ITSS SOFTWARE DEVELOPMENT

Lab 11: Design Concepts

Lecturer: NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn

1. SUBMISSION GUIDELINE

You are required to push all your work to the valid GitHub repository complying with the naming convention:

"<MSTeamName>-<StudentID>.<StudentName>".

For this lab, you have to turn in your work twice before the following deadlines:

- **Right after class**: Push all the work you have done during class time to Github.
- **10 PM the day after the class**: Create a branch named "***release/lab10***" in your GitHub repository and push the full submission for this lab, including in-class tasks and homework assignments, to this branch.

2. IN-CLASS ASSIGNMENTS

In this section, we will get familiar with the two design concepts: cohesion and coupling. You will meet the term "design patterns" frequently in this lab, yet we will focus on this topic in the future, not in this lab.

2.1. COUPLING

In any program, coupling is inevitable, or else, each module would be an independent program. Coupling, however, is still one of the earliest indicators of design quality: the lower, the better. In this part, we will consider each level of coupling among modules in the sample project.

Below are some examples for coupling levels in the source code. You may need to have a full review for it and make changes where necessary.

2.1.1. Content Coupling

Fortunately, the sample project does not have this kind of coupling level yet. On the other hand, there are still some potentials.

To illustrate, let's see the class Order in the package entity.order. Currently, this class has an attribute deliveryInfo of HashMap type and a public getter getDeliveryInfo() for the attribute. In the future, there might be a module which manages to get deliveryInfo object by calling getDeliveryInfo() of Order class and then change the deliveryInfo by calling put() of HashMap class. Consequently, the value of deliveryInfo will be changed while its object Order knows nothing about the modification of its attribute from the external (!).

To decrease the level of coupling, we can:

- Omit redundant accessors/getters
- Choose the suitable accessibility/immutability.
- **Encapsulate** data with a specialized class/object.
- Use design patterns like builder patterns.

Remember to think of the dependency levels among modules when design

2.1.2. Common Coupling

In Java, there is an infamous indicator of common coupling: static method/attribute since we tend to use static when certain data is shared with multiple classes. However, being static is not a sufficient condition.

As can be seen in the class Cart of the package enity.cart, the two static method and attribute are not the signal of common coupling but one of the solutions for common coupling: Singleton pattern.

```
public class Cart {

    private List<CartMedia> lstCartMedia;
    private static Cart cartInstance;

    public static Cart getCart(){
        if(cartInstance == null) cartInstance = new Cart();
        return cartInstance;
    }
```

Obviously, we have a plenty of modules that need to communicate with the class Cart like the handler of Home screen when adding items to the cart, the handler of Cart screen when showing current items in the cart, and the handler of Invoice screen when showing the items to pay. All of them need to access/modify the data of the cart, and we would tend to use a **public static** attribute of class Cart to handle the needs of communication among classes. However, this public static is not bound to object(s), just like imperative programming paradigm (e.g., C programming).

To decrease the level of coupling, we can:

- Pass the **instantiated** object(s) which encapsulate the shared data to **all** objects/classes which need the shared data, i.e., dependency injection. Note that this may result in stamp coupling.
- Apply design patterns like Singleton.

2.1.3. Control Coupling

A module is said to be control coupled when the flow of execution is decided by some **external flags passed**. The infamous indicator of this kind of coupling is the control structures such as if-else conditions or switch-case statements. However, it is also not a sufficient condition.

What if we had more than one payment method, what would you do?

Normally, we might use a control structure like if-else and face with control coupling as follows:

```
void pay(String paymentMethod, args) {
      if (paymentMethod == “NGAN_LUONG”) {
            NGANLUONGSubsystem nganLuong = new NGANLUONGSubsystem();
            nganLuong.pay(args);
      } else {
            OceanBankSubsystem oceanbank = new OceanBankSubsystem();
            oceanbank.pay(args);
      }
}
```

To decrease the level of coupling, we can:

- Separate into different modules
- Using parent classes or interface to have a common method signature, but different implementations in the children classes
- Apply design patterns like strategy pattern or factory pattern

2.1.4. Stamp Coupling

Two classes are said to be stamp coupled if one class sends a collection or object as parameter and only a few data members of it is used in the second class. To decrease the level of coupling, we just need to pass only the necessary parameters (even primitive types if need be) in the function call. Note that stamp coupling is acceptable if this is a choice made by insightful designer, not lazy programmer.

For example, in the sample code, we can find out the method: checkMediaInCart(Media media) is in the stamp coupling, since we only need the media id to check, but pass all media object. To solve that problem, you can only pass the media id as the only parameter.

2.1.5. Data Coupling

If your modules are data coupled, your design is good.

2.2. COHESION

Below are some examples for cohesion levels in the sample source code. You may need to have a full review for it and make changes where necessary.

In general, to increase the level of cohesion, we can try:

- Put each part in another module, which is more suitable (create new modules if need be).

2.2.1. Coincidental Cohesion

Clearly, we can see this type of cohesion in the class Configs or class Utils in the package utils.

2.2.2. Logical Cohesion

Look back the section **2.1.3. Control Coupling**, if we separate the codes into 2 methods and put them in the same class, we might face with logical cohesion. Thus, we should consider put them in the different classes/packages.

2.2.3. Temporal Cohesion

Frequently, we would need this kind of cohesion in initialization or clean-up modules. Quite often, such classes/modules need to have this type of cohesion, and any attempts to change it to more cohesive types are highly ineffective.

2.2.4. Procedural Cohesion

Loops, multiple conditions, and steps in the code can show evidence of procedural cohesion. Those are usually classes that, while being analyzed, make us want to take a piece of paper, a pencil and start to draw a block diagram.

We can see this kind of cohesion in the class PlaceOrderController in the package control. We validate data fields one-by-one with the validation methods.

2.2.5. Communicational Cohesion

Modules in this kind of cohesion work on the same input or return the same type of output data (e.g., InterbankSubsystemController both works on the same input and returns the same type PaymentTransaction of output data).

2.2.6. Sequential Cohesion

There could be only one small issue here. If we have such a sequence, we can cut it in different ways. It means that created classes can do more than one functionality, or quite the opposite – only some part of the functionality.

2.2.7. Functional Cohesion

Look back the code of class API which you have refactored in the previous lab, you can clearly see that the output of one method is the input to another. However, the

also reuse the related functions. Thus, we can conclude this class has functional cohesion. If your module has this kind of cohesion, your design is good.

The references are listed as follows.

Six Shades of Coupling - Mr. Picky (mrpicky.dev)

The forgotten realm of Cohesion - Mr. Picky (mrpicky.dev)

2.3. REVIEWING ALL SAMPLE PROJECT AND YOUR DESIGN

In this section, you are asked to:

- Review coupling and cohesion level and determine those levels in the modules of your design (sample project and the design that you already did by yourself). If your design is not good, propose improvement plan.

- Comment in your source code as follows

+ Coupling/Cohesion level

+ Reason why

- Write a report on the above issues and update the design and source code corresponding to your proposals.

After completing the tasks, please put your report and its exported file in the directory "GoodDesign/DesignConcepts".

Here is a recommended structure for your report.

1. Coupling
   1.1. Content Coupling

Related modules	Description	Improvement Direction

   1.2. …
2. Cohesion
   2.1. Coincidental Cohesion

Related modules	Description	Improvement Direction

2.2. ...