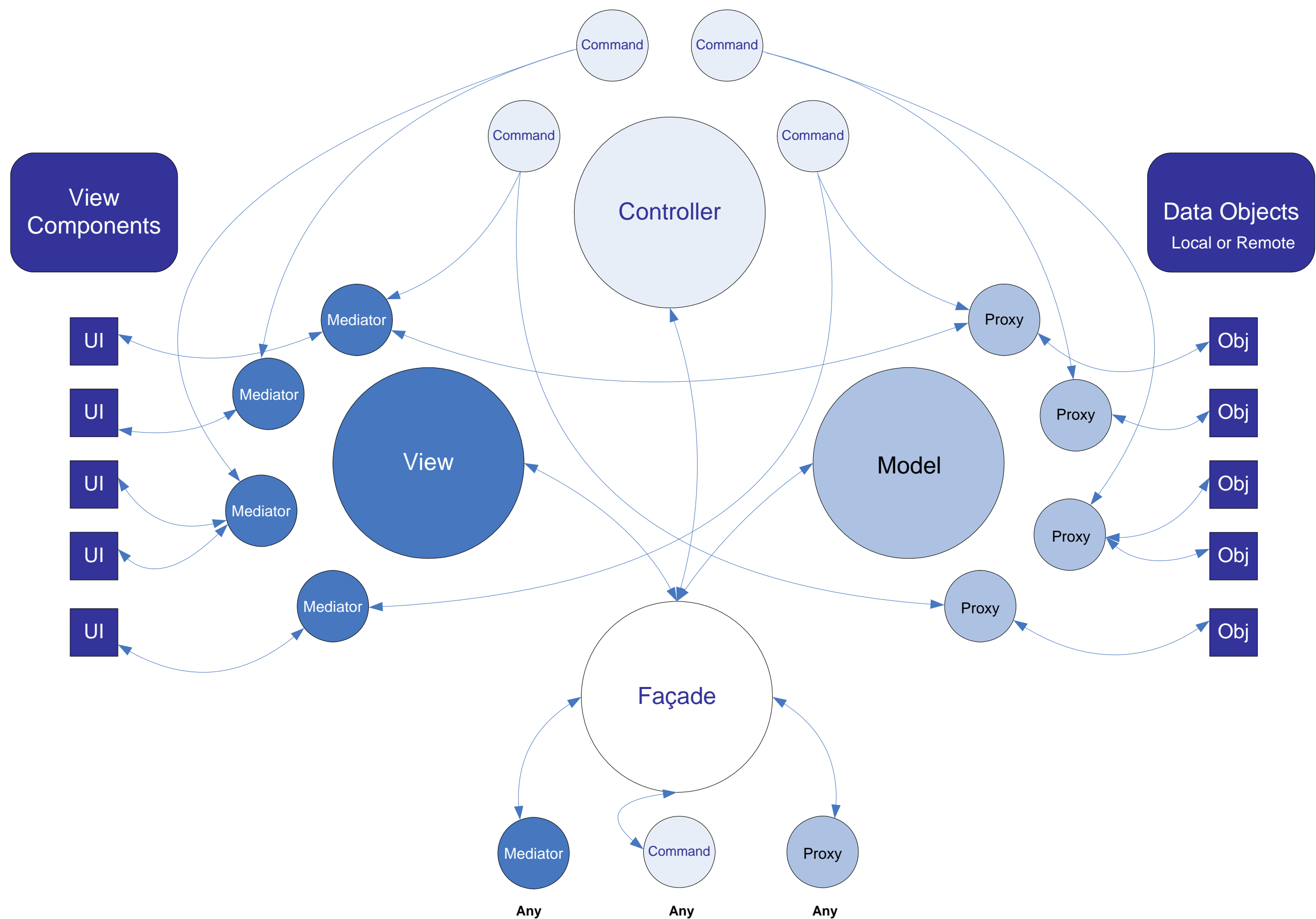
pure mvc

Implementation Idioms and Best Practices

Building Robust, Scalable and Maintainable Client Applications using PureMVC
with Examples in ActionScript 3 and MXML



AUTHOR: Cliff Hall <cliff@puremvc.org>
LAST MODIFIED: 3/02/2008

PureMVC Gestalt 4

- Model & Proxies 4
- View & Mediators 4
- Controller & Commands 5
- Façade & Core 5
- Observers & Notifications 5
- Notifications Can Be Used to Trigger Command Execution 6
- Mediators Send, Declare Interest In, and Receive Notifications 6
- Proxies Send, But Do Not Receive Notifications 6

Façade 7

- What is a Concrete Façade? 8
- Creating a concrete Façade for your Application 8
- Initializing your concrete Façade 11

Notifications 13

- Events vs. Notifications 13
- Defining Event and Notification Constants 14

Commands 16

- Use of Macro and Simple Commands 16
- Loosely-coupling Commands to Mediators and Proxies 17
- Orchestration of Complex Actions and Business Logic 17

