Code Smells and Refactoring

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

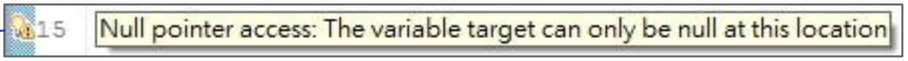
Refactoring 線上教材

- https://refactoring.guru
- https://www.industriallogic.com/xp/refactoring/catalog.html

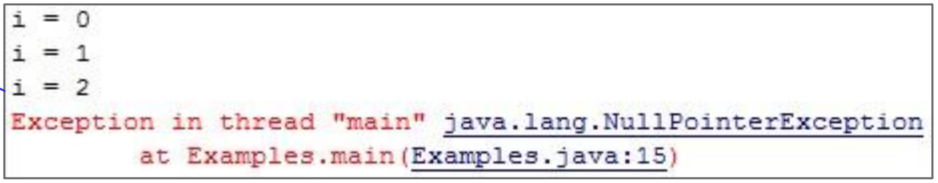
Unresolved warnings

- 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 classic smell is a method that seems more interested in a class other than the one it actually is in.

```
public void doSomething() {
    ClassA a = new ClassA();
    int x = a.getX();
    int y = a.getY();
    int z = a.calculateSomething(x +
y, y);
    a.setZ(z);
}
```

```
public ClassA() {
    public void doSomeThing() {
      z = calculateSomething(x + y, y);
    }
}
```

- Use *Move Method* to move the method to another class

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;
    }
}
```

All assigned variables have proper type consistency or casting (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.

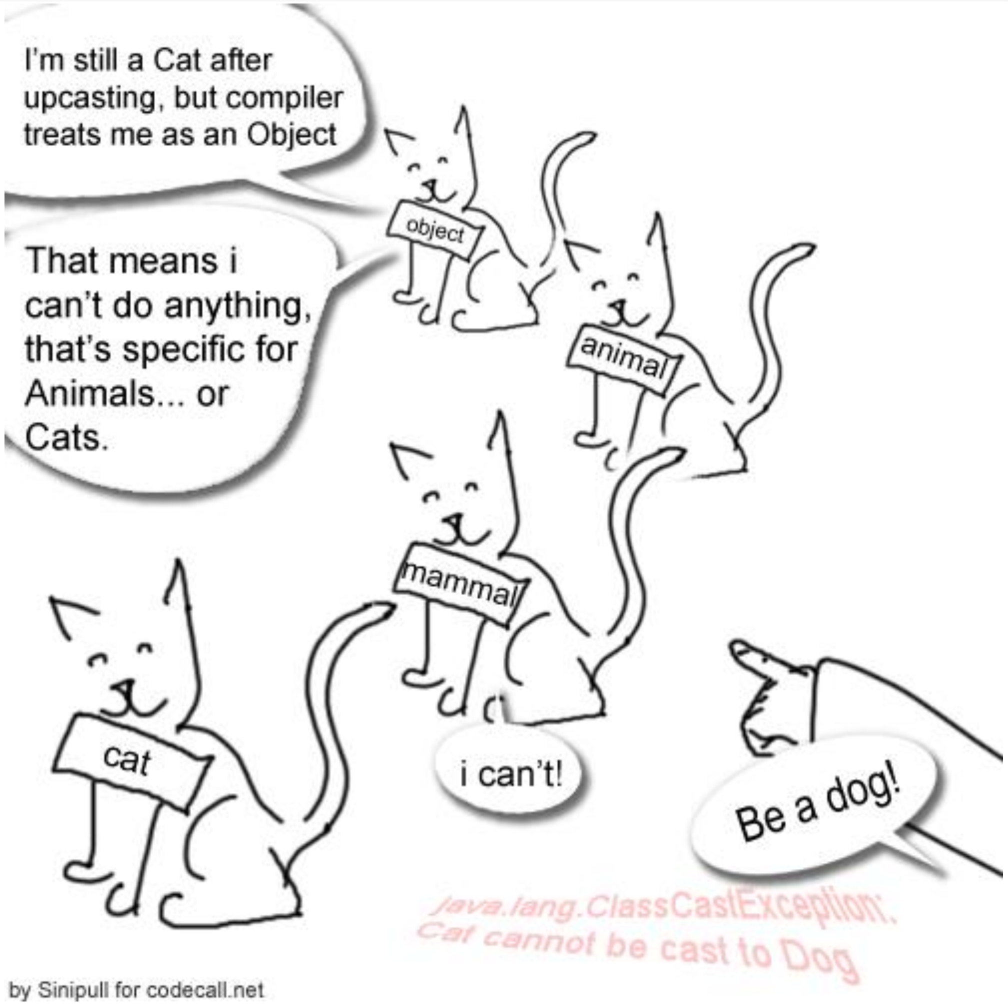
```
void testType() {
   unsigned short x = 65535;
   short y = x;

