ITSS SOFTWARE DEVELOPMENT
Lab 11: Design Principles & Design Patterns

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/lab11***" 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

From this lab, we will have additional requirements. Then we would try to modify our design and see if the old design violates SOLID design principles. We also could try to modify our design with the help of a few design patterns.

You are asked to work individually for the tasks of this section.

2.1. ADDITIONAL REQUIREMENTS

1. Change how shipping fee is charged:

   The shipping fees now also depend on the actual weight, the distance, and the bulkiness of the product (i.e., the sizes of the product: length, width, and height).

   The conversion formula is given as follows.

   Alternative weight (kg) =Length (cm) × Width (cm) × Height (cm) / 6000

   The weight of products to charge is equal to the sum of actual weight of the product plus the alternative weight.

2. Add new way to make a payment:
    - Domestic debit card:
      + Issuing bank, e.g., VietinBank
      + Card number: 16 digits
      + Valid-from date, e.g., 12/33
      + Cardholder's name, e.g., DO MINH HIEU

For this additional requirement, assume the API stays remained, and only the card info is changed.

3. Change the API:

The interbank decides to release a new API to replace the old one.

2.2. SINGLE RESPONSIBILITY PRINCIPLE

As the name suggests, this principle states that a class should only be responsible for a particular task. Otherwise, it would be more than one reason for it to change, which could have negative impact on software quality like maintainability and readability.

As stated in Lab 06 – Class Design, the InterbankSubsystemController is responsible for 02 tasks: (1) data flow control and (2) data conversion (e.g., to convert the requests with data to the required format and to handle the response). Thus, this class must be changed when either the data flow changes (e.g., new features are added) or the data conversion changes (e.g., the required data format changes), which is an indicator of a poor design.

This principle may sound simple, yet implementing it is tricky and hard.

Question: How can we know the responsibility of a class?

It depends on the business domain, the problem statements, the application architecture, and the perspective of designers. So many men, so many minds; however, we still can have a reasonable solution by considering the cohesion since having more responsibilities in a class must result in lower cohesion.

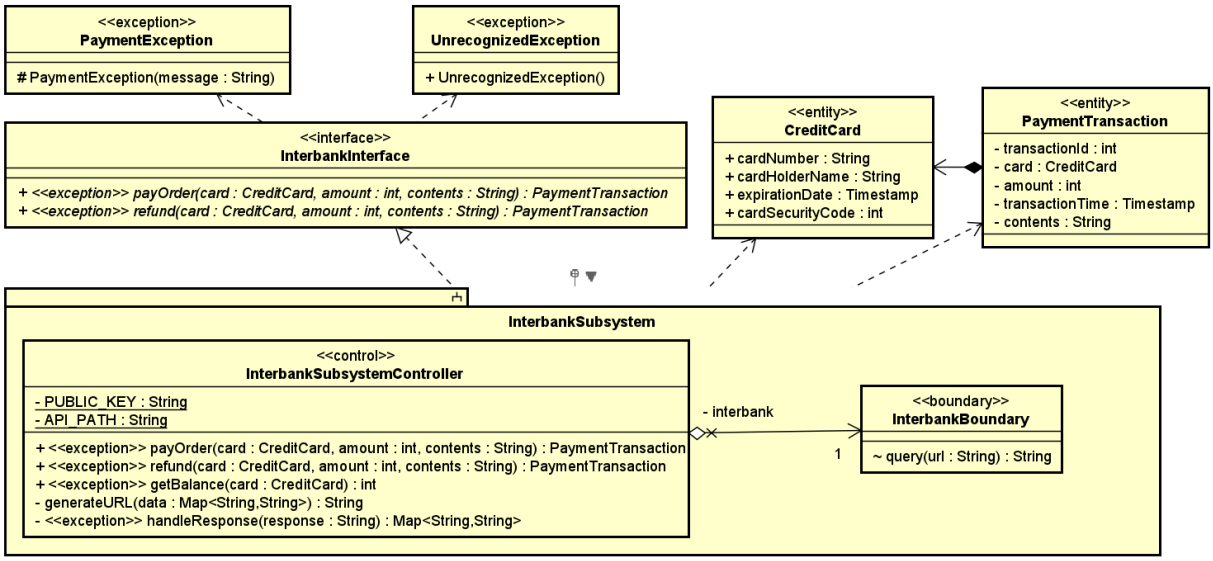
Clearly, the class InterbankSubsystemController has issues with cohesion.

