Bookmark it

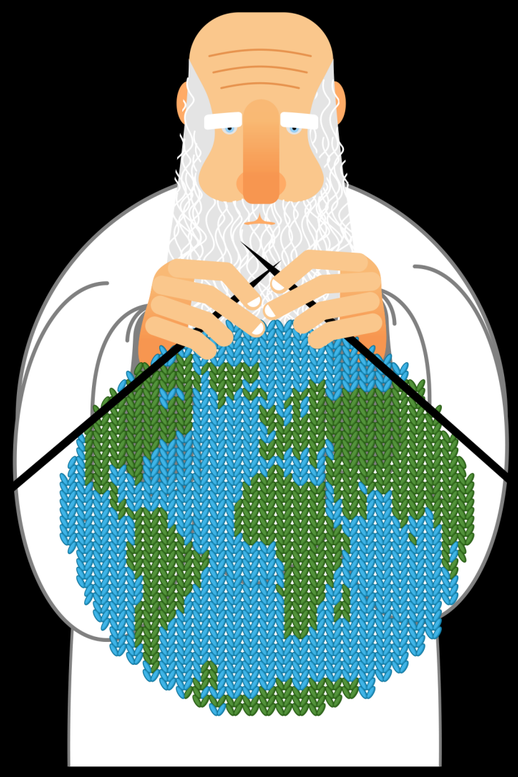
8 Anti Patterns

Java Coding

Follow

Jatin Shharma

God Object



- This anti-pattern occurs when a single class or **object becomes overly complex and takes on too many responsibilities, violating the principle of single responsibility.**

- It leads to code that is difficult to understand, maintain, and test.

Jatin Shharma

```java
public class GodObject {
    private List<Integer> numbers;
    private String name;

    public GodObject(List<Integer> numbers, String name) {
        this.numbers = numbers;
        this.name = name;
    }

    public void performOperations() {
        calculateSum();
        calculateAverage();
        validateInput();
        displayResults();
    }

    private void calculateSum() {
        int sum = 0;
        for (int number : numbers) {
            sum += number;
        }
        System.out.println("Sum: " + sum);
    }

    private void calculateAverage() {
        int sum = 0;
        for (int number : numbers) {
            sum += number;
        }
        double average = (double) sum / numbers.size();
        System.out.println("Average: " + average);
    }

    private void validateInput() {
        if (name == null || name.isEmpty()) {
            System.out.println("Invalid name");
        }
        if (numbers == null || numbers.isEmpty()) {
            System.out.println("No numbers provided");
        }


}
```

Explanation

- The GodObject class takes on multiple responsibilities. It performs operations on a list of numbers, calculates the sum and average, validates the input.
-
- This violates the principle of single responsibility, as the class is responsible for too many distinct tasks.

Spaghetti Code

- Spaghetti code refers to code that is highly tangled and **lacks a clear structure or organization.**

- It is characterized by **excessive and uncontrolled branching, jumping between different code sections, and poor separation of concerns**

Explanation

```java
public class SpaghettiCodeExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter a number:");
        int number1 = scanner.nextInt();

        if (number1 % 2 == 0) {
            System.out.println("Even number");
        } else {
            System.out.println("Odd number");
        }

        System.out.println("Enter another number:");
        int number2 = scanner.nextInt();

        int sum = number1 + number2;
        System.out.println("Sum: " + sum);

        if (sum > 10) {
            System.out.println("Sum is greater than 10");
        } else {
            System.out.println("Sum is less than or equal to 10");
        }

        System.out.println("Enter a third number:");
        int number3 = scanner.nextInt();

        int product = number1 * number2 * number3;
        System.out.println("Product: " + product);

        if (product % 2 == 0) {
            System.out.println("Product is even");
        } else {
            System.out.println("Product is odd");
        }

        // More code...
    }
}
```

- The code **lacks a clear structure and organization, leading to spaghetti code**.
- The logic is intertwined, making it difficult to follow and maintain.
- The code jumps between different sections without a clear separation of concerns.

