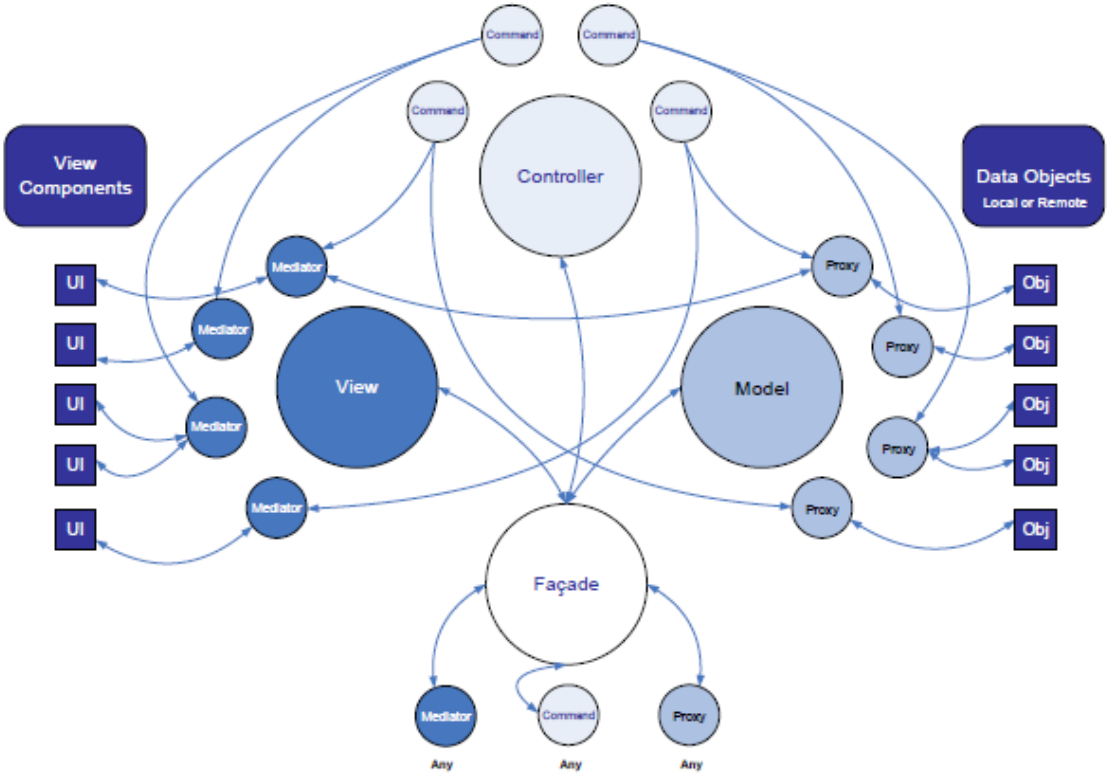
Implementation Idioms and Best Practices

ADOBE FORMS EDITION

Building Robust, Scalable and Maintainable Client Applications using PureMVC with Examples in **JavaScript for Adobe Forms**



AUTHOR: Cliff Hall cliff@puremvc.org
LAST MODIFIED for Adobe Forms: 10/22/2012
Adobe Forms Mods by: Carman Lawrick <Carman@lawrick.com>

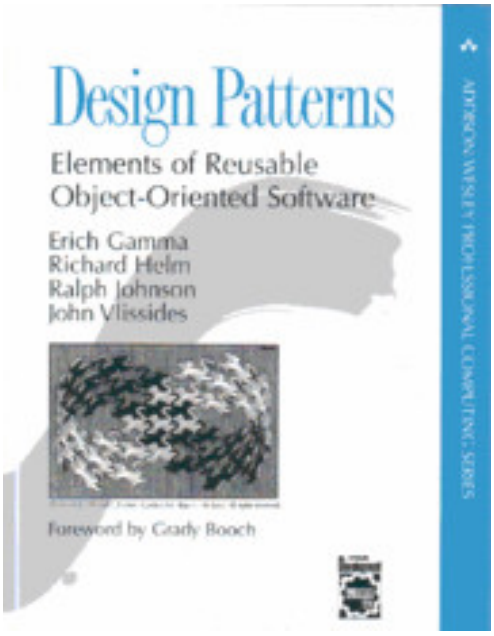
puremvc framework
IMPLEMENTATION IDIOMS AND BEST PRACTICES

Note to the Adobe Forms Edition

This document was originally written by Cliff Hall for Flex (Flash) users with examples in ActionScript. It was modified throughout to speak to creators of Adobe LiveCycle Forms.

Inspiration

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



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

PureMVC Gestalt

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

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

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

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

Since JavaScript does not currently allow for creation of true Object Oriented Classes, the Core actors are implemented in a JavaScript environment using pseudo-classes or "classlets" based on the JavaScript Function/Prototype facilities. In this implantation, they extend the xfa (eXtensible Forms Architecture, defined by the W3C as a standard for on-line forms) object and become part of the Adobe Forms Scripting core. Part of the PureMVC library allows you to create your own classlets and further extend the xfa object, so that all of your form scripting can be object oriented.

Model & Proxies

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

This results in portable Model tier code. You can change the data Model without necessarily breaking the rest of the form scripting.

View & Mediators

The View primarily caches named references to Mediators. Mediator code stewards View Components (forms design components), listening to form-events, sending and receiving notifications to and from the rest of the system on their behalf and directly manipulating their state.

This separates the View definition from the logic that controls it. You can change the visual form design without necessarily breaking the scripting that controls it.

Controller & Commands

The Controller maintains named mappings to Command classes, which are stateless, and only created when needed. Commands may retrieve and interact with Proxies, send Notifications, execute other Commands, and are often used to orchestrate complex or system-wide activities such as application startup and shutdown. They are the home of your application's Business Logic.

Facade & Core

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

By extending the Facade, your application gets all the benefits of Core actors without having to import and work with them directly.

You will implement a concrete Facade for your application only once and it is simply done.

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

Observers & Notifications

PureMVC applications may run in environments without access to Acrobat/Reader Form-Events, so the framework implements an Observer notification scheme for communication between the Core MVC actors and other parts of the system in a loosely-coupled way.

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

Notifications Can Be Used to Trigger Command Execution

Commands are mapped to Notification names in your concrete Facade, and are

automatically executed by the Controller when their mapped Notifications are sent. Commands typically orchestrate complex interaction between the interests of the View and Model while knowing as little about each as possible, so that they can be changed without affecting each-other.

Mediators Send, Declare Interest In and Receive Notifications

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

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

Additionally, in the Adobe Forms environment, some Mediators are created with reference to a form design object that they will manage and steward. When created in this way, the Mediator, managing a form design object, automatically receives *private* notification of form-events fired by the form design object. The managing Mediator can then do whatever is necessary to react to the form-event, if anything, including sending notifications to other Mediators or Commands, or manipulating data through a Proxy.

Proxies Send, But Do Not Receive Notifications

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

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

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

However View and Controller tiers should be able to vary without affecting the data Model tier, so they are not required to (cannot) send notifications to Proxies.

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

View/Controller arrangements operating against the same Model.

Facade

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

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

Typically, the framework Facade will be sub-classed in your application and used to initialize the Controller with Command mappings.

Preparation of the Model and View are then orchestrated by Commands executed by the Controller.

What is a Concrete Facade?

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

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

This *concrete* Facade is then used to access and notify the Commands, Mediators and Proxies that do the actual work of the system. By convention, it is named 'ApplicationFacade', but you may call it whatever you like. When you subclass the framework Facade, you make it part of the xfa object, globally referenced throughout your form scripting.

Generally, your application's View hierarchy (display components) will be created by whatever process your platform normally employs. In LiveCycle, Acrobat or Reader, Adobe Designer creates a form that automatically instantiates all its form design objects. Once the application's View hierarchy has been built, the facade is used to get the PureMVC apparatus started and the Model and View regions are prepared for use, assigning Mediators to manage the form design objects and their form-events.

Your concrete Facade is also used to facilitate the startup process in a way that keeps the form code from knowing much about the PureMVC apparatus to which it will be connected. The form scripting merely attaches your concrete Facade's Singleton instance to the xfa object, making the PureMVC Framework globally available to all of your scripting.

Creating a Concrete Facade for Your Application

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

The MVC_APPLICATIONFACADE Script Object:

```
// make sure the classlet definition runs only once
var evaluatejso;(function(scope){if (null==scope)scope=xfa;if
(xfa.facadedef){return;}xfa.puremvc.define({name: 'facadedef',scope: scope});
xfa.puremvc.define// defining a JavaScript classlet
(
    // CLASS INFO
    {
        name: 'ApplicationFacade',
        scope: scope,
        parent: xfa.puremvc.Facade
    },

    // INSTANCE MEMBERS
    {
        /**
         * A convenience method to start the PureMVC apparatus
         */
        startup: function()
        {
            if (!this.initialized)
            {
                xfa.mvcLogger.trace("starting ApplicationFacade");
                xfa.mvcLogger.trace("xfa.appconstants.STARTUP: \n" +
                    xfa.appconstants.STARTUP);
                this.initialized=true;

                // associate the SetupCommand with the STARTUP notification
                this.registerCommand( xfa.appconstants.STARTUP,
                    xfa.controller.command.StartupCommand );

                // issue the SETUP notification to execute StartupCommand
                xfa.mvcLogger.trace("sending STARTUP notification to \
                    command.StartupCommand");
                this.sendNotification( xfa.appconstants.STARTUP );
```

```
            } else {
                xfa.mvcLogger.warn("Startup called on ApplicationFacade that \
                    is already initialized!");
            }

        }
    },

    // STATIC MEMBERS
    {
        /**
         * Retrieve an instance of ApplicationFacade. If one has not yet been
         * instantiated, one will be created for you.
         *
         */
        getInstance: function(multitonKey)
        {
        // all Facade instances, including Facade subclass instances, are stored
        // on Facade.instanceMap. When implementing you own #getInstance factory
        // method, ensure that follow the general pattern implemented here or else
        // puremvc.Facade#hasCore and puremvc.Facade#removeCore will not work if
        // you ever need to use them.
            var instanceMap=xfa.puremvc.Facade.instanceMap;
            instance=instanceMap[multitonKey]; // read from the instance map

            if (instance) // if there is an instance...
                return instance; // return it

            // otherwise create a new instance and cache it on Facade.instanceMap;

            return instanceMap[multitonKey]=new
                    xfa.ApplicationFacade(multitonKey);
        },
        NAME: 'formname'
    }
);
})(xfa);
```

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

- It is wrapped in run-once code, to which the xfa object Is provided as “scope,” meaning that the classlet definition becomes attached to the xfa object
- It extends the PureMVC Facade class, which in turn implements the IFacade interface.

- It does not override the constructor. If it did, the class definition would become invalid.
- It defines a static getInstance method that returns the Singleton instance, creating and caching it if need be.

  The reference to the instance is kept in a protected property of the super class (Facade).
- It does not define constants for Notification names. These are defined as static constants in a similarly defined classlet, attached to the xfa object, in the MVC_CONSTANTS Script Object. A separate Script Object is used to define all constants for code organization purposes. Because constants are attached to the xfa object, they are globally accessible throughout your form scripting.
- It initializes the Controller with a StartupCommand that will be executed when corresponding Notification is sent.
- It provides a startup method which sends a Notification of type STARTUP, registered to the StartupCommand mentioned in the previous bullet. So running this startup metiod invokes the StartupCommand, which starts the whole PureMVC framework, registering other Commands and Mediators and Proxies as well.

With these simple implementation requirements, your concrete Facade will inherit quite a bit of functionality from the abstract super class, making it all globally available to all of your form scripting as an attachment to the xfa object.

Initializing your Concrete Facade

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

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

So, where and how does the Facade fit into the scheme of things in an actual application? Consider the following Form-level *Initialize Form-Event* Script:

```
SCRIPTS.my_init()
```

And this in the SCRIPTS Script Object:

```
function my_init(){
    if ((!xfa.initdone) || (xfa.initdone == null) || (xfa.initdone.length==0)){//
run only once per form session.
        xfa.initdone="x";
        // force evaluation of Script Objects to define MVC classes - this order
of sequence of definitions is important
    MVC_PUREMVC.evaluatejso=null;OO_UTIL.evaluatejso=null;LOG4JS.evaluatejso=
null;MVC_CONSTANTS.evaluatejso=null;MVC_PROXIES.evaluatejso=null;MVC_MEDIATOR
S.evaluatejso=null;MVC_COMMANDS.evaluatejso=null;MVC_APPLICATIONFACADE.evalua
tejso=null;MY_CLASSLETS.evaluatejso=null;
        //xfa.mvcException = new xfa.Log(xfa.Log.FATAL, xfa.Log.consoleLogger,
"Exception Handler"); //uncomment this and change it if you don't want the default
implementation of console:FATAL
        //xfa.mvcLogger = new xfa.Log(xfa.Log.WARN, xfa.Log.consoleLogger,
"pureMVC");//uncomment this and change it if you don't want the default
implementation of console:WARN
        //xfa.appLogger = new xfa.Log(xfa.Log.WARN, xfa.Log.consoleLogger,
"APP");//uncomment this and change it if you don't want the default implementation
of console:WARN
        //instantiate facade
        xfa.facade =
xfa.ApplicationFacade.getInstance(xfa.ApplicationFacade.name); //!!! cjl: store
an instance of facade globally, all actors can access the facade as
xfa.ApplicationFacade
        xfa.mvcLogger.trace("my_init calling xfa.facade startup");
        xfa.facade.startup();//startup the MVC framework
        xfa.mvcLogger.trace("my_int xfa.facade startup complete");
    }
   }
```

That's it. Pretty simple.

Build make the JavaScript Interpreter run through all of the Script Objects, get the ApplicationFacade instance and invoke its startup method.

Key things to notice about this example are:

- We wrap the code in this function in run-once closure
- We reference an arbitrary item in each Script Object to force the Interpreter to evaluate the Script Object, defining classes, which are designed to attach themselves to the xfa object.

