Model-View-Controller: A Design Pattern for Software

June 2004

Introduction

- Why focus on Model-View-Controller Architecture?
- What's a Software Design Pattern?
- Why should programmers care about Design Patterns?

Why focus on Model-View-Controller Architecture?

The topic of this presentation is MVC - Model-View-Controller - architecture in web applications. This topic was selected because it allows us to look at web appliation development in overview, before getting into the nitty gritty of the many component parts of implementing a web app.

MVC is a fundamental and high-level design pattern for programming in the J2EE architecture toward which the UC Berkeley campus is gravitating in response to the E-Architecture guidelines established by the campus's Information Technology Architecture Committee (ITAC) in 2001.

What's a Software Design Pattern?

The idea of Design Patterns in software engineering grew out of work by Emeritus Professor Christopher Alexander of UC Berkeley's own architecture department.

Professor Alexander's ideas were most notably applied to Software Engineering by four authors of the book Design Patterns: Elements of Reusable Object-Oriented Software, collectively nicknamed the "Gang of Four." This book identifies about two dozen fundamental patterns for solving recurring problems in software engineering.

Brad Appleton's article Patterns and Software: Essential Concepts and Terminology gives a good overview of Design Patterns applied to software programming. The Design Patterns... book, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, is a resource of inestimable value.

Why should programmers care about Design Patterns?

It has become widely accepted among software engineers and architects that designing applications in explicit conformity to these patterns facilitates the re-use of insight and experience gleaned by the best and brightest among us over the course of thousands of real-world software development efforts.

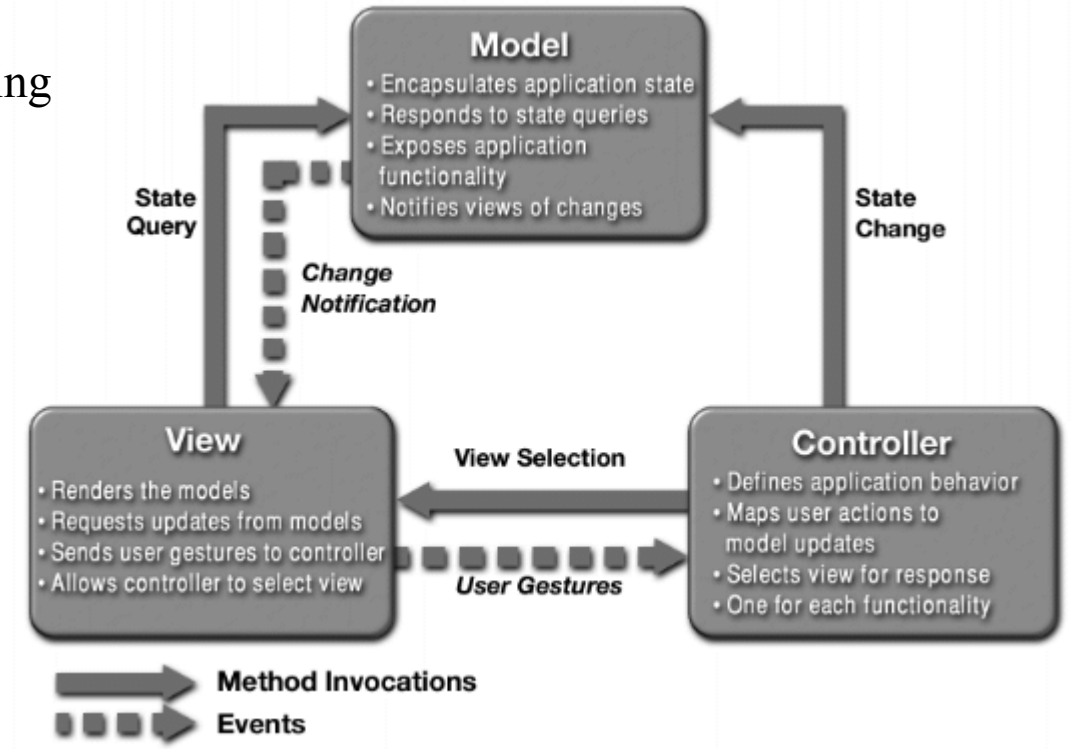
In short, thinking in terms of Design Patterns will make you a better programmer.

What is MVC?

MVC - Model-View-Controller - is a design pattern for the architecture of web applications. It is a widely adopted pattern, across many languages and implementation frameworks, whose purpose is to achieve a clean separation between three components of most any web application:

- Model: business logic & processing
- View: user interface (UI)
- Controller: navigation & input

In general, J2EE applications - including those built on top of the Streek project - follow the MVC design pattern.



Generic MVC Structure, courtesy of Sun (http://java.sun.com/blueprints/patterns/MVC-detailed.html)

What are the benefits of MVC?

MVC benefits fall into multiple categories:

- Separation of concerns in the codebase
- Developer specialization and focus
- Parallel development by separate teams

In looking at these in a little more detail, it will become clear that the second two bullet-points are corollaries of the first.

