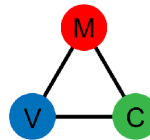


Model-View-Controller



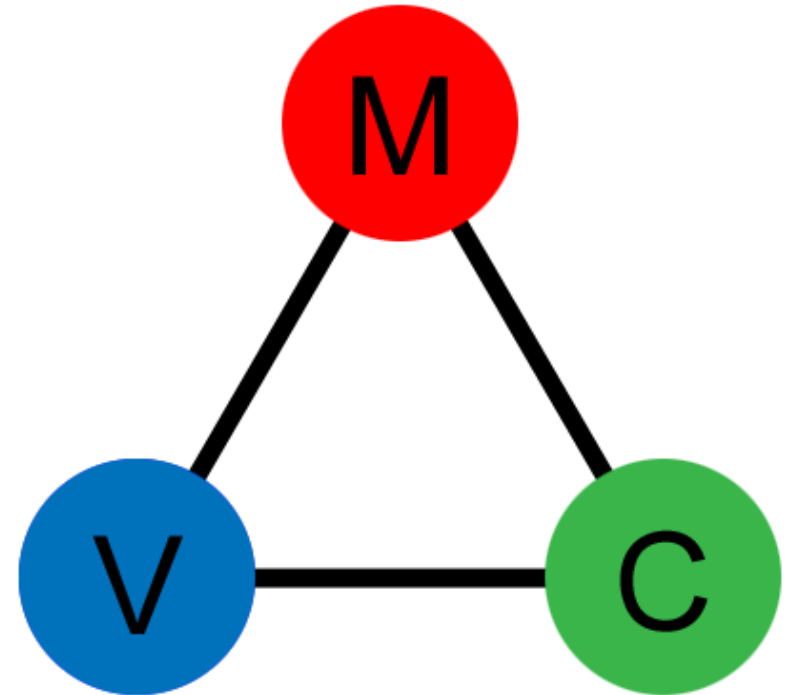
Seminar Software-Entwurf

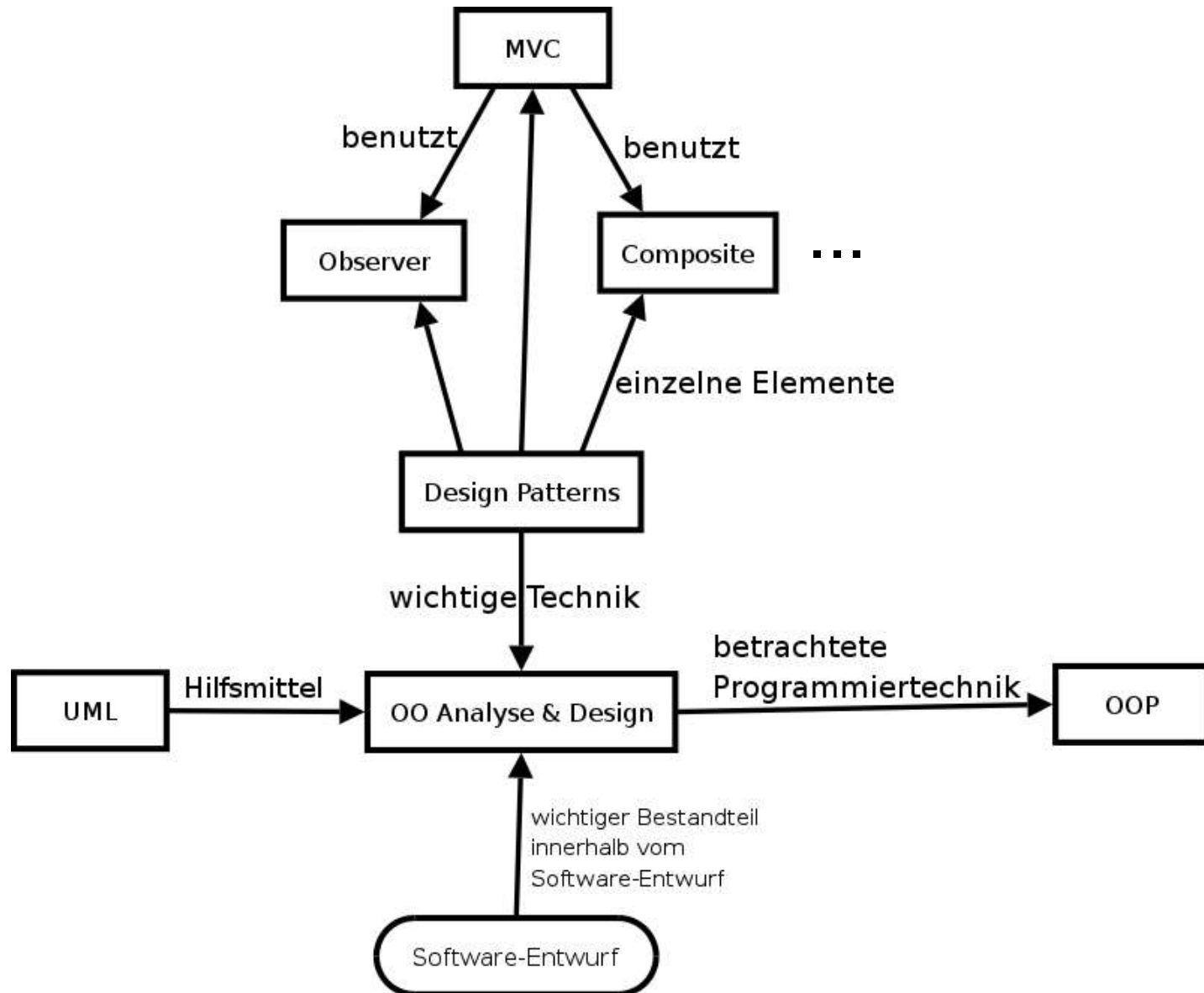
Leif Singer

<leif@singer.sh>

Gliederung

- Einordnung
- Überblick & Geschichte
- Aufgabenverteilung & Beispiel
 - Model
 - View
 - Controller
- Zusammenspiel
- Web Anwendungen
- Fazit





Überblick & Geschichte

- trennt Präsentation, Daten und Interaktion
 - bekanntes und erfolgreiches Pattern
 - z. Bsp. Swing ist MVC
 - benutzt selbst andere Patterns
-
- **1978** im Xerox PARC entstanden (Smalltalk)
 - wird Trygve Reenskaug zugerechnet
 - offizielle Veröffentlichung erst 1988

Überblick & Geschichte

- Siegeszug:
 - erst Smalltalk
 - dann Java / Swing
 - nun Web Anwendungen
 - ... und nicht nur dort

- der bedeutendste Schritt für GUI-Entwicklung!
... aber wie funktioniert MVC genau?

Überblick & Geschichte

- Problem:
 - Programmlogik mit GUI-Code verwoben
 - erschwert Erweiterung & Wartung
- Lösung:
 - einmal Programmlogik schreiben (Model)
 - daran beliebig viele GUIs binden (Views)
 - Kommunikation zwischen Model & View jeweils durch Controller
- Heutige Probleme werden auch gelöst!
 - verschiedene Endgeräte
 - Arbeitsteilung besser möglich

