面向对象的设计原则

中科院软件所互联网软件技术实验室
陈烨（chenye@intec.iscas.ac.cn）

面向对象的设计原则

- 软件设计中存在的问题
- 什么是好的设计？
- 面向对象的基本设计原则

当前存在的问题

- 过于僵硬（Rigidity）
  - 很难加入新功能
- 过于脆弱（Fragility）
  - 很难修改
- 复用率低（Immobility）
  - 高层模块无法重用
- 黏度过高（Viscosity）
  - 破坏原始设计框架

什么是好的设计？

一个好的系统设计应该有如下的性质：可扩展性、灵活性、可插入性。

—Peter Code [CODE99]

设计目标

- 可扩展性（Extensibility）
  容易添加新的功能
- 灵活性（Flexibility）
  代码修改平稳 地发生
- 可插入性（Pluggability）
  容易将一个类抽出去，同时将另一个有同样接口的类加入进来

面向对象的设计原则

- “开—闭”原则（OCP）
  对可变性封装
- 里氏替换原则（LSP）
  如何进行继承
- 依赖倒转原则（DIP）
  针对接口编程
- 接口隔离原则（ISP）
  恰当的划分角色和接口
- 合成/聚合复用原则（CARP）
  尽量使用合成/聚合、尽量不使用继承
- Demeter法则（LoD）
  不要跟陌生人说话

目标与原则的关系

- 可扩展性（Extensibility）
  “开—闭”原则、里氏替换原则、依赖倒转原则、合成/聚合复用原则
- 灵活性（Flexibility）
  “开—闭”原则、 Demeter法则、接口隔离原则
- 可插入性（Pluggability）
  “开—闭”原则、里氏替换原则、依赖倒转原则、合成/聚合复用原则

开放—封闭原则（OCP）
The Open-Closed Principle

任何系统在其生命周期中都会发生变化。如果我们希望开发出的系统不会在第一版本后就被抛弃，那么我们就必须牢牢记住这一点。

— Ivar Jacobson

OCP定义

软件组成实体（类，模块，函数，等等）应该是可扩展的，但是不可修改的。

[SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.]

—Bertrand Meyer 1988

OCP特征

- 可扩展（对扩展是开放的）

  模块的行为功能可以被扩展，在应用需求改变或需要满足新的应用需求时，我们可以让模块以不同的方式工作

- 不可更改（对更改是封闭的）

  这些模块的源代码是不可改动的。任何人都不许修改模块的源代码。

自相矛盾？

开放—封闭原则（OCP）

怎么解决？

关键是抽象！

模块可以操作一个抽象体。由于模块依赖于一个固定的抽象体，因此它可以是不允许修改（closed for modification）的；同时，通过从这个抽象体派生，也可扩展此模块的行为功能。

开放—封闭原则（OCP）

- 符合OCP原则的程序只通过增加代码来变化而不是通过更改现有代码来变化，因此这样的程序就不会引起象非开放—封闭（open-closed）的程序那样的连锁反应的变化。