- A script block is used to declare and initialize a global variable (on xfa) with the Singleton instance of the concrete ApplicationFacade.
- Since we are initializing the variable with a call to the static ApplicationFacade.getInstance method, this means that by the time the form's *initialize* form-event finishes, the Facade will have been created and along with it.
- In this *initialize* handler of form, we invoke the startup method, which sends a Notification to the startupCommand, which creates and registers other Commands and Mediators and Proxies. Now the entire PureMVC framework is running.
- Notice that three log4js logger objects are created, and here set to WARN level logging. Change this to TRACE level logging if you want to see the framework startup in your Acrobat/Reader console, or on the LiveCycle server, the system log.

Note that ordinary View Components have no need to know or interact with the Facade, but this top-level *initialize* form-event is the exception to the rule.

In summary, this top-level *initialize* form-event initializes the Facade then starts up the PureMVC apparatus.

Notifications

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

The ActionScript language does not provide the Events model that is used in Flex and Flash, those come from the Flash package. Likewise, JavaScript in Adobe Forms does not provide the Events model that is used in the Adobe Forms environment, they come from the Adobe Forms platform. The framework has been ported to other platforms such as JavaScript, Perl, Python, PHP, Objective C, ColdFusion, C++, Flex and Flash, C#,J2ME and others, because the framework manages its own internal communications rather than relying on those provided by any platform.

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

Form-Events vs. Notifications

Form-Events are dispatched from form design objects. In the Adobe Forms implementation, the form-events are passed privately as Notifications to Mediators that are declared as managers or stewards of the form design objects.

This happens because in the Adobe Forms implementation of PureMVC, every form-event handler (the ones usually scripted by Adobe Forms Designers) is "propagated" from the highest form object (usually called form1) through all design elements on the form. These propagated form-event handlers all call a function that searches for Mediators that have been declared as managers of the form design object that generates the form-event. Upon finding one or more such Mediator, the function directly (privately) calls that Mediator's *handleNotification* method, passing it a normal Notification object.

So, all you need do to handle an form-event in PureMVC is create a Mediator that declares itself manager of a form design object, and place a handler for the desired form-event in the Mediator's handleNotification method. (To keep communication between a form design component and its stewarding Mediator private and hidden, explicit strings are used for Notification names, instead of global constants. Copy/Paste these explicit strings into your Mediator from the bottom of the MVC_MEDIATORS Script Object.) This Mediator could send Notifications of this form-event to other Mediators or Commands if desired.

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

Notifications may have an optional 'body', which can be any ActionScript object. (When a Mediator is declared as managing a form design object, and it receives private Notification of a form-event on its managed form design object, the 'body' is the form design object being managed. So your Notification handler can use this 'body' to examine the content or state or methods of the form design object.)

It is rarely necessary to create a custom Notification class for each form design object or each special purpose, since it can carry a payload (the 'body') 'out of the box'. You can of course create custom Notification classes for special purposes but JavaScript does not do type checking so you can put anything in the provided Notification class at all, making the overhead of maintaining many Notification classes a question of programming style.

Notifications also have an optional 'type' that can be used by the Notification recipient as a discriminator. This 'type' can be a simple string or constant, or a complex object, just as the 'body' of the Notification can be.

For instance, in a document editor application, there may be a Proxy instance for each document text field that is opened and a corresponding Mediator for the text field used to edit the document. They might all share the same Notification message, and provide the text field as the 'body', but the Proxy and Mediator might share a unique key that the Proxy passes as the Notification type, identifying which document the Notification applies to.

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

Defining Notification and Form-Event Constants

In the Adobe Forms implementation of PureMVC, we have set constants into their own Script Object, which attaches them to the xfa object, making them globally available to all scripting throughout the form. (The constants are static properties of the xfa.appconstants class)

Centralized, globally available definition of constants for Notification names ensures that when one of the notification participants needs to refer to a Notification name, we can do so in a safe way, as opposed to using literal strings which could be misspelled, but not cause an error.

As a best practice, it is advised that you do not, however, define the names of *Form-Events* in the global constants class. *It is advised that you define Event name constants statically on the boundary classes that generate them (in the form design components themselves), or in custom Event classes that are dispatched.* Since in the Adobe Forms implementation we can do neither, the form design components communicate form-events privately with their stewarding Mediators, not through static constants of the appconstants class where your own constants are defined in the MVC_CONSTANTS Script Object, but through explicit strings copied from the bottom of the MVC_MEDIATORS Script Object. The intention is that this difference will discourage a programmer from using these reserved strings in their own code.

Representing the physical boundaries of the Application, Form Design Components and

Data Objects may remain reusable if they communicate to their associated Mediator or Proxy by dispatching custom Events instead of making method calls or sending Notifications. Unfortunately this is not possible in the Adobe Forms environment, so the component form-event handlers make method calls into the managing Mediator or Proxy.

This paragraph does not apply to Adobe Forms because Form Design Components cannot dispatch custom form-events. But if a Form Design Component or Data Object dispatches a form-event that the stewarding Mediator or Proxy is listening for, then it is likely that only that collaboration pair need ever know the particular custom form-event name. In the Adobe Forms implementation we are having the Form Design Component form-event handler call a method in the Mediator or Proxy, instead of dispatching a custom form-event, which it cannot do. Further communication between the stewarding Mediator or Proxy and the rest of the PureMVC system may occur through the use of Notifications.

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

Adobe forms-events are handled only by their stewarding Mediators. The Mediators then send Notifications to other Mediators or Commands, or retrieve Proxies to work on the data model.

Commands

The concrete Facade generally initializes the Controller with the set of Notification to Command mappings needed at *startup*. (A prepareControllerCommand is typically triggered in the startup Command to create all the other application specific Command mappings.)

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

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

Use of Macro and Simple Commands

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

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

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

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

Loosely Coupling Commands to Mediators and Proxies

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

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

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

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

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

Orchestration of Complex Actions and Business Logic

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

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

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

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

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

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

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

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

Commands interact with Mediators and Proxies, but should be insulated from boundary implementations (such as webservices or form design components).

Consider the following Commands used to prepare the system for use:

The Script Object called MVC_COMMANDS:

```
// A MacroCommand executed when the application starts.
xfa.puremvc.define({
     name: 'controller.command.StartupCommand',
     parent: xfa.puremvc.MacroCommand,
     scope: xfa
 },
 {
     // Initialize the MacroCommand by adding its subcommands.
     initializeMacroCommand: function()
     {
         this.addSubCommand( xfa.controller.command.PrepareModelCommand );
         this.addSubCommand( xfa.controller.command.PrepareViewCommand );
         this.addSubCommand(
                     xfa.controller.command.PrepareControllerCommand );
     }
 }
);
```

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

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

Before the user can be presented or interact with any of the application's data, the Model must be placed in a consistent, known state. Once this has been achieved, the View can be prepared to present the Model's data and allow the user to manipulate and interact with it. Finally any mappings between Notifications and Commands (other than this startup comment) are made, to prepare the Controller.

Therefore, the startup process usually consists of three broad sets of activities – preparation of the Model, followed by preparation of the View, and finally preparation of the Controller.

The Script Object called MVC_COMMANDS also contains:

```
// Create and register Proxies with the Model.
xfa.puremvc.define({
     name: 'controller.command.PrepareModelCommand',
     parent: xfa.puremvc.SimpleCommand,
     scope: xfa
 },
 {
     // Called by the MacroCommand
     execute: function(note)
     {
         this.facade.registerProxy( new xfa.model.proxy.SearchProxy(
                                        'SearchProxy') );
         this.facade.registerProxy( new xfa.model.proxy.PrefsProxy(
                                        'PrefsProxy')  );
         this.facade.registerProxy( new xfa.model.proxy.UsersProxy(
                                        'UsersProxy')   );
     }
 }
);
```

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

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

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

The Script Object called MVC_COMMANDS also might contain:

```
// Create and register Mediators with the View.
xfa.puremvc.define({
     name : 'controller.command.PrepareViewCommand',
     parent : xfa.puremvc.SimpleCommand,
     scope: scope
 },

 {
     execute : function(note) {
         // note that if the Mediator is to "manage" or steward a
         // form design component, you should include optional constructor
         // parameters and shown in this example, where form1
         // is being managed in this example. This allows the Mediator
         // to receive private notification of the design component's events

         this.facade.registerMediator(
                     new xfa.Form1Listener('Form1Listener',form1));
         this.facade.registerMediator(
                     new xfa.Button1Listener('Button1Listener',Button1));
     }
 }
);
```

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

Notice that the Mediator this Command creates and registers is the Form1Listener, which stewards the "form1" form design component.

Further, it passes the name of the Mediator as a string, and the form design component (object) that the Mediator stewards into the constructor of the Mediator. This is optional in the constructor, and when done, causes the Mediator to receive private notifications of events generated by the form design component. Though optional in theory, it is mandatory if you want the Mediator to receive and handle Form Events for the form design object it stewards..

To communicate with the rest of the system, the form design components need to have Mediators. And creating those Mediators requires a reference to the form design Components they will mediate, so the "optional" object in the registerMediator method is actually mandatory if events are to be handled by the Mediator.

The form design component's stewarding Mediator is the only class we're allowing to know anything about the form design component's implementation, so we handle the creation of the remaining Mediators in a similar way.

So, with the above three Commands in the MacroCommand, (`PrepareModelCommand`, `PrepareViewCommand and PrepareControllerCommand`) we have orchestrated an ordered initialization of the Model, View and Controller. In doing so, the Commands did not need to know very much about the Model, the View, or the Controller.

When the details of the Model or the implementation of the View change, the Proxies and Mediators are refactored as needed, without needing to change these commands.

Business Logic in the Commands should be insulated from refactoring that takes place at the application's boundaries (the form design components, or the data connections).

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

Mediators

A Mediator class is used to mediate the user's interaction with one or more of the application's View Components (such as form design fields, checkboxes, buttons) and the rest of the PureMVC application.

In the Adobe Forms implementation, a Mediator can be declared as steward of a form design component, and then made to react to events generated by that component to handle user gestures and requests for data from the Component. It sends and receives Notifications to communicate with the rest of the application.

Responsibilities of the Concrete Mediator

The Adobe Forms environment provides a variety of interactive UI components. These provide possibilities for presenting the data model to the user and allowing them to interact with it.

The PureMVC framework has been ported to nearly every popular development language

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

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

To the PureMVC-based application, a View Component is any UI component, regardless of what framework it is provided by or how many sub-components it may contain. A View Component (such as a field, or button) should encapsulate as much of its own state and operation as possible, exposing a simple API of events, methods and properties. This is the case with form design components in the Adobe Forms environment.

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

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

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

Designating a View Component Steward

The base Mediator implementation that comes with PureMVC accepts a name and a generic Object as (a form design component) its sole constructor arguments. While these constructor arguments are optional, providing them designates the Mediator as the steward of the provided form design component.

Your concrete Mediator's constructor will be made immediately available internally as a protected property called viewComponent.

You may also dynamically set a Mediator's View Component after it has been constructed

by calling its setViewComponent method.

However it was set, as steward of the viewComponent, the concrete Mediator will automatically receive private notification of events generated by this vewComponent. These private notifications are not received by any other part of the PureMVC framework. The concrete Mediator must not pass on these private notifications, but instead use application constants for notifications to generate public notifications to trigger activity in the rest of the system.

Listening and Responding to the View Component

A Mediator usually has only one View Component (form design component), but it might manage several, such as a subform and its contained buttons or controls. We can contain a group of related form design components in a single View Component (such as a subform) and the children will be available to the Mediator, and the Mediator will receive automatic private notifications of events generated by the children.(This means that events automatically "bubble up" through Mediators of all parent form design objects, as happens in ActionScript in Flash, for example), sending Notifications of events to the Mediator(s). But it is best to encapsulate as much of the form design component's implementation as possible.

So, a stewarding Mediator will automatically receive notification of events generated not only by the form design component it stewards (the viewComponent), but also those generated by any form design components that are children of the form design component that the Mediator was registered with (children of the one in viewComponent). Stewarding mediators for the child components receive notification of the events first, mediators of the higher ancestors receiving notification last. It is important therefore, that when your stewarding Mediator receives notification of a form-event, it check which form design component generated the form-event notification (was it the viewComponent form design component that generated the form-event, or was it one of its children, if any?)

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

In the Adobe Forms implementation of PureMVC, we set the View Component when the Mediator is constructed (when it is registered) or has its setViewComponent method called. By doing so, the Mediator receives form-event notifications from the named form design object, and any of its children. What the Mediator does in response to that notification is, of

course governed entirely by the requirements of the application.

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

- Inspect the notification for name or type, if expected.
- Inspect the notification body to see if the form-event is coming from a child or not
- Inspect or modify exposed properties (or call exposed methods) of the form design component.
- Inspect or modify exposed properties (or call exposed methods) of a Proxy.
- Send one or more Notifications that will be responded to by other Mediators or Commands (or possibly even the same Mediator instance).

IMPORTANT:The form-event notification name received by the Mediator is intended to be private between the form design component, and its Mediator(s). Do not forward this notification 'note object' or notify name using the public sendNotification. Instead, create a new Notification Name in the constants section (MVC_CONSTANTS) and send that Notification, including the received Notification body in the new notification. This will avoid an infinite loop if you happen to place the private notification name in the `listNotificationInterests` *section of the Mediator.*

Some good rules of thumb are:

- If a number of other Mediators must be involved in the overall response to a Form-Event, then update a common Proxy or send a Notification, which is responded to by interested Mediators, all of whom respond appropriately.
- If a great amount of coordinated interaction (ordered steps) with other Mediators is required, a good practice is to use a Command to encode the steps in one place.
- Consider it a bad practice to retrieve other Mediators and act upon them, or to design Mediators to expose such manipulation methods.
- To manipulate and distribute application state information to the Mediators, set values or call methods on Proxies created to maintain state. Let the Mediators be interested in the Notifications sent by the state-keeping Proxies.

Handling Notifications in the Concrete Mediator

In contrast to the explicit adding of form-event notification handlers, stewarding Mediators (Mediators that steward specific form design components and their children), the process of coupling the Mediator to the PureMVC system is a simple and automatic one.

