70. Programmierung interaktiver Systeme

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

TU Dresden

Version 13-1.1, 05.11.13

Obligatorische Literatur

- [PassiveView] Martin Fowler. Passive View. http://www.martinfowler.com/eaaDev/PassiveScreen.html. Strikte Schichtung und passiver View.
- Spring http://docs.spring.io/spring/docs/3.1.x/
- Rod Johnson. J2EE development frameworks. IEEE Computer, 38(1):107-110, 2005.
  - http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1381270

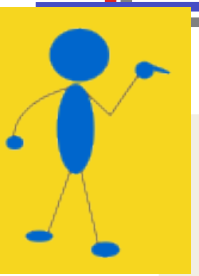
Andere Literatur

- F. Buschmann. N. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-orientierte Software-Architektur. Addison-Wesley.
  - Entwurfsmuster und Architekturstile. MVC, Pipes, u.v.m.
- [Herrmann] M. Veit, S. Herrmann. Model-View-Controller and Object Teams: A Perfect Match of Paradigms. Aspect-Oriented System Development (AOSD) 2003, ACM Press
- Mike Potel. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java. VP & CTO Taligent, Inc.
  - ftp://www6.software.ibm.com/software/developer/library/mvp.pdf
- html web frameworks
  - STRUTS http://exadel.com/tutorial/struts/5.2/guess/strutsintro.html
  - Web Application Component Toolkit http://www.phpwact.org/pattern/model_view_controller

70.1 Einfache Kopplung über Play-In und Play-Out mit Java-AWT

- Nur Modell und Sicht, kein Controller

Vereinfachte Schichtenarchitektur (zunächst ohne Controller)



Graphische Benutzungsoberfläche
(Fensteroberfläche mit Widget-Hierarchie, graphical user interface, GUI)

Window

Menu

Button

Play-In (asynchron, push)

Play-out (synchron, aber multiple Sichten und pull-Steuerung)

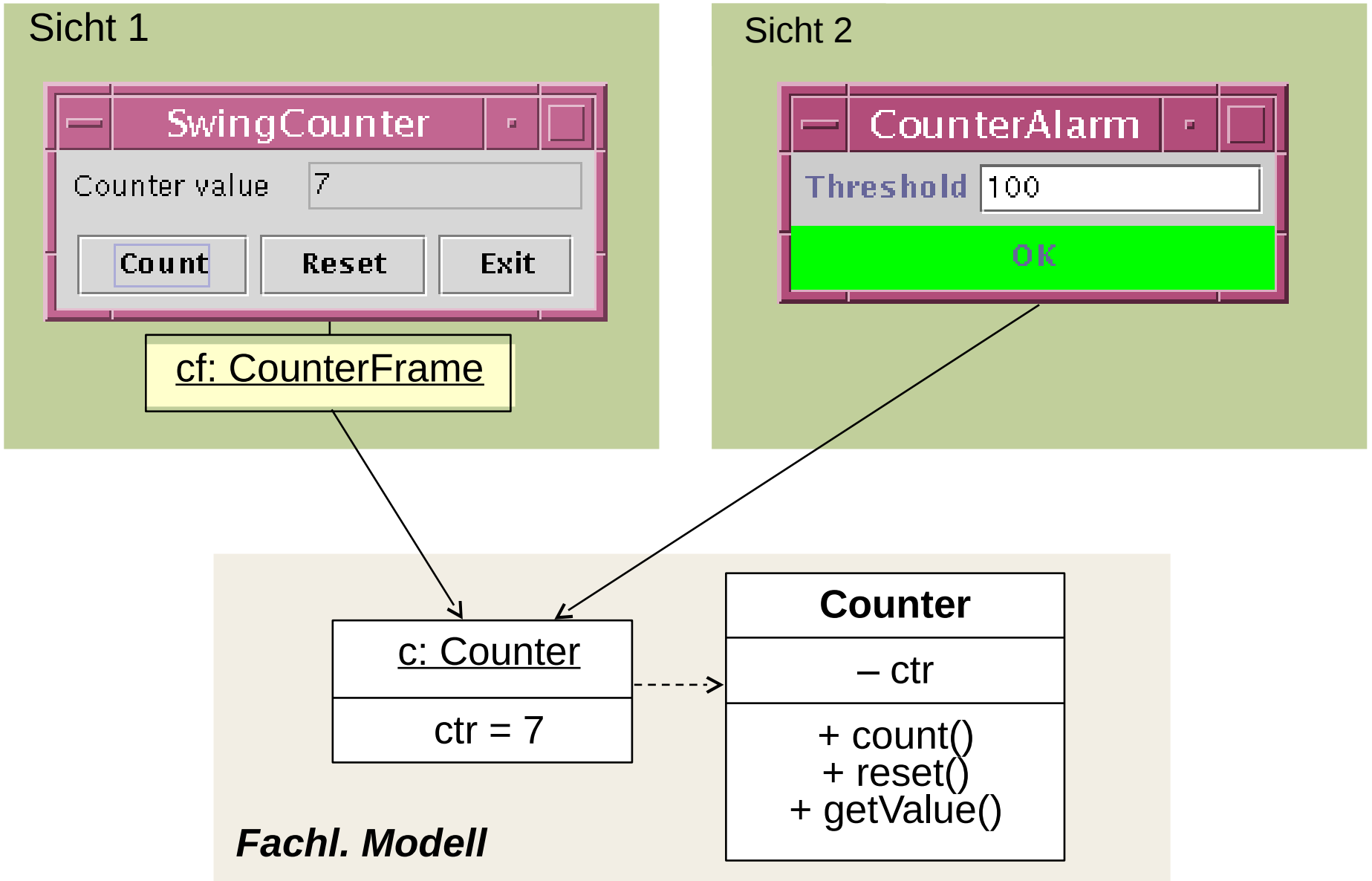
Unsichtbare Ablauf-Steuerung (Ereignis-Verwaltung)

Teammitglied

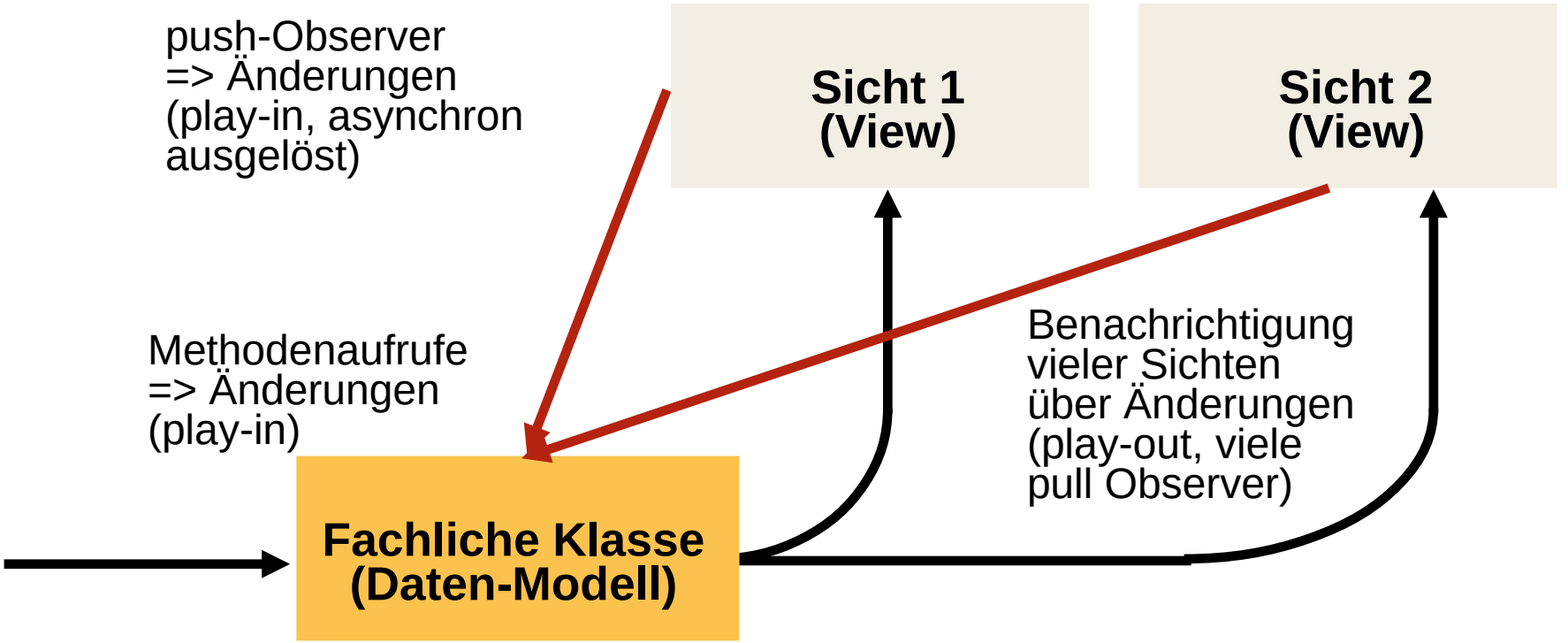
Fachliches Modell (Anwendungslogik)

Teambesprechung

Sichten: Zähler als motivierendes Beispiel



Daten-Modell und Sicht (ohne Controller)



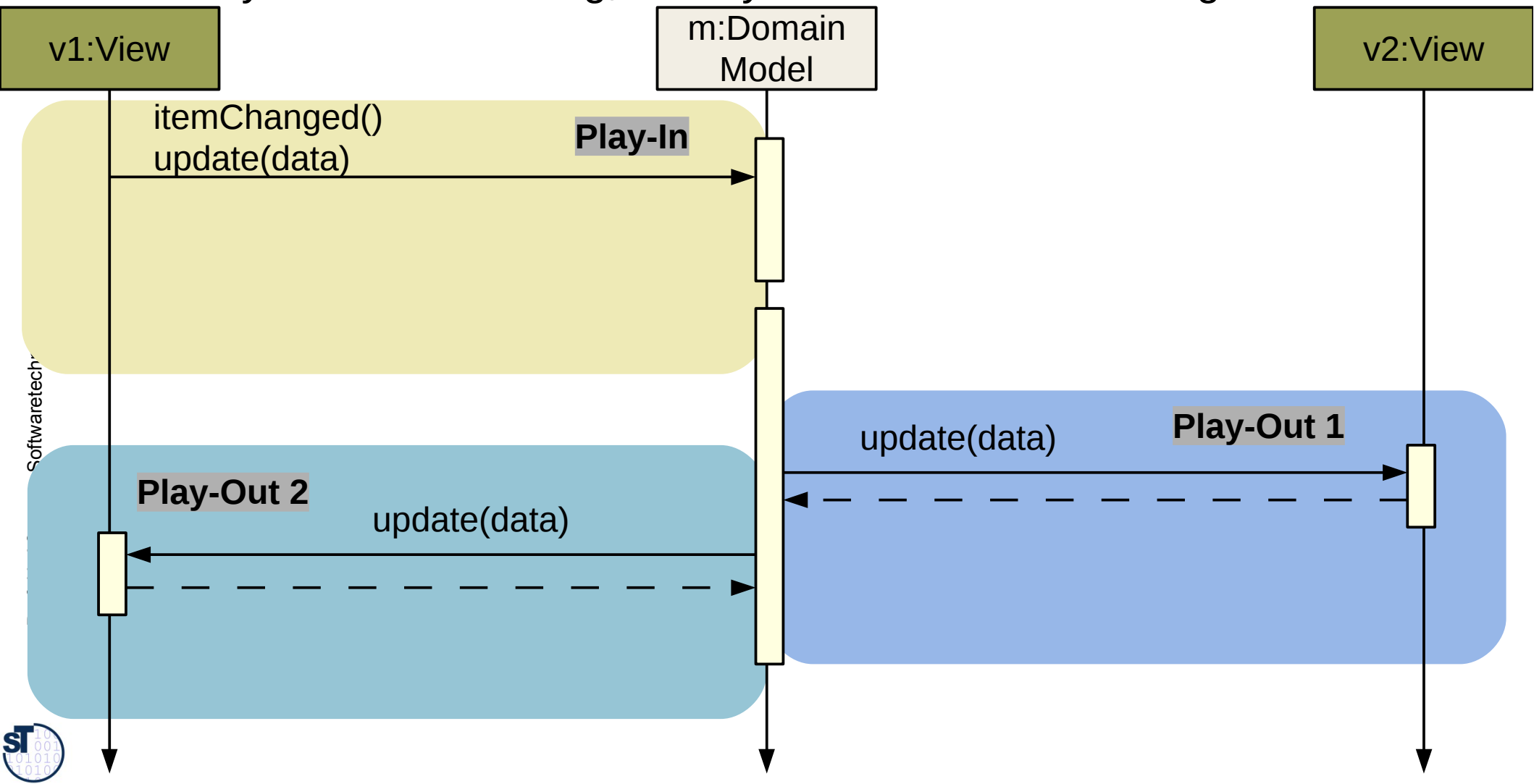
Beispiele: Verschiedene Dokumentenansichten, Statusanzeigen, Verfügbarkeit von Menüpunkten

Frage: *Wie hält man das Modell in der Anwendungslogik unabhängig von den beliebig vielen Sichten darauf ?*

Muster "Observer"

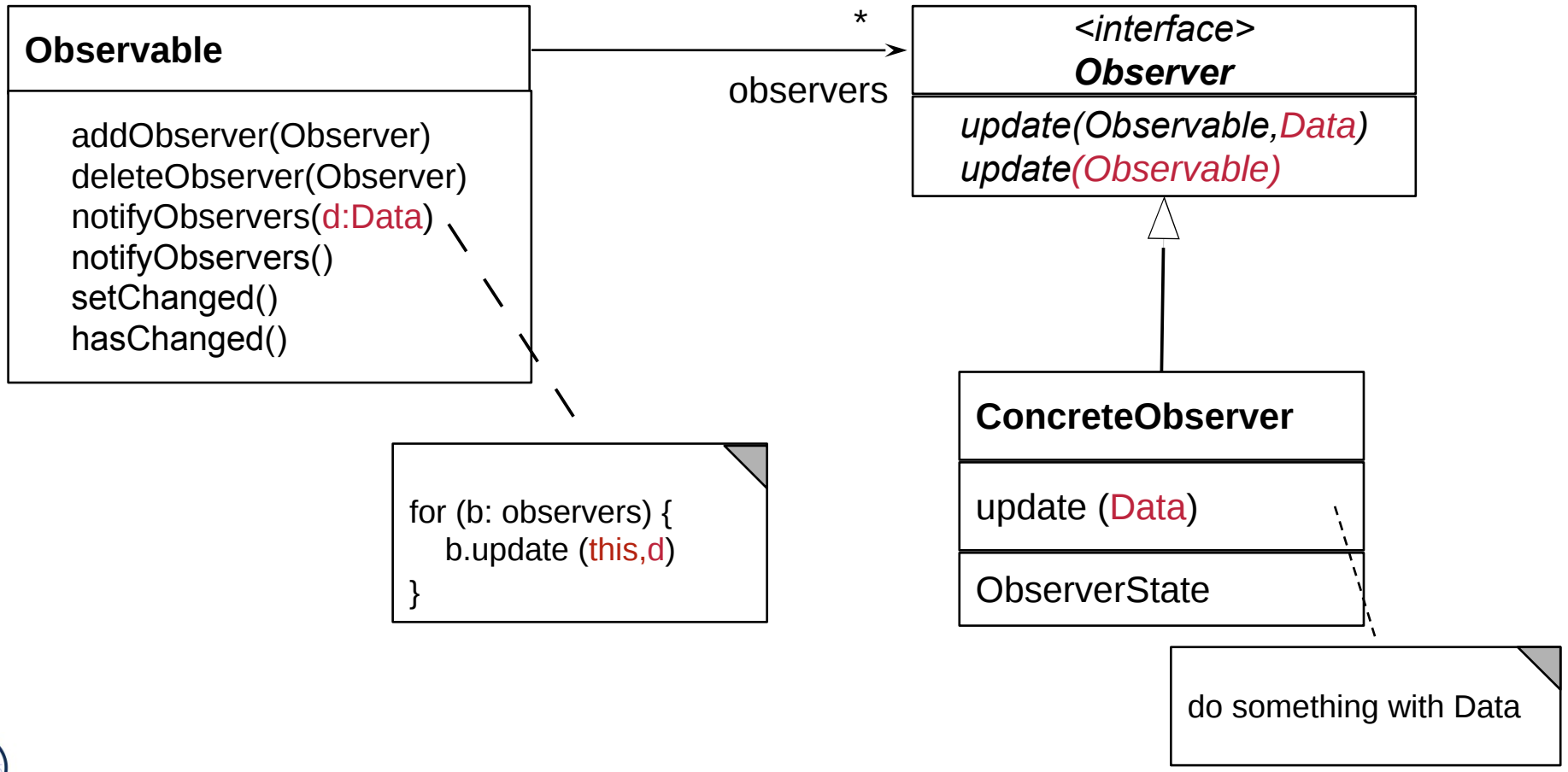
Play-In mit push-Observer; Play-Out mit push-Observer

- Push-Observer werden vom JDK unterstützt
- Nachteil: Daten werden mit dem update() geschoben
- Play-In-Observer nötig, falls dynamisch viele Views registrieren



Struktur java.util.Observer (push-Observer) für Play-Out

- Das JDK bietet mehrere Implementierungen des Entwurfsmusters Observer an
  - java.util Observer, java.awt.Window
- java.util.Observer folgt dem Muster “Subject-Passing push-Observer”
- Subjekt `Observable` schiebt Daten mit `update(this,Data)`
- Abweichung: `Observable` ist konkrete Klasse

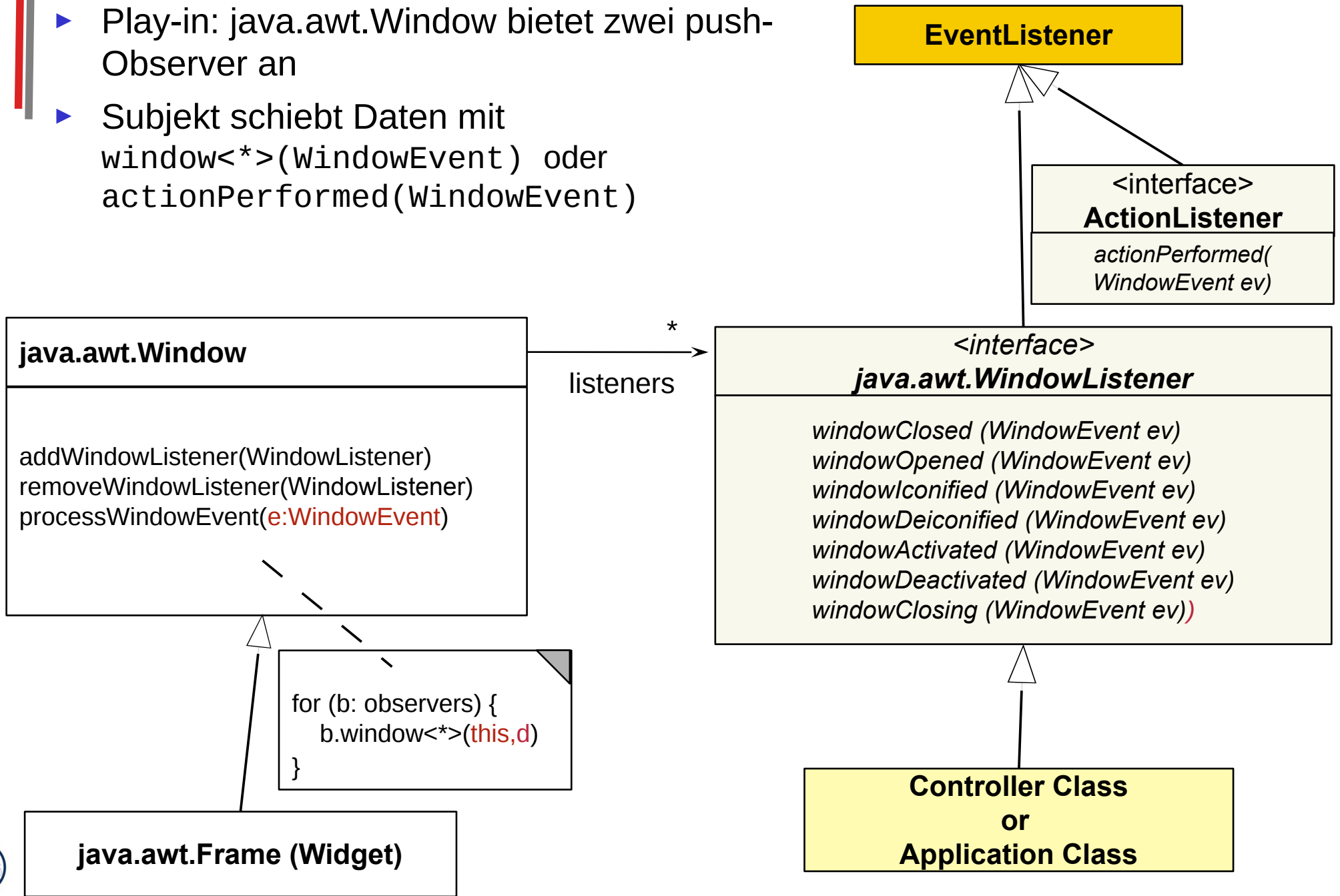


Grundversion: Hauptprogramm für Fensteranzeige


```
import java.awt.*;

class ExampleFrame extends Frame {
  // Fensteroberfläche
  public ExampleFrame () {
     setTitle("untitled");
     setSize(150, 50);
     setVisible(true);
   }
}

class GUI1 {
  public static void main (String[] argv) {
    // Phase 1: Aufbau der Fensteroberfläche
    ExampleFrame f = new ExampleFrame();
    // .. implizites Betreten der Reaktionsschleife:
    // Phase 2: reaktives Programm
  }
}
```

Play-In mit java.awt.WindowListener (push-Observer)

- Play-in: java.awt.Window bietet zwei push-Observer an
- Subjekt schiebt Daten mit `window<*>(WindowEvent)` oder `actionPerformed(WindowEvent)`



Registrierung für java.awt.WindowListener (Play-In)

- Im „Auslöser“ java.awt.Frame findet sich eine Registrierungsprozedur (ererbt von java.awt.Window):

```
public class Frame ... {
  public void addWindowListener
     (WindowListener l)
}
```

Observable (Subject)

Observer

- java.awt.event.WindowListener ist eine Schnittstelle eines Observers, die von der Anwendungsklasse implementiert werden kann:

```
public interface WindowListener {
  ... Methoden zur Ereignisbehandlung
}
```

Observing (Observer)

Erste Verbesserung: Hauptprogramm für schließbares Fenster (play-in)


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

class WindowCloser implements WindowListener {
   ... siehe später ...
}

class ExampleFrame extends Frame {

  public ExampleFrame () {
     setTitle("untitled");
     setSize(150, 50);
     addWindowListener(new WindowCloser());
     setVisible(true);
   }
}
class GUI2 {
  public static void main (String[] argv) {
    ExampleFrame f = new ExampleFrame();

  }
}
```

Listener des play-in

Anmelden des play-in

Subjekt starten (Ereignisverwaltung)

reagieren (Phase 2)

Ein Zähler (Beispiel für Domänenobjekt im fachliches Anwendungsmodell)


```
class Counter           {
  private int ctr = 0;
  public void count () {
    ctr++;


