



Code Smells

If it stinks, change it.

A Code We Know

```
3 public class Trip {
4     private String rideType;
5     private int dist;
6     private int minutes;
7     private int noPass;
8
9     public Trip(String rideType,
10                int dist,
11                int minutes,
12                int noPass) {
13         this.rideType = rideType;
14         this.dist = dist;
15         this.minutes = minutes;
16         this.noPass = noPass;
17     }
18
19     public void requestTrip() {
20         if (rideType == "MOTOR_BIKE") {
21             System.out.println("== Ride Swift in Bike ==");
22         } else if (rideType == "SEVEN_SEATER") {
23             System.out.println("== Ride with Friends and Family in Seven-Seater ==");
24         } else {
25             System.out.println("== Comfortable Sedan Ride ==");
26         }
27
28         if (canTakeTrip()) {
29             System.out.println(dist + " KM");
30             System.out.println(minutes + " Minutes");
31             System.out.println(perHeadFare() + " Taka Per Person");
32         } else {
33             System.out.println("Invalid Trip Request");
34         }
35     }
36 }
```

```
37     public int perHeadFare() {
38         int fare = -1;
39         switch (rideType) {
40             case "SEDAN":
41                 fare = (50 + dist * 30 + minutes * 2) / noPass;
42                 break;
43             case "MOTOR_BIKE":
44                 fare = Math.max(25, dist * 20) / noPass;
45                 break;
46             default:
47                 if (dist < 10)
48                     fare = 300 / noPass;
49                 else
50                     fare = (dist * 30) / noPass;
51
52                 break;
53         }
54
55         return fare - (fare % 5);
56     }
57
58     public boolean canTakeTrip() {
59         if (noPass < 1)
60             return false;
61
62         switch (rideType) {
63             case "SEDAN":
64                 return noPass <= 4 && dist <= 25;
65             case "SEVEN_SEATER":
66                 return noPass <= 7 && dist >= 10;
67             default:
68                 return noPass <= 1 && dist <= 10;
69         }
70     }
71 }
```

Inappropriate name

- Use full form of phrases, Examples
 - Use *BookCover* instead of *BC*
 - Use *handler*, not *hdlr*
- Some short forms are *well known*^{*}. It is OK to use them
 - GPA and *GradePointAverage* are both OK
- Some short forms are *better known*^{*} than full forms
 - Better use *USB* instead of *UniversalSerialBus*

- A variable name should say exactly what it is
- A method should say exactly what it does

double calc(**double** slr)



double calculateTax(**double** salary)



Duplicated code

- Literal Duplication
 - Exact copy of a code
- Semantic Duplication
 - Example, loop vs repeated lines.
- Data Duplication
 - Example same constant declared in two classes.
- Conceptual Duplication
 - Example interchangeably used 2 Algorithm to sort elements without good reason
- Structural Duplication
 - Similar structure of code with in several places.
 - Condition on same value again and again.

Find Duplicates

```
public boolean isRideValid() {  
    if (vehicleType == "sedun") {  
        return numberOfPassengers <= 4 && distance <= 25;  
    } else if (vehicleType == "motor-bike") {  
        return numberOfPassengers <= 1 && distance <= 10;  
    } else if (vehicleType == "seven-seater") {  
        return numberOfPassengers <= 7 && distance >= 10;  
    } else {  
        throw new RuntimeException("invalid vehicle type");  
    }  
}
```

```
public int getFare() {  
    if (vehicleType == "sedun") {  
        return (distance * 30) / numberOfPassengers;  
    } else if (vehicleType == "motor-bike") {  
        return (distance * 15) / numberOfPassengers;  
    } else if (vehicleType == "seven-seater") {  
        return (distance * 40) / numberOfPassengers;  
    } else {  
        throw new RuntimeException("invalid vehicle type");  
    }  
}
```

Dead Code

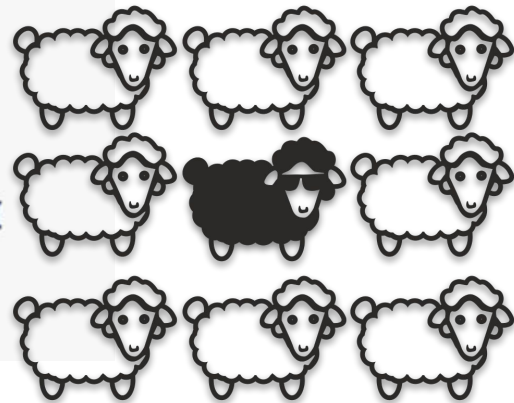
- Code that is no longer required are dead
 - Commented code
- Bury (remove) it, with courage
- What if we need it back?
 - Find it in the source control



