Hochschule für
Technik und Wirtschaft
Dresden (FH)
University of Applied Sciences

Fachbereich Informatik/Mathematik

**DIPLOMARBEIT**

im Studiengang Medieninformatik

**Thema:**

# Entwicklung einer „Rich Internet Application" mit Flex unter Verwendung von Architektur-Frameworks


| | |
|---|---|
| eingereicht von | Mirko Kunath |
| Matrikelnummer | 17452 |
| eingereicht am | 07.04.2010 |
| Betreuer | Prof. Dr. Teresa Merino |

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listingverzeichnis

# 0 Einleitung

Die Anforderungen an Internetapplikationen sind in den letzten Jahren enorm gestiegen: Sie müssen Daten in Echtzeit zwischen tausenden Nutzern vermitteln und täglich verbessert werden um im Markt mithalten zu können. Um diese Anforderungen zu bewältigen werden RIA-Technologien wie Flex eingesetzt und mit bekannten Konzepten aus der Software Architektur auf neuartige Weise kombiniert. Erst eine optimale Wartbarkeit und Wiederverwendbarkeit von Komponenten ermöglicht es einem Webentwickler, neue RIA-Features schnell zu erstellen. Diese Diplomarbeit zeigt wie derartige Anforderungen umgesetzt werden können.

## 0.1 Motivation

Die Anzahl der Internetnutzer stieg in den letzten Jahren stetig an. Nutzten 1997 lediglich 6,5% der Deutschen das Internet gelegentlich, waren es im Jahr 2009 bereits 67,1% [EF09]. Die Abbildung 0.1 verdeutlicht diesen immensen Anstieg innerhalb der letzten 12 Jahre. Gerade die Altersgruppe zwischen 14 und 39 Jahren ist heute nahezu vollständig im Internet aktiv (ca. 90%). Bei den 40- bis 49-Jährigen ist das Online-Verhalten ähnlich hoch (ca.82%) und auch die so genanten „Silver Surfer“ (50-59Jahre) nutzen das Internet inzwischen ausgiebig (ca. 67%). Sogar bei den über 60-Jährigen „surft“ über ein Viertel (ca. 29%) im Internet [Eli09].

Betrachtet man diese Zahlen, so verwundert es nicht, dass das Internet in der heutigen Gesellschaft einen großen Stellenwert einnimmt. Die ständig wachsende Online-Vernetzung führte in den letzten Jahren zu einer enormen Veränderung des Lebens- und Arbeitsumfeldes. Das Internet ist daher heute zu einem allgegenwärtigen Instrument für die Geschäftswelt, das Bildungs- und Sozialwesen sowie für den täglichen sozialen Kontakt zwischen Menschen geworden. Auch die Politik hat das Internet für sich entdeckt. Im US-Amerikanischen Wahljahr 2008 erhielt Barack Obama allein über das Internet Spenden in Höhe von ca. 500 Millionen US-Dollar. Das verdeutlicht eindrucksvoll, wie etabliert

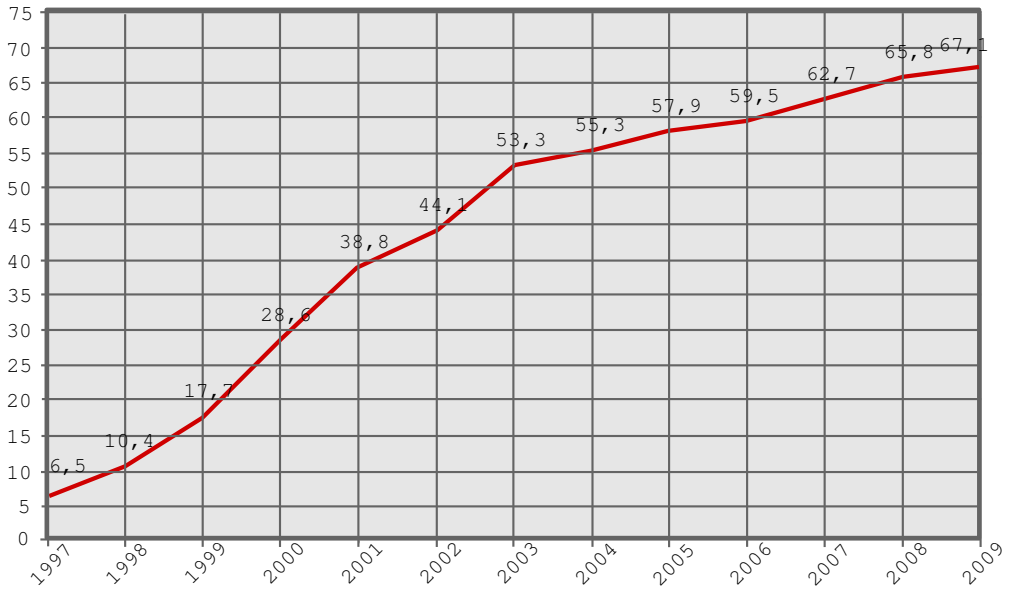


Abbildung 0.1: Anstieg der Internetnutzer 1997 bis 2009
(in Anlehnung an Birgit Eimeren und Beate Frees [EF09])

das Internet in unserer heutigen Gesellschaft ist. Ein Leben ohne Internet ist für einen hohen Bevölkerungsanteil heute nicht mehr vorstellbar.

Durch die hohe Involvierung des Internets in das tägliche Leben, hat sich bei vielen Nutzern ein hoher Anspruch an Dienste und Webseiten herausgebildet. In der Anfangzeit des Internets stand der reine Informationsaustausch im Vordergrund. Wollen heutige Webseiten erfolgreich sein, reicht der Informationsgehalt allein jedoch nicht mehr aus. Web-Design und Nutzungsqualität (Usability) werden bei heutigen Webseiten vom Nutzer als selbstverständlich vorausgesetzt. In den letzten Jahren hat sich besonders der Leistungsumfang der Webseiten stark erhöht. So entstehen heute mehr und mehr Internet-Anwendungen mit komplexen Angeboten wie Video- und Audio-Streaming und Echtzeit-Kommunikation. Gerade die in jüngerer Vergangenheit massenhaft entstandenen Internet-Plattformen wie zum Beispiel „YouTube“, „studiVZ“ oder „Last.fm“ zeigen, wie komplex Internetpräsenzen heutzutage sein können.

Diese Entwicklung von einfachen Webseiten hin zu hochkomplexen Internetanwendungen wird dabei nicht nur durch den Einsatz innovativer Technologien sondern auch durch leistungsfähige Internetanschlüsse ermöglicht. Die Bundesregierung beschloss bei der Kabinettklausur im November 2009 das „schnelle Internet“ für alle. „Bereits bis Ende 2010 sollen in Stadt und Land leistungsfähige Breitbandanschlüsse ($>$ 1Mbit/s)

verfügbar sein“ [Bun09]. Darüber hinaus sollen bis 2014 drei Viertel der Bevölkerung mit Hochleistungsnetzen (> 50Mbit/s) ausgestattet sein. Dies ermöglicht zukünftig noch reichhaltigere Internet- Anwendungen, welche dem Nutzer große Datenmengen zur Verfügung stellen können. Diese hoch komplexen Anwendungen bleiben oft mehrere Jahre Online und müssen ständig verbessert und angepasst werden. So sind neben leistungsfähigen Internetanschlüssen und neuen Technologien besonders solide Software-Architekturen eine wichtige Vorraussetzung bei der Entwicklung von Internet-Anwendungen.

## 0.2 Zielsetzung

Viele Web-Entwickler wollen qualitativ hochwertige Web-Anwendungen erstellen und dabei Entwicklungs- bzw. Wartungszeiten minimieren. Dadurch sollen Kosten eingespart werden. Seit einigen Jahren werden dazu objektorientierte Technologien wie Flex eingesetzt. Dadurch ist es heutzutage möglich, Software-Architektur Konzepte aus dem Desktop-Bereich in den Web-Bereich zu übertragen und anzupassen. Eine gute Software-Architektur führt in der Regel zu flexiblen und soliden Web-Anwendungen, die die Basis für ein konkurrenzfähiges Web-Unternehmen bilden.

Ziel des theoretischen Teils dieser Diplomarbeit ist es, einen effizienten Umgang mit Software-Architektur in Verbindung mit der Web-Technologie Flex aufzuzeigen. Dabei wird auf Software-Architektur und Flex im Allgemeinen eingegangen. Im Fokus steht jedoch die Verwendung vorgegebener Software-Architekturen in Form so genannter Architektur-Frameworks. Besonderer Wert wird dabei auf den Zusammenhang zwischen allgemeiner Software-Architektur und Architektur-Frameworks gelegt. Ein wesentlicher Teil dieser Diplomarbeit fokussiert sich auf die Erläuterung drei ausgewählter Architektur-Frameworks. Dabei werden die Vor- und Nachteile sowie die daraus resultierenden Einsatzgebiete dieser Architektur-Frameworks ausgearbeitet.

Ziel des praktischen Teils ist es, möglichst wiederverwendbare und frei kombinierbare Einzelprogramme (Module) in einer Web-Anwendung zusammen zu stellen. Diese Teile sowie die Web-Anwendung selbst werden mit dem im theoretischen Teil ausgewählten Architektur-Framework umgesetzt. Dabei werden besonders die Stärken dieses Architektur-Frameworks in Verbindung mit der Technologie Flex genutzt, um den Prototypen eines Modulsystems sowie weitere beispielhafte Module zu erstellen.

In der Schlussbetrachtung werden die theoretischen Erkenntnisse und die praktischen

Erfahrungen verglichen.

## 0.3 Aufbau der Diplomarbeit

Kapitel 1 beschäftigt sich mit der Technologie Flex. In diesem Kapitel wird zunächst der Begriff „Rich Internet Application“ erläutert und anschließend Flex in die Web-Programmierung eingeordnet. Dabei wird auf die historische Entwicklung, den grundlegenden Aufbau und auf effizientes Arbeiten mit dieser Technologie eingegangen. Damit Flex bestmöglich in die Web-Programmierung eingeordnet werden kann, werden außerdem weitere Technologien zum Erstellen hochinteraktiver Web-Anwendungen vorgestellt.

Kapitel 2 fokussiert die Thematiken Software-Architektur, Software-Muster und Architektur-Frameworks. Dabei wird einleitend erläutert, warum bewusst gewählte Software- Architekturen für den Prozess der Softwareentwicklung von Bedeutung sind. Anschließend wird gezeigt werden, was Software-Architektur beinhaltet und welche Prinzipien zur Umsetzung von Software-Architektur-Zielen eingesetzt werden können. Abschließend wird der Zusammenhang zwischen Software-Architektur, Software-Muster und Architektur-Frameworks hergestellt.

Nachdem die Technologie Flex und auch die Thematik Software-Architektur bzw. Architektur-Frameworks eingehend erläutert wurden, führt Kapitel 3 diese Thematiken zusammen. Dazu werden drei ausgewählte Flex-Architektur-Frameworks vorgestellt und deren Funktionsweise beschrieben. Hierfür wird eingangs für jedes Architektur-Framework eine allgemeine Aussage erarbeitet, im Anschluss folgt die Beschreibung der Bestandteile jedes Architektur-Frameworks. Für Entwickler, welche die vorgestellten Architektur-Frameworks nicht nur beurteilen, sondern auch einsetzen wollen, wurde im Anhang der Diplomarbeit für jedes Architektur-Frameworks eine detaillierte Erläuterung an Hand eines bestehenden Programms (Café Townsend) erstellt.

Nachdem Aufbau und Funktionsweise der ausgewählten Architektur-Frameworks vorgestellt wurde, wird in Kapitel 4 eine Bewertung dieser Frameworks vorgenommen. Dabei werden jeweils die Kriterien Zugänglichkeit, Entwicklungseffizienz und Wartbarkeit betrachtet und bewertet. Am Ende dieses Kapitel werden die Bewertungen zusammengefasst und Aussagen über Stärken und Schwächen dieser Architektur-Frameworks aufgezeigt. Auf Basis dieser Betrachtungen wird ein Architektur-Framework für den Praxisteil ausgewählt. Kapitel 5 beschreibt die praktische Umsetzung einer „Rich Internet Application“,

welche wiederverwertbare und frei kombinierbare Module enthält. Nach Einführung in die praktische Entwicklung wird der grundlegende Aufbau der entstandenen Anwendung erläutert. Danach wird die Beispielanwendung, in welche die Module inkludiert werden, beschrieben und Konzepte zum Umgang mit diesen Modulen erläutert. Abschließend wird auf die innere Funktionsweise der Module am Beispiel eines ausgewählten Moduls eingegangen. In Kapitel 6 werden abschließend die theoretischen Betrachtungen an den praktischen Erfahrungen gemessen. Dabei wird geprüft, ob sich die theoretischen Annahmen in der Praxis bestätigt haben. Außerdem wird ein Ausblick auf die zukünftige Entwicklung der Software-Architektur im Bereich der Flex- bzw. der Web-Entwicklung gegeben.

# 1 Flex - eine „Rich Internet Application"-Technologie

Dieses Kapital ordnet die in dieser Diplomarbeit thematisierte Technologie Flex in die Webprogrammierung ein und erläutert die grundlegenden Bestandteile von Flex. Dazu wird als erstes der Begriff der „Rich Internet Application" (RIA) eingeführt und die Unterschiede zu herkömmlichen Webanwendungen aufgezeigt. Um die Technologie Flex einordnen zu können, werden anschließend andere RIA-Technologien vorgestellt und die Flex Evolution aufgezeigt. Die grundlegende Flex-Funktionsweise wird im Abschnitt 1.4 vorgestellt und abschließend Ansätze für effizientes Arbeiten mit der Flex Technologie dargestellt.

## 1.1 RIA

Das „World Wide Web" (WWW) wurde ursprünglich für den einfachen Dokumenten- und Informationsaustausch entwickelt. Man erkannte aber schnell die Leistungsfähigkeit von Webservern und -browsern auf Client-Seite und so entstand in den 90er Jahren eine Vielzahl von Webapplikationen auf der Basis der textbasierten Auszeichnungssprache „Hypertext Markup Language" (HTML). Auf Grund der steigenden Ansprüche an Webseiten wurde das Potential von HTML durch clientseitige Sprachen wie Javascript erweitert, wodurch es erstmalig möglich war, auf Benutzereingaben direkt am Client zu reagieren. Serverseitig wurden Sprachen wie zum Beispiel PHP („Hypertext Preprocessor") eingeführt, die es ermöglichen, dynamisch Webseiten zu erstellen und auf Datenbanken zuzugreifen. In den letzten zehn Jahren entstanden so unzählige und vielfältige Webangebote vom Email- Programm bis hin zum Online-Shop, welche in der Literatur oft als „klassische" Webanwendung bezeichnet werden [Rüt09]. Die Vorteile dieser Webanwendungen sind neben der Plattformunabhängigkeit die Online-Erreichbarkeit und das Fehlen

von Installation und Updates. Nach dieser ersten Welle der Webanwendungen ist zurzeit die zweite Welle, in Form der RIA, zu beobachten [NH05]. Diese zeichnen sich ebenso wie klassische Webanwendungen durch Plattformunabhängigkeit, Online-Erreichbarkeit sowie Fehlen von Installation und Updates aus. Die wichtigsten Neuerungen gegenüber klassischen Webanwendungen sind die hohe Interaktivität der Benutzeroberfläche und die Art der Kommunikation zwischen Webserver und Client. Mittels RIA ist es nun möglich, auf komplexe Benutzerinteraktionen im Webbrowser mit dem Nachladen einzelner Seitenteile zu reagieren und nicht wie bei herkömmlichen Seiten durch das Neuladen des gesamten Inhalts. RIA versuchen somit die Stärken von Desktopanwendungen und klassischen Webanwendungen zu kombinieren, um so eine Basis für effiziente Internetanwendungen zu schaffen. Um dies zu erreichen, gibt es verschiedene technologische Ansätze, die wiederum dazu beigetragen haben, dass der Begriff RIA unterschiedlich definiert wird. Grundsätzlich kann man sagen, dass RIAs hochinteraktive Webanwendungen sind, die ein desktopähnliches Arbeitsgefühl erzeugen, Anwendungslogik auf den Client verlagern und meist in einer Browser-Laufzeitumgebung ausgeführt werden. Dieser Ansatz ist nicht neu, jedoch ist es erst heute, dank ausreichender Bandbreite des Internetanschlusses und der Leistungsfähigkeit der Hardware in Verbindung mit neuartigen Technologien im Bereich „Grafical User Interface“ (GUI- deutsch: Grafische Benutzeroberfläche) möglich, diesen Ansatz umzusetzen.

## 1.2 RIA-Technologien

Im Zusammenhang mit RIA werden neben Flex oft drei Technologien genannt: AJAX, Silverlight und JavaFX. Im nachfolgenden werden die einzelnen Technologien kurz vorgestellt, wobei der Hauptfokus dieser Arbeit auf Flex liegt. Die Einführung in die Technologien AJAX, Silverlight und JavaFX erscheint aber wichtig, um die Position von Flex im RIA-Umfeld zu verdeutlichen. Der Vollständigkeit halber seien die Technologien Java-Applets, OpenLaszlo und ActiveX genannt. Diese werden aber nicht weiter ausgeführt, da sie derzeit weniger verbreitet sind.

### 1.2.1 AJAX

„Asynchronous JavaScript and XML“ (AJAX) schien lange die Lösung für RIA- Anforderungen zu sein, da es mittels asynchroner Datenübertragung möglich ist, Daten vom

Webserver anzufordern ohne die gesamte Seite neu laden zu müssen. Grundsätzlich ist AJAX aber nur eine konsequente Umsetzung bekannter Technologien wie (X)HTML, Javascript und XMLHttpRequest. Das Problem besteht vor allem in Javascript, welches zwar von nahezu jedem Browser angeboten, aber nicht von jeden Browser gleich interpretiert wird. Obwohl AJAX recht leistungsfähig in der Benutzeroberfläche ist, sogar Drag & Drop kann dank verschiedenen Bibliotheken und Frameworks eingebunden werden, ist der Entwicklungsaufwand auf Grund der Browserunterschiede verglichen mit den nachfolgenden RIA-Technologien recht hoch [Rüt09, Wid08].

### 1.2.2 JavaFX

Mit JavaFX[1] erweiterte die Firma Sun Microsystems im Dezember 2008 ihre Produktpalette um einen weiteren Meilenstein. JavaFX soll insbesondere lokale Computer mit Netzwerken und mobilen Endgeräten verschmelzen und die Verwendung von User Interfaces, Java2D und Multimediatechniken erleichtern. Da JavaFX auf der „Java Runtime Environment“ (JRE) basiert, ist es möglich, die meisten Javabibliotheken in einfacher Weise zu nutzen und die so implementierten Anwendungen aufgrund der Plattformunabhängigkeit von JAVA unter verschiedenen Betriebssystemen benutzen zu können Mit Hilfe von „JavaFX Mobile“ (ab Version 1.1) kann die Funktionalität auch auf ein mobiles, mit Java-Laufzeitumgebung ausgestattetes Endgerät übertragen werden. Ergänzend zu Java stellt JavaFX einen produktiven und gemeinsam nutzbaren „Developer-Designer-Workflow“ (deutsch: Entwickler-Designer-Arbeitsprozess) in Form einer typisierten, deklarativen Sprache, JavaFX-Script, zur Verfügung. Die Firma Sun Microsystems versucht damit explizit auch Designer in den Arbeitsprozess mit einzubinden, um den Produktionszyklus zu beschleunigen. Obwohl sich weiterhin Javacode ausführen lässt, weicht Sun Microsystems, um den Entwicklungsprozess zu vereinfachen, von den üblichen Javakonventionen ab und vergleicht JavaFX Script selbst eher mit Javascript als mit Java [Ste09]. Für JavaFX-Anwendungen wird heutzutage nur noch eine standard Java-Laufzeitumgebung ab Version 1.5 benötigt. Diese Laufzeitumgebung ist angesichts der recht weiten Verbreitung von Java auf vielen Endgeräten installiert und wird, wenn nicht vorhanden, beim ersten Start einer JavaFX Anwendung automatisch auf eine Windows, Mac OS, Linux und Open Solaris Plattform geladen, die Java 1.5 oder höher unterstützt.

[1] http://javafx.com/faq/

### 1.2.3 Silverlight

Silverlight[2] ist eine auf „Microsoft .Net Framework“ basierende Technologie mit dem Ziel leistungsfähige browser- und plattformunabhängige Internetanwendungen zu ermöglichen. Die Basis hierfür bildet „Windows Presentation Foundation“ (WPF), welches mit dem „Microsoft.Net Framework 3.0“ eingeführt wurde [Mor06]. WPF ist eine Grundlage der grafischen Nutzeroberfläche bei „Windows Vista“ und „Windows 7“ Anwendungen und zeigt, wie leistungsfähig die grafische Entwicklung auf dem Desktop sein kann. Um diese Vorteile auch im Web anbieten zu können, enthalten alle Silverlight-Anwendungen eine Teilmenge von WPF [SG09]. Den Kern bildet dabei die „Extensible Application Markup Language“ (XAML), eine auf XML („Extensible Markup Language“) basierende Auszeichnungssprache. XAML ist ein gemeinsames Beschreibungsformat für Designer und Entwickler, welche durch Entkopplung der grafischen Benutzeroberfläche vom Rest der Applikation, die Lücke zwischen Design und Programmierung schließt. Hatte Silverlight nach seiner Einführung bei der MIX-2007-Konferenz noch wenige Komponenten, bietet die aktuelle Version 3 (Juni 2009) ein umfassendes Angebot an GUI-Controls und GUI-Funktionen wie HD-Video, 3D-Grafik oder Multitouch-Unterstützung. Diese breite Funktionalität wird in einer Laufzeitumgebung im Browser abgerufen und ist somit unabhängig von dem .Net Framework. Das freie und kleine Browser-Plugin (Silverlight 3 Laufzeitumgebung für Windows ist 4.7 MB groß) ist für alle gängigen Windows- und MAC-Browser erhältlich und wird derzeit von Microsoft und Novell auf Linux portiert. Ab Version 3 ist es auch möglich, Anwendungen aus dem Internet herunterzuladen und ohne zusätzliche Laufzeitumgebung auf dem Desktop zu starten.

## 1.3 Flex - eine Evolution

Um Flex besser zu verstehen, ist es notwendig, neben dem Blick auf andere RIA-Ansätze einen kurzen Blick auf die Wurzeln von Flex: Flash zu werfen. Flash wurde im Jahr 1997 nach der Übernahme der Firma FutureWave durch Macromedia als zeitleistenbasiertes Vektor-Animationsprogramm für 2D-Grafiken veröffentlicht [For05]. Durch ständige Verbesserungen von Flash sowie die Einführung des Javascript ähnlichen Actionscript 1.0 im Jahr 2000, begann Macromedia im Jahr 2002 (FlashMX) die Leistungsfähigkeit von Flash jenseits des Einsatzes für Animationen zu erkennen [Wid08]. Schon ein Jahre später,

[2] http://www.silverlight.net/

mit Einführung von Actionscript 2, einem Java ähnelnden Ansatz zur objektorientierten Programmierung [Grä04], zeigten sich ungeahnte Möglichkeiten für Flash-Entwickler, Browser-basierende Anwendungen zu erstellen. Da Flash aber unter dem Hauptfokus Animation entwickelt wurde, hat sich schnell gezeigt, dass eine eher nach Grafik- oder Videoprogramm anmutende Arbeitsumgebung mit Bühne, Zeitleiste und Malwerkzeugen nicht ideal ist, um reichhaltige Webanwendungen zu erstellen. Aus diesem Grund wurde ein Framework entwickelt, das auch auf Actionscript und dem Flash Player basiert, aber den Ansprüchen der modernen Webentwicklung besser entspricht: Flex. Dieses neue Framework konnte sich jedoch, nach der Einführung von Flex 1.0 im Jahr 2004, angesichts hoher Lizenzgebühren und geringem Funktionsumfang nicht etablieren. Erst mit der Übernahme von Macromedia durch Adobe Systems (Adobe) und der Vorstellung von Flex 2.0 (2004) erhielt Flex, dank der vollwertigen Programmiersprache Actionscript 3.0, eines stark erweiterten Funktionsumfangs und der Freigabe der Flex SDK (Software Development Kid) unter einer „open Source Lizenz“, die erwünschte Aufmerksamkeit. Mit der aktuellen Version Flex 3.0 ist es nunmehr möglich Flex Anwendungen mittels „Adobe Integrated Runtime“ (AIR) auf dem Desktop auszuführen. Bis heute begleitet der von Macromedia im Jahr 1997 eingeführte „Flash Player“ die Flex Entwicklung. Die zu „Shockwave Flash“ (SWF) kompilierter Flash/Flex-Programme werde in dieser Laufzeitumgebung „abgespielt“. Der „Flash Player“ ist somit die Sandbox aller Flash/Flex-Anwendungen und unterstützt seit Version 9 die Programmiersprache Actionscript 3. Die enthaltene „Actionscript Virtual Machine“ 2 (AVM 2), die zum Ausführen des Codes in der SWF genutzt wird, ist im Vergleich zur AVM 1 zehnfach schneller [Rüt09] und bietet neue Funktionen wie zum Beispiel eine Laufzeitfehlerbehandlung. Das zeigt, dass die Optimierung der Laufzeitumgebung seit ihrer Einführung ebenfalls immense Fortschritte gemacht hat.

## 1.4 Bestandteile von Flex 3

Nachdem in Abschnitt 1.2 die Bedeutung und Einordnung von Flex innerhalb bekannter RIA-Technologien und in Abschnitt 1.3 die Entstehung von Flex aufgezeigt wurde, wird nachfolgend auf die Bestandteile und Funktionsweise von Flex-3-Anwendungen eingegangen. Laut Petra Waldminghaus [Wal09] handelt sich es bei Flex um ein „leistungsstarkes Quartett“, bestehend aus Framework, MXML („Magic/Macromedia Extensible Markup

Language“), Actionscript 3 und Flash Player API (Application Programming Interface).

### 1.4.1 Flex-Framework

Das Open-Source-Flex-Framework bildet die Basis für alle Flex-Anwendungen und umfasst alle notwendigen Bibliotheken, um leistungsfähige Web-Programme zu entwerfen. Das Flex-SDK definiert Steuerelemente (Controls) und Datenkomponenten ebenso wie Eventmanagement oder das Verwalten von Styles und Skins. Steuerelemente sind vorwiegend sichtbare Benutzer-Schnittstellen-Komponenten wie Textfelder, Schaltflächen, Listen oder Datengitter. Aber auch unsichtbare Layout-Container wie Canvas[3]- oder ViewStack-Elemente[4], die häufig zur Strukturierung von Oberflächen verwendet werden, bezeichnet man als Steuerelemente. Einen umfassenden Überblick über alle sichtbare Komponenten bietet der „Adobe Flex 3 Component Explorer“[5]. Dieser zeigt die nahezu grenzenlosen Möglichkeiten, ein ansprechendes Flex-GUI zu erstellen, auf und ist besonders für neue Flex-Entwickler zu empfehlen. Mit Hilfe der Datenkomponenten des Flex-SDK ist es möglich, dass Flex-Anwendungen mit anderen Rechnern kommunizieren können. Für die Kommunikation mit serverseitigen Diensten kann man die Funktionalität natürlich auch selbst erstellen, die Datenkomponenten bieten Flex jedoch eine einfache Möglichkeit zur Kommunikation über einen HTTP-Service, Webservice oder Remote-Objekte, was den Programmieraufwand stark verringert. Um zum Beispiel auf die Antworten eines Webservices oder auf Benutzerinteraktion mit dem GUI reagieren zu können, bietet das Flex-SDK ein Eventmanagement. Dieses bildet einen zentralen Punkt im Flex-Framework, da im Gegensatz zur traditionellen Webanwendungen die Anwendungslogik auf den Client ausgelagert ist (vgl. Abschnitt 1.1) und somit auch dort auf Benutzerinteraktion reagiert werden muss. Neben diesen „User Events“ gibt es auch „System Events“ die an definierten Stellen im Lebenszyklus von Komponenten oder beim Erreichen verschiedener Zustände der Anwendungen selbst ausgelöst werden [Wid08]. „Event Listener“ werden an Events gebunden und werden erst beim Auftreten der Events aktiv, um ihre Funktion auszuführen. Die meisten Flex-Komponenten bieten bereits vordefinierte Events. Daneben ist es aber auch möglich, eigene Events zu erstellen und so die Leistungsfähigkeit der Anwendung zu erhöhen.

[3] http://www.adobe.com/livedocs/flex/3/langref/mx/containers/Canvas.html
[4] http://livedocs.adobe.com/flex/3/html/help.html?content=navigators_3.html
[5] http://examples.adobe.com/flex3/componentexplorer/explorer.html

### 1.4.2 MXML

Der zweite Hauptbestandteil des „leistungsstarken Quartetts“ von Flex ist MXML. MXML ist eine deklarative Auszeichnungssprache die, wie der Name schon vermuten lässt, wie Microsofts XAML, auf XML basiert. XML ermöglicht „die Darstellung von hierarchisch strukturierter Daten in Form von Textdateien“ [Wid08]. Dies gilt auch für MXML, aber es werden hier nicht nur Daten sondern, ähnlich wie mit HTML, Steuer- oder Datenkomponenten definiert. Die MXML-Elemente werden dabei in Form von Tags erstellt, deren Eigenschaften man „Inline“ über Tag-Attribute verändern kann. Eine Flex Anwendung besteht dabei aus mindestens einer MXML-Datei. In solche so genannten Haupt-MXML-Dateien können alle anderen MXML-Definitionen eingebettet werden. MXML wird insbesondere genutzt, um effektiv ansprechende Nutzeroberflächen wie Layout und sichtbaren Komponenten, zu erstellen und die Designentwicklung vom Rest der Anwendung zu trennen. Einfache Flex-Anwendungen können vollständig in MXML erstellt werden. Eine Umsetzung komplexer Anwendungslogik jenseits von „Databinding“ ist mittels MXML allein jedoch nicht möglich. Das Fehlen von Kontrollstrukturen oder Schleifen und fehlende Möglichkeiten zur dynamischen Erzeugung von Komponenten mittels MXML machen den Einsatz der Flex-Programmiersprache Actionscript 3 erforderlich.

### 1.4.3 Actionscript 3

Wie bereits im Abschnitt 1.3 aufgezeigt, durchlief Actionscript (AS) eine gewaltige Entwicklung. Dabei orientiert sich die objektorientierte Sprache seit Version 2 an ECMAScript-Standarts[6] und befindet sich heute in der dritten Generation. Diese dritte Version von AS gleicht eher umfangreichen Sprachen wie Java als Scriptsprachen wie Javascript und bietet, dank objektorientierter Programmierung (OOP), die Möglichkeit für strukturiertes und effizientes Arbeiten an einer Flex-Anwendung. Mit Actionscript 3 (AS3) sind somit Flex-Entwicklungen, die allein mit MXML nicht möglich wären, realisierbar. Abgesehen von dem Haupt-MXML-Tag können selbst komplexe Anwendungen ausschließlich mit AS3 programmiert werden, was die zentrale Stellung von AS3 in Flex-Anwendungen verdeutlicht. In AS werden nicht nur Flex-Programme programmiert, sondern auch das Flex-SDK selber wurde mittels Actionscript erstellt. „*Adobe selbst bezeichnet AS3 als Program-*

[6] http://www.ecmascript.org/docs.php

*miersprache für den Adobe Flash Player und die Adobe AIR Run-Time-Environments*“ [Rüt09].

### 1.4.4 Flash Player API

Der letzte zu betrachtende Bestandteil von Flex ist die Flash Player API. Diese ist ein wichtiger Bestandteil des Flash Players und stellt die wesentlichen Klassen wie beispielsweise String oder Array, sowie verschiedene Sprachkonstrukte wie Schleifen oder Bedingungen bereit. Aber auch komplexere Funktionalitäten wie z.B. der Umgang mit Sound sind in der Flash Player API enthalten. Das Flex-SDK, mit allen Flex-spezifischen Komponenten und Funktionalitäten baut auf die Flash Player API auf. Auf Grund dieser zentralen Rolle von AS ist es nicht verwunderlich, dass beim Kompilieren einer Anwendung zu einer SWF zunächst alle MXML-Komponenten zu AS-Code übersetzt werden und erst im Anschluss daran ausführbarer Code erstellt wird. Man könnte annehmen, dass MXML damit überflüssig ist, doch ein besonderer Vorteil von Flex zeigt sich erst in der Kombination der beiden Sprachen. Mittels MXML ist es problemlos möglich, übersichtliche Benutzeroberflächen zu erstellen und dann mit Hilfe von AS weiterführende Anwendungslogik zu implementieren.

## 1.5 Effizientes Arbeiten mit Flex

Nachfolgend wird effizientes Arbeiten mit der Flex-Technologie beschrieben. Dazu wird zunächst die bekannteste Entwicklungsumgebung vorgestellt und anschließend ein Überblick über den Aufbau und gängige Standards einer Flex-Anwendung gegeben.

### 1.5.1 Adobe Flex Builder

MXML- und AS-Dateien sind einfache Textdateien, weshalb diese in einem beliebigen Texteditor bearbeitet und anschließend per Kommandozeile einer Konsole kompiliert werden können. Für die Praxis ist diese Herangehensweise allerdings wenig komfortabel, weshalb die meisten Entwickler eine Entwicklungsumgebung bevorzugen. Eine kostenfreie Arbeitsumgebung bietet zum Beispiel „FlashDevelop“[7]. Aber auch mit dem

[7] http://www.flashdevelop.org/wikidocs/index.php?title=Main_Page

unentgeltlichen „Amethyst“[8], einem auf Visual Studio aufbauenden Ansatz, ist eine Flex-Entwicklung möglich. Eine weit verbreitete aber kostenpflichtige Entwicklungsumgebung ist der „Adobe Flex Builder“ (FB), eine auf Eclipse basierende Lösung zur effizienten Erstellung von Flex Anwendungen von Adobe. In seiner aktuellen Version 3 unterstützt der FB den Flex-Entwicklungsprozess mit verschiedenen Werkzeugen für Fehlersuche (Debugging), Code-Erstellung oder die Gestaltung von Nutzeroberflächen. Zusätzliche Funktionen wie die Erstellung von Diagrammen, Leistungsprofilen oder automatische Programmtests [Ado08] bietet der „Adobe Flex Builder Professional“, welcher aber auch wesentlich teurer als die Standard-Version ist. Mit dem im „Adobe Flex Builder“ und „Adobe Flex Builder Professional“ enthaltenen „Design-Modus“ können verschiedene Steuerkomponenten wie Buttons und Listen aber auch Layoutcontainer, Navigationselemente oder selbst erstellte Komponenten einfach und intuitiv mittels Drag-and-Drop in eine grafisch dargestellte MXML integriert werden. Diese grafische MXML-Darstellung ist ähnlich wie in Flash eine Art Bühne, auf der alle in der MXML enthaltenen Elemente in ihrer tatsächlichen Form und Position betrachtet, angeordnet oder verändert werden können. In einem anderen Modus, dem „Source-Modus“, sieht man diese Elemente in gewohnter XML-Form. Natürlich ist es auch möglich, in diesem textbasierten Modus Komponenten mittels MXML-Tag zu erstellen. Ergänzend dazu kann per MXML-Script-Tag Actionscript-Code ausgeführt werden. So entstehen sehr schnell hochinteraktive und ansprechende Nutzeroberflächen, die in Verbindung mit beispielsweise einem Webservice erstaunlich funktional im Vergleich zum Aufwand sind.

### 1.5.2 Verbreitete Flex-Entwicklungsstandards

Grundlegend gibt es viele Möglichkeiten, Flex-Anwendungen zu erstellen. Gängige Praxis ist es jedoch, eine große MXML in mehrere kleine MXMLs zu zerlegen, um einen besseren Überblick zu behalten. Die einzelnen Teile werden dabei hierarchisch angeordnet, oft auch durch Container zusammengefasst bzw. verwaltet und bilden so eine bessere Basis für Teamarbeit und Fehlersuche [KL08]. Um wiederverwendbare Komponenten zu erhalten, sollten diese vom Rest der Anwendung möglichst unabhängig sein.

Die Komponentenkommunikation ist in klassischen Flex-Anwendungen daher in Form von Events bzw. durch Databinding[9] implementiert. Eine Komponente sendet dazu ein

[8] http://www.sapphiresteel.com/Adobe-Flex-In-Visual-Studio
[9] http://livedocs.adobe.com/flex/3/html/help.html?content=databinding_2.html

Event, welches von hierarchisch übergeordneten Komponenten empfangen und verarbeitet werden kann. Die Daten werden hierarchisch tiefer liegenden Komponenten durch Databinding von einem sogenannten Eltern-Element zu Verfügung gestellt. Abbildung 1.1 zeigt diese hierarchische Anordnung von Komponenten klassischer Flex-Anwendungen, aufgeteilt in Container, Komponenten (View) und die Haupt-MXML (Main View) sowie deren Kommunikation. Programmlogik wird häufig mit verschiedensten Ansätzen in der Haupt-MXML implementiert.

