DVIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**MINI ACTICLE OF DESIGN PATTERN**

**BAD SMELL IN CODE**

*Lecturer:* **MSc NGUYỄN THANH PHƯỚC**

*Students*: **LÊ TẤN TÀI - 518H0114**

**PHAN AN DUY – 518h0616**

Class **: 18H50302, 18H50301**

Course  **: 22**

**HO CHI MINH CITY, YEAR 2021**

VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**MINI ACTICLE OF DESIGN PATTERN**

**BAD SMELL IN CODE**

*Lecturer:* **MSc NGUYỄN THANH PHƯỚC**

*Students*: **LÊ TẤN TÀI - 518H0114**

**PHAN AN DUY – 518H0616**

Class **: 18H50302, 18H50301**

Course  **: 22**

**HO CHI MINH CITY, YEAR 2021**

ACKNOWLEDGEMENT

I am using this opportunity to express my gratitude to everyone who supported me throughout the course of this DESIGN PATTERN. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work. I am sincerely grateful to them for sharing their truthful and illuminating views o­n a number of issues related to this project.

I express my special thanks to Mr. Nguyen Thanh Phuoc for his support and guidance at Ton Duc Thang University.

I would also like to thank my project external all the people who provided me with the facilities being required and conductive conditions for my project.

Sincerely thanks with all my love and respect.

# THE PROJECT WAS COMPLETED AT TON DUC THANG UNIVERSITY

I pledge that this is a product of my own project and is under the guidance of Mr. Nguyen Thanh Phuoc. The content of research, results in this subject is honest and not published in any form before. The data in the tables used for the analysis, comment, and evaluation were collected by the authors themselves from various sources indicated in the reference section.

In addition, many comments and assessments as well as data from other authors and organizations have been used in the project, with references and annotations.

**If any fraud is found, I am fully responsible for the content of my project.** Ton Duc Thang University is not involved in any copyright infringement or copyright infringement in the course of implementation (if any).

*Ho Chi Minh City, March 03rd 2021*

*Author*

*(Sign and write full name)*

*Lê Tấn Tài*

*Phan An Duy*

# EVALUATION OF INSTRUCTING LECTURER

**Confirmation of the instructor**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ho Chi Minh City, March 03rd 2021

(Sign and write full name)

**The assessment of the teacher marked**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ho Chi Minh City, March 03rd 2021

(Sign and write full name)

**TABLE OF CONTENTS**

[ACKNOWLEDGEMENT 1](#_Toc41218920)

[THE PROJECT WAS COMPLETED AT TON DUC THANG UNIVERSITY 2](#_Toc41218921)

[EVALUATION OF INSTRUCTING LECTURER 3](#_Toc41218922)

[TABLE OF CONTENTS 4](#_Toc41218923)

[CHAPTER 1 – INTRODUCTION 6](#_Toc41218924)

[1.1 Concept of the tree data structure 6](#_Toc41218925)

[1.2 Binary Search Tree (BST) 7](#_Toc41218926)

[1.3 AVL Tree 9](#_Toc41218927)

[1.4 `Heap 9](#_Toc41218928)

[1.5 Overview of Red-Black Tree and B-Tree 10](#_Toc41218929)

[CHAPTER 2 – RED-BLACK TREE 10](#_Toc41218930)

[2.1 Definition 10](#_Toc41218932)

[2.2 Operations 11](#_Toc41218933)

[2.2.1 Rotation in a Red-Black Tree 11](#_Toc41218934)

[2.2.2 Insertion 13](#_Toc41218935)

[2.2.3 Deletion 17](#_Toc41218936)

[2.3 Implementation of the corresponding data structures 19](#_Toc41218937)

[CHAPTER 3 – B-TREE 20](#_Toc41218938)

[3.1 Definition 20](#_Toc41218939)

[3.2 Operations 22](#_Toc41218940)

[3.2.1 Searching a B-tree 22](#_Toc41218941)

[3.2.2 Insertion 23](#_Toc41218942)

[3.2.3 Deletion 26](#_Toc41218943)

[3.3 Implementation of the corresponding data structures 30](#_Toc41218944)

[CHAPTER 4 – DEMO 31](#_Toc41218945)

[CHAPTER 5 – CONCLUSION AND DISCUSIONS 31](#_Toc41218946)

[REFERENCES 32](#_Toc41218947)

[SELF-EVALUATION 33](#_Toc41218948)

[DEMO 35](#_Toc41218949)

CHAPTER 1 – INTRODUCE THE PROBLEM

* 1. Introduction
* Code smells are indications of poor coding and design choices that can cause problems during the later phase of software development.
* Bad Smell is a term that has been used for messy or dirty coding, this term says there is part of the code which needs to be clean in term of future.
* Bad smell is known as “bad smelling code” indicate that the code needs to be refactored.
* Refactoring is about how to change code by applying refactoring.
* Bad smells allow us to identify what needs to be changed in order to improve the code.

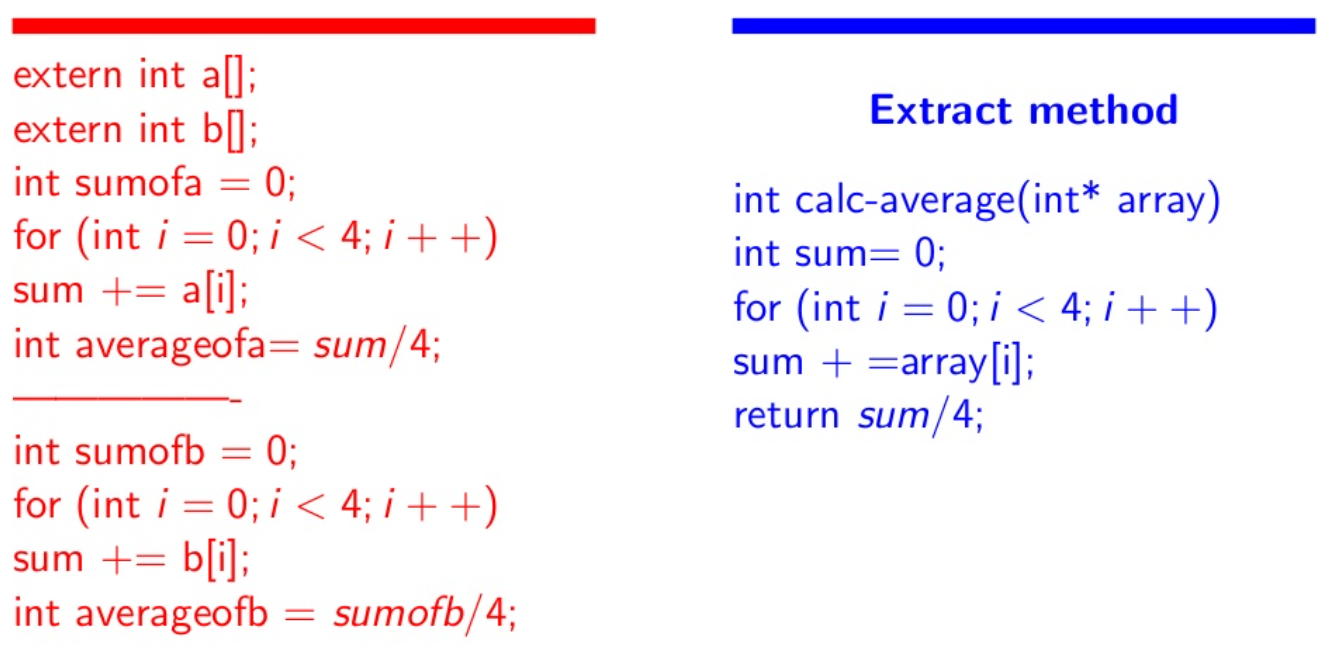
CHAPTER 2 – BAD SMELL OF CODE IN DETAILS

