ITSS SOFTWARE DEVELOPMENT
11. DESIGN PRINCIPLES

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

Review: Design levels

- Architectures/Framework (Financial System, J2EE,…)
- OOD Patterns
- OOD Principles
- Specific Data Structures Algorithmic Approaches
- General + OO Concepts
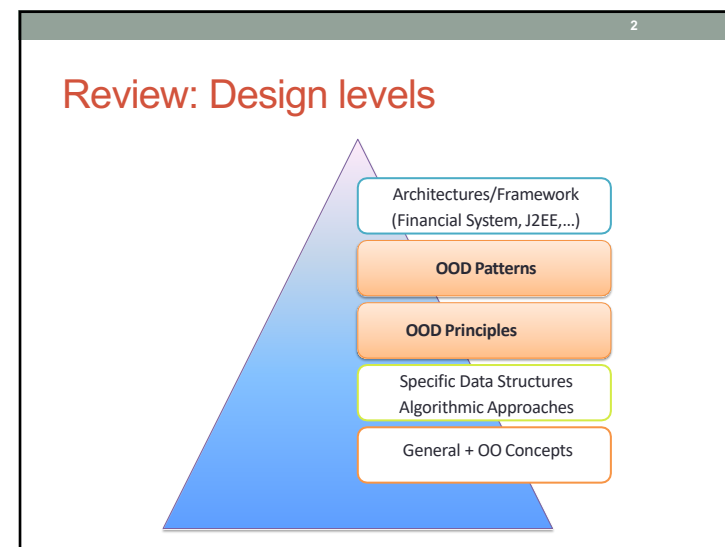
S.O.L.I.D Principles of OOD

- 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

Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

Principles of OO Class Design
SRP: The 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."

5

Principles of OO Class Design
SRP: The Single Responsibility Principle (cont)

- Two applications are using this Rectangle class:
  - Computational Geometry Application uses this class to calculate the Area
  - Graphical Application uses this class to draw a Rectangle in the UI
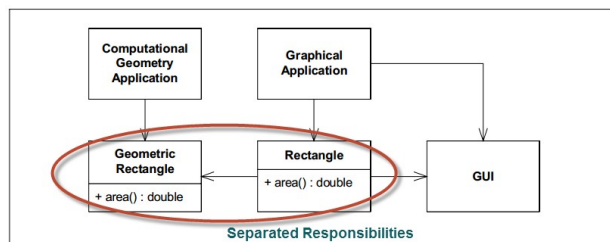

The **Rectangle** class has two responsibilities
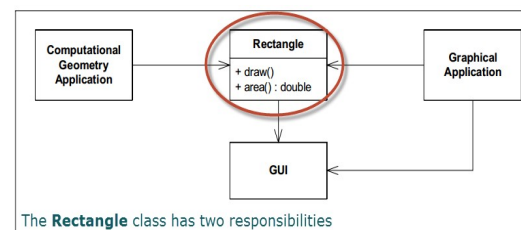
6

Principles of OO Class Design
SRP: The Single Responsibility Principle (cont)

- ❑ A better design is to separate the two responsibilities into two completely different classes

Separated Responsibilities

- ❑ Why is it important to separate these two responsibilities into separate classes?

7

Principles of OO Class Design
SRP: The Single Responsibility Principle (cont)

- 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 repositories!

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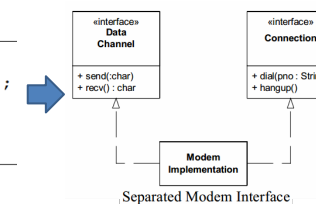

Separated Modem Interface

8

2

Content

9

Principles of OO Class Design
OCP: The 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"

- "Open for Extension"
  - The behavior of the module/class can be extended
  - The module behave in new and different ways as the requirements changes, or to meet the needs of new aplications
- "Closed for Modification"
  - The source code of such a module is inviolate
  - No one is allowed to make source code changes to it

10

Principles of OO Class Design
OCP: The Open Closed Principle (cont)

- Client & Supplier classes are concrete
- If the Supplier implementation/class is changed, Client also needs change.
- ➔ How to resolve this problem?