  }
  public void reset () {
    ctr = 0;


  }
  public int getValue () {
    return ctr;
  }
}
```

Beobachtbares Anwendungsmodell *(Play-out)*

- Counter wird durch Sicht beobachtet, z.B. mit jdk-Implementierungsmuster java.util.*Observer*
- Counter ist *Subjekt (Klasse java.util.Observable)*
  - bei Veränderung des Counter werden die Sichten mit `setChanged()` benachrichtigt (play out)

```
// Application logic
import java.lang.util.*;
class Counter extends Observable {
  private int ctr = 0;
  public void count () {
    ctr++;
    setChanged();
    notifyObservers();
  }
  public void reset () {
    ctr = 0;
    setChanged();
    notifyObservers();
  }
  public int getValue () {
    return ctr;
  }
}
```

```
class Connector {
  Counter counter;
  counter = new Counter();
  view = new GraphicCounterView();
  view2= new TextualCounterView();

  // wire view and model (play-out)
  counter.addObserver(view);
  counter.addObserver(view2);
}
```

Nachteile der Architektur ohne Controller

- GUI und Aufrufe an die Anwendungslogik sind *vermischt* (*tangled*)
- Aufrufe an die Anwendungslogik sind über den GUI *verstreut* (*scattered*)
- Keine Trennung möglich
- Koordination fest zwischen GUI und Anwendungslogik eingebacken
  - Kein Wechsel der Strategie der Koordination möglich
  - Keine echte Asynchronität möglich

70.2 Phase 1: Aufbau der Schichten

Anwendungslogik: Der Aufbau der Anwendungslogik wurde bereits in der Vorlesung besprochen.

Die 3-Phasen des reaktiven GUI:

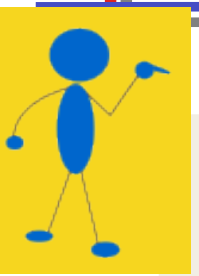
1)Schichtenaufbau

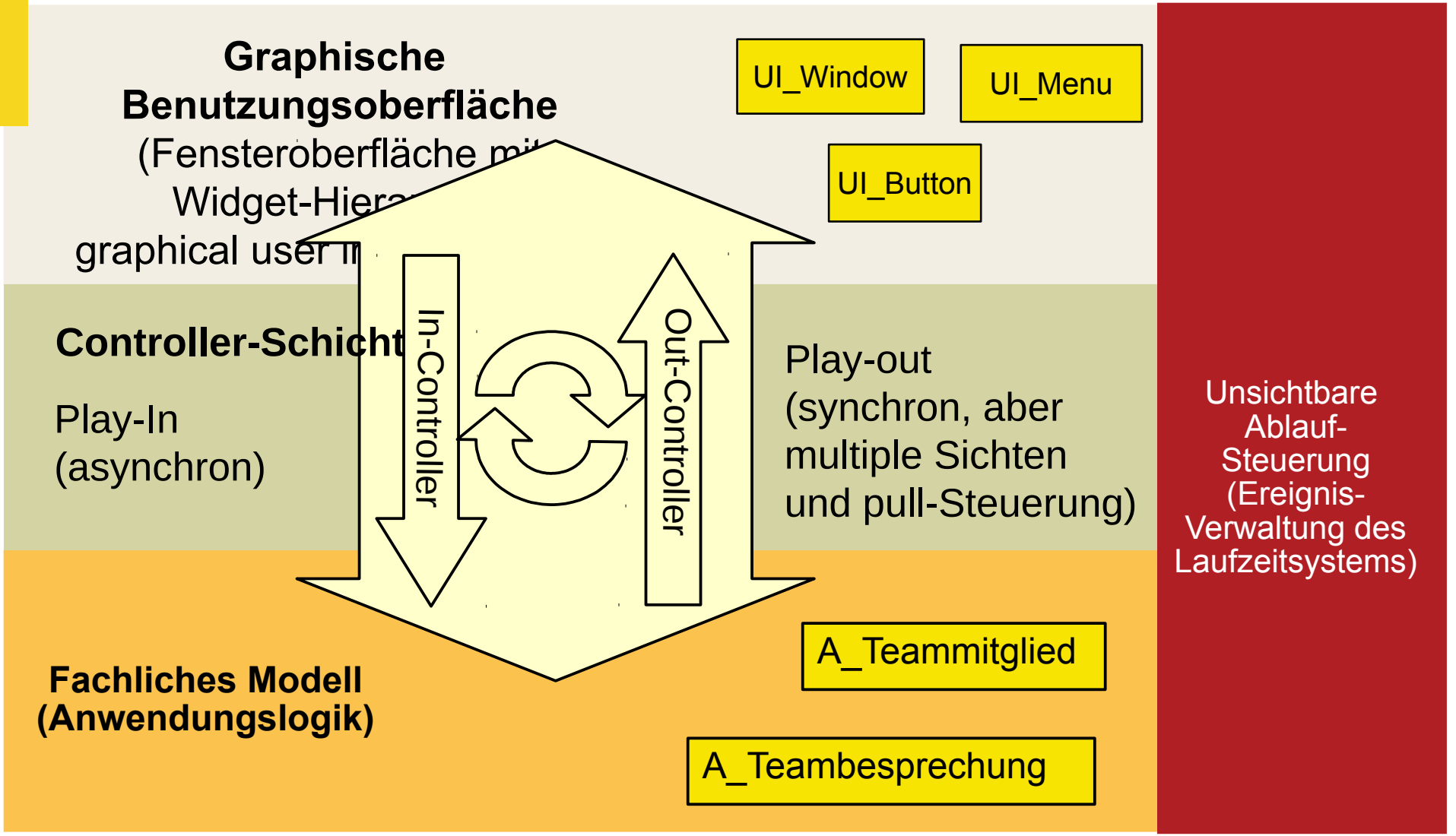
- Aufbau Anwendungslogik
- Aufbau des Input-Controller: empfängt Ereignisse
- Aufbau der Widgets: Aufbau der Fensteroberflächen

2)Netzaufbau

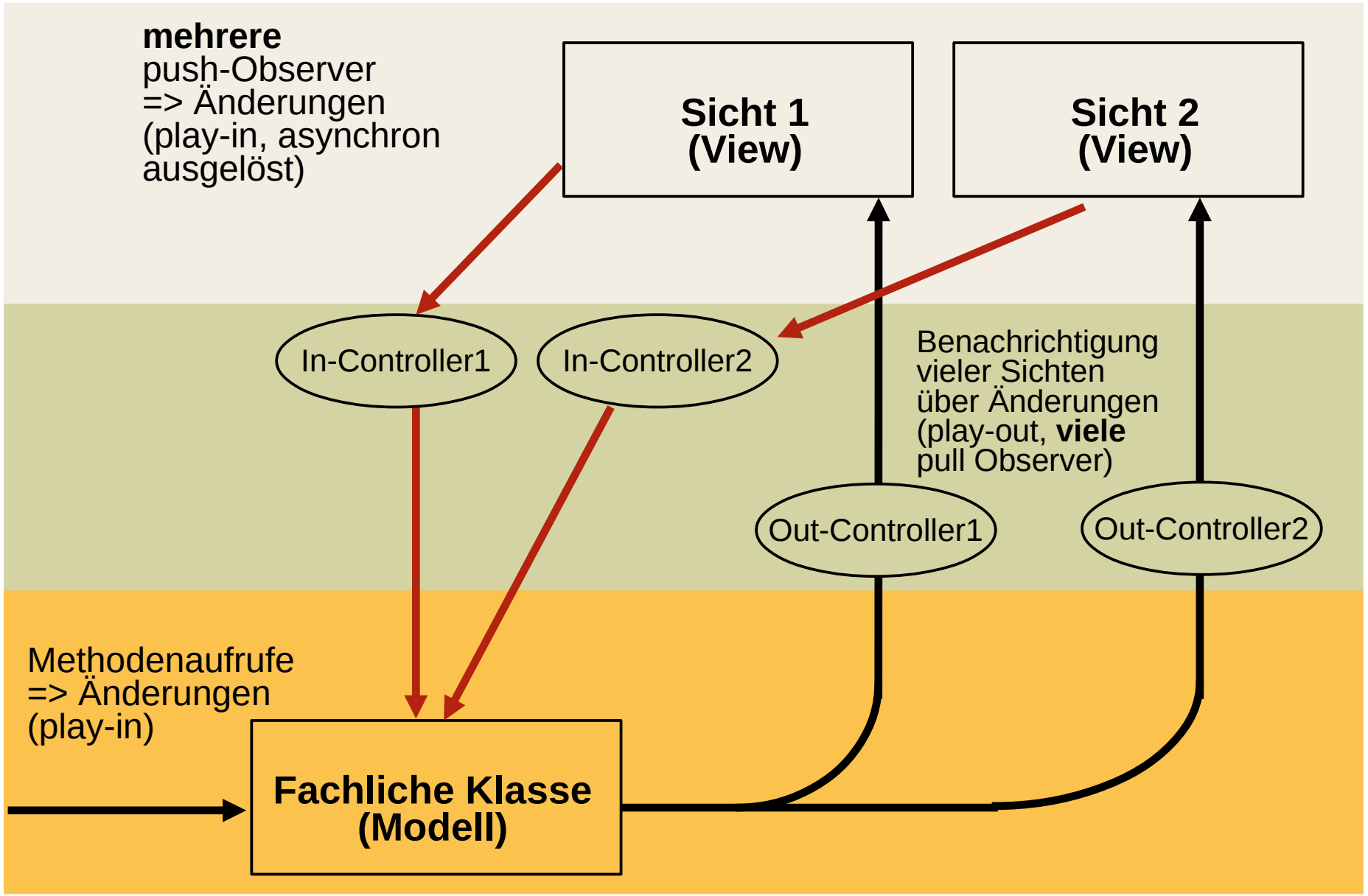
3)Reaktionsphase

Erinnerung: Schichtenarchitektur der reaktiven Benutzungsoberfläche (GUI)



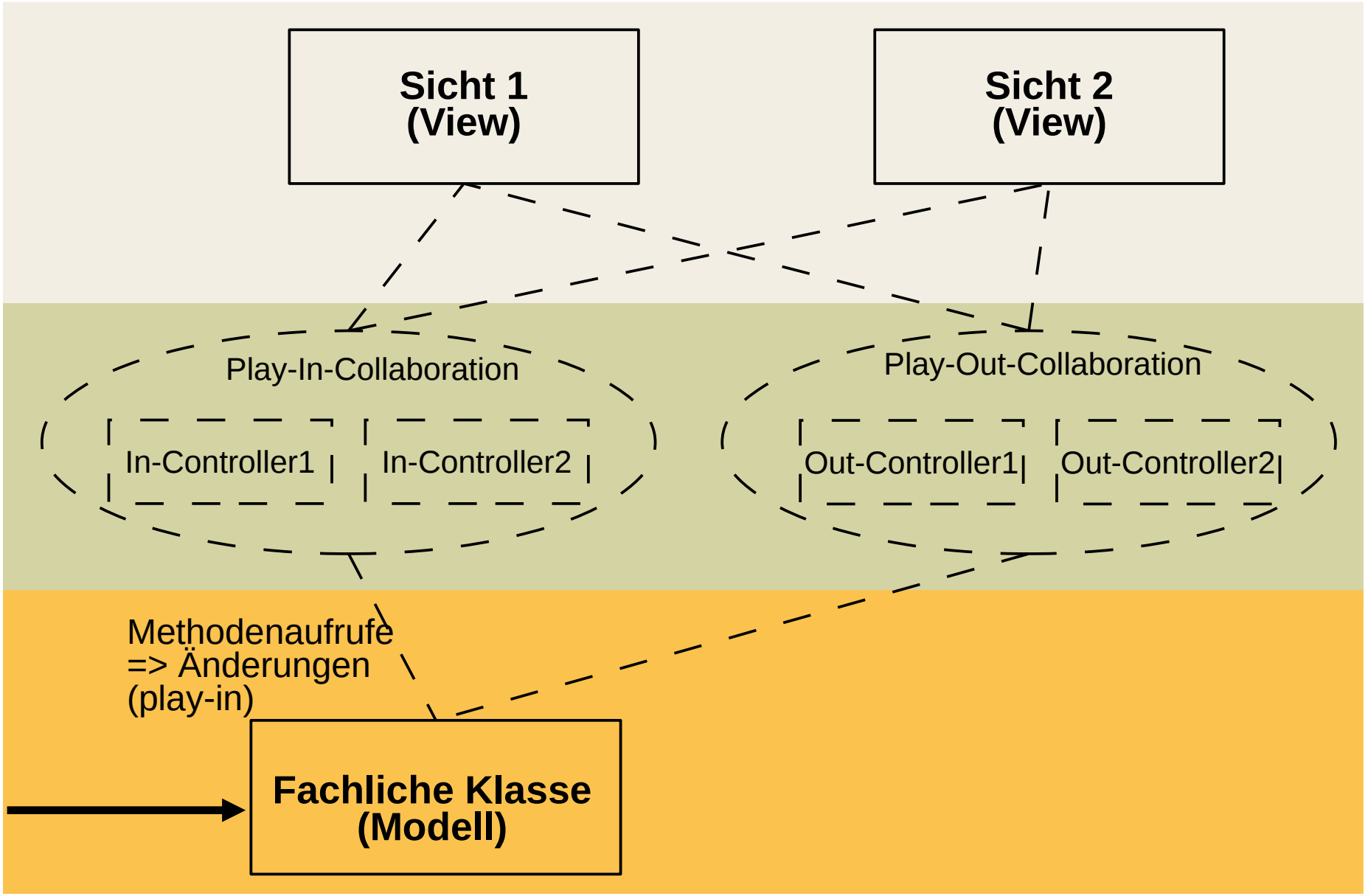


Erinnerung: Modell, Controller und Views in strikter Schichtung

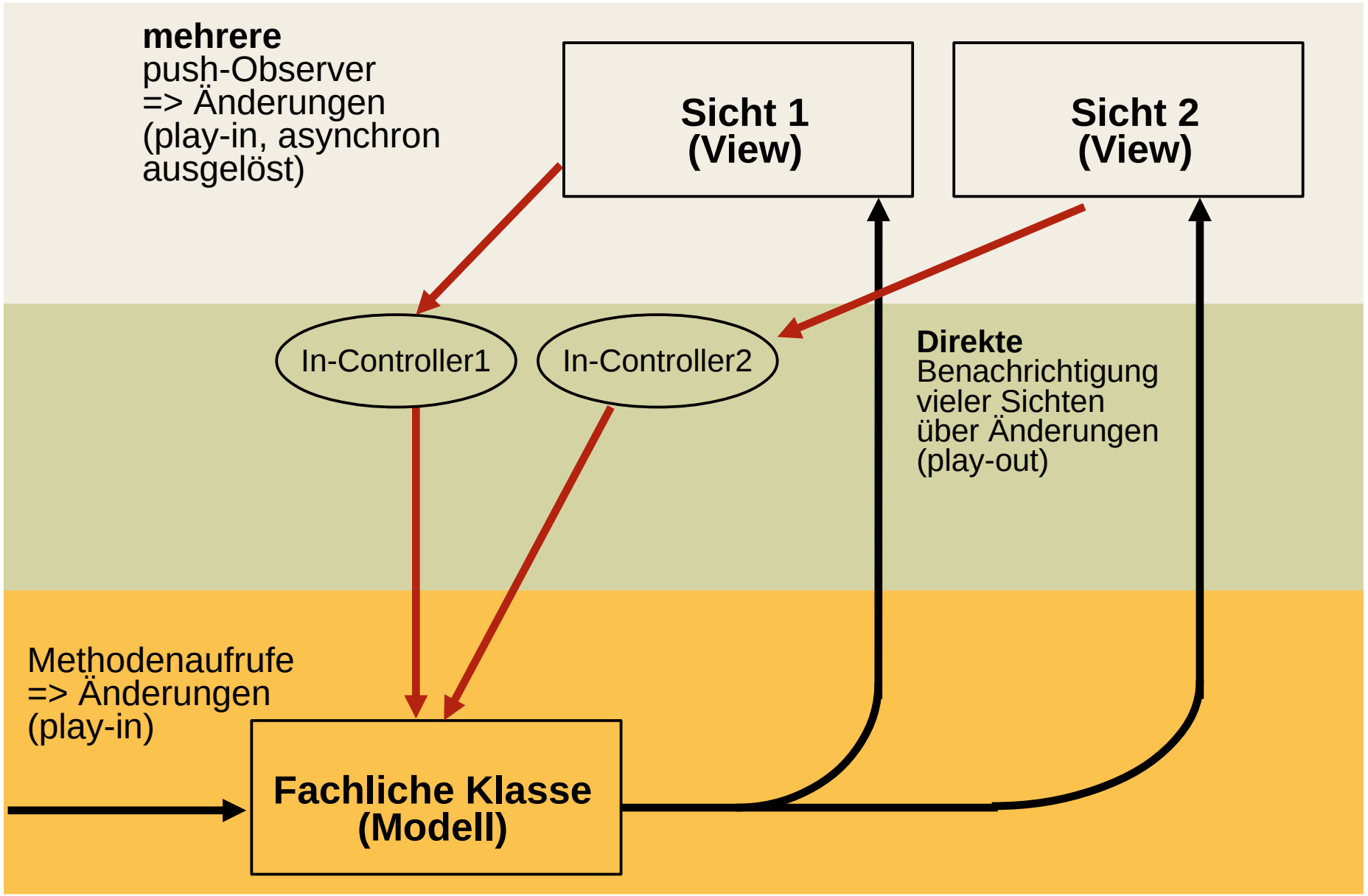


Controller sind Kollaborationen zwischen Model und View

- Gibt es ein Hauptobjekt in der Kollaboration, ist der Controller ein Konnektor



Modell, Controller und Views in schwacher Schichtung



70.2.1 Ereignismeldung beim Play-In in Java-AWT-Anwendungen

Asynchrone Änderungen des Benutzers;
push-Observer beim Play-In zwischen GUI und Input-Controller

I claim not to have controlled events,
but confess plainly that events have controlled me.
Abraham Lincoln, 1864

Ereignisse und Benutzerinteraktionen

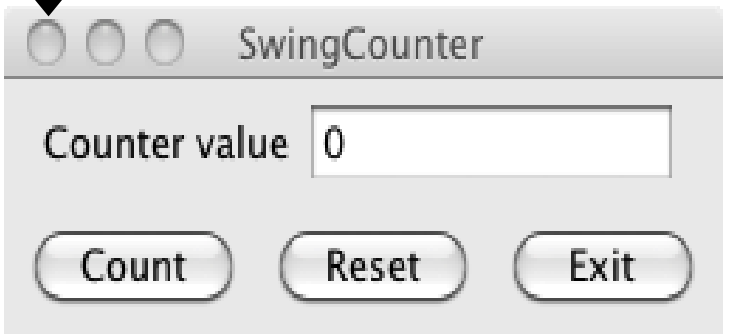
- Ein **Ereignis** ist ein Vorgang in der Umwelt des Softwaresystems von vernachlässigbarer Dauer, der für das System von Bedeutung ist.
- Der Input-Controller empfängt vom View über die Ablaufsteuerung Ereignisse und muss darauf reagieren, in dem der sie in Aktionen auf der Anwendungslogik übersetzt
- Eine wichtige Gruppe von Ereignissen sind **Benutzeraktionen**, Ereignisse, die eine *Aktion* in einem *Kontext* der Benutzeroberfläche ausdrücken:
  - Drücken eines Knopfs
  - Auswahl eines Menüpunkts
  - Verändern von Text
  - Zeigen auf ein Gebiet
  - Schließen eines Fensters
  - Verbergen eines Fensters
  - Drücken einer Taste
  - Mausklick über einem Gebiet

Beispiel für Ereignisverarbeitung

- Das erste Java-Programm in der Vorlesung mit einer "graphischen Benutzungsoberfläche"...
- Aufgabe: Ein leeres, aber schliessbares Fenster anzeigen

Schliessknopf

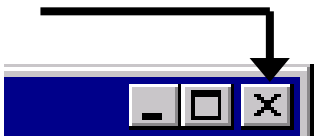
Mac:



Schliessknopf

Motif:

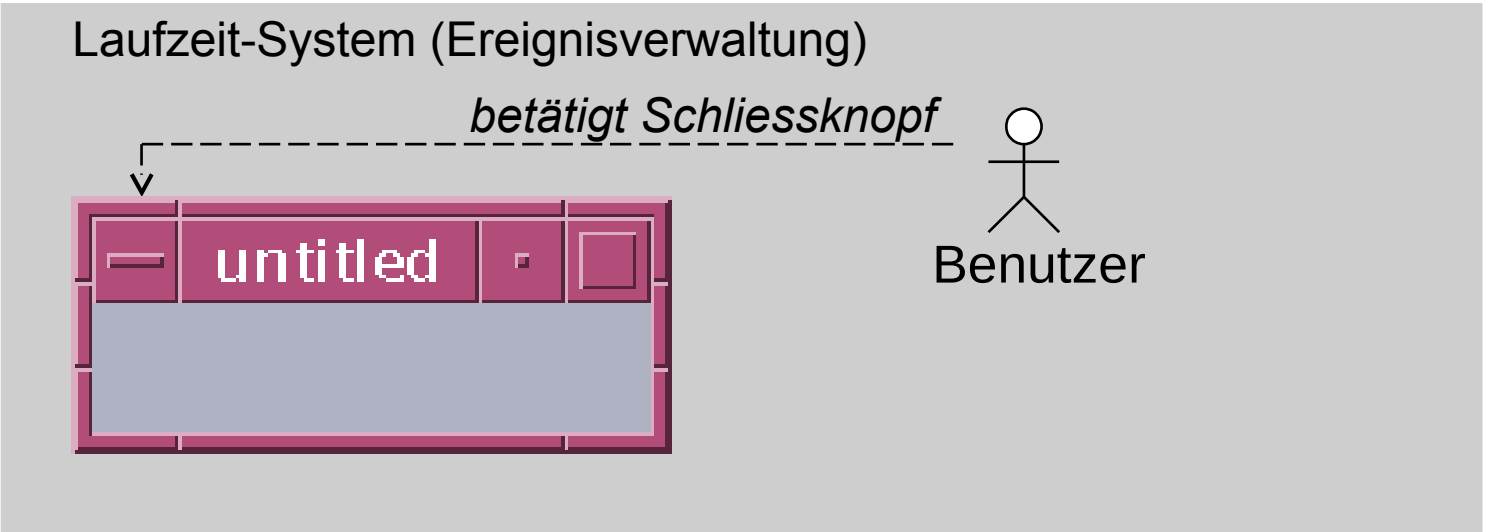
Fensterdarstellung ("look and feel") gemäß Windows:



Ereignis-Klassen und ihre “auslösenden” Oberflächenelemente

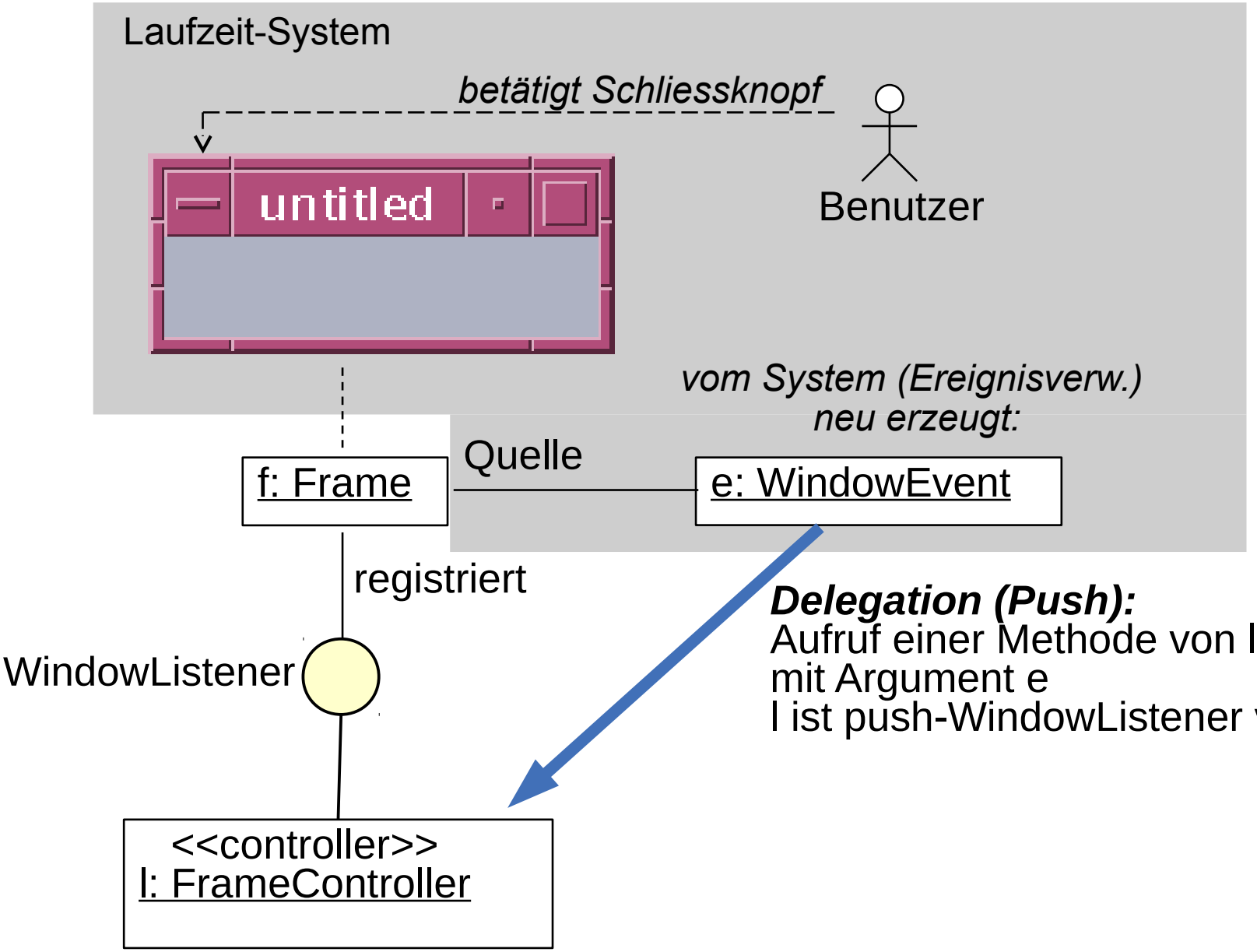
- Ereignis-Klassen in (Java-)Benutzungsoberflächen drücken die Aktion oder den Kontext der Benutzerinteraktion aus:
  - WindowEvent
  - ActionEvent
  - MouseEvent, KeyEvent, ...
- Bezogen auf Klassen für Oberflächenelemente (Kontexte)
  - Window
  - Frame
  - Button
  - TextField, ...
- Zuordnung (Beispiele):
  - Window (mit Frame) erzeugt WindowEvent
    - z.B. Betätigung des Schliessknopfes
  - Button erzeugt ActionEvent
    - bei Betätigung des Knopfes

Ereignis-Bearbeitung beim play-in (1)



- Reaktion auf ein Ereignis durch Programm:
  - Ereignis wird vom Laufzeitsystem (Ablaufsteuerung, Ereignisverwaltung) erkannt und in ein Ereignisobjekt (`WindowEvent`) umgewandelt

Ereignis-Bearbeitung beim play-in (2)



Der Input-Controller interpretiert beim play-in die Ereignisse, die der Benutzer auf der Oberfläche auslöst und steuert die Operationen des Modells an.

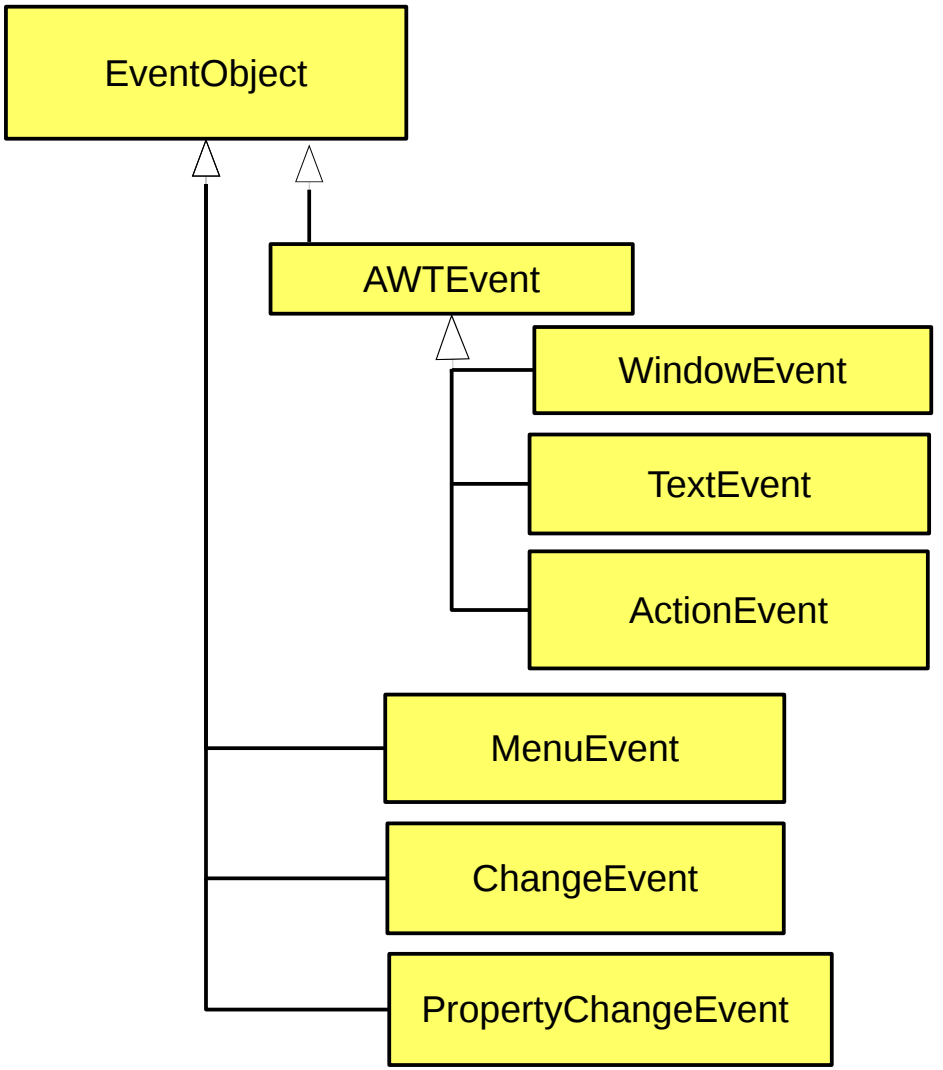
Er stellt eine Steuerungsmaschine dar, der das fachliche Modell ansteuert.

Die Ansteuerung kann geschehen durch

- Aufruf von Methoden der Anwendungsklassen
- Senden von Botschaften mit http an den Server
- Entfernter Aufruf von Methoden in Anwendungsklassen auf dem Server

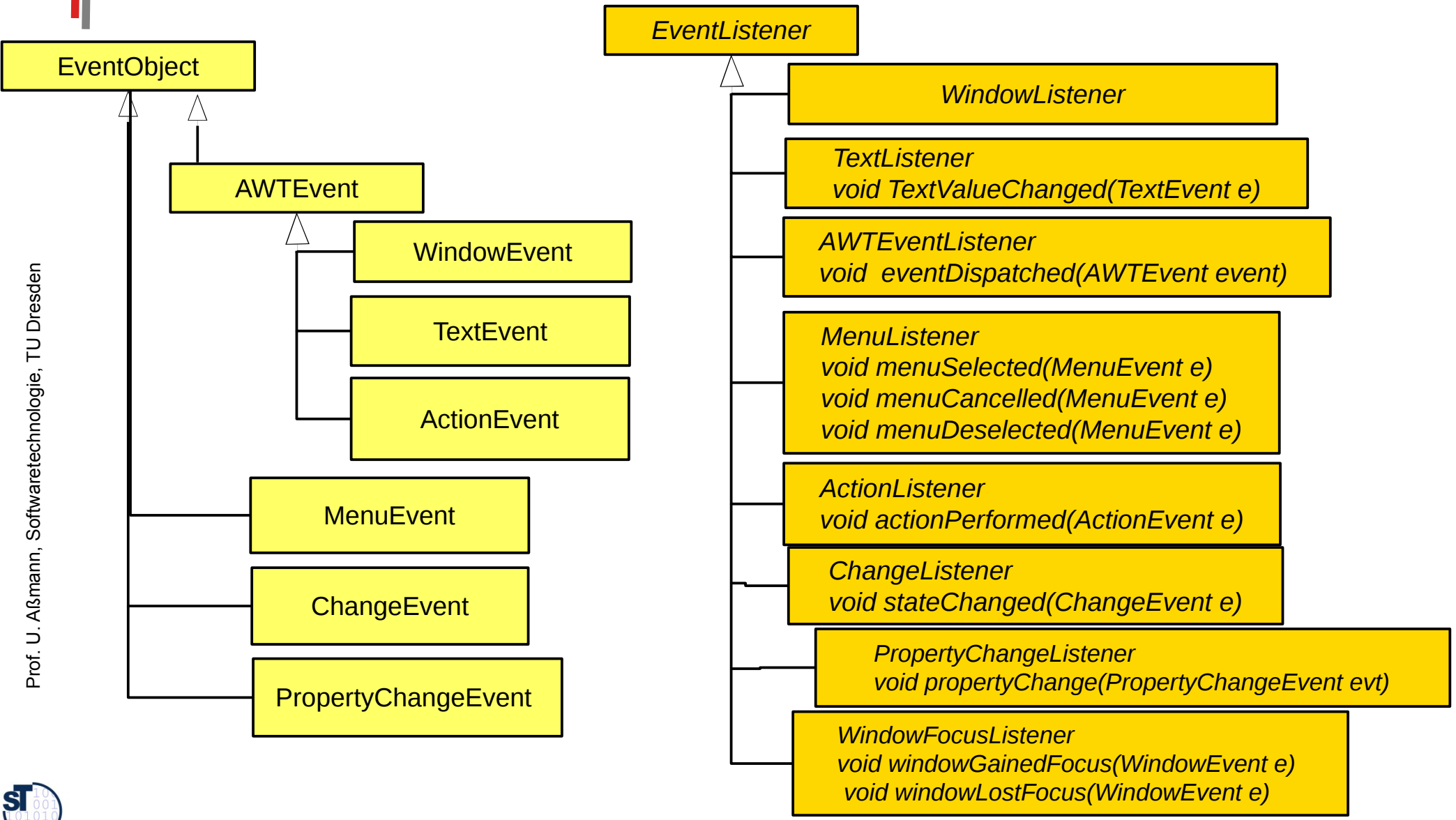
Hierarchie der AWT EventListener (Widget-Listeners) und Event-Objekte

- AWTEvent ist die Klasse der Ereignisobjekte, die ein EventListener (Controller) empfangen kann (push)



Hierarchie der AWT EventListener (Widget-Listeners) und Event-Objekte

- Die Vererbungshierarchien EventListener (für den Controller) und AWTEvent werden parallel variiert (Entwurfsmuster ParallelHierarchies)



java.awt.event.WindowListener für play-in Observer