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

All assigned variables have proper type consistency or casting (2/2)

- Upcasting

```
class Animal() {}

class Mammal extends Animal()
{}

class Cat extends Mammal() {}

class Dog extends Mammal() {}
```

```
Mammal m = new Cat()
Dog c = (Dog)m;
```


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

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);
}
```

Fat View (1/2)

```
  // codes that create menus, buttons, and connects signals to slots (omitted)
MainWindown::loadMindMap() {
   /** open dialogue box that le                    h, and read the text
      * file using ifstream. Also,                   v the
      * varaibles that we need t
     */
  …
  while (fin.eof()) { // fin is a ifstream object.
    fin >> line;
    if (line == "//ROOTNODE") {
      fin >> nodeId >> nodeDescription;
      newRoot = new AbstractNode(nodeId, nodeDescription);
      fin >> coordinateX >> coordinateY >> width >> height;
      newRoot->setX(coordinateX);
      … // more bussiness logic
}
```

```
// ROOTNODE
0 MindMind_Topic
50 50 40 60
// NODE
1 10 Node_Description
150 0 40 60
…
```

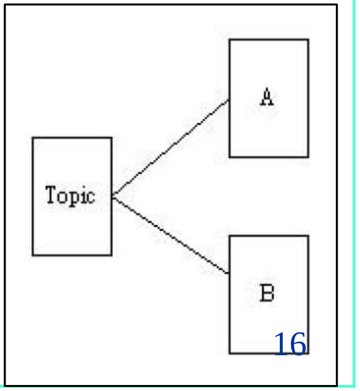
含

Fat View (2/2)

```
  // codes that create menus, buttons, and connects signals to slots (omitted)
MainWindown::loadMindMap() {
   /** open dialogue box that lets user specify a file path, and read the text
     * file using ifstream. Also, assume we have properly declared the
     * varaibles that we need to restore a mind map.
    */
  …
  m_mindMap->loadMindMap(filePath);
}
  … // more methods

```

```
void MindMap::loadMindMap(string filePath) {
  while (fin.eof()) {
    fin >> line;
    if (line == "//ROOTNODE") {
      fin >> nodeId >> nodeDescription;
      newRoot = new AbstractNode(nodeId, nodeDescription);
      fin >> coordinateX >> coordinateY >> width >> height;
      newRoot->setX(coordinateX);
      … // set coordinateY, width, and height for root node.
    } else if (line == //NODE) {
      fin >> parentId >> nodeId >> nodeDescription;
      … // more bussiness logic
}
  … // more methods
```

MindMap object is now responsible for loading exisitng mind map.



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.

Each class have appropriate constructors and destructors

```
Class Student {
public:
    ~Student () {
        delete _fullName; // release source
    }
    Student (int id, char *fullName) {
        _id = id;
        int length;
        _fullName = new char [length + 1]; // allocate memory space
       strcpy(_fullName, fullName);
   }
   …
private:
    int _id;
    char* _fullName;
}
```

Now we have Constructor and Destructor

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

```
成員變數名稱前應加底線。
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.