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

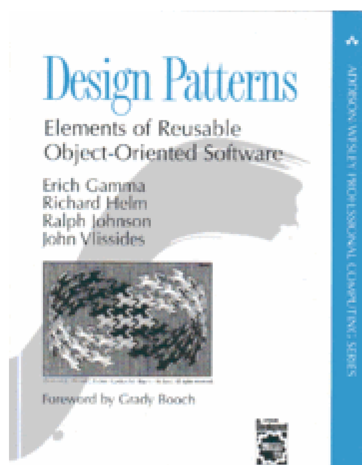
Mediators 23

- Responsibilities of the Concrete Mediator 23
- Casting the View Component Implicitly 24
- Listening and Responding to the View Component 25
- Handling Notifications in the Concrete Mediator 27
- Coupling of Mediators to Proxies and other Mediators 30
- User Interaction with View Components and Mediators 32

Proxies 36

- Responsibilities of the Concrete Proxy 36
- Casting the Data Object Implicitly 37
- Prevent Coupling to Mediators 39
- Encapsulate Domain Logic 40
- Interacting with Remote Proxies 41

Inspiration



PureMVC is a pattern-based framework originally driven by the currently relevant need to design high-performance RIA clients. It has now been ported to other languages and platforms including server environments. This document focuses on the client-side.

While the interpretation and implementations are specific to each platform supported by PureMVC, the patterns employed are well defined in the infamous 'Gang of Four' book: **Design Patterns: Elements of Reusable Object-Oriented Software** (ISBN 0-201-63361-2)

Highly recommended.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

PureMVC Gestalt

The PureMVC framework has a very narrow goal. That is to help you separate your application's coding interests into three discrete tiers; Model, View and Controller.

This separation of interests, and the *tightness* and *direction* of the couplings used to make them work together is of paramount importance in the building of scalable and maintainable applications.

In this implementation of the classic MVC Design *meta*-pattern, these three tiers of the application are governed by three Singletons (a class where only one instance may be created) called simply Model, View and Controller. Together, they are referred to as the 'Core actors'.

A fourth Singleton, the Façade simplifies development by providing a single interface for communication with the Core actors.

Model & Proxies

The Model simply caches named references to Proxies. Proxy code manipulates the data model, communicating with remote services if need be to persist or retrieve it.

This results in portable Model tier code.

View & Mediators

The View primarily caches named references to Mediators. Mediator code stewards View Components, adding event listeners, sending and receiving notifications to and from the rest of the system on their behalf and directly manipulating their state.

This separates the View definition from the logic that controls it.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

PureMVC Gestalt

Controller & Commands

The Controller maintains named mappings to Command classes, which are stateless, and only created when needed.

Commands may retrieve and interact with Proxies, send Notifications, execute other Commands, and are often used to orchestrate complex or system-wide activities such as application startup and shutdown. They are the home of your application's Business Logic.

Façade & Core

The Façade, another Singleton, initializes the Core actors (Model, View and Controller), and provides a single place to access all of their public methods.

By extending the Façade, your application gets all the benefits of Core actors without having to import and work with them directly. You will implement a concrete Façade for your application only once and it is simply done.

Proxies, Mediators and Commands may then use your application's concrete Façade in order to access and communicate with each other.

Observers & Notifications

PureMVC applications may run in environments without access to Flash's Event and EventDispatcher classes, so the framework implements an Observer notification scheme for communication between the Core MVC actors and other parts of the system in a loosely-coupled way.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

PureMVC Gestalt

Observers & Notifications

You need not be concerned about the details of the PureMVC Observer/Notification implementation; it is internal to the framework. You will use a simple method to send Notifications from Proxies, Mediators, Commands and the Façade itself that doesn't even require you to create a Notification instance.

Notifications Can Be Used to Trigger Command Execution

Commands are mapped to Notification names in your concrete Façade, and are automatically executed by the Controller when their mapped Notifications are sent. Commands typically orchestrate complex interaction between the interests of the View and Model while knowing as little about each as possible.

Mediators Send, Declare Interest In and Receive Notifications

When they are registered with the View, Mediators are interrogated as to their Notification interests by having their listNotifications method called, and they must return an array of Notification names they are interested in.

Later, when a Notification by the same name is sent by any actor in the system, interested Mediators will be notified by having their handleNotification method called and being passed a reference to the Notification.

Proxies Send, But Do Not Receive Notifications

Proxies may send Notifications for various reasons, such as a remote service Proxy alerting the system that it has received a result or a Proxy whose data has been updated sending a change Notification.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

PureMVC Gestalt

Proxies Send, but Do Not Receive Notifications

For a Proxy to *listen* for Notifications is to couple it too tightly to the View and Controller tiers.

Those tiers must necessarily listen to Notifications from Proxies, as their function is to visually represent and allow the user to interact with the data Model held by the Proxies.

However View and Controller tiers should be able to vary without affecting the data Model tier.

For instance, an administration application and a related user application might share the same Model tier classes. If only the use cases differ they can be carried out by different View/Controller arrangements operating against the same Model.

Façade

The three Core actors of the MVC meta-pattern are represented in PureMVC by the Model, View and Controller classes. To simplify the process of application development, PureMVC employs the Facade pattern.

The Facade brokers your requests to the Model, View and Controller, so that your code does not need import those classes and you do not need to work with them individually. The Façade class automatically instantiates the Core MVC Singletons in its constructor.

Typically, the framework Facade will be sub-classed in your application and used to initialize the Controller with Command mappings. Preparation of the Model and View are then orchestrated by Commands executed by the Controller.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Façade

What is a Concrete Façade?