对可变性的封装

- 考虑系统中什么可能会发生变化
- 一种可变性不应当散落在代码的很多角落里，而应当被封装到一个对象里

正确理解继承

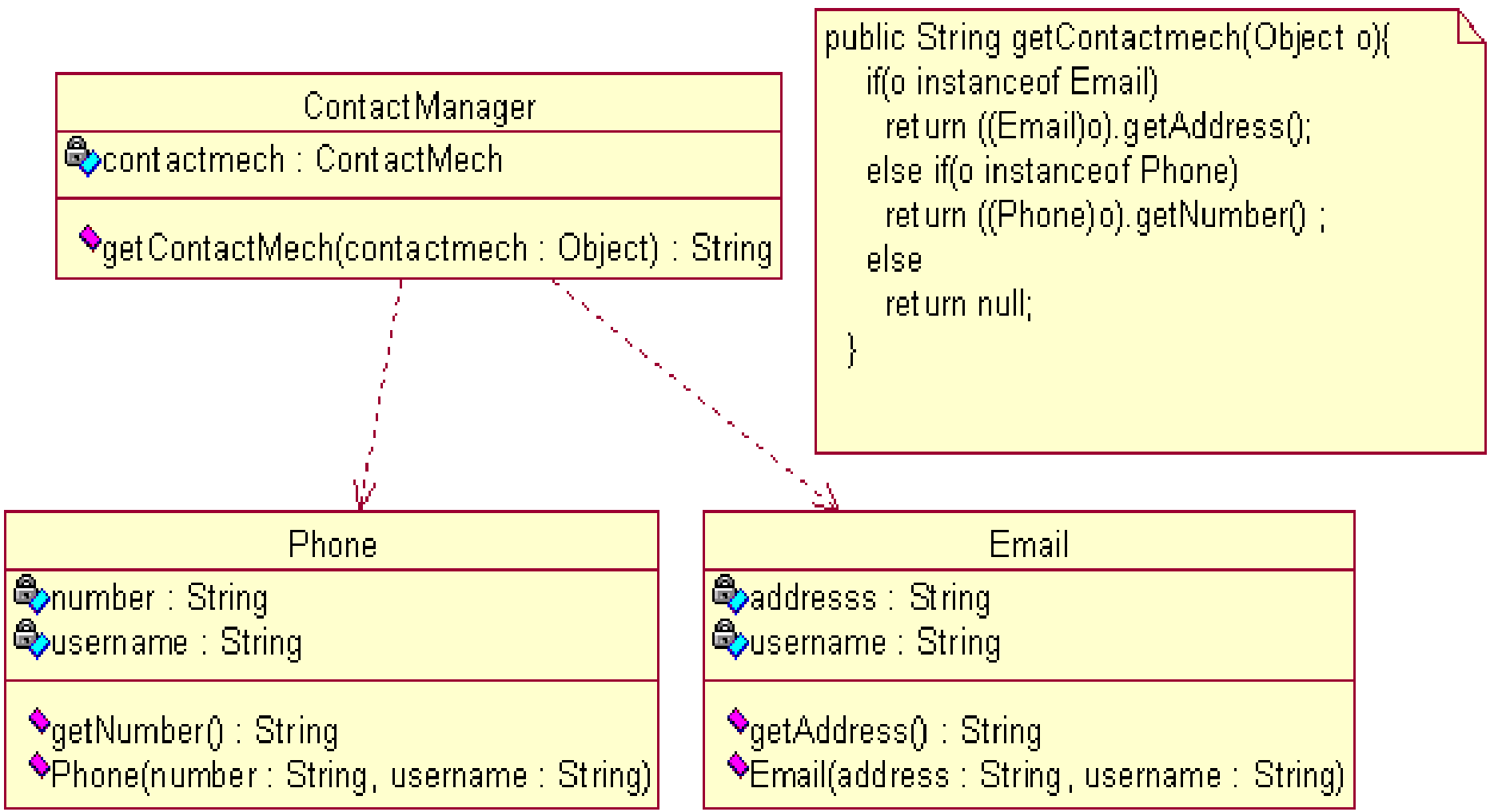
- 一种可变性不应当与另一个可变性混合在一起

开放—封闭原则（OCP）

- 选择性的封闭（Strategic Closure）

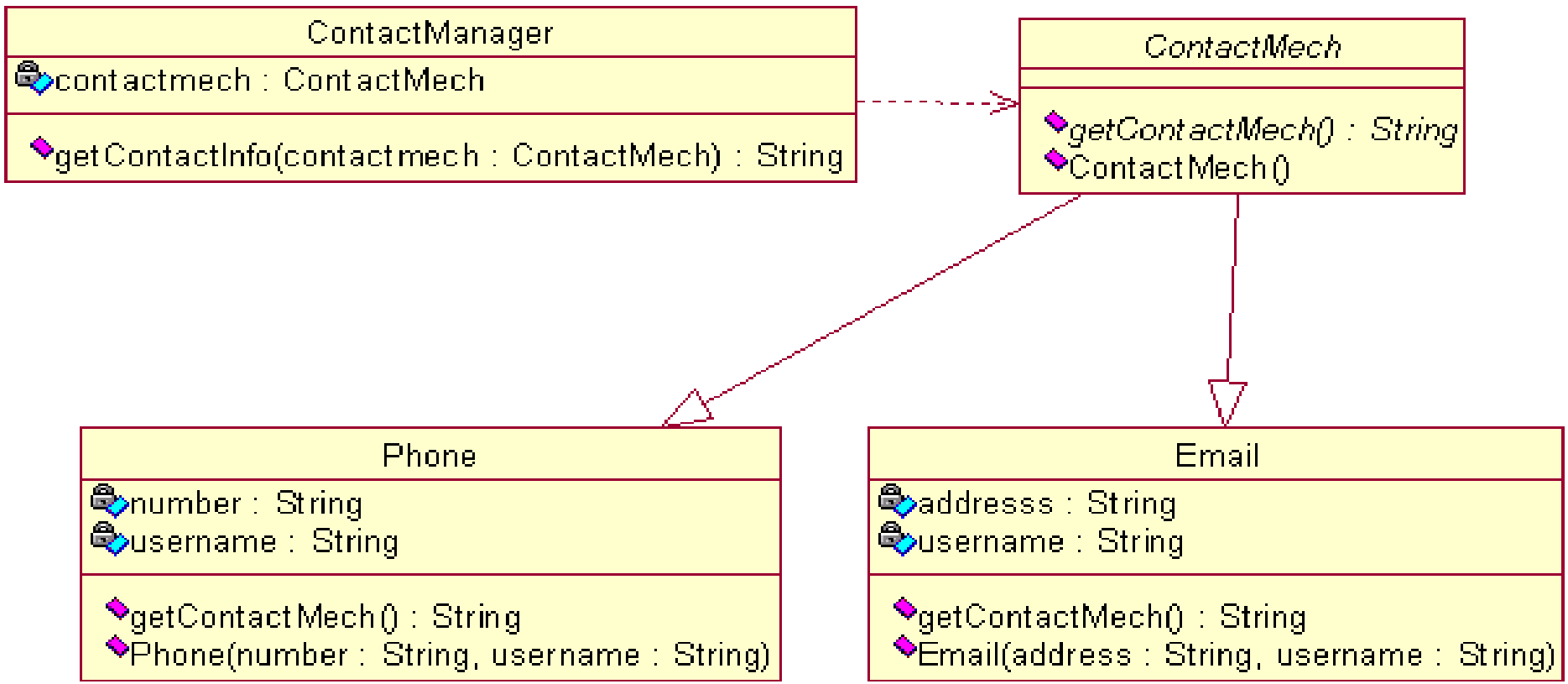
  没有任何一个大的程序能够做到100%的封闭。一般来讲，无论模块是多么的“封闭”，都会存在一些无法对之封闭的变化。既然不可能完全封闭，因此就必须选择性地对待这个问题。也就是说，设计者必须对于他（她）设计的模块应该对何种变化封闭做出选择。

实例一解决方案1


ContactManager
contactmech : ContactMech
getContactMech(contactmech : Object) : String
public String getContactmech(Object o){
if(o instanceof Email)
return ((Email)o).getAddress();
else if(o instanceof Phone)
return ((Phone)o).getNumber() ;
else
return null;
}
Phone
number : String
username : String
getNumber() : String
Phone(number : String, username : String)
Email
addresss : String
username : String
getAddress() : String
Email(address : String, username : String)

实例一解决方案2

```
public String getContactInfo(ContactMech contactmech){
    return contactmech.getContactMech();
  }
```



OCP与设计模式

- 几乎所有的设计模式都是对不同的可变性进行封装，从而使系统在不同角度上达到“开－闭”原则的要求

Liskov替换原则LSP

The Liskov Substitution Principle

OCP原则背后的主要机制是抽象和多态。
支持抽象和多态的关键机制是继承。

当前存在的普遍的现象：滥用继承

LSP的定义

若对于每一个类型S的对象o1，都存在一个类型T的对象o2，使得在所有针对T编写的程序P中，用o1替换o2后，程序P的行为功能不变，则S是T的子类型。

What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

—Liskov 1987

例子

```
public class Rectangle{
    private long width;
    private long height;
   public void setWidth(long width){
        this.width = width;
   }
     public void setHeight(long height) {
        this.height = height;
     }
    public void resizeWidth(){
         this.width+1;
    }
}
```

```
public class Square extends Rectangle {
  public void setWidth(long width){
    this.width = width;
    this.height=width;
  }
  public void setHeight(long height) {
    this.height = height;
    this.width = height;
  }
  public void resizeWidth(){
        this.width+1;
        this.height+1;
    }
}
```

