Web-Application Development Using the Model/View/Controller Design Pattern

Avraham Leff, James T. Rayfield
IBM T. J. Watson Research Center
{avraham,jtray}@us.ibm.com

Abstract

The Model/View/Controller design pattern is very useful for architecting interactive software systems. This design pattern is partition-independent, because it is expressed in terms of an interactive application running in a single address space. Applying the Model/View/Controller design pattern to web-applications is therefore complicated by the fact that current technologies encourage developers to partition the application as early as in the design phase. Subsequent changes to that partitioning require considerable changes to the application's implementation -- despite the fact that the application logic has not changed. This paper introduces the concept of Flexible Web-Application Partitioning, a programming model and implementation infrastructure, that allows developers to apply the Model/View/Controller design pattern in a partition-independent manner. Applications are developed and tested in a single address-space; they can then be deployed to various client/server architectures without changing the application's source code. In addition, partitioning decisions can be changed without modifying the application.

1. Introduction

1.1. The Model/View/Controller Design Pattern

The well-known *Model/View/Controller*[15][7] (or *MVC*) design pattern is a useful way to architect interactive software systems. Also known as the *Presentation/Abstraction/Control*[8] (or *PAC*) design pattern, the key idea is to separate user interfaces from the underlying data represented by the user interface. The "classic" MVC design pattern actually applies to low-level user interaction such as individual keystrokes or activation of mouse buttons. In MVC, the View displays information to the user and, together with the Controller which processes the user's interaction, comprises the application's user interface. The Model is the portion of the application that contains both the information represented by the View and the logic that changes this information in response to user interaction. The PAC design pattern similarly decouples the application's information from the user interface. Here, the MVC's View and Controller are combined into a *Presentation*; the application's data is termed the *Abstraction* component; and the *Control* component is responsible for communication between the decoupled Presentation and Abstraction components. Use of the MVC and PAC design pattern makes it easier to develop and maintain an application since:

- the application's "look" can be drastically changed without changing data structures and business logic.
- the application can easily maintain different interfaces, such as multiple languages, or different sets of user permissions.

Colloquially (e.g., [5]), the term "MVC" has been extended to describe the way that large-scale changes to an application's Model are driven by a Controller that is responsible, not only for accepting and processing user interactions, but for logic that changes an application's overall state in response to the event created by the user's interaction. In response to changes in the Model, the Controller initiates creation of the application's new View. This paper uses the term MVC in this, more general and "PAC-like", sense of architecting an application so that business logic, presentation logic, and request processing are deliberately separated.

1.2. Web-Applications and the MVC Design Pattern

Web-applications, like other interactive software systems, can benefit by being architected with the MVC design pattern. For example, one "all Java" approach uses Entity Enterprise JavaBeans (EJBs [9]) as the Model, constructs the View through HTML and JavaServer Pages, and implements the Controllers through Servlets and Session EJBs.

0-7695-1345-X/01/$10.00 © 2001 IEEE

The problem with using the MVC design pattern to develop web-applications arises from the fact that web-applications are intrinsically partitioned between the client and the server. The View, of course, is displayed on the client; but the Model and Controller can (theoretically) be partitioned in any number of ways between the client and server. The developer is forced to partition the web-application "up-front" -- certainly at the implementation stage, and often as early as the design phase. In contrast, MVC is *partition-independent*: i.e., the Model, View, and Controller reside and execute in a single-address space in which partitioning issues do not arise. Partition-independence is one of MVC's *features* since location-dependent issues should not drive architecture or design decisions. Unfortunately, partitioning implies that web-applications are *location-dependent*, and this characteristic means that it is much harder to apply the MVC design pattern in the web-application context.

Of course, developers can simply partition the web-application, deciding that method *A* will run on the server and method *B* on the client. Once partitioning is done, the MVC design pattern can be applied, in parallel, to implement the client and server portions of the application. The problem with this approach is that it is often impossible to make correct partitioning decisions early in the design phase since these decisions depend on application requirements that change considerably over the course of the project[3]. The problem is made even more difficult by the fact that the correct partitioning decision depends on static (e.g., relative power of client to server machines) and dynamic (e.g., network congestion) environmental factors. Applying the MVC design pattern in an environment where the partitioning decision is not fixed is a difficult task.

To make matters worse, "controller partitioning" is often not an independent, "tunable" feature of the development process. Rather than being a function only of the web-application's intrinsic characteristics, partitioning depends on technology decisions that have nothing to do with the application. Take, for example, the "thin-client" *versus* "fat-client" dichotomy. Often, application developers are constrained in their choice of implementation technologies. Perhaps they are told "applets are bad because they are heavy-weight and because of security concerns". Perhaps they are told "you can't build sophisticated views in HTML". Once technology selections are made, they greatly determine how the application will be partitioned. When deployed as a thin-client, only the web-application's View is resident on the client; the Controllers (excluding the actual window screen controls) and the Model are resident on the server. When deployed as a fat-client, the web-application's Model and Controllers are also resident on the client. Technology choices, in other words, mean that developers may not be able to make the right partitioning decisions -- even if they know what the right decision is.

