Object Oriented Analysis & Design
面向对象分析与设计

Lecture_08 通用的职责分配软件原则 GRASP（二）

主讲: 姜宁康 博士

7、其他面向对象设计原则3: 依赖倒置原则DIP

- The Dependency Inversion Principle

7.1 依赖倒置原则DIP

The Dependency Inversion Principle

I. 高层模块不应当依赖低层模块，两者都依赖抽象 High-level modules should not depend on low-level modules，Both should depend on abstractions

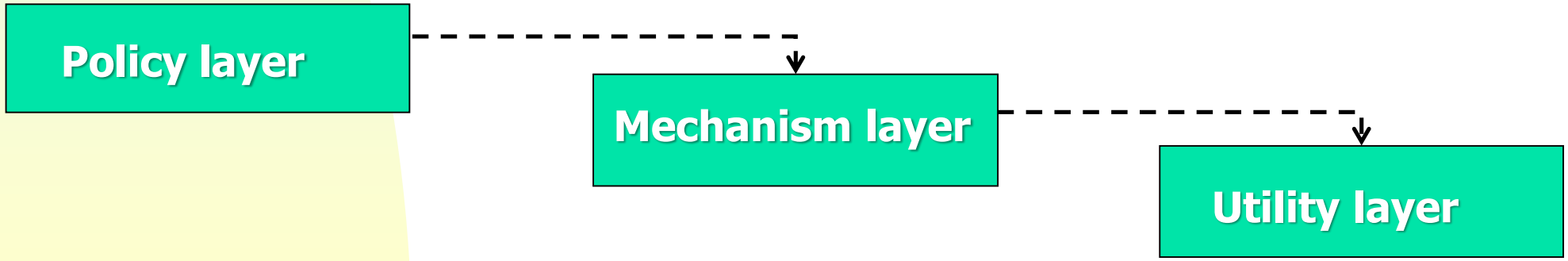
II.抽象不能依赖细节，细节应当依赖抽象 Abstractions should not depend on details，Details should depend on abstractions

R. Martin, 1996

- **引导**
  - **继承层次关系中，基类不应当知道任何子类** A base class in an inheritance hierarchy should not know any of its subclasses
  - **不能依赖一个有详细实现的模块，而这个模块本身也应当依赖抽象** Modules with detailed implementations are not depended upon, but depend themselves upon abstractions
- **OCP宣扬了目标，DIP宣扬了机制** OCP states the goal; DIP states the mechanism

