**Code Refactoring: Code Smells**

# **Part 1: Code Smells and Refactoring Patterns**

## ***Code Smells***

1. Duplicated Code
2. Divergent Change
3. Parallel Inheritance Hierarchies

## ***Refactoring Patterns***

Duplicated Code – Extract Method

*Turn the fragment into a method whose name explains the purpose of the method.* (Fowler & Beck, 2013)

The basic idea of extract method is to identify code of certain “domain” and encapsulate that code into its own method. Identifying the domain of the code requires some practice.

|  |  |
| --- | --- |
| Before Refactoring | After Refactoring |
| 0 | **def** calculate\_salary(employee):  1 |     gross\_salary = employee["baseSalary"] \* 12  2 |     bonus = employee.get("bonus", 0)  3 |     commission = employee.get("commission", 0)  4 |     total\_salary = gross\_salary + bonus + commission  5 |     **return** total\_salary  6 | ​  7 | employee = {  8 |     "name": "John Doe",  9 |     "baseSalary": 5000,  10|     "bonus": 2000,  11|     "commission": 1000  12| }  13| print(calculate\_salary(employee)) | 1 | **def** calculate\_gross\_salary(employee):  2 |     gross\_salary = employee["baseSalary"] \* 12  3 |     **return** gross\_salary  4 | ​  5 | **def** calculate\_salary(employee):  6 |     gross\_salary = calculate\_gross\_salary(employee)  7 |     bonus = employee.get("bonus", 0)  8 |     commission = employee.get("commission", 0)  9 |     total\_salary = gross\_salary + bonus + commission  10|     **return** total\_salary  11| ​  12| employee = {  13|     "name": "John Doe",  14|     "baseSalary": 5000,  15|     "bonus": 2000,  16|     "commission": 1000  17| }  18| print(calculate\_salary(employee)) |

Divergent Change – Extract Class

*You have one class doing work that should be done by two.* (Fowler & Beck, 2013)

Some refactoring techniques introduce code instead of removing it. The extract class technique might introduce more overhead than the extract method technique. In the following example, the author of this text introduces some basic Objects.

|  |  |
| --- | --- |
| Before Refactoring | After Refactoring |
| 1 | **class** Employee:  2 |     **def** \_\_init\_\_(self, name, department, salary):  3 |         self.name = name  4 |         self.department = department  5 |         self.salary = salary  6 |         self.bonus = 0  7 | ​  8 |     **def** calculate\_total\_compensation(self):  9 |         **return** self.salary + self.bonus  10| ​  11| employee1 = Employee("John Doe", "Sales", 5000)  12| employee2 = Employee("Jane Smith", "Marketing", 6000)  13| ​  14| print(employee1.calculate\_total\_compensation())  15| print(employee2.calculate\_total\_compensation()) | 1 |  2 | **class** Bonus:  3 |     **def** \_\_init\_\_(self, department):  4 |         self.department = department  5 | ​  6 |     **def** calculate\_bonus(self):  7 |         **raise** NotImplementedError()  8 | ​  9 | **class** SalesEmployee(Employee):  10|     **def** \_\_init\_\_(self, name, salary):  11|         super().\_\_init\_\_(name, "Sales", salary)  12| ​  13|     **class** Bonus(Bonus):  14|         **def** calculate\_bonus(self):  15|             **return** 1000  16| ​  17| **class** MarketingEmployee(Employee):  18|     **def** \_\_init\_\_(self, name, salary):  19|         super().\_\_init\_\_(name, "Marketing", salary)  20| ​  21|     **class** Bonus(Bonus):  22|         **def** calculate\_bonus(self):  23| ​  24|             **return** 2000  25| ​  26| sales\_employee = SalesEmployee("John Doe", 5000)  27| marketing\_employee = MarketingEmployee("Jane Smith", 6000)  28| ​  29| print(sales\_employee.calculate\_total\_compensation())  30| print(marketing\_employee.calculate\_total\_compensation()) |

Parallel Inheritance Hierarchies – Move Method

*Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation or remove it altogether.* (Fowler & Beck, 2013)