Closed Client
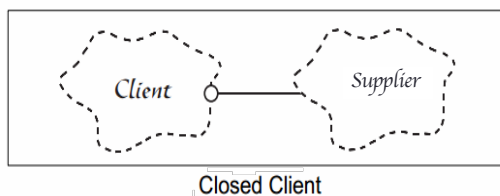
11

Principles of OO Class Design
OCP: The Open Closed Principle (cont)

- Change to support Open-Closed Principle.
- ➔ Abstraction is the key.
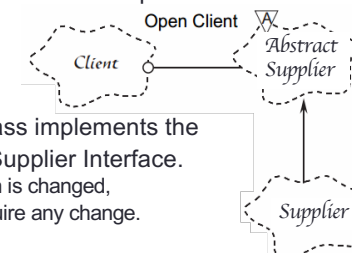
- The Concrete Supplier class implements the Abstract Supplier class / Supplier Interface.
  - The Supplier implementation is changed,
  - the Client is likely not to require any change.

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

21

12

3

Content

13

Principles of OO Class Design
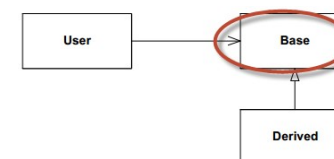## LSP: The Liskov Substitution Principle

- *"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."

User, Based, Derived, example.
```
void User(Base& b);

Derived d;
User(d);
```
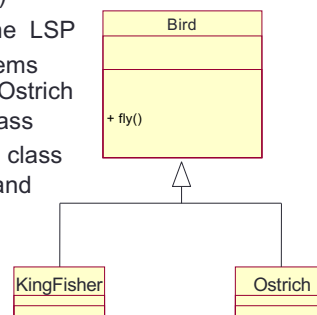

22

14

Principles of OO Class Design
LSP: The Liskov Substitution Principle (cont)

- 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.

15

Principles of OO Class Design
LSP: The Liskov Substitution Principle (cont)

- "Inheritance" ~ "is a" relationship
  - But, easy to get carried away and end up in wrong design with bad inheritance.
  - ➔ The LSP is a way of ensuring that inheritance is used correctly
- Why The LSP is so important? If not 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!!!

16

4

Content

17

Principles of OO Class Design
ISP: The 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."*
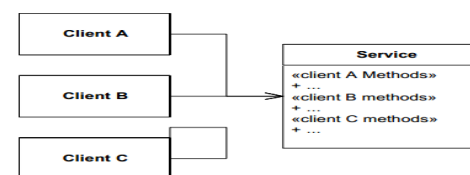
Fat Service with Integrated Interfaces

18

19

Principles of OO Class Design
ISP: The Interface Segregation Principle

{Applying ISP}

Fat Service with Integrated Interfaces

Segregated Interfaces

19

20
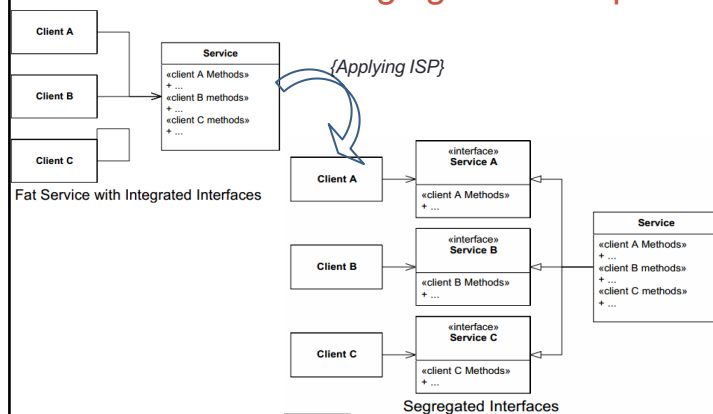
Principles of OO Class Design
ISP: The Interface Segregation Principle (cont.)

- Interfaces with too many methods are less re-usable.
- Such "fat interfaces" with additional useless methods lead to inadvertent coupling between classes.
- Doing this also introduce unnecessary complexity and reduces maintainability or robustness in the system.

→ The ISP ensures that, Interfaces are developed so that, each of them have their own responsibility and thus they are re-usable.

20

5

Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

21

Principles of OO Class Design
DIP: The 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."

22

Principles of OO Class Design
DIP: The Dependency Inversion Principle

- Strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes.
  - A well designed object-oriented application.
  - E.g. Layers of application

Simple Layers

Policy Layer → Mechanism Layer → Utility Layer

Abstract Layers

Policy Layer → Mechanism Interface *Abstract* ← Mechanism Layer → Utility Interface *Abstract* ← Utility Layer
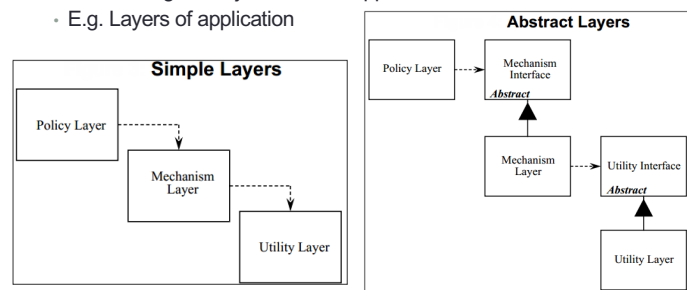
23

