EatSmart Vending Machine

Java Desktop Application using design patterns

Project Discussion Document

Kavya Gautam

Ramya Bangaluru Gopalakrishna

Swathi Damera

1. Project Description 3

1.1 Problem Statement 3

1.2 Scope of the project 3

1.3 Requirements 4

1.4 Assumptions 4

1.5 Limitations 5

2. Design Methodology 6

2.1 Language 6

2.2 Tools 6

2.3 Approach for Design 6

2.4 Patterns used 6

3. Detailed Class Diagram in UML 8

3.1 Overview of the design 8

3.2 Detailed UML of each pattern 11

4. Design Discussion 24

4.1 Discussion of individual patterns 24

5. Future scope for more design patterns 41

5.1 FUTURE DESIGN PATTERNS 41

# 1. Project Description

## 1.1 Problem Statement

Schools and Universities are in need of vending machines that encourage healthy eating. No vending machine shows calorie information and suggests healthy items to consumers.

EatSmart vending machine focuses on fulfilling the above need. It also provides a free EatSmart card for users who do not want to carry around cash and want to enjoy other EatSmart member privileges.

## 1.2 Scope of the project

The Eat Smart Vending Machine is used to provide easy access to the food products to the users at schools and Universities. To monitor all the vending machines installed at different locations, a remote administration system is setup. The remote administration system is used to monitor the vending machines and monitoring stations. Monitoring stations are setup where Each Monitoring Station has Vending Machines assigned to it. Administrators are assigned to each monitoring stations to monitor the Vending Machines.

1.2.1 Functionalities

**USER:**

The user of the Vending Machine can either be a member or a non-member.

The basic features that are applicable to both member and non-member are:

* Buying an item.
* Check calorie/ingredient information
* Ask Eat Smart for suggestions
* View Eat Smart’s most popular items

The additional features which are applicable only to members of Eat Smart Vending Machine having Eat Smart Card are:

* **My Favorites:** While buying an item the member can add items to his favorites. The favorite items can then be accessed directly using this module and member can navigate to the buy page directly by selecting one of his favorite item to buy the item.
* **History:** The Member of the Eat Smart Card has a privilege to view the history of the items he has purchased using his card and the corresponding purchase date. The member can navigate to the buy page directly by selecting an item from history to buy the item.
* **Redeem Points:** For every dollar of purchase, one point is credited into their point balance. So the user will be able to redeem the points when it reaches 100 and 100 points corresponds to 5$. So, once the user Redeems 100 points, 5$ will be updated into his Card Balance.
* **Reload Card:** If the member is buying an item using his Card and the purchased item transaction exceeds the available card balance, then the user is prompted to reload his Eat Smart Card. The Member reloads the Card using Cash or Coin.

**OPERATOR:**

The operator is responsible for managing the Cash collected in the Vending Machine.

**ADMIN:**

The Admin monitors the Vending Machine like adding an Item, replacing an item, updating the price of an item in the Vending Machine, adding the nutritional information of an Item etc.

## 1.3 Requirements

Your Choices Inc. is a company which is planning to introduce Eat Smart Vending Machines at various schools and Universities. This project provides a simulation of how a Vending Machine works and the remote administration functionality for managing the vending machines.

## 1.4 Assumptions

The Eat Smart Vending Machine Project has few assumptions for building a Prototype of the simulation like:

* The UI launches the prototype of only one Vending Machine at a time.
* When admin tries to restock or replace an item in the Vending Machine, it is assumed that the operator is notified and he goes and does it manually.
* The Item section of Eat Smart Vending Machine has a fixed number of racks i.e., 3. So if admin wants to introduce an item he can only replace an existing item which is low in sales.
* The Item section of Eat Smart Vending Machine is assumed to have a fixed rack quantity i.e. 10.

## 1.5 Limitations

Although all the solid principles have been implemented to the maximum extent possible the following are the limitations:

* A builder pattern is used build a vending machine. So When a new slot in a vending machine is added like a slot to swipe credit card, the VMBuilder class in the builder pattern has to be opened which violated OCP.

# 2. Design Methodology

## 2.1 Language

Java language is used for the development of the project:

JDK Version: 1.8.0\_31

## 2.2 Tools

The following tools are used in the project.

  IDE : Eclipse Luna SR1 (4.4.1)

UML Tool : Astah Professional

Database : Oracle 12 c

Database IDE : SQL Developer 4.0

## 2.3 Approach for Design

## The project requirements were analyzed and then the additional requirements were defined.

## Then a commonality and variability matrix was defined for the set of requirements.

## By keeping the commonality and variability matrix as reference, different patterns were identified.

## Once all the patterns were identified, the individual patterns were connected to each other to get the entire workflow of the project.

## Based on the design, the database architecture and the tables required were created.

* Then the code was implemented for each of the pattern along with the various functionalities and the code was integrated.

## 2.4 Patterns used

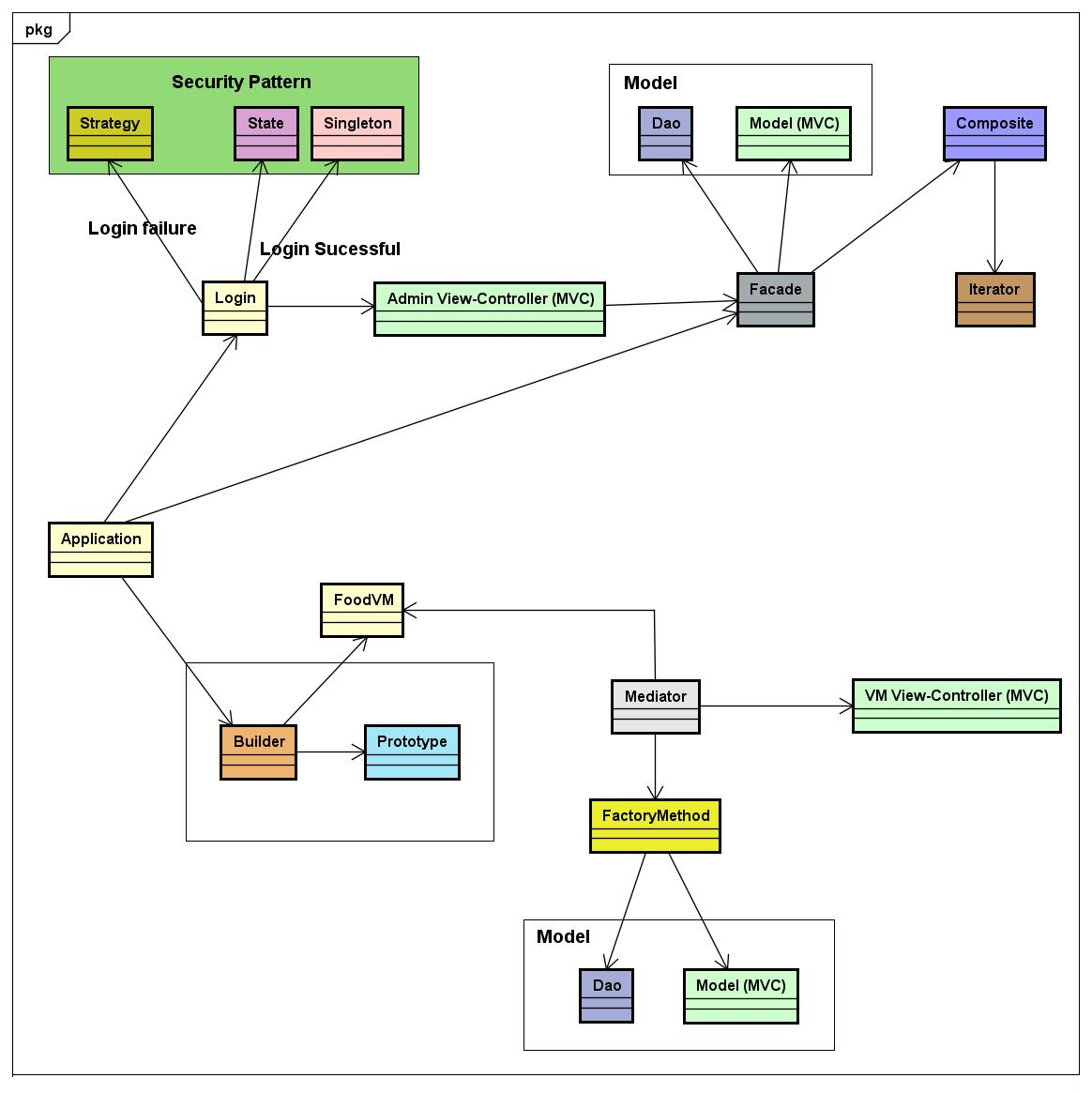
We have used a total of **13 patterns** of which 10 are GOF patterns and 3 are Non GOF patterns:

1. **GOF patterns:**
   1. **Creational Patterns:**
      * 1. Builder
        2. Prototype
        3. FactoryMethod
        4. Singleton
   2. **Structural Patterns:**
      * 1. Composite
        2. Façade
   3. **Behavioral Patterns:**
      * 1. Strategy
        2. State
        3. Mediator
        4. Iterator
2. **Non GOF Patterns:**
   * + 1. Dao
       2. MVC
       3. Security

# 3. Detailed Class Diagram in UML

## 3.1 Overview of the design

This section details about the workflow of the entire design of the project by detailing on how patterns are linked together to provide the required functionality. The overview design diagram is as follows:



The main GUI page of the application is comprised of two tabs: one for VM simulation and the other for the remote administration system. The application class is connected to the builder pattern which builds the VM and the login class which allows an admin to login to the remote system administration.

VM Simulation:

1. The application is launched by invoking the director and the builder classes of the builder pattern with the Vending Machine (VM) name as a parameter that has to be build. All parts of the VM are constant except the item section which can contain different items for each VM.
2. The items are passed to the item section of the builder through a prototype pattern which retrieves all the items from the Dao classes connected to a database, registers the items and passes only the items list that are requested by the particular VM being created.
3. Hence builder and the prototype pattern together builds the VM product which is the food VM.
4. The mediator connects to the FoodVM along with MVC pattern to display the VM to the user when the application is launched. The GUI has a lot of panel pages to display the VM simulation. These various panel pages are connected to a mediator to reduce coupling between the GUI pages and to provide a centralized control for interacting with the other UI pages.
5. Once the GUI is launched, the user is given a choice of using the Vending Machine either as a member or a non-member. Clicking on either member or non-member a FactoryMethod pattern is invoked which instantiates the correct class based on the user click.
6. The factory method is internally connected to the Dao and the model of an MVC pattern to fetch and display the data from the database required for the user.
7. The model of the Dao pattern and the view controller together constitutes the MVC pattern.

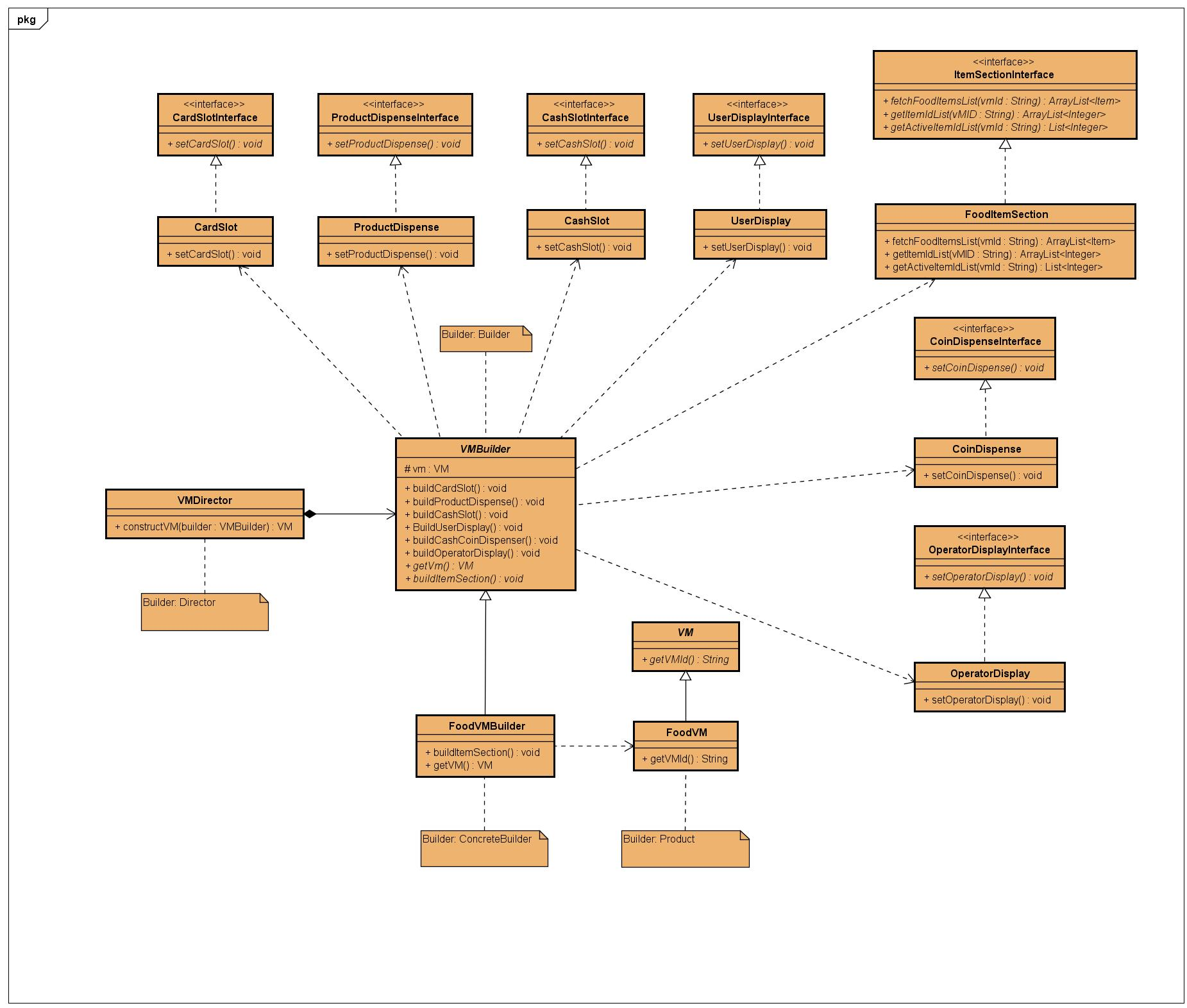
Remote Administration System:

1. When an admin switches to the remote administration tab, a login screen is seen. The security pattern implementation starts here with a single access point i.e. a single pint of access for an admin to login to the remote administration system.
2. The login functionality has two scenarios: a login authentication failure which is implemented by the strategy pattern where multiple strategies could be provided for the login failure scenario along with the check point access of security pattern which performs a user authentication and authorization. Currently, only an error message is displayed indicating the login failure.
3. If the login authentication is successful then a role is created. Currently there is only a single role as admin and based on the role a limited access is provided with the functionalities that an admin is allowed to do which is implemented by the state pattern along with the security pattern for role and limited access and then the admin UI pages are displayed using the MVC pattern.
4. The admin GUI communicates to the Dao and the model classes of the MVC pattern to fetch data from the database and display it and also the composite pattern. Since the admin pages need to connect to all these different classes a façade is implemented that interfaces with all these classes to make the communication easier.
5. The façade is also connected to the composite pattern. The composite pattern has a part whole hierarchy with VM”s depicting as leaves and the monitoring stations depicting the composite containing multiple VM’s. The composite pattern is used to calculate the revenue and the sale quantity across individual VM’s and the monitoring station.
6. An iterator is connected to a composite to iterate through the aggregate object which is the monitoring station.

