Bad Smells in Code

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

Unresolved warnings

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

```
1  public void printSomething() {
2     int size = 3
3     String target = null;
4
5     for(int i = 0; i < size; i++) {
6        System.out.println("i = " + i);
7     }
8
9
      System.out.println(target.toString(
      ));
10 }
```

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.

```
1   int main() {
2     int size = 10;
3      int result = 0;
4      int array = new int[size];
5
6      // Assign value to the array
7      for(int i = 0; i < size; i++) {
8        array[i] = i;
9      }
10
11     for(int i = 0; i < size; i++) {
12       result += array[i];
13     }
14  }
```

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*

```
1   public void createPartControl(Composite parent) {
2       _failnodes = new HashSet<Object>();
3       _comps = new ConcurrentLinkedQueue<IComponent>();
4       _viewer = new TreeViewer(parent, SWT.MULTI |
    SWT.H_SCROLL);
5       _viewer.setInput(getViewSite());
...     ...
59      _selectionHandler = new SelectionChangHandler();
60      _selectionHandler.setViewer(_viewer);
61  }
```

Feature envy

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

```
1  public void doSomething() {
2     ClassA a = new ClassA();
3     int x = a.getX();
4     int y = a.getY();
5     int z = a.calculateSomething(x +
   y, y);
6     a.setZ(z);
7  }
```

```
1  public ClassA() {
2     public void doSomeThing() {
3        z = calculateSomething(x + y, y);
4     }
5  }
```

- 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

```
1  public class T() {
2      boolean b = false;
3
4      public int xyz(int x, int y, int z) {
5          int r = 0;
6          r = (x + y) * z / 2;
7          return r;
8      }
9  }
```

```
1  public class Trapezoid() {
2      boolean isIsosceles = false;
3
4      public int calculateArea(int top, int bottom, int
   height) {
5          int area = 0;
6          area = (top + bottom) * height / 2;
7          return area;
8      }
9  }
```

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.

```
1  void testType() {
2     unsigned short x = 65535;
3     short y = x;
4
5      for(int i = 0; i < y; i++) {
6        Do something
7     }
8  }
```

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

- Upcasting

```
1  class Animal() {}
2
3  class Mammal extends Animal()
   {}
4
5  class Cat extends Mammal() {}
6
7  class Dog extends Mammal() {}
```

```
1  Mammal m = new
   Mammal()
2  Cat c = (Cat)m;
```

Loop termination conditions are obvious and invariably achievable

```
1   for(int i = 1; (i % 2) ? ((i + 100) < 200) : ((i* 30) < 50);
    i++) {
2       Do something
3   }
4
5   for(int i = 0; i < 100; i++) {
6       Do something
7       i = i * 5;
8   }
9
10  int i = 0;
11  while(i < 10) {
12      Do something
13  }
```

```
1   for(int i = 1; i < 10; i++) {
2       Do something
3   }
4
5   for(int i = 0; i < 100; i++) {
6       Do something
7   }
8
9
10  int i = 0;
11  while(i < 10) {
12      Do something
13       i++;
14  }
```

Parentheses are used to avoid ambiguity

- Use parentheses to increase the readability and prevent logical errors

```
1  public int trapezoidArea(int top, int bottom, int height) {
2      int area = top + bottom * height / 2;
3      return area;
4  }
5
6  if (isOK && getX() * getY() == 2000 && !isFinished) {
7      Do something
8  }
```

```
1  public int trapezoidArea(int top, int bottom, int height) {
2      int area = (top + bottom) * height / 2;
3      return area;
4  }
5
6  if ((isOK) && (getX() * getY() == 2000) && (!isFinished)) {
7      Do something
8  }
```

