Refactoring and code smells

BCS1430

Dr. Ashish Sai

Week 3 Lecture 1


BCS1430.ashish.nl


EPD150 MSM Conference Hall

Table of contents


“Just keep coding.
We can always fix it later.”

```
1 // This is a disgusting hack but I can't think of any other way to do this.
2 // TODO - SERIOUSLY FIX THIS BEFORE IT ENDS IN TEARS
3 // Date - twelve years ago
4 // Author - some guy who left nine years ago and
5 //          has since become a Buddhist monk that has taken a vow of silence
```

Introduction to Refactoring

Introduction to Refactoring

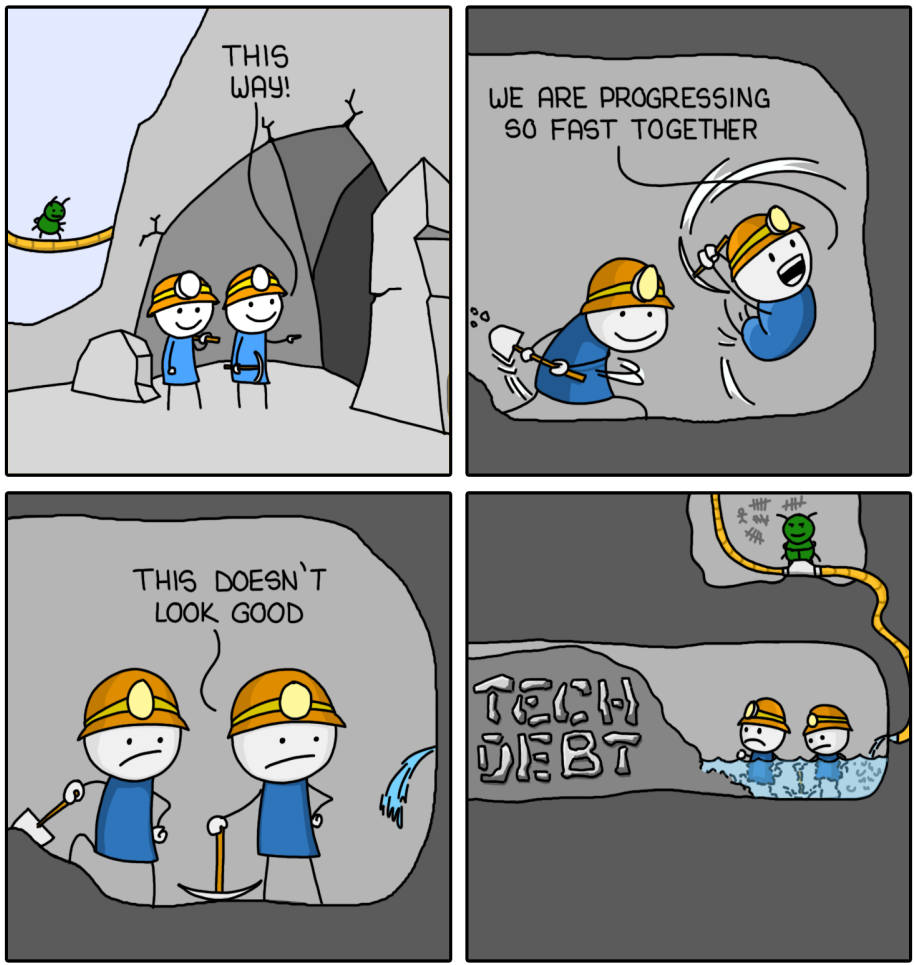
- **Refactoring:** The process of improving the internal structure of code without changing its external behavior.

Motivations for Refactoring

- **Enhance Code Clarity**
- **Mitigate Technical Debt** [1]
- **Ease Future Modifications**
- **Spot and Resolve Defects**

1. ***Technical Debt***: **The implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.**

TECH DEBT



MONKEYUSER.COM

Refactoring Challenges

- **Legacy Code:** Navigating complex, undocumented systems.
- **Time Management:** Juggling refactoring and new features.
- **Bug Risk:** Ensuring thorough testing.
- **Stakeholder Buy-In:** Advocating the benefits of refactoring.

When to Refactor

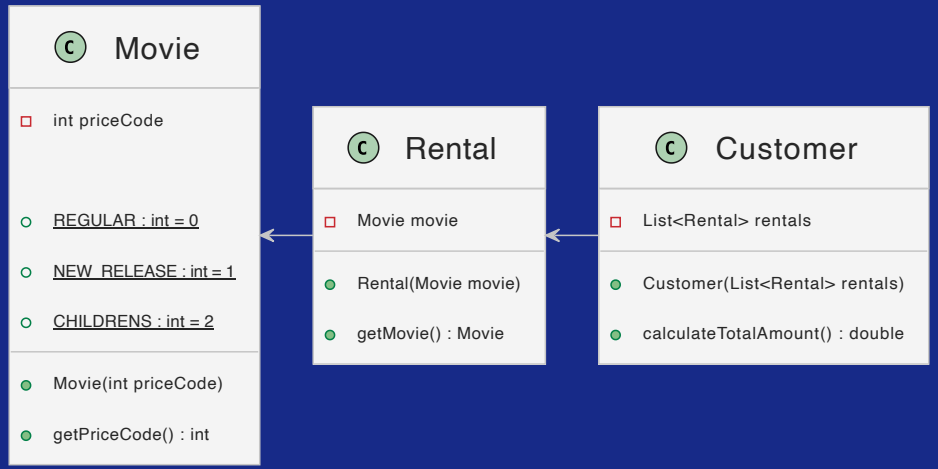
- **Before Adding a New Feature:** Tidy code before adding new features.
- **When Fixing a Bug:** Simplify surrounding code for clarity and easier bug resolution.
- **During Code Review:** Implement identified enhancements.
- **As You Go:** Continually refine code during development.

Refactoring Example

Refactoring

- **Small, Behavior-Preserving Transformations:** Minor yet cumulative modifications for major restructuring.
- **Systematic Process:** Ensures the external behavior remains consistent while improving internal architecture.

The Starting Code



The Starting Code

Consider a video store's calculation logic:

```
1  class Customer {
2      private List<Rental> rentals;
3
4      public double calculateTotalAmount() {
5          double totalAmount = 0;
6          for (int i = 0; i < rentals.size(); i++) {
7              Rental rental = rentals.get(i);
8              double amount = 0;
9
10             switch (rental.getMovie().getPriceCode()) {
11                 case Movie.REGULAR:
12                     amount = 2;
13                     break;
14                 case Movie.NEW_RELEASE:
15                     amount = 3;
16                     break;
17                 case Movie.CHILDRENS:
18                     amount = 1.5;
19                     break;
20                 default:
```

Identifying the First Refactor - Extract Method

Goal: Simplify the `calculateTotalAmount` method by creating smaller, more readable chunks of code.

Extract the switch statement into `amountFor` in the `Rental` class:

```
1   public double amountFor() {
2           switch (getMovie().getPriceCode()) {
3               case Movie.REGULAR:
4                   return 2;
5               case Movie.NEW_RELEASE:
6                   return 3;
7               case Movie.CHILDRENS:
8                   return 1.5;
9               default:
10                  return 0;
11          }
12      }
```

Refactoring Step by Step

1. **Identify a section to extract:** Target a self-contained code chunk.
2. **Create a new method:** Move the code into a new method in the appropriate class.
3. **Replace old code:** Substitute the original code with the new method call.
4. **Test:** Ensure behavior remains consistent.

Benefits of Extract Method

- **Improved Readability:** The main method becomes concise.
- **Reusability:** Other system parts can now use the new method.
- **Separation of Concerns:** Enhances encapsulation.

Continuing the Refactoring

Look for opportunities to:

- Break down long methods.
- Move operations closer to data.
- Replace conditionals with polymorphism.

Polymorphism Over Conditionals

Before: Switch statements based on movie type.

After: Use polymorphism with subclasses like `RegularMovie`, `NewReleaseMovie`, and `ChildrensMovie`.

Implementing Polymorphism

1. **Define a common interface:** All movie types should respond to `getCharge`.
2. **Create subclasses:** Each represents a different movie type.
3. **Move the logic:** Shift charge calculation to the respective subclass.

