S.O.L.I.D: Software Engineering Principles

Eduardo Figueiredo

http://www.dcc.ufmg.br/~figueiredo

S.O.L.I.D Principles

- These principles intend to create systems that are easier to maintain and extend

S Single-responsibility principle

O Open-closed principle

L Liskov substitution principle

I Interface segregation principle

D Dependency inversion principle

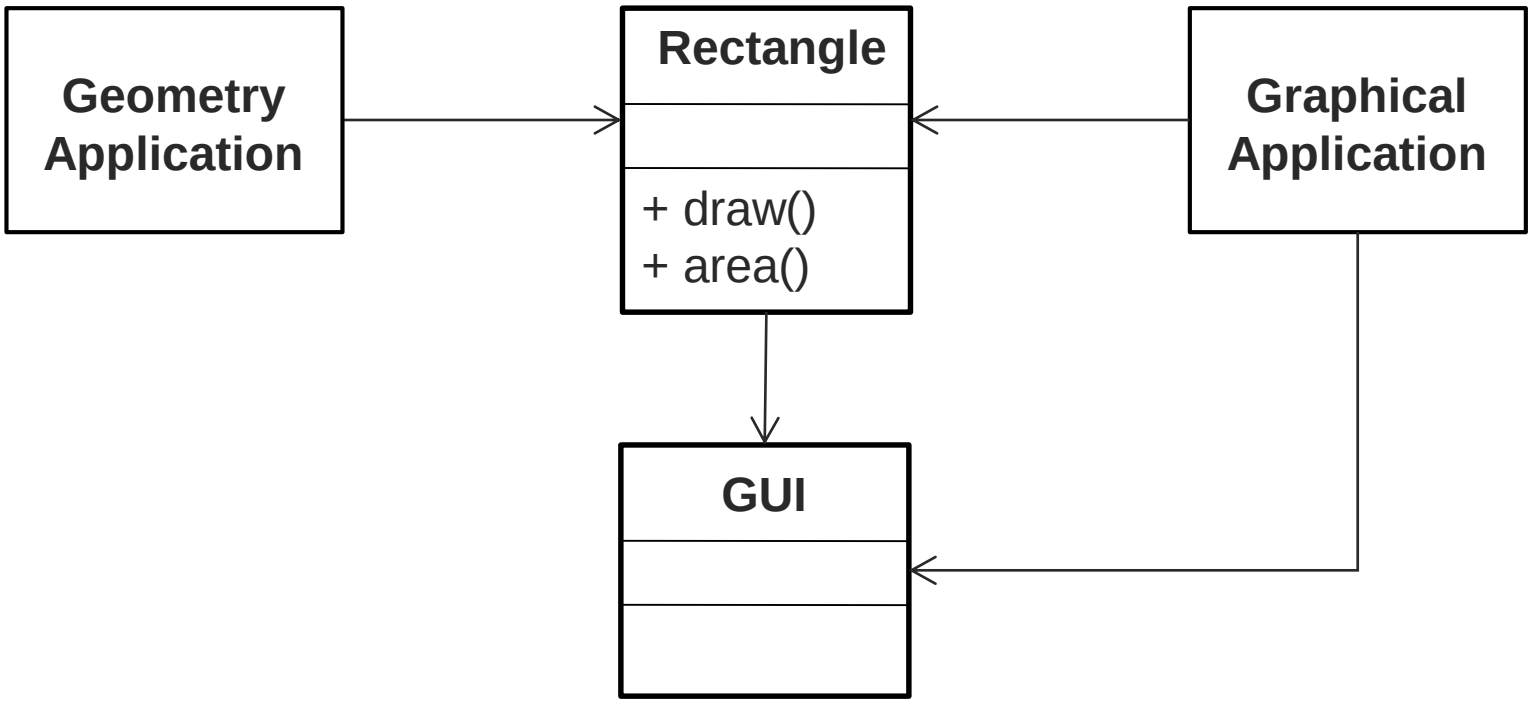
Single-Responsibility Principle

Single Responsibility Principle

A class should have only one reason to change

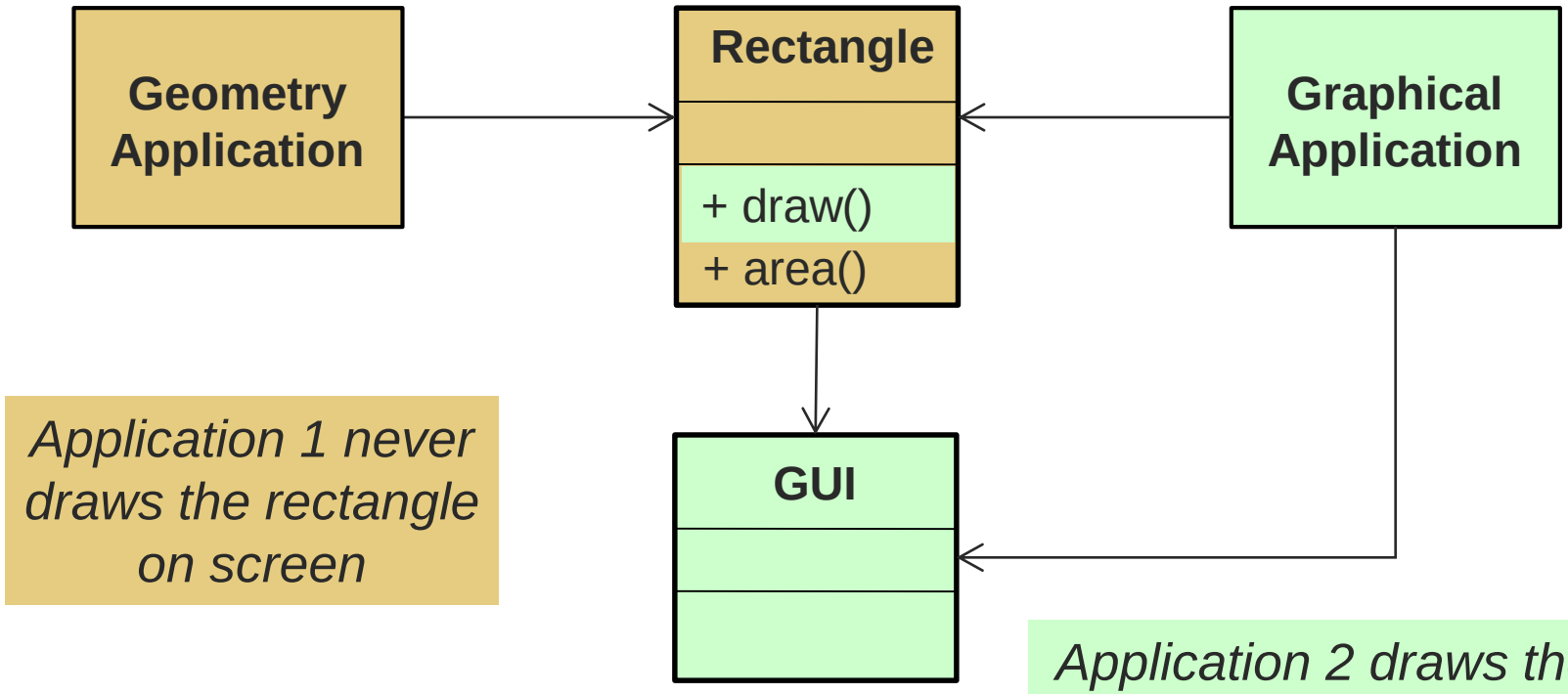
- A responsibility is a reason for change
  - If a class assumes more than one responsibility, it will have more than one reason for change
- This principle is related to low cohesion

Example of Violation



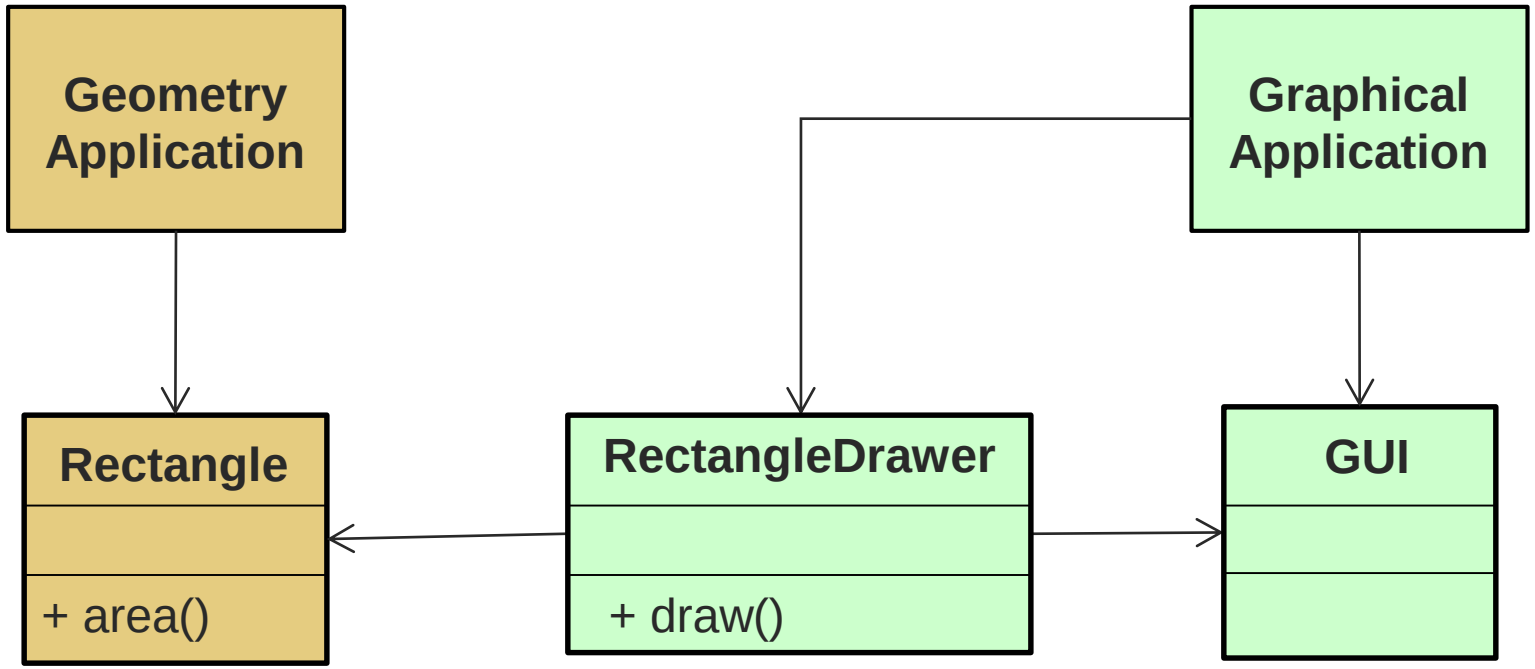
Example of Violation

The Rectangle class has two responsibilities



Separated Responsibilities

Solution: move the drawing responsibility to another class



Open-Closed Principle

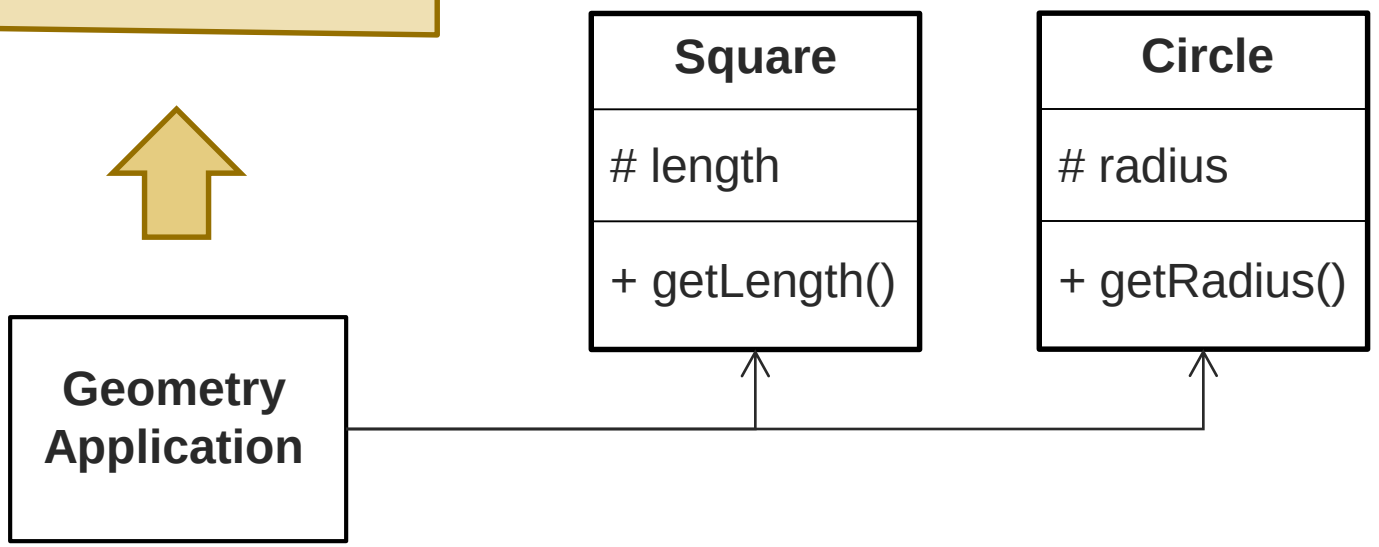
Open-Closed Principle

Software entities should be open for extension, but closed for modification

- Modules should never change
- Behavior of modules can be extended
  - When requirements change, you extend the behavior of modules by adding code
- Encapsulation is a key concept
  - Avoid public and global variables

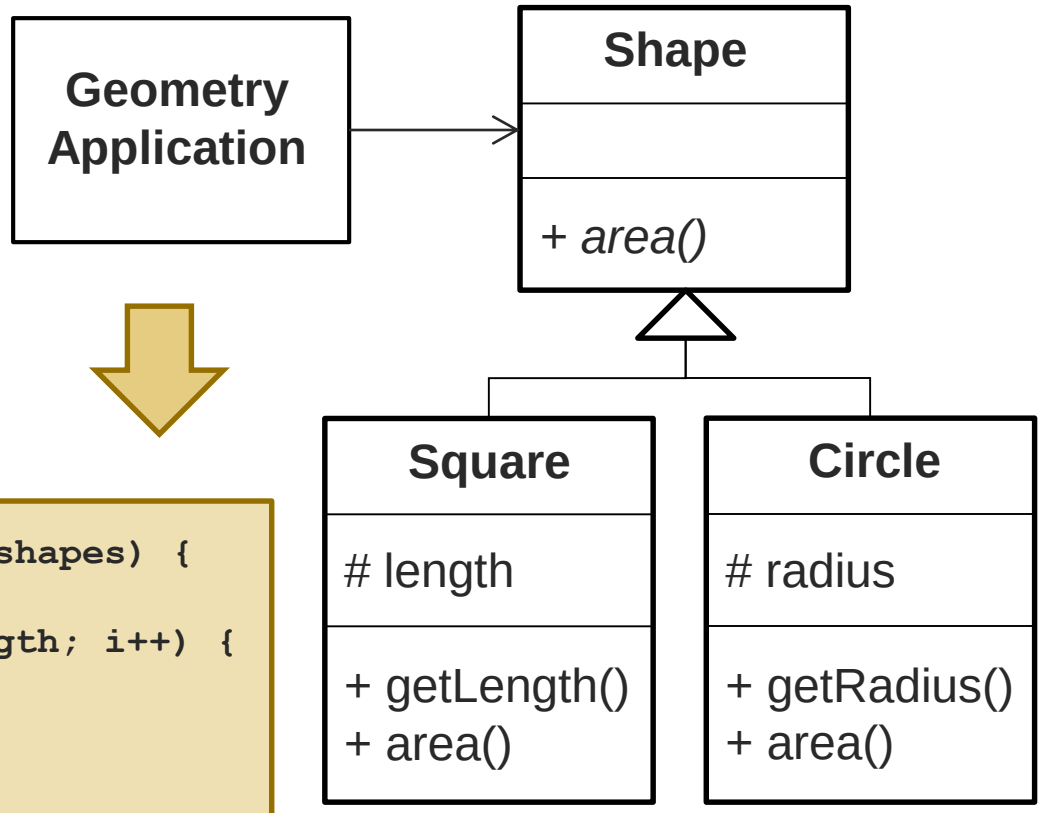
Example of Violation

```
public double sumArea(Object[] shapes) {
    double area = 0;
    for (int i = 0; i<shapes.length; i++) {
      if (shapes[i] instanceof Square)
        area += Math.pow(((Square)shapes[i]).getLength(), 2);
      if (shapes[i] instanceof Circle)
        area += Math.PI * ( Math.pow(((Circle)shapes[i]).getRadius(), 2) );
    }
    return area;
}
```



Open for Extension

Solution: client is coupled only to abstraction




```
public double sumArea(Shape[] shapes) {
  double area = 0;
  for (int i = 0; i<shapes.length; i++) {
    area += shapes[i].area();
  }
  return area;
}
```

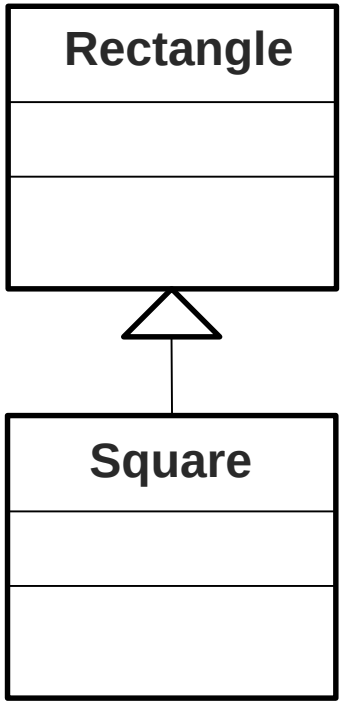
Liskov Substitution Principle

Liskov Substitution Principle

Functions that use references to classes must be able to use objects of subclasses without knowing it

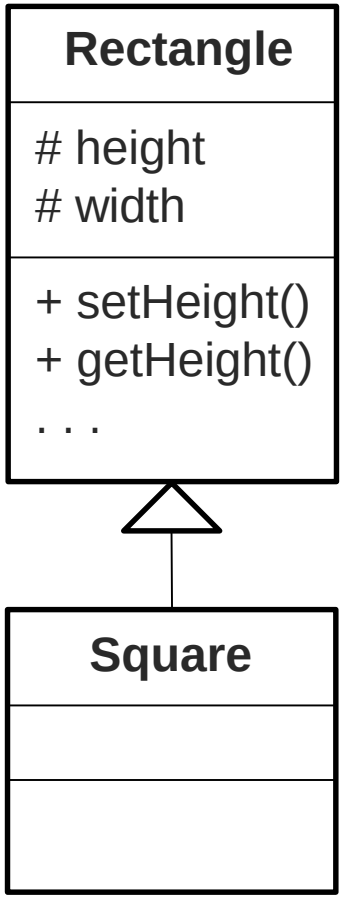
- This principle implies in careful use of inheritance (“is a” relationship)
- All subclasses must conform to the behavior that clients expect
  - Functions which use base classes should be reused without penalty

Square extends Rectangle



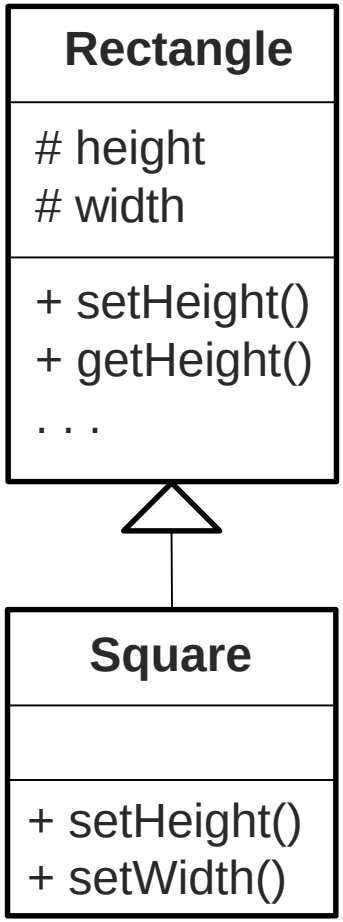
- An application has to manipulate squares in addition to rectangles
- Inheritance represents the “is a” relationship
  - Square is a rectangle (“is a” relationship holds)

Useless Inherited Members



- A square does not need both *height* and *width* fields (and other members)
  - It inherits them anyway
- The methods *setHeight()* and *setWidth()* are inappropriate
  - How to fix them?

Example of Violation



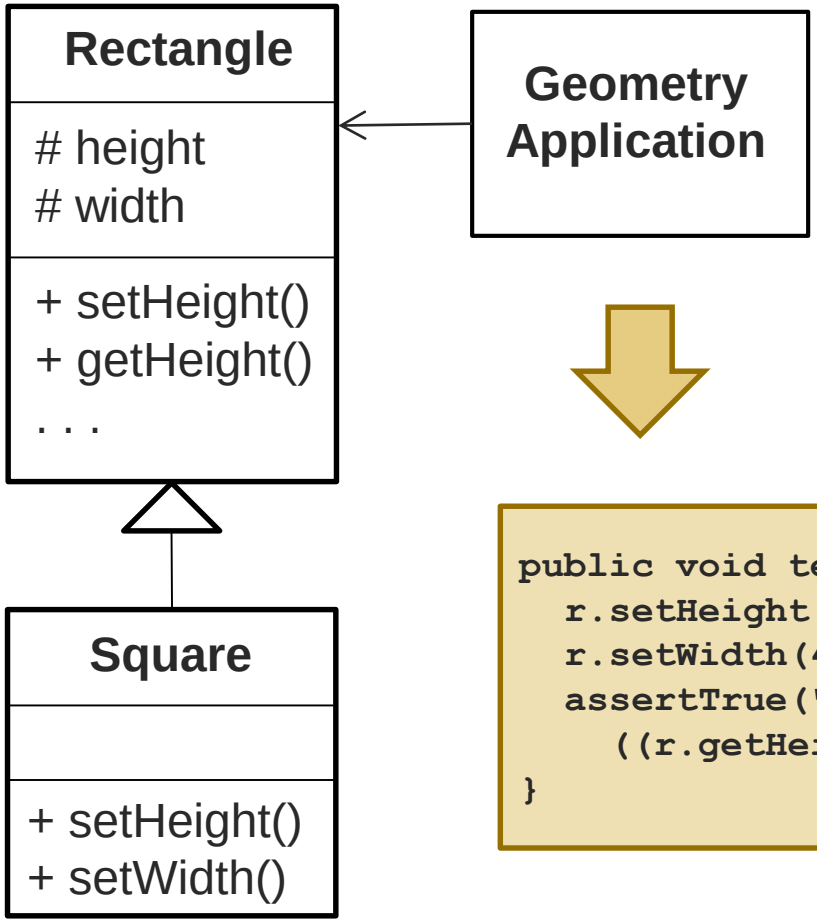
Workaround: override the *setHeight()* and *setWidth()* methods

```
public class Square extends Rectangle {

  public void setHeight(int h) {
    this.height = h;
    this.width = h;
  }

  public void setWidth(int w) {
    this.height = w;
    this.width = w;
  }
}
```

Example of Violation



Problem: *testArea()* works for Rectangle, but not for Square

```
public void testArea(Rectangle r) {
  r.setHeight(5);
  r.setWidth(4);
  assertTrue("Test rectangle area",
    ((r.getHeight() * r.getWidth()) == 20) );
}
```

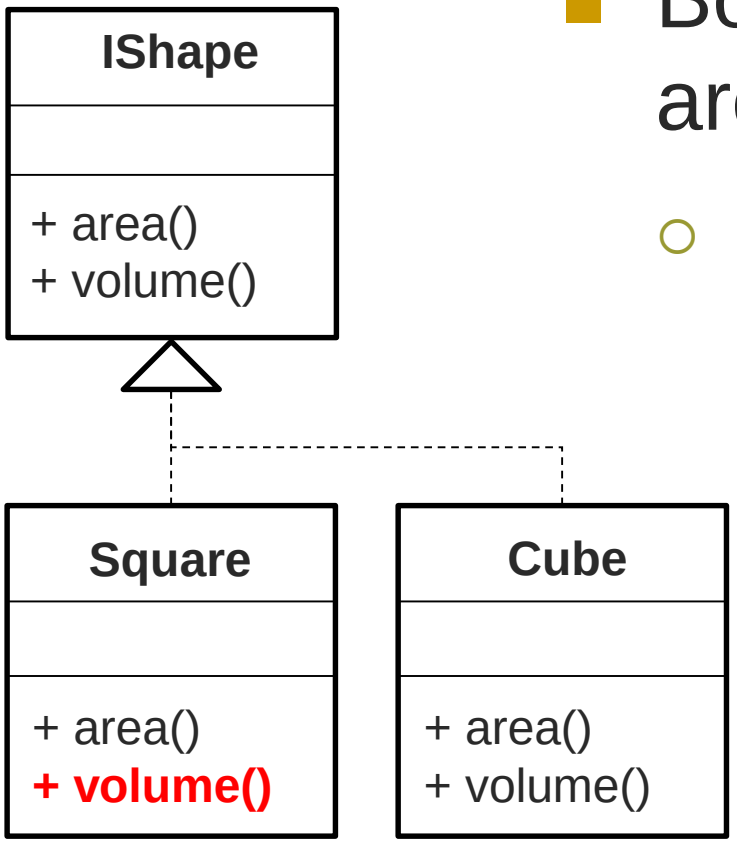
Interface Segregation Principle

Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use

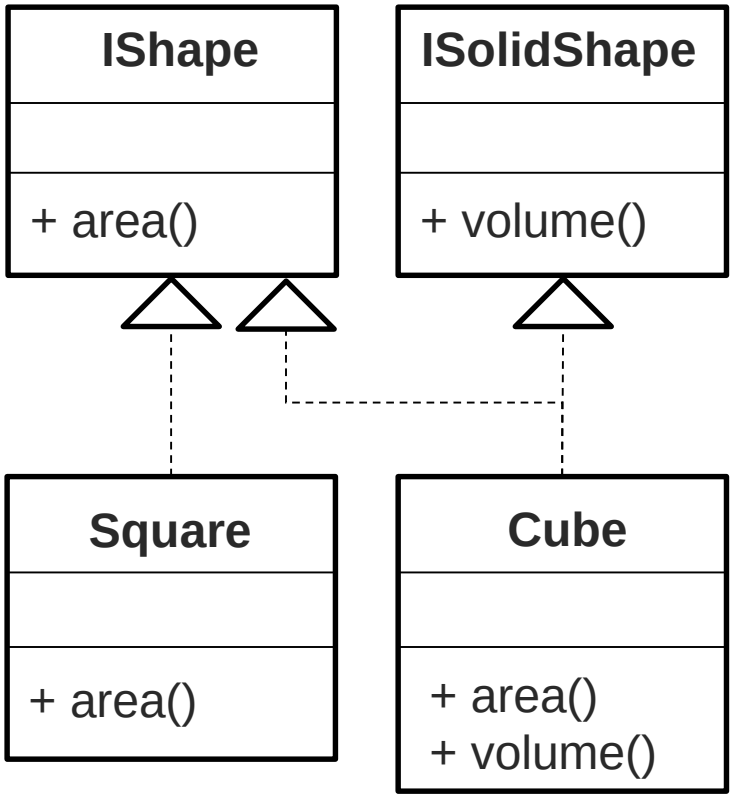
- This principle deals with the disadvantages of “fat” interfaces
  - Classes with “fat” interfaces are not cohesive
- There are objects with non-cohesive functionalities, but clients should know them by their (many) cohesive interfaces

Example of Violation



- Both square and cube are shapes
  - The “is a” relationship holds
    - A square does not need the *volume()* method
      - It inherits this method anyway

Segregated Interfaces



Solution: break down a "fat" interface into smaller interfaces with cohesive sets of responsibilities

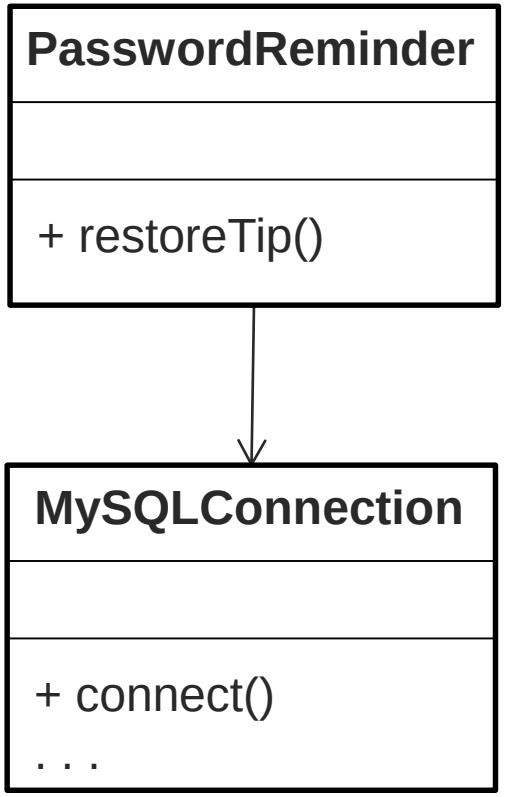
Dependency Inversion Principle

Dependency Inversion Principle

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

- We want to reuse high level modules
  - High level modules are hard to reuse, when they depend on details
  - We easily use low level modules as functions or libraries