例子

```
public void f(Rectangle r){
    while (r. getWidth() <= r. getHeight() ){
      r. resizeWidth();
    }
  }
```

为什么不对？？

- LSP原则清楚地指出，OOD中Is-A关系是就行为功能而言。行为功能（behavior）不是内在的、私有的，而是外在、公开的，是客户程序所依赖的。

行为功能（behavior）才是软件所关注的问题！所有派生类的行为功能必须和客户程序对其基类所期望的保持一致。

LSP和DBC

DBC（Design by Contract） 定义

把类和其客户之间的关系看作是一个正式的协议，明确各方的权利和义务。

DBC对类的要求

类的方法声明为先决条件（precondition）和后续条件（postcondition）。为了让方法得以执行，先决条件必须为真。完成后，方法保证后续条件为真。

DBC对派生类的要求

当重新定义派生类中的例行程序时，我们只能用更弱的先决条件和更强的后续条件替换之。

when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

例子

```
public class Rectangle{
    private long width;
    private long height;
   public void setWidth(long width){
        this.width = width;
   }
     public void setHeight(long height) {
        this.height = height;
     }
    public void resizeWidth(){
         this.width+1;
    }
}
```

```
public class Square extends Rectangle {
  public void setWidth(long width){
    this.width = width;
    this.height=width;
  }
  public void setHeight(long height) {
    this.height = height;
    this.width = height;
  }
  public void resizeWidth(){
        this.width+1;
        this.height+1;
    }
}
```

LSP—结论

- LSP原则是符合OCP原则应用程序的一项重要特性。仅当派生类能完全替换基类时，我们才能放心地重用那些使用基类的函数和修改派生类型。

提示

- 面向对象程序设计中的继承概念与通常的数学法则和生活常识是不同范畴的事物，有不可混淆的区别。

依赖倒置原则DIP

Dependence Inversion Principle

再看糟糕的设计，想想原因

- 很难添加新功能，因为每一处改动就会影响系统中过多的模块。（缺乏灵活性）
- 当你做了一处改动，却导致系统的另一个模块发生了问题。（脆弱性）
- 很难在别的应用程序中重用这个模块，因为不能将它从现有的应用程序中独立的提取出来。（不可重用性）