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)

```
1  public RSSIMapCollection() {
2     _maps = new Hashtable<String, RSSIMap>();
3     _listeners = new Vector<RSSIMapCollectionEventListener>();
4     _stabilizes = new SelectionProperty(STABILIZES_LABEL);
5     _stabilizes.addElement(Stabilize.NONE);
6     _stabilizes.addElement(Stabilize.THRESHOLD);
7     _stabilizes.addElement(Stabilize.AVERAGE);
8     _stabilizes.addElement(Stabilize.WIEGHTED);
9     _stabilizes.setSelectedItem(Stabilize.THRESHOLD);
10 }
```

```
1  public RSSIMapCollection() {
2     _maps = new Hashtable<String, RSSIMap>();
3     _listeners = new Vector<RSSIMapCollectionEventListener>();
4
5     // Initialize a selection property for multiple stabilizations
6     _stabilizes = new SelectionProperty(STABILIZES_LABEL);
7     _stabilizes.addElement(Stabilize.NONE);
8     _stabilizes.addElement(Stabilize.THRESHOLD);
9     _stabilizes.addElement(Stabilize.AVERAGE);
10    _stabilizes.addElement(Stabilize.WIEGHTED);
11    _stabilizes.setSelectedItem(Stabilize.THRESHOLD);
12 }
```

Fat View (1/2)

```
1    // codes that create menus, buttons, and connects signals to slots
     (omitted)
32   MainWindown::loadMindMap() {
33      /** open dialogue box that le                    h, and read
     the text
34         * file using ifstream. Also,                   y
     the
35         * varaibles that we need t
36       */
37     …
45     while (fin.eof()) { // fin is a ifstream object.
46       fin >> line;
47       if (line == "//ROOTNODE") {
48         fin >> nodeId >> nodeDescription;
49         newRoot = new AbstractNode(nodeId, nodeDescription);
50         fin >> coordinateX >> coordinateY >> width >> height;
51         newRoot->setX(coordinateX);
52         … // more bussiness logic
100  }
```

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

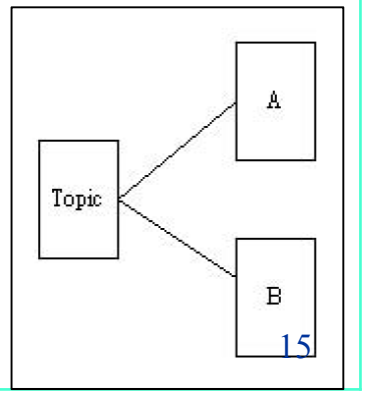
含

Fat View (2/2)

```
1   // codes that create menus, buttons, and connects signals to slots (omitted)
32  MainWindown::loadMindMap() {
33    /** open dialogue box that lets user specify a file path, and read the text
34      * file using ifstream. Also, assume we have properly declared the
35      * varaibles that we need to restore a mind map.
36      */
37    …
38    m_mindMap->loadMindMap(filePath);
39  }
40    … // more methods
41
```

```
10  void MindMap::loadMindMap(string filePath) {
11    while (fin.eof()) {
12      fin >> line;
13      if (line == "//ROOTNODE") {
14        fin >> nodeId >> nodeDescription;
15        newRoot = new AbstractNode(nodeId, nodeDescription);
16        fin >> coordinateX >> coordinateY >> width >> height;
17        newRoot->setX(coordinateX);
18        … // set coordinateY, width, and height for root node.
19      } else if (line == //NODE) {
20         fin >> parentId >> nodeId >> nodeDescription;
21         … // more bussiness logic
50  }
51    … // 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。

```
1    ... // include necessary header files.
5  using namespace std;
6  int main () {
7     ifstream inputFileStream;
8    inputFileStream.open("MyText.txt");
9    char output[100];
10    while (!inputFileStream.eof()) {
11       inputFileStream >> output;
12       ... // process read-in data
16   }
17   inputFileStream.close();
18 }
```

read in lines without checking file existence.

```
1    ... // include necessary header files.
5  using namespace std;
6  int main () {
7     ifstream inputFileStream;
8    inputFileStream.open("MyText.txt");
9    char output[100];
10    if (inputFileStream.is_open()) {
11       while (!inputFileStream.eof()) {
12          inputFileStream >> output;
13          ... // process read-in data
16       }
17   } else {
18       ... // error-handling code
20   }
22 }
```

Check if file has been opened successfully.

Each class have appropriate constructors and destructors

```
1   Class Student {
2   public:
3       ~Student () {
4           delete _fullName; // release source
5       }
6       Student (int id, char *fullName) {
7           _id = id;
8           int length;
9           _fullName = new char [length + 1]; // allocate memory space
10          strcpy(_fullName, fullName);
11      }
12      ...
20  private:
21      int _id;
22      char* _fullName;
23  }
```

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.

