9. Architekturmuster

9.1. Motivation

Diagramme werden unübersichtlich, wenn die Anzahl dargestellter Komponenten (Klassen, Module, etc.) zunimmt. Das menschliche Gehirn kann 5 bis 7 Elemente gut überblicken, aber nicht mehr. Darum sind z.B. Klassendiagramme auch schon kleinerer Systeme, nicht mehr immer „auf einen Blick“ zu erfassen.

Module bilden eine Möglichkeit, von einzelnen Klassen zu abstrahieren und gröbere Einheiten aufzuzeichnen, als die in den Modulen enthaltenen Klassen. Diese vergröberten Strukturen sind dann wieder leichter zu verstehen.

Gleichzeitig reduziert die gröbere Abstraktion die Möglichkeiten für Beziehungen (in der Development View) zwischen Klassen, denn zwei Klassen aus verschiedenen Modulen können nur dann voneinander abhängig sein, wenn auch zwischen diesen Modulen eine entsprechende Beziehung besteht.

Eine weitere Hilfe zur Strukturierung – z.B. um mit einer zunehmenden Anzahl von Modulen zurecht zu kommen – sind *Architekturmuster*. Sie helfen auch auf gröberer Stufe, Komponenten nach bestimmten Kriterien zu gruppieren und zu definieren, welche Abhängigkeiten dazwischen erlaubt sind.

9.2. Lernziele

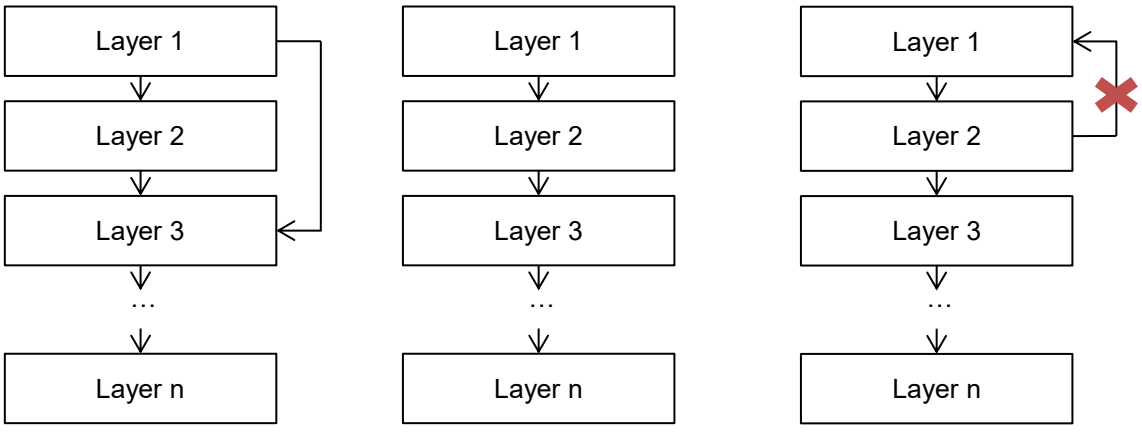
- Sie können wichtige *Architekturmuster* nennen, beschreiben und erkennen.
- Sie können den Sinn von *Schichtenarchitekturen* erklären, zwischen *strikten* und *nicht-strikten* Schichtenarchitekturen unterscheiden, bestehende Schichtenarchitekturen auf Korrektheit überprüfen und selbst welche entwerfen.
- Sie können Varianten des *MVC Musters* einschliesslich *MVVM* erklären, bestehende Architekturen auf Anwendung dieser Muster überprüfen und selbst Architekturen auf der Basis dieser Muster entwerfen.

9.3. Schichtenarchitekturen

Das vermutlich verbreitetste Schema, Software zu strukturieren, sind *Layers* oder *Schichten*: Komponenten werden nach gewissen Kriterien anhand ihrer Aufgabe (bzw. *Verantwortung*) in Blöcken zusammengefasst. Diese Blöcke stellt man sich schichtweise aufeinandergestapelt vor. Dabei dürfen Komponenten aus jedem Block nur Komponenten aus darunterliegenden Blöcken kennen, aber nie umgekehrt. Innerhalb eines Blocks sind die Abhängigkeiten jedoch nicht eingeschränkt.

Zwischen den Schichten gibt es keine gegenseitigen oder zyklischen Abhängigkeiten.