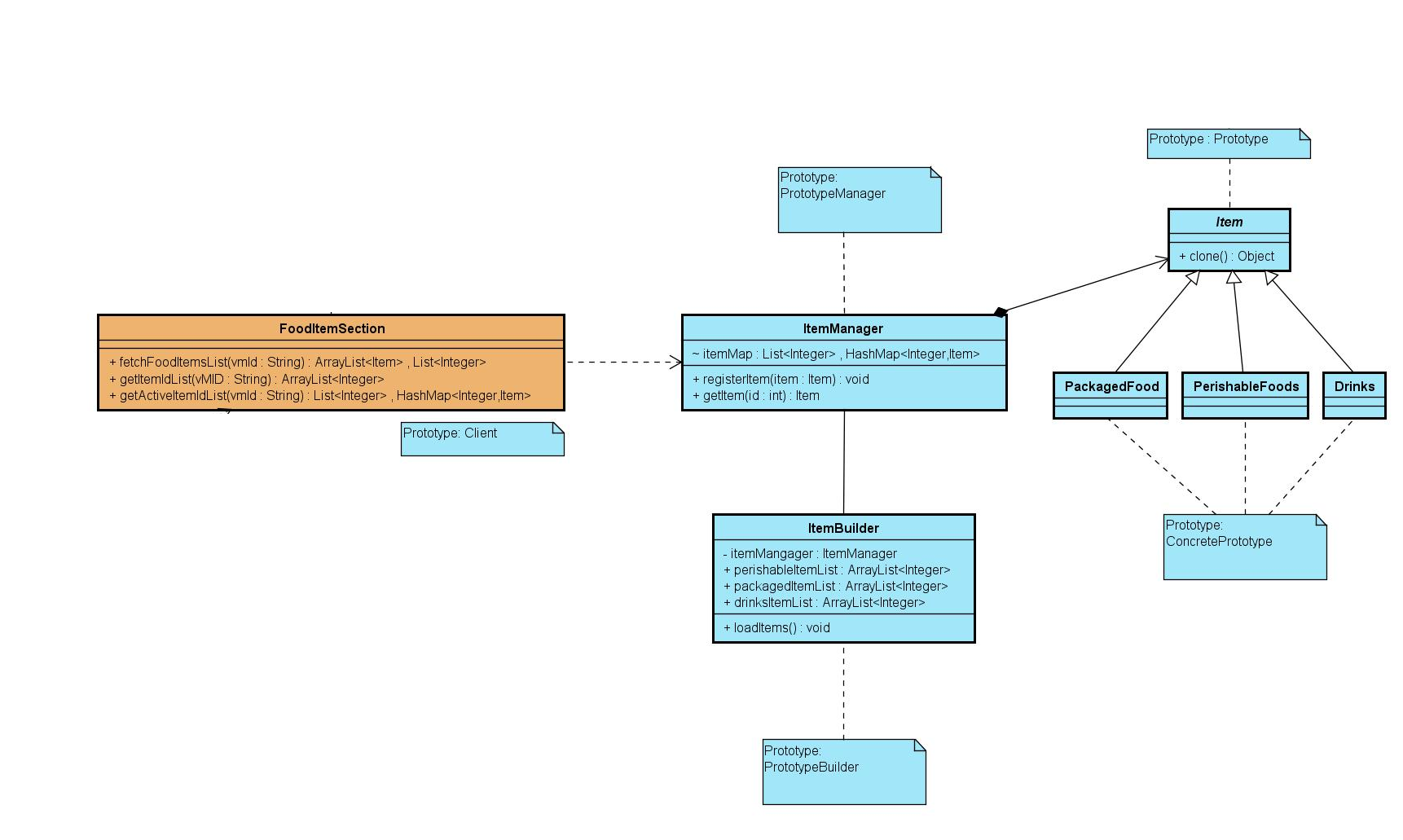
All the 13 patterns are implemented and connected to each other as mentioned above to provide the expected functionality.

