Programmierung 2
Studiengang MI / WI

Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm

Hochschule Harz

FB Automatisierung und Informatik

mwilhelm@hs-harz.de

Raum 2.202

Tel. 03943 / 659 338

Inhalt der Vorlesung

Überblick:

- Objekte und Methoden
- Swing
- Exception
- I/O-Klassen / Methoden
- Threads
- Algorithmen (Das Collections-Framework)
- **Design Pattern**
- Graphentheorie
- JUnit

Internet-Adressen

- www.dofactory.com/Patterns/
- www.patterndepot.com
- www.javapractices.com
- http://www.se.uni-hannover.de/documents/kurz-und-gut/creational-patterns_tliro.pdf

Literatur

- Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

- Holub on Patterns:

Learning Design Patterns by Looking at Code, 978-1590593882

- Pattern-Oriented Software Architecture

Volume 1, ISBN 978-0471958697

- Pattern-Oriented Software Architecture

Volume 2, ISBN: 978-0471606956

Definition von Muster mit Lösungsschemen

Der Begriff des Musters wurde vom Architekten Christopher Alexander 1977 wie folgt definiert:

„Jedes Muster beschreibt ein in unserer Umwelt beständiges, wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.“

Ein Muster stellt also eine bewährte Lösung nach Art einer Schablone für ein häufiges Probleme bereit. So gesehen sind bereits die länger existierenden Algorithmensammlungen wie z.B. „The Art of Computer Programming“ von Donald Knuth eine Katalogisierung von Mustern (vergl. [Gamma97, S.392]).

Die Beschreibung eines Musters enthält im allgemeinen folgende Abschnitte:

- Name: Im Idealfall Charakterisierung der Auswirkungen des Musters in einem Wort.
- Zweck: Was macht das Muster? Was ist das Grundprinzip, was ist sein Zweck? Welches Problem, in welchem Umfeld macht den Einsatz des Musters sinnvoll?
- Auch bekannt als
- Struktur: Beschreibung des eigentlichen Musters, textuell und Klassen-, eventuell ein Sequenzdiagramm.
- Konsequenzen: Vor- und Nachteile, die durch den Einsatz des Musters in einem Entwurf entstehen.
- Implementierung: Der Abschnitt präsentiert die Fallen, Tipps oder Techniken, die man kennen sollten, wenn man das Muster einsetzt.

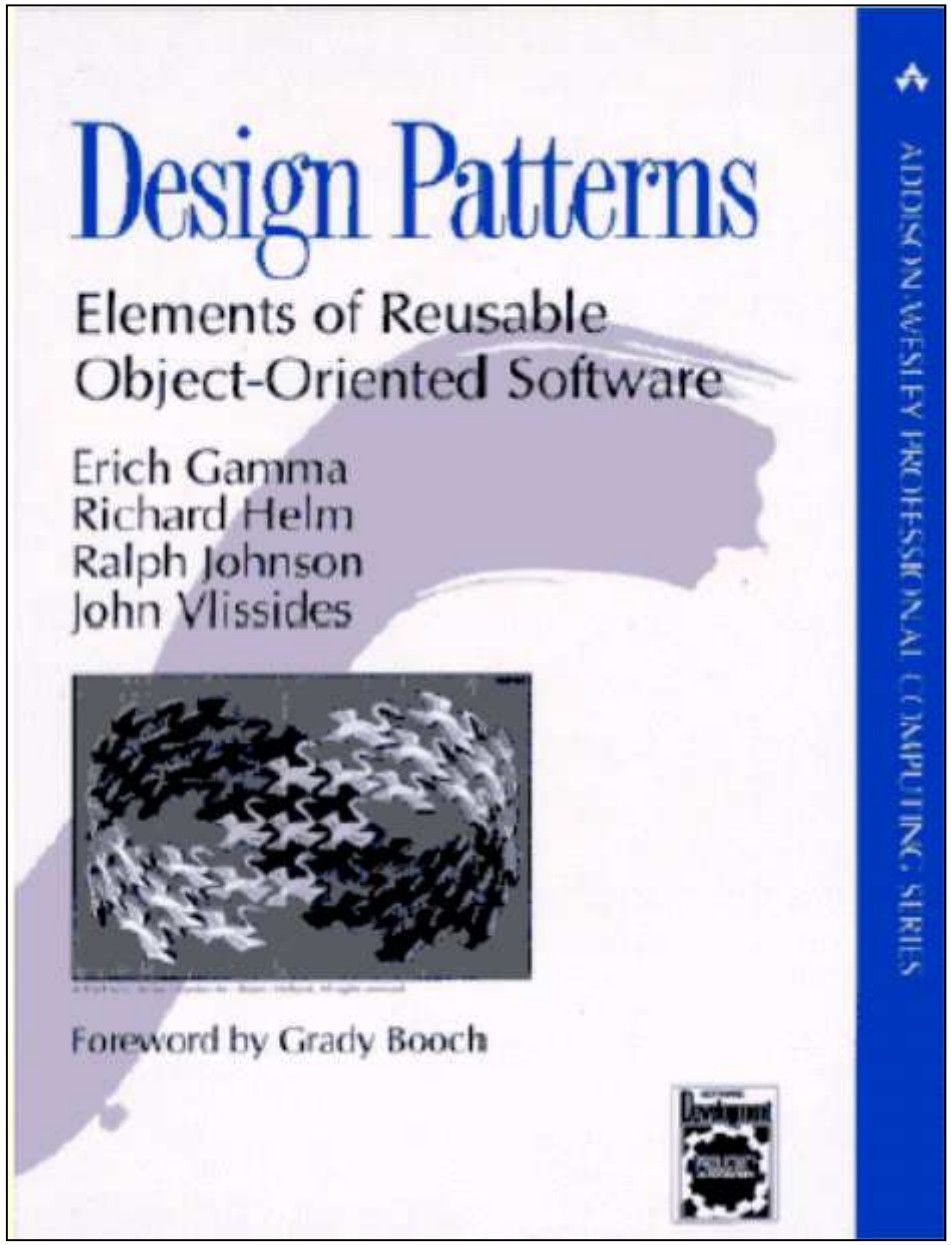
Klassifikation von Patterns

- Architektur Patterns
  - Beschreiben die grundlegende Struktur eines Softwaresystems
  - Client- Schicht, Server- Schicht und Daten- Schicht
- Design Patterns
  - Beschreiben auch Strukturen, allerdings auf einer niedrigeren Ebene als Architektur Patterns
  - Typischer Weise mit Klassen und deren Beziehung zueinander
  - Kann somit zur Ausgestaltung von Teilsystemen führen
- Idioms
  - Behandeln Detailprobleme die z.B. bei der Umsetzung von Design Pattern entstehen können
  - Implementierungsaspekte
  - Programmiersprachenspezifische Probleme (Sockets/Java)

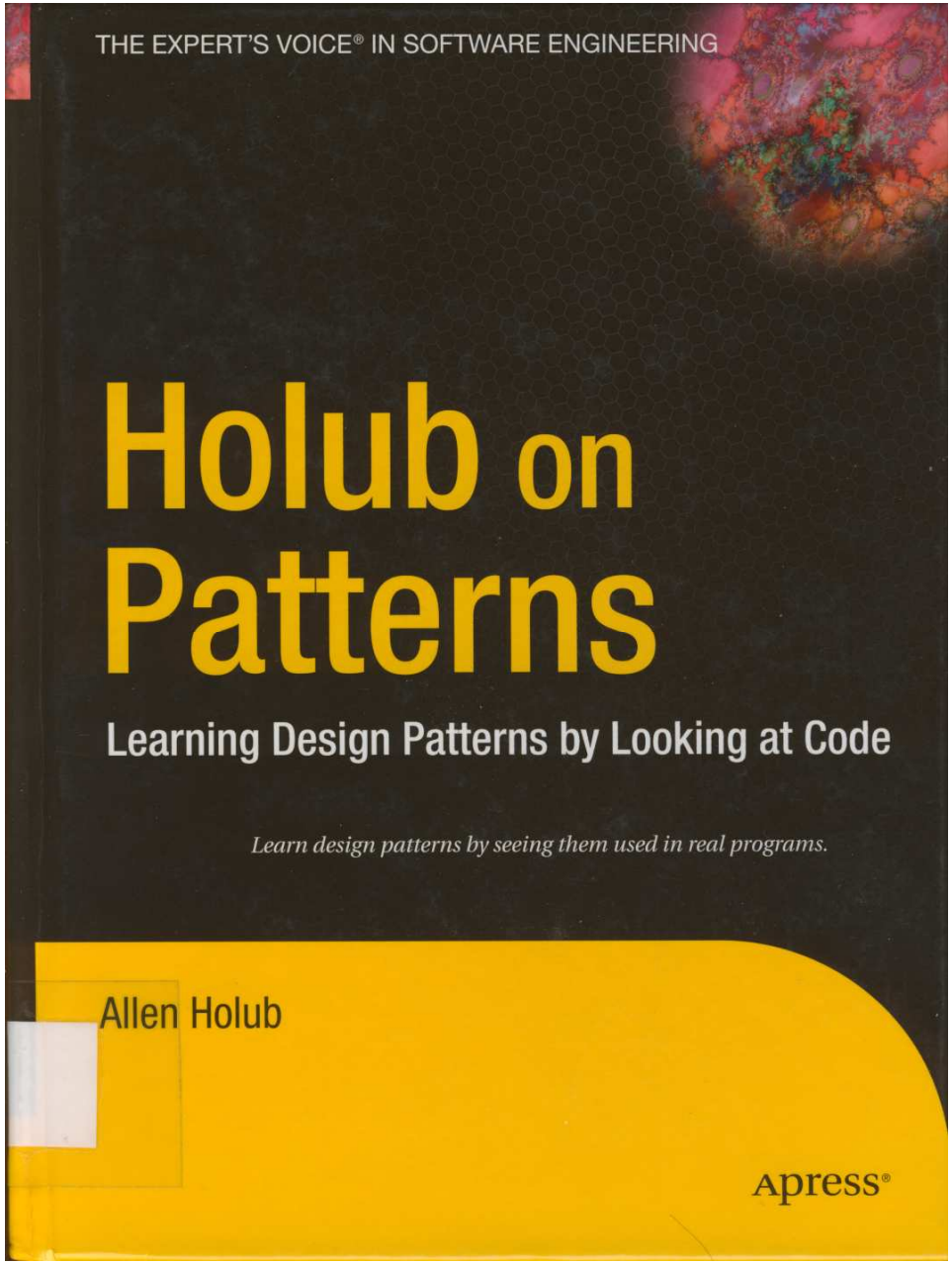
Entwurfsmuster - Konsequenzen

- (+) Robustheit des Entwurfes
- (+) Höherer Wiederverwendungsgrad
- (+) Kommunikation
  - gemeinsames Vokabular
  - höhere Abstraktion
  - leichtere Dokumentation und Verständnis
- (-) höhere „Kosten“

Das Standardwerk



Holub on Patterns



Entwerfen von Systemen:

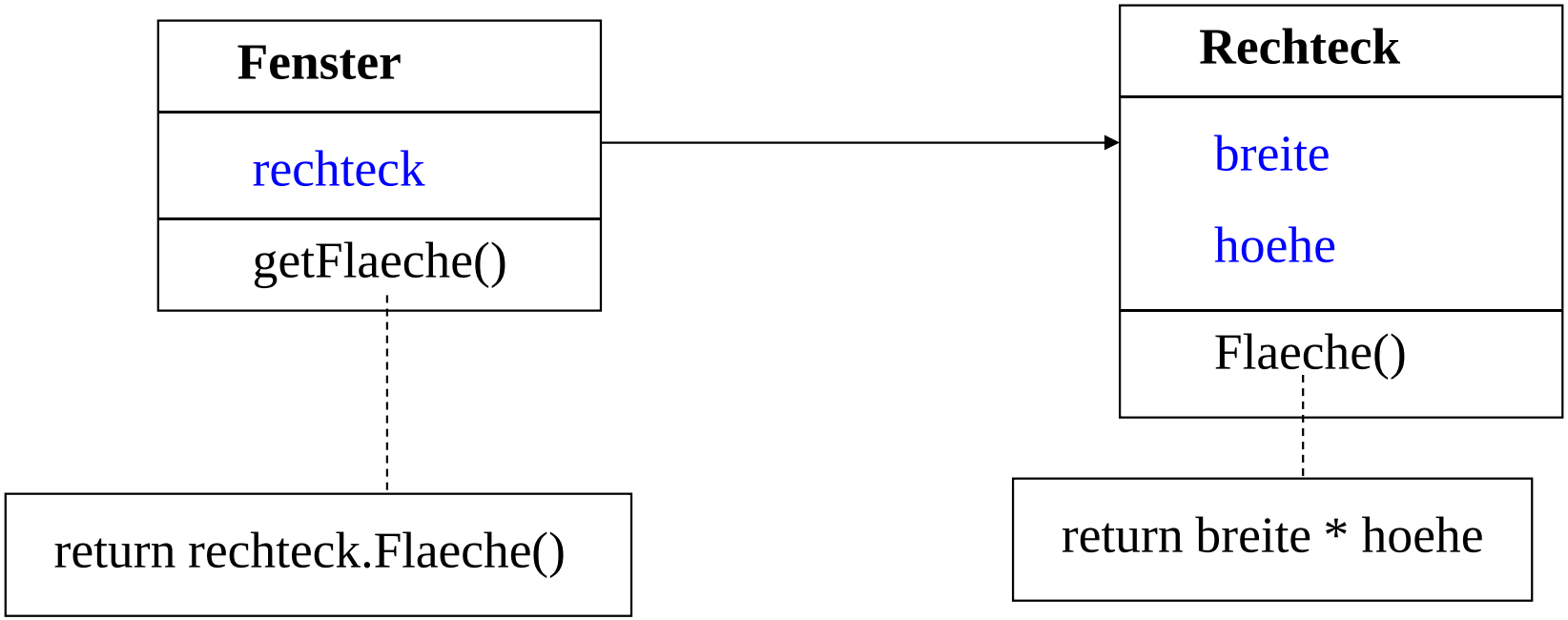
- Klassenvererbung
- Schnittstellenvererbung
- Aggregation bzw.
- Komposition (Spezialfall der Aggregation)
- Generische Klassen (Templates C++, Parametisierbare Typen)
- Klassenbibliotheken
- Frameworks

Verbung vs. Aggregation / Komposition

- Klassenvererbung nennt man auch White-Box-Wiederverwendung.
- Objektkomposition: Neue komplexe Funktionalität wird durch das Zusammenführen oder der Komposition erreicht (**hat vs. ist**)
- Die Komposition erwartet von Objekten, dass sie Schnittstellen der anderen Objekte respektieren.
- Es sind keine internen Details bekannt (Black-Box-Wiederverwendung).
- Klassenhierarchien bleiben klein.
- Es gibt eine größere Anzahl von Objekten. Das Verhalten hängt von ihren Beziehungen untereinander ab.
- Bei der Aggregation können die "Teile" des "Ganzen" auch einzeln existieren, bei der Komposition nur, wenn auch das "Ganze" existiert (z.B. UNummer mit Student).

Komposition

- Bei der Komposition sind zwei Objekte mit der Abarbeitung einer Anfrage beteiligt.
- Ein empfangenes Objekt delegiert Operationen an ein **Kompositionsobjekt** (Ähnlich einer Unterklassen).



Komposition:

- Der Hauptvorteil der Komposition besteht darin, dass es die Zusammensetzung von Verhalten zur Laufzeit vereinfacht.

- Das Fenster kann zur Laufzeit rechteckig oder kreisförmig sein.

- Nachteile:
  - Dynamische, hochgradig parametrische Software ist schwieriger zu verstehen.
  - Laufzeitineffzienten

Entwurfsmuster: Gang of Four

		Aufgabe		
		Erzeugungsm.	**Strukturm.**	**Verhaltensmuster**
Gültigkeits-bereich	**klassenbasiert**	Fabrik	Adapter	Interpreter Schablonenmethode
	objektbasiert	**Abstrakte Fabrik** Erbauer Prototyp **Singleton**	Adapter Brücke **Dekorierer** Fassade Fliegengewicht Kompositum Proxy	Befehl **Beobachter** Besucher Iterator Memento Strategie Vermittler Zustand Zuständigskette

Weitere Entwurfsmuster

- **Architekturmuster**
  - Modell View Controller
- **Design Pattern**
  - Modell View Controller
- **Erzeugungsmuster**
  - Multiton
  - Objektpool
- **Verhaltensmuster**
  - Interceptor
  - Nullobjekt
  - Protokollstapel
- **Andere**
  - Business
  - Delegate
  - Repository
  - Data Access Object
  - Transferobjekt
  - Dependency Injection
  - Extension Interface
  - Fluent Interface
  - Inversion of Control (IoC)
  - Threadpool
  - **Model View Controller (MVC)**
  - **Model View Presenter (MVP)**
  - **Model View ViewModel (MVVM)**

Entwurfsmusterauswahl: **Klassifizierung**

1. **Aufgabe:** (was macht das Muster)

- **Erzeugungsmuster** betreffen den Prozess der Objekterzeugung.
- **Strukturmuster** befassen sich mit der Zusammensetzung von Klassen und Objekten.
- **Verhaltensmuster** charakterisieren die Art und Weise, in der Klassen und Objekte zusammenarbeiten und Zuständigkeiten aufteilen

Entwurfsmuster: **Klassifizierung**

2. Gültigkeitsbereich:

- **Klassenbasierte Muster** befassen sich mit Klassen und ihren Unterklassen. Beziehungen durch Vererbung (statisch).

- **Objektbasierte Muster** befassen sich mit Objektbeziehungen, die zur Laufzeit geändert werden können (dynamisch).

Vorgehen:

- Finden passender Objekte
- Bestimmen von Objektgranularität
- Spezifizieren von Objektschnittstellen
- Spezifizieren von Objektimplementierungen
- Wiederverwendungsmechanismen anwenden
- Strukturen der Laufzeit- und Übersetzungszeit aufeinander beziehen
- Veränderungen in Entwürfen vorhersehen

Behandelten Design Pattern

- Singleton
- Observer
- Modell View Controller
- Decorator
- Fabrik
- Iterator
- Command / Befehlsmuster

1. Beispiel: Singleton

Problem:

- **Sicherstellen, dass nur eine Instanz existiert.**
- **Allgemeiner Zugriff auf diese Instanz muss sichergestellt werden.**

Singleton

```
public class Singleton {
    int value = 0;

    public Singleton(int value){
        this.value = value;
    }

    static public void main(String[] args) {
        Singleton a = new Singleton(1);
        Singleton b = new Singleton(2);
        Singleton c = new Singleton(3);
     }

}
```

Singleton2: nicht Threadsicher

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

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

Singleton3

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

     public static Singleton getInstance(){
        synchronized(Singleton.class) {
            if (instance == null) {
                instance = new Singleton(42);
            }
        }
        return instance;
    }  // getInstance
}
```

Singleton4: Double Checked Locking

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

    public static Singleton getInstance(){
        if (instance == null) {
            synchronized(Singleton.class) {
                if (instance == null) {     // wichtig
                    instance = new Singleton(4711);
                }
            }  // syn
        }  // if
        return instance;
    }
}
```

Erst ab JDK 1,4

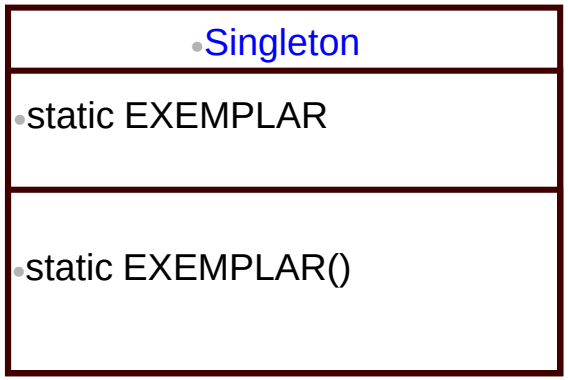
Singleton5

```
public class Singleton {
 private static Singleton instance = new Singleton(2);

   private int i=33;

   protected Singleton(int i) {
     this.i = i;
   }

   public static Singleton getInstance(){
        return instance;
   }

}
```

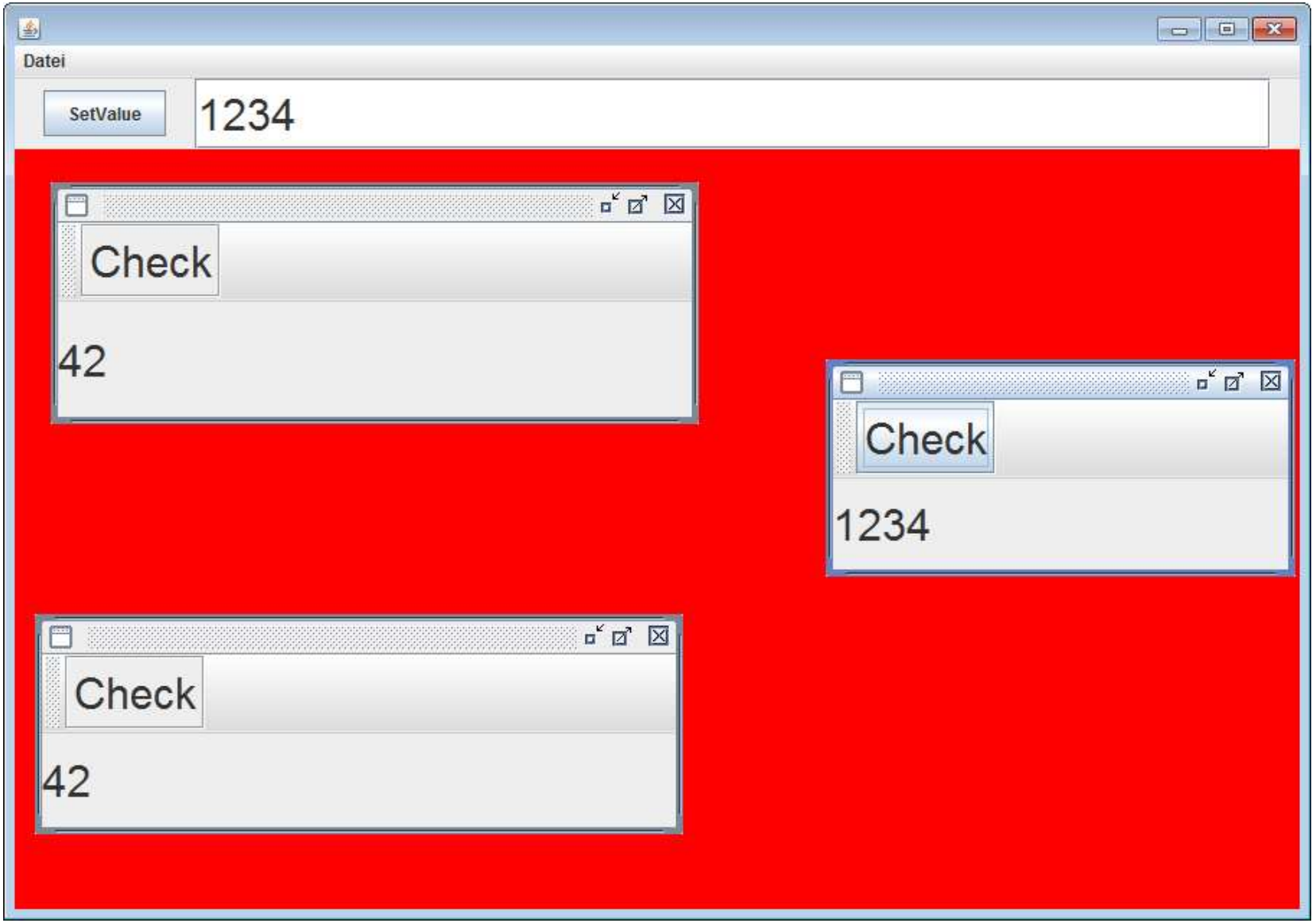
UML: Singleton

Problem:

- **Sicherstellen, dass nur eine Instanz existiert.**
- **Allgemeiner Zugriff auf diese Instanz muss sichergestellt werden.**



Singleton6: Beispiel mit JInternalFrame



Singleton

Zweck:

- Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.

