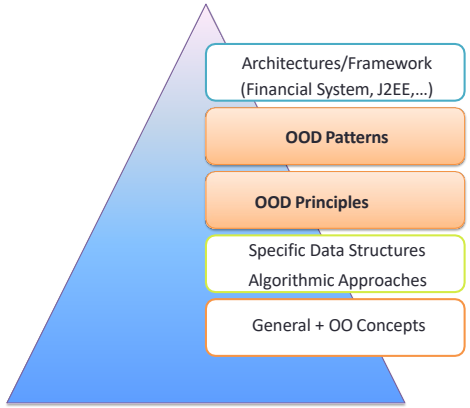
ITSS SOFTWARE DEVELOPMENT

11. DESIGN PRINCIPLES

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

1

Review: Design levels



2

S.O.L.I.D Principles of OOD

- SRP: The Single Responsibility Principle
- OCP: The Open Closed Principle
- LSP: The Liskov Substitution Principle
- ISP: The Interface Segregation Principle
- DIP: The Dependency Inversion Principle

3

SOLID Principles – Why necessary?

- S.O.L.I.D. is useful as reference while designing applications
- Managing dependencies makes maintainability easier
- Various principles and techniques available for diagnose problems with designs

4

5

Content

1. **SOLID: Single Responsibility Principle**
2. SOLID: Open-Close Principle
3. SOLID: Liskov Substitution Principle
4. SOLID: Interface Segregation Principle
5. SOLID: Dependency Inversion Principle
6. Case study: Reminder program

5

6

Review: Tight/High Cohesion

- Cohesion
  - Measure of how **strongly-related** or **focused the responsibilities** of a single module are
  - The degree to which all elements of a module are directed towards **a single task/ responsibility**
- Highest cohesion (Cohesive)
  - Modules have a **single and clear responsibility**
  - **Elements** in a module are **functionally related**

6

7

SRP: Single Responsibility Principle

There should **never** be **more than one reason** for a class to **change**

Or

A class should have **one, and only one type** of **responsibility**

7

8

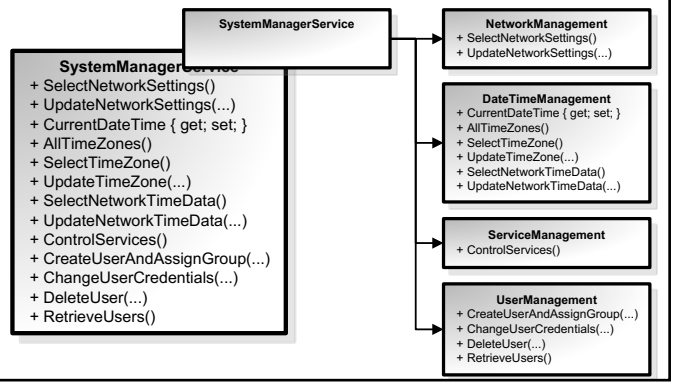
SRP: Single Responsibility Principle

- Simplest principle, hardest to get right
- Every module should do just one thing and do it well
- Finding and separating responsibilities may be hard to do
  - How much is enough?
  - Measured by Cohesion (and Coupling)
- When violated: Fragile design that breaks in unexpected ways when changed

8

SRP: Single Responsibility Principle

- How much is enough for "one" responsibility?



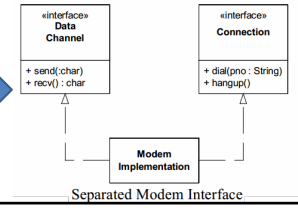
9

SRP: Single Responsibility Principle

- What is a Responsibility?
  - A reason for change
- "Modem" sample
  - dial & hangup functions for managing connection
  - send & recv functions for data communication

➔ Should separate into 2 modules!

```
Modem.java -- SRP Violation
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```



10

11

SRP: Example

- Often we need to sort students by their name, or ssn.

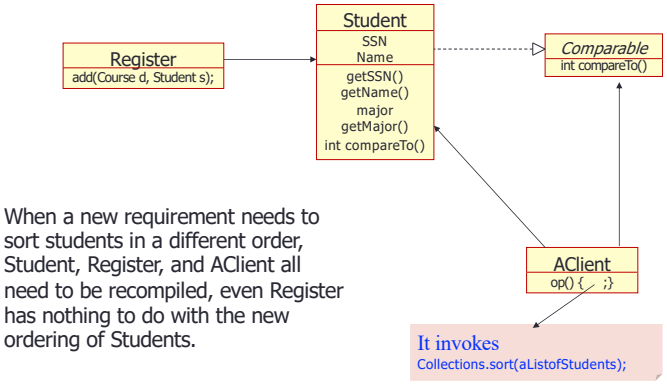
➔ Make Class Student implement the Java Comparable interface

