Sandia National Laboratories

MVC Framework
Software Architecture Document

Version 1.1

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Revision History

Date	Version	Description	Author
10/17/07	1.0	Section(s) Changed: All Summary of Change(s): • Preliminary internal draft release.	Jonathan T. McClain, and Zachary O. Benz
06/25/08	1.1	Section(s) Changed: All Summary of Change(s): • Converted to use the RUP template. • Removed references to AMR and YM. • Pulled more information from the Data Trace Tool design document into this one.	Jonathan T. McClain

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the MVC Framework, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2 Scope

This document specifically applies to the library contained in the Sandia.MVCFramework namespace. It has direct influence on the functionality of the MVC Shell application, and the various MVC plug-ins that have been developed.

1.3 Definitions, Acronyms, and Abbreviations

MVC – Model, View, Controller

2. Architectural Representation

2.1 Design Overview

The MVC Framework will be constructed using a Model-View-Controller design pattern[1]. In this design pattern there are three primary components, as outlined below:

- Model – A domain specific representation of the data on which the application operates
- View – The user interface presented to the user, in which data from the model is rendered in a form suitable for interaction
- Controller – Processes and responds to events, typically user actions, and may invoke changes on the model and view

Control flow in this architecture proceeds as follows:

1. The user interacts with the user interface
2. The controller handles the input event from the user interface via a registered collection of actions (handlers)
3. The controller accesses the model to retrieve needed data and possibly updating it in a way appropriate to the user's action
4. The view updates to reflect the data returned to it via the controller
5. The user interface waits for further user interactions, which begins the cycle anew.

The MVC Framework will be written in C# against the .NET 2.0 framework and will operate in Windows 2000 and Windows XP desktop environments.

2.1.1 Model-View-Controller Architecture

The design of the MVC Framework is built around a model-view-controller (MVC) architecture in which visualizations, data representations, and request handling are all abstracted from one another. This allows, for example, for a visualization to be developed independently of how the raw data it depends on is actually stored or generated. It also facilitates a robust plug-in architecture, which is discussed in section 2.1.2.

[1] Description taken and adapted from http://en.wikipedia.org/wiki/MVC_Design_Pattern on 11/14/2006

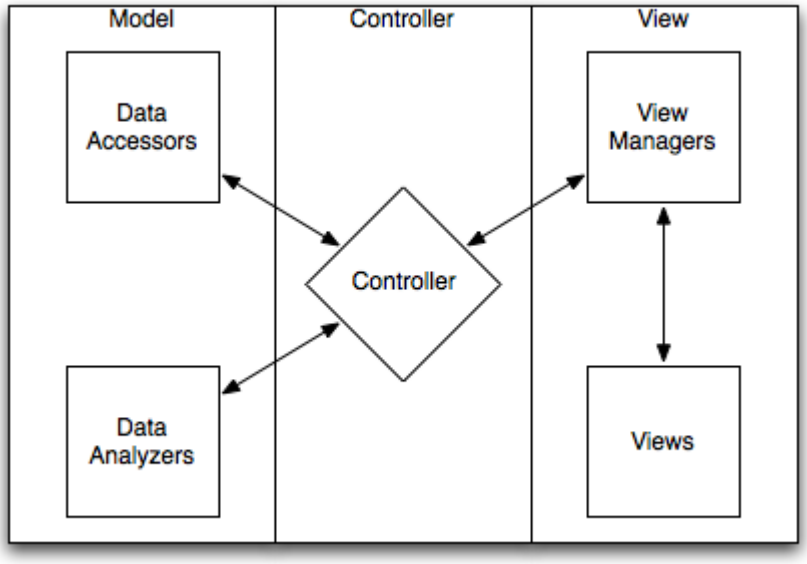

Figure 1: MVC Framework

The three main elements of the MVC Framework are:

- Data Accessors and Data Analyzers
- Views and View Managers
- A Controller

Data Accessors provide a standardized mechanism for retrieving raw data from data sources such as ODBC accessible data stores. Data Analyzers provide a mechanism for analyzing data within the scope of the application. In both cases the data types provided are registered with the Controller to facilitate servicing data requests from Views, as will be discussed below. DataAnalyzers are required to act as subscribers to the controller when they require data from data accessors.

Views are the actual visualizations (user interfaces) presented to the user. A collection of Views making up a particular mode of operation for the application are grouped together under the control of a View Manager. In this sense, a View Manager corresponds to one user interface window, with Views corresponding to elements of the window. Views or View Managers may make requests that the Controller then services; typically these will consist of requests for data based on a user action in the user interface.

