* 1. **Primary presentation:** 
     1. **Layered Style**



* + 1. **Uses Style**



* 1. **Element catalog:**
     1. **Elements and their properties**

|  |  |  |
| --- | --- | --- |
| Elements | | Properties |
| Controller | **ProductControllers** | The ProductController class contains action methods that render view pages (AddProduct, EditProduct, ViewProduct) |
| **Category**  **Controllers** | The CategoryController class contains action methods that render view pages (AddCategory, EditCategory, ViewCategory) |
| **StatisticsControllers** | The StatisticsController class contains action methods that render view pages (Statistics) |
| **SaleControllers** | The SaleController class contains action methods that render view pages (BillManagement, Checkout, PriceLog) |
| **StoreControllers** | The StoreController class contains action methods that render view pages (AddStore, ViewStore, EditStore) |
| **LoyalMember**  **Controllers** | The LoyalMemberController class contains action methods that render view pages (AddLoyalMember, ViewLoyalMember, EditLoyalMember) |
| **StoreCategory**  **Controllers** | The StoreCategoryController class contains action methods that render view pages (AddStoreCategory, ViewStoreCategory, EditStoreCategory) |
| **ComputerControllers** | The POSController class contains action methods that render view pages (AddPOS, ViewPOS, EditPOS) |
| **UserControllers** | The UserController class contains action methods that render view pages (AddUser, ViewUser, EditUser) |
| View | **AddStore** | This GUI helps the user to add a new Store. |
| **ViewStore** | This GUI helps the user to view the Store. |
| **EditStore** | This GUI helps the user to modify some information about the Store. |
| **AddCategory** | This GUI helps the user to add a new Product Category. |
| **ViewCategory** | This GUI helps the user to view the Product Category. |
| **EditCategory** | This GUI helps the user to modify some information about the Product Category. |
| **AddProduct** | This GUI helps the user to add a new Product. |
| **ViewProduct** | This GUI helps the user to view the Product. |
| **EditProduct** | This GUI helps the user to modify some information about the Product. |
| **BillManagement** | This GUI helps the user to view the list of Bills and the user can see some detail information of each Bill. |
| **PriceLog** | This GUI shows all Price history that was used for each product |
| **Checkout** | This GUI helps the user check bill and make a payment. |
| **AddUser** | This GUI helps the user to add a new User. |
| **ViewUser** | This GUI helps the user to view the User. |
| **EditUser** | This GUI helps user to modify some information about the User. |
| **AddLoyalMember** | This GUI helps the user to add a new loyal Customer. |
| **ViewLoyalMember** | This GUI helps the user to view the loyal Customer. |
| **EditLoyalMember** | This GUI helps user to modify some information about the loyal Customer. |
| **AddStoreCategory** | This GUI helps the user to add a new Retail Store Category. |
| **ViewStoreCategory** | This GUI helps the user to view the Retail Store Category. |
| **EditStoreCategory** | This GUI helps user to modify some information about the Retail Store Category. |
| **AddComputer** | This GUI helps the user to add a new Computer. |
| **EditComputer** | This GUI helps user to modify some information about the Computer. |
| **ViewComputer** | This GUI helps the user to view the Computer. |
| **Statistics** | To make statistics about the total amount of product (or product category) was bought on month. |
| Model | **ProductModels** | Storing and retrieving the Product information and return a message back to view pages (AddProduct, EditProduct, ViewProduct) |
| **SaleModels** | Storing and retrieving the Sale information and return a message back to view pages (BillManagement, Checkout, PriceLog) |
| **StoreModels** | Storing and retrieving the Retail Store information and return a message back to view pages (AddStore, EditStore, ViewStore) |
| **LoyalMember**  **Models** | Storing and retrieving the Customer information and return a message back to view pages (AddLoyalMember, EditLoyalMember, ViewLoyalMember) |
| **UserModels** | Storing and retrieving the Userinformation and return a message back to view pages (AddUser, EditUser, ViewUser) |
| **CategoryModels** | Storing and retrieving the Product Categoryinformation and return a message back to view pages (AddProductCategory, EditProductCategory, ViewProductCategory) |
| **StoreCategory**  **Models** | Storing and retrieving the Store Category information and return a message back to view pages (AddStoreCategory, EditStoreCategory, ViewStoreCategory) |
| **ComputerModels** | Storing and retrieving the POSinformation and return a message back to view pages (AddPOS, EditPOS, ViewPOS) |
| **StatisticsModels** | Storing and retrieving the some information related Statistics and return a message back to view pages (Statistics) |
| **DBContext** | Context class is the primary class for interacting with data as objects that are instances of entity types that are defined in an Entity Data Model (EDM). |

