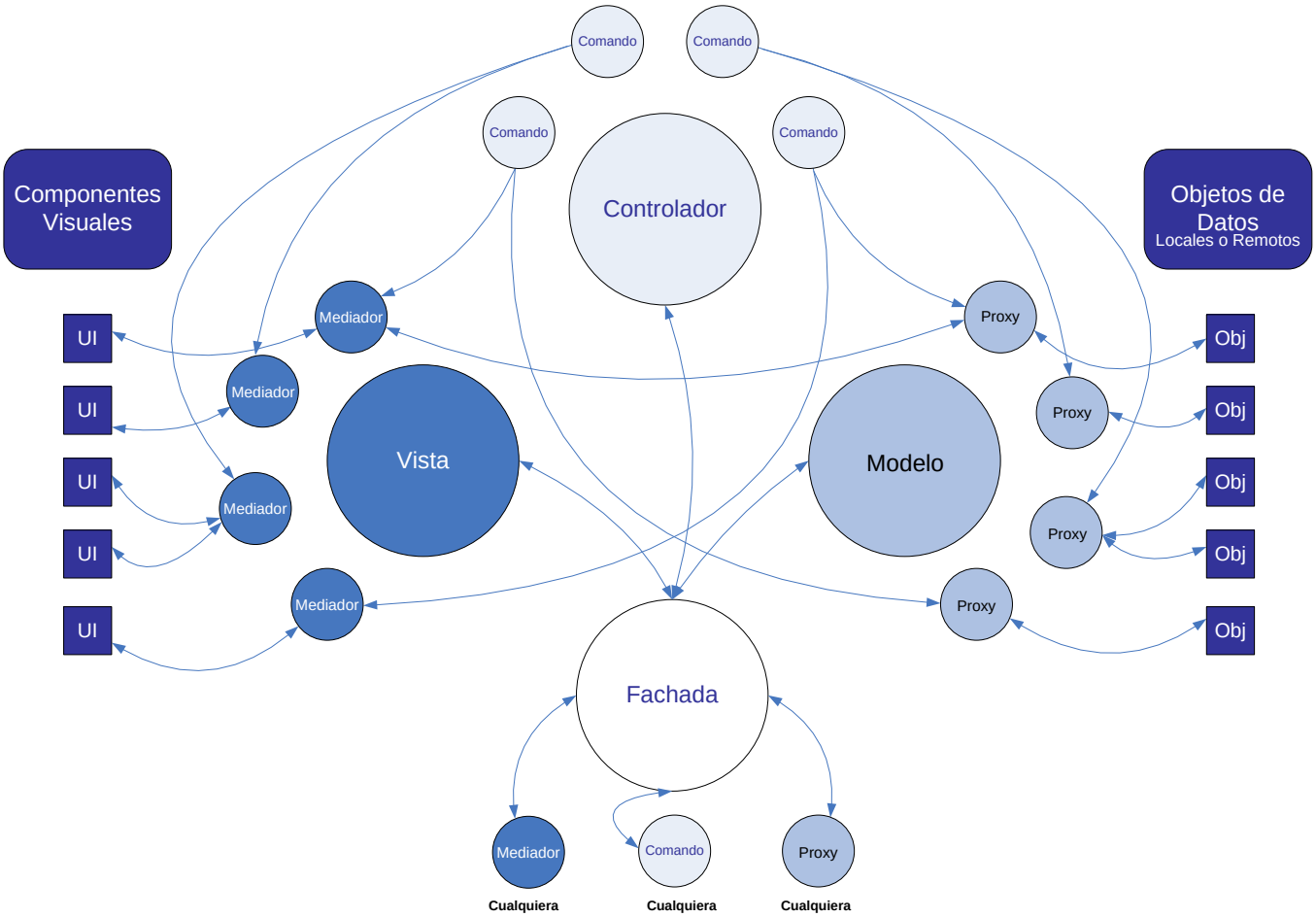
puremvc framework
IMPLEMENTATION IDIOMS AND BEST PRACTICES

pure mvc

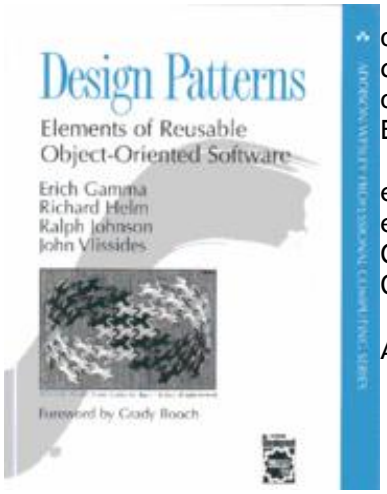
Estilo de Aplicación y Mejores Prácticas

Construcción de aplicaciones cliente robustas, escalables y de fácil mantenimiento utilizando PureMVC con ejemplos en ActionScript 3 y MXML.



puremvc framework
IMPLEMENTATION IDIOMS AND BEST PRACTICES

Inspiración



PureMVC es un patrón basado en un framework originalmente dirigido por la necesidad actual en el interés de diseño de clientes RIAs de alto desempeño. Ahora ha sido portado a otros estilos y plataformas, entre ellas los entornos de servidores. Este documento se centra en el lado cliente.

Si bien la interpretación y las implementaciones son específicas a cada plataforma soportada por PureMVC, los patrones empleados están bien definidos en el famoso libro “Brigada de Cuatro": Patrones de Diseño: Elementos del Software Reutilizable Orientado a Objetos (ISBN 0-201-63361-2).

Altamente recomendado.

puremvc framework
IMPLEMENTATION IDIOMS AND BEST PRACTICES

PureMVC Gestalt

El framework PureMVC tiene un objetivo muy específico. Esto es para ayudarle a separar los objetivos de codificación de su aplicación en tres niveles discretos; Modelo, Vista y Controlador.

Esta separación de objetivos, y la tensión y la dirección de los acoplamientos utilizados para hacerlos trabajar en conjunto es de suma importancia en la construcción de aplicaciones escalables y de fácil mantenimiento.

En esta implementación del clásico meta - patrón de diseño MVC, estos tres niveles de la aplicación se rigen por tres Singleton (una clase de la que sólo una instancia se puede crear) llamados simplemente Modelo, Vista y Controlador. Juntos, ellos se conocen como los "actores principales".

Un cuarto Singleton, la fachada (facade) simplifica el desarrollo, proporcionando una única interfaz para la comunicación con los actores principales.

Modelos y Proxies

El Modelo simplemente cachea referencias con nombres a los Proxies. El código Proxy manipula el modelo de datos, comunicando con los servicios remotos si necesita hacerlos persistentes o recuperarlos.

Esto resulta en el código portable de nivel de Modelo.

Vistas y Mediadores

La Vista primeramente cachea las referencias con nombres a los Mediadores. El código Mediador administra los Componentes Visuales, agregando oyentes (listeners) de eventos, enviando y recibiendo notificaciones hacia y desde el resto del sistema en su nombre y directamente manipulando su estado.

Esto separa la definición de la Vista de la lógica que la controla.

PureMVC Gestalt

Controlador y Comandos

El Controlador mantiene mapeos con nombres a clases de comandos, el mismo es “stateless” (no guarda estados), y sólo se crea cuando se necesita.

Los Comandos deben recibir e interactuar con los Proxies, enviar Notificaciones, ejecutar otros Comandos, y a menudo son utilizados para organizar actividades complejas o sistemas amplios tales como inicios y cierres de aplicación. Ellos son el lugar de residencia de la Lógica de Negocio de su aplicación.

Fachada y principales

La Fachada (facade), otro Singleton, inicializa los “actores principales” (Modelo, Vista y Controlador), y provee un único espacio de acceso a todos sus métodos públicos.

