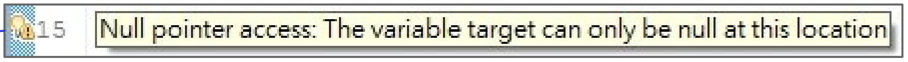
Code Smells

Shin-Jie Lee (李信杰)
Associate Professor
Department of CSIE
National Cheng Kung University

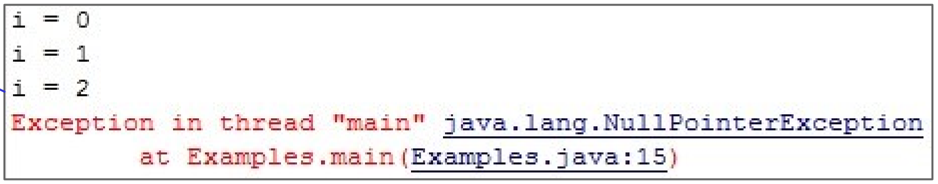
Unresolved warnings

- The program is still runnable, but may cause unexpected errors

```
public void printSomething() {
   int size = 3
   String target = null;

   for(int i = 0; i < size; i++) {
      System.out.println(“i = ” + i);
   }


System.out.println(target.toString());
}
```



```
15   Null pointer access: The variable target can only be null at this location
```



```
i = 0
i = 1
i = 2
Exception in thread "main" java.lang.NullPointerException
        at Examples.main(Examples.java:15)
```

Every dynamic allocated memory is de-allocated or there is garbage collection

- The memory may be fully occupied when an amount of instantiated objects are not deleted as they will no longer be used.

```
int main() {
  int size = 10;
  int result = 0;
  int array = new int[size];

  // Assign value to the array
  for(int i = 0; i < size; i++) {
    array[i] = i;
  }

  for(int i = 0; i < size; i++) {
    result += array[i];
  }
}
```

Memory Leak

Long method(1/2)

- The object programs that live best and longest are those with short methods.
- The longer a procedure is, the more difficult it is to understand.
- It’s not easy to name the long method

Long method(2/2)

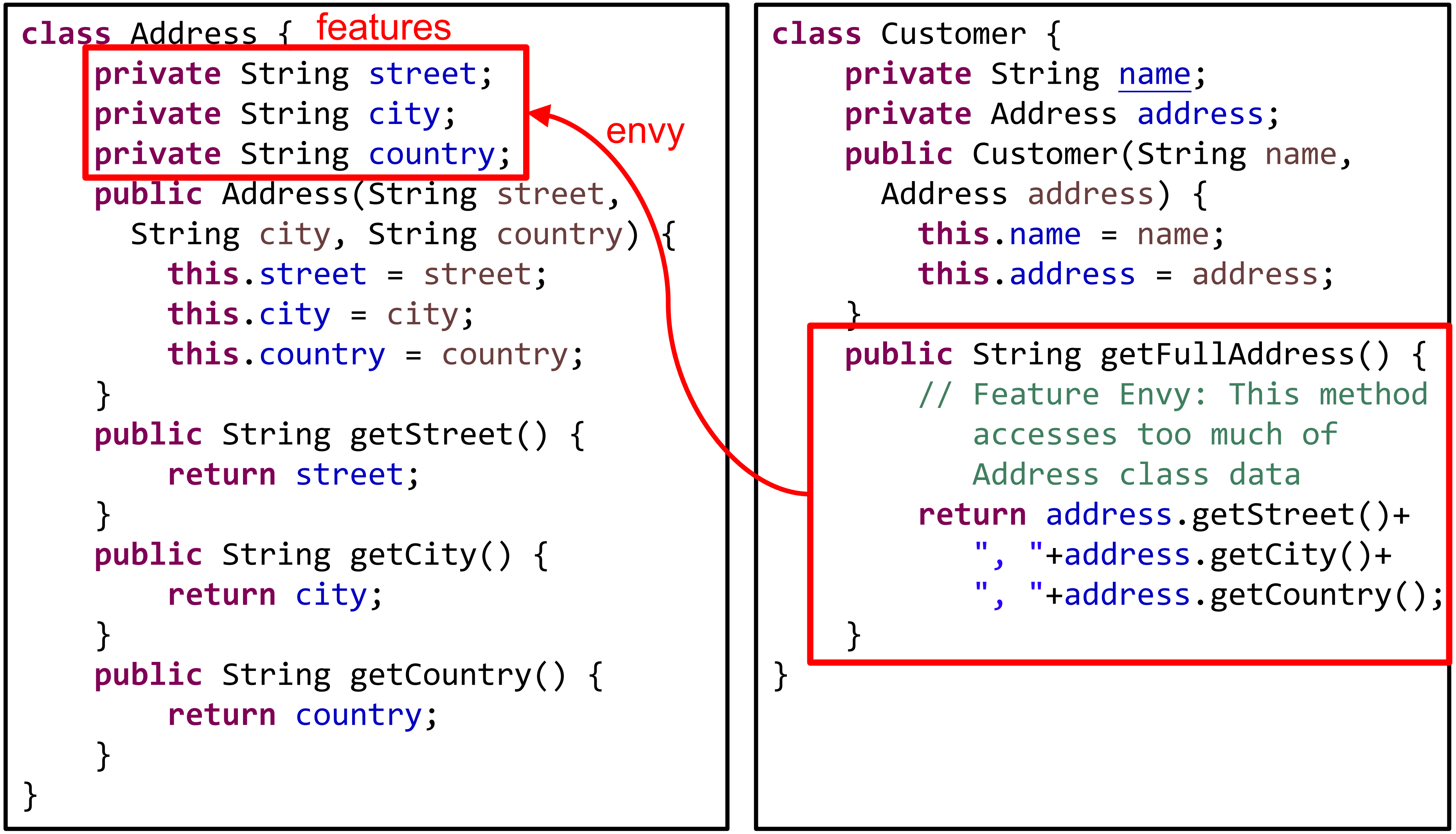
- Decompose the long method into short methods through *Extract Method*

```
public void createPartControl(Composite parent) {
    _failnodes = new HashSet<Object>();
    _comps = new ConcurrentLinkedQueue<IComponent>();
    _viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL);
    _viewer.setInput(getViewSite());
    …
    _selectionHandler = new SelectionChangHandler();
    _selectionHandler.setViewer(_viewer);
}
```

Feature Envy

- A method that seems more interested in a class other than the one it actually is in.
- Feature Envy introduces tight coupling between classes.
  - Changes in the "envied" class can affect the class with Feature Envy.

Example of Feature Envy



```
class Address {
    private String street;
    private String city;
    private String country;
    public Address(String street,
      String city, String country) {
        this.street = street;
        this.city = city;
        this.country = country;
    }
    public String getStreet() {
        return street;
    }
    public String getCity() {
        return city;
    }
    public String getCountry() {
        return country;
    }
}
```

```
class Customer {
    private String name;
    private Address address;
    public Customer(String name,
      Address address) {
        this.name = name;
        this.address = address;
    }
    public String getFullAddress() {
        // Feature Envy: This method
           accesses too much of
           Address class data
        return address.getStreet()+
           ", "+address.getCity()+
           ", "+address.getCountry();
    }
}
```

Lab – Refactoring for Feature Envy

- ❑ Refactor the code using Move Method
- ❑ Comparing the code before and after refactoring with adding an attribute `String street` to Address

Unsuitable naming

- Giving a suitable name for a class, a method, or a variable will make programmers easy to understand

```
public class T() {
    boolean b = false;

    public int xyz(int x, int y, int z) {
        int r = 0;
        r = (x + y) * z / 2;
        return r;
    }
}
```

```
public class Trapezoid() {
    boolean isIsosceles = false;

    public int calculateArea(int top, int bottom, int
height) {
        int area = 0;
        area = (top + bottom) * height / 2;
        return area;
    }
}
```

Downcasting (1/2)

- Casting is another bane of the Java programmer's life.
- As much as possible try to avoid making the user of a class do downcasting.

```
void testType() {
   unsigned short x = 65535;
   short y = x;

    for(int i = 0; i < y; i++) {
       Do something
    }
}
```

Downcasting (2/2)

I'm still a Cat after upcasting, but compiler treats me as an Object
That means i can't do anything, that's specific for Animals... or Cats.
object
animal
mammal
cat
i can't!
Be a dog!
java.lang.ClassCastException: Cat cannot be cast to Dog
by Sinipull for codecall.net

```
class Animal() {}

class Mammal extends Animal() {}

class Cat extends Mammal() {}

class Dog extends Mammal() {}
```

```
Mammal m = new Cat()
Dog c = (Dog)m; // Throws ClassCastException
```

Lab

- ❑ 什麼樣的情況會使用到Downcasting?

- Interacting with APIs or Frameworks?
- Working with Legacy Code?
- Working with Collections of Mixed Types?
- Deserialization?

Loop termination conditions are obvious and invariably achievable

```
for(int i = 1; (i % 2) ? ((i + 100) < 200) : ((i* 30) < 50); i++) {
    Do something
}

for(int i = 0; i < 100; i++) {
    Do something
    i = i * 5;
}

int i = 0;
while(i < 10) {
    Do something
}
```

```
for(int i = 1; i < 10; i++) {
    Do something
}

for(int i = 0; i < 100; i++) {
    Do something
}


int i = 0;
while(i < 10) {
    Do something
     i++;
}
```

Parentheses are used to avoid ambiguity

- Use parentheses to increase the readability and prevent logical errors

```
public int trapezoidArea(int top, int bottom, int height) {
    int area = top + bottom * height / 2;
    return area;
}

if (isOK && getX() * getY() == 2000 && !isFinished) {
    Do something
}
```

```
public int trapezoidArea(int top, int bottom, int height) {
    int area = (top + bottom) * height / 2;
    return area;
}

if ((isOK) && (getX() * getY() == 2000) && (!isFinished)) {
    Do something
}
```

Lack of comments(1/2)

- A good time to use a comment is when you don't know what to do.

- In addition to describing what is going on, comments can indicate areas in which you aren't sure.

- A comment is a good place to say *why* you did something. This kind of information helps future modifiers, especially forgetful ones.

Lack of comments(2/2)

```
public RSSIMapCollection() {
   _maps = new Hashtable<String, RSSIMap>();
   _listeners = new Vector<RSSIMapCollectionEventListener>();
   _stabilizes = new SelectionProperty(STABILIZES_LABEL);
   _stabilizes.addElement(Stabilize.NONE);
   _stabilizes.addElement(Stabilize.THRESHOLD);
   _stabilizes.addElement(Stabilize.AVERAGE);
   _stabilizes.addElement(Stabilize.WIEGHTED);
   _stabilizes.setSelectedItem(Stabilize.THRESHOLD);
}
```

```
public RSSIMapCollection() {
   _maps = new Hashtable<String, RSSIMap>();
   _listeners = new Vector<RSSIMapCollectionEventListener>();

   // Initialize a selection property for multiple stabilizations
   _stabilizes = new SelectionProperty(STABILIZES_LABEL);
   _stabilizes.addElement(Stabilize.NONE);
   _stabilizes.addElement(Stabilize.THRESHOLD);
   _stabilizes.addElement(Stabilize.AVERAGE);
   _stabilizes.addElement(Stabilize.WIEGHTED);
   _stabilizes.setSelectedItem(Stabilize.THRESHOLD);
}
```

Files are checked for existence before attempting to access them

- ❑ 開啟檔案之後沒有測試檔案是否正確載入就進行操作。(using C++ as example language)
  - ➢ 開啟檔案之後應該測試檔案是否已正確開啟。

```
    … // include necessary header files.
using namespace std;
int main () {
    ifstream inputFileStream;
   inputFileStream.open("MyText.txt");
   char output[100];
    while (!inputFileStream.eof()) {
       inputFileStream >> output;
       … // process read-in data
   }
   inputFileStream.close();
}
```

read in lines without checking file existence.

```
    … // include necessary header files.
using namespace std;
int main () {
    ifstream inputFileStream;
   inputFileStream.open("MyText.txt");
   char output[100];
    if (inputFileStream.is_open()) {
        while (!inputFileStream.eof()) {
         inputFileStream >> output;
          … // process read-in data
       }
   } else {
        … // error-handling code
   }
}
```

Check if file has been opened successfully.

