Software-Architektur

Dozent: Florian Glufke

Inhalt

- Allgemeines zu Software-Architektur
- Schichtenarchitektur
- MVC
- MVVM
- REST

Allgemeines

- Was ist eine Software-Architektur?
  - Beschreibt die grobe Struktur der Software
  - Beschreibt den Aufbau der Software als Ganzes
  - Beschreibt Komponenten und wie sie interagieren
- Abgrenzung Software-Architektur / Design Pattern
  - Design Patterns sind feingranularer
  - Beschreiben die Details einer Architektur
  - Architektur verwendet eine Vielzahl von Design Patterns

Allgemeines

- Wieso braucht man eine Software-Architektur
  - Bietet klare Struktur
  - Einheitliche Umsetzung
  - Weiterentwicklung der Software ist weniger aufwendig
  - Änderungen und Erweiterungen sind einfach
  - Sichert die Erweiterbarkeit über den gesamten Lebenszyklus

Schichtenarchitektur

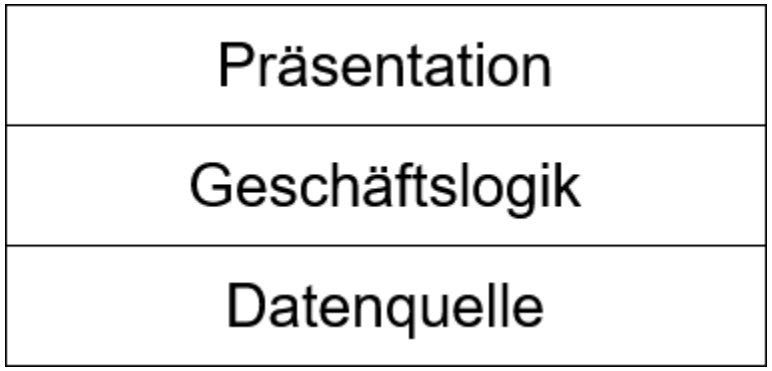
- Weitverbreitetes Architekturmuster
  - Computer-Protokolle basieren oft auf diesem
  - Bekanntester Vertreter TCP/IP
  - Client-Sever
- Aufbau
  - Komponenten der Software werden in einzelne Schichten (tier oder layer) aufgeteilt
  - Es gibt nur Abhängigkeiten von einer Schicht zur nächsten darunterliegenden Schicht
  - Abstraktionsgrad nimmt in höheren Schichten zu

Schichtenarchitektur

- Vorteile
  - Einzelne Schichten können einfach ausgetauscht werden
  - Wenig Abhängigkeiten zwischen den Schichten
  - Anwendung kann über mehrere Computer verteilt werden
- Nachteile
  - Änderungen an unteren Schichten ziehen ggf. Anpassungen in allen darüberliegenden Schichten mit sich

Schichtenarchitektur

- Einfache Schichtenarchitektur für eine Applikation



Schichtenarchitektur

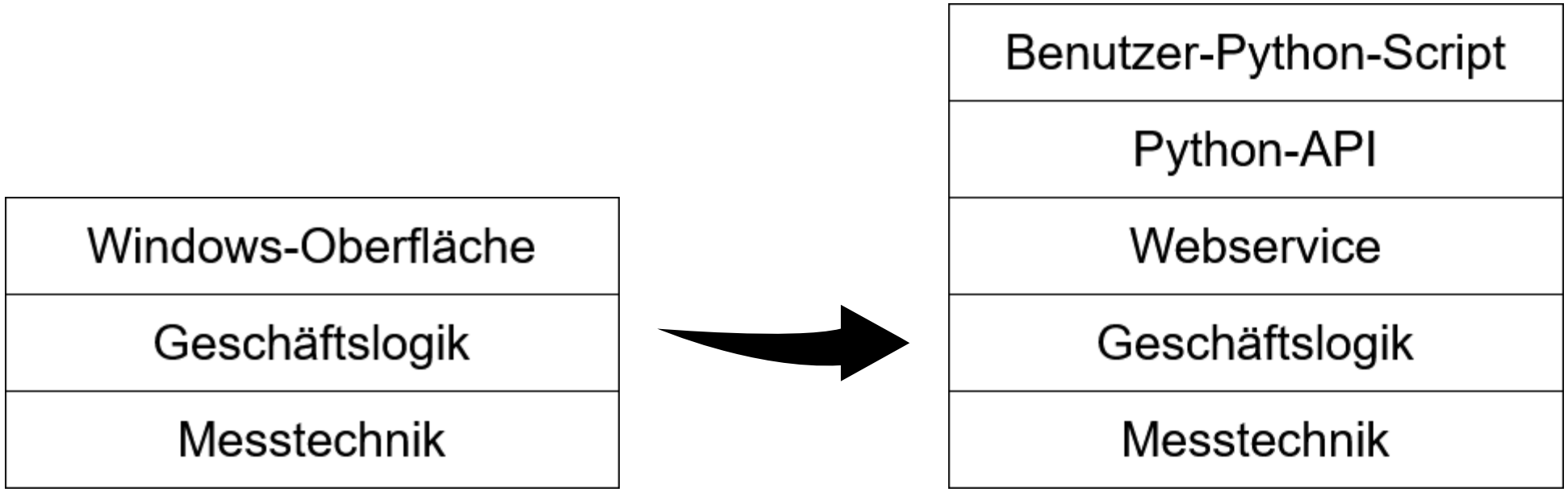
- Präsentationsschicht
  - Zeigt dem Anwender Informationen an
  - Nimmt Benutzereingabe entgegen und leitet diese weiter
  - Kann z.B. eine WinForms/WPF-Oberfläche sein oder eine HTML-Seite
- Geschäftslogik
  - Stellt den Kern der Anwendung dar
  - Bildet die Geschäftsprozesse ab
- Datenquelle
  - Kapselt meistens die Datenbank
  - Kommunikation mit anderen Softwaresystemen

Schichtenarchitektur

- Thin-Client vs. Fat-Client
  - Entscheidet wo die Geschäftslogik enthalten ist
  - Thin-Client besteht nur aus Präsentationsschicht
  - Fat-Client enthält zusätzlich die Geschäftslogik

