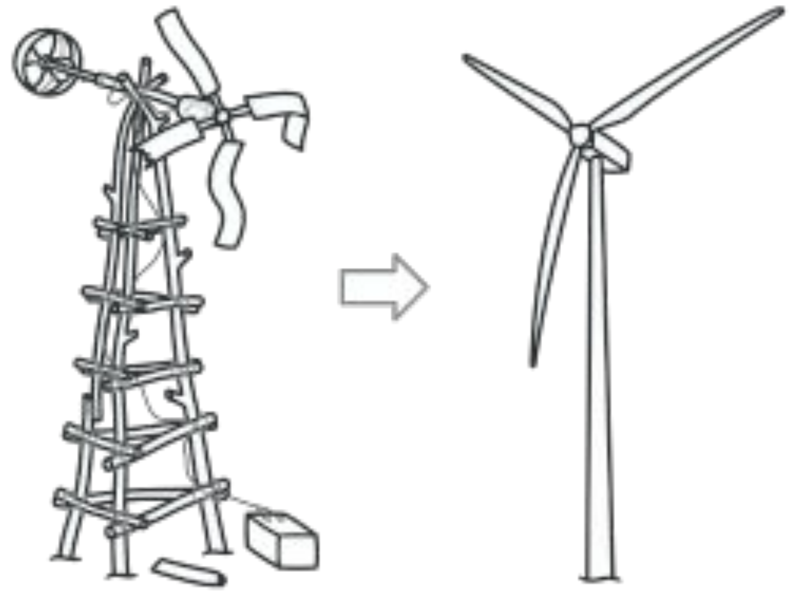
CODE REFACTORING

DEEPTHI R

(20PW08)

WHAT IS REFACTORING?

- Refactoring is a systematic process of improving code without creating new functionality that can transform mess into clean code and simple design.
- In simple terms, refactoring is fighting technical debt.
- Clean Code
  - Obvious to other programmers
  - Does not contain duplication
  - Contains minimum number of classes
  - Passes all the test cases
  - It is cheaper and easier to maintain



TECHNICAL DEBT

- The metaphor of "technical debt" in regards to unclean code was originally suggested by Ward Cunningham.
- If you get a loan from a bank, this allows you to make purchases faster. You pay extra for expediting the process - you don't just pay off the principal, but also the additional interest on the loan.
- Needless to say, you can even rack up so much interest that the amount of interest exceeds your total income, making full repayment impossible.
- The same thing can happen with code. You can temporarily speed up without writing tests for new features, but this will gradually slow your progress every day until you eventually pay off the debt by writing tests.

CAUSES OF TECHNICAL DEBT

- Business pressure
- Lack of understanding of the consequences of technical debt
- Failing to combat the strict coherence of components
- Lack of tests
- Lack of documentation
- Lack of interaction between team members
- Long-term simultaneous development in several branches
- Delayed refactoring
- Lack of compliance monitoring
- Incompetence

REFACTORING – RULE OF THREE

- **Rule of Three**
  - When you're doing something for the first time, just get it done.
  - When you're doing something similar for the second time, cringe at having to repeat but do the same thing anyway.
  - When you're doing something for the third time, start refactoring.
- If a certain code snippet causes your team trouble three times, it is time to put some work into making it simpler.

WHEN TO REFACTOR?

- **When adding a feature**
  - Refactoring helps you understand other people's code. If you have to deal with someone else's dirty code, try to refactor it first. Clean code is much easier to grasp. You will improve it not only for yourself but also for those who use it after you.
  - Refactoring makes it easier to add new features. It's much easier to make changes in clean code.
- **When fixing a bug**
  - Bugs in code behave just like those in real life: they live in the darkest, dirtiest places in the code. Clean your code and the errors will practically discover themselves.
  - Proactive refactoring as it eliminates the need for special refactoring tasks later.
- **During a code review**
  - This way you could fix simple problems quickly and gauge the time for fixing the more difficult ones.

HOW TO REFACTOR?

- Checklist of refactoring done the *right* way
  - The code should become cleaner

    Even after refactoring, the code may not become cleaner. It frequently happens when you move away from refactoring with small changes and mix a whole bunch of refactoring into one big change. But it can also happen when working with extremely sloppy code. Whatever you improve, the code as a whole remains a disaster. Better to rewrite the entire erroneous part.
  - New functionality shouldn't be created during refactoring.

    Don't mix refactoring and direct development of new features. Try to separate these processes at least within the confines of individual commits.

CODE SMELLS

- Code Smells are not the bugs of the program. With code smells too, your program might work just fine. They do not prevent the program from functioning or are incorrect.
- They just signify the weakness in design and might increase the risk of bugs and program failure in the future.
- Code Smells motivates for Code Refactoring, the size of code decreases, confusing coding is properly restructured.
- Code refactoring is defined as the process of restructuring computer code without changing or adding to its external behavior and functionality.

CHANGE PREVENTERS

- These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.
- Divergent change
- Shotgun surgery
- Parallel inheritance hierarchies

DIVERGENT CHANGE

- Divergent Change smells occur when a single class needs to be edited many times when changes are made outside the class.
- **Signs and Symptoms** : You find yourself having to change many unrelated methods when you make changes to a class.
- **Reason** : Poor program structure or "copypasta programming”.
- **Treatment** :
  - Split up the behavior of the class via Extract Class.
  - If different classes have the same behavior, you may want to combine the classes through inheritance (Extract Superclass and Extract Subclass).

CONT..

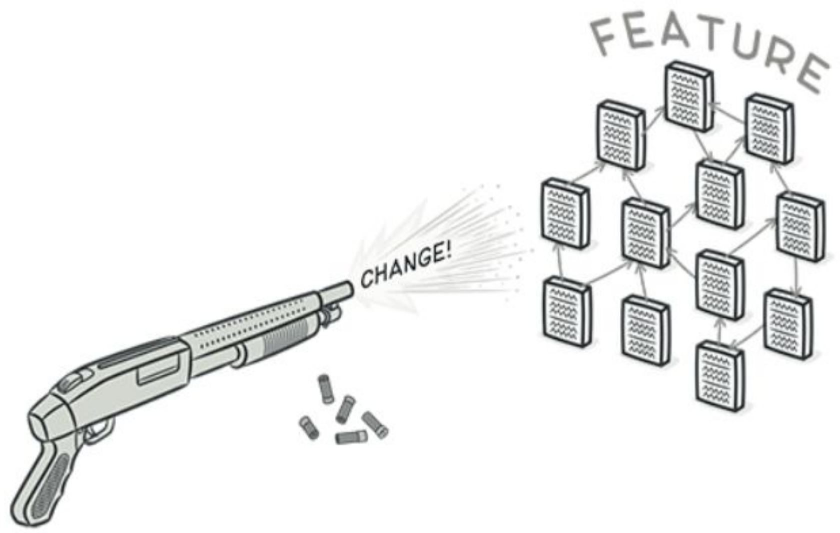
- **Payoffs**
  - Improves code organization.
  - Reduces code duplication.
  - Simplifies support.

SHOTGUN SURGERY

- **Signs and Symptoms** : Making any modifications requires that you make many small changes to many different classes.
- **Reason** : A single responsibility has been split up among a large number of classes. This can happen after overzealous application of Divergent Change.
- **Treatment** :
  - Use Move Method and Move Field to move existing class behaviors into a single class. If there’s no class appropriate for this, create a new one.
  - If moving code to the same class leaves the original classes almost empty, try to get rid of these now-redundant classes.

CONT..

- **Payoffs**
  - Better organization.
  - Less code duplication.
  - Easier maintenance.

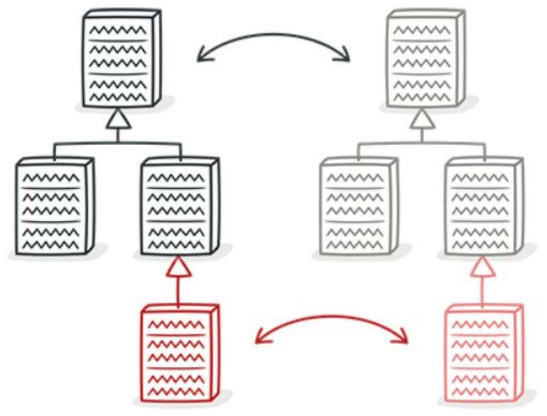


PARALLEL INHERITANCE HIERARCHIES

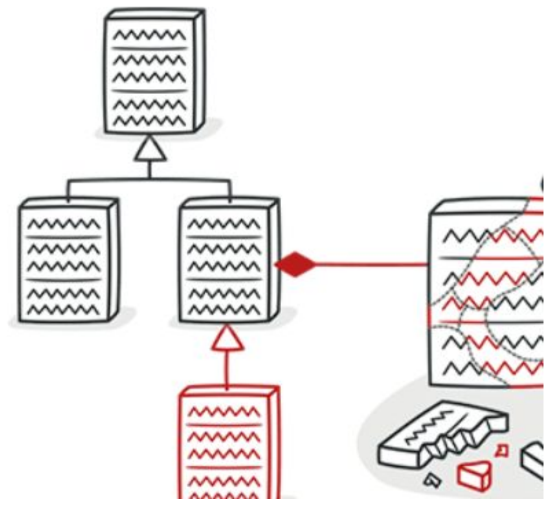
- **Signs and Symptoms** : Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.
- **Reason** : All was well as long as the hierarchy stayed small. But with new classes being added, making changes has become harder and harder.
- **Treatment** :
  - You may deduplicate parallel class hierarchies in two steps. First, make instances of one hierarchy refer to instances of another hierarchy.
  - Then, remove the hierarchy in the referred class, by using Move Method and Move Field.

CONT..

- **Payoffs**
  - Reduces code duplication.
  - Can improve organization of code.



to



REFERENCE

https://refactoring.guru/refactoring/smells/change-preventers

HEMA VARSHIKA V

(20PW11)

BLOATERS

WHAT IS BLOATERS?

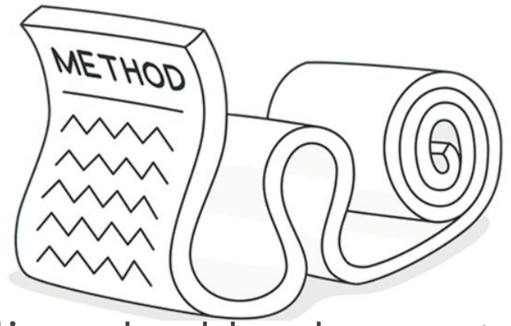
Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with.

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).

TYPES:

- **Long Method**
- **Large Class**
- **Primitive Obsession**
- **Long Parameter List**
- **Data Clumps**

LONG METHOD



Signs and Symptoms:

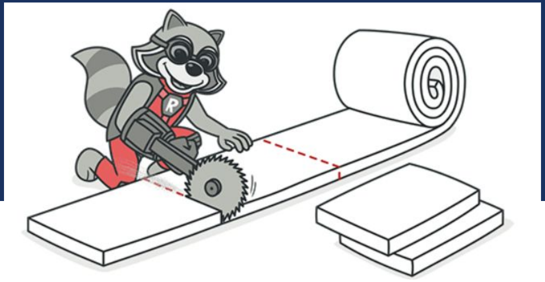
A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.

Reason for the problem

Since it's easier to write code than to read it, this "smell" remains unnoticed until the method turns into an ugly, oversized beast.

Mentally, it's often harder to create a new method than to add to an existing one: "But it's just two lines, there's no use in creating a whole method just for that..." Which means that another line is added and then yet another, giving birth to a tangle of spaghetti code.

SOLUTION

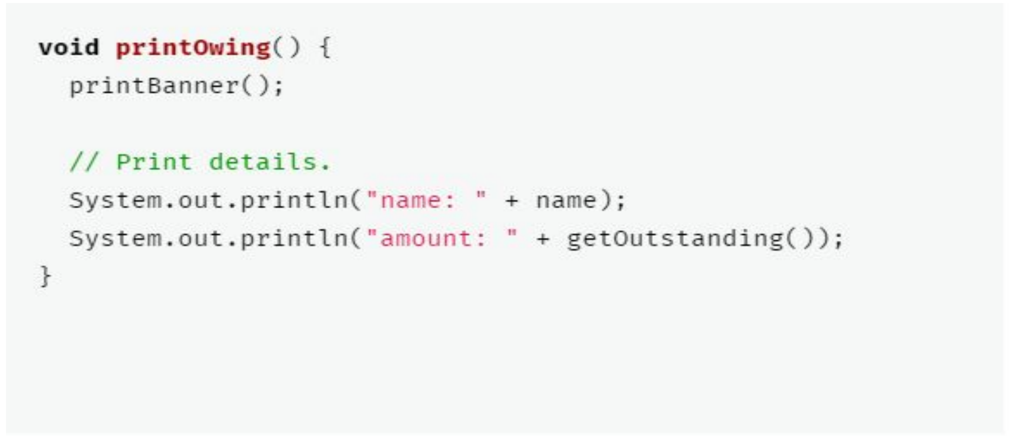


If you feel the need to comment on something inside a method, you should take this code and put it in a new method.

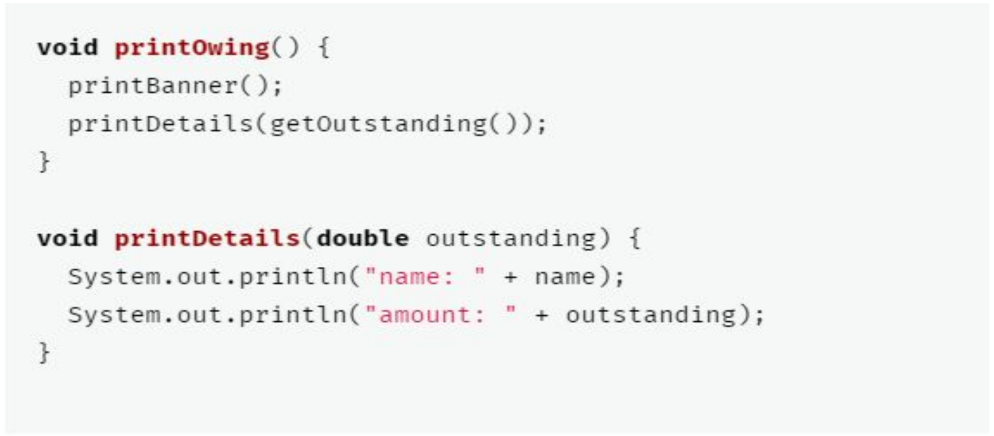
Even a single line can and should be split off into a separate method, if it requires explanations.

And if the method has a descriptive name, nobody will need to look at the code to see what it does.

To reduce the length of a method body, use **Extract Method**.

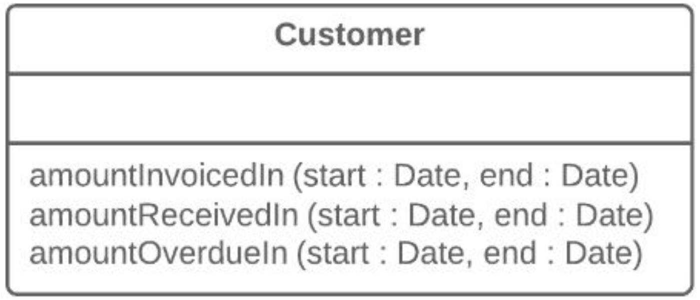


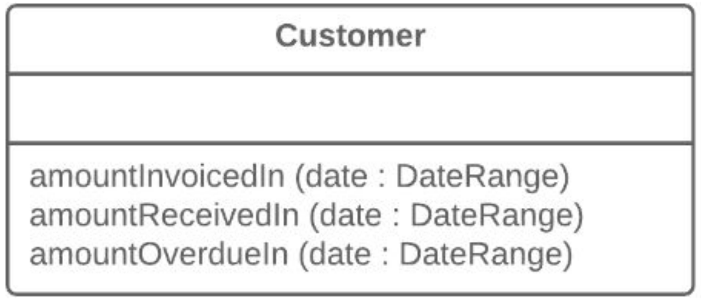
```java
void printOwing() {
  printBanner();

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



```java
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

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

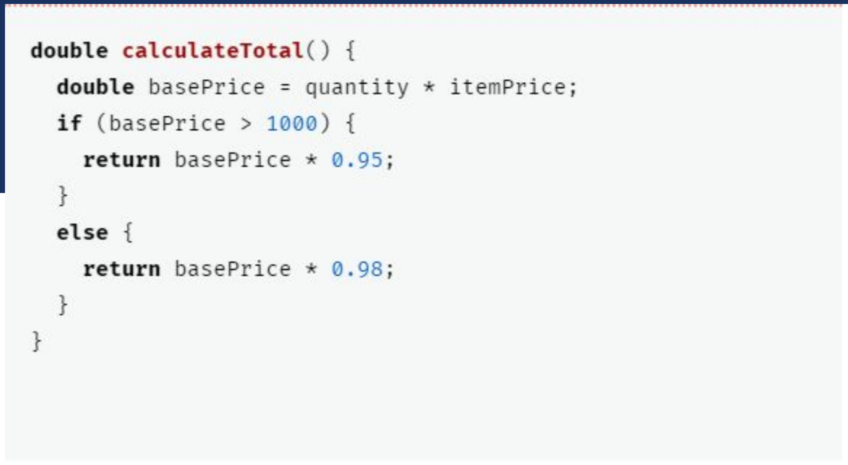
If local variables and parameters interfere with extracting a method **Introduce Parameter Object** , **Preserve Whole Object** and , use **Replace Temp with Query**.