```
class Student implements Comparable{
    …
    int compareTo(Object o){ … }
    …
};
```

12

13

SRP: Example (2)



When a new requirement needs to sort students in a different order, Student, Register, and AClient all need to be recompiled, even Register has nothing to do with the new ordering of Students.

13

14

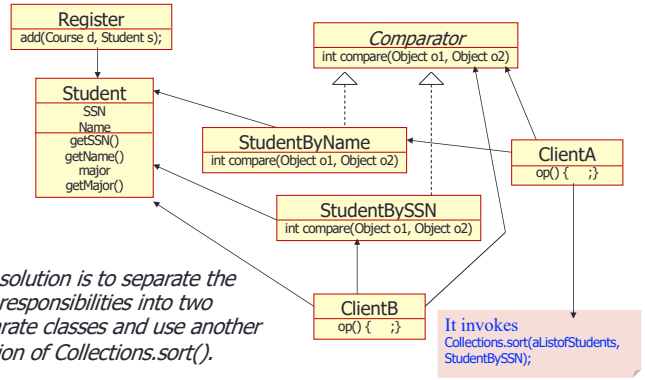
Java Comparable

- Each time students are ordered differently, the **`compareTo()`** method needs to be changed and **`Students`** needs to be recompiled and also its clients
- Student is a business entity, it does not know in what order it should be sorted since the order of sorting is imposed by the client of **`Student`**.
- Cause of the problem: we bundled two separate responsibilities (i.e., a business entity and ordering) into one class – a violation of SRP

14

15

The single-responsibility principle



The solution is to separate the two responsibilities into two separate classes and use another version of Collections.sort().

15

16

Content

1. **SOLID: Single Responsibility Principle**
2. SOLID: Open-Close Principle
3. SOLID: Liskov Substitution Principle
4. SOLID: Interface Segregation Principle
5. SOLID: Dependency Inversion Principle
6. Case study: Reminder program

16

17

OCP: Open-Close Principle

Open chest surgery is not needed when putting on a coat

17

18

OCP: Open Closed Principle

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"

Bertrand Meyer, 1988

Or

"You should be able to extend a classes behavior, without modifying code"

➔ Extend behavior instead of changing old code that already works

18

19

OCP: Open Closed Principle

- Open for Extension
  - The behavior of the module/class can be extended
  - The module behaves in new and different ways as the requirements changes, or to meet the needs of new applications

→ Reusability and maintainability

- Closed for Modification
  - The source code of an existing module is inviolate
  - No one is allowed to make source code changes to it

→ When violated: Cascading changes to dependent modules during changes

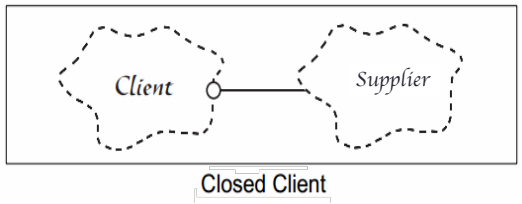
19

20

OCP: Abstraction is the KEY

- Client & Supplier classes are concrete
- If the Supplier implementation/class is changed, Client also needs change.

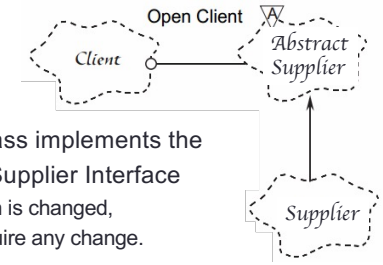
➔ How to resolve this problem?



20

21

OCP: Abstraction is the KEY (2)



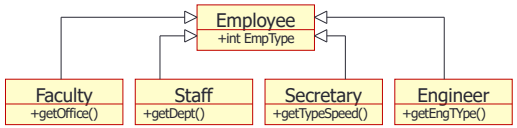
- The Concrete Supplier class implements the Abstract Supplier class / Supplier Interface
  - The Supplier implementation is changed,
  - the Client is likely not to require any change.

➔ The Abstract Supplier class here is closed for modification and the Concrete class implementations here are Open for extension

21

22

Violation of the OCP



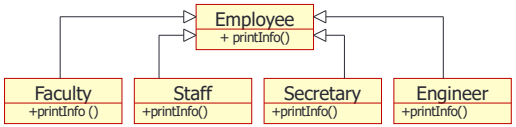