Darüber hinaus werden *strikte* und *nicht strikte* Schichtenarchitekturen unterschieden:



Abhängigkeiten in einer Schichtenarchitektur

*Abhängigkeiten in einer **strikten** Schichtenarchitektur*

Verbotene Abhängigkeit

Die in dieser Darstellung eingezeichneten Pfeile werden meist weggelassen. Gemeint ist dann zumeist, dass obere Schichten von unteren abhängig sein dürfen, aber nicht umgekehrt. Andere Darstellungen – mit anders gerichteten Pfeilen – sind grundsätzlich möglich aber eher selten.

Der Begriff der *Abhängigkeit* bezieht sich hier auf die *Development View*, das heisst auf *Import-Beziehungen* zwischen Klassen und Modulen – nicht auf Beziehungen zwischen Objekten.

Vorteile strikter gegenüber nicht-strikten Schichtenarchitekturen:

Development View

Logical View

Vorteile nicht-strikter gegenüber strikten Schichtenarchitekturen:

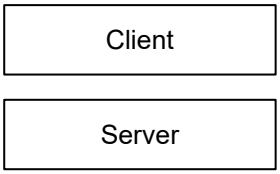
Development View

Logical View

Der Trade-Off zwischen strikter oder nicht-strikter Schichtenarchitektur findet also meist statt zwischen

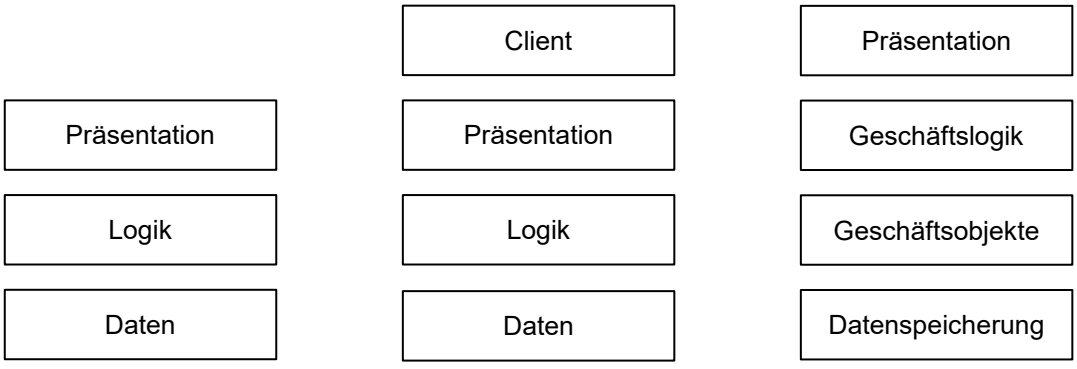
Typische Beispiele für Schichtenarchitekturen

Schichtenarchitekturen sind sowohl in Datenverwaltungssystemen als auch in Produkten, die als Plattformen für solche Systeme verwendet werden, verbreitet. Die Schichten werden dabei manchmal auch als *Tiers* bezeichnet. Einfache *Client-Server-Architekturen* stellen bereits zwei Schichten dar (s. links).



Client kennt Server, aber nicht umgekehrt

Besonders häufig sind die Drei-Schichten-Architektur (*three tier architecture*) und Vier-Schichten-Architektur (*four tier architecture*) vertreten. Einige Beispiele:

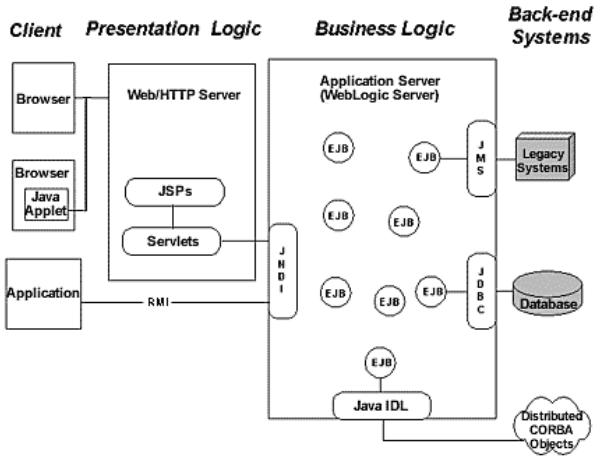


Drei-Schichten-Architektur

Vier-Schichten-Architektur für Web-Applikationen (z.B. JEE)

Vier-Schichten-Architektur mit Geschäftsobjekten

Grundsätzlich ist bei solchen Architekturen die Trennung nach Bereitstellung und persistenter Speicherung von *Daten*, *Logik* zur systematischen Manipulation dieser Daten sowie zur *Präsentation* in Benutzerschnittstellen. Je nachdem werden einzelne Schichten weiter aufgeteilt.