Black Sheep

- A method does not fit in its class
- A subclass does not fit in the family

```
public class StringUtil {  
    public static String pascalCase(String string) {  
        return string.substring(0,1).toUpperCase() + string.substring(1);  
    }  
  
    public static String camelCase(String string) {  
        return string.substring(0,1).toLowerCase() + string.substring(1);  
    }  
  
    public static String numberAndNoun(int number, String noun) {  
        return number + " " + noun + (number != 1 ? "s" : "");  
    }  
  
    public static String extractCommandNameFrom(Map parameterMap) {  
        return ((String[]) parameterMap.get("command"))[0];  
    }  
}
```



Long method

- Makes it difficult to understand the method
- Hides behavior that should have been shared, resulting duplicated code

```
<div id="container" class="red">
  Hello world
  <div name="banner">
    Again
  </div>
</div>
```

```
private String toHtmlString(StringBuffer result) {
    result.append("<");
    result.append(name);

    for (int i=0; i<attributeNames.size(); i++){
        result.append(" ");
        result.append(attributeNames.get(i));
        result.append("=");
        result.append("\"");
        result.append(attributeNames.get(i));
        result.append("\");
    }

    result.append(">");
    if (value != null)
        result.append(value);
    for (HtmlNode child: children){
        child.toHtmlString(result);
    }
    result.append("</");
    result.append(name);
    result.append(">");
    return result.toString();
}
```


Long method

- Makes it difficult to understand the method
- Hides behavior that should have been shared, resulting in duplicated code

```
<div id="container" class="red">  
    Hello world  
</div>
```

```
if (value != null)  
    result.append(value);
```

```
result.append("</");  
result.append(name);  
result.append(">");
```

```
private String toHtmlString(StringBuffer result) {  
    startTag(result);  
  
    addValue(result);  
    for (HtmlNode child: children){  
        child.toHtmlString(result);  
    }  
    endTag(result);  
    return result.toString();  
}
```

```
result.append("<");  
result.append(name);
```

```
for (int i=0; i<attributeNames.size(); i++){  
    result.append(" ");  
    result.append(attributeNames.get(i));  
    result.append("=");  
    result.append("\"");  
    result.append(attributeNames.get(i));  
    result.append("\"");  
}  
result.append(">");
```

Large class

- A class that's too large and doing so many things
- Often called a god class
- Possibility of SRP violation
- Name of large class often contains words like Controller, Manager, Processor.



Conditional Complexity

```
if ((month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10) && day == 31) {  
    day = 1;    month++;  
} else if ((month == 4 || month == 6 || month == 9 || month == 11) && day == 30) {  
    day = 1;    month++;  
} else if ((month == 12) && day == 31) {  
    day = 1;    month++;    year++;  
} else if ((month == 2)) {  
    if (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0)) {  
        if (day == 29) {  
            day = 1;    month++;  
        } else {  
            day = day + 1;  
        }  
    } else {  
        if (day == 28) {  
            day = 1;    month++;  
        }  
    }  
} else {  
    day = day + 1;  
}
```



What is the method doing?

Can you rewrite it?

Feature Envy

A method that seems more interested in some other class than the one it is in

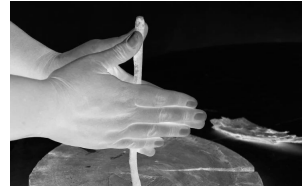


```
double capital(Investment inv) {  
    if (inv.getExpiry() == null && inv.getMaturity() != null) {  
        return inv.commitmentAmount() * inv.duration() * inv.riskFactor();  
    }  
  
    if (inv.getExpiry() != null && inv.getMaturity() == null) {  
        if (inv.unusedPercentage() != 1.0) {  
            return inv.commitmentAmount() * inv.unusedPercentage() * inv.duration() *  
                inv.riskFactor();  
        } else {  
            return (inv.outstandingRiskAmount() * inv.duration() * inv.riskFactor())  
                + (inv.unusedRiskAmount() * inv.duration() * inv.unusedRiskFactor());  
        }  
    }  
  
    return 0.0;  
}
```