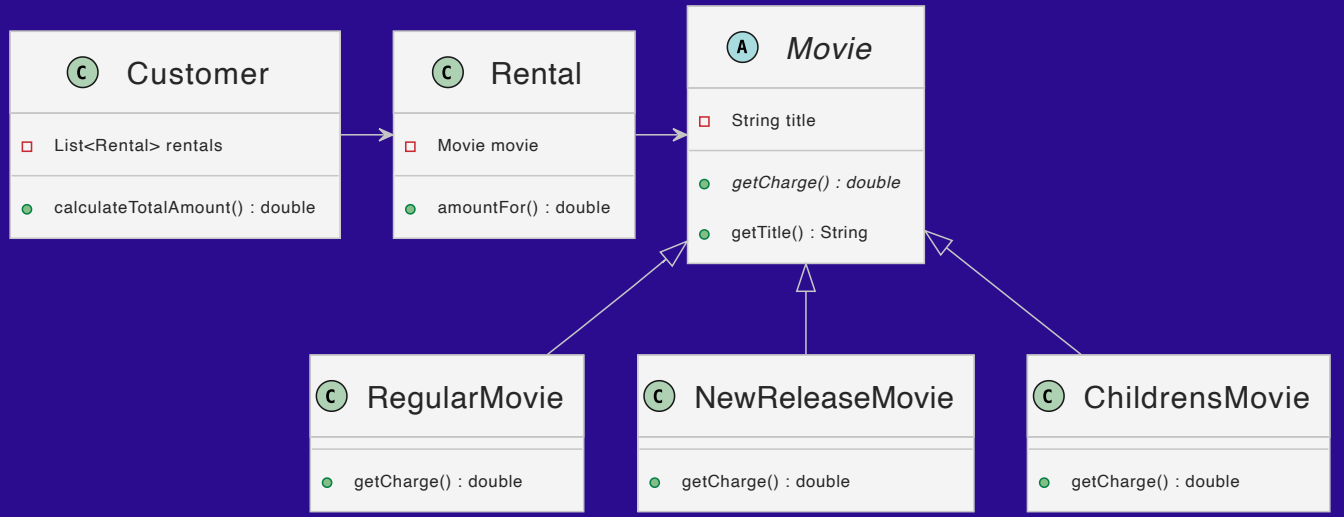
Example:

```
 1  public abstract class Movie {
 2      private String title;
 3
 4      public Movie(String title) {
 5          this.title = title;
 6      }
 7
 8      public String getTitle() {
 9          return title;
10      }
11
12      abstract double getCharge();
13  }
14
15  public class RegularMovie extends Movie {
16      public RegularMovie(String title) {
17          super(title);
18      }
19
20      @Override
```

Example:

```
 1  class Rental {
 2      private Movie movie;
 3
 4      public Rental(Movie movie) {
 5          this.movie = movie;
 6      }
 7
 8      public double amountFor() {
 9          return movie.getCharge();
10      }
11  }
```

Example:


Customer
List<Rental> rentals
calculateTotalAmount() : double
Rental
Movie movie
amountFor() : double
Movie
String title
getCharge() : double
getTitle() : String
RegularMovie
getCharge() : double
NewReleaseMovie
getCharge() : double
ChildrensMovie
getCharge() : double

Benefits of Polymorphism

- **Flexibility:** Adding a new movie type is straightforward.
- **Adherence to Open/Closed Principle:** System is open for extension but closed for modification.

So far

- Encapsulation Achieved: Broke down a complex method into smaller, manageable parts, enhancing readability and flexibility.
- Refactoring Impact: Small, progressive enhancements yielding substantial overall improvements.

Principles of Refactoring

- **Core Principles:**
  - Do not change the observable behavior of the code.
  - Make small, incremental changes.
  - Test after each change.
  - Focus on simplicity and clarity.

Introduction to Code Smells

What? Can code “smell” 👃 ????

> Well it doesn’t have a nose…. but it definitely can stink 😷!

Introduction to Code Smells

- **Code Smells:** Indicators of potential issues in your code that may require refactoring.
- **Importance:** Recognizing smells is the first step towards improving code quality.

Introduction to Code Smells

- **Common Smells:** Include Duplicated Code, Long Method, Large Class, and more.
- Use smells as a guide, not a strict rule, to identify areas for improvement.

Types of Code Smells

- **Bloaters:** Oversized constructs that complicate code management.
- **Object-Orientation Abusers:** Poor OOP design choices.
- **Change Preventers:** Structures that make changes difficult and widespread.
- **Dispensables:** Redundant elements that reduce clarity and efficiency.
- **Couplers:** Excessive class coupling or delegation issues.

Bloaters

Bloaters

- **Definition:** Oversized code, methods, and classes becoming difficult to manage.
- **Characteristics:** Grow gradually over time, complicating program evolution.
- **Cause:** Lack of effort to refine or reduce code complexity.

Long Method

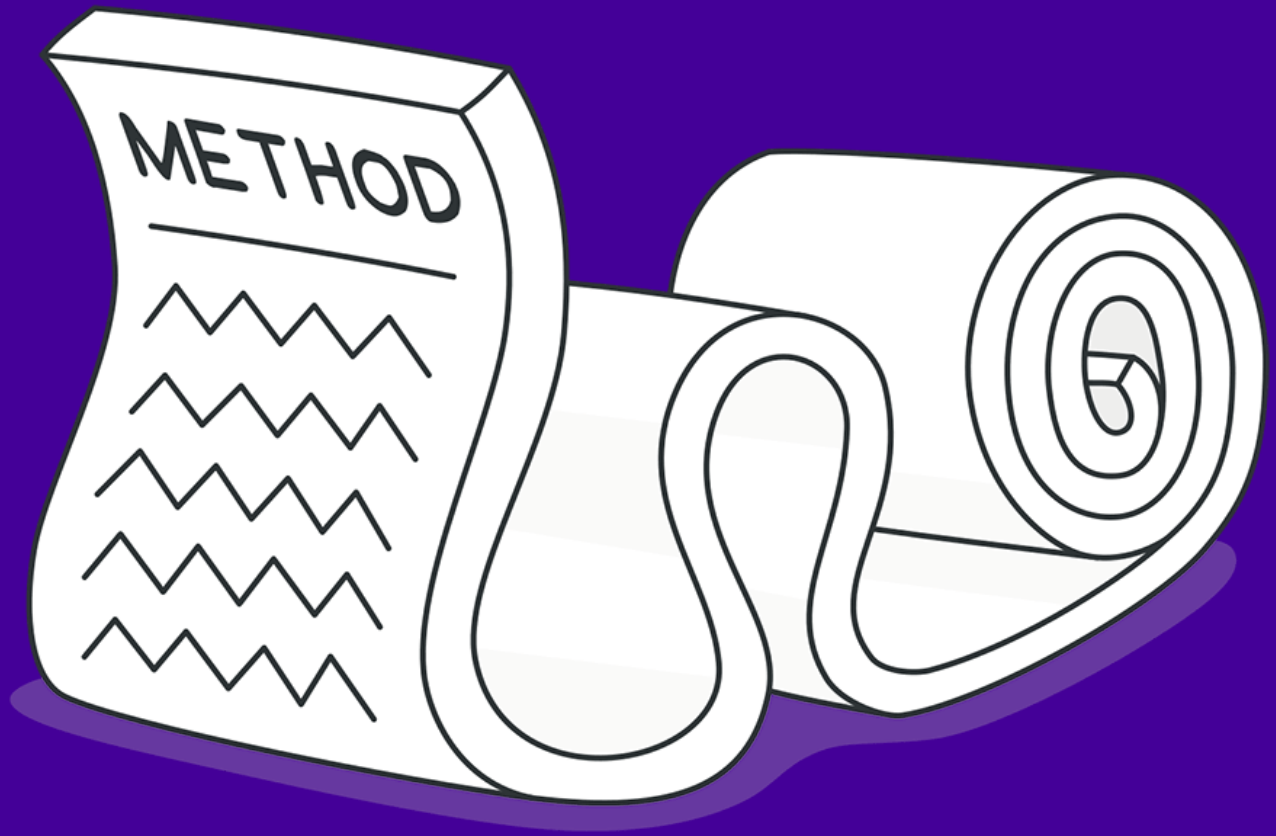


Image Credit: Refactoring.Guru

1. Long Method - Problem

- **What It Is:** Methods with too many lines of code.
- **Problems:** Hard to understand and maintain. Often contain hidden bugs.

1. Long Method - Problem

```
1     public class ReportGenerator {
2         public void generateReport(DataSet data) {
3             // Initialization and setup
4             String report = "";
5             // Complex data processing
6             for (DataPoint point : data) {
7                 // Detailed data analysis and report generation
8                 report += analyzeDataPoint(point);
9             }
10            // More processing and formatting
11            // Final report compilation
12            System.out.println(report);
13        }
14    }
```

1. Long Method - Solution

- Break down into smaller methods, each with a clear purpose.

```java
public class ReportGenerator {
    public void generateReport(DataSet data) {
        String report = processDataForReport(data);
        outputReport(report);
    }

    private String processDataForReport(DataSet data) {
        String report = "";
        for (DataPoint point : data) {
            report += analyzeDataPoint(point);
        }
        return report;
    }

    private void outputReport(String report) {
        // Final report compilation and output
        System.out.println(report);
    }

    private String analyzeDataPoint(DataPoint point) {
        // Detailed data analysis
        // Return analysis result as String
    }
}
```

1. Refactoring Techniques: Long Method

- **Extract Method:** Break down into smaller methods with clear names indicating their purpose.
- **Replace Temp with Query:** Replace temporary variables with queries to the method itself.
- **Introduce Parameter Object or Preserve Whole Object:** Group parameters into objects.

1. Long Method: Extract Method

- **Technique:** Break down a long method into smaller methods with clear names indicating their purpose.

Problem Code:

```
1 public void printReport(List<Report> reports) {
2     for(Report report : reports) {
3         // Print header
4         // Print details
5         // Print footer
6     }
7 }
```

1. Long Method: Extract Method

Refactored Code:

```
1  public void printReport(List<Report> reports) {
2      for(Report report : reports) {
3          printHeader(report);
4          printDetails(report);
5          printFooter(report);
6      }
7  }
8
9  private void printHeader(Report report) { /*...*/ }
10 private void printDetails(Report report) { /*...*/ }
11 private void printFooter(Report report) { /*...*/ }
```

- **Benefits:** Each part of the code has a clear purpose, improving readability and maintainability.

1. Long Method: Replace Temp with Query

- **Technique:** Replace temporary variables with queries to the method itself.

