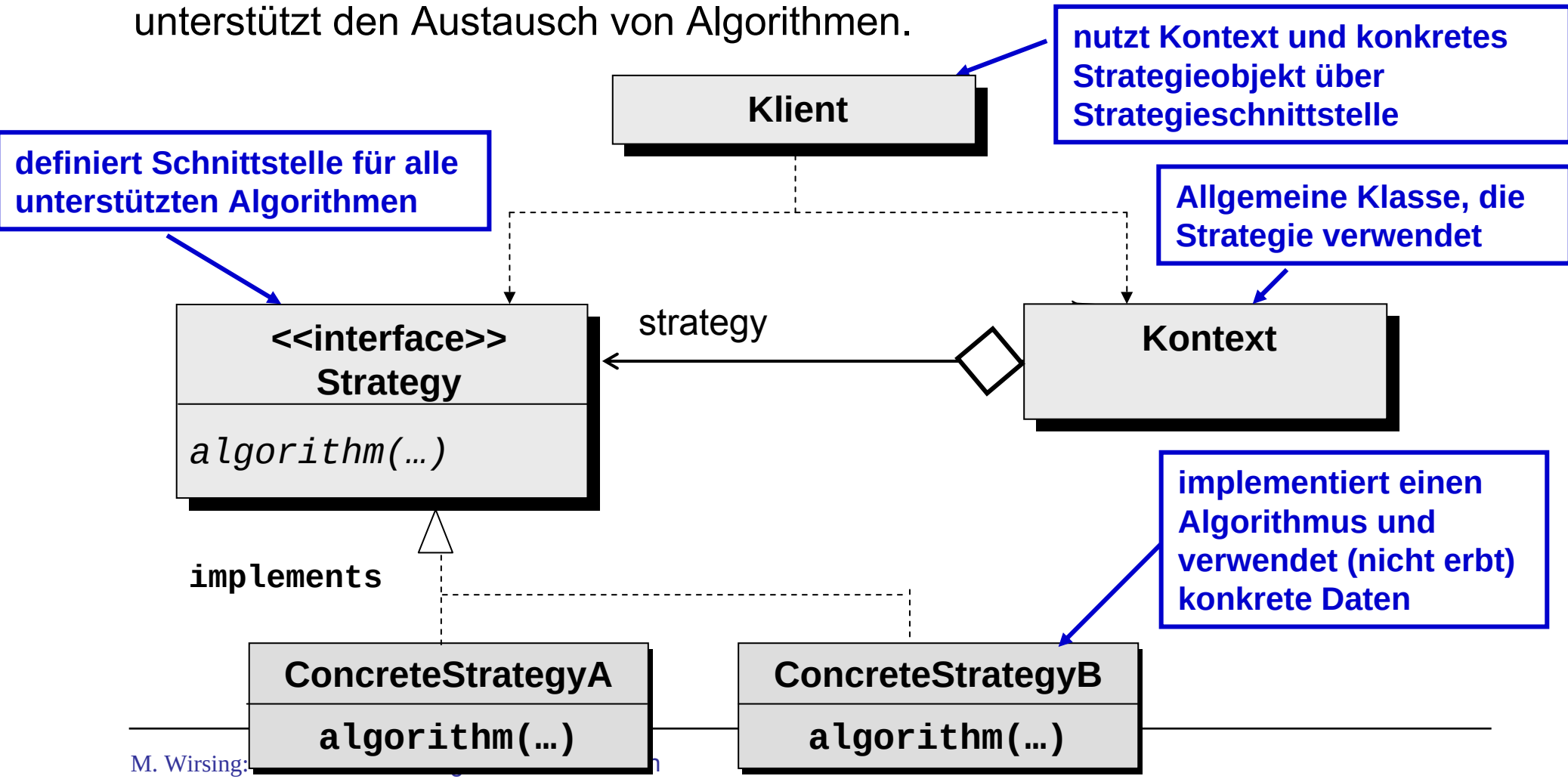
Strukturmuster:
- **Adapter (139)**
- Bridge (151)
- **Composite** (163) [später]
- Decorator (175)
- Facade (185)
- Flyweight (195)
- Proxy (207)

Verhaltensmuster:
- Chain of Responsibility (223)
- Command (233)
- Interpreter (243)
- Iterator (257)
- Mediator (273)
- Memento (283)
- **Observer** (293)
- State (305)
- **Strategy** (315)
- Template Method (325)
- Visitor (331)

behandeln Komposition von Klassen und Objekten

Wiederholung: Das Strategiemuster

- Das **Strategiemuster** entkoppelt Objekte von ihrem Verhalten und unterstützt den Austausch von Algorithmen.



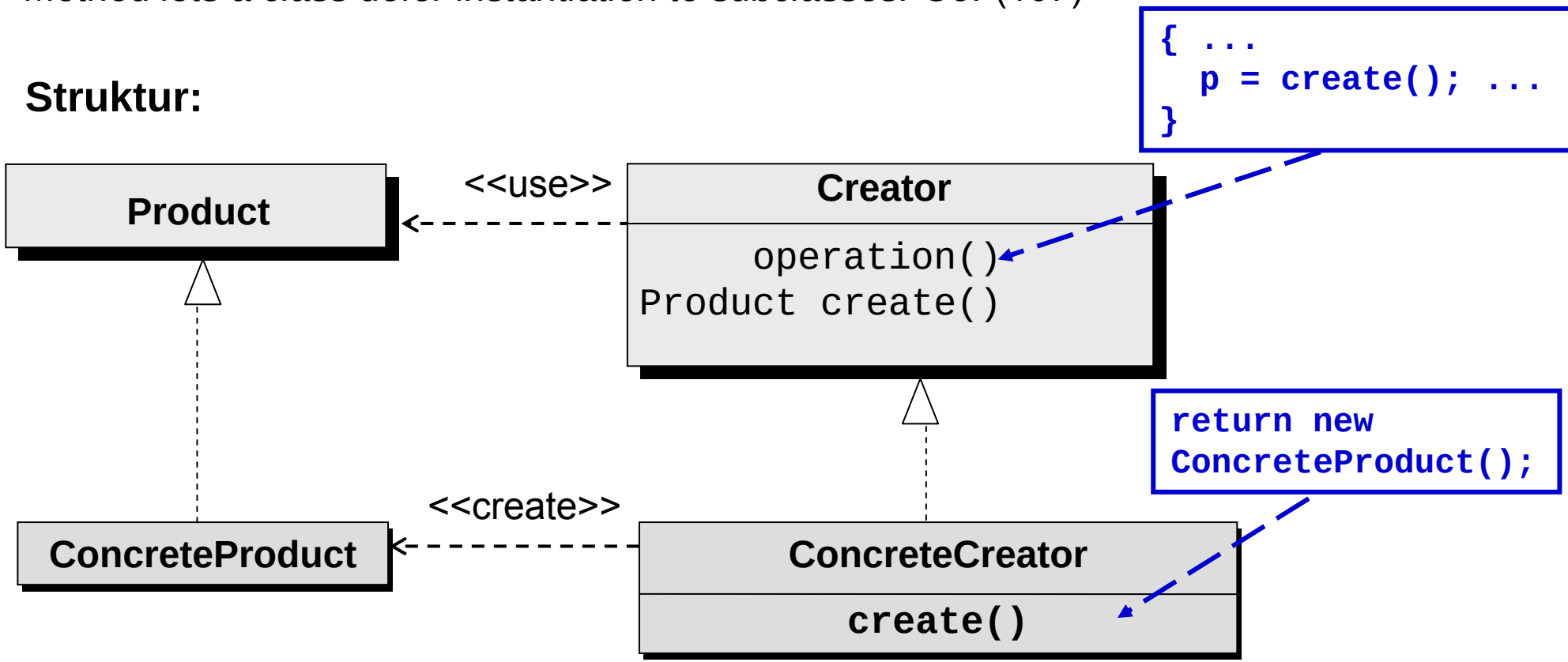
Wiederholung: Das Strategiemuster: Vorteile/Nachteile