© SOICT – HUST

2.3. OPEN/CLOSED PRINCIPLE

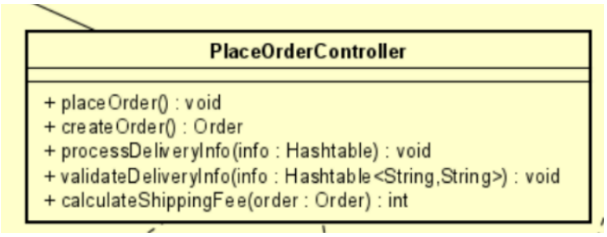
Simply put, whenever we add something new like a new functionality, we write a new class/module extended the old ones, rather than changing the written contents.

As can be seen in the design of the interbank subsystem, we have managed to follow this principle. Abstraction is the key: InterbankInterface. In the future, when the requirements ask us to trade via another API version, we just need to write another subsystem (the supplier) implementing the interface (the abstract supplier).



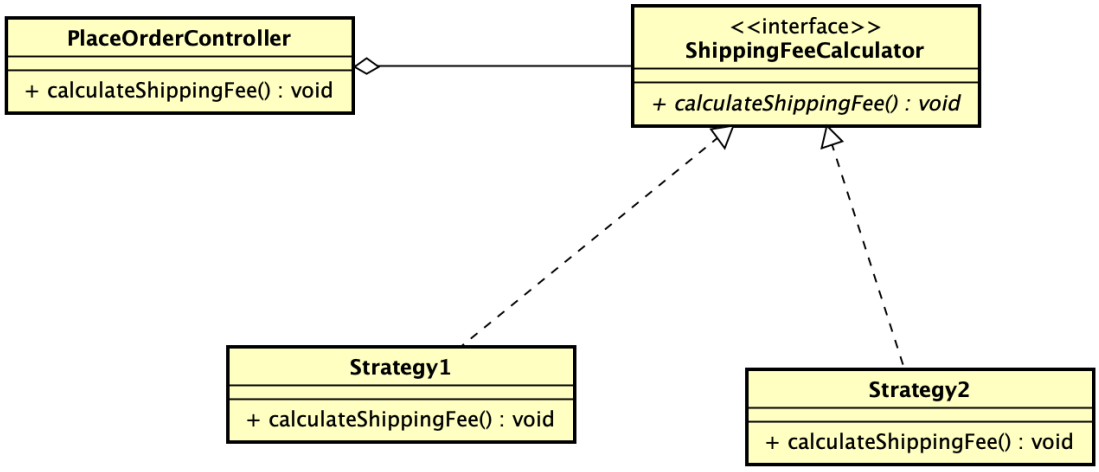
Note that we still might have to modify the codes of the caller/client to call the new callee. Also, you might want to change the name of interface/class/modules into more suitable terms.

On the other hand, the class PlaceOrderController has the method calculateShippingFee to calculate shipping fees.



Clearly, this design violates the principle since we must modify the old codes when changing the formula of shipping fees. We could deal with this issue by using

abstraction: create an interface ShippingFeeCalculator and let the controller depends on it instead of a concrete implementaion.