To summarize, web-applications can certainly use the MVC design pattern when:

- the correct partitioning is known *and*
- the available technology infrastructure is compatible with that partitioning.

In practice, developers simplify the problem by making *a priori* assumptions about the technology and partitioning solution. This approach is discussed in more detail later (Section 3.2.3).

We introduce the concept of *Flexible Web-Application Partitioning* (or *fwap*) to enable web-applications to use the MVC design pattern more naturally. Before discussing *fwap* and other, related work, in Section 3, we first provide a simple motivating example of web-application partitioning in Section 2.

2. Partitioning a Web-Application: Example

Picture a web-application that enables clients to get information about a company's employees. We simplify the example by allowing the client to perform only two operations.

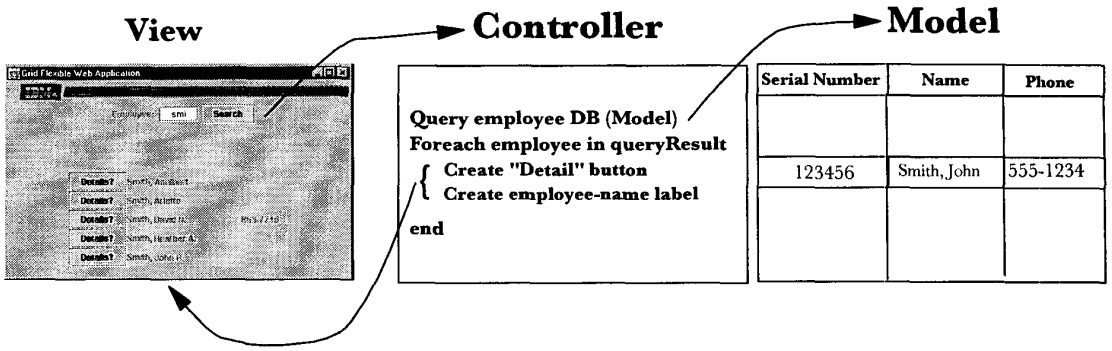
1. By supplying a name, and clicking on a "search" button, search the employee directory "by name". The search returns the set of employees that match the search criteria in a format that displays an abbreviated employee record for each member of the returned set.

2. By clicking on a "details?" button, get detailed information about a specific employee.

Implemention in a stand-alone, single address-space, environment, is straightforward. From the perspective of the MVC design pattern (see Figure 1):

- The Model consists of the records in the employee directory.
- There are four Views: a "search" panel; a display of abbreviated information about a set of employee records; a display of detailed information about a specific employee; and a report that no employees match the search criteria.
- There are two Controllers: one that, given a "search" directive, drives the process of querying the Model and returns a result set; and one that, given a "details" directive, queries the Model to get the full set of information about the specified employee.

However, implementation as a web-application in a client/server environment raises the issue of partitioning which is conceptually orthogonal to, but in practice complicates, the MVC design pattern. Naively, as there are two Controllers, the application can be implemented in one of four ways. Either both Controllers execute exclusively on the client or server, or one Controller executes on the cli-



ent and the other executes on the server. Each partitioning decision greatly affects the way that the application is implemented. For example, if both Controllers run on the client (the "fat-client" approach), the entire Model must be downloaded to the client -- which is often impractical. If both Controllers run on the server (the "thin-client" approach), two round trips between client and server must be performed each time that the client searches for an employee and then asks for more detail about that employee.

In fact, for many environments, neither the thin-client or the fat-client is ideal. Instead, using a *dual-mvc* approach **[6]**, we partition the Controllers between the client and server. Specifically, the "search" Controller executes on the server in association with a Model consisting of the complete employee directory. However, when returning relatively small sets of employee records, the Controller *also* returns the full record for each of the employees, so that they can be maintained in the client-side Model. The dual-mvc approach allows requests for detailed employee information to be served by the client, thus eliminating a client/server interaction. (This implementation is beneficial only when application scenarios typically consist of a preliminary search for an employee using a "partial name", followed by request for more information after the specific employee is determined by inspection. Remember: this is only a motivating example!)

Of course, what we really want is to do *avoid partitioning* while implementing the application, since the correct partitioning decision depends on factors that are not necessarily determined until actual deployment. For example, if the employee directory is relatively small, the "fat-client" approach with both Controllers executing on the client makes sense and would provide better performance. Conversely, if the application is deployed in a "internet" environment in which users want minimal customization of their environment, the "thin-client" approach may be the only solution possible. Delaying application partitioning for as long as possible is even more attractive because partitioning gets in the way of designing the Views and developing the business logic needed by the Controllers. Flexible web-application partitioning addresses these needs. In fact, flexible web-application partitioning goes further, allowing partitioning decisions to vary *dynamically*, during application execution.

