# CptS 487
# Software Design and Architecture

## Lesson 4

## OO Design Principles

Washington State University
World Class. Face to Face.

**Instructor: Bolong Zeng**

# Outline

- 5 OO Design Principles [MARTIN] Section 2
  - SRP: Single-Responsibilty
  - OCP: Open-Closed
  - LSP: Liskov Substitution
  - DIP: Dependency-Inversion
  - ISP: Interface-Segregation
- And others…
  - Proposed by the same author

# SRP: The Single-Responsibility Principle

- Aka: Cohesion again
- Reminder:
  - A system/subsystem has high cohesion if it keeps together things that are related to each other, and keeps out other things
  - Single/Focused responsibility

# SRP: The Single-Responsibility Principle

- Exercise question:
  - Someone designed a module named “FindSet”; the module is capable of the following:
    - Reading user input of which set of data the user needs
    - Accessing the database to fetch said set of data
    - Organize the data into a report and display them for the user.
  - Does this module have high or low cohesion?

# SRP: The Single-Responsibility Principle

- Fall 2015 students: 50/50 on High/Low
- Correct answer
  - Low
- In the context of the SRP, we define a responsibility to be “a reason for change” [MARTIN]
  - If you can think of more than one motive for changing a class/module/subsystem, then it has more than one responsibility.
- http://www.oodesign.com/single-responsibility-principle.html

# OCP: The Open-Closed Principle

- *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*
  - Design shall allow the extension without modification to internal logics
  - E.g. use superclasses to maintain the abstraction and interface of sensor classes, and if a future sensor fits to the interface defined by the superclass, it can be plugged into the system without changing its internal logics.

- *Abstraction* is the key.

# OCP: The Open-Closed Principle

- Open for extension
  - We can extend the behaviors of the module
- Closed for modification
  - …without modifying the original code

- Requires we have a higher level of abstraction/interface available.
- We may do it through inheritance and/or composition, for instance.
  - Will address in discussion Object Design.

- http://www.oodesign.com/open-close-principle.html

# LSP: The Liskov Substitution Principle

- *Subtypes must be substitutable for their base types.*
  - Proposed by Barbara Liskov in 1988

- The Liskov Substitution Principle (LSP) seems obvious given all we know about polymorphism
- For example:

```
public void drawShape(Shape s) {
// Code here.
}
```

- The `drawShape` method should work with any subclass of the `Shape` superclass (or, if Shape is a Java interface, it should work with any class that implements the Shape interface)
- But we must be careful when we implement subclasses to insure that we do not unintentionally violate the LSP
  - Example from http://www.oodesign.com/liskov-s-substitution-principle.html

# LSP Example

- Consider the following Rectangle class:

```
// A Rectangle class.
public class Rectangle {

   private double width;
   private double height;

   public Rectangle(double w, double h) {
     width = w;
     height = h;
   }

   public double getWidth() {return width;}
   public double getHeight() {return height;}
   public void setWidth(double w) {width = w;}
   public void setHeight(double h) {height = h;}
   public double area() {return (width * height);
}
```

# LSP Example (Continued)

- How about a `Square` class? Clearly, a square is a rectangle, so the `Square` class should be derived from the `Rectangle` class.
- Observations:
  - A square does not need both a width and a height as attributes, but it will inherit them from Rectangle anyway. So, each `Square` object wastes a little memory, but this is not a major concern.
  - The inherited `setWidth()` and `setHeight()` methods are not really appropriate for a `Square`, since the `width` and `height` of a square are identical. So we'll need to override `setWidth()` and `setHeight()`. Having to override these simple methods is a clue that this might not be an appropriate use of inheritance!

# LSP Example (Continued)

- Here is the Square class:

```
// A Square class.
public class Square extends Rectangle {

 public Square(double s) {super(s, s);}

 public void setWidth(double w) {
   super.setWidth(w);
   super.setHeight(w);
 }

 public void setHeight(double h) {
   super.setHeight(h);
   super.setWidth(h);
 }
}
```

- Did you detect the problem?

# LSP Example (Continued)

- Everything looks good. Let’s test it:

```
public class TestRectangle {

// Define a method that takes a Rectangle reference.
 public static void testLSP(Rectangle r, float w, float h) {

    r.setWidth(w);
    r.setHeight(h);


    System.out.println("Width is ”+w+” and Height is “+h+
    ", so Area is " + r.area());

    if (r.area() == w*h)
       System.out.println("Looking good!\n");
    else
       System.out.println("Huh?? What kind of rectangle is
     this??\n");
  }
}
```

# LSP Example (Continued)

```
public static void main(String args[]) {

  //Create a Rectangle and a Square
  Rectangle r = new Rectangle(4.0, 5.0);
  Square s = new Square(5.0);

  testLSP(r, 4.0, 5.0);
  testLSP(s, 4.0, 5.0);
  }
}
```

According to the LSP, it should work for both Rectangles or Squares. Does it??

Test program output:

```
Width is 4.0 and Height is 5.0, so Area is 20.0
Looking good!
Width is 4.0 and Height is 5.0, so Area is 25.0
Huh?? What kind of rectangle is this??
```

Looks like we violated the LSP!

# LSP Example (Continued)

- What's the problem here? The programmer of the testLSP() method made the reasonable assumption that changing the width of a `Rectangle` leaves its height unchanged.
  - A mathematical square might be a rectangle, but a Square object is not a Rectangle object, because the behavior of a Square object is not consistent with the behavior of a Rectangle object!
  - Behaviorally, a Square is not a Rectangle! A Square object is not polymorphic with a Rectangle object.

- Passing a `Square` object to such a method results in problems, exposing a violation of the LSP

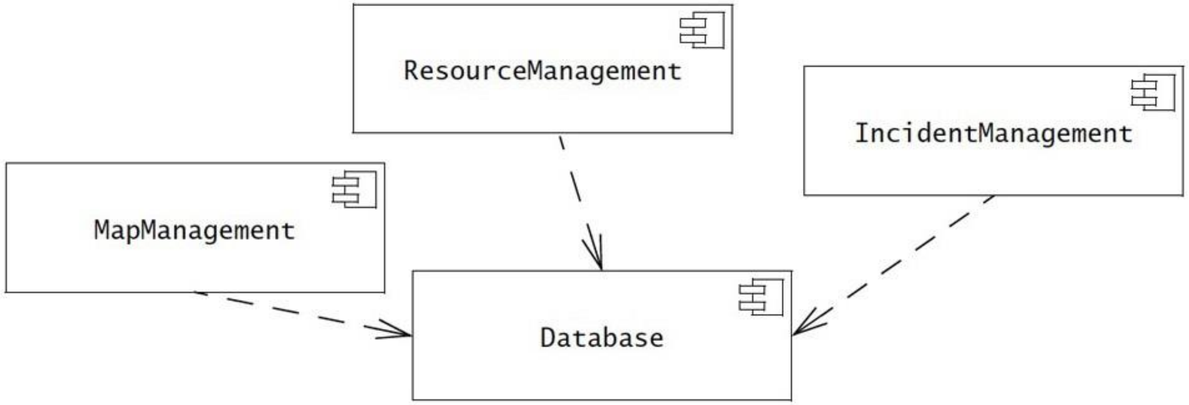
# LSP: The Liskov Substitution Principle

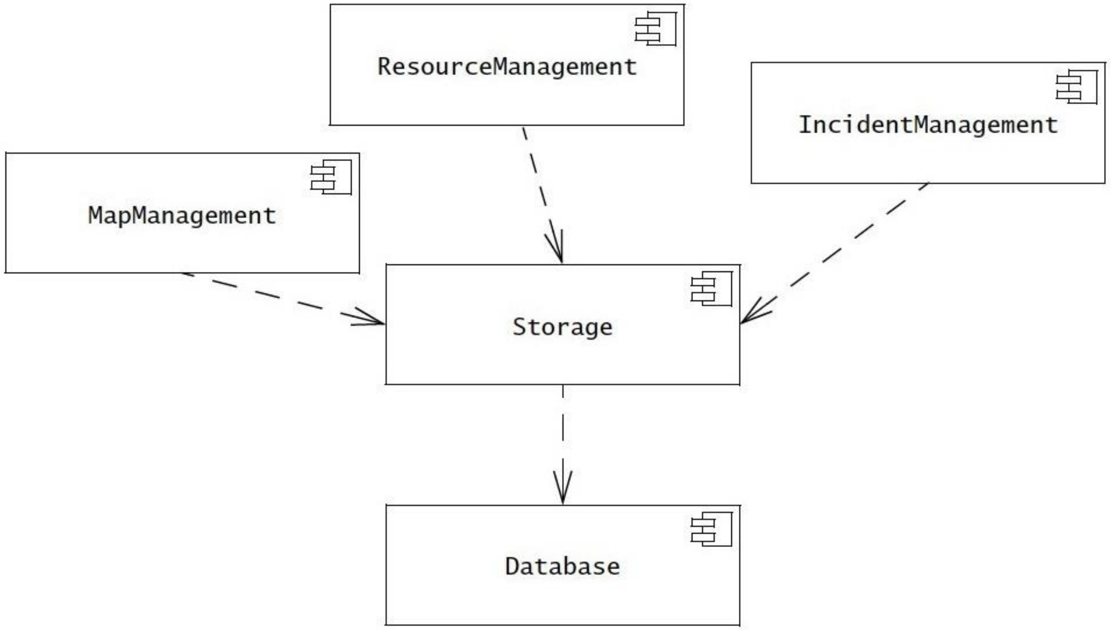
- In order for the LSP to hold all subclasses must conform to the behavior that clients expect of the base classes they use
- A subtype must have no more constraints than its base type, since the subtype must be usable anywhere the base type is usable
- If the subtype has more constraints than the base type, there would be uses that would be valid for the base type, but that would violate one of the extra constraints of the subtype and thus violate the LSP!
- The guarantee of the LSP is that a subclass can always be used wherever its base class is used!

# DIP: The Dependency-Inversion Principle

- 1) High-level modules should not depend on low-level modules. Both should depend on abstraction
  - "Copy" example.
- 2) Abstractions should not depend on details. Details should depend on abstractions.
  - Programming to interfaces, not implementations.
- Low coupling

# DIP: The Dependency-Inversion Principle





# DIP: The Dependency-Inversion Principle

- Can be applied wherever one class sends a message to another.

- http://www.oodesign.com/dependency-inversion-principle.html

# ISP: The Interface-Segregation Principle

- *Clients should not be forced to depend on methods that they do not use.*
- Many client-specific interfaces are better than one general purpose interface.
  - Client-specific interfaces control the access to a class or a component.
  - E.g. separate the security interface from the ordinary user interface.
- Cohesion & Coupling

# ISP: The Interface-Segregation Principle

- http://www.oodesign.com/interface-segregation-principle.html

- The ATM user interface example in [MARTIN] Chapter 12
  - Provides a good guidance on how to design the structure of your GUI code!
    - Avoid gigantic “form1.cs”

# Basic Design Principles*

- The Release Reuse Equivalency Principle (REP). *"The granule of reuse is the granule of release."*
  - The update on the design of a component shall guarantee the downward (backward) compatibility.
- The Common Closure Principle (CCP). *"Classes that change together belong together."*
  - Changes are localized to a component for a better change control and release management.
- The Common Reuse Principle (CRP). *"Classes that aren't reused together should not be grouped together."*
  - If a class not reused together is included in a package, it will be tested with other classes when changes and reuse occur. This causes the unnecessary overhead of testing and maintenance.

***Source: Martin, R., "Design Principles and Design Patterns," downloaded from http:www.objectmentor.com, 2000.**