Topic: E-Commerce Application
Members: Eleanor, Jack, Katie, Varsha

Our group focused on developing an e-commerce platform that would allow users to buy and sell products to other users. We made this to be used through the terminal in order to focus on functionality and not worry about user interface design. The primary design issues we focused on were allowing users to make products, store these products, and allowing users to buy these products using multiple payment methods. Users will be able to create multiple types of products and be able to fill various information fields for the product based on its type. When purchasing a product, the user will be able to specify a quantity and which payment method they wish to use. The program will ensure there is enough quantity before allowing the user to purchase a product. A part of the purchasing process will prompt the user to enter information relating to the method, such as credit card details. Once this information is entered and validated, the payment will be processed, and the quantity of the product will be removed from the inventory.

In designing the architecture, we encountered 3 primary problems, which we addressed with patterns. The first was how we could represent products and create them. The second was how to handle product bundles and possible discounts. And the last one was how to allow users to pay in different ways. We addressed these with Prototype, Composite, and strategy, respectively.

In an e-commerce application, the primary problem the Prototype pattern addresses is the inefficient and inflexible creation of new products. Sellers need to add various items to the platform, many of which share common characteristics based on their type (e.g., all electronics have a warranty period, all perishable goods have an expiry date). If we don't use a prototype, a naive approach would be to use constructors for each product type. For example, a `new ElectronicProduct("name", "brand", 1.2, "2-year", "model-xyz", ...)` In the client code, when we do `productTypes.put("Electronic", new ElectronicProduct());`, we create a template for future use, so when a seller creates a product: `Product userProduct = productTypes.get(type).clone();` we call the `clone` method, which performs a shallow copy to create a new object instead of trying to instantiate a new object with the constructor. As for the `ProductBundle`, the `clone` method is overridden to perform a deep copy since it contains a list of `Product`.

While Abstract Factory is good for managing the creation of related families of objects, the Prototype pattern provides a more direct, simpler, and more extensible solution for our specific need of creating customized copies of pre-configured template objects. Especially when it comes to creating similar product bundles, Prototype allows the client code to create product bundles easily without calling multiple factory methods every time. One drawback that comes with this pattern is Shallow vs. Deep Copy. A shallow copy is easy but can lead to bugs where the original object and its clone share internal objects. In `Product`, we use a shallow copy because String in Java is immutable, and it's unlikely that the values will be changed by accident. On the other hand, in `ProductBundle,` we use a deep copy because it contains a list of `Product`.

The ProductBundle is our Composite pattern. In our E-Commerce domain, the bundle solves the practical problem of handling bundled discounts, such as buy one get one free on a certain product. It allows the client to treat all Products, including any ProductBundles, in a uniform way.

The composite pattern allows us to treat all Products and ProductBundles uniformly. Without the Composite pattern, we would need to have conditionals everywhere (for example, in calculatePrice) to check whether each item was an individual Product or ProductBundle. You would also have to do a bunch of casting, and it would get messy quickly. The composite design helps increase maintainability and adheres to the Open-Closed Principle because if you want to change the logic in one type of Product, you only have to change that singular class, whereas without the composite, you would have to go and update all of those if-statements. The Composite pattern also helps increase extensibility because new Product or ProductBundle concrete classes can be added without needing to change any previous code.

The main drawback of using the composite pattern is the number of methods needed in the Product class. By putting all of our composite methods in the Product class, we violate the Interface Segregation Principle because the leaf nodes are forced to depend on methods they don't use and just throw exceptions if they are called on them.

In the checkout domain, the core problem is that we support multiple payment options like credit card, gift card, wallet, UPI, and cash on delivery. Each one has distinct validation rules, data fields, fees, and processing logic. If all were coded directly inside PaymentProcessor.checkout, the class would become a huge if/else or switch mess that is hard to read, test, and extend. Every time a payment type is added or changed, there is a risk of breaking others and violating the Open/Closed and Single Responsibility principles. The Strategy pattern solves this by extracting the varying payment behaviour into separate PaymentStrategy implementations (CardPayment, GiftCardPayment, WalletPayment, UPIPayment, etc.). PaymentProcessor just delegates to the chosen strategy through a common interface (pay, verify, fillInformation). This keeps checkout logic stable while letting us plug in new payment methods without modifying existing code, making the system more modular, testable, and extensible.