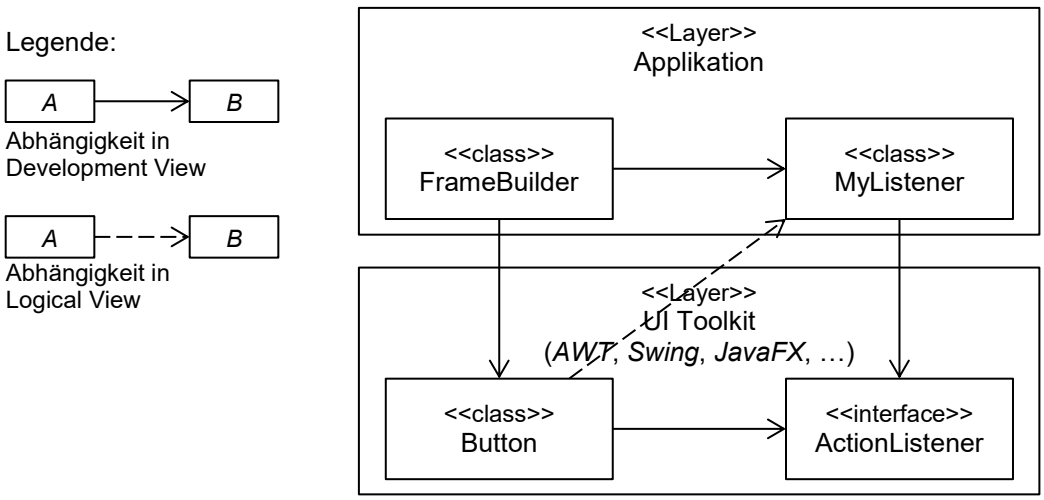
Bei Web-Applikationen unterteilt man die Präsentationsschicht zusätzlich auf Grund der physischen Verteilung in einen *Client* und einen *Server*-Teil. Bemerkenswert ist dabei, dass der anwendungsspezifische Code in der *Client* Schicht aus der *Server*-Schicht bezogen wird. Diese wird auch als *Presentation Logic* bezeichnet.

Nebenstehende Graphik zeigt eine solche Umsetzung am Beispiel von *Java EE*.

Manchmal unterteilt man auch die *Daten*-Schicht in eine objektorientierte *Geschäftsobjekte*-Schicht und eine meist klassisch mit relationalen Modellen implementierte *Datenbank*-Schicht (in JEE Teil der *Back-end Systems*).

Wie wir schon bei Modulen gesehen haben, gibt es oft in der Development View Restriktionen für Abhängigkeiten, die notwendigen Beziehungen in der Logical View entgegenstehen, also Beziehungen zwischen Objekten zur Laufzeit. In der Development View bedeutet „*A* ist von *B* abhängig“ – wobei *A* und *B* beliebige Klassen oder Interfaces bezeichnen – dass *B* zur Kompilations-Zeit von *A* vorhanden sein muss. Sei dies, weil *B* als Typ von Variablen oder Parametern in *A* vorkommt oder weil in *B* deklarierte Operationen aufgerufen werden sollen, und sei dies „nur“ ein Konstruktor. Solche Abhängigkeiten sollten möglichst nur zu Klassen bestehen, die auch inhaltlich sehr eng gekoppelt sind.

Aufrufe zwischen Objekten sind oft auch entgegen der Schichtenstruktur nötig. Der Widerspruch lässt sich auflösen durch Mechanismen, wie das allgegenwärtige *Observer Pattern*, das Sie aus allen gängigen UI-Toolkits kennen.



