# Bad smells of code

Lecture handouts

Haroon Zafar

NUCES, Peshawar Campus

# Bad smells in code

- Martin Fowler, Refactoring: Improving the design of existing code. Addison Wesley.

# Introduction

- Bad Smells=“Bad smelling code”
  - Indicate that your code is ripe for refactoring

- Refactoring is about
  - How to change code by apply refactoring.

- Bad smells are about
  - When to modify it

# Bad Smells

- Allow us to identify
  - What needs to be change in order to improve the code
- A recipe book to help us to choose the right refactoring path
  - No precise criteria
  - More to give an intuition and indications
- Goal: a more “Habitable” code.

# Habitable code

- Habitable code is in which developer feels at home while developing a software system.

  (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.

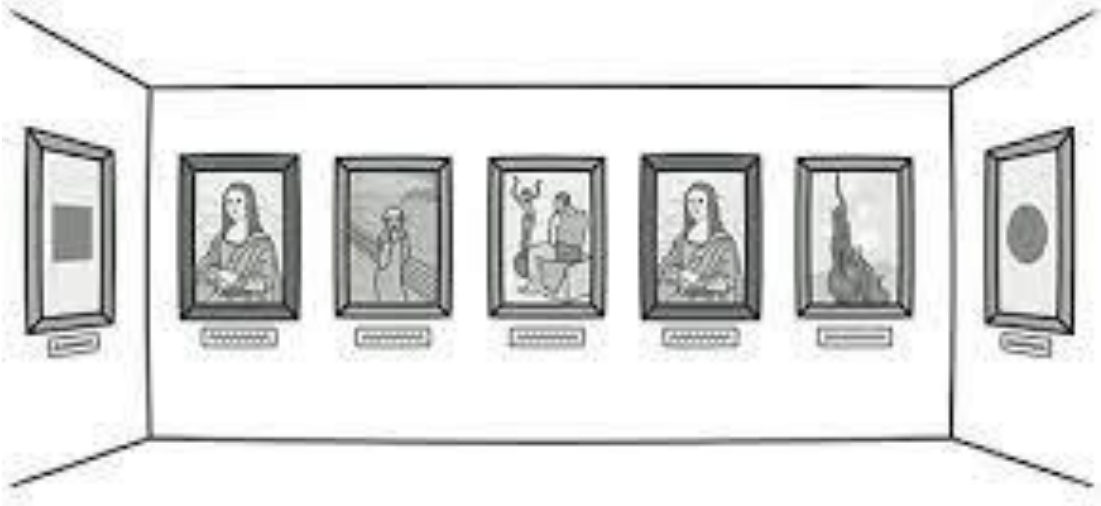
# Bad smells Classification

- Bloaters (Top crime)
- Class/method organization
- Lack of loose coupling or cohesion
- Too much or too little delegations
- Non Object Oriented control or data structure
- Mixed methods

# Alternative Classification

- Bloaters
- Object Oriented abusers
- Change preventers
- Dispensables
- Couplers
- Other smells

# Bloaters/Top crime



Code duplication

# Code Duplication

- Duplicated code is the number 1 in the stink parade:

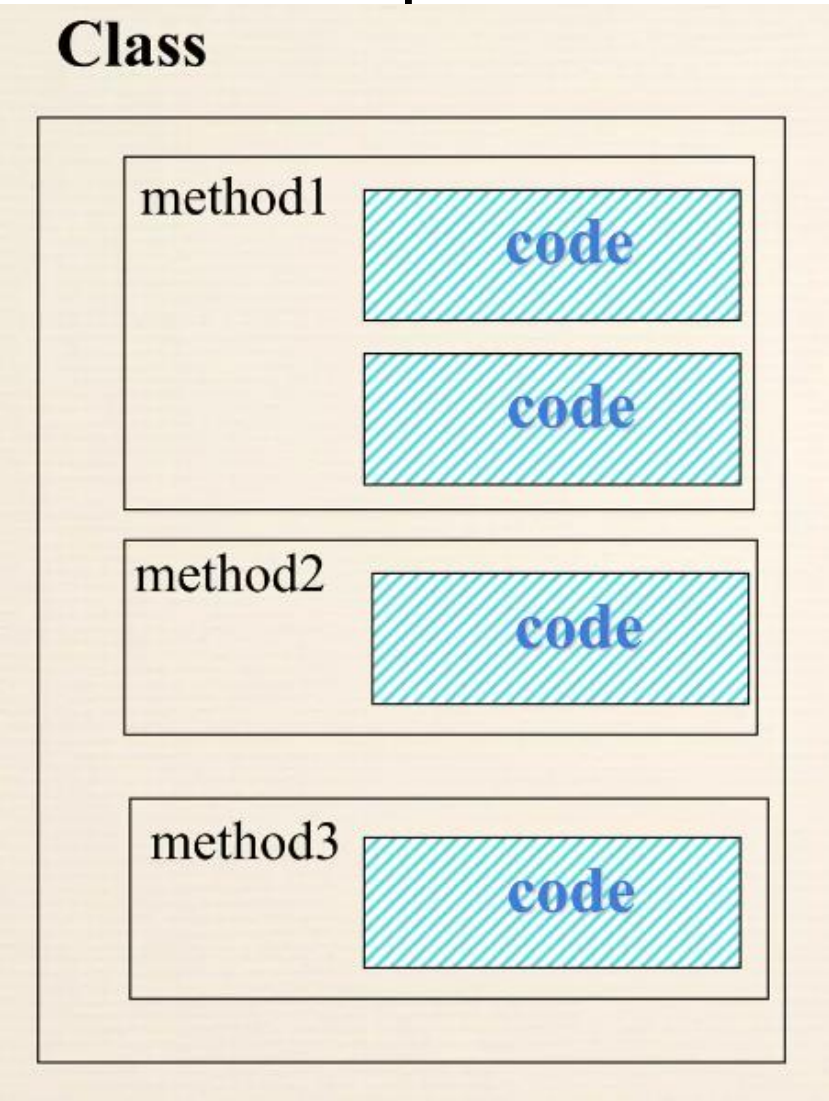
  We have duplicated code when we have the same code structure in more then one place

- Why is duplicated code bad?