```
/** empty marker interface, from which all listeners have to
  inherit */
public interface EventListener { };
public interface WindowListener extends EventListener {
  public void windowClosed (WindowEvent ev);
  public void windowOpened (WindowEvent ev);
  public void windowIconified (WindowEvent ev);
  public void windowDeiconified (WindowEvent ev);
  public void windowActivated (WindowEvent ev);
  public void windowDeactivated (WindowEvent ev);
  public void windowClosing (WindowEvent ev);
}
public class WindowEvent extends AWTEvent {
  // Konstruktor, wird vom System aufgerufen
  public WindowEvent (Window source, int id);
  // Abfragen
  public Window getWindow();
}
```

java.awt.event.ActionEvent, ActionListener


```
public class ActionEvent extends AWTEvent {
   ...
   // Konstruktor, wird vom System (Ereignisverwaltung) aufgerufen
  public ActionEvent(Window source, int id, String command);
   // Abfragen (queries)
   public Object getSource ();
   public String getActionCommand();
   ...
}

public interface ActionListener extends EventListener {
  public void actionPerformed (ActionEvent ev);
}
```

70.2.2 Phase 1b) Sehr einfache Input-Controller als Implementierungen von EventListener-Schnittstellen (Play-In)

Wer reagiert denn hier auf Ereignisse?

- Der Input-Controller ist ein Ereignis-Listener
  - Ereignis-Listener sind zunächst Schnittstellen, keine Implementierungsklassen
  - Der Input-Controller hat Ereignis-Listener-Schnittstelle
- Wie programmiert man (einfach) die Klassen, die die abhörenden Schnittstellen implementieren?
  - Eine neue Klasse (Implementierungsklasse)
  - Anhang:
    - Eine Default-Implementierung benutzen (WindowAdapter)
    - Eine Unterklasse der Default-Implementierung WindowAdapter
    - Eine innere Klasse
    - Eine anonyme Klasse

a) Implementierungsklasse WindowCloser für Ereignis "Schließen" aus WindowListener


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

class WindowCloser implements WindowListener {
  // Reagiert nur auf Schließen
  public void windowClosed (WindowEvent ev) {}
  public void windowOpened (WindowEvent ev) {}
  public void windowIconified (WindowEvent ev) {}
  public void windowDeiconified (WindowEvent ev) {}
  public void windowActivated (WindowEvent ev) {}
  public void windowDeactivated (WindowEvent ev) {}

  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}
```

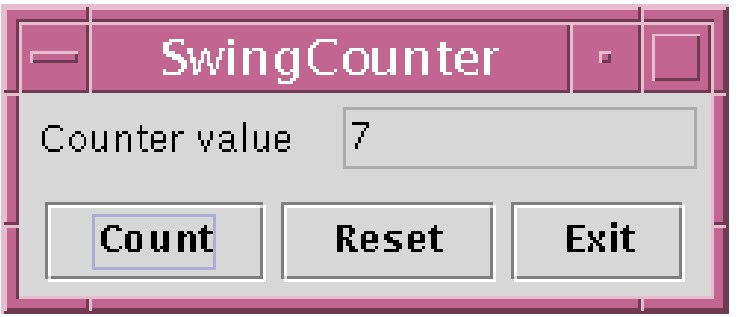
Hauptprogramm für schließbares Fenster


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

class WindowCloser implements WindowListener {
  ... siehe vorige Folie ...
}

class ExampleFrame extends Frame {

  public ExampleFrame () {
     setTitle("untitled");
     setSize(150, 50);
     addWindowListener(new WindowCloser());
     setVisible(true);
   }
}

class GUI2 {
  public static void main (String[] argv) {
    ExampleFrame f = new ExampleFrame();}
}
```

70.2.3 Phase 1c) Hierarchischer Aufbau der Benutzungsoberfläche (Widget-Hierarchie) mit Swing

Graphische Benutzungsoberflächen
Graphical User Interfaces (GUI)

- 1980: Smalltalk-80-Oberfläche (Xerox)
- 1983/84: Lisa/Macintosh-Oberfläche (Apple)
- 1988: NextStep (Next)
- 1989: OpenLook (Sun)
- 1989: Motif (Open Software Foundation)
- 1987/91: OS/2 Presentation Manager (IBM)
- 1990: Windows 3.0 (Microsoft)
- 1995-2001: Windows 95/NT/98/2000/ME/XP (Microsoft)
- 1995: **Java AWT** (SunSoft)
- 1997: **Swing** Components for Java (SunSoft)
- 2002: SWT von Eclipse
- 2006: XAML, Silverlight (Microsoft)
  - Xswt, xswing, XUL (Mozilla) etc.



Hier: Expliziter Aufbau mit AWT und Swing

- Abstract Window Toolkit (AWT):
  - Umfangreiche Bibliothek von Oberflächen-Bausteinen
  - Plattformunabhängige Schnittstellen, aber grosse Teile plattformspezifisch realisiert ("native code")
- Swing-Bibliotheken
  - Erweiterung von AWT
  - Noch umfangreichere Bibliothek von Oberfächen-Bausteinen
  - Plattformunabhängiger Code
    (d.h. Swing ist weitestgehend selbst in Java realisiert)
  - Wesentlich größerer Funktionsumfang
    (nicht auf den "kleinsten Nenner" der Plattformen festgelegt)

Bibliotheken von AWT und Swing

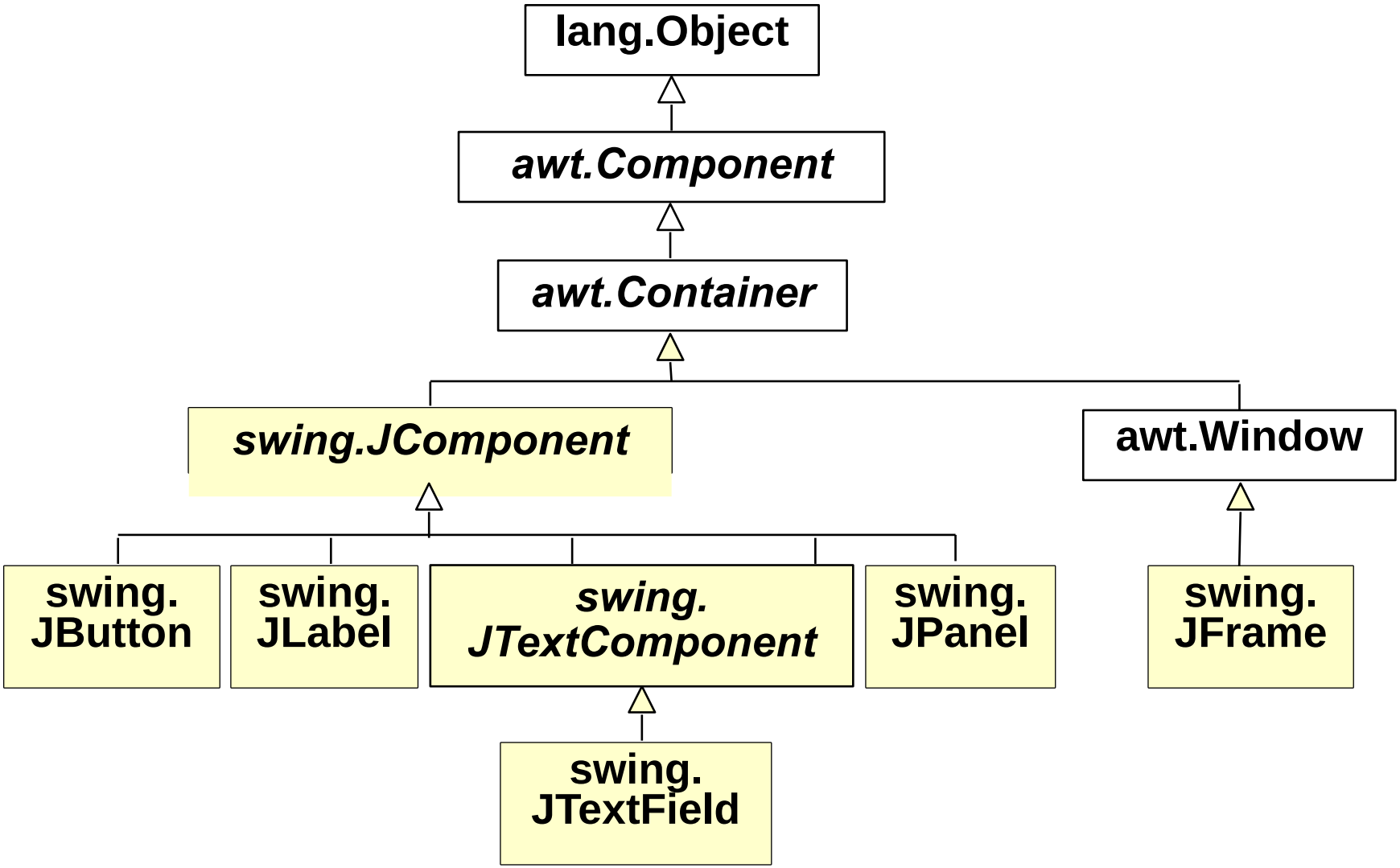
- Wichtigste AWT-Pakete:
  - **java.awt**: u.a. Grafik, Oberflächenkomponenten, Layout-Manager
  - **java.awt.event**: Ereignisbehandlung
  - Andere Pakete für weitere Spezialzwecke
- Wichtigstes Swing-Paket:
  - **javax.swing**: Oberflächenkomponenten
  - Andere Pakete für Spezialzwecke
  - Viele AWT-Klassen werden auch in Swing verwendet!
- Standard-Import-Vorspann:

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

- (Naiver) Unterschied zwischen AWT- und Swing-Komponenten:
  - AWT: Button, Frame, Menu, ...
  - Swing: JButton, JFrame, JMenu, ...

AWT/Swing-Klassenhierarchie (Ausschnitt)

- Dies ist nur ein sehr kleiner Ausschnitt
  - Präfixe "java." und "javax." hier weggelassen.



Component, Container, Window, Frame, Panel

- ***awt.Component*** (abstrakt):
  - Oberklasse aller Bestandteile der Oberfläche

```
public void setSize (int width, int height);
public void setVisible (boolean b);
```

- ***awt.Container*** (abstrakt):
  - Oberklasse aller Komponenten, die andere Komponenten enthalten

```
public void add (Component comp);
public void setLayout (LayoutManager mgr);
```

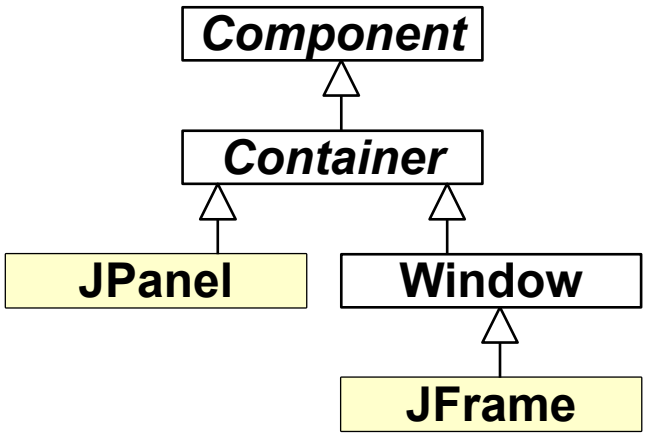
- **awt.Window**
  - Fenster ohne Rahmen oder Menüs

```
public void pack (); //Größe anpassen
```

- **swing.JFrame**
  - Größenveränderbares Fenster mit Titel