- Das **Strategiemuster** entkoppelt Objekte von ihrem Verhalten und unterstützt den Austausch von Algorithmen.
- **Vorteile**
  - Es wird eine Familie von Algorithmen definiert.
  - Strategien bieten eine Alternative zur Unterklassenbildung,
    helfen Mehrfachverzweigungen zu vermeiden und
    verbessern dadurch die Wiederverwendung.
  - Strategien ermöglichen die Auswahl aus verschiedenen Algorithmen-Implementationen ("Algorithmen-Polymorphie") und erhöhen dadurch die Flexibilität.
- **Nachteile**
  - Klienten müssen die unterschiedlichen Strategien kennen,
    um zwischen ihnen auswählen zu können.
  - Gegenüber der direkten Implementation der Algorithmen im Klienten erzeugen Strategien zusätzlichen Kommunikationsaufwand zwischen Strategie und Klient.
  - Die Anzahl der Objekte wird erhöht.

Fabrikmethode (Factory Method)

Define an interface for creating an object, but let subclasses decide which class to instanciate. Factory Method lets a class defer instantiation to subclasses. GoF(107)

Struktur:



Fabrikmethode (Factory Method)

- Das **Fabrikmethodenmuster** definiert eine Schnittstelle zur Erzeugung eines Objektes, wobei es den Unterklassen überlassen bleibt, von welcher Klasse das zu erzeugende Objekt ist.
- **Beteiligte Klassen**
  - **Product**
    Basistyp (Klasse oder Schnittstelle) für das zu erzeugende Produkt
  - **Creator (Klasse oder Schnittstelle)**
    definiert die Fabrikmethode, um ein solches Produkt zu erzeugen
    (mitunter wird für die Fabrikmethode eine Implementierung vorgegeben, die ein "Standard-Produkt" erzeugt)
  - **ConcreteProduct**
    implementiert die Produkt-Schnittstelle (Subtyp von Produkt)
  - **ConcreteCreator**
    überschreibt die Fabrikmethode um konkrete*(re)* Produkte zu erzeugen, determiniert damit das konkrete Produkt (z.B. durch Aufruf des Konstruktors einer konkreten Klasse)

Fabrikmethode (Factory Method)

- **Folgerungen:**
  - Die abstrakte Methode `create()` sehr spezifisch: Sie erzeugt ein Objekt.
  - Das „Wie“ der Objekterzeugung ist hinter `create()` versteckt (Geheimnisprinzip).
- **Vorteile:**

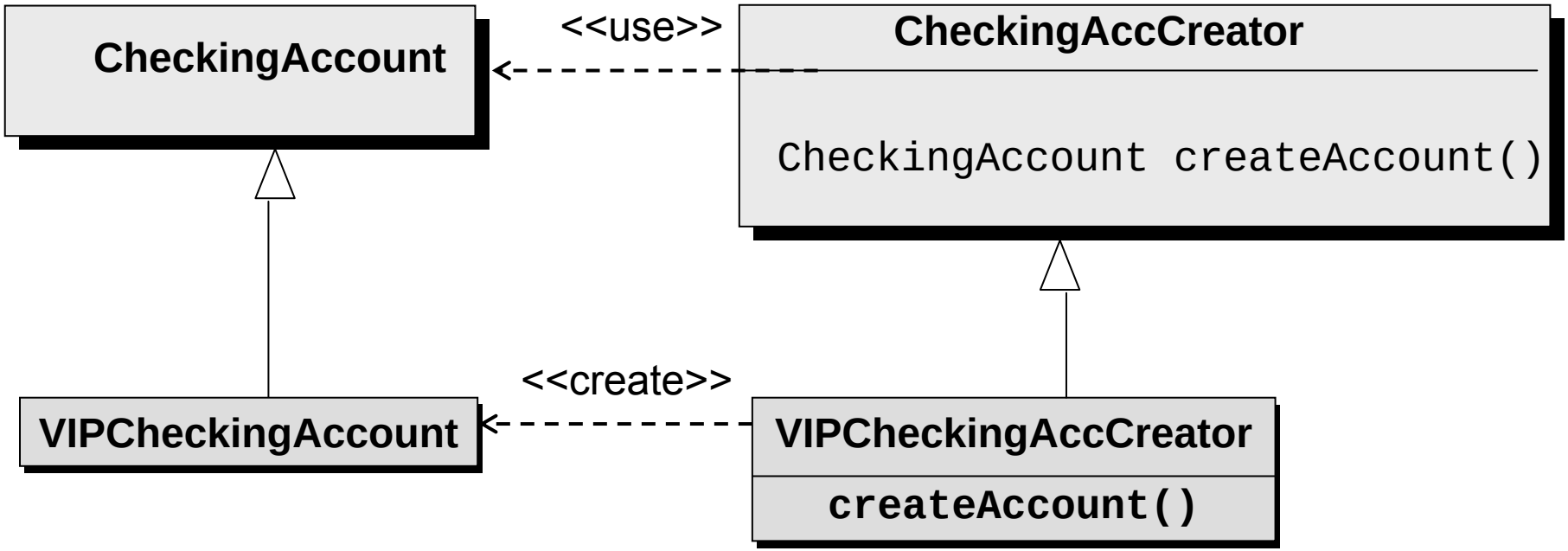
  Fabrikmethoden entkoppeln ihre Aufrufer von Implementierungen konkreter Produktklassen.
- **Nachteile:**

  Die Verwendung dieses Erzeugungsmusters läuft auf Unterklassenbildung hinaus.
- **Implementierungsvarianten:**
  - Der Methode `create()` können auch Parameter übergeben werden, die direkt als Parameter für den Konstruktor verwendet werden können.
  - Konkrete Implementierungen der `create()` Methode müssen nicht bei jedem Aufruf ein neue Instanz erzeugen. Es können Instanzen wieder verwendet werden, wenn
    - entweder ihr Zustand unveränderlich ist (eng. immutable objects)
    - oder sie in einem Zustand sind, der Wiederverwendung erlaubt (z.B. Datenbankverbindungen).

