# Einsatz von JSP und Servlets im Vignette Content Mangement System (VCMS) unter Verwendung von Page-Caching

Studienarbeit
am
Fachbereich Informatik,
AB Softwaretechnik,
Universität Hamburg
Oktober 2002

Betreuerin:
Prof. Dr. Christiane Floyd

Roman Schlömmer
Meerweinstraße 17
22303 Hamburg
Matr.Nr.: 4 960 590

# Inhaltsverzeichnis


**1 Einleitung ........ 3**
**2 Theoretische Konzepte und Technologien des Content Management ........ 5**
2.1 Webseiten ........ 5
2.2 Web Content Management Systeme ........ 6
2.3 Servlets ........ 7
2.4 JavaServer Pages (JSP) ........ 7
2.5 Trennung von Darstellung, Inhalt und Logik ........ 8
2.6 Model-View Controller (MVC) ........ 8
2.7 Observer / Observable ........ 9
2.8 Interaktionskomponente (IAK) / Funktionskomponente (FK) ........ 10
**3 Trennung von Präsentation, Logik und Inhalt in Web-Anwendungen ........ 11**
3.1 Darstellung des Problems ........ 11
3.2 Einsatzgebiete von Page-Caching ........ 12
3.3 Konsequenzen von Page-Caching ........ 12
3.4 Umsetzung von Page-Caching ........ 13
3.5 Beispielanwendung ........ 13
3.5.1 Redaktionssystem der Mitarbeiterverwaltung ........ 13
3.5.2 Anzeigesystem der Mitarbeiterverwaltung ........ 14
3.5.3 Verhalten den Anwendung bei Benutzeraktivität ........ 15
**4 Lösungsansätze für einen Page-Caching Mechanismus ........ 16**
4.1 Reine JSP-Lösung ........ 16
4.2 Realisierung über Datenbank-Trigger ........ 16
4.3 Servlet / JSP Lösung mit Cache-Manager ........ 17
4.3.1 Spezifika bei der Verwendung von Servlets und JSPs ........ 19
4.4 Diskussion der vorgestellten Lösungsansätze ........ 20
**5 Architektur der Lösung ........ 22**
5.1 Verteilung der Verantwortlichkeit auf die Teile des Cache-Managers ........ 22
5.2 Szenario für das Update des Page-Cache ........ 23
5.3 Implementationsdetails ........ 24
5.4 Erforderliche Schritte bei Portierung und Weiterentwicklung ........ 28
5.5 Spezifika beim Vignette Content Management System (VCMS) ........ 29
**6 Resümee ........ 30**
**7 Literaturverzeichnis ........ 31**

# 1 Einleitung

Durch den technischen Fortschritt in Bezug auf Hardware und Software ist es in der Vergangenheit dazu gekommen, dass die Ansprüche an eine Website gestiegen sind. Mittlerweile dienen Seiten im Internet nicht mehr nur der reinen Darstellung von Informationen, sondern erlauben dem Benutzer das Interagieren sowie die Verwendung ganzer Anwendungen. Zusätzlich sollen immer mehr Informationen über das Internet verfügbar sein, so dass Last in Bezug auf Rechenzeit und damit Kosten in Bezug auf Hardware eine immer bedeutendere Rolle einnehmen.
Content Management Systeme wie Vignette können helfen, diese Probleme zu lösen. Zum einen können große Mengen von Informationen bereitgestellt werden und zum anderen werden Mechanismen geboten, die eine Optimierung der Kosten möglich machen. Durch die Verwendung von Page-Caching, einem Mechanismus, der den Zugriff auf dynamische Seiten im Internet effizienter macht, kann die Rechenzeit und somit indirekt der Hardwarebedarf gesenkt werden. Der Umstand, dass im Internet ganze Anwendungen abgebildet werden, die Internettechnologien selber aber keine Mechanismen zur Entwicklung von Anwendungen bereithalten, beziehungsweise sich nicht den üblichen Entwurfsmustern zur Anwendungsentwicklung bedienen, führt zu folgendem Problem: Wie können Anwendungen für das Internet entwickelt werden, die auf herkömmlichen Entwurfsmustern basieren? Die Tragweite des Problems wird deutlich, wenn man die Ursache betrachtet, warum Entwurfsmuster zur Anwendungsentwicklung herangezogen werden. Durch eine Entkopplung der einzelnen Teile einer Anwendung wird diese flexibler, weil die Teile ersetzt oder bearbeitet werden können, ohne dass dies Auswirkungen auf die anderen Komponenten im System nach sich zieht. Sowohl eine höhere Wiederverwendbarkeit der Komponenten, als auch die sinnvolle Aufteilung der Entwicklungsarbeit auf mehrere Programmierer, sind direkte Folgen des Einsatzes geeigneter Entwurfsmuster (vgl. [DC00]). Bestehende Internetanwendungen wurden häufig statt von Softwarespezialisten von Grafikdesignern entworfen. Deren Augenmerk liegt mehr auf einer ansprechenden Präsentation als auf einer guten Architektur. Abgesehen davon wurden die meisten Web-basierten Systeme „ad hoc“ entworfen und mittels „kontinuierlicher Patches am Laufen gehalten“ (aus [MDG01]). Daher sind herkömmliche Internetanwendungen Arbeitsplatzanwendungen in Bezug auf Flexibilität und Wiederverwendbarkeit unterlegen.
Es gilt deshalb, die neuen Technologien, die das Internet mit sich bringt, auf alte Entwurfprinzipien abzubilden, um nicht auf deren Vorzüge verzichten zu müssen (vgl. [DMH01]). Eine Frage, die es zu beantworten gilt, ist, welche Auswirkungen die Verwendung von Page-Caching auf die Architektur einer Anwendung hat. Je größer die Unterschiede sind, desto mehr Kosten entstehen durch steigende Entwicklungszeit, da die Entwickler sich dementsprechend mit neuen Gegebenheit konfrontiert sehen und nicht auf gewohnte Konzepte zurückgreifen können. Die Akzeptanz gegenüber solchen Systemen hängt darüber hinaus entscheidend davon ab, ob und mit welchem Aufwand schon bestehende Anwendungen darauf portiert werden können. Da beispielsweise der Performanzgewinn durch die Verwendung von Page-Caching eine Rolle bei der Entscheidung für die Portierung auf ein Content Management System spielen könnte, wäre im gleichen Zuge ein erhöhter Anpassungsaufwand die entkräftende Kehrseite.
Optimal wäre daher eine Lösung, die Page-Caching verwendet, aber auch von den gewohnten Entwurfsmustern Gebrauch macht. Dies sind insbesondere das aus dem Smalltalk-Kontext stammende Model-View Controller Entwurfsmuster (MVC) und das Observer / Observable Entwurfsmuster. Beide Muster dienen dazu, eine Anwendung wartbarer, wiederverwendbarer und modularer zu gestalten, da sie auf einer strikten Trennung von Logik, Präsentation und Steuerung basieren. Auch eine Unterteilung der Entwicklungsarbeit auf mehrere Programmierer mit unterschiedlichen Fachkenntnissen ist somit machbar. Der Mechanismus, der das Page-Caching realisiert, muss also entweder auf den genannten Techniken aufsetzen oder in diese integriert werden.
Eine mögliche Lösung wäre eine Instanz eines Cache-Managers, der sich als Teil der View-Komponente im MVC Entwurfsmuster bei der Anwendung anmeldet und das Löschen von nicht aktuellen Seiten aus dem Page-Cache übernimmt. Anbieten würde sich die Realisierung als Teil des Observer Musters, da dem Aktualisieren einer Oberfläche bei Desktopanwendungen und dem Löschen von Seiten aus dem Page-Cache bei Internet-Anwendungen dasselbe Prinzip zugrunde liegt. Wenn sich Daten ändern, wird der Cache-Manager darüber benachrichtigt und kann dann den Page-Cache auf den aktuellen Stand setzen, indem er die erforderlichen Seiten daraus entfernt.

Durch den Einsatz bekannter Entwurfsmuster ist es gelungen, neue Technologien mit gewohnten Paradigmen umzusetzen. Daher ist sowohl die Entwicklung neuer als auch die Portierung bestehender Anwendungen in kürzerer Zeit realisierbar.
Es wird bei der Lösung des Problems ebenfalls darauf Wert gelegt, einen nicht proprietären Ansatz zu finden, d.h. eine Übertragung auf andere Content Management Systeme soll mit möglichst geringem Aufwand erfolgen können.

Diese Studienarbeit befasst sich mit dem Thema Page-Caching von JavaServer Pages im Vignette Content Management System (VCMS). Im ersten Teil der Arbeit werden grundlegende Technologien und Konzepte erläutert und vorgestellt. Dieses Kapitel soll als Grundlage dienen, um die folgende Darstellung des Problems und die verschiedenen möglichen Lösungen nachvollziehen zu können. Es wird ein Überblick über das Anwendungsfeld gegeben, in dem die vorgestellte Architektur einsetzbar ist. Anschließend sollen die gesammelten Erkenntnisse diskutiert werden, um eine Lösung des Problems finden zu können. Diese wird im nächsten Abschnitt ausführlich anhand einer Beispielanwendung vorgestellt.
Im abschließenden Resümee wird Bilanz gezogen und auf eventuelle Schwierigkeiten bei der Umsetzung hingewiesen.

# 2 Theoretische Konzepte und Technologien des Content Management

In diesem Kapitel werden die verwendeten Konzepte und Technologien eingeführt und vorgestellt. Dies soll dazu dienen, die im Folgenden herausgearbeiteten und diskutierten Lösungsansätze besser nachvollziehen und fundieren zu können.

## 2.1 Webseiten

Will man über Webseiten Information bereitstellen, ist es von hoher Relevanz, wie beständig die Informationen sind, d.h. wie oft und schnell sich diese ändern. Bleiben die Informationen lange die gleichen, kann man die Seite als statische Datei hinterlegen. Ändern sich die Informationen aber häufig, sollte man die Seite, oder zumindest die sich ändernden Teile dynamisch halten, damit die neuen Informationen automatisch zur Verfügung stehen, ohne dass die Seite selber editiert werden muss.

- **Statische Webseiten**
  Wie weiter oben schon angedeutet, zeichnen sich statische Informationen durch genau einen Zustand aus, der nach Erstellung nur durch ein erneutes Editieren der Datei verändert werden kann.
  Als Beispiel kann man sich hier Biographien verstorbener Leute, geschichtliche Aufzeichnungen, Gebrauchsanweisungen, etc. vorstellen.
  Diese Informationen müssen nach dem Zusammentragen lediglich optisch aufbereitet zur Verfügung gestellt werden. Da sich danach der Inhalt nicht mehr ändert, kann man die fertige HTML-Datei dann im Dateisystem ablegen und verfügbar machen (vgl. [BTZZ01] Kap. 2).
  Statische Webseiten bilden die schnellstmögliche Art der Auslieferung, da der komplette Inhalt bereits fertig und formatiert vorliegt und nicht zur Laufzeit generiert werden muss. Dem gegenüber steht ein hoher Aufwand bei der Umgestaltung der Webseite oder Änderung des Inhalts, weil jede Seite auf die sich die Änderung auswirkt, zum Beispiel beim Austausch einer Grafik, einzeln editiert werden muss (vgl. [CID98] SS.1).

- **Dynamische Webseiten**
  Um hohe Aktualität zu gewährleisten, bieten sich statische Seiten allerdings nicht an. Ändern sich die Informationen nach bestimmten Ereignissen beziehungsweise Gesetzmäßigkeiten oder sogar stetig ist der Arbeits- und Verwaltungsaufwand, um diese Änderungen in statischen Seiten darzustellen, nicht akzeptabel. Man stelle sich als Beispiel eine Seite vor, die Börsenkurse, den Wasserstand, Lagerbestände oder die aktuelle Uhrzeit beinhaltet.
  Hier bedient man sich dynamischer Elemente, um die Anzeige der Informationen immer aktuell zu halten, ohne die Seite selber bearbeiten zu müssen. Diese zeichnen sich dadurch aus, dass sie den Code enthalten, der zur Laufzeit die darzustellenden Informationen in die Seite hineingenerieren. Die Informationen können zum Beispiel aus einer Datenbank kommen oder sich direkt zur Laufzeit ergeben.
  Seiten, die dynamische Elemente enthalten, werden auch als Templates bezeichnet, weil an dafür vorgesehenen Stellen die gewünschten Informationen eingefügt werden. Sie können statische Teile wie Überschriften, Bilder, etc. enthalten, und in ihnen wird auch die Darstellung der Informationen festgelegt.
  Es ist also nicht erforderlich, ganze Seiten dynamisch zu halten. Es besteht die Möglichkeit, bestimmte Teile der Seite statisch zu halten, während andere zur Laufzeit dynamisch berechnet werden. Dies wird auch als semidynamisch bezeichnet (vgl. [BTZZ01] Kap. 2).