The Controller directs the actions of the application based on inputs from the user in the form of requests from Views. When servicing a request for data, the Controller consults a table of mappings from requests to actions. Each request can map to multiple actions (one-to-many), where actions will typically consist of queries for data from either a Data Accessor or a Data Analyzer. If data is to be returned as part of handling a request, a publish-subscribe interface is used to make the data available to the Views. Each View subscribes (registers) for data packet types of interest with its View Manager, which in turn subscribes to the Controller. When the Controller publishes data, it consults this list of subscribers to determine where the data should be sent (published).

The publish-subscribe mechanism for data provides the following design advantages:

- Views can be kept synchronized regardless of which View originated the request
- ViewManagers, acting as a subscription intermediary between Views and the Controller, can mediate data publishing to member Views as needed

- The Controller is able to act on requests in a manner that is abstracted from how the results will be presented to the user by the View Managers and Views

The examples in the following sections are meant to further elucidate the operation of the MVC architecture as it applies to an application such as the AMR Exploration Tool.

In addition the functionality already discussed, the MVC Framework will also provide the following:

1. ExportTools – a suite of classes that allows for exporting data from an application, including in the form of reports and printouts of visualizations
2. Toolbar – an element of the user interface that provides buttons for the user to use to control the functionality of the application
3. DataStore – a place for settings, saved traces, and other persistent data storage needs to be kept across uses of the tool

2.1.2 Plug-in Architecture

The MVC Framework is designed to support plug-ins in order to quickly add on to the existing system when provided with new requirements. The core of this functionality revolves around two different interfaces, IPlugin and IController. This approach is characterized by allowing plug-ins to provide additional functionality to controllers through delegates.

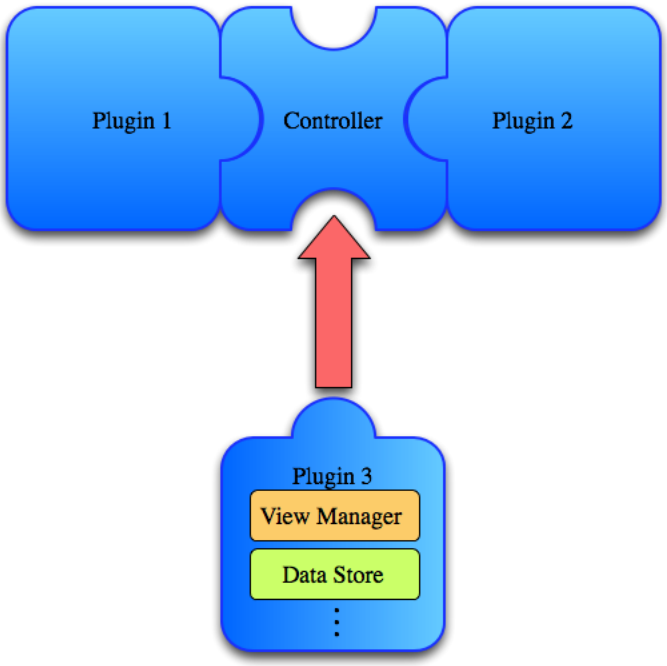

Figure 2: The MVC Framework Plug-in Architecture

2.1.2.1 Plug-in Versioning

All IPlugins will have a version number associated with it. This will help with backwards compatibility in the future, and with user support efforts by allowing the user to reference the specific plug-in version where there is a problem.

In addition, the plug-in architecture itself will also have a version number, which individual plug-ins will reference. This will provide support for future upgrades to the plug-in architecture by allowing for backwards compatibility with plug-ins that use earlier versions of the architecture.

2.1.2.2 IPlugin Functionality

The IPlugin interface defines the core functionality of a plug-in within the MVC architecture. A plug-in can consist of any combination of components within the MVC framework (e.g. data accessors, data analyzers, view managers, etc.). A few capabilities will not be supported by the plug-in architecture, these include the controller (since the controller contains the logic for managing plug-ins), the PluginLoader, and networking and print tools (although this may change in the future). All plug-ins have a RegisterPlugin method that a controller to call in order to initialize the plug-ins components. This method requires a controller to be passed in. From here, the plug-in adds its various components and delegate methods to the controller.

2.1.2.3 IController Functionality

The IController is responsible for managing requests within the system. Request handling will be defined as a collection of actions (e.g. delegate functions) associated with a particular requests. When the controller receives a request, it will identify the type of the request and execute the associated actions. Finally, the controller will collect the results of each of these actions and pass them on to subscribers. The ability to dynamically add action delegates to the system lends itself well to the plug-in system.