Singleton Abuse

- Overuse of the singleton design pattern can lead to anti-patterns.
- **Creating global mutable state through singletons can make code harder to test, maintain, and reason about.**
- It can also **introduce tight coupling and hinder scalability.**

Magic Numbers and Strings

- Using **hard-coded numeric or string values directly in the code without assigning them to named constants** is considered an anti-pattern.
- Drawback: It makes the **code less readable, harder to maintain, and prone to error**

Example

```java
public class HardcodingExample {
    public static void main(String[] args) {
        int numberOfDays = 7;

        String[] weekdays = new String[numberOfDays];
        weekdays[0] = "Sunday";
        weekdays[1] = "Monday";
        weekdays[2] = "Tuesday";
        weekdays[3] = "Wednesday";
        weekdays[4] = "Thursday";
        weekdays[5] = "Friday";
        weekdays[6] = "Saturday";

        for (int i = 0; i < numberOfDays; i++) {
            System.out.println("Day " + (i + 1) + ": " + weekdays[i]);
        }
    }
}
```

- **Hardcoding values directly in the code is considered an anti-pattern because it makes the code less flexible and more difficult to maintain**
  If the names of the weekdays were to change or if we wanted to support different languages, we would have to modify the code in multiple places.
- ***A better approach would be to use named constants or external configuration files to store these values, allowing for easier modification and reusability***

Deep Nesting

```
public MappedField getMappedField(final String storedName) {
    for (final MappedField mf : persistenceFields) {
        for (final String n : mf.getLoadNames()) {
            if (storedName.equals(n)) {
                return mf;
            }
        }
    }
    return null;
}
```

- **Excessive nesting of conditional statements, loops, or blocks of code can make the code hard to follow, understand, and debug.**

- **It can lead to code duplication, increased cyclomatic complexity, and decreased readability.**

Example

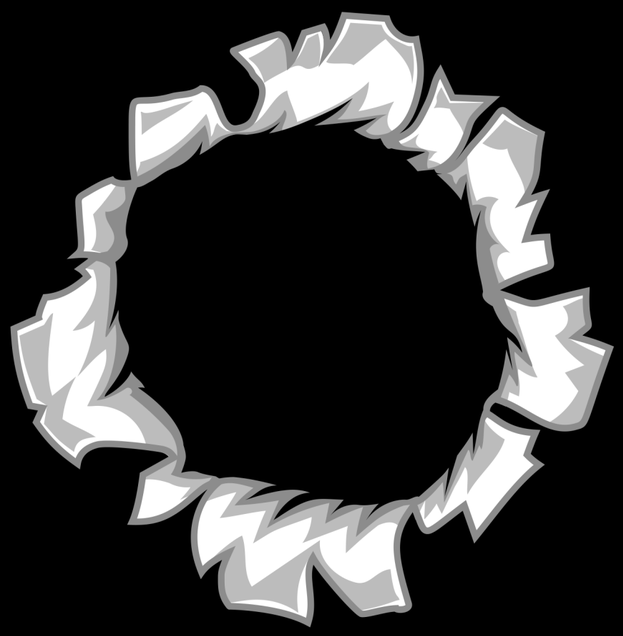
```
public void processData(List<List<List<List<Item>>>> data) {
    if (data != null && !data.isEmpty()) {
        for (List<List<List<Item>>> itemList : data) {
            if (itemList != null && !itemList.isEmpty()) {
                for (List<List<Item>> subItemList : itemList) {
                    if (subItemList != null &&
!subItemList.isEmpty()) {
                        for (List<Item> subSubItemList :
subItemList) {
                            if (subSubItemList != null &&
!subSubItemList.isEmpty()) {
                                for (Item subSubSubItem :
subSubItemList) {
                                    if (subSubSubItem != null) {
                                        // Process subSubSubItem
                                    } else {
                                        // Handle missing
subSubSubItem
                                    }
                                }
                            } else {
                                // Handle missing subSubItemList
                            }
                        }
                    } else {
                        // Handle missing subItemList
                    }
                }
            } else {
                // Handle missing itemList
            }
        }
    } else {
        // Handle missing data
    }
}
```

- To improve the code, **it's recommended to refactor it using more modular** and structured approaches, such as **using helper methods or recursion, to handle nested data structures in a more readable and maintainable manner.**

