BAD CODE SMELLS

LINGI2252 – PROF. KIM MENS
A. INTRODUCTION
Bad Smells in Code

Reference


Chapter 3: *Bad Smells in Code*, by Kent Beck and Martin Fowler

Overview of this presentation

Introduction

A classification of bad smells, including a detailed illustration of some of them

Conclusion
Introduction

Bad Smells = “bad smelling code”

indicate that your code is ripe for refactoring

**Bad smells** are about

*when* to modify your code

**Refactoring** is about

*how* to change code by applying refactorings
Bad Smells

Allow us to identify

what needs to be changed in order to improve the code

A recipe book to help us choose the right refactoring pattern

No precise criteria

More to give an intuition and indications

Goal: a more “habitable” code.
Side note: *Habitable* code

*Habitable* code is code in which developers feel at home (even when the code was not written by them)

Symptoms of *inhabitable* code include overuse of abstraction or inappropriate compression

Habitable code should be easy to read, easy to change

Software needs to be habitable because it always has to change

B. CLASSIFICATION OF BAD SMELLS
INCLUDING A DETAILED DISCUSSION OF 5 OF THEM
An Online Classification

https://sourcemaking.com/refactoring

Refactoring

Bad code smells

**Bloaters**

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps
Bad Smells : Classification

- **The top crime**

- Class / method organisation
  
  Large class, **Long Method**, Long Parameter List, Lazy Class, Data Class, …

- Lack of loose coupling or cohesion
  
  Inappropriate Intimacy, **Feature Envy**, Data Clumps, …

- Too much or too little delegation
  
  Message Chains, **Middle Man**, …

- Non Object-Oriented control or data structures
  
  Switch Statements, Primitive Obsession, …

- Other : **Comments**
Bad Smells: Alternative Classification

- **Bloaters** are too large to handle
- **Object-orientation abusers** do not respect OO principles
- **Change preventers** stand in the way of change
- **Dispensables** are things you could do without
- **Couplers** contribute to excessive coupling between classes

Other smells
Bad Smells: Classification

- The top crime
- Class / method organisation
  - Large class, **Long Method**, Long Parameter List, Lazy Class, Data Class, …
- Lack of loose coupling or cohesion
  - Inappropriate Intimacy, **Feature Envy**, Data Clumps, …
- Too much or too little delegation
  - Message Chains, **Middle Man**, …
- Non Object-Oriented control or data structures
  - Switch Statements, Primitive Obsession, …
- Other: **Comments**
The top crime

Code duplication
Code duplication

_Duplicated code_ is the number 1 in the stink parade!

We have duplicated code when we have the same code structure in more than one place.

Why is duplicated code bad?

A fundamental _rule of thumb_:

it’s always better to have a unified code.
public double ringSurface(r1, r2) {
    // calculate the surface of the first circle
    double surf1 = bigCircleSurface(r1);
    // calculate the surface of the second circle
    double surf2 = smallCircleSurface(r2);
    return surf1 - surf2;
}

private double bigCircleSurface(r1) {
    pi = 4* ( arctan 1/2 + arctan 1/3 );
    return pi*sqr(r1);
}

private double smallCircleSurface(r2) {
    pi = 4* ( arctan 1/2 + arctan 1/3 );
    return pi*sqr(r2);
}
Code duplication example 2

Class

- method1
  - code
  - code

- method2
  - code

- method3
  - code

Same expression in two or more methods of the same class
Code duplication example 3

Same expression in two sibling classes

SubClassA

methodA

code

Class

inherits
from

SubClassB

methodB

code
Code duplication example 4

Same expression in two unrelated classes
public double ringSurface(r1, r2) {
    // calculate the surface of the first circle
    double surf1 = surface(r1);
    // calculate the surface of the second circle
    double surf2 = surface(r2);
    return surf1 - surf2;
}

private double surface(r) {
    pi = 4 * (arctan 1/2 + arctan 1/3);
    return pi * sqr(r);
}
Code duplication: Refactoring Patterns (2)

