S.O.L.I.D. Principles

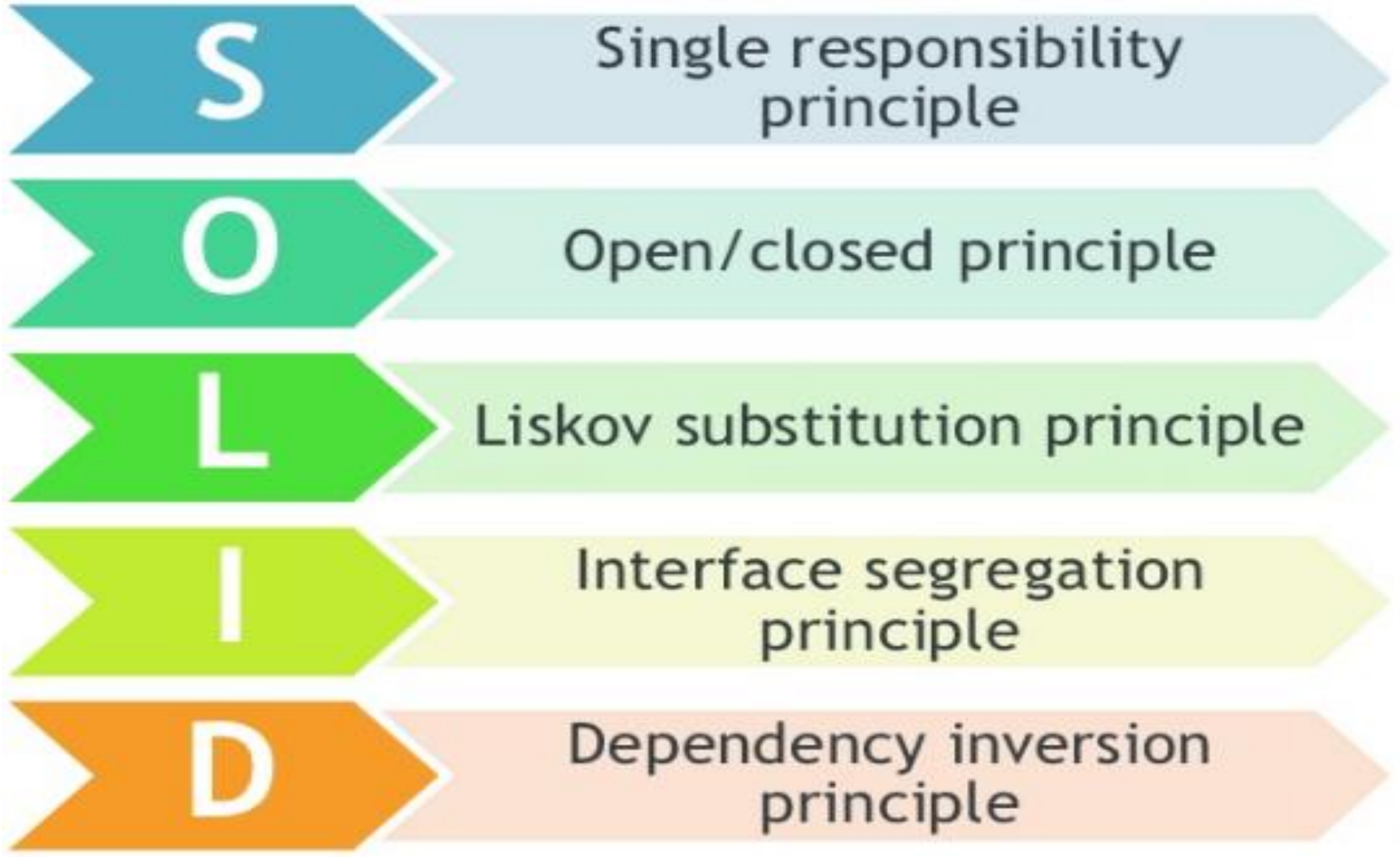
SRP

OCP

LSP

ISP

DIP


S
Single responsibility principle
O
Open/closed principle
L
Liskov substitution principle
I
Interface segregation principle
D
Dependency inversion principle

SRP: The Single Responsibility Principle

Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class. ~Wikipedia

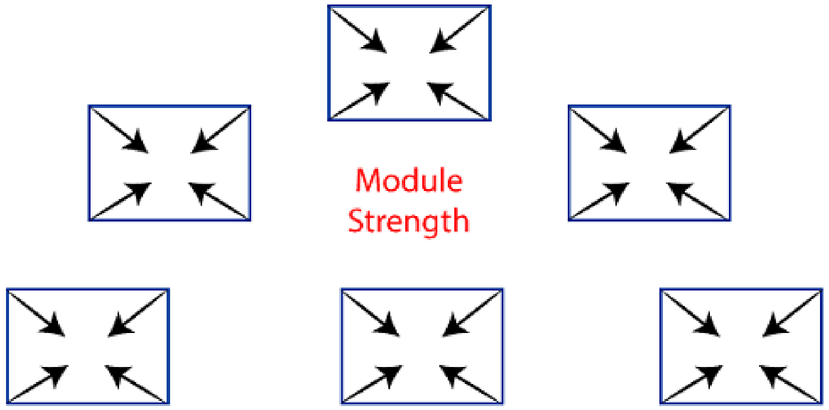
There should never be more than one reason for a class to change.

~Robert C. “Uncle Bob” Martin

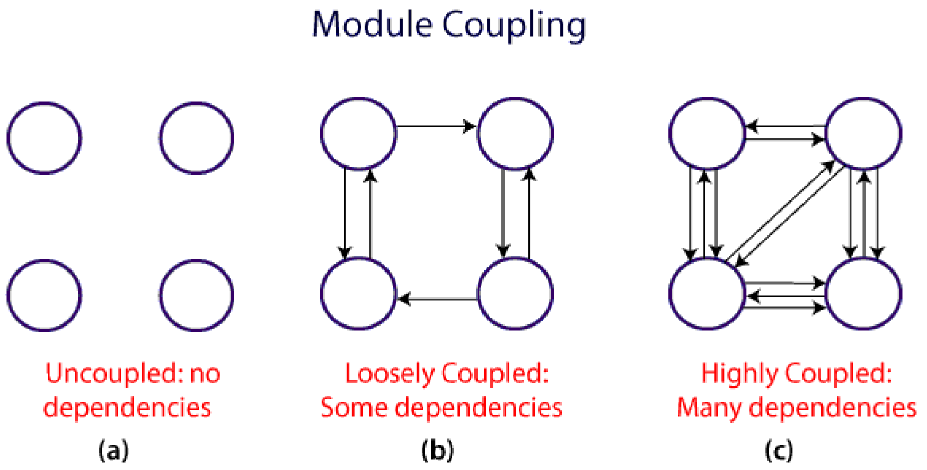
A module should have one, and only one, reason to change.

Cohesion and Coupling

- ✓**Cohesion:** how strongly-related and focused are the various responsibilities of a module
- ✓ A measure of how closely related the members (classes, methods, functionality within a method) of a module are to the other members of the same module. It is desirable to **increase cohesion** as that indicates that a module has a very specific task and does only that task.
- ✓**Coupling:** the degree to which each program module relies on each one of the other modules
- ✓A measure of how much a module (package, class, method) relies on other modules. It is desirable to **reduce coupling**, or reduce the amount that a given module relies on the other modules of a system



Cohesion= Strength of relations within Modules


Module Coupling
Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

Strive for low coupling and high cohesion!

What is a Responsibility?

- ✓“a reason to change”
- ✓A difference in usage scenarios from the client’s perspective

Responsibilities are Axes of Change

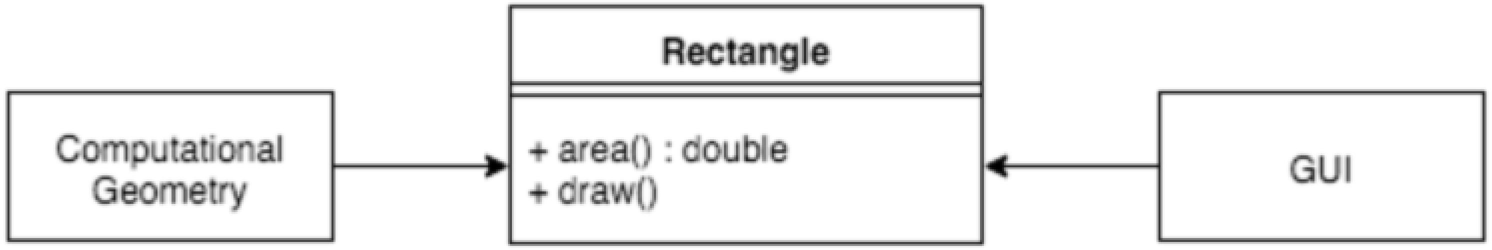
- ✓ Requirements changes typically map to responsibilities
- ✓ More responsibilities == More likelihood of change
- ✓ Having multiple responsibilities within a class couples together these responsibilities

The more classes a change affects, the more likely the change will introduce errors.

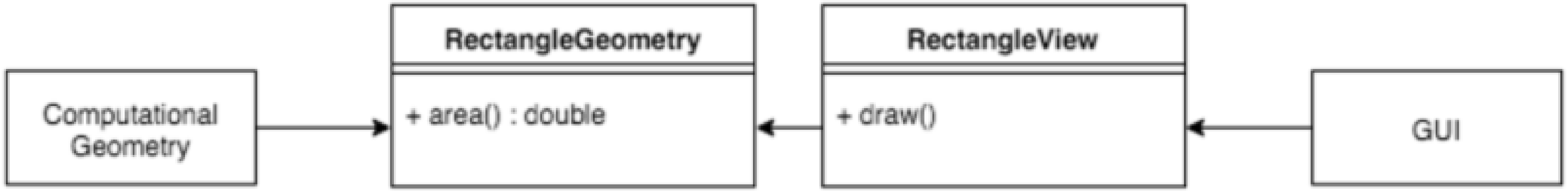
SRP: The Single Responsibility Principle

- ✓each class should have **one responsibility**, one **single purpose**.
- ✓This means that a class will do only one job, which leads us to conclude it should have only **one reason to change**.
- ✓if we have a class that we change a lot, and for different reasons, then this class should be broken down into more classes, each handling a single concern. Surely, if an error occurs, it will be easier to find.

SRP EXAMPLE



- ✓ Here , a rectangle class that has two methods; Area () & Draw ().
  - Area () has the responsibility to return the area of a rectangle
  - Draw () has the responsibility to draw the rectangle itself
- ✓ if there are changes in the GUI, we must modify the Rectangle class,
- ✓ divide the class in two, so that each class has a unique responsibility.



Now, One will be responsible for calculating and another one for painting.

More Example

Consider a class for trading application that buys and sells. And we implement a class, named **Transaction** , has two responsibilities: ***buying*** and ***selling***.

```
// Class to handle both Buy and Sell actions
class Transaction {
        // Method to Buy, implemented in Transaction class
        private void Buy(String stock, int quantity, float price){
                // Buy stock functinality implemented here
        }
        // Method to Sell, implemented in Transaction class
        private void Sell(String stock, int quantity, float price){
        // Sell stock functionality implemented here
        }
}
```

if the requirements for either method change it would necessitate a change in the **Transaction** class. In other words, **Transaction** has multiple responsibilities.

More Example

Refactoring these methods into separate classes would be one approach to applying the Single Responsibility Principle to this application.

```
class Transaction{
        private void Buy(String stock, int quantity, float price){
                Buy.execute(stock, quantity, price);
        }
        private void Sell(String stock, int quantity, float price){
        Sell.execute(stock, quantity, price);
        }
}
class Buy{
        static void execute(String ticker, int quantity, float price){
        }
}
class Sell{
        static void execute(String ticker, int quantity, float price){
        }
}
```

- ✓ Now, Each of those refactored classes is assigned with a single responsibility.
- ✓ The **Transaction** class can still perform two separate tasks but is no longer responsible for their implementation.

SRP - Summary

- ✓ Following SRP leads to lower coupling and higher cohesion
- ✓ Many small classes with distinct responsibilities result in a more flexible design
- ✓ When we split responsibilities between smaller methods and classes, usually the system becomes easier to learn overall.
- ✓ It becomes possible to reuse components when they have a single, narrow responsibility.
- ✓ Most methods with a narrow responsibility shouldn’t have side effects
- ✓ t’s easier to write and maintain tests for methods and classes with focused, independent concerns
- ✓ Having methods with single responsibilities also helps to quickly pinpoint performance problems
- ✓ the more focused responsibilities of the components we make, the more code we will need to write.
- ✓ it takes more time and effort to develop the first version of the application with finely separated responsibilities than with larger components.

OCP: Open/Closed Principle

- The Open / Closed Principle states that software entities (classes, modules, functions, etc.) should be open for **extension**, but closed for **modification**

A software artifact should be open for extension but closed for modification.

The Open / Closed Principle

- ✓Open to Extension
  - ➢ New behavior can be added in the future
- ✓Closed to Modification
  - ➢ Changes to source or binary code are not required
- ✓ The "closed" part of the rule states that once a module has been developed and tested, the code should only be changed to correct bugs.
- ✓ The "open" part says that you should be able to extend existing code in order to introduce new functionality.

Dr. Bertrand Meyer originated the OCP term in his 1988 book, Object Oriented Software Construction

The Open / Closed Principle

This approach allows **extended** functionality via concrete implementation classes without necessitating changes to **base classes**. In other words, a developer **can add new functionality without changing the existing code** of related functions, classes, and methods!

OCP Example

Let's say that we've got a Rectan

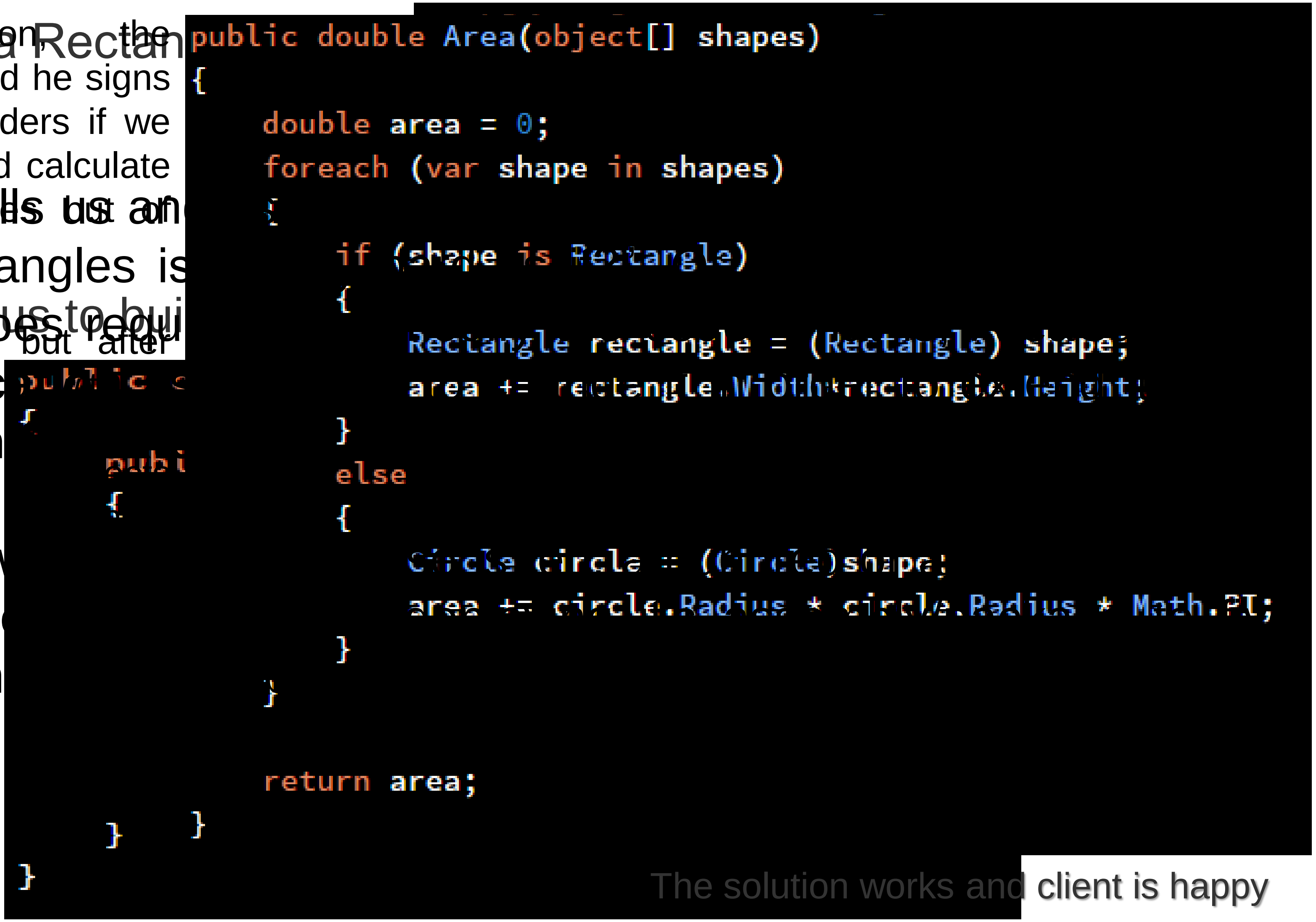
We present our solution, the AreaCalculator class to client and he signs us his praise. But he also wonders if we couldn't extend it so that it could calculate the area of not only rectangles but of circles as well.

Only, a week later he calls us an calculate the area of triangles is scenario it isn't but it does requ isn't closed for modific other words: it isn't open

Now our customer wants us to bui a collection of rectangles

That complicates things a bit but after some considering we come up with a solution where we change method to accept a collection instead of the more specific type.

In a real world scenario w larger and modifying th different servers that can



```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

```
public c
{
    publi
    {

    }
}
```

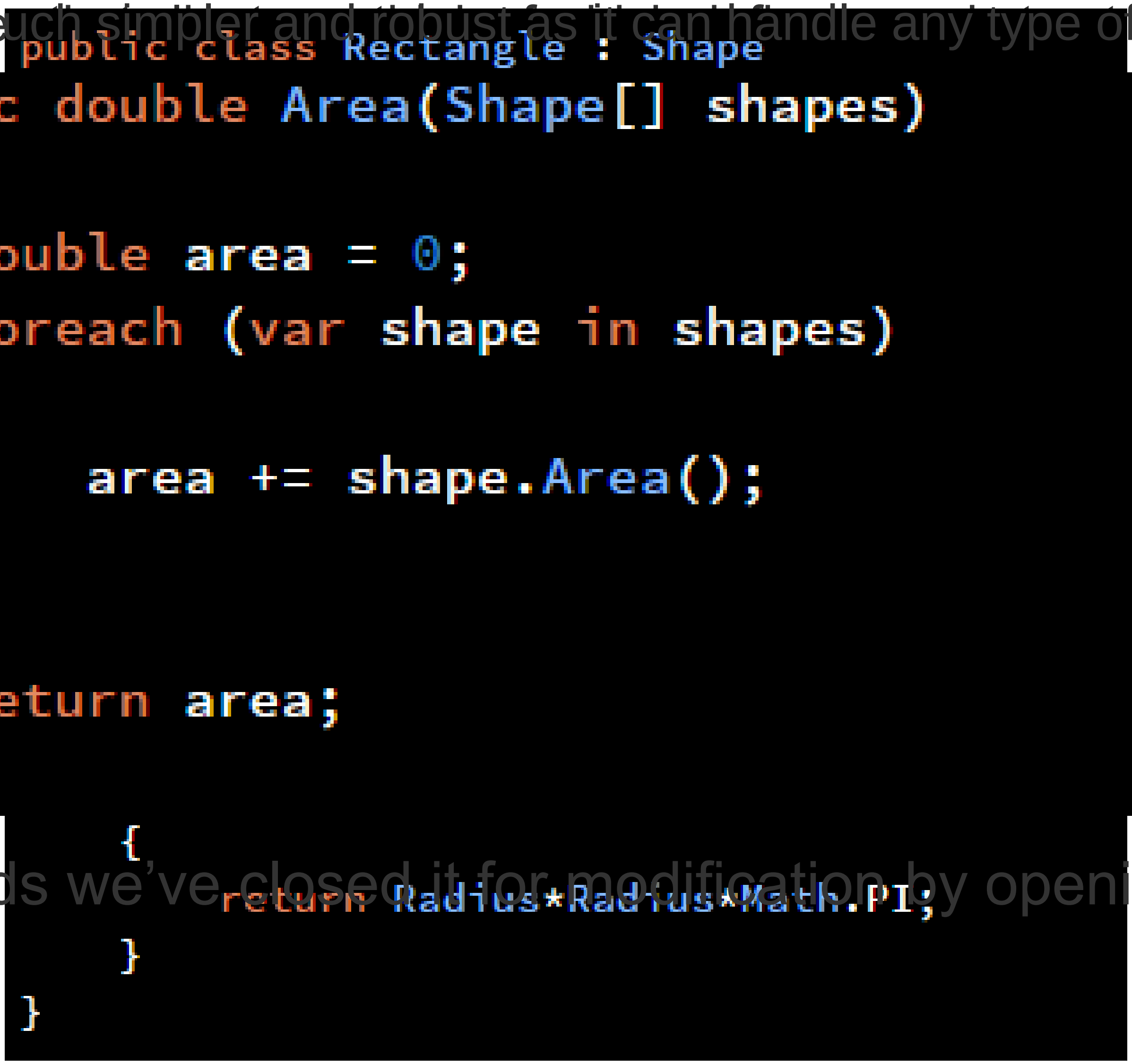
The solution works and client is happy

A SOLUTION ABIDES BY THE OPEN/CLOSED PRINCIPLE

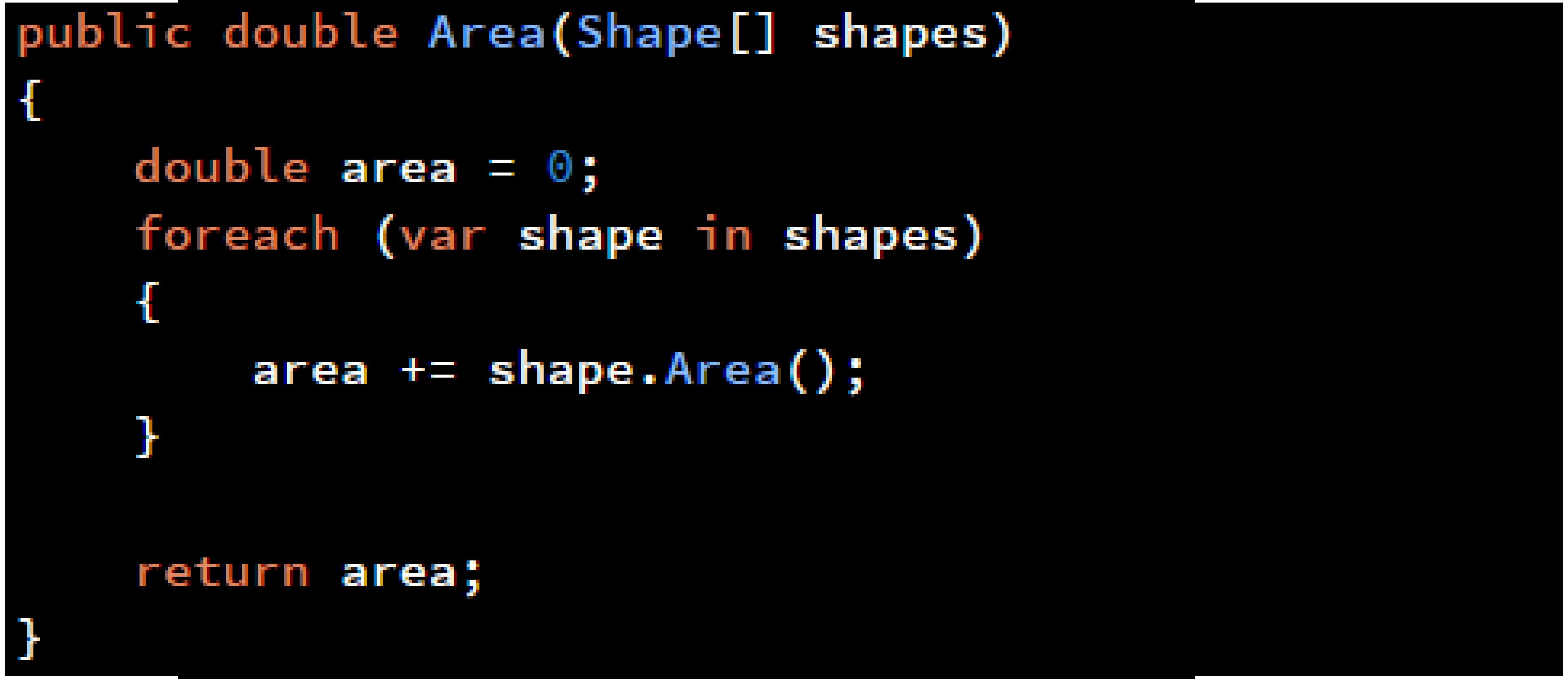
One way of solving this puzzle would be to create a base class for both rectangles and circles as well as any other shape [illegible] for calculating it's area.

Inheriting from Shape the Rectangle and Circle classes now looks like this:

As we've moved the responsibility of actually calculating the area away from AreaCalculator's Area method it is now much simpler and robust as it can handle any type of Shape that we throw at it.



```
public class Rectangle : Shape
[illegible]
    {
        return Radius*Radius*Math.PI;
    }
}
```



```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

In other words we've closed it for modification by opening it up for extension.

When do we apply OCP?

Experience Tells You

- ✓ If you know from your own experience in the problem domain that a particular class of change is likely to recur, you can apply OCP up front in your design

Otherwise – “Fool me once, shame on you; fool me twice, shame on me”

✓Don’t apply OCP at first

✓If the module changes once, accept it.

✓If it changes a second time, refactor to achieve OCP

Remember

✓**OCP adds complexity to design**

✓**No design can be closed against all changes**

LSP: The Liskov Substitution Principle

Child classes should never break the parent class' type definitions.

The concept of this principle was introduced by Barbara Liskov in a 1987

Their original definition is as follows:

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

This leads us to the definition given by Robert C. Martin:

Subtypes must be substitutable for their base types.

SUBSTITUTABILITY

Child classes must not:

1) Remove base class behavior
2) Violate base class invariants

Substitution Principle: If B is a subtype of A, everywhere the code expects an A, a B can be used instead and the program still satisfies its specification

In general, OOP teaches use of IS-A to describe child classes’ relationship to base classes

BUT

LSP suggests that IS-A should be replaced with IS-SUBSTITUTABLE-FOR

Let's do an example!

Let’s check the below code:

```
class Transportation
{
        String name;
        String getName() { ... }
        void setName(String n) { ... }
        double speed;
        double getSpeed() { ... }
        void setSpeed(double d) { ... }
        Engine engine;
        Engine getEngine() { ... }
        void setEngine(Engine e) { ... }
        void startEngine() { ... }
}
class Car extends Transportation
{
        @Override
        void startEngine() { ... }
}
```

There is no problem here, right? A car is definitely a **transportation**, and here we can see that it overrides the startEngine()   method of its superclass.

Let’s add another **transportation**:

```
class Bicycle extends Transportation
{
        @Override
        void startEngine() /*problem!*/
}
```

Everything isn’t going as planned now!

a bicycle is a transportation , however, it does not have an engine so the method startEngine() cannot be implemented.

Let's do an example!

We can refactor our **Transportation** class as follows:

```
class Trasportation
{
   String name;
   String getName() { ... }
   void setName(String n) { ... }
   double speed;
   double getSpeed() { ... }
   void setSpeed(double d) { ... }
}
```

we can extend Transportation for non-motorized devices.

```
class TransportWithoutEngine extends Transportation
{
        void startMoving() { ... }
}
```

Now we can extend Transportation for motorized devices.

```
class TransportWithEngine extends Transportation
{
   Engine engine;
   Engine getEngine() { ... }
   void setEngine(Engine e) { ... }
   void startEngine() { ... }
}
```

And Bicycle class is also in compliance with the Liskov Substitution Principle.

Now,Car class becomes more specialized, while adhering to the Liskov Substitution Principle.

```
class Bicycle extends TransportWithoutEngine
{
        @Override
        void startMoving() { ... }
}
```

LET'S DO ANOTHER EXAMPLE!

Now we want to make a method that will let us give these cuties some treats. Let's make this method and call it GiveTreatTo:

let's make an Animal superclass, and a Dog and Cat subclass and capture their favorite kinds of food.

```java
public static void GiveTreatTo(Animal animal) {
  String msg = "You fed the " + animal.getClass().getSimpleName() + " some "  + animal.favoriteFood;
  System.out.println(msg);
}
```

```java
public static class Dog extends Animal {
  public Dog(String favoriteFood) {
    super(favoriteFood);
  }
}

public static class Cat extends Animal {
  public Cat(String favoriteFood) {
    super(favoriteFood);
  }
}
```

Here GiveTreatTo takes any Animal as a parameter. Since our Animal constructors assign the animal's favorite food, we can pretty much count on that data always being there.

This means we don't have to make a method for each animal, i.e., GiveTreatToDog and GiveTreatToCat. Because we implemented LSP, we have one method.

Let's do another example!

Let's see it in action:

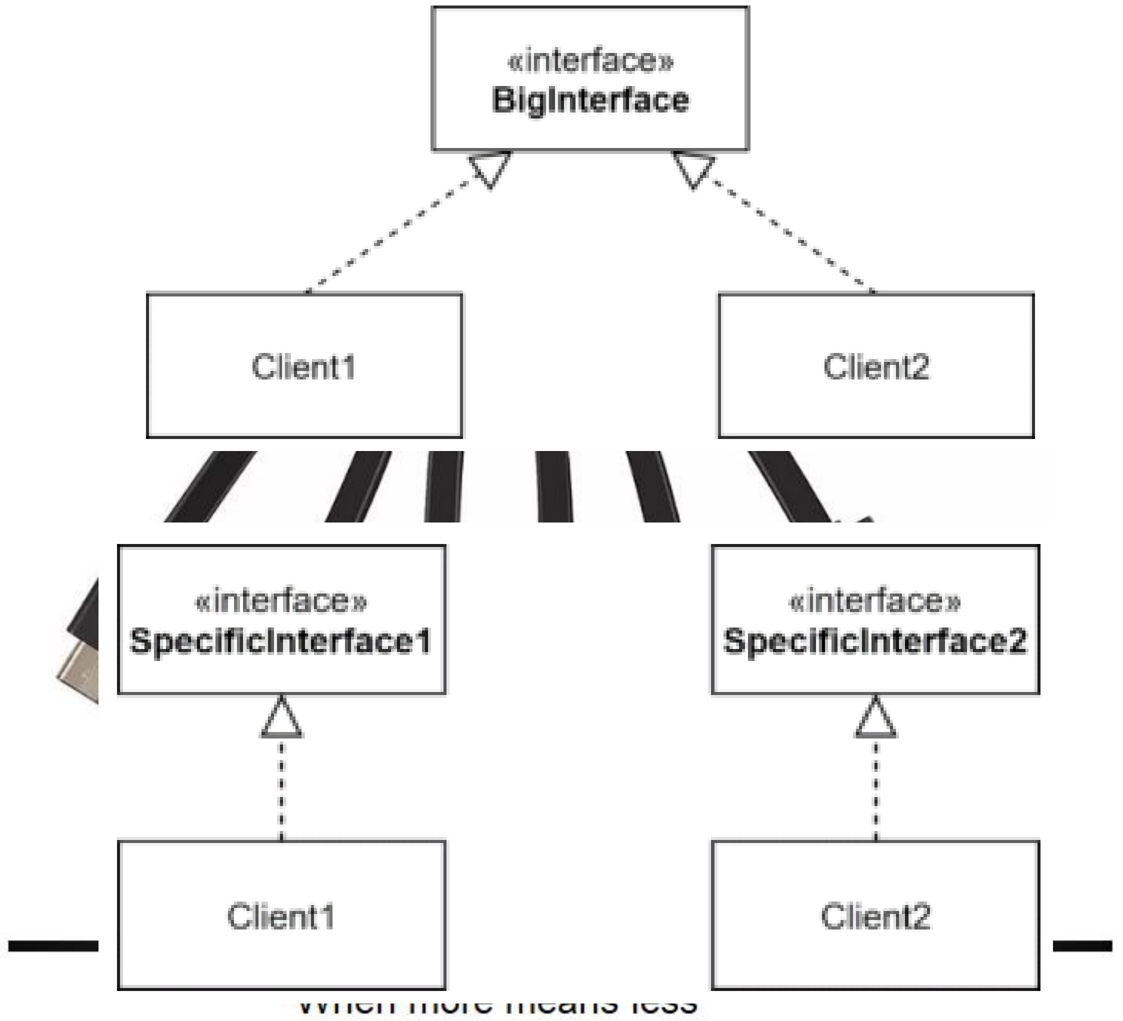
```
public static void main(String[] args) {
  Dog rover = new Dog("bacon");
  Cat bingo = new Cat("fish");

  GiveTreatTo(rover);
  GiveTreatTo(bingo);
}
```

if we properly implemented the LSP, this program should run just fine. Let's check the output:

```
You gave the Dog some bacon
You gave the Cat some fish
```

INTERFACE SEGREGATION PRINCIPLE

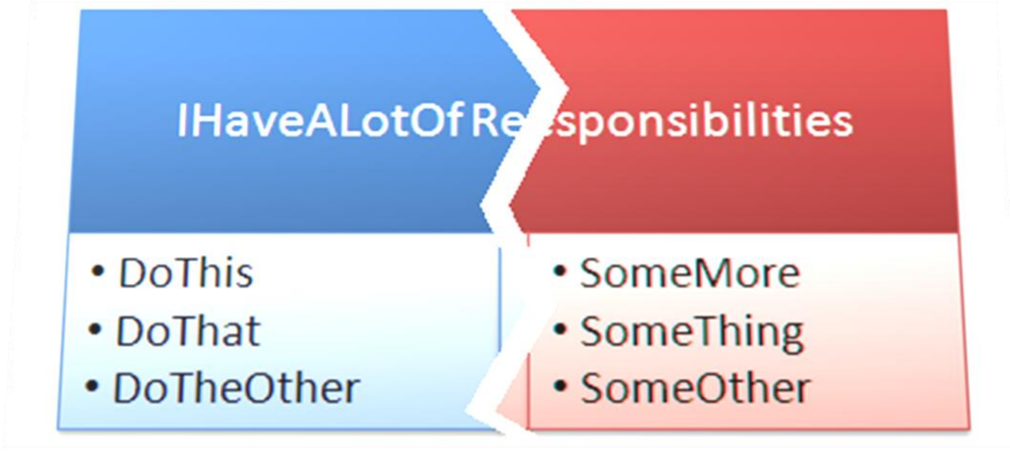
- ✓ Clients should not be forced to depend on methods they do not use.
- ✓ This means the methods of an interface can be broken up into multiple groups, where each group serves a different client.
- ✓ In other words, one group is there for one client, while the other group is there for another one.
- ✓ is very much related to the Single Responsibility Principle
- ✓ According to the interface segregation principle, you should break down "fat" interfaces into more granular and specific ones.



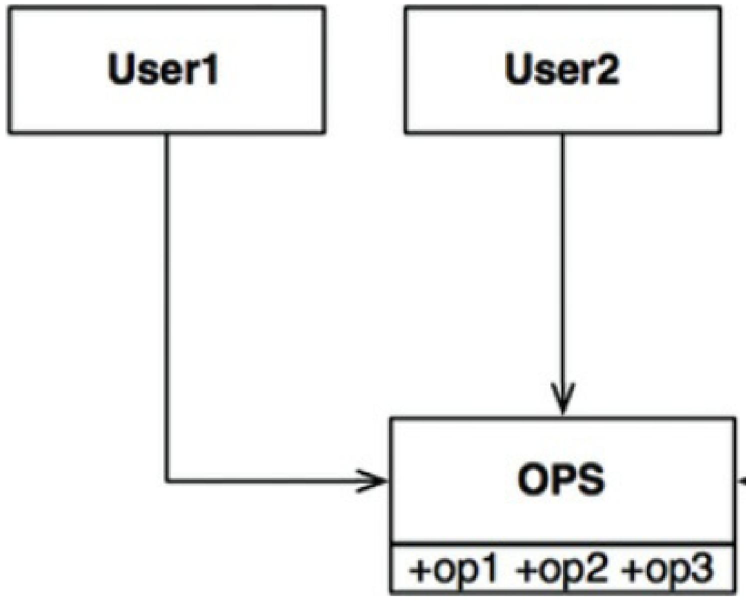
Interface Segregation Principle

"This principle deals with the disadvantages of 'fat' interfaces. Classes that have 'fat' interfaces are classes whose interfaces are not cohesive. In other words, the interfaces of the class can be broken up into groups of member functions. Each group serves a different set of clients. Thus some clients use one group of member functions, and other clients use the other groups."

- Robert Martin

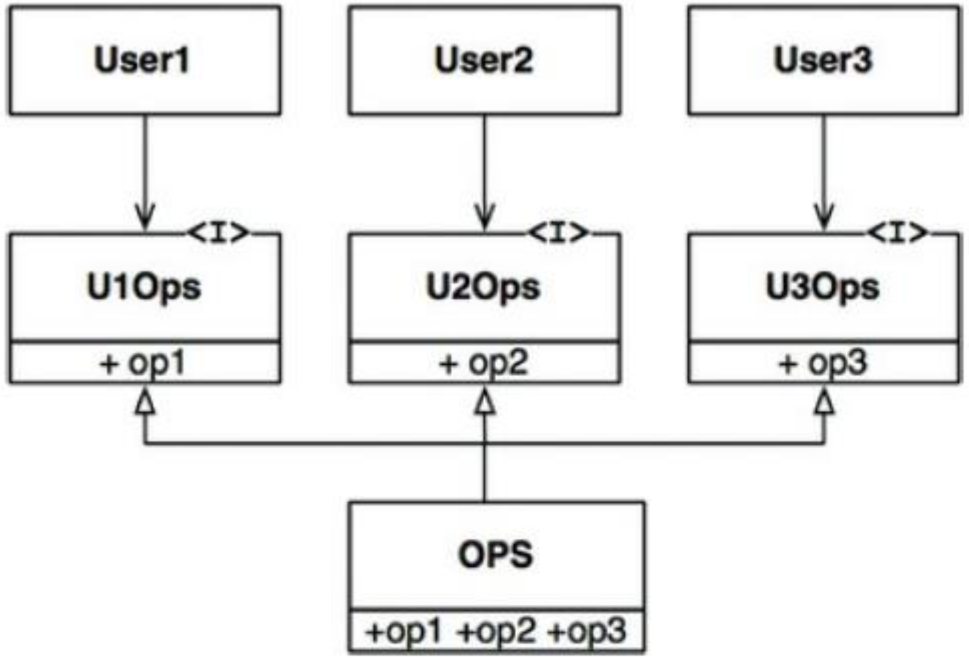


Interface Segregation Principle



assume that `User1` uses only `op1`, `User2` uses only `op2`, and `User3` uses only `op3`.

the source code of `User1` will inadvertently depend on `op2` and `op3`, even though it doesn't call them. This dependence means that a change to the source code of `op2` in `OPS` will force `User1` to be recompiled and redeployed, even though nothing that it cared about has actually changed.



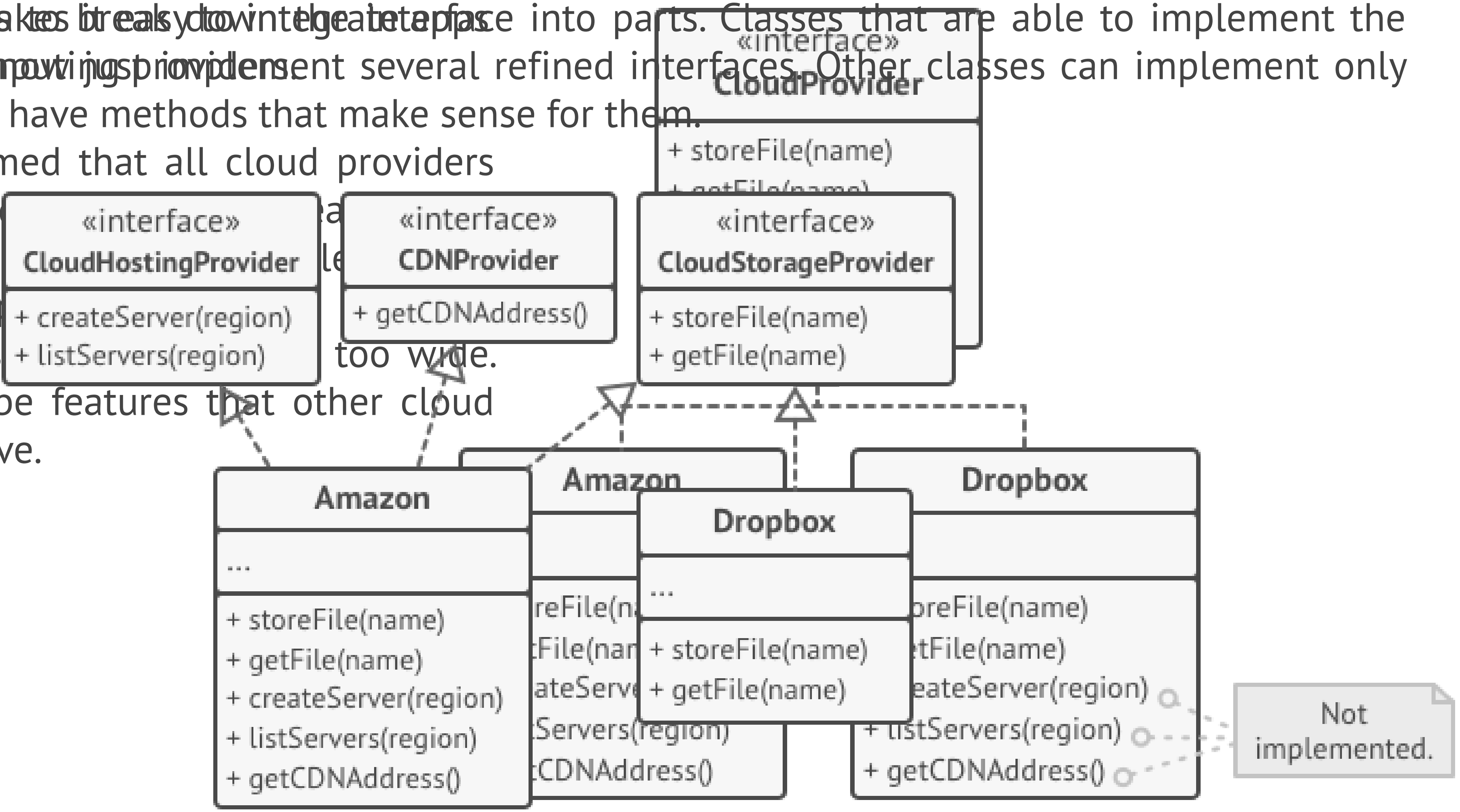
Now the source code of `User1` will depend on `U1Ops`, and `op1`, but will not depend on `OPS`. Thus a change to `OPS` that `User1` does not care about will not cause `User1` to be recompiled and redeployed.

Let's do another example!

Create a library that makes it easy to integrate apps with various cloud computing providers.

The better approach is to break down the interface into parts. Classes that are able to implement the original interface can now just implement several refined interfaces. Other classes can implement only those interfaces which have methods that make sense for them.

At the time you assumed that all cloud providers have the same broa… Amazon. But when … support for another … most of the interfaces … too wide. Some methods describe features that other cloud providers just don't have.



one bloated interface is broken down into a set of more granular interfaces

not all clients can satisfy the requirements of the bloated interface.

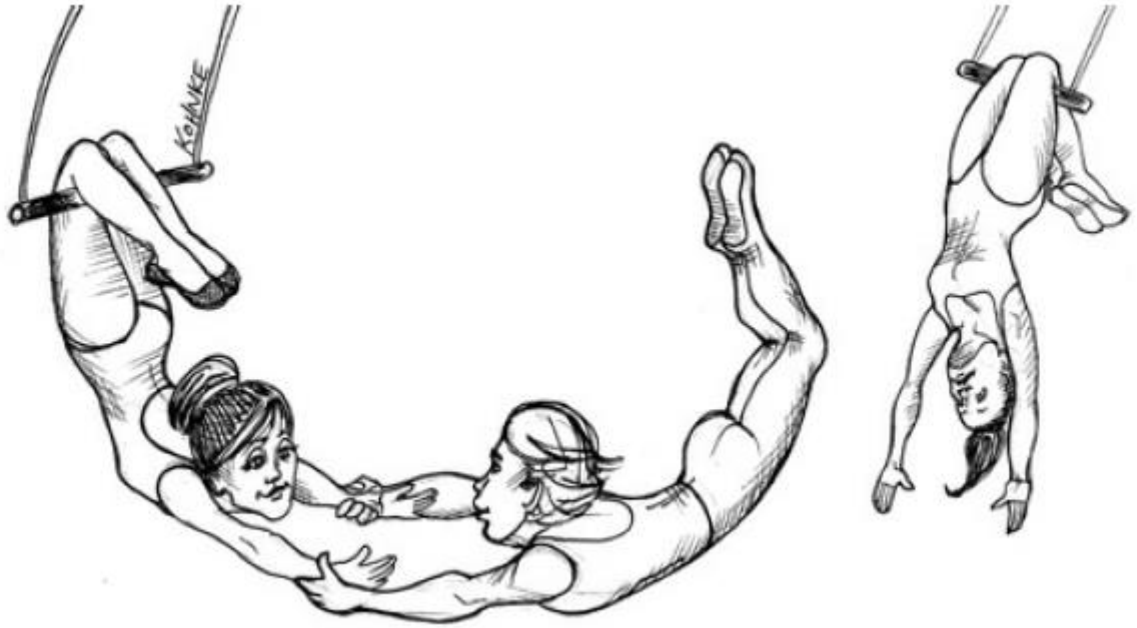
When do we fix ISP?

- ✓Once there is pain
  - ➢If there is no pain, there’s no problem to address.

In fat interfaces, there are too many operations, but for most objects, these operations are not used.

Dependency Inversion Principle

- ✓ High level modules shall not depend on low-level modules. Both shall depend on abstractions.
- ✓ Abstractions shall not depend on details. Details shall depend on abstraction pain, there’s no problem to address.



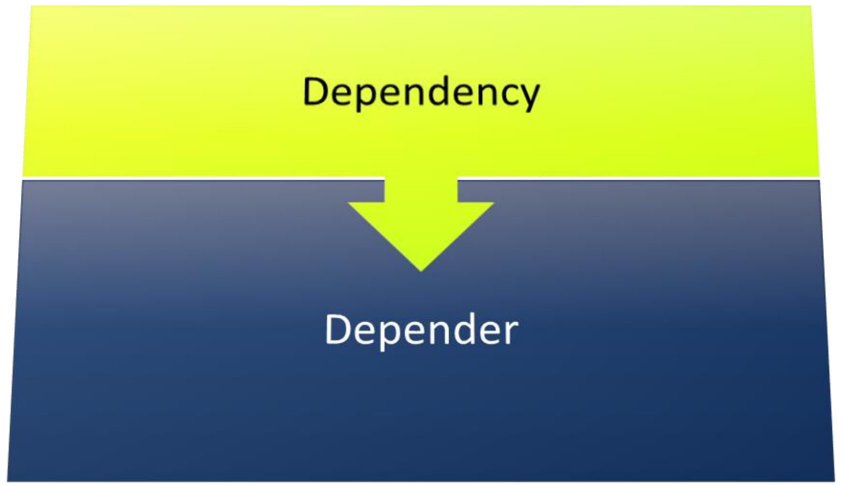
most flexible systems are those in which source code dependencies refer only to abstractions.

Dependency Inversion Principle

What is it that makes a design rigid, fragile and immobile?

It is the interdependence of the modules within that design. A design is rigid if it cannot be easily changed. Such rigidity is due to the fact that a single change to heavily interdependent software begins a cascade of changes in dependent modules.”

- Robert Martin



WHAT IS DEPENDENCY

A dependency is something - another class, or a value - that your class depends on, or requires. For example:

- ✓ If a class requires some setting, it might call *ConfigurationManager.AppSettings["someSetting"].* *ConfigurationManager* is a dependency. The class has a dependency on *ConfigurationManager*.
- ✓ If you write two classes - let's say *Log* and *SqlLogWriter* - and *Log* creates or requires an instance of *SqlLogWriter*, then *Log* depends on *SqlLogWriter*. *SqlLogWriter* is a dependency.

DEPENDENCIES….

- ✓ Framework
- ✓ Third Party Libraries
- ✓ Database
- ✓ File System
- ✓ Email
- ✓ Web Services
- ✓ System Resources (Clock)
- ✓ Configuration
- ✓ The new Keyword
- ✓ Static methods
- ✓ Thread.Sleep
- ✓ Random

High Level Modules and Low Level Modules

Forget modules and think about classes.

The principle says that “both should depend on abstractions.” means that when it comes to applying the principle, the difference between high level and low level doesn’t matter.

What Are Abstractions?

- ✓ Abstractions are generally interfaces, abstract classes, and delegates (in .NET terms.)
- ✓ We call it an “abstraction” because it’s an indirect representation of a concrete class or method.
- ✓ Interfaces are used most commonly.
- ✓ applying dependency inversion means that we depend on interfaces.

DIP: Let's look at a very simple example

- ✓ Suppose, We have the manager class which is a high level class, and the low level class called Worker.
- ✓ We need to add a new module to our application to model the changes in the company structure determined by the employment of new specialized workers.
- ✓ We created a new class SuperWorker for this.
- ✓ Let's assume the Manager class is quite complex, containing very complex logic. And now we have to change it in order to introduce the new SuperWorker.
- ✓ Let's see the disadvantages:
  - ➢ we have to change the Manager class (remember, it is a complex one and this will involve time and effort to make the changes).
  - ➢ some of the current functionality from the manager class might be affected.
  - ➢ the unit testing should be redone.

DIP: Let's look at a very simple example

```
// Dependency Inversion Principle - Bad example
class Worker {
public void work() {
		// ....working
	}
}

class Manager {
Worker worker;

public void setWorker(Worker w) {
		worker = w;
}

public void manage() {
		worker.work();
	}
}

class SuperWorker {
public void work() {
		//.... working much more
	}
}
```

The situation would be different if the application had been designed following the Dependency Inversion Principle.

It means we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface.

When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it.

No additional changes in the existing classes.

DIP: Let's look at a very simple example

```
// Dependency Inversion Principle - Good example
interface IWorker {
public void work();
}

class Worker implements IWorker{
public void work() {
// ....working
}
}

class SuperWorker implements IWorker{
public void work() {
//.... working much more
}
}

class Manager {
IWorker worker;

public void setWorker(IWorker w) {
worker = w;
}

public void manage() {
worker.work();
}
}
```

This code supports the Dependency Inversion Principle. In this new design a new abstraction layer is added through the IWorker Interface. Now the problems from the above code are solved(considering there is no change in the high level logic):

- ✓ Manager class doesn't require changes when adding SuperWorkers.
- ✓ Minimized risk to affect old functionality present in Manager class since we don't change it.
- ✓ No need to redo the unit testing for Manager class.

COMPONENT PRINCIPLES

- ✓ If SOLID principles tell us how to arrange the bricks into walls, then the component principles tell us how to arrange the rooms into buildings.
- ✓ Large software systems, like large buildings, are built out of smaller components.
- ✓ Components are the units of deployment.
- ✓ smallest entities that can be deployed as part of a system.
- ✓ In Java, they are jar files. In Ruby, they are gem files. In .Net, they are DLLs.
- ✓ well-designed components always retain the ability to be independently deployable and, therefore, independently developable.

