ISW: Software Engineering WS 2015/16

Entwurfsmuster

Marcus Seiler
Institute of Computer Science
Chair of Software Engineering
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
marcus.seiler@informatik.uni-heidelberg.de

Entwurfsmuster

- Definition

- Musterarten + Beispiele
  - Erzeugungsmuster (Creational patterns)
  - Strukturmuster (Structural patterns)
  - Verhaltensmuster (Behavioral patterns)

- (vgl. Kapitel Entwurfsmuster aus Vorlesung)

Definition

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Christopher Alexander, 1977

Bestandteile eines Entwurfsmuster

- Name
  - Beschreibt das Entwurfsmuster, seine Lösungen und Konsequenzen in wenigen Worten
- Problem
  - Wann kann man das Entwurfsmuster anwenden?
  - Erklärt das Problem und seinen Kontext
- Lösung
  - Beschreibt die Elemente, ihre Beziehungen, ihre Verantwortlichkeiten und ihre Zusammenarbeit
- Diskussion
  - Beschreibt Vor- und Nachteile, Abhängigkeiten und Einschränkungen, Spezialfälle und bekannte Verwendung des Entwurfsmusters

Erzeugungsmuster

- Erzeugung von Objekten
  - versuchen Erzeugung zu verstecken, zu vereinheitlichen, zu vereinfachen
  - geben an, was erzeugt wird, wie es erzeugt wird und wann es erzeugt wird

- Beispiele
  - Einzelstück (Singelton)
  - Abstrakte Oberklasse
  - Fabrikmethode (Factory Method)
  - Prototyp
  - Builder

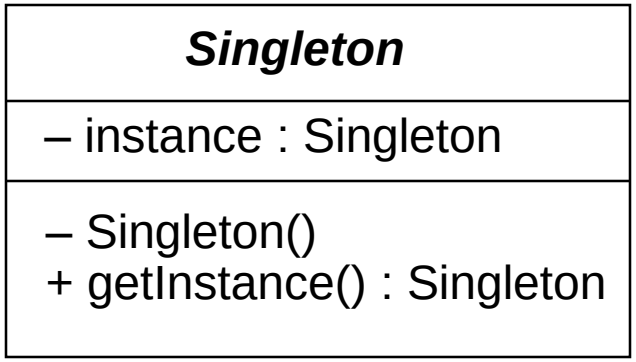
Einzelstück (1)

- engl. Singleton

- Problem:
  - Es soll von einer Klasse nur ein Objekt erzeugt werden
  - Das Objekt soll global verfügbar sein

- Lösung:
  - Erzeugung eines Einzelstücks, welche das Objekt nur ein einziges Mal erzeugt