## 3.2 Detailed UML of each pattern

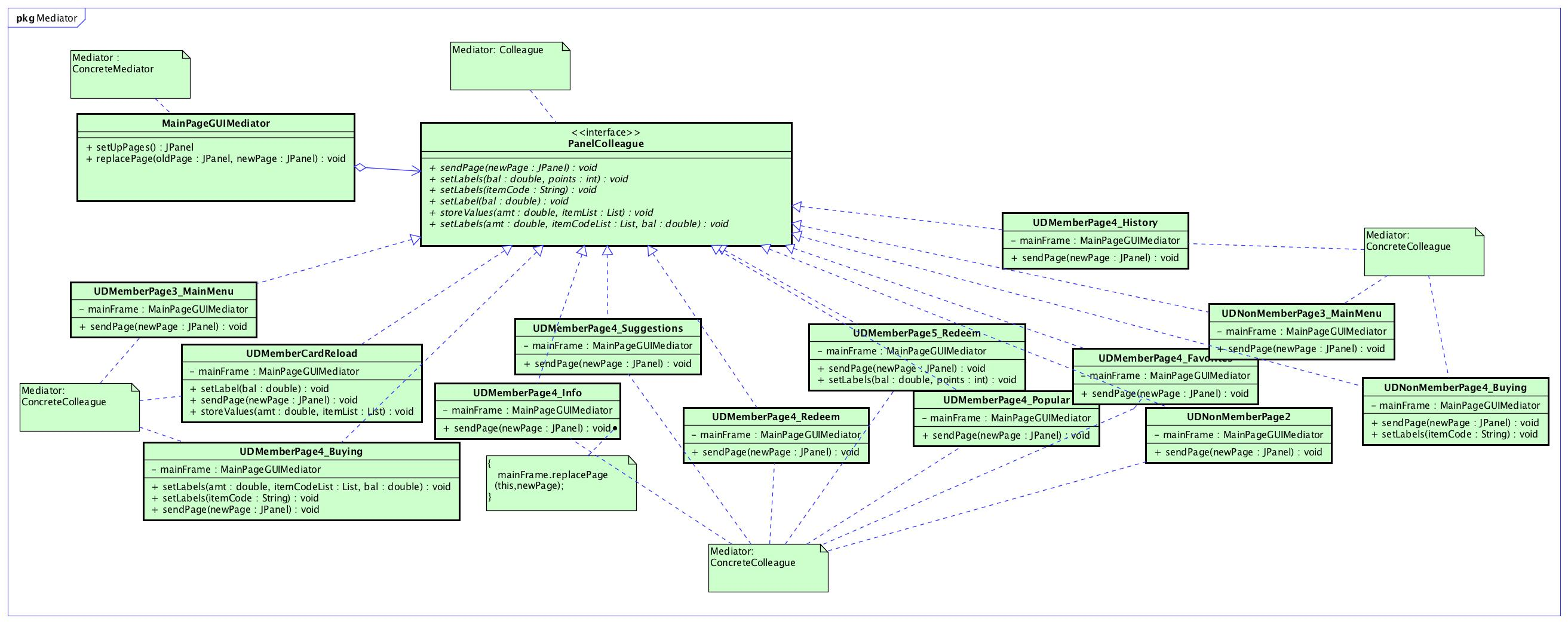
1. **Builder Pattern:**



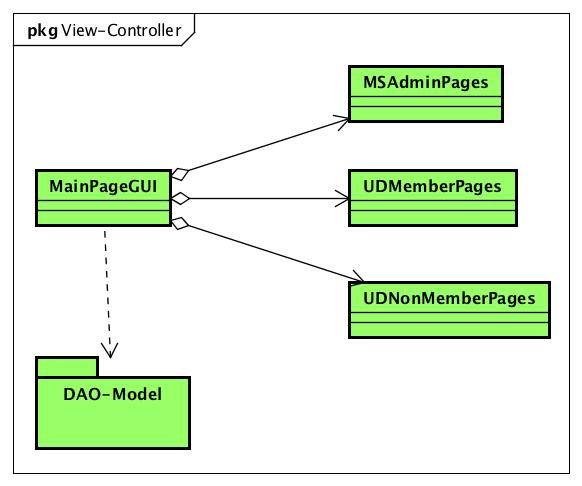
1. **Prototype Pattern:**

****

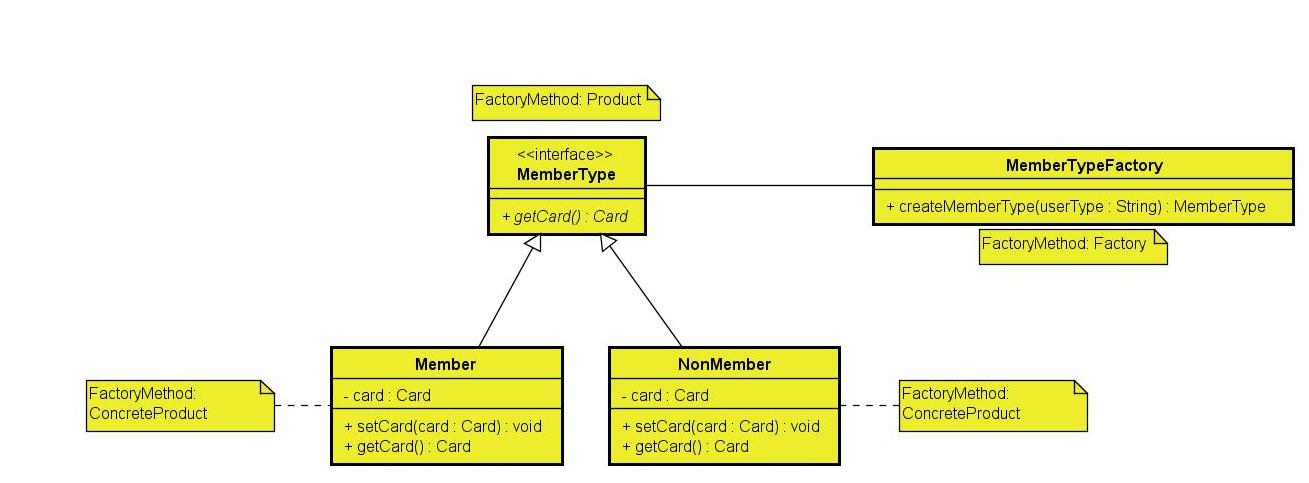
1. **Mediator Pattern:**

****

1. **MVC pattern:**

****

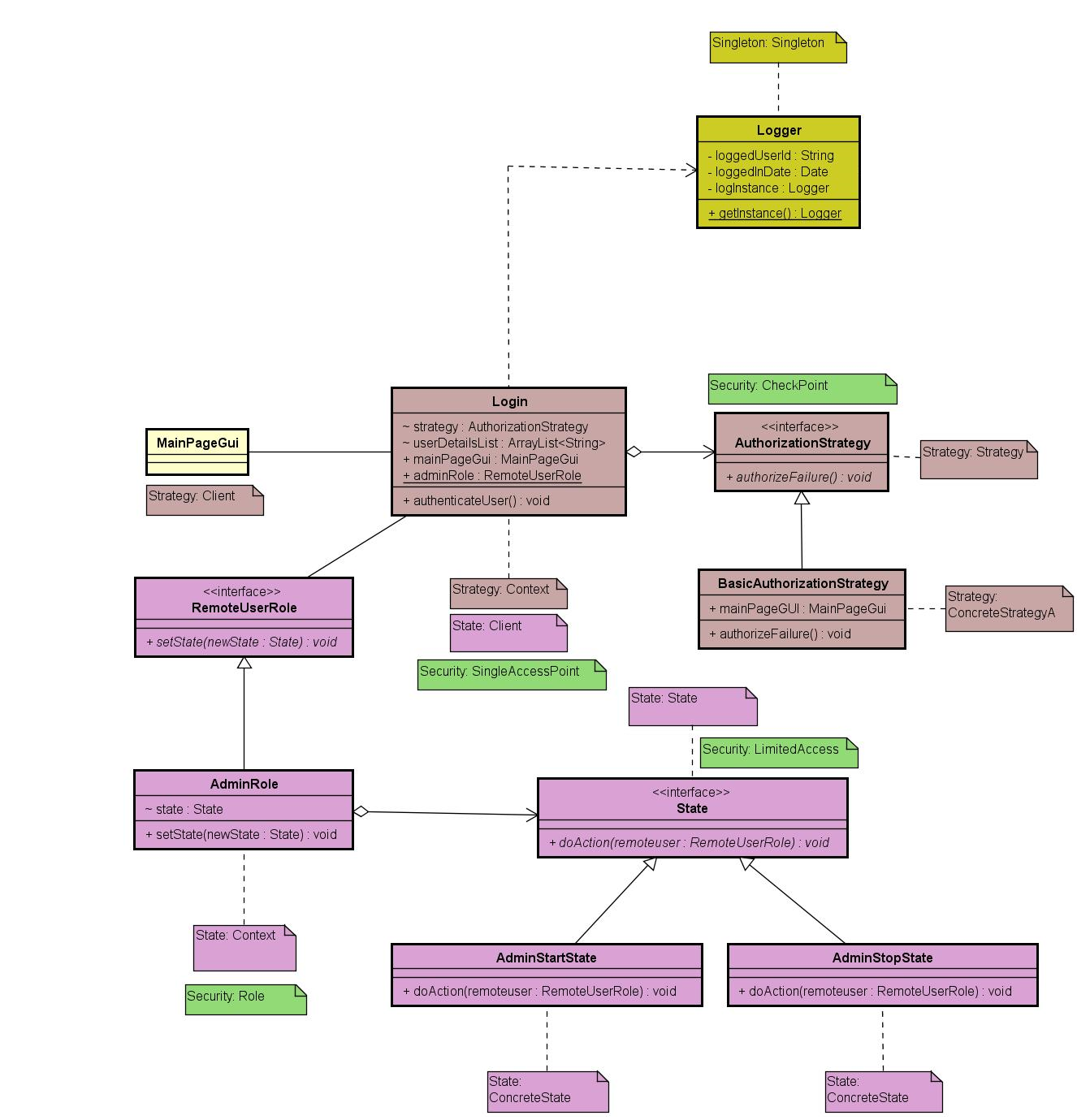