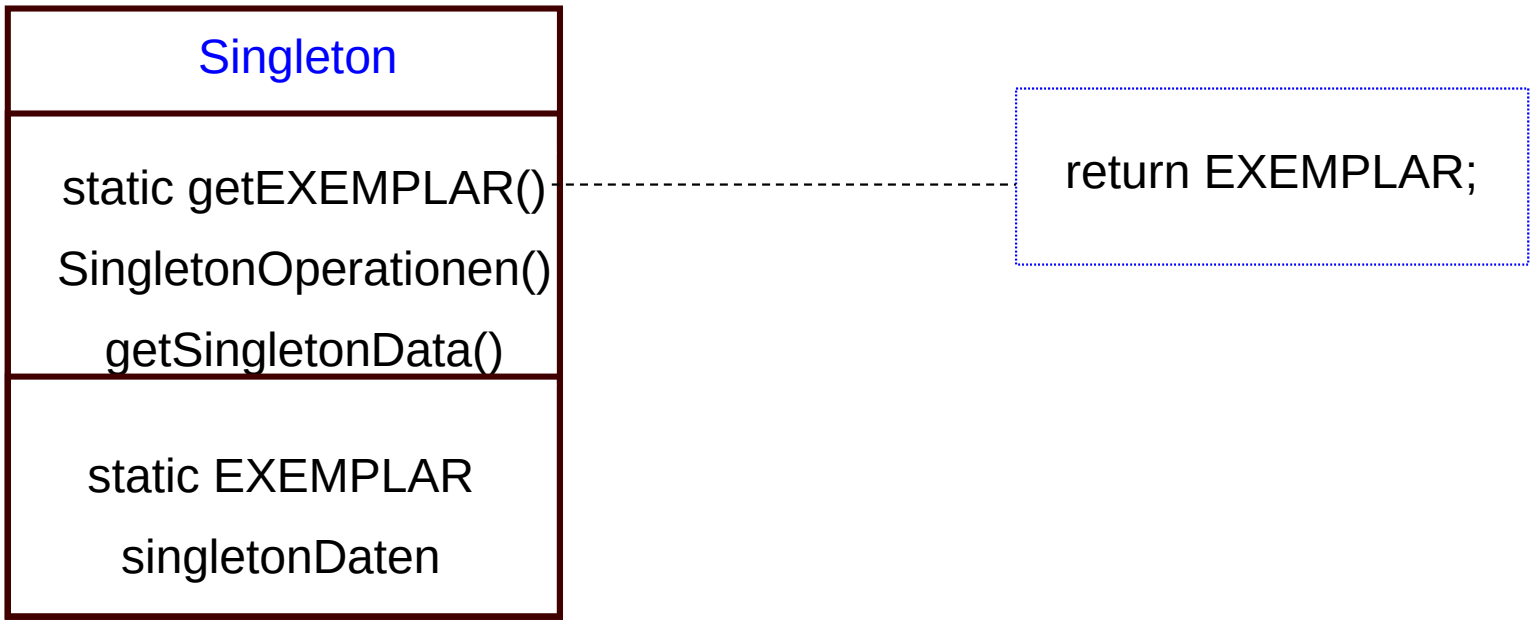
Motivation:

- Eindeutiger Druckerspooler, ein Dateisystem, ein Fenstersystem
- Globale Variable ermöglicht Zugriff auf das Objekt, eindeutig?
- Einzigartigkeit in der Klasse abgefangen

Anwendbarkeit:

- Verwenden Sie das Singletonmuster, wenn
- es genau ein Exemplar einer Klasse geben muss und der Zugriff eindeutig sein soll.
- das einzige Exemplar durch Bildung von Unterklassen erweiterbar sein soll. Anwender sollen die Erweiterungen ohne Modifikation weiter verwenden.

Struktur des Singleton-Pattern



Singleton

Teilnehmer:

- definiert eine Exemplaroperation, die es Klienten ermöglicht, auf sein einziges Exemplar zuzugreifen.
- ist für die Erzeugung seines einzigen Exemplars zuständig.

Interaktion:

- Klienten greifen ausschließlich durch die getExemplar-Operation der Singletonklasse zu.

Konsequenzen / Vorteile:

- *Zugriffskontrolle auf das Exemplar.* Aus der Kapselung des Konstruktors folgt die strikte Kontrolle.
- *Kleinerer Namensraum.* **Das Singletonmuster ist eine Verbesserung gegenüber globalen Variablen**

Singleton

Konsequenzen / Vorteile:

- *Verfeinerung von Operationen und Repräsentation*. Die Klasse kann abgeleitet und spezialisiert werden. Die Konfiguration zur Laufzeit ist möglich.
- *Variable Anzahl von Exemplaren*. Änderung der Methode getExemplar() ist einem Konstruktor stellt den Normalzustand ohne weitere Codeänderung her.

Implementierung:

Problem:

- Ableitung durch mehrere Klassen.
- Wer bekommt welche Instanz?

Singleton mit enums

```
enum Singleton {
    INSTANCE;
  public int i=33;
  private Singleton() {
    this(0);
  }
  private Singleton(int i) {
    this.i = i;
  }
}  // Singleton_enum
```

- Der Vorteil dieser Variante ist, dass sie selbst aufwändigen Manipulationsversuchen durch Serialisierung und/oder Introspektion um eine Mehrfachinstanziierung zu ermöglichen widersteht.
- Es ist die derzeit beste Methode, Singletons zu implementieren.
- Der Nachteil ist jedoch, dass diese Methode erst ab Java Version 1.5 funktioniert.

Singleton mit enums

```
static public void main(String[] args) {
    System.out.println(" ");

    Singleton  a  = Singleton.INSTANCE;
    System.out.println("a: "+ a.getValue()  );
    a.setValue(2222);
    System.out.println(" ");

    Singleton  b = Singleton.INSTANCE;
    b.setValue(1111);
    System.out.println("a: "+ a.getValue()  );
    System.out.println("b: "+ b.getValue()  );
}
```

Singleton mit Vererbung

```
class Ball {
  private static Ball instance = new Ball(5);
  public int r=10;
    protected Ball(int r) {
      this.r = r;
    }
    public static Ball getInstance(){
        return instance;
    }
}
class Fussball extends Ball {
  private static Fussball instance = new Fussball(15, true);
  public boolean leder;
    protected Fussball(int r, boolean leder) {
      super(r);
      this.leder = leder;
    }
    public static Fussball getInstance(){
        return instance;
    }
} // Fussball
```

Singleton mit Vererbung

```
class WMFussball extends Fussball {
  private static WMFussball instance = new WMFussball(20, true, "Oddibas");
  private String hersteller;
    protected WMFussball(int r, boolean leder, String hersteller) {
      super(r, leder);
      this.hersteller = hersteller;
    }
    public static WMFussball getInstance(){
        return instance;
    }
} // WMFussball
```

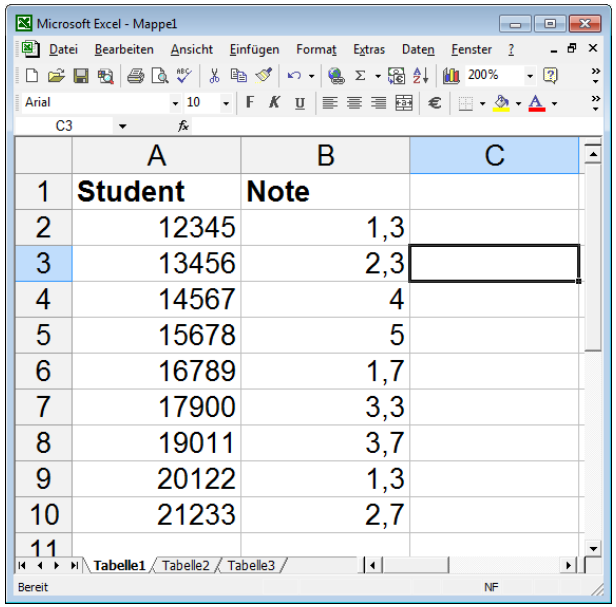
Singleton mit Vererbung

```
Ball b1  = Ball.getInstance();
   System.out.println("b1: "+ b1 +"\n\n" );
   b1.setRadius(12);

   Ball b2  = Ball.getInstance();
   System.out.println("b1: "+ b1  );
   System.out.println("b2: "+ b2  );

   Fussball f1  = Fussball.getInstance();
   System.out.println("\n\nf1: "+ f1 );
   WMFussball wf1  = WMFussball.getInstance();

   System.out.println("wf1: "+ wf1+"\n\n\n" );
   Ball b3 = (Ball) wf1;
   b3.setRadius(42);
   System.out.println("b1: "+ b1 );
   System.out.println("b2: "+ b2 );
   System.out.println("b3: "+ b3 );
```

```
b1: Ball, radius: 5

b1: Ball, radius: 12
b2: Ball, radius: 12

f1: Fussball: Ball, radius: 15    leder: true
wf1: WMFussball: Fussball: Ball, radius: 20
                leder: true    herst.: Oddibas


b1: Ball, radius: 12
b2: Ball, radius: 12
b3: WMFussball: Fussball: Ball, radius: 42
                leder: true    herst.: Oddibas
```

MDI-Fenster:

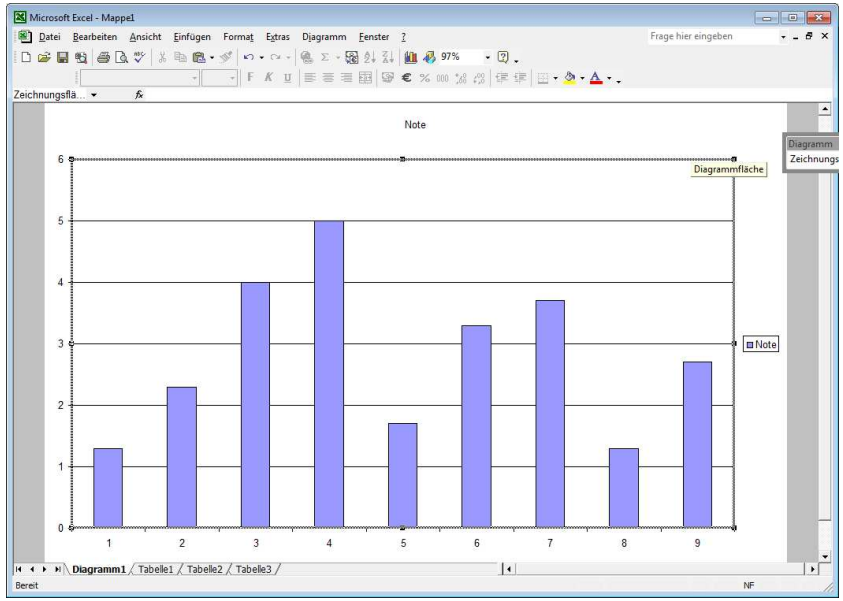
Daten in einer Tabelle

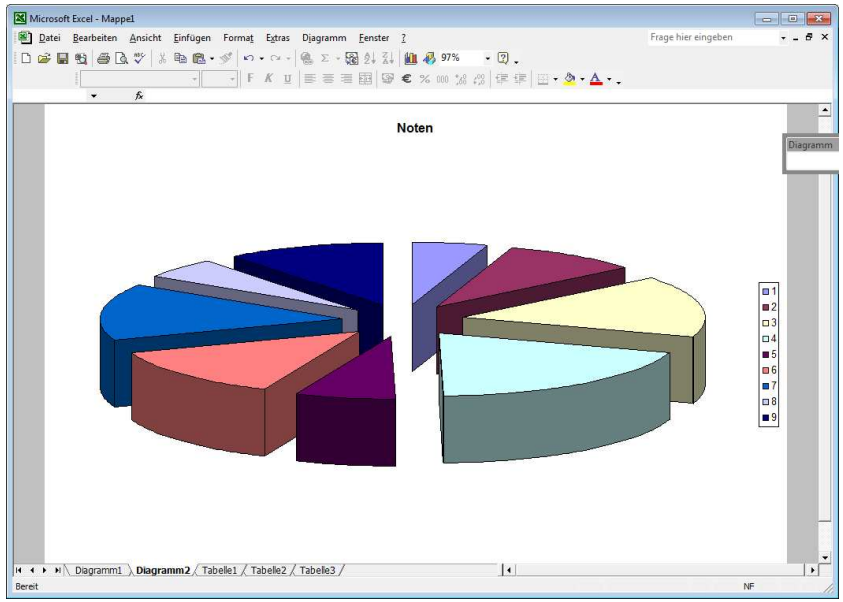


	A	B	C
1	**Student**	**Note**	
2	12345	1,3	
3	13456	2,3	
4	14567	4	
5	15678	5	
6	16789	1,7	
7	17900	3,3	
8	19011	3,7	
9	20122	1,3	
10	21233	2,7	
11			

Problem:
Datenaktualisierung

Daten in einer Grafik





Beispiel: MVC / Observer Pattern

Problem:

Mehrere Objekte einer Gruppe müssen sich benachrichtigen, wenn einige Attribute geändert wurden. Die Viewer sind nicht „bekannt“.