Problem Code:

```
1 public double calculateTotal() {
2     double basePrice = quantity * itemPrice;
3     if(basePrice > 1000) {
4         return basePrice * 0.95;
5     } else {
6         return basePrice * 0.98;
7     }
8 }
```

1. Long Method: Replace Temp with Query

Refactored Code:

```
1  public double calculateTotal() {
2      if(basePrice() > 1000) {
3          return basePrice() * 0.95;
4      } else {
5          return basePrice() * 0.98;
6      }
7  }
8
9  private double basePrice() {
10     return quantity * itemPrice;
11 }
```

- **Benefits:** Reduces the clutter of temporary variables and ensures that the calculation is always up to date.

1. Long Method: Parameter or Whole Object

- **Technique:** Group parameters into objects.

Problem Code:

```
1 public void createReservation(Date start, Date end, Guest guest, Room room) {
2     // Logic using start, end, guest, and room
3 }
```

1. Long Method: Parameter or Whole Object

Refactored Code:

```
1  public class ReservationRequest {
2      private Date start;
3      private Date end;
4      private Guest guest;
5      private Room room;
6      // Constructor and accessors
7  }
8
9  public void createReservation(ReservationRequest request) {
10     // Logic using request object
11 }
```

- **Benefits:** Simplifies method signatures, makes the code more readable, and groups related data together.

Large Class

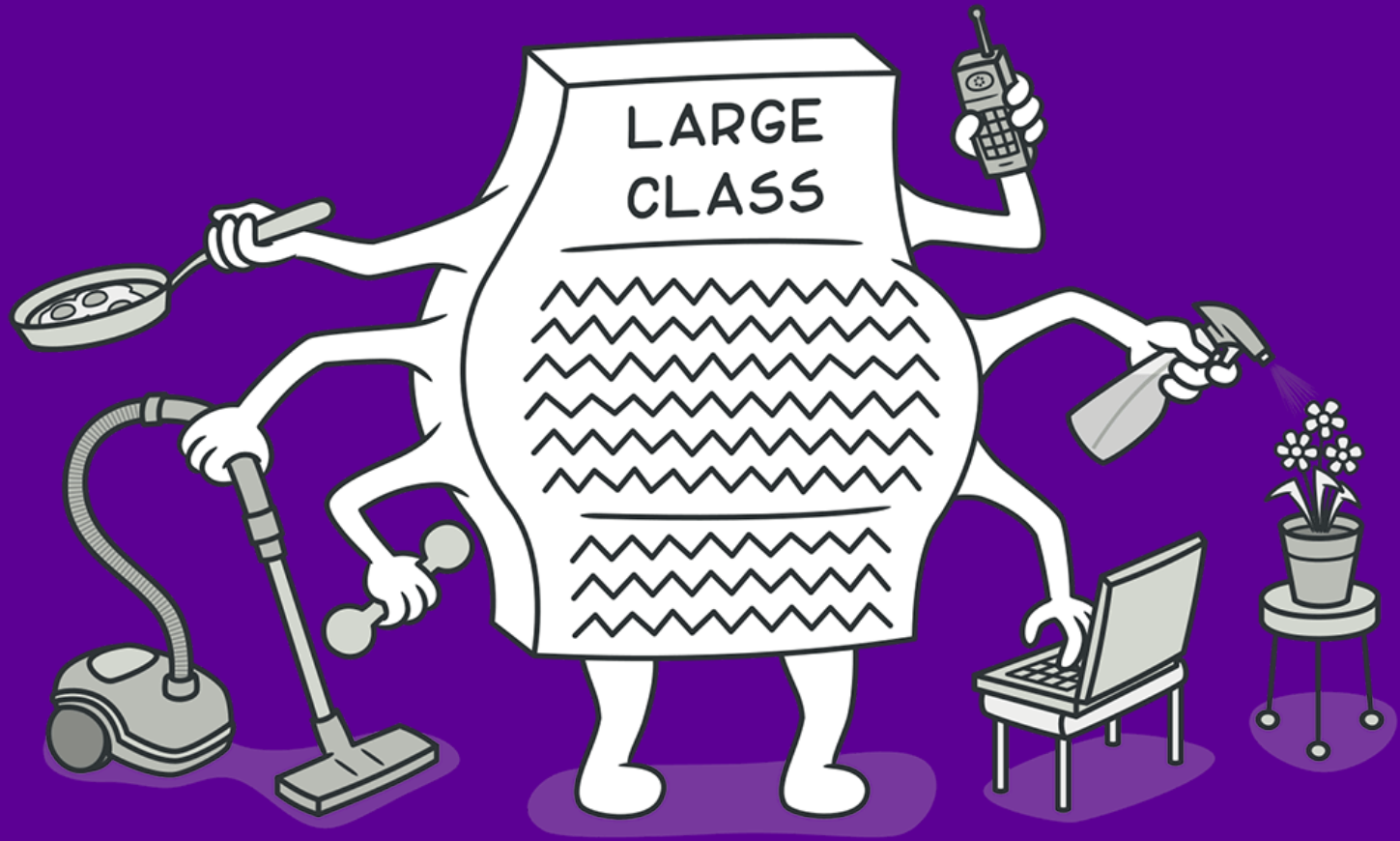


Image Credit: Refactoring.Guru

2. Large Class: Problem

- **What It Is:** Classes with too many responsibilities.
- **Problems:** Difficult to understand, maintain, and modify.

2. Large Class: Problem

```java
public class Employee {
    private String name;
    private String address;
    private String phoneNumber;
    private double salary;
    // ... many more attributes

    public void calculatePay() {
        // Method to calculate pay
    }

    public void save() {
        // Method to save employee details
    }

    // ... many more methods
}
```

- This `Employee` class is handling personal details, pay calculations, and data storage, among other things.

2. Large Class: Solution

- **Solution:** Use techniques like Extract Class to divide into smaller, more focused classes.

```
public class Employee {
    private String name;
    private EmployeeDetails details;
    private PayCalculator payCalculator;

    // Employee class now delegates responsibilities
}

public class EmployeeDetails {
    private String address;
    private String phoneNumber;
    // ... other personal details
}

public class PayCalculator {
    private double salary;

    public void calculatePay() {
        // Method to calculate pay
    }
}
```

2. Large Class

- **Refactoring Techniques:**
  - **Extract Class:** Create new classes to handle parts of the functionality.
  - **Extract Subclass or Extract Interface:** Use when part of the behavior is used in only some instances.

2. Large Class: Extract Class

- **Technique:** Create new classes to handle parts of the functionality.

Problem Code:

```
class Order {
    private Customer customer;
    private List<Item> items;
    private Address shippingAddress;
    private Address billingAddress;
    // ... many more fields and methods related to payment, shipping, etc.

    void processOrder() { /* ... */ }
    void calculateTotal() { /* ... */ }
    // ... many more methods
}
```

2. Large Class: Extract Class

Refactored Code:

```
class Order {
    private Customer customer;
    private List<Item> items;
    private PaymentDetails paymentDetails;
    private ShippingDetails shippingDetails;
    // Simplified Order class
}

class PaymentDetails {
    private Address billingAddress;
    // ... payment related fields and methods
}

class ShippingDetails {
    private Address shippingAddress;
    // ... shipping related fields and methods
}
```

- **Benefits:** Reduces complexity by delegating responsibilities to new classes, improving readability and maintainability.

2. Large Class: Extract Subclass or Interface

- **Technique:** Use when part of the behavior is used in only some instances.

Problem Code:

```
class Order {
    private boolean isPriorityOrder;
    // ... many fields and methods

    void processOrder() {
        if (isPriorityOrder) {
            // Priority order processing
        } else {
            // Normal order processing
        }
    }
}
```

2. Large Class: Extract Subclass or Interface

Refactored Code (Extract Subclass):

```
class Order {
    // ... common fields and methods
}

class PriorityOrder extends Order {
    // Priority order specific fields and methods
    @Override
    void processOrder() {
        // Priority order processing
    }
}

class NormalOrder extends Order {
    // Normal order specific fields and methods
    @Override
    void processOrder() {
        // Normal order processing
    }
}
```

2. Large Class: Extract Subclass or Interface

Refactored Code (Extract Interface):

```
interface Order {
    void processOrder();
}

class PriorityOrder implements Order {
    // Priority order specific fields and methods
    public void processOrder() {
        // Priority order processing
    }
}

class NormalOrder implements Order {
    // Normal order specific fields and methods
    public void processOrder() {
        // Normal order processing
    }
}
```

- **Benefits:** Separates different behaviors into distinct classes or interfaces, enhancing the Single Responsibility Principle and making the system easier to understand and modify.