Schichtenarchitektur

- Beispiel: Erweiterung einer Anwendung um eine Python-API



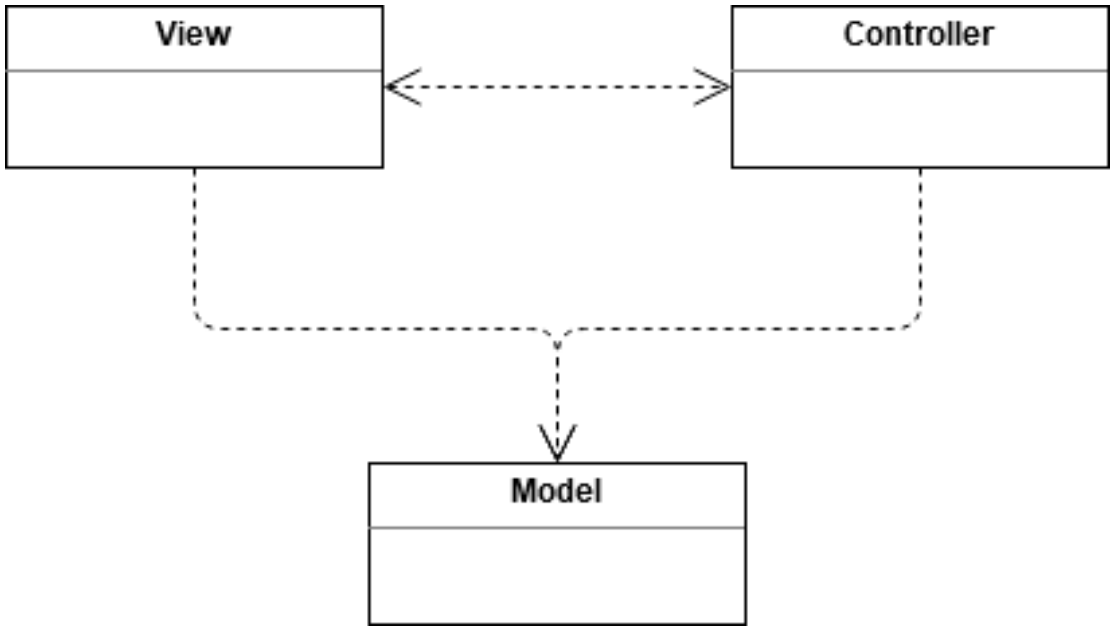
MVC

- Architektur für Anwendungssoftware
- Entwickelt in den 80er-Jahren von Trygve Reenskaug
- Implementierung in SmallTalk-80
- Bis heute in unterschiedlichen Ausprägungen aktuell

MVC

- Besteht aus 3 verschiedenen Komponenten
- Model
  - Daten und Geschäftslogik
- View
  - Darstellung auf dem Bildschirm
- Controller
  - Steuerung durch den Benutzer

MVC – Abhängigkeiten



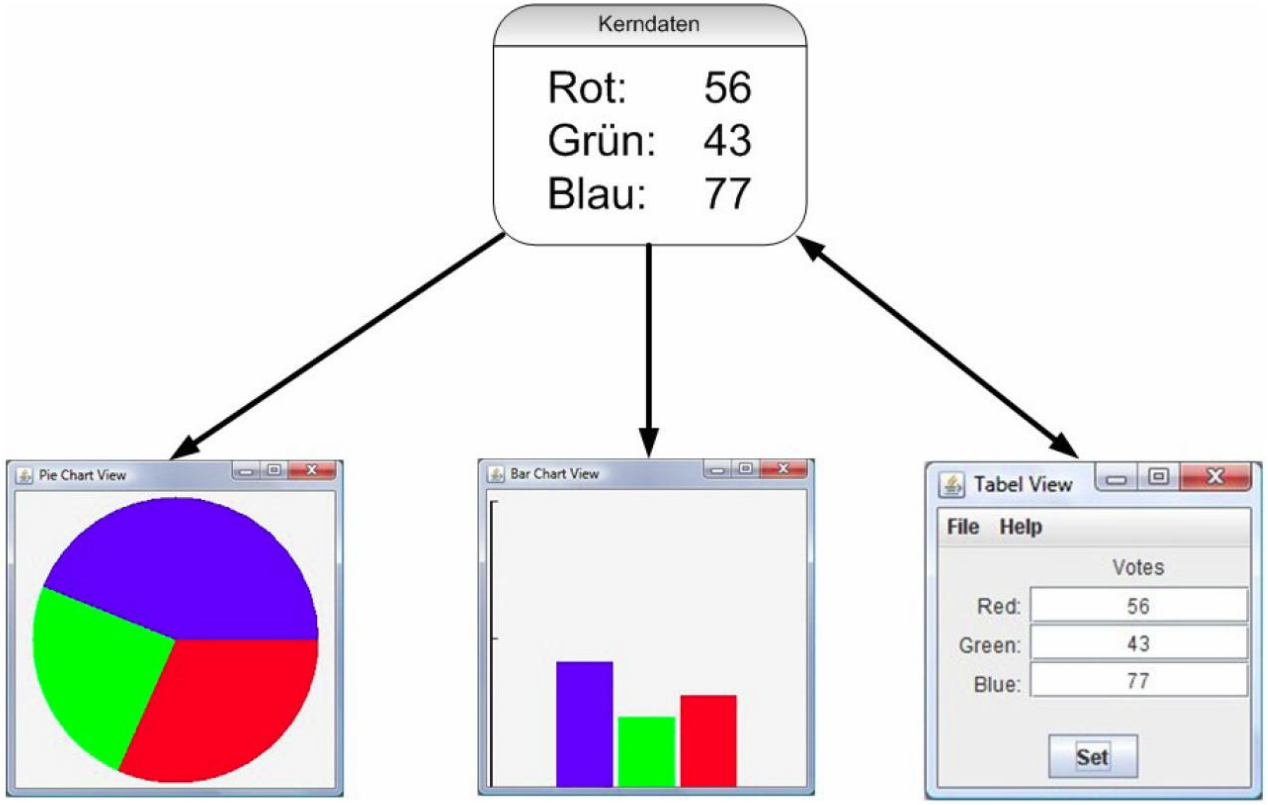
MVC – Eigenschaften

- Entkopplung der 3 Komponenten
- Unabhängigkeit des Models
- Benachrichtigungsmechanismus
- Verbesserung von Flexibilität, Wartbarkeit und Wiederverwendbarkeit