- Smalltalk: Model-View-Controller
- MFC Dokument-View-Architektur
- Java: Model-View-Architektur

Observer

- **Model**
  - Das Modell (engl. model) im Observer-Muster enthält die der Darstellung im View zugrunde liegenden Daten.
  - Die Views registrieren sich beim Modell.
  - Änderungen an den Daten werden dem Model mitgeteilt.
    - setter /getter
  - Der View muss sich ändern, sobald sich die Daten im Modell verändern. Daher ist das Modell beobachtbar.

- **View**
  - Der View hat die Aufgabe die Daten des Models auf irgend eine Art darzustellen. Dabei hat der View nur die Aufgabe, die Daten darzustellen (Visualisierung, Darstellung)

MVC/ Observer

- **Anwendbarkeit**
  - Eine Abstraktion hat zwei Aspekte, einer abhängig vom Anderen.
  - Aufnahme jedes Aspekts in separate Objekte erlaubt jedes unabhängig zu benutzen.
  - Die Änderung eines Objekts zieht Veränderungen in anderen Objekten nach sich, es ist aber nicht klar wie viele Objekte verändert werden müssen.
  - Ein Objekt soll andere Objekte benachrichtigen, ohne nähere Kenntnis dieser Objekte zu haben.

MVC/ Observer

- **Teilnehmer**
  - Subjekt (beobachtbare Instanz, Observable):
    - kennt seine Beobachter. Eine beliebige Anzahl von Beobachtern ist möglich.
      Stellt Schnittstelle zum Anbinden und Lösen von Beobachterobjekten zur Verfügung.
  - Beobachter (Observer):
    - Definiert eine Schnittstelle zum Auffrischen für Objekte

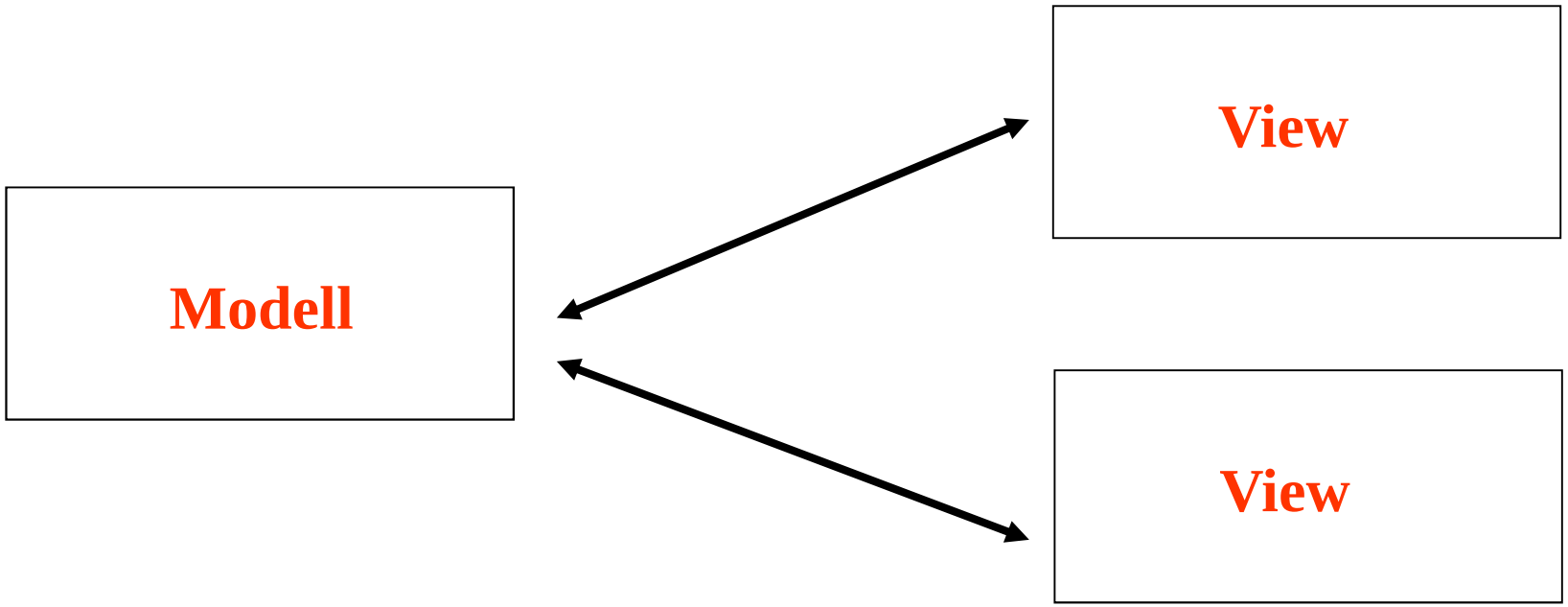
MVC/ Observer

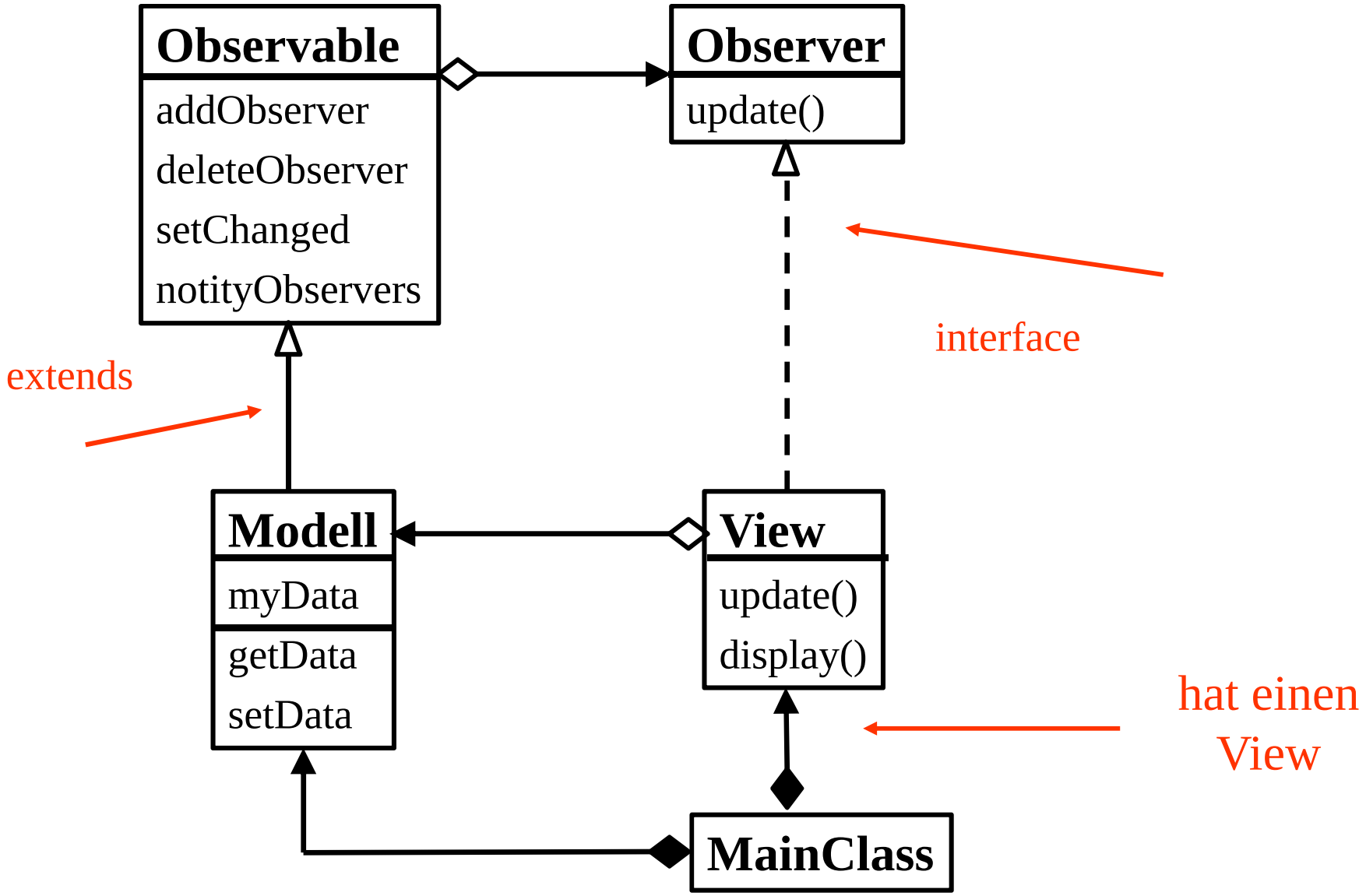
- **Konsequenzen**
  - Das Beobachtermuster erlaubt Subjekte und Beobachter unabhängig zu variieren. Jedes kann unabhängig vom anderen wiederbenutzt werden. Es können neue Beobachter hinzugefügt werden, ohne das Subjekt oder die anderen Beobachter zu verändern.
- **Vorteil**
  - Die Subjekte unterstützen **broadcast** Kommunikation. Es gibt keinen spezifischen Empfänger. Die Empfänger sind mit dem Subjekt registriert und erhalten automatisch update Nachrichten.
- **Nachteil**
  - Beobachter wissen nichts voneinander und damit auch nichts über die **Kosten** einer Datenauffrischung. Eine einfache Änderung kann eine Kaskade von Änderungen nach sich ziehen. Es wird auch nicht mitgeteilt was verändert wurde - also muss immer der Gesamtzustand an die Beobachter übermittelt werden.

Realisierung MVC / Observer

- **Implementierung**
  - Abbildung der Subjekte auf Beobachter: Subjekt hält Referenz auf alle Beobachter. Für viele Subjekte mit wenigen Beobachtern benötigt dies viel Speicher. Es ist möglich, Speicher gegen Zugriffszeit zu tauschen, indem man, z.B. eine Hash-Tabelle benutzt (vgl. Programmiervorlesung 1).
  - Ein Beobachter kann mehr als ein Subjekt verfolgen. In diesem Fall sollte eine Referenz auf das Subjekt mit an die update Methode übergeben werden.

Beispiel: Observer Pattern (Modell und Views)

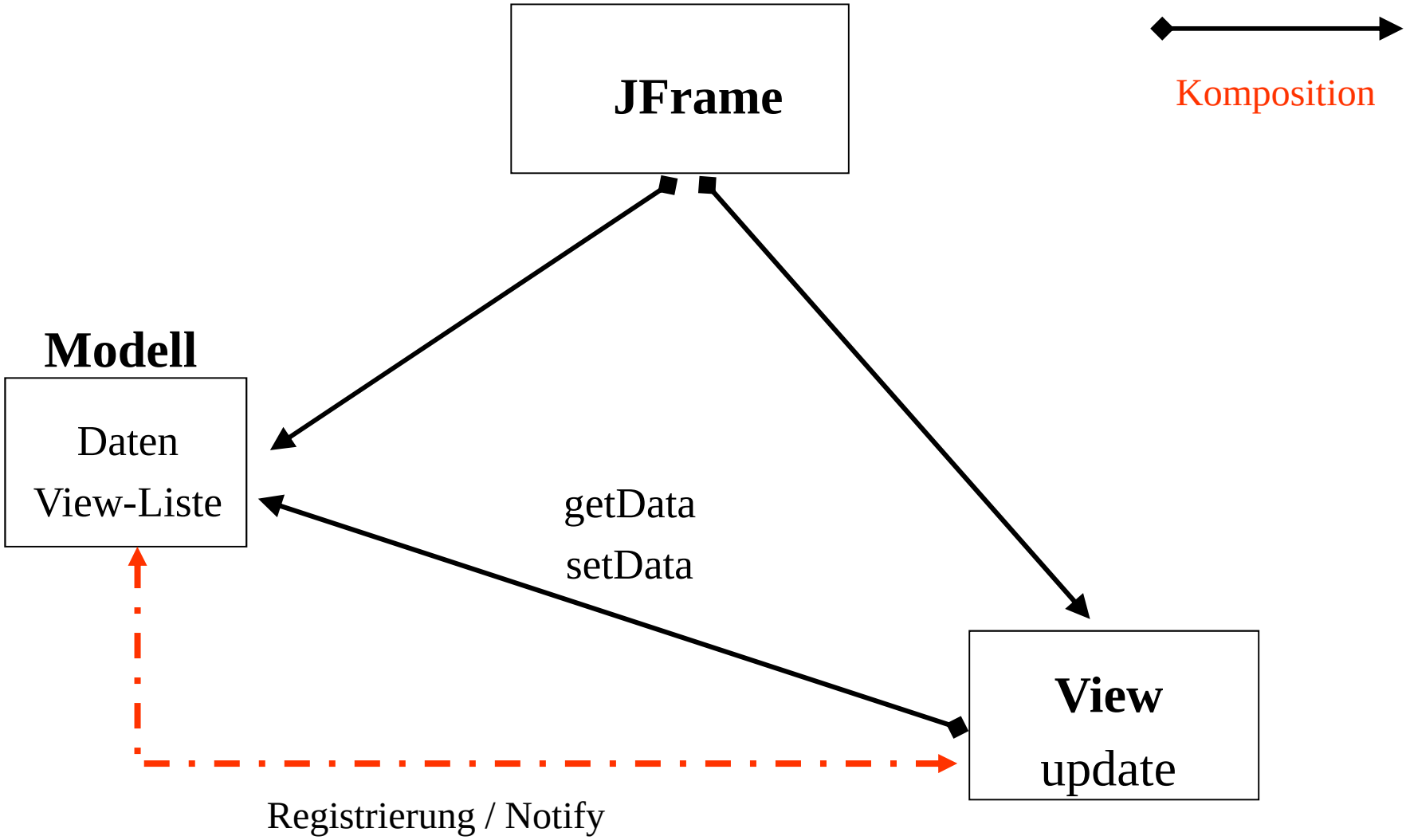



Observable
addObserver
deleteObserver
setChanged
notityObservers
Observer
update()
interface
extends
Modell
myData
getData
setData
View
update()
display()
hat einen
View
MainClass
interface
Komposition
Aggregation