Duplicated Code (1/2)

- If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

```
public class ClassAReport {
    …
    public int calculateAverage(List<Integer>
scores) {
        int sum, average = 0;
        for (int i = 0; i < scores.size(); i++) {
            sum += scores.get(i);
        }
        average = sum / scores.size();
        retrun average;
    }
    …
}
```

```
public class ClassBReport {
    …
    public int calculateAverage(List<Integer>
scores) {
        int sum, average = 0;
        for (int i = 0; i < scores.size(); i++) {
            sum += scores.get(i);
        }
        average = sum / scores.size();
        return average;
    }
    …
}
```

This piece of code occurs more than once!

Duplicated Code (2/2)

- The simplest duplicated code problem is when you have the same expression in two methods of the same class.
  - Then all you have to do is *Extract Method* and invoke the code from both places.

```
public class AverageCalculator {
    public int calculateAverage(List<Integer> scores) {
        int sum, average = 0;
        for (int i = 0; i < scores.size(); i++) {
            sum += scores.get(i);
        }
        average = sum / scores.size();
        retrun average;
    }
    …
}
```

This class is responsible for calculating average.

```
public class ReportCardManager {
    public static void main (String args[]) {
        AverageCalculator ac = new AverageCalculator();
        ClassAReport classAReport = new ClassAReport();
        ClassBReport classBReport = new ClassBReport();
        int classAAverage = classAReport.calculateAverage(ac);
        int classBAverage = classBReport.calculateAverage(ac);
        ….
    }
}
```

```
public class classAReportCard {
    private List<Integer> classAScores;
    … // initialize scores
    public int calculateAverage (AverageCalculator ac) {
        retrun ac.calculateAverage(classAScores);
    }
// Another Class
public class classBReportCard {
    private List<Integer> classBScores;
    … // initialize scores
    public int calculateAverage (AverageCalculator ac) {
        retrun ac.calculateAverage(classBScores);
    }
```

All methods have appropriate access modifiers and return types (1/2)

- The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes.

```
Class Account {
public:
    string _password;
    string getPassword();
    ...
};
```

```
Class Account {
public:
   string getPassword();
   ...
private:
    string _password;
    ...
};
```

All methods have appropriate access modifiers and return types (2/2)

- Add an appropriate return type to help check if the method executes successfully.

```
bool openAndProcessFile(string filePath) {
   ifstream ifs;
    ifs.open(filePath.c_str());
    if (!ifs.is_open())
        return false;
   …
   return true;
}
```

Return false if file is not opened successfully.

Are there any redundant or unused variables?

❑ Remove unused variables from source code

```
public int calculateClassAverage (List<Integer> scores) {
    int rank = 0; // never used
    int sum, average = 0;
    for (int i = 0; I < scores.size(); i++) {
        sum += scores.get(i);
    }
    return average;
}
```

```
public int calculateClassAverage (List<Integer> scores) {
    int sum, average = 0;
    for (int i = 0; I < scores.size(); i++
        sum += scores.get(i);
    }
    return average;
}
```

Delete unused variable

Indexes or subscripts are properly initialized, just prior to the loop

- Variables used in the termination conditions should be initialized properly

```
int i;
while (i < 0) {
    doSomething();
    i++;
}
```

```
int i = -10;
while (i < 0) {
    doSomething();
    i++;
}
```

initialized

```
int i;
for (i ; i < someInt; i++) {
    doSomething();
}
```

```
int i = 0;
for (i ; i < someInt; i++) {
    doSomething();
}
```

initialized

Is overflow or underflow possible during a computation?

- An overflow or underflow during a computation may cause system crash

```
int main () {
   short int addend = 30000;
    short int augend = 30000;
    short sum = addend + augend;
    doSomething(sum);
};
```

```
int main () {
   short int addend, augend;
    cin >> addend;
    cin >> augend

   if (addend + augend > numeric_limits<short>::max() ||
          (addend + augend < numeric_limits<short>::min()) {
      throw “short integer overflow / underflow”
    short int sum = addend + augend;
}
```

Are divisors tested for zero?

- Divisors should not be zero at runtime

```
    int divisor;
    int dividend;
    cin >> divisor;
    cin >> dividend;
    int quotient = dividend / divisor;
    …
}
```

```
    int divisor;
    int dividend;
    cin >> divisor;
    cin >> dividend;

    if (divisor == 0) {
      throw “divisor is 0”;
    }
    int quotient = dividend / divisor;
    …
}
```

Inconsistent coding standard

- ❑ To use meaningful names
- ❑ To use an underline as the prefix of an attribute of a class

1 成員變數名稱前應加底線。
2 To use meaningful names

```
class Car {
public:
    int getAbc();
    string getXyz();
    ...
private:
   int id;
    string manufactureDate;
    ...
};
```

meaningless naming

Inconsistent coding standard

```
class Car {
public:
    int getVehicleId ();
    string getManufactureDate();
    ...
private:
   int _id;
    string _manufactureDate;
    ...
};
```

Data clumps$_1$

- Often you'll see the same three or four data items together in lots of places: fields in a couple of classes, parameters in many method signatures.

```java
public class Customer {
    private String name;
    private String title;
    private String house;
    private String street;
    private String city;
    private String postcode;
    private String country;
    …
}
```

```java
public class Staff {
    private String lastname;
    private String firstname;
    private String house;
    private String street;
    private String city;
    private String postcode;
    private String country;
    …
}
```

Data clumps$_2$

- Often you'll see the same three or four data items together in lots of places: fields in a couple of classes, parameters in many method signatures.

```java
public class Address {
    private String house;
    private String street;
    private String city;
    private String country;
    …
}
```

```java
public class Customer {
    private String name;
    private String title;
    private Address customerAddr;



    …
}
```

```java
public class Staff {
    private String lastname;
    private String firstname;
    private Address staffAddr;



    …
}
```

Simulated Polymorphism$_1$

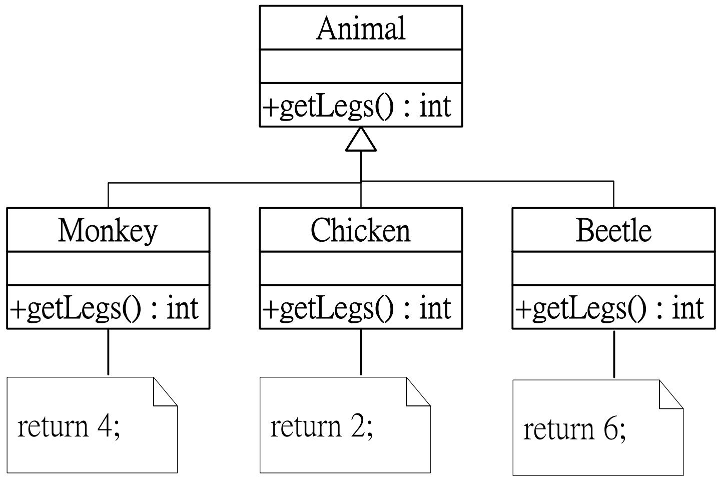
- Simulate the behavior of polymorphism by switch/if-else

- Not good:

```
public int getLegsNum() {
    switch(animal) {
    case ‘chicken’:
        return 2;
    case ‘monkey’:
        return 4;
    case ‘beetle’:
        return 6;
    default:
        return 0;
    }
}
```

- Better solution:

```
public int getLegsNum(Animal a) {
    return a.getLegs();
}
```

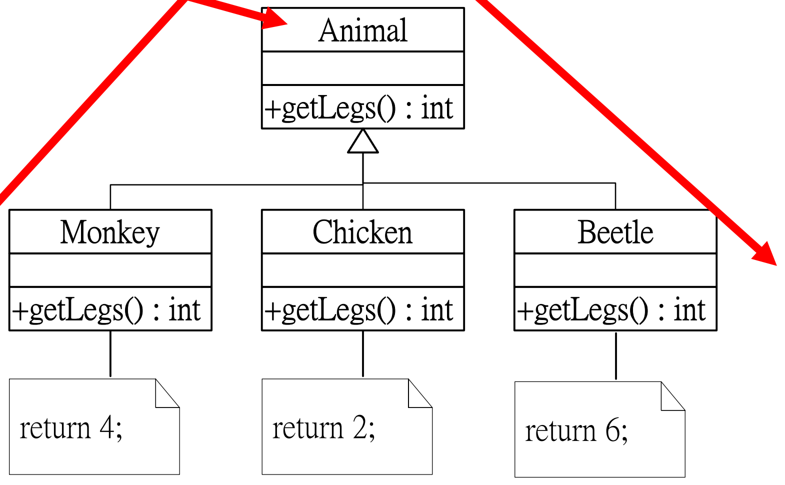


Simulated Polymorphism$_2$

- 適合以Polymorphism代替switch/if-else的時機?

Condition為分類概念時　且　Branch行為未來可能需要擴展時

```
public int getLegsNum() {
   switch(animal) {
   case 'chicken':
      return 2;
   case 'monkey':
      return 4;
   case 'beetle':
      return 6;
   //case '預想可能性'
   default:
      return 0;
   }
}
```



Lab – Simulated Polymorphism

- 請將以下程式改寫為Polymorphism

```
class Animal {
    String type;
    public Animal(String type) {
        this.type = type;
    }
    public void makeSound() {
        if ("Dog".equals(type)) {
            System.out.println("Woof!");
        } else if ("Cat".equals(type)) {
            System.out.println("Meow!");
        } else if ("Cow".equals(type)) {
            System.out.println("Moo!");
        } else {
            System.out.println("Unknown animal sound.");
        }
    }
}
```

Large class

- A class contains many fields/methods/lines of code.
- Large classes often violate the Single Responsibility Principle (SRP).

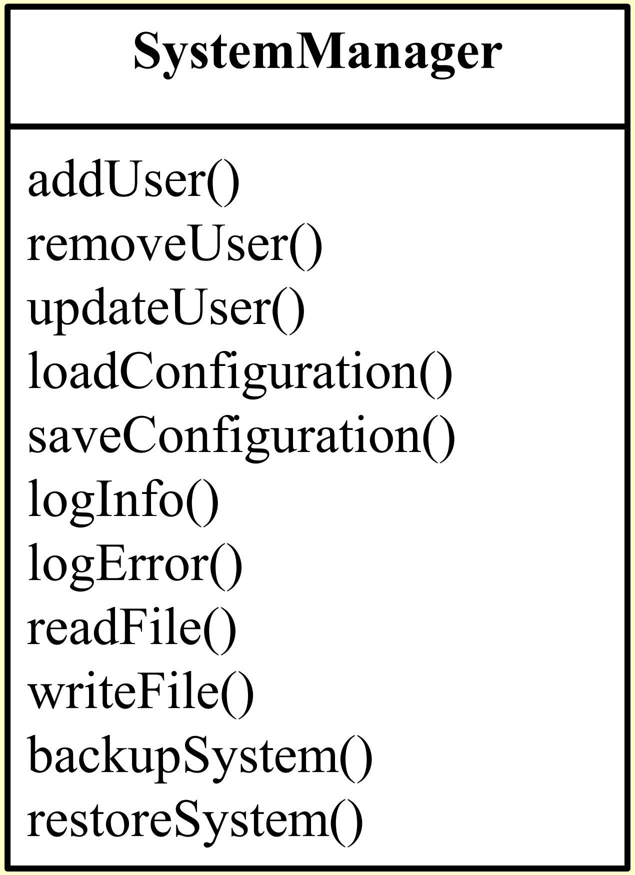
- Use class names as a guide

Ask: ***Can I give this class a meaningful name that clearly defines its purpose?***

If the name feels vague or **generic** (e.g., Manager, Helper), the class probably has unclear responsibilities.

Lab – Large Class

- Is this class a large class? Why?