Beispiel: Fabrikmethode für Konten

- **Beispiel:**

  Mit dem Fabrikmethodenmuster kann eine Bank unterschiedliche Angebote für Konten zusammenstellen.



Beispiel Fabrikmethode: Unterschiedliche Konten

```
public class Account {
  public Account() { . . .}
  public double getBalance() { ... }
  public double deposit(double amount) { ... }
  ...
}

public class CheckingAccount extends Account {
  // implementiert die geforderten Funktionen ... }

public class SavingsAccount extends Account {
  // implementiert die geforderten Funktionen ... }
```

Beispiel Fabrikmethode: Unterschiedliche Konten

Im Gegensatz zu einem VIP muss ein normaler Kunde Überziehungszinsen zahlen u. darf sein Konto nicht überziehen . . .

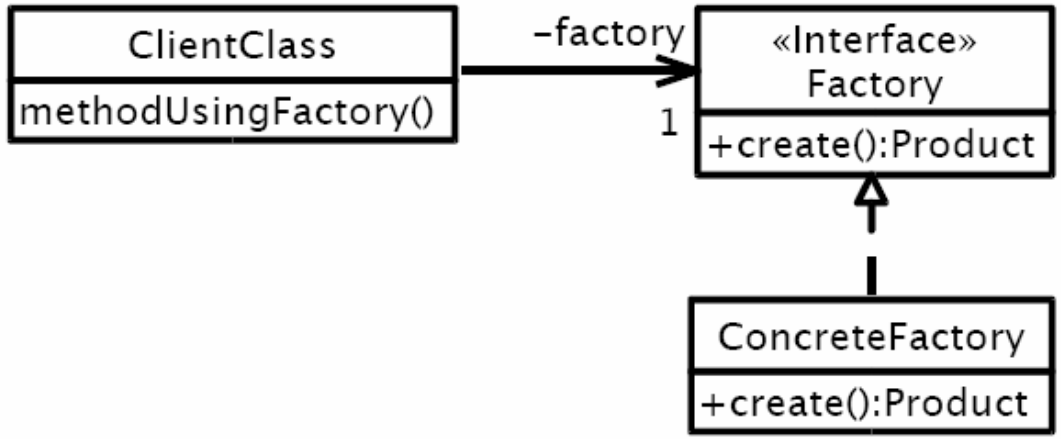
```
public class CheckingAccCreator {
 public CheckingAccount createAccount() {
       return new CheckingAccount(0.0, 0);} }

public class VIPCheckingAccCreator extends CheckingAccCreator{
 public CheckingAccount createAccount(){
       return new CheckingAccount(0.0, -50000, 0, 0.0);}   }

public class NormalCheckingAccCreator extends CheckingAccCreator{
 public CheckingAccount createAccount() {
       return new CheckingAccount(0.0, 0, 10, 0.3);}}
public class SavingsAccCreator {
 public SavingsAccount createAccount() {
       return new SavingsAccount(0.0, 0, 1.5);}     }

public class WerbungSavingsAccCreator extends SavingsAccCreator{
 public SavingsAccount createAccount() {
       return new SavingsAccount(0.0, 2000, 3.0);}  }
```

Beispiel Fabrikmethode: Unterschiedliche Konten

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

public class MyAccounts {
    private List <Account> kontos = new ArrayList<Account>();
    public MyAccounts() {
        createWithBonus(new WerbungSavingsAccountCreator(), 0.0);
        createWithBonus(new VIPCheckingAccountCreator(), 999.0);
        for(int i=0; i < kontos.size(); i++) {
            System.out.println("Kontostand“ + i + ": "
                    + kontos.get(i).getBalance());
        }
    }

    private void createWithBonus(AccountCreator creator, double bonus) {
        this.kontos.add(creator.createAccount());
        this.kontos.get(1).deposit(bonus);
    }
}
```

Varianten der Fabrikmethode

- **Factory Interface: Factory via Strategy Pattern:**



Varianten der Fabrikmethode

- **Static Factory Method:**
  - ist sehr ähnlich zu einem Konstuktor. Das Erzeugerobjekt ist eine Klasse.
  - **Beispiel:** Klasse java.awt.Font hat folgende statische Methode, welche eine neue Instanz der Klasse erzeugt:

```
public static Font decode(String description)
```

  - Factories selber werden oft mit statischen Factory Methoden erzeugt.
  - **Vorteile** statischer Fabrikmethoden sind:
    - Im Gegensatz zu Konstruktoren kann man die Namen statischer Factory-Methoden sebst wählen
    - Sie können die Performanz verbessern, da nicht jedes Mal neue Objekte erzeugt werden müssen.
    - Sie können einen Subtyp des Resultattyps zurückgeben; deshalb können sie ein Interface als Resultattyp besitzen.

Varianten der Fabrikmethode

- **Beispiel für Umwandlung eines Konstruktors in mehrere statische Fabrikmethoden**

  Der Typ Foo besitzt einen „normalen“ öffentlichen Konstruktor:

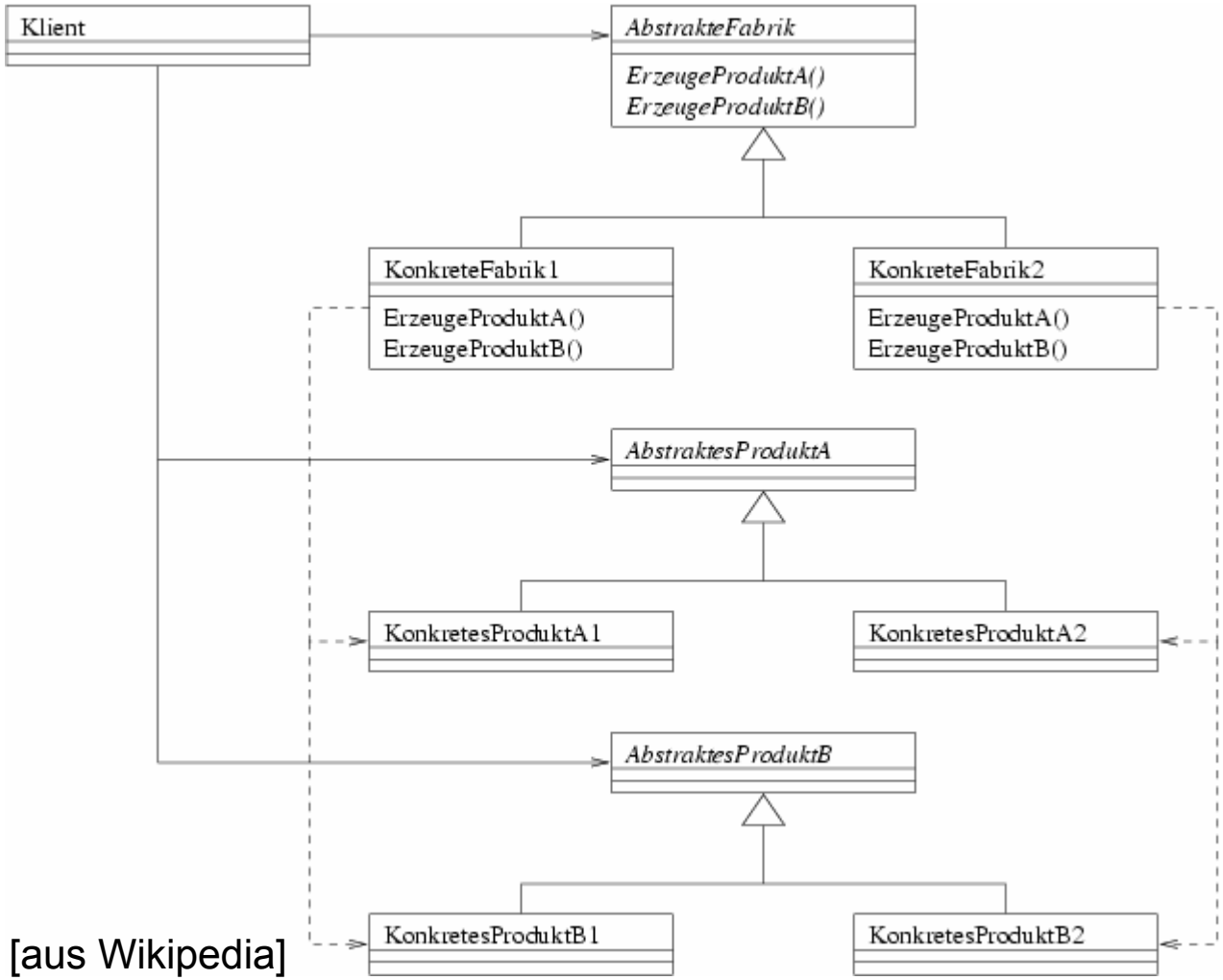
```
public class Foo {
    // normaler Konstruktor
    public Foo() {
    }
}
```

Varianten der Fabrikmethode

Die neue Klasse hat einen **privaten** Konstruktor und **statische Fabrik-Methoden mit aussagekräftigen Namen**:

```
public class Foo {
    private static Foo cached = new Foo();
    public static Foo createEmptyFoo() {
       return this.cached; // erzeuge nur einmal eine Instanz... }
    // Gib bei Bedarf Unterklassen zurueck:
    public static Foo createEmptyFoo(FooKind kind) {
        switch(kind) {
            case BASIC:       return new Foo();
            case EXTENDED:    return new SubFoo();
            default:          return this.cached;  }
    }
    private Foo() {
    }
}

