

# Model-View-Controller

SE3 Team

Juni 2009

Model-View-Controller

Analyse

Entwurf und Entwicklung

# Model-View-Controller

# Allgemeine Betrachtungen

## Zu graphischen Bedienoberflächen

- ▶ Interaktiv, nutzerfreundlich und komfortabel
- ▶ Haben sich in Software-Systemen durchgesetzt
- ▶ Heutige Akzeptanz und Verbreitung zeigt
  - ▶ Wichtiger Bestandteil von Anwendungssystemen
  - ▶ Interaktive SW-Systeme haben sehr hohen Stellenwert
- ▶ Architekturmuster **MVC**
  - ▶ Grundlegende strukturelle Organisation
  - ▶ Unabhängigkeit des funktionalen Teils von der Bedienschnittstelle

# Das MVC Muster

## Die Komponenten

Teilt eine interaktive Anwendung in 3 Komponenten auf.

### ► **Model**

- Enthält die gesamte Daten, Zustands- und Anwendungslogik
- Zustandsänderung über Schnittstelle
- Benachrichtigungen über Änderungen an Beobachter

### ► **View**

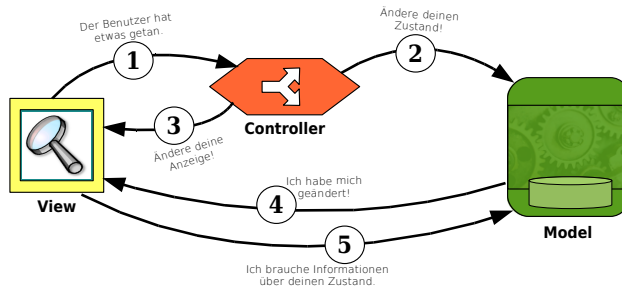
- Bildschirmrepräsentation des Anwendungsobjektes
- Erhält Zustand und Daten direkt vom Model

### ► **Controller**

- Nimmt Eingaben des Nutzers entgegen und verarbeitet sie

# Das MVC Muster

## Die Komponenten



View- und Controller beschreiben die Bedienschnittstelle.

# MVC etwas genauer betrachtet

## Das Observer-Muster

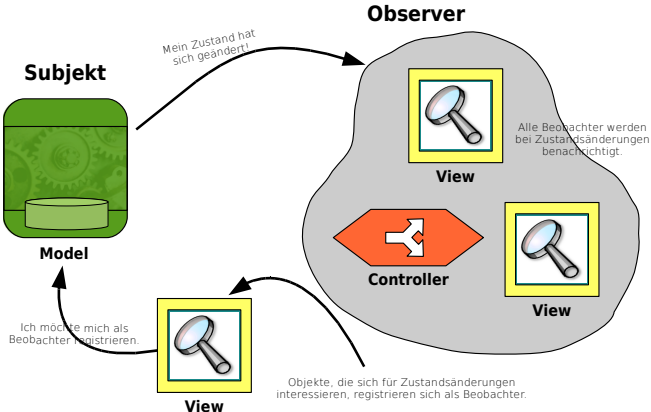
Wichtigstes Muster für Verständnis des MVC.

### ► Zweck

- Definiere eine 1-zu-n-Abhängigkeit, zwischen Objekten, so dass die Änderung des Zustands eines Objektes dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

# MVC etwas genauer betrachtet

## Das Observer-Muster

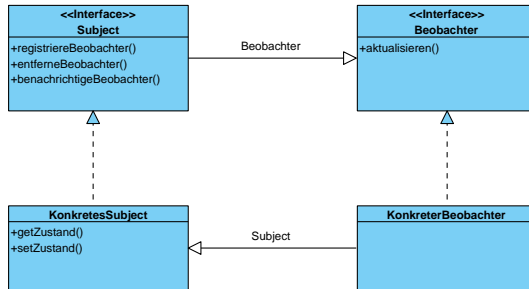


Es macht das Model völlig unabhängig von View und Controller.



# MVC etwas genauer betrachtet

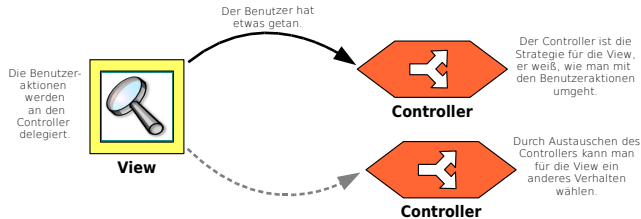
## Das Observer-Muster als Klassendiagramm



- ▶ Beobachter registriert sich beim Subjekt
- ▶ Subjekt fügt es Liste seiner Beobachter hinzu
- ▶ Subjekt benachrichtigt alle registrierten Beobachter
- ▶ Subjekt bietet Zugriff über Schnittstelle an

# MVC etwas genauer betrachtet

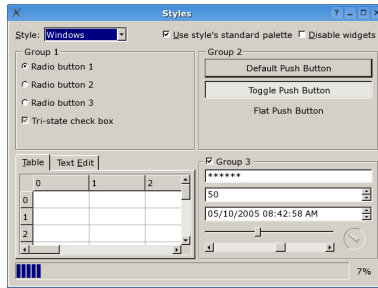
## Das Strategy-Muster



- ▶ View ist mit einer Strategie konfiguriert
- ▶ Controller ist das Verhalten der View
- ▶ Kann ausgetauscht werden
- ▶ View delegiert Benutzeraktionen an den Controller

# MVC etwas genauer betrachtet

## Das Composite-Muster



Die GUI ist ein Kompositum.

