

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

## Softwaretechnik



### Kapitel 6



1. Wieso Software Engineering?
2. Vom Einzelkämpfer zum Großprojekt

#### **Systematische Softwareentwicklung**

3. Anforderungen und Test: Basis des Projekts
4. Entwurf: Strukturen und nicht-funktionale Eigenschaften
5. Entwürfe notieren mit UML: Modelle im SE
- 6. Design Patterns: Entwurfserfahrungen nutzen**
7. Management von Technik und Projekt

Leibniz Universität Hannover

SWT 2015/16 · 231

### Inhalt von Kapitel 6



#### Systematische Softwareentwicklung

##### 6. Design Patterns: *Entwurfserfahrungen nutzen*

- Idee der Design Patterns
- Beispiel „Adapter“-Pattern
- Beispiel „Composite“-Pattern
- Beispiel „Observer“-Pattern
- Model-View-Controller

Leibniz Universität Hannover

SWT 2015/16 · 232

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Design Patterns

- Prinzip: Entwurfserfahrungen verpacken, wiederholt nutzen
- Ansatz
  - Gewisse Probleme im Entwurf tauchen immer wieder auf
  - Erfahrene Entwickler finden auch ähnliche, bewährte Lösungen
  - Idee: Problem-Lösungs paar wird als Muster beschrieben
  - Wenn das Problem noch einmal auftaucht, gleich die Lösung einsetzen
- Vorteile
  - Erfahrung sind in Patterns „codiert“, nicht im Hirn
  - Anwendung auch für nicht so erfahrene Entwerfer möglich
  - Nur das beste Muster muss man sich merken (Auslese)
- Hinweis: Muster sind meist abstrakter als Code
  - Daher eher UML-Muster als Code. Nur Fragmente von Code.
  - Vorteil: Nicht nur in einer Programmiersprache anwendbar
  - Problem ist nur teil-formal beschrieben, Entwickler muss es erkennen

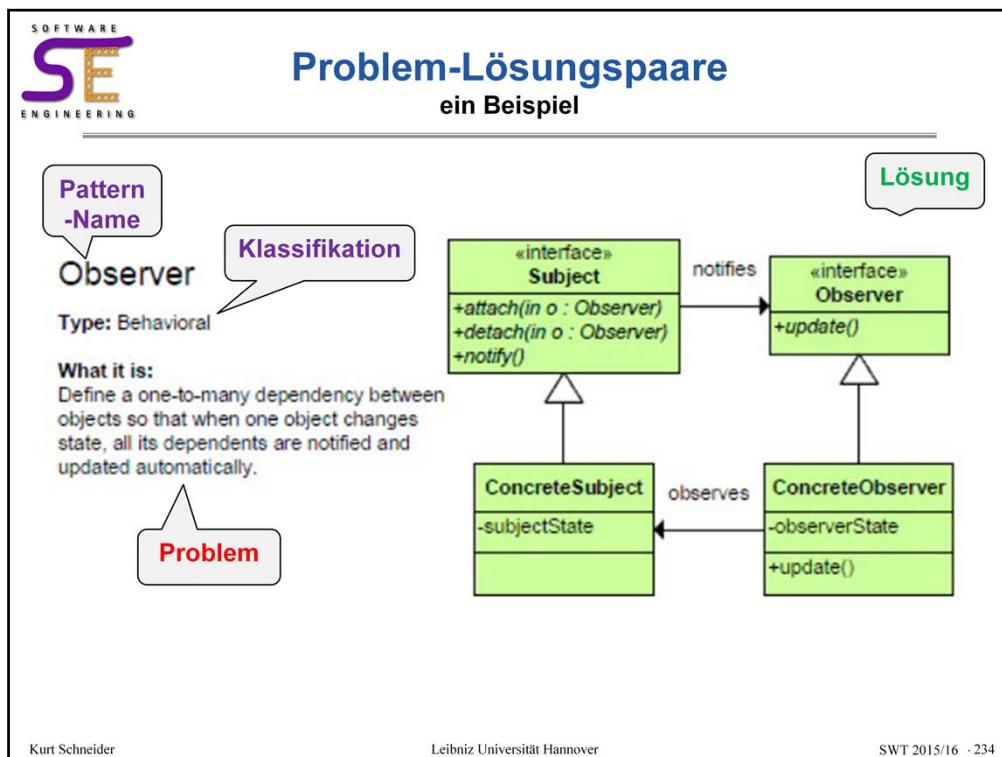
Kurt Schneider

Leibniz Universität Hannover

SWT 2015/16 · 233

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Ein Pattern besteht im Kern aus einem Problem und einer Lösung.

Ähnliche Probleme sollen mit ähnlichen Lösungen angegangen werden.

Die Beschreibungen sind so konkret wie möglich, um bei der Lösung zu helfen. Aber sie sind gleichzeitig so allgemein wie nötig, damit viele ähnliche Probleme auch mit dem Pattern erschlagen werden können.

Die Lösung wird oft als UML-Diagramm dargestellt, an dem ein Teil besonders wichtig ist und die Lösung ausmacht. Das wird dann textuell beschrieben.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

**SE**  
SOFTWARE  
ENGINEERING

## Flexibilität durch Adapter: Beispiel Wetterstation

- Messstationen in vielen Städten
- Werden automatisiert abgefragt
  - Temperatur, Luftdruck etc.
  - Explizites Logging
  - Zeichnen Verlaufskurven auf

Probleme treten auf,  
als USA dazukommt ...



Software zum Werteabruf:

```
public class Thermometer {  
    ...  
    public double temperature () { ... }  
    public void logNow (Time now) {...}  
}
```

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 235

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Problem: Wie vereinheitlichen?

**Messzentrale will u.a.:**

- Einzelne Temperaturen abrufen
- Alle Temperaturen abrufen
- Alle Stationen zum Loggen auffordern

**Und insbesondere**

- Dabei nicht zwischen D- und US-Stationen unterscheiden müssen
- Bei nächster Anwendung nicht wieder darüber nachdenken müssen, wie man Fahrenheit umrechnet.



Kurt Schneider

Leibniz Universität Hannover

SWT 2015/16 · 236

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

**Lösungsidee: Pattern “Adapter”**

**Problem**  
Client möchte von einer Klasse eine Methode benutzen, die diese nicht anbietet.

**Blauer Pfeil: kein UML**

```
graph LR; Client --> Target[Target<br/>Request()]; Target --> Adaptee[Adaptee<br/>SpecificRequest()]; Client -.-> Adapter[Adapter<br/>Request(), SpecificRequest()]; Adapter -.-> Adaptee;
```

**Lösung: Adapter**  
**Klassen-Adapter (li)**

- Gibt einer Klasse ein neues Methoden-Interface
- Erbt/implementiert Interface von Target, erbt echte Methoden von Adaptee

**Objekt-Adapter (re)**

- Erbt nicht von Adaptee
- Benutzt aber adaptee-Objekte, um Request () doch noch auszuführen

Nach Gamma et al. (1994) Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley  
Leibniz Universität Hannover

SWT 2015/16 · 237

Der Adapter ist oft eher als „Wrapper“ bekannt. Hier ist also der allgemeine Ablauf bzw. die Struktur, die hinter dem Drucker-Beispiel stand.

Die blaue Linie zeigt den logischen Ablauf:

Der Client schickt „Request()“ an ein Target. Target ist aber als definierte Schnittstelle eingeführt worden und damit eine *abstrakte Oberklasse*. Weil ihre Methoden nicht ausgeführt sind, müssen konkrete Unterklassen das übernehmen. Adapter ist eine solche konkrete Unterklasse. Im Endeffekt kennt der Client also nur Instanzen von Adapter.

Wenn diese den „Request()“ erhalten, bearbeiten sie ihn, indem sie ihn an eine *eigene* Methode mit anderem Namen (hier SpecificRequest()) weitergeben.

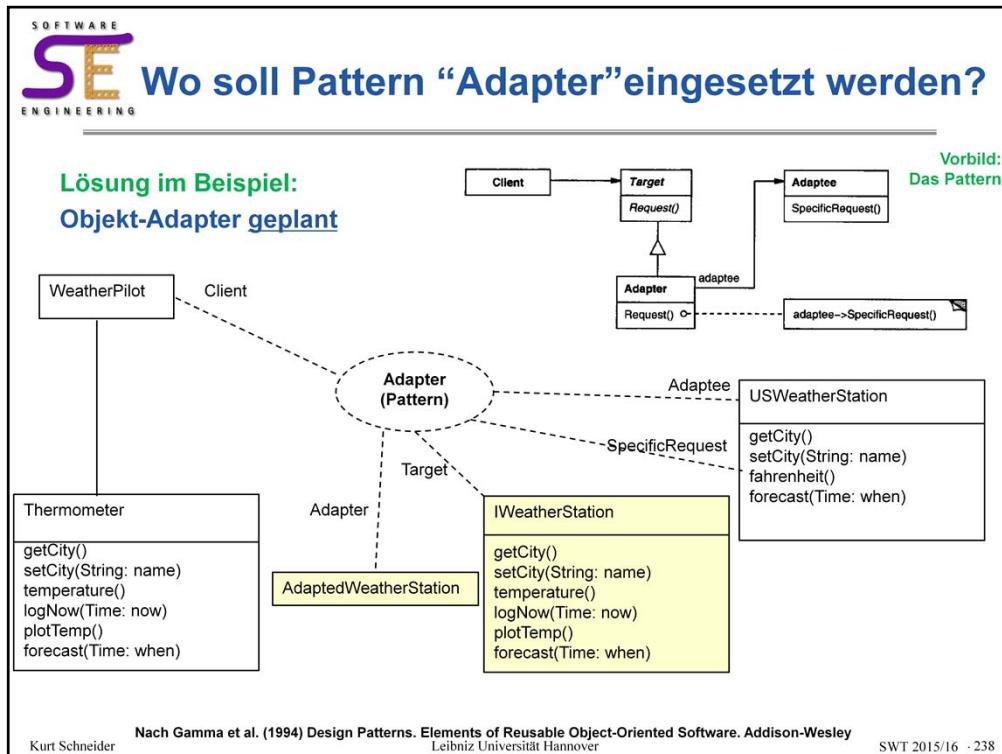
Der *Adaptee* (der sich anpassen lässt, bzw. um den eine Hülle, der *Wrapper* gezogen wird) kennt nämlich nicht den Request(), wohl aber den SpecificRequest().