Extendiendo de la Fachada, su aplicación obtiene todos los beneficios de los “actores principales” sin tener que importar y trabajar con ellos directamente. Implemente una Fachada específica para su aplicación sólo una vez y estará simplemente lista.

Proxies, Mediadores y Comandos deben entonces utilizar la Fachada específica de su aplicación a fin de acceder y comunicarse con los demás.

Observadores y Notificaciones

Las aplicaciones PureMVC deben ejecutarse en entornos sin acceso a Eventos Flash y clases EventDispatcher, por que el framework implementa un esquema Observador de notificaciones para comunicaciones entre los actores principales MVC y otras partes del Sistema en una vía de acoplamiento libre.

PureMVC Gestalt

Observadores y Notificaciones

A usted no le conciernen los detalles acerca de la implementación de Observadores/Notificaciones PureMVC, eso es interno del framework. Usted sólo utilizara un único método para enviar Notificaciones desde los Proxies, Mediadores, Comandos y la Fachada misma que ni siquiera requieren que cree una instancia de Notificación.

Las Notificaciones pueden ser utilizadas para disparar la ejecución de un Comando

Los Comandos están mapeados a nombres de Notificaciones en su Fachada específica, y son automáticamente ejecutados por el Controlador cuando las Notificaciones mapeadas son enviadas. Los Comandos típicamente organizan complejas interacciones entre los objetivos de las Vistas y el Modelo, sabiendo tan poco de cada uno como sea posible.

Los Mediadores envían, declaran interés en, y reciben Notificaciones

Cuando son registrados con la Vista, los Mediadores son interrogados por los objetivos que tengan sus métodos llamados “listNotifications” (lista de Notificaciones), y devuelven un arreglo de nombres de Notificaciones en los que está interesado.

Luego, cuando una Notificación con el mismo nombre es enviada por cualquier actor del sistema, los Mediadores interesados son notificados por sus métodos llamados “handleNotification” (manejador de Notificaciones) y se pasa una referencia a la Notificación.

Los Proxies envían, pero no reciben Notificaciones.

Los Proxies envían Notificaciones por varias razones, tales como un servicio remoto de Proxy alertando al sistema que reciba un resultado o un Proxy cuyos datos se han actualizado enviando un cambio de Notificación.

PureMVC Gestalt

Los Proxies envían, pero no reciben Notificaciones.

Cuando un Proxy *escucha* Notificaciones se acopla fuertemente a los niveles de Vista y Controladores.

Estos niveles deben necesariamente escuchar las Notificaciones desde los Proxies, cuando su función es visualmente representada y permite que el usuario interactúe con el Modelo de Datos mantenido por el Proxy.

Además los niveles de Vista y el Controlador deben poder variar sin afectar al nivel de Modelo.

Por ejemplo, una aplicación de administración y una aplicación de usuario relacionada podrían compartir la misma clase de nivel de Modelo. Si sólo los casos de usos difieren pueden ser llevados fuera por diferentes acuerdos Vista/Controlador operando contra el mismo Modelo.

Fachada

Los tres actores principales del meta-patrón MVC son representados en PureMVC por las clases Modelo, Vista y Controlador. Para simplificar el proceso de desarrollo de la aplicación, PureMVC emplea el patrón Fachada (Facade).

La Fachada dirige sus solicitudes al Modelo, Vista y Controlador, así su código no necesita importar otras clases y usted no necesita trabajar con ellos individualmente. La clase Fachada automáticamente instancia el Singleton MVC Fundamental en su constructor.

Típicamente, el framework Fachada se crea una sub clase en su aplicación y es utilizada para inicializar el Controlador con mapeo de Comandos. La preparación del Modelo y la Vista se organizan por Comandos ejecutados por el Controlador.

Fachada

¿Qué es un Comando Específico?

Aunque los actores principales son completos, las implementaciones utilizables, la Fachada proporciona una implementación que debe ser considerada abstracta, en que nunca pueda crear una instancia directamente.

En su lugar, cree una subclase del framework Fachada y añada o escriba sus mismos métodos para hacerlo útil para su aplicación.

Esta Fachada específica es para utilizarla para acceder y notificar Comandos, Mediadores y Proxies que actualmente hacen el trabajo del sistema. Por Convención, es llamada “ApplicationFacade” (Fachada de Aplicación), pero puede llamarlo como guste.

Generalmente, la jerarquía de Vistas de su aplicación (componentes de pantallas) son creados para cualquier proceso que su plataforma normalmente emplea. En aplicaciones Flex, y MXML se instancian todos los hijos o las animaciones Flash crean todos los objetos en el Escenario. Una vez que la jerarquía de Vistas de la aplicación ha sido construida, la maquina PureMVC arranca y el Modelo y las regiones de Vistas son preparadas para su uso.

Su Fachada específica es también utilizada para facilitar el proceso de arranque en un modo que mantiene el código de la aplicación principal conociendo mucho acerca de la maquina PureMVC al que se va a conectar. La aplicación simplemente pasa un parámetro de sí mismo al método llamado “startup” (arranque) en su instancia Singleton de Fachada específica.

Creando una Fachada específica para su aplicación

Su Fachada específica no necesita hacer mucho para proveer a su aplicación de mucho poder. Considere la siguiente implementación:

Fachada

Creando una Fachada específica para su aplicación

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.*;
	//una fachada especifica para MyApp
	public class ApplicationFacade extends Façade implements IFacade
	{
		//definimos los nombres constantes de Notificaciones
		public static const STARTUP:String = "startup";
		public static const LOGIN:String = "login";
		//método de creación del Singleton ApplicationFacade
		public static function getInstance() : ApplicationFacade
		{
			if ( instance == null ) instance = new ApplicationFacade( );
			return instance as ApplicationFacade;
		}
		//registramos los Comandos con el Controlador
		override protected function initializeController( ) : void
		{
			super.initializeController();
			registerCommand( STARTUP, StartupCommand );
			registerCommand( LOGIN, LoginCommand );
			registerCommand( LoginProxy.LOGIN_SUCCESS, GetPrefsCommand );
		}
	//arrancamos  la máquina PureMVC, pasando una referencia a la aplicación
		public function startup( app:MyApp ) : void
		{
			sendNotification( STARTUP, app );
		}
	}
}
```

Fachada

Creando una Fachada específica para su aplicación

Éstas son algunas pocas cosas a notar en el código precedente:

- Extiende de la clase Facade PureMVC, e implementa la interface IFacade.
- No sobrescribe el constructor. Si lo hizo, deberá llamar al constructor de la superclase antes de hacer nada.
- Define un método estático “getInstance” (obtener instancia) que devuelve la instancia Singleton, creándola y cacheándola si lo necesita. La referencia a la instancia se mantiene en una propiedad protegida (protected), de la superclase (Facade) y debe ser casteada al tipo de subclase antes de ser devuelta.
- Define constantes para nombres de Notificaciones. Puesto que es el actor todo lo demás en el sistema lo utilizan para acceder y comunicarse con cada uno de los otros, la Fachada específica es el lugar perfecto para definir los nombres de constantes que son compartidos entre las notificaciones participantes.
- Inicializa el Controlador con Comandos que deben ser ejecutados cuando las correspondientes Notificaciones son enviadas.
- Provee un método de arranque que toma un argumento (en este caso) de tipo MyApp, que se pasa por Notificación al StartupCommand(registrado por la notificación llamada “STARTUP”).

Con éstos simples requerimientos implementación, su Fachada específica heredará un poco de la funcionalidad de la superclase abstracta.

puremvc framework
IMPLEMENTATION IDIOMS AND BEST PRACTICES

Fachada

Inicializando su Fachada específica

El constructor de Fachada (Facade) PureMVC llama a los métodos protegidos para inicializar las instancias del Modelo, Vista y Controlador, y cachearlos para referenciarlos.

Por su composición, la Fachada implementa y expone las características del Modelo, Vista y Controlador; añadiendo sus funcionalidades y evitando la interacción directa del desarrollador con los actores principales del framework.

Entonces, ¿dónde y cómo encaja la fachada en el esquema de las cosas en una aplicación real? Considere la siguiente aplicación Flex:

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[
                        //obtenemos el ApplicationFacade
                        import com.me.myapp.ApplicationFacade;
                        private var facade:ApplicationFacade = ApplicationFacade.getInstance();
                ]]>
        </mx:Script>
        <!-- el resto de la jerarquía de pantalla se define aquí -->
</mx:Application>
```

Esto es. Muy sencillo.

