

# 4.2 Construction for Reuse

CMU 17-214: Oct 10、Oct 15、Oct 17、Oct 22

Java 编程思想：第 14、15 章

Effective Java：第 5 章

一级标题, 二级标题, 三级标题, 四级标题, 注, 拓展, 不懂

## 目录

4.2.1 the Designing reusable classes	2
Subtyping Polymorphism	2
Definition	2
Liskov Substitution Principle:	2
用处:	2
具体判断:	3
粗略介绍 Covariance and contravariance (协变与反协变)	4
Formal definition	4
Java 中数组是协变的:	4
Java 中的泛型是不变的	5
泛型容器中使用通配符来完成协变特性的匹配	5
拓展: 使用通配符的类型数据的类型	6
拓展: Producer Extends Consumer Super (java 中 PECS 原则)	6
类型擦除	6
Class (类型类)	6
methods for obtaining an object of type Class	7
泛型的类型擦除:	7
Delegation (委托)	8
Definition	8
分类	8
Delegation 和 Inheritance 的区别	9
Types 类型:	9
Composite over inheritance principle (CRP 原则)	11
Definition	11
CRP 原则的一个实现	11
优点:	14
缺点	15
4.2.2 Designing system-level reusable libraries and frameworks	15
拓展: 某些专有名字介绍	15
API 设计	16
一个好的 API 应该具有的特性:	16
设计原则	16
Framework design	16
Whitebox frameworks	16
Blackbox Frameworks	16

基本设计原则 .....	19
对于一个框架的运行 .....	20
Java Collections Framework (举例, 但我不理解) .....	20

## 4.2.1 the Designing reusable classes

- A. Encapsulation and information hiding 封装与信息隐藏
- B. Inheritance and overriding 继承与重写
- C. Polymorphism, subtyping and overloading 多态, 子类型, 重载
- D. Generic programming 泛型编程
- E. Behavioral subtyping and Liskov Substitution Principle (LSP)
- F. Delegation and Composition

**注:** A, B, C, D 可以想见 3-4 关于对象编程的部分。在某种程度上来说, 设计一个复用类也会使用设计普通类的技术, 下面将主要探讨 E 和 F。

### Subtyping Polymorphism

#### Definition

If S is a subtype of T, the subtyping relation is often written  $S <: T$ , to mean that any term of type S can be safely used in a context where a term of type T is expected.

Subtyping 是一种关系: S 是 T 的子类型, 如果任何使用 T 的程序中的 T 均可以被 S 替代。

另一种解释就是 Liskov Substitution Principle。

#### Liskov Substitution Principle:

Let  $q(x)$  be a property provable about objects  $x$  of type T, then  $q(y)$  should be provable for objects  $y$  of type S where S is a subtype of T. (如果对于类型 T 的对象  $x$ ,  $q(x)$  成立, 那么对于类型 T 的子类型 S 的对象  $y$ ,  $q(y)$  也成立。)

#### 用处:

Subtype 可以将 supertype 进行代替, 此时的 Subtype 不同于 supertype, 其中包含有其他的功能。

## 具体判断：

Compiler-enforced rules (编译器强制规则)：

- Subtypes can add, but not remove methods 子类型可以增加方法，但不可删
- Concrete class must implement all undefined methods 子类型需要实现抽象类型 (接口、抽象类)中所有未实现的方法
- Overriding method must return same type or subtype 子类型中重写的方法必须有相同或子类型的返回值或者符合co-variant的参数
- Overriding method must accept the same parameter types 子类型中重写的方法必须使用同样类型的参数或者符合contra-variant的参数(此种情况Java目前按照重载overload处理)
- Overriding method may not throw additional exceptions 子类型中重写的方法不能抛出额外的异常


人为判断角度为：

- **Preconditions** cannot be strengthened in a subtype. 前置条件不能强化
- **Postconditions** cannot be weakened in a subtype. 后置条件不能弱化
- **Invariants** of the supertype must be preserved in a subtype. 不变量要保持
- **Contravariance** of method arguments in a subtype 子类型方法参数：逆变
- **Covariance** of return types in a subtype. 子类型方法的返回值：协变
- No new **exceptions** should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype. 异常类型：协变 (This is to be discussed in Section 7-2)