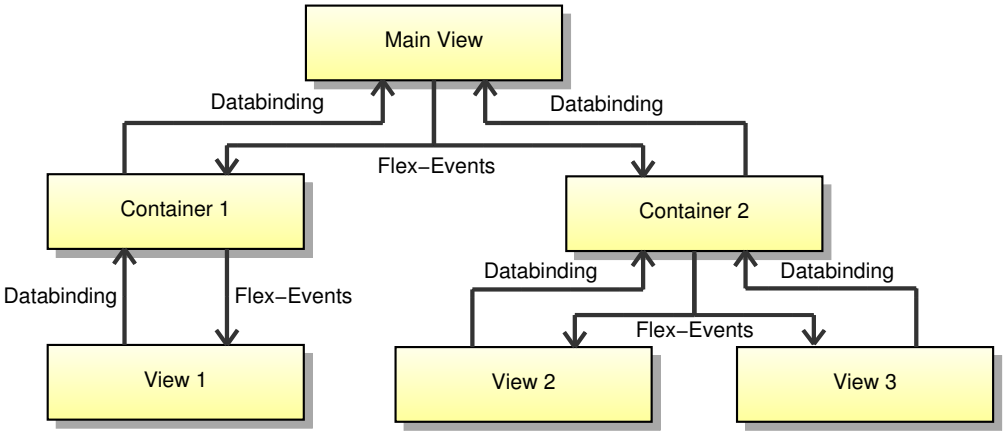


Abbildung 1.1: Hierarchiche Anordung von MXML-Komponenten

Die grafischen Eigenschaften aller Komponenten können, wie in Abschnitt 1.4.2 beschrieben, Inline definiert werden. Chafic Kazoun und Joey Lott [KL08] empfehlen jedoch, diese in CSS-Datei(en) auszulagern. Dadurch haben Nicht-Flex-Entwicklern die Möglichkeit, Änderungen am Styling vorzunehmen, ohne zeitaufwendige nach Inline-Attributen in den MXML-Dateien suchen zu müssen. Des Weiteren hat es sich als vorteilhaft erwiesen, Dateien geordnet abzuspeichern. Dateien können somit schneller wiedergefunden werden und helfen Namenskonflikte zu vermeiden. Es empfiehlt sich beispielsweise, alle Views und Container einem Unterordner „Views“ zuzuordnen. Analog dazu werden erstellte Event-, Service- oder Daten-Klassen häufig in „Packages“ (Pakete) wie „events“, „services“ oder „models“ organisiert. Werden diese einfachen und leicht umzusetzenden Empfehlungen befolgt, können kleine Projekte besser organisiert und übersichtlich gestaltet werden.

Was aber passiert, wenn sich die Anwendung um ein Vielfaches vergrößert, der Webserver sich ändert oder Nutzeroberflächen im großen Stil angepasst werden müssen [Ber06]? Endet der Ansatz leistungsfähiger RIA-Flex-Anwendungen dort, wo man Anforderungen aus dem wahren Leben mit einbezieht? Wenn neben den eben eingeführten gängigen

Praxen eine Software-Architektur eingeführt wird, die über die Aufteilung von Komponenten und deren Kommunikation mit Events und Databinding hinausgeht, muss das nicht der Fall sein. Im Gegenteil, Software-Architekturen machen Flex-Anwendungen flexibel, wartbar und übersichtlich weshalb sie eine wichtige Basis für professionelles Arbeiten mit dieser Technologie bilden.

# 2 Software-Architektur und Architektur-Frameworks

In diesem Kapitel werden grundlegende Fragen zur Thematik Softwarearchitektur beantwortet, Muster in Softwarearchitektur eingeordnet und der Zusammenhang zu Architektur-Frameworks dargestellt. Bevor Abschnitt 2.2 zeigt was Softwarearchitektur umfasst und wo diese einzuordnen ist, erläutert Abschnitt 2.1 warum Softwarearchitektur in der Softwareindustrie überhaupt von Bedeutung ist. Anschließend wird in Abschnitt 2.3 gezeigt, welche Ziele gute Softwarearchitektur verfolgt und womit diese erreicht werden können. Abschnitt 2.4 befasst sich nachfolgend mit Mustern in der Softwareentwicklung, um anschließend den Zusammenhang zu Architektur-Frameworks, welche im Abschnitt 2.5 eingeführt werden, herstellen zu können.

## 2.1 Warum Softwarearchitektur

Heutige Softwareunternehmen können sich am Markt nur dann durchsetzen, wenn sie in der Lage sind, effiziente und effektive Softwaresysteme herzustellen. Posch und Birken [PBG07] nennen drei wesentliche Faktoren, die entscheidend sind, um eine hohe Wettbewerbsfähigkeit zu erreichen: die Verkürzung der Entwicklungszeit, die Reduzierung der Wartungs- und Entwicklungskosten sowie die Verbesserung der Softwarequalität. Das in Abbildung 2.1 dargestellte Spannungsdreieck zeigt eben diese Faktoren im Zusammenhang mit der in den letzten Jahren stark gestiegenen Softwarekomplexität. Auf Grund steigender Komplexität kann oft beobachtet werden, dass bei unbewussten oder zufällig entstandenen Softwarearchitekturen (SA) die Entwicklungszeiten und vor allem die Wartungszeiten außerordentlich ansteigen. Da Kunden meist aber kein Verständnis für lange Wartezeiten, weder bei Entwicklung noch Wartung haben und auch die damit verbundenen Kosten die Vorgaben übersteigen, werden Softwaresysteme im Fehler- oder

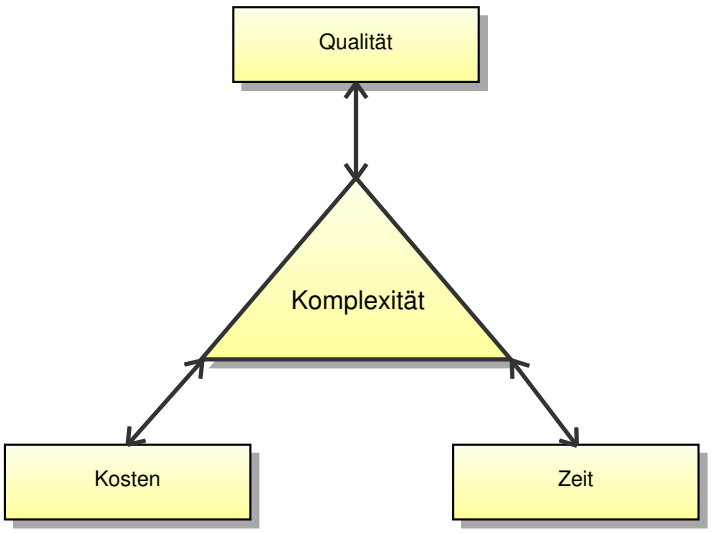


Abbildung 2.1: Spannungsdreieck
(in Anlehnung an Torsten Posch, Klaus Birken und Michael Gerdom [PBG07])

Erweiterungsfall oft nur ausgebessert oder oberflächlich bearbeitet. Das entstandene Flickwerk verwischt die vorhandenen systematischen Strukturen was dazu führt, dass Qualitätsmerkmale wie Funktionalität, Effizienz oder Änderbarkeit [Sta08] der Software negativ beeinflusst werden. Softwaresysteme, ohne bewusst gewählte SA, verändern sich daher im Laufe der Zeit meist hin zu undurchschaubaren Gebilden, bei denen oft nicht deutlich wird, warum sie noch funktionieren [VAC+05]. Diese Problematik verdeutlicht die Relevanz geplanter SA im Prozess der Softwareentwicklung. Um später aufzeigen zu können womit eine effiziente SA erreicht werden kann, ist es erst nötig zu erläutern, was SA eigentlich genau umfasst. Nachfolgender Abschnitt geht auf diese Frage ein.

## 2.2 Was ist Softwarearchitektur

Um das Wesen der SA zu verdeutlichen, wird diese zunächst in den Prozess des Software-Engineering[1] eingeordnet. Nachdem der Zusammenhang zwischen verschiedenen Teilprozessen des Software-Engineering hergestellt wurde, wird eine Definition des Begriffs SA erarbeitet.

Durch die Komplexität heutiger Softwareprojekte ist es notwendig geworden, neben

[1] *„Software Engineering ist jede Aktivität, bei der es um eine Erstellung oder Veränderung von Software selbst geht, soweit mit der Software Ziele verfolgt werden, die über die Software selbst hinaus geht"[LL07]*

der SA andere Aspekte der Architektur wie zum Beispiel Daten-, Netzwerk- oder Sicherheitsarchitektur zu berücksichtigen, welche im Rahmen dieser Arbeit aber nicht näher betrachtet werden [VAC+05].

### 2.2.1 Softwarearchitektur als Prozess des Software-Engineering

Softwarearchitektur ist laut Jochen Ludewig [LL07] als Übergang zwischen der Anforderungsanalyse und der Implementierung des Softwareentwicklungsprozesses eingeordnet. Der Entstehungsprozess einer Softwarearchitektur ist somit ein Teilprozess des Software-Engineering und folgt auf Systemanalyse und Festlegung der Softwarespezifikationen, aber vor Implementierung und Test. Abbildung 2.2 verdeutlicht diese vereinfachte Abfolge. In Softwareprojekten überlappen sich diese Aktivitäten und werden mehrfach unter Berücksichtigung verschiedener Schwerpunkte, wiederholt. Auffallend in Abbildung 2.2 ist der bisher noch nicht erwähnte Begriff „Softwareentwurf“. Auf den Zusammenhang zwischen SA, Softwareentwurf und Implementierung wird im nächsten Abschnitt eingegangen.

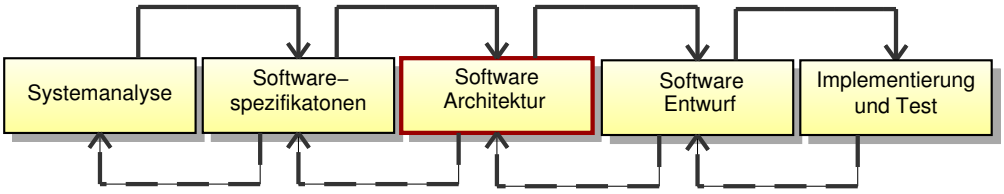


Abbildung 2.2: Prozesse im Software-Engineering

### 2.2.2 Softwarearchitektur, Softwareentwurf und Implementierung

Um später Architektur-Frameworks bzw. Softwaremuster in das Gebiet der SA einordnen zu können ist es wichtig die Grenzen zwischen SA, Softwareentwurf (oder Softwaredesign) und Implementierung der SA herauszuarbeiten. Abbildung 2.3 verdeutlicht den Zusammenhang zwischen diesen Gebieten. Die Verbindungspfeile stellen lediglich einen beispielhaften Zusammenhang zwischen den jeweiligen Elementen dar. In Anlehnung an Gernot Starke [Sta08] wurde SA in zwei Ebenen unterteilt. Dabei gehört sowohl die Interaktion zwischen Systemen (Subsystem) als auch die Interaktion zwischen System-Komponenten zu einer SA. Die programmiersprachenabhängige Implementierung der SA gehört, wie bereits in Abschnitt 2.2.1 dargelegt, nicht zur SA. Strittig ist in der Literatur jedoch ob der Softwareentwurf ein Teilprozess der SA ist.

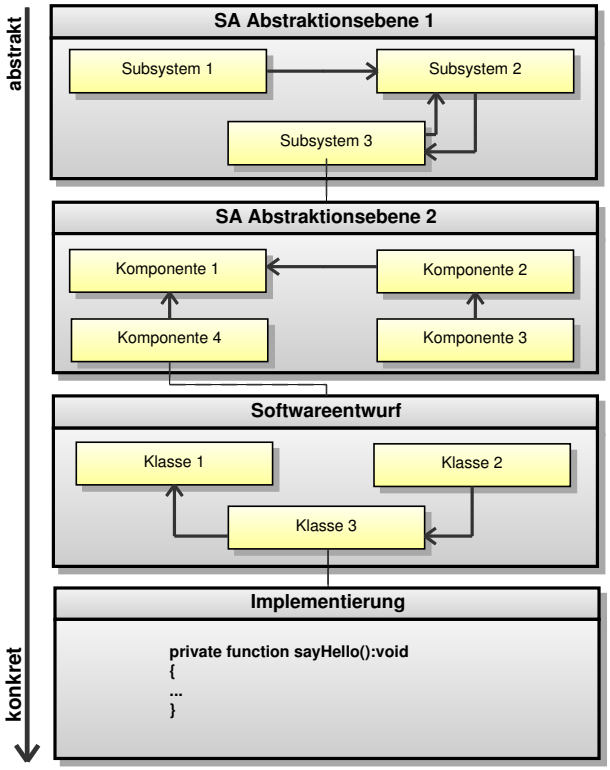


Abbildung 2.3: Softwarearchitektur, Softwareentwurf und Implementierung

Der innere Aufbau der SA, in Form von Klassen, gehört laut Sven Busse [Bus09] nicht mehr zur eigentlichen Softwarearchitektur, und wird Software Entwurf oder Software Design genannt. Anders Gernot Starke [Sta08], der die Auffassung vertritt, dass der Übergang von Softwareentwurf und SA fließend ist und mit diesen Begriffen pragmatisch umgegangen werden sollte. Aufgrund dieser unterschiedlichen Auffassungen kann SA und Softwareentwurf nicht eindeutig getrennt werden. Festzuhalten bleibt, dass ausgehend von der SA hin zur Implementierung das entstehende Programm mehr und mehr konkretisiert wird. Dabei haben Änderungen auf einer „höheren Ebene“ (abstrakt) meist Einfluss auf darunter liegende Ebenen. Wie man in der Abbildung 2.3 erkennen kann, befinden sich auf den unterschiedlichen Ebenen verschiedene Elemente („Subsystem“, „Komponente“, „Klasse“). Diese Elemente werden in der Literatur meist als Baussteine zusammengefasst. Sofern eine genauere Unterteilung nicht nötig ist, wird dieser allgemeine Begriff in dieser Arbeit analog zur Literatur genutzt.

### 2.2.3 Definition des Begriffs Softwarearchitektur

Ansätze für SA findet man bereits in den 60er Jahren [Sim05]. Bis heute hat sich allerdings keine allgemein anerkannte Definition für SA herausgebildet. An dem renommierten „Software Engineering Institute“ (SEI) der „Carnegie-Mellon University“ in Pittsburgh wurden unzählige Definitionen für SA gesammelt, was verdeutlicht,wie groß der Interpretationsspielraum von SA ist. SA ist dabei nicht nur die Beschreibung strukturierter Bausteine und deren Beziehungen zueinander, sondern auch die damit verbundenen architektonischen oder organisatorischen Prozesse zur Konzeption einer SA dar. In der Literatur wird häufig die Definition nach Bass [BCK04, VAC+05], welche auf die Beschreibung der Software Strukturen abzielt, zitiert.

> *„Die Software-Architektur eines Systems beschreibt dessen Software-Struktur respektive dessen –Struktur, dessen Software- Bausteine sowie deren sichtbaren Eigenschaften und Beziehungen zueinander“*

Eine Definition die SA als Disziplin beschreibt, findet man bei O. Vogel [VAC+05].

> *„Software-Architektur als Disziplin befasst sich mit den architektonischen Tätigkeiten und den hiermit verbundenen Entscheidungen zu Konzeption und Realisierung einer Software-Architektur“*

Das SEI bietet eine Definition von SA, welche die Definition von Bass und Vogel allgemeiner, aber dennoch treffend zusammenfasst [Per95]. Softwarearchitektur wird dabei verstanden als:

> *„The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.“*

## 2.3 Prinzipen der Softwarearchitektur

Nachdem geklärt wurde, warum SA wichtig ist und wie dieser Begriff definiert ist, soll nachfolgend darauf eingegangen werden, unter Anwendung *welcher Prinzipen* Softwarearchitektur erstellt, weiterentwickelt und beschrieben werden kann. Der Fokus dieser Arbeit liegt auf anwendbaren Prinzipien, die beim Erstellen und Bewerten einer SA beachtet werden sollten. Ziel dieser Prinzipien ist vorrangig die Reduktion der Softwarekomplexität und die Erhöhung der Flexibilität bzw. Änderbarkeit einer Architektur. Eine Missachtung

der Prinzipen aus Abschnitt 2.3.1 bzw. Abschnitt 2.3.2 deutet auf eine schlecht oder unbewusst erstellte SA hin, kann im Einzellfall aber auch beabsichtigt worden sein [VAC+05].

### 2.3.1 Reduktion der Softwarekomplexität

Ein weit verbreitetes Vorgehen zur Reduktion der Softwarekomplexität ist das *Prinzip des „Separation of Concerns“* [VAC+05] (SoC; deutsch: Trennung von Angelegenheiten). Mit Hilfe dieses allgemeinen Prinzips werden in der SA vielseitige Bereiche wie beispielsweise die Zerlegung der Prozesse zum Erstellen einer SA oder die Trennung von SA-Anforderungen, bearbeit. Der wichtigste Einsatz des SoC ist aber die Unterstützung der Modularisierung, die Bildung von Bausteinen einer SA (= *Prinzip der Modularisierung*). Durch die Bildung von Bausteinen werden Teile des Softwaresystems auf unterschiedlichen Abstraktionsebenen nach Aufgaben, Angelegenheiten oder Aspekten identifiziert und gekapselt. Die entstandenen Systemteile tragen sehr zur Verständlichkeit des Gesamtsystems bei und unterstützen durch das Verteilen von Verantwortlichkeiten das Arbeiten in Teams. Nach dem Prinzip der Modularisierung haben Systembausteine klar abgegrenzte Zuständigkeiten, sind in sich selbst geschlossen und bieten klar definierte Schnittstellen zur Kommunikation mit anderen Systembausteinen. Eine gute Modularisierung erkennt man laut Bertrand Meyer [MH97], an fünf Kriterien. Systembausteine sollen nach diesen Kriterien möglichst...

- MODULARE DEKOMPOSITION: ...unabhängig voneinander sein, so dass man sie unabhängig voneinander bearbeiten kann.
- MODULARE KOMPOSITION: ...in verschiedenen SA untereinander frei kombiniert werden können.
- MODULARE VERSTEHBARKEIT: ...ohne weitere Systembausteine untersuchen zu müssen verständlich sein.
- MODULARE KONTINUITÄT: ...unverändert bleiben wenn andere Systembausteine geändert werden müssen
- MODULARE PROTEKTION: ... Auswirkungen intern entstandener Fehler auf das Gesamtsystem vermeiden

Die Anwendung dieser fünf Kriterien unterstützen Softwarearchitekten bei der Erstellung von Systembausteinen und führen meist zu verständlichen und weniger komplexen Gesamtsystemen.

### 2.3.2 Erhöhung der Softwareflexibilität

Modularität unterstützt, neben Komplexitätsreduktion durch Kapselung eines Softwaresystems, auch die Flexibilität einer SA. Entstandene Softwarebausteine können bei Änderung der Systemspezifikation ausgetauscht oder verändert werden, ohne andere Bausteine oder gar das Gesamtsystem anpassen zu müssen. Das ist jedoch nur möglich, wenn geringe Abhängigkeiten zwischen den Softwarebausteinen bestehen. Diese Abhänhigkeiten (Kopplungen) können quantitativ und qualitativ betrachtet werden, wobei die Flexibilität des Systems durch wenige und „schmale“ Kopplungen profitiert. Diese Herangehensweise wird als *Prinzip der losen Kopplung* [VAC+05] bezeichnet und gilt als Grundlage für Änderungen oder den Austausch einzelner Bausteine, ohne ausführliche Betrachtung der Systemumgebung. Um eine lose Kopplung zwischen Systembausteinen zu erreichen, gibt es verschiedene Ansätze. Eine Möglichkeit ist das Einsetzen von Mustern wie dem Beobachter-Muster (vgl. Abschnitt 2.4.3) oder das Nutzen von Interfaces. Eine Umsetzung der losen Kopplung mittels Interfaces zeigt sich in der Anwendung des *Prinzips „Inversion of Control“* (IoC) [VAC+05]. Auf Klassenebene besagt dieses Prinzip, dass ein Interface-Objekt einer Klasse nicht von der Klasse selbst konkretisiert, sondern von einer anderen Klasse übergeben wird. Den konkreten Objekttyp muss die Klasse, welche das Interface implementiert, dazu nicht kennen, was zu einer hohen Entkopplung, aber auch zu mehr Komplexität führt. Zirkuläre Abhängigkeiten zwischen Bausteinen eines Systems führen zu besonders hohen Abhängigkeiten und sind deshalb möglichst zu vermeiden.

## 2.4 Softwaremuster

Die im Abschnitt 2.3 erläuterten Prinzipien und Kriterien der Modularisierung sind nicht einfach einzuhalten und erfordern viel Erfahrung in der Softwarearchitektur bzw. der Programmierung. Muster sind eine gute Möglichkeit, auf das Wissen erfahrener Softwarearchitekten zuzugreifen und so die Qualität der Software zu verbessern. Dieser Abschnitt beschreibt was Muster sind, welche Eigenschaften Muster haben und welche

Kategorien von Mustern in der SA zu finden sind.

### 2.4.1 Was ist ein Softwaremuster

Um auftretende Probleme der Softwarearchitektur lösen zu können, erarbeiten erfahrene Softwarearchitekten meist keinen komplett neuen Lösungsansatz, sondern wenden eine Grundidee, welche auf bereits gefundene Lösungen ähnlicher Probleme basiert, an. Derartige Grundideen entstehen durch das Abstrahieren von Gemeinsamkeiten bekannter Problem-Lösungs-Paare und werden als Muster (engl. Pattern) bezeichnet [BMR+98]. Solche Muster können nun für die Lösung anderer, ähnlicher Probleme eingesetzt werden. Nicht nur in der SA, sondern auch in anderen Bereichen des Lebens beschreiben Muster Lösungsansätze für wiederkehrende Probleme. Die Definition, auf die sich die meisten SA Bücher stützen, stammt von dem Architekten Christopher Alexander, der Muster als „...dreiteilige Regel, die die Beziehung zwischen einem bestimmten Kontext, einem Problem und einer Lösung ausdrückt“ [BMR+98] treffend beschreibt. In der Softwarearchitektur besteht ein Muster in der Regel aus mehreren Bausteinen und beschreibt deren Zuständigkeiten, Zusammenarbeit und Beziehungen zueinander. Alle vom Muster beschriebenen Elemente lösen das spezielle Problem gemeinsam und das, im Hinblick auf die entstehende SA, gewöhnlich effektiver, als es beispielsweise eine Einzelkomponente realisieren könnte [BMR+98]. Muster dokumentieren dazu ihr Verhalten bzw. Konsequenzen für zum Beispiel Performance oder Wiederverwendbarkeit und definieren Vokabular für enthaltene Bausteine. Sie vereinfachen somit die Kommunikation zwischen Softwarearchitekten, erhöhen die Verständlichkeit des Gesamtsystems, helfen SA treffend zu dokumentieren [VAC+05] und verbessern meist die Software Qualität. Untersucht man unterschiedliche Softwaremuster, wie zum Beispiel im Buch der „Gang- of- Four“ (Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides) [GHJV95], fällt auf, dass sich Muster auf verschiedene, im Abschnitt 2.2.2 eingeführte SA-Ebenen beziehen. Daraus resultierend wird in der Literatur oft zwischen Architekturmustern und Entwurfsmustern [BMR+98] unterschieden.

### 2.4.2 Architekturmuster

Architekturmuster (engl. architectural pattern) beschreiben die grundsätzliche Struktur des Gesamt- oder Subsystems und beeinflussen oft hierarchisch tiefer liegende Abstrakti-

onsebenen. Ein Architekturmuster kann als Schablone für konkrete SA angesehen werden. Derartige Schablonen stellen eine Grundsatzentscheidung im Erstellungsprozess der SA dar, welche grundlegende Bausteine und Regeln vordefinieren. Dabei beschreiben die Regeln die Beziehungen und Zuständigkeiten der einzelnen Bausteine. Beispiele für Architekturmuster sind z.B. das Schichten- (Layer) oder das Pipes&Filter-Muster [BMR+98]. Ein weit verbreitetes Architekturmuster für Programme mit hoch interaktiven Benutzeroberflächen wie bei Flex-Anwendungen, ist das Model-View-Controller-Muster (MVC) [SC07]. Wie der Name vermuten lässt, handelt es sich dabei um ein Architekturmuster mit drei Bausteinen: Model (Daten), View (Benutzerschnittstellen), Controller (Logik). Der sichtbare Teil der Anwendung ist die View, welche Daten aus dem Model nutzt, um diese für den Systembenutzer darzustellen. Das Model beinhaltet die Objekte zur Datenhaltung, die nach Nutzerinteraktion in der View durch den Controler geändert werden. Das MVC-Muster macht keine genauen Aussagen zum Aufbau der Bausteine, stellt aber grundlegende Regelungen zu deren Kommunikation auf, was in Abbildung 2.4 vereinfacht dargestellt ist.

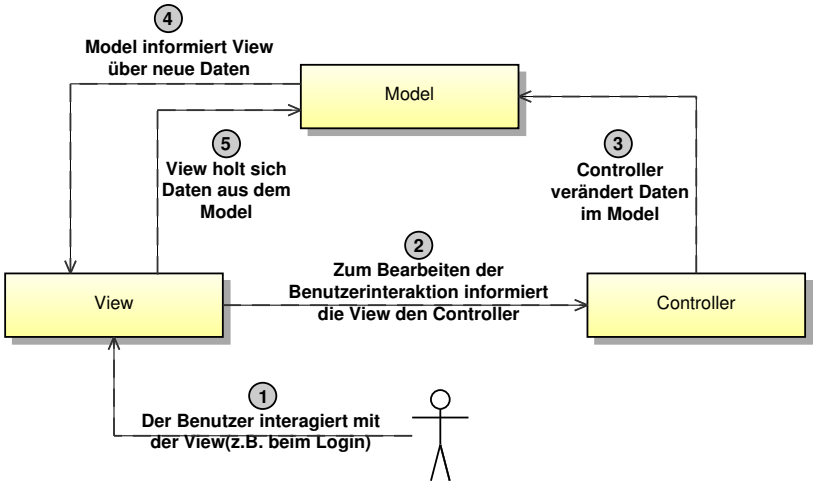


Abbildung 2.4: MVC Aufbau und Funktion
(in Anlehnung an William Sanders und Chandima Cumaranatunge [SC07])

Hoch interaktive MVC-Systeme profitieren vor allem davon, dass die Programmlogik von der Benutzerschnittstelle des Systems unabhängig ist, was eine spätere Änderung der Benutzerschnittstelle vereinfacht und damit Kosten und Wartungszeit verringert.

Die Verbindungen der einzelnen Bausteine zeigen die quantitative Kopplung zwischen den Komponenten. Diese Abhängigkeiten können nach dem Prinzip der losen Kopplung

(vgl. Abschnitt 2.3.2) mit Hilfe von Entwurfsmustern qualitativ verbessert werden.

### 2.4.3 Entwurfsmuster

Entwurfsmuster (engl. design pattern) wirken im Gegensatz zu Architekturmustern innerhalb von Bausteinen des Gesamtsystems. Eine Anwendung von Entwurfsmustern hat daher wenig Einfluss auf die Gesamtarchitektur, weshalb Entwurfsmuster als Muster mittlerer Abstraktionsebene bezeichnet werden. Entwurfsmuster können dazu beitragen, die abstrakteren Architekturmuster effektiver umzusetzen, ohne dabei von einer bestimmten Programmiersprache abhängig zu sein [PBG07]. Die „Gang of Four“ unterteilen die Entwurfmuster weiterführend in Erzeugungsmuster (engl. Creational Patterns), Strukturmuster (engl. Structural Patterns) und Verhaltensmuster (engl. Behavioral Patterns). Die Anwendung eines Entwurfsmusters, wie beispielsweise des Beobachtermusters (engl. observer pattern), ein Strukturmuster, in Verbindung mit einem MVC-Architektur-Muster, führt zu einer höheren Unabhängigkeit zwischen den einzelnen Bausteinen. Laut dem Autor Gernot Starke [Sta08] ist ein Beobachter-Muster *„eine kontrollierte Abhängigkeit zwischen Objekten, so dass die Änderung eines Objektes die Benachrichtigung und Aktualisierung aller abhängigen Objekte auslöst“* (vgl. Abbildung 2.5).

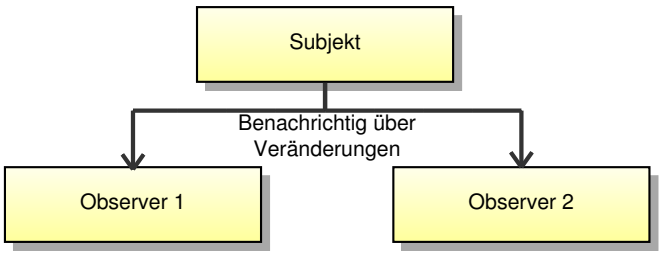


Abbildung 2.5: Observer-Muster

So kann sich eine View (Observer) des MVC- Musters beim Model (Subjekt) anmelden und wird benachrichtig wenn sich dieses ändert, um anschließend die neuen Daten dem Benutzer anzuzeigen. Das Subjekt muss dazu keine Details der Observer kennen, sondern lediglich wissen, welche Observer sich für die Aktualisierung angemeldet haben. Der Einsatz des Beobachtermusters führt somit zu einer losen Kopplung von Systembausteinen, was, wie bereits in Abschnitt 2.3.2 dargestellt, die Softwareflexibilität erhöht.

## 2.5 Architektur-Frameworks

Nachdem in Abschnitt 2.4 Softwaremuster mit SA in Beziehung gesetzt wurden, soll dieser Abschnitt den Zusammenhang zwischen Softwaremustern und Architektur-Frameworks (AF) aufzeigen. Dazu wird einleitend erläutert was AF sind, um nachfolgend auf die Vorteile des Einsatzes von AF einzugehen. Abschließend wird aufgezeigt unter welchen Umständen AF nicht eingesetzt werden sollten.

### 2.5.1 Was sind Architektur-Frameworks

Während Architektur-Muster und Entwurfs-Muster eine Lösung für ein bestimmtes Problem aufzeigen, kann die gezielte Zusammenstellung mehrerer solcher Muster ein Grundgerüst für SA bilden. Diese Zusammenstellung ist allgemein genug, um für verschiede Softwareprojekte eine gemeinsame Basisstruktur zu bilden und wird als AF bezeichnet. In der Softwareentwicklung wird der Begriff Framework jedoch häufig genutzt, wie beispielsweise Application-Framework, Debugging-Framework oder Architektur-Framework. Dabei werden aber zum Beispiel Application-Framework und AF oft nicht differenziert, was zu Missverständnissen und daraus resultierenden Unklarheiten im Software Entwicklungsprozess führen kann. Ein Application-Framework, wie beispielsweise Flex, bietet eine Sammlung von flexiblen Bibliotheken, welche vom Benutzer eingesetzt werden, um Funktionalitäten seines Programms umzusetzen (vgl. Abschnitt 1.4). Ein AF bietet jedoch lediglich eine architektonische Grundstruktur, die genutzt werden kann, um Programmfunktionalität im Sinne der SA, effektiv zu strukturieren [WT08]. Dazu wendet ein AF im Allgemeinen nur ein grundlegendes Architekturmuster, wie zum Beispiel MVC, aber mehrere Entwurfsmuster an. Für Flex-Anwendungen gibt es eine Fülle von Architektur-Frameworks welche sich alle den Herausforderungen der Flex-Entwicklung stellen, aber jeweils einen anderen Fokus auf die SA legen. Allen gemeinsam sind aber verschiedene Vorteile, welche sich teilweise aus den Vorteilen der Softwaremuster ableiten lassen und im nachfolgenden Abschnitt erklärt werden.

### 2.5.2 Vorteile von Architektur Frameworks

Durch fest definierte Bezeichnungen der AF-Bestandteile werden Missverständnisse vermieden und somit die Verständigung innerhalb des Entwicklerteams stark vereinfacht. Das betrifft, im Unterschied zu Software-Mustern, nicht nur einzelne Programmteile,

sondern das gesamte Software Projekt. Dieser Umstand macht es einfach, neue Entwickler in einen bestehenden Entwicklungsprozess einzubinden, wenn diese im Umgang mit dem verwendeten AF bereits vertraut sind [Gra09]. Die unter Verwendung von AF entstehenden Programme sind, im Sinne der SA, allgemein qualitative hochwertige Anwendungen, welche dem Entwicklerteam die Modularisierung vereinfachen, Kommunikationsregeln zwischen den Bausteinen vorgeben, Anordnung einzelner Programmteile abnehmen und damit helfen, Entwicklungszeiten zu verkürzen bzw. Kosten zu senken. Gerade die vorgegebene Anordnung von Programmteilen ermöglicht es auch unerfahrenen Softwareentwicklern, gute SA zu erstellen, ohne das gesamte, vom Softwarearchitekten gewählte, AF verstehen zu müssen. Zusammengefasst bieten AF Unternehmen langfristig die Möglichkeit, Entwicklungs- und Wartungszeiten durch erhöhte Softwarequalität zu minimieren, Kosten zu sparen, und sich für die steigende Softwarekomplexität, gerade im Segment der RIA, zu wappnen.

### 2.5.3 Wann sollten Architektur-Frameworks nicht angewendet werden

Wenn Softwareentwickler erstmalig mit SA-Werkzeugen, wie Softwaremustern oder AF in Kontakt kommen, führt die Begeisterung über die zur Verfügung stehenden Möglichkeiten oft zu einer Überbenutzung dieser Werkzeuge. Steven Webster, ein leitender Mitarbeiter für RIA-Anwendungen bei Adobe, beschreibt diesen Umstand mit der Redewendung *„wenn alles was du hast ein Hammer ist, sieht alles aus wie ein Nagel“* [WT08] recht treffend. Es ist aber unabdingbar zu wissen, warum ein Softwaremuster oder gar ein gesamtes AF eingesetzt werden soll, und ob es wirklich notwendig ist, diese zu implementieren. Dazu sollten die Anforderungen an ein Projekt ebenso vordefiniert sein wie die Eigenschaften des angestrebten AF, um abschätzen zu können, in wie weit das AF die SA im Hinblick auf die speziellen Projekteigenschaften unterstützen kann. Einige Eigenschaften eines AF sind für paralleles Arbeiten konzipiert mit dem Ziel Softwarekomplexität zu reduzieren. Der Einsatz von AF für sehr kleine Projekte mit nur wenigen Entwicklern, ist demnach nicht zwingend erforderlich und kann Softwarekomplexität sogar erhöhen. Im Idealfall sollten AF also eingesetzt werden wenn ein AF eine passende Grundstruktur für eine spezielle Softwareanforderung bietet.

# 3 Flex-Architektur-Frameworks

Dieses Kapital führt die im ersten Kapitel vorgestellte RIA-Technologie Flex mit dem im zweiten Kapitel erarbeiteten Mitteln für gute Softwarearchitektur zusammen. Zeigte Kapitel zwei allgemeine Vorteile von Mustern bzw. Architektur-Frameworks, wird in diesem Kapitel auf Aufbau und Funktionalität konkreter Flex-Architektur-Frameworks eingegangen. Da der Flex-Gemeinschaft in der jüngeren Vergangenheit sehr viele Flex-Architektur-Frameworks vorgestellt wurden, beschränkt sich diese Diplomarbeit auf die derzeit Präsentesten. Die Auswahl wurde dabei analog zu Yakov Fain, Anatole Tartakovsky und Victor Rasputnis [FTR10] getroffen, welche die Flex-Architektur-Frameworks Cairngorm, PureMVC und MATE favorisieren. Neben diesen drei AF wurde bei der letzten MAX-Konferenz [Coe09] das AF Swiz besprochen, welches abschließend zusammen mit anderen Flex-Architektur-Frameworks in Kurzform vorgestellt wird.

## 3.1 Cairngorm

Cairngorm ist eines der ältesten und bekanntesten AF für Flash/Flex RIA-Anwendungen und wurde von Steven Webster und Alistair McLeod während der Tätigkeit in ihrer Firma iterration::two erstellt, bevor diese von Macromedia und später Adobe übernommen wurden. Webster und Alistair erkannten, dass viele SA-Herausforderungen bereits in der J2EE Entwicklung gelöst und umgesetzt wurden und dass diese Ansätze auf RIA-Anwendungen übertragbar sind [WT08]. Daher ist es nicht verwunderlich, dass sich die verwendeten Muster und Herangehensweisen des Cairngorm-AF an der J2EE (Java 2 Enterprise Edition) Architektur[1] orientieren. Bei der MAX Konferenz 2004 wurde Cairngorm offizielles Open-Source-Projekt und liegt derzeit in der Version 2.2.1 vor [Wis09]. Das Hauptanliegen von Cairngorm ist nach der Auffassung von Adobe die Aufteilung des Codes nach dem Prinzip des „Separation of Concerns“ (vgl. Abschnitt 2.3.1)

[1] http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

bzw. die Organisation von Komponenten nach Funktionalität und Verantwortlichkeiten. Cairngorm soll so die Software-Wartung, die Teamarbeit, das Debugging, das Arbeiten mit Servern sowie automatische Tests positiv beeinflussen [BSB08]. Das verwendete MVC-Architektur-Muster, welches bereits im Abschnitt 2.4.2 eingeführt wurde, wird zum Erreichen dieser Ziele durch Entwurfs-Muster, wie dem Command-, Singleton- [GHJV95] oder Delegate-Muster [KS09], ergänzt. Die einfachste Form, um auf die Cairngorm Funktionalitäten zugreifen zu können, ist die Cairngorm-SWC[2]-Datei (Cairngorm Version 2.2 ist nur 12kB groß)[3] dem „libs“ Ordner ihrer Flex Anwendung hinzuzufügen.