# Code duplication: Example

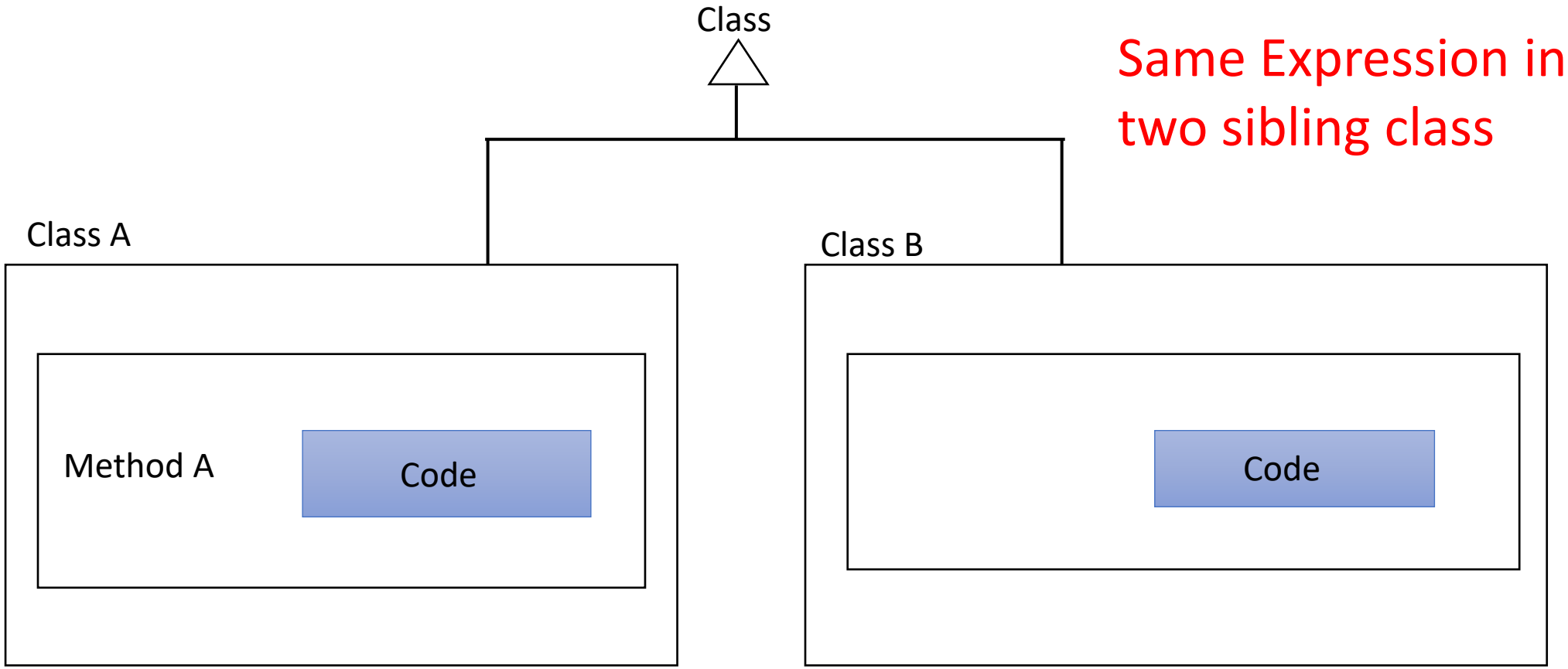
```
public double ringSurface(r1,r2) {
        // calculate the surface of the first circle
        double surf1 = bigCircleSurface(r1);
        // calculate the surface of the second circle
        double surf2 = smallCircleSurface(r2);
        return surf1 - surf2;
}
private double bigCircleSurface(r1) {
        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




Same expression in two or more methods of the same class

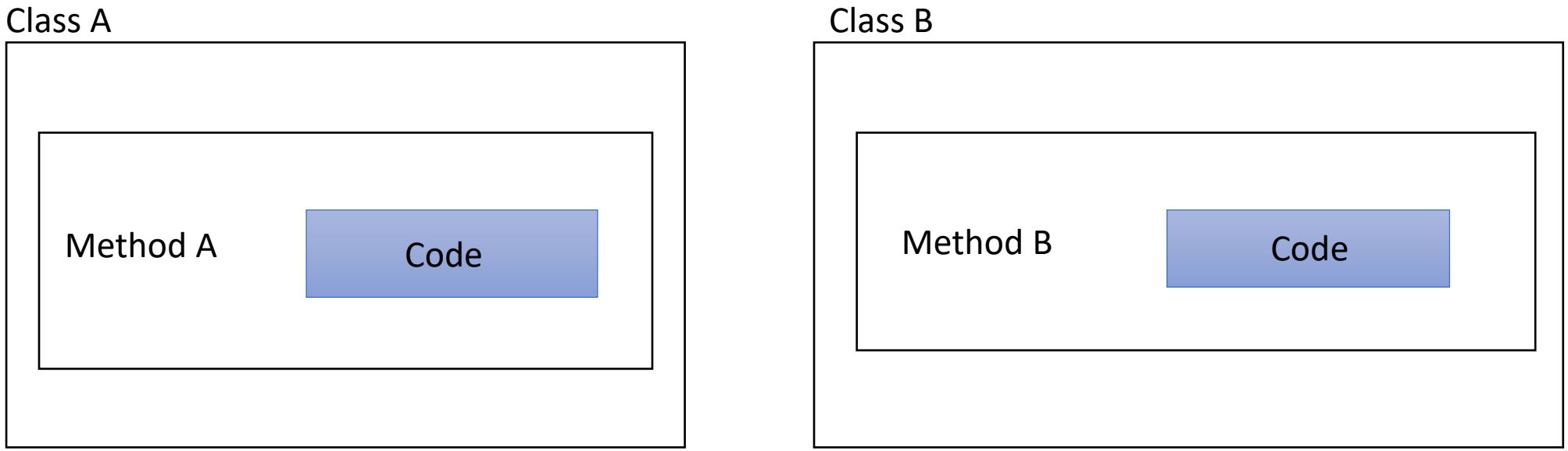
# Code Duplication: Example 3




Same Expression in two sibling class

# Code Duplication: Example 4

Same expression in
two unrelated class



# Class/Method Organization

- Large class
- long Method
- Long Parameter List
- Lazy class
- Data class

# Large Class

- A large class is a class that is trying to do too much tasks
- Often show up as too many instance variables
- Use Extract class or Extract subclass to bundle variable
  - Choose variable that belong together in the extracted class
  - Common prefixes and suffixes may suggest which ones may go together e.g depositAmont and depositCurrency

# Large Class

- 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

# Long Parameter List

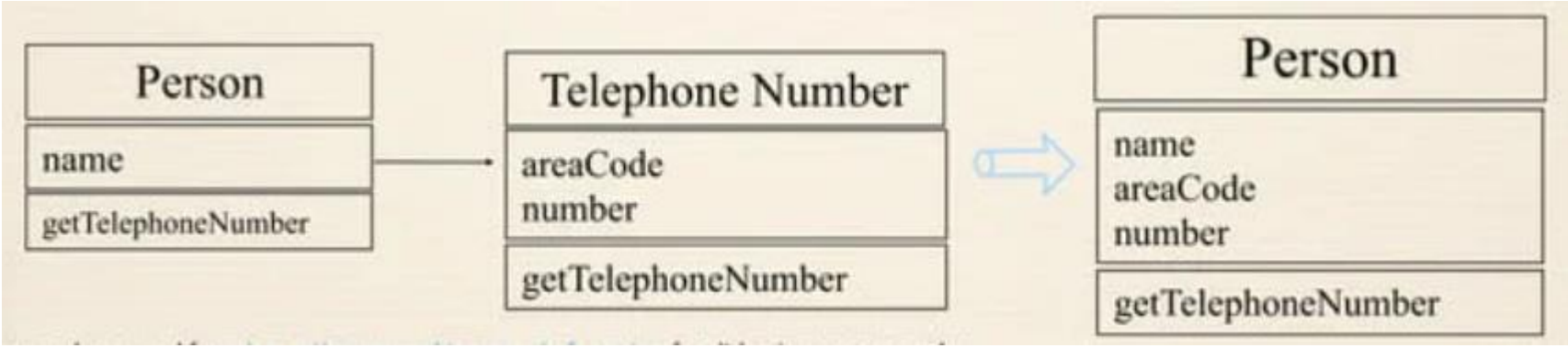
- In procedural programming languages, we pass as parameters everything needed by a subroutine
- Because the only alternative is ***global variable***
- With objects you don’t pass everything the method needs

# Long Parameter List

- Long Parameter list are hard to understand
- Pass only the needed number of variables
- Use Replace parameter with methods when you can get the data in one parameter by making a request of an object you already known 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 eliminated the class



# Data Class

- Classes with just fields, getter, setter and nothing else
- If there are public fields, use Encapsulation Field
- For fields that should not be changed use remove setting method

# Long Method

- Object programs live best and longest with short methods
- New OO programs feel that OO programs are endless sequence of delegation
- Older languages carried an overhead in subroutine calls which deterred people from that small methods
  - There is still an overhead to the reader of the code because you have to switch context to see what subprocedure does.

- Important to have a good name for small method
  - Rename method

# Long Method: Example