1. 1. **Duplicated Code**
      1. **Definition**

* Duplicated code is the number 1 in the stink parade.
* Duplication usually occurs when multiple programmers are working on different parts of the same program at the same time. Since they are working on different tasks, they may be unaware their colleague has already written similar code that could be repurposed for their own needs.
* There is also more subtle duplication, when specific parts of code look different but actually perform the same job. This kind of duplication can be hard to find and fix.
  + 1. **Solutions**
* If the same code is found in two subclasses of the same level:
  + Use Extract Method for both classes, followed by Pull Up Field for the fields used in the method that developers are pulling up.
  + If the duplicate code is inside a constructor, use Pull Up Constructor Body.
  + If the duplicate code is similar but not completely identical, use Form Template Method.
  + If two methods do the same thing but use different algorithms, select the best algorithm and apply Substitute Algorithm.
* If duplicate code is found in two different classes:
  + If the classes are not part of a hierarchy, use Extract Superclass in order to create a single superclass for these classes that maintains all the previous functionality.
  + If it is difficult or impossible to create a superclass, use Extract Class in one class and use the new component in the other.
* If a large number of conditional expressions are present and perform the same code (differing only in their conditions), merge these operators into a single condition using Consolidate Conditional Expression and use Extract Method to place the condition in a separate method with an easy-to-understand name.

If the same code is performed in all branches of a conditional expression: place the identical code outside of the condition tree by using Consolidate Duplicate Conditional Fragments

* + 1. **Example**



*Fig 2.1.3.1 Example 1 of code duplication*

Extract method will give source code that has no loop duplication

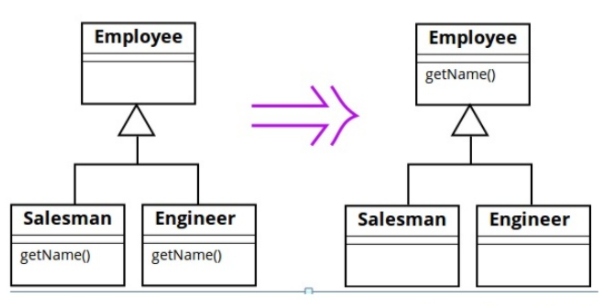
extern int a[];

extern int b[];

int avereageofa = calcAverage(a[]);

int avereageofb = calcAverage(b[]);

* Another problem is when we have same code in two subclasses.
* Solution: (Extract method + pull up method) extract similar codes frim both the classes inform a method and then put this method in the superclass.



*Fig 2.1.3.2 Example 2 of code duplication*

* 1. Long Method

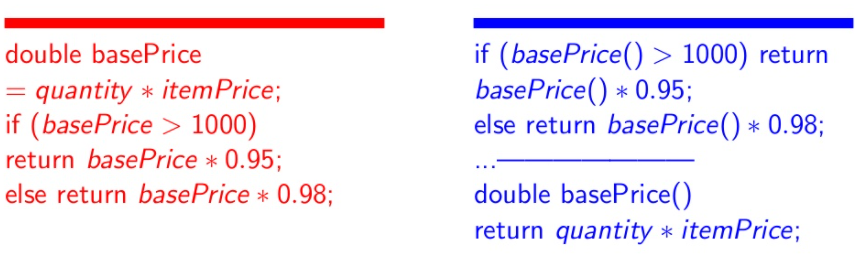
1. Definition

* Long method is a method contains too many lines of code. Generally, any method longer than ten lines should make programmers start asking questions.
* The longer procedure is, the more difficult it is to understand what the code does
  + More difficult to read.
  + Bad for maintainability.
  + More difficult to make modifications.

1. Solutions

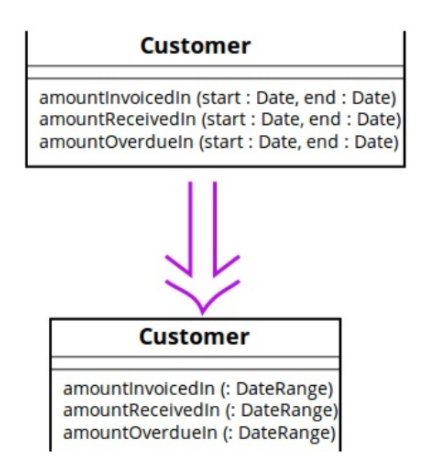
* To reduce the length of a method body, use Extract Method.
* If local variables and parameters interfere with extracting a method, use Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object.
* If none of the previous recipes help, try moving the entire method to a separate object via Replace Method with Method Object.
* Conditional operators and loops are a good clue that code can be moved to a separate method. For conditionals, use Decompose Conditional. If loops are in the way, try Extract Method.

1. Example



*Fig 2.2.3.1 Example of replace temp with Query*

With figure 1, we can see the method

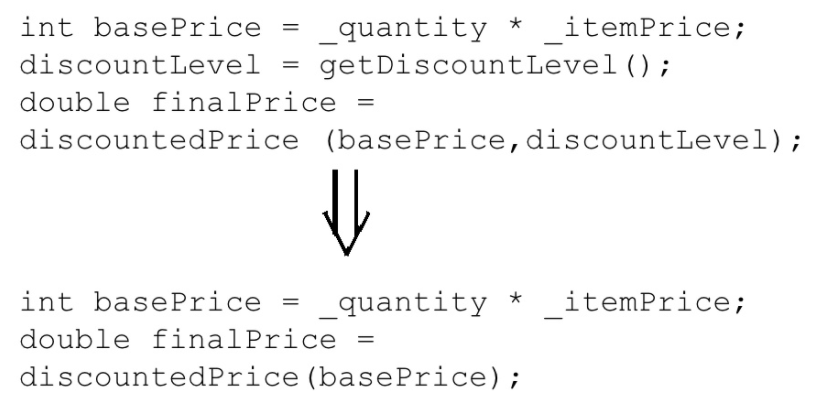


*Fig 2.2.3.2 Example of New parameter object*

* 1. Large Class
     1. Definition
* When a single class is doing to much, it often shows up too many variables and instances
* Basically, a class having many fields/methods/lines of code is called as large class
* 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.
  + 1. Solutions
* 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 is necessary to have a list of the operations and behaviors that the client can use.
* If a large class is responsible for the graphical interface, you may try to move some of its data and behavior to a separate domain object. In doing so, it may be necessary to store copies of some data in two places and keep the data consistent. Duplicate Observed Data offers a way to do this.
  + 1. Example
  1. Long Parameter List
     1. Definition
* Long parameter list is a method call requires passing long list of parameters or we can say that there are more than three or four parameters for a method.
* 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.
  + 1. Solutions

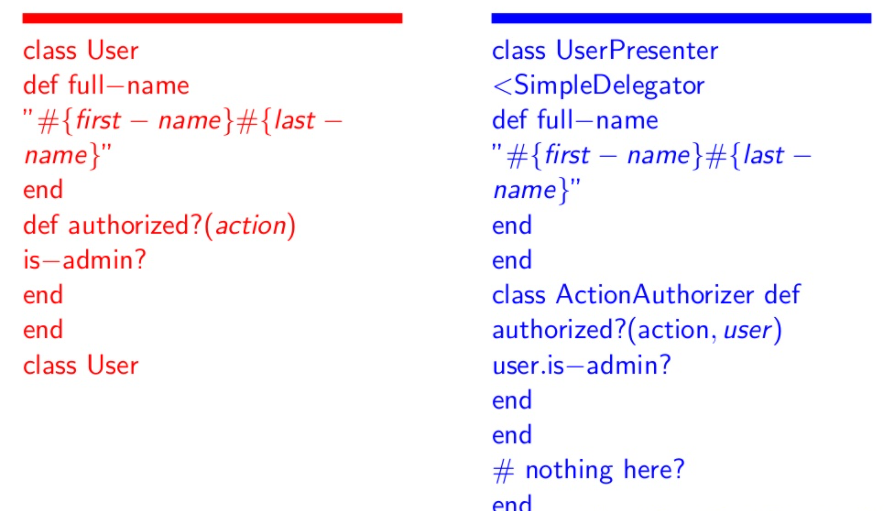
Check what values are passed to parameters. If some of the arguments are just results of method calls of another object, use Replace Parameter with Method Call. This object can be placed in the field of its own class or passed as a method parameter.