Indem nun Adapter die Schnittstelle zum Kunden (Client) von Target erbt – nur dafür ist Target da –, andererseits aber die Implementierung unter anderem Namen von Adaptee, vermittelt Adapter zwischen dem Client und einem versteckten Leistungserbringer (Adaptee).

Was bisher auf Klassenebene beschrieben wurde, kann auch ohne Vererbung von Adaptee auskommen. Es reicht, den Request() an eine Instanz von Adaptee als SpecificRequest() weiterzureichen. Diese Instanz *adaptee* muss sich der Adapter dann eben beschaffen.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Der Adapter ist oft eher als „Wrapper“ bekannt. Hier ist also der allgemeine Ablauf bzw. die Struktur, die hinter dem Drucker-Beispiel stand.

Die blaue Linie zeigt den logischen Ablauf:

Der Client schickt „Request()“ an ein Target. Target ist aber als definierte Schnittstelle eingeführt worden und damit eine *abstrakte Oberklasse*. Weil ihre Methoden nicht ausgeführt sind, müssen konkrete Unterklassen das übernehmen. Adapter ist eine solche konkrete Unterklasse. Im Endeffekt kennt der Client also nur Instanzen von Adapter.

Wenn diese den „Request()“ erhalten, bearbeiten sie ihn, indem sie ihn an eine *eigene* Methode mit anderem Namen (hier SpecificRequest()) weitergeben.

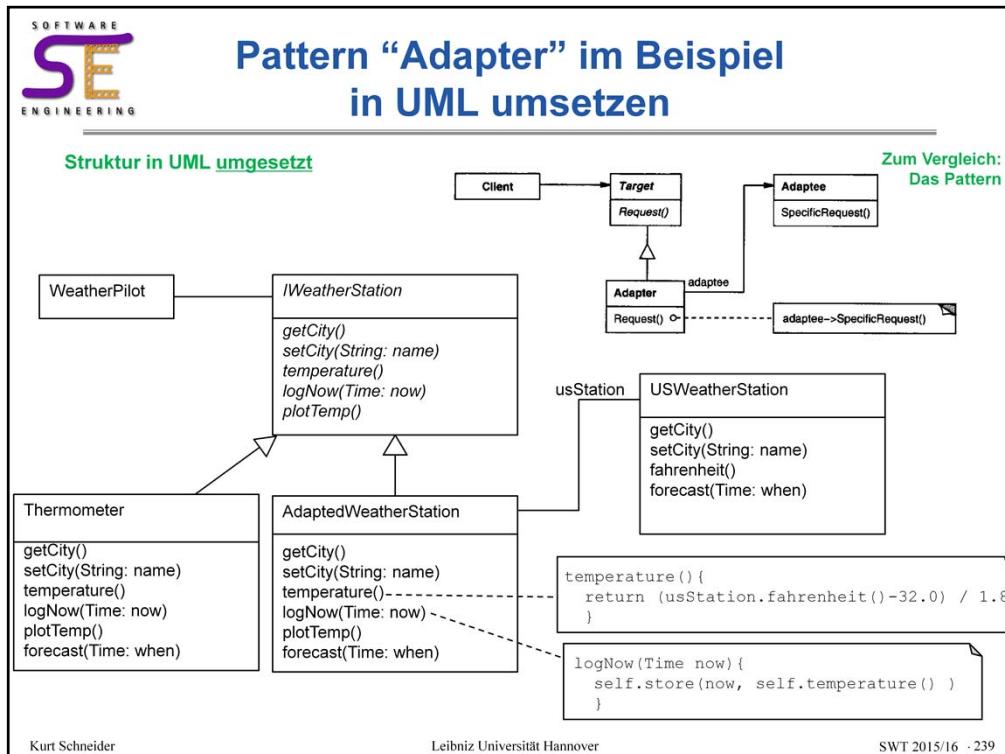
Der *Adaptee* (der sich anpassen lässt, bzw. um den eine Hülle, der *Wrapper* gezogen wird) kennt nämlich nicht den Request(), wohl aber den SpecificRequest().

Indem nun Adapter die Schnittstelle zum Kunden (Client) von Target erbt – nur dafür ist Target da –, andererseits aber die Implementierung unter anderem Namen von Adaptee, vermittelt Adapter zwischen dem Client und einem versteckten Leistungserbringer (Adaptee).

Was bisher auf Klassenebene beschrieben wurde, kann auch ohne Vererbung von Adaptee auskommen. Es reicht, den Request() an eine Instanz von Adaptee als SpecificRequest() weiterzurichten. Diese Instanz adaptee muss sich der Adapter dann eben beschaffen.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Der Adapter ist oft eher als „Wrapper“ bekannt. Hier ist also der allgemeine Ablauf bzw. die Struktur, die hinter dem Drucker-Beispiel stand.

Die blaue Linie zeigt den logischen Ablauf:

Der Client schickt „Request()“ an ein Target. Target ist aber als definierte Schnittstelle eingeführt worden und damit eine *abstrakte Oberklasse*. Weil ihre Methoden nicht ausgeführt sind, müssen konkrete Unterklassen das übernehmen. Adapter ist eine solche konkrete Unterklasse. Im Endeffekt kennt der Client also nur Instanzen von Adapter.

Wenn diese den „Request()“ erhalten, bearbeiten sie ihn, indem sie ihn an eine *eigene* Methode mit anderem Namen (hier *SpecificRequest()*) weitergeben.

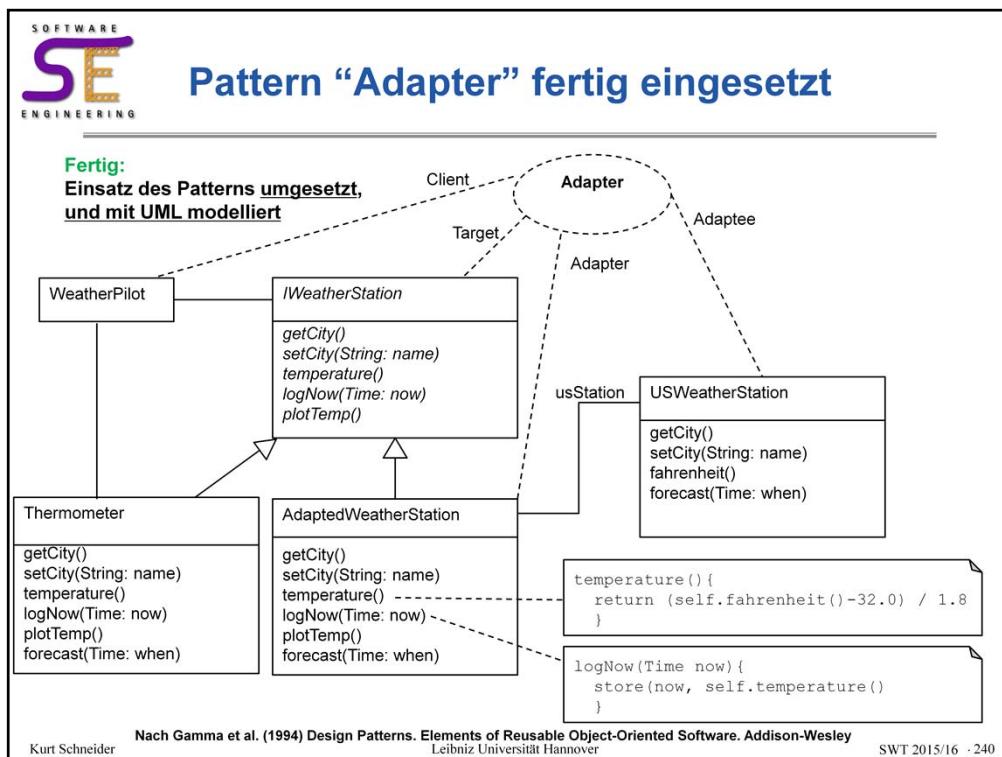
Der *Adaptee* (der sich anpassen lässt, bzw. um den eine Hülle, der *Wrapper* gezogen wird) kennt nämlich nicht den *Request()*, wohl aber den *SpecificRequest()*.

Indem nun Adapter die Schnittstelle zum Kunden (Client) von Target erbt – nur dafür ist Target da –, andererseits aber die Implementierung unter anderem Namen von Adaptee, vermittelt Adapter zwischen dem Client und einem versteckten Leistungserbringer (Adaptee).

Was bisher auf Klassenebene beschrieben wurde, kann auch ohne Vererbung von Adaptee auskommen. Es reicht, den *Request()* an eine Instanz von Adaptee als *SpecificRequest()* weiterzurichten. Diese Instanz *adaptee* muss sich der Adapter dann eben beschaffen.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Der Adapter ist oft eher als „Wrapper“ bekannt. Hier ist also der allgemeine Ablauf bzw. die Struktur, die hinter dem Drucker-Beispiel stand.