MVC – Eigenschaften

- Oberfläche kann einfach ausgetauscht werden
- Unterschiedliche Darstellung derselben Daten
- Alle Darstellungen werden sofort aktualisiert

MVC – Eigenschaften



MVC – Model

- Kapselung der Daten
- Implementierung der Anwendungslogik
- Benachrichtigung der angemeldeten Views bei Änderung der Daten (Active Model)
- Ermöglicht Registrierung von Views
- Bietet Möglichkeiten zur Datenabfrage und Datenänderung

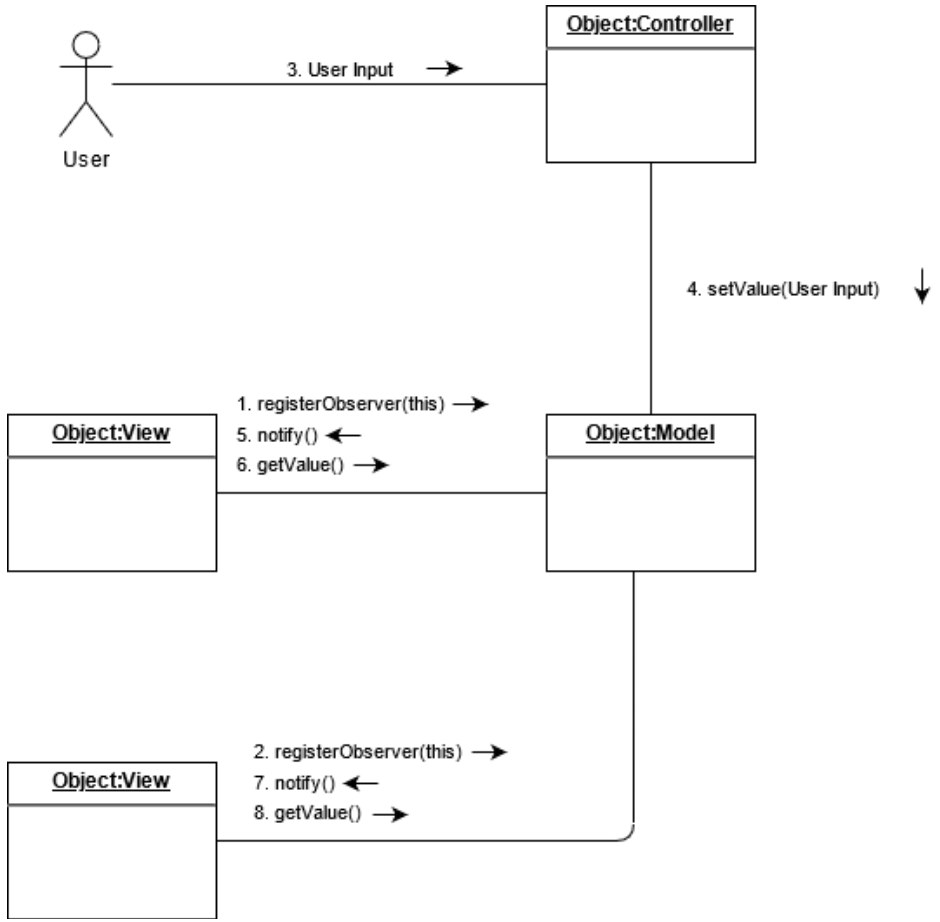
MVC – View

- Darstellung bestimmter Daten des Models
- An- und Abmeldung beim Model
- Aktualisierung der Darstellung nach Änderung der Daten
- Weiterleiten von Daten und Ereignissen an den Controller

MVC – Controller

- Verarbeitung der Eingabe des Benutzers
- Änderung der Daten des Models
- Benachrichtigung der Views (Passive Model)
- Anpassung des Zustandes der View

