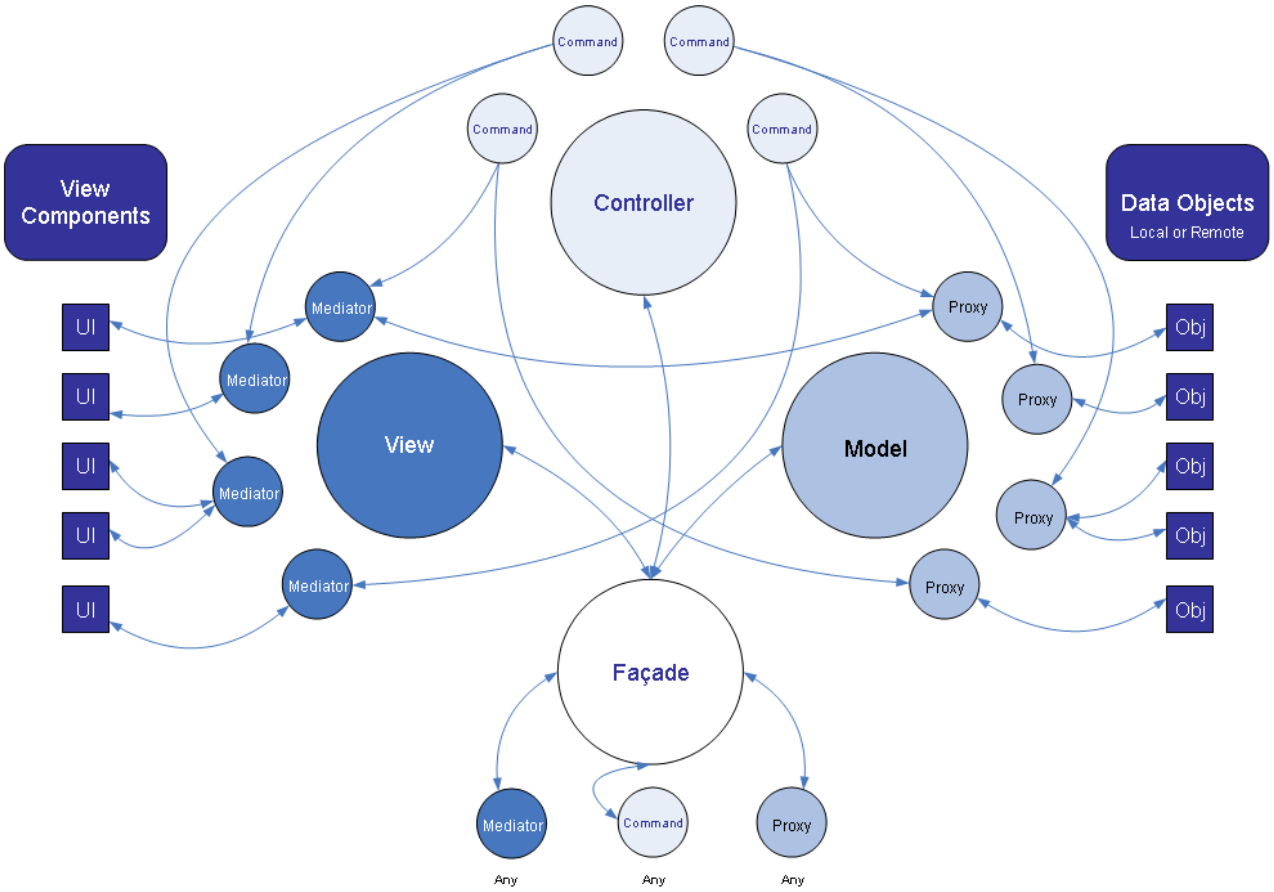
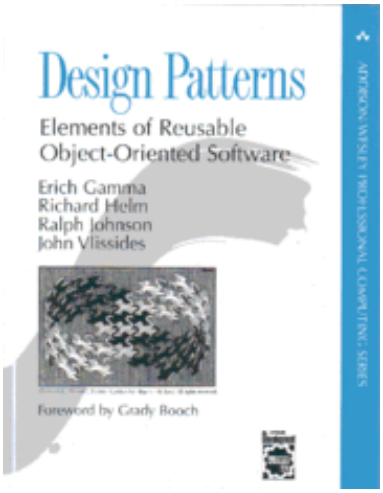
Идиомы реализации и лучшие практики

Создание устойчивых, масштабируемых и удобных в сопровождении клиентских приложений с PureMVC, с примерами на ActionScript 3 и MXML



AUTHOR: Cliff Hall <cliff@puremvc.org>
TRANSLATORS: Denis Sheremetov <flexc0dr@gmail.com> sheremetov.com
Denis Volokh <denis.volokh@gmail.com> denisvolokh.blogspot.com
Dmitry 'Reijii' Kochetov <kodjii@gmail.com> reijii.solartxit.com
Roman Pavlenko < russia.roman@gmail.com> riactive.ru
Rostislav 'RostislavR' Siryk < rostislav.siryk @gmail.com> en.flash-ripper.com
LAST MODIFIED: 20/05/2008

Духовная литература



PureMVC — это фреймворк, основанный на шаблонах проектирования. Он появился из насущной необходимости проектирования высокопроизводительных RIA-клиентов. Сейчас он уже портирован на другие языки и платформы, включая серверные среды. Этот документ посвящен клиентским приложениям.

Хотя интерпретация и реализация имеют свои особенности для каждой поддерживаемой
PureMVC платформы, используемые паттерны описаны в известной книге 'Gang of Four': **Design Patterns: Elements of Reusable Object-Oriented Software (GoF)**
(ISBN 0-201-63361-2)

Очень рекомендуем.

puremvc framework
IMPLEMENTATION IDIOMS AND BEST PRACTICES

Вступительное слово

Друзья, представляем вашему вниманию перевод официальной документации к PureMVC на русский язык. Перевести этот документ предложил Денис "mrJazz" Шереметов в чате UAFPUG 9 февраля 2009 года. На его предложение отозвались Роман Павленко, Денис "Barmaleychik" Волох, Дмитрий "Reijii" Кочетов и Ростислав "RostislavR" Сирык.

Отозвавшись, они разбили документ на пять частей, и тайно перевели его на русский язык. Тут с ними связался Денис «Хитрый» Романко и спросил, есть ли материалы по PureMVC на русском языке, он очень хотел его изучить (не язык, а PureMVC). Обрадовавшись этому совпадению, заговорщики предложили Хитрому прочесть перевод, но при условии, что он напишет свои критические комментарии к нему.

Денис, как человек, на тот момент мало знакомый с PureMVC, идеально подошел на роль бета-тестера. Он написал свои комментарии к документу и они были тут же учтены в финальной версии.

Так появилось нижеследующее.

Как мы перевели термины — примечания переводчиков

В переводе учтены существующие популярные русскоязычные материалы по паттернам, такие, как этот **обзор паттернов проектирования** и Википедия. Но, невзирая на это, многие термины переведены иначе, т.е. лучше ☺

#	Термин	Перевод
1	Pattern	При первом упоминании: "паттерн (шаблон проектирования)", при всех следующих упоминаниях: "паттерн"
2	tier	Уровень
3	Core Actors	Базовые классы
4	Mediator	Медиатор (Посредник)
5	View component	Компонент представления, Представление
6	Notification	Оповещение
7	Proxy	Прокси
8	mapping	Связывание, присваивание, назначение, проецирование
9	named references	Именованные ссылки
10	Control	Компонент
11	custom typed Object	Специализированный объект
12	Controller	Контроллер
13	Concrete class	Конкретный класс
14	Flash display objects	Объекты Flash
15	dispatcher	Отправитель/вещатель
16	Observer	Наблюдатель
17	loosely-coupled	Слабо связанные (компоненты системы)
18	use case	Сценарий использования
19	override	Переопределять (унаследованный метод класса)
20	Singleton	Синглтон, Одиночка, синглетон, псих-одиночка
21	actor	Игрок (игрок системы)
22	Data Object	Объект данных
23	Framework	Фреймворк, фреймверк, фреймуорк, фрамеворк
24	Domain Logic	Логика предметной области

Концепция PureMVC

Фреймворк PureMVC преследует очень узкую цель. Она в том, чтобы помочь вам разделить интересы кода вашего приложения на три отдельных уровня: *Модель* (Model), *Представление* (View) и *Контроллер* (Controller).

