Software-Projekt

Prof. Dr. Rainer Koschke

Fachbereich Mathematik und Informatik
Arbeitsgruppe Softwaretechnik
Universität Bremen

Wintersemester 2008/09

Überblick I

1 Architekturstile und Entwurfsmuster

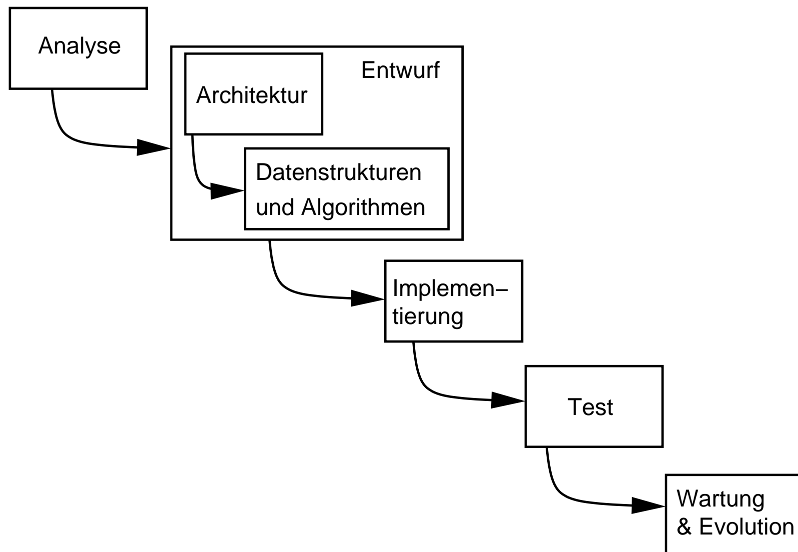
Architekturstile und Entwurfsmuster

1. Architekturstile und Entwurfsmuster
   - Was ist ein Entwurfsmuster?
   - Bestandteile eines Entwurfsmusters
   - Kategorien von Entwurfsmustern
   - Entwurfsmuster Factory Method
   - Entwurfsmuster Observer
   - Architekturstile
   - Architekturstil Schichtung
   - Architekturstil Model-View-Controller
   - Wiederholungsfragen

Lernziele

- Verstehen, was Entwurfsmuster und Architekturstile sind
- Qualitäten und Einsetzbarkeit der Entwurfsmuster/Architekturstile kennen

Kontext



Entwurfsmuster

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 (Architekt und Mathematiker),
"A pattern language", 1977.

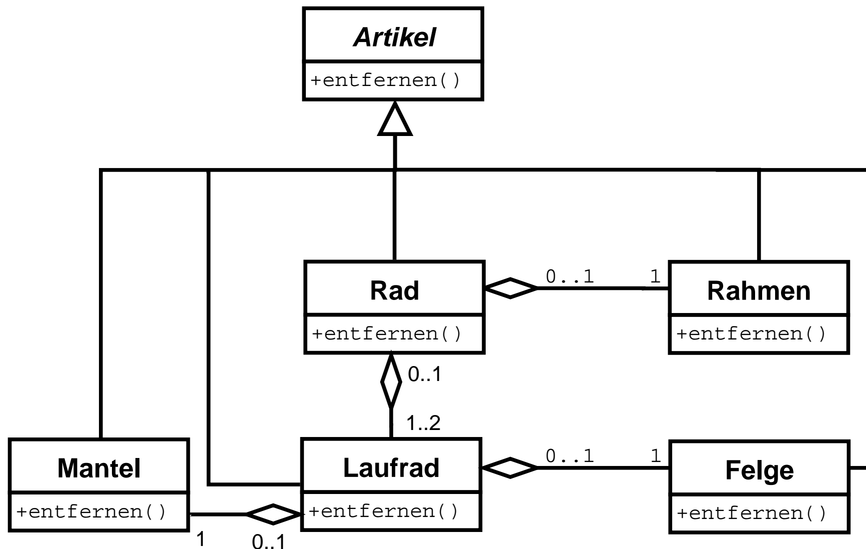
Definition

Entwurfsmuster: „Musterlösung“ für ein wiederkehrendes Entwurfsproblem.