Strategy fits perfectly here because the choice of payment method is made at runtime based on the user's preference during checkout. PaymentProcessor can dynamically swap different payment strategies using setPaymentMethod(), treating them as interchangeable algorithms through a common interface (PaymentStrategy). This supports the Open/Closed Principle, since adding a new payment option like Apple Wallet or PayPal UPI only requires creating a new strategy class without modifying the existing PaymentProcessor. Additionally, the design follows the Dependency Inversion Principle by ensuring that PaymentProcessor depends only on the abstraction (PaymentStrategy) and not on concrete implementations, leading to a more loosely coupled and flexible architecture.

A drawback of using the Strategy pattern here is the increased number of classes and files to manage, which can feel heavy for smaller projects.

UML of Architecture Components:

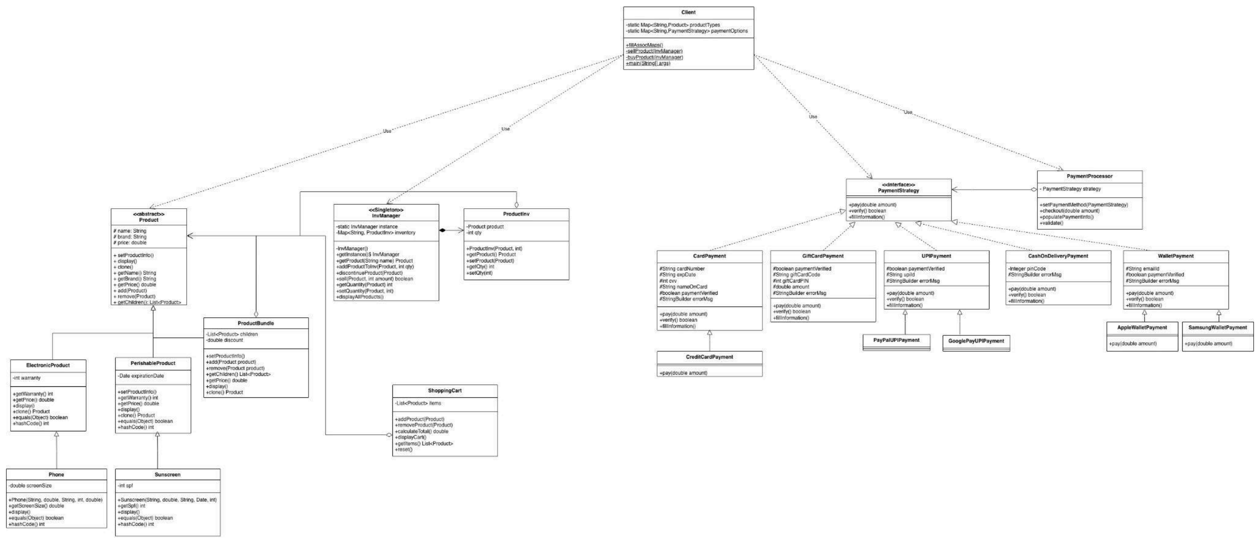

Figure 1: Overall Architecture

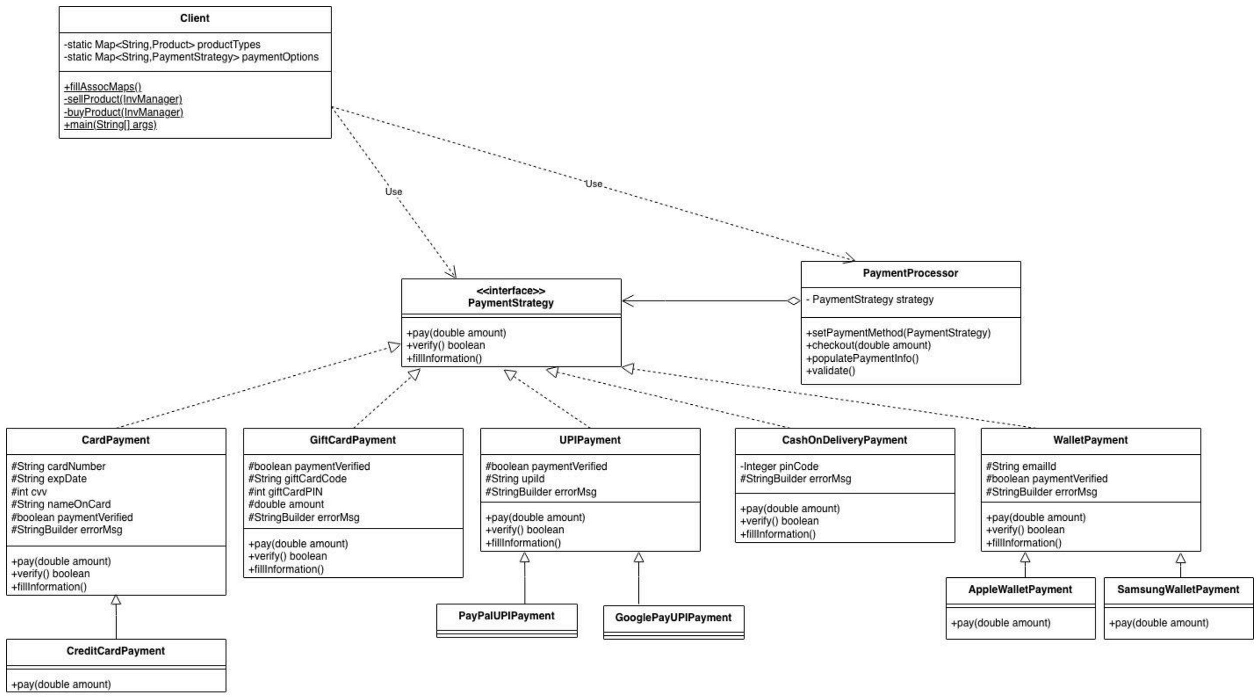

Figure 2: Payment Architecture

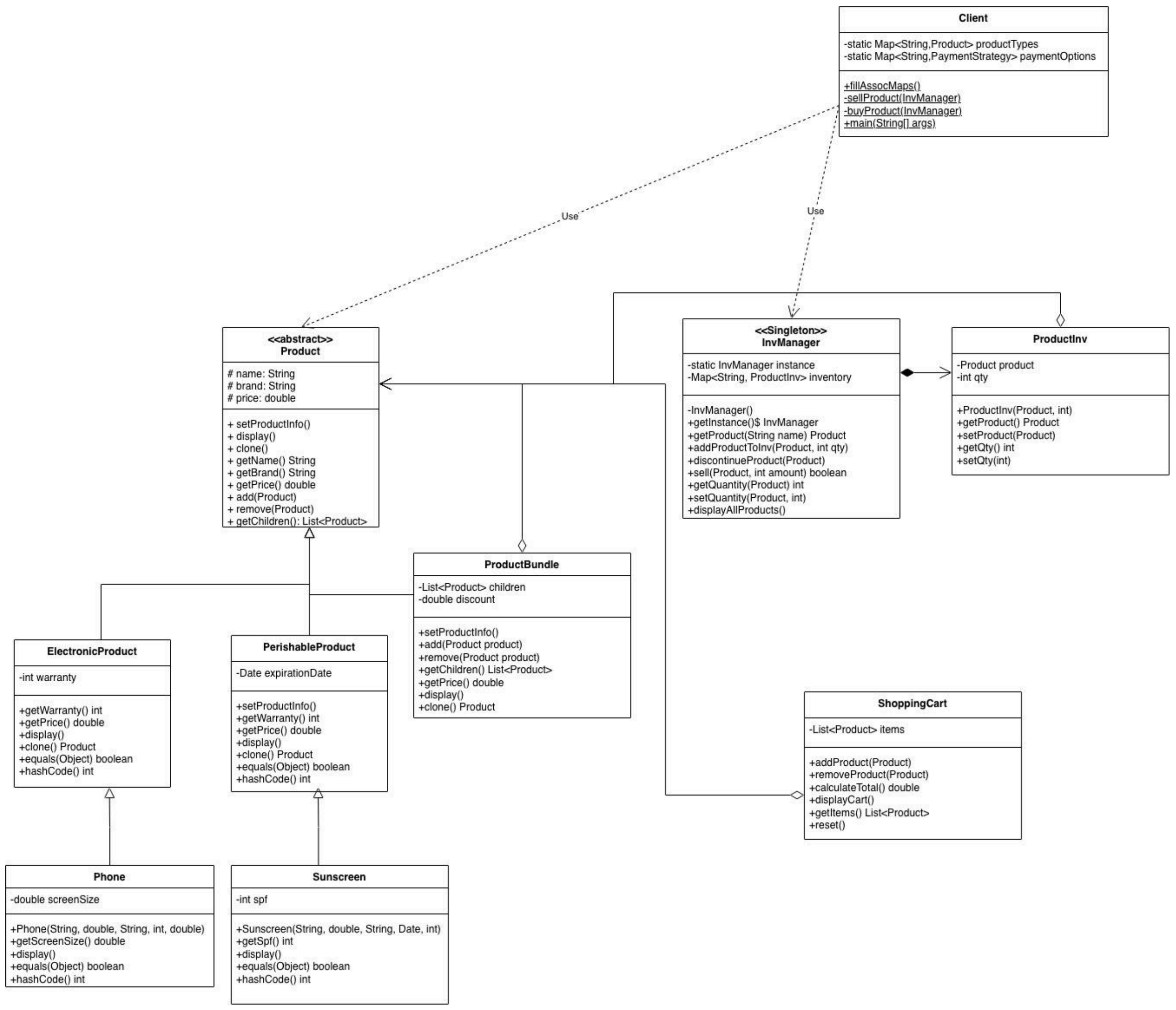

Figure 3: Product/Inventory Architecture

Our design has some advantages for extensibility, but also some restrictions that are present due to the time constraints of the project. If a developer wants to add a new product type, this can be done by inheriting from the existing product class. This new class must override the fill function to fill any class-specific fields, and this fill function must call super before adding any new functionality. Additionally, the new class must override the hashCode, equals, clone, and display methods in order to share proper functionality with other Products. Once