Long Parameter List

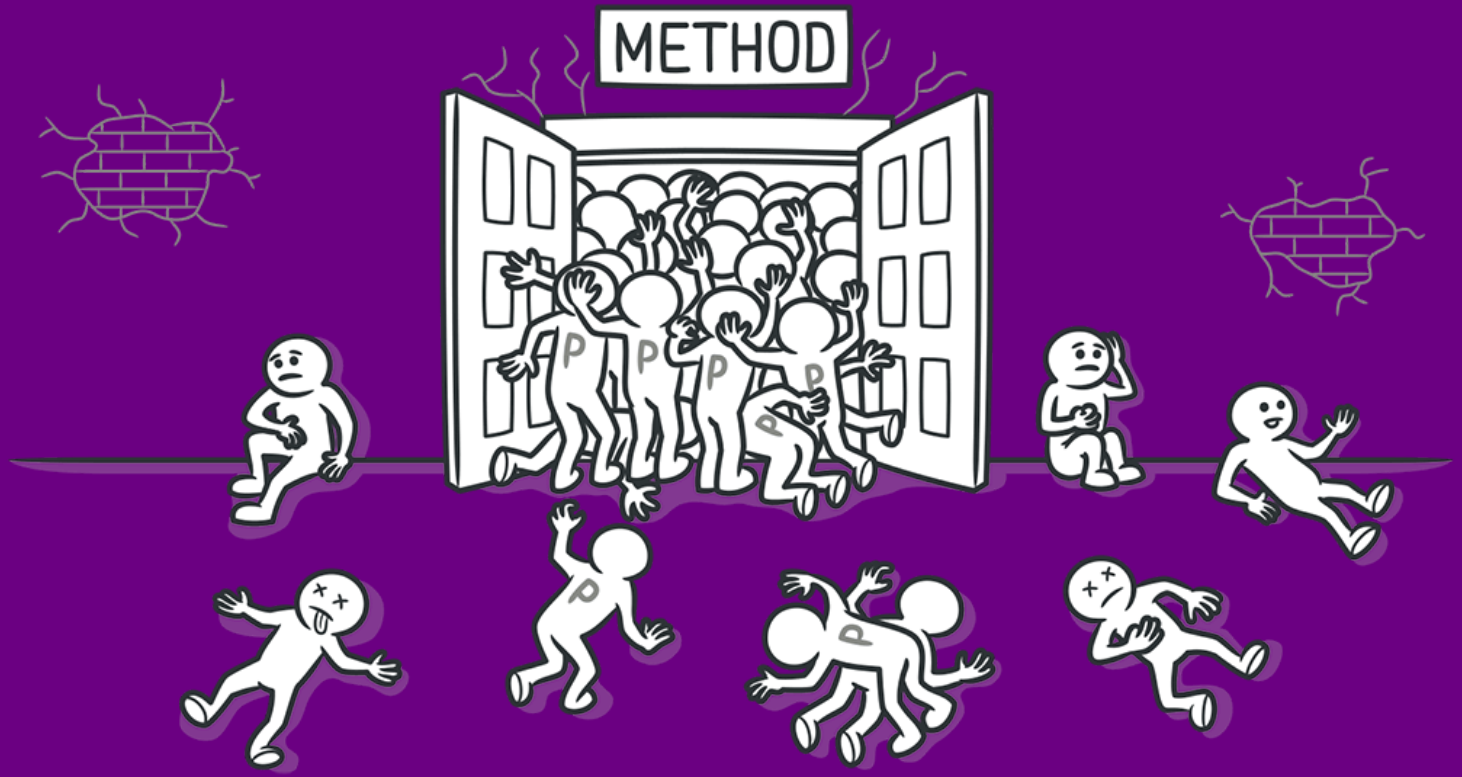


Image Credit: Refactoring.Guru

3. Long Parameter List

- **What It Is:** Methods with a high number of parameters.
- **Problems:** Hard to understand and use. Increases the risk of errors.

3. Long Parameter List

```
1  public void processOrder(String customerName, String customerEmail, String shippingAddress,
2                           String billingAddress, String orderItem, int quantity,
3                           String paymentMethod, String cardNumber, String expiryDate,
4                           String cvv) {
5      // Method logic for processing the order...
6  }
```

3. Solution to Long Parameter List

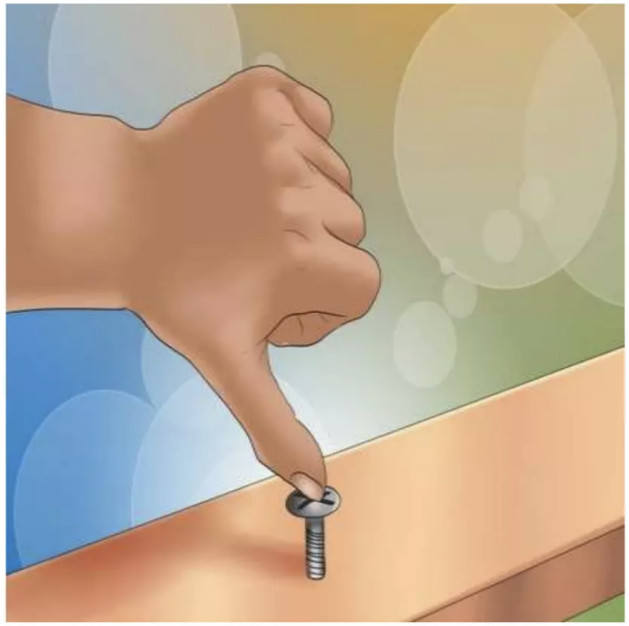
- **Solution:** Use objects to group parameters or replace parameters with method calls.

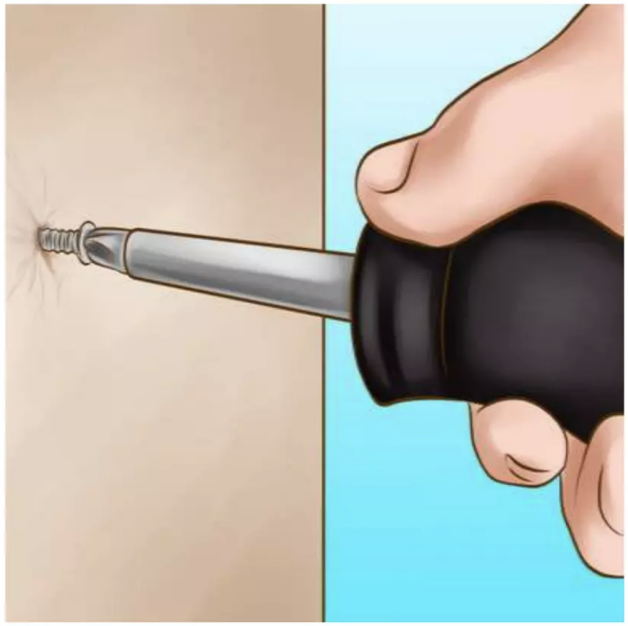
```java
public class CustomerInfo {
    private String name;
    private String email;
    // Constructors, getters, and setters...
}

public class OrderDetails {
    private String orderItem;
    private int quantity;
    // Constructors, getters, and setters...
}

public class PaymentInfo {
    private String paymentMethod;
    private String cardNumber;
    private String expiryDate;
    private String cvv;
    // Constructors, getters, and setters...
}

// The method now takes these objects as parameters:
public void processOrder(CustomerInfo customerInfo, OrderDetails orderDetails, PaymentInfo paymentInfo) {
    // Method logic using the provided objects...
}
```

3. Long Parameter List: Alternative Solution

- **Alternative:** Replace parameters with method calls to retrieve the needed information.
- Example: Method Calls Instead of Parameters

```
// Assuming there are methods to retrieve customer and order details:
CustomerInfo customerInfo = getCustomerInfo(customerId);
OrderDetails orderDetails = getOrderDetails(orderId);
PaymentInfo paymentInfo = getPaymentInfo(paymentId);

// The method can be simplified further:
public void processOrder(CustomerInfo customerInfo, OrderDetails orderDetails, PaymentInfo paymentInfo) {
    // Method logic...
}
```





4. Primitive Obsession

- **Identification:** Overuse of primitive types instead of small objects for simple tasks.
- **Example:**

```
1 public void processDate(String date) {
2     // Complex manipulation using string
3 }
```

4. Primitive Obsession

- **Refactoring Techniques:**
  - **Replace Data Value with Object:** Create an object for the type of data you have.
  - **Replace Array with Object:** Use an object to represent a group of related data instead of an array.

4. Primitive Obsession: Data Value

- **Technique:** Create an object for the type of data you have.

Problem Code:

```java
class User {
    private String name; // Primitive type
    private String phone; // Primitive type

    public void displayUserInfo() {
        System.out.println("Name: " + name + ", Phone: " + phone);
    }
}
```

4. Primitive Obsession: Data Value

Refactored Code:

```java
class Phone {
    private String number;

    public Phone(String number) {
        this.number = number;
    }

    public String formatNumber() {
        // Format number (e.g., add dashes)
        return number;
    }
}

class User {
    private String name; // Still primitive type, appropriate here
    private Phone phone; // Replaced with object

    public User(String name, String phoneNumber) {
        this.name = name;
        this.phone = new Phone(phoneNumber);
    }

    public void displayUserInfo() {
        System.out.println("Name: " + name + ", Phone: " + phone.formatNumber());
    }
}
```

- **Benefits:** Encapsulates data and behavior related to phone numbers, making the code more understandable and maintainable.

4. Primitive Obsession: Array

- **Technique:** Use an object to represent a group of related data instead of an array.

Problem Code:

```
class UserData {
    String[] userInfo; // [0] for name, [1] for phone, [2] for address

    public void displayUserInfo() {
        System.out.println("Name: " + userInfo[0] + ", Phone: " + userInfo[1] + ", Address: " + userInfo[2]);
    }
}
```

4. Primitive Obsession: Array

Refactored Code:

```java
class User {
    private String name;
    private String phone;
    private String address;

    public User(String name, String phone, String address) {
        this.name = name;
        this.phone = phone;
        this.address = address;
    }

    public void displayUserInfo() {
        System.out.println("Name: " + name + ", Phone: " + phone + ", Address: " + address);
    }
}
```

- **Benefits:** Each piece of data is now clearly defined, improving readability and reducing the risk of errors like incorrect array index access.