Construir la jerarquía de vistas inicial, obtener la instancia de “ApplicationFacade” (Fachada de Aplicación) e invocar su método “startup” (arranque).

NOTA: en AIR, debemos utilizar “aplicationComplete”, y en Flash podríamos instanciar la Fachada y hacer la llamada a “startup” en el Frame 1 o en una Clase Documento separada.

Fachada

Aspectos claves a notar en este ejemplo:

- Construimos la interface en el modo usual declarativo de MXML; iniciando con una etiqueta <mx:Application>, conteniendo componentes y contenedores, ya sea personalizados o de almacenes.

- Un bloque de script es utilizado para declarar e inicializar una variable privada con una instancia de Singleton de la "ApplicationFacade" específica (Fachada de Aplicación).

- Como estamos inicializando la variable con la llamada al método estático "ApplicationFacade.getInstance", significa que en el momento en que se dispara el evento "creationComplete" de la aplicación, la Fachada se ha creado y con ella, la Vista, el Modelo y el Controlador, aunque los Mediadores y los Proxies no se han creado aún.

- En el manejador "creationComplete" de la etiqueta "Application", invocamos el método "startup", pasando una referencia a la aplicación principal.

Tenga en cuenta que los Componentes de Vistas ordinarios no necesitan conocer o interactuar con la Fachada, pero la aplicación de nivel superior es la excepción a la regla.

La aplicación de nivel superior (o Animación Flash) construye la jerarquía de Vistas, inicializa la Fachada, entonces arranca la máquina PureMVC.

Notificaciones

PureMVC implementa el patrón Observador (Observer) de modo que los “actores principales” y sus colaboradores pueden comunicarse en un modo de acoplamiento flexible, y sin dependencia de la plataforma.

El lenguaje ActionScript no proporciona modelos de Eventos que es utilizado en Flex y Flash, los que vienen desde el paquete de Flash. El framework ha sido portado de otras plataformas como C# y J2ME, por lo que el framework maneja sus comunicaciones internas en lugar de depender de las proporcionadas por la plataforma Flash.

No es simplemente un reemplazo para Eventos, las Notificaciones operan en un modo fundamentalmente diferente, y trabajan sinérgicamente con Eventos para producir Componentes Visuales extremadamente re-utilizables que no necesitan conocer de sus acoplamientos al sistema PureMVC en absoluto si están bien diseñados.

Eventos y Notificaciones

Los Eventos son despachados desde los objetos de visualización de Flash que implementan la interfaz IEventDispatcher. El evento esta en una 'burbuja' en la jerarquía de la pantalla, permitiendo que el objeto padre controle el evento, o de los padres del padre, etc.

Esto es una *cadena de mecanismo de responsabilidades* donde sólo en el linaje padre/hijo tiene responsabilidades para recibir y actuar sobre el Evento a menos que tengan una referencia al despachador y puedan establecer un “escucha” (listener) directamente sobre él.

Las Notificaciones son enviadas por la Fachada y los Proxies; escuchadas y enviadas por los Mediadores; mapeados y enviados por Comandos. Es un mecanismo de publicador/subscriptor donde muchos Observadores pueden recibir y actuar sobre la misma Notificación.

Notificaciones

Eventos y Notificaciones

Las Notificaciones deben tener un “cuerpo” opcional, puede ser cualquier objeto ActionScript.

A diferencia de los Eventos Flash, es raramente necesario crear una clase de Notificación personalizada, ya que puede transportar una carga útil “fuera de la caja”. Por supuesto, puede crear clases de notificación personalizados con el fin de interacciones de tipo fuerte con ellos, pero los beneficios del mundo real, de verificación del tiempo de compilación (para Notificaciones específicamente), sopesan frente a los costos generales de mantenimiento de muchas clases de notificaciones para reducirlo a una cuestión de estilo de programación.

Las Notificaciones también tienen un “tipo” opcional que puede ser utilizado por la el recipiente del Notificador como un discriminador.

Por ejemplo, en una aplicación de edición de documentos, ahí puede tener una instancia de Proxy para cada documento y un correspondiente Mediador para que el Componente Visual lo utilice para editar el documento. El Proxy y el Mediador podrían compartir una única clave que el Proxy pasa como tipo de Notificación.

Todas las instancias de Mediadores se registran para que las Noficaciones del Proxy deban ser notificadas (¿?), pero debe utilizar el tipo de propiedad para determinar si deben actuar o no en consecuencia.

Definiendo Eventos y Constantes

Hemos visto que la Fachada específica es un buen lugar para definir constantes de Notificación, comunes. Siendo el mecanismo central la interacción con el sistema, todas las notificaciones participantes deberán por defecto colaborar con la Fachada.

Notificaciones

Definiendo Eventos y Constantes

Una clase separada “ApplicationConstants” es utilizada en todo momento para éste propósito en vez de la Fachada específica si los nombres de constantes sólo necesitan conocerse por otras aplicaciones.

En cualquier caso, la definición centralizada de constantes para nombres de Notificaciones asegura que cuándo una de las Notificaciones participantes necesita referirse a un nombre de Notificación, podemos hacerlo de un modo seguro que el compilador puede verificar, en lugar de utilizar las cadenas literales que podrían estar mal escritas, pero no producirán error.

Además, no se deben definir nombres de Eventos en la Fachada específica. Definir nombres de constantes estáticamente en las clases límites que generan, o en clases de Eventos personalizados que se despacha.

Representando los límites físicos de la Aplicación, los Componentes Visuales y Objetos de Datos (Data Object) pueden permanecer reutilizables si se comunican con su Mediador o Proxy que despachan Eventos mediante métodos que hacen llamadas o envíos de Notificaciones.

Si un Componente Visual o un Objeto de Datos despacha un Evento que el Mediador manejador o el Proxy está escuchando, es probable que sólo el par de colaboración siempre necesite conocer el nombre particular del Evento. La comunicación entre el oyente y el resto del sistema PureMVC pueda ocurrir a través del uso de notificaciones.

Aunque las relaciones pares (Mediador/Componente Visual y Proxy/Objeto de Dato) son necesariamente un tanto estrechas, se acoplan débilmente al resto de la arquitectura de la aplicación; ofreciendo mayor refactorización contenida del modelo de datos o las interfaces de usuario cuando son requeridas.

Comandos

La Fachada específica generalmente inicializa el Controlador con el seteo de Notificaciones a Comandos necesarios al iniciar.

Para cada mapeo, el Controlador se registra a sí mismo como Observador para la Notificación dada. Cuando es notificado, el Controlador instancia el Comando apropiado, finalmente el Controlador llama a los métodos “execute” de los Comandos, pasados en la Notificaciones.