依赖倒置原则DIP
Dependence Inversion Principle

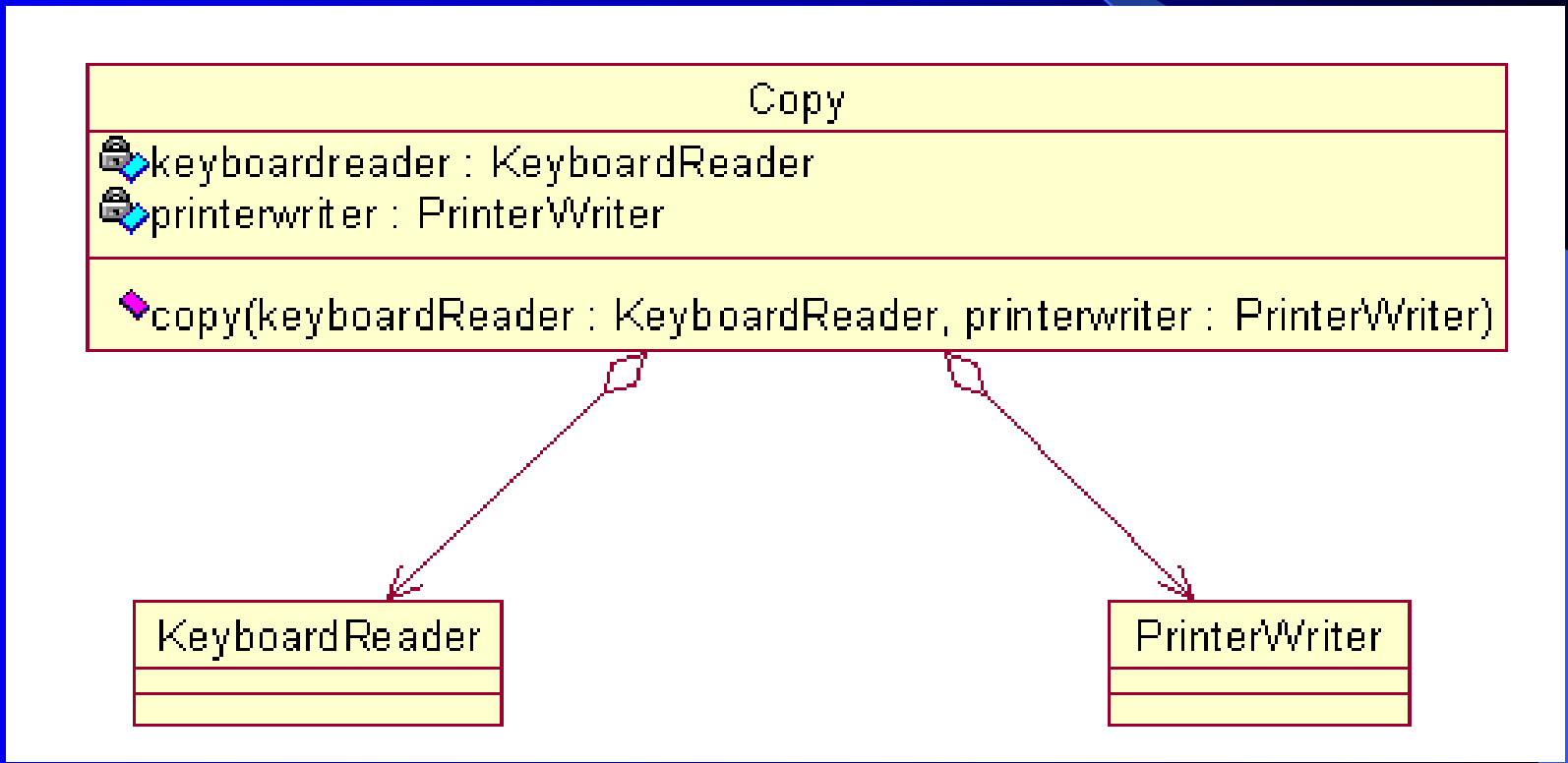
- 什么原因？

耦合关系

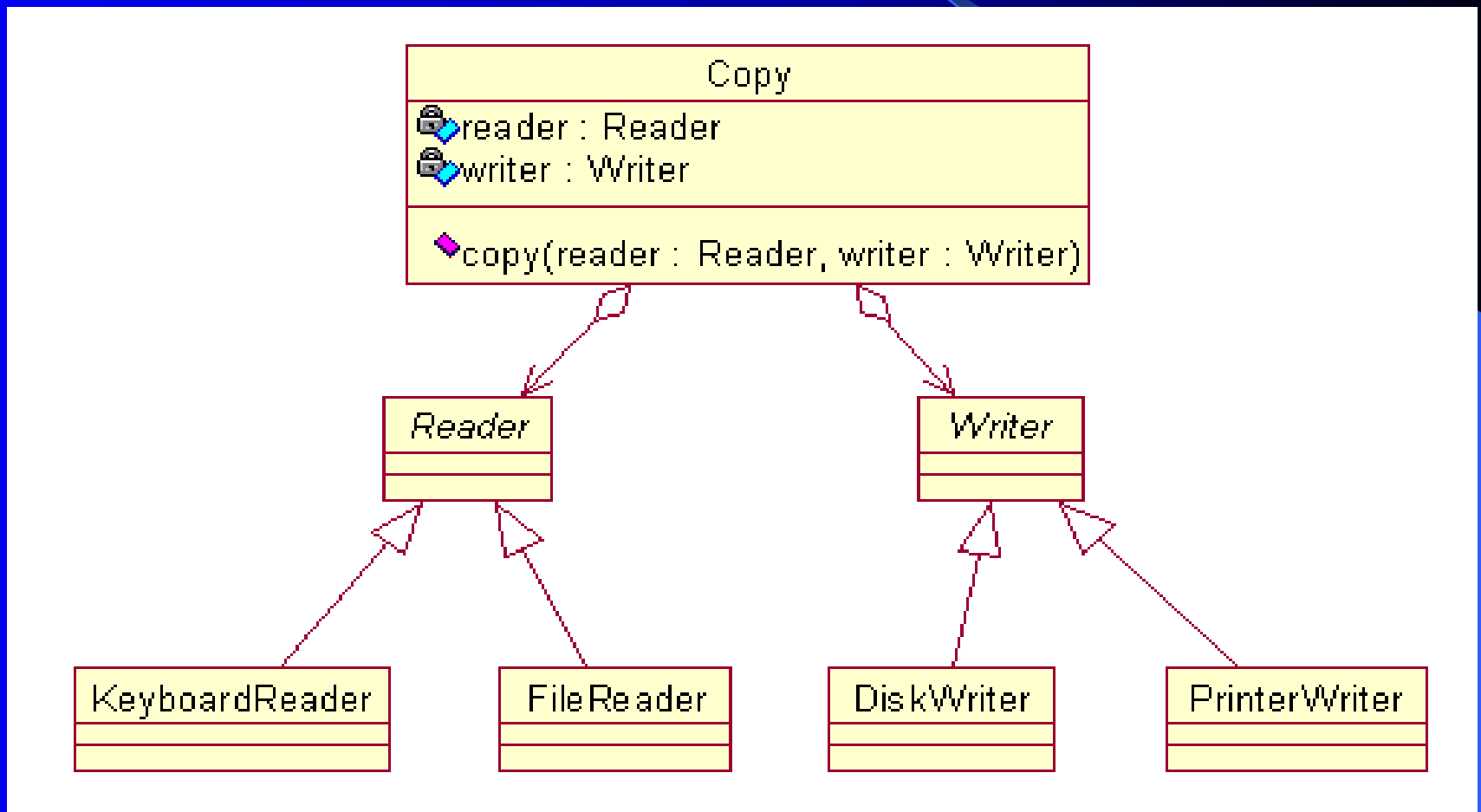
类之间耦合关系

- 零耦合（Nil Coupling）
  两个类没有耦合关系
- 具体耦合（Concrete Coupling）
  两个具体的（可实例化的）类之间，经由一个类对另一个具体类的直接引用造成的耦合关系
- 抽象耦合（Abstract Coupling）
  一个具体类和一个抽象类（或接口）之间的耦合关系

具体耦合例子


Copy
keyboardreader : KeyboardReader
printerwriter : PrinterWriter
copy(keyboardReader : KeyboardReader, printerwriter : PrinterWriter)
KeyboardReader
PrinterWriter

抽象耦合例子


Copy
reader : Reader
writer : Writer
copy(reader : Reader, writer : Writer)
Reader
Writer
KeyboardReader
FileReader
DiskWriter
PrinterWriter