Long parameter list

- ❑ Long parameter lists are hard to understand, and they become inconsistent and difficult to use

- Not good:

```
public class Member {
    public createMember(
        Name name,
        String country,
        String postcode,
        String city,
        String street,
        String house) {
        …
    }
}
```

- Better solution:

```
public class Member {
    public createMember(
        Name name,
        Address address) {
        …
    }
}
```

Message Chains

- You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on.
- It creates tightly coupled code and violates the Law of Demeter (a design principle that says "don't talk to strangers")

Example

```java
public class Company {
    private Person manager;
    public Company(Person manager) {
        this.manager = manager;    }
    public Person getManager() {
        return manager;    }
}
```

```java
public class Person {
    private Address address;
    public Person(Address address) {
        this.address = address;    }
    public Address getAddress() {
        return address;    }
}
```

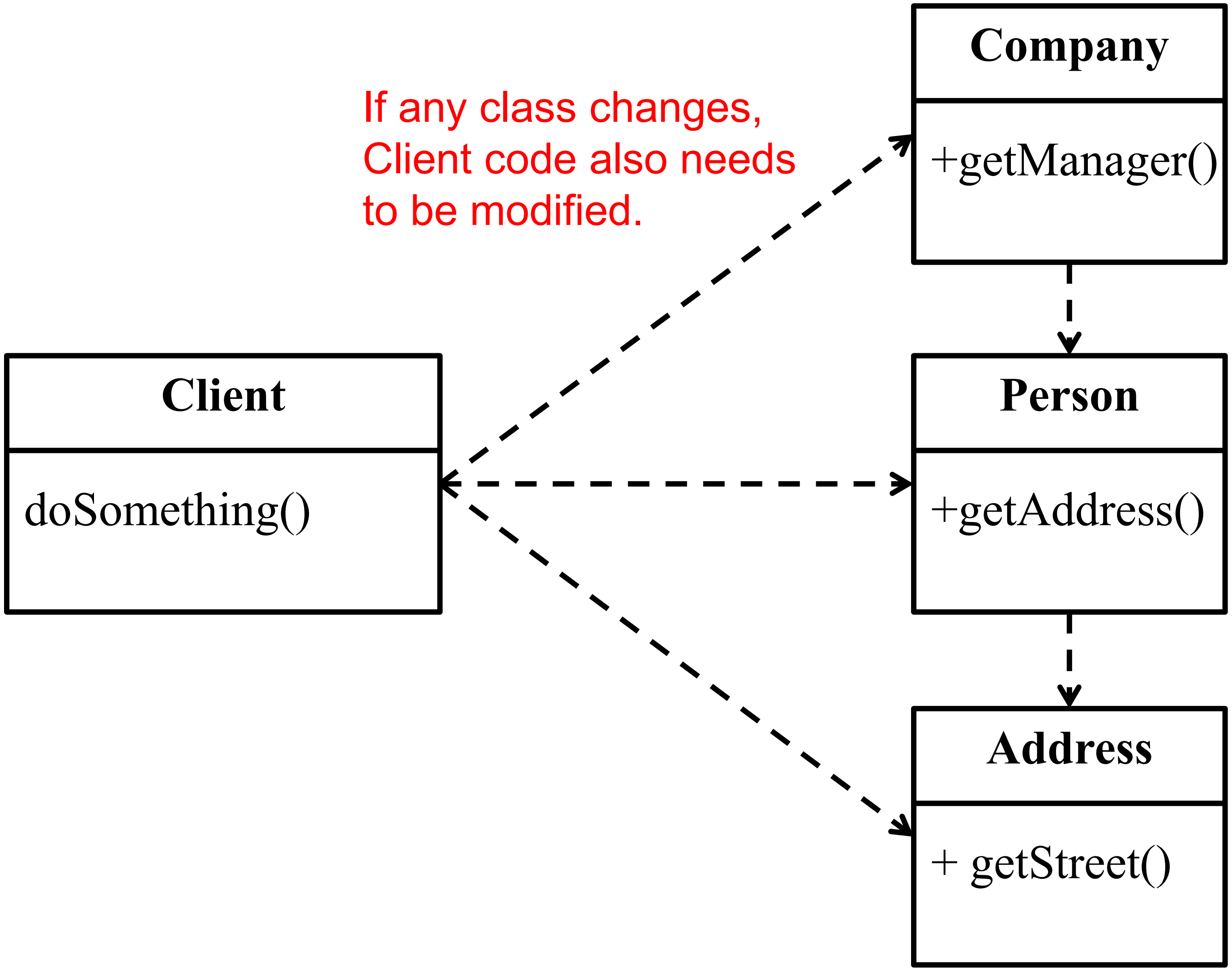
```java
public class Address {
    private String street;
    public Address(String street) {
        this.street = street;    }
    public String getStreet() {
        return street;    }
}
```

Example

```
public class Client {
    public void doSomething(Company company){
        // Message Chain
        String street = company.getManager().getAddress().getStreet();
        System.out.println("Manager's street: " + street);
    }
}
```

Example

If any class changes,
Client code also needs
to be modified.



重構方法1：新增Method

- You can refactor the code by introducing methods that hide the message chain and expose only necessary information.

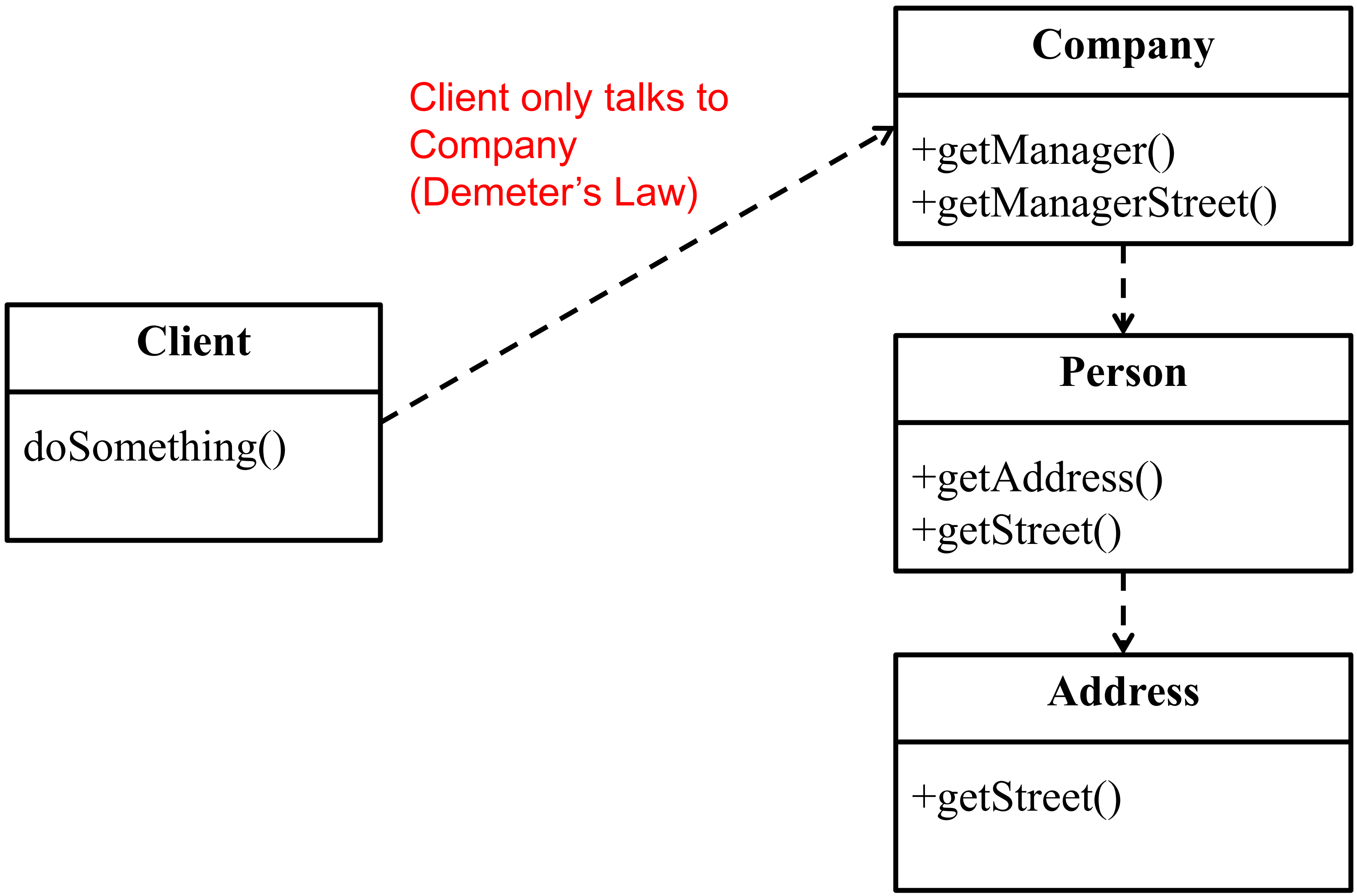
```
public class Company {
    private Person manager;
    public Company(Person manager) {
        this.manager = manager;    }
    public Person getManager() {
        return manager;    }
    public Person getManagerStreet(){
        return manager.getStreet(); }
}
```

```
public class Person {
    private Address address;
    public Person(Address address) {
        this.address = address;    }
    public Address getAddress() {
        return address;    }
    public Address getStreet() {
        return address.getAddress();}
}
```

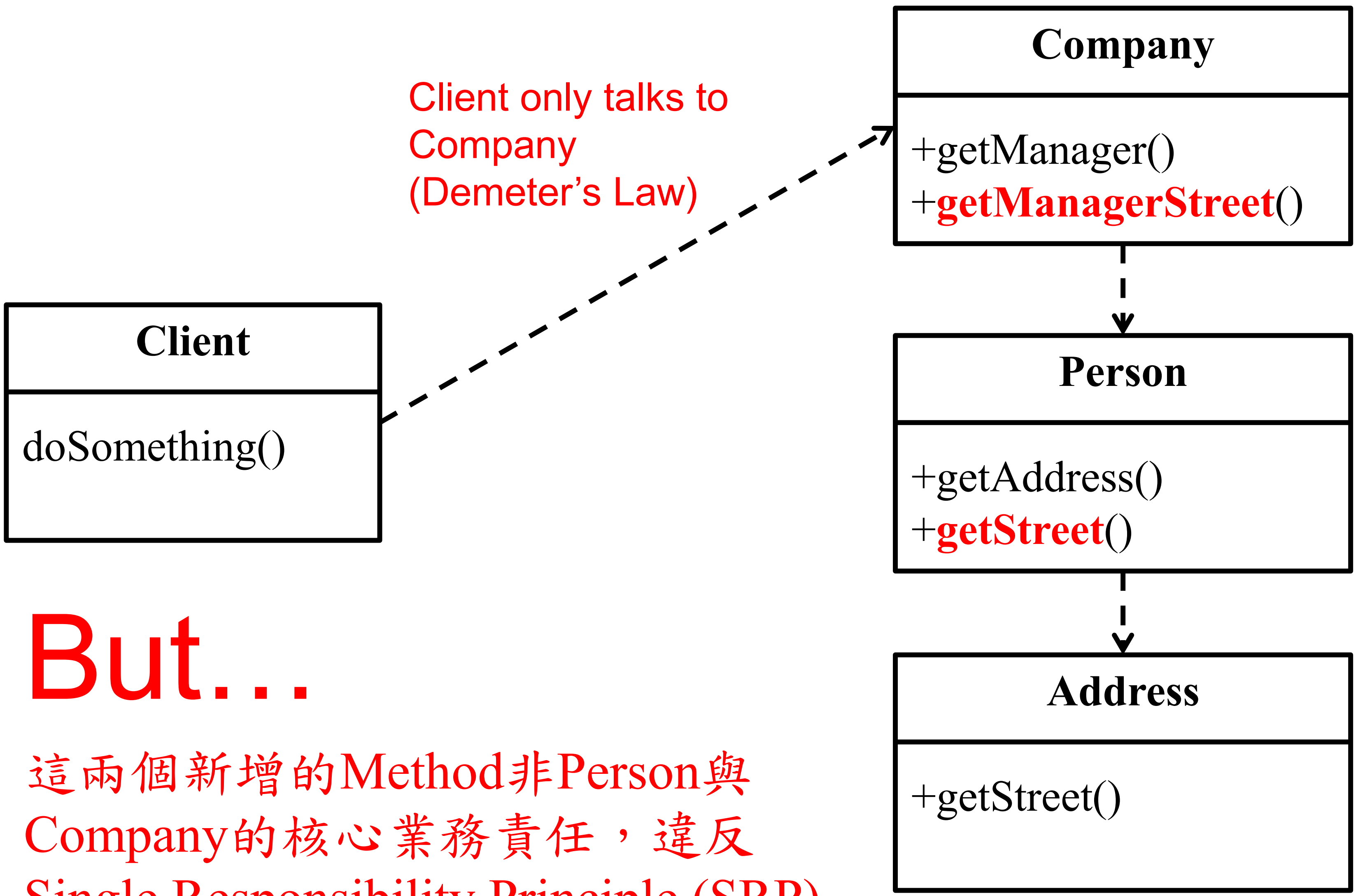
重構方法1

```java
public class Client {
    public void doSomething(Company company)
        // Message Chain
        String street = company.getManagerStreet();
        System.out.println("Manager's street: " + street);
    }
}
```