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

NOTE: DO NOT include the private strings used for form-event notification in this array. Doing so is unnecessary, and opens the possibility of an infinite loop, should the incoming event notification be forwarded as a public event with sendNotification (contrary to best practices).

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

NOTE: Due to quirks of the Adobe Forms JavaScript Interpreter, DO NOT place the array definition square brackets on separate lines by themselves. While opening and closing square brackets may be on separate lines, the brackets themselves must not be on a line by themselves, they must be on a line with some other expression, such as one of the notification name constants.

Defining the Mediator's list of Notification interests is easy (these constants were defined in the MVC_CONSTANTS Script Object:

```
listNotificationInterests: function()
{
    return [xfa.appconstants.SEARCH_FAILED, //notice the brackets are not alone
    xfa.appconstants.SEARCH_SUCCESS ]; //...even though they are on separate lines
}
```

When one of the Notifications named in the Mediator's response is sent by *any* actor in the system (including the Mediator itself), the Mediator's handleNotification method will be called, and the Notification passed in. (So be careful, forwarding a Notification that the same Mediator lists in its own listNotificationInterests, will cause an infinite loop.)

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

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

```
handleNotification: function(note) {
    switch ( note.getName() )
    {
            case xfa.appconstants.SEARCH_FAILED:
                    controlBar.rawValue = xfa.appconstants.STATUS_FAILED;
                    xfa.host.setFocus(controlBar.somExpression);
                    break;
            case xfa.appconstants.SEARCH_SUCCESS:
                    controlBar.rawValue = xfa.appconstants.STATUS_SUCCESS;
                    break;
    }
}
```

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

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

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

With private Form-Events Notification, we have a number of handler switch/case blocks, usually one for each Form-Event Notification the Mediator handles. Generally these switch/case blocks just send public Notifications. They should not be complicated.

With public Notifications, you have a single handler method, inside which you handle all public Notifications the Mediator is interested in using the switch/case statements.

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

There has been much debate on the usage of 'switch/case' as many developers consider it too limiting since all cases execute within the same scoped method. However, the single Notification method and 'switch/case' style was specifically chosen to limit what is done inside the Mediator, and remains the recommended construct. The switch/case block should be simple, usually sending a notification to a Command or other Mediator that might handle more complex processing.

The Mediator is meant to mediate communications between the Form Design Component and the rest of the system, not to perform complex processing. This way you can change the Mediator and/or the form design component it stewards, without affecting the more complex processing.

Consider the role of a translator mediating the exchange of conversation between her Ambassador and the rest of the members at a UN conference. She should rarely be doing more than simple translation and forwarding of messages, occasionally reaching for an appropriate metaphor or fact. The translator will not be making political decisions. The same is true of the Mediator's role within PureMVC.

Coupling of Mediators to Proxies and other Mediators

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

In terms of form design, the Model here refers to data connections or XML data manipulation. It also refers to the XML that is saved or submitted from the Adobe Form. While the form design components have a data model behind them, that is part of the View, and not part of the Model, which maintains its own version of form data, which is submitted or saved by the form. In the Adobe Forms implementation of PureMVC, you have have a Mediator that manages the *content* of a form design field while a separate Proxy manages the form data that is submitted and saved by the form. By default, they contain the same values, but the Proxy has final say over what is saved and submitted, so they might contain different values. For example, a form field might show a Social Security Number, and the form-data-model in the Adobe Form always contains what is shown on the display. However, because a separate data model is maintained by a Proxy for what is saved and submitted, the Proxy can over-ride the default reflection of what is seen on the screen, and store something else to be saved and submitted. The representation of an on-screen

field might be absent from the Proxy's data model, or it might be present but contain a different value than is displayed – it might contain a redacted or encrypted version of the Social Security Number. Hidden form design fields are a special case, since they do not interact with the user, are therefore not part of the View, so they are usually entirely managed by a Proxy. Hidden fields are one example of many data types that proxies might manage.

Mediators may freely retrieve Proxies from the Model, and read or manipulate the Data Object via any API exposed by the Proxy.

However, doing the work in a Command will loosen the coupling between the View and the Model.

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

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

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

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

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

User Interaction with View Components and Mediators

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

The collaboration between the LoginPanel subform and the LoginPanelMediator is that when the user clicks a "login button" an automatic private button click Notification is sent to the button's Mediator when the user has entered their credentials and wants to try logging in. The LoginPanelMediator handles the private Form-Event Notification by sending a public Notification with component's populated subform object as the body. This private Event Notification "bubbles up", that is, any Mediator that stewards the button itself first receives the private notification, then the Mediator that stewards the subform that encloses the button receives the private notification, and Mediators that steward any ancestor form design components then receive it, ending with any stewarding Mediator of the top-level form object (if one exists).

The Form Design Components (username field, password field, button) hide their internal implementation. Their entire API used by the Mediator are hidden from the rest of the application. These API consisting of the private Click Form-Event Notification, the formattedValue properties of the form design fields to be checked (username and password fields), and the Panel (subform) "login status" field.

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

(For a full working form design with the following code, see the attachment, to this document, login.xdp. For an explanation of attachments in PDF documents like this one you're reading, see Appendix C – Attachments in PDFs. If you're reading here a Word Doc instead, then see the attachment attached to the Word document, in that appendix.)

A value-only class defined in MVC_MYCLASSLETS:

```
xfa.puremvc.define(
    {name:'LoginVO'
    , scope: xfa
    },
    {
            username:''
          , password:''
          , authtoken:''
          , reset:function(){
                username='';
                password='';
                authtoken=null;
          }
    },
    {
          NAME:'LoginVO'
    }
);
```

The following Mediator is registered in the classlet in MVC_COMMANDS (called `PrepareViewCommand`) like this, declaring this Mediator as steward (receiver of private Form-Event Notifications) of the "subLogin" form design subform:

```
this.facade.registerMediator(
new xfa.LoginPanelMediator(xfa.LoginPanelMediator.NAME, subLogin));
```

A Mediator for interacting with the subLogin subform, in MVC_MEDIATORS:

```
xfa.puremvc.define(
    {name:'LoginPanelMediator'
    , parent: xfa.puremvc.Mediator
    , scope: xfa
    },
    {
          listNotificationInterests: function()
          {
                return [xfa.appconstants.LOGIN_FAILED,
                      xfa.appconstants.LOGIN_SUCCES];
          },
          handleNotification: function(note)
          {
                switch (note.getName())
                {
                   //handle private notifications

                   case 'click_form_event_PRIVATE_only':
                   if (note.getBody().name == 'butLogin')
                      // User clicked Login Button; try to log in
                      if (!xfa.loginvo)xfa.loginvo=new xfa.LoginVO();

                      xfa.loginvo.username=
                      viewComponent.txtUserName.formattedValue();

                      xfa.loginvo.password=
                      viewComponent.txtPassword.formattedValue();

                      sendNotification(
                      xfa.appconstants.LOGIN, xfa.loginvo );
                   break;
                   //handle public notifications

                   case xfa.appconstants.LOGIN_FAILED:
                   viewComponent.loginStatus.rawValue =
                   xfa.appconstants.NOT_LOGGED_IN;
                   xfa.loginvo.reset();
                   break;

                   case xfa.appconstants.LOGIN_SUCCESS:
                   viewComponent.loginStatus.rawValue = xfa.appconstants.LOGGED_IN;
                   xfa.loginvo.reset();
                   break;
                }
          }

    },
    {
          NAME:'LoginPanelMediator'
    }
);
```

Note that the LoginPanelMediator automatically receives private notifications of form-events fired by children, such as "butLogin" in this case, so that the handleNotification method will be invoked when the user has clicked the Login button. In the handleNotification method, the getBody() form design object (the button) is checked to see what child (by name – the name of the button) fired the click private Notification. The public LOGIN Notification is sent, bearing the a populated value-only object (xfa.loginvo), which contains the entered username and password. We sent this, instead of the subform itself (in viewComponent) because we don't want the recipient of the public Notification to know anything about the subform itself. This allows us to change the subform, and only make corresponding changes to its stewarding Mediator, without breaking any other code in the form design.

Earlier, we registered the LoginCommand to this same public Notification. That Command will invoke the LoginProxy's login method, passing the loginvo. The LoginProxy will attempt the login with the remote service, and in turn send a public LOGIN_SUCCESS or LOGIN_FAILED Notification. (These classes are defined at the end of the section on Proxies.)

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

Notice that we didn't list the private Form-Event Notification string 'click_form_event_PRIVATE_only' in the notificationInterests. This is because we declared this Mediator as steward of the subform, so it will automatically receive private Form-Event Notifications of this type for all events fired by itself and all its children. (So in the notificationHandler, we check to see the name of the object that fired the private Form-event Notification, to see which child object fired.) In fact, it is a good idea not to list the private Form-Event Notification in notificationInterests in case the programmer should inadvertently use sendNotification, including "note" as the Notification object sent. If these unlikely conditions were to happen, an infinite loop would be created where this handler keeps sending a notification that it catches, itself, and forwards again, and catches again…

To further discourage this bad practice, 'click_form_event_PRIVATE_only' is used as a string, and no public constant (that could be used elsewhere) is defined, because this Form-Event Notification is intended to be private between a form design component and its stewarding Mediator. The string is copy/pasted from all possible Form-Event Notification strings, at the bottom of the MVC_MEDIATORS Script Object.)

Proxies

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

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

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

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

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

A Proxy might also manage form data of various types: the form-data that is saved and submitted by the form, hidden fields on the form, nodes in the xml data associated with the form, design-time xml information embedded in the form (using the XML Source tab in Adobe Designer), run-time xml data embedded in the form, Designer's "form variables" and other data items specific to the Adobe Forms environment.

Responsibilities of the Concrete Proxy

The concrete Proxy allows us to encapsulate a piece of the data model, wherever it comes from and whatever its type, by managing the Data Object and the application's access to it. This allows us to, for example, move a table of values from a server database to design-time embedded xml, removing a dependency of the form on a server, without having to change any of the code in the application except the Proxy that manages this table of values.

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

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

Common variations on the Proxy pattern include:

- *Remote Proxy*, where the data managed by the concrete Proxy is in a remote location and will be accessed via a service of some sort.
- *Proxy and Delegate*, where access to a service object needs to be shared between multiple Proxies. The Delegate class maintains the service object and controls access to it, ensuring that responses are properly routed to their requestors.
- *Protection Proxy*, used when objects need to have different access rights.
- *Virtual Proxy*, which creates expensive objects on demand.
- *Smart Proxy*, loads data object into memory on first access, performs reference counting, allows locking of object to ensure no other object can change it.

Prevent Coupling to Mediators

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

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

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

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

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

Encapsulate Domain Logic in Proxies

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

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

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

For instance, the computation of sales tax is a Domain Logic function that should reside in a Proxy, not a Mediator or Command. Encoding and decoding data, or accessing different storage types are other examples of what should reside in, and hidden by the Proxy.

Though these operations could sometimes be performed in any of those places (Mediator or Command), placing them in a Proxy is not only logical, but also keeps the other tiers lighter and easier to refactor.

A Mediator may retrieve the Proxy; call its sales tax function, passing in some form items perhaps. But placing the actual computation in the Mediator would be embedding Domain Logic in the View tier. Sales tax computation is a rule belonging to the Domain Model. Sales tax computation is not part of what the user actually sees or interacts with. It is solely data related. Even though the user sees the result of the computation, the computation itself is a data related operation. The View merely sees it as a property of the Domain Model, available if the appropriate inputs are present.

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

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

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

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

Interacting with Remote Proxies

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

How the Proxy gets its data depends on the client platform, the remote service implementation, and the preferences of the developer. In an Adobe Forms environment we might use a WebService Data Connection to make service requests from within the Proxy.

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

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

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

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

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

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

- A View Component initiates a search by dispatching a private Form-Event Notification (say, from a button).
- Its Mediator responds by retrieving the appropriate Remote Proxy and setting a searchCriteria property with a setter function call.

- The setter function, stores the value and initiates the search via a webservice Data Connection in the form design, and checks the result field(s) bound to the webservice Data Connection response for result and fault messages.
- The Proxy then sends a public Notification indicating success or failure, bundling a reference to the data object as the body of the Notification (the data object is probably a hidden field on the form design).
- Another Mediator has previously expressed interest in that particular Notification and responds by taking the data from the Notification body, and setting the data into a display format, filling fields, creating table rows, etc.

Or consider a LoginProxy, who holds a LoginVO (a Value Object; a simple data carrier class). The LoginVO might look like this, as shown earlier in the Mediator section:

In MY_CLASSLETS Script Object:

```
xfa.puremvc.define(
   {name:'LoginVO'
   , scope: xfa
   },
    {
            username:''
          , password:''
          , authtoken:''
          , reset:function(){
                username='';
                password='';
                authtoken=null;
          }
    },
    {
          NAME:'LoginVO'
    }
);
```

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

MVC_CONSTANTS has these constants present:

```
, LOGIN_SUCCESS: 'loginSuccess'
, LOGIN_FAILED: 'loginFailed'
, LOGGED_OUT: 'loggedOut'
```

The MVC_PROXIES script object has this:

```
// A proxy to log the user in
xfa.puremvc.define(
   {name:'LoginProxy'
   , parent: xfa.puremvc.Proxy
   , scope: xfa
   },
   {
          loginvo:{},
          getLoginVO: function(){
                return this.loginvo;
          },
          setLoginVO: function(LoginVO){
                this.loginvo=LoginVO;
          },

          // The user is logged in if the login VO contains an auth token
          loggedIn: function(){
                return (this.loginvo.authToken !=null)? true : false;
          },

          // Subsequent calls to services after login must include the auth token
          getAuthToken: function(){
                return (this.loginvo.authToken != null)?
this.loginvo.authToken : null;
          },

          // Set the users credentials and log in, or log out and try again
          login: function(tryLogin){
                if ( ! this.loggedIn() ) {
                      this.loginvo.username = tryLogin.username;
                      this.loginvo.password = tryLogin.password;
                      //set the webservice input fields here
                      //call the service here
                      //check the results (output fields) here
                      if ((this.loginvo.username="Administrator") &&
(this.loginvo.password == "password")) { //if output return code is positive

                                  //setData(event.result);
                                  this.facade.sendNotification(
```

```
xfa.appconstants.LOGIN_SUCCESS, this.loginvo.authToken );
                    } else { //else output return code is negative

  this.facade.sendNotification(xfa.appconstants.LOGIN_FAILED, 'login
failed' );
                    }
              } else {
                    this.logout();
                    this.login(tryLogin);
              }
        },

        // To log out, simply clear the LoginVO
        logout: function()
        {
              if ( this.loggedIn() ) loginVO = new LoginVO( );
              this.facade.sendNotification( xfa.appconstants.LOGGED_OUT );
        }
  },
  {
        NAME:'LoginProxy'
  }
);
```

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

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

In the MVC_COMMANDS Script Object:

LoginCommand:

```
xfa.puremvc.define(
   {name:'LoginCommand'
          , parent: xfa.puremvc.SimpleCommand
          , scope: xfa
   },
   {
          execute: function(note)
          {
                 var loginVO = note.getBody();
                 var loginProxy = this.facade.retrieveProxy( xfa.LoginProxy.NAME );
                 loginProxy.login(loginVO);
          }
   },
   {
   }
);
```

GetPrefsCommand:

```
xfa.puremvc.define(
   {name:'GetPrefsCommand'
          , parent: xfa.puremvc.SimpleCommand
          , scope: xfa
   },
   {
          execute:function(note){
                 var authToken = note.getBody();
                 var prefsProxy = this.facade.retrieveProxy( xfa.PrefsProxy.NAME );
                 prefsProxy.getPrefs(authToken);
          }
   },
   {
   },
);
```

DETAILS of the Adobe Forms Implementation

PureMVC is a lightweight framework for creating applications based upon the classic Model, View and Controller concept.