Bestandteile eines Entwurfsmusters

- **Name** (kurz und beschreibend)
- **Problem**: Was das Entwurfsmuster löst
- **Lösung**: Wie es das Problem löst
- **Konsequenzen**: Folgen und Kompromisse des Musters

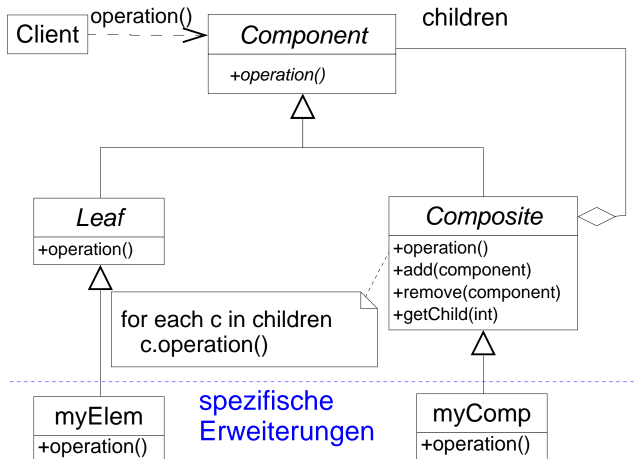
Beispielentwurfsproblem



Beschreibung von Mustern (Gamma u. a. 2003) I

- **Name**: *Composite*
- **Zweck**: Teil-von-Hierarchie mit einheitlicher Schnittstelle beschreiben (überall wo ein Ganzes benutzt werden kann, kann auch ein Teil benutzt werden und umgekehrt)
- **Motivation**: . . . Einführung anhand eines konkreten Beispiels. . .
- **Anwendbarkeit**:
  - wenn Teil-von-Beziehung beschrieben werden soll
  - uniforme Schnittstelle für alle Elemente der Hierarchie

- **Struktur**:



Beschreibung von Mustern (Gamma u. a. 2003) I

Teilnehmer:

- *Client*:
  - manipuliert Objekte der Komponenten nur durch die Schnittstelle von *Composite*
- *Component*:
  - deklariert einheitliche Schnittstelle
  - (optional) implementiert Standardverhalten

```
public abstract class Component {
    public abstract void operation ();
}
```

Beschreibung von Mustern (Gamma u. a. 2003) II

- *Leaf*:
  - repräsentiert atomare Komponente
  - definiert Verhalten für atomare Komponenten

```
public abstract class Leaf extends Component {
    public abstract void operation ();
}
```

Beschreibung von Mustern (Gamma u. a. 2003) III

- *Composite*:
  - definiert Standardverhalten für zusammengesetzte Komponenten
  - speichert Teile
  - implementiert Operationen zur Verwaltung von Teilen

```java
import java.util.List;
import java.util.ArrayList;
public abstract class Composite extends Component {

 private List<Component> children
  = new ArrayList<Component>();

 public void operation () {
   for (Component c : children) c.operation ();
 }
 public void add (Component c) {children.add (c);}
 public void remove (Component c) {children.remove (c);}
 public Component getChild (int i) {return children.get (i);
}
```

Beschreibung von Mustern (Gamma u. a. 2003) I

Kollaborationen:

- *Clients* benutzen *Component*-Schnittstelle
- falls Empfänger ein *Leaf* ist, antwortet es direkt
- falls Empfänger ein *Composite* ist, wird die Anfrage an Teile weitergeleitet (möglicherweise mit weiteren eigenen Operationen vor und/oder nach der Weiterleitung)

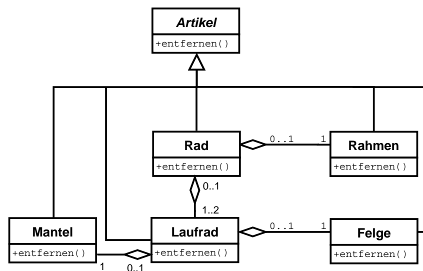
Beschreibung von Mustern (Gamma u. a. 2003) II