Customer
amountInvoicedIn (start : Date, end : Date)
amountReceivedIn (start : Date, end : Date)
amountOverdueIn (start : Date, end : Date)

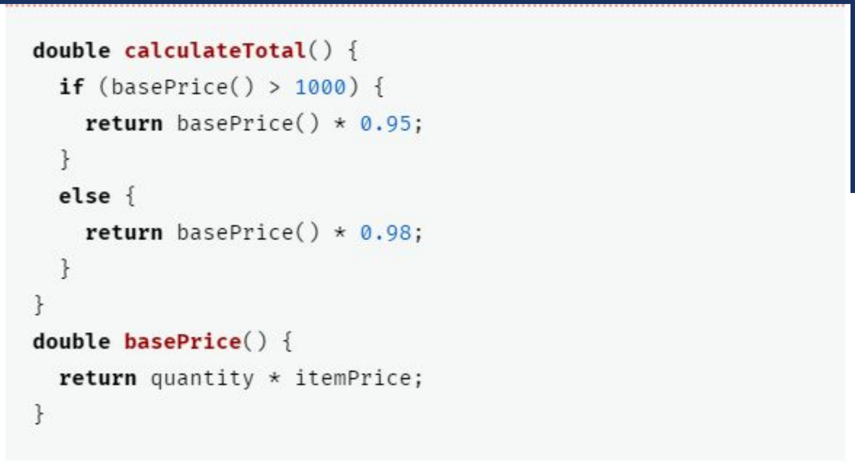

Customer
amountInvoicedIn (date : DateRange)
amountReceivedIn (date : DateRange)
amountOverdueIn (date : DateRange)

```java
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

```java
boolean withinPlan = plan.withinRange(daysTempRange);
```

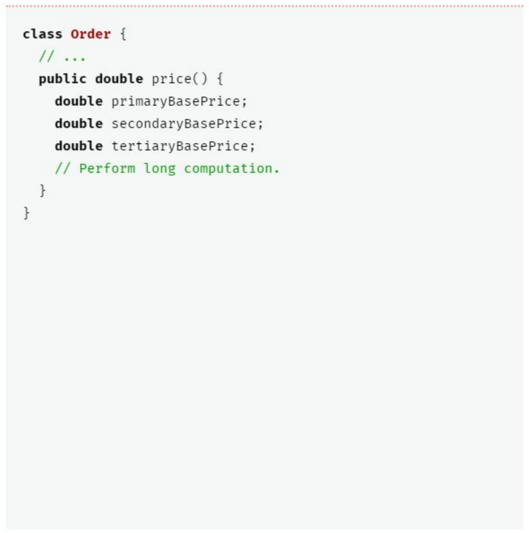


```
double calculateTotal() {
  double basePrice = quantity * itemPrice;
  if (basePrice > 1000) {
    return basePrice * 0.95;
  }
  else {
    return basePrice * 0.98;
  }
}
```

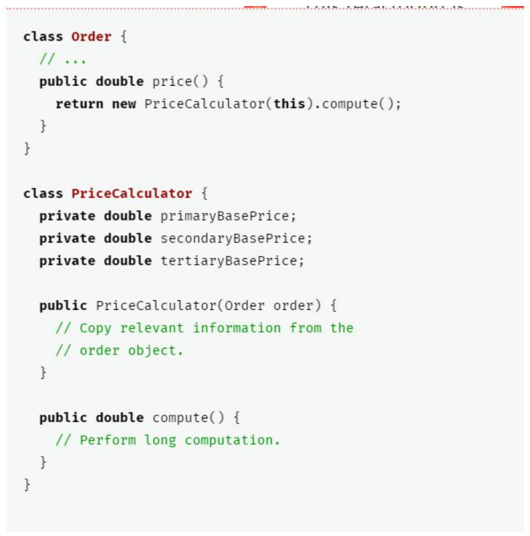


```
double calculateTotal() {
  if (basePrice() > 1000) {
    return basePrice() * 0.95;
  }
  else {
    return basePrice() * 0.98;
  }
}
double basePrice() {
  return quantity * itemPrice;
}
```

- If none of the previous recipes help, try moving the entire method to a separate object via **Replace Method with Method Object**



```
class Order {
  // ...
  public double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // Perform long computation.
  }
}
```



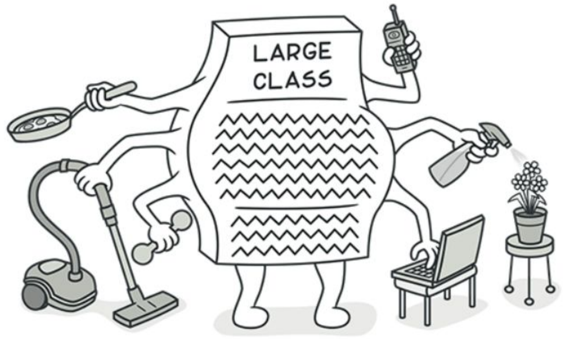
```
class Order {
  // ...
  public double price() {
    return new PriceCalculator(this).compute();
  }
}

class PriceCalculator {
  private double primaryBasePrice;
  private double secondaryBasePrice;
  private double tertiaryBasePrice;

  public PriceCalculator(Order order) {
    // Copy relevant information from the
    // order object.
  }

  public double compute() {
    // Perform long computation.
  }
}
```

Payoff

Among all types of object-oriented code, classes with short methods live longest. The longer a method or function is, the harder it becomes to understand and maintain it.

In addition, long methods offer the perfect hiding place for unwanted duplicate code.

LARGE CLASS



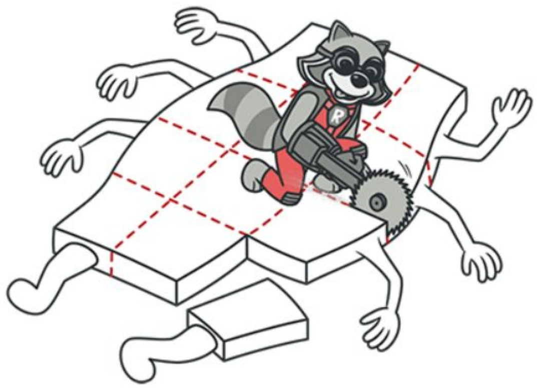
Sign and Symptoms

A class contains many fields/methods/lines of code.

Reason for the Problem

Classes usually start small. But over time, they get bloated as the program grows.

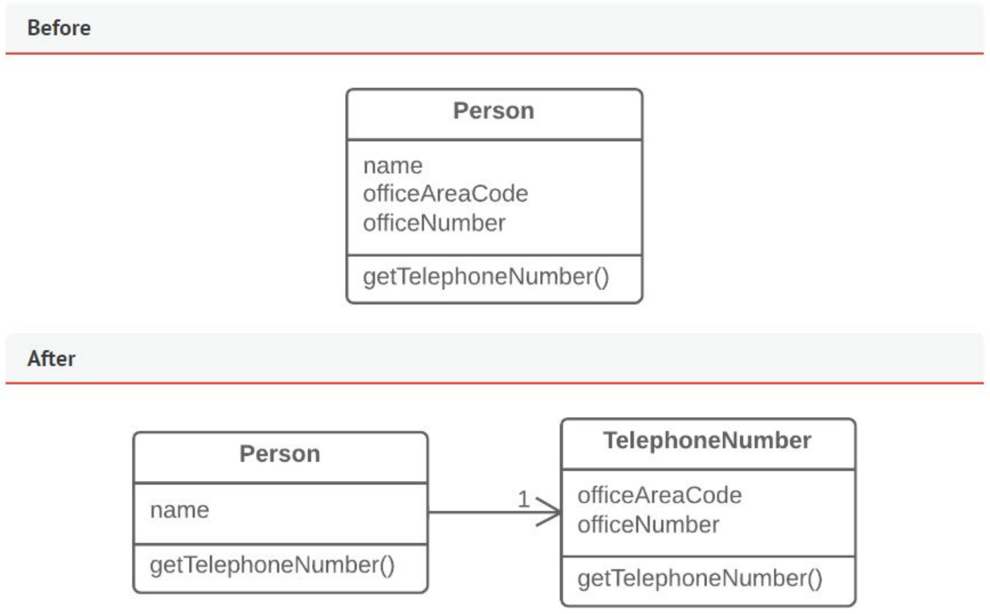
As is the case with long methods as well, programmers usually find it mentally less taxing to place a new feature in an existing class than to create a new class for the feature.



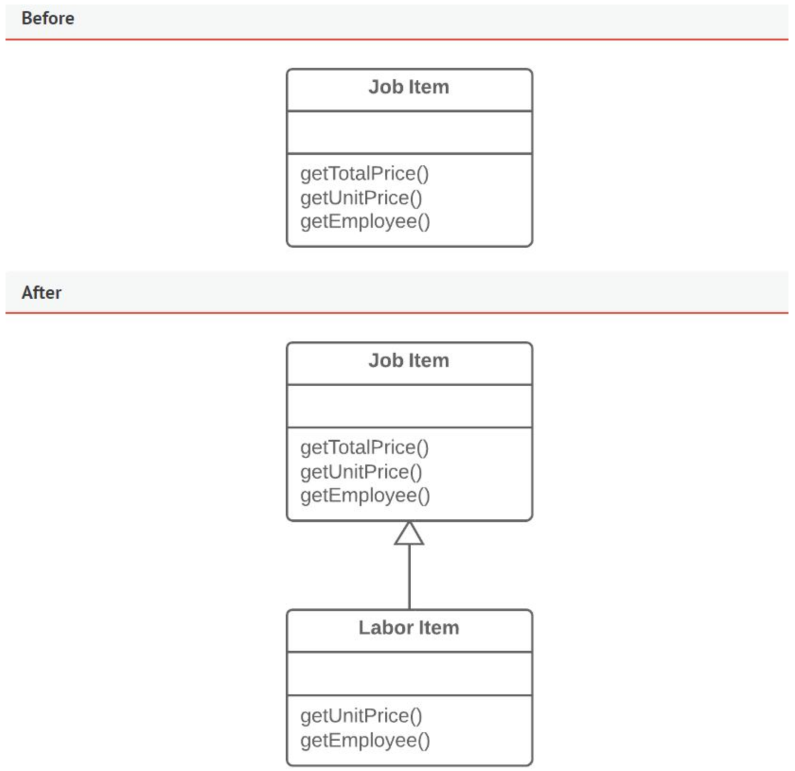
SOLUTION

When a class is wearing too many (functional) hats, think about splitting it up:

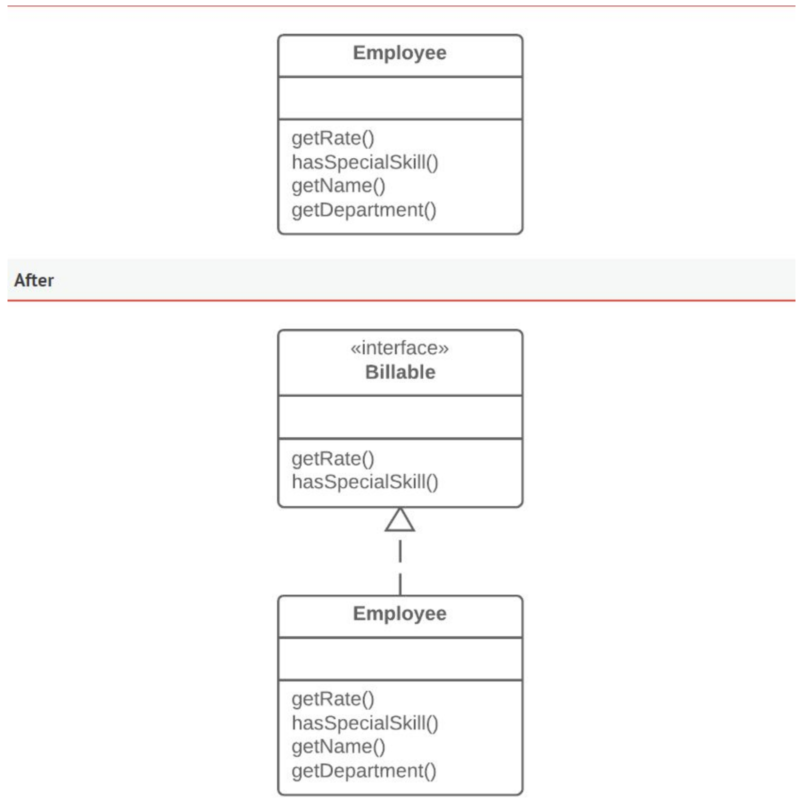
- **Extract Class** helps if part of the behavior of the large class can be spun off into a separate component.



- **Extract Subclass** helps if part of the behavior of the large class can be implemented in different ways or is used in rare cases.



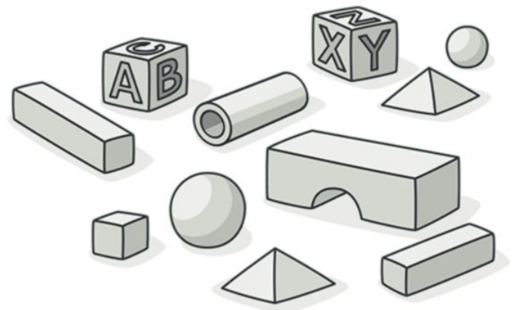
- **Extract Interface** helps if it’s necessary to have a list of the operations and behaviors that the client can use.



Payoff

- Refactoring of these classes spares developers from needing to remember a large number of attributes for a class.
- In many cases, splitting large classes into parts avoids duplication of code and functionality.

PRIMITIVE OBSESSION



Sign and Symptoms

Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)

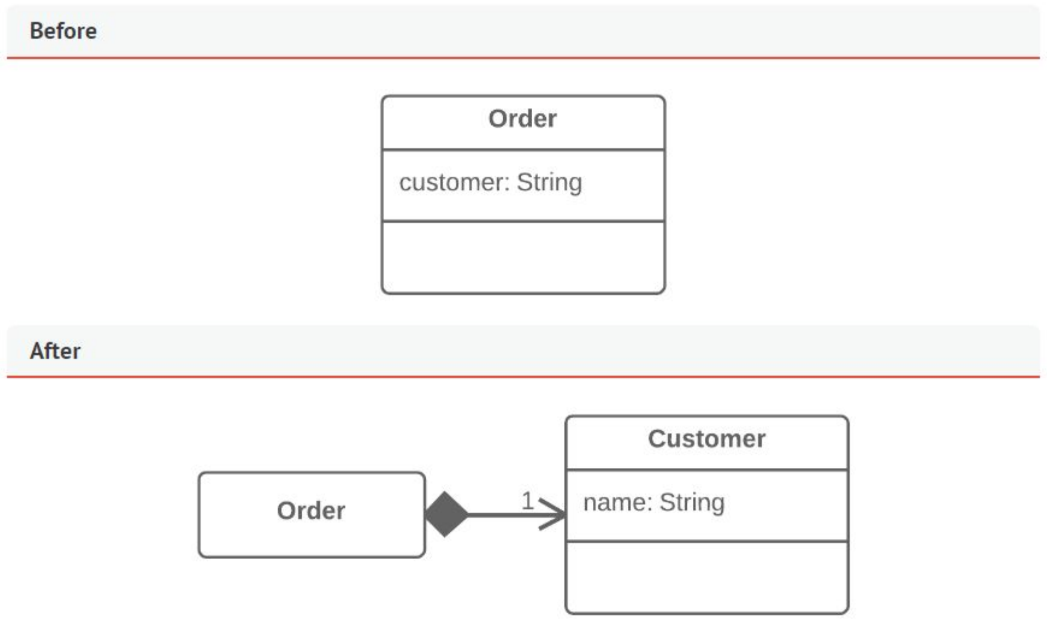
Use of constants for coding information (such as a constant `USER_ADMIN_ROLE = 1` for referring to users with administrator rights.)

Reason for the Problem

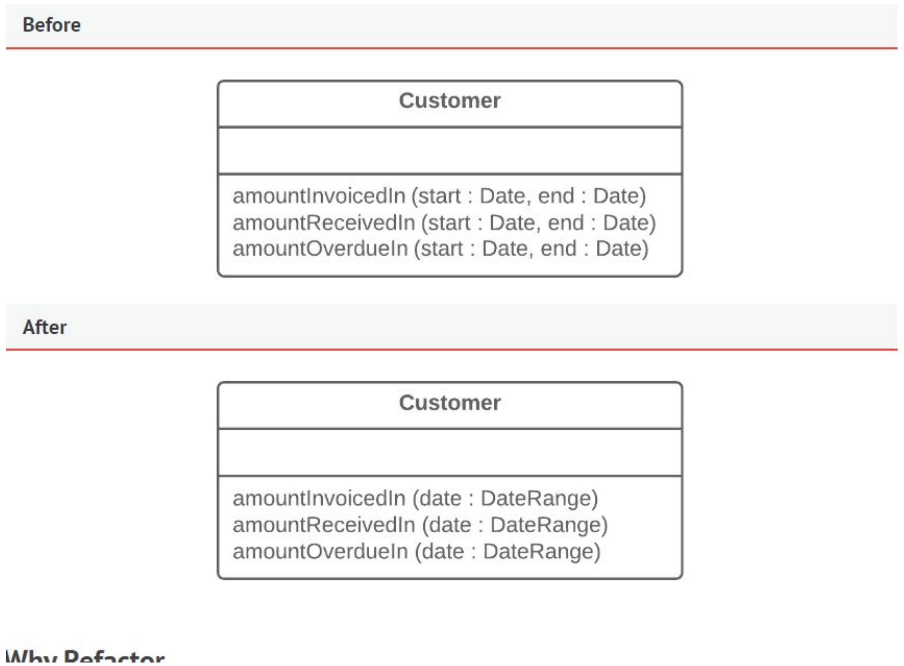
Like most other smells, primitive obsessions are born in moments of weakness. "Just a field for storing some data!" the programmer said. Creating a primitive field is so much easier than making a whole new class, right? And so it was done.

SOLUTION

- If you have a large variety of primitive fields, it may be possible to logically group some of them into their own class. Even better, move the behavior associated with this data into the class too. For this task, try **Replace Data Value with Object**.



- If the values of primitive fields are used in method parameters, go with **Introduce Parameter Object** or **Preserve Whole Object**.

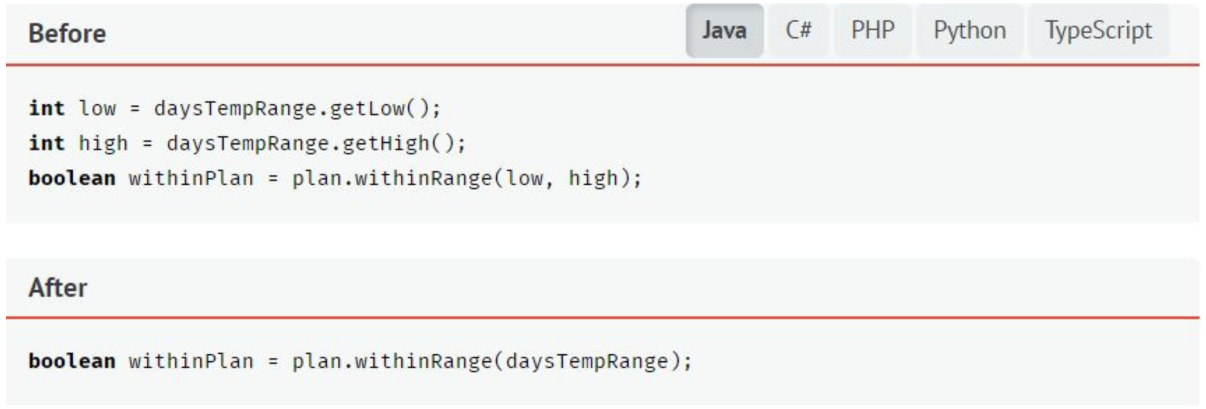

Before