```
void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;
        // Print banner
        System.out.println("*******************");
        System.out.println("***** Customer *****");
        System.out.println("*******************");
        // Calcultate 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 methods that seems 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

# Long Method: Refactoring Patterns

- But how to identify the clumps of the code to extract?
- Look for comments…
  - A block od statements with a comments that tells you what it is doing can be replaced b a method whose based on the comments
- Loops also give for extraction…
  - Extract the loop and code within the loop into its own method.

# 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("**********************");
        // Calcultate 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("**********");
        // Calcultate 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;
        printBanner();
        // Calcultate 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("*******************");
}
```

1.
Extract Method

# Long Method Example revisited

```
void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;
        printBanner();
        // Calcultate 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

```
void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;
        printBanner();
        // Calcultate 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 variable

# Long Method Example revisited

```
void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;
        printBanner();
        // Calcultate 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() { … }
```

# Long Method Example revisited

```
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*

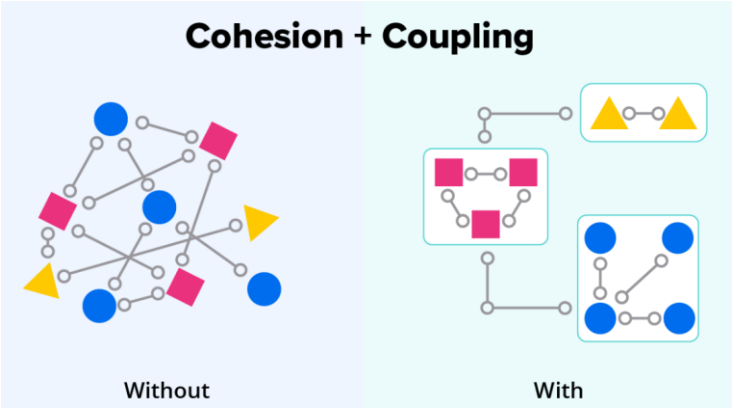
Reassigning a Local Variable

# Lack of loose coupling or cohesion

- Coupling and Cohesion
- Inappropriate Intimacy
- Data clumps
- Feature Envy
- Shotgun surgery

# Coupling and cohesion

- Coupling is the degree to which different software components depends 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

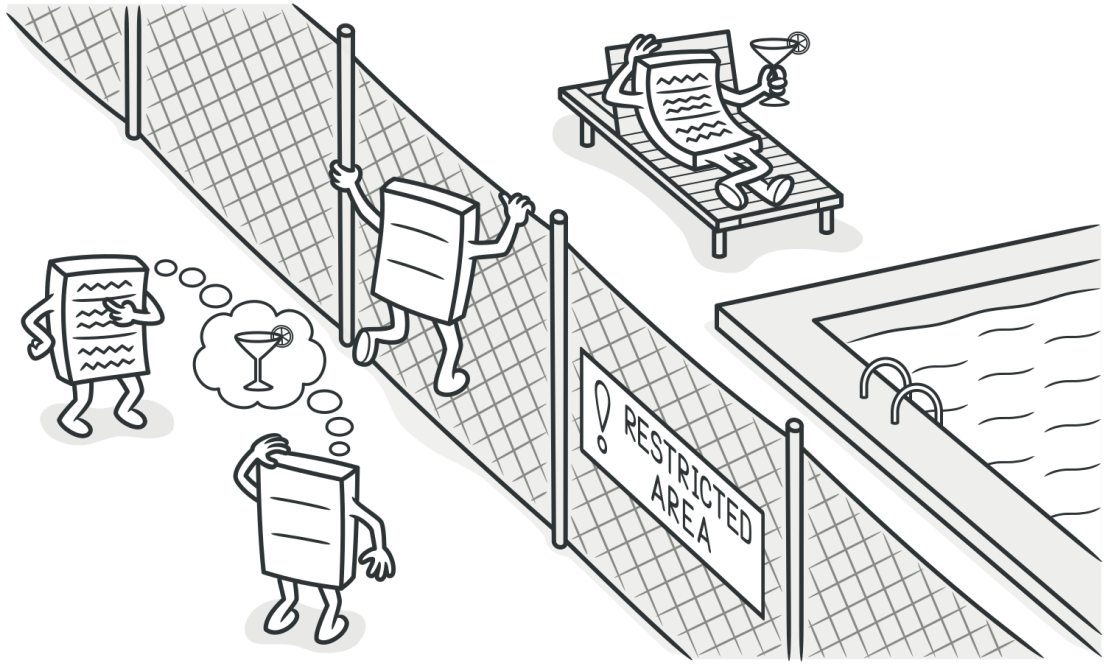
- Pairs of classes that know too much about each other's private details
  - Use **Move Method** and **Move Fields** to separate the pieces to reduce intimacy
- If subclasses know more about their parents than their parents would like them to know
- Apply Replace Inheritance with delegation

# Data Clumps

- A certain number of data items in a lots of places
- Example: Fields in a couple of classes, parameters in many method signatures
- Ought to be made into their own objects
- When the clumps fields, use Extract class to turn them into an object
- When the clumps are parameters, use Introduce parameters objects to slim them down