Als Beispiel: ein Applet, dass einen Ball bewegt

- eigentliche Programmlogik
- Daten mit Änderungsmethoden
- wird von Views beobachtet (extends Observable)
- benachrichtigt Views bei Änderungen
- weiss sonst nichts über die Views / Controller
- hilfreich: Model Interface festlegen, um Models leicht austauschen zu können



```
class Model extends Observable {  
    private Point limit = new Point(200,200);  
    private Point speed = new Point(6,4);  
    private Point position = new Point(0,0);  
  
    public void lauf() {  
        position.translate(speed.x, 0);  
        /*  
         * neue Position berechnen ...  
         */  
        this.setChanged();  
        this.notifyObservers();  
    }  
  
    public Point getPosition() {  
        return position;  
    }  
  
    public void setLimit(Point limit) {  
        this.limit = limit;  
    }  
}
```



```
class Model extends Observable {  
    private Point limit = new Point(200,200);  
    private Point speed = new Point(6,4);  
    private Point position = new Point(0,0);  
  
    public void lauf() {  
        position.translate(speed.x, 0);  
        /*  
         * neue Position berechnen ...  
         */  
        this.setChanged();  
        this.notifyObservers();  
    }  
  
    public Point getPosition() {  
        return position;  
    }  
  
    public void setLimit(Point limit) {  
        this.limit = limit;  
    }  
}
```



```
class Model extends Observable {  
    private Point limit = new Point(200,200);  
    private Point speed = new Point(6,4);  
    private Point position = new Point(0,0);  
  
    public void lauf() {  
        position.translate(speed.x, 0);  
        /*  
         * neue Position berechnen ...  
         */  
        this.setChanged();  
        this.notifyObservers();  
    }  
  
    public Point getPosition() {  
        return position;  
    }  
  
    public void setLimit(Point limit) {  
        this.limit = limit;  
    }  
}
```

```
class Model extends Observable {  
    private Point limit = new Point(200,200);  
    private Point speed = new Point(6,4);  
    private Point position = new Point(0,0);  
  
    public void lauf() {  
        position.translate(speed.x, 0);  
        /*  
         * neue Position berechnen ...  
         */  
        this.setChanged();  
        this.notifyObservers();  
    }  
  
    public Point getPosition() {  
        return position;  
    }  
  
    public void setLimit(Point limit) {  
        this.limit = limit;  
    }  
}
```



- Präsentation des Modells für den Benutzer
- beobachtet Model (implements Observer)
- aktualisiert sich bei Änderungen
- toll: schachtelbar! (Composite Pattern)
- läßt sich sehr einfach austauschen



```
class View extends Canvas implements Observer {  
    public final int BALL_SIZE = 20;  
    private Model model;  
  
    public View(Model model) {  
        super();  
        this.model = model;  
        this.model.addObserver(this);  
    }  
  
    public void paint(Graphics g) {  
        g.setColor(Color.red);  
        g.fillOval(model.getPosition().x, model.getPosition().y, this.BALL_SIZE, this.BALL_SIZE);  
    }  
  
    public void update(Observable obs, Object obj) {  
        this.repaint();  
    }  
}
```



```
class View extends Canvas implements Observer {  
    public final int BALL_SIZE = 20;  
    private Model model;  
  
    public View(Model model) {  
        super();  
        this.model = model;  
        this.model.addObserver(this);  
    }  
  
    public void paint(Graphics g) {  
        g.setColor(Color.red);  
        g.fillOval(model.getPosition().x, model.getPosition().y, this.BALL_SIZE, this.BALL_SIZE);  
    }  
  
    public void update(Observable obs, Object obj) {  
        this.repaint();  
    }  
}
```



```
class View extends Canvas implements Observer {  
    public final int BALL_SIZE = 20;  
    private Model model;  
  
    public View(Model model) {  
        super();  
        this.model = model;  
        this.model.addObserver(this);  
    }  
  
    public void paint(Graphics g) {  
        g.setColor(Color.red);  
        g.fillOval(model.getPosition().x, model.getPosition().y, this.BALL_SIZE, this.BALL_SIZE);  
    }  
  
    public void update(Observable obs, Object obj) {  
        this.repaint();  
    }  
}
```



