Model-View-Controller-Paradigma

Dieser Artikel erfüllt die GlossarWiki-Qualitätsanforderungen **nur teilweise**:

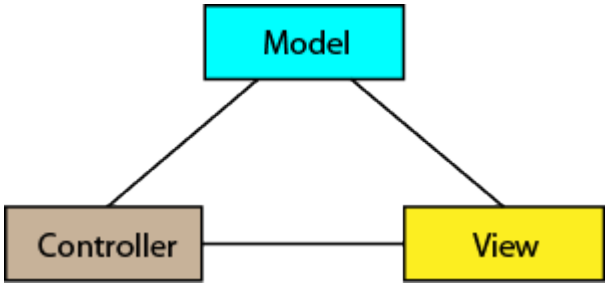
Korrektheit: 4 (großteils überprüft)	**Umfang**: 3 (einige wichtige Fakten fehlen)	**Quellenangaben**: 4 (fast vollständig vorhanden)	**Quellenarten**: 4 (sehr gut)	**Konformität**: 4 (sehr gut)

Diese Bewertungen beziehen sich auf alle im nachfolgenden Menü genannten Artikel gleichermaßen.

MVC-Paradigma:	Model (Data) | View | Controller	MVC-Pattern:	Singletons | Dependency Injection | Observers
MVCS-Paradigma:	Service	MVCS-Pattern:	Singletons | Dependency Injection | Observers
LDVCS-Paradigma:	Logic	VCLSD-Pattern:	Singletons | Dependency Injection | Observers

Inhaltsverzeichnis

1 Definition



MVC-Module

Als Model-View-Controller-Paradigma (engl. Model-view-controller paradigm), oder -Architektur (engl. architecture), kurz MVC-Paradigma, MVC-Architektur oder auch nur MVC, bezeichnet man ein Architekturmuster, bei der eine Anwendungs-Komponente in drei eigenständige Module unterteilt wird: Model (Modell), View (Darstellung, Präsentation) und Controller (Steuerung).

1.1 Model (Modell)

Ein MVC-Modell (engl. MVC model) einer MVC-Anwendung dient zur Speicherung bestimmter Daten, d. h. zur Speicherung von Teilen des aktuellen Zustands der Anwendung.

Ein MVC-Modell kann weitere Aufgaben übernehmen:

anderen Modulen Zugriff auf die Zustandsdaten gewähren
andere Module über Änderungen informieren (meist mittels des Observer-Patterns)
Umsetzung der Komponentenlogik
Kommunikation mit externen Datenquellen (zum Zweck der Datensynchronisation)

1.2 View (Darstellung, Präsentation)

MVC-Views (engl. MVC views) sind die grafischen, akustischen, haptischen und olfaktorischen Schittstellen einer MVC-Anwendung. Sie „visualisieren“ Daten, die in Modellen der Anwendung enthalten sind.

Eine MVC-View kann weitere Aufgaben übernehmen:

bei Daten-Änderungen in den zugehörigen Modellen der Anwendung die View-Repräsentation dieser

Daten automatisch anpassen
Benutzeraktionen, die über grafische Eingabeelemente – wie Textfelder oder Buttons – erfolgen, an einen Controller weiterleiten
Visualisierung von Status- und Fehlermeldungen von Controllern

1.3 Controller (Steuerung)

MVC-Controller (engl. MVC controllers) dienen zur Steuerung einer MVC-Anwendung. Dazu nimmt ein Controller Eingaben aus verschiedensten Quellen entgegen (z. B. Sensor-Daten oder Daten, die ein Benutzer über eine beliebige Benutzer-Schnittstelle wie eine Tastatur oder eine Maus eingibt) und leitet diese bereinigt und normalisiert an ein Modell weiter.

Ein MVC-Controller kann weitere Aufgaben übernehmen:

Verarbeitung von Systemsignalen, wie z. B. einer Systemuhr (z. B. „Spielzeit ist abgelaufen“)
Umsetzung der Komponentenlogik
Kommunikation mit externen Datenquellen (zum Zweck der Datensynchronisation)

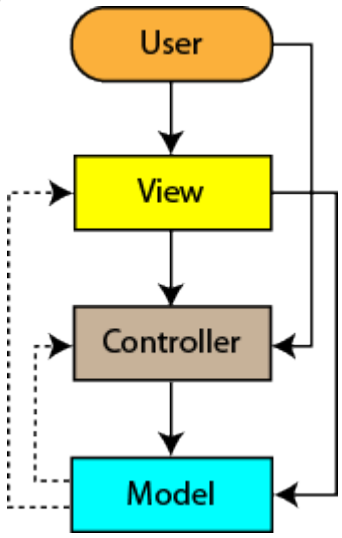
2 Anmerkungen

Views könnten theoretisch auch Controller-Informationen (wie z. B. Fehler-Meldungen oder Service-Status-Meldungen) visualisieren. Allerdings würde dies dem Grundgedanken wiedersprechen, dass Views nur auf möglichst wenige andere Module zugreifen soll (normalerweise greifen Views nur schreibend auf Controller zu). Insofern ist es besser, Controller-Statusmeldungen in speziellen Status- oder Fehlermodellen zu speichern, die dann von einer View visualisiert werden.

Logische *Sichten* zu verwalten, wie z. B. die Regelung des Zugriffs auf verschiedene Daten abhängig von den Benutzerrechten, ist nicht die Aufgabe einer View, sondern Aufgabe eines Modells oder eines Controllers.

Gerade weil umstritten ist, ob die Komponentenlogik besser im Modell oder besser im Controller angesiedelt wird, sollte man das Logik-Modul als eigenständiges Modul realisieren. Dies wird beispielsweise im LDVCS-Paradigma umgesetzt.

2.1 MVC-Paradigma: Varianten (Definitionen nach Kowarschick (MMProg))



Besipiel für eine VCM-Schichtenarchitektur

2.2 VCM-Paradigma

Ein MVC-Paradigma wird **VCM-Paradigma** genannt, wenn die drei zugehörigen Module folgende Schichtenarchitektur bilden: *View*, *Controller*, *Model*.