1. **FactoryMethod Pattern:**

****

1. **Dao Pattern:**

****

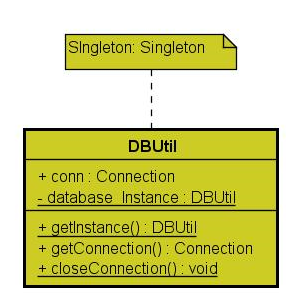
1. **Security Pattern:**



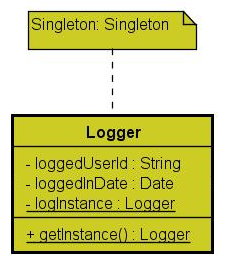
1. **Strategy Pattern:**



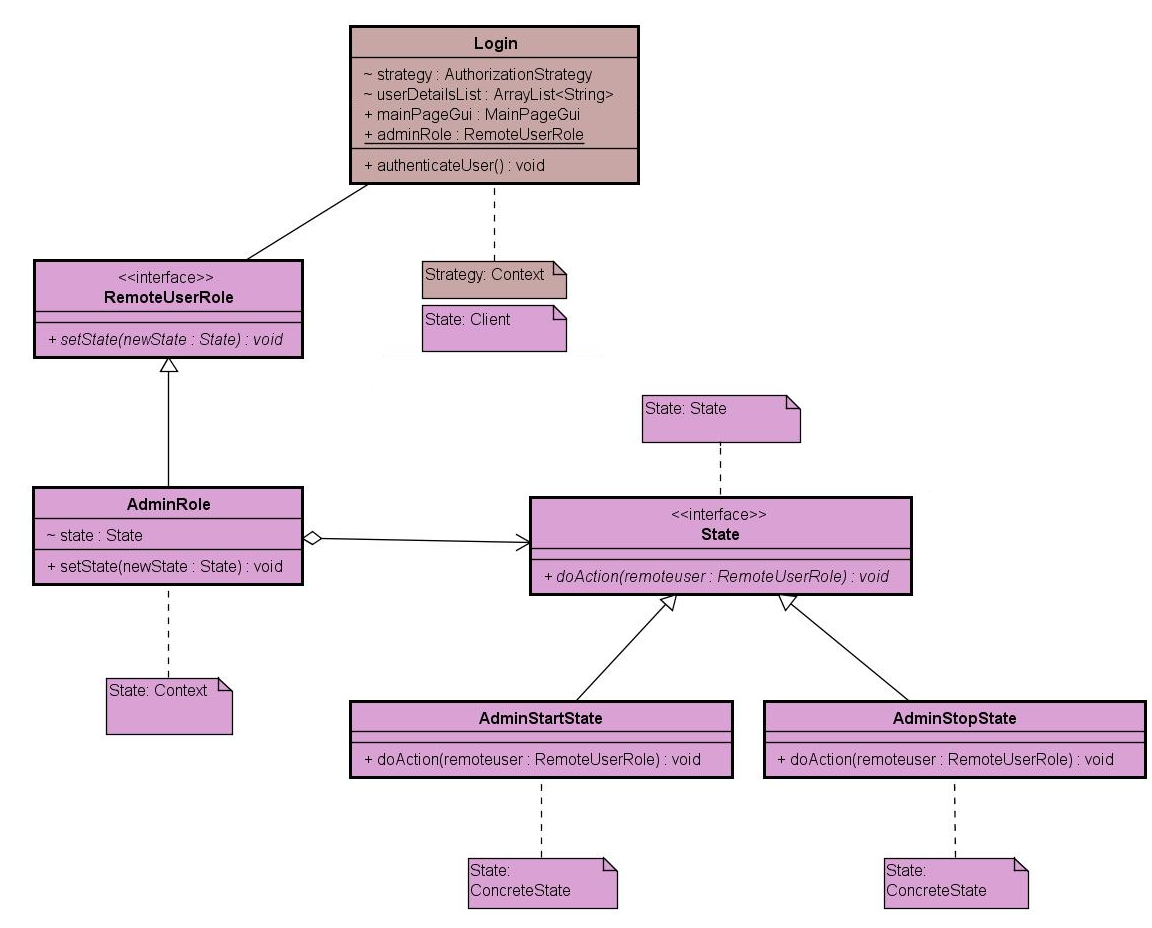
1. **Singleton Pattern:**
2. **DBUtil Class**