2.1.2.4 PluginLoader Functionality

The PluginLoader class is responsible for loading the plug-ins in the system on startup. This process consists of two parts, plug-in discovery and plug-in registration. Upon program initialization, the controller will pass itself to the plug-in loader via a method called RegisterPlugins. The plug-in loader is responsible for identifying plug-in assemblies (this will probably be done by having a plug-in directory or directories). Once an assembly is identified, the plug-in loader will use reflection to identify the IPlugins within the assembly. For each of these plug-ins, the plug-in loader will call RegisterPlugin and pass the controller to the plug-in. The plug-in then adds the new components and delegate functions to the controller.

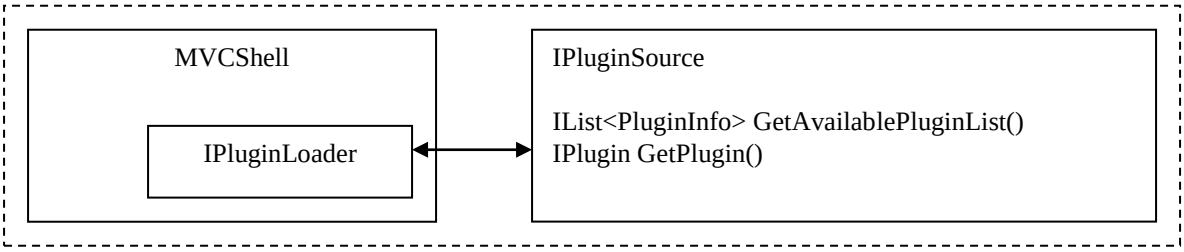

Figure 3 Plug-in Loader knows how to talk to a plug-in source to discover and load plug-ins. The plug-in loader is instantiated inside of the application. The plug-in source might be inside of the application (such as one that loads DLL's from file shares) or they might be on other machines (like in a web service).

3. Architectural Goals and Constraints

3.1 Design Conventions

We make use of several design conventions in the design of the system, as outlined below.

3.1.1 Interface, Abstract Class, and Implementation Classes

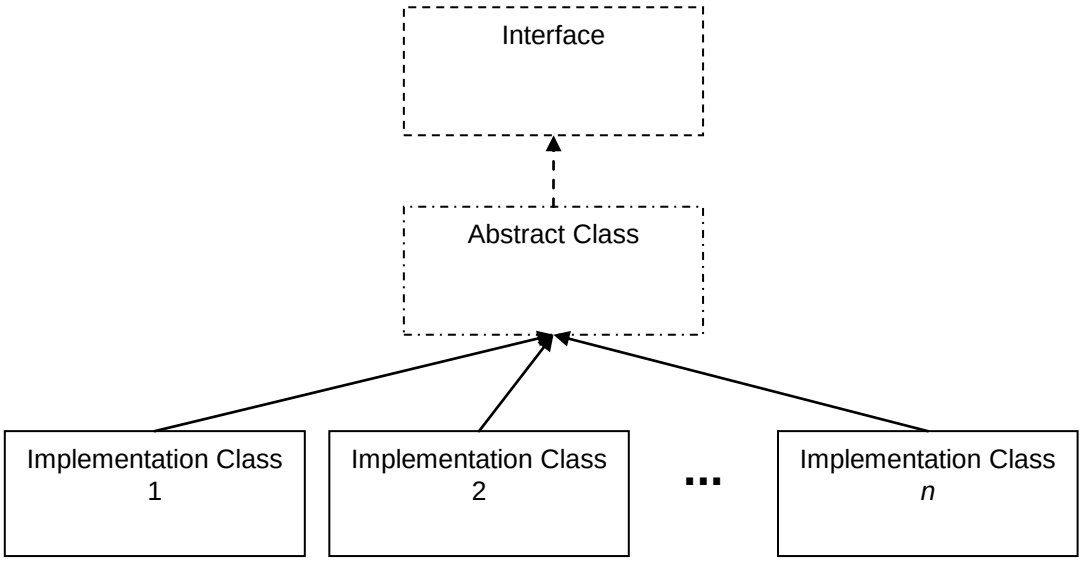

Figure 4: Class Design Pattern

The design convention of defining an interface, abstract class, and implementation class is used throughout the design of the MVC Framework. The convention is to break the design of a component into three parts: an interface, an abstract class, and implementation classes. The interface defines the functionality required of the component but contains no implementation of it. The abstract class contains a generally useful partial implementation of the interface. The implementation classes extend the abstract class to complete the implementation of the interface, typically for an application-specific purpose.

