# ICT365

# Software Development Frameworks

Dr Afaq Shah

Murdoch
UNIVERSITY

# Refactoring

Murdoch
UNIVERSITY

## In this Topic

- Code maintenance
- Refactoring
  - What?
  - Why?
  - When?
- Bad code smell and its types
- Refactoring Types/Techniques

## Code Maintenance

- Problem: "Bit rot‟
- After several months and new versions, many codebases reach one of the following states:

*rewritten* : Nothing remains from the original code.

*abandoned* : Original code is thrown out, rewritten from scratch.

- Why?

Systems evolve to meet new needs and add new features. If the code's structure does not also evolve, it will "rot"

This can happen even if the code was initially reviewed and well-designed at the time of checkin

## Maintenance

- ...Modification or repair of a software product after it has been delivered.

Purposes:

fix bugs

improve performance

improve design

add features

Studies have shown that ~80% of maintenance is for non-bug-fix-related activities such as adding functionality (Pigosky 1997).

## Maintenance is hard

- It's harder to maintain code than write your own new code.

"house of cards" phenomenon (don't touch it!)

must understand code written by another developer, or code you wrote at a different time with a different mindset most developers dislike code maintenance

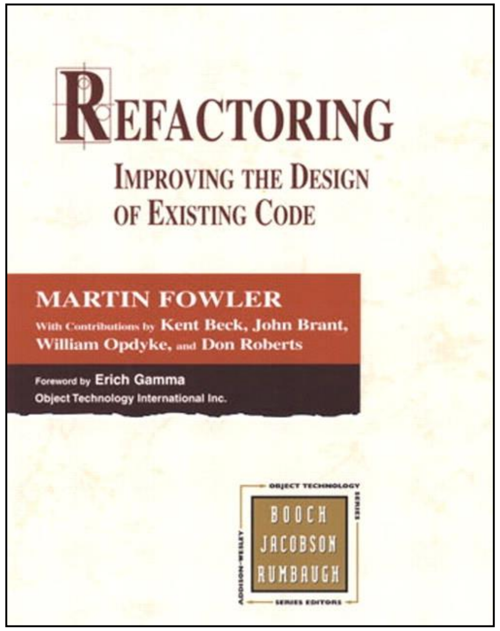
- Maintenance is how developers spend much of their time.
- It pays to design software well and plan ahead so that later maintenance will be less painful.

Capacity for future change must be anticipated

# What is Refactoring?

- The process of *changing a software system* in such a way that it *does not alter the external behaviour* of the code, yet *improves its internal structure*.
- — Fowler, et al., Refactoring, 1999.

- http://www.refactoring.com/catalog/



REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH JACOBSON RUMBAUGH

## Refactoring

- Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.
- The main purpose of refactoring is to fight technical debt. It transforms a mess into clean code and simple design.

## Clean Code

- **Clean code is obvious for other programmers.**
- Not necessarily super sophisticated algorithms. Poor variable naming, bloated classes and methods, magic numbers -you name it- all of that makes code sloppy and difficult to grasp.
- **Clean code doesn’t contain duplication.**
- Each time you have to make a change in a duplicate code, you have to remember to make the same change to every instance. This increases the cognitive load and slows down the progress.

## Clean Code

- **Clean code contains a minimal number of classes and other moving parts.**
- Less code is less maintenance. Less code is fewer bugs. Code is liability, keep it short and simple.
- **Clean code passes all tests.**
- You know your code is dirty when only 95% of your tests passed. You know you're screwed when you test coverage is 0%.
- **Clean code is easier and cheaper to maintain!**

# Why refactor?

- Why fix a part of your system that isn't broken?
- Each part of your system's code has 3 purposes:
  1. to execute its functionality,
  2. to allow change,
  3. to communicate well to developers who read it.

  If the code does not do one or more of these, it is "broken."
- Refactoring:

  improves software's design

  makes it easier to understand

## When to refactor?

- When is it best for a team to refactor their code?

best done **continuously** (like testing) as part of the process

hard to do well late in a project (like testing)

- Refactor when you identify an area of your system that:

isn't well designed

isn't thoroughly tested, but seems to work so far

now needs new features to be added

## Signs you should refactor

- code is **duplicated**
- a routine is **too long**
- a loop is too long or **deeply nested**
- a class has poor **cohesion**
- a class uses too much **coupling**
- inconsistent level of **abstraction**
- too many **parameters**
- to modify an **inheritance hierarchy** in parallel
- to **group related data** into a class
- a **"middle man"** object doesn't do much
- **poor encapsulation** of data that should be private
- a **weak subclass** doesn't use its inherited functionality
- a class contains **unused code**

## The Basic Rule of Refactoring

- "Refactor the low hanging fruit" http://c2.com/cgi/wiki?RefactorLowHangingFruit
- Low Hanging Fruit (def): "The thing that gets you most value for the least investment."
- In other words, don't spend much time on it.

## Checklist of refactoring

- The code should become cleaner.
- New functionality shouldn’t be created during refactoring.
- All existing tests must pass after refactoring.

# Some advice from Fowler

- *"When should I refactor? How often? How much time should I dedicate to it?"*

It's not something you should dedicate two weeks for every six months …

… rather, you should do it as you develop!

Refactor when you recognize a warning sign (a "bad smell") and know what to do

… when you add a function

Likely it's not an island unto itself

… when you fix a bug

Is the bug symptomatic of a design flaw?

… when you do a code review

A good excuse to re-evaluate your designs, share opinions.