Separation of concerns (1)

Separation of Model, View, and Controller:

- allows re-use of business-logic across applications
- allows development of multiple UIs without touching business logic codebase
- discourages "cut-&-paste" repetition of code, streamlining upgrade & maintenance tasks

Looking at the other side of the same coin, intermingling M, V, and C code makes for:

- bigger, knarlier, harder-to-maintain classes
- repetition of code in multiple classes (upgrade & maintenance issues)

Separation of concerns (2)

Here's a snippet of what Martin Fowler, a respected and much-published author of design-oriented software literature, has to say on the subject of separation of concerns:

Following this principle leads to several good results. First, this presentation code [View] separates the code into different areas of complexity. Any successful presentation requires a fair bit of programming, and the complexity inherent in that presentation differs in style from the domain [Model] with which you work. Often it uses libraries that are only relevant to that presentation. A clear separation lets you concentrate on each aspect of the problem separately—and one complicated thing at a time is enough. It also lets different people work on the separate pieces, which is useful when people want to hone more specialized skills.

- Martin Fowler, *Separating User Interface Code*

http://www.martinfowler.com/ieeeSoftware/separation.pdf

Developer specialization and focus

- UI developers can focus exclusively on UI, without getting bogged down in business logic rules or code

- Business logic developers can focus on business logic implementation and changes, without having to slog through a sea of UI widgets

Parallel Development by Separate Teams

- Business logic developers can build "stub" classes that allow UI developers to forge ahead before business logic is fully implemented.

- UI can be reworked as much as the customer requires without slowing down the development of code that implements business rules.

- Business rule changes are less likely to require revision of the UI (because rule changes don't always affect structure of data the user sees) .

APIs: Contracts that Bind

- APIs: The Glue between Model, View, and Controller
- Simple MVC Example
- Controller can call methods on model in different ways
- Model can read/write from different or multiple databases

APIs: The Glue between Model, View, and Controller

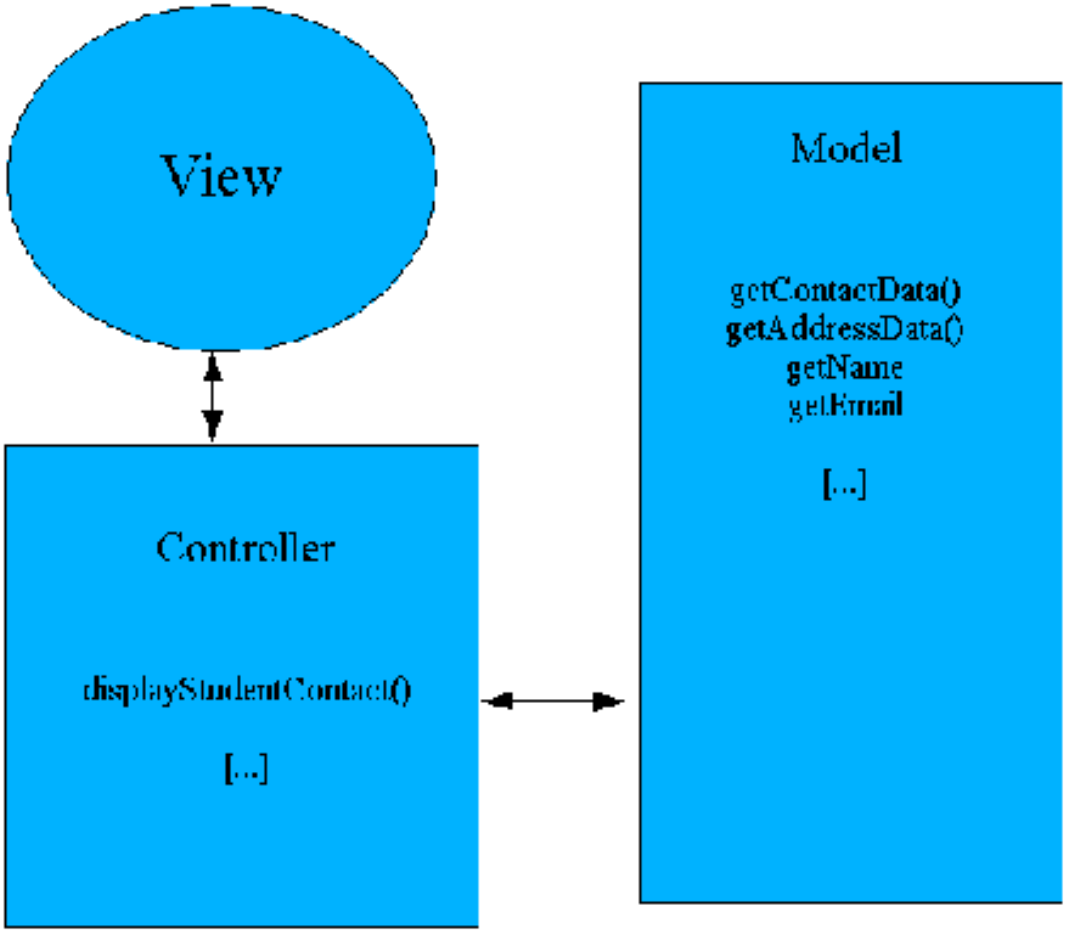
- "Separation of concerns" means that one layer doesn't care how another layer is implemented

- Each layer relies soley on the behavior specified in the API of other layers.