```
Customer
---
amountInvoicedIn (start : Date, end : Date)
amountReceivedIn (start : Date, end : Date)
amountOverdueIn (start : Date, end : Date)
```

After

```
Customer
---
amountInvoicedIn (date : DateRange)
amountReceivedIn (date : DateRange)
amountOverdueIn (date : DateRange)
```

Why Refactor



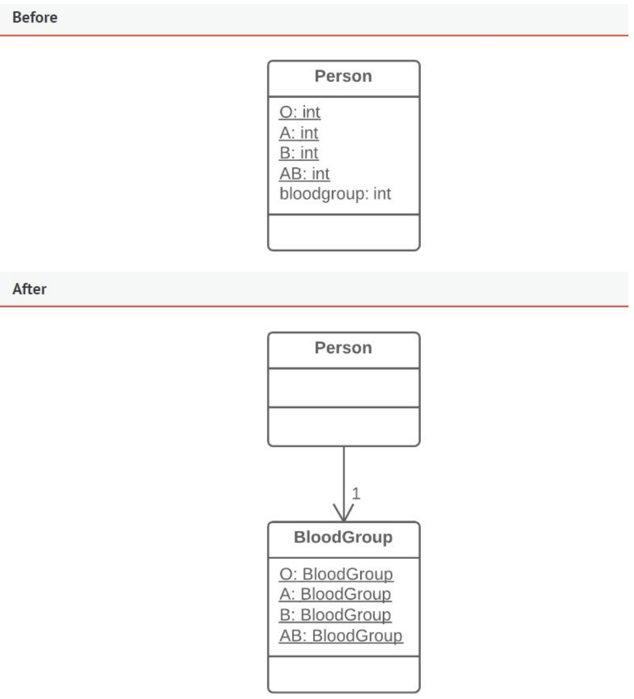
Before

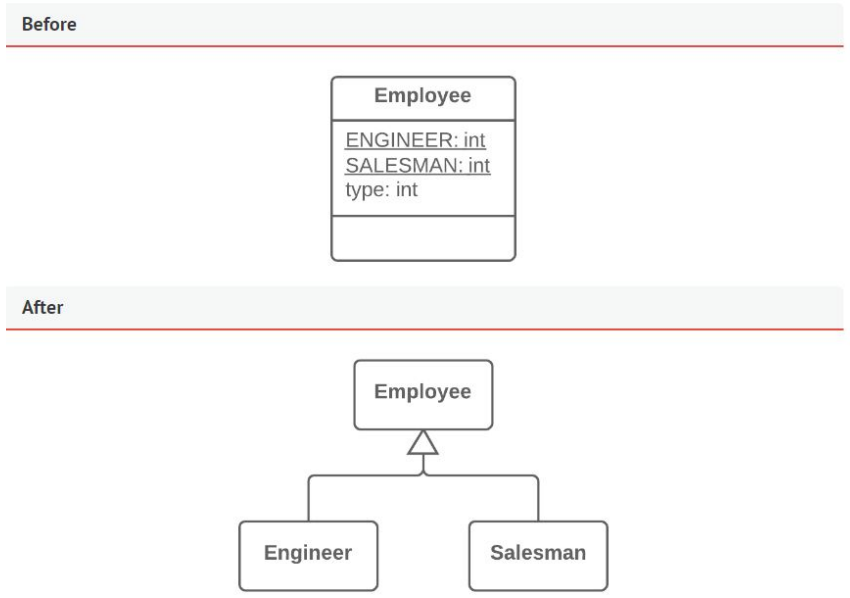
```java
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

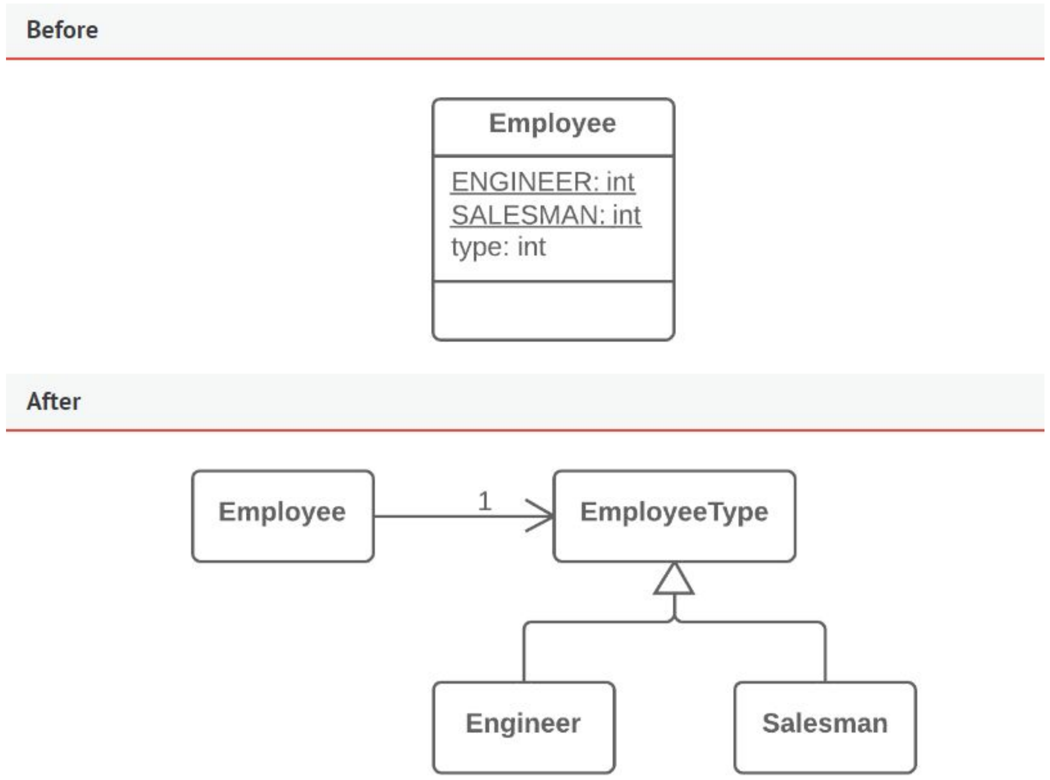
After

```java
boolean withinPlan = plan.withinRange(daysTempRange);
```

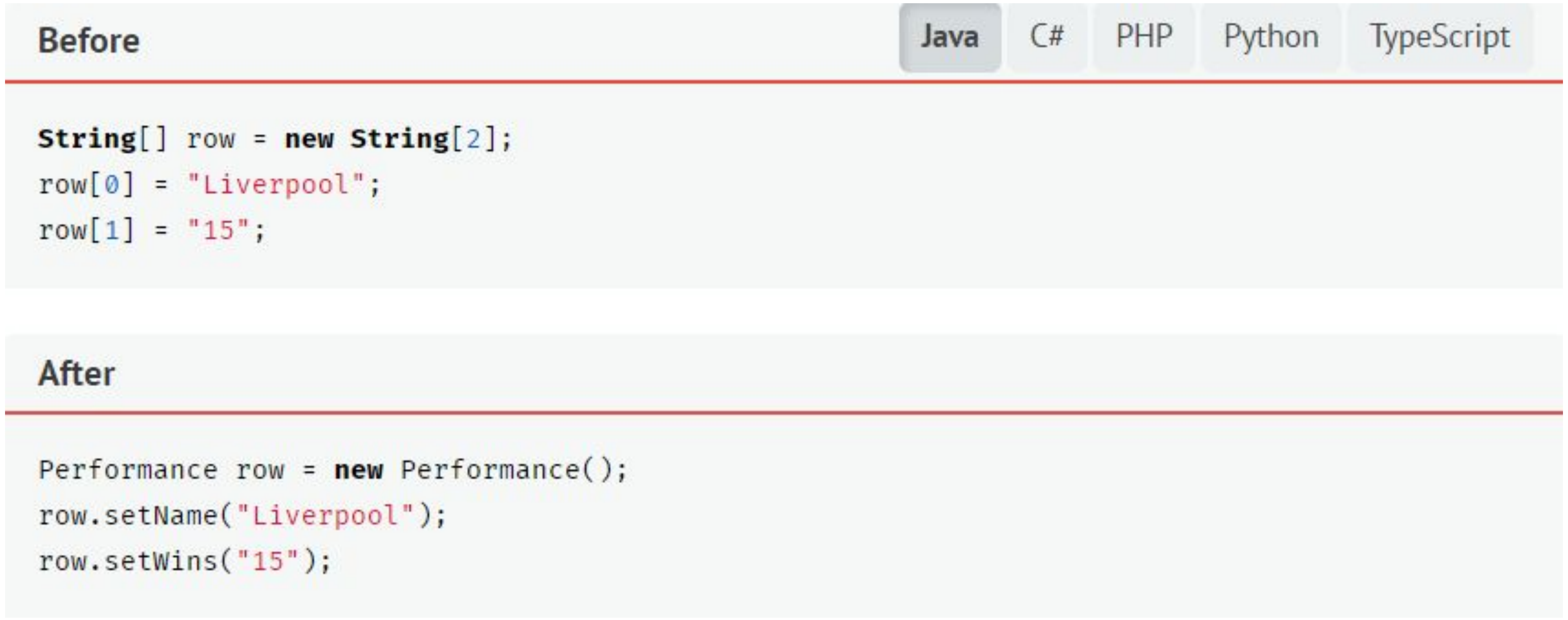
When complicated data is coded in variables, use **Replace Type Code with Class**, **Replace Type Code with Subclasses** or **Replace Type Code with State/Strategy**.







- If there are arrays among the variables, use **Replace Array with Object**.



Before

```
String[] row = new String[2];
row[0] = "Liverpool";
row[1] = "15";
```

After

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

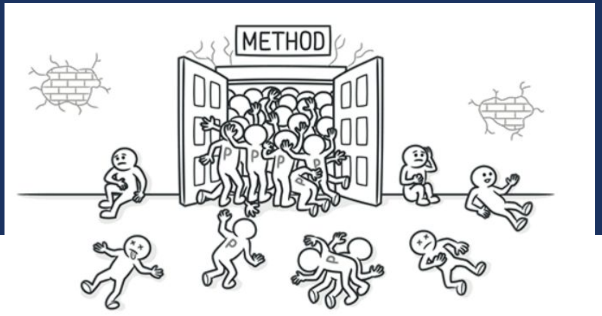
Payoff

Code becomes more flexible thanks to use of objects instead of primitives.

Better understandability and organization of code. Operations on particular data are in the same place, instead of being scattered. No more guessing about the reason for all these strange constants and why they're in an array.

Easier finding of duplicate code.

LONG PARAMETER LIST



Signs and Symptoms

More than three or four parameters for a method.

Reason for the Problem

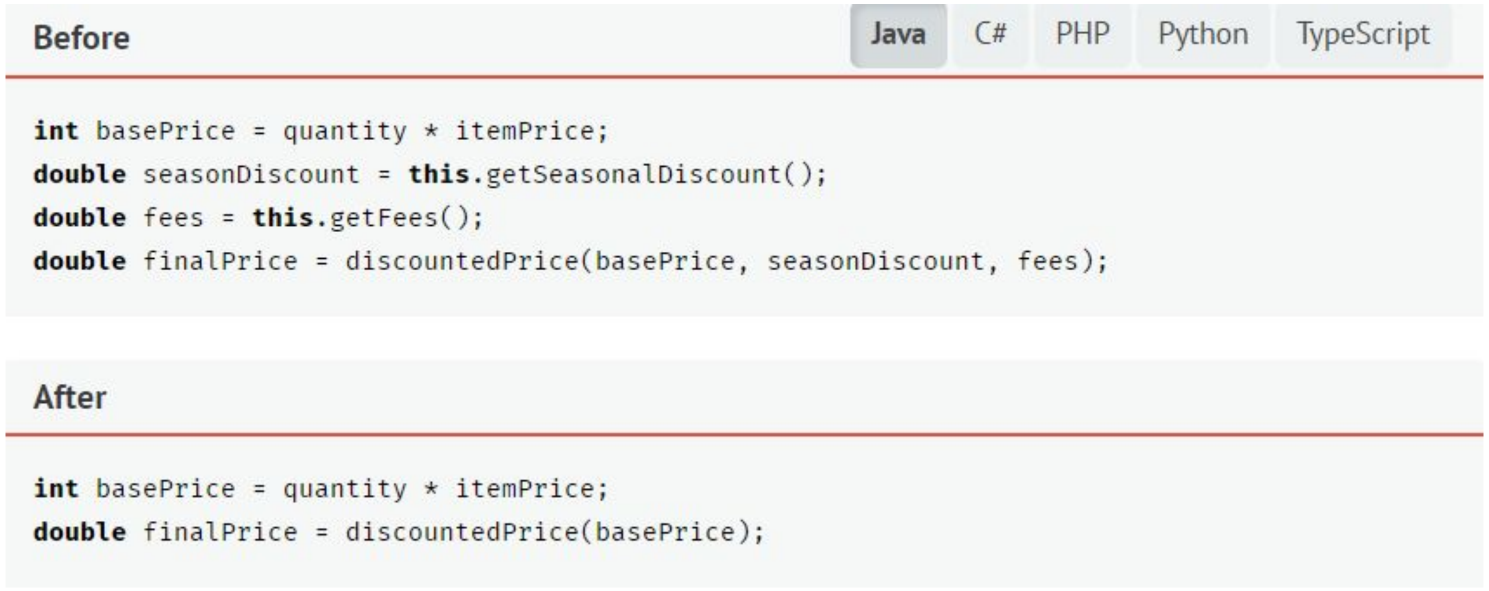
A long list of parameters might happen after several types of algorithms are merged in a single method. A long list may have been created to control which algorithm will be run and how.

It's hard to understand such lists, which become contradictory and hard to use as they grow longer.

Instead of a long list of parameters, a method can use the data of its own object. If the current object doesn't contain all necessary data, another object (which will get the necessary data) can be passed as a method parameter.

