Detailed Design Document

ArtisanHub

Group Members:

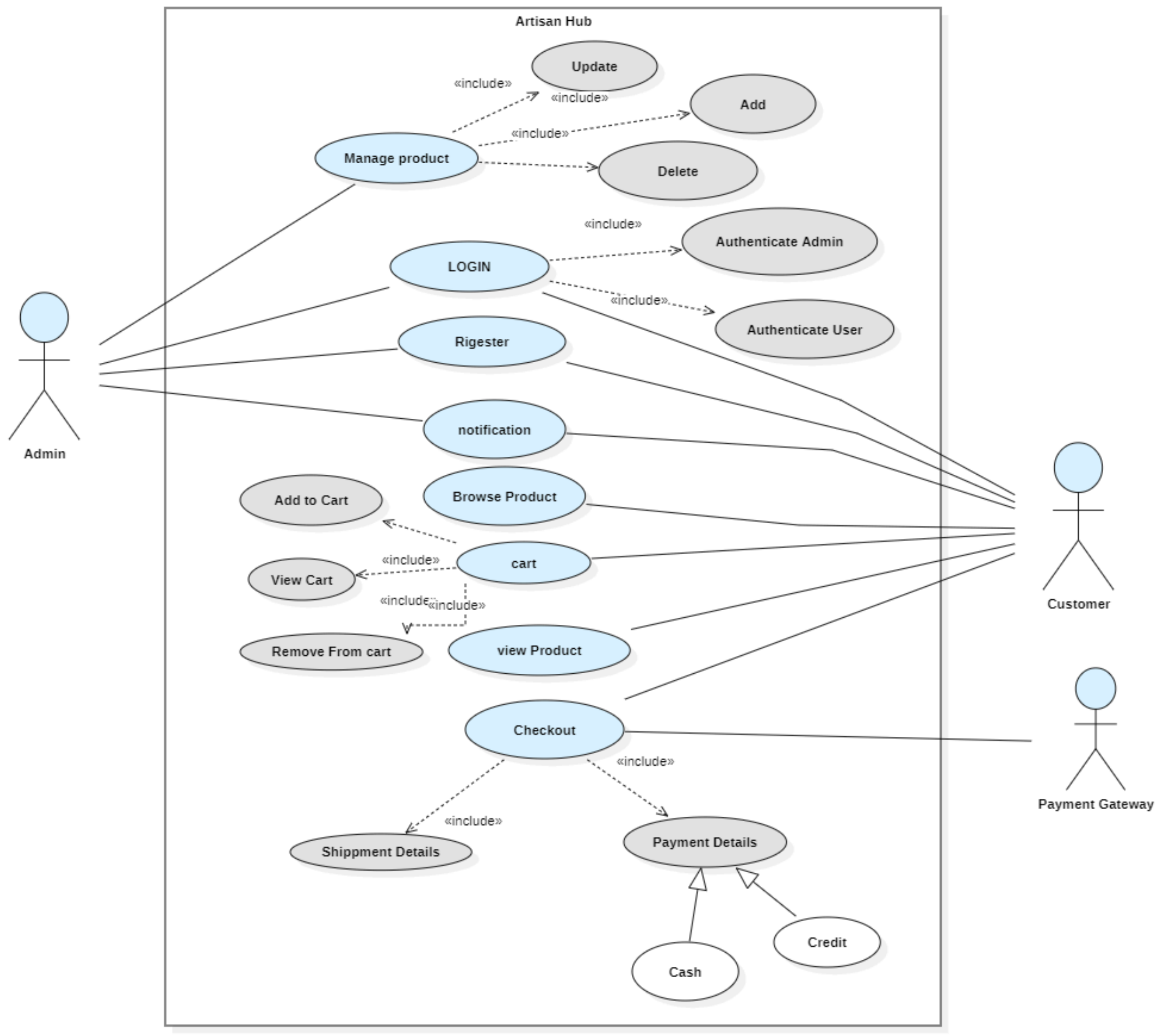
Namra Imtiaz SE-22060

Urwa Qadir SE-22061

Fatima Hassan SE-22098

Jassica Allen SE-22099

USECASE DIAGRAM:



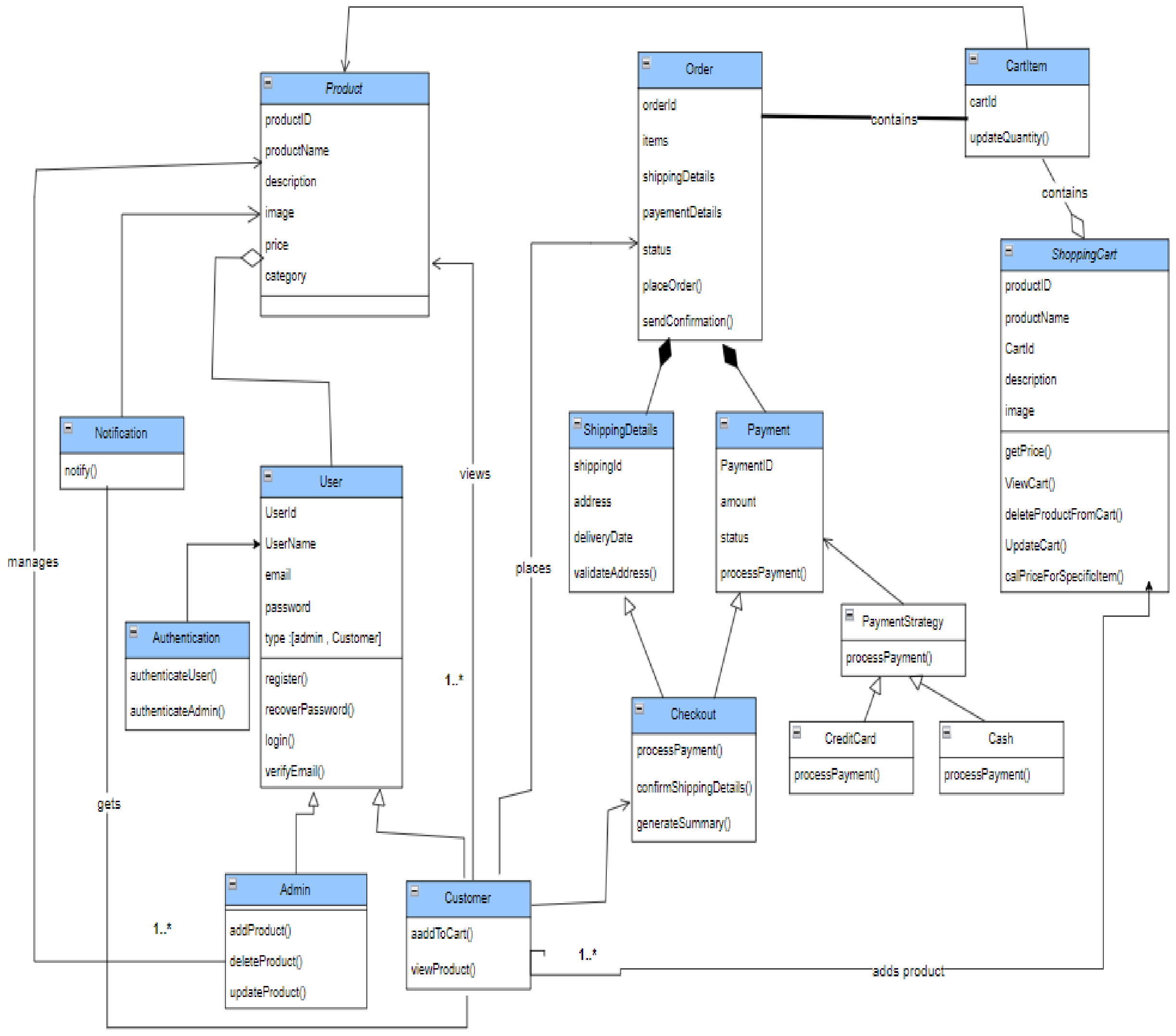
Use Case Diagram for the Artisan Hub platform, illustrating the interactions between users (Admin, Customer, and Payment Gateway) and the system's functionalities. It represents how various actors engage with the system to complete different tasks, with emphasis on the management of products, user authentication, and order processing.