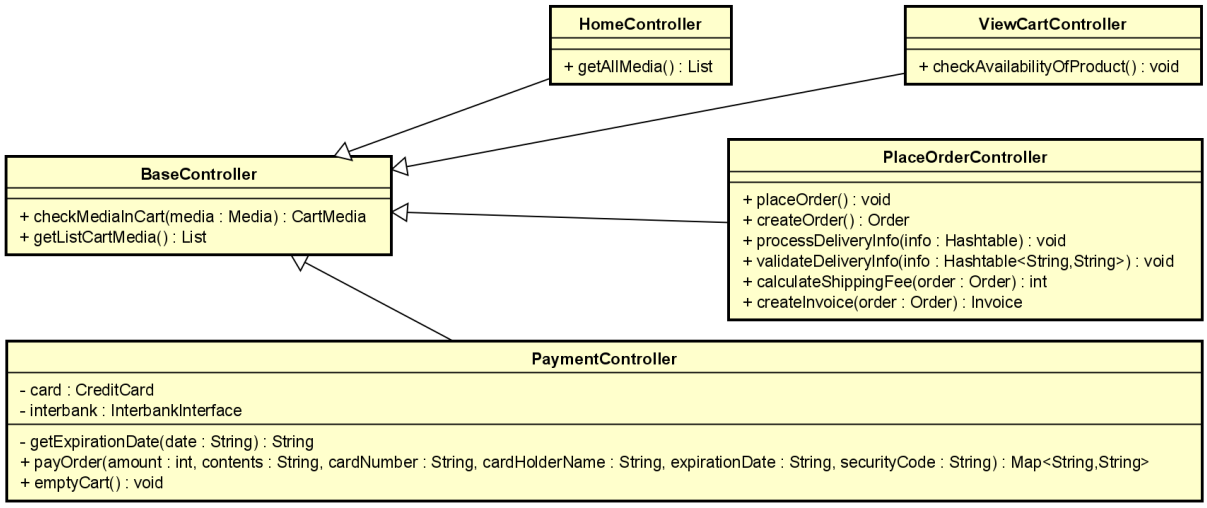


2.4. LISKOV SUBSTITUTION PRINCIPLE

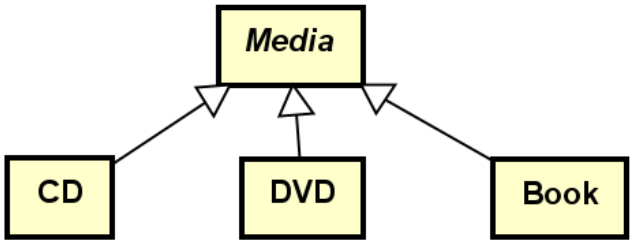
Liskov substitution principle (LSP) defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. Or else, we probably have the wrong abstraction. We can think of the methods defined in the supertype as defining a contract. Every subtype is expected to stick to this contract. If a subclass does not adhere to the superclass's contract, it is violating the LSP[1].

As can be seen, the class hierarchy in the inheritance of the class BaseController has followed LSP.

[1] https://reflectoring.io/lsp-explained/



Now, look at the codes and consider the inheritance hierarchy of Media, is there any LSP violation?




```
public List getAllMedia() throws SQLException{
    Statement stm = AIMSDB.getConnection().createStatement();
    ResultSet res = stm.executeQuery("select * from Media");
    ArrayList medium = new ArrayList<>();
    while (res.next()) {
        Media media = new Media()
            .setId(res.getInt("id"))
            .setTitle(res.getString("title"))
            .setQuantity(res.getInt("quantity"))
            .setCategory(res.getString("category"))
            .setMediaURL(res.getString("imageUrl"))
            .setPrice(res.getInt("price"))
            .setType(res.getString("type"));
        medium.add(media);
    }
    return medium;
}
```

Figure 1- The method getAllMedia() in the class Media

```
@Override
public List getAllMedia() {
    return null;
}
```

Figure 2- The method getAllMedia() in the class DVD

Yes – the method Media.getAllMedia() is expected to return a List, yet all the child classes override it and return null only (!), which might cause NullPointerException if we tried to replace the Media by its child classes. A solution for this case is that we delete the method in the child class since we do not have to override it.

2.5. INTERFACE SEGREGATION PRINCIPLE

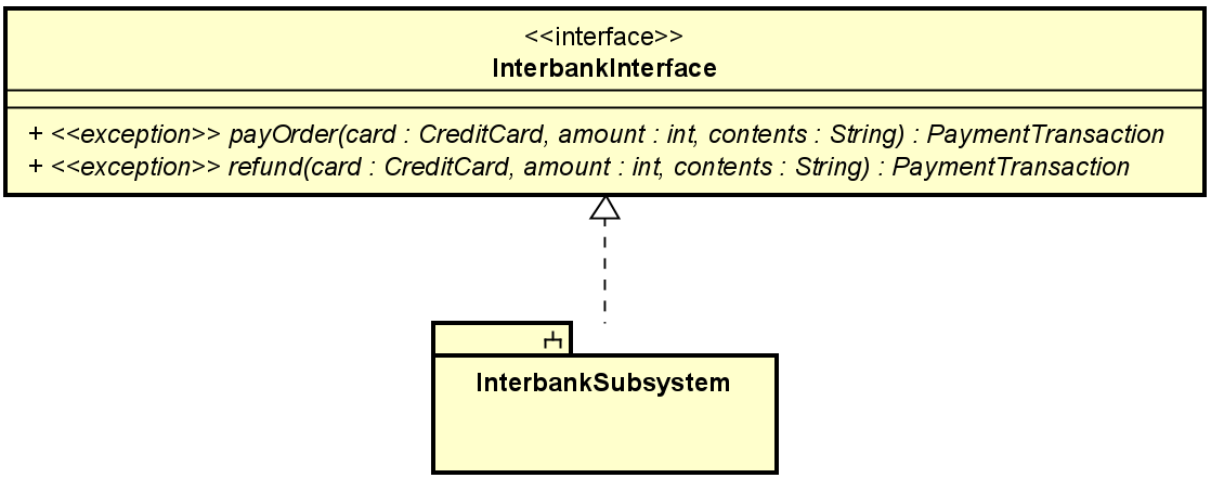
"Clients should not be forced to depend upon interfaces that they do not use." — Robert Martin, paper "The Interface Segregation Principle." In other words, interface segregation principle (ISP) states that declaring methods in an interface that the client does not need pollutes the interface and leads to a "bulky" or "fat" interface.

Clearly, the solution for code refactoring is to divide the fat interface into smaller ones. This solution will surely solve the problems of both ISP violation. Applying the principle to the extreme will result in single-method interfaces, also known as role interfaces. Still, it can result in a violation of cohesion in interfaces, resulting in the scattered codebase that is hard to maintain[2].

