Refactoring and Code Smells

Dr. Ashish Sai

BCS1430

EPD150 MSM Conference Hall

Week 2 Lecture 1 & 2


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

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

Introduction to Refactoring

Introduction to Refactoring

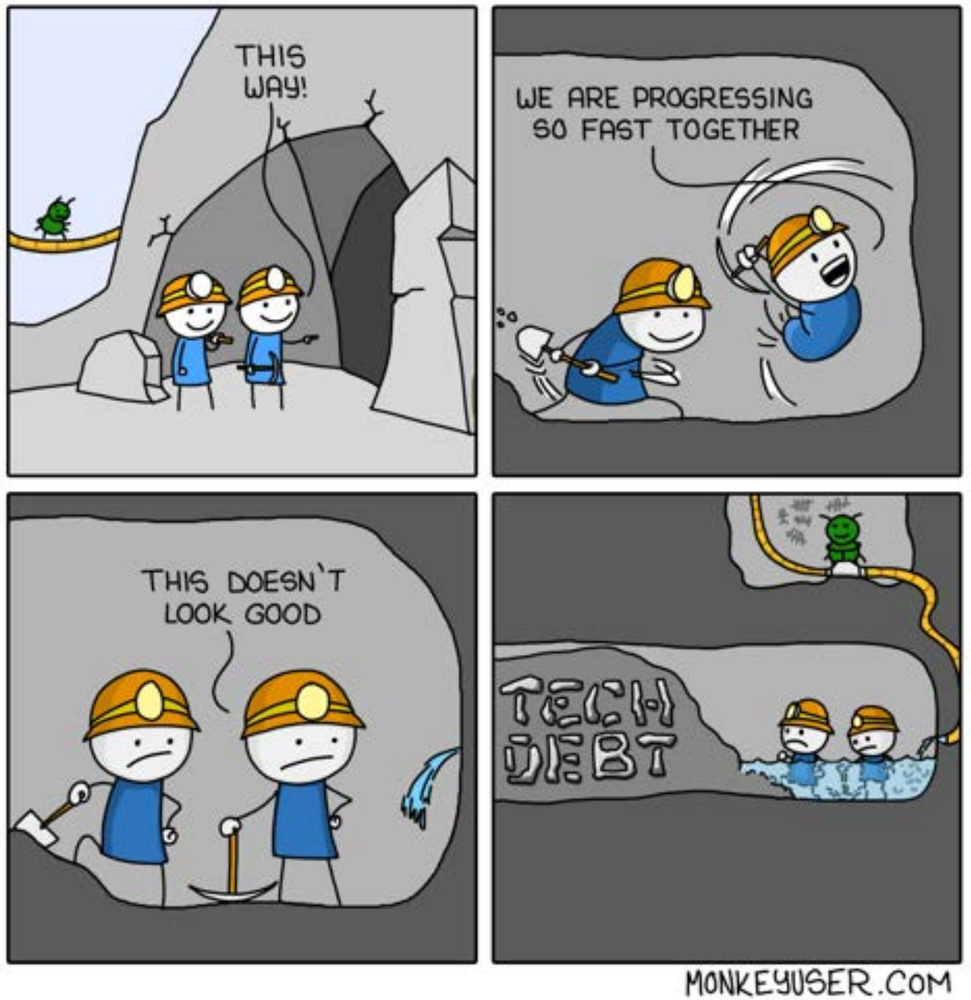
- **Refactoring**: The process of improving the internal structure of code *without* changing its external behavior.

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 might take longer.