Los Comandos son “stateless” (no guardan estados); son creados cuando se necesitan y están destinados a desaparecer cuando se han ejecutado. Por esta razón, es importante no instanciar o almacenar referencias a los Comandos en objetos de larga duración.

Uso de Macros y Simples Comandos

Los Comandos, como todas las clases del framework PureMVC, implementa una interface, llamada “ICommand”. PureMVC incluye dos implementaciones de “ICommand” que usted puede extender fácilmente.

La clase “SimpleCommand” simplemente tiene un método de ejecución que acepta una instancia de “INotification”. Inserte su código en el método “execute” y listo.

La clase “MacroCommand” le permite ejecutar múltiples sub-comandos secuencialmente, cada una es creada y pasa una referencia a la Notificación original.

Un “MacroCommand” llama a su método “initializeMacroCommand” desde su constructor interno. Sobrescriba éste método en sus sub-clases para llamar al método “addSubCommand” una vez por cada Comando a agregar. Debe agregar cualquier combinación de “SimpleCommands” o “MacroCommands”.

Comandos

Comandos Acoplados débilmente a Mediadores y Proxies

Los Comandos son ejecutados por el Controlador como un resultado de Notificaciones que son enviadas. Los Comandos nunca se deben instanciar y ejecutar por cualquier otro actor del Controlador.

Para comunicar e interactuar con cualquier otra parte del sistema, los Comandos pueden:

- Registrar, remover o verificar el registro existente de Mediadores, Proxies y Comandos.

- Los Envíos de Notificaciones responden a otros Comandos o Mediadores.

- Recuperar y Proxies y Mediadores y manipularlos directamente.

Los Comandos nos permiten activar fácilmente los elementos de la Vista en estados apropiados, o transportar datos a varias partes del mismo.

Se pueden usar para llevar a cabo interacciones transaccionales con el Modelo que abarca múltiples proxies, y requiere Notificaciones para enviar cuando toda la transacción se ha completado, o manejar excepciones y tomar acción en la falla.

Organización de Acciones Complejas y de Lógica Empresarial

Con algunos lugares en la aplicación en que podría colocar código (Comandos, Mediadores y Proxies); las preguntas que inevitablemente y en repetidas ocasiones surgen son:

Qué código va aquí? Qué, exactamente, debe hacer un Comando?

Comandos

Organización de Acciones Complejas y de Lógica Empresarial

La primera diferenciación a realizar acerca de la lógica en su aplicación es entre la Lógica de Negocio y la Lógica de Dominio.

Lo Comandos tienen la *Lógica de Negocio* de su aplicación; la implementación técnica de los casos de uso de nuestra aplicación espera llevar a cabo contra el *Modelo de Dominio*. Esto involucra coordinación del Modelo y los estados de Vistas.

El Modelo mantiene su integridad mediante el uso de Proxies, que tiene la *Lógica de Dominio*,y expone un API para la manipulación de Objetos de Datos. Ellos encapsulan todos los accesos al modelo de datos ya sea en el cliente o en el servidor, así que todo el resto de la aplicación que es relevante esta ya sea que los datos puedan acceder síncrona o asíncronamente.

Los Comandos pueden utilizar organización de comportamiento complejo de sistema que deben ocurrir en un orden específico, y donde es posible que los resultados de una acción podrían alimentar a la siguiente.

Los Mediadores y Proxies deben exponer una interface de curso-granular a los Comandos (y los demás), que ocultan la implementación de sus Componentes Visuales manejados u Objetos de Datos.

Note que cuando hablamos acerca de Componente Visual nos referimos a un botón o widget con el que el usuario interactúa directamente. Cuando hablamos acerca de Objetos de Datos que incluyen estructuras arbitrarias que mantienen datos así como servicios remotos los utilizamos para recibirlos o almacenarlos.

Los Comandos interactúan con Mediadores y Proxies, pero deben ser aislados desde las implementaciones límites. Considere los siguientes Comandos utilizados para preparar el sistema para su uso:

Comandos

Organización de Acciones Complejas y de Lógica Empresarial

StartupCommand.as:

```
package com.me.myapp.controller
{
	import org.puremvc.as3.interfaces.*;
	import org.puremvc.as3.patterns.command.*;
	import com.me.myapp.controller.*;
	//un Macro Comando se ejecuta cuando la aplicacion inicia.
	public class StartupCommand extends MacroCommand
	{
		//se inicializa el Macro Comando añadiendo sus subcomandos.
		override protected function initializeMacroCommand() : void
		{
			addSubCommand( ModelPrepCommand );
			addSubCommand( ViewPrepCommand );
		}
	}
}
```

Este es un Macro Comando que añade dos sub-comandos, que se ejecutan en orden FIFO cuando el Macro Comando es ejecutado.

Esto proporciona una 'cola' de acciones a ser completadas al inicio. ¿Pero qué debemos hacer exactamente, y en qué orden?.

Antes de que el usuario pueda presentar o interactuar con cualquiera de los datos de aplicaciones, el Modelo debe colocarse en un consistente, estado conocido. Una vez que esto se ha logrado, la Vista se prepara para presentar los datos del Modelo y permite al usuario manipularlos e interactuar con ellos.

Por lo tanto, el proceso de inicio de la aplicación consiste de dos amplios seteos de actividades – preparación del Modelo, seguido de la preparación de la Vista.

Comandos

Organización de Acciones Complejas y de Lógica Empresarial

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.*;
	//creamos y registramos Proxies con el Modelo
	public class ModelPrepCommand extends SimpleCommand
	{
		//llamando al Macro Comando
		override public function execute( note : INotification ) : void
		{
			facade.registerProxy( new SearchProxy() );
			facade.registerProxy( new PrefsProxy() );
			facade.registerProxy( new UsersProxy() );
		}
	}
}
```

La preparación del Modelo es usualmente un simple asunto de creación y registro de todos los Proxies que el sistema necesita al iniciarse.

La clase “ModelPrepCommand” anterior es un Comando Simple que prepara el Modelo para su uso. Es el primero de los sub-comandos del Macro Comando previo, y por tanto es ejecutado primero.

Por medio de la Fachada específica, crea y registra las varias clases Proxy que el sistema utiliza al iniciarse. Note que el Comando no hace ninguna manipulación o inicialización del del Modelo de datos. El Proxy es responsable por cualquier recuperación de datos, creación o inicialización necesaria para preparar los Objetos de Datos para el uso del sistema.

Comandos

Organización de Acciones Complejas y de Lógica Empresarial

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.*;
	//creamos y registramos los Mediadores con la Vista.
	public class ViewPrepCommand extends SimpleCommand
	{
		override public function execute( note : INotification ) : void {
			var app:MyApp = note.getBody() as MyApp;
			facade.registerMediator( new ApplicationMediator( app ) );
		}
	}
}
```

Esto es un Comando Simple que prepara la Vista para su uso. Es el último de los sub-comandos del Macro Comando previo, y por tanto, el último en ser ejecutado.

Note que el único Mediador creado y registrado es el “AplicationMediator”, que administra el Componente Visual de la Aplicación.

Más adelante, se pasa el cuerpo de la Notificación dentro del constructor del Mediador. Esto es una referencia a la Aplicación, enviado por la Aplicación misma como cuerpo de la Notificación cuando la Notificación de INICIO original fue enviada. (Relacionado al ejemplo previo MyApp).

Comandos

Organización de Acciones Complejas y de Lógica Empresarial

La Aplicación es un Componente Visual especial en que se instancian y tiene como hijos a todos los demás Componentes Visuales que son inicializados en tiempo de arranque.