DIP原则

- 高层模块不应该依赖于低层模块。二者都应该依赖于抽象。
- 抽象不应该依赖于细节。细节应该依赖于抽象。

DIP实施重点

从问题的具体细节中分离出抽象，以抽象方式对类进行耦合

DIP的不足

- 导致生成大量的类
- 假定所有的具体类都是会变化的，这也不总是正确的

DIP与设计模式

DIP以LSP为基础，是实现OCP的主要手段，是设计模式研究和应用的主要指导原则，

接口隔离原则ISP
Interface Segregation Principle

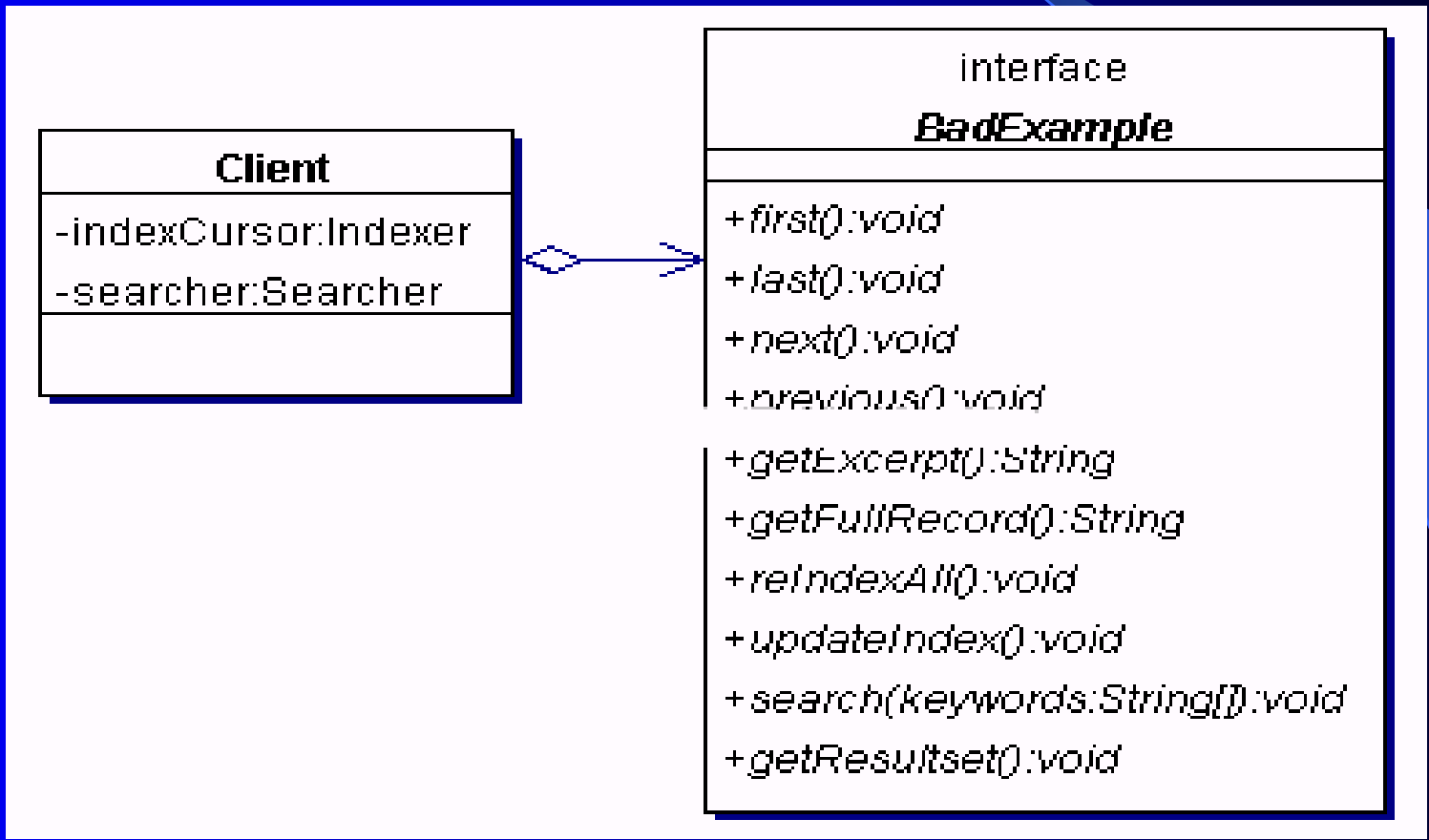
- 接口的污染（Interface Contamination）
  一个没有经验的设计师往往想节省接口的数目，将一些功能相近或功能相关的接口合并，并将这看成是代码优化的一部分。