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

```
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.

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

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



    …
}
```

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



    …
}
```

Switch statement

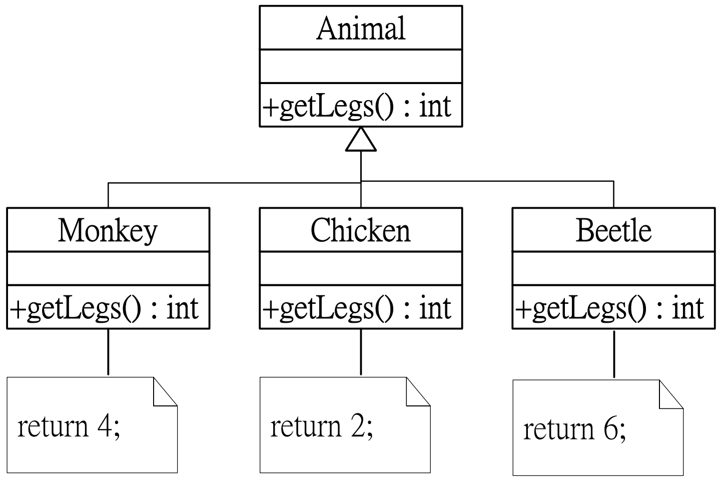
- To use polymorphism instead of switch statement。

- 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();
}
```



Large class

- As with a class with too many instance variables, a class with too much code is prime breeding ground for duplicated code, chaos, and death.

```
public class A() {
   public void method_A() {
      …
      m1();
      m2();
      m3();
   }
   public void m1() {…}
   public void m2() {…}
   public void m3() {…}
}
public class A() {
```

```
public class A() {
   public void method_A() {
      …
      b.m1();
      c.m2();
      d.m3();
   }



}
```

```
public class B () {
   public void m1() {
      …
   }
}
public class C() {
   public void m2() {
      …
   }
}
public class D() {
   public void m3() {
      …
   }
}
```

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:

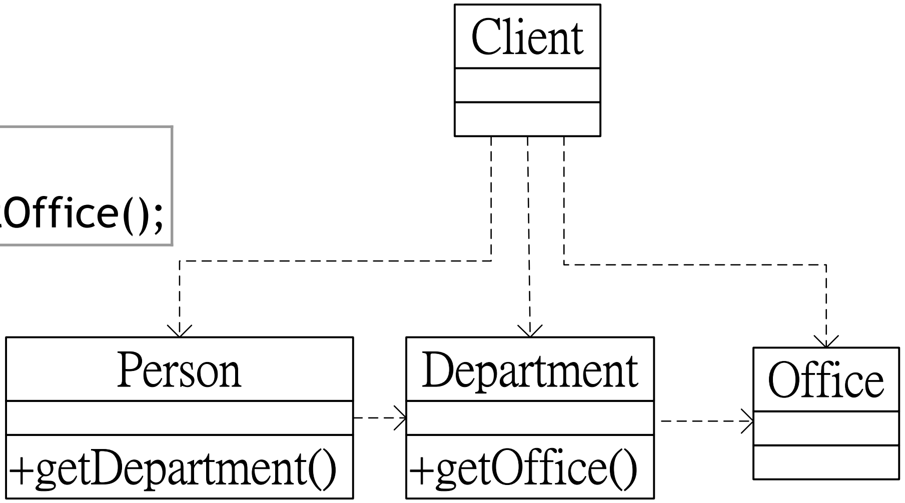
```
public class Member {
    public createMember(
        Name name,
        Address address) {
        …
    }
}
```

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.

- Not good:

```
Person jack = new Person();
Office office = jack.getDepartment().getOffice();
```



- Better solution:

```
Person jack = new Person();
Office office = jack.getOffice();
```