重構方法1


Client only talks to Company (Demeter's Law)
Company
+getManager()
+getManagerStreet()
Client
doSomething()
Person
+getAddress()
+getStreet()
Address
+getStreet()

重構方法1



But…

這兩個新增的Method非Person與Company的核心業務責任，違反Single Responsibility Principle (SRP)

重構方法2：新增中介Class (Façade Pattern)

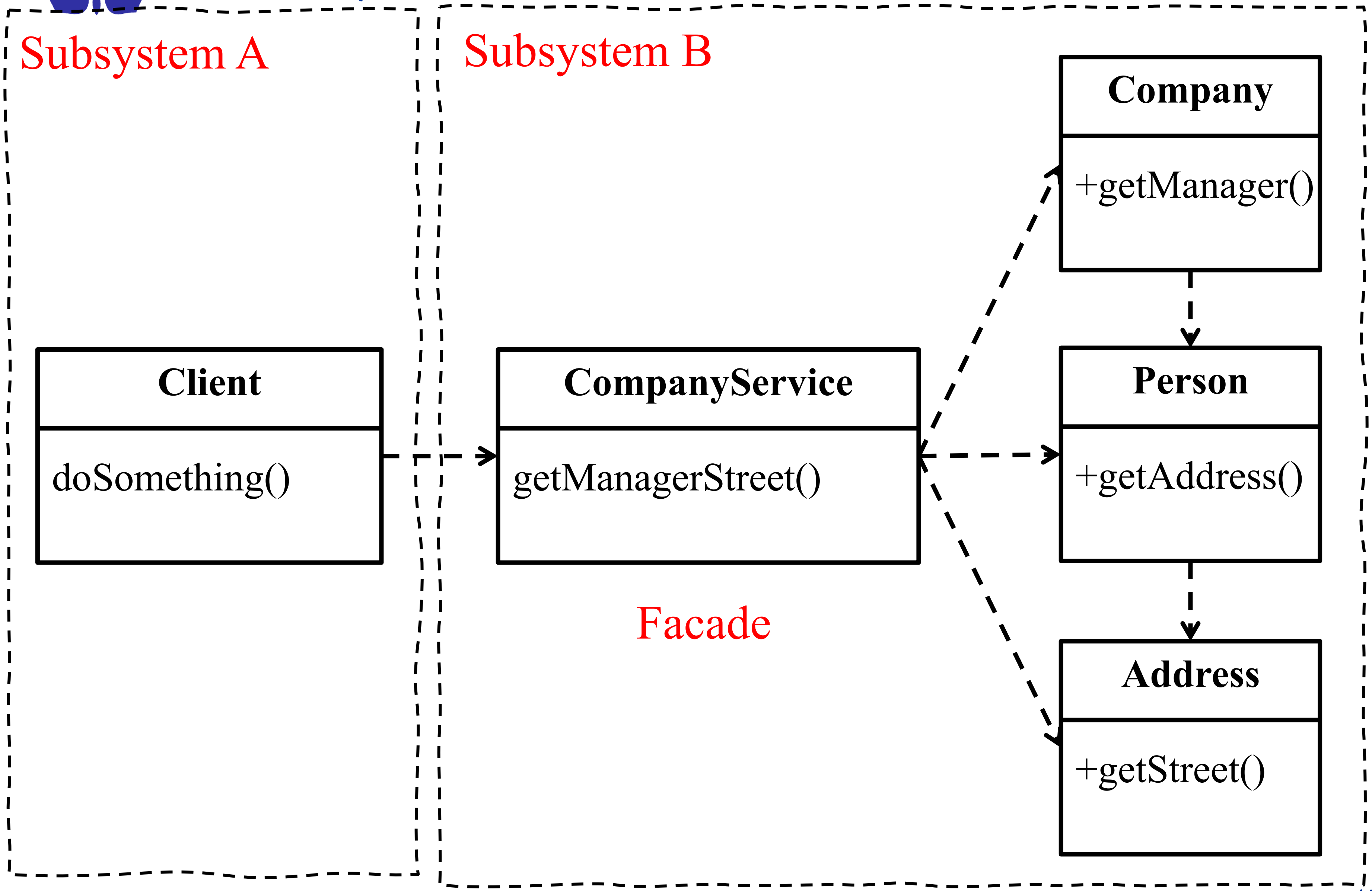
- 原本Address, Person, Company保持不變
- 新增一個CompanyService提供Message Chain服務讓Client呼叫

```
package company;
public class CompanyService {
    private static Company company;
    public String getManagerStreet(){
        String street = company.getManager().getAddress().getStreet();
        return street;
    }
}
```

重構方法2：新增中介Class (Façade Pattern)

```
package client;
import company.CompanyService;
public class Client {
    public void doSomething(){
        CompanyService cservice = new CompanyService();
        String street = cservice.getManagerStreet();
        //…
    }
}
```

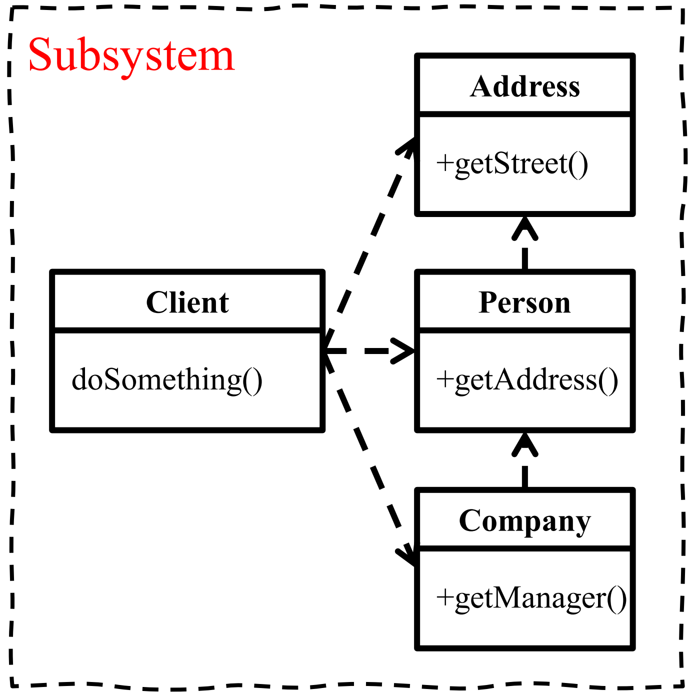
重構方法2

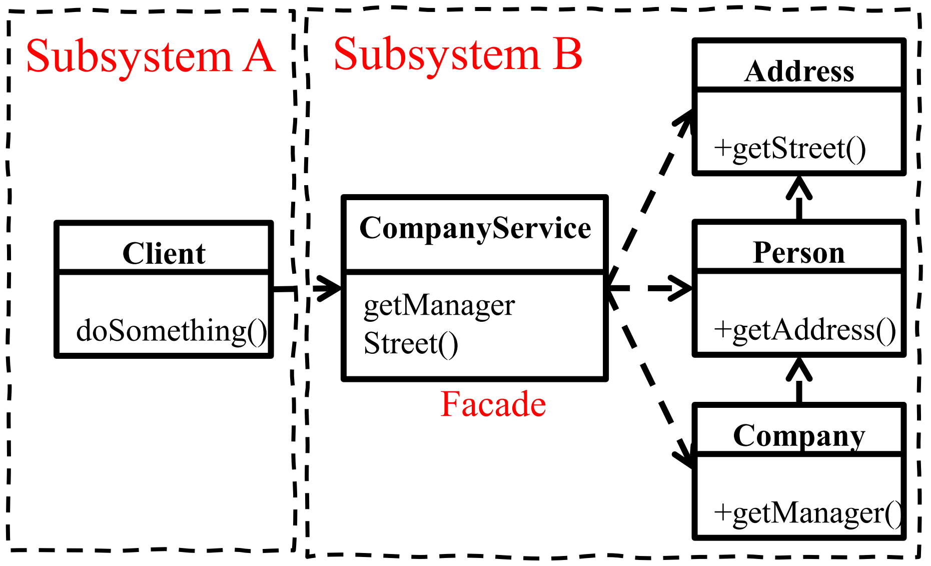

Subsystem A
Subsystem B
Client
doSomething()
CompanyService
getManagerStreet()
Facade
Company
+getManager()
Person
+getAddress()
Address
+getStreet()

Message Chain重構總結

- 如果Message Chain發生在**同一個Subsystem內**，保持原設計
- 符合Single Responsibility Principle

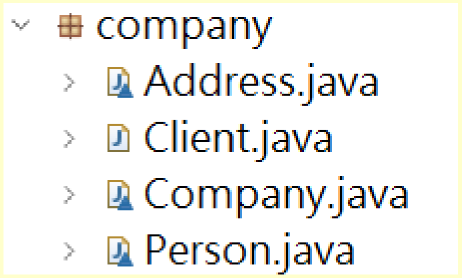
- 如果Message Chain發生在**跨Subsystem間**，採Façade Pattern
- 符合Single Responsibility Principle，亦符合Demeter’s Law (Don’t talk to strangers)

