SOLID Principles

Ingegneria Del Software

Università degli Studi di Padova
Dipartimento di Matematica
Corso di Laurea in Informatica

rcardin@math.unipd.it

Summary

- Introduction
- Single Responsibility Principle
- Open-Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Introduction

- Structured programming and Object-oriented programming
  - Two of the most important revolution of IT industry
    - Everyone uses OO languages, but...
  - Today's programmers are unaware of the principles that are the foundation of Object Orientation
- Dependency management
  - The art of making code flexible, robust, and reusable
    - It's too easy to get a bunch of tangled legacy code
  - SOLID principles
    - A set of class design principles that helps to manage dependency

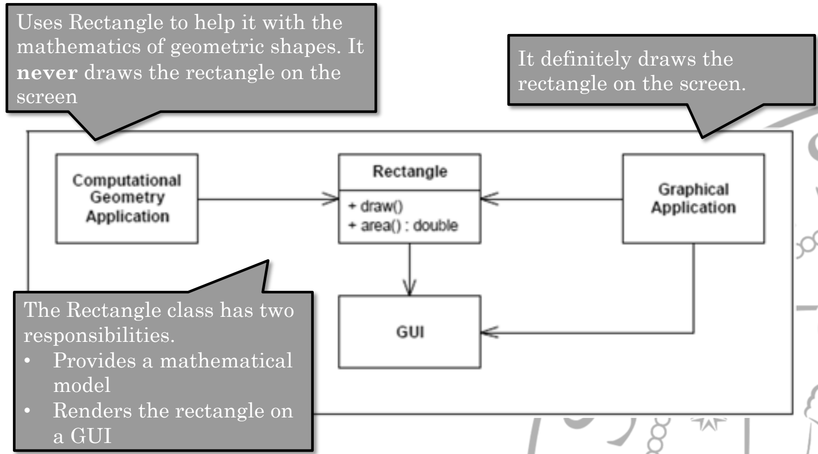
Introduction

- SOLID principles
  - **S**ingle Responsibility Principle
    - A class should have one, and only one, reason to change
  - **O**pen Closed Principle
    - You should be able to extend a classes behavior, without modifying it
  - **L**iskov Substitution Principle
    - Derived classes must be substitutable for their base classes
  - **I**nterface Segregation Principle
    - Make fine grained interfaces that are client specific
  - **D**ependency Inversion Principle
    - Depend on abstractions, not on concretions

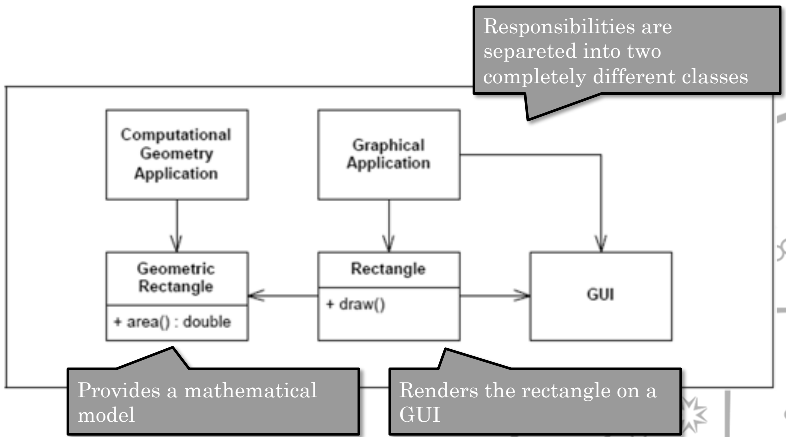
SINGLE RESPONSIBILITY PRINCIPLE

- Also known as cohesion
  - Functional relatedness of the elements of a module
  - A module should have only one reason to change
    - We call this reason of change responsibility
- Coupled responsibilities
  - Changes to one responsibility may impair or inhibit the class' ability to meet the others
  - Fragile design that break in unexpected ways
    - Recompilation, test, deploy, …

SINGLE RESPONSIBILITY PRINCIPLE



SINGLE RESPONSIBILITY PRINCIPLE



SINGLE RESPONSIBILITY PRINCIPLE

- What is really a responsibility?
  - An axis of change is only an axis of change if the changes actually occur
  - The context of the application is also important
    - Needless complexity

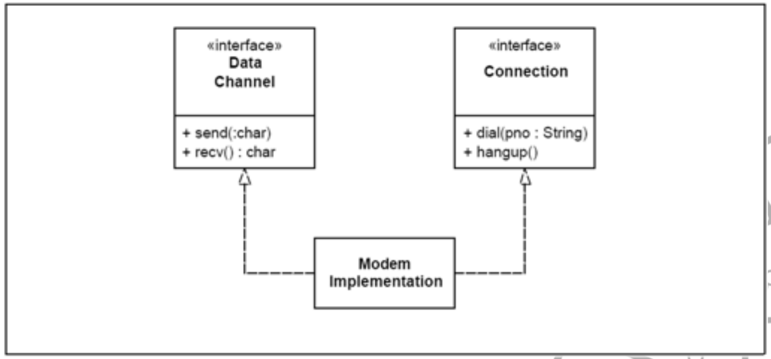
```
public interface Modem {
    public void dial(String pno);
    public void hangup();

    public void send(char c);
    public char recv();
}
```

Connection management

Data communication

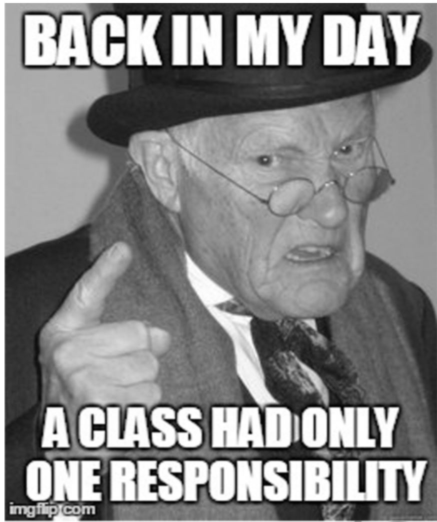
- Should these two responsibilities be separated?
  - That depends upon how the application is changing.

Single Responsibility Principle



- Eventually separate responsibilities avoids rigidity
  - They are still coupled in `ModemImplementation`, but clients don't need to worry about interface implementations

Single Responsibility Principle



Open-Close Principle

- There are many heuristics in OOD

"All member variables should be private", "Global variables should be avoided", "Using run time type identification (RTTI) is dangerous"

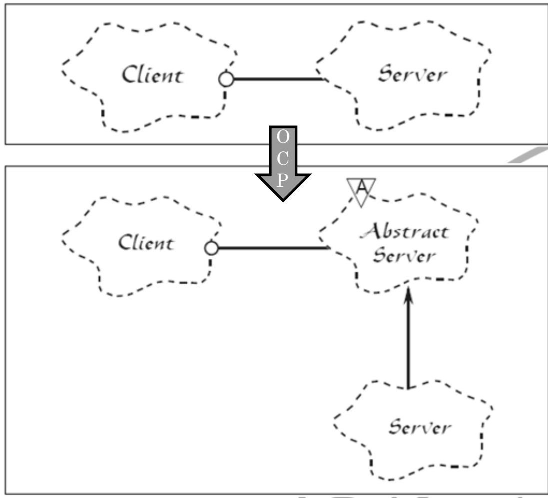
  - Software entities should be open for extension, but closed for modification
    - You extend behaviour adding new code, not changing the old
  - The Open-Close Principle underlines these heuristics

- Abstraction is the key
  - Abstract types are the fixed part, derivate classes points of extension

Open-Close Principle

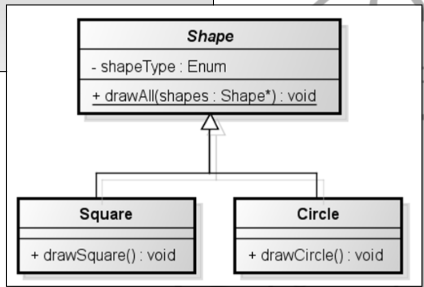
`Client` class must be changed to name the new server class.

If we want `Client` objects to use a different server class, then a new derivative of the `AbstractServer` class can be created. The `Client` class can remain unchanged.



OPEN-CLOSE PRINCIPLE

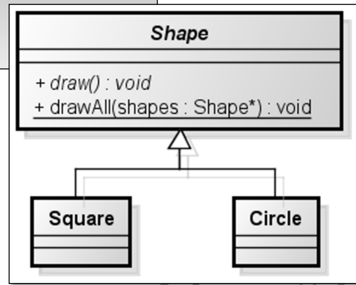
```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        switch (shape.shapeType) {
            case Square:
                ((Square) shape).drawSquare();
                break;
            case Circle:
                ((Circle) shape).drawCircle();
                break;
        }
    }
}
```



Does not conform to the open-closed principle because it cannot be closed against new kinds of shapes. If I wanted to extend this function, I would have to modify the function

OPEN-CLOSE PRINCIPLE

```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```



Solution that conforms to open-close principle. To extend the behavior of the `drawAll` to draw a new kind of shape, all we need do is add a new derivative of the `Shape` class.

- Programs conforming to OCP do not experience «cascade of changes»
  - Changes are obtained adding new code

OPEN-CLOSE PRINCIPLE

- No program can be 100% closed
  - Closure must be strategic
- Closure can be gained through abstraction
  - Using interfaces and polimorphim
    - The `draw` abstract method in the `Shape` class
- ...or can be gained in a «data-driven» fashion
  - Sometimes using information configured in external structure can be the only solution
    - What if we want to draw shapes in a specific order that depends from type!?

OPEN-CLOSE PRINCIPLE

OPEN-CLOSE PRINCIPLE

- Conventions and heuristics derived from OCP
  - Make all member variables private
    - When the member variables of a class change, every function that depends upon them must be changed
      - *Encapsulation*
  - No global variables (ever)
    - No module that depends upon a global variable can be closed against any other module that might write to that variable
      - There are very few cases that can disobey (i.e. `cin`, `cout`)
  - RTTI is dangerous
    - The `Shape` example shows the bad way to use RTTI
    - But there are also good cases…

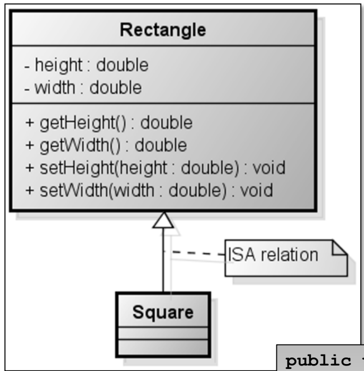
LISKOV SUBSTITUTION PRINCIPLE

- Abstraction and polymorphism
  - At the basis of OOD and OCP
    - What are the characteristics of the best inheritance hierarchies? What are the traps?
- Liskov Substitution Principle

> *Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.*

- It a special case of the real LSP ;)
  - Violating this principle means violating OCP
    - Function that uses a pointer or reference to a base class, but must know about all the derivatives of that base class.

LISKOV SUBSTITUTION PRINCIPLE



A `Square` does not need both `height` and `width` member variables. Yet it will inherit them anyway. Clearly this is wasteful.

`Square` will inherit the `setWidth` and `setHeight` functions. These functions are utterly inappropriate for a `Square`.
But, we could override them…

```java
public void setWidth(double width) {
    super.setWidth(width);
    super.setHeight(width);
}
public void setHeight(double height) {
    this.setWidth(height);
}
```


LISKOV SUBSTITUTION PRINCIPLE

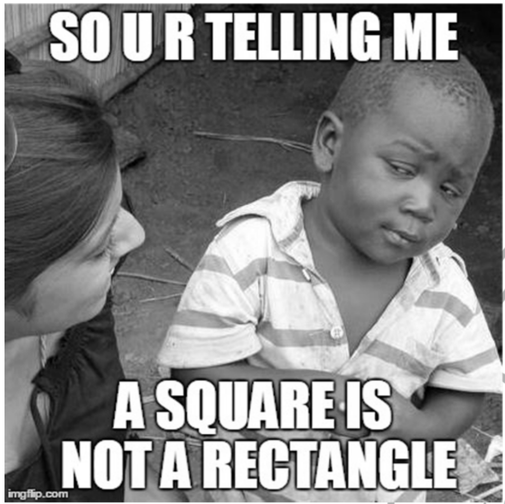
If we pass a reference to a `Square` object into this function, and the height will be changed too.

This is a clear violation of LSP. The `f` function does not work for derivatives of its arguments.

```java
public void f(Rectangle r) {
    r.setWidth(42);
}
@Test
public void testF() {
    Rectangle r = new Square();
    r.setHeight(15);
    f(r);
    // This test will not pass!!!
    assertEquals(15, r.getHeight());
}
```

- A model, viewed in isolation, can not be meaningfully validated
  - The validity of a model can only be expressed in terms of its clients

LISKOV SUBSTITUTION PRINCIPLE



LISKOV SUBSTITUTION PRINCIPLE

- What went wrong?
  - What counts is extrinsic public behavior
    - Behavior that clients depend upon
    - The relation between `Square` and `Rectangle` is not a IS-A relation in OOD
- Design by contract

> *...when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.*

  - Methods of classes declare preconditions and postconditions (invariants)

```
// Rectangle.setWidth(double w) postconditions
assert((width == w) && (height == old.height));
```


LISKOV SUBSTITUTION PRINCIPLE

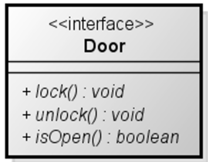
- Design by contract
  - In a derivate class preconditions must not be stronger than in the base class
    - Using base class interface a client knows only base class preconditions
  - In a derivate class postconditions must be stronger than in the base class
    - Derived class must conform to all base class prostcondition. The behaviors and outputs must not violate any of the constraints established for the base class
  - Java and JVM base languages have `assert` primitive. C++ does not have anything such this

INTERFACE SEGREGATION PRINCIPLE

- Reducing coupling means to depend upon interfaces, not implementations
  - The risk is to depend upon a «fat» or «polluted» interfaces
  - Fat interfaces are not cohesive
    - Methods can be broken up into groups of functions
    - Clients must view only the part they are interested to
- Interface Segregation Principle

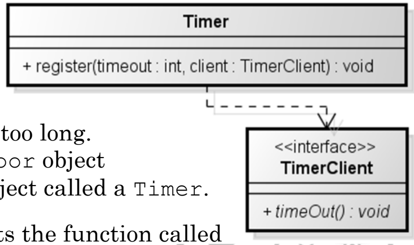
> *Clients should not be forced to depend upon interfaces that they do not use*

INTERFACE SEGREGATION PRINCIPLE



In this system there are `Door` objects that can be locked and unlocked, and which know whether they are open or closed.

Clients used this interface to managed doors.

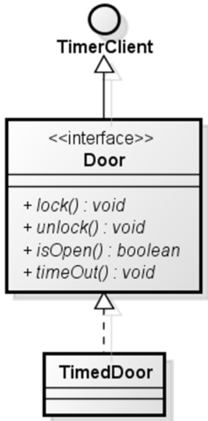


Now consider that one such implementation. `TimedDoor` needs to sound an alarm when the door has been left open for too long. In order to do this the `TimedDoor` object communicates with another object called a `Timer`.

`TimeClient` method represents the function called when the timeout expires

INTERFACE SEGREGATION PRINCIPLE

- First solution



The `Door` class now depends upon `TimerClient`. Not all varieties of `Door` need timing. Moreover, the applications that use those derivatives will have to `import` the definition of the `TimerClient` class, even though it is not used.

The interface of Door has been **polluted** with an interface that it does not require. Each time a new interface is added to the base class, that interface must be implemented in derived classes.

Default implementations violate the Liskov Substitution Principle (LSP)

INTERFACE SEGREGATION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE

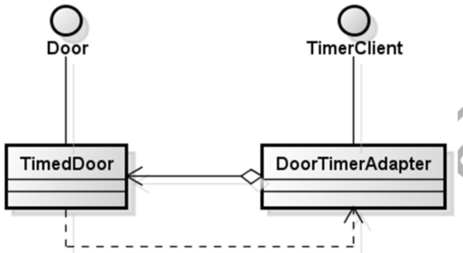
- Clients of `Door` and `TimerClient` are different
  - The interfaces should remain separate too
  - Sometimes it is the client that forces a change to an interface

```
public class Timer {
    void register(int timeout, int timeOutId, TimerClient client);
}
public interface TimerClient {
    // A change to Timer implies a change to TimerClient
    void timeOut(int timeOudId);
}
```

  - Also the `Door` interface have to be changed
    - Clients that does not need timer doors will also be affected
  - The result is a inadvertent coupling between all the clients

INTERFACE SEGREGATION PRINCIPLE

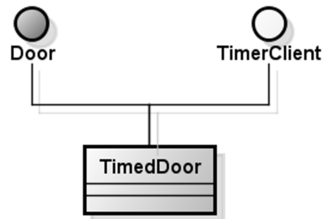
- Separation by delegation
  - *Object* form of the Adapter design pattern



    - `DoorTimerAdapter` translates a `Door` into a `TimerClient`
  - Clients of `Door` and `TimerClient` are not coupled anymore

INTERFACE SEGREGATION PRINCIPLE

- Separation through multiple inheritance
  - *Class* form of the Adapter design pattern

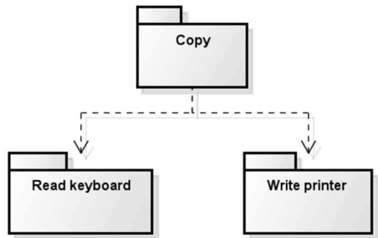


  - Client can use the same object through different and separate interfaces
    - Possible only when multiple inheritance is supported
    - Less types used wrt the solution that uses delegation

DEPENDENCY INVERSION PRINCIPLE

- Bad design often derives from degradation due to new requirement and maintanance
  - Rigidity – hard to change because every change affects to many part of the system
  - Fragility – when you make a change, unexpected parts of the system break
  - Immobility – It is hard to reuse in another application because it cannot be easily disentangled
  - TNTWIWHDI – That's not the way I would have done it
- Interdependence of the modules

DEPENDENCY INVERSION PRINCIPLE



Consider a simple program that is charged with the task of copying characters typed on a keyboard to a printer.

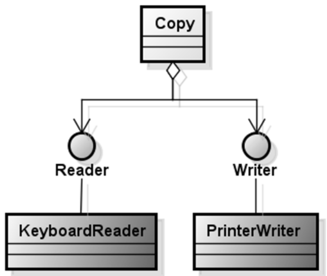
"Read keyboard" and "Write printer" are quite reusable. However the "Copy" module is not reusable in any context that does not involve keyboard and printer

Violates OCP