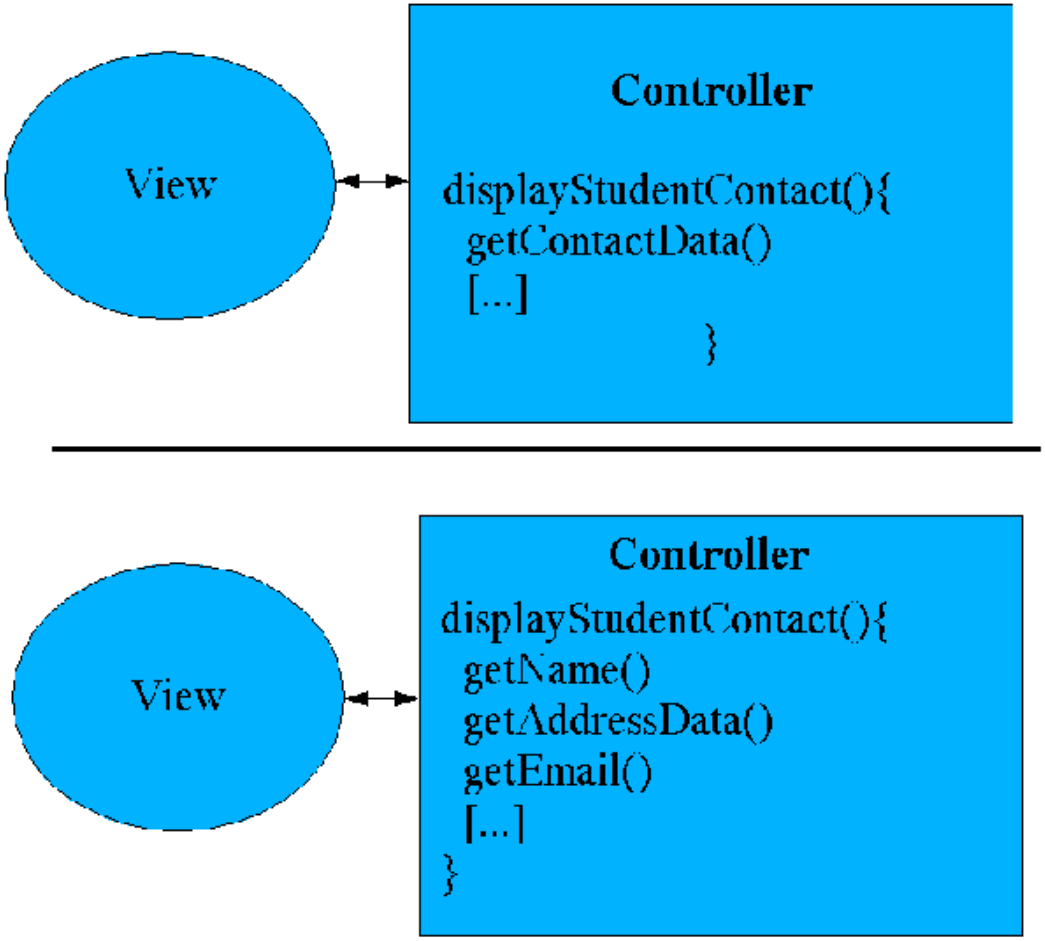
- The API specifies the behavior of the interface between a layer and the layers that interact with it.

- The APIs may be considered contracts to which each development team agrees before going off to implement the behavior promised by the interface.

Simple MVC Example

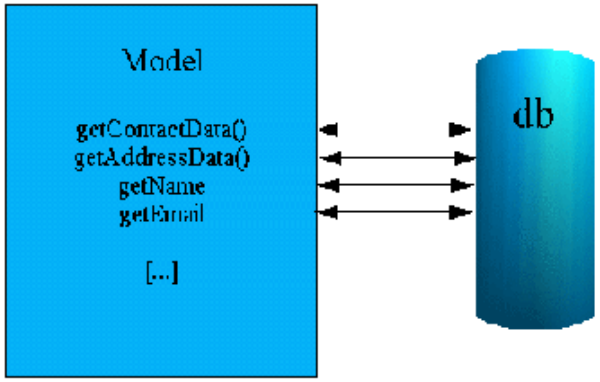


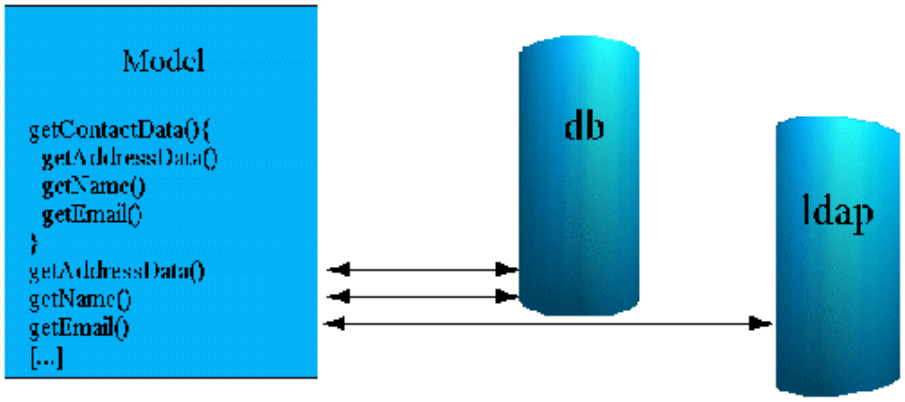
Controller can call methods on model in different ways



The controller can call whatever model methods it wishes to accomplish the desired behavior

Model can read/write from different or multiple databases





Two ways to persist contact information

Model (Detail)

The application's model includes:

- Application State
- Application Rules
- Persistent data

Application State

The data model represents the "noun" objects needed by the application and/or its users, such as:

- people (faculty, staff, students, customers, visitors, application administrators)
- widgets needed by application users (shopping carts, transcripts, contact information, merchandise)

The data model defines these objects abstractly.