- interpretiert die Events vom View, leitet sie an Model weiter
- zu jedem View ein Controller
 - beliebig viele Paare für ein Model
- in Swing oft als `ActionListener` im View
 - Sun nennt das *Delegate* (!= C# Delegates!)



```
public class Controller extends Applet {  
    Panel buttonPanel = new Panel();  
    Button button = new Button("Lauf!");  
    Model model = new Model();  
    View view = new View(model);  
  
    public void init() {  
        /*  
        * View zusammenbauen ...  
        */  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                model.lauf();  
            }  
        });  
    }  
  
    public void start() {  
        // ausgeblendet, Applet starten  
    }  
}
```



```
public class Controller extends Applet {  
    Panel buttonPanel = new Panel();  
    Button button = new Button("Lauf!");  
    Model model = new Model();  
    View view = new View(model);  
  
    public void init() {  
        /*  
        * View zusammenbauen ...  
        */  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                model.lauf();  
            }  
        });  
    }  
  
    public void start() {  
        // ausgeblendet, Applet starten  
    }  
}
```

```
public class Controller extends Applet {  
    Panel buttonPanel = new Panel();  
    Button button = new Button("Lauf!");  
    Model model = new Model();  
    View view = new View(model);
```

```
    public void init() {
```

```
        /*  
        * View zusammenbauen ...  
        */
```

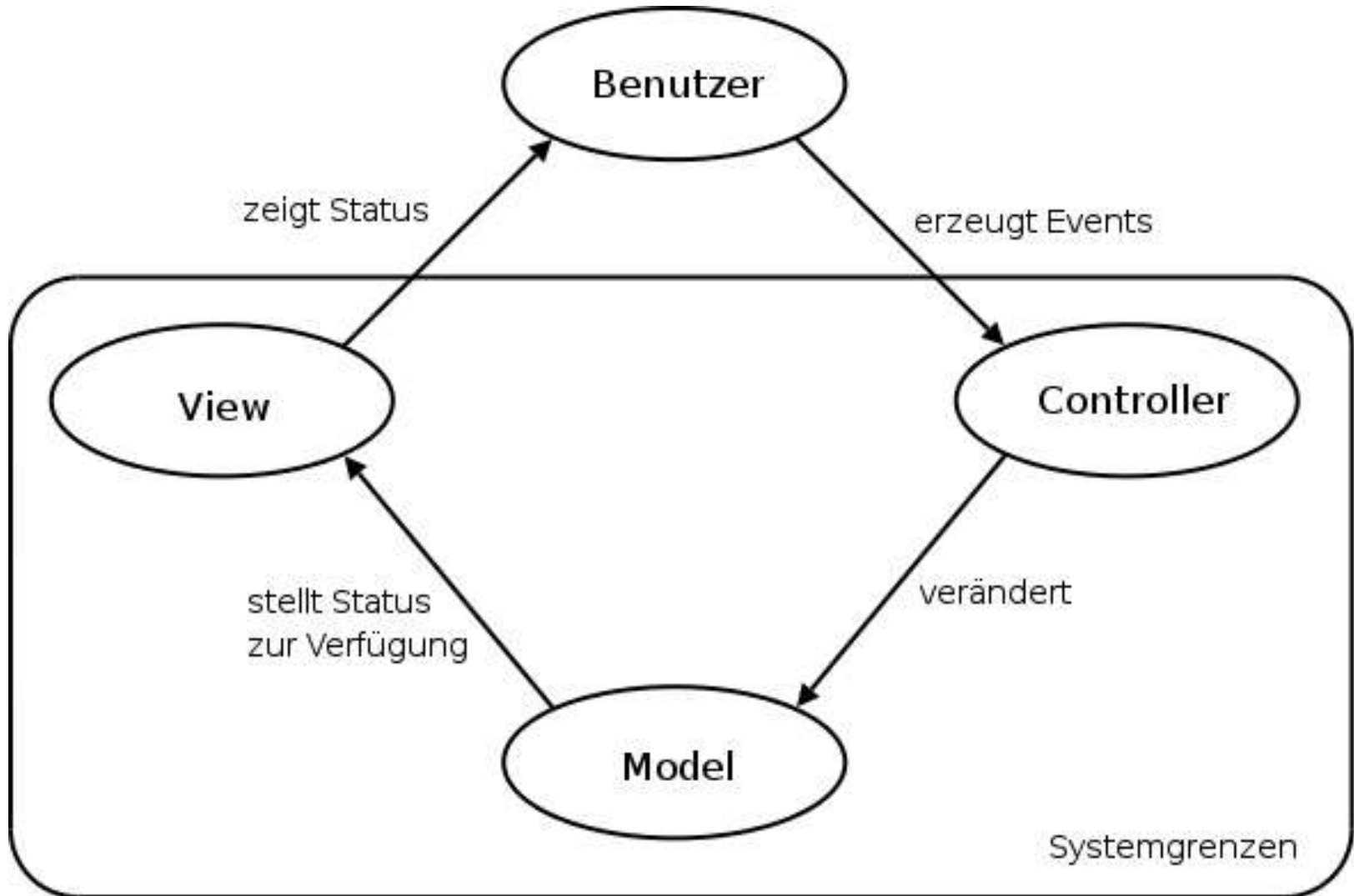
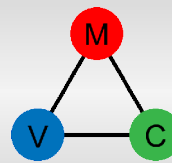
```
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent event) {  
                model.lauf();  
            }  
        });
```

```
    }
```

```
    public void start() {
```

```
        // ausgeblendet, Applet starten
```

```
    }
```