The *fwap* programming model explictly supports the MVC design pattern, and enables programs executing in smvc mode (Section 3.1) to execute in a single address-space. When deployed, these programs can be flexibly partitioned without changing the source code used during smvc development. We refer to such *fwap* applications as *fwaplications.*

3. Flexible Web-Application Partitioning

Although web-applications must intrinsically deal with partitioning issues, Section 2 illustrates why it is preferable to implement web-applications in a manner that is partition-independent. Partition-independence enables developers to be flexible about partitioning, because it allows them to deal with changing technology infrastructures, changing application characteristics, and even changing environment conditions such as network congestion. The problem, of course, is how to maintain partition-independence given that web-applications are location-dependent. The *fwap* programming model, implementation, and infrastructure, enable developers to build web-applications in precisely such a partition-independent manner. Applications are developed and tested in a single-address space; they can then be deployed to various client/server architectures without changing the application's source code. In addition, partitioning decisions can be changed without modifying the application.

In order to provide partition-independence, *fwap* must address a set of difficult problems. These problems

involve partition-independent technologies for the individual Model, View, and Controller components of an application; integration of these components in a consistent application programming model; and the construction of transforms that map the partition-independent representation of an application to a partitioned application running on actual client/server platforms. We are currently focusing our efforts on enabling *fwap* to support a variety of *deployment architectures*. In order to make progress on this front, we have made certain simplifying assumptions that allow us to avoid some of the above problems for the moment. We discuss these assumptions below (Section 3.2).

3.1. *fwap* Architectures

fwap supports the following architectures:

- *single-mvc* (or *smvc*), which serves as the *fwap development* architecture. The smvc architecture corresponds to the classic MVC design pattern (e.g., Figure 1): the Model, View, and Controllers all reside and execute in a single address space in which client-server and partitioning issues do not arise.
- thin-client, a *deployment* architecture in which the Model and Controllers reside and execute in a single address space on the server, and generate the View that is rendered on the client.
- dual-mvc (or *dmvc*), a *deployment* architecture **[6]** in which Controllers and Model reside on both the client and server. Either the client or server can generate the View, as needed, to be displayed on the client.

3.2. Related Work

3.2.1. View Specification and Generation. In order for Views to be rendered on different client platforms, they must be specified in a manner that is independent of platform-specific characteristics. One approach to this problem **[1]** is to use XML as the basis of a universal, appliance-independent, markup language for user interfaces. Our approach differs only in that we use a Java API to specify a library of GUI components and to construct Views out of these components. Then, at runtime, platform-specific implementation libraries render the Views on the client device (e.g., using HTML for web-browsers). We are not currently focusing on issues relating to how such "canonical" Views should be rendered on devices with considerably different form factors from desktop and web-browser environments (e.g., PDAs and web-phones).

3.2.2. Controller Specification and Execution. Because we want fwaplications to be deployable to the dual-mvc architecture, Controllers must be able to execute "as is" on the client as well as on the server. Since we code fwaplication Controllers to the Java Virtual Machine, this excludes a number of client platforms as fwaplication deployment platforms. PDAs and web-phones, for example, may not support the JVM -- although we do expect J2ME **[14]** to increasingly enable such devices to support Java applications. Even in the case of web-browsers, the requirement of device independence (e.g., Microsoft Internet Explorer *versus* Netscape Navigator) implies that it might better to write Controller in JavaScript **[11]** than in Java. At present, we simplify the problem of "single source" Controller specification by using the Java Plug-in **[10]**: this allows standard web-browsers to run applications written to the JDK 1.2 libraries.

3.2.3. Thin-Client Presentation Frameworks. It is important to note that a number of other projects **[2][16][17]** are also engaged in the application of the Model/View/Controller design pattern to web-application development. The key contribution of *fwap* is its emphasis of partition-independence. In contrast, these other efforts are explictly *partition-dependent*, and assume that web-applications are deployed to the thin-client architecture.

These presentation frameworks are being developed in the following context. The introduction of Java Servlets **[13]** in the late 1990s encouraged a style of web-application design in which servlet code generated Views through a series of `println` statements that wrote HTML to the client's web browser. JavaServer Pages **[5]**, introduced in 1999, inverted this paradigm by allowing developers to embed scripting code (that accessed server-side Model and Controller code) into HTML pages. Both of these approaches violate the MVC design pattern because they tend to inextricably mix View generation code with Controller and Model code. Thin-client presentation frameworks are being developed with the intention of facilitating the use of MVC design pattern to build web-applications. The basic idea is to use servlets and server-side JavaBeans as an application's Controllers (application logic); use EJBs to represent the Model and the business logic; and use JavaServer Pages to extract Model data and generate Views. Individual frameworks emphasize different issues. Thus, Barracuda uses a "push MVC" approach in which the Model knows about, and is responsible for generating the View; WebWork and Struts use a "pull MVC" approach in which the View accesses the Model as needed. Struts is more tightly coupled to the Servlet API than WebWork.