Для создания масштабируемых и легко поддерживаемых приложений высоким приоритетом являются разделение интересов, а так же герметичность и направление связей между уровнями MVC"

В данной реализации классического *мета*-паттерна MVC эти три уровня приложения управляются тремя синглтонами (классами, для которых возможно существование одного и только одного экземпляра), называемыми просто: *Модель* (Model), *Представление* (View) и *Контроллер* (Controller). Все вместе они называются *Базовыми классами* (Core actors).

Четвертый синглтон, Фасад (Façade), упрощает разработку, предоставляя единый интерфейс для сообщения с *Базовыми классами*.

Модель и Прокси (Model & Proxies)

Модель просто кэширует именованные ссылки к *Прокси*. Код *Прокси* манипулирует моделью данных, связываясь с удаленными сервисами, если нужно сохранить или запросить данные.

Таким образом, *Модель* данных изолирована от контроллеров и представления. Это приводит к переносимому коду *Модели*.

Представление и Медиаторы (View & Mediators)

Представление в первую очередь кэширует именованные ссылки на *Медиаторы*. Код *Медиатора* обслуживает компонент(ы) *Представления*, добавляя к ним *Слушателей событий* и от их имени отправляя и получая оповещения к и от остальной системы, при этом непосредственно управляя их состоянием.

Это отделяет определение *Вида* от управляющей им логики.

Концепция PureMVC

Контроллер и Команды (Controller & Commands)

Контроллер кеширует ссылки на классы Команд, создавая экземпляры класса команды в тот момент, когда возникает необходимость выполнения этой команды и уничтожая после выполнения.

Команды могут запрашивать *Прокси* и взаимодействовать с ними, отправлять *Оповещения*, выполнять другие *Команды*, и часто используются для *дирижирования* сложными, охватывающими всю или почти всю систему действиями, такими, как запуск или остановка приложения. Это альма-матер *бизнес-логики* вашего приложения.

Фасад и Ядро (Façade & Core)

Фасад (Façade), еще один синглтон, инициализирует *Базовые классы* (Модель, Предтавление и Контроллер) и предоставляет единую точку доступа ко всем их публичным методам.

Расширяя *Фасад*, ваше приложение получает в распоряжение весь функционал *Базовых классов* без необходимости их импорта и прямой работы с ними. Вы реализуете *конкретный Фасад* в своем приложении только один раз, и это делается очень просто.

Прокси, *Медиаторы* и *Команды* могут использовать *конкретный Фасад* вашего приложения для того, чтобы получать доступ и связываться друг с другом.

Наблюдатели и Оповещения (Observers & Notifications)

PureMVC-приложения могут выполняться в средах, отличных от Flash Player, не имея доступа к его событиям и диспетчерам (классы Event и EventDispatcher), так что фреймворк реализует схему оповещений *Наблюдатель* (Observer) для сообщения между *Базовыми классами MVC* и другими частями системы в манере слабого связывания.

Концепция PureMVC

Наблюдатели и Оповещения (Observers & Notifications)

Вам не нужно беспокоиться о деталях реализации *Наблюдателя/Оповещения* (Observer/Notification) в PureMVC; это внутренняя часть фреймворка. Вам нужно только использовать простой метод для отправки *Оповещений* от *Прокси*, *Медиаторов*, *Команд* и *Фасада*, который даже не требует создавать экземпляр *Оповещения*.

Оповещения можно использовать для запуска Команд

Команды связаны с именами *Оповещений* в вашем *конкретном Фасаде* и автоматически выполняются *Контроллером*, когда отправляются назначенные им *Оповещения*. *Команды* обычно дирижируют сложным взаимодействием между интересами *Представления* и *Модели*, при этом зная о них настолько мало, насколько это возможно.

Медиаторы отправляют и получают Оповещения, а так же заявляют о заинтересованности в них

При регистрации в *Представлении Медиаторы* опрашиваются на предмет их заинтересованности в *Оповещениях*, для чего вызывается метод listNotifications, и должны возвращать массив имен *Оповещений*, в которых они заинтересованы.

Позже, когда кто-то в системе отправляет одноименное *Оповещение*, заинтересованные *Медиаторы* будут оповещены через вызов их метода handleNotification, которому будет передана ссылка на *Оповещение*.

Прокси отправляют, но не получают Оповещений

Прокси могут отправлять *Оповещения* по различным поводам. Например, *Прокси* для удаленного сервиса может *оповестить* систему о том, что он получил результат с сервера. Или другой *Прокси* может оповестить систему о том, что изменились его данные.

Концепция PureMVC

Прокси отправляют, но не получают Оповещений

Для *Прокси слушать Оповещения* — это слишком сильное связывание с уровнями *Вида* и *Контроллера*.

Эти уровни обязаны слушать *Оповещения* от *Прокси*, так как их функция заключается в визуальном представлении и обеспечении взаимодействия пользователя с данными *Модели*, за которые отвечают *Прокси*.

Тем не менее, уровни *Вида* и *Контроллера* должны иметь возможность изменений, не влияющих на уровень *Модели данных*.

Например, административное приложение и связанное с ним пользовательское приложение могут иметь общие классы уровня *Модели*. Если отличаются только сценарии использования, то эти отличия можно реализовать за счет различных комбинаций *Представления* и *Контроллера*, работающих с одной и той же *Моделью*.

Фасад (Façade)

Три *Базовых класса* мета-паттерна MVC представлены в PureMVC классами *Модели*, *Представления* и *Контроллера*. Чтобы упростить процесс разработки приложения, PureMVC задействует паттерн *Фасад*.

Фасад распределяет ваши запросы к *Модели*, *Представлению* и *Контроллеру*, так что ваш код не нуждается в импорте этих классов и вам не нужно работать с ними индивидуально. Класс *Фасада* автоматически создает экземпляры базовых синглтонов MVC в своем конструкторе.

Обычно *Фасад* самого фреймворка становится над-классом вашего приложения и используется для инициализации *Контроллера* с назначениями *Команд*. Подготовка *Модели* и *Представления* затем дирижируется *Командами*, выполняемыми *Контроллером*.

Фасад

Что такое Конкретный Фасад (Concrete Façade)?

Хотя *Базовые классы* являются завершенными, готовыми к использованию реализациями, *Фасад* предоставляет реализацию, которую следует рассматривать как *абстрактную* в том смысле, что вы никогда не создаете его экземпляр непосредственным образом.

Вместо этого вы наследуете *Фасад из фреймворка* и / или переопределяете некоторые его методы, чтобы сделать их полезными конкретно для вашего приложения. *(Примечание переводчика: вы конкретизируете абстрактный Фасад фреймворка в конкретный Фасад своего приложения)*.

Этот *конкретный Фасад* затем используется для доступа и оповещения *Команд*, *Медиаторов* и *Прокси*, которые делают свою работу в системе. По соглашению, он называется 'ApplicationFacade', но вы можете назвать его как хотите.