this is done, this class will be treated as a Product and can be used as a leaf in the composite pattern. If a developer wants to add a new payment strategy, this can be done by implementing it from the existing payment strategy interface. The developer must have the class override all methods from the PaymentStrategy interface to share all proper functionality needed for payment strategies. The primary pitfall of the extensibility of the design resides in the client, due to the client using a map of products and payment methods, which allows user input to be mapped to the respective class.

Due to this, the design does not fully follow the open-closed principle as developers would need to alter these maps with their new classes.

The current code is designed to be largely flexible, allowing users to add products that might originally be their own class as a basic product. This helps to decrease the need for maintenance and revisions to the code base. For instances where the code might need to be updated, all functionality is divided into methods to help narrow down specific problems and open the design to being tested using unit tests. Additionally, the design works largely off a modular approach, which would allow new functionality to be added freely into the system. For instance, if the software changed to add regular restocking of products from other vendors, a new portion of the code could be created to interface with these vendors and add new products to the inventory. Currently, any new sections of functionality would need to be added to the client, but at some point, this could be changed to interface with a facade pattern to separate the client from all the various portions of the code base.

This project offered a great opportunity to face challenging problems and educate ourselves through the experience of development.

We debated about using Singleton in `InvManager`. We are aware that the Singleton pattern is often considered an anti-pattern in modern software design due to its drawbacks:

- It introduces a global state, which can make the application harder to maintain.
- It creates tight coupling. Any class that calls InvManager.getInstance() is directly tied to that concrete implementation.
- It makes unit testing more difficult, as the singleton cannot be easily replaced with a mock or fake version for testing purposes.

Despite the valid criticisms of the Singleton pattern, we concluded that it was a justifiable choice for the specific scope and context of this project. In our e-commerce application, the `InvManager` class serves as the single source of truth for all products. A critical requirement of the system is that every action must operate on one consistent set of inventory data. Any divergence would lead to data corruption, such as selling items that are out of stock or having multiple conflicting records of product quantities.

Another challenge, although not specifically related to the design patterns we chose, was approaching design with pre-selected patterns in mind. Looking back, we spent most of our time reverse engineering a solution to cater to our chosen design patterns (Prototype, Composite, and Strategy), but the process turned out to be harder than actually implementing the design patterns. While our final mapping of patterns to problems was logical, it's important to note that patterns are frameworks for solving common problems, not a checklist to be completed.