CS2030 Programming Methodology II

Lecture V

Shengdong Zhao
Spring 2019

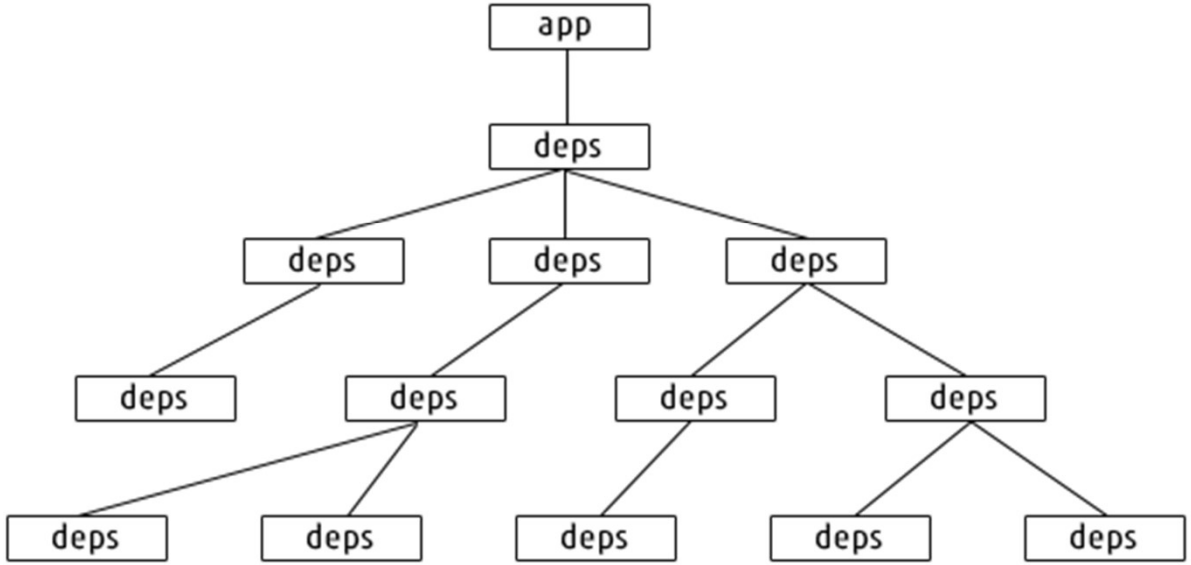
Acknowledge: slides are adapted from Henry Chia

Lecture Outline

- **Poor code & Dependency problems**
- **SOLID Design Principles**
  - **S**ingle Responsibility Principle
  - **O**pen–Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle
- **Programming to an interface**