SOLUTION

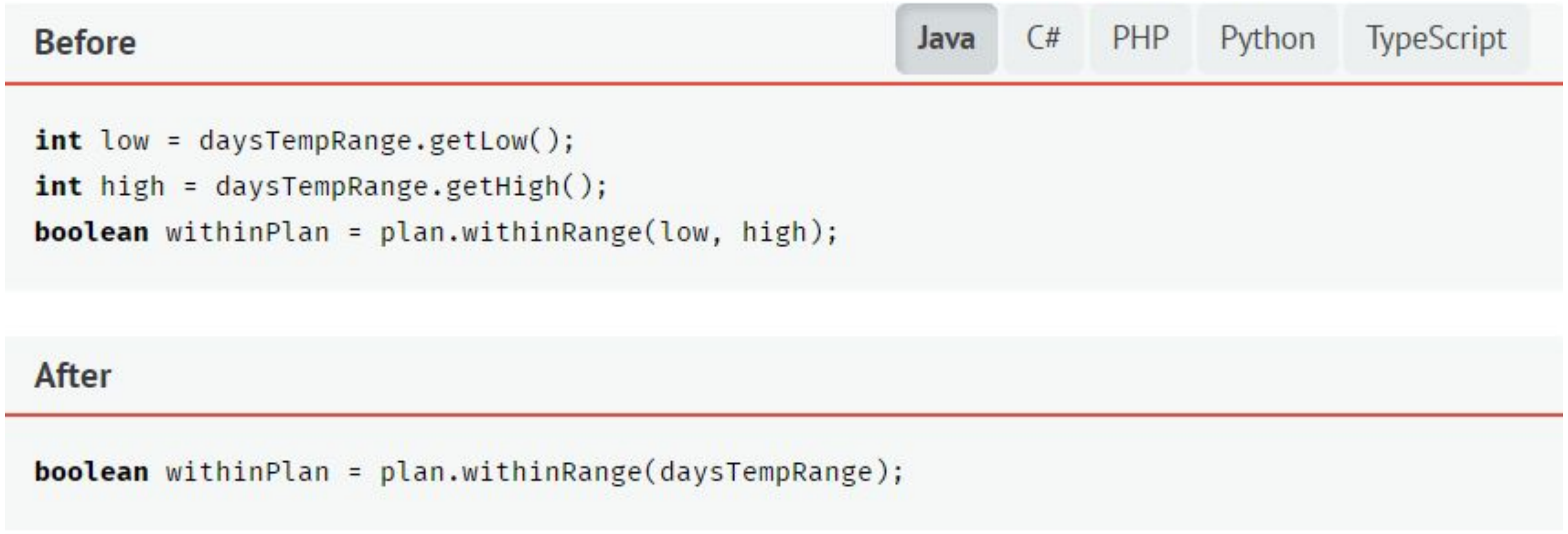
- Check what values are passed to parameters. If some of the arguments are just results of method calls of another object, use **Replace Parameter with Method Call**. This object can be placed in the field of its own class or passed as a method parameter.



```java
int basePrice = quantity * itemPrice;
double seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);
```

```java
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

- Instead of passing a group of data received from another object as parameters, pass the object itself to the method, by using **Preserve Whole Object**.



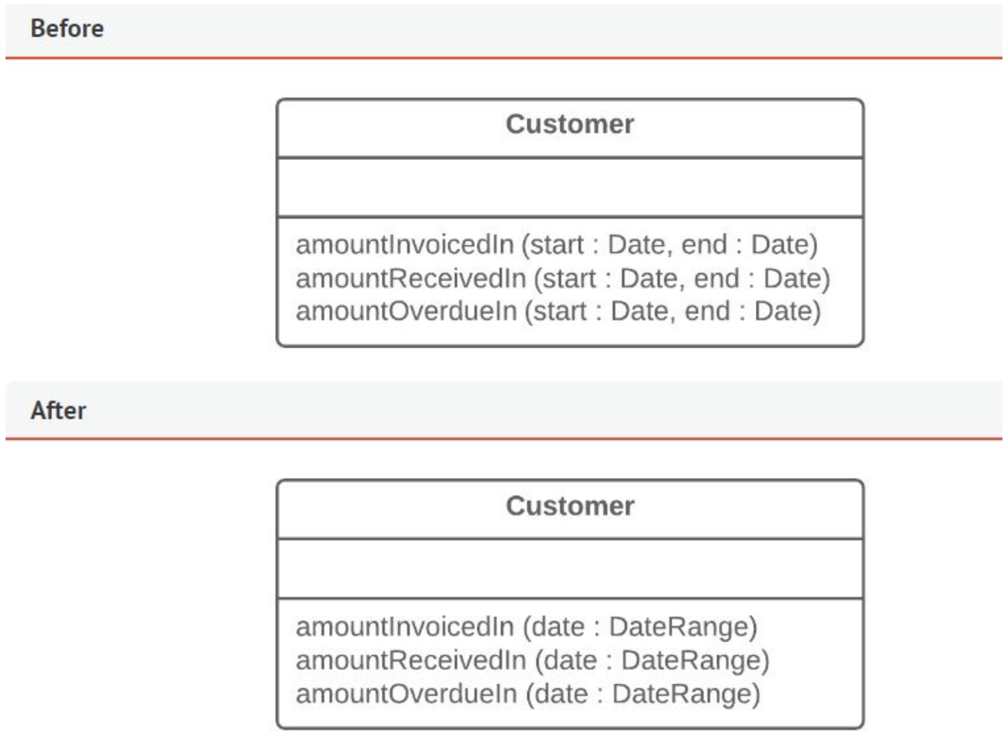
Before

```java
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

After

```java
boolean withinPlan = plan.withinRange(daysTempRange);
```

- If there are several unrelated data elements, sometimes you can merge them into a single parameter object via **Introduce Parameter Object**.

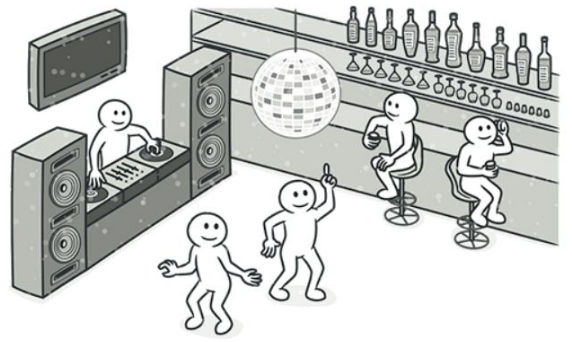

Before
Customer
amountInvoicedIn (start : Date, end : Date)
amountReceivedIn (start : Date, end : Date)
amountOverdueIn (start : Date, end : Date)
After
Customer
amountInvoicedIn (date : DateRange)
amountReceivedIn (date : DateRange)
amountOverdueIn (date : DateRange)

Payoff

More readable, shorter code.

Refactoring may reveal previously unnoticed duplicate code.

DATA CLUMPS



Sign and Symptoms

Sometimes different parts of the code contain identical groups of variables (such as parameters for connecting to a database). These clumps should be turned into their own classes.

Reason for the Problem

Often these data groups are due to poor program structure or "copypaste programming”.

If you want to make sure whether or not some data is a data clump, just delete one of the data values and see whether the other values still make sense. If this isn’t the case, this is a good sign that this group of variables should be combined into an object.

SOLUTION

- If repeating data comprises the fields of a class, use **Extract Class** to move the fields to their own class.
- If the same data clumps are passed in the parameters of methods, use **Introduce Parameter Object** to set them off as a class.
- If some of the data is passed to other methods, think about passing the entire data object to the method instead of just individual fields. **Preserve Whole Object** will help with this.

Payoff

Improves understanding and organization of code. Operations on particular data are now gathered in a single place, instead of haphazardly throughout the code.

Reduces code size.

CHANDRASEHAR R

(20PW06)

OBJECT ORIENTATION ABUSERS

All these smells are incomplete or incorrect application of object-oriented programming principles.

"Object orientation abusers" is a term that is sometimes used to describe programmers who overuse or misuse object-oriented programming (OOP) concepts in their code.

These programmers may create overly complex class hierarchies or use inheritance in inappropriate ways, resulting in code that is difficult to understand, maintain, and extend.

SWITCH STATEMENTS

Signs and Symptoms

You have a complex `switch` operator or sequence of `if` statements.

Reasons for the Problem

Relatively rare use of `switch` and `case` operators is one of the hallmarks of object-oriented code. When a new condition is added, you have to find all the `switch` code and modify it.

As a rule of thumb, when you see `switch` you should think of polymorphism.

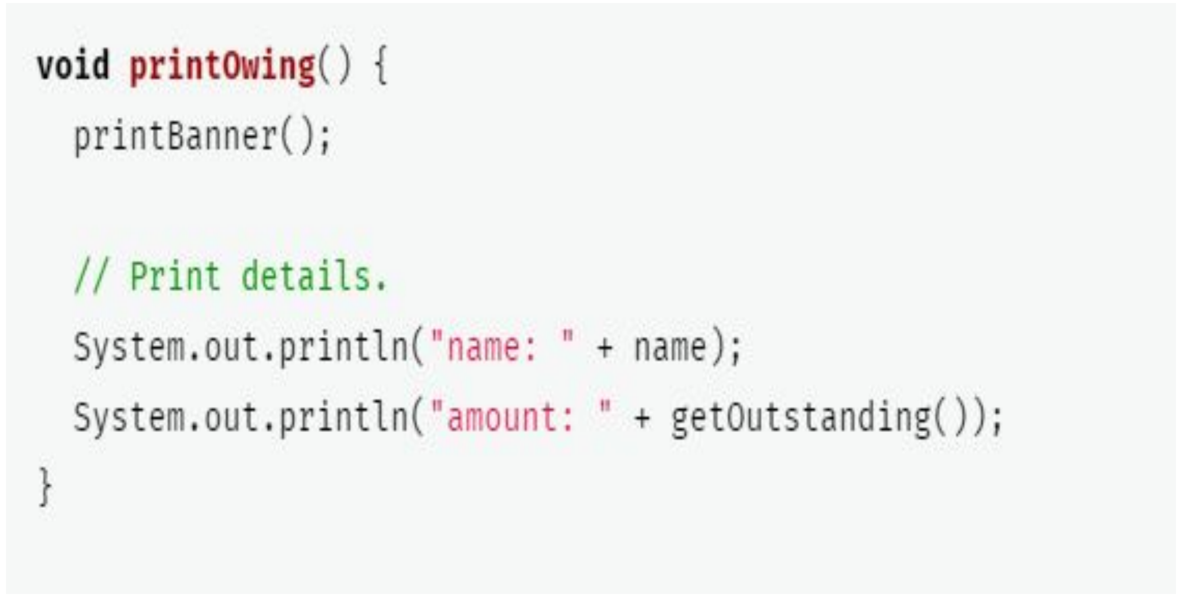
EXTRACT METHODS

Problem

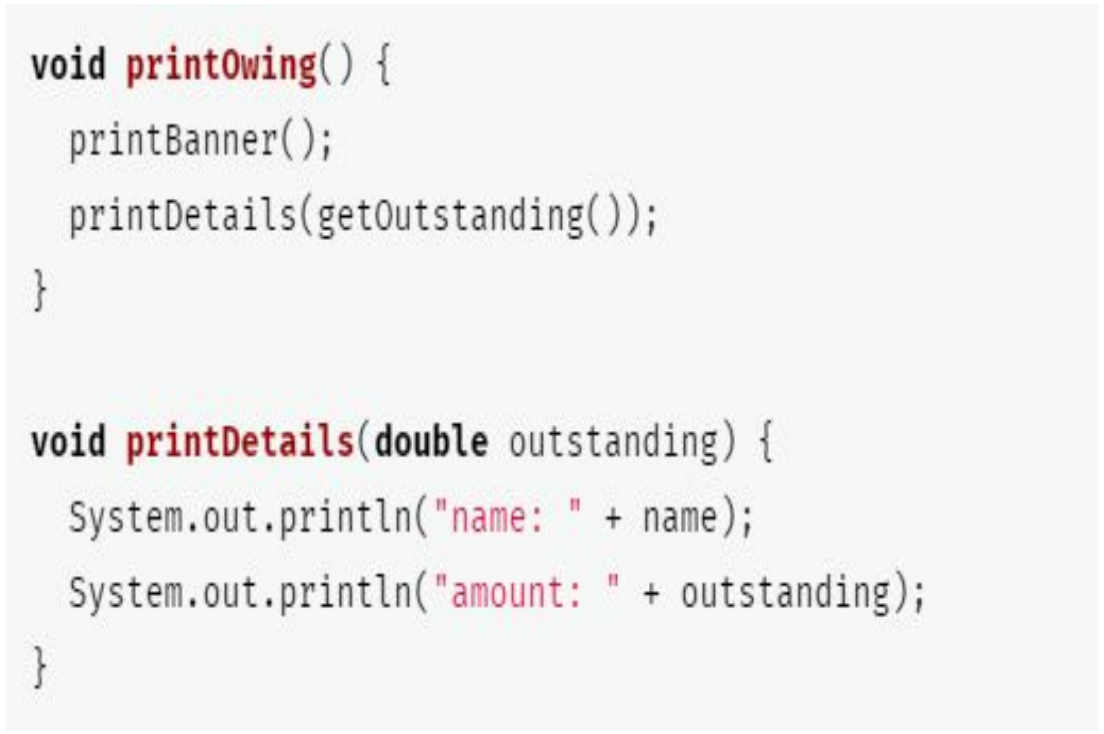
You have a code fragment that can be grouped together.

Solution

Move this code to a separate new method (or function) and replace the old code with a call to the method.



```java
void printOwing() {
  printBanner();

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



```java
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

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

Why Refactor

The more lines found in a method, the harder it's to figure out what the method does. This is the main reason for this refactoring.

Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches.

Benefits

- More readable code! Be sure to give the new method a name that describes the method's purpose: `createOrder()`, `renderCustomerInfo()`, etc.
- Less code duplication. Often the code that's found in a method can be reused in other places in your program. So you can replace duplicates with calls to your new method.

MOVE METHOD

Problem

A method is used more in another class than in its own class.

Solution

Create a new method in the class that uses the method the most, then move code from the old method to there. Turn the code of the original method into a reference to the new method in the other class or else remove it entirely.

Why Refactor

1. You want to move a method to a class that contains most of the data used by the method. This makes classes more internally coherent.
2. You want to move a method in order to reduce or eliminate the dependency of the class calling the method on the class in which it's located. This reduces dependency between classes.

REPLACE CONDITIONAL WITH POLYMORPHISM

Problem

You have a conditional that performs various actions depending on object type or properties.

Solution

Create subclasses matching the branches of the conditional. In them, create a shared method and move code from the corresponding branch of the conditional to it. Then replace the conditional with the relevant method call. The result is that the proper implementation will be attained via polymorphism depending on the object class.

Benefits

- If you need to add a new execution variant, all you need to do is add a new subclass without touching the existing code (*Open/Closed Principle*).

```
class Bird {
  // ...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCoco
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

```
abstract class Bird {
  // ...
  abstract double getSpeed();
}

class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts
  }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}

// Somewhere in client code
speed = bird.getSpeed();
```

INTRODUCE NULL OBJECT

Problem

Since some methods return `null` instead of real objects, you have many checks for `null` in your code.

Solution

Instead of `null`, return a null object that exhibits the default behavior.

Why Refactor

Dozens of checks for `null` make your code longer and uglier.

Drawbacks

The price of getting rid of conditionals is creating yet another new class.

```java
if (customer == null) {
  plan = BillingPlan.basic();
}
else {
  plan = customer.getPlan();
}
```

```java
class NullCustomer extends Customer {
  boolean isNull() {
    return true;
  }
  Plan getPlan() {
    return new NullPlan();
  }
  // Some other NULL functionality.
}

// Replace null values with Null-object.
customer = (order.customer != null) ?
  order.customer : new NullCustomer();

// Use Null-object as if it's normal subclass.
plan = customer.getPlan();
```

REPLACE PARAMETER WITH EXPLICIT METHODS

Problem

A method is split into parts, each of which is run depending on the value of a parameter.

Solution

Extract the individual parts of the method into their own methods and call them instead of the original method.