MVC – Interaktion



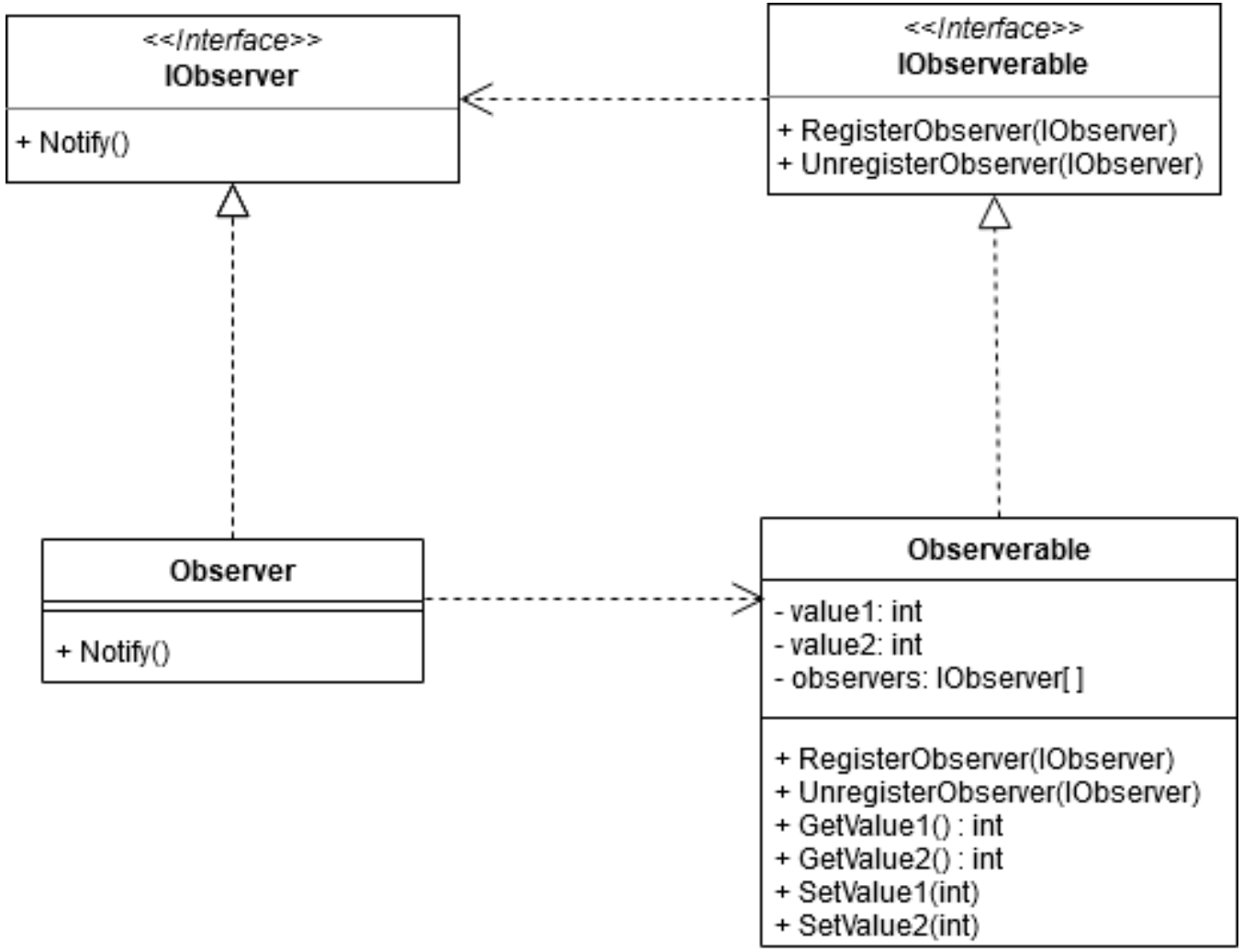
MVC – Design Patterns

- MVC verwendete verschiedene Design Patterns:
  - Observer (Beobachter)
  - Strategy (Strategie)
  - Composite (Kompositum)
  - …

MVC – Observer-Pattern

- Entkopplung von Model und View
- Änderungen an der View haben keine Auswirkung auf Model
- Model ist Observable
- View ist Observer
- View meldet sich beim Model an und ab
- Wird auch Publisher-Subscriber-Pattern genannt

MVC – Observer-Pattern



MVC – Observer-Pattern

- Model kennt nur IObserver
- Model bleibt unabhängig von Observer (View)
- View ist abhängig von Observable (Model)
- Nachteil: View aktualisiert sich unter Umständen unnötig

MVC – Observer-Pattern

- Codebeispiel für Registrierung von View beim Model:

```
View(Model model) {
        this.model = model;
        model.RegisterObserver(this);
}
```

MVC – Observer-Pattern

- Codebeispiel für Benachrichtigung der View vom Model:

```
void SetValue1(int value) {
	this.value = value;
	for(IObserver observer : observers){
		observer.Notify();
	}
}
```

MVC – Observer-Pattern

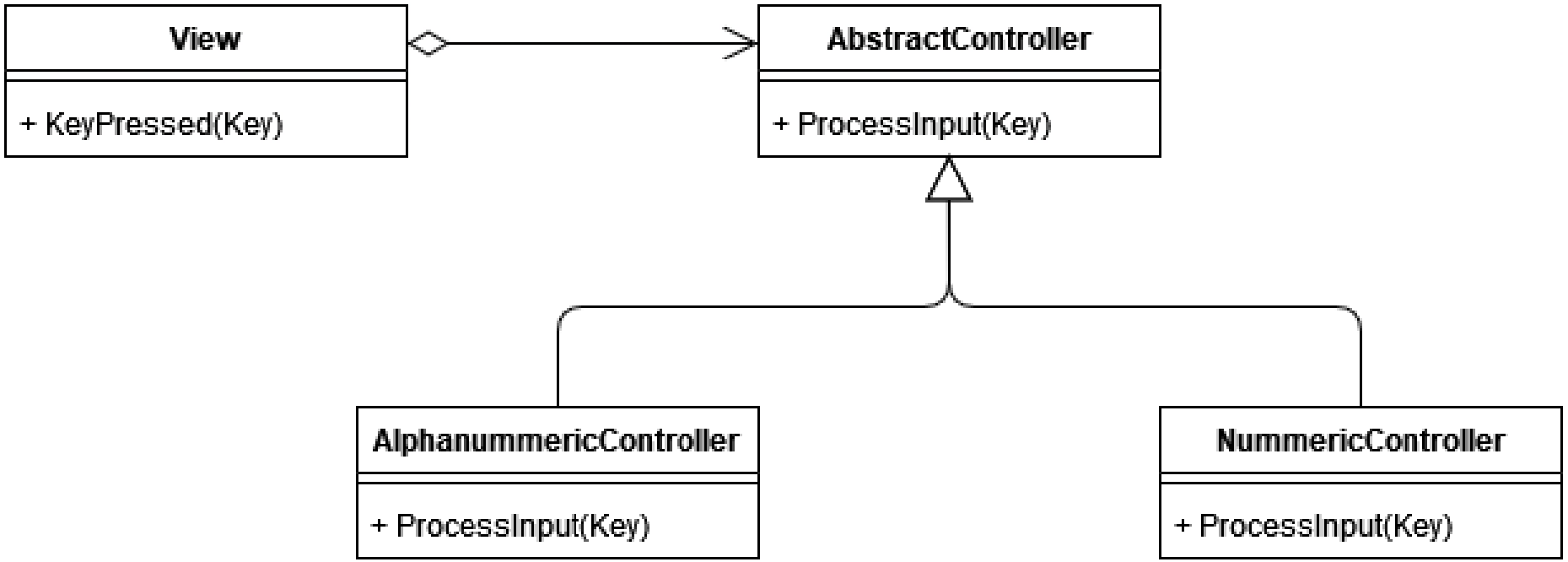
- Codebeispiel für Aktualisierung der View:

```
void Notify() {
	bars[0].height = model.GetValue1();
	bars[1].height = model.GetValue2();
	…
	Redraw();
}
```

MVC – Strategy

- Realisierung der Beziehung zwischen View und Controller
- Controller entspricht der Strategie der View
- Strategie ist die Logik für die Interpretation der Benutzereingabe
- Austausch des Controllers zur Laufzeit möglich

MVC – Strategy



MVC – Strategy

- Codebeispiel Controller Registrierung:

```
SetController(AbstractController controller) {
	this.controller = controller;
}
...
AlphanummericController controller = new AlphanummericController();
textField.SetController(controller);
```

MVC – Strategy

- Codebeispiel View verwendet Controller:

```
void KeyPressed(Key pressedKey) {
       controller.ProcessInput(pressedKey);
}
```

MVC – Strategy

- Codebeispiel AlphanummericController

```
void ProcessInput(Key pressedKey) {
	if(isAlphanummeric(pressedKey)){
		model.SetValue(
		pressedKey.GetChar());
	}
	if(isBackspace(pressedKey)) {
		model.SetValue(‘‘)
	}
}
```