Though the Core actors are complete, usable implementations, the Façade provides an implementation that should be considered *abstract*, in that you never instantiate it directly.

Instead, you subclass the framework Façade and add or override some of its methods to make it useful in your application.

This *concrete* Facade is then used to access and notify the Commands, Mediators and Proxies that do the actual work of the system. By convention, it is named 'ApplicationFacade', but you may call it whatever you like.

Generally, your application's View hierarchy (display components) will be created by whatever process your platform normally employs. In Flex, an MXML application instantiates all its children or a Flash movie creates all the objects on its Stage. Once the application's View hierarchy has been built, the PureMVC apparatus is started and the Model and View regions are prepared for use.

Your concrete Façade is also used to facilitate the startup process in a way that keeps the main application code from knowing much about the PureMVC apparatus to which it will be connected. The application merely passes a reference to itself to a 'startup' method on your concrete Façade's Singleton instance.

Creating a Concrete Façade for Your Application

Your concrete Façade doesn't need to do much to provide your application with a lot of power. Consider the following implementation:

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Façade

Creating a Concrete Façade for Your Application

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 );
		}
	}
}
```

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Façade

Creating a Concrete Façade for Your Application

There are a few things to note about the preceding code:

- It extends the PureMVC Façade class, which in turn implements the IFacade interface.

- It does not override the constructor. If it did, it would call the super class constructor before doing anything.

- It defines a static getInstance method that returns the Singleton instance, creating and caching it if need be. The reference to the instance is kept in a protected property of the super class (Façade) and must be cast to the subclass type before it is returned.

- It defines constants for Notification names. Since it is the actor all others in the system use to access and communicate with each other, the concrete Façade is the perfect place to define the constant names that are shared between notification participants.

- It initializes the Controller with Commands that will be executed when corresponding Notifications are sent.

- It provides a startup method which takes an argument (in this case) of type MyApp, which it passes by Notification to the StartupCommand (registered to the notification name STARTUP).

With these simple implementation requirements, your concrete Façade will inherit quite a bit of functionality from the abstract super class.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Façade

Initializing your Concrete Façade

The PureMVC Façade's constructor calls protected methods for initializing the Model, View and Controller instances, and caching them for reference.

By composition then, the Façade implements and exposes the features of the Model, View and Controller; aggregating their functionality and shielding the developer from direct interaction with the Core actors of the framework.

So, where and how does the Façade fit into the scheme of things in an actual application? Consider the following Flex Application:

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>
```

That's it. Pretty simple.

Build the initial view hierarchy, get the ApplicationFaçade instance and invoke its startup method.

NOTE: In AIR, we would use 'applicationComplete', and in Flash we might instantiate the Façade and make the startup call on Frame 1 or in a separate Document Class.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Façade

Initializing your Concrete Façade

Key things to notice about this example are:

- We build the interface in the usual, declarative MXML way; beginning with an <mx:Application> tag, containing components and containers, either custom or stock.

- A script block is used to declare and initialize a private variable with the Singleton instance of the concrete ApplicationFacade.

- Since we are initializing the variable with a call to the static ApplicationFacade.getInstance method, this means that by the time the Application's creationComplete event fires, the Façade will have been created and along with it, the Model, View and Controller, though no Mediators or Proxies will have been created yet.

- In the creationComplete handler of the Application tag, we invoke the startup method, passing a reference to the main application.

Note that ordinary View Components have no need to know or interact with the Façade, but the top-level Application is the exception to the rule.

The top-level Application (or Flash Movie) builds the View hierarchy, initializes the Facade, then starts up the PureMVC apparatus.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Notifications

PureMVC implements the Observer pattern so that the Core actors and their collaborators can communicate in a loosely-coupled way, and without platform dependency.

The ActionScript language does not provide the Events model that is used in Flex and Flash, those come from the Flash package. The framework has been ported to other platforms such as C# and J2ME, because the framework manages its own internal communications rather than relying on those provided by the Flash platform.

Not simply a replacement for Events, Notifications operate in a fundamentally different way, and work synergistically with Events to produce extremely reusable View Components that need not even know that they are coupled to a PureMVC system at all if engineered properly.

Events vs. Notifications

Events are dispatched from Flash display objects that implement the IEventDispatcher interface. The Event is 'bubbled' up the display hierarchy, allowing the parent object to handle the Event, or the parent's parent, etc.

This is a *chain of responsibility mechanism* whereby only those in the parent/child lineage have the opportunity to receive and act upon the Event unless they have a reference to the dispatcher and can set a listener directly upon it.

Notifications are sent by the Façade and Proxies; listened for and sent by Mediators; mapped to and sent by Commands. It is a *publish/subscribe mechanism* whereby many Observers may receive and act upon the same Notification.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Notifications

Events vs. Notifications

Notifications may have an optional 'body', which can be any ActionScript object.

Unlike Flash Events, it is rarely necessary to create a custom Notification class, since it can carry a payload 'out of the box'. You can of course create custom Notification classes in order to strongly type interactions with them, but the limited real-world benefits of the compile-time checking (for Notifications specifically) weighed against the overhead of maintaining many Notification classes reduce it to a question of programming style.

Notifications also have an optional 'type' that can be used by the Notification recipient as a discriminator.