Content

1. S: The Single Responsibility Principle
2. O: The Open Closed Principle
3. L: The Liskov Substitution Principle
4. I: The Interface Segregation Principle
5. D: The Dependency Inversion Principle
6. Case study: Reminder program

24

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)*

25

TimeToStretch suggests exercises

```java
public class TimeToStretch {
  public void run() {
    System.out.println("Stop typing!");
    suggestExercise();
  }
  public void suggestExercise() {
  ...
  }
}
```

26

Timer calls run() periodically

```java
public class Timer {
  private TimeToStretch tts = new TimeToStretch();
  public void start() {
    while (true) {
      ...
      if (enoughTimeHasPassed) {
        tts.run();
      }
      ...
    }
  }
}
```
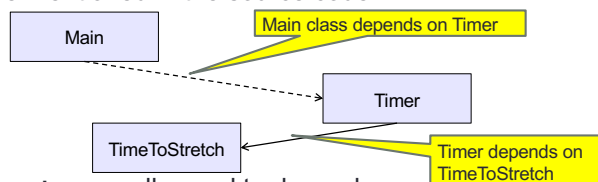
27

Main class puts it together

```java
class Main {
  public static void main(String[] args) {
    Timer t = new Timer();
    t.start();
  }
}
```

28

Module dependency diagram

- An arrow in a module dependency diagram indicates "depends on" or "knows about" – simplistically, "any name mentioned in the source code"

Main

Main class depends on Timer

Timer

TimeToStretch

Timer depends on TimeToStretch

- Does **Timer** really need to depend on **TimeToStretch**?
- Is Timer re-usable in a new context?

29

Decoupling

- **Timer** needs to call the **run** method
  - **Timer** doesn't need to know what the **run** method does
- Weaken the dependency of **Timer** on **TimeToStretch**
- Introduce a weaker specification, in the form of an interface or abstract class

```
public abstract class TimerTask {
    public abstract void run();
}
```

- **Timer** only needs to know that something (e.g., **TimeToStretch**) meets the **TimerTask** specification

30

TimeToStretch (version 2)

```
public class TimeToStretch extends TimerTask {
  public void run() {
    System.out.println("Stop typing!");
    suggestExercise();
  }

  public void suggestExercise() {
    ...
  }
}
```