接口隔离原则ISP

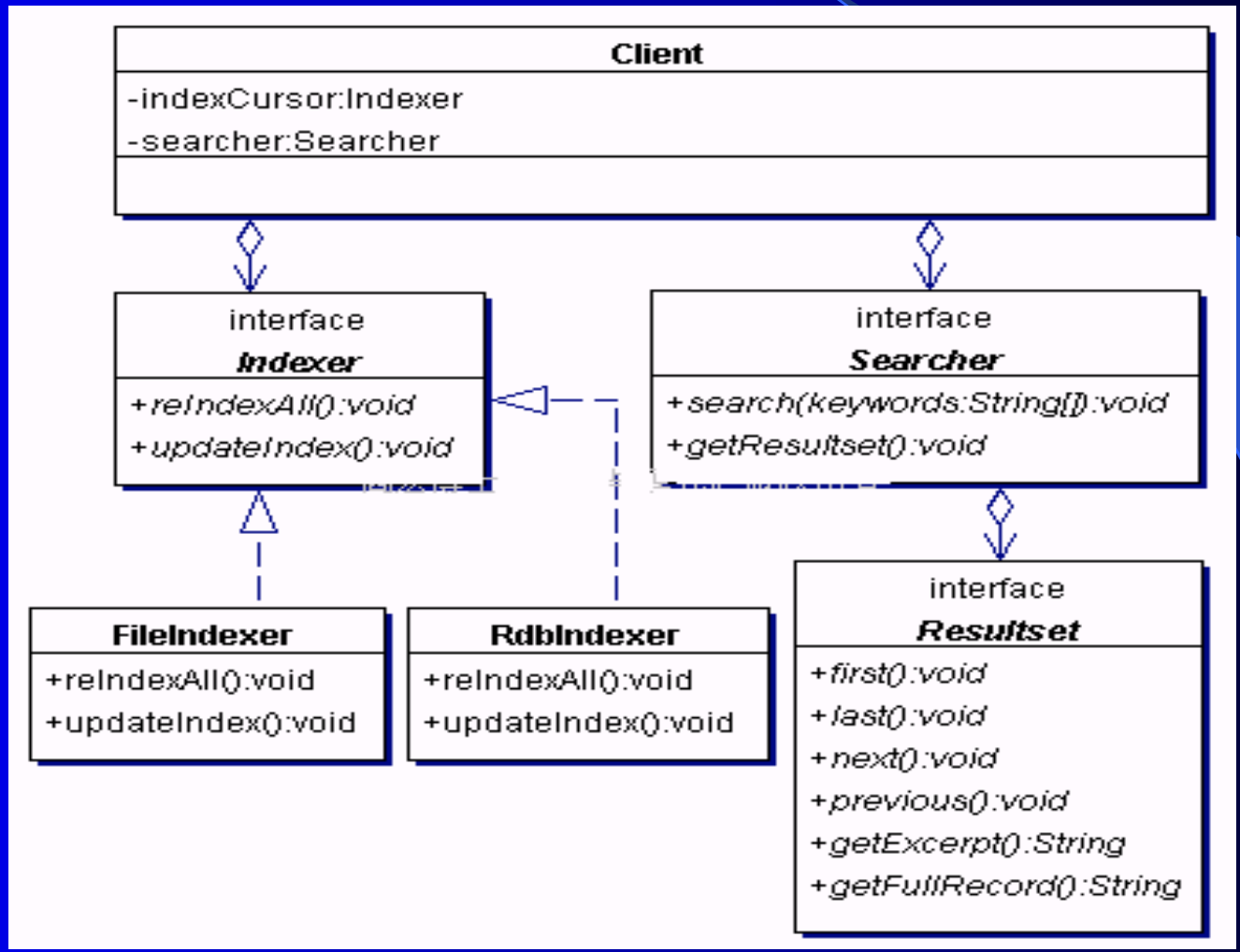
定义：

从一个客户类的角度来讲：一个类对另外一个类的依赖性应当是建立在最小的接口上的。使用多个专门的接口比使用单一的总接口要好

ISP例子


Client
-indexCursor:Indexer
-searcher:Searcher
interface
BadExample
+first():void
+last():void
+next():void
+previous():void
+getExcerpt():String
+getFullRecord():String
+reIndexAll():void
+updateIndex():void
+search(keywords:String[]):void
+getResultset():void

ISP例子


Client
-indexCursor:Indexer
-searcher:Searcher
interface
Indexer
+reIndexAll():void
+updateIndex():void
interface
Searcher
+search(keywords:String[]):void
+getResultset():void
FileIndexer
+reIndexAll():void
+updateIndex():void
RdbIndexer
+reIndexAll():void
+updateIndex():void
interface
Resultset
+first():void
+last():void
+next():void
+previous():void
+getExcerpt():String
+getFullRecord():String

合成/聚合复用原则 CARP
Composite/Aggregate Reuse Principle

- 定义：在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；新的对象通过向这些对象的委派达到复用这些对象的目的