В общем случае иерархия *Представления* вашего приложения (компоненты дисплея) будет создаваться согласно принятого для вашей платформы процесса. Во Flex — MXML-приложения создает всех своих потомков или Flash-приложение создает все свои объекты на *Сцене*. Как только построена иерархия *Представления* приложения, стартует аппарат PureMVC и регионы *Модели* и *Вида* готовятся к использованию.

Ваш *конкретный Фасад* также используется для облегчения процесса старта приложения, в том смысле, что он освобождает основной код приложения от необходимости глубокой осведомленности об аппарате PureMVC, к которому это приложение будет подключено. Приложение просто передает ссылку на себя в метод 'startup' синглтон-экземпляра вашего *конкретного Фасада*.

Создание Конкретного Фасада для вашего приложения

Вашему *конкретному Фасаду* не обязательно прилагать много усилий, чтобы передать приложению силу PureMVC. Рассмотрим следующую реализацию:

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.*;

     // Конкретный фасад для MyApp
    public class ApplicationFacade extends Façade implements IFacade
    {
        // Определим имена констант для Оповещений
        public static const STARTUP:String = "startup";
        public static const LOGIN:String = "login";

        // Метод фабрики Синглтона ApplicationFacade
        public static function getInstance() : ApplicationFacade
        {
            if ( instance == null ) instance = new ApplicationFacade( );
            return instance as ApplicationFacade;
        }

        // Регистрируем Команды через Контроллер
        override protected function initializeController( ) : void
        {
            super.initializeController();
            registerCommand( STARTUP, StartupCommand );
             registerCommand( LOGIN, LoginCommand );
            registerCommand( LoginProxy.LOGIN_SUCCESS, GetPrefsCommand );
        }

        // Запускаем аппарат PureMVC, передавая внутрь ссылку на
        public function startup( app:MyApp ) : void
        {
            sendNotification( STARTUP, app );
        }
    }
}
```

Фасад

Создание Конкретного Фасада для вашего приложения

- Отметим несколько моментов в приведенном выше коде:
- Он расширяет класс Facade PureMVC, который в свою очередь реализует интерфейс IFacade.
- Он не переопределяет конструктор. Если бы он это делал, то нужно было бы вызвать конструктор суперкласса прежде, чем делать что-либо.
- Он определяет статический метод getInstance, возвращающий экземпляр Синглтона, создавая и кэшируя его при необходимости. Ссылка на экземпляр сохраняется в защищенном (protected) свойстве супер-класса (Façade) и должна приводиться к типу подкласса перед возвращением.
- Он определяет константы для имен *Оповещений*. Поскольку это игрок, используемый всеми другими участинками системы для доступа и связи друг с другом, *конкретный Фасад* является идеальным местом для определения констант, общих для всех участников обмена оповещениями.
- Он инициализирует *Контроллер Командами*, которые будут выполняться при отправке соответствующих *Оповещений*.
- Он предоставляет метод startup, который принимает один аргумент (в данном случае) типа MyApp, который с помощью *Оповещения* передается *Команде* StartupCommand (зарегистрированной на имя оповещения STARTUP).

С этими простыми требованиями реализации ваш конкретный Фасад унаследует ощутимую часть функциональности своего абстрактного класса-родителя.

Фасад

Инициализация вашего Конкретного Фасада

Конструктор *Фасада* PureMVC вызывает защищенные методы для инициализации экземпляров *Модели*, *Представления* и *Контроллера* и кэширует их ссылки на них.

Затем по композиции *Фасад* реализует и делает доступными возможности *Модели*, *Представления* и *Контроллера*; агрегируя их функциональность и защищая разработчика от прямого взаимодействия с *Базовыми классами* фреймворка.

Итак, где и как *Фасад* внедряется в реальное положение вещей конкретного приложения? Рассмотрим следующее Flex-приложение:

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[
            // Получить ApplicationFacade
            import com.me.myapp.ApplicationFacade;
            private var facade:ApplicationFacade = ApplicationFacade.getInstance();
    ]]>
</mx:Script>
<!— Здесь определена остальная часть иерархии представления  -->
</mx:Application>
```

Вот и все. Достаточно просто.

Создайте эту начальную иерархию вида, получите экземпляр ApplicationFaçade и вызовите его метод startup.

ПРИМЕЧАНИЕ: В AIR, мы бы использовали событие 'applicationComplete', а во Flash мы могли создать экземпляр Фасада и вызвать startup в первом кадре или в отдельном Классе Документа (Document Class).

Фасад

Инициализация вашего Конкретного Фасада

Ключевые моменты этого примера:

- Мы создавали интерфейс как обычно, в декларативном стиле MXML; начиная с тэга <mx:Application>, содержащего компоненты и контейнеры.
- Блок кода используется для объявления и инициализации приватной переменной-ссылки на экземпляр синглтона конкретного ApplicationFacade.
- Поскольку мы инициализируем переменную вызовом статического метода ApplicationFacade.getInstance, то это означает, что в момент события Application creationComplete будет создан *Фасад*, а вместе с ним и *Модель*, *Представление* и *Контроллер*, хотя ни *Медиаторы*, ни *Прокси* не будут пока созданы.
- В обработчике события creationComplete тэга Application мы вызываем метод startup, передавая ему ссылку на главное приложение в качестве аргумента.

Заметьте, что обычные компоненты *Представления* не нуждаются в знании того, как им нужно взаимодействовать с Фасадом, но объект верхнего уровня Application является исключением для этого правила.

Верхнеуровневый объект Application (или Flash-приложение) строит иерархию *Вида*, инициализируют *Фасад*, а затем запускает аппарат PureMVC.

Оповещения

В PureMVC реализован паттерн *Наблюдатель* (Observer) так, что *Базовые классы* и взаимодействующие с ними классы могут общаться «слабо сцепленным» образом, и без привязок к платформе.

ActionScript не предоставляет событийную модель и то что используется во Flex и Flash реализовано в классах пакета.

Фреймворк портирован на другие платформы, такие как C# и J2ME, поэтому используется внутренний механизм взаимодействия вместо привязки к реализации на Flash.

Это не просто замена для *Событий* (Events). *Оповещения* (Notifications) работают в корне иначе, и органичное совмещение с *Событиями* (Events) предоставляет возможность создавать компоненты *Представления* для многократного использования, которые могут даже не знать о том, что они связаны с PureMVC приложением, если всё построено правильно.

События и Оповещения

События вещаются из объектов Flash, реализующих интерфейс IEventDispatcher. *Событие* «поднимается вверх» по иерархии, позволяя родительским объектам перехватывать *Событие*, либо передавать выше по иерархии.

Это механизм цепочки ответственностей, посредством которого возможность получить *Событие* имеют дети/родители, выполняя действия через *Событие*, либо имея ссылку на отправителя и подписываясь «слушателем» на событие.

Оповещения рассылаются *Фасадом* и *Прокси*; слушаются и отсылаются *Медиаторами*; отправляются *Командам*. Это механизм публикации/подписки, посредством которого многие *Наблюдатели* (Observers) могут получать и обрабатывать одно и тоже *Оповещение*.

Оповещения

События и Оповещения

Оповещения могут иметь, если нужно, «тело», любой объект ActionScript.

В отличии от событий Flash, создание специализированных классов *Оповещений*, требуется редко, потому может быть использовано прямо «из коробки». Вы можете, конечно, создавать специализированные классы *Оповещений* для строгой типизации при взаимодействии с ними, но, нужно соизмерять преимущества проверки времени компиляции (в частности для *Оповещений*) и накладных расходов по поддержки множественных классов *Оповещений*, так что это, скорее, вопрос стиля программирования.

Оповещения также имеют опциональный «тип», который моет быть использован получателем для классификации.

