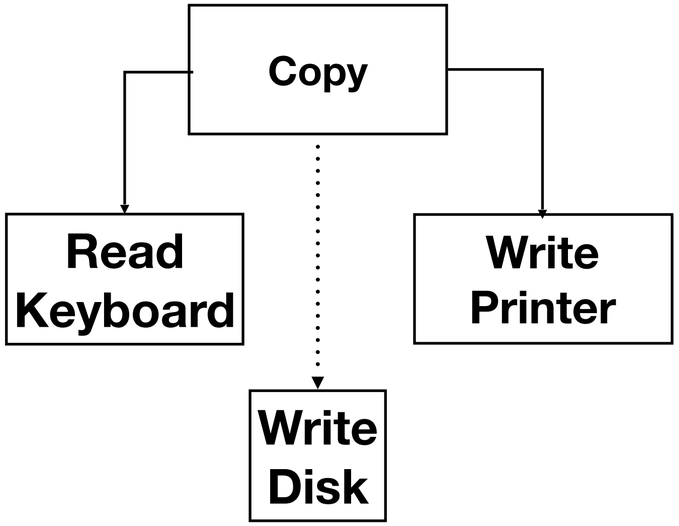
“面向对象”的信息隐藏

知识点

- 类的职责（P9）
- 封装实现的细节（P17）
- 权限最小化原则（P36）
- OCP（P48）
- DIP（P58，P64）

Quiz




```
void Copy(ReadKeyboard& r,
WritePrinter& wp, WriteDisk& wd,
OutputDevice dev){
   int c;
   while((c = r.read())!= EOF)
      if(dev == printer)
         wp.write(c);
      else
         wd.write (c);
}
```

```
void Copy(ReadKeyboard& r, WritePrinter& w){
   int c;
   while ((c = r.read ()) != EOF)
     w.write (c);
}
```

Outline

- 封装类的职责
  - 类的职责
  - 类的封装
- 为变更而设计

回顾结构化设计的信息隐藏

Information Hiding

- Each module hides the implementation of an important design decision (secrets) so that only the constituents of that module know the details

Design Secrets need to hide…

- Primary Secret: Responsibility Change
  - Hidden information that was specified to the software designer
  - From SRS
- Secondary Secret: Implementation Change
  - The implementation decisions made by the designer when implementing the module designed to hide the primary secret
  - 变化；

类的职责

类的职责

- 什么是职责?
- 职责来源于哪?
- 职责如何体现?

什么是职责?

- 类或对象维护一定的状态信息
- 基于状态履行行为职能的能力。

职责来源于哪?

- 来源于需求

- 业务类
  - Sales、Order
- 辅助类
  - View、Data、exception、transaction

职责如何体现?

- 封装

封装

- 信息隐藏
- 分为接口和实现
  - The interface is the visible surface of the capsule.
    - describes the essential characteristics of objects of the class which are visible to the exterior world
  - The implementation is hidden in the capsule.
    - The implementation hiding means that data can only be manipulated, that is updated, within the class, but it does not mean hiding interface data.

面向对象中的接口

- 对象之间交互的消息（方法名）
- 消息中的所有参数
- 消息返回结果的类型
- 与状态无关的不变量
- 需要处理的异常

实现的细节

- Data
- Structure
- Other object
- Type
- Change/vary
- …

类的封装

封装实现的细节

- A 封装数据和行为
- B 封装内部结构
- C 封装其他对象的引用
- D 封装类型信息
- E 封装潜在变更

A 封装数据类型

封装的源头 — ADT

- ADT = Abstract Data Type
  - A concept, not an implementation
  - A set of (homogeneous) objects together with a set of operations on those objects
  - No mention of how the operations are implemented
  - Example：栈
- Encapsulation = data abstraction + type
  - data abstraction: group data and operation
  - Type: hiding implementation, make usage correctly

Why type?

- A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use.
- It provides a protective covering that hides the underlying representation and constrains the way objects may interact with other objects.
- In an untyped system untyped objects are naked in that the underlying representation is exposed for all to see.

```
public class Position{
  //私有成员变量
    private double latitude;
    private double longtitude;

    public double getLatitude(){
    }
    public double getLongtitude(){
    }
    public void setLatitude(double latitude){
    }
    public void setLongtitude (double longtitude){
    }

    public double calculateDistance(Position pos){
    // 计算当前点到pos点的距离
    }
    public double calculateDirection(Position pos){
    // 计算当前点到pos点的方向
    }
}
```

封装数据和行为

数据的封装 — Accessors and Mutators

- If needed, use Accessors and Mutators, Not Public Members
- Accessors and Mutators is meaningful behavior
  - Constraints, transformation, format…

```
public void setSpeed(double newSpeed) {
    if (newSpeed < 0) {
        sendErrorMessage(...);
        newSpeed = Math.abs(newSpeed);
    }
    speed = newSpeed;
}
```

B 封装内部结构

```
public class Route {

    private Position[] positions;

    public Route( int segments )
    {
        positions = new Position[ segments + 1 ];
    }
    // 暴露的接口也是直接对内部接口进行操作
    public void setPosition( int index, Position position )
    {
        positions[ index ] = position;
    }
    public Position getPosition( int index )
    {
        return position[ index ];
    }
    // 暴露内部结构
    public Position[] getPositions()
    {
        return positions;
    }
    public double distance( int segmentNumber )
    {
        // 计算分段的距离
    }

    public double heading( int segmentNumber )
    {
        // 计算分段方向
    }
}
```

暴露了内部结构

```
public class Route {

        private Position[] positions;

        // 暴露的接口是抽象的行为
        public void append( Position position )
        {
            positions.append( position );
        }
        // 隐藏了类的内部结构
        public Position getPosition( int index )
        {
            return positions.get( index );
        }
        ...
}
```

隐藏内部结构

Collection暴露了内部的结构

- See chapter 16 Iterator Pattern

```
class Album {
  private List tracks =new ArrayList();
  public List getTracks() {
    return tracks;
   }
}
```

References and Collection Data-Type !

```
f()
{
     Collection list = new HashSet();
     g(c.iterator());
}
g(Iterator i)
{
      while(i.hasNext())
        do_something_with(i.next());
}
```

迭代器实现

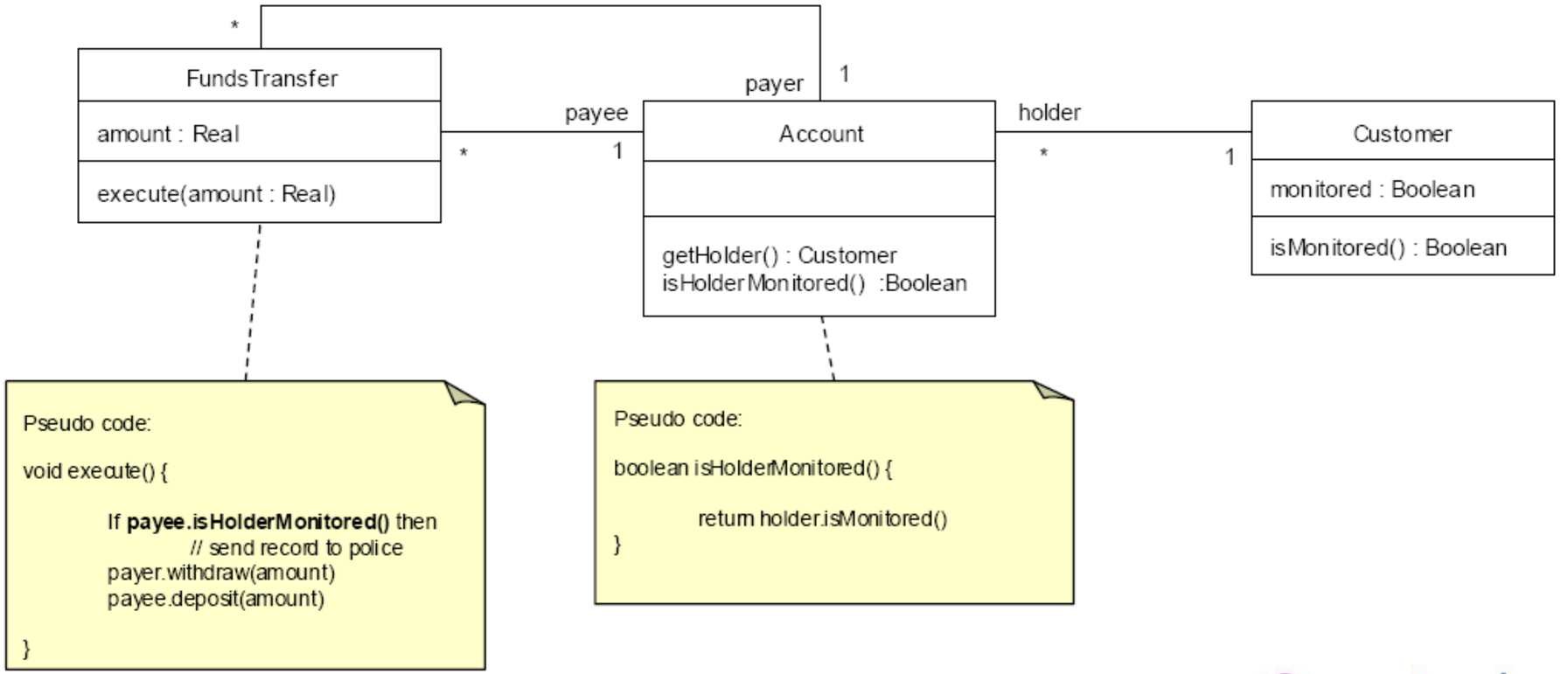
C 封装其他对象的引用

隐藏内部对象

```
public Position getPosition( int index )
{
        // 重新构造了一个对象返回，隐藏了实现细节
        Position position = new Position( positions.get( index ) );
        return position;
}
```

委托隐藏了与其他对象的协作

- Collaboration Design
  - Composition; delegation



D 封装类型信息

LSP的隐藏

- LSP
  - pointers to superclasses or interfaces;

- All derived classes must be substitutable for their base class

- 使用父类的接口隐藏子类的类型信息

E 封装潜在的变更

Encapsulate Change

- Identify the aspects of your application that may change and separate them from what stays the same.

- Take the parts that change and encapsulate them, so that later you can alter or extend the parts that vary without affecting the parts that don’t.

```
public class Position
{     //私有成员变量
      private double phi;
      private double theta;

      public double getLatitude(){
      }
      public double getLongtitude(){
          //极坐标向经纬度转换
          //返回经度
      }
      public void setLatitude(double latitude){
          //极坐标向经纬度转换
          //返回经度
      }
      public void setLongtitude (double longtitude){
      }

      public double calculateDistance(Position pos){
      //计算当前点到pos点的距离
      }
      public double calculateDirection(Position pos){
      //计算当前点到pos点的方向
      }
}
```

封装变更

Principle 10: Minimize The Accessibility of Classes and Members

- Abstraction
  - An abstraction focuses on the outside view of an object and separates an object's behavior from its implementation
- Encapsulation
  - Classes should not expose their internal implementation details
- 权限最小化原则

Access Specifier	Class	Package	Subclass	World
`private`	x			
None	x	x	x*	
`protected`	x	x	x**	
`public`	x	x	x	x

类和成员的可访问性

* Subclasses within the same package can also access members that lack access specifiers (default or package-private visibility). An additional requirement for access is that the subclasses must be loaded by the class loader that loaded the class containing the package-private members. Subclasses in a different package cannot access such package-private members.
** To reference a protected member, the accessing code must be contained either in the class that defines the protected member or in a subclass of that defining class. Subclass access is permitted without regard to the package location of the subclass.

Noncompliant Code Example (Public Class)

This noncompliant code example defines a class that is internal to a system and not part of any public API. Nonetheless, this class is declared public.

```
public final class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Even though this example complies with OBJ01-J. Declare data members as private and provide accessible wrapper methods, untrusted code could instantiate `Point` and invoke the public `getPoint()` method to obtain the coordinates.

Example

Compliant Solution (Final Classes with Public Methods)

This compliant solution declares the `Point` class as package-private in accordance with its status as not part of any public API:

```
final class Point {
  private final int x;
  private final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Example

构造方法包内可见

Compliant Solution (Nonfinal Classes with Nonpublic Methods)

This compliant solution declares the `Point` class and its `getPoint()` method as package-private, which allows the `Point` class to be nonfinal and allows `getPoint()` to be invoked by classes present within the same package and loaded by a common class loader:

```
class Point {
  private final int x;
  private final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Example

get方法包内可见

Noncompliant Code Example (Public Class with Public Static Method)

This noncompliant code example again defines a class that is internal to a system and not part of any public API. Nonetheless, the class `Point` is declared public.

```
public final class Point {
  private static final int x = 1;
  private static final int y = 2;

  private Point(int x, int y) {}

  public static void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Even though this example complies with OBJ01-J. Declare data members as private and provide accessible wrapper methods, untrusted code could access `Point` and invoke the public static `getPoint()` to obtain the default coordinates. The attempt to implement instance control using a private constructor is futile because the public static method exposes internal class contents.

Example

Compliant Solution (Package-Private Class)

This compliant solution reduces the accessibility of the class to package-private:

```
final class Point {
  private static final int x = 1;
  private static final int y = 2;

  private Point(int x, int y) {}

  public static void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Access to the `getPoint()` method is restricted to classes located within the same package. Untrusted code is prevented from invoking `getPoint()` and obtaining the coordinates.

Example

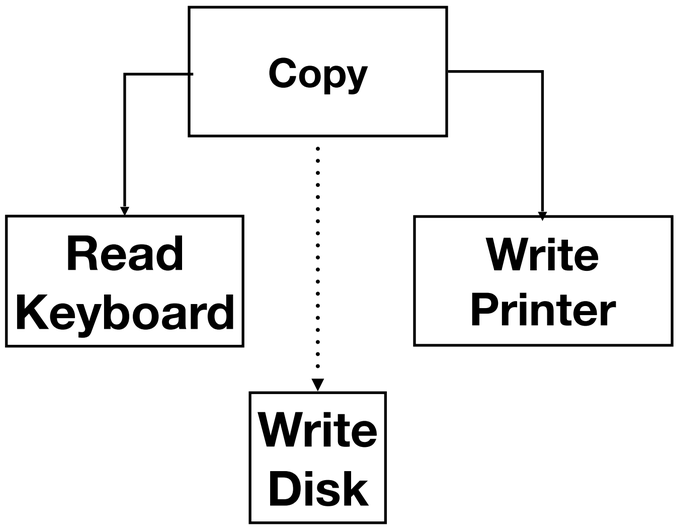
类也是包内可见

Outline

- 封装类的职责
- 为变更而设计
  - OCP
  - 多态
  - DIP

OCP

Example of Responsibility Change

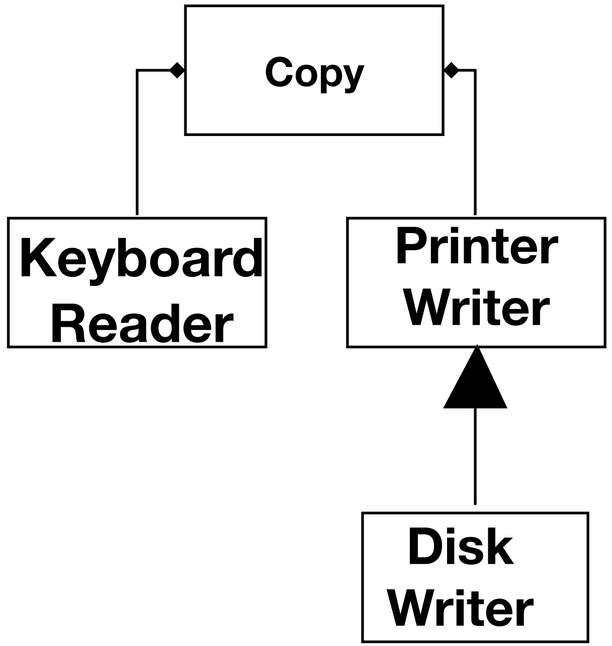



```
void Copy(ReadKeyboard& r,
WritePrinter& wp, WriteDisk& wd,
OutputDevice dev){
   int c;
   while((c = r.read())!= EOF)
      if(dev == printer)
         wp.write(c);
      else
         wd.write (c);
}
```

How to …

- Abstraction is Key
  - ...using polymorphic dependencies (calls)

Example of Responsibility Change




```
DiskWriter::Write(c)
{
    WriteDisk(c);
}
```

```
void Copy(ReadKeyboard& r, WritePrinter& w){
  int c;
  while ((c = r.read ()) != EOF)
    w.write (c);
}
```

Principle 11: Open/Closed Principle (OCP)

- Software entities should be open for extension, but closed for modification
- —- B. Meyer, 1988 / quoted by R. Martin,1996
- Be open for extension
  - module's behavior can be extended
- Be closed for modification
  - source code for the module must not be changes
- 统计数据表明，修正bug最为频繁，但是影响很小；新增需求数量一般，但造成了绝大多数影响
- Modules should be written so they can be extended without requiring them to be modified

OCP

- RTTI is Ugly and Dangerous!
  - RTTI = Run-Time Type Information
  - If a module tries to dynamically cast a base class pointer to several derived classes, any time you extend the inheritance hierarchy, you need to change the module
  - recognize them by type switch or if-else-if structures

// RTTI violating the //open-closed principle and LSP

- class Shape {}
- class Square extends Shape {
  void drawSquare() {
  // draw
- }
- }
  class Circle extends Shape {
  void drawCircle() {
  // draw
- }
- }

- void drawShapes(List<Shape> shapes) {
  for (Shape shape : shapes) {
  if (shapes instanceof Square) {
  ((Square) shapes).drawSquare();
  } else if (shape instanceof Circle) {
  ((Circle) shape).drawCircle();
- }
- }
- }

多态

表 15-1　多态的分类

多态	一般性多态	子类型多态
		参数化多态
	临时性多态	重载（overloading）
		强制转换

多态的分类

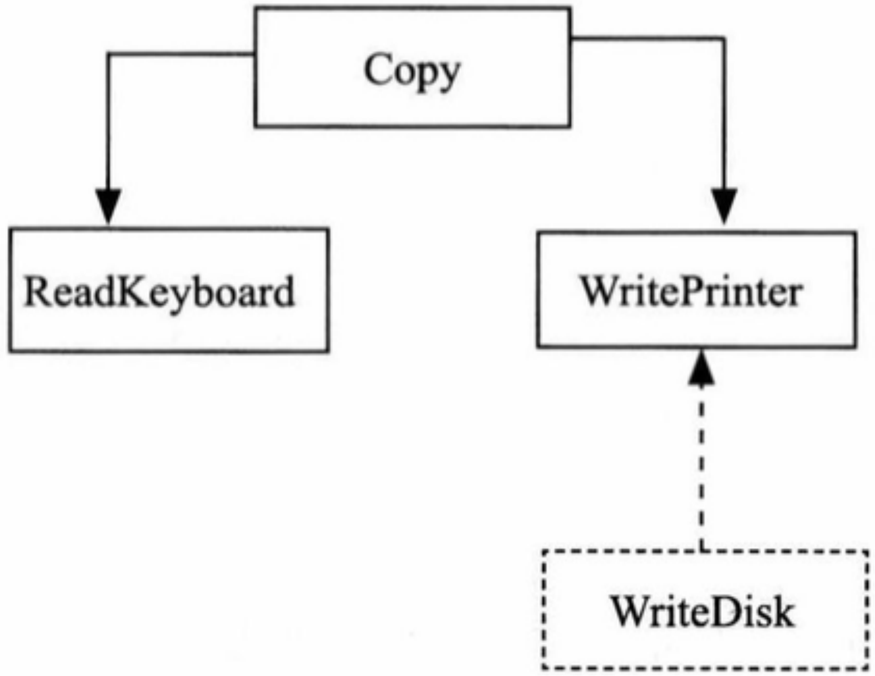
// Abstraction and Polymorphism that does // not violate the open-closed principle and LSP

```
interface Shape {
    void draw();

}

class Square implements Shape {
    void draw() {
        // draw implementation

    }

 }

 class Circle implements Shape {
    void draw() {
        // draw implementation

    }

 }
void drawShapes(List<Shape> shapes) {
    for (Shape shape : shapes) {
        shape.draw();

    }

}
```

```
void execute::Copy(ReadKeyboard&r, WritePrinter&wp, writeDisk&wd, OutputDevice
  dev){int c;
  while ((c=r.read())!=EOF)
    if (dev==printer)
      wp.write(c);
    else
      wd.write(c);
}
```

图 15-8　违反 OCP 的修改方案

违反OCP的方案




```
WriteDisk :: Write(c){
    WriteDisk(c);
}
```

图 15-9　符合 OCP 的多态方案

符合OCP的方案

OCP Summary

No significant program can be 100% closed
R.Martin, “The Open-Closed Principle,” 1996

- Use abstraction to gain explicit closure
- Plan your classes based on what is likely to change.
  - minimizes future change locations
- OCP needs DIP && LSP

DIP

Principle 12: Dependency Inversion Principle (DIP)

- I. High-level modules should not depend on low-level modules.
- Both should depend on abstractions.
- II. Abstractions should not depend on details.
- Details should depend on abstractions
- R. Martin, 1996

```
public class Client {
   …
    public static void main(string []
args){
        A a=new A(x, y);
        int result=a.getAddedValue();
        …
   }
}

public class A {
     private int x;
     private B b;
     A(int i, int j){
         x=i;
         b=new B(j);
     }
     public int getAddedValue(){
             return x+b.getY();
     }
     …
}

public class B {
     private int y;
     B(int i){
         y=i;
     }
     public int getY(){
             return y;
     }
     …
}
```

方案 1：A 依赖于 B

```
public class Client {
   …
    public static void main(string []
args){
        B b=new B(x, y);
        int result=b.getAddedValue();
        …
   }
}

public class B {
     private int y;
     private A a;
     A(int i, int j){
         y=j;
         a=new A(i);
     }
     public int getAddedValue(){
             return a.getX()+y;
     }
     …
}

public class A {
     private int x;
     A(int i){
         x=i;
     }
     public int getX(){
          return x;
     }
     …
}
```

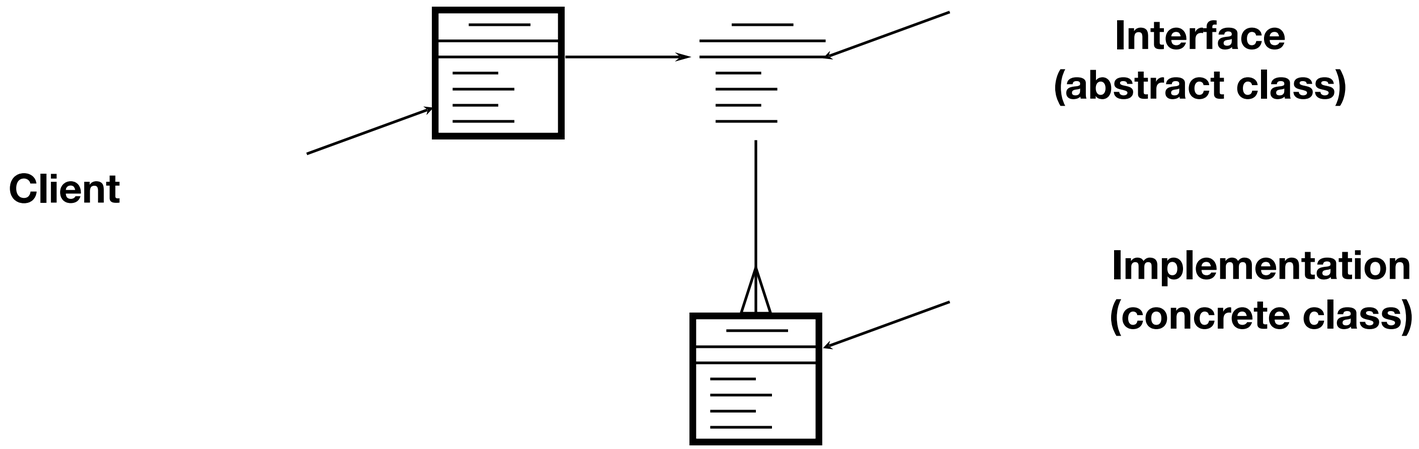
方案 2：B 依赖于 A

耦合的方向性

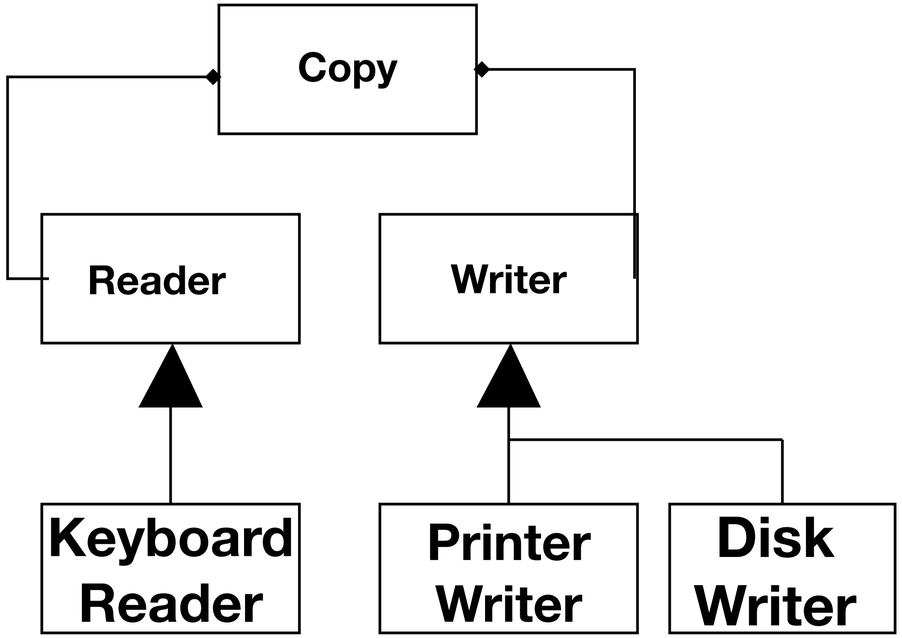
DIP : separate interface from implementation — — abstract

Design to an interface,
not an implementation!

- Use inheritance to avoid direct bindings to classes:



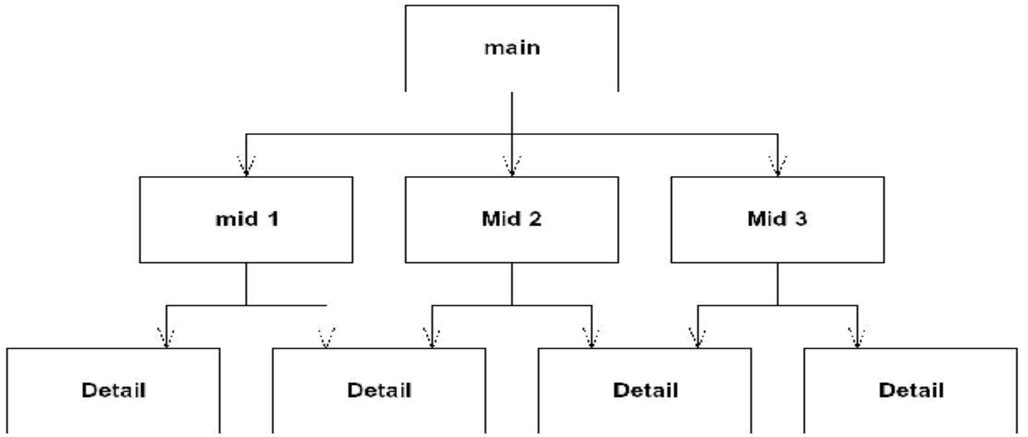
DIP Example



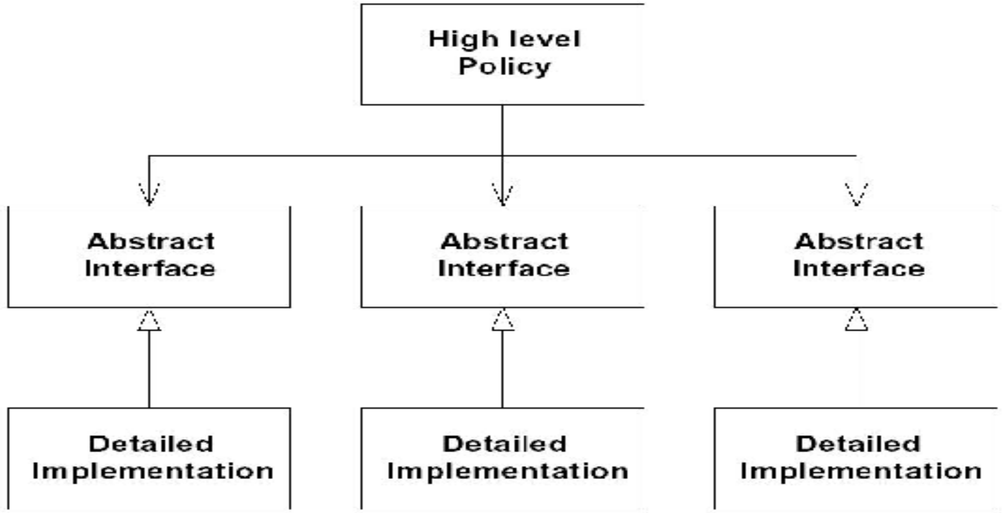

```
class Reader {
  public:
    virtual int read()=0;
};

class Writer {
  public:
    virtual void write(int)=0;
};

void Copy(Reader& r, Writer&
w){
   int c;
   while((c = r.read()) != EOF)
     w.write(c);
}
```

DIP Procedural vs. OO Architecture

Procedural Architecture



Object-Oriented Architecture



DIP summary

- Abstract classes/interfaces:
  - tend to change less frequently
  - abstractions are ‘hinge points’ where it is easier to extend/modify
  - shouldn’t have to modify classes/interfaces that represent the abstraction (OCP)

- Exceptions
  - Some classes are very unlikely to change;
    - therefore little benefit to inserting abstraction layer
    - Example: String class
  - In cases like this can use concrete class directly
    - as in Java or C++

How to deal with change

- OCP states the goal; DIP states the mechanism;
- LSP is the insurance for DIP

总结

Information Hiding: Design changes!

- the most common kind of secret is a design decision that you think might change.
- You then separate each design secret by assigning it to its own class, subroutine, or other design unit.
- Next you isolate (encapsulate) each secret so that if it does change, the change doesn’t affect the rest of the program.

Summary

- Principle 10: Minimize The Accessibility of Classes and Members
- Principle 11: Open/Closed Principle (OCP)
- Principle 12: Dependency Inversion Principle (DIP)