```
public void setTitle (String title);
```

- **swing.JPanel**
  - Zusammenfassung von Swing-Komponenten



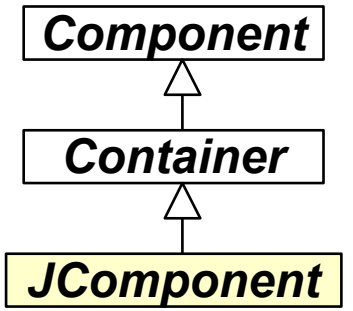
JComponent

- Oberklasse aller in der Swing-Bibliothek neu implementierten, verbesserten Oberflächenkomponenten. Eigenschaften u.a.:
  - Einstellbares "Look-and-Feel"
  - Komponenten kombinierbar und erweiterbar
  - Rahmen für Komponenten

    ```
    void setBorder (Border border);
    ```

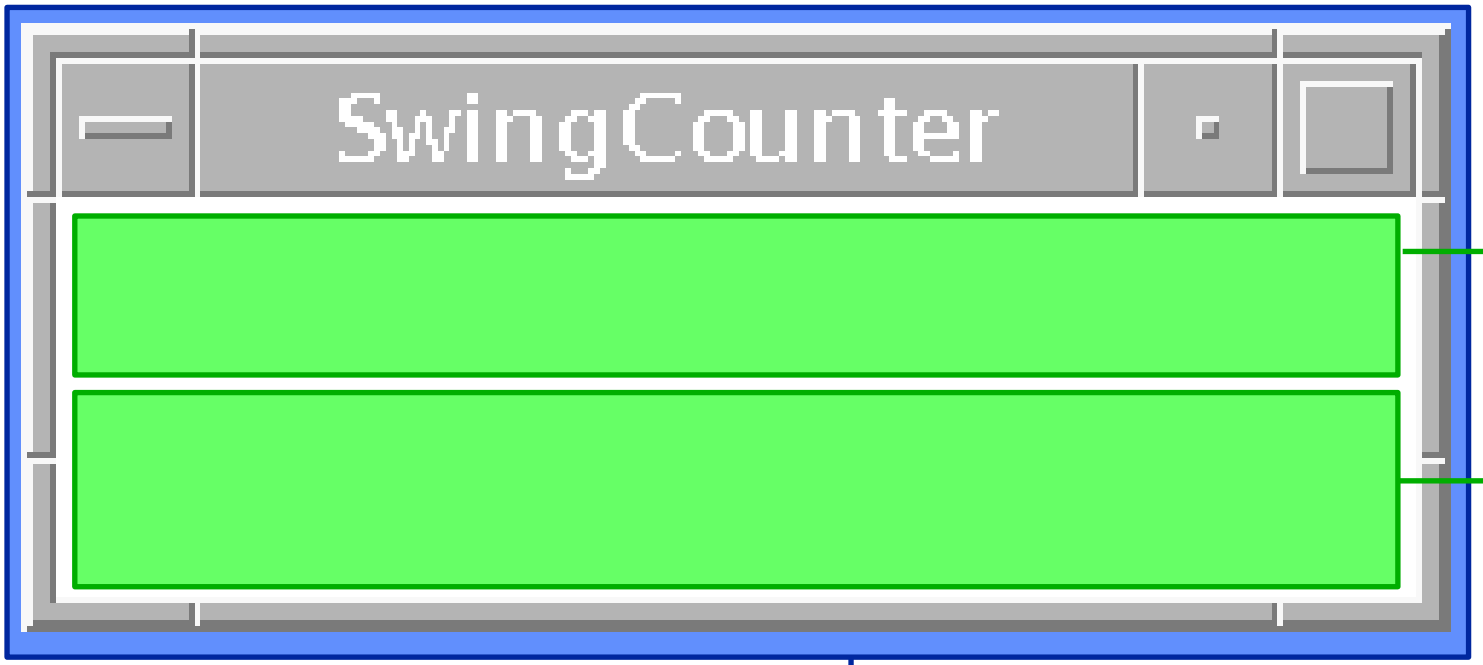
    (Border-Objekte mit **`BorderFactory`** erzeugbar)
  - ToolTips -- Kurzbeschreibungen, die auftauchen, wenn der Cursor über der Komponente liegt

    ```
    void setToolTipText (String text);
    ```
  - Automatisches Scrolling



- Beispiele für weitere Unterklassen von JComponent:
  - JList: Auswahlliste
  - JComboBox: "Drop-Down"-Auswahlliste mit Texteingabemöglichkeit
  - JPopUpMenu: "Pop-Up"-Menü
  - JFileChooser: Dateiauswahl

Zähler-Beispiel: Grobentwurf der Oberfläche



JPanel
valuePanel

JPanel
buttonPanel

CounterFrame cf
abgeleitet von JFrame

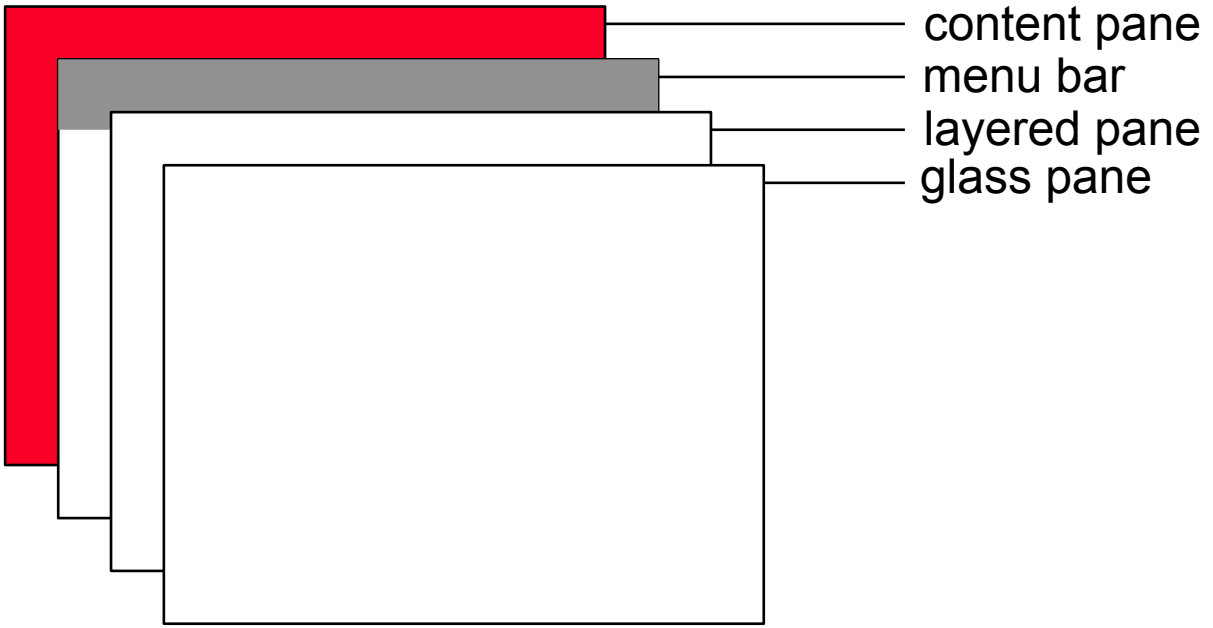
Die Sicht *(View)* mit Swing: Gliederung, 1. Versuch


```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");

  public CounterFrame (Counter c) {
    setTitle("SwingCounter");
    valuePanel.add(new JLabel("Counter value"));
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    // .. value panel hinzufügen
    getContentPane().add(valuePanel, BorderLayout.NORTH);
    buttonPanel.add(countButton);
    buttonPanel.add(resetButton);
    buttonPanel.add(exitButton);
    // .. button panel hinzufügen
    getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    pack();
    setVisible(true);
  }
}
```

Subjekt starten (Ereignisverwaltung)

Hinzufügen von Komponenten zu JFrames

- Ein JFrame ist ein "Container", d.h. dient zur Aufnahme weiterer Elemente.
- Ein JFrame ist intern in verschiedene "Scheiben" *(panes)* organisiert. Die wichtigste ist die *content pane*.



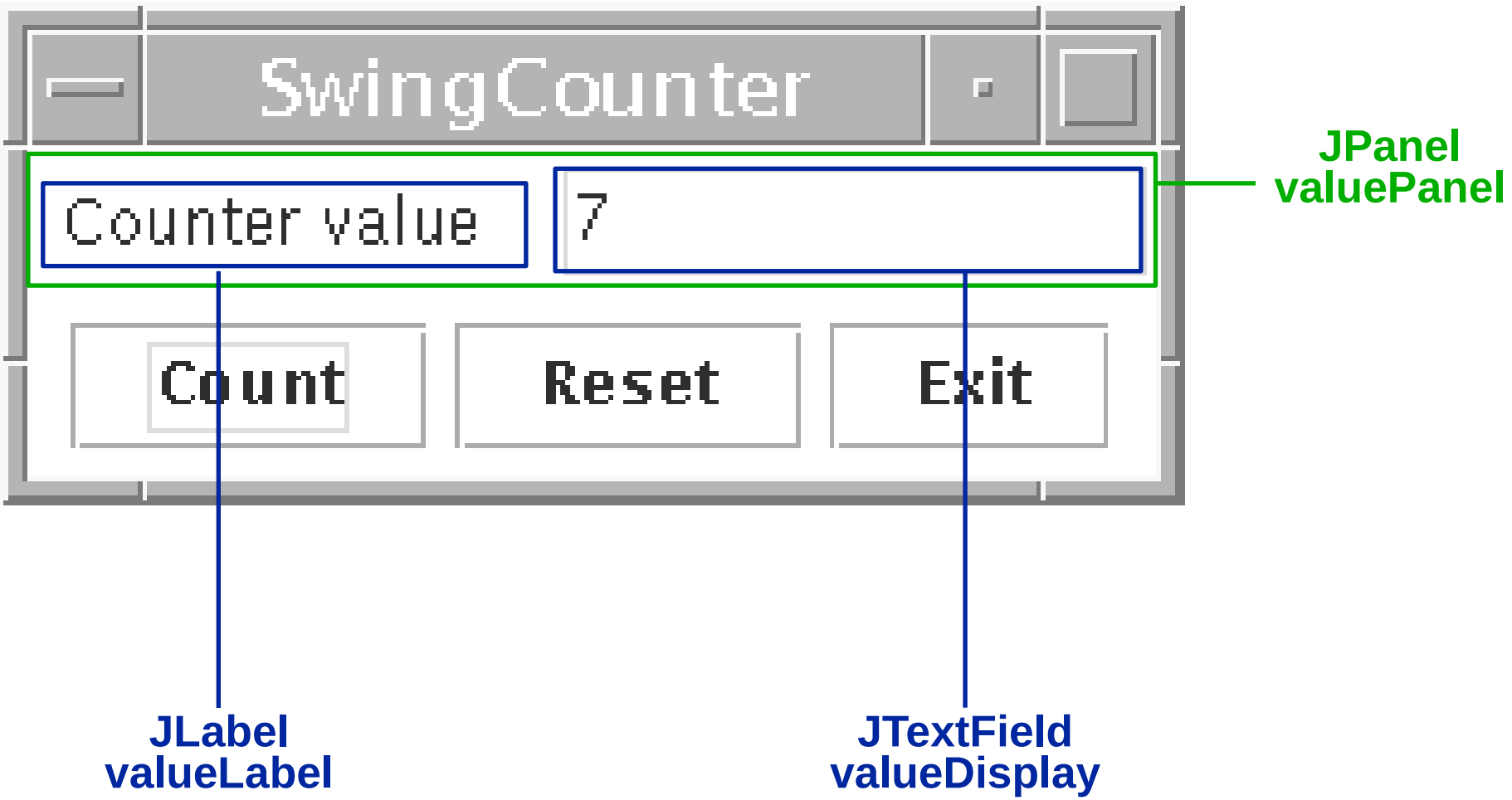
- In JFrame ist definiert:

```
Container getContentPane();
```

Die Sicht *(View):* Gliederung, 2. Versuch


```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();

  JPanel buttonPanel = new JPanel();




  public CounterFrame (Counter c) {
    setTitle("SwingCounter");



    getContentPane().add(valuePanel);



    getContentPane().add(buttonPanel);
    pack();
    setVisible(true);
  }
}
```

Zähler-Beispiel: Entwurf der Wertanzeige



TextComponent, TextField, Label, Button

- ***JTextComponent:***
  - Oberklasse von JTextField und JTextArea

```
public void setText (String t);
public String getText ();
public void setEditable (boolean b);
```

- **JTextField:**
  - Textfeld mit einer Zeile

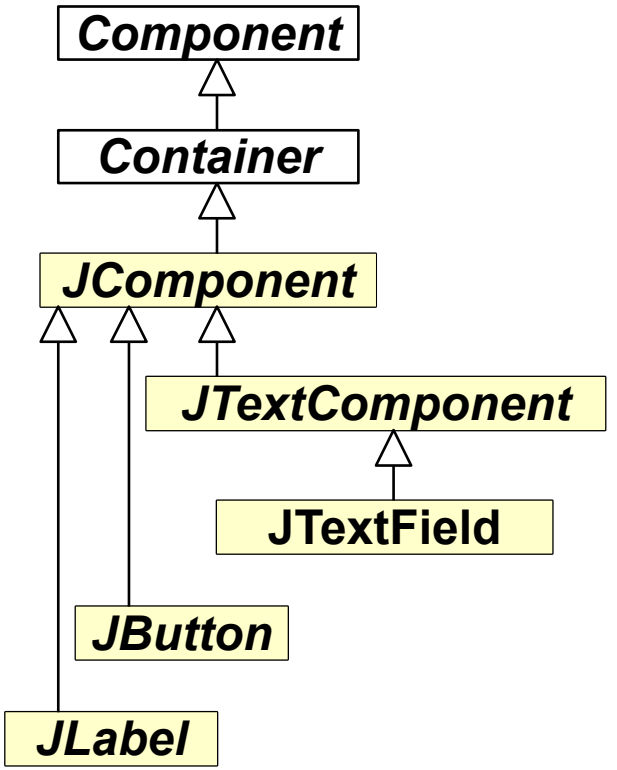
```
public JTextField (int length);
```

- **JLabel:**
  - Einzeiliger unveränderbarer Text

```
public JLabel (String text);
```

- **JButton:**
  - Druckknopf mit Textbeschriftung

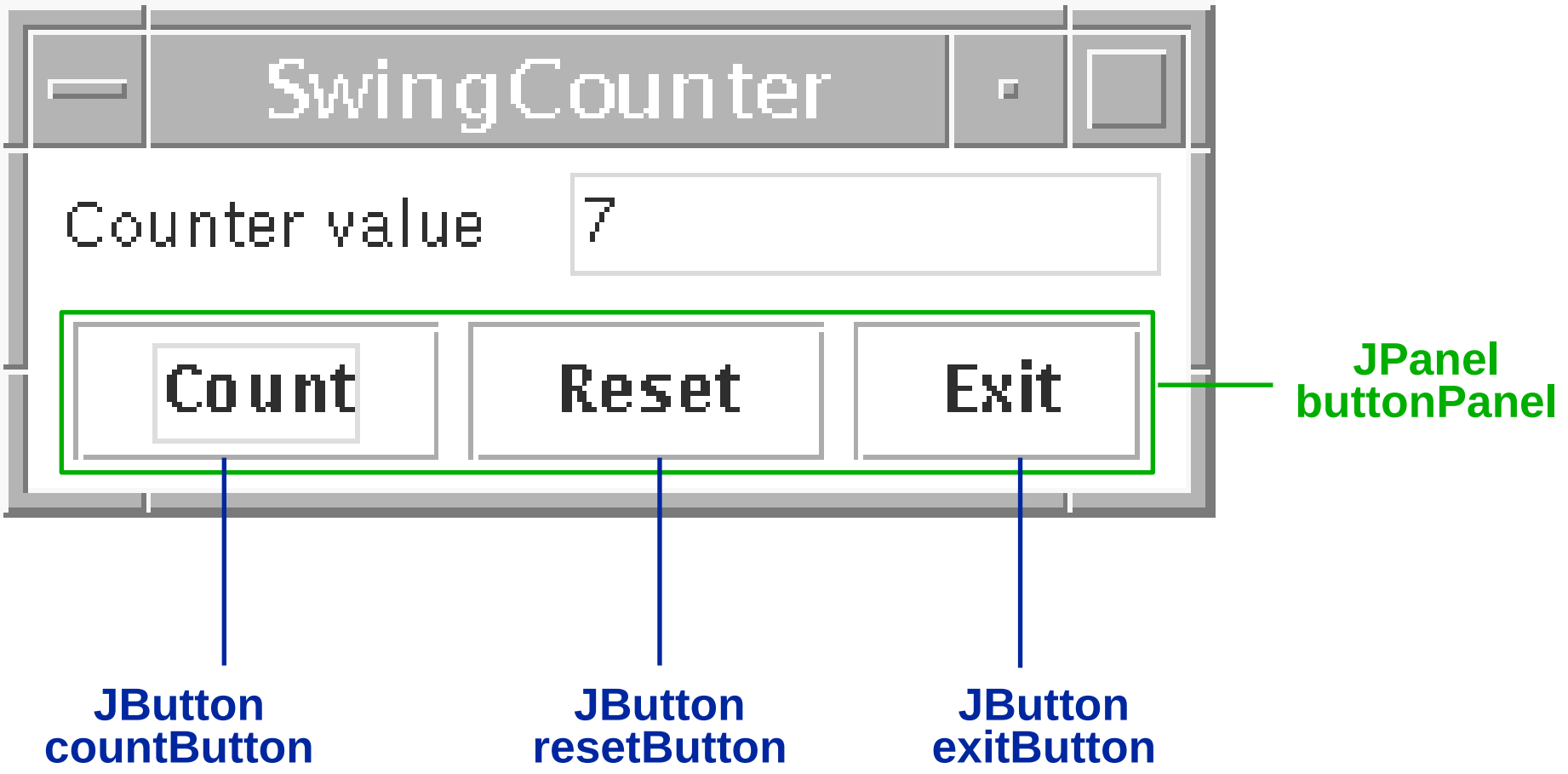
```
public JButton (String label);
```

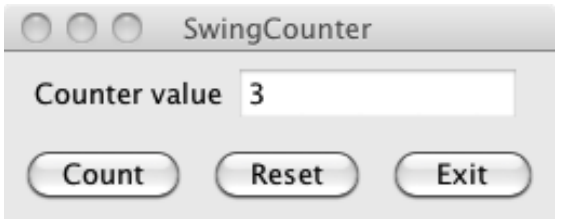


Die Sicht *(View):* Elemente der Wertanzeige


```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();


  public CounterFrame (Counter c) {
    setTitle("SwingCounter");
    valuePanel.add(new JLabel("Counter value"));
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);

    getContentPane().add(valuePanel);



    getContentPane().add(buttonPanel);
    pack();
    setVisible(true);
  }
}
```

Zähler-Beispiel: Entwurf der Bedienelemente der Sicht





Der Aufbau der Sicht *(Swing View)*: Bedienelemente


```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");

  public CounterFrame (Counter c) {
    setTitle("SwingCounter");
    valuePanel.add(new JLabel("Counter value"));
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    getContentPane().add(valuePanel);
    buttonPanel.add(countButton);
    buttonPanel.add(resetButton);
    buttonPanel.add(exitButton);

