**Report No.4: Software Design Description (SDD)**

# 4.1. Design Overview

# 4.2. System Architectural Design

## 4.2.1 Choice of System Architecture

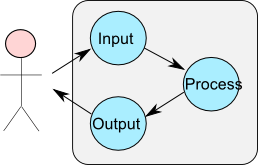
The following diagram represents the Model-View-controller pattern:



Model-View-Controller Diagram[1]

Model-View-Controller (herein referred to as MVC) is a software design pattern that can be used to organize code in such a way that the business logic and data presentation are separate. The premise behind this approach is that if the business logic is grouped into one section, then the interface and user interaction that surrounds the data can be revised and customized without having to reprogram the business logic. MVC was originally developed to map the traditional input, processing, output roles into a logical GUI architecture.

[1] <http://netbeans.org/images_www/articles/69/javaee/ecommerce/design/mvc-model.png>



These three main roles are the basis for the Joomla MVC. They are described here in brief, but for a more thorough explanation, please refer to the links provided at the end of this tutorial.

### Model

The model is the part of the component that encapsulates the application's data. It will often provide routines to manage and manipulate this data in a meaningful way in addition to routines that retrieve the data from the model. In our case, the model will contain methods to add, remove and update information about the greetings in the database. It will also contain methods to retrieve the list of greetings from the database. In general, the underlying data access technique should be encapsulated in the model. In this way, if an application is to be moved from a system that utilizes a flat file to store its information to a system that uses a database, the model is the only element that needs to be changed, not the view or the controller.

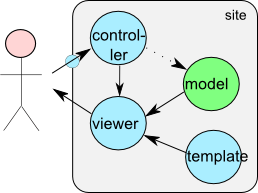
### View

The view is the part of the component that is used to render the data from the model in a manner that is suitable for interaction. For a web-based application, the view would generally be an HTML page that is returned to the user. The view pulls data from the model (which is passed to it from the controller) and feeds the data into a template which is populated and presented to the user. The view does not cause the data to be modified in any way, it only displays data retrieved from the model.

### Controller

The controller is responsible for responding to user actions. In the case of a web application, a user action is (generally) a page request. The controller will determine what request is being made by the user and respond appropriately by triggering the model to manipulate the data appropriately and passing the model into the view. The controller does not display the data in the model, it only triggers methods in the model which modify the data, and then pass the model into the view which displays the data.

### MVC connection

[](http://docs.joomla.org/File:MVC_joomla.png)

The simplified picture on the right depicts the basic components being used within Joomla. Besides the Model, the View and the Controller, an Entry Point has been added that is depicted as a small circle. Attached to the viewer (view) a Template has been added. With these five components you should be able to understand this tutorial about making a basic Joomla! MVC component.

Part 1 of the tutorial only focuses on the Controller and the View (with the use of the Template); these are marked with the blue color in the picture. Part 2 adds and Part 3 extends the model functionality for the data manipulation abstraction; marked with the green color in the picture.

Keep in mind that this simplified picture only applies for the site section (i.e the front-end). An identical picture is applicable for the admin section (i.e. the back-end). The administrative section is taken care of in Parts 4 through 6 of this component development tutorial. Both the site and the admin section are maintained and configured in an XML based installation file (typically termed a manifest file).

**Why should MVC?**

* **Re-use of Model components:** 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 some controller logic and wire them into the existing enterprise application.
* **Increased design complexity:** This pattern introduces some extra classes due to the separation of model, view, and controller.

**Disadvantage**

* Require more technical skill to understand and implement
* Time consuming
* Easy debugging
* Maintaining file structure is import

## 4.2.2 Discussion of Alternative Designs

<<>>

## 4.2.3 Description of System Interface

# 4.3. Component Diagram

# 4.4. Detailed Description of Components

## 4.4.1. CRC Cards (Class-Responsibility-Collaborators)

4.4.1.*n.* Component-*n*

4.4.1.n.1. Class Diagram

4.4.2.n.2. Class Diagram Explanation

*<Provide a brief explanation of the class diagram above. You do not need to explain “obvious” parts of your class diagram>*

4.4.3.n.3. Algorithms of important methods in each class specified in pseudo code or by Flow-Chart

# 4.5. Sequence Diagram

# 4.6. User Interface Design

## 4.6.1 Description of the User Interface

4.12.1.1 Screen Images

4.12.1.2 Objects and Actions

# 4.7. Database Design or Data Structures

*<Provide the detailed database design for the system here. If your team uses a file or in-memory storage facility instead of database, remove this section; use the ‘Data Structures’ section. >*

# 4.8. Other material (if any)