TECH DEBT



MONKEYUSER.COM

Refactoring Challenges

- **Legacy Code**: Complex, undocumented systems
- **Time Management**: Balancing refactoring vs. new features
- **Bug Risk**: Thorough testing is crucial
- **Stakeholder Buy-In**: Convincing others of refactoring’s value

When to Refactor

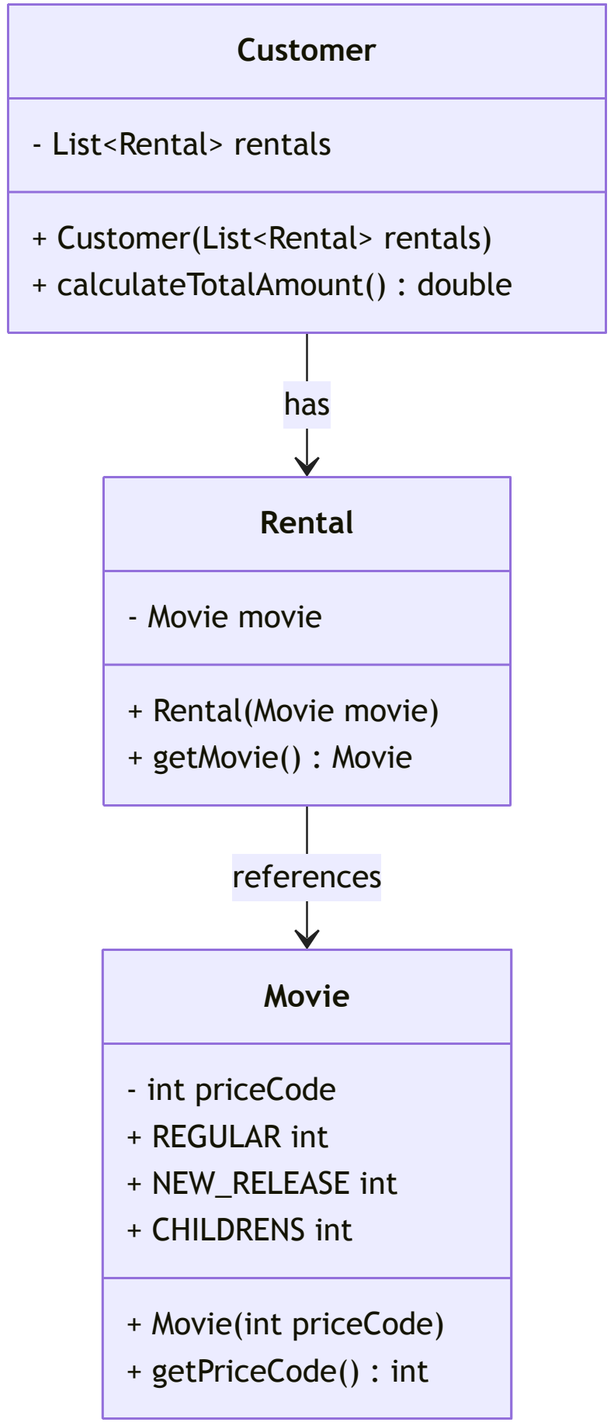
- **Before Adding a New Feature**: Clean up code first
- **When Fixing a Bug**: Simplify surrounding code
- **During Code Review**: Implement improvements immediately
- **As You Go**: Ongoing small refactorings

Refactoring Example

Refactoring

- **Small, Behavior-Preserving Transformations**: Minor changes add up to major restructurings
- **Systematic Process**: Keep external behavior consistent, improve internal architecture

The Starting Code



The Starting Code

Consider a video store’s calculation logic:

```
class Customer {
    private List<Rental> rentals;

    public double calculateTotalAmount() {
        double totalAmount = 0;
        for (int i = 0; i < rentals.size(); i++) {
            Rental rental = rentals.get(i);
            double amount = 0;

            switch (rental.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                    amount = 2;
                    break;
                case Movie.NEW_RELEASE:
                    amount = 3;
                    break;
                case Movie.CHILDRENS:
                    amount = 1.5;
                    break;
                default:
                    amount = 0;
                    break;
            }

            totalAmount += amount;
        }
        return totalAmount;
    }
}
```

Identifying the First Refactor – Extract Method

Goal: Simplify `calculateTotalAmount` by extracting smaller, more readable methods.

```
public double amountFor() {
    switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
            return 2;
        case Movie.NEW_RELEASE:
            return 3;
        case Movie.CHILDRENS:
            return 1.5;
        default:
            return 0;
    }
}
```

