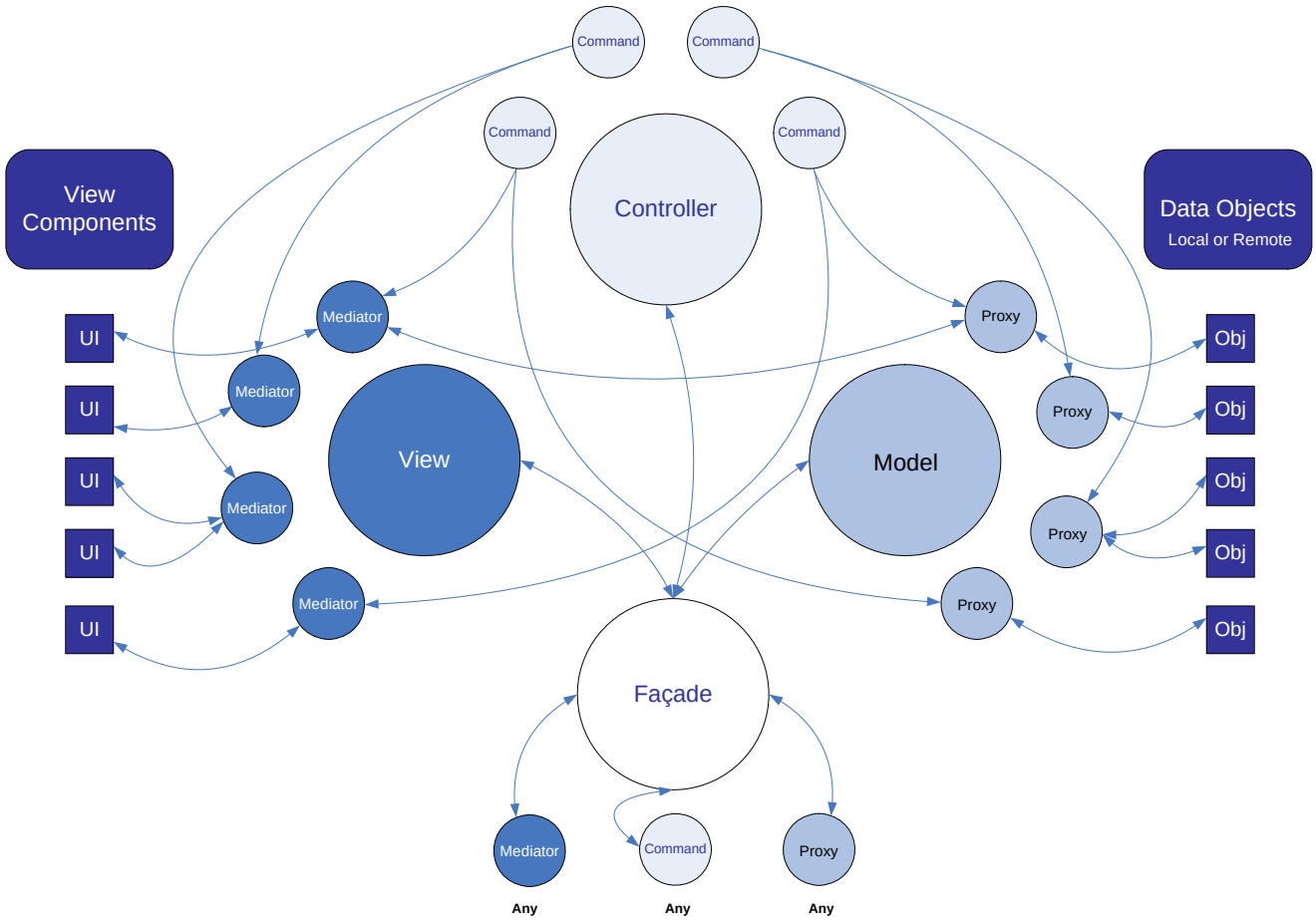
Implementierung Idiome und Optimale Anwendung

Entwicklung von robusten, skalier- und pflegbaren
Client-Anwendungen mit PureMVC.
Mit Beispielen in ActionScript 3 und MXML.



AUTHOR: Cliff Hall <cliff@puremvc.org>
ÜBERSETZUNG: Jens Krause, www.websector.de
STAND: 11.05.2008

Inspiration

PureMVC ist ein auf Entwurfsmuster basierendes Framework, welches einst aus der Notwendigkeit heraus entstand, leistungsstarke Rich Internet Anwendungen zu entwickeln. PureMVC ist mittlerweile in weiteren Sprachen und Plattformen (inklusive Serverumgebungen) portiert worden. Dieses Dokument ist schwerpunktmäßig für die Client-Seite geschrieben.

Während die Interpretation und Implementation für jede von PureMVC unterstützte Plattform spezifisch ist, basieren alle benutzten Entwurfsmuster auf den Definitionen des genialen 'Gang of Four' Buches: **Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software** (ISBN 3-8273-2199-9) Äußerst empfehlenswert!

PureMVC Aufbau

Das PureMVC Framework hat ein sehr klares Ziel: Eine Hilfe zu bieten, um die Coding-Interessen einer Applikation in drei eigenständige Bereiche zu unterteilen: Model, View und Controller.

Diese Unterteilung von Interessen sowie die geringe Verwendung von engen Verbindungen und Ausrichtungen zwischen den drei Bereichen untereinander sind höchst bedeutsam für die Entwicklung von skalierbaren und pflegbaren Anwendungen.

Bei der Implementation des klassischen MVC-Architekturmusters werden diese drei Bereiche über Singletons (eine Klasse, bei der nur eine Instanz erstellt wird) verwaltet. Diese Singletons sind auch so benannt: Model, View, Controller. Zusammen werden sie als 'Hauptakteure' bezeichnet.

Ein vierter Singleton, die sogenannte Façade, bietet ein Interface, um die Kommunikation zwischen den Hauptakteuren einfach zu halten.

Model & Proxies

Das Model hält benannte Referenzen zu den Proxies. Der Code vom Proxy beinflusst das Daten-Model und kommuniziert mit Remote-Services, falls die Daten persistent gehalten oder neu abgefragt werden sollen.

Das Ergebniss ist ein portierbarer Code im Bereich des Models.

View & Mediators

Der View hält in erster Linie benannte Referenzen zu den Mediators. Der Code der Mediatoren verwaltet View Components und fügt dafür Event Listener hinzu. Ein Mediator versendet oder empfängt

PureMVC Aufbau

View & Mediators

Notifications zum bzw. vom Rest des Systems, um den State der View Component zu beeinflussen.

Das trennt die Definition des Views und seiner Logik, von welcher es gesteuert wird.

Controller & Commands

Der Controller hält benannte Mappings zu Command-Klassen, welche zustandslos sind und nur erstellt werden, falls sie benötigt werden.

Commands können mit Proxies interagieren und diese abfragen. Sie können Notifications senden und weitere Commands ausführen. Auch werden sie oft benutzt, um komplexe oder systemweite Aktionen auszuführen, wie z.B. das Starten oder Beenden einer Applikation. Commands stellen innerhalb der Applikation den Bereich der Business Logic dar.

Façade & Core

Die Façade, ein weiteres Singleton, initialisiert die Hauptakteure (Model, View und Controller). Sie bietet den Ort, um alle öffentlichen Methoden der Hauptakteure aufzurufen.

Mit der Vererbung der Façade durch eine konkrete Façade, erhält die Applikation alle Leistungen der Hauptakteure, ohne jedoch diese zu importieren oder mit ihnen direkt arbeiten zu müssen. Dazu ist lediglich einmal eine konkrete Façade in die Applikation zu implementieren.

PureMVC Aufbau

Façade & Core

Proxies, Mediators und Commands können so die konkrete Façade der Applikation nutzen, um untereinander zu kommunizieren.

Observers & Notifications

PureMVC Applikationen können in Umgebungen laufen, welche keine Event- oder EventDispatcher-Klassen von Flash benutzen. Aus diesem Grund benutzt das Framework ein eigenes Beobachter-Benachrichtigungs-Model, um zwischen den Hauptakteuren von MVC und anderen Teilen des Systems auf eine lose gekoppelte Art zu kommunizieren.

Über die Details des Beobachter-Benachrichtigungs-Models innerhalb von PureMVC braucht man sich keine weiteren Gedanken machen. Es ist eine interne Angelegenheit des Frameworks. Um eine Notification von einem Proxie, Mediator, Command oder von der Façade selbst zu versenden, ist lediglich ein Aufruf einer einfachen Methode notwendig. Dazu muss nicht einmal eine Instanz einer Notification erstellt werden.

Notifications können benutzt werden, um Commands auszuführen

Innerhalb der konkreten Façade werden Commands mit den Namen der Notifications gemappt. Dadurch führt der Controller automatisch den Command aus, sobald die zuvor gemappte Notification versendet wird. Commands führen üblicherweise komplexe Interaktionen zwischen den Interessen des Views und des Models aus, obwohl sie über beide so wenig wie möglich wissen.

PureMVC Aufbau

Mediators versenden, empfangen und bekunden Interesse an Notifications

Sobald die Mediators mit dem View registriert werden, werden ihre Interessen an Notifications über das Aufrufen der Methode 'listNotifications' abgefragt. Diese Methode muss als Rückgabewert ein Array mit den Namen von Notifications geben, an denen ein Mediator interessiert ist.