# Feature Envy

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

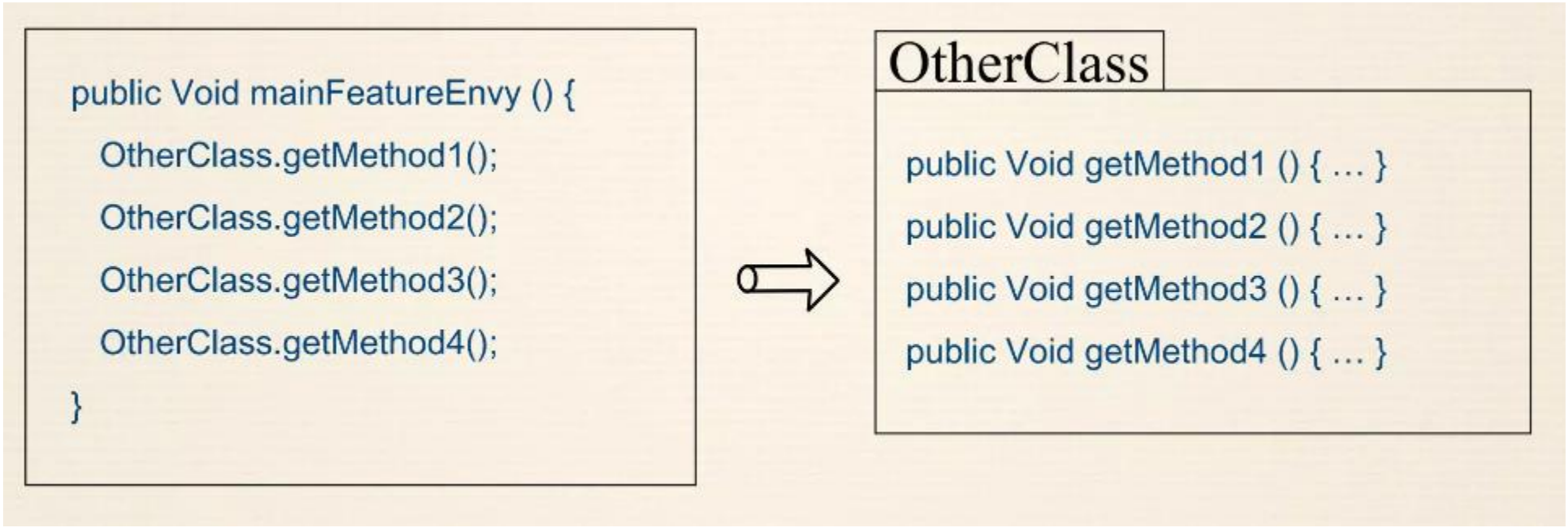


# Feature Envy

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

- Why is it bad to invoke a zillion time 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