```java
void setValue(String name, int value) {
  if (name.equals("height")) {
    height = value;
    return;
  }
  if (name.equals("width")) {
    width = value;
    return;
  }
  Assert.shouldNeverReachHere();
}
```

```java
void setHeight(int arg) {
  height = arg;
}
void setWidth(int arg) {
  width = arg;
}
```

CONT OF SWITCH STATEMENTS

Treatment

- To isolate `switch` and put it in the right class, you may need **Extract Method** and then **Move Method**.
- If a `switch` is based on type code, such as when the program's runtime mode is switched, use **Replace Type Code with Subclasses** or **Replace Type Code with State/Strategy**.
- After specifying the inheritance structure, use **Replace Conditional with Polymorphism**.
- If there aren't too many conditions in the operator and they all call same method with different parameters, polymorphism will be superfluous. If this case, you can break that method into multiple smaller methods with **Replace Parameter with Explicit Methods** and change the `switch` accordingly.
- If one of the conditional options is `null`, use **Introduce Null Object**.

Payoff

- Improved code organization.

When to Ignore

- When a `switch` operator performs simple actions, there’s no reason to make code changes.

TEMPORARY FIELD

Signs and Symptoms

Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they're empty.

Reasons for the Problem

Oftentimes, temporary fields are created for use in an algorithm that requires a large amount of inputs. So instead of creating a large number of parameters in the method, the programmer decides to create fields for this data in the class. These fields are used only in the algorithm and go unused the rest of the time.

This kind of code is tough to understand. You expect to see data in object fields but for some reason they're almost always empty.

Treatment

- Temporary fields and all code operating on them can be put in a separate class via **Extract Class**. In other words, you're creating a method object, achieving the same result as if you would perform **Replace Method with Method Object**.
- **Introduce Null Object** and integrate it in place of the conditional code which was used to check the temporary field values for existence.

Payoff

- Better code clarity and organization.

Basic Examples of Temporary Fields

```
function nameToObject (name) {
   var fullName = name.split(' ');
   var firstName = fullName[0];
   var lastName = lastName[1];

   var name = {
      firstName: firstName,
      lastName: lastName
   };

   return name;
}
```

```
function sum (a, b) {
  var total = a + b;
  return total;


}
```

Example Solution

```
function nameToObject (name) {
   var fullName = name.split(' ');


   return {
      firstName: fullName[0],
      lastName: fullName[1]
   };


}
```

```
function sum (a, b) {
  return a + b;


}
```

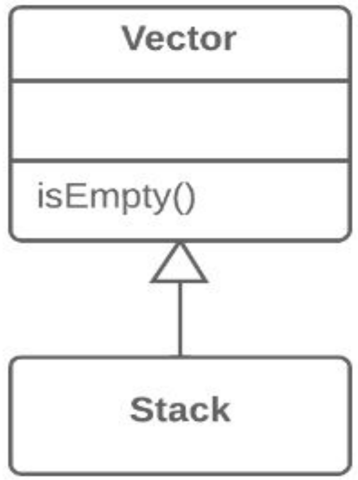
REPLACE INHERITANCE WITH DELEGATION

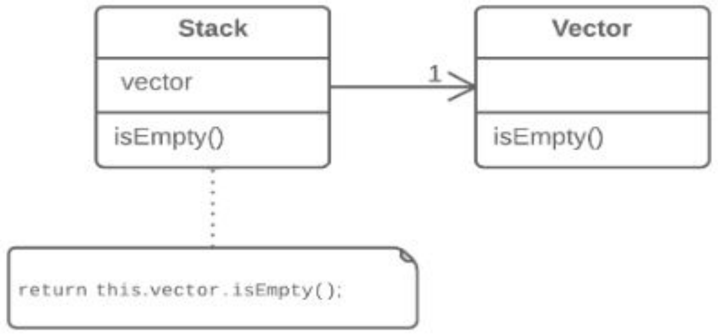
Problem

You have a subclass that uses only a portion of the methods of its superclass (or it’s not possible to inherit superclass data).

Solution

Create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance.





Why Refactor

Replacing inheritance with composition can substantially improve class design if:

- Your subclass violates the *Liskov substitution principle*, i.e., if inheritance was implemented only to combine common code but not because the subclass is an extension of the superclass.
- The subclass uses only a portion of the methods of the superclass. In this case, it's only a matter of time before someone calls a superclass method that he or she wasn't supposed to call.

In essence, this refactoring technique splits both classes and makes the superclass the helper of the subclass, not its parent. Instead of inheriting all superclass methods, the subclass will have only the necessary methods for delegating to the methods of the superclass object.

Benefits

- A class doesn't contain any unneeded methods inherited from the superclass.
- Various objects with various implementations can be put in the delegate field. In effect you get the **Strategy** design pattern.

Drawbacks

- You have to write many simple delegating methods.

How to Refactor

1. Create a field in the subclass for holding the superclass. During the initial stage, place the current object in it.
2. Change the subclass methods so that they use the superclass object instead of `this`.
3. For methods inherited from the superclass that are called in the client code, create simple delegating methods in the subclass.
4. Remove the inheritance declaration from the subclass.
5. Change the initialization code of the field in which the former superclass is stored by creating a new object.

REFUSED BEQUEST

Signs and Symptoms

If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be redefined and give off exceptions.

Reasons for the Problem

Someone was motivated to create inheritance between classes only by the desire to reuse the code in a superclass. But the superclass and subclass are completely different.

Treatment

- If inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance in favor of **Replace Inheritance with Delegation**.
- If inheritance is appropriate, get rid of unneeded fields and methods in the subclass. Extract all fields and methods needed by the subclass from the parent class, put them in a new superclass, and set both classes to inherit from it (**Extract Superclass**).

Payoff

- Improves code clarity and organization.

RENAME METHOD

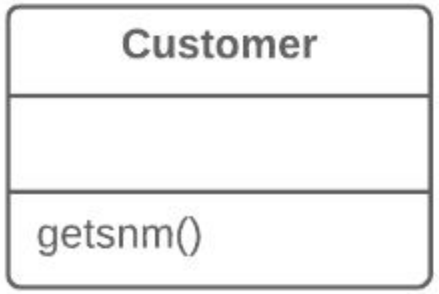
Problem

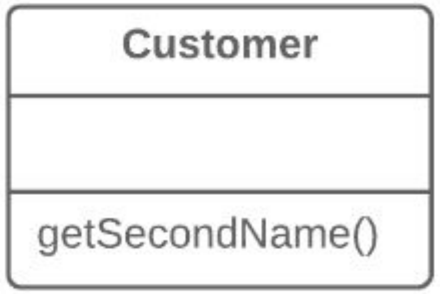
The name of a method doesn't explain what the method does.

Why Refactor

Perhaps a method was poorly named from the very beginning—for example, someone created the method in a rush and didn't give proper care to naming it well.

Or perhaps the method was well named at first but as its functionality grew, the method name stopped being a good descriptor.





ADD PARAMETER

Problem

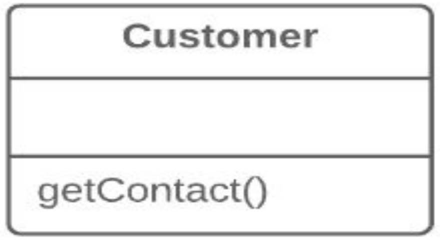
A method doesn’t have enough data to perform certain actions.

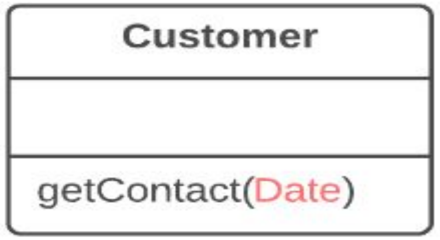
Solution

Create a new parameter to pass the necessary data.

Why Refactor

You need to make changes to a method and these changes require adding information or data that was previously not available to the method.





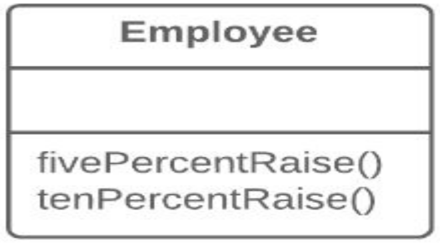
PARAMETERIZE METHOD

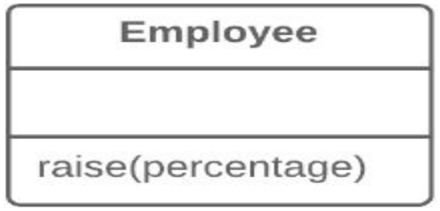
Problem

Multiple methods perform similar actions that are different only in their internal values, numbers or operations.

Solution

Combine these methods by using a parameter that will pass the necessary special value.





EXTRACT SUPERCLASS

Problem

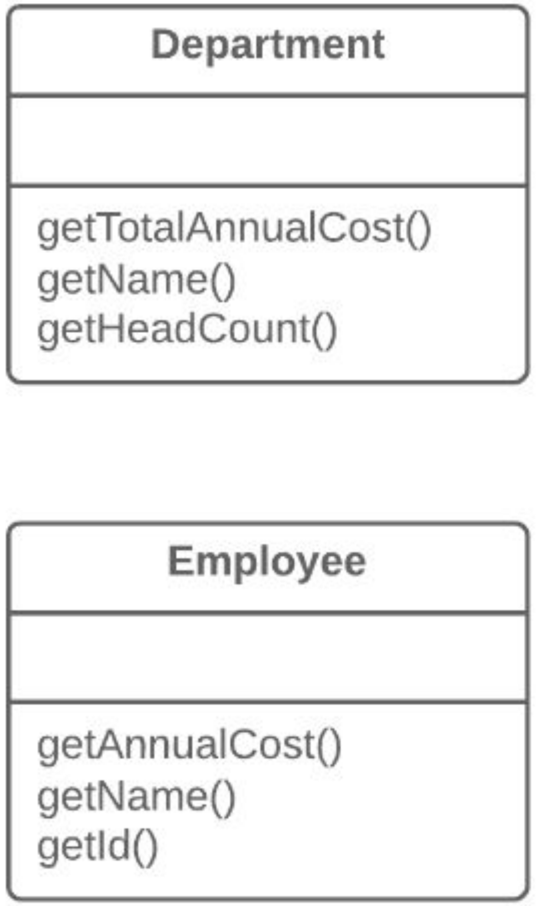
You have two classes with common fields and methods.

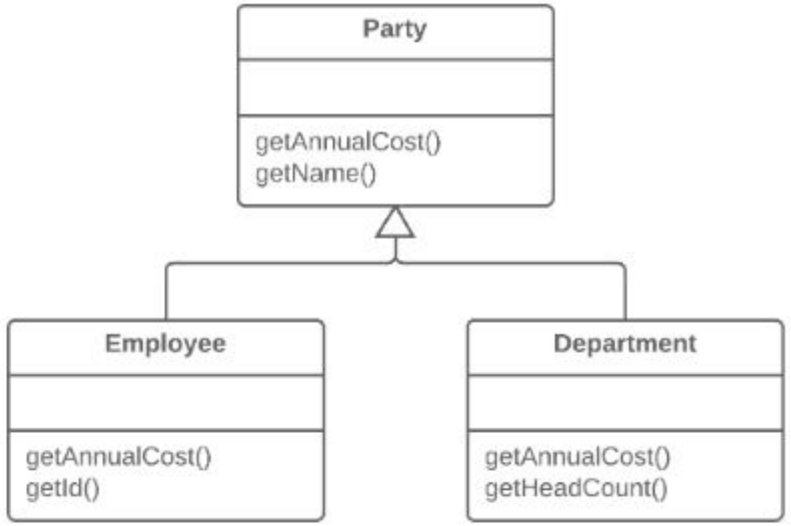
Solution

Create a shared superclass for them and move all the identical fields and methods to it.

Why Refactor

One type of code duplication occurs when two classes perform similar tasks in the same way, or perform similar tasks in different ways.


Department
getTotalAnnualCost()
getName()
getHeadCount()
Employee
getAnnualCost()
getName()
getId()


Party
getAnnualCost()
getName()
Employee
getAnnualCost()
getId()
Department
getAnnualCost()
getHeadCount()

ALTERNATIVE CLASS WITH DIFFERENT INTERFACES

Signs and Symptoms

Two classes perform identical functions but have different method names.

Reasons for the Problem

The programmer who created one of the classes probably didn’t know that a functionally equivalent class already existed.

Try to put the interface of classes in terms of a common denominator:

- **Rename Method**s to make them identical in all alternative classes.
- **Move Method**, **Add Parameter** and **Parameterize Method** to make the signature and implementation of methods the same.
- If only part of the functionality of the classes is duplicated, try using **Extract Superclass**. In this case, the existing classes will become subclasses.
- After you have determined which treatment method to use and implemented it, you may be able to delete one of the classes.

DHIVYA LAKSHMI S
(20PW10)

DISPENSIBLES

WHAT ARE DISPENSABLES?

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

TYPES:

->COMMENTS

->DUPLICATE CODE

->LAZY CLASS

->DATA CLASS

->DEAD CODE

->SPECULATIVE GENERALITY

1) COMMENTS

SIGNS AND SYMPTOMS:

A method is filled with too much of explanatory comments.

REASONS FOR THE PROBLEM:

Comments are usually created with the best of intentions, when the author realizes that his or her code is not intuitive or obvious. In such cases, comments are like a deodorant masking the smell of fishy code that could be improved

SOLUTION:

If you feel that a code fragment cannot be understood without comments, try to change the code structure in a way that makes comments unnecessary.

1)If a comment explains a section of code, this section can be turned into a separate method via **extract method.**

2)If a method has already been extracted, but comments are still necessary to explain what the method does, give the method a self-explanatory name.

3)If you need to assert rules about a state that is necessary for the system to work, use **assertion.**

EXTRACT METHOD

The Extract Method refactoring lets you take a code fragment that can be grouped, move it into a separated method, and replace the old code with a call to the method.

BEFORE:

```
void printOwing() {
  printBanner();


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

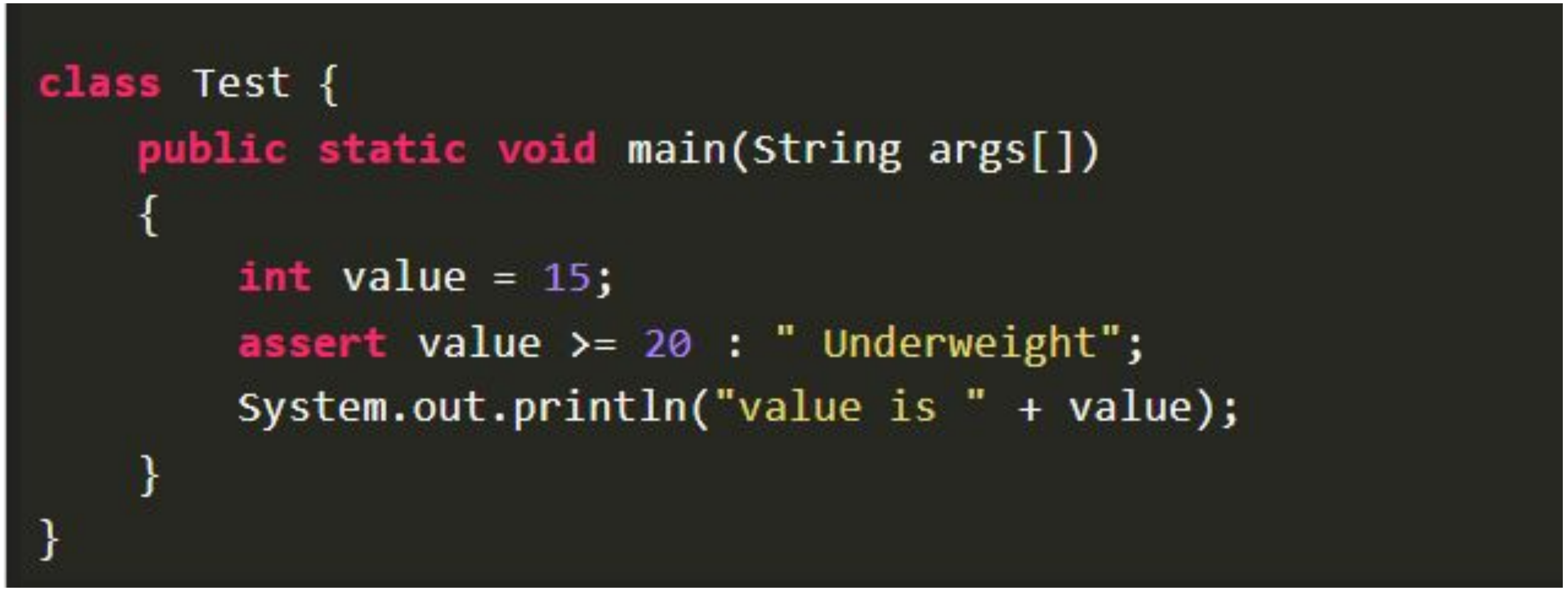
EXTRACT METHOD

AFTER:

```java
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

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

ASSERTION

An assertion allows testing the correctness of any assumptions that have been made in the program.While executing assertion, it is believed to be true. If it fails, JVM throws an error named AssertionError.