Same expression in two or more methods of the same class

Class

method1

code

code

method2

code

method3

code

Class

method1

Call methX()

Call methX()

method2

Call methX()

method3

Call methX()

methX()

Return xcode

Extract method
Code duplication: Refactoring Patterns (3)

- **Same expression in two sibling classes**
  - SubClass A
  - SubClass B

- **Same code**
  - Extract method
    - + Pull up field
  - Extract method
    - + Form Template Method

- **Different algorithm**
  - Substitute algorithm
Code duplication: Refactoring Patterns (4)

Same expression in two unrelated classes

ClassA

methodA

code

Extract class

methX()

return

code

ClassB

methodB

code

Extract class
Same expression in two unrelated classes

If the method really belongs in one of the two classes, keep it there and invoke it from the other class.
Bad Smells: Classification

- The top crime

- Class / method organisation
  - Large class, Long Method, Long Parameter List, Lazy Class, Data Class, …

- Lack of loose coupling or cohesion
  - Inappropriate Intimacy, Feature Envy, Data Clumps, …

- Too much or too little delegation
  - Message Chains, Middle Man, …

- Non Object-Oriented control or data structures
  - Switch Statements, Primitive Obsession, …

- Other: Comments
A *large class* is a class that is trying to do too much

Often shows up as too many instance variables

Use **Extract Class** or **Extract Subclass** to bundle variables

choose variables that belong together in the extracted class

common prefixes and suffixes may suggest which ones may go together, e.g. *depositAmount* and *depositCurrency*

* Cartoon borrowed from [https://sourcemaking.com/refactoring](https://sourcemaking.com/refactoring) for didactic purposes only.
A class may also be too large in the sense that it has too much code likely some code inside the class is duplicated

solve it by extracting the duplicated code in separate methods using **Extract Method**

or move part of the code to a new class, using **Extract Class** or **Extract Subclass**

if need be, move existing or extracted methods to another class using **Move Method**
In procedural programming languages, we pass as parameters everything needed by a subroutine because the only alternative is global variables.

With objects you don’t pass everything the method needs.

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
Long Parameter List

Long parameter lists are hard to understand

Pass only the needed number of variables

Use **Replace Parameter with Method** when you can get the data in one parameter by making a request of an object you already know about
Lazy Class

Each class cost money (and brain cells) to maintain and understand.

A class that isn't doing enough to pay for itself should be eliminated.

It might be a class that was added because of changes that were planned but not made.

Use **Collapse Hierarchy** or **Inline Class** to eliminate the class.

* Cartoon borrowed from [https://sourcemaking.com/refactoring](https://sourcemaking.com/refactoring) for didactic purposes only.
Data Class

Classes with just fields, getters, setters and nothing else

If there are public fields, use **Encapsulate Field**

For fields that should not be changed use **Remove Setting Method**

* Cartoon borrowed from [https://sourcemaking.com/refactoring](https://sourcemaking.com/refactoring) for didactic purposes only.
Object programs live best and longest with short methods.

New OO programmers feel that OO programs are endless sequences of delegation.

Older languages carried an overhead in subroutine calls which deterred people from small methods.

There is still an overhead to the reader of the code because you have to switch context to see what the sub-procedure does (but the development environment helps us).

Important to have a good name for small methods.

Rename Method
Long Method

The longer a procedure is, the more difficult it is to understand what the code does

More difficult to read

Bad for maintainability

More difficult to make modifications

To summarise… less *habitable* !