There is a JavaScript port of the framework which can be used with Forms created with Adobe Forms Designer. The home for the JavaScript port is here: http://trac.puremvc.org/PureMVC_JS

A set of files, including an Adobe Forms Designer *Template* has been included with this file, containing a working implementation of a blank form that implements the PureMVC framework. This template can be used when starting a form from scratch, with the PureMVC framework already in place. Other files detailed in Appendix D, help add PureMVC to an existing form, and offer components to build working forms on the PureMVC framework.

This document is intended to explain how PureMVC for JavaScript was worked into Adobe Designer, and give best practices as to its use in the Adobe Forms environment.

For official documentation, see the JavaScript port home at the URL shown above. The particular files used in this implementation were the ones listed for download as "Native Multicore". Documentation for best practices can be found on this page: http://puremvc.org/content/view/98/189/ (on the "Docs" menu selection at the puremvc.org site. That document is the original, upon which this document for an Adobe Forms implementation was based.)

The rest of this document assumes familiarity with those materials, and concentrates on implementation details in Adobe Forms Designer.

As mentioned, everything discussed here is exemplified in the accompanying Adobe Designer *Template*, called "`pureMVC.tds`", and other files provided. The .tds file is a blank form with only the script objects and form-event scripting present that implement the PureMVC framework in an Adobe Form.

Script Objects

Present are 6 Script Objects involved in the pureMVC framework. Their names indicate their intended use. All names start with "MVC_" to indicate that they are part of the framework implementation for Adobe Forms. While 6 Script Objects contain parts of the framework, only one of them contains the PureMVC library and API. One contains utilities required of this implementation, and the other 4 contain your code that you use to work with and extend the PureMVC framework into an application. This makes it easy to upgrade to new editions of the PureMVC library and API: you only need to update one Script Object. The other 5 Script objects are used to organize and isolate your implementation code.

A small discussion of the mechanics behind Script Objects in Adobe Designer is necessary.

About Script Objects and OO JavaScript

INITIALIZATION CODE

Script Objects are intended to contain only JavaScript functions, to be called from form-event handlers throughout the form. However, any variables or script not contained in a function body is considered initialization or constructor code for that Script Object. Such variables or script are evaluated upon first reference to the Script Object by any other code in the form design.

NON-PERSISTENT:

Theoretically, these variables (the ones declared outside function code in Script Objects), if assigned a value, the value will stay until re-assigned, or until Reader/Acrobat/LiveCycle shuts down the form. Unfortunately, in practice, this is not the case, as they can lose their values during a user session, so cannot be trusted to hold their assignments.

NON-SINGLE:

Theoretically, any JavaScript not inside a function body (in a Script Object), will run only once: it will run only the first time the Script Object is referenced from other script in the form design. Again, unfortunately, there are conditions where the script can run more than once during a user session, so cannot be trusted to run only one time. It cannot be trusted to be "initialization code" or "constructor code."

DISTRUST WORK-AROUND:

For these reasons of distrust, code that is outside a function body in these Script Objects

has been wrapped in a check for having previously run, so can now be trusted to run only once. No variable in these Script Objects, outside function body, is expected to hold a value. Instead, values are stored as extensions to the Adobe XFA Forms Model, which is native to Adobe Forms.

CLASSLET DEFINITIONS ONLY:

A *classlet* is similar to a class. It is a pseudo implementation of the concept of the object oriented class in JavaScript. JavaScript at this time is not an object-oriented language, but script that behaves like a class – a pseudo-class – is possible in JavaScript today. Such a pseudo-class is called a *classlet*.

Almost all of the code in the MVC_ Script Objects is outside function bodies! These Script Objects are present only to hold classlet (pseudo-class) definitions, which will be instantiated when the Adobe Forms-Event model is running, or by other already instantiated MVC classes. These Script Objects define the PureMVC library and API and your own PureMVC code for the form, in classlets, and these are instantiated and attached to the XFA Forms Model for global use throughout the form. Almost all of the provided code is object-oriented, encapsulated in classlet definitions.

FORCED REFERENCE:

Because code in a Script Object is never evaluated until it is first referenced from a form-event handler (or other form script), the form-level initialize form-event of the form design simply references each of the MVC_ Script Objects one at a time, in a specific order, by assigning a value (null, actually) to a variable that appears at the beginning of each of these Script Objects. Thus, when the form is initialized, all of the MVC_ Script Objects are evaluated. The variable that appears at the beginning of each Script Object is called called `evaluatejso`, and this variable is outside any function body. So by referencing this variable, the Script Object gets evaluated, which means that all code outside function bodies is immediately evaluated (executed).

EXTENDING THE XFA MODEL WITH CLASSLET DEFINITIONS:

As stated, these Script Objects have code that is *outside* function bodies, that is run immediately upon being referenced. This code simply defines classlets as extensions of the XFA Forms Model, so that the classlet definitions are available globally throughout the form. All classlet definitions are globally available. One of the MVC_ Script Objects, the MVC_CONSTANTS Script Object, defines a globally available class containing only static constants, making those constants globally available throughout the form.

EXTENDING THE XFA MODEL WITH INSTANCE VARIABLES:

Instantiation of PureMVC framework classes occurs by having the form Initialize form-event instantiate ApplicationFacade, and adding the *instantiated* object to the XFA Forms Model, making the facade instance globally available throughout the form. (Note that, we're talking about the class *instance* here, as opposed to the classlet *definition*. Both the classlet definition, and the instance are added to the XFA Forms Model, and both are available throughout the form.)

KICK-OFF BY INSTANTIATION:

The Initialize form-event then calls `startup` on the ApplicationFacade *instance*, thereby kicking off the *instantiation* of all the necessary classes that were *defined* in the rest of the MVC_ Script Objects, including the PureMVC library and API itself (which is contained in the Script Object called MVC_PUREMVC).

JAVASCRIPT INTERPRETER QUIRK:

It has been found that under *certain* circumstances, the following coding can cause the JavaScript interpreter to ignore your code:

- Any space between the word "function" and the left bracket "("
- Any space before or after an equal sign
- Having [or] (in array initialization) on separate lines from at least one of the array contents

For these reasons, such coding has been changed throughout the template, and it is highly recommended that you refrain from introducing these elements in your own code.

The MVC_PUREMVC Script Object

This contains minified JavaScript copy of the PureMVC objects and library. (All minified code is included, in original format, in the files that accompany this document. See Appendix C, and Appendix D.) This is the very first MVC_ Script Object that is evaluated when the form Initialization script runs. It has been changed for Adobe Forms in these 2 minor ways:

- A variable was added to the beginning, called `evaluatejso`, using a line like this:

```
var evaluatejso;//pureMVC version x.x
```

(to allow the form Initialize form-event to force evaluation of the Script Object)

- In the original JavaScript implementation of PureMVC, at the end of the code, a scope variable was set to "`this`". It was changed this to "`xfa`" so that the classlet definitions will become global extensions to the XFA Forms Model, instead of becoming extensions of the "`this`" object which, in the Adobe Forms environment means several different undesirable things, depending on the context.

No other changes were made to the PureMVC framework library and API which is contained in this MVC_ Script Object

HOW TO UPGRADE PureMVC TO A NEW VERSION

Simply delete the contents of the Script Object called MVC_PUREMVC, and paste the minimized version of the new PureMVC bin into that Script Object.

Next, add a new line with this statement to the very beginning of what you pasted, and change `x.x`:

```
var evaluatejso;//pureMVC version x.x
```

Finally, replace the last "`this`" at the end of the code, with "`xfa`".

The MVC_CONSTANTS Script Object

As always, this Script Object contains a variable called `evaluatejso`, used by the form Initialization to force this Script Object to be evaluated. Also, this Script Object is wrapped in code that ensures its evaluation only once per user session.

Now that the PureMVC library and API have been defined and made into global extensions of the XFA model, its capabilities are available for further classlet definitions in the rest of the MVC_ Script Objects. The second such Script Object evaluated is the MVC_CONSTANTS Script Object. It uses the PureMVC API to extend the XFA Forms Model with a simple class that contains only static constants. These constants are thereby made available to the rest of the form-events, classes and scripting. You reference these constants thus: `xfa.appconstants.CLICK_NOTIFY` where "`CLICK_NOTIFY`" is an example of a constant being referenced. Look inside this MVC_CONSTANTS Script Object for examples of how you may define your own constants in this same Script Object: just add yours to the list of constants already there. (Don't forget to begin each new line with a comma.)

The MVC_PROXIES Script Object

As always, this Script Object contains a variable called `evaluatejso`, used by the form Initialization to force this Script Object to be evaluated (such as defining classlets). As before, this Script Object is wrapped in code that ensures its evaluation only once per user session.

This Script Object is an organized place to put your Proxy definitions. Have them registered in the MVC_COMMANDS Script Object, in the place indicated therein. (See Below.)

Where would you keep data that is managed by a Proxy? There are several possibilities.

DATA MANAGEMENT OBJECTS

Included in the template file and re-usable Script Object component included with this document is a Script Object called OO_UTIL. This Script Object defines Dictionary and Collection classlets that are analogous to map and list objects, but have more functionality that is relevant to the Adobe Forms environment. While the data in these objects do not persist along with other form data, these objects can be made to cause their data to become persistent along with other form data by associating the Dictionary or Collection instance with a hidden text field. These objects automatically re-load their data from the hidden text field in future sessions, if used as prescribed. These objects are described below, in the explanation of the OO_UTIL script object.

HIDDEN FORM FIELDS

While it is advised that you go through a Mediator that is responsible for a form field, a *hidden* form field, having no user interaction, might exist exclusively for the purpose of persistent data storage (persistent along with other form data).

XFA EXTENSIONS

You can manage your data as xml nodes in the scripting environment. You can use `loadXML` to place XML from a string into the XFA model, and `saveXML` to retrieve a branch of nodes from the XFA model into a string.

Form data that has been mapped to fields (and possibly conforms to an XML schema associated with the form) starts at: `xfa.data.form`. (NOTE: The data at this node has an alias location: `xfa.datasets.data`. This XML tree is *normally* what is saved and submitted by the form, and represents the data model behind the View in this PureMVC implementation.

However, in this PureMVC implementation, a different XML tree is saved and submitted by the form, one that is controlled by the included `FormDataProxy`. See the next paragraph.) You can programmatically add your own nodes to `xfa.datasets`.

You can also add XML to the form design, in the XML Source tab, just below the `<xfa:datasets` tag, and reference and manipulate this data at `xfa.datasets`. The passwordGen example design has over 2,000 words added in this manner to the form design, at the node: `<xfa.datasets <commonWords>`.

THE FORMDATAPROXY

As stated above, without this PureMVC implementation, the data in the form View is the same data that is saved and submitted by the form. However, in this PureMVC implementation for Adobe Forms, a separate data model is maintained for saving and submitting. This separate data model is managed by a Proxy that is included in the provided framework for Adobe Forms. This Proxy is called the `FormDataProxy`. The location of the separate data model is `xfa.datasets.MVCDataModel`, while, as stated above, the location for the data model one sees in View components, and saves or submits in non-MVC Adobe forms, is at `xfa.data.form` (or the alias: `xfa.datasets.data`).

One of the provided Proxy registrations registers this FormDataProxy at form initialization time. The `exit` form-event in the provided framework automatically calls this FormDataProxy to update the representation of the field being exited, with data that will be saved or submitted by the form. When the FormDataProxy is registered at form initialization time, the FormDataProxy creates a separate XML tree exactly the same as the one defined by the form's schema, example XML, Data Model Connection, or form hierarchy.

As stated, on every `exit` form-event, this Proxy is called. There is a switch statement in that Proxy that controls what goes into the XML tree that gets saved or submitted by the form. The `default:` block of that switch simply stores the rawValue of the field being exited, as is. This is the same behavior as if there were no separate XML tree being saved or submitted.

What is important here is this: you can insert your own scripting in that switch block, to give your own Proxy scripting control over what is saved or submitted by this form. The effect is this: there is a View, with its own visible data, and there is the FormDataProxy with possibly different data that gets saved or submitted by the form. This means that the PureMVC *Model*, and specifically, the FormDataProxy, has full control over persistent form data (what

gets saved or submitted).

The FormDataProxy exposes an API of two methods for you to use, in storing information in the separate XML tree that gets saved or submitted. They are `updateBoundObj`, and `updateXML`.