For instance, in a document editor application, there may be a Proxy instance for each document that is opened and a corresponding Mediator for the View Component used to edit the document. The Proxy and Mediator might share a unique key that the Proxy passes as the Notification type.

All the Mediator instances registered for that Proxy's Notifications will be notified, but will use the type property to determine if they should act upon it or not.

Defining Notification and Event Constants

We have seen that the concrete Façade is a good place to define common, Notification constants. Being the central mechanism for interaction with the system, all notification participants will by default be collaborators with Façade.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Notifications

Defining Notification and Event Constants

A separate 'ApplicationConstants' class is sometimes used for this purposed instead of the concrete Façade if the constant names alone need to be known to another application.

In any case, centralized definition of constants for Notification names ensures that when one of the notification participants needs to refer to a Notification name, we can do so in a type-safe way that the compiler can check, as opposed to using literal strings which could be misspelled, but not cause an error.

Do not, however, define the names of Events on the concrete Façade. *Define Event name constants statically on the boundary classes that generate them, or in custom Event classes that are dispatched.*

Representing the physical boundaries of the Application, View Components and Data Objects may remain reusable if they communicate to their associated Mediator or Proxy by dispatching Events instead of making method calls or sending Notifications.

If a View Component or Data Object dispatches an Event that the stewarding Mediator or Proxy is listening for, then it is likely that only that collaboration pair need ever know the particular Event name. Further communication between the listener and the rest of the PureMVC system may occur through the use of Notifications.

Though the relationships of these collaboration pairs (Mediator/View Component & Proxy/Data Object) are necessarily somewhat close, they are loosely-coupled to the rest of the application architecture; affording more contained refactoring of the data model or user interface when required.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Commands

The concrete Façade generally initializes the Controller with the set of Notification to Command mappings needed at startup.

For each mapping, the Controller registers itself as an Observer for the given Notification. When notified, the Controller instantiates the appropriate Command. Finally, the Controller calls the Command's execute method, passing in the Notification.

Commands are stateless; they are created when needed and are intended to go away when they have been executed. For this reason, it is important not to instantiate or store references to Commands in long-living objects.

Use of Macro and Simple Commands

Commands, like all PureMVC framework classes, implement an interface, namely ICommand. PureMVC includes two ICommand implementations that you may easily extend.

The SimpleCommand class merely has an execute method which accepts an INotification instance. Insert your code in the execute method and that's it.

The MacroCommand class allows you to execute multiple sub-commands sequentially, each being created and passed a reference to the original Notification.

MacroCommand calls its initializeMacroCommand method from within its constructor. You override this method in your subclasses to call the addSubCommand method once for each Command to be added. You may add any combination of SimpleCommands or MacroCommands.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Commands

Loosely Coupling Commands to Mediators and Proxies

Commands are executed by the Controller as a result of Notifications being sent. Commands should never be instantiated and executed by any other actor than the Controller.

To communicate and interact with other parts of the system, Commands may:

- Register, remove or check for the existing registration of Mediators, Proxies, and Commands.
- Send Notifications to be responded to by other Commands or Mediators.
- Retrieve and Proxies and Mediators and manipulate them directly.

Commands allow us to easily trigger the elements of the View into the appropriate states, or transport data to various parts of it.

They can be used to perform transactional interactions with the Model that span multiple Proxies, and require Notifications to be sent when the whole transaction completes, or to handle exceptions and take action on failure.

Orchestration of Complex Actions and Business Logic

With several places in the application that you might place code (Commands, Mediators and Proxies); the question will inevitably and repeatedly come up:

What code goes where? What, exactly, should a Command *do*?

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Commands

Orchestration of Complex Actions and Business Logic

The first distinction to make about the logic in your application is that of Business Logic and Domain Logic.

Commands house the *Business Logic* of our application; the technical implementation of the use cases our application is expected to carry out against the *Domain Model*. This involves coordination of the Model and View states.

The Model maintains its integrity through the use of Proxies, which house *Domain Logic*, and expose an API for manipulation of Data Objects. They encapsulate all access to the data model whether it is in the client or the server, so that to the rest of the application all that is relevant is whether the data can be accessed synchronously or asynchronously.

Commands may be used orchestrate complex system behaviors that must happen in a specific order, and where it is possible that the results of one action might feed the next.

Mediators and Proxies should expose a *course-grained* interface to Commands (and each other), that hides the implementation of their stewarded View Component or Data Object.

Note that when we talk about a View Component we mean a button or widget the user interacts with directly. When we speak about Data Objects that includes arbitrary structures that hold data as well as the remote services we may use to retrieve or store them.

Commands interact with Mediators and Proxies, but should be insulated from boundary implementations. Consider the following Commands used to prepare the system for use:

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Commands

Orchestration of Complex Actions and Business Logic

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 );
        }
    }
}
```

This is a MacroCommand that adds two sub-commands, which are executed in FIFO order when the MacroCommand is executed.

This provides a top level 'queue' of actions to be completed at startup. But what should we do exactly, and in what order?

Before the user can be presented or interact with any of the application's data, the Model must be placed in a consistent, known state. Once this has been achieved, the View can be prepared to present the Model's data and allow the user to manipulate and interact with it.

Therefore, the startup process usually consists of two broad sets of activities – preparation of the Model, followed by preparation of the View.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Commands

Orchestration of Complex Actions and Business Logic

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() );
        }
    }
}
```

Preparing the Model is usually a simple matter of creating and registering all the Proxies the system will need at startup.