* Cartoon borrowed from [https://sourcemaking.com/refactoring](https://sourcemaking.com/refactoring) for didactic purposes only.
Long Method

Too avoid too long methods, decompose methods in many small ones

Heuristic: whenever you feel the need to comment something, write a method instead

containing the code that was commented

named it after the intention of the code rather than how it does it

Even a single line is worth extracting if it needs explanation
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    // Print banner
    System.out.println(" ****************");
    System.out.println(" ***** Customer *****");
    System.out.println(" ****************");
    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    // Print details
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}
Long Method: Refactoring patterns

99% of the time, all we have to do to shorten a method is **Extract Method**

Find parts of the method that seem to go together nicely and extract them into a new method.

It can lead to problems…

Many temps: use **Replace Temp with Query**

Long lists of parameters can be slimmed down with **Introduce Parameter Object**
But how to identify the clumps of code to extract?

Look for comments…

A block of code with a comment that tells you what it is doing can be replaced by a method whose name is based on the comments.

Loops also give signs for extractions…

Extract the loop and the code within the loop into its own method.
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // Print banner
    System.out.println("*****************");
    System.out.println("***** Customer *****");
    System.out.println("**************");

    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    // Print details
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}
Long Method Example revisited

void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // Print banner
    System.out.println("******************");
    System.out.println("***** Customer *****");
    System.out.println("******************");

    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    // Print details
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    // Print details
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}

void printBanner() {
    System.out.println("******************");
    System.out.println("***** Customer ****");
    System.out.println("******************");
}
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    printBanner();
    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    // Print details
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}
void printBanner() {
    System.out.println("********************");
    System.out.println("***** Customer ****");
    System.out.println("********************");
}
Long Method Example revisited

```java
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    printBanner();
    // Calculate outstanding
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    printDetails(outstanding);
}

void printDetails(double outstanding) {
    System.out.println("name: "+_name);
    System.out.println("amount" + outstanding);
}

void printBanner() { … }

2. Extract Method
Using Local Variables
```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}

void printDetails(double outstanding) {
    System.out.println("name: " + _name);
    System.out.println("amount" + outstanding);
}

void printBanner() { … }
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    Enumeration e = _orders.elements();
    double result = 0.0;
    While (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}

void printDetails(double outstanding) {...}
void printBanner() { … }

3. **Extract Method**
Reassigning a Local Variable
Bad Smells: Classification

- The top crime
- Class / method organisation
  - Large class, **Long Method**, Long Parameter List, Lazy Class, Data Class, …
- Lack of loose coupling or cohesion
  - Inappropriate Intimacy, **Feature Envy**, Data Clumps, …
- Too much or too little delegation
  - Message Chains, **Middle Man**, …
- Non Object-Oriented control or data structures
  - Switch Statements, Primitive Obsession, …
- Other: **Comments**
Coupling and cohesion

Coupling is the degree to which different software components depend on each other

Cohesion is the degree to which the elements within a software module belong together

Low cohesion and tight coupling are bad smells (why?)
Inappropriate Intimacy

Pairs of classes that know too much about each other's private details

Use **Move Method** and **Move Field** to separate the pieces to reduce the intimacy

If subclasses know more about their parents than their parents would like them to know

Apply **Replace Inheritance with Delegation**

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
Data Clumps

A certain number of data items in lots of places

Examples: fields in a couple of classes, parameters in many method signatures

Ought to be made into their own object

When the clumps are fields, use Extract Class to turn them into an object

When the clumps are parameters, use Introduce Parameter Object to slim them down
Feature Envy

When a method seems more interested in a class other than the one it actually is in

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
Feature Envy

In other words, when a method invokes too many times methods on another object to calculate some value

Why is it bad to invoke a zillion times methods from another class?

Because, in general, it is not logical from an OO point of view.

Put things together that change together!
Feature Envy: Example (1)

```java
public void mainFeatureEnvy () {
    OtherClass.getMethod1();
    OtherClass.getMethod2();
    OtherClass.getMethod3();
    OtherClass.getMethod4();
}
```

```java
OtherClass

public void getMethod1 () { ... }
public void getMethod2 () { ... }
public void getMethod3 () { ... }
public void getMethod4 () { ... }
```
First solution: **Move Method**

```java
public class OtherClass {
    public void getMethod1 () { … }
    public void getMethod2 () { … }
    public void getMethod3 () { … }
    public void getMethod4 () { … }
    public void mainFeatureEnvy () {
        getMethod1();
        getMethod2();
        getMethod3();
        getMethod4();
    }
}
```

Could we use **Extract method**?