**注：**在上述判断条件中的 Covariance (协变)与 Contravariance (反协变、逆变)指的是：子类型方法返回值的类型或者抛出异常的类型可以是原类型的协变类型 (更加准确的讲：是返回类型与原类型的关系可以是协变关系)，同理，子类型方法的参数类型可以是原类型的参数类型。

Ex: 子类型的返回值以及异常的类型

```
class T {  
    Object a() { ... }  
}  
  
class S extends T {  
    @Override  
    String a() { ... }  
}
```



```
class T {  
    void b( ) throws Throwable {...}  
}  
  
class S extends T {  
    @Override  
    void b( ) throws IOException {...}  
}  
  
class U extends S {  
    @Override  
    void b( ) {...}  
}
```


Ex: 子类型的参数

```

class T {
    void c( String s ) { ... }
}

class S extends T {
    @Override
    void c( Object s ) { ... }
}

```



**注：**在 java 中是不允许这种情况（即覆盖中参数发生变化）出现的，这种逆变被认为是方法重载。

## 粗略介绍 Covariance and contravariance (协变与反协变)

### Formal definition

Many programming language type systems support subtyping. Variance refers to how subtyping between more complex types relates to subtyping between their components.

个人认为是这个意思：逆变和协变描述了具有继承关系的类型，通过类型构造器映射到另一范畴时所具有的继承关系。通俗来说，就是一组具有子类型关系的类型被构造到另一组类型时，这两组类型之间的关系。

Within the type system of a programming language, a typing rule or a type constructor is:

- A. covariant if it preserves the ordering of types ( $\leq$ ), which orders types from more specific to more generic; (如果被构造的类型保持了原有类型的子类型序关系，则称为协变)
- B. contravariant if it reverses this ordering; (如果被构造的类型逆转了原有类型的子类型序关系，则称为逆变)
- C. bivariant if both of these apply (i.e., both  $I<A> \leq I<B>$  and  $I<B> \leq I<A>$  at the same time); (既是协变，又是逆变，称为双变)
- D. variant if covariant, contravariant or bivariant. (协变，逆变，双变都是变)
- E. invariant or nonvariant if not variant. (非变就是不变)

### Java 中数组是协变的:

对于 T 类型的数组，其中可以包含类型 T 的子类型的数据。

总的来讲，java 将数组成协变是存在某种缺陷的，是在没有泛型出现时进行的妥协。这里利用了数组的性质：一个数组会保留其内元素的类型，并且会在程序运行时对其进行检查。但是与之相反，泛型容器是不会检查的。

其中是可以保存其类型（new 类型）及其子类型的数据，这里是保证利用了上述所讲子类型多态性（替换性）。

具体可以参见 <https://www.zhihu.com/question/21394322>

## Java 中的泛型是不变的

对于泛型而言，其包装后的类型（容器类型）与原类型不具有协变关系，其是不变的，即：invariant。

Ex: `List<String>` is not a subtype of `List<Object>`。

原因是该类型参数信息在编译后会被丢弃，在运行时不可用。这个丢弃过程也被称为 type erasure（类型擦除），所以直接在编译时就不会通过

Ex: 虽然 `integer` 和 `number` 之间是子类型关系，但是其包装后的容器类型是不具有协变关系的。

```
List<Integer> myInts = new ArrayList<>();
myInts.add(1);
myInts.add(2);
List<Number> myNums = myInts; //compiler error
myNums.add(3.14); //如果编译通过，此处操作会成功，不安全
```

```
static long sum(List<Number> numbers) {
    long summation = 0;
    for(Number number : numbers) {
        summation += number.longValue();
    }
    return summation;
}
```

We cannot consider a list of integers to be subtype of a list of numbers.