The `updateBoundObj` method allows you to take advantage of the visual, GUI binding tools offered by Adobe Designer. The form designer can do as always, with non-MVC forms, drag-and drop or pick from menus, a binding relationship between a form design object and the XML nodes that get saved and submitted. When you make a call to `updateBoundObj`, you provide the `somExpression` of any form object that has been so bound to an XML node, and a value. The method simply gets the XML node that was visually bound to the form design object, and places the provided value in that XML node. This supports all of the binding types, including *none*, *name*, *global*, and *data connection*. So with `updateBoundObj`, everything operates as it does in a non-MVC form, except you have a place to put scripting that might make what is saved and submitted different from what is visible. This creates an important separation between View and Model, allowing one to change, without affecting the other.

The `updateXML` method allows you to change the XML that is saved and submitted without regard for any visual binding, or any relationship between View components and Model data. You name an xml node using dot-notation, and provide with it, a value to be placed at the node. Because this method has no regard whatsoever for anything in the View, it is the preferred approach, because now the view can change drastically, without affecting the data that is saved or submitted, and vice-versa. However, the cost of this de-coupling is that you can no longer take advantage of visual binding techniques offered by Adobe Designer. The two methods are offered to cater to MVC purists (who would use only `updateXML`) and Adobe Designer enthusiasts, who would cater to the visual tools offered (who would use `updateBoundObj`).

For `updateXML`, the dot-notation node locations available for updating are defined the same as would be with `updateBoundObj`, or with a non-MVC Adobe Form. The XML structure that is saved and submitted, and that you would update using this technique, is still controlled by the form hierarchy, or if present, a schema data connection, an example XML data connection, or an Adobe Data Model data connection.

Those two methods allow you to write to the XML tree at xfa.datasets.MVCDataModel, which is what gets saved or submitted. To read from this XML tree, simply use as your

source in any JavaScript statement:
`xfa.resolveNode("xfa.datasets.MVCDataModel.root.branch.leaf ")` where `"root.branch.leaf"` will be specific to the location you want to refer to.

NOTE: best practices for the PureMVC programming pattern would insist that you only refer to this data (either reading or writing) with code that is encapsulated in a Proxy, and preferably the Proxy called `FormDataProxy`. Since `FormDataProxy` is part of the provided framework, you might want to define a second Proxy of your own to perform your own application-specific manipulation of the data that gets saved and submitted, the data at `xfa.datasets.MVCDataModel`.

WEB SERVICE CALLS

You would create hidden fields for each of the input parameters to the webservice call defined in the data connection (if any). You would create hidden fields for each of the output parameters defined in the data connection. Finally, you might create a hidden button that executes the webservice call. Setup of the whole webservice would be exactly as you would do in a non-MVC Adobe Form.

Now you would create a Proxy that manages those hidden fields, and the hidden invocation button. Calls to this Proxy would set the content of the hidden fields that are associated with input parameters. A call to this Proxy would "click" the hidden button associated with executing this webservice. The Proxy next handle output from the webservice call by sending a public notification with the received data as payload to the notification.

By encapsulating in a Proxy, all activity around the data connection created in Adobe Designer, you are now free to change how the notification is dealt with by other actors in the application. You are also free to change how the other actors in the application prepare input for the webservice call, before handing that input to the Proxy. You are also finally free to change the actual webservice call itself in any way you need to, without affecting the rest of the application.

The MVC_MEDIATORS Script Object

As always, this Script Object contains a variable called `evaluatejso`, used by the form Initialization to force this Script Object to be evaluated. As before, this Script Object is wrapped in code that ensures its evaluation only once per user session.

This Script Object is an organized place to put your Mediator definitions. Have them

registered in the MVC_COMMANDS Script Object, in the place indicated therein. (See the discussion of the MVC_COMMANDS Script Object, below.)

NOTE: In order to take advantage of *form-event propagation* which is used to direct form-events to your stewarding mediators, you must make your form design compatible with Acrobat/Reader 9.1 or later. *File -> Form Properties -> Defaults -> Choose Target Version*.

Form-events are directed to your stewarding mediators through a function at the end of the MVC_MEDIATORS Script object called `DispatchEvent`.

THE DISPATCHEVENT FUNCTION

Recall that Adobe Forms follow a form-event model, much like Visual Basic and other development environments. Adobe Forms cannot create and issue *custom* events. Recall also that, by contrast, PureMVC does not follow a form-event model, but an Observer/Notify model instead. So how do we make the bridge between events, and a Notification system?

One way is presented here, and in the accompanying form design template. All events call a *DispatchEvent* function. This function searches through all mediators to find mediators that manage or steward the design component that generated the form-event. When found, the `handleNotification` method is called directly, privately, sending it a Notification Object that contains an explicit private string, not a global constant, so that no other part of the system is aware of the communication between the Mediator and the design component it manages. Because a string is used and not a constant, it is more difficult to use this string in the application (doing so would be contrary to best practice). So the Mediator receives a notification like any other, but it is private. The names of the notifications match the names of the events in the form. The Mediator can then act on the notification, changing the data Model through proxies, or sending notifications to other mediators and commands.

But how do the events reach this Mediator? Some manual editing of the Source XML for the form template has been done, to make every form-event on every object (with few exceptions to be explained), all call the provided `DispatchEvent` function in the MVC_MEDIATORS Script Object. Sounds inefficient, but due to the clever efficiency of the PureMVC library and API, actually, for a form, this is not a concern.

So this `DispatchEvent` function simply creates private PureMVC Notification objects for

almost every form-event on every object in the form design. All you need do is have the handleNotification section of your Mediator include the correct string in a case statement, the correct string for the form-event that your Mediator is interested in. Copy/paste the desired case statement with special string, from the bottom of the MVC_MEDIATORS Script Object. The object that generated the form-event will always be the object that your Mediator is managing, as declared in the Mediator registration, or a child of that object, if any. A reference to the object that generated the form-event will be sent in the notification for your Mediator to examine, to see which child object generated the form-event, what a field's contents might be for example, or the status of a check-box for another example.

So, how does EventDispatcher know which Mediator is managing or stewarding the design component that generated the form-event; which Mediator should receive Notification of this form-event? You register all your mediators in the MVC_COMMANDS Script Object with with xfa.facade.registerMediator calls. These involve using the "new" keyword to instantiate your Mediator. According to the PureMVC API documentation, this "new" keyword can accept two optional parameters for the constructor of a Mediator. In this implementation of PureMVC for Adobe Forms, when your Mediator manages or stewards a form design component, these parameters are *not* optional. You must supply them. The first is a string containing the name of the Mediator. The second is the form design object that the Mediator is managing. By providing this object in the Mediator constructor, your Mediator will automatically and privately receive Notifications of any events triggered by the object it is managing or stewarding. Eg:

```
xfa.facade.registerMediator(new xfa.Form1Listener('Form1Listener',form1));
```

or

```
xfa.facade.registerMediator(new xfa.Button1Listener('Button1Listener',Button1));
```

Then you simply place the correct strings in your switch statement of the handleNotification method, to select and act on the form-event notifications you are interested in. (eg.

```
handleNotification: function(note)
      {
            switch (note.getName())
            {
                  case 'initialize_form_event_PRIVATE_only':
                  case 'indexChange_form_event_PRIVATE_only':
                  case xfa.appconstants.DISPLAYRESULTS:
```

NOTE: There is a propagated form-event handler that calls the "DispatchEvent" method in MVC_MEDIATORS for every form-event of every object. (Form-event Propagation is a relatively new feature in the Adobe Forms model – Acrobat/Reader v9.1 and later – where form-event handling script can be set on a form design subform, and all children of that

subform will execute the so-marked form-event.) This is how a private PureMVC NOTIFY is generated for every form-event on every object. It is private in that it goes ONLY to the Mediator that manages or stewards the design component that generated the form-event, and to the stewarding mediators of any hierarchy ancestors of that design object. If you place your own script in the form-event handler for a form-event on an object, it will get executed *after* the *notify* for the form-event is processed by PureMVC. This is because propogated form-event handlers are executed before any specific code that is placed in an object's form-event handler. You can change this, so that a *notify* will *also* execute *after* your own code in the form-event handlers. Simply place your own call to the Form-event Dispatcher in your form-event handler *after* your own code. For instructions, see the note at the beginning of Appendix F. Keep in mind, however, in this case the call to Event Dispatcher, the notify, will occur twice: once from the propagation, and once for the call to Event Dispatcher that you manually placed after your script, in the form-event handler script.

NOTE: In a PureMVC notification handler for a form-event on an object, `getBody()` will normally return the object (field, button, etc) on which the form-event occurred, while `getName()` will return the string identifying which form-event it was that occurred. However, some notifications do not return the correct object in `getBody()`. In these cases Acrobat/Reader is not telling us which propagation of the form-event is the form design object that generated the form-event. So, those form-events offer `xfa.form.[topFormElement]` for `getBody()` in the notify (so for these events, your notify handler cannot know which specific form design object generated the form-event):

- `initialize, indexChange` [because of a bug in Acrobat/Reader, "null" is reported as the form design object that generates these form-events, making these notifications useless until the bug is fixed – bug is present in v11.0.0 though reported to Adobe in v9.1 – *When a new version is released by Adobe, you can test to see whether or not the bug has been fixed by running any of the forms provided with this document, after temporarily uncommenting the second comment line in DispatchEvent at the end of the MVC_MEDIATORS Script Object.*]

- `formReady, layoutReady` – these return the form1 object (or whatever the topFormElement is called in your design) when you call getBody(), because they are not form-events that are associated with any particular design object.

- `pre/postSave, pre/postPrint, docReady, docClose` – these also return the form1 object (or whatever the top form object is called in your design)

In the case of the bug that gives "null" for xfa.event.target in the propagated form-event, the form-event handler provided with this documentation explicitly sends the form1 object (or whatever the top form object is called in your design) instead of xfa.event.target for getBody() in the `initialize` and `indexChange` events. For these two form-events, while the bug persists, you can get better results, where the actual object is included in the notification body, if you simply add the following code line (notice the keyword `this`) to the form-event handler of the design object you want to be able to pick up in getBody().

For the `initialize` form-event:

```
MVC_MEDIATORS.DispatchEvent(this, 'initialize_form_form-event_PRIVATE_only');
```

and this line for the indexChange form-event:

```
MVC_MEDIATORS.DispatchEvent(this,'indexChange_form_event_PRIVATE_only');
```

Some of these events (such as initialize) fire several times, once for each object that handles the form-event. If you wish to ensure that your notify handler code runs *only once*, then here's a suggestion (replace `id` with something unique for this script that should run only once):

```
if ((!xfa.id) || (xfa.id == null) || (xfa.id.length==0)){
     xfa.id="x";
    ... other script that will run only once ...
}
```

An example of this may be found in the examples that accompany this file, in the SCRIPTS Script Object, where there is a function called `my_init()` that must never run more than once per document session. In this example, "`id`" has been replaced with "`initdone`".

NOTE: See Appendix F for a description of how to edit the XML Source of your *existing form design*, if you wish to cause all events of all objects call this `DispatchEvent` function. (Appendix E steps you through the whole process of installing PureMVC on an *existing form design*.)

CALC AND VALIDATE EVENTS:

These are the two form-events that are not worked into the automatic calls to DispatchEvent function for translation into PureMVC Notifications. There are two reasons for this. Both are characteristics of these form-events that are not shared by the other form-events in the Adobe Forms model:

- These events do not support *Form-event Propagation*. So you must implement your own explicit calls to `DispatchEvent` for these events.

- These events expect a return value from the script being run in the form-event. For Calculate, the return value (the last value on the stack – the last value referenced) becomes the data content of the object running the script. For Validate, a return (last value on stack – last value referenced) of true indicates "pass validation" and a return of false indicates "fail validation."

- You must implement your handler to *return a valid value* for the form-event. You do this by simply ending the case block in the `handleNotification` switch of the stewarding Mediator, with a return statement that returns a value. For example:

  In the stewarding Mediator for a hypothetical 'TextField1' we have something like this:

```
handleNotification: function(note) {
    try{
    switch (note.getName()) {
    //handle private notifications
          case 'click_form_event_PRIVATE_only':
                if (note.getBody().name == 'butLogin'){
                      .
                      .
                      .
                }
                break;
          case 'calculate_form_event_PRIVATE_only': //copied from bottom of
                                                    //MVC_MEDIATORS
                if (note.getBody().name == 'TextField1'){
                      var val = "default value";
                      .
                      . //maybe access a Proxy to refer to any data
                      . //maybe send a Notification for a Command or
                      . //other Mediator that changes data in the Proxy
                      .
                      return val; // <==== returning a value here
                }
                break;
```

  Then in the *calculate* form-event for the form-design object (TextField1 in our example), you place this call to the `DispatchEvent` function (the string was copied from the bottom of the "MVC_MEDIATORS" Script Object):

```
MVC_MEDIATORS.DispatchEvent(this, 'calculate_form_event_PRIVATE_only');
```

  For a Validate form-event, you would do exactly the same things, but in each case

use this string instead, copied from the bottom of MVC_MEDIATORS: `'validate_form_event_PRIVATE_only'`

- Because these form-events do not go through the `DispatchEvent` function in the MVC_MEDIATORS Script Object, there is no *bubbling up* of these form events through stewarding mediators of ancestor hierarchy design objects.

- If you deliberately register more than one Mediator as steward for a form design component, the return value arriving back at the form design form-event handler is undefined. It could be the result of the handler in any of the stewarding Mediators. If a stewarding Mediator does not process the calculate or validate Notification, then "null" is returned. Therefore, if you register more than one Mediator as steward for a for design component, make sure the calculate or validate handlers in each are identical, returning the same value.

The MVC_COMMANDS Script Object

As always, this Script Object contains a variable called `evaluatejso`, used by the form Initialization to force this Script Object to be evaluated. As before, this Script Object is wrapped in code that ensures its evaluation only once per user session.

This Script Object is an organized place to put your Command definitions. Have them registered in this Script Object, in the PrepareController Command. (See Below.)

This Script Object comes with 4 commands aready placed there, and registered in the MVC_APPLICATIONFACADE Script Object. Simply copy one that's there, change its name and coding. Three of the four are `PrepareModel`, `PrepareView` and `PrepareController`. These are centralized places to register your application proxies, mediators, and commands.