Einzelstück (2)




```
public final class Singleton {

   private static Singleton instance;

   private Singleton() {}

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

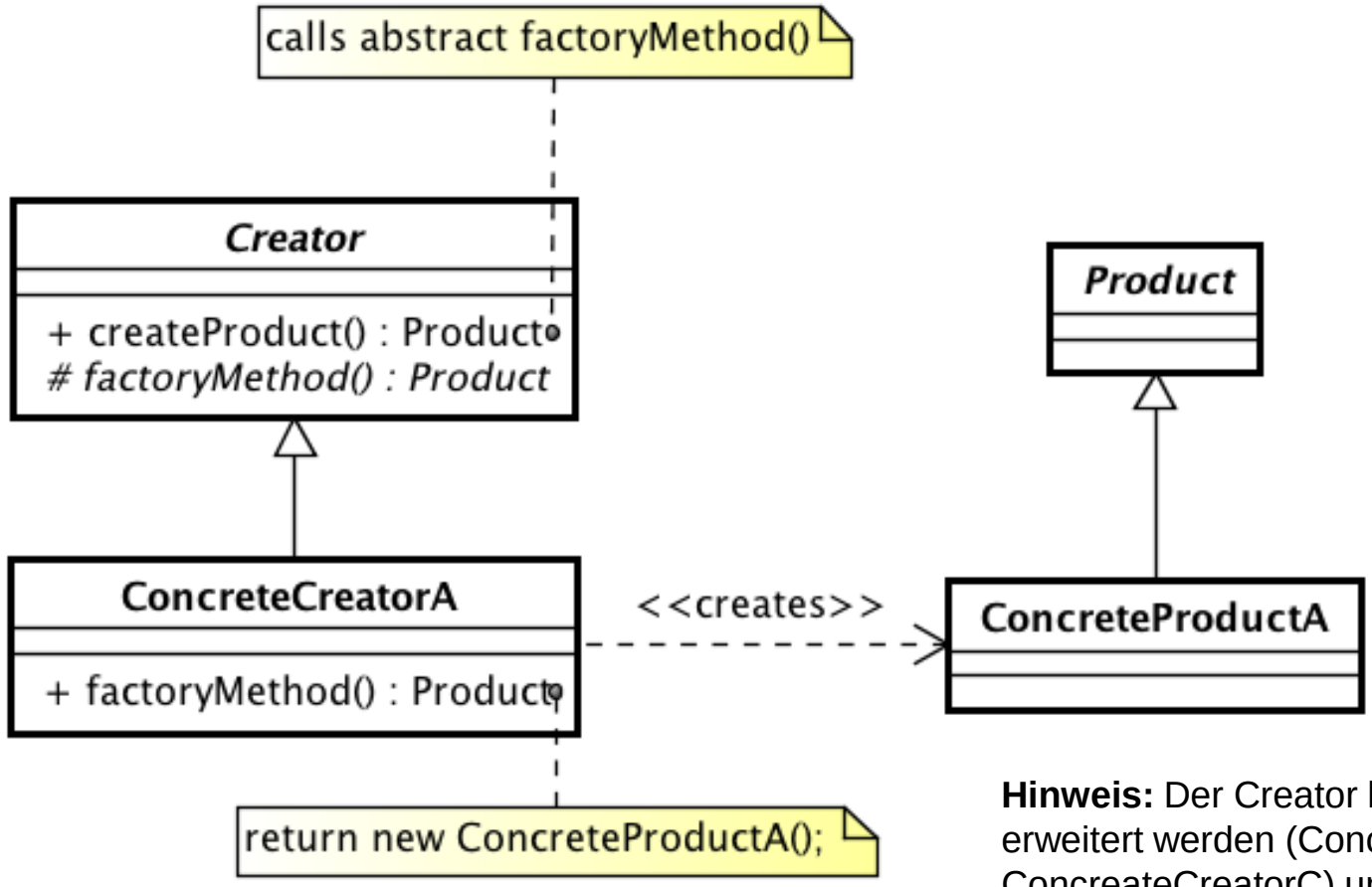
Quelle: http://www.oodesign.com/singleton-pattern.html

Fabrikmethode (1)

- engl. Factory Method

- Problem:
  - Bei Erzeugung von Objekten soll zwischen verschiedenen Varianten des zu erzeugenden Produkts gewählt werden
- Lösung:
  - Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.
  - Der Erstellungscode eines Objektes (Product genannt) wird in einer eigenen Klasse (Creator, Factory) ausgelagert.
  - Dieser Creator ist abstrakt und delegiert die konkrete Objektinstanziierung wiederum an seine Unterklassen. Erst die Unterklasse entscheidet, welches Product erstellt wird.

Fabrikmethode (2)



Hinweis: Der Creator kann beliebig erweitert werden (ConcreteCreatorB, ConcreateCreatorC) und somit verschiedene Products liefern.

Quelle: http://www.philipphauer.de/study/se/design-pattern/factory-method.php

Fabrikmethode (3)


```
public abstract class Creator {
    public Product createProduct() {
        Product product = factoryMethod();
        return product;
    }
    protected abstract Product factoryMethod();
}
```

```
class ConcreteCreatorA extends Creator {
    protected Product factoryMethod() {
        ConcreteProductA concProd = new
              ConcreteProductA();
        return concProd;
    }
}
```

```
public abstract class Product {
     public abstract int getPrice();
     // more methods of abstract Product
}
```

```
class ConcreteProductA extends Product {
    @Override
    public int getPrice() {
        return 42;
    }
    // must override all abstrat methods of Product
}
```