Primitive Obsession

Examples

- Using array instead of list
- Using array instead of object
- Using primitive values instead of objects
- Not using library methods



Primitive Obsession

Replace array with object

```
Object[] result = new Object[3];  
result[0] = "Error calculating statistics";  
result[1] = succeededCount;  
result[2] = failedCount;
```

```
public class Result {  
    private String errorMessage;  
    private int succeededCount;  
    private int failedCount;  
}
```



Primitive Obsession Example

```
if (someString.indexOf("substring") != -1)
```

Instead of

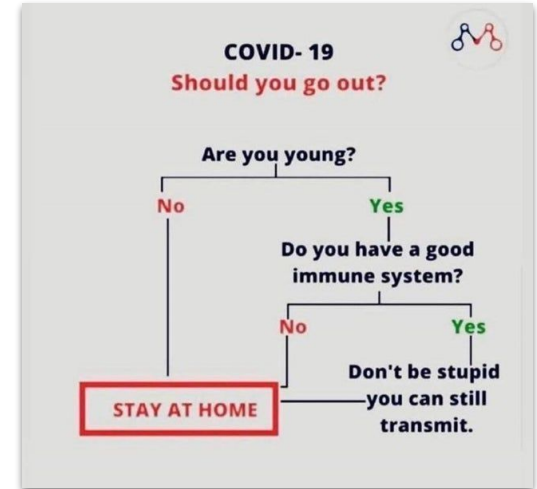
```
if(someString.contains("substring"))
```

```
private void Grow() {  
    Object[] newElements = new Object[elements.length + 10];  
    for (int i = 0; i < size; i++)  
        newElements[i] = elements[i];  
  
    elements = newElements;  
}
```

```
private void Grow() {  
    Object[] newElements = new Object[elements.length + INITIAL_CAPACITY];  
    System.arraycopy(elements, 0, newElements, 0, size);  
    elements = newElements;  
}
```

Conditional Statement

```
private static String getSoundIfElseWay(String animal) {  
    if (animal == null)  
        return "not a animal"  
    if (animal.equalsIgnoreCase("Dog"))  
        return "Bark";  
    else if (animal.equalsIgnoreCase("Cat"))  
        return "Mew";  
    else if (animal.equalsIgnoreCase("Lion"))  
        return "Roar";  
    return "Unknown Animal";  
}
```



Lazy Class

A class that does not do much. Examples -

- An intermediate base class that isn't really required
- A container of another object that does not add any additional functionality



Lazy Class Example 1

```
interface SomeInterface {  
    void methodOne();  
    void defaultMethod();  
}  
  
abstract class IntermediateBaseClass implements SomeInterface {  
    public abstract void methodOne();  
    public void defaultMethod() {  
        //do nothing  
    }  
}  
  
class WorkerClazz extends IntermediateBaseClass {  
    public void methodOne() {  
        // some actual code here  
    }  
    public void defaultMethod() {  
        //some more actual code  
    }  
}
```

Unnecessary
intermediate
base class

Lazy Class Example 2

```
class Letter {  
    private final String content;  
  
    public Letter(String content) {  
        this.content = content;  
    }  
  
    public String getContent() {  
        return content;  
    }  
}
```

Unnecessary
container class

Speculative Generality

- Existence of code that "might be" required later
- Example - A super class with just one subclass
- Often done considering "Good"

**Premature
optimisation is the
root of all evil.**

Refused Bequest

- Subclasses inherit code that they don't want.
- Possible violation of LSP

```
public class Bird {  
  
    void fly() {  
        System.out.println("Flying!!");  
    }  
}  
  
public class Ostrich extends Bird {  
  
    void fly() {  
        throw new IllegalStateException("An ostrich can't fly");  
    }  
}
```