## Bad Code Smells

- Refactoring literature has notion of “code smells”
- “If it stinks, change it” (M. Fowler, *Refactoring*)
- A characteristic of a design that is a strong indicator it has poor structure, and should be refactored

## Bad Smells in Code

- Duplicated Code—cut and pasted everywhere
- Long Method—hard to understand
- Large Class—everything including kitchen sink
- Long Parameter List—multi-line calls
- Divergent Change—Single Responsibility Principle.
- Shotgun Surgery—can’t change just one thing
- Feature Envy—a class needs methods from another class.
- Data Clumps—data always used together
- Primitive Obsession—procedural coding style
- Switch Statements—and duplicated cases

## Code Smells: Duplicated Code

- Duplicated code (code clones)

  The same, or very similar code, appears in many places

  Problem

  - A bug fix in one code clone may not be propagated to all
  - Makes code larger that it needs to be

  Fix: *extract method* refactoring

  - Create new method that encapsulates duplicated code
  - Replace code clones with method call

# Bad smells in code

- ***Duplicated code***

  "The #1 bad smell"

  Same expression in two methods in the same class?

    Make it a `private` ancillary routine and parameterize it

      *(Extract method)*

  Same code in two related classes?

    Push commonalities into closest mutual ancestor and parameterize

    Use *template method* DP for variation in subtasks

      *(Form template method)*

## Bad smells in code

- **Payoff**
- Merging duplicate code simplifies the structure of your code and makes it shorter.
- Simplification + shortness = code that’s easier to simplify and cheaper to support.

## Code smells: Long Method

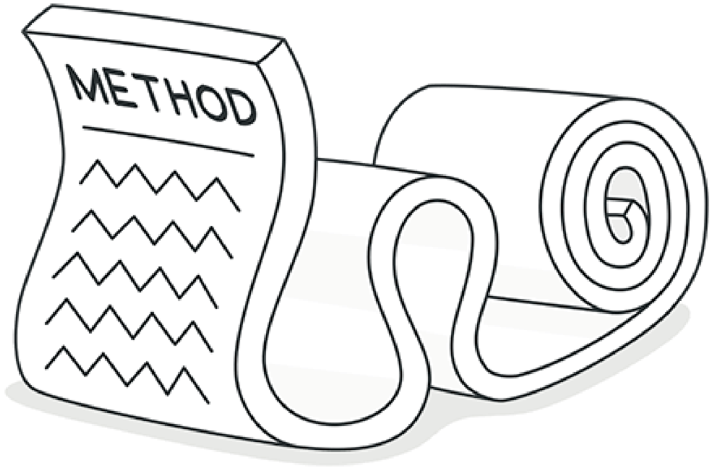
- Long method
- A method that has too many lines of code

  How long is too long? Depends.

  Over 20 is usually a bad sign.

  Under 10 lines is typically good.

  Still, no hard and fast rules.



Problem

  The longer a method, the harder it is to understand, change, and reuse

Fix: extract method

  Take chunks of code from inside long method, and make a new method

  Call new method inside the now-not-so-long method.

## Bad smells in code

- ***Long method***

  Often a sign of:

  Trying to do too many things

  Poorly thought out abstractions and boundaries

  Best to think carefully about the major tasks and how they inter-relate. Be aggressive!

  Break up into smaller `private` methods within the class

  *(Extract method)*

  Delegate subtasks to subobjects that "know best" (*i.e.*, template method DP)

  *(Extract class/method, Replace data value with object)*

# Bad smells in code

- ***Long method***

  Fowler's heuristic:

  *When you see a comment, make a method.*

  Often, a comment indicates:

  - The next major step
  - Something non-obvious whose details detract from the clarity of the routine as a whole.

  In either case, this is a good spot to "break it up".

## Code smells: Feature Envy

- Feature Envy

  A method in one class uses primarily data and methods from another class to perform its work

  Problem:

  - Indicates abstraction fault.
  - Ideally want data, and actions on that data, to live in the same class.
  - Feature Envy indicates the method was incorrectly placed in the wrong class

  Fix:

  - Move method
    - Move the method with feature envy to the class containing the most frequently used methods and data items

## Bad smells in code

- ***Feature envy***

  A method seems more interested in another class than the one it's defined in

  *e.g.*, a method `A::m()` calls lots of get/set methods of class `B`

  Solution:

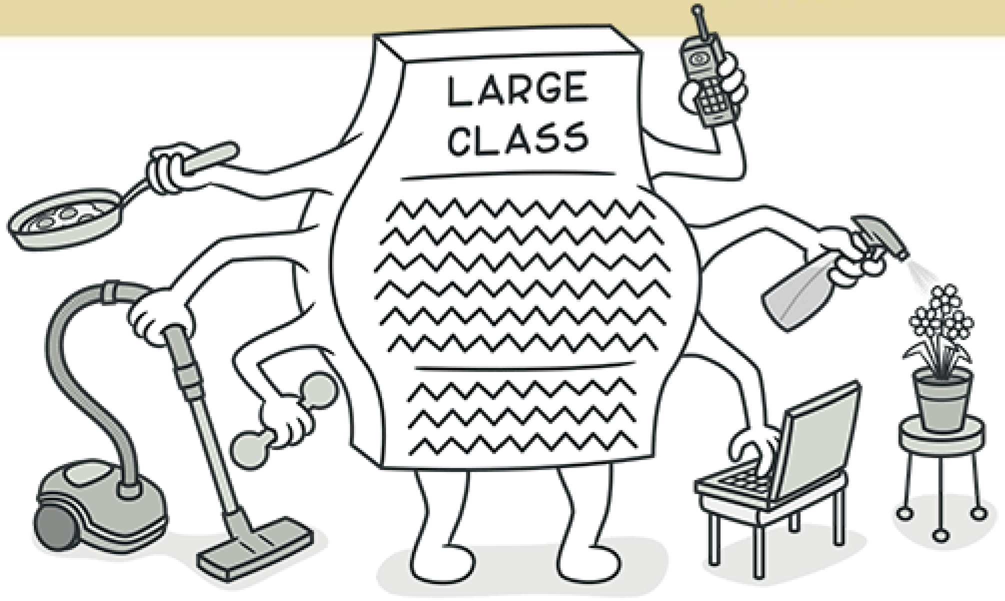
  Move `m()` (or part of it) into `B`!

