Design Principles III

Ergude Bao

Beijing Jiaotong University

Contents

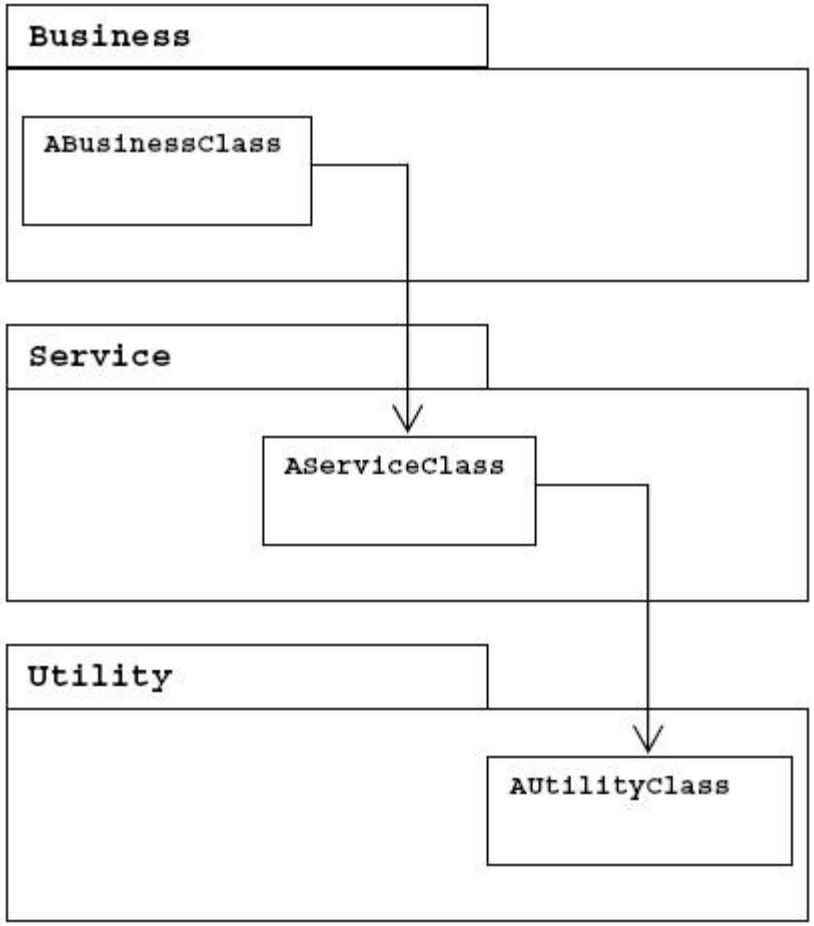
- Dependency-Inversion Principle (DIP)
- Interface-Segregation Principle (ISP)

Dependency-Inversion Principle (DIP)

Dependency-Inversion Principle (DIP)

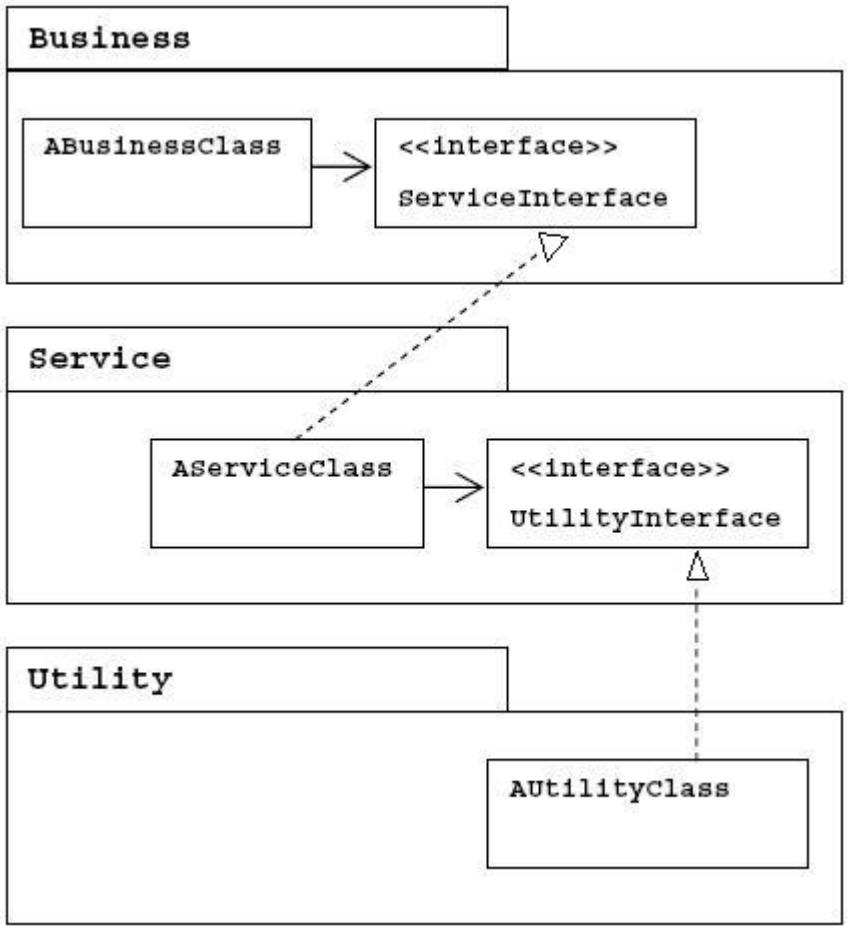
- Direct dependency between two modules should be inverted with an abstraction

Dependency-Inversion Principle (DIP)



- If the business depends on concrete services in the service layer and the services depends on concrete utilities in the utility layer, the business depends transitively on the utilities
  - Changes in the service or utility have effect on the business
  - The business will be difficult to reuse in other contexts

Dependency-Inversion Principle (DIP)



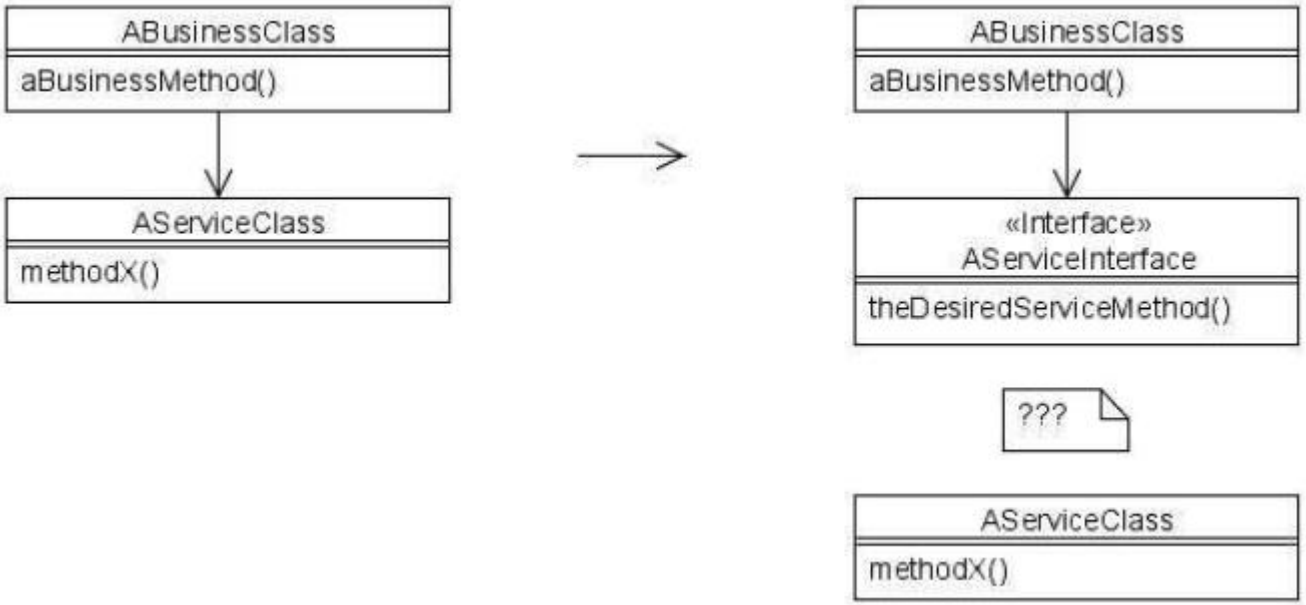
- We should invert the dependencies by using interfaces declared in the upper layer (the client “owns” the interface)
  - Now, the business can be reused with different implementations of the service and utility

Key of DIP

- We should rely on abstractions, such as abstract classes and interfaces
  - A little too strict, since there seems no reason to follow this heuristic for classes that are concrete and not volatile
  - Useful in volatile parts of the system and in parts we want to be loosely coupled, e.g. between layers
- Design for DIP usually meets OCP, and vice versa

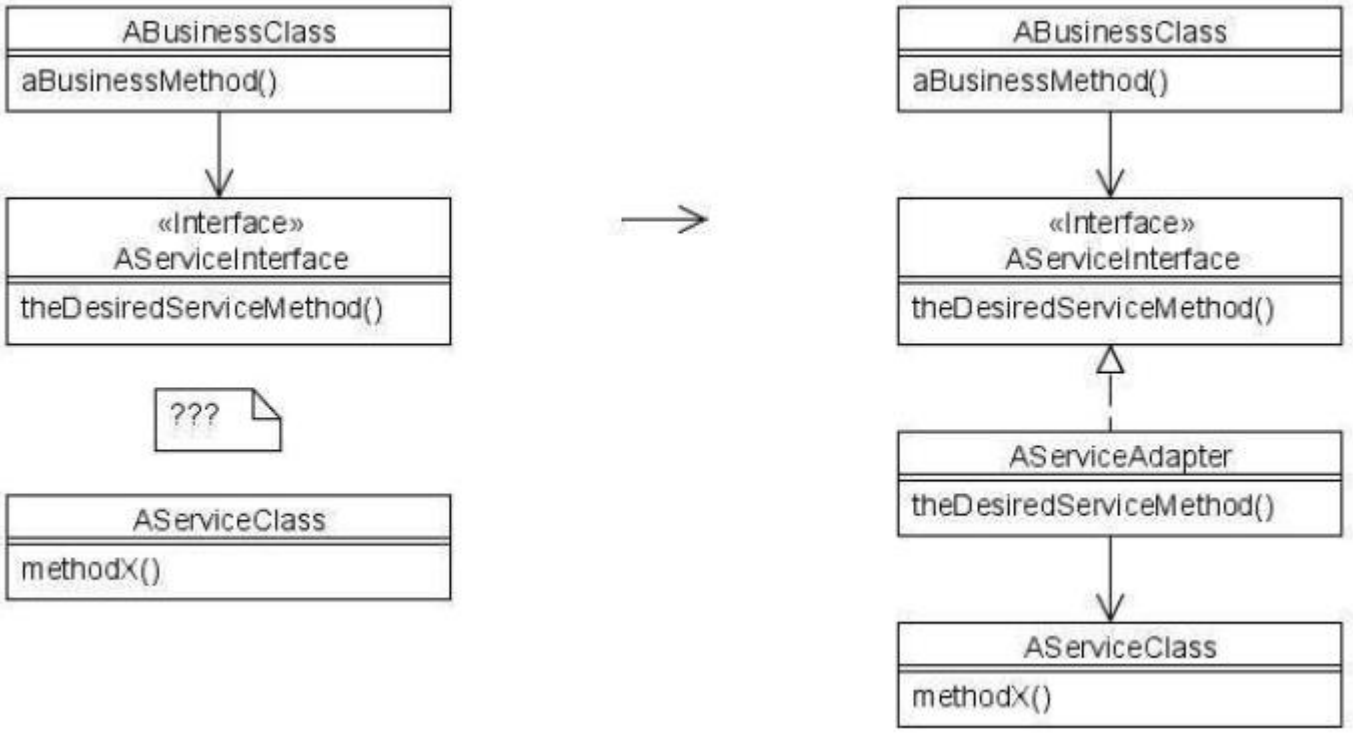
Problem

- What if AServiceClass already exists and do not conform to the desired ServiceInterface?



Solution

- Adapter design pattern



Example I

```
// Bad example
class Worker {
        public void work() {  // working  }
}

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

class Manager {
        Worker m_worker;
        public void setWorker(Worker w) {  m_worker=w;  }
        public void manage() {   m_worker.work();   }
}
```

Example I

```
// 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 m_worker;
        public void setWorker(IWorker w) {  m_worker=w;  }
        public void manage() {  m_worker.work();  }
}
```

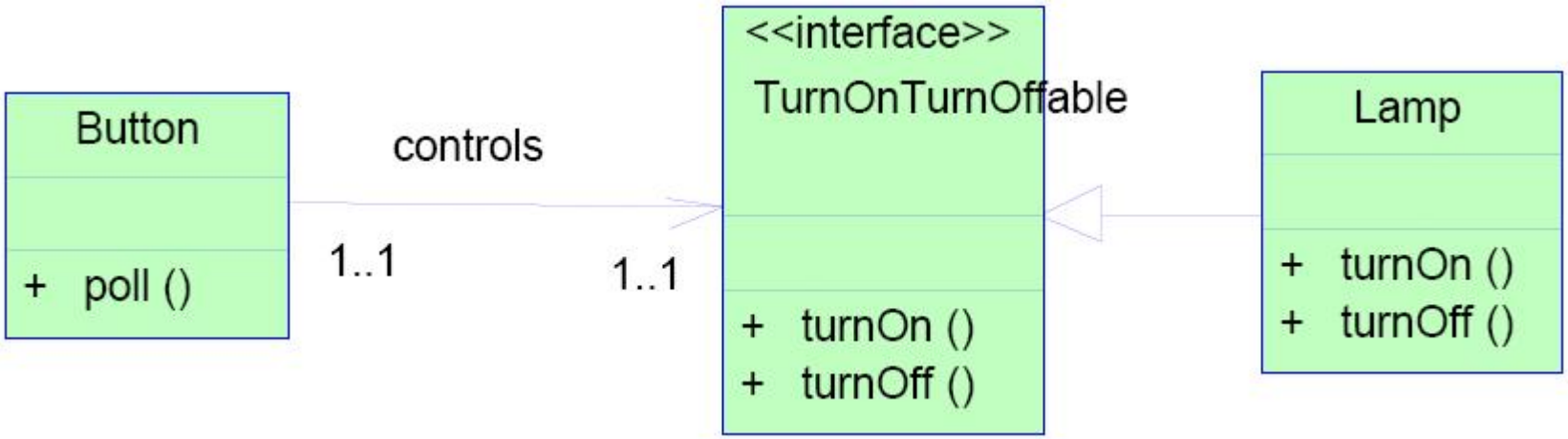
Example II


```
Public class Button  {
   private Lamp itsLamp;

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

Example II

- Violation of DIP
  - Button depends on Lamp
  - Button cannot be reused in other contexts
- Solution
  - Make a general interface for turning on and off things
  - Make button depend on this interface
  - Different appliances can implement this interface and be controlled by an on/off button

Example II


Button
+ poll ()
controls
1..1
1..1
<<interface>>
TurnOnTurnOffable
+ turnOn ()
+ turnOff ()
Lamp
+ turnOn ()
+ turnOff ()

Non-DIP vs. DIP

- High-level depends on low-level
- Layers not separated by interfaces
- Changes in low-level may affect high-level
- Low-level owns the interface and high level adapts

- Both low-level and high-level depend on abstractions
- Changes in low-level usually do not affect high-level
- High-level owns the interface and low-level adapts

Questions

- Please give an example that violates DIP and explain why? How to modify it to comply with DIP?
- How to handle changes in interface?

Interface-Segregation Principle (ISP)

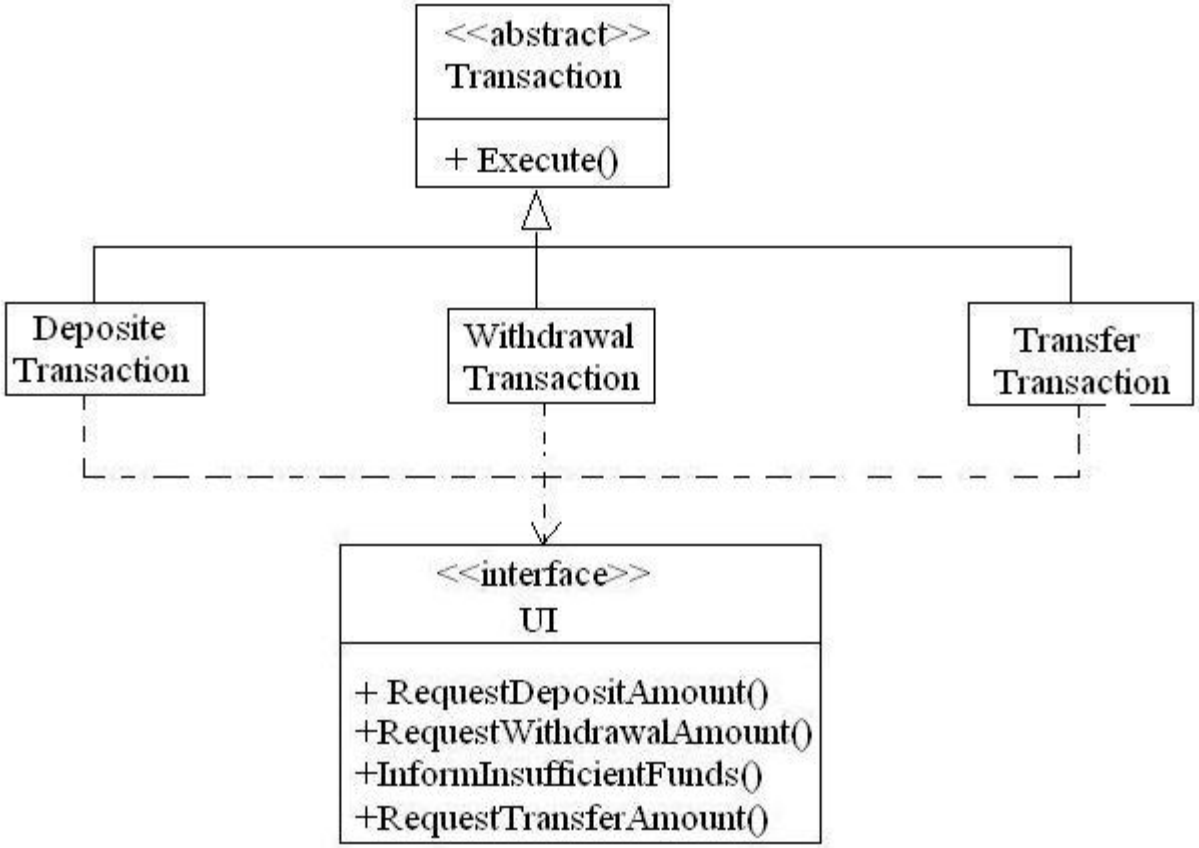
Interface-Segregation Principle (ISP)

- Class should not be forced to implement methods that it does not use
  - Avoid class whose interface is not cohesive with too many responsibilities

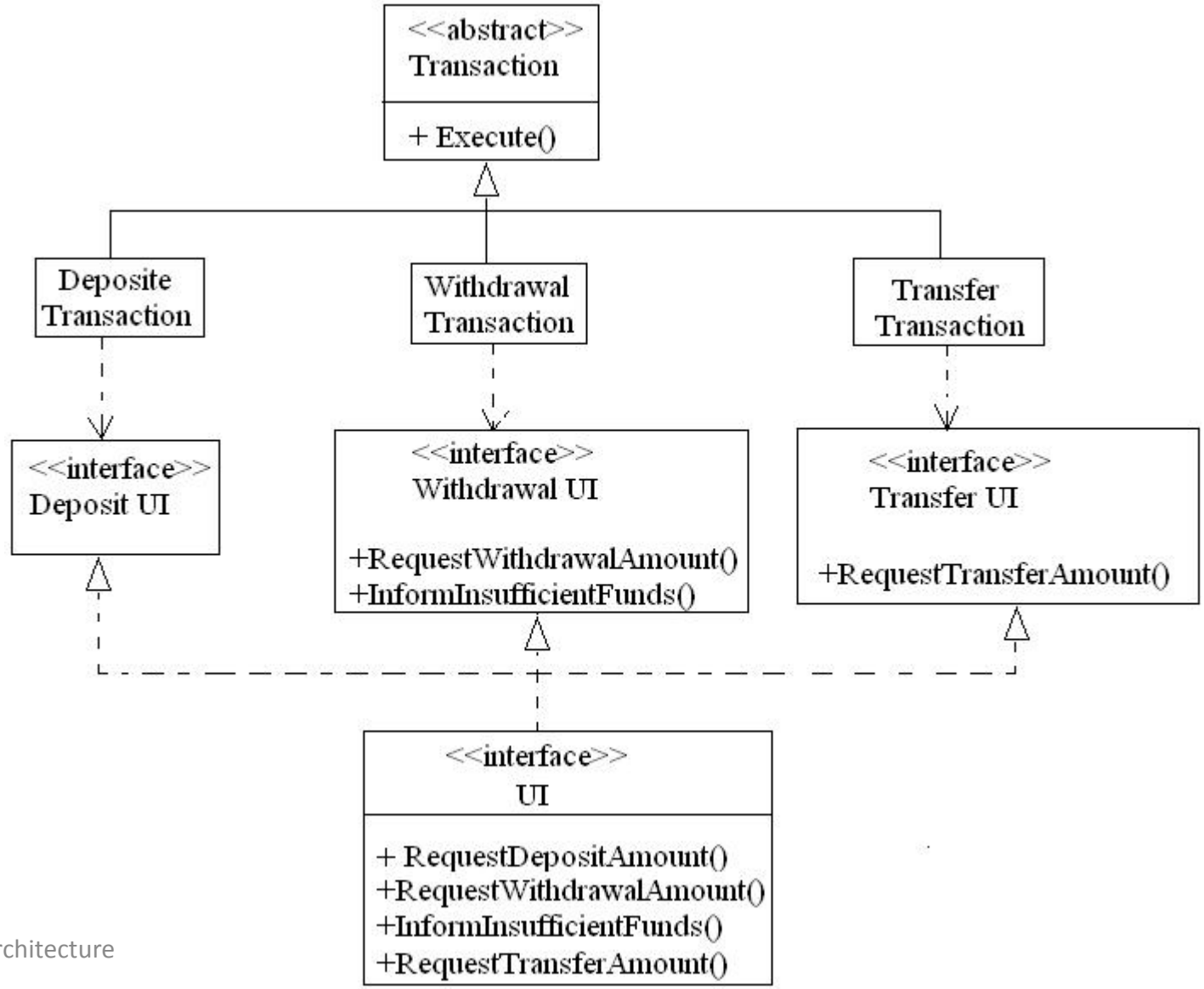
Key of ISP

- We should break interfaces up into cohesive groups of methods, each serving a certain kind of class
- Design for ISP usually meets SRP

Examples I


<<abstract>>
Transaction
+ Execute()
Deposite Transaction
Withdrawal Transaction
Transfer Transaction
<<interface>>
UI
+ RequestDepositAmount()
+RequestWithdrawalAmount()
+InformInsufficientFunds()
+RequestTransferAmount()

Examples I


<<abstract>>
Transaction
+ Execute()
Deposite Transaction
Withdrawal Transaction
Transfer Transaction
<<interface>>
Deposit UI
<<interface>>
Withdrawal UI
+RequestWithdrawalAmount()
+InformInsufficientFunds()
<<interface>>
Transfer UI
+RequestTransferAmount()
<<interface>>
UI
+ RequestDepositAmount()
+RequestWithdrawalAmount()
+InformInsufficientFunds()
+RequestTransferAmount()

Example II

```
// Bad example
interface IWorker {
      public void work();
      public void eat();
}

class Worker implements IWorker{
      public void work() { // working  }
      public void eat() { // eating in launch break }
}
```

Example II

```
class SuperWorker implements IWorker{
        public void work() {  // working much more  }
        public void eat() {  // eating in launch break  }
}

class Robot implements IWorker{
        public void work() {  // working much more  }
        public void eat() {  // empty }
}

class Manager {
        IWorker worker;
        public void setWorker(IWorker w) {  worker=w;  }
        public void manage() {  worker.work();  }
}
```

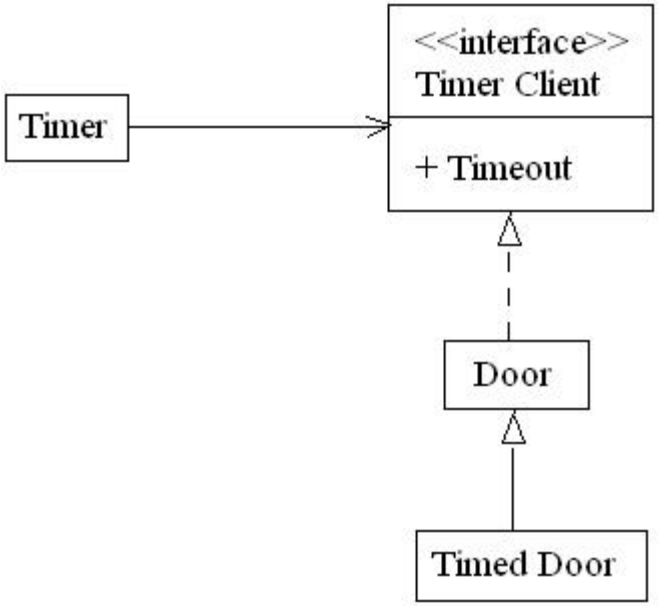
Example II

```
// Good example
interface IWorker extends Feedable, Workable { }

interface IWorkable {  public void work();  }

interface IFeedable{  public void eat();  }

class Worker implements IWorkable, IFeedable{
      public void work() {  // working  }
      public void eat() {  // eating in launch break  }
}
```

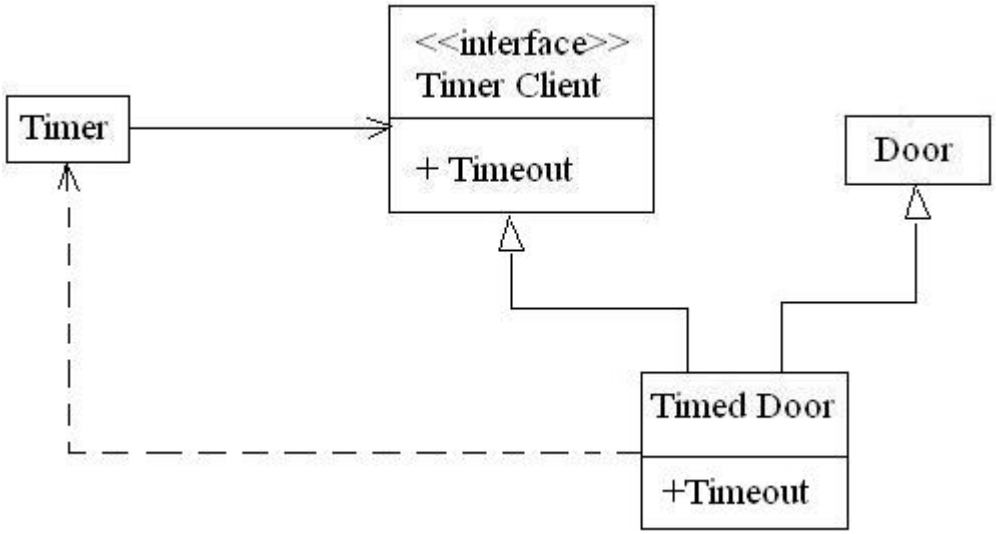
Example II

```
class SuperWorker implements IWorkable, IFeedable{
        public void work() {  // working much more  }
        public void eat() {  // eating in launch break  }
}

class Robot implements IWorkable{
        public void work() {  // working  }
}

class Manager {
        IWorkable worker;
        public void setWorker(IWorkable w) {  worker=w;  }
        public void manage() {  worker.work();  }
}
```

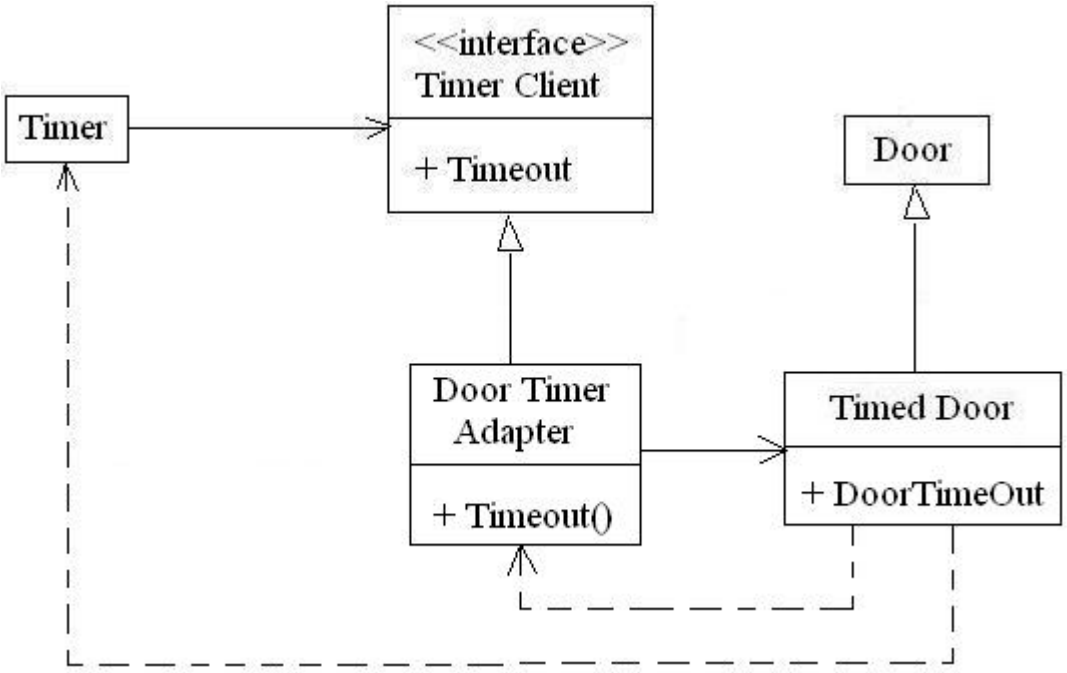
Example III



Example III

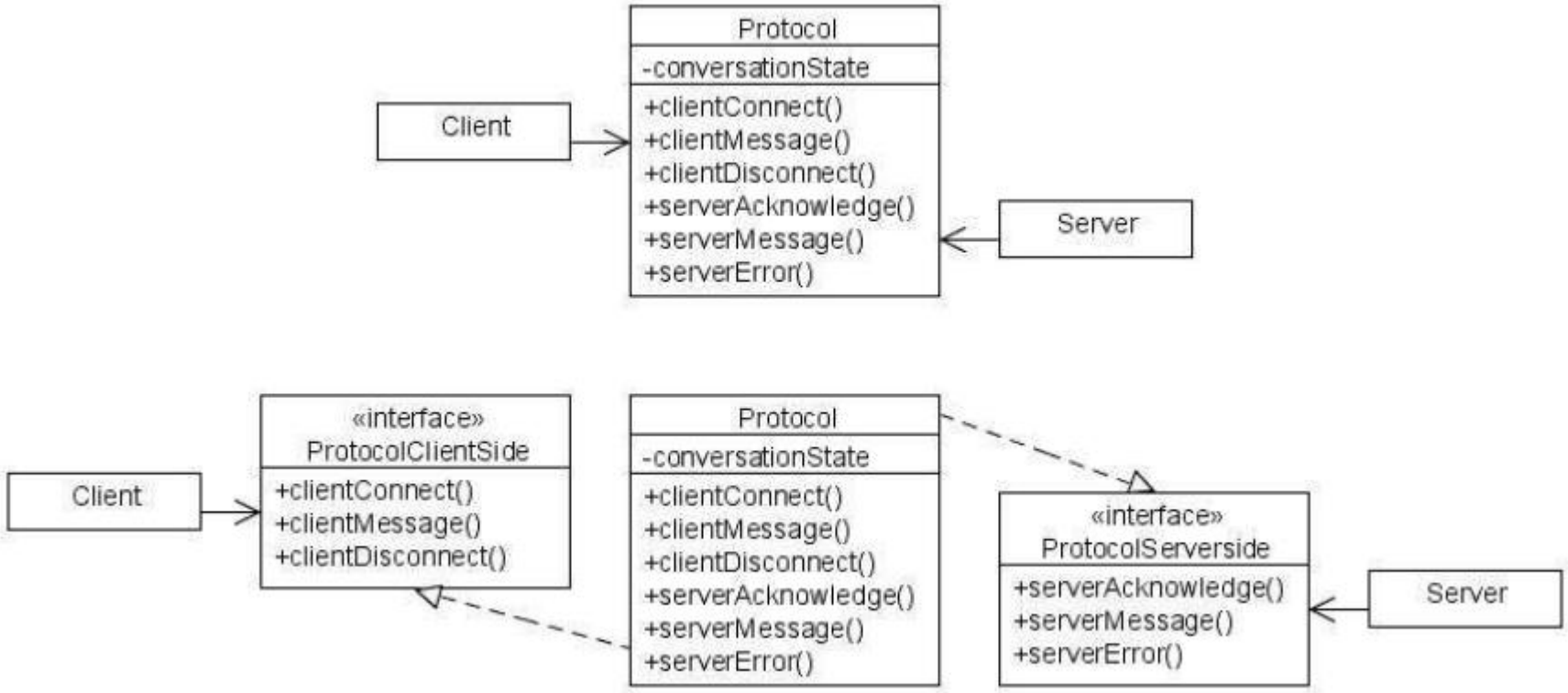

<<interface>>
Timer Client
+ Timeout
Timer
Door
Timed Door
+Timeout

Example III


<<interface>>
Timer Client
+ Timeout
Timer
Door
Door Timer
Adapter
+ Timeout()
Timed Door
+ DoorTimeOut

Example IV

- An object representing an application-level protocol for communication between a client and a server



Summary

- Fat interface causes coupling among its classes
  - When one class forces a change on the fat interface, all the other classes are affected
- The fat interface should be broken into many class-specific interfaces
  - This breaks the dependence of the classes on methods that they do not invoke, and it allows the classes to be independent on each other

Questions

- Please give an example that violates ISP and explain why? How to modify it to comply with ISP?