```
public class Client {
    public static void main(String[] args) {
        Creator creator = new ConcreteCreatorA();
        Product product = creator.createProduct();
        System.out.println(product.getPrice());
    }
}
```

Strukturmuster

- Befassen sich mit Komposition von Klassen zur Bildung größerer Strukturen

- Beispiele
  - Adapter
  - Kompositum (Composite)
  - Proxy
  - Bridge
  - Decorator
  - Fassade
  - Fliegengewicht (Flyweight)

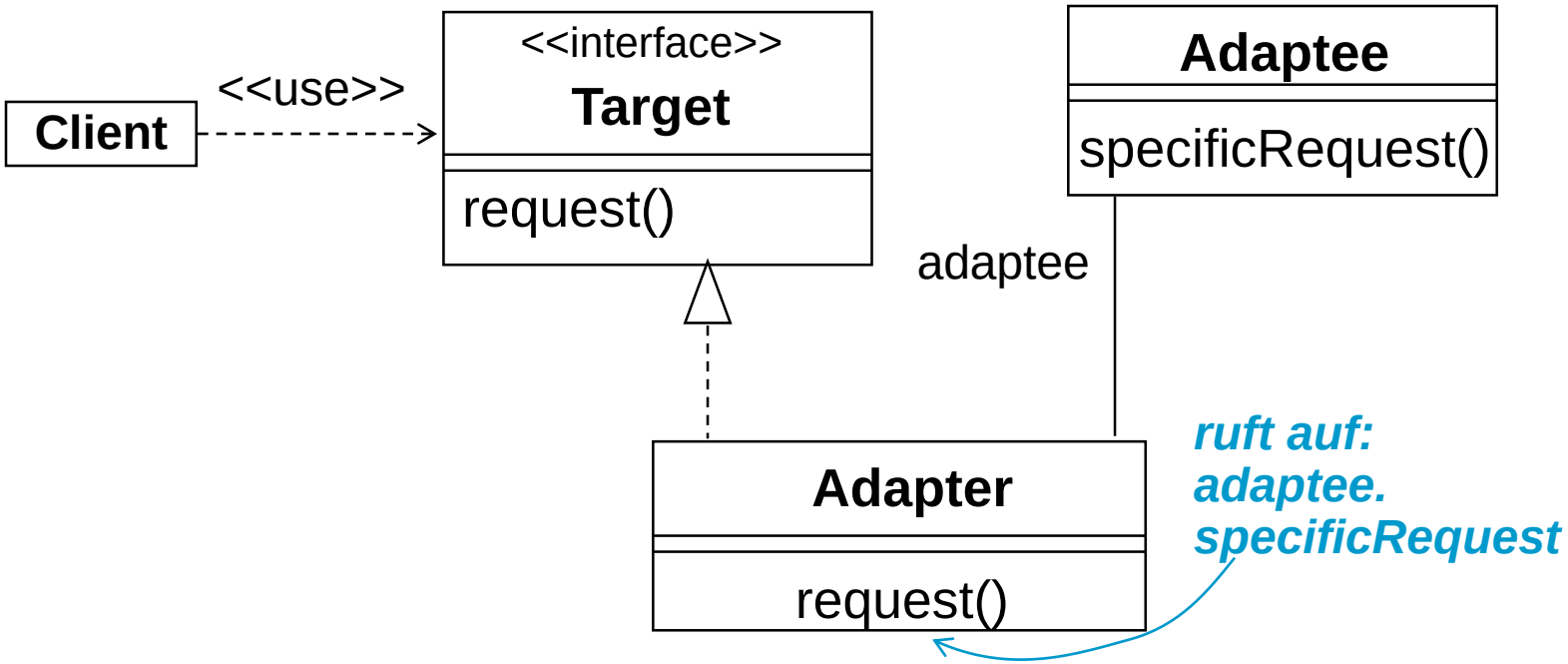
Adapter (1)

- Weiterer Name: Wrapper

- Problem:
  - Klassen können aufgrund inkompatibler Schnittstellen nicht zusammenarbeiten
  - Aber Schnittstellen haben große Überlappung

- Lösung:
  - Anpassung der Schnittstelle einer/s vorgegebenen/s Klasse/Objekts (adaptee) an erwartete Schnittstelle (target)

Adapter (2)

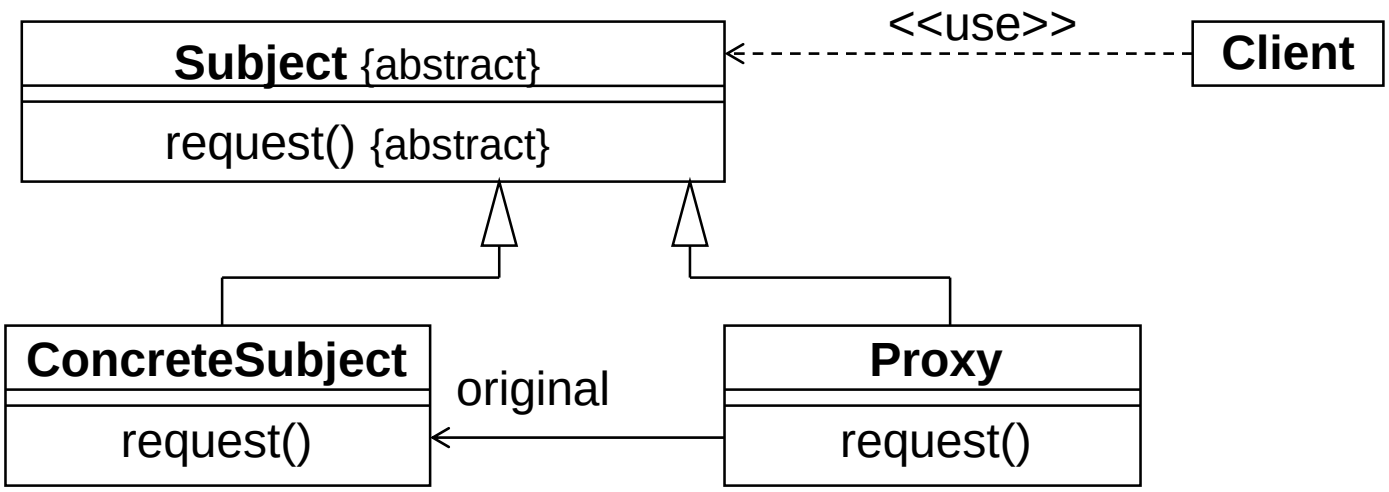


Quelle: http://www.oodesign.com/adapter-pattern.html

Proxy (1)

- Deutsch: Stellvertreter
- Problem:
  - Direkter Zugriff auf ein Objekt problematisch:
    - Großer Aufwand, Sicherheitsprobleme
- Lösung:
  - Einführung eines Stellvertreter - Objekts

Proxy (2)



Quelle: http://www.oodesign.com/proxy-pattern.html

Proxy (3)