That would be considered unsafe for the type system and compiler rejects it immediately.

```
List<Integer> myInts = Arrays.asList(1,2,3,4,5);
List<Long> myLongs = Arrays.asList(1L, 2L, 3L, 4L, 5L);
List<Double> myDoubles = Arrays.asList(1.0, 2.0, 3.0, 4.0, 5.0);
sum(myInts); //compiler error
sum(myLongs); //compiler error
sum(myDoubles); //compiler error
```

## 泛型容器中使用通配符来完成协变特性的匹配

### A. 无限定通配符

应用于该方法的实现不依赖于类型参数或者只依赖 `Object` 类中的功能。

Ex:

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

The goal of `printList` is to print a list of any type, but it fails to achieve that goal — it prints only a list of `Object` instances; it cannot print `List<Integer>`, `List<String>`, `List<Double>`, and so on, because they are not subtypes of `List<Object>`. 本意期望可打印任意类型的 `List`，但由于泛型不协变，只能打印 `List<Object>`

To write a generic `printList` method, use `List<?>`

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}

List<Integer> li = Arrays.asList(1, 2, 3);
List<String> ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

### B. Lower Bounded Wildcards: `<? super A>` 下限通配符

在此函数接口可以认为由 `A` 的超类型或者其本身包装的容器类型可以是该类型的子类型。

### C. Upper Bounded Wildcards: `<? extends A>` 上限通配符

在此函数接口可以认为由 A 的子类型或者其本身包装的容器类型可以是该类型的子类型。

**注：**这里表明 A 是上限类型或者下限类型

## 拓展：使用通配符的类型数据的类型

这里虽然完成了关于 list 的协变性的某些拓展，但是它会产生一个问题：是否可以向由 List<? extend A> 类型定义的对象之中添加 A 的子类型的数据，因为 list 不会保留其初始化的类型，只会记录其签名，所以无法判断添加的类型是否满足初始化类型的要求，因而编译器拒绝了 A 的子类型的数据的添加。

比如说，cat extend animals, whitecat extend cat, blackcat extend cat。

List<? extend cat> list = new ArrayList<cat>();初始化了一个 list，这里使用了通配符给 list 带来的协变。但是向其中添加 new whiteCat(), new Cat(), new Animal()都会报错，因为对于 list 而言，只能添加 cat 类型及其子类型，但是编译器并不会记住 list 的初始化类型是 cat，只会记住 List<? extend cat>，而对于这个通配符定义的类型而言，其有可能以 whitecat 初始化，那么添加 cat, blackcat 就是不合适的。所以编译器会拒绝这样的请求。

所以在 List<? extend A>定义的对象中不会保存 A 的子类型的数据。

那么与之相反，List<? super A>定义的对象自然是可以保存 A 的子类型数据的啦。

## 拓展：Producer Extends Consumer Super (java 中 PECS 原则)

在一个函数的参数中，Producer 使用 List<? extend A>，而 consumer 使用 super。

**原因：**

由“使用通配符的类型数据的类型”可以拓展得到无论 List<? extend A>的具体实现是什么，它的返回类型至少是 A 类型（下限），而无论 List<? super A>的具体实现是什么，可以向内保存 A 的子类型数据（上限）。

利用这一点，对于一个 Producer 而言，只向外提取数据，故而使用 List<? extend A>类型可以确定提取类型，对于 Consumer 而言，知识向内加入类型，那么，可以确定加入类型

## 类型擦除

## Class（类型类）

当程序在运行时，java 运行时系统会为所有对象维护一个运行时类型标识，这个标识会记录该对象所属的类，这个表示也被称为 Class 类，有些争议，且这个类名有点易混淆。

## methods for obtaining an object of type Class

```
//method1
Employee e;
...
Class c1 = e.getClass();

//method2
String className = "java.util.Random";
Class c1 = Class.forName(className);

//method3
Class c11 = Random.class;
Class c12 = int.class;
Class c13 = Double[].class;
```

## 泛型的类型擦除：

Java 的泛型基本上都是在编译器这个层次上实现的，在生成的字节码中是不包含泛型中的类型信息的，使用泛型的时候加上类型参数，在编译器编译的时候会去掉，替换为原始泛型，这个过程成为类型擦除。

在定义泛型对象时，都会定义一个符合的原始类型。

原始类型就是擦除了泛型信息，最后在字节码中的类型变量的真正类型。

该原始类型就是该去掉类型参数后代替的类型。当类型参数无限定类型时，类型参数会被替换为 object，否则会被替换为限定类型的第一个类型。

Ex:

未指定类型，将原始类型指定为 object。

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```



```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data,
                Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() {
        return data; }
    // ...
}
```

指定了多个类型，使用第一个类型



```
public class Interval<T extends Comparable & Serializable> implements
Serializable
{
    private T lower;
    private T upper;
    . . .
    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; } //此处因为限定了T的具体类型，
                                                所以可以调用类型相关的具体方法
    }
}
```

```
public class Interval implements Serializable
{
    private Comparable lower;
    private Comparable upper;
    . . .
    public Interval(Comparable first, Comparable second) { . . . }
}
```

**注：**关于类型擦除拓展可以参见 <https://www.cnblogs.com/wuqinglong/p/9456193.html#1-2%E9%80%9A%E8%BF%87%E4%B8%A4%E4%B8%AA%E4%BE%8B%E5%AD%90%E8%AF%81%E6%98%8Eja va%E7%B1%BB%E5%9E%8B%E7%9A%84%E7%B1%BB%E5%9E%8B%E6%93%A6%E9%99%A4>

**注：**对于泛型而言，我们必须明确不可使用 instanceof 运算符，会产生编译错误，因为泛型在运行时会擦除其类型，故而

```
if (a instanceof Pair<String>) // compile error
if (a instanceof Pair<T>) // compile error
```

会产生 Pair<String>和 Pair<T>相同，二者互为子类型，而这明显是不对的，故而产生编译错误。

## Delegation (委托)

### Definition

Delegation is simply when one object relies on another object for some subset of its functionality (one entity passing something to another entity)

一个对象的函数实现依赖于另一个的对象的函数。

Delegation can be described as a low level mechanism for sharing code and data between entities. (继承可以被认为是一个在两个实体之间分享代码和数据) Judicious delegation enables code reuse (继承是复用的一种常见形式)

### 分类

Explicit delegation (显式继承) : passing the sending object to the receiving object

Implicit delegation (隐式继承) : by the member lookup rules of the language

**Ex：**一个隐式继承的例子



```

class A {
    void foo() {
        this.bar();
    }

    void bar() {
        print("a.bar");
    }
}

class B {
    private A a; // delegation link

    public B(A a) {
        this.a = a;
    }

    void foo() {
        a.foo(); // call foo() on the a-instance
    }

    void bar() {
        print("b.bar");
    }
}

A a = new A();
B b = new B(a); // establish delegation between two objects
b.foo();

```

Ex: 使用继承可以进行功能扩展，下面利用隐式委托的形式，并且定义了一个 list 类功能扩展的 logginglist 类

- Consider java.util.List

我需要一个能log的List

```

public interface List<E> {
    public boolean add(E e);
    public E remove(int index);
    public void clear();
    ...
}

```

- Suppose we want a list that logs its operations to the console...

- The LoggingList is composed of a List, and delegates (the non-logging) functionality to that List.

```

public class LoggingList<E> implements List<E> {
    private final List<E> list;
    public LoggingList<E>(List<E> list) { this.list = list; }
    public boolean add(E e) {
        System.out.println("Adding " + e);
        return list.add(e);
    }
    public E remove(int index) {
        System.out.println("Removing at " + index);
        return list.remove(index);
    }
}

```

这里实现动态绑定

这里进行功能委派

## Delegation 和 Inheritance 的区别

委托关系可以多委托，继承关系在 Java 中只有单继承。

继承具有比委托更强的耦合关系。

当超类被修改时，从超类继承的子类必须进行重新编译，但是委托于超类的类不需要进行重新编译。

behaviors are 继承得到的子类中可能会存在大量无用的方法。

## Types 类型:

Dependency (临时的委托)

A 将其功能委托于 B, 将 B 作为 A 中实现该功能方法的参数传入, 并且将 A 并未将 B 作为 A 的某个成员。

至于临时, 则是当且仅当 A 调用该方法时, 二者才会具备委托关系, 在某种程度上, 等效于隐形委托。

Ex:

```
Flyable f = new FlyWithWings();    class Duck {
Quackable q = new Quack();          //no field to keep Flyable object

Duck d = new Duck();               void fly(Flyable f) {
d.fly(f);                          f.fly();
d.quack(q);                        }
}
```