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

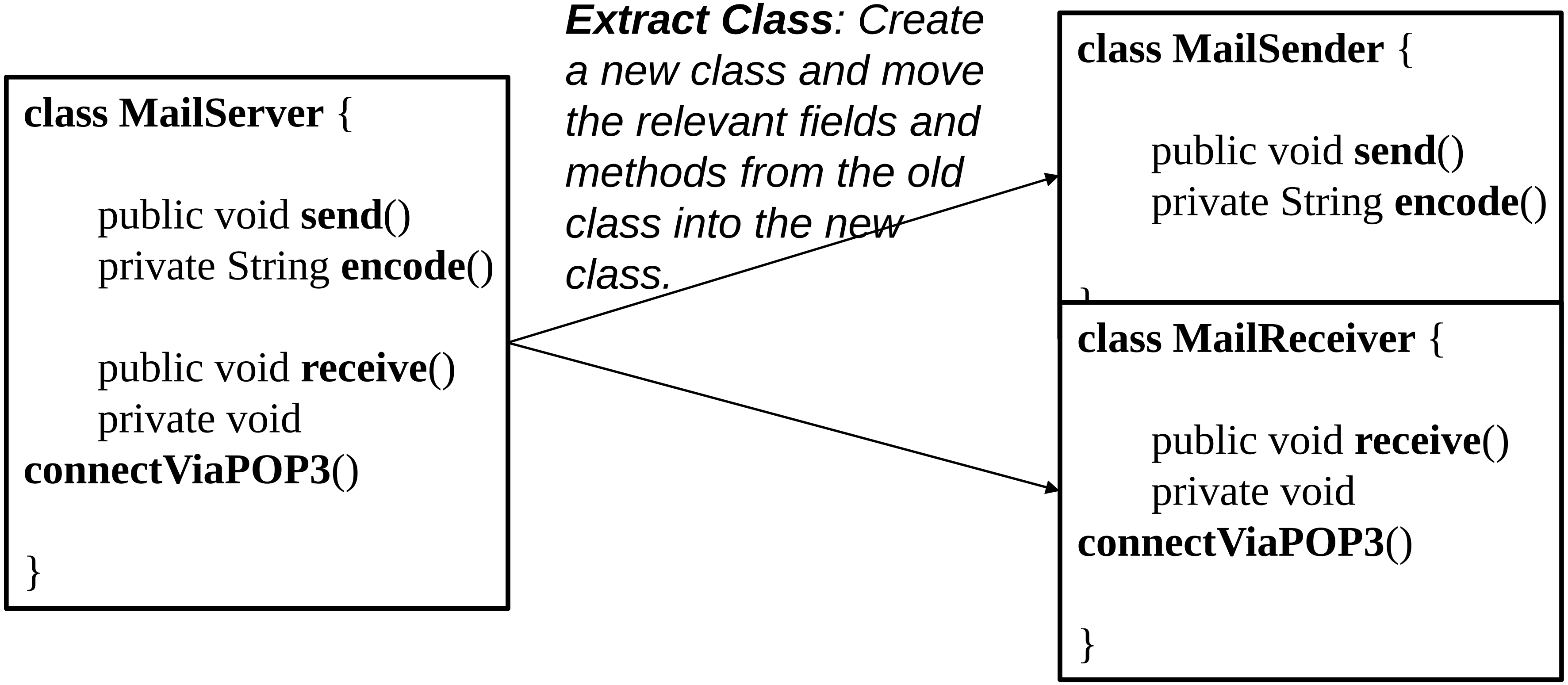
範例：此Class因二種不同行為而需變更

```
class MailServer {

            public void send(String from, String to, String
content) {
                        //…
                        String encodedContent = encode(content,
"UTF-8");
            }
            private String encode(String content, String
charset){//encode content; }


            public void receive(String account){
                        connectViaPOP3();
                        //…
            }
```

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*



***Extract Class**: Create a new class and move the relevant fields and methods from the old class into the new 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，因此建議讓檢視是否同一個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