7.2 为什么依赖倒置原则DIP？

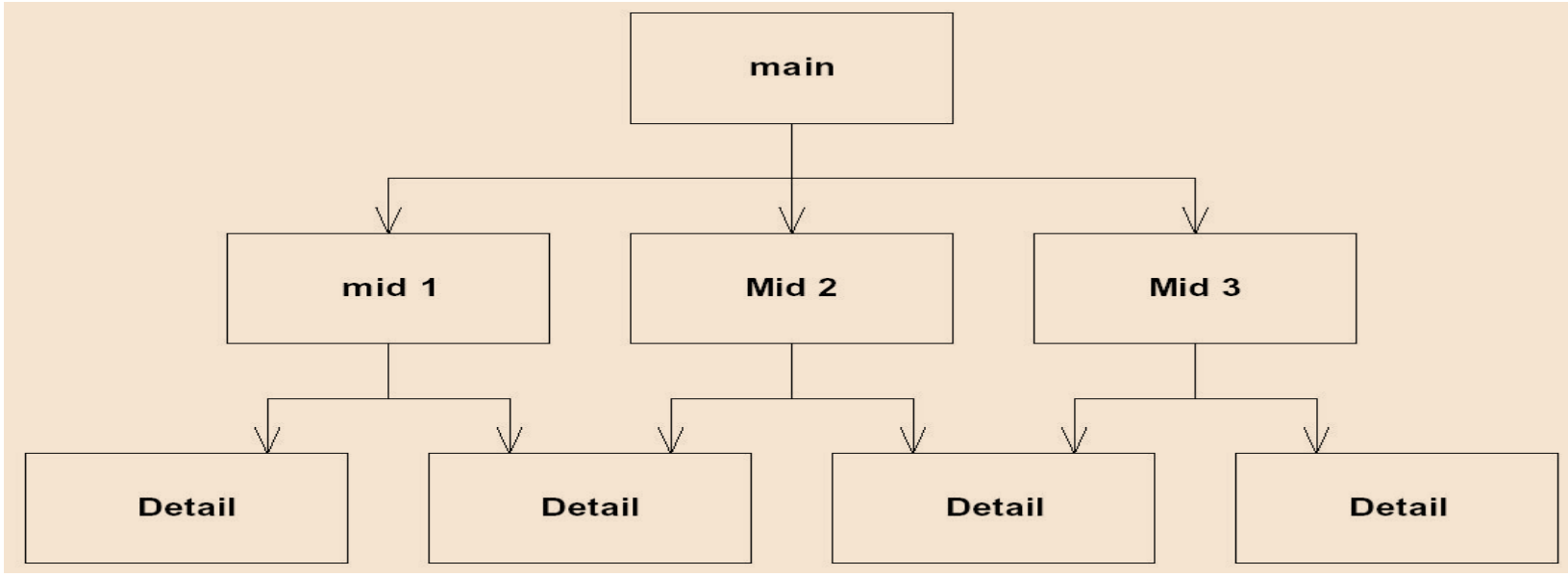
- 传统的面向过程的程序设计，以功能划分系统
  - 高层模块是业务/应用规则 High level modules: business/application rules
  - 低层模块是对这些规则的实现 Low level modules: implementation of the business rules
  - 高层模块完全依赖调用低层模块提供的功能来完成自己的功能 High level modules complete their functionality by calling/invoking the low level implementation provided by the low level modules
- 因此，高层依赖底层 High level depends on the lower level



