LECTURE 12

SOLID Design Principles & Law of Demeter (LoD)

What's a design pattern?

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

Single Responsibility Principle(SRP)

Definition: The Single Responsibility Principle (SRP: Single Responsibility Principle) is also known as the Single Function Principle. The Single Responsibility Principle stipulates that there should never be more than one reason for a class to change. The so-called responsibility refers to the reason for the change of the class.

Principle: If a class takes on too many responsibilities, the responsibilities may be coupled to each other. A change in one responsibility may affect other responsibilities. To avoid this phenomenon, it is necessary to abide by the single responsibility principle as much as possible.

Core: The core of the single responsibility principle is the decoupling of responsibilities and the enhanced cohesion of classes.

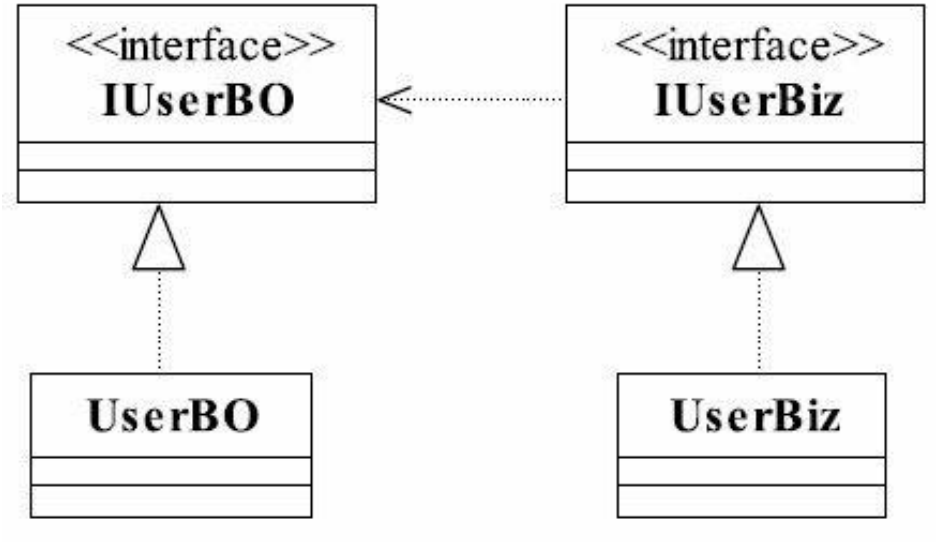
Necessity: The original intention of the single responsibility principle is to limit a class from taking too many responsibilities. If a class takes too many responsibilities, there will be at least the following disadvantages:

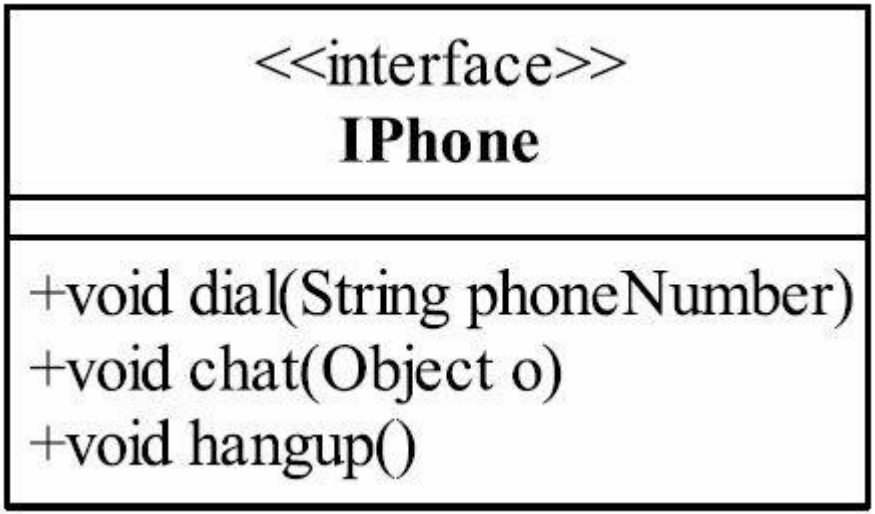
1. High degree of responsibility coupling. A class assumes multiple responsibilities, and these responsibilities are likely to be coupled together. A change in one responsibility may affect other responsibilities.
2. The reusability of the class is low. Classes with larger granularity have lower reusability.
3. The cost of invoking responsibility is high. When the client needs a certain responsibility of the object, it has to include all other unnecessary responsibilities, resulting in redundant code or waste of code.