Например, в приложении редактора документов, это может быть экземпляр *Прокси* для каждого документа, который открыт, и соответствующий *Медиатор* для компонента Представления, используемый для редактирования документа. *Прокси* и *Медиатор* могут совместно использовать уникальные ключ, который *Прокси* передаст как тип *Оповещения*.

Все экземпляры *Медиаторов* регистрируются для перехвата *Оповещений* от *Прокси*, используя тип *Оповещения* для принятия решения о его обработке.

Определение Оповещений и констант Событий

Мы видим, что конкретный *Фасад* это хорошее место для определения общего в приложении, констант *Оповещений*. Поскольку это центральный механизм для взаимодействий с системой, все подписавшиеся на оповещения будут по умолчанию взаимодействовать с *Фасадом*.

Оповещения

Определение Оповещений и констант Событий

Иногда для этих целей вместо определения констант в *Фасаде* приложения используется отдельный класс “Констант Приложения”, в случае если эти константы должны быть доступны другому приложению.

В любом случае, централизованное определение констант для *Оповещений* гарантирует, что когда один из подписчиков должен сослаться к имени *Оповещения*, мы можем делать это безопасным образом, оставляя проверку мелких ошибок в синтаксисе компилятору.

Однако не следует определять имена *Событий* в *конкретном фасаде*. Определите их как статичные константы классов, которые их вызывают, или же в случае кастомных *Событий*, внутри классов этих *Событий*.

Реализации частей Приложения, компонентов *Представления* и *Объектов Данных* могут оставаться многократно используемыми, если они взаимодействуют со связанными *Медиаторами* и *Прокси* не через вызовы методов а через отсылку *Оповещений*.

Если компонент *Представления* либо *Объект Данных* вещает *Оповещение* в котором заинтересован *Медиатор* или *Прокси*, то знание имени этого события нужно только этой паре, всё остальное взаимодействие между слушателем и остальной частью PureMVC приложения может быть организовано посредством *Оповещений*.

Несмотря на то, что отношения этих взаимодействующих пар (*Медиатор*/*Представление* и *Прокси*/*Объект Данных*) довольно близки, они остаются «слабо связанными» с остальными частями приложения; при желании модифицировать пользовательский интерфейс либо модель данных могут быть легко подвергнуты рефакторингу.

Команды

Определенный *Фасад* инициализирует *Контроллер,* передавая *Командам* связанные с ними *Оповещения* необходимые для старта приложения.

Для каждой связки (*Mapping*), *Контроллер* регистрируется как *Наблюдатель* над данным *Оповещением*. Когда происходит *Оповещение*, *Контроллер* создает экземпляр соответствующей *Команды*. И, наконец, *Контроллер* вызывает у созданной команды метод execute(), передавая в него данное *Оповещение*.

Команды единоразовы; они создаются по требованию (приходит *Оповещение*), и после их исполнения (вызов метода execute) они должны удалятся. В связи с чем, важно не создавать экземпляры и не хранить ссылки на *Команды* в объектах, которые будет использоваться в течении долго срока в вашем приложении.

Использование Макро- и Простых команд

Команды, как и все классы фреймворка PureMVC, реализовывают интерфейс, а именно интерфейс ICommand. PureMVC содержит две реализации ICommand которые вы можете с легкостью расширять.

Класс SimpleCommand (Простая команда) всего-навсего содержит метод execute, который принимает в качестве параметра экземпляр INotification. Вписываете вашу функциональность в метод execute и все готово к работе.

Класс MacroCommand (Макрокоманда) позволяет выполнять несколько подкоманд последовательно, каждой из которых будет после создания передан экземпляр текущего Оповещения.

Класс MacroCommand из конструктора вызывает свой метод initializeMacroCommand. Вы должны переопределить его в своем классе макрокоманды, для вызова метода addSubCommand для каждой команды, которую вы ходите добавить в макрокоманду. Вы можете добавлять любые *Команды* или *Макрокоманды*.

Команды

Слабое связывание Команд с Медиаторами и Прокси

Команды выполняются *Контроллером* в результате отправки *Оповещения*. *Команды* не должны быть созданы и выполнены никем кроме *Контроллера*.

Чтобы общаться и взаимодействовать с другими частями системы, *Команды* могут:

- Регистрировать, удалять или проверять существуют ли *Медиаторы*, *Прокси* и *Команды*.
- Посылать *Оповещения*, для получения ответной реакции от других команд или *Медиаторов*.
- Получать экземпляры *Прокси* и *Медиаторов* и управлять ими напрямую.

Команды позволяют нам легко переключать элементы *Представления* между различными состояниями (через *Медиаторы*), или передавать данные к различным частям этих элементов.

Они могут быть использованы для осуществления транзакционного взаимодействия с *Моделью,* которую охватывают несколько *Прокси*, требовать отправки *Оповещения* по окончанию транзакции, или обрабатывать непредвиденные ситуации (exceptions) и принимать соответствующие действия.

Взаимодействие сложных действий и бизнес-логики

В местах вашего приложения, где вы могли бы разместить код (*Команд*, *Медиаторов* и *Прокси*) неизбежно и периодически будет возникать вопрос:

Какой код и где писать? Что конкретно должна делать *Команда*?

Команды

Взаимодействие сложных действий и бизнес-логики

Первое разделение логики в вашем приложении коснется бизнес-логики и логики предметной области.

В командах находится бизнес-логика нашего приложения; ожидается, что техническая реализация сценариев использования нашего приложения будет выполняться на уровне *модели предметной области* (Domain Model). Это подразумевает координацию *Модели* и состояниями *Представления*.

Модель поддерживает свою целостность, используя *Прокси*, которые размещены в *логике предметной области* (Domain Logic), открывая API для работы с объектами данных. Они инкапсулируют весь доступ к *моделям данных*, находящимся на клиенте или на сервере, для того чтобы остальная часть приложения которая работает с данными, могли иметь к ним синхронный или асинхронный доступ.

Команды могут быть использованы для управления рядом действий в системе, которые должны происходить в определенном порядке, с возможностью того что результат предыдущего действия может быть использован для последующего.

Медиаторы и *Прокси* должны предоставлять крупномодульные интерфейсы *Командам* (и наоборот), которые скрывают реализацию объектов данных и компонентов представления, которыми они управляют.

Заметьте, что когда мы говорим про *компонент представления*, мы имеем в виду кнопку или виджет, с которым непосредственно взаимодействует пользователь. Когда мы говорим об *объекте данных*, то подразумеваем произвольные структуры данных, а также удаленные сервисы, которые могут быть использованы для хранения или получения данных.

Команды взаимодействуют с *Медиаторами* и *Прокси*, но должны быть изолированы от граничных реализаций. Вот пример того как Команда используется для подготовки приложения к работе:

Команды

Взаимодействие сложных действий и бизнес-логики

StartupCommand.as:

```
package com.me.myapp.controller {

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

        // MacroCommand запускается при старте приложения
        public class StartupCommand extends MacroCommand {
                // Инициализация макрокоманды добавлением подкоманд
                override protected function initializeMacroCommand() : void  {
                        addSubCommand( ModelPrepCommand );
                        addSubCommand( ViewPrepCommand );
                }
        }
}
```

Это *макрокоманда,* которая содержит две подкоманды, которые при вызове *макрокоманды* исполняются в порядке очереди (FIFO).

Это создает 'очередь' верхнего уровня из действий, которые должны быть выполнены на старте приложения. Но что конкретно должны мы сделать, и в каком порядке?

Прежде чем пользователь увидит или сможет взаимодействовать с приложением и его данными, *Модель* должна быть надлежащим образом подготовлена. Как только это сделано, *Представление* может быть подготовлено для отображения данных *Модели* и позволить пользователю взаимодействовать с ними.

