

Software Design Pattern

Model-View-Controller

Michael Lühr

Gliederung

- Einführung und Problemstellung
- Ansatz durch MVC
- Detaillierte Darstellung der Komponenten
 - Model
 - View
 - Controller
- Vor- und Nachteile
- Zusammenfassung

Einführung

- Ursprung: Smalltalk (1978)
- Problem: Erstellen einer Interaktiven GUI-Anwendung
 - Portabilität auf unterschiedliche Endgeräte
 - Änderungen bzw. Erweiterungen der Nutzerschnittstelle häufiger als an der Geschäftslogik
 - Mix von Code für Darstellung und Geschäftslogik erschwert Erstellung einer portierbaren Anwendung

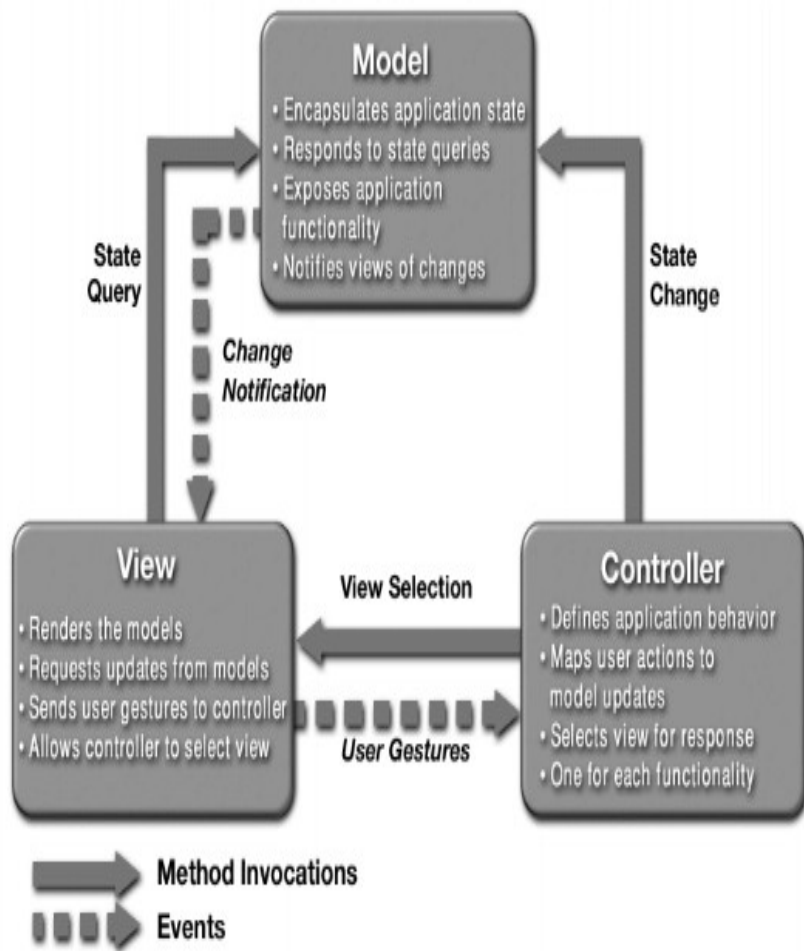
Anforderungen

- Änderungen der Oberfläche soll Code der Geschäftslogik nicht beeinflussen
- Portierung der Anwendung soll Verhalten der Anwendung nicht beeinflussen
- Datenänderungen sollen sichtbar sein

Ansatz durch MVC

- Strikte Trennung der Geschäftslogik von der Darstellung
- Entspricht Eingabe – Verarbeitung – Ausgabe Prinzip
- Unterteilt in 3 Komponenten
 - Model (Verarbeitung)
 - View (Ausgabe)
 - Controller (Eingabe)

Ansatz durch MVC



- **Model**
 - Enthält Geschäftslogik, unabhängig von den anderen Komponenten
- **View**
 - Enthält Darstellung der Daten des Models, daher Abhängigkeit zum Model
- **Controller**
 - Zwischenschicht, die Eingaben verarbeitet und zwischen View und Model vermittelt

Ansatz durch MVC

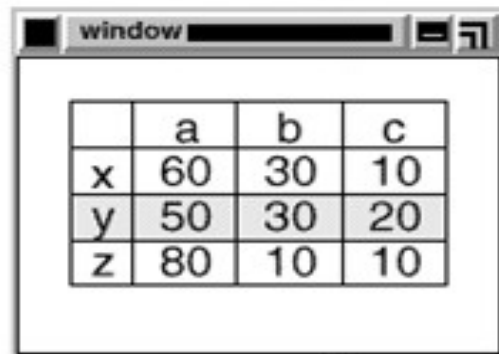
- MVC ist Strukturmuster
 - Teilt Aufgaben auf verschiedene Klassen auf
 - Einzelne Komponenten adaptieren andere Entwurfsmuster
 - Model: Beobachter-Muster
 - View: Composite-Muster
 - Controller: Strategie-Muster