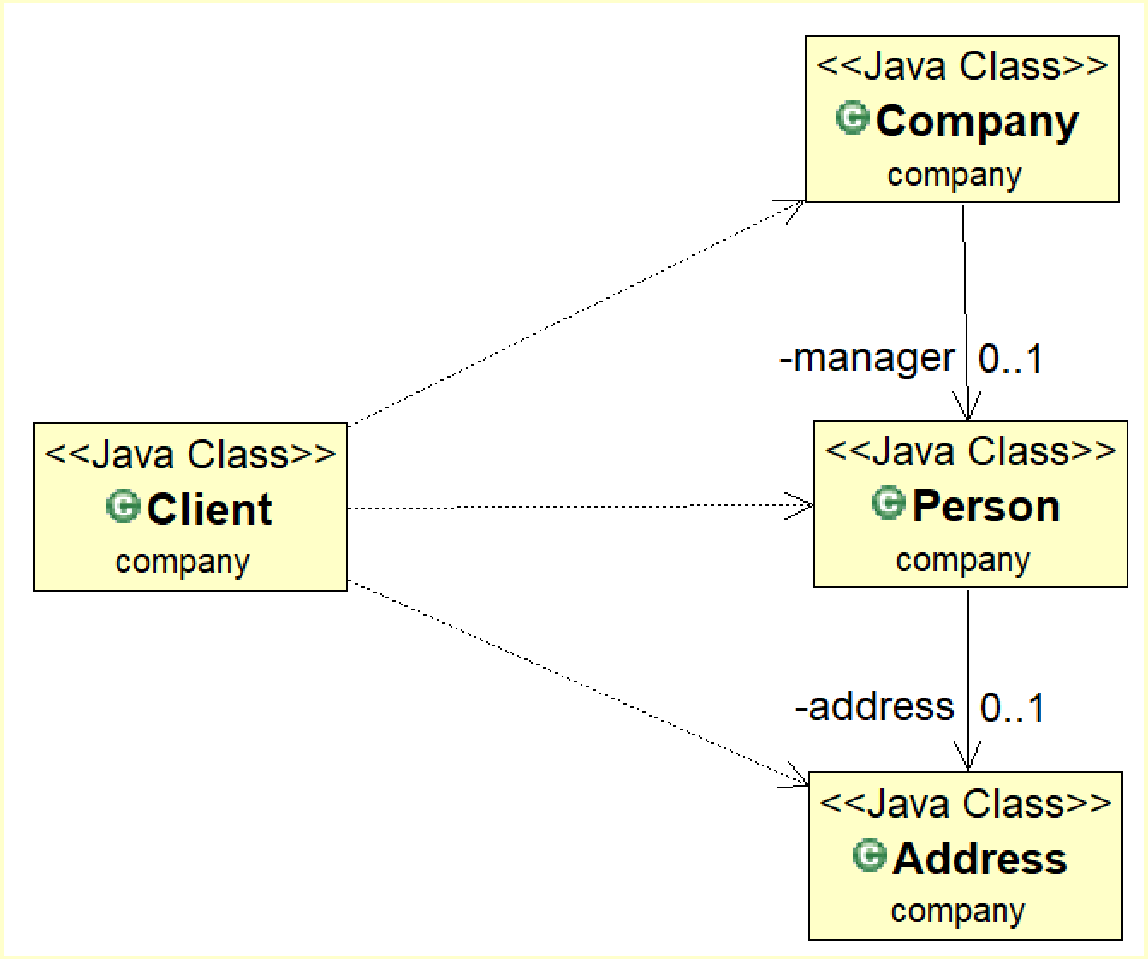




Lab

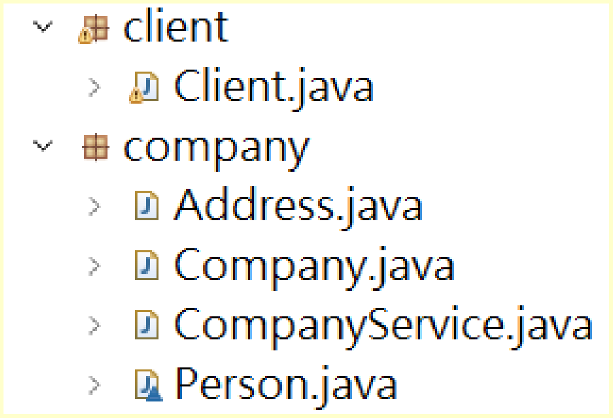
- 首先，請繪製出上述範例初始的class diagram



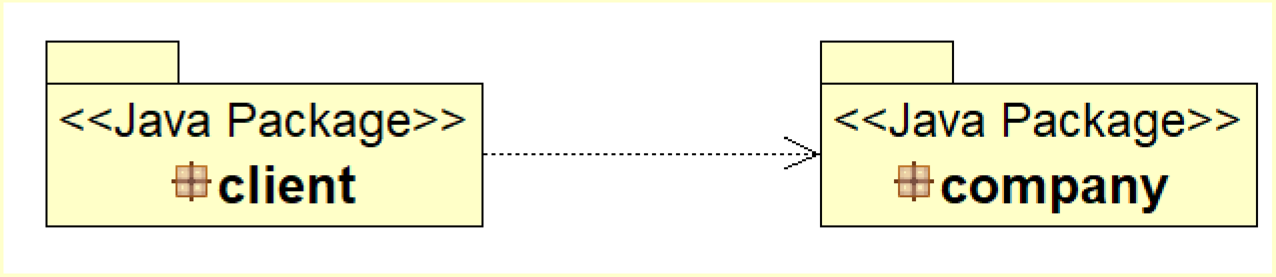


Lab (重構)

- 接著，依據【重構方法2】繪製出兩個package間的關係圖(這兩個package假設可視為兩個subsystem)

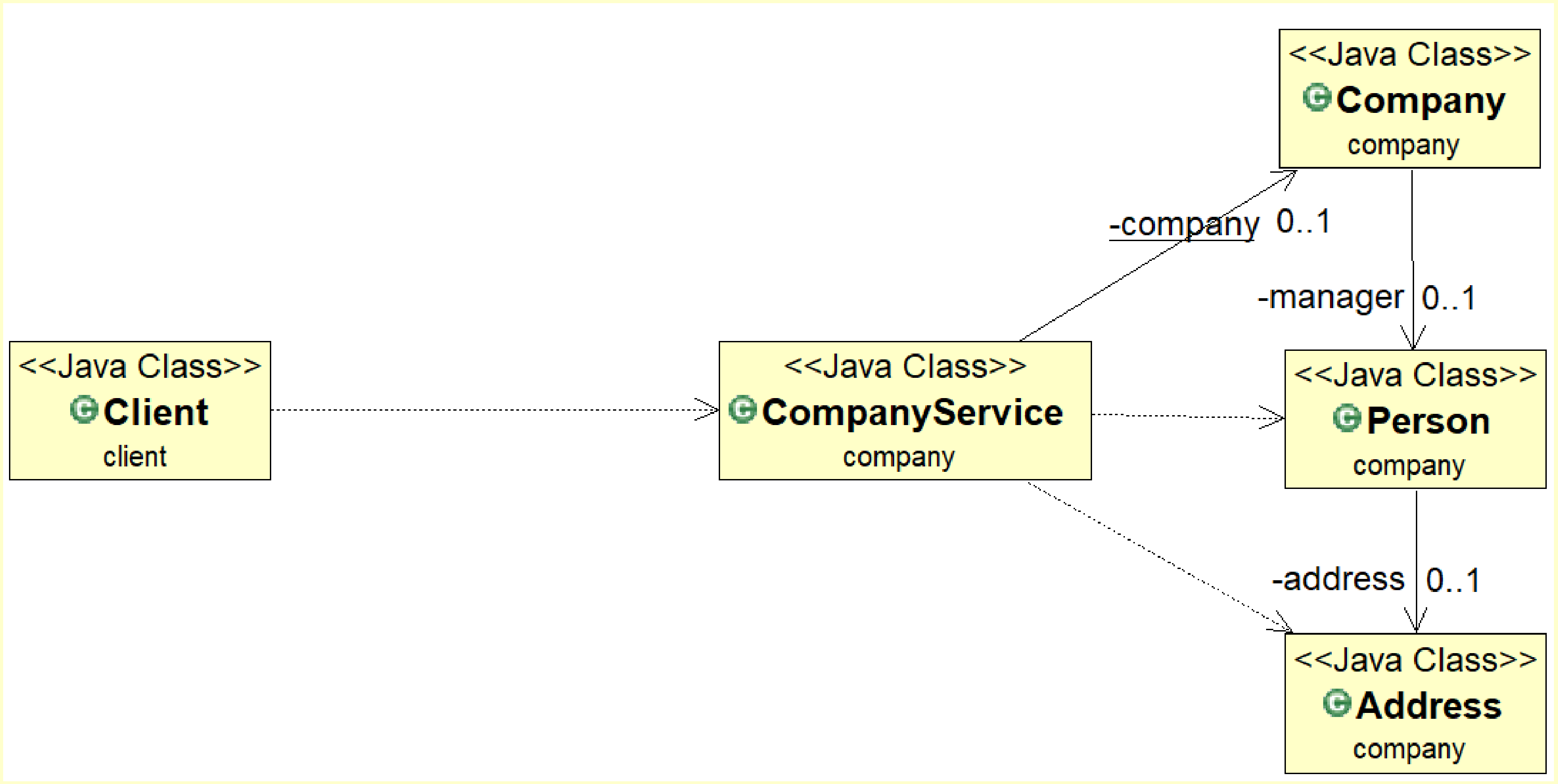


```
client
  Client.java
company
  Address.java
  Company.java
  CompanyService.java
  Person.java
```



Lab (重構)

- 繪製出這兩個package的cross-package class diagram



Literal constants

- To use keyword (*static*) *const* or *define* to define constants

- Not good:

```
public double potentialEnergy(double mass, double height) {
   return mass * 9.81 * height;
}
```

- Better solution:

```
public double potentialEnergy(double mass, double height) {
   final static double GRAVITATION = 9.81;
   return mass * GRAVITATION * height;
}
```

Every variable is properly initialized

- Not good:

```
Person person;
Manager = person.getManager();
int workHours, hourlyWage;
Int salary = workHours * hourlyWage;
```

- Better solution:

```
Person person = new Person();
Manager = person.getManager();
int workHours = 40, hourlyWage = 120;
Int salary = workHours * hourlyWage;
```

There are uncalled or unneeded procedures or any unreachable code

- ❑ Uncalled, unneeded, or unreachable code may occupy unnecessary memory

- ❑ Time and effort may be spent maintaining and documenting a piece of code which is in fact unreachable.

There are uncalled or unneeded procedures or any unreachable code

```
if(i < 60) {
    //unreachable
    if(i == 60) {
        System.out.println(“PASS”);
    }
    else{
        System.out.println(“NOT PASS”);
    }
}
else{
    System.out.println(“PASS”);
}
```

```
public class Client {
    public createMember(Name name)
    {
        Name name = new Name();
        Member.createMember(name);
    }
}
```

```
public class Member {
    public Member createMember(
        Name name
    ) {…}
    //uncalled or unneeded procedure
    public Member createMember(
        String lastName,
        String firstName,
    ) {…}
}
```

Does every switch statement have a default?

❑ Every switch-case should define a default action

- Not good:

```
switch(weekday) {
   case ‘Monday’:
      System.out.println(“國文課”);break;
   case ‘Tuesday’:
      System.out.println(“英文課”);break;
   case ‘Thursday’:
      System.out.println(“數學課”);break;
}
```

- Better solution:

```
switch(weekday) {
   case ‘Monday’:
      System.out.println(“國文課”);break;
   case ‘Tuesday’:
      System.out.println(“英文課”);break;
   case ‘Thursday’:
      System.out.println(“數學課”);break;
   default:
      System.out.println(“休息”);break;
}
```

The code avoids comparing floating-point numbers for equality

- ❑ Suggest to prevent comparing two floating-point numbers
- ❑ Not good:

```
double x = 1e-10, y1 = 20e-10, y2 = 19e-10;
double y = y1 – y2;
if(x == y) {
    System.out.println("X == Y");//並不會成立
}
```

- Better solution:

```
double x = 1e-10, y1 = 20e-10, y2 = 19e-10;
double y = y1 – y2;
if(Math.abs(x – y) < 1e-5) {
    System.out.println("X == Y");//成立
}
```

All comments are consistent with the code

- Not good:

```
// 計算一年獲利, 傳入參數(int amount)
public void countProfit(int amount, double rate) {
    _profit = amount * ( 1 + rate );
}
```

- Better solution:

```
// 計算一年獲利, 傳入參數(int amount, double rate)
public void countProfit(int amount, double rate) {
    _profit = amount * ( 1 + rate );
}
```

Divergent Change (發散式改變)

Divergent Change (發散式改變)

- Divergent change：一個類別會因為因應太多的變更原因而需修改
- 可透過*Extract Class*操作來進行重構
  - 將不同的行為抽出至不同的Class

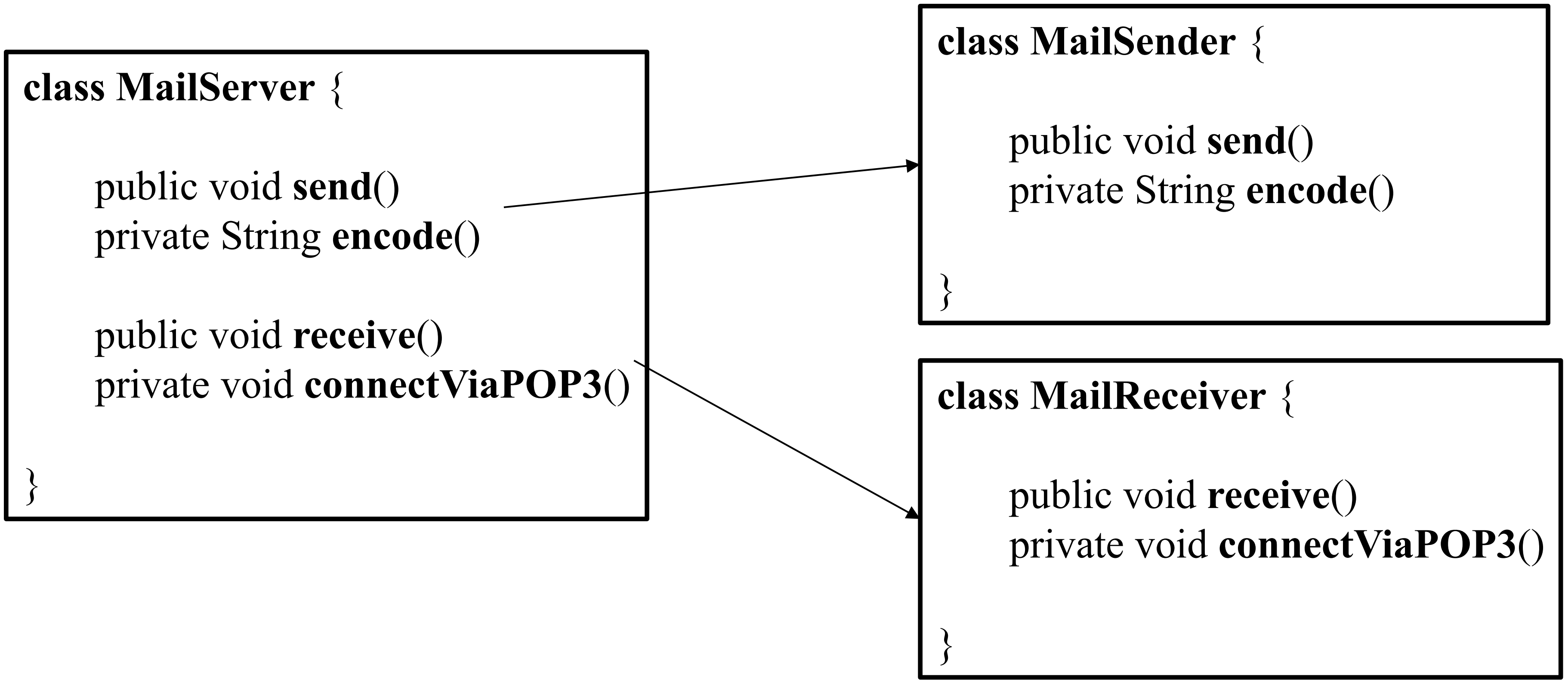
範例：此Class因二種不同行為而需變更

```
class MailServer {

   public void send(String to, String content){
      //…
   }
   private String encode(String content){
      // encode content;
   }

   public void receive(String account) {
      connectViaPOP3();
      // …
   }
   private void connectViaPOP3(){
      // connect to a server via POP3 protocol;
   }
}
```

