Advance Java Presentation

第九組
407262079 資工二乙 李承翰
407262110 資工二乙 王麗婷
407262433 資工二乙 王一鑫
407262354 資工二乙 林鈺恩

Rotting Design

Symptoms of Rotting Design

Rigidity

程式一旦被改動，即使是很小的改動，也會造成一連串的程式跟著被影響。然而若是太怕造成這種效應，而拒絕改動，則會造成使用者不便，使用者一旦使用了這種程式，接下來只能繼續使用，要捨棄則得整個捨棄，這樣的結果不符合效益，故這是個糟糕的設計。

Symptoms of Rotting Design

Fragility

如上面所說的，程式改動往往容易造成一些損壞，而為了修復這些錯誤時，又是在做改動，因此又有可能造成了一些損壞，因此只會有越修復越糟糕的情況。然而經常性的發生這些損壞時，會導致使用者對這套軟件的不信任。

Symptoms of Rotting Design

Immobility

當我們在撰寫程式時，經常會遇到自己想撰寫的模組和另一位工程師已寫完的模組很相似，這時候我們就會想要reuse這個模組。但是此時我們卻發現這個模組所依賴的baggage太多(baggage 可能是當時工程師在撰寫這個模組時所寫的一些相關模組，這些模組之間可能有繼承關係，所以要使用此模組可能又要追溯到他的父類等等，導致我們需要花很多時間去去辨別哪些部分的軟件是我們想要的、哪些是我們不要的，而在做這個分離的動作時很有可能導致很多錯誤及不必要的風險，因此工程師會放棄euse這個模塊乾脆直接自己重新寫一個模組。

Symptoms of Rotting Design

Viscosity

黏度來自兩個方面:設計黏度與環境黏度。當需要對軟件做更改時，工程師可以選擇保留或是不保留設計的更改方法。而當保存設計的方法比不保存的方法還難實行的時候，則稱＂該設計的黏度很高＂。當開發環境很慢而且效率很低時，則會產生環境的黏度。例如若編譯時間很長，工程師會傾向做不用大量重新編譯的更改，就算這種更改方式在設計的角度上看不是最佳的。

SOLID Design

Single Responsibility Principal

- 定義:

「把一個類或一個模組只負責一項工作，

類別的特有性區分出來。」

One class should have one and only one reasonability.

- 優點:

要是有要修改功能或者是出現BUG，需要改的地方只有相關的類別，可以減少動到的部分。

Bad example

```java
public class badExample {
    public static void main(String[] args) {
        Plane p = new Plane();
        p.job("Airliner");
        p.job("Fighter");
        p.job("Drone");
    }
}
class Plane{
    public void job(String name){
        System.out.println(name + " can carry people.");
    }
}
```

這三者皆可以載人但是並沒有把特別的特徵凸顯出來

Good example：

```java
public class goodExample {
    public static void main(String[] args) {
        AirLiner ap = new AirLiner();
        ap.job("AirLiner");
        Drone dp = new Drone();
        dp.job("Drone");
        Fighter fp = new Fighter();
        fp.job("Fighter");
    }
}
```

客機

無人機

戰鬥機

明顯區別每一個機種
專門負責做什麼事

```java
class AirLiner{
    public void job(String name){
        System.out.println(name + " can carry people.");
    }
}
class Drone {
    public void job(String name){
        System.out.println(name + " can detect and take picture.");
    }
}
class Fighter {
    public void job(String name){
        System.out.println(name + " can fight enemy.");
    }
}
```

客機負責在客人

無人機負責偵查拍照

戰鬥機負責打敵人

Open-Closed Principle

- 定義:

  「對於需要擴展的程式, 盡量對外部做擴充開放(對於需求者), 對內部修改關閉(對於使用方), 避免修改內部程式而造成程式複雜化。」

  Software components should be open for extension, but closed for modification.

