# Information System Design

## Lecture 3:

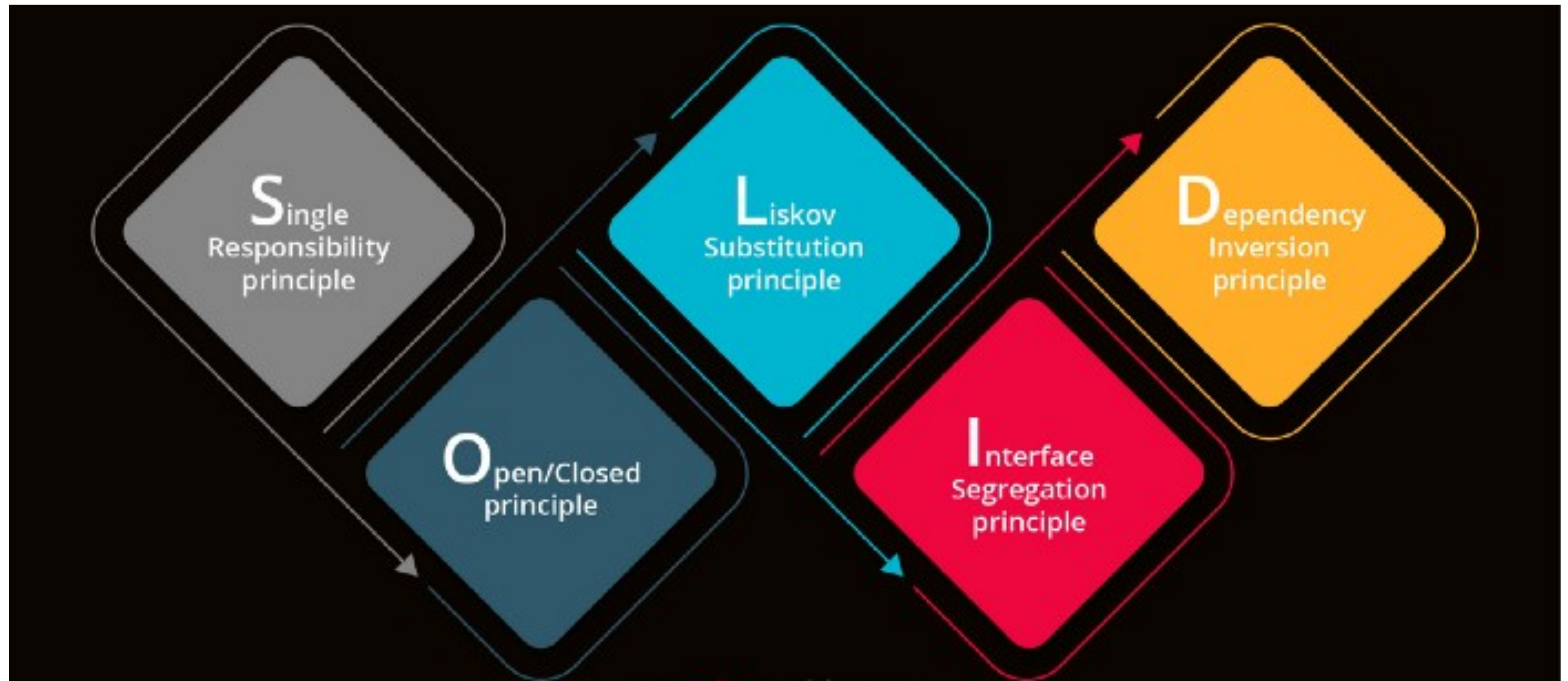
**Dr. Moustafa Alzantot**

# SOLID principles revised.

What are the SOLID principles ?

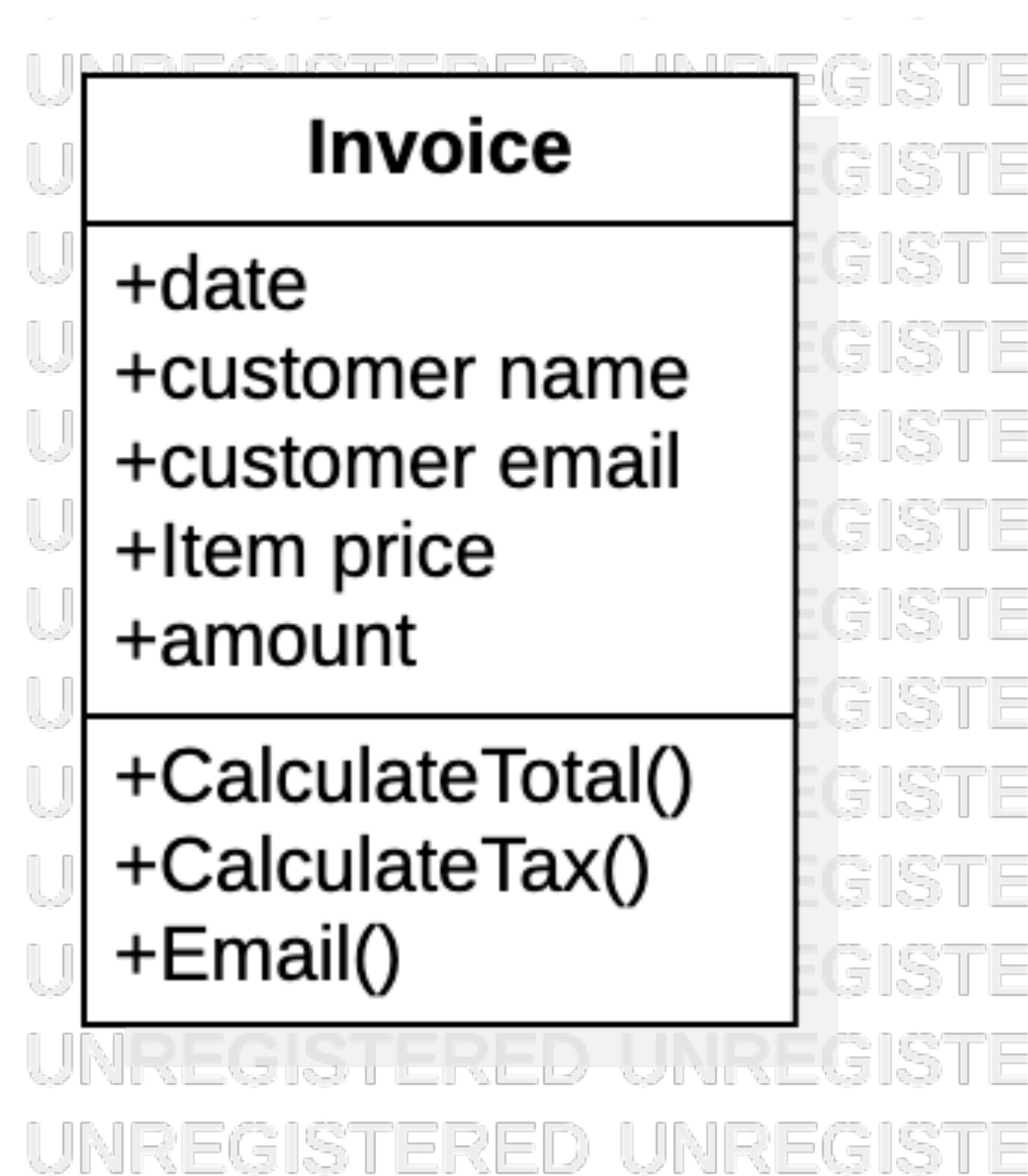
# SOLID principles revised.