Data Clumps

Image Credit: Refactoring.Guru

5. Data Clumps

- **Identification:** Groups of data that always appear together but aren’t organized into a structure.
- **Example:**

```
    public void createCustomer(String firstName, String lastName, String street, String city, String zip) {
        // Method body
    }
```

5. Data Clumps

- **Refactoring Techniques:**
  - **Introduce Parameter Object or Class:** Group the clumped data into a single object representing the entire concept.

5. Data Clumps: Parameter Object or Class

- **Technique:** Group the clumped data into a single object representing the entire concept.

Problem Code:

```
class CustomerService {
    public void createCustomer(String firstName, String lastName, String street, String city, String zip) {
        // Logic to create a customer
    }
}
```

- **Issues:** The `createCustomer` method takes multiple parameters related to customer and address, making it cumbersome and prone to errors.

5. Data Clumps: Parameter Object or Class

Refactored Code:

```
class Customer {
    String firstName;
    String lastName;
    Address address;
}

class Address {
    String street;
    String city;
    String zip;
}

class CustomerService {
    public void createCustomer(Customer customer) {
        // Logic to create a customer
    }
}
```

- **Benefits:** Encapsulates related data into coherent structures, simplifying method signatures and promoting code reuse and clarity. This makes the code more maintainable and understandable, as well as easier to extend with new customer-related attributes or behaviors.

Object-Orientation Abusers

Object-Orientation Abusers

- **Definition:** Poor OOP application leading to design flaws.
- **Characteristics:** Complex and rigid code; misused inheritance and polymorphism.
- **Cause:** Misunderstanding or misapplying OOP principles; forcing problems into unsuitable solutions.

Switch Statements

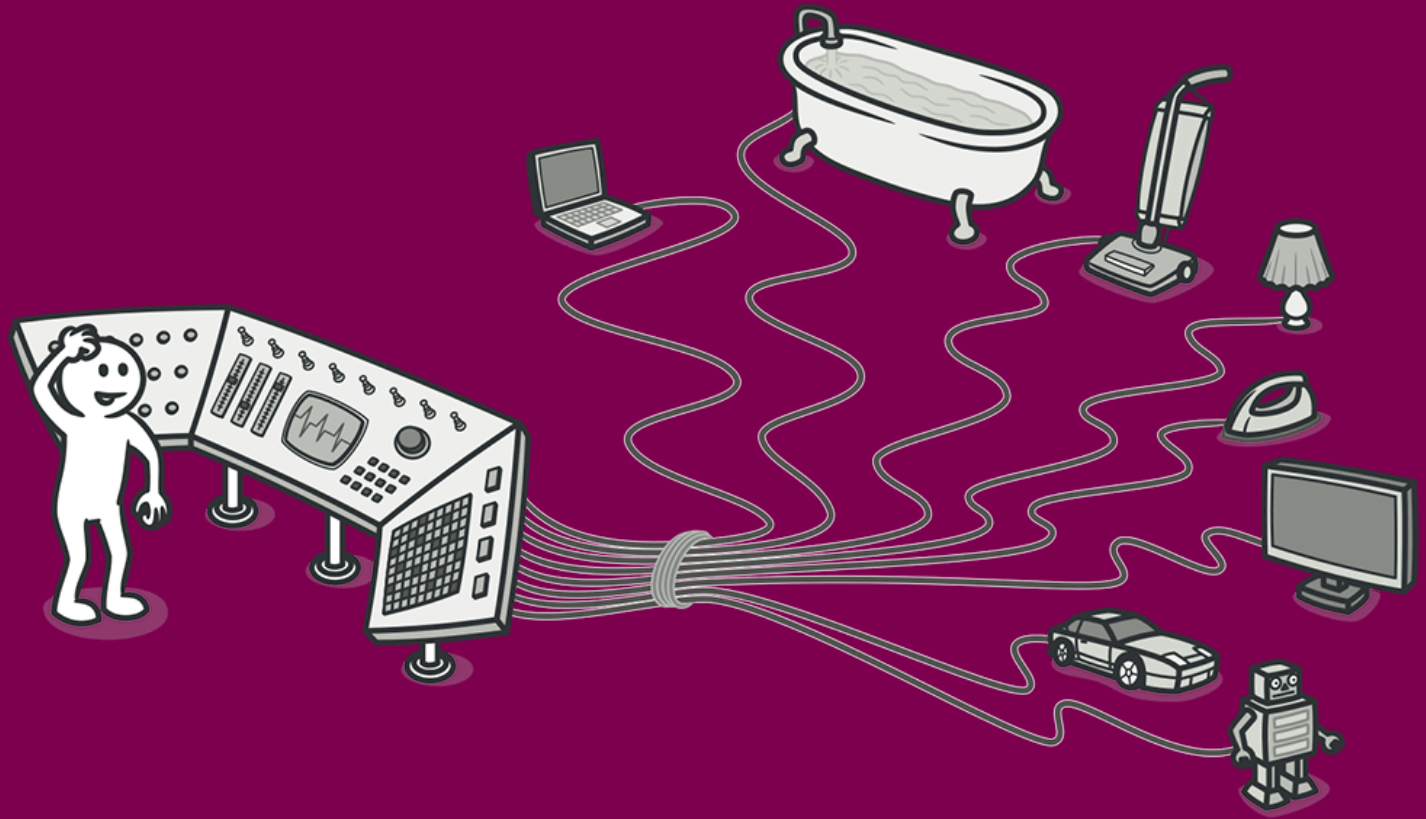


Image Credit: Refactoring.Guru

6. Switch Statements

- **What It Is:** Excessive use of switch or complex if-else chains.
- **Problems:** Hard to modify and extend. Often violates Open/Closed Principle.


E480
Backspace
Enter
Shift

6. Switch Statements

```java
public class AnimalSound {
    public String makeSound(String animalType) {
        switch (animalType) {
            case "dog":
                return "Bark";
            case "cat":
                return "Meow";
            case "cow":
                return "Moo";
            // More cases for different animals
            default:
                throw new IllegalArgumentException("Unknown animal type");
        }
    }
}
```

- **Issues:** Each new animal type requires modifying the `makeSound` method.

6. Switch Statements - Solution

- **Solution:** Use polymorphism and other design patterns to handle varying behavior more gracefully.

```java
interface Animal {
    String makeSound();
}

class Dog implements Animal {
    public String makeSound() { return "Bark"; }
}

class Cat implements Animal {
    public String makeSound() { return "Meow"; }
}

class Cow implements Animal {
    public String makeSound() { return "Moo"; }
}

// Using the Animal interface
public class AnimalSound {
    public String makeSound(Animal animal) {
        return animal.makeSound();
    }
}
```

- **Benefits:** Easily extendable by adding new classes. No need to modify existing code to add new animal types, adhering to the Open/Closed Principle.

6. Switch Statements

- **Refactoring Techniques:**
  - **Replace Conditional with Polymorphism:** Use polymorphism to handle varying behavior based on type.
  - **Replace Type Code with Subclasses:** Create subclasses for each type code.

6. Switch Statements: Polymorphism

- **Technique:** Use polymorphism to handle varying behavior based on an object's type instead of a switch or complex if-else chains.

Problem Code:

```
class Employee {
    Type type;

    double calculatePay() {
        switch (type) {
            case COMMISSIONED:
                return calculateCommissionedPay();
            case HOURLY:
                return calculateHourlyPay();
            // other cases
        }
    }
}
```

6. Switch Statements: Polymorphism

Refactored Code:

```
abstract class Employee {
    abstract double calculatePay();
}

class CommissionedEmployee extends Employee {
    double calculatePay() {
        return calculateCommissionedPay();
    }
}

class HourlyEmployee extends Employee {
    double calculatePay() {
        return calculateHourlyPay();
    }
}

// Usage:
Employee employee = new CommissionedEmployee();
double pay = employee.calculatePay();
```

- **Benefits:** Eliminates the switch statement by encapsulating the varying behavior within each subclass. This makes the code easier to extend and maintain, as new types can be added without modifying existing code.

6. Switch Statements: Subclasses

- **Technique:** Create subclasses for each type code, moving the behavior dependent on the type into these subclasses.

Problem Code:

```
class Employee {
    enum Type { COMMISSIONED, HOURLY, SALARIED }
    Type type;

    double calculatePay() {
        switch (type) {
            case COMMISSIONED:
                return calculateCommissionedPay();
            case HOURLY:
                return calculateHourlyPay();
            // other cases
        }
    }
}
```

6. Switch Statements: Subclasses

Refactored Code:

```
abstract class Employee {
    abstract double calculatePay();
}

class CommissionedEmployee extends Employee {
    double calculatePay() {
        return calculateCommissionedPay();
    }
}

class HourlyEmployee extends Employee {
    double calculatePay() {
        return calculateHourlyPay();
    }
}

class SalariedEmployee extends Employee {
    double calculatePay() {
        return calculateSalariedPay();
    }
}

// Usage:
Employee employee = new HourlyEmployee();
double pay = employee.calculatePay();
```

- **Benefits:** Each subclass clearly represents a specific type of employee and its corresponding calculation method. It adheres to the Open/Closed Principle, as new types can be added without affecting existing classes.