Actors:

1. **Admin:**
   a. Responsible for managing the product inventory on the platform.
   b. Can add new products, update existing products, and delete products.
   c. Must authenticate themselves to perform these actions.

2. **Customer:**
    a. Engages with the system to explore available products and manage their cart.
    b. Customers can add products to the cart, remove them if necessary, and proceed to checkout.
    c. They must register or log in to the System to access the order features.
3. **Payment Gateway:**
    a. An external system that handles the payment process.

CLASS DIAGRAM:



Class Diagram for the Artisan Hub platform, illustrating the structure and relationships of the system's key classes, attributes, and methods. It focuses on how different components, such as users, products, orders, and payments, interact with each other. This diagram provides an overall blueprint for how the system operates and handles its main functionalities.

1. **User:**
   - *Attributes:*
     - UserId, UserName, email, password, and type (admin or customer).
   - *Methods:*
     - register(): Registers a new user.

        - recoverPassword(): Allows users to recover their password.
        - login(): Authenticates users logging into the platform.
        - verifyEmail(): Verifies the email address during registration or recovery.
    - *Relationships:*
        - Can either be an Admin or a Customer.

2. **Admin (inherits from User):**
    - *Methods:*
        - addProduct(), deleteProduct(), updateProduct(): Admins manage product inventory, enabling CRUD operations.

3. **Customer (inherits from User):**
    - *Methods:*
        - addToCart(): Adds products to the shopping cart.
        - viewProduct(): Views available products in the catalog.

- Product:
    - *Attributes:*
        - productID, productName, description, image, price, category.
    - *Relationships:*
        - Linked to the CartItem and ShoppingCart, representing products being added to the shopping cart.
    - Products can be viewed and added to carts by customers.

4. **ShoppingCart:**
    - *Attributes:*
        - productID, productName, cartId, description, image.
    - *Methods:*
        - getPrice(), ViewCart(), deleteProductFromCart(), UpdateCart(), calPriceForSpecificItem(): Handles cart functionalities like adding, removing, and viewing products.
    - *Relationship:*
        - The ShoppingCart contains CartItems and interacts with the Order and Checkout process.
5. **CartItem:**
    - *Attributes:*
        - cartId: Represents individual items within the shopping cart.
    - *Methods:*
        - updateQuantity(): Updates the quantity of products in the cart.
    - *Relationships:*
        - A ShoppingCart contains multiple CartItems.

6. **Order:**
    - *Attributes:*
        - orderId, items, shippingDetails, paymentDetails, status.

- *Methods:*
  - placeOrder(), sendConfirmation(): Handles placing orders and sending confirmations to customers.
- *Relationships:*
  - Links to the ShippingDetails and Payment for processing the order.

7. **ShippingDetails:**
   - *Attributes:*
     - shippingId, address, deliveryDate.
   - *Methods:*
     - validateAddress(): Ensures the address provided is valid for delivery.
   - *Relationships*:
     - Linked to the Order class, providing essential shipment information for deliveries.

8. **Payment:**
   - *Attributes:*
     - paymentId, amount, status.
   - *Methods:*
     - processPayment(): Handles payment processing.
   - *Relationships:*
     - Connected to the Order and the PaymentStrategy pattern, which supports various payment types.

9. **PaymentStrategy:**
   - *Methods:*
     - processPayment(): Implements the strategy design pattern for payment processing.
   - *Subclasses:*
     - CreditCard: A specific payment method that processes payment via credit card.
     - Cash: A specific payment method for processing cash payments.