The instances of the data objects currently in-play, and the value of those objects, constitutes the "application state."

Application Rules

Application rules are the business logic of the application.

Examples of business logic are:

- A student may not alter the data in her transcript
- Only the student herself and application administrators may modify a student's contact information
- A faculty member may view a student transcript only if she is the student's designated advisor

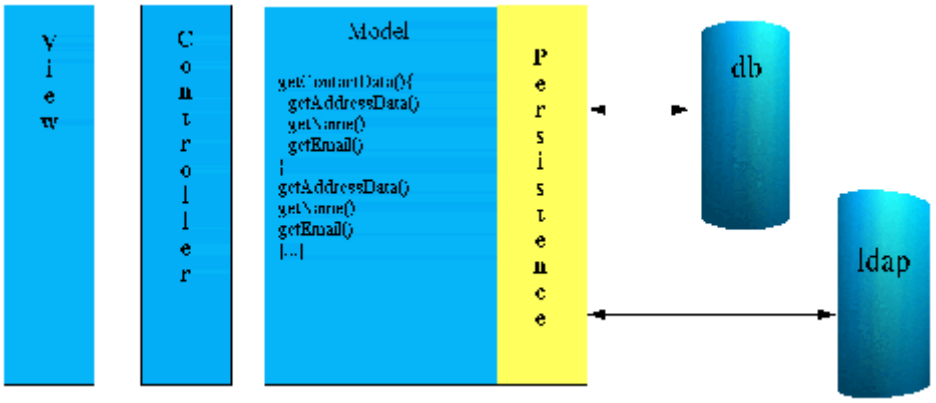
Persistent data (1)

Persistence refers to long-term storage of data - beyond the scope and time-frame of the user's session in the application.

Persistence is taken care of "behind the facade" of the data model. That is, the application's Model layer exposes a facade representing the data model; and behind-the-scenes the Model takes care of reading & writing data that populates instances of the model objects.

If the data store used by the application changes (from flat files to a relational database, for example), other layers of the application (such as the View) don't need to be altered: they are shielded from persistence details by the Model facade.

Persistent Data (2)



The persistence layer is part of the Model

Model Implementation Technologies

Within the J2EE framework, there are a range of technologies that may be applied to implementation of the model layer. These include:

- Object-Relational Mapping Frameworks (Java Objects <-> Relational Databases):
  - Hibernate, an open-source persistence framework
  - Oracle's TopLink
  - Sun's EJB (Enterprise Java Beans)
  - Sun's JDO (Java Data Objects)
- Hand-coded data accessors via the JDBC API

The attributes and merits of each of these solutions are outside the scope of this presentation. In general, however, Object-Relational mapping frameworks involve a layer of abstraction and automation between the application programmer and data persistence, which tends to make application development faster and more flexible. Differences between specific O-R Mapping technologies turn on the array of services offered by each technology, vendor specificity, and how well these differences map to a particular application's development and deployment requirements.

View (Detail)

- Elements of the View
- View Techologies
- Templating
- Styling
- The Decorator Pattern

Elements of the View

The application's View is what the user sees. In a web application, the View layer is made up of web pages; in a web application that can be accessed by non-traditional browsers, such as PDAs or cell-phones, the View layer may include user interfaces for each supported device or browser type.

In any case, the View includes:

- Core data - the subject of a page's business
- Business logic widgets (e.g., buttons to perform Edit or Save)
- Navigation widgets (e.g., navigation bar, logout button)
- Skin (standard look of the site: logos, colors, footer, copyright, etc.)

View Technologies (1)

In the context of a web application, view technologies are used to render the user interface (UI) in a way that properly and dynamically links it to the application's business-logic and data layer (i.e., to the model).

Java Server Pages (JSPs) are a widely-utilized technology for constructing dynamic web pages. JSPs contain a mix of static markup and JSP-specific coding that references or executes Java. In brief:

- JSP Tags call compiled Java classes in the course of generating dynamic pages.
- JSP Directives are instructions processed when the JSP is compiled. Directives set page-level instructions, insert data from external files, and specify custom tag libraries.
- Java code - in sections called "scriptlets" - can be included in JSP pages directly, but this practice is strongly discouraged in favor of using JSP Tags

View Technologies (2)

XML Pipelining is another technique for rendering the User Interface. Apache's Cocoon and Orbeon's OXF are technologies that use this technique. (For the record, XML pipelining simply means executing a series of transformations of XML documents in a proscribed order. For example, a data structure pulled from a persistent store may be rendered as an XML document, then - in a pipeline - may be augmented with data from a separate data structure; sorted; and rendered in HTML format using a series of XSLT transformations. A simple example of an XML pipeline using Apache's Ant tool can be viewed in this article.)

Templating

In order to maintain a consistent look-and-feel on a site, pages are often templated.

Templated web-page layout, courtesy of Sun

The web page's <body> is where core business content of a page can be found. Using a template to render a consistent header, footer, and navigation UI around the core business content standardizes a site's graphic presence, which tends very strongly toward making users' experience smoother and less prone to error.

Styling Web Pages with CSS

Cascading Style Sheets are a critical component of rendering a consistent look and feel in a web site (cf. this CSS Tutorial for more on Cascading Style Sheets).

CSS is used to specify colors, backgrounds, fonts and font-size, block-element alignment, borders, margins, list-item markers, etc. By specifying the designer's choices in a single file (or group of files), and simply referring to the style sheet(s) in each web page, changes in style can be accomplished without altering the pages themselves.