Although fwaplications *can* also deploy to the thin-client architecture, *fwap* emphasizes the ability to deploy to the dual-mvc architecture so as to improve performance through client-side execution. As a result, *fwap* is concerned with issues, such as Model synchronization and transparent Controller delegation, that are not of interest to the thin-client presentation frameworks.

3.2.4. Model Synchronization. All applications which involve human interaction raise the issue of model synchronization with the database. This is because the timescale of a human "transaction" is several orders of magnitude greater than a reasonable database transaction time. A human interaction may take several minutes, but a database system usually cannot afford to hold transaction locks for more than about a second.

For example, consider a user updating an employee record. First, a database transaction is begun, and the current state of the record is loaded from the database and displayed by the application. Typically the database transaction is immediately committed. Then, the user might spend several minutes looking at the (stale) state of the data, updating some of the fields. Finally, the user presses the "Update" button and the modified data is stored back into the database under the scope of a second transaction. Note that the naive approach of keeping the first transaction active until the update button is pressed requires that database locks be held for the whole interaction, which is unacceptable because it eliminates concurrent access to the database[12].

During the time that the first user updates the employee record, another user may also update that record, introducing the possibility that the second update be inconsistent with the first update. Consider the case in which the first update transferred the employee to a work location in another state, and the second update updated income tax information for the employee in the previous state. Because the two updates occurred in two transactions, the database record now appears to show an employee simultaneously working in two states with different income tax regulations.

Techniques for dealing with this are well known in the literature. A common solution is for the application to keep a copy of the initial state of the database record, and to verify during the second transaction that the record has not changed since it was copied during the first transaction. In the example above, the application would detect during the attempted update of the income-tax information that another user had modified the record. The application would then reject the second update. This technique is often referred to as "optimistic concurrency control"[12], because it optimistically assumes that there will be no conflict, and thus does not lock the database. This assumption is validated only at commit time.

A more sophisticated approach to this problem is the "Predicate & Transform" approach[4], which maintains a log of the updates (the transforms) made by the application during the interaction, and applies the log inside a single transaction at commit time. The transaction is aborted only if concurrent updates by other users violate the application-specific predicates.

Because *fwap* is a human-interaction paradigm, these model-sychronization issues also apply. Even the single-mvc architecture must maintain temporary copies of database records outside the scope of a transaction in order to prevent long lock-hold times.

Since these issues apply to even the single-address-space deployment of an application, they are not consequences of the partition-independent (dual mvc) fwap architecture. Rather, they are fundamental consequences of interactive applications that can (and must) be addressed with techniques such as the ones mentioned above. The dual-mvc architecture introduces the additional complication of sychronization between the client and server temporary copies of database records maintained on behalf of the user session state. We discuss this further in Section 5.

3.3. *fwap* Programming Model

One way that the *fwap* infrastructure supports the programming model is by providing a base fwaplication interface which specifies that every fwaplication is associated with a Model and View, each accessed through their own API. Web-applications are developed by supplying an interface that extends the base fwaplication interface with application-specific methods.

3.3.1. View. *fwap* Views are composed from a suite of GUI components such as buttons and input text fields. The programmer can associate method invocations with user-interaction events (e.g., a button click) through the *registerEvent* method; the *fwap* infrastructure ensures that the specified method executes when the event occurs. The GUI components are specified through interfaces; this allows the implementation provided in the smvc development infrastructure to be replaced by platform-specific implementations when the application is deployed.

Components are created, removed, and accessed *only* through the *fwap* (key-based) API invoked against the fwaplication's *getView* method. The fwaplication may not directly refer to a component by storing a reference to the component. Because components are accessed only through the API, the *fwap* infrastructure can, for example, transparently synchronize a stale server-side View with the

current client-side View. Transparency is a basic requirement, since the *fwap* programming model precludes explicit code to transmit user input to the server. Without this ability, a Controller repartitioned so that it executes on the server rather than on the client, could not access the contents of an input field. The *fwap* infrastructure ensures that Controller code, partitioned to run on the server, and accessing the View through *getView*, will access up-to-date values.

3.3.2. Model. A fwaplication's Model follows the same principles as a fwaplication's View: components are specified by interface, and components are accessed only via a key-based API. Although many component models are suitable for fwaplications (e.g., COM or CORBA), the fwap Model components follow the Enterprise JavaBeans component specification **[9]**.