Detaillierte Darstellung der Komponenten

Model
View
Controller

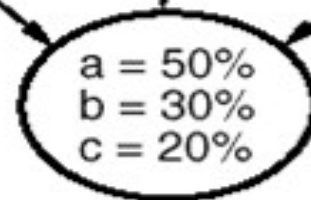
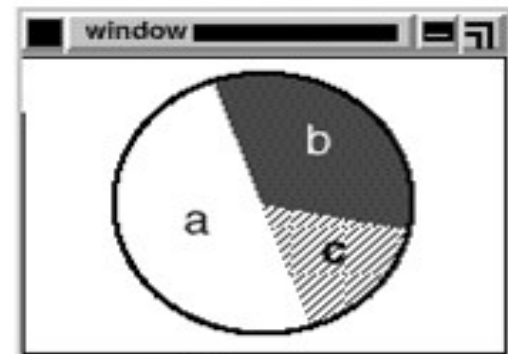
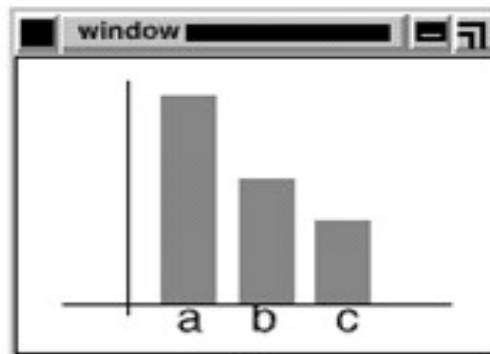
Model Beispiel

views



A window titled 'window' containing a table with 4 columns and 4 rows. The columns are labeled 'a', 'b', and 'c'. The rows are labeled 'x', 'y', and 'z'.

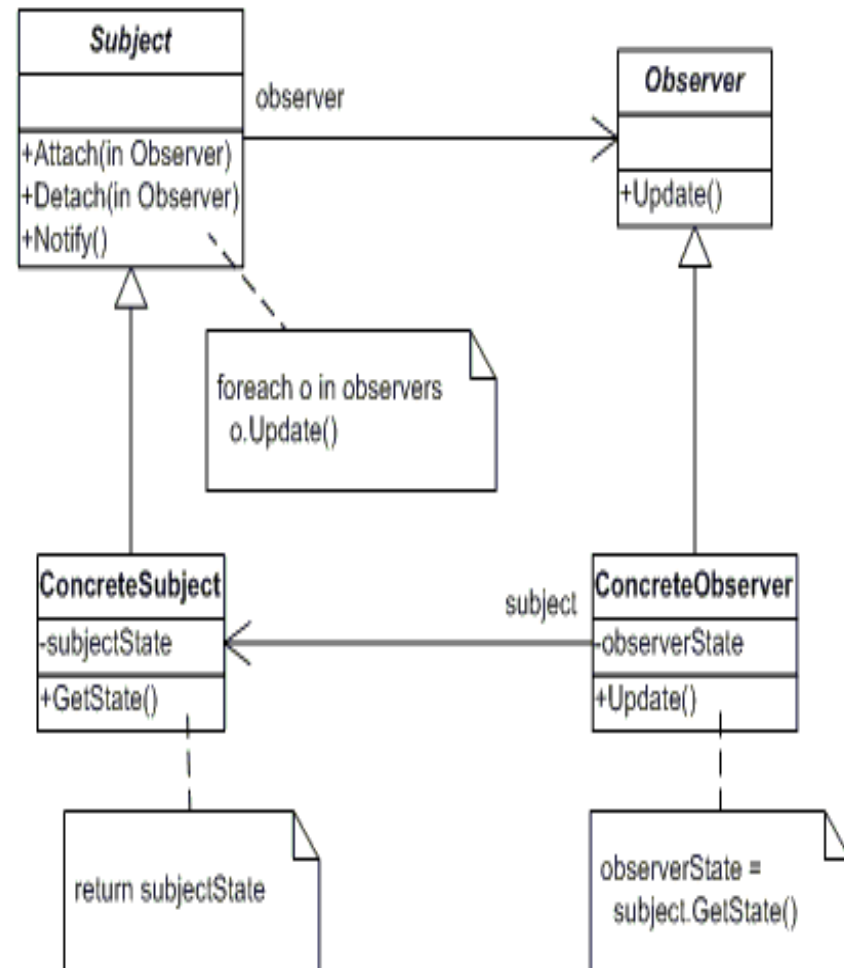
	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10