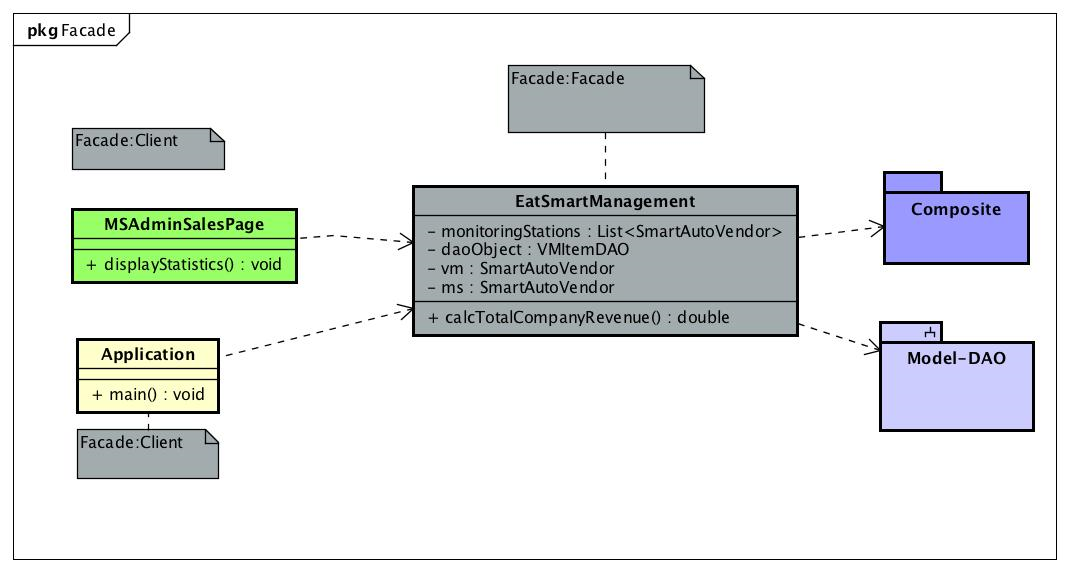
1. **Logger Class:**



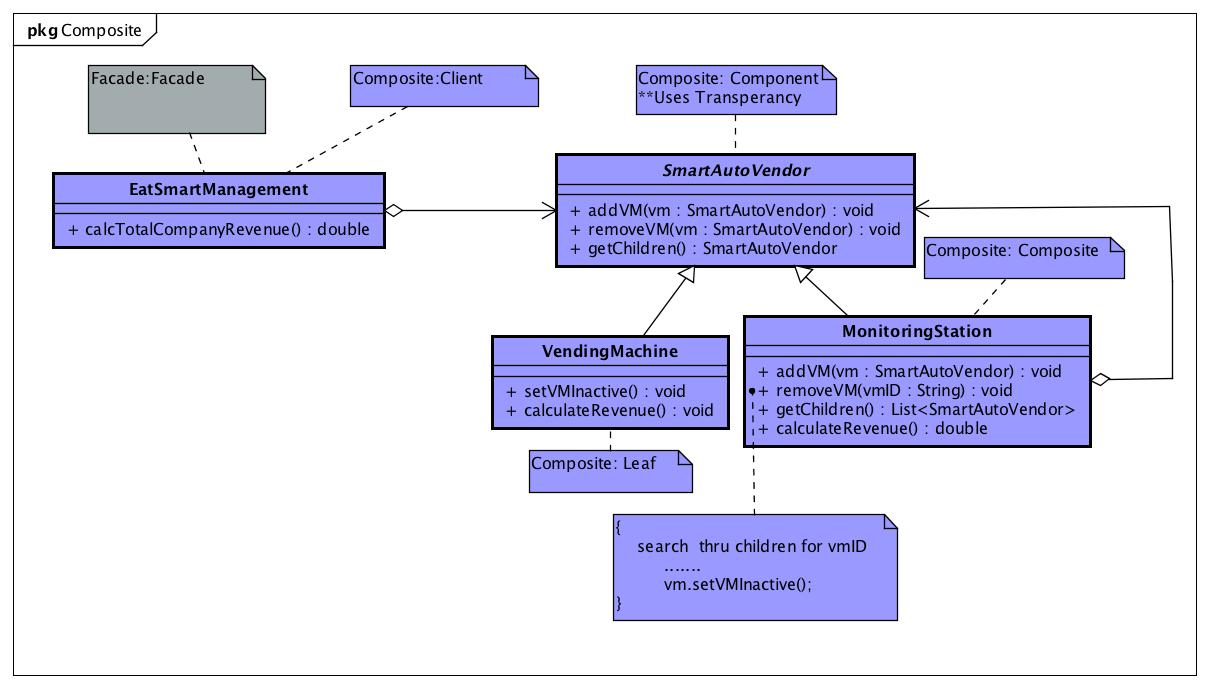
1. **State Pattern:**



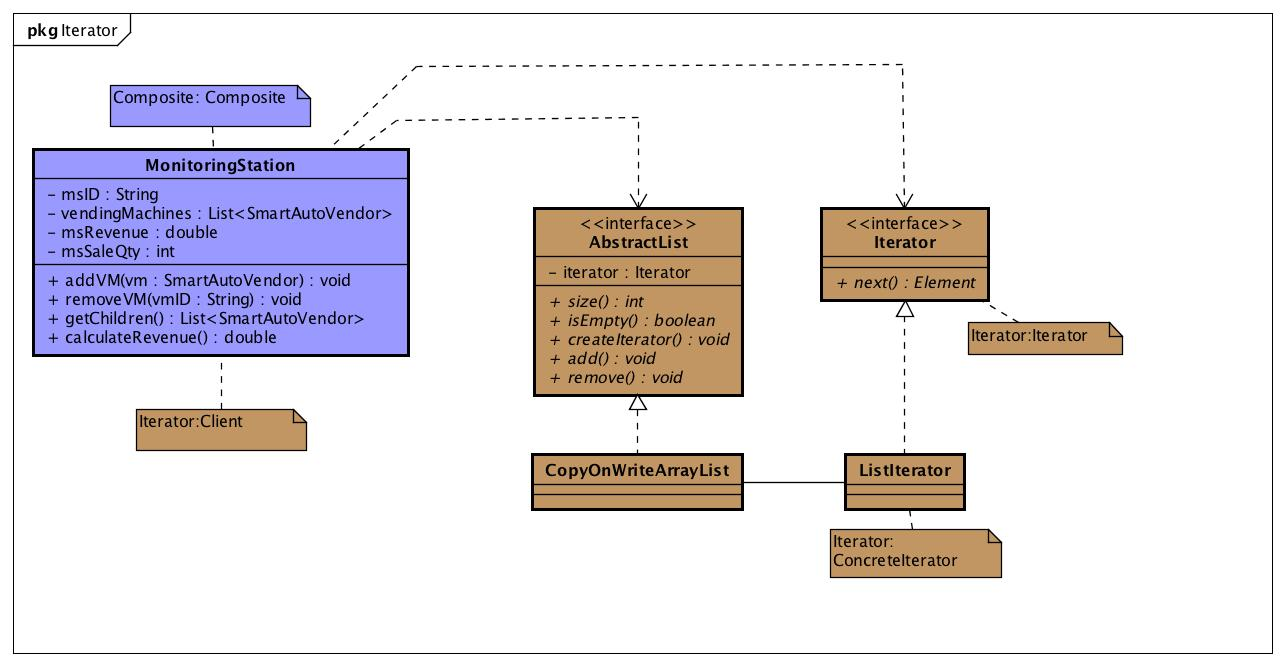
1. **Façade Pattern:**

****

1. **Composite Pattern:**

****

1. **Iterator Pattern:**

****

# 4. Design Discussion

## 4.1 Discussion of individual patterns

This section details on how each of the pattern is implemented along with the relevant code snippets.

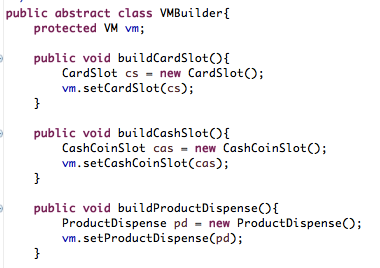
1. **Builder pattern:**

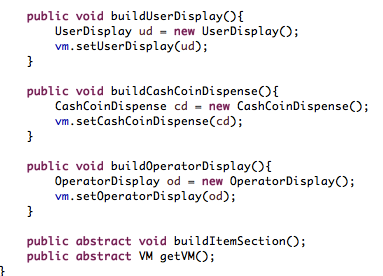
The Builder pattern is used to build the Vending Machine since the Vending Machine (VM) is a complex object. VM is a complex object because of the fact that it contains various parts: Card Slot, Cash Slot, Product Display Slot, Operator Display Slot, Product Dispense slot, Card Dispense slot and Item section. Out of these many slots, only the item section is the variable part i.e. the item section could be different for each VM. All other slots are constant for all the VM’s.

Currently, the builder has only one concrete builder called FoodVMBuilder which is used to build a FoodVm product consisting of only food items. In future, the design could be extended to include more concrete builders like StationaryVMBuilder which is used to build a stationary VM or a MedicineVMBuilder which is used to build a medicine VM for hospitals. The builder pattern was chosen mainly due to the concept of VM being a complex object and keeping extensibility in mind.

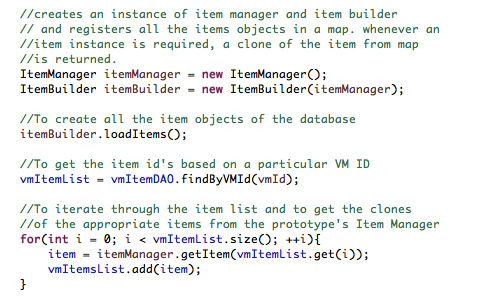
The Builder builds a complex Vending Machine with all its constant and varying parts. The abstract product is a VM and its concrete product is FoodVM.

The Abstract VMBuilder has default method definitions for constant parts and abstract methods for varying parts as follows:

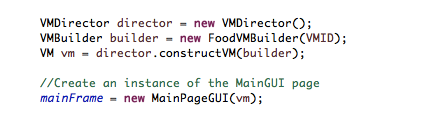




The Varying part, which is mainly the **FoodItemSection** that displays list of active items in a specific Vending machine is fetched from the prototype’s ItemManager. The FoodItemSection class acts as the client to the prototype by invoking the item builder and manager. The following code snippet shows the code from the FoodItemSection class:



And the Application class takes care of invoking the Builder’s Director to launch the VM by giving the VM ID as the input.

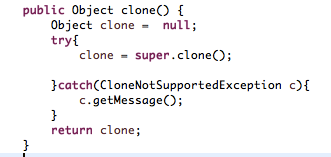


1. **Prototype Pattern:**

Each item section of the Vending Machine can have same or distinct items in it. The items can be fetched from Prototype pattern. Considering Item to be a complex object as it has many parameters involved with it like nutritional information, price etc. Prototype fetches the related parameters from the database using DAO Pattern and registers the item.

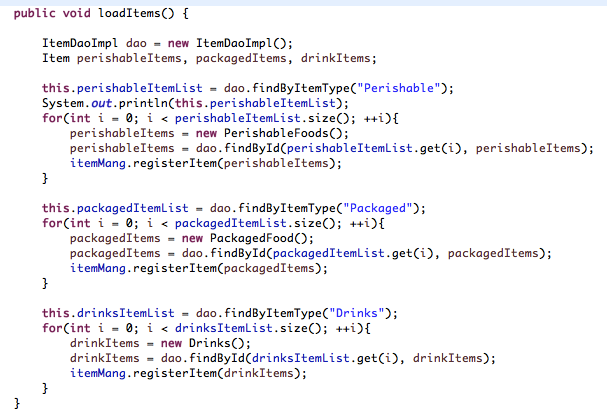
The Prototype Pattern was used keeping in mind the future extensibility. Initially we had thought of considering Classes like Burger, Sandwich, Chips etc. But, we realized that it wouldn’t be a possible to add a new item in the Vending Machine, other than the Burger or Sandwich, we will not be able to create a new Class dynamically. In order to achieve this, the Items have been classified as Packaged, Perishable and Drinks. So whenever a user tries to add a new item now, it would be under any of these three categories.

The main prototype Item, which is also the Model class corresponding to ItemDAO class, has the clone method implemented in it and all its subclasses (Packaged, perishable and Drinks classes) as follows:

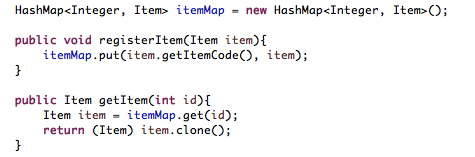


The Item Builder has a reference to Item Manager and it registers all the items from the database and any other newly introduced items into the Item Manager’s HashMap.