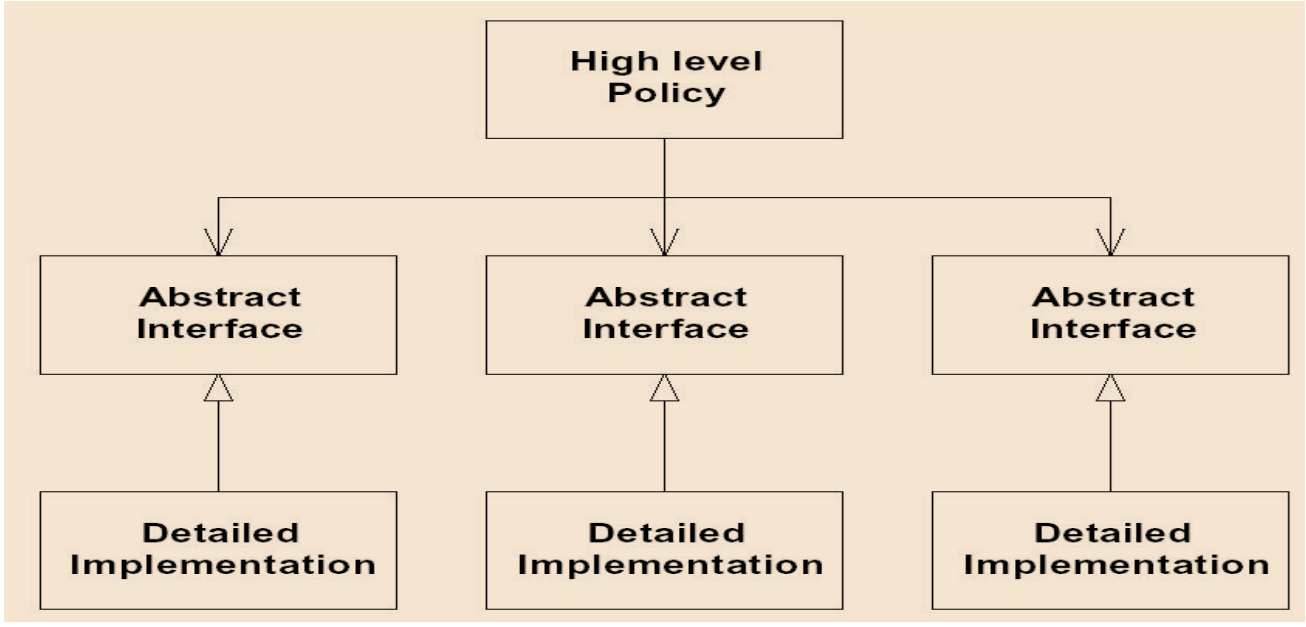
7.2 为什么依赖倒置原则DIP？

比较：过程化程序与面向对象架构

过程化程序架构



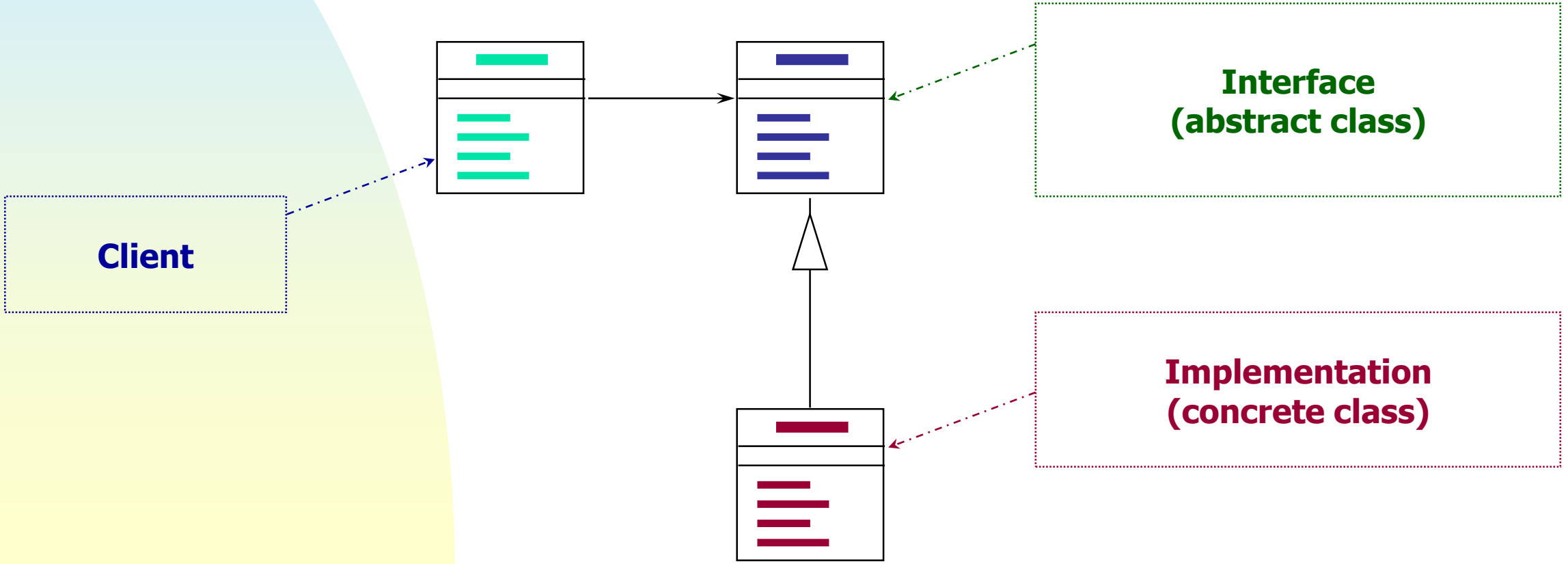
面向对象系统架构
Object-Oriented Architecture



7.3 依赖倒置原则的启发1

面向接口设计，而不是面向实现设计 Design to an interface, not an implementation!