Следовательно, процесс старта обычно состоит из двух обширных групп операций — подготовка *Модели*, и последующая подготовка *Представления*.

Команды

Взаимодействие сложных действий и бизнес-логики

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.*;

        // Создание и регистрация Прокси с Моделью
        public class ModelPrepCommand extends SimpleCommand {
                // Вызов Макрокомманды
                override public function execute( note : INotification ) : void  {
                        facade.registerProxy( new SearchProxy() );
                        facade.registerProxy( new PrefsProxy() );
                        facade.registerProxy( new UsersProxy() );
                }
        }
}
```

Подготовка *Модели,* обычно, является простым созданием и регистрацией всех *Прокси*, в которых приложение будет нуждаться при запуске.

Пример команды ModelPrepCommand, это Простая команда, которая подготавливает *Модель* к дальнейшей работе. Это первая из подкоманд макрокоманды, и поэтому она будет выполнена первой.

Через *конкретный Фасад*, она создает и регистрирует те *Прокси*, которые система будет использовать при запуске. Заметьте, что *Команда* не делает никаких манипуляций или инициализаций с данными *Модели*. *Прокси* отвечает за любое получение данных, создание или инициализацию необходимых *Объектов Данных* для использования в системе.

Команды

Взаимодействие сложных действий и бизнес-логики

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.*;

	// Создание и регистрация Медиатора с Представлением
	public class ViewPrepCommand extends SimpleCommand {
		override public function execute( note : INotification ) : void {
			var app:MyApp = note.getBody() as MyApp;
			facade.registerMediator( new ApplicationMediator( app ) );
		}
	}
}
```

Это простая *команда* которая подготавливает *Представление* для работы. Это последняя из подкоманд *макрокоманды*, следовательно, будет выполнена последней.

Отметьте, что создается и регистрируется только *Медиатор* ApplicationMediator, который обслуживает *компонент представления* приложения.

В дальнейшем, тело *Оповещения* передается в конструктор медиатора. Это ссылка на приложение, переданная самим приложением вместе с *Оповещением*, когда первоначальное *Оповещение* STARTUP было послано. (Согласно предыдущему примеру приложения MyApp.)

Команды

Взаимодействие сложных действий и бизнес-логики

Приложение это в некоторой степени специальный компонент *Представления*, в котором реализуются и находятся в качестве потомков все другие компоненты *Представления*, которые инициализируются при запуске приложения.

Для взаимосвязи с остальной частью системы, компоненты *Представления* должны обладать *Медиаторами*. И создание этих *Медиаторов* требует ссылку на компоненты *Представления*, с которыми они будут работать, а это на данном этапе известно только приложению.

Медиатор приложения - это единственный класс, которому допускается знать все про реализацию Представления приложения, так что создание оставшихся *Медиаторов* будет размещена внутри его конструктора.

Используя эти три команды, мы обеспечили последовательную инициализацию Модели и Представления. При этом *Команды* не должны знать много о *Модели* или о *Представлении*.

При изменении *Модели* или реализации *Представления*, *Прокси* и *Медиаторы* должны быть изменены соответствующим образом.

Бизнес-логика внутри Команд не должна зависеть от изменений, происходящих в областях применения.

Модель должна формировать «логику предметной области», поддерживая целостность данных внутри *Прокси*. *Команды* выполняют «транзакционную» или «бизнес» логику в Модели, формируя координацию транзакций мульти-Прокси или обрабатывая и сообщая об исключительных ситуациях.

Медиаторы

Медиаторы используются для взаимодействия пользователей с одним или более компонентами *Представления*, (например Flex DataGrid либо Flash MovieClip), и остальными частями PureMVC приложения.

Во Flash-приложении *Медиатор* является местом, где обычно устанавливают обработчики событий *Представления* для обработки пользовательских действий и запросов данных от компонента. Он посылает и принимает *Оповещения* (Notifications) для взаимодействия с приложением.

Задачи конкретного Медиатора

Фреймворки для Flash, Flex и AIR включают немало компонент для построения графических интерфейсов. Вы можете использовать стандартные компоненты, либо реализовывать свои для представления модели данных пользователю, и позволять ему взаимодействовать с ними.

В скором будущем будут реализованы новые платформы для запуска ActionScript приложений. Также фреймворк портируется на другие платформы, например уже существуют реализации для Silverlight и J2ME, расширяя горизонты его применения.

Задача PureMVC - оставаться нейтральным к используемым технологиям в приложении, предоставляя простые идиомы для работы с любыми компонентами графического интерфейса или *Моделью Данных*.

Для PureMVC-приложения компонент *Представления* это любой элемент управления графического интерфейса, либо контейнер с несколькими компонентами, независимо от конкретной платформы. *Представление* должно инкапсулировать как можно больше собственных состояний и операций, предоставляя простой интерфейс для взаимодействия с ним.

Медиаторы

Задачи конкретного Медиатора

Конкретный *Медиатор* позволяет использовать один или более компонентов *Представления* в приложении используя только ссылки и предоставленное API.

Основная задача Медиатора - обработка событий (Events), инициированных компонентом Представления и касающихся его Оповещений (Notifications).

Медиаторы также часто взаимодействуют с *Прокси*. Довольно распространенная практика получения и хранения локальных ссылок на часто используемые экземпляры *Прокси* в конструкторе. Это уменьшает многочисленные вызовы retrieveProxy для получения одних и тех же ссылок.

Неявное преобразование типов компонентов Представления

Базовый класс *Медиатор* реализованный в PureMVC принимает в качестве аргументов своего конструктора имя и объект *Представления* типа Object.

Конструктор вашего конкретного *Медиатора* будет принимать компонент *Представления*, делая его сразу же доступным в защищенном (protected) свойстве класса viewComponent, обычно типа Object.

Вы также можете, используя метод setViewComponent, динамически устанавливать экземпляр *Представления* в *Медиаторе* после вызова конструктора.

После присвоения вы часто будете приводить этот объект к конкретному типу, что может быть неудобно, а также способствовать распространению повторяемого кода.

Медиаторы

Неявное преобразование типов компонентов Представления

Язык ActionScript предоставляет возможность называемую геттеры и сеттеры (getters/setters). Геттер выглядит как метод, но используется как свойство класса. Эта возможность очень полезна для решения проблемы частого приведения типа.

Полезная идиома применяемая в вашем конкретном Медиаторе, - использование геттера для приведения типа Представления к его настоящему типу с понятным именем.

Например так:

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

Затем, в любом месте вашего *Медиатора* чем делать это так:

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

Мы *вместо* этого делаем так:

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

Взаимодействие с Представлением

Медиатор обычно имеет только один компонент *Представления*, но может взаимодействовать и с несколькими, например: с ApplicationToolBar и содержащимися в нем кнопками, либо другими аналогичными компонентами. Мы можем иметь группу связанных компонентов (как форма) в одном *Представлении* и обращаться *Медиатором* к её элементам как к свойствам. Но лучше инкапсулировать в *Представлении* как можно больше компонентов. Иметь специализированный объект для обмена данными тоже хорошая практика.

Медиаторы

Взаимодействие с Представлением

Медиатор отвечает за взаимодействие уровня *Контроллера* и *Модели*, обновляя *Представление* когда получает соответствующие *Оповещения* (Notifications).

Во Flash, мы обычно подписываем слушателей событий к компоненту *Представления*, в момент создания *Медиатора* или по вызову метода setViewComponent, определяя метод обработчика:

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

Действия *Медиатора,* в ответ на возникшее *Событие,* определяется требованиями логики.