Example of Violation

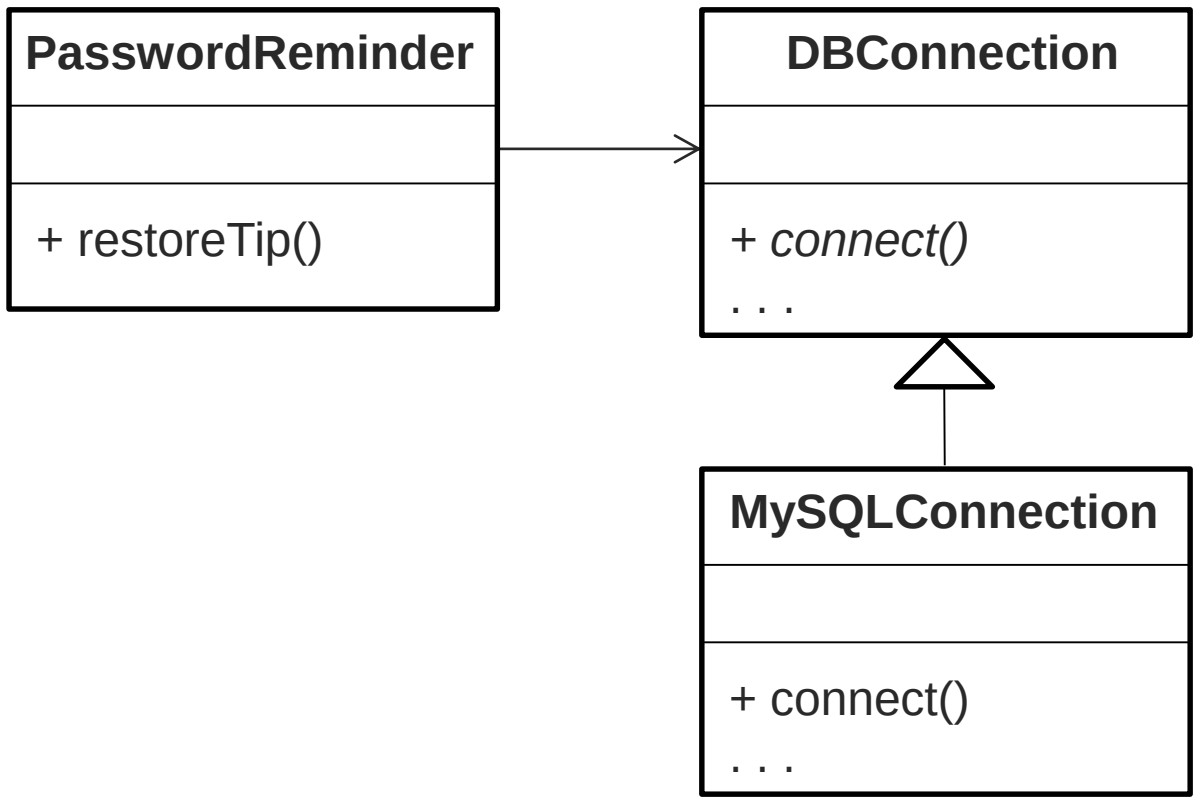



```
public String restoreTip() {
  String tip = "";
  MySQLConnection c = new MySQLConnection();
  // connect to MySQL database
  // recovery password tip
  // close database connection
  return tip;
}
```

Problem: if you change your database engine later, you have to edit the *PasswordReminder* class

Dependency Inversion

Solution: the *PasswordReminder* class can connect to the database without knowing the engine



Template Method Example

- The Template Method design pattern is often an example of this principle

```
public class Trip {
  public final void performTrip(){
    arrive();
    doDayA();
    doDayB();
    doDayC();
    leave();
  }

  public void arrive() { ... }

  public abstract void doDayA();

  public abstract void doDayB();

  public abstract void doDayC();

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

Bibliography

- Robert C. Martin. **Agile Software Development, Principles, Patterns, and Practices**. Pearson Education Limited, 2013.
  - Chapter 8 to 12