Später, wenn eine Notification von irgendeinem Akteur im System versendet wird, wird bei den Mediators, welche ihr Interesse über den Namen dieser Notification angegeben haben, die Methode 'handleNotification' aufgerufen. Dieser Methode wird eine Referenz zu der Notification übergeben.

Proxies senden, aber empfangen keine Notifications

Proxies können Notification aus unterschiedlichen Gründen versenden: Zum Beispiel benachrichtigt ein Remote Service Proxy das System, wenn ein Ergebnis vorliegt. Oder ein Proxy, von welchem die Daten aktualisiert wurden, versendet eine 'change' Notification.

Ein Proxy, welcher auf eine Notification hört, ist zu sehr an den Bereichen von View und Controller gekoppelt.

Diese beiden Bereiche wiederum müssen unbedingt auf Notifications von Proxies hören, falls sie die Funktion haben, Model-Daten, welche von den Proxies gehalten werden, zu visualisieren oder dem User eine Interaktion mit diesen Daten zu ermöglichen.

Dennoch sollten die Bereiche von View und Controller auch veränderbar sein, ohne dass der Bereich des Models in Mitleidenschaft gezogen wird.

PureMVC Aufbau

Proxies senden, aber empfangen keine Notifications

Zum Beispiel: Eine Administration- und eine dazugehörige User-Anwendung teilen sich möglicherweise die gleichen Klassen im Bereich des Models. Falls sich nur die Anwendungsfälle unterscheiden, können diese über unterschiedliche View- und Controller-Zusammenstellungen mit dem gleichen Model ausgeführt werden.

Façade

Die drei Hauptakteure des MVC Architekturmusters werden bei PureMVC durch die Model-, View- und Controller-Klassen repräsentiert. Um den Prozess der Anwendungsentwicklung zu vereinfachen, verwendet PureMVC das Façade Entwurfsmuster.

Die Façade vermittelt die Anfragen zum Model, View und Controller. Dadurch müssen die Klassen der Hauptakteure nie im Code importiert werden, auch wird nicht mit diesen Klassen direkt gearbeitet. Die Façade instanziiert automatisch die Haupt-MVC-Singletons in ihrem Konstruktor.

Üblicherweise wird die Façade des Frameworks durch eine konkrete Façade vererbt, welche den Controller mit dem Command Mapping initialisiert. Die Bereitstellung von Model und View erfolgt über Commands, welche von dem Controller ausgeführt werden.

Façade

Was ist eine konkrete Façade?

Auch wenn die Hauptakteure komplett und einsetzbar implementiert sind, bietet die Façade eine abstrakte und wohlüberlegte Implementierung, damit diese niemals direkt instanziiert werden muss.

Stattdessen wird eine über die Vererbung der Framework Façade eine konkrete Façade erstellt, welche Methoden überschreibt oder auch hinzufügt, damit sie in der Applikation nützlich ist.

Über diese *konkrete* Façade werden Commands, Mediators und Proxies aufgerufen und benachrichtigt. Damit erledigt sie die eigentliche Arbeit des Systems. Gewöhnlich wird die konkrete Façade 'ApplicationFacade' genannt. Aber man kann sie auch bennen, wie man möchte.

Generell wird die View Hierarchie (Display Komponenten) nach einem Prozess erstellt, welcher von der Plattform abhängt. In Flex instantiiert eine MXML Applikation ihre Children oder ein Flashfilm erstellt alle Objekte auf seiner Bühne. Sobald die View Hierarchie der Applikation erstellt ist, wird das PureMVC System gestartet und die Bereiche des Models und Views für die Benutzung vorbereitet.

Die konkrete Façade wird ebenso eingesetzt, um den Startup Prozess in einem Weg zu ermöglichen, dass die Hauptapplikation so gut wie nichts von dem PureMVC System weiß. Dazu wird lediglich eine Referenz der Applikation über die 'startup' Methode der Singleton-Instanz der konkreten Façade angegeben.

Façade

Erstellen einer konkreten Façade in einer Applikation

Die konkrete Façade braucht nicht viel zu tun, um die Applikation mit genügend Power zu versehen. Dazu folgende Implementierung:

ApplicationFacade.as:

```
package com.me.myapp
{
	import org.puremvc.as3.interfaces.*;
	import org.puremvc.as3..patterns.facade.*;

	import com.me.myapp.view.*;
	import com.me.myapp.model.*;
	import com.me.myapp.controller.*;

	// A concrete Facade for MyApp
	public class ApplicationFacade extends Façade implements IFacade
	{
		// Define Notification name constants
		public static const STARTUP:String		= "startup";
		public static const LOGIN:String		= "login";

		// Singleton ApplicationFacade Factory Method
		public static function getInstance() : ApplicationFacade
		{
			if ( instance == null ) instance = new ApplicationFacade( );
			return instance as ApplicationFacade;
		}

		// Register Commands with the Controller
		override protected function initializeController( ) : void
		{
			super.initializeController();
			registerCommand( STARTUP,  StartupCommand );
			registerCommand( LOGIN,  LoginCommand );
			registerCommand( LoginProxy.LOGIN_SUCCESS,  GetPrefsCommand );
		}

		// Startup the PureMVC apparatus, passing in a reference to the application
		public function startup( app:MyApp ) : void
		{
			sendNotification( STARTUP, app );
		}
	}
}
```

Façade

Erstellen einer konkreten Façade in einer Applikation

Ein paar Erläuterungen zum vorangegangenen Code:

- Die 'ApplicationFacade' erbt von der PureMVC Façade, welche das 'IFacade' Interface implementiert.

- Die 'ApplicationFacade' überschreibt nicht den Konstruktor. Wenn sie dies täte, müsste sie den Konstruktor der Super-Klasse aufrufen, bevor sie irgendetwas machen kann.

- Die 'ApplicationFacade' definiert eine statische 'getInstance' Methode, welche die Singleton-Instanz zurückgibt, die zuvor erstellt und ggf. zwischengespeichert wird. Die Referenz zu der Instanz wird in einer 'protected' Eigenschaft in der Super-Klasse (Façade) gehalten und muss in der Sub-Klasse gecastet werden, bevor sie zurückgegeben wird.

- Die 'ApplicationFacade' definiert die Namen der Notifications als Konstanten. Da die 'ApplicationFacade' der zentrale Ort für alle anderen im Systems ist, um sich untereinander zu verbinden und miteinander zu kommunizieren, ist sie auch der perfekte Ort für diese Konstanten, welche von den Partizipanten der Notifications benutzt werden.

- Die 'ApplicationFacade' initialisiert den Controller mit Commands. Diese Commands werden ausgeführt, sobald die zugeordneten Notifications versendet werden.

Façade

Erstellen einer konkreten Façade in einer Applikation

- Die ApplicationFacade stellt eine 'startup' Methode zu Verfügung, welche als Argument eine Referenz zur Applikation erhält (in diesem Fall vom Typ 'MyApp'). Durch das Versenden der 'STARTUP' Notification wird der 'StartupCommand' ausgelöst und ebenfalls eine Referenz zur Applikation übergeben. Der 'StartupCommand' ist dem Notification Namen STARTUP zugeordnet.

Mit dieser einfachen Implementierung von Anforderungen übernimmt die konkrete Façade ziemlich viel von der Funktionalität ihrer Abstrakten Super-Klasse.

Initialisierung einer konkreten Façade

Der Konstruktor der PureMVC Façade ruft die 'protected' Methoden für die Initialisierung der Model-, View- und Controller-Instanzen auf und hält die Referenzen zu ihnen.

Mittels Komposition implementiert die Façade die Features von Model, View und Controller und stellt sie dadurch frei zu Verfügung. Damit vereinigt die Façade die Funktionen der Hauptakteure des Frameworks und hält zugleich die Entwickler vor der direkten Interaktion mit ihnen ab.

Wo und wie wird die Façade in einer tatsächlichen Applikation passend eingebracht? Betrachten Sie dazu die folgende Flex Applikation:

Façade

Initialisierung einer konkreten Façade

MyApp.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="façade.startup(this)”>

    <mx:Script>
    <![CDATA[
            // Get the ApplicationFacade
            import com.me.myapp.ApplicationFacade;
            private var facade:ApplicationFacade = ApplicationFacade.getInstance();
    ]]>
    </mx:Script>

