Model View Controller Design for Multiple Mode Access

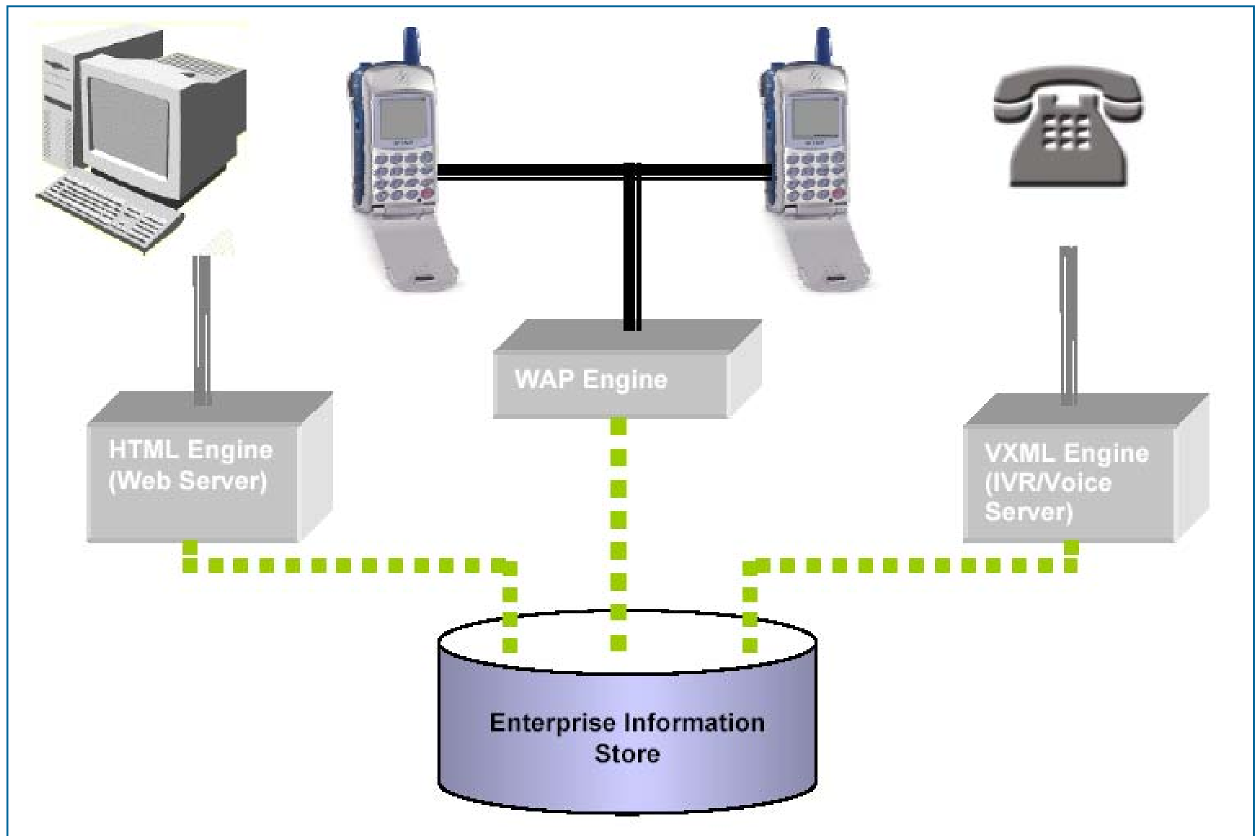
A White Paper

This document is a property of Analytica India Pvt. Ltd. No part of this document may be copied or reproduced in any form or by any means, electronic, mechanical or otherwise without the prior

Analytica India Pvt. Ltd.
472/9 AVS Compound, 80 Ft Road
Koramangala, Bangalore INDIA 560 034
Phone: +011 91 80 552 7130 Email: info@analytica-india.com

Introduction

Now, more than ever, enterprise applications need to support multiple types of users with multiple types of interfaces. For example, an online store may require an HTML front for Web customers, a WML front for wireless customers, a JFC/Swing interface for administrators, and an When developing an application to support a single type of client, it is sometimes beneficial to interweave data access logic with interface-specific logic for presentation and control.



Such an approach, however, is inadequate when applied to enterprise systems that need to support multiple types of clients:

- Different applications need to be developed, one to support each type of client interface.
- Non-interface-specific code is duplicated in each application, resulting in duplicate efforts in implementation (often of the copy-and-paste variety), as well as testing and maintenance.
- The task of determining what to duplicate is expensive in itself, since interface-specific and non-interface-specific code are intertwined.
- The duplication efforts are inevitably imperfect. Slowly, but surely, applications that are supposed to provide the same core functionality evolve into different systems.