範例

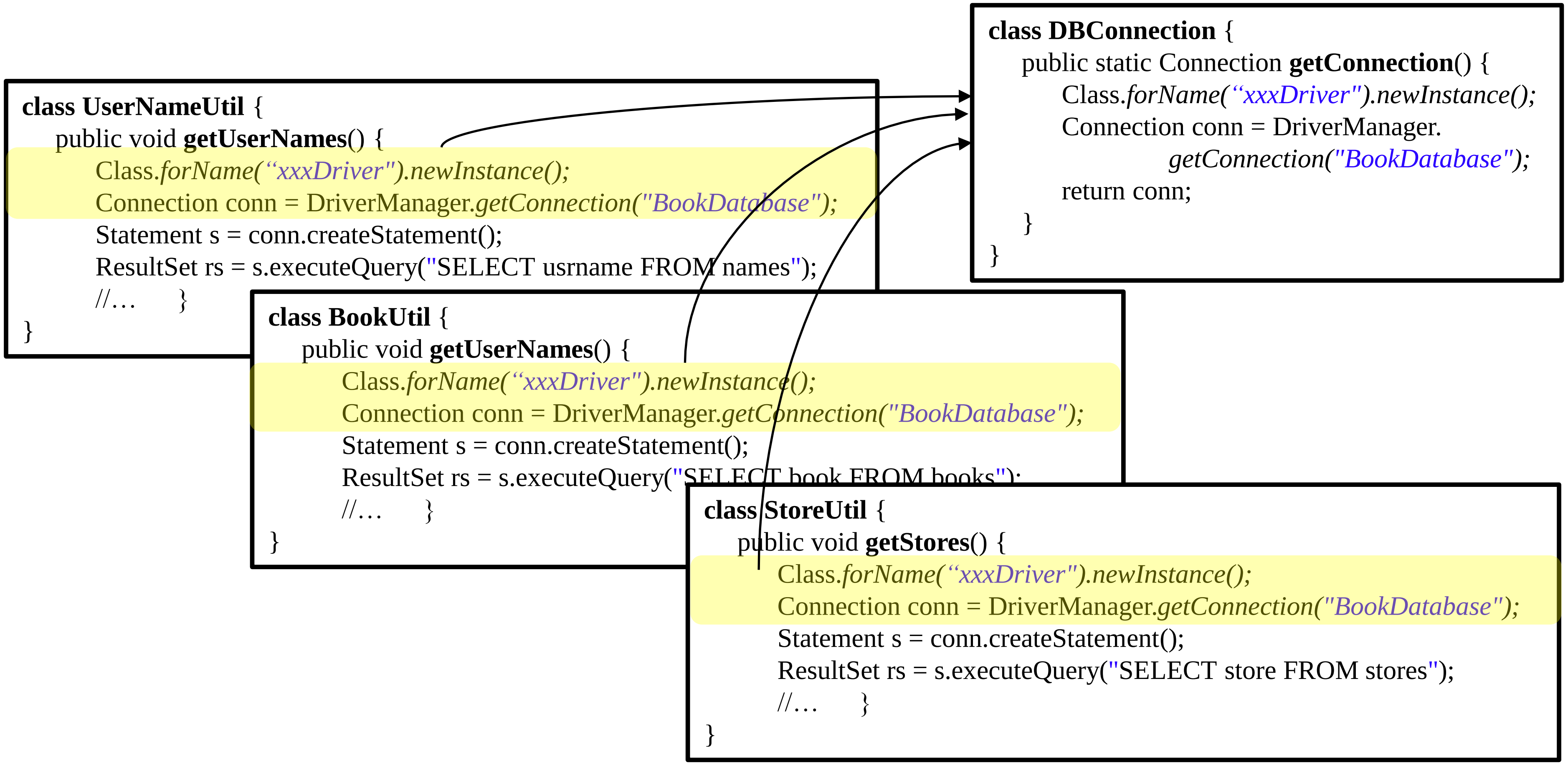
```
class UserNameUtil {
    public void getUserNames() {
        Class.forName(“xxxDriver").newInstance();
        Connection conn = DriverManager.getConnection("BookDatabase");
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT usrname FROM names");
       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