The ModelPrepCommand above is a SimpleCommand that prepares the Model for use. It is the first of the previous MacroCommand's sub-commands, and so is executed first.

Via the concrete Façade, it creates and registers the various Proxy classes that the system will use at startup. Note that the Command does not do any manipulation or initialization of the Model data. The Proxy is responsible for any data retrieval, creation or initialization necessary to prepare its Data Object for use by the system.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Commands

Orchestration of Complex Actions and Business Logic

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 ) );
        }
    }
}
```

This is a SimpleCommand that prepares the View for use. It is the last of the previous MacroCommand's subcommands, and so, is executed last.

Notice that the only Mediator it creates and registers is the ApplicationMediator, which stewards the Application View Component.

Further, it passes the body of the Notification into the constructor of the Mediator. This is a reference to the Application, sent by the Application itself as the Notification body when the original STARTUP Notification was sent. (Refer to the previous MyApp example.)

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Commands

Orchestration of Complex Actions and Business Logic

The Application is a somewhat special View Component in that it instantiates and has as children all the other View Components that are initialized at startup time.

To communicate with the rest of the system, the View Components need to have Mediators. And creating those Mediators requires a reference to the View Components they will mediate, which only the Application knows at this point.

The Application's Mediator is the only class we're allowing to know anything about the Application's implementation, so we handle the creation of the remaining Mediators inside the constructor of the Application's Mediator.

So, with the above three Commands, we have orchestrated an ordered initialization of the Model and the View. In doing so, the Commands did not need to know very much about the Model or the View.

When the details of the Model or the implementation of the View change, the Proxies and Mediators are refactored as needed.

Business Logic in the Commands should be insulated from refactoring that takes place at the application's boundaries.

The Model should encapsulate 'domain logic', maintaining the integrity of the data in the Proxies. Commands carry out 'transactional' or 'business' logic against the Model, encapsulating the coordination of multi-Proxy transactions or handling and reporting exceptions in the fashion called for by the application.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

A Mediator class is used to mediate the user's interaction with one or more of the application's View Components (such as Flex DataGrids or Flash MovieClips) and the rest of the PureMVC application.

In a Flash-based application, a Mediator typically places event listeners on its View Component to handle user gestures and requests for data from the Component. It sends and receives Notifications to communicate with the rest of the application.

Responsibilities of the Concrete Mediator

The Flash, Flex and AIR frameworks provide a vast array of richly-interactive UI components. You may extend these or write your own in ActionScript to provide endless possibilities for presenting the data model to the user and allowing them to interact with it.

In the not so distant future, there will be other platforms running ActionScript. And the framework has been ported and demonstrated on other platforms already including Silverlight and J2ME, further widening the horizons for RIA development with this technology.

A goal of the PureMVC framework is to be neutral to the technologies being used at the boundaries of the application and provide simple idioms for adapting whatever UI component or Data structure/service you might find yourself concerned with at the moment.

To the PureMVC-based application, a View Component is any UI component, regardless of what framework it is provided by or how many sub-components it may contain. A View Component should encapsulate as much of its own state and operation as possible, exposing a simple API of events, methods and properties.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

Responsibilities of the Concrete Mediator

A concrete Mediator helps us adapt one or more View Components to the application by holding the only references to those components and interacting with the API they expose.

The responsibilities for the Mediator are primarily handling Events dispatched from the View Component and relevant Notifications sent from the rest of the system.

Since Mediators will also frequently interact with Proxies, it is common for a Mediator to retrieve and maintain a local reference to frequently accessed Proxies in its constructor. This reduces repetitive retrieveProxy calls to obtain the same reference.

Casting the View Component Implicitly

The base Mediator implementation that comes with PureMVC accepts a name and a generic Object as its sole constructor arguments.

Your concrete Mediator's constructor will pass its View Component to the superclass and it will be made immediately available internally as a protected property called viewComponent, generically typed as Object.

You may also dynamically set a Mediator's View Component after it has been constructed by calling its setViewComponent method.

However it was set, you will frequently need to cast this Object to its actual type, in order to access whatever API it exposes, which can be a cumbersome and repetitive practice.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

Casting the View Component Implicitly

ActionScript provides a language feature called implicit getters and setters. An implicit getter looks like a method, but is exposed as a property to the rest of the class or application. This turns out to be very helpful in solving the frequent casting problem.

A useful idiom to employ in your concrete Mediator is an implicit getter that casts the View Component to its actual type and gives it a meaningful name.

By creating a method like this:

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

Then, elsewhere in our Mediator, *rather* than doing this:

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

We can *instead* do this:

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

Listening and Responding to the View Component

A Mediator usually has only one View Component, but it might manage several, such as an ApplicationToolBar and its contained buttons or controls. We can contain a group of related controls (such as a form) in a single View Component and expose the children to the Mediator as properties. But it is best to encapsulate as much of the component's implementation as possible. Having the component exchange data with a custom typed Object is better.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

Listening and Responding to the View Component

The Mediator will handle the interaction with the Controller and Model tiers, updating the View Component when relevant Notifications are received.

On the Flash platform, we typically set Event listeners on the View Component when the Mediator is constructed or has its setViewComponent method called, specifying a handler method:

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

What the Mediator does in response to that Event is, of course governed entirely by the requirements of the moment.

Generally, a concrete Mediator's Event handling methods will perform some combination of these actions:

- Inspect the Event for type or custom content if expected.
- Inspect or modify exposed properties (or call exposed methods) of the View Component.
- Inspect or modify exposed properties (or call exposed methods) of a Proxy.
- Send one or more Notifications that will be responded to by other Mediators or Commands (or possibly even the same Mediator instance).

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

Listening and Responding to the View Component

Some good rules of thumb are:

- If a number of other Mediators must be involved in the overall response to an Event, then update a common Proxy or send a Notification, which is responded to by interested Mediators, all of whom respond appropriately.

- If a great amount of coordinated interaction with other Mediators is required, a good practice is to use a Command to encode the steps in one place.

- Consider it a bad practice to retrieve other Mediators and act upon them, or to design Mediators to expose such a manipulation methods.

- To manipulate and distribute application state information to the Mediators, set values or call methods on Proxies created to maintain state. Let the Mediators be interested in the Notifications sent by the state-keeping Proxies.

Handling Notifications in the Concrete Mediator

In contrast to the explicit adding of event listeners to View Components, the process of coupling the Mediator to the PureMVC system is a simple and automatic one.

Upon registration with the View, the Mediator is interrogated as to its notification interests. It responds with an array of the Notification names it wishes to handle.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

Handling Notifications in the Concrete Mediator

The simplest way to respond is with a single expression that creates and returns an anonymous array, populated by the Notification names, which should be defined as static constants, usually in the concrete Façade.

Defining the Mediator's list of Notification interests is easy:

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

When one of the Notifications named in the Mediator's response is sent by any actor in the system (including the Mediator itself), the Mediator's handleNotification method will be called, and the Notification passed in.

Because of its readability, and the ease of which one may refactor to add or remove Notifications handled, the 'switch / case' construct is preferred over the 'if / else if' expression style inside the handleNotifications method.

Essentially, there should be little to do in response to any given Notification, and all the information needed should be in the Notification itself. Occasionally some information may be retrieved from a Proxy based on information inside the Notification, but there should be no complicated logic in the Notification handler. If there is, then it is a sign that you are trying to put the Business Logic that belongs in a Command into the Notification handler of your Mediator.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

Handling Notifications in the Concrete Mediator