  *(Move method/field, extract method)*

## Code smells: Feature Envy

- **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).

## Code smells: Large class



- Large class
  - A class is trying to do too much
  - Many instance variables
  - Many methods
- Problem:
  - Indicates abstraction fault
    - There is likely more than one concern embedded in the code
    - Or, some methods belong on other classes
  - Associated with duplicated code
- Fix:
  - *Extract class* refactoring
    - Take a subset of the instance variables and methods and create a new class with them
    - This makes the initial (long) class shorter
  - *Move method* refactoring
    - Move one or more methods to other classes

## Bad smells in code

- ***Large class***

  *i.e.,* too many different subparts and methods

  Two step solution:

  1. Gather up the little pieces into aggregate subparts.

     *(Extract class, replace data value with object)*

  2. Delegate methods to the new subparts.

     *(Extract method)*

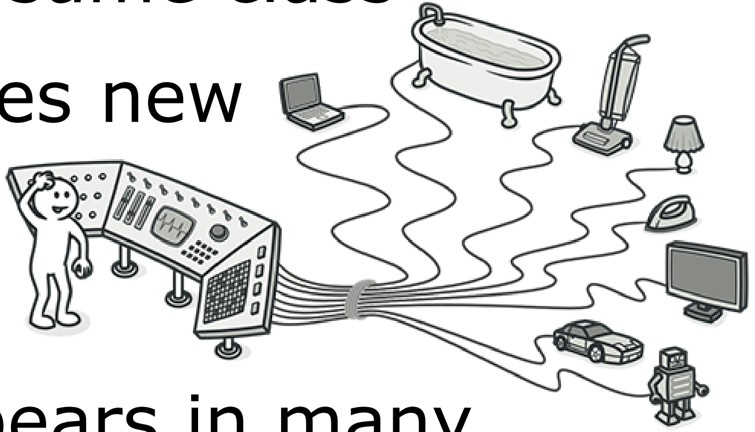
  Likely, you'll notice some unnecessary subparts that have been hiding in the forest!

## Code smells: switch statements

- Switch statements

  The cases in a switch statement contain logic for different types of instances of the same class

  In object-oriented code, this indicates new subclasses should be created



- Problem

  The same switch/case structure appears in many places

- Fix

  Create new subclasses

  Extract method to move case block logic into methods on the new subclasses

# Bad smells in code

- ***Switch statements***

  Here's Fowler's example:

```
Double getSpeed () {
     switch (_type) {
    case EUROPEAN:
         return getBaseSpeed();
    case AFRICAN:
         return getBaseSpeed() -
        getLoadFactor() * _numCoconuts;
    case NORWEGIAN_BLUE:
         return (_isNailed) ? 0
        : getBaseSpeed(_voltage);
     }
}
```

# Code smells: Data class

- **Data class**

  A class that has only class variables, getter/setter methods/properties, and nothing else

  Is just acting as a data holder

- Problem

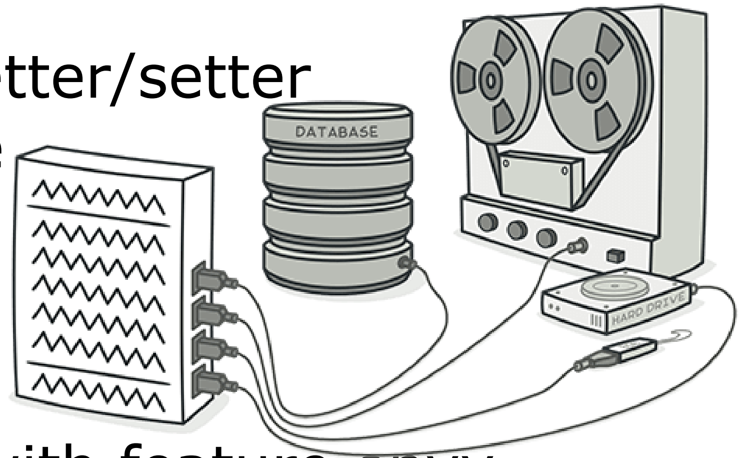
  Typically, other classes have methods with feature envy

  That is, there are usually other methods that primarily manipulate data in the data class

  Indicates these methods should really be on the data class

- Fix

  Examine methods that use data in the data class, and use move method refactoring to shift methods



## Bad smells in code

- **Data clumps**

You see a set of variables that seem to "hang out" together

*e.g.*, passed as parameters, changed/accessed at the same time

Usually, this means that there's a coherent subobject just waiting to be recognized and encapsulated

```
void Scene::setTitle (string titleText,
                      int titleX, int titleY,
                      Colour titleColour){…}

void Scene::getTitle (string& titleText,
                      int& titleX, int& titleY,
                      Colour& titleColour){…}
```

# Bad smells in code

- ***Data clumps***

  In the example, a `Title` class is dying to be born

  If a client knows how to change a title's `x`, `y`, `text`, and `colour`, then it knows enough to be able to "roll its own" `Title` objects.

  - However, this does mean that the client now has to talk to another class.

  This will greatly shorten and simplify your parameter lists (which aids understanding) and makes your class conceptually simpler too.