For example, the Collection interface in Java has many methods like size() and isEmpty() which are often used together, so it makes sense for them to be in a single interface.

On the other hand, any payment subsystem for a payment actor is expected to have the two operations payOrder() and refund(). Thus, it is inevitable that they are in the same interface.

[2] https://reflectoring.io/interface-segregation-principle/



See the following for more details.

RoleInterface (martinfowler.com)[3]

2.6. DEPENDENCY INVERSION PRINCIPLE

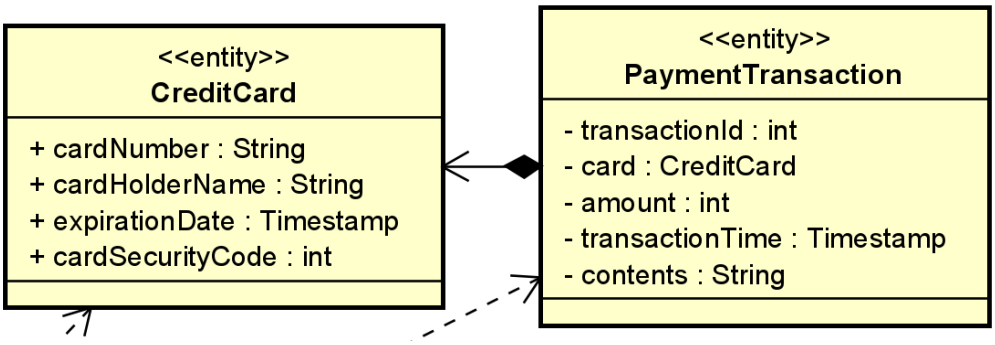
The dependency inversion principle (DIP) states that

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions (Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices).

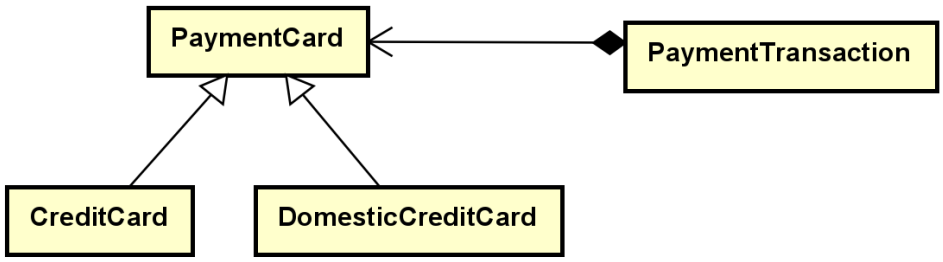
By dictating that both high-level and low-level objects must depend on the same abstraction, this design principle ***inverts*** the way some people may think about object-oriented programming (Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). Hendrickson, Mike; Loukides, Mike (eds.). Head First Design Patterns).

To illustrate, the class PaymentTransaction is tightly couple with the class CreditCard (a concrete implementation). In case the requirement is changed, and we need to have another payment method like domestic debit card, this design cannot be modified without modifying the class PaymentTransaction.

[3] https://martinfowler.com/bliki/RoleInterface.html



A solution is to create an abstract class PaymentCard, and the class PaymentTransaction should depend on this abstract class, rather than the concrete implementation. This solution is an example of applying Strategy Pattern.



2.7. REVIEWING YOUR DESIGN OF UC "PLACE RUSH ORDER"

Here is the list of additional requirements for UC "Place Rush Order":

4. Change how shipping fee is charged:

   The shipping fees now also depend on the actual weight, the distance, and the bulkiness of the product (i.e., the sizes of the product: length, width, and height).

   The conversion formula is given as follows.

   Alternative weight (kg) =Length (cm) × Width (cm) × Height (cm) / 6000

   The weight of products to charge is equal to the sum of actual weight of the product plus the alternative weight.

5. Add new way to make a payment:
   - Domestic debit card:
     + Issuing bank, e.g., VietinBank
     + Card number: 16 digits
     + Valid-from date, e.g., 12/33
     + Cardholder's name, e.g., DO MINH HIEU

For this additional requirement, let's assume the API stays remained, only the card info is changed.

In this section, you are asked to:

- Verify if your design of UC "Place Rush Order" has complied with the SOLID principles before and after having additional requirements. If not, please propose an improved design so that the SOLID principles are hold.

- Write a report on the above issues and modify your codes to implement your proposal. Note that you do not have to implement the additional requirements.

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

Here is a recommended structure for your report.

1. Single Responsibility Principle

#	Related modules	Description	Improvement
1.1.			

2. Open/Closed Principle

#	Related modules	Description	Improvement
2.1.			

...