MVC – Strategy

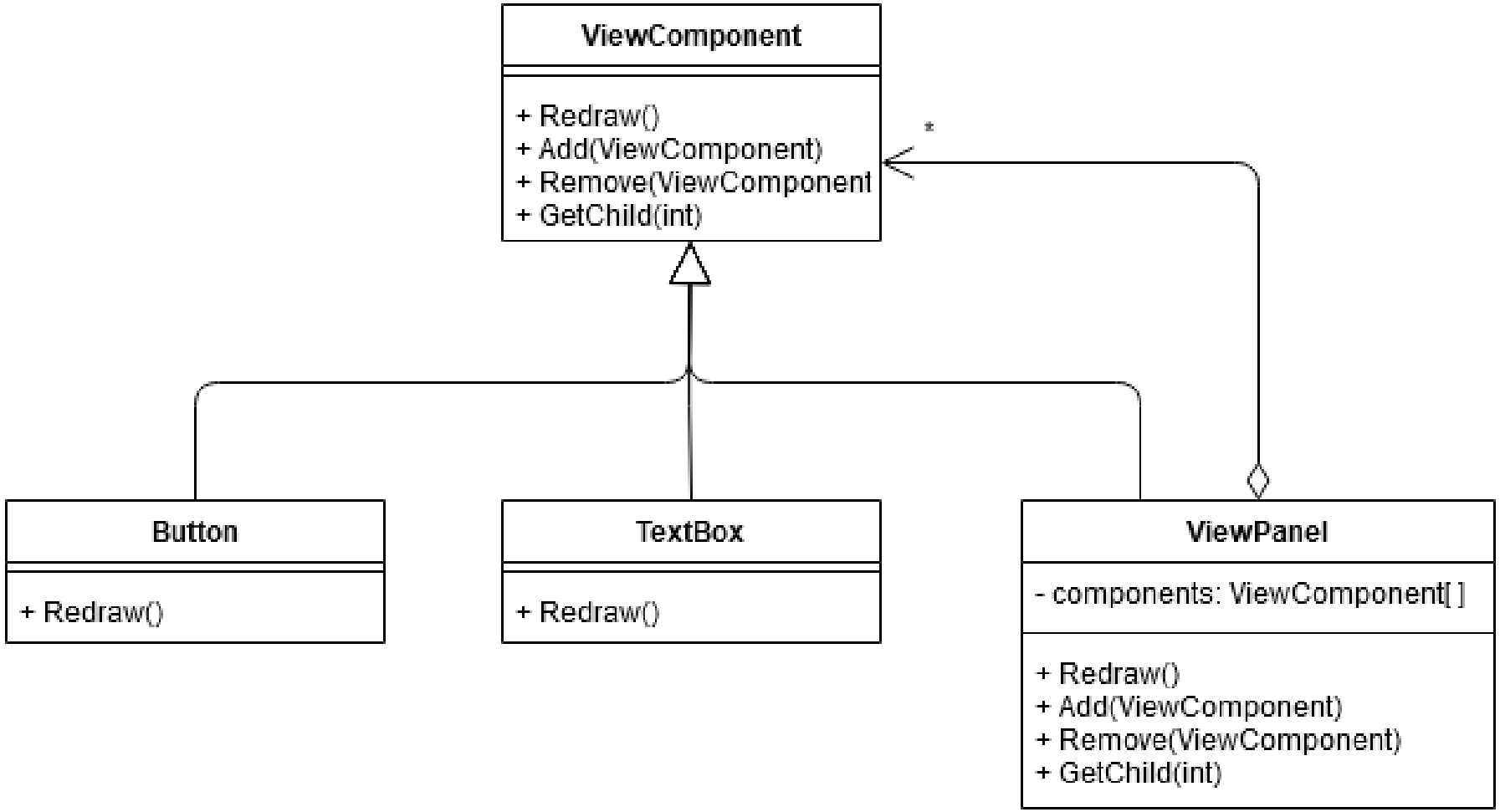
- Codebeispiel NummericController:

```
void ProcessInput(Key pressedKey) {
	if(isNummeric(pressedKey)){
		model.SetValue(pressedKey.GetChar());
	}
	if(isBackspace(pressedKey)) {
		model.SetValue(``)
	}
}
```