  Moving the data may create ***feature envy*** initially

  - May have to iterate on the design until it feels right.
  - *(Preserve whole object, extract class, introduce parameter object)*

## Bad smells in code

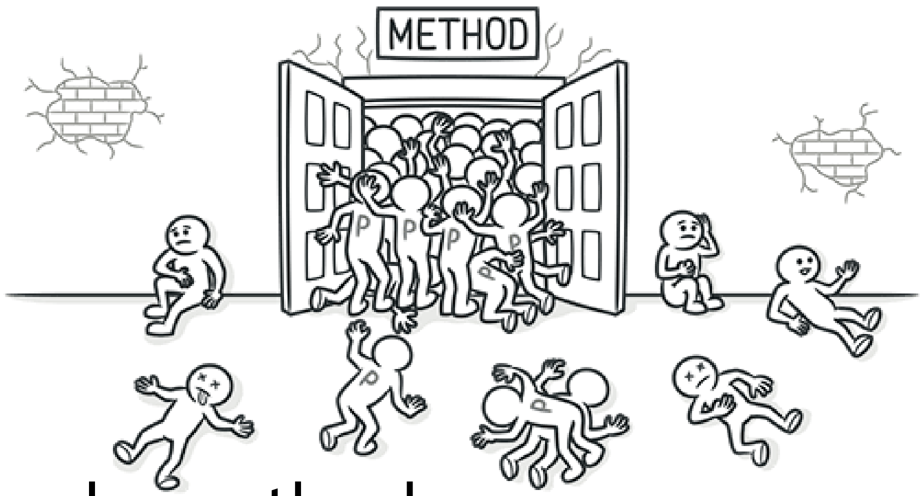
- **Long parameter list**

  Long parameter lists make methods difficult for clients to understand

  This is often a symptom of

    Trying to do too much

    With too many disparate subparts



- **Reasons for the Problem**

several algorithms merged in a single method

byproduct of efforts to make classes  independently

## Bad smells in code

- ***Long parameter list***

  Solution:

  - Trying to do too much?
    - Break up into sub-tasks
      - *(Extract method)*
  - … too far from home?
    - Localize passing of parameters;

## Bad smells in code

- **Divergent change**

Occurs when one class is commonly changed in different ways for different reasons

*Divergent Change* is when many changes are made to a single class

Likely, this class is trying to do too much and contains too many unrelated subparts

This is a sign of *poor cohesion*

Unrelated elements in the same container

Solution:

Break it up, reshuffle, reconsider relationships and responsibilities *(Extract class)*

# Bad smells in code

- **Shotgun surgery**

… the opposite of divergent change

Shotgun Surgery refers to when a single change is made to multiple classes simultaneously.

Each time you want to make a single, seemingly coherent change, you have to change lots of classes in little ways

Also a classic sign of poor cohesion

Related elements are *not* in the same container!

Solution:

Look to do some gathering, either in a new or existing class.

*(Move method/field)*

## Bad smells in code

- **Primitive obsession**

All subparts of an object are instances of primitive types

(`int, string, bool, double,` *etc.*)

*e.g.,* dates, currency, tel.#, ISBN, special string values

Often, these small objects have interesting and non-trivial constraints that can be modelled

*e.g.,* fixed number of digits/chars, check digits, special values

Solution:

Create some "small classes" that can validate and enforce the constraints.

This makes your system mode strongly typed.

# Bad smells in code

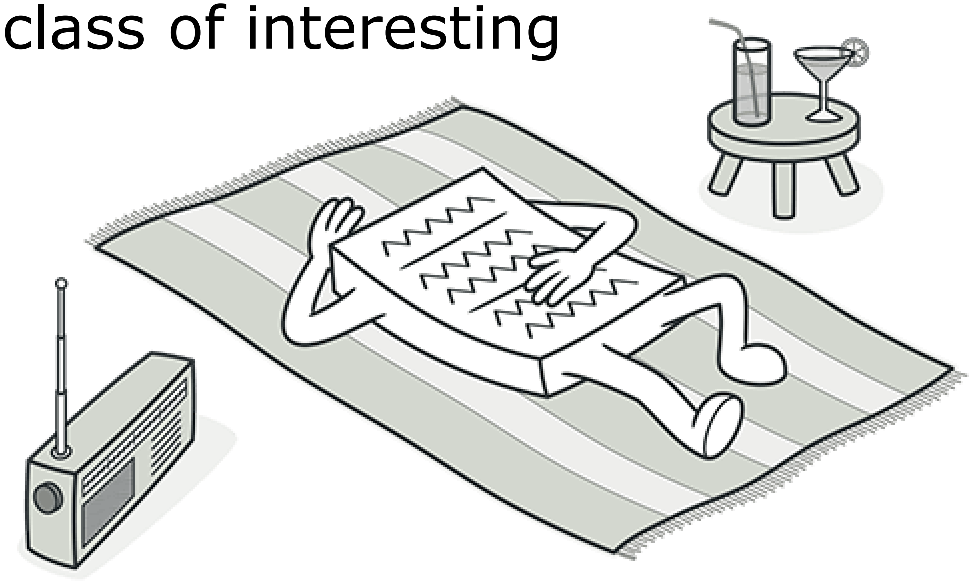
- **Lazy class**

Classes that don't do much that's different from other classes.

If there are several sibling classes that don't exhibit polymorphic behavioural differences , the consider just collapsing them back into the parent and add some parameters

Often, ***lazy classes*** are legacies of ambitious design or a refactoring that gutted the class of interesting behaviour

*(Collapse hierarchy, inline class)*



## Bad smells in code

- **Speculative generality**

  *"We might need this one day …"*

  - Fair enough, but did you really need it after all?
  - Extra classes and features add to complexity.

  XP philosophy:

  - "As simple as possible but no simpler."

  Keep in mind that refactoring is an ongoing process.

  - If you really do need it later, you can add it back in.

## Bad smells in code

- **Message chains**

Client asks an object which asks a subobject, which asks a subobject, …

Multi-layer "drill down" may result in sub-sub-sub-objects being passed back to requesting client.

Sounds like the client already has an understanding of the structure of the object, even if it is going through appropriate intermediaries.

Probably need to rethink abstraction …

## Bad smells in code

- **Middle man**

*"All hard problems in software engineering can be solved by an extra level of indirection."*

If you notice that many of a class's methods just turn around and beg services of delegate subobjects, the basic abstraction is probably poorly thought out.

An object should be more than the some of its parts in terms of behaviours!