The breadth of design choices that can be achieved simply by applying different CSS to a web page is vividly illustrated at the CSS Zen Garden website.

The Decorator Pattern (1)

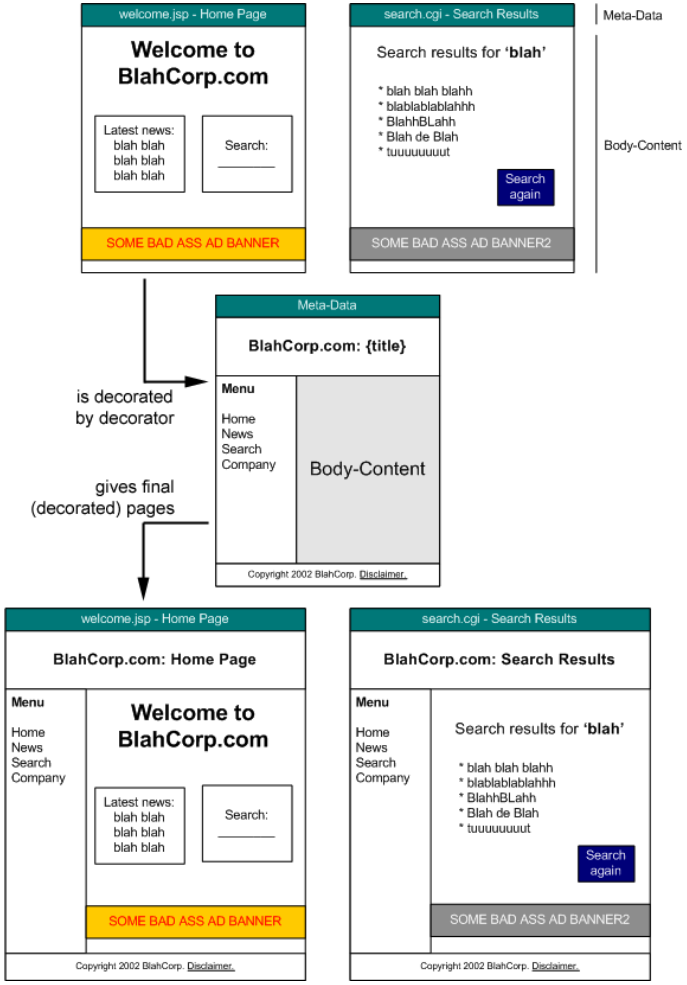
Templating and styling are aspects of an important "Gang of Four" design pattern, called the Decorator Pattern. Using the Decorator Pattern simply means wrapping some core object in something that gives it additional functionality. In the context of building a user interface for the web - the View layer of an MVC-architected application - this might mean wrapping:

- core business information in a
- template (e.g., header + footer + navigation), which includes (in the common HTML header) reference to a
- CSS stylesheet (where background, colors, fonts, etc. are specified)

In order to implement this design pattern in the view layer, page decorating frameworks, such as OpenSymphony's SiteMesh can simplify the process. SiteMesh intercepts HTML pages produced statically and/or dynamically, and applies a uniform "skin" to them, as specified in configuration files.

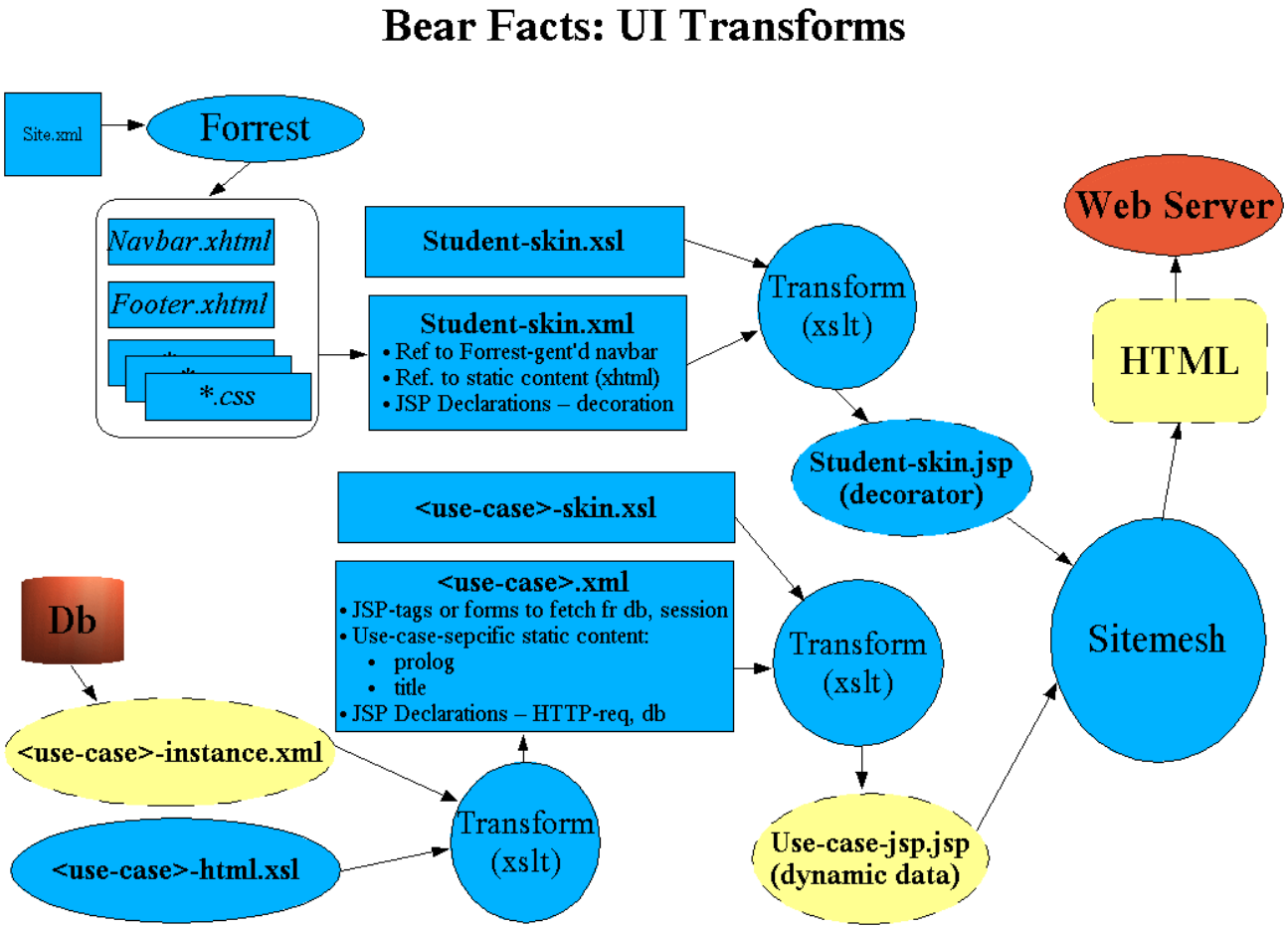
Here's a diagram showing how SiteMesh works.

