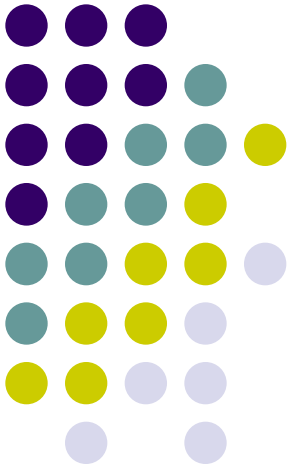
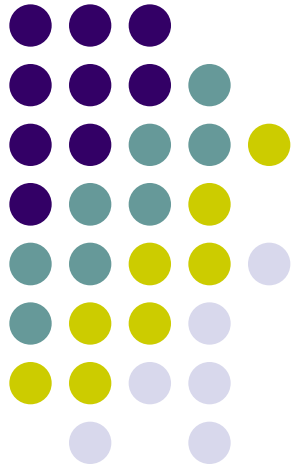
Software Architecture

Design Principles – DIP and ISP



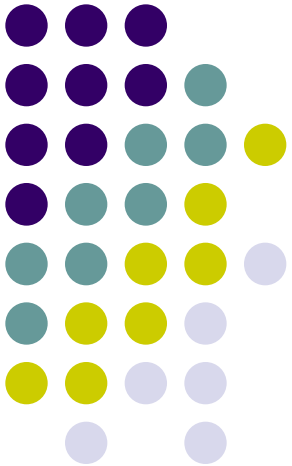
Design Principles: DIP and ISP

- DIP --The Dependency-Inversion Principle
- ISP --The Interface-Segregation Principle

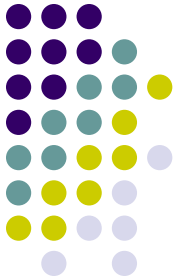


DIP:

The Dependency-Inversion Principle

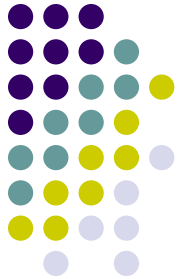


DIP - The Dependency-Inversion Principle



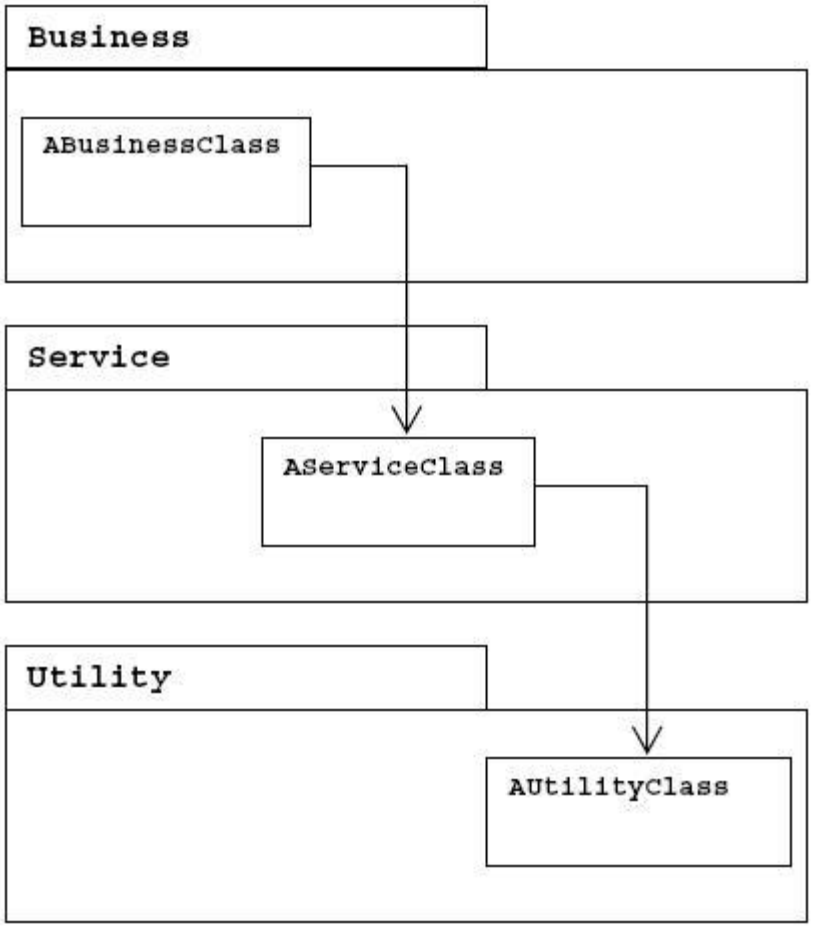
- The Dependency-Inversion Principle:
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.

DIP - The Dependency-Inversion Principle



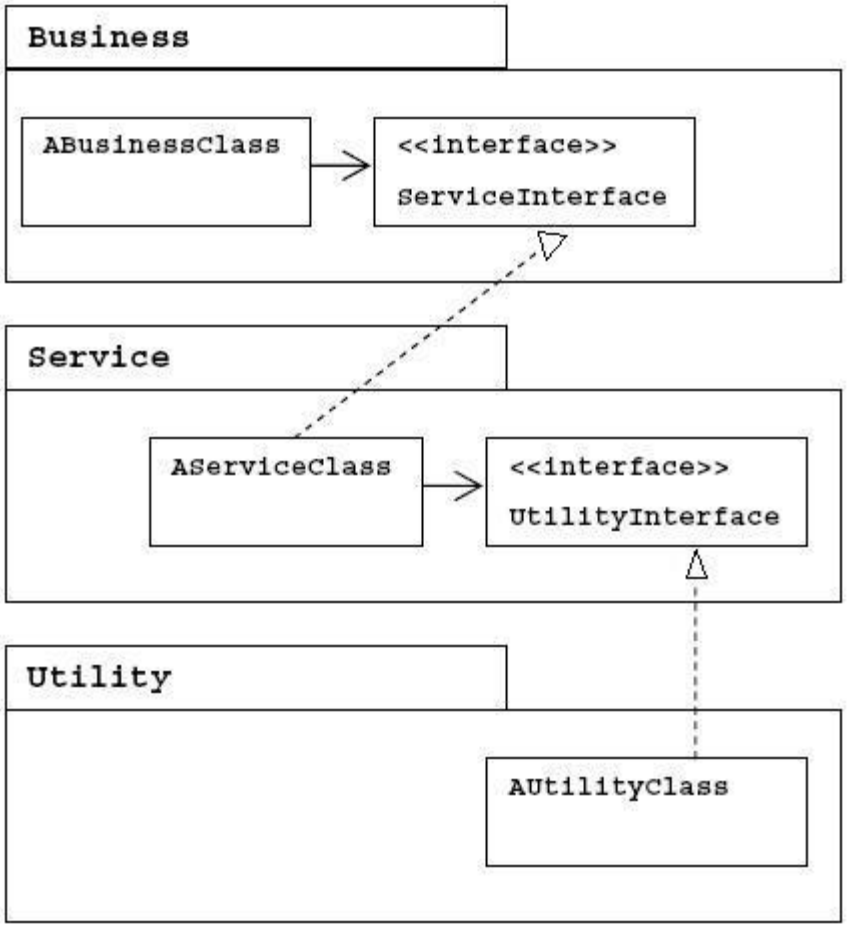
- Structured Analysis and Design tend to create software structures in which high-level modules depend on low-level modules, and in which policy depends on detail.
- It is the high-level modules that contain the important policy decisions and business models of an application.

DIP – Violating 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.
- This is very unfortunate because changes in low level modules have effect on high-level modules.
- High-level modules will be difficult to reuse in other contexts

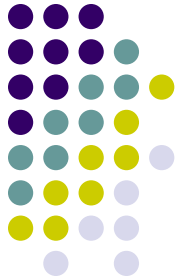
DIP – Conforming to DIP



- You should invert the dependencies by using interfaces declared in the upper layer (the client “owns” the interface).
- Now, the business no longer depends on a concrete service and can be reused with different implementations of the service.
- *NOTE: The book uses a different naming convention for the interfaces.*

Example

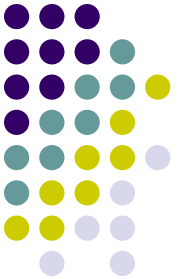
```
// Dependency Inversion Principle - Bad example
class Worker {
    public void work() {  // ....working  }
}
class Manager {
    Worker m_worker;
    public void setWorker(Worker w) {  m_worker=w;  }
    public void manage() {   m_worker.work();   }
}
class SuperWorker {
    public void work() {  //.... working much more   }
}
```



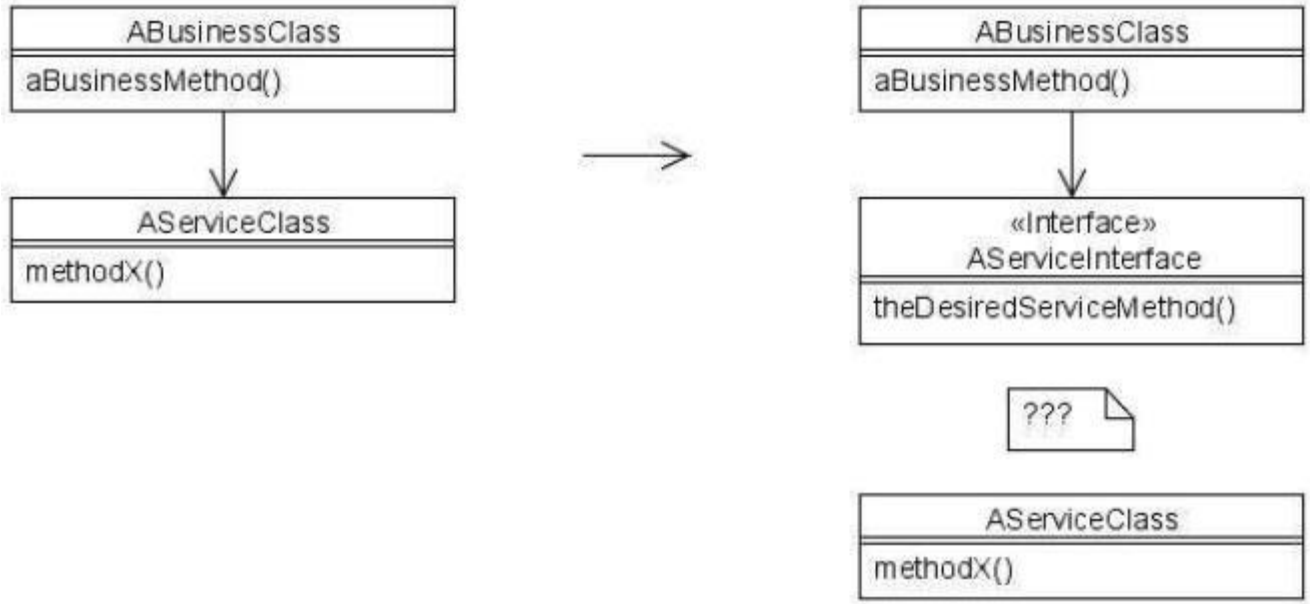
Example

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

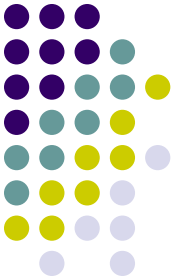
DIP – Adapting to existing classes



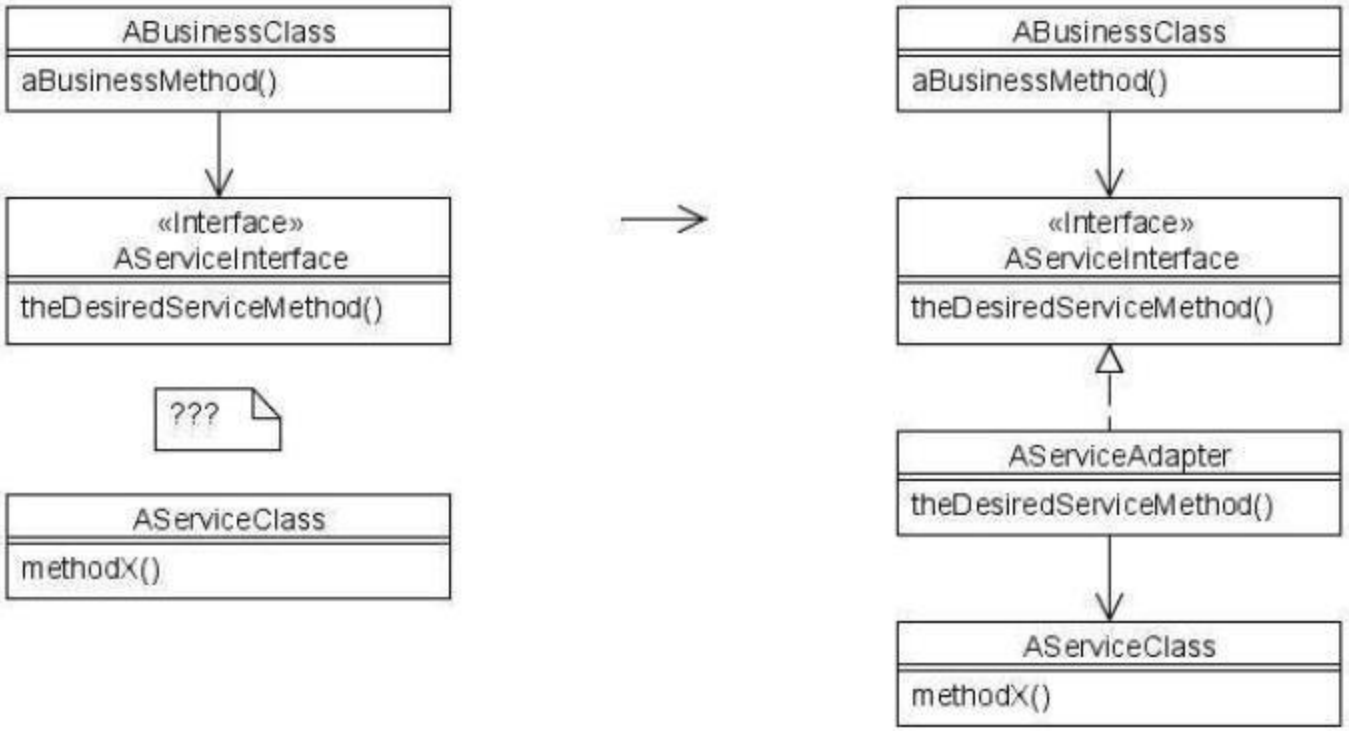
- **The problem:** what if AServiceClass already exists and do not conform to the desired ServiceInterface?



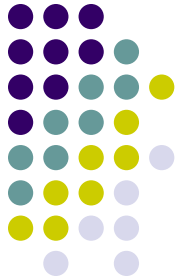
DIP – Adapting to existing classes



- **The solution (the Adapter Pattern):**

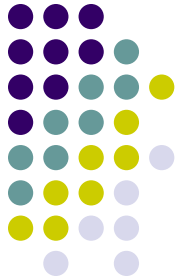


DIP – Depend on abstractions



- A “naive”, but useful interpretation of DIP:
  - No variable should hold a reference to a concrete class
  - No class should derive from a concrete class
  - No method should override an implemented method of any of its base classes

DIP – Depend on abstractions

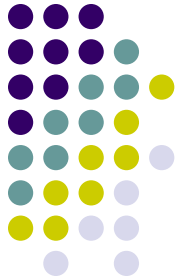


- The idea is that we should only rely on abstractions, that is abstract classes and interfaces.
- This heuristic is too strict, there seems no reason to follow this heuristic for classes that are concrete but nonvolatile.
- It can be used in volatile parts of the system and in parts you want to be loosely coupled, e.g. between layers.

DIP – Ex: Button - Lamp (1)

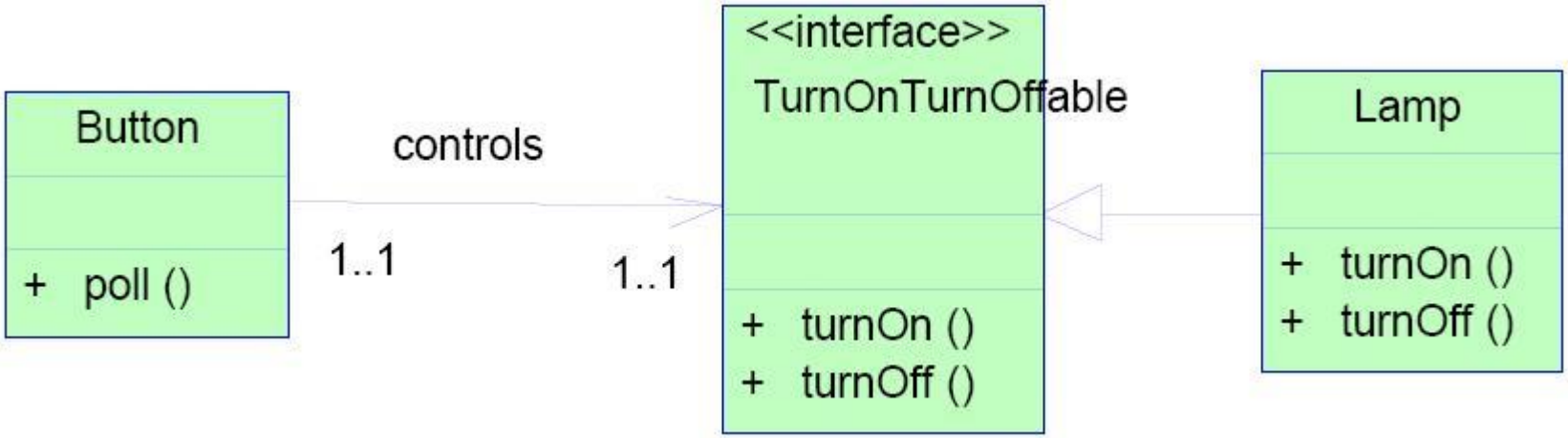

```
Public class Button  {
   private Lamp itsLamp;

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

DIP – Ex: Button – Lamp (2)

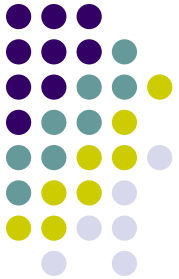


- Violation of DIP
  - Button depends on Lamp
  - Button cannot be reused in other contexts
- Solution
  - Make a general interface for turning on and off things.
  - Make button depend on this interface. Different appliances can implement this interface and be controlled by an on/off button.
  - … or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

DIP – Ex: Button - Lamp (1)

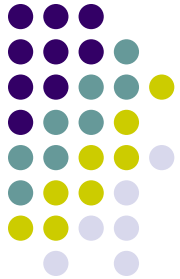


DIP - Summary



- Traditionally:
  - High-level depends on low-level
  - Layers not separated by interfaces
  - Changes in low-level may affect high-level
  - Low-level owns the interface – high level adapts
  - ”Procedural Design”
- DIP:
  - Both low-level and high-level depend on abstractions
  - Changes in low-level usually don’t affect high-level
  - High-level owns the interface – low-level adapts
  - ”Object Oriented Design”

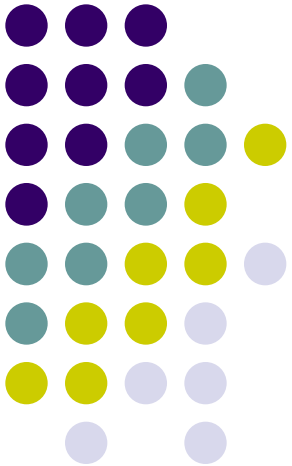
Exercise



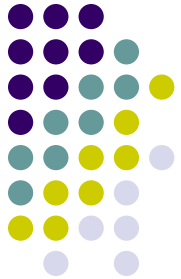
- Please give an example that violates DIP and explain why? How to modify it to comply with DIP?

ISP :

Interface-Segregation Principle



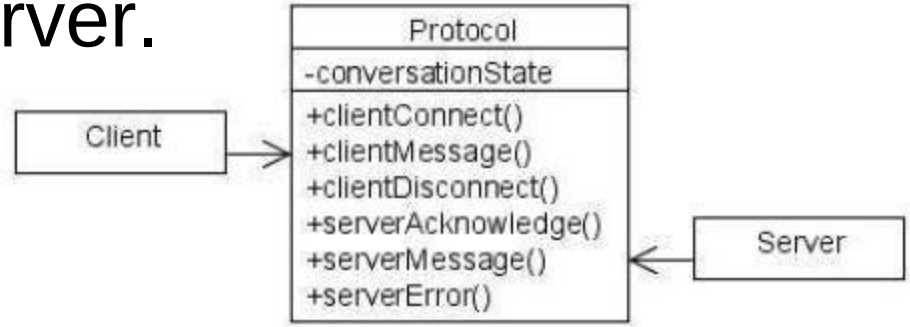
ISP - Interface-Segregation Principle

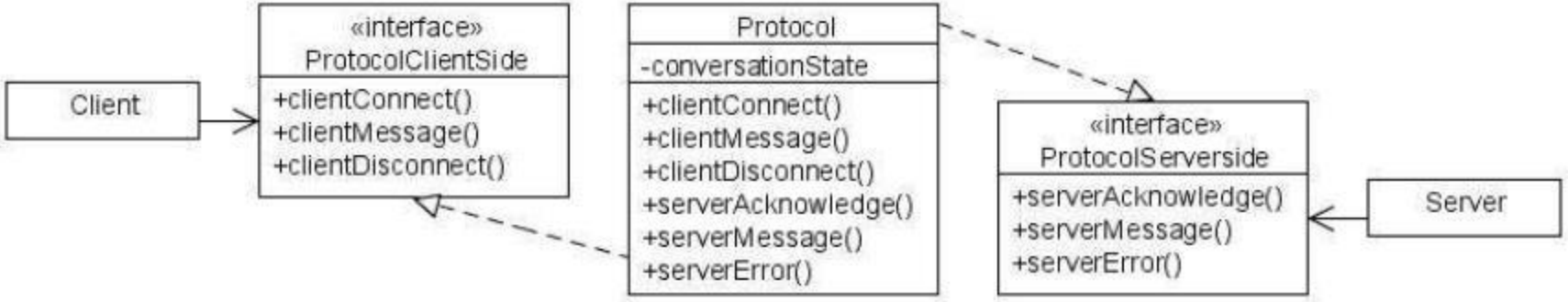


- ISP deals with the disadvantages of “fat” interfaces.
- Interface-Segregation Principle:
  - Clients should not be forced to depend on methods that they do not use.
- Avoid classes with too many responsibilities (classes whose interfaces are not cohesive).
- Break interfaces up into cohesive groups of methods, each serving a certain kind of clients.