    <!—Rest of display hierarchy defined here -->
</mx:Application>
```

Das ist alles. Super einfach.

Aufbau der ersten View Hierarchie, Instanziierung der ApplicationFaçade und Ausführen ihrer 'startup' Methode.

Beachte: In AIR würden wir ‘applicationComplete’ nehmen. Und in Flash könnten wir die Façade instanziieren und den 'startup' Aufruf im Frame 1 oder in einer seperaten Dokumenten-Klasse ausführen.

Folgende grundlegende Dinge sind bei dem oberen Beispiel zu beachten:

- Üblicherweise wird das Interface über MXML erstellt, welcher mit einem <mx:Application> Tag beginnt und weitere Standard- oder speziell angefertigte Komponenten und Container enthält.

- Innerhalb des Skript Blocks wird eine 'private' Variable für die Singleton-Instanz der konkreten 'ApplicationFacade' erstellt und intialisiert.

Façade

Initialisierung einer konkreten Façade

- Beim Abfeuern des 'creationComplete' Events der Applikation ist die 'ApplicationFacade' bereits über den Aufruf der statischen 'ApplicationFacade.getInstance' Methode initialisiert und ebenso Model, View und Controller bereits erstellt. Zu diesem Zeitpunkt sind allerdings noch keine Mediators oder Proxies kreiert.

- Beim 'creationComplete' Event Handler im Application-Tag wird die 'startup' Methode der 'ApplicationFacade' ausgeführt und eine Referenz zur Hauptapplikation mitgegeben.

Dabei ist zu beachten, dass die normalen View Components nichts über die Façade zu wissen brauchen und auch nicht mir ihr interagieren. Eine Ausnahme stellt die Top-Level Applikation.

Die Top-Level Applikation (oder Flash-Film) erstellt die View Hierarchy, initialisiert die Façade und startet das PureMVC System.

Notifications

PureMVC verwendet das Beobachter-Entwurfsmuster (Observer pattern), damit die Hauptakteure und ihre Kollaborateure in einer losen gekoppelten Art miteinander kommunizieren können, ganz unabhängig von der verwendeten Plattform.

ActionScript verwendet kein Event-Model, welches in Flex und Flash gleich verwendet wird - es stammt von Flash.

Notifications

Das PureMVC Framework ist mittlerweile für weitere Plattformen, wie z.B. C# und J2ME, portiert worden und verwaltet dadurch eine eigene interne Kommunikation, um nicht von dem Event-Model der Flash Plattform abhängig zu sein.

Es ist nicht nur ein Ersatz für Events, sondern Notifications agieren auf einem ganz anderen Weg. Sie arbeiten mit Events zusammen, damit extrem wiederverwendbare View Components erstellt werden können, die nichts von dem PureMVC System und auch nichts von der Koppelung zu dem System wissen. Vorausgesetzt, sie sind richtig gebaut.

Events vs. Notifications

Events werden von Flash Display-Objekten versendet, welche das 'IEventDispatcher' Interface implementiert haben. Das Event kann durch die Display Hierarchy 'blubbern' und so dem Eltern-Objekt, dem Eltern-Eltern-Objekt usw. erlauben, auf das Event zu reagieren.

Das ist der sogenannte '*chain of responsibility* mechanism', wodurch nur diejenigen in der Eltern/Kind Abstammung die Möglichkeit haben, auf das Event zu reagieren, die eine Referenz zu dem Dispatcher besitzen, um sich als Listener anzumelden.

Notifications werden durch die Façade und Proxies versendet. Ebenso können sie von Mediators empfangen und versendet werden. Notification können zu Commands gemappt werden und Commands können wiederum auch Notifications versenden. Dahinter verbirgt sich ein Veröffentlichen-/Abonnieren-Mechanismus, wobei beliebige Observer (Beobachter) die gleiche Notification empfangen und darauf reagieren können.

Notifications

Events vs. Notifications

Notifications haben optional einen 'body', welcher ein beliebiges ActionScript-Objekt sein kann.

Es ist nicht wie bei Flash Events notwendig, eine eigene Notification-Klasse zu erstellen, da die Notifications 'out of the box' bereitgestellt werden. Dennoch können auch eigene Notification-Klassen erstellt werden, um eine strenge Typisierung sicherzustellen. Allerdings sollte man dabei abwägen, inwieweit die enorme Überzahl an zusätzlichen Notification-Klassen die eingeschränkten Vorteile einer Typen-Überprüfung zur Kompilierzeit (im Besonderen bei Notifications) rechtfertigt. Das ist eine Frage des Programmierstils.

Notifications haben außerdem einen optionale Eigenschaft 'type', welche zur Unterscheidung bei den Empfänger benutzt werden kann.

Zum Beispiel: Bei einer Applikation zur Dokumenten-Bearbeitung kann es eine Proxy-Instanz für jedes Dokument, welches geöffnet ist, und einen Mediator für die dazugehörige View Component, welche das Editieren des Dokuments ermöglicht, geben. Proxy und Mediator teilen sich dabei eindeutige Keys, welche von dem Proxy als 'type' Eigenschaft der Notification versendet werden.

Alle Mediators, welche sich für die Notification des Proxies angemeldet haben, werden beim Versenden dieser informiert. Dabei können sie zusätzlich über die 'type' Eigenschaft der Notification entscheiden, ob sie darauf reagieren oder nicht.

Notifications

Definieren von Notifications- und Event-Konstanten

Wie wir gesehen haben, ist die konkrete Façade ein guter Ort um Konstanten für Notifications zu definieren. Da die konkrete Façade der zentrale Machanismus für die Interaktion mit dem System ist, werden alle Beteiligten von Notifications standardmäßig auch Mitwirkende bei der Façade.

Um die Definitionen der Konstanten allein und auch in anderen Applikationen zu verwenden, kann man auch eine separate 'ApplicationConstants' Klasse erstellen, anstatt alle Konstanten in der konkreten Façade zu halten.

In jedem Fall stellen die Namen von Notifications durch zentral definierte Konstanten eine Typisierung sicher. Der Kompiler kann diese überprüfen, wogegen bei der Verwendung von einfachen Zeichenketten bei Schreibfehlern keine Fehlermeldungen auftauchen werden.

Namen von Events sind niemals in der konkreten Façade zu erstellen. *Event-Namen sind nur in den Klassen, welche die Events generieren oder in den Event-Klassen selbst zu erstellen.*

Aufgrund der äußerlichen Grenzen der Applikation, sollten View Components und Daten-Objekte nur über Events mit ihren Mediator's oder Proxies kommunizieren. Sie sollten keine Notification versenden und auch keine Methoden von ihren Mediators oder Proxies aufrufen.

Wenn eine View Component oder ein Daten-Objekt einen Event abfeuert, welcher vom dazugehörigen Mediator oder Proxy abgehört wird, dann brauchen nur die Beteiligten den entsprechendem Event-Namen kennen. Weitere Kommunikation zwischen den Empfängern und dem Rest des PureMVC Systems erfolgt über Notifications.

Notifications

Definieren von Notifications- und Event-Konstanten

Die Beziehung dieser Paare (Mediator/ViewComponent und Proxy/Data-Object) ist notwendigerweise etwas eng, zum Rest der Applikation jedoch lose gekoppelt. Das schafft Raum für mehr eigenständiges Refactoring der Model-Daten oder dem User-Interface, falls es notwendig wird.

Commands

Die konkrete Façade initialisiert üblicherweise den Controller über das Mapping von Notification und Commands, welche beim Start benötigt werden.

Für jedes Mapping registriert sich der Controller selbst als Observer (Beobachter) für die vorgegebenen Notifications. Sobald der Controller informiert wird, instanziiert er den Command, welcher der Notification zugeordnet wurde, und ruft die 'execute' des Commands auf. Dabei wird dem Command die Notification übergeben.

Commands sind zustandslos. Sie werden nur erstellt, wenn sie benötigt werden und verschwinden wieder, sobald sie ausgeführt wurden. Aus diesem Grund ist es wichtig, dass niemals ein Command instanziiert und auch keine Referenz in 'länger-lebenden' Objekten gehalten wird.

Commands

Verwendung von Macro- und SimpleCommands

Wie alle Klassen des PureMVC Frameworks implementieren Commands ein Interface. Es ist das 'ICommand' Interface. PureMVC beinhaltet zwei 'ICommand' Implementationen, welche sich leicht erweitern lassen.

Die 'SimpleCommand' Klasse hat lediglich eine 'execute' Methode, welche eine 'Inotification' Instanz akzepiert. Alles was zu tun ist, ist eine 'execute' Methode im Code einzufügen.

Die 'MacroCommand' Klasse erlaubt das sequenzielle Ausführen von mehreren Subcommands, denen ebenfalls eine Referenz zu der original Notification übergeben wird.

Ein 'MacroCommand' ruft seine 'initializeMacroCommand' Methode innerhalb seines Konstruktors auf. Diese Methode ist von allen Unterklassen zu überschreiben, um darin die 'SubCommands' über die 'addSubCommand' Methode hinzuzufügen. Dabei können beliebige Kombinationen aus 'Simple-' oder 'MacroCommands' hinzugefügt werden.

Loses Koppeln von Commands zu Mediators und Proxies

Commands werden durch den Controller als Ergebnis einer gesendeten Notification ausgeführt. Commands sollten nur vom Controller und niemals von einem anderen Akteur instanziiert und ausgeführt werden.

Um mit anderen Teilen des Systems zu kommunizieren und zu interagieren, können Commands:

Commands

Loses Koppeln von Commands zu Mediators und Proxies

- die bestehenden Registrierungen von Mediators, Proxies und Commands überprüfen, diese entfernen oder registrieren
- Notifications versenden, welche von anderen Commands oder Mediators beantwortet werden
- Proxies und Mediators abfragen und diese direkt manipulieren

Durch Commands können recht einfach die Elemente vom View in einen bestimmten State gebracht oder Daten in verschiedene Teile des Views transportiert werden.

Sie können auch benutzt werden, um transaktionsbezogene Interaktionen mit dem Model auszuführen, welche über mehrere Proxies verteilt sind und Notifications versendet werden müssen, sobald die gesamte Transaktion komplett ist. Oder auch zum Verarbeiten von Ausnahmen und weiteren Aktionen bei Fehlern.

Inszenierung von komplexen Aktionen und die 'Business Logic'

An verschiedenen Orten wird innerhalb der Applikation Code platziert (Commands, Mediators und Proxies). Die Frage wird immer wieder und notwendigerweise sein:

Welcher Code gehört wohin? Was genau sollte ein Command tun?

Der erste Schritt ist die Unterscheidung der Applikation in eine 'Business Logic' und in eine 'Domain Logic'.

Commands

Inszenierung von komplexen Aktionen und die 'Business Logic'

Commands beherbergen die 'Business Logic' der Applikation. Technisch gesehen bedeutet die Implementation von Anwendungsfällen in einer Applikation das Abfragen des 'Domain Models'. Das beinhaltet die Koordination der Zustände von Model und View.

Das Model hält seine Ganzheit durch die Verwendung von Proxies aufrecht. Diese beinhalten die 'Domain Logic' und bieten eine API für die Manipulation von Daten-Objekten. Damit kapseln sie den gesamten Zugriff auf das Daten-Model, welches sich entweder auf Client- oder Serverseite befindet. Der Rest der Applikation kann dadurch auf die Daten synchron oder asynchron zugreifen.

Commands können für komplexe Systemverhalten eingesetzt werden, welche in einer besonderen Reihenfolge erfolgen müssen und, falls es möglich ist, das Ergebnis von der einen für die nächste Aktion zuführen.

Mediators und Proxies sollten den Commands (und jedem anderen auch) ein '*course-grained*' Interface offenlegen, welches die Implementation der zugeordneten View Components oder Data Objects verdeckt.

Wenn von View Components die Rede ist, sind zum Beispiel ein Button oder ein Widget gemeint, mit denen der User direkt interagiert. Mit Data Objects sind Objekte gemeint, welche beliebige Strukturen aufweisen können. Diese Objekte beinhalten Daten oder auch Remote-Services, welche aufgerufen oder gehalten werden können.

Commands

Inszenierung von komplexen Aktionen und die 'Business Logic'

Commands interagieren mit Mediators und Proxies, sollten aber isoliert von den angrenzenden Implementationen sein. Betrachten Sie dazu die folgenden Commands, welche zur Vorbereitung des Systems eingesetzt werden:

StartupCommand.as:

```
package com.me.myapp.controller
{
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.command.*;

    import com.me.myapp.controller.*;

    // A MacroCommand executed when the application starts.
    public class StartupCommand extends MacroCommand
    {
        // Initialize the MacroCommand by adding its subcommands.
       override protected function initializeMacroCommand() : void
       {
              addSubCommand( ModelPrepCommand );
              addSubCommand( ViewPrepCommand );
       }
    }
}
```

Das ist ein 'MacroCommand', der zwei Subcommands hinzufügt. Sobald der 'MacroCommand' ausgeführt wird, werden die Subcommands in der angegebenen Reihenfolge nacheinander ausgeführt.

Das bildet einen Top Level Ablauf von Aktionen, welche beim Start durchgeführt werden. Aber was sollte genau und in welcher Reihenfolge gemacht werden?

Bevor dem User irgendwelche Daten der Applikation präsentiert werden können und bevor der User mit diesen Daten interagieren

Commands

Inszenierung von komplexen Aktionen und die 'Business Logic'

kann, muss das Model in einen konsistenten und bekannten Zustand gebracht werden. Sobald dieser Zustand erreicht ist, kann der View vorbereitet werden, um die Daten des Models darzustellen. Dann kann der User mit diesen Daten interagieren und manipulieren.

Dafür ist es normalerweise notwendig, beim Start der Applikation zwei größere Aktivitäten auszuführen: Die Vorbereitung des Models, gefolgt von der Vorbereitung des Views.

ModelPrepCommand.as:

```
package com.me.myapp.controller
{
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.observer.*;
    import org.puremvc.as3.patterns.command.*;

    import com.me.myapp.*;
    import com.me.myapp.model.*;

    // Create and register Proxies with the Model.
    public class ModelPrepCommand extends SimpleCommand
    {
        // Called by the MacroCommand
        override public function execute( note : INotification ) : void
        {
                facade.registerProxy( new SearchProxy() );
                facade.registerProxy( new PrefsProxy() );
                facade.registerProxy( new UsersProxy() );
        }
    }
}
```

Die Vorbereitung des Models besteht normalerweise nur aus einer einfachen Erstellung und Registierung von Proxies, welche beim Start benötigt werden.

Commands

Inszenierung von komplexen Aktionen und die 'Business Logic'

Der obere 'ModelPrepCommand' ist ein 'SimpleCommand', welcher die Verwendung des Models vorbereitet. Da er der erste Subcommand des 'MacroCommands' ist, wird er zuerst ausgeführt.

Mit Hilfe der konkreten Façade werden die Proxies erstellt und registriert, welche das System zu Beginn benötigt. Beachten Sie, dass der Command dabei keine Manipulation oder Initialisierung der Model Daten vornimmt. Dafür sind die Proxies selbst verantwortlich, wozu auch das Abfragen von Daten und die Vorbereitung der Data Objects, welche von dem System verwendet werden, zählen.

ViewPrepCommand.as:

```
package com.me.myapp.controller
{
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.observer.*;
    import org.puremvc.as3.patterns.command.*;

    import com.me.myapp.*;
    import com.me.myapp.view.*;

    // Create and register Mediators with the View.
    public class ViewPrepCommand extends SimpleCommand
    {
        override public function execute( note : INotification ) : void
        {
            var app:MyApp = note.getBody() as MyApp;
            facade.registerMediator( new ApplicationMediator( app ) );
        }
    }
}
```

Der 'ViewPrepCommand' ist ein 'SimpleCommand', welcher die Verwendung der Views vorbereitet. Es ist der letzte Subcommands innerhalb des o.g. 'MacroCommands' und wird deshalb auch zuletzt ausgeführt.

Commands

Inszenierung von komplexen Aktionen und die 'Business Logic'

Beachten Sie, dass dabei nur ein Mediator erstellt und registriert wird. Es ist der 'ApplicationMediator', welcher der View Component 'Application' zugeordnet ist.

Dem Konstruktor des Mediators wird ferner die 'body' Eigenschaft der Notification übergeben. Das ist eine Referenz zur Applikation, welche die Applikation selbst als 'body' Eigenschaft der STARTUP Notification angab, als diese versendet wurde. Vergleichen Sie dazu das vorherige 'MyApp' Beispiel.

Die Applikation ist soetwas wie eine spezielle View Component, in welcher sie selbst instanziiert wird und als Children alle anderen View Components trägt, welche ebenfalls beim Start der Applikation initialisiert werden.

Um mit dem Rest der Applikation zu kommunizieren, braucht eine View Component einen Mediator. Und um diese Mediators zu erstellen, sind Referenzen zu den View Components notwendig, welche vermittelt werden sollen. Zu diesem Zeitpunkt kennt nur die Applikation diese Referenzen.

Nur dem Mediator der Applikation ('ApplicationMediator') ist es erlaubt, alles über die Implementation der Applikation zu wissen. Darum werden die restlichen Mediatoren innerhalb des Konstruktors des 'ApplicationMediators' erstellt.

Mit den oberen drei Commands sind die Initialisierungen des Models und des Views in einer bestimmten Reihenfolge vorgenommen worden. Um dies zu tun, brauchten die Commands nicht viel über Model oder View zu wissen.

Commands

Inszenierung von komplexen Aktionen und die 'Business Logic'

Falls die Inhalte des Models oder die Implementation der Views geändert werden, müssen die Proxies und die Mediatoren nur so wenig wie nötig verändert.

Die 'Business Logic' in den Commands sollte isoliert von den Veränderungen der angrenzenden Bereichen bleiben.

Das Model sollte die 'Domain Logic' kapseln und die Datenintegrität in den Proxies beibehalten. Commands führen 'transaktionale-' oder 'business Logic' gegen das Model aus, kapseln die Koordination von Transaktionen mit mehreren Proxies und verarbeiten oder melden Fehler.

Mediators

Eine Mediator-Klasse wird benutzt, um zwischen den Interaktionen zwischen Usern und einer oder mehreren View Components der Applikation (z.B. Flex DataGrids oder Flash MovieClips) und dem Rest der PureMVC Anwendung zu vermitteln.

In einer Flash basierten Applikation beinhaltet ein Mediator typischerweise Event Listener für seine View Components, um auf Usergesten und Datenanfragen der Component zu reagieren. Ein Mediator sendet und empfängt Notifications, um mit dem Rest der Applikation zu kommunizieren.

Mediators

Zuständigkeit des konkreten Mediators

Flash, Flex und AIR liefern eine Unzahl an hervorragenden UI Components. Möglicherweise werden diese erweitert oder es werden eigene Komponenten in ActionScript geschrieben, um die endlosen Möglichkeiten für die Darstellung von Model-Daten auszuschöpfen und dem User es zu erlauben, mit diesen zu interagieren.

In der nicht allzu fernen Zukunft werden es weitere Plattformen geben, welche nicht auf ActionScript basieren. PureMVC ist bereits in andere Plattformen portiert worden, inklusive für Silverlight und J2ME, um den Horizont für RIA Anwendungen für diese Technologien zu erweitern.

Ein Ziel des PureMVC Frameworks ist es, neutral gegenüber der eingesetzten Technologie zu sein. Dafür bietet es einfache Idiome, um beliebige UI Components oder Datenstrukturen und -services anzupassen.

Eine View Component kann in einer PureMVC Applikation eine beliebige UI Component sein. Ganz unabhängig davon, von welchem Framework diese stammt und wieviele Subcomponents sie beinhaltet. Eine View Component sollte soviel wie möglich von ihrem Zustand und ihrer Funktionsweise kapseln und eine einfache API über Events, Methoden und Eigenschaften bieten.

Ein konkreter Mediator hilft dabei, mit einer oder mehreren View Components über ihre APIs zu interagieren. Der Mediator hält dafür lediglich die Referenzen zu den View Components.

Die Zuständigkeit der Mediators liegt vor allem in dem Handeln auf Events und Notifications, welche von den View Components bzw. vom Rest des System versendet werden.

Mediators

Zuständigkeit des konkreten Mediators

Da die Mediators oft mit Proxies interagieren, ist es üblich, innerhalb des Konstruktors eine lokale Referenz zu den oft benötigten Proxies zu erstellen. Das verringert das ständige Aufrufen der 'retrieveProxy' Methode über die Façade.

Erforderliches Casten der View Component

Bei der normalen Einbindung von Mediators innerhalb von PureMVC akzeptiert ein Mediator beim Aufruf seines Konstruktors zwei Argumente: Seinen Namen und ein generisches Object.

Der Konstruktor des konkreten Mediator übergibt der Superklasse das generische Object als seine View Component, welche intern als 'protected' Eigenschaft 'viewComponent' von Typ 'Object' benutzt wird.

Auch ist es möglich, dem Mediator dynamisch eine Referenz zur View Component zu setzen, nachdem er bereits instanziiert wurde. Dafür dient die Methode 'setViewComponent'.

Wie auch die View Component gesetzt wird, sie sollte immer in ihren Typ gecastet werden, um den Zugriff zu ihrer API sauber zu ermöglichen.

ActionScript bietet ein Feature, welches 'implicit getters and setters' genannt wird. Ein 'implicit getter' sieht aus wie eine Methode, wird aber wie eine Eigenschaft innerhalb der Klasse oder einer Applikation behandelt. Das ist sehr hilfreich, um das häufige Casting-Problem der View Components zu lösen.

Mediators

Erforderliches Casten der View Component

Für Anwendung eines konkreten Mediators ist es ein hilfreiches Idiom, einen 'implicit getter' für das Casten der View Component in ihren Typ zu verwenden und ihr damit einen aussagekräftigen Namen zu geben.

Wie zum Beispiel beim Erstellen folgender Methode:

```
protected function get controlBar() : MyAppControlBar
{
    return viewComponent as MyAppControlBar;
}
```

Dann, irgendwo im Mediator, kann anstatt:

```
MyAppControlBar ( viewComponent ).searchSelction = MyAppControlBar.NONE_SELECTED;
```

dies hier getan werden:

```
controlBar.searchSelction = MyAppControlBar.NONE_SELECTED;
```

View Components zuhören und antworten

Ein Mediator hat normalerweise nur eine View Component. Aber er kann auch mehrere managen, wie beispielsweise eine 'ApplicationToolBar' mit den enthaltenen Buttons und anderen Komponenten. Es können zusammengehörige Komponenten (z.B. ein Formular) in einer einzelnen View Component als Gruppe zusammengefasst werden, welche den Zugriff bestimmter Elemente über Eigenschaften dem Mediator zu Verfügung stellt. Aber es ist das Beste, soviel wie möglich von der Implementierung der View Componente zu kapseln und die auszutauschenden Daten der View Component als eigene typisierte Objekte zu Verfügung zu stellen.

Mediators

View Components zuhören und antworten

Der Mediator erledigt die Interaktion mit den Bereichen des Controllers und Models. Außerdem aktualisiert er die View Component, wenn er relevante Notifications empfängt.

Unter der Flash-Plattform wird typischerweise der Mediator als Event Listener bei der View Component angemeldet, sobald der Mediator instanziiert ist oder die Methode ' setViewComponent' aufgerufen wurde. Dazu dient das Hinzufügen von Handler-Methoden, wie in diesem Beispiel:

```
controlBar.addEventListener( AppControlBar.BEGIN_SEARCH, onBeginSearch );
```

Was der Mediator genau macht, um auf die Events zu reagieren, hängt von den genauen Umständen ab.

Allgemein reagiert ein konkreter Mediator auf Events über Handler-Methoden, um einige Kombinationen der folgenden Aktionen auszuführen:

- Prüfen des Events auf seinen Typ oder auf eigenen Inhalt, falls dieser erwartet wird
- Prüfen oder Anpassen offener Eigenschaften (oder Aufrufen offener Methoden) der View Component
- Prüfen oder Anpassen offener Eigenschaften (oder offener Methoden) von einem Proxy
- Versenden von einer oder mehreren Notifications, auf welche andere Mediators oder Commands (oder möglicherweise diegleiche Medatior-Instanz) reagieren

Mediators

View Components zuhören und antworten

Einige gute Faustregeln sind:

- Wenn eine Anzahl von anderen Mediators in die Reaktion auf einem Event eingebunden sein sollen, dann ist ein bestimmter Proxy zu aktualisieren oder eine Notification zu senden, auf welche alle anderen interessierten Mediators entsprechend reagieren können.

- Falls eine große Menge an koordinierten Interaktionen mit anderen Mediators erforderlich ist, ist es eine gute Praktik, einen Command einzusetzen, um die erforderlichen Schritte an einem Ort zu organisieren.

- Es ist eine schlechte Praktik, andere Mediator direkt anzusprechen, um auf diese einzuwirken. Ebenso sollten keine Mediators umprogrammiert werden, um diese Manipulationen zu ermöglichen.

- Um den Zustand der Applikation zu manipulieren oder zu verbreiten, sind in dem Proxy, welcher den Zustand hält, die notwendigen Werte zu verändern oder die entsprechenden Methoden aufzurufen. Der Proxy reagiert auf diese Änderungen mit dem Versenden von Notifications, auf welche alle interessierten Mediators reagieren können.

Mediators

Verarbeiten von Notifications in dem konkreten Mediator

Im Gegensatz zum explizitem Hinzufügen als Event Listener zu der View Component werden die Mediators zum PureMVC-System einfach und automatisch hinzugefügt.

Bei der Registrierung mit dem View wird der Mediator nach seinen Notifications-Interessen befragt, welche er mit einem Array von Namen der Notifications beantwortet, auf die er reagieren möchte.

Der einfachste Weg hierfür ist es, über einen einzelnen Ausdruck ein anonymes Array zurückzugeben, welches die Namen auf die zu reagierenen Notifications enthält, welche üblicherweise als statische Konstanten in der konkreten Façade definiert werden.

Das Definieren der Interessen des Mediators an Notifications ist einfach:

```
override public function listNotificationInterests() : Array
{
    return [
              ApplicationFacade.SEARCH_FAILED,
              ApplicationFacade.SEARCH_SUCCESS
            ];
}
```

Wenn einer dieser angegebenen Notifications von irgendeinem Akteur des Systems (inklusive dem Mediator selbst) versendet wird, wird innerhalb des Mediators die Methode 'handleNotification' aufgerufen, welche die Notification als Argument übergeben wird.

Aufgrund der Lesbarkeit und dem einfachen Refactoring beim Hinzufügen oder Entfernen von Notifications, ist innerhalb der 'handleNotifications' Methode ein 'switch / case' Statement einer 'if / else' Abfrage vorzuziehen.

Mediators

Verarbeiten von Notifications in dem konkreten Mediator

Bei der Beantwortung einer beliebigen Notifications ist immer etwas zu tun und alle dazu benötigten Informationen sollten in der Notification selbst vorhanden sein. Mitunter werden basierend auf den Informationen in der Notification weitere Informationen von einem Proxy benötigt. Aber innerhalb des Notification-Handlers sollte keine komplizierte Logik stattfinden. Falls diese besteht ist es ein Zeichen dafür, dass versucht wurde, die 'Business Logic' in den Notification-Handler, statt in einem Command zu packen.