### 3.1.1 Cairngorm-Aufbau und -Funktion

Cairngorm-Anwendungen werden laut Yakov Fain, Anatole Tartakovsky und Victor Rasputnis [FTR10] in sechs Komponenten unterteilt, deren Verantwortlichkeiten nachfolgend erläutert werden. Abbildung 3.1 verdeutlicht den Zusammenhang zwischen diesen Bausteinen.

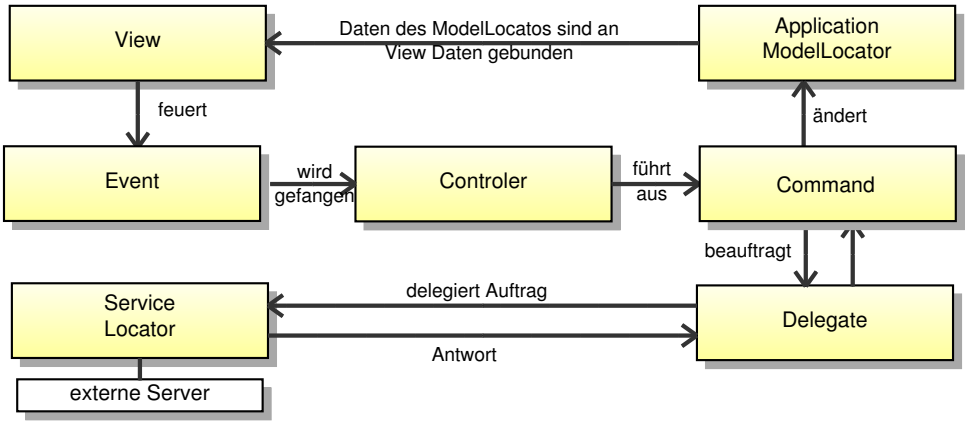


Abbildung 3.1: Aufbau und Funktion Cairngorm

- View: Die View ist das GUI, das heißt, der sichtbare Teil einer Cairngorm-Anwendung in welcher die Nutzerinteraktion stattfindet. Um eine Nutzerinteraktion an andere Cairngorm Bausteine weiterzugeben, ist es erforderlich Cairngorm-Events zu senden. Um Daten der View im Cairngorm-System nutzen zu können, besteht die Möglichkeit, View-Daten mit diesem Event mitzuschicken.

[2]http://livedocs.adobe.com/flex/3/html/help.html?content=compilers_30.html
[3]http://opensource.adobe.com/wiki/display/cairngorm/Downloads

- FrontController (Singleton): Die Cairngorm-Events werden vom FrontController behandelt. Dabei hat der FrontController lediglich die Aufgabe ein Event mit einem Command zu verbinden, führt die angeforderte Aktion aber nicht selbst aus. Vereinfacht kann der FrontController als Schaltwerk zwischen Cairngorm-Events und Cairngorm-Command betrachtet werden.

- Command: Ein Command führt die vom Nutzer angeforderte Aktion aus und verändert Daten im Model. Außerdem kann das Command externe Daten mittels Delegate anfordern und verarbeiten.

- ServiceLocator (Singleton): Im ServiceLocator befinden sich alle externen Datenquellen einer Cairngorm Anwendung. Gesteuert wird der ServiceLocator durch Delegates, welche das Ergebnis nach dem Abfragen der externen Datenquellen zurückgeliefert bekommen.

- Delegate: Ein Delegate beschreibt das Bindeglied zwischen Command und ServiceLocator. Es delegiert die Anfragen eines Commands an die konkrete Datenquelle des ServiceLocators, bzw. reicht die vom ServiceLocator erhalten Daten zurück an das Command. Delegates vermitteln somit zwischen Commands und dem ServiceLocator und helfen bei der Entkopplung von Anwendungslogik und externen Datenquellen.

- ModelLocator (Singleton): Das Model wird von Commands verändert und repräsentiert ausschließlich die Daten des Cairngorm-Systems, sollte jedoch keine Programmlogik enthalten. Die View ist mittels Flex-Databinding an die Daten des Models gebunden, was die Repräsentation der Systemdaten für den Benutzer sicherstellt. Das Model wird in Cairngorm als ModelLocator bezeichnet.

Diese Beschreibung ist eine grundlegende und komprimierte Zusammenfassung der Cairngorm-AF-Bausteine. Eine umfangreiche Beschreibung der AF-Bausteine sowie Prozesse innerhalb des Cairngorm-AF kann im Anhang (vgl. Anhang A.1) nachgelesen werden.

## 3.2 PureMVC

PureMVC wurde von Clifford Hall, Präsident und leitender Softwarearchitekt der Firma Futurescale, entwickelt. Die Firma Futerescale ist offizieller Solution Partner von Adobe und befasst sich vorwiegend mit der Erstellung von Flash, Flex und AIR-Anwendungen. Ihr wohl erfolgreichstes Produkt ist das PureMVC-AF, welches nach der Einführung für Actionscript 2.0, auf mehrere Umgebungen portiert wurde. Für RIA-Browseranwendungen, wie Silverlight oder JavaFX, steht PureMVC ebenso zur Verfügung wie für ColdFusion- oder PHP-Server Umgebungen. Darüber hinaus ist seit der Portierung von PureMVC für Flash Lite, .Net Compact Framework oder J2ME, die Anwendung auf mobilen Plattformen möglich [Hal08a]. Das ist allerdings nur möglich, da PureMVC keine programmiersprachenabhängige Features anwendet, sondern auf grundlegende Muster zurückgreift. Neben dem Architektur-Muster MVC wurde Clifford Hall von dem bekannten Buch der „Gang of Four“ beeinflusst [Hal08b] und verwendet Entwurfsmuster wie das Observer-, Singelton-, Mediator-, Command-, Facade- oder das Proxie-Muster [GHJV95]. Das Hauptanliegen von PureMVC ist die klare Trennung von Model, View und Control nach dem Prinzip des „Separation of Concerns“ (vgl. Abschnitt 2.3.1) sowie eine möglichst lose Kopplung (vgl. Abschnitt 2.3.2) zwischen diesen Komponenten. Dabei soll eine Plattformabhängigkeit vermieden, Komplexität vom Benutzer fern gehalten, Skalierbarkeit erhöht und Zeiten für Wartung und Entwicklung verringert werden [Hal08a]. Die einfachste Form, um dieses open Source AF in einer Flex-Anwendung zu nutzen, ist eine PureMVC swc-Datei dem libs Ordner des Flex-Projektes hinzuzufügen. Dazu stehen für Actionscript 3 zwei grundlegende swc-Datein zur Verfügung: Standard- (engl. Singlecore)[4] und Mehrkern- (engl. Multicore)[5] PureMVC-AF (jeweils ca. 200kB). Eine Mehrkern-PureMVC-Anwendung benötigt man für den Gebrauch mehrerer PureMVC-Module in einer Flex Anwendung. In Abschnitt 4.2.2 sowie im praktischen Kapitel (vgl. Kapitel 5) wird der Vorteil eines Mehrkern-Konzepts eingehender verdeutlicht. Die folgende Einführung beschäftigt sich ausschließlich mit der Standard-Variante, da diese für das Verständnis der Funktionsweise von PureMVC ausreichend ist.

[4] http://trac.puremvc.org/PureMVC_AS3/wiki/Downloads
[5] http://trac.puremvc.org/PureMVC_AS3_MultiCore/wiki/Downloads

### 3.2.1 PureMVC-Aufbau und -Funktion

Clifford Hall unterteilt PureMVC in fünf grundlegenden Akteure: Observer und Notifications, Model und Proxies, View und Mediator, Controller und Command sowie Facade [Hal08b], welche nachfolgend erläutert werden. Dabei wird auf die Kommunikationsmöglichkeiten der einzelnen Elemente eingegangen, da diese für das Verständnis des PureMVC-AF wichtig sind. Da diese Diplomarbeit die Technologie Flex nutzt, bezieht sich die nachfolgende Erklärung ausschließlich auf den Flex spezifischen Einsatz des PureMVC-AF. Abbildung 3.2 verdeutlicht die AF Bausteine des PureMVC-AF.

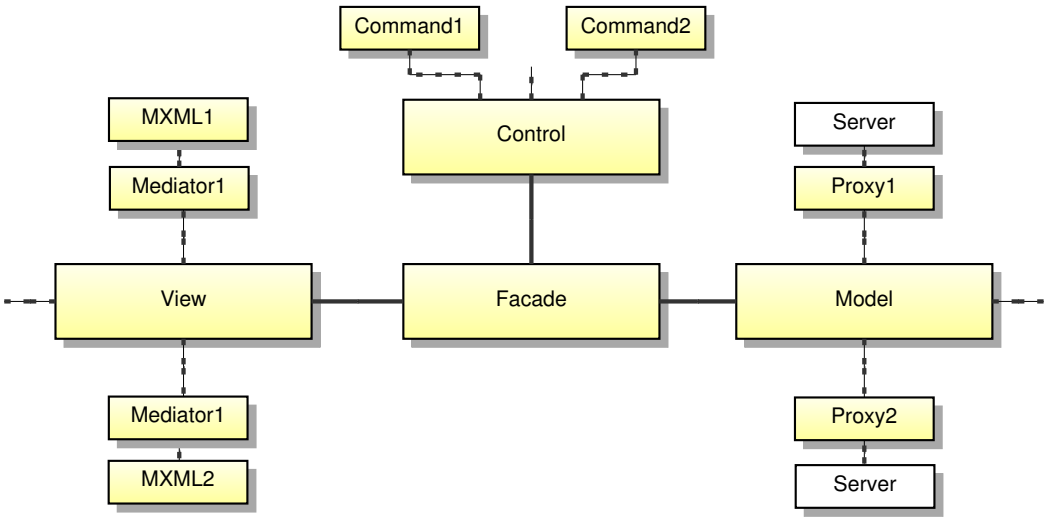


Abbildung 3.2: Aufbau und Funktion PureMVC

- Observer und Notifications: Um PureMVC-Anwendungen auf andere Plattformen portieren zu können, wurde auf den Gebrauch von Flash-Events oder Event-Dispatcher-Klassen verzichtet und stattdessen ein eigenes Observer Model eingeführt, welches PureMVC-Notifications empfangen und senden kann[6]. Damit Notifications vom Command, Mediator, Proxy oder von der Facade gesendet werden können, muss lediglich eine Methode aufgerufen werden. Die Erstellung eines Notification Objekts ist nicht nötig.

- Model (Singleton) und Proxies: Das PureMVC-Model enthält mehreren PureMVC-Proxies. Proxies beinhalten die Datenobjekte des PureMVC-Systems und können

[6] Ausgenommen ist die Kommunikation von MXML-Dateien zu Mediatoren

zur Beschaffung externer Daten genutzt werden (meist unter Verwendung von Delegates). Im Gegensatz zum Cairngorm ModelLocator ist es einem Proxy erlaubt, dem System Funktionen zum Manipulieren seiner Proxy-Daten anzubieten. Ein Proxy kann Notifications senden, jedoch nicht auf gesendete Notifications reagieren.

- View (Singleton) und Mediator: So wie das Model Proxies enthält, enthält die View mehrere Mediatoren. Jedem Mediator ist wiederum eine MXML-Datei zugeordnet. Die Hauptaufgabe des Mediators ist die Vermittlung zwischen dieser MXML-Datei und dem restlichen PureMVC-System. Ein Mediator kann dazu auf Flash-Events der MXML-Datei reagieren und Notifications an das System senden. Ebenso ist es dem Mediator möglich, Notifications aus dem System zu fangen. Dabei werden nur die Notifications in der handleNotification-Funktion des Mediators bearbeitet, die in der listNotificationInterests-Funktion dieses Mediators definiert sind. Ein Mediator darf auf alle öffentlichen Variablen und MXML-Komponenten seiner View zugreifen.

- Controller (Singleton) und Command: Einem Controller können, ähnlich wie in Cairngorm, Notification-Command Verbindungen hinzugefügt werden. Nach Erhalt einer Notification wird das entsprechende Command erstellt und ausgeführt. In PureMVC werden Commands eingesetzt, um systemweite und/oder komplexe Aktionen auszuführen. Dazu können die Commands über eine Facade Mediatoren und Proxies auslesen und bearbeiten, sowie selbst Notifications senden. Dafür stehen in PureMVC zwei verschiedene Command-Typen zur Verfügung. Ein SimpleCommand, analog zum Cairngorm Command, sowie ein MacroCommand, welchem mehrere Commands zur Ausführung übergeben werden können. Das MacroCommand ermöglicht somit die Ausführung mehrerer Commands als Reaktion auf nur eine Notification.

- Facade (Singleton): Die Facade initialisiert die Hauptbausteine Model, View und Controller und ermöglicht einen Zugang zu dessen Funktionalitäten. In einer PureMVC-Anwendung wird die Facade zu einer konkreten Facade abgeleitet. Diese konkrete Facade wird beim Start einer PureMVC-Anwendung generiert und initialisiert die MVC-Komponenten automatisch im Hintergrund. Der Zugang zu diesen Systemteilen erfolgt ausschließlich über die Facade. So werden z.B. die Notification-Command-Verbindungen durch die Facade definiert und im Hintergrund an den

Controller weitergereicht.

In der Beschreibung der AF-Bausteine des PureMVC-AF wurde bereits darauf eingegangen, dass es ein Notification-System in PureMVC gibt und welche AF-Bausteine diese Notification senden und empfangen können. Abbildung 3.3 verdeutlicht zusammenfassend die Möglichkeiten der Notification-Kommunikation.

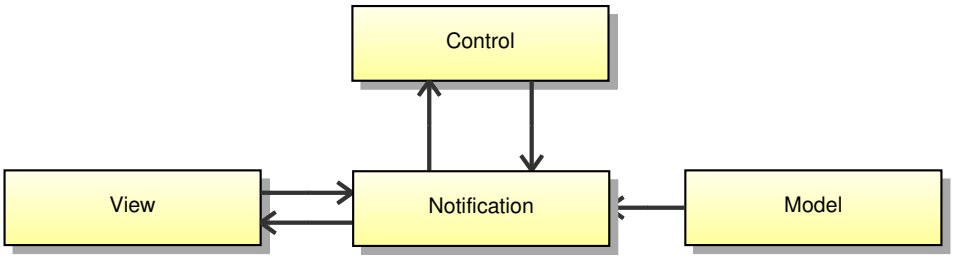


Abbildung 3.3: Notification-Kommunikation PureMVC

Eine weiterführende Beschreibung des PureMVC-AF kann, wie auch beim Cairngorm-AF im Anhang (vgl. Anhang A.2 ) nachgelesen werden. Das empfiehlt sich besonders für Entwickler, welche das PureMVC-AF einsetzen wollen.

## 3.3 MATE

MATE ist ein Flash-Event und MXML-Tag basiertes Flex-AF. Der Name MATE (ausgesprochen „Mah-Te“), ist abgeleitet von einem traditionellen südamerikanischen Getränk. So verwundert es nicht, dass die Entwickler von MATE, Laura Arguello und Nahuel Faronda, aus Argentinien stammen. Laura Arguello, eine Mitbegründerin der RIA Firma AsFusion, stellte MATE bereits Ende 2008 bei der MAX Konferenz in San Jose vor[7]. Offiziell wurde MATE von AsFusion bisher jedoch noch nicht veröffentlicht [FTR10].

MATE ist ein auf Flex-Features ausgerichtetes AF und wurde speziell für den Einsatz in Flex konzipiert. David Tucker, bekannter RIA Software Ingenieur bei Universial Mind, ließ sich deshalb dazu hinreißen, MATE (und Swiz) als „second generation Flex Framework“ zu bezeichen [Tuc09]. MATE nutzt Flex-spezifische Ansätze wie Databinding oder MXML sowie verschiedene Softwaremuster. Das angestrebte Grundkonzept ist, ebenso wie bei Cairngorm oder PureMVC, das MVC-Architekturmuster. Ergänzend können

[7] http://tv.adobe.com/watch/360flex-conference/mate-flex-framework-by-laura-arguello/

Softwaremuster wie Adapter-, Mediator- oder das Presentation-Model-Muster [Wil07] eingesetzt werden. MATE setzt, anders als Cairngorm und PureMVC, das Dependency-Injection-Konzept ein, welches ein Verfahren des IoC-Prinzips ist (vgl. Abschnitt 2.3.2). Laut Laura Arguello ist das Hauptanliegen von MATE, die Nachvollziehbarkeit von Systemreaktionen auf System-Events zu erhöhen, sowie bekannte Flex Problemstellungen, wie z.B. Datenbeschaffung von externen Datenquellen für kleine und Enterprise-Flex-RIA-Anwendungen, effizient zu lösen [Arg09]. Des Weiteren wird ein großes Augenmerk auf die Entkopplung von Systembausteinen gelegt. Die derzeitige Version 0.8.9. steht als swc-Datei zum Download bereit[8] und ist 87 KB klein. Diese Version sowie alle Vorgängerversionen sind „open Source“ und können unter „Google Code“ eingesehen werden[9].

### 3.3.1 MATE-Aufbau und -Funktion

MATE kann im Gegensatz zu Cairngorm oder PureMVC nicht einfach in verschiedene Systembausteine aufgeteilt werden. Genau genommen ist lediglich ein Teil, die EventMap, einer MATE-Anwendung, von MATE abhängig. Dennoch erfolgt in den meisten Beispielen die Speicherung der Systemdaten (Model) in einem so genannten Manager, weshalb dieser in die nachfolgende Beschreibung mit aufgenommen wird. Verschiedene Ansätze findet man jedoch in der Art und Weise, wie diese Daten einer View zur Verfügung gestellt werden. Muster, wie das Adapter-Muster oder das Mediator-Muster, können darüber hinaus vom MATE AF eingesetzt werden. Im „Cafe Townsend“-Beispiel (vgl. Anhang A.3) wird jedoch das Presentation-Model-Muster eingesetzt, weshalb dieses in die nachfolgende Unterteilung mit einbezogen wird. Aus dieser Vorauswahl ergeben sich vier wesentliche Systembausteine des MATE AF, welche nachfolgend in Bezug auf den Einsatz des Presentation-Musters erläutert werden. Abbildung 3.4 zeigt die AF Bausteine des MATE AF sowie die injizierten Abhängigkeiten.

[8] http://mate-framework.googlecode.com/files/Mate_08_9.swc
[9] http://code.google.com/p/mate-framework/source/browse#svn/tags/version_0.8.9/src/com/asfusion/mate

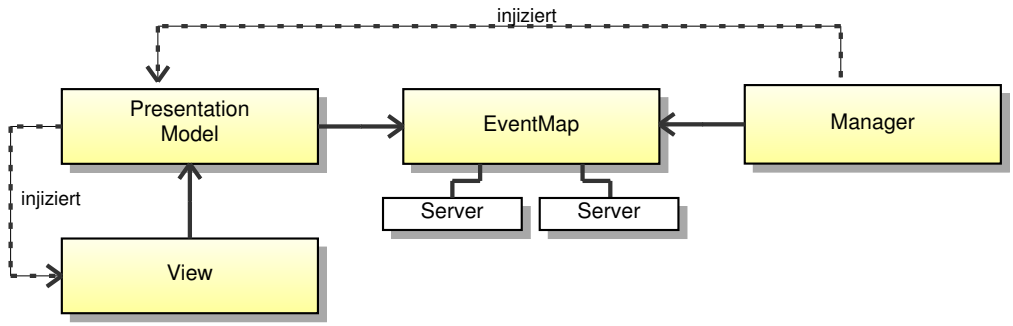


Abbildung 3.4: Aufbau und Funktion MATE

- Presentation Model: Ähnlich wie ein PureMVC-Mediator vermittelt ein Presentation-Model zwischen View und Gesamtsystem. Das jeweilige Presentation-Model stellt dazu einer View die aufbereiteten Modeldaten eines Managers zur Verfügung. Wie auch ein Mediator, definiert das Presentation-Model die Logik seiner View. Anders ist jedoch der Zugriff auf die Presentation-Daten und -Funktionen. Steuert ein Mediator die View vollständig von außen, greift eine View beim Einsatz eines Presentation-Model selbständig auf dessen Funktionen und Daten zu.

- View: Views definieren das GUI der Anwendung und stellen so den sichtbaren Teil der Anwendung dar (analog zum Cairngorm AF). Zu visualisierende Daten werden, wie in der Beschreibung des Presentation-Models angedeutet, direkt aus dem Presentation-Model referenziert und meist an Elemente der View gebunden. Benutzerinteraktionen in einer View werden an das entsprechende Presentation-Model weitergeben, welches seinerseits mit dem Feuern eines oder mehrerer Events reagiert oder die Anfrage intern bearbeiten kann.

- EventMap (Controller): Dieser Systembaustein stellt den eigentlichen MATE-Kern dar. Grundlegend übernimmt die EventMap zwei zentrale Aufgaben. Zum einen fängt sie Events aus dem MATE-System, kommuniziert mit externen Datenquellen und speichert die Ergebnisse in Manager-Objekten. Zum anderen ist die EventMap die Stelle im MATE-System, an der die Injektionen von Abhängigkeiten vorgenommen wird. Beispielsweise injiziert die EventMap Presentation-Models in die jeweilige View. Alle Definitionen in der EventMap werden mittels MATE-MXML-Tags vorgenommen, welche in sich verschachtelt werden können.

- MANAGER (Model): Wie bereits bei der EventMap beschrieben, definieren Manager das Model des Systems. Manager können mehrer Datenobjekte beinhalten, definieren jedoch thematisch zusammengehörige Daten. Manager stellen dem System Funktionen zur Manipulation seiner Daten zur Verfügung. Diese Daten werden meist nicht direkt von einer View referenziert, sondern in aufbereiteter Form von einem oder mehreren Presentation-Models zur Verfügung gestellt.

Analog zu Cairngorm und PureMVC wurde im Rahmen dieser Diplomarbeit ebenfalls ein ausführliches Beispiel erarbeitet. Dieses kann im Anhang (vgl. Anhang A.3) nachgelesen werden.

## 3.4 Weitere Architektur-Frameworks

Nachdem in den bisherigen Abschnitten dieses Kapitels ausführlich auf die Architektur-Frameworks Cairngorm, PureMVC und MATE eingegangen wurde, stellt dieser Abschnitt weitere Flex- bzw. Actionscript-3-Architektur-Frameworks vor. Dabei wird sich auf eine kurze Beschreibung der Kernfunktionalitäten des jeweiligen AF beschränkt, um den Rahmen dieser Arbeit nicht zu sprengen. Außerdem werden wenig verbreitete Frameworks wie Glue, easyMVC, Ruboss, Penne, Model Clue:Flex oder Parsley an dieser Stelle nur genannt und nicht näher erläutert.

### 3.4.1 Swiz

Swiz[10] ist ein AF zum expliziten Einsatz in Adobe Flex. Chris Scott, der Entwickler dieses Frameworks, legte seinen Hauptfokus auf eine möglichst hohe Vereinfachung des Softwareentwicklungsprozesses. Swiz nutzt „Inversion of Control“ sowie „Meta Statements“ und orientiert sich laut Sebastian Martens [Mar08] an Java. Swiz ist einfach und reduziert den geschriebenen Code, was zu einer Erhöhung der Produktivität führen soll.

### 3.4.2 Photomac

Photomac[11] ist ein AF, welches sich stark an der in der Java Enteprise Entwicklung sehr populären OSGi-Technologie orientiert. Ein OSGi Modul, genannt „Bundle“, kann

[10] http://code.google.com/p/swizframework/
[11] http://www.potomacframework.org/

zur Programmlaufzeit dynamisch geladen werden. Ein „Bundle“ enthält Meta-Daten, welche das Modul und dessen Abhängigkeiten beschreiben. Eine OSGi-Anwendung ist im Grunde eine Sammlung solcher „Bundles“, die wiederum miteinander verbunden werden können. Chris Gross entwickelte das Photomac-AF und versucht damit, die Idee des OSGi in die Flex-Welt zu übertragen. Besonderen Wert legt er auf die hohe Skalierbarkeit der Anwendung sowie auf die Wiederverwendbarkeit der Module. Neben dem ActionScript/MXML-Photomac-Code steht ebenso ein Flex-Builder-Plugin zur Verfügung.

### 3.4.3 Robotlegs

Robotlegs[12] ist ein kleines einfaches Actionscript-3-AF. Das Hauptanliegen besteht in der Verbindung verschiedener Objekte nach dem Prinzip der losen Kopplung. Das geschieht durch die Anwendung von „dependency injection“ und Meta-Tags weitgehend automatisiert. Hierdurch sollen der zu schreibenden Code und die aufzuwendende Entwicklungszeit verkürzt werden. Indem Singletons und statischen Variablen vermieden werden, soll der Code zusätzlich besser testbar sein.

### 3.4.4 Fireclay

Fireclay[13] wurde von Shashank Tiwari entwickelt [THEM08] und bereits 2008 in seinem Buch vorgestellt. Als Basis nutzt Shashank Tiwari das MVC-Architektur-Muster, welches er mit einem Service- und einem Eventbus-Baustein erweitert hat. Der Eventbus baut auf das Flex-Event-System auf und bildet die Kommunikationsgrundlage des Fireclay-AF. Ein großes Anliegen von Fireclay ist, neben loser Kopplung und hoher Skalierbarkeit, die möglichst komfortable Anbindung verschiedener externer Datenquellen.

### 3.4.5 Spring Actionscript

Das von Christophe Herreman entwickelte „Inversion of Control“-Framework Spring-Actionscript[14] war ehemals unter dem Namen Prana bekannt. Wie der Name vermuten lässt, ist die Vorlage bzw. die Basis bei Java-Spring zu finden. Mit Spring-Actionscript

[12] http://www.robotlegs.org/
[13] http://code.google.com/p/fireclay/
[14] http://www.springactionscript.org/

ist es möglich, Java-Spring-Mechanismen auch in Flex anzuwenden [Hei08]. Spring-Actionscript ist nach Abschnitt 2.5.1 nicht nur ein AF, sondern bietet zusätzlich Erweiterungen für PureMVC und Cairngorm.

### 3.4.6 Guasax

Guasax[15] basiert auf dem MVC-Architekturmuster und soll insbesondere die Erstellung von skalierbaren und übersichtlichen Flex-Anwendungen unterstützen. Der zu schreibende Code ist Framework-unabhängig, da alle Aktionen in einer XML Datei ausgelagert und mittels „Inversion of Control" ausgeführt werden. Die entstehenden Komponenten sind deshalb auch in Nicht-Guasax-Projekten wiederverwendbar.

[15] http://www.guasax.com/guasax/web/en/index.php

# 4 Vergleich Flex-Architektur-Frameworks

Nachdem in Kapitel 3 die Architektur-Frameworks Cairngorm, PureMVC und MATE ausführlich hinsichtlich Aufbau und Funktion beschrieben wurden, werden diese drei AF nachfolgend unter Anwendung der Kriterien Zugänglichkeit, Entwicklungseffizienz und Wartbarkeit miteinander verglichen. Für die Bewertung der zu vergleichenden AF-Basisversionen wird ein einfaches Punktsystem eingeführt. Jedem Architektur-Framework wird für jedes Kriterium eine Punktezahl zwischen 0 (schlecht) und 5 (sehr gut) vergeben. Damit der Leser dieser Arbeit einfach und übersichtlich nachvollziehen kann, was die Vor- und Nachteile der drei AF sind, gibt Abschnitt 4.4 eine tabellarische Auswertung der AF hinsichtlich der untersuchten Kriterien.

## 4.1 Zugänglichkeit

Unter der Zugänglichkeit eines AF wird in dieser Diplomarbeit verstanden, wie komplex ein AF und somit wie hoch bzw. niedrig der Einarbeitungsaufwand in das AF ist. Der Einarbeitungsaufwand kann durch Hilfsmittel wie eine gute Dokumentation und Tutorials, einführende Beispielprogramme oder eine große und aktive Community stark vereinfacht werden. Grundlegend ist ein AF dann einfach Zugänglich, wenn es einfach strukturiert ist und/oder eine hohe Qualität der Hilfsmittel den Einstig in das AF erleichtern.

### 4.1.1 Cairngorm

Wie bereits in Abschnitt 3.1.1 dargestellt, arbeitet eine Cairngorm Anwendung beispielsweise nach einer Benutzerinteraktion, immer nach demselben Schema. Dank dieser immer wiederkehrenden, vom AF vorgegebenen Abfolge von Bearbeitungsschritten, ist

Cairngorm wenig komplex. Dennoch ist der Einarbeitungsaufwand gerade für unerfahrene Entwickler nicht zu unterschätzen. Hilfe und Unterstützung erwartet man auf der offiziellen Adobe-Open-Source-Seite[1], die sich Cairngorm mit anderen Projekten teilt. Doch erst circa ein Jahr nach Veröffentlichung der aktuellen Cairngorm-Version im Jahr 2007, stellte Adobe eine umfassende und nachvollziehbare Dokumentation [BSB08] zur Verfügung. Eine weitere Dokumentation wird dem Cairngorm-Nutzer von Steven Webster und Leon Tanner vom Adobe-Consulting-Team [WT08] zur Verfügung gestellt. Nachteilig ist jedoch, dass Beispielanwendungen oder Tutorials auf dieser Seite nicht zu finden sind. Auch das offizielle Forum[2] für Cairngorm wird von der großen Community recht wenig genutzt. Auf Grund dieser eher mangelhaften Unterstützung auf der offiziellen Webseite hat sich die Community damit geholfen, Beispielprogramme, Tutorials sowie aktive Foren selbst anzubieten. Das wohl erfolgreichste Tutorial liefert David Tucker mit seinen exzellenten Cairngorm Videos [Tuc08]. Die Community-Seite „CairngormDocs“[3] enthält weiterführend sehr gute Beispielprogramme zum Einstieg in das Cairngorm-AF. Zusammenfassend lässt sich sagen, dass die Zugänglichkeit trotz der relativ geringen Komplexität des Cairngorm-AF nur auf Grund der Eigeninitiative der Community als „gut“ bezeichnet werden kann (vgl. Abbildung 4.1).

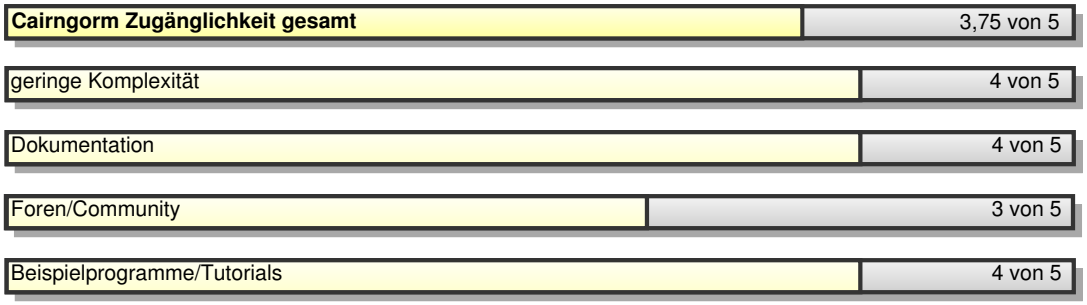


Abbildung 4.1: Zugänglichkeit Cairngorm

### 4.1.2 PureMVC

PureMVC-Systemabläufe laufen im Vergleich zu Cairngorm deutlich weniger linear ab. Besonders während des Einarbeitungsprozesses in das AF stellt sich daher häufig die Frage, welche Aufgaben die jeweiligen AF-Bausteine übernehmen. Bedenkt man,

[1] http://opensource.adobe.com/wiki/display/cairngorm/Cairngorm
[2] http://forums.adobe.com/community/opensource/cairngorm
[3] http://www.cairngormdocs.org/

dass diese Bausteine in der Lage sind, untereinander zu kommunizieren, wird die hohe Komplexität des PureMVC-AF deutlich. Auch für erfahrene Softwareentwickler ist der Einarbeitungsaufwand daher als relativ hoch einzuordnen. Erfreulicherweise gibt es eine ausgezeichnete PureMVC-Informationsseite[4] für nahezu alle Fragen. Auf dieser Webseite findet man neben qualitativ hochwertigen Dokumentationen, zum Teil sogar in deutscher Sprache, Framework-Übersichten sowie UML Diagramme. Daneben gibt es mehrere Beispielprogramme in verschiedenen Programmiersprachen. Die große PureMVC-Community gibt im Webseiten-Forum gern ihre Erfahrungen weiter und diskutiert über mögliche Erweiterungen. Fasst man all diese Punkte zusammen, ist es trotz der relativ hohen Einstiegshürde nicht zu erwarten, dass sich ein Programmierer allein gelassen fühlt. Daraus resultiert die insgesamt „gute“ Bewertung der PureMVC-Zugänglichkeit (vgl. Abbildung 4.2).

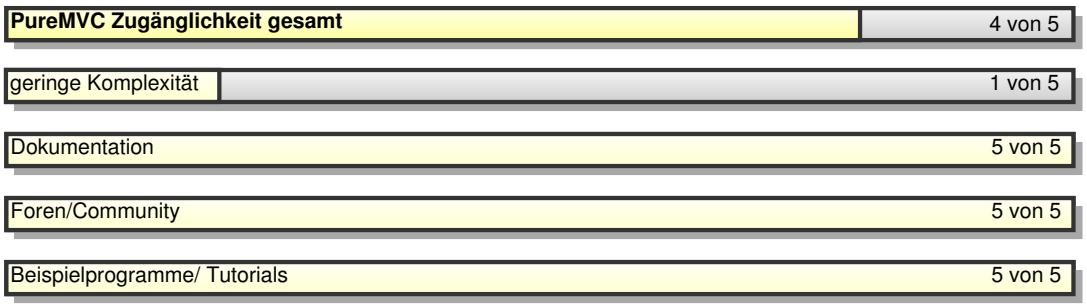


Abbildung 4.2: Zugänglichkeit PureMVC

### 4.1.3 MATE

Im Gegensatz zu PureMVC und Cairngorm ist die Komplexität einer MATE-Anwendung nicht gleichmäßig auf die AF-Bausteine verteilt. MATE-unabhängige Bausteine, wie Manager oder Presentation-Models, sind oftmals viel weniger komplex als die zentrale MATE-EventMap. Bedenkt man, dass diese EventMap auf alle relevanten Events im System mit beispielsweise Serveranfragen oder Datenmanipulation reagiert und zusätzlich das Injizieren von Objektabhängigkeiten übernimmt, wird der Umfang deutlich. Überschaubar ist dieser enorme Umfang lediglich durch das konsequente Einsetzen von MATE-MXML-Komponenten in der EventMap. Da man diese MXML-Komponenten einfach einsetzen kann und der entstehende „Code“ übersichtlich ist, kann die Gesamt-

[4] http://puremvc.org/

komplexität einer MATE-Anwendung als mittel eingestuft werden. Der Einstieg in dieses AF wird durch die sehr guten Dokumentationen auf der MATE-Webseite[5] erleichtert. Außerdem stehen auf dieser Webseite aussagekräftige Diagramme sowie ein Anfänger-Tutorial zur Verfügung. Darüber hinaus gibt es verschiedene Beispiele und eine sehr aktive Community sowie ein sehr aktuelles Forum. Die Qualität dieser Hilfsmittel ist derart hoch, dass die Zugänglichkeit in Verbindung mit der ohnehin nur mittleren Komplexität als „sehr gut“ eingeschätzt werden kann (vgl. Abbildung 4.3).

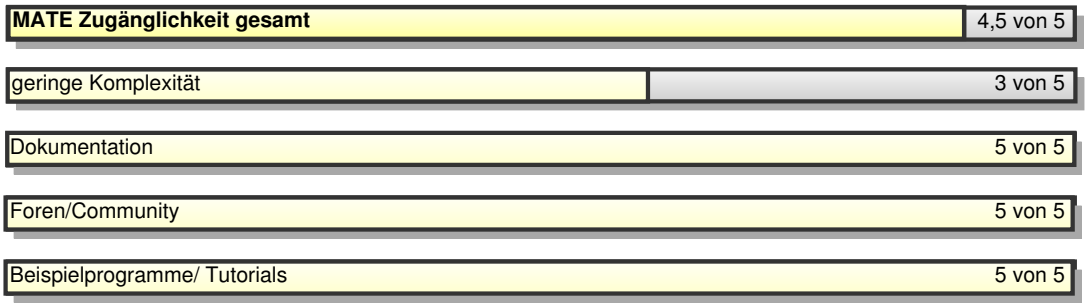


Abbildung 4.3: Zugänglichkeit MATE

## 4.2 Entwicklungseffizienz

Nachdem in Abschnitt 4.1 die Zugänglichkeit der AF betrachtet wurde, beschäftigt sich dieser Abschnitt mit der AF-Entwicklungseffizienz während der Implementierung von Programmanforderungen. Eine hohe Entwicklungseffizienz ist dann gegeben, wenn sich die Programmanforderungen mittels eines AF zügig und direkt umsetzen lassen. Die zu schreibende Code-Menge ist dabei ebenso ein Kriterium wie die intuitive Einordnung des Codes in den „richtigen“ AF-Baustein. Bietet das AF eine komfortable Möglichkeit, auf externe Datenquellen zuzugreifen, sind viele AF-Erweiterung vorhanden oder können AF-Bausteine bzw. ganze AF-Module wieder verwendet werden, so wirkt sich das ebenfalls positiv auf die Entwicklungseffizienz aus.