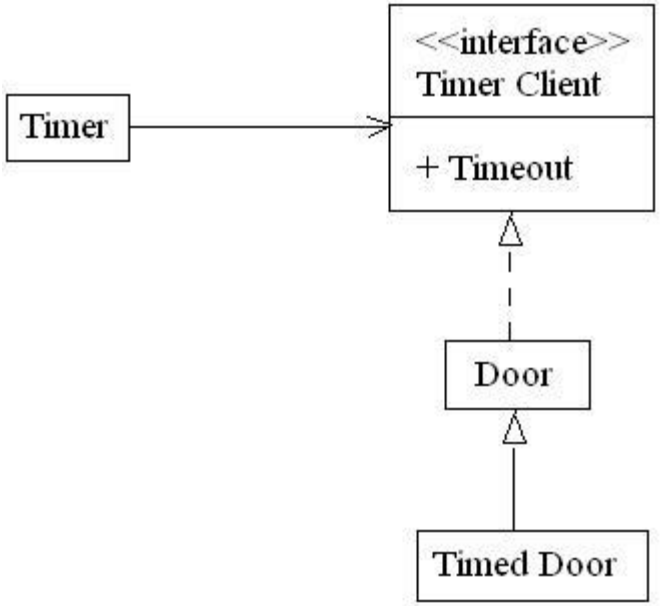
ISP – Example 1

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

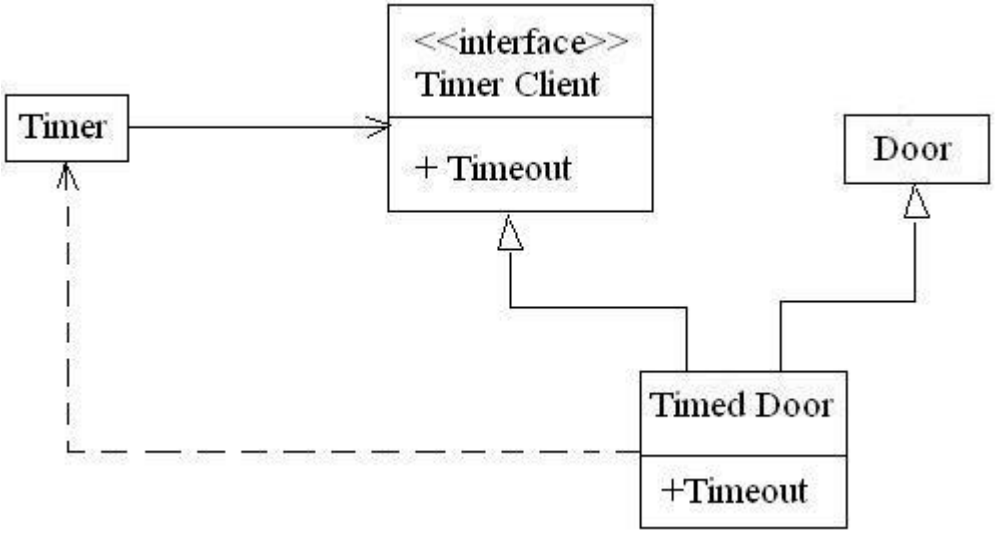




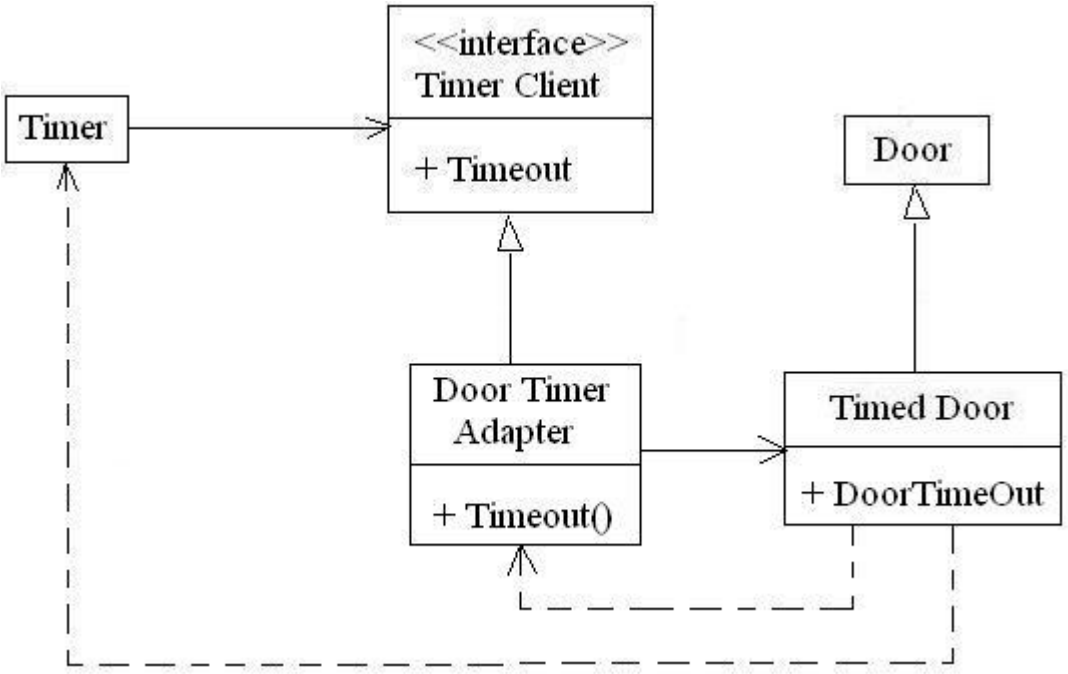
Examples II



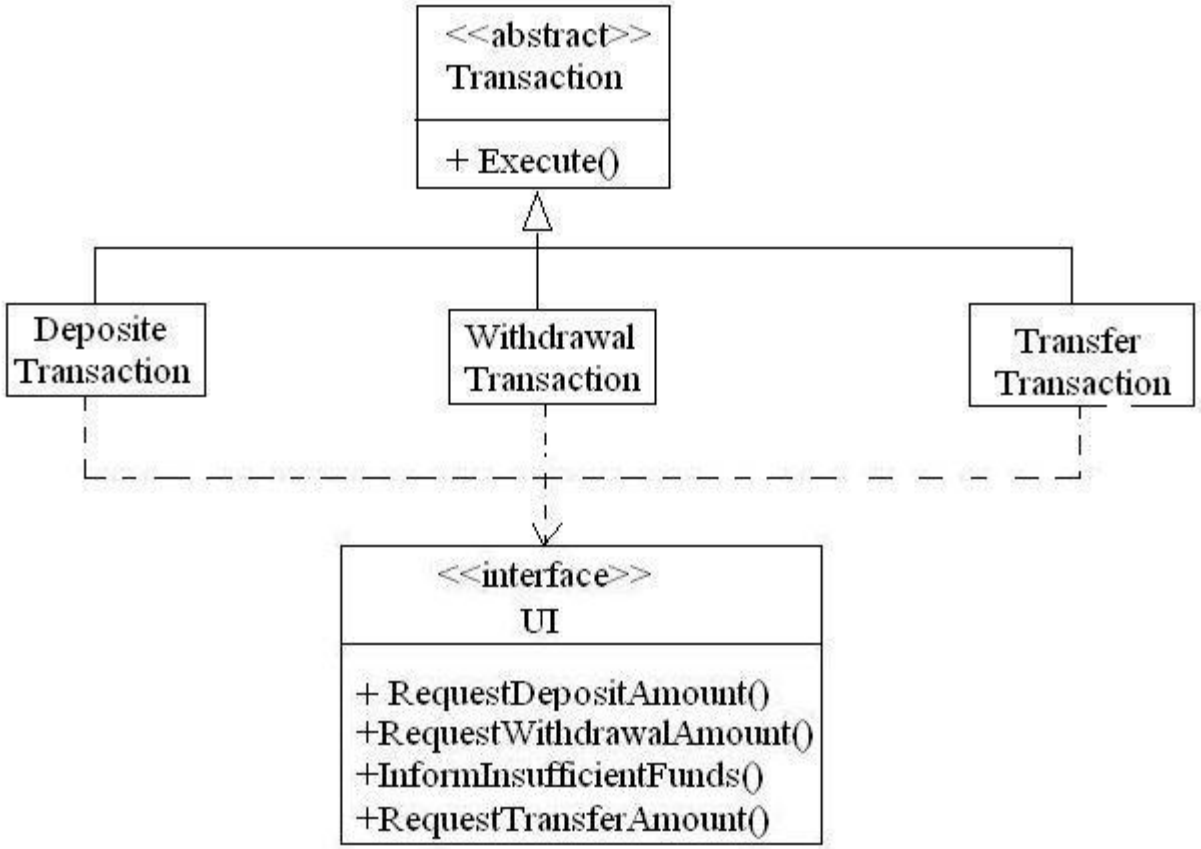
Examples II


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

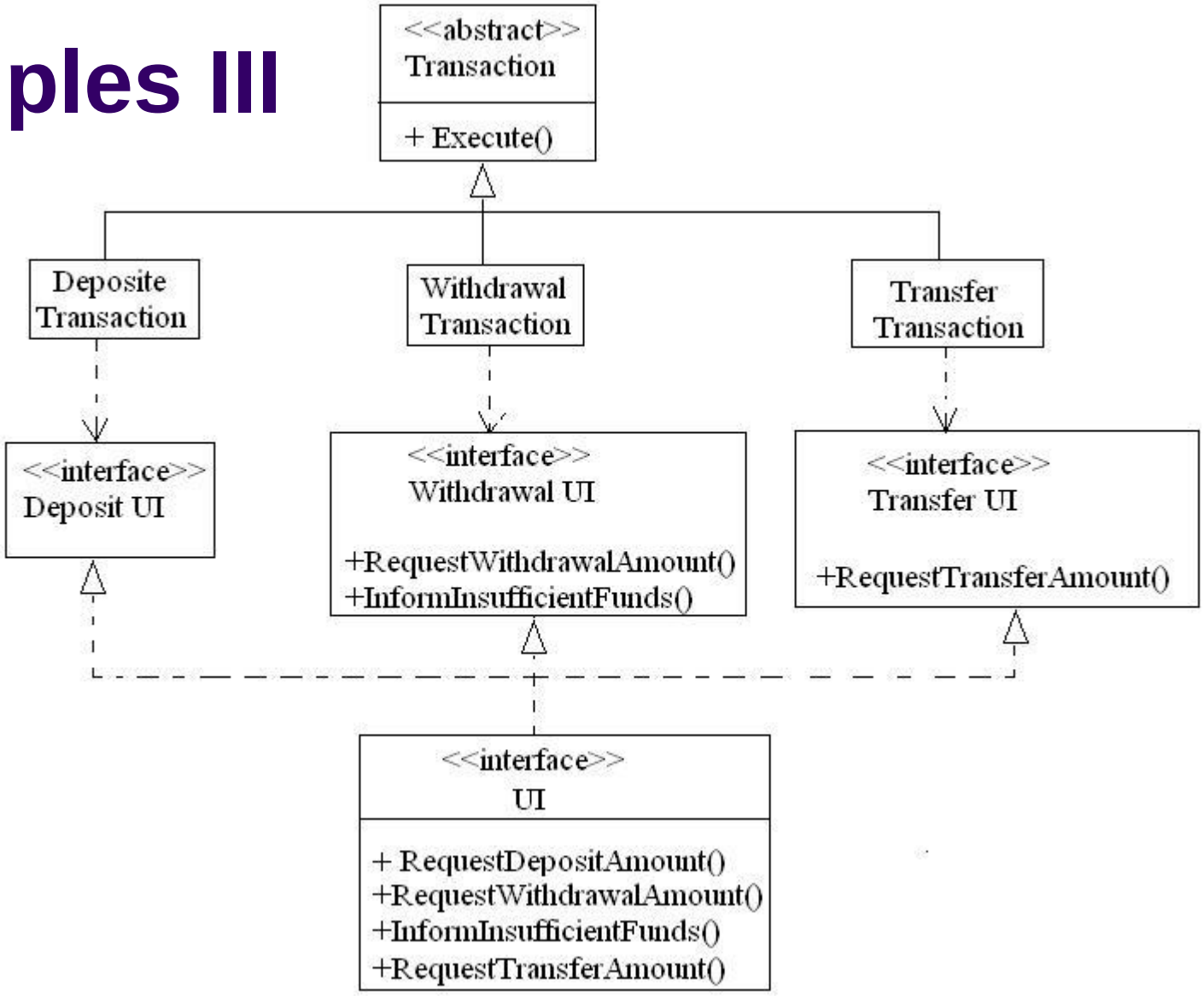
Examples II


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

Examples III



Examples III



Example IV

```
// interface segregation principle - 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 IV

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

class Manager {
    IWorker worker;

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

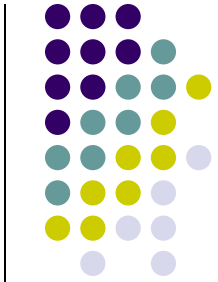
Example IV

```
// interface segregation principle - 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 IV

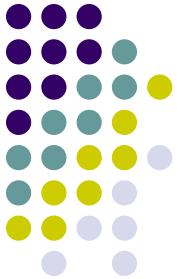
```
class Robot implements IWorkable{
    public void work() {     // ....working    }
}
class SuperWorker implements IWorkable, IFeedable{
    public void work() {    //.... working much more   }

    public void eat() {     //.... eating in launch break    }
}
class Manager {
    Workable worker;

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

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

Conclusion



- Fat classes cause coupling between their clients.
  - When one client forces a change on the fat class, all the other clients are affected.
- The interface of the fat class should be broken into many client-specific interfaces.
  - This breaks the dependence of the clients on methods that they don’t invoke, and it allows the clients to be independent of each other.

Exercise



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