```
abstract class Subject {
    …
    abstract void request() ;
    …
}
```

```
class ConcreteSubject extends Subject {
    public void request() {
        doSomethingElse();
        …
    }
    …
}
```

```
class Client {
    private Proxy proxy;
    …
    public void doSomething() {
        proxy.request();
    }
}
```

```
class Proxy extends Subject {
    private ConcreteSubject
        concreteSubject;

    Proxy (ConcreteSubject cs ) {
        this.concreteSubject = cs;
    }

    public void request() {
        if(isAllowed()) {
            concreteSubject.request();
        }
        else {
            errorMessage(„Access Denied“);
        }
    }
    …
}
```

Verhaltensmuster

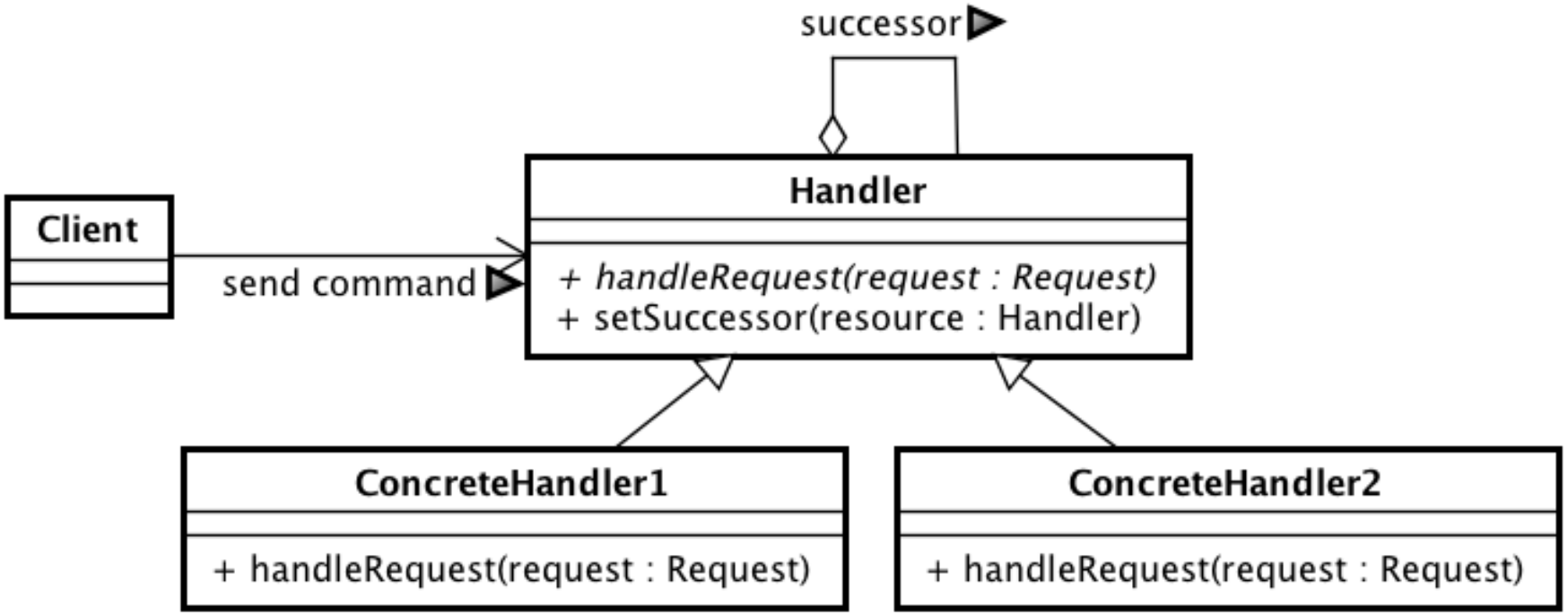
- Beschäftigen sich mit Algorithmen und der Zuweisung von Zuständen zu Objekten
- Beschreiben neben Klassen und Objekten auch die Interaktion zwischen Ihnen
- Beispiele
  - Zuständigkeitskette (Chain of Responsibility)
  - Befehl (Command)
  - Interpreter
  - Vermittler (Mediator)
  - Memento
  - Iterator
  - Beobachter (Observer)
  - Zustand (State)
  - Strategie
  - Schablonenmethode (Template Method)
  - Besucher (Visitor)

Zuständigkeitskette (1)

- engl. Chain of Responsibility

- Problem:
  - Ein Ereignis (Event), welches von einem Objekt erzeugt wird, muss von einem anderen Objekt verarbeitet werden. Zusätzlich hat man keinen Zugriff auf das Objekt, welches das Ereignis verarbeiten soll.
- Lösung:
  - Eine Zuständigkeitskette (engl. “Chain of Responsibility”), welches einem Objekt erlaubt ein Ereignis zu erzeugen, ohne zu wissen, welches Objekt das Ereignis empfängt und verarbeitet.
  - Die Objekte werden Teil einer Zuständigkeitskette
  - Der Aufruf wird solange die Zuständigkeitskette entlang weitergereicht, bis ein Objekt den Aufruf bearbeiten kann.

Zuständigkeitskette (2)



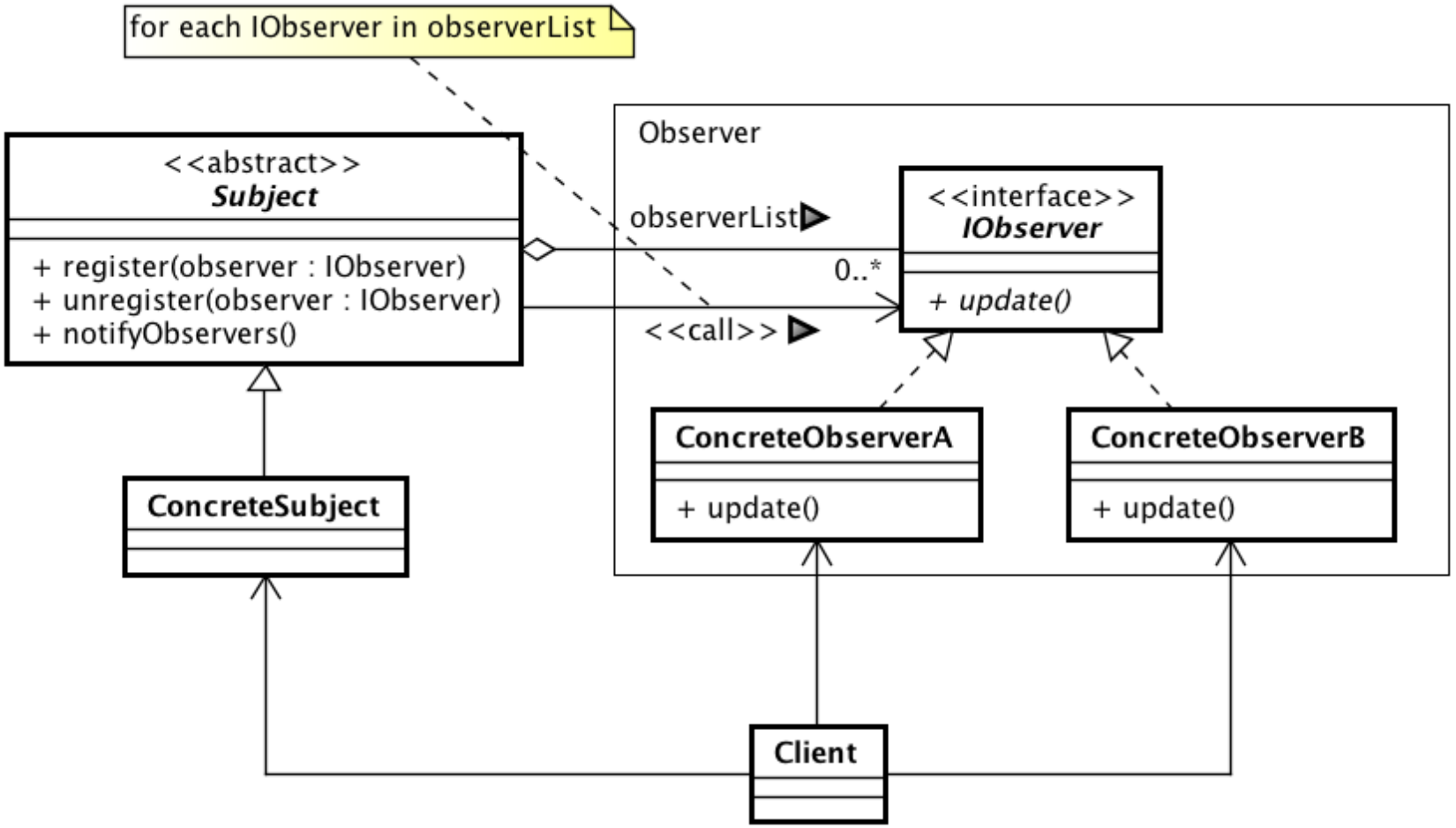
Quelle: http://www.oodesign.com/chain-of-responsibility-pattern.html

Beobachter (1)

- engl. Observer

- Problem:
  - Objekte (Beobachter) wollen über Änderungen an einem anderen Objekt (Subjekt) informiert werden
- Lösung:
  - Für die Beobachter wird eine einheitliche Schnittstelle mit mindestens einer Aktualisierungsmethode (update) definiert, welche vom Subjekt im Falle von Aktualisierungen aufgerufen (notifyObserver) wird.
  - Konkrete Beobachter implementieren die Schnittstelle und damit die Aktualisierungsmethode und bestimmen somit, wie sie auf die Benachrichtigung reagieren.
  - Das Subjekt benötigt Administrationsmethoden, damit sich Beobachter an- und abmelden können.

Beobacher (2)



Quelle: http://www.philipphauer.de/study/se/design-pattern/observer.php

Beobachter (3)