* 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.
* If there are several unrelated data elements, sometimes you can merge them into a single parameter object via Introduce Parameter Object.
  + 1. Example



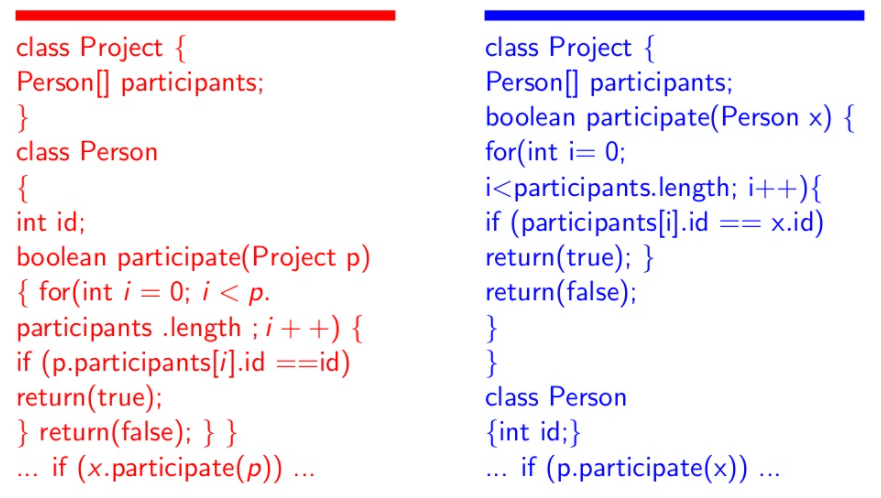
*Fig 2.4.3.1 Example of replace parameter with method*

* 1. Divergent Change
     1. Definition
* Divergent change occurs when one class is commonly change in different ways for different reasons.
* Often these divergent modifications are due to poor program structure or "copy-paste programming”.
  + 1. Solutions
* 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).
  + 1. Example



*Fig 2.5.3.1 Example of Divergent Change*

* 1. Shotgun Surgery
     1. Definition
* Shotgun Surgery resembles Divergent Change but is actually the opposite smell. Divergent Change is when many changes are made to a single class. Shotgun Surgery refers to when a single change is made to multiple classes simultaneously.
* When changes are all over the place, they are hard to find, and it is easy to miss an important change.
  + 1. Solutions
* Use Move Method and Move Field to move existing class behaviors into a single class. If there is 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 via Inline Class.
  + 1. Example

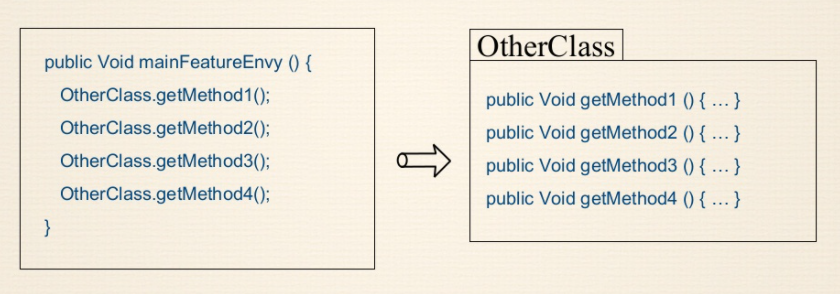


*Fig 2.6.3.1 Example of Move Method*

* 1. Feature Envy
     1. Definition
* A method accesses the data of another object more than its own data.
* 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.
  + 1. Solutions

As a basic rule, if things change at the same time, you should keep them in the same place. Usually, data and functions that use this data are changed together (although exceptions are possible).

* 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.
  + 1. Example



*Fig 2.7.3.1 Example of Feature Envy*

* + - Solution: Move Method

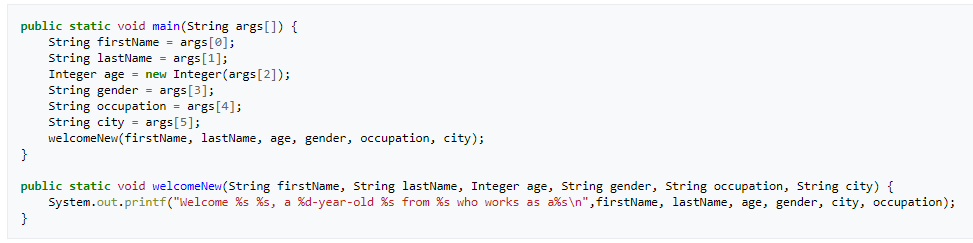


*Fig 2.7.3.2 Example of Move Method*

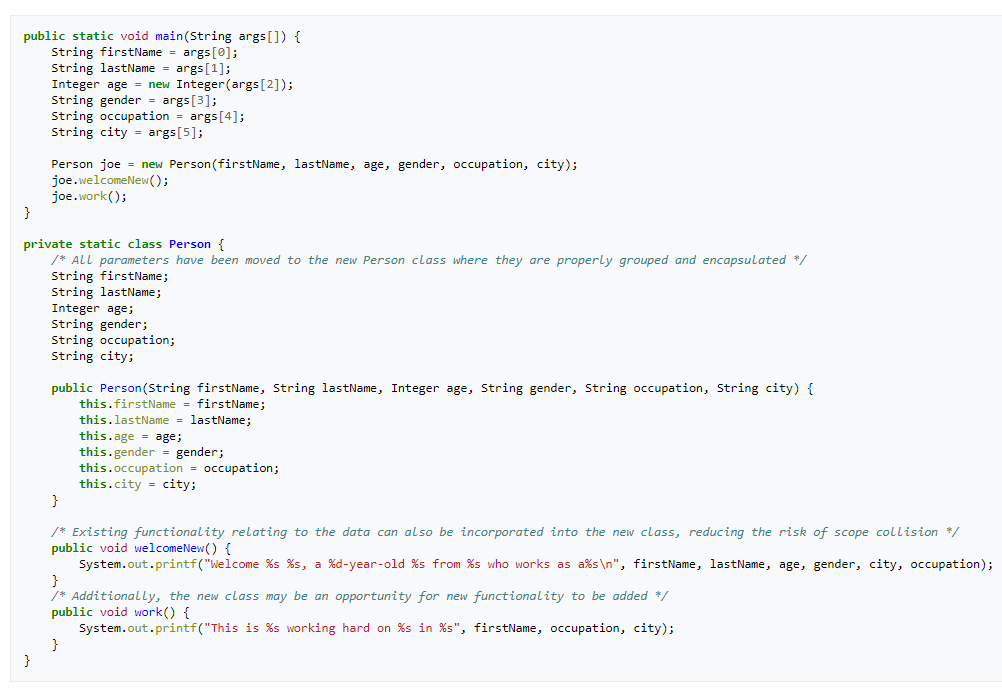
* 1. Data Clumps
     1. Definition

In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), "data clump" is a name given to any group of [variables](https://en.wikipedia.org/wiki/Variable_(computer_science)) which are passed around together (in a clump) throughout various parts of the program. A data clump, like other [code smells](https://en.wikipedia.org/wiki/Code_smell), can indicate deeper problems with the program design or implementation. The group of variables that typically make up a data clump are often closely related or interdependent and are often used together in a group as a result. A data clump is also known as a specific kind of class-level [code smell](https://en.wikipedia.org/wiki/Code_smell) that may be a symptom of poorly written source code.