The following method shows how the prototype is invoked and registered:



Item manager manages the clones in a HashMap and, when asked by the FoodItemSection of Builder pattern, it returns the requested clones of items as a list. This is shows in the below code snippet from ItemManager class



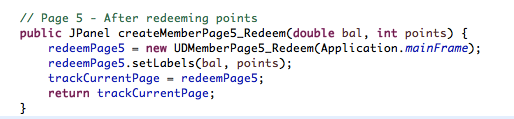
1. **Mediator Pattern:**

The MVC pattern consisted of several view-controller classes that needed to interact with each other, pass information and communicate. This gives a very good ground for introducing a mediator among the View-Controller classes. The main view page that will be first loaded from Application class is the first class to connect to the first of several colleagues, and makes the best case of a mediator. So the design is as follows:

* A bunch of view panels, all implement a common interface called PanelColleague and inherit JPanel
* Mediator class MainPageGUI communicates to all the concrete colleague classes to pass on any update that one colleague wants to send to the other.

The MainPageGUI class acts as the mediator and communicates with colleagues for them. This mediator does not communicate the changes requested by a colleague to all other colleagues, but it only updates one specific other colleague (say colleague2 and this request comes from colleague1).

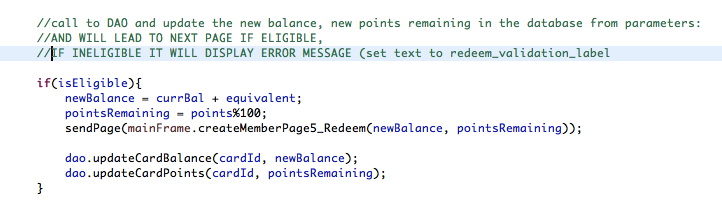
The following sample snippet from MainPageGUI class shows the mediator specific method that is used by a colleague to pass information to the mediator and replace itself with its successor colleague.





So these two methods above will be used in MemberPage4\_Redeem, which is a predecessor page of Page5\_Redeem as follows:

It basically sends information to mediator and asks mediator to replace it with the next page because the Redeem Button has been clicked.



And sendPage method from above is defined as follows:



1. **MVC pattern:**

The design for EatSmart follows MVC pattern throughout the structure. The View-Controller classes both display the view and act as controllers. This view-controller subsystem is connected to Model subsystem. The model subsystem has one model class each linked to a DAO class.

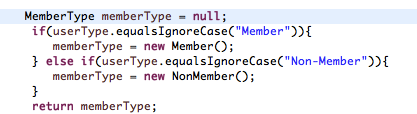
The Model classes fetch the information needed from DAO classes and pass them on to the V-C classes.

The MVC model is implemented for both the Vending Machine Prototype flow and the Remote Admin System flow.

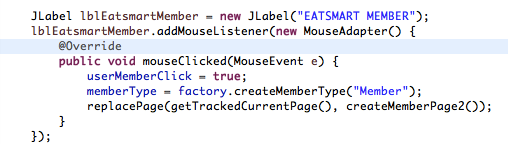
1. **FactoryMethod Pattern:**

In the Vending Machine, currently the user can either be a member or a non-member. It can be extended by introducing Privileged User (a person who has been an Eat Smart Card member since 5 yrs with additional privileges) in future. Whenever the user selects a Member or a non-Member, the corresponding instance should be created. So we have used Factory method in order to create an instance of member or a non-Member.

MemberTypeFactory instantiates the required member or non member class as follows:



User click on Member will trigger the memberTypeFactory as follows: (Non member is triggered the same way)



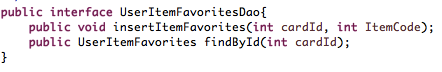
1. **Dao Pattern:**

The DAO (also called as Data Access Object) Pattern is used to provide an abstract interface to a database or any other persistence mechanism. currently, the application uses Oracle, but in future other persistence mechanisms could be implemented using Files or other databases. It maps the persistence layer to the application and provides us with data specific operations or CRUD(Create, Read, update and Delete) operations without exposing the details of the Database.

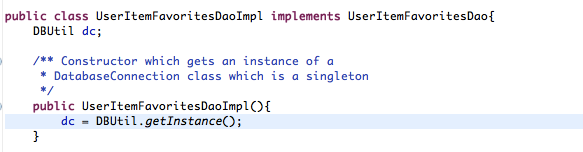
|  |  |  |  |
| --- | --- | --- | --- |
| S. No | Table Name | Classes | Features |
| 1 | Item | Item, ItemDao,  ItemDaoImpl | This table represents the Item and it’s corresponding parameters like nutritional info like calorie count, gluten free and price etc. of the Item Section in the Vending Machine. |
| 2 | userrole | UserRole, UserRoleDao,  UserRoleDaoImpl | This table is used to implement the roles for the user. |
| 3 | MonitoringStation | MonitoringStation, MonitoringStationDao, Monitor,ingStationDaoImpl |  |
| 4 | loguserinfo | Logger, LoggerDao, LoggerDaoImpl | This table is used to log the logged in user’s information. |
| 5 | VMOperator | Operator, VMOperatorDAO, VMOperatorDAOImpl | This table stores the Operator with respect to the Vending Machine. |
| 6 | VMItem | VMItem, VMItemDAO, VMItemDaoImpl | This table stores the list of the items and corresponding information with respect to each Vending Machine. |
| 7 | CARD | Card, CardDao, CardDaoImpl | This table stores the Car information like Card Id, Points, PurchaseDate, Balance. |
| 8 | CARD\_ITEM\_HISTORY | CardItemHistory, CardItemHistoryDao, CardItemHistoryDaoImpl | This table shows the history of the card i.e., the iems brought by that particular card and date of purchase of the item. |
| 9 | CARD\_ITEM\_FAVORITES | CardItemFavorites, CardItemFavoritesDao, CardItemFavoritesDaoImpl | This table is used to store the items, the member had added to his favourite list while buying. |
| 10 | Coin | Cash\_Coin CashDao, CashDaoImpl | This table stores the amount collected for a particular Vending Machine on a particular date. |

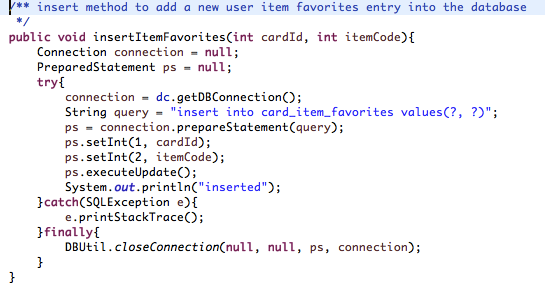
UserItemFavouritesDao interface corresponds to the CardFavorites table, where the member’s “Add to my favorites” feature inserts the item code as a favorite against the card Id.

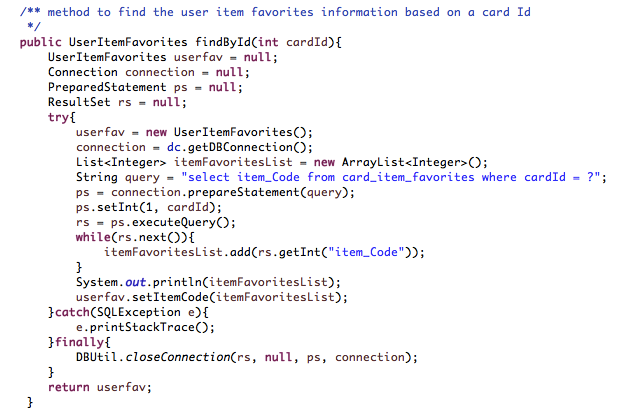
From the above list of DAO classes, the following code shows the implementation of UserItemFavouritesDao. The rest of the DAO classes are implemented the same way.