Refactoring Step by Step

1. **Identify** a self-contained code chunk
2. **Create a new method** in the correct class
3. **Replace old code** with the new method call
4. **Test** to ensure behavior is unchanged

Benefits of Extract Method

- **Improved Readability**
- **Reusability**
- **Separation of Concerns** (Better encapsulation)

Continuing the Refactoring

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

Polymorphism Over Conditionals

- **Before**: Switch statements for movie type
- **After**: Subclasses `RegularMovie`, `NewReleaseMovie`, `ChildrensMovie`

Implementing Polymorphism

1. **Common Interface**: All movie types respond to `getCharge`.
2. **Create Subclasses**: Each class has its own `getCharge` logic.
3. **Move Logic**: Each subclass calculates its own charge.

Example:

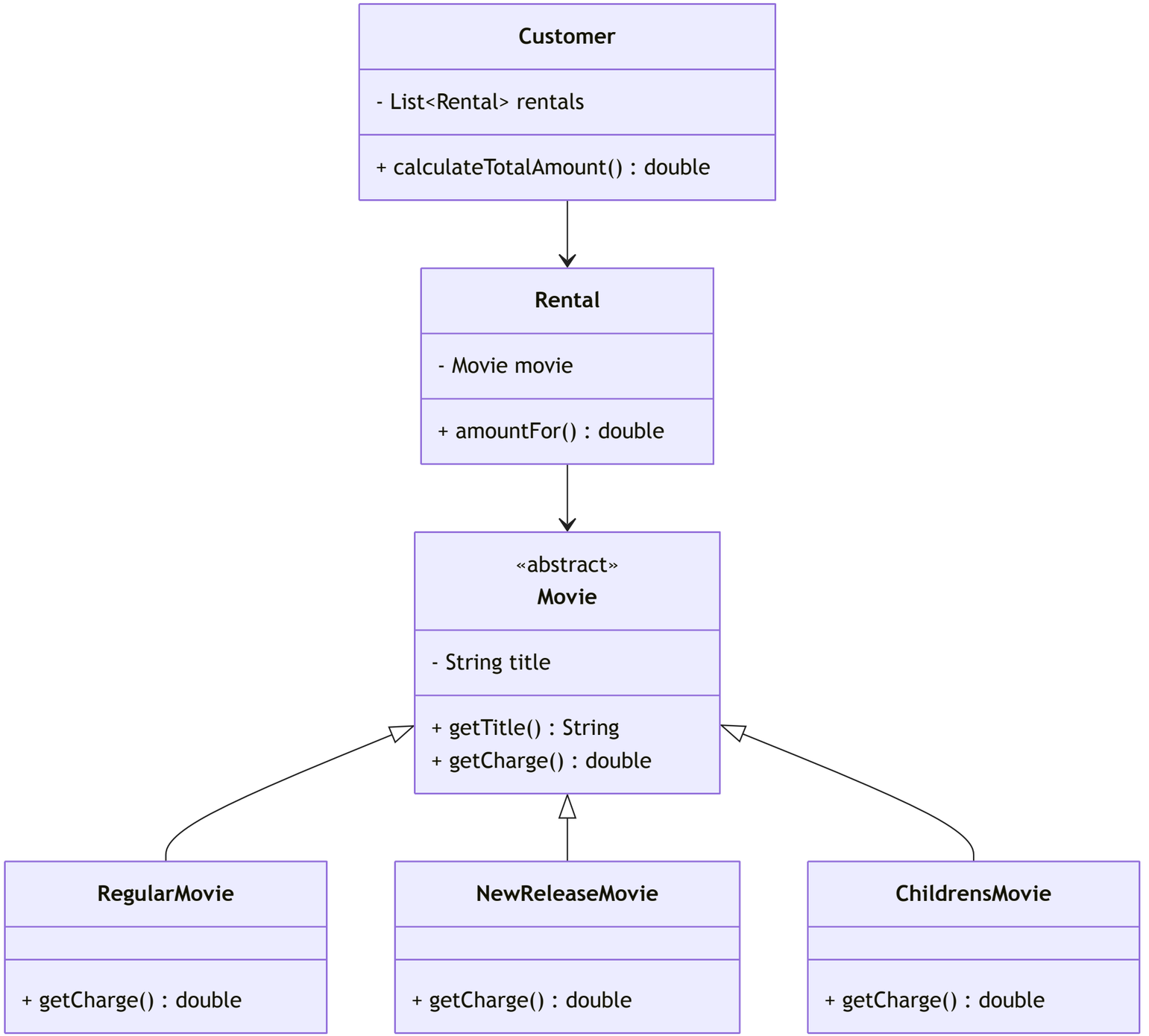
```
public abstract class Movie {
    private String title;
    public Movie(String title) { this.title = title; }
    public String getTitle() { return title; }
    abstract double getCharge();
}

public class RegularMovie extends Movie {
    public RegularMovie(String title) { super(title); }
    double getCharge() { return 2; }
}

// similarly for NewReleaseMovie, ChildrensMovie
```