* + 1. Solutions
* If repeating data comprises the fields of a class, use [Extract Class](https://sourcemaking.com/refactoring/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](https://sourcemaking.com/refactoring/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](https://sourcemaking.com/refactoring/preserve-whole-object) will help with this.
* Look at the code used by these fields. It may be a good idea to move this code to a data class.
  + 1. Example



Solution => Make it more specific with those parameters



* 1. Primitive Obsession
     1. Definition
* 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.)
* Use of string constants as field names for use in data arrays.
  + 1. Solutions
* If the values of primitive fields are used in method parameters, go with [Introduce Parameter Object](https://sourcemaking.com/refactoring/introduce-parameter-object) or [Preserve Whole Object](https://sourcemaking.com/refactoring/preserve-whole-object).
* When complicated data is coded in variables, use [Replace Type Code with Class](https://sourcemaking.com/refactoring/replace-type-code-with-class), [Replace Type Code with Subclasses](https://sourcemaking.com/refactoring/replace-type-code-with-subclasses) or [Replace Type Code with State/Strategy](https://sourcemaking.com/refactoring/replace-type-code-with-state-strategy).
* If there are arrays among the variables, use [Replace Array with Object](https://sourcemaking.com/refactoring/replace-array-with-object).
  + 1. Examples

This is the example code of using the Bad Smell: Primitive Obsession:



Solution to fix it?

The best way is to start from the beginning, with a small step incremental, not huge leap of code.

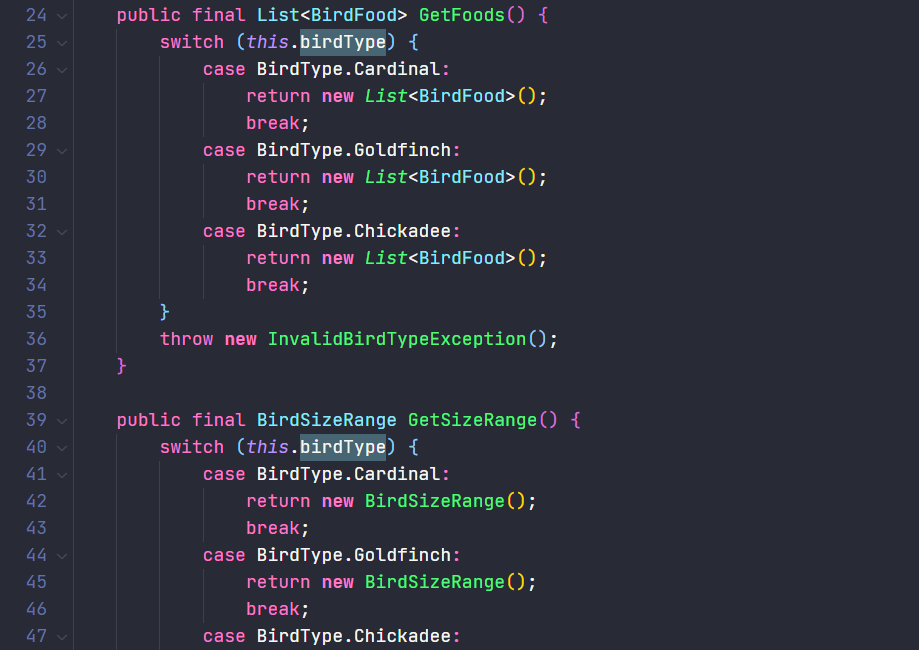
Pushing the Arguments out

* 1. Switch Statements
     1. Definition

It simply means that you have a complex switch operator or sequence of if statements.

* + 1. Solutions
* 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 are not 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.
  + 1. Examples

Below here are the example of using the switch statement incorrectly and make the code more confused even for the developers



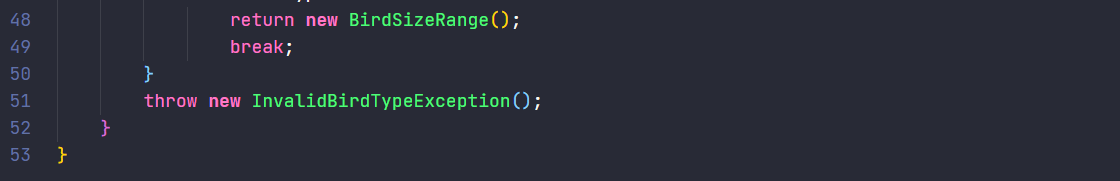


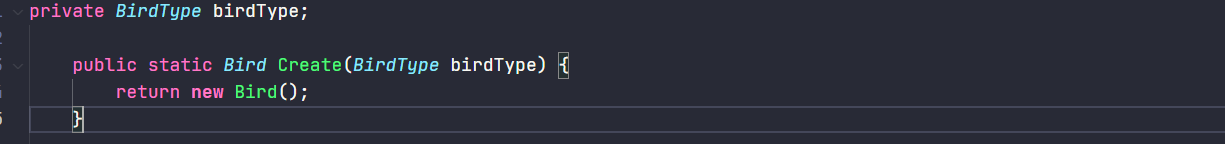
Figure 1: Code Smell Switch Statements

As you can see from the code perspective, it may look long and confused, dig down deeper is only switch and case statement use inside class Bird. How do we approach fixing it?

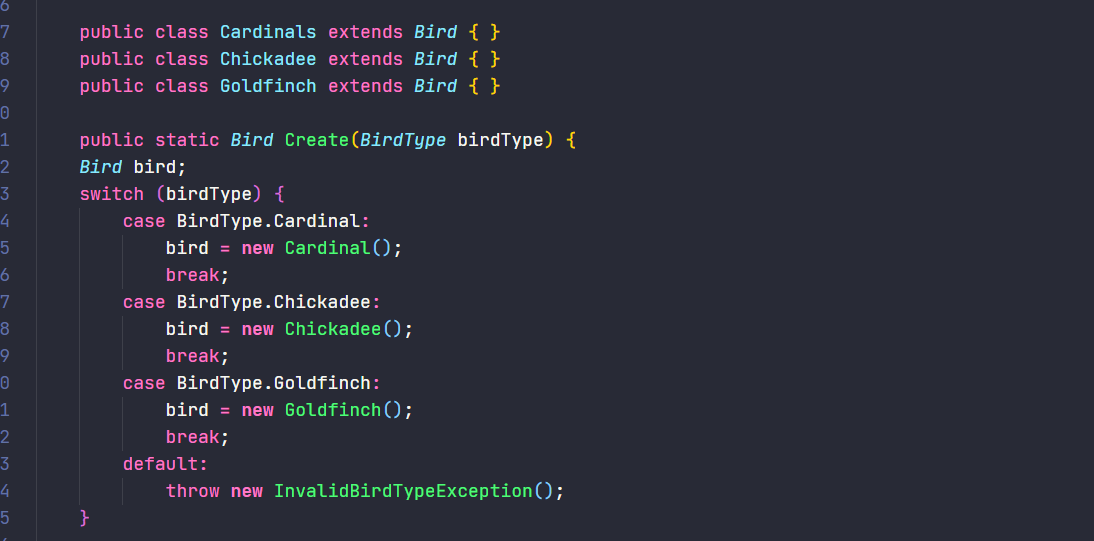
One of the best methods is to use Factory Method

First, we need to replace the constructor with Factory Method:

* Delete Bird() constructor.
* Add static method Create(BirdType).
* . Make the birdType field private. Note: We need to keep this field around until the end because it’s used in all of the methods, and we’ll be refactoring the methods one at a time.