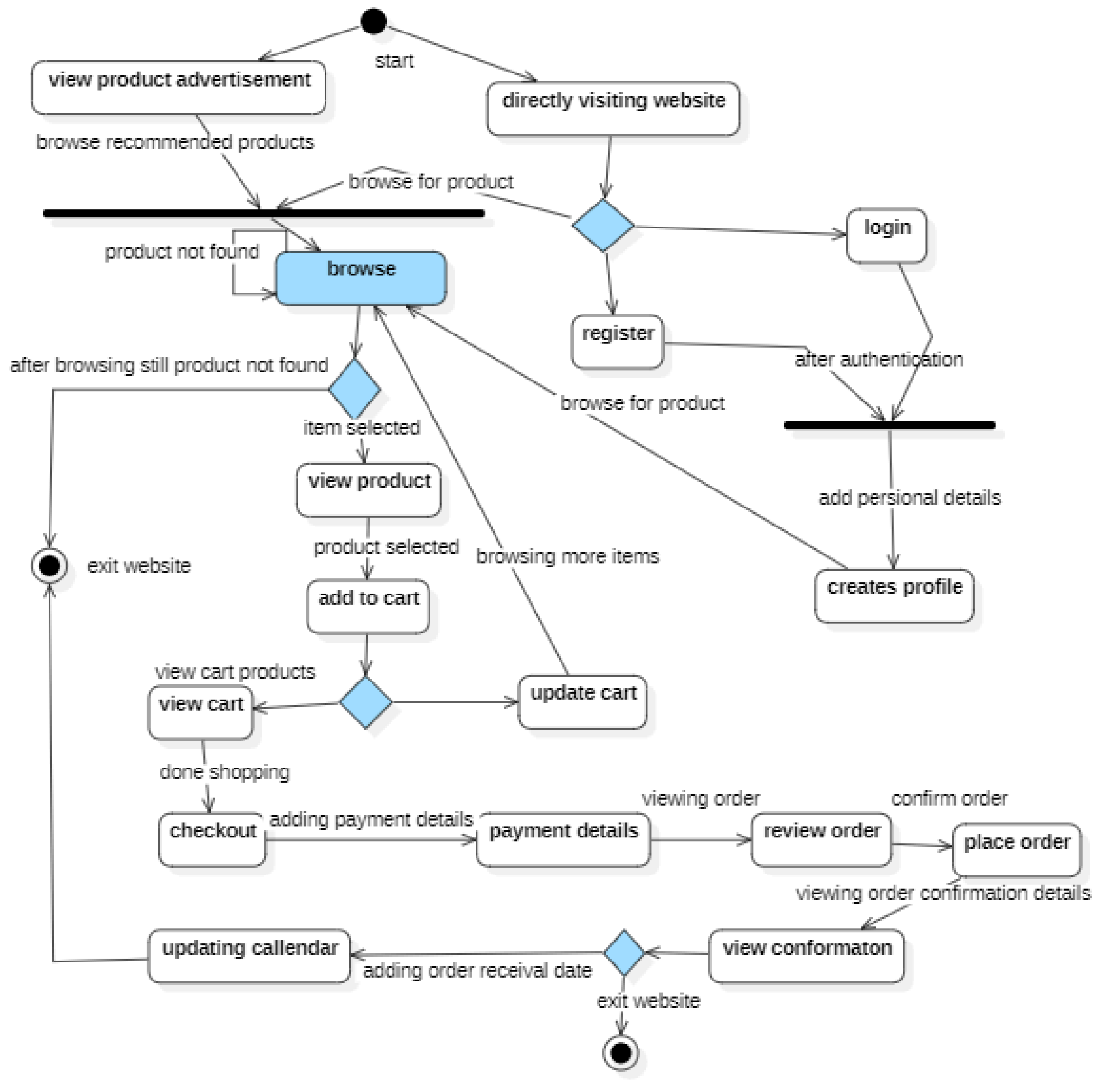
10. **Checkout:**
    - *Methods:*
      - processPayment(), confirmShippingDetails(), generateSummary(): Manages the final steps in the purchasing process, ensuring payment and shipment details are confirmed before order completion.

11. **Notification:**
    - *Methods:*
      - notify(): Sends notifications to users about their order, shipping, or other platform updates.
    - *Relationships:*

        - Managed by the Admin, providing notifications to users.

12. **Authentication:**
    - *Methods:*
        - authenticateUser(), authenticateAdmin(): Handles authentication for both customers and admins, ensuring only verified users can access certain functionalities.