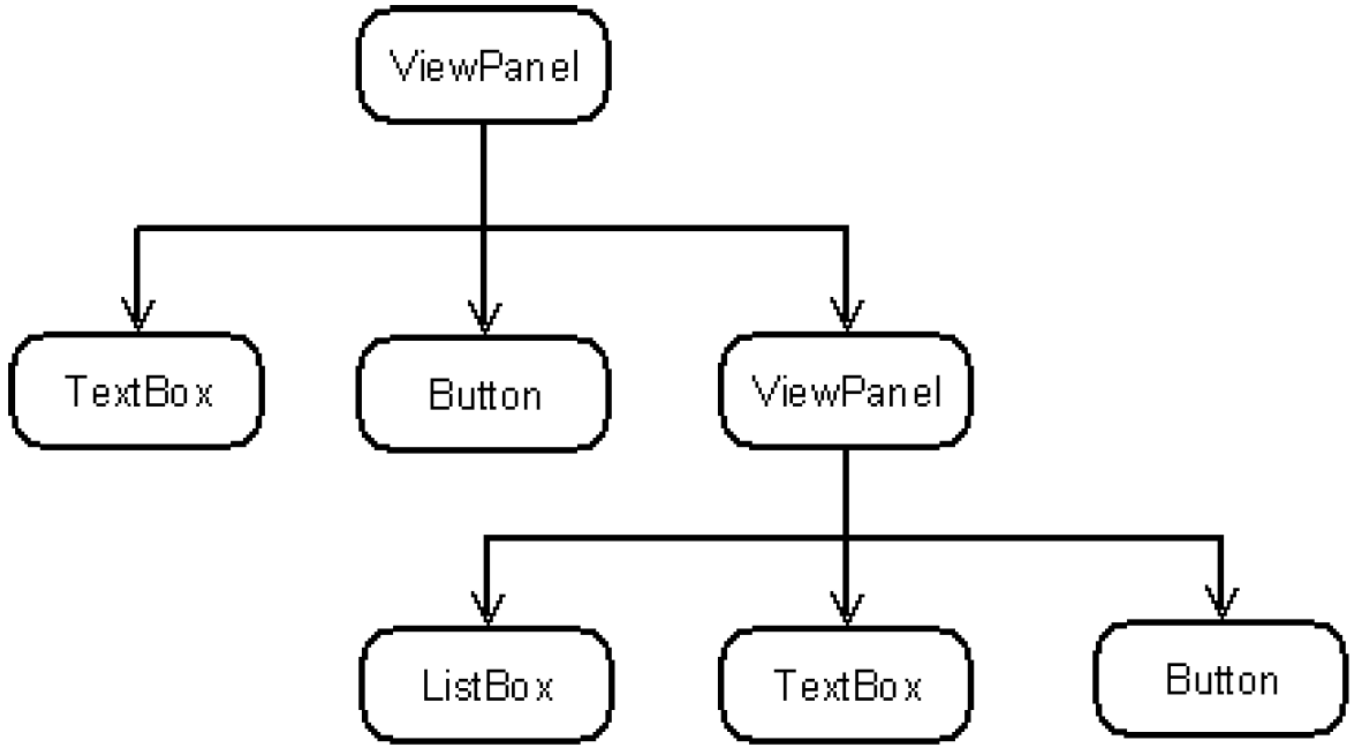
MVC – Composite

- Erlaubt das Verschachteln von View-Elementen
- Baumstruktur mit View-Elementen
- Gemeinsame abstrakte Basisklasse
- Delegation von Methodenaufruf an untergeordnete Elemente (z. B. Resize, Draw…)

MVC – Composite



MVC – Composite



MVC – Composite

- Codebeispiel Struktur erstellen:

```
ViewPanel viewPanel1 = new ViewPanel();
ViewPanel viewPanel2 = new ViewPanel();
Button button1 = new Button();
TextBox textBox1 = new TextBox();
...
viewPanel1.add(viewPanel2);
viewPanel2.add(button1);
viewPanel2.add(textBox1);
```

MVC – Composite

- Codebeispiel Delegation:

```
void Redraw() {
       drawBorder();
       for(ViewComponent child : childs){
              child[i].Redraw();
       }
}
…
void Redraw() {
       drawText(model.GetText());
}
```

MCV im Einsatz

- ASP.NET
  - .NET-Erweiterung zum Erstellen von Webseiten
- Symfony
  - PHP-Framework zum Erstellen von Webseiten
- JavaFX
  - GUI-Bibliothek für Java

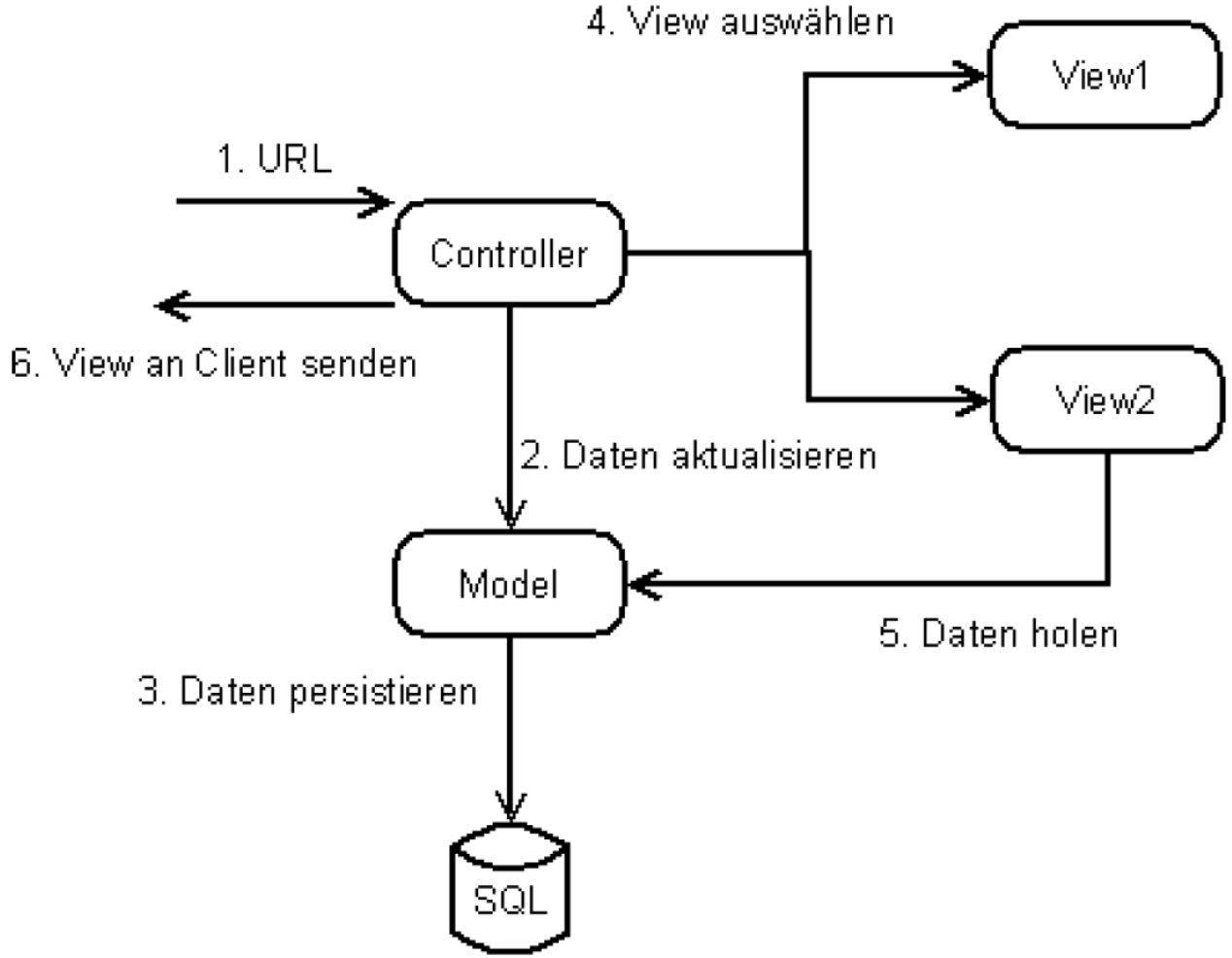
MVC – ASP.NET

- APS.NET MVC zum Erstellen von Webseiten
- Vereinfacht Handhabung von Komplexität
- Ermöglicht Tests einzelner Komponenten
- Geeignet für große Entwickler-Teams