Para comunicarse con el resto del sistema, los Componentes Visuales necesitan tener Mediadores. Y crear los Mediadores requieren una referencia a los Componentes Visuales que deben mediar, que sólo se conocen en éste punto de la Aplicación.

El Mediador de la Aplicación es la única clase que estamos permitiendo conocer cualquier cosa acerca de la implementación de la Aplicación, así manejamos la creación de los demás mediadores dentro del constructor del Mediador de la Aplicación.

Por tanto, con los tres anteriores Comandos, tenemos una inicialización organizada y ordenada del Modelo y de la Vista. Al hacerlo así, los Comandos no necesitan conocer mucho acerca del Modelo y la Vista.

Cuando los detalles del Modelo o la implementación de la Vista cambian, los Proxies y Mediadores son refactorizados necesariamente.

La Lógica de Negocios en los Comandos deben ser aislados de refactorización que tienen lugar en los límites de la Aplicación.

El Modelo debe encapsular la 'lógica de dominio' manteniendo la integridad de los datos en los Proxies. Los Comandos llevan a cabo lógicas “transaccionales” o “de negocio” contra el Modelo, encapsulando la coordinación de transacciones multi-Proxy o manejando y reportando excepciones a medida que la Aplicación exige.

Mediadores

Una clase Mediador es utilizada para mediar en las interacciones del usuario con uno o más de los Componentes Visuales (como los FlexDatagrids o los Flash Movie Clips) y el resto de la Aplicación PureMVC.

En una Aplicación basada en Flash, un mediador típicamente sitúa los oyentes de eventos (events listeners) en sus Componentes Visuales para manejar los gestos del usuario y peticiones de datos desde el Componente. Envía y recibe Notificaciones para comunicarse con el resto de la aplicación.

Responsabilidades del Mediador específico

Los frameworks Flash, Flex y AIR proveen un gran vector de componentes IU (UI) ricos-interactivos. Puede extender éstos o escribir los suyos en ActionScript para proporcionar infinitas posibilidades de representar el modelo de datos al usuario y permitirle interactuar con ellos.

En un futuro no muy distante, habrán otras plataformas ejecutando ActionScript. Y el framework ha sido portado y demostrado en otras plataformas incluyendo Silverligh y J2ME, expandiendo los horizontes del desarrollo de aplicaciones RIA con éstas tecnologías.

Un objetivo del framework PureMVC es ser neutral a las tecnologías siendo utilizadas en los límites de la Aplicación y proporcionando un único estilo para adaptar cualquier componente IU o Dato de estructura/servicio con el que podría encontrarse tratando en el momento.

Para la Aplicación basada en PureMVC, un Componente Visual es cualquier componente IU, a pesar de en qué framework sea proporcionado por o cómo muchos sub-componentes que lo contienen. Un Componente Visual debe encapsular como mucho su propio estado y operaciones posibles, exponiendo una única API de eventos, métodos y propiedades.

Mediadores

Responsabilidades del Mediador específico

Un Mediador específico nos ayuda a adaptar uno o más Componentes Visuales para la aplicación teniendo las únicas referencias a los componentes e interactuando con la API que exponen.

Las responsabilidades del Mediador primeramente la manipulación de Eventos despachados desde el Componente Visual y relevando Notificaciones enviadas desde el resto del sistema.

Desde que los Mediadores interactúan frecuentemente con los Proxies, es lo común para un Mediador que reciba y mantenga una referencia local para acceso frecuentemente a Proxies en su constructor. Esto reduce las llamadas repetitivas de solicitud de Proxy para obtener la misma referencia.

Casteando Implícitamente un Componen Visual
La implementación basada en el Mediador que viene con PureMVC acepta un nombre y un Objeto genérico como único argumento del constructor.

El constructor de su Mediador específico pasará su Componente Visual a la superclase e inmediatamente la hará disponible internamente como una propiedad disponible llamada “viewComponent”, tipada genéricamente como Objeto.

Debe también configurar dinámicamente un Componente Visual del Mediador después de ser construido llamando a su método “setViewComponent”.

Sin embargo cuando ya fue seteado, frecuentemente necesitará castear este Objeto a su tipo actual, a fin de acceder cualquier API que expone, puede ser una práctica engorrosa y repetitiva.

Mediadores

Responsabilidades del Mediador específico

ActionScript proporciona una característica de lenguaje llamada “getter” y “setter” implícitos. Un “getter” implícito es como un método, pero se expone como una propiedad al resto de la clase o aplicación. Esto resulta ser muy útil en la solución del problema de casteo frecuente.

Un estilo útil a emplear en su Mediador específico es un “getter” específico que castee el Componente Visual a su tipo actual y le dé un nombre significativo.

Crear un método como este:

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

Entonces, en cualquier lugar de nuestro Mediador, en lugar de hacer esto:

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

Podemos reeamplazarlo por esto:

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

Escuchando y Respondiendo al Componente Visual

Un Mediador usualmente sólo tiene un Componente Visual, pero podría manejar algunos, como una Barra de Herramientas de Aplicación y los botones o controles que contiene. Podemos contener un grupo de botones relacionados (como un formulario) en un único Componente Visual y exponer los “hijos” al Mediador como propiedades. Pero lo mejor es encapsular tantas implementaciones de los componentes como sea posible. Teniendo el intercambio de datos de componente con un tipo de Objeto personalizado mejor.

Mediadores

Escuchando y Respondiendo al Componente Visual

El Mediador maneja los la interacción con los niveles de Controlador y Modelo, actualizando el Componente Visual cuando las Notificaciones relevantes son recibidas.

En la plataforma Flash, típicamente seteamos los oyentes de Eventos en el Componente Visual cuando el Mediador es construido o cuando el método “setViewComponent” es llamado, especificando un manejador de método:

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

Lo que el mediador haga en respuesta a ese evento, por supuesto depende completamente las exigencias del momento.

Generalmente, un evento del Mediador específico maneja métodos que realizan algunas combinaciones de éstas acciones:

- Inspecciona el Evento para el tipo o contenido personalizado si se espera.
- Inspecciona o modifica las propiedades expuestas (o llama métodos expuestos) de un Componente Visual.
- Inspecciona o modifica las propiedades expuestas (o llama métodos expuestos) de un Proxy.
- Envía una o más Notificaciones que son respondidos por otros Mediadores o Comandos (o posiblemente una instancia del mismo Mediador).

Mediadores

Escuchando y Respondiendo al Componente Visual

Algunas buenas reglas de oro son:

- Si un numero de otros Mediadores deben involucrarse en en todas las respuestas a un Evento, entonces actualiza un Proxy común o envía Notificación, que es respondida por los Mediadores interesados, los cuales responder adecuadamente.

- Si se quiere una gran cantidad de acciones coordinadas con los otros Mediadores, una buena práctica es utilizar un Comando para codificar los pasos en un sólo lugar.

- Considere una mala práctica recibir otros mediadores y actuar sobre ellos, o diseñar Mediadores que expongan métodos de manipulación.

- Para manipular y distribuir información de estado de la aplicación a los Mediadores, establecemos valores o llamando métodos en Proxies creados para mantener estados. Deje Mediadores interesados en las Notificaciones enviadas por los Proxies que mantienen los estados.

Manejando Notificaciones en el Mediador Específico

En contraste a la adición explícita de oyentes de eventos a los Componentes Visuales, el proceso de acoplamiento del Mediador al sistema PureMVC es simple y automático.

Tras el registro con la Vista, el Mediador es interrogado en cuanto a sus objetivos de Notificación. Responde con un arreglo de nombres de Notificaciones que desea manejar.

Mediadores

Manejando Notificaciones en el Mediador Específico

La forma más sencilla es con una única expresión que crea y devuelve un único arreglo anónimo, publicado por el nombre de la Notificación, se deben definir como constantes estáticas, normalmente en la Fachada específica.