```
override public function handleNotification( note : INotification ) : void
{
    switch ( note.getName() )
    {
            case ApplicationFacade.SEARCH_FAILED:
                    controlBar.status = AppControlBar.STATUS_FAILED;
                    controlBar.searchText.setFocus();
                    break;

            case ApplicationFacade.SEARCH_SUCCESS:
                    controlBar.status = AppControlBar.STATUS_SUCCESS;
                    break;

            }
    }
}
```

Außerdem sollte eine typische 'handleNotification' Methode nicht mehr als 4 oder 5 Notifications abhandeln.

Eine größere Anzahl von Notifications ist ein Zeichen dafür, dass die Zuständigkeit des Mediators in mehrere Mediators aufgeteilt werden sollte. Statt für alle Subcomponents nur einen gigantischen Mediator zu verwenden, sollten dann für die Subcomponents eigene Mediators erstellt werden.

Mediators

Verarbeiten von Notifications in dem konkreten Mediator

Das Benutzen von einer vorher festgelegten Notification ist der große Unterschied, wie Mediators auf Events und wie auf Notifications hören.

Bei Events werden mehrere Handler-Methoden benötigt. Normalerweise eine für jedes Event, auf welches der Mediator reagieren will. Generell versenden dann diese Methoden nur Notifications. Sie sollten nicht zu kompliziert sein, auch sollten diese nicht zu zu tief in die Details der View Components eingreifen. Deshalb sollten die View Components so geschrieben sein, dass sie die Details ihrer Implementation kapseln und dem Mediator eine API zu Verfügung stellen.

Bei Notifications ist nur eine einzige Handler-Methode notwendig, welche die Notifications abarbeitet, an denen der Mediator interessiert ist.

Es ist das Beste, direkt in der 'handleNotification' Methode auf die Notifications zu reagieren. Mit Hilfe eines 'switch / case' Statement werden die Notifications in ihren Namen unterschieden.

Es gab bisher eine große Debatte um die Verwendung von 'switch / case' Statements. Viele Entwickler erwägen diese weniger einzusetzen, weil alle Fälle innerhalb dergleichen Methode ausgeführt werden. Wie auch immer, die einzige Notification-Methode und der 'switch / case' Stil wurden dafür verwendet, um die Dinge einzugrenzen, welche innerhalb des Mediators passieren. Und es gilt als das empfohlene Konstrukt.

Der Mediator hat die Aufgabe, zwischen der View Component und dem Rest des Stystem zu vermitteln.

Mediators

Verarbeiten von Notifications in dem konkreten Mediator

Denken Sie an die Rolle eines Übersetzers, welcher bei Gesprächen zwischen seinem Botschafter und dem Rest der Mitglieder einer UN-Konferenz vermittelt. Der Übersetzer sollte selten mehr tun, als einfach zu übersetzen, Mitteilungen weiterleiten und ggf. eine angemessene Metapher verwenden. Das gleiche trifft für den Rolle des Mediators innerhalb von PureMVC zu.

Verbindung von Mediators zu Proxies und anderen Mediators

Da eigentlich der View mit der Darstellung der Model-Daten in einer graphischen und interaktiven Weise beauftragt ist, wird eine enge 'one-way' Verbindung zu den Proxies erwartet. Der View muss das Model kennen, aber das Model braucht nichts über einen beliebigen Aspekt des Views zu wissen.

Mediatoren können ungehindert auf Proxies des Models zugreifen, das Data Object über eine API lesen oder manipulieren, welche von dem Proxies ausgestellt wird.

In der gleichen Art und Weise können Mediators Referenzen zu anderen Mediators des Views aufrufen. Der aufgerufene Mediator bietet Möglichkeiten, um gelesen oder manipuliert zu werden.

Allerdings ist das keine gute Praktik. Denn es erhöht die Abhängigkeiten zwischen den Teilen des Views, welche sich negativ auf die Fähigkeiten eines Refactoring von einem Teil des Views auswirken, ohne andere mit zu beinflussen.

Ein Mediator, welcher mit anderen Teilen des Views kommunizieren möchte, sollte eine Notification senden, anstatt andere Mediatoren direkt abzufragen oder zu manipulieren.

Mediators

Verbindung von Mediators zu Proxies und anderen Mediators

Mediators sollten keine Methoden für die Manipulation ihrer zugeordneten View Component(s) bieten und stattdessen nur auf Notifications reagieren, um solche Dinge umzusetzen.

Wenn interne Dinge einer View Component in großem Maße innerhalb eines Mediators manipuliert werden, sollte dies geändert werden. Diese Arbeit sollte in eine Methode der View Component verlegt werden, um die Implementation der View Component so weit wie möglich zu kapseln und somit eine größere Wiederverwendung zu gewährleisten.

Falls Daten der Proxies in großem Maße innerhalb eines Mediators manipuliert werden, sollte dies geändert werden. Dazu wird diese Arbeit in einen Command verlegt und dadurch der Mediator vereinfacht. Mit dem Halten der 'Business Logic' in Commands können zudem andere Teile des Views diese wiederverwenden. Außerdem werden die Verbindungen von View und Model in einem größten möglichen Maße gelöst.

User Interaktion mit View Components und Mediators

Stellen Sie sich eine 'LoginPanel' Komponente vor, welche ein Formular beinhaltet. Dafür gibt es einen 'LoginPanelMediator', welcher es dem User erlaubt, seine Daten über das 'LoginPanel' zu kommunizieren und diese die Interaktion mit dem Beginn des Loginprozesses beantwortet.

Die Zusammenarbeit zwischen der 'LoginPanel' Komponente und dem 'LoginPanelMediator' erfolgt über das Versenden eines TRY_LOGIN Events vom 'LoginPanel', sobald der User seine Daten angegeben hat und versucht, sich einzuloggen. Der

Mediators

User Interaktion mit View Components und Mediators

'LoginPanelMediator' verarbeitet diesen Event mit dem Versenden einer Notification. Diese Notification trägt als 'body' ein 'LoginVO', welches von der 'LoginPanel' Komponente gefüllt wurde.

LoginPanel.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml"
    title="Login" status="{loginStatus}">

    <!—
    The events this component dispatches. Unfortunately we can't use
    the constant name here, because MetaData is a compiler directive
     -->
    <mx:MetaData>
            [Event('tryLogin')];
    </mx:MetaData>

    <mx:Script>
    <![CDATA[
            import com.me.myapp.model.vo.LoginVO;
            // The form fields bind bidirectionally to this object's props
            [Bindable] public var loginVO:LoginVO = new LoginVO();
            [Bindable] public var loginStatus:String = NOT_LOGGED_IN;

            // Define a constant on the view component for event names
            public static const TRY_LOGIN:String='tryLogin';
            public static const LOGGED_IN:String='Logged In';
            public static const NOT_LOGGED_IN:String='Enter Credentials';
    ]]>
    </mx:Script>

    <mx:Binding source="username.text" destination="loginVO.username"/>
    <mx:Binding source="password.text" destination="loginVO.password"/>
```

Mediators

User Interaktion mit View Components und Mediators

```
<!—The Login Form -->
<mx:Form id="loginForm" >
        <mx:FormItem label="Username:">
                <mx:TextInput id="username" text="{loginVO.username}" />
        </mx:FormItem>
        <mx:FormItem label="Password:">
                <mx:TextInput id="password" text="{loginVO.password}"
                displayAsPassword="true" />
        </mx:FormItem>
        <mx:FormItem >
            <mx:Button label="Login" enabled="{loginStatus == NOT_LOGGED_IN}”
                click="dispatchEvent( new Event(TRY_LOGIN, true ));"/>
        </mx:FormItem>
  </mx:Form>
</mx:Panel>
```

Die 'LoginPanel' View Component füllt bei Formular-Eingaben des Users ein neues 'LoginVO' mit Daten. Bei Click des 'Login' Buttons wird ein Event abgefeuert, auf welches der 'LoginPanelMediator' hört.

Das belässt die View Component mit der einfachen Rolle für das Zusammenstellen der Daten. Sobald dies vollendet ist, wird das System benachrichtigt.

Eine noch mehr vollendete Komponente würde erst dann den 'Login' Button aktivieren, sobald die Angaben zu Username und Passwort valide sind.

Die View Component versteckt ihre interne Implementation. Ihre gesamte API, welche von dem Mediator genutzt wird, beinhaltet einen TRY_LOGIN Event, eine 'loginVO'- und die Panel 'loginStatus' Eigenschaft.

Der LoginPanelMediator beantwortet ebenso die LOGIN_FAILED und LOGIN_SUCCESS Notification und setzt daraufhin den LoginPanel status.

Mediators

User Interaktion mit View Components und Mediators

LoginPanelMediator.as:

```
package com.me.myapp.view
{
	import flash.events.Event;
	import org.puremvc.as3.interfaces.*;
	import org.puremvc.as3.patterns.mediator.Mediator;
	import com.me.myapp.model.LoginProxy;
	import com.me.myapp.model.vo.LoginVO;
	import com.me.myapp.ApplicationFacade;
	import com.me.myapp.view.components.LoginPanel;

	// A Mediator for interacting with the LoginPanel component.
	public class LoginPanelMediator extends Mediator implements IMediator
	{

		public static const NAME:String = 'LoginPanelMediator';

		public function LoginPanelMediator( viewComponent:LoginPanel )
		{
			super( NAME, viewComponent );
			LoginPanel.addEventListener( LoginPanel.TRY_LOGIN, onTryLogin );
		}

		// List Notification Interests
		override public function listNotificationInterests( ) : Array
		{
			return [
					LoginProxy.LOGIN_FAILED,
					LoginProxy.LOGIN_SUCCESS
				];
		}

		// Handle Notifications
		override public function handleNotification( note:INotification ):void
		{
			switch ( note.getName() )  {
				case LoginProxy.LOGIN_FAILED:
					LoginPanel.loginVO = new LoginVO( );
					loginPanel.loginStatus = LoginPanel.NOT_LOGGED_IN;
					break;
				case LoginProxy.LOGIN_SUCCESS:
					loginPanel.loginStatus = LoginPanel.LOGGED_IN;
					break;
			}
		}
```

Mediators

User Interaktion mit View Components und Mediators

```
            // User clicked Login Button; try to log in
            private function onTryLogin ( event:Event ) : void  {
                    sendNotification( ApplicationFacade.LOGIN, loginPanel.loginVO );
            }

            // Cast the viewComponent to its actual type.
            protected function get loginPanel() : LoginPanel  {
                    return viewComponent as LoginPanel;
            }
    }
}
```

Beachten Sie, dass der 'LoginPanelMediator' einen Event Listener zum 'LoginPanel' innerhalb seines Konstruktors platziert, um die 'onTryLogin' Methode auszuführen, sobald der User den 'Login' Button geklickt hat. Innerhalb der 'onTryLogin' Methode wird die LOGIN Notification mit der vom User gefüllten 'LoginVO' versendet.