```java
class Test {
    public static void main(String args[])
    {
        int value = 15;
        assert value >= 20 : " Underweight";
        System.out.println("value is " + value);
    }
}
```

After assertion:

OUTPUT:
Exception in thread "main" java.lang.AssertionError: Underweight

2)DUPLICATE CODE

SIGNS AND SYMPTOMS:

Two code fragments look almost identical.

REASONS:

Duplication usually occurs when multiple programmers are working on different parts of the same program at the same time. Since they're working on different tasks, they may be unaware their colleague has already written similar code that could be repurposed for their own needs.

SOLUTION:

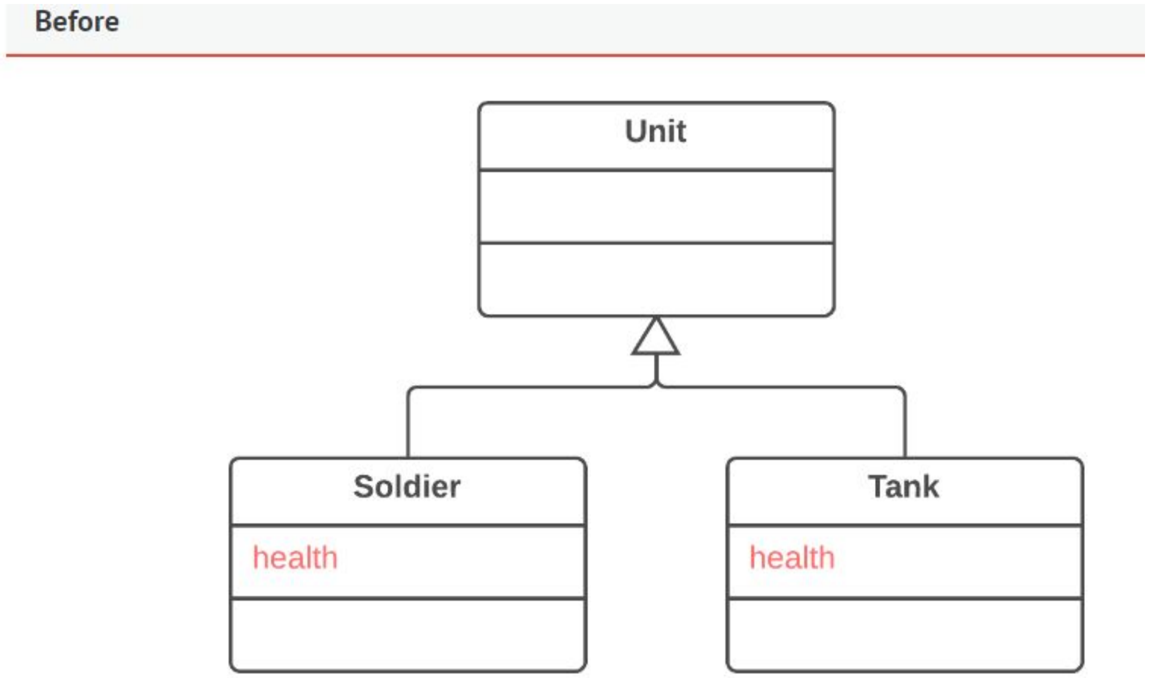
If the same code is found in two or more methods in the same class: use **extract method.**

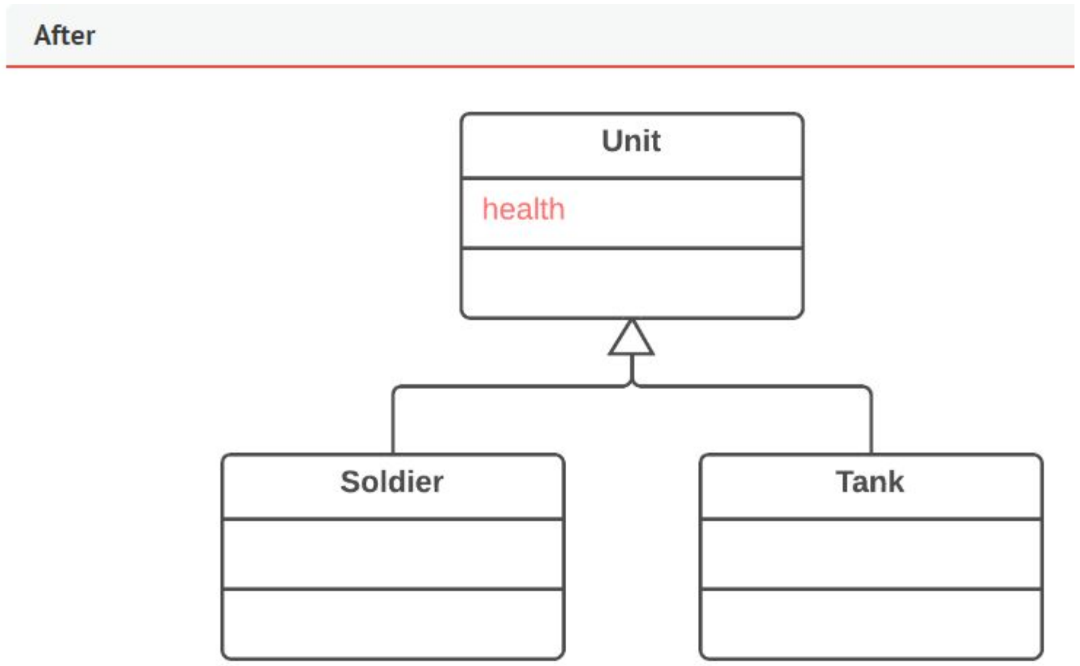
if the same code is found in two subclasses of the same level:

- If the same variables/fields are found in both the classes, use **Pull Up Field** .
- If the duplicate code is inside a constructor, use **Pull up constructor field.**

PULL-UP FIELD

Remove the field from subclasses and move it to the superclass.





PULL-UP CONSTRUCTOR

PROBLEM: Your subclasses have constructors with code that's mostly identical.
SOLUTION:Create a superclass constructor and move the code that's the same in the subclasses to it. Call the superclass constructor in the subclass constructors.

BEFORE:

```
class Manager extends Employee {
  public Manager(String name, String id, int grade) {
    this.name = name;
    this.id = id;
    this.grade = grade;
  }
  // ...
}
```

PULL-UP CONSTRUCTOR

AFTER:

```
class Manager extends Employee {
  public Manager(String name, String id, int grade) {
    super(name, id);
    this.grade = grade;
  }
  // ...
}
```

3)LAZY CLASS

SIGNS AND SYMPTOMS:

A Lazy Element is an element that does not do enough. If a method, variable, or class does not do enough to pay for itself , it should be either removed or combined into another entity.

REASONS:

Perhaps a class was designed to be fully functional but after some of the refactoring it has become ridiculously small. Or perhaps it was designed to support future development work that never got done.

SOLUTION:

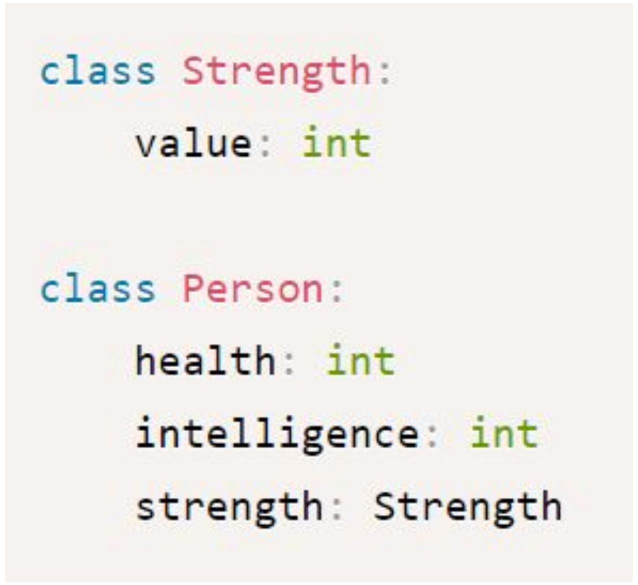
Components that are near-useless should be given the **inline class** treatment.

For subclasses with few functions, try **Collapse Hierarchy**.

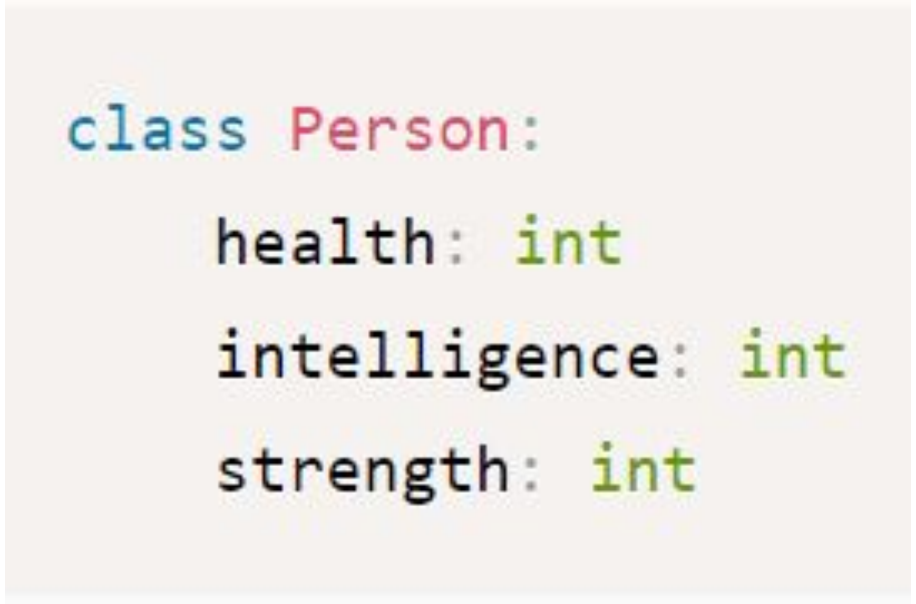
INLINE CLASS

PROBLEM:
A class does almost nothing and isn’t responsible for anything, and no additional responsibilities are planned for it.

SOLUTION:
Move all features from the class to another one.



```
class Strength:
    value: int

class Person:
    health: int
    intelligence: int
    strength: Strength
```

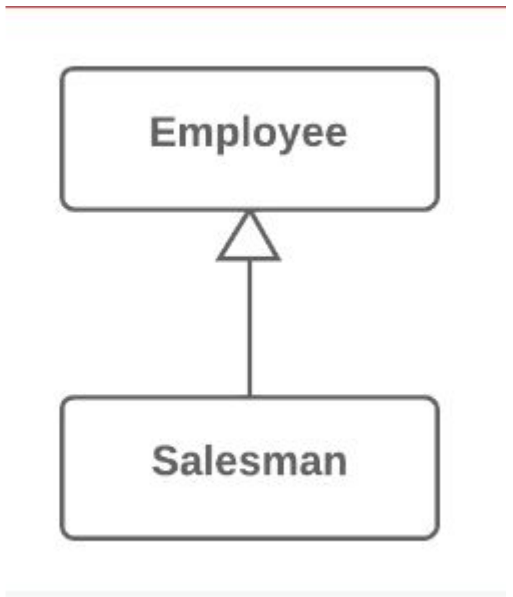


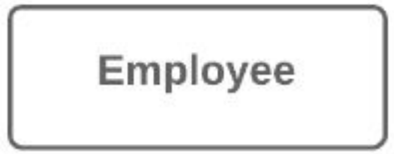
```
class Person:
    health: int
    intelligence: int
    strength: int
```

COLLAPSE HIERARCHY

PROBLEM:
You have a class hierarchy in which a subclass is practically the same as its superclass.

SOLUTION:
Merge the subclass and superclass.





DATA CLASS

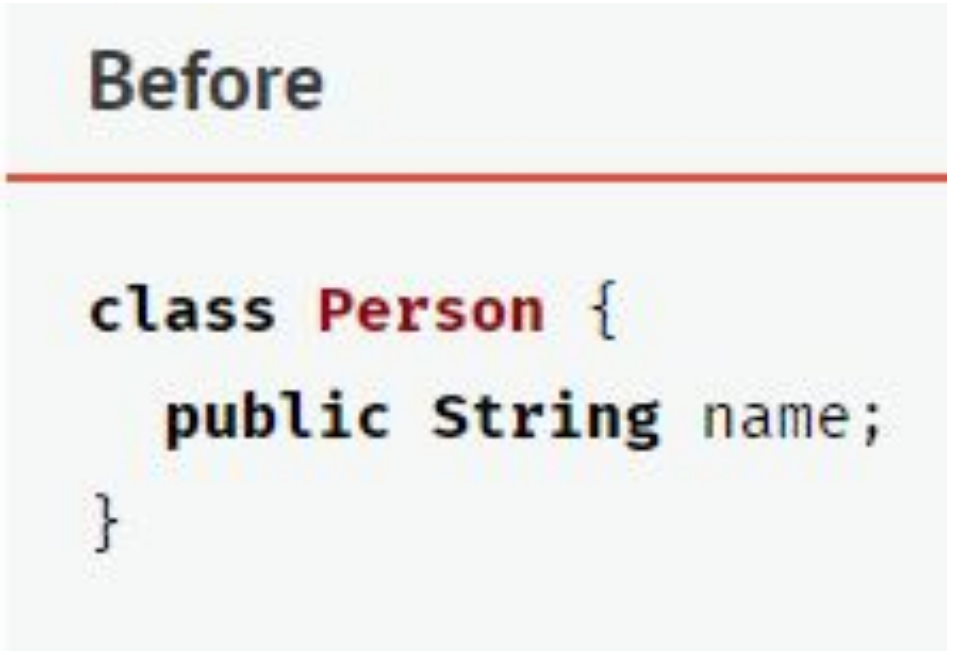
SIGNS AND SYMPTOMS:
A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes don’t contain any additional functionality and can’t independently operate on the data that they own.

SOLUTION:

- If a class contains public fields, use **Encapsulate Field** to hide them from direct access and require that access be performed via getters and setters only.
- Use **Encapsulate Collection** for data stored in collections (such as arrays).
- Review the client code that uses the class. In it, you may find functionality that would be better located in the data class itself. If this is the case, migrate this functionality to the data class.

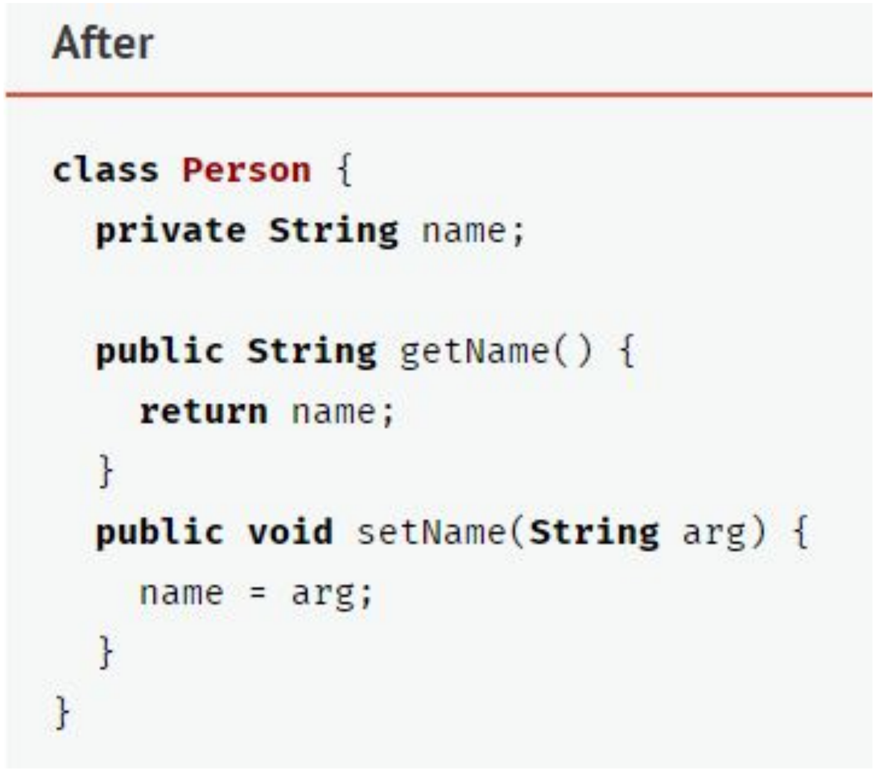
ENCAPSULATE FIELD

PROBLEM: You have a public field.
SOLUTION: Make the field private and create access methods for it.



Before

```
class Person {
  public String name;
}
```

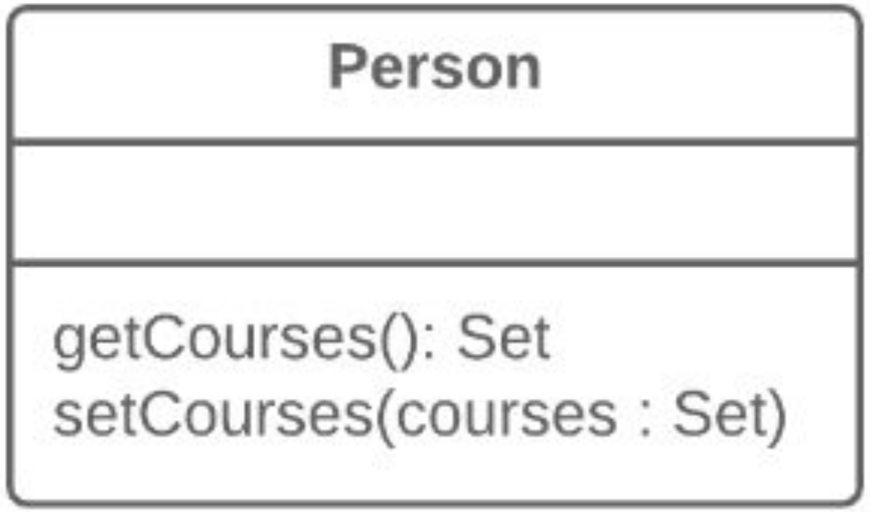


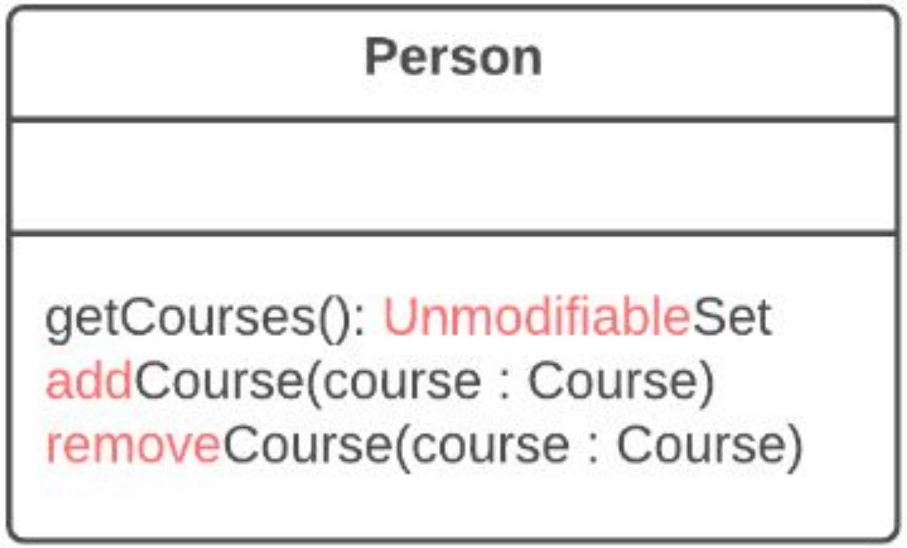
After