## 2.2 Web Content Management Systeme

Web Content Management Systeme dienen dazu, die Veröffentlichung von Daten im Internet zu erleichtern und zu organisieren. Folgende Komponenten und Techniken sind Teil eines solchen Systems (aus [BTZZ01]):

Da eine dynamische Webseite nicht direkt verfügbar ist, sondern erst zur Laufzeit generiert wird, ist sie in Bezug auf Performanz einer statischen unterlegen. Um diesem Nachteil entgegenzuwirken, kann man sich der Technik des **Page-Cachings** bedienen, die es ermöglicht, eine einmal fertig generierte Seite statisch abzulegen und somit die Antwortzeit zu verringern (vgl. [IC97] S. 1). Man erreicht dadurch zum einen die Variabilität dynamischer, als auch die hohe Performanz statischer Webseiten. Ein Page-Cache lässt sich auf verschiedene Weisen realisieren. Einige Systeme schreiben einmal generierte Seiten als HTML-Dokumente auf die Festplatte, andere halten diese im Hauptspeicher bereit. Die Persistenz auf der Festplatte gespeicherter Dokumente ist auch noch einem Neustart des Systems gegeben, wohingegen der Page-Cache im Hauptspeicher hierbei verloren ginge. Die Funktionsweise eines Page-Cache kann am Beispiel des Vignette Content Management Systems (VCMS) folgendermaßen aufgefasst werden: bei jedem Aufruf einer Webseite, sucht das System an einer dafür vorgesehenen Stelle nach einer vorher abgelegten statischen Version dieser Seite. Sollte diese vorhanden sein, kommt diese direkt zur Anzeige. Liegt die gewünschte Seite noch nicht im Page-Cache, wird sie generiert und ausgeliefert. Nach dem Generieren wird die statische Version dann im Page-Cache abgelegt, um für erneute Aufrufe zur Verfügung zu stehen (vgl. [cmsWatchRes1]). Die Wahrscheinlichkeit, dass eine Seite mehrfach ausgewählt wird, hängt in hohem Maße von der Anzahl der Benutzer eines Systems ab (vgl. [Wan99] S. 38).
Eine typische Anwendung in einem Content Management System besteht aus einem **Redaktionssystem** und einem **Anzeigesystem**.
Im Redaktions- oder Autorensystem wird, wie der Name schon sagt, der Inhalt der Seite, also die Information verwaltet. Sie stellt Masken zur Verfügung, in denen Informationen in das System eingepflegt, vorhandene Informationen geändert oder gelöscht werden können. Durch die Verwendung intuitiver Systeme ist es auch Personen ohne besondere Kenntnisse von Internettechnologien möglich, Inhalte für das Internet zu erstellen und zu verwalten.
Dieser Teil einer Webanwendung enthält die Strategie zum Verwalten des Page-Cache. Ändert ein Redakteur den Inhalt einer Seite, wird dafür gesorgt, dass die Änderung sichtbar wird, in dem eventuell im Page-Cache vorhandene Versionen der Seite entfernt werden. Beim erneuten Aufruf wird dann die Seite neu generiert und enthält die Änderungen.
Das Anzeigesystem ist für die Darstellung der Informationen zuständig. Sie enthält Seiten, die die Informationen optisch aufbereitet und eventuell für den Benutzer angepasst zur Verfügung stellen. Diese Seiten, auch **Templates** genannt, werden dynamisch mit den vom Benutzer gewünschten Inhalten gefüllt und zur Anzeige gebracht. Im Gegensatz zu üblichen HTML-Seiten, bei denen Inhalt und Darstellung nebeneinander in einem Dokument stehen, setzt man in Content Management Systemen auf deren Trennung. Die Darstellung wird in den Templates gespeichert, in die auf Anfrage die gewünschten Informationen eingebunden werden. So kann ein Template Grundlage vieler Dokumente sein und eine Änderung in der Darstellung würde alleine durch Bearbeiten eines Templates erreicht. Die Inhalte werden an anderer Stelle, zum Beispiel in einer Datenbank, gehalten und bei Bedarf ausgelesen und in die Templates integriert.
Internet-Seiten oder Templates, die aus mehreren Teilen bestehen, können im **Components** aufgeteilt werden. Dabei wird jedes Component wie eine eigenständige Seite entwickelt und zur Laufzeit in den Kontext einer umgebenden Seite eingebettet. Diese Technik ist mehrstufig möglich, so dass ein Component selber wieder Components enthalten kann. Besonders bei der Verwendung von Page-Caching macht sich der Einsatz von Components bezahlt. Da Components einzeln in den Page-Cache geschrieben werden können, muss eine Seite, die aus mehreren Teilen besteht, nicht komplett neu generiert werden, wenn sich nur ein kleiner Teil ändert. Dieses führt zur zusätzlichen Entlastung der Ressourcen.
Ein weiterer Vorteil von Content Management Systemen besteht in der **zentralen Datenhaltung**. Dadurch können mehrere Anwendungen bzw. mehrere Templates auf die gleichen Daten zugreifen, und somit ist gewährleistet, dass eine Änderung der Daten sich auch konsistent verhält und in allen

Teilen sichtbar wird. Abgesehen davon muss der Entwickler eines Redaktionssystems sich nicht um eventuelle redundante Daten kümmern, die aktualisiert werden müssten.
Durch den Einsatz von **Personalisierung** ist es möglich, dass die dargestellten Inhalte auf einer Webseite dem Benutzer angepasst werden. Dies kann durch eine Verfolgung des Klickverhaltens oder durch Auswahl durch den Benutzer selber geschehen. Ist ein Benutzer beispielsweise an einem bestimmten Aktienkurs besonders interessiert, kann dieser auf der Startseite angezeigt werden oder auf einer anderen Seite mit mehreren Kursen an oberster Stelle erscheinen. Sinnvoll eingesetzt kann Personalisierung die Attraktivität einer Seite deutlich erhöhen, da dem Benutzer das wiederholte Navigieren durch dieselben Menüs, um an die von ihm gewünschte Stelle zu gelangen, abgenommen wird (vgl. [Rie00], [cmsWatchRes2]).
Um die Performanz des Systems zu verbessern, ist es möglich, **Load-Balancing** zu betreiben. Hierbei werden die Anfragen an das System nicht von einem einzigen Server abgearbeitet, sondern das System wird auf mehrere Rechner aufgeteilt. Bei der Delegation der Anfragen auf die verschiedenen Server handelt es sich um einen nicht deterministischen Prozess, das bedeutet, dass vorher nicht bekannt ist, welcher Server die Bearbeitung übernimmt. Somit ist bei der Verwendung von Load-Balancing zu beachten, dass die für die Abarbeitung der Anfrage erforderlichen Informationen wie Datenbankverbindungen, Sessions oder andere Kontexte für alle Server zugänglich sind.

## 2.3 Servlets

Servlets bilden die Java-Alternative zur Common Gateway Interface (CGI)-Programmierung. Sie werden auf einer Servlet-Engine ausgeführt und bilden eine Zwischenschicht zwischen einer vom Webbrowser oder einem anderen HTTP-Client kommenden Anfrage und auf einem HTTP-Server laufenden Anwendungen und/oder Datenbanken. Nach ihrem Aufruf haben Servlets folgende Aufgaben:
Sie lesen die vom Benutzer in Form eines Formulars übermittelten Informationen aus, generieren die gewünschten Ergebnisse, formatieren diese Ergebnisse (üblicherweise in eine HTML-Seite) und schicken diese meist in HTTP-Format an den Client zurück.
Zum Einsatz können Servlets kommen, wenn die anzuzeigende Seite auf Daten basiert, die vom Benutzer übermittelt werden, die Seite vielfachen Änderungen unterliegt oder die Seite Informationen aus Datenbanken oder anderen Server-seitigen Quellen verwendet. (vgl. [Hal01], [HC01])

## 2.4 JavaServer Pages (JSP)

JSP bietet die Möglichkeit, reguläres, statisches HTML mit dynamisch generierten Inhalten zu mischen. Die dynamischen Teile, hier bestehen diese aus Java-Code, werden zur Laufzeit in die statischen Teile hineingeneriert. Der Vorteil zu reinem HTML stellt sich in der Trennung von Darstellung und Inhalt dar. Eine einzige JSP kann verwendet werden, um viele Daten gleicher Natur wie etwa bei Mitarbeiter-Stammdaten, darzustellen. Ändert sich das Layout der Seite, muss dieses nur ein einziges Mal, nämlich in der JSP, angepasst werden. Reine Servlet-Lösungen und andere Common Gateway Interface (CGI)-Varianten haben wiederum den Nachteil, dass dort auch die statischen Teile gewissermaßen jedes Mal neu generiert werden, was zu Performanzverlust führt. Technisch ist eine JSP einem Servlet gleichzusetzen, da sie eine Weiterentwicklung der Java Servlet Technologie darstellt (vgl. [SunRec1]). Die Servlet-Engine, die aus der JSP ein HTML-Dokument generiert, übersetzt die JSP zunächst in ein Servlet mit reinen Java-Code. Beim ersten Aufruf wird dieser kompiliert und dann serverseitig ausgeführt. Das Ergebnis ist dann die fertige HTML-Seite, die an den Browser ausgeliefert wird. Beim erneuten Aufruf kann dann das vorher kompilierte Servlet ausgeführt werden und die Seite generieren. JavaServer Pages können wegen der erläuterten technischen Gegebenheiten verwendet werden, um Templates zu entwickeln (vgl. [Hal01], [Tur00]).

## 2.5 Trennung von Darstellung, Inhalt und Logik

Ziel der Trennung dieser verschiedenen Komponenten einer Anwendung ist es, die Instanzen, die diese Teile realisieren, voneinander zu entkoppeln und unabhängig zu machen. Dadurch erlangt man eine höhere Flexibilität bei Weiterentwicklungen, weil jede der Komponenten ausgetauscht werden kann, ohne dass Änderungen in den anderen Komponenten erforderlich sind. Außerdem kann nur so eine sinnvolle Aufteilung der Entwicklungsarbeit auf Personen mit Detailwissen in den erforderlichen Disziplinen erfolgen. So kann ein Spezialist für Web-Technologien die Gestaltung der Webseiten übernehmen und ein Java-Spezialist kann die Logik und die Datenbankzugriffe realisieren. Die Administration einer zugrundeliegenden Datenbank wiederum wird durch einen Datenbank-Spezialisten abgedeckt. (vgl. [KK01] S. 146)

## 2.6 Model-View Controller (MVC)

Ziel des MVC-Entwurfsmusters ist es, die Anwendungsobjekte (Model) von der Art der Präsentation für den Benutzer (View) und der Steuerung durch den Benutzer (Controller) zu trennen. Durch die gegenseitige Entkopplung werden eine höhere Flexibilität und bessere Möglichkeiten zur Wiederverwendung gegeben. Im Folgenden werden kurz die Teile des Musters erläutert (aus [KP88]):

- **Model**
  Das Model hält sämtliche Daten vor, die zur Anzeige kommen können. Nur das Model erlaubt es mit geeigneten Operationen die Objekte zu verändern. Es enthält weder Informationen darüber, wie die Daten dargestellt werden, noch welche Benutzeroperationen benutzt werden, um die Daten zu ändern. Die Methoden, die die Daten verändern sind von der Darstellung entkoppelt und somit völlig unabhängig davon.

- **View**
  Die View oder die Ansicht referenziert das Model. Es führt darauf definierte Methoden aus, um die Daten abzufragen und diese dann darzustellen. Die Art und Weise, wie die Daten zur Anzeige kommen, geht alleine aus der View-Komponente hervor.

- **Controller**
  Der Controller wiederum empfängt die Benutzeraktivität und leitet diese an die für die gewünschte Operation erforderliche Methode im Model weiter. Der Controller bildet also die Kopplung zwischen View und Model.