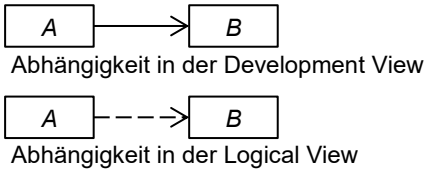
9.4. Model-View-Controller (MVC)

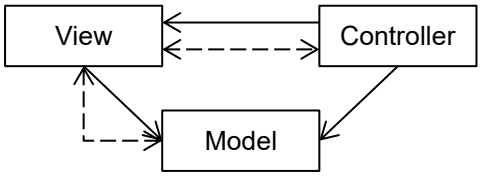
Ein zweites häufiges Architekturmuster ist das *Model-View-Controller-Muster* (*MVC*). Es wurde in seiner ursprünglichen Form 1979 für die Sprache und Programmierumgebung *Smalltalk* entworfen. Damals wurden durch die Entwicklung bei der Hardware von ASCII-Terminals zu Bitmap-Displays und Zeigegeräten erstmals komplexere Benutzerinteraktionen möglich. Es entstand das Bedürfnis, klar zwischen Datenverarbeitung und der Maschinerie für Benutzerinteraktion zu trennen, unter anderem, um Benutzerschnittstellen rasch an neue Entwicklungen anpassen zu können, ohne die Berechnungen für die Geschäftsprozesse zu tangieren.

Die ursprüngliche Idee ist die einer Zwei-Schichten-Architektur: Auf einer Datenverarbeitungs-Schicht (mit allen Daten und Prozessen) gibt es eine Benutzerschnittstellenschicht. Diese wurde dann nochmals auf-

geteilt, in einen Teil, der für die Darstellung der Daten für die Benutzer verantwortlich ist (*View*) und einen der umgekehrt die Aktionen des Benutzers mit Maus und Tastatur zu interpretieren vermag (*Controller*).

Diese ursprüngliche Definition des Controllers hat sich mittlerweile weitgehend verloren. Früher war es Aufgabe des Controllers, überhaupt zu ermitteln was am Ort der Maus dargestellt wurde, und was bei einem Mausklick darauf passieren sollte. Heute werden diese Aufgaben von den einzelnen GUI-Widgets und Containern übernommen, die Benutzer-Interaktionen in spezifische Ereignisse übersetzen.





Die primären Ziele des *MVC*-Musters sind neben der klaren Trennung von Verantwortung:

1. Entwicklung alternativer *Views* und *Controller* zu ermöglichen, ohne dass die Geschäftslogik dafür verändert werden muss
2. mehrere *Views* gleichzeitig pro *Model* zu ermöglichen
3. gleich aussehende *Views* mit unterschiedlichem Verhalten bei Benutzeraktionen zu ermöglichen
4. leicht zu automatisierende Tests auf der Geschäftslogik zu ermöglichen

Beziehungen zwischen Model und View

Um die obigen Ziele zu erreichen, dürfen die Klassen, die das *Model* ausmachen, nicht von jenen der *View* abhängig sein. Trotzdem muss die *View* Änderungen der dargestellten Daten darstellen können. Änderungen an den Model-Daten müssen also Änderungen in der Darstellung bewirken können.

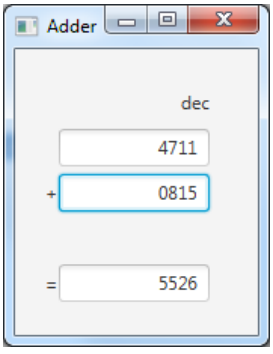
Deswegen ist das *Observer Pattern* ein fester Bestandteil von MVC. Views werden beim Model als Observer registriert und dann von diesem über Änderungen informiert.

Um die Programmierer davon zu entlasten, die Infrastruktur des Observer-Mechanismus selbst programmieren zu müssen, bieten viele Frameworks und Toolkits unter dem Titel *MVC-Unterstützung* Basismechanismen für das Observer Pattern an. Beim Entwurf der Applikation muss man dann nur noch entscheiden, wie man diesen Rahmen füllt. Es müssen konkrete Nachrichten, Parameter usw. festgelegt werden und – je nach Möglichkeiten des Frameworks – auch wann die Meldungen ausgelöst werden sollen. Diese anwendungsabhängigen Definitionen bilden einen wesentlichen Teil der Schnittstelle zwischen den MVC-Komponenten. Deswegen sollten die dann auch dokumentiert werden und nicht nur, welches Framework benutzt wird.

Micro-MVC

Sowohl *Swing* als auch *JavaFX* verwenden selbst eine Model-View-Trennung. Die Controls sind *Views* im Sinne von MVC, die jeweils von kleinen einfachen Model-Objekten abhängen. Bei *Swing* fällt diese Tatsache bei der alltäglichen Verwendung allerdings wenig auf, weil nur wenige Programme die hinter den Controls versteckten Modelle explizit benutzen, z.B. um zwei Controls zu synchronisieren.

