Implementing Design Patterns

Application or use cases of various design patterns

Assignment #1

Professor:

Maxim Glaida

Students:

Lucía Digón (se24m504@technikum-wien.at)
Saúl Ariel Castañeda (se24m027@technikum-wien.at)
Axel Preiti Tasat (se24m503@technikum-wien.at)

Due date:

Thursday 20th March 2025

Application idea

The idea consists of a restaurant management API that helps order handling, inventory tracking and kitchen operations. The system allows waiters to see the menu, check dish availability and place orders for different tables. The kitchen staff can monitor incoming orders, to help a more efficient preparation. Additionally, the API includes a warehouse management component that tracks ingredients and sends notifications when stock is added or runs out, making it easier to manage dish availability.

The API is designed to be adaptable, as restaurants can integrate it with a UI to have an efficient digital ordering system but it can also be used as the backend for a restaurant simulation video game, for example.

Limitations and possible improvements

The system has some limitations and opportunities for improvement. For example, the kitchen currently processes orders simultaneously up to a certain limit, without considering the number of dishes in each order. While this could work for a videogame, it is not realistic for a restaurant since orders would be prepared based on priority available resources (like cooks, ovens, pans, etc) and based on cooking time. Additionally, restaurants might start preparing individual dishes from an order instead of all at once. A possible improvement would be to implement a more advanced order management system that takes these aspects into account.

Some other improvement could be adding more flexibility to dish customization. For instance, allowing customers to remove base ingredients or keeping dishes available even when an ingredient is out of stock, if it can be prepared with some modifications.

From the code aspect, it would be worth noting that the order status lifecycle can be improved by implementing the **State** design pattern. By doing this, the OrderService doesn't have to worry about the current status of the order, as it currently has if statements to check this. Instead, it tries to execute the corresponding method. In case the order is in an invalid status for that operation, the method will throw an error and abort it.

Design patterns implemented

In this section, we will explain the design patterns that were implemented for this project, and how they helped to decouple and improve the flexibility of the application business logic.

Decorator

The **Decorator** pattern was implemented to allow customers to customize the menu dishes to their liking, by adding ingredients, applicable to the dish. Instead of modifying it, the base `Dish` class implements the `OrderableDish` interface in order to accept extra ingredients by wrapping it with the `OrderableDishIngredientDecorator` class. In the following extract of the `OrderServiceImpl` code we can clearly see how the decorator allowed us to easily add new ingredients to the base dish and get the `OrderedDish` as result.

```
OrderableDish orderableDish = dish;
List<Ingredient> optionalDishIngredients = dish.getOptionalIngredients();
for (Long addedIngredientId : addedIngredientIds) {
   Optional<Ingredient> maybeIngredient =
ingredientRepository.getIngredientById(addedIngredientId);
   if (maybeIngredient.isEmpty()) {
       throw new IllegalArgumentException("The ingredient " +
addedIngredientId + " does not exist");
   }
   Ingredient ingredient = maybeIngredient.get();
   if (!optionalDishIngredients.contains(ingredient)) {
       throw new IllegalArgumentException("The ingredient " + ingredient +
" is not usable in the dish");
   }
   Stock stock = ingredient.getStock();
   if (stock.isEmpty()) {
       throw new IllegalStateException("The ingredient " + ingredient + "
has no stock");
   }
   orderableDish = new OrderableDishIngredientDecorator(dish,
ingredient);
}
OrderedDish orderedDish = new OrderedDish(orderableDish.getDish(),
orderableDish.getIngredients());
```

Figure 1: Usage of the Decorator pattern

Strategy

The **Strategy** pattern is applied to the stock notification handling, allowing the system to determine the appropriate response when there is no more stock or a restock. Instead of hardcoding responses, different strategies were implemented to handle the different stock changes, improving flexibility and making it easier to add future strategies if needed or new notifications are implemented.

We can see the implementation of this design pattern in the notify method of the `DishServiceImpl` class. First it asks the `StrategyManager` for the corresponding strategy, given the type of the notification, and then it executes the `processNotification` method of the class.

```
@Override
public void notify(Notification<Stock> notification) {
   StockNotificationDishStrategy strategy =
strategyManager.getStrategy(notification.getNotificationType());
   strategy.processNotification(notification);
}
```

Figure 2: Usage of the Strategy pattern

For this project, we implemented two `StockNotificationDishStrategy` strategies:

- `OutOfStockNotificationDishStrategy`: Makes the dishes that use the ingredient unavailable.
- `AddedStockNotificationDishStrategy`: For every dish that uses the notified stock ingredient, checks if it can be available again.

Visitor

The **Visitor** pattern is implemented to make bills more flexible and easier to manage. It is used to generate bills for different orders by separating the billing logic from the order classes themselves. This way, if billing rules change or new types of orders are introduced, we can update the visitor without modifying the existing order classes, keeping things clean and maintainable.

For this project, the `Bill` is constructed from the ordered dishes and the table assigned to the order, as we can see in the following extract of the `OrderServiceImpl` class.

```
@Override
public Bill getBill(long orderId) {
    ...
    SimpleBillVisitor billVisitor = new SimpleBillVisitor();
    order.accept(billVisitor);
    table.accept(billVisitor);
    return billVisitor.getBill();
}
```

Figure 3: Usage of the Visitor pattern

As we can see in Fig. 3, the double dispatch approach appears in the `accept` method of the `BillVisitor` interface, implemented by both `Order` and `Table` classes.

Observer

The **Observer** pattern is used for real-time stock updates and order tracking. When an ingredient runs out of stock, the system automatically notifies the necessary components so the issue can be addressed. This ensures that unavailable dishes are updated accordingly, preventing waiters from offering them to customers. Similarly, when an order is ready, waiters are notified instantly instead of having to manually check with the kitchen.

Once an UI is implemented, this could be extended to push notifications, making the process smoother, and allowing waiters and chefs to focus on other tasks without constantly checking in with the kitchen and counter, respectively.

Furthermore, from a coding perspective, this pattern improves the decoupling of services, improving manageability and flexibility.

As a usage case, it is used by a waiter to notify the kitchen that a new order must be queued for cooking. As we can see in Fig. 4, the `OrderServiceImpl` class acts as the producer of the communication by notifying the new placed order. This message is received by the `KitchenServiceImpl` class, that implements the `Observer` interface, to add the order to the corresponding kitchen order queue, as it is shown in Fig. 5.

```
@Override
public void queueOrder(long orderId) {
    ...

notifier.notify(newOrderNotificationFactory.createNotification(order));
}
```

Figure 4: Usage of the Observer pattern (publisher)

```
@Override
public void notify(Notification<Order> notification) {
   Order order = notification.getPayload();
   Kitchen kitchen = order.getRestaurant().getKitchen();
   ordersToCookByKitchen.putIfAbsent(kitchen, new LinkedList<>());
   ordersToCookByKitchen.get(kitchen).add(order);

   getNextOrder(kitchen.getId());
}
```

Figure 5: Usage of the Observer pattern (subscriber)

Facade

The **Facade** pattern is implemented throughout the project by having controllers, services, and repositories. The service layer acts as a facade that hides the complexity and allows clients, via controllers, to interact with the system without worrying about the underlying business logic. In the same way, the repository layer abstracts services from the persistence proceeding.

This keeps the code cleaner, more organized, and easier to maintain. Also, by working with interfaces, this pattern enables decoupling between layers and allows multiple implementations of the same service or repository.

Factory

The **Factory** pattern was used for handling different types of notifications across the application. Currently, since notifications are pretty simple, it might not seem necessary but it sets things up for future changes. If notifications get more complex, such as needing different formats, priorities or more processing, the factory will take care of it, thus keeping it flexible for the future. Also, they abstract the way notifications are built to services.

One use case of the notification factory is to send a message to the waiter when an order is ready to be delivered. When the `setOrderReady` method is executed, the `KitchenServiceImpl` creates an order-ready notification by using the `OrderReadyNotificationFactory` instance.

```
@Override
public void setOrderReady(long orderId){
  ...

notifier.notify(orderReadyNotificationFactory.createNotification(order));
}
```

Figure 6: Usage of the Factory pattern

Other

Spring automatically manages unique instances of services and repositories, meaning we don't manually implement the **Singleton** pattern. However, the idea is still there since these layers work as single shared instances.

Reflection

When we were brainstorming ideas for the project, our goal was to choose something we liked and find ways to incorporate as many design patterns as possible. At first, we planned a restaurant management system that would allow waiters to view the menu, place orders, and track available dishes. While designing the architecture, we initially identified several design patterns that we could implement and structured our system with those in mind. However, as we started developing the project, we realized that some patterns naturally fit into the system's needs, rather than being forced into the design.

One example of this is the use of the Strategy pattern. While we were thinking of the idea, we tried to think of all of the patterns that we could incorporate into the project and one idea that we had was to have the Strategy pattern to allow the restaurants to choose different ways of sorting the orders, for example by allowing the restaurants to have VIP tables that would be prioritized by the kitchen. Therefore, different strategies could be applied based on the restaurant preferences. However, while we were developing the project, we realized that the strategy pattern would be useful to handle the different stock notifications as different types of stock changes required different behaviors. This allows for a more maintainable and scalable design, making it easier to incorporate new strategies in the future if new types of notifications are needed.

In conclusion, we observed that forcing patterns or trying to make them fit the project was unnecessary. Instead, most patterns emerged naturally based on the system's needs. Design patterns help make the code more flexible, ensuring smoother future changes and improving maintainability. We saw how different patterns work together within a project and how they could be beneficial for future expansions.