model

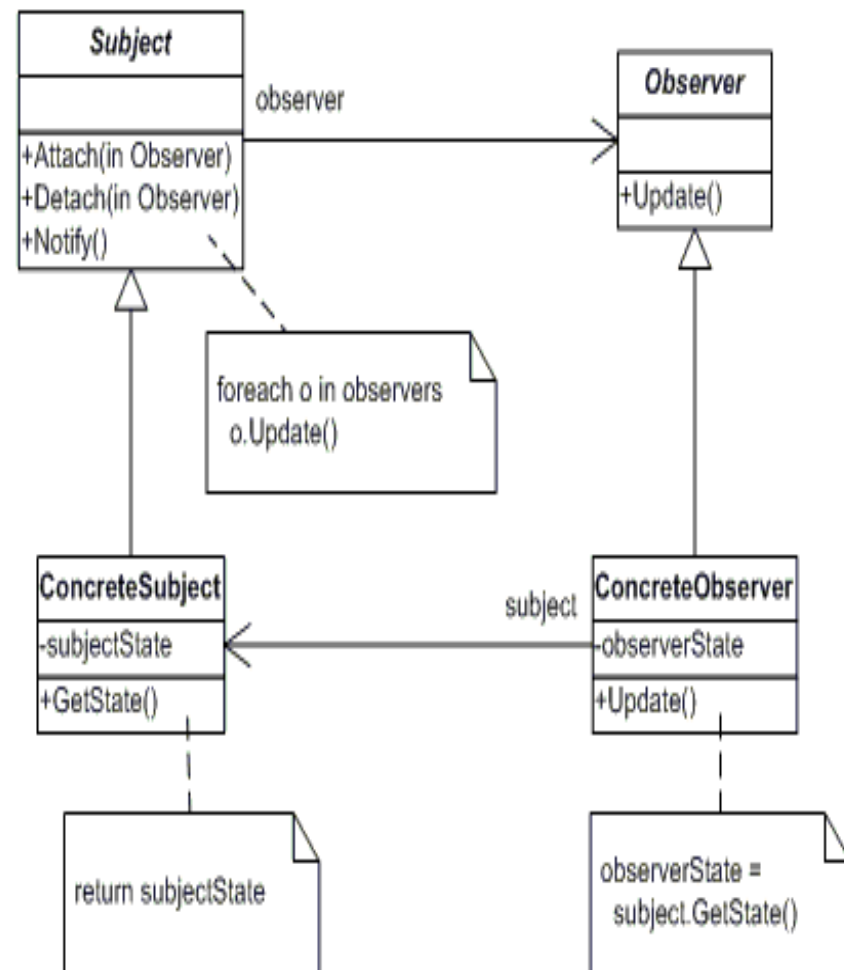
Model

- Enthält Daten und Geschäftslogik
- Implementiert Beobachter-Muster (aka Publish-Subscribe)
- Ist selbst das beobachtete Objekt
- Hat 1...n Beobachter => die Views