JavaFX ermutigt stärker, die Modell-Objekte direkt zu verwenden. Diese heissen dort jedoch nicht *Model* sondern *Property*. Sie zeichnen sich dadurch aus, dass sie sowohl als *Observable* Werteänderungen an registrierte *Listeners* melden, als auch selbst bei anderen Properties *Listeners* installieren können. Dadurch lassen sich relativ leicht ganze Geflechte von solchen Properties aufbauen.



Beispiel

In zwei Textfeldern können Zahlen eingetragen werden und ein drittes Textfeld zeigt ständig die Summe dieser beiden Zahlen an. Dazu wird in den mit den beiden Eingabefeldern verbundenen *TextProperties* je ein Listener installiert, der dann bei jeder Text-Änderung jeweils in einer *IntegerProperty* den entsprechenden Zahlenwert setzt.

Die beiden *IntegerProperties* wiederum werden mit einer dritten verbunden, welche die Summe berechnet und dann wiederum bei Änderungen die *TextProperty* des dritten Textfelds neu setzt.

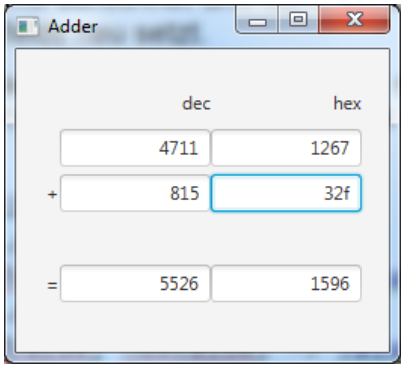
Alle diese Verbindungen brauchen nur einmal bei der Initialisierung der *JavaFX*-Scene hergestellt zu werden, wenn die *initialize*-Methode aufgerufen wird:

```
@FXML
private void initialize() {
 op1.textProperty().addListener(
    (observable, oldValue, newValue) -> value1.set(intFromString(newValue, 10)));
 op2.textProperty().addListener(
    (observable, oldValue, newValue) -> value2.set(intFromString(newValue, 10)));
 sum.textProperty().bind(value1.add(value2).asString());
}

private int intFromString(String s, int base) {
 try { return Integer.parseInt(s, base); }
 catch (NumberFormatException e) { return 0; }
}
```

Während *JavaFX-Properties* und *Controls* das MVC-Muster „im Kleinen“ realisieren (Micro-MVC), ist das *Adder*-Programm als Ganzes nicht nach MVC strukturiert (sondern ein Monolith). Welche Teile des obigen Code-Ausschnitts gehören eher zu *Model*, zu *View* oder *Controller*? Markieren Sie diese farbig.

Lernaufgabe JavaFX-Properties



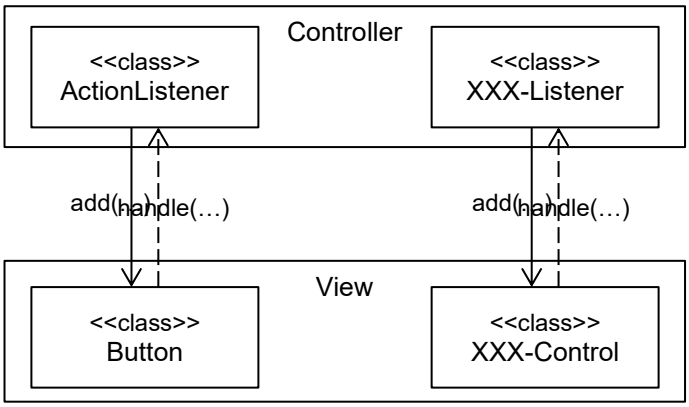
Erweitern Sie dieses Programm so, dass die Werte auch als Hexadezimalzahlen angezeigt werden bzw. eingegeben werden können. Im bereitgestellten Eclipse-Projekt sind drei entsprechende Felder bereits vorbereitet. Sie brauchen nur noch die Properties geeignet zu verbinden.

Eine Methode *stringFromInt* zur Umwandlung von Zahlenwerten in einen String zu einer beliebigen Basis steht Ihnen ebenfalls zur Verfügung.

Beziehungen zwischen Controller und View

Das dritte der oben aufgelisteten Ziele des MVC-Musters bedingt, dass der Code für die Interpretation von Benutzereingaben separat von *View* und *Model* programmiert wird. Ort dafür ist der *Controller*.