[5] http://mate.asfusion.com/

### 4.2.1 Cairngorm

Bei Cairngorm fällt besonders die hohe Code-Menge auf, welche geschrieben werden muss, um Programmanforderungen umzusetzen. Soll im Cairngorm-System beispielsweise auf eine Nutzerinteraktion reagiert werden, muss neben einer Cairngorm-Event-Klasse eine Command-Klasse sowie ein Eintrag im Front-Controler erstellt werden. Im „Café Townsend“ Beispiel (vgl. Anhang A.1) mussten beispielsweise 25 Klassen erstellt werden. Ein ähnlich hoher Schreibaufwand zeigt sich bei der Verwendung von Model- oder Service-Locator. Jeder AF-Baustein der diese verwenden möchte, muss die Instanz des jeweiligen Singletons immer wieder vom Cairngorm-System erfragen. Dies hört sich zunächst nicht nachteilig an, bedenkt man aber, dass jede View und jedes Command Modeldaten aus dem ModelLocator benötigt und der ServiceLocator in jedem Delegate genutzt wird, wird die hohe Code-Redundanz deutlich. Die große Code-Menge führt wiederum zu einer erhöhten Entwicklungszeit und damit zu einer geringen Entwicklungseffizienz. Diese Problematik versucht Adobe mittels „Cairngorm Code Generator Plugin“[6] für den „Flex Builder“ zu verringern. Das gelingt auf Grund des eingeschränkten Funktionsumfangs des Generators aber nur bedingt.

Der Umgang mit externen Datenquellen kann im Gegensatz zur Code-Menge jedoch als effizient angesehen werden. So ist es auffallend einfach, verschiedenste Datenquellen, wie z.B. Web Services, ColdFusion-Komponenten oder http-Services, in ein Cairngorm Programm einzubinden. Der ServiceLocator verwaltet alle Services und stellt sie den Delegates in immer gleicher Art und Weise zur Verfügung. Dieses standardisierte Vorgehen erleichtert und beschleunigt das Arbeiten mit externen Datenquellen und verbessert daher die Entwicklungseffizienz.

Die Aufgabenbereiche aller Cairngorm-Bausteine sind klar voneinander getrennt. Die Programmlogik wird beispielsweise immer mittels Commands implementiert, alle externen Datenquellen hingegen im ServiceLocator. Daher ist stets ersichtlich, wo entsprechender Code erstellt werden muss. Dies führt allerdings nur solange zu einer erhöhten Entwicklungseffizienz, wie diese strikten Vorgaben die Implementierung von Features nicht behindern. Das Cairngorm-AF stößt beispielsweise an seine Grenzen, wenn zur Programmlaufzeit eine View mittels eines Commands erweitert werden soll. Dieser Fall tritt immer dann ein, wenn ein Command zur Programmlaufzeit eine GUI-Komponente wie z.B. einen Button erstellen soll. Obwohl es Aufgabe eines Commands ist, alle logischen Prozesse des

[6] http://opensource.adobe.com/wiki/display/cairngorm/Plugin

AF zu steuern, hat das Command keinen direkten Zugriff auf die View. Solche Probleme sollten im Sinne einer guten SA sauber gelöst werden, was die Umsetzung dieser sehr einfachen Anforderung wiederum sehr zeitaufwändig gestaltet.

Eine Lösung für mehrere solcher Cairngorm-Implementierungsprobleme kommt aus der Cairngorm-Community und nicht von Adobe selbst. Die Firma Universal Mind veränderte Teile des Cairngorm-AF und fügte mehrere Erweiterungen hinzu, wodurch die Arbeit mit Cairngorm vereinfacht und damit eine höhere Produktivität bzw. Entwicklungseffizienz erreicht wird.

Doch trotz dieser Erweiterungen gibt es beim Cairngorm-AF keinen erkennbaren Ansatz zur Wiederverwendung einzelner AF-Bausteinen. Dies resultiert vor allem aus der recht starken Kopplung zwischen den verschiedenen Bausteinen. So „kennen“ zum Beispiel alle Views und alle Commands Inhalte bzw. Daten des ModelLocator, was eine Wiederverwendung auf Grund hoher Abhängigkeiten sehr ineffektiv gestaltet. Darüber hinaus besteht keine Möglichkeit, gesamte Cairngorm-Anwendungen als Modul in einer anderen Cairngorm-Anwendung wiederzuverwenden.

Zusammenfassend lässt sich sagen, dass die Entwicklungseffizienz lediglich als mittelmäßig beschrieben werden kann (vgl. Abbildung 4.4). Die enorme Code-Menge und die aus den strikten Vorgaben des Cairngorm-AF resultierenden Implementierungsprobleme wiegen schwerer als die gute Anbindung von externen Daten oder die guten Erweiterungen durch die Community.

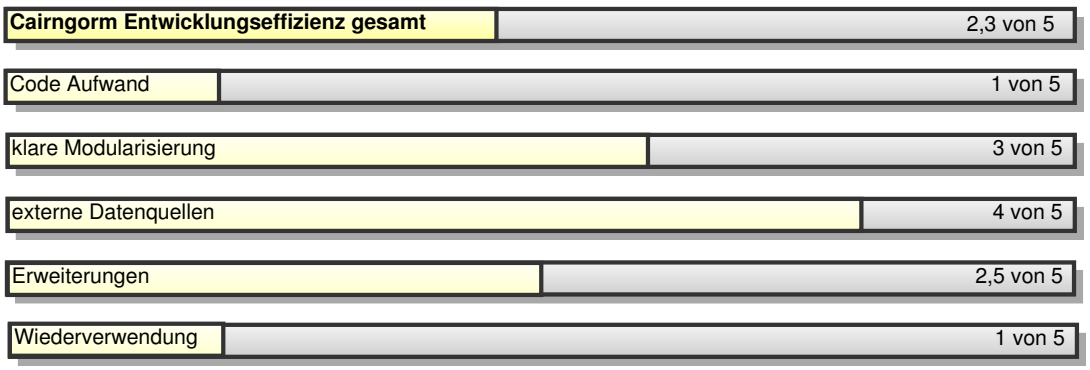


Abbildung 4.4: Entwicklungseffizienz Cairngorm

### 4.2.2 PureMVC

Nutzt man das PureMVC-AF muss lediglich beim Initialisierungsvorgang ein recht hoher Aufwand betrieben werden. Nachdem alle Mediatoren und Proxies im System registriert sind, ist der zu schreibende AF-Code relativ gering. Dies resultiert daraus, da beispielsweise Mediatoren, View, Logik und Proxies Funktionen zur Datenmanipulation enthalten können, wodurch, im Gegensatz zum Cairngorm-AF, nicht zwangsläufig für jede Programmfunktionalität eine Command-Klasse erstellt werden muss. Das reduziert vor allem die im Projekt enthaltene Klassenanzahl (vgl. Anhang A.2: 16 Klassen mussten angelegt werden) sowie den Implementierungsaufwand. Aber auch die Kommunikation zwischen den AF-Bausteinen wurde sehr komfortabel und effizient gestaltet. Wie in Anhang A.2 nachzuvollziehen ist, muss zum Senden einer Notification diese nicht selber erstellt werden. Das bedeutet, dass keine Notification-Klasse selber programmiert werden muss. Es genügt vielmehr, eine bereits im AF implementierte Funktion aufzurufen, welche diese Aufgabe für den Entwickler übernimmt. Unnötige Komplexität sowie redundanter Code werden somit weitgehend vermieden. Zusätzlich wird die Komplexität im PureMVC-AF durch die PureMVC-Facade verringert. Wie bereits bei der Funktionsbeschreibung dieses AF aufgezeigt (vgl. Abschnitt 3.2.1), können mit einer Facade interne Systemprozesse zentralisiert gesteuert werden, ohne die internen AF-Gegebenheiten kennen zu müssen. Dazu muss die Facade allerdings nicht, wie z.B. der ModelLocator in Cairngorm, jedes mal neu angelegt werden. Jede Basisklasse der AF-Bausteine enthält bereits eine Facade-Referenz, die direkt genutzt werden kann. Diese Herangehensweise reduziert die Entwicklungszeit, was wiederum zu einer höheren Entwicklungseffizienz führt.

Im Gegensatz zum ServiceLocator von Cairngorm, verfügt das PureMVC-AF über keinen zentralen AF-Baustein zur Verwaltung aller externen Datenquellen. Vielmehr verwaltet und erstellt ein Proxy seine Datenquelle, meist unter Verwendung eines Delegates, selbstständig. Wenn zwei unterschiedliche Proxies allerdings die gleiche externe Datenquelle benötigen, müssen diese Datenquellen doppelt angelegt werden. Dies ist im Vergleich zum Cairngorm-AF deutlich aufwendiger. Der Vorteil dieser Herangehensweise zeigt sich aber in eben dieser Eigenständigkeit der Proxies. Da jeder Proxy vom restlichen System entkoppelt ist, gestaltet sich die Wiederverwendung in einem anderen Projekt als recht einfach. Eine Wiederverwendung von AF-Bausteinen führt dabei zu einer erhöhten Entwicklungseffizienz.

Aus der hohen Flexibilität des PureMVC-AF resultieren jedoch auch Unklarheiten und

damit Nachteile. Oftmals ist es nicht einfach, den „richtigen“ AF-Baustein für den zu implementierenden Code zu finden. Die Entscheidung, ab welchen Umfang Mediator-Logik bzw. Proxy-Logik in Commands ausgelagert werden, ist zum Beispiel vom Entwickler selbst zu treffen. Das führt besonders bei unerfahrenen Entwicklern zu länger andauernden Abschätzungen und damit zu einer reduzierten Entwicklungseffizienz.

Besonderes effizient können Nutzer unter Verwendung der in Abschnitt 3.2 genannten Mehrkernvariante entwickeln. Die Anwendung dieser PureMVC-Variante ermöglicht den Einsatz mehrer PureMVC-Kerne in einer Anwendung. Diese Kerne beinhalten jeweils eigene MVC-Bausteine sowie eine eigene Facade und werden Module genannt. Jedes Modul ist damit vollkommen unabhängig von einem anderen Modul, was sie besonders gut wieder verwendbar macht. Dies zeigt die Relevanz für die Entwicklungseffizienz. Der Programmieraufwand für die Erstellung dieser Module kann, im Vergleich zur immensen Effizienzsteigerung durch Wiederverwertung, nahezu vernachlässigt werden.

Um die Entwicklungseffizienz noch weiter zu erhöhen, stehen für PureMVC mehrere Erweiterungen zur Verfügung. Besonders das asynchrone Command[7] oder die Pipe&Filter-Erweiterung[8] für die Mehrkernvariante helfen, die Entwicklungszeit gering zu halten und so eine Effizienzsteigerung zu erzielen.

Zusammenfassend ist der Umgang mit dem PureMVC AF gerade für unerfahrene Softwareentwickler nicht einfach. Auch der Aufwand zur Initialisierung des AF ist deutlich höher als beispielsweise beim Cairngorm-AF. Dennoch ist das PureMVC-AF als sehr effektiv einzuordnen, was aus der geringen Code-Menge, welche zur Umsetzung von Programmanforderungen benötigt wird, und dem hohen Grad der Wiederverwertung von AF-Bausteinen bzw. ganzer PureMVC-Module, resultiert (vgl. Abbildung 4.5).

[7] http://trac.puremvc.org/Utility_AS3_AsyncCommand
[8] http://trac.puremvc.org/Utility_AS3_MultiCore_Pipes

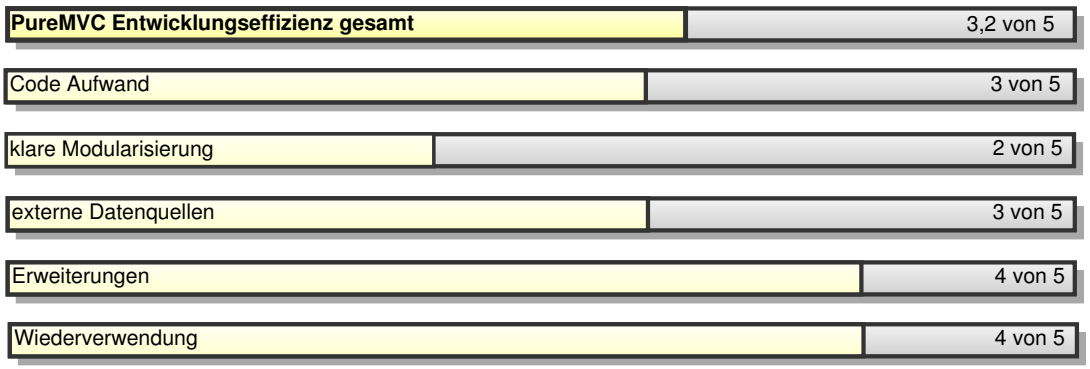


Abbildung 4.5: Entwicklungseffizienz PureMVC

### 4.2.3 MATE

Wie bereits bei dem Kriterium Zugänglichkeit beschrieben (vgl. Abschnitt 4.1.3), zeichnen sich MATE-Anwendungen in erster Linie durch den Einsatz der EventMap aus. Dennoch ist es üblich, neben dem Manager AF-Baustein auch Presentation-Models einzusetzen. Die nachfolgende Erläuterung beschreibt, analog zum „Café Townsend“- Beispiel (vgl. Anhang A.3), die Entwicklungseffizienz des MATE-AF unter Einbeziehung dieser Bausteine.

Besonders auffällig beim MATE-AF ist die besonders geringe Code-Menge, die erforderlich ist, um Programmanforderung umzusetzen bzw. das AF zu initialisieren. Die einzigen Klassen, die zum Beginn erstellt und „verbunden“ werden müssen, sind die Manager- und Presentation-Model-Klassen. Diese sind aber wenig komplex und daher sehr schnell und unkompliziert zu erstellen.

Um Benutzerinteraktionen an das System, genauer die EventMap zu übermitteln, ist es theoretisch ausreichend, ein standard Flash/Flex-Event abzusenden. In der Praxis zeigt sich jedoch oft die Notwendigkeit eigener Event-Klassen, wodurch die benötigte Entwicklungszeit und Klassenmenge leicht erhöht wird. Im „Café Townsend“-Beispiel wurden daher zum Beispiel 19 Klassen benötigt (vgl. Anhang A.3). Besonders effektiv ist die Verarbeitung der Events in der MATE-EventMap. Hierbei ist es selten erforderlich, eigene Klassen zu erstellen. Die verfügbaren MATE-MXML-Komponenten sind derart umfangreich, dass mit ihnen nahezu alle Anforderungen umgesetzt werden können. Dadurch lassen sich Implementierungen sehr schnell und komfortabel umsetzen, was die Entwicklungszeit und die benötigte Klassenanzahl reduziert und somit zu einer sehr hohen Entwicklungseffizienz führt. Sollte es aber dennoch notwendig sein, eigene MXML-Komponenten zu erstellen, steigt der Zeitaufwand stark an, da, wie bei Cairngorm, interne

Prozesse des AF berücksichtigt werden müssen. Da dieser Fall aber praktisch nie eintritt, wird er in der Bewertung nicht mit berücksichtigt.

Hohe Effektivität zeigt sich ebenfalls im Umgang mit den externen Datenquellen. Diese Datenquellen werden meist, ähnlich wie in Cairngorm, in einer zentralen MXML-Komponente und nur in kleinen Projekten in der EventMap selbst, gehalten. Besonders komfortabel zeigt sich der Einsatz dieser Datenquellen, wofür die MATE-MXML-Tags RemoteObjectInvoker, HTTPServiceInvoker und WebserviceInvoker sowie Responder-Tags für Ergebnisse und Fehlerfälle zur Verfügung[9] stehen. Auf eine weitere Beschreibung der Vorgehensweise soll an dieser Stelle verzichtet werden, da diese im Anhang (vgl. Anhang A.3) nachzulesen ist. Festzuhalten bleibt die hohe Entwicklungseffizienz beim Umgang mit externen Datenquellen, welche vor allem aus der Zeitersparnis während der Implementierung resultiert.

Da bereits in der MATE-Basisversion umfangreiche Möglichkeiten zur Umsetzung von Programm Funktionalitäten zur Verfügung stehen, gibt es derzeit nur drei offizielle MXML-Komponenten Erweiterungen, welche den grundlegenden Umgang mit dem MATE-AF jedoch nicht wesentlich vereinfachen und somit die Entwicklungseffizienz nur im Ausnahmefall erhöhen.

Trotz der vielfältigen Möglichkeiten beim Einsatz des MATE-AF, kommt es kaum zu Unklarheiten bezüglich der Einordnung von Code in entsprechende AF-Bausteine. Der Programmablauf einer MATE-Anwendung wird, bedingt durch den breiten Aufgabenbereich der EventMap, ohnehin nahezu ausschließlich in dieser definiert. Auch die Unterscheidung zwischen Managern und Presentation-Models ist nach dem Prinzip von SoC (vgl. Abschnitt 2.3.1) klar definiert. Damit ist es für den Nutzer nach der Einarbeitung fast niemals nötig, Überlegungen zur Einordnung von Code in den „richtigen“ AF-Baustein vorzunehmen. Diese Tatsache ist, gerade im Vergleich zum PureMVC-AF, als effektiv zu bewerten.

Ein Kritikpunkt bei der Verwendung des MATE-AF besteht in den vergleichsweise geringen Möglichkeiten zur Wiederverwendung. Obwohl die Abhängigkeit der einzelnen AF-Bausteine dank Injektion von Abhängigkeiten sehr gering ist, sind Manager, Presentation-Models und die EventMap derart anwendungsspezifisch, dass eine Wiederverwendung kaum möglich ist. Durch die konsequente Anwendung von lokalen EventMaps können zwar wieder verwendbare Module gebildet werden, dennoch ist dieser Ansatz

[9] http://mate.asfusion.com/page/documentation/tags/services

gerade auf Grund der mangelhaften Möglichkeiten zur Kommunikation zwischen diesen Modulen zu vernachlässigen.

Zusammengefasst resultiert die hohe Entwicklungseffizienz somit vor allem aus der geringen Code-Menge und der intuitiv handhabbaren MATE-MXML Tags in der EventMap bzw. aus der daraus resultierenden Zeiteffizienz. Lediglich die schlechten Möglichkeiten zur Modulbildung, ihrer Wiederverwendung und der Kommunikation fallen negativ auf (vgl. Abbildung 4.6).

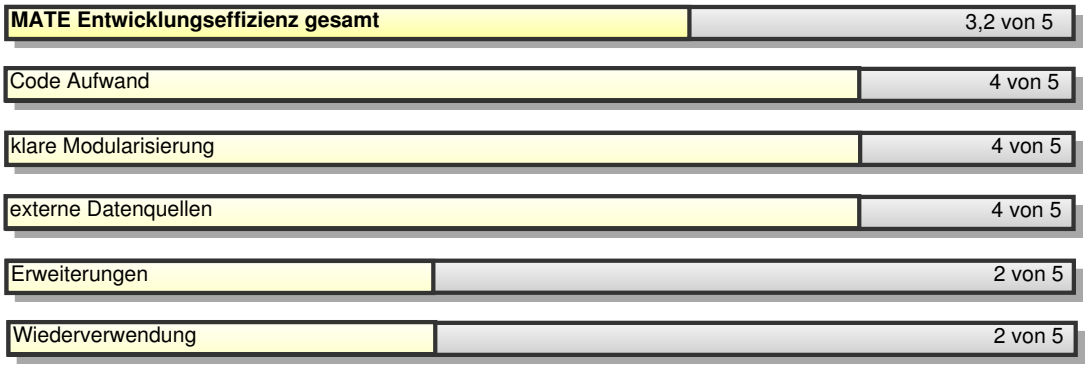


Abbildung 4.6: Entwicklungseffizienz MATE

## 4.3 Wartbarkeit

Nachdem in Abschnitt 4.1 die Zugänglichkeit und in Abschnitt 4.2 die Effizienz während des Entwickelungsprozesses mit dem jeweiligen AF aufgezeigt wurden, befasst sich der folgende Abschnitt mit der Wartbarkeit bestehender AF-Anwendungen. Dazu wird das Kriterium Wartbarkeit in folgende drei Unterkriterien unterteilt: Lesbarkeit bzw. Nachvollziehbarkeit sowie Veränderbarkeit und Testbarkeit (Unit Test, Debuggen).

Gut lesbare Programme, welche leicht verändert und getestet werden können, werden als besonders gut wartbar eingeordnet. Ein großer Vorteil für die Wartbarkeit einer Anwendung, ist die Erweiterbarkeit. Cairngorm, PureMVC und MATE bieten allesamt die Möglichkeit sehr gut erweiterbare Anwendung zu erstellen, weshalb in der folgenden Bewertung von einem Vergleich der Erweiterbarkeit abgesehen wird.

### 4.3.1 Cairngorm

Auf Grund der klaren Modularisierung und der immer gleichen Abläufe in einer Cairngorm-Anwendung, ist es für Cairngorm-Entwickler, welche ein bestehendes Cairngorm-Projekt erstmalig bearbeiten, relativ einfach, bestehende Funktionalitäten nachzuvollziehen. Besonders gut lesbar ist Cairngorm-Code, wenn, ausgehend von Benutzerinteraktionen, Command-Funktionalitäten und Model-Änderungen, untersucht werden müssen. Im Gegensatz dazu ist es schwieriger, ausgehend vom ModelLocator festzustellen, welche Commands bestimmte Model-Daten verändern. Dies ist jedoch immer dann wichtig und auch notwendig, wenn man herausfinden möchte, welche Commands für Model-Anpassungen und damit View-Veränderungen verantwortlich sind. Doch trotz dieser Schwierigkeiten sind Cairngorm-Anwendung gut lesbar.

Durch die bereits in Abschnitt 4.2.1 thematisierte enge Kopplung der Views bzw. der Commands an den ModelLocator, ist eine Veränderung dieser AF-Bausteine oftmals sehr schwierig. Sollen beispielsweise notwendige Datentypen oder Datenstrukturen im ModelLocator verändert werden, müssen alle gebundenen View-Daten und alle Commands, welche auf diese Daten zugreifen, angepasst werden. Dies ist gerade bei größeren Anwendungen vergleichsweise aufwendig. Besonders einfach ist hingegen die Wartung externer Datenquellen. Durch eine zentrale Verwaltung externer Datenquellen im ServiceLocator können diese komfortabel angepasst werden. Da die Datenquellen durch Delegates vom restlichen Cairngorm-System isoliert sind, kann das System trotz Veränderungen in der Datenquelle meist unverändert bleiben. Ebenso wartungsfreundlich kann die Logik in Commands angepasst werden, was vor allem aus der guten Entkopplung der Views von den Commands resultiert.

Für eine bessere Testbarkeit wird von der Firma KapLab eine Debug-Console zur Verfügung[10] gestellt. Mit Hilfe dieser Konsole kann der Entwickler einen aufgetretenen Fehler sehr einfach aufspüren. Besonders hervorzuheben ist dabei die Möglichkeit, Abläufe im AF, wie etwa das Auslösen von Events oder das Ausführen von Commands, übersichtlich darzustellen. Außerdem enthält diese Cairngorm-Debug-Console Funktionalitäten zur Überwachung von externen Datenquellen. Zu bemerken ist aber, dass die Durchführung von Unit-Tests eher schwierig ist. Das liegt vor allem an der hohen Anzahl von Singletons und den damit verbundenen Kopplungen. Außerdem erschweren asynchrone Operationen die Testbarkeit eines einzelnen AF-Bausteines.

[10] http://lab.kapit.fr/display/cairngormconsole/Cairngorm+Console

Zusammengefasst können Cairngorm-Anwendungen als gut lesbar bezeichnet werden. Gerade externen Datenquellen können besonders einfach angepasst oder ausgetauscht werden. Bedingt durch eine teilweise enge Kopplung in dem AF ist eine Anpassung von Model-Daten sowie das Durchführen von Unit-Tests nicht immer problemlos möglich. Die verfügbare gute Debug-Console wirkt diesem Nachteile aber entgegen. Programme, welche unter Verwendung des Cairngorm-AF entstanden sind, können demnach als einfach und effizient wartbar bezeichnet werden (vgl. Abbildung 4.7).

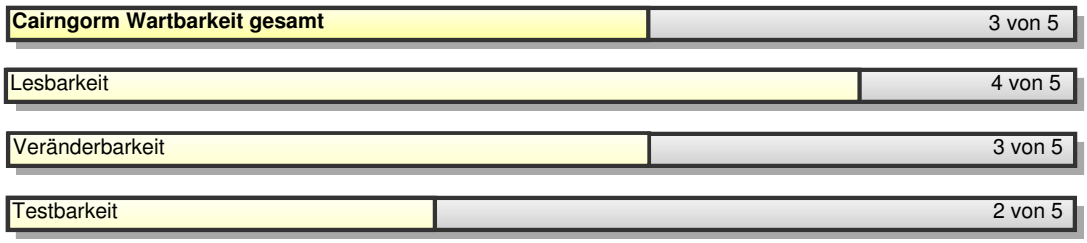


Abbildung 4.7: Wartbarkeit Cairngorm

### 4.3.2 PureMVC

Beim PureMVC-AF fallen zwei grundlegende Problemfelder bezüglich der Lesbarkeit bestehender Anwendungen auf. Wie bereits in Abschnitt 4.2.2 erläutert, kann bei der Verwendung des PureMVC-AF Programmlogik in Mediatoren, Proxies oder Commands implementiert werden. Dadurch ist es während der Wartung nicht immer einfach nachvollziehbar, wo der zu modifizierende Code zu finden ist. Resultierend aus der hohen Entkopplung der AF-Bausteine, durch Notifications, ist es zusätzlich ziemlich aufwendig Wartungen an der Kommunikation zwischen den AF-Bausteine durchzuführen. So ist es beispielsweise schwierig herauszufinden, welche Mediatoren oder Commands sich für eine konkrete Notification angemeldet haben. Durch diese mangelhafte Lesbarkeit des PureMVC-AF ist es während der Wartung immer wieder nötig, Code in den AF-Bausteinen zu durchsuchen, was vor allem bei großen Anwendungen zu erheblichem Wartungsaufwand führen kann.

Führte die hohe Entkopplung der AF-Bausteine bei der Lesbarkeit zu Problemen, ist diese für die Veränderbarkeit einer PureMVC-Anwendung von großem Vorteil. So können etwa Änderungen an Modeldaten in Proxies, Anpassungen von Nutzeroberflächen in Views oder Mediatoren sowie umfangreiche Umstrukturierungen in Commands weitestgehend ohne Auswirkungen auf andere AF-Bausteine vorgenommen werden. Ändert sich zum

Beispiel eine externe Datenquelle, müssen lediglich die entsprechenden Delegates, aber nicht die Proxies selbst angepasst werden. Daraus resultiert eine sehr gute Veränderbarkeit von PureMVC-Anwendungen und somit eine effiziente Wartbarkeit.

Ebenso wie für Cairngorm steht für PureMVC eine Debug-Console[11] der Firma Kap Lab zur Verfügung. Diese Console veranschaulicht, wie bei Cairngorm, die Abfolge von Prozessen in einem PureMVC-Programm und ermöglicht eine eingehende Untersuchung aller AF-Bausteine, einschließlich externen Datenquellen. Dies führt zu einer effizienteren Fehlersuche und somit zu einer erhöhten Wartbarkeit. Besonders hervorzuheben ist hierbei die Möglichkeit, Notification-Interessen von Mediatoren einzusehen, wodurch die Lesbarkeit der Notification-Kommunikation verbessert wird. Auf Grund der geringen Kopplung zwischen den AF-Bausteinen, sind Unit-Tests im Vergleich zu Cairngorm einfacher durchzuführen. Außerdem ist es mittels der PureMVC-Variante des Adobe Flex-Unit-Frameworks, PureMVC-FlexUnit[12], zusätzlich möglich asynchrone Operationen in die Unit-Tests einzubeziehen.

Betrachtet man zusammenfassend die Wartbarkeit von PureMVC-Anwendungen, fallen zwei Extreme auf. Einerseits sind bestehende Anwendungen, welche mit diesem AF erstellt wurden, schlecht zu lesen und nachzuvollziehen. Andererseits ist die Veränderbarkeit einer solchen Anwendung ausgezeichnet. Auch die Fehlersuche kann dank der sehr guten Testbarkeit von PureMVC-Anwendungen sehr effizient gestaltet werden. Die gute Testbarkeit sowie die flexible Veränderbarkeit gleichen den Nachteil der schlechten Lesbarkeit weitestgehend aus, weshalb PureMVC-Anwendungen insgesamt als gut wartbar eingeordnet werden können (vgl. Abbildung 4.8).

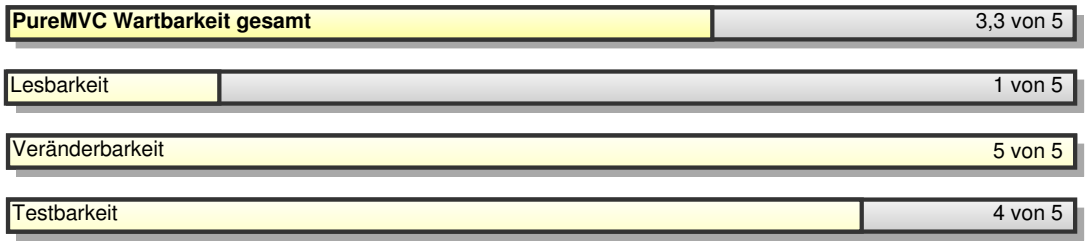


Abbildung 4.8: Wartbarkeit PureMVC

[11] http://lab.kapit.fr/display/puremvcconsole/PureMVC+Console
[12] http://code.google.com/p/puremvc-flexunit-testing/

### 4.3.3 MATE

Bei der Betrachtung der Wartbarkeit des MATE-AF, wird analog zu Abschnitt 4.2.3, vom Einsatz von Presentation-Models und Managern ausgegangen.

Bestehende MATE-Anwendungen sind sehr gut lesbar, aber nicht immer gut nachvollziehbar. Einerseits ist gerade die EventMap durch den Einsatz der MATE-MXML-Tags, besonders gut lesbar und übersichtlich, weshalb es auch unerfahrenen Entwicklern einfach möglich ist, AF-Systemreaktionen auf Events nachzuvollziehen. Untersucht man aber zum Beispiel eine Presentation-Model-Klasse, so ist es, bedingt durch den Einsatz von „dependency injection“, nicht immer leicht ersichtlich, wer die injizierten Objekte „liefert“. Diese Informationen müssen zusätzlich in der EventMap (Injector-Tags) ausfindig gemacht werden, was die zu untersuchenden Klassen damit oftmals schwer nachvollziehbar und somit schlechter wartbar macht.

Wie bereits in Abschnitt 4.2.3 angedeutet, ist die Abhängigkeit zwischen den einzelnen AF-Bausteinen durch den Einsatz von „dependency injection“ relativ gering. Gerade Manager sind daher recht unabhängig von Presentation-Models, was diese wiederum einfach veränderbar macht. Obwohl auch Presentation-Models ihrerseits in Views injiziert werden, greifen diese Views selbstständig auf Daten ihrer Presentation-Models zu. Im Vergleich zum PureMVC-AF sind Views und dessen Presentation-Model demnach enger gekoppelt, was beide schwerer veränderbar macht. Analog dazu macht das direkte Zugreifen der EventMap in verschiede Klassen eine Wartung dieser Klassen aufwendiger. Besonders effizient lassen sich allerdings Veränderungen in der EventMap selbst realisieren. Hierfür reicht oft eine Veränderung eines Tags aus, um die Funktionalität einer bestehenden Anwendung anzupassen. Auch der Umgang mit externen Daten ist, dank der zentralen Verwaltung, unkompliziert anpassbar und somit gut veränder- bzw. wartbar.

Nachteilig ist, dass es für das MATE-AF keine Debug-Console wie für PureMVC oder Cairngorm gibt. Für die Fehlersuche steht lediglich ein Debugger-MXML-Tag[13] zur Verfügung, welcher zum Beispiel in die EventMap oder hierarchisch höher eingefügt werden kann. Um ein MATE-MXML-Tag untersuchen zu können, muss dessen inline-Attribute „debug“ auf „true“ gesetzt werden. Nach dem Starten der Anwendung im Debug-Modus, werden die Debug-Informationen im Debug-Fenster des Flex-Builders ausgegeben. Im Vergleich zu den umfangreichen Informationen, welche die Cairngorm- oder PureMVC-Console zur Verfügung stellt, sind diese Informationen allerdings recht

[13] http://mate.asfusion.com/page/documentation/tags/debugger

mager und viel aufwendiger zu erzeugen.

Da alle Klassen, ausgenommen der EventMap, unabhängig vom MATE-AF sind und Singleton-Abhängigkeiten nicht vorhanden sind, steht Unit-Tests für Manager und Presentation-Models nichts im Wege. Unit-Tests in der EventMap selbst sind jedoch nicht möglich, im Regelfall aber auch nicht nötig.

Zusammengefasst steht der gut lesbaren EventMap eine relativ schlechte Nachvollziehbarkeit der einzelnen Klassen gegenüber. Somit kann die Lesbarkeit einer MATE-Anwendung lediglich als mittelmäßig eingestuft werden. Die Veränderbarkeit einer MATE-Anwendung hingegen ist gut. Die direkten Zugriffe auf Klassen, Objekte und Funktionen werden von der gut modifizierbaren EventMap und den unkompliziert anpassbaren externen Datenquellen ausgeglichen. Lediglich die Testbarkeit muss im Vergleich zu Cairngorm und PureMVC als schlechter beurteilt werden. Fasst man all diese Punkte zusammen, kann eine MATE-Anwendung insgesamt als gut wartbar bezeichnet werden (vgl. Abbildung 4.8).

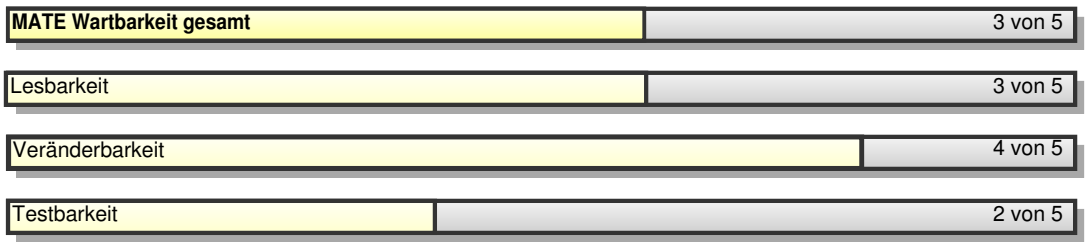


Abbildung 4.9: Wartbarkeit MATE

## 4.4 Bewertung

Nachdem in den Abschnitten 4.1 bis 4.3 Zugänglichkeit, Entwicklungseffizienz und Wartbarkeit der Architektur-Frameworks Cairngorm, PureMVC und MATE untersucht wurden, werden in diesem Abschnitt die AF hinsichtlich dieser drei Kriterien bewertet. Dazu werden die erreichten Punktzahlen in tabellarischer Form gegenübergestellt und auf besondere Vorteile bzw. Nachteile der einzelnen AF eingegangen. In Tabelle 4.1 wird die Bewertung der Kriterien wiedergegeben. Für einen besseren Überblick werden die besonders positiv bewerteten Merkmale eines AF grün und die problematischen Eigenschaften rot hinterlegt.

| | Cairngorm | PureMVC | MATE |
|---|---|---|---|
| **Zugänglichkeit** | **3,75 von 5** | **4 von 5** | **4,5 von 5** |
| geringe Komplexität | 4 von 5 | 1 von 5 | 3 von 5 |
| Dokumentation | 4 von 5 | 5 von 5 | 5 von 5 |
| Foren/ Community | 3 von 5 | 5 von 5 | 5 von 5 |
| Beispielprogramme/Tutorials | 4 von 5 | 5 von 5 | 5 von 5 |
| | | | |
| **Entwicklungseffizienz** | **2,3 von 5** | **3,2 von 5** | **3,2 von 5** |
| Code Aufwand | 1 von 5 | 3 von 5 | 4 von 5 |
| Vorteile durch klare Modularisierung | 3 von 5 | 2 von 5 | 4 von 5 |
| Umgang externe Datenquellen | 4 von 5 | 3 von 5 | 4 von 5 |
| verfügbare Erweiterungen | 2,5 von 5 | 4 von 5 | 2 von 5 |
| Wiederverwendung | 1 von 5 | 4 von 5 | 2 von 5 |
| | | | |
| **Wartbarkeit** | **3 von 5** | **3,3 von 5** | **3 von 5** |
| Lesbarkeit | 4 von 5 | 1 von 5 | 3 von 5 |
| Veränderbarkeit | 3 von 5 | 5 von 5 | 4 von 5 |
| Testbarkeit | 2 von 5 | 4 von 5 | 2 von 5 |
| | | | |
| **Gesamt** | **3 von 5** | **3,5 von 5** | **3,56 von 5** |

Tabelle 4.1: Bewertung Cairngorm, PureMVC und MATE

Betrachtet man die Tabelle 4.1, fällt schnell auf, dass die erreichten Gesamtpunktzahlen von Cairngorm, PureMVC und MATE nur geringfügig voneinander abweichen, während es bei Bewertungen einzelner Kriterien, bzw. deren Unterkategorien, deutlichere Unterschiede gibt. Es kann daher nicht von einem „richtigen“ oder „besten“ AF gesprochen werden. Vielmehr muss nach jeweiligem Einsatzgebiet jedes mal neu entschieden werden, welches AF für den speziellen Einsatz am besten geeignet ist.