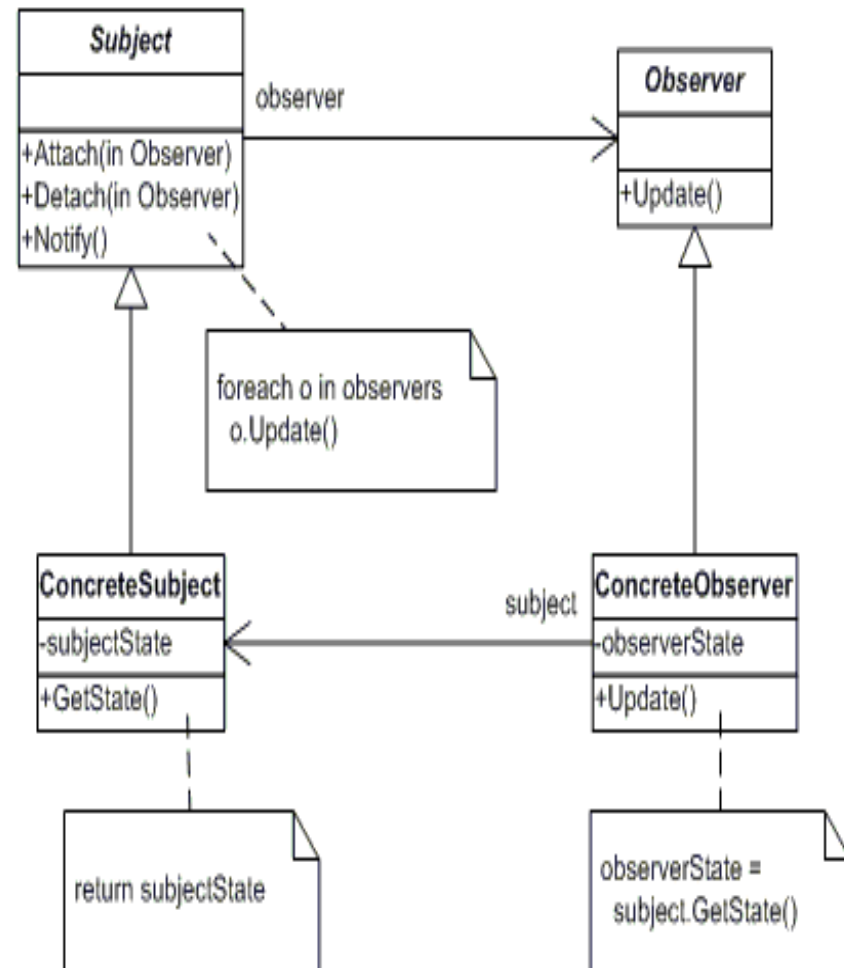
Model

- Jede Änderung der Daten verursacht Aktualisierung der Views
- Kennt die konkreten Views nicht (lose Kopplung)
- Besitzt Referenz auf View-Basisklasse



Model

- Bietet Mechanismus bei denen sich die Beobachter registrieren
- Datenänderung über update-Methode der Views



Model-Beispielcode

- Schnittstellen nach außen hier durch setter/getter
- Beim Ändern des Status wird Statusänderung nach außen sichtbar und Observer werden benachrichtigt

```
public class MyModel extends java.util.Observable {  
  
    private int value = 10;  
  
    public void setValue(int value) {  
        this.value = value;  
        setChanged();  
        notifyObservers();  
    }  
  
    public int getValue() {  
        return this.value;  
    }  
}
```

Model Vor- und Nachteile

- Vorteile

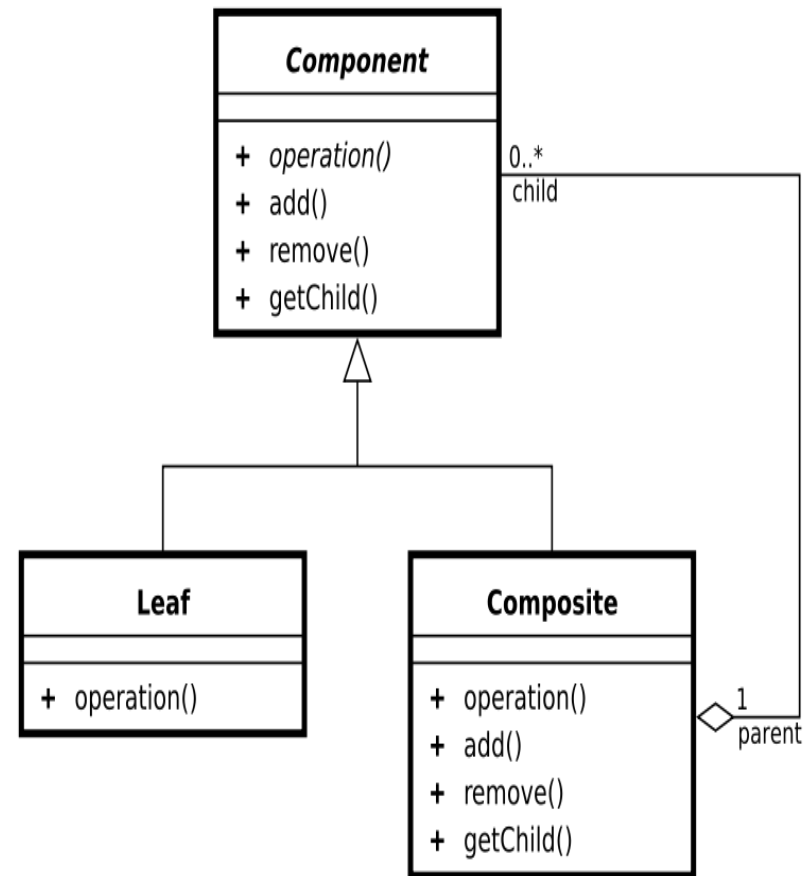
- Lose Kopplung zwischen Subjekt und Beobachter
- Unterstützung Broadcast-Kommunikation

- Nachteile

- Kommunikations-overhead
- Aktualisierungen können weitere Änderungen an Models hervorrufen