- 優點:
  - 可以把東西都分離出來, 哪一個功能出現了bug很快就可以找到並解決
  - 要修改/增加某一功能的話可以避免其他的功能被動到

Bad example

```
3   public class badExample {
        Run | Debug
4       public static void main(String[] args) {
5           Fighter fp = new Fighter(1000, "Mamba");
6           fp.getInfo();
7           System.out.println();
8           Drone dp = new Drone(500, "Bermuda Triangle");
9           dp.getInfo();
10      }
11  }
```

```java
class Fighter {
    Fighter(int gas, String name){
        this.gas = gas;
        this.planeID = name;
    }
    public void move(int flag) {
        if(flag == 1) System.out.println("forward");
        else if(flag == 2) System.out.println("back");
        else if(flag == 3) System.out.println("left");
        else if(flag == 4) System.out.println("right");
    }
    private int gas = 10;
    private String planeID;
    public void getInfo(){
        System.out.println("I am " + planeID);
        System.out.println("Maximum gas: " + gas);
        System.out.println("Maximum bullet: " + bullet);
        System.out.println("Maximum grenade: " + grenade);

    }
    public void shooting(){
        System.out.println("Start Shooting");
        for(int i = 0; i < 5; i++){
            for(int j = 0; j <= i; j++)
                System.out.print("Da ");
            System.out.println();
        }
    }
    private final int bullet = 10000;
    private final int grenade = 10;
}
```

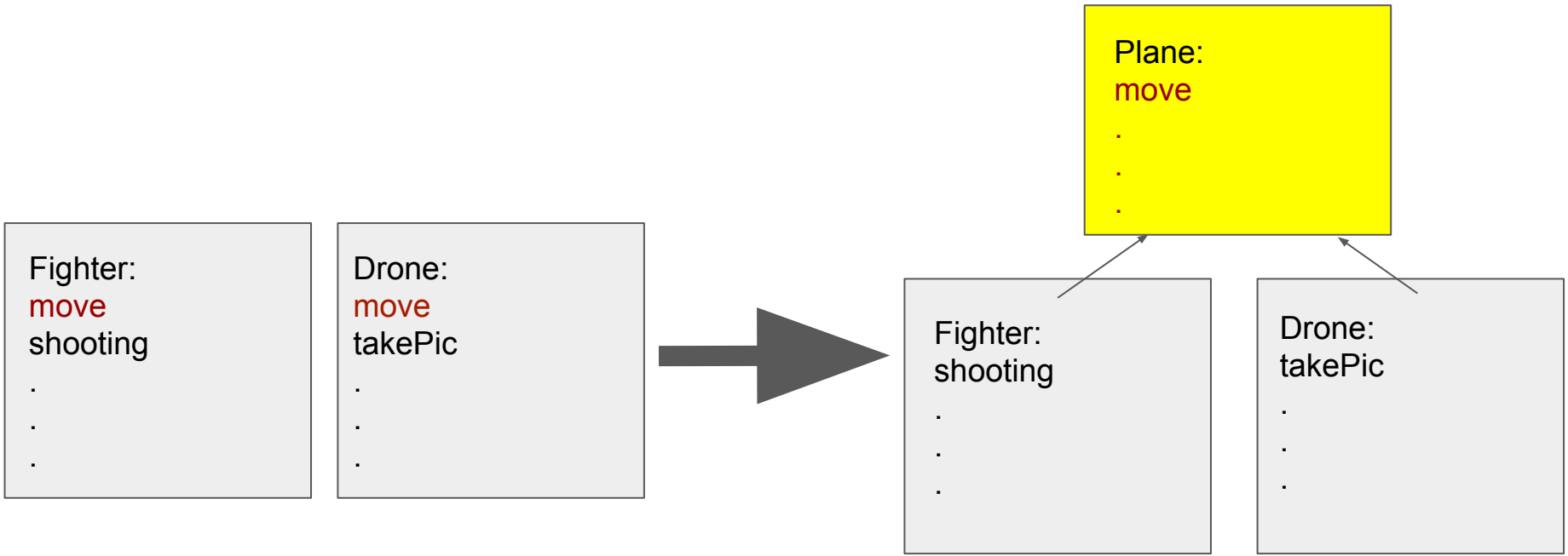
功能特別並沒有特別的凸顯出來，隨然有包含在裡面但這樣的情形會造成程式複雜化

```
45  class Drone {
46      Drone(int gas, String name){
47          this.gas = gas;
48          this.planeID = name;
49      }
50      public void move(int flag) {
51          if(flag == 1) System.out.println("forward");
52          else if(flag == 2) System.out.println("back");
53          else if(flag == 3) System.out.println("left");
54          else if(flag == 4) System.out.println("right");
55      }
56      private int gas = 10;
57      private String planeID;
58      public void getInfo(){
59          System.out.println("I am " + planeID);
60          System.out.println("Maximum gas: " + gas);
61      }
62      public void takePic(){
63          System.out.println("The photo is save.");
64      }
65
66  }
67
```