The concrete DAO implements the abstract methods from its interface as follows:







1. **Security Pattern:**

The following security patterns are used in the admin context for logging into the remote administration system.

1. **Single Access Point:** The single access point is implemented in the remote administration system login. There is only one path allowed to the admin users to login to the system and a successful login logs the user information which is implemented with a singleton.
2. **Check Point Access:** The information from the single access point should pass through the check point access which provides a way to authenticate admin users. A strategy pattern is combined along with the check point access where multiple failure strategies could be implemented when a user attempts to break through the system.
3. **Role:** Once the login is successful, a role is created based on the login credentials of the user. In our project, we have only a single role currently created for the remote administration system which is the role of an admin. This implementation can be further extended to create new roles.
4. **Limited Access:** Based on the role of the logged in user, a limited access is provided to the user with the functionalities that the user is allowed to do. This implementation of providing limited access based on the role is combined with a state pattern.

1. Login failure Scenario

This is where Strategy Pattern has been implemented by providing different strategies.

2. Login Successful Scenario:

The logger’s information is captured by the Logger class and stored in the database. This Logger is a Singleton is used.

3. The State Pattern is then invoked and provides a limited view of the Admin.

Code Snippet for the above two scenarios is shown below



1. **Strategy Pattern:**

The strategy pattern is implemented in an admin context. Whenever an admin tries to login into the remote administration system, the admin credentials are authenticated. When the authentication fails, an error log message is displayed. The strategy pattern is used to implement the various strategies for implementing the failed authentication & authorization process. Currently, only a single failure strategy is been implemented which is called the BasicAuthorizationStrategy which displays just an error message to the admin. In future, other failure strategies like locking the admin account after certain incorrect attempts or allowing the admin to define a new password after certain incorrect attempts could be implemented. The strategy pattern was chosen in this context because of the requirement to implement various strategies and keeping extensibility in mind.

1. **Singleton Pattern:**

The singleton pattern has been used for two classes in the project :

1. **DBUtil class:** This class is used for connecting the application to the database. A singleton pattern is used because we need only a single instance of a database object so that the data is secure and improves performance cost.
2. **Logger class:** This class is implemented for the admin scenario. Whenever an admin tries to login into the remote administration system, the admin credentials are authenticated. If the authentication goes through then the user details and login time is logged into the database using the logger class which is a singleton.
3. **State Pattern:**

The State pattern is used in an admin context. Once the admin login session is successful, a new session for the admin is created by providing a limited access to the functionalities that admin is allowed to do.

This limited access of functionalities is provided to the admin when the admin login is successful and ends when the admin clicks on the log out button. This way a session is started for an admin user and ended. This implementation of the state pattern is not a full-fledged state pattern in the sense that a start state for an admin does not automatically transition to the next state (stop state) based on an operation rather the states are transitioned based on the user inputs.

1. **Façade Pattern:**

When an admin logs in successfully, statistics that populate on the admin UI pages are eclectically fetched from the different classes that belong to different subsystems.

Several admin related classes should communicate with subsystems like DAO-Model and Composite to fulfill the limited view requirements of Admin View pages. All this must be done in the Application class as soon as the remote admin system is launched and the login is successful. Given the above-said scenario, it is highly complicated for several clients to talk to several objects. So, A Façade here simplifies the complexity by acting as a single communication point for clients by collating and displaying results from several subsystems.

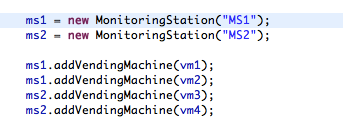
In our design this Façade class is named “EatSmartManagement” as this is the class that collates administration level statistics to display on to the admin pages

There are at least two clients that contact the Façade class – EatSmartManagement

This class basically contacts the Component of the composite pattern and the DAOModel subsystem objects to fetch and collate information.

It acts as a client for composite pattern and adds vending machines to monitoring stations

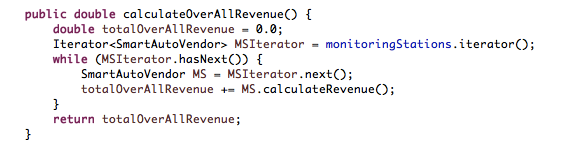
Some sample snippet of how it adds VMs and maintains a list of Monitoring stations to calculate the overall revenue of the company





It then uses the DAO class of DAO pattern to fetch the revenues VM wise and set them amongst other things.





1. **Composite Pattern:**

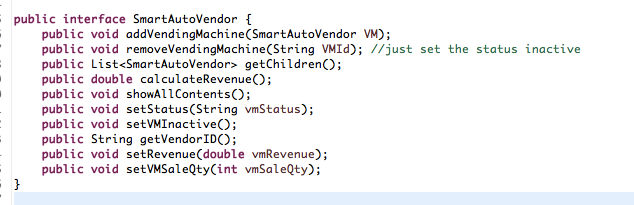
When a specific admin logs in, the view pages need the information about the total revenue and total sales statistics for all vending machines and monitoring stations under the admin's scope.

This requirement has a part-whole analogy, part being the statistics of a single VM and  whole being the cumulative statistics of all the VMs under a monitoring station.

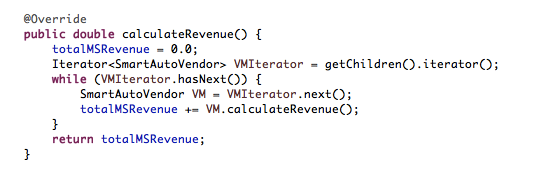
The composite pattern simplifies this process with  vending machine as a leaf and monitoring station as a composite.

The SmartAutoVendor is the Component class, which is implemented by VendingMachine (Leaf) and MonitoringStation (Composite) and uses Transperancy

The composite class iterates through the list of children and performs operations like calculating revenue, setting a VM inactive by ID etc. The following code snippet will describe one of the operations the composite performs:



To calculate revenue of the entire Monitoring station



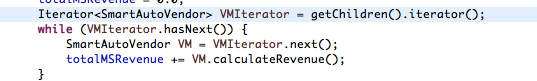
To remove a VM from an MS i.e, to make the VM inactive



1. **Iterator Pattern:**

The iterator is used to provide a way to access the elements of an aggregate object sequentially without exposing the underlying representation. This pattern is used along with Composite pattern having Monitoring station (Composite) and Vending Machine (Leaf Node) to traverse through the Composite i.e., Monitoring Station.

Considering MonitoringStation as a Composite, iterator is used to traverse through the list of vending machines



# 5. Future scope for more design patterns

## 5.1 FUTURE DESIGN PATTERNS

These are the following patterns that can be implemented along with the existing design:

1. Memento:

In the application, when a user tries to buy items, and if the purchase transaction exceeds the user’s Card Balance, then the user is prompted to reload his card. When the user reloads and navigates back to the buying page, the data of the user like the items he had purchased, the amount due etc. still has to persist. A Memento Pattern could be implemented to save the state of the object for retrieval at a later time.

1. Observer

The observer pattern could be implemented in a way that the model classes of the Dao pattern are depicted as the subjects and whenever a model changes a notification is sent to all the observers which are the View controller classes. The view controller classes receives the notification and updates the view as expected.

1. Bridge

Currently the various payment modes: Eat Smart Card and Cash or Coin are connected directly to both member and non-member. In future, a bridge pattern could be introduced which separates the abstraction (member and non-member) and the various implementations of payment modes (EatSmart card and cash or coin) in which the two can vary independently like a new user type called a special privileged member and a new payment mode like MasterCard, VISA could be added.

1. Decorator

Currently, the application is single threaded. In future, if Admin OR Manager triES to login and perform the operations simultaneously, then the changes done by the Admin does are directly visible to the Manager and vice-versa.