Medientechnik

Übung - Einführung

Übungsbetrieb

- Informationen zu den Übungen: http://www.mimuc.de/mt

- (beinahe) jede Woche
- Programmierübungen und Vorbereitung auf die Praktika

Scheinvoraussetzungen

- an allen drei Praktika teilnehmen
- alle Hausaufgaben zu den Praktika abgeben
- mindestens 50% der Punkte aller Übungsblätter erreichen (ein Freischuss)
- am Ende des Semesters (Praktikums-) Ergebnisse präsentieren
- KEINE Klausur!

Zeitplan

	Datum		Übungen	Einführung Praktikum	Praktikum
April	24.04.-01.05		**Übung 1: SWING + MVC**		
Mai	01.05.-08.05			**Einführung Foto (Retusche)**	
	08.05.-15.05.		**Übung 2: Java2D**		Praktikum Foto
	15.05.-22.05.		**Übung 3: Bildbearbeitung**		
	22.05.-29.05			**Einführung Video**	
Juni	29.05.-05.06.			**Einführung Rechte**	Praktikum Video
	05.06.-12.06.		**Übung 4: Effekte in Video**		
	12.06.-19.6.			**Einführung Audio**	
	19.06.-26.06.		**Übung 5: Java Sound**		Praktikum Audio
Juli	26.06.-03.07.				
	03.07.-10.07.				
	10.07.-17.07.				
	17.07.-24.07.				

Heute

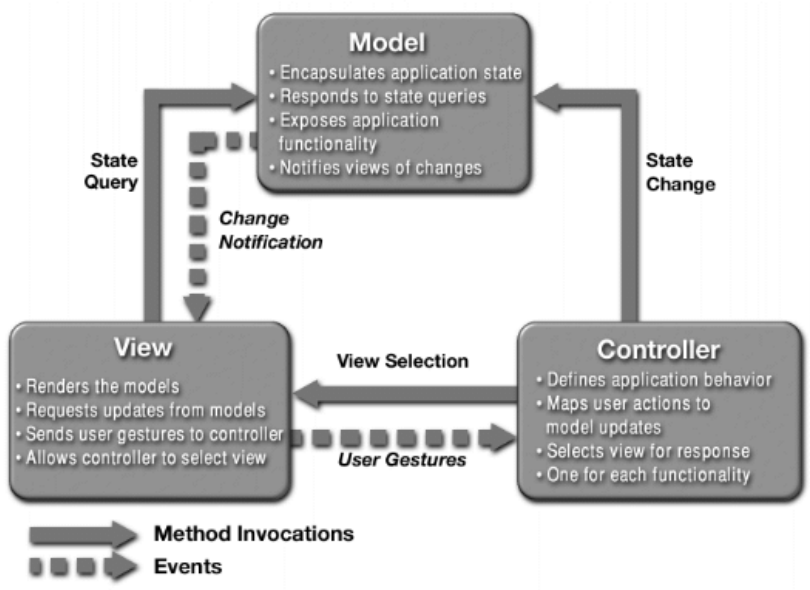
- Model-View-Controller
- Observer
- Java Swing
- Programmieren und Debuggen

Entwurfsmuster

- wiederverwendbare Lösung für ein regelmäßig auftretendes Problem
- grobe Klassen- und Methodenvorgaben, daher vor allem für objektorientierte Programmierung
- vom Prinzip her aber in gesamter Softwareentwicklung anwendbar (z.B. für Projektmanagement, GUIs etc.)

Model-View-Controller

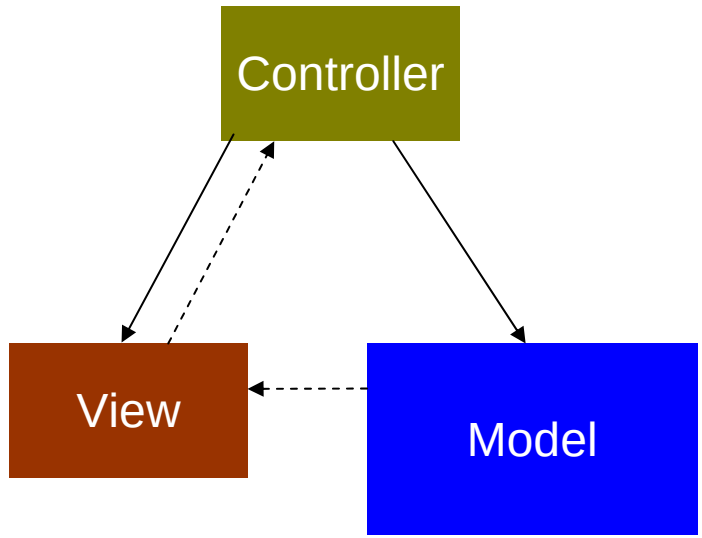
- Model repräsentiert Informationen
- View zeigt diese Informationen an und stellt dem Benutzer Interaktionsmöglichkeiten zur Verfügung
- Controller reagiert auf Eingaben die über die View kommen und verändert das Model



Quelle: http://java.sun.com/developer/technicalArticles/javase/mvc/

Model-View-Controller

- Vorteile (u.a.):
  - Unabhängigkeit zwischen Modell und Präsentation => sowohl Präsentation als auch Modell sind austauschbar
  - mehrere Views können auf das gleiche Modell zugreifen und werden automatisch aktualisiert (Multiple Coordinated Views)
  - klarere Trennung => übersichtlicher, weniger Fehler



MVC

```
import java.awt.event.*;

public class Controller implements ActionListener {

     private MyObservable model;
     private MyObserver view;

     public Controller(MyObserver view){
          this.model = new MyObservable();
          this.view = view;
          this.model.addObserver(this.view);
     }

     public void actionPerformed(ActionEvent arg0) {
          this.model.increase();
     }
}
```

Controller

Controller fungiert als ActionListener

Implementiert die *actionPerformed*-Methode

Controller kennt Model und View

Fügt dem Model einen Observer, den View, hinzu

Controller.java

Swing - Textfeld

```
public class MyObserver extends JFrame implements Observer {

	private JTextArea zahl;

	public MyObserver(){
		super("Observer");
		this.setDefaultCloseOperation(
					WindowConstants.EXIT_ON_CLOSE);
		this.setSize(200, 100);

		this.setLayout(new BorderLayout());
		zahl = new JTextArea("zahl");
		zahl.setEditable(false);
		this.getContentPane().add(zahl);

		JButton but = new JButton("zählen!");
		but.addActionListener(new Controller(this));
		this.getContentPane().add(but, BorderLayout.SOUTH);
	}
```

Observer

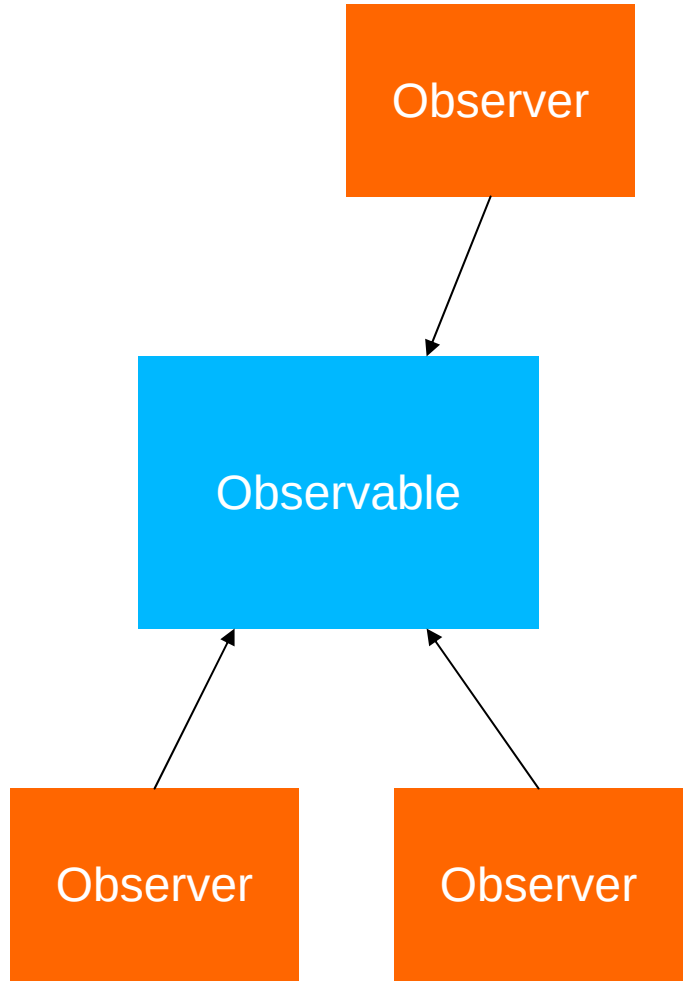
View

Controller hört nun auf ActionEvents auf dem Button

MyObserver.java

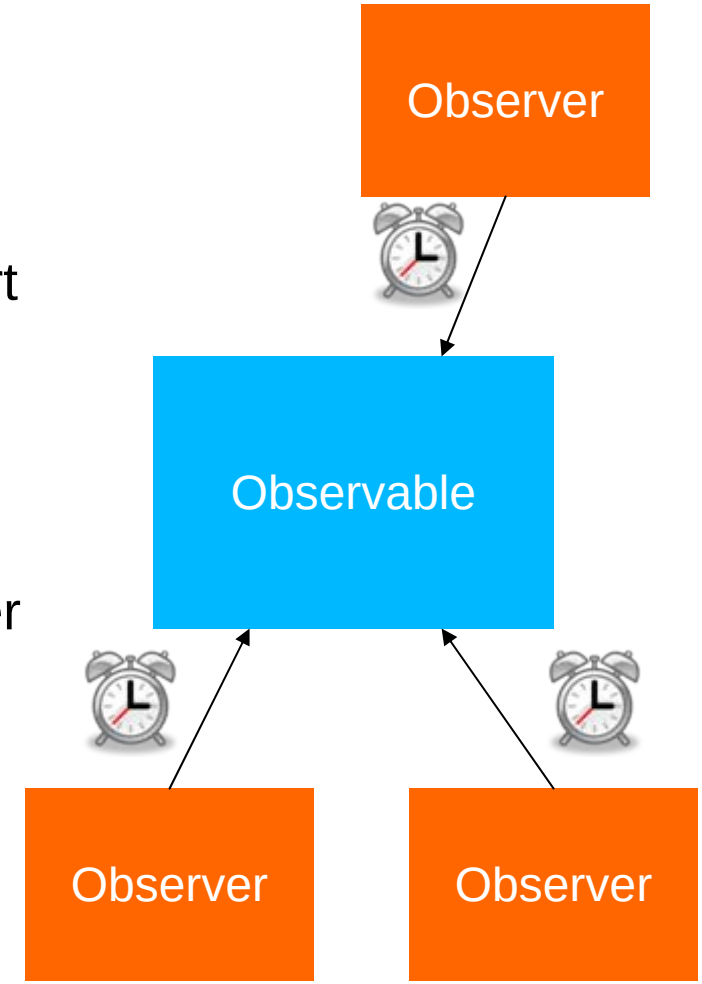
Observer

- Observer interessieren sich für Information in Observable, die sich ändern kann
- Beispiel: Eingabegerät (Maus, Tastatur)



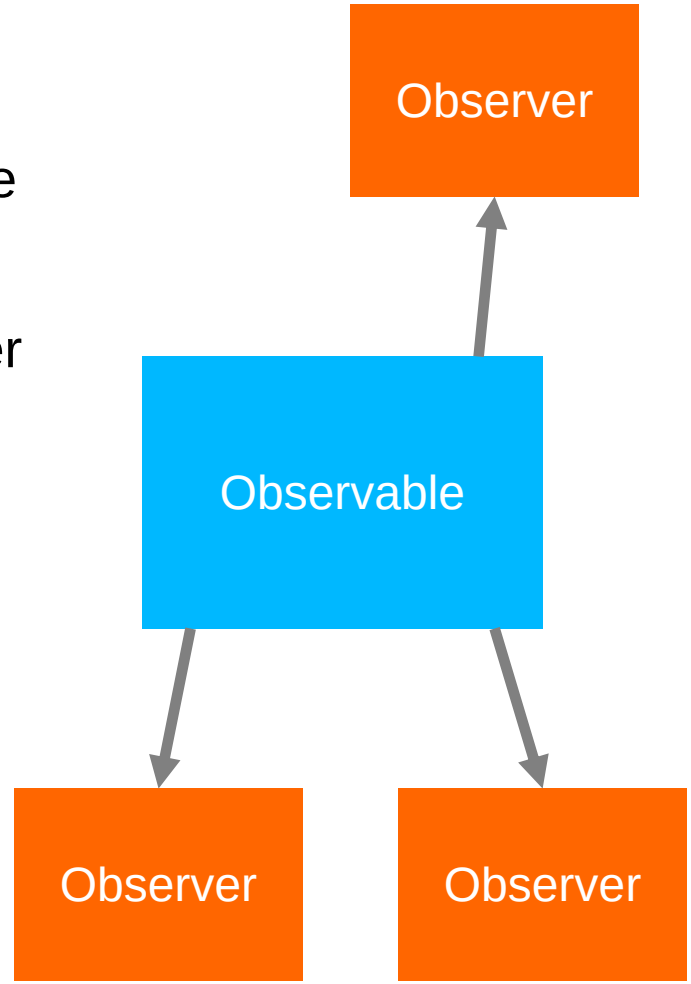
Observer

- Lösung:
  - Jeder Observer fragt in regelmäßig zeitlichen Abständen bei Observable nach, ob sich die Information geändert hat (Polling)
- Problem:
  - Funktioniert zwar ganz gut z.B. bei der Maus, aber:
  - Wenn sich die Information nicht oft ändert ist der Großteil der Anfragen überflüssig



Observer

- Besser:
  - Observable weiß genau, wann sich die relevante Information ändert und benachrichtigt dann die Observer!
  - Dazu müssen sich die Observer vorher bei Observable anmelden, damit dieses weiß, wer zu benachrichtigen ist.



Observer in Java

Observer

- java.util.Observer (Interface!)
  - Methoden:

```
update(Observable o, Object
    arg)
```

Observable

- java.util.Observable
  - Methoden:

```
addObserver(Observer o)
deleteObserver(Observer o)

setChanged()
notifyObservers()
notifyObservers(Object arg)
```

Beispiel

Observer

```
import java.util.*;

public class MyObserver implements Observer {

    public void update(Observable arg0, Object arg1)
    {
        //do something...
    }

}
```

MyObserver.java

Beispiel

Observable

```
import java.util.*;
public class MyObservable extends Observable {
	private int counter;

	public MyObservable(){
		this.counter = 0;
	}

	public void increase(){
		this.counter++;
		this.setChanged();
		this.notifyObservers();
	}

	public int getCounter(){
		return this.counter;
	}
}
```

Model benachrichtigt Observer nach einer Änderung

MyObservable.java

Swing

```
import java.util.*;
import javax.swing.*;

public class MyObserver extends JFrame
                        implements Observer {

    public MyObserver(){
        super("Observer");
        this.setDefaultCloseOperation(
                    WindowConstants.EXIT_ON_CLOSE);
        this.setSize(200, 100);
    }

    public void update(Observable arg0, Object arg1)
    {
```

Observer

Funktionen auf einem JFrame:
Reagieren auf Schließen des Fensters und Setzen der Größe

MyObserver.java

Swing

```
    public void update(Observable arg0, Object a
    {
        //do something...
    }

    public static void main(String[] args){
        MyObserver obs = new MyObserver();
        obs.setVisible(true);
    }
}
```

Observer

MyObserver.java

Swing - Textfeld

```
import java.util.*;
import javax.swing.*;
import java.awt.*;

public class MyObserver extends JFrame implements Observer {

    private JTextArea zahl;

    public MyObserver(){
        super("Observer");
        this.setDefaultCloseOperation(
                        WindowConstants.EXIT_ON_CLOSE);
        this.setSize(200, 100);

        this.setLayout(new BorderLayout());
        zahl = new JTextArea("zahl");
        zahl.setEditable(false);
        this.getContentPane().add(zahl);
    }
```

Observer

Layout festlegen
Neues Element erzeugen
Zum Frame dazufügen

MyObserver.java

Swing - Textfeld

```
public void update(Observable arg0, Object arg1) {

    MyObservable observed = (MyObservable)arg0;
    zahl.setText(String.valueOf(observed.getCounter()));

}

public static void main(String[] args){
    MyObserver obs = new MyObserver();
    obs.setVisible(true);
}
}
```

Observer

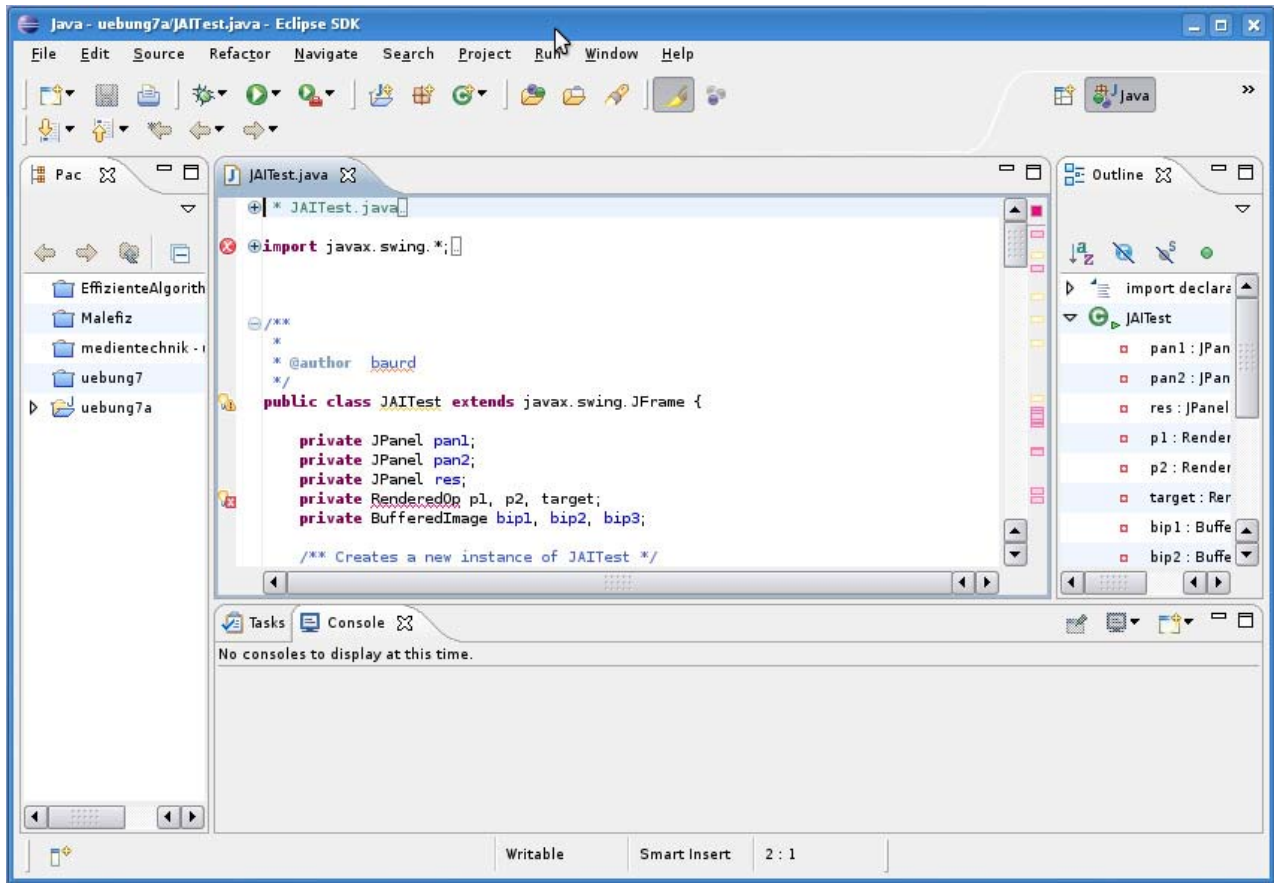
Ändern des Inhalts eines Textfelds nach Änderungen im Model

MyObserver.java

Eclipse

Im CIP-Pool:

/soft/bin/eclipse-ide-3.3



Eclipse

- Kurze Demo
  - Projekt öffnen
  - Neue Klassen erstellen
  - Ausführen
  - Debuggen