Association: 永久性的 delegation

A 将其功能委托于 B, 并将 B 作为 A 的一个成员。

至于永久, 因为 B 已经是该成员的一个成员变量, 无论 A 的行为如何, 其都会有委托关系。

Ex:

```
Flyable f = new FlyWithWings();    class Duck {
Duck d = new Duck(f);              Flyable f = new CannotFly();
Duck d2 = new Duck();              void Duck(Flyable f) {
d.fly();                          this.f = f;
                                }
                                void Duck() {
                                f = new FlyWithWings();
                                }
                                void fly() { f.fly(); }
                                }
```

Composition: 更强的 association, 但难以变化

此类型属于 association, 但其中的委托对象为自动生成, 难以变化。

Ex:

```
Duck d = new Duck();               class Duck {
d.fly();                          Flyable f = new FlyWithWings();

                                void fly() {
                                f.fly();
                                }
                                }
```

Aggregation: 更弱的 association, 可动态变化

此类型属于 association, 但其中的委托对象为外部传入, 难以变化。

Ex:

```

Flyable f = new FlyWithWings();
Duck d = new Duck(f);
d.fly();
d.setFlyBehavior(new CannotFly());
d.fly();

class Duck {
    Flyable f;

    void Duck(Flyable f) {
        this.f = f;
    }
    void setFlyBehavior(f) {
        this.f = f;
    }
    void fly() { f.fly(); }
}

```

## Composite over inheritance principle (CRP 原则)

### Definition

Classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement desired functionality) rather than inheritance from a base or parent class.

一个类为了实现多态行为或者代码重用, 尽量使用类的组合而不是由继承产生的子类型关系。

### CRP 原则的一个实现

可以在需要使用某个功能的类中的定义某个接口, 之后实现这个接口的多个类, 利用这个接口和实现类的子类型关系, 进而实现多态性为, 这种方式可以认为是多个实现类的组合, 而不是具有继承关系。

**注:**从这个角度来看, CRP 原则确实是委托, 因为需要实现某个功能的类将该功能委托给了这个功能的接口的多个实现类。

**Ex:** 一个例子来源于 [https://en.wikipedia.org/wiki/Composition\\_over\\_inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance)

```

#include <vector>

class Model;

class GameObject {
public:
    virtual ~GameObject() = default;

    virtual void Update() {}
    virtual void Draw() {}
    virtual void Collide(const std::vector<GameObject*>& objects) {}
};

class Visible : public GameObject {
public:
    void Draw() override {
        // Draw /model_/ at position of this object.
    };

private:
    Model* model_;
};

class Solid : public GameObject {
public:
    void Collide(const std::vector<GameObject*>& objects) override {
        // Check and react to collisions with /objects/.
    };
};

class Movable : public GameObject {
public:
    void Update() override {
        // Update position.
    };
};

```

之后进行了多重继承。(C++允许多继承)

Then, we have concrete classes:

- class `Player` - which is `Solid`, `Movable` and `Visible`
- class `Cloud` - which is `Movable` and `Visible`, but not `Solid`
- class `Building` - which is `Solid` and `Visible`, but not `Movable`
- class `Trap` - which is `Solid`, but neither `Visible` nor `Movable`

采用 CRP 实现的：

```

class Program {
    static void Main() {
        var player = new GameObject(new Visible(), new Movable(), new Solid());
        player.Update(); player.Collide(); player.Draw();

        var cloud = new GameObject(new Visible(), new Movable(), new NotSolid());
        cloud.Update(); cloud.Collide(); cloud.Draw();

        var building = new GameObject(new Visible(), new NotMovable(), new Solid());
        building.Update(); building.Collide(); building.Draw();

        var trap = new GameObject(new Invisible(), new NotMovable(), new Solid());
        trap.Update(); trap.Collide(); trap.Draw();
    }
}

interface IVisible {
    void Draw();
}

class Invisible : IVisible {
    public void Draw() {
        Console.WriteLine("I won't appear.");
    }
}