```
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");
        //…     }
}
```

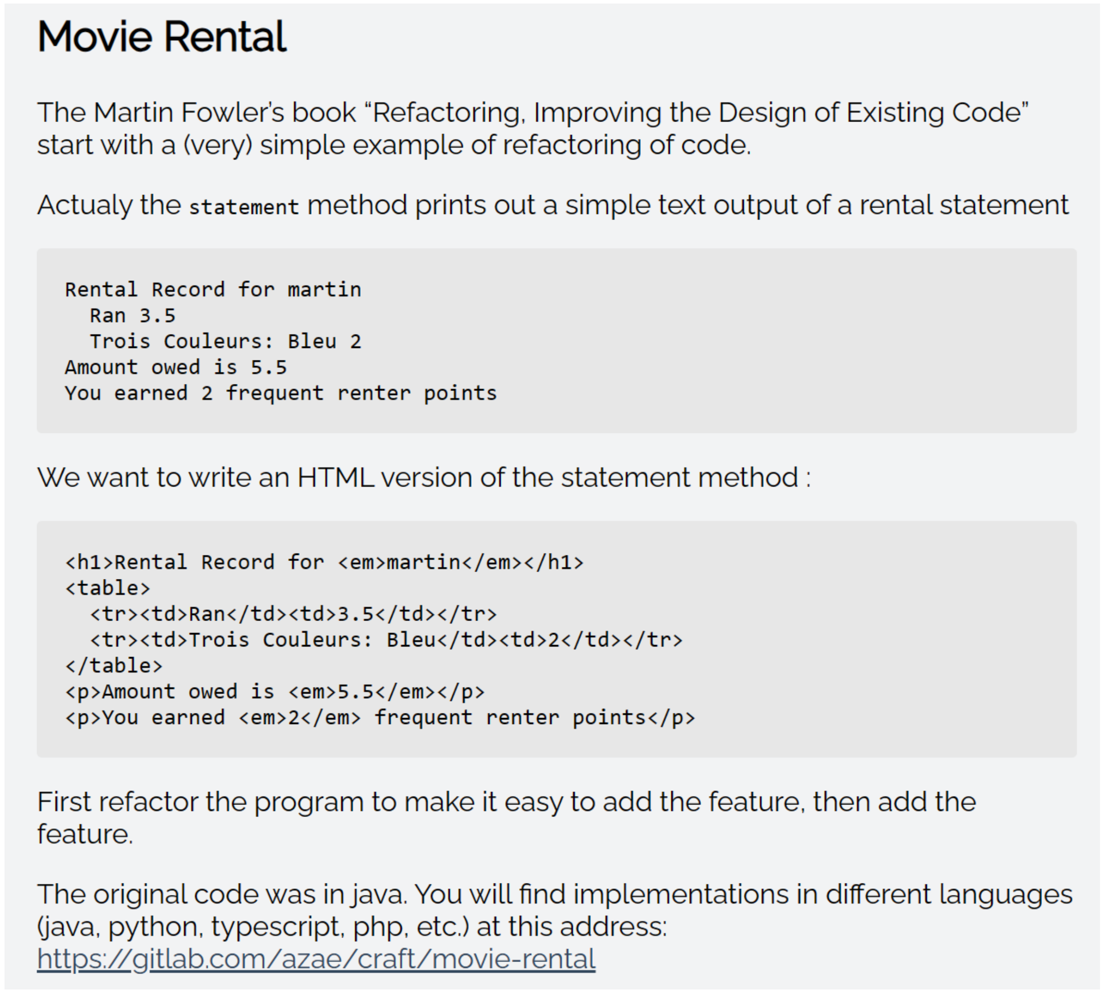
```
class BookUtil {
    public void getUserNames() {
        Connection conn = DBConnection.getConnection();
        Statement s = conn.createStatement();
        ResultSet rs = s.executeQuery("SELECT book FROM books");
        //…     }
}
```

```
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;
    }
}
```

Homework

1. 目標系統如右圖說明
2. 選擇一個程式語言實作版本 (https://gitlab.com/azae/craft/movie-rental)
3. 繪製出未重構前的Class Diagram
4. 重構此範例
5. 繪製出重構後的Class Diagram
6. 講解重構後的Class Diagram與程式碼並錄製為Video，將Video上傳至個人雲端空間或Youtube
7. 上傳以下資料至Moodle
   - 重構前Class Diagram
   - 重構後Class Diagram
   - 重構後程式碼
   - 講解Video連結 (可設為"知道連結者才能讀取")



Movie Rental

The Martin Fowler's book "Refactoring, Improving the Design of Existing Code" start with a (very) simple example of refactoring of code.

Actualy the `statement` method prints out a simple text output of a rental statement

```
Rental Record for martin
  Ran 3.5
  Trois Couleurs: Bleu 2
Amount owed is 5.5
You earned 2 frequent renter points
```

We want to write an HTML version of the statement method :

```
<h1>Rental Record for <em>martin</em></h1>
<table>
  <tr><td>Ran</td><td>3.5</td></tr>
  <tr><td>Trois Couleurs: Bleu</td><td>2</td></tr>
</table>
<p>Amount owed is <em>5.5</em></p>
<p>You earned <em>2</em> frequent renter points</p>
```

First refactor the program to make it easy to add the feature, then add the feature.

The original code was in java. You will find implementations in different languages (java, python, typescript, php, etc.) at this address: https://gitlab.com/azae/craft/movie-rental

https://codingdojo.org/kata/movie-rental/