可預想這兩個Methods會因為「寄信」行為改變(如需加入Encryption)而需變更。

可預想這兩個Methods會因為「收信」行為改變(如需加入IMAP協定)而需變更。

Refactoring by *Extract Class*



```
class MailServer {

    public void send()
    private String encode()

    public void receive()
    private void connectViaPOP3()

}
```

```
class MailSender {

    public void send()
    private String encode()

}
```

```
class MailReceiver {

    public void receive()
    private void connectViaPOP3()

}
```

Refactoring後遵循Single Responsibility Principle

- Single Responsibility Principle: 每個Class必須專注於提供整個系統中單一部分的功能，使得Class更Robust。每個Class必須僅因一個理由而有所修改。

- 在實務上，判定是否滿足此原則是主觀的。如果你瞇著眼仔細檢視程式碼，會發現一個Class常常存在因為多個理由而需修改，因此建議檢視**是否同一個Class中的Methods相互依賴或共用屬性**，若是，則內聚力較高。

```
class MailSender {

    public void send()
    private String encode()

}
```

```
class MailReceiver {

    public void receive()
    private void connectViaPOP3()

}
```

Shotgun Surgery (散彈式修改)

Shotgun Surgery (散彈式修改)

- Shotgun Surgery：每次為因應同一種變更，你必須同時在許多類別上做出許多修改。
  - 當有太多需修改的地方時，將造成難以尋找所有需修改處，並容易遺漏。
  - 常發生於Copy and Paste Programming

- 可透過*Extract Method*, *Move Method*或*Move Field*來進行重構
  - 將所有需修改之Methods或Fields移至一個類別，若無存在既有適合的類別，可建立一個新的類別

Duplicate Code引起的Shotgun Surgery

```
class UserNameUtil {
    public void getUserNames() {
        Class.forName(“xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT usrname FROM names");
        //…     }
}
```

```
class BookUtil {
    public void getUserNames() {
        Class.forName(“xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT book FROM books");
        //…     }
}
```

```
class StoreUtil {
    public void getStores() {
        Class.forName(“xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT store FROM stores");
        //…     }
}
```

若欲變更Driver或資料庫名稱時，這三個類別中的此二行程式碼皆需一併修改，若有遺漏則會造成連線錯誤。

Refactoring by *Extract Method*

```
class UserNameUtil {
    public void getUserNames() {
        Class.forName("xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT usrname FROM names");
        //...    }
}
```

```
class BookUtil {
    public void getUserNames() {
        Class.forName("xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT book FROM books");
        //...    }
}
```

```
class StoreUtil {
    public void getStores() {
        Class.forName("xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT store FROM stores");
        //...    }
}
```

```
class DBConnection {
    public static Connection getConnection() {
        Class.forName("xxxDriver").newInstance();
        Connection conn = DriverManager.
                    getConnection("BookDatabase");
        return conn;
    }
}
```

After Refactoring

```
class UserNameUtil {
    public void getUserNames() {
        Connection conn = DBConnection.getConnection();
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT usrname FROM
names");
        //…     }
}
```

```
class BookUtil {
    public void getUserNames() {
        Connection conn = DBConnection.getConnection();
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT book FROM books");
        //…     }
}
```

```
class StoreUtil {
    public void getStores() {
        Connection conn = DBConnection.getConnection();
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT store FROM stores");
        //…     }
}
```

```
class DBConnection {
    public static Connection getConnection() {
        Class.forName("xxxDriver").newInstance();
        Connection conn = DriverManager.
                getConnection("BookDatabase");
        return conn;
    }
}
```

非Duplicate Code引起的Shotgun Surgery

```
public class Button {
  private String color;
  public Button(String theme) {
    if (theme.equals("Light")) {
      color = "White";
    } else if (theme.equals("Dark")) {
      color = "Black";
    }
  }
}
```

```
public class TextBox {
  private String bgColor;
  public TextBox(String theme) {
    if (theme.equals("Light")) {
      bgColor = "Light Gray";
    } else if (theme.equals("Dark")) {
      bgColor = "Dark Gray";
    }
  }
}
```

```java
public class ThemeApp {
  public static void main(String[] args) {
    ThemeApp app = new ThemeApp();

    String theme = "Dark";

    Button button = new Button(theme);
    TextBox textBox = new TextBox(theme);
  }
}
```

Lab

- 重構ThemeApp

Primitive Obsession

- Using primitives instead of small, dedicated objects for simple tasks
  - **Loss of Type Safety**: It becomes easy to mistakenly assign a value to an incorrect field since primitives lack specific type distinctions.
  - **Lack of Encapsulated Behavior**: Primitives cannot contain any logic or behavior, so all related logic must be handled within the containing class, leading to less cohesive code.

```java
public class Order {
    private String customerName;
    private String address;
    private String postalCode;

    public Order(String customerName, String address, String postalCode) {
            this.customerName = customerName;
            this.address = address;
            this.postalCode = postalCode;
    }
}
```

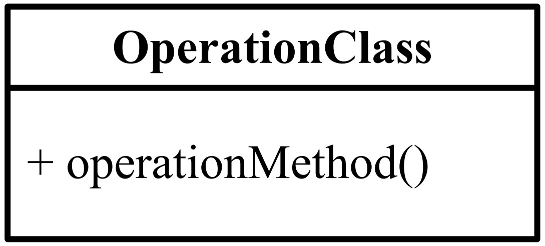
Replacing Primitives with (Value) Objects

```
public class PostalCode {
    private String code;
    public PostalCode(String code) {
        if (!isValidPostalCode(code)) //throw exception;
        this.code = code;
    }
    private boolean isValidPostalCode(String code) {
        return code != null && code.matches("\\d{5}");
    }
}
```

```
public class Order {
    private String customerName;
    private String address;
    private PostalCode postalCode;

    public Order(String customerName, String address, PostalCode code) {
        this.customerName = customerName;
        this.address = address;
        this.postalCode = code;
    }
}
```

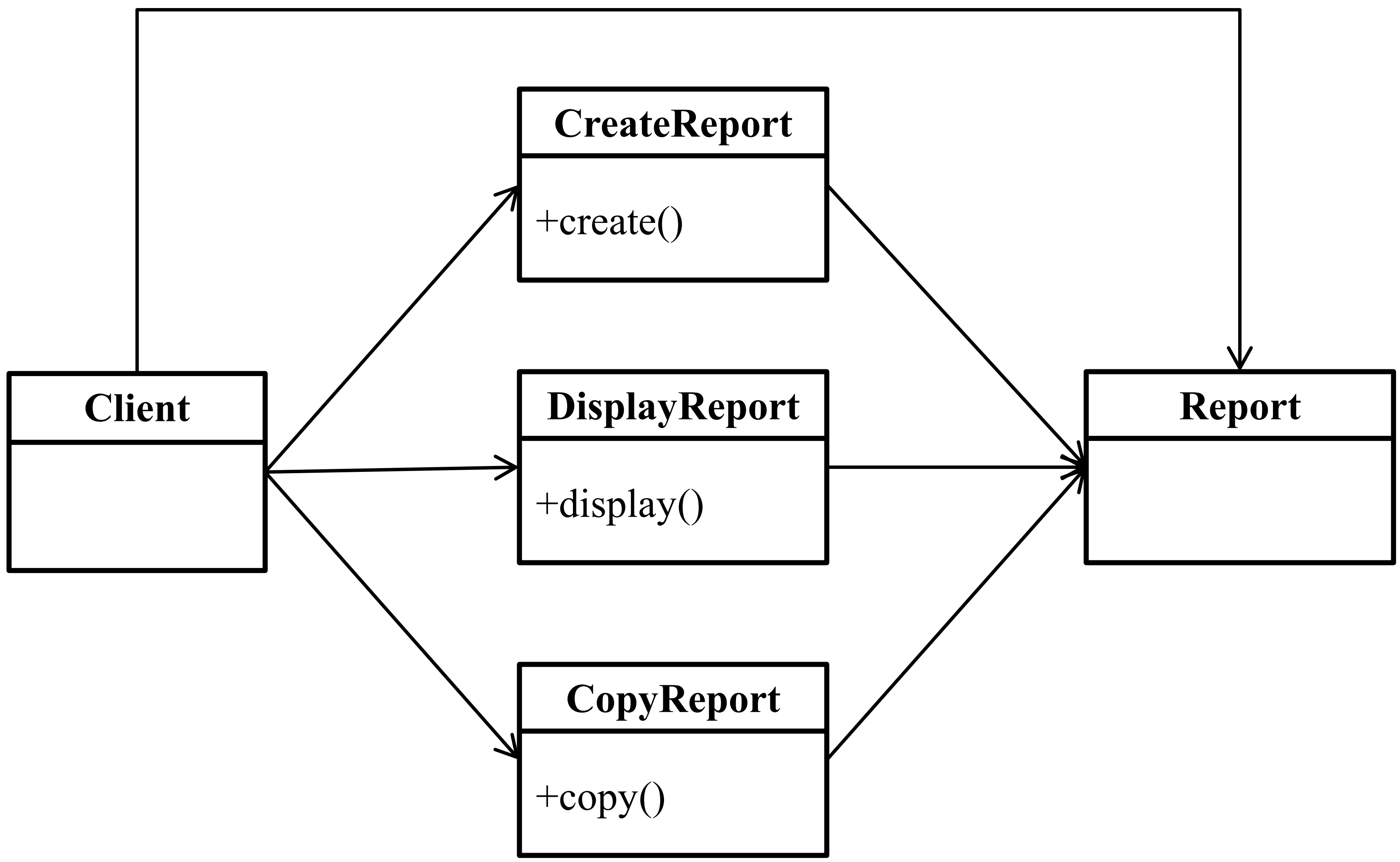
Operation Class

- Class Name通常為動詞(CreateReport)，而非物件名詞(Report)
- 通常一個Class包含只有一個Method
- 由於Class Name已經限制了語意，因此很難再擴充Method，造成須相對創建了許多Class
- 由於Class Name為功能特性思維去命名，因此較難以物件導向思維去創建繼承關係以及動態多型的優勢

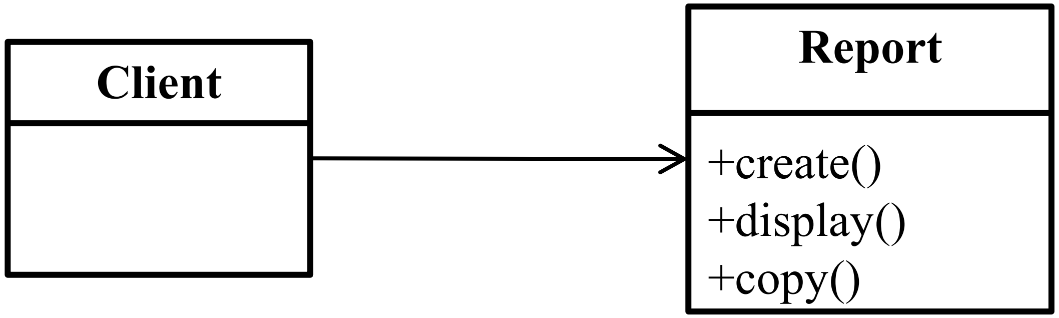


This anti-pattern is described in
- Llano, Maria Teresa, and Rob Pooley. "UML specification and correction of object-oriented anti-patterns." *2009 Fourth International Conference on Software Engineering Advances*. IEEE, 2009.
- Suryanarayana, Girish, Ganesh Samarthyam, and Tushar Sharma. Refactoring for software design smells: managing technical debt. Morgan Kaufmann, 2014.