Sucht man ein einfaches AF mit geringer Komplexität und hoher Lesbarkeit, empfiehlt es sich, das im Vergleich zu MATE oder PureMVC weniger effiziente Cairngorm-AF zu wählen. Besonders für Projekte mit geringem Funktionsumfang und voraussichtlich wenig Wartungsaufwand, welche von unerfahrenen Softwareentwicklern erstellt werden, ist Cairngorm eine sehr gute Wahl. Bei Projekten, welche in hohem Maße wieder verwendet werden sollen, ist vom Gebrauch des Cairngorm-AF jedoch genauso abzusehen, wie für sehr umfangreiche Anwendungen mit voraussichtlich hohem Wartungsbedarf.

Bei PureMVC-AF ist das Anwendungsgebiet ein völlig anderes als bei Cairngorm. Benötigt der Entwickler eine Softwarearchitektur für die Erstellung besonders einfach

veränderbarer Anwendungen, welche im hohen Maße wieder verwendet werden sollen, empfiehlt sich der Einsatz des PureMVC-AF. Die hohe Komplexität dieses AF erfordert jedoch recht viel Erfahrung der Entwickler und ist mit einem hohen Einarbeitungsaufwand verbunden. Aus diesem Grund empfiehlt sich der Einsatz dieses AF vor allem bei größeren Anwendungen, welche voraussichtlich lange Zeit Verwendung finden sollen und daher oft verändert bzw. gewartet werden müssen. Für kleine Projekte ist der Einarbeitungsaufwand aber zu hoch.

Das MATE-AF ist, anders als Cairngorm und PureMVC, sehr breit anwendbar. Abgesehen von der, gerade im direkten Vergleich mit PureMVC, schlechteren Wiederverwendung von AF-Bausteinen oder kompletten MATE-Modulen, konnten keine grundlegenden Schwächen gefunden werden. Gerade die geringe Code-Menge, welche benötig wird, um Anforderungen umzusetzen, in Verbindung mit der komfortablen Programmierung in der EventMap mittels MXML-Tags, machen den Einsatz des MATE-AF für viele Anwendungen sinnvoll.

Wie schon in der Einleitung dieser Arbeit beschrieben, ist es für den folgenden praktischen Abschnitt besonders wichtig, eine hohe Wiederverwendung einzelner Module zu erreichen bzw. umzusetzen, um diese im Rahmen eines Softwareunternehmens effektiv nutzen zu können. Bei den theoretischen Betrachtungen konnte festgestellt werden, dass diese Anforderung am besten vom PureMVC-AF erfüllt wird, weshalb für die Umsetzung des nachfolgenden Softwareprojektes (vgl. Kapitel 5) auch das PureMVC-AF gewählt wurde.

# 5 Praktische Umsetzung einer Rich-Internet-Application

Nachdem im theoretischen Teil Adobe-Flex (Kapitel 1) und Softwarearchitektur (Kapitel 2) eingeführt, verschiedene Architektur-Frameworks vorgestellt (Kapitel 3) und bewertet (Kapitel 4) wurden, befasst sich dieses Kapitel mit der Beschreibung des praktischen Teils der Arbeit. Für den praktischen Teil wurden verschiedene PureMVC-Module erstellt. Diese Module können in verschiedenen RIA-Anwendungen einfach wieder verwendet werden. In dieser Diplomarbeit wird beispielhaft eine komplexe RIA-Anwendung, welche alle erstellten Module nutzt, vorgestellt.

Nach der Einführung in den praktischen Teil wird das Konzept zum Umgang mit diesen Modulen vorgestellt. Abschließend wird beispielhaft auf konkrete Implementierungen innerhalb der Module eingegangen.

## 5.1 Einführung in die praktische Arbeit

In diesem Abschnitt werden zunächst die Ziele der praktischen Arbeit konkretisieret und anschließend die Herangehensweise bei der praktischen Beschreibung verdeutlicht.

### 5.1.1 Ziele der praktischen Arbeit

Wie sich bereits in Kapitel 4 gezeigt hat, ist das PureMVC-AF am besten für wieder verwendbare Anwendungen geeignet. Im Vergleich mit anderen AF hat sich gezeigt, dass der Einsatz von PureMVC-Modulen besonders effektiv ist (vgl. Abschnitt 4.4). Die grundlegende Motivation dieser praktischen Arbeit ist es, mittels des PureMVC-AF, eine beispielhafte RIA zu erstellen, welche aus wieder verwendbaren und unabhängigen PureMVC-Modulen besteht. Aus dieser Grundmotivation lassen sich zwei wesentliche Ziele ableiten.

1. Erstellung verschiedener PureMVC-Module (Modul intern).
2. Erarbeitung von Konzepten für den Umgang mit diesen PureMVC Modulen (Module extern).

Das erste Ziel der praktischen Arbeit bezieht sich auf eine einfache Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit innerhalb der PureMVC-Module. Das zweite Ziel beschäftigt sich hingegen damit, Module möglichst einfach und komfortabel in verschiedenen Anwendungen einsetzten zu können.

### 5.1.2 Herangehensweise zur Beschreibung der praktischen Arbeit

Um das praktische Vorgehen bestmöglich zu verdeutlichen, werden zunächst die wesentlichen Zusammenhänge vorgestellt und später auf die jeweiligen Implementierungen eingegangen. Die Beschreibung der entstandenen Anwendung erfolgt demnach von abstrakten Betrachtungen hin zu konkreten Implementierungen. Abbildung 5.1 verdeutlicht dieses Vorgehen.

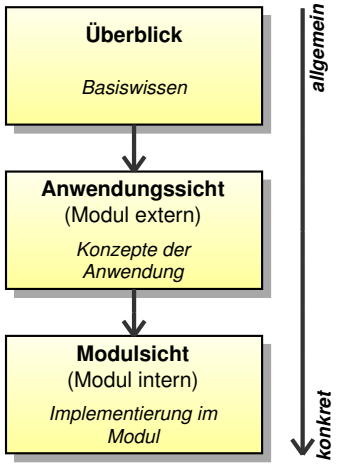


Abbildung 5.1: Aufbau der praktischen Beschreibung

- Überblick (Abschnitt 5.2): Dieser Teil der Beschreibung der praktischen Arbeit ist sehr allgemein gehalten und erläutert die Basis für das Verständnis der erstellten RIA-Anwendung.
- Externe Modulsicht (Abschnitt 5.3): Dieser Teil ist weniger abstrakt als der Überblick. Im Zentrum dieser Betrachtung steht die Beispielanwendung, welche

alle selbst erstellten Module enthält. Dabei wird insbesondere auf die Aufgaben dieser Anwendung eingegangen und Konzepte zum Umgang mit den verschiedenen Modulen erläutert. Darüber hinaus werden die in dieser praktischen Arbeit erstellten Module vorgestellt. Auf die Implementierungen innerhalb dieser Module wird jedoch nicht eingegangen. Dieser Abschnitt beschreibt die Umsetzung des zweiten Ziels (vgl. Abschnitt 5.1.1).

- Interne Modulsicht (Abschnitt 5.4): In diesem Abschnitt werden konkrete Implementierungen innerhalb von PureMVC-Modulen betrachtet. Wurde bei der externen Sicht, auf die Konzepte der Anwendung sowie auf die grundlegenden Funktionen der Module eingegangen, zeigt die interne Sicht, wie diese Konzepte innerhalb eines Moduls umgesetzt werden. Dabei wird beispielhaft auf die Implementierung von Proxies, Mediatoren/Views sowie auf verschiedene Command eingegangen. In der internen Modulsicht wird ausschließlich das betrachtete Modul vorgestellt und erläutert, andere Module spielen bei dieser Betrachtung eine untergeordnete Rolle. Dieser Abschnitt beschreibt die Umsetzung des ersten Ziels (vgl. Abschnitt 5.1.1).

## 5.2 Überblick

Dieser Abschnitt gibt einen ersten Überblick über das entstandene, praktische Projekt. Dabei wird zunächst auf den Zusammenhang zwischen Beispielanwendung und Modulen näher eingegangen und anschließend werden die verwendeten externen Server bzw. externen Datenquellen vorgestellt.

### 5.2.1 Programm-Hierarchie

Alle Module befinden sich in einer Beispielanwendung. Diese Anwendung wurde als PureMVC-Anwendung konzipiert. Die grundsätzliche Aufgabe dieser beispielhaften Anwendung ist das Erstellen und Konfigurieren der Module. Die in dieser Anwendung eingefügten Module nutzen ebenfalls das PureMVC-AF. Jedes Modul stellt dabei eine geschlossene und eigenständige Einheit dar, welche unabhängig von anderen Modulen funktionsfähig ist. Dadurch sind die Module einfach austausch- bzw. wieder verwendbar. Abbildung 5.2 verdeutlicht diesen Zusammenhang von Modulen und der Beispielanwendung.

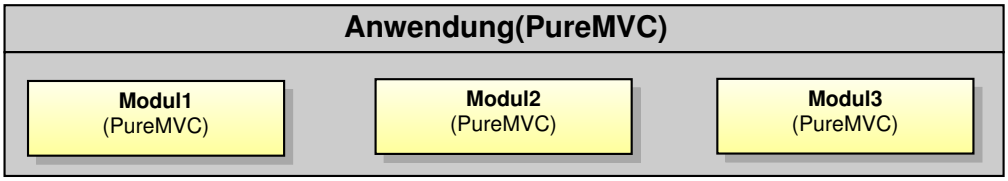


Abbildung 5.2: Zusammenhang Module und Beispielanwendung

### 5.2.2 Verwendete externe Datenquellen und Server

Nachdem im vorangegangenen Abschnitt der hierarchische Zusammenhang von Modulen im Bezug auf die Beispielanwendung verdeutlicht wurde, wird nachfolgend auf die verwendeten externen Datenquellen bzw. Server eingegangen. Abbildung 5.3 zeigt mögliche Elemente, mit denen ein Modul kommunizieren kann. Dazu gehören Webserver, Red5 Server oder RSS Server, welche nachfolgend kurz erläutert werden.

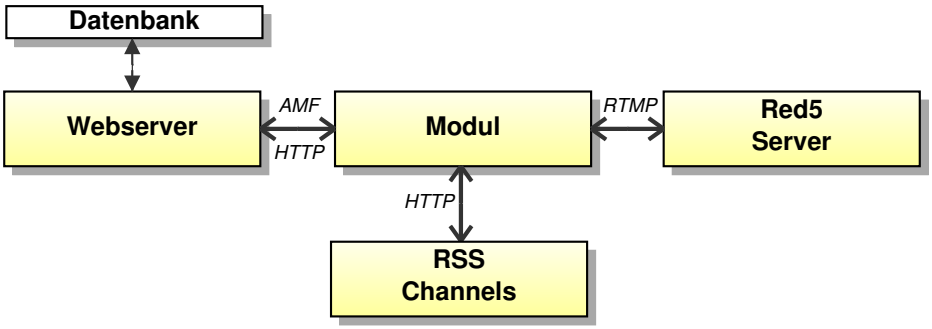


Abbildung 5.3: Verwendete externe Datenquellen und Server

- WEBSERVER: Als Webserver werden in dieser Arbeit Apache Webserver[1] genutzt. Der Webserver wurde unter Verwendung der Sprache PHP implementiert und kommuniziert mit Flex über das binäre AMF[2] (Action Message Format). An diese Webserver können Datenbanken, wie beispielsweise eine MySQL[3] Datenbank angeschlossen werden (im Prototyp wurde darauf verzichtet). Diese Arbeit nutzt Webserver, um Nutzerdaten zu überprüfen und benötigte Daten für die Module bereitzustellen.

[1] http://www.apache.org/
[2] http://www.amfphp.org/
[3] http://www.mysql.de/

- RED5-SERVER: Diese Diplomarbeit nutzt die open Source Red5-Technologie[4] über RTMP (Real Time Messaging Protocol). Mit Hilfe dieser Server ist es möglich, Audio und Video Streams zu empfangen bzw. zu senden. Um Daten unterschiedlicher Flex-Clients in Echtzeit synchron zu halten, können Red5-Shared Objects eingesetzt werden. Red5 ist eine kostenfreie Alternative zum Adobe Flash-Media-Server.

- RSS-CHANNELS: RSS-Channels werden von verschiedenen Webseiten angeboten und versorgen den Adressaten mit aktuellen Daten. Die Bereitstellung dieser Daten werden RSS-Feeds genannt und können sowohl Text-, Bild-, Audio- oder Videodaten sein. Der RSS-Channel wird von dem jeweiligen Betreiber aktualisiert und sendet nach Anfrage RSS-Feeds zurück.

Welche Server von welchem Modul benötigt werden, wird im nachfolgenden Abschnitt erläutert.

## 5.3 Externe Modulsicht

Bevor in diesem Abschnitt auf die Aufgaben der Beispielanwendung bzw. die Konzepte zum Umgang mit Modulen eingegangen wird, werden alle selbst erstellten/entwickelten Module kurz vorgestellt und der Kern der jeweiligen Modulfunktionalität erläutert. Als Anwendung wird nachfolgend der Teil des Gesamtprogramms verstanden, welcher die Module beinhaltet und diese konfiguriert.

### 5.3.1 Vorstellung der umgesetzten Module

Es wurden sechs Module erstellt: PODCAST- (RSS), MEMO-, LOGIN-, CHAT,- VIDEO CHAT- UND GLOBAL CHAT-MODUL, welche nachfolgend erläutert werden. In Abbildung 5.4 sind diese Module sowie die Server und externen Datenquellen (vgl. Abschnitt 5.2.2) der jeweiligen Module dargestellt.

[4] http://osflash.org/red5

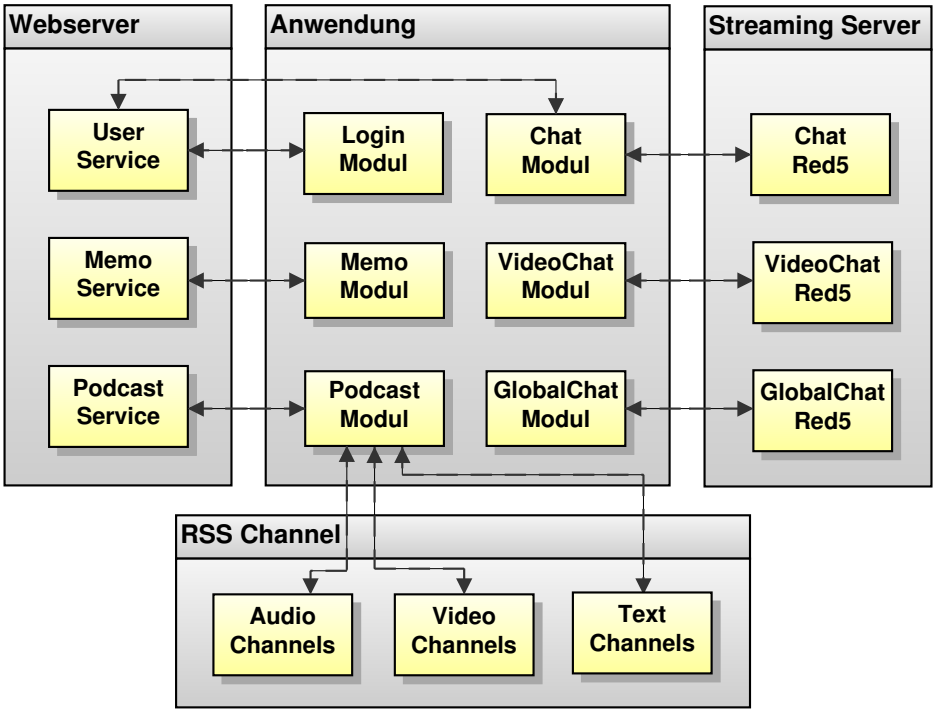


Abbildung 5.4: Externe Datenquellen und Server der Module

Das Login-Modul (Abbildung 5.5) ist von allen erstellten Modulen das einfachste Modul. Die Hauptaufgabe des Login-Modul besteht in der Anmeldung von Benutzern.


Abbildung 5.5: Login-Modul

Zum Überprüfen der Nutzerdaten benötigt das Login-Modul einen Webserver (UserService) (vgl. Abbildung 5.4). Nach Abfragen der Daten und erfolgreichem Login informiert das Login-Modul andere verbundene Module über den neu angemeldeten Benutzer.

Mit dem MEMO-MODUL (vgl. Abbildung 5.6) kann der Benutzer Notizen erstellen und löschen. Diese Notizen sind ausschließlich für den Nutzer sichtbar welcher diese Notiz erstellt hat. Bei der Umsetzung des Memo-Moduls wurde besonders darauf geachtet, dass


Abbildung 5.6: Memo-Modul

Notizen schnell und unkompliziert erstellt werden können und die Darstellung dieser Notizen übersichtlich gestaltet ist. Hierfür wurde eine Flex-Akkordeon-Komponente genutzt. Damit eine Notiz schnell wieder gefunden werden kann, werden in der Überschrift das Erstellungsdatum und die erste Zeile der Notiz angezeigt. Hinsichtlich der Nutzung von externen Datenquellen und Servern zeigt Abbildung 5.4, dass das Memo-Modul lediglich einen Webserver (MemoService) nutzt. Dieser wird dazu verwendet, Notizen des eingeloggten Nutzers zu verwalten bzw. diese zur Verfügung zu stellen.

Das GLOBAL CHAT-MODUL (vgl. Abbildung 5.7) wurde erstellt, damit sich alle angemeldeten Benutzer, unabhängig davon ob diese sich kennen, miteinander unterhalten können. Das Global Chat-Modul unterstützt eine textbasierte Unterhaltung (Chat) zwischen den Nutzern. Nachrichten, welche durch dieses Modul versendet werden, sind für alle angemeldeten Benutzer sichtbar. Damit es einen besseren Überblick gibt, welche Nutzer gerade im Global Chat-Modul angemeldet sind, werden diese in einer Liste angezeigt. Die Abbildung zeigt neben allen eingeloggten Nutzern (drei Personen sind eingeloggt: „s“, „musterman“ und „m“) auch den Chatverlauf dieser Benutzer. Aus Abbildung 5.4 ist zu


Abbildung 5.7: Global Chat-Modul

erkennen, dass das Global Chat-Modul einen Red5-Server (GlobalChatSteamingServer) benötigt. Dieser Red5-Server hat die Aufgabe, alle angemeldeten Flex-Clients in Echtzeit über Ein- und Ausloggen der Nutzer sowie des Chatverlaufs zu informieren.

Das Chat-Modul (vgl. Abbildung 5.8) dient der Kommunikation mit Freunden, ohne dass andere angemeldete Benutzer mitlesen können. Für die Kommunikation mit einem Freund steht jeweils ein Fenster zur Verfügung, wobei bei der Kommunikation mit mehreren Freunden gleichzeitig mehrere Chat Fenster geöffnet sein müssen. Um Freunde hinzufügen zu können, enthält die Freundesliste ein Suchfenster, welches durch Anklicken von „SearchFriend“ aktiviert werden kann. Die Abbildung 5.8 zeigt beispielhaft das Chat-


Abbildung 5.8: Chat-Modul

Modul mit Freundesliste ohne Suchfenster sowie ein Chat Fenster. In der Freundesliste werden Benutzer, welche eingeloggt sind mit einem grünen Punkt gekennzeichnet (Status

in Echtzeit). Zusätzlich bietet das Chat-Modul einen Video Chat-Button. Wird dieser Button angeklickt, wird mittels Video Chat-Modul eine Video Chat Anfrage an den Chat Partner geschickt.

Für das Chat-Modul wird ein Webserver (UserService) zum Verwalten der Freunde und ein Red5-Server (ChatStreamingServer) benötigt (vgl. Abbildung 5.4). Der Java-Red5-Server informiert die Flex-Clients über eingeloggte Freunde sowie über neue Chat-Nachrichten. Verschiedene Implementierungen dieser Funktionalität werden im Abschnitt 5.4 erläutert.

Das Video Chat-Modul wurde, wie bereites beim Chat-Modul erwähnt, implementiert, um eine Video/Audio-Kommunikation zu ermöglichen. Nach Starten des Video Chats erscheinen auf dem Bildschirm zwei Fenster. In einem Fenster wird das eigene Video Bild und in dem anderen Fenster das Bild des Video Chat Partners gezeigt. Ist jedoch keine Webcam angeschlossen bleibt das jeweilige Fenster schwarz. Der Ton wird, sofern ein Mikrofon angeschlossen ist, dennoch übertragen. Da dieses Modul neben den beiden Fenstern keine Steuerelemente besitzt, wird auf eine Abbildung dieses Moduls verzichtet. Im Rahmen dieser praktischen Arbeit wird die Video Chat Funktionalität nur für Freunde, ausgehend vom Chat-Modul, angeboten. An dieser Stelle soll jedoch darauf hingewiesen werden, dass es mit geringem Aufwand möglich ist, ausgehend von anderen Modulen wie zum Beispiel dem Global Chat-Modul einen solche Video Chat zu starten. Wie in Abbildung 5.4 ersichtlich, benötigt das Video Chat-Modul einen Red5-Server (VideoChatStreamingServer). Dieser Server ermöglicht das Streamen von Video- und Audiodaten zwischen verschiedenen Flex-Clients.

Das Podcast-Modul (RSS-Modul) ermöglicht es einem Benutzer Text-, Audio- und Video-RSS-Feeds zu verfolgen. Dazu kann der Nutzer beliebig viele RSS-Channels anlegen. Abbildung 5.9 zeigt die Ausgangsansicht des RSS-Moduls. Die Abbildung des RSS-Moduls zeigt neben der Liste der RSS-Channel, welche vom Benutzer bereits eingetragen wurden, ebenso zwei Texteingaben und einen „create“ Button zum Eintragen neuer Channels. Nach Anklicken eines Channels werden die aktuell enthalten RSS Feeds des Channels vom entsprechenden Server erfragt.

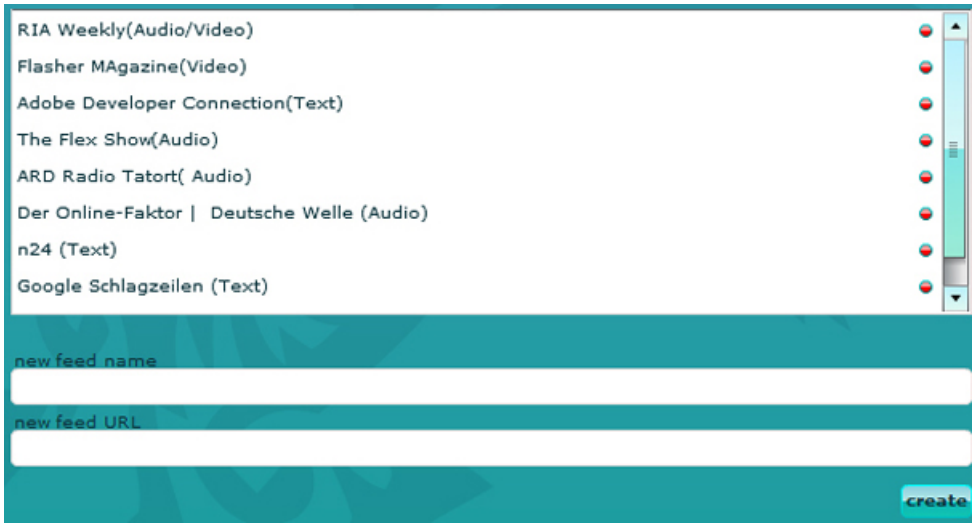


Abbildung 5.9: Podcast-Modul: Ausgangsansicht

Die beispielhafte zweite Ansicht des Podcast-Moduls, nachdem vom Benutzer der Channel „ARD Radio Tatort“ angewählt wurde, ist in Abbildung 5.10 dargestellt.


Abbildung 5.10: Podcast-Modul: Detailansicht

In dieser Ansicht erhält der Benutzer eine Beschreiung des gewählten Channels mit einem Bild sowie einen Überblick über die vorhandenen Feeds. Entsprechend der Art des Feeds (Text, Audio oder Video) ändert sich nach dem Anwählen eines Feeds die Ansicht des RSS-Moduls. Während bei Text-Feeds lediglich ein Textfeld angezeigt wird, erscheint bei Video-Feeds ein Video Fenster mit Video Player. Ist der angewählte Feed

ein Audio-Feed, erscheint ein Audio Player.

Da hier nur eine grundlegende Einführung in die einzelnen Module gegeben werden soll, wird auf eine genauere Beschreibung des RSS-Moduls, speziell der Ansichten für die verschiedenen Feed-Arten (Text, Audio und Video), im Rahmen dieser Arbeit verzichtet.

### 5.3.2 Aufgaben der Beispielanwendung

Während in Abschnitt 5.3.1 zunächst alle umgesetzten Module beschrieben wurden, sollen in diesem Abschnitt die Aufgaben der Beispielanwendung, in welcher sich die einzelnen Module befinden, aufgezeigt werden. Der hierarchische Zusammenhang zwischen einer Anwendung und Modulen wurde bereits im Abschnitt 5.2 erläutert. Im Nachfolgenden werden alle Anwendungsaufgaben bezüglich des Umgangs mit Modulen aufgeführt und anschließend eingehender beschrieben.

- Erzeugen der Module: Die Anwendung ist dafür zuständig, die verschiedenen Module anzulegen.
- Übergeben und Definieren von Datenquellen: Jedes in dieser Arbeit umgesetzte Modul benötigt externe Server bzw. Datenquellen (vgl. Abbildung 5.4). Die Spezifikationen dieser Datenquellen werden von der Anwendung in die Module injiziert.
- Verbinden der Module: Es ist Aufgabe der Anwendung, Verbindungen zwischen Modulen bzw. zwischen Anwendung und Modulen zu erstellen. Durch diese Verbindungen können Module untereinander bzw. Module und die Anwendung kommunizieren.
- Anzeigen der Modul-Views: Ein Modul besitzt Views. Diese Views werden von dem Modul selber aber nicht angezeigt, sondern an die Anwendung übergeben. Aufgabe der Anwendung ist es, diese Views entgegenzunehmen und zu einer grafischen Benutzeroberfläche anzuordnen.

Das Erzeugen der Module wurde möglichst komfortabel umgesetzt. Dazu muss lediglich im Haupt-Tag (Application Tag) der Anwendung für jedes Modul ein MXML-Tag erzeugt werden. Nachfolgender Listing 5.1 zeigt den zum Anlegen aller Module benötigten Code. In dieser Arbeit wurden alle Module in einen Namensraum (*modules.**)

erstellt, was das Erstellen der Module noch übersichtlicher gestaltet. Die Module befinden sich nun in der Anwendung.

**Listing 5.1** DiplomPraxis.mxml, Anlegen der Module

```
<mx:Application xmlns:modules="modules.*"... />
        ...
        <modules:VideoChatModule id="videoChatModule"... />
        <modules:ChatModule id="chatModule"... />
        <modules:GlobalChatModule id="globalChatModule"... />
        <modules:LoginModule id="loginModule"... />
        <modules:MemoModule id="memoModule"... />
        <modules:PodCastModule id="podCastModule"... />
...
</mx:Application>
```

Für das Übergeben und Definieren von Datenquellen wurde in dieser Arbeit ein eigenes Konzept eingeführt. Ausgangspunkt ist der Einsatz von Delegates analog dem „Café Townsend“-Beispiel (vgl. Anhang A.2). Jedes Delegate beschreibt dabei eine benötigte Datenquelle eines Moduls (vgl. Abbildung 5.4). Aufgabe der Anwendung ist es, diese Delegates zu erstellen und anschließend an Module zu übergeben (injizieren). Module definieren ihre Delegates demnach nicht selber, sondern nutzen lediglich übergebene Delegates. Das hat den Vorteil, dass bei Änderungen der Datenquellen eines Moduls kein Code im Modul verändert werden muss. Es ändert sich lediglich das Delegate, welches von der Anwendung definiert wird. Das macht die Nutzung der Module deutlich flexibler und erhöht somit die Wiederverwendbarkeit.

Bei der Umsetzung dieses Konzeptes wurde besonders darauf geachtet, die Handhabung dieser Injektion von Delegates möglichst einfach zu gestalten. Daher wurde der Ansatz gewählt, die Delegates bereits beim Anlegen der Module in den Modul-MXML zu übergeben. Listing 5.2 zeigt die Injizierung von Delegates in der Beispielanwendung am Beispiel des Chat Moduls. Wie man dem Code entnehmen kann, werden dem Chat-Modul keine Delegate-Objekte sondern Delegate-Klassen übergeben (*ChatRed5Delegate und UserDelegate*). Neben der einfachen und komfortablen Inline-Übergabe der Delegates, entfällt somit sogar das Erstellen eines Delegate-Objektes. Die genaue Anwendung dieser Delegates in den Modulen wird erst im Abschnitt „Modul intern“ (vgl. Abschnitt 5.4) erläutert.

**Listing 5.2** DiplomPraxis.mxml, Übergeben von Delegate-Klassen

```
<mx:Application xmlns:modules="modules.*..."/>
        <mx:Script>
                <![CDATA[
                import application.controller.delegates.ChatRed5Delegate;
                import application.controller.delegates.MemoDelegate;
                ...]]>
        </mx:Script>
        ...
        <modules:ChatModule id="chatModule"
                chatDelegateClass="{ChatRed5Delegate}"
                userDelegateClass="{UserDelegate}"/>
        ...
</mx:Application>
```

In Bezug auf die Delegates bleibt dennoch eine Frage offen: Woher weiß man beim Erstellen der Delegate-Klassen innerhalb der Anwendung, welche Modul-Funktionalitäten ein Delegate erfüllen muss? Durch den Einsatz von Interfaces wurde in dieser Arbeit die beschriebene Problematik gelöst. Das genaue Vorgehen ist in Abbildung 5.11 am Beispiel des Delegates für den UserService dargestellt.

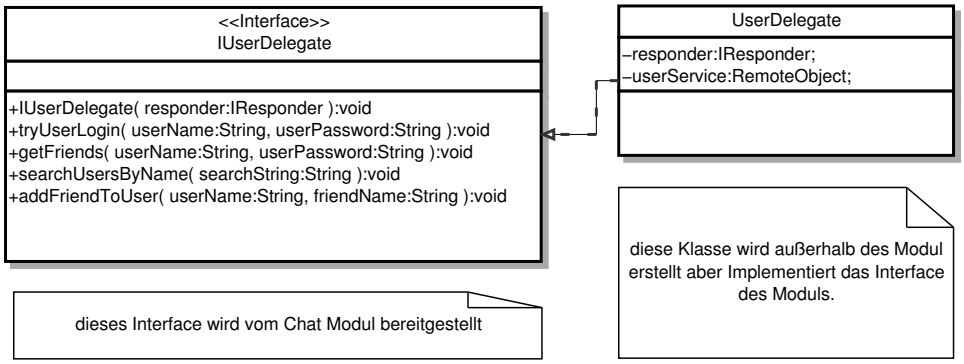


Abbildung 5.11: UserDelegate und IUserDelegate

Jedes Modul definiert für jedes benötigte Delegate ein eigenes Interface, welches das Anwendungs-abhängige Delegate implementieren muss. Alle benötigten Funktionen des Moduls bezüglich einer Datenquelle sind in diesem Interface definiert. Nutzt man ein Modul in einer neuen Anwendung, muss ein neues Delegate erstellt werden, welches das Interface aus dem Modul implementiert. Das Modul selbst bleibt jedoch unverändert.

Für das Verbinden der Module wurde auf die PureMVC-Pipes&Filter-Erweiterung[5] aufgebaut. Pipes&Filter ist ein Architekturmuster [BMR+98], welches für PureMVC von Cliff Hall entwickelt wurde. Zwar wurden die Module wie z.B. das Login- und Chat-Modul so entwickelt, dass beide unabhängig voneinander arbeiten können, jedoch müssen Daten zwischen beiden Modulen ausgetauscht werden. So kann das Chat-Modul nur korrekt arbeiten wenn es die Login-Daten des Nutzers kennt (Sonst wäre nur ein absolut anonymer Chat möglich). Diese Daten könnte sich das Chat-Modul beispielsweise aus dem in dieser Arbeit entwickelten Login-Modul beschaffen. Damit wäre aber die Unabhängigkeit des Chat-Moduls nahezu nicht mehr vorhanden. Auch das Login-Modul könnte so nicht mehr ausgetauscht werden. Die Frage ist also wie man eine lose Kopplung zwischen den Modulen (bzw. zwischen Modulen und der Beispielanwendung) erreichen kann. Mit der PureMVC Pipes&Filter-Erweiterung ist es möglich, Anwendung und Module sowie Module untereinander zu verbinden ohne diese eng miteinander zu koppeln. Dabei ist es einem Modul niemals bekannt, ob weitere Module in der Anwendung existieren. Ein Modul kann sich daher nicht selbstständig mit anderen Modulen verbinden. Der Anwendung, welche die Module inkludiert hat, sind allerdings alle Module bekannt, weshalb es auch Aufgabe der Anwendung ist, die Module miteinander zu verbinden. Abbildung 5.12 verdeutlicht den einfachsten Fall einer Verbindung von zwei Modulen.

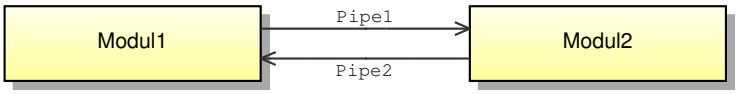


Abbildung 5.12: Verbindung von zwei Modulen per Pipe

Wie diese Abbildung zeigt, ist eine Kommunikation über eine Pipe immer nur in eine Richtung möglich. Wenn zwei Module miteinander kommunizieren möchten, müssen demnach zwei Pipes angelegt werden. Durch diese Verbindungen ist es möglich, PureMVC-Messages zwischen Modulen bzw. zwischen Modulen und Anwendung über Pipes auszutauschen. Im praktischen Teil dieser Arbeit hat sich hingegen gezeigt, dass eine solche einfache Kommunikation nicht immer ausreichend ist. Teilweise ist es nötig, über Pipe versendete Messages zu modifizieren, bevor diese empfangen werden. Für diese Aufgabe sind Filter zuständig. Filter werden zwischen Pipes platziert, um den Datenstrom modifizieren zu können. Abbildung 5.13 verdeutlicht dieses Vorgehen.

[5] http://trac.puremvc.org/Utility_AS3_MultiCore_Pipes

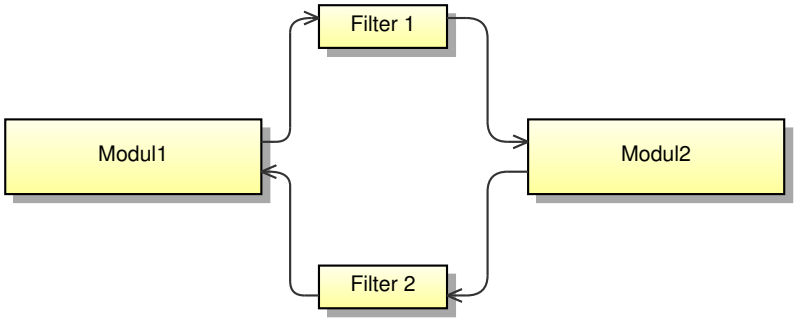


Abbildung 5.13: Verbindung von zwei Modulen per Pipe&Filter

Um Datenveränderungen übersichtlich und leicht wartbar zu gestalten, ist es möglich beliebig viele Filter zu verketten. Alle Pipes und Filter, die im Rahmen der praktischen Arbeit erstellt wurden, sind in Abbildung 5.14 dargestellt.

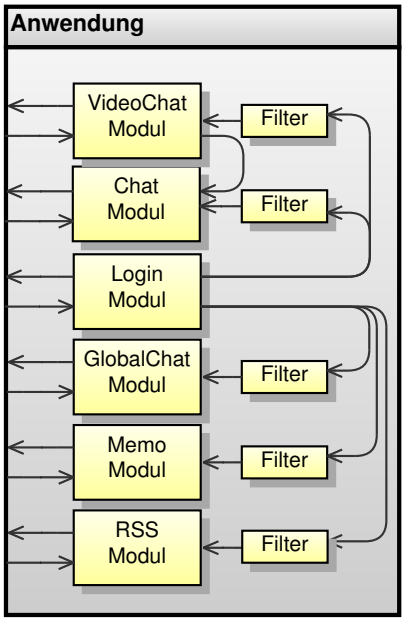


Abbildung 5.14: Pipes&Filter von allen Modulen

Wie aus der Grafik ersichtlich ist, war es beim Erstellen der hier beschriebenen RIA nötig, für die Pipes zwischen dem Login-Modul und den anderen Modulen, einen Filter zu platzieren. Nach Einloggen des Nutzers sendet das Login-Modul ein Datenobjekt (Value Object) des angemeldeten Nutzers an die angeschlossenen Module. Dieses Datenobjekt (*LoginUserVO*) ist ein Objekt einer im Login-Modul definierten Klasse. Da die Module unabhängig voneinander einsetzbar bleiben sollen, dürfen die anderen Module diese

Klasse des Login-Moduls nicht kennen. Deshalb ist es erforderlich, das Login-Modul Datenobjekt in ein dem jeweiligen Modul bekanntes Datenobjekt umzuwandeln. Dies geschieht durch einen Filter. Dieses Vorgehen ist in Abbildung 5.15 am Beispiel eines Filters der Pipe vom Login-Modul zum Chat-Modul dargestellt.