The Move method technique is very similar to the Extract Method. In the following contrived example, we should use this method by moving the "Complex" area calculation into its own space. This new space can be in a new object, module, or service. In real-world scenarios, we often have simple methods that require a lot of testing around them (e.g., date conversion) or long methods that require lengthy implementation with simple testing around them (like adding specific numbers to a particular currency and rounding behavior). The technique would be the same: move these behaviors to another place so we can have more eyes around them.

|  |  |
| --- | --- |
| Before Refactoring | After Refactoring |
| 1 | **class** Shape:  2 |     **def** \_\_init\_\_(self, color):  3 |         self.color = color  4 | ​  5 | **class** Rectangle(Shape):  6 |     **def** \_\_init\_\_(self, width, height, color="red"):  7 |         super().\_\_init\_\_(color)  8 |         self.width = width  9 |         self.height = height  10| ​  11|     **def** area(self):  12|         **return** self.width \* self.height  13| ​  14| **class** Circle(Shape):  15|     **def** \_\_init\_\_(self, radius, color="red"):  16|         super().\_\_init\_\_(color)  17|         self.radius = radius  18| ​  19|     **def** area(self):  20|         **return** 3.14 \* self.radius \*\* 2  21| ​  22| rectangle = Rectangle(5, 6)  23| circle = Circle(3)  24| ​  25| print("Rectangle area:", rectangle.area())  26| print("Circle area:", circle.area())  27| ​ | 1 | **import** math  2 | **import** shapes  3 | ​  4 | **def** area\_rectangle(width, height):  5 |     **return** width \* height  6 | ​  7 | **def** area\_circle(radius):  8 |     **return** 3.14 \* radius \*\* 2  9 | ​  10| **class** Shape:  11|     **def** \_\_init\_\_(self, color):  12|         self.color = color  13| ​  14| **class** Rectangle(Shape):  15|     **def** \_\_init\_\_(self, width, height, color="red"):  16|         super().\_\_init\_\_(color)  17|         self.width = width  18|         self.height = height  19| ​  20|     **def** area(self):  21|         **return** shapes.area\_rectangle(self.width, self.height)  22| ​  23| **class** Circle(Shape):  24|     **def** \_\_init\_\_(self, radius, color="red"):  25|         super().\_\_init\_\_(color)  26|         self.radius = radius  27| ​  28|     **def** area(self):  29|         **return** shapes.area\_circle(self.radius)  30| ​  31| rectangle = Rectangle(5, 6)  32| circle = Circle(3)  33| ​  34| print("Rectangle area:", rectangle.area())  35| print("Circle area:", circle.area())  36| ​ |

## ***Code Quality Improvement***

These three refactoring methods all improve a particular area of code quality: internal software quality.

Per the ISO/IEC 25010 quality model, all these patterns greatly help with the maintainability quality characteristic. Extracting or Moving methods help testability and modifiability over time. Extracting Classes helps with the reusability and modularity of critical parts of the software.

In Seriously Good Software, they discuss how these qualities are internal. They also explain that the end user will be able to perceive these qualities, but the boundary between these internal quality issues and something the user can see (e.g., the correctness of the software) is not clear-cut.

Extract Method

The Extract method technique provides significant quality improvements by keeping functions atomic. The previous examples might be too simple to understand the advantages of the Extract Method function. Let's look at a more complex calculation:

1 | **import** cmath

2 | **import** math

3 | ​

4 | **def** fourier\_transform(x):

5 |     n = len(x)

6 |     y = [0] \* n

7 |     **for** k **in** range(n):

8 |         w = cmath.exp(-2j \* math.pi \* k / n)

9 |         term = sum([x[m] \* w \*\* m **for** m **in** range(n)])

10|         y[k] = term

11|     **return** y

12| ​​

The previous code is a simple Fourier calculation in Python from scratch, though it might not be correct as it is not production-ready. Notice how many local variables are required to perform this calculation. We create four local variables to create this calculation. If this calculation were embedded next to other calculations, like, for example, the calculation of the average, detecting noise in the signal, and maybe other statistics like variance, then the number of variables would be so confusing and complicated to discern that people would be afraid even to touch the code.