3.3.3. Controllers. In the *fwap* programming model, a Controller is an functional unit that may be invoked to run on either the web-application client or on the server. The actual location of a Controller's execution is determined by explicit directives in a *controller deployment descriptor*. This is in contrast to "utility" code that always executes in the same address space as the caller. It also contrasts to classic "client/server" code in which client code must explictly call out of its address space to invoke code on the server.

The first step in fwaplication development is to specify the set of methods (i.e., Controllers) that can be "flexibly partitioned". These methods comprise the interface extending the base fwaplication interface, supplied as part of the *fwap* infrastructure. A method can be "flexibly partitioned" only if it's part of this application-specific interface. Note that these methods are not constrained to take only user-interaction events as their input (as in "classic" Controllers). Rather, they may take arbitrary input and return arbitrary output. For example, one well-known performance optimization performs "syntactic" validation of user input on the client so that the client can immediately reprompt the user for valid input if it detects a problem. The client only sends the request to the server after the user's input has been validated. This optimization is usually implemented by JavaScript code running in a thin-client browser which then sends the request to a Servlet after it has validated the user's input. In *fwap* terms, the programmer simply specifies two methods *validate(...)* and *process(...)* in which *validate* calls *process* only if it determines that the input is syntactically valid. At deployment, the two methods can be partitioned so that *validate* runs on the client, and transparently calls *process* which runs on the server.

3.4. Development

fwaplication development consists of supplying an implementation for the subclassed fwaplication interface. Because this is completely application-specific, the *fwap* infrastructure provides no additional support for this phase. Following the Model/View/Controller design pattern, the programmer supplies the logic that generates the required Views, accesses the Model in order to generate Views, and updates the Model as necessary.

4. Current Implementation

This section describes the status of the current implementation of the *fwap* programming model and infrastructure.

4.1. smvc

The smvc View components are rendered using Java Swing components, and the fwaplication is developed and tested as a stand-alone Swing application.

4.2. thin-client

Figure 2 shows how the MVC structure of the sample application is deployed to the thin-client architecture.

Using a factory-pattern, the Swing-based View implementation of smvc mode is replaced by components that generate HTML that produce the equivalent View when rendered in a standard web-browser.

The fwaplication's `.class` files are installed in a standard web-server such as Tomcat, and made available to web-clients by creating a corresponding *servlet* entry in the `web.xml` file **[13]**.

A web-client's initial `GET` request to the URL designated in the *servlet-mapping* entry causes a *controller servlet* to:

- instantiate a new fwaplication instance that will be associated with this client.
- request that the fwaplication generate the initial View. In thin-client deployment, this consists of a stream of HTML that will be rendered in the client's web-browser.
- returns the HTML to the client.

Because all fwaplications inherit the same lifecycle methods, the controller servlet is generic so that a single instance can service all web requests to all installed fwaplications.

The web-client proceeds to interact with the fwaplication's GUI precisely as in smvc mode, with the identical Controller and Model executing (on the server) as in smvc

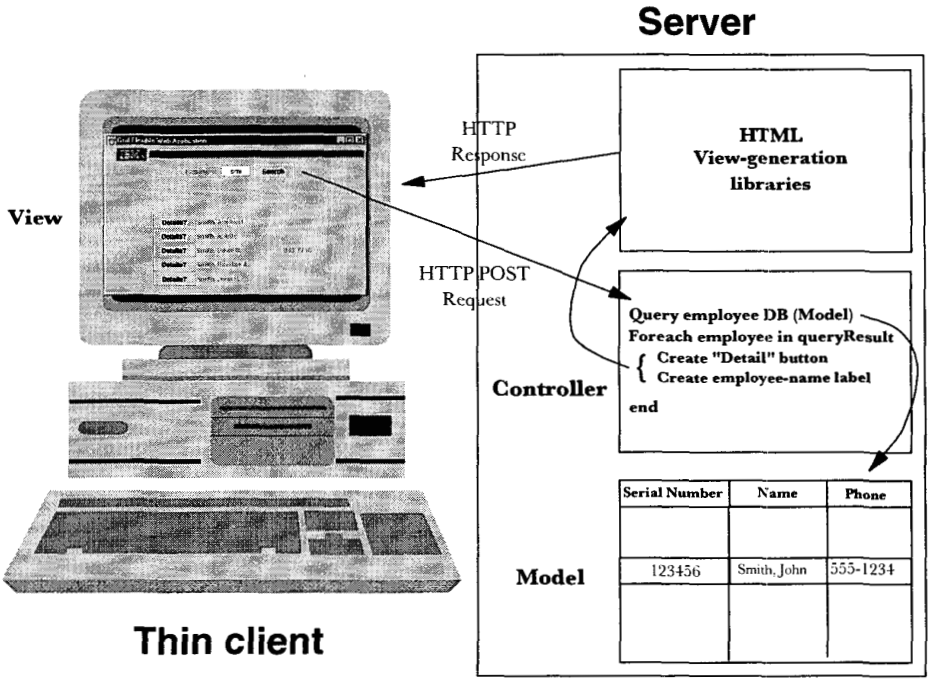


FIGURE 2. The Sample Application Deployed to the Thin-Client Architecture

development. Each time that the user triggers an interaction event (e.g., clicks a button), embedded (and generic) JavaScript generates a POST request to the controller servlet. The request specifies the Controller invocation associated, through the *registerEvent* method, with this interaction event. Using reflection, the controller servlet invokes the specified method, which proceeds to execute precisely as it does when the user clicks a button in smvc mode.