* + 1. **Relations and their properties**

|  |  |
| --- | --- |
| **Connector** | **Properties** |
| **Allowed to use** | The layers are related to each other by the strictly ordered relation allowed to use. |
| **Uses** | The uses style shows how modules depend on each other; it is helpful for planning because it helps define subsets and increments of the system being developed. |

* 1. **Context diagram:**
  2. **Architecture background:**

System was separate into three layers include: Controller, Model, View

The reason why we use three layers instead of four layers (such as: Presentation layer, Business Logic layer, Data Object Transfer Layer, Data Access Layer) is:

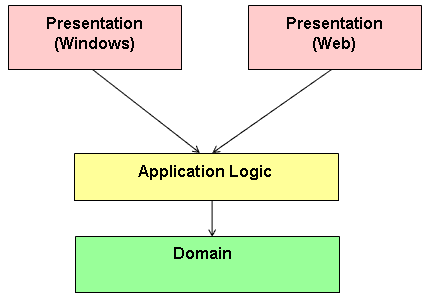
* The controller in MVC undertake tasks that is implemented on Business Logic Layer and Data Access Layer. Moreover, View render a webpage to client as a UI and controller can support a view by returning appropriate view to client base on its request.
* On Web environment, increasing performance is important. Therefore, eliminating a layer is needed.
* Also, the scope of system isn’t large and doesn’t require much more complex business logic calculations, thus, MVC framework can fulfill its job.

Let’s get to the bottom of MVC framework:

* The view is responsible for providing the user interface (UI) to the user. It is given a reference to the model, and it transforms that model into a format ready to be presented to the user.
* The controller is responsible for responding to user input, often making changes to the model in response to user input. In this way, controllers in the MVC pattern are concerned with the flow of the application, working with data coming in, and providing data going out to the relevant view.
* The model that is used to send information to the Data Access Layer, perform business calculations, and even render in a view. Otherwise, these objects represent the domain of the application focuses on, and the models are the objects you want to save, create, update, and delete. In model, we use entity framework because its benefit:
  + With Entity Framework, the developers issue queries using LINQ, then retrieve and manipulate data as strongly typed objects. The Entity Framework’s ORM implementation provides services like change tracking, identity resolution, lazy loading, and query translation so that developers can focus on their application-specific business logic rather than the data access fundamentals.
  + Its goal is to decrease the amount of code and maintenance required for data-oriented applications. Entity Framework applications provide the following benefits:
    - Applications can work in terms of a more application-centric conceptual model, including types with inheritance, complex members, and relationships.
    - Applications are freed from hard-coded dependencies on a particular data engine or storage schema.
    - Mappings between the conceptual model and the storage-specific schema can change without changing the application code.
    - Developers can work with a consistent application object model that can be mapped to various storage schemas, possibly implemented in different database management systems.
    - Multiple conceptual models can be mapped to a single storage schema.
    - Language-integrated query (LINQ) support provides compile-time syntax validation for queries against a conceptual model.

There is the list of MVC# framework features that is the reason why we use this:

* *Views and controllers get connected automatically*. The MVC framework automatically establishes links between views and corresponding controllers. Therefore, the developers do not care about the associating views and controllers that linked to their views.
* *Multiple GUI platforms supported.* MVC allows targeting different GUI platforms such as: Window, Web, Silverlight, etc,… Therefore, the same application can be used with quite different presentation layers - one for Windows, the other for Silverlight or Web environment, etc.:



### *Platform-independent navigation to views*. To make application logic fully independent of the presentation layer, MVC provides a platform-independent way of navigating to views. Instead of  activating a Windows form or redirecting to a Web page a developer just simply call a uniform Navigator.Navigate(...) method. For example:

public class OrderDetailsController

...

public void ProcessOrder()

{

// No Response.Redirect(...) or Form.Show() calls

Task.Navigator.Navigate(OrderSupportTask.ProcessOrder);

}

### *Tasks concept.* Sometime, we have to unites several views with their controllers to do some job, this is called a task. For example a checkout task may consists of two views, one to choose a product (such as: Milk, Drink, Cake,… we can order in supermarket), the other – to do the payment. In MVC# all controllers within a task are given a link to the task object. Generally a task can be expressed as a workflow or a state machine.

### 