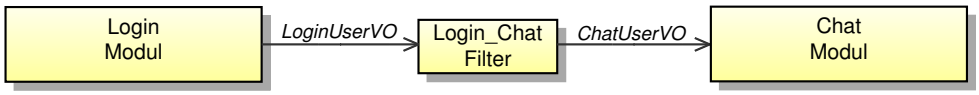


Abbildung 5.15: Pipes&Filter von Login-Modul und Chat-Modul

Im Rahmen des praktischen Teils dieser Diplomarbeit wurde besonders darauf geachtet, den Umgang mit dem Pipe&Filter-Konzept möglichst einfach zu gestallten. Deshalb wurde das PureMVC-Pipes&Filter-Konzept vereinfacht. So ist für das Erstellen einer Pipe mit angeschlossenem Filter lediglich eine Zeile Code notwendig. Listing 5.3 zeigt das Verbinden von Login-Modul und Chat-Modul unter Verwendung einer eigens entwickelten statischen Hilfsklasse.

**Listing 5.3** ConnectModulesCommand.as, Verbinden von Login- und Chat-Modul

```
ConnectionHelper.connect(loginModuleFacade, chatModuleFacade,
        PipeNameConstants.LOGIN_CHAT, new Login_Chat_Filter() );
```

Die Verbindung alle Module bzw. der Module mit der Anwendung erfolgt im *ConnectModulesCommand* der Anwendung. Ebenso einfach wie das Erstellen einer Pipe ist das Senden einer Nachricht über diese Pipe. Dies geschieht ebenfalls durch nur eine Zeile Code. Listing 5.4 zeigt, wie eine bereits erstellte Nachricht (*message*) gesendet wird.

**Listing 5.4** ConnectModulesCommand.as, Senden einer Pipe-Nachricht

```
sendNotification( JunctionMediator.SEND_MESSAGE, message );
```

Diese eine Codezeile bewirkt das Senden der Message über alle dem Modul bzw. der Anwendung angeschlossenen Pipes. Diese aufgezeigten Vereinfachungen erleichtern das Arbeiten mit Pipes&Filtern erheblich, da dadurch die Komplexität vom Programmierer fern gehalten wird. Eine genaue Beschreibung der zugrunde liegenden Funktionalität wird erst im Abschnitt „Modul intern“ (vgl. Abschnitt 5.4) gegeben.

Für das Anzeigen der Modul-Views wurde ebenfalls ein eigenes Konzept entwickelt. Damit die Views eines Moduls flexibel eingesetzt werden können, zeigt ein Modul seine Views nicht selbständig an, sondern sendet diese über eine Pipe an die Anwendung. Abbildung 5.16 zeigt beispielhaft das Versenden eines Chat Fensters (*ChatWindow.mxml*) des Chat-Moduls an die Anwendung über eine Pipe.

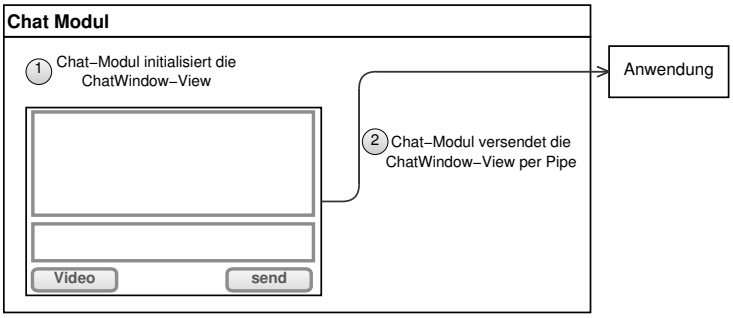


Abbildung 5.16: Chat-Modul versendet ChatWindow-View per Pipe

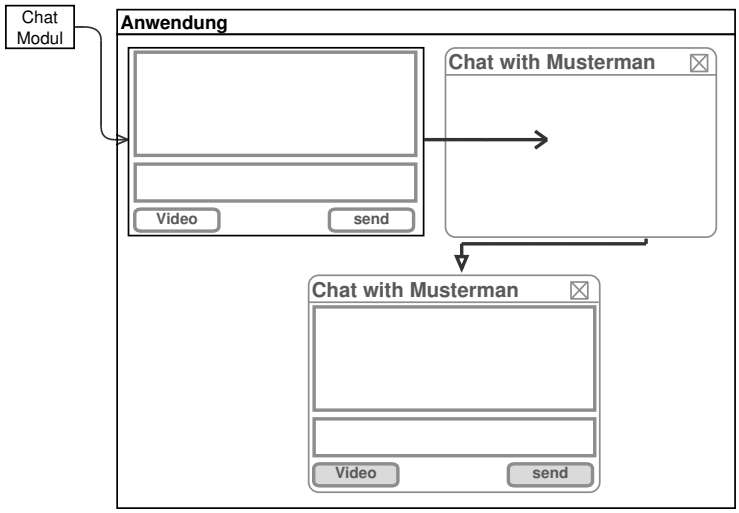


Abbildung 5.17: ChatWindow-View und Fenster

Die Anwendung kann die über die Pipe erhaltenen GUI-Elemente beliebig anordnen. Dabei spielt es keine Rolle, zu welchem Modul diese View gehört. In der praktischen Arbeit wurde entschieden, eine fensterbasierende grafische Benutzeroberfläche zu erstellen. Dazu wurde die über Pipe empfangene View in ein Fenster integriert. Abbildung 5.17 zeigt wie die ChatWindow-View (*ChatWindow.mxml*) des Chat-Moduls von der Anwendung in

ein Fenster integriert wird.Solche Fenster können verschoben, in der Größe verändert und teilweise geschlossen werden. Dabei wird ein an Windows angelegtes Arbeiten ermöglicht. Zusammenfassend kann man sagen, dass alle Views der Module durch die Anwendung verwaltet werden. Für ein Modul ist es dabei unerheblich, wie und wo seine Views angeordnet sind, da diese Anordnung die Funktionalität nicht beeinträchtigt. Dieses Konzept ermöglicht die Erstellung verschiedenster Benutzeroberflächen, ohne dabei die Module verändern zu müssen. Ergänzt wird dieses Konzept durch den konsequenten Einsatz von CSS-Dateien, durch welche die Gestaltung der Modul-Elemente zusätzlich verändert werden kann. Im folgenden Abschnitt wird ausführlich erläutert, wie die Module ihre Views versenden.

## 5.4 Interne Modulsicht

Der Abschnitt „Interne Modulsicht“ beschreibt die Funktionalität innerhalb eines Moduls. Da die Beschreibung eines jeden Moduls den Rahmen dieser Arbeit deutlich übersteigen würde, wurde das Chat-Modul ausgewählt, um beispielhaft die innere Funktionsweise eines Moduls aufzuzeigen. Da es sich bei den Modulen (vgl. Abschnitt 5.3.1) um PureMVC-Module mit dem MVC-Architektumuster (vgl. 2.4.2) handelt, wird die Beschreibung des Chat-Moduls anhand der grundlegenden MVC-Unterteilung strukturiert. Als erstes werden die View-Elemente des Moduls vorgestellt, um anschließend auf den Controller sowie auf das Model des Chat-Moduls näher einzugehen. Den abschließenden Punkt bildet die Beschreibung der Umsetzung der Pipes&Filter-Erweiterung innerhalb der Module am Beispiel des Chat-Moduls.

### 5.4.1 Implementierungen der View

Bei der Beschreibung der View des Chat-Moduls wird stellvertretend für alle anderen Module darauf eingegangen, wann und wie die Modul-Views an die Anwendung übergeben werden. Zuvor werden aber zunächst alle vorhandenen Views des Chat-Moduls vorgestellt und der Zusammenhang zu den Mediatoren hergestellt. Um zu verdeutlichen, wie bei der Kommunikation zwischen Mediatoren und deren Views vorgegangen wurde, wird abschließend ein für alle Module repräsentatives Beispiel erläutert.

In der praktischen Arbeit wurden alle Steuerelemente als MXML-Tags in MXML-Dateien (Views) erstellt. Damit der Aufbau einer grafischen Benutzeroberfläche übersicht-

lich bleibt und einfach verändert werden kann, wurde diese GUI auf mehrere hierarchisch gegliederte MXML-Dateien verteilt. Abbildung 5.18 zeigt die View-MXML-Datein des Chat-Moduls.

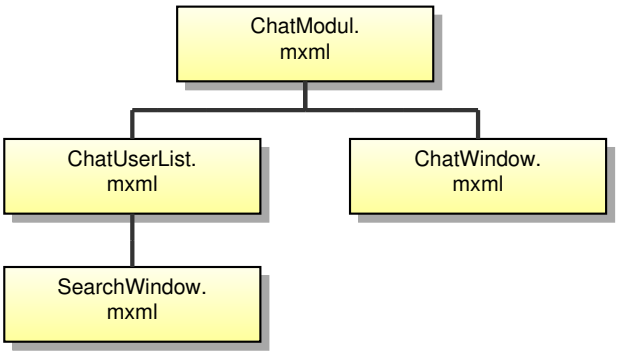


Abbildung 5.18: Chat-Modul View-Hierarchie

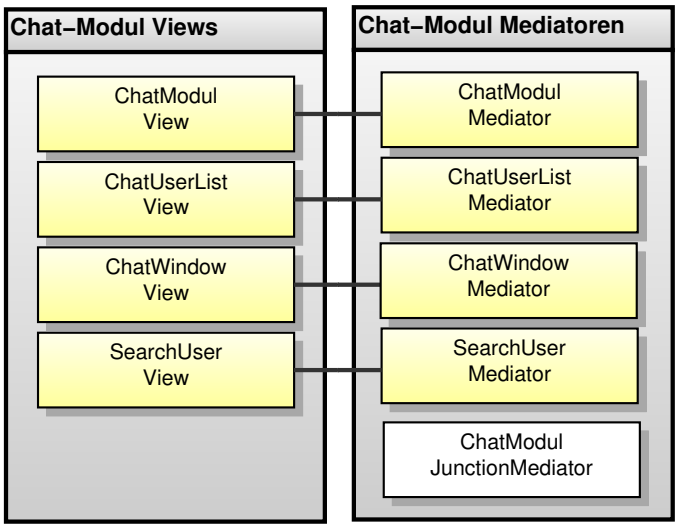


Abbildung 5.19: Chat-Modul Views mit PureMVC-Mediatoren

Um diese Views komfortabel steuern zu können, besitzt jedes dieser View-Elemente einen eigenen Mediator (vgl. Abbildung 5.19). Diese Mediatoren steuern die View-Elemente und vermitteln Benutzerinteraktionen an andere PureMVC-AF-Bausteine. In der Abbildung 5.19 gibt es einen Mediator, dem keine View zugeordnet ist: der *ChatModulJunctionMediator*. Dieser Mediator dient nicht zur Vermittlung zwischen MXML-Views und dem restlichen PureMVC-Modul-System, sondern zur Vermittlung bzw. Kom-

munikation mit anderen Modulen oder der Anwendung. Eine genaue Beschreibung dieses Mediators wird in Abschnitt 5.4.4 gegeben.

Das Konzept für den Umgang mit Modul-Views wurde bereits in Abschnitt 5.3.2 vorgestellt. Dabei wurde erläutert, dass es die Aufgabe der Anwendung ist, die in einem Modul enthaltenen Views anzuzeigen. Jetzt soll gezeigt werden, wie und wann diese Views versendet werden. Nachdem die Anwendung alle Module erstellt und verbunden hat, versendet die Anwendung über die zuvor erstellten Pipes eine Nachricht (System:Ready) an die Module. Diese Nachricht führt in jedem Modul zum Auslösen eines Commands, dem *CreateStdUICompCommand*. Diese Command erstellt alle benötigten Views und sendet diese zurück zur Anwendung. Listing 5.5 zeigt stellvertretend für alle in dieser Diplomarbeit erstellten Module das *CreatStdUICommand* des Chat-Moduls.

**Listing 5.5** CreateStdUICompCommand, Erstellen und Senden der ChatUserList-View

```
...
var list:ChatUserList = new ChatUserList();

facade.registerMediator( new ChatUserListMediator( list ) );
facade.registerMediator( new SearchUserMediator( list.searchUser )) ;

var msg:Message = new Message(null, msgHeader, list );
sendNotification( JunctionMediator.SEND_MESSAGE, msg );
...
```

Dieses Listing zeigt das Erstellen und Versenden der ChatUserList. Nachdem die ChatUserList angelegt ist, werden im PureMVC-Kern des Chat-Moduls die Mediatoren für die ChatUserList (*ChatUserListMediator*) sowie für die SearchUser View (*SearchUserMediator*) registriert. Anschließend erfolgt das Erstellen der Nachricht (*msg-Variable*), welche zum Versenden der View benötigt wird. Bevor die Nachricht verschickt werden kann, wird die ChatUserList-View (*list-Variable*) an die Message übergeben. Auf die *msgHeader-Variable* aus dem Listing 5.5 wird später in diesem Abschnitt eingegangen.

Das ChatUserWindow wird nicht in diesem Command erstellt. Ein neues ChatUserWindow wird erst nach dem Anklicken eines Freundes aus der ChatUserList dynamisch generiert. Das funktioniert analog zum *CreatStdUICommand* im *CreateNewChatWindowCommand*. Die Anwendung, welche das Modul beherbergt, empfängt diese Views und ordnet diese beliebig an.

Nachdem darauf eingegangen wurde, welche Views und Mediatoren im Chat-Modul

enthalten sind und wann bzw. wie diese an die Anwendung versendet werden, wird nachfolgend ein Beispiel für eine Nutzerinteraktion beschrieben. Ziel dieses Beispiels ist es aufzuzeigen, wie bei der Umsetzung aller Module die Trennung zwischen Views und Mediatoren umgesetzt wurde und welche Vorteile daraus resultieren. Die Wahl eines repräsentativen Beispiels fiel auf die Beschreibung des Vorgangs beim Versenden einer Nachricht an einen Freund. Gegenstand der nachfolgenden Betrachtung ist demnach die *ChatWindow.mxml-Datei* bzw. der *ChatWindowMediator* des Chat-Moduls. Listing 5.6 zeigt relevante Teile der ChatWindow.mxml.

**Listing 5.6** ChatWindow.mxml, Umgang mit Events und Flex-Databinding

```
<mx:Canvas creationComplete="init( event )"...>

<mx:Script> <![CDATA[
        public static const SEND_MESSAGE:String = "sendMessage";
        public static const VIDEO_BTN_CLICKED:String = "videoBtnClicked";

        [Bindable]public var chatOutputText:String = "";
        [Bindable]public var chatInputText:String = "";

        private function init( e:Event ):void
        {
                chatInput.addEventListener( KeyboardEvent.KEY_UP, onKeyUp );
        }
        private function onKeyUp( event:KeyboardEvent ):void
        {
                if( event.keyCode == Keyboard.ENTER )
                        dispatchEvent( new Event( SEND_MESSAGE ) );
        }
]]></mx:Script>

<mx:Binding source="chatOutput.text" destination="chatOutputText" />
<mx:Binding source="chatInput.text" destination="chatInputText" />

<mx:Button click="{dispatchEvent( new Event( VIDEO_BTN_CLICKED ) )}"
        label="VideoChat" id="videoChatButton".../>

<mx:Button click="{dispatchEvent( new Event( SEND_MESSAGE ) )}"
        label="send" id="sendButton".../>

<mx:TextArea text="{chatOutputText}" id="chatOutput"  ...>
<mx:TextArea text="{chatInputText}" id="chatInput".../>
...
</mx:Canvas>
```

Wie man diesem Listing entnehmen kann, befinden sich im ChatWindow lediglich vier sichtbare MXML-Elemente: zwei Buttons und zwei Textfelder (vgl. Abbildung 5.8). In die *chatInput-TextArea* (TA) wird der zu sendende Text eingetragen und die *chatOutPut-TA* zeigt den bisherigen Chat Verlauf an. Betätigt man den „*sendButton*“ wird ein Event (*SEND_MESSAGE*) gesendet, welches den Mediator veranlasst, eine neue Chat-Nachricht abzuschicken. Der „*videoChatButton*” soll an dieser Stelle vernachlässigt werden.

Neben der Möglichkeit die eingegebene Nachricht per „*sendButton*“ zu versenden, wurde ebenso die Möglichkeit implementiert, die Nachrichten per Enter-Taste abzuschicken. Um auf die Enter-Taste reagieren zu können, wurde für die *chatInput-TA* ein EventListener für das KeyboardEvent „*KEY_UP*“ definiert und überprüft, ob die Enter-Taste gedrückt wurde. In diesem Fall wird das selbe Event an den *ChatWindowMediator* gesendet, wie beim Betätigen des „*sendButtons*“. Dabei ist es für den Mediator unwichtig zu wissen, wer das Event gesendet hat, da seine Funktionsweise dadurch nicht beeinträchtigt wird. Dieses Vorgehen erleichtert den Umgang mit Nutzerinteraktionen und hilft Code-Redundanz zu vermeiden.

In Listing 5.6 sind vor allem die Daten Bindungen (Databinding) zwischen jeder TA und einer Variable bzw. zwischen Variable und TA auffallend. Der Vorteil der aus diesem Vorgehen resultiert, wird nachfolgend anhand der *chatInput-TA* bzw. der *ChatInputText-Variable* beschrieben. Durch eine Bindung in zwei Richtungen sind die Inhalte der *chatInputText-Varibale* immer identisch mit der *chatInput.text-Variable*. Der Mediator verändert Daten der *chatInput-TA* nur über die *chatInputText-Variable*. Ein direkter Zugriff der Mediatoren auf MXML-Steuerelemente der Views ist somit unnötig. Dadurch werden innerhalb der View Daten von MXML-Elementen getrennt, was eine einfache Anpassung dieser Steuerelemente ermöglicht. Außerdem ist es möglich die Daten der View an mehrere MXML-Komponenten zu binden ohne den Mediator anpassen zu müssen.

Am Beispiel des ChatWindow-Views können zwei wesentliche Vorteile für alle in dieser Diplomarbeit umgesetzten Module abgeleitet werden. Zum einen wird die Code-Menge durch die einheitliche Reaktion des Mediators auf View-Events reduziert und zum anderen greift der Mediator nicht direkt auf MXML-Steuerelemente zu, was eine Anpassung der View-Elemente stark vereinfacht. Listing 5.7 zeigt dies am Beispiel des *ChatWindowMediators*.

**Listing 5.7** ChatWindow.mxml, Kommunikation mit der ChatWindow-View

```
public function ChatWindowMediator( viewComp:Object=null )
{
        ...
        view.addEventListener( ChatWindow.SEND_MESSAGE, onSendMsg );
}
private function onSendMsg( e:Event ):void
{
        sharedObjProxy.writeSO( view.SOName, "message",
                userProxy.user.name + ": " + view.chatInputText + "\n" );

view.chatInputText = "";
}
```

Dieser Code zeigt, dass nur ein EventListener für das Senden der Nachricht eingerichtet wird. Es ist dabei nebensächlich, ob der „*sendButton*“ oder die Entertaste das *„SEND_MESSAGE”-Event* auslöst. Das Schreiben der neuen Nachricht übernimmt der *sharedObjProxy*, welcher in Abschnitt 5.4.2 vorgestellt wird. Wichtig ist an dieser Stelle, dass der *ChatWindowMediator* den Inhalt der *chatInput-TA* über die gebundene *chatInputText-Variable* ausliest. Auch das Zurücksetzen des Textes der *chatInput-TA* kann wegen der Bindung in die Gegenrichtung durch das Setzen der *chatInputText-Variable* auf einen Leerstring erreicht werden. Ein Mediator muss dank solcher Implementierung nicht wissen, welche MXML-Elemente in einer View enthalten sind. Das erleichtert vor allem die Veränderung von MXML-View-Elementen.

### 5.4.2 Implementierungen des Models

Eine detaillierte Beschreibung jedes Models bzw. aller Proxies der umgesetzten Module würden den Rahmen dieser Diplomarbeit übersteigen, weshalb sich diese Darstellung, wie bereits in Abschnitt 5.4.1, auf das Chat-Modul konzentriert. Dazu werden einführend alle verwendeten Chat-Modul-Proxies vorgestellt und deren Zuständigkeit erläutert. Im Anschluss wird auf die Implementierung eines einfachen Proxies des Chat-Moduls eingegangen. Die daraus resultierenden Vorteile sind auf alle andere Proxies der entstandenen RIA übertragbar. Darüber hinaus thematisiert dieser Abschnitt den Umgang mit den von der Anwendung übergebenen Delegate-Klassen (vgl. Abschnitt 5.3.2) und zeigt, wie diese Klassen innerhalb eines Moduls verwendet werden.

Bei der Programmierung des Chat-Moduls wurden vier Proxies erstellt. In Abbildung

5.20 sind die Proxies des Chat-Moduls sowie die wesentlichen Daten, welche von diesen Proxies verwaltet werden, dargestellt.

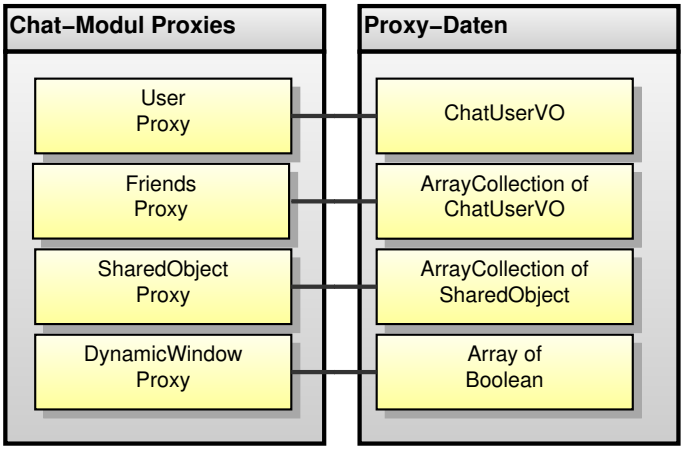


Abbildung 5.20: Chat-Modul PureMVC-Proxies

- UserProxy: Der UserProxy beinhaltet den angemeldeten Benutzer in Form eines ChatUserVO und benachrichtigt das Chat-Modul über Änderungen dieses Nutzers.
- FriendsProxy: Der FriendProxy beinhaltet die Freunde des angemeldeten Nutzers sowie deren Online-Status. Dazu wird eine ArrayCollection mit allen Freunden (ChatUserVO) des angemeldeten Benutzers benötigt. Außerdem ist es Aufgabe des FriendsProxies das Chat Modul über Änderungen der Freunde des angemeldeten Nutzers zu informieren.
- SharedObjectProxy: Der SharedObjektProxy beinhaltet alle benötigten SharedObjects des Chat-Moduls in Form einer ArrayCollection. Darüber hinaus ist er für das Verbinden, Schreiben und Schließen von SharedObjects verantwortlich. Ändert sich ein SharedObject, informiert der SharedObjectProxy die anderen AF-Bausteine des Chat-Moduls.
- DynamicWindowProxy: Der DynamicWindowProxy enthält Informationen über den momentanen Zustand der Chat-Views. AF-Bausteine des Chat-Moduls können über den DynamicWindowProxy erfahren, welche Fenster gerade geöffnet oder geschlossen sind. Dazu wird ein assoziatives Array benötigt, welches mit booleschen Werten gefüllt werden kann.

Jedem Proxy können von einem Mediator, Command oder einem anderen Proxy-Daten übergeben werden. Diese Aktualisierung der Daten löst eine Notification an das PureMVC-System des Moduls aus, welches wiederum auf die Veränderung der Proxy-Daten reagieren kann. Listing 5.8 zeigt dieses Vorgehen am Beispiel des UserProxies.

**Listing 5.8** UserProxy.as

```
public class UserProxy extends Proxy implements IProxy
{
        public static const NAME:String = "ChatUserProxy";
        public function UserProxy( )
        {
                super( NAME, new ChatUserVO() );
        }
        public function get user():ChatUserVO
        {
                return data as ChatUserVO;
        }
        public function set user( user:ChatUserVO ):void
        {
                data = user;
                sendNotification( ChatModuleFacade.USER_UPDATED, user );
        }
}
```

Die Daten der implementierten Proxies können lediglich über *getter-* bzw. *setter-Funktionen* erfragt oder verändert werden. Gerade durch den Einsatz von *setter-Funktionen* ist es möglich, das System durch Senden einer Notification über die Veränderung von Proxy-Daten zu informieren. Im praktischen Teil dieser Arbeit wurde sich dazu entschieden, die aktualisierten Daten zusammen mit der Änderungs-Notification (*ChatModuleFacade.USER_UPDATED*) zu senden. Der Vorteil dieses Vorgehens wird auf Seite des Empfängers einer solchen Notification deutlich. Die Empfänger können die veränderten Daten direkt aus der Notification abrufen und müssen nicht extra den Proxy über die Änderungen befragen. Das hilft dabei, Code zu reduzieren und komfortabel auf die veränderten Daten zugreifen zu können. Neben diesem grundlegenden Aufbau der Proxies, wurden bei der Umsetzung der Anforderungen dieser praktischen Arbeit weitere Funktionalitäten in die Proxies implementiert. Im Wesentlichen handelt es sich dabei um Funktionen, welche die Daten in den Proxies verändern oder bestimmte Inhalte der Proxies zur Verfügung stellen. Zum Beispiel kann der FriendsProxy befragt werden, ob ein Nutzer zu den Freunden des angemeldeten Benutzers gehört oder nicht.

Zur Beschaffung der benötigten Modul Daten, welche in den Proxies gespeichert sind, werden meist Commands beauftragt. Commands nutzen die von der Anwendung übergebenen Delegates zur Kommunikation mit den externen Datenquellen wie z.B. Webservern. Nachdem ein Command Server-Daten erhalten hat, speichert es diese in einem Proxy ab. Der Proxy wiederum sendet nach dieser Datenaktualisierung eine Notification an das Modul. Dieses Vorgehen ist typisch für die umgesetzte RIA und wird deshalb vereinfacht in Abbildung 5.21 dargestellt.

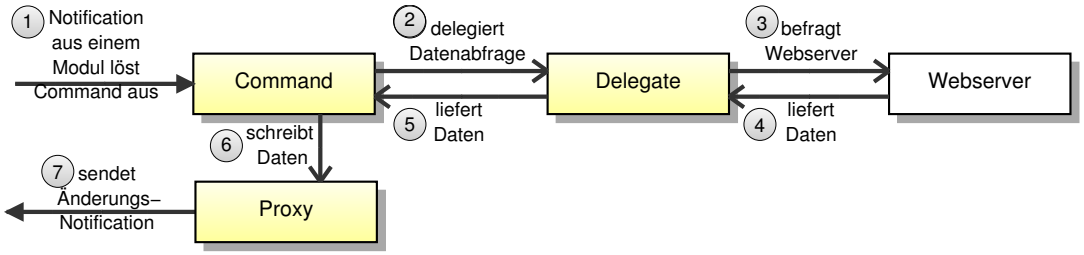


Abbildung 5.21: Kommunikation mit externen Datenquellen (vereinfacht)

Damit ein Command ein Delegate mit der Beschaffung von externen Daten beauftragen kann, muss ein Delegate-Objekt vorhanden sein. Die Anwendung übergibt dem Modul allerdings keine Delegate-Objekte sondern lediglich Delegate-Klassen (vgl. Abschnitt 5.3.2), weshalb zunächst ein Objekt der Delegate-Klasse erzeugt werden muss. Dazu wurde in dieser praktischen Arbeit eine Delegate-Fabrik eingeführt, welche beliebig viele Delegate-Objekte einer Delegate-Klasse zur Programmlaufzeit erzeugen kann. Damit jeder AF-Baustein eines Moduls auf die Funktionalität der Delegate-Fabrik zugreifen kann, ist es notwendig, dass diese zentral zugänglich ist. Da auf die PureMVC-Facade ohnehin von jedem AF-Baustein eines Moduls zugegriffen werden kann, wurde die Delegate-Fabrik dort integriert. Dafür wurde die PureMVC-Facade abgeleitet und um eine Delegate-Fabrik erweitert. Wird von einem Command ein Delegate benötigt, befragt das Command die Delegate-Fabrik, welche sich in der Facade des Moduls befindet, nach einem Delegate-Objekt. Aufbauend auf Abbildung 5.21 zeigt die Abbildung 5.22 die tatsächlichen Vorgänge während der Datenbeschaffung für den FriendsProxy durch das UserGetFriendsCommand im Chat-Modul.

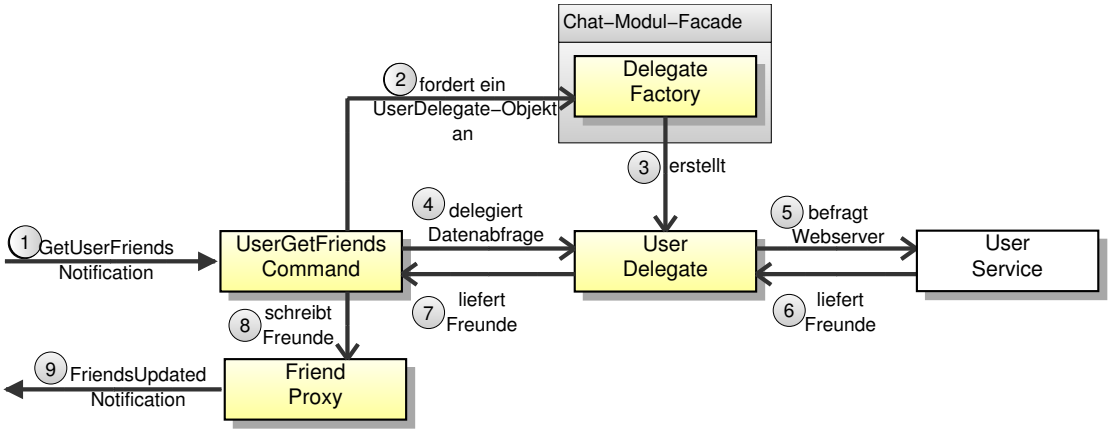


Abbildung 5.22: Kommunikation mit externen Datenquellen (tatsächlich)

Listing 5.9 verdeutlicht dieses Vorgehen durch die *execute-* und *result-Funktion* des *UserGetFriendsCommand*.

**Listing 5.9** UserGetFriendsCommand.as

```
override public function execute(notification:INotification):void
{
        var user:ChatUserVO = notification.getBody() as ChatUserVO;
        var delegate:IUserDelegate =
                (facade as FacadeBase).factory.getUserDelegate( this );

        delegate.getFriends( user.name, user.password );
}
public function result(data:Object):void
{
        var friendsProxy:FriendsProxy =
                facade.retrieveProxy( FriendsProxy.NAME ) as FriendsProxy;

        friendsProxy.friends = ...( data.result as Array );
}
```

Dieser Code zeigt, wie das *UserGetFriendsCommand* auf die Fabrik (*factory*), welche sich in der Facade des Moduls (*facade*) befindet, zugreift. Das Command erwartet von der Fabrik ein Objekt zurück, welches das *IUserDelegate* Interface implementiert. In Abschnitt 5.3.2 wurde der Zusammenhang zwischen *IUserDelegate* und UserDelegate bereits verdeutlicht und ist zusätzlich noch in Abbildung 5.11 dargestellt. Nachdem das Command das Delegate-Objekt erhalten hat, beauftragt es das Delegate mit der Beschaffung der Freunde für den Benutzer. Ist die Abfrage durch das Delegate erfolgreich,

wird das Ergebnis im *friendsProxy* gespeichert.

### 5.4.3 Implementierungen des Controllers

In diesem Abschnitt soll die Umsetzung des Controllers bzw. der Commands während der praktischen Arbeit am Beispiel der Chat-Modul Commands dargelegt werden. Dazu werden einleitend die Chat-Modul-Commands in Gruppen unterteilt, um anschließend diesen Gruppen, die in dem Chat-Modul eingesetzten Commands zuzuordnen und deren Funktionalität zu erläutern.

Für die Erstellung der Chat-Modul Funktionalität wurden elf Commands implementiert. Diese Commands können in vier Gruppen unterteilt werden.

- Commands zur Initialisierung: Diese Gruppe von Commands ist notwendig, um das PureMVC-Modul zu initialisieren. Dabei werden Proxies und Mediatoren an der Modul Facade angemeldet.
- Commands zum Erzeugen der Modul Views: Diese Commands erzeugen die Views des Chat-Moduls und senden diese an die übergeordnete Anwendung zurück.
- Commands zum Umgang mit SharedObjects: Commands, welche dieser Gruppe zugeordnet werden, dienen dem komfortablen Umgang mit SharedObjects.
- Benutzerabhängige Commands: Diese Command sind für solche Aufgaben zuständig, welche abhängig vom angemeldeten Benutzer sind. Das sind vor allem Commands, welche Webservices nach Benutzerdaten befragen.

Das Chat-Modul sowie alle anderen Module benötigen drei Commands zur Initialisierung: *StartUpCommand, InitMediatorCommand* und *InitProxyCommand*. Das *StartUpCommand* ist ein Macro-Command, welches die beiden anderen Commands (*InitMediatorCommand, InitProxyCommand*) enthält. Es wird ausgelöst, nachdem das Modul fertig generiert wurde. Die beiden anderen Commands melden Mediatoren bzw. Proxies an der Modul-Facade an.

Für das Erzeugen von Modul Views werden beim Chat Modul zwei Command benötig: *CreateStdUICommand* und *CreateNewChatWindowCommand*. Nachdem die Anwendung das Modul verbunden hat, wird das *CreateStdUICommand* ausgelöst, welches alle statischen Views an die Anwendung zurücksendet. Dieses Command findet man bei jedem in dieser Diplomarbeit umgesetzten Modul. Einzigartig beim Chat-Modul ist

allerdings das *CreateNewChatWindowCommand*, welches dafür zuständig ist, ein neues Chat-Fenster zur Kommunikation mit einem Freund zu erstellen und dieses zur Laufzeit an die Anwendung zurückzugeben.

Eine weitere Gruppe von Commands bilden die Commands zum Umgang mit SharedObjects. Dafür wurden zwei Commands benötigt: *SharedObjectCreateCommand* und *SharedObjectClientListChangedCommand*. Das *CreateSharedObjectCommand* ist nötig, um beliebig viele verschiedene SharedObjects beim *SharedObjectProxy* anzumelden. Das *SharedObjectClientListChangedCommand* hingegen behandelt die Veränderungen eines SharedObjekts, welches alle angemeldeten Benutzer verwaltet. Dieses Command wird vom *SharedObjectProxy* ausgelöst, wenn dieser eine Veränderung im SharedObject für die Clientliste registriert.

Die letzte Gruppe bilden die vom angemeldeten Benutzer abhängigen Commands. Zu diesen Commands gehören: *UserGetFriendsCommand, UserAddFriendCommand, UsersSearchByNameCommand* und *UserUpdatedCommand*. Das *UserGetFreindsCommand* wurde bereits im Listing 5.9 ausführlich vorgestellt und wird dazu genutzt, die Freunde des angemeldeten Benutzers vom Webserver zu besorgen. Wie der Name vermuten lässt, fügt das *UserAddFriendCommand* dem angemeldeten Nutzer einen neuen Freund hinzu. Das *UserUpdatedCommand* wird ausgelöst, wenn der angemeldete Benutzer im *UserProxy* geändert wurde. Daraufhin werden die Freunde des neuen Nutzers durch das *UserGetFreindsCommand* aktualisiert.

Bei allen Modulen wurde darauf geachtet, dass für einen besseren Überblick die Namen der Commands entsprechend ihrer Funktion vergeben wurden. Die Gruppenzugehörigkeit wird durch einen Gruppenpräfix symbolisiert. So heißt zum Beispiel das Command zum Erstellen eines SharedObjects nicht *CreateSharedObjectCommand* sondern *SharedObjectCreateCommand*. Da relativ viele Commands in Modulen enthalten sind, erleichtert dieses Vorgehen die Orientierung, was wiederum die Übersichtlichkeit eines Moduls erhöht.

Der Einsatz eines Commands ist relativ schwer zu pauschalisieren (vgl. Abschnitt 4.2.2). Im Rahmen dieser Diplomarbeit wurde dennoch ein einheitliches Vorgehen festgelegt. Ein Command wird demnach immer dann erstellt, wenn die zu implementierende Logik:

- auf mehrere AF-Bausteine zugreift: Ist es z.B. notwendig, dass mehrere Daten von verschiedenen Proxies/ Mediatoren zum Umsetzen einer Aufgabe benötigt werden, wird ein Command verwendet.

- den Zugriff auf externe Server beinhaltet: Im Rahmen dieser Diplomarbeit sollen nur Commands über Delegates auf externe Datenquellen zugreifen.
- in einem Mediator über die Zuständigkeit dieses Mediators hinausgeht: Die Zuständigkeit des Mediators beschränkt sich auf die Manipulation der zugeordneten View und auf das Vermitteln zwischen dieser View und dem restlichen PureMVC-Modul.
- in einem Proxy über die Zuständigkeit dieses Proxies hinausgeht: Proxies beinhalten Daten, sind zuständig, diese Daten dem PureMVC-System zur Verfügung zu stellen und bieten eine Schnittstelle zur Datenmanipulation durch andere AF-Bausteine.
- mehr als zehn Zeilen Code benötigt: Übersteigt die Implementierung der Funktionalität mehr als ca. zehn Zeilen Code, wird dieser Code in ein Command ausgelagert.