    getContentPane().add(buttonPanel);
    pack();
    setVisible(true);
  }
}
```

70.3 Phase 2) Verdrahtung GUI und Anwendungslogik mit MVC

mit starker Schichung und aktivem Controller

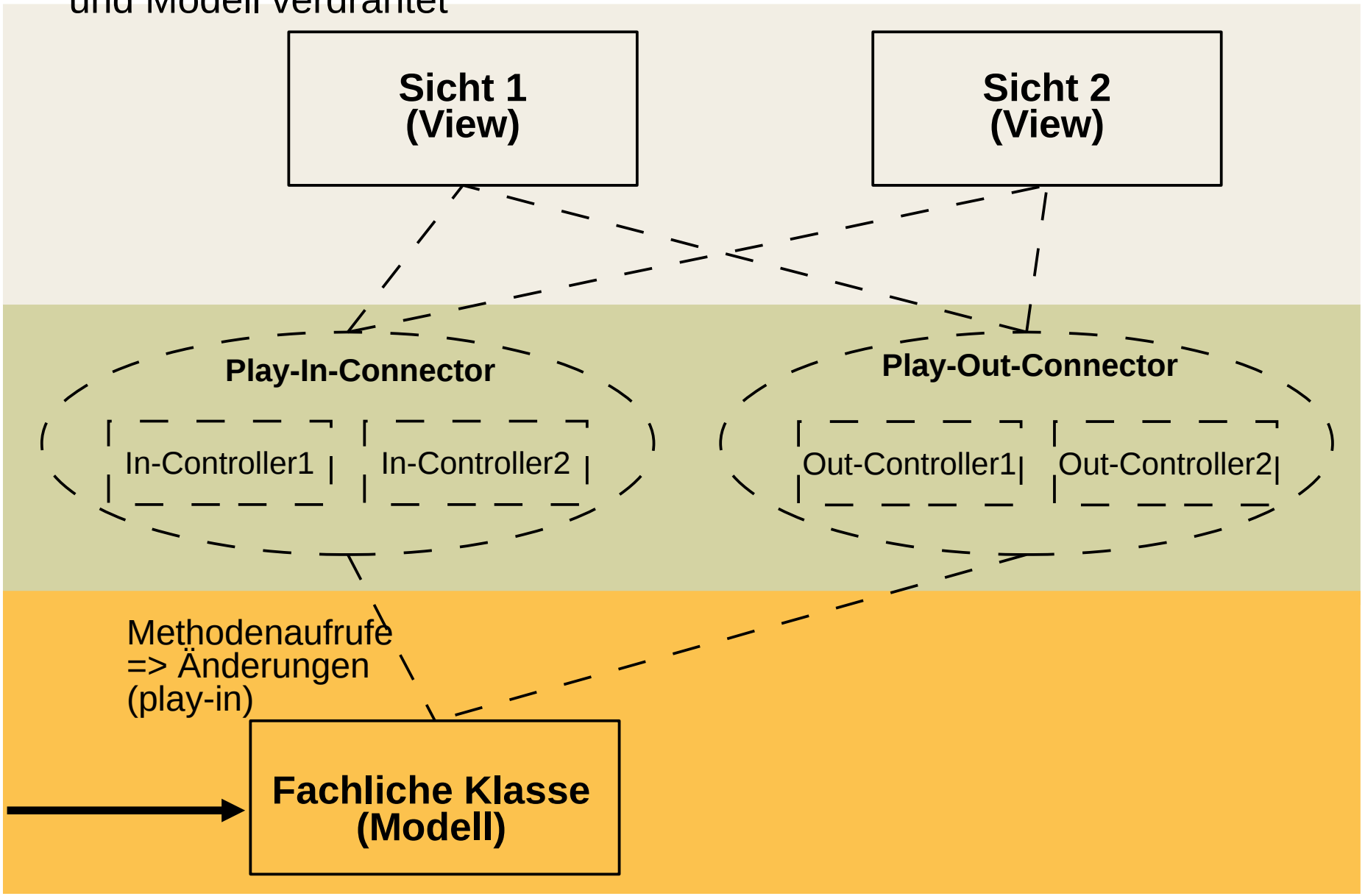
- Die folgenden Beispiele nutzen die Präfixe:
- UI_ : gehört zur UI-Schicht
- C_ : gehört zum Controllerschicht
- A_ : gehört zur Anwendungslogik

70.3.1 Architekturvarianten: Strikte, semi-strikte oder schwache Schichtung des Play-Out

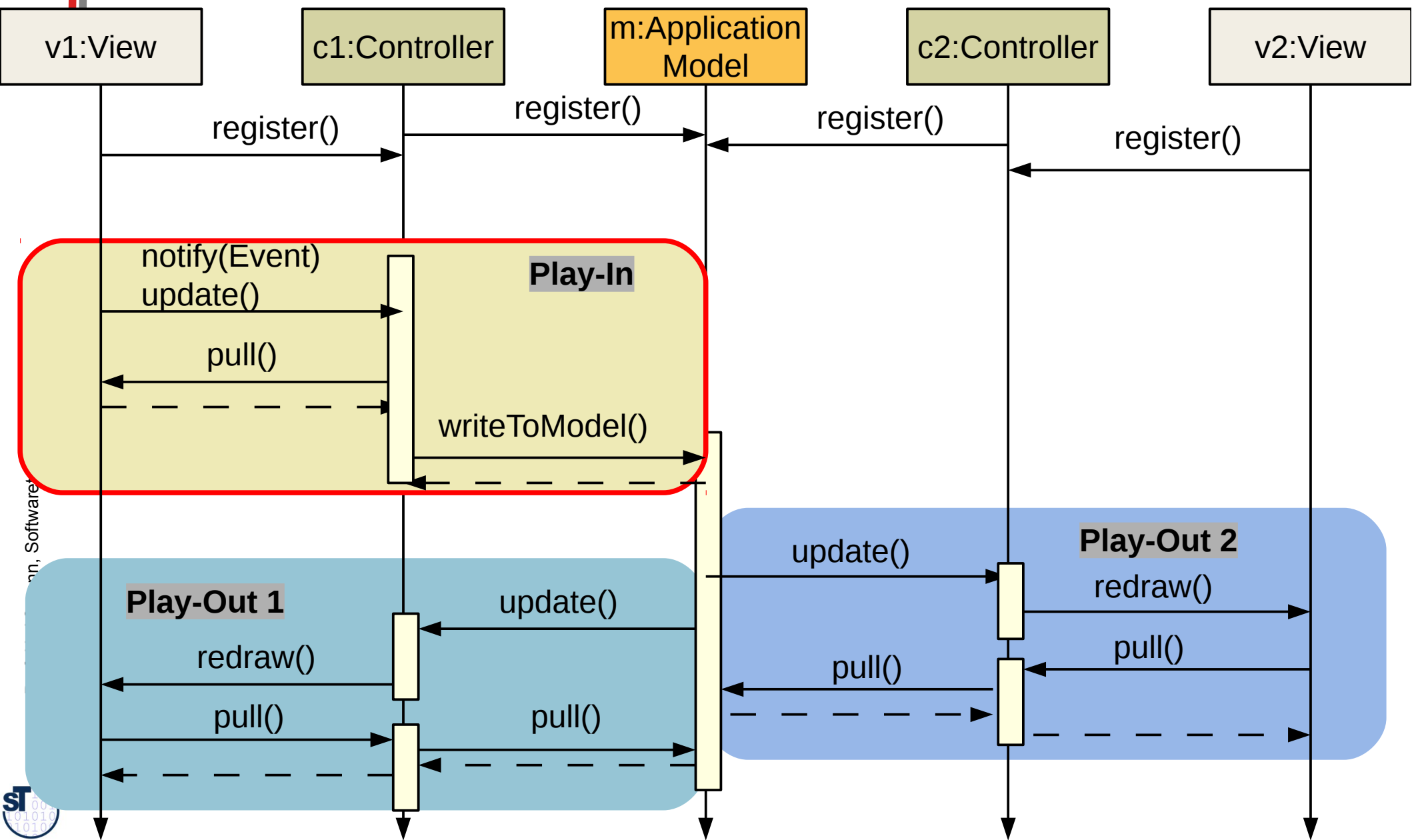
- Controller für IN oder auch für OUT

Controller sind Konnektoren zwischen Model und View

- Im Folgenden gibt es ein Hauptobjekt, den Konnektor, der View, Controller und Modell verdrahtet

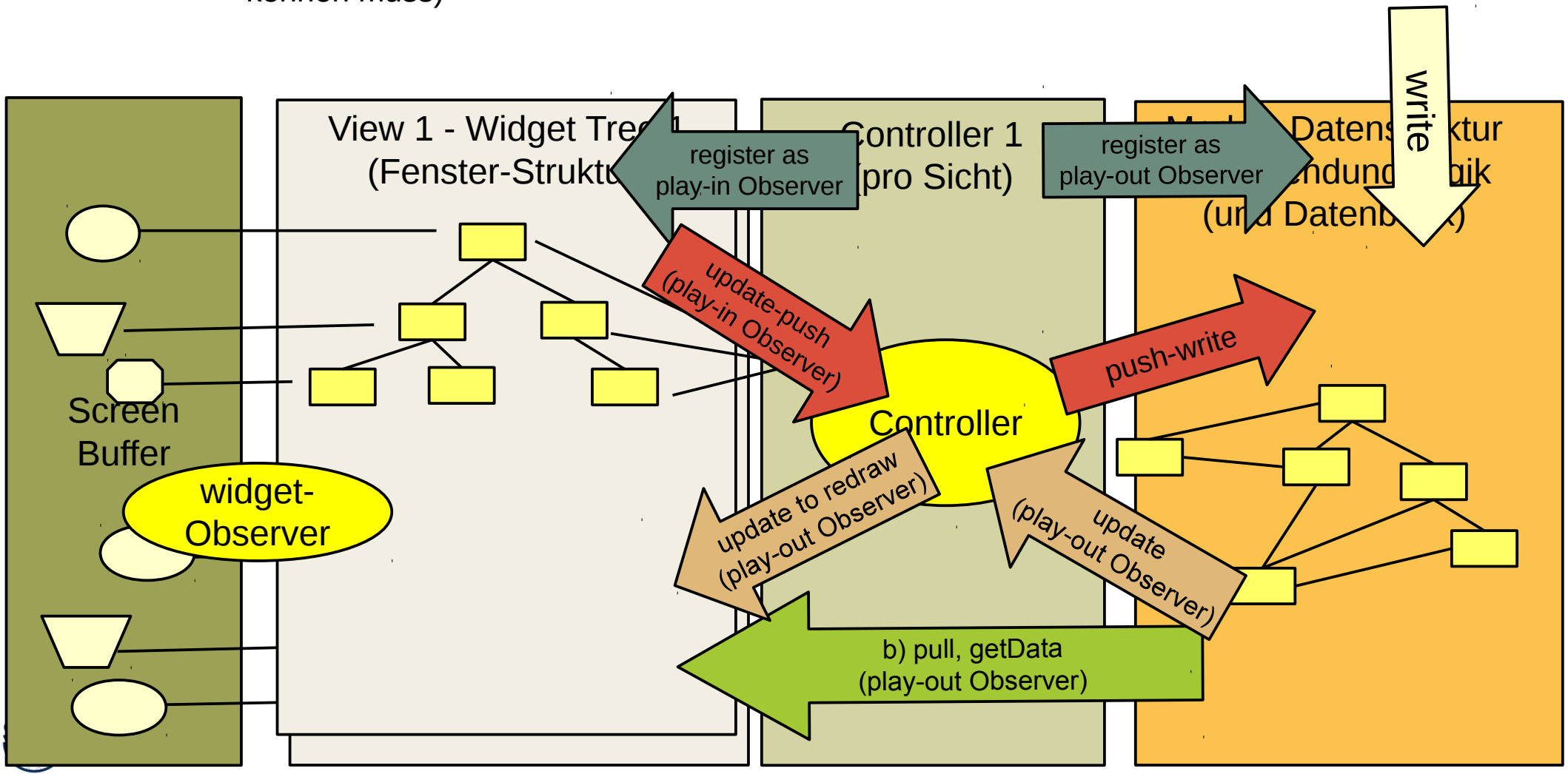


Wdh.: Strikte Schichtung, indirekte Kommunikation
Play-In mit passivem View und pull-In-Controller;
Passives Play-Out mit indirektem pull-Out-View



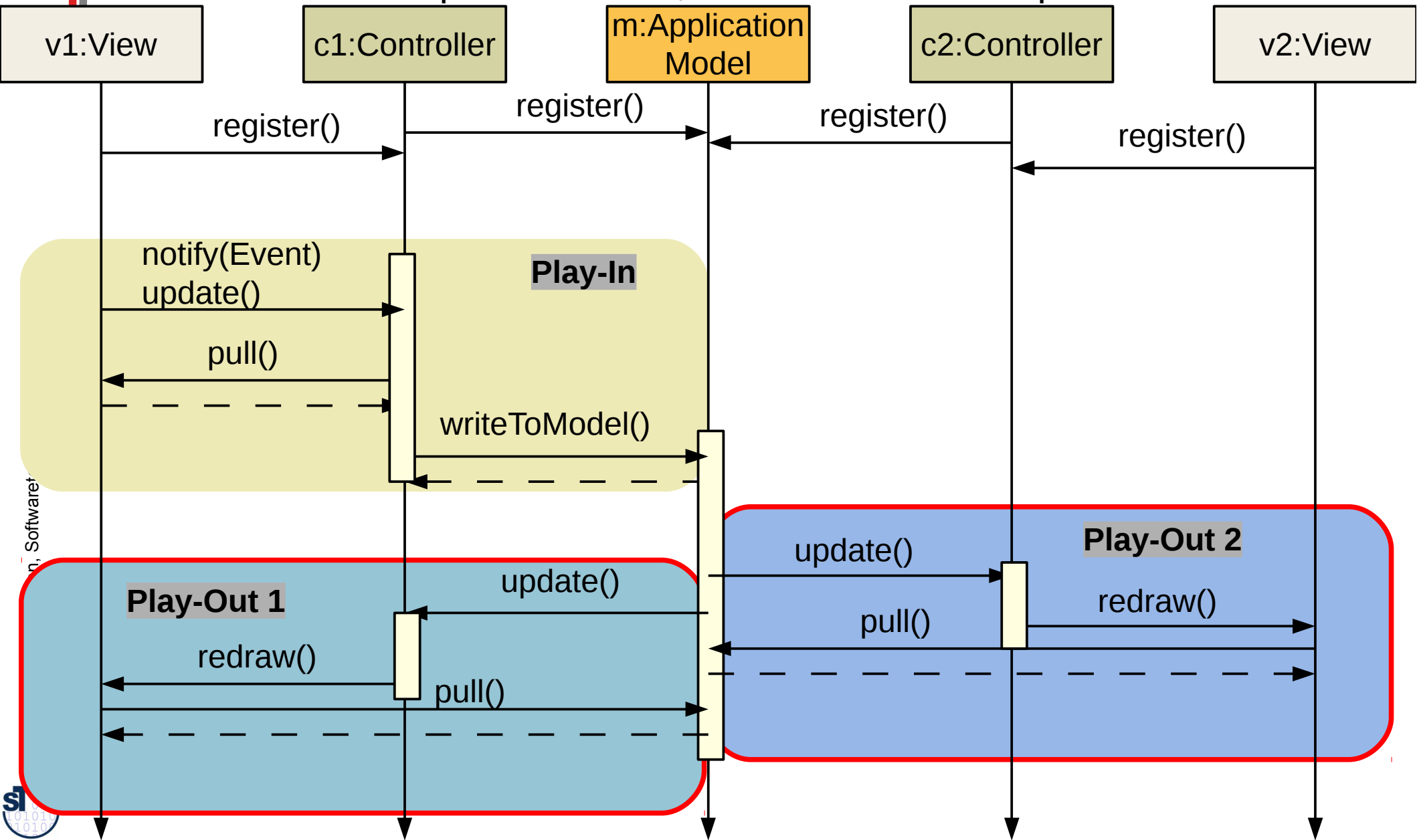
MVC Dynamik (Semi-striktes Play-Out)

- Semi-striktes Schichtung des Play-Out erlaubt dem View, das Model zu sehen (weniger modular, mehr Abhängigkeiten)
  - View wird *indirekt* benachrichtigt (indirekter Update)
  - View greift beim pull *direkt* auf das Modell zu (stärker gekoppelt, da das View das Modell kennen muss)



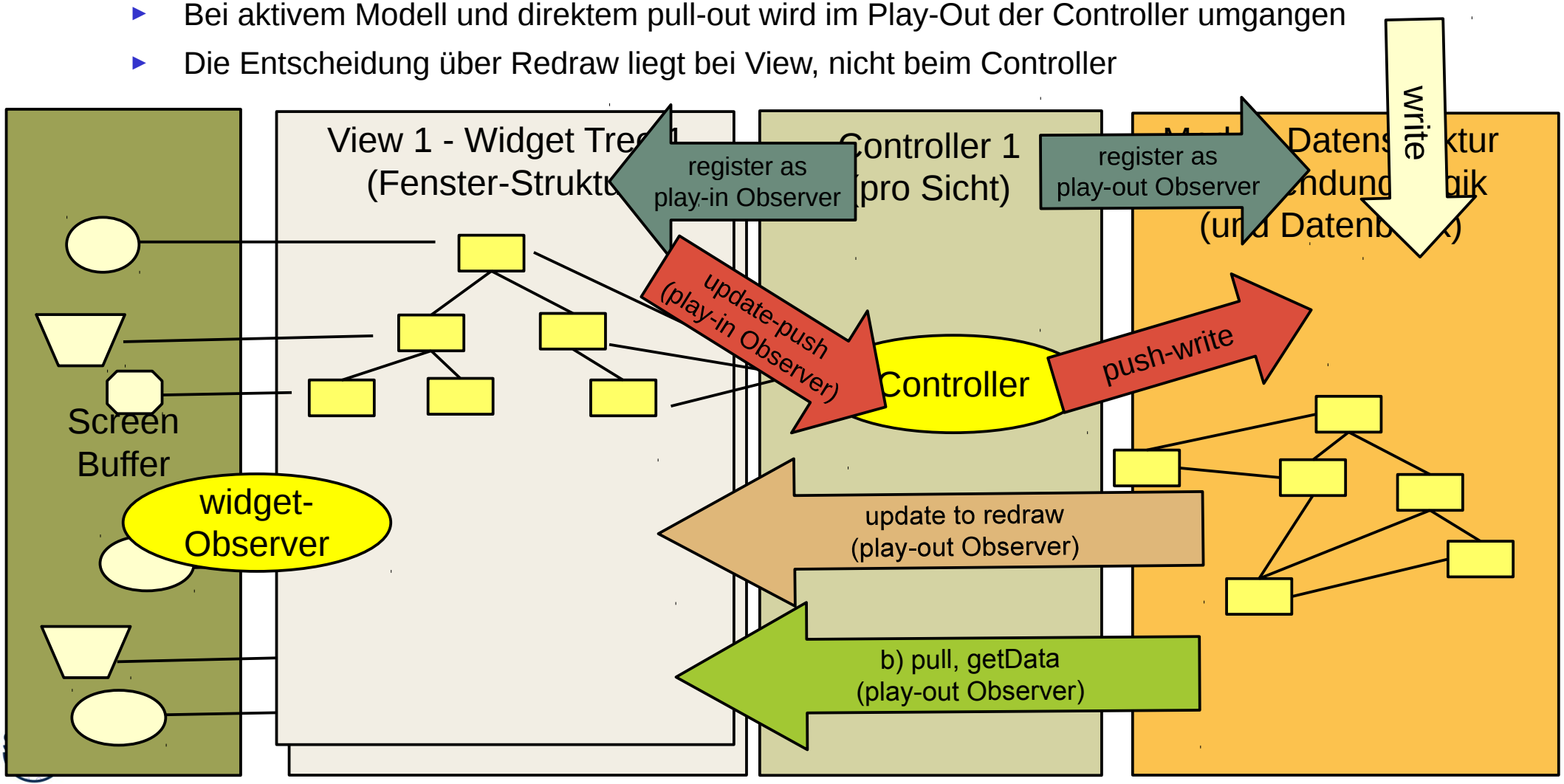
Semi-strikte Schichtung des Play-Out
Play-In mit pull-In-Controller;
Passives Play-Out mit direktem pull-Out-View

- Mehrere Views pro Controller, mehrere Controller pro Model



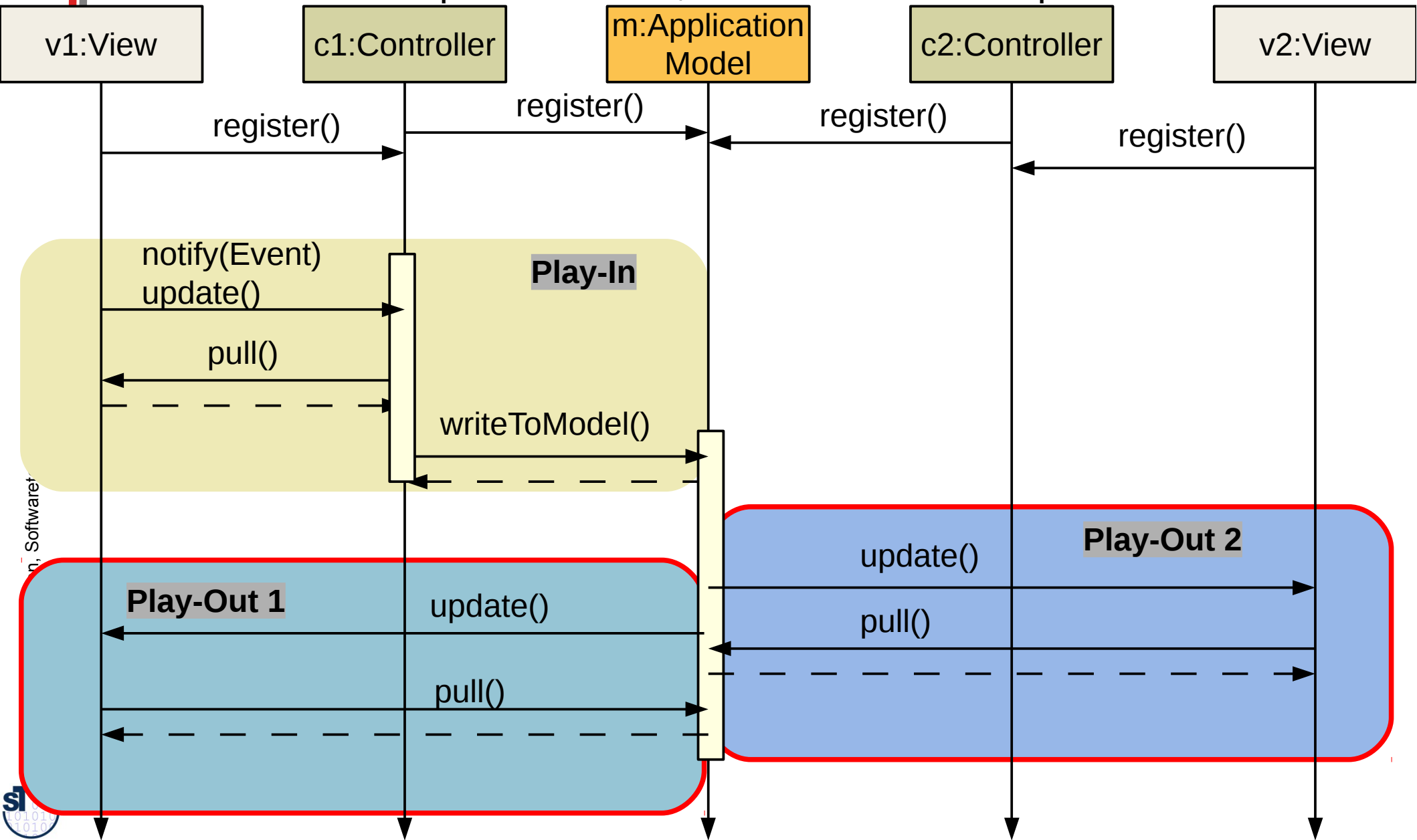
MVC Dynamik (Schwache Schichtung des Play-Out)

- Play-Out-Observer mit direkter Kopplung von Model und View
  - Modell ist aktiv, benachrichtigt View *direkt*
  - Widget greift mit pull *direkt* auf das Modell zu (stärker gekoppelt, da das View das Modell kennen muss)
- Bei aktivem Modell und direktem pull-out wird im Play-Out der Controller umgangen
- Die Entscheidung über Redraw liegt bei View, nicht beim Controller



Schwache Schichtung des Play-Out
Play-In mit pull-In-Controller;
Passives Play-Out mit direktem pull-Out-View

- Mehrere Views pro Controller, mehrere Controller pro Model



Varianten von MVC

Architekturprinzip	Strikte Schichtung (strictly passive model)	Semi-strikte Schichtung (weakly passive model)	Schwache Schichtung
Wer benachrichtigt, dass das Modell geändert ist? (update)	indirekter Update via Controller (passive model)	indirekter Update Modell → Controller → Observer	direkter Update Modell → View
Wer transportiert die Daten des geänderten Zustandes?	indirekter pull-out View → Observer → Model	direkter pull-out View → Model	direkter pull-out View → Model

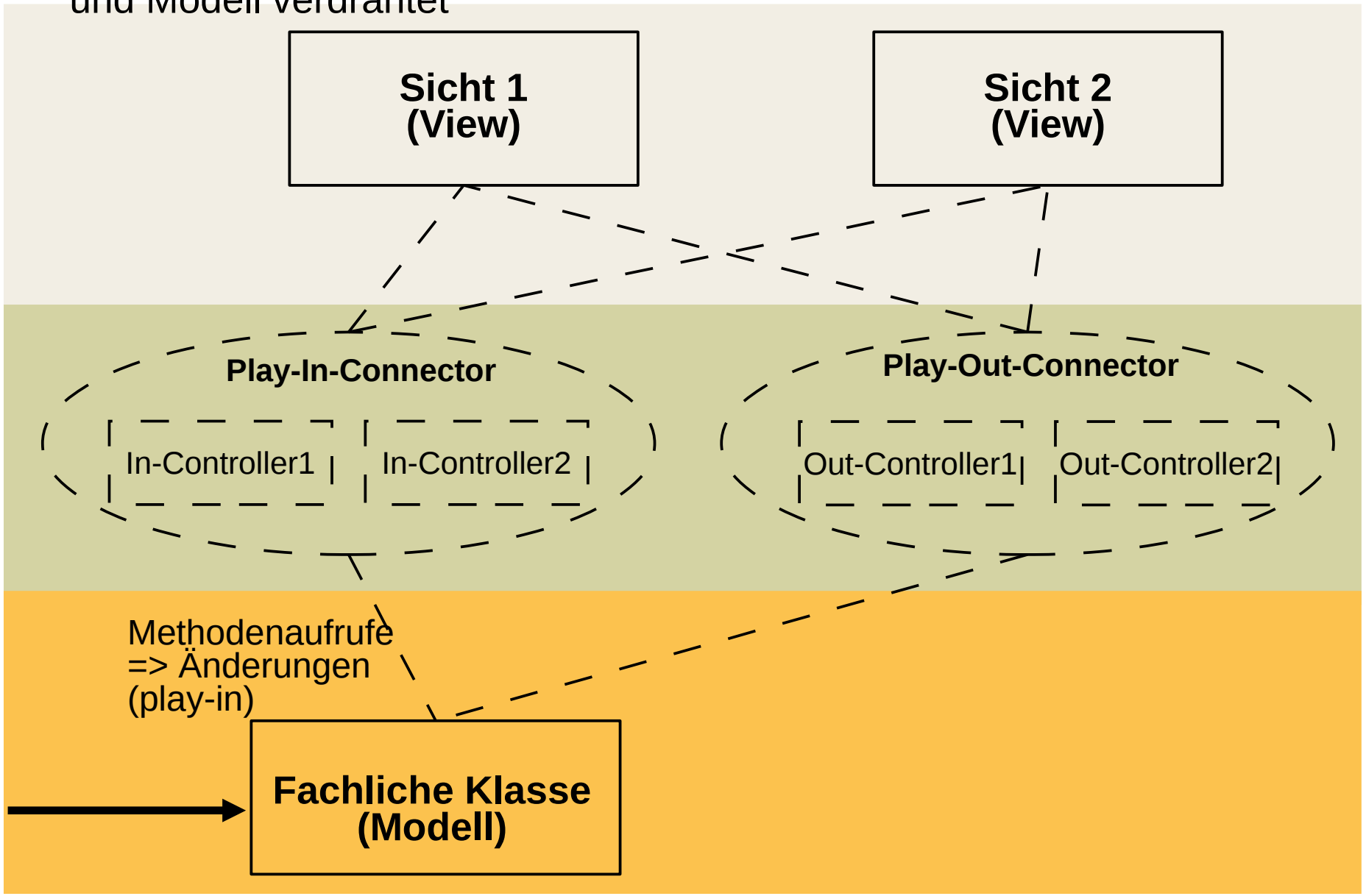
70.3.2 Beispiel für Phase 2) Verdrahtung GUI und Anwendungslogik mit Modularem MVC

mit schwacher Schichung und aktivem IN-Controller

- Die folgenden Beispiele nutzen die Präfixe:
- UI_ : gehört zur UI-Schicht
- C_ : gehört zum Controllerschicht
- A_ : gehört zur Anwendungslogik

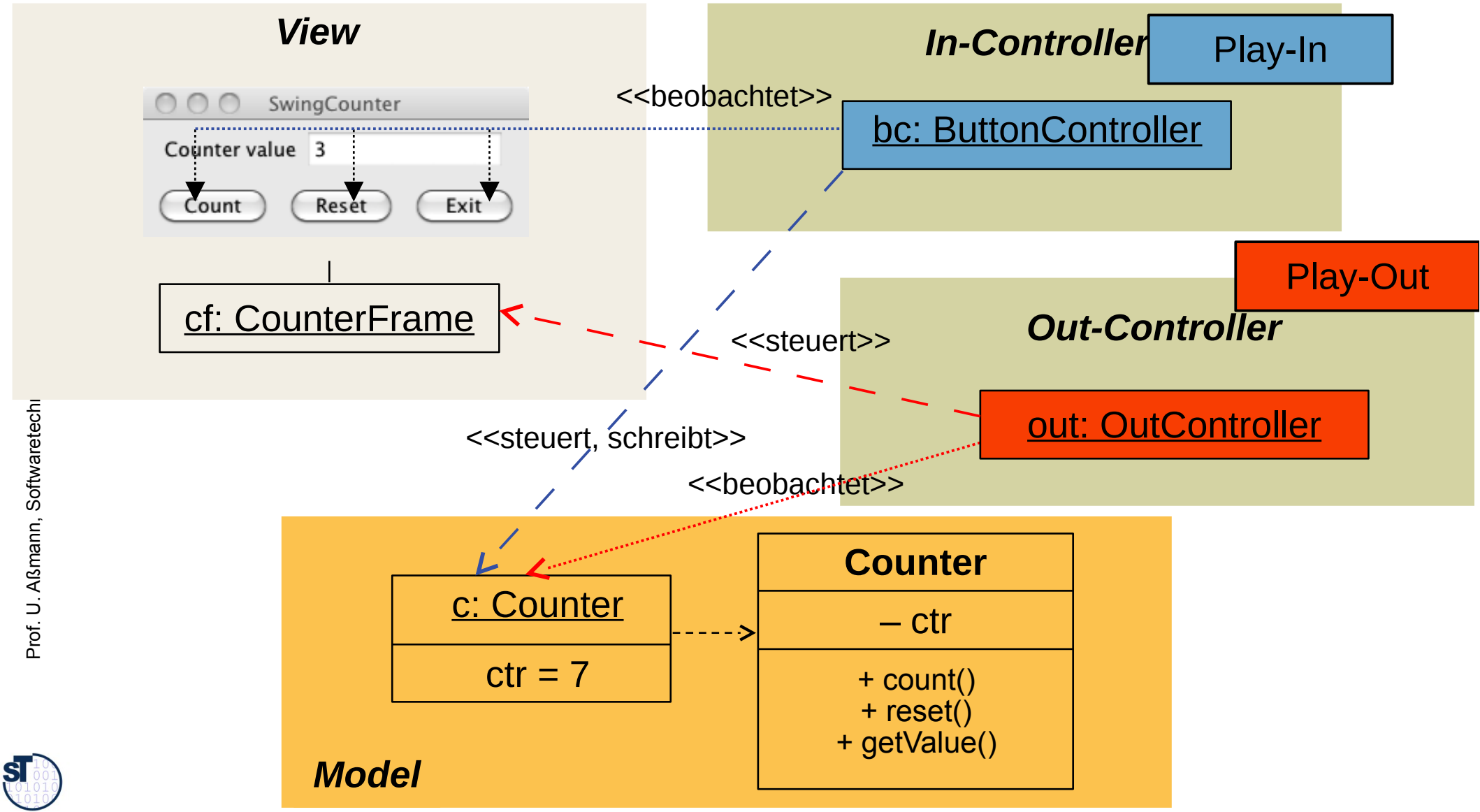
Controller sind Konnektoren zwischen Model und View

- Im Folgenden gibt es ein Hauptobjekt, den Konnektor, der View, Controller und Modell verdrahtet



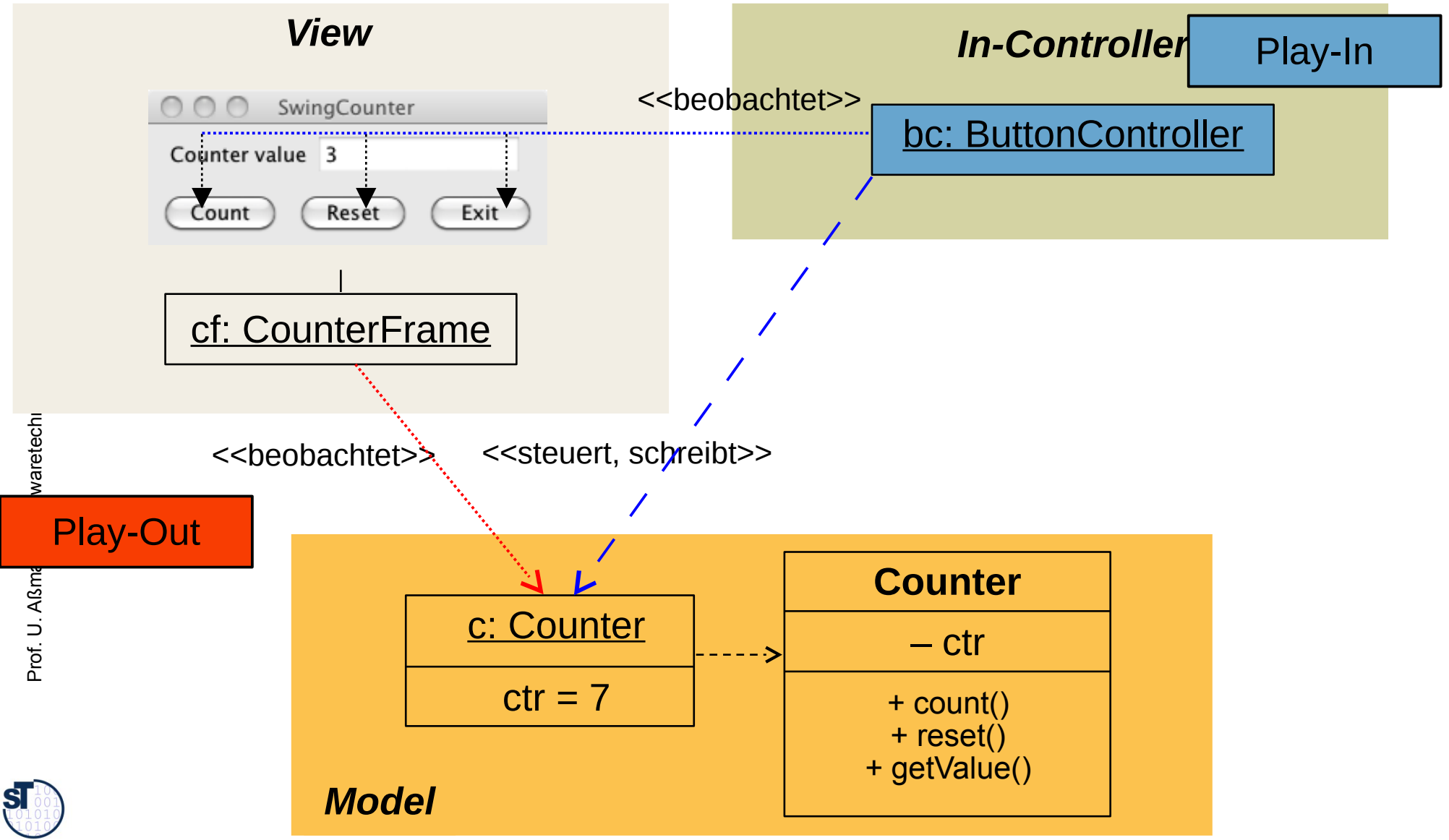
Geschichtete Model-View-Controller-Architektur

- GUI-Reagiert-Auf-Modelländerungen mit Out-Controller (push-Observer, Play-Out)
- Modell-Reagiert-Auf-View mit In-Controller (push-Observer, Play-In)



Semi-strikte Model-View-Controller-Architektur

- Modell-Reagiert-Auf-View mit In-Controller (push-Observer, Play-In)
- GUI-Reagiert-Auf-Modelländerungen: direkt (push-Observer, Play-Out)



Beispiel „Counter“ für das Play-In: Anbindung View an Model