Die Ereignisse aus der Benutzeroberfläche (Tastatur-, Maus-, Touch-Events usw.) werden über die Struktur der Views an einzelne Controls geleitet. Die GUI-Toolkits bilden die Maus-Koordination gemäss der graphischen Erscheinung der View ab und leiten Tastatureingaben an das Control, das aktuell den Fokus enthält.



Damit der Controller über Ereignisse informiert werden kann, ohne dass die View-Klassen von den Controller-Klassen abhängig werden, braucht man wieder Callbacks. Im Sinne des *Strategy Pattern* kann man den Controller als Strategie der View sehen, um mit Benutzerereignissen variabel umgehen zu können.

Praktischerweise bieten Controls als Bauelemente der Views bereits einen Installationsmechanismus für solche Callbacks an: Listener. Oft wird deswegen der Controller nicht über einen einzigen Callback-Handler mit der View verbunden, sondern gerade über eine Vielzahl kleiner Listener-Registrierungen.

Der Controller steuert die Reaktionen der Software auf Benutzereingaben. Dazu gehört meist, entsprechende Geschäftslogik in Gang zu setzen (s. nächster Abschnitt). Es gehört aber auch dazu, visuelles Feedback der View zu steuern. Beispielsweise werden einzelne Controls auf passiv (disabled, „grau“) geschaltet, wenn sie nicht benutzt werden können.

Der MVC-Controller ist also für die *GUI-Logik* verantwortlich, nicht jedoch für die *Geschäftslogik*, die ausschliesslich Teil des Models ist (s. unten).

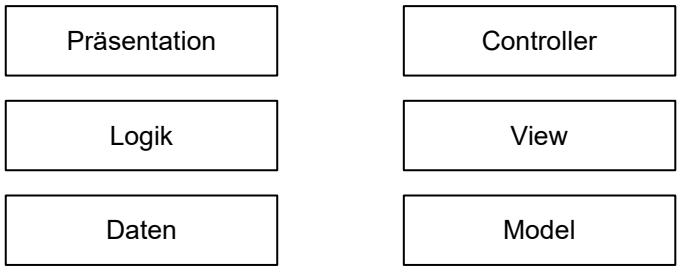
In der Praxis wird das hinter der Trennung zwischen *Controller* und *View* stehende Ziel oft als weniger wichtig empfunden. Dann wird auf die Trennung verzichtet und die beiden Elemente werden zu einem zusammengefasst.

Beziehungen zwischen Controller und Model

Nicht vernachlässigt werden darf die Unterscheidung zwischen *GUI-Logik* und *Geschäftslogik*. Letzteres meint die Logik, die hinter Manipulationen der (persistenten) Daten steht, also die Ablaufsteuerung (komplexer) Geschäftsprozesse. Während die *GUI-Logik* ein Teil der Verantwortung der Benutzerschnittstelle ist, unterliegen die Geschäftsprozesse anderen Veränderungen und anderen Qualitätsmassstäben. Typischerweise sind auch andere Kernkompetenzen bei der Programmierung nötig, als beim GUI-Design. Daher möchte man die Geschäftslogik auch bei Änderungen des GUI unberührt lassen.

Umgekehrt darf natürlich das *Model* nicht von Implementierungs-Details der Darstellung abhängig gemacht werden, weshalb die *GUI-Logik* eben nicht dorthinein gehört.

Will man Ablauflogik von der Datenhaltung trennen, heisst das eigentlich, dass man das Model selbst nochmals in zwei Schichten gliedert. Da Softwareteile, deren Verantwortung schwerpunktmässig in der Koordination von Aktivitäten besteht, auf Englisch als *Controller* bezeichnet werden, kann es durchaus sinnvoll sein, die obere dieser Schichten ebenfalls so nennen. Man sollte diesen *Business Logic Controller* aber nicht mit dem *GUI-Logic Controller* des MVC-Musters verwechseln.



Drei-Schichten-Architektur *MVC*

Grenzen von MVC

Charakteristisch für das *MVC-Muster* sind vor allem zwei Merkmale:

Sind diese beiden Merkmale nicht vorhanden, sollte man genau genommen nicht mehr von einem *MVC-Muster* sprechen, sondern allenfalls von einem davon abgeleiteten Muster.[1]

Einen unmittelbaren Konflikt gibt es bei Web-Applikationen zwischen dem dafür sinnvollen REST-Modell und dem MVC-Muster. RESTful-Services sind unter anderem dadurch charakterisiert, dass der Client eine Anfrage an den Server stellt, dieser einmal antwortet und danach – bis zu einer nächsten Anfrage des Clients – keine Verbindung zwischen Client und Server mehr besteht. Damit sind speziell Notifikationen über Zustandsänderungen vom Server an den Client nicht möglich.