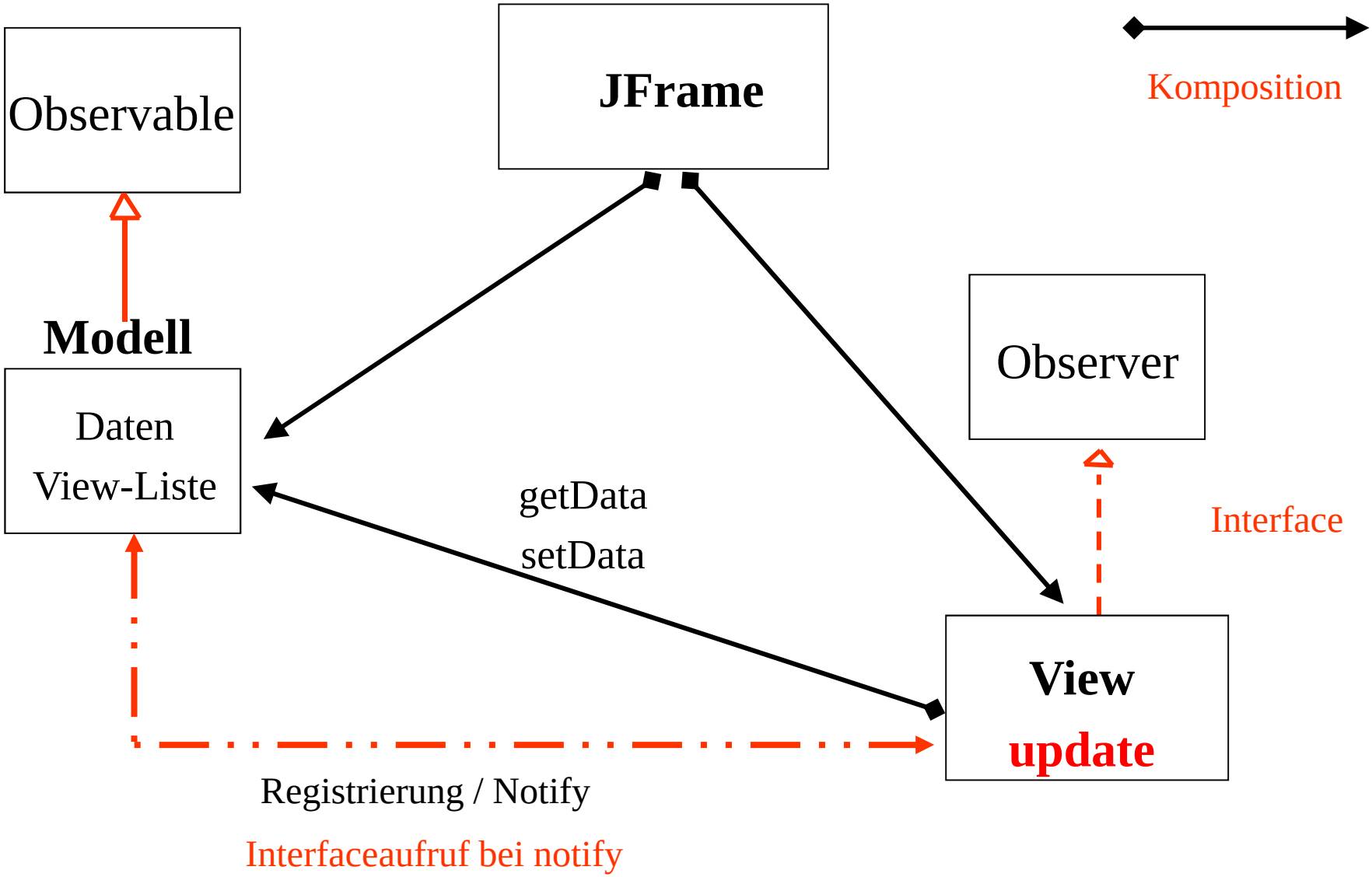
Konkret

- **Observable Methoden**
  - addObserver um einen Beobachter hinzuzufügen
  - deleteObserver entfernt einen spezifizierten Beobachter
  - notifyObservers sendet Nachrichten an alle Beobachter
  - setChanged registriert eine Änderung
  - hasChanged fragt, ob Änderungen vorhanden sind
  - vgl. Java Dokumentation

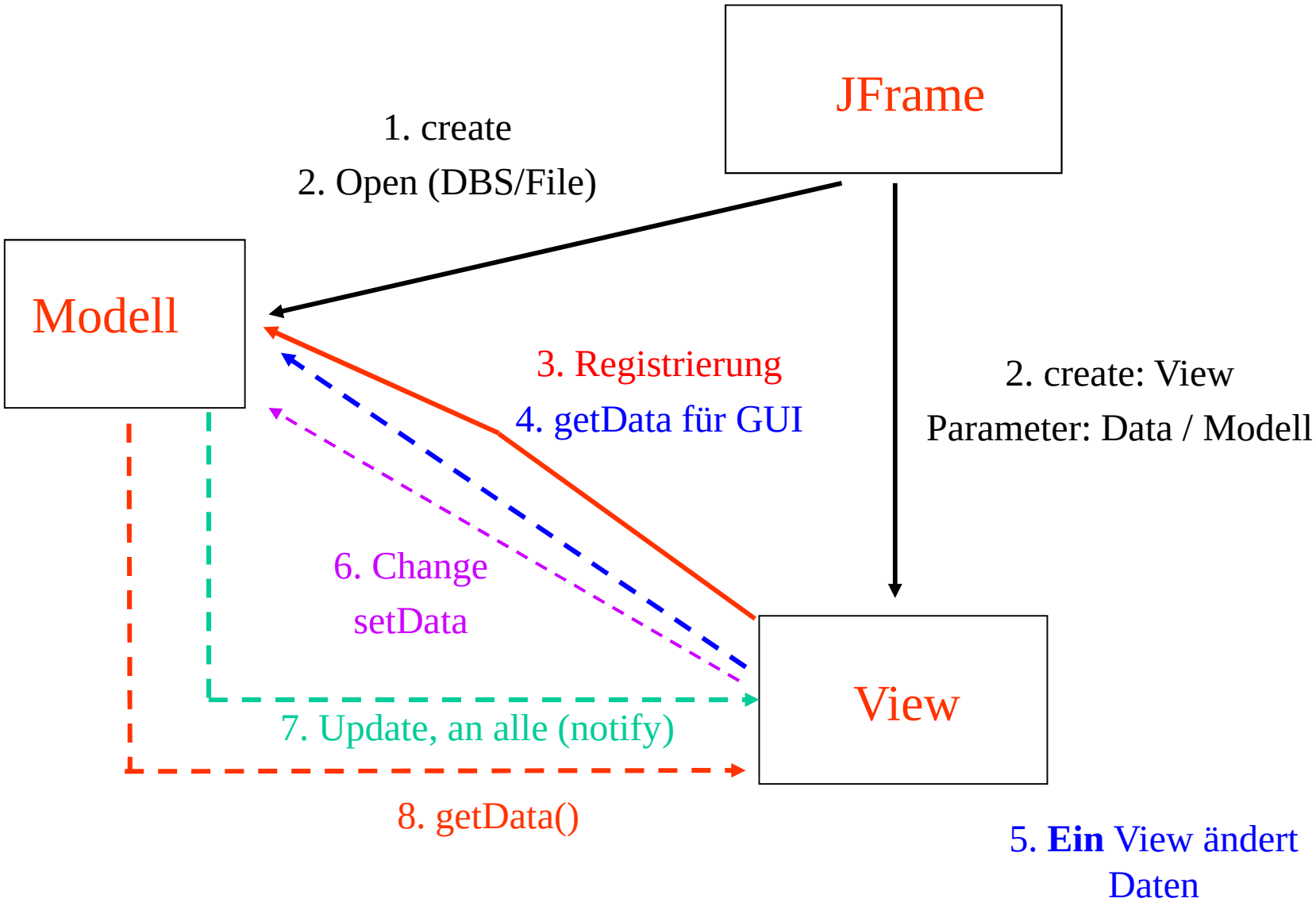
Observer Pattern



Observer Pattern



Observer Pattern: Reihenfolge



Bemerkungen zum MVC/Observer Pattern

- Entkopplung
  - Subjekt kennt nur Observer-Interface, keine konkreten Observer
  - update = dynamisch gebunden => upcalls
- Falls Observer von mehreren Subjekten abhängen sollen:
  - update(Observable o)
- Falls Art der Änderung übergeben werden soll: push vs pull
  - notifyObservers(Object arg)
  - update(Observable o, Object arg)
- Wann wird notify aufgerufen?
  - Zustandsänderungsfunktion => i.a. viele updates
  - Verzögert & explizit: changed-Flag: setChanged / clearChanged
    - notifyObservers wird nur aktiv wenn isChanged() == true
    - notifyObservers ruft clearChanged auf

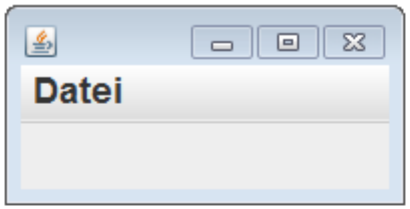
Bemerkungen zum MVC/Observer Pattern

- Kausalität der Änderungen
  - Verbieten von Änderungen während update
  - Abhilfe: Queue zum Speichern der Änderungen

- Multithreading
  - Während der Meldung „notify“ kann add/remove des Observers aufgerufen werden. Die Listenerliste wird geändert (InvalidStateException bei Iterator)
    - synchronized(this){
      - for(int i=0; i<observers.size(); i++){…..
    - Vector v;
      - synchronized(this){v = (Vector)observers.clone();}
      - for(int i=0; i<v.size(); i++){…

- Interfaces vs. Classes

1. Java-Beispiel: Observer1.java



1. Java-Beispiel: Observer1.java

```
// 1. Import
import java.util.Observable;
import java.util.Observer;

public class Observer1 extends JFrame {
  int anz=0;

   // 4. Schritt  die Modell Variable im JFrame deklarieren
  private Modell modell;   // Modell

public Observer1() {
// 5. Schritt  die Modell Variable erzeugen
    modell = new Modell();
    modell.setName("Paul");
  }
```

1. Java-Beispiel: Observer1.java

```
void MnNew_click() {
  // 7. Schritt  Viewer in der Methode "New_click" erzeugen

 JObserverFrame frame = new JObserverFrame(
        Integer.toString(anz), anz%3, modell);

  frame.setLocation(  (anz%2)*390+10, 180+110*(anz/2)  );
  frame.setSize(300,80);
  anz++;
 }
} // Observer1
```

1. Java-Beispiel: Observer1.java

```
// 2. Schritt ableiten von Observable, damit Modell und Observable
class Modell  extends java.util.Observable {
  private String name= "Zappa";

    // Diese Methode wird von einem Viewern aufgerufen
// 3. Schritt: Im Controller eine Methode für die Viewer zum Updaten schreiben
  public void DataChangedFromViewer(int nr, String s) {
    System.out.println("s im Modell: "+s);
    this.setName(s);  // geänderte Daten speichern
    super.setChanged(); // wichtig
      // Sende Nachricht an ALLE Viewer
    super.notifyObservers( new Integer(1) );
  }

} // Modell
```