MVC – ASP.NET

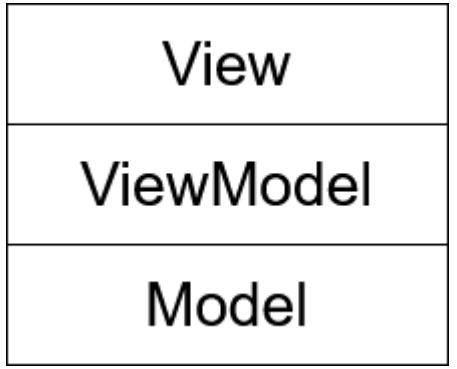
- Controller:
  - Verarbeitet HTTP Request
  - Definiert Ablaufsteuerung
- View:
  - HTML
  - Scripts
- Model:
  - Anwendungslogik
  - Datenzugriffslogik

MVC – ASP.NET


4. View auswählen
View1
1. URL
Controller
6. View an Client senden
View2
2. Daten aktualisieren
Model
5. Daten holen
3. Daten persistieren
SQL

MVVM

- Model-View-ViewModel ist eine Variante von MVC
- Einsatz z.B. bei:
  - WPF (Windows Presentation Foundation)
  - knockout.js
- View:
  - Rein deklarativ (XML oder HTML)
  - Command- und Data-Binding
  - Keine Logik



MVVM

- ViewModel:
  - Abstraktion des Models
  - Stellt Befehle und Daten zur Verfügung
  - Benachrichtigt View bei Änderungen
  - Ermöglicht Testbarkeit mit Unit-Tests
- Model:
  - Stellt Daten zur Verfügung
  - Benachrichtigt ViewModel bei Änderung
  - Beinhaltet Geschäftslogik

MVVM – WPF

- View wird mit XAML (Extensible Application Markup Language) definiert
- XAML basiert auf XML
- ViewModel-Klassen sind die Abstraktion der Model-Klassen
- ViewModel-Klassen stellen Daten und Befehle (Commands) passend für die View zur Verfügung
- ViewModel-Klassen delegieren Befehle und Änderungen an den Daten an die Model-Klassen
- Model-Klassen enthalten die Daten und Geschäftslogik

MVVM – knockout.js

- Einsatz von HTML5 und Javascript
- Ermöglicht deklaratives Databinding
- Website wird automatisch aktualisiert bei Datenänderungen
- View ist eine HTML-Seite
- ViewModel ist ein reines Java-Script-Objekt
- Model existiert serverseitig

REST

- REST steht für Representational State Transfer
- Architektur für Webservices
- Basiert auf bestehenden Internettechnologie
  - HTTP
  - URI/URL
  - XML/JSON/HTML
- Stellt Regel zur Realisierung von Webservices zur Verfügung
- Werden diese eingehalten kann ein Webservices als RESTful bezeichnet werden

REST

- REST basiert auf 5 Grundprinzipien:
  - Ressourcen mit eindeutiger Identifikation
  - Verknüpfungen/Hypermedia
  - Standardmethoden
  - Unterschiedliche Repräsentationen
  - Zustandslose Kommunikation

REST

- Ressourcen mit eindeutiger Identifikation
  - Ressourcen sind Daten, welche vom Webservice zur Verfügung gestellt werden
  - Weltweit eindeutige Identifikation wird über URI erreicht
  - Ressourcen können einzelne Entitäten oder auch Mengen sein
  - Beispiele:
    - http://mycompany.de/customers/100
    - http://mycompany.de/products

REST

- Verknüpfungen/Hypermedia
  - Ressourcen werden mit Links verknüpft
  - Ermöglicht die Angabe von Ressourcen anderer Applikationen
  - Ermöglichen den Programmablauf zu steuern
  - HTML-Formulare definieren die Struktur der Daten
  - Client weiß dadurch wie ein Request an den Server aufgebaut sein muss

REST

- Standardmethoden
  - Ressourcen können mit verschiedenen Methoden aufgerufen werden
  - Hierfür werden die bestehenden Verben aus HTTP verwendet
  - Jede Ressource hat somit die gleiche Schnittstelle
  - HTTP-GET liefert immer die Repräsentation (Daten) der Ressource
  - Weitere Verben sind: POST, PUT, DELETE, HEAD, OPTIONS
  - Regeln für die einzelnen Verben müssen eingehalten werden
  - Alle Ressourcen müssen das Standardanwendungsprotokoll (HTTP) implementieren

REST

- Unterschiedliche Repräsentationen
  - Eine Ressource kann unterschiedlich repräsentiert werden
  - Die Daten einer Ressource können z.B. als HTML zur Anzeige im Browser repräsentiert werden
  - Oder als XML damit eine andere Applikation die Daten verarbeiten kann

REST

- Zustandslose Kommunikation
  - Server hält keinen Zustand der Clientkommunikation über mehrere Anfragen hinweg
  - Verringert die Kopplung zwischen Server und Client
  - Erhöht die Skalierbarkeit
  - Jede Anfrage ist unabhängig von vorigen Anfragen
  - Jede Anfrage enthält sämtliche Daten, die sie benötigt

REST

- Identifikation der Ressourcen aus den Anforderungen
  - z.B. Kunde, Lieferadresse, Bestellung usw.
- Abbilden der Anforderungen auf HTTP-Verben
  - z.B. GET auf Ressource Orders, um alle Bestellung abzufragen
  - z.B. POST auf Ressource Orders, um eine Bestellung anzulegen

REST – Ressourcen

- Zentrales Konzept von REST
- Werden über URI eindeutig identifiziert
- Haben ein oder mehrere Repräsentationen
- Über diese werden sie nach außen zur Verfügung gestellt und bearbeitet
- Die Auswahl der Repräsentation erfolgt mittels Content Negotiation (Mechanismus im HTTP-Protokoll)

REST – Ressourcen

- Das Design der Ressourcen ist eine Kernaufgabe bei der Entwicklung von REST-Schnittstellen
- Wichtige Ressourcen sind die fachlichen Konzepte, werden auch Primärressourcen genannt. Z.B. bei ERP-System: Kunde, Angebot, Rechnung, Bestellung, usw.
  - http://mycomapny.de/customers/1234
  - http://mycomapny.de/offers/4321
  - Usw.
- Implementierungsdetails bzgl. der Ressource sind transparent
  - Client kennt nur URI und die Repräsentationen