What are the SOLID principles ?



# SOLID principles revised.

**Quiz:** comment about this OO design and improve, if possible.



# Open-Closed Principle

- The Open-Closed Principle

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

Modules that satisfy (OCP) principle are :

**Open for extension**: this means their behavior can be extended. If the requirements of the application change, we can extend the module with new behaviors to satisfy the requirements change.

**Closed for modification**: extending the behavior of the module doesn't result in changes to source or binary of the module. The binary executable version (e.g. DLL or java JAR) remains unchanged.

# Open-Closed Principle

**How can a module be both open for extension and closed for modification at the same time ?**

Use abstractions and Polymorphism. Abstractions are abstract base classes and that could be extended by an unbounded group of possible behaviors through derivative classes.

A module that relies on abstract class is closed for modification because the abstract class remains unchanged. Yet the behavior can be extended by creating a new derivative of the abstraction.

# Open-Closed Principle

**Figure 9-1** Client is not open and closed

Source : Agile Software Development, Principles, Patterns and Practices

Both `Client` and `Server` are concrete classes.

The `Client` uses `Server` class, if we wish to change a different server object, the `Client` class must be changed.

# Open-Closed Principle

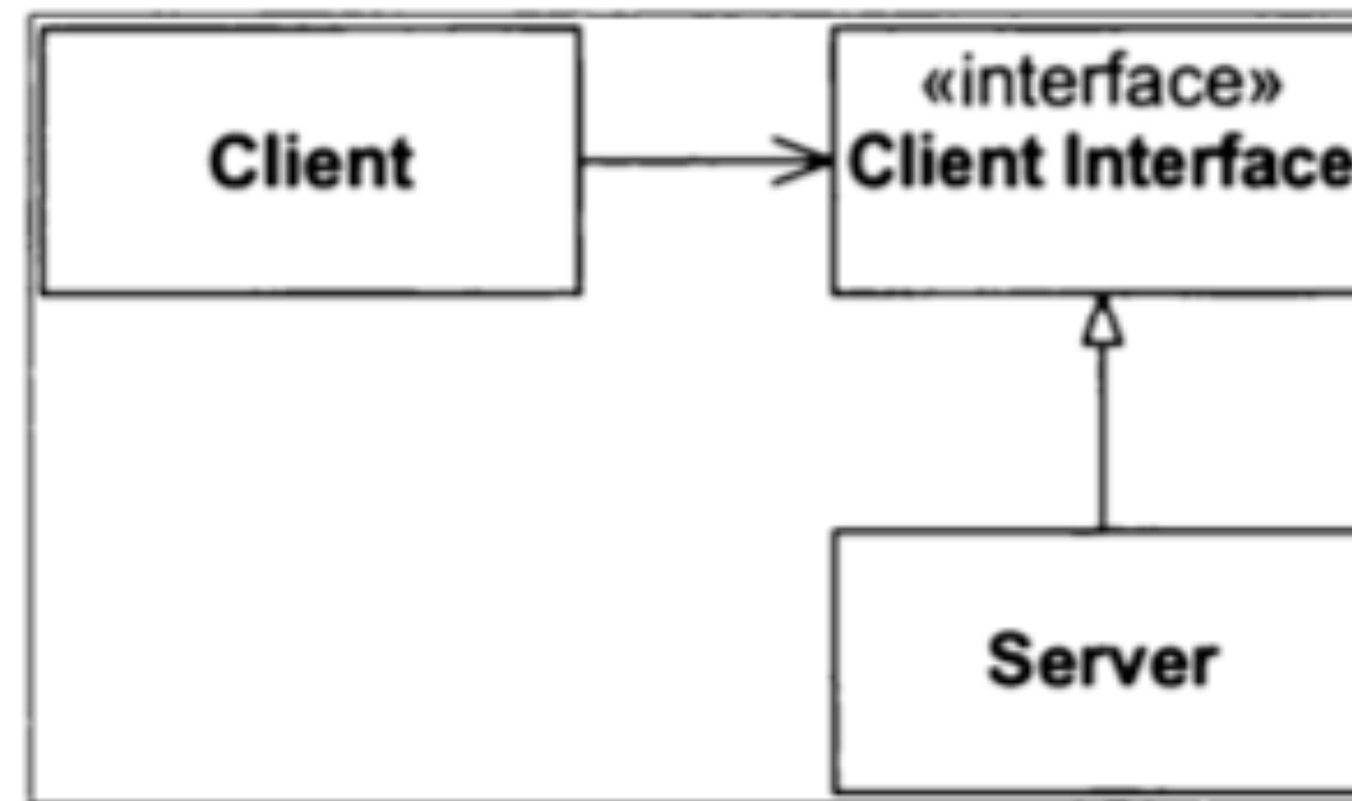



**Figure 9-2** STRATEGY pattern: Client is both open and closed

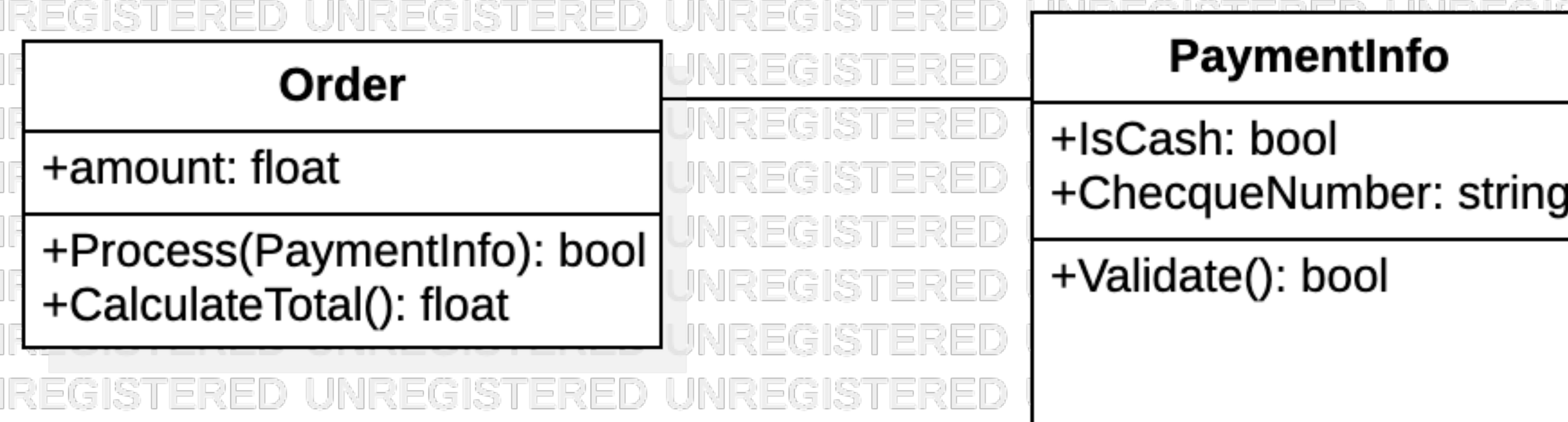