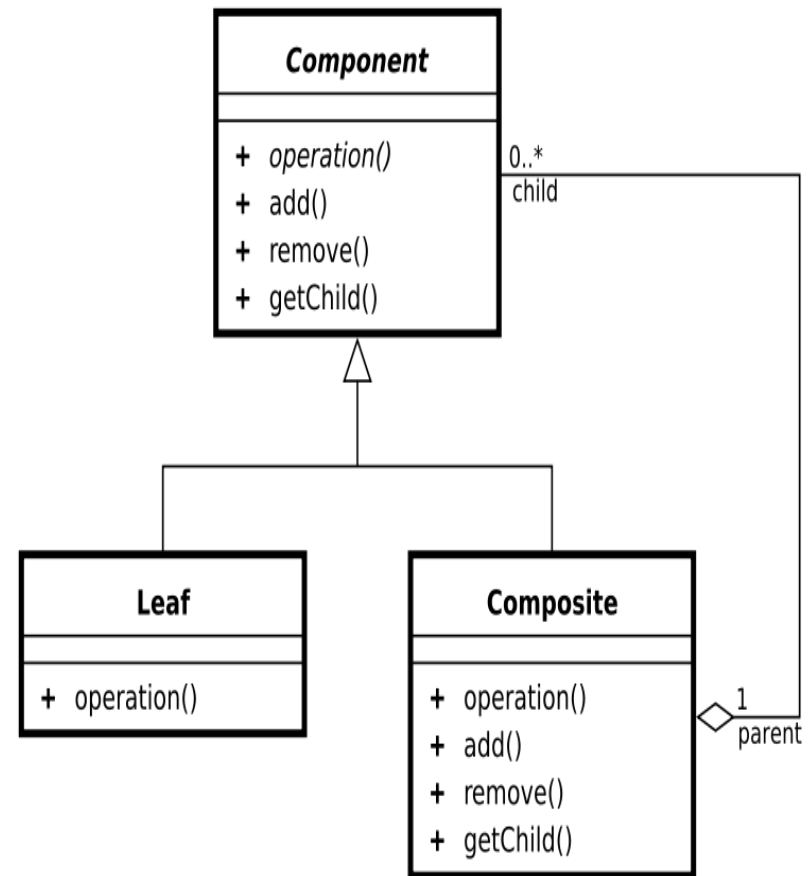
View

- (Graphische)
Darstellung der Daten
- Implementiert
Composite Pattern
- Interaktion mit dem
Nutzer
- Kann geschachtelt
werden



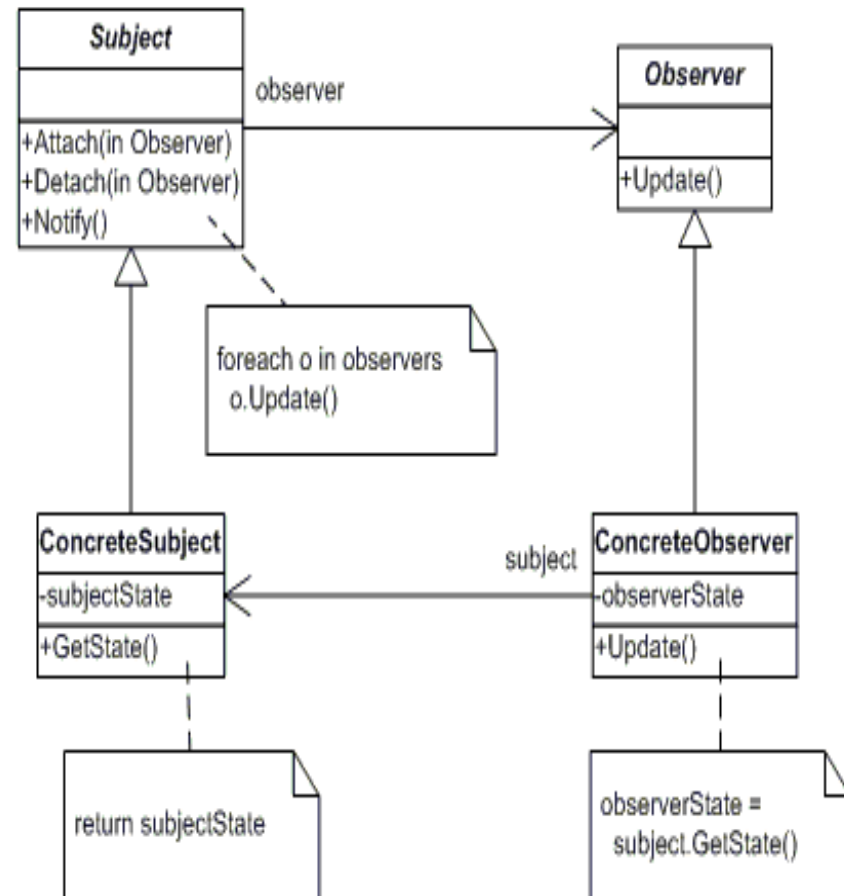
View

- Kennt Model
- Kennt Controller
- Aktionen des Nutzers werden an Controller weitergegeben



View

- Statusänderung über Abfrage am Model durch getState()
- Anschliessend Darstellung der geänderten Daten



View – Model Interaktion Beispiel

- Konkrete View-Implementierung von View-Basisklasse abgeleitet
- Diese sorgt für die Registrierung der Views am Model

```
public abstract class AbstractView implements java.util.Observer {  
  
    private MyModel model;  
    private Frame myFrame;  
    private TextField display = new TextField();  
    private Button upButton = new Button("Raise");  
    private Button downButton = new Button("Lower");  
  
    AbstractView(String label, MyModel model, int h, int v)  
    {  
        this.label = label;  
        this.model = model;  
        ...  
        model.addObserver(this); // Connect the View to the Model  
        ...  
    }  
    ...  
}
```