public class SubFoo extends Foo() {
}
```

Varianten: Abstrakte Fabrik



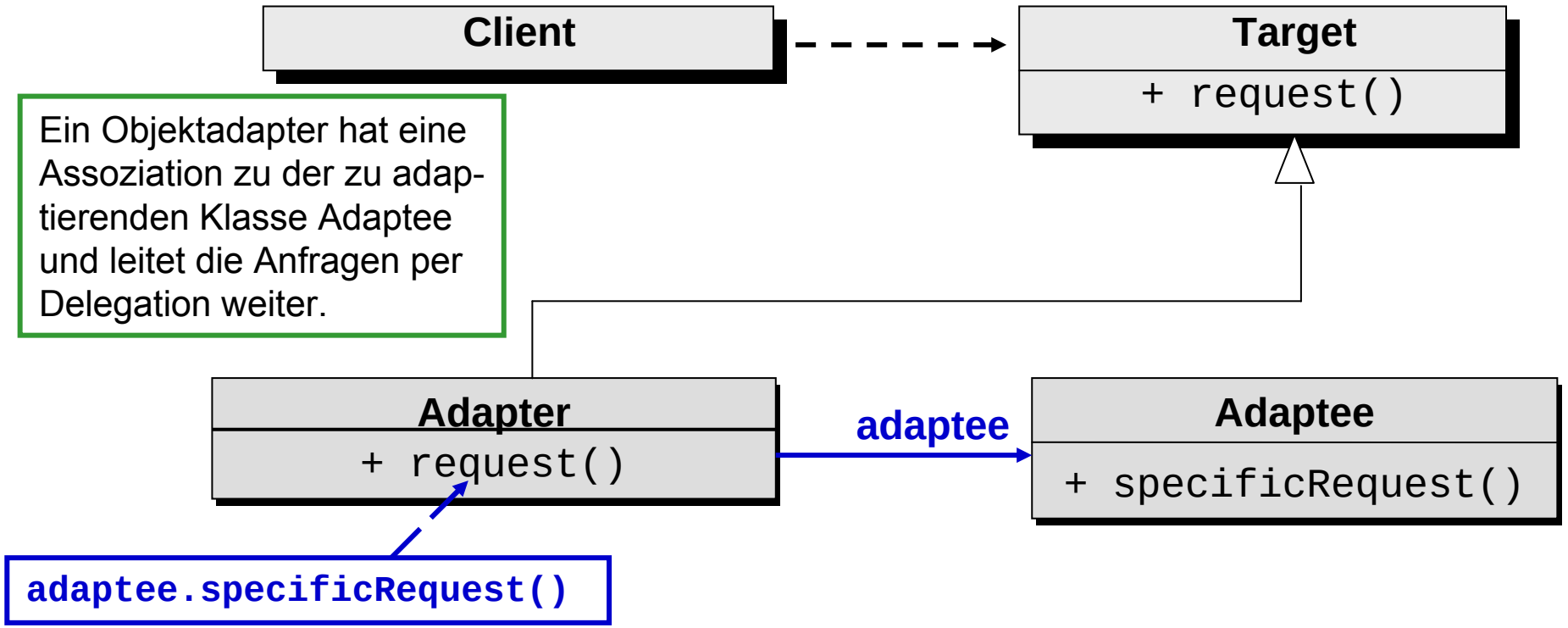
[aus Wikipedia]

- Das Muster der **Abstrakten Fabrik** ist eine Verallgemeinerung zum Austausch von Produktfamilien.
- **Z.B**. könnte eine Bank verschiedene Kontopakete, wie VIP, Normal, Internetangebot anbieten, die alle unterschiedliche Spar- und Girokonto-Bedingungen beinhalten.
- **Vorteile:** Einfacher Austausch von Produktfamilien
- **Nachteile:** Es ist schwierig, neue Produkte hinzuzufügen

Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn‘t otherwise because of incompatible interfaces. GoF(293)

Struktur des Objektadapters (Adapter mit Delegation):



Adapter

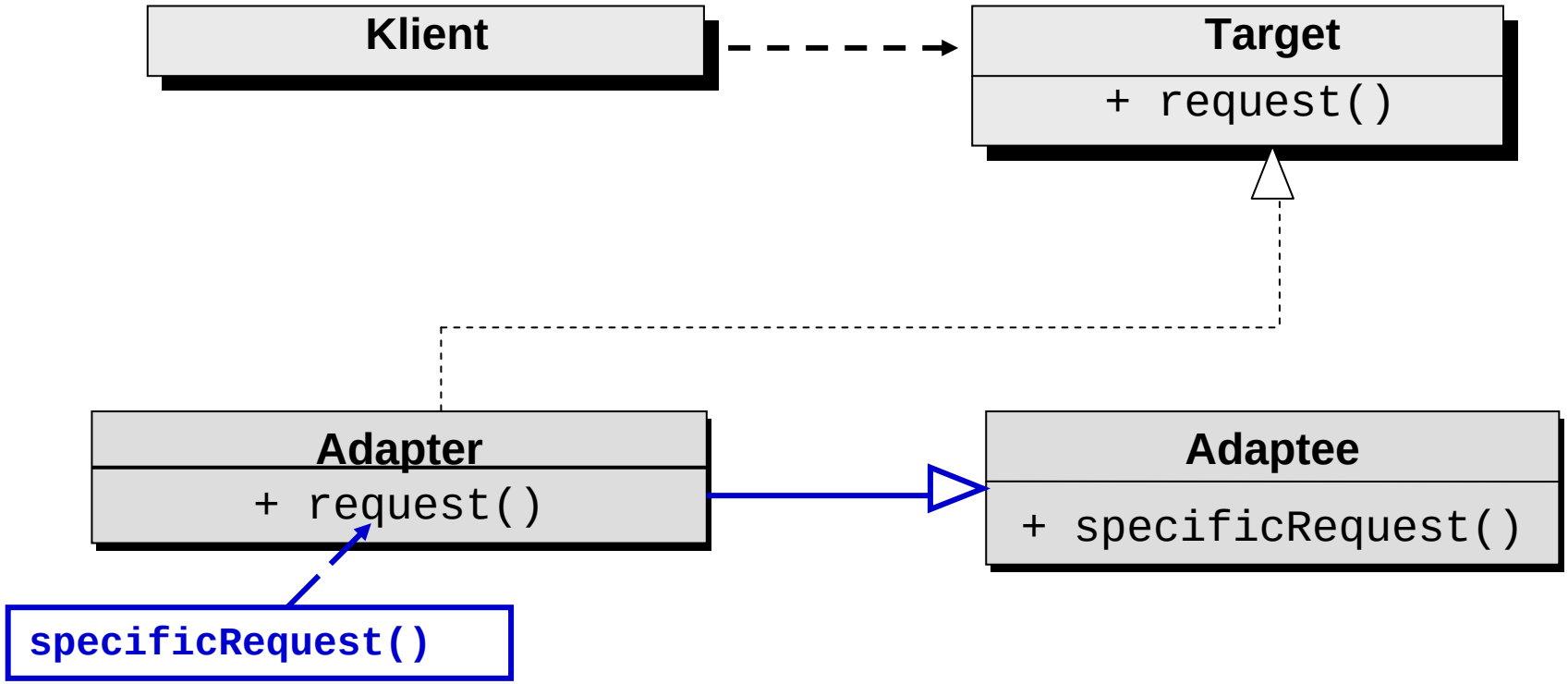
- Das **Adaptermuster (englisch *Adapter*, *Wrapper*)** übersetzt eine Schnittstelle in eine andere. Dadurch können Klassen miteinander kommunizieren, die zueinander inkompatible Schnittstellen zur Verfügung stellen.

- **Beteiligte Klassen**
  - **Adaptee (Dienst)**
    - bietet Dienstleistungen mit fest definierter Schnittstelle an (, die an die Zielschnittstelle angepasst werden soll)
  - **Client**
    - nutzt Dienste über eine inkompatible Schnittstelle und greift dabei auf die adaptierte Schnittstelle zurück
  - **Target (Ziel)**
    - definiert die Schnittstelle, die der Klient nutzen kann
  - **Adapter**
    - adaptiert die Schnittstelle des Dienstes (d.h. des Adaptee) auf die Schnittstelle des Ziels

Klassenadapter (Adapter mit Vererbung)

Eine Klassenadapter bildet eine Unterklasse der zu adaptierenden Klasse (`Adaptee`) und erbt damit die entsprechenden Methoden. In Java muss `Target` eine Schnittstelle sein.

Struktur des Klassenadapters (Adapter mit Vererbung):



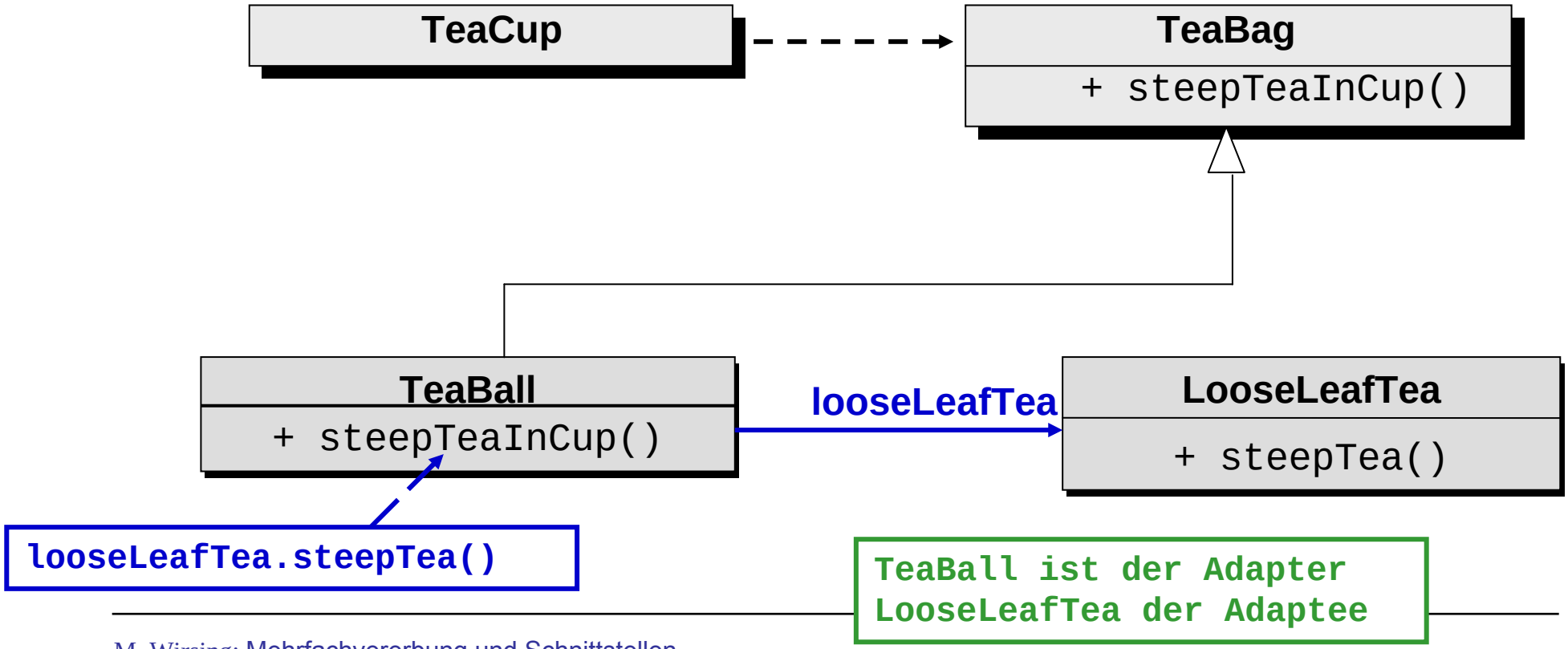
Adapter

- **Vorteile:**
  - Ein Klassenadapter passt genau eine Dienstklasse (Adaptee) an.
  - Ein Klassenadapter kann dadurch das Verhalten des Adaptees überschreiben.
  - Ein Klassenadapter wird eingesetzt, wenn ein Teil einer ganz konkreten Schnittstelle variiert werden soll.
  - Ein Objektadapter kann mehrere Dienstklassen (Adaptees) anpassen und alle Adaptees auf einmal um zusätzliche Operationen erweitern.
- **Nachteile:**
  - Für Objektadapter ist es schwieriger (als für Klassenadapter), das Verhalten des Adaptees zu überschreiben.
  - Klassenadapter lassen sich nur realisieren, wenn die Programmiersprache Schnittstellen oder Mehrfachvererbung unterstützt.

Beispiel für Objektadapter: Teetasse

In diesem Beispiel soll eine Tasse Tee mit einem Teebeutel zubereitet werden.

Falls nur loser Tee vorhanden ist, kann dieser mit Hilfe eines Teeballs an einen Teebeutel angepasst werden.



Beispiel für Objektadapter: Teetasse

- Das Klasse **`TeaCup`** verwendet den Dienst `steepTeaInCup` der Klasse **`TeaBag`** zur Zubereitung von Tee.

