Software Engineering Methods: Assignment 1 Part 2

Group 04a: Bogdan Iordache, Kyongsu Kim, Matyáš Kollert, Anna Lantink,
Kristóf Lévay, Javier Páez Franco

December 23rd, 2022

1 Design Patterns

For the following design patterns, we used the lecture slides as our guide.

1.1 Chain of Responsibility

A chain of responsibility design pattern takes a request and passes it along a chain of handlers, that each take the request and process it in their own way. The UML for this can be seen in Figure. 1 We use the chain of responsibility design pattern when we create a coupon. Before we implemented this design pattern, the function that created a coupon had several layers of conditionals that checked if a coupon was valid in a specific sequence. This led to a method that was very clogged and unnecessarily complex. We replaced this with a chain of responsibility pattern by creating a handler interface that defined a method to handle the coupon request, and a method to set the next handler. Then we turned each validity check into its own handler. If a validity check fails, then that validity check throws its corresponding exception, the chain stops, and a coupon is not created. If none of the handlers throw an exception, and the chain completes execution, then that means a coupon is valid and it can be created correctly.

We chose to use this pattern because, as stated above, certain validity checks must occur before others. For example, the format of an expiration date must be checked before the expiration date can be analyzed to see if it is before the current date. We also did it because this chain allows us to determine which validity checks are executed at runtime. This is necessary because not all coupons need to be validated the same way, for example the discount percent of a coupon doesn't need to be checked if the coupon is not of discount type. Using this pattern allows us to abstract a way a lot of the complexity of validity checking from the creating a coupon service. This would also allow other validity handlers to be added to the chain in the future without breaking the current code, which many be necessary if different coupon types are to added at a later date.

The chain of responsibility can be found in the coupon package in the order microservice. The validators are in their own package, and the design pattern is used in the create coupon service.

1.2 Strategy

A strategy design pattern allows the strategy for the implementation of a certain function to be decided at runtime. We use a strategy design pattern for the Coupon component when applying different types of coupons. Since we have different coupon types, such as discount and buy one, get one free, these require different implementations for when the coupon is applied. So, to implement this design pattern we create an interface for the pricing strategy, which has a *computePrice* method that takes in a list