*(Remove middle man, replace delegation with inheritance)*

# Bad smells in code

- **Inappropriate intimacy**

Sharing of secrets between classes, esp. outside of the holy bounds of inheritance

*e.g.*, `public` variables, indiscriminate definitions of get/set methods, C++ friendship, `protected` data in classes

Leads to data coupling, intimate knowledge of internal structures and implementation decisions.

Makes clients brittle, hard to evolve, easy to break.

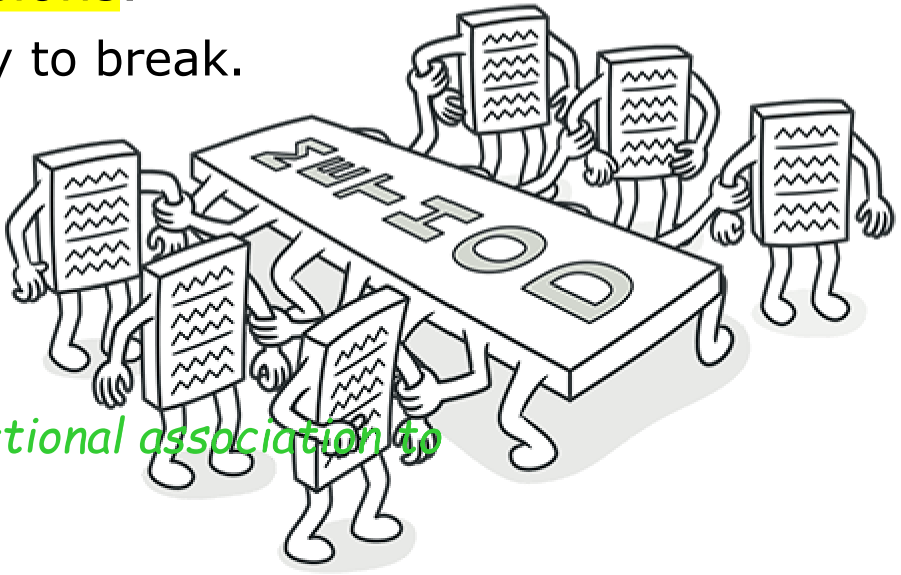
Solution:

*Appropriate* use of get/set methods

Rethink basic abstraction.

Merge classes

*(Move/extract method/field, change bidirectional association to unidirectional, hide delegate)*



## Bad smells in code

- **Alternative classes with different interfaces**

Classes/methods seem to implement the same or similar abstraction yet are otherwise unrelated.

This is not a knock against overloading, just haphazard design.

Solution:

Move the classes “closer” together.

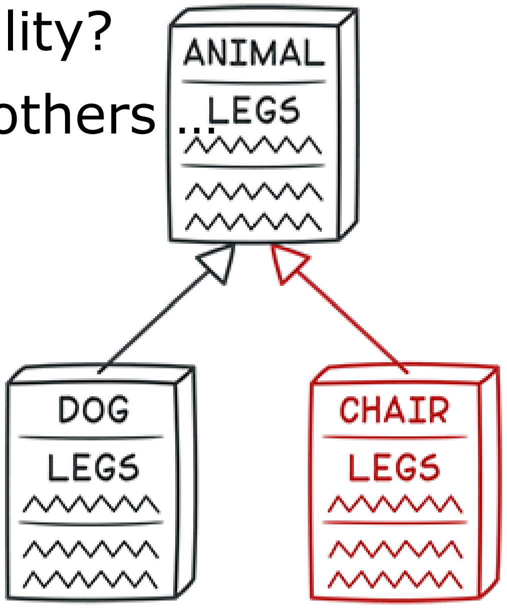
Find a common interface.

Find a common subpart and remove it.

*(Extract [super]class, move method/field, rename method)*

# Bad smells in code

- **Refused bequest**
  - Subclass inherits methods/variables but doesn't seem to use some of them.
    - In a sense, this might be a good sign:
      - The parent manages the commonalities and the child manages the differences.
    - Might want to look at typical client use to see if clients think child is-a parent
      - Do clients use parent's methods? … use parent as static type?
  - Did the subclass inherit as a cheap pickup of functionality?
    - Fowler/Beck claim this isn't as bad a smell as the others …
    - Might be better to use delegation
      - *(Replace inheritance with delegation)*



# Bad smells in code

- **Refused bequest**

  Another perspective:

  - Parent has features that are used by only some of its children.

  Typical solution is to create some more intermediate abstract classes in the hierarchy.

  - Move the peculiar methods down a level.

    *(Push down field/method)*

# Bad smells in code

- **Comments**

A method is filled with explanatory comments.

In the context of refactoring, Fowler claims that long comments are often a sign of opaque, complicated, inscrutable code.

They aren't against comments so much as in favour of self-evident coding practices.

Rather than explaining opaque code, restructure it!

*(Extract method/class, [many others applicable] …)*

Comments are best used to document rationale

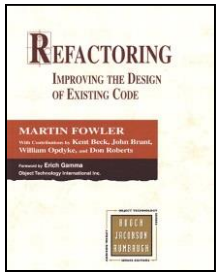
*i.e.*, explain *why* you picked one approach over another.

