Medientechnik

Übung

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 in jedem Übungsblatt erreichen (ein Freischuss)
- am Ende des Semesters (Praktikums-) Ergebnisse präsentieren
- KEINE Klausur!

Zeitplan

	Woche		Übung			Praktikum
April	**21.04. - 25.04.**			**Swing**		Foto 1
Mai	**28.04. - 02.05.**		**Bildbearbeitung**			Foto 2
	05.05. - 09.05.					Foto 3
	12.05. - 16.05.			**Java 2D**		Foto 4
	19.05. - 23.05.			**Java 3D**		Video 1
	26.05. - 30.05.		**Visual Effects**			Video 2
Juni	**02.06. - 06.06.**			**Java Media Framework**		Video 3
	09.06. - 13.06.					Video 4
	16.06. - 20.06.					Audio 1
	23.06. - 27.06.		**Audiotechnik**			Audio 2
Juli	**30.06. - 04.07.**			**Java Sound**		Audio 3
	07.07. - 11.07.					Audio 4
	14.07. - 18.07.					Abschlußpräsentation

Heute

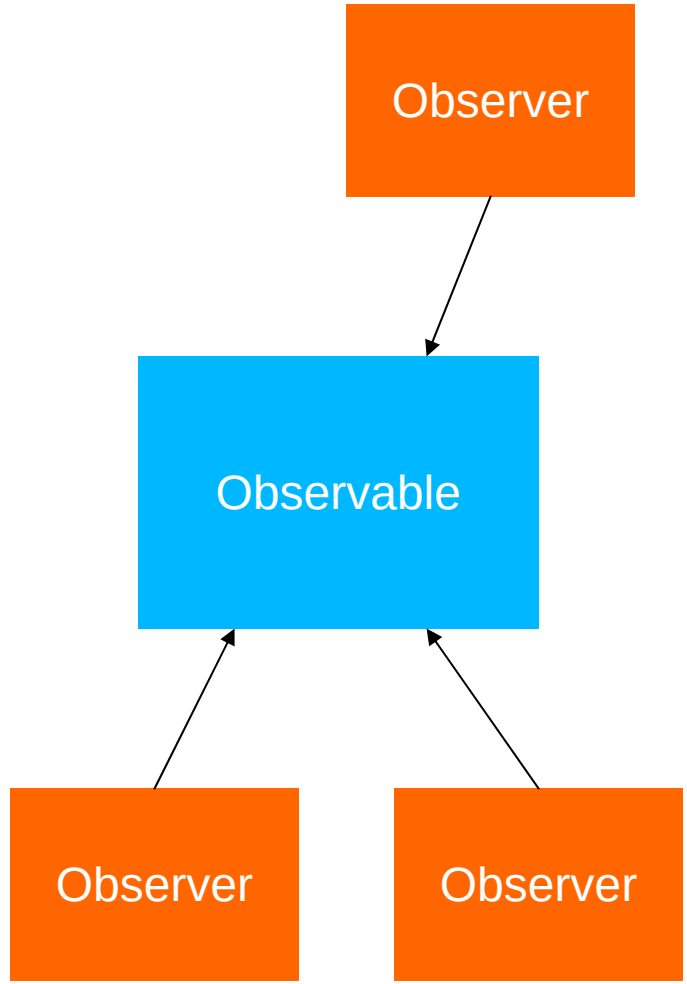
- Observer
- Model-View-Controller

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

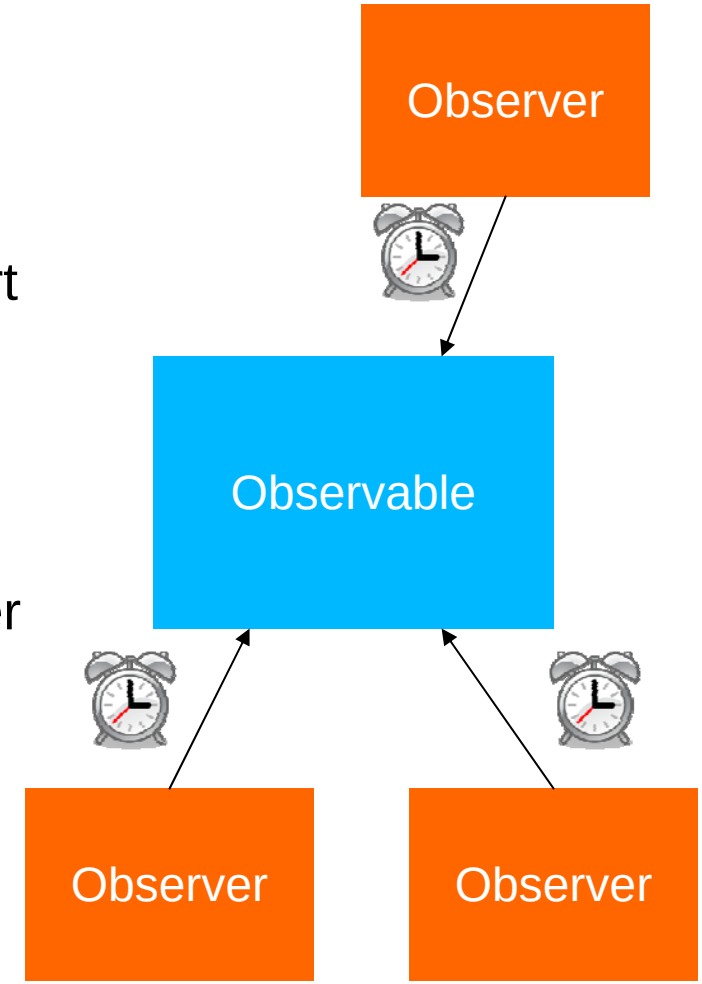
Observer

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



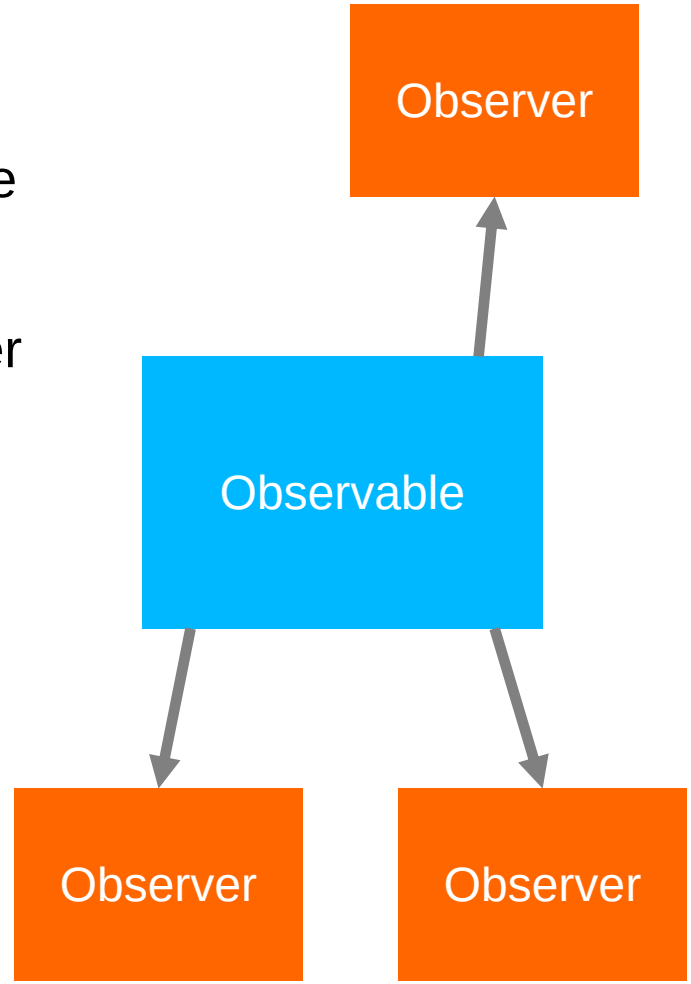
Observer

- 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

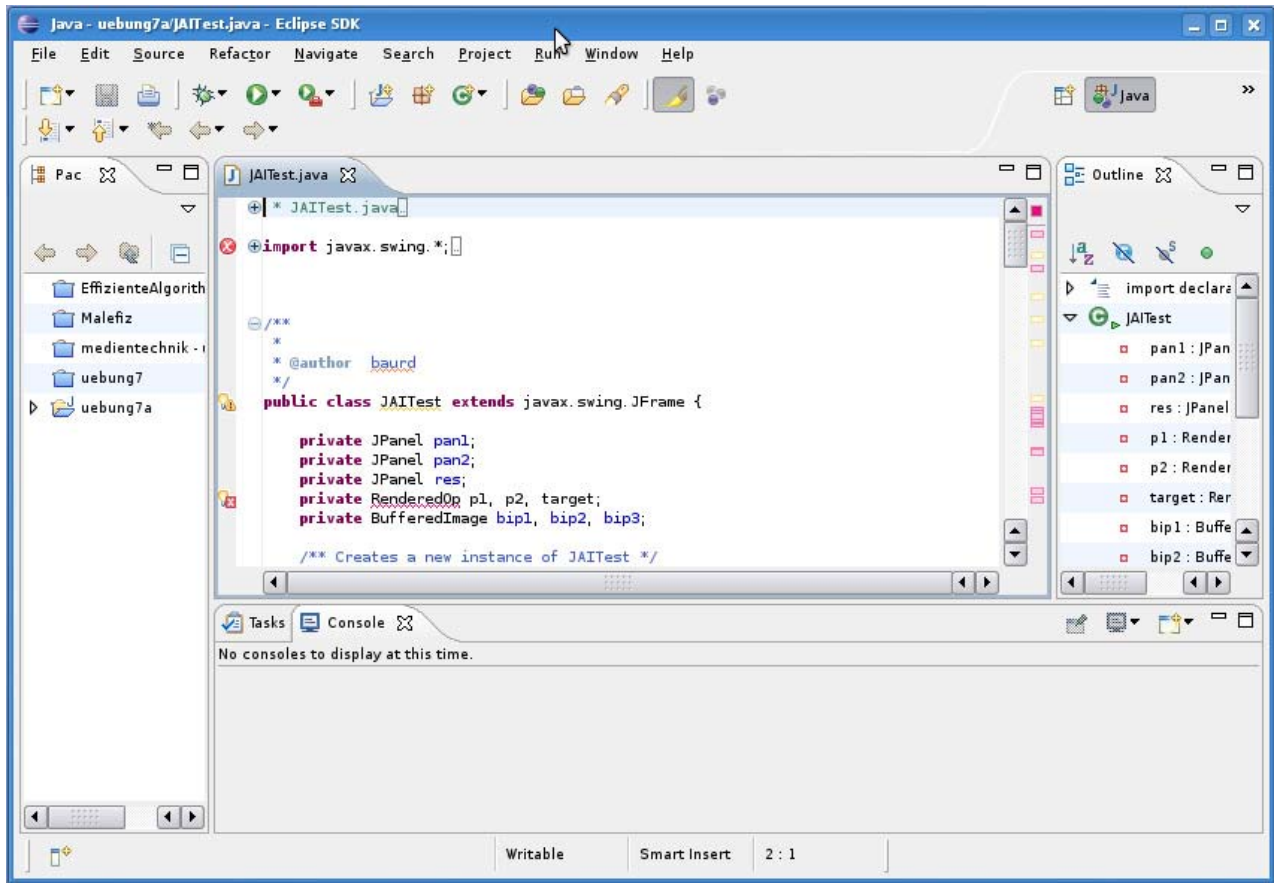
- java.util.Observable
  - Methoden:

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

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

Eclipse

Im CIP-Pool:

/soft/bin/eclipse-ide-3.3



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;
	}
}
```

MyObservable.java

Swing

Observer