- **使用继承，避免类之间的直接绑定 Use inheritance to avoid direct bindings to classes:**



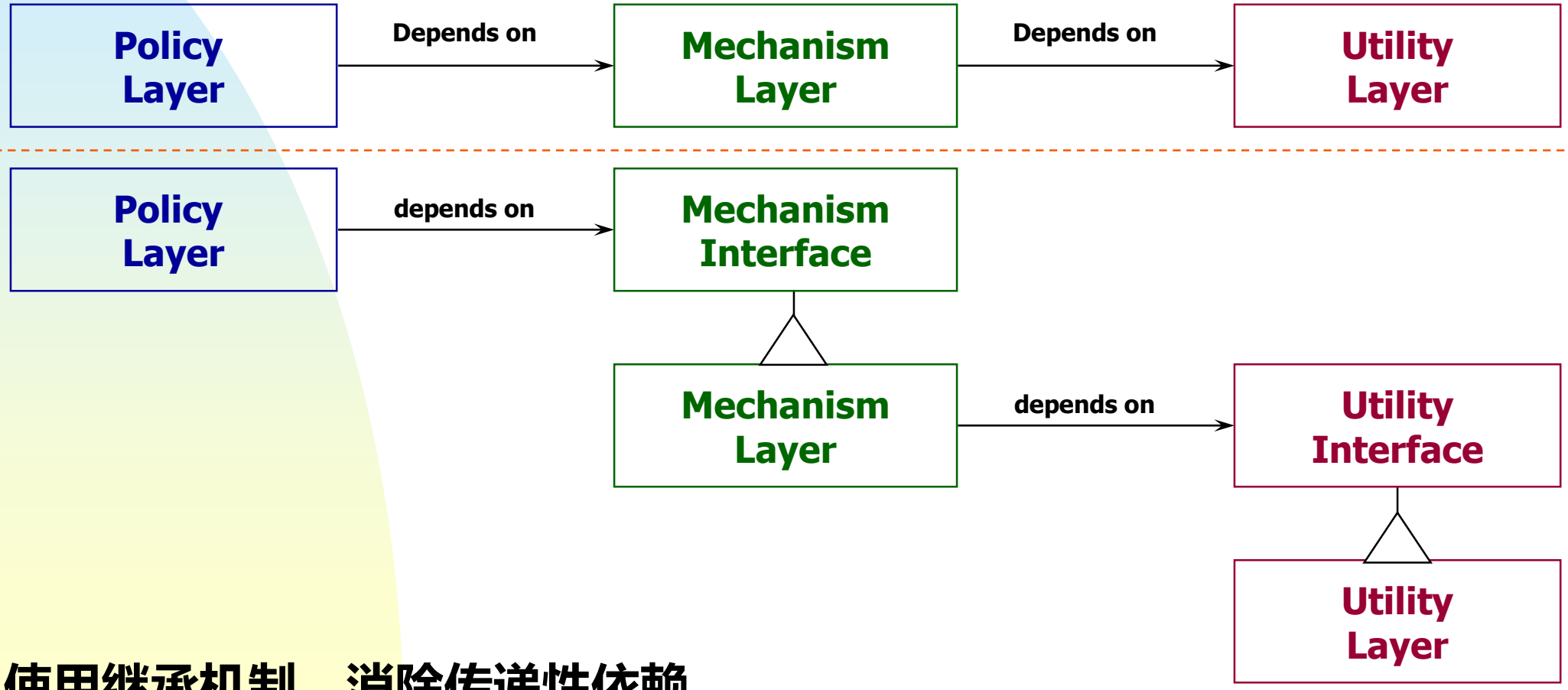
7.3 依赖倒置原则的启发1

- 为什么面向接口设计 Design to an Interface
- 因为
  - 抽象类/接口修改的概率偏低 tend to change much less frequently
  - 抽象概念容纳的范围广，易于扩展/修改 abstractions are 'hinge points' where it is easier to extend/modify
  - 不应当修改代表抽象的类/接口，符合OCP原则 shouldn't have to modify classes/interfaces that represent the abstraction (OCP)
  - 举例：中央的政策不能轻易修改，而乡镇的政策，错了马上改
- 例外情况 Exceptions
  - 有些类非常成熟、稳定 Some classes are very unlikely to change
    - 插入抽象层，好处不多了，例如 String class，这里就可以直接使用具体类
    - 此时，不考虑依赖倒置的问题了