```
public abstract class Subject {
    private final List<IObserver> observerList =
            new ArrayList<IObserver>();
    public void register(IObserver pNewObserver){
        observerList.add(pNewObserver);
    }
    public void unregister(IObserver pNewObserver){
        observerList.remove(pNewObserver);
    }
    protected void notifyObservers(int pState){
        for (IObserver observer : observerList) {
            observer.update(pState);
        }
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        ConcreteSubject concreteSubject =
            new ConcreteSubject();
        concreteSubject.register(new ConcreteObserverA());
        concreteSubject.setState(42);
        //Concrete Observer A is updated with 42
    }
}
```

```
public class ConcreteSubject extends Subject {
    private int state;
    public void setState(int pState) {
        this.state = pState;
        notifyObservers(pState);
    }
    public int getState() {
        return state;
    }
}
```

```
public interface IObserver {
    public void update(int pState);
}
```

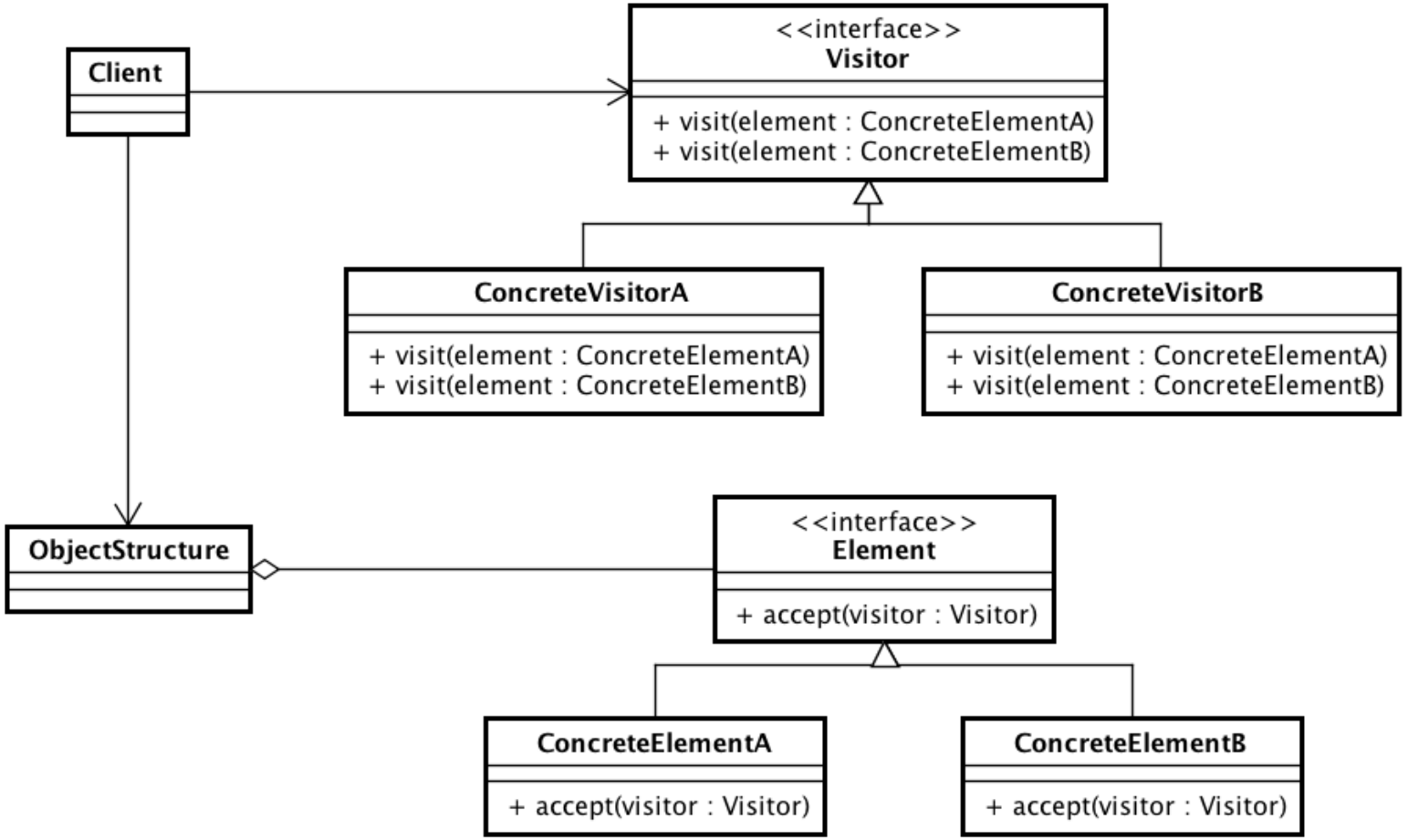
```
public class ConcreteObserverA implements IObserver {
    public void update(int state) {
        System.out.println("Concrete Observer A is updated with "
            + state);
    }
}
```

Besucher (1)

- engl. Visitor

- Problem:
  - Eine Objektstruktur enthält viele Objekte (Element) aus unterschiedlichen Klassen und man möchte eine oder mehrere Operationen auf diesen Objekten ausführen, die aber von den konkreten Klassen abhängen.
- Lösung:
  - Schnittstelle Besucher (Visitor) deklariert eine visit()-Operation für jede ConcreteElement-Klasse in der Objektstruktur. Der Parameter benennt die aufzurufende Klasse. Dies ermöglicht dem Besucher, eine Operation für unterschiedliche Klassen unterschiedlich auszuführen.
  - Spezifische Besucher (ConcreteVisitor) implementieren die Schnittstelle Besucher .
  - Schnittstelle Element deklariert eine accept()-Operation, damit sich Besucher registrieren können. Diese Operation ruft dann die visit()-Operation des spezifischen Besuchers auf.
  - Spezifische Elemente (ConcreteElement) implementieren die Schnittstelle Element.

Besucher (2)



Quelle: http://www.oodesign.com/visitor-pattern.html

Besucher (3)