這情形跟fighter一樣，可以發現move這個功能大家都有卻都沒有把它分開在一起

解決方法:

要找出每一個共同擁有的功能, 並把它放在同一個類別



Good example：

```java
public class goodExample {
    public static void main(String[] args) {
        Fighter fp = new Fighter(1000, "Mamba");
        Run | Debug
        fp.getInfo();
        System.out.println();
        Drone dp = new Drone(500, "Bermuda Triangle");
        dp.getInfo();
    }
}
```

```java
class Plane{
    public void getInfo(){
        System.out.println("I am " + getPlaneID());
        System.out.println("Maximum gas: " + getGas());
    }
    public void move(int flag) {
        if(flag == 1) System.out.println("forward");
        else if(flag == 2) System.out.println("back");
        else if(flag == 3) System.out.println("left");
        else if(flag == 4) System.out.println("right");
    }
    private int gas = 10;
    private String PlaneID;
    public void setGas(int gas){
        this.gas = gas;
    }
    public void setPlaneID(String name){
        this.PlaneID = name;
    }
    public int getGas(){
        return gas;
    }
    public String getPlaneID(){
        return PlaneID;
    }
}
```

把所有共同一樣功能的放在plane這一塊

```java
class Fighter extends Plane{
    Fighter(int gas, String name){
        setGas(gas);
        setPlaneID(name);
    }
    @Override
    public void getInfo(){
        System.out.println("I am " + getPlaneID());
        System.out.println("Maximum gas: " + getGas());
        System.out.println("Maximum bullet: " + bullet);
        System.out.println("Maximum grenade: " + grenade);

    }
    public void shooting(){
        System.out.println("Start Shooting");
        for(int i = 0; i < 5; i++){
            for(int j = 0; j <= i; j++)
                System.out.print("Da ");
            System.out.println();
        }
    }
    private final int bullet = 10000;
    private final int grenade = 10;

}
```

```java
class Drone extends Plane{
    Drone(int gas, String name){
        setGas(gas);
        setPlaneID(name);
    }
    @Override
    public void getInfo(){
        System.out.println("I am " + getPlaneID());
        System.out.println("Maximum gas: " + getGas());
    }
    public void takePic(){
        /* do something camera can do */
        System.out.println("The photo is save.");
    }
```

shooting是戰鬥機有的功能而其他都有的共同功能就挪下來用就可以了

takePic是無人機的特別功能

Liskov Substitution Principle

- 定義:

「child class必須能夠替換parent class，並且行為正常」

objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

繼承性的利與弊:

- 繼承包含的一層涵義:
  parent class中已經實現好的方法實際上就像是在設定一個規範和契約, 雖然沒有強制要求child class要遵守這個規定, 但是如果child class對已經實現好的方法任意修改那麼可能會對整個的繼承體系照成破壞

- 在程式設計中繼承會帶來許多便利性但是同時也會帶來弊端, 例如: 使用繼承會給城市帶來侵略性, 程式可移植性會降低, 增加對象間的耦合性, 所以當要修改parent class時要考慮到所有的child class, 否則可能會造成child class的功能產生故障

  (耦合性: class B extend class A-> 一旦 A 變化了就可能會影響到B)