Die blaue Linie zeigt den logischen Ablauf:

Der Client schickt „Request()“ an ein Target. Target ist aber als definierte Schnittstelle eingeführt worden und damit eine *abstrakte Oberklasse*. Weil ihre Methoden nicht ausgeführt sind, müssen konkrete Unterklassen das übernehmen. Adapter ist eine solche konkrete Unterklasse. Im Endeffekt kennt der Client also nur Instanzen von Adapter.

Wenn diese den „Request()“ erhalten, bearbeiten sie ihn, indem sie ihn an eine *eigene* Methode mit anderem Namen (hier `SpecificRequest()`) weitergeben.

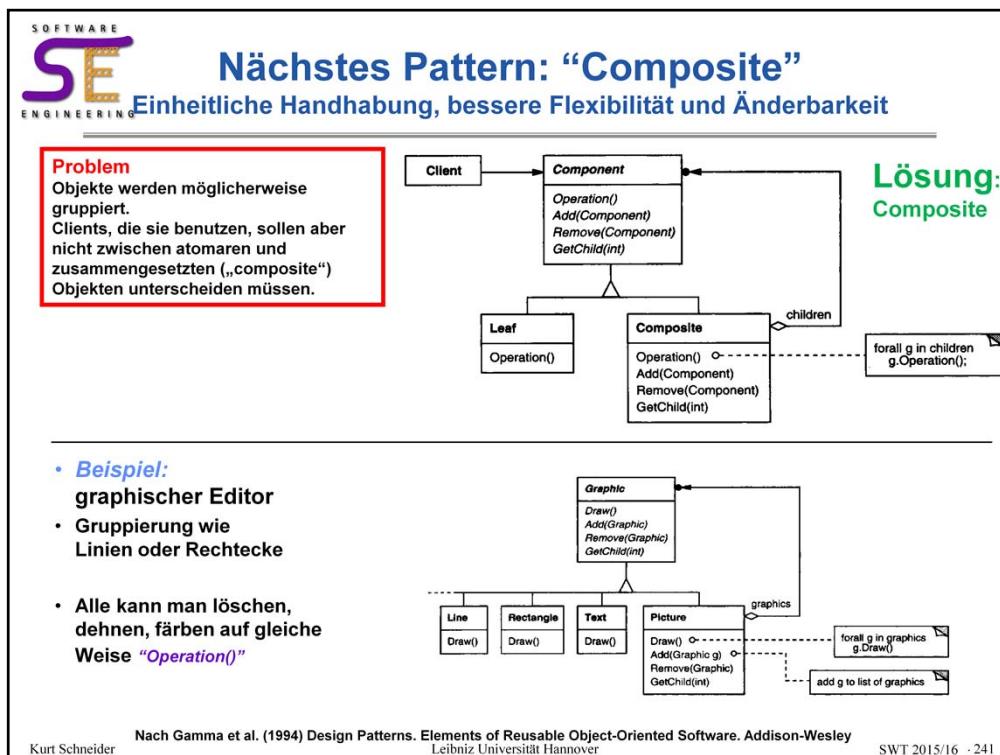
Der *Adaptee* (der sich anpassen lässt, bzw. um den eine Hülle, der *Wrapper* gezogen wird) kennt nämlich nicht den `Request()`, wohl aber den `SpecificRequest()`.

Indem nun Adapter die Schnittstelle zum Kunden (Client) von Target erbt – nur dafür ist Target da –, andererseits aber die Implementierung unter anderem Namen von Adaptee, vermittelt Adapter zwischen dem Client und einem versteckten Leistungserbringer (Adaptee).

Was bisher auf Klassenebene beschrieben wurde, kann auch ohne Vererbung von Adaptee auskommen. Es reicht, den `Request()` an eine Instanz von Adaptee als `SpecificRequest()` weiterzurichten. Diese Instanz *adaptee* muss sich der Adapter dann eben beschaffen.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



*Composite* schafft den Vorteil, zwischen atomaren und zusammengesetzten *Components* auf Client-Seite nicht unterscheiden zu müssen. Auch die anderen *Components* und die *Component-Oberklasse (abstrakt)* kriegt nicht mit, wenn eine neue *Component* – atomar oder zusammengesetzt als *Composite* – hinzukommt. Das ist gutes Information Hiding.

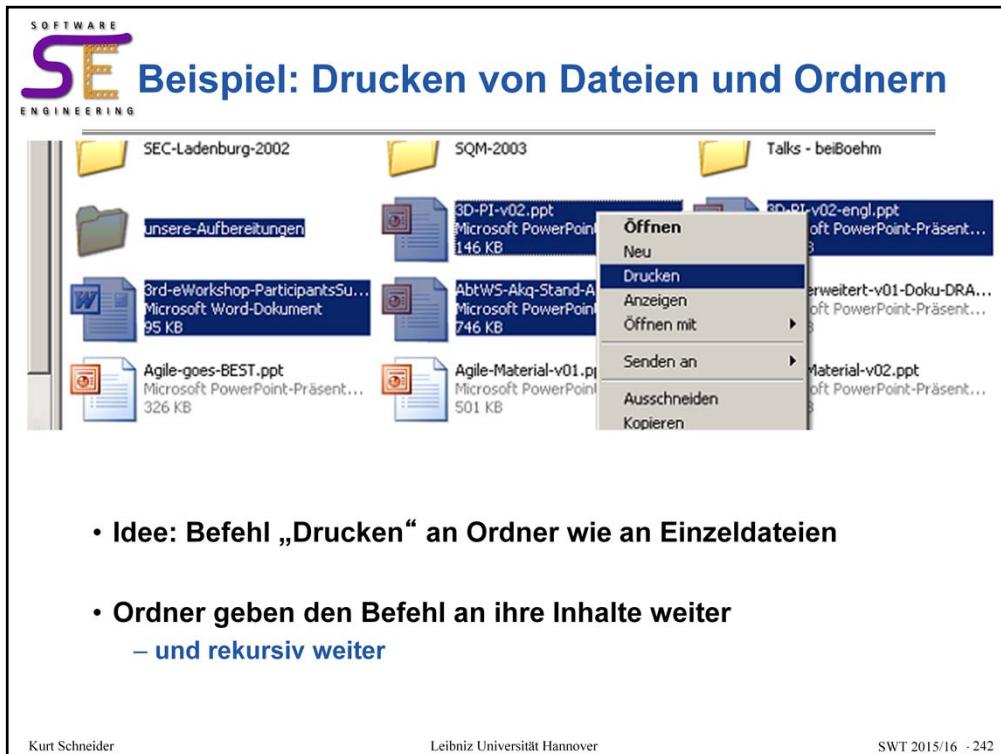
Der Trick liegt hauptsächlich in dieser Vererbungsstruktur und darin, die Iteration über die Elemente eines *Composites* innerhalb seiner Operationen vorzunehmen – und nicht etwa in *Component* als „Verteilerklasse“ mit Fallunterscheidung.

Jede *Component* muss für sich selbst wissen, wie sie mit den *Operation()*-Aufrufen umgeht. Und wenn dies die Iteration durch eine Liste erfordert, dann soll das kein anderer wissen müssen.

Die Client-Schnittstelle wird dadurch sehr einfach. Auch der Client braucht ja keine Fallunterscheidungen.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



- Idee: Befehl „Drucken“ an Ordner wie an Einzeldateien
- Ordner geben den Befehl an ihre Inhalte weiter
  - und rekursiv weiter

Dies ist ein Beispiel für Composite.

Auf einer normalen Windows-Oberfläche sind gleichzeitig mehrere Objekte angewählt worden (blau). Dadurch entsteht implizit ein Composite, denn jede Nachricht die dieser Sammlung geschickt wird, geht an alle Objekte weiter, zum Beispiel also Drucken.

Zu beachten ist noch der Ordner links oben, der ebenfalls mit ausgewählt wurde. Der Ordner spielt die Rolle eines Composites im Composite der blau-ausgewählten. Der „Drucken“-Befehl wird also an alle Elemente in diesem Ordner weitergegeben – und der Benutzer merkt keinen Unterschied, ob er sich an ein atomares Element oder an einen Ordner gewandt hat. Das ist der Effekt des Composites.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

The slide title is "Beispiel: Ordner drucken ohne Composite" and subtitle is "Naiver Ansatz". It features a logo for "SOFTWARE ENGINEERING" with a stylized "SE".

```
public class MausMenue {
    public static void main(String[] a) {
        Ordner o1=new Ordner();
        Ordner o2=new Ordner();

        o1.add(new WordDatei());
        o1.add(new WordDatei());
        o1.add(new PdfDatei());

        o2.add(new PdfDatei());
        o2.add(o1);
    }
}
```

A brace on the right side of the code groups the two `o1` and `o2` objects, labeled "Ordnerstruktur". Below the code is a tree diagram:

- `o2` (yellow box)
  - `o1` (blue box)
    - `WordDatei`
    - `WordDatei`
    - `PdfDatei`

Menü will `o2` auf „naive Weise“ drucken:

Alle Elemente von `o2` durchgehen  
WENN sie pdf oder Word sind: drucken  
SONST alle Elemente davon durchgehen  
WENN sie pdf oder Word sind: ...

Schon besser: Rekursiv

**SCHLECHT: bei JEDEM AUFRUF nötig!**  
**Aufwändig, fehleranfällig, änderungsfeindlich**

Kurt Schneider

Leibniz Universität Hannover

SWT 2015/16 · 243

Dies ist der „Kern“ einer Implementierung für das Ordner-Composite.

Es werden zwei Ordner aufgesetzt (der innere und der „äußere, blaue“). Dann werden Word- und pdf-Dateien dort hineingesteckt und der innere in den äußeren Ordner gebracht.