# Feature Envy: Refactoring Pattern

First solution : **Move Method**

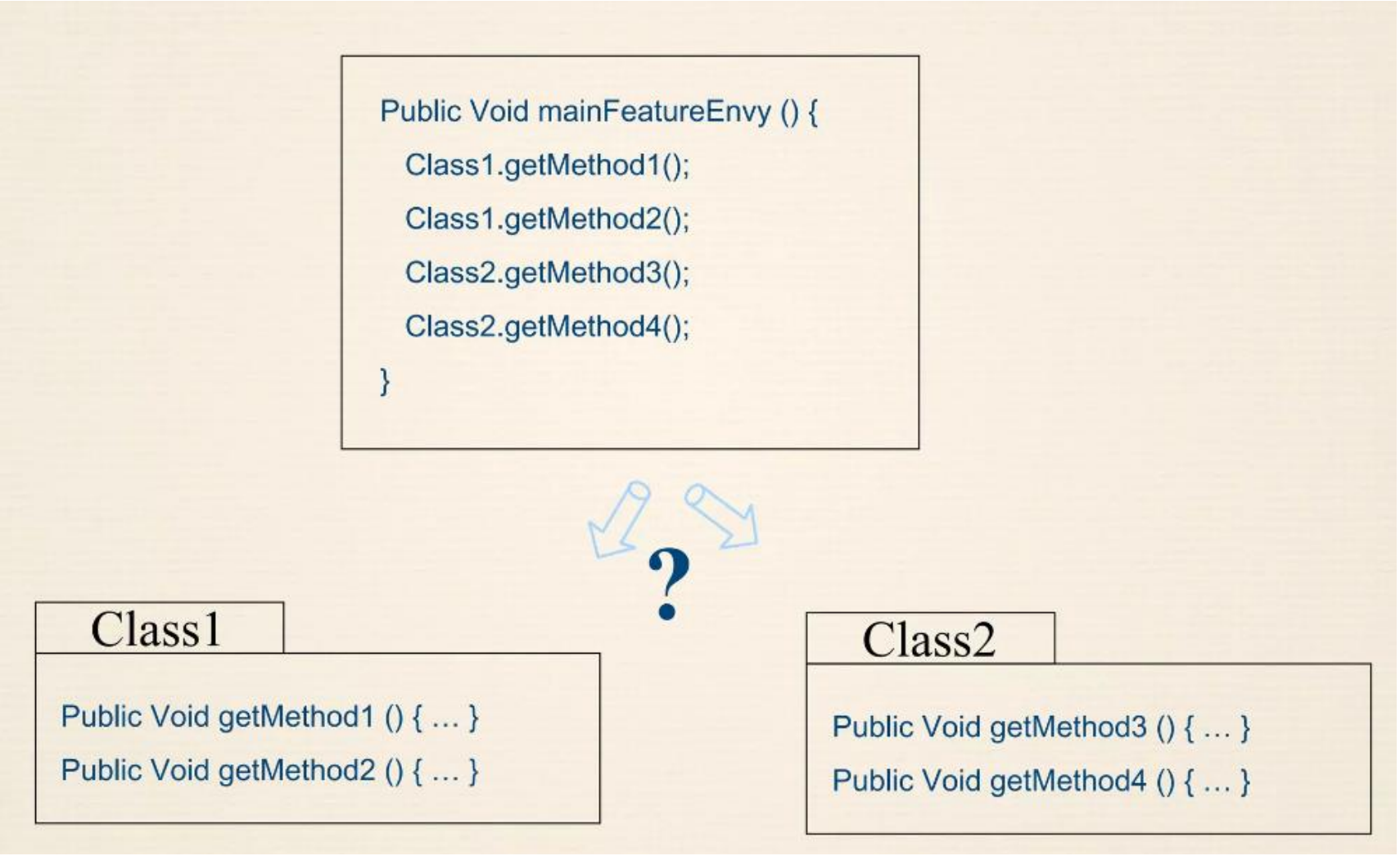
OtherClass

```
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)



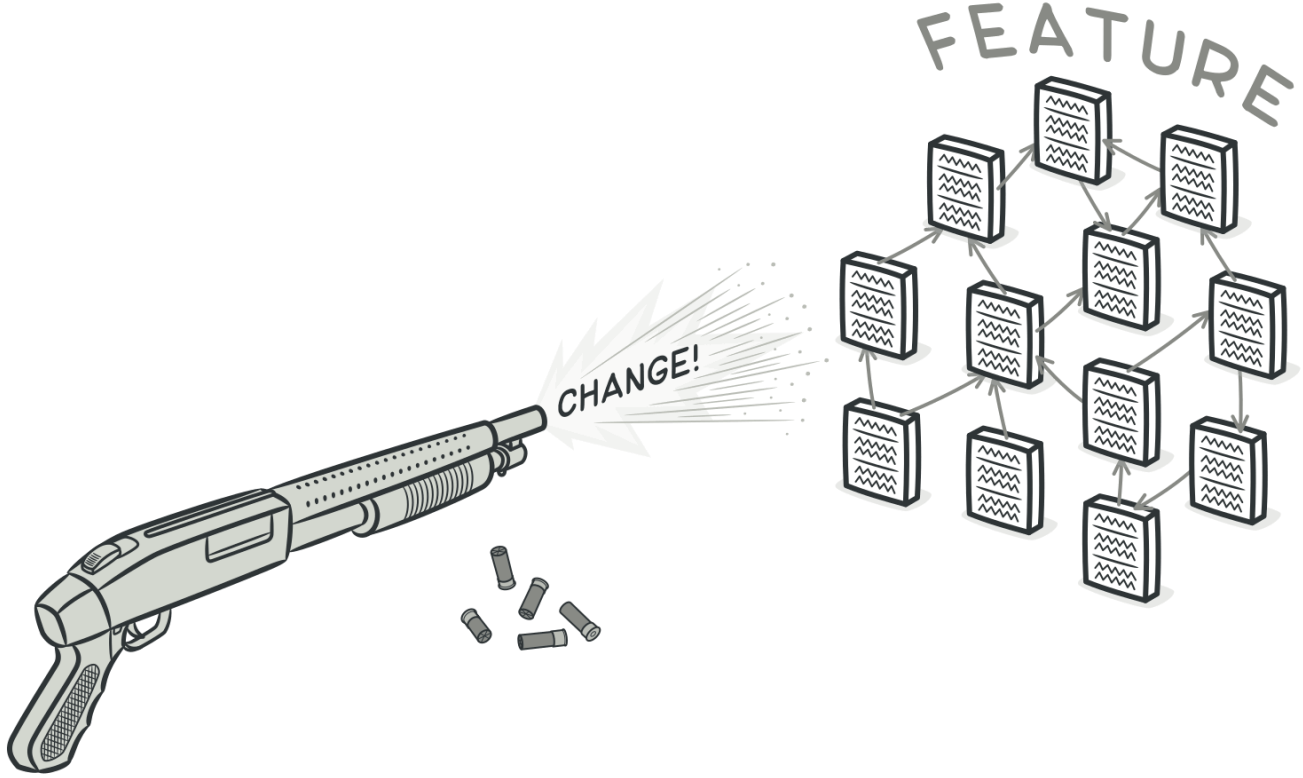
# Feature Envy: Refactoring Pattern

- 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 the most of the data and put the method with that data*

# Shotgun Surgery

- When making one kind of change requires many small changes to a lot of different classes