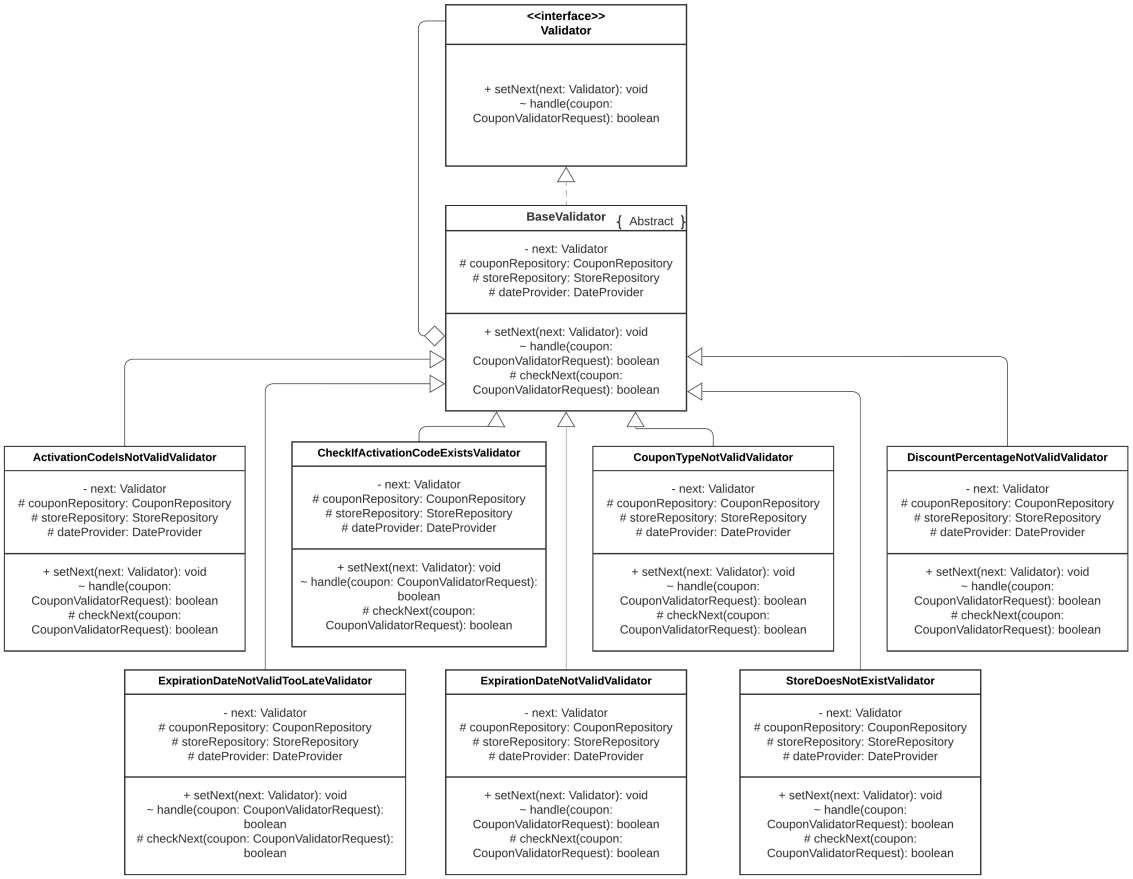

Figure 1: UML diagram of the chain of responsibility design pattern

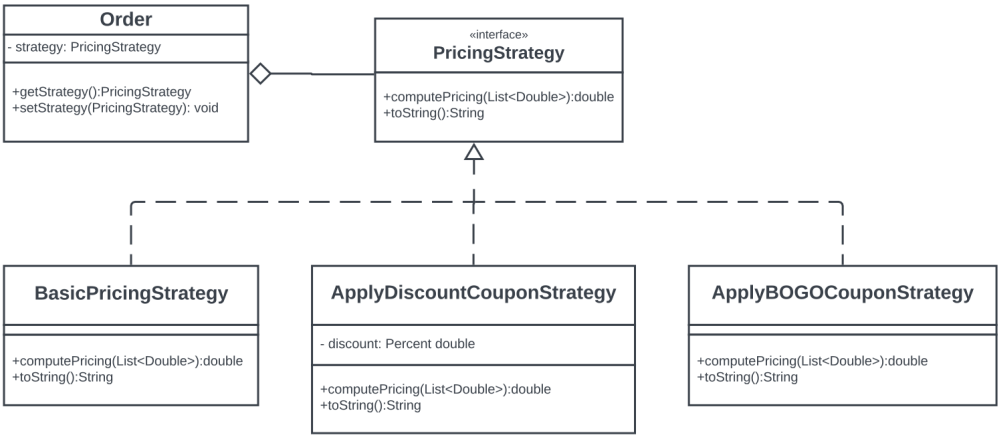

Figure 2: UML diagram of the strategy design pattern

of doubles that represents prices. Then we create three classes, one for when the discount coupon is applied, one for when the buy one get one free coupon is applied, and one for when no coupon is applied - a 'basic' strategy. All three of these classes implement the pricing strategy interface. The Order entity is the context for the strategy, since Order is where all of the pricing is computed. When Order initializes a Pricing strategy, it defaults to the basic strategy. Order's strategy implementation is changed at runtime by ApplyCouponService when a user access the apply coupon endpoint. In this way, Order never has to know what strategy is being implemented, which creates a layer of abstraction.

We decided to use a strategy design pattern because it allows us to separate and organize the different implementations for applying coupons. This design pattern allows us to change implementations at runtime, which is important because the pricing of an order can change several times at runtime, depending on which coupons are applied. We also chose to use this pattern because it makes it easy to extend the types of coupons that are used, since if another coupon application method is desired it can just be added as another class extending the interface.

The strategy design pattern can be found in the order microservice. It is implemented in both the order entity and the apply coupon service.