Source : Agile Software Development, Principles, Patterns and Practices

`Client` needs some work to get done, it can describe it in terms of abstract interface “`ClientInterface`”.

Sub-types of `ClientInterface` can implement the interface in any manner the choose.

# SOLID principles revised.

**Quiz 2:** How can we handle the new requirements of adding "CreditCard Payment" ?




```
bool Process(const PaymentInfo& payment_info) {
  If (payment_info.IsCash) {
  // Process cash payment
  } else {
  // Process cheque payment using checqueNumber
  }
}
```

# Liskov Substitution Principle

Liskov Substitution Principle can be phrased as :

Subtypes must be substitutable for their base types.

**In other words,**

if an object inherits from another, it should be able to replace its parent elements in a program and not have the program break or have to create exceptions.

# Liskov Substitution Principle

## Example

IS-A relationship represented by inheritance

`Square` class inherits from `Rectangle` class.

**Listing 10-2**

**Rectangle class**

```
class Rectangle
{
  public:
    void   SetWidth(double w)  {itsWidth=w;}
    void   SetHeight(double h) {itsHeight=w;}
    double GetHeight() const   {return itsHeight;}
    double GetWidth() const    {return itsWidth;}
  private:
    Point  itsTopLeft;
    double itsWidth;
    double itsHeight;
};
```

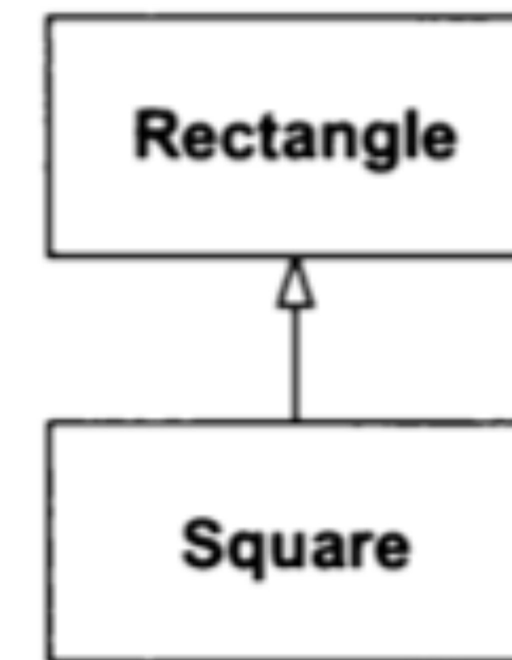



**Figure 10-1** Square inherits from Rectangle

# Liskov Substitution Principle

## Example

Does `Square` need both height and width ?

```
void Square::SetWidth(double w)
{
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}
```