```
public interface Visitor {
   public void visit(ConcreteElementA a);
   public void visit(ConcreteElementB b);
}
```

```
public class ConcreteVisitorA implements Visitor {
   public void visit(ConcreteElementA a) {
      System.out.println("ConcreteVisitorA " + a.getName());
   }
   public void visit(ConcreteElementB b) {
      System.out.println("ConcreteVisitorA " + b.getName());
   }
}
```

```
public class ConcreteVisitorB implements Visitor {
   public void visit(ConcreteElementA a) {
      System.out.println("ConcreteVisitorB " + a.getName());
   }
   public void visit(ConcreteElementB b) {
      System.out.println("ConcreteVisitorB " + b.getName());
   }
}
```

```
public interface Element {
   public void accept(Visitor v);
}
```

```
public class ConcreteElementA implements Element {
   public void accept(Visitor v) {
     v.visit(this);
   }
   public String getName() {
      return "ConcreteElementA";
   }
}
```

```
public class ConcreteElementB implements Element {
   public void accept(Visitor v) {
     v.visit(this);
   }
   public String getName() {
      return "ConcreteElementB";
   }
}
```

```
public class VisitorDemo {
   public static Element[] list = {new ConcreteElementA(),
        new ConcreteElementB()};
   public static void main( String[] args ) {
      ConcreteVisitorA visitorA = new ConcreteVisitorA();
      ConcreteVisitorB visitorB = new ConcreteVisitorB();
      for (int i=0; i < list.length; i++) {
         list[i].accept(visitorA);
      }
      for (int i=0; i < list.length; i++) {
         list[i].accept(visitorB);
      }
