* What is Code refactoring?
* Function what does --- it will be the same.
* Just change internal structure
* Not add new functionality or edit the existing functionality.
* How Prevent Change in External Behavior?
* Testing
* Using right IDE
* Formal code analysis
* Why need refactoring?
* Improves design of code
* More readable code
* Easy to maintain and understand
* Deliver more business value faster
* Increase reusability
* When refactoring?
* Refactor existing code so that you can add new functionality easily.
* To find bugs
* For code reviews
* Some Code Smells and Solution (Refactoring):

1. Inappropriate Naming: Explanation: Inappropriate or misleading names for variables, functions, classes, or methods can make the code harder to understand and maintain. Solution: Choose meaningful and descriptive names that accurately reflect the purpose and functionality of the code.

|  |  |
| --- | --- |
| Problem | Solution |
| // Inappropriate Naming  int a; // Variable name lacks context | // Improved Naming  int age; // Variable name clearly indicates it represents a person's age |

1. Comments: Explanation: Excessive or unclear comments can indicate code that is not self-explanatory or lacks proper documentation. Solution: Focus on writing self-explanatory code that reduces the need for comments. Use comments sparingly and only when necessary to provide additional context or clarify complex parts of the code.

|  |  |
| --- | --- |
| Problem | Solution |
| // Comments  // This loop iterates through the array  for (int i = 0; i < array.length; i++) {  // Process each element  } | // Improved Code  for (int i = 0; i < array.length; i++) {  // Process each element  } |

1. Dead Code: Explanation: Dead code refers to code that is no longer used or executed, making it unnecessary and potentially confusing for developers. Solution: Remove any unused or unreachable code to declutter the codebase and improve readability.

|  |  |
| --- | --- |
| Problem | Solution |
| // Dead Code  int result = calculateSum(5, 10);  // int unusedVariable = 20; | // Improved Code  int result = calculateSum(5, 10); |

1. Primitive Obsession: Explanation: Primitive Obsession occurs when primitive types, such as integers or strings, are used excessively instead of creating custom objects or classes to represent domain-specific concepts. Solution: Introduce appropriate classes or objects to encapsulate behavior and data related to the domain concept, improving code clarity and maintainability.

|  |  |
| --- | --- |
| Problem | Solution |
| // Primitive Obsession  double[] numbers = {1.5, 2.0, 3.7};  double sum = calculateSum(numbers); | // Improved Code  NumberCollection collection = new NumberCollection(numbers);  double sum = collection.calculateSum(); |

1. Duplicated Code: Explanation: Having the same or very similar code blocks in multiple places leads to code redundancy, making maintenance and updates more difficult. Solution: Extract the duplicated code into reusable functions or methods to eliminate redundancy and improve code maintainability.

|  |  |
| --- | --- |
| Problem | Solution |
| // Duplicated Code  public void calculateArea(int length, int width) {  int rectangleArea = length \* width;  System.out.println("Rectangle area: " + rectangleArea);  }  public void calculatePerimeter(int length, int width) {  int rectanglePerimeter = 2 \* (length + width);  System.out.println("Rectangle perimeter: " + rectanglePerimeter);  }  public void calculateAreaAndPerimeter(int length, int width) {  int rectangleArea = length \* width;  int rectanglePerimeter = 2 \* (length + width);  System.out.println("Rectangle area: " + rectangleArea);  System.out.println("Rectangle perimeter: " + rectanglePerimeter);  } | // Refactored Code  public void calculateArea(int length, int width) {  int rectangleArea = length \* width;  System.out.println("Rectangle area: " + rectangleArea);  }  public void calculatePerimeter(int length, int width) {  int rectanglePerimeter = 2 \* (length + width);  System.out.println("Rectangle perimeter: " + rectanglePerimeter);  }  public void calculateAreaAndPerimeter(int length, int width) {  calculateArea(length, width);  calculatePerimeter(length, width);  } |

1. Large Class: Large Class refers to a class that has grown too large in terms of responsibilities and functionality. It violates the Single Responsibility Principle and becomes difficult to understand, maintain, and extend. Large classes often contain a lot of methods and fields, making it harder to navigate and comprehend the codebase.

public class Employee {

private String name;

private int age;

private String address;

private String department;

// ...many more fields...

public Employee(String name, int age, String address, String department) {

this.name = name;

this.age = age;

this.address = address;

this.department = department;

// ...initialize other fields...

}

public void displayInformation() {

// ...code to display employee information...

}

public void calculateSalary() {

// ...code to calculate employee salary...

}

public void updateAddress(String newAddress) {

// ...code to update employee address...

}

// ...many more methods...