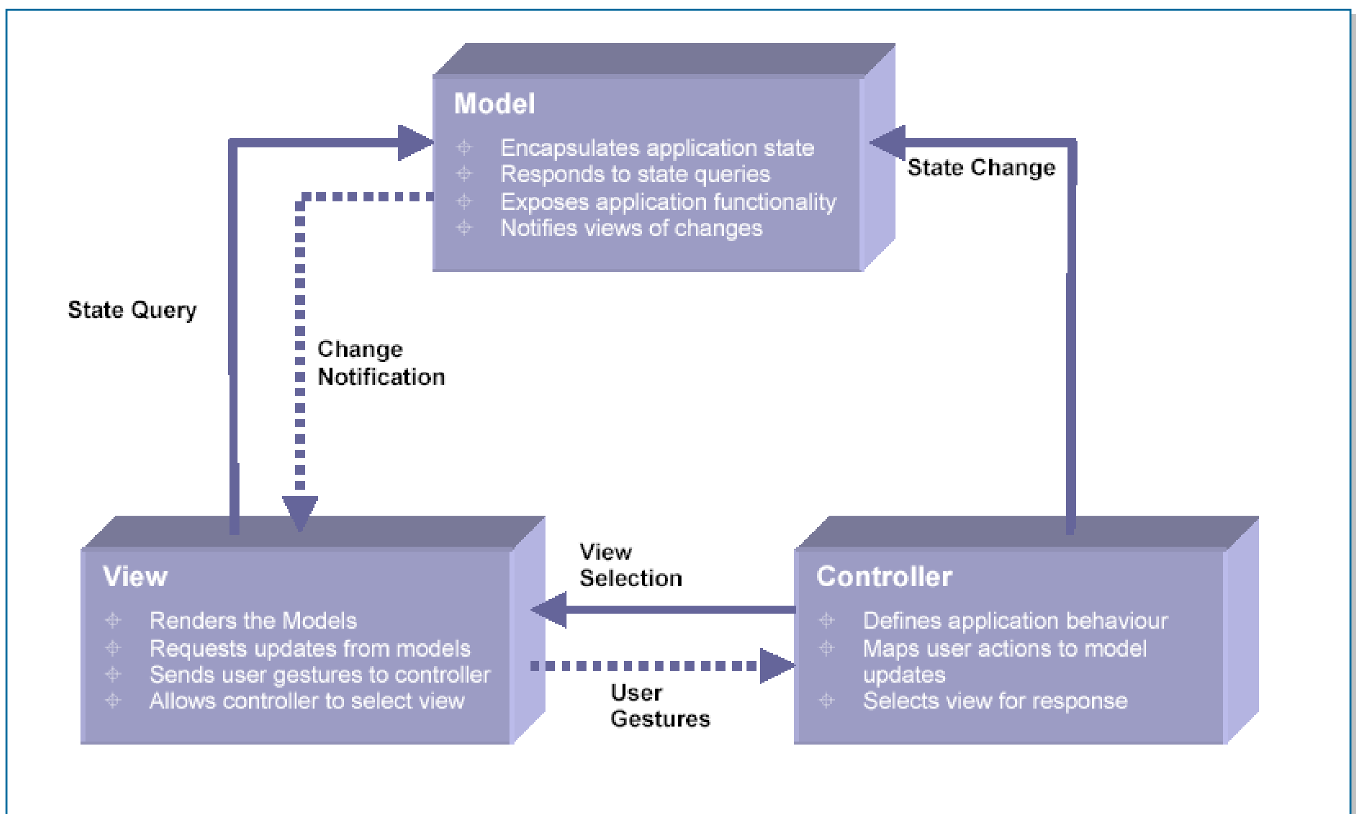
The same enterprise data needs to be accessed when presented in different views: *e.g.* HTML, WML, JFC/Swing, XML. The same enterprise data needs to be updated through different interactions: *e.g.* link selections on an HTML page or WML card, button clicks on a JFC/Swing GUI, SOAP messages written in XML. Supporting multiple types of views and interactions should not impact the components that provide the core functionality of the enterprise application.

By applying the Model-View-Controller (MVC) architecture to a J2EE application, we separate core data access functionality from the presentation and control logic that uses this functionality. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain.

The **model** represents enterprise data and the business rules that govern access to and updates of this data. Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.