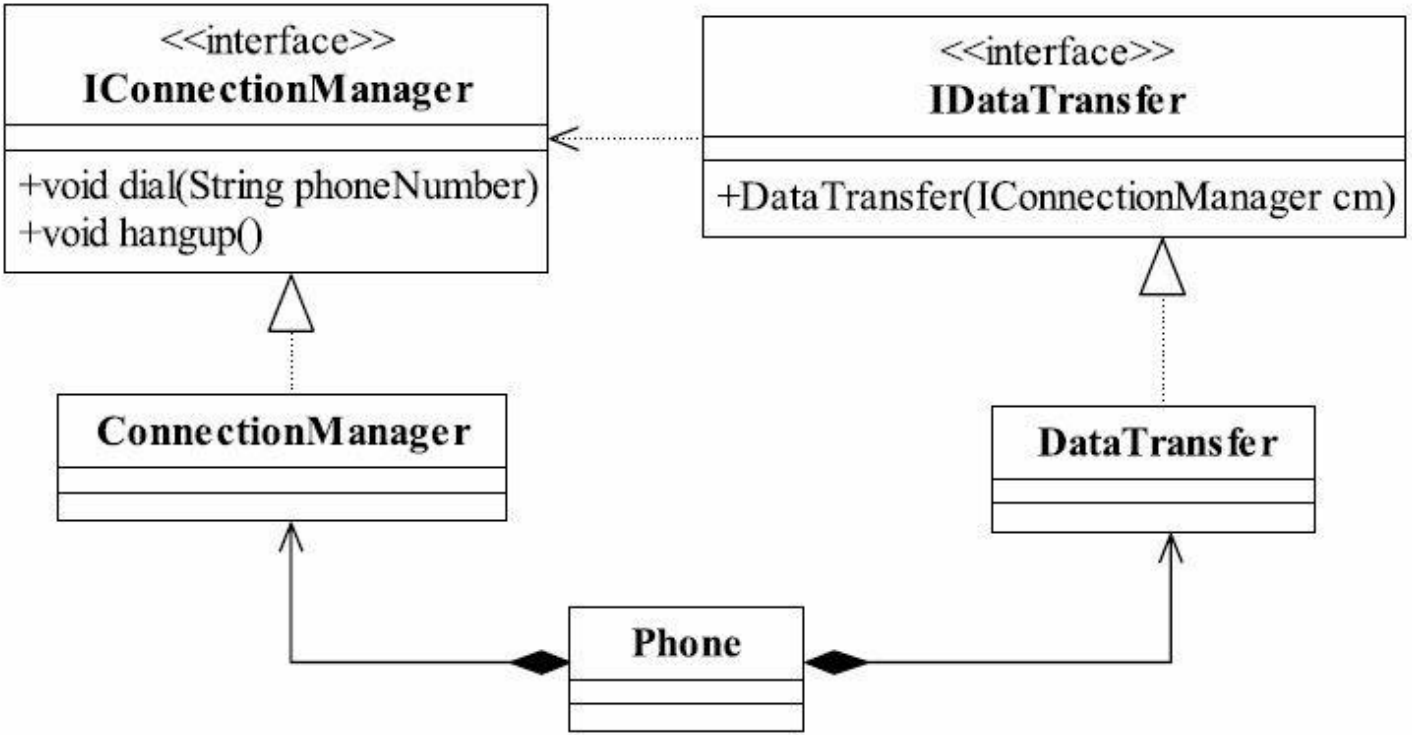
Function: Following the Single Responsibility Principle has the following roles in a software system:

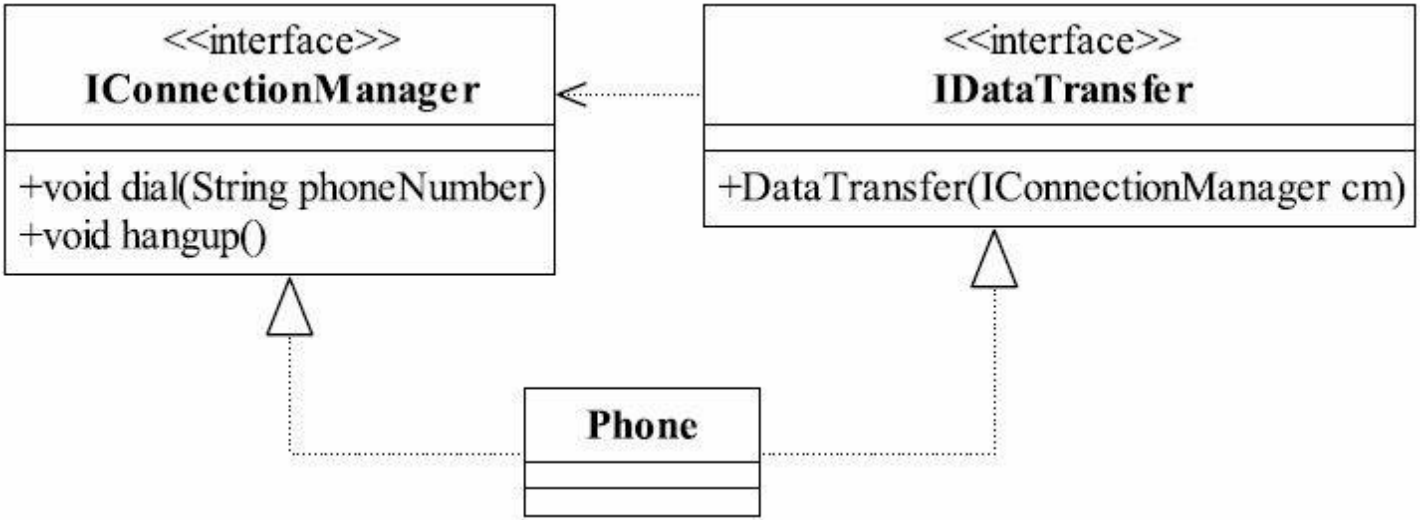
- Reduce class complexity. A class is responsible for only one responsibility, and its logic is definitely much simpler than that for multiple responsibilities.
- Improve class readability. With less complexity, naturally its readability increases.
- Improve system maintainability. The readability is improved, which is naturally easier to maintain.
- Reduce the risk of change. Changes are inevitable. If the single responsibility principle is followed well, when a function is modified, the impact on other functions can be significantly reduced.Improve class reusability.
- The lower the granularity of the class, the stronger the reusability.

Implementation method: The single responsibility principle is the simplest but most difficult principle to apply. It requires designers to discover and separate the different responsibilities of classes, and then encapsulate them into different classes or modules. The discovery of multiple responsibilities of classes requires designers to have strong analysis and design capabilities and relevant refactoring experience.


<<interface>>
IUserBO
<<interface>>
IUserBiz
UserBO
UserBiz


<<interface>>
IPhone
+void dial(String phoneNumber)
+void chat(Object o)
+void hangup()


<<interface>>
IConnectionManager
+void dial(String phoneNumber)
+void hangup()
<<interface>>
IDataTransfer
+DataTransfer(IConnectionManager cm)
ConnectionManager
DataTransfer
Phone


<<interface>>
IConnectionManager
+void dial(String phoneNumber)
+void hangup()
<<interface>>
IDataTransfer
+DataTransfer(IConnectionManager cm)
Phone

Open/Closed Principle

Definition: Software entities like classes,modules and functions should be open for extension but closed for modifications. It emphasizes the use of abstraction to build the framework and the implementation of extended details, which can improve the reusability and maintainability of the software system. The principle of opening and closing is the most basic design principle in object-oriented design, which guides us how to build a stable and flexible system.

Necessity: If a software system conforms to the principle of opening and closing, then from the perspective of software engineering, it has at least the following benefits:

- Good reusability. After the software is completed, we can still expand the software and add new functions, which is very flexible. Therefore, this software system can meet the changing needs by continuously adding new components.
- Good maintainability. Since the components of the existing software system, especially its abstract bottom layer, are not modified, we don't have to worry about the stability of the original components in the software system, which makes the changing software system have certain stability and continuity .

Function: The principle of opening and closing is the ultimate goal of object-oriented programming, which enables software entities to have certain adaptability and flexibility, as well as stability and continuity. Specifically, it works as follows:

- Simplified testing: If the software obeys the principle of opening and closing, only the extended code needs to be tested during software testing, because the original test code can still run normally.
- Improve reusability: The smaller the granularity, the greater the possibility of being reused; in object-oriented programming, programming according to atoms and abstractions can improve the reusability of code.
- Improve maintenance and scalability: Software that follows the principle of opening and closing has high stability and continuity, making it easy to expand and maintain.