```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        if (emps[i].empType == FACULTY)
            printfFaculty((Faculty)emps[i]);
        else if (emps[i].empType ==STAFF)
            printStaff((Staff)emps[i]);
        else if (emps[i].empType == SECRETARY)
            printSecretary((Secretary)emps[i]);
    }
}
```

What if we need to add Engineer??

22

23

OCP – Dynamic Polymorphism




```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        emps[i].printInfo();
    }
}
```

When Engineer is added, printEmpRoster() does not even need to recompile.

➔ printEmpRoster() is open to extension, closed for modification.

23

24

OCP – Dynamic Polymorphism

- Three versions of SORT
  - **`sort(List list)`**
    - Elements of list must implement Comparable interface
  - **`sort(List list, StringComparator sc)`**
    - Elements of list are not required to implement Comparable
    - StringComparator orders objects of String only
  - **`sort(List list, Comparator comp)`**
    - Elements of list are not required to implement Comparable
    - Comparator may compare objects of any type.
    - Open to extension since it can sort objects of any type at any order specified in the second parameter.

24

OCP – Static Polymorphism

- Generics in Java: The container can take objects of any type

```
public class LinkedList<E> {
    boolean add(E e);
    boolean add(int index, E e);
    boolean addFirst(E e);
    boolean addLast(E e);
    void clear();
    E get(int index);
    ...
}
```

25

Practice: Checking S & O in codebase

- Checking if all classes donot follow the SRP & OCP in code base

26

Content

1. **SOLID: Single Responsibility Principle**
2. SOLID: Open-Close Principle
3. SOLID: Liskov Substitution Principle
4. SOLID: Interface Segregation Principle
5. SOLID: Dependency Inversion Principle
6. Case study: Reminder program

27

LSP: Liskov Substitution Principle

- *"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."*
- Or

*"Subclasses should be **substitutable** for their base classes."*

- Every implementation of an interface needs to fully comply with the requirements of this interface
- Any algorithm that works on the interface, should continue to work for any substitute implementation

28

29

LSP: Liskov Substitution Principle (2)

- Demand no more: The subclass would accept any arguments that the superclass would accept.
- Promise no less: Any assumption that is valid when the superclass is used must be valid when the subclass is used.

- Interface Inheritance: LSP should be conformed to
- Implementation (or Class) Inheritance:
  - Multi-layer inheritance hierarchy: Provide more abstraction levels for subclasses
  - Use composition instead of inheritance (in Java) or use private base classes (in C++)

29

30

LSP: Liskov Substitution Principle (3)

- Why LSP is so important? If violates LSP:
  - Class hierarchy would be a **mess** and if subclass instance was passed as parameter to methods method, strange behavior might occur.
  - Unit tests for the Base classes would never succeed for the subclass.

➔ LSP is just an extension of Open-Close Principle!!!

30

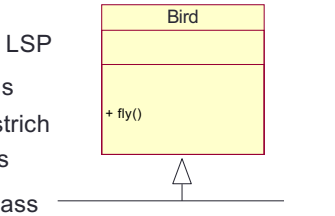
31

LSP: An Example

- Ostrich is a Bird (definitely!!!)
- Can it fly? No! => Violates the LSP

➔Even if in real world this seems natural, in the class design, Ostrich should not inherit the Bird class

➔There should be a separate class for birds that can't really fly and Ostrich inherits that

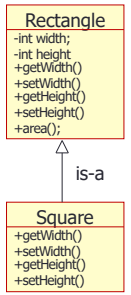


31

32

LSP: More example