- A **view** renders the contents of a model. It accesses enterprise data through the model and specifies how that data should be presented. It is the view's responsibility to maintain consistency in its presentation when the model changes. This can be achieved by using a *push* model, where the view registers itself with the model for change notifications, or a *pull* model, where the view is responsible for calling the model when it needs to retrieve the most current data.

- A **controller** translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.



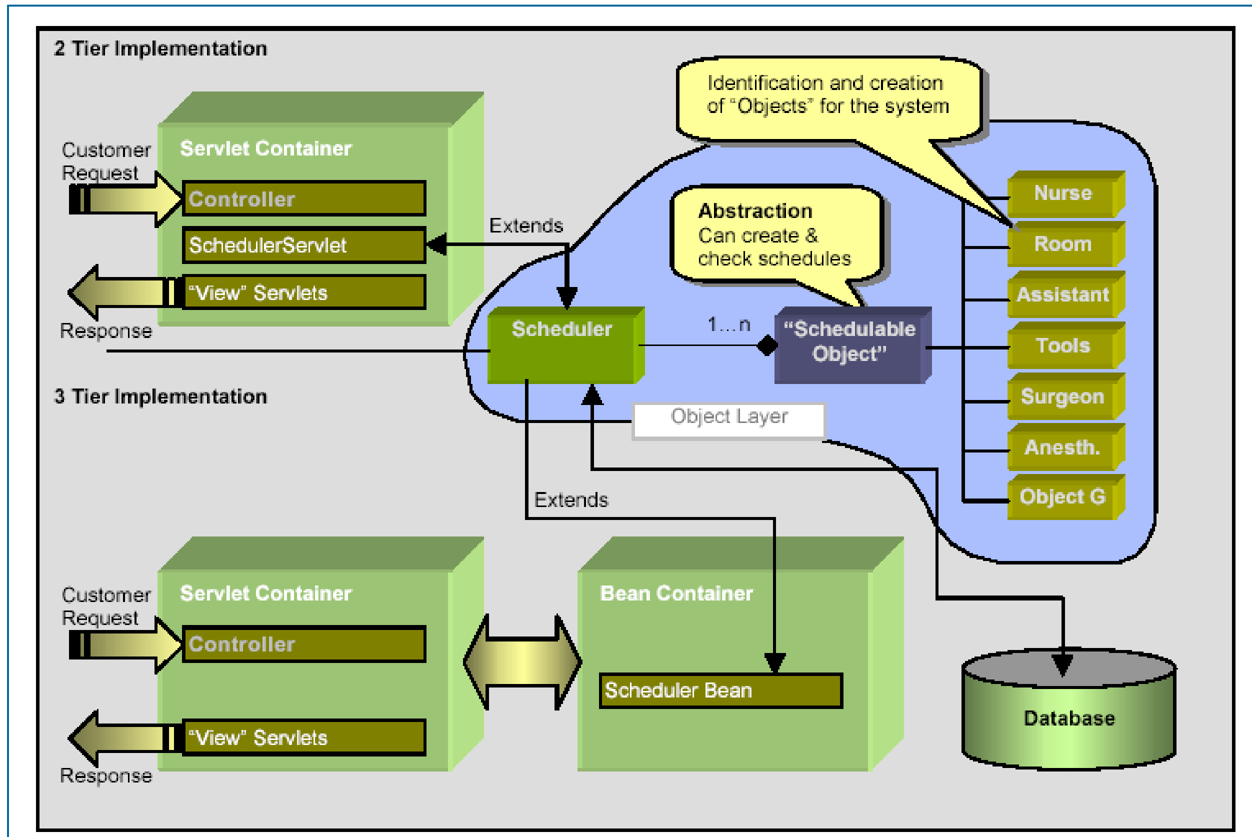
The MVC architecture has the following **benefits**:

- **Multiple views using the same model.** The separation of model and view allows multiple views to use the same enterprise model. Consequently, an enterprise application's model components are easier to implement, test, and maintain, since all access to the model goes through these components.

- **Easier support for new types of clients.** To support a new type of client, you simply write a view and controller for it and wire them into the existing enterprise model.

Design Guidelines

The following design guidelines ensure that all the criteria of modularity, extensibility, rational functional decomposition, code reuse, and multiple-view access are met.



General Guidelines

1. Identification of all "objects" in the design, along with their properties. These objects are implemented, with the proper "get" and "set" methods, in order to build modularity and code-reuse into the system from the beginning.

2. Abstraction at the object level should be used as much as possible in order to make the logic engines more general and widely usable. For instance, the creation of a general "SchedulableObjects" object for a scheduler, instead of specific resources like "Room", "Nurse" etc.