Implementation method: The principle of opening and closing can be realized through "abstract constraints and encapsulation of changes", that is, define a relatively stable abstraction layer for software entities through interfaces or abstract classes, and encapsulate the same variable factors in the same concrete implementation class.

The Liskov Substitution Principle

Definition: Inheritance should ensure that any property proved about supertype objects also holds for subtype objects.

The Liskov substitution principle mainly expounds some principles about inheritance, that is, when inheritance should be used, when inheritance should not be used, and the principles contained in it. Liskov substitution is the basis of inheritance reuse, it reflects the relationship between the base class and the subclass, it is a supplement to the principle of opening and closing, and it is a specification for the concrete steps to realize abstraction.

Function:

- The Liskov substitution principle is one of the important ways to realize the opening and closing principle.
- It overcomes the disadvantage of poor reusability caused by rewriting the parent class in inheritance.
- It is the guarantee of the correctness of the action. That is, the extension of the class will not introduce new errors to the existing system, reducing the possibility of code errors.
- Strengthen the robustness of the program, and at the same time achieve very good compatibility when changing, improve the maintainability and scalability of the program, and reduce the risks introduced when the requirements change.

Implementation method: Generally speaking, the Liskov substitution principle is: subclasses can extend the functions of the parent class, but cannot change the original functions of the parent class. That is to say: When a subclass inherits the parent class, in addition to adding new methods to complete the newIn addition to adding functions, try not to override the methods of the parent class. If the new function is completed by rewriting the method of the parent class, although it is simple to write, the reusability of the entire inheritance system will be greatly reduced.It is relatively poor, especially when polymorphism is used frequently, the probability of program running errors will be very high. If the program violates the Liskov substitution principle, the objects of the inherited class are placed where the base class appearsA run error will occur. At this time, the correction method is: cancel the original inheritance relationship and redesign the relationship between them.

Interface Segregation Principle

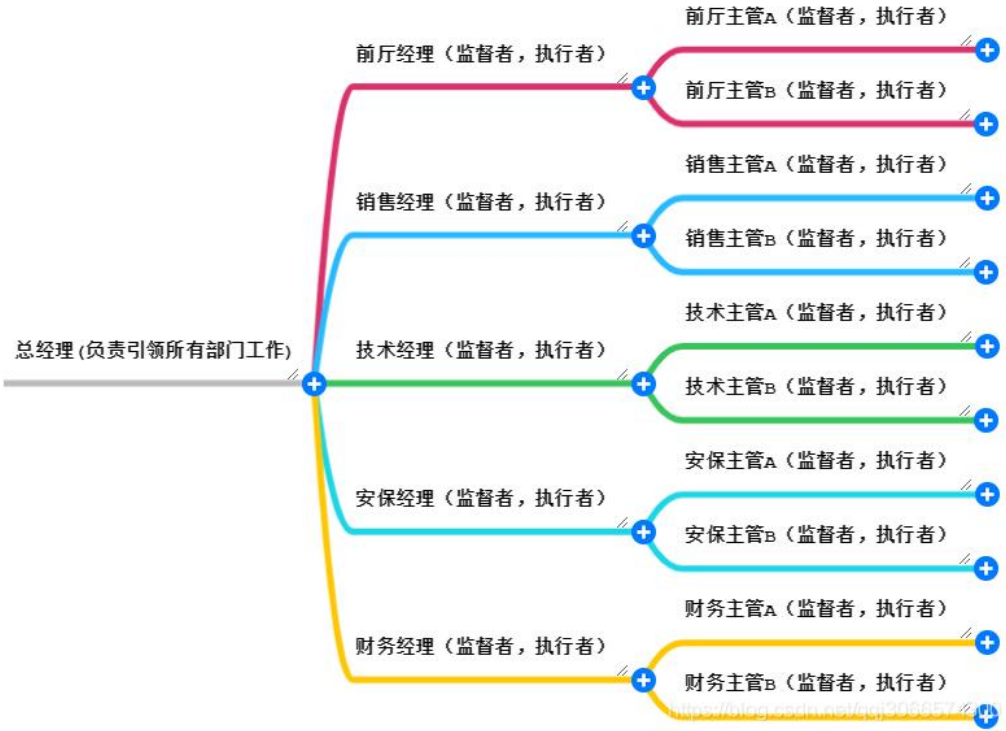
Definition: Clients should not be forced to depend upon interfaces that they do not use. The principle of interface isolation and single responsibility are both to improve the progress within the class and reduce the coupling between them, which embodies the idea of encapsulation.