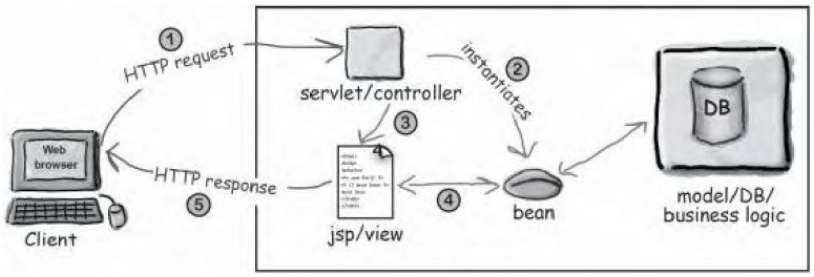
Das MVC-Muster lässt sich also bestenfalls auf dem Server einsetzen. Die wirklich dargestellte View wird nur aktualisiert, wenn der Client danach fragt.

Im Zusammenhang mit Java Web-Technologie basierend auf *Servlets* und *JSP* [2] spricht man deswegen vom *MVC2*-Muster – eine leichte Abwandlung des MVC-Musters.

[1] Wegen seines Bekanntheitsgrads wird das MVC-Muster heute vielfach quasi zu Werbezwecken herangezogen, indem verschiedene Produkte für die Programmierung (wie Frameworks) damit angepriesen werden, das MVC-Muster zu unterstützen. Oft findet sich dann aber vor allem Unterstützung für eine Drei-Tier-Architektur.

[2] JavaServer Pages

1. HTTP-Request wird von *Servlet* empfangen
2. *Servlet* agiert als *Controller*. Der Request wird üblicherweise in Form eines JavaBeans an das *Model* weitergereicht.
3. *Servlet* leitet die Kontrolle an *JSP* weiter (redirect)
4. *JSP* produziert HTML-Output mit den *Model*-Daten aus dem JavaBean.
5. *JSP* sendet neue Seite mittels HTTP-Response an Browser zurück.



Die Verantwortung der *Controller*-Komponente ist also

und die der *View*-Komponente ist

9.5. Model-View-ViewModel (MVVM)

Ein Anliegen der Software-Strukturierung in der *Development View* ist, dass man Dinge, die sich unabhängig voneinander ändern, über die man unabhängig voneinander entscheiden möchte, oder für die unterschiedliche Fähigkeiten oder Werkzeuge nötig sind, in jeweils separate Einheiten verpacken möchte. Bei Benutzerschnittstellen haben sich dabei in neuerer Zeit folgende *Verantwortungsbereiche* herauskristallisiert, nach denen das Architektur-Muster *MVVM* gegliedert ist:

1. *Eigentliche Datenverarbeitung* unabhängig von der Benutzerschnittstelle: Entspricht dem *Model* im *MVC*-Muster oder den *Logik*- und *Daten*-Schichten in klassischen Schichten-Architekturen.

   Diese Programmteile sollen von Benutzerschnittstellen völlig unabhängig sein, sowohl was die von ihnen übernommene Verantwortung als auch was die verwendete Technologie betrifft. (Beispielsweise werden dort eher keine JavaFX-Properties verwendet.)

   Im MVVM-Muster heisst diese Schicht *Model*.

2. *Operationen und Datenverknüpfungen* für die Benutzerschnittstelle: Ein spezifisches *Model* mit Fokus auf die vorgesehenen Benutzerinteraktionen.

   Diese Programmteile setzen auf der Datenverarbeitung auf und enthalten zusätzliche Mechanismen spezifisch für die Benutzerinteraktion im Sinne von *GUI-Logik*. Dazu gehört, für die Darstellung relevante Daten bereitzustellen und ggf. zu verknüpfen oder Benutzer-Eingaben zu validieren und ggf. zurückzuweisen.

   Im MVVM-Muster heisst diese Schicht *ViewModel*.