```
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;

            }
    }
}
```

Also, a typical Mediator's handleNotification method handles no more than 4 or 5 Notifications.

More than that is a sign that the Mediator's responsibilities should be divided more granularly. Create Mediators for sub-components of the ViewComponent rather than try to handle them all in one monolithic Mediator.

The use of a single, predetermined notification method is the key difference between the way a Mediator listens for Events and how it listens for Notifications.

With Events, we have a number of handler methods, usually one for each Event the Mediator handles. Generally these methods just send Notifications. They should not be complicated, nor should they be micro-managing many View Component details, since the View Component should be written to encapsulate implementation details, exposing a course-grained API to the Mediator.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

Handling Notifications in the Concrete Mediator

With Notifications, you have a single handler method, inside which you handle all Notifications the Mediator is interested in.

It is best to contain the responses to Notifications entirely inside the handleNotification method, allowing the switch statement to separate them clearly by Notification name.

There has been much debate on the usage of 'switch/case' as many developers consider it to limiting since all cases execute within the same scoped method. However, the single Notification method and 'switch/case' style was specifically chosen to limit what is done inside the Mediator, and remains the recommended construct.

The Mediator is meant to mediate communications between the View Component and the rest of the system.

Consider the role of a translator mediating the exchange of conversation between her Ambassador and the rest of the members at a UN conference. She should rarely be doing more than simple translation and forwarding of messages, occasionally reaching for an appropriate metaphor or fact. The same is true of the Mediator's role within PureMVC.

Coupling of Mediators to Proxies and other Mediators

Since the View is ultimately charged with representing the data model in a graphical, interactive way, a relatively tight one-way coupling to the application's Proxies is to be expected. The View must know about the Model, but the Model has no need to know any aspect of the View.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

Coupling of Mediators to Proxies and other Mediators

Mediators may freely retrieve Proxies from the Model, and read or manipulate the Data Object via any API exposed by the Proxy. However, doing the work in a Command will loosen the coupling between the View and the Model.

In the same fashion, Mediators *could* retrieve references to other Mediators from the View and read or manipulate them in any way exposed by the retrieved Mediator.

However this is not a good practice, since it leads to dependencies between parts of the View, which negatively impacts the ability to refactor one part of the View without affecting another.

A Mediator that wishes to communicate with another part of the View should send a Notification rather than retrieving another Mediator and manipulating it directly.

Mediators should not expose methods for manipulating their stewarded View Component(s), but should instead respond only to Notifications to carry out such work.

If much manipulation of a View Component's internals is being done in a Mediator (in response to an Event or Notification), refactor that work into a method on the View Component, encapsulating its implementation as much as possible, yielding greater reusability.

If much manipulation of Proxies or their data is being done in a Mediator, refactor that work into a Command, simplifying the Mediator, moving Business Logic into Commands where it may be reused by other parts of the View, and loosening the coupling of the View to the Model to the greatest extent possible.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

User Interaction with View Components and Mediators

Consider a LoginPanel component presenting a form. There is a LoginPanelMediator which allows the user to communicate their credentials and intention to log in by interacting with the LoginPanel and responding to their input by initiating a login attempt.

The collaboration between the LoginPanel component and the LoginPanelMediator is that the component sends a TRY_LOGIN Event when the user has entered their credentials and wants to try logging in. The LoginPanelMediator handles the Event by sending a Notification with component's populated LoginVO as the body.

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>
```