An Operation Class Example

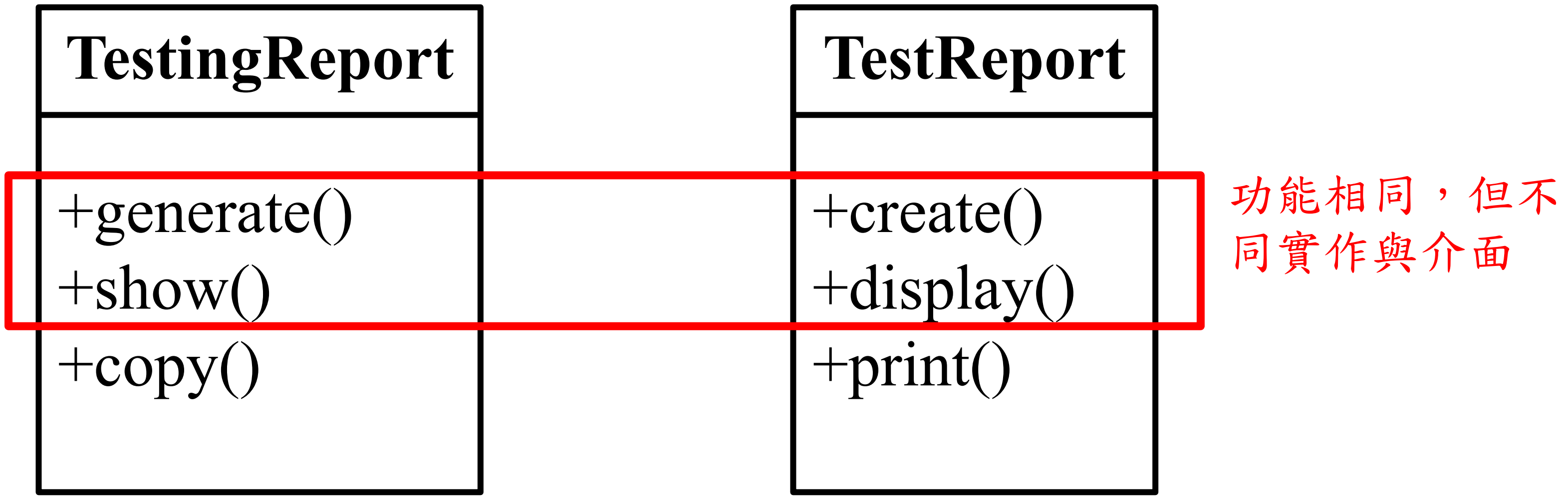


Refactored



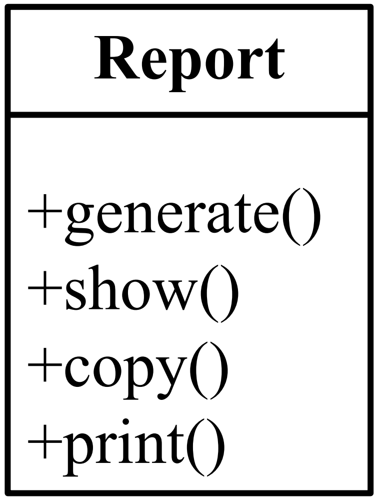
Alternative Classes with Different Interfaces

- The programmer who created one of the classes probably didn’t know that a functionally equivalent class already existed.



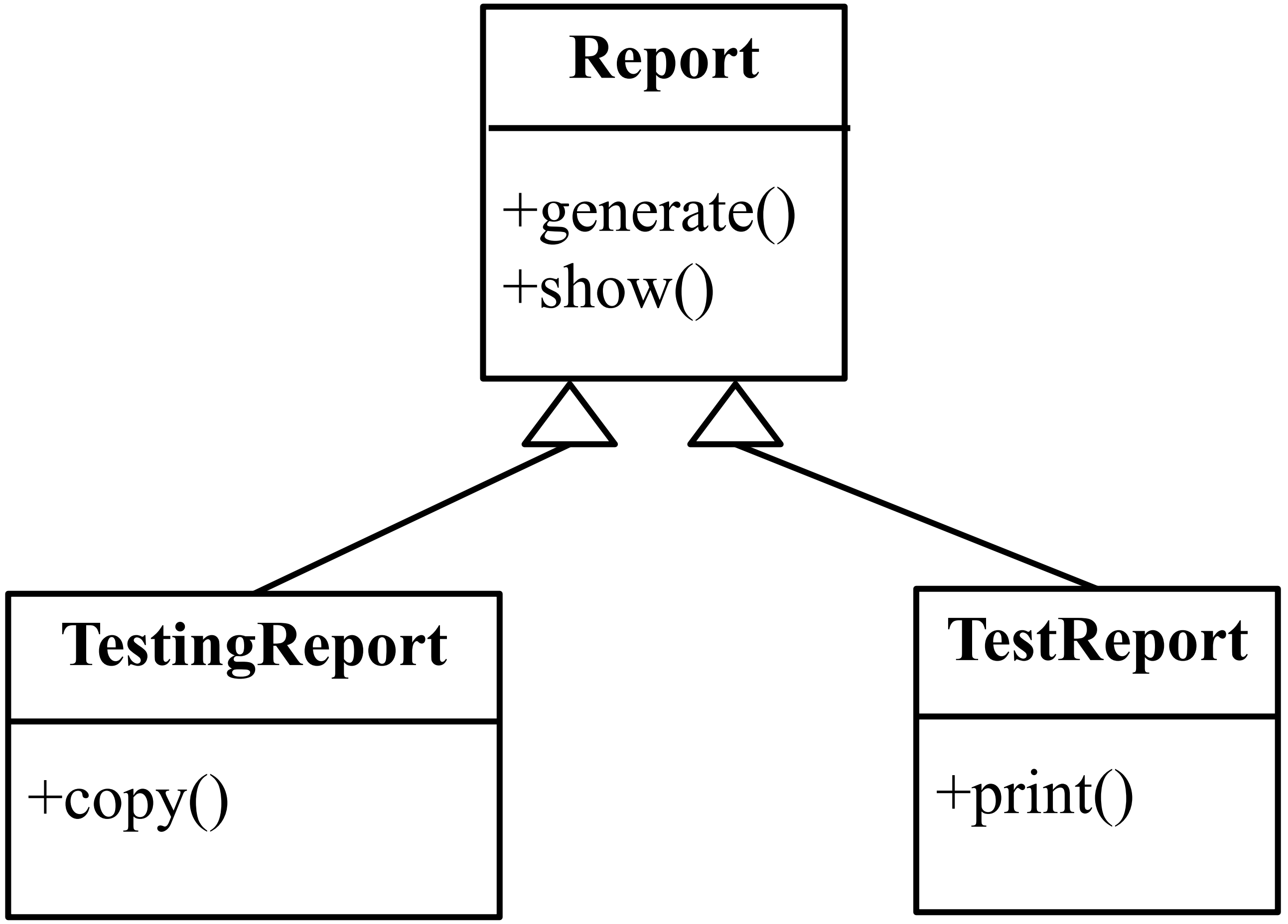
Refactored (若兩個Class在同一個Library)

取Method聯集，綜合出一個Class就好



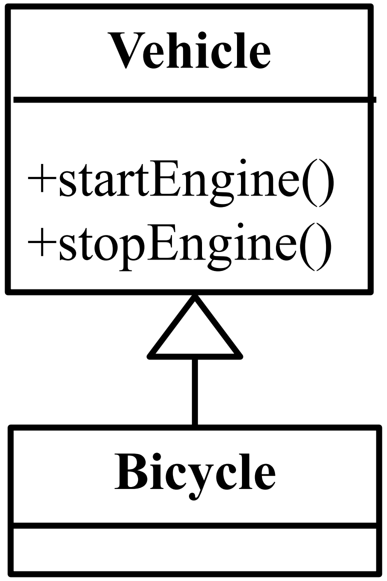
Refactored (若兩個Class在不同Library)

Extract Superclass：創建一個Super Class，將重複Method調整為單一通用Method，放置到Super Class

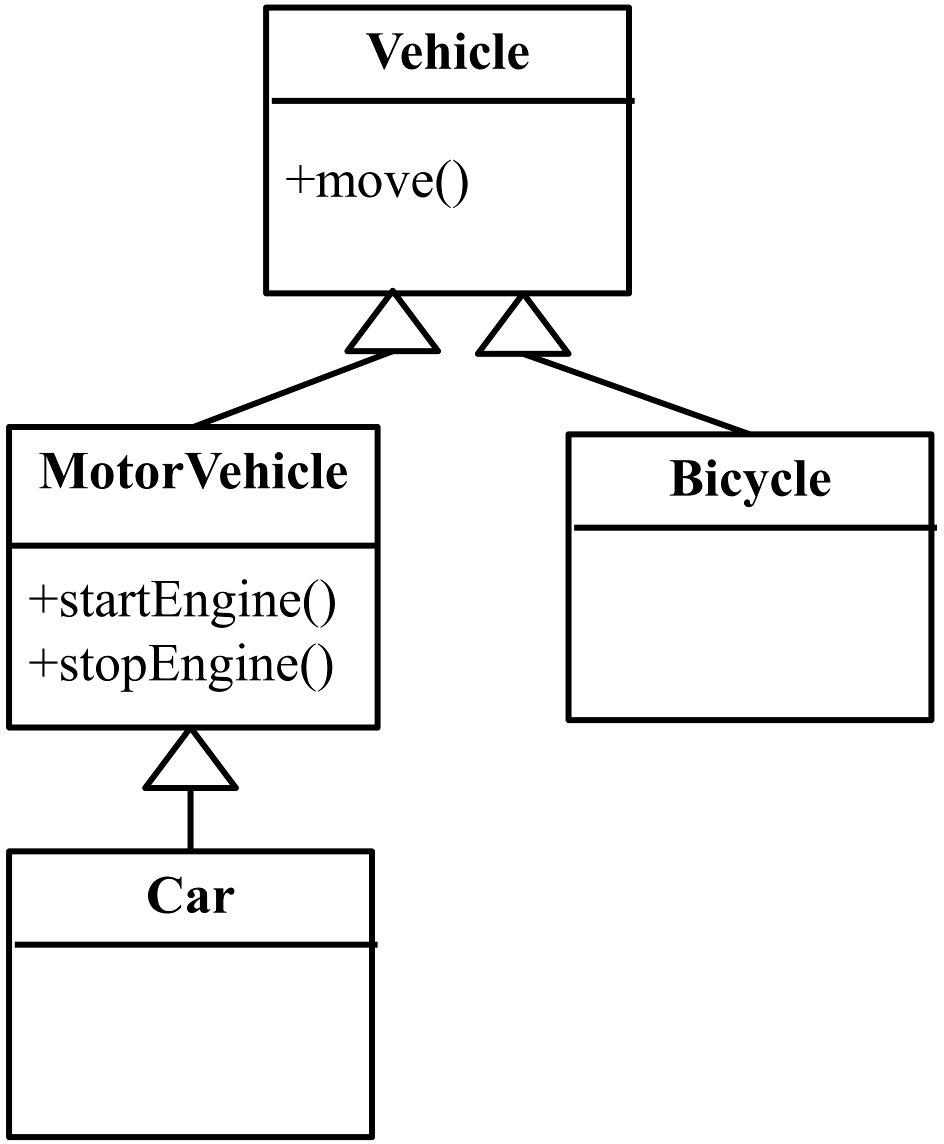


Refused Bequest

- If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter.
- The **unneeded methods** may simply go unused or be redefined and **give off exceptions**.
- 違反Liskov Substitution Principle (A subclass should be usable as a substitute for its superclass)，會讓動態多型不合理