- 要怎麼正確使用繼承? ⟶ 滿足liskov substitution principle

Bad example

```java
package LiskovSubstitution.bad;

import java.util.*;
public class badExample {

    public static void main(String[] args){

        Airplane a = new Airplane();
        Fighter f = new Fighter();
        Drone d = new Drone();

        System.out.println("Airplane:");
        a.fly();
        a.engine();
        a.manned();
        System.out.println();

        System.out.println("Fighter:");
        f.fly();
        f.engine();
        f.manned();
        f.fight();
        System.out.println();

        System.out.println("Drone:");
        d.fly();
        d.engine();
        d.manned();

    }
}

```

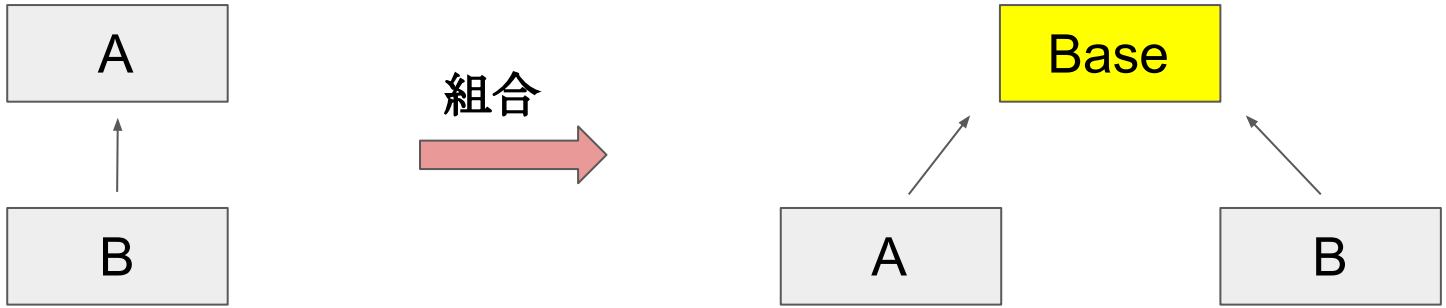
```java
class Airplane {

    public void fly() {
        System.out.println("can fly");
    }
    public void engine() {
        System.out.println("have engine");
    }
    public void manned() {
        System.out.println("can carry people");
    }

}

class Fighter extends Airplane {

    public void fight() {
        System.out.println("can fight");
    }

}

class Drone extends Airplane {

    @Override
    public void manned() {
        System.out.println("can't carry people");
    }

}
```

這裡Drone重寫了Airplane的manned(), 就代表Drone在manned這裡沒有符合Airplane的特性, 所以Drone不能繼承Airplane

⟶ 違反LSP

如何降低違反LSP情形？

- 在child class中**盡量不要**override parent class的方法
- 避免使用繼承，盡量用**聚合(aggregation)，組合(compostion)，依賴(dependency)**來解決問題