Alternative Classes with Different Interfaces

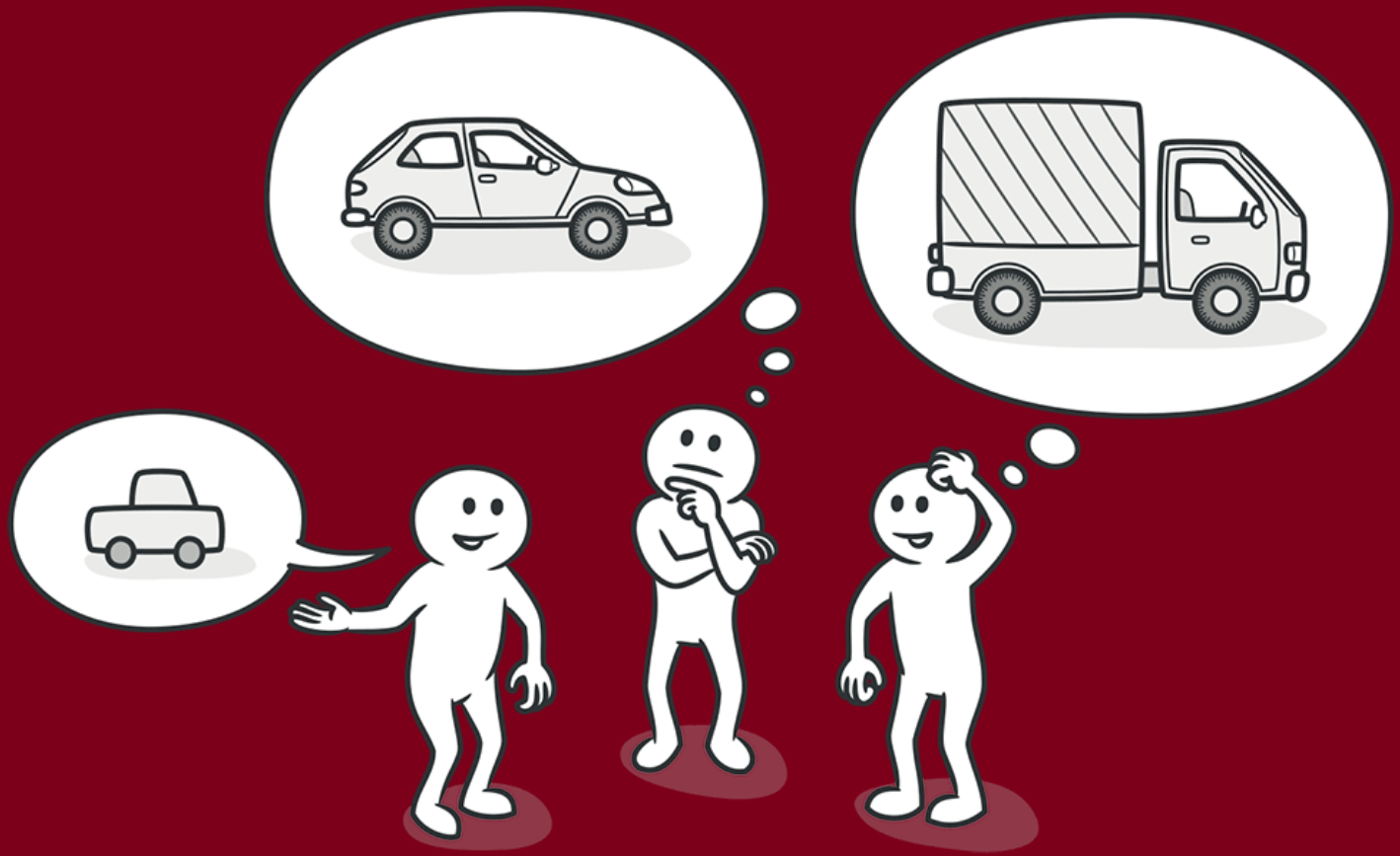


Image Credit: Refactoring.Guru

7. Alternative Classes with Different Interfaces

- **What It Is:** Two classes with identical functions but different method names.
- **Problems:** Causes unnecessary code duplication and complicates maintenance.
- **Reasons for the Problem:** Lack of awareness of existing classes leading to redundant implementations.

7. Alternative Classes with Different Interfaces

```
1  class AudioPlayer {
2      public void playSound(String file) {
3          // Implementation to play sound
4      }
5  }
6
7  class MusicPlayer {
8      public void startMusic(String track) {
9          // Implementation to play music
10     }
11 }
```

7. Alternative Classes with Different Interfaces - Solution

- **Solution Strategies:**
  - **Rename Methods:** Align method names across classes.
  - **Extract Superclass:** For partially duplicated functionality, create a common superclass.
- **Benefits:** Reduces code duplication, enhances readability, and simplifies future maintenance.

7. Alternative Classes with Different Interfaces - Solution

- Rename Methods

```
1
2  class AudioPlayer {
3      public void play(String file) {
4          // Implementation to play sound
5      }
6  }
7
8  class MusicPlayer {
9      public void play(String track) {
10         // Implementation to play music
11     }
12 }
13
```

7. Alternative Classes with Different Interfaces - Solution

- Extract Superclass

```
1
2  abstract class Player {
3      abstract void play(String source);
4  }
5
6  class AudioPlayer extends Player {
7      void play(String file) {
8          // Implementation to play sound
9      }
10 }
11
12 class MusicPlayer extends Player {
13     void play(String track) {
14         // Implementation to play music
15     }
16 }
17
```

Change Preventers

Change Preventers

- **Definition:** Code that necessitates widespread changes for a single adjustment.
- **Impact:** Increases complexity and cost of program development.
- **Characteristic:** Makes code modifications labor-intensive and interconnected.

Divergent Change

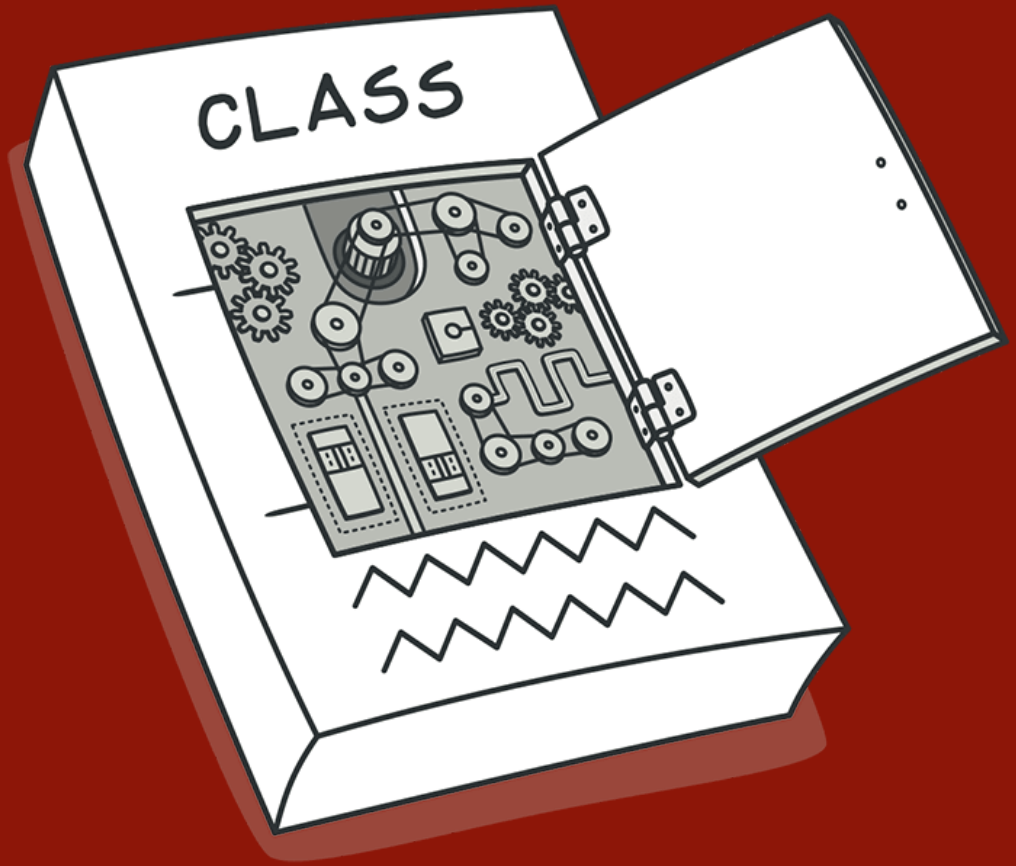


Image Credit: Refactoring.Guru

8. Divergent Change

- **What It Is:** A single class requires multiple changes for different reasons.
- **Problems:** Leads to complex, hard-to-maintain classes.

8. Divergent Change Example

```java
public class ProductManager {
    public void addProduct(Product product) {
        // Add logic
    }
    public void displayProduct(Product product) {
        // Display logic
    }
    public void orderProduct(Product product) {
        // Order logic
    }
    // More methods related to products
}
```

- **Issues:** Adding a new product type affects multiple unrelated methods.

8. Divergent Change - Solution

- **Solution Strategies:**
  - **Extract Class:** Split up responsibilities into focused classes.
  - **Use Inheritance:** Combine common behaviors using superclass or subclass extraction.