```
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)
    {
```

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

Observer

```
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);
	}
```

MyObserver.java

Swing - Textfeld

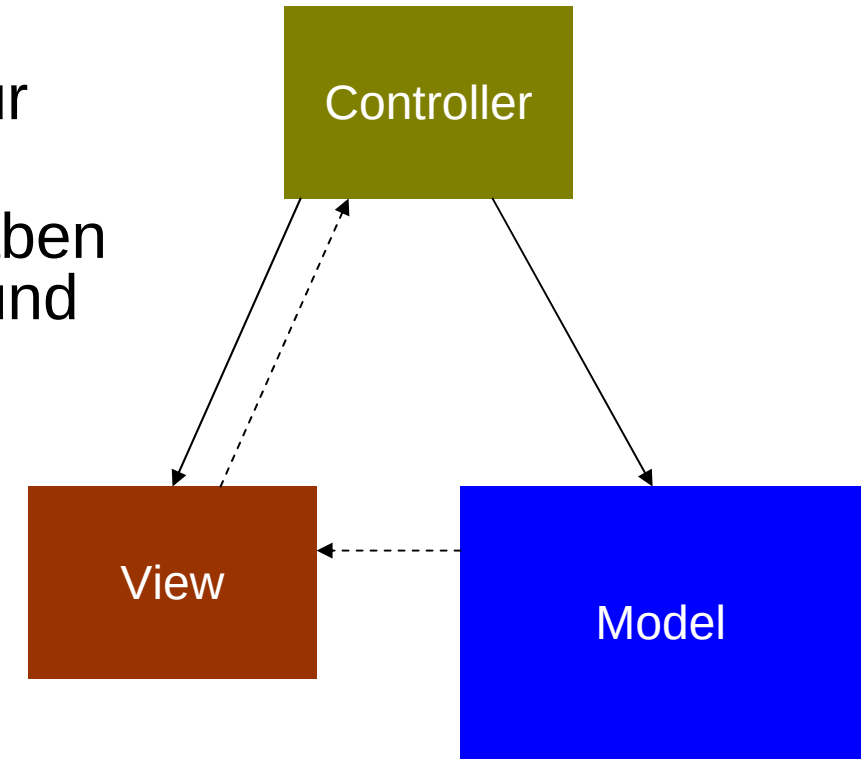
```
    public void update(Observable arg0, Object arg1) {

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

    }

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

Observer

MyObserver.java

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



MVC

Controller

```
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.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

MyObserver.java

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