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

User Interaction with View Components and Mediators

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

    <!—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>
```

The LoginPanel View Component populates a new LoginVO with the user's form input and when they click the 'Login' button, an Event is dispatched. Then the LoginPanelMediator will take over.

This leaves the View Component with the simple role of gathering the data and alerting the system when it is ready.

A more complete component would only enable the login button when both the username and password fields have content preventing a bad login attempt before it is ever made.

The View Component hides its internal implementation, its entire API used by the Mediator consisting of a TRY_LOGIN Event, a LoginVO property to be checked, and the Panel status property.

The LoginPanelMediator will also respond to LOGIN_FAILED and LOGIN_SUCCESS Notifications and set the LoginPanel status.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

User Interaction with View Components and 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;
```

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Mediators

User Interaction with View Components and Mediators

```
                        case LoginProxy.LOGIN_SUCCESS:
                            loginPanel.loginStatus = LoginPanel.LOGGED_IN;
                            break;
                }
            }

            // 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;
            }
    }
}
```

Note that the LoginPanelMediator places an Event listener on the LoginPanel in its constructor, so that the onTryLogin method will be invoked when the user has clicked the Login button. In the onTryLogin method, the LOGIN Notification is sent, bearing the user populated LoginVO.

Earlier, we registered the LoginCommand to this Notification. It will invoke the LoginProxy's login method, passing the LoginVO. The LoginProxy will attempt the login with the remote service, and send a LOGIN_SUCCESS or LOGIN_FAILED Notification. These classes are defined at the end of the section on Proxies.

The LoginPanelMediator lists LOGIN_SUCCESS and LOGIN_FAILED as its Notification interests, and so regardless of the outcome, it will be notified, and will set the LoginPanel's loginStatus to LOGGED_IN on success; and to NOT_LOGGED_IN on failure, clearing the LoginVO.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Generally speaking, the Proxy pattern is used to provide a placeholder for an object in order to control access to it. In a PureMVC-based application, the Proxy class is used specifically to manage a portion of the application's data model.

A Proxy might manage access to a locally created data structure of arbitrary complexity. This is the Proxy's Data Object.

In this case, idioms for interacting with it probably involve synchronous setting and getting of its data. It may expose all or part of its Data Object's properties and methods, or a reference to the Data Object itself. When exposing methods for updating the data, it may also send Notifications to the rest of the system that the data has changed.

A Remote Proxy might be used to encapsulate interaction with a remote service to save or retrieve a piece of data. The Proxy can maintain the object that communicates with the remote service, and control access to the data sent and received from the service.

In such a case, one might set data or call a method of the Proxy and await an asynchronous Notification, sent by the Proxy when the service has received the data from the remote endpoint.

Responsibilities of the Concrete Proxy

The concrete Proxy allows us to encapsulate a piece of the data model, wherever it comes from and whatever its type, by managing the Data Object and the application's access to it.

The Proxy implementation class that comes with PureMVC is a simple data carrier object that can be registered with the Model.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Responsibilities of the Concrete Proxy

Though it is completely usable in this form, you will usually subclass Proxy and add functionality specific to the particular Proxy.

Common variations on the Proxy pattern include:

- *Remote Proxy*, where the data managed by the concrete Proxy is in a remote location and will be accessed via a service of some sort.

- *Proxy and Delegate*, where access to a service object needs to be shared between multiple Proxies. The Delegate class maintains the service object and controls access to it, ensuring that responses are properly routed to their requestors.

- *Protection Proxy*, used when objects need to have different access rights.

- *Virtual Proxy*, which creates expensive objects on demand.

- *Smart Proxy*, loads data object into memory on first access, performs reference counting, allows locking of object to ensure no other object can change it.

Casting the Data Object Implicitly

The base Proxy implementation that comes with PureMVC accepts a name and a generic Object as constructor arguments. You may dynamically set a Proxy's Data Object after it is constructed by calling its setData method.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Casting the Data Object Implicitly

As with the Mediator and its View Component, you will need to cast this Object frequently to its actual type, in order to access whatever properties and methods it exposes; at best a cumbersome and repetitive practice, but possibly leading to idioms that expose more of the Data Object's implementation than necessary.

Also, since the Data Object is often a complex structure we frequently need to have handy named references to several parts of the structure in addition to a typecast reference to the structure itself.

Again the ActionScript language feature called implicit getters and setters prove to be very helpful in solving the frequent casting and improper implementation knowledge problem.

A useful idiom to employ in your concrete Proxy is an implicit getter that casts the Data Object to its actual type and gives it a meaningful name.

Additionally, it may define multiple differently-typed getters to retrieve specific parts of the Data Object.