Example (Rental)

```
class Rental {
    private Movie movie;
    public Rental(Movie movie) {
        this.movie = movie;
    }
    public double amountFor() {
        return movie.getCharge();
    }
}
```


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

Example (UML)

Benefits of Polymorphism

- **Flexibility**: Adding new movie types is easy
- **Open/Closed Principle**: Extend without modifying existing code

So far...

- **Encapsulation**: Complex methods are split into smaller chunks
- **Refactoring Impact**: Incremental enhancements → major improvements

Principles of Refactoring

- Don’t change observable behavior
- Make small, incremental changes
- Test after each change
- Aim for simplicity & clarity

Introduction to Code Smells

What? Code can "smell"?

Well, it does not have a nose... but it can definitely stink!

Code Smells: Indicators of potential issues that may require refactoring.

- Recognizing smells is the first step to improving code quality.
- Common smells: **Duplicated Code**, **Long Method**, **Large Class**, etc.

Use smells as a guide, not a strict rule, to identify areas for improvement.

Types of Code Smells

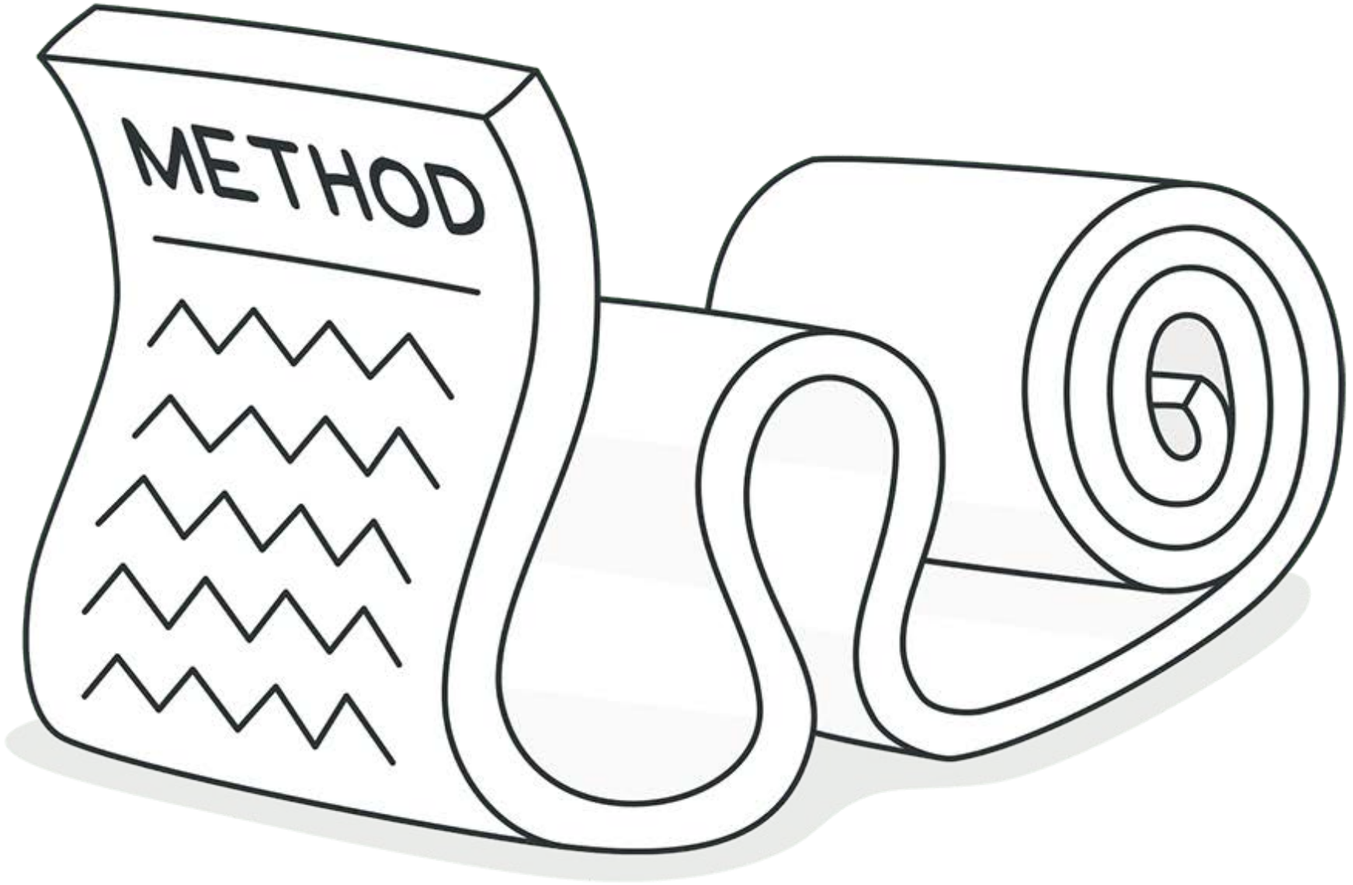
- **Bloaters**: Oversized constructs
- **Object-Orientation Abusers**: Poor OOP design
- **Change Preventers**: Code that resists modifications
- **Dispensables**: Redundant or obsolete
- **Couplers**: Excessive coupling or delegation

Bloaters

Bloaters

- **Definition**: Oversized code elements (methods/classes)
- **Grow Gradually**: Harder to maintain over time
- **Cause**: Not refining or reducing complexity

Long Method



Refactoring Guru

1. Long Method - Problem

- Methods with too many lines
- Hard to understand & maintain
- Often hide bugs

```
public class ReportGenerator {
    public void generateReport(DataSet data) {