ACTIVITY DIAGRAM:



Activity Diagram Description:

The diagram provides a detailed flow of the customer's interaction with the ArtisianHub system. Each activity phase have been clearly defined below;

1. **Start:**

   The activity begins when the user either views a product advertisement or directly visits the website.

2. **Browse Products:**

   a. The user can browse recommended products or search for specific products. This step aligns with the requirement that users can browse the product catalog and search for items.

    b. If a product is not found, the user can either continue browsing or exit the website.

3. **User Registration/Login:**
    a. If the user is not logged in, they can choose to either register or login.
    b. Upon successful registration or login, the user is authenticated and can browse products further.
    c. After registration, the user creates a profile with personal details, aligning with the requirement that users must provide name, email, and password for registration, and an email verification is required.

4. **View Product:**
    a. Once a product is found, the user can select and view product details, including name, description, price, and image. This aligns with the functional requirement for product listings with descriptions and prices.

5. **Add to Cart:**
    a. After viewing the product, the user can add the product to the cart, satisfying the requirement that users can add products to their cart.

6. **View Cart & Update Cart:**
    a. The user can view their cart, which displays the selected products and their quantities. The user can also update the cart by changing quantities or removing items.
    b. This process meets the requirements for the shopping cart system, where users can view and update the quantity of items.

7. **Checkout:**
    a. Once the user is done shopping, they proceed to checkout.
    b. At this stage, they provide payment details and shipping information, as required by the checkout process.

8. **Review Order & Place Order:**
    a. Before placing the order, the user can review the order, including product details and pricing.
    b. The user then confirms the order by clicking place order, meeting the requirement for a confirmation message or email after a successful order.

9. **Order Confirmation:**
    a. After placing the order, the user is shown the order confirmation details.
    b. The system also adds the order receipt date to the calendar, as part of managing order delivery.

10. **End:**
    a. The activity concludes either when the user exits the website or after completing the order process and viewing the confirmation.

Singleton Patterns:

The following classes are modified to adhere SOLID principles more effectively, in singleton pattern the classes are following Single Responsibility Principle (SRP). By allowing this one class is made to handle the instance creation and its logic, the rest of classes will be handling the other logics.

Singleton with SRP

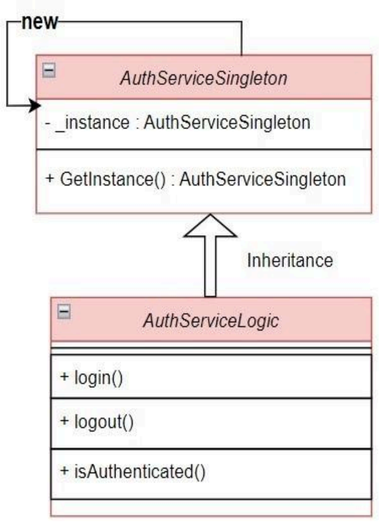
Each feature will have:

1. A Singleton Manager that handles the instance creation (ensuring only one instance).
2. A Service Class that contains the actual business logic (methods).

1. Singleton Manager for AuthService (User Authentication)

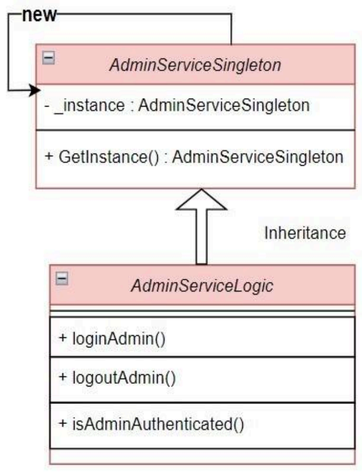
Class AuthServiceSingleton ensures that only one instance is created and class AuthServiceLogic

responsible for the actual business logic related to user authentication.