```
class Person {
  private String name;

  public String getName() {
    return name;
  }
  public void setName(String arg) {
    name = arg;
  }
}
```

ENCAPSULATE COLLECTION

PROBLEM: A class contains a collection field and a simple getter and setter for working with the collection.
SOLUTION: Make the getter-returned value read-only and create methods for adding/deleting elements of the collection.




Person
getCourses(): UnmodifiableSet
addCourse(course : Course)
removeCourse(course : Course)

DEAD CODE

Signs and Symptoms:

A variable, parameter, field, method or class is no longer used (usually because it's obsolete).

Reasons for the Problem

When requirements for the software have changed or corrections have been made, nobody had time to clean up the old code.

SOLUTION:

The quickest way to find dead code is to use a good **IDE**.

- Delete unused code and unneeded files.
- In the case of an unnecessary class, **Inline Class** or **Collapse Hierarchy** can be applied if a subclass or superclass is used.

SPECULATIVE GENERALITY

SIGNS AND SYMPTOMS:

There's an unused class, method, field or parameter.

REASONS FOR THE PROBLEM:

Sometimes code is created "just in case" to support anticipated future features that never get implemented. As a result, code becomes hard to understand and support.

TREATMENT:

For removing unused abstract classes, try **Collapse Hierarchy.**

Unused methods? Use **Inline method** to get rid of them**.**

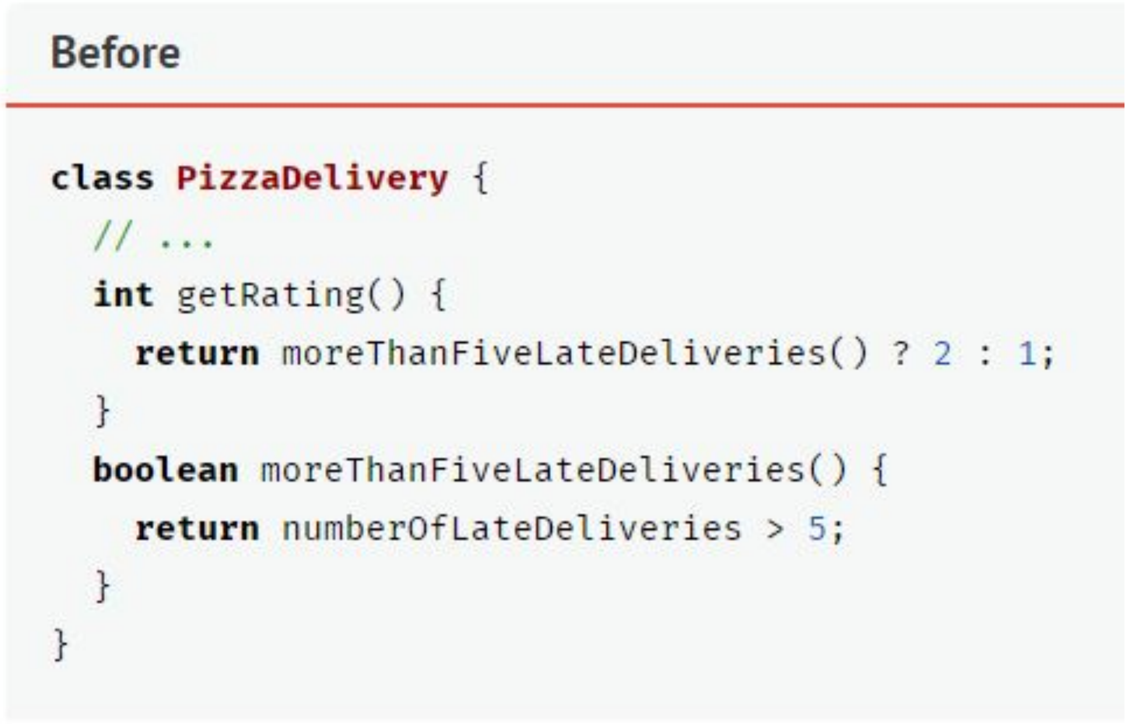
INLINE METHOD

Problem

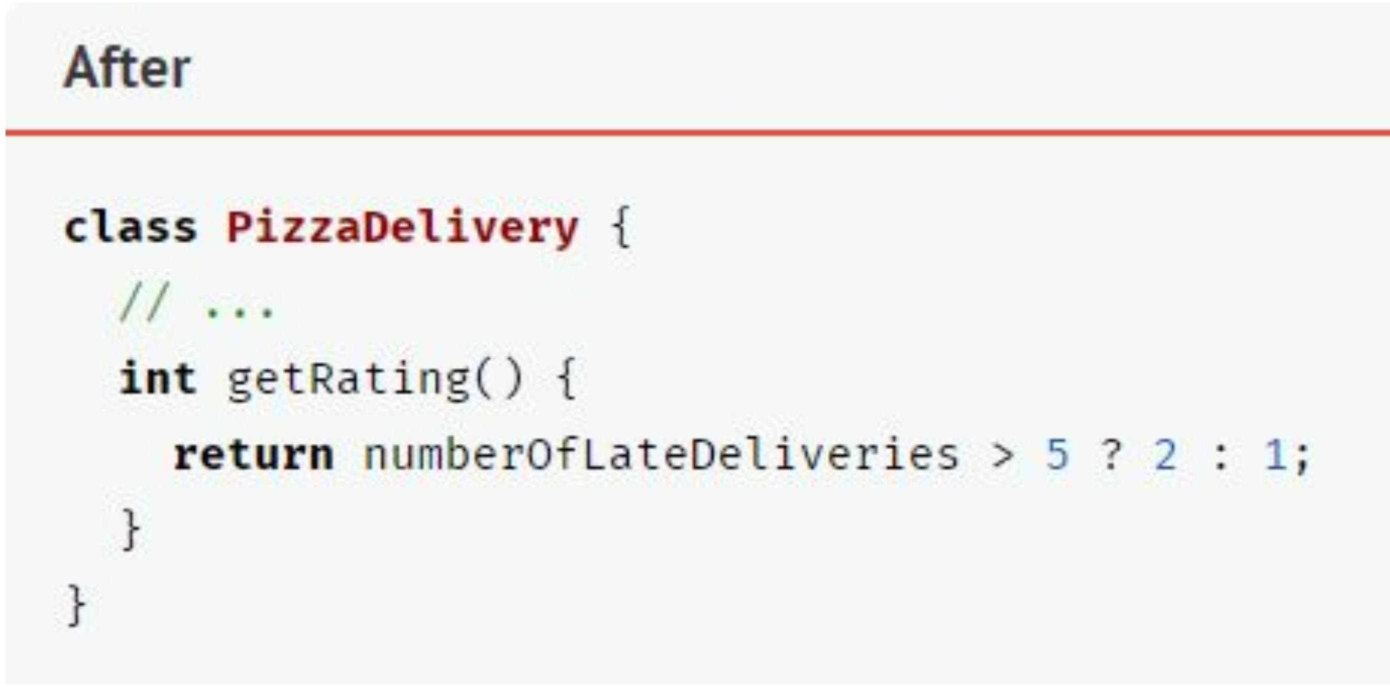
When a method body is more obvious than the method itself, use this technique.

Solution

Replace calls to the method with the method’s content and delete the method itself


Before

```
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ? 2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```


After

```
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

DHIVYA DHARSHNI S V

20PW09

COUPLERS

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

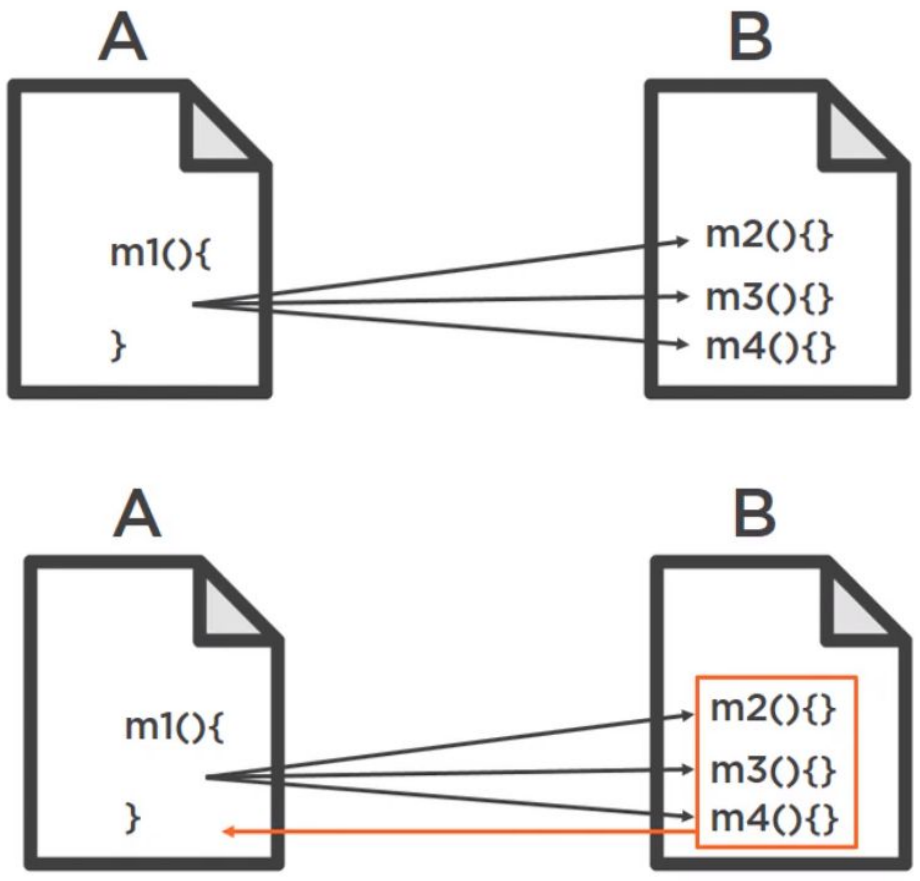
FEATURE ENVY

Signs and Symptoms :

A method accesses the data of another object more than its own data.

Reasons for the Problem

This smell may occur after fields are moved to a data class. If this is the case, you may want to move the operations on data to this class as well.

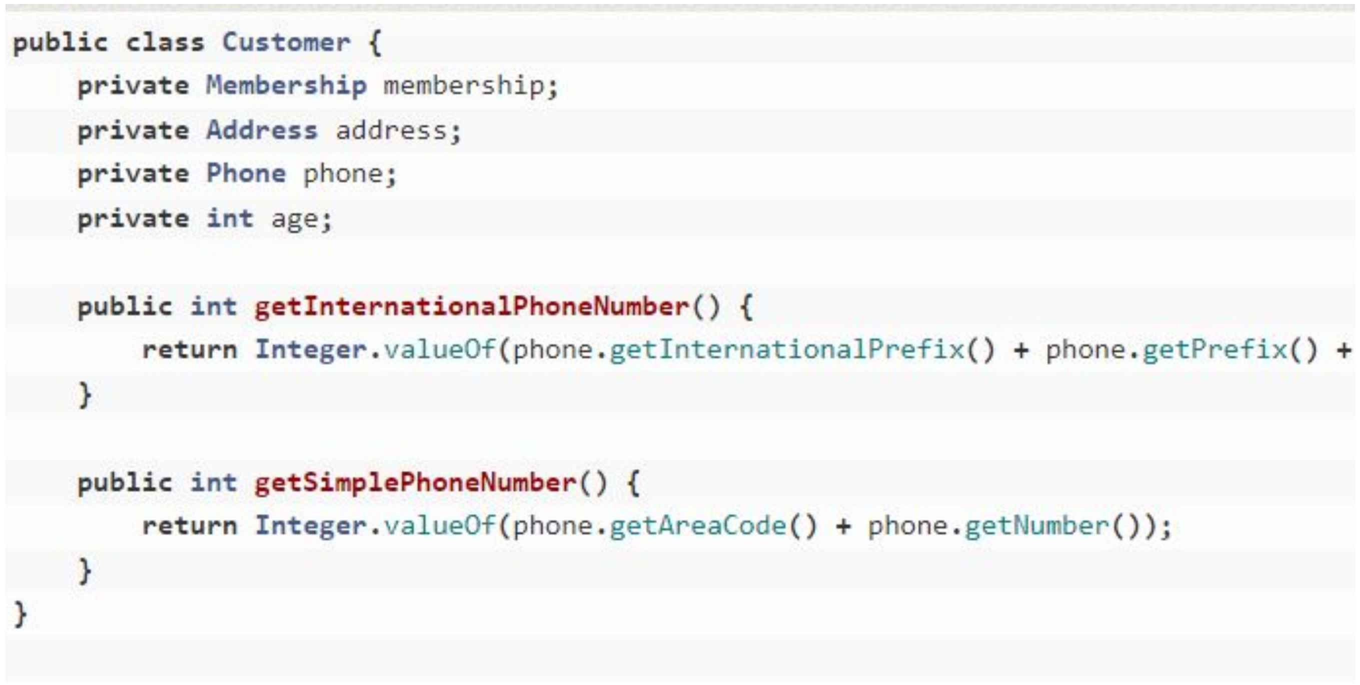


CONT.

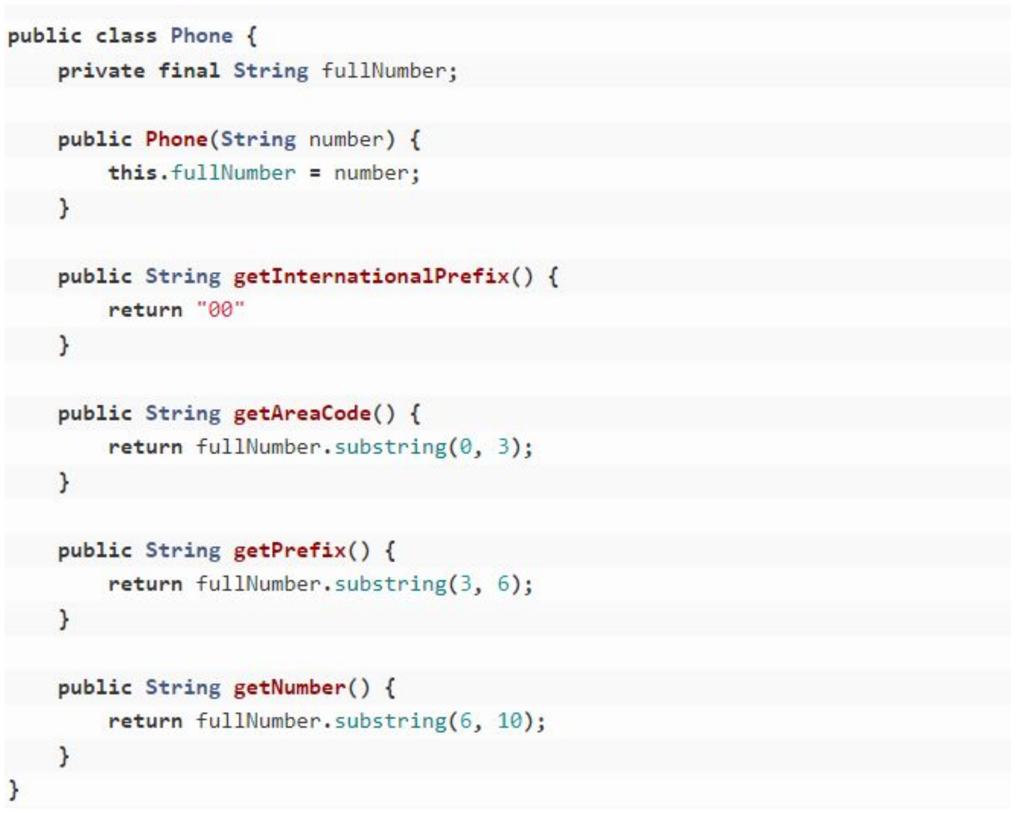
Treatment

- If a method clearly should be moved to another place, use **Move Method**.
- If only part of a method accesses the data of another object, use **Extract Method** to move the part in question.
- If a method uses functions from several other classes, first determine which class contains most of the data used. Then place the method in this class along with the other data. Alternatively, use **Extract Method** to split the method into several parts that can be placed in different places in different classes.

Example :

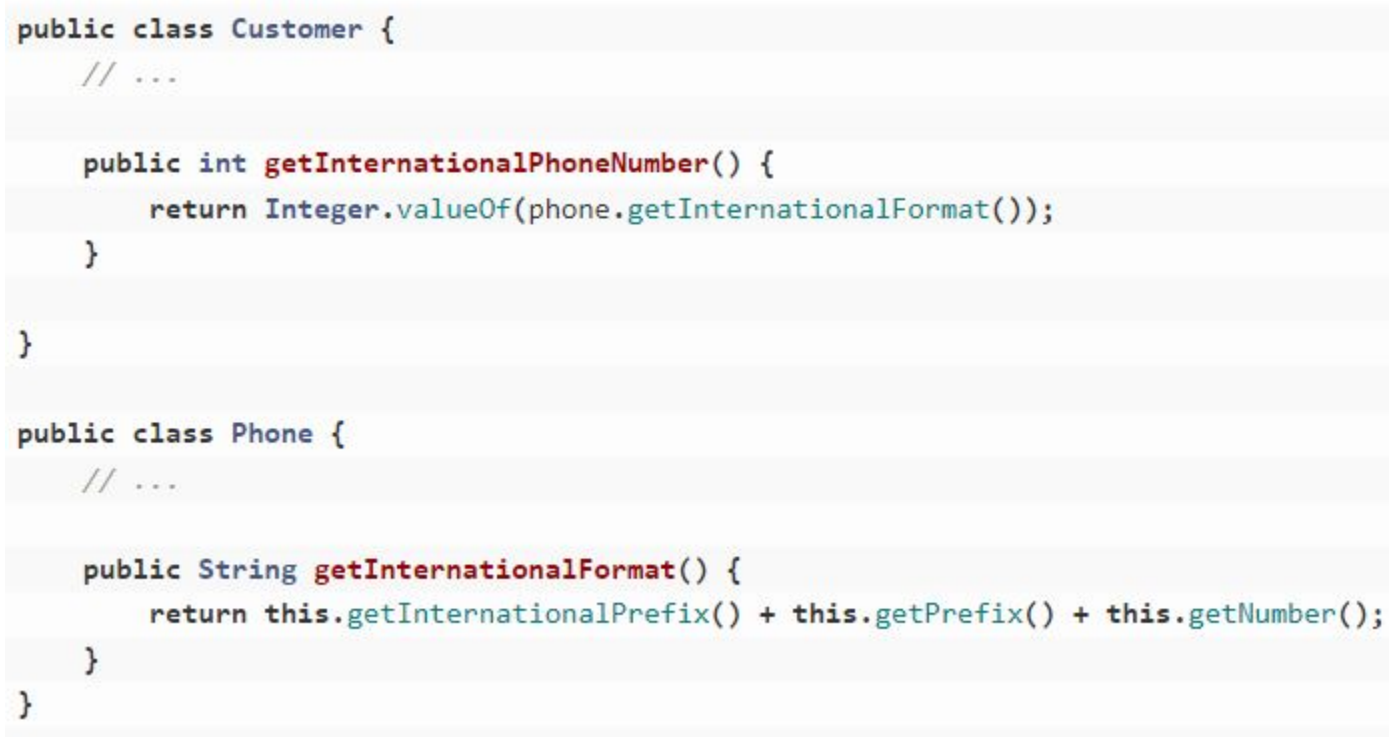