View-Model-Interaktion

- Views bieten Update-Methode, die vom Model bei jeder Statusänderung aufgerufen wird

```
- class SampleView extends AbstractView
- { public SampleView(MyModel model, int h, int v)
-   { super("sample view", model, h, v);
-     ...
-   }
-
-   public void update(Observable t, Object o) // Called from the Model
-   {
-       setDisplay("" + model().getValue());
-   }
- }
```

View Vor- und Nachteile

- Vorteile

- Einheitliche Behandlung von komplexen und primitiven Objekten
- Vereinfachung des Klienten
- Vereinfachung Hinzufügen neuer Komponenten

- Nachteile

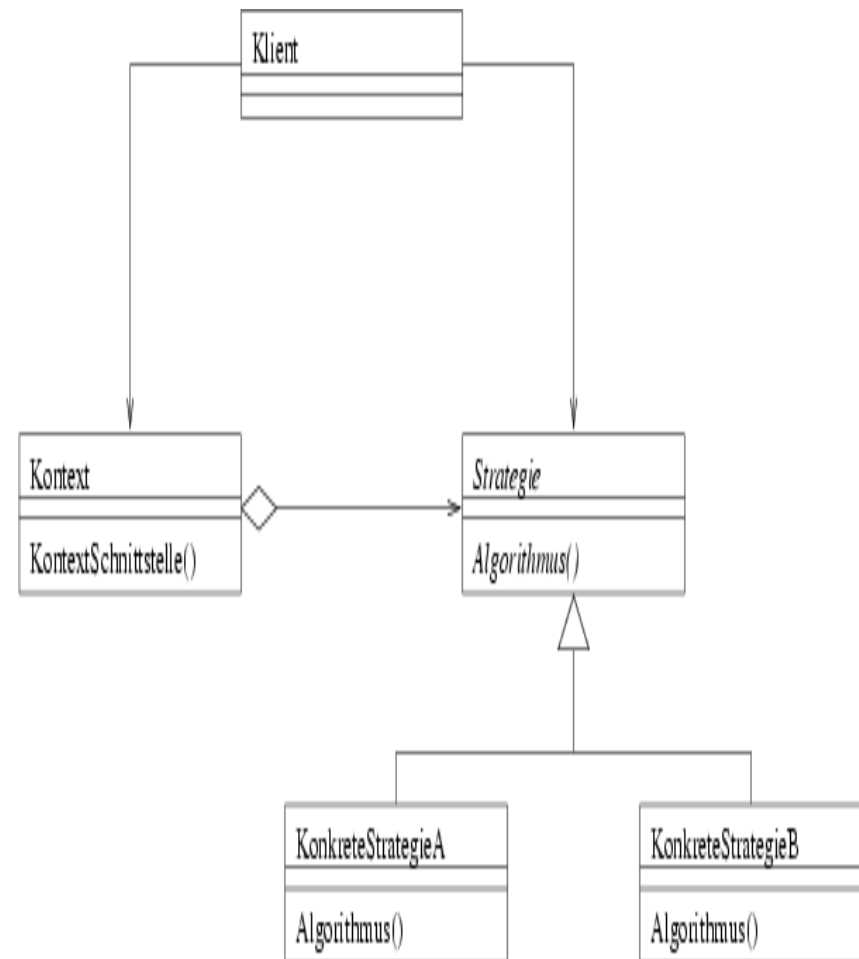
- Nicht garantiert zyklensfrei
- Zu allgemein, d.h. Typen der Komponenten schlecht einschränkbar

Controller

- Zwischenschicht zwischen Views und Model
- Reagiert auf Nutzereingaben (Tastatureingaben, Mausklicks, etc)
- Ermittelt ggf. weitere Views als Reaktion auf Benutzerinteraktion
- Ermittelt an den Daten die Models deren Daten sich ändern (Strategie Pattern)

Controller

- Kennt View und Model
- View stellt den Kontext dar
- Controller ermittelt an den Daten das von den Änderungen betroffene Model und nimmt ggf. Änderungen am View vor



