3 Model View Controller (MVC) architecture

3.1 Design patterns

As we discussed in the previous section, a design pattern describes a proven solution to a recurring design problem, placing particular emphasis on the context and forces surrounding the problem, and the consequences and impact of the solution. There are many good reasons to use design patterns:

1. *They are proven*: You tap the experience, knowledge and insights of developers who have used these patterns successfully in their own work.
2. *They are reusable*: When a problem recurs, you don't have to invent a new solution; you follow the pattern and adapt it as necessary.
3. *They are expressive*: Design patterns provide a common vocabulary of solutions, which you can use to express larger solutions succinctly.

It is important to remember, however, that design patterns do not guarantee success. You can only determine whether a pattern is applicable by carefully reading its description, and only after you've applied it in your own work can you determine whether it has helped. One of these patterns is Model-View-Controller (MVC). The programming language Smalltalk first defined the MVC concept it in the 1970's. Since that time, the MVC design idiom has become commonplace, especially in object-oriented systems.

3.2 The architecture

As we discussed in the previous section, it is common to think of an application as having three main layers: presentation (UI), application logic, and resource management. In MVC, the presentation layer is split into controller and view. The most important separation is between presentation and application logic. The View/Controller split is less so. MVC encompasses more of the architecture of an application than is typical for a design pattern. Hence the term architectural pattern may be useful, or perhaps an aggregate design pattern.

- *Model*: The domain-specific representation of the information on which the application operates. The model is another name for the application logic layer (sometimes also called the domain layer). Application (or domain) logic adds meaning to raw data (e.g., calculating if today is the user's birthday, or the totals, taxes and shipping charges for shopping cart items). Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the resource management layer because it is understood to be underneath or encapsulated by the Model.
- *View*: Renders the model into a form suitable for interaction, typically a user interface element. MVC is often seen in web applications, where the view is the HTML page and the code which gathers dynamic data for the page.

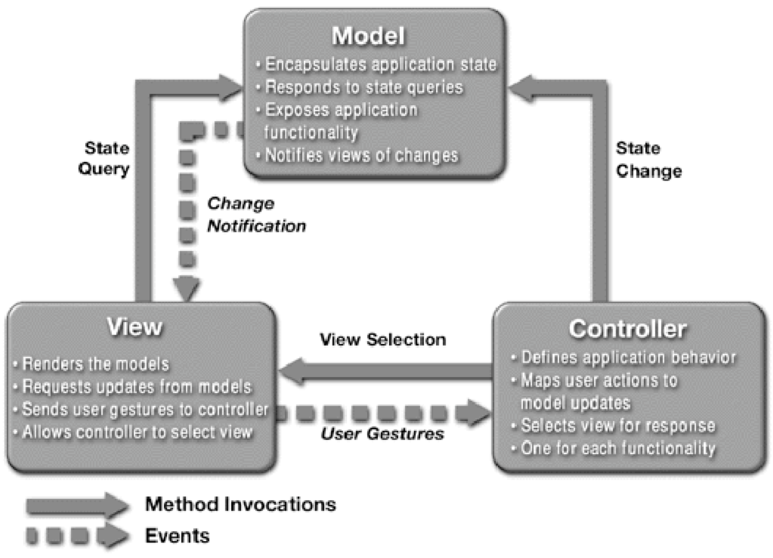
Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Figure 4: Model View Controller architecture

- *Controller*: Processes and responds to events, typically user actions, and may invoke changes on the model and view.

Though MVC comes in different flavours, the control flow generally works as follows:

1. The user interacts with the user interface in some way (e.g., user presses a button)
2. A controller handles the input event from the user interface, often via a registered handler or callback.
3. The controller accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller updates user's shopping cart). Complex controllers are often structured using the command pattern to encapsulate actions and simplify extension.
4. A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view. (However, the observer pattern can be used to allow the model to indirectly notify interested parties, potentially including views, of a change.)
5. The user interface waits for further user interactions, which begins the cycle anew.

Fig 4 summarises the relationship between the Model, View, and Controller is provided below. Note: the solid lines indicate a direct association, and the dashed line indicate an indirect association (e.g., observer pattern).

3.3 Implementations

The MVC pattern was first described in 1979 by Trygve Reenskaug, then working on Smalltalk at Xerox research labs. The original implementation is described in depth in the influential paper *"Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller"*.