Durch das Einhalten dieser einfachen Regeln, wurde die wesentliche Logik des Moduls in Commands implementiert. Dadurch können Veränderungen der Modul Logik unabhängig von anderen AF-Bausteinen komfortabel durchgeführt werden.

### 5.4.4 Implementierungen der Erweiterung Pipes&Filter

Bereits in Abschnitt 5.3.2 wurde aufgezeigt, warum es Aufgabe der Anwendung ist, Module zu verbinden und warum Filter bei der Pipe-Kommunikation eingesetzt werden. Außerdem zeigte dieser Abschnitt, wie durch das Versenden einer Notification (vgl. Listing 5.4) innerhalb eines Moduls bzw. der Anwendung automatisiert eine Nachricht über eine Pipe versendet wird.

In diesem Abschnitt soll nun gezeigt werden, wie diese Automatisierung innerhalb der Module umgesetzt wurde und wie Pipe-Nachrichten empfangen werden. Dabei wird genauer auf Pipe-Nachrichten eingegangen und es wird gezeigt, wie der Nachrichten-Header eingesetzt werden kann. Wie bereits bei der Beschreibung von Model, View und Control, werden, sofern nötig, diese Erklärungen am Beispiel des Chat Moduls verdeutlicht.

Nachfolgend wird der Begriff „Kern“ verwendet. Unter einem Kern wird ein abgeschlossenes PureMVC-System, also ein PureMVC-Modul oder die PureMVC-Anwendung selbst

verstanden.

Bereits in Abschnitt 5.4.1 wurde der *ChatModuleJunctionMeditor* kurz erwähnt. Jedes Modul sowie die Anwendung selbst besitzt solch einen Junction Mediator. Dieser Mediator vermittelt nicht zwischen Modul-Views und anderen AF-Bausteinen eines Kerns, sondern zwischen verschiedenen PureMVC-Kernen. Diese Junction Mediatoren können, wie auch jeder andere Mediator, Notifications aus ihrem PureMVC-Kern empfangen oder senden. Zusätzlich senden und empfangen diese Junction Mediatoren auch Pipe-Nachrichten von anderen Kernen.

Wie bereits erwähnt, wurde sich im Rahmen dieser praktischen Arbeit dazu entschieden, das Senden einer Pipe-Nachricht durch das Senden einer Notification auszulösen. Der jeweilige Junction Mediator fängt diese Notification und sendet die in der Notification enthaltene Nachricht an alle angeschlossenen Pipes. Nachdem ein anderer Junction Mediator die Pipe-Nachricht erhalten hat, kann dieser seinerseits eine Notification an seinen Kern versenden. Abbildung 5.23 verdeutlicht dieses Vorgehen.

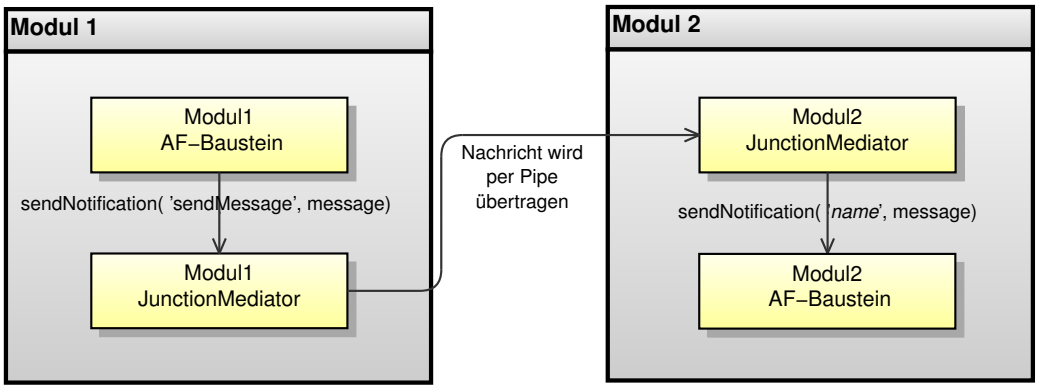


Abbildung 5.23: Pipe-Nachrichten und Notifications

Um das Senden eine Pipe-Nachricht durch eine Notification auslösen zu können, wurde der originale PureMVC Junction Mediator erweitert und alle Junction Mediatoren der Module von diesem abgeleitet. Das Senden einer Pipe-Nachricht ist in Listing 5.10 dargestellt, welches die Reaktion der Junction Mediator Basisklasse auf den Erhalt der *SEND_MESSAGE-Notification* zeigt.

**Listing 5.10** JunctionMediator.as, Senden einer Pipe-Nachricht

```
private var outputPipeNames:Array = new Array();
override public function handleNotification( note:INotification ):void
{
        switch( note.getName() )
        {
                case JunctionMediator.SEND_MESSAGE:
                        ...
                        var message:Message = note.getBody() as Message;
                        for each( var pipename:String in outputPipeNames )
                                junction.sendMessage( pipename, message );
                                break;
                                ...
                        }
}
```

Wie in diesem Listing zu erkennen ist, befindet sich in der *JunctionMediator*-Klasse ein Array (*outputPipeNames-Variable*), das die Pipe-Namen der ausgehenden Pipes enthält. Dieses Array wurde bereits beim Verbinden des Moduls durch die Anwendung gefüllt. Durch das Aufrufen der Junction *sendMessage-Funktion* für jeden dieser Pipe-Namen, wird die in der *SEND_MESSAGE-Notification* enthaltenen Nachricht (*message*) versendet. Dieser Junction ist originaler Bestandteil der PureMVC-Pipes&Filter-Erweiterung und wird deshalb nicht näher erläutert.

Durch dieses Vorgehen ergeben sich zwei wesentliche Vorteile. Zum einen ist es durch das Senden der *SEND_MESSAGE-Notification* sehr einfach von einem beliebigen AF-Baustein eines PureMVC-Kerns eine Pipe-Nachricht zu versenden. Zum anderen ist es nicht erforderlich, dass dieser AF-Baustein weiß, wer diese Nachricht empfangen muss. Die Nachricht wird einfach an alle angeschlossenen Module versendet und die angeschlossenen Module bzw. die Anwendung entscheiden selbst, auf welche Pipe-Nachrichten sie reagieren wollen. Da eine Referenz dieser Nachricht versendet wird, ergeben sich dadurch keine Performance-Probleme.

Aber wie genau werden Pipe-Nachrichten von einem Junction Mediator empfangen? Diese Funktionalität wurde im Rahmen dieser Diplomarbeit nicht in die Basisklasse (*JunctionMediator*) sondern in die jeweiligen abgeleiteten Klassen integriert. Alle eingehenden Pipe-Nachrichten werden somit von der handlePipeMessage-Funktion eines verbundenen Junction Mediators empfangen. In Listing 5.11 ist diese Funktion des vom *JunctionMediators* abgeleiteten *ChatModulJunctionMediators* dargestellt.

**Listing 5.11** ChatModulJunctionMediators.as, Empfangen einer Pipe-Nachricht

```
override public function handlePipeMessage( message:IPipeMessage ):void
{
        var msgHeader:MessageHeader = message.getHeader()
                as MessageHeader;

        if( msgHeader.type == MessageConstants.SYSTEM )
                handleSystemMessage(message, msgHeader );

        if( msgHeader.type == MessageConstants.USER )
                handleUserMessage( message, msgHeader );
}
```

Diese Funktion kann analog der *handleNotification-Function* eines View-Mediators betrachtet werden. Anders ist nur, dass hier keine Notifications, sondern Pipe-Nachrichten behandelt werden. Wie man aus dem Listing erkennen kann, wird anhand des Nachrichten-Header (*msgHeader-Variable*) entschieden, wie die eingegangene Nachricht innerhalb des Kerns weiter verarbeitet wird.

Dieser Nachrichten-Header wurde im Rahmen dieser Diplomarbeit eingeführt und ist Teil einer gesendeten Nachricht. Dadurch können zusätzliche Informationen mit der Nachricht übertragen werden. Dabei wird zwischen folgenden Nachrichten-Header Informationen unterschieden: Typ, Name und Name des Nachrichtenerzeugers. Diese Kategorisierung der Nachrichten erleichtert den Umgang mit erhaltenen Nachrichten und gestaltet den Code übersichtlich. Wie in Listing 5.11 zu sehen ist, können z.B. System-Nachrichten und User-Nachrichten in verschiedenen Funktionen behandelt werden.

# 6 Zusammenfassung

## 6.1 Zusammenfassung Theorie

Ein Ziel des theoretischen Teils dieser Diplomarbeit war es Softwarearchitektur, Implementierung der Architektur, Software Muster und Architektur-Frameworks miteinander in Beziehung zu setzen. Dabei wurde aufgezeigt, dass gerade der Übergang zwischen SA und Implementierung von verschiedenen Autoren unterschiedlich betrachtet und damit nicht eindeutig festgelegt werden kann. Der Zusammenhang zwischen SA und AF in Bezug auf Flex wurde mit Hilfe der Software-Muster hergestellt.

Das wesentliche Ziel des theoretischen Teiles war es jedoch, AF zu untersuchen, welche in Flex eingesetzt werden können. Nachdem die Funktionsweise erläutert wurde, zeigte die Auswertung des Vergleichs der AF Cairngorm, PureMVC und MATE überraschende Ergebnisse (vgl. Abschnitt 4.4). Nachdem verdeutlicht wurde, wie unterschiedlich die AF aufgebaut sind, war zu erwarten, dass die Gesamtpunktzahlen für die einzelnen AF stark voneinander abweichen. Mit 3 (Cairngorm), 3,5 (PureMVC) und 3,56 (MATE) Punkten stellte sich jedoch heraus, dass die Gesamtbewertung, trotz der großen Unterschiede im Aufbau, recht ähnlich ausfällt. Damit ein AF für die praktische Arbeit ausgewählt werden konnte, musste die Bewertung der gewählten Kriterien (Zugänglichkeit, Entwicklungseffizienz, Wartbarkeit) eingehender untersucht werden. Dabei stellte sich heraus, dass jedes AF für verschiedene Projekte durchaus sinnvoll einsetzbar ist. Tabelle 6.1 zeigt zusammengefasst die wesentlichen Stärken (grün) und Schwächen (rot) der einzelnen AF.

| | Cairngorm | PureMVC | MATE |
|---|---|---|---|
| | | | |
| geringe Komplexität | 4 von 5 | 1 von 5 | 3 von 5 |
| Code Aufwand | 1 von 5 | 3 von 5 | 4 von 5 |
| Wiederverwendung | 1 von 5 | 4 von 5 | 2 von 5 |
| Lesbarkeit | 4 von 5 | 1 von 5 | 3 von 5 |
| Veränderbarkeit | 3 von 5 | 5 von 5 | 4 von 5 |

Tabelle 6.1: Stärken und Schwächen von Cairngorm, PureMVC und MATE

Für den praktischen Teil der Arbeit wurde sich nach eingehender Betrachtung des jeweiligen AF für den Einsatz des PureMVC AF entschieden. Besonders im Vergleich zu Cairngorm und MATE fielen die guten Voraussetzungen einer Modulimplementierung in Verbindung mit guter Wartbarkeit und Wiederverwendbarkeit auf. Diese Vorteile wirken sich positiv auf die gesetzten Ziele der praktischen Umsetzung aus. Die weitaus schlechtere Lesbarkeit des PureMVC-Codes und die hohe Komplexität des PureMVC-AF wurden für diese Vorteile in Kauf genommen.

## 6.2 Zusammenfassung Praxis

Ziel des praktischen Teils war es, eine Flex RIA-Anwendung zu erstellen, welche eigenständige und wieder verwendbare Module enthält. Dazu wurde das im theoretischen Teil ausgewählte AF PureMVC in Flex eingesetzt. In der Praxis wurden verschiede Konzepte zum Umgang mit den Modulen erarbeitet. Dabei wurde gezeigt, wie Module erzeugt, angezeigt und miteinander verbunden werden. Außerdem wurde erarbeitet, wie externe Datenquellen in Module injiziert werden können.

Ein weiteres Ziel der Praxis was es, die Module selbst einfach wartbar umzusetzen. Dazu wurde in der Beschreibung der praktischen Arbeit verdeutlicht, wie die Technologie Flex mit dem ausgewählten AF PureMVC hinsichtlich der Modul View Elemente harmoniert. Außerdem wurden einheitliche Richtlinien zum Einsatz von Commands erarbeitet und der wesentliche Aufbau aller Proxies aufgezeigt. Die Ziele des praktischen Teils dieser Diplomarbeit wurden somit erreicht.

Dennoch kann die entstandene Anwendung lediglich als Prototyp verstanden werden. Wesentliches Ziel des praktischen Teils war es, Konzepte und Richtlinien zum Umgang mit Modulen bzw. den Aufbau der Module selbst zu erarbeiten. Dabei entstanden

funktionsfähige, beispielhafte Module. Um einen Einsatz in der Realität zu ermöglichen, ist es dennoch notwendig einige Funktionalitäten der Module zu ergänzen. Zum Beispiel wäre es von großem Vorteil das Chat Modul mit einer Konferenz Erweiterung auszustatten um sich besser von der bereits am Markt etablierten Konkurrenz abzuheben. Der serverseitige Code wurde so konzipiert, dass die Anbindung an eine beliebige Datenbank problemlos möglich ist.

## 6.3 Theorie vs. Praxis

Wie gerade beschrieben, wurden die Zielsetzungen des theoretischen sowie des praktischen Teils weitgehend erreicht. Dennoch soll in dieser abschließenden Betrachtung gezeigt werden, in wie weit sich die theoretischen Bewertungen des PureMVC AF in der Praxis bestätigt haben. Die praktischen Erfahrungen führen zu verschiedenen Veränderungen in der Bewertung der Entwicklungseffizienz des PureMVC AF, welche in Tabelle 6.2 verdeutlicht werden. Eine grün eingefärbte Bewertung zeigt eine Verbesserung und eine rote eine Verschlechterung des theoretischen Werts.

| | **Theorie** | **Praxis** |
|---|---|---|
| **Entwicklungseffizienz** | **3,2 von 5** | **3,2 von 5** |
| Code Aufwand | 3 von 5 | 2 von 5 |
| Vorteile durch klare Modularisierung | 2 von 5 | 2 von 5 |
| Umgang externe Datenquellen | 3 von 5 | 4 von 5 |
| verfügbare Erweiterungen | 4 von 5 | 4 von 5 |
| Wiederverwendung | 4 von 5 | 4 von 5 |

Tabelle 6.2: PureMVC Theorie vs. Praxis

Die in der Theorie erarbeiteten Bewertungen haben sich während der praktischen Arbeit weitgehend bestätigt. Bei den Kriterien Zugänglichkeit und Wartbarkeit mussten keine und beim Kriterium Entwicklungseffizienz nur marginale Änderungen vorgenommen werden. Die Verschlechterung des Code-Aufwands resultiert dabei aus dem Einsatz der PureMVC Module. Bevor ein neues Modul mit Funktionalität ausgestattet werden kann muss jedes Mal ein neues Modulgrundgerüst implementiert werden. Das führt zu erhöhtem Code-Aufwand. Diesem Problem könnte durch die Erstellung eines Flex-Builder-Plugins entgegengewirkt werden, welches ein leeres Modul selbstständig erzeugt (Code Generator).

Positiv hingegen wirkt sich der Einsatz des Konzeptes zum Injizieren von Datenquellen aus (vgl. Abschnitt 5.3.2).

## 6.4 Ausblick

Die Vernetzung der Gesellschaft wird in Zukunft vermutlich noch weiter zunehmen. Betrachtet man die aktuelle Entwicklung, fällt auf, dass auch auf mobilen Endgeräten Internetanwendungen möglich sind. Soziale Netzwerke, Stadtpläne oder die aktuellen Aktienkurse sind schon heute, z.B. durch das Apple IPhone, jederzeit mobil abrufbar. Sehr wahrscheinlich werden ständige Internetzugriffe durch mobile Endgeräte oder Desktop-Anwendungen in naher Zukunft selbstverständlich sein.

Dabei werden die Technologien kontinuierlich weiterentwickelt. Die neue Flex Version 4.0[1] steht derzeit kurz vor der offiziellen Veröffentlichung. Mit dieser Version ist es zum Beispiel möglich AIR 2.0 Anwendungen zu erzeugen, welchen neben dem Desktop auch auf dem IPhone lauffähig sind. Adobe legt mit Flex 4 seinen Fokus dabei auf GUI und Design, Entwickler-Produktivität und Framework-Evolution.

Diese offiziellen Ziele bzw. Eigenschaften des neuen Flex 4.0 Application-Frameworks sind von Adobe natürlich nicht zufällig ausgewählt. Vielmehr lassen sich daraus wesentliche Problemfelder der nahen Zukunft im Bereich der Web-Entwicklung ableiten.

Diese resultieren vor allem aus den zukünftig zu erwartenden hohen Ansprüchen der RIA Benutzer. Diese hohen Ansprüche sind sowohl im Bereich der GUI (Design, Bedienbarkeit) aber auch im Bezug auf den Inhalt von Web-Anwendungen zu erwarten (z.B. 3D Grafiken, HD-Video). RIA-Technologien, welche zukünftig eingesetzt werden, müssen demnach sowohl flexible Konzepte zum Erstellen von Benutzerschnittstellen, aber auch eine zeitgemäße technische Basis bereitstellen. Außerdem sollten RIA-Technologien zukünftig Möglichkeiten zur Produktivitätssteigerung bereitstellen, um der steigenden Komplexität der Web-Anwendungen entgegenwirken zu können. Flex ist dabei auf einem guten Weg, aber der Konkurrenzkampf gerade im Bezug auf Silverlight wird in naher Zukunft noch hart ausgefochten werden.

Unabhängig davon, welche Technologien sich im Web-Bereich durchsetzten, ist zu erwarten, dass die Software-Komplexität weiter ansteigt. Software-Architektur oder Architektur-Frameworks werden deshalb in Zukunft vermutlich eine noch größere Rolle

[1] http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+4

spielen. Ist die Software-Architektur derzeit in Form von Architektur-Frameworks wie Cairngorm und PureMVC deutlich an die Vorbilder aus den Desktop-Bereich angelehnt, zeigen neuere AF wie MATE oder Swiz bereits den Trend hin zu einer Web-Technologie-abhängigen Software-Architektur. Dabei ist gerade bei zukünftigen Flex-AF die Nutzung von Inversion of Control Prinzipen und die vollständige Ausnutzung von Flex Features im Bezug auf die SA zu erwarten. Einen anderen Weg geht das derzeit in den Startlöchern stehende Cairngorm 3.0. Die neue Cairngorm Version ist kein AF, sondern im wesentlichen eine Sammlung von mehr oder minder nützlichen Bibliotheken, welche auch in anderen AF eingesetzt werden können. Es bleibt abzuwarten ob diese Strategie von Adobe angemessen ist.

# A Cafe Townsend

Die „Cafe Townsend“ Anwendung wurde ursprünglich als Beispiel für das Cairngorm-AF entwickelt. Diese Beispielanwendung ist einfach nachzuvollziehen, aber komplex genug um die Funktionsweise eines AF darzustellen. Aus diesem Grund wurde das Programm „Cafe Townsend“ mit Hilfe verschiedener AF implementiert, um nachvollziehbar in das jeweilige AF einzuführen. Ziel dieser Anwendung ist die beispielhafte Verwaltung von persönlichen Daten der Cafe-Mitarbeiter. Nach Benutzeridentifizierung (vgl. Abbildung A.1) werden alle im System gefundenen Mitarbeiter in Form einer Liste angezeigt und können in einer gesonderten Ansicht erstellt, gelöscht oder verändert werden (Abbildung A.2. Alle Ansichten enthalten ein „Café Townsend“-Banner und eine CSS-Datei, welche das Design der Anwendung definiert.

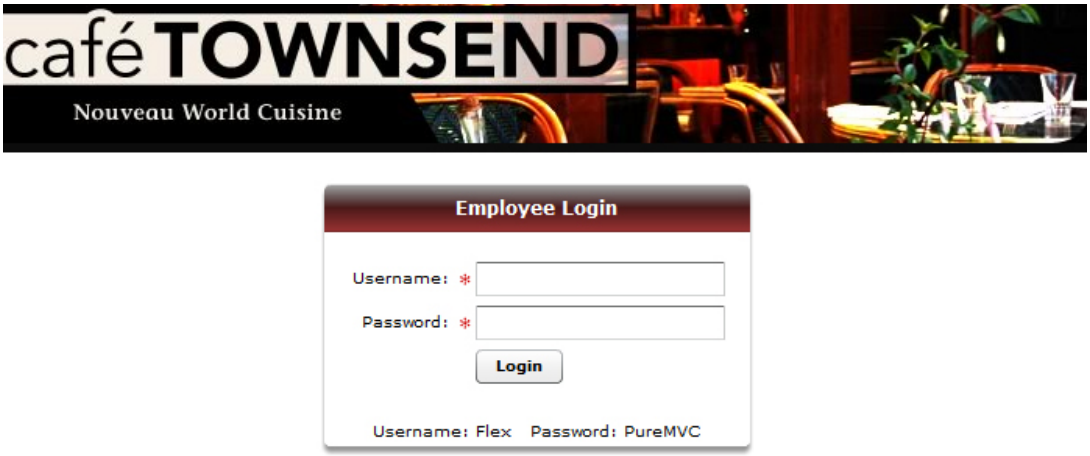


Abbildung A.1: Café Townsend Login

Die Umsetzung der „Cafe Townsend“-Anwendung mit Hilfe der jeweiligen AF dient lediglich der grundlegenden Einführung in die verschiedenen SA-Ansätze. Daher sind einige Prozesse, wie z.B. die Authentifizierung von Nutzern oder die Datenquelle für Mitarbeiterdaten, vereinfacht dargestellt. Beschrieben werden lediglich Programmteile,

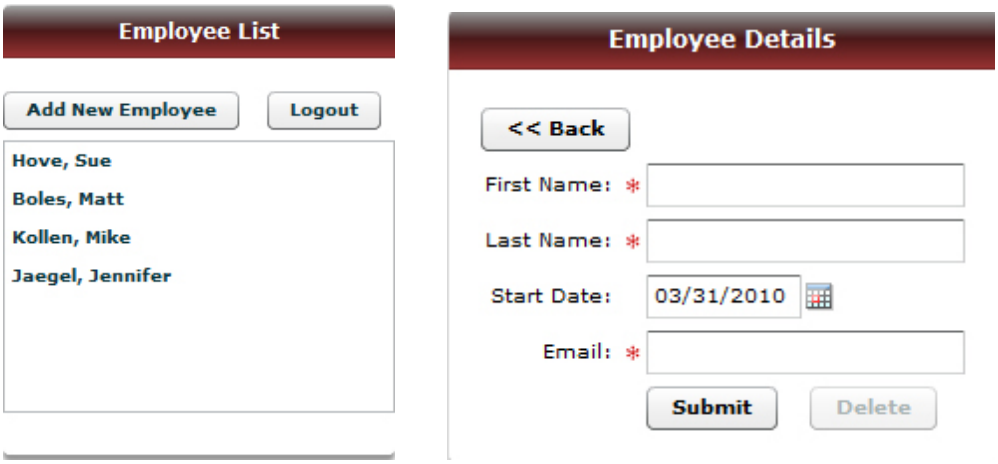


Abbildung A.2: Employee List und Employee Details

die für das Verständnis des jeweiligen AF erforderlich sind und nicht der gesamte Programmcode. Bei den verschiedenen AF wird neben dem Initialisierungsprozess des AF besonders auf den Login-Vorgang sowie auf die Beschaffung externer Daten eingegangen.

Am effektivsten ist es dabei, der jeweiligen Erklärung an Hand der nummerierten Abbildungen zu folgen. Außerdem ist es angebracht die jeweiligen „Cafe Townsend“-Anwendungen in den Flex Builder zu importieren, um ggf. den gesamten Code untersuchen zu können.

## A.1 Cairngorm „Cafe Townsend“

Um die Funktionsweise einer Cairngorm-Anwendung besser zu verdeutlichen, wird nachfolgend die in Anhang A beschriebene „Cafe Townsend“ Anwendung mit dem Cairngorm-AF umgesetzt. Die Erläuterungen beziehen sich auf den veröffentlichten Cairngorm „Cafe Townsend“ Code[1]. Eine ausführliche Darstellung des Codes würde an dieser Stelle aber viel zu weit führen, weshalb diese Arbeit nur die Teile des Beispielprogramms erläutert, die zum grundlegenden Verständnis des Cairngorm-AF beitragen.

Zur Initialisierung werden FrontController (*AppController*) und ServiceLocator (*Services*) als MXML-Komponenten auf der obersten Ebene einer Anwendung, in diesem Falle also in der *Main.mxml* (Haupt MXML), eingefügt (Listing A.1).

[1] http://cairngormdocs.org/blog/?p=19

**Listing A.1** Main.mxml, Anlegen von FrontController und ServiceLocator

```
<business:Services id="services" />
<control:AppController id="appController" />
```

Die „Cafe Townsend“ Anwendung ist in drei Ansichten bzw. Views (EmployeeLogin, EmployeeList, EmployeeDetail) unterteilt, welche mittels einer ViewStack-Steuerkomponente in der Haupt-MXML umgeschaltet werden (vgl. Listing A.8).

Der Cairngorm Funktionsüberblick soll anhand des Login-Vorgangs dargestellt werden (Abbildung A.3). Den Startpunkt des Login-Vorgangs bildet die Betrachtung der EmployeeLogin-View, also der Datei *EmployeeLogin.mxml*.

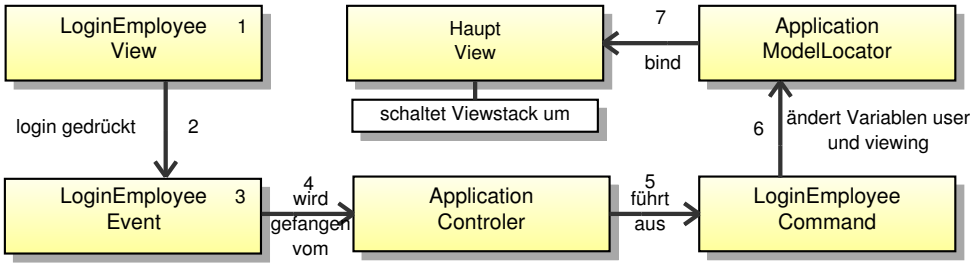


Abbildung A.3: Cairngorm, Login

(1) Reduziert man die Steuerelemente der *EmployeeLogin.mxml* Datei auf das wesentliche, erkennt man drei GUI- Elemente. Jeweils ein Textinput für Nutzername und Passwort sowie einen Button zum Abschicken der Login-Anfrage (Listing A.2).

**Listing A.2** EmployeeLogin.mxml, MXML-Elemente

```
<mx:TextInput id="username" />
<mx:TextInput id="password" displayAsPassword="true" />
<mx:Button id="loginBtn" label="Login" click="loginEmployee()"/>
```

(2) Durch das Betätigen des Login-Buttons (*loginBtn*) wird die Funktion „*loginEmployee()*“ (Listing A.3) der *EmployeeLogin.mxml* aufgerufen. Diese Funktion erzeugt im wesentlichen eine Instanz des *LoginEmployeeEvents*. Beim Erzeugen dieses Events werden die Daten für Nutzernamen und Passwort dem Event übergeben und dort abgespeichert. Abschließend wird das Event mittels CairngormEventDispatcher (Singleton) gesendet.

**Listing A.3** EmployeeLogin.mxml, loginEmployee-Funktion

```
private function loginEmployee() : void
{
        ...
        var cgEvent:LoginEmployeeEvent =
                new LoginEmployeeEvent( username.text, password.text );

        CairngormEventDispatcher.getInstance().dispatchEvent( cgEvent );
        ...
}
```

(3) Das *LoginEmployeeEvent* (Listing A.4), abgeleitet vom Cairngorm Event, enthält die View Daten für Nutzername und Passwort. Es ist außerordentlich wichtig ein Cairngorm-Event eindeutig zu benennen, damit der FrontController dieses Event einem Command zuordnen kann. Mit der Übergabe eines Namens (*AppController.LOGIN_ EMPLOYEE_EVENT*) an den Basisklassen-Konstruktor im *LoginEmployeeEvent*-Konstruktor wird der Event Name zugeordnet.

**Listing A.4** LoginEmployeeEvent.as

```
public class LoginEmployeeEvent extends CairngormEvent
{
        public var username:String;
        public var password:String;

        public function LoginEmployeeEvent( username : String,
                password : String )
        {
                super(AppController.LOGIN_EMPLOYEE_EVENT);
                this.username = username;
                this.password = password;
        }
}
```

(4) Der FrontController (*AppController*) kann nunmehr die „Weiterleitung“ vom *LoginEmployeeEvent* an das *LoginEmployeeCommand* durch Hinzufügen einer *addCommand-Funktion* realisieren. Listing A.5 zeigt die entsprechende Umsetzung im *FrontController*. Dazu wurde der Cairngorm-FrontController dieser Anwendung zu einem *AppController* abgeleitet. Das geschieht in allen Cairngorm-Anwendungen, da es dadurch möglich ist, die Funktionalität des Cairngorm-FrontControllers zu nutzen und die programmspezifischen Event-Command-Verbindungen zu ergänzen.

**Listing A.5** AppControler.as, Event-Command-Verknüpfung

```
public class AppController extends FrontController
{
        public function AppController()
        {
                addCommand( AppController.LOGIN_EMPLOYEE_EVENT,
                        LoginEmployeeCommand );
                ...
        }
}
```

(5) Nach dem Erhalt des *LoginEmployeeEvents* löst der *AppController* die execute-Funktion (Listing A.6) des *LoginEmployeeCommand* aus und übergibt diesem automatisch das vollständige *LoginEmployeeEvent*. Somit ist es dem Command möglich, die im *LoginEmployeeEvent* enthaltenen Daten zu nutzen.

**Listing A.6** LoginEmployeeCommand.as, execute-Funktion

```
public function execute( cgEvent : CairngormEvent ) : void
{
        private var model : AppModelLocator =
                AppModelLocator.getInstance();

        var username : String = LoginEmployeeEvent( cgEvent ).username;
        var password : String = LoginEmployeeEvent( cgEvent ).password;

        if ( username == "Flex" && password == "Cairngorm" )
        {
                model.user = new User( username, password );
                model.viewing = AppModelLocator.EMPLOYEE_LIST;
        }
        ...
}
```

(6) War die Überprüfung der Nutzerdaten erfolgreich, werden im ModelLocator die Benutzerdaten (*user*) und eine Index-Variable (*viewing*) gesetzt. Der ModelLocator heißt in dieser Anwendung *AppModelLocator* (Listing A.7) und stellt die Daten für die gesamte Anwendung zur Verfügung.

**Listing A.7** AppModulLocator.as, Daten

```
        public var viewing : Number = EMPLOYEE_LOGIN;
        public var user : User;
```

Die *viewing-Variable* des *AppModelLocator*, die vom Command verändert wurde, steuert den Viewstack (Listing A.8), der sich, wie bereits beim Initialisierungsprozess erwähnt, in der Haupt-MXML befindet.

**Listing A.8** Main.mxml, Viewstack

```
<mx:ViewStack width="100%" paddingBottom="10" paddingTop="10"
        resizeToContent="true" selectedIndex="{model.viewing}">

                <view:EmployeeLogin />
                <view:EmployeeList />
                <view:EmployeeDetail />

</mx:ViewStack>
```

(7) Der „*selectedIndex*“ des Viewstacks ist mittels Flex-Databinding an die Modelvariable „*viewing*“ gebunden. Eine Änderung dieser Variable, wie in Listing A.6 beschrieben, bewirkt somit eine Umstellung der Nutzeroberfläche. Nach erfolgreichem Login befindet sich der Benutzer der „Cafe Townsend“ Anwendung demnach in der zweiten View, der *EmployeeList*.

Der geschilderte Ablauf einer Cairngorm-Anwendung ist grundlegend immer der selbe wie im eben erläuterten Login-Vorgang. Ein Command hat jedoch zusätzlich die Möglichkeit auf externe Daten zuzugreifen. Die Beschaffung externer Daten ist ein wichtiger Bestandteil des Cairngorm-AF und wird daher am Beispiel der Mitgliederdatenbeschaffung dargestellt. Der Ablauf im Cairngorm-System ist bis zum *LoadEmployeeCommand* analog zum Login-Beispiel und wird deshalb nicht noch einmal beschrieben. In Abbildung A.4 wird dieser Ablauf aber der Vollständigkeit halber grau unterlegt.

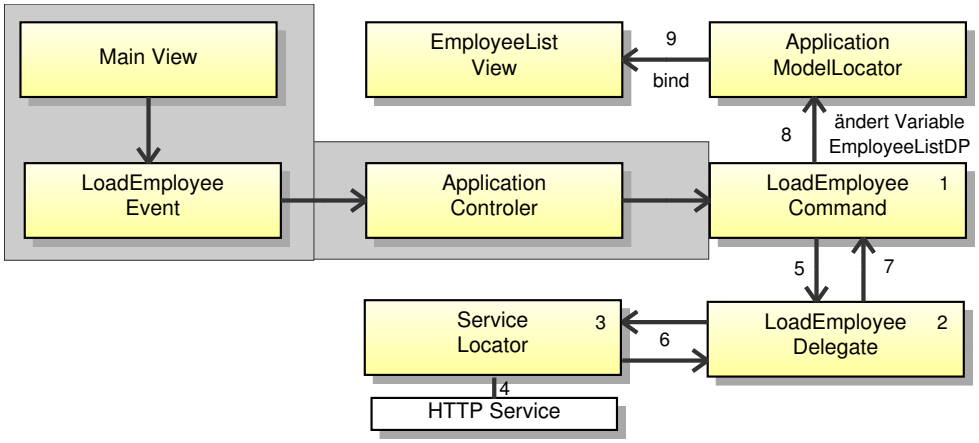


Abbildung A.4: Cairngorm, Beschaffung externer Daten

(1) Start der Betrachtung ist das *LoadEmployeeCommand*, welches vom *AppController* erstellt und ausgeführt wird.

(2) Zur Beschaffung externer Daten wird in der *execute-Funktion* des *LoadEmployeeCommands* (Listing A.9) ein *LoadEmployeeDelegate* erstellt. Dem Konstruktor des LoadEmployeeDelegate wird dabei eine Referenz für das Objekt (*Responder*) übergeben, welches die Antwort des Delegates bearbeiten soll. In diesem Beispiel bearbeitet das *LoadEmployeeCommand* die Anfrage selbst.

**Listing A.9** LoadEmployeeCommand.as, execute-Funktion

```
public function execute( cgEvent:CairngormEvent ) : void
{
        var delegate : LoadEmployeesDelegate =
                new LoadEmployeesDelegate( this );

        delegate.loadEmployeesService();
}
public function result( rpcEvent : Object ) : void
{
        model.employeeListDP = rpcEvent.result.employees.employee;
}
```

(3) Im Konstruktor des *LoadEmployeeDelegate* (Listing A.10) wird diesem Delegate ein Service für die Bearbeitung der Command-Anfrage zugewiesen. Dazu wird im ServiceLocator mit Hilfe einer ID („*loadEmployeesService*“) eine Datenkomponente gesucht.

**Listing A.10** LoadEmployeeDelegate.as

```
private var command : IResponder;
private var service : Object;

public function LoadEmployeesDelegate( command : IResponder )
{
        this.service = ServiceLocator.getInstance().getHTTPService(
                'loadEmployeesService' );        this.command = command;
}
public function loadEmployeesService() : void
{
        var token:AsyncToken = service.send();
        token.addResponder( command );
}
```

(4) Entspricht diese ID einer MXML-Datenkomponente wird diese an das Delegate gebunden. Im Fall des „Cafe Townsend“ Beispiels wurde diese ID einem HTTPService zugeordnet, welcher lediglich auf eine lokale Datei, die *Employee.xml*, zugreift (Listing A.11). Services müssen, wie bereits beim Initialisierungsvorgang angedeutet, im ServicesLocator definiert werden. Jede Anwendung definiert dabei spezifische externe Datenquellen, die dem Cairngorm-System durch den ServiceLocator bereitgestellt werden.

**Listing A.11** Services.mxml, HTTPService

```
<mx:HTTPService id="loadEmployeesService" url="assets/Employees.xml" />
```

(5) Anschließend wird das *LoadEmployeeDelegate* durch das Aufrufen der Funktion *loadEmployeesService* im Command mit der Beschaffung der Daten beauftragt (vgl. Listing A.9).

(6) Das Delegate ruft den eben beschrieben HTTPService des ServiceLocators auf, welcher die Daten aus der *Employees.xml*-Datei liest und diese an das Delegate zurückgibt.

(7) Die Antwort dieses Services wird vom Delegate an den Responder zurück delegiert. In diesem Bespiel ist der Responder, wie bereits bei Punkt (2) beschrieben, das aufrufende *LoadEmployeeCommand* selbst. Die Bearbeitung der Service-Antwort geschieht daher in der *result-Funktion* bzw. im Fehlerfall in einer gesonderten *fault-Funktion*, welche hier nicht erläutert wird.

(8) Die erhaltenen Daten werden vom *LoadEmployeeCommand* in den *AppModelLocator* geschrieben.