- ▶ Besteht aus Label, Buttons, Texteingabefelder, ...
- ▶ Komponenten enthalten andere Komponenten
- ▶ Wird intern verwendet um Bestandteile der Anzeige zu verwalten

# Nachteile von MVC

In bestimmten Fällen

- ▶ Größere Komplexität der Anwendung ohne Zugewinn an Flexibilität
- ▶ Potential für eine übermäßige Anzahl von Aktualisierungen
- ▶ Enge Verbindung zwischen View- und Controllerkomponenten

# Framework

## Ein kurzer Überblick

- ▶ Besteht aus einer Menge von zusammenarbeitenden Klassen
- ▶ Wiederverwendbarkeit für den Entwurf einer bestimmten Klasse von Software
- ▶ Definiert:
  - ▶ Die Struktur im Großen
  - ▶ Unterteilung in Klassen und Objekte
  - ▶ Die jeweiligen zentralen Zuständigkeiten
  - ▶ Zusammenarbeit und Kontrollfluß
- ▶ Legt Entwursparameter im voraus fest
- ▶ Komponenten beinhalten Erfahrungen und sind erprobt

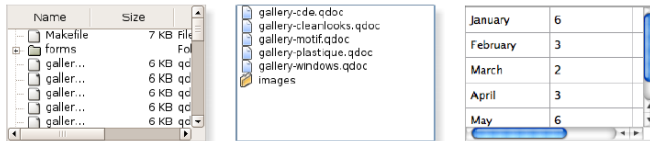
# Model/View Programmierung mit dem Qt Framework

Was ist Qt?

- ▶ De facto Standard C++ Framework für die Entwicklung von Cross-Platform-Software
- ▶ Enthält Widgets mit Standard GUI-Funktionalität
- ▶ Open Source Edition ist Grundlage von KDE

# Model/View Programmierung mit Qt

## Item Views



- ▶ Item-View-Widgets sind Standard GUI-Bedienungselemente
- ▶ List-, Tree-, Table-Views
- ▶ Äquivalente Model/View Komponenten
  - ▶ *QListView*
  - ▶ *QTableView*
  - ▶ *QTreeView*

# Model/View Programmierung mit Qt

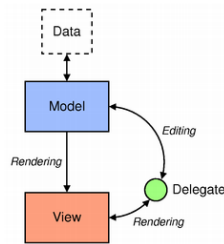
## Das Model/View Framework

- ▶ Variante des MVC speziell angepaßt für Qt's Item Views
- ▶ Verwendet Models um Daten anderen Komponenten zur Verfügung zu stellen
- ▶ Views präsentieren Daten
- ▶ Delegates behandeln Rendering- und Bearbeitungsprozesse
- ▶ Ermöglicht eine ganze Reihe Vorteile gegenüber den klassischen ItemViews



# Model/View Programmierung mit Qt

## Die Model/View Architektur



- ▶ Resultiert aus der Kombination von View und Controller in einer Komponente
- ▶ Dies ermöglicht einen Framework basierten Ansatz auf der Grundlage des MVC
- ▶ Mit Delegates kann man individuell auf Benutzereingaben reagieren

Mit *Proxy Models* können Daten von Models transformiert werden.

# Model/View Programmierung mit Qt

## Die Model/View Architektur

- ▶ **Model**
  - ▶ Kommuniziert mit Datenquelle
  - ▶ Bietet Standardinterface für Zugriff der anderen Komponenten
- ▶ **View**
  - ▶ Bekommt Model-Indizes vom Model
  - ▶ Diese referenzieren Daten-Items
- ▶ **Delegate**
  - ▶ Rendert die Daten-Items in View
  - ▶ Wird Item bearbeitet werden ebenfalls Model-Indizes verwendet

Komponenten werden von abstrakten Klassen definiert, welche Standardinterfaces anbieten.

# Model/View Programmierung mit Qt

## Die Model/View Architektur

Kommunikation der Komponenten mittels *Signals* und *Slots*<sup>1</sup>.

