**Shopping Cart**

**This is for you, if:**

* You want to get hands on feel of JSP and Servlets in real life scenarios
* You are dying to write some code and see it work
* You have heard enough about shopping carts and finally want to see what the heck, they are made of!

**This is not for you, if:**

* You do not have any idea about JSP, Servlets or core Java
* You are looking for a production quality or ready to use shopping cart 

**Pre-requisites:**

* Knowledge of Java fundamentals and collection framework
* JSP and servlet knowledge, especially, about session handling and JSTL
* Setup of environment: Tomcat and IDE.

**Requirement:**

* Create **simple catalog page** with hard coded list of models with an ‘Add to Cart’ button.
* The catalog model list page should **contain a list of models** with model number, description and price.
* On clicking the ‘Add to Cart’ button, the **model should get added to the shopping cart**.
* The cart should display all the models added to it. For each line item, it should **display model number, description and unit price of item and total price.** It should also display the total price for the whole order.
* For each item in the cart, there should be **a delete button next to it for removing the item** from cart.
* For each item in the cart, there should be a **text input to enter a quantity value** with update button to update the quantity of item in the cart.
* The cart should have **a link back to catalog model list** page.

**Shopping Cart overview:**

Shopping cart is the heart of any e-commerce application. It is something highly talked about. But, what do you mean by a session based shopping cart? Well, shopping carts can be mainly session based or database based. As the name indicates, session based carts are completely maintained in the user’s session. Only after the checkout is completed, the information is persisted in the database. On the other hand, a database based shopping cart is maintained in the database. Every time cart items are added or updated, it gets persisted in the database. Both of them have their own advantages and disadvantages. For each add or update in database based cart, there will be a hit in the database. On the other hand, it gives us the advantage of cart persistence. You can leave your shopping halfway and come back after sometime and resume the shopping session.

**Analysis and Design:**  
  
Designing the components prior to coding is pretty useful. Better the designing, easier it is to code and lesser the rework. We will be digging pretty deep into the design to throw light on some of the design principles. If you are not interested in designing and stuff, you may want to glance through the UML diagrams and move on to code.  
  
*Identifying components:*  
Let us analyze the requirements, and try to identify the different components needed for this workshop. For our analysis, we try to identify components based on **Model-View-Controller (MVC) design pattern**.  
  
We need to create a **ModelList page**, with ‘Add to cart’ button.  
  
Technically speaking, what does a cart comprise? **It is a collection of items added to the session.** Let us use, **ArrayList** for our cart. So, the basic idea is to have an ArrayList added to session object.   
  
We will refine this a bit. Going forward, we may need the cart item to hold other things than just CartItems. Something like Promotion code data. So, we will maintain a special cart object. And the **cart will contain ArrayList of cart items**.   
  
But, what will the **ArrayList** comprise? String will not do for sure. Probably, we need **a special item bean**. Right, so we will have **ItemBean** added to ArrayList, and the ArrayList will be added to session. This sounds great!  
  
We need a **shopping cart jsp**. It would need to pull ArrayList out of the session and iterate this, and display it in the JSP.  
  
Aren’t we missing something? We need something to control the events coming from the ‘Add to cart’, ‘Update Item’ and ‘Delete Item’. We need a **controller** to handle these events. The **servlet** should be the one to do this job.   
  
Based on the above analysis, let us summarize the components we had identified:  
**1. ModelList.jsp   
2. ShoppingCart.jsp  
3. CartBean.java  
4. CartItemBean.java  
5. CartController.java**  
  
**Identifying attributes for cart:**  
Now, let us try to identify more information for the cart related classes.   
  
**CartBean**:

* ArrayList of CartItemBean
* totalCost of items in cart

**CartItemBean**:

* Part number for the model
* Model description
* Unit cost
* Quantity
* Total item cost

**Identifying methods:**  
**CartController**:   
This servlet controller needs to implement the **doGet/doPost methods**. It needs to handle the events for **addToCart**, **updateCart** and **deleteCart**. It is a better design to handle each of these events separately. So, based on this discussion, the CartController will need to have the following methods:

* **doGet/doPost** – Delegates request to different event methods like addToCart etc.
* **addToCart**  – Handle the add to cart functionality
* **updateCart** – Handle the update cart item functionality
* **deleteCart** – Handle delete cart item from cart functionality

Let us work through the ‘Calculate the cart total’ and ‘Add to cart’ functionality to analyze and come up with methods related to it.

**Calculating the cart total**:  
This is a major functionality, which needs to be utilized every time an item is added, removed or updated. Let us discuss considering different design principles and try to come up with a realization for this functionality.

**General responsibility assignment software patterns** (or **principles**), abbreviated **GRASP**, consist of guidelines for assigning responsibility to classes and objects in [object-oriented design](https://en.wikipedia.org/wiki/Object-oriented_design).

The different patterns and principles used in GRASP are: controller, creator, indirection, information expert, high [cohesion](https://en.wikipedia.org/wiki/Cohesion_(computer_science)), low [coupling](https://en.wikipedia.org/wiki/Coupling_(computer_science)), [polymorphism](https://en.wikipedia.org/wiki/Polymorphism_(object-oriented_programming)), protected variations, and pure fabrication. All these patterns answer some [software](https://en.wikipedia.org/wiki/Software) problem, and these problems are common to almost every [software development](https://en.wikipedia.org/wiki/Software_development) project. These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested [programming](https://en.wikipedia.org/wiki/Computer_programming) principles in object-oriented design.

