Gigih Agung

An example of OOP code smell is “repeated calls of methods in sequence”. Here’s an example in my own personal project, a flashcard app called Kanji Cards:

```
 9   const submitForm = async (e: any) => {
10     e.preventDefault()
11     if (!deckName) { return }
12
13     let obj = {
14       name: deckName,
15       cards: []
16     }
17     store.compileAllDecks()
18     store.replaceDeck(obj)
19     store.saveToLocalStorage()
20     props.history.push("/")
21   }
```

```
23   const uploadJSON = (deckname:string, result:Array<string>) => {
24     let obj = {
25       name: deckName,
26       cards: result || []
27     }
28     store.compileAllDecks()
29     store.replaceDeck(obj)
30     store.saveToLocalStorage()
31     props.history.push("/")
32   }
```

Picture 1 & 2: Lines of code smell in Kanji Cards
Site: https://gigihsigap.github.io/kanjicards
Github: https://github.com/gigihsigap/kanjicards

Here I have a set of actions: (1) Compile All Decks, (2) Replace Deck, and (3) Save To Local Storage. This sequence is executed whenever a Deck is updated, and it happens frequently in the application logic.

We know this sequence will remain the same: (1) --> (2) --> (3), and yet the code is written in duplicates. To solve this smell, we can just make a new class method that executes (1), (2), (3) in order, and just call that new method instead.

Dependency injection separates an object (dependency) construction and its behavior. This is achieved by **passing down dependencies through parameter** instead of instantiating it. This is important to make sure the code does not instantiate a new function/class every time it relies on a dependency. It is run once and the data is passed around.

Example of do's and don'ts when handling request using GET and POST method:

[GET]
Do : If there's a lot of data about to be retrieved from the database (usually when reading ALL data), limit the size. This can be achieved by using pagination, either server-side or client-side.

Don't : GET parameters are passed as part of the URL. Don't reveal sensitive information on the URL.

[POST]
Do : Always authenticate and authorize, because POST request normally changes a resource (I.e., database).

Don't : Forget to validate data in client-side or server-side, causing an error during the request.

Make a design system and transaction flow:

- ✧ User register:
  - POST/user/register
  - Creates a profile with type "user"

- ✧ User input the address:
  - PUT/profile/:id
  - Apply some data verification to ensure data are valid

✧ User choose the products to be purchased with subscription and/or one-time purchase scheme:
- One time purchase: create a transaction history after purchase (POST/profile/:id:/history)
- Subscription: create a new subscription data for the profile (POST/profile/:id:/subscription)

✧ User pay the bill:
- Use a payment library to process transaction
- Create a delivery status for seller and user (POST/delivery)
- Seller checks the delivery status after sending the product from “Preparing” to “Sent” (PUT/delivery/:deliveryId)
- User checks the delivery status once the product is received from “Waiting” to “Received” (PUT/delivery/:deliveryId)

✧ User skip the delivery due to certain reasons (ex: They have other agenda that prevent them to receive the delivered goods):
- After certain period of time, if recipient user still has not checked the delivery status (inactive), there will be an automated process that checks them given the proof of delivery by supplier.

✧ User cancel the order:
- Allow user to verify their delivery status (“Waiting”, “Canceled”, “Received”). User can cancel their order if the supplier didn’t properly send the product.
- This is achieved by (PUT/delivery/:deliveryId), changing the status to “Canceled”

■ Supplier register as seller:
- POST/supplier/register
- Creates a profile with type “supplier”

■ Supplier create the store and complete the address:
- PUT/profile/:id
- Apply some data verification to ensure data are valid
- Supplier would not be authorized to create/edit products until they complete their profile information

■ Supplier create products that can be purchased either daily or one-time purchase:
- If product is a new entry (POST/product)
- If seller is updating an purchase type information (PUT/product/:productId)

- Note: It is also possible to separate database and API endpoints between one-time-purchase product and subscription product.

- Supplier determine the price of each product:
  - If product is a new entry (POST/product)
  - If seller is updating an item price information (PUT/product/:productId)

- Supplier determine the selling area:
  - If product is a new entry (POST/product)
  - If seller is updating an item selling area information (PUT/product/:productId)

- If a product can be sold by more than one seller:
  - When an user searches for a specific item, it will make a GET request to the database to find all product with the required information (for example: "Handcrafted Flower Decoration" might return 25 entries, separated by pagination)
  - User can select which criteria to prioritize (sort by delivery distance, sort by price, sort by rating)
  - For example, if user wants to sort by delivery distance, there would be a logic that 1) checks if User Area is within the Product Seller Area, and 2) calculates the distance between Product Seller Area and User Area.
  - If user has not placed their address yet, they will be redirected to profile page to update their recipient address (PUT/profile/:id)

- There is a cut-off time everyday, which is the latest time an order can be placed for the next day delivery.
  - All orders placed beyond cut-off time will get a pending status.
  - Apply time-sensitive logic to automatically change the status on the day after tomorrow.