Framework Overview

This document discusses the classes and interfaces of the PureMVC framework; illustrating their roles, responsibilities and collaborations with simple UML (Unified Modeling Language) diagrams.

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

In this implementation of the classic MVC design meta-pattern, the application tiers are represented by three Singletons (a class where only one instance may be created).

A fourth Singleton, the Façade, simplifies development by providing a single interface for communications throughout the application.

**The Model** caches named references to **Proxies**, which expose an API for manipulating the **Data Model** (including data retrieved from remote services).

**The View** primarily caches named references to **Mediators**, which adapt and steward the **View Components** that make up the user interface.

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

**The Façade** initializes and caches the Core actors (**Model, View and Controller**), and provides a single place to access all of their public methods.
The **Facade** class makes it possible for the **Proxies**, **Mediators** and **Commands** that make up most of our final application to talk to each other in a loosely coupled way, without having to import or work directly with the Core framework actors.

When we create a concrete **Facade** implementation for our application, we are able to use the Core actors ‘out of the box’, incidental to our interaction with the **Facade**, minimizing the amount of API knowledge the developer needs to have to be successful with the framework.

The Core actors **Model**, **View** and **Controller** implement **IModel**, **IView** and **IController** interfaces respectively. The **Facade** implements **IFacade**, which implements all the Core interfaces, by composition.
View, Mediators and View Components

The View class is implemented as a Singleton that caches and provides access to concrete IMediator instances.

Mediators help us to create or reuse existing user interface components without the need to imbue them with knowledge about the PureMVC application they communicate with. Concrete Mediators must implement the IMediator interface, usually by sub-classing the framework Mediator class.

View Components display data or accept user gestures. In a Flash-based application, they typically communicate with their Mediators using Events and exposing some properties for the concrete Mediators to inspect or manage. A Mediator connects a View Component with its data and communicates with the rest of the system on its behalf.

When a concrete Mediator is registered with the View, it is queried as to its Notification interests. It must return an Array of all the names of the Notifications it wishes to be informed of.

Because it must implement the IMediator interface, the concrete Mediator will have a handleNotification method. When it is registered with the View, an Observer instance is created and registered for each Notification in the Array so that the Mediator’s handleNotification method is invoked whenever a Notification of interest to the Mediator is sent.

The Mediator framework class implements INotifier and so has a sendNotification method, which takes the parameters for a new Notification, constructs the Notification and uses the Singleton IFacade instance to send it.
View, Mediators and View Components

The Mediator's protected façade property is initialized to the registered IFacade instance, and therefore the Mediator must be constructed after you have initialized your Application's concrete Façade.
Model, Proxies and Data Objects

The **Model** class is implemented as a Singleton that caches and provides access to concrete **IProxy** instances.

**Proxies** help us to expose data structures and entity classes (and the domain logic and services that support them) to our application in such a way that they may be easily reused elsewhere, or refactored with a minimum amount of impact to the rest of the application.

We might use a concrete **Proxy** to simply manage a reference to a local data object, in which case idioms for interacting with it might involve synchronous setting and getting of its data.

A **Proxy** might also encapsulate the application's interaction with a remote service to save or retrieve a piece of data, in which case, one might call a method or set data upon the **Proxy** and await a **Notification** sent when the **Proxy** has retrieved the data from the service.

The **Proxy** framework class implements **INotifier** and so has a **sendNotification** method, which takes the parameters for a new **Notification**, constructs the **Notification** and uses the **IFacade** Singleton instance to send it.

Its protected **façade** property is initialized to the registered **IFacade** instance, and therefore the **Proxy** must be constructed **after** you have initialized your Application's concrete **Facade**.
Model, Proxies and Data Objects

### Model
- instance : IModel
- proxyMap : Array
- SINGLETON_MSG : String

#### IModel

### Proxy
- NAME : String
- data : Object
- proxyName : String
- facade : IFacade

#### IProxy

### DataObject

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 Unported 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.
Controller and Commands

The Controller class is implemented as a Singleton that maintains a mapping between Notification names and Command class references.

A Command may retrieve and interact with Proxies, communicate with Mediators, or execute other Commands. Commands are often used to orchestrate complex or system-wide activities such as application startup and shutdown.

When it is initialized (typically by an IFacade implementation), the Controller creates and registers with the View an appropriate Observer instance for each Notification to Command mapping, such that when any of the registered Notifications are broadcast, the Controller’s executeCommand method is called with the Notification.

When Notifications are broadcast by the View, the Controller instantiates the appropriate Command class and calls the execute method, passing in the Notification.

PureMVC includes two ICommand class implementations that you may easily extend. Both implement INotifier, and so have a sendNotification method and a protected façade property, initialized to the Singleton IFaçade instance.

The SimpleCommand class merely has an execute method which is called with the Notification object.