Computer scientist [Craig Larman](https://en.wikipedia.org/wiki/Craig_Larman) states that "the critical design tool for software development is a mind well educated in design principles. It is not [UML](https://en.wikipedia.org/wiki/Unified_Modeling_Language) or any other technology."[[1]](https://en.wikipedia.org/wiki/GRASP_(object-oriented_design))Thus, GRASP are really a mental toolset, a learning aid to help in the design of object-oriented software.

**Information expert** (also **expert** or the **expert principle**) is a **principle** used to determine where to delegate responsibilities. These responsibilities include methods, computed fields, and so on. ...**Information expert** will lead to placing the responsibility on the class with the most **information** required to fulfill it.

**Information expert principle** says that, any object which has the information to perform an operation should do the job.  
For calculating the total price, we need to first know the total for each line item. So, let us apply expert principle to get the line item total. Since, **CartItemBean** has both quantity and unit cost, based on expert, it is the right candidate.   
  
Just to understand the **association** between the different objects, we know, that CartBean houses a list of CartItemBean. The CartController on the other hand will be one to add the CartBean in session. **Now, this means, CartBean is associated with CartItemBean and CartBean is associated also with CartController.**   
  
Coupling design principle says that, **the association or coupling between different objects should be as less as possible.** More the coupling, more **spaghetti** code syndrome. **You make changes to code in one class; all the associated classes need changes.**  
  
Since CartController has access to CartBean, we can write a calculateTotal method in the CartController, where we will have to pull the list of CartItemBeans from CartBean in this method and calculate the total price. However, this means, iterating through the CartItemBean list in CartController and also calls on CartItemBean.getSubtotal(). This introduces CartItemBean to CartController coupling.  
  
But, do we have a choice? Going again by expert, CartBean does have access to all CartItemBeans. So, it is an expert to calculate the order total. Also, it does not have a CartItemBean to CartController coupling. So, this is a better choice.

**Add to Cart functionality**:

We said that CartBean will house an ArrayList of CartItemBeans. So, we will have a **getter/setter** for setting this ArrayList inside CartBean. But, who will create the ArrayList of CartItemBeans and then call setCartItems? It has to be CartController. This intern means, CartItemBean creation, setting values and adding to ArrayList, needs to be performed in CartController. But, we have been working so hard eliminate the CartController to CartItemBean coupling while designing the ‘calculate order total’ functionality. Even with this, creating the ArrayList of CartItemBeans and setting it in CartBean every time is a tall order. We need something like addCartItem(CartItemBean) in CartBean. This will add to the ArrayList of CartItemBeans every time this is called. So, the CartController does not have to worry above the ArrayList and setting the ArrayList using the setCartItem method. That is a great relief!!   
  
But, when we still will call setters in CartItemBean from CartController and then call the CartBean.addCartItem(CartItemBean). Isn’t it not adding coupling between CartController and CartItemBean. Well, sad but true! Let us try to look for any some more options.  
  
It would be dreamy, if CartBean can take complete control of the CartItemBean, without any CartController intervention. Let us try to delegate the task of creating CartItemBean and setting values into it to CartBean. But, is it possible; is the question to be answered? How will the CartBean know, what should be the state of the CartItemBean created? Well, since the CartController has this information, it can pass on this information to CartBeans method as parameters. Like

* addCartItem(String strModelNo, String strDescription, String strUnitCost, String strQuantity)

Passing 4 parameters in the method call may not be that bad! But, in a real production cart scenario, we may need more parameters. We can generally work around this by using a **value object**. But for our example this is fine.   
  
We just designed the addToCart without increasing the coupling. Note here that, CartController is the information expert. The CartBean may be considered as a partial expert. However, we considered both coupling and expert to come up with this design. This leads us to something very important while using design principles. No single design principle can be used in isolation to design functionality. We need to consider different design principles to come up with a design.  
  
We will use the same principle for updateCartItem and deleteCartItem functionality. You may want to think over it.  
  
So, based on the above discussion, below the **class diagram** for our cart sub-system.

**Sequence diagram** for Add To Cart:



**Setting up the environment for development:**

We will be using **Tomcat 8.0.x** for running and testing our shopping cart.

We will be developing using **Eclipse Java EE IDE for Web Developers. Version: Oxygen.2 Release (4.7.2).**  
  
Let us create a new Tomcat Project, with context name as /ShoppingCart.  
Set the source path and build path correctly. We will need **JSTL libraries (jstl.jar and standard.jar) as we will be using JSTL/EL for the shopping cart page.**  
  
We should be able to start Tomcat from Eclipse, and access the any JSP page inside /ShoppingCart context which we created using the URL [http://localhost:8080/ShoppingCart/<AnyFile.jsp>](http://localhost:8080/ShoppingCart/%3CAnyFile.jsp%3E)  
  
The following will be approximate directory structure of the ShoppingCart project.



**Coding’ the Cart:**  
Such detailed design generally makes coding just a formality. Below is the code for each of the element identified during design. The code is pretty straight forward. 

**CartItemBean.java**

**CartBean.java**

**CartController.java**

Do not ever maintain any state in servlet’s instance variables. You may find mixing of data between requests or user sessions. The best part of this is that, it will only happen in a production environment and you win a severity 1 ticket and that too completely free!

**ModelList.jsp**

**ShoppingCart.jsp**

For iterating the elements in the cart session, we are using simple JSTL tags with EL.

**web.xml**

**Putting the cart to test:**  
We mainly want to test the below three functionalities:  
1. Add to cart  
2. Update the quantity  
3. Remove an item from cart

**How to Run the Application**  
Start access URL <http://localhost:8080/ShoppingCart/ModelList.jsp> to test the above functionality Or right click on ModelList.jsp, Run as server.