```
class UI_CounterFrame extends JFrame implements Observer {
  JTextField valueDisplay = new JTextField(10);
  /** Assembling the widget tree of the counter is
     separate from linking it to the Controller */
  public UI_CounterFrame (int i) {
        setTitle("SwingCounter");
        valuePanel.add(new JLabel("Counter value"));
        valuePanel.add(valueDisplay);
        valueDisplay.setEditable(false);
        valueDisplay.setText(String.valueOf(i));
        getContentPane().add(valuePanel,BorderLayout.NORTH);

        buttonPanel.add(countButton);
        buttonPanel.add(resetButton);
        buttonPanel.add(exitButton);
        getContentPane().add(buttonPanel,BorderLayout.SOUTH);
        /* Show the widget tree on the screen */
        pack();
        setVisible(true);
  }
  /** Separate: Linking the private buttons in the widget tree
      to the C_ButtonInputController.      */
  public void linkButtonsToInputController(
    C_ButtonInputController bic) {
     countButton.addActionListener(bic);
     resetButton.addActionListener(bic);
    exitButton.addActionListener(bic);
  }
```

Connector: Anbindung aller Schichten


```
/** The Connector class connects UI, A, and the in- and
 * out-controllers (C) (wiring). */
class ControlConnector {
    /** Linking the window in the widget tree to the
     C_WindowInputController. */
    public void linkWindowInputController(UI_CounterFrame cf,
       C_WindowInputController wic) {
        // IN Observer for Windows
        cf.addWindowListener(wic);
    }
    /** This links the buttons in the widget tree to the
     * C_ButtonInputController. The layout of the widget tree
     * is private to UI_CounterFrame; therefore, we delegate
     * to UI_CounterFrame.linkButtonToInputController.*/
    public void linkButtonInputController(UI_CounterFrame cf,
    C_ButtonInputController bic) {
        cf.linkButtonsToInputController(bic);
    }
    // Linking model and input controller
    public void linkInputControllerToModel(C_ButtonInputController bic,
      A_Counter counter) {
        bic.setModel(counter);
    }
    // OUT-Observer: link Model and View directly (weak layering)
    public void linkWidgetsToModelDirectly(A_Counter counter,
      UI_CounterFrame cf) {
        counter.addObserver(cf);
    }
}
```

Die Steuerung des Play-In mit dem *In-Controller*

- Zum Play-In wird der Controller ein Listener der View-Widgets (hier buttons)
- Die update-Methode heisst `java.awt.ActionListener.actionPerformed(ActionEvent)`, da die Ablaufsteuerung des JDK diese sucht und aufruft
- Der Controller interpretiert die Eingaben des Benutzers und setzt sie auf das Model um (Steuerungsmaschine)
- Nutzt Entwurfsmuster State, Integerstate oder ImplicitIntegerState

```
// Die Steuerung belauscht die Knöpfe und interpretiert die Ereignisse
class ButtonInputController implements ActionListener { // In-Controller
  Counter myCounter;
  // push-Observer interpretiert Kommandos (Steuerungsmaschine)
  public void actionPerformed (ActionEvent event) {
  // Hier nur ein Grundzustand
    String cmd = event.getActionCommand();
    if (cmd.equals("Count"))
      myCounter.count();  // write: Aktion auf fachlichem Modell
    if (cmd.equals("Reset"))
      myCounter.reset();  // write: Aktion auf fachlichem Modell
    if (cmd.equals("Exit"))
      System.exit(0);     // Aktion: Beende Programm
  }

   /** Record the model object in the InputController,
     for read and write actions.  */
    public void setModel(A_Counter c) {      myCounter = c;    }
}
```

Alles zusammen: MVCModularDirectPlayOut.java

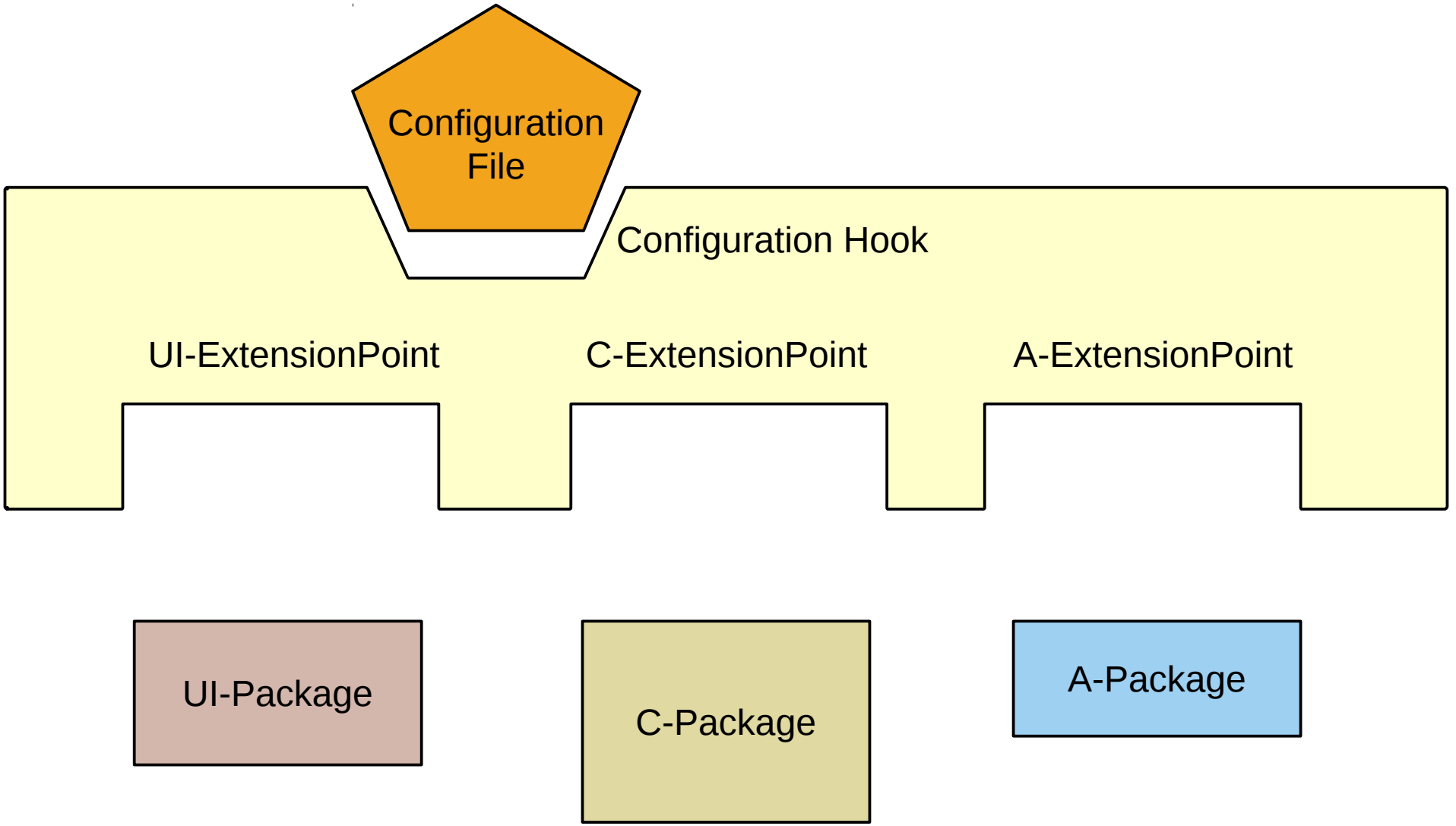

```
class MVCModularDirectPlayOut {
  public static void main (String[] argv) {
    // Phase 1a: Build application logic layer
    A_Counter counter = new A_Counter();

    // Phase 1b: Build Controller Layer
    ControlConnector connector = new ControlConnector();
    C_ButtonInputController bic = new C_ButtonInputController();
    C_WindowInputController wic = new C_WindowInputController();

    // Phase 1c: Build widget tree for display on screen
    UI_CounterFrame cf= new UI_CounterFrame(counter.getValue());

    // Phase 2: Connect Widget Tree and Application Logic, by
    // executing the wiring methods of the
    // GUI-Application-Logic connector
    connector.linkWindowInputController(cf,wic);
    connector.linkButtonInputController(cf,bic);
    connector.linkInputControllerToModel(bic,counter);
    connector.linkWidgetsToModelDirectly(counter,cf);