```
public class TeaCup {
  public TeaCup() {}
  public void steepTeaBag(TeaBag teaBag) {
    teaBag.steepTeaInCup();  }
}

public class TeaBag {
   protected boolean teaBagIsSteeped;
   public TeaBag() {
       teaBagIsSteeped = false;}

   public void steepTeaInCup() {
       teaBagIsSteeped = true;
       System.out.println("tea bag is steeping in cup");
   }
}
```

Beispiel für Objektadapter: Teetasse

- Die Klasse **`LooseLeafTea`** beschreibt losen Tee mit dem Dienst `steepTea`.
- Die Klasse **`TeaBall`** adaptiert diesen Dienst an den von `TeaCup` verwendeten Dienst `steepTeaInCup` der Klasse `TeaBag`.

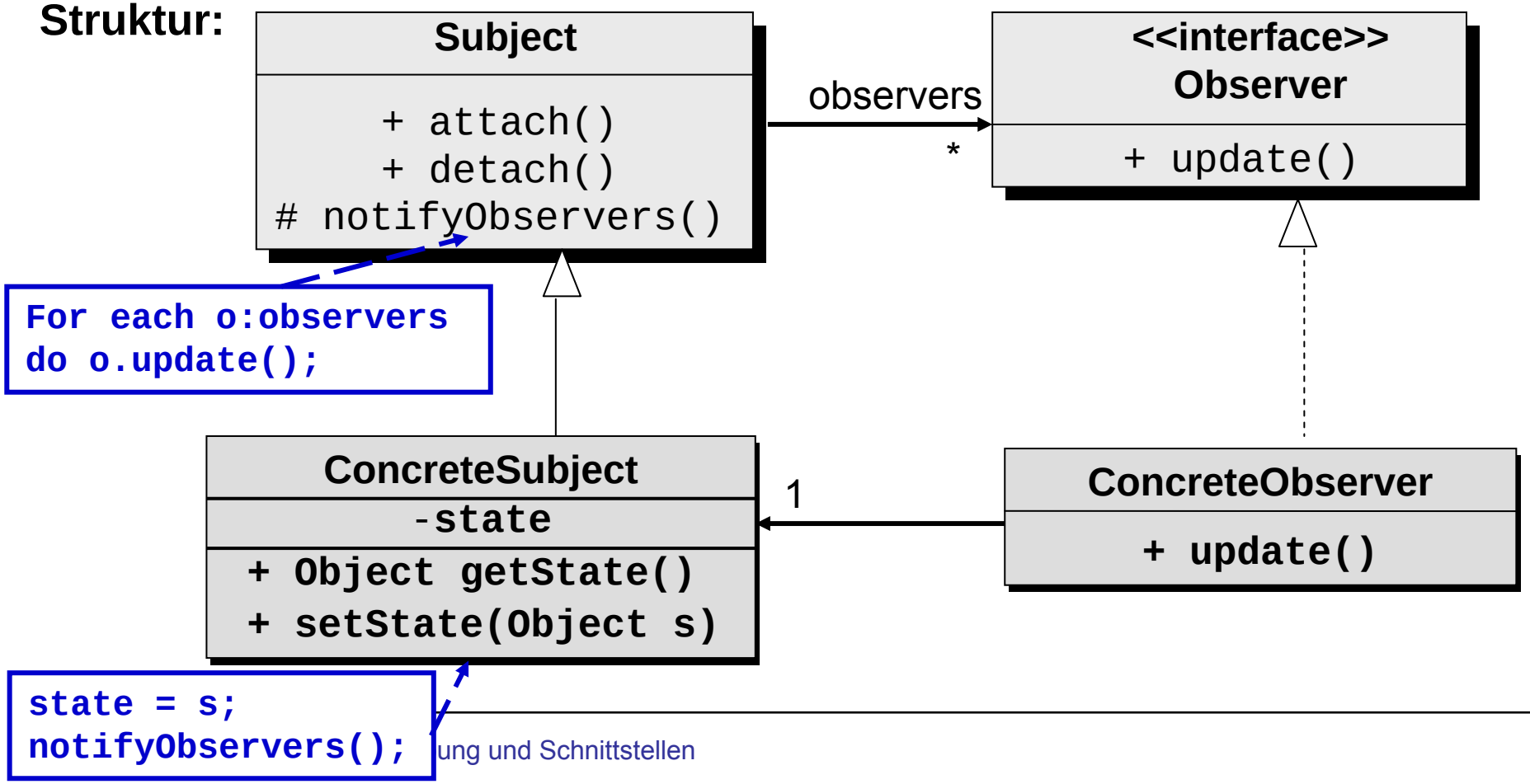
```
public class LooseLeafTea {
   private boolean teaIsSteeped;