The fourth Command already supplied is a `StartupCommand` which, unlike the other commands which all inherit from `puremvc.SimpleCommand`, this one inherits from `puremvc.MacroCommand`. It simply runs the three commands mentioned in the previous paragraph. So when the ApplicationFacade runs this `StartupCommand`, the proxies, mediators, and commands in your application all get registered.

NOTE: `PrepareController` should probably be the last Command definition in your MVC_COMMANDS Script Object, to ensure that other commands it might reference will have been already defined before it. **ALSO:** Macro Command definitions should also probably be at the bottom, just before the `PrepareController` Command definition, for the

same reason.

The MVC_APPLICATIONFACADE Script Object

As always, this Script Object contains a variable called `evaluatejso`, used by the form Initialization to force this Script Object to be evaluated. As before, this Script Object is wrapped in code that ensures its evaluation only once per user session.

The purpose and use of this Script Object is covered in depth in other PureMVC documentation. The class instance is exposed globally as `xfa.facade` . The form Initialize form-event runs getInstance on the classlet definition, this object, stores the instance as a global extension of the XFA framework (as `xfa.facade`), then calls the startup Command on this instance, which causes this the `StartupCommand` in MVC_COMMANDS to run, which together, initialize and start the entire PureMVC framework)

The MY_CLASSLETS Script Object

As always, this Script Object contains a variable called `evaluatejso`, used by the form Initialization to force this Script Object to be evaluated. As before, this Script Object is wrapped in code that ensures its evaluation only once per user session.

PureMVC and the Adobe Forms model are purely object oriented. Why shouldn't your own code be too? The purpose and use of this Script Object is to offer you a place to use the `xfa.define` API exposed by the PureMVC framework to define your own classlets for use in the code that you place in the MVC_PROXIES, MVC_MEDIATORS and MVC_COMMANDS Script Objects. In this manner, all of your own code that you place in the PureMVC framework can be completely object oriented too.

The OO_UTIL Script Object

As always, this Script Object contains a variable called `evaluatejso`, used by the form Initialization to force this Script Object to be evaluated. As before, this Script Object is wrapped in code that ensures its evaluation only once per user session.

This script object is not intended to be changed from one form design to the next, and includes code that the author has used often in many forms designs:

TRIMSTRING:

One of the most often used functions is not part of the JavaScript language. It has been included here, and applied to the xfa model so that it is globally available like this: `xfa.trimString(…)`. (It is true, we could have just typed `OO_UTIL.trimString(…)` but this is a bit less typing, simply for the sake of convenience.)

DICTIONARY AND COLLECTION CLASSES

The typical JavaScript programmer has used the language in the browser environment, and come to expect such classes (or `map` and `list` classes) to be available. There are no such classes in the Adobe Forms environment. For convenience, the OO_UTIL Script Object contains extensions to the xfa model that make such classes globally available for instantiation, using the classlet definition API in PureMVC. These classlet implementations are much more capable than those in the browser environment, and have features specific to the Adobe Forms environment..

Collection

Eg: `var obj = new xfa.Collection();`

`Var obj` could be `xfa.obj`, in which case you make the collection globally available, though you probably only want to access it through a Proxy. More importantly though, you make it persistent outside the context of the currently running form-event or notification.

Properties and methods available:

Method/Property	Example	Description
isEmpty	obj.isEmpty()	Returns true if the collection is empty.
Clear	obj.clear()	Empties the collection
count	obj.count()	Returns the number of objects in the collection
add	obj.add(newItem)	Appends the new item to the collection
remove	obj.remove(index)	Removes the item that occupies position

		"index" in the collection (0-based)
clone	obj.clone()	Returns a copy of the collection object
exists	obj.exists(item)	Returns true if "item" exists in the collection
item	obj.item(index)	Returns the object that occuplies position "index" in the collection
setPersist	obj.setPersist(txtObj)	Sets a text field to hold contents of this Collection object. Use this only if you want the Collection object to persist after the form is submitted to server. Content is held in the field in delimited format where ~ separates items
setListBox	obj.setListBox(lstObj)	Associates a listbox or drop-down listbox. (To make this list persistent, use this b4 setPersist)
selectListItem	obj.selectListItem()	Selects the first item in the associated listbox or drop-down.
getSelectedItem	obj.getSelectedItem()	If associated with a listbox or drop-down, returns the Item of the selection. ("" if nothing is selected)
serialize	obj.serialize()	stores the collection in the text object previously associated with the setPersist method (this happens automatically, whenever a change is made to the collection)
sortAlph	obj.sortAlph()	Sorts items in alphabetical order eg: 1, 12, 2, 22, 3
sortNum	obj.sortNum()	Sorts items in numeric order: 1, 2, 12, 22, 3. Note: sorts only items that are numberic, leaving other items undisturbed.
pushItem	obj.pushItem(newItem)	Like "add", appends the new item to the end of the collection.
popItem	obj.popItem()	Removes and returns the last item from the collection.

joinItems	obj.joinItems()	Returns a string containing all items in the collection, separated by commas.
	obj.joinItems(delim)	Returns a string containing all items in the collection, separated by the given delimiter character.
reverseItems	obj.reverseItems()	Reverses the order of items in the collection
shiftItems	obj.shiftItems()	Removes and returns the first item in the Collection.
unShiftItems	obj.unShiftItems(newItem)	pre-pends the new item to the beginning of the collection.
getArray	obj.getArray()	Returns the collection as an array
loadFromArray	obj.loadFromArray(a)	Loads the collection obj, from the array "a"
selectFromDlg	obj. selectFromDlg(sInstructions)	Presents all items in a pop-up dialog in a listbox. The listbox is headed by the instructions you specify in sInstructions. Returns the selected text.

Dictionary

Eg: `var obj = new xfa.Dictionary();`

`Var obj` could be `xfa.obj`, in which case you make the collection globally available, though you probably only want to access it through a Proxy. More importantly though, you make it persistent outside the context of the currently running form-event or notification.

Properties and methods available:

Method/Property	Example	Description
error	obj.error	Returns true if the added key already there or key to remove wasn't there or item to return doesn't exist
count	obj.count	Returns number of items in the collection

exists	obj.exists(key)	Returns true if a key exists, else false
setPersist	obj.setPersist(txtObj)	Sets a text field to hold contents of this Dictionary object. Use this only if you want the Dictionary object to persist after the form is submitted to server. Content is held in the field in delimited format where ~ separates pairs, and elements of pairs are separated by ^
setPersistXML	obj.setPersistXML(txtObj)	Same as setPersist (above) but text field holds content in XML format instead. Keys are element names and items are values
setListBoxKy	obj.setListBoxKy(lstObj)	Associates a listbox or drop-down listbox. - keys display, items are bound values. (To make this list persistent, use this b4 useing setPersist) Changes in the Dictionary object are automatically reflected in the lisbox or drop-down listbox
setListBoxIt	obj.setListBoxIt(lstObj)	Associates a listbox or drop-down listbox. - items display, keys are bound values. (see setListBoxKy)
selectListItem	obj.selectListItem()	Selects the first item in the associated listbox or drop-down.
getSelectedKey	obj.getSelectedKey()	If associated with a listbox or drop-down, returns the Key of the selection. ("" if nothing is selected)
getSelectedItem	obj.getSelectedItem()	If associated with a listbox or drop-down, returns the Item of the selection.

		("" if nothing is selected)
add	obj.add(key,item)	Adds an item and key pair to the collection
item	obj.item(key)	Returns the item associated with a key
	obj.item(key,item)	When used with 2 parameters, same as add"
items	obj.items()	Returns an array of all items (not keys)
removeAll	obj.removeAll()	Removes everything from the collection
clear	obj.clear()	Same as removeAll
remove	obj.remove(key)	Removes the item and key pair
keys	obj.keys()	Returns an array containing all keys
key	obj.key(item)	Returns the key corresponding to an item
sortKeysAlph	obj.sortKeysAlph()	Sorts item and key pairs in order by Key alphabetically: "1", "12", "2", "22", "3"
sortItemsAlph	obj.sortItemsAlph()	sorts item and key pairs in order by Item alphabetically: "1", "12", "2", "22", "3"
sortKeysNum	obj.sortKeysNum()	Sorts item and key pairs in order by Key in numeric order: 1,2,3,12,22 (sorts only numeric keys, leaving others undisturbed)
sortItemsNum	obj.sortItemsNum()	sorts item and key pairs in order by Item in numeric order: 1,2,3,12,22 (sorts only numeric items, leaving others undisturbed)
selectItemFromDlg	obj.selectItemFromDlg(sInstructions)	

selectKeyFromDlg obj. selectKeyFromDlg(sInstructions)

Presents either items or keys in a lisbox in a popup dialog box, with sInstructions showing above the listbox. Returns the selected string as shown in the listbox.

The LOG4JS Script Object

As always, this Script Object contains a variable called `evaluatejso`, used by the form Initialization to force this Script Object to be evaluated. As before, this Script Object is wrapped in code that ensures its evaluation only once per user session.

This script object is not intended to be changed from one form design to the next. It contains tools to help with the develop/debug process.

LOGGING

Often script writers for Adobe Designer will write app.alert or console.println statements for debugging purposes. After they have finished serving their purpose, they must be removed or commented-out. Then if a future need requires them again, they must be replaced, or the commenting removed.

Java exhibits a more efficient messaging system in log4j, used by LiveCycle itself. Included with the template file that accompanies this document is log4js, a similar approach, designed for JavaScript, and altered specifically for the Adobe Forms environment.

Here is the page describing it: http://log4js.sourceforge.net/, and here is the documentation: http://log4js.sourceforge.net/jsdoc/ .

Several alterations have been made, including removal two loggers. The "popup logger" was removed since popups in the Adobe Forms Environment are application Modal (the application underneath the popup cannot be used while the popup is present), making the popup logger the same as the alertLogger. The writeLogger was removed because in the Adobe Forms Environment, there is no place to write messages to (Reader/Acrobat is not allowed to write to files on your hard drive). This leaves the **alertLogger** and the **consoleLogger** for your use in the Adobe Forms Environment.

A TRACE level and Log.trace(…) method added.

Other alterations were made to make the classlets and constants herein behave like the PureMVC classlets, added to the xfa model.

To instantiate a logger you use something like this:

```
xfa.mvcLogger = new xfa.Log(xfa.Log.TRACE, xfa.Log.consoleLogger, "pureMVC");
```

To send a log message, refer to the logger instance you created ,above, something like this:

```
xfa.mvcLogger.trace("calling ApplicationFacade startup");
```

To change the logging level (the level of messages that the logger will actually report):

```
xfa.mvcLogger.setLevel("TRACE");
```

where "TRACE" is case insensitive, and can be any of the levels listed two paragraphs below.

Note the use of "`xfa`" in references to your logger instance, and to the classlet, and to its constants, making them globally available in a manner that is trusted to stay existing throughout the form session.

Here are the constants you can use when creating a logger instance, determining which level of messages that logger instance will report:

`xfa.Log.TRACE` - reports all messages
`xfa.Log.DEBUG` - reports all messages except trace
`xfa.Log.INFO` - reports all messages except debug and trace
`xfa.Log.WARN` - reports all messages except info, debug and trace
`xfa.Log.ERROR` - reports only error and fatal messages
`xfa.Log.FATAL` - reports only fatal messages
`xfa.Log.NONE` - reports no messages at all

Having defined your logger instance with one of the above constants, you send a message with one of these, determining the message level of the message (replacing `[myLogger]` with the name of the logger instance you created with one of the above constants):

```
xfa.[myLogger].trace("this is a trace level message");
xfa.[myLogger].debug("this is a debug level message");
xfa.[myLogger].info("this is a info level message");
xfa.[myLogger].warn("this is a warn level message");
```

```
xfa.[myLogger].error("this is a error level message");
xfa.[myLogger].fatal("this is a fatal level message");
```

There are a couple of qualities in this particular implementation of log4js worth noting:

SERVER AWARE:

Regardless of which logger you decide to use (alertLogger or consoleLogger) if the form finds itself running on Adobe LiveCycle Server (eg. Adobe LiveCycle Forms, or Adobe LiveCycle Output) then the your log messages will automatically show up in the application system log instead.

MULTIPLE CONTEXTS:

The home page for this implementation log4js shows an example of use, where three instances of the logger are created. The point here is the idea that you might create different instances of the logger for different contexts. For example, to debug one aspect of your application, a logger instance could be created. To debug another aspect, a different logger instance could be created. Then your messages identify with an optional "prefix" which instance of the debugging system is sending you your messages.

NOTE: the log4js docs are not too clear on this point: when creating a logger instance, you can supply as a third parameter to instantiation (to the constructor) a "prefix" which will be added by this instance to your log messages, to identify in the console or alert, which context (which instance) it is that is sending you this message.

OBJECT DECONSTRUCTOR:

Another notable aspect of this log4js API is the static dumpObject method also shown in the example on the home page for this implementation of log4js. (Being a static method, use `xfa.Log.dumpObject(obj)`.) It deconstructs a JavaScript object, showing in string form, each of the members, each indented according the depth within the object that child members reside. This has been altered to be more aware of the Adobe Forms Environment, giving the type and full SOM name of objects encountered in the deconstruction. So you could include a call to this method in the construction of your log message string, to see inside a form design object.

DESCRIPTIVE EXCEPTION HANDLING:

When an exception or error occurs you can have a debug window appear, that includes a console to which error messages are posted. Usually, these error messages do not give you

much information. You can get better information by using this kind of error handling:

```
try{
... your code ...
} catch (_er) {LOG4JS.handleError(_er);}
```

or optionally include a string that describes what you were trying to do, for example:

```
} catch (_er) {LOG4JS.handleError(_er, "Trying to divide by zero");}
```

This will give a better description of what happened, along with a stack trace which you don't normally get. This stack trace is PureMVC aware and Adobe Forms aware. This means the stack trace names the Script Object, classlet, and method, or form-event handler along with the line number of the script throwing the exception. You also get parameters supplied to each function or method called, and this is Adobe Forms Environment aware, so that when a form design object is passed as an argument, you get the name of the form design object shown. In addition, this code is Adobe LiveCycle Server aware. It means that if you run this form on the server, the error messages will automatically appear on the application server system log instead, along with much more information than you typically get from Adobe LiveCycle Forms, or Adobe LiveCycle Output.

