ECE444: Software Engineering

Design Patterns 2 (SOLID)

Shurui Zhou

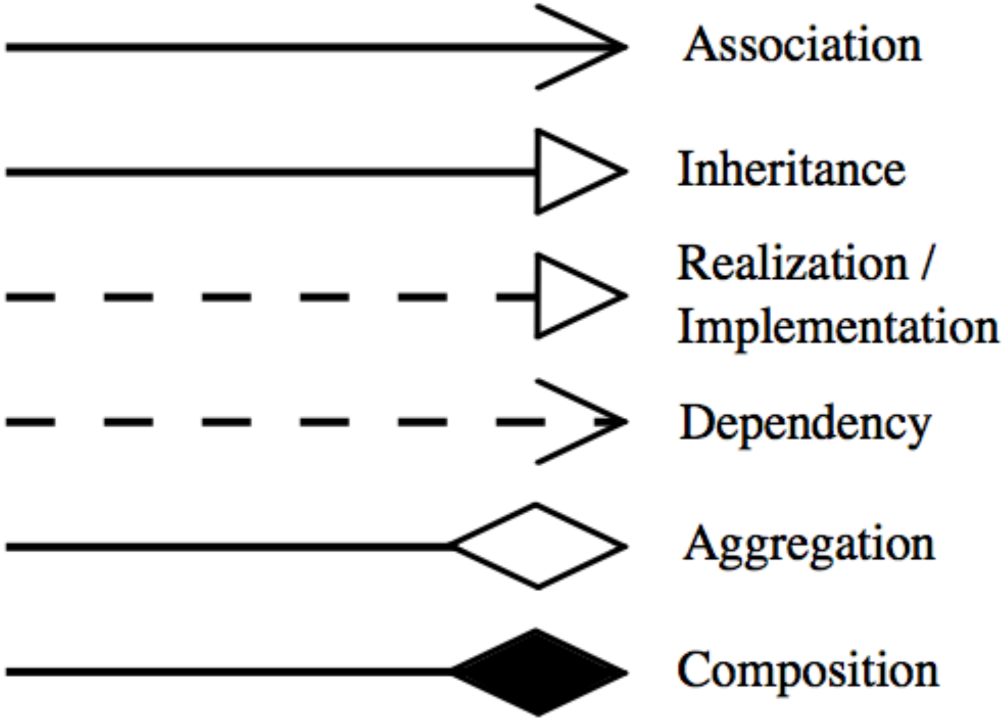
The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

- About posting questions on Piazza
- About Milestone 3 deadline
  - Group&Individual report (Monday) 10/5 11:59pm EST
  - Peer review (Thursday) 10/8 11:59pm EST
- About Milestone 5 deadline 11/18 11:59 EST
- About workload – incoming survey

Learning goals

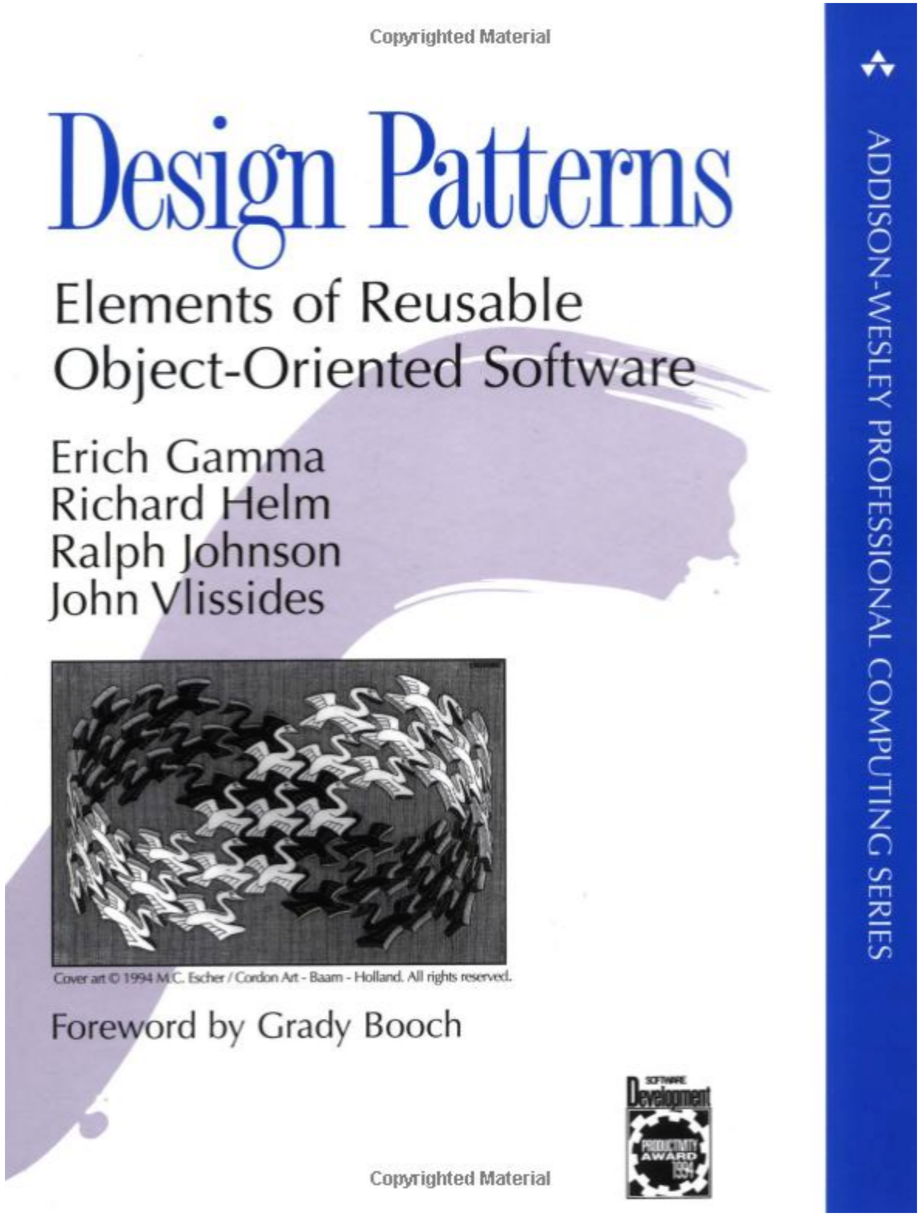
- Understand SOLID principle

UML Relationships



OO Design Principles

CARL QUIT. HE'S THE ONLY ONE WHO KNOWS HOW TO PROGRAM THE LEGACY SYSTEM.
scottadams@aol.com
www.dilbert.com
IT CAN'T BE THAT HARD. GO FIGURE IT OUT.
© 2006 Scott Adams, Inc./Dist. by UFS, Inc.
12-8-06
?
FRACK.



Copyrighted Material

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

Copyrighted Material

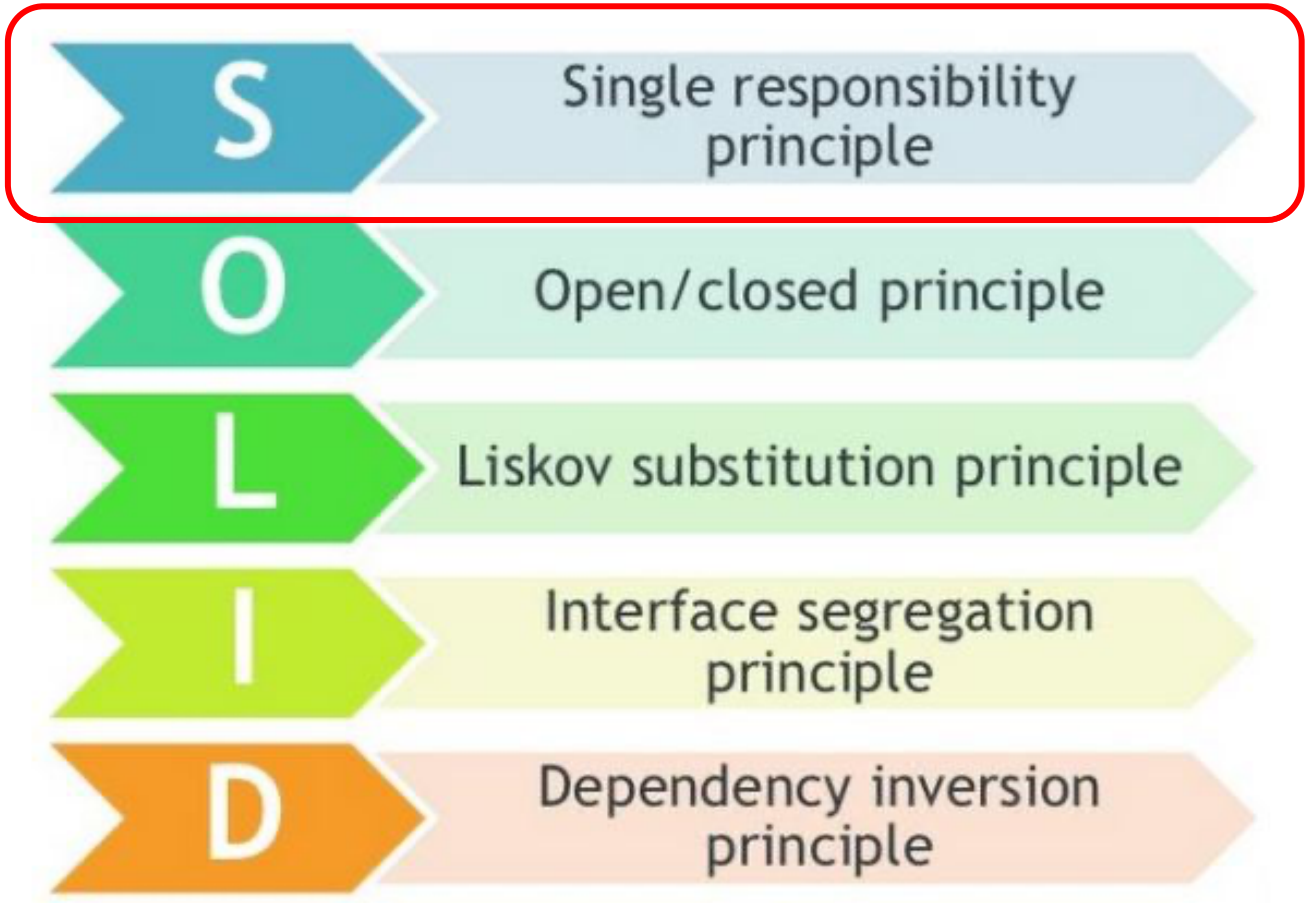
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Elements of Reusable Object-Oriented Software
- 23 OO patterns

Why Patterns?

- They offer solutions for specific problems
- They are easily applicable because the purpose and application are consistently described
- They make work more efficient
- They can be adapted to specific contexts
- They make communication between developers easier
- Goal: Understandable, reusable, testable, maintainable and flexible

OO Design Principles



S Single responsibility principle

O Open/closed principle

L Liskov substitution principle

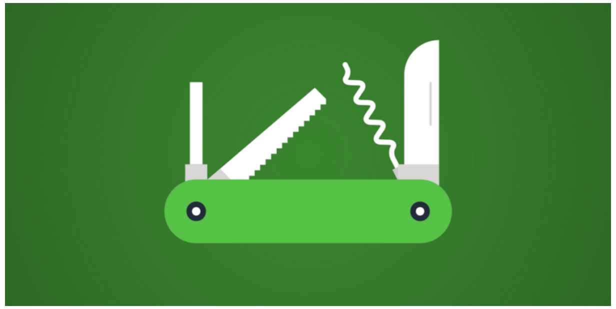
I Interface segregation principle

D Dependency inversion principle

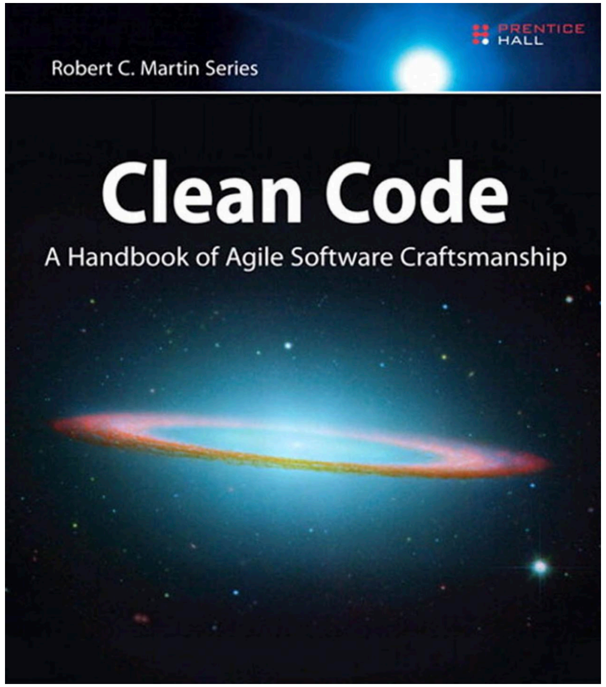
Single Responsibility Principle

Guildelines to partition your logic into classes

Single Responsibility Principle



A class should have one, and only one, reason to change.
Just because you can, doesn't mean you should

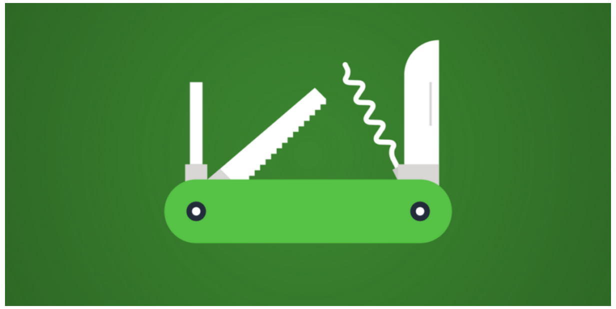


Benefits:

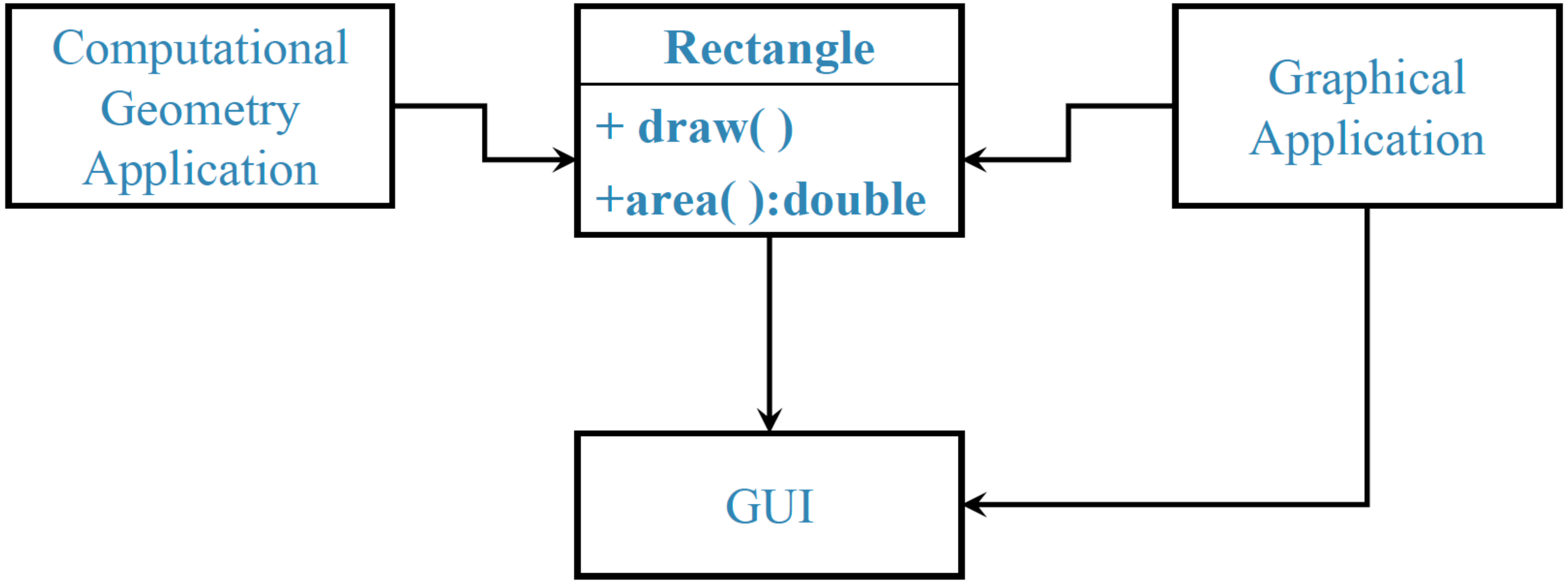
- Frequency and Effects of Changes
- Easier to Understand

Q: What is the responsibility of your class/component/microservice?

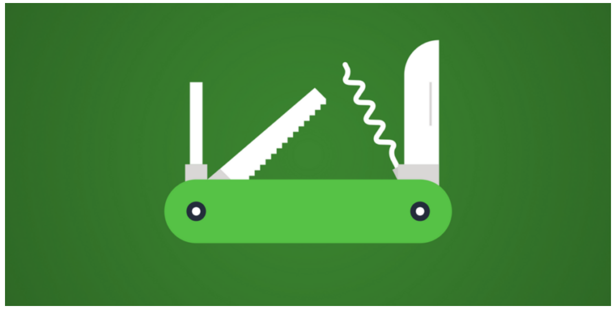
Single Responsibility Principle

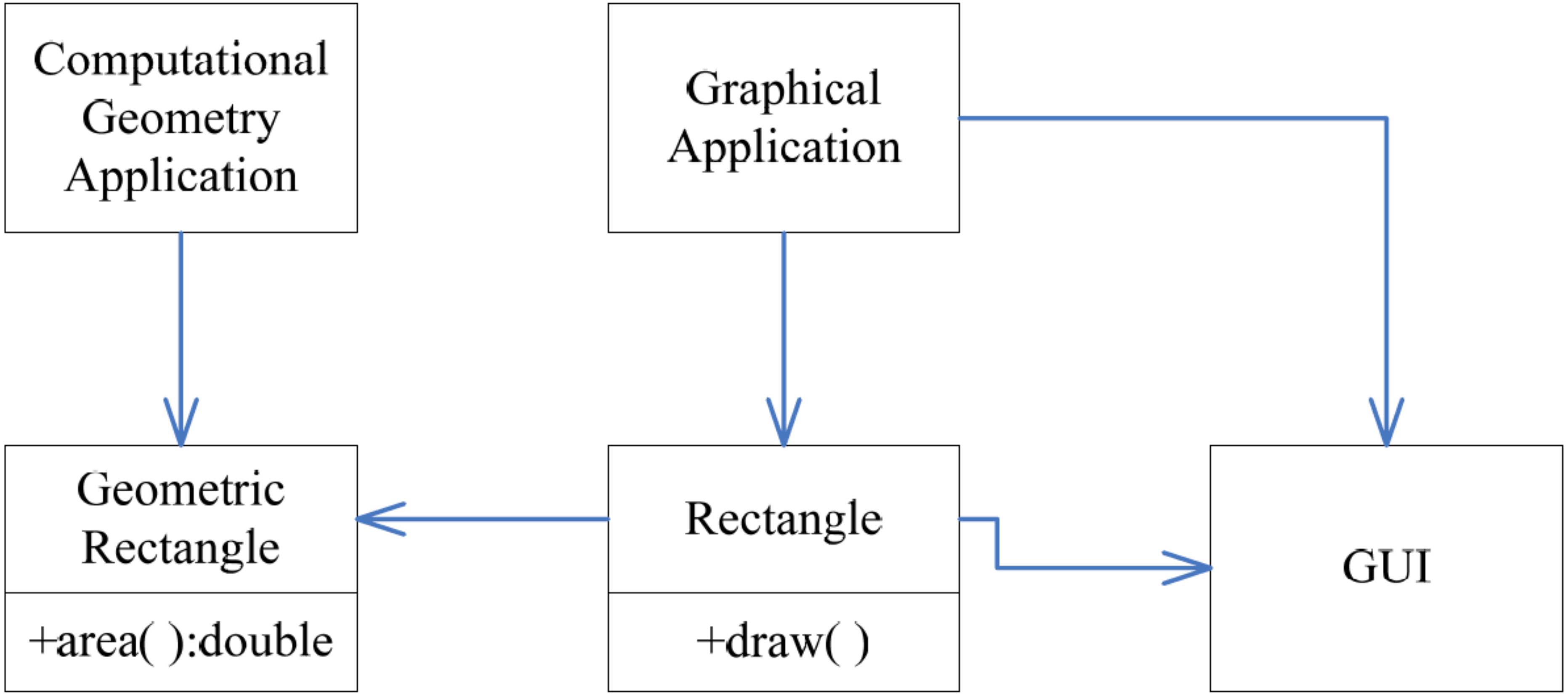


A class should have one, and only one, reason to change.

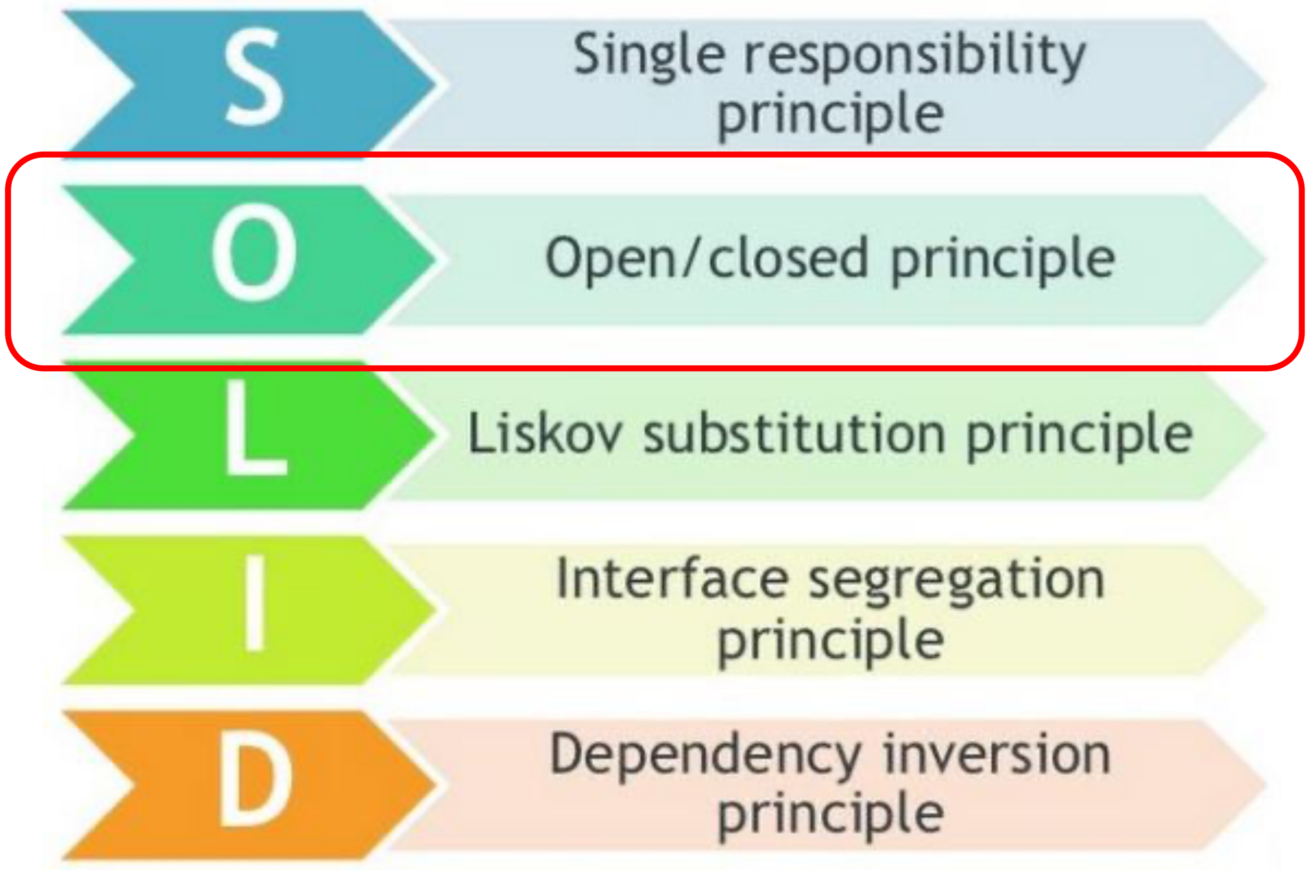


Single Responsibility Principle





OO Design Principles



S Single responsibility principle

O Open/closed principle

L Liskov substitution principle

I Interface segregation principle

D Dependency inversion principle

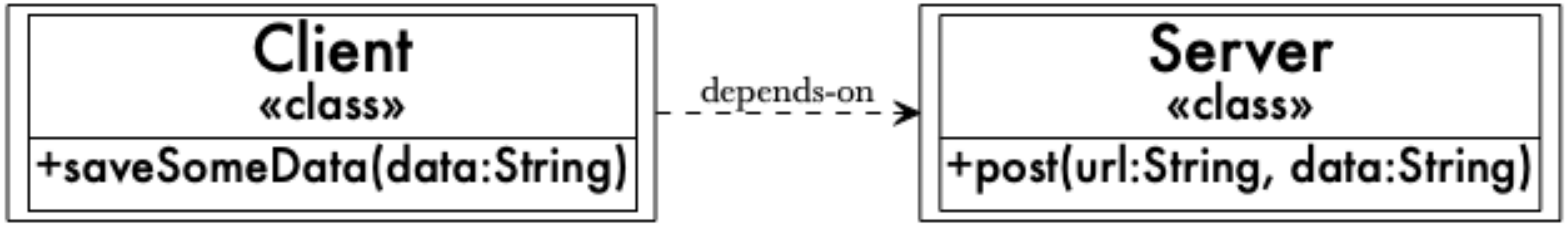
Open-Closed Principle (OCP)

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

Open-Closed Principle

- Implementation:
  - inheritance
  - composition
- Benefits:
  - extend a component's logic without breaking backward compatibility
  - test different component implementations (that have the same logic) against each other.

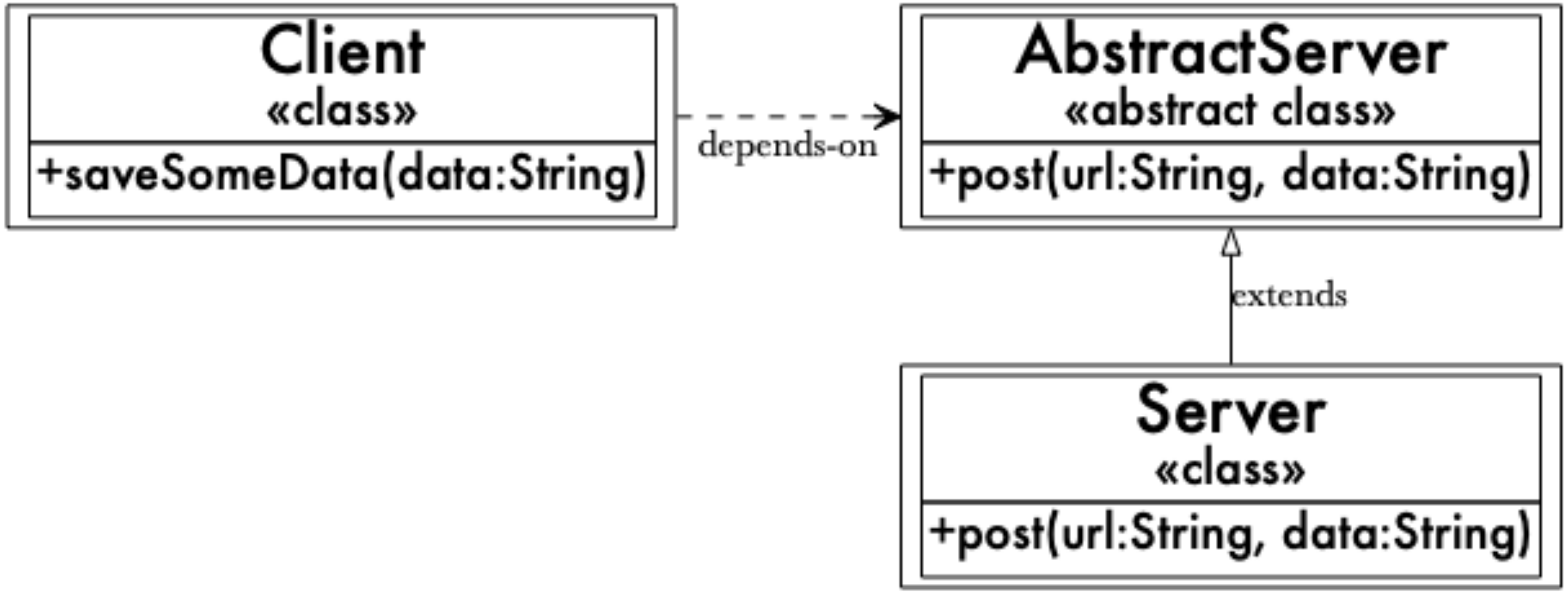
Open-Closed Principle (Example: Client&Server)



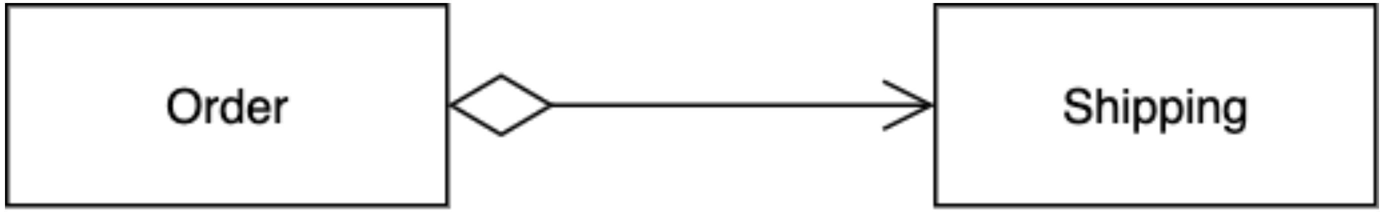
The class is:

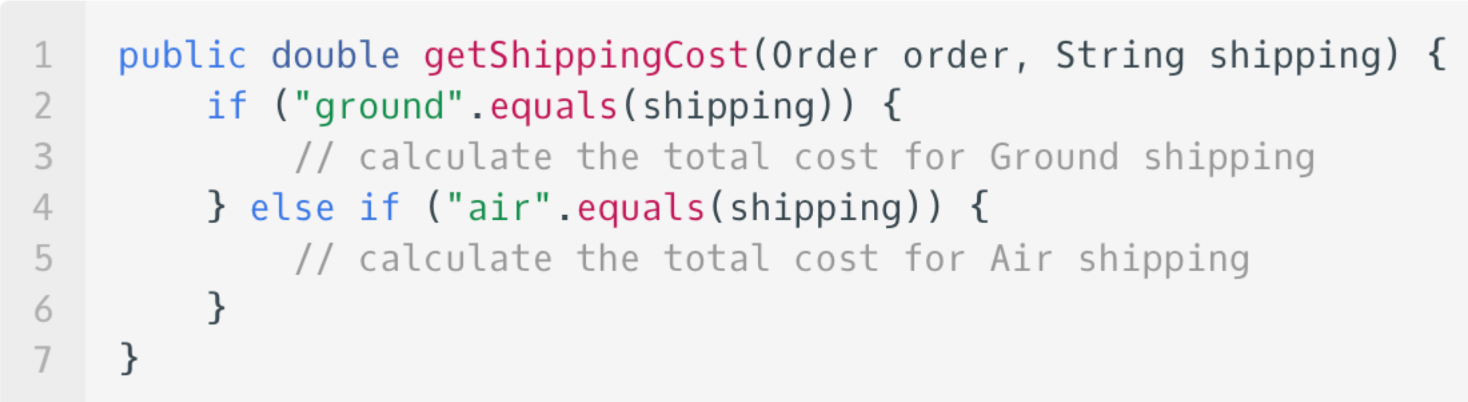
- not open for extension, since we always use a concrete Server instance
- not closed for modification, because if we wish to change to another type of server, we must change the source code.

Open-Closed Principle (Example: Client&Server)


Client
«class»
+saveSomeData(data:String)
depends-on
AbstractServer
«abstract class»
+post(url:String, data:String)
extends
Server
«class»
+post(url:String, data:String)

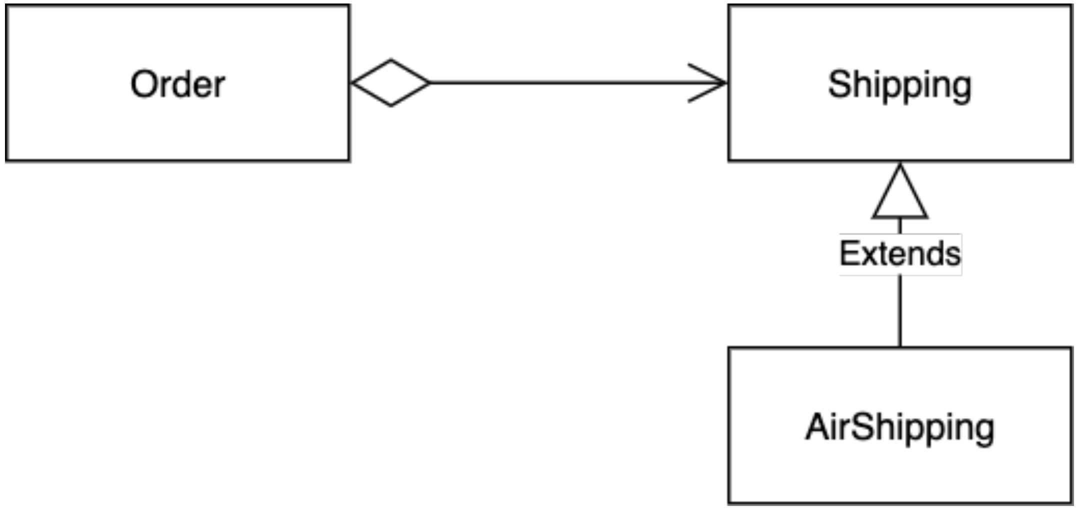
Open-Closed Principle (Example: Order&Shipping)

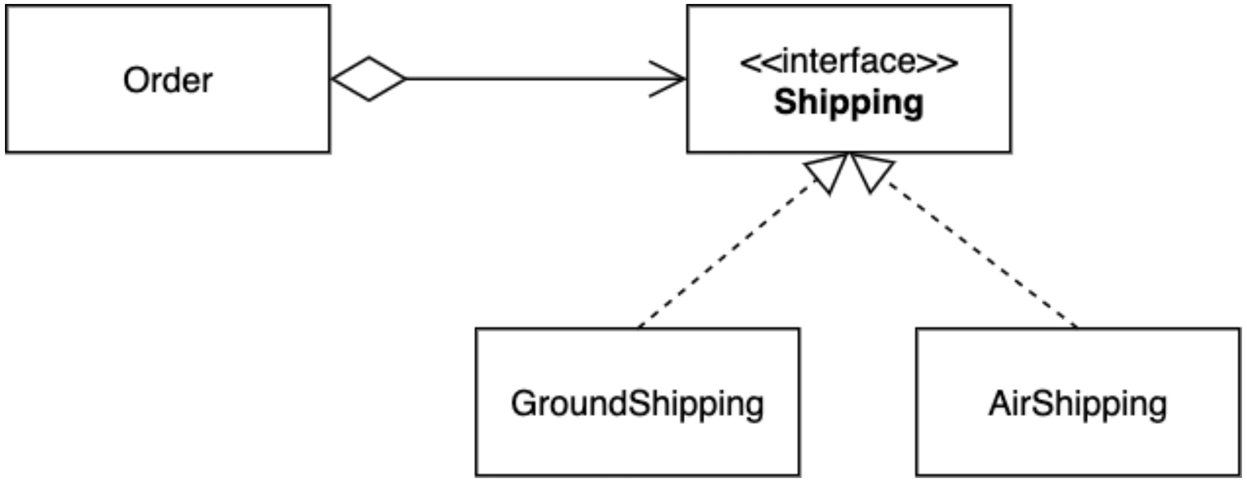




```java
public double getShippingCost(Order order, String shipping) {
    if ("ground".equals(shipping)) {
        // calculate the total cost for Ground shipping
    } else if ("air".equals(shipping)) {
        // calculate the total cost for Air shipping
    }
}
```

Open-Closed Principle (Example: Order&Shipping)

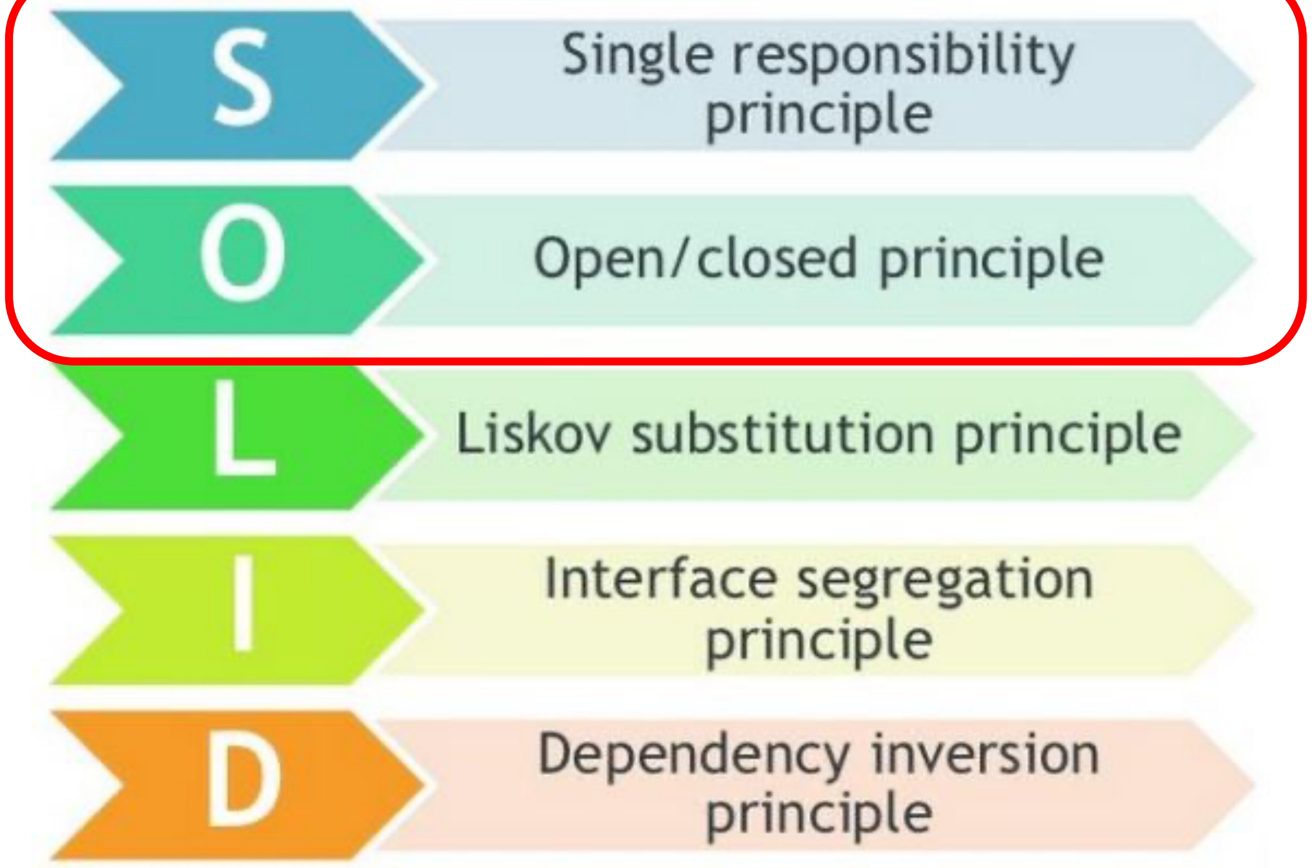




Thoughts? Critiques on OCP

- Adding un-needed flexibility to code (to make it open for extension) breeds complexity and carrying cost.
- It requires imagining all sorts of use-cases that don’t exist in order to make it ultimately flexible.
- Principle != you should always do this

OO Design Principles



S Single responsibility principle

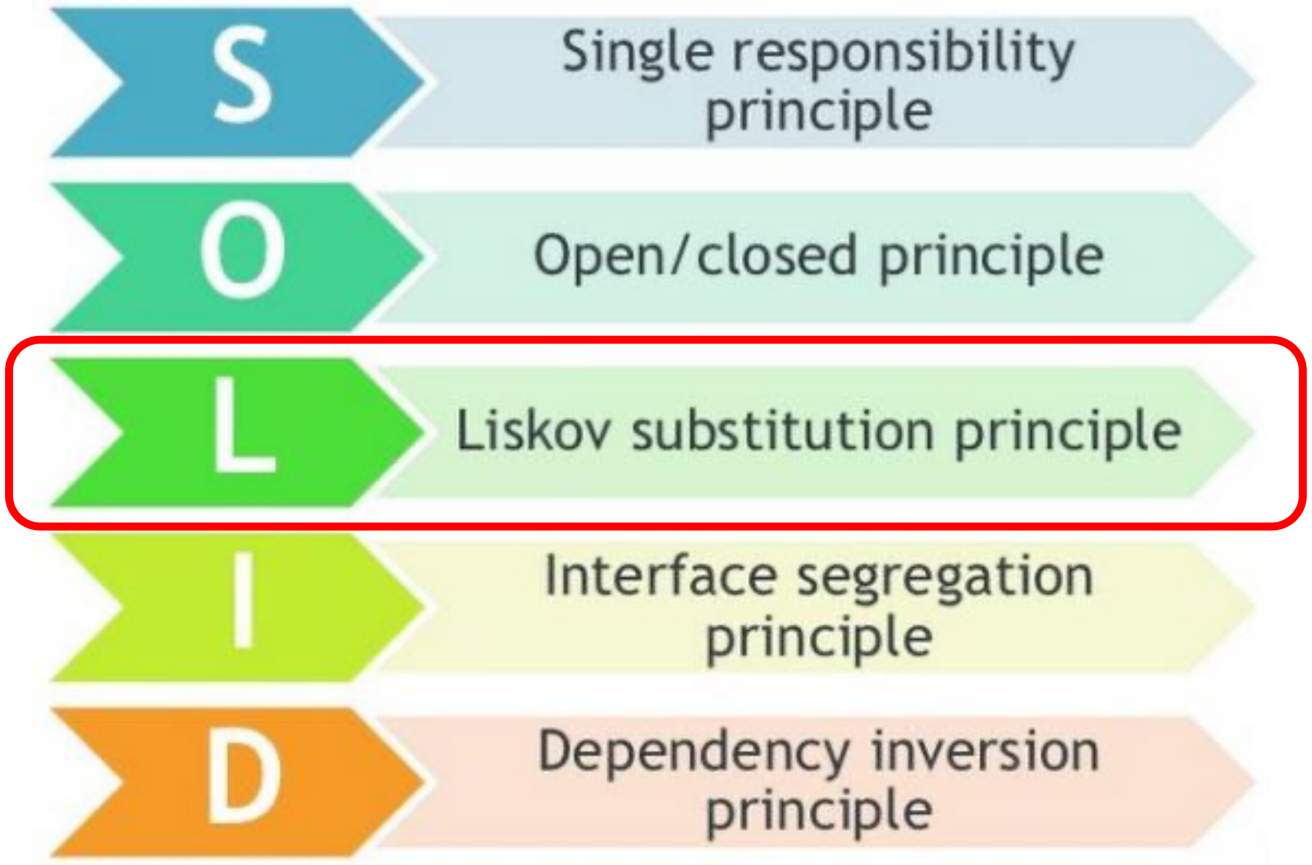
O Open/closed principle

L Liskov substitution principle

I Interface segregation principle

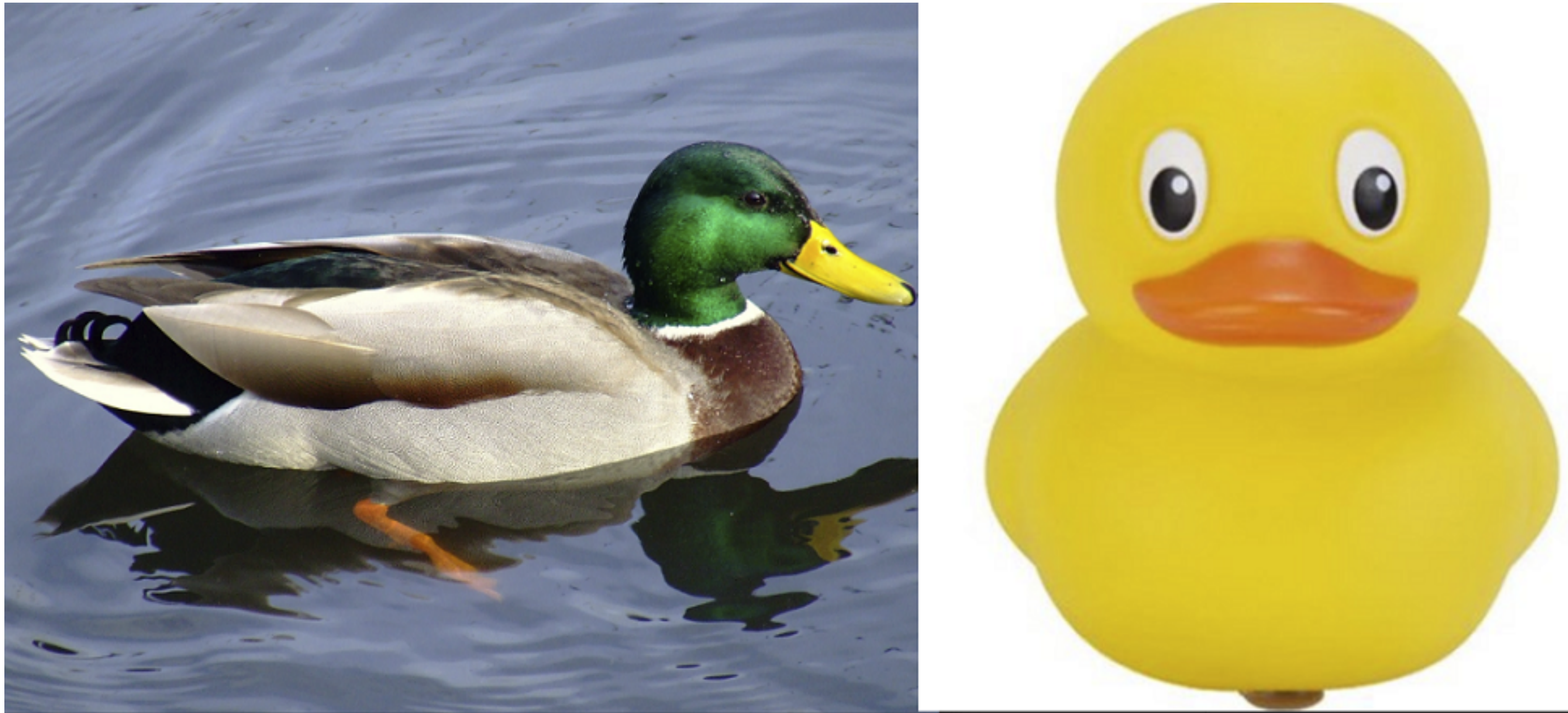
D Dependency inversion principle

OO Design Principles



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

Duck Tesk



If it looks like a duck and quacks like a duck but it needs batteries,
you probably have the wrong abstraction.

Liskov Substitution Principle (LSP)

- The object of a derived class should be able to replace an object of the base class without bringing any errors in the system or modifying the behavior of the base class.

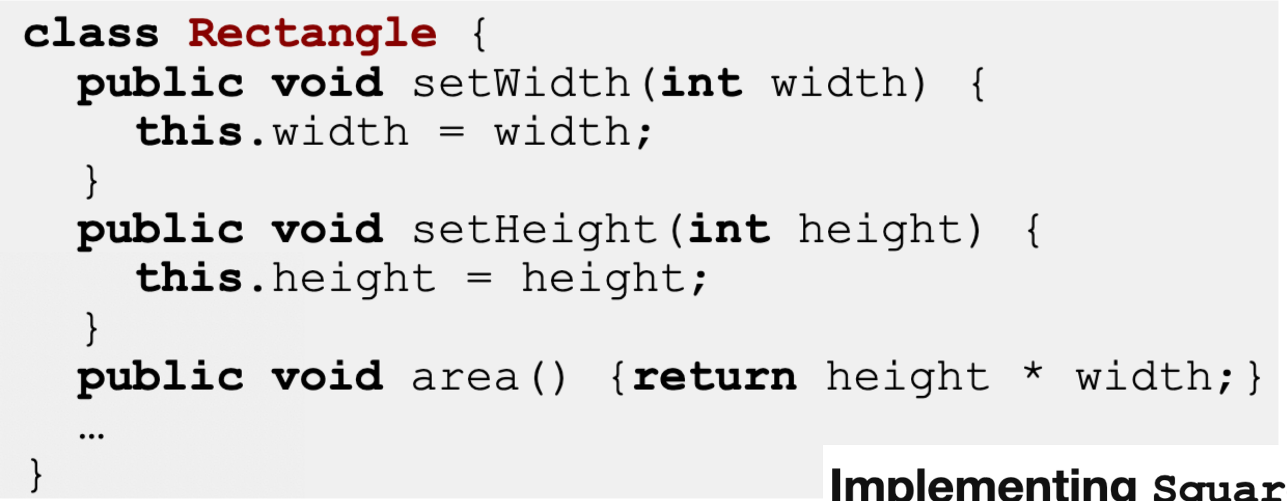
Benefit of LSP

- Code that adheres to LSP is loosely dependent to each other and encourages code reusability.

Disadvantages to violating the LSP

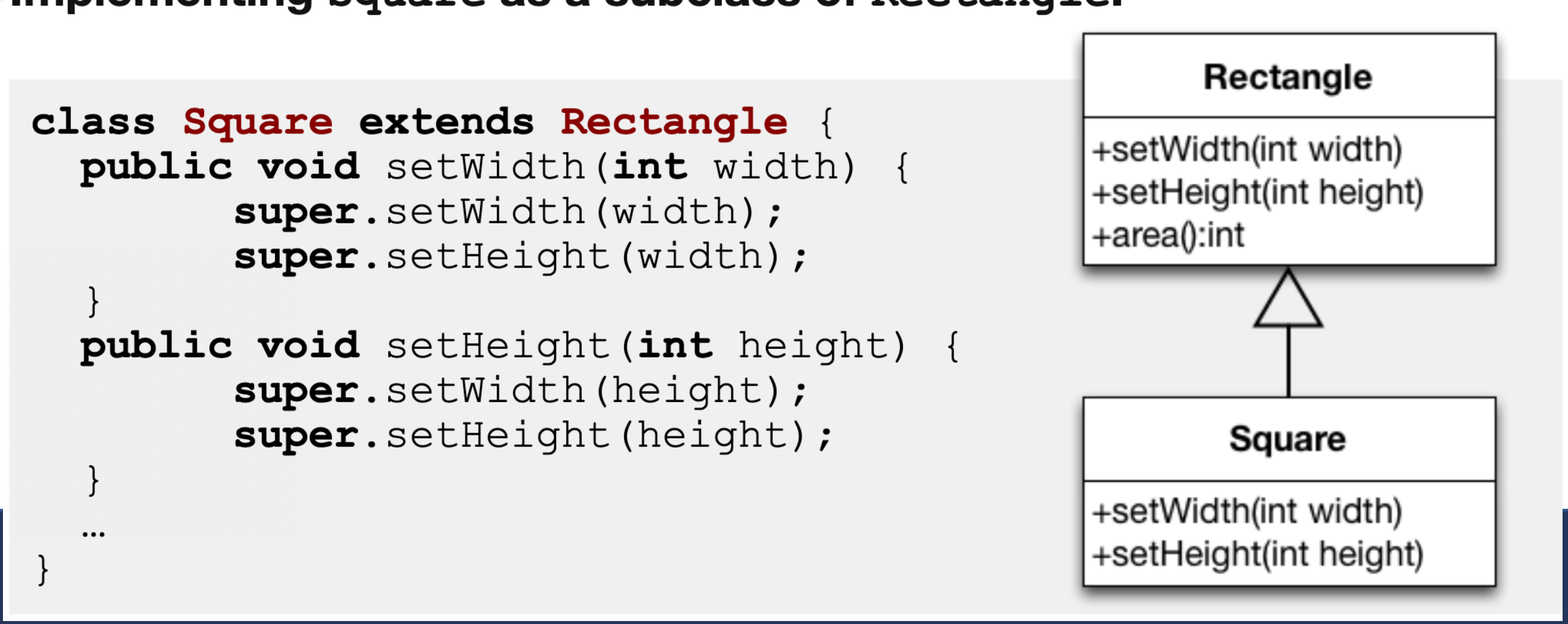
- Code that does not adhere to the LSP is tightly coupled and creates unnecessary entanglements.
- E.g. when a subclass can not substitue its parent class there would have to be multiple conditional statements to determine the class or type to handle certain cases differently.

Violating the Liskov Substitution Principle



```
class Rectangle {
  public void setWidth(int width) {
    this.width = width;
  }
  public void setHeight(int height) {
    this.height = height;
  }
  public void area() {return height * width;}
  …
}
```

IS-A relationship

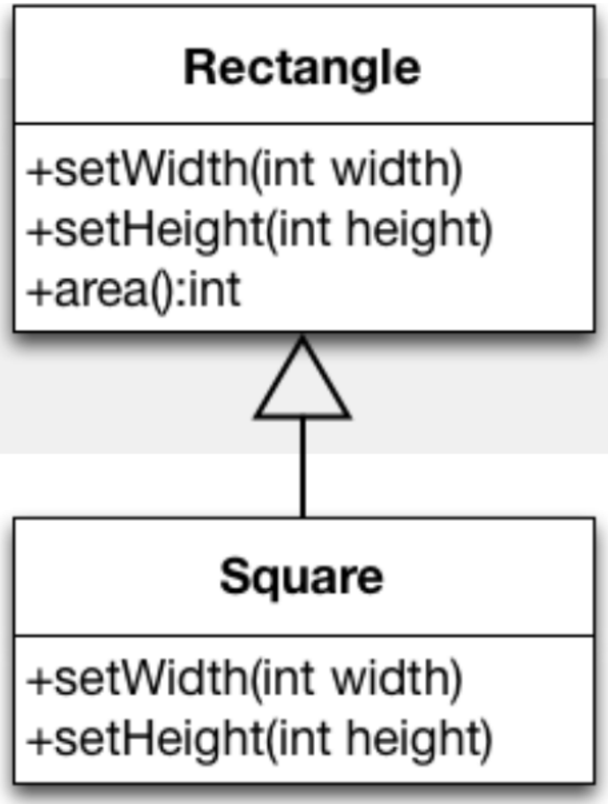


Implementing `Square` as a subclass of `Rectangle`:

```
class Square extends Rectangle {
  public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
  }
  public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
  }
  …
}
```

Violating the Liskov Substitution Principle

```
void clientMethod(Rectangle rec) {
  rec.setWidth(5);
  rec.setHeight(4);
  assert(rec.area() == 20);
}
```



Rectangle

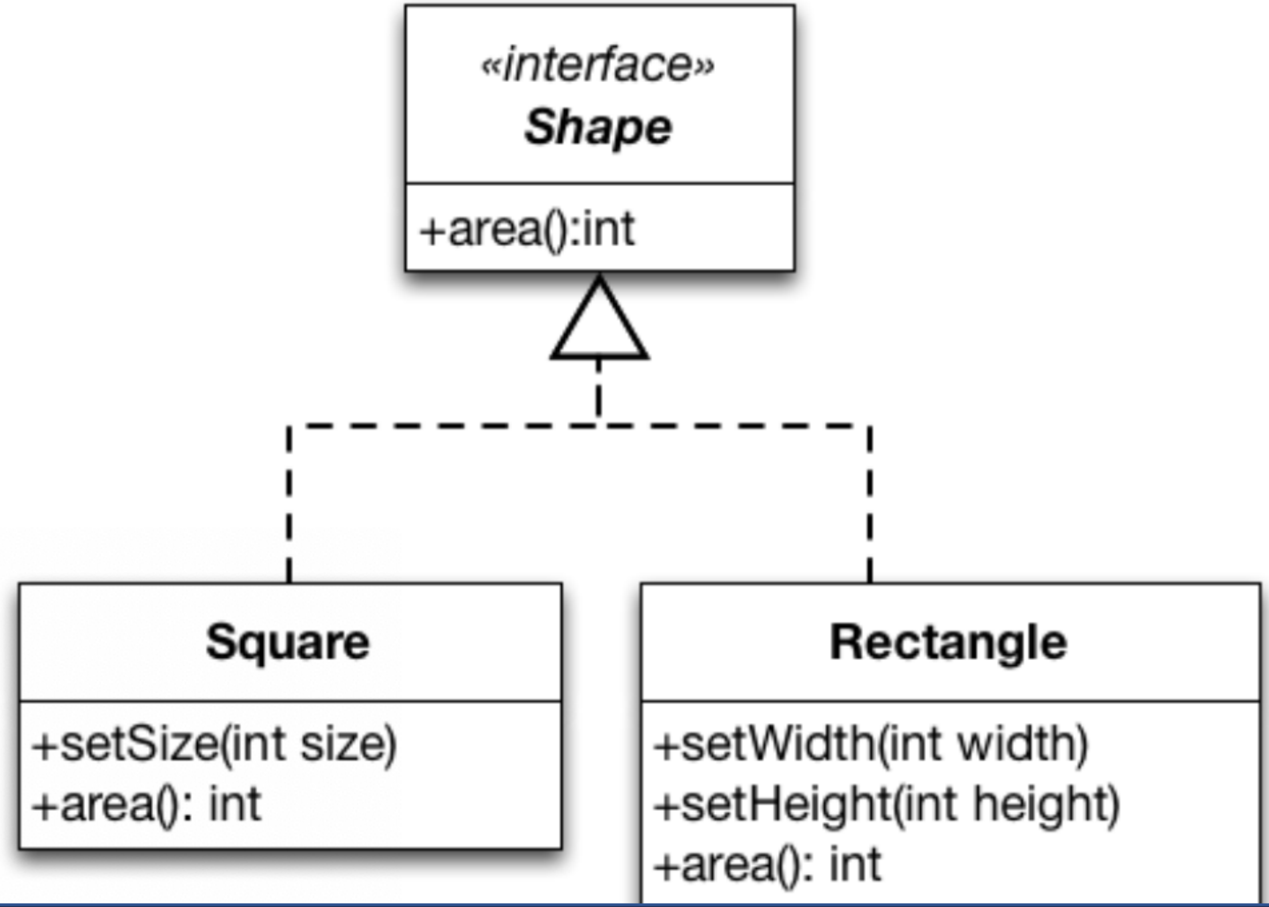
+setWidth(int width)
+setHeight(int height)
+area():int

Square

+setWidth(int width)
+setHeight(int height)

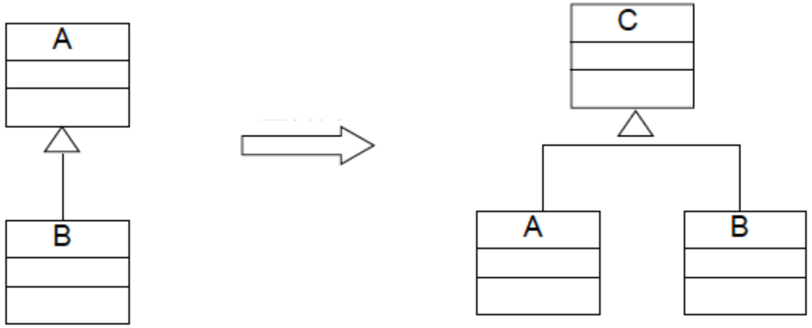
Liskov Substitution Principle (LSP)

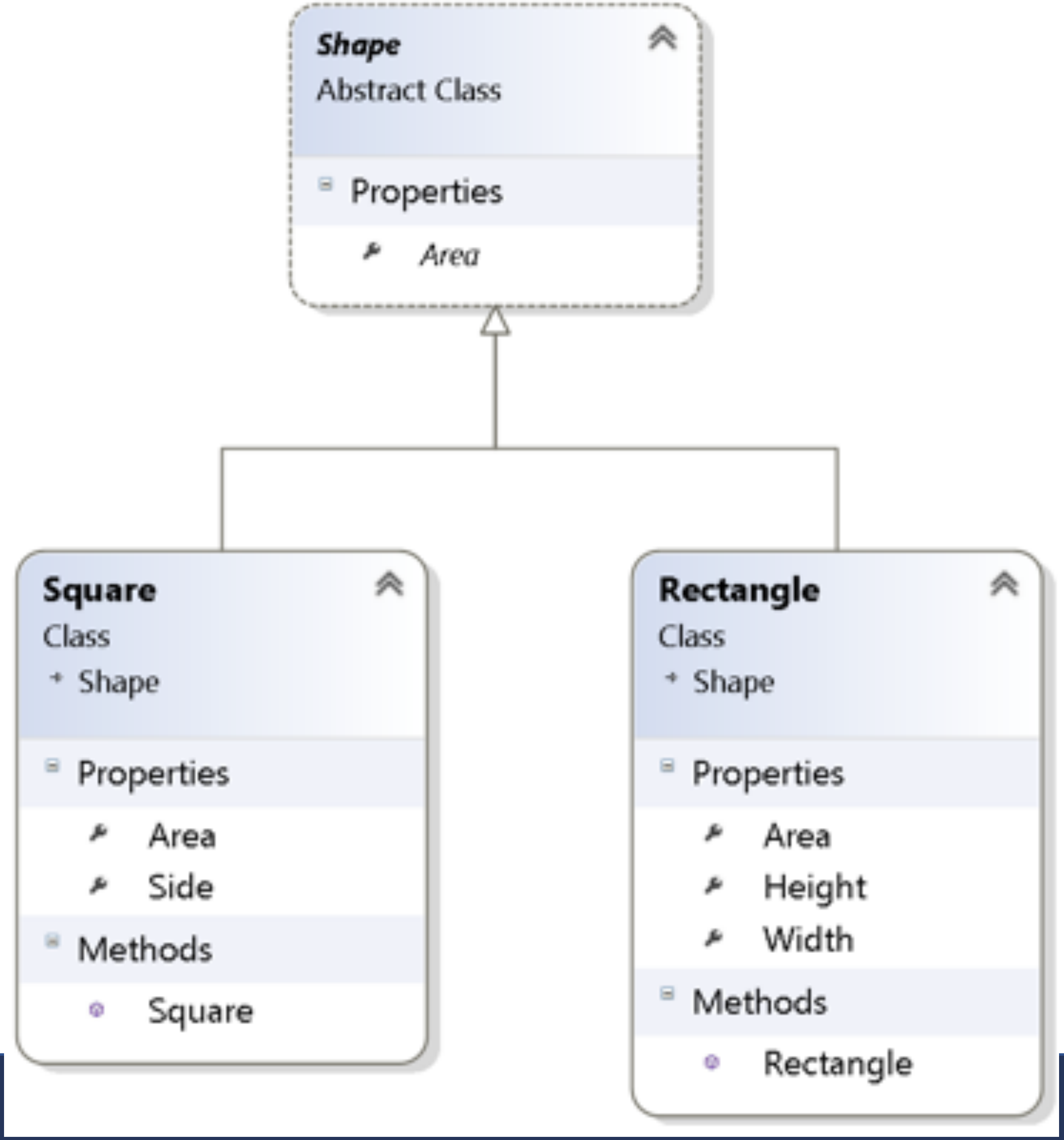
- A LSP compliant solution
- Introduce the interface Shape



Solution

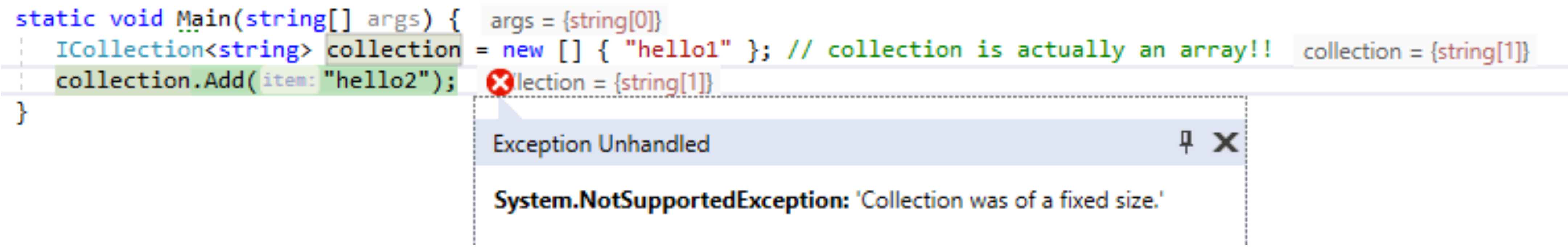
- To encapsulate what varies and to provide a generic interface we introduce an abstract Shape class.





Violating the Liskov Substitution Principle

- .NET System.Array implementing the ICollection<T> interface
- The C# compiler doesn't even warn on such simple erroneous program.



```
static void Main(string[] args) {   args = {string[0]}
    ICollection<string> collection = new [] { "hello1" }; // collection is actually an array!!   collection = {string[1]}
    collection.Add(item: "hello2");   lection = {string[1]}
}
```

Exception Unhandled

System.NotSupportedException: 'Collection was of a fixed size.'