- ▶ Signals vom Model informieren View über Datenänderungen
- ▶ Signals von der View bieten Informationen über Benutzeraktionen auf Daten-Items
- ▶ Signals vom Delegate während der Editierung verwendet, um Model und View über aktuellen Bearbeitungszustand zu informieren

---

<sup>1</sup>Qt-Mechanismus für die Kommunikation zwischen Objekten

# Model/View Programmierung mit Qt

Weitere Informationen im Internet

**<http://www.qtsoftware.com>**

# Analyse

# Darstellung des Model-View-Controller- Konzeptes

- ▶ Welche Art der Applikation?
  - ▶ Fahrplan- Applikation?
    - ▶ Anzeige des Zugfahrplans gesamt
    - ▶ Anzeige des Fahrplans an bestimmter Haltestelle
    - ▶ *Gab es schon...*
  - ▶ Kinoinformation?
    - ▶ Anzeige der aktuell laufenden Filme
    - ▶ Anzeige der demnächst laufenden Filme
  - ▶ Wetterinformation? **weatherinfo**

# Anforderungen an die Beispielapplikation

- ▶ Welche Wetterdaten sollen dargestellt werden, für welchen Zeitraum und für welche Städte? **Model**
- ▶ Welche Anzeigearten wollen wir implementieren? **View**
- ▶ Welche Funktionalitäten in den Views sollen implementiert werden? *Controller*

# Das Model

- ▶ Was ist als Wetterinformation sinnvoll?
  - ▶ Temperatur
  - ▶ Bewölkung
  - ▶ Windstärke
  - ▶ Windrichtung
- ▶ Ein Zeitraum von 5 Tagen (*längere Vorhersagen grenzen an Wahrsagerei*)
- ▶ Welche Städte und welche Zusatzinformationen?
  - ▶ Dresden, Oslo, Springfield, ...
  - ▶ Weltkoordinaten (Längen- und Breitengrad) für die Ortsbestimmung



# Die Views

- ▶ Welche Views?
  - ▶ Verlaufskurve der Temperatur: **temperature\_view**
  - ▶ Wetterinformationen für eine bestimmte Stadt: **day\_view**
  - ▶ Anzeige der Temperatur und Bewölkung in Tabellenform: **table\_view**
  - ▶ Anzeige der Bewölkung auf einer Weltkarte: **world\_view**
- ▶ Design der Oberflächen der Views
  - ▶ Per Handzeichnung im ersten Schritt diskutiert und definiert
  - ▶ Nachfolgend dann von Implementierer durch das Framework realisiert

# Die Funktionalität der Views

- ▶ **temperature\_view**
  - ▶ Auswahl der Stadt
- ▶ **day\_view**
  - ▶ Auswahl der Stadt
  - ▶ Auswahl des angezeigten Tages
- ▶ **table\_view**
  - ▶ Einschränkung der angezeigten Städte durch einen Filter
    - ▶ Filter soll case-insensitive sein
  - ▶ Änderung der Temperatur für eine Stadt und einen Tag
    - ▶ Temperatureintrag soll editierbar werden nach einem Doppelklick
- ▶ **world\_view**
  - ▶ Auswahl des angezeigten Tages

# Entwurf und Entwicklung

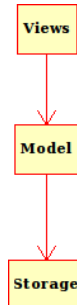
# Allgemeines zum Entwurf

- ▶ Ausgehend von Analyse
- ▶ Kleines Projekt
- ▶ Keine "Kundenwünsche"

# Grobentwurf

- ▶ Grobentwurf durch MVC impliziert
- ▶ Unterteilung in
  - ▶ Views
  - ▶ Model
  - ▶ Storage
- ▶ 3-Schichten-Architektur

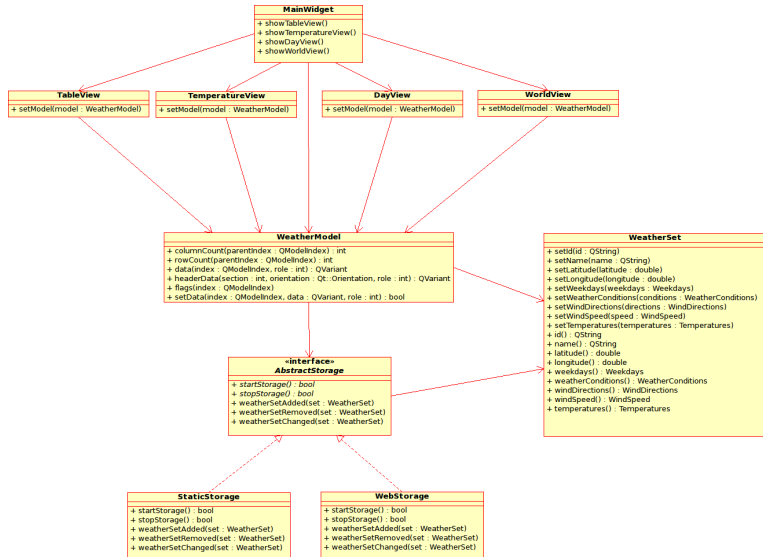
# Grobentwurf



# Feinentwurf

- ▶ Verfeinerung des Views-Moduls
- ▶ Definition der Model-Eigenschaften
- ▶ Festlegung der Strukturen zum Datenaustausch
- ▶ Definition des Storage-Interfaces

# Feinentwurf





# Allgemeines zur Entwicklung

- ▶ Basiert auf C++/Qt
  - ▶ Entwicklung unter Linux
  - ▶ Produkt lauffähig unter MS Windows
- ▶ Nutzung des MVC-Frameworks von Qt

# Demo

Programmvorführung