For instance:

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

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

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Casting the Data Object Implicitly

Elsewhere in a Mediator, *rather* than doing this:

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

We can *instead* do this:

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

Prevent Coupling to Mediators

The Proxy is not interrogated as to its Notification interests as is the Mediator, nor is it ever notified, because it should not be concerned with the state of the View. Instead, the Proxy exposes methods and properties to allow the other actors to manipulate it.

The concrete Proxy should not retrieve and manipulate Mediators as a way of informing the system about changes to its Data Object.

It should instead send Notifications that will be responded to by Commands or Mediators. How the system is affected by those Notifications should be of no consequence to the Proxy.

By keeping the Model tier free of any knowledge of the system implementation, the View and Controller tiers may be refactored often without affecting the Model tier.

The obverse is not quite true. It is difficult for the Model tier to change without affecting the View and possibly Controller tiers as a result. After all, those tiers exist only to allow the user to interact with the Model tier.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Encapsulate Domain Logic in Proxies

A change in the Model tier will almost always result in some refactoring of the View/Controller tiers.

We increase the separation between the Model tier and the combined interests of the View and Controller tiers by ensuring that we place as much of the Domain Logic, into the Proxies as possible.

The Proxy may be used not only to control access to data but also to perform operations on the data as may be required to keep that data in a valid state.

For instance, the computation of sales tax is a Domain Logic function that should reside in a Proxy, not a Mediator or Command.

Though it could be performed in any of those places, placing it in a Proxy is not only logical, it keeps the other tiers lighter and easier to refactor.

A Mediator may retrieve the Proxy; call its sales tax function, passing in some form items perhaps. But placing the actual computation in the Mediator would be embedding Domain Logic in the View tier. Sales tax computation is a rule belonging to the Domain Model. The View merely sees it as a property of the Domain Model, available if the appropriate inputs are present.

Imagine that the application you are working on is currently an RIA delivered in the browser for desktop-scale resolutions. A new version is to be delivered for PDA resolution with a reduced use case set, but still with full Model tier requirements in place with today's app.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Encapsulate Domain Logic in Proxies

With the right separation of interests, we may be able to reuse the Model tier in its entirety and simply fit new View and Controller tiers to it.

Though placing the actual computation of sales tax in the Mediator might seem efficient or easy at the moment of implementation; you just took data from a form and you want to compute sales tax and poke it into the model as an order, perhaps.

However on the each version of your app you will now have to duplicate your effort or copy/paste sales tax logic into your new, completely different view tier, rather than have it show up automatically with the inclusion of your Model tier library.

Interacting with Remote Proxies

A Remote Proxy is merely a Proxy that gets its Data Object from some remote location. This usually means that we interact with it in an asynchronous fashion.

How the Proxy gets its data depends on the client platform, the remote service implementation, and the preferences of the developer. In a Flash/Flex environment we might use HTTPService, WebService, RemoteObject, DataService or even XMLSocket to make service requests from within the Proxy.

Depending on its requirements, a Remote Proxy may send requests dynamically, in response to having a property set or a method called; or it might make a single request at construction time and provide get/set access to the data thereafter.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Interacting with Remote Proxies

There are a number of optimizations that may be applied in the Proxy to increase efficiency of the communication with a remote service.

It may be built in such a way as to cache the data, so as to reduce network 'chattiness'; or to send updates to only certain parts of a data structure that have changed, reducing bandwidth consumption.

If a request is dynamically invoked on a Remote Proxy by another actor in the system, the Proxy should send a Notification when the result has returned.

The interested parties to that Notification may or may not be the same that initiated the request.

For example, the process of invoking a search on a remote service and displaying the results might follow these steps:

- A View Component initiates a search by dispatching an Event.

- Its Mediator responds by retrieving the appropriate Remote Proxy and setting a searchCriteria property.

- The searchCriteria property on the Proxy is really an implicit setter, which stores the value and initiates the search via an internal HTTPService to which it is listening for result and fault Events.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Interacting with Remote Proxies

- When the service returns, it dispatches a ResultEvent which the Proxy responds to by setting a public property on itself to the result.

- The Proxy then sends a Notification indicating success, bundling a reference to the data object as the body of the Notification

- Another Mediator has previously expressed interest in that particular Notification and responds by taking the data from the Notification body, and setting it as the dataProvider property of its stewarded View Component.

Or consider a LoginProxy, who holds a LoginVO (a Value Object; a simple data carrier class). The LoginVO might look like this:

```
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
        }
}
```

The LoginProxy exposes methods for setting the credentials, logging in, logging out, and retrieving the authorization token that will be included on service calls subsequent to logging in using this particular authentication scheme.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Interacting with 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;
            }
```

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Interacting with 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 );
		}
	}
}
```

A LoginCommand might retrieve the LoginProxy, set the credentials, and invoke the login method, calling the service.

A GetPrefsCommand might respond to the LOGIN_SUCCESS Notification, retrieve the authToken from the Notification body and make a call to the next service that retrieves the User's preferences.

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.

Proxies

Interacting with 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 );
            }
    }
}
```

PureMVC is a free, open source framework created and maintained by Futurescale, Inc. Copyright © 2006-08, Some rights reserved.
Reuse is governed by the Creative Commons 3.0 Attribution US License. PureMVC, as well as this documentation and any training materials or demonstration source code downloaded from Futurescale's websites is provided 'as is' without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of fitness for a purpose, or the warranty of non-infringement.