There is a lot of discussion around when is a function too long. Simon and Schuster, 2021, argue that *no method should have more than five lines*. But while in the book that’s what they call for, it is more of a north star rather than a rule set in stone. They also introduce a new quality improvement achievable by refactoring, Compiler Optimization. *Compiler improvements should mean our code automatically gets faster over time if we write good idiomatic code.* (Clausen, 2021)

Extract Class

Extract Class is an example that is hard to explain in code but becomes apparent in real-world scenarios. In our previous example, we added a class bonus for marketing and sales employees. That example might be contrived because it went from a single line of code to many lines of code. In another real-world scenario, we might have currency behavior. Currency behavior is one of those scenarios in which the behavior starts adding up on the base class. Then, we include rounding clauses depending on currencies and tax options depending on currencies. If all that behavior stays in the same class, we will quickly have a long class that does many things. The extract class would be great in this scenario, allowing us to reduce that complexity by moving Currency behavior and Tax behavior to different classes. The refactor would improve the codebase by moving the complexity to their modules and allowing better audits from people focusing solely on those areas.

Move Method

The move method might sound like the extract class technique, and that's because they both belong to the area of moving features around. The contrived example around shapes and areas is an excellent example. The area calculation of both shapes might be simple enough not to move those methods, and that would be true. Refactors are essential if someone has a valid reason to do so. In Refactoring at Scale, the authors tell us how developer productivity and bug identification are benefits of refactoring. However, there's also a valid risk of expensive regressions, unearthing dormant bugs, scope creep, and the most common among these examples, unnecessary complexity.

The move method is usually straightforward to see, but it is unclear whether "Should I move it now?" Sometimes, we might be on a time crunch to deliver, so making changes that cost extra regression is not feasible. At other times, it might be an ancient area of the code base that is hard to test because it impacts a small part of the customer base. These reasons are why writing tests becomes essential.

# **Part 2: Refactoring Tool and Refactoring Patterns**

## ***Refactoring Tool***

1. Visual Studio Code
2. IntelliJ IDEA
3. Visual Studio

## ***Refactoring Tool Capabilities***

Visual Studio Code

VS Code is a versatile, open-source editor developed by Microsoft. It has some basic capabilities out-of-the-box, but for language specific behavior we will need to install Extensions. The library of extensions is vast with free or paid options available. Here is an example snippet of the base refactor option inside VSCode.

IntelliJ IDEA

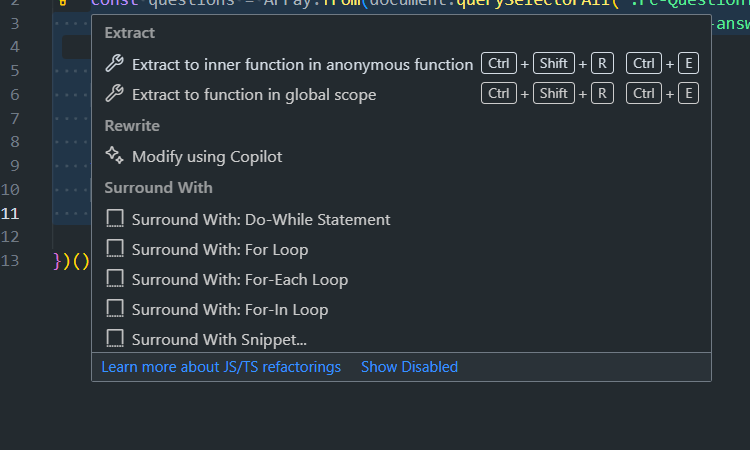
IntelliJ IDEA is a popular integrated development environment (IDE) developed by JetBrains for Java, Kotlin, Groovy, Scala, and other programming languages. It offers advanced features like code completion, intelligent code assistance, on-the-fly error checking, and refactoring capabilities that help developers write high-quality, maintainable, and efficient code. IntelliJ IDEA comes with built-in tools for testing, debugging, and deployment. Its user interface is customizable and highly extensible through plugins, making it a versatile and powerful tool for developers in various industries and domains.

