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

Our group decided to focus on designing an E-Commerce application which would allow users to buy items and would allow sellers to post items on the platform based on various product outlines. For the sake of the class and the ease of development, we will start with a basic terminal interface and possibly move to GUI implementation as time goes on. We have outlined multiple subsystems and associated patterns which we might use to solve them. But as we continue with development and learn more about behavioral patterns we believe that we may add more to our architecture.

Given the model we intend to go with we currently see three primary problems which need to be solved. Different products need to be created by sellers so they can be bought by users; likely many products will be made and will all be made in a similar fashion based on the type of product. Additionally many stores offer bundles of items whether it's for a user discount or for user convenience, our E-commerce website would also like to include this functionality and as such we need to find a way to group products and have a user be able to buy multiple products by buying the group. Lastly users will likely want to pay with different means and as such a system needs to be implemented so users can utilise multiple different payment options and pick their platform of choice.

To handle the product creation we determined that a Prototype pattern would be most optimal. This will allow for many products to be created using standardised blueprints designed for larger groups of products. For instance all electronics being sold on the platform could be cloned from a standardised electronic product Prototype. This would allow for quick and effective creation of many products and save the complexity of making multiple products off a single template and having to redefine fields which may not need to be changed or making a new class for each new product when many will likely hold the same fields.

To handle being able to purchase a bundle of multiple products we decided to utilise the composite pattern. This allows us to treat a bundle as a product itself and allows for users to simply reference this bundle in the code to buy multiple items at once. This reduces the complexity of the code base to be able to implement this as it allows for the avoidance of special case products or a form of iterative data structure to store all of the products to be purchased.

Lastly, as we intend to make a flexible payment processing system which allows users to pay through multiple methods, we want to make it easy for users to use multiple payment platforms at will and be able to switch between them. Because of this we opted to use strategy as a pattern to allow us to have an abstract payment function which is handled by the abstracted "algorithms" of the strategy implementations. We think it will solve the changing implementation better than other counter parts as it allows for

flexibility for the user while still keeping the logic abstracted from the high level implementations.

The system will comprise two profiles, a customer and seller profile. These profiles will control the functionality from the site and communicate with a central store class (whose primary purpose will be to store the products). Seller profiles will be able to use the prototype pattern to create new products associated with the composite structure and place the products into the general store class. Meanwhile, the customer profile will be able to access the general store class and select a product. When the product is selected it will default to the payment system specified by the user on the home page (which will update the strategy used by the profile for payments).

Pattern(s)	Design problem	Justification for chosen pattern
Strategy	Flexible Payment Processing: Users must be able to select and switch between multiple payment methods (e.g., Credit Card, PayPal, Crypto) without the core checkout process needing to know the implementation details of each method.	Strategy encapsulates each payment method into its own "algorithm" class. The main user profile or checkout context simply holds a reference to the active payment strategy. This makes the payment system highly flexible and extensible, as new payment methods can be added as new strategy classes without modifying the core system logic.
Composite	Product Bundling and Uniform Treatment: The system must allow users to purchase a group of items (a bundle) as a single entity, and the code (e.g., pricing, inventory) must be able to treat the bundle the same way it treats a single product.	Composite allows for the creation of a tree-like hierarchy where individual products and product bundles (which contain other products/bundles) implement the same interface. This ensures that the application can use a common API to calculate the price, list contents, or add items to the cart, regardless of whether it's a simple product or a complex group.
Prototype	Efficient Product Creation: Sellers need a rapid, standardized way to create many different products (like multiple types of electronics) that share a common structure or "blueprint."	Prototype allows new products to be created by cloning existing standard product blueprints, avoiding the overhead of re-initialization and complex factory hierarchies. This is highly effective for mass creation of similarly-structured products, saving development complexity and ensuring consistency.