Smalltalk's MVC implementation inspired many other GUI frameworks such as:

- The NeXTSTEP and OPENSTEP development environments encourage the use of MVC. Cocoa and GNUstep, based on these technologies, also use MVC.
- Microsoft Foundation Classes (MFC) (also called Document/View architecture)
- Java Swing
- The Qt Toolkit (since Qt4 Release).
- XForms has a clear separation of model (stored inside the HTML head section) from the presentation (stored in the HTML body section). XForms uses simple bind commands to link the presentation to the model.

The following sections provide a basic overview of how MVC is implemented in a number of different frameworks - don't worry if you don't follow all the details yet, as you may not be familiar with the frameworks. We shall be reviewing many of these frameworks in lecture 4.

3.3.1 ASP.NET

In ASP.NET, the patterns for the view and controller are well defined. The model is left to the developer to design.

- *View*: The ASPX and ASCX files handle the responsibilities of the view. With this design, the view object actually inherits from the controller object. This is different from the Smalltalk implementation, in which separate classes have pointers to one another.
- *Controller*: The duties of the controller are split between two places. The generation and passing of events is part of the framework and more specifically the Page and Control classes. The handling of events is usually done in the code-behind class.
- *Model*: ASP.NET does not strictly require a model. The developer has the option to create a model class, but may choose to forgo it and have the event handlers in the controller perform any calculations and data persistence. That said, using a model to encapsulate business rules and database access is both possible and preferable.

3.3.2 Windows Forms

In WinForms, a .NET framework, the patterns for the view and controller are well defined. The model is left to the developer to design.

- *View*: A class inheriting from either Form or Control handles the responsibilities of the view. In the case of WinForms, the View and Controller are compiled into the same class. This differs from ASP.Net, which uses inheritance, and Smalltalk, which have separate classes with pointers to one other.
- *Controller*: The duties of the controller are split between three places. The generation and passing of events starts at the OS level. Inside the .Net framework, the Form and Control classes route the event to the proper event handler. The handling of events is usually done in the code-behind class.
- *Model*: Just like ASP.Net, WinForms does not strictly require a model. The developer has the option to create a model class, but may choose to forget it and have the event handlers in the controller perform any calculations and data persistence. Again, using a model to encapsulate business rules and database access is both possible and preferable.

3.3.3 JSP Model 2 (MVC-2)

JSP Model 2 (or MVC 2) is Sun's attempt to wrap Java Server Pages (JSP) within the MVC paradigm. It's not so so much a product offering (or even an API) as it is a set of guidelines that go along with Sun's packaging of Java-based components and services under the umbrella of J2EE. The general structure of a Web application using the JSP Model 2 architecture is:

1. User requests are directed to the controller servlet.
2. The controller servlet accesses required data and builds the model, possibly delegating the processing to helper classes.
3. The controller servlet (or the appropriate sub-ordinate task) selects and passes control to the appropriate JSP responsible for presenting the view.
4. The view page is presented to the requesting user.
5. The user interacts with the controller servlet (via the view) to enter and modify data, traverse through results etc.

Data access and application logic should be contained entirely within the controller servlet and its helper classes. The controller servlet (or the helper class) should select the appropriate JSP page and transfer control to that page object based on the request parameters, state and session information. One of the major advances that comes with JSP Model 2 is Sun's specification of the Java Standard Tak Library (JSTL). It specifies the standard set of tags for iteration, conditional processing, database access and many other formatting functions. In addition to the guidelines associated with JSP Model 2, Sun also provided a set of blueprints for building application using the MVC paradigm. These blueprints were eventually renamed the *J2EE Core Patterns*. They are too numerous and complex to examine in detail here, but some of the more important patterns are described below:

- *Front Controller*: a module (often a servlet) acting as the centralised entry point into a Web application, managing request processing, performing authentication and authorisation services, and ultimately selecting the appropriate view.
- *Service-To-Worker* and *Dispatcher View*: strategies for MVC application where the front controller module defers processing to a dispatcher that is selected based on the request context. In the Dispatcher View pattern, the dispatcher performs static processing to select the ultimate presentation view. In the Service-To-Worker pattern, the dispatcher's processing is more dynamic, translating logical task names into concrete task module references, and allowing tasks to perform complex processing that determines the ultimate presentation view.
- *Intercepting Filter*: allows for pluggable filters to be inserted in the "request pipeline" to perform pre and post processing of incoming requests and outgoing responses. These filters can perform common services required for all or most application tasks, including authentication and logging.
- *Value List Handler*: a mechanism for caching results from database queries, presenting discrete subsets of those results, and providing iterative traversal through the sequence of subsets.
- *Data Access Object (DAO)*: A centralised mechanism for abstracting and encapsulating access to complex data sources, including relational databases, LDA directories and CORBA business objects. The DAO acts as an adapter, allowing the external interface to remain constant even when the structure of the underlying data sources changes.

3.4 Introducing Apache Struts

You may need to refer to some of the technology overview and defintions material in Chapter 4 to understand this section. In particular, ensure that you understand what Java Server Pages (JSP) are about. This section is intended to familiarise the reader with the basic concepts associated with the Struts framework. There is a Struts tutorial exercise that will deal in the lower level implementation details. You should read this section before looking at the tutorial exercise. If you intend installing the Struts framework on your own PC, be advised that we are using Struts v1.3.5. Struts 2 was only released in Autumn 2006 and is in beta form at the time of writing.

Apache Struts is a free, open-source framework for creating Java web applications developed by the Apache Software Foundation. Web applications differ from conventional websites in that web applications can create a dynamic response. Many websites deliver only static pages. A web application can interact with databases and business logic engines to customize a response.

Struts is highly configurable, and has a large (and growing) feature list, including a Controller, action classes and mappings, utility classes for XML, automatic population of server-side JavaBeans, Web forms with validation, and some internationalization support. It also includes a set of custom tags for accessing server-side state, creating HTML, performing presentation logic, and templating. Some vendors have begun to adopt and evangelise Struts. Struts can be considered an industrial-strength framework suitable for large applications. But Struts is not yet a "standard" for which J2EE product providers can interoperably and reliably create tools. The main attraction of the Struts framework is that developers can make use of configurable application components (e.g. the controller servlet) that come with the Struts distribution, instead of having to implement these components themselves. The whole application comes together with the XML configuration file names `struts-config.xml` that is located in the application's `WEB-INF` directory.

Web applications based on JavaServer Pages sometimes mingle database code, page design code, and control flow code. In practice, we find that unless these concerns are separated, larger applications become difficult to maintain. One way to separate concerns in a software application is to use a Model-View-Controller (MVC) architecture. The Struts framework provides a robust infrastructure for Model 2 application development using Front Controller and Service-To-Worker patterns to provide a true framework for Web application development. The Model represents the business or database code, the View represents the page design code, and the Controller represents the navigational code.

Be sure to understand that Struts is a MVC framework, not a container. A MVC framework can reside inside a web container, but the container in its most abstract form will not specify a particular design pattern, although some containers are implemented in such a way as to facilitate implementations using particular design patterns.

3.4.1 Basic implementation details

As Struts is intended to build MVC compliant web applications, we can consider the functionality the framework provides according to the MVC design pattern.

The Model portion of an MVC-based system can be often be divided into two major subsystems; the internal state of the system and the actions that can be taken to change that state. In grammatical terms, we might think about state information as nouns (things) and actions as verbs (changes to the state of those things).

Many applications represent the internal state of the system as a set of one or more JavaBeans. The bean properties represent the details of the system' state. Depending on your application's complexity, these beans may be self contained (and know how to persist their own state), or they may be facades that know how to retrieve the system's state from another component. This component may be a database, a search engine, an Entity Enterprise JavaBean, a LDAP server, or something else entirely.

Large-scale applications will often represent the set of possible business operations as methods that can be called on the bean or beans maintaining the state information. For example, you might have a shopping cart bean, stored in session scope for each current user, with properties that represent the current set of items that the user has decided to purchase. This bean might also have a `checkOut()` method that authorizes the user's credit card and sends the order to the warehouse to be picked and shipped. Other systems will represent the available operations separately, perhaps as Session Enterprise JavaBeans (Session EJBs). In a smaller scale application, on the other hand, the available operations might be embedded within the `Action` classes that are part of the framework control layer. This can be useful when the logic is very simple or where reuse of the business logic in other environments is not contemplated. The framework architecture is flexible enough to support most any approach to accessing the Model, but we strongly recommend that you separate the business logic ("how it's done") from the role that Action classes play ("what to do").