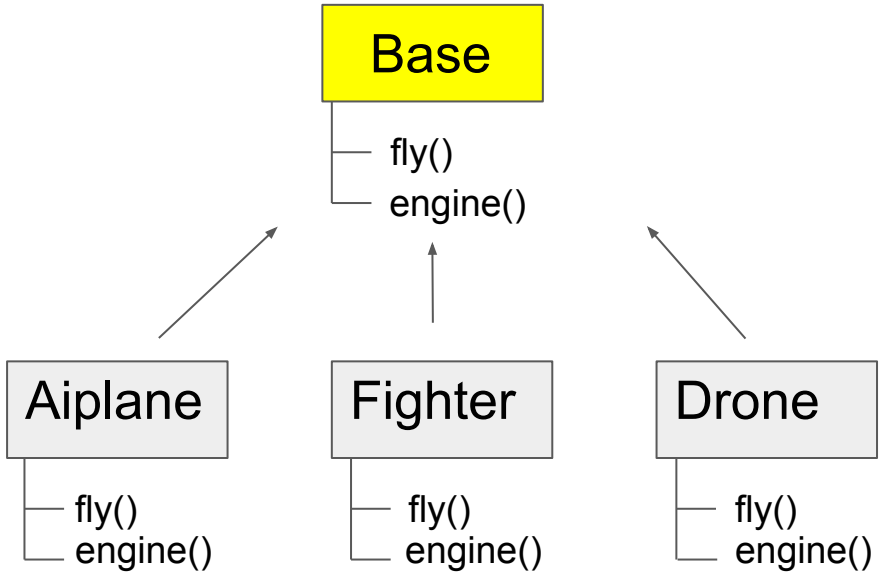


Good example

```java
package LiskovSubstitution.good;

import java.util.*;
public class goodExample {

    public static void main(String[] args){

        Airplane a = new Airplane();
        Fighter f = new Fighter();
        Drone d = new Drone();

        System.out.println("Airplane:");
        a.fly();
        a.engine();
        a.manned();
        System.out.println();

        System.out.println("Fighter:");
        f.fly();
        f.engine();
        f.manned();
        f.fight();
        System.out.println();

        System.out.println("Drone:");
        d.fly();
        d.engine();
        d.manned();

    }
}

```

```java
class Base {

    public void fly() {
        System.out.println("can fly");
    }
    public void engine() {
        System.out.println("have engine");
    }
}

interface Plane {

    public abstract void manned();

}
```

建立一個 interface，在裡面創一個 abstract method 以達到客製化每種飛機的 manned()



打破 Airplane 和 Fighter & Drone 的繼承關係
建立一個更基礎的 parent class，把他們都拉到 Base，而 Base 裡的 method 都是大家都有的功能

```java
class Airplane extends Base implements Plane {
    @Override
    public void manned() {
        System.out.println("can carry people");
    }

}

class Fighter extends Base implements Plane {
    @Override
    public void manned() {
        System.out.println("can carry people");
    }

    public void fight() {
        System.out.println("can fight");
    }

}

class Drone extends Base implements Plane {
    @Override
    public void manned() {
        System.out.println("can't carry people");
    }

}
```

manned()會因機種不同而異

Interface Segregation Principle

- 定義:

「不應該強迫用戶依賴他們沒有要用的方法,

應該最小化類別與類別之間的介面。」

Clients should not be forced to depend on methods that they do not use.

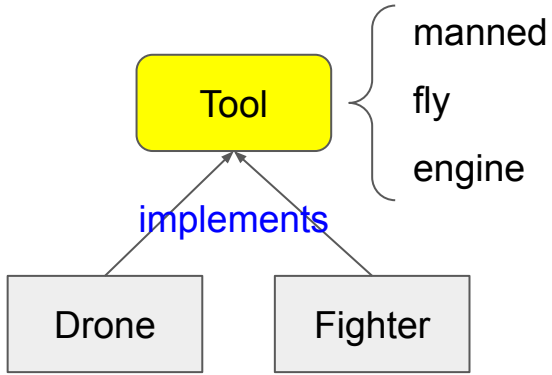
- 如果介面過於肥胖, 可能會造成的問題:

1. **繼承或抽象類別:** 多餘的介面在子類別可能會有空實作的狀況, 造成不可預期的錯誤。
2. **Compile:** 假設A是一個有BCD方法的介面, E繼承了A並使用其中的BC, 若有人更動了D的內容, E雖然沒有使用到D, 但還是得重新Compile。

Bad example

```java
public class badExample {
    Run | Debug
    public static void main(String[] args){
        Airplane ap = new Airplane();
        ap.action(new Drone());
        ap.action(new Fighter());
    }
}

class Airplane{
    public void action(Tool a1){
        a1.manned();
        a1.fly();
        a1.engine();
    }
}

interface Tool{
    void manned();
    void fly();
    void engine();
}
```

一個叫做Tool的介面，中有manned、fly、engine三個方法，Drone(無人機)、fighter(戰鬥機)皆implement此介面，因此兩類別都繼承了三個方法。