```
1   public class ClassAReport {
2       ...
3       public int calculateAverage(List<Integer>
    scores) {
4           int sum, average = 0;
5           for (int i = 0; i < scores.size(); i++) {
6               sum += scores.get(i);
11          }
20          average = sum / scores.size();
21          retrun average;
22      }
23      ...
    }
```

```
1   public class ClassBReport {
2       ...
3       public int calculateAverage(List<Integer>
    scores) {
4           int sum, average = 0;
5           for (int i = 0; i < scores.size(); i++) {
6               sum += scores.get(i);
11          }
20          average = sum / scores.size();
21          return average;
22      }
23      ...
    }
```

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.

```
1  public class AverageCalculator {
2      public int calculateAverage(List<Integer>
   scores) {
3          int sum, average = 0;
4          for (int i = 0; i < scores.size(); i++) {
5              sum += scores.get(i);
6          }
7          average = sum / scores.size();
8          retrun average;
9      }
10     ...
   }
```

This class is responsible for calculating average.

```
1  public class ReportCardManager {
2      public static void main (String args[]) {
3          AverageCalculator ac = new AverageCalculator();
4          ClassAReport classAReport = new ClassAReport();
5          ClassBReport classBReport = new ClassBReport();
6          int classAAverage = classAReport.calculateAverage(ac);
7          int classBAverage = classBReport.calculateAverage(ac);
8          ….
9      }
10 }
```

```
1  public class classAReportCard {
2      private List<Integer> classAScores;
3      … // initialize scores
4      public int calculateAverage (AverageCalculator ac) {
5          retrun ac.calculateAverage(classAScores);
6      }
   // Another Class
1  public class classBReportCard {
2      private List<Integer> classBScores;
3      … // initialize scores
4      public int calculateAverage (AverageCalculator ac) {
5          retrun ac.calculateAverage(classBScores);
6      }
```

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.

```
1  Class Account {
2  public:
3      string _password;
4      string getPassword();
5      ...
   };
```

```
1  Class Account {
2  public:
3     string getPassword();
4     ...
5  private:
6      string _password;
7      ...
   };
```

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

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

```
1   bool openAndProcessFile(string filePath) {
2      ifstream ifs;
3       ifs.open(filePath.c_str());
4       if (!ifs.is_open())
5           return false;
        …
10     return true;
11  }
```

Return false if file is not opened successfully.

Are there any redundant or unused variables?

- Remove unused variables from source code

```
1  public int calculateClassAverage (List<Integer> scores) {
2      int rank = 0; // never used
3      int sum, average = 0;
4      for (int i = 0; I < scores.size(); i++) {
5          sum += scores.get(i);
6      }
7      return average;
8  }
```

```
1  public int calculateClassAverage (List<Integer> scores) {
2      int sum, average = 0;
3      for (int i = 0; I < scores.size(); i++)
4          sum += scores.get(i);
5      }
6      return average;
7  }
```

Delete unused variable

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

- Variables used in the termination conditions should be initialized properly

```
1  int i;
2  while (i < 0) {
3      doSomething();
4      i++;
5  }
```

```
1  int i = -10;
2  while (i < 0) {
3      doSomething();
4      i++;
5  }
```

initialized

```
1  int i;
2  for (i ; i < someInt; i++) {
3      doSomething();
4  }
```

```
1  int i = 0;
2  for (i ; i < someInt; i++) {
3      doSomething();
4  }
```

initialized

Is overflow or underflow possible during a computation?

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

```
1  int main () {
2     short int addend = 30000;
3      short int augend = 30000;
4      short sum = addend + augend;
5      doSomething(sum);
6  };
```

```
1  int main () {
2     short int addend, augend;
3      cin >> addend;
4      cin >> augend
5
6     if (addend + augend > numeric_limits<short>::max() ||
7           (addend + augend < numeric_limits<short>::min()) {
8        throw "short integer overflow / underflow"
9     short int sum = addend + augend;
12 }
```

Are divisors tested for zero?

- Divisors should not be zero at runtime