The View portion of a Struts-based application is most often constructed using JavaServer Pages (JSP) technology. JSP pages can contain static HTML (or XML) text called "template text", plus the ability to insert dynamic content based on the interpretation (at page request time) of special action tags. The JSP environment includes a set of standard action tags, such as `<jsp:useBean>` whose purpose is described in the JavaServer Pages Specification. In addition to the built-in actions, there is a standard facility to define your own tags, which are organized into "custom tag libraries."

The framework includes a set of custom tag libraries that facilitate creating user interfaces that are fully internationalized and interact gracefully with `ActionForm` beans. Action Forms capture and validate whatever input is required by the application.

Struts provides the Controller portion of the application. The Controller is focused on receiving requests from the client (typically a user running a web browser), deciding what business logic function is to be performed, and then delegating responsibility for producing the next phase of the user interface to an appropriate View component. The primary component of the Controller in the framework is a servlet of class `ActionServlet`. This servlet is configured by defining a set of ActionMappings. An ActionMapping defines a path that is matched against the request URI of the incoming request and usually specifies the fully qualified class name of an `Action` class. All Actions are subclassed from `org.apache.struts.action.Action`. Actions encapsulate calls to business logic classes, interpret the outcome, and ultimately dispatch control to the appropriate View component to create the response. While the framework dispatches to a View, actually rendering the View is outside its scope.

The framework also supports the ability to use ActionMapping classes that have additional properties beyond the standard ones required to operate the controller. This allows you to store additional information specific to your application and still utilize the remaining features of the framework. In addition, the framework lets you define logical "names" to which control should be forwarded so that an action method can ask for the "Main Menu" page (for example), without knowing the location of the corresponding JSP page. These features greatly assist you in separating the control logic (what to do) with the view logic (how it's rendered).

The framework provides several components that make up the Control layer of a MVC-style application. These include a controller component (servlet), developer-defined request handlers, and several supporting objects.

The Struts `Taglib` component provides direct support for the View layer of a MVC application. Some of these tags access the control-layer objects. Others are generic tags found convenient when writing applications. Other taglibs, including JSTL, can also be used with the framework. Other presentation technologies, like Velocity Templates and XSLT can also be used with the framework.

The Model layer in a MVC application is often project-specific. The framework is designed to make it easy to access the business-end of your application, but leaves that part of the programming to other products, like JDBC, Enterprise Java Beans, Object Relational Bridge, or iBATIS, to name a few.

Let's step through how this all fits together.

When initialized, the controller parses a configuration file (`struts-config.xml`) and uses it to deploy other control layer objects. Together, these objects form the Struts Configuration. The Configuration defines (among other things) the `ActionMappings` (`org.apache.struts.action.ActionMappings`) for an application.

The controller component consults the `ActionMappings` as it routes HTTP requests to other components in the framework. Requests may be forwarded to JavaServer Pages or `Action` (`org.apache.struts.action.Action`) subclasses provided by the application developer. Often, a request is first forwarded to an `Action` and then to a JSP (or other presentation page). The mappings help the controller turn HTTP requests into application actions.

An individual `ActionMapping` (`org.apache.struts.action.ActionMapping`) will usually contain a number of properties including:

- a request path (or "URI"),
- the object type (`Action` subclass) to act upon the request, and
- other properties as needed.

The `Action` object can handle the request and respond to the client (usually a Web browser) or indicate that control should be forwarded elsewhere. For example, if a login succeeds, a login action may wish to forward the request onto the mainMenu page. `Action` objects have access to the application's controller component, and so have access to that members's methods. When forwarding control, an Action object can indirectly forward one or more shared objects, including JavaBeans, by placing them in one of the standard contexts shared by Java Servlets. For example, an `Action` object can create a shopping cart bean, add an item to the cart, place the bean in the session context, and then forward control to another mapping. That mapping may use a JavaServer Page to display the contents of the user's cart. Since each client has their own session, they will each also have their own shopping cart.

Most of the business logic in an application can be represented using JavaBeans. An Action can call the properties of a JavaBean without knowing how it actually works. This encapsulates the business logic, so that the Action can focus on error handling and where to forward control.