Mit `o2.drucken()` fordert das nutzende Programm die ganze Struktur (das verschachtelte Composite) auf, alle seine Inhalte zu drucken.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

Software  
SE  
ENGINEERING

## Beispiel: Ordner drucken

Das möchte man tun können

```
public class MausMenue {  
    public static void main(String[] a) {  
        Ordner o1=new Ordner();  
        Ordner o2=new Ordner();  
  
        o1.add(new WordDatei());  
        o1.add(new WordDatei());  
        o1.add(new PdfDatei());  
  
        o2.add(new PdfDatei());  
        o2.add(o1);  
    }  
}
```

Einfache  
Ordnerstruktur  
aufbauen

o2.drucken();

Gesamte Struktur drucken  
(wie eine Einzeldatei)

Kurt Schneider Leibniz Universität Hannover SWT 2015/16 · 244

Mit o2.drucken() fordert das nutzende Programm die ganze Struktur (das verschachtelte Composite) auf, alle seine Inhalte zu drucken.

Durch das Composite-Pattern weiß o2 selbst, dass es ein Composite (zusammengesetzt) ist und iteriert über alle seine Components. Eines davon ist o1, alle anderen sind Leafs, also Einzelkomponenten. Die Einzelkomponenten drucken sich einfach, sie können das. O1 kann es auch, aber nur, indem es wieder über seine Teil-Teilkomponenten iteriert. Vom aufrufenden aus (MausMenue bzw. o2) ist das jeweils aufgerufene Composite nicht anders als ein Leaf. Im Composite passiert aber etwas anderes: es iteriert selbstständig und gibt den Befehl an alle Teile weiter.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Beispiel: Ordner drucken

Die Bestandteile vorher *und nachher*

```
public class WordDatei implements Komponente {  
    public void drucken () {  
        System.out.println("WordDatei wird ausgedruckt.");  
    }  
}  
  
public class PdfDatei implements Komponente {  
    public void drucken () {  
        System.out.println("Pdf-Datei wird ausgedruckt.");  
    }  
}  
  
public interface Komponente {  
    public void drucken();  
}
```

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 245

Wichtig dafür ist, mit einem interface für Komponenten zu arbeiten. Dann sind sowohl die atomaren als auch die zusammengesetzten Teile Komponenten, gehorchen also dem gemeinsamen Interface.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Beispiel: Ordner drucken

Bindeglied: Composite/Ordner

```
import java.util.LinkedList;
import java.util.ListIterator;

public class Ordner implements Komponente {
    LinkedList inhalt = new LinkedList();

    public void add(Komponente k) {
        inhalt.add(k);
    }

    public void drucken() {
        Komponente komp;
        for (ListIterator it=inhalt.listIterator(); it.hasNext(); ) {
            komp = (Komponente) it.next();
            komp.drucken();
        }
    }
}
```

**Der Witz:**

Iteration im Composite  
(zusammenges. Objekt)  
statt im Aufrufer

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 246

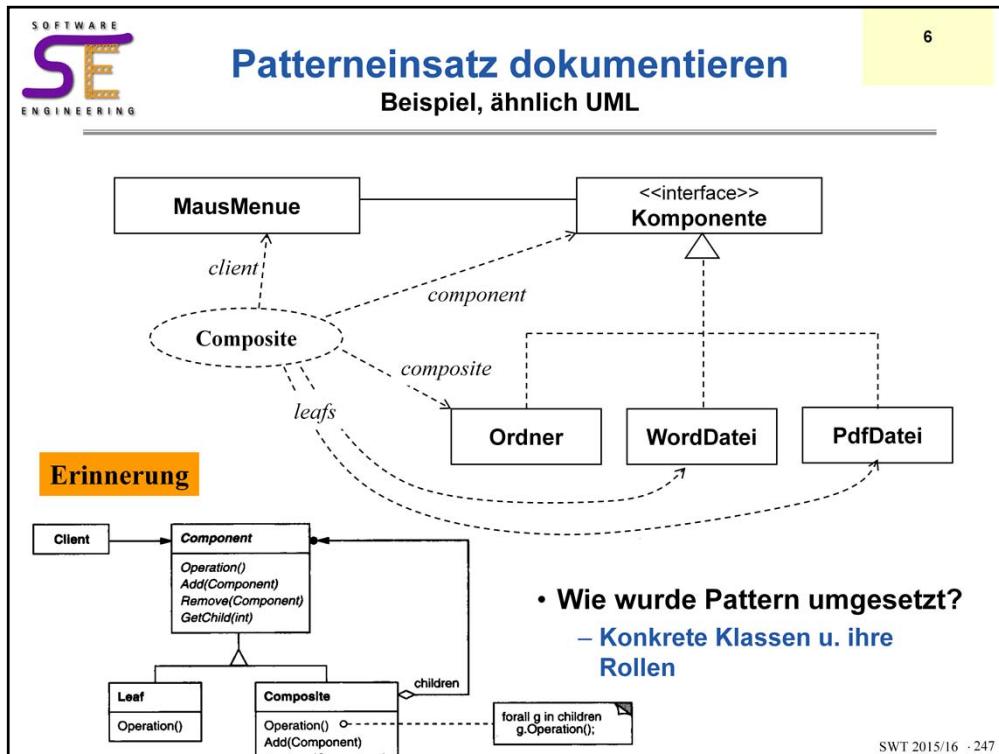
Und hier – im Rahmen – sieht man den zentralen Trick im Composite-Pattern: Nicht der Verwender muss abfragen, ob jemand zusammen gesetzt ist und dann Schleifen darüber laufen lassen. Sondern das zusammengesetzte Teil (hier: der Ordner) sorgt für die Iteration.

Nach außen bleibt das verborgen, damit kann ein Ordner genau wie ein atomares Element (eine Datei) gedruckt werden.

Wenn man nun verschiedene Operationen hätte, müsste für jede davon so eine Composite-Struktur aufgebaut werden, also eine innere Schleife programmiert.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

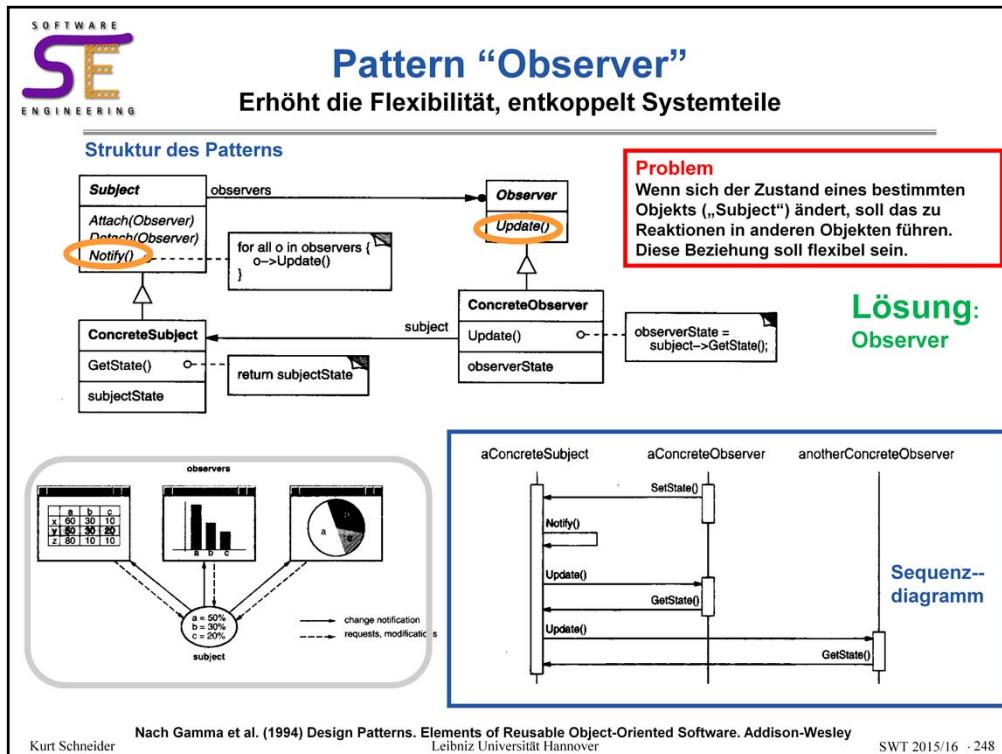
Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Vergleichen Sie das allgemeine Pattern unten mit dem konkreten Anwendungsfall (oben). Dazwischen ist die UML-Notation für Pattern-Anwendung. Mithilfe des Ovals und seiner gestrichelten Pfeile kann man erkennen, welches Pattern wie eingesetzt wurde. Das ist wichtig, damit Ihre Kollegen im Team später sofort erkennen: hier wurde ein Composite-Pattern eingesetzt und darf nicht „kaputtgemacht“ werden (z.B.. Um etwas zu Vereinfachen). Außerdem spart man sich auf diese Weise viel Dokumentation, denn alles, was man allgemein zu Composite-Patterns weiß, muss ja nicht mehr wiederholt werden. Das kann man im Gamma-Buch nachlesen.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