REST – Ressourcen

- Listen können auch Ressourcen sein
- In der Regel gibt es zu jeder Primärressource eine Listenressource z.B.
  - http://mycomapny.de/customers
- Per GET können damit alle Kunden abgefragt werden und per POST ein neuer Kunde angelegt werden
- Listenressourcen können um Filter erweitert werden, um z.B. alle Kunden eines bestimmten Vertriebsmitarbeiters abzufragen
- Paginierung zur Begrenzung der Menge möglich

REST – Ressourcen

- Ressourcen können auch Prozesses sein
- Alterativ zu mehreren Repräsentationen einer Ressource, kann auch eine Konzeptressource angelegt werden
- Diese verweist auf andere Ressourcen

REST – Ressourcen

- „Eine eigene URI für jedes Informationselement“
- Aufbau einer URI:
  - Erster Teil Schema: http:// oder https://
  - Gefolgt von Host: Rechnername, Domaine oder IP-Adresse
  - Optional gefolgt von Port z.B.: :1234
  - Danach kommen Pfad, Query und Fragment

REST – Ressourcen

- Aufbau einer URI
  - Auf Schema, Host und Port folgt der Pfad
  - Einzelne Elemente im Pfad werden durch / getrennt
  - Vergleich: Pfad in Dateisystem
  - z.B. http://mydomain.de:1234/main/customers/3
  - Auch relative Angaben möglich:
    - Z.B. ../orders/4
    - Verweist auf: http://mydomain.de:1234/main/orders/4

REST – Ressourcen

- Aufbau einer URI
  - Auf den Pfad kann optional das Query folgen
  - Wird durch ? vom Pfad getrennt
  - Enthält Query-Parameter die mit & getrennt werden
  - Z.B.: http://mydomain.de/main/customers?name=Mayer&city=Stuttgart
    - Query: name=Mayer&city=Stuttgart

REST – Ressourcen

- Aufbau einer URI
  - Auf das Query kann optional das Fragment folgen
  - Wird durch # vom Pfad bzw. Query getrennt
  - Wird vom Client aufgelöst nicht vom Server
  - Verweist auf einen bestimmten Teil in einem HTML-Dokument
  - Z.B.: http://mydomain.de/main/documents/3#section9

REST – Ressourcen

- URI Design
  - Sollten auf Ressourcen (Dinge bzw. Substantive) verweisen und keine Verben enthalten
    - Schlechtes Beispiel: http://mydomain.de/main/customers/create?name=Mueller
  - Sollten Hierarchien abbilden
    - Beispiel: http://mydomain.de/sales/germany/customers

REST – Ressourcen

- URI Design
  - Einsatz von Schlüsseln zur Identifikation von Ressourcen
    - Können entweder technische Schlüssel (z.B. ID aus Datenbank) sein oder fachliche Schlüssel (z.B. Postleitzahl)
    - Sollten möglichst stabil sein
    - Beispiel: http://mydomain.de/sales/germany/customers/3245
  - URIs sollten stabil sein
    - Bei Änderungen Redirect verwenden
    - Falls nicht mehr vorhanden Statuscode 410 Gone zurückliefern

REST - Verben

- Definierte und begrenzte Menge an Operationen, die für alle Ressourcen gültig sind
- HTTP definiert eine Liste an Operationen (Verben oder Methoden):
  - GET
  - HEAD
  - PUT
  - POST
  - DELETE
  - OPTIONS
  - (TRACE)
  - (CONNECT)

REST - Verben

- GET
  - Grundlegendes und wichtiges Verb
  - Ruft Informationen ab, die durch eine URI identifiziert werden
  - Server liefert Repräsentation der Ressource an den Client
  - Ist als sicher definiert
    - Führt zu keinen unerwünschten Seiteneffekten (Zustandsänderung auf dem Server)
  - Ist idempotent
    - Mehrfacher Aufruf führt zum selben Ergebnis wie einzelner Aufruf

REST - Verben

- HEAD
  - Ähnlich wie GET
  - Liefert keine Repräsentation der Ressource sondern Metadaten
  - Liefert gleiche Metadaten wie GET
  - Metadaten können sein:
    - Existiert die Ressource
    - Größe der Repräsentation
    - Zeitpunkt der letzten Änderung

REST - Verben

- PUT
  - Verändert eine Ressource oder erzeugt diese falls nicht vorhanden
  - Client gibt die URI vor unter der die Ressource angelegt wird
  - PUT ist inverse Operation zu GET
  - Repräsentation der Ressource wird im HTTP-Body an Server gesendet
  - HTTP-Header enthält das Format der Repräsentation
  - Ist wie HEAD und GET idempotent

REST - Verben

- POST
  - Legt eine neue Ressource an wobei der Server die URI festlegt
  - Zum Angelegen werden Listenressourcen aufgerufen
  - Kann für alles verwendet werden wofür andere Verben nicht passen
    - Zum Anstoßen beliebiger Funktionalität
  - Ist nicht sicher und nicht idempotent

REST - Verben

- DELETE
  - Löscht eine Ressource
  - Ist idempotent
  - Muss nicht zwangsweise in der Datenbank gelöscht werden. Markieren des Datensatzes als gelöscht ist ausreichend

REST - Verben

- OPTIONS
  - Liefert die Verben, die eine Ressource unterstützt
  - Ist idempotent und sicher

Referenzen

- Design Patterns – Elements of Reusable Object-Oriented Software,
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Patterns of Enterprise Application Architecture
  - Martin Fowler
- Reenskaugs Website
  - http://heim.ifi.uio.no/~trygver/
- ASP.NET Core MVC
  - https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-5.0

Referenzen

- WPF Apps With The Model-View-ViewModel Design Pattern
  - https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern
- Knockout MVVM
  - https://knockoutjs.com/documentation/observables.html#mvvm_and_view_models

Referenzen

- REST: the quick pitch
  - https://quoderat.megginson.com/2007/02/15/rest-the-quick-pitch/
- REST und HTTP – Entwicklung und Integration nach dem Architekturstil des Web
  - Stefan Tilko, Martin Eigenbrodt, Silvia Schreier und Oliver Wolf
- Swagger
  - https://swagger.io/
- Fiddler
  - https://www.telerik.com/fiddler/fiddler-classic