4.3. dual-mvc

One way to look at the current implementation of the dmvc deployment architecture is that two instances of the smvc fwaplication implementation execute: one in the client, and one in the server. Figure 3 shows how the sample application of Figure 1 is deployed to the dual-mvc architecture. In order for the smvc fwaplication implementation to execute "as is" on a web-client, we deploy the client-side fwaplication into a generic *loader applet* that runs in the web-browser. An HTML file is associated with each type fwaplication. It specifies the name of the fwaplication interface class; the location of the controller deployment descriptor file; and the URL through which the dmvc controller servlet can be contacted. By using the Java Plug-in **[10]**, even standard web-browsers can run applications written to the JDK 1.2 libraries. As a result, the View and Model code of a dmvc fwaplication are identical to the code used in smvc mode.

Coordination of the two activated fwaplication instances is simplified by the following observation. Although the purpose of the dmvc architecture is to enable Controller execution on either the client or server, dmvc execution always proceeds *serially*, never in parallel. When a user interaction with a GUI element triggers a Controller execution, the Controller will execute only at the location specified by the controller deployment descriptor. During the Controller's execution, fwaplication execution at the other location will not occur.

However, the client-side fwaplication must be enhanced with the ability to delegate Controller execution to the server when directed to do so by the controller deployment descriptor, or when the client requires Model components that are available only on the server (see *Ongoing Work*). We do this by having the client-side implementation class *extend* the smvc, server-side, implementation.

The client-side implementation class overrides the smvc implementation of each Controller with the following algorithm.

1. At runtime, access the controller deployment descriptor to determine whether the Controller is to be executed locally (on the client), or remotely (delegated to the server).