```
1     int divisor;
2     int dividend;
3     cin >> divisor;
4     cin >> dividend;
5     int quotient = dividend /
divisor;
6     …
}
```

```
1     int divisor;
2     int dividend;
3     cin >> divisor;
4     cin >> dividend;
5
6     if (divisor == 0) {
7       throw “divisor is 0”;
8     }
9     int quotient = dividend /
divisor;
10    …
}
```

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
```

```
1  class Car {
2  public:
3      int getAbc();
4      string getXyz();
5      ...
6  private:
7     int id;
8      string manufactureDate;
9      ...
10 };
```

meaningless naming

Inconsistent coding standard

```
1  class Car {
2  public:
3      int getVehicleId ();
4      string getManufactureDate();
5      ...
6  private:
7      int _id;
8      string _manufactureDate;
9      ...
10 };
```

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.

```
1  public class Customer {
2      private String name;
3      private String title;
4      private String house;
5      private String street;
6      private String city;
7      private String postcode;
8      private String country;
9      …
10 }
```

```
1  public class Staff {
2      private String lastname;
3      private String firstname;
4      private String house;
5      private String street;
6      private String city;
7      private String postcode;
8      private String country;
9      …
10 }
```

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.

```
1  public class Address {
2      private String house;
3      private String street;
4      private String city;
5      private String country;
6      …
7  }
```

```
1  public class Customer {
2      private String name;
3      private String title;
4      private Address customerAddr;
5
6
7
8      …
9  }
```

```
1  public class Staff {
2      private String lastname;
3      private String firstname;
4      private Address staffAddr;
5
6
7
8      …
9  }
```

Switch statement

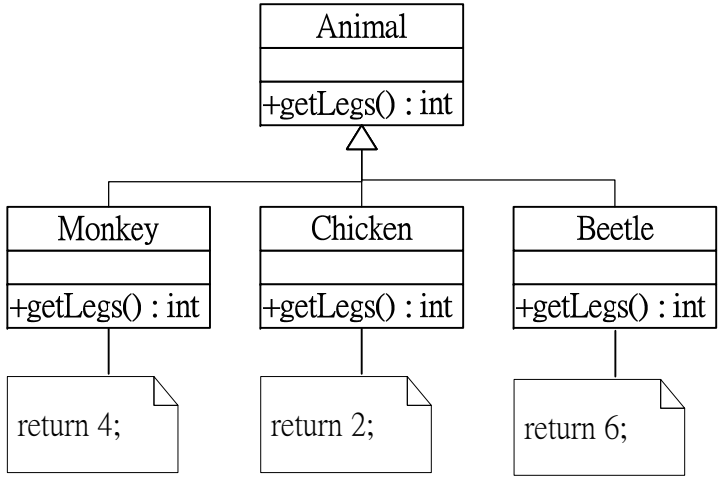
- To use polymorphism instead of switch statement。

- Not good:

```
1  public int getLegsNum() {
2     switch(animal) {
3     case ‘chicken’:
4        return 2;
5     case ‘monkey’:
6        return 4;
7     case ‘beetle’:
8        return 6;
9     default:
10       return 0;
11    }
12 }
```

- Better solution:

```
1  public int getLegsNum(Animal a) {
2     return a.getLegs();
3  }
```



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.

```
1  public class A() {
2     public void method_A() {
3        …
4        m1();
5        m2();
6        m3();
7     }
8     public void m1() {…}
9     public void m2() {…}
10    public void m3() {…}
11 }
12 public class A() {
```

```
1  public class A() {
2     public void method_A() {
3        …
4        b.m1();
5        c.m2();
6        d.m3();
7     }
8
11
12
13 }
```

```
1  public class B () {
2     public void m1() {
3        …
4     }
5  }
6  public class C() {
7     public void m2() {
8        …
9     }
10 }
11 public class D() {
12    public void m3() {
13       …
14    }
15 }
```

Long parameter list

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

- Not good:

```
1  public class Member {
2     public createMember(
3        Name name,
4        String country,
5        String postcode,
6        String city,
7        String street,
8        String house) {
9        …
10    }
11 }
```

- Better solution:

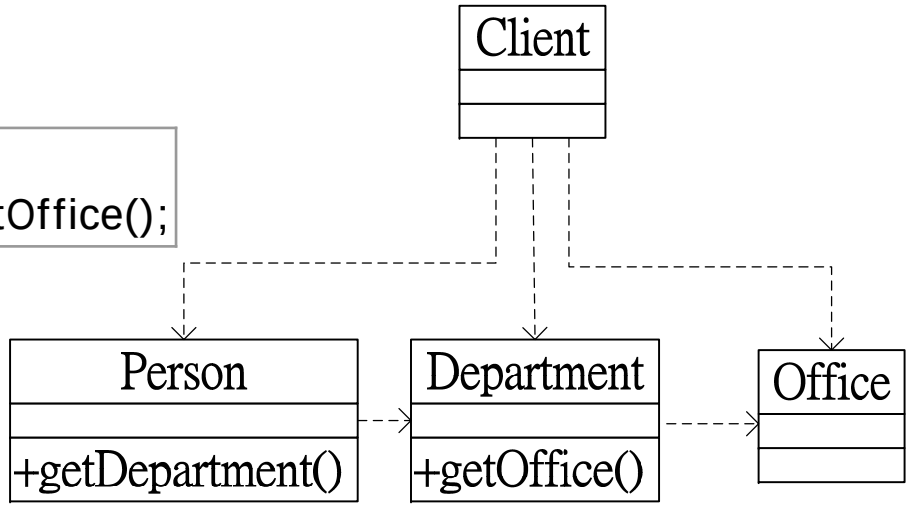
```
1  public class Member {
2     public createMember(
3        Name name,
4        Address address) {
5        …
6     }
7  }
```

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:

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



- Better solution:

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

Literal constants

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

- Not good:

```
1  public double potentialEnergy(double mass, double height) {
2     return mass * 9.81 * height;
3  }
```

- Better solution:

```
1  public double potentialEnergy(double mass, double height) {
2     final static double GRAVITATION = 9.81;
3     return mass * GRAVITATION * height;
4  }
```

Every variable is properly initialized

- Not good:

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

- Better solution:

```
1  Person person = new Person();
2  Manager = person.getManager();
3  int workHours = 40, hourlyWage = 120;
4  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

```
1  if(i < 60) {
2     //unreachable
3     if(i == 60) {
4        System.out.println("PASS");
5     }
6     else{
7        System.out.println("NOT PASS");
8     }
9  }
10 else{
11    System.out.println("PASS");
12 }
```

```
1  public class Client {
2     public createMember(Name name)
3     {
4        Name name = new Name();
5        Member.createMember(name);
6     }
7  }
```

```
1  public class Member {
2     public Member createMember(
3        Name name
4     ) {…}
5     //uncalled or unneeded procedure
6     public Member createMember(
7        String lastName,
8        String firstName,
9     ) {…}
10 }
```

Does every switch statement have a default?

- Every switch-case should define a default action

- Not good:

```
1  switch(weekday) {
2     case ‘Monday’:
3        System.out.println(“國文課”);break;
4     case ‘Tuesday’:
5        System.out.println(“英文課”);break;
6     case ‘Thursday’:
7        System.out.println(“數學課”);break;
8  }
```

- Better solution:

```
1  switch(weekday) {
2     case ‘Monday’:
3        System.out.println(“國文課”);break;
4     case ‘Tuesday’:
5        System.out.println(“英文課”);break;
6     case ‘Thursday’:
7        System.out.println(“數學課”);break;
8     default:
9        System.out.println(“休息”);break;
12 }
```

The code avoids comparing floating-point numbers for equality

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

```
1  double x = 1e-10, y1 = 20e-10, y2 = 19e-10;
2  double y = y1 – y2;
3  if(x == y) {
4     System.out.println(“X == Y”);//並不會成立
5  }
```

- Better solution:

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

All comments are consistent with the code

- Not good:

```
1  // 計算一年獲利, 傳入參數(int amount)
2  public void countProfit(int amount, double rate) {
3      _profit = amount * ( 1 + rate );
4  }
```

- Better solution:

```
1  // 計算一年獲利, 傳入參數(int amount, double rate)
2  public void countProfit(int amount, double rate) {
3      _profit = amount * ( 1 + rate );
4  }
```