   public LooseLeafTea() {
       teaIsSteeped = false;
   }

   public void steepTea() {
       teaIsSteeped = true;
       System.out.println(
            "tea is steeping");
   }

   public boolean isTeaSteeped(){
       return teaIsSteeped;   }
}
```

```
public class TeaBall extends TeaBag {
   private LooseLeafTea looseLeafTea;

   public TeaBall(){}

   public TeaBall(LooseLeafTea TeaIn) {
       looseLeafTea = TeaIn;
       teaBagIsSteeped =
            looseLeafTea.isTeaSteeped();
   }

   public void steepTeaInCup() {
       looseLeafTea.steepTea();
       teaBagIsSteeped = true;
   }
}
```

Beobachter (Observer)

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. GoF(293)

Struktur:



Beobachter (Observer)

- Das **Beobachtermuster** (Observer) ermöglicht die Weitergabe von Änderungen eines Objekts an abhängige Objekte.
- **Beteiligte Klassen**
  - **Subject (Beobachtbares Objekt, auch Publisher, also „Veröffentlicher“, genannt)**
    - Abstrakte Klasse zur Verwaltung einer beliebige Zahl von Observern
    - kennt Liste von Beobachtern, aber keine konkreten Beobachter
    - bietet Schnittstelle zur An- und Abmeldung von Beobachtern
    - bietet Schnittstelle zur Benachrichtigung von Beobachtern über Änderungen
  - **Observer (auch Subscriber, also „Abonnent“, genannt):**

    Interface für Objekte, die an Zustandsänderungen eines ConcreteSubject Objekts interessiert sind.
  - **ConcreteSubject (Konkretes, beobachtbares Objekt)**
    - Hält einen Zustand, an dem Observers Interesse haben, und benachrichtigt alle Beobachter bei Zustandsänderungen über deren Aktualisierungsschnittstelle
    - Schnittstelle zur Erfragung des aktuellen Zustands
  - **ConcreteObserver (Konkreter Beobachter):**

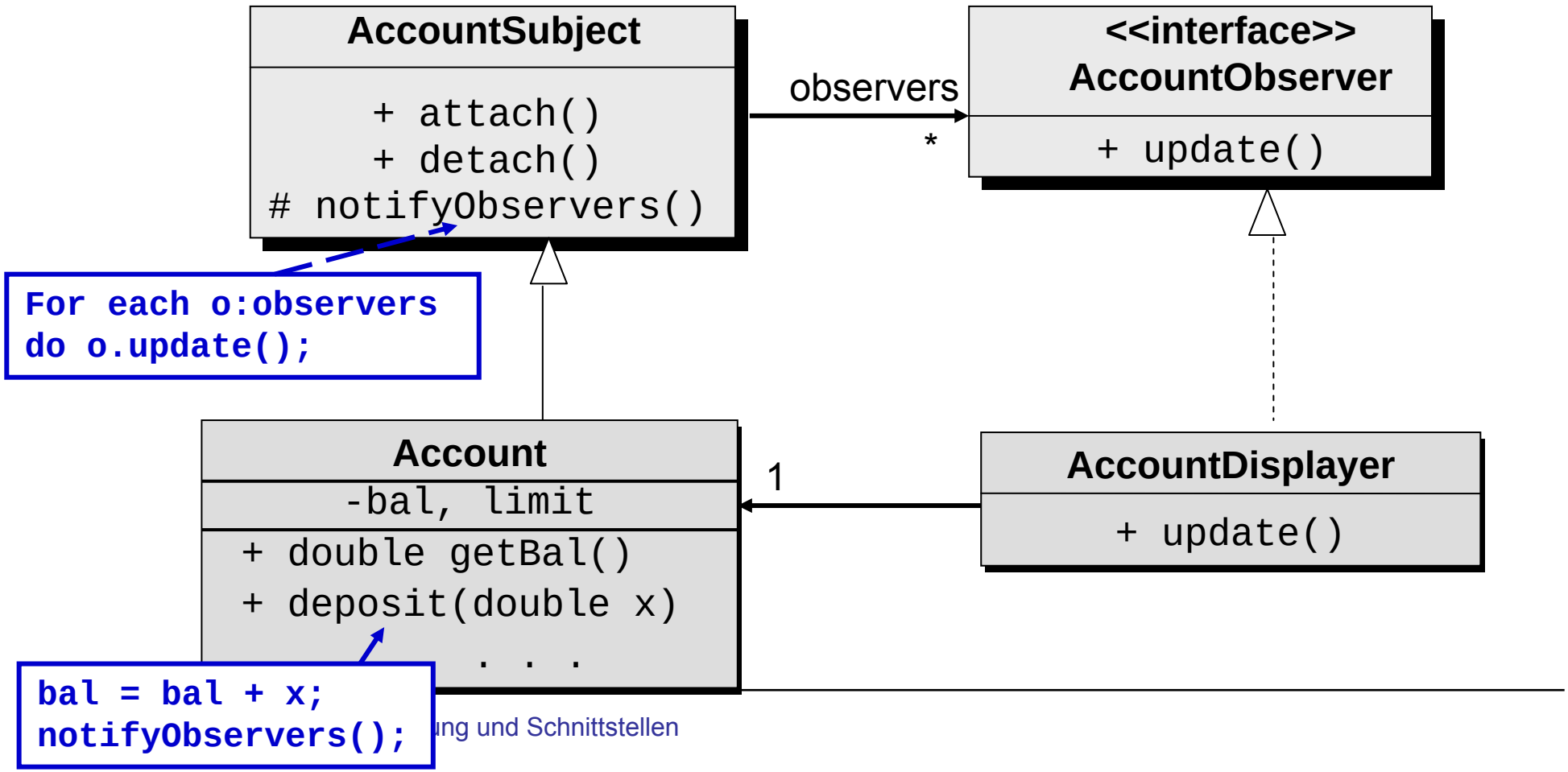
    Reagiert auf Zustandsänderungen der assoziierten Instanz von ConcreteSubject.

Beobachter (Observer)

- **Vorteile:**
  - Subjekte und Beobachter können unabhängig variiert werden.
  - Subjekt und Beobachter sind auf abstrakte und minimale Art lose gekoppelt. Das beobachtete Objekt braucht keine Kenntnis über die Struktur seiner Beobachter zu besitzen sondern kennt diese nur über die Beobachter-Schnittstelle.
  - Ein abhängiges Objekt erhält die Änderungen automatisch.
- **Nachteile:**
  - Änderungen am Objekt führen bei großer Beobachteranzahl zu hohen Änderungskosten. Einerseits informiert das Subjekt jeden Beobachter, auch wenn dieser die Änderungs-information nicht benötigt. Zusätzlich können die Änderungen weitere Änderungen nach sich ziehen und so einen unerwartet hohen Aufwand verursachen.
  - Ruft ein Beobachter während der Bearbeitung einer gemeldeten Änderung wiederum Änderungsmethoden des Subjektes auf, kann es zu Endlosschleifen kommen.
  - Der Mechanismus liefert keine Information darüber, was sich geändert hat. Die daraus resultierende Unabhängigkeit der Komponenten kann sich allerdings auch als Vorteil herausstellen.

Beispiel Beobachter: Kontobeobachtung

- Die Kontobewegungen werden von Observern beobachtet.



Beispiel Beobachter: Kontobeobachtung

- Die Klasse **`AccountSubject`** verwaltet eine Menge von Kontobeobachtern, kennt aber nur die Schnittstelle **`AccountObserver`**.