Necessity:

- The complex and huge interface is decomposed into many small interfaces, thereby reducing the dependence of classes on interfaces and improving the flexibility and maintainability of the system.
- Interface isolation improves the cohesion of the system, reduces external interaction, and reduces the coupling of the system.
- Whether the granularity of the interface is defined reasonably determines the stability of the system. If the granularity is too large, the program will be complicated, and if it is too large, the flexibility will be reduced.

Implementation method:

1. An interface only serves one submodule or business logic.
2. Develop services for dependent interface classes, provide methods that callers need, and block unnecessary methods.
3. Improve cohesion and reduce external interaction.


总经理 (负责引领所有部门工作)
前厅经理（监督者，执行者）
前厅主管A（监督者，执行者）
前厅主管B（监督者，执行者）
销售经理（监督者，执行者）
销售主管A（监督者，执行者）
销售主管B（监督者，执行者）
技术经理（监督者，执行者）
技术主管A（监督者，执行者）
技术主管B（监督者，执行者）
安保经理（监督者，执行者）
安保主管A（监督者，执行者）
安保主管B（监督者，执行者）
财务经理（监督者，执行者）
财务主管A（监督者，执行者）
财务主管B（监督者，执行者）

Dependency Inversion

Definition: High level modules should depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions. The Dependence Inversion Principle (Dependence Inversion Principle) is that the program depends on the abstract interface, not on the concrete implementation. Simply put, it is required to program the abstraction, not the implementation, which reduces the coupling between the client and the implementation module.

Principle: If a class takes on too many responsibilities, the responsibilities may be coupled to each other. A change in one responsibility may affect other responsibilities. To avoid this phenomenon, it is necessary to abide by the single responsibility principle as much as possible.

Core: The core of the single responsibility principle is the decoupling of responsibilities and the enhanced cohesion of classes.

Necessity: In process-oriented development, the upper layer calls the lower layer, and the upper layer depends on the lower layer. When the lower layer changes drastically, the upper layer also changes accordingly, which will reduce the reusability of modules and greatly increase the cost of development.

Function:

Object-oriented development solves this problem very well. Generally, the probability of abstract change is very small, so that user programs depend on abstraction, and implementation details also depend on abstraction. Even if the implementation details are constantly changing, as long as the abstraction remains the same, the client program does not need to change. This greatly reduces the coupling between the client program and the implementation details.

Implementation method:

1. Pass dependent objects through constructorsFor example, the parameters that need to be passed in the constructor are implemented in the form of abstract classes or interfaces.
2. Pass the dependent object through the setter methodThat is, the parameters in the setXXX method we set are abstract classes or interfaces to realize the transfer of dependent objects
3. Interface declaration implements dependent objects

Law of Demeter (LoD)

Definition: Talk only to your immediate friends. Also known as the principle of least knowledge, that is to say, an object should know as little as possible about other objects. Do not talk to strangers. Dimit's law can reduce the coupling degree of the system and keep the loose coupling state between classes.

Principle: Demeter's law does not want to establish a direct connection between classes. If there is really a need to establish a connection, I hope it can be conveyed through its friend class. Therefore, one of the consequences that may be caused by the application of Dimit's law is that there are a large number of intermediary classes in the system. the complexity.

Core: When one of the object changes, it will affect other software entities as little as possible.

Friends:

1) The current object itself (this)
2) The object passed in as a parameter to the current object method. The method input parameter is an object, which is a friend of this object and the current class
3) The object directly referenced by the instance variable of the current object Define a class, and the properties in it refer to other objects, then the instance of this object and the current instance are friends
4) If the instance variable of the current object is an aggregation, then the elements in the aggregation are also friends If the property is an object, then the property and the elements in the object are friends
5) 5) The object created by the current object

Function: There are some well-known abstractions in object-oriented programming, such as encapsulation, cohesion, and coupling, that can theoretically be used to generate clean designs and good code. While these are very important concepts, they are not practical enough to be used directly in a development environment, and these concepts are relatively subjective and very dependent on the experience and knowledge of the user. The same is true for other concepts like Single Responsibility Principle, Open Closed Principle etc. What makes Demeter's Law unique is its succinct and precise definition, which allows for direct application when writing code, almost automatically applying appropriate encapsulation, low cohesion, and loose coupling.

Implementation: