南京大学
NANJING UNIVERSITY

《软件工程与计算II》

Ch15　面向对象的信息隐藏

Encapsulation (2)

- **hide irrelevant details** from the user
- a class can be divided into two parts, the **user interface** and the **implementation**.
- 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.

New View of Encapsulation

- Old/Beginner/Implementation view of encapsulation: **hiding data inside an object**
- New view: **hiding anything**, including:
  - Data (implementation)
  - Structure (implementation)
  - Other object (implementation)
  - Type (derived classes)
  - Change/vary (design details)
  - …

Encapsulation Correctly —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
- 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.

Encapsulate Data

- 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;
}
```

Encapsulate structures

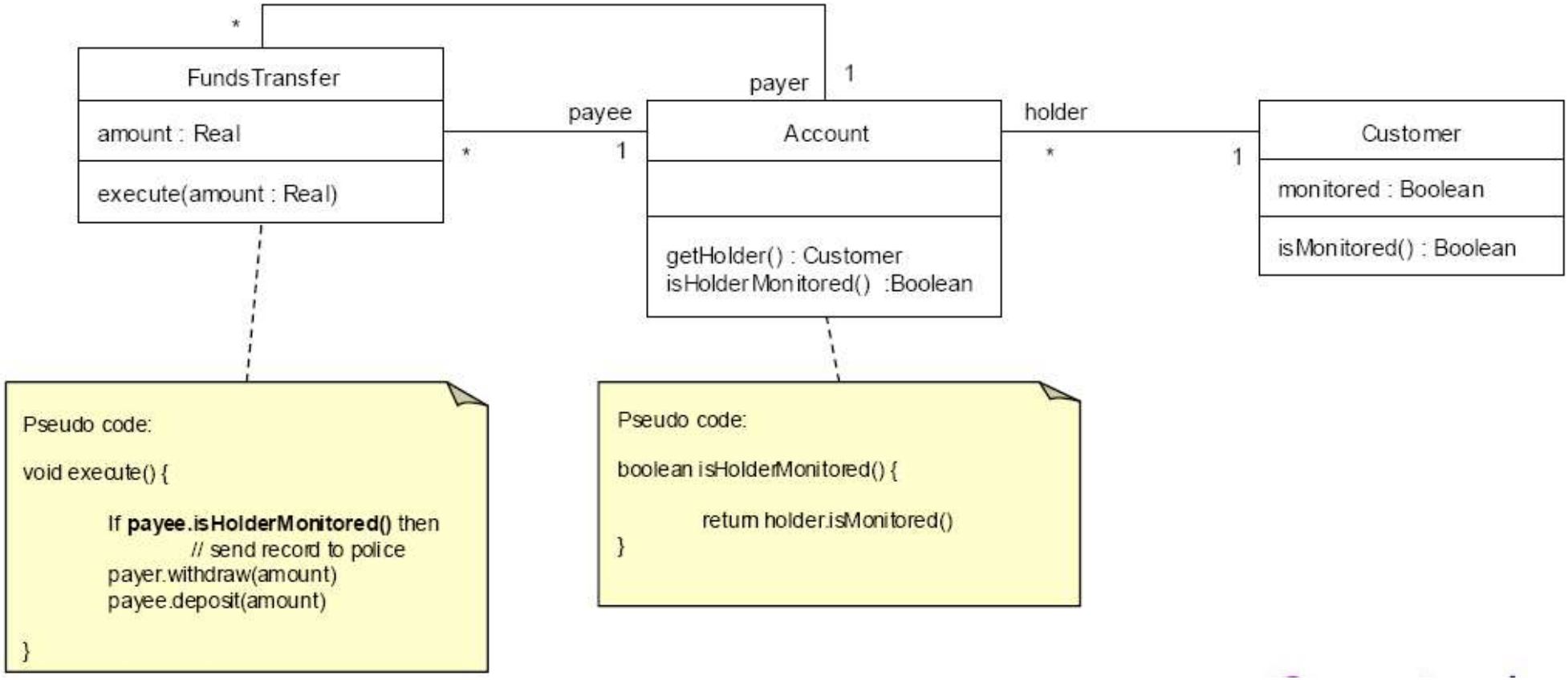
See chapter 16 Iterator Pattern

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

References and Collection Data-Type !

Encapsulate other objects

- Collaboration Design
  - Composition; delegation



Encapsulate type(subclass)

- LSP
  - pointers to superclasses or interfaces;

All derived classes must be
substitutable
for their base class

Encapsulate Change (or vary)

- Identify the aspects of your application that may change (or vary) and separate them from what stays the same.
- Take the parts that change(vary) and encapsulate them, so that later you can alter or extend the parts that vary without affecting the parts that don’t.

- See DIP and OCP Later

Encapsulate Implementation Detail

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

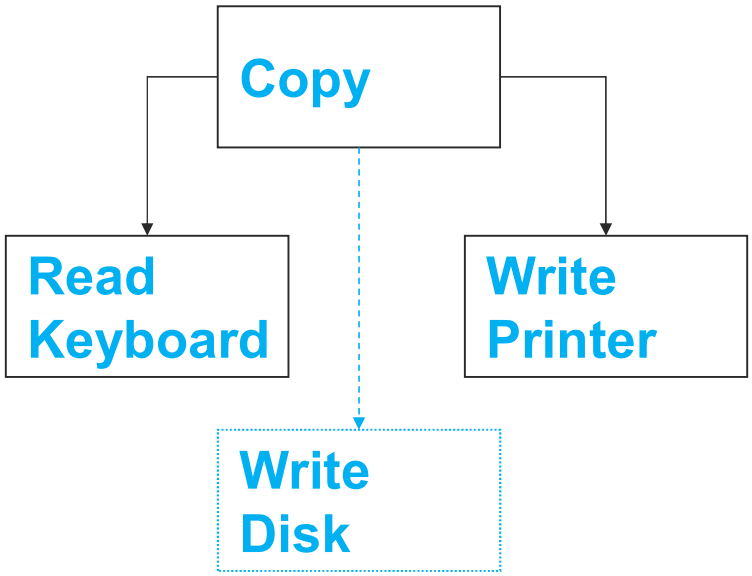
Principle #1: Minimize The Accessibility of Classes and Members

- 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

Main Contents

1. Encapsulation
2. How to hiding change?

Example of Responsibility Change



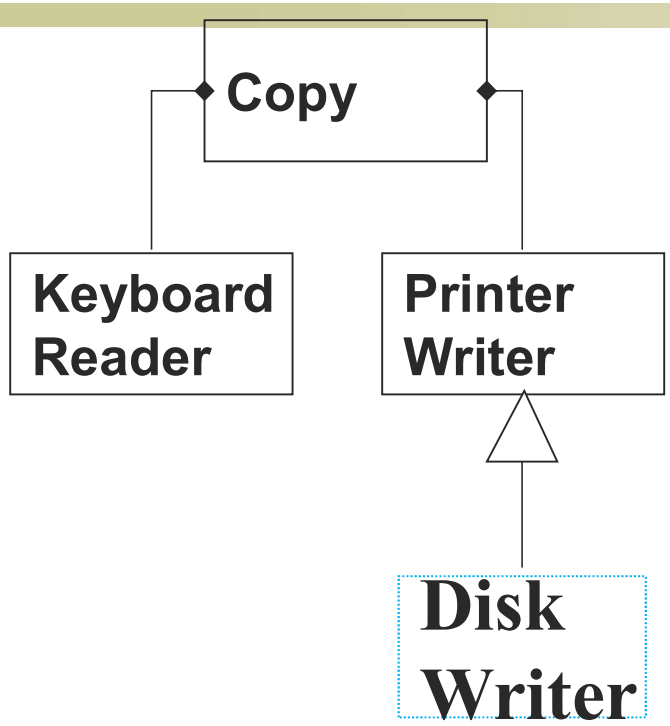

```
void Copy(ReadKeyboard& r,
WritePrinter& wp, WriteDisk&
wd, OutputDevice dev){
    int c;
    while((c = r.read())!= EOF)
        if(dev == printer)
            wp.write(c);
        else
            wd.write (c);
}
```

```
void Copy(ReadKeyboard& r, WritePrinter& w){
    int c;
    while ((c = r.read ()) != EOF)
        w.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 10: 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*

RTTI is Ugly and Dangerous!

- RTTI is ugly and dangerous
  - RTTI = **R**un-**T**ime **T**ype **I**nformation
  - 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();          }        }    }
```

// 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();        }        }
```

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

Principle 11: 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.

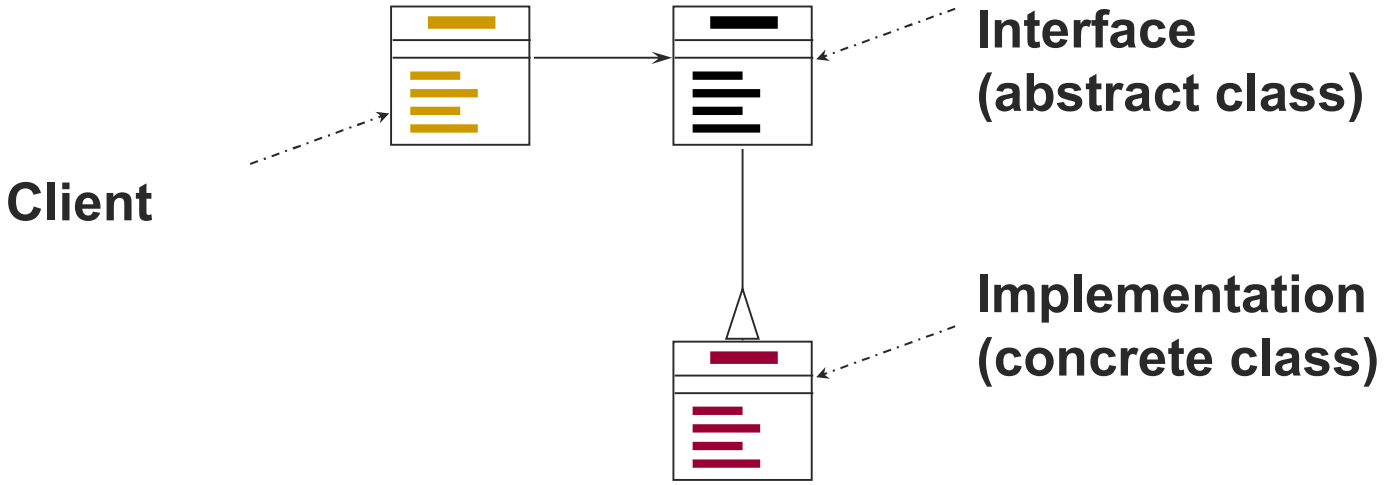
Details should depend on abstractions

R. Martin, 1996

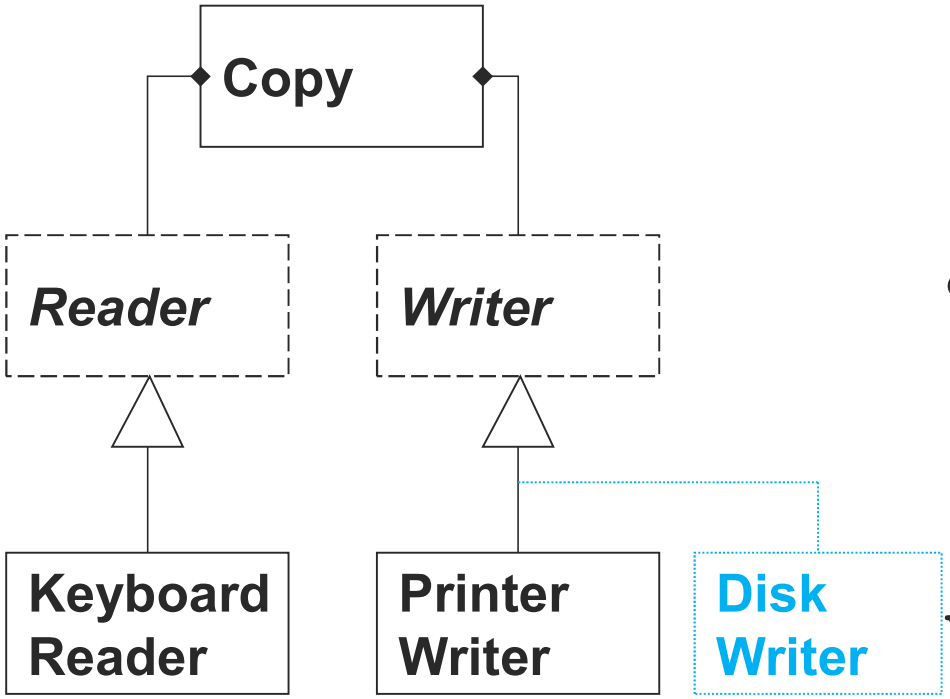
DIP : separate interface from implementation ——abstract

Design to an interface,
not an implementation!

Use inheritance to avoid direct bindings to classes:



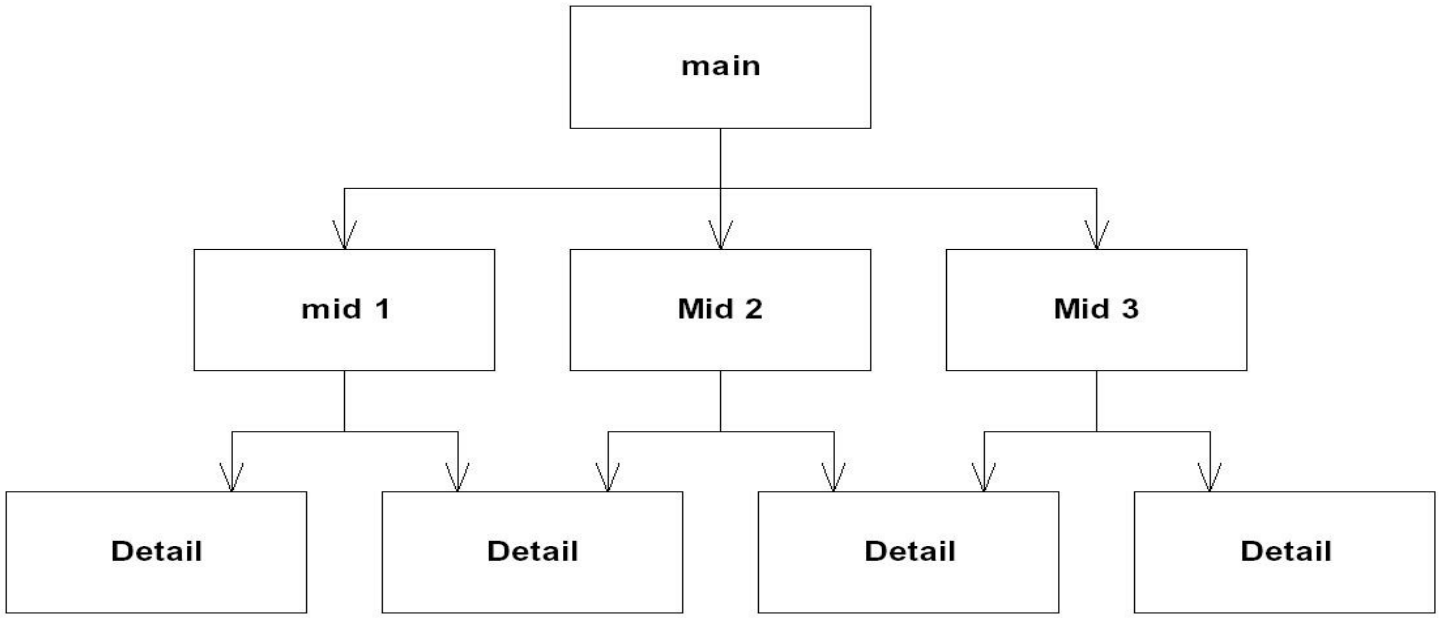
DIP Example



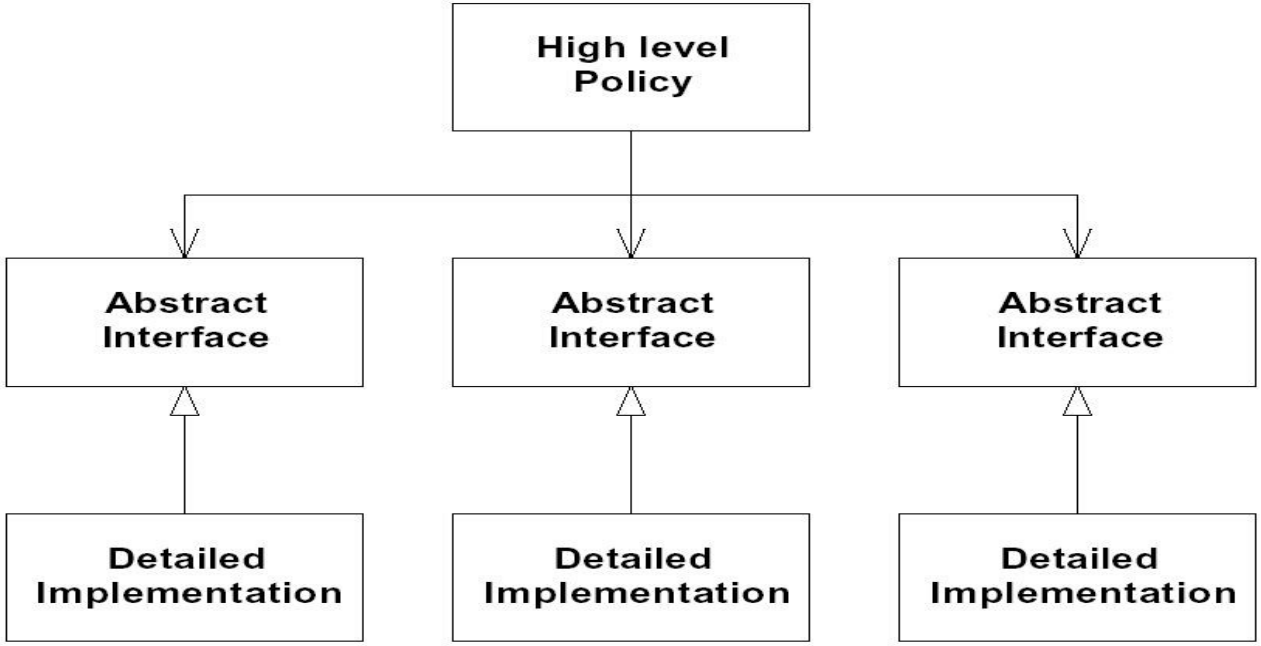

```
class Reader {
  public:
    virtual int read()=0;
};

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++

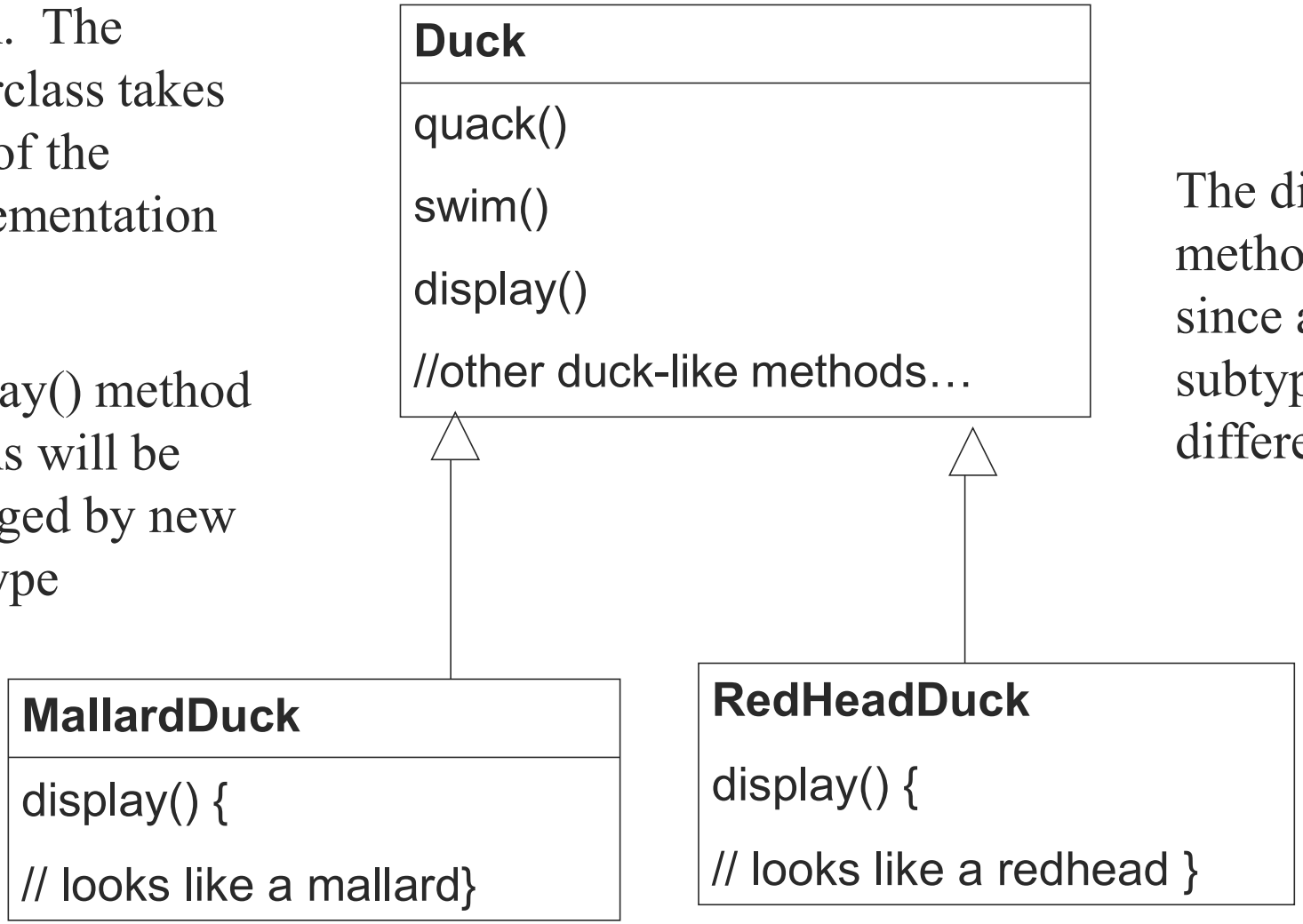
How to deal with change

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

Example of Implementation Change

All ducks quack and swim. The superclass takes care of the implementation code

display() method seems will be changed by new subtype



The display() method is abstract, since all duck subtypes look different

Single Responsibility Principle (SRP)

A class should have only one reason to change

Robert Martin

Duck
quack()
swim()
display()
//other duck-like methods…

SRS

and

display()

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.

See Chapter 16 Strategy Pattern