COMPONENT COHESION

Which classes belong in which components?

Three principles of component cohesion:

- **REP:** The Reuse/Release Equivalence Principle
- **CCP:** The Common Closure Principle
- **CRP:** The Common Reuse Principle

THE REUSE/RELEASE EQUIVALENCE PRINCIPLE

Classes and modules that are grouped together into a component should be releasable together.

People who want to reuse software components cannot, and will not, do so unless those components are tracked through a release process and are given release numbers.

THE COMMON CLOSURE PRINCIPLE

Gather into components those classes that change for the same reasons and at the same times.

Separate into different components those classes that change at different times and for different reasons

Just as the SRP says that a class should not contain multiples reasons to change, so the Common Closure Principle (CCP) says that a **component should not have multiple reasons to change**.

Gather together those things that change at the same times and for the same reasons. Separate those things that change at different times or for different reasons.

THE COMMON REUSE PRINCIPLE

Don’t force users of a component to depend on things they don’t need.

CRP is another principle that helps us to decide which classes and modules should be placed into a component. It states that classes and modules that tend to be reused together belong in the same component.

CRP says that classes that are not tightly bound to each other should not be in the same component.

COMPONENT COUPLING

Relationships between components

Three principles of component Coupling:

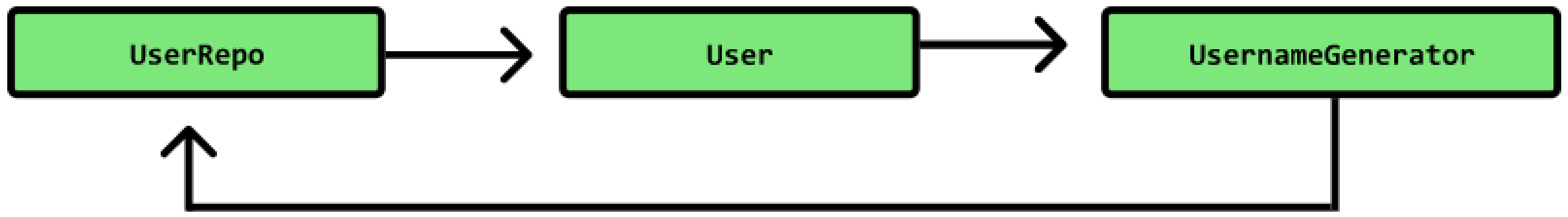
- The Acyclic Dependencies Principle
- The Stable Dependencies Principle
- The Stable Abstractions Principle

THE ACYCLIC DEPENDENCIES PRINCIPLE

Allow no cycles in the component dependency graph.

Design your software in such a way that the dependencies between your components do not form a cycle.

dependency graph of components should have no cycles



The above example demonstrates a violation of ADP. A cycle exists where **UsernameGenerator** relies on **UserRepo**, which relies on **User**, which relies on **UsernameGenerator**.

How to fix it

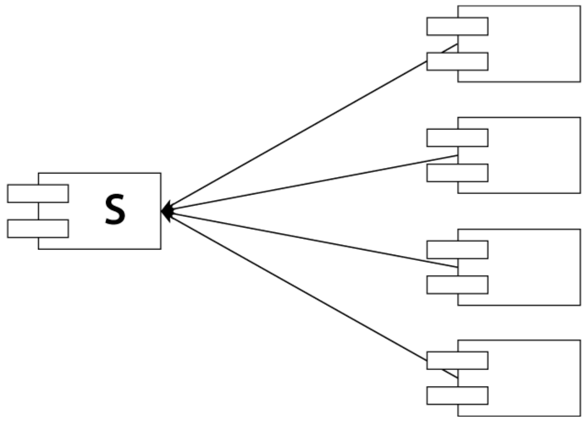
THE STABLE DEPENDENCIES PRINCIPLE

Depend in the direction of stability.

What is stability.

Stability is related to the amount of work required to make a change.
"**Stable**" roughly means "**hard to change**", whereas "**instable**" means "**easy to change**"

A component with lots of incoming dependencies is very stable because it requires a great deal of work to reconcile any changes with all the dependent components.



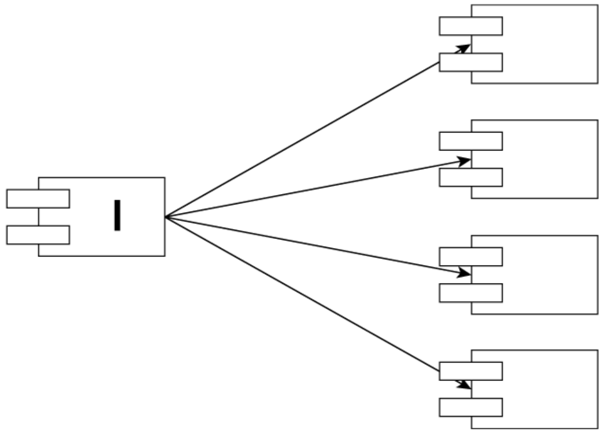
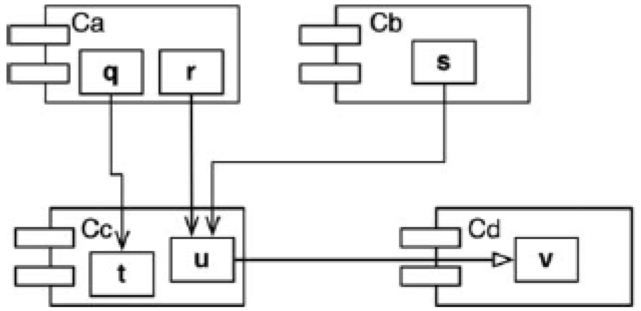


Figure shows S, which is a stable component. Four components depend on X, so it has four good reasons not to change. We say that S is *responsible* to those four components. Conversely, S depends on nothing, so it has no external influence to make it change. We say it is *independent*.

CALCULATING COMPONENTS STABILITY

is to count the number of dependencies that enter and leave that component.

Fan-in: Incoming dependencies. This metric identifies the number of classes outside this component that depend on classes within the component.
• *Fan-out*: Outgoing dependencies. This metric identifies the number of classes inside this component that depend on classes outside the component.
• *I*: Instability: *I* = *Fan-out* / (*Fan-in* + *Fan-out*). This metric has the range [0, 1]. 0 indicates a maximally stable component. 1 indicates a maximally unstable component.



Let's say we want to calculate the stability of the component `Cc`. We find that there are three classes outside `Cc` that depend on classes in `Cc`. Thus, *Fan-in* = 3. Moreover, there is one class outside `Cc` that classes in `Cc` depend on. Thus, *Fan-out* = 1 and *I* = 1/4.

Not All Components Should Be Stable

THE STABLE ABSTRACTIONS PRINCIPLE

A component should be as abstract as it is stable.

The Stable Abstractions Principle (SAP) sets up a relationship between stability and abstractness.

The abstractness of a package should be in proportion to its stability. the more stable a component is, the more abstract it should be.

if a component is to be stable, it should consist of interfaces and abstract classes so that it can be extended.

Measuring Abstraction Is simply the ratio of interfaces and abstract classes in a component to the total number of classes in the component.

Nc: The number of classes in the component.
Na: The number of abstract classes and interfaces in the component.
A: Abstractness. $A = Na / Nc$.
The *A* metric ranges from 0 to 1. A value of 0 implies that the component has no abstract classes at all. A value of 1 implies that the component contains nothing but abstract classes.