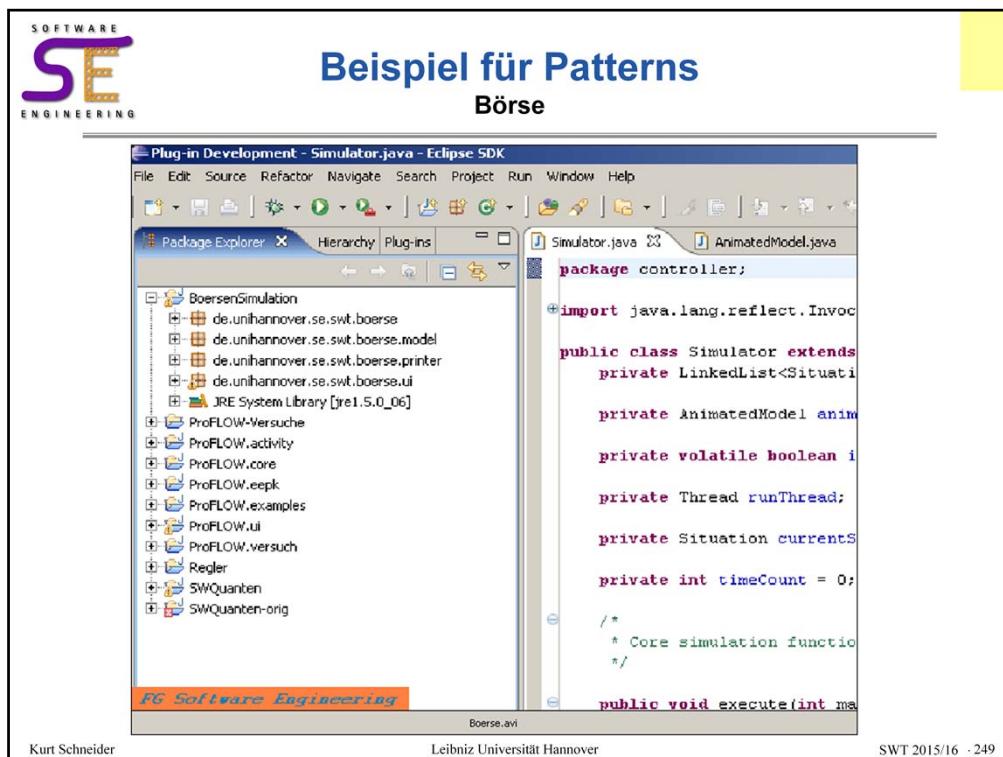
Wenn der Zustand eines Subjekts sich ändern kann und dies dann zu Reaktionen mehrerer anderer führen soll, so kann man dafür das Observer-Pattern einsetzen.

Jedes konkrete `concreteSubject` ist Instanz einer Unterklasse von `Subject`. `Subject` selbst kennt seine `Observer` und unterrichtet sie, wenn sich sein Zustand geändert hat. Diese Fähigkeit erben die konkreten Unterklassen.

Wichtig: Sowohl bei `Composite` als auch bei `Observer` ist es einfach, sich grafische Anwendungen zu überlegen. Denken Sie aber auch einmal über andere, nicht-grafische Anwendungen dieser Patterns nach! Denn die von ihnen adressierten Probleme haben zunächst einmal nichts mit Grafik oder Oberflächen zu tun.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

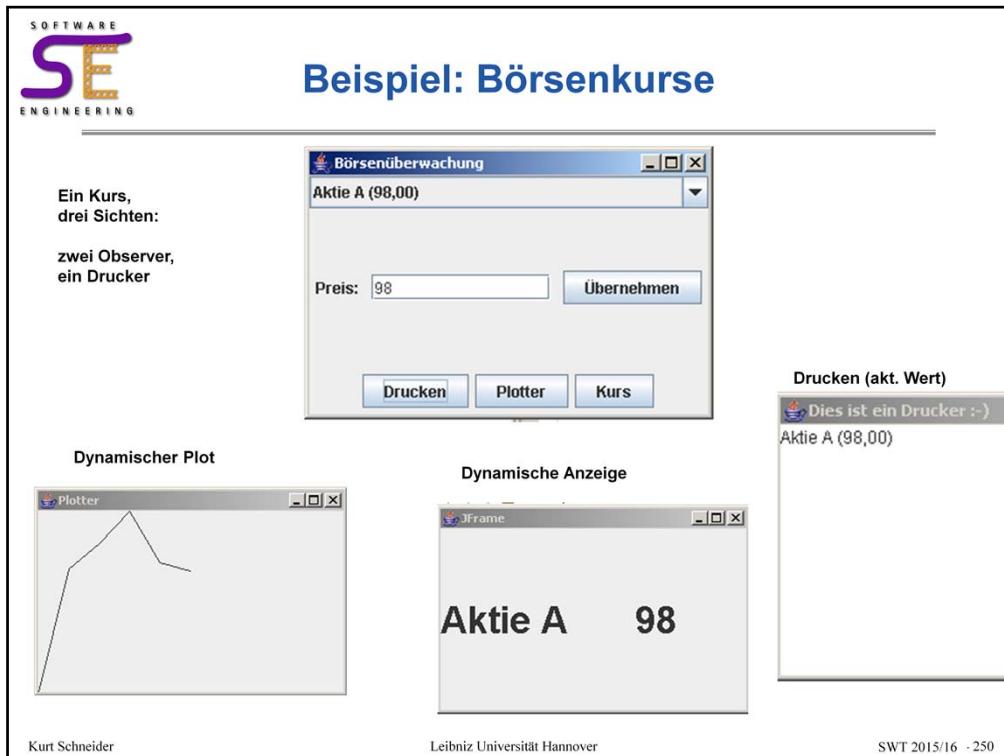
Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Die Paketstruktur für das Börsenbeispiel.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Wir erzeugen ein Observable, nämlich eine Aktie mit Kurs (oben).

Dazu schreiben wir zwei einfache Observer (links und mitte unten) und eine Ausgabe, die kein Observer ist, damit man den Unterschied sieht.

Wenn man den Preis der Aktie oben ändert, verändern sich die beiden Observer mit.

Der „Ausdruck“ rechts dagegen nicht. Der wurde einmal erstellt und beobachtet nicht, was danach mit der Aktie passierte.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Beispiel: Börsenkurse

### Ausgangssituation

```
public class Stock {  
    private String name;  
    private int price; // in cents  
  
    public Stock(String name) {  
        this.name = name;  
    }  
  
    public String toString() {  
        return this.getName() + " ("  
            + String.format("%1.2f", this.getPrice() / 100.0) + ")";  
    }  
  
    public String getName() { return name; }  
    public int getPrice() { return price; }  
  
    public void setPrice(int price) {  
        if (price >= 0) {  
            this.price = price;  
        }  
    }  
}
```

**Aufruf**

```
print1.print(MainFrame.this.getSelectedStock().toString());
```

```
MainFrame.this.getSelectedStock().setPrice(4711);
```

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 251

Hier ein ganz ungewöhnliches, von der Bildschirmdarstellung unabhängiges Beispiel:

Aktienkurse können beobachtet werden, die an der Börse notiert sind.

Dazu gibt es zwei Observer, die auf Änderungen reagieren. Hier zunächst einmal das inhaltliche Modell, noch ganz ohne Observer.

So könnte man das implementieren. Die fetten Teile sind für uns besonders interessant. Das Beispiel ist aber so einfach gewählt, dass möglichst wenig Drumherum dem Blick auf das Prinzip verstellt. Die gelben Rechtecke zeigen, wie man die Funktionen aufruft. Diese Aufrufe sind nicht besonders schön, aber hier nicht unser Thema. Im wesentlichen landet man dann eben bei den zwei Operationen `toString()` und `setPrice()`.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Börse, nun mit Observer-Pattern

```
Import java.util.Observable;
public class Stock extends Observable {  
  
    private String name;  
    private int price; // in cents  
  
    public Stock(String name) {  
        this.name = name;  
    }  
  
    public String toString() {  
        return this.getName() + " ("  
            + String.format("%1.2f", this.getPrice() / 100.0) + ")";  
    }  
  
    public String getName() { return name; }  
    public int getPrice() { return price; }  
  
    public void setPrice(int price) {  
        if (this.price >= 0) {  
            this.price = price;  
            this.setChanged();  
            this.notifyObservers();  
        }  
    }  
}
```

Erbt von Java-Klasse Observable  
Kann damit alles, was Observable kann

Insbesondere:

1. sich bei Änderungen selbst benachrichtigen
2. `notifyObservers()` meldet an alle Observer weiter
3. Dann müssen sich Observer aktualisieren (`update()`)

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 252

Jetzt wurde statt der ganz normalen Implementierung ein Observer eingesetzt.  
Viel ändert sich für den beobachteten Aktienwert nicht. Nur muss er keine Sichten mehr versorgen, er meldet nur an die Observer, dass er sich verändert hat.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

**SOFTWARE  
SE  
ENGINEERING**

## Observer an- und abmelden beim Observable; der verwaltet sie (geerbt)

Jeder Observer trägt sich beim Observable ein

```
public GraphComponent(Stock stock) {  
    super();  
    this.stock = stock;  
    stock.addObserver(this);  
}  
  
public StockQuoteFrame(Stock stock) {  
    super();  
    initialize();  
    this.stockNameLabel.setText(stock.getName());  
    this.stockQuoteLabel.setText(""+stock.getPrice()/100);  
    this.stock = stock;  
    stock.addObserver(this);  
}
```

Nicht vergessen: nicht mehr nötige abhängen!

```
public void dispose() {  
    super.dispose();  
    stock.deleteObserver(this);  
}
```



Kurt Schneider Leibniz Universität Hannover SWT 2015/16 · 253

Damit das funktioniert, müssen sich die Observer beim Observable registrieren, dann kann er sie später benachrichtigen.

Die Aktie (stock) erfährt dann, dass sie einen weiteren Observer hat. Natürlich kann man Observer mit delete auch wieder entfernen.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Was tut „notifyObservers“?

Ganz einfach: schickt an alle eingetragenen Observer eine „update()“-Aufforderung

```
public void notifyObservers() {  
    Observer[] allObservers;  
  
    ...  
    for (ListIterator it=allObservers.listIterator();  
         it.hasNext(); ) {  
        obs=it.next();  
        obs.update();  
    }  
}
```

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 254

Im wesentlichen muss dazu die Oberklasse der Beobachtbaren (Aktienkurse) dafür sorgen, dass bei jeder notify/benachrichtigen-Nachricht alle Observer informiert werden. Jeder kriegt den Aufruf „aktualisieren“ (allgemein: update).