Abbildung 1 zeigt anschaulich, wie die einzelnen Teile der Anwendung im MVC-Entwurfsmuster zusammenhängen und interagieren. Die Vorteile des MVC-Entwurfsmuster liegen in der in 2.5 erläuterten Trennung der Komponenten Anwendungsobjekte, Logik und Präsentation. Es besteht die Möglichkeit, auf die gleichen Anwendungsobjekte mit unterschiedlichen Views zuzugreifen und diese unterschiedlich darzustellen. Bei einer Änderung der Anwendungsobjekte müssen lediglich alle Views benachrichtigt werden, sich zu aktualisieren.

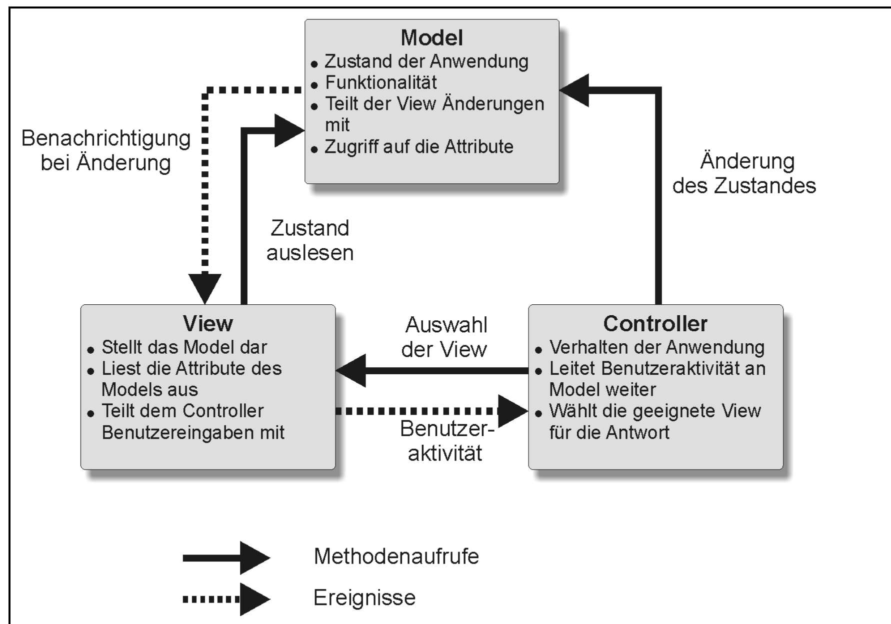


Abb. 1: MVC-Entwurfsmuster

## 2.7 Observer / Observable

Um eine abstrakte Kopplung von Inhalt und Darstellung zu realisieren, bedient man sich häufig des Observer Entwurfsmusters. Das Objekt (Observable), welches den Inhalt darstellt weiß lediglich, dass es eine Liste von Observern besitzt, die sämtlich die gleiche einfache Schnittstelle besitzen. Ändert sich der Inhalt, benachrichtigt der Observable alle in der Liste beinhalteten Observer und fordert diese auf, sich zu aktualisieren, in dem diese die Daten vom Observable neu einlesen. Somit hat man die Kopplung der beiden Teile auf ein Minimum reduziert und kann diese unabhängig voneinander variieren. Beinhaltet das Observable beispielsweise den Ausgang einer Wahl, können, wie in Abbildung 2 dargestellt, verschiedene Observer auf die gleichen Daten zurückgreifen, und diese unterschiedlich darstellen.
Eine Anwendungsmöglichkeit für dieses Entwurfsmuster ist die Umsetzung des Model-View Controller Entwurfsmuster. Dabei kann das Model als Observable und die View als Observer aufgefasst und umgesetzt werden (aus [GHJV96]).

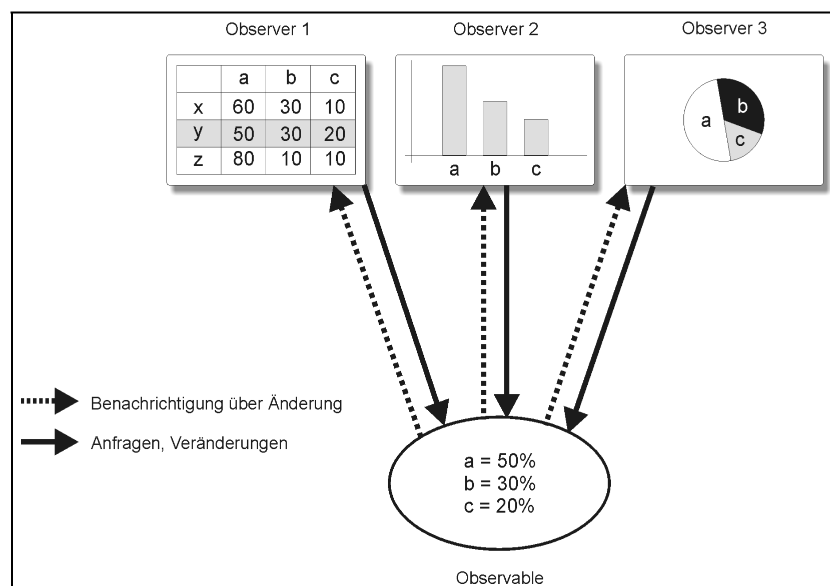


Abb. 2: Observer / Observable (aus [GHJV96])

## 2.8 Interaktionskomponente (IAK) / Funktionskomponente (FK)

Eine Anwendung kann in eine Interaktionskomponente, die die Präsentation und Interaktion realisiert, und Funktionskomponente, die für die Funktionalität zuständig ist, unterteilt werden, so dass beide Teile softwaretechnisch weitgehend unabhängig voneinander modelliert und konstruiert werden können. Somit greift dieses Konzept die in 2.5 erläuterte Trennung der einzelnen Komponenten eines Systems auf.
„Die Entkopplung der IAK von der FK wird dadurch realisiert, dass die FK keinerlei Kenntnis von der IAK hat, wohl aber umgekehrt die IAK die FK kennt. Somit ist eine Kontrollflussrichtung vom Benutzer in das System gewährleistet.“

„Die Funktionskomponente ist der bewirkende und sondierende Teil des Werkzeugs. In ihr wird die fachliche Funktionalität des Werkzeugs festgelegt. Die FK bearbeitet das Material und kennt den Arbeitszusammenhang, der durch das Werkzeug unterstützt wird. Zur Bearbeitung von Materialien benutzt die Funktionskomponente Operationen, die in einer oder mehreren Aspekten spezifiziert sind. Um den Arbeitszusammenhang unterstützen zu können, verwaltet die FK einen Arbeitszustand, das Werkzeuggedächtnis. [...] Die Interaktionskomponente legt die Benutzungsschnittstelle des Werkzeugs fest. Dazu nimmt sie Ereignisse entgegen, ruft die FK und steuert die Präsentation an der Oberfläche....“ (aus [Zül98]). In einer Architektur, die auf dem MVC-Entwurfsmuster basiert, kann die View zusammen mit dem Controller als Interaktionskomponente betrachtet werden, und das Model stellt die Funktionskomponente dar und realisiert den fachlichen Service. Dies wird in Abbildung 3 veranschaulicht.

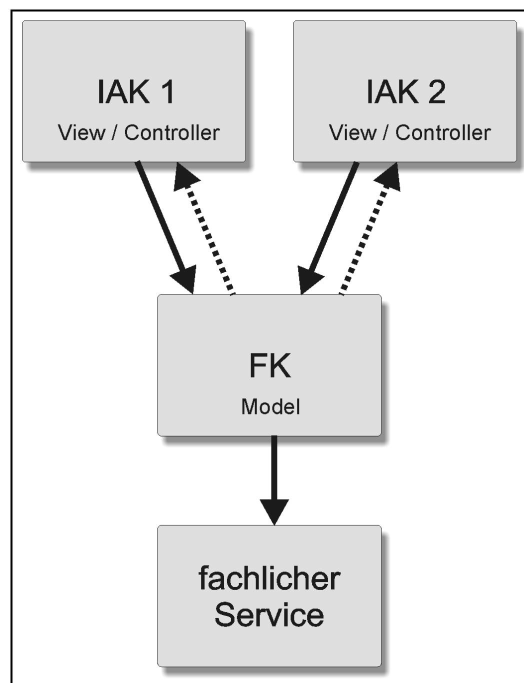


Abb. 3: Aufteilung der MVC-Komponenten auf IAK und FK

# 3 Trennung von Präsentation, Logik und Inhalt in Web-Anwendungen

Dieses Kapitel greift die in der Einleitung erwähnte Problemstellung „Wie können Anwendungen für das Internet entwickelt werden, die auf herkömmlichen Entwurfsmustern basieren?“ auf und führt diese am Beispiel Page-Caching weiter aus. Es wird erläutert, in welchen Anwendungsfeldern Page-Caching zum Einsatz kommen kann und warum diese Technik überhaupt eingesetzt wird.

## 3.1 Darstellung des Problems

Die Ansprüche an eine Webseite definieren sich durch die darzustellenden Inhalte. Diese können mannigfaltiger Natur sein. Im Gegensatz zu den Anfangstagen des Internet, in denen eine Webseite lediglich die Präsenz - zum Beispiel eines Unternehmens - im Internet realisierte, ist die heutige Rolle eine weit größere. Eine Webseite stellt nicht mehr nur Inhalt dar, sondern bietet die Oberfläche für komplexe Anwendungen im Internet. Oft bildet eine Webseite die Hauptkommunikationsplattform zwischen dem Unternehmen und seinen Kunden. Dieser Aspekt, wie auch die kurzen Änderungszyklen und große Mengen von Informationen, machen die Verwendung von bewährten Entwurfsmustern bei der Erstellung von Webseiten erforderlich (vgl. [Per01] S. 3). Dennoch findet man in heutigen Web-Anwendungen eine Vermischung von Präsentation, Logik und Inhalt vor. Die Ursache dafür liegt in der Entwicklung der Internet-Technologien und deren Verwendung. Viele Webseiten sind mit der Zeit gewachsen, basieren aber noch auf alten Technologien. Die Erfordernisse, damit eine Anwendung im Internet erfolgreich eingesetzt werden kann, sind die gleichen wie bei einer Desktop-Anwendung, speziell die Entkopplung der einzelnen Komponenten des Systems (vgl. [KK01] SS. 137). Da die Technologien und Paradigmen für Internetseiten nicht in gleichem Maße gewachsen und weiterentwickelt worden sind wie die Bedürfnisse, sieht man sich jetzt vor der Aufgabe, Anwendungen im Internet auf gewohnte Entwurfsmuster abzubilden, um deren bekannte Vorzüge nutzen zu können. Wie in Abbildung 4 dargestellt, macht die Entkopplung der einzelnen Komponenten diese wartbarer, flexibler und wiederverwendbarer, im Gegensatz zu einer Lösung, bei der die Komponenten vermischt vorliegen. Abgesehen davon kann die Entwicklung einer Web-Anwendung so auf mehrere Mitarbeiter mit den jeweils erforderlichen Fachkenntnissen aufgeteilt werden. Ein HTML-Programmierer kann die Präsentation, ein Java-Entwickler die Logik und eine Datenbank-Spezialist die Verwaltung der Inhalte realisieren.
Neben den genannten Vorteilen hat die Aufteilung des Systems in Komponenten aber einen entscheidenden Nachteil gegenüber statischen HTML-Seiten. Die erforderliche Generierung der Seiten zur Laufzeit wirkt sich negativ auf die Performanz einer Webseite aus, weil die Informationen bei jedem Aufruf aus der Datenbank geholt, formatiert und in die Templates eingefügt werden muss. Die daraus resultierende erhöhte Rechenlast hat höhere Ansprüche an die Hardware zur Folge.

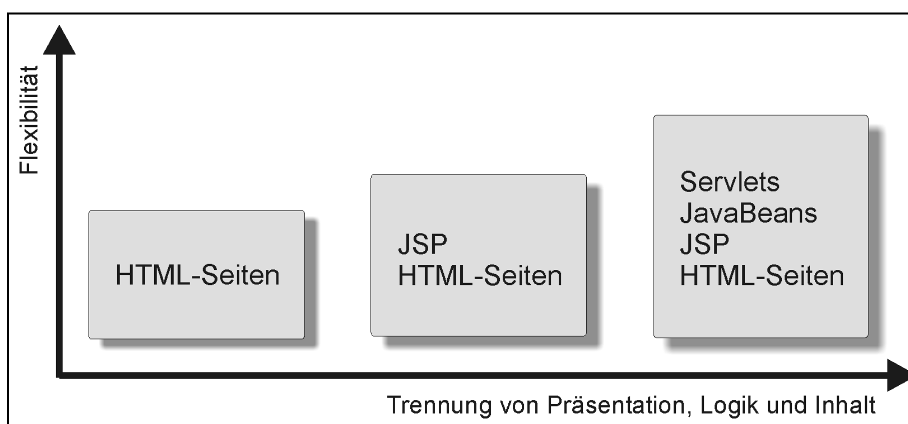


Abb. 4: Auswirkung der Entkopplung von Präsentation, Logik und Inhalt auf die Flexibilität einer Web-Anwendung

Eine Möglichkeit, die diesen erheblichen Nachteil entkräften kann, ist die Verwendung von Page-Caching, welches allein durch softwaretechnische Mittel die Performanz einer Webseite verbessern kann.

## 3.2 Einsatzgebiete von Page-Caching

Grundsätzlich ergibt die Verwendung von Page-Caching bei allen Webseiten Sinn, die mit dynamischen Seiten oder Elementen (Components) arbeitet. Wie in Kapitel 2.2 erläutert, wird hierbei eine dynamische Seite beim ersten Aufruf generiert und danach als statische Version abgelegt. Somit kann beim erneuten Aufruf direkt auf die statische Version zurückgegriffen werden. Dies bedeutet einen - je nach Komplexität der Seite - nicht unerheblichen Performanzgewinn gegenüber der Generierung. Dieser Performanzgewinn lässt sich ansonsten nur durch bessere und somit teurere Hardware realisieren. Durch die Verwendung von Page-Caching kann also direkt unnötige Rechenlast vermieden werden, und die Kosten für ein System können gesenkt werden.
Besonders zum Tragen kommt der Einsatz von Page-Caching bei Webseiten, die große Mengen von Informationen bereitstellen und bei Seiten, die eine Anwendung realisieren. Bei großen Mengen von Informationen macht das Speichern der Daten direkt in HTML-Dokumenten keinen Sinn mehr, da schon eine Umstellung des Layouts ein Bearbeiten aller Seiten nach sich ziehen würde. Hier wird daher mit Templates gearbeitet, in die die gewünschten Daten hineingeneriert werden. Die Daten selber werden dabei zum Beispiel in einer Datenbank gehalten. Auch bei Anwendungen, die im Internet angeboten werden, kommen Templates zum Einsatz. Die Maske einer Desktopanwendung kann mit Webtechnologien als Template umgesetzt werden. Wählt der Benutzer einen bestimmten Datensatz aus, kommt dieser zur Anzeige, indem die Anwendung die erforderlichen Daten aus einer Datenbank ausliest, diese in ein Template hineingeneriert und die fertige Seite dann zur Anzeige bringt. Beim erneuten Aufruf entfällt bei der Verwendung von Page-Caching zum einen der Datenbankzugriff und auch die Generierung der Seite.
Bei großen Datenmengen wie auch bei Webanwendungen kommen Web Content Management Systeme zum Einsatz. Diese Systeme liefern die Grundlagen, um eine Webseite, die im Hochlastbereich arbeitet, umzusetzen. Die grundlegenden Konzepte eines Web Content Management Systems sind in Kapitel 2.2 dargestellt. Je nach Anbieter können diese in Umfang und Umsetzung variieren.

## 3.3 Konsequenzen von Page-Caching

Neben den Performanzgewinnen bringt der Einsatz von Page-Caching allerdings auch gewisse Probleme mit sich, die es zu lösen gilt, um die genannten Vorzüge nicht direkt wieder zu entkräften.
Es gibt zwei Szenarien, die die Aktualisierung der Anzeige einer Webseite zur Folge haben. Dies ist bei der Verwendung von Page-Caching dem Löschen der statischen Seite aus dem Page-Cache gleichzusetzen, da es nicht möglich ist, die auf dem Browser angezeigte Seite direkt zu aktualisieren (vgl. [VgnRes2] S. 4).

- **Änderung des Templates**
  Ändert sich ein Template, dies ist zum Beispiel der Fall, wenn ein neues Feld hinzukommt oder eine Grafik verschoben wird, muss der Page-Cache dieses Templates geleert werden. Dies bedeutet, dass alle statischen Seiten, die auf diesem Template basieren, aus dem Page-Cache gelöscht werden müssen, damit sich die Änderung in allen Seiten niederschlägt. Beim erneuten Aufruf einer Seite, die auf dem geänderten Template basiert, wird diese neu generiert und enthält somit alle Änderungen.

- **Änderung der Daten**
  Ändert sich nur ein einziger Datensatz, ist es hingegen erforderlich, alle im Page-Cache vorhandenen Seiten zu löschen, die Daten des geänderten Datensatzes enthalten. Diese Seiten basieren in der Regel auf unterschiedlichen Templates. Beim Löschen eines Datensatzes sollten ebenfalls alle darauf beruhenden Seiten entfernt werden, um Speicherplatz zu sparen und um eine unerwünschte Anzeige gelöschter Daten zu verhindern. Dazu könnte es

kommen, wenn ein Benutzer die URL direkt eingibt und nicht über Links versucht, auf die Seite zuzugreifen oder ein Szenario wie in Kapitel 3.5.3 auftritt.

## 3.4 Umsetzung von Page-Caching

Die meisten Web Content Management Systeme nehmen einem das Page-Caching ab. Das heißt, es kann für jedes Template einzeln festgelegt werden, ob Page-Caching betrieben werden soll. Die generierten Seiten werden dann im Hintergrund in den Page-Cache geschrieben, wenn sie das erste Mal generiert worden sind. Anders sieht dies aber beim Löschen von Seiten aus dem Page-Cache aus. Beim Ändern eines Templates kann das System dafür sorgen, dass der Page-Cache aktuell gehalten wird, sprich die auf dem geänderten Template basierenden Seiten entfernt werden. Das Verwalten des Page-Cache bei Änderung der Daten wird einem allerdings nicht abgenommen. Hier hat der Entwickler einer Anwendung dafür Sorge zu tragen, dass die erforderlichen Seiten gelöscht werden. Die Logik zum Verwalten des Page-Cache ist normalerweise Teil des Redaktionssystems.

## 3.5 Beispielanwendung

Die in diesem Kapitel vorgestellte Anwendung soll die Konzepte und Konsequenzen von Page-Caching verdeutlichen. Sie ist bewusst einfach gehalten und konzentriert sich auf die in den vorangegangenen Kapiteln genannten Aspekte.
Als Beispiel ist eine Anwendung gewählt worden, die Mitarbeiterstammdaten mithilfe eines Redaktionssystems verwaltet und diese mit einem Anzeigesystem darstellt. Die Besonderheit dieser Anwendung liegt in der Tatsache, dass die Änderung eines Datensatzes nicht direkt in die Datenbank geschrieben wird, sondern zunächst im Anwendungskontext zur Laufzeit erhalten bleibt. Erst wenn der Benutzer alle Änderungen gemacht hat, werden alle geänderten beziehungsweise neuen Datensätze auf einmal aktualisiert, eingefügt oder gelöscht.

### 3.5.1 Redaktionssystem der Mitarbeiterverwaltung

Das Redaktionssystem der Anwendung besteht aus zwei Masken. Eine dient zur Auswahl eines Mitarbeiters, die andere zum Bearbeiten von Mitarbeiterstammdaten.

- **Auswahlmaske**
  Die Auswahlmaske beinhaltet neben Buttons zum Anlegen eines neuen Mitarbeiters (*Neu*), zum Bearbeiten eines vorhandenen Mitarbeiters (*Bearbeiten*) und zum Löschen eines Mitarbeiters (*Löschen*) noch die Liste aller im System vorhandenen Mitarbeiter. Die Daten, die je Mitarbeiter angezeigt werden sind der Nachname, der Vorname und der Status. Der Status bezieht sich auf den jeweiligen Bearbeitungszustand der Mitarbeiter. Der Button *Änderungen speichern* speichert die Änderungen in die Datenbank.

  Anmerkung:
  Der Button *Änderungen speichern* hat seinen Ursprung in einer Anwendung, die der Mitarbeiterverwaltung zugrunde liegt. In dieser Anwendung war es erforderlich, dass alle Änderungen auf einmal in die Datenbank geschrieben wurden.

- **Eingabemaske**
  Die Eingabemaske enthält die Felder zum Bearbeiten eines Mitarbeiters. Der Button *Änderung speichern* speichert die Änderungen, *Reset* verwirft alle Änderungen seit dem Öffnen der Maske und *Zurück* kehrt ohne Aktion zur Mitarbeiterauswahl zurück.

  Anmerkung:
  Der Button *Änderung speichern* schreibt die Änderungen noch nicht in die Datenbank, sondern speichert diese zunächst im Anwendungskontext. Erst durch Klick auf den Button *Änderung speichern* in der Mitarbeiterauswahl werden die Daten in die Datenbank geschrieben.

Neu | Bearbeiten | Löschen | Änderungen speichern

**Mitarbeiterauswahl:**
*Nachname, Vorname (Status)*

Knoxville, Johnny (keine Aktion)
Koch, Beate (neu)
Krause, Karsten (keine Aktion)
Müller, Markus-Oliver (löschen)
Schlömmer, Roman (keine Aktion)
Schneider, Ingo (neu)
Schulze, Stefan (keine Aktion)
Schumacher, Martin (keine Aktion)
Weintraub, Shlomo (keine Aktion)

Zur Zeit gewählter Mitarbeiter: kein Mitarbeiter ausgewählt

Abb. 5: Auswahlmaske des Redaktionssystems der Mitarbeiterverwaltung

Änderung speichern | Reset | Zurück

**Vorname:** Johnny
**Nachname:** Knoxville
**Geburtsdatum:** 01.02.1977

Abb. 6: Eingabemaske des Redaktionssystems der Mitarbeiterverwaltung

### 3.5.2 Anzeigesystem der Mitarbeiterverwaltung

Im Gegensatz zum Redaktionssystem besteht auch das Anzeigesystem der Mitarbeiterverwaltung nur aus einer Maske. Diese dient zur Auswahl und zur Anzeige eines Mitarbeiters.

- **Auswahl- und Anzeigemaske**
  Die Auswahl- und Anzeigemaske enthält eine Liste mit allen im System vorhandenen Mitarbeitern, den Button *Mitarbeiter anzeigen* und die Daten des ausgewählten Mitarbeiters. Der Bereich *Mitarbeiterdaten* ist als Component realisiert und wird zur Laufzeit mit den Daten des jeweilig gewählten Mitarbeiters befüllt.

Mitarbeiter anzeigen

**Mitarbeiterauswahl:**
***Nachname, Vorname***

Knoxville, Johnny
Koch, Beate
Krause, Karsten
Müller, Markus-Oliver
Schlömmer, Roman
Schneider, Ingo
Schulze, Stefan
Schumacher, Martin
Weintraub, Shlomo

**Mitarbeiterdaten:**

**Vorname:** Johnny
**Nachname:** Knoxville
**Geburtsdatum:** 01.02.1977

Zur Zeit gewählter Mitarbeiter: Knoxville, Johnny

Abb. 7: Auswahl- und Anzeigemaske des Anzeigesystems der Mitarbeiterverwaltung

### 3.5.3 Verhalten der Anwendung bei Benutzeraktivität

Dieses Kapitel erläutert, wie sich die Benutzeraktivität auf die Anwendung auswirkt. Besonderes Augenmerk gilt hier dem Page-Caching.
Für die Templates, die der Auswahl- und Anzeigemaske des Anzeigesystems zugrunde liegen, ist jeweils festgelegt worden, dass für die generierten Seiten Page-Caching betrieben werden soll. Das bedeutet, dass sowohl die Auswahlliste als auch die Datenmaske für jeden Mitarbeiter nach dem ersten Aufruf der jeweiligen Seite als statische Version in den Page-Cache geschrieben werden. Werden über das Redaktionssystem die Daten eines Mitarbeiters geändert oder ein Mitarbeiter wird gelöscht, muss sowohl die Auswahlliste des Anzeigesystems, wie auch die Datenmaske des editierten beziehungsweise entfernten Datensatzes aus dem Page-Cache gelöscht werden. Somit ist gewährleistet, dass bei einem erneuten Aufruf des Anzeigesystems keine zuvor entfernten Mitarbeiter in der Auswahlliste stehen, die Daten in der Auswahlliste aktuell sind und die Datenmaske eines editierten Datensatzes beim erneuten Aufruf neu generiert wird und somit aktuelle Daten enthält.
Bei paralleler Arbeit am Redaktionssystem und am Anzeigesystem kann unter Umständen die Situation entstehen, dass die Auswahlliste der Anzeigesystems noch Mitarbeiter enthält, die zwischenzeitlich aus dem System entfernt worden sind. Zu diesem Fall kann es kommen, weil zwar die Auswahlliste des Anzeigesystems aus dem Page-Cache entfernt wird, aber kein Einfluss auf die im Browser angezeigten Dokumente genommen werden kann. Dieses Szenario erfordert ein spezielles Verhalten der Anwendung bei Auswahl eines gelöschten Mitarbeiters im Anzeigesystem. Da die Datenmaske nicht mehr vorhanden ist, weil diese, wenn vorher überhaupt vorhanden, aus dem Page-Cache entfernt wurde, muss diese neu generiert werden. Beim Zugriff auf die Datenbank stellt das System aber fest, dass der Mitarbeiter nicht vorhanden ist, und gibt daher eine Meldung aus, die eben besagt, dass der gewählte Mitarbeiter aus der Datenbank entfernt wurde. Dieses Szenario verdeutlicht warum es erforderlich ist, auch die Seiten, die sich auf gelöschte Daten beziehen, aus dem Page-Cache zu entfernen. Da das Anzeigesystem zunächst nach einer statischen Version einer Seite im Page-Cache sucht, wäre es dem Anzeigesystem nicht möglich festzustellen, dass ein Mitarbeiter gar nicht mehr vorhanden ist, weil der entsprechende Datenbankzugriff gar nicht zustande käme.

# 4 Lösungsansätze für einen Page-Caching Mechanismus

Im aktuellen Kapitel werden verschiedene Ansätze vorgestellt, bewertet und diskutiert, die die Verwendung von Page-Caching ermöglichen. Es werden die Vor- und Nachteile der unterschiedlichen Ansätze erläutert, und es soll ein Modell gefunden werden, dass die gewünschten Ergebnisse am besten erzielen kann. Diskutiert werden die reine JSP-Lösung und die Realisierung über Datenbank-Trigger, als gängige Methoden und eine Servlet / JSP-Lösung mit Cache-Manager als Alternative.

## 4.1 Reine JSP-Lösung

Reine JSP-Lösungen stellen – abgesehen von statischen HTML-Seiten - die einfachste Gattung von Anwendungsarchitekturen dar.
Bei diesem Ansatz ist auch das Page-Caching besonders einfach. Nachdem im Redaktionssystem Daten eingegeben worden sind, wird in einer JSP der Code zum Löschen des Page-Cache direkt ausgeführt, oder eine Methode einer Klasse die den Code beinhaltet wird aufgerufen. Unbedingt zu beachten ist bei diesem Vorgehen, dass die Seiten, deren Code jedes Mal ausgeführt werden soll, nicht selber in den Page-Cache geschrieben werden. Dadurch würde ein statische Version abgelegt, die den auszuführenden Code nicht mehr enthält. Dieser käme somit nur beim ersten Aufruf zur Ausführung.

- **Vorteile**
  Dieser Ansatz ist dadurch, dass er sich einer einfachen Architektur bedient, schnell und unkompliziert umzusetzen. Er kann daher bei einfachen Anwendungen, die im Nachhinein nicht erweitert oder verändert werden müssen, zum Einsatz kommen.

- **Nachteile**
  Die Nachteile, die diese Architektur mit sich bringt, sind vielfältiger Natur. Durch das Fehlen des Controllers besteht eine starke Kopplung und Abhängigkeit der Komponenten im System. Somit zieht eine Änderung in einer Komponente auch Änderungen in den anderen Teilen nach sich. Die Steuerung der Anwendung liegt nicht zentral vor, sondern ist auf mehrere JSPs verteilt, da jede JSP selber „entscheiden“ muss, welche Seite als nächstes zur Anzeige kommen soll. Eine Änderung des Workflows der Anwendung ist somit mit viel Arbeit verbunden, da eben nicht an zentraler Stelle, sondern an mehreren Stellen im System Änderungen vorgenommen werden müssen. Die Wahrscheinlichkeit, an einer oder mehreren Stellen die nötigen Änderungen zu vergessen, ist speziell bei mangelhafter Dokumentation sehr groß. Dies kann zu einer hohen Fehlerquote und somit zu einem erhöhten Wartungsaufwand führen. Die bei der Entwicklung eingesparte Zeit geht hier also bei jeder Erweiterung verloren.

## 4.2 Realisierung über Datenbank-Trigger

Bei diesem Ansatz wird der Page-Caching Mechanismus komplett aus der Anwendung ausgegliedert und von der Datenbank übernommen. Benötigt werden Datenbank-Trigger, die bei Änderung oder Entfernen bestimmter Datensätze über einen Systembefehl die erforderlichen Seiten aus dem Page-Cache entfernen. Jede Tabelle, die Geschäftsobjekte beinhaltet, die in irgendeiner Form am Page-Caching beteiligt sind, muss über solch einen Trigger verfügen. Es ist vorstellbar, dass ein Geschäftsobjekt, oder Teile davon, in mehreren Seiten angezeigt wird. Somit müssen eventuell mehrere Seiten aus dem Page-Cache entfernt werden, nämlich genau die, in denen Teile des Geschäftsobjekts angezeigt werden.

- **Vorteile**
  Durch die Verwaltung des Page-Cache direkt an der Datenbasis besteht nicht die Gefahr, dass andere Anwendungen ohne einen Page-Caching Mechanismus Daten löschen, ohne

dass der Page-Cache auf einen aktuellen Stand gebracht wird. Ein weiterer Vorteil der Verlagerung des Page-Caching in die Datenbank ist, dass es prinzipiell keine Rolle spielt, welche Art Architektur die Anwendung beinhaltet, da das Page-Caching völlig unabhängig davon verwaltet wird.

- **Nachteile**
  Durch die Verwaltung des Page-Cache mit Datenbank-Triggern, wird auch die damit verbundene Rechenlast an den Datenbankserver delegiert. Dies kann bei Hochlastsystemen einen nicht unerheblichen Performanzverlust nach sich ziehen. Ein weiteres Manko an dieser Lösung ist, dass sie nur die Möglichkeit bietet, Seiten aus dem Page-Cache zu entfernen, die auf Geschäftsobjekten basieren. Seiten, deren Inhalt sich nicht aus Daten in einer Datenbank ergeben, können somit nicht mit Page-Caching betrieben werden.
  Um diesen Lösungsansatz zu verfolgen, ist es erforderlich, dass die Anwendungsentwickler entweder Administrationsrechte und -kenntnisse für den Datenbankserver besitzen oder ein Datenbankadministrator die erforderlichen Trigger in die Datenbank einpflegt. Die Aufteilung auf mehrere Entwickler ist also an dieser Stelle nicht umzusetzen oder kann bei personellen Engpässen zu Zeitverlust in der Entwicklungsphase führen.

## 4.3 Servlet / JSP Lösung mit Cache-Manager

Dieser Lösungsansatz basiert auf bekannten Entwurfsmustern wie Model-View Controller und Observer / Observable. Das Model ist in Java Klassen und JavaBeans modelliert, die View besteht aus JSPs und der Controller wird als Servlet realisiert. Ziel bei dieser Architektur ist es, den Page-Caching Mechanismus in Form eines Cache-Managers in die View Komponente zu integrieren.
Bei Webanwendungen ohne Page-Caching stellt sich das Model-View Controller Muster aus 2.6 in leicht veränderter Form dar. Dadurch dass auf die aktuell im Browser angezeigte Seite kein Einfluss genommen werden kann, entfällt die Benachrichtigung der View durch das Model bei Änderungen. Beim erneuten Aufruf der Seite sind die Änderungen dann aber sichtbar, weil diese neu generiert wird. Das MVC-Muster für Webanwendungen ist in Abbildung 8 dargestellt. Der genannte Umstand tritt aber bei der Verwendung von Page-Caching nicht mehr auf. Zwar ist es auch hier nicht möglich, die Seite im Browser direkt zu manipulieren, allerdings kann ein Cache-Manager als Teil der View entwickelt werden, der die Benachrichtigungen durch das Model empfängt und den Page-Cache verwaltet. Wie schon in 2.7 erwähnt, kann die View-Komponente einer MVC-Architektur als Observer und das Model als Observable realisiert werden. Daraus ergibt sich für den Cache-Manager, dass dieser die Benachrichtigung über Änderungen im Model erhält und für die JSPs, dass diese die Informationen über geeignete Methoden aus dem Model herauslesen. Durch diese Entkopplung kann die gesamte Verantwortlichkeit darüber, welche Seiten aus dem Page-Cache zu entfernen sind, im Cache-Manager liegen. Die Auswirkungen auf das MVC-Muster erläutert Abbildung 9.

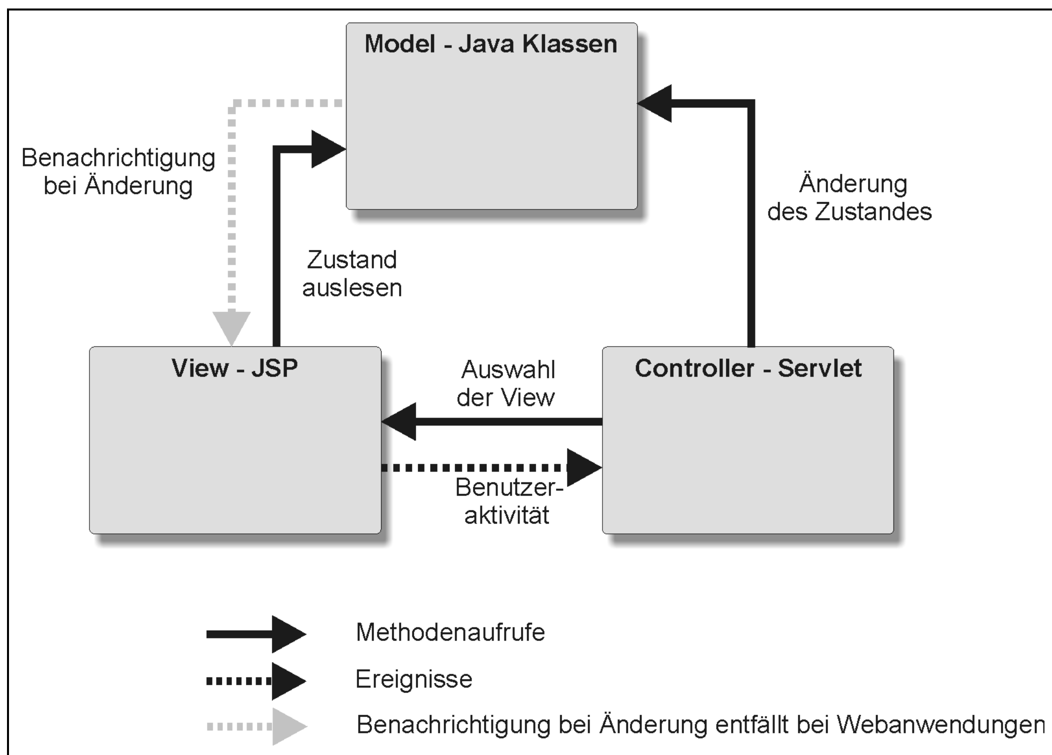


Abb. 8: MVC-Muster bei Webanwendungen

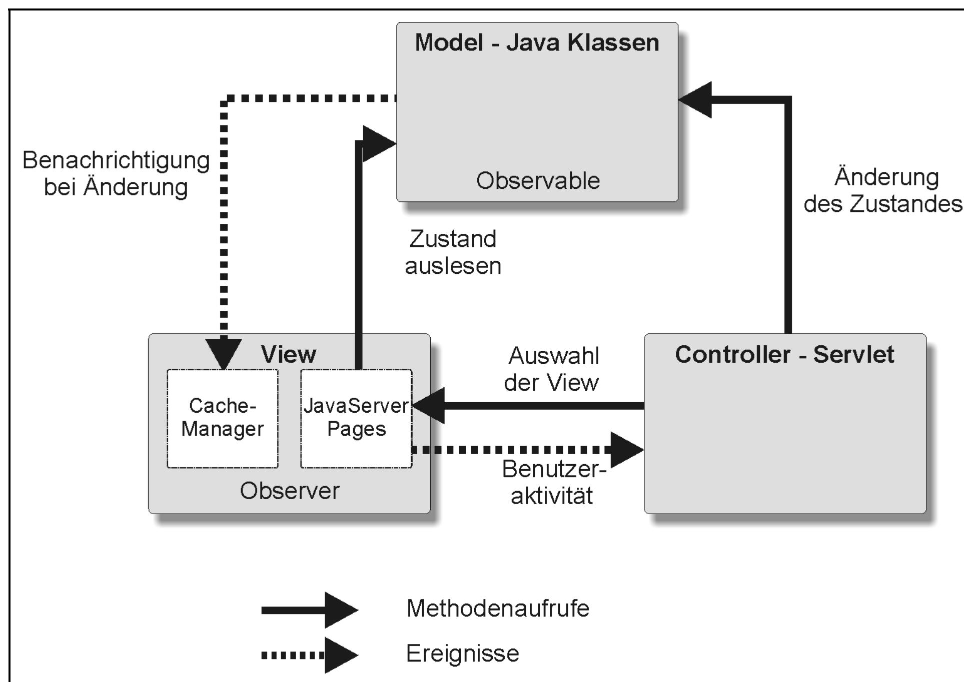


Abb. 9: Verwendung von Observer / Observable im Web

Eine Anwendung, die nach diesen Gesichtspunkten entwickelt wurde, beinhaltet auch die Trennung in Interaktionskomponente und Funktionskomponente, wie in 2.8 erläutert. Wie Abbildung 10 zeigt, wird die Interaktionskomponente durch die View und den Controller, also die JSPs und das Servlet, realisiert, und die Funktionskomponente macht das Model aus.