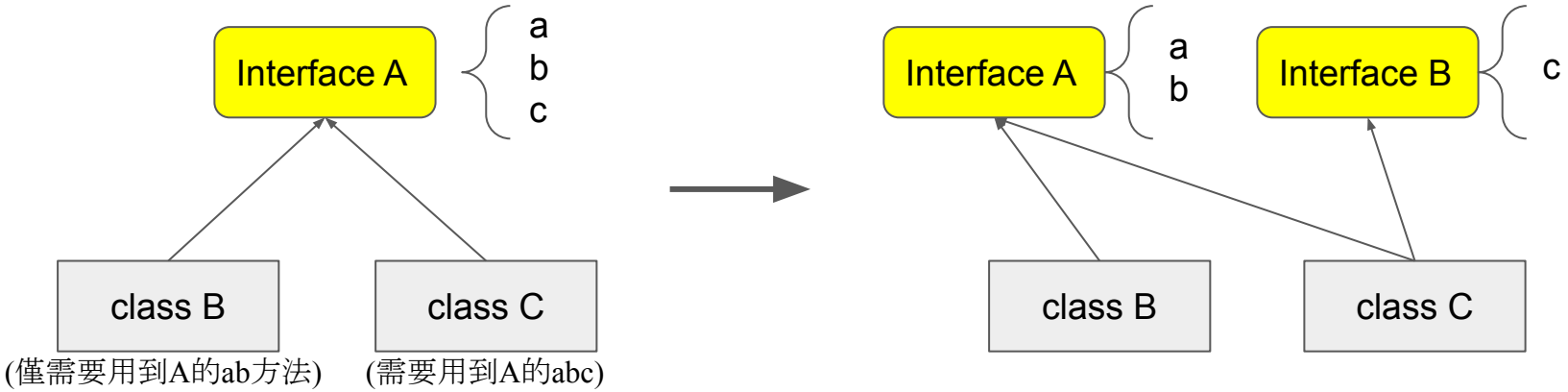


```java
class Drone implements Tool{
    public void manned(){
        System.out.println("Drone can carry people.");
    }
    public void fly(){
        System.out.println("Drone can fly.");
    }
    public void engine(){
        System.out.println("Drone have engine.");
    }
}
class Fighter implements Tool{
    public void manned(){
        System.out.println("Fighter can carry people.");
    }
    public void fly(){
        System.out.println("Fighter can fly.");
    }
    public void engine(){
        System.out.println("Fighter have engine.");
    }
}
```