Visual Studio

Visual Studio is a comprehensive integrated development environment (IDE) developed by Microsoft for building various applications, including web, desktop, mobile, cloud, and games. It offers advanced features like IntelliSense, which provides intelligent completions based on context and code patterns; debugging tools that help identify and fix issues in real-time; version control integration; and a rich ecosystem of extensions for various languages, frameworks, and tools. Visual Studio supports programming languages like C#, F#, VB.NET, Python, JavaScript, TypeScript, and more. It offers accessible features for building, testing, debugging, and deploying applications, making it a powerful tool for developers in industries ranging from software development to gaming and scientific research.

## ***Addressing Refactoring Patterns***

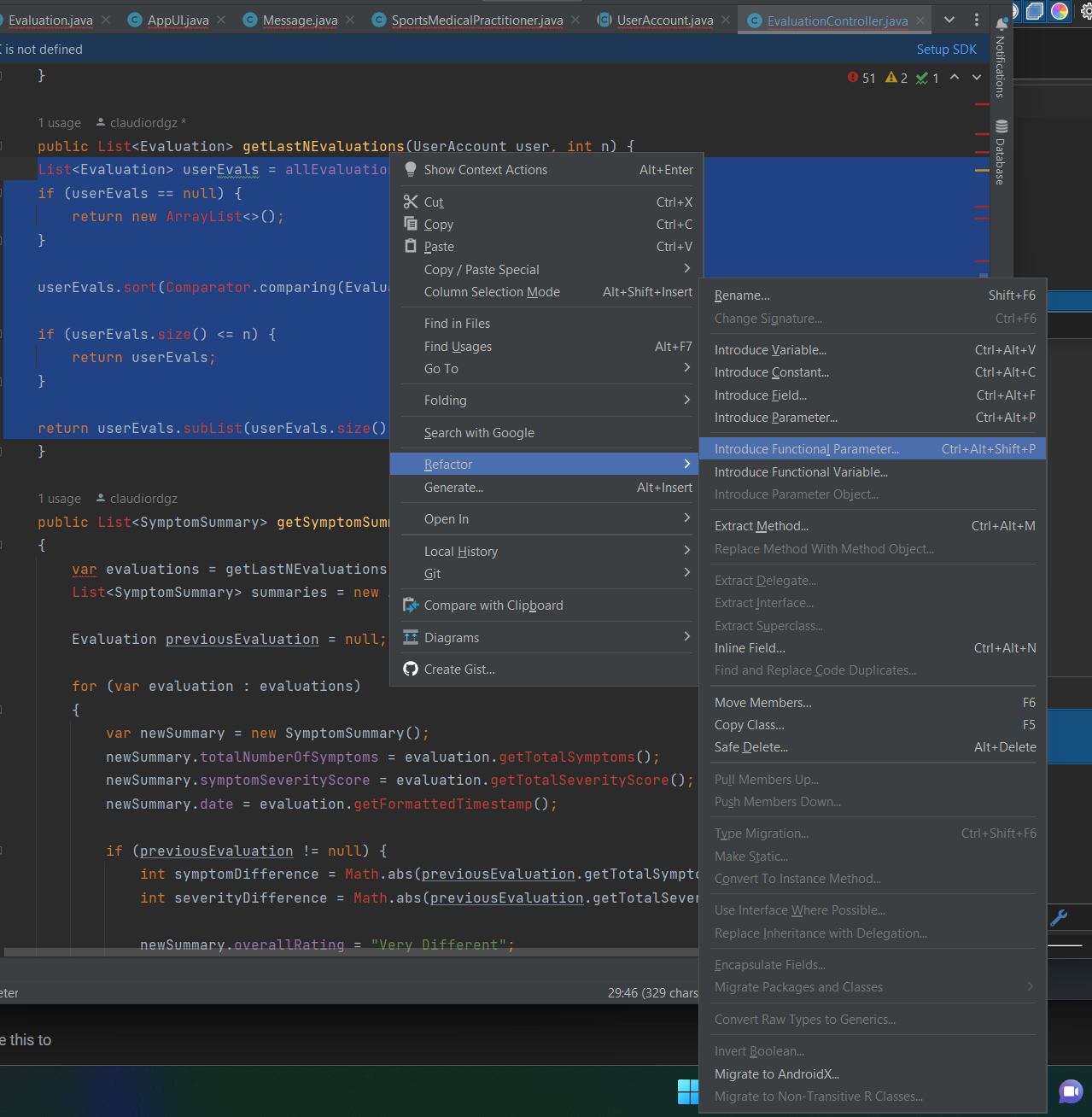
Visual Studio Code



1 Visual Studio Code Base Refactoring

Out of the box VSCode can perform basic Extract Method or Move Method actions. For more complicated refactors in specific languages it offers extensions of all kinds provided by the community.