MVC-Interaktion

- Listener sind hier die Controller
- Action führt zur Statusänderung des Models
- Views werden durch update-Mechanismus aktualisiert

```
class SampleView extends AbstractView
{
    public SampleView(MyModel model, int h, int v)
    {
        super("sample view", model, h, v);
        setDisplay(""+model.getValue());
        addUpListener(new UpListener());
        addDownListener(new DownListener());
        addDisplayListener(new DisplayListener());
        model.addObserver(this);
    }

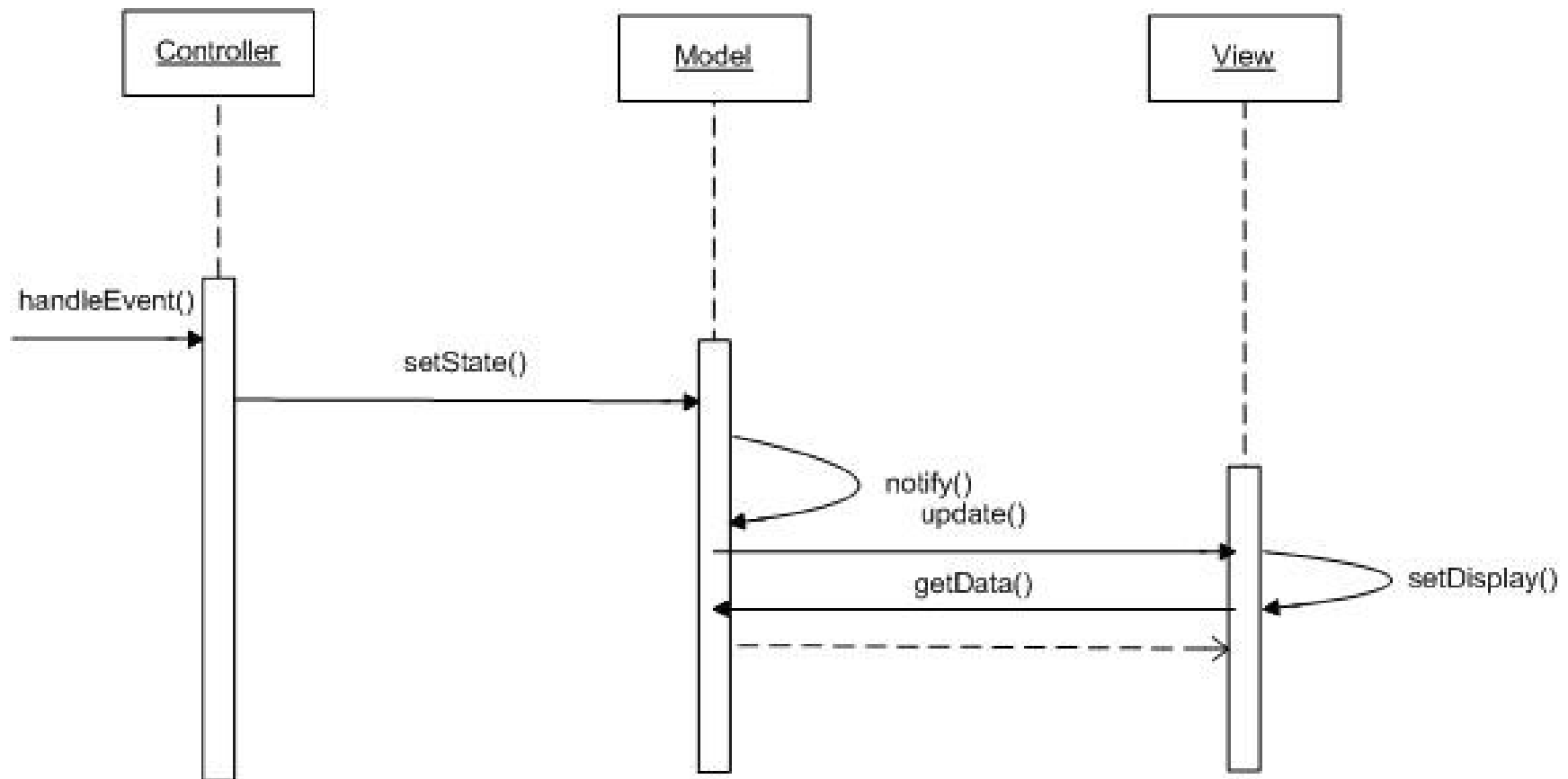
    public void update(Observable t, Object o)
    {
        setDisplay("" + model().getValue());
    }
}

class UpListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        model().setValue(model().getValue() + 1);
    }
}

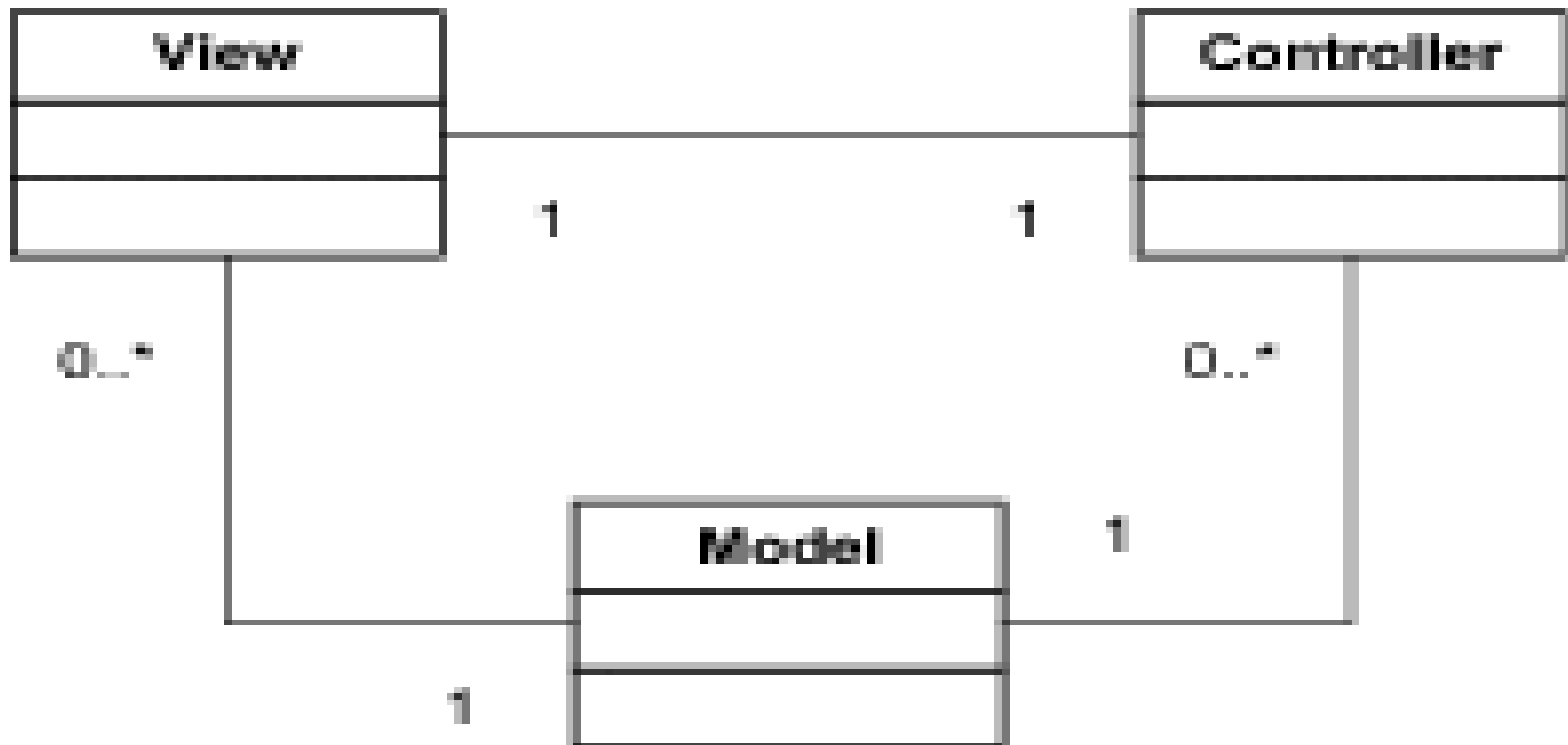
class DownListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        model().setValue(model().getValue() - 1);
    }
}

class DisplayListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        int value = getDisplay();
        model().setValue(value);
    }
}
}
```