Definir la lista de Notificaciones en las que el Mediador esta interesado es fácil:

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

Cuando una de las Notificaciones nombradas en la respuesta del Mediador es enviada por cualquier actor en el sistema (incluyendo el mismo Mediador), los métodos “handleNotifications” del Mediador serán llamados, y la Notificación pasada.

Debido a su legibilidad, y la facilidad con la que uno puede refactorizar para añadir o remover manejadores de Notificaciones, el constructor “switch/case” es preferible antes que la expresión de estilo “if/else” dentro del método “handleNotifications”.

En esencia, debe haber poco que hacer para responder a cualquier cambio de Notificación, y toda la información necesaria debe estar en la misma Notificación. Ocasionalmente alguna información debe ser recibida desde algún Proxy basado en información interna de una Notificación, pero esto no debe complicar la lógica de negocio en el manejador de Notificaciones. Si esto es así, entonces es una señal de que está tratando de poner la lógica de negocio que pertenece a un Comando en el manejador de la Notificación de su Mediador.

Mediadores

Manejando Notificaciones en el Mediador Específico

```
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;
                }
        }
}
```

También, un típico método “handleNotifications” maneja no más de 4 o 5 Notificaciones.

Más que eso, es una señal de que las responsabilidades del Mediador deben ser divididas granularmente. Crear Mediadores para sub-componentes del Componente Visual en lugar de tratar manejarlos todos en un Mediador monolítico.

El uso de un único, método de notificación predeterminado es la diferencia clave entre el modo en que un Mediador oye los Eventos y cómo oye las Notificaciones.

Con los Eventos, tenemos un número de métodos manejadores, normalmente uno por cada Evento que el Mediador maneja. Generalmente éstos métodos sólo envían Notificaciones. No deben ser complicados. No debe micro-administrar demasiados detalles de Componente Visual, ya que los Componentes Visuales deben ser escritos para encapsular los detalles de implementación, exponiendo una API proceso-granular al Mediador.

Mediadores

Manejando Notificaciones en el Mediador Específico

Con las Notificaciones, usted tiene un único método de manipulación, en su interior debería manejar todas las Notificaciones en las que el Mediador esta interesado.

Es mejor contener las respuestas para Notificaciones enteramente dentro del método "handleNotifications", permitiendo que la sentencia "switch" las separe claramente por nombre de Notificación.

Ha habido mucho debate sobre el uso de la sentencia "switch / case" como muchos desarrolladores lo consideran una limitación ya que todos los casos se ejecutan dentro del ámbito del mismo método. Sin embargo, el único método de notificación y el estilo "switch / case" fue elegido específicamente para limitar lo que se hace en el interior del Mediador, y sigue siendo el recomendado a utilizar.

El Mediador tiene la intención de mediar las comunicaciones entre el Componente Visual y el resto del sistema.

Considere el rol de un traductor mediando el intercambio de conversación entre el Embajador y el resto de los miembros en la conferencia de la ONU. Raramente debería de hacer algo más que una simple traducción y el envió de mensajes, a veces de gran alcance para una metáfora apropiada o hecho. Lo mismo ocurre con el rol del Mediador en PureMVC.

Acoplamiento de Mediadores a Proxies y otros Mediadores

Ya que la Vista es la última instancia cargada con representación del modelo de datos en un modo gráfico, interactivo, un relativo modo único de acoplamiento a los Proxies de la aplicación es de esperar. La Vista debe saber acerca del Modelo, pero el Modelo no necesitar conocer ningún aspecto de la Vista.

Mediadores

Manejando Notificaciones en el Mediador Específico

Los Mediadores puede libremente recuperar Proxies desde el Modelo, y leer o manipular los Objetos de Datos via cualquier API expuesta por el Proxy. Sin embargo, haciendo el trabajo en un Comando se afloja el acoplamiento entre la Vista y el Modelo.

De la misma manera, los Mediadores podrían recuperar referencias a otros Mediadores desde la Vista y leer o manipular de cualquier modo expuesto por el Mediador recuperador.

Además ésto no es una buena práctica, ya que conduce a las dependencias entre las partes de la Vista, que impacta negativamente la capacidad de refactorizar una parte de la Vista sin afectar a otra.

Un Mediador que desea comunicarse con otra parte de la Vista debe enviar una Notificación en lugar de recuperar otro Mediador y manipularlo directamente.

Los Mediadores no deben exponer métodos para manipular directamente sus Componentes Visuales manejados, sino que deben responder sólo a Notificaciones que llevan a cabo esa labor.

Si demasiada manipulación de los Componentes Visuales internos se realiza en un Mediador (en respuesta a un Mediador o Notificación), refactorizar uno que trabaje en un método en el Componente Visual, encapsulando su implementación tanto como sea posible, obtieniendo mayor reusabilidad.

Si demasiada manipulación de Proxies o sus datos es realizada en un Mediador, refactorizar uno que trabaje en un Comando, simplificando el Mediador, moviendo la Lógica de Negocios dentro de Comandos donde deben ser reutilizados por otras partes de la Vista, y aflojando el acoplamiento de la Vista al Modelo en la mayor medida posible.

Mediadores

Interacción del Usuario con Componentes Visuales y Mediadores

Considere un componente Panel de Inicio de Sesión presentando un formulario. Éste tiene un “PanelLoginMediator” que permite al usuario comunicar sus credenciales e intensión de acceder al sistema al interactuar con el Panel de Inicio de Sesión y responder a sus entradas al iniciar un intento de acceso.

La colaboración entre el componente Panel de Inicio de Sesión y el “PanelLoginMediator” es que el componente envía un Evento TRY_LOGIN cuando el usuario ha ingresado sus credenciales y busca acceder al sistema. El “PanelLoginMediator” maneja el Evento al enviar una Notificación con los componentes publicados del LoginVO como cuerpo.

LoginPanel.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml"
    title="Login" status="”{loginStatus}”>
    <!-- El evento que dispara este componente. Desafortunadamente no podemos usar
    los nombres de constantes aqui, porque MetaData es una directiva del compilador -->
    <mx:MetaData>
        [Event('tryLogin')];
    </mx:MetaData>
    <mx:Script>
        <![CDATA[
            import com.me.myapp.model.vo.LoginVO;
            //los campos del formulario  bindean bidireccionalmente a estos objetos
            [Bindable] public var loginVO:LoginVO = new LoginVO();
            [Bindable] public var loginStatus:String = NOT_LOGGED_IN;
            //define una constant en el componente de la vista para todos los nombres de
            eventos
            public static const TRY_LOGIN:String='tryLogin';
            public static const LOGGED_IN:String='Logged In';
            public static const NOT_LOGGED_IN:String='Enter Credentials';
        ]]>
    </mx:Script>
    <mx:Binding source="username.text" destination="loginVO.username"/>
```

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

El componente Panel de Inicio de Sesión publica un nuevo LoginVO con las entradas del formulario de usuario y cuando hace click en el botón “Login” un Evento es despachado. Entonces el “LoginPanelMediator” de hará cargo.

Esto deja a los Componente Visuales con el único papel de recopilación de los datos y alertar al sistema cuando está listo.

Un componente más completo sólo habilita el botón “Login” cuando, tanto los campos de nombre de usuario y contraseña tienen contenido, previniendo un intento de inicio de sesión fallido antes de que sea hecho.

El Componente Visual oculta su implementación interna, su API entera utilizada por el Mediador consistente de un Evento TRY_LOGIN, una propiedad LoginVO a ser verificada, y una propiedad Panel de estado.

Mediadores

Interacción del Usuario con Componentes Visuales y Mediadores

El "LoginPanelMediator" también responderá a las Notificaciones LOGIN_FAILED y LOGIN_SUCCESS y seteara el estado de "LoginPanel".

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;
	//un mediador para interactuar con el component Panel de Login.
	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 );
		}
		//lista de notificaciones que nos interesan
		override public function listNotificationInterests( ) : Array
		{
			return [
				LoginProxy.LOGIN_FAILED,
				LoginProxy.LOGIN_SUCCESS
				];
		}
		//manejador de notificaciones
		override public function handleNotification( note:INotification ):void
		{
			switch ( note.getName() ) {
```