7.4 依赖倒置原则的启发2

避免传递性依赖 Avoid Transitive Dependencies

- 在下面的例子中，Policy layer 依赖 Utility layer

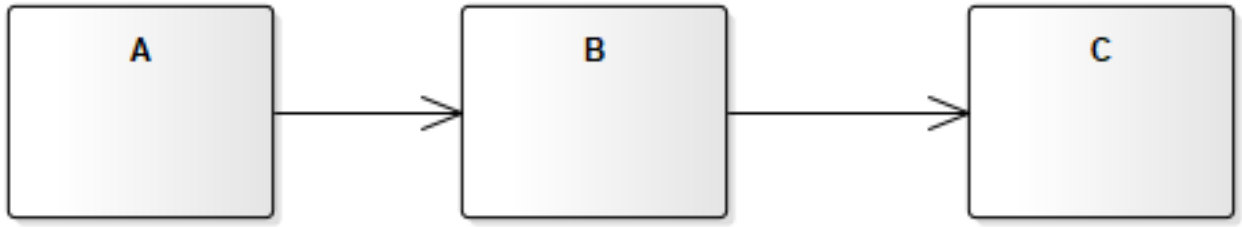


- 使用继承机制，消除传递性依赖

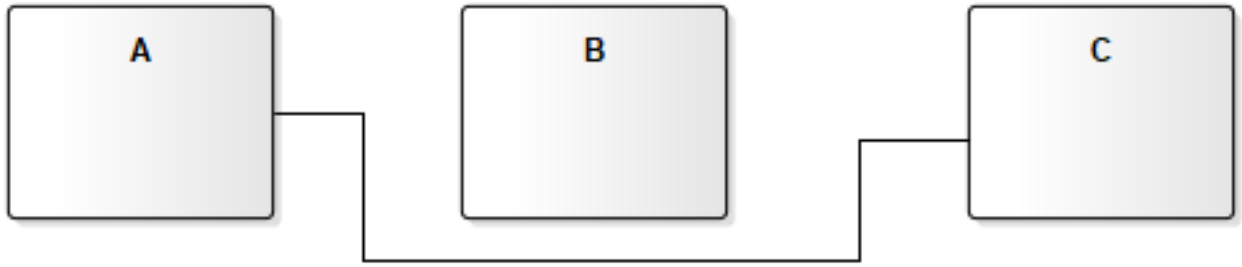
7.4 依赖倒置原则的启发3

每当有疑虑时，增加一个间接层 When in doubt, add a level of indirection

- **如果对自己设计的类找不到一个满意的解决方案，尝试把职责委派其他一个或者多个类** If you cannot find a satisfactory solution for the class you are designing, try delegating responsibility to one or more classes



- **如果B调用C违反了某些原则，考虑让A承担一下职责，让A来调用C**



7.5 依赖倒置原则 Example

- 有两个对象，Button and Lamp
  - Button:
    - senses the external environment
      - receives poll message
      - Determines whether or not user has “pressed” it
  - Lamp:
    - Affects the external environment
      - On receiving turnOn message, illuminates the light
      - On receiving turnOff message, extingishes the light
  - Actual physical mechanism for the Lamp and the Button is irrelevant

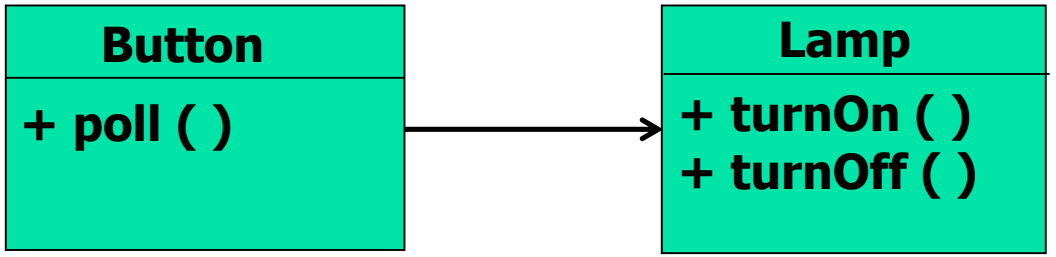
7.5 依赖倒置原则 Example: Naïve Design

```
public class Button {
     private Lamp itsLamp;
     public Button (Lamp l) { itsLamp
    = l;}

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

}

public class Lamp {
     public void turnOn ( );
     public void turnOff ( );
 }
```