```
class Square extends Rectangle {
   public void setWidth(int width) {
      super.setWidth(width);
      super.setHeight(width);
   }
   public void setHeight(int height) {
      super.setHeight(height);
      super.setWidth(height);
   }
}
void clientOfRectangle(Rectangle r) {
   r.setWidth(10);
   r.setHeight(20);
   print(r.area());
}
Rectangle r = new Square(…);
clientOfRectangle(r); // what would be printed?
```

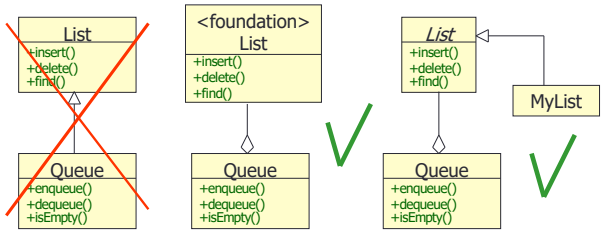


32

33

LSP: Implementation Inheritance

- Implementation inheritance
  - When you use List to implement Queue (in Java), use composition, not inheritance.
  - The intention is that you use only List's implementation



33

34

Content

1. **SOLID: Single Responsibility Principle**
2. SOLID: Open-Close Principle
3. SOLID: Liskov Substitution Principle
4. SOLID: Interface Segregation Principle
5. SOLID: Dependency Inversion Principle
6. Case study: Reminder program

34

35

ISP: Interface Segregation Principle

- *"Client should not be forced to depend upon interface that they do not use."*

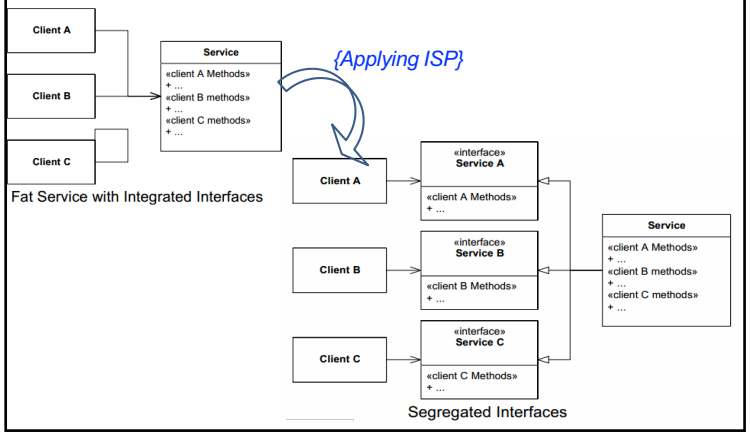
Or

- *"Many client specific interfaces are better than one general purpose interface."*

35

36

ISP: Interface Segregation Principle (2)

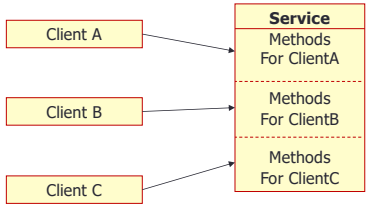


36

37

ISP – A Bad Design

Fat Service with integrated interface

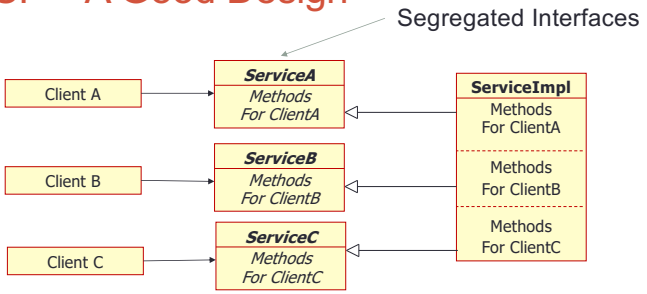


The problem is that any change in Service introduced by one client (e.g., Client A) causes Service and the other clients (Client A and Client C) to be recompiled.

37

38

ISP – A Good Design



- Changes introduced by one client (e.g., Client A) may require its designated interface (ServiceA) and the service implementation to change.
- However, other clients (Client A & Client C) and their Interfaces (ServiceB & ServiceC) are not required to change or recompile.

38

39

ISP: Interface Segregation Principle (3)

- Interfaces with too many methods (fat interfaces) are less re-usable
- Unnecessary complexity and reduces maintainability or robustness in the system
- Unnecessary coupling among the clients due to additional useless methods
  - When one client causes the interface to change, all other clients are forced to recompile

➔ Interfaces have their own responsibility and thus they are re-usable.

39

40

Content

1. **SOLID: Single Responsibility Principle**
2. SOLID: Open-Close Principle
3. SOLID: Liskov Substitution Principle
4. SOLID: Interface Segregation Principle
5. SOLID: Dependency Inversion Principle
6. Case study: Reminder program

40

41

DIP: Dependency Inversion Principle