Diese Architektur ist für interaktive Anwendungen wie z. B. Spiele sehr gut geeignet.

Die drei Module einer MVC-Komponente sind gemäß dem Schichtenparadigma in Ebenen angeordnet. Die höheren Ebenen können auf tiefergelegene Ebenen zugreifen, aber nicht umgekehrt.

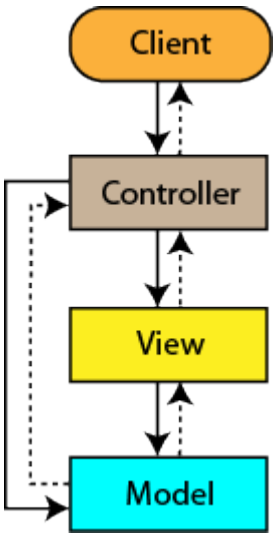
Die unserste Ebene enthält das Modell. Das zugehörige Modul weiß nichts von den über ihr liegenden Module und kann mit diesen nur mit indirekt – z. B. durch Antworten auf Nachrichten oder mit Hilfe des Observer-Patterns – kommunizieren.

Ein Controllermodul kann direkt auf ein Modellmodul zugreifen, um die Inhalte zu manipulieren.

Eine View kommuniziert i. Allg. direkt nur mit Controllermodulen, um diesen Benutzeraktionen, die über die View erfolgen, mitzuteilen. Ein direkter Zugriff auf ein Datenmodul ist nur dann notwendig, wenn die Nachrichten, die von den Datenmodulen verschickt werden, nicht alle relevanten Informationen enthalten. Zugriffe auf Servicemodule sind nicht vorgesehen.

Der Benutzer stellt das „oberste Modul“ dar. Er kommuniziert nur mit View- und Controllermodulen.

Man beachte, dass in diesem Paradigma die Logik sowohl im Modell als auch im Controller realisiert werden kann.



Besipiel für eine CVM-Schichtenarchitektur

2.2.1 CVM-Paradigma

Ein MVC-Paradigma wird **CVM-Paradigma** genannt, wenn die drei zugehörigen Module folgende Schichtenarchitektur bilden: *Controller*, *View* und *Model*.

Diese Architektur ist für Web-Anwendungen sehr gut geeignet.

Die Controller-Module können direkt (z. B. via Unicast-Nachrichten) auf die View- und Modell-Module zugreifen und die View-Module können direkt auf Modell-Module zugreifen. Alle anderen „Zugriffe“ erfolgen nur indirekt (i. Allg. als Antworten auf Unicast-Nachrichten, evtl. auch mittels Callback-Routinen).

Man beachte, dass in diesem Paradigma die Logik der Anwendung normalerweise im Controller und nicht im Modell realisiert wird (siehe CVM-Prozess, Model 2).

2.2.2 MVC-Multicast-Paradigma

Ein MVC-Paradigma wird MVC-Multicast-Paradigma genannt, wenn alle Module nur indirekt, d. h. mit Hilfe von Multicast-Nachrichten kommunizieren.

Ein typischer Vertreter dieser Architektur ist PureMVC[1].

Hier ist es denkbar, dass eine View auch Status- und Fehlermeldungen von Controllern direkt visualisert (ohne einen Umweg über spezielle Status- und Fehlermodule zu nehmen), weil eine View den Absender einer Nachricht sowieso nicht explizit zu kennen braucht.

2.3 MVC-Pattern (Definitionen nach Kowarschick (MMProg))

Wenn für ein MVC-Paradigma – im Sinne eines Entwurfmusters – eine konkrete Klassen- und Objekt-Struktur für die drei Module Model, View und Controller vorgegeben ist, spricht man von einem MVC-Pattern.

2.3.1 VCM-Pattern

Ein MVC-Pattern, das ein VCM-Paradigma realisiert, wird auch **VCM-Pattern** genannt.

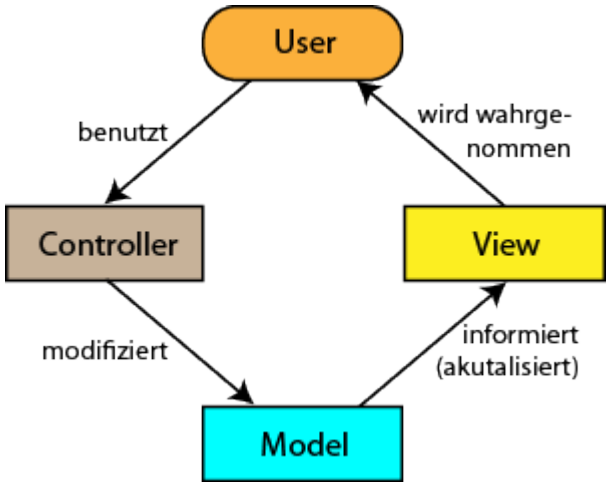
2.3.2 CVM-Pattern

Ein MVC-Pattern, das ein CVM-Paradigma realisiert, wird auch **CVM-Pattern** genannt.

2.3.3 MVC-Multicast-Pattern

Ein MVC-Pattern, das nur auf Multicast-Nachrichten basiert, wird auch **MVC-Multicast-Pattern** genannt.

3 MVC-Prozesse (Definitionen nach Kowarschick (MMProg))



MVC-Prozess (in Anlehnung an Reenskaug (2003), Slide 17 auf Seite 10, sowie Commons:File:MVC-Process.svg)

Die Aufgaben, die die einzelnen Module des MVC-Paradigmas wahrnehmen, lassen sich als Interaktionsprozesse zwischen diesen Modulen darstellen. Im Folgenden werden mehrere Varianten von derartigen Interaktionsprozessen beschrieben.

3.1 MVC-Prozess nach Reenskaug

Ursprünglich wurde das MVC-Paradigma von Reenskaug entwickelt. In Reenskaug (2003) ist der grundlegende MVC-Prozess beschrieben: Die Anwendungsdomäne wird im so genannten Modell (Model) nachgebildet. Dieses informiert eine oder mehrere Views über jede Änderung des aktuellen Zustands. Dem Benutzer (User) wird der aktuelle Zustand des Modells von einer oder mehrere dieser Views präsentiert. Über die Steuerung (Controller) kann der Benutzer das Modell manipulieren, das heißt, dessen Zustand (und damit auch die zugehörigen Darstellungen) ändern.

3.1.1 Beispiel Jump 'n' Run

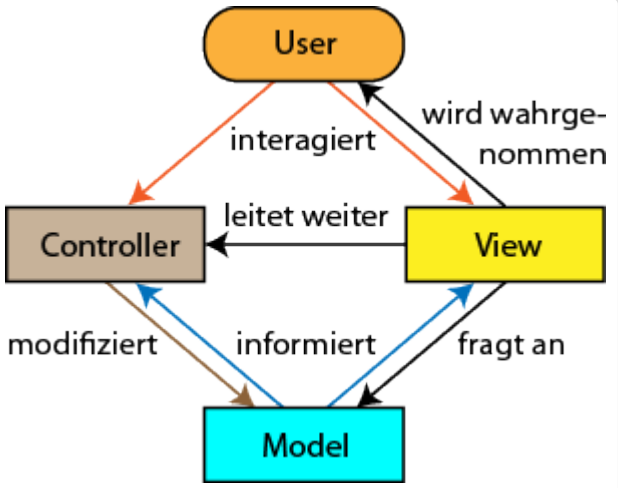
In einem Jump-'n'-Run-Spiel werden im Modell Daten über die Spielfigur (Position, Laufrichtung, Geschwindigkeit ...), die Gegner, die Gegenstände etc. gespeichert.

Das Darstellungsmodul visualisiert die Elemente des Spiels mit Hilfe von Bildern und Animationen (Walk cyles etc.). Jede Änderung im Modell hat, sofern sie sich im für den Spieler sichtbaren Bereich befindet, eine Anpassung der Darstellung zur Folge.

Der Spieler steuert die Spielfigur mit Hilfe der Tastatur. Jeder Tastendruck wird vom Steuerungsmodul analysiert und zur Manipulation der Spielfigur an das Modell weitergeleitet.

Man beachte, dass das Modell i. Allg. aktiv ist, d. h. seinen Zustand selbstständig auch ohne Manipulatation durch die Steuerkomponete verändern kann. Beispielsweise werden die gegnerischen Figuren, sofern es welche gibt, vom Modell selbst bewegt.

3.2 Erweiterung des MVC-Prozesses



MVC-Prozess 2 (mit erweiterter Interaktion)

Im zurvor beschriebenen MVC-Prozess kommuniziert der Benutzer direkt mit einer Steuerkomponente. Dies ist z. B. bei einer Tastatur-Steuerung, bei Verwendung von Webcam und Mikrofon oder bei einer Kommunikation über eine Daten-Schnittstelle (wie z. B. eine serielle Schnittstelle oder einen USB-Controller) möglich.

Meist werden jedoch dem Benutzer Interaktionselemente (wie Button, Slider und Texteingabe-Felder) über eine View präsentiert. In so einem Fall sollte die Darstellungkomponete die Benutzereingaben nicht selbst verarbeiten, sondern an ein zuständiges Steuerungsmodul weiterleiten.

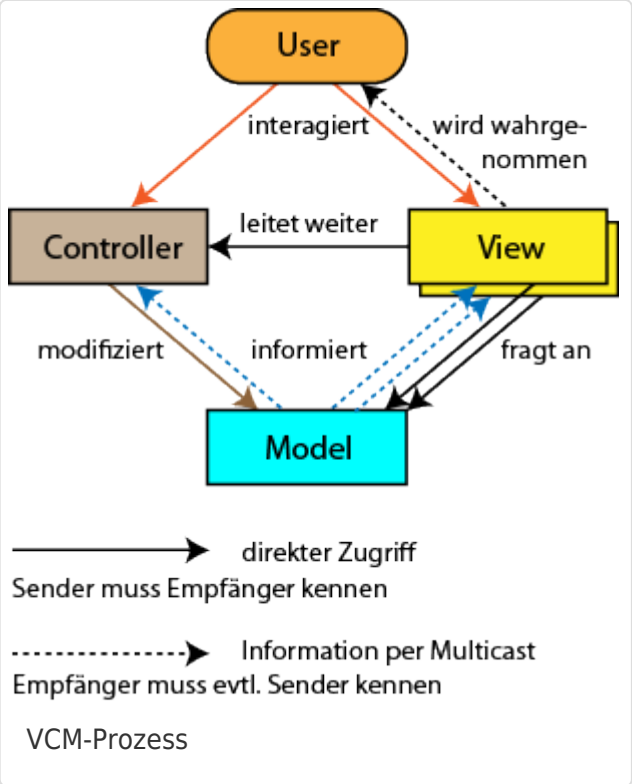
Eine weitere Verfeinerung des zuvor beschriebenen Prozessablaufs wird bei der Kommunikation zwischen Modell und Controller vorgenommen: Das Modell kann nicht nur die View, sondern auch die auch die Steuerung direkt über Änderungen informieren, damit dieser dem User gegebenfalls ein Feedback (z. B. Force Feedback) geben kann. Hier kann man allerdings auch argumentieren, dass diese Art der Informations-„Visualisierung“ Aufgabe einer View wäre. Das heißt, im Allgemeinen kann die Steuerung darauf verzichten, auf Modellereignisse zu hören und zu reagieren.

Zu guter Letzt ist in manchen Fällen auch noch eine direkte Kommunikation zwischen View und Modell notwendig. Wenn ein Modell ein anderes Modul (wie eine View, einen Controller oder evtl. auch ein anderes Modell) lediglich darüber informiert, dass sich etwas am Datenbestand des Modells geändert

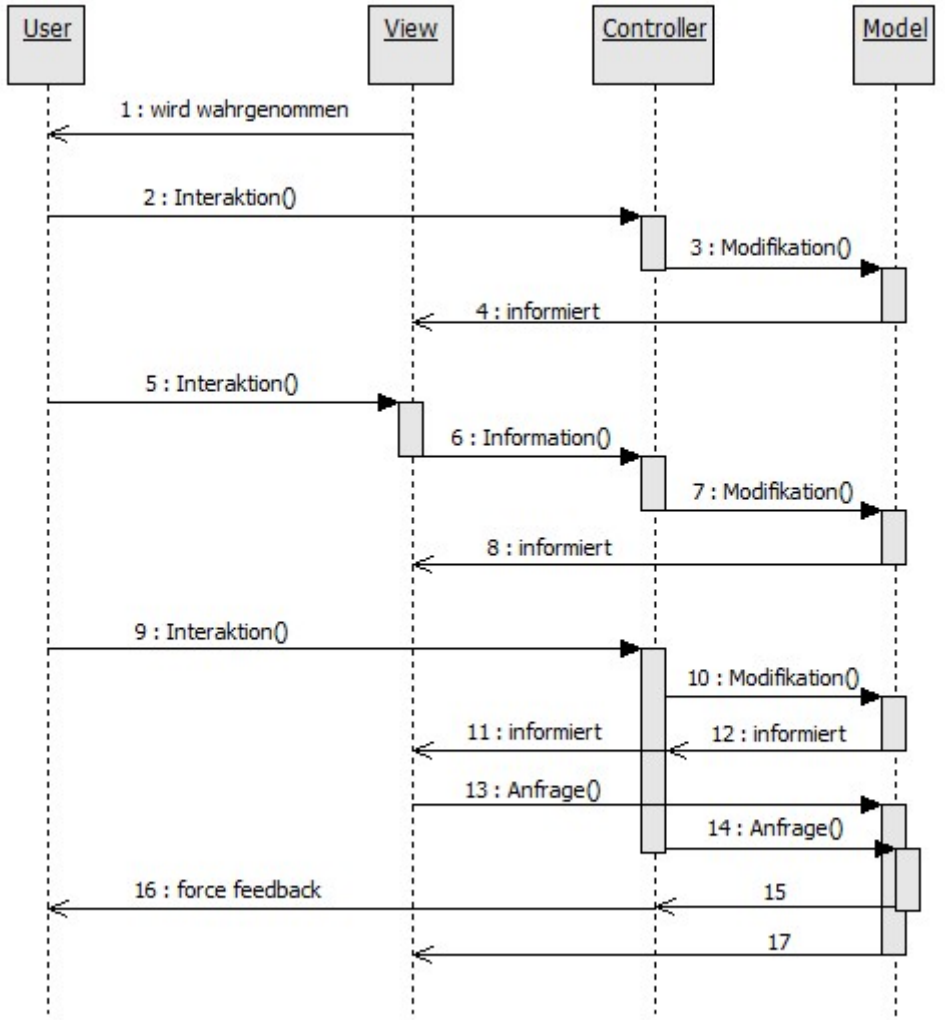
hat, dabei aber nicht mitteilt, welche Änderungen sich ergeben haben, müssen die Empfänger dieser Informations-Nachrichten gegebenenfalls beim Sender der Nachricht „nachfragen“, d. h. auf bestimmte, für sie interessante Daten explizit zugreifen.

Man beachte, dass Module immer auch mit gleichartigen Modulen kommunizieren können: Modelle mit Modellen, Views mit Views und Controller mit Controllern. Dies wird in der nebenstehenden Grafik nicht explizit dargestellt.

3.3 VCM-Prozess



VCM-Prozess



VCM-Prozess, Sequenzdiagramm

Die zuvor beschriebenen erweiterten Kommunikationsmöglichkeiten können sehr elegant als VCM-Prozess realisert werden. Das heißt, eine View leitet Benutzeraktionen direkt an einen geeigneten Controller weiter. Jeder Controller bereitet die eingehenden Daten geeignet auf und aktiviert die zugehörigen Kommandos zum Modifizieren von Modelldaten. Ob diese Kommandos (= Komponentenlogik) im Controller oder im Modell angesiedelt sind, hängt von der speziellen Umsetzung des VCM-Paradigmas ab. Jedes Modell informiert per Multicast-Nachricht alle interessierten Controller und Views. Diese können auf die Änderungen geeignet reagieren. Wenn das Model in seinen Nachrichten nicht alle Änderungsinformation mitschick, können die View und die Controller per direktem Zugriff nach weitere Informationen zur jeweiligen Änderung einholen.

Die direkte Kommunikation per Unicast-Nachrichten zwischen View und Controller, ziwchen View und Modell sowie zwischen Controller und Modell ist i. Allg. problemlos möglich, da in jedem Fall die Kommunikationspartner beim Initialisieren der Komponente eindeutig festliegen. Dies ist jedoch bei den Kommunikationspartnern der Modelle nicht der Fall: Die Anzahl der Views (und evtl. auch der Controller), die nach einer Modelländerung aktualisiert werden müssen, liegt nicht von vorn herein fest und kann sich jederzeit ändern. Daher ist in diesem Fall eine indirekte Kommunikation mit Hilfe von Multicast-Nachrichten deutlich sinnvoller.

3.3.1 Sequenzdiagramm

Im zugehörigen Sequenzdiagramm werden drei verschiedene Kommunikationsvorgänge exemplarisch dargestellt. Der Benutzer registriert Änderungen an einer View stets asynchron. Das heißt, die View hat keinen Einfluss darauf, wann der Benutzer welchen Teil der View betrachtet.

Der erste Kommunikationsvorgang ist relativ simpel. Der Benutzer aktiviert einen Controller (z. B. per Tastenklick). Dies hat eine Modifikation des zugehörigen Modells zur Folge. Das Modell informiert die zugehörige View per Multicast. Die daraus resultierende Änderung der View wird irgendwann vom Benutzer wahrgenommen.

Der zweite Kommunikationsvorgang unterscheidet sich vom ersten dadurch, dass der Benutzer mit der View interagiert. Diese muss die Benutzeraktion (z. B. Aktivierung eines Button per Mausklick) zunächst an einen Controller weiterleiten. Danach läuft der Kommunikationsprozess genauso wie zuvor ab.