Zuvor wurde der 'LoginCommand' mit der entsprechenden Notification registriert. Dieser Command ruft die 'login' Methode vom 'LoginProxy' auf und übergibt dabei das 'LoginVO'. Der 'LoginProxy' führt das Login mit einem Remote-Service aus und sendet daraufhin eine LOGIN_SUCCESS oder LOGIN_FAILED Notification. Diese Klassen sind am Ende der Kapitels 'Proxies' definiert.

Der 'LoginPanelMediator' hat die LOGIN_SUCCESS und LOGIN_FAILED Notification als seine Notification Interessen gelistet und wird benachrichtigt, wann auch immer diese versendet werden. Mit Empfang setzt er den 'loginStatus' vom 'LoginPanel' bei Erfolg auf LOGGED_IN und bei Misserfolg auf NOT_LOGGED_IN. Das 'LoginVO' wird geleert.

Proxies

Verallgemeinert gesagt wird das Proxy Muster dazu benutzt, um einen Platzhalter für ein Object zu bieten, auf welches kontrolliert zugegriffen werden kann. In einer PureMVC basierten Anwendung wird eine Proxy Klasse speziell dafür genutzt, um einen Teil des Daten-Model der Applikation zu managen.

Ein Proxy kann den Zugang zu lokal erstellten Datenstrukturen in beliebiger Komplexität managen. Das ist das Data Object des Proxys.

In diesem Fall beinhalten die Idiome für die Interaktion mit einem Proxy vermutlich das synchrone Setzen und Abfragen seiner Daten. Dafür stellt ein Proxy alle Eigenschaften und Methoden seines Data Objects oder einer Referenz zu diesem zur Verfügung. Wenn Methoden zum Aktualisieren der Daten ausgeführt werden, kann ein Proxy eine Notification senden, um dem Rest des Systems diese Änderung mitzuteilen.

Ein Remote Proxy kann benutzt werden, um Interaktionen über einen Remote Service für das Abrufen und Speichern von Teilen der Daten zu kapseln. Der Proxy kann ein Objekt, das mit dem Remote Service kommuniziert, und den Zugriff der Daten, welche über den Service gesendet und empfangen werden, verwalten.

In solch einem Fall kann beim Ändern der Daten oder beim Aufruf einer Methode des Proxys eine asynchrone Notification erwartet werden. Diese Notification werden vom Proxy versendet, sobald der Service die Daten vom Remote Endpunkt erhält.

Proxies

Zuständigkeit des konkreten Proxy

Der konkrete Proxy ermöglicht es, einen Teil der Model-Daten zu kapseln, wo auch immer dieser herkommen und was auch immer sein Typ sein mag. Der Proxy verwaltet das Data Object und seinen Zugang durch die Applikation.

Die Implementation der Proxy Klasse innerhalb von PureMVC ist eine einfaches Object mit Daten, welches beim Model registriert werden kann.

Um es allerdings komplett nutzbar zu machen, muss es eine Unterklasse vom Proxy sein und einige für den konkreten Proxy notwendige Funktionalitäten besitzen.

Gebräuchliche Varianten des Proxy Musters beinhalten:

- *Remote Proxy:* Ein konkreter Proxy, der serverseitige Daten verwaltet. Auf diese Daten wird über irgendeine Art von Service zugegriffen.

- *Proxy und Delegate:* Mehrere Proxies benötigen den Zugriff auf das gleiche Service Object. Die Delegate Klasse verwaltet dieses Service Object und kontrolliert den Zugriff sowie die Weiterleitung der Ergebnisse an die Anforderer.

- *Protection Proxy:* Wird benutzt, falls ein Objekt verschiedene Zugriffsrechte benötigt.

- *Virtual Proxy:* Erstellt bei Bedarf ein 'teueres' Objekt

Proxies

Zuständigkeit des konkreten Proxy

- *Smart Proxy:* Lädt beim ersten Zugriff das Data Object in den Speicher. Führt Zählungen von Referenzen durch. Erlaubt das Sperren eines Objects, damit keine anderen Objekte es ändern kann.

Erforderliches Casten des Data Object

Bei der normalen Einbindung von Proxies innerhalb von PureMVC akzeptiert ein Proxy beim Aufruf seines Konstruktors zwei Argumente: Seinen Namen und ein generisches Object. Das Data Object kann nach der Initialisierung des Proxies über die Methode 'setData' dynamisch gesetzt werden

Wie bei dem Mediator und seiner View Component muss auch dieses Objekt regelmäßig in seinen Typ gecastet werden, um den Zugriff auf die Eigenschaften und Methoden zu ermöglichen, die es offenlegt. Im günstigsten Fall ist es nur eine lästige und mühsehlige Praktik, jedoch führt es möglicherweise zu Idiome, welche mehr als nötig von der Implementation des Data Objects offenlegen.

Weil Data Objects oft auch komplexe Strukturen besitzen, werden zu den verschiedenen Teilen dieser Strukturen einfache Referenzen benötigt, welche wiederum typsicher sein sollten.

Wie bereits erwähnt, bietet ActionScript ein Feature namens 'getter and setter'. Damit kann sehr hilfreich das Problem des mehrmaligen Casten und des unzulässigen Wissens über die Implementation des Data Objects gelöst werden.

Proxies

Erforderliches Casten des Data Object

Ein hilfreiches Idiom beim Einsatz des konkreten Proxy ist ein 'implicit getter', welcher das Data Object in seinen eigentlichen Typ castet und ihm einen aussagekräftigen Namen gibt.

Außerdem können über mehrere, unterschiedliche Typen von 'getters' spezielle Teile des Data Objects abgefragt werden.

Zum Beispiel:

```
public function get searchResultAC () : ArrayCollection
{
        return data as ArrayCollection;
}

public function get resultEntry( index:int ) : SearchResultVO
{
        return searchResultAC.getItemAt( index )  as SearchResultVO;
}
```

Statt irgendwo in einem Mediator das folgende zu tun,

```
var item:SearchResultVO =
ArrayCollection ( searchProxy.getData() ).lastResult.getItemAt( 1 ) as SearchResultVO;
```

kann dieses getan werden:

```
var item:SearchResultVO = searchProxy.resultEntry( 1 );
```

Vermeidung von engen Verbinden zu Mediators

Der Proxy wird nicht wie ein Mediator nach seinen Interessen and Notifications abgefragt, auch wird er beim Versenden von Notifications nie benachrichtigt. Denn er sollte nicht mit dem

Proxies

Vermeidung von engen Verbinden zu Mediators

Zustand des Views befasst sein. Stattdessen bietet der Proxy Methoden und Eigenschaften an, um andere Akteure die Möglichkeit zu bieten, ihn zu manipulieren.

Der konkrete Proxy sollte nicht auf Mediatoren zugreifen und diese auch nicht manipulieren, um das System über Änderungen seines Data Objects zu informieren.

Er sollte stattdessen Notifications senden, welche durch Commands oder Mediators beantwortet werden. Wie das System auf diese Notifications reagiert, sollte keine Konsequenzen für den Proxy haben.

Mit dem Freihalten der Model-Ebene von Wissen über die Implementation des Systems, können die Bereiche des Views und Controllers oft geändert werden, ohne den Bereich des Models in Mitleidenschaft zu ziehen.

Bei der Gegenseite sieht es nicht ganz so aus. Es ist schwierig den Bereich des Models zu ändern, ohne View- und mögliche Controller-Bereiche in Mitleidenschaft zu ziehen. Schließlich existieren diese Bereiche dafür, den User mit dem Model interagieren zu lassen.

Kapseln der 'Domain Logic' in Proxies

Eine Änderung auf Model-Ebene hat meistens auch ein Refactoring der View- und Controller-Bereiche zu Folge.

Die Trennung zwischen den Interessen des Models und den kombinierten Interessen von View und Controller kann durch eine Platzierung der 'Domain Logic' in den Proxies erhöht werden.

Proxies

Kapseln der 'Domain Logic' in Proxies

Der Proxy kann nicht nur dazu benutzt werden, um auf Daten zuzugreifen, sondern auch für die Sicherstellung der Daten in einem validen Zustand.

So sollte zum Beispiel die Berechnung einer Umsatzsteuer in einer 'Domain Logic' Funktion erfolgen, welche sich im Proxy befindet, nicht aber in einem Mediator oder Command.

Obgleich dies auch in irgendeinem anderen Platz passieren könnte, ist das Platzieren in einem Proxy nicht nur logisch, sondern es lässt auch die anderen Bereiche leichter und einfacher ändern.