```
public void copy(OutputDevice dev) {
    int c;
    while ((c = readKeyboard()) != EOF)
        if (dev == PRINTER)
            writePrinter(c);
        else
            writeDisk(c);
}
```

DEPENDENCY INVERSION PRINCIPLE

- Module containing high level policy should be independent upon low level details modules



We have performed dependency inversion. The dependencies have been inverted; the "Copy" class depends upon abstractions, and the detailed readers and writers depend upon the same abstractions.

Now we can reuse the "Copy" class, independently of the "Keyboard Reader" and the "Printer Writer".

- We have use abstraction to limit dependency

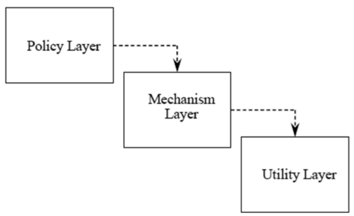
DEPENDENCY INVERSION PRINCIPLE

- Dependency Inversion Principle

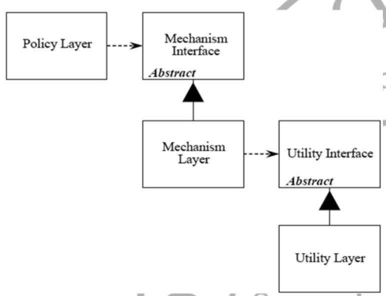
High level modules should not depend upon low level modules. Both should depend upon abstractions.
Abstractions should not depend upon details. Details should depend upon abstractions.

- Important policy decisions are in high level modules
  - It's these modules we want to be able to reuse
- Template method design pattern
- In layered application, each layer should expose a proper level of abstraction (interface)
  - A naive implementation can force wrong dependency among modules

DEPENDENCY INVERSION PRINCIPLE

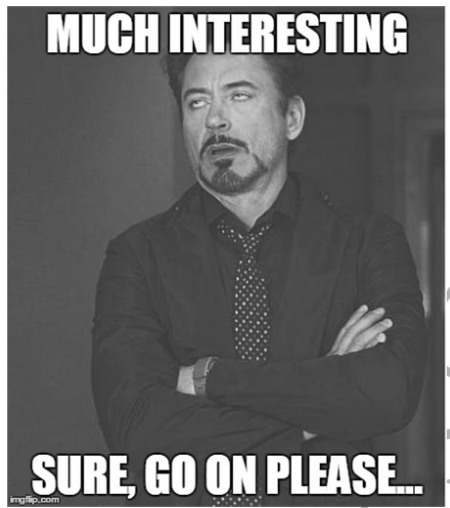


The high level policy class uses a lower level Mechanism; which in turn uses a detailed level utility class. The Policy Layer is sensitive to changes all the way down in the Utility Layer.



Each of the lower level layers are represented by an abstract class. Each of the higher level classes uses the next lowest layer through the abstract interface. Thus, none of the layers depends upon any of the other layers.

DEPENDENCY INVERSION PRINCIPLE



REFERENCES

- The Principles of OOD
  http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- Chap. 8 “The Single-Responsibility Principle (SRP)”, Agile Principles, Patterns, and Practices in C#, Robert C. Martin, 2006, Prentice Hall
- Chap. 9 “The Open/Closed Principle (OCP)”, Agile Principles, Patterns, and Practices in C#, Robert C. Martin, 2006, Prentice Hall
- Chap. 10 “The Liskov Substitution Principle (LSP)”, Agile Principles, Patterns, and Practices in C#, Robert C. Martin, 2006, Prentice Hall
- Chap. 11 “The Dependency-Inversion Principle (DIP)”, Agile Principles, Patterns, and Practices in C#, Robert C. Martin, 2006, Prentice Hall
- Chap. 12 “The Interface Segregation Principle (ISP)”, Agile Principles, Patterns, and Practices in C#, Robert C. Martin, 2006, Prentice Hall

REFERENCES

- Single-Responsibility Principle done right
  http://blog.rcard.in/solid/srp/programming/2017/12/31/srp-done-right.html
- The Secret Life of Objects: Inheritance
  http://blog.rcard.in/design/programming/oop/fp/2018/07/27/the-secret-life-of-objects-part-2.html

GITHUB REPOSITORY

https://github.com/rcardin/swe