Im dritten Fall hat die Benutzeraktion und die darufhin ausgeführte Modellmodifikation mehrere Aktionen zur Folge. Es werden sowohl der Controller, als auch die View über die Änderung informiert. Beide erfragen weitere Details über diese Änderung vom Modell, bevor sie ihren Zustand ändern, der dann jeweils vom Benutzer wahrgenommen werden kann.

3.3.2 Beispiel „Warenkorb“

In einer clientseitigen Warenkorb-Anwendung werden im Modell Daten über den Warenkorb und – sobald sich der Besteller eingeloggt hat – den zugehörigen Besitzer gespeichert: der Warenkatalog, eine aktuelle Auswahl des Warenkataloges, der Inhalt des Warenkorbs, der Gesamtpreis der Waren im Warenkorb, die Anschrift des Bestellers etc. Die Modelldaten oder zumindest Teile davon (wie z. B. der gesamte Warenkatalog) werden häufig in einer Datenbank abgelegt (siehe auch Model-View-Controller-Service-Paradigma).

Ein Darstellungsmodul visualisiert Modelldaten, wie z. B. die aktuelle Auswahl des Warenkataloges, mit Hilfe von Texten, Bildern und Videos. Der Benutzer kann die Auswahl der visualierten Elemente durch Filter oder Navigation abändern, er kann Elemente des Kataloges in seinen Warenkorb legen und dort wieder entfernen etc. Für all diese Aktionen werden dem Benutzer in der View spezielle Eingabefelder (Checkboxes, Drop-Down-Menüs, Textfelder, Links etc.) angeboten. Jedes mal, wenn der Benutzer eines dieser Elemente mit Hilfe der Maus oder der Tastatur bedient, leitet das zugehörige Darstellungsmodul eine entsprechende Nachricht an die Steuerung weiter.

Die Steuerung analysiert die gewünschte Aktion des Benutzers und führt die entsprechenden Änderungen im Modell durch. Zum Beispiel kann sie veranlassen, dass die aktuelle Auswahl des Warenkataloges den Wünschen des Benutzers gemäß geändert wird.

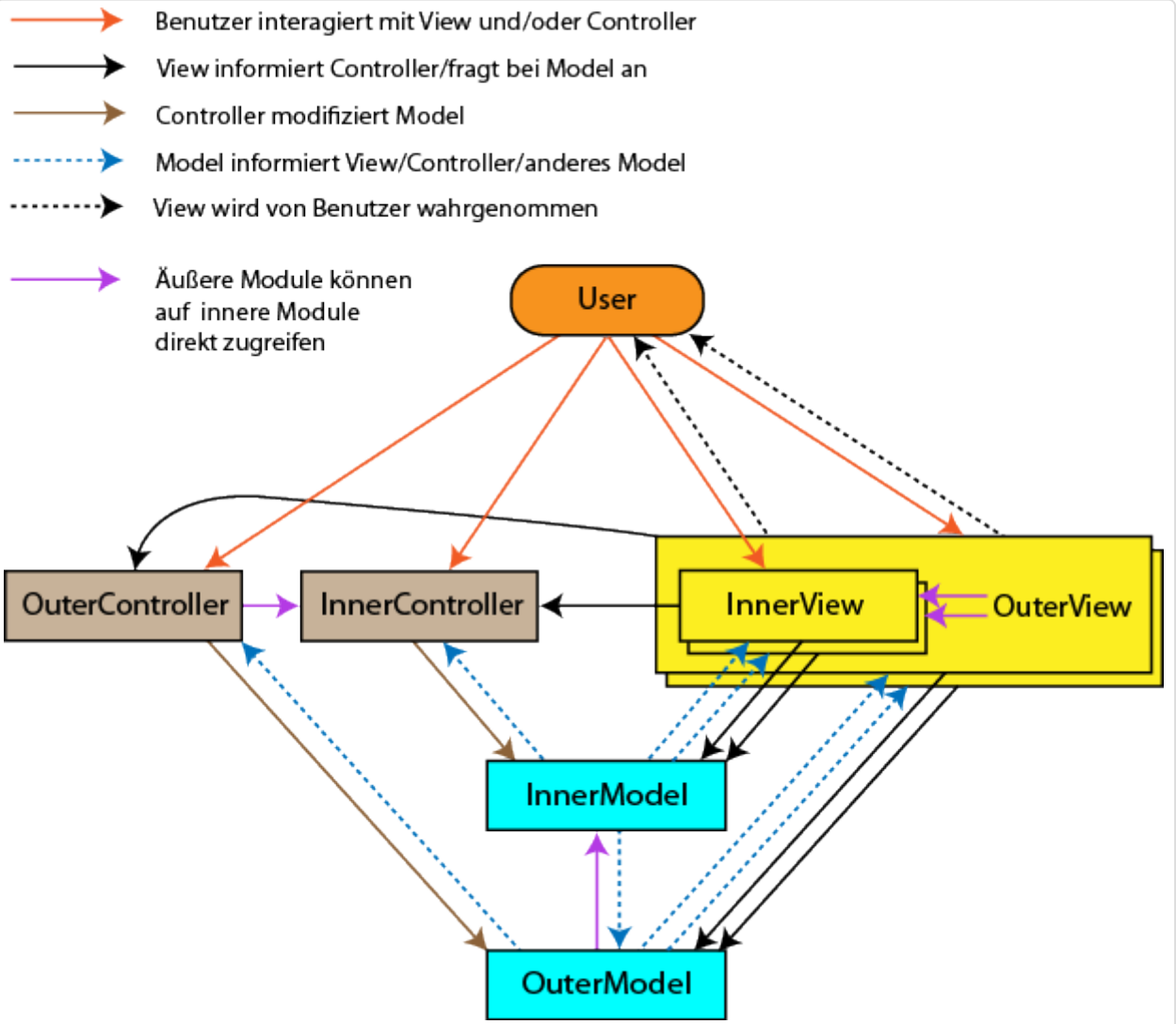
3.3.3 Beispiel „Advanture Game“

In einem Abenteuerspiel wird häufig sowohl die aktuelle Szene, als auch eine Übersichtskarte mit allen bereits erkundeten Bereichen visualisiert. Wenn an diesem Spiel mehr als ein Spieler beteiligt sind, gibt es sogar zahlreiche verschiedene Visualisierungen (aus Sicht eines jeden Spielers mindestens eine!).