1. Java-Beispiel: Observer1.java

```
// 6. Schritt  im JFrame  Implements einbauen
class JObserverFrame extends JFrame implements Observer {
  private JTextField edit = new JTextField("XXXXX");

  // 7. Schritt  Modell deklarieren
  private Modell modell;
  private String Caption;
  int nr;
  // 8. Schritt  Konstruktor
  public JObserverFrame( String Caption, int nr, Observable DatenModell) {
    this.Caption = Caption;
    setTitle(Caption);
    this.nr=nr;
    …
  }
```

1. Java-Beispiel: Observer1.java

```
public JObserverFrame( String caption, int nr, Observable datenModell) {
  …
   // 9. Schritt  globale Variable setzen
   modell = (Modell) datenModell;  //

   // 10. Schritt  im JFrame  registrierung
   modell.addObserver(this);
   setGUI();
   setVisible(true);
 }
 public void setGUI() {
   // 11. Schritt  aus dem Modell die Daten holen
   edit.setText(  modell.getName()     );
 }  // setGUI
```

1. Java-Beispiel: Observer1.java

```
// 12. Schritt  Schnittstellen implementieren: Methode Update
// aufgerufen vom Modell
public void update(Observable o, Object arg) {
   Integer I = (Integer) arg;
   // 13. Schritt  neue Daten holen
    edit.setText( modell.getName() );
    edit.updateUI();
//  }
}
// 14. Schritt  Methode edit_change (Enter) impl., Aufruf des Modell/notify
void edit_Change(ActionEvent e) {
    modell.DataChangedFromViewer( nr,edit.getText());
}
}  // JObserverFrame
```

2. Java-Beispiel: Observer2.java onChange-Event

```
class JObserverFrame extends JFrame implements Observer, ActionListener{

JTextFieldDocListener edit = new JTextFieldDocListener(this,"XXXXX");


public void actionPerformed(ActionEvent e){
    if (e.getSource()==edit) {
      modell.DataChangedFromViewer( tag, edit.getText() );
      modell.setUpdateModus(false);
     }
}
```

2. Java-Beispiel: Observer2.java

```
class JObserverFrame extends JFrame implements Observer, ActionListener{

JTextFieldDocListener edit = new JTextFieldDocListener(this,"XXXXX");


public void actionPerformed(ActionEvent e){
   if ( modell.getUpdateModus() ) {
     System.out.println(" actionPerformedaction:  modell.getUpdateModus() ist an");
   }
   else {
     if (e.getSource()==edit) {
       modell.DataChangedFromViewer( tag, edit.getText() );
       modell.setUpdateModus(false);
     }
   }
}
```

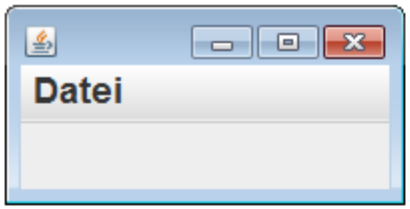
2. Java-Beispiel: Observer2.java

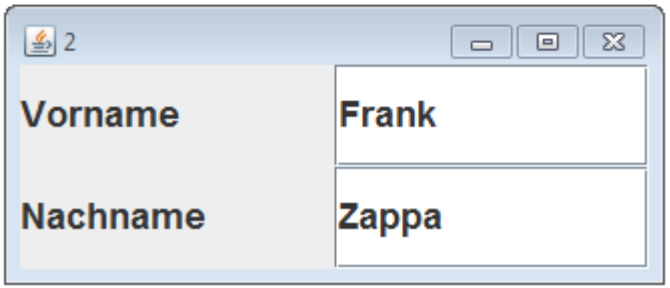
```
class JTextFieldDocListener extends JTextField  {
 ArrayList<ActionListener> listeners = new ArrayList<ActionListener>();

 public synchronized void addEventListener(ActionListener listener)  {
  listeners.add(listener);
 }
 public synchronized void removeEventListener(ActionListener listener)  {
  listeners.remove(listener);
 }
  JTextFieldDocListener thiss;
  public JTextFieldDocListener(ActionListener listener) {
    super();
    init(listener);
  }
  public JTextFieldDocListener(ActionListener listener, String sText) {
    super(sText);
    init(listener);
  }
```

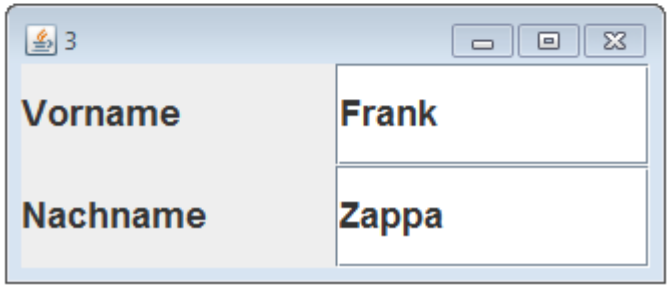
2. Java-Beispiel: Observer2.java

```
private void init(ActionListener listener) {
    addEventListener(listener);
    thiss = this;
    this.getDocument().addDocumentListener(new DocumentListener(){
      public void changedUpdate(DocumentEvent ev) {
        for (ActionListener listener : listeners) {
          ActionEvent e = new ActionEvent( thiss,0,"update",0,0);
          listener.actionPerformed(e);
        }
      }
      public void insertUpdate(DocumentEvent ev) {
        ActionEvent e = new ActionEvent(thiss,0,"insert",0,0);
        for (ActionListener listener : listeners) {
          listener.actionPerformed(e);
        }
      }
      public void removeUpdate(DocumentEvent ev) {
        ActionEvent e = new ActionEvent(thiss,0,"remove",0,0);
        for (ActionListener listener : listeners) {
          listener.actionPerformed(e);
        }
      }
```

3. Java-Beispiel: Observer3.java
Zwei Editorzeilen







3. Java-Beispiel: Observer3.java

```
private JTextFieldDocListener vorname = new JTextFieldDocListener(this,"XXXXX");
private JTextFieldDocListener nachname = new JTextFieldDocListener(this,"XXXXX");

public void actionPerformed(ActionEvent e){
   if ( modell.getUpdateModus() ) {
     System.out.println(" actionPerformedaction:  modell.getUpdateModus() ist an");
   }
   else {
    if (e.getSource()==vorname) {
       modell.setVorname(vorname.getText() );
     }
     if (e.getSource()==nachname) {
       modell.setNachname(nachname.getText() );
     }
     modell.DataChangedFromViewer( tag );
     modell.setUpdateModus(false);
   }
 }
```

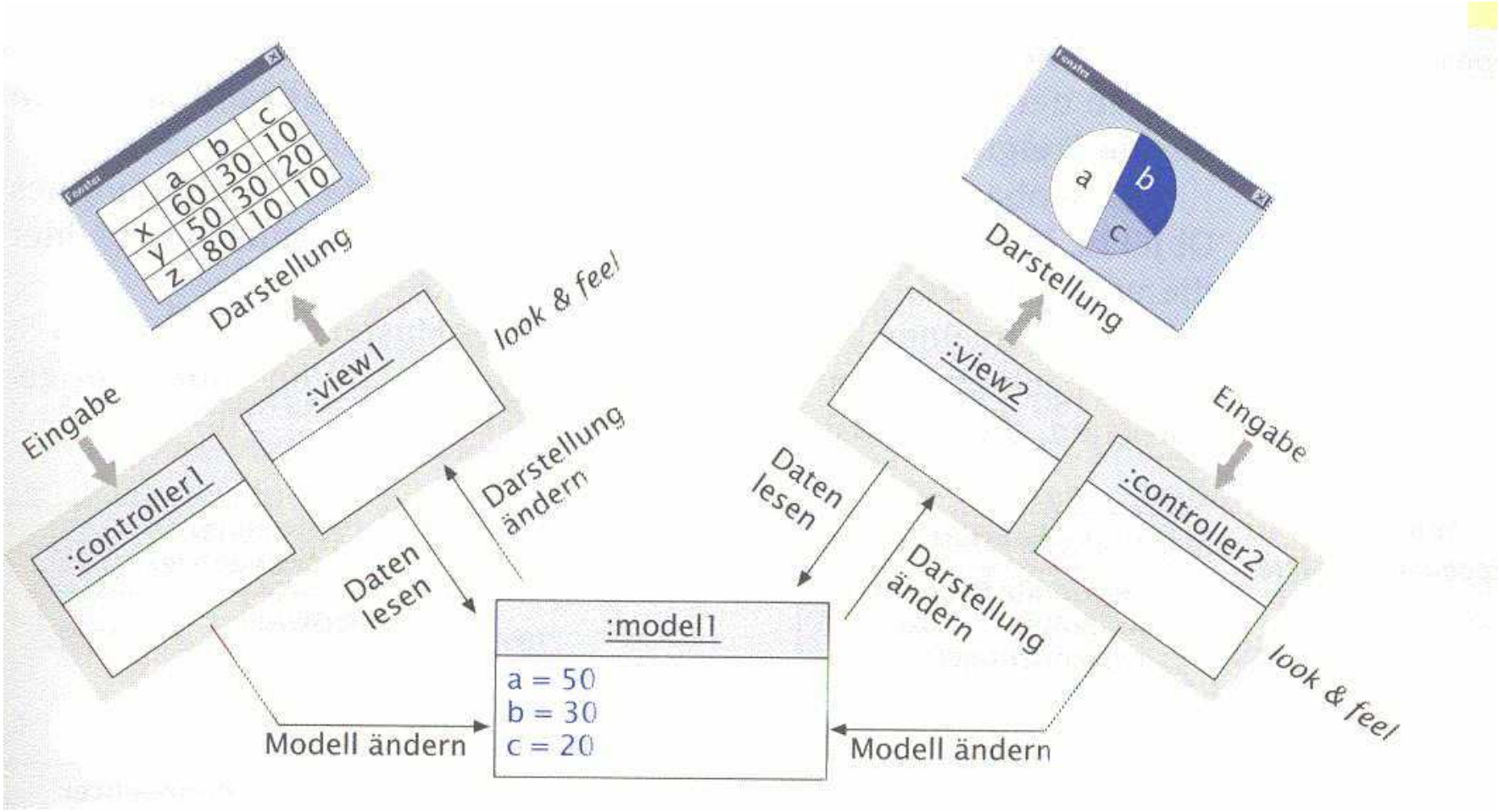
Erweiterung: Modell-View-Controller

- Das Model kann man sich als das Datenmodell (Datei, Datenbank) vorstellen, der den aktuellen Zustand und das Verhalten des gesamten Systems repräsentiert (Datenhaltung, Anwendung)