Hinweis: in manchen Umfeldern hat update mehrere Parameter (z.B. in Java). Die würden dann hier auch stehen.

Für das Prinzip sind sie aber unerheblich, daher fehlen sie hier.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

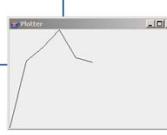
Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

**Beispiel: Börsenkurse**  
Die Beobachter

```
public void update(Observable o, Object arg) {  
    if (this.stock == o) {  
        this.stockQuoteLabel.setText(""+this.stock.getPrice()/100);  
    }  
}
```



```
public void update(Observable observable, Object param) {  
    if(observable == this.stock) {  
        this.addValue(this.stock.getPrice());  
        this.repaint();  
    }  
}
```



Kurt Schneider

Leibniz Universität Hannover

SWT 2015/16 · 255

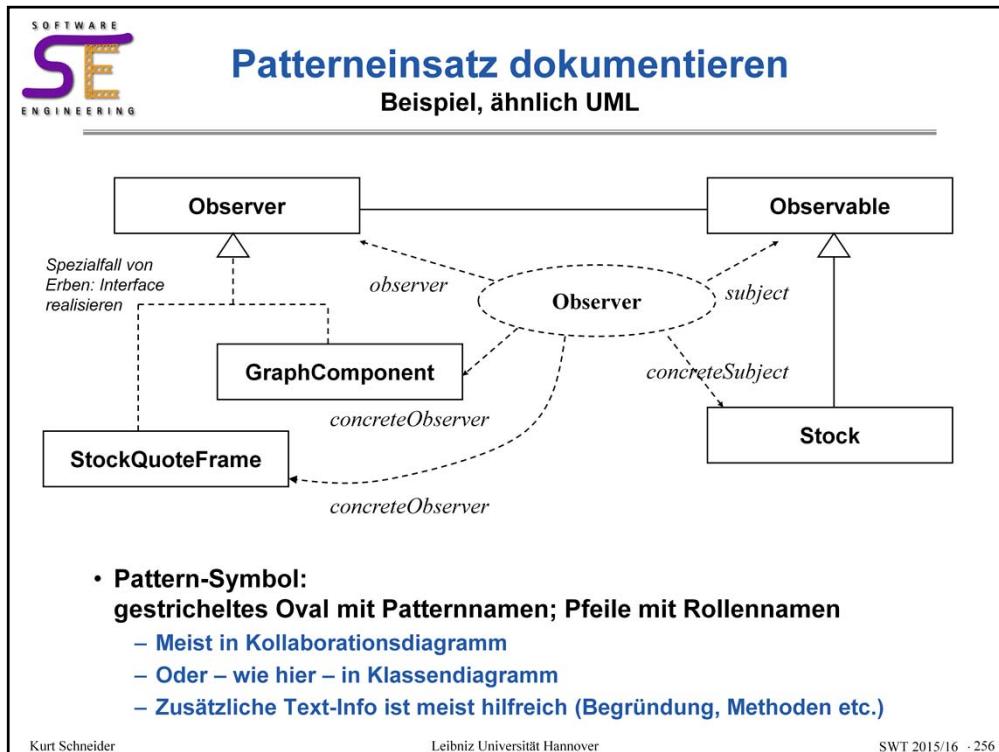
Die Observer sind ganz einfach, nur der Kern ist ja wichtig.

Sie prüfen beim Aktualisierungsauftrag, ob sie aktiv werden müssen und tun dann, was ihnen angemessen erscheint.

Im wesentlichen geben sie den Wert oder einen Kurventeil aus.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Wieder sieht man hier, wie die Anwendung des Patterns in ein konkretes Klassendiagramm des Beispiels eingebettet ist.

Wieder zeichnet man das Pattern als gestricheltes Oval in UML, mit gestrichelten Pfeilen zu den verschiedenen Teilen („Rollen im Pattern“).

So weiß man, welche Klasse der Börse welche Rolle im Pattern spielt. Die ganze Hintergrundinformation über das Observer-Pattern kann man aus dem Gamma-Buch entnehmen und muss es nicht in die Dokumentation schreiben. Allenfalls noch, warum man sich hier für dieses Pattern entschieden hat.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

## Methode im Zusammenhang

### Patterns verwenden

**WENN eines der Patterns mit seinem Problem-Teil zu Ihrem Problem passt**  
**DANN überlegen Sie, ob Sie es einbauen wollen**  
**WENN es sich zu lohnen scheint, DANN bauen Sie es ein**  
**SONST nicht!**

- 1. Geeignete Patterns finden: Im Entwurf**
  - 1. Sie sollten einige Patterns auswendig kennen: im Geiste durchgehen
  - 2. Haben Sie ein wichtiges Problem, lesen Sie Problemteile weiterer Patterns durch
- 2. Anwendbarkeit prüfen**
  - 1. Passt der „Problem“ –Teil?
  - 2. Wie wichtig ist Ihnen die angestrebte Eigenschaft (oft Flexibilität)?
  - 3. Abwägen und dokumentieren von Nutzen und Aufwand
- 3. Einsatzort festlegen**
  - Genau festlegen, welche konkrete Klasse o.ä. zu welcher Rolle im Pattern passt
- 4. Pattern-Lösungsteil einfügen**
  - In UML-Entwurf, evtl. auch in schon existierenden Code
- 5. In der Regel erst Pattern aussuchen, dann programmieren**

Kurt Schneider                                   Leibniz Universität Hannover                                   SWT 2015/16 · 257

Wichtige Aussage: Patterns sind eine gute Idee, aber kein Selbstzweck.

Ein paar müssen Sie auswendig kennen, und Sie müssen wissen, wo Sie die anderen finden.

Wenn einmal ein Pattern nicht passt, ist das auch in Ordnung: sie sind ja kein Selbstzweck.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

Das Diagramm zeigt die 3-Schichten-Architektur (3-Tier) und das MVC-Pattern. Auf der linken Seite ist eine 3-Schichten-Architektur dargestellt, bestehend aus 'Anwendung (mit GUI)', 'Geschäftslogik' und 'DB Infrastruktur'. Zwischen den Schichten befinden sich vertikale Pfeile, die die Datenflüsse anzeigen. Rechts davon befindet sich ein Abschnitt mit dem Titel 'Sehr nützlich: MVC Model View Controller'. Dieser Abschnitt enthält eine Liste von Punkten, die das MVC-Pattern erläutern:

- WENN Sie ein Programm entwickeln
  - Mit intensiver Logik/Datenhaltung
  - UND nicht-trivialer Oberfläche
- DANN trennen Sie Oberfläche vom Rest UND verwenden Sie dabei das MVC-Pattern
- Ziel: Entkoppelung zwischen Inhalt (Model) und Oberfläche (View-Controller)
  - Gibt genaue Interaktion vor
  - Damit kann man fast jede Komponente separat ändern (information hiding, divide et impera)

Vgl.: Zuse et al. (2004): Software Engineering mit UML und dem Unified Process. Pearson Studium

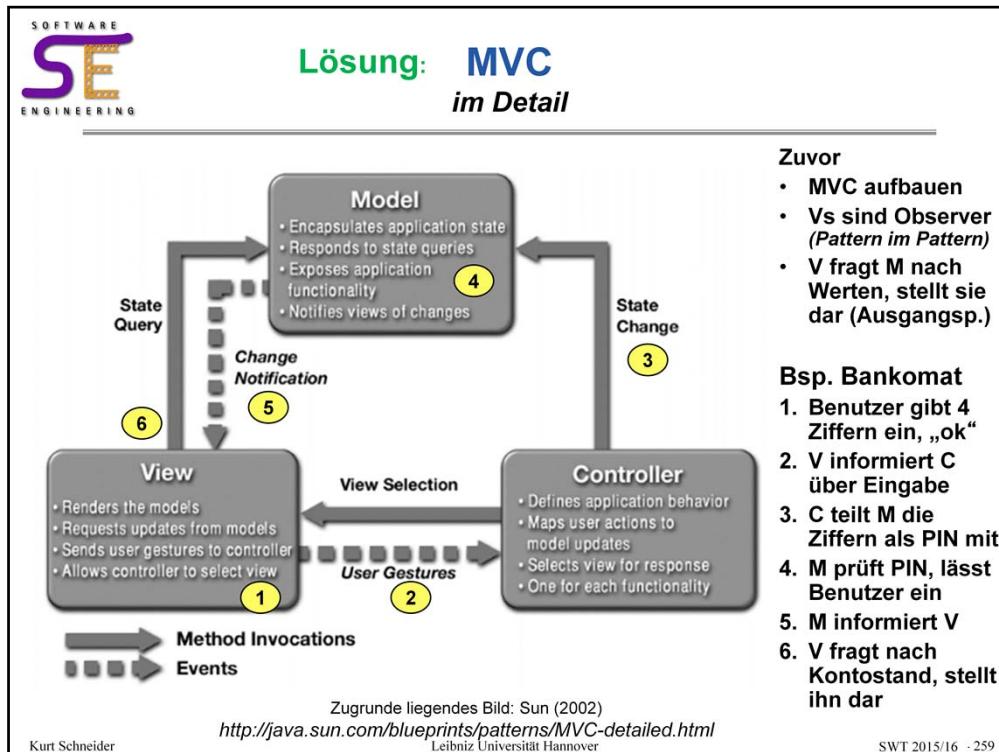
Kurt Schneider Leibniz Universität Hannover SWT 2015/16 · 258

Rechts sieht man sogar eine doppelte 3-Schichten-Architektur. Oben (in grau, mit Kommentaren dran) und in blau noch einmal auf die Verteilung bezogen.