3.1.2 Default Implementations

The design convention of providing a default implementation for interfaces is also used throughout the design of the MVC Framework. The default implementation is intended to be a full implementation of the class that provides basic non-specialized functionality.

4. Use-Case View

4.1 Use-Case Realizations

4.1.1 Visualizing Raw Data

Consider a simple example of a single raw data source consisting of nodes in the tree and a single visualization displaying an interactive hierarchical tree.

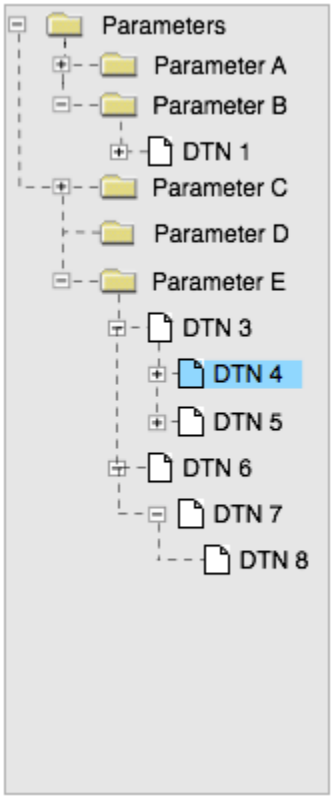

Figure 5: MVC Example – Visualizing Raw Data

At startup the View for the visualization sends a "get root node" request to the View Manager it is a member of. For this simple example the View Manager simply shuttles requests and published data between the View and the Controller. Beyond that it provides basic functionality such as a menu bar.

The Controller, in turn, consults its mapping table to obtain a reference to the sequence of actions to be executed for the request. For this example there is only one action: query the DataAccessor for the data source and obtain the root node. The data returned by the data source is published in a data packet and is sent to all subscribers, which in this example consists of the single View (via the View Manager).

The View interprets the data it has received back in order to display the root node in the hierarchical tree. For here on out, each time the user clicks the disclosure box (the '+' or '-' box) next to a node in the hierarchical tree, starting with the root node, a "get all children for node" request is sent via the View Manager to the controller. This results in a similar sequence of events as for the "get root node" request, except that the retrieved data this time is a collection of nodes, rather than a single node.

4.1.2 Visualizing Analyzed Data

Expanding on the previous example, a Data Analyzer is added for the purposes of adding implicit links to the graph. Consider an expansion of the previous example where we add a Data Store with its own Data Accessor and we add a Data Analyzer for performing semantic analysis.

When servicing the "get all children for node" request the Controller now has two actions to take: the previously described query of the raw node Data Accessor, and the new query of the Data Analyzer for related DIRS nodes that could be considered to be children.

The Controller passes on the request to the Data Analyzer, which in turn accesses the Data Store via its respective Data Accessor. The Data Analyzer performs the semantic analysis to dynamically discover documents that can be considered to be children of the parent node. The complete collection of nodes is published in a data packet as in the previous example.

4.1.3 *Multiple View Managers*

The role of the View Manager is to manage a collection of views as well as to provide core user interface services such as menus, view arrangement, and managing interface state persistently across application launches. Using multiple View Managers allows the user to have different windows up at the same time for different tasks.