   }
}
```

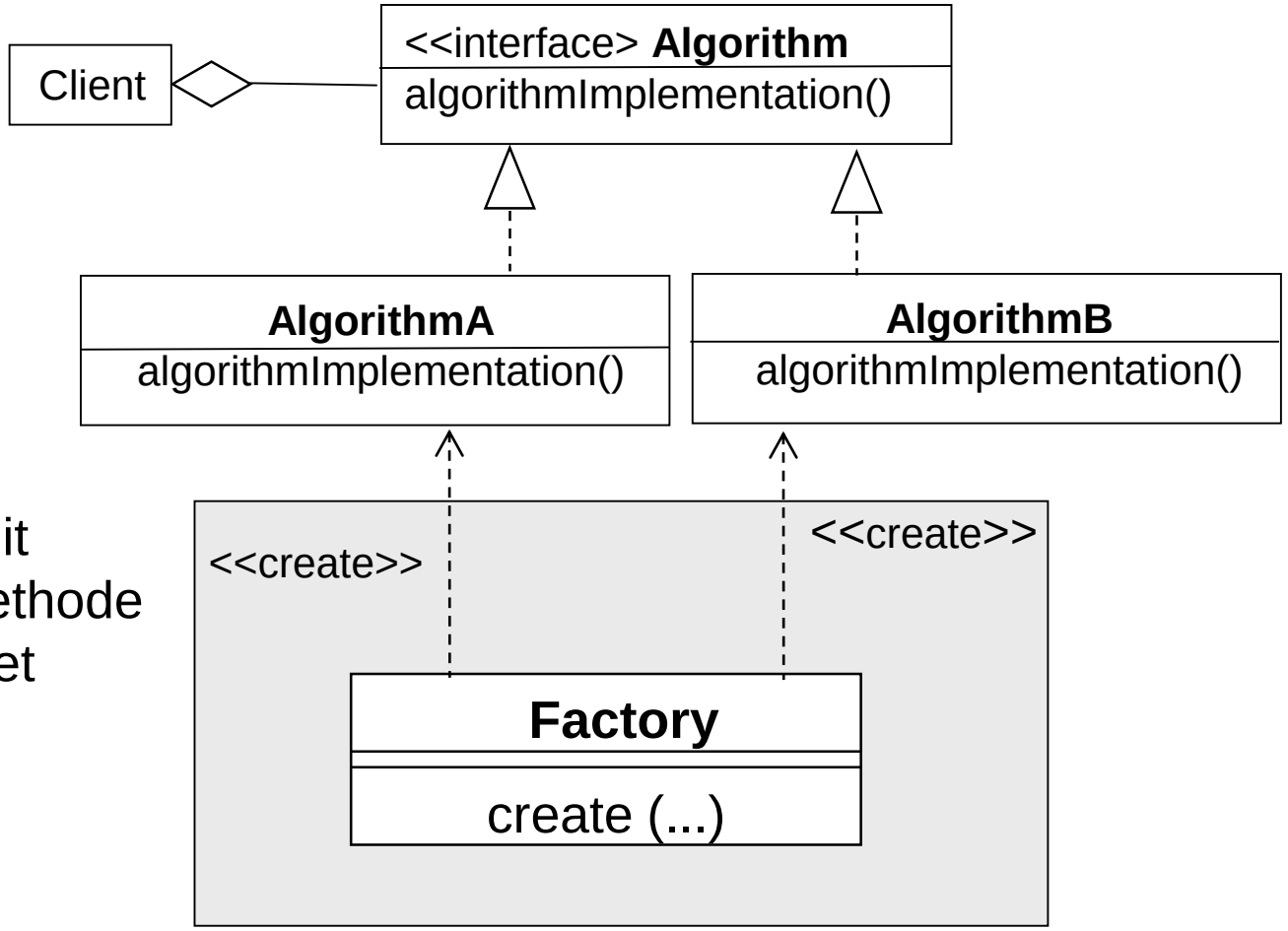
Strategie (1)

- engl. Strategy

- Problem:
  - Algorithmen austauschbar gestalten (weil Algorithmus kontextabhängig)
  - Clients interessieren die Vorgehensweise des Algorithmus nicht

- Lösung:
  - Definition einer geeigneten Schnittstelle um Algorithmus zu kapseln
  - Kapselung jedes einzelnen Algorithmus in eigene Klasse

Strategie (2)



- Häufig mit Fabrikmethode verwendet

Zusammenfassung

- Entwurfsmuster geben Erfahrungswissen wieder
- Geben Lösungsvorschläge für eine Menge ähnlicher Probleme
- 3 Arten von Entwurfsmuster
  - Erzeugungsmuster
  - Strukturmuster
  - Verhaltensmuster

Literatur

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  “Design Patterns: Elements of Reusable Object-Oriented Software” Addison Wesley, 1994
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal "Pattern-Oriented Software Architecture Volume 1: A System of Patterns" Wiley, 1996
- Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann "Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects" Wiley, 2000
- Martin Fowler "Patterns of Enterprise Application Architecture" Addison Wesley, 2002
- Steve McConnel "Code Complete: A Practical Handbook of Software Construction", Microsoft Press, 2004
- Philipp Hauer, http://www.philipphauer.de/study/se/design-pattern.php

Marcus Seiler
Institute of Computer Science
Chair of Software Engineering
Im Neuenheimer Feld 326
69120 Heidelberg, Germany
http://se.ifi.uni-heidelberg.de
marcus.seiler@informatik.uni-heidelberg.de

RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG