Design principle - SOLID

in object-oriented programming

How good is your design?

SOLID

S: Single responsibility principle

O: Open/closed principle

L: Liskov substitution principle

I: Interface segregation principle

D: Dependency inversion principle

Single responsibility principle

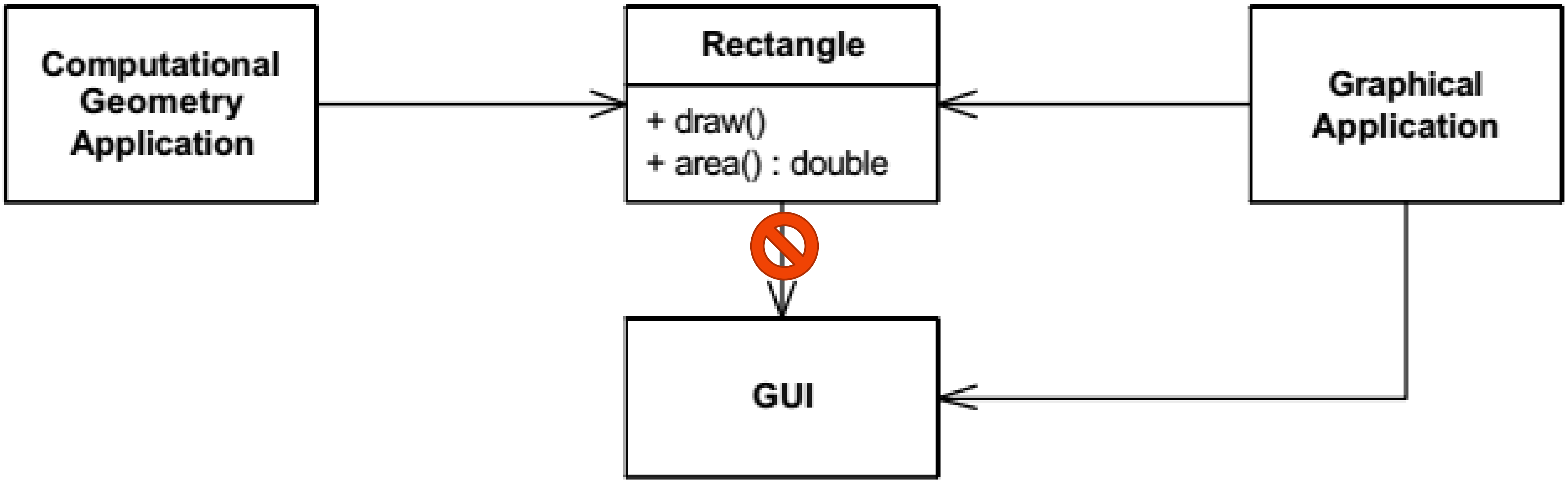
A class should have only one reason to change

Otherwise: responsibility coupled

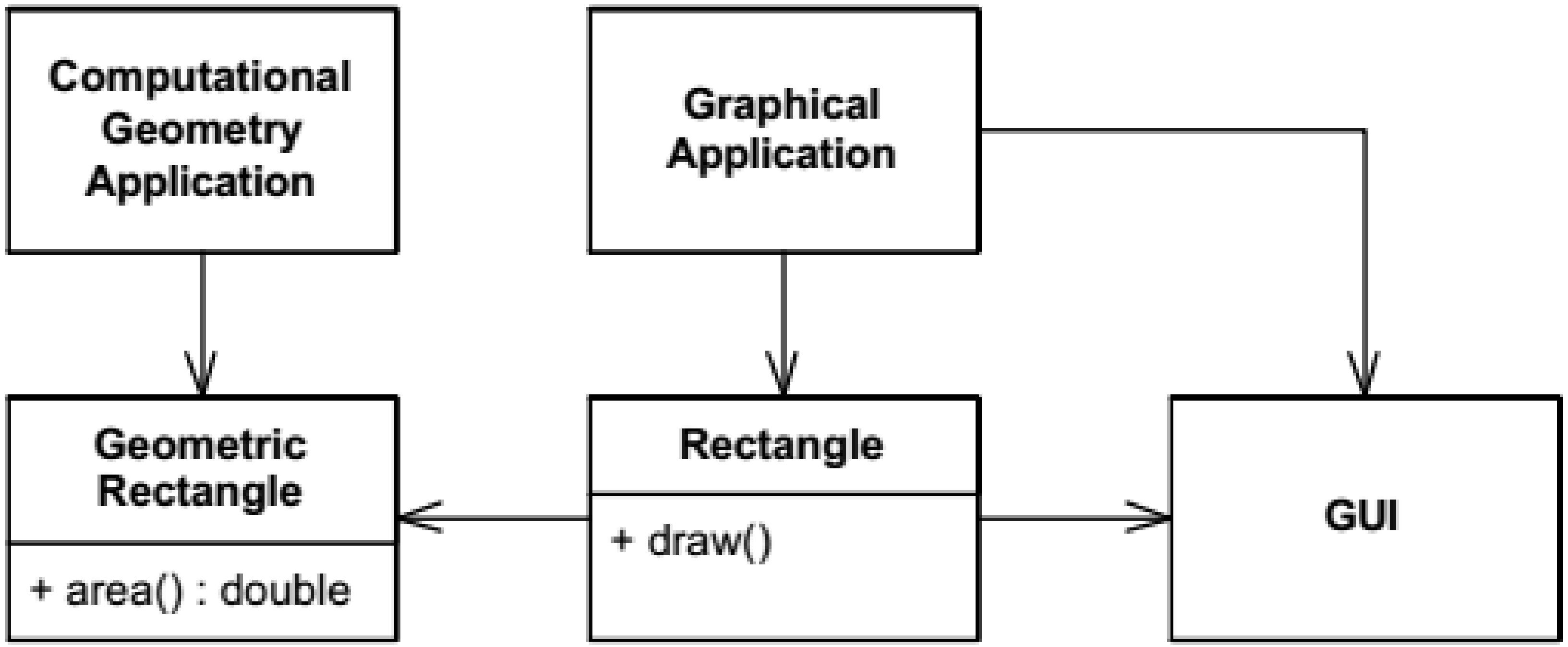
- Changes in one responsibility may **impair or inhibit** the class's ability to meet the others

→ Fragile design: one change my break the code in unexpected way

Example – violation of SRP



Example - SRP



Computational Geometry Application
Graphical Application
Geometric Rectangle
+ area() : double
Rectangle
+ draw()
GUI

Is it a violation of SRP ?

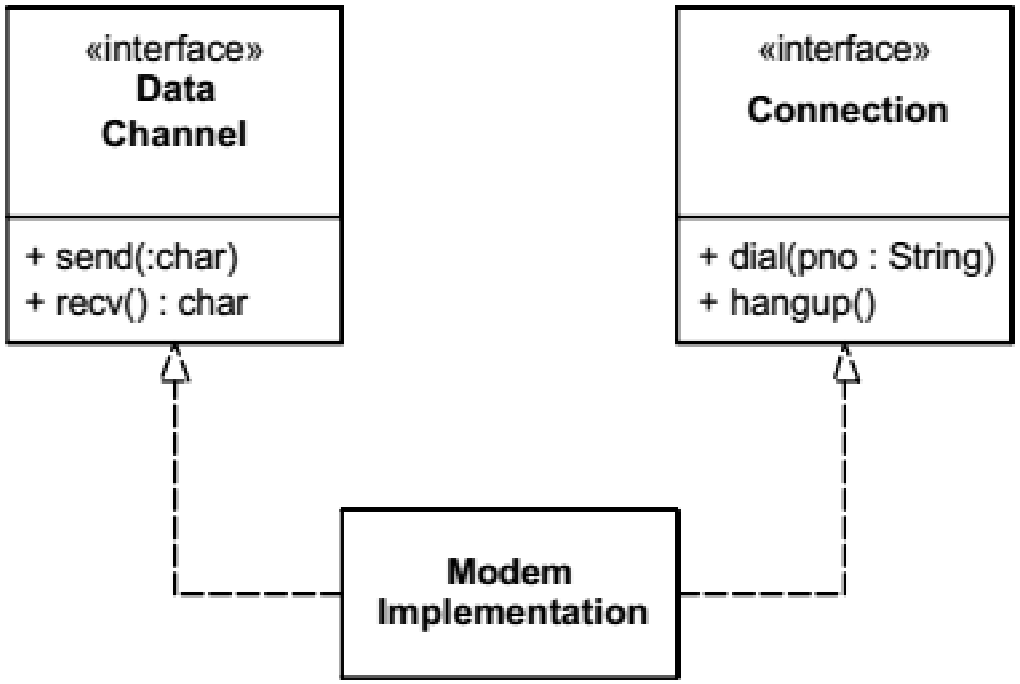
```
public class Modem {
  public void Dial(string pno);
  public void Hangup();
  public void Send (char c);
  public char Recv();
}
```

Connection management

Communication

Be careful with Needless Complexity

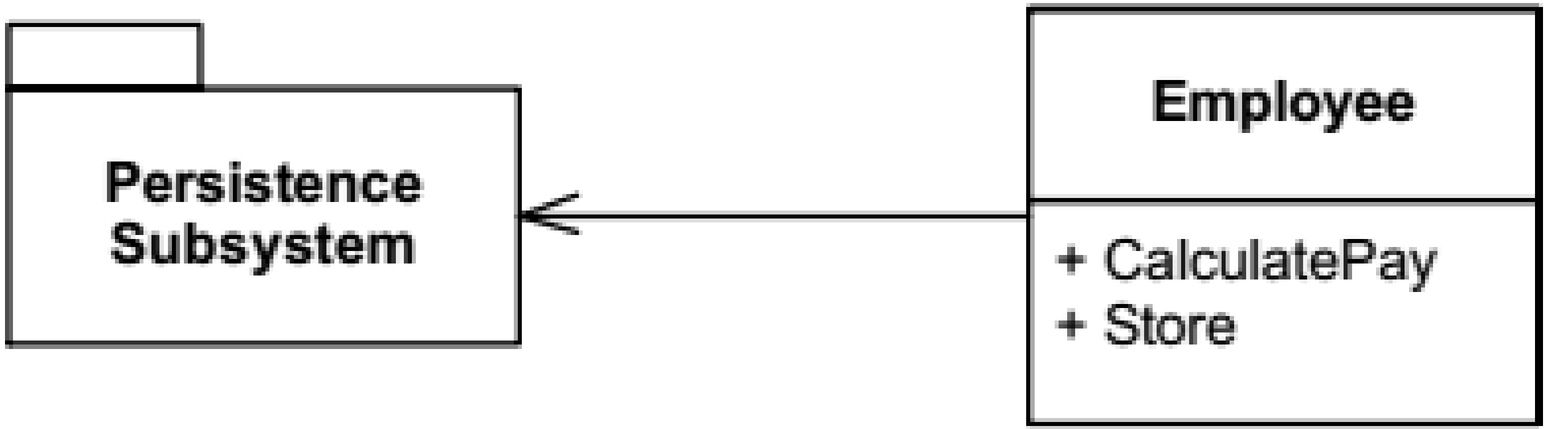
A better design?



Separate interfaces
vs
Coupled implementation

- Nobody except **`main`** needs to know the implementation (How?)

A common violation



Conclusion on SRP

- It is one of the simplest of the principles and one of the hardest to get right

- Finding and separating those responsibilities ≈ designing

Open/closed principle

Open/closed principle

Software entities (classes, modules, functions, ...) should be open for extension but closed for modification

Violation symptom:

A single change results in a cascade of changes to dependent modules

Open/closed principle

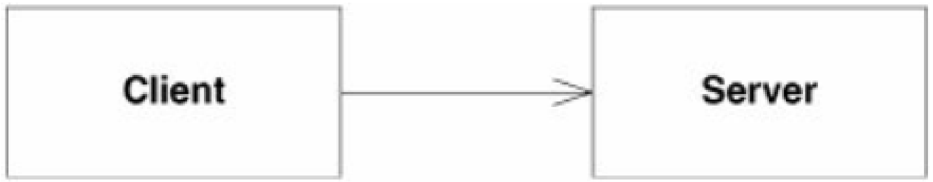
Open for extension: The behaviors of modules can be **extended**
- we can change what a module does

Closed for modification: Extending the behavior of a module **does not result in changes** to the source, or binary, code of the module

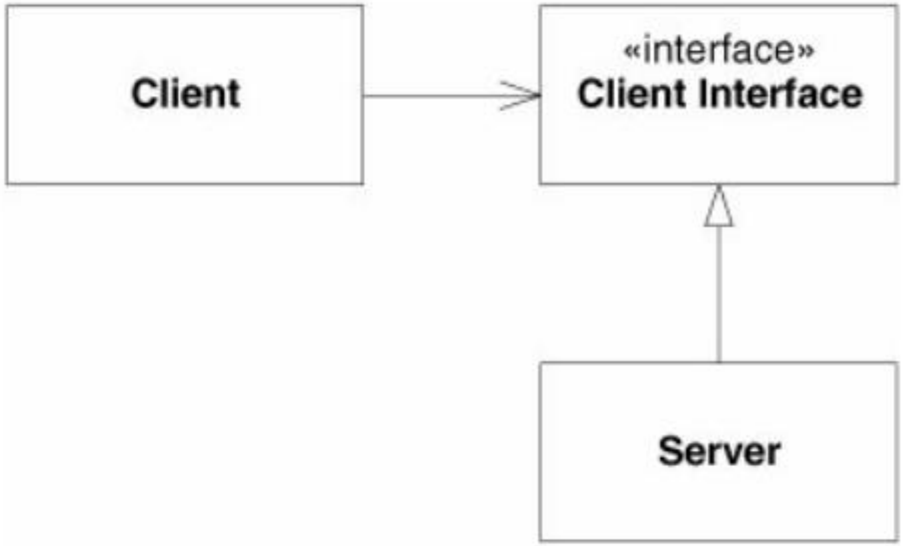
How?