**Second, create a Bird subclass for each type of bird specified by BirdType**

****

Same for all the later method that are currently available.

* 1. Parallel Inheritance Hierarchies
     1. Definition

Parallel Inheritance Hierarchies occur when an inheritance tree depends on another inheritance tree by composition, and they maintain a special relationship where one subclass of a dependent inheritance must depend on one a particular subclass of another Inheritance.

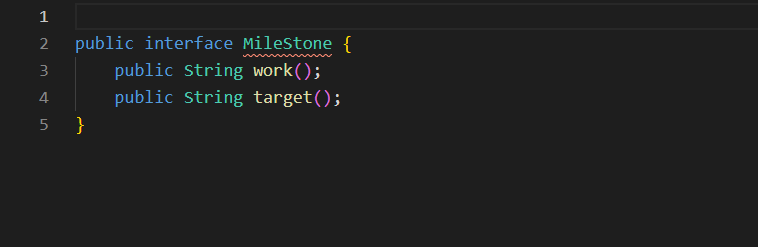
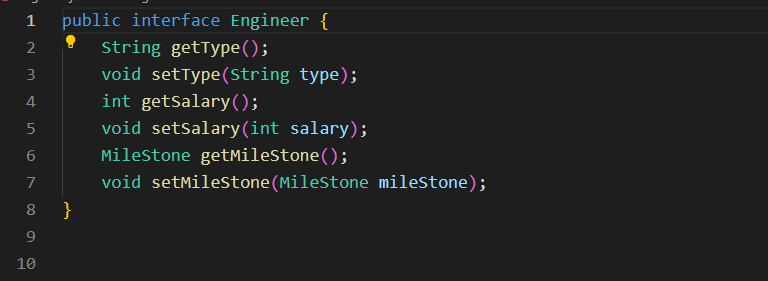
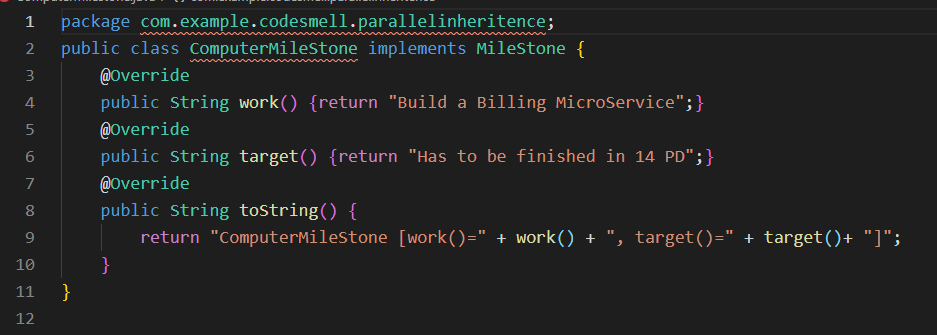
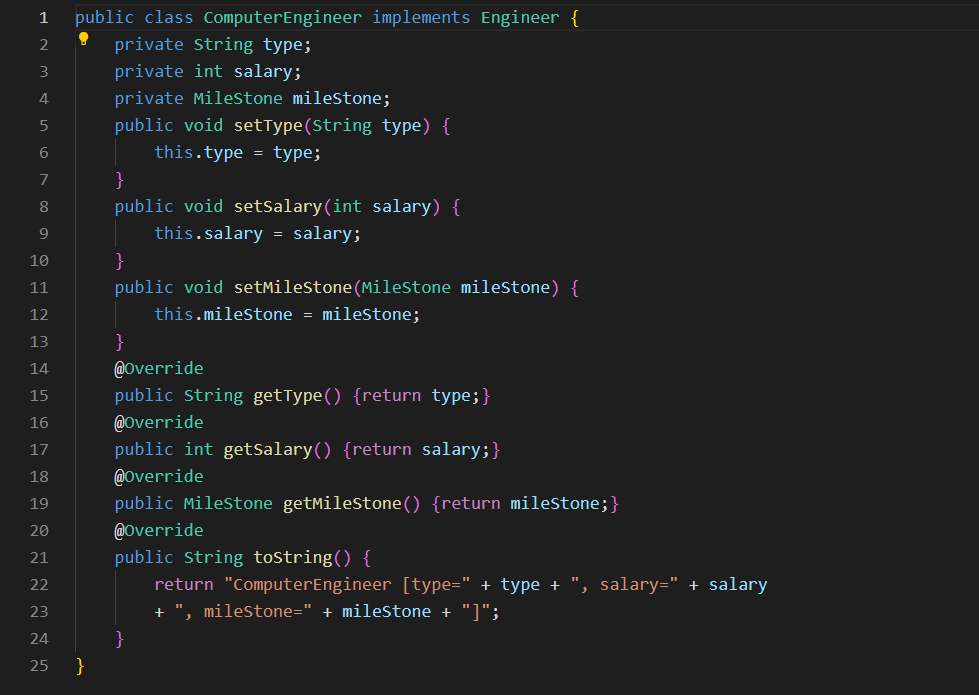
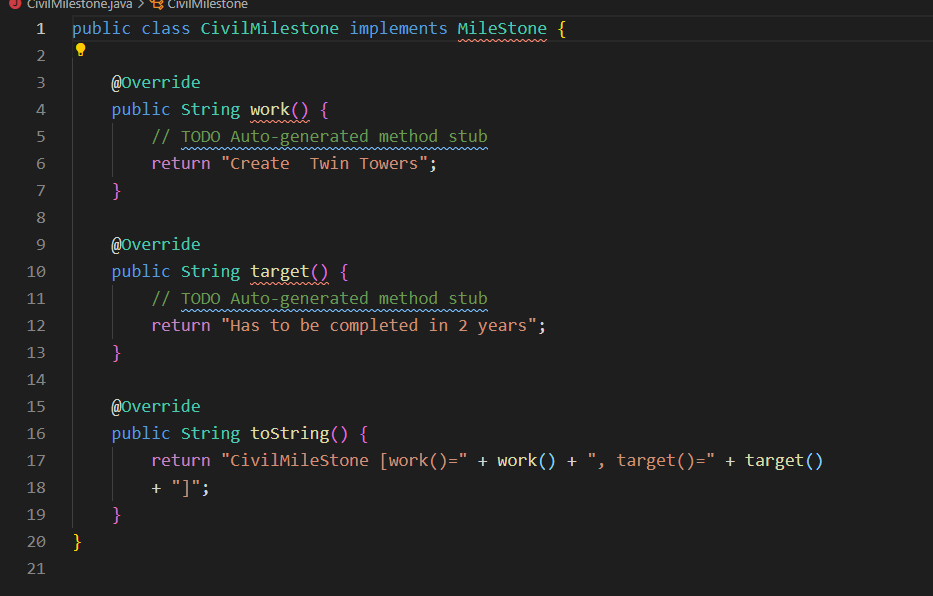
* + 1. Solutions

You may de-duplicate 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.