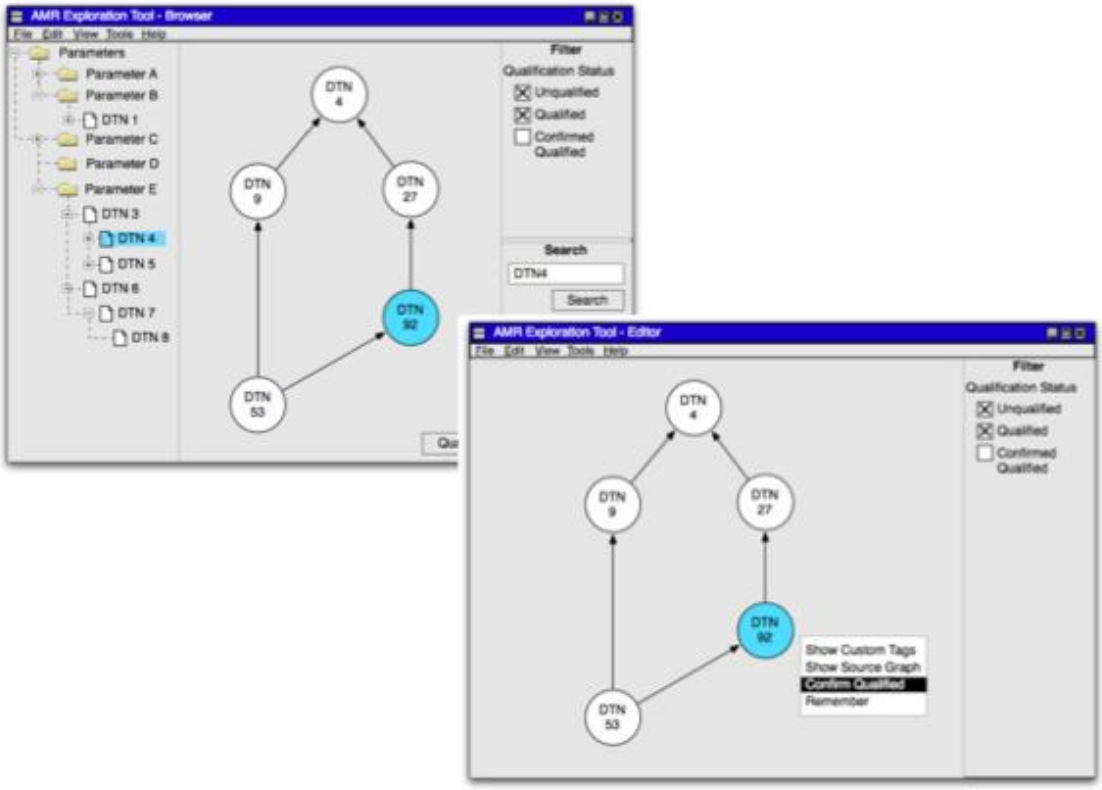

Figure 6: MVC Example - Multiple View Managers

Each View Manager embodies the type of functionality described in the previous two examples. However each also has an independent state, providing for the ability to conduct multiple simultaneous traces.

View Managers also mediate data flow to their member Views. For example, one View could be locked (i.e. its data subscriptions temporarily suspended) in order to provide the user with a static view of important information while other Views are still being dynamically updated with published data.

4.1.4 *Plug-in Example 1*

An example plug-in is shown below. In this example, the plug-in adds a new node type called STNNode to the system. No new views are added, since other views support the general Node type, however a view could have been added that only displays STN nodes. The plug-in adds a new data accessor to the

controller for the STN data store, and also adds a request delegate to handle requests for nodes.