Ein Mediator kann den Proxy und somit die Umsatzsteuer-Funktion aufrufen und einige Formular-Werte übergeben. Aber eine Platzierung dieser Berechung in den Mediator würde die beinhaltete 'Domain Logic' in den Bereich des Views verschieben. Die Berechung der Umsatzsteuer ist ein typischer Fall für das 'Domain Model'. Der View erkennt es als eine Eigenschaft des 'Domain Models', welche abrufbar ist, sobald die entsprechenden Eingaben vorhanden sind.

Stellen Sie sich vor, Ihre Applikation, an der Sie gerade arbeiten, ist eine Rich Internet Anwendung, welche im Browser unter Desktop-Auflösungen zum Einsatz kommt. Nun soll eine neue Version für den Einsatz unter PDA-Auflösungen ausgeliefert werden, mit eingeschränkten Features, aber immer noch mit den gleichen Anforderungen an den Model-Bereich.

Mit der richtigen Trennung von Interessen ist es möglich, den Bereich des Models in seiner Ganzheit wieder zu verwenden und neue View und Controller an diesen einfach zu binden.

Proxies

Kapseln der 'Domain Logic' in Proxies

Vielleicht erscheint das Platzieren der gerade besprochenen Berechnung der Umsatzsteuer in den Mediator effizienter oder einfacher, da lediglich Daten vom Formular genommen, umgerechnet und vielleicht in das Model gepackt werden.

Aber in jeder weiteren Version der Applikation wird diese Berechung in eine neue, ganz andere View dupliziert, anstatt sie mit der Einbindung des Model-Bereich automatisch zu Verfügung zu stellen.

Interaktion mit Remote Proxies

Ein Remote Proxy ist lediglich ein Proxy, welcher sein Data Object serverseitig zu Verfügung gestellt bekommt. Das bedeutet normalerweise, dass der Proxy mit dem Data Object in einer asynchronen Weise interagiert.

Wie der Proxy seine Daten erhält, hängt von der Clientseite, von der serverseitigen Implemenation und von den Vorlieben des Entwicklers ab. In einer Flash- / Flex-Umgebung kommen möglicherweise 'HTTPService', 'WebService', 'RemoteObject', 'DataService' oder auch 'XMLSocket' zum Einsatz.

Abhängig von den Anforderungen muss ein Remote Proxy Anfragen dynamisch senden, sobald eine Eigenschaft verändert oder eine Methode aufgerufen wird. Oder es macht eine einzige Anfrage bei der Instanziierung und stellt für den Zugriff der Daten eine 'getter/setter' Methode zu Verfügung.

Es gibt eine Menge an Optimierungen, welche möglicherweise im Proxy vorgenommen werden müssen, um die Effizienz der serverseitigen Kommunikation zu erhöhen.

Proxies

Interaktion mit Remote Proxies

Es kann ein Caching der Daten sein, um serverseitige Anfragen zu verringern. Oder auch nur das Senden von bestimmten Teilen der Datenstruktur, welche geändert wurden, um den Verbrauch an Bandbreite zu verringern.

Wenn bei einem Remote Proxy ein Request dynamisch von einem anderen Akteuer im System aufgerufen wird, sollte der Proxy eine Notification senden, sobald das Ergebnis vorliegt.

Die Interessierten dieser Notification können die gleichen sein, welche die Anfrage beim Proxy ausgelöst haben oder auch nicht.

Zum Beispiel: Der Prozess für den Aufruf einer Suche über einen Remote Service und das Anzeigen der Ergebnisse könnten so aussehen:

- Eine View Component leitet die Suche mit dem Abfeuern eines Event ein
- Der dazugehörige Mediator reagiert darauf mit dem Aufrufen des Remote Proxy's, um die Eigenschaft 'searchCriteria' zu setzen
- Die 'searchCriteria' Eigenschaft im Proxy ist in Wirklichkeit ein 'implicit setter', welche den Wert speichert und die Suche über einem internen 'HTTPService' einleitet, um dann auf die 'Result' und 'Fault' Events zu reagieren
- Sobald der Service ein 'ResultEvent' abfeuert, reagiert der Proxy mit der Übergabe des Ergebnisses an eine eigene 'public' Eigenschaft

Proxies

Interaktion mit Remote Proxies

- Der Proxy sendet daraufhin eine Notification, welche als 'body' eine Referenz zum Data Object trägt.

- Ein anderer Mediator, welcher zuvor Interesse an dieser Notification bekundet hat, empfängt das Data Object über den 'body' der Notification und setzt es bei seiner zugeordneten View Component als Wert der 'dataProvider' Eigenschaft.

Oder denken Sie an einen 'LoginProxy', welcher ein LoginVO hält (VO = 'Value Object', welches ein simples Daten-Objekt ist). Das LoginVO könnte so aussehen:

```
package com.me.myapp.model.vo
{
	// Map this AS3 VO to the following remote class
	[RemoteClass(alias="com.me.myapp.model.vo.LoginVO")]

	[Bindable]
	public class LoginVO
	{
		public var username: String;
		public var password: String;
		public var authToken: String; // set by the server if credentials are valid
	}
}
```

Der 'LoginProxy' stellt Methoden für das Setzen der Login-Daten, für das Ein- und Ausloggen sowie für das Abrufen eines Authentifikationkürzel, welches später bei serverseitigen Abfragen immer benötigt wird, um den Login-Status zu überprüfen.

Proxies

Interaktion mit Remote Proxies

LoginProxy:

```
package com.me.myapp.model
{
    import mx.rpc.events.FaultEvent;
    import mx.rpc.events.ResultEvent;
    import mx.rpc.remoting.RemoteObject;
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.proxy.Proxy;
    import com.me.myapp.model.vo.LoginVO;

    // A proxy to log the user in
    public class LoginProxy extends Proxy implements IProxy {
        public static const NAME:String          = 'LoginProxy';
        public static const LOGIN_SUCCESS:String = 'loginSuccess';
        public static const LOGIN_FAILED:String  = 'loginFailed';
        public static const LOGGED_OUT:String    = 'loggedOut';
        private var loginService: RemoteObject;

        public function LoginProxy () {
            super( NAME, new LoginVO ( ) );
            loginService = new RemoteObject();
            loginService.source = "LoginService";
            loginService.destination = "GenericDestination";
            loginService.addEventListener( FaultEvent.FAULT, onFault );
            loginService.login.addEventListener( ResultEvent.RESULT, onResult );
        }

        // Cast data object with implicit getter
        public function get loginVO( ) : LoginVO {
            return data as LoginVO;
        }

        // The user is logged in if the login VO contains an auth token
        public function get loggedIn():Boolean {
            return ( authToken != null );
        }

        // Subsequent calls to services after login must include the auth token
        public function get authToken():String {
            return loginVO.authToken;
        }
```

Proxies

Interaktion mit Remote Proxies

```
                // Set the users credentials and log in, or log out and try again
                public login( tryLogin:LoginVO ) : void {
                        if ( ! loggedIn )    {
                                loginVO.username= tryLogin.username;
                                loginVO.password = tryLogin.password;
                        } else {
                                logout();
                                login( tryLogin );
                        }
                }

                // To log out, simply clear the LoginVO
                public function logout( ) : void
                {
                        if ( loggedIn ) loginVO = new LoginVO( );
                        sendNotification( LOGGED_OUT );
                }

                // Notify the system of a login success
                private function onResult( event:ResultEvent ) : void
                {
                        setData( event.result ); // immediately available as loginVO
                        sendNotification( LOGIN_SUCCESS, authToken );
                }

                // Notify the system of a login fault
                private function onFault( event:FaultEvent) : void
                {
                        sendNotification( LOGIN_FAILED,  event.fault.faultString );
                }
        }
}
```

Ein 'LoginCommand' kann den 'LoginProxy' abfragen, die Login-Daten setzen und die 'login' Methode aufrufen, welche den Service startet. Ein 'GetPrefsCommand' kann auf die LOGIN_SUCCESS Notification reagieren, um den 'authToken', welcher sich im 'body' der Notifications befindet, der 'getPrefs' Methode des 'PrefsProxy' zu übergeben. Diese Methode ruft wiederum einen Service auf, um die Präferenzen des Users zu erhalten.

Proxies

Interaktion mit Remote Proxies

LoginCommand:

```
package com.me.myapp.controller {
	import org.puremvc.as3.interfaces.*;
	import org.puremvc.as3.patterns.command.*;
	import com.me.myapp.model.LoginProxy;
	import com.me.myapp.model.vo.LoginVO;

	public class LoginCommand extends SimpleCommand {
		override public function execute( note: INotification ) : void  {
			var loginVO : LoginVO = note.getBody() as LoginVO;
			var loginProxy: LoginProxy;
			loginProxy = facade.retrieveProxy( LoginProxy.NAME ) as LoginProxy;
			loginProxy.login( loginVO );
		}
	}
}
```

GetPrefsCommand:

```
package com.me.myapp.controller {
	import org.puremvc.as3.interfaces.*;
	import org.puremvc.as3.patterns.command.*;
	import com.me.myapp.model.LoginProxy;
	import com.me.myapp.model.vo.LoginVO;

	public class GetPrefsCommand extends SimpleCommand {
		override public function execute( note: INotification ) : void  {
			var authToken : String = note.getBody() as String;
			var prefsProxy : PrefsProxy;
			prefsProxy = facade.retrieveProxy( PrefsProxy.NAME ) as PrefsProxy;
			prefsProxy.getPrefs( authToken );
		}
	}
}
```