Drone繼承了Tool介面，因此繼承了此三個方法，但是事實上Drone不能夠載人，manned這個function出現在Drone類別中是錯誤的。

解決方法？

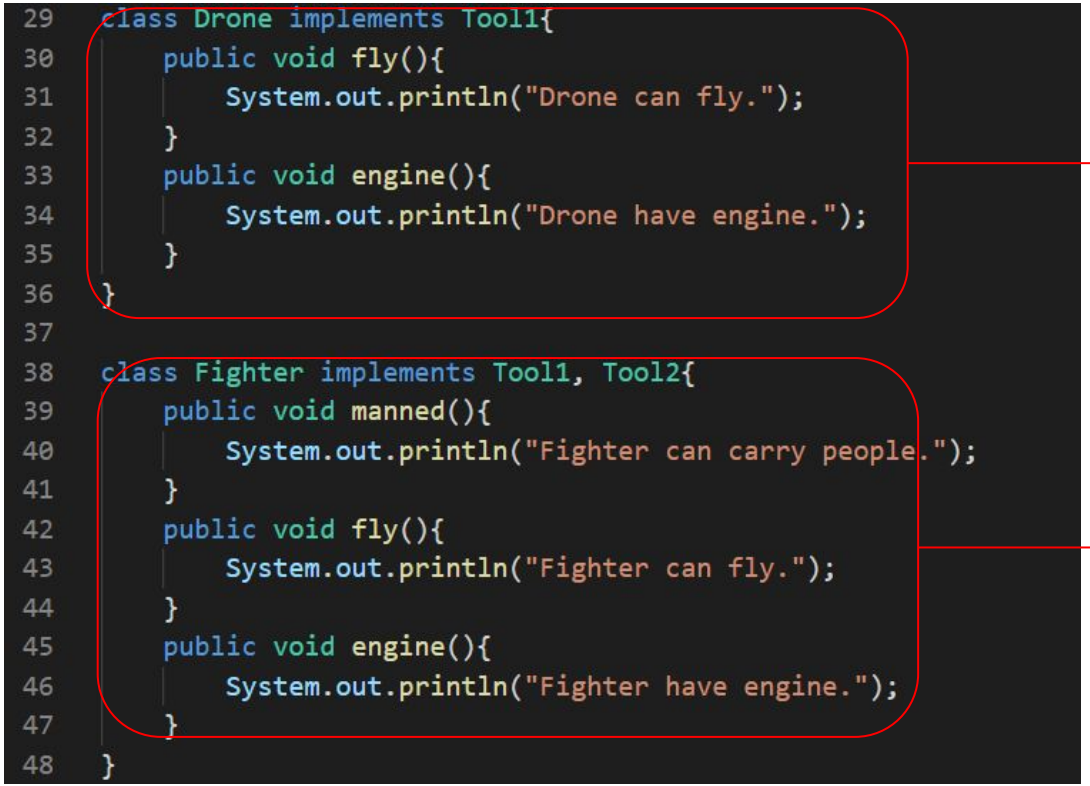
- **客製化介面：**將類別的Interface客製化，只提供他需要用到的。
- **分割組合：**將類別分割後，需要implements的類別再利用組合的方式實現，這裡可以使用多重繼承或Delegate等等。



Good example

```java
public class goodExample {
    Run | Debug
    public static void main(String[] args){
        Airplane ap = new Airplane();
        ap.action1(new Drone());
        ap.action1(new Fighter());
        ap.action2(new Fighter());
    }
}
class Airplane{
    public void action1(Tool1 a1){
        a1.fly();
        a1.engine();
    }
    public void action2(Tool2 a2){
        a2.manned();
    }
}

interface Tool1{
    public void fly();
    public void engine();
}

interface Tool2{
    public void manned();
}
```

把原本的Tool分為兩個interface，其中Tool1有fly和engine方法，Tool2有manned方法。



Drone只需要fly和engine兩個方法，因此implements有這兩個方法的Tool1 interface即可。

Fighter需要三個方法，因此兩個interface：Tool1和Tool2都需要implements。

Dependency Inversion Principle

- 定義:

「高層模組不應該依賴於低層模組。

兩者皆應該依賴抽象(Interface)。」

Depend on abstraction, not on concreations.

Bad example

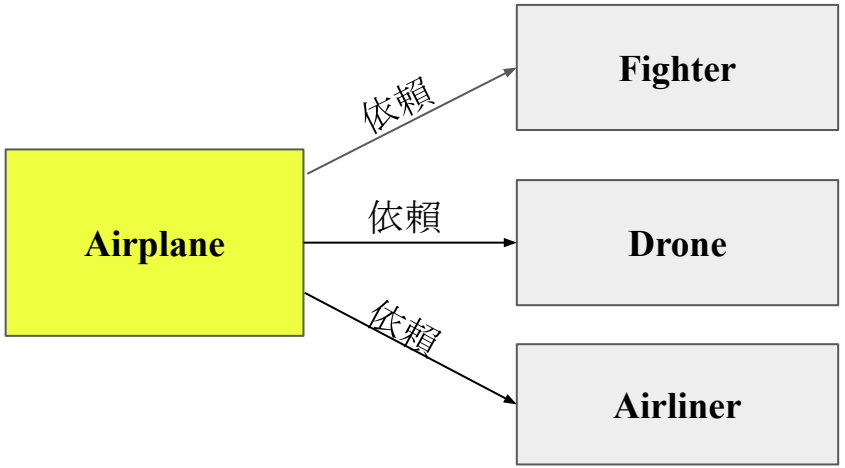
```java
public class badExample {
    public static void main(String[] args) {
        Airplane ap = new Airplane();
        ap.siren(new Fighter());
        ap.siren(new Drone());
        ap.siren(new Airliner());
    }
}

class Airplane{
    public void siren(Fighter sound){
        System.out.println(sound.getSiren());
    }
    public void siren(Drone sound){
        System.out.println(sound.getSiren());
    }
    public void siren(Airliner sound){
        System.out.println(sound.getSiren());
    }
}

class Fighter{
    public String getSiren() {
        return "Da Da Da Da!";
    }
}
```

