UML Diagrams:

a) Use Case Diagram:

A use case diagram represents the interactions between the system (Grocery Sales App) and the actors (Customers). Based on the given requirements, the following use case diagram can be created:

Actors:

Cashier/User

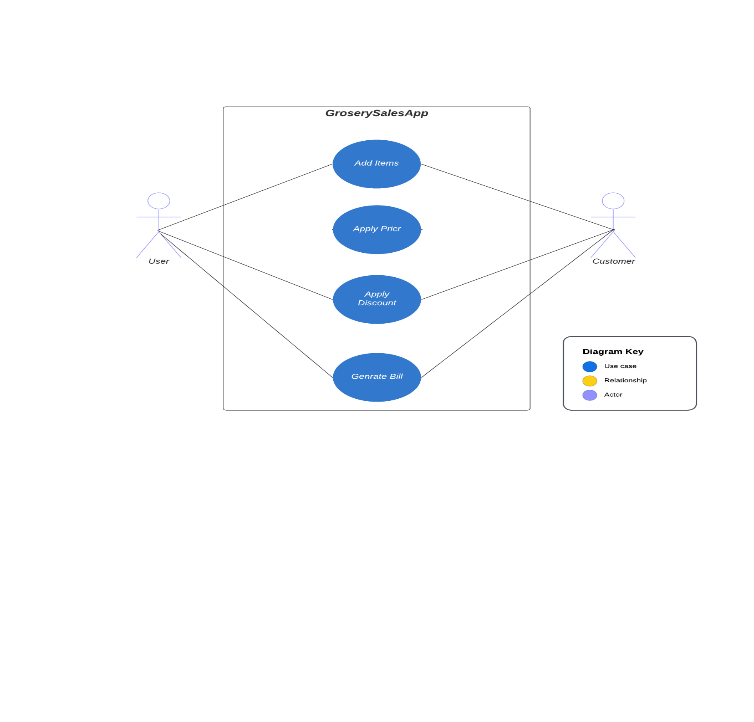
Customer

Use Cases:

Add Item to Bill: This is initiated by the User. The user enters the item name and price, and adds it to the bill.

Apply Discount: This use case is triggered when the customer presents a royalty card. The user checks the discount checkbox and enters the card number.

Generate Bill: The User calculates the total bill amount, the discount amount (if any), and the net payable amount. The results are shown in the bill area.



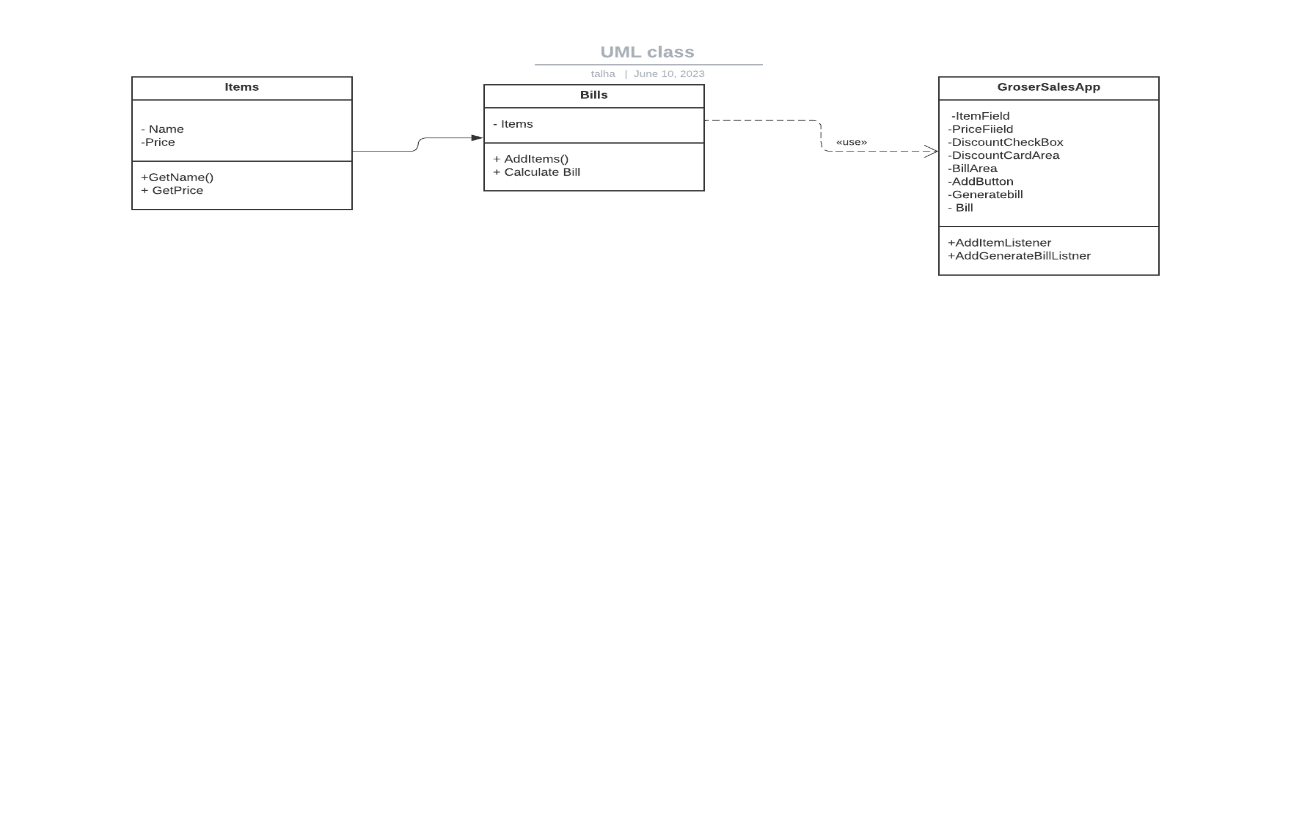
Class Diagram:

A class diagram represents the classes and their relationships. Based on the code and requirements, the following class diagram can be created: Classes:

Item: This class represents an item, with properties name and price, and their getters.

Bill: This class represents the bill, with an attribute items (a list of Item objects), a method addItem(name, price) to add a new item to the bill, and a method calculateTotalAmount() to calculate the total bill amount.

GrocerySalesApp: This class represents the GUI of the application. It contains attributes like itemField, priceField, discountCheckBox, discountCardField, billArea, addButton, generateBillButton, bill (an instance of Bill). It also contains inner classes AddItemListener and GenerateBillListener which handle events for adding items and generating the bill respectively.



c) Sequence Diagram:

A sequence diagram represents the interactions between objects over time. Based on the code and requirements, the following sequence diagram can be created:

User -> GrocerySalesApp: Enter item name and price

User -> GrocerySalesApp: Clicks "Add Item"

GrocerySalesApp -> Bill: addItem(item, price)

User -> GrocerySalesApp: Checks discount checkbox (optional)

User -> GrocerySalesApp: Enter discount card number (optional)

User -> GrocerySalesApp: Clicks "Generate Bill"

GrocerySalesApp -> Bill: calculateTotalAmount()

A picture containing text, diagram, screenshot, parallel

Description automatically generated

Design Pattern:

The Observer design pattern can be used depending on the demands of the mini-grocery sales application.

Justification: The Observer pattern is appropriate for this application because it enables numerous objects to watch for changes in another object's state and get notifications of those changes. When an item is added to the "Bill" in this scenario, the "GrocerySalesApp" must update the GUI. The "GrocerySalesApp" can monitor the "Bill" object and get alerts when new items are added by using the Observer pattern.The advantages of applying the Observer pattern are as follows:

The Observer pattern encourages loose connectivity between the topic being observed and the observers. It's not necessary for the "GrocerySalesApp" to be intimately familiar with the "Bill" implementation specifics. It merely needs to understand how to modify the GUI in response to changes in observable state.

The Observer design is extensible because it makes it simple to include new observers without changing the topic. Future features and components can easily implement the observer interface and register themselves as observers if they need to keep track of the "Bill" object in the future.

Maintainability: By separating the concerns of the subject and the observers, the Observer pattern makes the code simpler to comprehend, maintain, and modify.

Changes to the "Bill" object or the GUI can be made independently without affecting each other.