MVC-Interaktion



MVC Struktur



Controller Vor- und Nachteile

- Vorteile

- Hohe Wiederverwendung von verwandten Algorithmen (im Bsp. ActionPerformed)
- Verringerung Bedingungsanweisungen
- Schneller Austausch bzw. Wartung der Algorithmen

- Nachteile

- Enge Kopplung an View
- Enge Kopplung an Modell

Vor- und Nachteile

Vorteile

- Mehrere Ansichten eines Models
- Austauschbarkeit von Views
- Leichtes Hinzufügen von Views
- Verbesserte Wartbarkeit und Wiederverwendung
- Synchronisierte Ansichten

Nachteile

- Hohe Änderungskosten bei vielen Observern, Overhead durch Information der Datenänderung an jeden Observer, auch wenn dieser die Daten evtl. gar nicht benötigt (z.B. Einfache Ansicht benötigt weniger Daten als Expertenansicht)
- ggf. Endlosschleifen, falls Observer wieder Änderungsoperationen aufruft
- Erhöhte Komplexität
- Enge Kopplung von Controller und View sowie View und Model

Zusammenfassung

- MVC trennt Logik von Darstellung
- Ist zusammengesetztes Muster (aus Observer, Composite, Strategy)
- Verbessert Wiederverwendung und Wartbarkeit
- Erhöht Komplexität

Quellen

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; Entwurfsmuster, Addison Wesley, 2004

Freeman, E; Robson, E.; Sierra, K.; Bates B;
Head First Design Patterns, o'Reilly Media 2004

Kerievsky, J; Refactoring to Patterns; Addison-Wesley, 2005

<http://java.sun.com/blueprints/patterns/MVC-detailed.html>
(zuletzt abgerufen am 16.06.2009)

<http://csis.pace.edu/~bergin/mvc/mvcgui.html>
(zuletzt abgerufen am 16.06.2009)

Vielen Dank für die Aufmerksamkeit