Yes! If only a part of the method suffers from envy
Feature Envy: Example (2)

```java
public void mainFeatureEnvy()
    { public void getMethod1()
    Class1.getMethod1();
    Class1.getMethod2();
    Class2.getMethod3();
    Class2.getMethod4();
    }
```

```
public void getMethod1()
    { … }
public void getMethod2()
    { … }
```
Feature Envy: Refactoring Patterns (2)

Use the same method as the first example: **Extract Method** or **Move Method**

To choose the good class we use the following heuristic:

*determine which class has most of the data and put the method with that data*
When making one kind of change requires many small changes to a lot of different classes

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
Shotgun Surgery

Hard to find all changes needed; easy to miss an important change

Use **Move Method** and **Move Field** to put all change sites into one class

*Put things together that change together!*

If a good place to put them does not exist, create one.
Parallel Inheritance Hierarchies

*Special case of Shotgun Surgery*

Each time I add a subclass to one hierarchy, I need to do it for all related hierarchies

Use **Move Method** and **Move Field**

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.*
Bad Smells : Classification

imeline

The top crime

Class / method organisation

Large class, **Long Method**, Long Parameter List, Lazy Class, Data Class, …

Lack of loose coupling or cohesion

Inappropriate Intimacy, **Feature Envy**, Data Clumps, …

Too much or too little delegation

Message Chains, **Middle Man**, …

Non Object-Oriented control or data structures

Switch Statements, Primitive Obsession, …

Other : **Comments**
Message Chains

A client asks an object for another object who then asks that object for another object, etc.

Bad because client depends on the structure of the navigation

Use **Extract Method** and **Move Method** to break up or shorten such chains

* Cartoon borrowed from [https://sourcemaking.com/refactoring](https://sourcemaking.com/refactoring) for didactic purposes only.
Middle Man

Objects hide internal details (encapsulation)

Encapsulation leads to delegation

It is a good concept but...

Sometimes it goes to far…
Middle Man

Real-life example:

You ask a director whether she is free for a meeting.

She delegates the message to her secretary that delegates it to the diary.

Everything is good… but, if the secretary has nothing else to do, it is better to remove her!
Middle Man

If a class performs only one action, delegating work to other classes, why does it exist at all?

Sometimes most methods of class just delegate to another class

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
The class `Person` is hiding the `Department` class.

To find a person’s manager, clients ask:

```
Manager = john.getManager();
```

and the person then needs to ask:

```
_department.getManager();
```
Middle Man: Refactoring

client class

Person
getManager()

Department
getManager()

client class

Person
getDepartment()

Department
getManager()
Remove Middle Man…

First step : Create an accessor for the delegate.

```java
class Person {
    Department _department;
    public Person getManager() {
        return _department.getManager();
    }
    public Department getDepartment() {
        return _department;
    }
}
```
**Middle Man: Refactoring**

**Second step** : For each client use of a delegated method, remove the method from the middle man and replace the call in the client to call a method directly on the delegate.

\[
\text{Manager} = \text{john}.\text{getDepartment().getManager();}
\]

**Last step** : Compile and test.
Bad Smells : Classification

- **The top crime**

- Class / method organisation
  - Large class, **Long Method**, Long Parameter List, Lazy Class, Data Class, …

- Lack of loose coupling or cohesion
  - Inappropriate Intimacy, **Feature Envy**, Data Clumps, …

- Too much or too little delegation
  - Message Chains, **Middle Man**, …

- Non Object-Oriented control or data structures
  - Switch Statements, Primitive Obsession, …

- Other : **Comments**
Switch Statements

Switch statements ("cases")

often cause duplication

adding a new clause to the switch requires finding
all such switch statements throughout your code

OO has a better ways to deal with actions depending on types:

polymorphism!