## Refactoring Types/Techniques

## Some types of refactoring

- refactoring to fit design **patterns**
- **renaming** (methods, variables)
- **extracting** code into a method or module
- **splitting** one method into several to improve cohesion and readability
- changing method **signatures**
- performance **optimization**
- **moving** statements that semantically belong together near each other
- naming (extracting) **constants**
- **clarifying** a statement that has evolved over time or is unclear

See also http://www.refactoring.org/catalog/

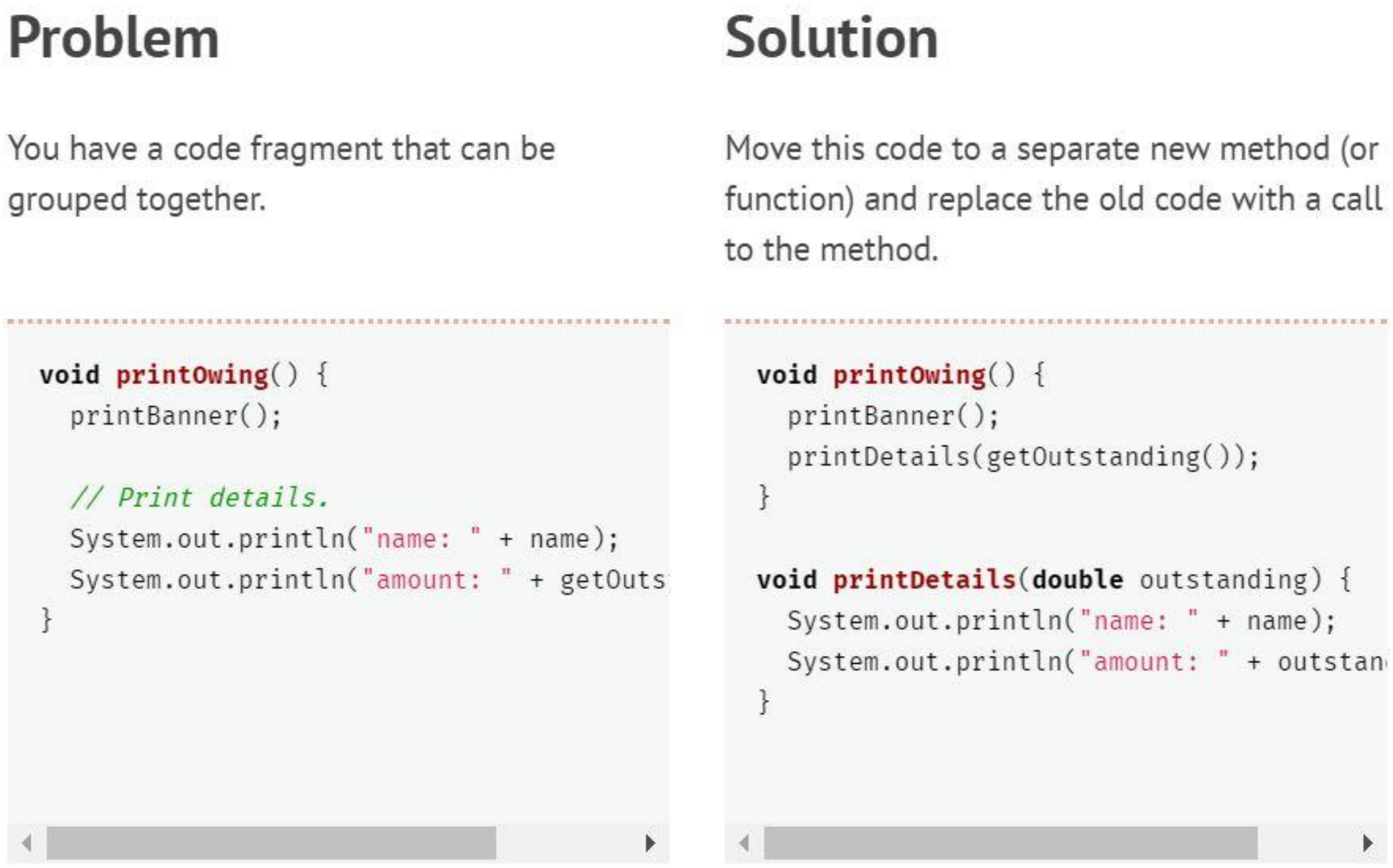

REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH JACOBSON RUMBAUGH

## Types of Refactoring

- Extract Method Refactoring
- Rename Refactoring
- Encapsulate Field Refactoring
- Remove Parameters Refactoring
- Reorder Parameters Refactoring
- Extract Interface Refactoring

**Your Task:** Review and complete Lab tasks.
Important for Assignment 2

# Extract Method Refactoring



## Problem

You have a code fragment that can be grouped together.