```

```

class Visible : IVisible {
    public void Draw() {
        Console.WriteLine("I'm showing myself.");
    }
}

interface ICollidable {
    void Collide();
}

class Solid : ICollidable {
    public void Collide() {
        Console.WriteLine("Bang!");
    }
}

class NotSolid : ICollidable {
    public void Collide() {
        Console.WriteLine("Splash!");
    }
}

interface IUpdatable {
    void Update();
}

class Movable : IUpdatable {
    public void Update() {
        Console.WriteLine("Moving forward.");
    }
}

```

```

class NotMovable : IUpdatable {
    public void Update() {
        Console.WriteLine("I'm staying put.");
    }
}

class GameObject : IVisible, IUpdatable, ICollidable {
    private readonly IVisible _visible;
    private readonly IUpdatable _updatable;
    private readonly ICollidable _collidable;

    public GameObject(IVisible visible, IUpdatable updatable, ICollidable collidable) {
        _visible = visible;
        _updatable = updatable;
        _collidable = collidable;
    }

    public void Update() {
        _updatable.Update();
    }

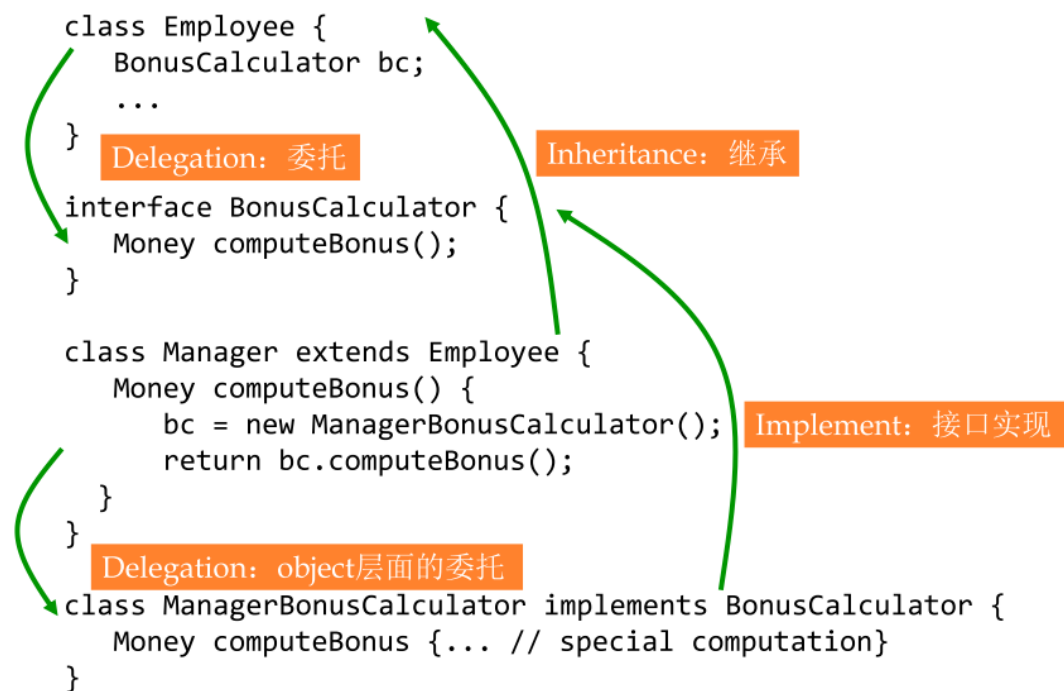
    public void Draw() {
        _visible.Draw();
    }

    public void Collide() {
        _collidable.Collide();
    }
}

```

Ex: 另一个例子

在这个例子，需要明白的是，它仍然遵循上述的实现，我们的重点是计算佣金的实现，其使用的就是 CRP 原则，而不是 employer，employer 与 manager 除去计算佣金的方法之外，二者具有清晰的继承关系，故而可以采用这种继承模式，但计算佣金则继承关系不明确，很难得到清晰的关系树。



## 优点：

可以赋予设计者更强的灵活性，可以更加灵活的构建多个方法，而继承时，需要考虑多个方法之间的继承关系，这正是在某种程度上弥补继承的不足。这是采用组合最为根本的原

因。

另外它也不用考虑采用继承的方法时，所产生的继承的无用方法的干扰。

## 缺点

在维基百科上，其认为缺点时可能会产生大量的重复，因为其不具有继承关系，无法利用继承所得到的代码的重用。

## 4.2.2 Designing system-level reusable libraries and frameworks

之所以 library 和 framework 被称为系统层面的复用，是因为它们不仅定义了 1 个可复用的接口/类，而是将某个完整系统中的所有可复用的接口/类都实现出来，并且定义了这些类之间的交互关系，调用关系，从而形成了系统整体性架构。