Обычно, конкретный метод обработчика События Медиатора выполняет такие действия:

- Изучает тип События либо поля специализированного типа *События*, который ожидается.
- Читает или модифицирует доступные свойства (либо вызывает методы) компоненты *Представления*.
- Читает или модифицирует доступные свойства (либо вызывает методы) *Прокси*.
- Шлет одно или более *Оповещение*, на которое будут реагировать *Медиаторы* и *Команды* (или даже тот же самый *Медиатор*).

Медиаторы

Взаимодействие с Представлением

Несколько хороших правил:

- Если несколько других *Медиаторов* должно быть вовлечено в обработку ответа на *Событие*, изменяйте общие *Прокси* либо отсылайте *Оповещения*, которые обработают соответствующие *Медиаторы*.
- Если требуется большое количество согласованных взаимодействий, хорошая практика использовать *Команду*, локализируя шаги в одном месте.
- Плохой практикой считается получение других *Медиаторов*, либо *Медиаторов* с открытыми методами для непосредственной манипуляции.
- Для манипулирования и распространения информации о состоянии приложения к *Медиаторам*, устанавливайте значения свойств *Медиатора*, или вызывайте методы *Прокси*, созданные для установки состояния. Пусть медиатор будет заинтересован в оповещениях, отправляемых *Прокси*, хранящих состояние приложения.

Обработка оповещений в Медиаторе

Вместо явного добавления слушателей событий для *Представления*, можно очень просто и почти автоматически связать между собой *Медиатор* и *PureMVC*.

После регистрации *Представления*, *Медиатор* опрашивается на наличие заинтересованных оповещений и в ответ возвращает массив имен *Оповещений* (Notifications name), которые он хотел бы обрабатывать.

Медиаторы

Обработка оповещений в Медиаторе

Ответ проще всего реализовать с помощью одного простого выражения, которое будет возвращать массив с именами оповещений, которые должны быть определены как статические константы, обычное это делается в Фасаде приложения.

Создать список заинтересованных *Оповещений* (Notification Interests) для конкретного *Медиатора* можно следующим образом:

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

Перечисленные оповещения, будут сразу обрабатываться *Медиатором* как только они будет отосланы любым из *игроков* системы, даже если они отосланы самим *Медиатором*.

Внутри обработчика оповещений, для удобства и читабельности кода вместо «if / else if» лучше использовать конструкцию «switch / case».

По сути, для каждого оповещения достаточно небольшого обработчика, а вся нужная информация должна находиться в самом объекте оповещения. Правда иногда информация может быть получена от Посредника, основанная на данных из объекта оповещения. Желательно избегать нагромождения логики в одном обработчике, иначе это верный признак того, что вы стараетесь перенести бизнес логику из Команды (Command) в обработчик оповещений (notification) медиатора.

Медиаторы

Обработка оповещений в Медиаторе

```
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;
                }
        }
}
```

Практика показывает, что лучше всего обрабатывать 4-5 оповещений в одном обработчике.

Если оповещений больше, желательно разбить этот обработчик на несколько менее громоздких. Создавая медиаторы для отдельных компонентов *Представления*, можно избежать накопления оповещений в одном обработчике.

Использование единого, заранее определенного метода оповещения, является главной отличительной чертой *Медиатора,* как он обрабатывает события и оповещения.

Для событий мы можем определить целый ряд обработчиков, но обычно для каждого события есть свой единственный обработчик. Эти методы не должны быть слишком сложными, или заниматься слишком мелкими задачами по работе с каждым компонентом *Представления*, поскольку *Представление* должен быть написан так, чтобы включать в себя детали своей реализации, предоставляя *Медиатору* четко разделенный по направлениям интерфейс.

Медиаторы

Обработка оповещений в Медиаторе

Используя *Оповещения*, у вас есть единственный метод-обработчик, в котором вы обрабатываете все *Оповещения* нужные для *Медиатора*.

Лучше всего обрабатывать все оповещения в одном методе, используя switch конструкцию для различия оповещения по именам.

Было много обсуждений по поводу использования switch конструкции, многие разработчики считают, что эта конструкция ограничивает одним способом обработки. Однако рекомендуемая практика это использования одного метода обработчика оповещения и switch-конструкция.

Медиатор является посредником между *Представлением* и остальной частью системы.

Рассмотрим роль переводчицы беседы между послом и остальными членами конференции ООН. Она редко делает что-либо большее чем перевод и получение сообщений, разве что иногда подбирая подходящие метафоры или факты. Тоже самое можно сказать о роли *Медиатора* в рамках PureMVC.

Связка Медиатора с Прокси и другими Медиаторами

В конечном счете, *Представление* наполняется данными *Модели* для графического их отображения, взаимодействия с данными через *Прокси.*

Представление должно знать о *Модели*, но *Модель*, в свою очередь, не должна знать о *Представлении*.

Медиаторы

Связка Медиатора с Прокси и другими Медиаторами

Медиатор может легко получить доступ к *Прокси* через *Модель* и читать и манипулировать данными, используя программный интерфейс *Прокси*. Если выполнять те же операции с помощью *Команд*, будет потеряна связь между *Моделью* и *Представлением*.

Так же *Медиатор* может получить ссылки на другие *Медиаторы* из *Представления* и манипулировать данным уже из других *Медиаторов*.

Однако это не очень хорошая практика, потому что может привести к зависимости между частями *Представления*, что негативно повлияет на рефакторинг одной части *Представления* без затрагивания других частей.

Медиатор, желающий взаимодействовать с другим областям Представления, должен послать Оповещение, а не манипулировать Представлением напрямую.

Медиатор не должен раскрывать методы для манипулирования своими компонентами - вместо этого он должен отвечать на оповещения и выполнять свою работу.

Если большая часть поведения *Компонентов Представления* реализована в *Медиаторе* (в ответ на *Событие* или *Оповещение*), желательно часть из них перенести в само *Представление*, так чтобы в дальнейшем можно было повторно использовать.

Если большинство взаимодействий с *Прокси* или их данными реализованы в *Медиаторе*, перенос этой функциональности в *Команду* упростит *Медиатор.* Также перенос бизнес логики в *Команды* позволяет использовать ее другими *Представлениями* без потери связки *Представление – Модель*.

Медиаторы

Взаимодействие пользователя с Представлением и Медиаторами

Рассмотрим компонент LoginPanel, представляющий собой форму. Этому компоненту будет соответствовать *Медиатор* LoginPanelMediator, который будет взаимодействовать с нашей формой и отвечать на попытку войти в систему.

Взаимодействие компонента LoginPanel и *Медиатора* LoginPanelMediator заключается в том, что компонент отсылает событие TRY_LOGIN, когда пользователь ввел логин/пароль и нажал на кнопку Login. *Медиатор* LoginPanelMediator обрабатывает событие, отсылая *Оповещение* с введенными данными (логином и паролем сохраненный в объекте LoginVO).