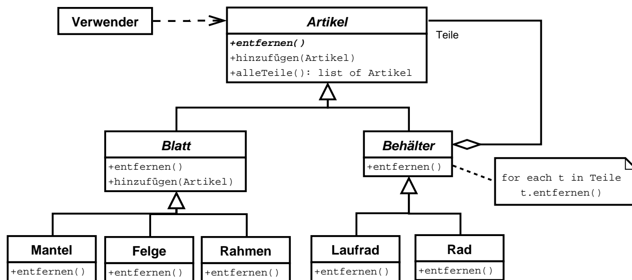
Konsequenzen:

- zweiteilt die Klassenhierarchie in *Leaf* und *Composite* mit einheitlicher Schnittstelle
- uniforme Verwendung auf Seiten des *Clients*
- neue Komponenten können leicht hinzugefügt werden
- könnte die Struktur unnötig allgemein machen (dynamische versus statische Typisierung)

Implementierung: . . . Hinweise zur Implementierung . . .

Dynamische versus statische Typisierung





Kategorien von Entwurfsmustern

- Erzeugungsmuster
  - betreffen die Erzeugung von Objekten
  - Beispiel: Factory Method
- Strukturelle Muster:
  - betreffen Komposition von Klassen und Objekten
  - Beispiel: Composite
- Verhaltensmuster:
  - betreffen Interaktion und Verantwortlichkeiten
  - Beispiel: Observer

Fahrradladenbeispiel: Erweiterung für andere Domänen

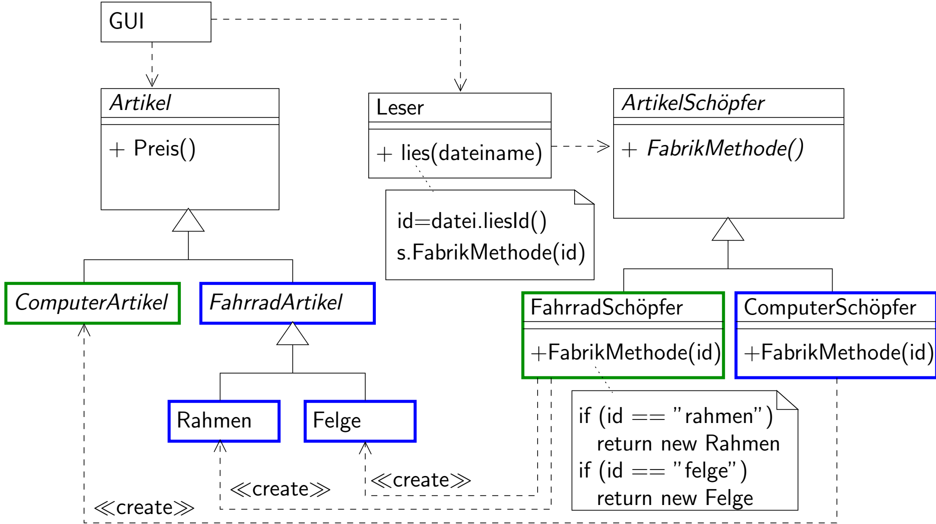
Anforderungen:

- Artikeldaten sollen von einer Datei gelesen werden können
- zukünftig sollen andere Domänen unterstützt werden (Fahrrad, Computer und Klettern)
- die Objekte dieser Domänen sind unterschiedlich
- notwendige Anpassungen sollen einfach realisiert werden können

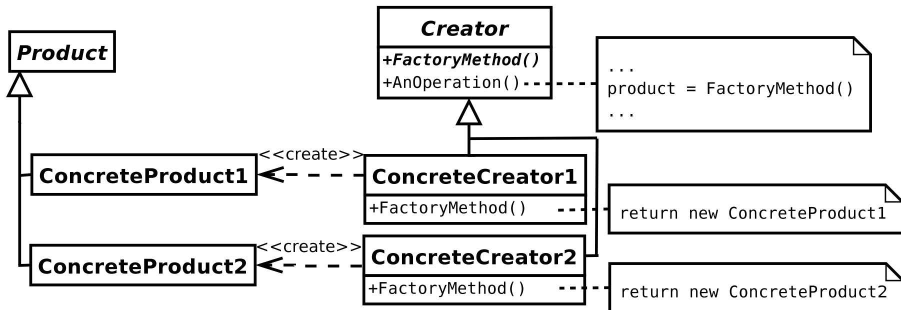
Lösungsstrategien:

- die Klassen der Benutzungsschnittstelle beziehen sich nur auf die Schnittstelle der abstrakten Klasse Artikel
- Datei hat gleiche Syntax für alle Domänen (nur die Inhalte variieren)
- die Artikel werden beim Einlesen der Datei als Objekte erzeugt
  - → aber der Leser muss doch die Konstruktoren der Objekte kennen, oder was?

Lösungsstrategie



Entwurfsmuster: *Factory Method*



Anwendbarkeit

- eine Klasse weiß in manchen Fällen nicht im Voraus, von welcher Klasse ein zu erzeugendes Objekt sein soll
- die konkreten Unterklassen einer Klasse sollen dies entscheiden
- Verantwortlichkeit wird an Unterklassen delegiert, und das Wissen über die Unterklasse, an die delegiert wird, soll nur an einem Punkt vorhanden sein

Teilnehmer

- Product
  - deklariert die Schnittstelle von Objekten, die die Fabrikmethode erschafft
- ConcreteProduct
  - implementiert die Product-Schnittstelle
- Creator
  - deklariert die Fabrikmethode
  - (optional) implementiert eine Standardfabrikmethode, die ein spezifisches konkretes Objekt erzeugt
  - kann Fabrikmethode aufrufen, um ein Product-Objekt zu erzeugen
- ConcreteCreator
  - überschreibt die Fabrikmethode, um konkretes Product-Objekt zu erschaffen

Entwurfsmuster Observer

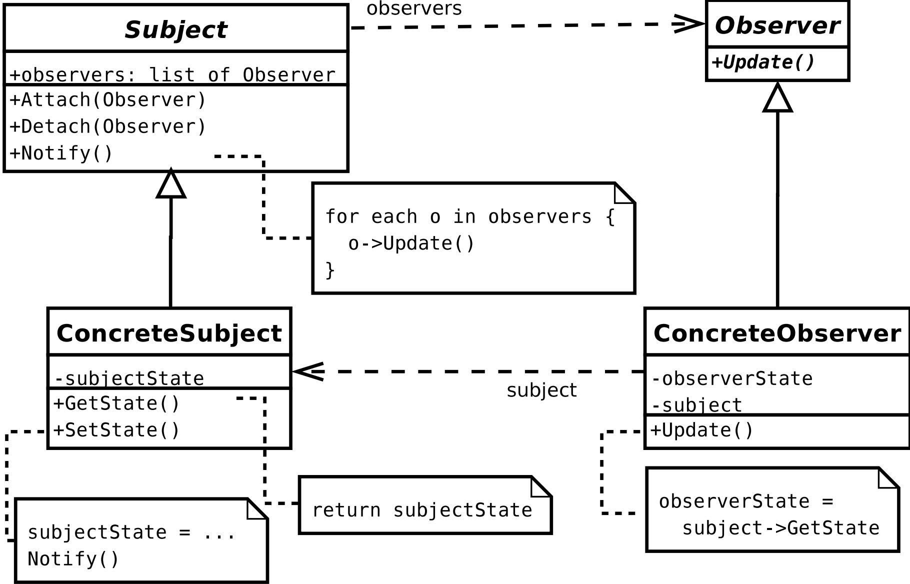
Anwendbarkeit

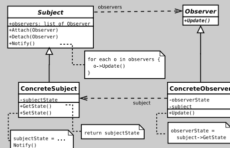
- Komponenten hängen von anderen Komponenten ab
- Änderung der einen Komponente muss Änderung der anderen nach sich ziehen
- Komponenten sollen lose gekoppelt sein: Komponente kennt seine Abhängigen nicht im Voraus (zur Übersetzungszeit)

Lösungsstrategie

- Abhängige registrieren sich bei Komponente
- Komponente informiert alle registrierten Abhängigen über Zustandsänderung

Entwurfsmuster Observer: Struktur





Man beachte die Beziehung zu den Architekturmustern Blackboard und Model-View-Controller.

Entwurfsmuster Observer: Teilnehmer *Subject*

- kennt seine Observer (zur Laufzeit)
- kann beliebig viele Observer haben
- stellt Schnittstelle zur Verfügung, um Observer zu registrieren und abzutrennen

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

public abstract class Subject {

 private List<Observer> observers
  = new ArrayList<Observer>();

 public void Attach (Observer c) {observers.add (c);}
 public void Detach (Observer c) {observers.remove (c);}
 public void Notify () {
   for (Observer o : observers) o.update ();
 }
}
```

Entwurfsmuster Observer: Teilnehmer *Observer*

- deklariert Schnittstelle für die Update-Nachricht

```
public abstract class Observer {
    public abstract void update ();
}
```

Entwurfsmuster Observer: Teilnehmer *ConcreteSubject*

- hat einen Zustand, der ConcreteObserver interessiert
- sendet Bekanntmachung via Notify(), wenn sich Zustand ändert (SetState)

```
public class Ampel extends Subject {

    private boolean rot = false;

    public boolean ist_rot () {return rot;}

    public void schalten () {
        set (!rot);
    }

    private void set (boolean newValue) {
        rot = newValue;
        Notify ();
    }
}
```

Entwurfsmuster Observer: Teilnehmer *ConcreteObserver*

- kennt ConcreteSubject-Objekt
- verarbeitet Zustand dieses Subjects
- implementiert Update, um auf veränderten Zustand zu reagieren

```
public class Auto extends Observer {
 private Ampel ampel;
 private boolean gaspedal = false;
 public void update () {
   if (!ampel.ist_rot()) fahren ();
 }
 public void fahren () {
  ampel.Detach (this);
  gaspedal = true;
 }
 public void stoppen (Ampel a) {
   ampel = a;
   ampel.Attach (this);
   gaspedal = false;
 }
}
```

Entwurfsmuster Observer: Konsequenzen

- abstrakte Kopplung zwischen Subject und Observer
- unterstützt Rundfunk (Broadcast)
- unerwartete Updates, komplizierter Kontrollfluss
- viel Nachrichtenverkehr, auch dann wenn sich ein irrelevanter Aspekt geändert hat

Entwurfsmuster Observer: Verfeinerungen

- Push-Modell
  - Subject sendet detaillierte Beschreibung der Änderung

  → umfangreiches Update

  → vermeidet GetState(), aber nicht Update()
- Pull-Modell
  - Subject sendet minimale Beschreibung der Änderung

  → Observer fragt gegebenfalls die Details nach

  → erfordert weitere Nachrichten, um Details abzufragen
- Explizite Interessen
  - Observers melden Interesse an spezifischem Aspekt an; Aspekt wird zusätzlicher Parameter von Update

Architekturstile

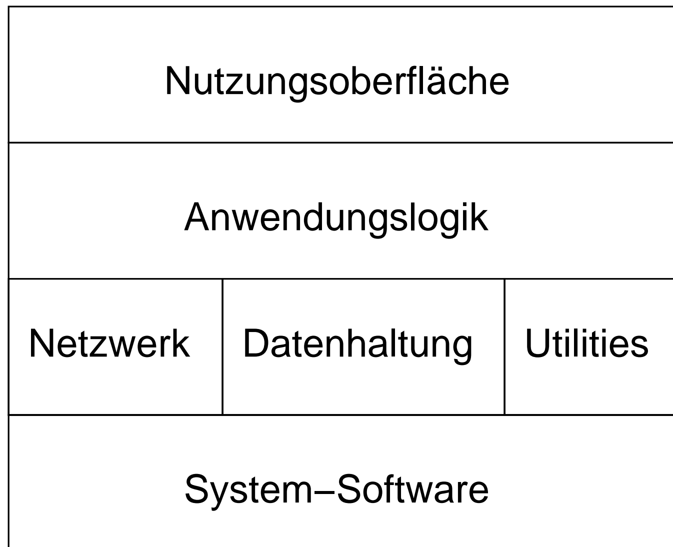
Definition

Architekturstil: beschreibt eine Familie von Architekturen/Systeme als ein Muster der strukturellen Organisation durch

- ein Vokabular (Komponenten- und Konnektorentypen)
- und eine Menge von Einschränkungen, wie Komponenten und Konnektoren verbunden werden dürfen.

Synonyme: Architekturmuster oder Architekturidiom.

Architekturstil: Schichtung



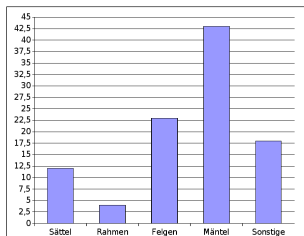
Architekturstil: Schichtung I

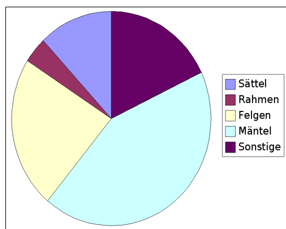
- Vokabular:
  - Komponenten: Module und Schichten
  - Konnektoren: Use-Beziehung
- Struktur:
  - Module sind eindeutig einer Schicht zugeordnet
  - Module einer Schicht dürfen nur auf Module derselben und der direkt darunter liegenden Schicht zugreifen
- Ausführungsmodell:
  - Aufruf von Methoden tieferer Schichten
  - Datenfluss in beide Richtungen (von der unteren Schicht zur oberen durch Rückgabeparameter)

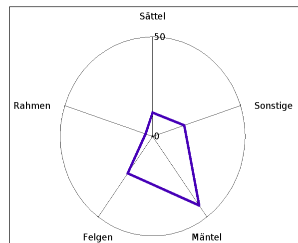
Architekturstil: Schichtung II

- Vorteile:
  - Schicht implementiert virtuelle Maschine, deren Implementierung leicht ausgetauscht werden kann, ohne dass höhere Schichten geändert werden müssen
- Nachteile:
  - höherer Aufwand durch das „Durchreichen“ von Information
  - Redundanz durch Dienste tieferer Schichten, die in hohen Schichten benutzt und auf allen Ebenen dazwischen repliziert werden

Anforderungen







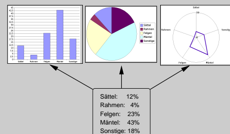
Sättel: 12%

Rahmen: 4%

Felgen: 23%

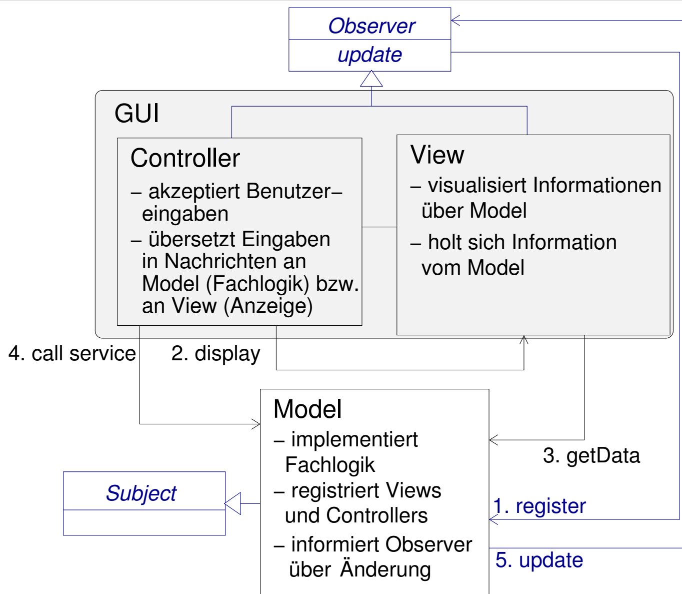
Mäntel: 43%

Sonstige: 18%



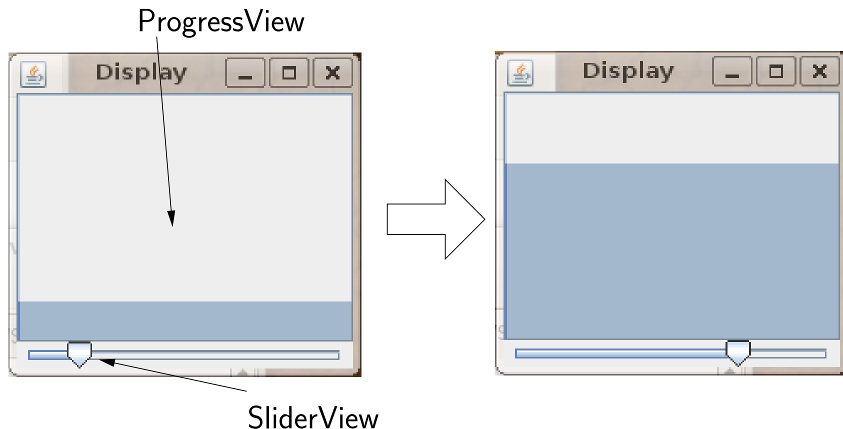
Unterschiedliche Diagrammarten stellen statistische Daten über die nachgefragten Artikel dar. Die Diagramme müssen alle konsistent zu den aktuellen Daten sein. Neue Diagramme sollen leicht hinzugefügt werden können.

Model-View-Controller (Buschmann u. a. 1996)

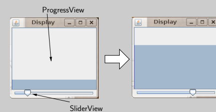


Model-View-Controller Beispiel

Ein Model: Wert in festem Bereich



Ein Controller: Verschiebung in SliderView verändert Model



Dieses Beispiel und der Code dazu stammen von Thilo Mende. Ihm gilt mein Dank.

Beispiel-Model: Zustand

```
1  import java.util.ArrayList;
2  import java.util.List;
3  import javax.swing.event.ChangeEvent;
4  import javax.swing.event.ChangeListener;
5
6  public class Model {
7
8    // Wertebereich:
9    private final int min = 0;   // untere Grenze
10   private final int max = 100; // obere Grenze
11
12   // Zustand
13   private int value = 50;
```

Beispiel-Model: Oberver-Behandlung

```
1    // alle Observer
2    private List<ChangeListener> observers;
3
4    // Kontruktor
5    public Model(){
6      observers = new ArrayList<ChangeListener>();
7    }
8
9    // neuen Observer registrieren
10   public void addChangeListener(ChangeListener observer) {
11     observers.add(observer);
12   }
13
14   // Benachrichtigung aller Observer;
15   // aufzurufen bei allen Zustandsänderungen
16   private void Notify(){
17     for(ChangeListener observer: observers){
18       observer.stateChanged(new ChangeEvent(this));
19     }
20   }
```

Beispiel-Model: Setter und Getter

```
1    public int getMin() {return min;}
2
3    public int getMax() {return max;}
4
5    public int getValue() {return value;}
6
7    public void setValue(int value) {
8      if (value < min){
9        this.value = min;
10     } else if (value > max){
11       this.value = max;
12     } else {
13       this.value = value;
14     }
15     this.Notify();
16     // Zustand hat sich geändert
17   }
```

Beispiel-Controller

```java
1 public class Controller {
2
3   private Model model; // das Model des Controllers
4
5   // In dieser Variante kennt der Controller seine
6   // View nicht; die Assoziation von View und Controller
7   // wird vom Hauptprogramm etabliert
8
9   public Controller(Model model) {
10    this.model = model;
11  }
12
13  public void setValue(int value) {
14    model.setValue(value);
15  }
16 }
```

Beispiel-ProgressView I

```
1  import javax.swing.JProgressBar;
2  import javax.swing.event.ChangeEvent;
3  import javax.swing.event.ChangeListener;
4
5  public class ProgressView extends JProgressBar
6                            implements ChangeListener {
7
8      private Controller controller; // View kennt Controller
9      private Model my_model;
10
11     public ProgressView(Controller controller, Model my_model){
12         super(VERTICAL);
13         this.controller = controller;
14         this.my_model   = my_model;
15
16         // Zustand von Model darstellen:
17         this.setValue(my_model.getValue());
18
19         // Registrierung bei Model
20         my_model.addChangeListener(this);
21     }
```

Beispiel-ProgressView II

```
1       // Redefinition; ererbt von ChangeListener;
2       // entspricht update
3       public void stateChanged(ChangeEvent arg0) {
4           synchronized(this){
5           this.setValue(my_model.getValue());}
6       }
7
8 } // Ende ProgressView
```

Beispiel-SliderView I

```
 1 import javax.swing.JSlider;
 2 import javax.swing.event.ChangeEvent;
 3 import javax.swing.event.ChangeListener;
 4
 5 public class SliderView extends JSlider
 6                          implements ChangeListener {
 7     private Controller my_controller; // View kennt Controller
 8     private Model my_model;
 9
10     // Redefinition; ererbt von ChangeListener;
11     // entspricht update
12     public void stateChanged(ChangeEvent arg0) {
13         this.setValue(my_model.getValue());
14     }
```

Beispiel-SliderView II

```
1      // Konstruktor
2      public SliderView(final Controller controller, Model model) {
3          super(model.getMin(), model.getMax(), model.getValue());
4          this.my_model      = model;
5          this.my_controller = controller;
6
7          // Registrierung bei Model
8          model.addChangeListener(this);
9
10         // Zustand von Model darstellen:
11         this.addChangeListener(new ChangeListener() {
12            public void stateChanged(ChangeEvent arg0) {
13                if (!getValueIsAdjusting()){
14                    my_controller.setValue(getValue());
15                }
16            }
17         });
18     }
19 } // Ende von SliderView
```

Beispiel-Hauptprogramm I

```
1 import java.awt.BorderLayout;
2 import javax.swing.JFrame;
3 import javax.swing.WindowConstants;
4
5 public class MVCDemo {
6   public static void main(String[] args) {
7     MVCDemo demo = new MVCDemo();
8   }
9 }
```

Beispiel-Hauptprogramm II

```
1      public MVCDemo() {
2        Model model = new Model();
3
4        // Assoziation desselben Controllers mit beiden Views
5        Controller controller = new Controller(model);
6        ProgressView progress = new ProgressView(controller, model);
7        SliderView slider   = new SliderView(controller, model);
8
9        // Fenster um alles
10       JFrame displayFrame = new JFrame("Display");
11       displayFrame.getContentPane()
12                         .add(progress, BorderLayout.CENTER);
13       displayFrame.getContentPane()
14                         .add(slider, BorderLayout.SOUTH);
15       displayFrame.setDefaultCloseOperation
16                         (WindowConstants.EXIT_ON_CLOSE);
17       displayFrame.pack();
18       displayFrame.setVisible(true);
19     }
20
21   } // End MVCDemo
```

Wiederholungsfragen

- Was ist ein Entwurfsmuster?
- Warum sind sie interessant für die Software-Entwicklung?
- Erläutern Sie eines der in der Vorlesung vorgestellten Entwurfsmuster.
- Was ist ein Architekturstil?
- Nennen Sie Beispiele für Architekturstile. Erläutern Sie die Stile.

Weiterführende Literatur

- Buschmann u. a. (1996) beschreiben Architekturstile bzw. -muster
- Shaw und Garlan (1996) geben eine Einführung in Software-Architektur und beschreiben einige Architekturstile bzw. -muster
- Das Standardbuch zu Entwurfsmustern ist das von Gamma u. a. (2003)

1 **Buschmann u. a. 1996** BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: Pattern-oriented Software Architecture: A System of Patterns. Bd. 1. Wiley, 1996

2 **Gamma u. a. 2003** GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: Desig Patterns—Elements of Reusable Object-Oriented Software. Addison Wesley, 2003

3 **Shaw und Garlan 1996** SHAW, Mary ; GARLAN, David: Software Architecture – Perspectives on an Emerging Discipline. Upper Saddle River, NJ : Prentice Hall, 1996. – ISBN ISBN 0-13-182957-2