???

View austauschen

Heute wieder dabei: Konqi!



View austauschen

Der neue View

- View beerben: ViewKonqi
- Bild laden, Bildgröße berücksichtigen
- Bild zeichnen

Sonstige Änderungen?

- im Controller KonqiView instanziiieren
- sonst nichts

View austauschen

```
class ViewKonqi extends View {  
    public final int BALL_SIZE = 118;  
    private Model model;  
    private Image konqi;  
  
    public ViewKonqi(Model model) {  
        super(model);  
        this.model = model;  
        this.model.addObserver(this);  
        this.konqi = Toolkit.getDefaultToolkit().getImage("konqi.png");  
    }  
  
    public void paint(Graphics g) {  
        g.drawImage(this.konqi, model.getPosition().x, model.getPosition().y, Color.WHITE, this);  
    }  
}
```

View austauschen

```
class ViewKonqi extends View {  
    public final int BALL_SIZE = 118;  
    private Model model;  
    private Image konqi;  
  
    public ViewKonqi(Model model) {  
        super(model);  
        this.model = model;  
        this.model.addObserver(this);  
        this.konqi = Toolkit.getDefaultToolkit().getImage("konqi.png");  
    }  
  
    public void paint(Graphics g) {  
        g.drawImage(this.konqi, model.getPosition().x, model.getPosition().y, Color.WHITE, this);  
    }  
}
```

View austauschen

```
class ViewKonqi extends View {  
    public final int BALL_SIZE = 118;  
    private Model model;  
    private Image konqi;  
  
    public ViewKonqi(Model model) {  
        super(model);  
        this.model = model;  
        this.model.addObserver(this);  
        this.konqi = Toolkit.getDefaultToolkit().getImage("konqi.png");  
    }  
  
    public void paint(Graphics g) {  
        g.drawImage(this.konqi, model.getPosition().x, model.getPosition().y, Color.WHITE, this);  
    }  
}
```

View austauschen

```
class ViewKonqi extends View {  
    public final int BALL_SIZE = 118;  
    private Model model;  
    private Image konqi;  
  
    public ViewKonqi(Model model) {  
        super(model);  
        this.model = model;  
        this.model.addObserver(this);  
        this.konqi = Toolkit.getDefaultToolkit().getImage("konqi.png");  
    }  
  
    public void paint(Graphics g) {  
        g.drawImage(this.konqi, model.getPosition().x, model.getPosition().y, Color.WHITE, this);  
    }  
}
```