Use Extract Method to extract the switch statement and then Move Method to get it into the class where polymorphism is needed.

Then use Replace Conditional with Polymorphism after you setup the inheritance structure.

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
Primitive Obsession

Characterised by a reluctance to use classes instead of primitive data types

The difference between classes and primitive types is hard to define in OO

Use **Replace Data Value with Object** on individual data value.

Use **Extract Class** to put together a group of fields

* Cartoon borrowed from [https://sourcemaking.com/refactoring](https://sourcemaking.com/refactoring) for didactic purposes only.
Bad Smells : Classification

- **The top crime**
- Class / method organisation
  - Large class, *Long Method*, Long Parameter List, Lazy Class, Data Class, …
- Lack of loose coupling or cohesion
  - Inappropriate Intimacy, *Feature Envy*, Data Clumps, …
- Too much or too little delegation
  - Message Chains, *Middle Man*, …
- Non Object-Oriented control or data structures
  - Switch Statements, Primitive Obsession, …
- Other : *Comments*, …
Some more bad smells

Temporary Field

Divergent Change

Speculative Generality

Alternative Classes with Different Interfaces

Incomplete Library

Refused Bequest

Comments
Temporary Field

Instance variables that are only set sometimes are hard to understand; you expect an object to need all its variables.

Use **Extract Class** to put the orphan variable and all the code that concerns it in one place.

Use **Introduce Null Object** when the variable is just around to deal with the null special case.
Divergent Change

When one class is commonly changed in different ways for different reasons.

When we make a change we want to be able to jump to a single clear point in the system and make the change. If you can’t do this you’ve got a bad smell.

To clean this up you identify everything that changes for a particular cause and use Extract Class to put them all together.

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
Speculative Generality

When someone says “I think we may need the ability to do this someday”

At this time you need all sorts of hooks and special cases to handle things that are not required

Use **Collapse Hierarchy – Inline Class – Remove Parameter – Rename Method**

* Cartoon borrowed from [https://sourcemaking.com/refactoring](https://sourcemaking.com/refactoring) for didactic purposes only.
Alternative Classes with Different Interfaces

Methods in different classes that do the same thing but have different signatures.

Use **Rename Method**

Keep using **Move Method** to move behaviour until protocols are the same

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
Incomplete Library Class

When a library or framework class doesn't provide all the functionality you need

But the solution to the problem, changing the library, is impossible since it is read-only.

Use **Introduce Foreign Method** and **Introduce Local Extension**

*See details of these refactorings for more information on how they solve the problem*
Refused Bequest

When a subclass ignores and doesn’t need most of the functionality provided by its superclass

Can give confusion and problems

You need to create a new sibling class and use **Push Down Method** and **Push Down Field** to push all the unused methods to the sibling.

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
Comments

Are comments bad?

Of course not! In general comments are a good thing to have.

But… sometimes comments are just an excuse for bad code

It stinks when you have a big comment which tries to explain bad code

such comments are used as a deodorant to hide the rotten code underneath

* Cartoon borrowed from https://sourcemaking.com/refactoring for didactic purposes only.
while (i<NRULES) {
    while (j<COL-1 && !(grammar[i][j+1].equals("N"))) {
        init(first);
        if (matrix[k][l] != 'R') {
            if (cs.indexOf(q)!=-1) {
                init(second);
                for (int p=0;p < stIndex.size();indexHeadGram++){
                    ...
                }
            }
        }
    }
}
Using refactorings, our first action is to remove the bad smells in the commented code.

After having done this, we often find that the comments have become superfluous.

To avoid bad smells we can often use **Extract Method**.