# 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 Fields**

# Too much or too little delegation

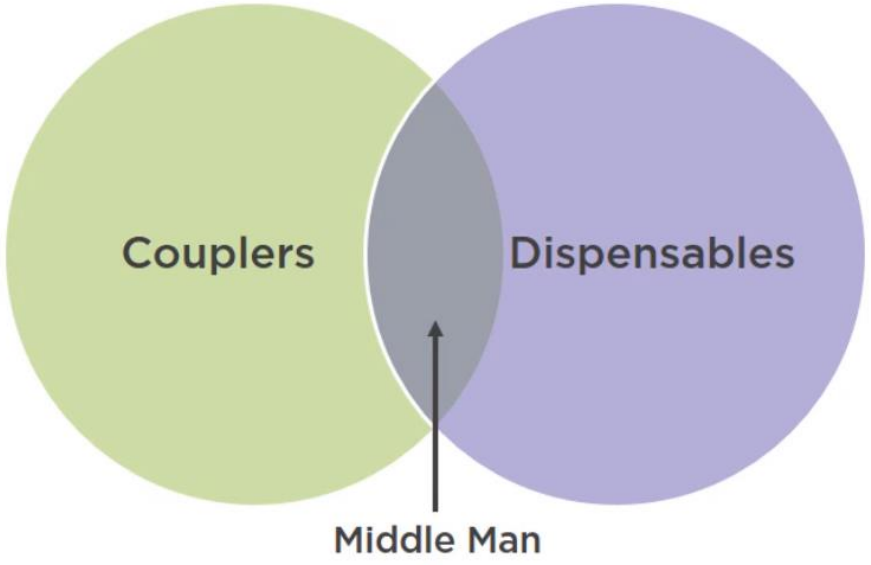
- Message chains
- Middle man

# Message chain

- A client asks an object for another object who then ask 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

# Middle Man

- Object hide internal details (encapsulation)
  - Encapsulation leads to delegation
- It is a good concept but…
  - Sometimes it goes to far



# Middle Man

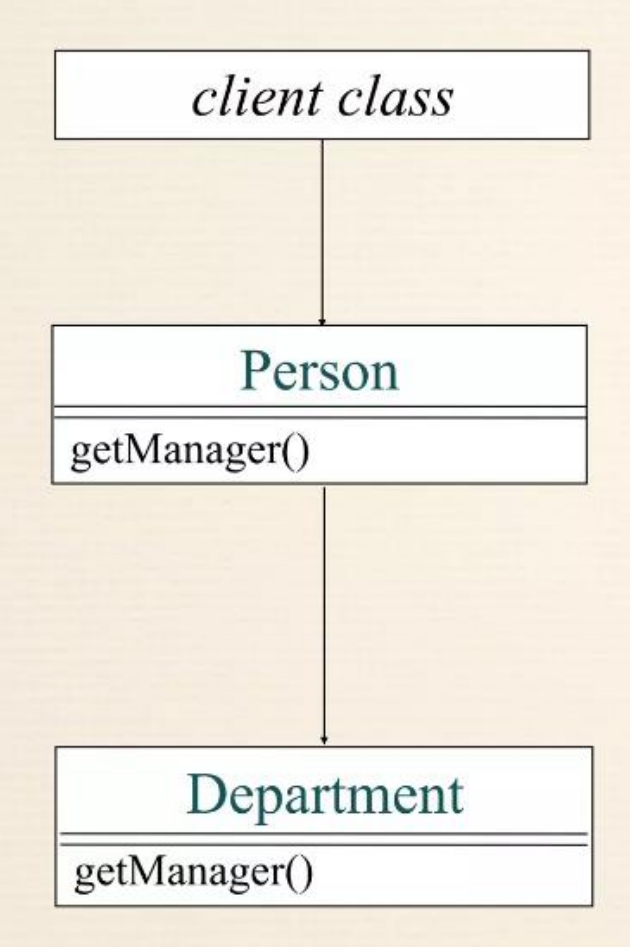