HOLLYWOOD
PRODUCTION
DIRECTOR
CAMERA
DATE
SCENE
TAKE
Canon

Coupling & Dependency



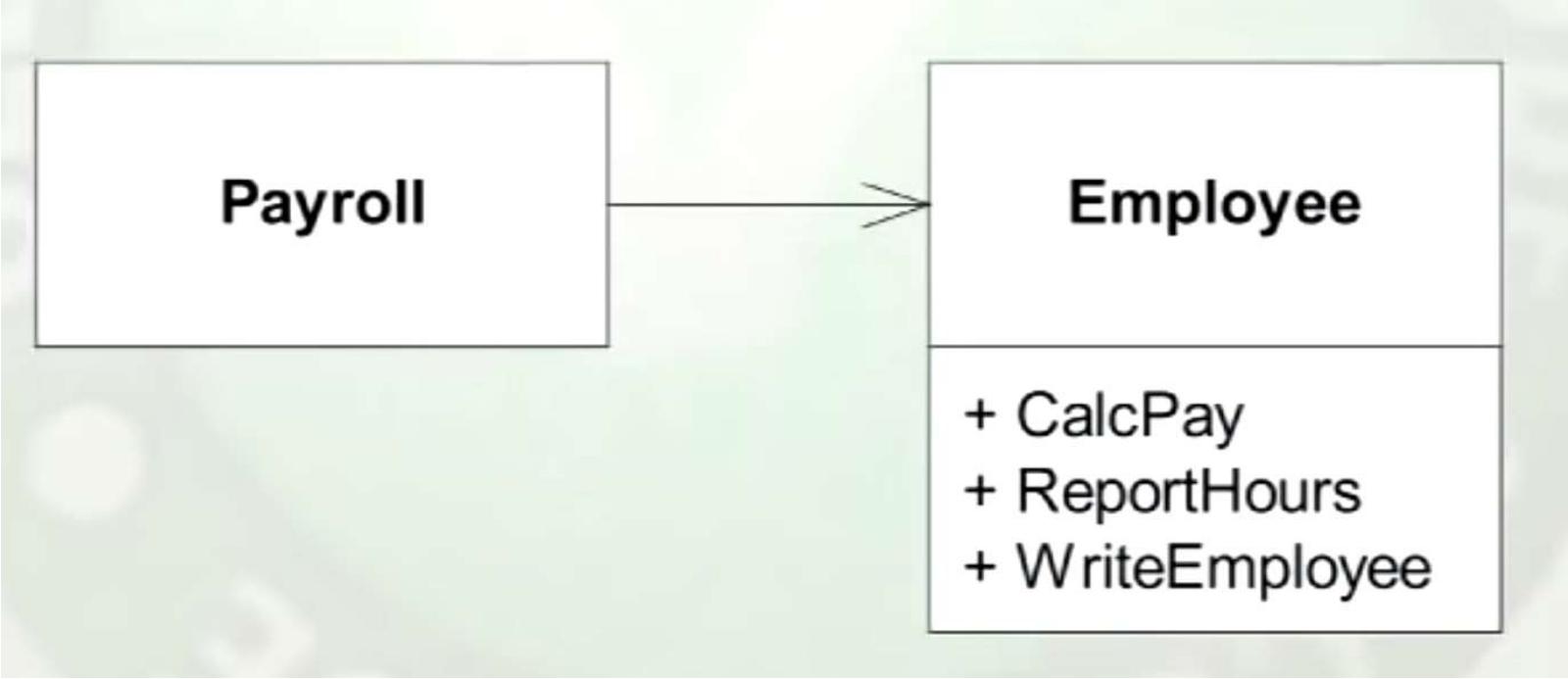
Key advantage/skill of OOP is managing dependencies

S.O.L.I.D Principles

- **SOLID Design Principles**
  - **S**ingle Responsibility Principle
  - **O**pen–Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle

Single Responsibility Principle (SRP)

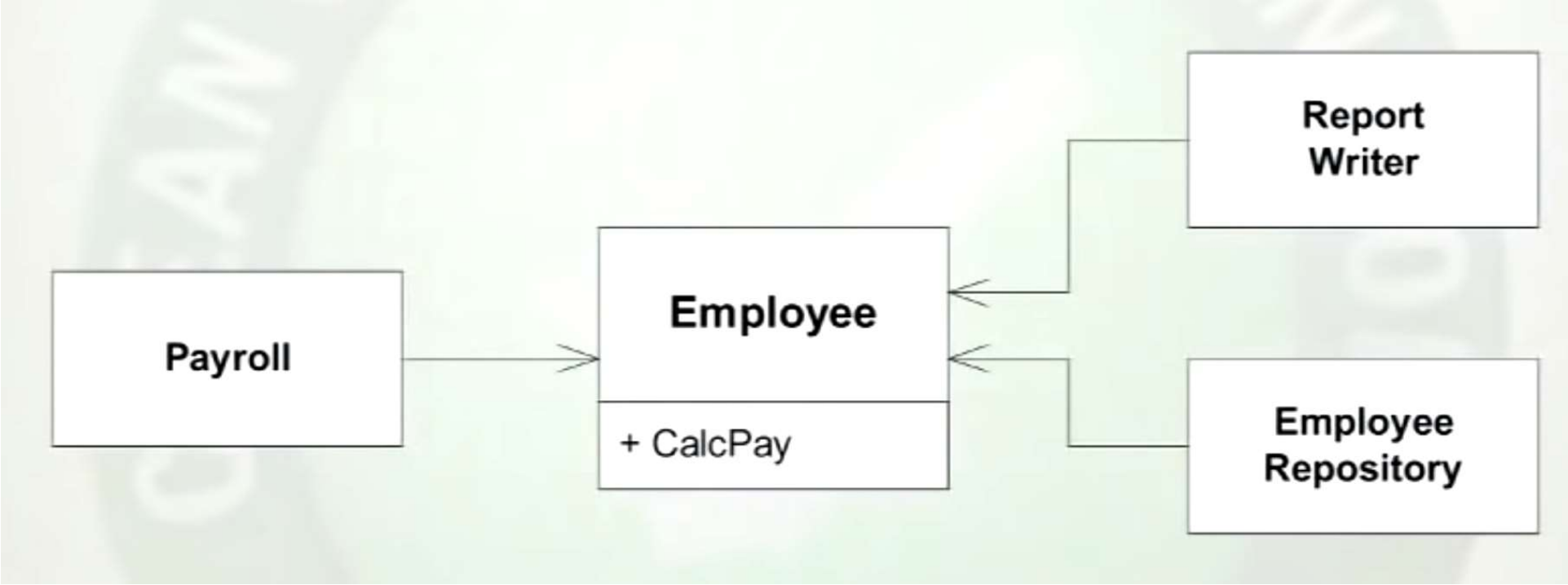
- "A class should have only one reason to change."
  - Robert C. Martin (aka Uncle Bob)
- Responsibility is defined as the "reason to change"
- A class/module should do only one thing and do it well