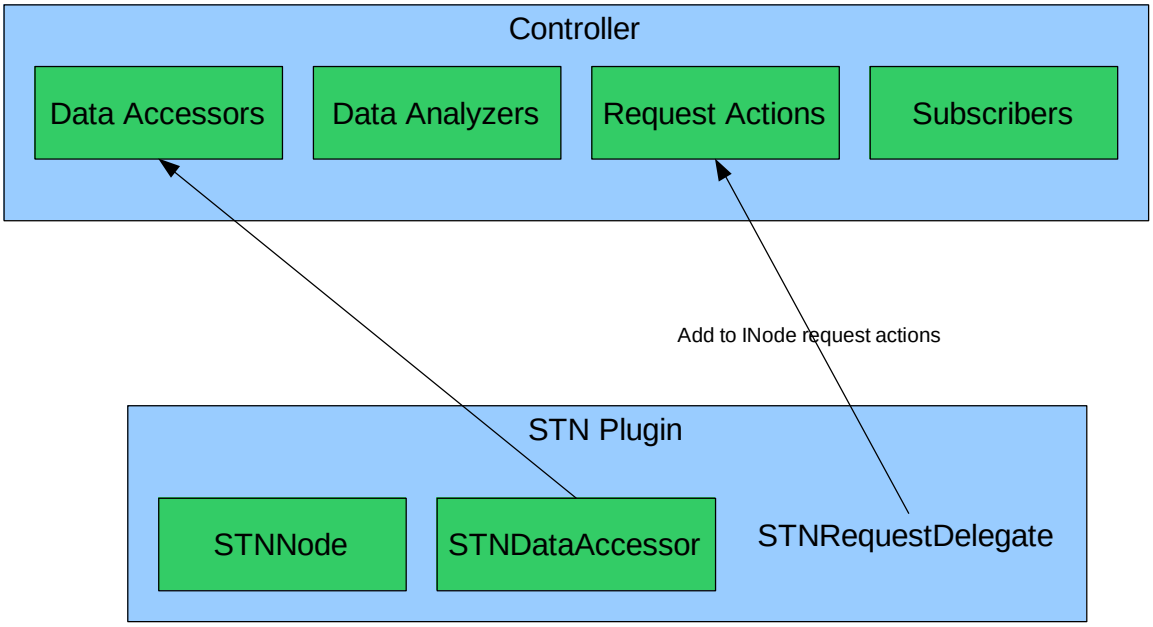

Figure 7: Plug-in Example 1

4.1.5 Plug-in Example 2

Another example plug-in is shown below. In this case, the plug-in adds a data type, data accessor, and a data analyzer. If the analyzer needs information from a data accessor, it is required to act as a subscriber and submit a request to the controller.

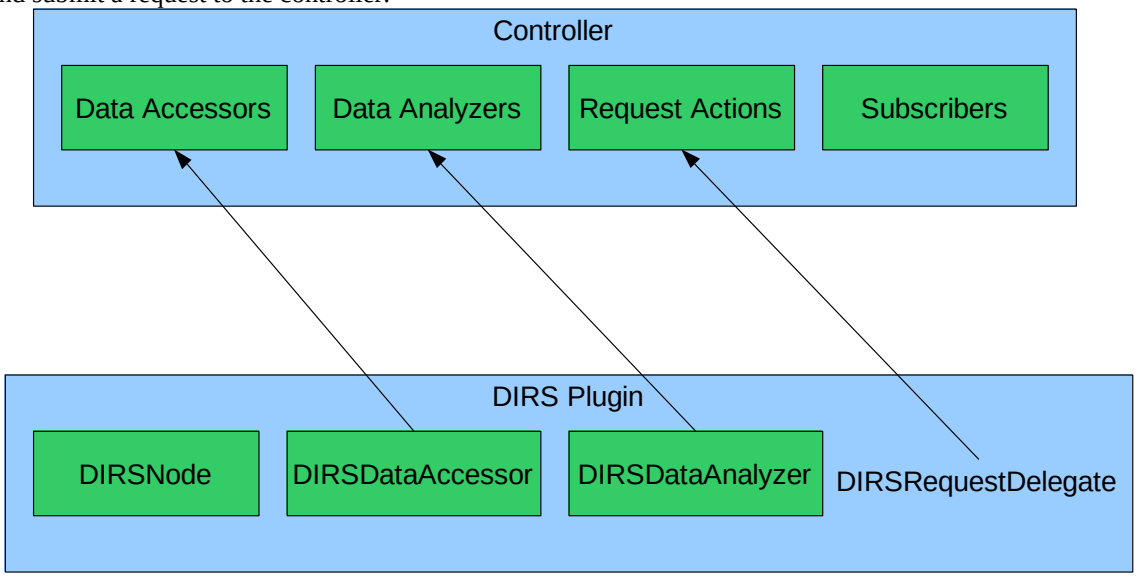

Figure 8: Plug-in Example 2

5. Logical View

5.1 Overview

Each part of th MVC Framework is characterized by a few important modules. Each of these modules

correspond to one of the primary components of the MVC Framework architecture. What follows is a brief description of each of these modules along with a description of their most important classes and interfaces.

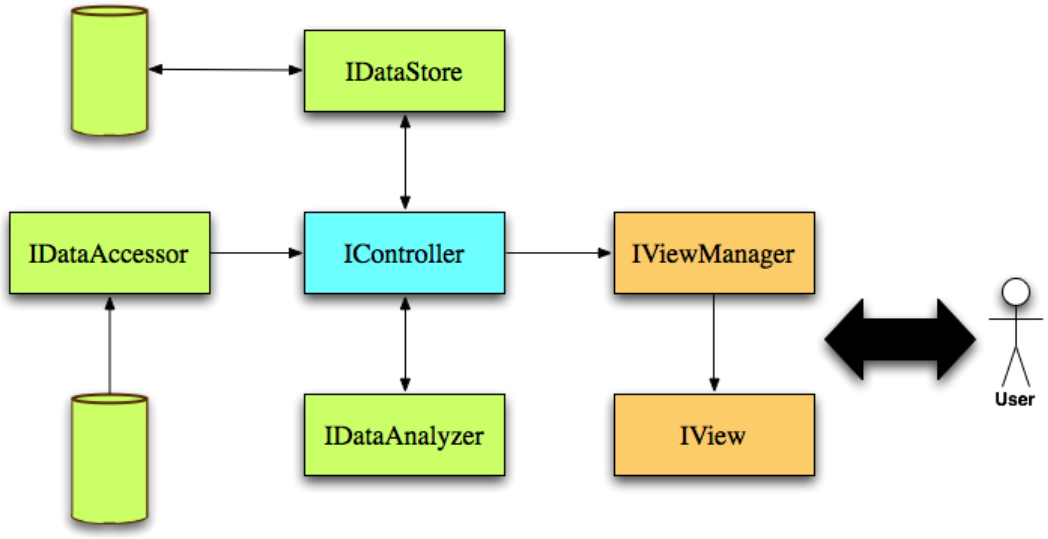

Figure 9: The primary components of the MVC Framework

5.2 Architecturally Significant Design Packages

5.2.1 Sandia.MVCFramework.Common

5.2.1.1 IRequestHandler

The IRequestHandler interface provides functionality for handling requests from ISubscribers within the MVC system.

Inherits

- *None*

Implements

- *None*

Properties

- *None*

Methods

- `void HandleRequest(IRequest request)`
  Handles a request from an ISubscriber.

5.2.1.2 ISubscriber

The ISubscriber interface provides functionality for subscribers within the MVC system.

Inherits

- *None*

Implements

- *None*

Properties

- *None*

Methods

- `void HandlePublishedData(DataPacket<IData> dataPacket)`
  A method for receiving and handling published data.

5.2.1.3 ISubscriptionHandler

The ISubscriptionHandler interface provides functionality for handling subscriptions within the MVC system.

Inherits

- *None*

Implements

- *None*

Properties

- *None*

Methods

- `void HandleSubscription(ISubscriber subscriber, Type dataType)`
  Handles a subscription request for the given data type and subscriber.

5.2.1.4 AbstractStandardSubscriptionHandler

The AbstractStandardSubscriptionHandler provides some general functionality for handling subscriptions within the MVC framework. In particular it provides the ability to handle subscription requests and publish to ISubscribers.

Inherits

- *None*

Implements

- `ISubscriptionHandler`

Properties

- *None*

Methods

- `AbstractStandardSubscriptionHandler()`
  Constructor for the AbstractStandardSubscriptionHandler class.
- `void PublishData(DataPacket<IData> dataPacket)`
  Publishes data to the ISubscribers that are subscribed to the given data packet data type.

5.2.2 Sandia.MVCFramework.Controller

5.2.2.1 IController

The IController interface provides the "controller" element of the AMR tool in the tool's model-view-controller design framework. The interface links data from the IDataAccessors (the "model") to the IViews presented to users based on user-directed control of the application.

Inherits

- *None*

Implements

- `IRequestHandler`
- `ISubscriptionHandler`

Properties

- `Collection<IDataAccessors> DataAccessors`
  The collection of IDataAccessors for the controller.
- `Collection<IDataAnalyzer> DataAnalyzers`
  The collection of IDataAnalyzers for the controller.
- `Collection<IViewManager> ViewManagers`
  The collection of IViewManagers for the controller.
- `Dictionary<IRequest, OnRequest> Actions`
  The mapping of requests to specific actions.

Methods

- *None*

5.2.2.2 AbstractController

5.2.2.3 AbstractStandardController (Reentrant)

An implementation of IController that provides general functionality for all controllers.

Inherits

- `AbstractStandardSubscriptionHandler`

Implements

- `IController`

Properties

- *None*

Methods

- *None*

5.2.3 Sandia.MVCFramework.Data

5.2.3.1 IData

The IData interface provides standard functionality for all data modules used within the MVC Framework.

Inherits

- *None*

Implements

- *None*

Properties

- *None*

Methods

- *None*

5.2.3.2 DataPacket

The DataPacket is a container class for IData modules. This class uses generics to limit the types of data an instantiated packet can hold.

Inherits

- *None*

Implements

- *None*

Properties

- `PacketID`
  A unique identifier for the packet.
- `Data`
  The data that the packet contains.
- `DataType`
  The type of the data that the packet contains.

Methods

- `DataPacket<IData>(IData data)`
  Constructor for the DataPacketClass.

5.2.4 Sandia.MVCFramework.DataAccessor

5.2.4.1 IDataAccessor

The IDataAccessor interface provides standard functionality for accessing data sources within the MVC Framework.

Inherits

- *None*

Implements

- *None*

Properties

- *None*

Methods

- `DataPacket<IData> RetrieveData(IRequest request)`
  Retrieves a data packet for the given request from the data source.
- `void ProcessData(IRequest request)`
  Process the data packet for the given request.