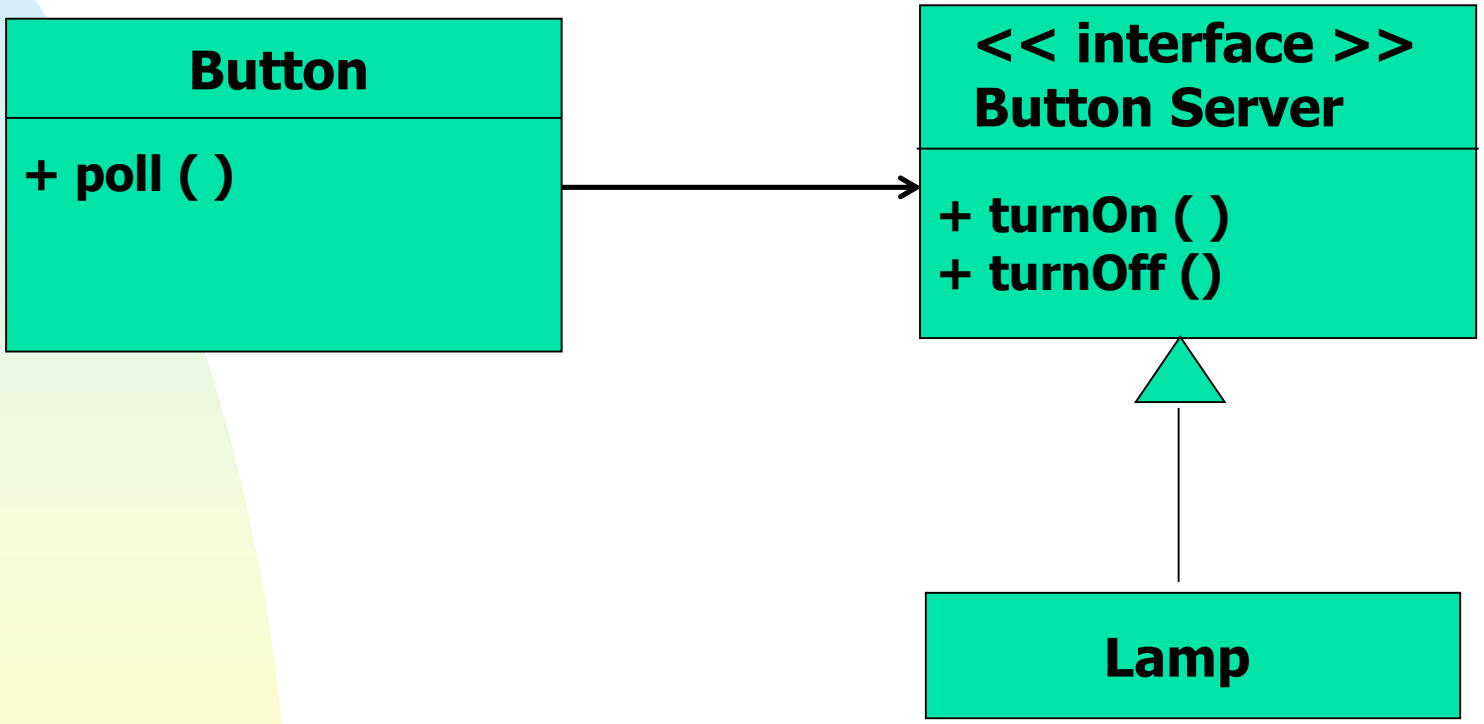
Why is this a naïve design?

7.5 依赖倒置原则 Example: Naïve Design

- **Why is the design naïve?**
  - **The dependency between Lamp and Button implies that Lamp cannot be modified without changing (at least recompiling) the code**
  - **Also - not possible to reuse the Button class to control a Motor/Portal object**
  - **The Button and Lamp code violates the DIP**

7.5 依赖倒置原则 Example

解决: Applying DIP (a)



7.5 依赖倒置原则 Example

- **Adding More Abstraction**
  - **If there can be multiple types of buttons or switching devices, abstraction can used to further refine the design!**

- 本讲结束