31

Timer v2

```
public class Timer {
  private TimerTask task;
  public Timer(TimerTask task) { this.task = task; }
  public void setTask(TimerTask task){this.task = task;}
  public void start() {
    while (true) {
      ...
      if (enoughTime)
          task.run();
    }
  }
}
```
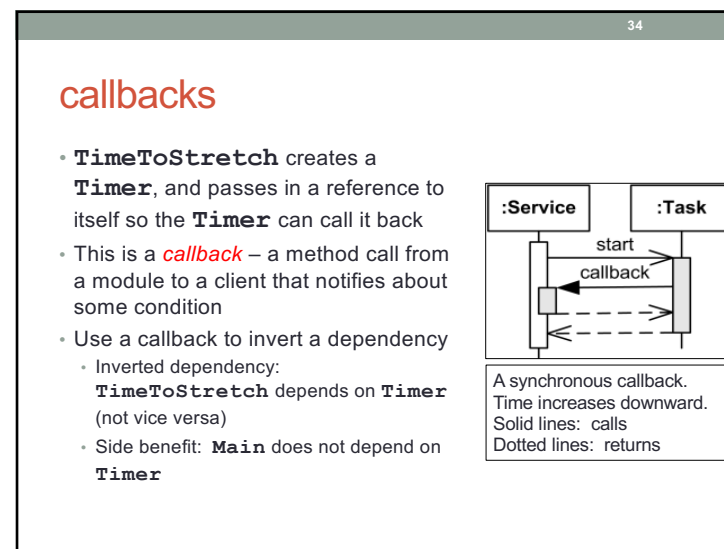- Main creates the **TimeToStretch** object and passes it to **Timer**

```
Timer t = new Timer(new TimeToStretch());
t.start();
t.setTask(new TimeToSave());
t.start();
```
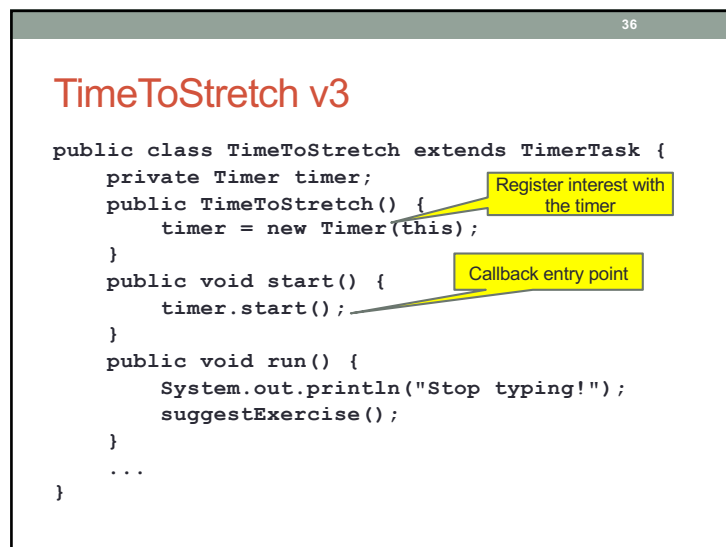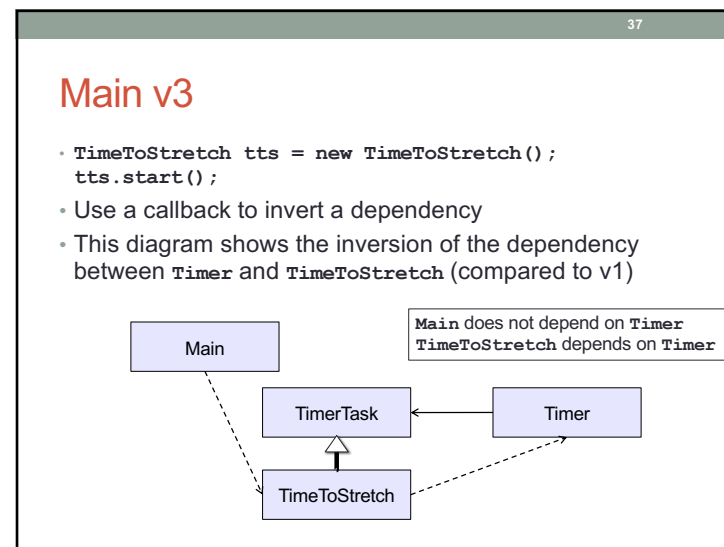
32

33

Module dependency diagram

- **Main** still depends on **Timer** (is this necessary?)
- **Main** depends on the constructor for **TimeToStretch**
- **Timer** depends on **TimerTask**, not **TimeToStretch**
  - Unaffected by implementation details of **TimeToStretch**
  - Now **Timer** is much easier to reuse

- -→ Dependence
- Subclassing

33

34

callbacks

- **TimeToStretch** creates a **Timer**, and passes in a reference to itself so the **Timer** can call it back
- This is a *callback* – a method call from a module to a client that notifies about some condition
- Use a callback to invert a dependency
  - Inverted dependency: **TimeToStretch** depends on **Timer** (not vice versa)
  - Side benefit: **Main** does not depend on **Timer**

A synchronous callback.
Time increases downward.
Solid lines: calls
Dotted lines: returns

34

36

TimeToStretch v3

```
public class TimeToStretch extends TimerTask {
    private Timer timer;
    public TimeToStretch() {
        timer = new Timer(this);      // Register interest with the timer
    }
    public void start() {
        timer.start();                // Callback entry point
    }
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
    ...
}
```

36

37

Main v3

- **TimeToStretch tts = new TimeToStretch();**
  tts.start();
- Use a callback to invert a dependency
- This diagram shows the inversion of the dependency between **Timer** and **TimeToStretch** (compared to v1)

Main does not depend on **Timer**
TimeToStretch depends on **Timer**

37

9

How do we design classes?

- One common approach to class identification is to consider the specifications
- In particular, it is often the case that
  - *nouns* are potential classes, objects, fields
  - *verbs* are potential methods or responsibilities of a class

38

Design exercise

- Suppose we are writing a birthday-reminder application that tracks a set of people and their birthdays, providing reminders of whose birthdays are on a given day

- What classes are we likely to want to have? Why?

Class shout-out about classes

39

More detail for those classes

- What fields do they have?
- What constructors do they have?
- What methods do they provide?
- What invariants should we guarantee?

In small groups, ~5 minutes

40