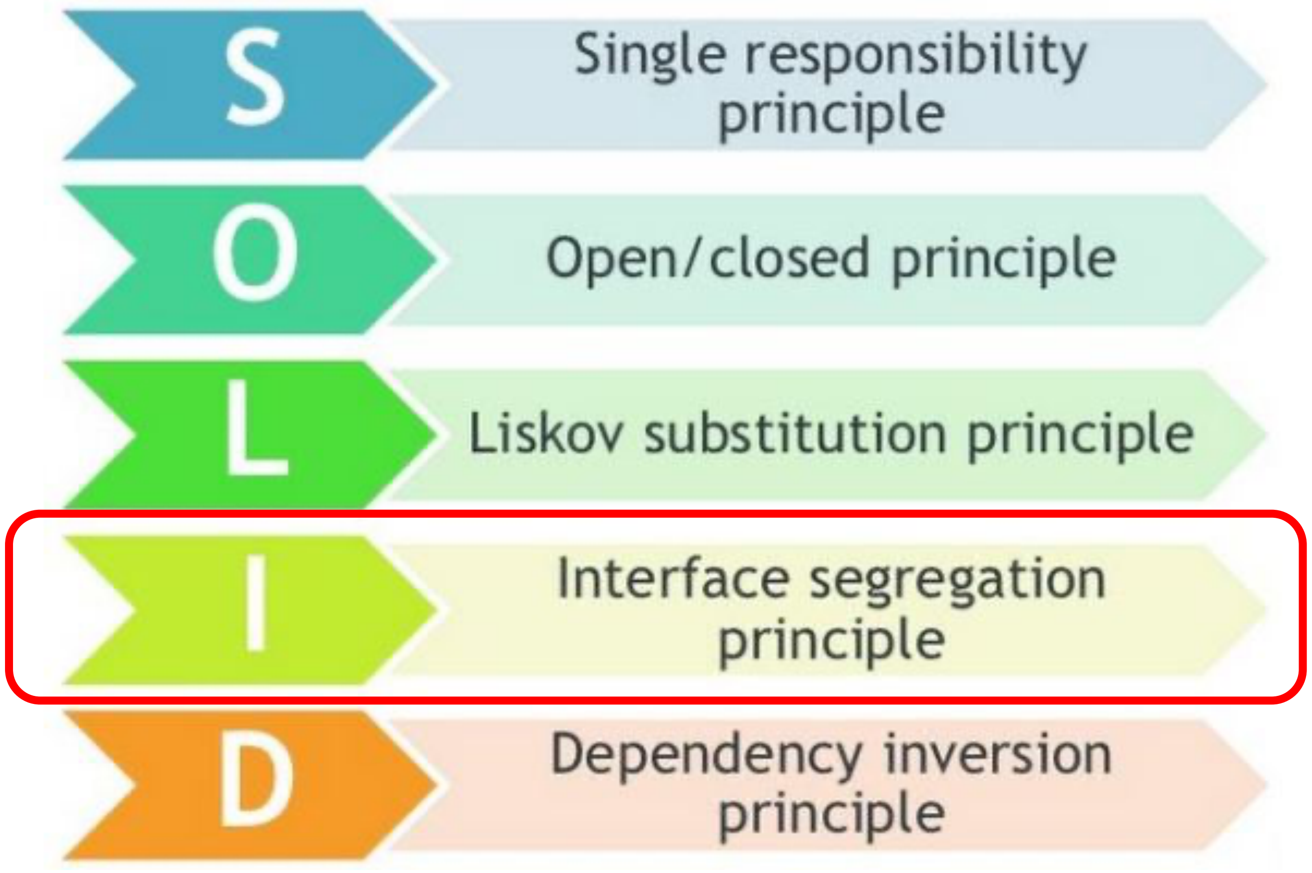
Liskov Substitution Principle (LSP)

- Think twice before applying the IS-A trick
- Use polymorphism with great caution
- Do this member applies seamlessly to all objects that will implement this interface?
- When writing an API first take the point of view of the client of your API
- Test-Driven Development (TDD), where client code must be written for test and design purposes before writing the code itself.

Corresponding Design Patterns

- Strategy
- Composite
- Proxy

OO Design Principles



S Single responsibility principle

O Open/closed principle

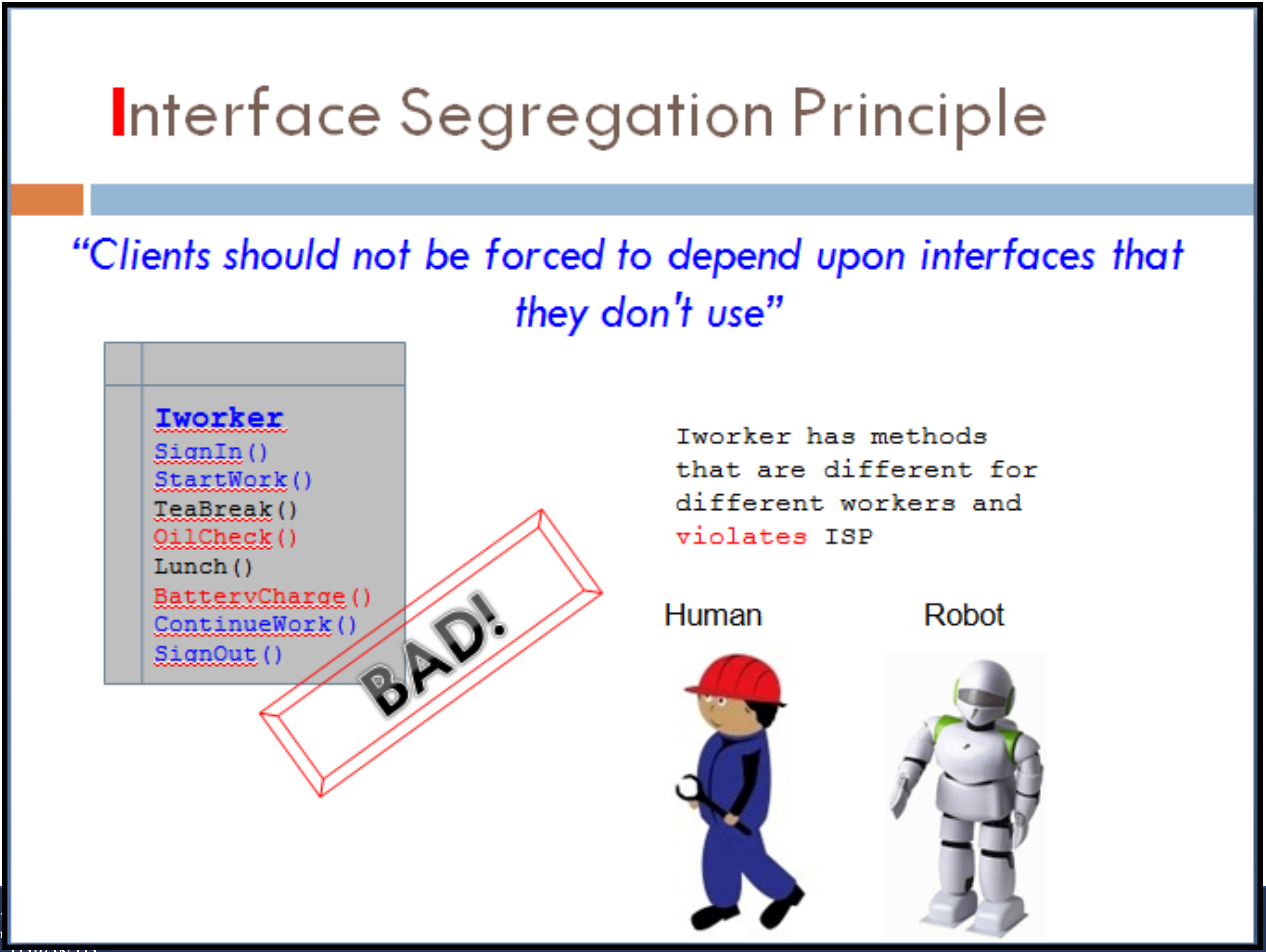
L Liskov substitution principle

I Interface segregation principle

D Dependency inversion principle

Interface Segregation Principle

When more means less

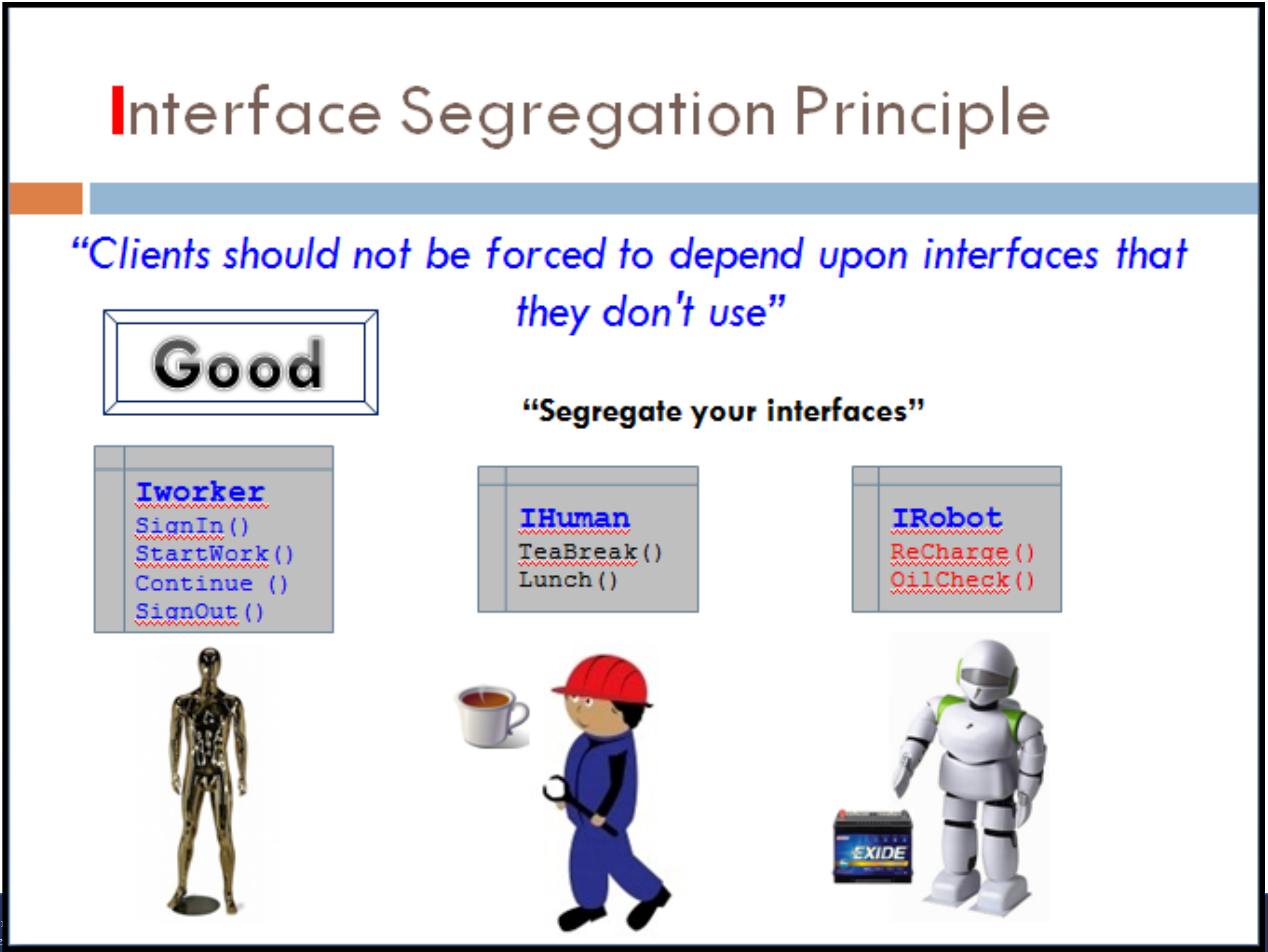


Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they don't use"

```
Iworker
SignIn()
StartWork()
TeaBreak()
OilCheck()
Lunch()
BatteryCharge()
ContinueWork()
SignOut()
```

Iworker has methods that are different for different workers and violates ISP



Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they don't use"

Good

"Segregate your interfaces"

```
Iworker
SignIn()
StartWork()
Continue ()
SignOut()
```

```
IHuman
TeaBreak()
Lunch()
```

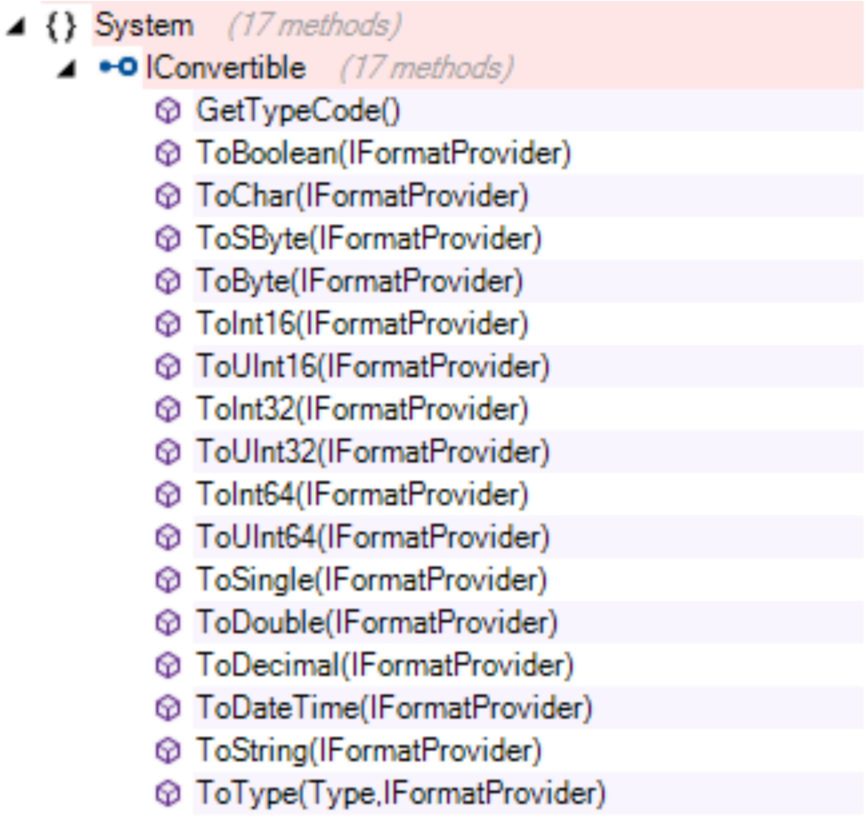
```
IRobot
ReCharge()
OilCheck()
```

Interface Segregation Principle (ISP)

- No client should be forced to depend on methods it does not use.
- The goal of ISP is similar to Single Responsibility principle : to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.

Interface Segregation Principle (ISP)

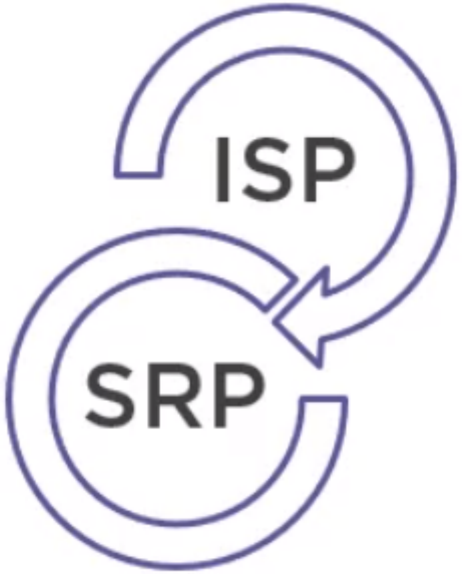
- A fat interface is not necessarily a design flaw



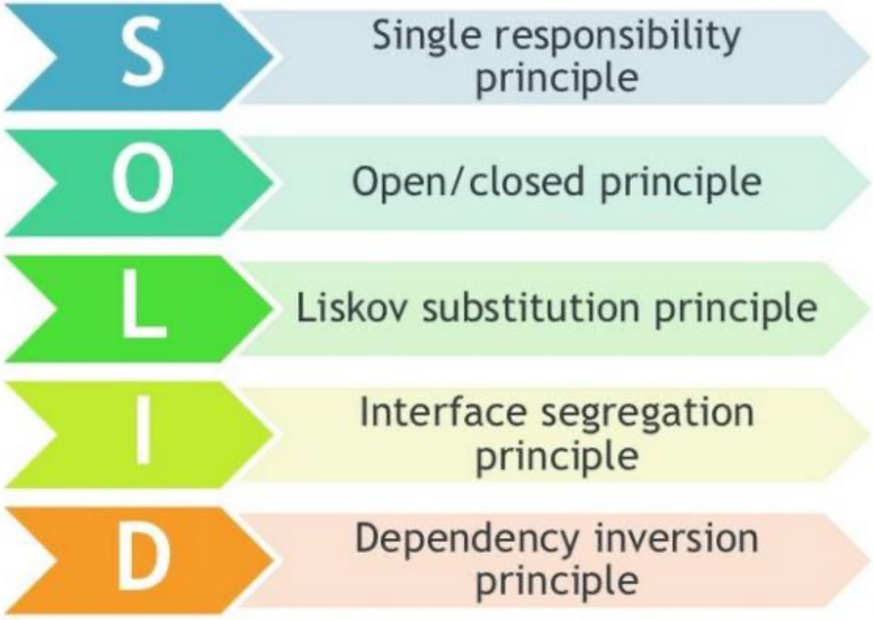
```
{} System  (17 methods)
  IConvertible  (17 methods)
    GetTypeCode()
    ToBoolean(IFormatProvider)
    ToChar(IFormatProvider)
    ToSByte(IFormatProvider)
    ToByte(IFormatProvider)
    ToInt16(IFormatProvider)
    ToUInt16(IFormatProvider)
    ToInt32(IFormatProvider)
    ToUInt32(IFormatProvider)
    ToInt64(IFormatProvider)
    ToUInt64(IFormatProvider)
    ToSingle(IFormatProvider)
    ToDouble(IFormatProvider)
    ToDecimal(IFormatProvider)
    ToDateTime(IFormatProvider)
    ToString(IFormatProvider)
    ToType(Type,IFormatProvider)
```

```
[SuppressMessage("NDepend", "ND1200:AvoidInterfacesTooBig",
Justification="This interface is fat because it needs to support all
primitive types"]
public interface IConvertible {
  ...
```

https://www.ndepend.com/docs/suppress-issues?_ga=2.63469095.983202201.1601605450-1723910178.1601605450

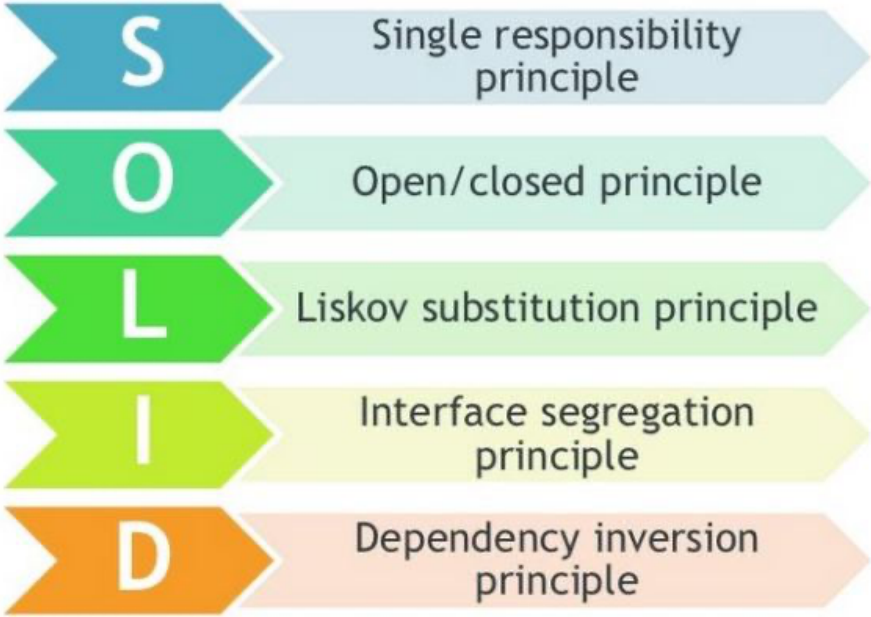


Classes that implement small interfaces are more focused and tend to have a single purpose



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

By keeping interfaces small, the classes that implement them have a higher chance to fully substitute the interface



S Single responsibility principle

O Open/closed principle

L Liskov substitution principle

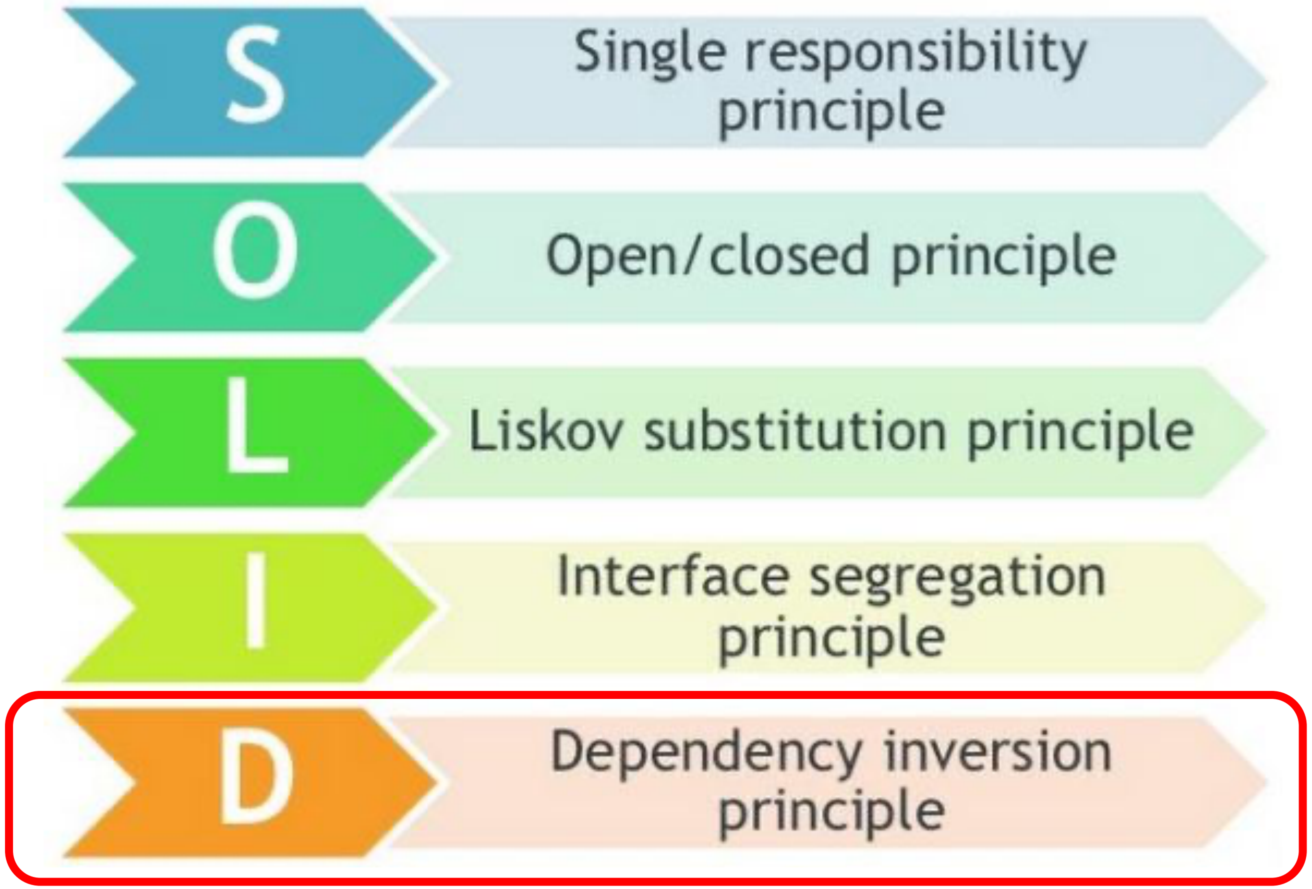
I Interface segregation principle

D Dependency inversion principle

Corresponding Design Patterns

- Memento
- Iterator

OO Design Principles



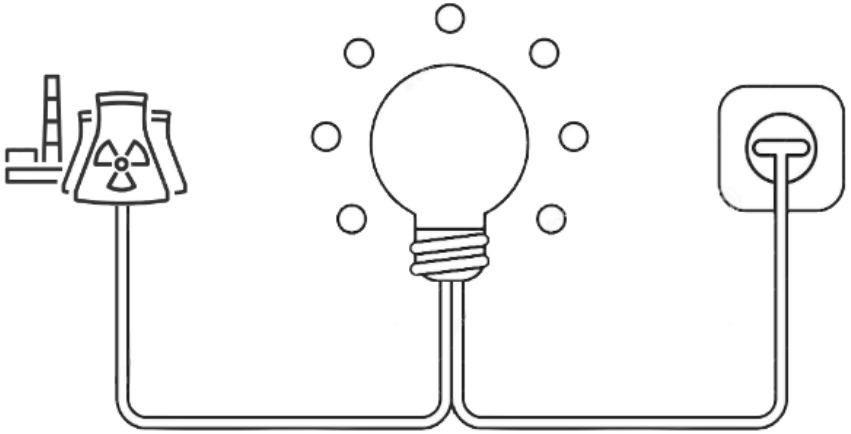
S — Single responsibility principle

O — Open/closed principle

L — Liskov substitution principle

I — Interface segregation principle

D — Dependency inversion principle



Dependency Inversion Principle

When knowing how things work becomes a burden

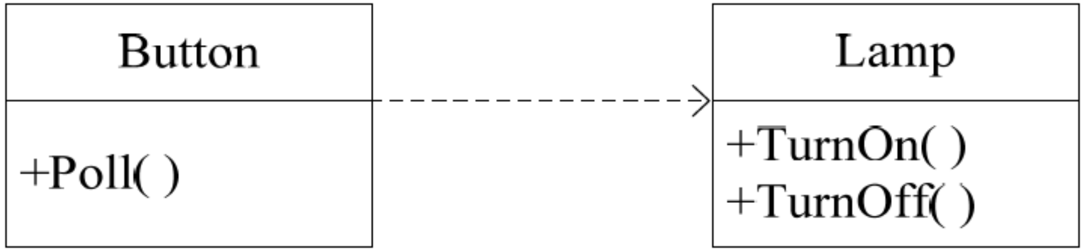
Dependency Inversion Principle (DIP)