Single Responsibility Principle (SRP)

- Despite that EventManager handles the capabilities of managing events, there are really multiple responsibilities:
  - CalcPay (Payroll - CFO)
  - ReportHours (Auditing – COO)
  - WriteEmployee (Write data to database – CTO)
- Employee changes if there are change requests from CFO, or COO, or CTO

Single Responsibility Principle (SRP)



One way to separate the responsibilities

Open-Closed Principle (OCP)

- OCP originated from Bertrand Meyer:
  - A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
  - A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).
- Uncle Bob simply states it as

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

Open-Closed Principle (OCP)

- Suppose we want to draw circles and rectangles, which inherits (or implements) shape

```
class Main {
        public void drawShape(Shape s) {
                if (s instanceof Circle) {
                        drawCircle((Circle) s);
                } else if (s instanceof Rectangle) {
                        drawRectangle((Rectangle) s);
                }
        }
        public static void drawCircle(Circle c) {
                :
        }
        public static void drawRectangle(Rectangle r) {
                :
        }
}
```

Open-Closed Principle (OCP)

- There are several problems here:
  - To add a new shape and make it drawable, we have to modify the client `Main`
  - Adding a new shape also entails that we understand the code in `Main`
  - Apart from `Shape`, the `Main` class depends on `Circle`, `Rectangle` and other shapes to be included in future
  - Any changes to `Circle` or `Rectangle` or other shapes might break `Main`
  - Cannot isolate the respective shapes for testing
  - **if..else** structure should be avoided

Open-Closed Principle (OCP)

```
public interface Drawable {
        public void draw();
        ...
public abstract class Shape implements Drawable {
        ...
public class Triangle extends Shape {
        @Override
        public void draw() {
                ...
        }
        ...

class Main {
        public static void drawShape(Shape s) {
                s.draw();
        }

        public static void main(String[] args) {
                Shape s = getAShape();
                drawShape(s);
        }
        ...
```

A Problem

Brew coffee in the morning

- Involves coffee machines
  - Input: Coffee bean
  - Process: Brewing method
  - Output: Type of coffee
- Some_coffee Some_brewing_method (Some coffee beans)
- Use Interface to maintain stability of the code while supporting change

Liskov Substitution Principle (LSP)

- Introduced by Barbara Liskov

  "Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T."

- This **substitutability** principle means that if S is a subclass of T, then an object of type T can be replaced by that of type S without changing the desirable property of the program.

- As an example, if Square is a subclass of Rectangle, then everywhere we can expect rectangles to be used, we can replace a rectangle with a square.
  - Introducing setSize(w, h) in Rectangle breaks this principle as square.setSize(w, h) does not make sense

Liskov Substitution Principle (LSP)

- Another square and rectangle example:

```
class Rectangle extends Shape {
        protected double width;
        protected double height;

        void setWidth(double w) {
                width = w;
        }
        void setHeight(double h) {
                height = h;
        }
        double getArea() {
                return width * height;
        }
}
```

Liskov Substitution Principle (LSP)

```
class Square extends Rectangle {

        @Override
        void setWidth(double w) {
                width = w;
                height = w;
        }

        @Override
        void setHeight(double h) {
                width = h;
                height = h;
        }
}
```

Liskov Substitution Principle (LSP)

- Looking at the main method, what should be the expected output?

```
class Main {
        static Rectangle getRectangle() {
                return new Square();
        }
        static void main(String[] args) {
                Rectangle r = getRectangle();

                r.setWidth(5);
                r.setHeight(10);
                System.out.println(r.getArea());
                assert r.getArea() == 5 * 10;
        }
}
```

- What is the actual output?