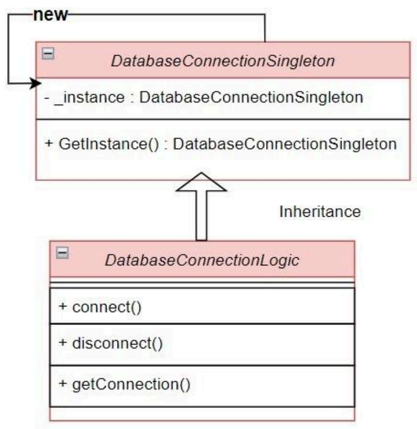
2. Singleton Manager for AdminService (Admin Authorization)

Class AdminServiceSingleton class ensures that only one instance is created and class AdminServiceLogic is responsible for the actual business logic related to admin-specific tasks.



3. Singleton Manager for Database Connection

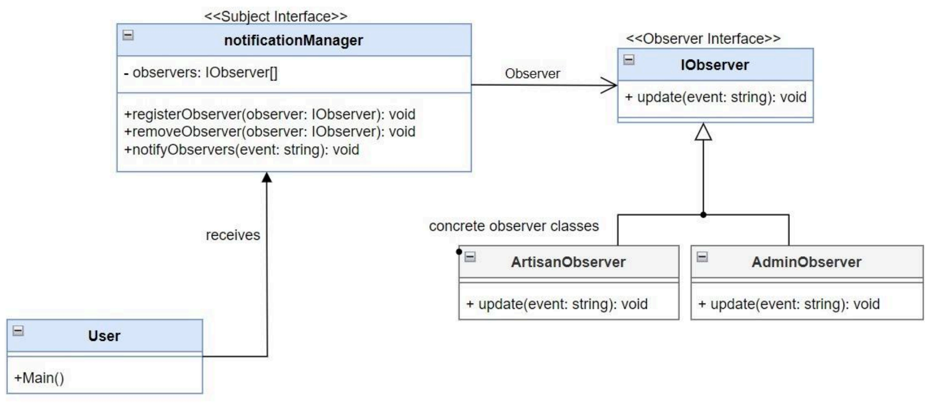
Class DatabaseConnectionSingleton class ensures that only one instance is created while the DatabaseConnectionLogic class handles the connection logic.


new
DatabaseConnectionSingleton
- _instance : DatabaseConnectionSingleton
+ GetInstance() : DatabaseConnectionSingleton
Inheritance
DatabaseConnectionLogic
+ connect()
+ disconnect()
+ getConnection()

Application of Observer Pattern in ArtisanHub:

In ArtisanHub, the Observer Pattern manages real-time notifications. When events such as new product listings or orders occur, relevant users (artisans, admins) need to be notified. This pattern decouples the notification logic from event generation, ensuring flexibility and scalability.

- ArtisanObserver receives updates for events like new orders or product reviews.
- AdminObserver is notified of user registrations or product listings.



<<Subject Interface>>
notificationManager
- observers: IObserver[]
+registerObserver(observer: IObserver): void
+removeObserver(observer: IObserver): void
+notifyObservers(event: string): void
Observer
<<Observer Interface>>
IObserver
+ update(event: string): void
receives
concrete observer classes
ArtisanObserver
+ update(event: string): void
AdminObserver
+ update(event: string): void
User
+Main()

Observer Pattern Description:

The Observer Pattern in ArtisanHub involves two main interfaces and concrete classes:

1. **NotificationManager (Subject Interface):**
    a. Manages observers and sends event notifications.
    b. Key methods:
        i. *registerObserver*: Registers observers.
        ii. *removeObserver*: Removes observers.
        iii. *notifyObservers*: Notifies all registered observers of an event.
2. **IObserver (Observer Interface):**
    a. Implemented by classes needing event updates.
    b. Method:
        i. *update*: Called when a notification is sent.
3. **Concrete Observers:**
    a. ArtisanObserver: Receives updates on artisan-related events (e.g., orders, reviews).
    b. AdminObserver: Handles notifications for admin-related events (e.g., user registrations, product listings).