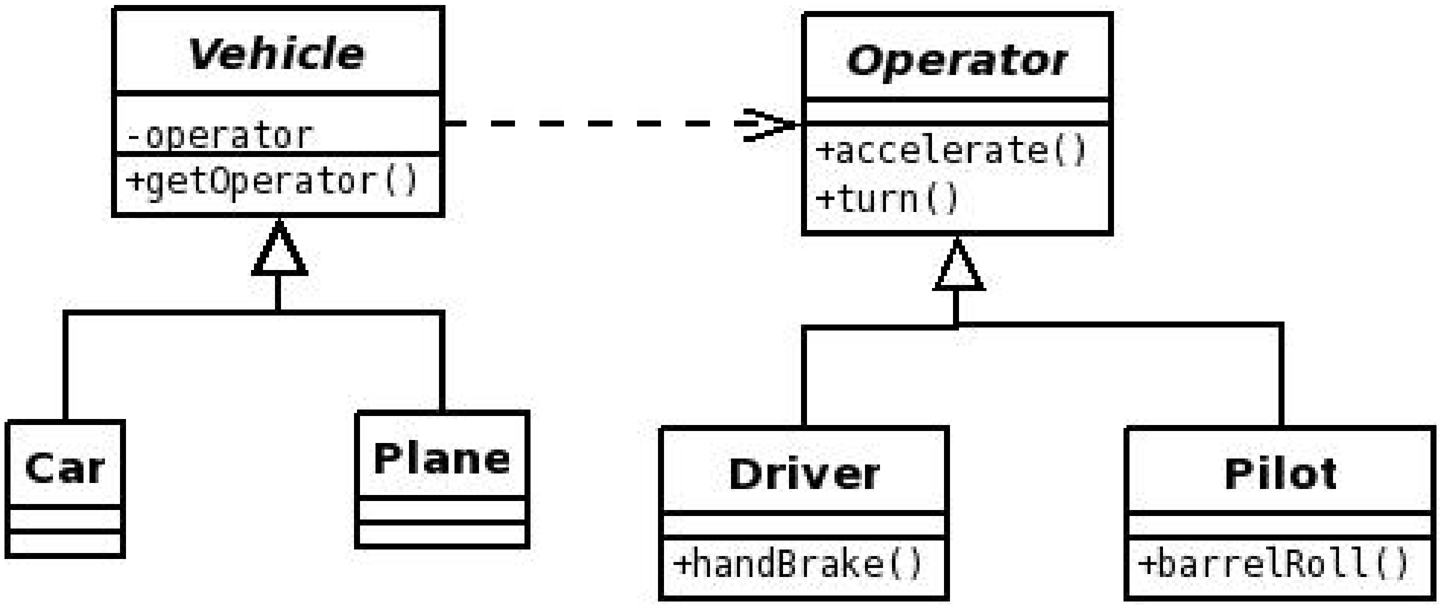


Refactored


Vehicle
+move()
MotorVehicle
+startEngine()
+stopEngine()
Bicycle
Car

Parallel Inheritances Hierarchies

- Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.
- 問題：無法滿足兩個樹底下的物件互相有特定配對依賴關係的要求



一個Car物件的operator屬性狀態可能會被設定為一個Pilot物件

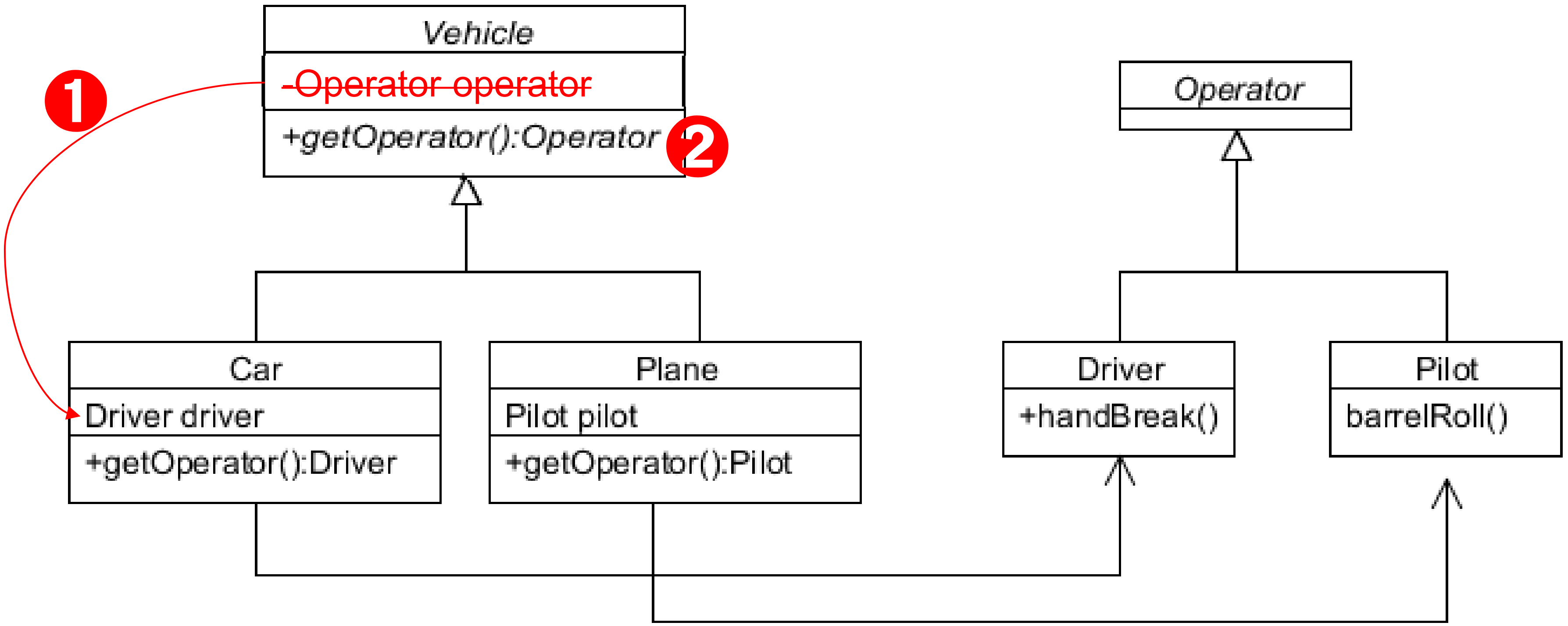
From: https://oowisdom.csse.canterbury.ac.nz/index.php/Parallel_hierarchies_problem

Lab

- 以Java實作上述Vehicle-Operator設計，並以ObjectAid畫出Class Diagram

Refactoring by *Defer Identification of State Variables Pattern*

- 第一步(屬性降階層)
  - 將Vehicle的operator屬性移除，並在Car與Plane中各別加入欲配對的屬性型態
- 第二步(加Abstract Accessor)
  - 在Vehicle中加入getOperator (稱之為Abstract Accessor)讓Car與Plane實作，以達成維持原本Vehicle與Operator的關係



Lab

- 基於Defer Identification of State Vehicle Pattern 重構上述Vehicle-Operator案例的Java實作

Middle Man

- The Middle Man code smell occurs when a class primarily delegates work to other classes, providing little to no added value beyond forwarding method calls.
- If a class performs only one action, delegating work to another class, why does it exist at all?

Example

```java
class PaymentProcessor {
  public void processPayment(double amount) {
    System.out.println("Processing payment of amount: $" + amount);
  }
}
```

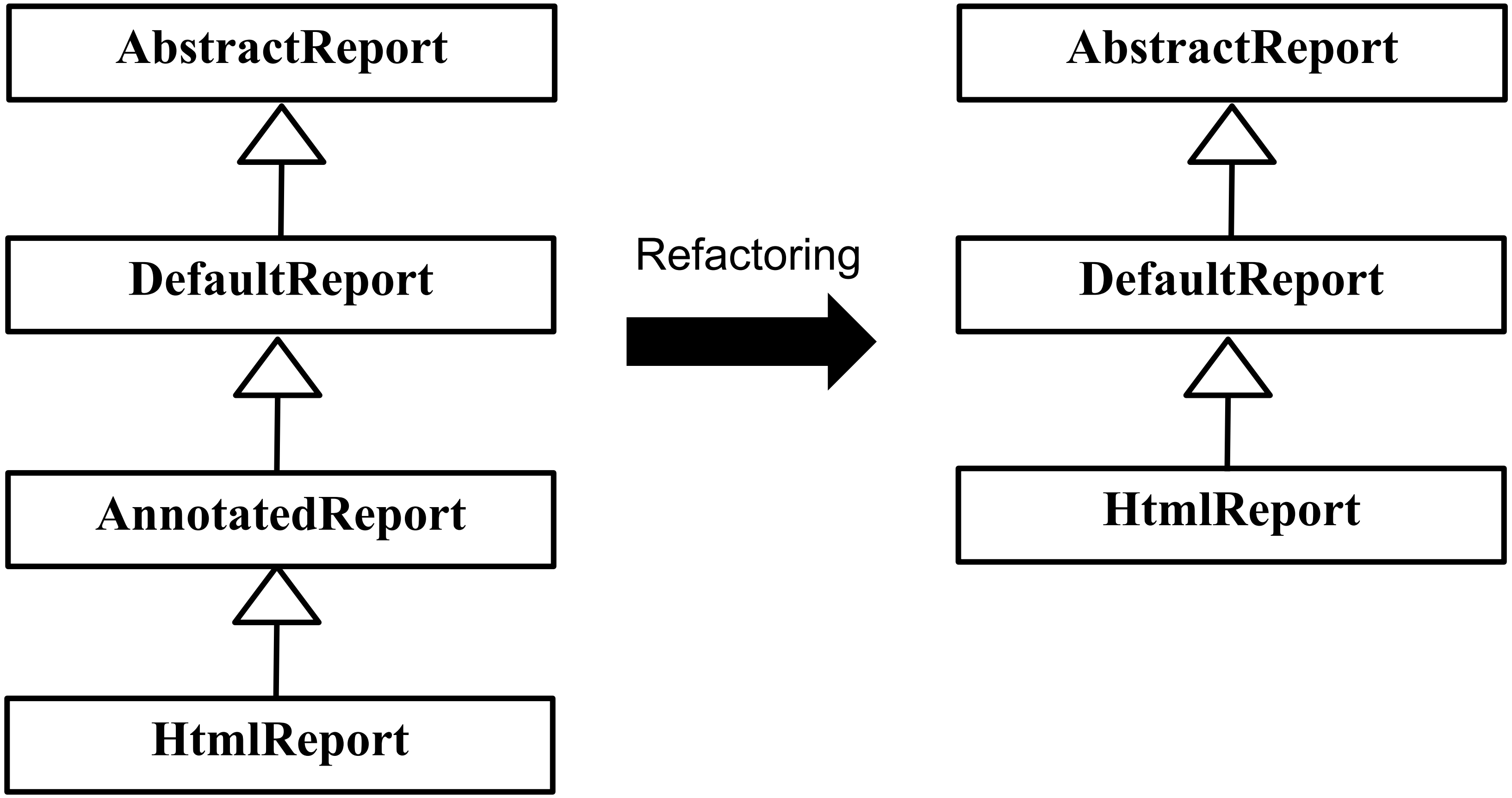
```java
//Middle Man class
class PaymentManager {
  private PaymentProcessor paymentProcessor;
  public   PaymentManager(PaymentProcessor paymentProcessor) {
    this.paymentProcessor =   paymentProcessor;
  }
  public void processPayment(double amount) {
    // Simply forwards the request to PaymentProcessor
    paymentProcessor.processPayment(amount);
  }
}
```

```java
//Client code
public class Main {
  public static void main(String[] args) {
    PaymentProcessor paymentProcessor = new PaymentProcessor();
    PaymentManager paymentManager = new   PaymentManager(paymentProcessor);

    // Client calls the method via the middleman
    paymentManager.processPayment(100.0);
  }
}
```

Speculative Generality

- ❑ Code is written to handle potential future requirements or extensions that are not currently needed.
- ❑ This often results in over-engineered solutions with unnecessary abstractions, interfaces, or patterns that add complexity without providing immediate benefits.

Example


AbstractReport
DefaultReport
AnnotatedReport
HtmlReport
Refactoring
AbstractReport
DefaultReport
HtmlReport