JavaBeans can also be used to manage input forms. A key problem in designing Web applications is retaining and validating what a user has entered between requests. You can define your own set of input bean classes, by subclassing `ActionForm` (`org.apache.struts.action.ActionForm`). The `ActionForm` class makes it easy to store and validate the data for your application's input forms.

The `ActionForm` bean is automatically saved in one of the standard, shared context collections, so that it can be used by other objects, like an `Action` object or another JSP.

The form bean can be used by a JSP to collect data from the user by an `Action` object to validate the user-entered data and then by the JSP again to re-populate the form fields. In the case of validation errors, the framework has a shared mechanism for raising and displaying error messages.

Another element of the Configuration are the `ActionFormBeans` (`org.apache.struts.action.ActionFormBeans`). This is a collection of descriptor objects that are used to create instances of the `ActionForm` objects at runtime. When a mapping needs an `ActionForm`, the servlet looks up the form-bean descriptor by name and uses it to create an `ActionForm` instance of the specified type.

Here is the sequence of events that occur when a request calls for an mapping that uses an `ActionForm`:

- The controller servlet either retrieves or creates the `ActionForm` bean instance.
- The controller servlet passes the bean to the `Action` object. If the request is being used to submit an input page, the `Action` object can examine the data. If necessary, the data can be sent back to the input form along with a list of messages to display on the page. Otherwise the data can be passed along to the business tier.
- If the request is being used to create an input page, the `Action` object can populate the bean with any data that the input page might need.

The Struts Taglib component provides custom tags that can automatically populate fields from a JavaBean. All most JavaServer Pages really need to know is the field names to use and where to submit the form. Other tags can automatically output messages queued by an `Action` or `ActionForm` and simply need to be integrated into the page's markup. The messages are designed for localization and will render the best available message for a user's locale.

The framework and Struts Taglib were designed from the ground-up to support the internationalization features built into the Java platform. All the field labels and messages can be retrieved from a message resource. To provide messages for another language, simply add another file to the resource bundle. Internationalism aside, other benefits to the message resources approach are consistent labeling between forms, and the ability to review all labels and messages from a central location.

For the simplest applications, an `Action` object may sometimes handle the business logic associated with a request. However, in most cases, an `Action` object should invoke another object, usually a JavaBean, to perform the actual business logic. This lets the `Action` focus on error handling and control flow, rather than business logic. To allow reuse on other platforms, business-logic JavaBeans should not refer to any Web application objects. The Action object should translate needed details from the HTTP request and pass those along to the business-logic beans as regular Java variables.

In a database application, for example:

- A business-logic bean will connect to and query the database,
- The business-logic bean returns the result to the `Action`,
- The `Action` stores the result in a form bean in the request,
- The JavaServer Page displays the result in a HTML form.

Neither the Action nor the JSP need to know (or care) from where the result comes. They just need to know how to package and display it.

3.4.2 Struts and Enterprise Java Beans

This sub-section is a note for advanced programmers!! Don't worry if you don't understand the issue raised here yet. We have mentioned so far that results available in the Struts view layer are in the form of `ActionForm` beans. This is not quite the whole story. It is true that the view layer send user input data to the controller through `ActionForm` (or its variant) beans, but communication can also be accomplished by any request or session parameters. If you are using EJB 2.1 this can be dangerous, but in EJB 3.0 detached entity beans are just Plain Old Java Objects (POJOs) that are disconnected from the EJB persistence layer. Changing them does not change anything in the application back end. In case a subset of properties or a detached entity bean is required, it os good practice to create a value object, which is essentially a POJO containing only the properties required with their values copied from the entity bean.

3.5 References and further reading

Buschmann, F. ,Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996). "Pattern-Oriented Software Architecture". John Wiley and Sons. ISBN 0-471-95869-7.

Shklar, L. and Rosen, R. (2003). "Web Application Architecture: Principles, Protocols and Practices". John Wiley and Sons. ISDN 0-471-48656-6.

Bodoff, S., Green, D., Haase, K., Jendrock, E., Pawlan, M. and Stearns, B. (2002). "The J2EE Tutorial". Addison Wesley.

`http://www.oracle.com/technology//sample_code/tech/java/j2ee/jintdemo/tutorials/Struts.html`

`http://struts.apache.org/1.3.5/userGuide/index.html`