Principles Followed by the Observer Pattern in ArtisanHub:

1. **Single Responsibility Principle (SRP):**

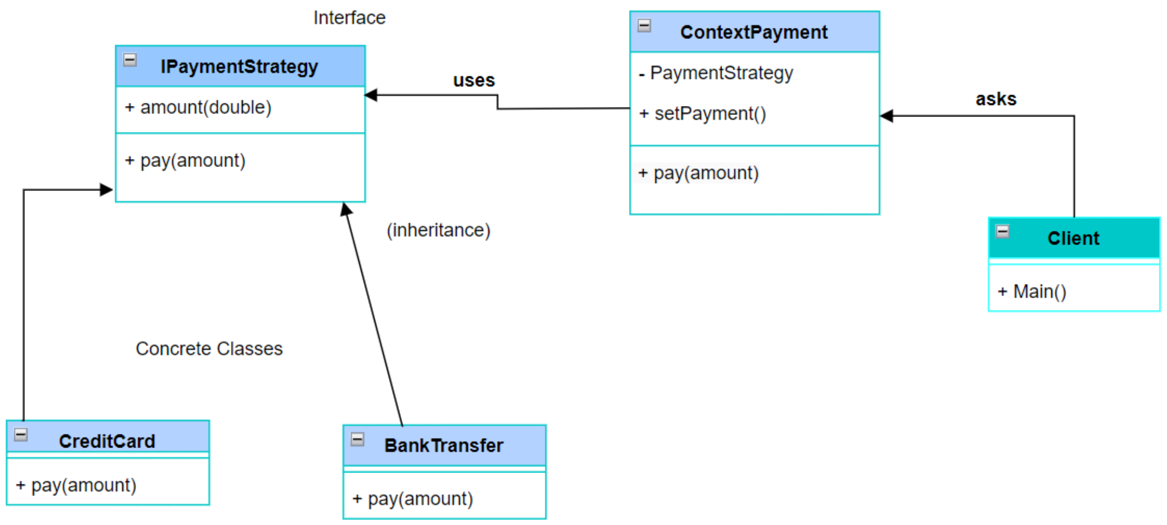
    a. Each class has a distinct responsibility. NotificationManager manages observers and event notifications, while ArtisanObserver and AdminObserver handle the specifics of receiving notifications.
2. **Open/Closed Principle (OCP):**
    a. New observers can be added (e.g., CustomerObserver) without modifying existing code. This makes the system easily extendable for future notification needs.
3. **Liskov Substitution Principle (LSP):**
    a. Any observer implementing the IObserver interface can replace another, ensuring flexibility in using different types of observers like ArtisanObserver or AdminObserver.

These principles ensure the flexibility, maintainability, and scalability of the notification system in ArtisanHub.

Application of Strategy Pattern in ArtisanHub:

In the ArtisanHub project, the Strategy pattern is ideal for managing the payment system by allowing different payment methods, like credit cards, or bank transfers, to be easily switched at checkout. Similarly, it can be used for offering various shipping methods, such as standard or express shipping, where each method represents a distinct strategy for calculating shipping costs and delivery times, giving users flexible options during the purchasing process.

- CreditCard: The customers could be able to pay for the purchased items through which ever credit card they want.
- BankTransfer: The customers could transfer money directly from a bank account.



Strategy Pattern Description:

The Strategy Pattern in ArtisianHub contain two interfaces and concrete classes.

1. **IPaymentStrategy (Interface):**
   a. It allows the users to pay for their purchased product online.
   b. Methods:
      i. *payamount: U*sers are allowed to pay.
2. **ContextPayment (Class):**
   a. PaymentStrategy: Users are asked which sort of payment method they are willing to opt.
   b. setPayment: The amount for the item is set.
   c. Method:
      i. *payamount: U*sers are allowed to pay.

3. Concrete Classes:

a. CreditCard: The customers could be able to pay for the purchased items through which ever credit card they want.
b. BankTransfer: The customers could transfer money directly from a bank account.

4. Client (Class):

a. The client interacts with the system by choosing a payment strategy and making the payment. It communicates with ContextPayment to execute the selected strategy.

Principles Followed by the Observer Pattern in ArtisanHub:

1. **Open/Closed Principle:**
   a. The system is designed to easily add new payment methods (e.g., PayPal) by creating new classes that implement the IPaymentStrategy interface. The existing classes (e.g ContextPayment and Client) do not need to be modified when new payment strategies are introduced.

By applying this principle, the system remains flexible and easy to maintain, allowing for future enhancements without affecting the core logic.