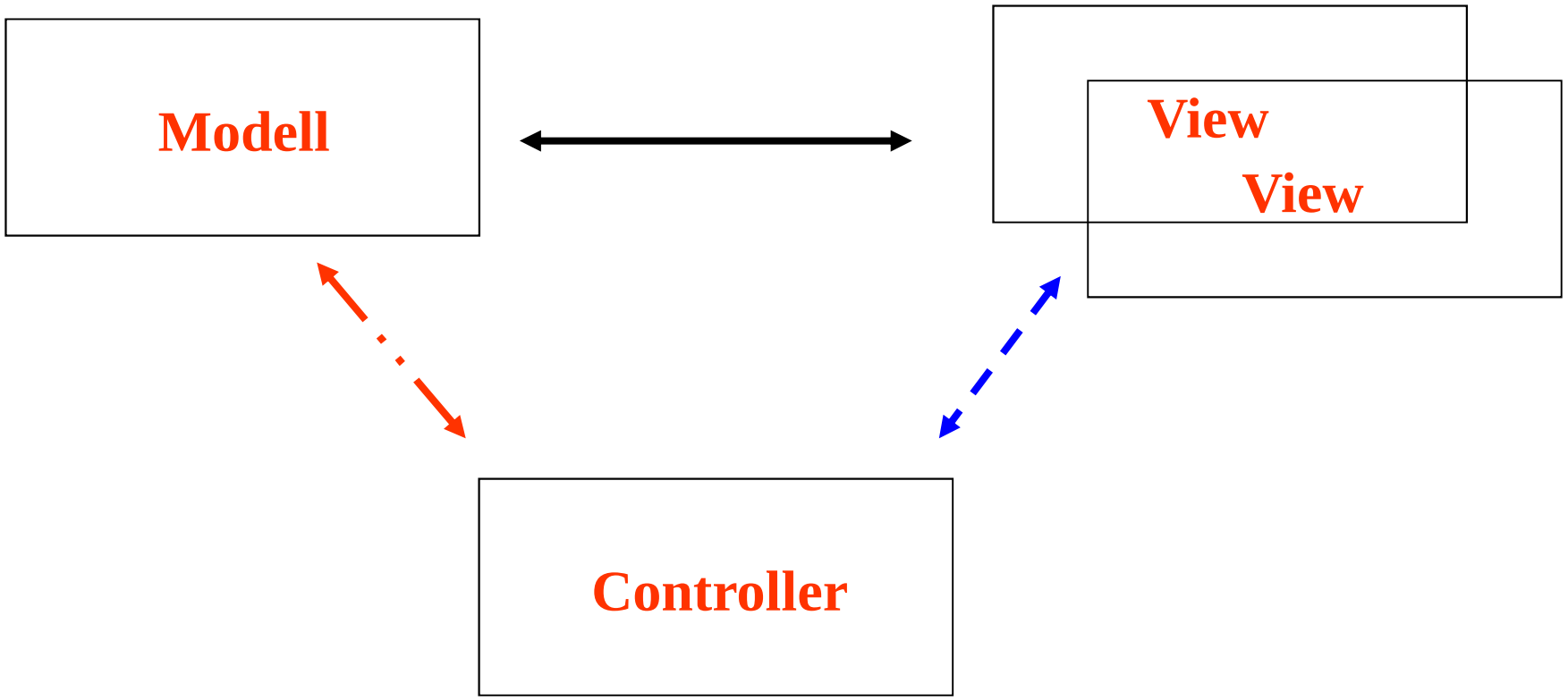
- Der View hat die Aufgabe die Daten des Models auf irgend eine Art darzustellen. Dabei hat der View nur die Aufgabe, die Daten darzustellen (Visualisierung, Darstellung)

- Der Controller setzt die eingehenden Anforderungen (z.B. Eingaben von der Tastatur oder Maus) in Methoden um, die das Model dazu veranlassen, die Daten entsprechen zu verändern (Datenmanipulation, Steuerung). Der Controller hat spezielle Aufgaben (z.B. Log-Datei etc.)

MVC Entwurfsmuster: Look & Feel



Beispiel: MVC Pattern



Mozilla Firefox

Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe

Aktuelle Nachrichten - Bild.de | uml mvc darstellung - Google-Suche | http://www.famil...c/mvc_03uml.htm | [MVC] Darstellung im UML - php.de | MVC.pdf (application/pdf-Objekt)

www.familiesteudle.de/tutorium_MVC_Muster/patterns/mvc/mvc_03uml.htm

Unwesentliche Elemente dürfen *zunächst* überlesen werden. Müssen aber dennoch verstanden werden.

Wichtige Elemente:

- Startpunkt des Programms (Hauptmethode)
- Businessmethoden des Models
- Ereignisbehandelnde Methoden des Controllers
- Anzeigemethoden des View

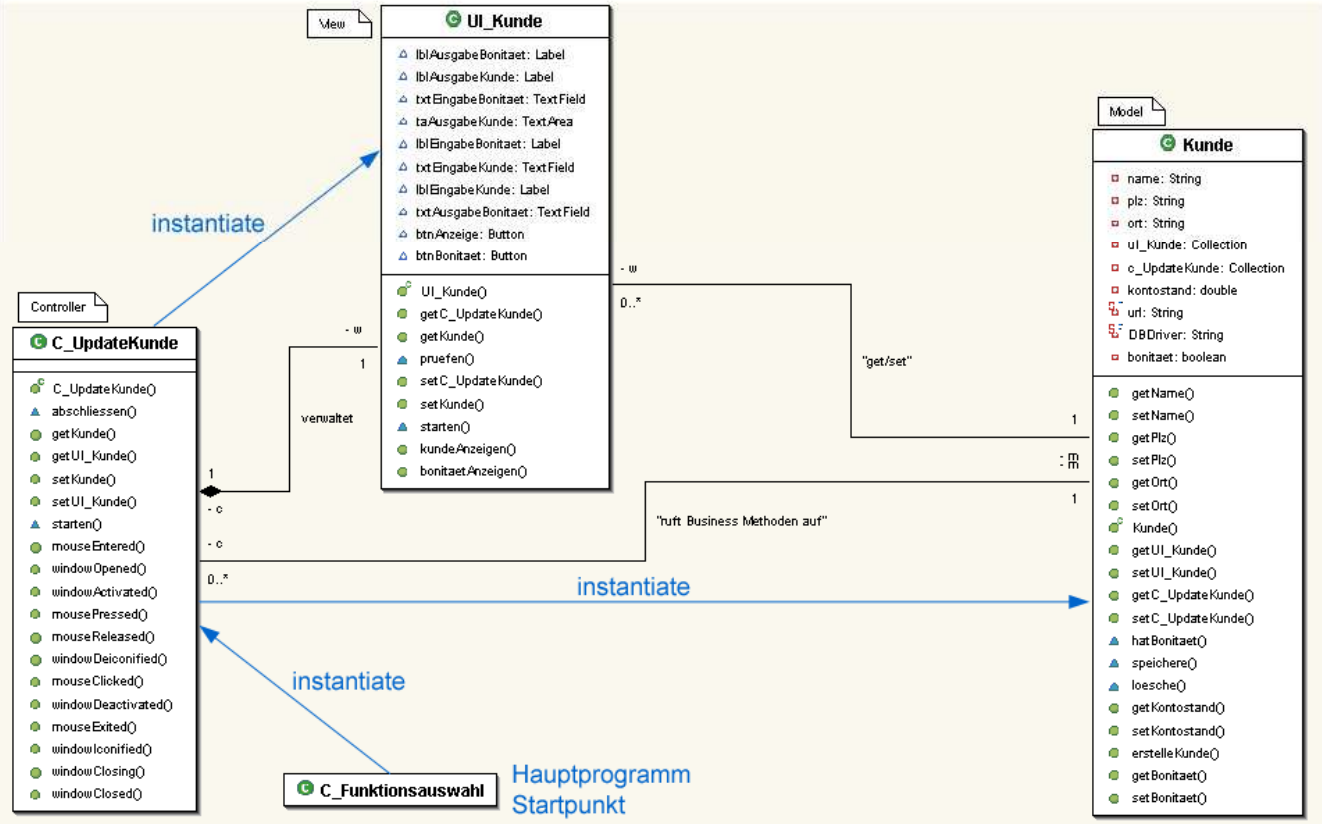
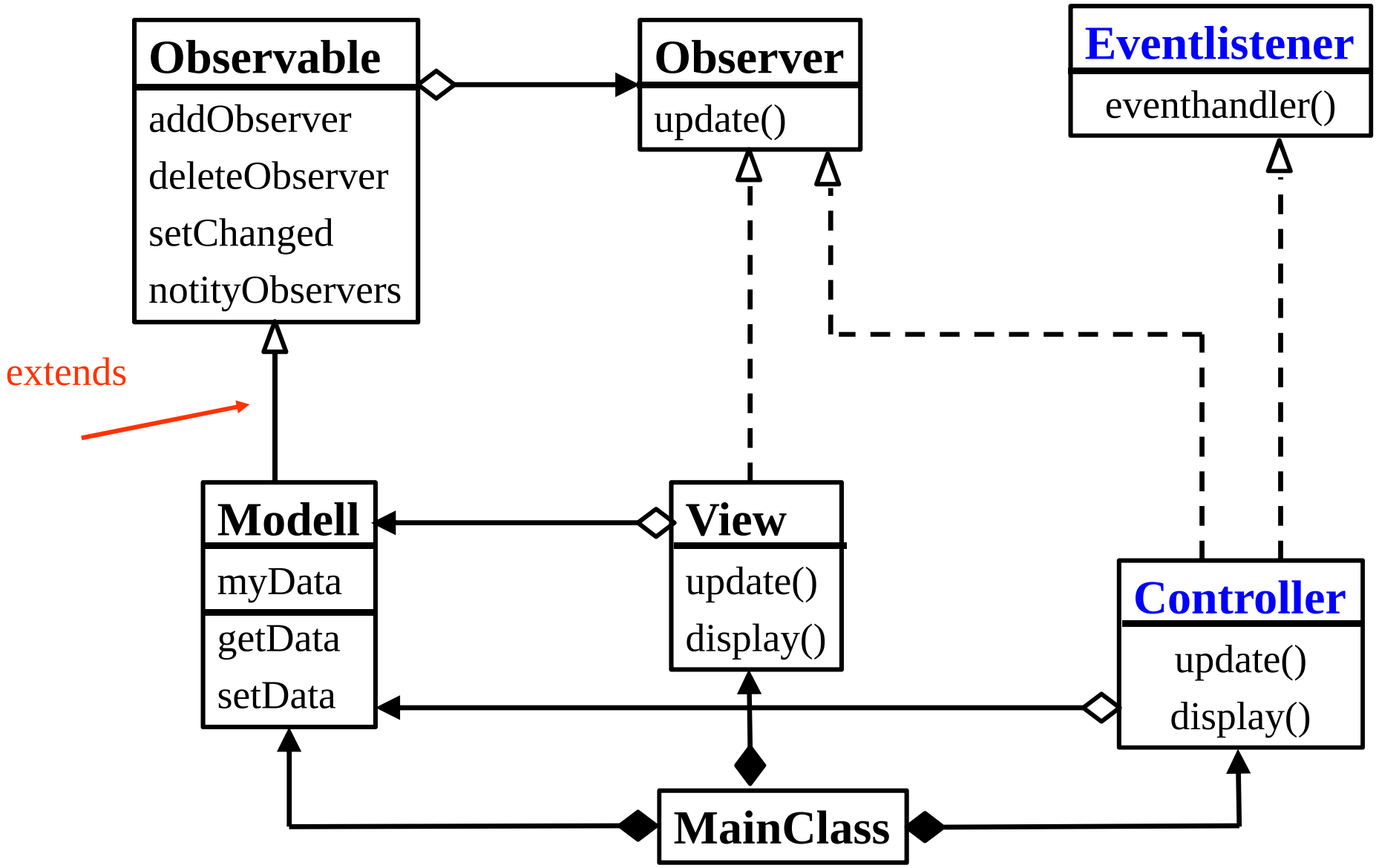
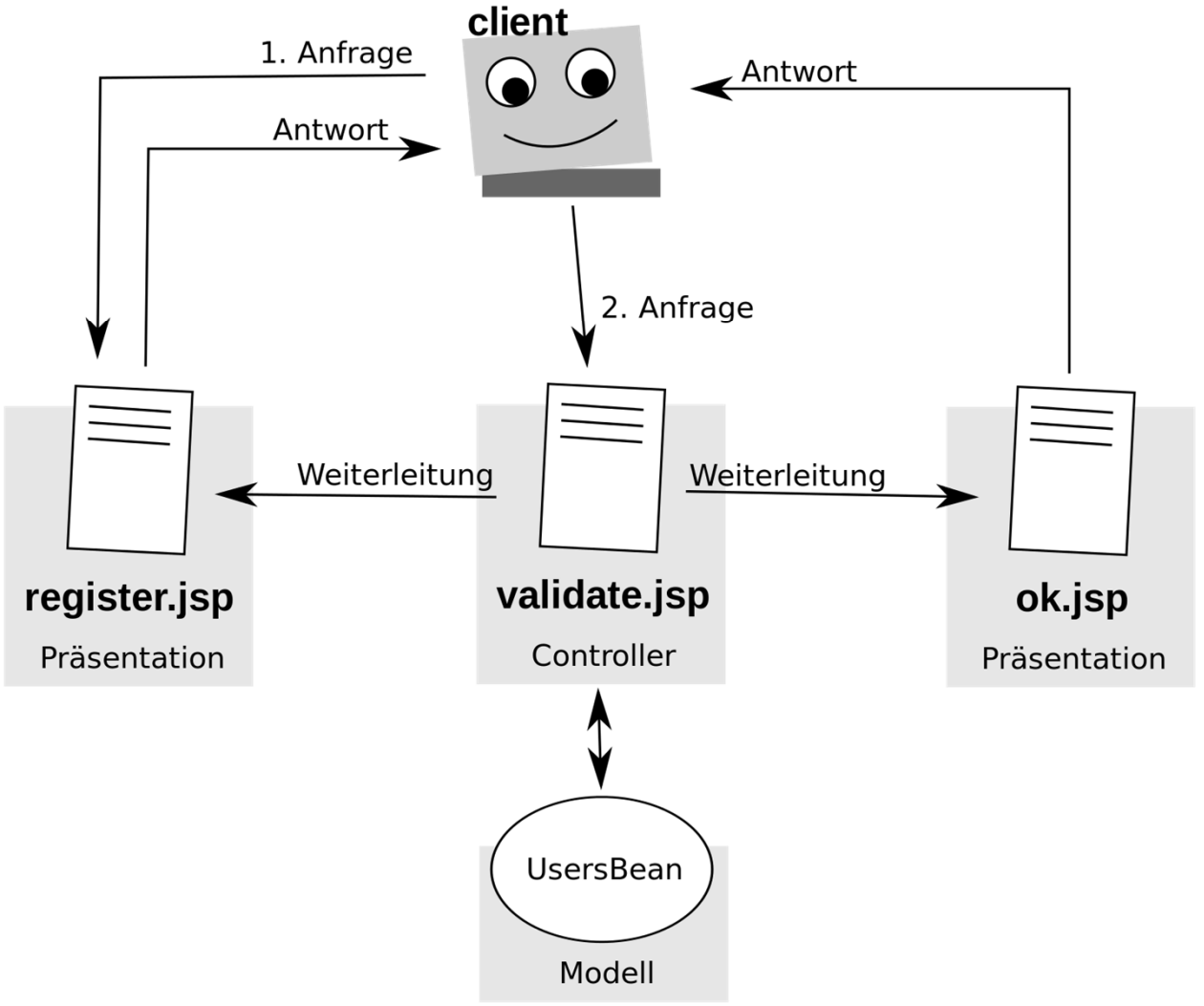