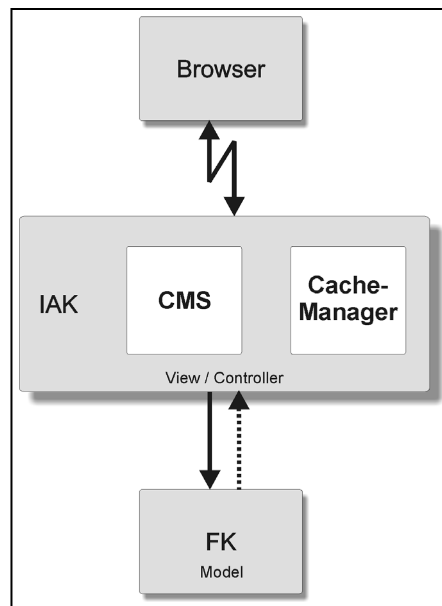


Abb. 10: Aufteilung der Komponenten auf die Interaktions- und Funktionskomponente

Das Verhalten zur Laufzeit stellt sich wie folgt dar. Ändert der Benutzer durch seine Aktivität die Daten, also das Model, erhält die View, genauer der Cache-Manager eine Benachrichtigung darüber, dass sich der Zustand des Models geändert hat. Im Gegensatz zum „üblichen“ Verhalten des MVC-Musters liest der Cache-Manager dann aber nicht den aktuellen Zustand des Models aus, sondern löscht die erforderlichen Seiten aus dem Page-Cache. Beim nächsten Aufruf einer Seite, die auf den geänderten Daten basiert, wird dann das System feststellen, dass im Page-Cache keine statische Version vorhanden ist, die Seite dann mit den aktuellen Daten neu generieren, dem Benutzer anzeigen und für eine erneute Anzeige im Page-Cache abgelegen.

- **Vorteile**
  Bei diesem Vorgehen wird auf bekannte Muster zurückgegriffen. Entwickler müssen also nicht extra angelernt werden, um Anwendungen basierend auf der Architektur umzusetzen. Durch die Trennung von Logik, und Präsentation in Interaktions- und Funktionskomponente sind die einzelnen Komponenten des Systems einfach auszutauschen, ohne das dies große Änderungen in den anderen Teilen nach sich ziehen würde. Außerdem kann die Entwicklungsarbeit sinnvoll auf mehrere Mitarbeiter aufgeteilt werden. Ein HTML-Spezialist mit geringen Java-Kenntnissen kann die JSPs entwickeln, da die einzigen Java-Konstrukte, die in den Seiten vonnöten sind, Iterationen und die *get-Methoden* der JavaBeans sind, die zur Laufzeit an die JSP übergeben werden. Er braucht sich weder um die Steuerung der Anwendung noch um Datenbankzugriffe zu kümmern, die durch andere Entwickler implementiert werden. Der mehrschichtigen Ansatz bringt eine höhere Robustheit der Anwendung durch entkoppelte Komponenten und durch Entwicklung der Komponenten durch Entwickler mit jeweiligen Fachkenntnissen mit sich. Siehe auch Kapitel 2.5.

- **Nachteile**
  Die einzigen Nachteile dieses Ansatzes sind der einmalig anfallende Entwicklungsaufwand des Cache-Managers und eine niedrigere Performanz im Gegensatz zu den Lösungen in 4.1 und 4.2. Der Cache-Manager sollte sehr gewissenhaft umgesetzt werden, um die Anwendung später durch einfache Schritte um wenig Entwicklung erweiterbar zu halten.

### 4.3.1 Spezifika bei der Verwendung von Servlets und JSPs

In gewöhnlichen Servlet / JSP-Anwendungen können Informationen, die in der JSP benötigt werden, über die Session zur Verfügung gestellt werden. Daraus entstehen diverse Komplikationen, die

man bedenken muss, wenn die Anwendung auf einem Web Content Management System laufen soll. Die gezeigten Lösungen beziehen sich auf das bei der Entwicklung der Beispielanwendung verwendeten Vignette Content Management Systems (VCMS).

- **Load-Balancing**
  In einer großen Vignette Anwendung ist es möglich, *Load-Balancing* zu betreiben. Dies bedeutet, dass die Anfragen an das System nicht von einem einzigen Server abgearbeitet werden, sondern das System auf mehrere Rechner aufgeteilt wird.
  Normalerweise würden so die Informationen in der Session verloren gehen, wenn der Request auf einen anderen Rechner weitergeleitet wird, da diesem die Session nicht bekannt ist. Um dies zu umgehen, bietet Vignette einen Workaround an:

  Der Administrator legt eine neue Plattform-Konfigurations-Variable an und passt die Regeln des Load-Balancing-Systems (in Falle von Vignette ist dies *Resonate*) an.
  Der Entwickler muss in den JSP-Templates einen Cookie vorsehen, der JSP-Sessions verwendet.

  (Weitere Informationen im [VOLSS] unter Knowledge Base Eintrag 2611).

- **REGENERATE_PAGE**
  Die Plattform-Variable *REGENERATE_PAGE* erlaubt es dem Administrator festzulegen, ob eine Seite nachdem sie aus dem Cache gelöscht wurde automatisch vom System neu generiert werden soll. Dies kann unter Umständen die Performanz einer Anwendung verbessern, weil die Seite nicht erst in dem Moment neu generiert wird, in dem ein Benutzer diese Aufruft, sondern eine fertig generierte Version schon im Cache vorliegt.
  Werden allerdings Informationen über die JSP-Session zur Generierung der Seite zur Verfügung gestellt, sind diese dem System zum Zeitpunkt der automatischen Generierung der Seite nicht zugänglich. Dies führt dazu, dass die Seite zwar eventuell im Cache abgelegt wird, allerdings die Informationen, die über die Session übergeben werden sollen, in der Seite fehlen.
  Da die Plattform-Variable standardmäßig auf *TRUE* steht, empfiehlt es sich, diese auf *FALSE* umzusetzen.

## 4.4 Diskussion der vorgestellten Lösungsansätze

Jede der in den vorangegangen Kapiteln vorgestellten Architekturen hat wie erläutert Vor- und Nachteile und kann für spezielle Anwendungsfälle die optimale Lösung darstellen. In der folgenden Diskussion soll dennoch durch Abwiegen der Vorzüge der einzelnen Architekturen die Lösung gefunden werden, die vom softwaretechnischen Standpunkt die beste und im allgemeinen Anwendungsfall die am erfolgversprechendste ist.

Eine reine JSP-Lösung hat sicherlich den Vorteil, dass sie in relativ kurzer Zeit entwickelt werden kann. Durch die Tatsache, dass die gesamte Logik verteilt in den JSPs liegt, ist eine spätere Erweiterung beziehungsweise Wartung einer solchen Anwendung relativ zeit- und kostenintensiv. Abgesehen davon birgt dieser Ansatz bei wachsenden Anwendungen die Gefahr eines hohen Fehlerpotentials, da eine große Anwendung basierend auf dieser Technologie unüberschaubar und dadurch nicht mehr sinnvoll wartbar wird. Oft ist die Folge ein Redesign, bei dem die komplette Anwendung basierend auf einer anderen Technologie neu entwickelt werden muss. Ein weiterer Nachteil ist die Tatsache, dass die unterschiedlichen Fachgebiete eines Projekts wie Oberflächengestaltung zum Beispielt mit HTML, Datenbankzugriffe und Umsetzen der Geschäftslogik nur schwer auf mehrere Mitarbeiter aufgeteilt werden können.
Konzeptionell gesehen scheint die Lösung des Problems mithilfe von Datenbank-Triggern ein vielversprechender Ansatz zu sein. Allerdings gilt es hier die personellen Komplikationen nicht außer Acht zu lassen, die sich durch die Tatsache ergeben könnten, dass ein Entwickler, der keine Admi-

nistrationsrechte und -kenntnisse auf dem Datenbankserver hat, immer auf eine weitere Person mit eben diesen Fähigkeiten angewiesen ist. Die Nichtverfügbarkeit einer solchen Person kann dann die Entwicklungsarbeit an einer Anwendung beeinträchtigen oder sogar komplett zum Erliegen bringen. Der augenscheinliche Vorteil, dass auch Anwendungen außerhalb des Content-Management-Kontextes schreibend auf die Daten zugreifen können, sollte dazu führen, dass sich die Entwickler solcher Anwendungen Gedanken darüber machen, ob dies überhaupt gewünscht ist. Denn offensichtlich ergibt es keinen Sinn, solche Anwendungen aus dem Content-Management-Kontext auszuschließen, denn dies würde ein Redaktionssystem in Frage stellen oder obsolet machen.
Die Umsetzung einer Lösung, basierend auf bekannten Entwurfsmustern wie in 4.3 beschrieben, scheint auf den ersten Blick sehr zeitaufwendig und komplex zu sein. Dieser „Nachteil“ lässt sich aber leicht entkräften. Die einmaligen Entwicklungskosten eines solchen Systems rentieren sich direkt bei Erweiterungen und Wartungsarbeiten, da die Verantwortlichkeiten klar voneinander getrennt vorliegen, Mitarbeiter ihren Fachgebieten entsprechend die Arbeiten an dem System vornehmen können und der Cache-Manager als Framework entwickelt werden kann, so dass spätere Erweiterungen ein Eingreifen tief in das System nicht erforderlich machen. Die Lösung als Model-View Controller bringt alle Vorteile mit sich, die eine robuste und flexible Anwendung erfordert und kann für alle Anwendungsfälle eingesetzt werden.

# 5 Architektur der Lösung

Basierend auf der in 3.5 vorgestellten Beispielanwendung soll in diesem Kapitel exemplarisch die Architektur eines Page-Caching Mechanismus mit einem Cache-Manager als Teil der Model-View Controller Entwurfsmusters erläutert werden. Wie in der Einleitung erwähnt, basiert die Anwendung auf dem Vignette Content Management System (VCMS). Dennoch soll die Lösung so unabhängig sein, dass eine Portierung auf ein anderes Systems mit möglichst wenig aufwand zu realisieren ist.