IntelliJ IDEA

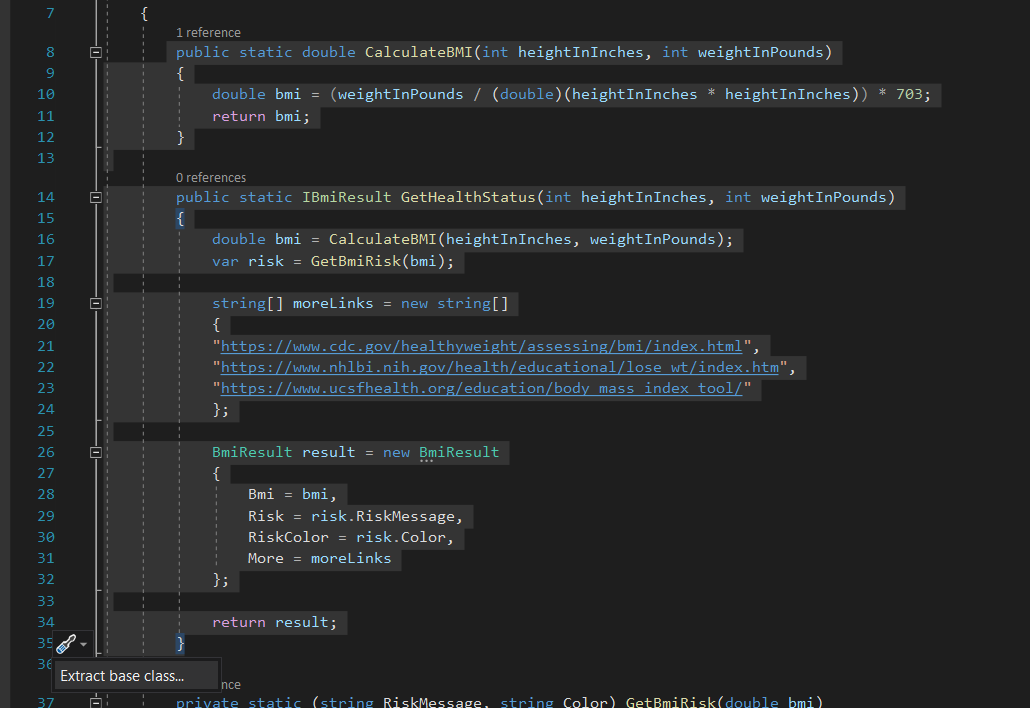


2 IntelliJ IDEA Refactoring Capabilities

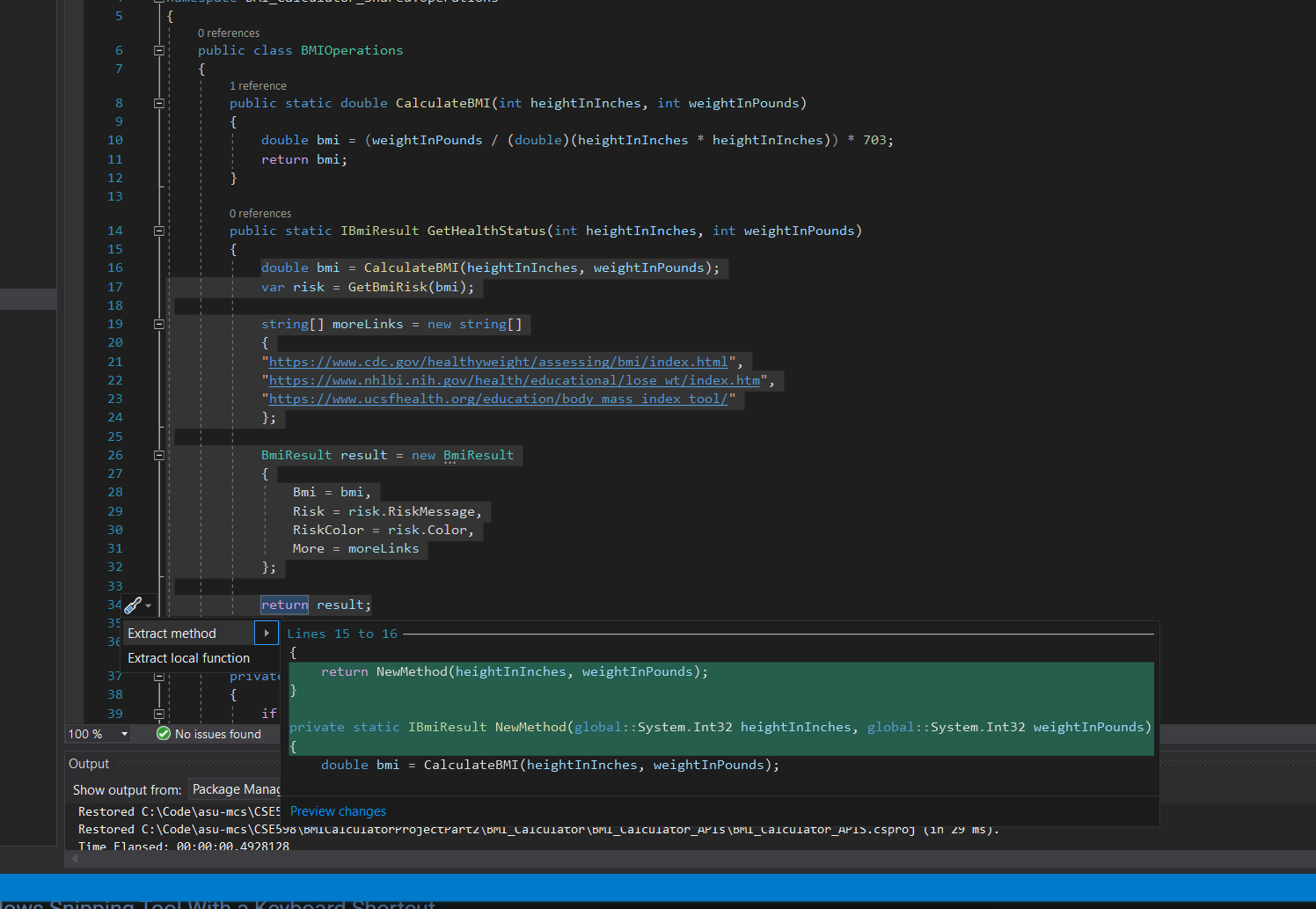
IntelliJ Ultimate comes with a wide array of refactoring options. The options of Extract Method, Move Method, and Extract Class are included together with many other options.

Visual Studio

Visual Studio has context specific tools. These options do include Extract Class, Extract Method, and Move Method. Visual Studio can also create different kinds of methods such as inline methods.



3 Visual Studio Extract Class, context aware from selecting multiple methods from inside an existing Class.



4 Visual Studio Extract or Move Method options, available from selecting code inside an existing method.

# References

Clausen, C. (2021, October). *Five Lines of Code.* Simon and Schuster.

Faella, M. (2020, March). *Seriously Good Software.* Manning Publications.

Fields, J., Harvie, S., Fowler, M., & Beck, K. (2009, October). *Refactoring: Ruby Edition.* Pearson Education.

Fowler, M., & Beck, K. (2013). *Refactoring Improving the Design of Existing Code.* Boston Addison-Wesley.

Lemaire, M. (2020). *Refactoring at Scale: Regaining Control of Your Codebase.* Sebastopol, CA, United States of America: O'Reilly Media, Inc.