Interface Segregation Principle (ISP)

- Uncle Bob (again)

  "no client should be forced to depend on methods it does not use."

- Keep interfaces minimal; avoid "fat" interfaces
  - Do not force classes to implement methods that they can't
  - Do not force clients to know of methods in classes that they don't need to

- Split interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them

Interface Segregation Principle (ISP)

```
public class Me {
        private String other;

        public Me(String other) {
                this.other = other;
        }
        public void mummysBoy() {
                System.out.println("Hi " + other + ". :)");
        }
        public void romantic() {
                System.out.println("Hey " + other + "... :x");
        }
        public void partyAnimal() {
                System.out.println("Supp " + other + "!!! :P");
        }
}
```

Interface Segregation Principle (ISP)

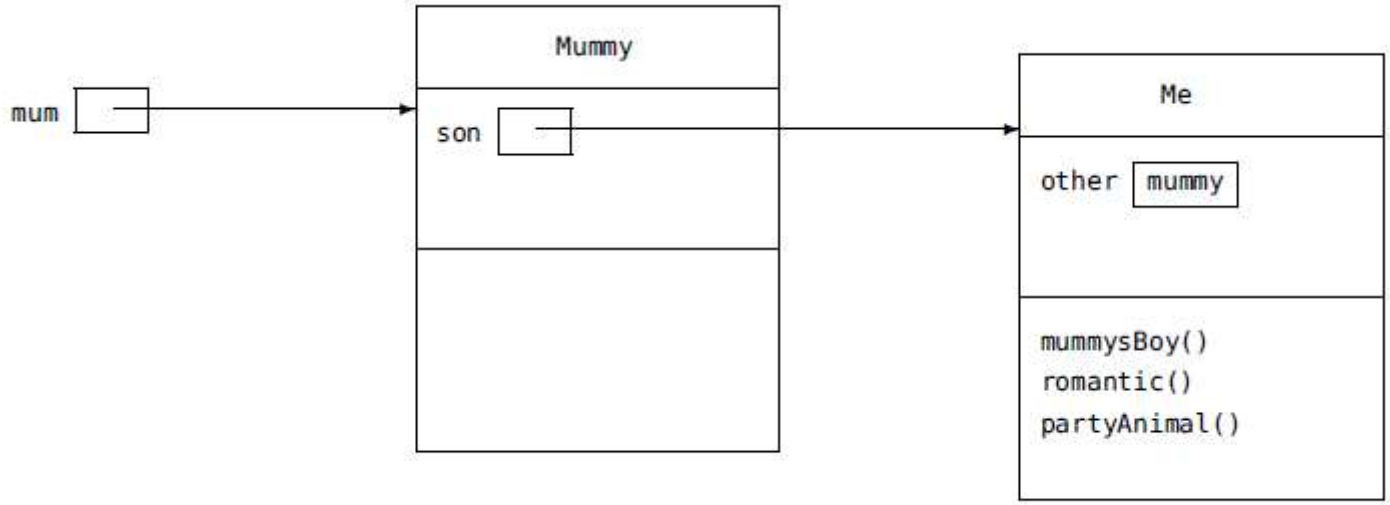
```
public class Mummy {
        Me son;
        public Mummy(Me son) {
                this.son = son;
        }
        public void greet() {
                son.mummysBoy();
                son.romantic();
                son.partyAnimal();
        }
}
```

```
$ java Main
Hi mummy. :)
Hey mummy... :x
Supp mummy!!! :P
```

```
class Main {
        public static void main(String[] args) {
                Mummy mum = new Mummy(new Me("mummy"));
                mum.greet();
        }
}
```

Interface Segregation Principle (ISP)

- Mummy knows (has access to) everything about Me



Interface Segregation Principle (ISP)