"High level modules should not depend upon low level modules. Both should depend upon abstractions"

Or

"Abstractions should not depend upon details. Details should depend upon abstraction."

Or

"Depend upon Abstractions. Do not depend upon concretions."

41

42

42

43

DIP: An Example

43

44

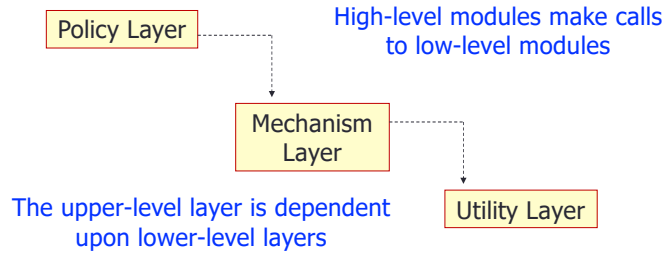
DIP: Dependency Inversion Principle

- Low-level modules are more likely to change
- High-level modules (business policies) are more likely to remain stable (i.e. purpose of the system) → They should:
  - Depend upon abstraction of low level modules
  - Force low level modules to change
- When violated: Lower level module changes can force high level modules to change

44

45

DIP: Dependency Inversion Principle

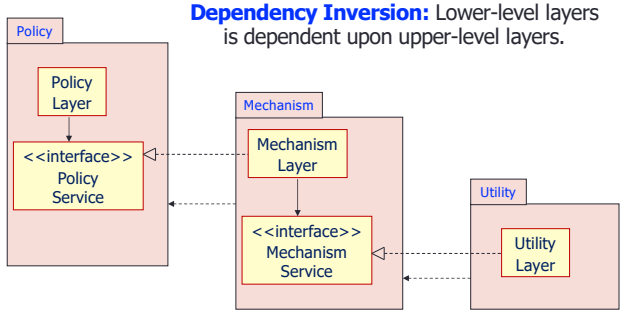


➔ *Depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes*

45

46

DIP: Dependency Inversion Principle



Dependency Inversion: Lower-level layers is dependent upon upper-level layers.

Ownership Inversion: The client (upper-level layer) owns the interface, not the lower-level layers

46

47

DIP and OCP

- High-level concepts are more stable than low-level implementation
- Concrete things change often and abstract things change less often.
- Abstractions are “hinge points”, places when the design can provide different implementations

47

48

Practice: Checking L ,I & D in codebase

- Checking if all classes donot follow the LSP, ISP & DIP in codebase

48

Content

1. **SOLID: Single Responsibility Principle**
2. SOLID: Open-Close Principle
3. SOLID: Liskov Substitution Principle
4. SOLID: Interface Segregation Principle
5. SOLID: Dependency Inversion Principle
6. Case study: Reminder program

49

Chương trình "Nhắc nhở nghỉ giải lao"

- Bạn hãy thiết kế và viết mã nguồn minh hoạ cho chương trình "Nhắc nhở nghỉ giải lao" như sau:
- Chương trình định kỳ kiểm tra thời gian
- Cứ sau 1 khoảng thời gian nhất định, chương trình:
  - Nhắc hoặc hoặc yêu cầu (bắt buộc) người dùng cần nghỉ ngơi. Thậm chí, chương trình có thể không cho người dùng sử dụng máy tính (bắc buộc nghỉ ngơi) trong vòng 1 số phút nghỉ ngơi đó.
  - Gợi ý làm 1 việc gì đó trong thời gian nghỉ ngơi:
    - Nghe nhạc
    - Tập thể dục tại chỗ theo hướng dẫn
    - ...

50

Design exercise

- Write a typing break reminder program
  - Offer the hard-working user occasional reminders of the health issues, and encourage the user to take a break from typing
- Naive design
  - Make a method to display messages and offer exercises
  - Make a loop to call that method from time to time

(Let's ignore multi-threaded solutions for this discussion)

51