复用技术—继承

优点：

- 新的实现较为容易
- 比较容易添加到已有系统中

缺点：

- 破坏封装
- 很难处理超类的变化
- 继承的实现是静态的

复用技术—合成/聚合

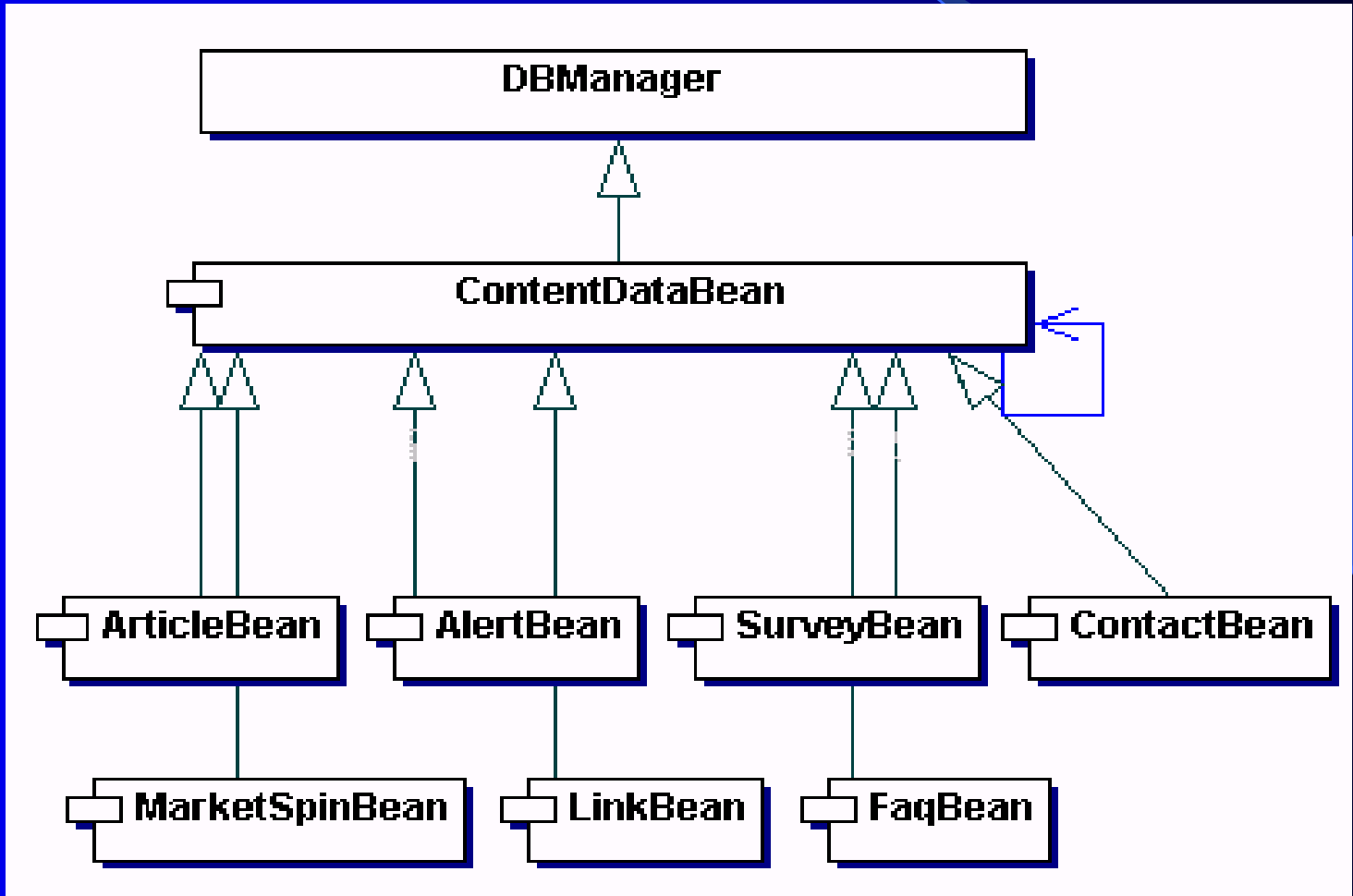
优点

- 支持封装
- 支持包装
- 复用可以动态进行

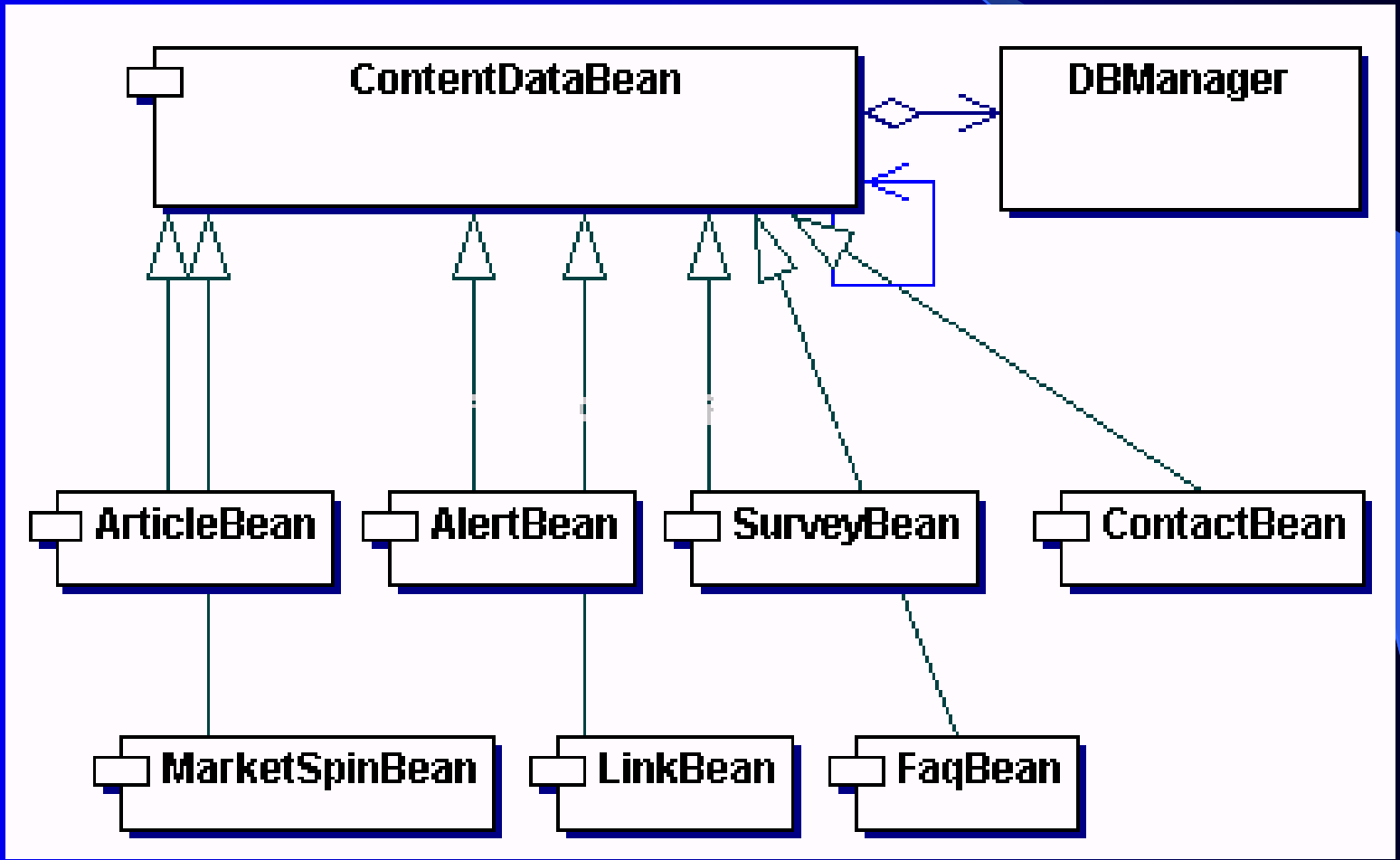
缺点

- 不易扩展已有系统
- 需要较多的对象管理

CARP例子


DBManager
ContentDataBean
ArticleBean
AlertBean
SurveyBean
ContactBean
MarketSpinBean
LinkBean
FaqBean

CARP例子


ContentDataBean
DBManager
ArticleBean
AlertBean
SurveyBean
ContactBean
MarketSpinBean
LinkBean
FaqBean