View austauschen

```
class ViewKonqi extends View {  
    public final int BALL_SIZE = 118;  
    private Model model;  
    private Image konqi;  
  
    public ViewKonqi(Model model) {  
        super(model);  
        this.model = model;  
        this.model.addObserver(this);  
        this.konqi = Toolkit.getDefaultToolkit().getImage("konqi.png");  
    }  
  
    public void paint(Graphics g) {  
        g.drawImage(this.konqi, model.getPosition().x, model.getPosition().y, Color.WHITE, this);  
    }  
}
```

View austauschen

```
public class Controller extends Applet {  
    Panel buttonPanel = new Panel();  
    Button button = new Button("Lauf!");  
    Model model = new Model();  
    ViewKonqi view = new ViewKonqi(model);  
  
    public void init() {  
        /*  
        * View zusammenbauen ...  
        */  
        button.addActionListener(new ActionListener() {  
        }  
    }  
  
    public void start() {  
        model.setLimit(new Point(view.getSize().width - view.BALL_SIZE, view.getSize().height - view.BALL_SIZE));  
        repaint();  
    }  
    // ausgeblendet, Applet starten  
}
```

Sind bisher Fragen aufgekomen?

Web Anwendungen

- früher: statisches HTML
 - offensichtlich: zu unflexibel
- CGI und Skriptsprachen
 - zu wenig strukturiert
 - zu viel Mehrfacharbeit
- J2EE, Frameworks, Application Server
 - Entwicklung noch aktiv (Java Server Faces!)
 - Model 1 & 2 Architekturen: MVC Variationen

Model 1

- "seiten-zentrisch"
- Trennung von View und Model z. Bsp. über Beans
 - provoziert aber doch Vermengung
 - Arbeitstrennung trotzdem problematisch
- Status über Get/Post
- traditionelle, harte Verlinkung von JSPs
 - starke Kopplung :(
- kein zentraler Controller
 - bspw. Input-Validierung auf *jeder* Seite