The Decorator Pattern (2)



http://www.opensymphony.com/sitemesh/

The Decorator Pattern (3)



Web page generation in a local J2EE application

Controller Detail

- Responsibilities of the Controller
- Controller Technologies

Responsibilities of the Controller

The Controller will do the following:

- Parse a user request (i.e., "read" it)
- Validate the user request (i.e., assure it conforms to application's requirements)
- Determine what the user is trying to do (based on URL, request parameters, and/or form elements)
- Obtain data from the Model (if necessary) to include in response to user
- Select the next View the client should see

The sequencing of calls to the Model (business-logic layer), and/or the sequencing of views and required input from the user defines the application's workflow. Workflow is thus defined in the Controller layer of the application.

The Controller layer is, essentially, a traffic cop...

Controller Technologies

There are numerous application frameworks that can be used to provied the control layer in an MVC-architected application. These include:

- Struts
- Java Server Faces
- WebWork
- Spring

Which of these (or others) is most appropriate to an application is a question to discuss with an experienced application architect.

Deployment Descriptors: Function

- What Deployment Descriptors Do
- Deployment Descriptors are Declarative
- Additional Externalization

What Deployment Descriptors Do

A "Deployment Descriptor" is a configuration file. It is not code, and it is never compiled.

Deployment Descriptors are used to configure an application as it starts up in its container (i.e., in its application server). Configuring an application includes wiring its components together: declaring what responses correspond to which requests.

In the J2EE environment, the mother of all deployment descriptors is the file web.xml. The contents of this file are specified in the Java Servlet specification; depending on the version of the specification the file must conform to a particular DTD or XSD:

- DTD for Servlet Spec 2.3 web.xml
- XSD for Servlet Spec 2.4 web.xml

Deployment Descriptors are Declarative

Deployment Descriptors are "declarative" and are utilized in the context of J2EE's declarative programming model. What that boils down to is that:

- Many of the important attributes of J2EE classes are not hard-coded
- Deployment Descriptors are read when the components are started (deployed), and used to configure deployed modules and classes
- This means one can declare (via the Deployment Descriptor) what such attributes ought to be without changing or recompiling any code.
- The attributes in question are ones that specify application components, workflow, security, etc.

There's nothing magical about this in the J2EE framework. What's important is that many of the attributes that might have been hard-coded in older frameworks are, in J2EE, "externalized." Externalization allows applications to be more modular.

Additional Externalization

Additional externalization is built on the pattern established with web.xml. Additional functionality declared outside outside the application codebase might include:

* page flow - defined by web application frameworks such as Struts
* page decoration (skins) - are defined by web page layout and decoration frameworks such as SiteMesh

Benefits of Deployment Descriptors (1)

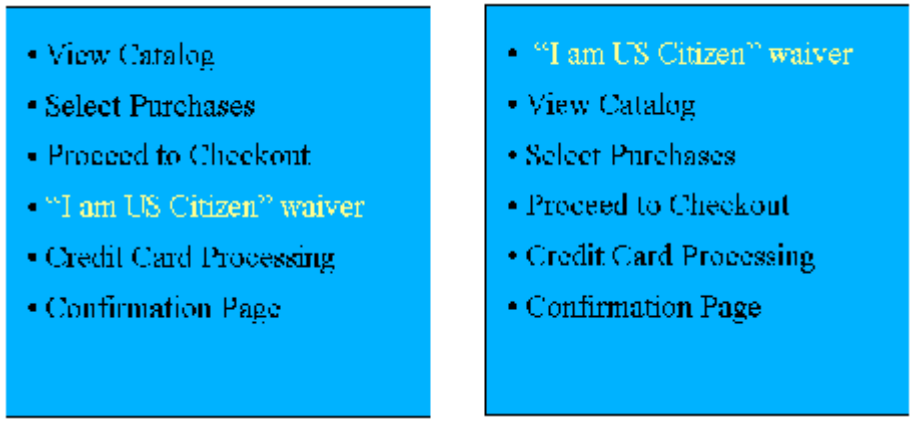
The chief advantage of externalizing configuration via Deployment Descriptors is that the binding between different modules or layers is "looser"; that is, changes can be implemented without rewriting or recompiling code.