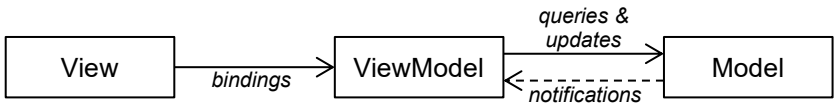
3. *Darstellung und Layout* in der Benutzerschnittstelle: Das eigentliche Erscheinungsbild der Benutzerschnittstelle.

   Diese Programmteile verwenden die vom *ViewModel* angebotenen Daten (*Properties*) und Operationen, um eine konkrete Benutzerschnittstelle damit aufzubauen. Dazu werden konkrete *Controls* platziert, konfiguriert und an die Properties *angebunden*. Es wird entschieden, mit welchen Controls genau gearbeitet wird (*TextFeld* oder *TextArea* beispielsweise, *Slider* oder Text-Eingabe oder beides, usw.), wo und wie gross diese dargestellt werden und wie sie genau aussehen sollen.

   Diese Schicht versucht man möglichst mit deklarativen Sprachen zu beschreiben. In *JavaFX* dient dazu *FXML* und *css*, in Microsofts .NET *XAML*. (Allerdings können in *FXML* direkt nur sehr eingeschränkt *Bindings* von Controls am Properties ausgedrückt werden. Deswegen braucht es zusätzlich eine Java-Klasse zur Vermittlung.)

   Im MVVM-Muster heisst diese Schicht *View*.

Für das *MVVM*-Muster sind *Bindings* von UI-Elementen an Datenelemente charakteristisch. Das entsprechende „micro *MVC*-Muster" der Controls dient hier zur Kopplung zwischen *View* und *ViewModel*. Für die Kopplung zwischen *ViewModel* und *Model* drängen sich makroskopische Notifikationsmuster auf.



Das *MVVM*-Muster wurde in dieser Form bei Microsoft im Zusammenhang mit WPF und Silverlight entwickelt, verbreitet sich aber allmählich auch darüber hinaus. Von JavaFX wird es nur eingeschränkt gut unterstützt, aber mit etwas Disziplin kann man die wesentlichen Elemente davon umsetzen.

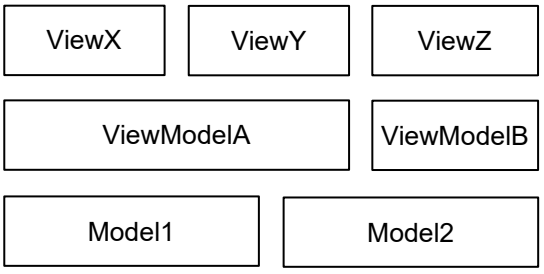
Projektion und Mashup

Wie auch beim *MVC*-Muster die View muss auch beim *MVVM*-Muster das *ViewModel* nicht den gesamten Inhalt eines angebundenen Modells und auch nicht alle Funktionalität davon benutzen bzw. wiedergeben. Man *projiziert* das Model auf einen für die jeweilige Benutzerinteraktion relevanten Teilbereich.

Als Beispiel stelle man sich für das Modell einer Personen-Daten-Verwaltung eine spezielle Benutzer-Schnittstelle vor, mit der nur Daten von Personen mit einer bestimmten Rolle bearbeitet werden können.

Umgekehrt eignet sich speziell das *ViewModel* des *MVVM*-Musters dazu, Informationen aus mehreren Modellen zu verbinden. Man spricht manchmal auch *Mashup*. Dabei werden Informationen aus verschiedenen Quellen so verbunden, dass eine Gesamtsicht entsteht, an deren Element dann die *View* angebunden werden kann.

Eine beliebte Anwendung sind beispielsweise geographische Informationssysteme bei denen Koordinaten aus einer relationalen Datenbank mit Kartenmaterial verbunden werden. Das *Model* für letztere wird dann meist von einem externen Web-Service bereitgestellt.



9.6. Zusammenfassung

Architekturmuster helfen grössere Mengen von Klassen so zu gruppieren, dass diese Gruppen dem System wieder eine übersichtliche Struktur geben. Die Abhängigkeiten zwischen Klassen verschiedener Gruppen, die beim Programmieren und Kompilieren (also in der *Development View*) bereits eine Rolle spielen, werden nach festen Regeln eingeschränkt. Die Zuteilung einzelner Klassen richtet sich je nach Architekturmuster nach der von den Klassen übernommenen Verantwortung (z.B. Datenvisualisierung / -berechnung / -speicherung) und nach technischen Eigenschaften (z.B. deklarative Programmierung).