LoginPanel.mxml:

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

<!—
Событие вещаемое компонентом.
К сожалению мы не можем использовать имя константы, так как MetaData является директивой компилятора
-->
<mx:MetaData>
    [Event('tryLogin')];
</mx:MetaData>


<mx:Script>
<![CDATA[
    import com.me.myapp.model.vo.LoginVO;
    // поля формы привязаны к свойствам объекта
    [Bindable] public var loginVO:LoginVO = new LoginVO();
    [Bindable] public var loginStatus:String = NOT_LOGGED_IN;

    // задаем константы в Представлении для событий
    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>
```

Медиаторы

Взаимодействие пользователя с Представлением и Медиаторами

```
<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>
```

Компонент LoginPanel заполняет объект LoginVO данным введенные пользователем и, когда он нажимает кнопку «Login», отправляется событие TRY_LOGIN. Затем за дело берется *Медиатор* LoginPanelMediator.

Это позволяет компоненту только собирать данные и оповестить систему о готовности.

Для того чтобы избежать случаев отправки пустых значений в полях объекта LoginVO, можно активировать кнопку «Login» только в случае если поля логин и пароль заполнены.

Вся логика компонента находится в *Медиаторе*, где обрабатывается событие TRY_LOGIN, проверяются значения полей LoginVO и устанавливается значение свойства status компонента LoginPanel.

Медиатор LoginPanelMediator также будет отвечать за обработку оповещений LOGIN_FAILED и LOGIN_SUCCESS и будет выставлять значение свойства status.

Медиаторы

Взаимодействие пользователя с Представлением и Медиаторами

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;

	// Медиатор взаимодействующий с Компонентом LoginPanel
	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 );
		}

		// Список подписанных Оповещений
		override public function listNotificationInterests( ) : Array  {
			return [
				LoginProxy.LOGIN_FAILED,
				LoginProxy.LOGIN_SUCCESS
			];
		}

		// Обработка Оповещений
		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;
```

Медиаторы

Взаимодействие пользователя с Представлением и Медиаторами

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

                // Пользователь пытается войти кликая кнопку Login
                private function onTryLogin ( event:Event ) : void {
                    sendNotification( ApplicationFacade.LOGIN, loginPanel.loginVO );
                }

                // Приведение типа Представление к конкретному типу
                protected function get loginPanel() : LoginPanel {
                    return viewComponent as LoginPanel;
                }
        }
}
```

Следует отметить, что *Медиатор* LoginPanelMediator в своем конструкторе, связывает событие TRY_LOGIN и метод onTryLogin, который будет обрабатывать это событие. Таким образом, метод onTryLogin будет вызван, когда пользователь кликнет на кнопку «Login». В самом же методе onTryLogin отсылается *Оповещение* вместе с заполненным объектом LoginVO.

Ранее мы зарегистрировали команду LoginCommand для этого *Оповещения*. Эта команда будет вызывать метод login *Прокси* LoginProxy, передавая объект LoginVO. *Прокси* LoginProxy попытается авторизоваться с помощью удаленного сервиса (remote service) и отослать оповещения LOGIN_SUCCESS или LOGIN_FAILED. Эти классы определены в конце раздела, посвященного *Прокси*.

Медиатор LoginPanelMediator имеет список заинтересованных оповещений, в который входят LOGIN_SUCCESS и LOGIN_FAILED. Поэтому, независимо от результата, *Медиатор* будет оповещен и присвоит значение LOGGED_IN переменной loginStatus (компонент LoginPanel) в случаи успеха, иначе - NOT_LOGGED_IN и очистит объект LoginVO.

Прокси

Вообще говоря, паттерн *Прокси (Proxy, далее Прокси)* используется для обеспечения хранения любого объекта в целях контроля доступа к нему. В приложениях, основанных на PureMVC, класс *Proxy* используется специально для управления частями *Модели Данных* приложения.

Прокси могут управлять доступом к созданным локально структурам данных произвольной сложности. Они называются *Объектами Данных Прокси (Proxy's Data Object)*.

В этом случае, идиома для взаимодействия с *Прокси* связана с синхронной установкой (setting) и получением (getting) данных. *Объекты Данных* могут подвергаться полному или частичному обновлению свойств, или же может меняться ссылка на сам *Объект Данных*. Когда выполняются методы для обновления данных, *Прокси* может отправлять *Оповещение* в остальную часть системы, что его данные изменились.

Прокси Удаленного Сервиса (Remote Proxy, далее Прокси Удаленного Сервиса) используется для инкапсуляции взаимодействия с удаленными сервисами, для сохранения или получения данных. *Прокси* может хранить объект, который взаимодействует с удаленным сервисом, а также контролировать доступ к данным, переданным и полученным от этого сервиса.

Таким образом, можно просто устанавливать данные или же вызывать методы *Прокси* и дожидаться асинхронного *Оповещения*, отправляемого *Прокси* когда данные от удаленного сервиса получены.

Обязанности конкретного Прокси

Конкретные *Прокси* позволяют нам инкапсулировать фрагменты модели данных, независимо откуда они получены и их типа, а так же необходимы для управления *Объектами Данных* и организации доступа приложения к ним.

Реализация класса Прокси, который входит в PureMVC, является простым контейнером для *Объектов Данных*, и может быть зарегистрирована в Модели.

Прокси

Обязанность конкретного прокси

Чтобы полностью использовать возможности в данной форме, мы, обычно, создаем подкласс класса Proxy и добавляем функциональность, характерную для конкретного *Прокси*.

Общие варианты использования Прокси включают в себя:

- *Прокси Удаленного Сервиса* - данные, управляемые конкретным *Прокси,* находится на удаленном сервере, и могут быть доступны через удаленный сервис.
- *Прокси и Делегат (Proxy and Delegate)* - доступ к объекту, обеспечивающему удаленный доступ, должен быть распределен между несколькими *Прокси*. Класс *Делегат* поддерживает объект удаленного доступа и контролирует доступ к нему, обеспечивая передачу ответов тем, кто послал запрос.
- *Защищенный Прокси (Protection Proxy)* - используется когда игроки системы должны иметь разные права доступа к *Объекту Данных*.
- *Виртуальный Прокси* - создает *Объекты Данных* по запросу.
- *Умный Прокси* - загружает объекты данных в память при первом доступе, осуществляет ведение учета ссылок, позволяет блокировать объект, чтобы другой объект не смог изменить его.

Неявное приведения типа объекта данных

Реализация базового класса *Proxy*, которая поставляется с PureMVC принимает имя *Прокси* и основной *Объект Данных* в качестве аргумента конструктора. Вы можете динамически устанавливать *Объект Данных Прокси* после того, как он создан, вызывая метод setData.

Прокси

Неявное приведения типа объекта данных

Как и в случае с *Медиатором* и его Компонентом *Представления*, вы будете часто приводить *Объект Данных* к фактическому типу, для того, чтобы получить доступ к свойствам и методам, которые он предоставляет; это утомительная и однообразная практика, но следование идиомам позволяют открывать больше о реализации *Объекта Данных,* чем это может потребоваться.

Кроме того, поскольку *Объект Данных* обычно имеет сложную структуру, часто необходимо иметь под рукой ссылки на несколько частей структуры, в дополнении к приведенной к типу ссылки на всю структуру.

В языке ActionScript функции называемые неявными геттерами (getters) и сеттерами (setter) оказываются очень полезными в решении приведения типа и устранении проблем неуместного применения типов.

Полезная практика - использовать в конкретном Прокси неявный getter, который возвращает Объект Данных фактического типа и имеет осмысленное название.

Дополнительно, вы можете указать несколько геттеров, возвращающих определенные части *Объекта Данных*.

Например:

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

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

Прокси

Неявное приведения типа объекта данных

В каком-то Медиаторе вам пришлось бы делать так:

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

Однако, используя практику, описанную выше, можно сделать так:

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

Запрет на привязку к медиаторам

У *Прокси* не запрашивают список интересующих его *Оповещений* как у *Медиатора*, да он и не получает *Оповещений*, потому что он не должен заботиться о состоянии *Вида*. Вместо этого Прокси предоставляет методы и свойства, позволяющие другим участникам манипулировать им.

Конкретный Прокси не должен извлекать и использовать Медиаторы для того, чтобы информировать систему об изменении своего Объекта Данных.

Вместо этого *Прокси* должен отправлять *Оповещения*, которые получат *Команды* или *Медиаторы*. От *Прокси* не должно зависеть, как эти *Оповещения* повлияют на систему.

В связи с тем, что уровень *Модели Данных* не содержит каких-либо знаний о системе реализации, уровни *Представления* и *Контроллера* могут быть отрефакторены без вмешательства в уровень *Модели Данных*.

Обратное не совсем верно. Очень трудно изменить уровень *Модели Данных*, не изменяя уровень *Представления* и, скорее всего, уровень *Контроллера*. В конце концов, эти уровни существуют только чтобы позволить пользователю взаимодействовать с уровнем *Модели Данных*.

Прокси

Инкапсуляция предметной области в прокси

Изменения на уровне *Модели Данных* почти всегда приводит к некоторому рефакторингу уровней *Контроллера* и *Представления*.

Мы увеличили разделение между уровнем Модели Данных и общих интересов уровней Представления и Контроллера путем максимального перемещения предметной области в Прокси.

Прокси может использоваться не только для контроля доступа к данным, но и выполнять операции над данными, которые могут быть необходимы для поддержания некоторого их валидного состояния.

Например, расчет налога с продаж является функцией *предметной области* и поэтому она должна находится в *Прокси*, а не в *Медиаторе* или *Команде*.

Несмотря на то, что эта функция может быть реализована в любом из этих мест, размещение её в *Прокси* не только логично, но и облегчает другие уровни и упрощает рефакторинг.

Медиатор может получить П*рокси*; вызвать функцию вычисления налога с продаж, и возможно, поместить результат в какую-либо форму. Но размещение фактического расчета в *Медиаторе* реализует в нем, с точки зрения уровня, *Предметную Область*. Вычислений налога является правилом относящемся к П*редметной Области*. *Представлению* оно может быть известно как свойство *Предметной Области*, доступное, если соответствующий Объект Данных присутствует.

Представьте, что вы в настоящее время работаете над RIA приложением для запуска в размерах рабочего стола персонального компьютера. Новая версия должна запускаться на КПК с соответствующим разрешением с сокращением *сценариев использования*, но по-прежнему использовать полную *Модель Данных* существующего приложения.

Прокси

Инкапсуляция предметной области в прокси

При правильном разделении интересов, мы можем использовать уровень *Модели Данных* во всей ее полноте и просто подгонять новые уровни *Представления* и *Контролера* к нему.

Размещение фактического расчета налога с продаж в *Медиаторе* может показаться эффективным и простым в момент реализации; например, вы только что получили данные из формы, и вы хотите рассчитать налог с продаж, и отправить его в *Модель* уже вычисленным.

Однако в каждой версии своего приложения вам теперь придется дублировать ваши усилия или копировать код вычисления налога с продаж в новый, совершенно другой уровень *Представления*, хотя эта логика могла бы появляться автоматически, как часть вашей *Модели Данных*.

Взаимодействие с *Прокси Удаленного Сервиса*

Прокси Удаленного Сервиса - это обычный *Прокси*, который получает свои *Объекты Данных* из удаленного местоположения. Это значит, что мы взаимодействуем с ним в асинхронном режиме.

Каким образом *Прокси* получает данные - зависят от платформы клиента, реализации удаленного взаимодействия, а также предпочтений разработчиков. В Flash / Flex мы можем использовать HTTPService, WebService, RemoteObject, DataService или XMLSocket для осуществления запросов из *Прокси*.

Прокси

Взаимодействие с *Прокси Удаленного Сервиса*

В зависимости от потребностей, удаленный *Прокси* может направлять запросы динамически, в ответ на вызов сеттеров или методов; или же может делать единственный запрос во время создания и обеспечивать доступ к данным впоследствии.

Есть ряд оптимизаций, которые могут быть применены в *Прокси* для повышения эффективности взаимодействия с удаленной службой.

Прокси может кешировать данные, и, таким образом, сократить количество запросов в сеть, или же отсылать обновления только тех частей структуры данных, которые были изменены, уменьшая сетевой трафик.

Если запрос динамически вызывается на *Прокси Удаленного Сервиса* другим игроком в системе, *Прокси* необходимо отправить *Оповещение,* когда будет получен ответ.

Заинтересованным в получении *Оповещения* может быть или не быть тот же игрок, что и инициировал запрос.

Например, для осуществления процесса поиска, происходящего на удаленном сервере и отображения результатов, возможна следующая последовательность действия:

- *Представление* инициирует поиск вызовом события (Event).
- Его *Медиатор* получает соответствующий удаленный прокси и устанавливает свойство "критерии поиска".
- Свойство "критерий поиска" в Прокси - это на самом деле неявный сеттер, который сохраняет значение и инициирует поиск запросом через внутренний HTTPService, у которого ожидает событий Result или Fault.

Прокси

Взаимодействие с *Прокси Удаленного Сервиса*

- После того, как HTTPService получает ответ, он вызывает ResultEvent, который получает *Прокси* чтобы установить результаты поиска, как свое публичное свойство.
- Затем *Прокси* отправляет *Оповещение*, информирующее об успехе поиска и содержащее ссылку на свой *Объект Данных* в качестве тела *Оповещения*.
- Другой *Медиатор*, выразивший заинтересованность в этом *Оповещении*, и, соответственно, принимающий его, устанавливает тело *Оповещения* как dataProvider своего *Компонента Представления*.

Или же рассмотрим LoginProxy, который содержит LoginVO (Value Object; простой класс-контейнер данных). LoginVO может выглядеть так:

```
package com.me.myapp.model.vo {

	// Привязка модели к «серверному классу»
	 [RemoteClass(alias="com.me.myapp.model.vo.LoginVO")]

	[Bindable]
	public class LoginVO {
		public var username: String;
		public var password: String;
		public var authToken: String; // устанавливается сервером если авторизация пройдена
	}
}
```

LoginProxy содержит методы для установления полномочий, входа и выхода пользователя, и получения метки авторизации, которая будет включена в последующие удаленные вызовы для опознания пользователя в данной конкретной схеме аутентификации.

Прокси

Взаимодействие с *Прокси Удаленного Сервиса* 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;

	// Прокси для авторизации пользователя
	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 );
		}

		// Приведение типа объекта к конкретному классу
		public function get loginVO( ) : LoginVO {
			return data as LoginVO;
		}

		// Пользователь авторизирован если authToken установлен
		public function get loggedIn():Boolean {
			return ( authToken != null );
		}

		// подвызов, возвращающий строку авторизации после логина
		public function get authToken():String {
			return loginVO.authToken;
		}
```

Прокси

Взаимодействие с Прокси Удаленного Сервиса

```
		// Установка имени/пароля и вход, либо выход и еще одна попытка
		public login( tryLogin:LoginVO ) : void {
			if ( ! loggedIn ) {
				loginVO.username= tryLogin.username;
				loginVO.password = tryLogin.password;
			} else {
				logout();
				login( tryLogin );
			}
		}

		// для выхода используется пустой объект LoginVO
		public function logout( ) : void {
			if ( loggedIn ) loginVO = new LoginVO( );
			sendNotification( LOGGED_OUT );
		}

		// Оповещение системы о том что вход прошел успешно
		private function onResult( event:ResultEvent ) : void  {
			setData( event.result ); // immediately available as loginVO
			sendNotification( LOGIN_SUCCESS, authToken );
		}

		// Оповещение системы о том что вход завершился неудачей
		private function onFault( event:FaultEvent) : void  {
			sendNotification( LOGIN_FAILED, event.fault.faultString );
		}
	}
}
```

LoginCommand может получить LoginProxy, установить полномочия, а также вызвать метод входа пользователя, инициировав удаленный запрос.

GetPrefsCommand может ответить на уведомление LOGIN_SUCCESS, получить authToken тела уведомления и вызвать следующий запрос, который вернет какие-то свойства или настройки пользователя.

Прокси

Взаимодействие с Прокси Удаленного Сервиса

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 );
		}
	}
}
```

AUTHOR: Cliff Hall <cliff@puremvc.org>