The MacroCommand class allows you to execute multiple 'subcommands' sequentially, each being created and passed the original Notification on its execute method.
Controller and Commands

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

```
Controller
---
-instance : IController
-view : IView
-commandMap : Array
-SINGLETON_MSG : String
+registerCommand(in notificationName : String, in commandClassRef : Class) : void
+executeCommand(in notification : INotification) : void
+removeCommand(in notificationName : String) : void
+Controller()
+getInstance() : IView
#initializeController() : void

IController
---

MacroCommand
---
-subCommands : Array
-facade : IFacade
+execute(in notification : INotification) : void
+sendNotification(in notificationName : String, in body : Object, in type : String) : String
+MacroCommand()
+initializeMacroCommand() : void
+addSubCommand(in commandClassRef : Class) : void

SimpleCommand
---
-facade : IFacade
+execute(in notification : INotification) : void
+sendNotification(in notificationName : String, in body : Object, in type : String) : String
```
View, Observer and Notification

Proxies, Mediators and Commands communicate with each other in a loosely-coupled and platform-neutral way by broadcasting Notifications.

- Proxies may broadcast, but do not listen for Notifications.
- Mediators listen for and may broadcast Notifications.
- Commands are triggered by and may broadcast Notifications.

Since PureMVC applications may also run in a pure ActionScript environment without the underlying `flash.events.Event` and `EventDispatcher` classes, the framework implements an Observer notification scheme for communication between the Core actors and other parts of the system.

PureMVC employs the Observer pattern for this purpose. An `IObserver` instance carries a reference to an object which wishes to be notified (the 'Notify Context'), and a method on that object to call when an `INotification` is broadcast (the 'Notify Method').

The View is responsible for managing the map of Notification names to Observer lists and for notifying all Observers when a Notification is sent.
View, Observer and Notification

**View**

- `instance : IView`
- `mediatorMap : Array`
- `observerMap : Array`
- `SINGLETON_MSG : String`

  + `registerObserver(in notificationName : String, in observer : IObserver) : void`
  + `notifyObservers(in notification : INotification) : void`
  + `registerMediator(in mediator : IMediator) : void`
  + `retrieveMediator(in mediatorName : String) : IMediator`
  + `removeMediator(in mediatorName : String) : void`
  + `hasMediator(in mediatorName : String) : Boolean`
  + `View()`
  + `getInstance() : IView`
  + `initializeView() : void`

**Observer**

- `notify : Function`
- `context : Object`

  + `setNotifyMethod(in notifyMethod : Function) : void`
  + `setNotifyContext(in notifyContext : Object) : void`
  + `getNotifyMethod() : Function`
  + `getNotifyContext() : Object`
  + `notifyObserver(in notification : INotification) : void`
  + `Observer(in method : Function, in context : Object)`

**Notification**

- `name : String`
- `type : String`
- `body : Object`

  + `getName() : String`
  + `setBody(in body : Object) : void`
  + `getBody() : Object`
  + `setType(in type : String) : void`
  + `getType() : String`
  + `toString() : String`
  + `Notification(in notificationName : String, in body : Object = null, in type : String = null)`

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 Unported 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.
Interfaces

IFacade

model (IModel)

view (IView)

controller (IController)

Notifier

IFacade

+registerProxy(in proxy : IProxy) : void
+retrieveProxy(in proxyName : String) : IProxy
+removeProxy(in proxyName : String) : void
+hasProxy(in proxyName : String) : Boolean
+registerMediator(in mediator : IMediator) : void
+retrieveMediator(in mediatorName : String) : IMediator
+removeMediator(in mediatorName : String) : void
+hasMediator(in proxyName : String) : Boolean
+registerCommand(in notificationName : String, in commandClassRef : Class) : void
+removeCommand(in notificationName : String) : void
+sendNotification(in notificationName : String, in body : Object, in type : String)

IModel

+registerProxy(in proxy : IProxy) : void
+retrieveProxy(in proxyName : String) : IProxy
+removeProxy(in proxyName : String) : void
+hasProxy(in proxyName : String) : Boolean

IView

+registerObserver(in notificationName : String, in observer : IObserver) : void
+notifyObservers(in notification : INotification) : void
+registerMediator(in mediator : IMediator) : void
+retrieveMediator(in mediatorName : String) : IMediator
+removeMediator(in mediatorName : String) : void
+hasMediator(in mediatorName : String) : Boolean

IController

+registerCommand(in notificationName : String, in commandClassRef : Class) : void
+executeCommand(in notification : INotification) : void
+removeCommand(in notificationName : String) : void
+hasCommand(in notificationName : String) : Boolean

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 Unported 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.
**Interfaces**

**IProxy**

```java
interface IProxy
{
    String getProxyName();
    void onRegister();
    void onRemove();
    void sendNotification(String notificationName, Object in, String type);
}
```

**IMediator**

```java
interface IMediator
{
    String getMediatorName();
    void setViewComponent(Object viewComponent);
    Object getViewComponent();
    Array getNotificationInterests();
    void handleNotification(INotification notification);
    void onRegister();
    void onRemove();
    void sendNotification(String notificationName, Object in, String type);
}
```

**ICommand**

```java
interface ICommand
{
    void execute(INotification notification);
    String sendNotification(String notificationName, Object in, String type);
}
```

**INotifier**

```java
interface INotifier
{
    void sendNotification(String notificationName, Object in, String type);
}
```

**IObserver**

```java
interface IObserver
{
    void setNotifyMethod(Function notifyMethod);
    void setNotifyContext(Object notifyContext);
    Function getNotifyMethod();
    Object getNotifyContext();
    void addObserver(INotification notification);
}
```
### Interfaces

**INotification**

```java
@interface interfaces: INotification

+getName() : String
+setBody(in body : Object) : void
+getBody() : Object
+setType(in type : String) : void
+getType() : String
+toString() : String
```