Refused Bequest

- Subclasses inherit code that they don't want.
- Possible violation of LSP

```
abstract class Shape {  
    List<Shape> components;  
  
    void draw() {  
        for (Shape c: components){  
            c.draw();  
        }  
    }  
  
    void add(Shape shape){  
        components.add(shape);  
    }  
  
    void remove(Shape shape){  
        components.remove(shape);  
    }  
}
```

```
class Line extends Shape {  
  
    @Override  
    public void draw() {  
        // draw a line  
    }  
  
    @Override  
    public void add(Shape shape) {  
        // do nothing  
    }  
  
    @Override  
    public void remove(Shape shape) {  
        // do nothing  
    }  
}
```

(Bad) Comments

- Comments represent a failure to express an idea in the code.
- Try to make your code self-documenting or intention-revealing.
- Thumb rule
 - ✘ Do not comment what is done in a code
 - ✔ Comment why it is done (in needed)



Data Clumps

```
boolean isInRange(int start, int end, int value) {  
    return value >= start && value <= end;  
}
```

```
boolean isInRange(int start1, int end1, int start2, int end2){  
    return start1 <= start2 && end1 >= end2;  
}
```

Some data are frequently passed together

Data Clumps

```
public class Order {  
    private String customerName;  
    private String customerEmail;  
    private String shippingAddress;  
}
```

```
public class ShoppingCart {  
    private List<CartItem> items;  
    private String customerName;  
    private String customerEmail;  
}
```

More Code Smells

- Shotgun Surgery
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class (correct: true sense of Object but actually deviate in practice)

[<https://softwareengineering.stackexchange.com/questions/338195/why-are-data-classes-considered-a-code-smell>]

Shotgun Surgery

- To introduce a small new change, a developer has to change many classes and methods, and most of the time has to write duplicated code, which violates the “Don’t Repeat Yourself” principle.
- Causes of Shotgun Surgery
 - Poor separation of concerns.
 - Failure to understand responsibilities, often due to misunderstanding (single responsibility principle).
 - Not identifying the common behavior or behaviors with a slight change.
 - Failure to introduce proper design patterns.
- Consequences of Shotgun Surgery:
 - Lots of duplicate code
 - Taking more time to develop small features
 - Unmaintainable code base

Shotgun Surgery

```
public void debit(int debit) throws Exception
{
    if(amount <= 500)
    {
        throw new Exception("Mininum balance
shuold be over 500");
    }
    amount = amount-debit;
    System.out.println("Now amount is" + amount);
}
```

```
public void transfer(Account from,Account to,int
cerditAmount) throws Exception
{
    if(from.amount <= 500)
    {
        throw new Exception("Mininum balance
shuold be over 500");
    }
    to.amount = amount+cerditAmount;
}
```

```
public void sendWarningMessage()
{
    if(amount <= 500)
    {
        System.out.println("amount should be
over 500");
    }
}
```

Solution: Create a common method and invoke that method

Temporary Field

- A Temporary Field smell occurs when a variable that should be defined within a method's scope is instead defined in the class' scope.
- It refers to variables only used in some situations or specific areas of a program.
- This violates the information hiding principle, since all the methods in the class will have access to this variable, when only one method needs it.
- Solution: Extract Class, Move Method

Temporary Field

```
class Person {
    private String name;
    private Address address;
    private boolean isValid;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
        this.isValid = false;
    }

    public void validateAddress() {
        if (address != null && !address.getStreet().isEmpty() &&
!address.getCity().isEmpty() && !address.getZipCode().isEmpty()) {
            isValid = true;
        } else {
            isValid = false;
        }
    }

    public boolean isValid() {
        return isValid;
    }
}
```

Incomplete Library Class

```
public class StringUtils {  
    public static String toUpperCase(String str)  
{  
        return str.toUpperCase();  
    }  
  
    public static String toLowerCase(String str)  
{  
        return str.toLowerCase();  
    }  
}
```

Client may need additional functionalities. For example:

```
    public static String toTitleCase(String str)  
    public static String toCamelCase(String str)  
    public static String toPascalCase(String str)
```

- UPPERCASE: "CODE SMELL AND REFACTORING"
- lowercase: "code smell and refactoring"
- Title Case: "Code Smell And Refactoring"
- camelCase: "codeSmellAndRefactoring"
- Pascal Case: "CodeSmellAndRefactoring"

Farther Reading

The items are more important as you move down the list

- *Clean Code* by Robert Martin
- *Refactoring* by Martin Fowler
- *Code older than a month* by You!