- Real life example:
  - You ask a director whether he is free for a meeting
  - He delegates the message to his secretary that delegates it into the dairy
  - Everything is good…but, if the secretary has nothing else to do, it is better to remove her!

# Middle Man

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

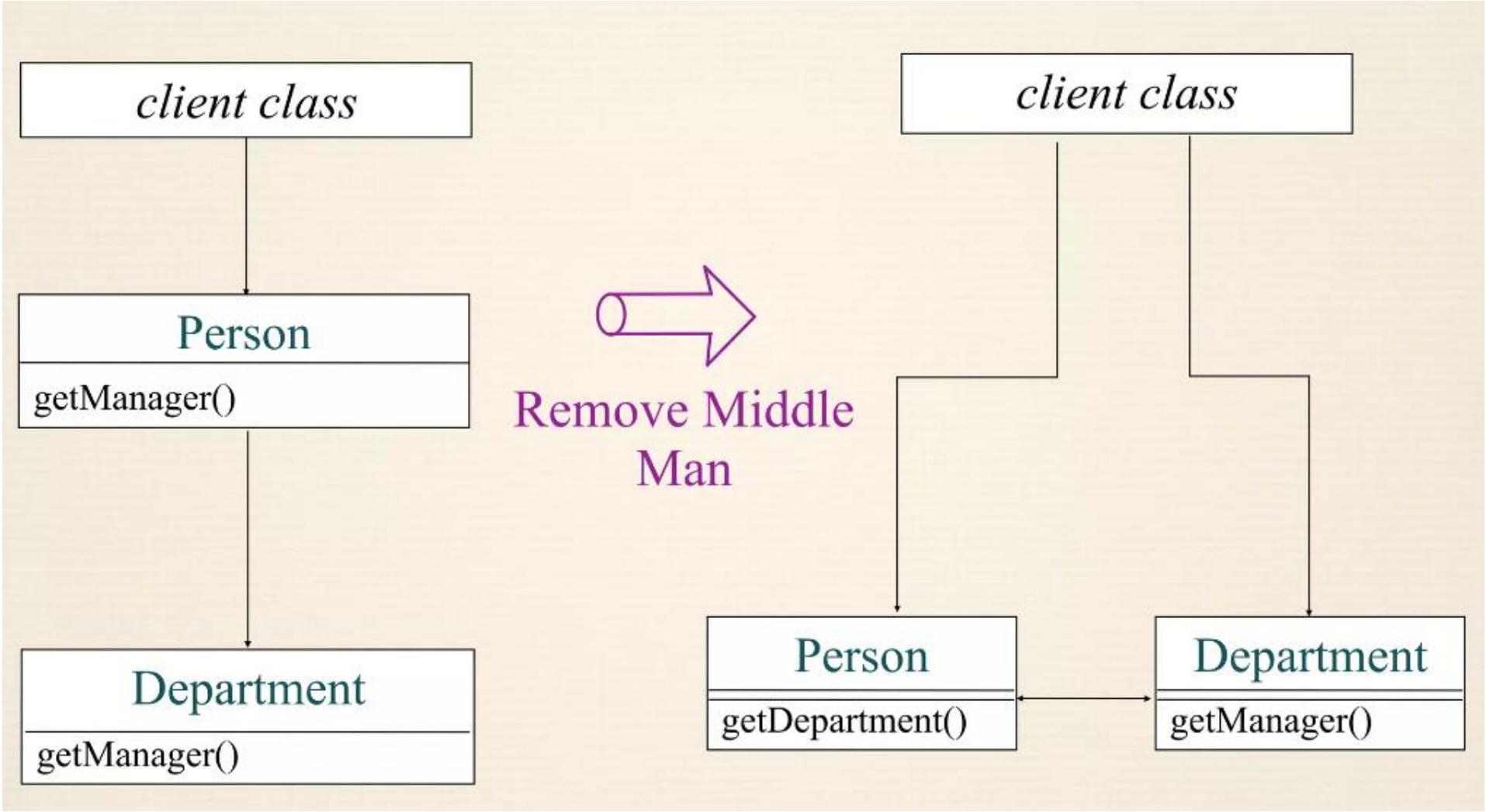
- Sometimes most methods of a class just delegate to another class.

# Middle Man: Example



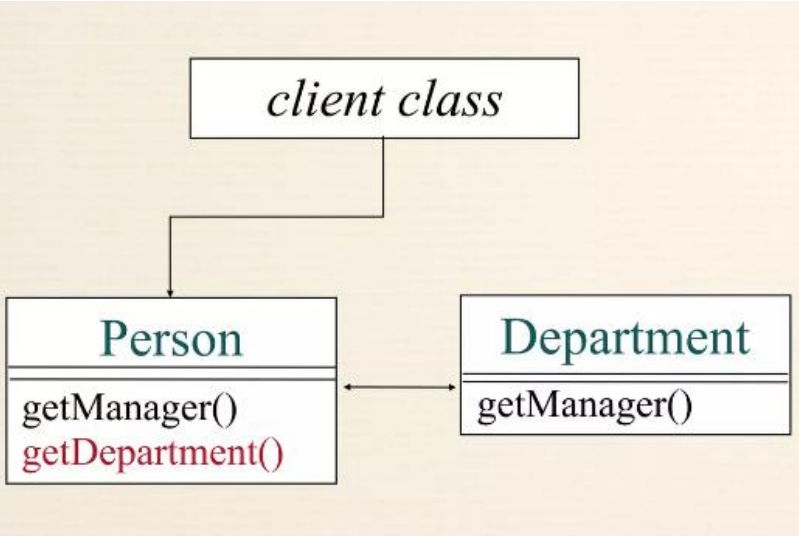