}

Solution to Large Class: To address the Large Class code smell, we can apply the refactoring technique called Extract Class. This involves breaking down the large class into smaller, more focused classes, each with its own responsibilities. By splitting the functionality into cohesive units, we improve code organization, understandability, and maintainability.

public class Employee {

private String name;

private int age;

private Address address;

private Department department;

// ...other fields...

public Employee(String name, int age, Address address, Department department) {

this.name = name;

this.age = age;

this.address = address;

this.department = department;

// ...initialize other fields...

}

// ...getter and setter methods...

public void displayInformation() {

// ...code to display employee information...

}

public void calculateSalary() {

// ...code to calculate employee salary...

}

}

public class Address {

private String address;

// ...other fields...

public Address(String address) {

this.address = address;

// ...initialize other fields...

}

// ...getter and setter methods...

public void updateAddress(String newAddress) {

// ...code to update address...

}

}

public class Department {

private String name;

// ...other fields...

public Department(String name) {

this.name = name;

// ...initialize other fields...

}

// ...getter and setter methods...

}

In the refactored code, the large Employee class has been broken down into smaller classes: Employee, Address, and Department. Each class now has a clear responsibility, making the codebase more maintainable and easier to understand.

1. Lazy Class: Lazy Class refers to a class that does not have enough significant functionality or does not contribute much to the system. It is a class that is underutilized and could be eliminated without affecting the overall system behavior.

Solution: Alternative Class with Different Interface To address the problem of a lazy class, one possible solution is to transform the class into an alternative class with a different interface. This means giving the class a new purpose or responsibility and defining a new interface that better represents its intended functionality. By doing so, we can make the class more relevant and ensure that it provides value to the system.

public class Calculator {

public double add(double a, double b) {

return a + b;

}

public double subtract(double a, double b) {

return a - b;

}

// Other methods for multiplication, division, etc.

}

In this case, the Calculator class is comprehensive and performs various mathematical operations. However, let's assume that over time, the project requirements change, and the system no longer needs the division and multiplication functionalities provided by the Calculator class. These methods become obsolete and are rarely used.

Solution: public class Adder {

public double add(double a, double b) {

return a + b;

}

}

By creating the Adder class, we have separated and extracted the addition functionality from the original Calculator class. This allows us to have a class that is specifically responsible for addition, making it more focused and purposeful. It also provides a different interface, as it only exposes the add method.

This approach ensures that each class has a clear responsibility and contributes meaningfully to the system. It eliminates the laziness of the original class and improves code maintainability by reducing unnecessary code and complexity.

1. Long Method:

public int calculateTotal(List<Item> items) {

int total = 0;

for (Item item : items) {

if (item.getQuantity() > 0) {

if (item.isDiscounted()) {

double discount = item.getPrice() \* 0.1;

total += (item.getPrice() - discount) \* item.getQuantity();

} else {

total += item.getPrice() \* item.getQuantity();

}

}

}

return total;

}

Solution (Refactor - Extract Methods):

public int calculateTotal(List<Item> items) {

int total = 0;

for (Item item : items) {

if (item.getQuantity() > 0) {

total += calculateItemTotal(item);

}

}

return total;

}

private int calculateItemTotal(Item item) {

if (item.isDiscounted()) {

double discount = item.getPrice() \* 0.1;

return (int) ((item.getPrice() - discount) \* item.getQuantity());

} else {

return item.getPrice() \* item.getQuantity();

}

}

1. Switch Statements:

public double calculateArea(String shape, double[] dimensions) {

double area = 0;

switch (shape) {

case "rectangle":

area = dimensions[0] \* dimensions[1];

break;

case "circle":

area = 3.14 \* dimensions[0] \* dimensions[0];

break;

case "triangle":

area = 0.5 \* dimensions[0] \* dimensions[1];

break;

}

return area;

}

Solution (Polymorphism/Strategy Pattern):

public interface ShapeAreaCalculator {

double calculateArea(double[] dimensions);

}

public class RectangleAreaCalculator implements ShapeAreaCalculator {

@Override

public double calculateArea(double[] dimensions) {

return dimensions[0] \* dimensions[1];

}

}

public class CircleAreaCalculator implements ShapeAreaCalculator {

@Override

public double calculateArea(double[] dimensions) {

return 3.14 \* dimensions[0] \* dimensions[0];

}

}

public class TriangleAreaCalculator implements ShapeAreaCalculator {

@Override

public double calculateArea(double[] dimensions) {

return 0.5 \* dimensions[0] \* dimensions[1];

}

}

public double calculateArea(String shape, double[] dimensions) {

ShapeAreaCalculator calculator;

switch (shape) {

case "rectangle":

calculator = new RectangleAreaCalculator();

break;

case "circle":

calculator = new CircleAreaCalculator();

break;

case "triangle":

calculator = new TriangleAreaCalculator();

break;

default:

throw new IllegalArgumentException("Invalid shape");

}

return calculator.calculateArea(dimensions);

}