Jetzt kommen wir zu dem ziemlich komplizierten MVC-Pattern. Es steht nicht im Gamma-Buch ist aber auch sehr weit verbreitet und sehr nützlich.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Diese Darstellung von Sun zeigt die Aufgaben und den Kontrollfluss zwischen den drei Teilen von MVC.

Die obige Struktur muss natürlich zuerst einmal aufgebaut werden. In der Anwendung werden dann die nummerierten Schritte ausgeführt.

MVC trägt dazu bei, dass jeder der drei Teile sich auf seine eigenen Aufgaben konzentriert und im Prinzip ohne allzu viele Änderungen an den anderen beiden Teilen ausgetauscht werden kann.

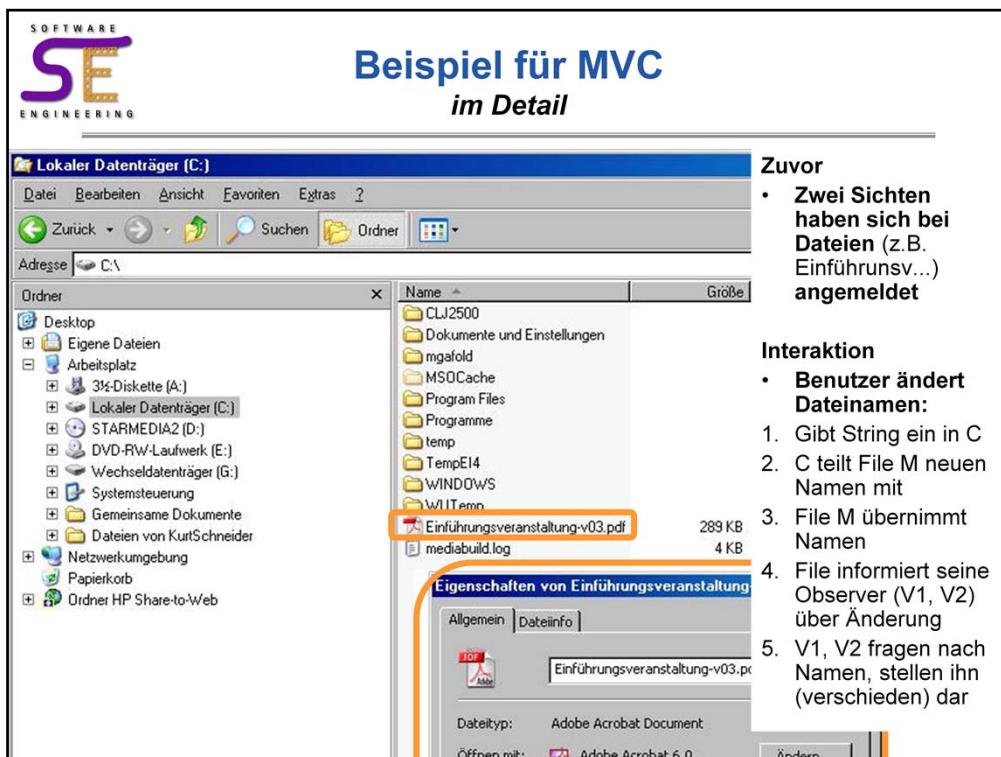
Das Model ist relativ passiv und meldet nur, wenn es sich verändert hat.

Darauf reagieren die Views, die als Observer beim Model eingetragen sind.

Eine Aktion startet aber damit, dass ein Controller eine Änderung wahrnimmt, sie in den Begriffen des Models interpretiert und das Model zu einer Änderung auffordert.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Hier sieht man zwei Views von einem Model.

Eine Datei wird dargestellt durch ein Symbol und ihren Namen in der Liste.

Andererseits aber kann man ein ganzes Eigenschaften-Fenster über die Datei ansehen, mit Größe, Speicherort und Namen.

Wenn man nun (1) den Namen in der Liste ändert, dann wird diese Änderung bemerkt, an das Model (die Datei) weitergegeben, die ihrerseits alle Views benachrichtigt, auch das Eigenschaften-Fenster.

Oft ist MVC hierarchisch/rekursiv aufgebaut: In der View eines Ordners (die Liste) steht die View einer Datei (Symbol und Name). Dem entspricht ein großes Model (Datei), das ein kleineres umfasst (Dateiname).

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

**Java-Beispiel**

Applet-Ansicht: de.unihannover.fgse.emailchecker.mod... Applet

Email Evaluator kurt.schneider@inf.uni-hannover.de Länge: 34

check

LengthView

BarView

Applet gestartet

- Emailchecker evaluiert Email-Adressen
  - Wie lang sind sie? → LengthView
  - Sind sie zu lang? → Bar View
- Ziel dieses Beispiels: „reines MVC“
  - Echtes Java-Beispiel
  - Nutzt MVC in Java
  - Sonst nichts, so klein und simpel wie möglich

Kurt Schneider Leibniz Universität Hannover SWT 2015/16 · 261

Wieder ein möglichst einfaches Beispiel, an dem man aber das Pattern gut sehen kann (hier MVC).

Oben gibt man eine Email-Adresse ein, drückt auf Check und erhält ihre Länge angezeigt. Darunter erscheint ein wertender Balken: ist die Adresse in Ordnung (grün), etwas zu lang (gelb) oder viel zu lang (rot).

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Relevanter Code

### Die Anwendung als Applet, MVC-Teile fett

```
public class EmailCheckerApplet extends JApplet {  
    EmailEvaluator model = new EmailEvaluator();  
    LengthView numbView = new LengthView(model); // Zahl  
    BarView barView = new BarView(model); // Balken  
  
    public void init(){  
        this.setSize(400,140);  
        GridLayout gridBag = new GridLayout(2,1);  
        getContentPane().setLayout(gridBag);  
  
        // Views in die GridBag einfügen  
        getContentPane().add(numbView);  
        getContentPane().add(barView);  
    }  
}
```

Kurt Schneider    Leibniz Universität Hannover                                    SWT 2015/16 · 262

Wie im Börsenbeispiel mit den Observern wird hier für MVC jeweils der interessante Teil fett hervorgehoben.

Hier werden die beiden Views dem Modell zugeordnet, vorher wird das Modell angelegt.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Relevanter Code

### Model

```
public class EmailEvaluator extends Observable {  
    private String email = "";  
  
    public void setEmail(String emailAddress){  
        email = emailAddress; // Model ändert sich  
  
        setChanged(); // Auf Änderung hinweisen  
        notifyObservers();  
    }  
}
```

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 263

In Java gibt es das Observer-Observable-Pattern schon, man kann sich einfach unter die Observable-Klasse hängen und erbt dann die Fähigkeit, Observer zu verwalten und sie zu benachrichtigen. Das ruft man (bei sich selbst) immer dann auf, wenn man sich als Observable geändert hat – hier die eingegebene Email-Adresse.

setChanged() ist eine Java-Spezialität ohne konzeptionelle Bedeutung. Sie muss dastehen, man braucht sich aber keine großen Gedanken zu machen. Dient der effizienten Umsetzung des Patterns.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Relevanter Code

### LengthView

```
public LengthView(EmailEvaluator anEvaluator) {
    this(); // baut GUI-Elemente zusammen

    model = anEvaluator;
    model.addObserver(this);
    evalController = new EmailEvalController(model);
    setPreferredSize(new Dimension(350, 100));
}

public void update(Observable obs, Object obj) {
    length.setText(String.valueOf(model.getLength()));
    repaint();
}
```

**LengthView weiß selbst**

- interessanten Aspekt in Model
- wie holen
- wie darstellen

Kurt Schneider Leibniz Universität Hannover SWT 2015/16 · 264

Die Längenanzeige tut nicht viel: Sie zieht aus dem beobachteten Model den Aspekt heraus, den es braucht – und zeigt die Länge einfach als Zahl an.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Relevanter Code

### BarView

```
public class BarView extends JPanel implements Observer{  
    private EmailEvaluator model;  
  
    public BarView(EmailEvaluator aModel) {  
        super();  
        model = aModel; // in MVC einbauen  
        model.addObserver(this);  
    }  
    public void update(Observable ignored, Object alsoIgnored) {  
        repaint(); // ruft paint() auf, aktualisiert Balken  
    }  
  
    public void paint(Graphics g){  
        super.paint(g);  
        ...  
        // Farben markieren, ob die Länge akzeptabel ist  
        if (numOfChars > 23){  
            gc.setColor(Color.RED);  
        }  
        ...  
        gc.fillRect(5, 5, barLength , 30); // Email-Balken  
    }  
}
```

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 265

Die Grenzen (23) gehören eigentlich zur „Logik“ und damit zum Model.

Für den Balken sieht es etwas komplizierter aus, aber der blaue Teil ist der Kern: hier wird der Balken gezeichnet.

Was hier die Farbe festlegt, nämlich die Grenzen für etwas und viel zu lange Email-Adressen, gehört nicht unbedingt zur View.

Man kann auch sagen, dass es eher eine Frage der „Logik“ ist, und die würde zum Model gehören.

Andersherum kann man argumentieren: Es hängt von dieser Sicht ab, wann eine Email für zu lang gehalten wird.