Abb. 3.4: Klassendiagramm MVC Beispiel


Observable
addObserver
deleteObserver
setChanged
notityObservers
Observer
update()
Eventlistener
eventhandler()
extends
Modell
myData
getData
setData
View
update()
display()
Controller
update()
display()
MainClass
interface
Komposition
Aggregation

MVC Entwurfsmuster mit Servern



Quelle: Wikipedia

MVC Server und Browser bei Webanwendungen

- Das MVC-Muster bei Webanwendungen über Server und Browser ist komplexer als das klassische MVC-Muster.
  - Abstrakt betrachtet übernimmt der Browser dabei die sichtbare Darstellung und unmittelbaren Nutzereingaben, sowie die nicht seitenspezifischen Funktionen von Controller und View.
  - Der Server kümmert sich um spezifische Steuerung des Browsers, indem er mit diesem über HTTP kommuniziert.

- Der Begriff MVC findet insbesondere im Rahmen solcher Zusatzprogramme zum Webserver Verwendung.
  - **Modell**:
    - Für den Browser ist die HTML-Seite der Datenkern seines Models. Aus der Perspektive des Gesamtsystems ist sie nur eine Sicht auf das Gesamtmodel, welches auf dem Server lokalisiert ist.
  - **View**
    - Der Browser kümmert sich um die allgemeinen Funktionen, wie die Darstellung von Text, Formularelementen und eingebetteten Objekten. Die Darstellung wird dabei im Speziellen durch den View-Programmteil des Servers per HTTP-Response gesteuert, deren Hauptteil der Darstellungsanweisung aus der HTML-Seite besteht.
  - **Controller**
    - Der Browser akzeptiert Formulareingaben und sendet diese ab oder nimmt das Anklicken eines Links entgegen. In beiden Fällen sendet er einen HTTP-Request an den Server. Der Controller-Programmteil verarbeitet die Daten der HTTP-Requests und stößt schließlich die Erstellung eines neuen Views an.
  - **Javascript**
    - Die Webseite kann Programmcode enthalten, normalerweise Javascript, z. B. für die browserseitige Validierung von Formulareingaben. Dieser Programmcode lässt sich wiederum nach dem MVC-Muster gliedern und so als Teil des Gesamtsystems betrachten.

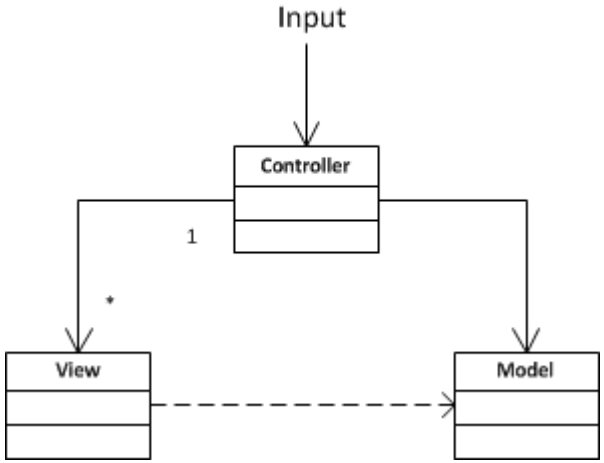
Verzicht auf das Observer-Muster

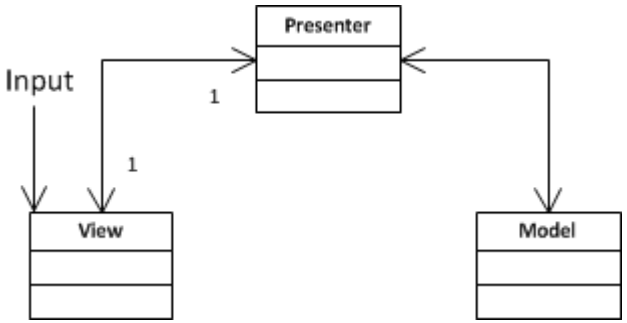
- Bei Webanwendungen kann der Browser nicht nach dem klassischen Observer-Muster unmittelbar auf Änderungen des Models auf dem Server reagieren. Jede Antwort (HTTP-Response) an den Browser setzt eine Anfrage (HTTP-Request) voraus. Man spricht vom Request-Response-Cycle. Daraus folgt, dass das Observer-Muster auch auf Seiten des Servers seine Vorteile nicht ausspielen kann. Weil es dann nur einen Mehraufwand bedeuten würde, kommt es typischerweise nicht zum Einsatz. Stattdessen tritt meist der **Controller** als aktiver Vermittler zwischen Model und View im Rahmen eines Request-Response-Cycles auf.
- Die besonderen Herausforderungen des Hyperlinks und der Form-Action
  - Der Hyperlink ist ein herausragendes Merkmal von Webapplikationen. In einer klassischen GUI-Applikation würde hier im View ein Button erzeugt, dessen Klickevent anhand seiner ID im Controller mit dem Wechsel der Ansicht verknüpft wird. Der Hyperlink enthält zwar auch eine ID, es ist aber nicht seine eigene, sondern die Zieladresse der neuen Ansicht. Gleiches gilt für die Action-Adresse eines HTML-Formulars.
  - Beides sind für den Benutzer eines Browsers Controller-Elemente, deren funktionales Ziel allerdings in der Webseite codiert ist. Hiermit stellt sich die Herausforderung, bei der Erzeugung der Webseite die reine Ansicht von der Funktionalität der URL zu trennen. Der Entwickler des Views soll sich keine Gedanken über die oft komplexe Controller-Funktionalität der URL machen müssen.

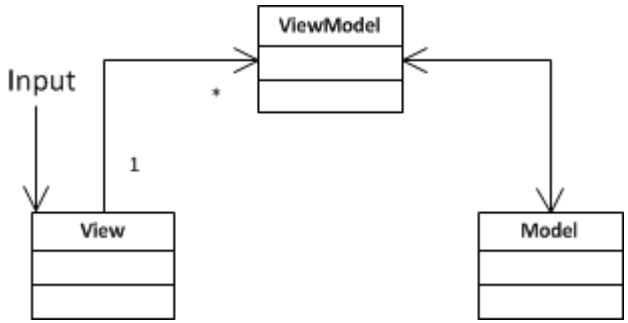
- Die Aufgabenteilung von View und Controller kann mittels einer ID einfach erreicht werden. In Analogie zur GUI-Applikation wird die ID im Controller mit der Zieladresse verknüpft. Im View kann die URL über eine Schnittstelle anhand der ID abgerufen werden oder die ID tritt als Platzhalter für die URL ein, zum Beispiel innerhalb eines Templates oder in Form von Link-Objekten im Rahmen eines Objektbaums.
- **JavaScript-Bibliotheken und AJAX-Anbindung**
  - Hier wird ein Teil der Programme der Model-View-Controller-Architektur clientseitig im Browser eingesetzt, während ein anderer Teil, insbesondere das Model, auf dem Server verbleibt. JavaScript-Bibliotheken stellen vielfältige Widgets zur Verfügung. Diese Anwendungen nehmen eine Zwischenstellung zwischen Webanwendungen und desktopartigen Widget-Bibliotheken ein.
- **Serverseitige Webanwendungen**
  - Der serverseitige Controller wertet in der Regel die eintreffenden Daten (Request) des Browsers aus. Meist tritt er dabei als Moderator zwischen Model und View auf. Serverseitig werden unter dem View diejenigen Programmteile verstanden, die den HTML-Code für die Antwort (Response) erzeugen. Häufig arbeitet der View mit HTML-Templates, deren Platzhalter mit den Daten des Models ersetzt werden.

MVC vs. MVP vs. MVVM

- **Gemeinsam: das Modell und mehrere Views**
- Die Logik liegt beim MVC im Controller
- Die Logik liegt beim MVP im Presenter
- Die Logik liegt beim MVVM im ViewModell
- **Unterschiede liegt in den Assoziationen**
  - Beim MVC steuert der Controller sowohl View, als auch Model.
  - Beim MVP bedingt sich alles gegenseitig.
  - Das Besondere an MVVM ist nun, dass das ViewModel nicht die View‘s steuert, sondern lediglich Daten für diese bereitstellt.
  - Das ganze funktioniert über eine Technik, die sich **Data Binding** nennt. Dies erlaubt eine sehr lose Kopplung zwischen den Schichten zu bekommen. Darüber hinaus ist es durch MVVM möglich, per **Unit Test** die GUI zu testen.







MVC Beispiel mit einem JPanel als View

```
public class View extends JPanel implements Observer {
   private JLabel label;
   private JSlider slider;

   public void update (Observable o ,Object arg ) {
          if (o instanceof Model) {
             Model model = (Model)o;
             int z = model.getZaehler();
             label.setText (Integer.toString(z));
             slider.setValue(z);
          }
   }
}
```

MVC Beispiel mit einem JPanel als View

```
private JButton buttonFactory(Controller c ,
        String name, String label, Object value) {
  JButton b = new JButton(label);
  b.setName (name);
  c.putValue (name, value);
  b.addActionListener(c);
  return b;
}
```

MVC Beispiel mit einem JPanel als View

```
private JSlider sliderFactory(Controller c,
    Model m, String name, Object value ) {
    JSlider s = new JSlider(m.getZaehlerMin(),
    m.getZaehlerMax(), m.getZaehler());
    s.setName(name);
    c.putValue(name, value);
    s.addChangeListener(c);
    return s;
}
private JLabel labelFactory (String name) {
   JLabel l = new JLabel (name, JLabel.CENTER);
   l.setFont (new Font("SansSerif", Font.BOLD, 30));
   return l;
}
```

MVC Beispiel mit einem JPanel als View

```
public View(Model m, Controller c ) {
   JButton up = buttonFactory (c, "UpButton", ">>>>>", 1);
   JButton dn = buttonFactory (c , "DownButton", "<<<<<" , -1);
   slider = sliderFactory (c, m, "Slider", 0);
   String name = Integer.toString( m.getZaehler() );
   label = labelFactory (name);
   setLayout (new GridLayout(0, 1));
   add (up) ;
   add (label) ;
   add (slider) ;
   add (dn) ;
}
```

MVC Beispiel mit einem JPanel als View

```
public class Controller extends AbstractAction
        implements ChangeListener {
  private Model model;
  public Controller (Model m) { model = m; }

  public void stateChanged (ChangeEvent e ) {
        Object changeObject = e.getSource();
        if (changeObject instanceof JSlider) {
          JSlider s = (JSlider) changeObject;
          int sliderValue = s.getValue();
          model.setZaehler(sliderValue);
    }
  }
```

MVC Beispiel mit einem JPanel als View

```
public void actionPerformed(ActionEvent e) {
  Object actionObject = e.getSource();
  if ( actionObject instanceof JButton ) {
     JButton b = (JButton)actionObject;
     String buttonName = b.getName();
     Object buttonValue = this.getValue (buttonName);
     if ( buttonValue != null && buttonValue instanceof Integer ) {
       int delta = (Integer)buttonValue;
       model.incZaehler(delta);
     }
   }
 }
}
```