```
public class Customer {
    private Membership membership;
    private Address address;
    private Phone phone;
    private int age;

    public int getInternationalPhoneNumber() {
        return Integer.valueOf(phone.getInternationalPrefix() + phone.getPrefix() +
    }

    public int getSimplePhoneNumber() {
        return Integer.valueOf(phone.getAreaCode() + phone.getNumber());
    }
}
```



```
public class Phone {
    private final String fullNumber;

    public Phone(String number) {
        this.fullNumber = number;
    }

    public String getInternationalPrefix() {
        return "00"
    }

    public String getAreaCode() {
        return fullNumber.substring(0, 3);
    }

    public String getPrefix() {
        return fullNumber.substring(3, 6);
    }

    public String getNumber() {
        return fullNumber.substring(6, 10);
    }
}
```

CONT.



```java
public class Customer {
    // ...

    public int getInternationalPhoneNumber() {
        return Integer.valueOf(phone.getInternationalFormat());
    }

}

public class Phone {
    // ...

    public String getInternationalFormat() {
        return this.getInternationalPrefix() + this.getPrefix() + this.getNumber();
    }
}
```

CONT.

Payoff

- Less code duplication (if the data handling code is put in a central place).
- Better code organization (methods for handling data are next to the actual data).

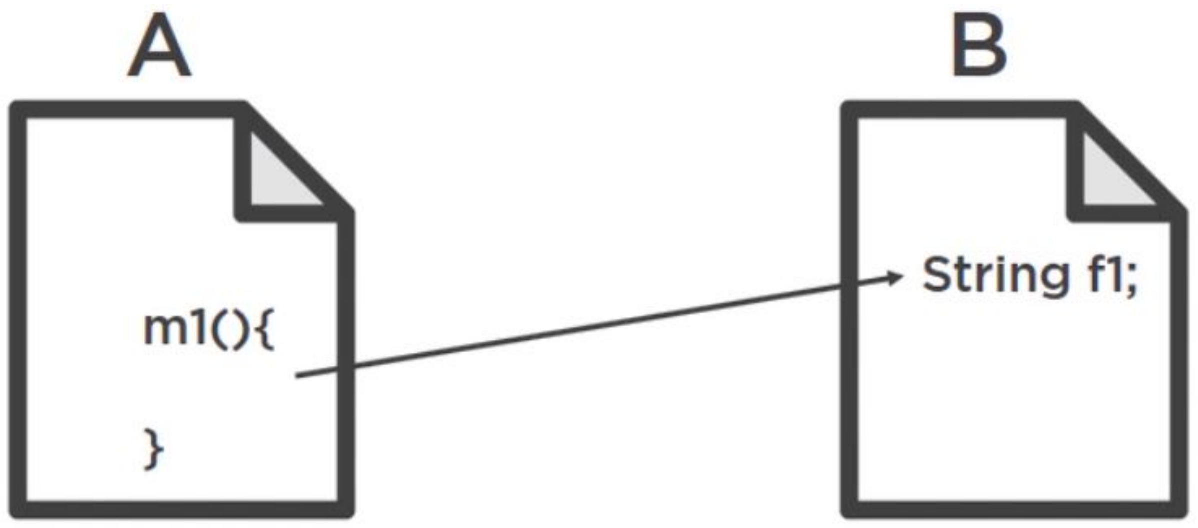
INAPPROPRIATE INTIMACY

Signs and Symptoms

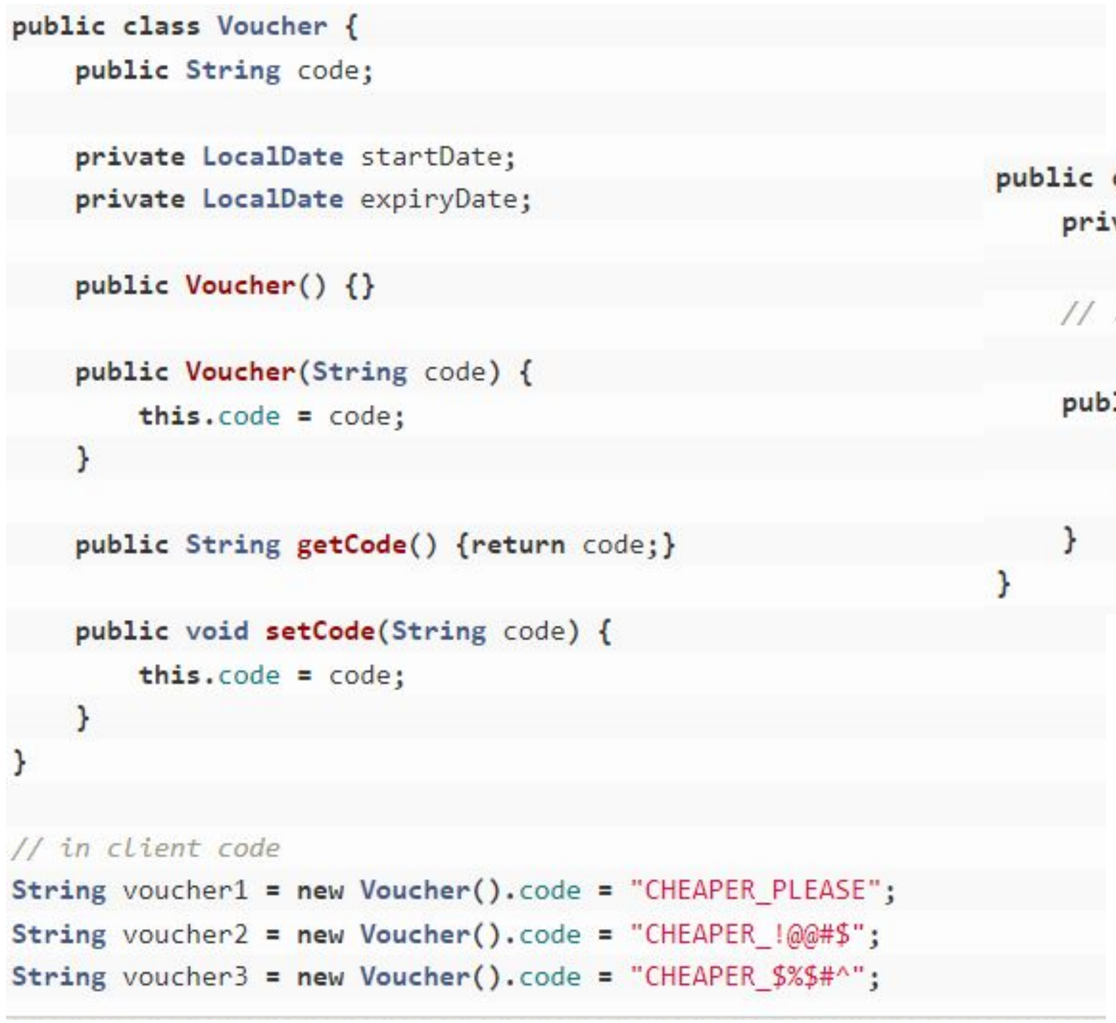
One class uses the internal fields and methods of another class.

Reasons for the Problem

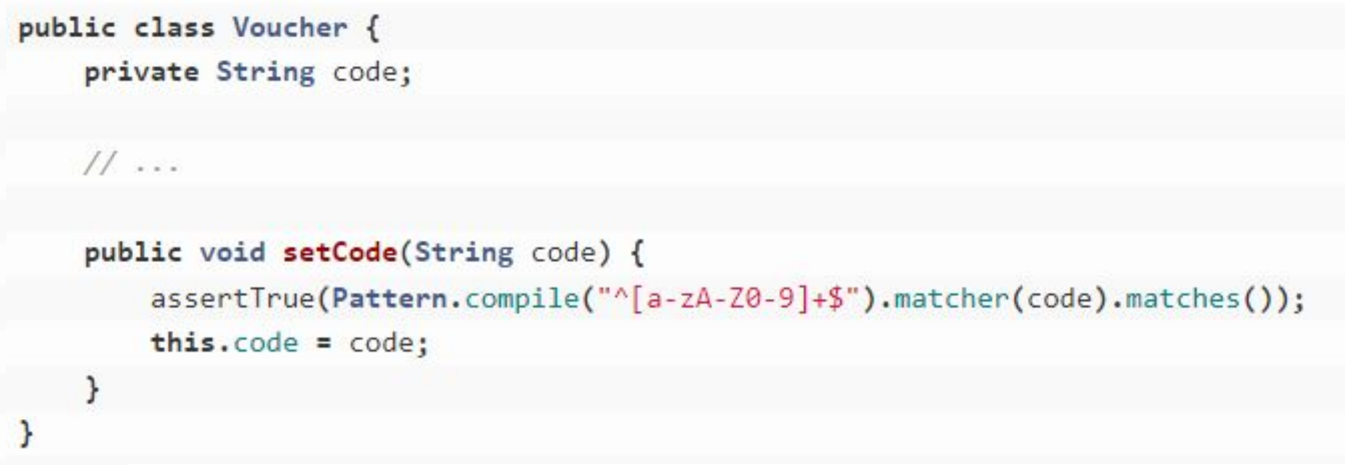
Keep a close eye on classes that spend too much time together. Good classes should know as little about each other as possible. Such classes are easier to maintain and reuse.



Example :



```java
public class Voucher {
    public String code;

    private LocalDate startDate;
    private LocalDate expiryDate;

    public Voucher() {}

    public Voucher(String code) {
        this.code = code;
    }

    public String getCode() {return code;}

    public void setCode(String code) {
        this.code = code;
    }
}

// in client code
String voucher1 = new Voucher().code = "CHEAPER_PLEASE";
String voucher2 = new Voucher().code = "CHEAPER_!@@#$";
String voucher3 = new Voucher().code = "CHEAPER_$%$#^";
```

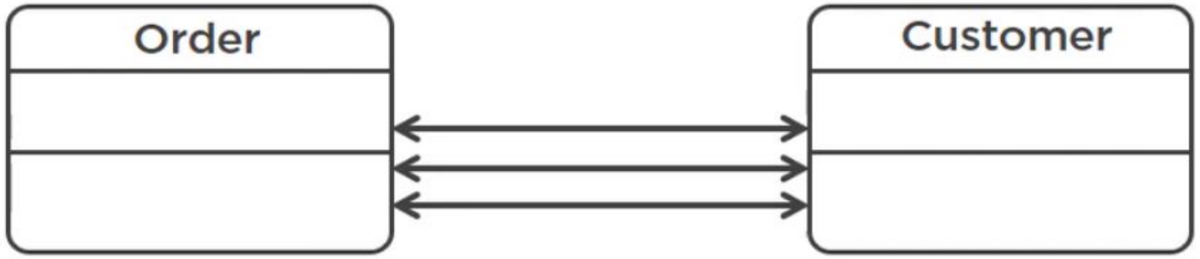


```java
public class Voucher {
    private String code;

    // ...

    public void setCode(String code) {
        assertTrue(Pattern.compile("^[a-zA-Z0-9]+$").matcher(code).matches());
        this.code = code;
    }
}
```

CONT.

Treatment

- The simplest solution is to use **Move Method** and **Move Field** to move parts of one class to the class in which those parts are used. But this works only if the first class truly doesn't need these parts.
- Another solution is to use **Extract Class** and **Hide Delegate** on the class to make the code relations "official".
- If the classes are mutually interdependent, you should use **Change Bidirectional Association to Unidirectional**.
- If this "intimacy" is between a subclass and the superclass, consider **Replace Delegation with Inheritance**.



CONT.

Payoff

- Improves code organization.
- Simplifies support and code reuse.

MESSAGE CHAINS

Signs and Symptoms

In code you see a series of calls resembling `$a->b()->c()->d()`

For example : `customer.getAddress().getCountry().toString();`

Reasons for the Problem

A message chain occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.

CONT.

Treatment

- To delete a message chain, use **Hide Delegate**.
- Sometimes it's better to think of why the end object is being used. Perhaps it would make sense to use **Extract Method** for this functionality and move it to the beginning of the chain, by using **Move Method**.

CONT.

Payoff

- Reduces dependencies between classes of a chain.
- Reduces the amount of bloated code.

MIDDLE MAN

Signs and Symptoms

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

Reasons for the Problem

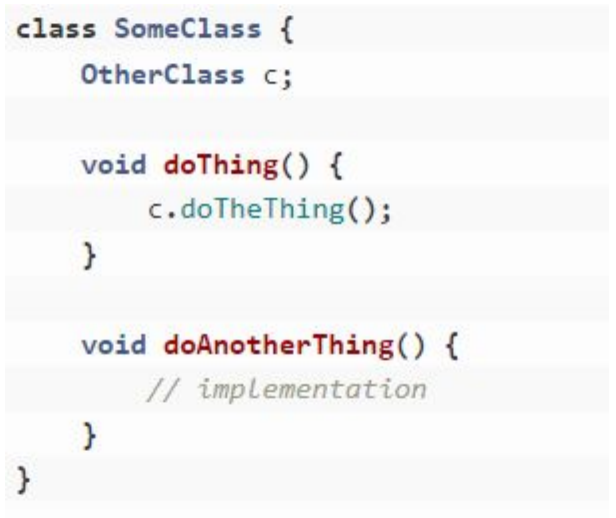
This smell can be the result of overzealous elimination of **Message Chains**.

In other cases, it can be the result of the useful work of a class being gradually moved to other classes. The class remains as an empty shell that doesn’t do anything other than delegate.

CONT.

Treatment

- If most of a method's classes delegate to another class, **Remove Middle Man** is in order.



```
class SomeClass {
    OtherClass c;

    void doThing() {
        c.doTheThing();
    }

    void doAnotherThing() {
        // implementation
    }
}
```

CONT.

Payoff

- Less bulky code.

INCOMPLETE LIBRARY CLASS

Signs and Symptoms

Sooner or later, **libraries** stop meeting user needs. The only solution to the problem—changing the library—is often impossible since the library is read-only.

Reasons for the Problem

The author of the library hasn't provided the features you need or has refused to implement them.

CONT.

Treatment

- To introduce a few methods to a library class, use **Introduce Foreign Method**.
- For big changes in a class library, use **Introduce Local Extension**.

Payoff

Reduces code duplication (instead of creating your own library from scratch, you can still piggy-back off an existing one).