The suggestion is that you bracket *all* of your functions and methods in a try/catch block as shown above. But this is not to deter you from nesting additional, more specific try/catch blocks with descriptions of what you were trying to do, as in the second example shown above.

Here is an example of what you get if you *do not* surround your methods and functions with the try/catch suggested:

```
xyz is not defined
52:1
ReferenceError: xyz is not defined
52:1
```

Notice that you have only a line number to go on, as an indication of where in your application the exception was thrown. You don't know whether it was in a Script Object, or a form-event handler, and you don't know how execution got to that erroneous line.

Here is what you get instead, if *you do* surround your methods and functions with the try/catch suggested:

```
FATAL:Exception Handler
 - ReferenceError: xyz is not defined
Stack:

line: 52 of MVC_PROXIES.LoginProxy.login(LoginVO)

line: 103 of MVC_COMMANDS.LoginCommand.execute(butLoginClicked)

line: 34 of
MVC_MEDIATORS.LoginPanelMediator.handleNotification(click_form_event_PRIVATE_only)

line: 213 of MVC_MEDIATORS.DispatchEvent(checkthis.pagex.butLogin,
'click_form_event_PRIVATE_only')

line: 1 in the click event of: checkthis
```

Notice that not only is a call-stack offered, but in it, the name of the script object, the name of the classlet, the name of the function, and the input parameters to the function are all listed along with the line numbers. You can see all the relevant functions or methods that were called, right from the originating "click" form-event of a button called "checkthis". You also see parameters supplied to each function or method called, be it string or be it the name of an object.

SUPPLIED LOGGERS

In the files provided with this document, there are already three loggers defined. They are defined like this:

```
xfa.mvcException = new xfa.Log(xfa.Log.FATAL, xfa.Log.consoleLogger, "Exception
Handler");

xfa.mvcLogger = new xfa.Log(xfa.Log.WARN, xfa.Log.consoleLogger, "pureMVC");

xfa.appLogger = new xfa.Log(xfa.Log.WARN, xfa.Log.consoleLogger, "APP");
```

The first one, `xfa.mvcException` is used for the try/catch blocks described in this section, under the heading of DISCRIPTIVE EXCEPTION HANDLING. Note that the logger is set to report only `FATAL` level messages, which is the level that is used in the try/catch block. This logger is not intended for any other use, as the prefix: "Exception Handler" implies.

The second one, `xfa.mvcLogger`, which is set to report messages of WARN or worse, is intended for debugging or investigating the operation of the PureMVC framework in your form. If you want to see mediators, proxies and commands registered and initialized, you can set this logger level to report `TRACE` level messages. There is a line in the "SCRIPTS"

Script Object that you can uncomment for this purpose.

The third pre-defined logger is `xfa.appLogger`, and it is also set to report messages of WARN or worse. It is intended for your own use in debugging your own form application. You can change the report level from `WARN` to any of the other reporting levels by using the `setLevel` method of the logger. There is a line in the "SCRIPTS" Script Object that you can uncomment for this purpose.

Re-usable Custom Components

Custom components created by the form designer can participate in the PureMVC pattern too. Custom Components are typically UI components, that have an Initialize form-event where you can add scripting.

To make a custom component participate in PureMVC simply add a stewarding Mediator and a registration for that Mediator to the Initialize form-event for the custom component. An example is provided in the attached file: 'test_re_use_button.xfo'.

You must enclose this script in a run-once enclosure as all the MVC_ Script Objects have.

Each of these enclosures has a name associated, which is checked to ensure that the enclosed script is run only once. What name should you assign to your custom component enclosure, to avoid name-conflicts with others? Clearly this is a use for an UUID.

IMPORTANT: Most UUIDs have dashes in them: 3e91a1d0-0ac7-470e-83c2-3a08fea6fffa Those dashes will not be interpreted properly by JavaScript in this use, so replace all of them with underscores (do not use dots either.: 3e91a1d0_0ac7_470e_83c2_3a08fea6fffa.

The enclosure , in the initialize form-event for the re-usable custom component, therefore looks like this (copied from one of the MVC_ Script Objects:

```
(function(scope){if (null==scope)scope=xfa;if
(xfa.a4ffe371_177b_4097_adb8_42bd3ce9c1fa){return;}xfa.puremvc.define({name:
'a4ffe371_177b_4097_adb8_42bd3ce9c1fa',scope: scope});

[your Mediator and registerMediator here]

})(xfa);
```

Notice that the UUID chosen is place in two places, where once it is used as an object name,

and the other time it is used as a string.

Recall that for a stewarding mediator, a second parameter to the registerMediator should be supplied, which is the object that the Mediator stewards (receives private form-events). Because this script is in the 'initialize' form-event of the object being stewarded, you might be tempted, as I was, to use "this" as the object in the second parameter:

```
xfa.facade.registerMediator(new
xfa.Re_use_button_Mediator('Re_use_button_Mediator',this));
```

Using "this" here doesn't work. You must use ctrl-shift-click to place the full somExpression of the stewarded in the registerMediator statement:

```
xfa.facade.registerMediator(new
xfa.Re_use_button_Mediator('Re_use_button_Mediator',checkthis.pagex.Button1));
```

Now you can add script to the Mediator, that participates as an actor in PureMVC, using global constants to send Notifications to Commands or other Mediators, or retrieving Proxies to perform data operations.

You might want to take the extra effort to move the Mediator into the MVC_MEDIATORS Script Object, and the registerMediator statement into the MVC_COMMANDS Script Object, in order to keep your code centralized and organized. If you do not, the component will still work as-is. If you do move them, remember NOT to move the run-once enclosure into those script objects.

Form Design Fragments

There are two main aspects to Form Design Fragments, two things that might be contained in a fragment:

- UI components
- Script objects

UI components are difficult to make participate in PureMVC, because you would need to include stewarding Mediators for each UI component, which would normally be customized to make participate as an actor with the rest of the application.

Script objects might contain Mediators, Commands, Proxies, and Constants. All but the Consntants would need to be altered to participate a PureMVC actors with the rest of the application.

At the time of this writing, no answer to this dilemma has been found, therefore, at this time, it is not recommended that Form Design Fragments be used in the PureMVC Adobe forms.

Appendix A – Why PureMVC for Adobe Forms - Features

- Completely object oriented scripting
- MVC means better functional separation of code function for easier maintainability
- The Form-Data and Model Data have been de-coupled. The View has its own data model, while the Model has a separate one. While still allowing the Designer GUI binding and Designer binding types, the View might display one thing, while the Model's Proxy has decided to provide a different thing for save or submit.
- Bubbling events. Form-event propagation is not the same thing. The former can be handled by every object ancestor, the latter fires only for one child of the object doing propagation.
- Public Notifications, easier to manage and maintain than local events.
- While not part of PureMVC, Log4js is included thanks to the classlet definition API in PureMVC, automatically directing to an app server system log when the script runs on a LiveCycle server.
- While not part of PureMVC, a Collection and Dictionary classes have been included, with special capabilities specifically for Adobe Forms, thanks to the classlet definition API in PureMVC.
- While not part of PureMVC, extended error messaging with stack trace has been included, if you use try/catch with the included LOG4JS.handleError(e) function. Works great when running on LiveCycle server or on Acrobat/Reader. This has been enhanced for Adobe Forms, to provide a call stack trace that names Script Objects, Classes and function names besides line numbers. You also see parameters supplied to each function or method called, be it string or be it the name of an object.
- This implementation for Adobe Forms includes separation of View Data from Model Data, so that Mediators work with View Data (such as rawValue of a form design field) and the FormDataProxy supplied works with the Model Data that gets saved and submitted by the form. The FormDataProxy allows other Proxies and other actors to work with the Model Data that gets saved and submitted by the form.
- This implementation for Adobe Forms extends the Exception Handling to provide a detailed stack trace including Script Object name, classlet name (if applicable), function

or method name, besides line number and descriptive version of the Exception itself. You also see parameters supplied to each function or method called, be it string or be it the name of an object. This is provided if you wrap all of your functions and methods in a particular try/catch block. (See DESCRIPTIVE EXCEPTION HANDLING: on page 65)

Appendix B – Disadvantages to PureMVC for Adobe Forms

- Learning Curve for those not familiar with the MVC pattern.
- Coding requires coordination among several Script Objects: among Model, View, Controller code and string constants.
- A coding error often causes Reader/Acrobat to freeze.

Appendix C – Attachments in PDFs

Some people are unaware that PDF documents may contain attachments. This PDF file contains an attachment, which is a working example of the login subform discussed in the section on Mediators.

When looking at a document in Reader, you will see a paper-clip icon in the left display column, or in the extreme lower-left corner. Clicking on this paper-click gives you access to attachments.

Simply right-click on the attachment, and Save-As, to extract the attachment, and in this case, open the Login.xdp file in Adobe LiveCycle Designer. You can save the xdp as a PDF and open that in Reader to try the application.

The purpose of Login.xdp is to provide an example of PureMVC implemented in an Adobe Form. The example does not actually communicate with a server for actual login. Communication with a server is simulated by the LoginCommand in the MVC_COMMANDS Script Object.

Appendix D – Files Associated With This Document

puremvc-1.1.xfa.js

Not-minified version 1.1 of content of the MVC_PUREMVC Script Object. If you've got a form that has the minified version, replace the content of that Script Object with this if you want to inspect it, debug it, or change it (though this is not intended to be changed on a form-by-form basis). Minified elsewhere with Google's `comple.bat` (see below). RENAME .JSS TO .JS

OO_UTIL.js

Not-minified version of the OO_UTIL Script Object. If you've got a form that has the minified version, replace the content with this if you want to inspect it, debug it or change it (though this is not intended to be changed on a form-by-form basis). Minified elsewhere with Google's `compile.bat` (see below). RENAME .JSS TO .JS

log4js.js

Not-minified version of the LOG4JS Script Object. If you've got a form that has the minified version, replace the content with this if you want to inspect it, debug it or change it (though this is not intended to be changed on a form-by-form basis). Minified elsewhere with Yahoo's `compileyui.bat`, because Google's `compile.bat` creates too many nested conditional ? : statements for the Adobe Forms Script interpreter. (see below for `compileyui.bat`.) RENAME .JSS TO .JS

FormDataProxy.js

The MVC_PROXIES Script object might have this classlet minified in the MVC_PROXIES Script Object. If you've got a form that has this classlet minified, replace the content with this if you want to inspect it, debug it or change it (though this is not intended to be changed on a form-by-form basis). Minified elsewhere with Google's `compile.bat` (see below). RENAME .JSS TO .JS

DispatchEvent.js

The MVC_PROXIES Script object might have this classlet minified in the MVC_PROXIES Script Object. If you've got a form that has this classlet minified, replace the content with this if you want to inspect it, debug it or change it (though this is not intended to be changed on a form-by-form basis). Minified elsewhere with Google's `compile.bat` (see below). RENAME .JSS TO .JS

compile.bat and compiler.jar

`compile.bat` runs `compiler.jar` accepting two command-line parameters: input .js file, and output file. Minifies JavaScript code using Google's minifier. Makes script intentionally unreadable and smaller, where that script is not intended to be altered on a form-to-form basis. RENAME .BBB TO .BAT.
NOTE: compiler.jar was NOT included with this document because it is 5.7MB in size, making this document too large. Download it from Google's developers site.

compileyui.bat and yuicompressor-2.4.7.jar

`compileyui.bat` runs `yuicompressor-2.4.7.jar` accepting two command-line parameters: input .js file, and output file. Minifies JavaScript code using Yahoo's minifier. Makes script intentionally unreadable and smaller, where that script is not intended to be altered on a form-to-form basis. This is used for the LOG4JS Script Object because in this case, Google's minifier creates too many nested conditional ? : statements for the Adobe Forms interpreter. RENAME .BBB TO .BAT. RENAME .JRR TO .JAR

Login.xdp, Login.pdf

An example PureMVC Adobe Form. Demonstrates button handling and FormDataProxy use where a password entered on the form is changed to "CENSORED!" by the FormDataProxy in the data that is saved and submitted by the form. Demonstrates how to inspect the data saved and submitted by the form with an email submit button, where you choose "Internet email", then save the xml, and do not follow through with actually emailing it, but inspecting it instead. Demonstrates Global binding in two fields, and how that looks in the resulting saved or submitted XML where the field is not enclosed in any parent, even though one of the pair of globally bound fields is actually inside a subform.

passwordGen.xdp, passwordGen.pdf

An example PureMVC Adobe Form. Demonstrates implementation of a Proxy that manages data that has been added to the XML Source tab in Designer (2,000+ common English words).

pureMVCnotMin.tds

A Design "template" file that you can use to start a new form with, that is all set up with PureMVC. This one contains no minified script.

pureMVCmin.tds

A Design "template" file that you can use to start a new form with, that is all set up with PureMVC. This one contains minified script.

PureMVCmin.xfo

A re-usable component that you can add to your Object Library for dragging onto an existing form, in order to add PureMVC to that form. (see Appendix E.) This one contains minified script.

PurMVCnotMin.xfo

A re-usable component that you can add to your Object Library for dragging onto an existing form, in order to add PureMVC to that form. (see Appendix E.) This one does not contain minified script.

test_re_use_button.xfo

A re-usable component that you can add to your Object Library for dragging onto an existing form, demonstrating a custom component (a button) with scripting to make that button participate as an actor in the PureMVC application. The scripting it comes with is in the Initialize form-event for the button, and includes a stewarding Mediator and registerMediator statement, enclosed in a run-once wrapper, that was named with a UUID.

APPENDIX E – Adding PureMVC for JavaScript to an Existing Form

This document is accompanied by an additional file that allows you to easily add PureMVC and the other classlets described here, to previously existing forms.

NOTE: You must make your form design compatible with Acrobat/Reader 9.1 or later.
File -> Form Properties -> Defaults -> Choose Target Version.

