SEM Assignment 1

Task 2: Design Patterns

Pattern 1: Strategy Pattern for Coupons

For coupons we have decided to implement the Strategy design pattern, since we can have different ways to calculate the total price of an order based on the provided coupons. As per the Strategy pattern dictates, we dynamically choose a price calculation algorithm at runtime based on the provided coupon type. We achieve this by having an abstract class *Coupon* and multiple implementations of it in the form of *PercentageCoupon* and *TwoForOneCoupon*. Each of these implement their own version of *calculatePrice*.

Whenever a request to create a new coupon is received, we check the *type* field and continue parsing the JSON body according to the desired type of coupon and then store the newly created object in the database. When a place/edit order request is received, we query the database for all the included coupons, and apply all of them on the order until we find the minimum price. Each implementation of the price calculation method gets run once, we decide which one is the best, and we stick with it for the final price. The runtime selection of the algorithm is enabled by the Liskov substitution principle, thanks to our abstract class approach.

We think this is the best approach when it comes to applying coupons because of its ease of use and extensible nature provided by its modularity.

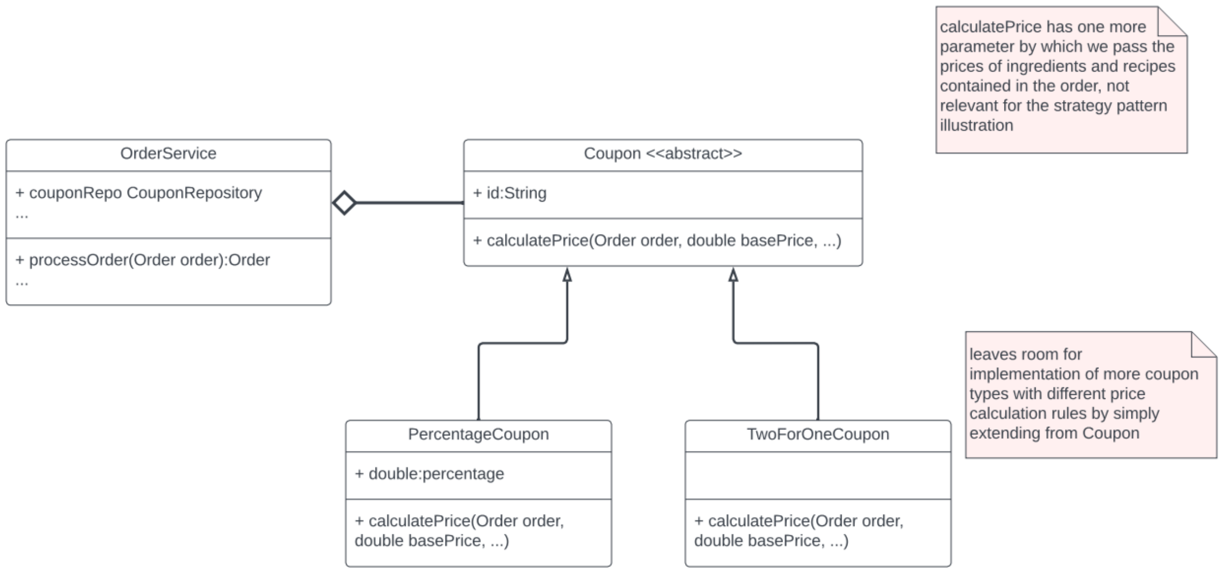

calculatePrice has one more parameter by which we pass the prices of ingredients and recipes contained in the order, not relevant for the strategy pattern illustration
OrderService
+ couponRepo CouponRepository
...
+ processOrder(Order order):Order
...
Coupon <<abstract>>
+ id:String
+ calculatePrice(Order order, double basePrice, ...)
leaves room for implementation of more coupon types with different price calculation rules by simply extending from Coupon
PercentageCoupon
+ double:percentage
+ calculatePrice(Order order, double basePrice, ...)
TwoForOneCoupon
+ calculatePrice(Order order, double basePrice, ...)

Figure 1: Class diagram for strategy pattern implementation in Coupons

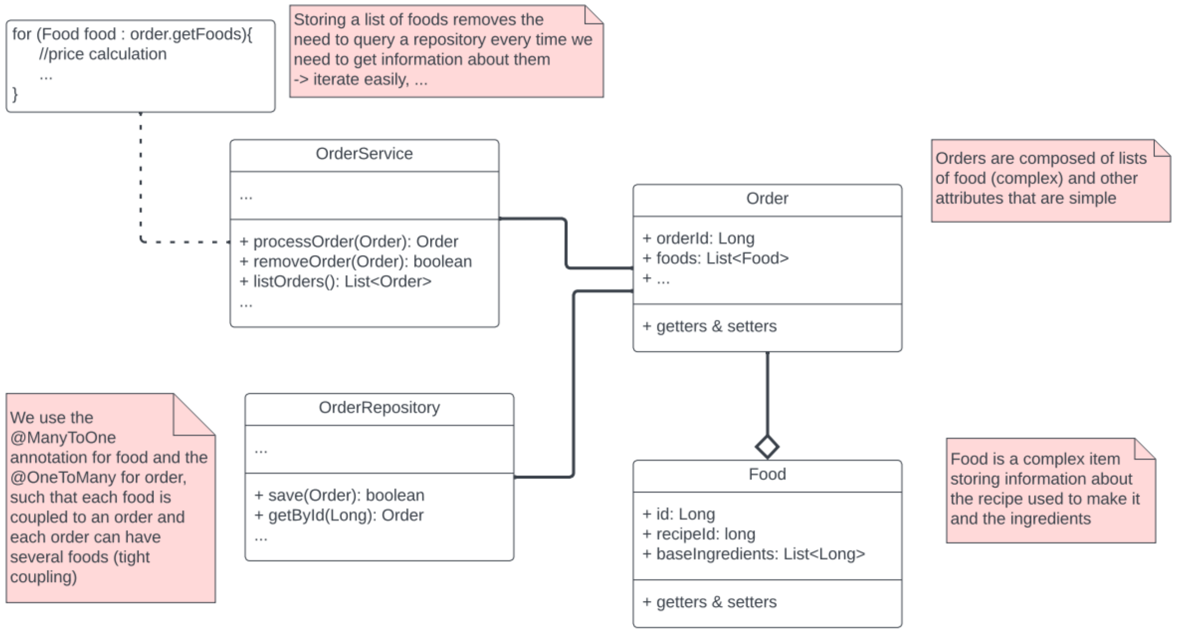
Pattern 2: Composite Pattern for Orders

We have analysed our needs for storing and processing orders and we have decided to implement the Composite pattern. We achieve the desired functionality by having an *Order* object keep multiple *Food* objects as composites. This two-tier structure is exactly what the Composite pattern describes.

From the perspective of controllers and service interfaces, orders are one uniform type. From the perspective of repositories and internal service logic, orders have hierarchical structures, as you can see in Figure 2. This way, data and business logic alike can be divided logically such that they can be easily accessed and processed by their respective handlers (repositories, services, etc.). As such, the concept of a food only exists in the context of an order.

We consider that the Composite pattern suits our use case perfectly, since *Food* is only used in the context of an *Order* and is tightly coupled with it as a result. Aggregation (storing food ids in orders) is also a possible alternative we have considered, but that would complicate the entire system unnecessarily due to a higher number of database accesses, and we think that the additional modularity and expandability is beyond the scope of our project. *

Currently we also make use of aggregation on top of compositions. However, we would like to increase the use of composition in our system, which we plan to do during the refactoring assignment.



for (Food food : order.getFoods){
//price calculation
...
}
Storing a list of foods removes the need to query a repository every time we need to get information about them -> iterate easily, ...
OrderService
...
+ processOrder(Order): Order
+ removeOrder(Order): boolean
+ listOrders(): List<Order>
...
Order
+ orderId: Long
+ foods: List<Food>
+ ...
+ getters & setters
Orders are composed of lists of food (complex) and other attributes that are simple
We use the @ManyToOne annotation for food and the @OneToMany for order, such that each food is coupled to an order and each order can have several foods (tight coupling)
OrderRepository
...
+ save(Order): boolean
+ getById(Long): Order
...
Food
+ id: Long
+ recipeId: long
+ baseIngredients: List<Long>
+ getters & setters
Food is a complex item storing information about the recipe used to make it and the ingredients

Figure 2: Class Diagram for composite pattern implementation in orders

* That being said, we do use a form of aggregation between *Order* and *Coupon* classes, since there is a co-dependency between these two (one does not wholly encapsulate the other).