8. Divergent Change - Solution

```
// After Extract Class
public class ProductDisplay {
    public void displayProduct(Product product) {
        // Display logic
    }
}

public class ProductOrder {
    public void orderProduct(Product product) {
        // Order logic
    }
}
```

- **Benefits:** Enhances code organization, reduces duplication, simplifies maintenance.

Dispensables

Dispensables

- **Definition:** Redundant code elements that reduce clarity and efficiency.
- **Characteristics:** Includes excessive comments, duplicate or obsolete code, and underused or functionality-lacking classes.
- **Cause:** Arises from inadequate maintenance or speculative coding without future utilization.

Duplicated Code

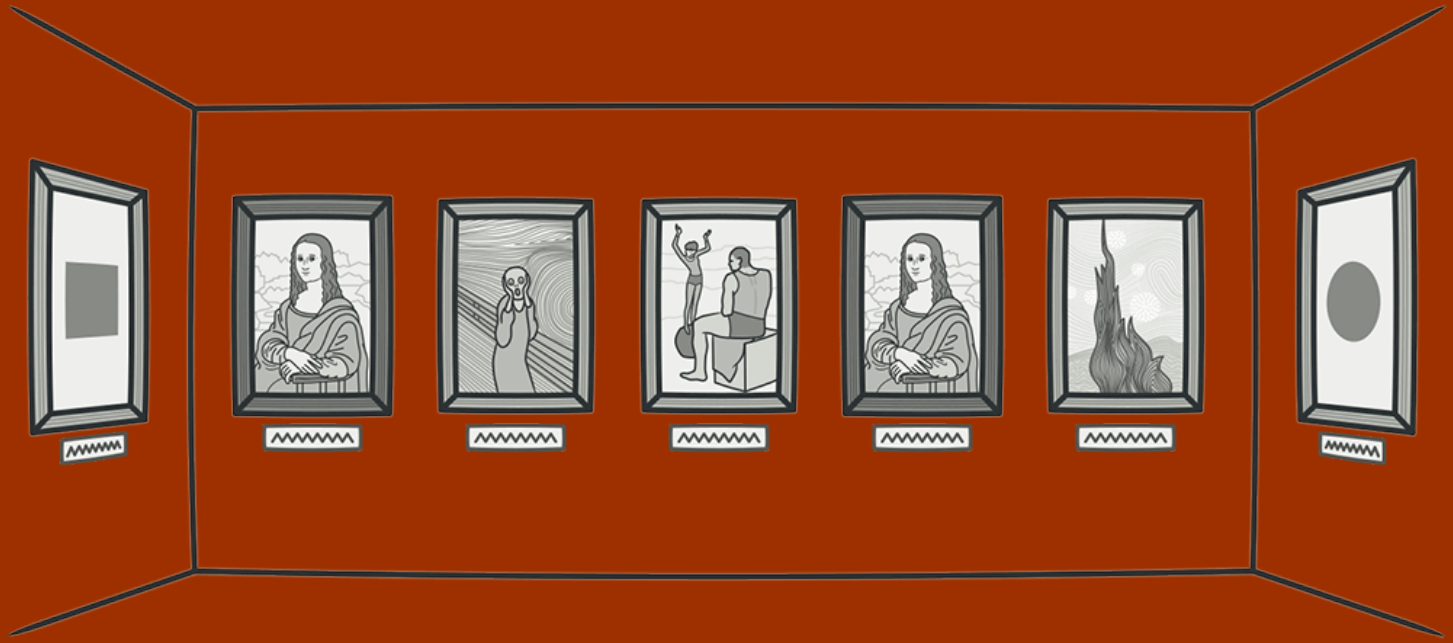


Image Credit: Refactoring.Guru

9. Duplicated Code

- **What It Is:** The same code structure appearing in more than one place.
- **Problems:** Increases the likelihood of errors and makes the code harder to maintain.

9. Example

```java
// Example of Duplicated Code
public void processOrder() {
    // ... some code ...
    double orderTotal = price * quantity;
    double tax = orderTotal * 0.05;
    double finalPrice = orderTotal + tax;
    // ... more code ...
}

public void calculateBill() {
    // ... some code ...
    double billTotal = itemPrice * itemCount;
    double tax = billTotal * 0.05;
    double finalAmount = billTotal + tax;
    // ... more code ...
}
```

9. Refactoring Duplicated Code

```java
// Extracted method to handle tax and final price calculation
public double calculateTotalWithTax(double total) {
    double tax = total * 0.05;
    return total + tax;
}

public void processOrder() {
    // ... some code ...
    double orderTotal = price * quantity;
    double finalPrice = calculateTotalWithTax(orderTotal);
    // ... more code ...
}

public void calculateBill() {
    // ... some code ...
    double billTotal = itemPrice * itemCount;
    double finalAmount = calculateTotalWithTax(billTotal);
    // ... more code ...
}
```

By extracting the common logic into a method, we reduce duplication and improve maintainability.

9. Identifying Duplicated Code

- **Identification:** Look for similar code segments across different methods or classes.
- **Examples:**
  - Copy-pasted loops or conditionals.
  - Repeated code blocks in different parts of the application.

9. Extract Method

- **Technique:** Isolate repeated code into a single method.

Problem:

```
1  class ReportGenerator {
2      void generateReport() {
3          // ... some code ...
4          System.out.println("Calculating...");
5          int sum = 0;
6          for(int i = 0; i < data.size(); i++) { sum += data.get(i); }
7          System.out.println("Sum: " + sum);
8          // ... more code ...
9          // Repeated sum calculation in another method
10     }
11 }
```

9. Extract Method

Refactored Code:

```java
class ReportGenerator {
    private int calculateSum(List<Integer> data) {
        int sum = 0;
        for(Integer value : data) { sum += value; }
        return sum;
    }

    void generateReport() {
        // ... some code ...
        System.out.println("Sum: " + calculateSum(data));
        // ... more code ...
    }
}
```

- **Benefits:** Reduces redundancy, centralizes changes, and improves code readability.

9. Pull Up Method/Field

- **Technique:** Move duplicate code to a common superclass.

Problem:

```
1  class Dog {
2      void eat() { System.out.println("Eating..."); }
3  }
4
5  class Cat {
6      void eat() { System.out.println("Eating..."); }
7  }
```

9. Pull Up Method/Field

Refactored Code:

```java
class Animal {
    void eat() { System.out.println("Eating..."); }
}

class Dog extends Animal {}

class Cat extends Animal {}
```

- **Benefits:** Eliminates duplicate code across subclasses, centralizes behavior for easier updates and maintenance.

Comments

Image Credit: Refactoring.Guru

10. Comments

- **What It Is:** Overuse of comments to explain complex or unintuitive code.
- **Problems:** Indicates potential code smells; comments often mask underlying issues.

10. Comments in Code

- **Common Issues:** Explanatory comments filling methods, suggesting the code is not self-explanatory.

```java
// Bad comments example
public class Calculator {
    // Method to add two numbers
    public int add(int a, int b) {
        // Return the sum of a and b
        return a + b; // Sum a and b
    }
}
```