Improved Code

```java
public void processData(List<List<List<List<Item>>>> data) {
    processItemList(data);
}

private void processItemList(List<List<List<List<Item>>>> itemList)
{
    if (itemList != null && !itemList.isEmpty()) {
        for (List<List<List<Item>>> subItemList : itemList) {
            processSubItemList(subItemList);
        }
    }
}

private void processSubItemList(List<List<List<Item>>> subItemList)
{
    if (subItemList != null && !subItemList.isEmpty()) {
        for (List<List<Item>> subSubItemList : subItemList) {
            processSubSubItemList(subSubItemList);
        }
    }
}

private void processSubSubItemList(List<List<Item>> subSubItemList)
{
    if (subSubItemList != null && !subSubItemList.isEmpty()) {
        for (List<Item> subSubSubItemList : subSubItemList) {
            processSubSubSubItemList(subSubSubItemList);
        }
    }
}

private void processSubSubSubItemList(List<Item> subSubSubItemList)
{
    if (subSubSubItemList != null && !subSubSubItemList.isEmpty()) {
        for (Item item : subSubSubItemList) {
            // Process item
        }
    }
}
```

Empty Catch Blocks:



```java
try {
    // Code that may throw an exception
    // ...
} catch (Exception e) {
    // Empty catch block
}
```

- Catching exceptions and doing nothing in the catch block, or logging the exception without taking any appropriate action, is an anti-pattern.
- It can hide bugs and make it difficult to identify and handle exceptions correctly

Tight Coupling

```java
public class ShoppingCart {
    private ProductCatalog catalog;

    public ShoppingCart() {
        this.catalog = new ProductCatalog();
    }

    public void addItemToCart(int productId) {
        Product product = catalog.getProductById(productId);
        // Add the product to the shopping cart
    }

    // Other methods related to the shopping cart
}

public class ProductCatalog {
    private DatabaseConnection dbConnection;

    public ProductCatalog() {
        this.dbConnection = new DatabaseConnection();
    }

    public Product getProductById(int productId) {
        // Query the database using dbConnection and retrieve the
product
        // Return the product
    }

    // Other methods related to the product catalog
}

public class DatabaseConnection {
    // Database connection details and methods to query the database
}
```

- **Strong dependencies between classes or modules can lead to code that is difficult to modify, test, and reuse.**
- **It is important to strive for loose coupling by using interfaces, dependency injection, and other design principles**

Improved Code

```java
public interface ProductCatalog {
    Product getProductById(int productId);
    // Other methods related to the product catalog
}

public class ShoppingCart {
    private ProductCatalog catalog;

    public ShoppingCart(ProductCatalog catalog) {
        this.catalog = catalog;
    }

    public void addItemToCart(int productId) {
        Product product = catalog.getProductById(productId);
        // Add the product to the shopping cart
    }

    // Other methods related to the shopping cart
}
```

```java
public class DatabaseProductCatalog implements ProductCatalog {
    private DatabaseConnection dbConnection;

    public DatabaseProductCatalog(DatabaseConnection dbConnection) {
        this.dbConnection = dbConnection;
    }

    public Product getProductById(int productId) {
        // Query the database using dbConnection and retrieve the
product
        // Return the product
    }

    // Other methods related to the product catalog
}

public class DatabaseConnection {
    // Database connection details and methods to query the database
}
```

- In this improved version, the ShoppingCart class depends on the ProductCatalog interface rather than a concrete implementation. The specific implementation (DatabaseProductCatalog) is provided through constructor injection. This allows for easier modification or extension of the code by providing different implementations of the ProductCatalog interface, such as an in-memory catalog or a web service-based catalog, without impacting the ShoppingCart class.

- By introducing abstractions and relying on interfaces, we reduce the tight coupling between classes, enabling better separation of concerns, easier testing, and improved maintainability.

Large Classes/Methods:

- **Having excessively large classes or methods violates the principle of single responsibility and makes code harder to understand, maintain, and test.**
- **Breaking down functionality into smaller, more cohesive units is preferable.**

Follow for more!

Jatin Shharma

TestAutomationAcademy.in