    // .. implizites Betreten der Reaktionsschleife:
    // Phase 3: reaktives Programm
  }
}
```

70.4. MVC Frameworks

MVC-Frameworks

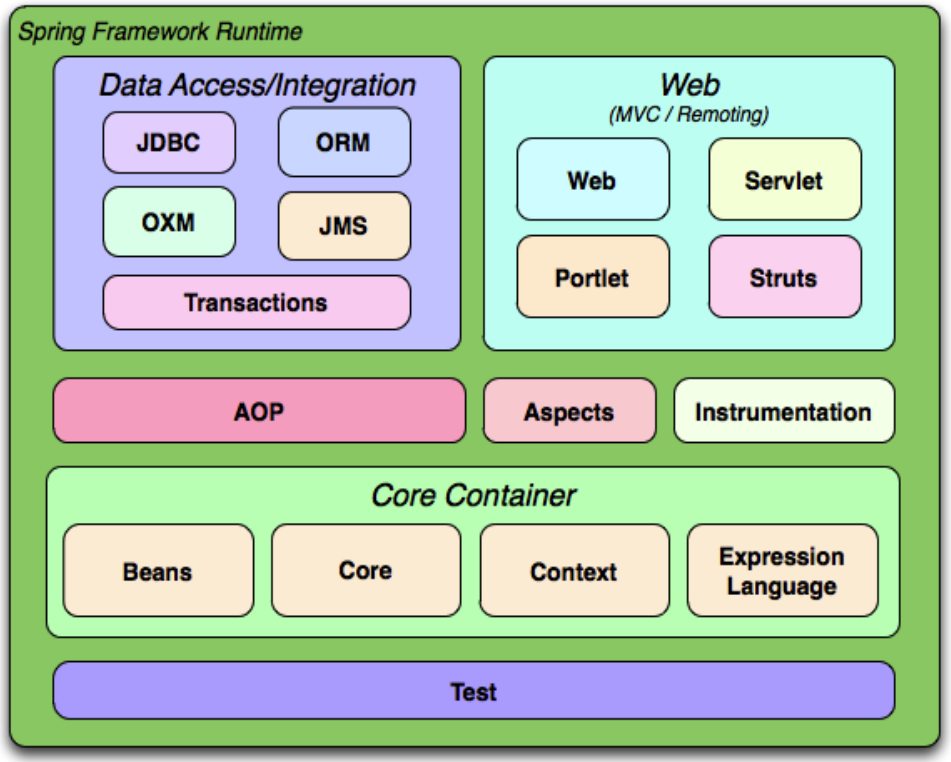
- Die Struktur einer Controllerschicht kann sich von Anwendungsklasse zu Anwendungsklasse sehr unterscheiden.
- Ein **MVC-Framework** gibt eine Struktur der Controllerschicht vor, definiert Protokolle für die Ereignismeldung und den Datenaustausch vor und kann durch den Entwickler erweitert werden.
  - MVC Frameworks benötigen Konfiguration und “Plugins”
  - **Oft folgt man dem Prinzip “Convention over configuration”:** Konventionen über Dateinverzeichnisse und Konfigurationsdateien vereinfachen dem MVC-Framework das Auffinden von Controller-, View-, Anwendungsklassen, sowie Hinweise zu ihrer Verdrahtung
  - Konfigurationsdateien meist in XML oder Java-Property-Lists
- Berühmte Beispiele:
  - Java: Spring, Struts
  - Ruby: Ruby on Rails
  - Groovy: Grails

MVC Frameworks



Spring Framework

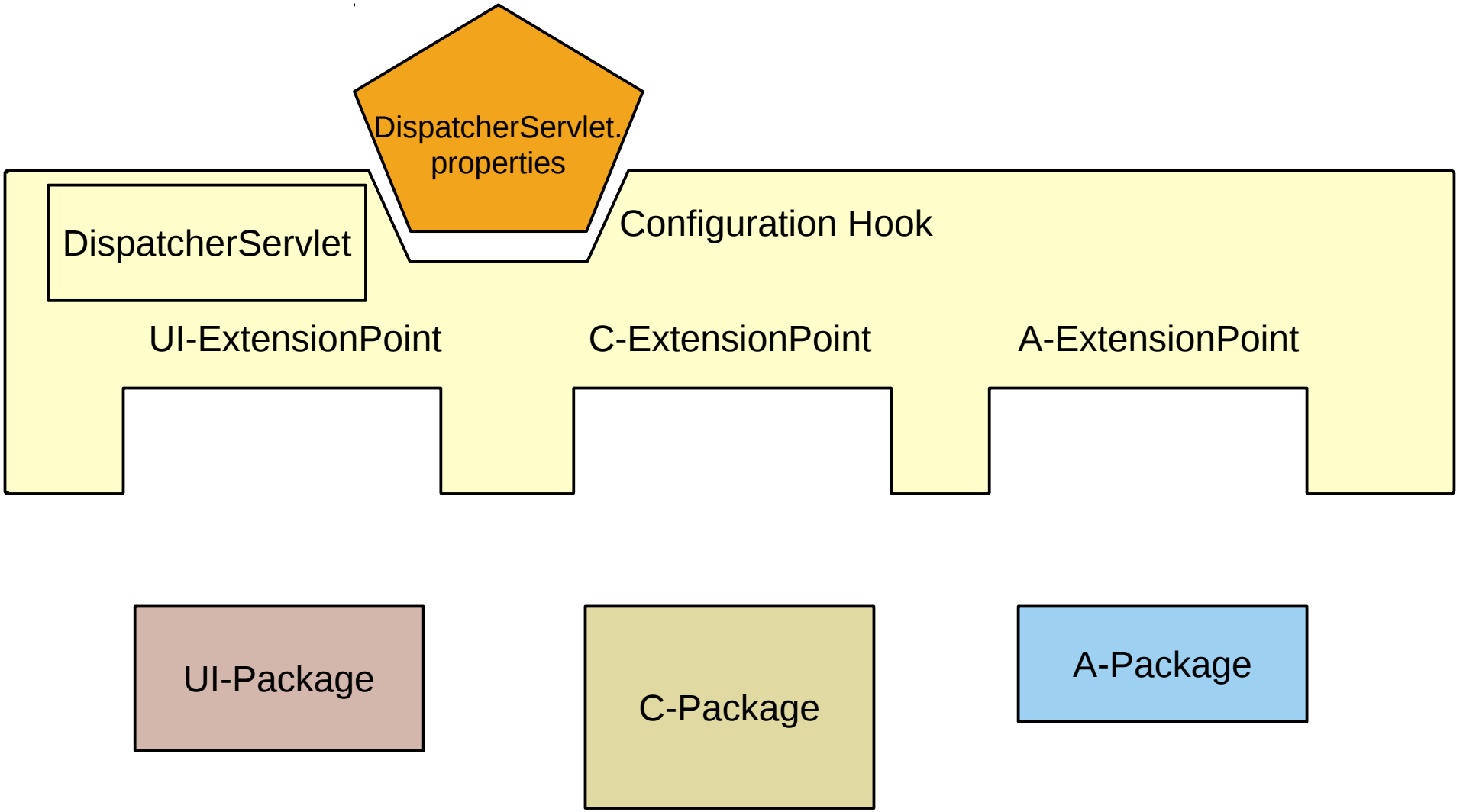
- *Spring* ist das im Praktikum im WS verwendete MVC-Framework
  - Webbasiert, d.h. Controllerschicht ist auf Client und Server verteilt implementiert
  - Konfigurierbar durch XML-Dateien und Java Property Files
  - Erweiterbar
- Web-MVC Frameworks brauchen *starke Schichtung*
- Bietet sehr viele verschiedene Pakete, nicht nur für Web-UIs



- http://docs.spring.io/spring/docs/3.1.x/

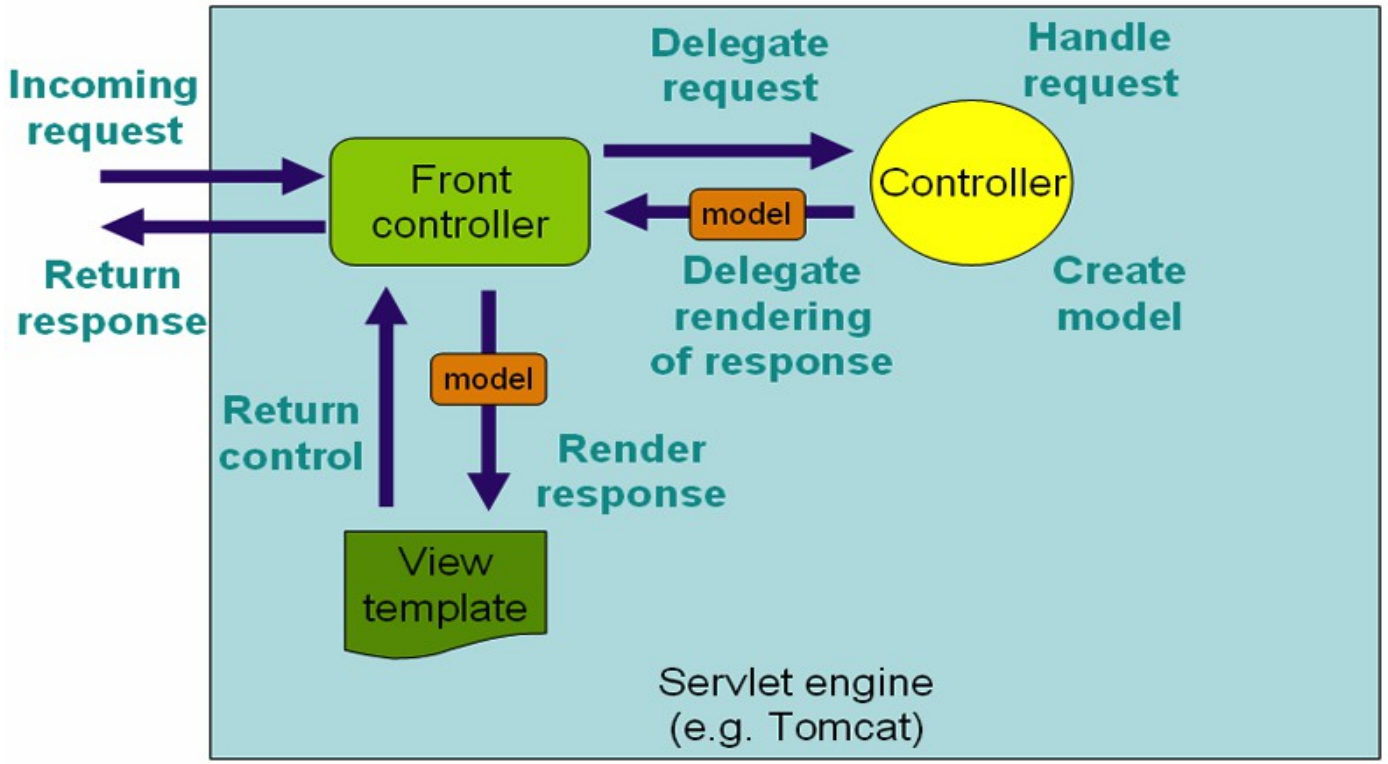
Spring Konfiguration

- Spring übernimmt das Management der Verteilung



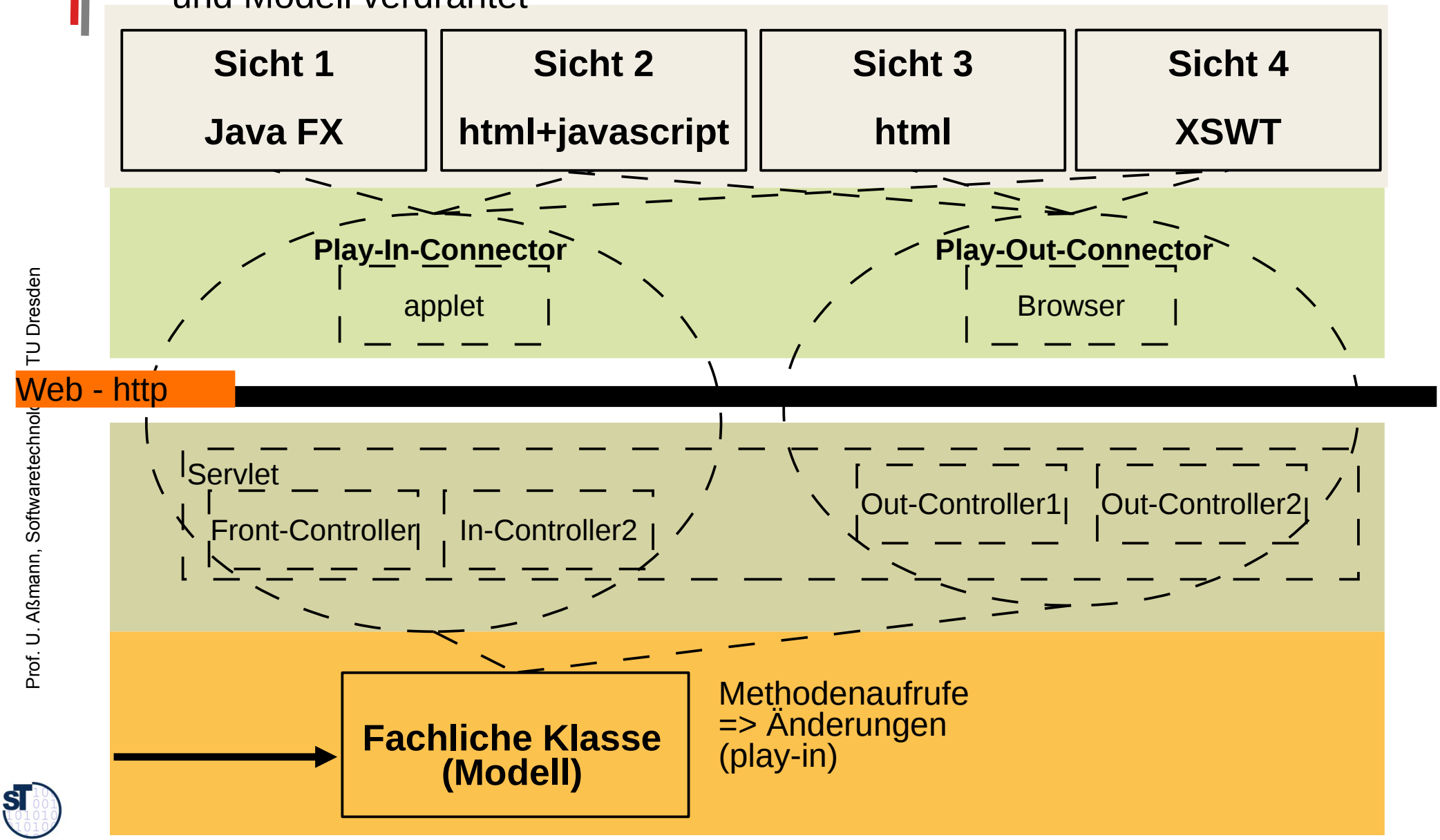
Implementation “Front Controller” in Web Systems (Server Side)

- Das Spring-DispatcherServlet wird mit einem “FrontController” realisiert, der das ankommende Ereignis interpretiert (Steuerungsmaschine) und an untergeordnete Controller bzw. Steuerungsmaschinen weiter leitet



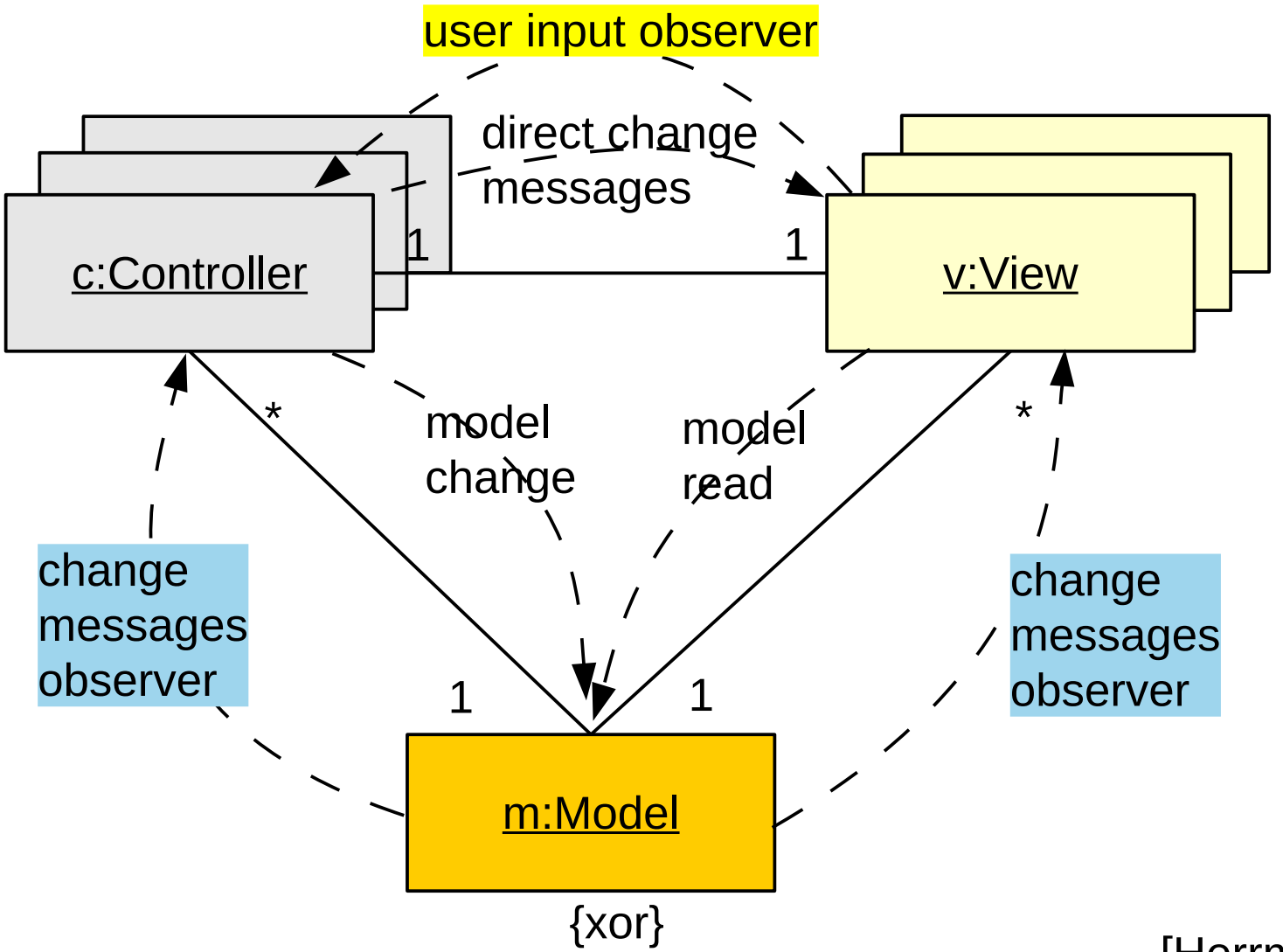
Controller sind Konnektoren zwischen Model und View

- Im Folgenden gibt es ein Hauptobjekt, den Konnektor, der View, Controller und Modell verdrahtet



70.5 Zusammenfassung

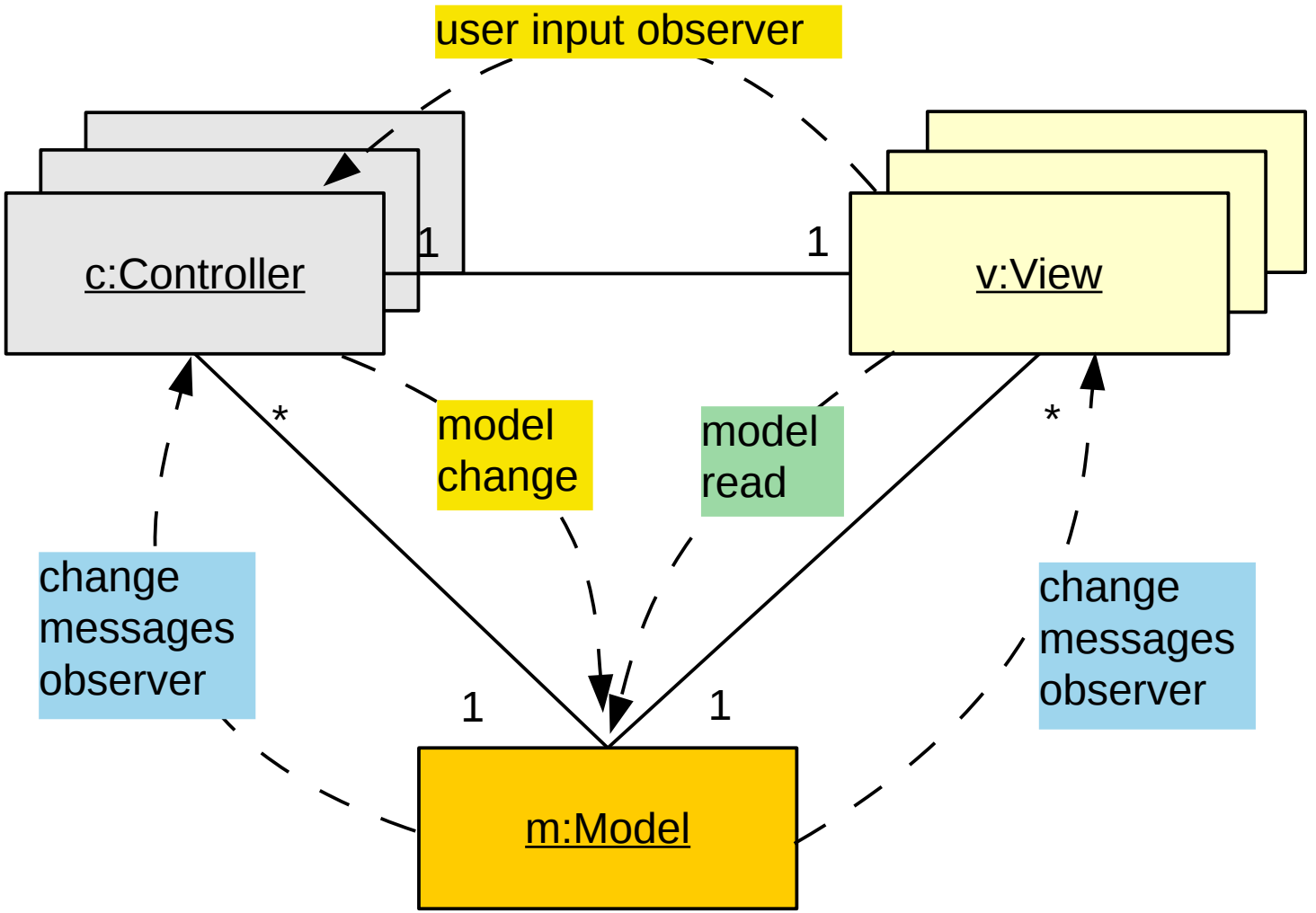
Überblick MVC ohne Zuordnung zur Schichtung



[Herrmann]

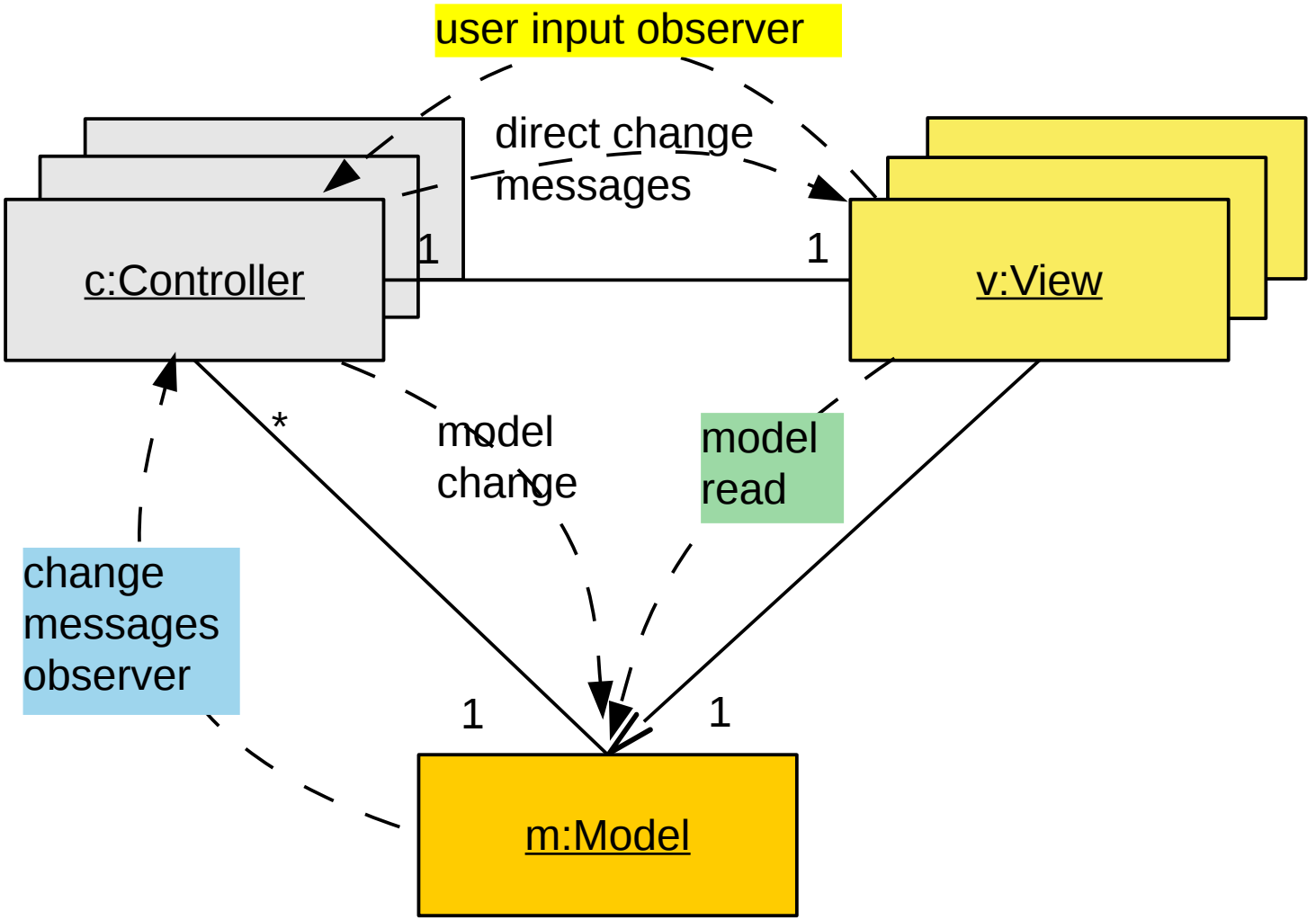
Überblick MVC (Active Model, Schwache Schichtung)

- Mit *aktivem Model* wird der View *direkt* vom Model benachrichtigt, und zieht danach bei Bedarf die Daten aus dem Model
- Entscheidung über Redraw liegt beim View



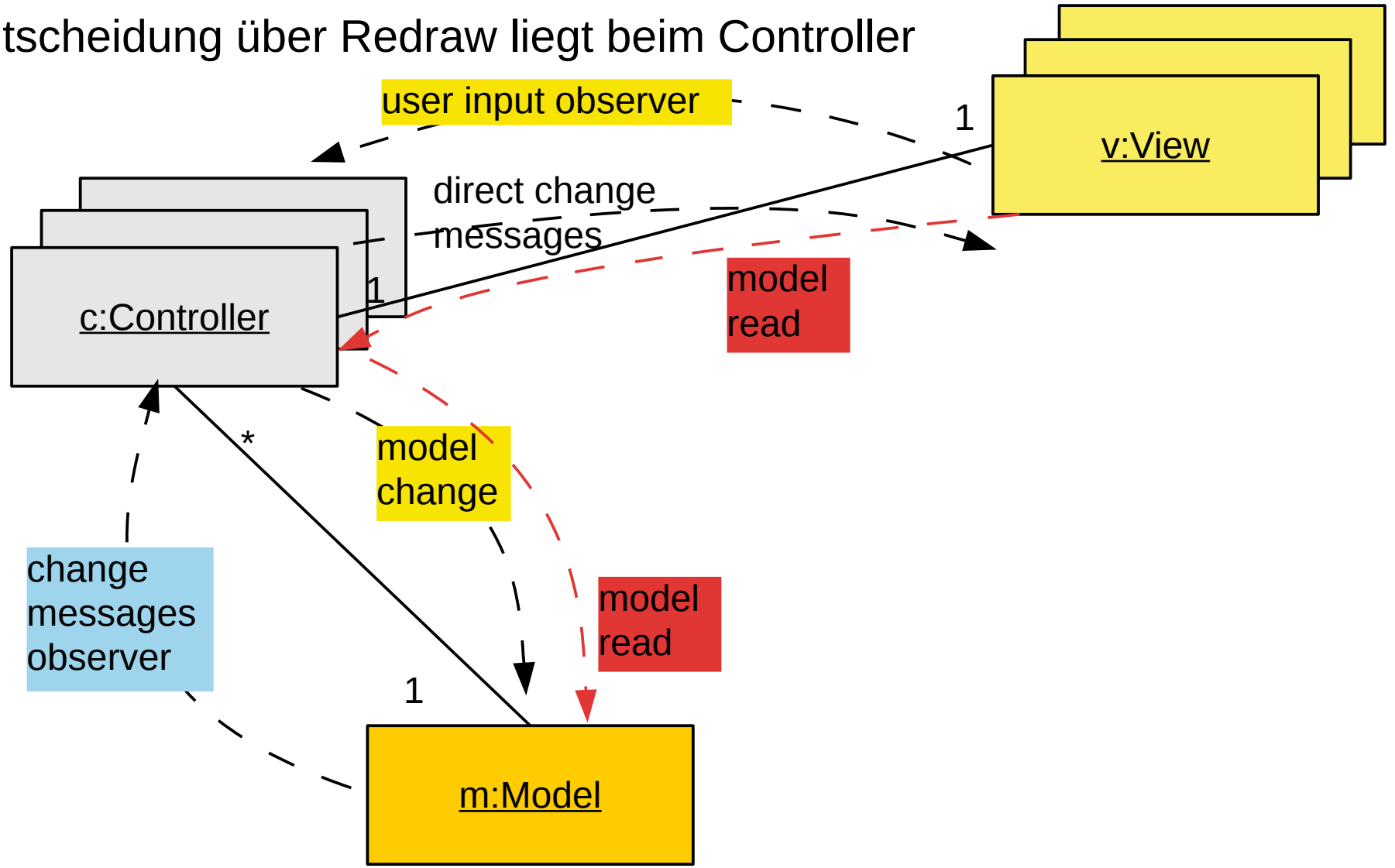
Überblick MVC (Passive Model, Semi-strikte Schichtung)

- Mit *passivem Model* zieht zwar der View die Daten aus dem Model, wird aber *indirekt* über den Controller benachrichtigt
- Entscheidung über Redraw liegt beim Controller



Überblick MVC (Passive Model, Strikte Schichtung)

- Mit *passivem Model* zieht zwar der View die Daten aus dem Model, wird aber *indirekt* über den Controller benachrichtigt
- Entscheidung über Redraw liegt beim Controller



Was haben wir gelernt?

- GUI-Programme laufen in 3 Phasen:
  - Aufbau der Fensterfronten (widget hierarchies) durch Konstruktoraufrufe und Additionen (embodiment)
  - Netzaufbau (Konnektor):
    - Vorbereitung Play-Out: Anschluß des View-Reaktionscodes als jdk-Observer des Modells
    - Vorbereitung Play-In: Anschluß des Controller als widget-Observer der Views, , oder mit Servlet in Spring
  - Reaktionsphase, bei der die Benutzeraktionen vom System als Ereignisobjekte ins Programm gegeben werden:
    - der Controller als Listener benachrichtigt und ausgeführt werden (Play-In)
    - die Views bzw. der Controller als Listener des Modells benachrichtigt werden (Play-Out)
- Der Kontrollfluß eines GUI-Programms wird *nie* explizit spezifiziert, sondern ergibt sich aus den Aktionen des Benutzers
  - Die Views reagieren auf Ereignisse im Screenbuffer, die von der Ablaufsteuerung gemeldet werden
  - Der Controller auf Widget-Veränderungen im View
  - Die Views auf Veränderungen im Modell

The End

- © Prof. H. Hussmann, Prof. U. Aßmann 1998-2013. used by permission. Verbreitung, Kopieren nur mit Zustimmung der Autoren.

70.A.1 Phase 1b) Weitere einfache Input-Controller als Implementierungen von EventListener-Schnittstellen (Play-In)

b) Vereinfachung 1: Unterklasse der Default-Implementierung WindowAdapter

- WindowAdapter bietet eine Default-Implementierung für die WindowListener-Funktionen an:

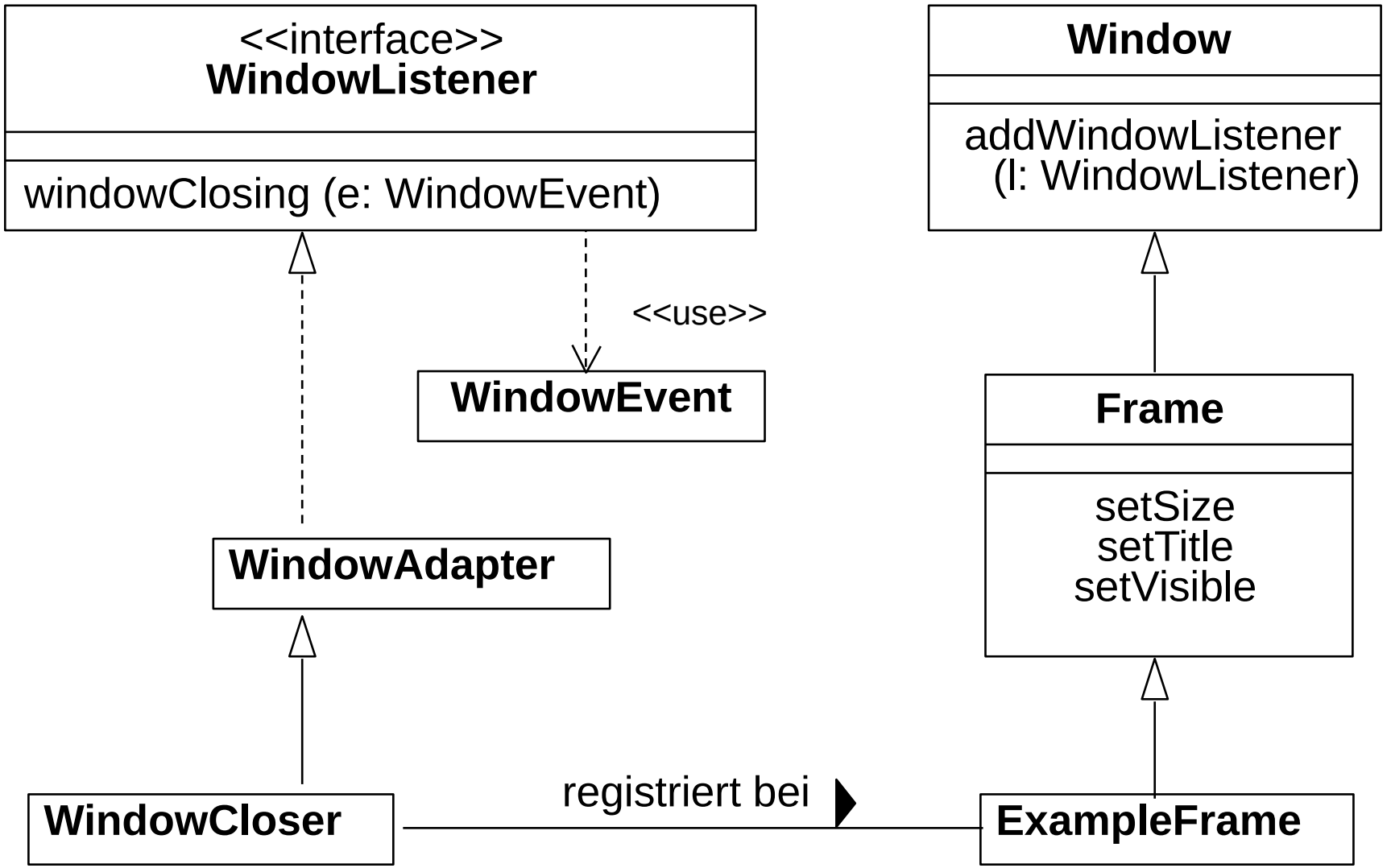
```
package java.awt.event;
public abstract class WindowAdapter
    implements WindowListener {
  public void windowClosed (WindowEvent ev) {}
  public void windowOpened (WindowEvent ev) {}
  public void windowIconified (WindowEvent ev) {}
  public void windowDeiconified (WindowEvent ev) {}
  public void windowActivated (WindowEvent ev) {}
  public void windowDeactivated (WindowEvent ev) {}
  public void windowClosing (WindowEvent ev)  {}
}
```

Vereinfachung 1: Unterklasse der Default-Implementierung WindowAdapter

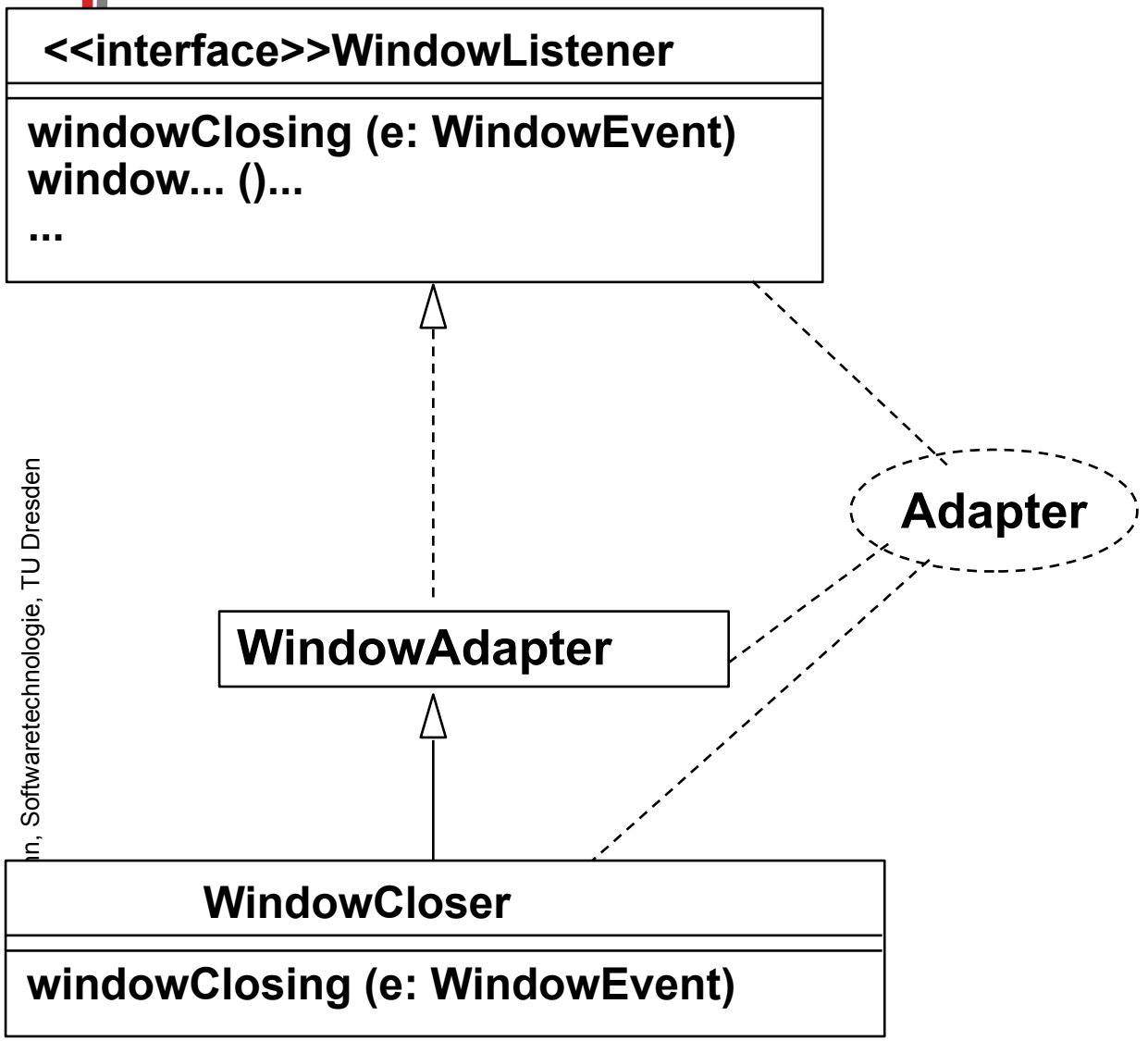
- Redefinition einer leeren Reaktionsmethode:

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

class WindowCloser extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}

class ExampleFrame extends Frame {

  public ExampleFrame () {
      setTitle("untitled");
      setSize(150, 50);
      addWindowListener(new WindowCloser());
      setVisible(true);
   }
}

class GUI3 {
  public static void main (String[] argv) {
      ExampleFrame f = new ExampleFrame();}
}
```