```
import java.util.HashSet;
import java.util.Set;

public class AccountSubject {
  private Set<AccountObserver> observers = new HashSet<AccountObserver>();

  private void notifyObservers() {
     for(AccountObserver observer : this.observers) {
        observer.update(); }
  }
  public void attach(AccountObserver observer) {
     this.observers.add(observer);
  }
  public void detach(AccountObserver observer) {
     this.observers.remove(observer);
  }
}
```

Beispiel Beobachter: Kontobeobachtung

- Das Klasse **`Account`** beschreibt sehr einfache, aber beobachtbare Konten

```
public class Account extends AccountSubject {
  private double bal = 0.0;
  private int limit = 0;

  public Account(double x, int l){
        bal = x; limit = l;}

  public void deposit (double x) {
    bal = bal+x;
    notifyObservers(); }

  public void withdraw (double x) {
    bal = bal-x;
    notifyObservers(); }

  public double getBal() { return bal;}
  public int getLimit() { return limit;}
}
```

Beispiel Beobachter: Kontobeobachtung

- Die Klasse **`AccountDisplayer`** ist ein Beobachter des Kontos, das über das Attribut `account` gegeben ist, und druckt nach jeder Kontobewegung den neuen Kontostand.

```
public interface AccountObserver {
  public void update();
}

public class AccountDisplayer implements AccountObserver{
  private Account account;

  public AccountDisplayer(){}

  public void update() {
        System.out.println("Neuer Kontostand = " + account.getBal());
  }
}
```

Beispiel Beobachter: Kontobeobachtung

- Die Klasse **AccountBenefactor** beschreibt ebenfalls Kontobeobachter. Sie spielt die Rolle eines Wohltäters, der bei zu geringem Kontostand einen Betrag auf das Konto überweist und so das Konto auf einen positiven Kontostand (101.0) bringt.

```
public class AccountBenefactor implements AccountObserver{
  private Account account;

  public AccountBenefactor(){}
  public void update() {
    double bal = account.getBal();
    int lim = account.getLimit();
    if (bal-lim<100){
        System.out.println("Der Wohltaeter ueberweist " + (101-bal));
        account.deposit(101-(int)bal);
  }
}
```

Zusammenfassung I

- **Entwurfsmuster** (engl. design pattern) sind **Regeln** oder **Richtlinien** (auch Muster), um häufig auftretende Probleme bei der Erstellung eines Programms zu lösen.
- Der **Nutzen** eines Entwurfsmusters liegt in der (programmiersprachen-unabhängigen) **Identifikation (Konzeptualisierung) einer Klasse von Entwurfsproblemen und der Beschreibung und Diskussion der Vor- und Nachteile einer Lösung** dafür.
- Die „Viererbande“ (Gamma, Helm, Vlissides und Johnson) unterscheidet drei Arten von **Entwurfsmustern: Erzeugungs-, Struktur- und Verhaltensmuster**. Z.B. sind Abstract Factory und Factory Method (Fabrikmethode) Erzeugungsmuster, Adapter ein Strukturmuster und Strategy und Observer Verhaltensmuster.

Zusammenfassung II

- Das **Fabrikmethodenmuster (Factory Method)** definiert eine Schnittstelle zur Erzeugung eines Objektes, wobei es den Unterklassen überlassen bleibt, von welcher Klasse das zu erzeugende Objekt ist.
- Das Muster der **Abstrakten Fabrik (Abstract Factory)** verallgemeinert das Fabrikmethodenmuster und dient zum Austausch von Produktfamilien.
- Das **Adaptermuster (Adapter, Wrapper)** übersetzt eine Schnittstelle in eine andere. Dadurch können Klassen miteinander kommunizieren, die zueinander inkompatible Schnittstellen zur Verfügung stellen.
- Das **Strategiemuster (Strategy)** entkoppelt Objekte von ihrem Verhalten und unterstützt den Austausch von Algorithmen.
- Das **Beobachtermuster (Observer)** ermöglicht die Weitergabe von Änderungen eines Objekts an abhängige Objekte.