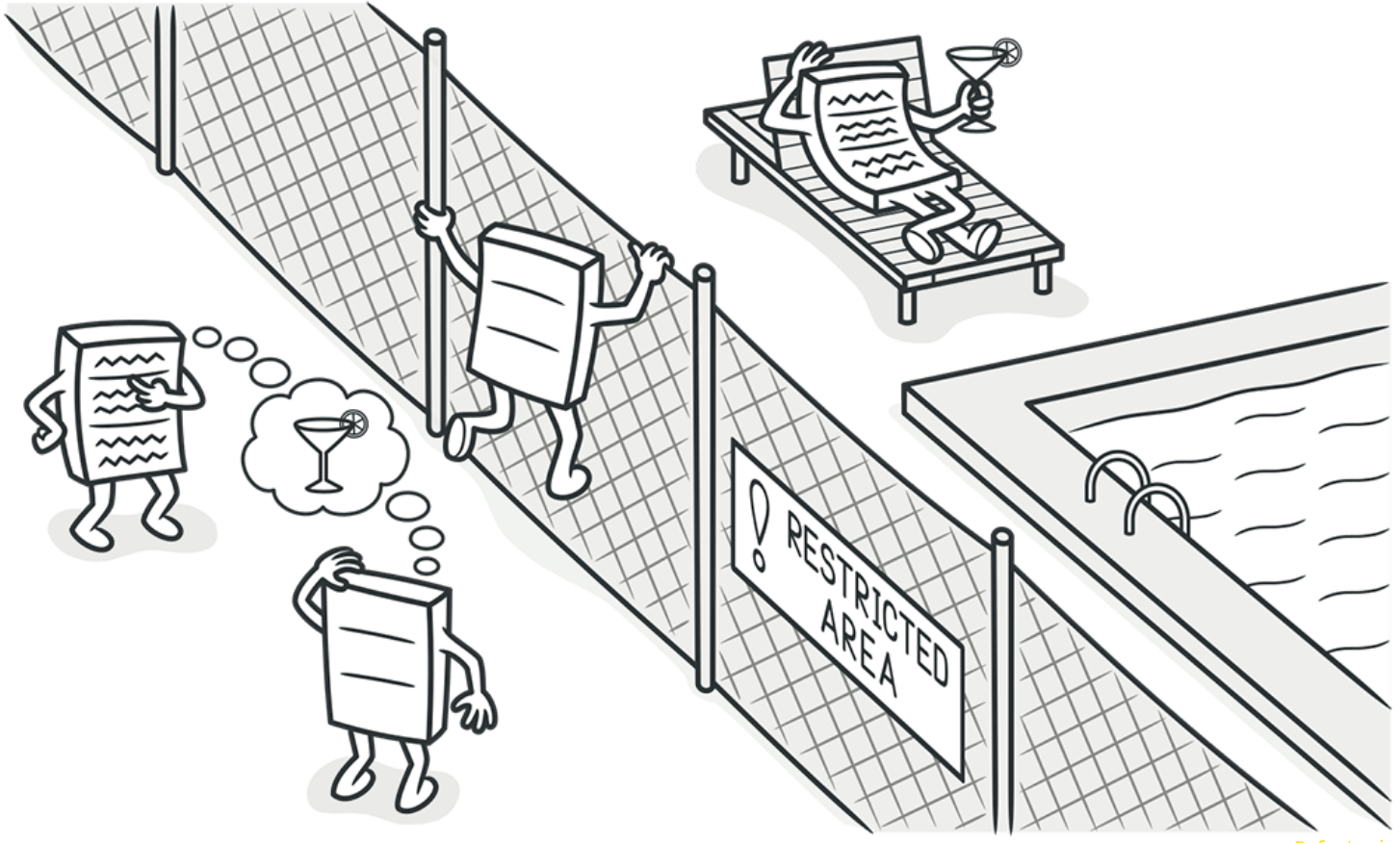
10. Comments - Solution

- **Solution Strategies:**
  - **Extract Variable:** For complex expressions, making them self-explanatory.
  - **Extract Method:** Turn commented sections into separate methods.
  - **Rename Method:** Ensure method names clearly indicate their purpose.
- **Benefits:** Results in more intuitive and maintainable code, reducing reliance on comments.

Couplers

Couplers

- **Definition:** Problems causing excessive class dependencies or overuse of delegation.
- **Characteristics:** Excessive inter-class dependencies, intrusive access, long method chains, and unnecessary delegation.
- **Cause:** Design flaws leading to tight coupling and delegation, affecting modularity and maintenance.



11. Feature Envy

- **Identification:** A method that seems more interested in a class other than the one it actually is in.
- **Example:**

```
1     public class ReportGenerator {
2         public void generateReport(Data data) {
3             // Method body heavily using methods and fields from 'Data' class
4         }
5     }
```

11. Feature Envy

- **Refactoring Techniques:**
  - **Move Method:** Move the method to the class it is most interested in.
  - **Extract Method:** If only part of the method suffers from feature envy, extract that part.

11. Feature Envy : Move Method

- **Technique:** Move the method to the class it is most interested in.

Problem Code:

```
class ReportGenerator {
    // ... other methods ...
    public void generateReport(Data data) {
        System.out.println("Report Title: " + data.getTitle());
        System.out.println("Report Data: " + data.getFormattedData());
        // Several other lines interacting with 'Data' class
    }
}

class Data {
    String getTitle() { /* ... */ }
    String getFormattedData() { /* ... */ }
    // ... other methods ...
}
```

11. Feature Envy : Move Method

Refactored Code:

```java
class ReportGenerator {
    // ... other methods ...
    public void generateReport(Data data) {
        data.printReportDetails();
    }
}

class Data {
    // ... other methods ...
    void printReportDetails() {
        System.out.println("Report Title: " + getTitle());
        System.out.println("Report Data: " + getFormattedData());
        // Moved method content here
    }
}
```

- **Benefits:** Aligns the method with the data it primarily operates on, improving cohesion and making the code more logical and easier to understand.

11. Feature Envy: Extract Method

- **Technique:** If only part of the method suffers from feature envy, extract that part.

Problem Code:

```
1  class ReportGenerator {
2      public void generateReport(Data data) {
3          // ... some code working with ReportGenerator's own data ...
4          System.out.println("Report Data: " + data.getFormattedData());
5          // ... more code working with ReportGenerator's own data ...
6      }
7  }
8
9  class Data {
10     String getFormattedData() { /* ... */ }
11     // ... other methods ...
12 }
```

11. Feature Envy: Extract Method

Refactored Code:

```java
class ReportGenerator {
    public void generateReport(Data data) {
        // ... some code working with ReportGenerator's own data ...
        printDataDetails(data);
        // ... more code working with ReportGenerator's own data ...
    }

    private void printDataDetails(Data data) {
        System.out.println("Report Data: " + data.getFormattedData());
        // Isolated the part that was showing feature envy
    }
}

class Data {
    // ... other methods ...
}
```

- **Benefits:** Separates the responsibilities more clearly, addressing the feature envy within the method by isolating the envious segment, making the code more modular and maintainable.

Message Chains

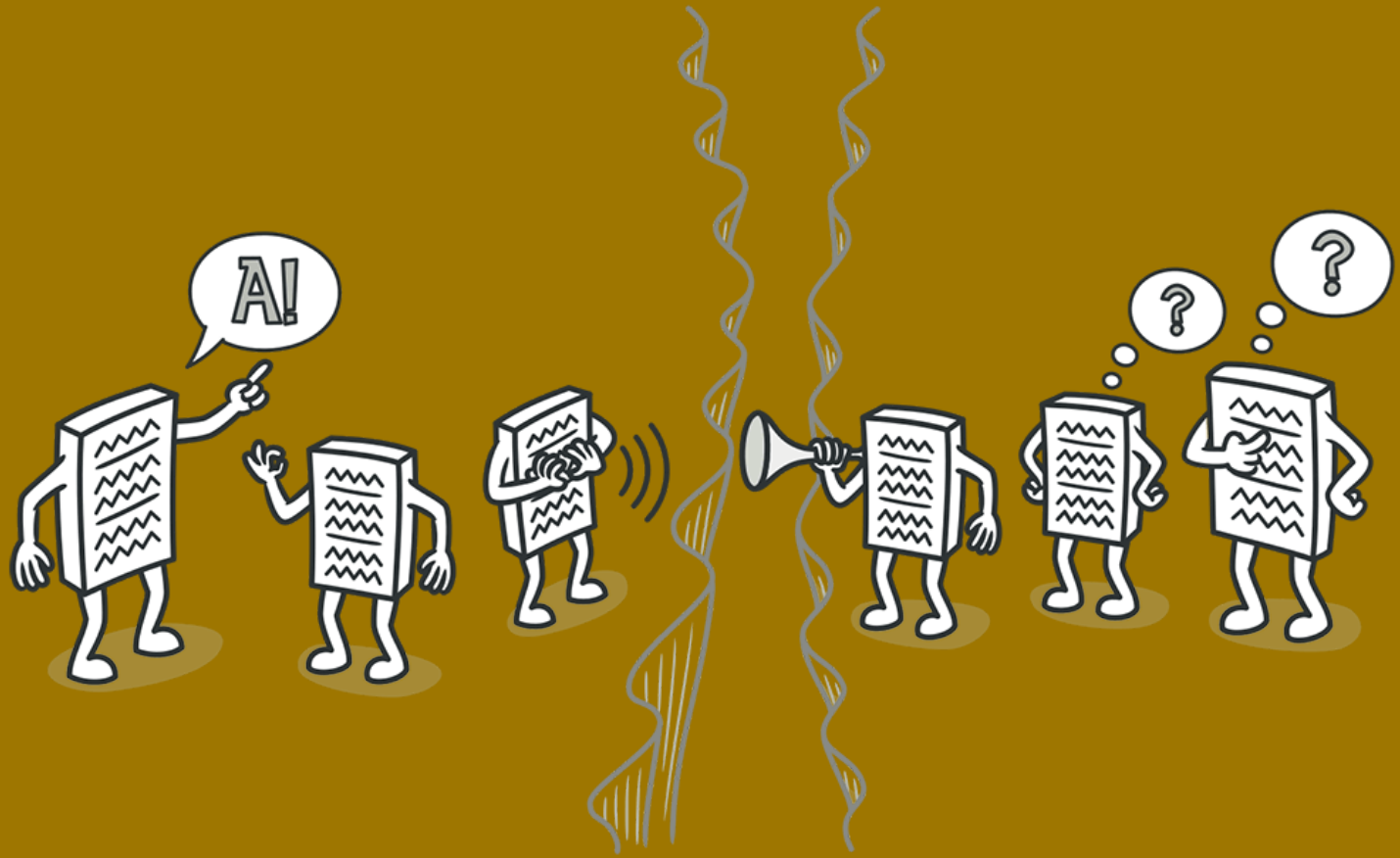


Image Credit: Refactoring.Guru

12. Message Chains

- **What It Is:** A series of method calls resembling `a->b()->c()->d()`.
- **Problems:** Increases dependency on class structure, complicating changes.

12. Message Chains Example

```
1 // Problematic message chain
2 order.getCustomer().getAddress().getZipCode();
```

- **Issues:** Client code navigates through multiple objects, creating a tight coupling.

12. Message Chains - Solution

- **Solution Strategies:**
  - **Hide Delegate:** Encapsulate the chain inside a method in the root object.
  - **Extract Method:** Isolate the chain, then move it closer to the data it manipulates.

12. Message Chains - Solution

```java
// After applying Hide Delegate
public class Order {
    public String getCustomerZipCode() {
        return this.getCustomer().getAddress().getZipCode();
    }
}
```

- **Benefits:** Simplifies client interactions, reducing direct dependencies and making code more resilient to changes.

See you tomorrow!