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

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

## Solution

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

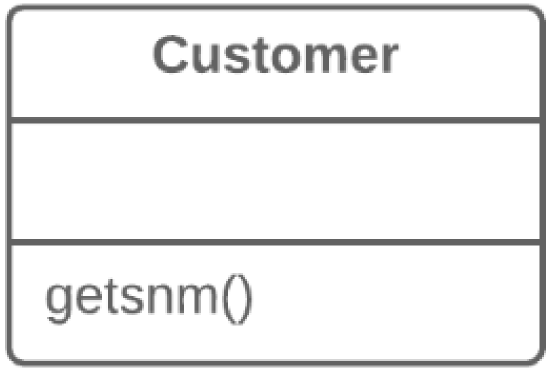
```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

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

## Rename Refactoring

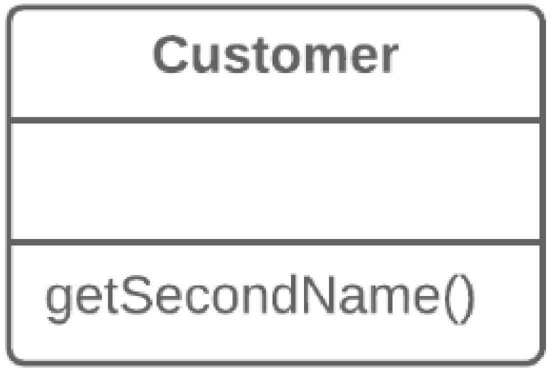
### Problem

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

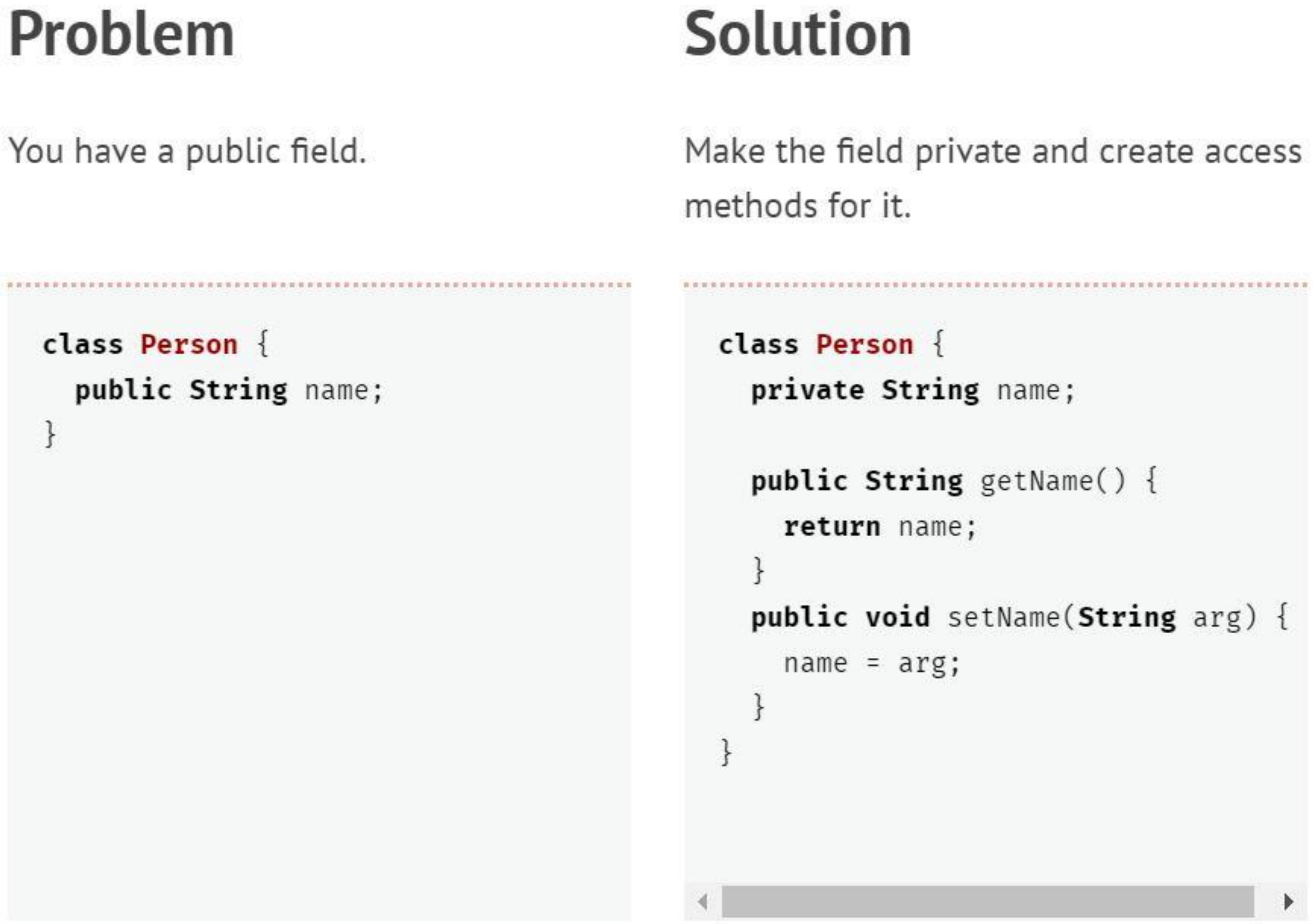


### Solution

Rename the method.



## Encapsulate Field Refactoring



### Problem

You have a public field.

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

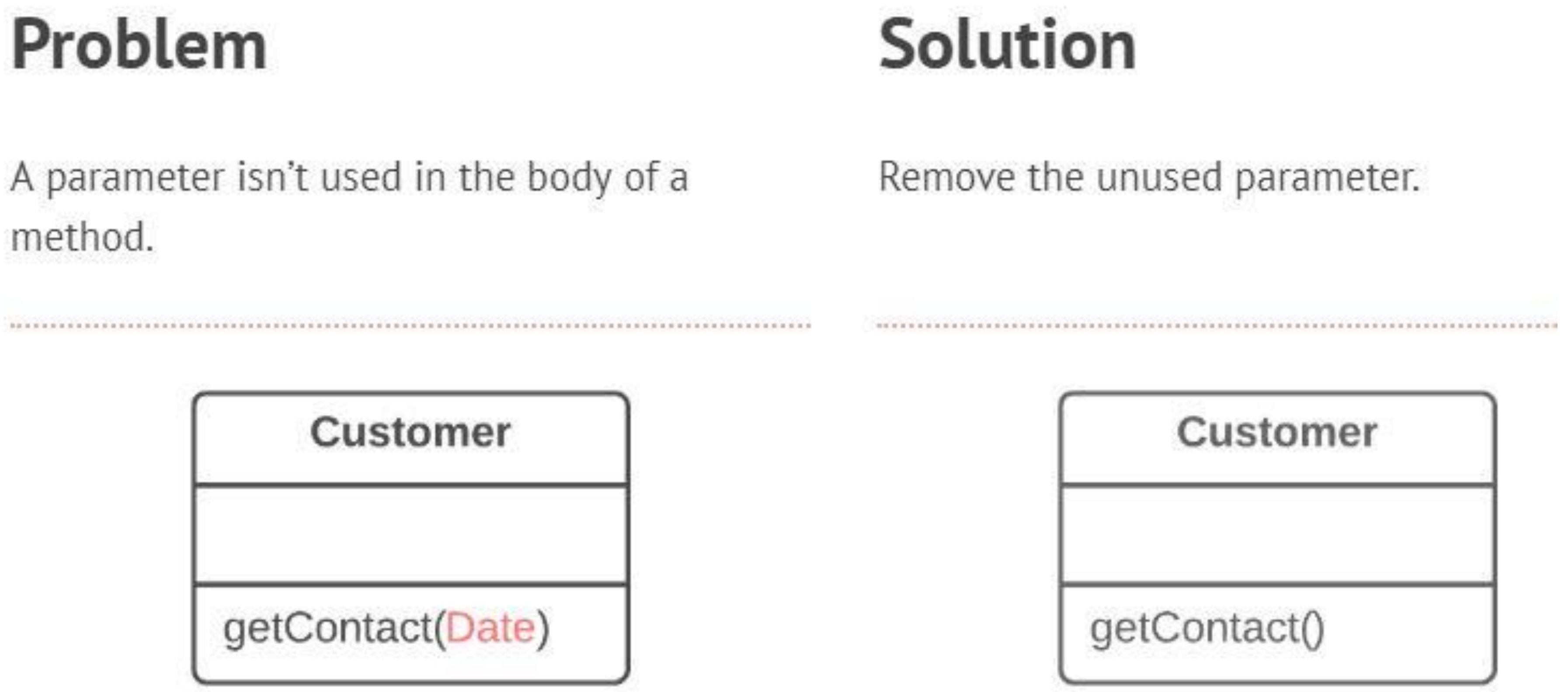
### Solution

Make the field private and create access methods for it.