3. All objects should be optimized to be useful to both display components ("View") as well as logic ("model") components, and should be, as far as possible, made common to both.

4. Communication between the different layers of the deployment architecture (2 or 3 tier) are identified, and encrypted wherever there is the possibility of a compromise. This includes almost all information access that happens from outside the system being designed.

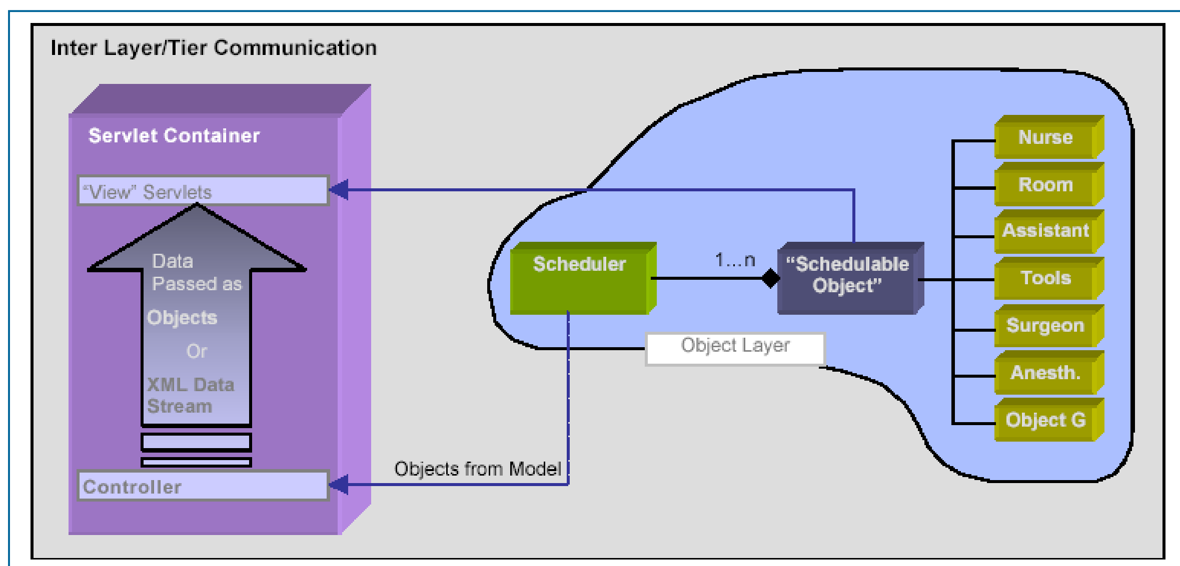
5. All login/password/authentication related communication is always encrypted. The encryption, by default, makes use of the Analytica AES encryption module, which conforms to NIST specifications for security.

View Guidelines

1. Identify the "View" technology – this may be either servlets, or JSP/ASP pages.

2. If the view technology is "servlets", then a BaseServlet + Template concept is used in order to provide further flexibility in changing the user interface/look and feel of the pages.

3. The view elements (servlets/JSP/ASP pages) should use the system objects with their get/set methods to retrieve values and display it on the page to the user. Alternatively, the view may receive all data from the controller in the form of an XML formatted data stream.



Controller Guidelines

1. Use a standard controller architecture with a XML based configuration for the maximum flexibility in deployment of the solution. Identify the sequence/logic in which the views are to be displayed to the user. The standard controller is written in Java in order to allow for container (web server) managed run-time pooling, but other technologies like JSP/ASP may also be used. As an alternative, an Apache Struts based implementation may also be used.

2. The controller should contain an integrated authentication and session tracking mechanism (which may be implemented with the “model”), in order to enforce and ensure security between the user and the system.

3. The controller should retrieve data from the “model” elements as system “objects”, and convey them to the “view” either as native objects, or as an XML formatted data stream, based upon the “view” request.

Model Guidelines

1. Identify the logic blocks for the entire system - these are the major “decision making” elements of the system. Typical cases include the “Scheduler” in a scheduling system, which can perform all activities like checking for free slots, creating a reservation/schedule etc. These engines should also be object oriented and modular.

2. One of the most important “blocks” that are implemented is one for authentication. This authentication module is implemented either as a separate class (2 tier architecture) or a bean. If the system is integrated with other parts of the enterprise, and if user authentication information is supplied external to the system, then the communication is encrypted to ensure security.

3. The blocks are then made into either objects on the system (for a 2-tier MVC architecture) or EJB’s to be deployed in a middleware server for a 3-tier implementation.