迪米特法则 LoD
Law of Demeter

- 最少知识原则
- 只与你直接的朋友们通信
- 不要跟“陌生人”说话

一个对象应当对其它对象有尽可能少的了解

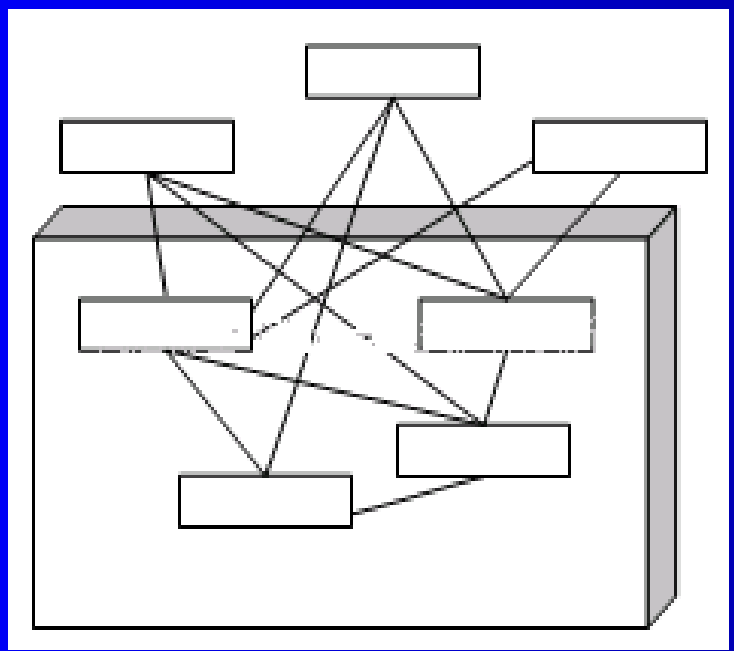
迪米特法则 LoD

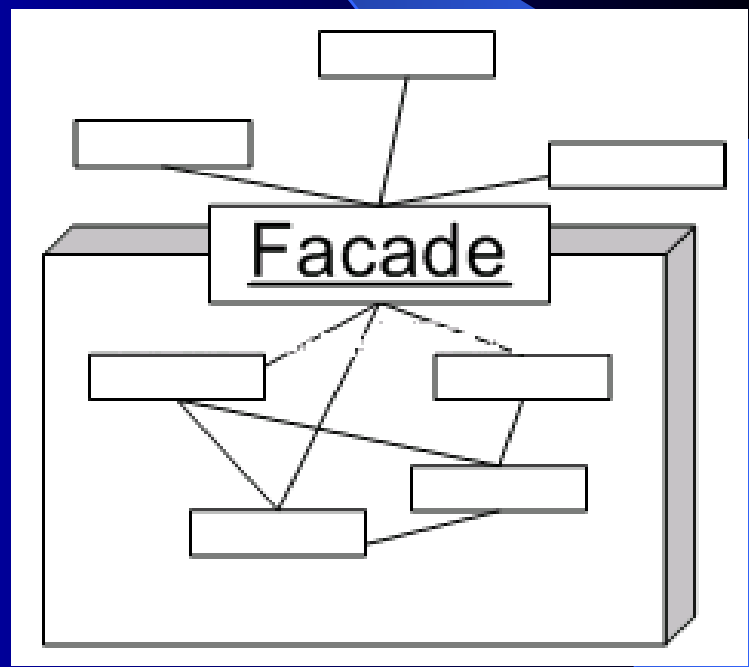
朋友的条件

- 当前对象本身（this）
- 以参量形式传入到当前对象方法中的对象
- 当前对象的实例变量直接引用的对象
- 当前对象的实例变量如果是一个聚集，那么聚集中的元素也都是朋友
- 当前对象所创建的对象

LoD与设计模式

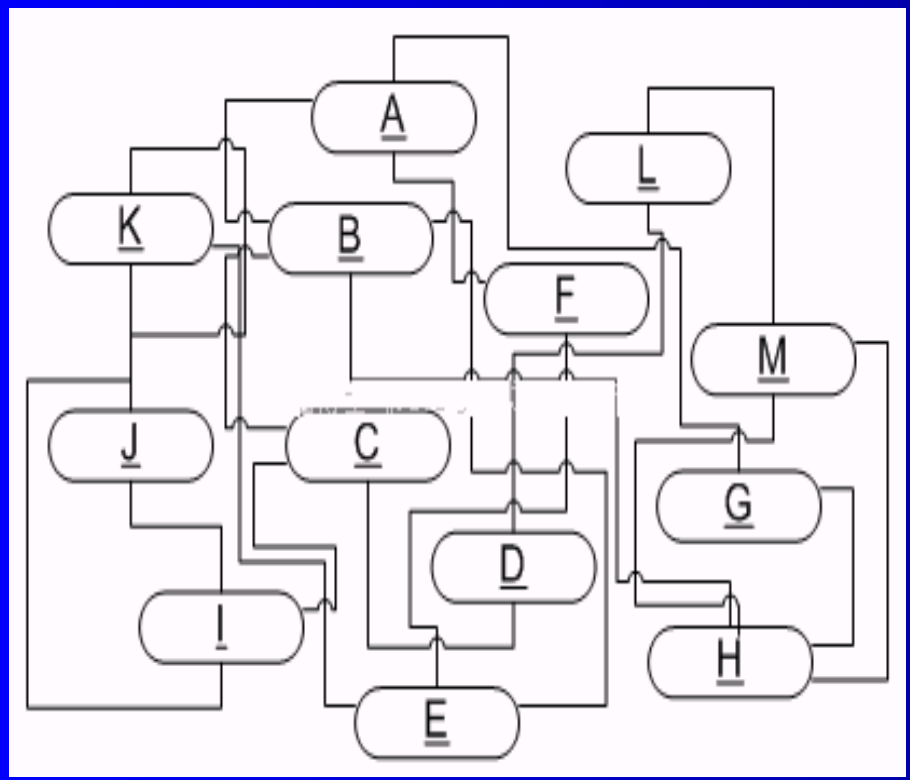
Façade模式

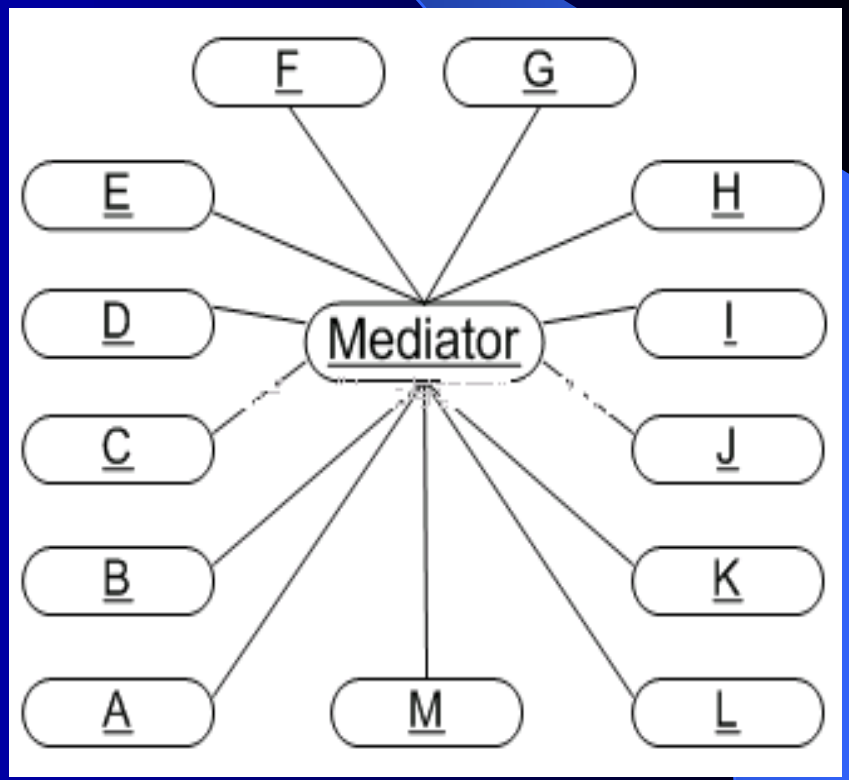




LoD与设计模式

调停者（Mediator）模式





总结

- “开—闭”原则（OCP）
  对可变性封装
- 里氏替换原则（LSP）
  如何进行继承
- 依赖倒转原则（DIP）
  针对接口编程
- 接口隔离原则（ISP）
  恰当的划分角色和接口
- 合成/聚合复用原则（CARP）
  尽量使用组合/聚合、尽量不使用继承
- Demeter法则（LoD）
  不要跟陌生人说话