Hier wäre eine direkte Kommunikation zwischen Modell und View extrem unpraktisch.

3.3.4 Kommunikation zwischen zwei VCM-

Komponenten



Zwei miteinander kommunizierende VCM-Prozesse

Auch der VCM-Prozess beschreibt das allgemeine Vorgehen häufig etwas ungenau. Viele Anwendungen bestehen aus mehreren VCM-Komponenten: Einer oder mehreren Kernanwendungen (Domain-Komponenten) sowie einer Rahmenanwendung (Frame-Komponente), die den Zugang zu und zwischen den Kernanwendungen steuert. In diesem Fall gibt es mehrere relativ unabhängige MVC-Prozesse. Die äußeren Komponenten (wie z. B. die Rahmenkomponente) können dabei mit den inneren Komponenten (wie z. B. die Kernkomponenten) kommunizieren. Der umgekehrte Weg sollte vermieden werden, damit Kernkomponenten problemlos in andere Umgebungen integriert werden können.

Man beachte, dass die innere View in die äußere View eingebettet werden kann. Ansonsten sind die einzelnen Komponenten möglichst eigenständig und kommunizieren nur über wohldefinierte und möglichst schlanke Schnittstellen miteinander.

3.3.4.1 Beispiel Jump-'n'-Run-Spiel mit Trainingsmodus und Highscore

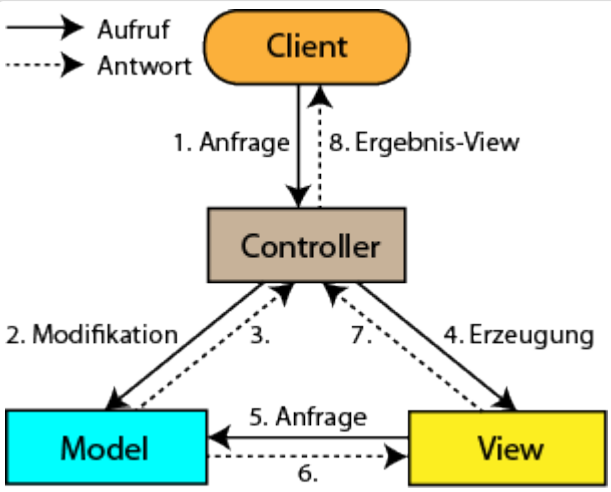
Man kann das Jump-'n'-Run-Spiel aus dem ersten Beispiel als Kernanwendung in einen Rahmen einbetten, der den Zugang zu diesem Spiel ermöglicht. Zum Beispiel kann die Rahmenanwendung verlangen, dass sich der Benutzer erst anmelden muss, bevor er mit dem Spiel beginnen kann. Danach kann der Benutzer wählen, ob er ein neues Spiel starten, ein altes Spiel fortsetzen oder ein

bestimmtes Level im Trainigsmodus üben will. Die Rahmenanwendung kann darüber hinaus das Spiel mit einer Highscore-Anwendung (eine weitere Kernanwendung) koppeln, die für beliebige Spiele benutzerspezifische Scores sowie jeweils einen Highscore verwaltet.

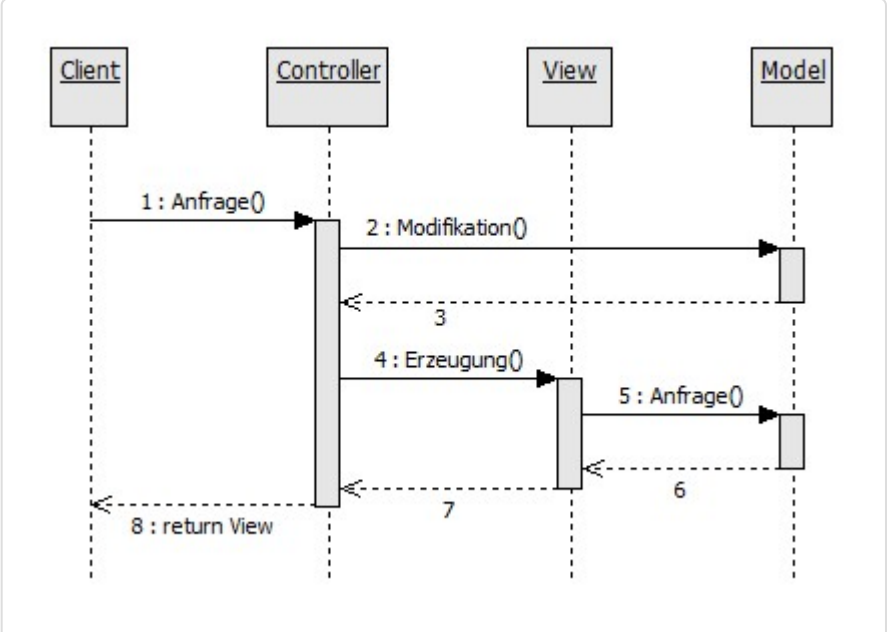
Wenn der Benutzer zum Beispiel den Trainigsmodus aktivieren möchte, kann die Steuerung der Rahmenkomponente zunächst mit Hilfe der Highscore-Anwendung ermitteln, ob der Benutzer schon genügend Punkte erspielt hat. Hierzu muss der die Highscore-Steuerung veranlassen, die gewünschten Daten im Higscore-Modell bereit zu stellen. Die Rahmen-Steuerung greift dann über das Rahmen-Modell darauf zu. Anschließend, sofern der Benutzer bereits genügend Spielerfahrung hat, leitet die Rahmen-Steuerung den Wunsch nach Aktivierung des Trainingsmoduses an die Steuerung des Spiels weiter.