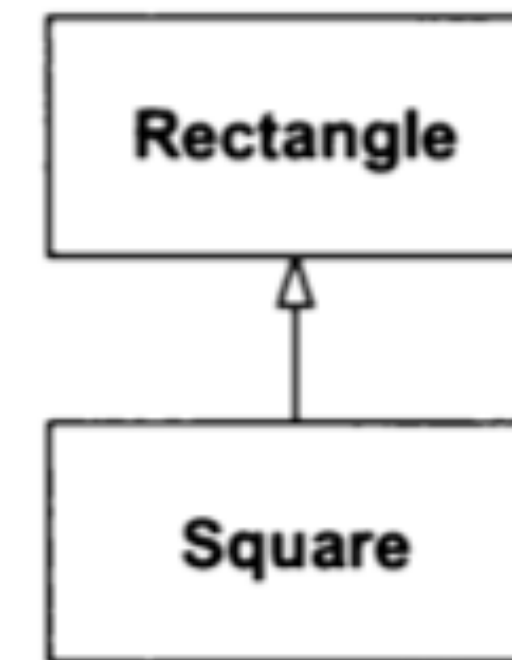



**Figure 10-1** Square inherits from Rectangle

# Liskov Substitution Principle

## Example

```
Square s;
s.SetWidth(1); // Fortunately sets the height to 1 too.
s.SetHeight(2); // sets width and height to 2. Good thing.
```

The `Square` invariants remain correct, but what if you pass a `Square` object to the following function.

```
void f(Rectangle& r)
{
  r.SetWidth(32); // calls Rectangle::SetWidth
}
```

**What has gone wrong ?**

# Liskov Substitution Principle

## Example

Program can be fixed by marking SetWidth and SetHeight methods as virtual.

However, if the creation of derived class causes us to make changes in base class, this is a sign of faulty design.

***Another reason this is a bad design:*** it is fair to assume that changing the height won't affect the width.

Therefore the author of Square has violated an invariant of Rectangle, not an invariant of Square.

# Interface Segragation Principle

**Interface-Segregation Principle states**

> Clients should not be forced to depend on methods that they do not need to use.

# Interface Segragation Principle

Example:

Suppose you want to represent a multifunction device that can print, scan and  also fax documents.

You can define an interface for it like that

```
1   struct IMachine
2   {
3     virtual void print(vector<Document*> docs) = 0;
4     virtual void fax(vector<Document*> docs) = 0;
5     virtual void scan(vector<Document*> docs) = 0;
6   };
```

# Interface Segragation Principle

What is the problem with that ?

- If there is some device that implements this interface but wants only to do scanning but not printing or sending faxes.

# Interface Segragation Principle

Better design

Define separate interfaces for each task.

A concrete class will implement only interfaces for tasks it can handle.

A concrete class can implement as many services as it needs.

```
1   struct IPrinter
2   {
3     virtual void print(vector<Document*> docs) = 0;
4   };
5
6   struct IScanner
7   {
8     virtual void scan(vector<Document*> docs) = 0;
9   };
```

# Dependency Inversion Principle

Dependency Inversion Principle:

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

# Dependency Inversion Principle

Example:

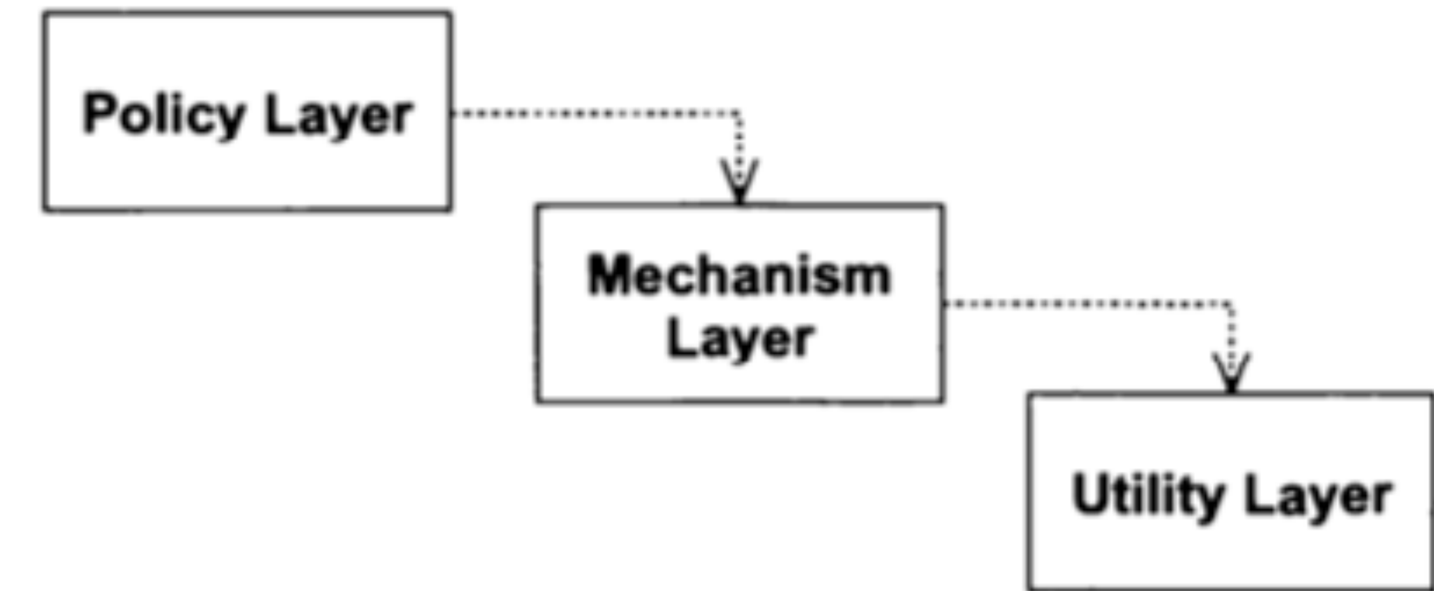


**Figure 11-1** Naive layering scheme

PolicyLayer uses a lower level MehchanismLayer.

MechanismLayer depends on a lower UtilityLayer.

Dependency is transitive, PolicyLayer depends on changes from both mechanism and utility layers.

# Dependency Inversion Principle

Re-design:

- Upper layer define an interface for services they need.
- Lower layers are realized from these abstract interfaces.
- Each higher level layer uses next lower layer through the abstract interface.

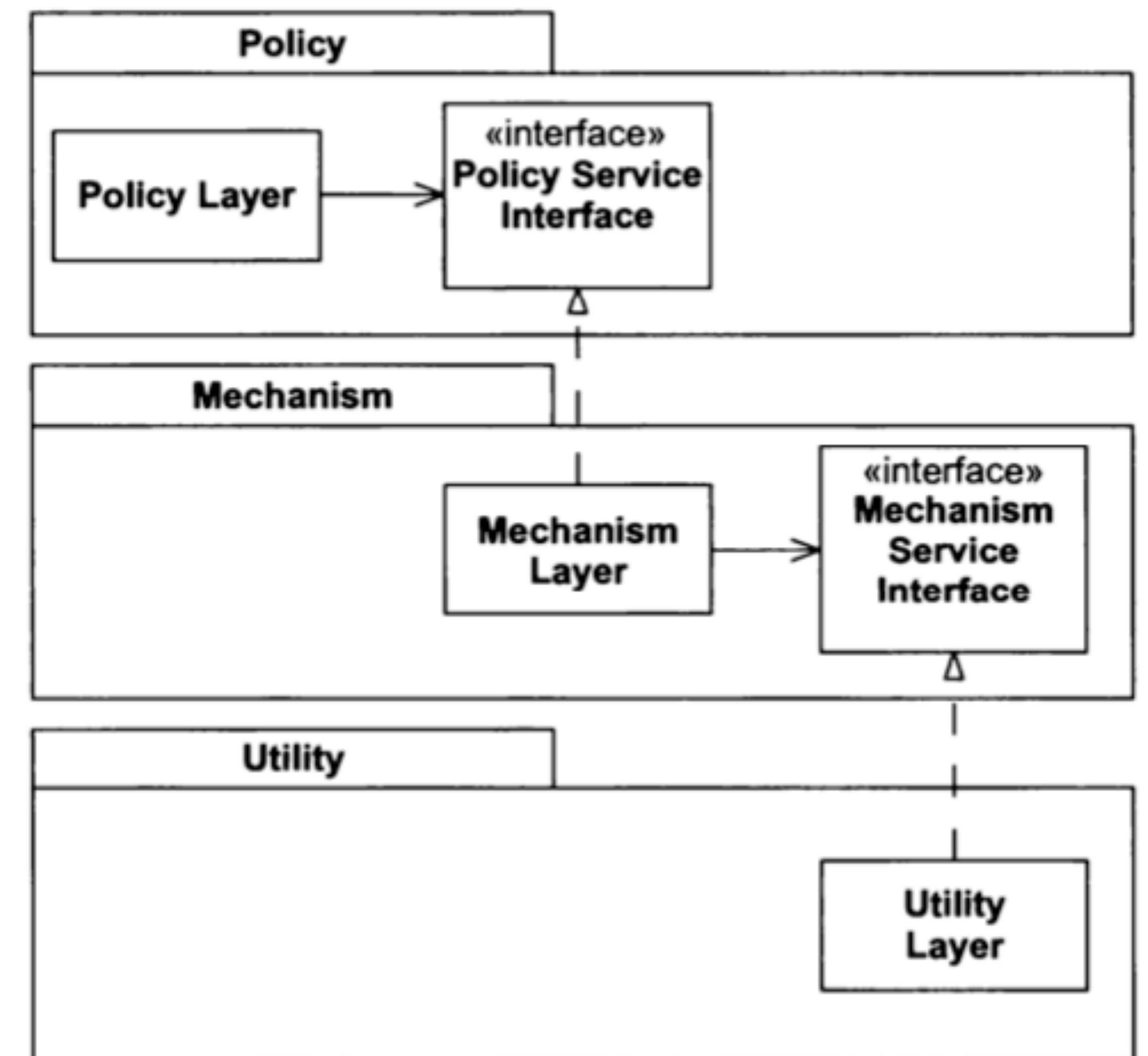


Figure 11-2 Inverted Layers

# Dependency Inversion Principle

Higher layer do not depend on lower layers, but instead lower layers depend on abstract services declared in upper layers.

This also breaks the transitive dependency between “`PolicyLayer`” and “`UtilityLayer`”.

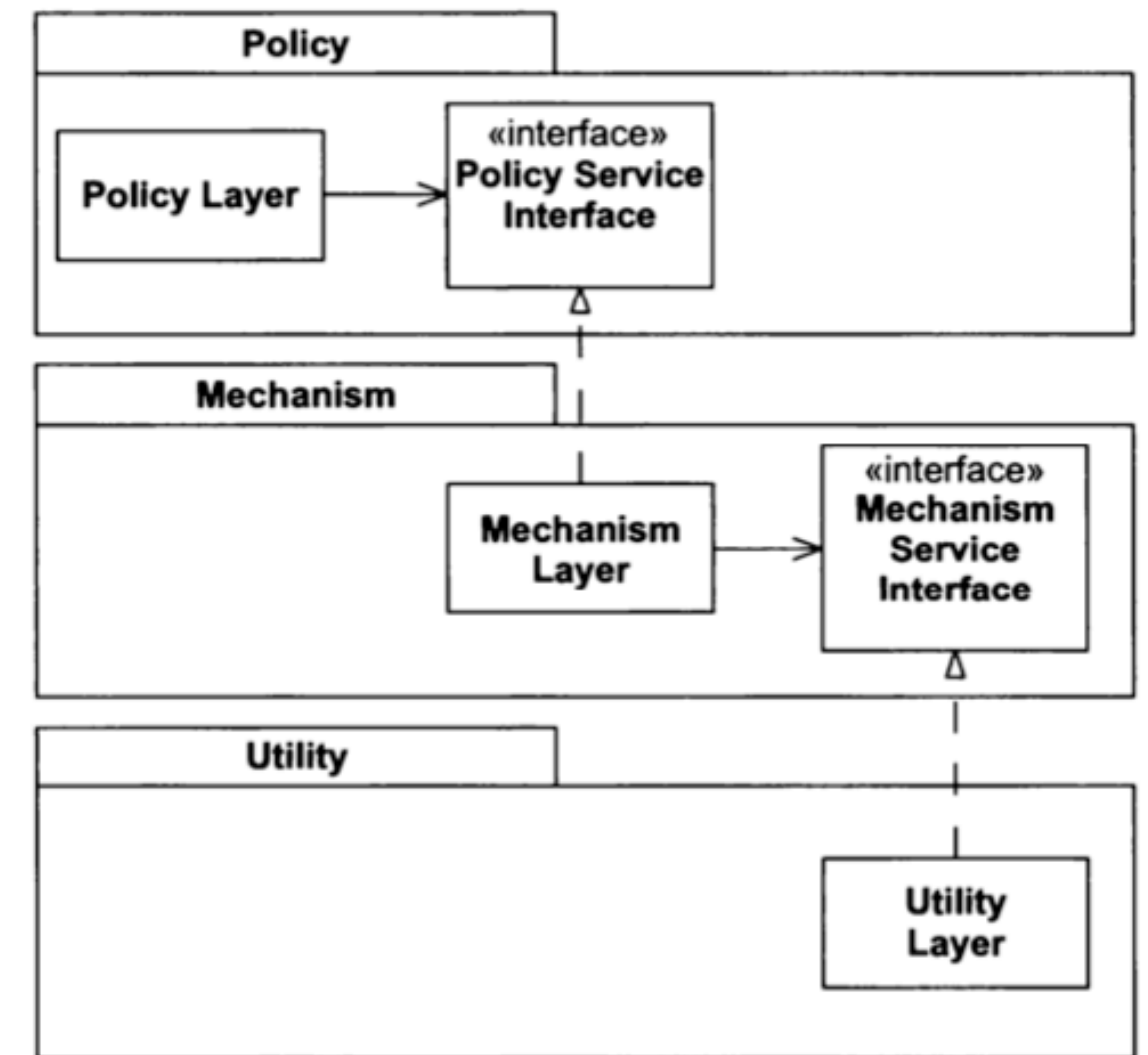


**Figure 11-2** Inverted Layers

# Dependency Inversion Principle

**Pros of DIP redesign:**

- Higher level (`PolicyLayer`) module is unaffected by changes in `MechanismLayer` or `UtilityLayer`.

- `PolicyLayer` can be reused in any context that defines low-level modules that conform to `PolicyServiceInterface`. Therefore, the structure is more flexible.

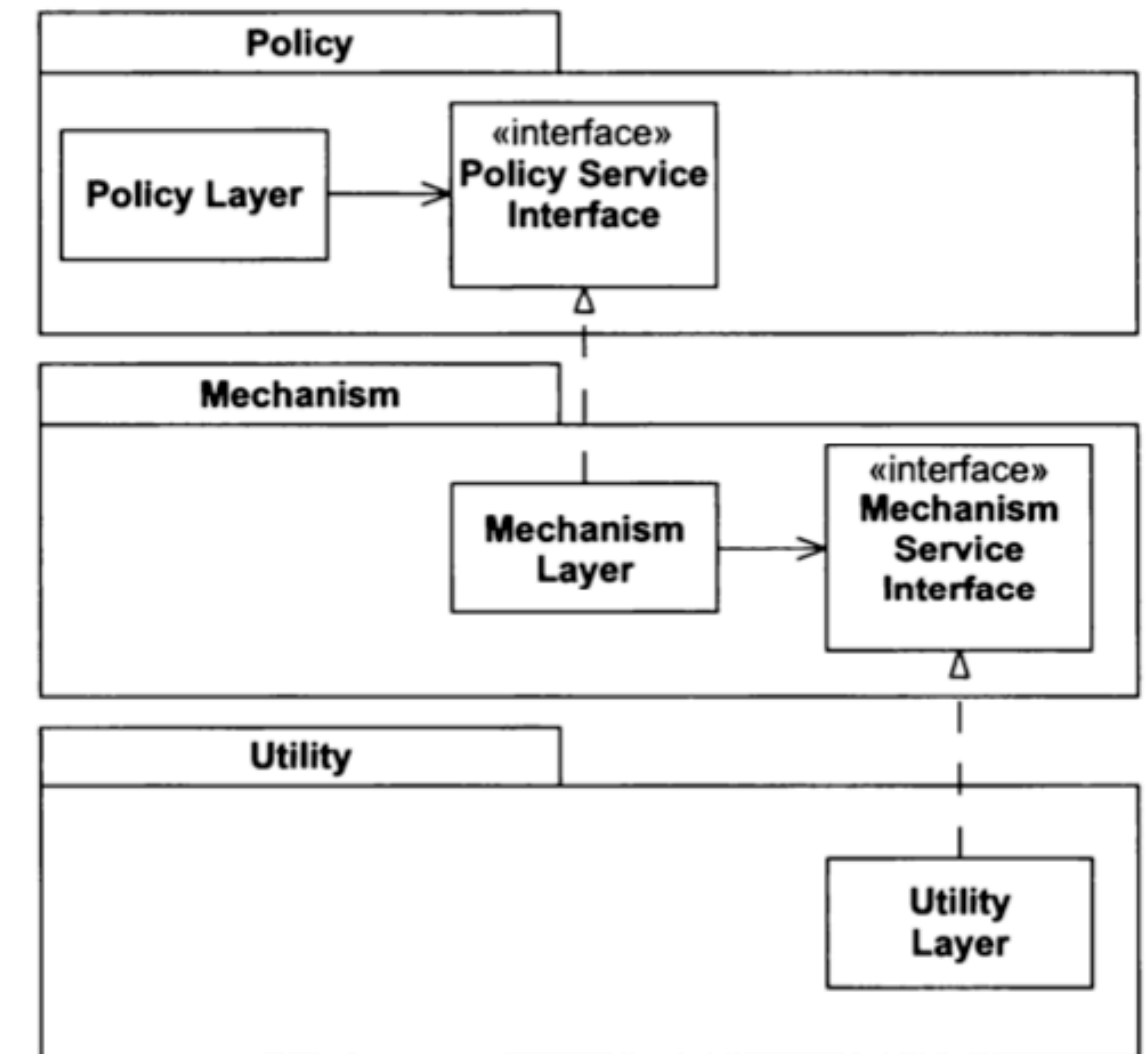



**Figure 11-2** Inverted Layers

# Dependency Inversion Principle

**Figure 11-3** Naive Model of a Button and a Lamp

`Button` object senses external environment

- On receiving `Poll` message, it determines whether or not the user has pressed it.

The `Lamp` object affects external environment

- On Receiving "`TurnOn`" message, it illuminates light.
- On Receiving "`TurnOff`", it extinguishes light.

# Dependency Inversion Principle

## Example

**Listing 11-1**

**Button.java**

```
public class Button
{
  private Lamp itsLamp;
  public void poll()
  {
    if (/*some condition*/)
      itsLamp.turnOn();
  }
}
```

**Figure 11-3** Naive Model of a Button and a Lamp

What is bad about this design?

# Dependency Inversion Principle

## Example

**Figure 11-3** Naive Model of a Button and a Lamp

`Button` depends directly on `Lamp` class.

This implies that changes in `Lamp` will affect `Button` class.

We can't reuse the `Button` to control other classes (e.g. `Motor` class)

**This solution violates DIP**

# Dependency Inversion Principle

## Example

`Button` now has an association called "`ButtonServer`".

`ButtonServer` provides abstract methods that `Button` can use to turn Something on or off.

`Lamp` implements `ButtonServer`.

Future devices, e.g. `Motor`, can also implement this abstract interface.

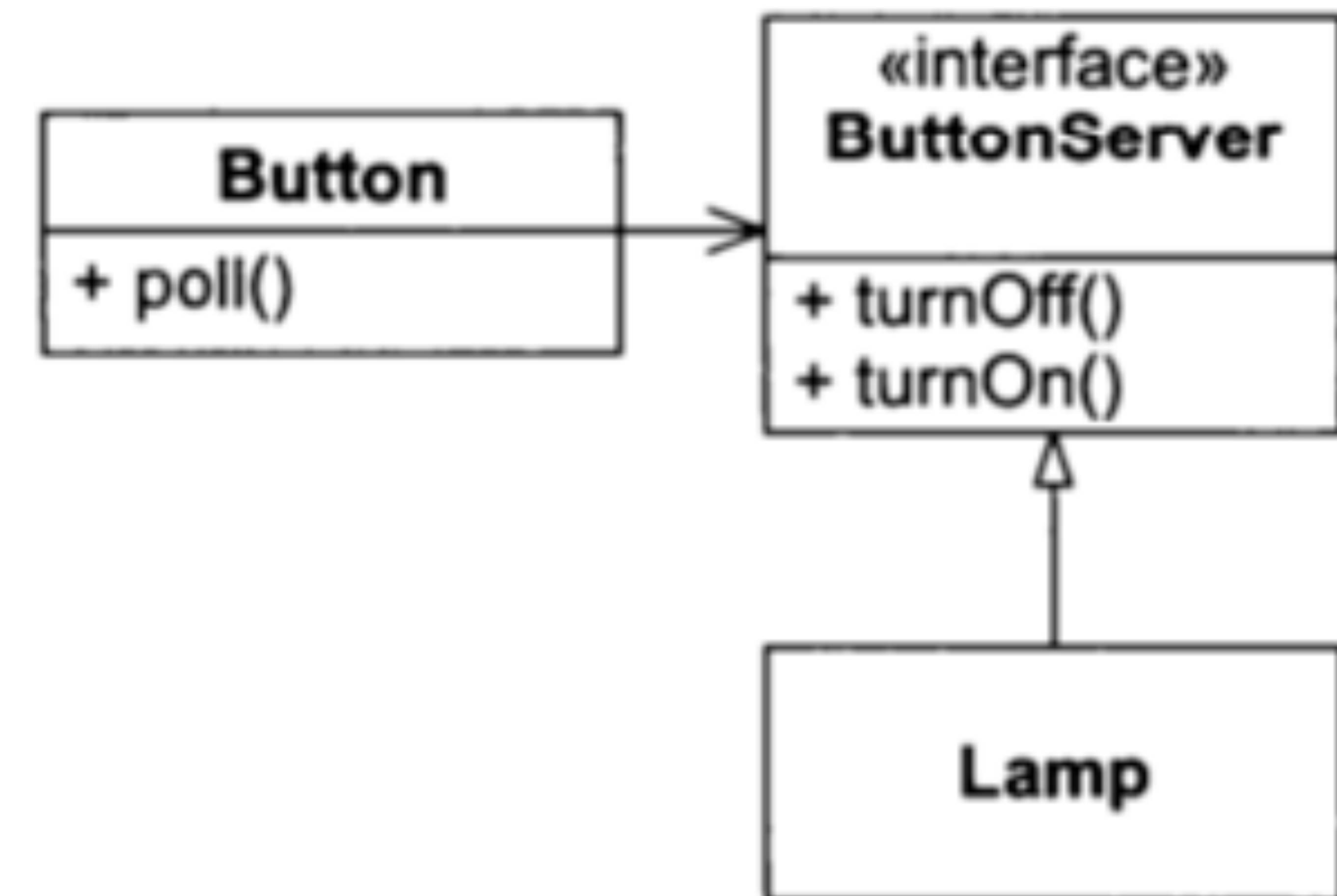



**Figure 11-4** Dependency Inversion Applied to the Lamp

# Dependency Inversion Principle
## Example

**Question:**

Does `Lamp` depend on `Button` ?

**Answer**

- Not really, `Lamp` depends on `ButtonServer` and does not depend on `Button`.

- We can keep `Button` and `ButtonServer` in separate libraries, and possibly rename `ButtonServer` as something else (e.g. `SwitchableDevice`).

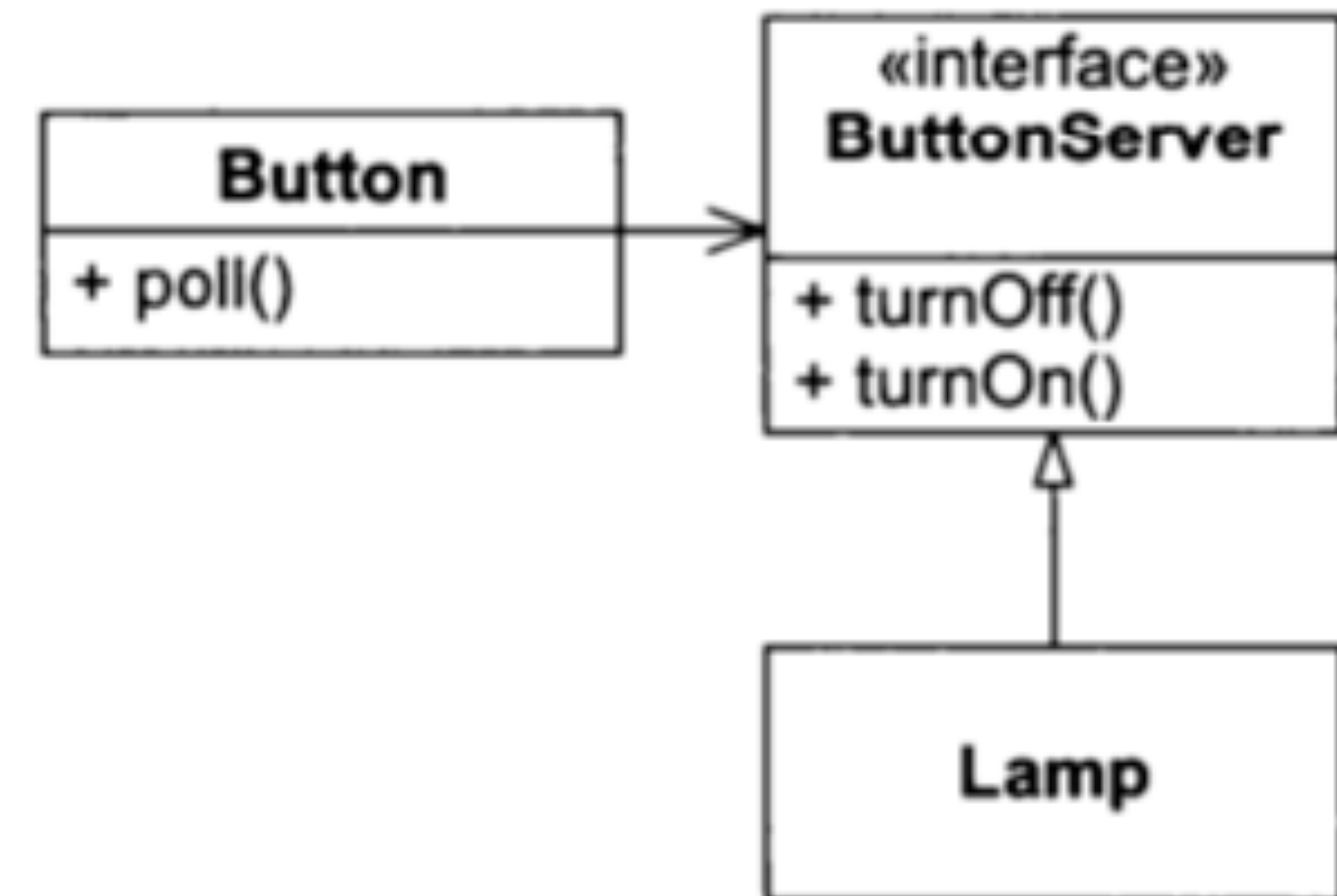



**Figure 11-4** Dependency Inversion Applied to the Lamp

# Dependency Inversion Principle