* + 1. Examples

Right now, I’ll show you the project name, it’s called ComputerEngineering()





As you can see with the code, the inheritances of each class is being used multiples times, creating a massive file count in the folder, as you can see with figure 2

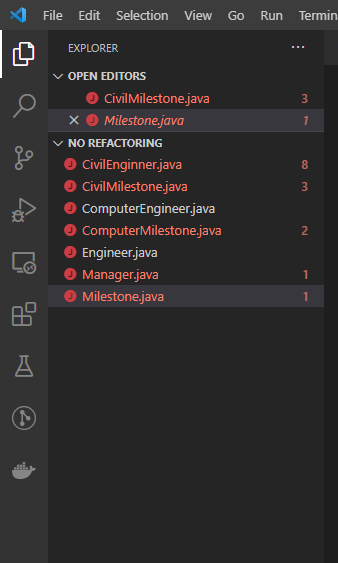
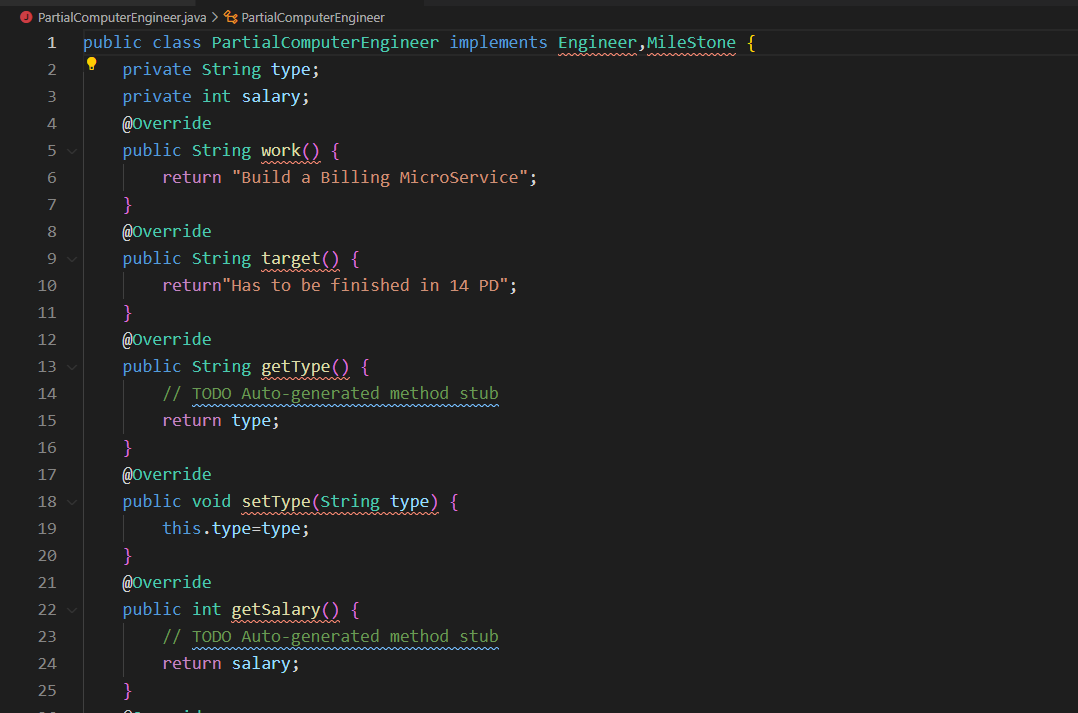


Figure 2: File that are associate with the this project.

The solution is to create one main interface for all the class that are related, i.e., Computer have Milestone and Engineer, so as Civil, so we just create a Milestone and Engineer, and from there we can implements to each class for better readability.

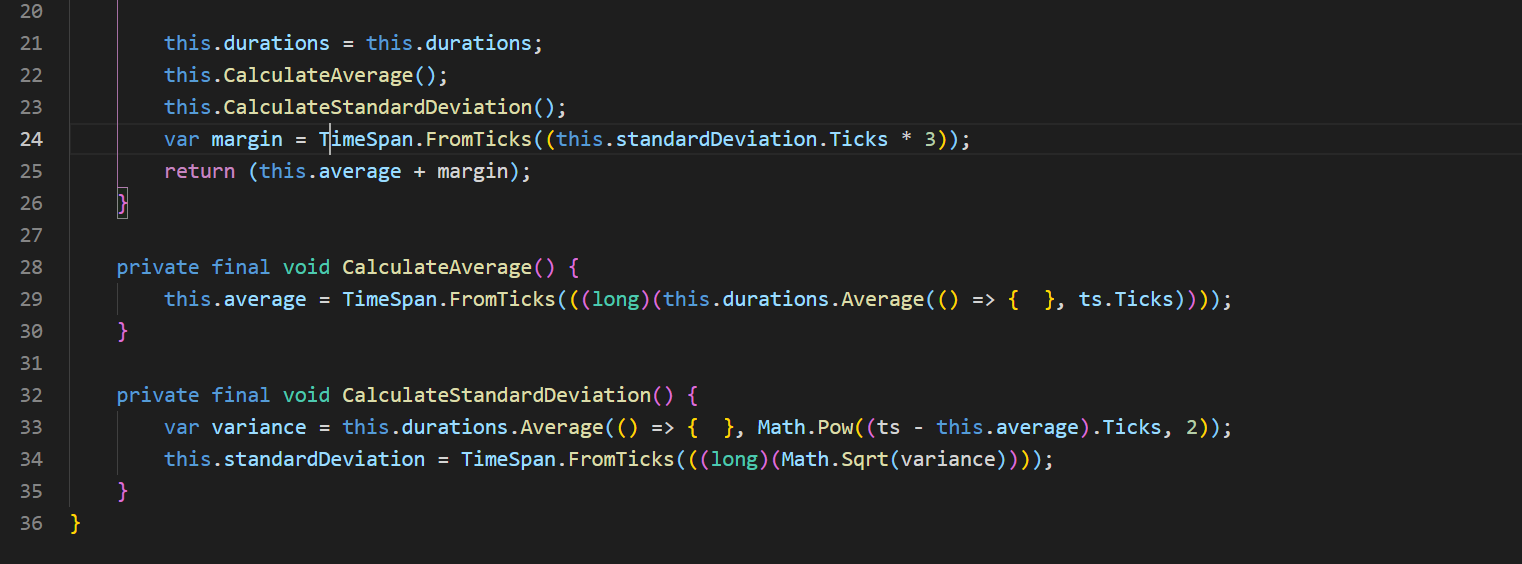
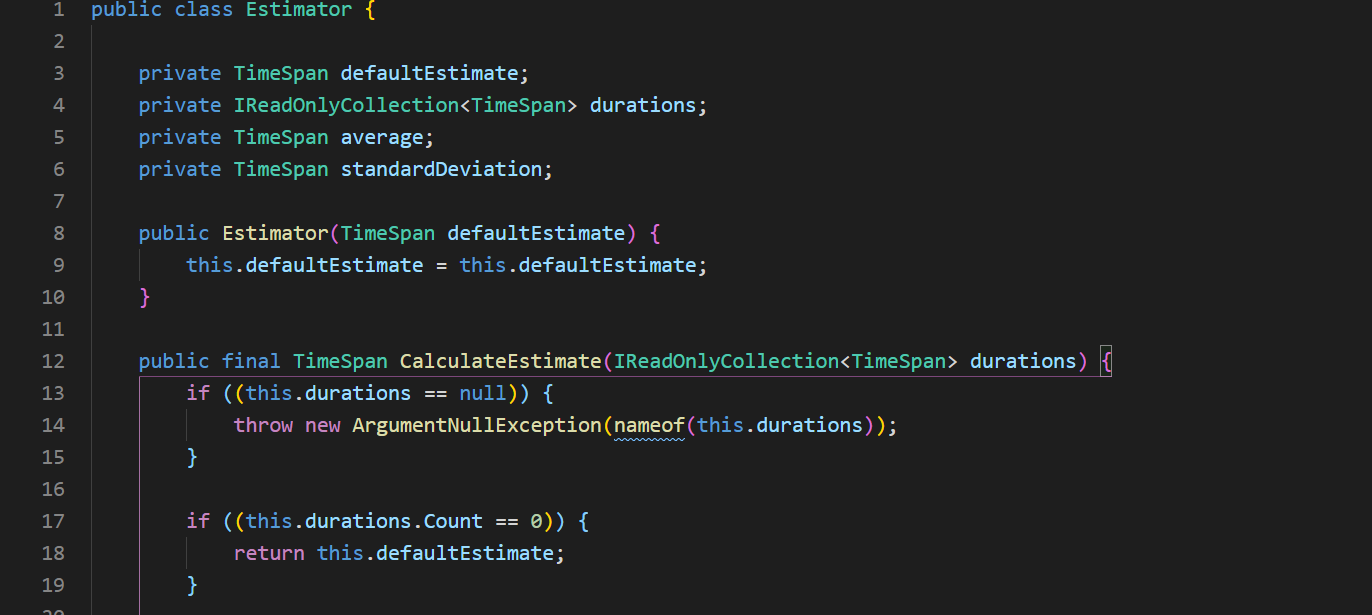
Code example for refactored:



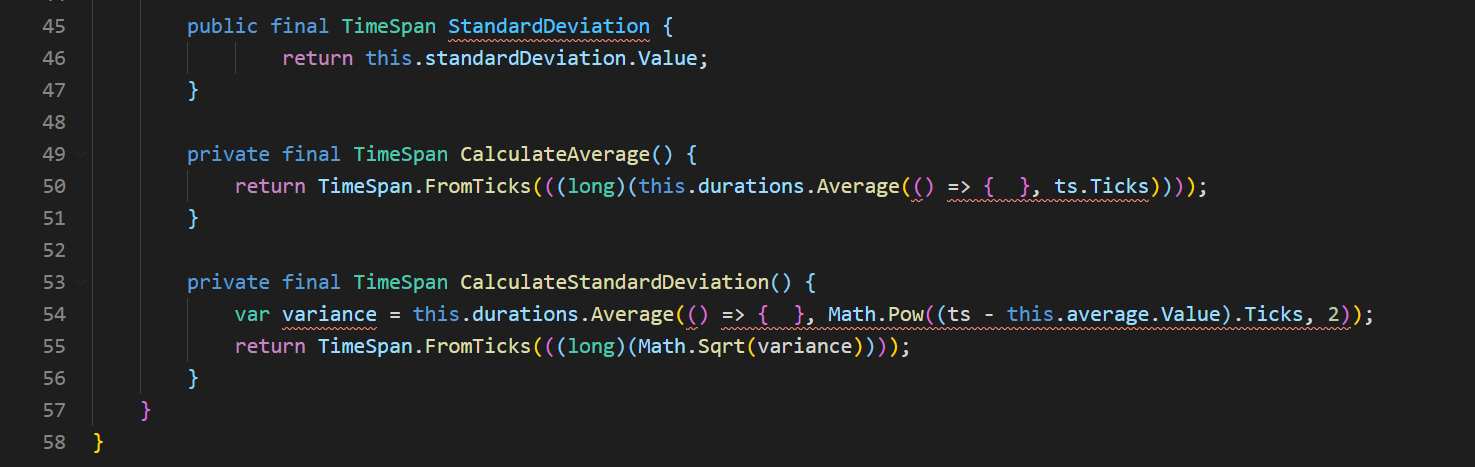
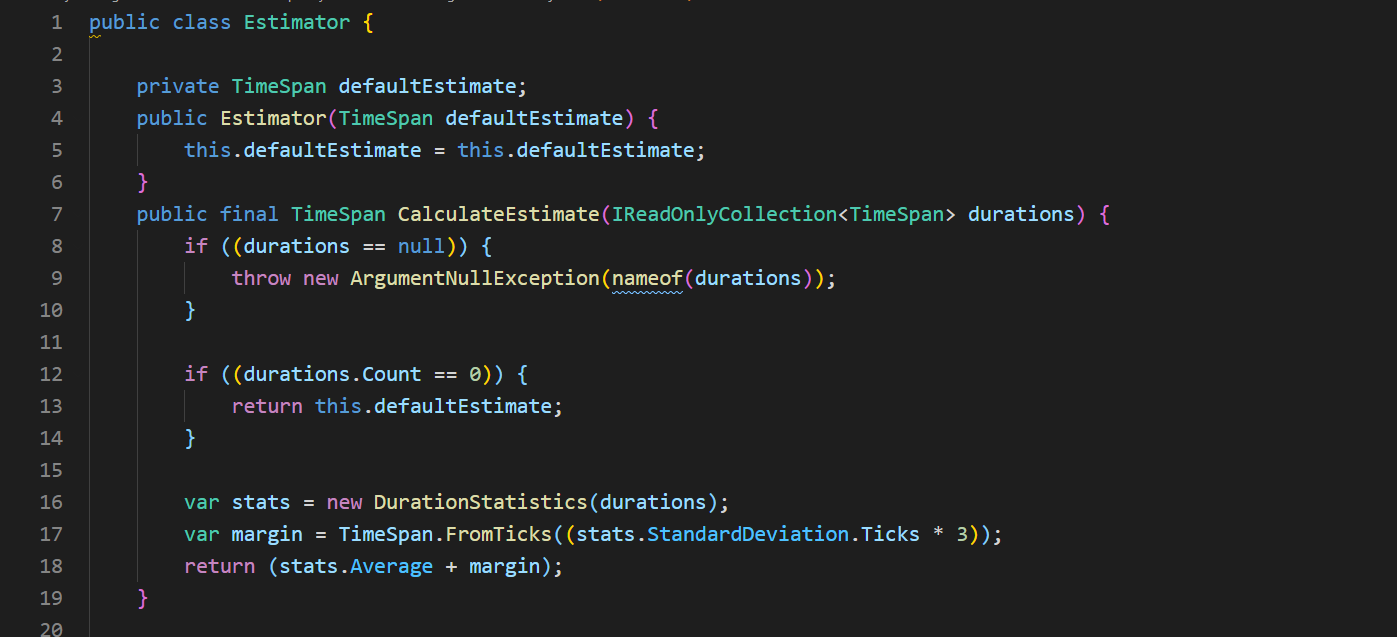
* 1. Lazy Class
     1. Definitions
* Understanding and maintaining classes always costs time and money. So, if a class doesn’t do enough to earn your attention, it should be deleted.
  + 1. Solutions
* Components that are near-useless should be given the [Inline Class](https://sourcemaking.com/refactoring/inline-class) treatment.
* For subclasses with few functions, try [Collapse Hierarchy](https://sourcemaking.com/refactoring/collapse-hierarchy).
  + 1. Examples
  1. Speculative Generality
     1. Definition

There are a lot of unused class, unused method field or parameters in the code. Input but never used

* + 1. Solutions
* For removing unused abstract classes, try Collapse Hierarchy.
* Unnecessary delegation of functionality to another class can be eliminated via Inline Class.
* Unused methods? Use Inline Method to get rid of them.
* Methods with unused parameters should be given a look with the help of Remove Parameter.
* Unused fields can be simply deleted.
  + 1. Examples
  1. Temporary Field
     1. Definitions
* Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they are empty.
* This kind of code is tough to understand. You expect to see data in object fields but for some reason they are almost always empty.
  + 1. Solutions
* Temporary fields and all code operating on them can be put in a separate class via Extract Class. In other words, you are 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.
  + 1. Examples



Solution are:



* 1. Message Chains
     1. Definition

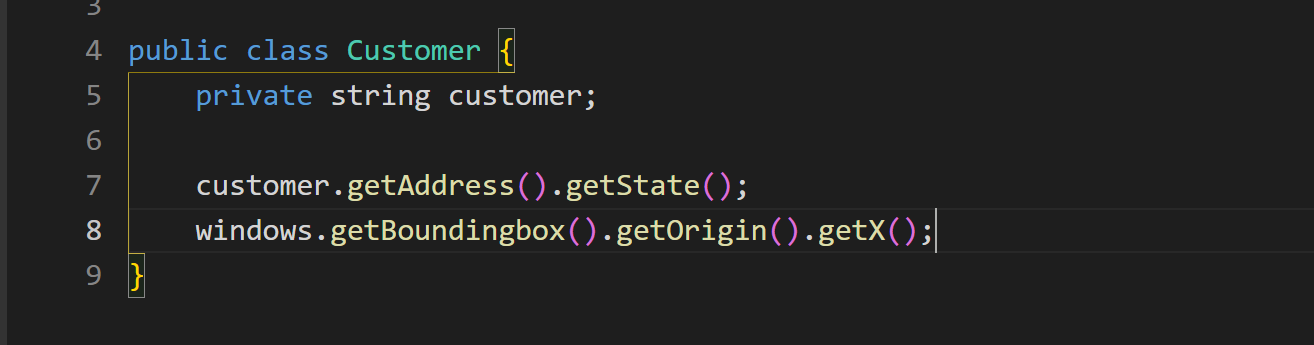
In code you see a series of calls resembling $a->b()->c()->d()

* + 1. Solutions

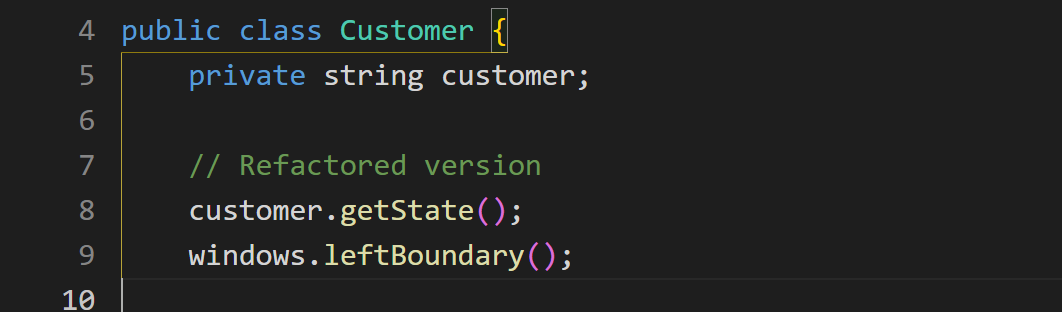
- To delete a message chain, use Hide Delegate.

Sometimes it is 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.

* + 1. Examples



Solution to fix this is, change to 1 variable with only one instruction



* 1. Middle Man
     1. Definitions

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

* + 1. Solutions
* If most of a method’s classes delegate to another class, [Remove Middle Man](https://sourcemaking.com/refactoring/remove-middle-man) is in order.
  + 1. Examples
  1. Alternative Classes with Different Interfaces
  2. Incomplete Library Class
  3. Data Class
     1. Definition:

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 do not contain any additional functionality and cannot independently operate on the data that they own.

* + 1. Solutions
* 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 is used by the class. In it, you may find functionality that would be better located in the data class itself. If this is the case, use Move Method and Extract Method to migrate this functionality to the data class.
  + 1. Examples

* 1. Refused Bequest

REFERENCES

Vietnamese

English

1. <https://ehsangazar.com/refactoring-bad-smells-in-code-8c8eccad9833?gi=a4f6ac7d5b3d> (04/03/2021)
2. <https://sourcemaking.com/refactoring/smells/duplicate-code> (04/03/2021)
3. <https://www.slideshare.net/annuvinayak/code-smells-and-its-type-with-example> (06/03/2021)
4. GeeksforGeeks. 2020. *AVL Tree | Set 1 (Insertion) - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/avl-tree-set-1-insertion/> [Accessed 14 May 2020].
5. CodesDope. 2020. *Red Black Trees: Rotations And Insertions*. [online] Available at: <https://www.codesdope.com/course/data-structures-red-black-trees-insertion/> [Accessed 14 May 2020].
6. Brilliant.org. 2020. *Red-Black Tree | Brilliant Math & Science Wiki*. [online] Available at: <https://brilliant.org/wiki/red-black-tree/> [Accessed 15 May 2020].
7. Programiz.com. 2020. *Red-Black Tree*. [online] Available at: <https://www.programiz.com/dsa/red-black-tree> [Accessed 14 May 2020].
8. Roy, T., 2020. *Red Black Tree Insertion*. [online] Available at: <https://www.youtube.com/watch?v=UaLIHuR1t8Q&t=497s> [Accessed 14 May 2020].
9. Brilliant.org. 2020. *Red-Black Tree | Brilliant Math & Science Wiki*. [online] Available at: <https://brilliant.org/wiki/red-black-tree/> [Accessed 21 May 2020].
10. Weiss, M., 2014. *Data Structures And Algorithm Analysis In C++*. 4th ed. Harlow: Pearson, pp.168-173.
11. Cormen, T., 2009. *Introduction To Algorithms*. 3rd ed. Cambridge, Mass.: MIT Press, pp.308-329 and 491-501.
12. Goodrich, M., Tamassia, R. and Goldwasser, M., 2015. *Data Structures And Algorithms In Java TM*. 6th ed. Hoboken, NJ: Wiley & Sons Singapore Pte, pp.510-525.
13. Mallawaarachchi, V., 2020. *All You Need To Know About Deleting Keys From B Trees*. [online] Medium. Available at: <https://medium.com/@vijinimallawaarachchi/all-you-need-to-know-about-deleting-keys-from-b-trees-9090f3334b5c> [Accessed 21 May 2020].

SELF-EVALUATION

| **Requirements** | **Score /10** | **Level 1** | **Level 2** | **Level 3** | **Self-evaluation** | **Reason(s)** |
| --- | --- | --- | --- | --- | --- | --- |
| **0 score** | **1/2 score** | **Full score** |  |  |
| **1/ Report** | **8.0** |  |  |  |  |  |
| In right format | 1.0 | Wrong format and outlines | Some errors | In right format and outlines, no error | 1.0 |  |
| Chapter 1 | 1.0 | Not enough content, bad written, no example | Full contents, not very well written, not enough examples | Full contents, well written, with examples | 1.0 |  |
| Chapter 2 | 2.0 | Not enough content, bad written, no example | Full contents, not very well written, not enough examples | Full contents, well written, with examples | 2.0 |  |
| Chapter 3 | 2.0 | Not enough content, bad written, no example | Full contents, not very well written, not enough examples | Full contents, well written, with examples | 1.0 |  |
| Chapter 4 | 1.0 | Not enough content, bad written, no example | Full contents, not very well written, not enough examples | Full contents, well written, with examples | 0.5 |  |
| Chapter 5 | 0.5 | Not enough content, bad written | Full contents, not very well written | Full contents, well written | 0.5 |  |
| References | 0.5 | No reference | Wrong format, < 3 references | Right format, ≥ 3 references | 0.5 |  |
| **2/ Demo** | **2.0** |  |  |  |  |  |
| Contents | 1.5 | Implement less than half of the operations | Implement half of the operations | Implement all of the operations | 1.5 |  |
| Program | 0.5 | Cannot be compiled | Runtime error for 1 formula or algorithm | Can be run correctly with no error | 0.5 |  |
| **Total** | 10.0 | Result: | | | 8.50 |  |