Eines muss man bei dieser Art der Modellierung unbedingt beachten. Die innere Kernanwendung kennt die Rahmenanwendung nicht und kann daher auch auf keine Komponenten dieser Anwendung zugreifen. Das einzige, was sie machen darf, ist, der Rahmenanwendung Schnittstellen zur Verfügung stellen, über die die Rahmenanwendung auf die Kernanwendung zugreifen kann.

3.4 CVM-Prozess, Model 2



CVM-Prozess, Model 2[2]



CVM-Prozess, Sequenzdiagramm

Den meisten „Web Application Frameworks“ liegt heutzutage ein MVC-Paradigma zugrunde (vgl. MVC-Paradigma/Umsetzungen). Fast immer wird das MVC-Paradigma dabei als CVM-Schichtenarchitektur (oder genauer: als CVMS-Schichtenarchitektur, vgl. CVMS-Prozess) realisiert. Beispielsweise wurde in der Spezifikation von JSP 0.92[2][3] das so genannte Model 2 eingeführt, welches eine CVM-Architektur beschreibt (siehe nachfolgendes Beispiel).

Bei einem CVM-Prozess läuft die Kommunikation i. Allg. synchron ab: Ein Client (wie z. B. ein Web-Browser) übermittelt an einen Controller eine Anfrage (z. B. in Form einer URL plus weiteren Daten) und wartet dann auf eine Antwort: eine View, die er dem Benutzer präsentieren kann. (Dies gilt sogar, wenn Ajax zu Einsatz kommt. Auch hier wartet der Client auf eine Anwort ohne allerdings den Browser zu blockieren. Zu guter Letzt wird die als Ergebnis gelieferte Teilview vom Client verwendet, um die aktuell dem Benutzer präsentierte View zu aktualisieren.)

Um eine Antwort generieren zu können, leitet der Controller die erhaltenen Daten an das Modell weiter. Sobald das Modell die Daten vollständig verarbeitet hat (evtl. mit Hilfe von speziellen Service-Modulen – vgl. CVMS-Prozess), meldet es den Erfolg oder Misserfolg dieser Verarbeitung an den Controller.

Daraufhin stößt der Controler die Erzeugung (das Rendering) einer View an. Welche View erzeugt wird, hängt dabei von den Daten (insbesondere der URL) ab, die dem Controller übergeben wurden, sowie von der Antwort des Modells auf die Übermittlung dieser Daten. Wenn das Modell beispielsweise meldet, dass die Berechtigung zum Zugriff auf die gewünschten Daten fehlt, wird der Controller eine Login-View generieren, anderenfalls wird er eine View erzeugen, die die gewünschten Informationen präsentiert.

Der View-Renderer kann beliebig oft auf bestimmte Modelle zugreifen, um Daten zu erfragen, die in die View integriert werden sollen. Sobald der Rendervorgang abgeschlossen ist, erhält der Controller die erzeugte View. Der Controller leitet diese als Antwort an den Client zurück.

3.4.1 Beispiel „JSP“

Java-basierte Web-Frameworks, wie z. B. Apache Struts, JSF oder auch das Spring Web MVC framework, rendern die Views häufig mit Hilfe von JavaServer Pages (JSP). Als Modelle kommen in derartigen Frameworks normalerweise JavaBeans zum Einsatz. Sofern in einer Web-Anwendung die JSP-Seiten auch die Aufgaben des Controllers übernehmen, spricht man von einer **Model-1-Architektur**.[2][3]

Wenn in einem Web-Framework dagegen spezielle Servlets die Controller-Funktionalität übernehmen, d. h. die Views von dieser Funktionalität befreit werden, handelt es sich um eine CVM-Architektur, die im JSP-Umfeld allerdings **Model-2-Architektur** genannt wird.[2][3]

Die Controller werden im Falle einer Model-2-Architektur normalerweise entweder in der `web.xml` des zugehörigen Web-Server konfiguriert oder mit Hilfe von speziellen XML-Dateien. Die Controller-Konfigurationsdatei `struts.xml` einer Hello-World-Anwendung in `Struts` könnte beispielsweise folgendermaßen aussehen:

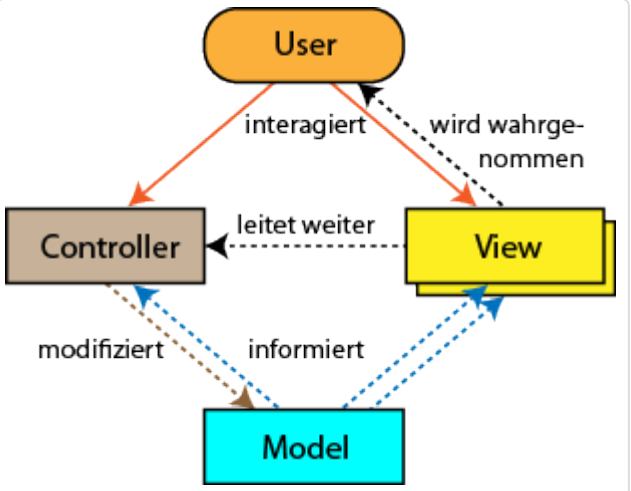
```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
  "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <package name="Login" extends="struts-default">
    <action name="LoginAction" class="hello.LoginAction">
      <result name="input">/Login.jsp</result>
      <result name="success">/HelloWorld.jsp</result>
    </action>
  </package>
</struts>
```

Die Klasse `hello.LoginAction`, übernimmt die eigentliche Control-Funktionalität. Von einem Objekt dieser Klasse wird die Methode `execute` aufgerufen. Diese Methode hat Zugriff auf die vom Benutzer übergebenen Daten, kann diese Daten in ein Modell eintragen oder anderweitig verarbeiten und liefert schließlich einen Ergebnis-String. Abhängig vom Wert dieses Strings (`input` oder `success`) wird die in der `struts.xml` spezifizierte View generiert.

