S.O.L.I.D Design Principles

CSCI 2300

S.O.L.I.D Decoded

- S – Single Responsibility Principle
- O – Open/Closed Principle
- L – Liskov Substitution Principle
- I – Interface Segregation Principle
- D – Dependency Inversion Principle

Single Responsibility Principle (SRP)

- Each class has one responsibility (and one reason to change)

SRP Violated (from Tic Tac Toe)

```
class Board{
   public void reset(){…}
   public boolean setPiece(…){…}
   public boolean hasWinner(…){…}
   public boolean hasOpenPosition(){…}
   public void display(){…}
}
```

Board is responsible for too much

How can we fix it:

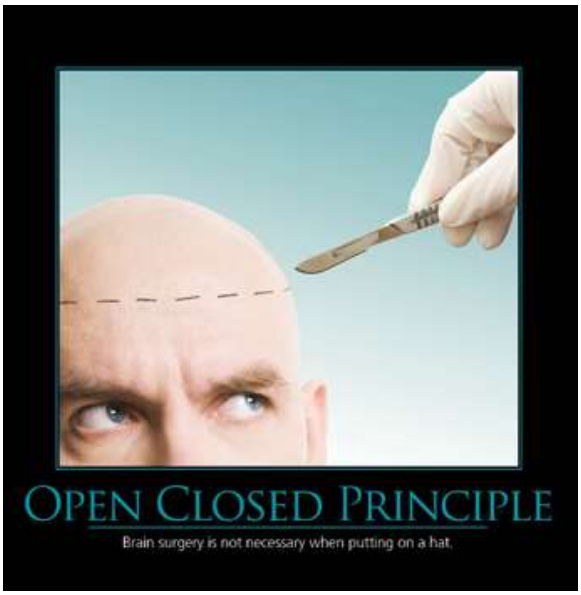
- Separate responsibilities into multiple classes:
  - Board
  - BoardView (or BoardDisplay)
- Use interfaces:
  - IBoard – interface for board
  - IBoardDisplay – interface for displaying a board
- Use composition:
  - Board – the model of the board
  - BoardDisplay has an instance of Board

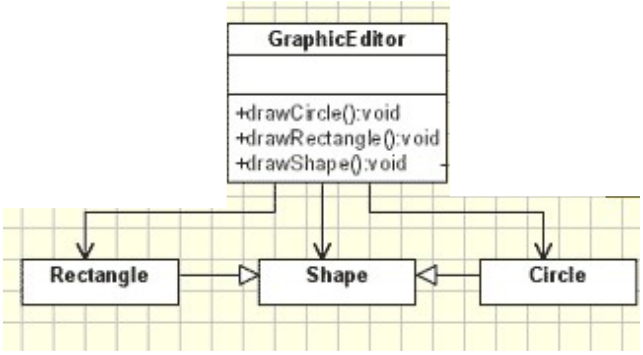
```
class Person {
    protected String firstName; //get and set methods
    protected String lastName;  //get and set methods
    protected Gender gender;    //get and set methods
    protected DateTime dateOfBirth;
    public string Format(string formatType) {
        switch(formatType) {
            case "XML":
               return xmlFormattedString; break;
            case "FirstAndLastName":
              return firstAndLastNameString; break;
            default:
              // implementation of default formatting
              return defaultFormattedString;
        }
    }
}
```

A. This class violates SRP because it encapsulates multiple attributes of a person
B. This class violates SRP because it does not have a constructor
C. This class violates SRP because it is responsible for encapsulating "person" attributes and formatting them
D. This class does not violate SRP

Open/Closed Principle (OCP)

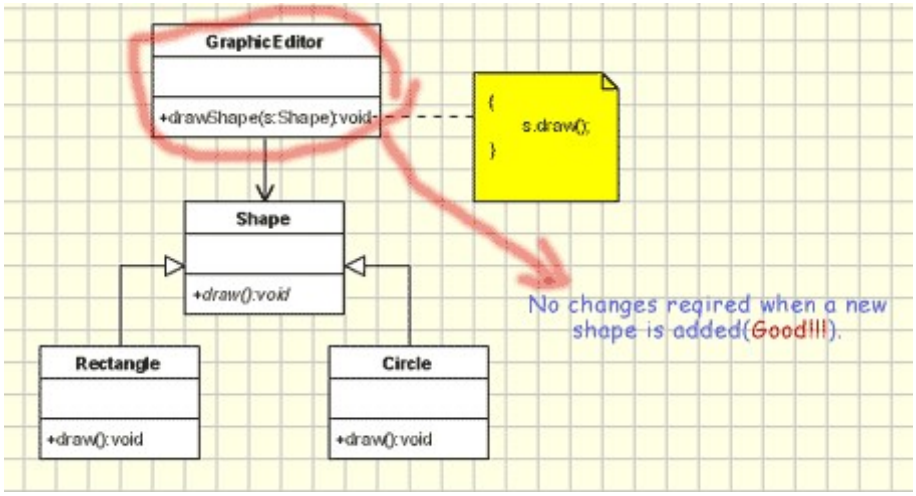
- Classes/methods/modules should be open for extension but closed for modification
- Create classes/methods/modules that whose behavior can change without recompiling the code
- If we can extend software to satisfy new requirements, without modifying existing code, the design satisfies OCP
- Simple example: pass parameters to methods instead of hard coding values.





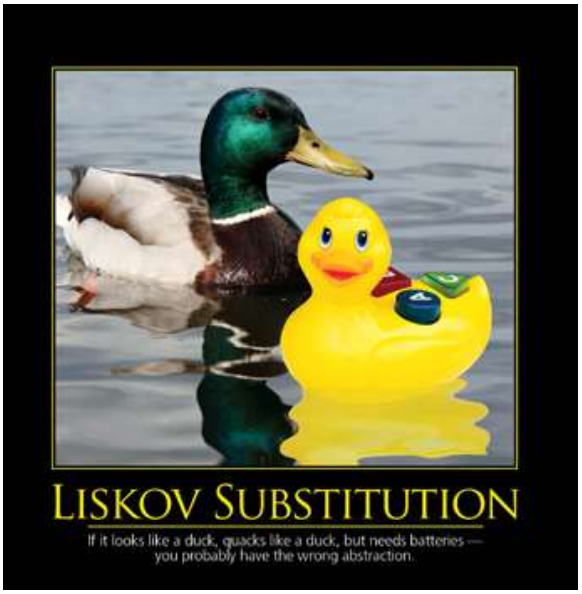
A. This design violates OCP because it has drawShape() method and shape is abstract
B. This design violates OCP because we'll need to change Graphic Editor if we add another shape.
C. This design does not violate OCP because if we add another shape, we don't have to change GraphicEditor: we can create a sub-class of GraphicEditor and add the necessary draw function there.
D. None of the above are true

Improved Design



Liskov Substitution Principle (LSP)

- Subtypes can be substituted for their base types
- Subclass IS-SUBSTITUTABLE-FOR base class



Violation of LSP

- Class “Rectangle”
- Class “Square” extends “Rectangle”
- “Square” enforces that height and width are equal
- Suppose the following code:

```
public double area(Rectangle r)
{
    r.setHeight(10);
    r.setWidth(5);
    return r.getArea();
}
```

```
public double calculate (Bill bill)
{
   if (bill instanceOf LargeGroupBill)
   {
      // add up the bill items and add 15% gratuity
   }
   else{
      // add up the bill items
   }
}
```

A. This code violates LSP because we are checking the sub-type of Bill to determine the logic
B. This code violates LSP because the calculation should be done in the Bill class
C. This code violates LSP because LargeGroupBill is not defined
D. This code does not violate LSP

```
public class Vehicle{
    public void drive(int miles){
        if (miles > 0 && miles < 300){…}
    }
}
 public class Scooter extends Vehicle{
     public void drive(int miles){
         if (miles > 0 && miles < 50){
            super.drive(miles);
         }
     }
 }
```

A. This code does not violate LSP.
B. This code violates LSP because it restricts the behavior of Vehicle.
C. This code violates LSP because the drive() method of Scooter calls parent's drive() method.

```
class ToyCar extends Vehicle{
    public void drive(int miles) {
      // Show flashy lights, make random sounds
    }
    public void fillUpWithFuel() {
       // silly lights and noises
    public int fuelRemaining { return 0;}
}
```

A. This code does not violate LSP
B. This code violates LSP because it completely changes the behavior of drive() and fillUpWithFuel()
C. This code violates LSP because fuelRemaining() returns 0
D. B and C.

Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods they do not use
- Clients should not implement methods if those methods are unused.

ISP Violation

```
public interface IMembership
{
  boolean Login(string username, string password);
  void Logout(string username);
  Guid Register(string username, string
                password, string email);
  void ForgotPassword(string username);
}
```

Improved Design

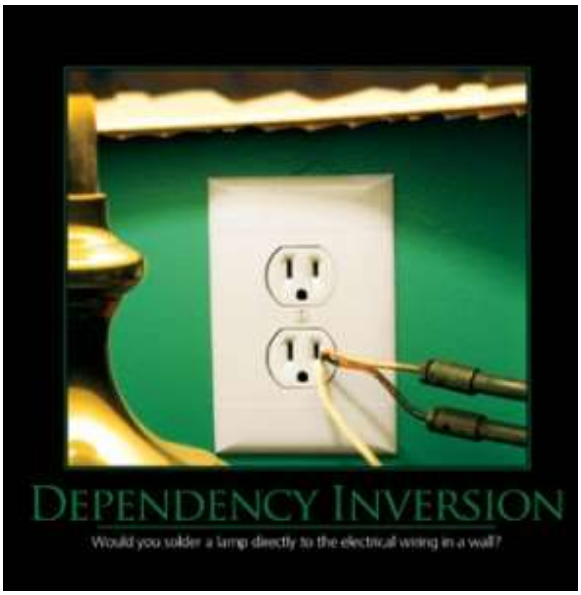
```
public interface ILogin
{
  boolean Login(String username, String password);
  void Logout(String username);
}

public interface IMembership extends ILogin
{
  Guid Register(String username, String password,
                String email);
  void ForgotPassword(string username);
}
```

```
public interface Movable{
   public void move();
   public void setSpeed(int speed);
}
public class Picture implements Movable{
   public void move(){// code for moving a
picture across the screen}
   public void setSpeed(int speed){return;}
}
```

A. This code violates ISP because Picture class has a "dummy" implementation of setSpeed()
B. This code violates ISP because Movable interface has more than one method
C. This code violates ISP because we should be able to control the speed at which Picture moves across the screen
D. This code does not violate ISP

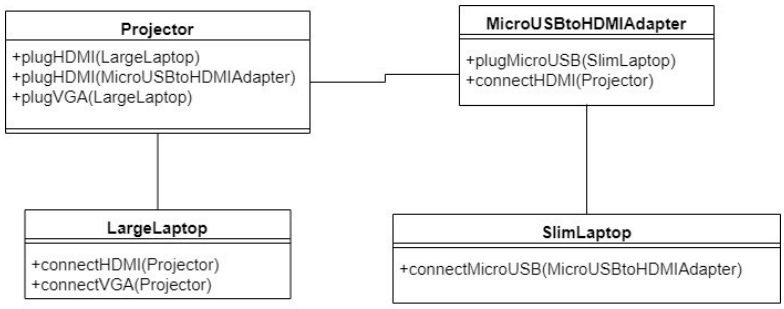
Dependency Inversion Principle (DIP)

- High level modules should not depend on low level modules
- Both should depend on abstraction
- Abstractions should not depend on details
- Details should depend upon abstraction



Dependency Inversion Principle Example

- There is a tight coupling between classes
- Dependency on low level modules
- If we add another device, we'll need to adjust existing code and add another dependency to Projector



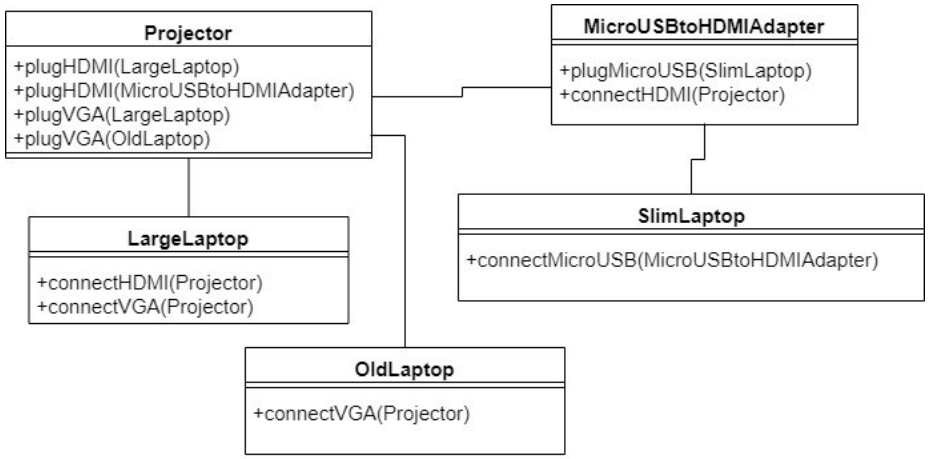
Suppose we have the following objects:

```
Projector projector
SlimLaptop slimLaptop
LargeLaptop largeLaptop
MicroUSBtoHDMIAdapter adapter
```

What operation(s) is/are possible?

A. projector.plug(slimLaptop)
B. projector.plug(largeLaptop)
C. projector.plug(adapter)
D. B and C
E. All of the above

Added a new device: Old Laptop



Consider the redesigned interfaces from the handout. What changes are needed to get the following code to work (assuming `projector` and `oldLaptop` have been instantiated).

```
projector.plug(oldLaptop)
```

A. OldLaptop implements HDMIPort
B. OldLaptop implements VGAPort
C. OldLaptop implements HDMIPlug
D. OldLaptop implements VGAPlug
E. Projector implements OldLaptop