(9) Die einzige Möglichkeit Cairngorm-Daten Steuerkomponenten einer View zur Verfügung zu stellen ist das Flex-Databinding. Die Verbindung der Mitgliederdaten mit einer Liste im EmployeeList-View ist sehr ähnlich der im Punkt (7) des Login Beispiels beschriebenen Bindung und wird deshalb nicht noch einmal erläutert.

## A.2 PureMVC „Cafe Townsend“

In diesem Teil des Anhangs wird auf die PureMVC-Umsetzung der „Cafe Townsend“ Anwendung eingegangen. Ebenso, wie bei Cairngorm im Anhang A.1 wird auf eine vollständige Beschreibung verzichtet. Es werden lediglich Programmteile aufgezeigt, welche für das Verständnis des PureMVC-AF wichtig sind und die Funktionsweise verdeutlichen. Die Beschreibungen beziehen sich dabei auf den originalen PureMVC „Cafe Townsend“ Code[2], welcher Routineprozesse, wie beispielsweise das Überprüfen von Login Daten, vereinfacht. Beschreibungen von Hintergrundvorgängen im PureMVC-System werden vernachlässigt, da sie für die Benutzung des PureMVC-AF nicht relevant sind.

Beim PureMVC-Initialisierungsvorgang müssen die benötigten Mediatoren und Proxies in das PureMVC-System eingebunden und die Notification-Command-Verbindungen definiert werden. Das ist für jedes PureMVC-Projekt notwendig, weshalb die grundlegende Vorgehensweise nachfolgend kurz erläutert wird.

Als Erstes wird eine konkrete Facade (*ApplicationFacade*) in der Haupt-MXML (*CafeTownsend.mxml*) erstellt. In dieser Facade sind die Notification-Command-Verknüpfungen definiert (Listing A.12), welche beim Erstellen dieser Facade automatisiert an den Controller übergeben werden.

**Listing A.12** ApplicationFacade.as, initializeController-Funktion

```
registerCommand( STARTUP, ApplicationStartupCommand );
```

Nachdem die Haupt-MXML vollständig erstellt wurde, wird eine *startup-Funktion* in der zuvor angelegten Facade aufgerufen. Diese Funktion sendet eine Notification an das PureMVC-System (Listing A.13).

[2] http://trac.puremvc.org/Demo_AS3_Flex_CafeTownsend

**Listing A.13** ApplicationFacade.as, startup-Funktion

```
sendNotification( STARTUP, app );
```

Da dem Controler durch die Facade bereits alle Notification-Command-Verknüpfungen übergeben wurden, wird durch das Senden dieser Notification ein MacroCommand (*ApplicationStartupCommand*) ausgelöst. Dieses Command führt seinerseits ein Command zum Registrieren der Mediatoren (*ViewPrepCommand*) und ein Command zum Registrieren der Proxies (*ModelPrepCommand*) aus. Im „Cafe Townsend“ Beispiel erstellt das *ModelPrepCommand* einen Proxy für den Programmbenutzer und einen für die Cafe Mitarbeiter. Das *ViewPrepCommand* registriert Mediatoren für die Haupt-MXML sowie für die in Anhang A eingeführten Ansichten (EmployeeLogin, EmployeeList, EmployeeDetail). Nachdem diese Commands ihre Aufgaben ausgeführt haben, ist das PureMVC-System bereit und erwartet Nutzereingaben. Nachfolgend wird der Prozess des Nutzer Logins aufgezeigt und anschließend auf die Beschaffung der Mitgliederdaten eingegangen.

Wie schon im Cairngorm „Cafe Townsend“ Beispiel (vgl. Anhang A.1) sind die drei Ansichten (Views) in der Haupt-MXML enthalten und werden mittels einer ViewStack-Steuerkomponente umgeschaltet. Nach der Initialisierung des PureMVC-Systems befindet sich der Benutzer in der EmployeeLogin-View und kann dort den Login-Vorgang beginnen. Ausgehend von dieser Startsituation zeigt Abbildung A.5 die Systemreaktion einer PureMVC-Anwendung auf das Betätigen des Login Buttons.

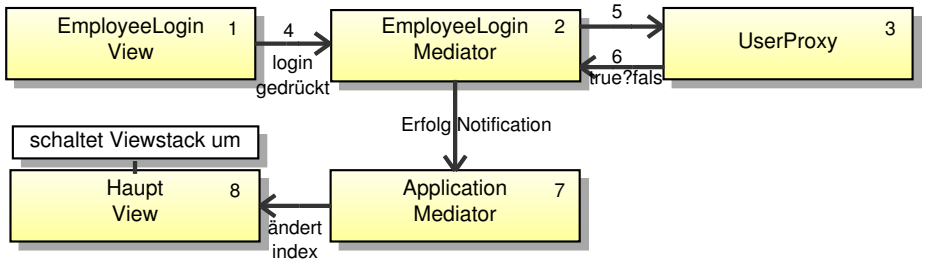


Abbildung A.5: PureMVC, Login

(1) In der EmployeeLogin View befinden sich drei Steuerelemente: zwei Textinputfelder für Login und Passwort sowie ein Login- Button zum Absenden der Login Anfrage.

(2) Im Verlauf des Initialisierungsvorgangs im *ViewPrepCommand* wird ein Mediator, der *EmployeeLoginMediator*, an die EmployeeLogin-View gebunden.

(3) Anschließend wird im Konstruktor des *EmployeeLoginMediators* eine Referenz des UserProxy von der Facade erfragt und in der *userProxy-Variable* des *EmployeeLoginMediators* gespeichert (Listing A.14).

**Listing A.14** EmployeeLoginMediator.as, Speichern des UserProxy

```
userProxy = UserProxy( facade.retrieveProxy( UserProxy.NAME ) );
```

(4)Nach der Eingabe des Nutzernamens und Passwortes wird nach Drücken des Login-Buttons ein Flash-Event aus der EmployeeLogin-View gesendet (Listing A.15). Um dieses Event im entsprechenden Mediator fangen zu können, wird im Konstruktor des *EmployeeLoginMediators* ein EventListener angelegt (Listing A.16).

**Listing A.15** EmployeeLogin.mxml, Senden eines Events

```
dispatchEvent( new Event( APP_LOGIN ) );
```

**Listing A.16** EmployeeLoginMediator.as, Fangen eines Events

```
employeeLogin.addEventListener( EmployeeLogin.APP_LOGIN, login );
```

(5) Der EventListener führt nach dem Erhalt des Events die *login-Funktion* des *EmployeeLoginMediators* aus, welche den UserProxy nach der Korrektheit der eingegebenen Nutzerdaten befragt (Listing A.18). Dazu stellt der UserProxy dem PureMVC-System seine *validate-Funktion* zu Verfügung. Da es in diesem Beispiel ohnehin nur eine gültige Nutzernamen-Passwort-Kombination gibt, verzichtet der PureMVC-„Cafe Townsend“-Entwickler an dieser Stelle auf die sonst übliche Involvierung eines externen Servers und hat sich dazu entschieden, die validate-Funktion des UserProxies entsprechend zu vereinfachen (Listing A.17 ).

**Listing A.17** UserProxy.as, validate-Function

```
public function validate( username:String, password:String ):Boolean
{
        if( username == "Flex" && password == "PureMVC" )
        {
                data = new User(username,password);
                return true;
        }
        else
        {
                return false;
        }
}
```

(6) Sind die Nutzerdaten korrekt, wird der angemeldete Nutzer im UserProxy gespeichert und die Antwort „*true*“ zurückgegeben. In diesem Fall sendet der *EmployeeLoginMediator* eine Notification an das PureMVC-System. Erhält der Mediator jedoch die Antwort „*false*“ zurück, wird mittels AlertBox eine Nutzernachricht auf dem Bildschirm ausgegeben. Die vollständige *login-Funkion* des *EmployeeLoginMediators* ist in Listing A.18 dargestellt und verdeutlicht die Prozesse der Punkt 4 und 5.

**Listing A.18** EmployeeLoginMediator.as, login-Funktion

```
private function login( event:Event = null ) : void
{
        var isValid:Boolean = userProxy.validate(
                employeeLogin.username.text, employeeLogin.password.text);
        if( isValid )
        {
                sendNotification( ApplicationFacade.VIEW_EMPLOYEE_LIST );
        }
        else
        {
                Alert.show( "Invaild username and/or password. Please try
                                again.","Login Failed" );
        }
}
```

(7) Die Notification, welche der EmployeeLoginMediator nach der Eingabe korrekter Nutzerdaten sendet, wird wiederum vom *ApplicationMediator* gefangen. Der *ApplicationMediator* ist der Haupt-MXML zugeordnet und hat somit Zugriff auf dessen MXML-Komponenten sowie auf dessen öffentliche Daten.

(8) In der Haupt-MXML befindet sich der ViewStack, welcher die drei „Cafe Townsend“-Views beinhaltet. Der SelectedIndex des Viewstacks wird nach Erhalt der Notification in der handleNotification-Funktion des *ApplicationMediators* umgesetzt, was eine Änderung der Benutzeroberfläche nach sich zieht. Nun ist die EmployeeList View mit allen Cafe Mitarbeitern auf dem Bildschirm zu sehen.

Im „Cafe Townsend“-PureMVC-Beispiel sind die EXTERNEN DATEN der Mitglieder zu diesem Zeitpunkt bereits geladen. Um die Verständlichkeit des Login- und Initialisierungsvorgangs nicht zu gefährden, wurde bis jetzt bewusst auf die Beschreibung dieses Prozesses verzichtet. Nachfolgend wird nun erklärt, wie die Daten der Cafe Mitarbeiter in den EmployeeProxy gelangen. Abbildung A.6 verdeutlicht zusätzlich dieses Vorgehen.

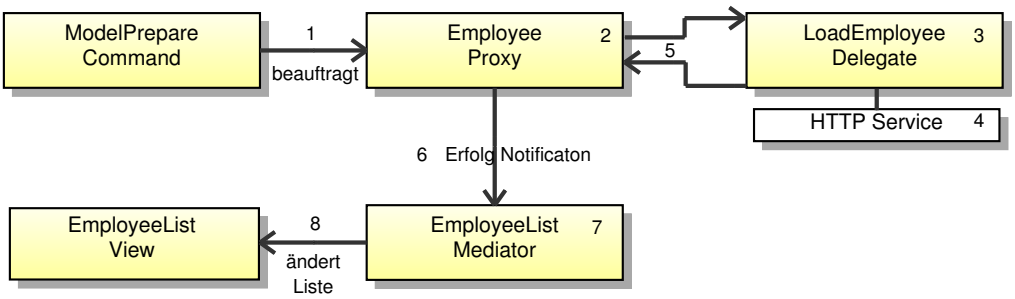


Abbildung A.6: PureMVC, Beschaffung externer Daten

(1) Das *ViewPrepCommand*, welches in der Initialisierungsbeschreibung am Anfang des Kapitels eingeführt wurde, weist den *EmployeeProxy* an, die Daten der Mitarbeiter zu beschaffen.

(2) Dazu erfragt das *ViewPrepCommand* von der Facade eine Referenz des *EmployeeProxy*, um anschließend die *loadEmployees-Funktion* des *EmployeeProxy* auszuführen (Listing A.19).

**Listing A.19** ViewPrepCommand.as, execute-Funktion

```
var employeeProxy:EmployeeProxy =
        facade.retrieveProxy( EmployeeProxy.NAME ) as EmployeeProxy;
employeeProxy.loadEmployees();
```

(3) Code A.20 zeigt die Erstellung eines Delegates in der loadEmployee-Funktion des EmployProxy. Dieses Delegate leitet, ebenso wie in Cairngorm (vgl. Anhang A.1), die

Anfragen an einen Server weiter und liefert das Ergebnis zurück an einen Responder. Der Responder wird dem Konstruktor des *LoadEmployeeDelegate* übergeben und ist in diesem Beispiel der *EmployeeProxy* selbst.

**Listing A.20** EmployeeProxy.as, loadEmployee-Funktion

```
public function loadEmployees():void
{
        var delegate : LoadEmployeesDelegate =
                new LoadEmployeesDelegate( this );
        delegate.loadEmployeesService();
}
```

(4) Das *LoadEmployeeDelegate* erstellt in seinem Konstruktor einen HTTP-Service, welchem eine lokale Datei (*Employee.xml*) als Quelle übergeben wird (Listing A.21). Gleichzeitig wird der übergebene Responder gesichert.

**Listing A.21** LoadEmployeeDelegate.as, Konstruktor

```
public function LoadEmployeesDelegate( responder : IResponder )
{
        this.service = new HTTPService();
        this.service.url="assets/Employees.xml";
        this.responder = responder;
}
```

(5) Nach Aufruf der *loadEmployee-Funktion* des Delagates durch den *EmployeeProxy* (Listing A.20) wird der Service ausgeführt und das Ergebnis über das Delegate zum Responder, dem *EmployeeProxy*, zurückgegeben. Diese Daten werden im *EmployeeProxy* gespeichert.

(6) Nach dem Erhalt der Daten sendet der *EmployeeProxy* eine Notification, um das PureMVC-System über seinen neuen Status zu informieren (Listing A.22).

**Listing A.22** EmployeeProxy.as, Senden einer Notification

```
sendNotification( ApplicationFacade.LOAD_EMPLOYEES_SUCCESS );
```

(7) Diese Notification wird vom *EmployeeListMediator*, welcher die EmployeeList View steuert, gefangen.

(8) Der *EmployeeListMediator* referenziert den *EmployeeProxy* und kann so die Daten aus diesem in die View übertragen (Listing A.23). Die Mitarbeiterliste ist nun gefüllt.

**Listing A.23** EmployeeListMediator.as, Füllen der Mitarbeiterliste

```
employeeList.employees_li.dataProvider = employeeProxy.employeeListDP;
```

## A.3 MATE „Cafe Townsend“

Nachfolgende Beschreibung bezieht sich auf die MATE „Cafe Townsend“ Anwendung[3]. Ebenso wie bei Cairngorm (vgl. Anhang A.1) und PureMVC (vgl. Anhang A.2) werden lediglich Programmteile beschrieben, welche für eine Einführung in die Funktionsweise des AF benötigt werden. Interne Abläufe des MATE-AF werden dabei vernachlässigt.

Bevor ein MATE-System einsetzbar ist, müssen verschiedene Initialisierungsprozesse ausgeführt werden. Diese Prozesse sind für alle MATE-Anwendungen sehr ähnlich, weshalb sie nachfolgend kurz erläutert werden. Der Haupt-MXML (*CafeTownsend.mxml*) wird in MATE-Anwendungen mindestens eine EventMap (*MainEventMap.mxml*) und ein Layout-Container (*MainUI.mxml*) hinzugefügt (Listing A.24).

**Listing A.24** CafeTownsend.mxml, Anlegen von EventMap und Layout-Container

```
<maps:MainEventMap />
<views:MainUI />
```

Der Layout-Container beinhaltet alle View-Elemente einer Anwendung. Im „Cafe Townsend“-Beispiel befindet sich in diesem Container ein Viewstack, welcher die Views umschaltet (*Login.mxml, EmployeeList.mxml, EmployeeDetail.mxml*). Wie schon in Abschnitt 3.3.1 beschrieben, nutzt dieses Beispielprogramm Presentation-Models. Eine „Cafe Townsend“-View referenziert dazu sein Presentation-Model mittels einer *model-Variable*, legt das Presentation-Model jedoch nicht selbst an. Mittels der EventMap wird das jeweilige Presentation-Model in eine View übergeben (injiziert). Listing A.25 zeigt beispielhaft das Injizieren des EmployeeListPresentationModel für die EmployeeList-Views mittels Injectors-Tag in der EventMap.

[3] http://mate.asfusion.com/page/examples/cafe-townsend

**Listing A.25** MainEventMap.mxml, Injizieren eines Presentation-Models

```
<Injectors target="{ EmployeeList }">
        <PropertyInjector targetKey="model"
                source="{ EmployeeListPresentationModel }" />
</Injectors>
```

Die EmployeeList-View (*EmployeeList.mxml*) nutzt das injizierte Presentation-Model zum Beispiel als Datenquelle für die Cafe-Mitarbeiter-Liste (Listing A.26).

**Listing A.26** EmployeeList.mxml, Nutzen eines injizierten Presentation-Models

```
<mx:Script>
        ...
        [Bindable] public var model:EmployeeListPresentationModel;
        ...
</mx:Script>
<mx:List id="employees_li" dataProvider="{ model.employees }"/>
...
```

Die Presentation-Models werden mittels der EventMap ihrerseits an verschiedene Manager der Anwendung „angeschlossen“. Dabei wird allerdings nicht das gesamte Manager-Objekt an das Presentation-Model übergeben, sondern lediglich verschiedene Manager-Daten. Listing A.27 zeigt das Injizieren der *employeeList-Variable* des EmployeeManagers in die *employee-Variable* des EmployeeListPresentationModel in der EventMap.

**Listing A.27** MainEventMap.mxml, Injizieren von Manager-Daten in ein Presentation-Model

```
<Injectors target="{ EmployeeListPresentationModel }">
        <PropertyInjector targetKey="employees"  source="{EmployeeManager}"
                sourceKey="employeeList" />
</Injectors>
```

Ändert sich nun die *employeeList-Variabel* des EmployeeManagers so ändert sich auch die *employee-Variable* des EmployeeListPresentationModels was eine Änderung der Mitarbeiter Liste in der EmployeeList View zur Folge hat[4]. Managerdaten können in

[4]Das ist möglich durch Einsetzen von Flex-Databinding und getter/setter-Funktion in den Manager bzw. PresentatorModel-Klassen

unterschiedliche Presentation-Models injiziert werden und bilden somit das Model einer MATE-Anwendung. Neben dem Injizieren von Abhängigkeiten definiert die EventMap Event-Listener und Services, welche an gegebener Stelle vorgestellt werden. Nachdem die EventMap und der Layout Container in das Projekt eingefügt wurden ist die Anwendung einsatzbereit.

Ausgehend von der Login-View wird die Systemreaktion auf das Betätigen des Login-Buttons beschrieben. Abbildung A.7 zeigt den LOGIN-VORGANG im „Cafe Townsend“ Beispiel mit Hilfe des MATE-AF.

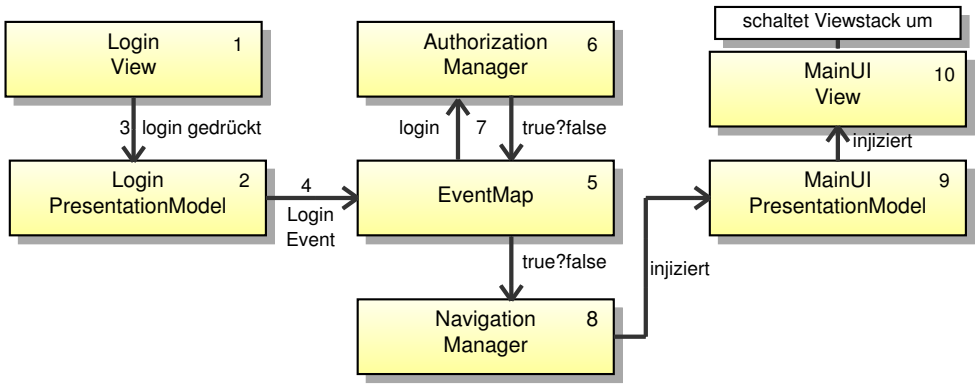


Abbildung A.7: MATE, Login

(1) Wie bereits in Cairngorm und PureMVC kann man die Login-View auf drei wesentliche Elemente reduzieren: zwei Textfelder für Nutzername und Passwort sowie einen Login-Button.

(2) Die Login-View referenziert ihr Presentation-Model, das LoginPresentationModel, mittels einer *model-Variable*. Das Anbinden des LoginPresentationModels an die Login-View ist analog dem Vorgehen bei der EmployeeList-View (Listing A.25).

(3) Da die Login-View sein Presentation-Model „kennt“, wird nach Betätigen des Login-Buttons (*loginBtn*) eine *login-Funktion* im LoginPresentationModels aufgerufen. Das Listing A.28 zeigt diesen Aufruf in der Login-View sowie die Übergabe des Benutzernamens und Passwortes.

**Listing A.28** Login.mxml, login-Funktion und MXML-Komponenten

```
<mx:Script>
...
[Bindable] public var model:LoginPresentationModel;

private function login():void
{
        model.login( username.text, password.text );
}
</mx:Script>
...
<mx:TextInput id="username..."/>
<mx:TextInput id="password" displayAsPassword="true"... />
<mx:Button id="loginBtn" label="Login" click="login()"/>
...
```

(4) Die *login-Funktion* des LoginPresentaionModels ist in Listing A.29 dargestellt und zeigt die Generierung und das Senden eines LoginEvents (*event*). Das LoginEvent besitzt zwei Variablen für Nutzername (*username*) und Passwort (*password*) in dem die übergebenen Nutzerdaten abgespeichert werden.

**Listing A.29** LoginPresentationModel.as, login-Funktion

```
public function login( userName:String, password:String ):void
{
        var event:LoginEvent = new LoginEvent( LoginEvent.LOGIN );
        event.username = userName;
        event.password = password;
        dispatcher.dispatchEvent( event );
}
```

(5) Dieses Event wird von der EventMap gefangen und bearbeitet (Listing A.30). Die Bearbeitung dieser Events wird innerhalb der EventMap mittels EventHandler-Tags definiert. Ein Event-Name wird dabei durch eine „inline“ Type-Variable, in diesem Fall die *LoginEvent.LOGIN-String-Konstante*, einem EventHandler zugeordnet. Nachdem das Event gefangen wurde, werden die Tags im EventHandler von oben nach unten abgearbeitet. Normalerweise würde an dieser Stelle eine externe Datenquelle befragt werden. Da aber nur eine Nutzernamen-Passwort-Kombination zugelassen ist, verzichtet das „Cafe Townsend“-Beispiel darauf.

**Listing A.30** MainEventMap.mxml, Events fangen und Verarbeiten

```
<EventHandlers type="{LoginEvent.LOGIN}">
        <MethodInvoker generator="{ AuthorizationManager }" method="login"
                arguments="{[event.username, event.password]}" />

        <MethodInvoker generator="{ NavigationManager }"
                method="updateAfterLogin" arguments="{ lastReturn }"/>
</EventHandlers>
```

(6) Der erste *MethodInvoker-Tag* erstellt, falls noch nicht vorhanden, einen *AuthorizationManager* und ruft die *login-Funktion* dieses Managers auf. Außerdem werden der *login-Funktion* der im Event enthaltene Nutzername und das Nutzerpasswort übergeben.

(7) Die *login-Funktion* des *AuthorizationManagers* ist denkbar einfach. Listing A.31 zeigt die Überprüfung der zugelassenen Nutzernamen-Passwort-Kombinationen sowie die Rückgabe des Ergebnisses an die EventMap.

**Listing A.31** AuthorizationManager.as, login-Funktion

```
public function login(username:String, password:String):Boolean
{...
        var isLogin:Boolean =
                (username == 'Flex' && password == 'Mate');

        return isLogin;
}
```

(8) Der zweite *MethodInvoker-Tag* in der EventMap erstellt, falls noch nicht vorhanden, einen *NavigationManager* und ruft dessen *updateAfterLogin-Funktion* auf. Dieser Funktion wird der Rückgabewert aus Punkt (7) mittels der *lastReturn-Variable* der EventMap übergeben. Abhängig vom übergebenen Wert führt die *upDateAfterLogin-Funktion* des *NavigationManagers* zu einer Veränderung der *navigationPath-Variable* dieses Managers.

(9) Die *navigationPath-Variable* des *NavigationManagers* wurde in das *MainUIPresentationModel* injiziert. Reduziert man den Code auf das Wesentliche, bewirkt eine Veränderung der *navigationPath-Variable* des *NavigationManagers* eine entsprechende Veränderung des *stackIndex* in dem *MainUIPresentationModel*.

(10) Das *MainUIPresentationModel* ist das Presentation-Model für den Layout-Container also für die *MainUI.mxml* Datei, welche den Viewstack mit den drei „Cafe Townsend“-Views beinhaltet. Bestätigt der in Punkt (6) beschriebene *AuthorizationManager* die

Login Daten, wird im *NavigationManager* ein neuer *navigationPath* gesetzt. Dies führt zu einer Änderung der *stackIndex-Varibale* des *MainUIPresentationModels*. Da diese *stackIndex-Variable* ihrerseits an den *selectedIndex* des Viewstacks gebunden wurde, ändert sich die Nutzeroberfläche (Listing A.32). Die EmployeeList-View mit allen Café Mitarbeitern ist nun zu sehen.

**Listing A.32** MainUI.mxml, Viewstack

```
<mx:Script>
        [Bindable]      public var model:MainUIPresentationModel;
</mx:Script>
<mx:ViewStack id="viewStack"
        selectedIndex="{ model.stackIndex }".../>
        <views:Login />
        <views:EmployeeList />
        <views:EmployeeDetail />
</mx:ViewStack>
...
```

Die Beschaffung der EXTERNEN DATEN der Cafe Mitarbeiter wurde vom System bereits erledigt. Abbildung A.8 zeigt den Prozess der Datenbeschaffung sowie die Verbindung der Daten.

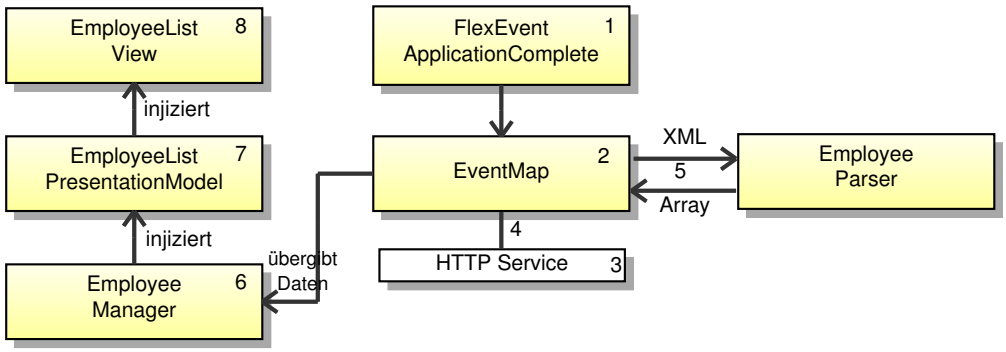


Abbildung A.8: MATE, Beschaffung externer Daten

(1) Nachdem die Flex Anwendung fertig erstellt ist, sendet diese ein ApplicationComplete-FlexEvent ab.

(2) In der EventMap des „Cafe Townsend“ Beispiels wird dieses Event gefangen und verarbeitet. Listing A.33 verdeutlicht das Vorgehen.

**Listing A.33** MainEventMap.mxml, externe Datenquellen

```
<EventHandlers type="{FlexEvent.APPLICATION_COMPLETE}">
        <HTTPServiceInvoker instance="{employeesService}">
                <resultHandlers>
                        <MethodInvoker generator="{EmployeeParser}"
                                method="loadEmployeesFromXML"
                                arguments="{resultObject}" />

                        <MethodInvoker generator="{EmployeeManager}"
                                method="saveEmpoyeeList"
                                arguments="{lastReturn}" />
                </resultHandlers>
        </HTTPServiceInvoker>
</EventHandlers>
```

Als Erstes definiert die EventMap einen *HTTPServiceInvoker-Tag*, welcher die Daten von einem *HTTPService* beschaffen soll. Der zu nutzende Service (*employeeService*) wird „inline“ per *instance-Variable* übergeben.

(3) Der übergebene *employeeService* nutzt die lokale *Employee.xml*-Datei und ist in Listing A.34 dargestellt. In diesem vereinfachten Bespiel wurde der *HTTPService* direkt in der EventMap definiert.

**Listing A.34** MainEventMap.mxml, HTTPService

```
<mx:HTTPService id="employeesService" url="assets/data/Employees.xml.../>
```

(4) Der *HTTPServiceInvoker* startet den *HTTPService* automatisch und liefert das Ergebnis in seinen *resultHandler* zurück.

(5) Nach erfolgreicher Datenbeschaffung involviert die EventMap mittels *MethodInvoker-Tag* den *EmployeeParser*. Die *loadEmployeesFromXML-Funktion* des *EmployeeParsers* wird das Ergebnis des *HTTPServicesInvokers* (*resultObject*) übergeben. Diese Funktion wandelt die erhaltenen XML-Daten in ein Array um und gibt diesen an die EventMap zurück.

(6) Zum Speichern der Daten im *EmployeeManager* wird mittels *MethodeInvoker*-Tag die *saveEmployeeList-Funktion* des *EmployeeManagers* aufgerufen und die Rückgabedaten (*lastResult*) des *EmployeeParsers* übergeben. Die Mitarbeiter-Daten sind nun in der *employeeList-Variable* des *EmployeeManagers* gespeichert.

(7) Während des Initialisierungsprozesses wurde diese Variable bereits in das *EmployeeListPresentationModel* injiziert (Listing A.25).

(8) Die EmployeeList-View nutzt die Daten seines Presentation-Models wiederum als Datenquelle für die Mitarbeiter-Liste (Listing A.26). Die Mitarbeiter des Cafes werden nun angezeigt.

# Literaturverzeichnis


[Ado08] Adobe. Adobe flex builder 3 create engaging, cross platforom rich internet applications. http://www.adobe.com/products/flex/pdfs/fb3_datasheet.pdf, 2008. zuletzt abgerufen am 27.03.2009.

[Arg09] Laura Arguello. Mate flex framework. http://www.flexcampmiami.com/page.cfm/agenda/laura-arguello-mate-flex-framework, 2009. zuletzt abgerufen am 28.03.2009.

[BCK04] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. SEI Series in Software Engineering, 2004.

[Ber06] Joe Berkovitz. An architectural blueprint for flex applications. http://www.adobe.com/devnet/flex/articles/blueprint.html, 2006. zuletzt abgerufen am 27.03.2009.

[BMR+98] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-orientierte Software-Architekturen: ein Pattern System*. Addison-Wesley, 1998.

[BSB08] Thomas Burleson, Leo Shuman, and Matt Boles. Flex 3: Introducing cairngorm. http://www.adobe.com/devnet/flex/articles/introducing_cairngorm/introducing_cairngorm.pdf, 2008. zuletzt abgerufen am 28.03.2009.

[Bun09] Bundesregierung. Kabinettklausur am 17. und 18. november 2009 im gästehaus der bundesregierung, schloss meseberg. http://www.antje-tillmann.de/cms/upload/files/pol_themen/2009_11_24_Beschluesse%20Meseberg.pdf, 2009. zuletzt abgerufen am 27.03.2009.

[Bus09] Sven Busse. *Flash Engineering: Agile Ansätze zum Bau von RIAs mit Flash, Flex und Actionscript*. Addison-Wesley, 2009.

[Coe09] Christophe Coenraets. Max frameworks session: One application, four implementations. `http://coenraets.org/`, 2009. zuletzt abgerufen am 28.03.2009.

[EF09] Birgit Eimeren and Beate Frees. Der internetnutzer 2009 multimedial und total vernetzt? `http://www.ard-zdf-onlinestudie.de/fileadmin/Online09/Eimeren1_7_09.pdf`, 2009. zuletzt abgerufen am 27.03.2009.

[Eli09] Nadja Elias. Internet facts 2009-iii online durchdringung nähert sich der 75-prozent marke. `http://www.agof.de/agof-pm-internet-facts-2009-iii.download.0fc449e0f640f1e77d63b315f8ef1132.pdf`, 2009. zuletzt abgerufen am 27.03.2009.

[For05] Maureen Forys. Getting to know flash. `http://media.wiley.com/product_data/excerpt/80/07821410/0782141080.pdf`, 2005. zuletzt abgerufen am 27.03.2009.

[FTR10] Yakov Fain, Anatole Tartakovsky, and Victor Rasputnis. *Enterprise Development with Flex*. O Reilly, 2010.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Pattern: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[Grä04] Tobias Gräning. *Macromedia Falsh MX 2004*. Galileo Design, 2004.

[Gra09] Jethro Grassie. Flex architectural frameworks. `http://blog.cjtech.co.uk/index.php/2009/02/24/flex-architectural-frameworks/`, 2009. zuletzt abgerufen am 28.03.2009.

[Hal08a] Clifford Hall. Puremvc: Framework goals&benefits. `http://puremvc.org/pages/docs/current/PureMVC_Framework_Goals_and_Benefits.pdf`, 2008. zuletzt abgerufen am 28.03.2009.

[Hal08b] Clifford Hall. Puremvc: Implementierung idiome und optimale anwendung. `http://puremvc.org/pages/docs/current/PureMVC_Implementierung_Idiome_und_Optimale_Anwendung.pdf`, 2008. zuletzt abgerufen am 28.03.2009.

[Hei08] Patrick Heinzelmann. Prana framework wird zum spring actionscript framework. `http://www.patrick-heinzelmann.de/blog/2008/12/18/prana-framework-wird-zum-spring-actionscript-framework/`, 2008. zuletzt abgerufen am 28.03.2009.

[KL08] Chafic Kazoun and Joey Lott. *Programming Flex 3*. O Reiley, 2008.

[KS09] Guido Krüger and Thomas Starke. *Handbuch der Java Programmierung*. Addison-Wesley, 2009.

[LL07] Jochen Ludewig and Horst Lichter. *Software-Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2007.

[Mar08] Sebastian Martens. 360flex live another flex framework: Swiz. `http://www.flexughh.de/2008/08/19/360flex-live-another-flex-framework-swiz/`, 2008. zuletzt abgerufen am 28.03.2009.

[MH97] Bertrand Meyer and Prentice Hall. *Object Oriented Software Construction; ; PTR; 1997*. Prentice Hall PTR, 1997.

[Mor06] Laurence Moroney. *Foundations of WPF: an introduction to Windows Presentation Foundation*. Apress, 2006.

[NH05] Tom Noda and Shawn Helwig. Rich internet applications. `http://www.google.de/url?sa=t&source=web&ct=res&cd=4&ved=0CBYQFjAD&url=http%3A%2F%2Fwww.uwebc.org%2Fopinionpapers%2Fdocs%2FRIA.pdf&rct=j&q=rich+internet+applications+tom+noda&ei=6B3gSonZAZme_Aackpi6AQ&usg=AFQjCNFX5EzifMXOmRSo9hDuIhAp8jQOOw`, 2005. zuletzt abgerufen am 27.03.2009.

[PBG07] Torsten Posch, Klaus Birken, and Michael Gerdom. *Basiswissen Softwarearchitekturen: verstehen, entwerfen, wiederverwenden*. dpunkt.verlag, 2007.

[Per95] Garlan Perry. Software architecture definitions. `http://www.sei.cmu.edu/architecture/start/classicdefs.cfm`, 1995. zuletzt abgerufen am 28.03.2009.

[Rüt09] Michael Rüttger. *Adobe Flex3*. mitp, 2009.

[SC07] William Sanders and William Cumaranatunge. *Actionscript 3 Design Patterns*. O Reilly, 2007.

[SG09] Jeff Scanlon and Ashish Ghoda. *Accelerated Silverlight 3:Gear up fast to develop rich internet applications using silverlight 3*. Apress, 2009.

[Sim05] Susan Sim. A small social history of software architecture. `http://www.ics.uci.edu/~ses/papers/smallsocial.pdf`, 2005. zuletzt abgerufen am 28.03.2009.

[Sta08] Gernot Starke. *Effektive Software-Architekturen: ein praktischer Leitfaden*. Hanser, 2008.

[Ste09] Ralp Steyer. *Einstig in JavaFX: Dynamische und interaktive Java-applikationen mit JavaFX*. Addison-Wesley, 2009.

[THEM08] Shashank Tiwari, Jack Herrington, Elad Elrom, and Joshua Mostafa. *AdvancED Flex 3*. Friensof, 2008.

[Tuc08] David Tucker. Cairngorm videos. `http://www.davidtucker.net/2008/04/01/cairngorm-videos-available-as-flv-downloads/`, 2008. zuletzt abgerufen am 29.03.2009.

[Tuc09] David Tucker. The current state of flex frameworks. `http://www.davidtucker.net/2009/10/13/the-current-state-of-flex-frameworks/`, 2009. zuletzt abgerufen am 28.03.2009.

[VAC+05] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, U. Mehlig, Th. Neumann, M. Völter, and U. Zdun. *Software-Architekturen: Grundlagen-Konzepter-Praxis*. Spektrum Akademischer Verlag, 2005.

[Wal09] Petra Waldminghaus. *Adobe Flex 3:Rich Interner Applications erstellen.* Galileo Computing, 2009.

[Wid08] Simon Widjaja. *Rich Internet Application mit Aobe Flex 3.* Hanser, 2008.

[Wil07] Paul Williams. Presentation patterns - presentation model. `http://blogs.adobe.com/paulw/archives/2007/10/presentation_pa_3.html`, 2007.

[Wis09] Jeremy Wischusen. *Professional Cairngorm.* wrox, 2009.

[WT08] Steven Webster and Leon Tanner. Developing flex rias with cairngorm microarchitecture part 1: Introducing cairngorm. `http://www.adobe.com/devnet/flex/articles/cairngorm_pt1.html`, 2008. zuletzt abgerufen am 28.03.2009.

# Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit zum Thema

*Entwicklung einer „Rich Internet Application" mit Flex unter Verwendung von Architektur-Frameworks*

selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt habe.

Dresden, den 07. April 2010

---

Mirko Kunath