There are two main changes that must be made:

- add Script Objects to your existing form
- add JavaScript to your existing form

This Appendix is concerned with the first of these two. Appendix F is concerned with the second.

Besides the .tds file (Design Template) included for starting from scratch, with PureMVC already installed, there is also an pureMVC.xfo custom library object that you can drag onto your form to add the features discussed in this document to your form.

Normally, a custom library object does not include Script Objects, but this one does. Actually, it consists only of a subform, that contains script objects.

After dragging it onto your existing form, you need to "unwrap Subform" to get the Script Objects out of the subform, and then, in the Hierarchy palette, drag the parent "variables" node that contains all the new Script Objects must below the word that names your top-most form object (usually called "form1").

Next, you need to add some JavaScript to your existing form that will activate the pureMVC framework on your existing form. That is the topic of Appendix F.

APPENDIX F – Adding JavaScript to an Existing Form to run PureMVC

NOTE: The instructions below, adding the interface between form-event handlers and PureMVC Notifications, where you change the form-event handlers on the top-level hierarchy object, cause the notifications to run BEFORE any of your own code that might be in the form-event handlers. If you want PureMVC Notifications to run AFTER your own code that might be in the form-event handlers, create an explicit MVC_MEDIATORS.DispatchEvent call at the end of your own form-event handler code. Eg: `MVC_MEDIATORS.DispatchEvent(this, xfa.appconstants.INITIALIZE_VEVENT);`

The first thing you must know, to add PureMVC to an existing form, is that the JavaScript you are about to add, is going to change all of the form-event handlers for the top-level form object in the hierarchy palette (usually called "form1"). The code you will add to all of these handlers connects the XFA model events to the PureMVC Notification system. For every form-event that fires, a PureMVC form-event will be send privately to the Mediator that manages the design component that generates the form-event.

So the first thing you must to is backup your form design file (xdp, or pdf). You are about to edit the XML Source, and this could permanently ruin your design if you make a mistake.

So the first thing you must do is remove to a backup location (cut/paste to notepad) any JavaScript you already have in any of the events for the top-level form object in the hierarchy palette. You will place this code back again later.

Once all the top-level item (form1) events are clear of any code, add this code to the "initialize" form-event:

```
var x = "find_me";
```

That should be the only code in any of the events for the top-level item.

Now open the XML Source tab for the existing form. Do a Find operation to get to "find_me".

You should see a "form-event" element that looks something like this:

```
      <event activity="initialize" name="event__initialize">
         <script contentType="application/x-javascript">var x = "find_me";
</script>
      </event>
```

Immediately below the </event> tag, and *before anything else*, paste the following lines (careful!!! paste it after the </event> tag, not after the </script> tag !!!):

```
<event activity="initialize" name="event__initialize" listen="refAndDescendents"><script
contentType="application/x-javascript">SCRIPTS.my_init();MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'initialize_form_event_PRIVATE_only');</script></event>
<event activity="indexChange" name="event__indexChange" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'indexChange_form_event_PRIVATE_only');</script></event>
<event activity="mouseEnter" name="event__mouseEnter" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'mouseEnter_form_event_PRIVATE_only');</script></event>
<event activity="mouseExit" name="event__mouseExit" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'mouseExit_form_event_PRIVATE_only');</script></event>
<event activity="change" name="event__change" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'change_form_event_PRIVATE_only');</script></event>
<event activity="full" name="event__full" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'full_form_event_PRIVATE_only');</script></event>
<event activity="mouseUp" name="event__mouseUp" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'mouseUp_form_event_PRIVATE_only');</script></event>
<event activity="mouseDown" name="event__mouseDown" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'mouseDown_form_event_PRIVATE_only');</script></event>
<event activity="click" name="event__click" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'click_form_event_PRIVATE_only');</script></event>
<event activity="enter" name="event__enter" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'enter_form_event_PRIVATE_only');</script></event>
<event activity="exit" name="event__exit" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'exit_form_event_PRIVATE_only');</script></event>
<event activity="preSign" name="event__preSign" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'preSign_form_event_PRIVATE_only');</script></event>
<event activity="postSign" name="event__postSign" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'postSign_form_event_PRIVATE_only');</script></event>
<event activity="ready" ref="$layout" name="event__layout_ready" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(xfa.form.form1,
'layoutReady_form_event_PRIVATE_only');</script></event>
<event activity="validationState" name="event__validationState" listen="refAndDescendents"><script
contentType="application/x-javascript">if (xfa.facade)MVC_MEDIATORS.DispatchEvent(xfa.event.target,
'validationState_form_event_PRIVATE_only');</script></event>
<event activity="preSubmit" ref="$form" name="event__preSubmit" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
'preSubmit_form_event_PRIVATE_only');</script></event>
<event activity="postSubmit" ref="$form" name="event__postSubmit" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
'postSubmit_form_event_PRIVATE_only');</script></event>
<event activity="ready" ref="$form" name="event__form_ready" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
'formReady_form_event_PRIVATE_only');</script></event>
<event activity="preSave" ref="$host" name="event__preSave" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
'preSave_form_event_PRIVATE_only');</script></event>
<event activity="postSave" ref="$host" name="event__postSave" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
'postSave_form_event_PRIVATE_only');</script></event>
<event activity="prePrint" ref="$host" name="event__prePrint" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
'prePrint_form_event_PRIVATE_only');</script></event>
<event activity="postPrint" ref="$host" name="event__postPrint" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
```

```
'postPrint_form_event_PRIVATE_only');</script></event>
<event activity="docReady" ref="$host" name="event__docReady" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
'docReady_form_event_PRIVATE_only');</script></event>
<event activity="docClose" ref="$host" name="event__docClose" listen="refAndDescendents"><script
contentType="application/x-javascript">MVC_MEDIATORS.DispatchEvent(this,
'docClose_form_event_PRIVATE_only');</script></event>
```

Now go back to "find_me" and delete the entire `<event>…</event>` element in which that appears, and go back to the "Design View" tab.

If you now click on the events "Show" drop-down in the Script Editor, for the top-most item in the Hierarchy palette, you should see that every form-event has an asterisk beside it. That is because you have placed a form-event handler for every form-event, with "form-event propagation" switched on, for every item in your design. Now the PureMVC Notification system will be aware of form events.

Now you must copy/paste the script you removed from the events in this top-level hierarchy element, back where they were. BUT when you go to these form-event handlers, you'll find script already there, that you've pasted into the XML Source view.

AND MOST IMPORTANTLY, "form-event propagation" has been turned on, so the script you paste back into the form-event handler will now run for EVERY object on the form! You probably only wanted it to run once.

So, past the existing script back into the form-event (before or after the sendNotification as you like), and then bracket your pre-existing code, with these lines, that will ensure that they run only once:

```
if ((!xfa.id) || (xfa.id == null) || (xfa.id.length==0)){   xfa.id="x";
      ... your original script, that will run only once ...
}
```

In that script, change "`id`" for each script segment (each form-event handler) to make them unique.

Now go back to the Design View Tab, and you're done. Whew.

APPENDIX G – Important Notes and Procedures

STEPS TO CREATE A NEW PUREMVC FORM

1. Start with one of the tds Forms Designer Templates provided with this document. This gets you started with the PureMVC Framework already installed and running. Delete the button that is present on that design.

2. Create your layout as usual.

3. Create your Data Connections as usual

4. Perform binding as usual

5. Add and register Mediators for your design components (described below).

   (Remember to encase *all* of your functions and methods in the try/catch blocks described in DESCRIPTIVE EXCEPTION HANDLING: on page 65.)

6. Add case statements to the FormDataProxy to over-ride default recording of View data, if desired. (described below)

7. Add and register other Proxies as needed (described below)

8. Add and register Commands as needed (described below)

ADDING A MEDIATOR

1. Copy and paste an existing Mediator into the "MVC_MEDIATORS" Script Object.

2. Change the "name" and "NAME" of the Mediator

3. Remove any methods and properties that got copied in your paste.

4. Add your own properties and methods (encased in try/catch blocks as described in DESCRIPTIVE EXCEPTION HANDLING: on page 65.)

5. If your Mediator is to steward a form design component, add to the switch statement, copied and pasted private form-event case statements from the bottom of the "MVC_MEDIATORS" Script Object.

6. In the "MVC_COMMANDS" Script Object, look for the "PrepareViewCommand" definition. Copy and paste a registerMediator statement, and adjust it for your own

Mediator. If your Mediator is to steward (catch form-events for) a form design component, be sure to include that object as the registerMediator second parameter, otherwise, delete or have not second parameter in the register Mediator statement.

FORMDATAPROXY

By default, this pre-defined Proxy creates a mirror data model to the one typically used in Adobe Forms, with the same structure and elements, then uses the "exit" form-event on every field to update the contents. If you wish to over-ride this default behavior:

1. Add a case statement to the switch in that Proxy, in the place indicated by comments. If you want to use the GUI binding techniques offered by Adobe Designer, then you can identify which component you want to over-ride data recording for, by using Ctrl-Shift-Click to place the complete SOM Expression in the case statement.

   NOTE: This technique has the down-side of requiring that you re-enter the SOM Expression whenever your form hierarchy changes.

2. You can also add your own methods to that Proxy, that can be used like any other Proxy to over-ride default behavior and otherwise manipulate the data that the form will save or submit. Remember to bracket all your functions and methods in try/catch blocks as described in DESCRIPTIVE EXCEPTION HANDLING: on page 65.

ADDING A PROXY

1. Copy/paste an existing Proxy into the "MVC_PROXIES" Script Object.
2. Change the "name" and the "NAME" of your new Proxy.
3. Remove any existing functionality that got copied
4. Add your own properties and methods (encased in try/catch blocks as described in DESCRIPTIVE EXCEPTION HANDLING: on page 65.)
5. In the "MVC_COMMANDS" Script Object, look for the "PrepareModelCommand" definition. Copy and paste a registerProxy statement, and adjust it for your own Proxy.

ADDING A COMMAND

1. Copy/paste an existing Command, of either SimpleCommand type, or MacroCommand type, into the "MVC_COMMANDS" Script Object.

2. Change the "name" and the "NAME" of your new Command.
3. Remove any existing functionality that got copied.
4. Add your own properties and methods (encased in try/catch blocks as described in DESCRIPTIVE EXCEPTION HANDLING: on page 65.)
5. In the "MVC_COMMANDS" Script Object, look for the "PrepareControllerCommand" definition. Copy and paste a registerCommand statement, and adjust it for your own Command. Remember that the *only* way a Command gets executed is by association with a Notification that has been broadcast. Put the name of the Notification that is to trigger this Command in your registerCommand statement.
6. Make sure the name of the Notification that triggers this Command is listed as a constant in MVC_CONSTANTS.

TIP: FUNCTION OR VARIABLE NOT DEFINED

Getting an exception for "function or variable not defined"? If you're calling the function or variable from within the same classlet, remember to prefix the method or variable name with "this."

TIP: COMMAND OR MEDIATOR NOT CATCHING NOTIFICATIONS

For a Command, was the associated Notification broadcast? Did you associate your Command with a Notification using a registerCommand statement in MVC_COMMANDS?

Did you put the name of the Notification in MVC_CONSTANTS?

For a Mediator, did you list the constant in "listNotificationInterests" in the Mediator definition?

Did you make the mistake of entering the name of a constant, in quotes? (It shouldn't be in quotes anywhere except in MVC_CONSTANTS.)

STATIC METHODS AND PROPERTIES

The third part of a classlet definition is static methods and properties. One typical static property is "NAME:". (You should make a habit of defining "NAME:" as a static property of all classlets you define.)

Static methods and properties are available for use with different syntaxes, depending on whether you're referring to the classlet definition, or an *instance* of the classlet. For example, a classlet called xfa.MyClass (notice the capital letter), might be *instantiated* thus: xfa.myClass = new xfa.MyClass(); (notice the lower-case).

To access a static method or property on the classlet definition itself, you use syntax like this: xfa.MyClass.NAME (notice capital MyClass, which refers to the classlet definition) and to access the same static method or property on the *instance*, you use syntax like this: xfa.myClass.constructor.NAME (notice the lowercase myClass which refers to the *instance*.)

So, when accessing a static method of property in an *instance* you need to include .constructor.

NOTE: by naming convention, class definitions begin with a capital letter, and class instances begin with a lower-case letter.

PREDEFINED OBJECTS

Sometimes your script needs to know the top-most hierarchy object in the form design. By default, this is called "form1" but if a Schema or Adobe Data Model or Example XML is assigned to the form design through a Data Connection, then "form1" is changed to the name of the root node in those. The form designer is also free to change "form1" to something else (unless one of those Data Connections is present). In any case, you might want to make your script independent or generalized, to refer to that top-most object, regardless of what it might be named.

You can get this top-most object here:
```
xfa.rootFormElement
```

For convenience, you can also get the root data element of the View data here:
```
xfa.rootDataElement
```

As mentioned earlier in this document, the root data element of the Model data (the data that gets saved and submitted, managed by the FormDataProxy described on page 48) is here:
```
xfa.datasets.MVCDataModel
```

INITIALIZE AND INDEXCHANGE BUG

Don't forget that because of a bug reported to Adobe in Acrobat/Reader version 9, which is still present in version 11.0.0, the propagation of these two form-events identifies the object that generates the form-event as 'null' instead of passing the object that generates the form-event.

So for every form-design object that you want to act on either of these two form-events, you must insert, into the form-event handler of the form-design object, an explicit call to DispatchEvent, as described on page 54:

For the `initialize` form-event:

```
MVC_MEDIATORS.DispatchEvent(this, 'initialize_form_form-event_PRIVATE_only');
```

and this line for the `indexChange` form-event:

```
MVC_MEDIATORS.DispatchEvent(this,'indexChange_form_event_PRIVATE_only');
```

As new versions of Acrobat/Reader are released, you can easily check to see whether the bug has been fixed, by simply uncommenting the indicated line in DispatchEvent, at the end of MVC_MEDIATORS. Uncommenting this line will display what object was sent by the propagated Initialize form-event. While the bug is still present, an alert box will show that "got nothing", and if the bug is fixed, the alert box will show the name of a form object that generated the initialize form-event.