Hier müsste der Entwerfer ein wenig erklären, wieso er sich für welche Lösung entschieden hat.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## MVC besser umgesetzt Model entscheidet, BarView stellt dar

Das Model: Semantik/Logik	Die View: Darstellung
<pre>public class EmailEvaluator extends Observable {  ...  public boolean isTooLong (int chars){     return (chars &gt; 23; }  public boolean isPerfect (int chars){     return (chars &lt; 15); }</pre>	<pre>public class BarView extends JPanel implements Observer{  ...  if (model.isTooLong(numOfChars)){     gc.setColor(Color.RED); } else     if (model.isPerfect(numOfChars))     {         gc.setColor(Color.GREEN);     } else {         gc.setColor(Color.YELLOW);     }</pre>

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 266

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Controller und Listener in Java

```
sendComButton.addActionListener (
    new ActionListener(){
        public void actionPerformed(
            ActionEvent buttonPressedEvent){
            try {
                // Wenn Button gedrückt: evalController soll
                // Emailadresse holen, Daten damit aktualisieren
                evalController.updateRawData(emailAdr.getText());
            }
            catch(RuntimeException e) {
                System.out.println("Fehler mit dem Button");
            }
        }
    );
}
```

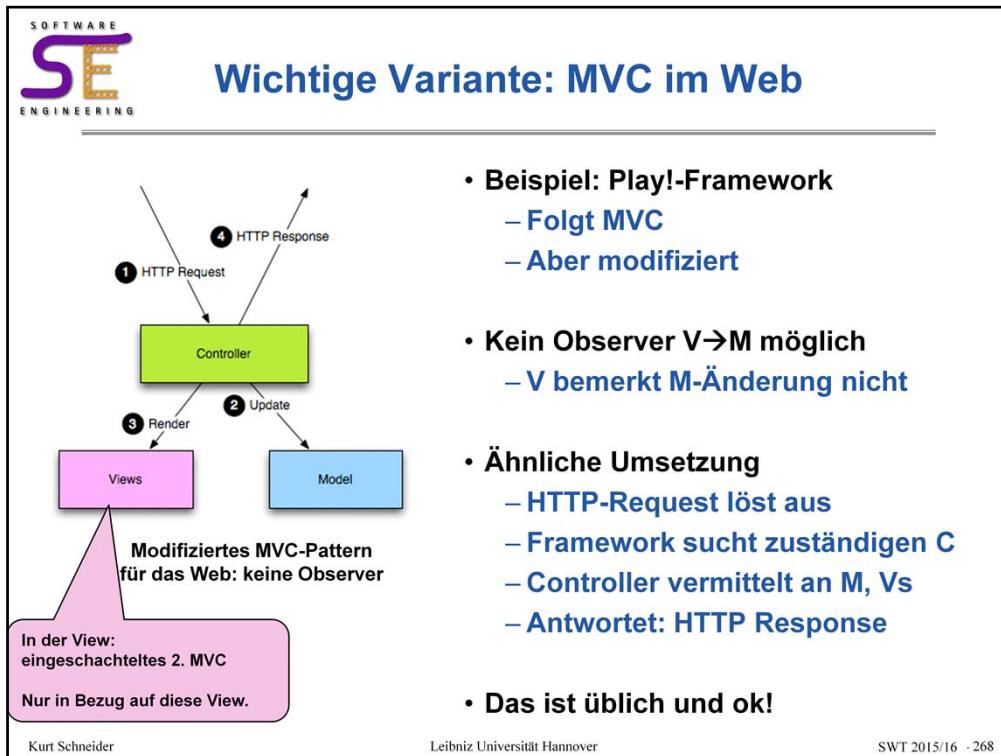
Hier wird ein Listener erzeugt, der den sendComButton belauscht.

Listener sind wie Observer auf GUI-Elementen in der View:  
Sie „lauschen“, was dort geschieht, darauf reagiert der Controller

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 267

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover

**Methode im Zusammenhang**  
MVC in die Architektur einbetten

WENN Sie in eine nicht-triviale GUI und mindestens einen der folgenden Aspekte haben:

- Nicht-triviale Logik
- Datenverwaltung/Verteilung

DANN verwenden Sie MVC zur Entkoppelung

– Legen Sie dabei View und Controller in eine Schicht (die oberste)

- Zunächst: jede Schicht ein Java-Package
- Darunter: Schicht bei Bedarf noch in feinere Pakete zerlegen

– View und Controller möglichst in unabhängige Klassen u. Pakete

- Aber gerade VC lassen sich nicht immer wirklich trennen
  - In Java arbeiten Sie mit Listenern für Controller; das geht
  - In C# oder Visual Basic ist die Trennung schwerer möglich
  - Andere Sprachen sind bei VC oft auch „eigenwillig“

– Legen Sie das Model in eine andere Schicht

- Model muss von VC besonders klar getrennt sein, in allen Sprachen
- Drei-Schichten-Architektur ist ein Spezialfall

• Vorteile

- Oberflächen ändern sich oft; nicht mit Model vermischt, Änderungen unabhängig
- Selbst ohne klaren Controller sind VC von M getrennt; geht in vielen Sprachen
- Ideal ist natürlich eine weitere Trennung von V und C

Kurt Schneider      Leibniz Universität Hannover      SWT 2015/16 · 269

Hier überlappen sich die 3-Schichten-Architektur und das MVC-Pattern.

MVC hat nichts mit der Datenschicht zu tun, aber V+C stecken in der Darstellungsschicht.

Das Model entspricht genau der Geschäftslogik.

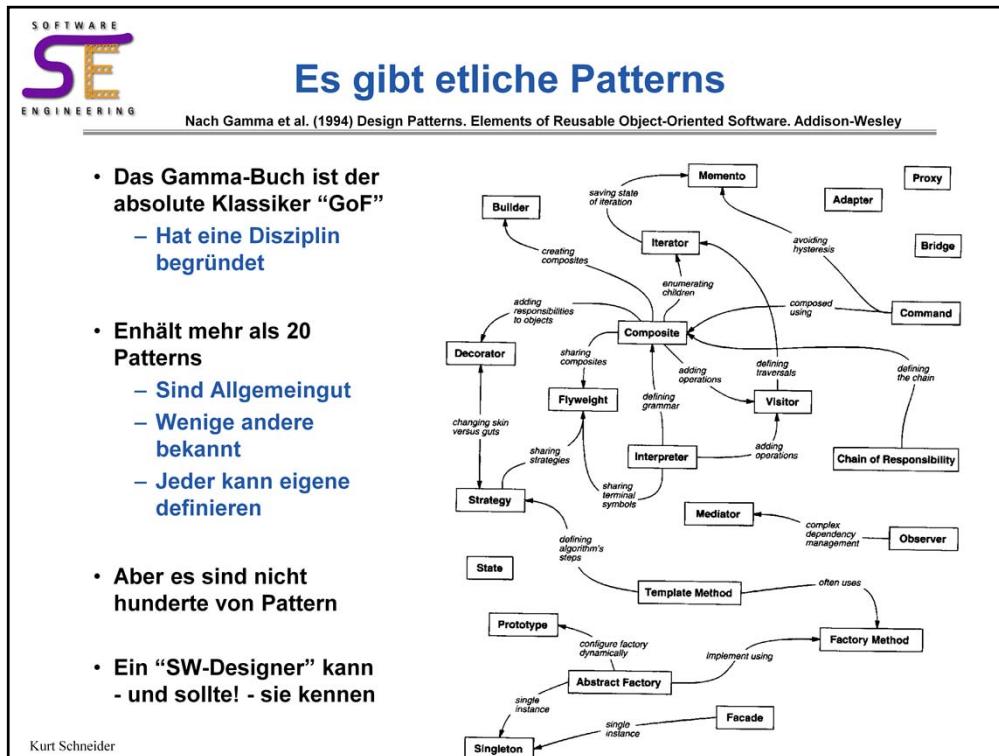
Diese Zusammenhänge drückt die Daumenregel aus.

Die Trennung von View und Controller ist nicht so streng und nicht so wichtig wie die Absonderung des Models.

Es hängt auch sehr von der Sprache und der Umgebung ab, wie V+C aussehen.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



Inzwischen gibt es etliche Bücher über Design Patterns.

Das hier genannte war das erste und sticht daher hervor.

Natürlich können Sie alle Patterns auswendig lernen; pragmatisch sinnvoller ist aber, sich erst einmal wenige zu merken, bei einer praktischen Aufgabe aber immer kurz zu prüfen, ob sich nicht auch mit Vorteil ein Pattern einsetzen lässt. So verfestigen Sie bekannte und erschließen sich unbekannte Patterns der Reihe nach.

# Vorlesung „Grundlagen der Softwaretechnik (SWT)“ im WS 2015/2016

Prof. Dr. Kurt Schneider, FG Software Engineering, Leibniz Universität Hannover



## Bewertung von Design Patterns

7

- Größere (und abstraktere) Entwurfsbestandteile als Klassen und Objekte; programmiersprachen-unabhängig
- Patterns sind konstruktive Qualitätssicherung
  - Qualität betroffen (oft: mehr Flexibilität)
  - Rationale/Begründung gehört zum Pattern
  - Erklärungen werden kompakter mit Patterns
  - Effiziente Kommunikation unter Experten möglich
  - Davon profitieren Lesbarkeit, Flexibilität, Wartbarkeit
  - Ein Pattern-Katalog ist wie ein Musterbuch für Architekten
- Sind in der OO-Zunft bekannt und werden vorausgesetzt
  - Fast jeder Entwickler hat davon gehört
  - Profis können sie selbst einsetzen

Kurt Schneider Leibniz Universität Hannover SWT 2015/16 · 271

Ganz wichtig ist die Verminderung der kognitiven Last:

- Sie brauchen nur ein Stichwort zu sagen oder zu schreiben und schon wissen alle anderen viele Details über ihren Entwurf.
- Sie brauchen sich auch nicht mehr alle Details zu merken und behalten bei der Entwurfstätigkeit den Kopf frei für andere Einzelheiten.