Mediadores

Interacción del Usuario con Componentes Visuales y Mediadores

```
                        case LoginProxy.LOGIN_FAILED:
                                LoginPanel.loginVO = new LoginVO( );
                                loginPanel.loginStatus = LoginPanel.NOT_LOGGED_IN;
                                break;
                        case LoginProxy.LOGIN_SUCCESS:
                                loginPanel.loginStatus = LoginPanel.LOGGED_IN;
                                break;
                        }
                }
                //el usuario clickea en el boton, trata de iniciar sesión.
                private function onTryLogin ( event:Event ) : void {
                sendNotification( ApplicationFacade.LOGIN, loginPanel.loginVO );
        }
        //casteamos el component visual a su tipo actual.
        protected function get loginPanel() : LoginPanel
        {
                return viewComponent as LoginPanel;
        }
    }
}
```

Note que el “LoginPanelMediator” sitúa un oyente de Evento en el Panel de Inicio de Sesion (LoginPanel) en su constructor, así que el método “onTryLogin” será invocado cuando el usuario haga click en el botón “Login”. En el método “onTryLogin”, la Notificación LOGIN es enviada, teniendo el LoginVO cargado.

Previamente, registramos el “LoginCommand” para esta Notificación. Éste invocará el método login del “ProxyLogin”,pasando el LoginVO. El “LoginProxy” intentará el logueo con el servicio remoto y enviará una Notificación LOGIN_SUCCESS o LOGIN_FAILED. Estas clases son definidas al final de la sección en los Proxies.

Las listas del “LoginPanelMediator” LOGIN_SUCCESS y LOGIN_FAILED son las Notificaciones

que interesan, por lo que independientemente del resultado, será notificado, y establecerá el "LoginStatus" del Panel de Inicio de Sesión a LOGGED_IN en caso de éxito; y NOT_LOGGED_IN en caso de falla, limpiando el LoginVO.

Proxies

En general, el patrón Proxy es utilizado para proporcionar un marcador de posición para un objeto con el fin de controlar el acceso a él. En una aplicación basada en PureMVC, la clase Proxy es utilizada exclusivamente para manejar una porción del modelo de datos de la aplicación.

Un Proxy puede administrar el acceso a una estructura de datos creada localmente y de una complejidad arbitraria. Esto es el Objeto de Datos del Proxy (Proxy's Data Object).

En este caso, las mejores prácticas para interactuar con ellos probablemente involucren configuración (setting) y obtención (getting) de sus datos en forma síncrona. Se pueden exponer todos o parte de las propiedades o métodos de sus Objetos de Datos, o una referencia a su mismo Objeto de Datos. Cuando expone métodos para actualización de datos, también envía Notificaciones, al resto del sistema, de que el dato ha cambiado.

Un Proxy remoto puede ser utilizado para encapsular interacción con un servicio remoto que guarde o devuelva una pieza de datos. El Proxy puede conservar el objeto que comunica con el servicio remoto, y controlar el acceso a los datos enviados y recibidos desde el servicio.

En tal caso, uno puede setear datos o llamar a un método del Proxy y esperar una Notificación asíncrona, enviada por el Proxy cuando el servicio ha recibido desde el extremo remoto.

Responsabilidades del Proxy específico

El Proxy específico nos permite encapsular una pieza del modelo de datos, venga de donde venga y sea cual sea su tipo, para administración del Objeto de Datos y el acceso de la Aplicación al mismo.

Proxies

Responsabilidades del Proxy específico

La implementación de la clase Proxy que viene con PureMVC es un simple objeto portador de datos que puede ser registrado con el Modelo.

A pesar de que es completamente utilizable de esta forma, normalmente deberá crear una sub-clase de Proxy y añadir funcionalidad específica para el Proxy particular.

Las variaciones comunes del patrón Proxy incluyen:

- Proxy Remoto, donde los datos son manejados por el Proxy específico estan en una ubicación remota y se accede por un servicio de algún tipo.
- Proxy y Delegado, donde acceder a un objeto servicio requiere compartirse entre múltiples Proxies. La clase Delegate (Delegado) mantiene el objeto servicio y controla el acceso al mismo, asegurando que las respuestas sean apropiadamente encaminadas a sus solicitantes.
- Proxy de Protección, utilizado cuando los objetos necesitan tener diferentes permisos de accesos.
- Proxy Virtual, que crean objetos costosos a pedido.
- Proxy Inteligente, carga objetos de datos en memoria en el primer acceso, realiza conteo de referencias, permite bloquear un objeto para asegurar que otro objeto no lo cambie.

Casting explícito del Objeto de Datos

La implementación base del Proxy que viene con PureMVC acepta un nombre y un Objeto genérico como argumento del constructor. Usted debe configurar dinámicamente y establecer un Objeto de Datos del Proxy después de ser construido y por llamadas

Proxies

Casting explícito del Objeto de Datos

a su método “setData”.

Así como con el Mediador y su Componente Visual, usted necesitará castear este Objeto a su tipo actual, para poder acceder a cualquiera de las propiedades y métodos que expone, en la mejor de las prácticas engorrosas y repetitivas, pero posiblemente conduzcan a prácticas que expongan más implementaciones de los Objetos de Datos de los que son necesarios.

Además, dado que el Objeto de Dato es a menudo una estructura compleja necesitaremos frecuentemente tener a mano referencias a algunas partes de la estructura, además de una referencia encasillada a la estructura misma.

Una vez más la característica del lenguaje ActionScript llamada implícitos “getters” y “setters” resulta muy útil en la resolución del casteo frecuente e inadecuado conocimiento de la implementación de la aplicación.

Una práctica muy útil a emplear en su Proxy específico es un “getter” implícito que castée el Objeto de Datos a su tipo actual y le asigne un nombre significativo.

Adicionalmente, puede definir múltiples tipos diferentes de “getters” para recuperar partes específicas del Objeto de Datos.

Por ejemplo:

```
public function get searchResultAC () : ArrayCollection
{
```

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

Proxies

Casting explícito del Objeto de Datos

En cualquier lugar de un Mediador, en lugar de hacer esto:

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

Podemos hacer esto:

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

Prever Acoplamiento con Mediadores

El Proxy no es interrogado en cuanto a sus objetivos de Notificaciones como lo es el Mediador, ni es notificado en cada ocasión, ya que no debe preocuparse del estado de la Vista. En su lugar, el Proxy expone métodos y propiedades para permitir a los otros actores manipularlo.

El Proxy específico no debería recuperar ni manipular Mediadores como una manera de informar al sistema acerca de los cambios a su Objeto de Datos.

En su lugar debe enviar Notificaciones que serán respondidas por Comandos o Mediadores. Cómo el sistema se vea afectado por éstas Notificaciones no debería tener consecuencias para el Proxy.

Al mantener el nivel del modelo libre de todo conocimiento de cualquier implementación del sistema, los niveles de Vista y Controlador podrán ser refactorizados sin que afecte al nivel de Modelo.