若我今天想輸入一個機種, 去得到他的聲音。則在Airplane這個class裡, 我卻需要寫三種siren funcions(假設目前只有三種機種), 由此馬上可以聯想的到若是以後有 100種機種, 則不就要寫100種siren funcions?

```java
class Drone{
    public String getSiren() {
        return "Hong Hong Hong~";
    }
}

class Airliner{
    public String getSiren() {
        return "__";
    }
}
```

解決方法？

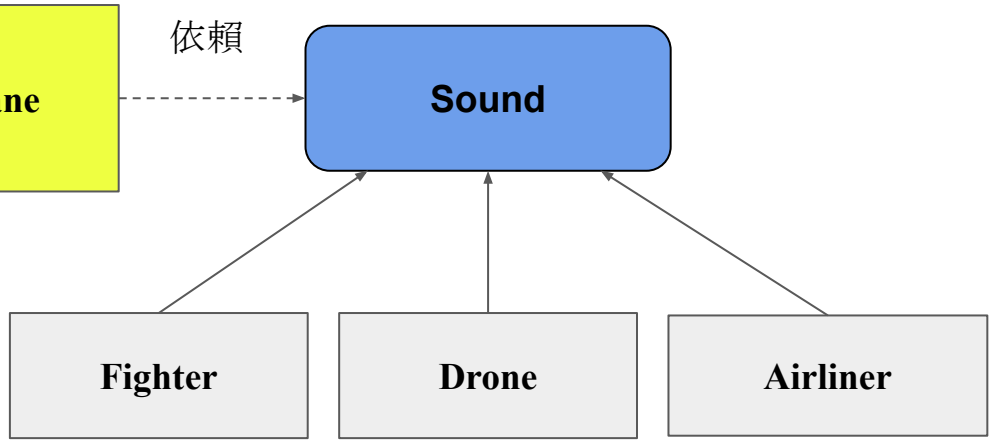
目前可以發現我們class的關係為 ：



由這個關係圖可以發現，Airplane依賴Fighter這些class等等，這種關係我們叫做"高階的class依賴低階的class"，這樣代表如果我低階層的class有所變動，我的高階的class也得做改動。所以我們應該依賴反轉(Dependecy Inversion)，使得高階不再依賴低階，而是中間透過抽象來達到依賴的反轉。

解決方法？

After Dependcy Inversion :



透過一個interface(一個抽象的概念), 讓高階和低階的class都依賴, 這樣以後高階或低階有修改, 都不會直接的影響到對方, 可以讓code維護起來更為容易, code也不會變的肥大。

Good example


```java
public class goodExample {
    public static void main(String[] args) {
        Airplane ap = new Airplane();
        ap.siren(new Fighter());
        ap.siren(new Drone());
        ap.siren(new Airliner());
    }
}

class Airplane{
    public void siren(Sound sound){
        System.out.println(sound.getSiren());
    }
}

interface Sound{
    public String getSiren();
}

class Fighter implements Sound{
    @Override
    public String getSiren() {
        return "Da Da Da Da!";
    }
}

class Drone implements Sound{
    @Override
    public String getSiren() {
        return "Hong Hong Hong~";
    }
}
class Airliner implements Sound{
    @Override
    public String getSiren() {
        return "__________";
    }
}
```

相關連結

Source code: https://github.com/darrenleeleelee1/Advance-Java-Presentation/tree/master

SOLID in Java: https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/

Symptoms of Rotting Design: https://rusirub.wordpress.com/2008/04/09/symptoms-of-rotting-design/