```
class Person{
        Department depart;
        public Person getManager(){
                return depart.getManager();
            }
class Department
        private Person manager;
        public Department (Person manager){
            this.manager=manager;
}
        public Person getManager(){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

# Middle Man: Refactoring Pattern



# Middle Man: Refactoring Pattern

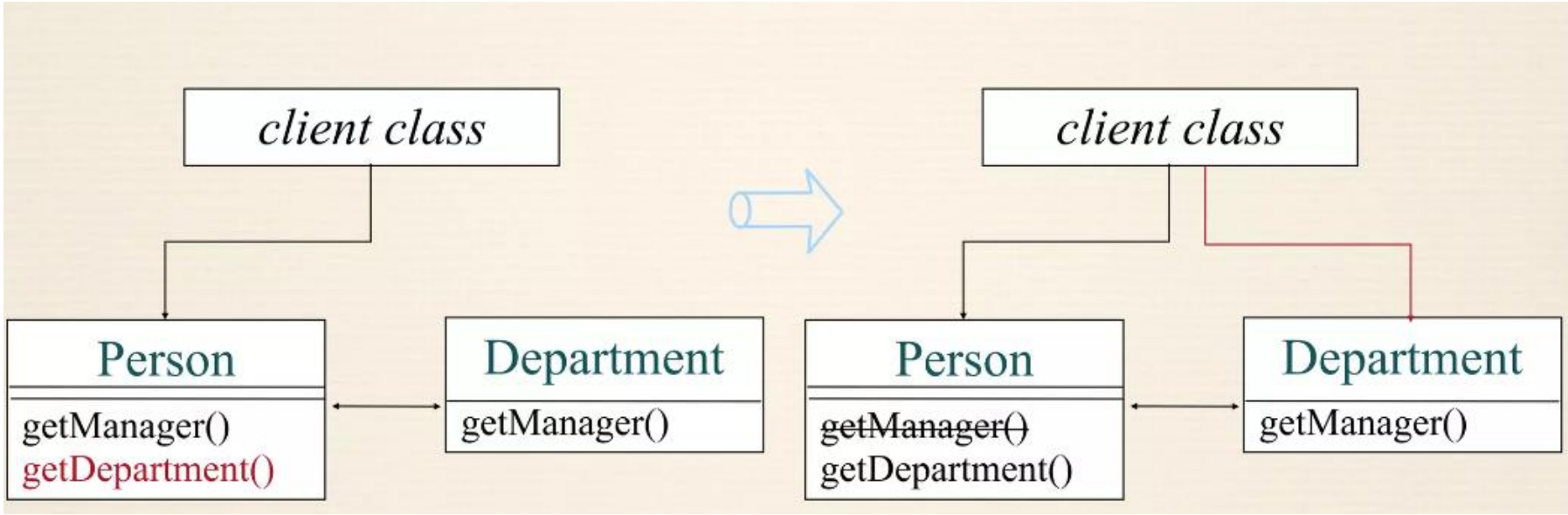
- Remove Middle Man…
- **First step:** Create an accessor for the delegation.




```
class Person{
                    Department depart;
                    public Person getManager(){
                            return depart.getManager();
}
public Department getDepartment(){
            return depart;
}
}
```

# Middle Man: Refactoring Pattern

- **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




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

# Middle Man: Refactoring Pattern

- Last step: Compile and test.

# Non Object-Oriented control or data structure

- Switch statement
- Primitive obsession

# 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 way 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 Condition with Polymorphism after you setup the inheritance structure.

# Switch Statement: Example

```
switch(input)
{
   case "a":
     CreateValue("Hello");
     myData+= "Hello";

   case "b":
     CreateValue("Hi");
     myData+= "Hi";

   case "C":
     CreateValue("Hey");
     myData+= "Hey";

   default:
     myData = string.empty;
}
```

Alternate Solution!

# Primitive Obsession

- Characterized 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 Objects on individual data value.
- Use Extract class to put together a group of fields

# Some other bad smells

- Comments
- Divergent class etc.

# 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

# Comment: Example

- */Look which rule to choose, and after we know which rule to take we initialize the array matrix with the correct value (depending on the rule). We do that until we have tested all rules and after that we……*/

```
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++){
                     …
```

# Problems with bad smells

- Only a recipe book , but nothing else
- Most of them are related to OO.

# Conclusion

- To have a habitable code:
  - **When?** Bad Smells
  - **How?** Refactoring