Example of violation

NOT OPEN AND CLOSED



OPEN AND CLOSED



Example of OCP

```
void DrawAllShapes (Shape list[], int nShapes) {
  for (int i = 0; i < nShapes; i++) {
    if (list[i] instance of Square)
      drawSquare(list[i]);
    elseif (list[i] instance of Circle)
      drawCircle(list[i]);
  }
}
```

OOP solution to Square/Circle problem?

Strategic closure of modules

In general, no matter how "closed" a module is,
there will always be some kind of changes.

Closure cannot be complete, it must be strategic

- Choose the kinds of changes to prevent (after guessing)

- Construct abstractions to protect again those changes

OCP is expensive

- It takes time and effort to create appropriate abstractions.
  → increase the complexity of the design
- Apply OCP to changes that are likely to happen
  - **To keep away from needless complexity**
- What kind of changes are likely?

  Guess, research,

  or wait until the changes happen, then redesign to prevent further changes of that kind

Good practices

- Make all Member variables **private**

  We expect that any other class, including subclasses are ***closed*** against changes in those variables.

- No global variables **ever**

- Run time type identification (instance of) is **dangerous**

Conclusion on OCP

- OCP is at the heart of OOP design

- Conformance to this principle: reusability and maintainability

Liskov substitution principle

Liskov substitution principle

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

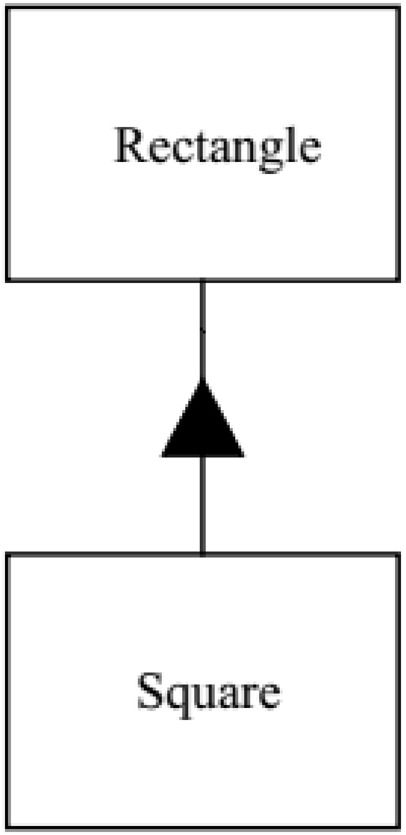
Violation consequence:

- One must know about all derivatives of that base class
- Violate Open/Closed principle

Example of LSP violation

```
void DrawAllShapes (Shape list[], int nShapes) {
  for (int i = 0; i < nShapes; i++) {
    if (list[i] instance of Square)
      drawSquare(list[i]);
    elseif (list[i] instance of Circle)
      drawCircle(list[i]);
  }
}
```

More subtle violation




```
public class Rectangle {
   private double height;
   private double width;

   public double getHeight();
    public double getWidth();
   public void setHeight(double
h);
   public void setWidth(double w);
 }
```

More subtle violation – some clues

```
public class Rectangle {
  private double height;
  private double width;

  public double getHeight();
  public double getWidth();
  public void setHeight(double h);
  public void setWidth(double w);
}
```

First clue of violation:
Square does not need both height and width

- waste of memory

But assume that the memory efficiency is not a concern

Second clue of violation

- The setHeight and setWidth functions are redundant.

More subtle violation – we tried

```
public void setWidth(double w)
{
  this.width = w;
  this.height= w;
}
public void setHeight(double
h) {
  this.width = h;
  this.height= h;
}
```

Usage

```
Square s = new Square();
f(s);

public void f(Rectangle
r) {
  r.setWidth(32);
}
```

BUT:

s.getHeight() returns 0;

Why?

Fixed with override/virtual

```
void g(Rectangle r) {
 r.setWidth(5);
 r.setHeight(4);
 area =
     r.getWidth() *
     r.getHeight();
 if (area == 20)
   //output OK;
 else
   // output error;
}
```

```
Rectangle r = new Rectangle();
Square s = new Square();
g(s);
g(r);
```

What are the output ?

What went wrong?

Isn't a `Square` a `Rectangle`?

No, a square might be rectangle but a `Square` object is not a `Rectangle` object

The Interface Segregation Principle

The interface segregation principle

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

- To deal with “fat” interfaces which can be broken up into groups of methods.
- Each group serves a different set of clients.

Segregate: keep one thing separate from another

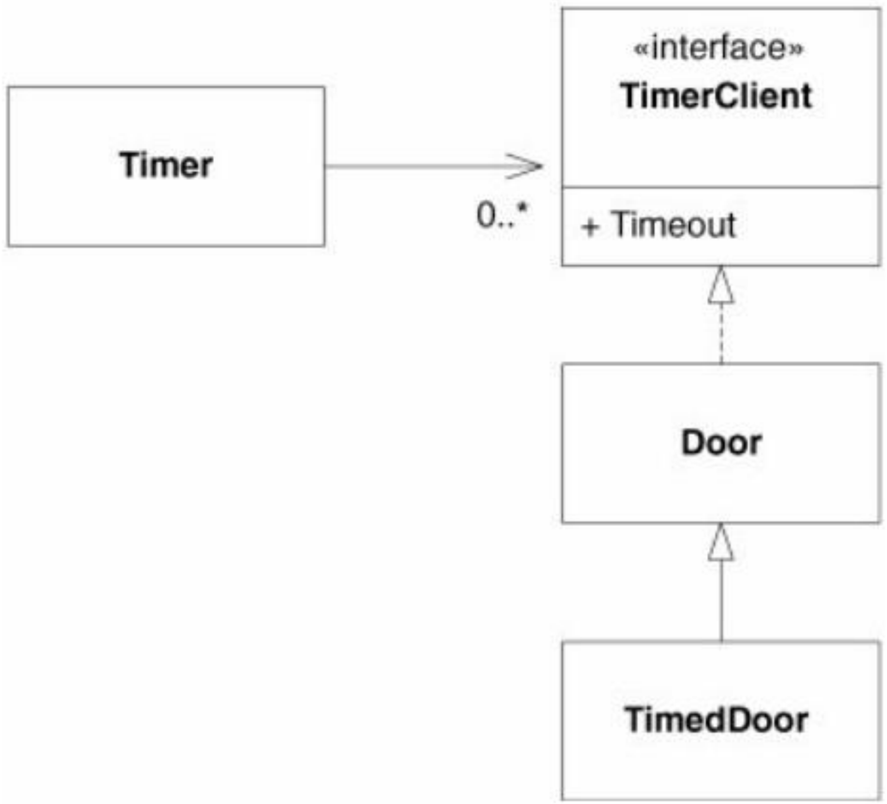
Example – Interface pollution

```
public interface Door {
  void Lock();
  void Unlock();
  bool IsDoorOpen();
}
```

Example – interface pollution

```
public class Timer {
  public void Register
    (int timeout,
     TimerClient client)
}
public interface TimerClient
{
  void TimeOut();
}
```

A COMMON APPROACH



Problem

The `Door` class is now depends on `TimerClient`
and the `Door` abstraction has nothing to do with timing !

If non-timing derivatives of `Door` are created,
an degenerate implementation of `TimeOut` must be provided

→A potential violation of Liskov Substitution Principle

→The application of those non-timing derivatives of Door has to import `TimerClient` even though it is not used

The “fat” interface

The interface of Door has been polluted with a method that it does not required

The added method is solely for the benefit of one of its subclass

→ Pursing that design, for the needs of many subclasses,
the interface of the Door become “fat”

Forces that cause change in software

1. Changes to interface will affect their users

2. And sometimes users forces a change to the interface

Back to our example on the `Door` and the `TimingDoor`

There could be multiple timeouts and we need to distinguish them

Handle multiple timeouts

```
public class Timer {

 public void Register
   (int timeout,
    int timeOutId,
    TimerClient client)


}


public interface TimerClient{
 void TimeOut(int timeOutID);
}
```

To fix a bug in Timer:

We have to change the `Door` class

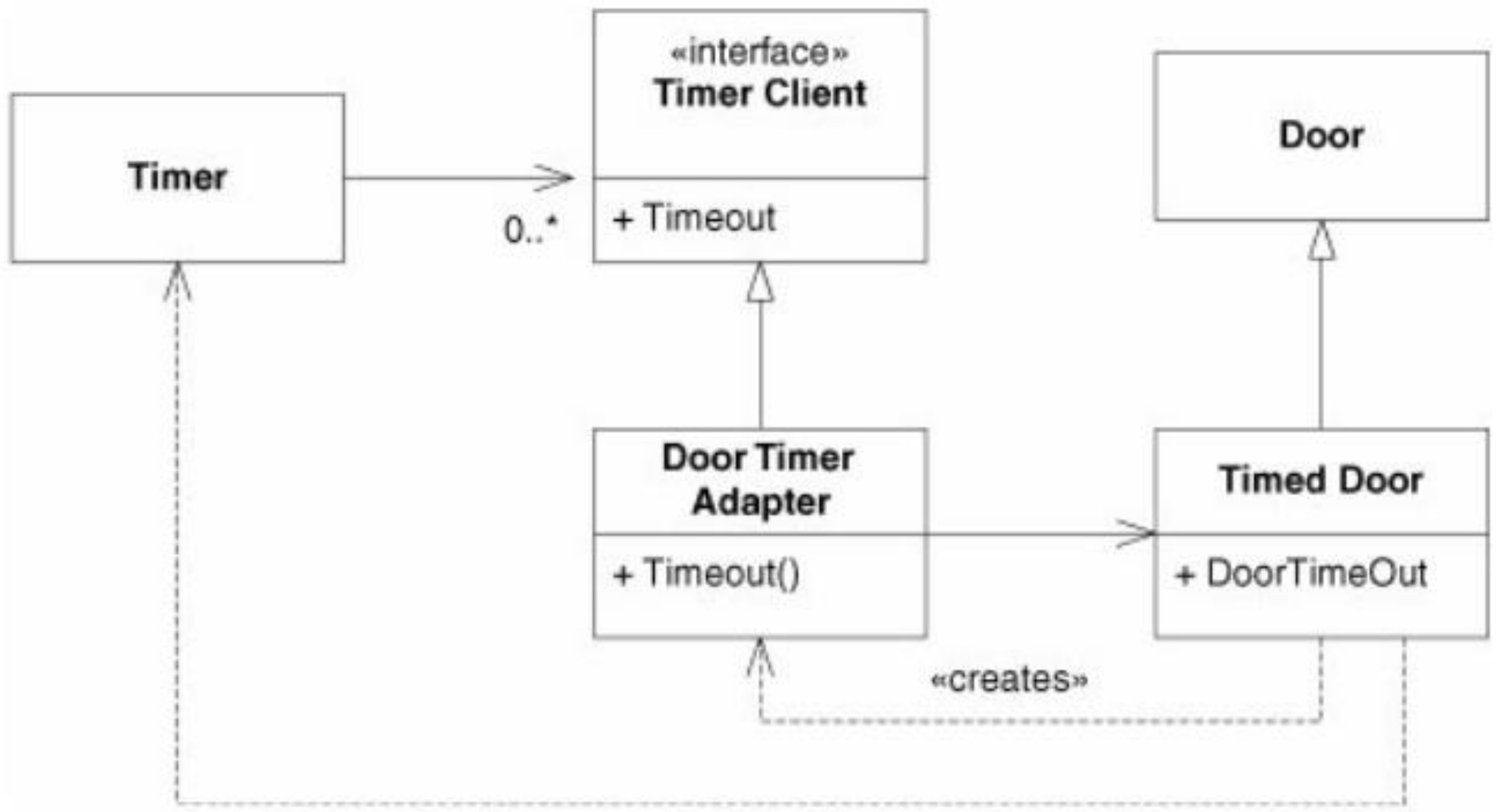
and all of its clients,

even though they don't use `Timer`

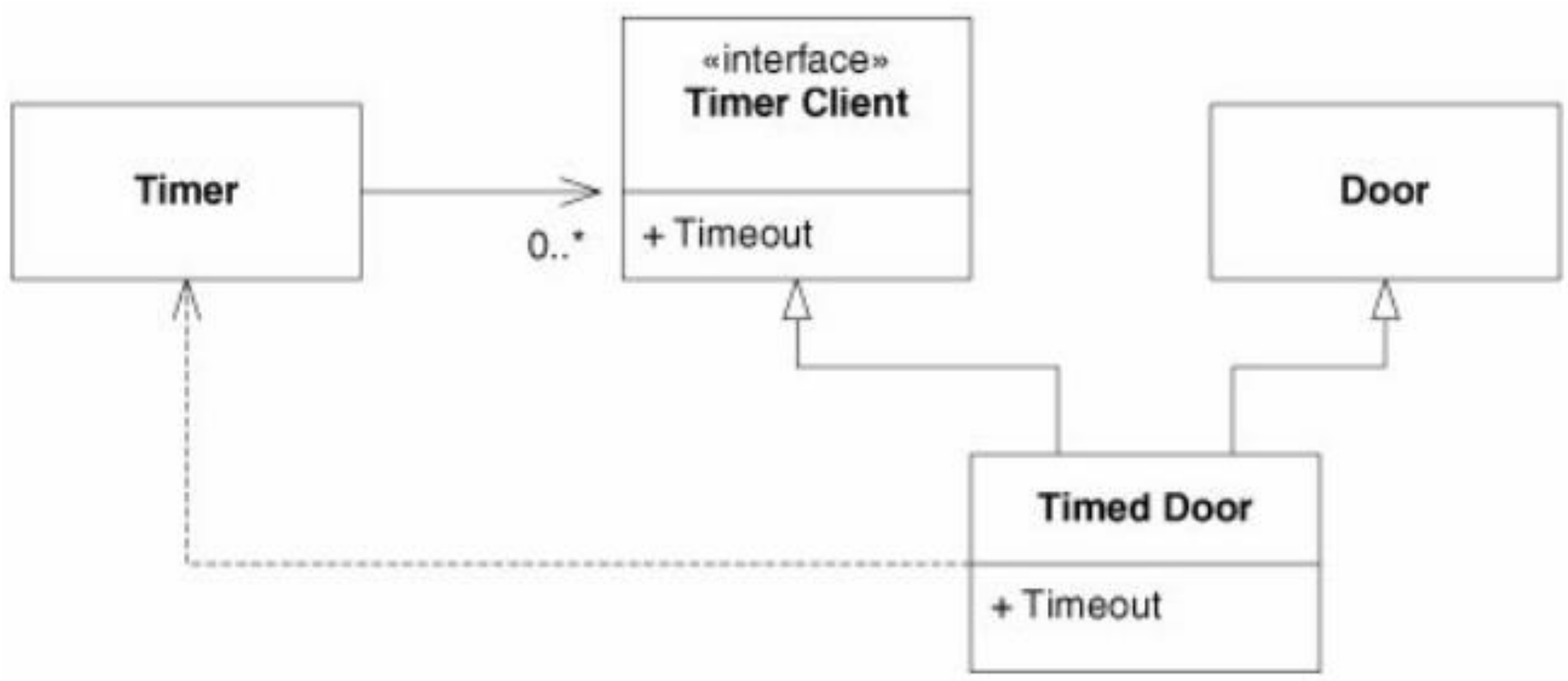
→ `should avoid such coupling`

→ `should separate the interfaces`

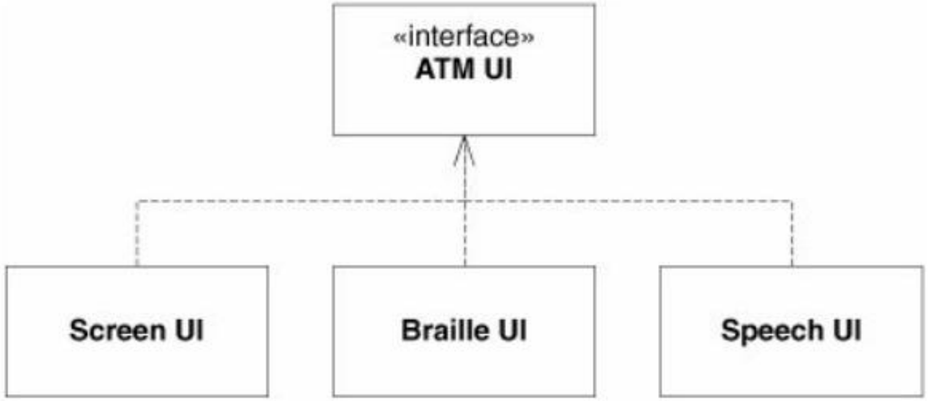
Interfaces Segregation – Through delegation

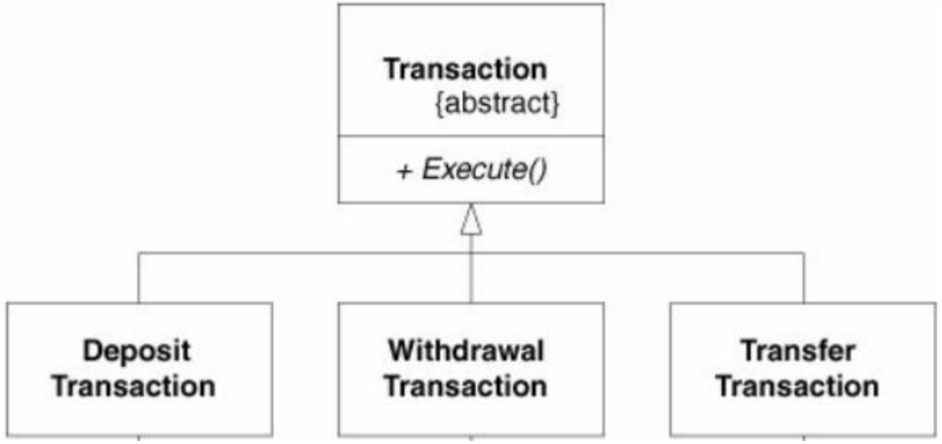


Interfaces Segregation – through multiple inheritance/implementation



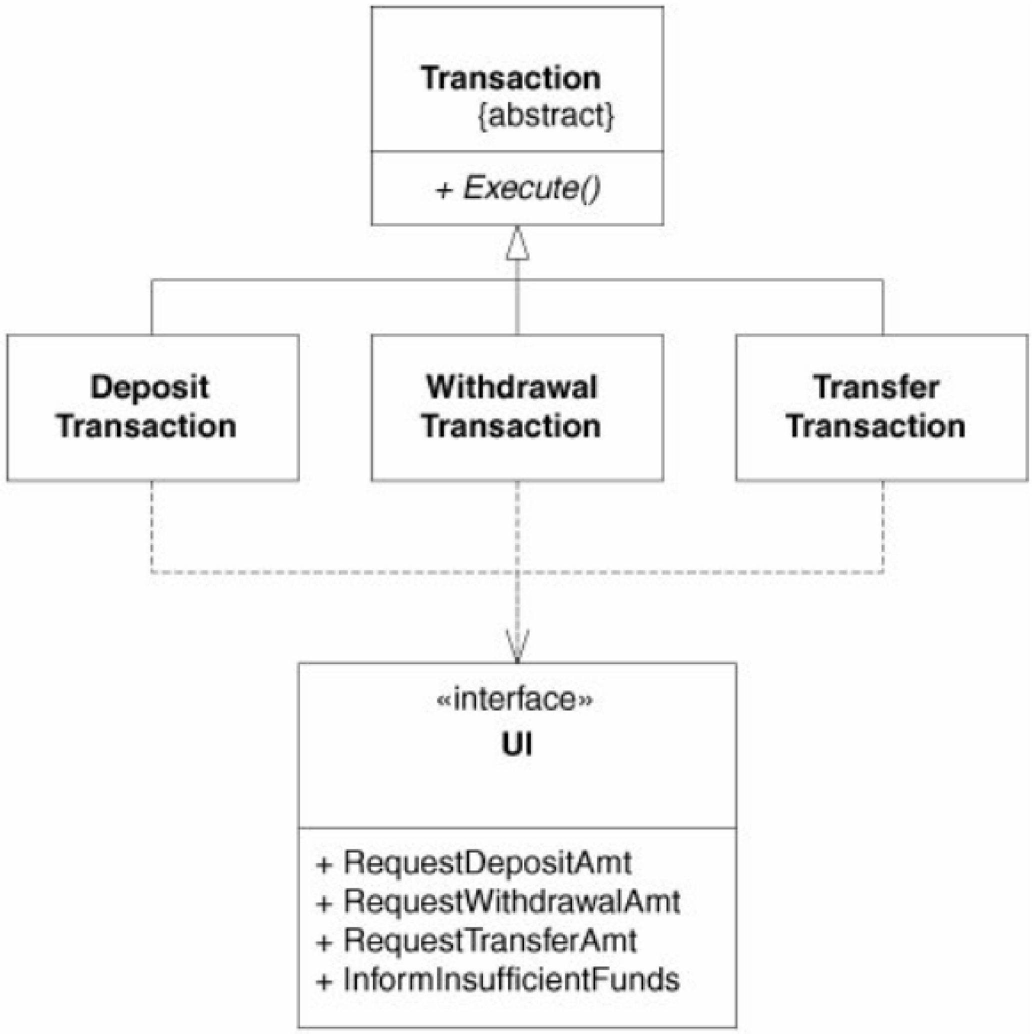
Example – ATM user interface





Each of these transaction classes invokes UI methods.
For example, in order to ask the user to enter the amount to be deposited,
the **DepositTransaction** object invokes the **RequestDepositAmount** method of the UI class.
Likewise, in order to ask the user how much money to transfer between accounts, the
TransferTransaction object calls the **RequestTransferAmount**, and
WithdrawalTransaction object calls the **RequestWithdrawalAmount** method of UI

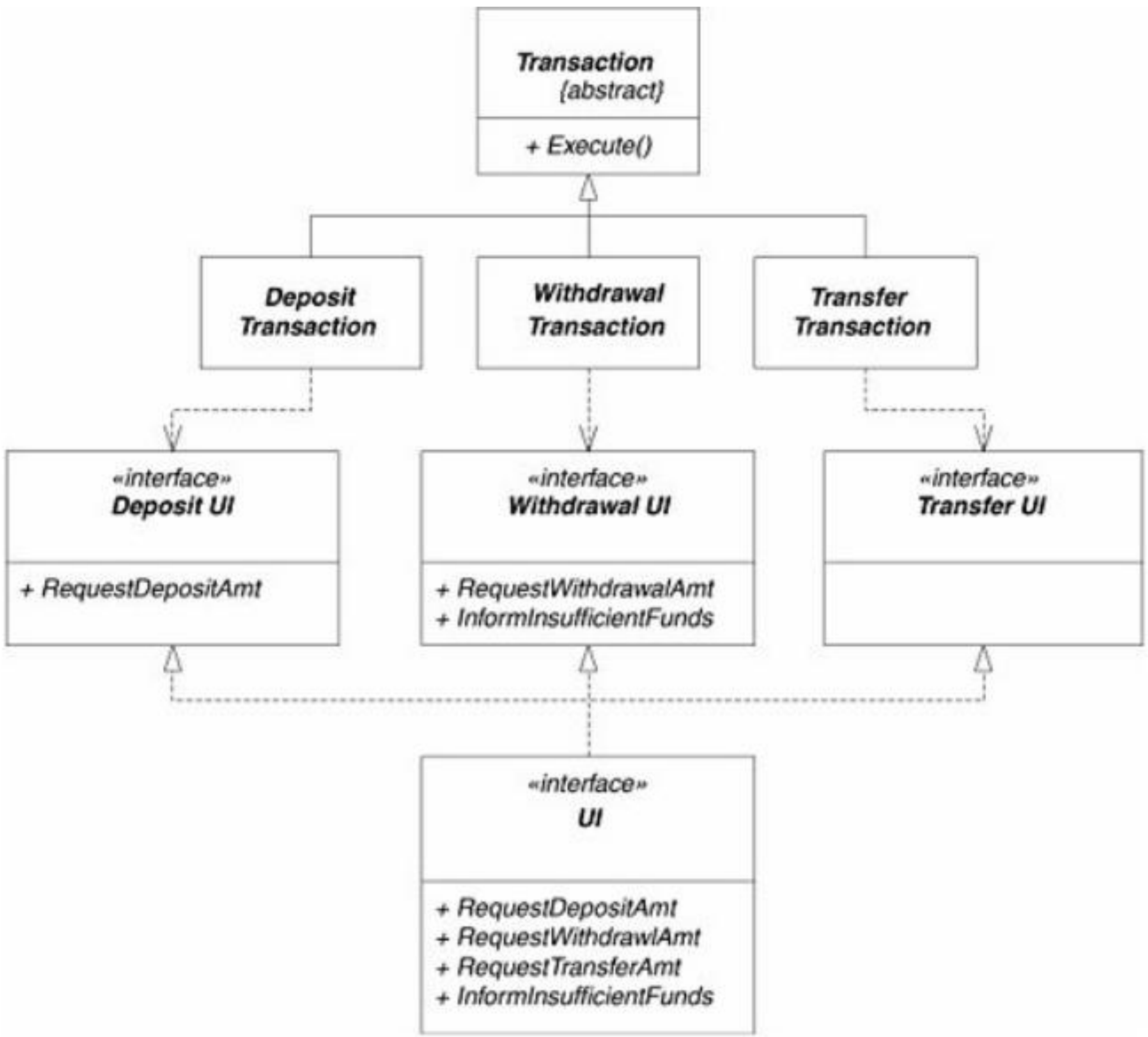
An solution for ATM interface



It violates the interface segregation principle

A good solution

- Each new derivative of the **Transaction** creates a new base interface for **UI**



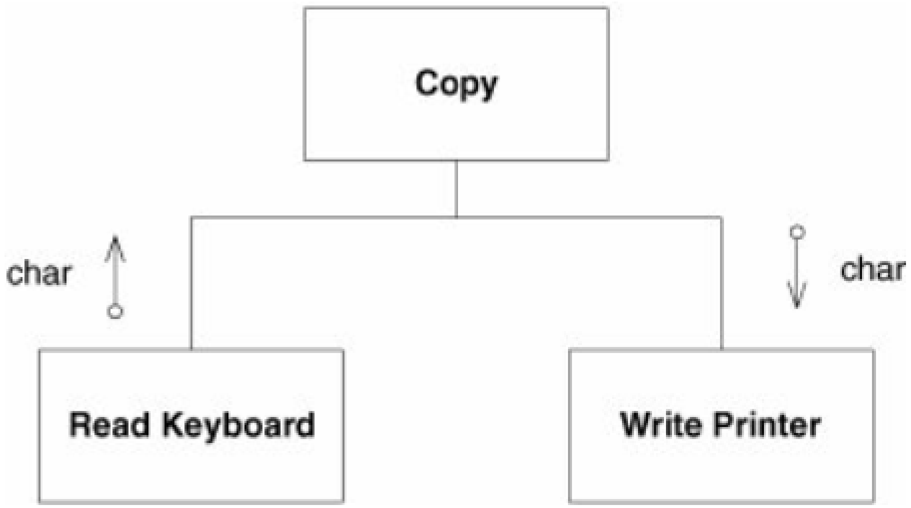
Dependency-Inversion Principle

Dependency-Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

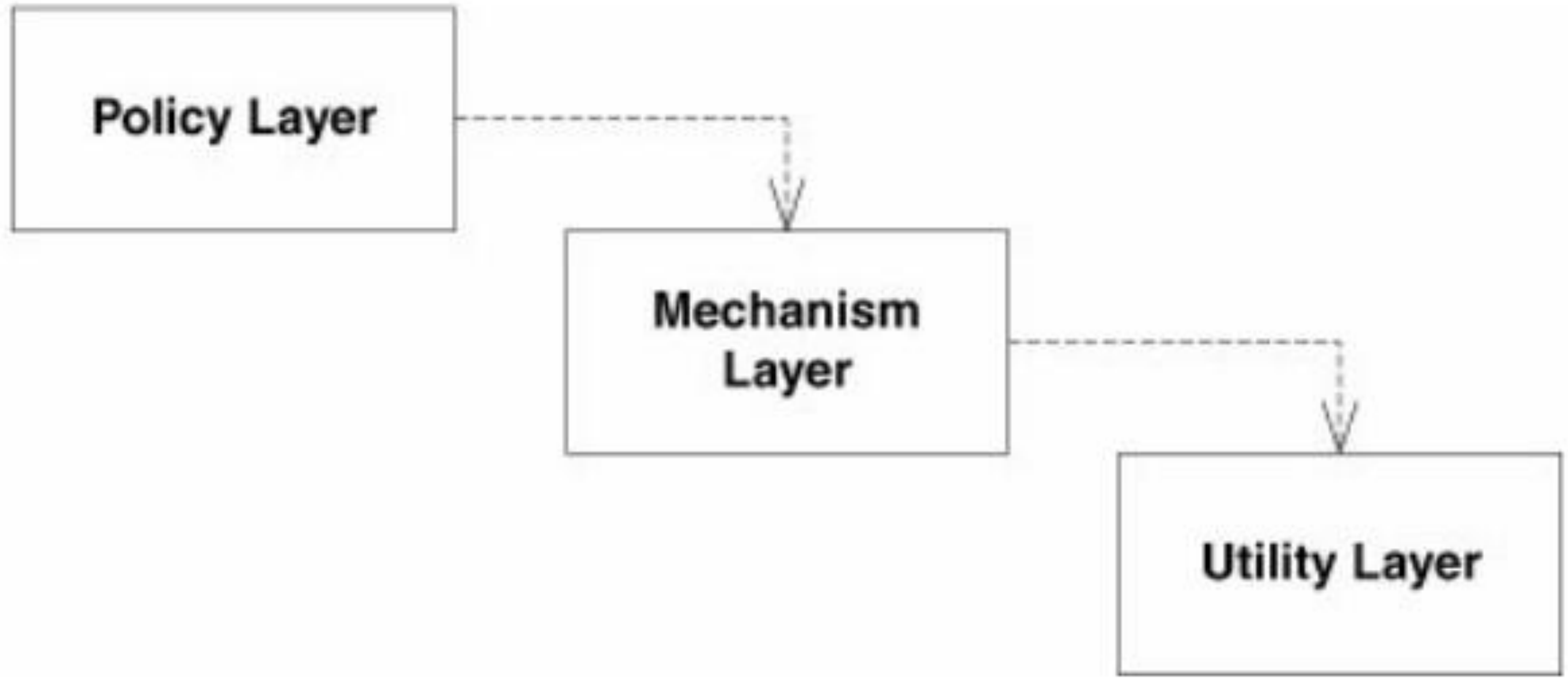
Abstractions should not depend upon details. Details should depend upon abstractions.

Example of violation

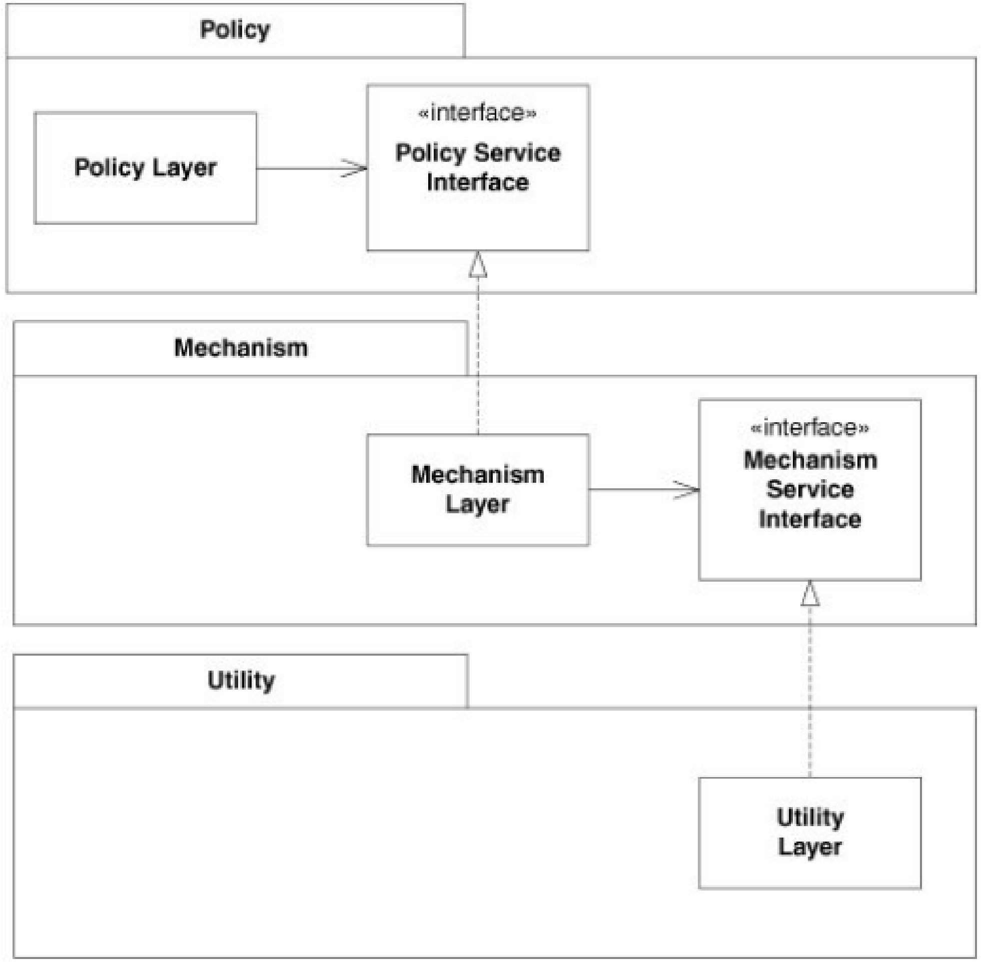



```
public static void Copy()
{
  int c;
  while((c=Keyboard.Read()) != -1)
    Printer.Write(c);
}
```

Example of violation - 2



A better design – dependency inversion



Ownership inversion

Don’t call us; we’ll call you

- The lower-level modules provides the implementation for interfaces that are declared within the upper-level modules.

Dependence on Abstractions

A heuristic to conforms the DIP:

A class/a client should not depend on a concrete class, and rather on an abstract class or an interface

Example

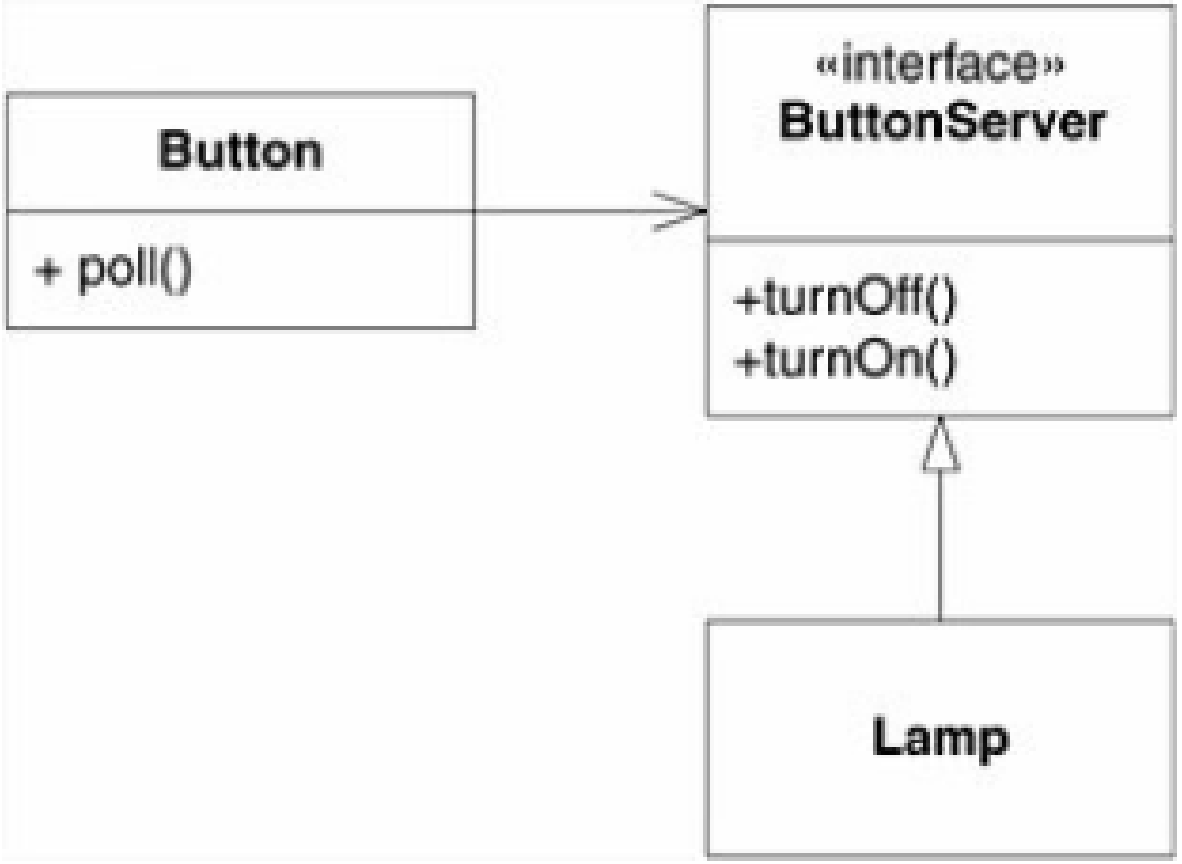
A **Button** object and a **Lamp** object

User press the button, then the **button** controls the **lamp.**


```
public class Button
{
  private Lamp lamp;
  public void Poll()
  {
    if (/*some condition*/)
      lamp.TurnOn();
  }
}
```

Can we reuse the button to control a motor ? NO

Example – a better design



- **Button** can control any device that implements the **ButtonServer** interface

- Lamp does not depend on **Button** but only **ButtonServer**

An example

Consider the software that might control the regulator of a furnace.

The software can **read** the current temperature from an I/O channel

and instruct the furnace to **turn on or off** by sending commands to a different I/O channel.

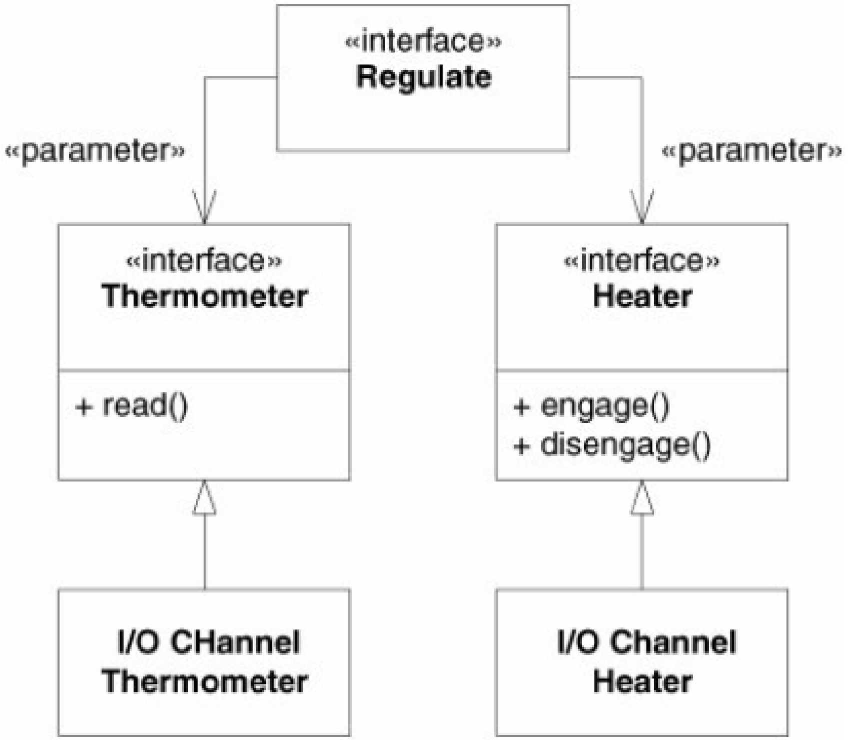
Furnace controller

```
const byte TERMOMETER = 0x86;
const byte FURNACE = 0x87;
const byte ENGAGE = 1;
const byte DISENGAGE = 0;

void Regulate(double minTemp, double maxTemp)
{
  for(;;)
  {
    while (in(THERMOMETER) > minTemp)
      wait(1);
    out(FURNACE,ENGAGE);

    while (in(THERMOMETER) < maxTemp)
      wait(1);
    out(FURNACE,DISENGAGE);
  }
}
```

A solution




```
void Regulate(Thermometer t, Heater h,
      double minTemp, double maxTemp)
{
  for(;;)
  {
    while (t.Read() > minTemp)
      wait(1);
    h.Engage();

    while (t.Read() < maxTemp)
      wait(1);
    h.Disengage();
  }
}
```

Conclusion

DIP is the hallmark of good object-oriented design

For a program,

If its dependencies are inverted, it has an OO design

If its dependencies are not inverted, it has an procedural design

Conclusion

DIP is necessary for the creation of reusable components

DIP is critically important for the construction of code that is resilient to change