### Libraries and frameworks in practice:

- A. Defines key abstractions and their interfaces
- B. Defines object interactions & invariants
- C. Defines flow of control
- D. Provides architectural guidance
- E. Provides defaults

## 拓展：某些专有名字介绍

- A. API: Application Programming Interface, the interface of a library or framework (调用接口)
- B. Client: The code that uses an API
- C. Plugin: Client code that customizes a framework
- D. Extension point: A place where a framework supports extension with a plugin (框架内预留的“空白”，开发者开发出符合接口要求的 代码(即 plugin)，框架可调用，从而相当于开发者扩展了框架的功能)
- E. Protocol: The expected sequence of interactions between the API and the client (用户端代码与接口交互的协议)
- F. Callback: A plugin method that the framework will call to access customized functionality (一个方法用于调用用户端代码)
- G. Lifecycle method: A callback method that gets called in a sequence according to the protocol and the state of the plugin



## API 设计

API 设计具有一旦发布就无法修改的特性

### 一个好的 API 应该具有的特性：

- A. Easy to learn
- B. Easy to use, even without documentation
- C. Hard to misuse
- D. Easy to read and maintain code that uses it
- E. Sufficiently powerful to satisfy requirements (足够强大, 可以满足需求)
- F. Easy to evolve 易于拓展

## 设计原则

这一部分进行照搬, 没有讲解, 个人理解难, 感觉个人应该不会使用, 当使用时, 将其翻译  
详见 PPT 79 页

## Framework design

框架 (framework) 是已经定义了各类方法的系统, 该系统可以调用用户使用的代码, 进行个性化定制, 快速的完成应用程序的编写。

### Whitebox frameworks

使用继承来实现用户代码 Plugin 的绑定。  
其中运行的框架代码来自于子类, 控制的权限在于框架。  
其设计模式常为: Template Method

### Blackbox Frameworks

使用委托来实现客户代码的绑定。  
其中运行的框架代码来自于基类, 控制的权限在于框架。  
其设计模式常为: Strategy, Observer  
通常可以实现多模块化。

**注:** 白盒框架的名称来源于: 使用框架者必须了解框架的内部实现原理。而黑盒框架只需要明白接口就可以了。

**注:** 在 java 中, 对于白盒框架而言, 其所有的特定实现方法只可以在一个子类当中完成, 因为 java 不支持多继承。黑盒框架可以将其特定实现方法委托给多个函数。

**注：**白盒框架必须一起编译，而黑盒框架可以进行单独部署

**Ex：**一个计算器的图形化程序的完整代码编写

```
public class Calc extends JFrame {
    private JTextField textField;
    public Calc() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText("calculate");
        contentPane.add(button, BorderLayout.EAST);
        textField = new JTextField("");
        textField.setText("10 / 2 + 6");
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        button.addActionListener(/* calculation code */);
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle("My Great Calculator");
        ...
    }
}
```

将其以白盒框架进行架构为：可以观察到，其将该类程序共性的进行总结，异性的将之提取出来构成一些新方法，当需要产生不同的程序时，可以继承该类，在子类可以对超类中异性的方法进行覆盖，这样可以简单的得到需要的程序。

```
public abstract class Application extends JFrame {
    protected String getApplicationTitle() { return ""; }
    protected String getButtonText() { return ""; }
    protected String getInitialText() { return ""; }
    protected void buttonClicked() { }
    private JTextField textField;
    public Application() {
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(getButtonText());
        contentPane.add(button, BorderLayout.EAST);
        textField = new JTextField("");
        textField.setText(getInitialText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        button.addActionListener((e) -> { buttonClicked(); });
        this.setContentPane(contentPane);
        this.pack();
        this.setLocation(100, 100);
        this.setTitle(getApplicationTitle());
        ...
    }
}
```