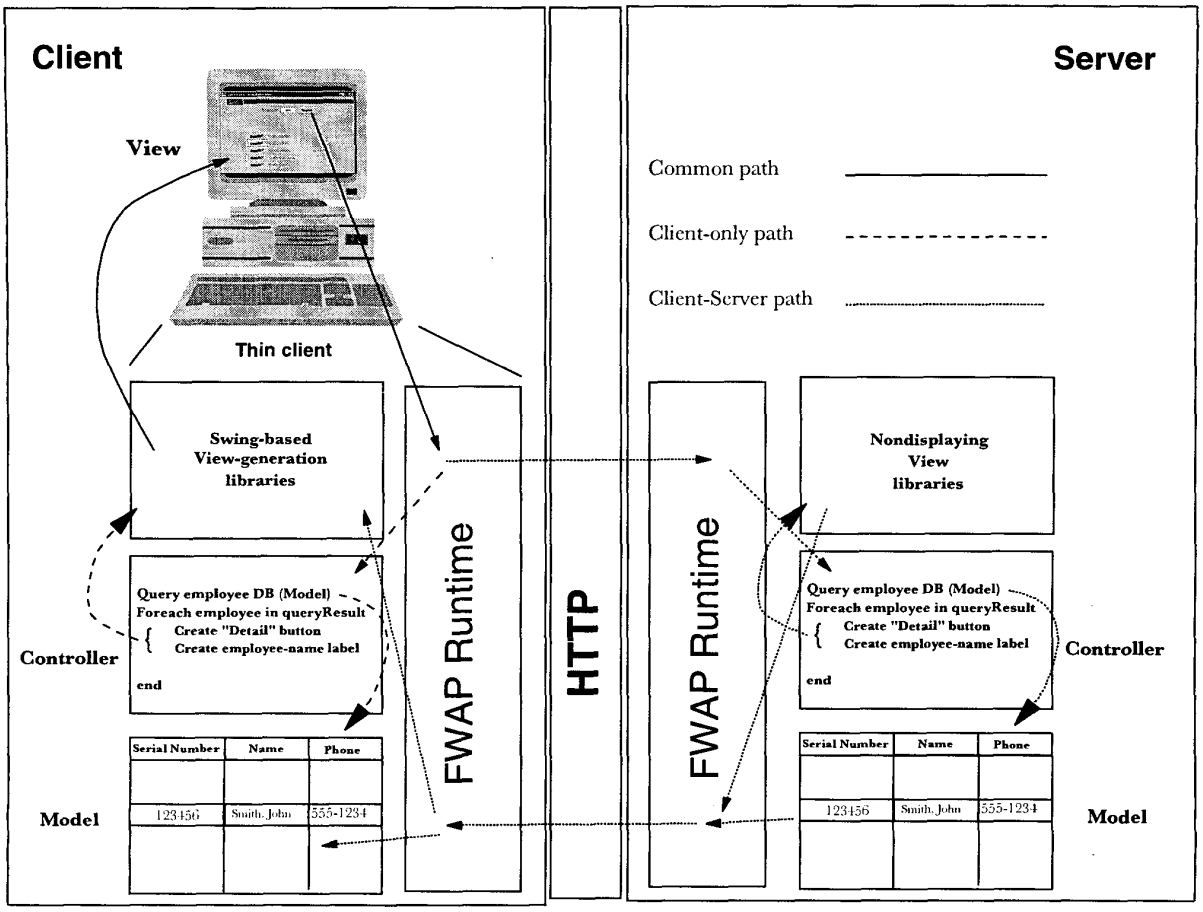


FIGURE 3. **The Sample Application Deployed to the Dual-MVC Architecture**

2. Perform local execution by calling *super.Method(...)*, i.e., by executing exactly the same code as in smvc mode. This is denoted by the "dashed arrow" execution path in the client side of Figure 3.

3. Perform server-side execution (denoted by the "dotted-arrow execution path of Figure 3) by:

a. Serializing the client-side Model.

b. "Serializing" the client-side View. We do not serialize the client-side GUI components which, after all, encapsulate Java Swing components that are meaningless to the server-side of the fwaplication. Instead, we serialize a "distilled" version of the client-side View, containing only the property values of each component.

   The server-side requires access to the current View for two reasons. First, as in the thin-client architecture, it needs to be aware of the state state changes caused by the user's interactions with the client-side View. Second, the server-side will likely construct the next View in terms of changes to the current View: e.g., by leaving one panel unchanged, and replacing a second panel.

   Serializing the View also involves serialization of all "event handlers" (Controller invocations) associated with View components through the *registerEvent* method. Although the user cannot interact with the server-side View, the server-side must include this information when it transfers control back to the client-side of the fwaplication.

c. Serializing the values of the parameters passed to the Controller.

d. Using Java's `ObjectInputStream` and `ObjectOutputStream` classes, the client-side Model, View, and Controller parameters are

passed to the dmvc controller servlet in an HTTP POST request.

e. The controller servlet synchronizes the state of the server-side fwaplication with the client-side Model and View.

f. Using Java dynamic invocation, the specified Controller is invoked on the server-side fwaplication instance. Recall that the server-side implementation of the Controller is identical to the code used in smvc mode.

g. The return value, (possibly) updated server-side Model, and new View (recall that Controller execution is responsible for constructing the fwaplication's next View), are serialized and transmitted back to the client-side fwaplication as the response to an HTTP POST request.

4. After its delegation to the server-side completes, the client-side Controller updates the client-side Model to reflect changes made to the server-side Model, "reconstitutes" the new View as client-side GUI components, and renders the View for the user.

5. Ongoing Work

fwap is a work in progress, and we discuss here some of the ongoing design issues and implementation that we are currently working on.

With respect to the smvc development architecture and the thin-client architecture, the ongoing work focuses on enriching the suite of View components. This relates both to the types of components supported and the number of properties provided by each type of component. The current framework and implementation, however, adequately supports the Model/View/Controller design pattern and deployment to a thin-client architecture. Our main research focus is on the dmvc deployment architecture.

5.1. Model Synchronization

Currently, when synchronizing the client and server portions of a deployed dmvc fwaplication, we simply copy the entire set of Model components from one location to the other. This is inefficient, since only components that have changed state need to be synchronized. We are therefore working on the use of "dirty bit" techniques to eliminate unnecessary Model synchronization operations.

5.2. Model Pre-Fetching

The basic motivation of the dmvc deployment architecture is to provide web-clients with the ability to directly service user requests rather than delegating requests to the server. Currently, we simply load the entire Model into both the client and server portions of a dmvc fwaplication; this trivially assures that the client will have the necessary Model components. Obviously, this approach does not scale beyond "proof of concept" applications. We are therefore adding infrastructure hooks to allow fwaplication programmers to "pre-fetch" Model components from the server side of a dmvc fwaplication to the client side. Once this is provided, fwaplication programmers can study typical flows of their application to determine which portions of the Model are best suited for client-side processing. For example, the example of *Section 2* assumes that directory queries often proceed in two stages: first, a broad query to retrieve a small set of possible employees matches; second, a "drill down" request to get more information about a specific employee from the first stage. Given such a scenario flow, it makes sense to pre-fetch the "details" information when processing the initial broad query; this allows client-side processing of the "drill down" request. However, if the result set of the broad query is too large, it will then be impractical (i.e., will degrade performance) to pre-fetch the detail information for the result set.

