**Chapter 3**

In this Chapter, we will discuss about a very famous design pattern, known as MVC(Model View Controller), so let's get started!!

Now let say, we are going to develop a dynamic web application with the help of servlets. So at first I considered having just a single servlet, with a bunch of IF tests, but then decided that separate servlets would be more OO- each servlet should have one responsibility like the query page, the sign up page, the search results page etc etc. SOUNDS GREAT!!

Now each servlet will have all the business logic it needs to modify or read the database and prints the HTML to the response stream back to client. Some thing like mentioned below:

public class MyServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws IOException{

// business logic goes here

PrintWriter out = response.getWrtiter();

// compose the dynamic HTMl page

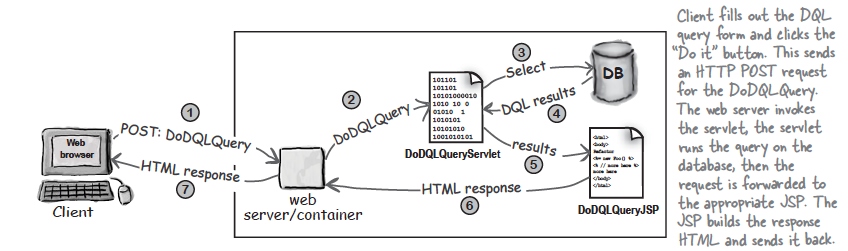
out.println(“something really ugly goes here!!”);

}

}

But then it gets ugly, so he adds JSPs

Those pesky println() statements for the output response get really ugly, really quickly. So he reads up on JSPs and decides to have each servlet do whatever business logic it needs to do (query the database, insert or update a new record, etc.) *then forward the request to a JSP* to do the HTML for the response. This also separates the *business logic* from the *presentation*... and since he’s been reading up on design, he knows that *separation of concerns* is a Good Thing.



But then I come to know about... MVC !!

One of my friend wants to know that if my web application from a Swing GUI application... No I hadn't thought of that.. then my friend replied Well it should n't be a problem because I am sure you

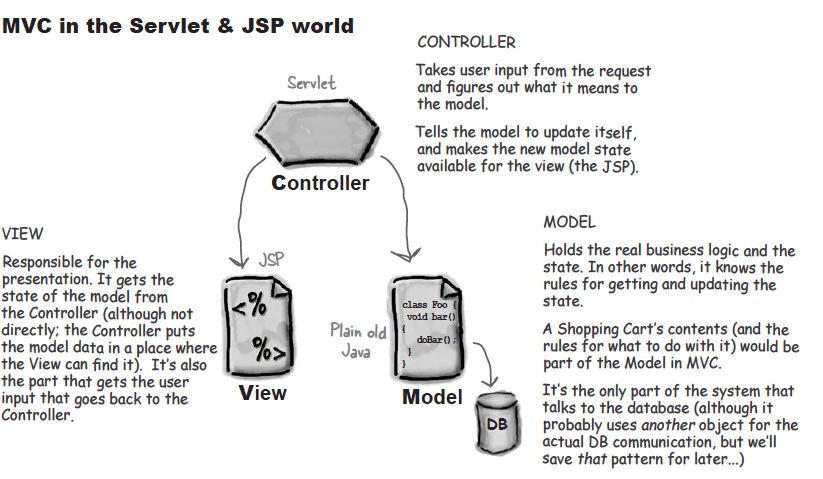
have used MVC so we can just whip up a Swing GUI client that can access the business logic classes then me replied ohhhh... WHAT !!! MVC??

Before digging into MVC, lets first discuss why we should use MVC architecture

With MVC the business logic is not only separated from the presentation.. it does n't even know that there is a presentation.

The essence of MVC is that you separate the business logic from the presentation, but put something between them so that the business logic can stand on its own as a reusable Java class, and doesn’t have to know anything about the view.

Model \* View \* Controller (MVC) takes the business logic out of the servlet, and puts it in a “Model”— a reusable plain old Java class. The Model is a combination of the business data (like the state of a Shopping Cart) and the methods (rules) that operate on that data.



Then one another friend of mine says...while it IS an MVC design, it’s a dumb one. Sure, the business logic has been pulled out into a Model, and the servlets act as the Controllers working between the Models and Views so that the Models can be brain-dead about the Views. That’s all good. But look at all those little servlets.

What do they even do? Now that the business logic is safely tucked away in the Model, the servlet Controller isn’t doing much except some generic application stuff for this app, and, oh yeah, it does update the Model and then it kicks the View into gear.

But the worst part is that all that generic application logic is duplicated in every single frickin’ servlet! If one thing needs to change, it has to change everywhere. A maintenance train wreck waiting to happen.

Yeah, I felt a little weird about the duplicate code, “but what else can I do? Surely you don’t mean for me to put everything in a single servlet again? How could that be good?”

Don't worry about the answer till now... we will cover this in the end.