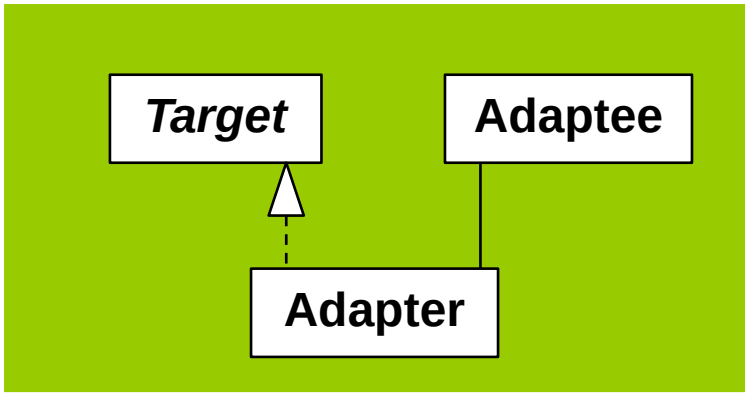
Vereinfachung 1: Unterklasse der DefaultImplementierung WindowAdapter



WindowAdapter ist eine Default-Implementierung



Ist das eine Anwendung des Adapter-Musters?



.. sollte besser DefaultWindowListener heissen..

b) Vereinfachung 2: Innere Klasse benutzen


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

class ExampleFrame extends Frame {

  /* inner */ class WindowCloser extends WindowAdapter {
    public void windowClosing(WindowEvent event) {
      System.exit(0);
    }
  }

  public ExampleFrame () {
      setTitle("untitled");
      setSize(150, 50);
      addWindowListener(new WindowCloser());
      setVisible(true);
    }
}

class GUI4 {
public static void main (String[] argv) {
  ExampleFrame f = new ExampleFrame();}}
```

c) Vereinfachung 3: Anonyme Klasse benutzen


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

class ExampleFrame extends Frame {

  public ExampleFrame () {
      setTitle("untitled");
      setSize(150, 50);
      addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent event) {
         System.exit(0);
       }});
      setVisible(true);
  }
}

class GUI5 {
  public static void main (String[] argv) {
      ExampleFrame f = new ExampleFrame();
  }
}
```

70.A.2 Layout Control von Widgets

Layout-Manager für Fensterelemente

- Def.: Ein **Layout-Manager** ist ein Objekt, das Methoden bereitstellt, um die graphische Repräsentation verschiedener Objekte innerhalb eines Container-Objektes anzuordnen.
- Formal ist LayoutManager ein Interface, für das viele Implementierungen möglich sind.
- In Java definierte Layout-Manager (Auswahl):
  - FlowLayout (java.awt.FlowLayout)
  - BorderLayout (java.awt.BorderLayout)
  - GridLayout (java.awt.GridLayout)

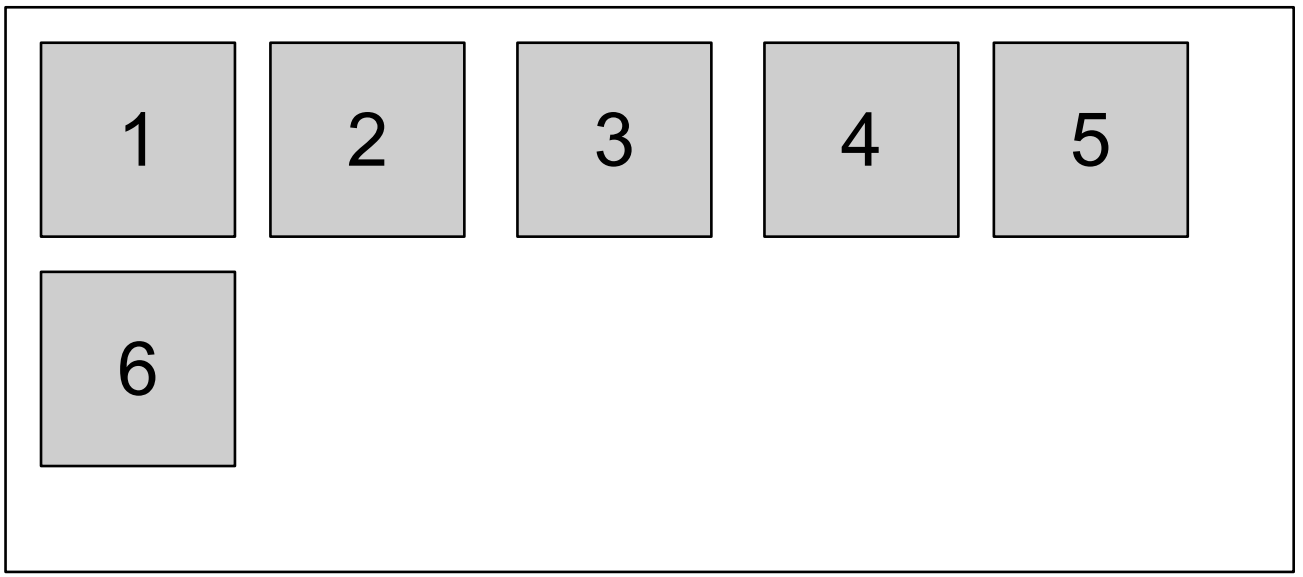
- In awt.Component:

```
public void add (Component comp, Object constraints);
```

erlaubt es, zusätzliche Information (z.B. Orientierung, Zeile/Spalte) an den Layout-Manager zu übergeben

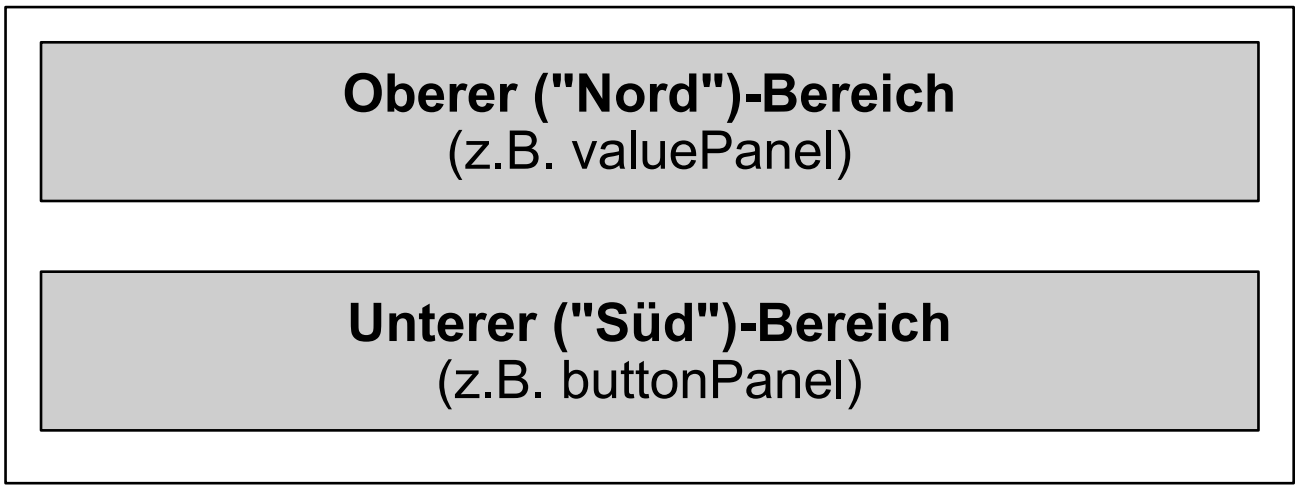
Flow-Layout

- Grundprinzip:
  - Anordnung analog Textfluß:
    von links nach rechts und von oben nach unten
- Default für Panels
  - z.B. in valuePanel und buttonPanel
    für Hinzufügen von Labels, Buttons etc.
- Parameter bei Konstruktor: Orientierung auf Zeile, Abstände
- Constraints bei **`add`**: keine



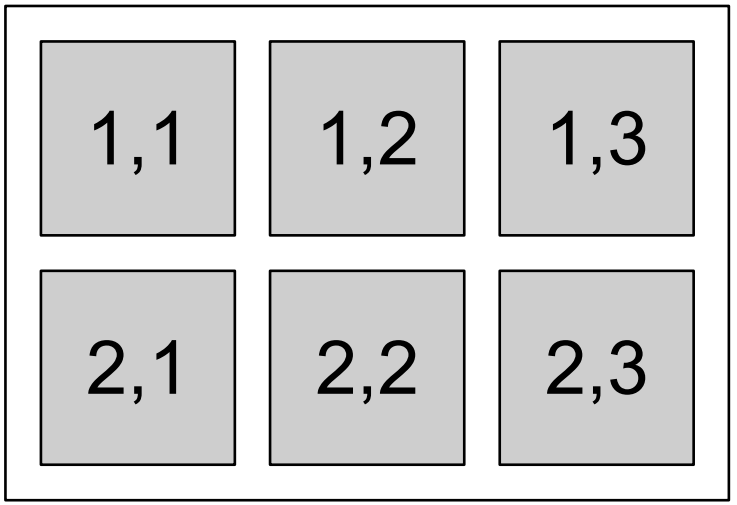
Border-Layout

- Grundprinzip:
  - Orientierung nach den Seiten (N, S, W, O)
    bzw. Mitte (center)
- Default für Window, Frame
  - z.B. in CounterFrame
    für Hinzufügen von valuePanel, buttonPanel
- Parameter bei Konstruktor: Keine
- Constraints bei **`add:`**
  - **`BorderLayout.NORTH, SOUTH, WEST, EAST, CENTER`**



Grid-Layout

- Grundprinzip:
  - Anordnung nach Zeilen und Spalten
- Parameter bei Konstruktor:
  - Abstände, Anzahl Zeilen, Anzahl Spalten
- Constraints bei **`add:`** Keine



Die Sicht *(View)*: Alle sichtbaren Elemente


```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");

  public CounterFrame (Counter c) {
    setTitle("SwingCounter");
    valuePanel.add(new JLabel("Counter value"));
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    getContentPane().add(valuePanel, BorderLayout.NORTH);
    buttonPanel.add(countButton);
    buttonPanel.add(resetButton);
    buttonPanel.add(exitButton);
    getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    pack();
    setVisible(true);
  }
}
```