子类中由开发者通过  
override完成  
定制的功能

```

public class Calculator extends Application {
    protected String getApplicationTitle() { return "My Great Calculator"; }
    protected String getButtonText() { return "calculate"; }
    protected String getInititalText() { return "(10 - 3) * 6"; }
    protected void buttonClicked() {
        JOptionPane.showMessageDialog(this, "The result of " + getInput() +
            " is " + calculate(getInput()));
    }
    private String calculate(String text) { ... }
}

```

Overriding

Extension via subclassing and overriding methods  
Subclass has main method but gives control to framework

```

public class Ping extends Application {
    protected String getApplicationTitle() { return "Ping"; }
    protected String getButtonText() { return "ping"; }
    protected String getInititalText() { return "127.0.0.1"; }
    protected void buttonClicked() { ... }
}

```

Overriding

将其以黑盒框架的形式进行架构时，将共性保留，异性构成方法，并将这些方法统一成为接口，需要实现该类时，只需要实现接口，并将实现类为参数构造该框架运行即可。

```

public class Application extends JFrame {
    private JTextField textField;
    private Plugin plugin;
    public Application() { }
    protected void init(Plugin p) {
        p.setApplication(this);
        this.plugin = p;
        JPanel contentPane = new JPanel(new BorderLayout());
        contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
        JButton button = new JButton();
        button.setText(plugin != null ? plugin.getButtonText() : "ok");
        contentPane.add(button, BorderLayout.EAST);
        textField = new JTextField("");
        if (plugin != null)
            textField.setText(plugin.getInititalText());
        textField.setPreferredSize(new Dimension(200, 20));
        contentPane.add(textField, BorderLayout.WEST);
        if (plugin != null)
            button.addActionListener((e) -> { plugin.buttonClicked(); });
        this.setContentPane(contentPane);
        ...
    }
    public String getInput() { return textField.getText(); }
}

```

```

public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked();
    void setApplication(Application app);
}

```

Ex: 另一个例子

```
public abstract class PrintOnScreen {
```

```
    public void print() {
        JFrame frame = new JFrame();
        JOptionPane.showMessageDialog(frame, textToShow());
        frame.dispose();
    }
```

Main body of  
framework  
REUSE

```
    protected abstract String textToShow();
}
```

Extension  
point

```
public class MyApplication extends PrintOnScreen {
```

```
    @Override
    protected String textToShow() {
        return "printing this text on "
            + "screen using PrintOnScreen "
            + "white Box Framework";
    }
}
```

Overriding

```
MyApplication m = new MyApplication();
m.print();
```

```
public final class PrintOnScreen {
    TextToShow textToShow;
    public PrintOnScreen(TextToShow tx) {
        this.textToShow = tx;
    }
```

Extension  
point by  
composition

```
    public void print() {
        JFrame frame = new JFrame();
        JOptionPane.showMessageDialog(frame,
            textToShow.text());
        frame.dispose();
    }
}
```

```
public interface TextToShow {
    String text();
}
```

```
public class MyTextToShow
    implements TextToShow {
```

```
    @Override
    public String text() {
        return "Printing";
    }
}
```

Plugin

```
PrintOnScreen m =
    new PrintOnScreen(new MyTextToShow());
m.print();
```

## 基本设计原则

- 一旦完成，很难进行修改，所以尽量一次完成。
- 确定共性以及异性，异性的行为太多，会造成开发较慢以及复用程度不高，异性的行为太少，会造成框架的使用范围很小，该框架的价值会减小。最大化的复用会带来最小范围的使用，这个平衡度很难进行掌握。
- 确定该框架的使用的领域，不同领域的框架的结构是不同的。
- 在构造一个框架时，越抽象的东西越早实现，抽象程度：接口>抽象类>具体方法

## 对于一个框架的运行

一些框架本身就可以自己运行，例如：eclipse；有些框架必须将扩展点补充完成才可以进行运行，例如 swing，junit 等等

有些框架会自己写 main 函数，之后将 plugin 作为命令行参数传入，有些框架为客户自己构造 main 函数，还有一些框架在特定位置寻找 config 文件或者。Jar 包，进行自动的装载于运行。

## Java Collections Framework（举例，但我不理解）

详见 PPT108