El adverso no es del todo cierto. Es difícil para el nivel del Modelo cambiar sin afectar el nivel de Vista y posiblemente de Controlador como resultado. Después de todo, éstos niveles existen sólo para permitir al usuario interactuar con el nivel de Modelo.

Proxies

Encapsular Lógica de Dominio en Proxies

Un cambio en el nivel de Modelo casi siempre resulta en algunas refactorizaciones de los niveles de Vista y Controlador.

Incrementamos la separación entre el nivel de Modelo y los objetivos combinados de los niveles de Vista y Controlador asegurando que colocamos tanto de la Lógica de Dominio, como sea posible, dentro de los Proxies.

Por ejemplo, el cálculo de impuestos de ventas es una función de Lógica de Negocio que debe residir en un Proxy, no en un Mediador o en un Comando.

A pesar de que se puede hacer en cualquiera de éstos lugares, colocarlo en un Proxy no es sólo lógico, sino que mantiene los otros niveles ligeros y fáciles de refactorizar.

Un Mediador puede recuperar el Proxy; llama a la función de impuesto de venta, quizás pasando algunos ítems de formulario. Pero colocar el cálculo actual en el Mediador sería incrustar la Lógica de Dominio en el nivel de Vista. El cálculo de impuesto de venta es una norma perteneciente al Modelo de Dominio. La Vista simplemente lo ve como una propiedad del Modelo de Dominio, disponible si las entradas apropiadas están presentes.

Imagine que la aplicación que está trabajando actualmente es un RIA entregado en un navegador para resoluciones a escala de escritorio. Una nueva versión es liberada para resolución de PDA con un pequeño grupo de casos de uso, pero sigue con todos los requerimientos de nivel de Modelo a día de hoy.

Proxies

Encapsular Lógica de Dominio en Proxies

Con la separación de los objetivos, es posible reutilizar el nivel de Modelo en su totalidad y simplemente ajustarle los nuevos niveles de Vista y Controlador.

A pesar de colocar el cálculo actual de impuestos de ventas en el Mediador, puede parecer eficiente o fácil al momento de implementar, usted acaba de tomar los datos de un formulario y desea calcular impuestos de ventas y surge en el modelo como una orden, tal vez.

Sin embargo, en cada versión de su aplicación tendrá que duplicar su esfuerzo o copiar y pegar la lógica de impuesto de venta en su nuevo, y completamente diferente, nivel de Vista, en lugar de que tener que mostrarlo en forma automática con la inclusión de su biblioteca en el nivel de Modelo.

Interactuando con Proxies Remotos

Un Proxy Remoto es simplemente un Proxy que obtiene su Objeto de Dato desde alguna ubicación remota. Ésto generalmente significa que interactuámos con él en forma asíncrona.

Cómo el Proxy obtenga sus datos depende de la plataforma del cliente, la implementación del servicio remoto, y las preferencias del desarrollador. En un entorno Flash/Flex podemos utilizar HTTPService, WebService, RemoteObject, DataService o incluso XMLSocket para hacer las solicitudes de servicios desde el interior del Proxy.

Dependiendo de los requerimientos, un Proxy Remoto enviará solicitudes dinámicamente,

a raíz de tener un conjunto de propiedades o un método llamado; o podría hacer una petición individual en tiempo de construcción y proveer get/set de acceso a los datos a partir de entonces.

Proxies

Encapsular Lógica de Dominio en Proxies

Hay una serie de optimizaciones que pueden ser aplicados en el Proxy para incrementar la eficiencia de la comunicación con un servicio remoto.

Se puede construir de forma que el cache de datos, a fin de reducir el “chattiness” de la red; o para enviar actualizaciones sólo a ciertas partes de una estructura de datos que tienen que cambiar, reduciendo el consumo de ancho de banda.

Si una la solicitud es dinámicamente invocada en un Proxy Remoto por otro actor en el sistema, el Proxy debe enviar una Notificación cuando el resultado ha sido devuelto.

La parte interesada en la solicitud, pueden ser o no la misma que inició la solicitud.

Por ejemplo, el proceso de invocación de una búsqueda en un servicio remoto y despliegue de resultados puede seguir estos pasos:

- Un Componente Visual inicia una búsqueda despachando un Evento.
- Su Mediador responde recuperando el Proxy Remoto apropiado y seteando una propiedad “serachCriteria”.
- La propiedad “serachCriteria” del Proxy es realmente un “setter” implícito, que

almacena un valor e inicia una búsqueda por medio de un HTTPService al que se escucha esperando un Evento de resultado o un fallo.

- Cuando el servicio devuelve, despacha un “ResultEvent” que el Proxy responde seteando una propiedad pública en sí mismo al resultado.

Proxies

Encapsular Lógica de Dominio en Proxies

- El Proxy entonces envía una Notificación indicando el éxito, agrupando una referencia al objeto de dato como el cuerpo de la Notificación.

- Otro Mediador ha expresado previamente interés en una Notificación particular y tomando el dato desde el cuerpo de la Notificación, y seteandose como la propiedad “dataService” de su Componente Visual manejado.

O considere un “LoginProxy”, que tiene un LoginVO (un Objeto de Valor(Value Object); una clase simple transportadora de datos). El LoginVO puede parecerse a esto:

```
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
        }
}
```

El “LoginProxy” expone métodos para configurar las credenciales, inicio de sesión, cierre de sesión, y recuperando la señal de autorización que debe incluirse en un servicio llamado posterior al login utilizando este esquema particular de autenticación.

Proxies

Encapsular Lógica de Dominio en 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;
        //un proxy para loguear la entrada del usuario
        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 f unction 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 );
        }
        //casteamos el data object con el getter implícito
        public function get loginVO( ) : LoginVO {
                return data as LoginVO;
        }
        //el usuario es logueado si el login VO contiene una ficha de autorización
        public function get loggedIn():Boolean {
                return ( authToken != null );
        }
```

Proxies

Encapsular Lógica de Dominio en Proxies

```
        // subsequentemente llamamos a los servicios despues de de que el login incluya
        la ficha de autorizacion
        public function get authToken():String {
                return loginVO.authToken;
        }
    //seteamos las credenciales del usuario y lo logueamos, o lo deslogueamos e intentamos de nuevo
        public login( tryLogin:LoginVO ) : void {
                if ( ! loggedIn ) {
                        loginVO.username= tryLogin.username;
                        loginVO.password = tryLogin.password;
                } else {
                        logout();
                        login( tryLogin );
                }
        }
        //al loguearse, simplemente limpiamos el LoginVO
        public function logout( ) : void
        {
        if ( loggedIn ) loginVO = new LoginVO( );
                sendNotification( LOGGED_OUT );
        }
```

```
                //notificamos al sistema del logueo exitoso
                private function onResult( event:ResultEvent ) : void
                {
                        setData( event.result ); //inmediatamente disponible como loginVO
                        sendNotification( LOGIN_SUCCESS, authToken );
                }
                // notificamos al sistema del logueo fallido
                private function onFault( event:FaultEvent) : void
                {
                        sendNotification( LOGIN_FAILED, event.fault.faultString );
                }
        }
}
```

Proxies

Encapsular Lógica de Dominio en Proxies

Un LoginCommand puede recuperar el LoginProxy, establecer sus credenciales, e invocar el método “login”, llamando al servicio.

Un GetPrefsCommand puede responder a la Notificación LOGIN_SUCCESS, recuperando el authToken desde el cuerpo de la Notificación y hacer una llamada al siguiente servicio que recupera las preferencias del usuario.

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 );
        }
```

```
    }
}
```

Proxies

Encapsular Lógica de Dominio en Proxies

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 );
                }
        }
}
```