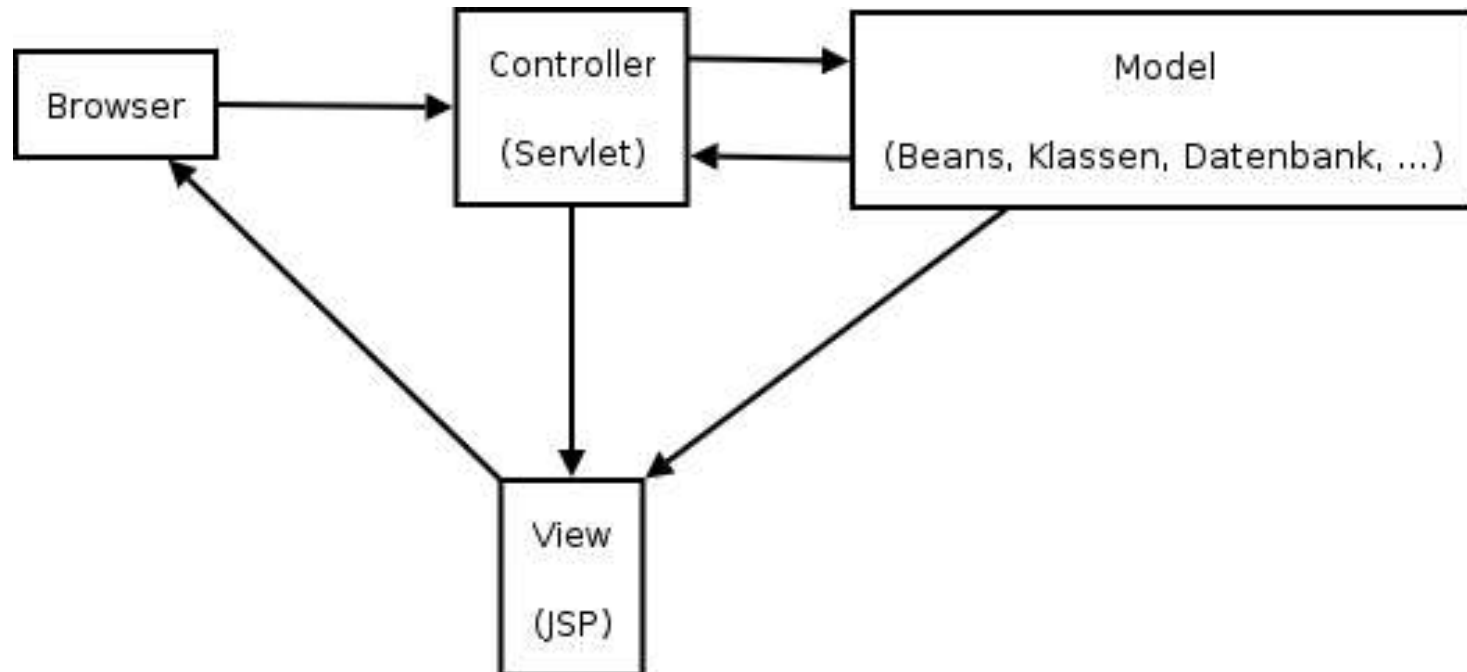
Model 2

- empfohlen für mittlere bis große Projekte
- "servlet-zentrisch"
- ein zentrales Controller-Servlet arbeitet als Dispatcher
 - relevante Patterns: Front Controller, Application Controller
 - validiert Eingabe, wählt Sprache, etc.
 - wählt View aus
- Views sind JSPs
 - interagiert mit Model (typischerweise JavaBean(s))
 - gibt Response zurück an Servlet
- z.Bsp. Struts Framework

Web Anwendungen

Model 2

- Browser sendet Request, Controller nimmt an
- Controller-Servlet benutzt ggf. das Model, leitet dann an entsprechenden View weiter
- View greift auf Model zu, generiert Response



Spezielles Framework: JavaServer Faces

- aktuelle Entwicklung (1.0: März 2004)
- entstanden durch Java Community Process
- Besonderheit: nur semantische Widgets
 - je nach Anwendungsbereich eigene Renderer bauen!
 - XHTML
 - WML
 - XUL
 - ...
- JSF: Fokus auf UI Rendering, Events, ...
- Struts: eher höhere Abstraktion; Forms, Actions, ...
- momentanes Ziel: JSF & Struts kombinierbar machen

Verwendete Patterns

- Observer
- Composite
- Front Controller, Application Controller (Model 2)
- Strategy

- MVC ist also ein Pattern aus anderen Patterns
- ein Meta-Composite?

Anmerkungen

Kontroverse: MVC ist nicht OO!

- Argument: lückenhafte Kapselung
 - View und Controller müssen Model kennen
- Konsequenz ist wichtig, aber praktische Anwendung wichtiger
- Na und? Nutzen maximieren, Paradigmen mischen

Nicht übertreiben!

- Risiko: MVC nur wegen Buzz-Faktor benutzen
 - deshalb immer fragen: ist das hier sinnvoll?
 - sicherlich Erfahrung nötig
- gilt allgemein für die Verwendung von Patterns!

Vorteile

- Austauschbarkeit der Komponenten
- Arbeitsteilung
- Wiederverwendbarkeit
- Übersichtlichkeit
- Wartbarkeit

Nachteile

- evtl. zu viel Aufwand für
 - *sehr* kleine Projekte
 - Proof of Concept Prototypen
- GUI-Tests fehleranfälliger: "Liar View" Bug Pattern
 - bspw. `notifyListeners()` vergessen

Danke für die Aufmerksamkeit!