5.2.4.2 AbstractDatabaseDataAccessor (Reentrant)

AbstractDatabaseDataAccessor provides general functionality for data accessors that use a database as a data source.

Inherits

- *None*

Implements

- `IDataAccessor`

Properties

- *None*

Methods

- *None*

5.2.5 Sandia.MVCFramework.DataAnalyzer

5.2.5.1 IDataAnalyzer

The IDataAnalyzer interface provides a collection of routines for extracting additional information from

raw data provided by IDataAccessors.

Inherits

- *None*

Implements

- *None*

Properties

- *None*

Methods

- `DataPacket<IData> RetrieveAnalysis(IRequest request)`
  Retrieves a data packet that is analyzed based on the given request.

5.2.6 Sandia.MVCFramework.DataStore

5.2.6.1 IDataStore

The IDataStore interface provides functionality for modules that store (e.g. write to disk) information related to the MVC Framework.

Inherits

- *None*

Implements

- *None*

Properties

- *None*

Methods

- `void Save(DataPacket<IData> dataPacket)`
  Saves the given data into the data store.

5.2.7 Sandia.MVCFramework.ExportTools

5.2.7.1 IExporter

The IExporter interface provides functionality for modules that can be exported.

Inherits

- *None*

Implements

- *None*

Properties

- *None*

Methods

- `void Export(DataPacket<IData> dataPacket, IDataStore dataStore)`
  Exports the given data.

5.2.8 Sandia.MVCFramework.Plugin

5.2.8.1 IPlugin

The IPlugin interface provides functionality for modules that act as plugins within the MVC Framework.

Inherits
- *None*

Implements
- *None*

Properties
- *None*

Methods
- `void RegisterPlugin(IController controller)`
  Registers the plugin with the given IController.

5.2.9 Sandia.MVCFramework.Request

5.2.9.1 IRequest

The IRequest interface provides functionality for requests that are passed within the MVC Framework.

Inherits
- *None*

Implements
- *None*

Properties
- `IRequestParameter this[string parameterName]`
  Gets the request parameter associated with the given parameter name.
- `Dictionary<string, IRequestParameter> Parameters`
  A mapping of parameter names to request parameters.

Methods
- None

5.2.9.2 AbstractStandardRequest

Provides general functionality for requests within the MVC Framework.

Inherits
- *None*

Implements
- `IRequest`

Properties
- *None*

Methods
- `AbstractStandardRequest()`
  Constructor for the AbstractStandardRequest.
- `protected abstract void initParameters()`
  Called by the constructor to allow inheriting classes to initialize the parameters of the request on construction.

5.2.10 Sandia.MVCFramework.View

5.2.10.1 IView

The IView interface provides the functionality for displaying and manipulating views.

Inherits

- *None*

Implements

- `ISubscriber`

Properties

- *None*

Methods

- *None*

5.2.11 Sandia.MVCFramework.ViewManager

5.2.11.1 IViewManager

The IViewManager interface provides for management of a collection of IViews, typically into a single window, for presentation to the user. Different IViewManagers can reflect different modes of use of the AMR tool.

Inherits

- *None*

Implements

- `IRequestHandler`
- `ISubscriptionHandler`
- `ISubscriber`

Properties

- *None*

Methods

- `void SetViewManagerState(IViewManagerState viewState)`
  Sets the current state of the view manager.
- `IViewManagerState GetViewManagerState()`
  Gets the current state of the view manager.

5.2.11.2 AbstractStandardViewManager

AbstractStandardViewManager provides general functionality for IViewManagers based on the System.Windows.Forms.Form object. It includes an abstract initialization method that allows subclasses to add to the form, as well as basic menu functionality.

Inherits

- *None*

Implements

- `IViewManager`

Properties

- *None*

Methods

- `AbstractStandardViewManager(IRequestHandler requestHandler, ISubscriptionHandler subscriptionHandler)`
  Constructor for the AbstractStandardViewManager. Requires a request handler and a subscription handler for passing subscriptions and requests from its internal views.
- `protected abstract void setupViewManager()`
  Called by the contstructor in order to allow inheriting classes to add to the form on construction.
- `protected void addMenuItem(ToolStripMenuItem menuItem)`
  Adds a menu item to the form's menu strip.
- `protected void addControl(Control control, double percent)`
  Adds a control to the form, with the given percent being applied to the size of the control relative to the size of the form.