## 5.1 Verteilung der Verantwortlichkeit auf die Teile des Cache-Managers

Die Rolle des Cache-Managers besteht darin, eine Verknüpfung zwischen den im Page-Cache vorhandenen Seiten und den diesen zugrundeliegenden Daten zu schaffen. Das bedeutet, dass der Cache-Manager dafür verantwortlich ist, dass die erforderlichen Seiten aus dem Page-Cache entfernt werden, wenn die dazugehörigen Daten verändert oder gelöscht werden (vgl. [CLLHA01] S. 534). Damit dies zum richtigen Zeitpunkt geschieht, wird der Cache-Manager als Teil der View-Komponente des Model-View Controller vom Model benachrichtigt, wenn sich dieses ändert.
Entscheidend für die Portierbarkeit und Wiederverwendbarkeit des Cache-Managers ist die Art der technischen Umsetzung. Es gilt, allgemeine Methoden in einer für alle Systeme passenden Schnittstelle zu realisieren, während die systemspezifischen Charakteristika in Spezialisierungen abgebildet werden. Wie in 4.3 erläutert, soll der Cache-Manager als *Observer* und die Geschäftsobjekte als *Observable* umgesetzt werden. Für die Geschäftsobjekte folgt daher, dass sie von der Klasse *java.util.Observable* erben. Diese Klasse beinhaltet unter anderem Methoden, um neue *Observer* an- und abzumelden und um die angemeldeten *Observer* zu benachrichtigen, wenn sich der Zustand des *Observable* geändert hat. Der Cache-Manager implementiert das *java.util.Observer* Interface.
Innerhalb des Cache-Managers ist die Verantwortlichkeit auf mehrere Klassen verteilt. Damit das System portierbar und wiederverwendbar bleibt, ist als oberste Klasse eine Schnittstelle eingezogen, die die erforderlichen Methoden für das Cache-Management unabhängig vom zugrundeliegenden System beinhaltet. Diese Schnittstelle wird durch die abstrakte Klasse *CMCacheManager* realisiert. Sie enthält die Methoden `clearCache(String[] paths)`, `generateURL(String path)` und `generateURL(String path, Integer oId)`. Mit der Methode `clearCache(String[] paths)` ist ein für alle Systeme einheitlicher Zugangspunkt zu den systemspezifischen APIs gegeben, die das Löschen der Seiten aus dem Page-Cache ermöglichen. Sollte das verwendete System keine API zur Verfügung stellen, kann in der erbenden Unterklasse direkt der Zugriff auf den Page-Cache realisiert werden. Um die URLs zu den gewünschten Webseiten zu generieren, wurden die Methoden `generateURL(String path)` und `generateURL(String path, Integer oId)` in die Klasse integriert. Je nach System kann in der erbenden Spezialisierung die für das System spezifische Form der URL erzeugt werden. Die erbende Unterklasse *CMSystemCacheManager* im Beispiel *VignetteCacheManager* implementiert die Methoden `clearCache(String[] paths)`, `generateURL(String path)` und `generateURL(String path, Integer oId)`. Der *CMSystemCacheManager* beinhaltet die Zugriffe auf das zugrundeliegende Web Content Management System und dessen API. Das Wissen darüber, welche Seiten tatsächlich bei einer Änderung eines Geschäftsobjekts aus dem Page-Cache entfernt werden müssen, liegt in der von *CMSystemCacheManager* erbenden Klasse *BusinessObjectCacheManager*. In deren Konstruktor werden die Pfade zu allen Templates ermittelt, deren statische Seiten Informationen enthalten, die auf dem zugehörigen Geschäftsobjekt basieren. Die Klasse *BusinessObjectCacheManager* implementiert auch die Methode `update(Observable o, Object arg)` des *Observers*. Als zweites Argument wird der Methode das geänderte oder gelöschte Geschäftsobjekt übergeben. Zusammen mit den Pfaden der Templates werden durch die Objekt-Id (oId) des Geschäftsobjekts die URLs der Seiten bestimmt, die aus dem Page-Cache entfernt werden müssen. Das Klassendiagramm in Abbildung 11 veranschaulicht diese Architektur. Den in Kapitel 4.3 gemachten Erläuterungen und den Abbildungen 9 und 10 zufolge verteilen sich in dieser Architektur die Klassen *Observable* und *BusinessObjectList*

auf das Model und somit auf die Funktionskomponente. Die restlichen Klassen sind Teil der View beziehungsweise der Interaktionskomponente.

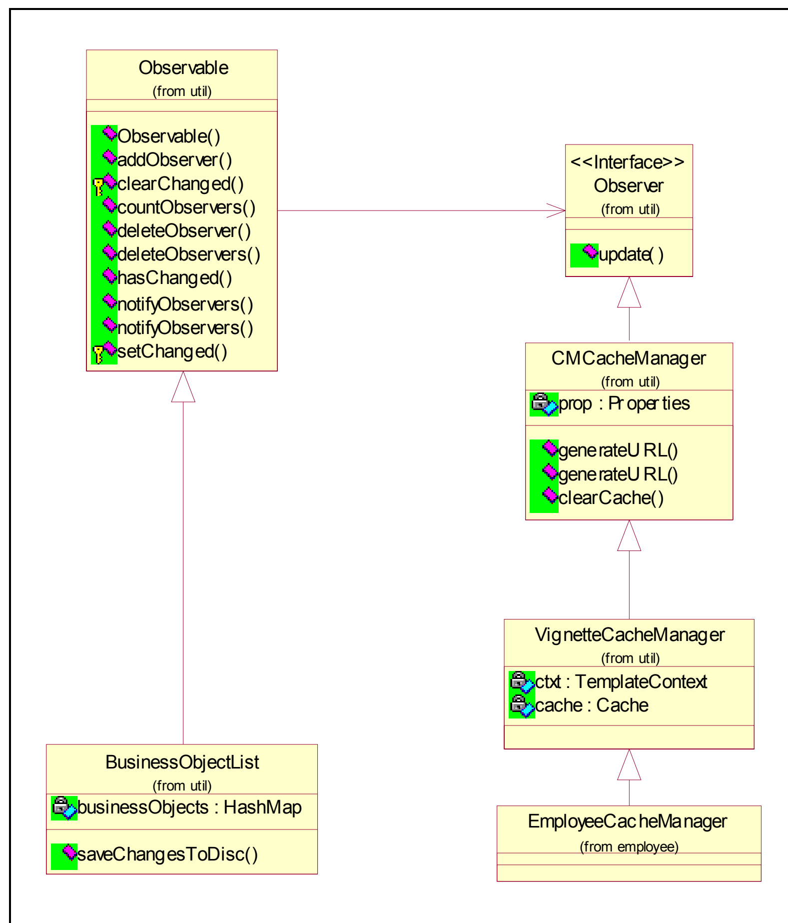


Abb. 11: Umsetzung des Observer / Observable Entwurfmusters

## 5.2 Szenario für das Update des Page-Cache

Nachdem der Benutzer durch Klick auf den Button *Änderungen speichern* in der Auswahlmaske des Redaktionssystems das Speichern der Änderungen veranlasst, ruft das Servlet die Methode `saveChangesToDisc()` der Klasse *BusinessObjectList* auf. Diese iteriert über die enthaltenen Geschäftsobjekte (*BusinessObject*) und schreibt die Änderungen mittels des *DBObjectHandler* in die Datenbank, setzt den Status der Geschäftsobjekte auf „keine Aktion" und ruft die von *Observable* geerbte Methode `notifyObservers(Object arg)` auf. Diese wiederum ruft die Methode `update(Observable o, Object arg)` aller angemeldeten Observer, in diesem Fall des *EmployeeCacheManager*. In dieser Methode werden zunächst über die von *CMSystemCacheManager* – hier *VignetteCacheManager* – Methode `generateURL(String path, Integer oId)` die URLs der Seiten bestimmt, die aus dem Page-Cache entfernt werden müssen. Die notwendigen Parameter sind die Pfade der Templates auf denen die Seiten basieren und die Objekt-Id (oId) des Geschäftsobjekts. Eine Ausnahme bilden hier die Auswahlmasken der Beispielanwendung. Da diese auf keinem Geschäftsobjekt basieren, kann die URL nur aus dem Pfad gewonnen werden. Dies geschieht mittels der Methode `generateURL(String path)`. Die geerbte Methode `clearCache(String[] paths)` entfernt dann die den übergebenen Pfaden entsprechenden Seiten aus dem Page-Cache. Im vorliegenden Beispiel liefert Vignette die APIs zum Generieren von Vignette-spezifischen URLs, den sogenannten Custom URLs (CURLs) und zum Löschen von Seiten aus dem Page-Cache mit. Das bedeutet, dass die Methoden `generateURL(String path)` und `generateURL(String path, Integer oId)` auf die

Vignette-Klasse *TemplateContext* und die Methode `clearCache(String[] paths)` auf die Vignette-Klasse *Cache* zugreifen. Das geschilderte Verhalten wird in Abbildung 12 verdeutlicht.

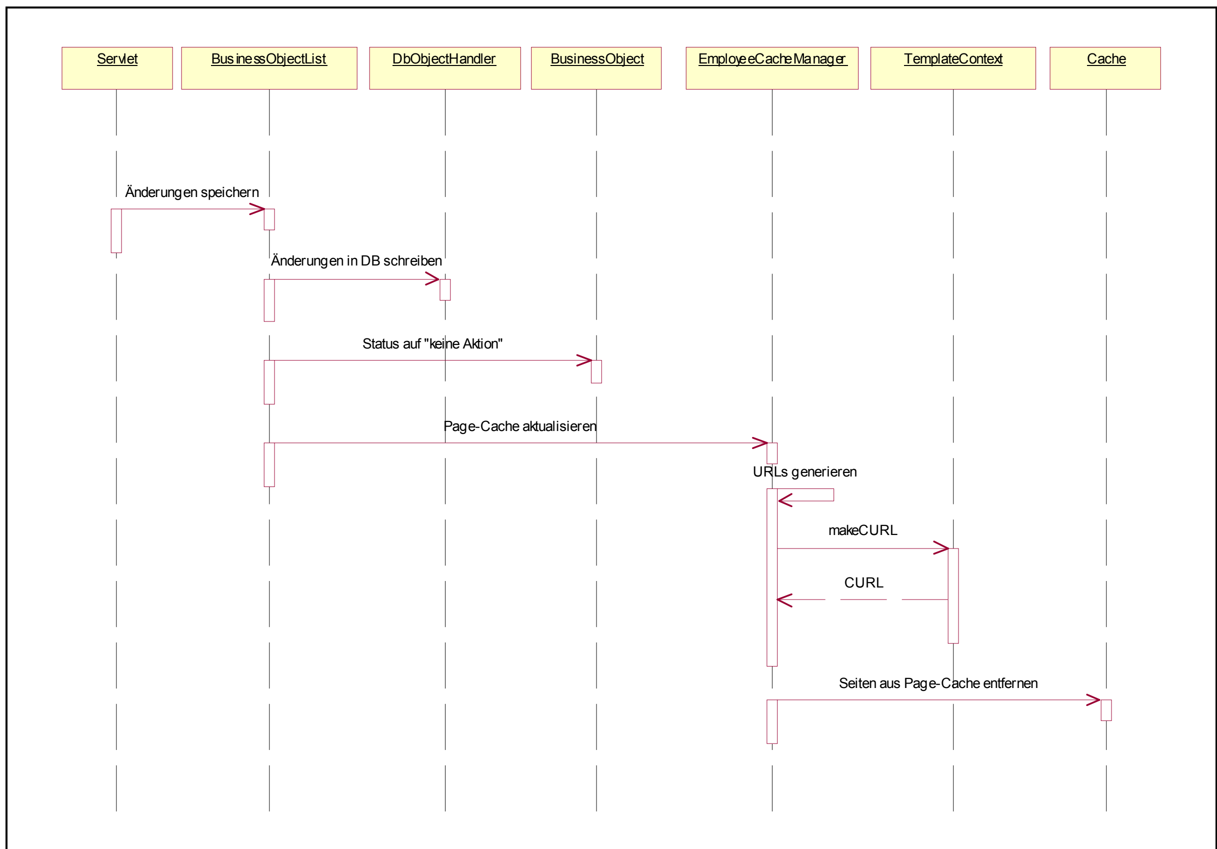


Abb. 12: Sequenzdiagramm für das Aktualisieren des Page-Cache

## 5.3 Implementationsdetails

Die Folgenden Code-Beispiele sollen das Konzept des Cache-Managers weiter verdeutlichen, in dem die Aufteilung des Codes auf die einzelnen Klassen dargestellt wird. Nicht aufgeführt sind die Klasse *Observable* und das Interface *Observer*, die Bestandteil des java.util Packages sind und die Klassen *TemplateContext* und *Cache*, die Teile der Vignette-API sind.

**BusinessObjectList**

Die Methode `saveChangesToDisc()` bildet den Einstiegspunkt für das Management des Page-Cache in der Beispielanwendung. Sie iteriert über die in der *BusinessObjectList* zur Laufzeit enthaltenen Mitarbeiter. Bei jedem einzelnen wird der Status ausgewertet und entschieden, wie sich dieser auf den Page-Cache auswirkt. Bis auf den Fall, dass der Datensatz des Mitarbeiters nicht editiert wurde, führen alle Statuus zur Aktualisierung des Page-Cache.

```
package de.resco.util;

public class BusinessObjectList extends Observable
{
   private int DEFAULT = 0 ;
   private int NEW = 1 ;
   private int UPDATE = 2 ;
   private int DELETE = 3 ;

   public BusinessObjectList()
```

```
    {
    }

          .
          .
          .

    public void saveChangesToDisc() throws SQLException
    {
          BusinessObject bo;
          Integer key;
          for(Iterator i = sortedKeyVector.iterator();
          i.hasNext();)

          {
                key = (Integer)i.next();
                bo = (BusinessObject)hashMap.get(key);
                int status = bo.getStatus();
                switch(status){
                      case NEW:
                            //bo is new => insert into db
                            dbObjectHandler.insertBo(bo);
                            bo.setStatus(DEFAULT);
                            notifyObservers(bo);
                            break;

                      case UPDATE:
                            //bo is updated => update in db
                            dbObjectHandler.updateBo(bo);
                            bo.setStatus(DEFAULT);
                            notifyObservers(bo);
                            break;

                      case DELETE:
                            //bo is deleted => delete from db
                            dbObjectHandler.deleteBo(bo);
                            hashMap.remove(bo.getKey());
                            notifyObservers(bo);
                            break;

                      default:
                            //bo is unchanged => do nothing
                }
          }
          sortKeyVector();
    }

          .
          .
          .
}
```

**CMCacheManager**
Diese abstrakte Klasse liefert die Schnittstelle, die alle Spezialisierung implementieren müssen. Da die Implementierung abhängig vom verwendeten Web Content Management System ist, muss für jedes System eine eigene Spezialisierung geschaffen werden.

```
package de.resco.util;

public abstract class CMCacheManager implements Observer
{
   public static Properties prop;

   public CMCacheManager(Properties prop)
   {
        this.prop = prop;
   }
   public abstract String generateURL(String path);

   public abstract String generateURL(String path, Integer oId);

   public abstract void clearCache(String[] paths);
}
```

**VignetteCacheManager**
Der *VignetteCacheManager* dient dazu, die CURLs, eine von Vignette verwendete Sonderform der URL zu generieren und Seiten aus dem Page-Cache zu entfernen. Sie enthält über die verwendeten Klassen *Cache* und *TemplateContext* die einzigen Zugriffe auf die von Vignette mitgelieferte API.

```
package de.resco.util;

import com.vignette.cds.client.beans.Cache;
import com.vignette.cds.client.beans.TemplateContext;


public abstract class VignetteCacheManager extends CMCacheManager
{
   public Cache cache = new Cache();
   public TemplateContext ctxt;

   public VignetteCacheManager(Properties prop)
   {
        super(prop);
        try
        {
             ctxt = new TemplateContext();
        }
        catch(Exception e)
        {
             System.out.println("VignetteCacheManager: " + e);

        }
   }

   public String generateURL(String path)
   {
        String url = "";
        try
```

```
        {
            url = ctxt.makeCURL(path);
        }
        catch(Exception e)
        {
            System.out.println("VignetteCacheManager: " + e);

        }

        return url;
    }

    public String generateURL(String path, Integer oId)
    {
        String url = "";
        try
        {
            url = ctxt.makeCURL(path, oId.toString(), "");
        }
        catch(Exception e)
        {
            System.out.println("VignetteCacheManager: " + e);
        }

        return url;
    }

    public void clearCache(String[] paths)
    {
        try
        {
            if (paths.size() != 0)
            {
                Enumeration cacheManagers =
                cache.getLocalCacheManagers();

                cache.clearDocumentCache(paths, cacheManagers);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

**EmployeeCacheManager**
Der *EmployeeCacheManager* generiert aus dem von der *BusinessObjectList* übergebenen Vector eine *ArrayList*, mit der als Array die Methode `clearCache(String[] paths)` des *VignetteCacheManager* aufgerufen wird.

```
package de.resco.employee;

public class EmployeeCacheManager extends VignetteCacheManager
{
   String[] curls ;

   public EmployeeCacheManager(Properties prop)
   {
        super(prop);
        String path1 = prop.getProperty("EMPLOYEE_LIST_URL");
        String path2 = prop.getProperty("EDIT_EMPLOYEE_URL");
        curls = new String[2];
        curls[0] = generateURL(path1)
   }

   public void update(Observable o, Object arg)
   {
        System.out.println("<Observer-update>");
        try
        {
             curls[1] = generateURL(path2);
             clearCache(curls);
        }
        catch(Exception ex){
             System.out.println("Exception: " + ex);
        }
   }
}
```

## 5.4 Erforderliche Schritte bei Portierung und Weiterentwicklung

Dieses Kapitel versucht den Aufwand abzuschätzen, der geleistet werden muss, wenn man eine Anwendung erweitert oder auf ein anderes Web Content Management System portiert.

- **Portierung**
  Bei einer Portierung ist es erforderlich den Cache-Manager an das neue System anzupassen. Daher muss die Klasse *VignetteCacheManager* durch eine entsprechende Klasse *XxxCacheManager* ersetzt werden. Diese muss die für das neue System erforderlichen Schritte zur Verwaltung des Page-Cache enthalten. Die Unterschiede liegen hier in den Methoden `generateURL(String path)`, `generateURL(String path, Integer oId)` und `clearCache(String[] paths)`.

- **Weiterentwicklung**
  Bei einer Weiterentwicklung einer Anwendung oder bei einer Neuerstellung muss für jedes Geschäftsobjekt eine Klasse *XxxCacheManager* geschaffen werden, die das Wissen vorhält, welche Seiten aus dem Page-Cache gelöscht werden müssen, wenn sich das zugehörige Geschäftsobjekt ändert oder es gelöscht wird. Werden neue JSPs in das System eingepflegt, die Daten bestehender Geschäftsobjekte beinhalten, müssen in der jeweiligen Klasse *XxxCacheManager* diese JSPs berücksichtigt werden, dass auch die auf diesen Templates basierenden statischen Seiten aus dem Page-Cache entfernt werden können.

## 5.5 Spezifika beim Vignette Content Management System (VCMS)

In einer MVC-Architektur ist es erforderlich, einmalig eine Vignette Initialisierung aufzurufen. Diese ist Bestandteil der vignetteeigenen API und kann entweder im Servlet oder in einer JSP geschehen. Der folgende Befehl führt die Initialisierung durch:

```
VgnInit.getInstance().initialize(install-dir, install-bin-path,
config-id, cds-cfgfile-path)
```

(Siehe auch [VgnRes1]).

Da die Vignette-Initialisierung einmalig geschehen muss, bietet sich bei der Verwendung eines Servlets dessen `init`-Methode an, um die Vignette-Initialisierung aufzurufen, da nur beim ersten Aufruf - und somit ebenfalls nur einmalig - ausgeführt wird.

# 6 Resümee

Die gezeigten Design-Entscheidungen erläutern am Beispiel Vignette, wie in einem Web Content Management System eine Anwendung entwickelt werden kann, die einerseits den Page-Caching-Mechanismus des Systems ausnutzt und andererseits guten und bekannten Design-Richtlinien folgt. Durch die Verwendung des MVC-Entwurfsmusters und die Abbildung der Page-Caching-Strategie auf das bekannte *Observer / Observable*-Entwurfsmuster wurden neue Probleme mit bekannten Methoden gelöst. Durch die Entkopplung der einzelnen Komponenten können diese beliebig ausgetauscht werden. Außerdem ist so eine Aufteilung der Entwicklungsarbeit auf mehrere Mitarbeiter mit entsprechenden Fachkenntnissen möglich. Ein erfahrener Java-Entwickler kann mit minimalem Aufwand in den Web Content Management-Kontext eingearbeitet werden. Durch die gezeigte Aufteilung des Codes auf die verschiedenen Klassen und Subklassen ist ein System ohne großen Aufwand erweiterbar. Auch einer Wiederverwendung der Cache-Manager Klassen steht nichts im Wege. Sogar eine Portierung auf ein anderes System ist möglich. Dazu muss lediglich die Klasse *VignetteCacheManager* durch eine an das neue System angepasste Klasse *XxxCacheManager* ersetzt werden. Durch die angesprochene Verteilung des Codes geschieht dies aber mit geringem Aufwand, da nur die systemspezifischen Methoden auf das neue System angepasst werden müssen. Die allgemeingültigen Methoden bleiben in allen Systemen bestehen und müssen daher nicht neu implementiert werden.

Die nachträgliche Integration eines Cache-Managers in eine bestehende Anwendung ist auch ebenfalls mit relativ geringem Aufwand realisierbar, da nur wenige bestehende Klassen umcodiert werden müssen. Zum einen müssen die Observer beim Observable angemeldet werden, und die Geschäftsobjekte müssen von Observable erben. Zu beachten ist, dass sich jedoch nichts am bestehenden Code ändert, sondern lediglich kurze Teile hinzukommen.

# 7 Literaturverzeichnis


[BTZZ01] Web Content Management; Büchner, Traub, Zahradka, Zschau; Galileo Press; 2001

[CID98] A Scalable System For Consistently Caching Dynamic Web Data; Jim Challenger, Arun Iyengar, Paul Dantzig; 1998

[CLLHA01] Enabling dynamic content caching for database-driven web sites; K. Selcuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, Divyakant Agrawal; ACM Press; 2001

[cmsWatchRes1] cmsWatch; www.cmsWatch.com/GlossaryTerm/?term_id=7; zuletzt besucht: 16.9.2002

[cmsWatchRes2] cmsWatch; www.cmsWatch.com/GlossaryTerm/?term_id=22; zuletzt besucht: 16.9.2002

[DC99] Teaching Object-Oriented Development with Emphasis on Pattern Application; Lew Della, David Clark; ACM Press; 2000

[DHM01] Web Egineering: Beyond CS, IS and SE Evolutionary and Non-engineering Perspectives; aus WebEngineering 2000; Yogesh Deshpande, San Murugesan, Steve Hansen; SS. 14-23; Springer-Verlag Berlin Heidelberg; 2001

[GHJV96] Entwurfsmuster; Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Addison-Wesley; 1996

[Hal01] Core Servlets und JavaServer Pages; Marty Hall; Markt+Technik;2001

[HC01] Java servlet programming; Jason Hunter, William Crawford; O´Reilly; 2001

[IC97] Improving Web Server Performance By Caching Dynamic Data; Arun Iyengar, Jim Challenger; 1998

[KK01] Layout, Content and Logic Seperation in Web Engineering; aus WebEngineering 2000; Clemens Kerer, Engin Kirda; Springer-Verlag Berlin Heidelberg; 2001

[KP88] A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming,* 1(3):26-29; Glenn E.Krasner, Stephen T. Pope; August/September 1988

[MDG01] Web Engineering: A New Discipline for Development of Web-Based Systems; aus WebEngineering 2000; San Murugesan; Yogesh Deshpande, Steve Hansen, Athula Ginige; SS. 3-13; Springer-Verlag Berlin Heidelberg; 2001

[Per01] Managing the Content Explosion into Content-Rich Applications; Internet Computing Strategies, Report Vol. 6, No. 2, May 2001;Robert Perry; 2001

[Rie00] Personalized Views of Personalization; Doug Riecken; Communications of the ACM; August 2000/Vol.43, No.8; SS. 47

[SunRec1] Sun Microsystems; http://java.sun.com/products/jsp; zuletzt besucht: 16.9.2002

[Tur00] Java Server Pages: dynamische Gestaltung von Web-Dokumenten; Volker Turau, Ronald Pfeiffer; dpunkt.verlag; 2000

[VgnRes1] Vignette Content Management V6 Servlet Support; http://support.vignette.com/VOLSS/Resources/View/Addenda/1,2001,3300,00.pdf?rd=1; zuletzt besucht: 1.10.2002

[VgnRes2] Caching Strategies and Caching FAQs, Performance White Paper, Version 1.0, 15. Juni 2001; http://global.vignette.com/download/PERF_CachingStrategiesAndFAQs.pdf; zuletzt besucht: 1.10.2002

[VOLSS] Vignette Online Support System; http://support.vignette.com; zuletzt besucht: 1.10.2002

[Wan99] A survey of web caching schemes for the Internet; Jia Wang; 1999

[Zül98] Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz; Heinz Züllighoven; dpunkt.verlag; 1998