5.3. Dealing with Cache Misses

Because the client-side of a dmvc fwaplication contains only transient Model components, it in effect serves as a Model cache for the server. As a cache, the client-side Model must be able to deal with situation of a "cache miss" -- i.e., where Controller execution requires a Model component that is not resident on the client. (Note that Model "pre-fetching", discussed above, is a technique used to *reduce* the number of cache misses. Even with perfect knowledge, cache misses may occur since the client can cache only a portion of the server-side Model.)

The *fwap* infrastructure can transparently catch the exception thrown when a cache miss occurs. At that point, two possibilities exist:

- As the next cache "tier", ask the server-side Model for the necessary Model component. (If the component is not resident in the server's transient Model, it will perform the required requests to persistent storage to bring the component into the transient Model). Once the component has been fetched into the client-side Model, proceed with the client-side Controller execution.
- Cease client-side execution of the Controller, and delegate execution to the server-side of the dmvc fwaplication.

The first approach, "cache miss recovery on the client", makes sense when the Controller will not trigger many more cache misses. Otherwise, the overhead of con-

tacting the server on a per-component basis will cause considerable performance degradation. Conversely, the second approach, "cache miss recovery on the server", makes sense in situations where the Controller will trigger many more cache misses. As with Model pre-fetching, determining which approach is appropriate for a given Controller greatly depends on the typical scenarios for a specific application.

6. Conclusion

This paper describes how the *partition-independent* Model/View/Controller design pattern can be used in the intrinsically *location-dependent* environment of partitioned web-applications. *Flexible web-application partitioning* enables the code used to develop a stand-alone application and in a single address-space, to be used (without modification) in deployments to various client platforms and with any desired partitioning scheme.

fwap encourages an approach to web-application deployment in which the application's "scenario flows" are continuously studied to get insight about how clients actually use the application. By understanding the scenario flows, the application can be partitioned in a way that improves performance. In contrast, traditional implementation techniques require that such analysis be performed only in the design and requirements phase because it is much too costly to repartition the application once it is deployed. Unfortunately, the necessary insights can often be made only after the application has been deployed and in production for some time. Repartitioning, under *fwap*, imposes no extra cost; an application can therefore be readily tuned after deployment based on feedback from actual client use.

We are currently implementing the algorithms and infrastructure needed to enable fwaplications to scale over non-trivial application Models. We are also working with a customer to validate the *fwap* concepts and implementation.

7. References

1. Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S., Shuster, J., *UIML: AnAppliance-Independent XML User Interface Language*, Proceedings of the Eight International World Wide Web Conference, May, 1999, 617-630.

2. *Barracuda: Open Source Presentation Framework*, http://barracuda.enhydra.org/, 2001.

3. Beck, K., *Extreme Programming Explained: Embrace Change (XP Series)*, Addison-Wesley Pub Co., 1999.

4. Bennett, B. et al, *A distributed object oriented framework to offer transactional support for long running business processes*, IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000).

5. Bergsten, Hans, *JavaServer Pages*, O'Reilly, 2000.

6. Betz, K., Leff, A., Rayfield, J., *Developing Highly-Responsive User Interfaces with DHTML and Servlets"*, Proceedings of the 19th IEEE International Performance, Computing, and Communications Conference -- IPCCC-2000, 2000.

7. Buschmann, F. et al, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, 1996, 123-168.

8. Coutaz, J., *PAC, An Object-Oriented Model for Dialog Design*, Elsevier Science Publishers, Proceedings of Human-Computer Interaction - INTERACT, 1987, 431-436.

9. *Enterprise JavaBeans Specifications*, http://java.sun.com/products/ejb/docs.html, 2001.

10. *JAVA PLUG-IN 1.2 SOFTWARE FAQ*, http://java.sun.com/products/plugin/1.2/plugin.faq.html , 2001.

11. Flanagan, David, *JavaScript: The Definitive Guide*, 3rd, O'Reilly, 1998.

12. Gray, G. and Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

13. *JAVA SERVLET TECHNOLOGY IMPLEMENTATIONS & SPECIFICATIONS*, http://java.sun.com/products/servlet/download.html#specs , 2001.

14. *Java 2 Platform, Micro Edition (J2ME)*, http://java.sun.com/j2me/, 2001.

15. G.E. Krasner and S.T. Pope, *A Cookbook for Using the Model-View-Controller User-Interface Paradigm in Smalltalk-80*, SIGS Publication, 26-49, Journal of Object-Oriented Programming, August/September, 1988.

16. *Struts*, http://jakarta.apache.org/struts/index.html, 2001.

17. *WebWork*, http://sourceforge.net/projects/webwork, 2001.