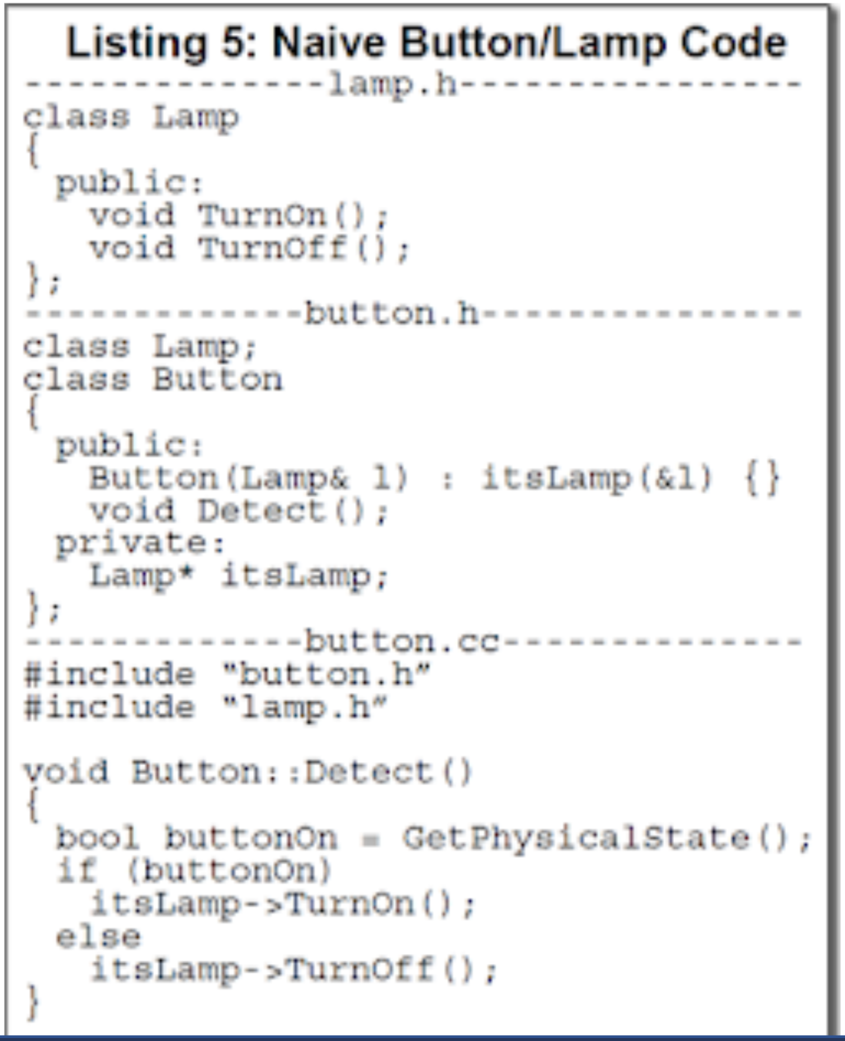
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details *(concrete implementation)*. Details should depend on abstractions.

Dependency Inversion Principle (DIP)

- A **High level module** is any module that contains the policy decisions and business model of an application. This can be regarded as the app identity. The higher level modules are primarily consumed by the presentation layer within an app.
- **Low level modules** are modules that contains detailed implementation that are required to execute the decisions and business policies.

Dependency Inversion Principle (DIP)



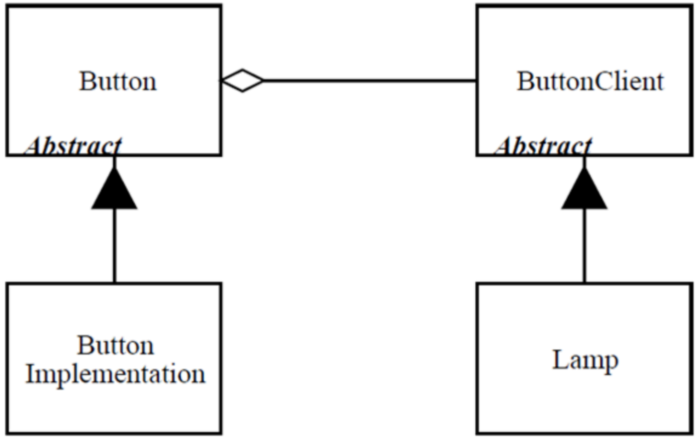


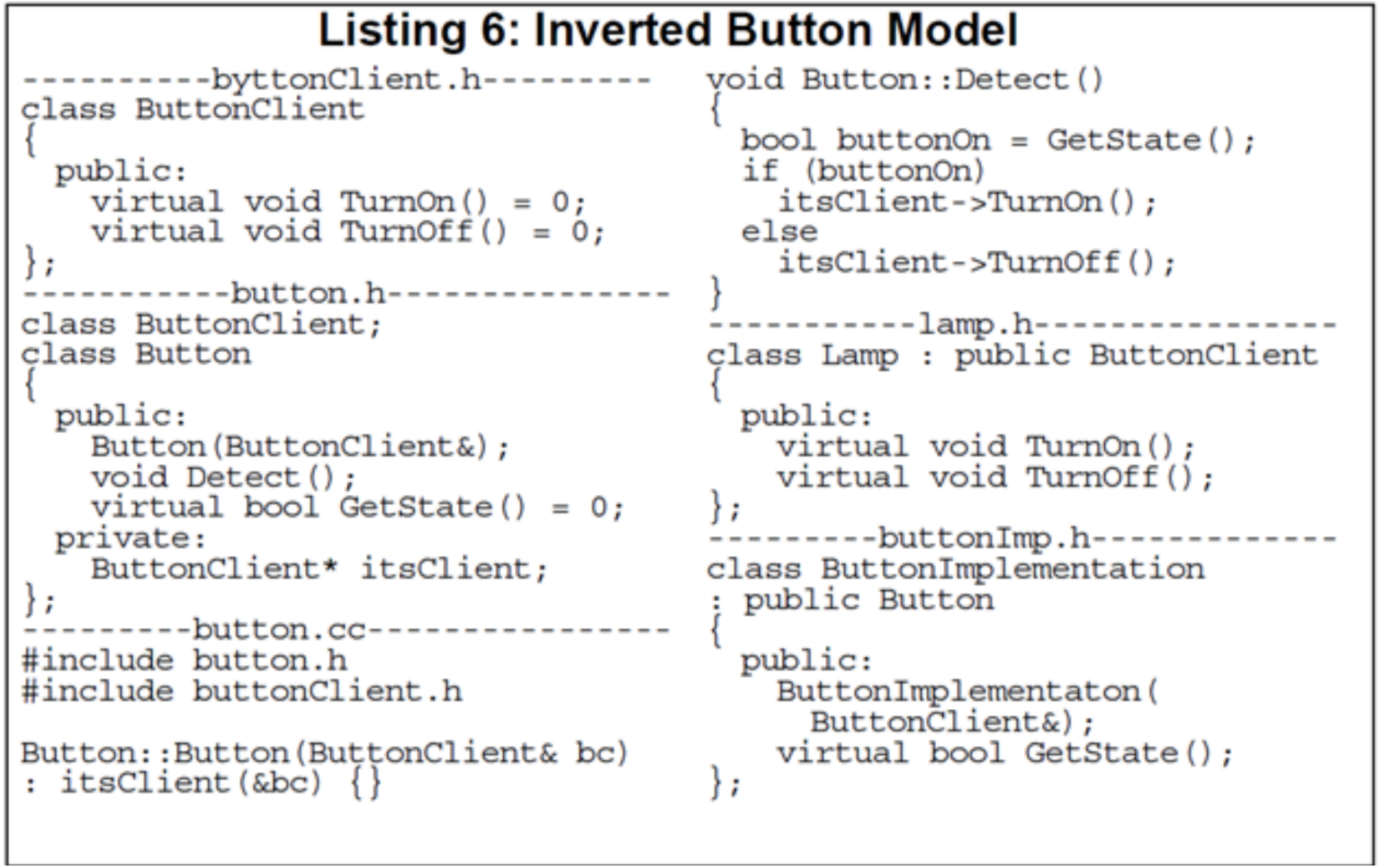
Listing 5: Naive Button/Lamp Code

```
--------------lamp.h----------------
class Lamp
{
  public:
    void TurnOn();
    void TurnOff();
};
-------------button.h---------------
class Lamp;
class Button
{
  public:
    Button(Lamp& l) : itsLamp(&l) {}
    void Detect();
  private:
    Lamp* itsLamp;
};
-------------button.cc--------------
#include "button.h"
#include "lamp.h"

void Button::Detect()
{
  bool buttonOn = GetPhysicalState();
  if (buttonOn)
    itsLamp->TurnOn();
  else
    itsLamp->TurnOff();
}
```

Dependency Inversion Principle (DIP)

Figure 6: Inverted Button Model





Listing 6: Inverted Button Model

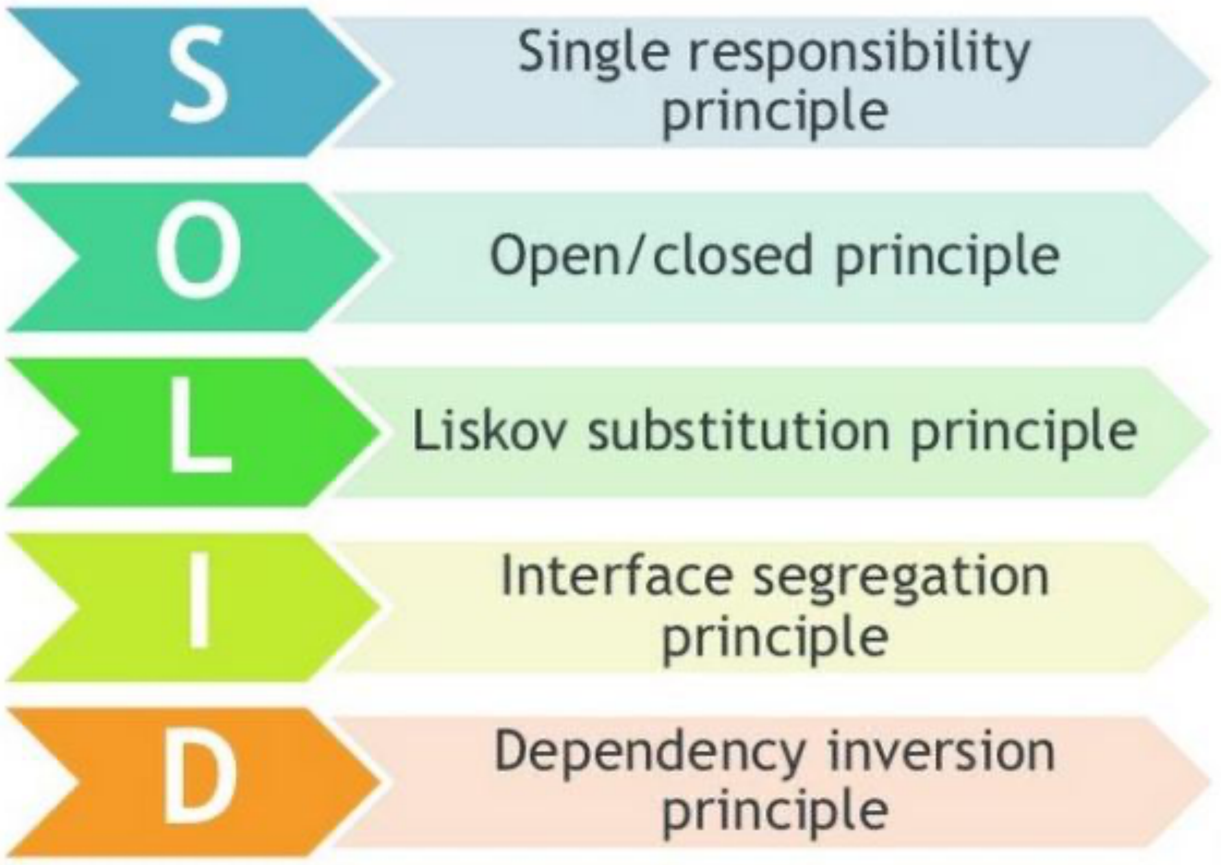
```
----------byttonClient.h---------
class ButtonClient
{
  public:
    virtual void TurnOn() = 0;
    virtual void TurnOff() = 0;
};
-----------button.h---------------
class ButtonClient;
class Button
{
  public:
    Button(ButtonClient&);
    void Detect();
    virtual bool GetState() = 0;
  private:
    ButtonClient* itsClient;
};
---------button.cc----------------
#include button.h
#include buttonClient.h

Button::Button(ButtonClient& bc)
: itsClient(&bc) {}
```

```
void Button::Detect()
{
  bool buttonOn = GetState();
  if (buttonOn)
    itsClient->TurnOn();
  else
    itsClient->TurnOff();
}
-----------lamp.h----------------
class Lamp : public ButtonClient
{
  public:
    virtual void TurnOn();
    virtual void TurnOff();
};
---------buttonImp.h-------------
class ButtonImplementation
: public Button
{
  public:
    ButtonImplementaton(
      ButtonClient&);
    virtual bool GetState();
};
```

Corresponding Design Patterns

- Factory Method
- Prototype
- Iterator

OO Design Principles



S Single responsibility principle

O Open/closed principle

L Liskov substitution principle

I Interface segregation principle

D Dependency inversion principle

Building stable and flexible systems

Cargo cult programming

Are SOLID principles Cargo Cult?

It looks like a plane, but will it fly?

https://blog.ndepend.com/are-solid-principles-cargo-cult/