- Son shows only “appropriate” side to the right people

```
public class Me implements MummysBoy, Romantic, PartyAnimal {
        private String other;
        public Me(String other) {
                this.other = other;
        }
        @Override
        public void mummysBoy() {
                System.out.println("Hi " + other + ". :)");
        }
        @Override
        public void romantic() {
                System.out.println("Hey " + other + "... :x");
        }
        @Override
        public void partyAnimal() {
                System.out.println("Supp " + other + "!!! :P");
        }
}
```

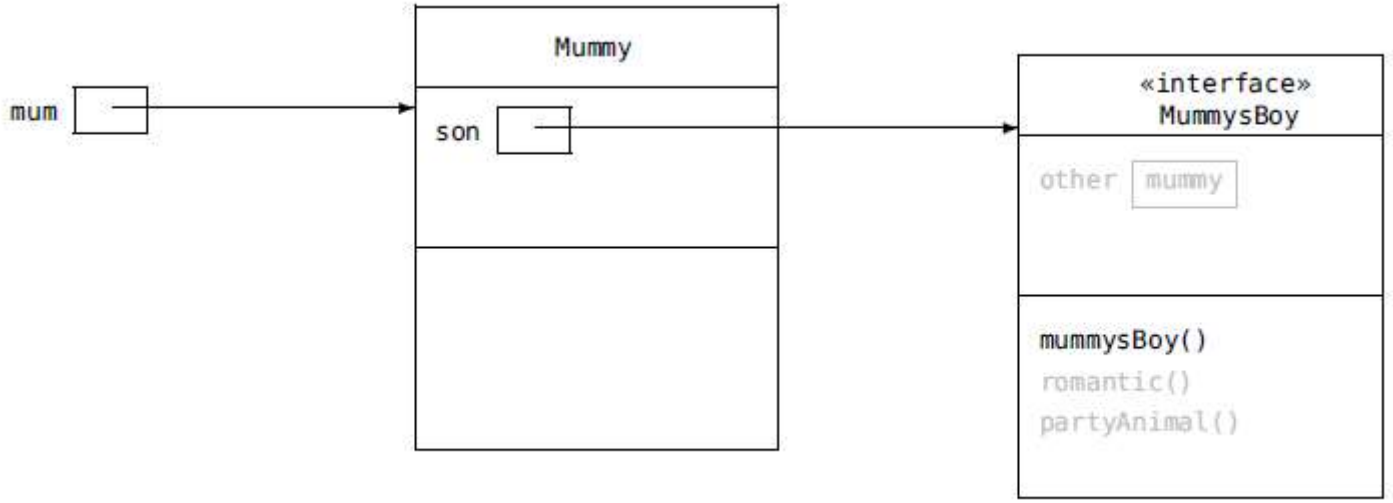
Interface Segregation Principle (ISP)

- Without changing the main method,
  - Mummy only wants to know the "filial" side of her son
  - Mummy references Me via the interface MummysBoy

```
public interface MummysBoy {
        public void mummysBoy();
}
public class Mummy {
        MummysBoy son;
        public Mummy(MummysBoy son) {
                this.son = son;
        }
        public void greet() {
                son.mummysBoy();
        }
}
```

Interface Segregation Principle (ISP)

- Mummy has restricted knowledge of (restricted access to) certain methods of Me



Dependency Inversion Principle (DIP)

- Uncle Bob (yet again)

  "High-level modules should not depend on low-level modules. Both should depend on abstractions."

  "Abstractions should not depend on details. Details should depend on abstractions."

- We shall not dwell on the specifics of DIP in this course, as they may not add too much value with the scale of the software is small
- It is sufficient to appreciate that DIP points to the practice of programming to an interface

Programming to an Interface

- Program to an interface, not an implementation
- Remember:
  - A **concrete class** is the actual implementation
  - An **interface** is a contract that specifies the abstraction that its implementer should implement
  - An **abstract class** is a trade off between the two, i.e. when we need to have the partial implementation of the contract
- A client should always use the interfaces from an application rather than the concrete implementations
- The benefits of this approach are maintainability, extensibility and testability

Lecture Summary

- Appreciate the application of the abstraction principle in the contexts of methods, as well as classes
- Demonstrate the application of SOLID principles in the design of object-oriented software, in particular
  - Single responsibility principle (SRP)
  - Open-closed principle (OCP)
  - Liskov substitution principle (LSP)
- Appreciate the advantages of programming to an interface which supports the maintainability, extensibility, and testing of the software