```java
class Person {
  private String name;

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

## Remove Parameter Refactoring



**Problem**

A parameter isn't used in the body of a method.

**Solution**

Remove the unused parameter.

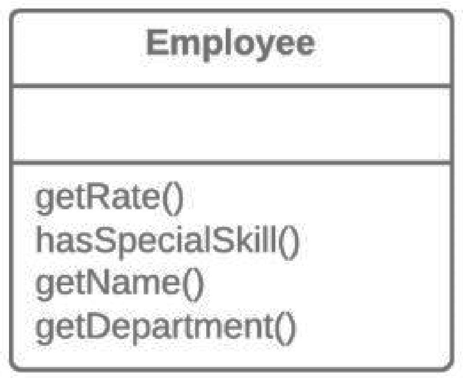
# Reorder Parameter Refactoring

- Reorder Parameters is a Visual C# refactoring operation that provides an easy way to change the order of the parameters for methods
- This **Refactoring** is used to change the method parameter order and automatically update all method references.
- Reorder Parameters changes the declaration, and at any locations where the member is called, the parameters are rearranged to reflect the new order.
- Use it when you need to standardize the method signature or make the parameters order more logical.

# Extract Interface Refactoring

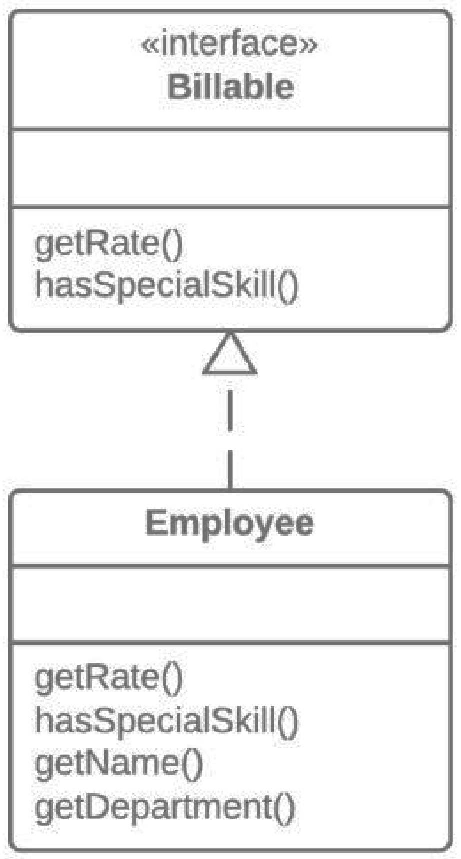
## Problem

Multiple clients are using the same part of a class interface. Another case: part of the interface in two classes is the same.



## Solution

Move this identical portion to its own interface.



## Learning Refactoring

- Pick a goal—something's wrong with the program. Resolve to fix it. Then fix it.
- **Stop** when you're unsure.
- Backtrack. Use test-driven development to make sure you haven't broken something. If you have, back up.
- Work with a friend. 'We explain to each other what we don't understand.' As long as you don't share your code, we'll never notice, and you'll learn a lot in the process.

## C# Reference

- **Refactoring (C#)**
- https://msdn.microsoft.com/en-us/library/719exd8s.aspx
- **Extract Method Refactoring (C#)**
- https://msdn.microsoft.com/en-us/library/0s21cwxk(v=vs.120).aspx
- **Rename Refactoring (C#)**
- https://msdn.microsoft.com/en-us/library/6kxxabwd(v=vs.120).aspx
- **Encapsulate Field Refactoring (C#)**
- https://msdn.microsoft.com/en-us/library/a5adyhe9(v=vs.120).aspx

## C# Reference

- **Extract Interface Refactoring (C#)**
- https://msdn.microsoft.com/en-us/library/fb3dyx26(v=vs.120).aspx
- **Remove Parameters Refactoring (C#)**
- https://msdn.microsoft.com/en-us/library/0c7wac46(v=vs.120).aspx
- **Reorder Parameters Refactoring (C#)**
- https://msdn.microsoft.com/en-us/library/5ss5z206(v=vs.120).aspx

## Reading:

- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.
- Previously:
- Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002.
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Mass., 1995.
- Sherman R. Alpert, Kyle Brown and Bobby Woolf, *The Design Patterns Smalltalk Companion*, Addison Wesley, 1998.

## Other reading:

- *Extreme Programming Explained*, by Kent Beck
- Anything by Jon Bentley - *Programming Pearls*, *More Programming Pearls*, *Writing Efficient Programs*

## Acknowledgements

Sources used in this presentation include:

- Refactoring and Code Maintenance by Marty Stepp
- Harry Erwin
- Refactoring.guru