Usually the name of the new method is enough then to explain what the code does.
public double price() {
    //price is base price – quantity discount + shipping
    return quantity * itemPrice –
    Math.max(0, quantity – 500) * itemPrice * 0.05 +
    Math.min(quantity * itemPrice * 0.1, 100.0) }

public double price() {
    return basePrice – quantityDiscount + shipping }

private double basePrice() {
    return quantity * itemPrice }

private double quantityDiscount () {
    return Math.max(0, quantity – 500) * itemPrice * 0.05 }

private double shipping () {
    Math.min(quantity * itemPrice * 0.1, 100.0) }
Comments: Refactoring Patterns

Sometimes the method is already extracted but still needs a comment to explain what it does.

One solution could be: **Rename Method**
Comments: Example

```java
public double price() { return basePrice – Price2 + shipping }

private double basePrice() { return quantity *itemPrice }

// Price2 represent the quantityDiscount
private double Price2 () { return Math.max(0, quantity – 500) * itemPrice * 0.05 }

private double shipping () {Math.min(quantity * itemPrice * 0.1, 100.0) }
```

Rename method

```java
public double price() { return basePrice – quantityDiscount + shipping }

private double basePrice() { return quantity *itemPrice }

private double quantityDiscount () { return Math.max(0, quantity – 500) * itemPrice * 0.05 }

private double shipping () {Math.min(quantity * itemPrice * 0.1, 100.0) }
```
A section of code assumes something about the state of the program.

A comment is required to state the rule.

To avoid it, we can use **Introduce Assertion**
Public double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit(); }

Public double getExpenseLimit() {
    assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit(); }
Comments: Refactoring patterns (Summary)

Bad comments

- Extract Method
- Rename Method
- Introduce Assertion

No more bad comments
Comments: some last remarks...

When is a comment needed / useful?

Tip: Whenever you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.

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

A comment is a good place to say why you did something.

This kind of information helps future modifiers, especially forgetful ones, including yourself.

A last case is to use comments when something has not been done during development.
C. CONCLUSION
Problems with bad smells

Only a good recipe book and nothing more

It isn't always easy or even useful to use

Sometimes depends on context and personal style / taste

Most of them are specific to OO
Conclusion

To have a good *habitable* code:

**When?** Bad Smells

**How?** Refactorings

Bad smells are **only** a recipe book to help us find the right refactoring patterns to apply
Learning objectives:
- Definition and difference between maintenance, evolution, reuse
- Different types of maintenance
- Causes of maintenance and changes
- Technical differences of evolution and re-evolution
POSSIBLE QUESTIONS (1)

- Which bad smells could be corrected by applying the “Introduce Parameter Object” refactoring? (Mention at least two different bad smells.)
- Which refactorings would you probably apply to address the “Large Class” bad smell?
- Explain and illustrate one of the following bad smells: Long Method, Feature Envy or Middle Man.
- Explain the Long Parameter List bad smell in detail. Why is it a bad smell? How could it be solved with a refactoring?
- What’s the relation between the Long Parameter List bad smell and the Data Clumps bad smell?
BAD CODE SMELLS

POSSIBLE QUESTIONS (2)

- Explain and illustrate what the notion of “coupling” is. Should we strive for loose coupling or tight coupling? What bad smell describes a situation that violates this principle? Name and explain at least one.

- Explain and illustrate what the notion of “cohesion” is. Should we strive for low cohesion or high cohesion? What bad smell describes a situation that violates this principle? Name and explain at least one.

- Some bad smells are based on the principle that “things that change together should go together”. Explain one of these bad smells, and the principle on which they are based, in detail.

- Name and explain at least one bad smell that explains a problem related to bad use of inheritance.

- When talking about “Comments” in the bad smells theory session, it was stated that comments are sometimes just there because the code is bad. Can you give an example of this and how such comments could become superfluous simply by refactoring the code?
CLASS... IS... DISMISSED.