Deployment Descriptors can be used in J2EE applications to specify:

- Which persistent data stores are to be used (e.g., which database on which host)
- Which security framework should be used
- What security should apply to each part of the application (i.e., which user-roles are authorized to do what actions or see what pages)
- Which class should handle each user-requested URL
- What the application workflow should be
- Which View page should be used to respond to each user request, and how application state (values of data in the Model) should be taken into account in this determination
- Which "skin" should be used to decorate each View page presented to the user

Benefits of Deployment Descriptors (2)

For example, deployment descriptor elements defining the Controller layer of an application map user-requested URLs to the classes that will fulfill the requests. The mapping defines the application's workflow. Because the mapping is declared in a Deployment Descriptor, the workflow can be changed without altering code, as in this example:



Workflow Changes for Secure Encryption Software On-line Purchase: the order in which a user is presented pages can be externalized in the Deployment Descriptor

Example Deployment Descriptor:

The next slides show an example of a struts-config.xml file, from the Streek client application BearFacts. Without getting into too many details, these are the steps that connect a user's request with the sequence of actions described by this deployment descriptor:

- At application startup, Struts ActionServlet is initialized with data in struts-config.xml
- web.xml deployment descriptor maps all URL requests of this form to a Struts ActionServlet (whose class is also specified in web.xml)
- User requests URL, of the form http://{context-root}/student/do/*
- The action path is the part of the URL that follows "student/do"
- Depending on the action-path part of the requested URL:
    - An "Action Class" specific to the type of operation requested is identified (the "type" attribute of <action>)
    - The specified "Action Class" calls appropriate Model classes to performs business logic, processing data (e.g., form data) in the user's request
    - The View that the user will see next is selected based on which forward name ("name" attribute of the <forward> element) is set as a result of processing performed by the Model class(es)

struts-config.xml (1 of 2)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <global-forwards>
    <forward name="systemMessage" path="/systemMessage.jsp"/>
  </global-forwards>
  <action-mappings>
    <action path="/studentMain"
      type="edu.berkeley.sis.bearfacts.student.action.StudentMainAction"
      scope="request" validate="false" parameter="bfaction">
      <forward name="welcome" path="/student/welcome.jsp"/>
      <forward name="logout" path="/login/logout.jsp" redirect="true"/>
    </action>
    <action path="/studentContact"
      type="edu.berkeley.sis.bearfacts.student.action.StudentContactAction"
      scope="request" validate="false" parameter="bfaction">
      <forward name="displayProfile" path="/student/personal/displayProfile.jsp"/>
      <forward name="displayContacts" path="/student/personal/displayContact.jsp"/>
      <forward name="updateContact" path="/student/personal/updateContact.jsp"/>
      <forward name="confirmContactUpdate" path="/student/personal/confirmContactUpdate.jsp"/>
    </action>
    <action path="/registration"
      type="edu.berkeley.sis.bearfacts.student.action.StudentRegistrationAction"
      scope="request" validate="false" parameter="bfaction">
      <forward name="classSchedules" path="/student/registration/classSchedules.jsp"/>
      <forward name="regFeeSummary" path="/student/registration/regFeeSummary.jsp"/>
      <forward name="summerRegFee" path="/student/registration/sumRegFee.jsp"/>
      <forward name="registrationBlocks" path="/student/registration/registrationBlocks.jsp"/>
      <forward name="finalExamSchedule" path="/student/registration/finalExamSchedule.jsp"/>
      <forward name="telebearsAppointment" path="/student/registration/telebearsAppointment.jsp"/>
    </action>
[...]
```

struts-config.xml (2 of 2)