Ein anderes Beispiel ist das Web MVC Framework von Spring[4]. Hier nimmt der so genannte Front Controller die Anfragen (Requests) vom Client entgegen und leitet diese an einen zweiten Controller weiter. Dieser erzeugt ein Modell-Objekt, initialisiert es und liefert es an den Front Controller zurück. Der Front Controller startet daraufhin einen View-Rendervorgang. Er übergibt dem View-Renderer das Model als Parameter, damit der Renderer, also die View, direkt darauf zugreifen kann.

> TO BE DONE
>
> Front Controller Pattern, insb. Spring

3.5 MVC-Multicast-Prozess



Alle Komponenten kommunizieren per

Multicast-Nachrichten

Wenn man die MVC-Komponenten besonders lose koppeln möchte, bietet sich eine reine Multicast-Kommunikation an. Vom Benutzer abgesehen, der stets explizit mit dem System kommuniziert, erfolgt jede Kommunikation per Multicast-Nachricht: Ein Sender schickt Nachrichten aus an beliebig viele, dem Sender zu Beginn in der Regel noch unbekannte Empfänger. Da die Empfänger nicht direkt (d. h. per Unicast-Nachricht) beim Sender weitere Details erfragen können, müssen die Sender mit jeder wichtigen Nachricht alle notwendigen Detail-Informationen mitschicken. Dieser Nachteil wird jedoch durch den Vorteil aufgewogen, dass beliebige Module als Nachrichten-Empfänger definiert werden können, ohne dass der jeweilige Sender modifiziert werden müsste. In der nebenstehenden Anwendung fungiert beispielsweie nicht nur ein Controller, sondern auch ein zweites View-Modul als Empfänger von Nachrichten des ersten View-Moduls.

Ein typischer Vertreter für diese Art von MVC-Kommunikation ist PureMVC[1].

3.5.1 Beispiele

Alle Anwendungen, die nach dem VCM-Paradigma erstellt werden, können alternativ auch mit reinen Multicast-Nachrichten realisiert werden.

3.5.2 Kommunikation zwischen zwei MVC-Multicast-Komponenten

Genauso wie mehrere VCM-Komponenten problemlos gekoppelt werden können, können auch mehrere MVC-Multicast-Komponenten gekoppelt werden. Die Kopplung ist sogar viel einfacher, da man einzelen Module der einen Komponenten einfach nur als Nachrichten-Empfänger von Modulen der anderen Komponente anmelden muss.

4 Bemerkungen

4.1 Vorteile

Das MVC-Paradigma ermöglicht ein flexibles Programmdesign, welches die Wiederverwendbarkeit der einzelnen MVC-Module und komplexer MVC-Komponenten sowie eine daraus resultierende reduzierte Gesamtkomplexität gewährleistet, insbesondere bei großen Anwendungen. Folgende Vorteile ergeben sich insbesondere:

Die Anwendungslogik ist von den dazugehörenden Darstellungen und den Benutzerinteraktionen klar getrennt: Seapration of Concern.
Ein Modell kann durch viele Darstellungsmodule repräsentiert werden (z. B. Detail-View und Übersichts-View wie z. B. eine Landkarte mit Akteuren, spielerspezifische Views bei Multiuser-Spielen, vierschiedene Ansichten eines 3D-Modells etc.).
Bei Multiuser-Spielen muss nur das Modell auf allen beteiligten Rechnenr synchronisiert werden. Eine realtive geringe Bandbreite reicht zur Übertragung aus.
Bestehende Systeme können einfach erweitert werden, indem neue Module und MVC-Komponenten hinzugefügt werden.

4.2 Nachteile

Bei kleinen Anwendungen bedeutet der Einsatz von MVC einen gewissen Mehraufwand.

5 Anwendungsgebiete

Allgemein: größere Softwareprojekte
GUI-Programmierung (bspw. Frameworks wie Swing)
Web-Anwendungen (bspw. Frameworks wie Ruby on Rails)
Beispiele für Umsetzungen des Model-View-Controller-Paradigmas

6 Quellen

1. PureMVC Framework
2. JavaServer™ Pages™ – Specification 0.92
3. Servlets and JSP Pages Best Practices
4. Spring Framework: Chapter 13. Web MVC framework

1. **Kowarschick (MMProg)**: Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)
2. **Reenskaug (1979)**: Trygve M. H. Reenskaug; Thing-Model-View-Editor – an Example from a planningsystem; http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf; 1979; Quellengüte: 2 (Web) (Erste Notiz mit exemplarischen Beispielen)
3. **Reenskaug (1979a)**: Trygve M. H. Reenskaug; Models-Views-Controllers; http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf; 1979; Quellengüte: 2 (Web) (Grundlegende Definition von MVC)
4. **Reenskaug (2003)**: Trygve M. H. Reenskaug; The Model-View-Controller (MVC) – Its Past and Present; http://home.ifi.uio.no/trygver/2003/javazone-jaoo/MVC_pattern.pdf; 2003; Quellengüte: 2 (Web)
5. **Stoiber (2005)**: Dietmar Stoiber; Implementierung des Model-View-Controller Paradigmas für das WeLearn-System (Web Environment for Learning); Hochschule: Johannes Kepler Universität; Adresse: Linz; Web-Link; 2005; Quellengüte: 3 (Diplomarbeit)
6. Martin Fowler: GUI Architectures

7 Siehe auch

Model-View-Controller-Paradigma/Umsetzungen
Model-View-Controller-Service-Paradigma
Logic-Data-View-Controller-Service-Paradigma
Homepage von Trygve M. H. Reenskaug, Erfinder von MVC
Reenskaugs eigene Seite über MVC
Wikipedia:Model View_Controller
WikipediaEN:Model–view–controller

Berkovitz (2006): Joe Berkovitz; An architectural blueprint for Flex applications; https://web.archive.org/web/20070105004310/http://www.adobe.com:80/devnet/flex/articles/blueprint.html; 2006; Quellengüte: 2 (Web)

Kategorien:
MVC
Objektorientierte Programmierung
Glossar
Kapitel:Multimedia-Programmierung