```
 [...]
  <action path="/academicRecord"
     type="edu.berkeley.sis.bearfacts.student.action.StudentAcademicRecordAction"
     scope="request" validate="false" parameter="bfaction">
     <forward name="currentTermGrades" path="/student/academic/studentCurrentGrades.jsp"/>
     <forward name="priorTermGrades" path="/student/academic/studentPriorGrades.jsp"/>
   </action>
   <action path="/financialAid"
     type="edu.berkeley.sis.bearfacts.student.action.StudentFinancialAidAction"
     scope="request" validate="false" parameter="bfaction">
     <forward name="offerLetter" path="/student/fin-aid/fin-aid-offer.html"/>
     <forward name="financialAidStatus" path="/student/fin-aid/finaidStatus.jsp"/>
     <forward name="financialAidAwardSummary" path="/student/fin-aid/finaidAwardSummary.jsp"/>
     <forward name="missingDocuments" path="/student/fin-aid/finaidMissingDocs.jsp"/>
   </action>
   <action path="/CARS"
     type="edu.berkeley.sis.bearfacts.student.action.StudentCARSAction"
     scope="request" validate="false" parameter="bfaction">
     <forward name="classSchedules" path="/student/personal/classSchedules.jsp"/>
     <forward name="refundSummary" path="/student/cars/refundSummary.jsp"/>
     <forward name="awardsSummary" path="/student/cars/awardsSummary.jsp"/>
     <forward name="paymentSummary" path="/student/cars/paymentSummary.jsp"/>
     <forward name="quickStatement" path="/student/cars/quickStatement.jsp"/>
   </action>
   <action path="/loans"
     type="edu.berkeley.sis.bearfacts.student.action.StudentLoansAction"
     scope="request" validate="false" parameter="bfaction">
     <forward name="directLoan" path="/student/loans/directLoan.jsp"/>
   </action>
  </action-mappings>
</struts-config>
```

Sample Application

- Requirements
- Elements in Model, View, and Controller
- Use-case: User Login

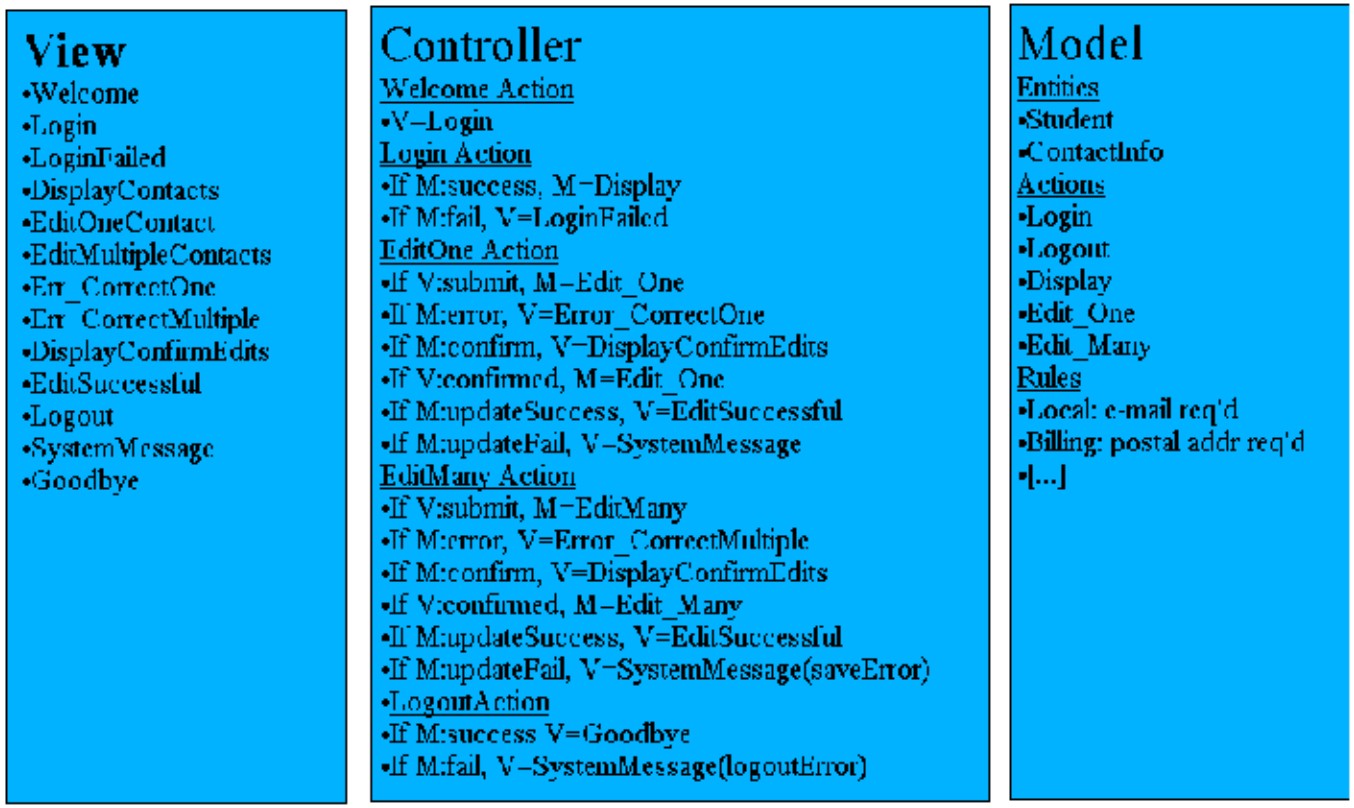
Sample App: Requirements

A student logs in. If authenticated, she is presented with multiple contact-information sets (e.g., Local, Billing, Permanent). She may choose to edit any set of information individually, or may edit all types at once. Certain rules must be enforced:

- Local contact info must include e-mail address
- Local contact info must be in California
- Billing contact info must include postal address
- Permanent contact info must include postal address
- (etc.)

Sample App: Elements in MVC

Views, Controller mappings, and Model elements derived from requirements:



Elements in Model, in View, and in Controller layers

- The **View** layer includes one view for each use-case with which the user will be presented.
- The **Model** contains objects representing entities; objects that perform actions; and representation of rules to which edited data are required to conform.
- The **Controller** maps user requests to Model actions, and the response of the Model to appropriate Views.

Sample App: Login Use-case

Key: user actions; application actions

- User points browser at welcome screen
- Controller receives root URL request, returns Welcome view
- User clicks a button to log in
- Controller receives Welcome request, presents Login view
- User supplies UID and Password, clicks button to authenticate
- Controller receives Login request, passes control to Login action in Model
- Controller receives response from Model (indicates login success or failure)
- Controller presents DisplayContacts or LoginFailure view, based on Model response
- [...]

Fini...