        // Initialization and setup
        String report = "";
        // Complex data processing
        for (DataPoint point : data) {
            // Detailed data analysis and report generation
            report += analyzeDataPoint(point);
        }
        // More processing and formatting
        // Final report compilation
        System.out.println(report);
    }
}
```

1. Long Method - Solution

Break down into smaller methods, each with a clear purpose:

```
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
    }
}
```

> Smaller, specialized methods are more readable and testable.

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**: Group parameters into objects.

Extract Method

Problem Code:

```
public void printReport(List<Report> reports) {
    for(Report report : reports) {
        // Print header
        // Print details
        // Print footer
    }
}
```

Solution:

```
public void printReport(List<Report> reports) {
    for(Report report : reports) {
        printHeader(report);
        printDetails(report);
        printFooter(report);
    }
}

private void printHeader(Report report) { /*...*/ }
private void printDetails(Report report) { /*...*/ }
private void printFooter(Report report) { /*...*/ }
```

Replace Temp with Query

Problem Code:

```
public double calculateTotal() {
    double basePrice = quantity * itemPrice;
    if(basePrice > 1000) {
        return basePrice * 0.95;
    } else {
        return basePrice * 0.98;
    }
}
```

Refactored Code:

```
public double calculateTotal() {
    if(basePrice() > 1000) {
        return basePrice() * 0.95;
    } else {
        return basePrice() * 0.98;
    }
}

private double basePrice() {
    return quantity * itemPrice;
}
```

> Fewer temp variables, always up to date.

Parameter Object

Problem Code:

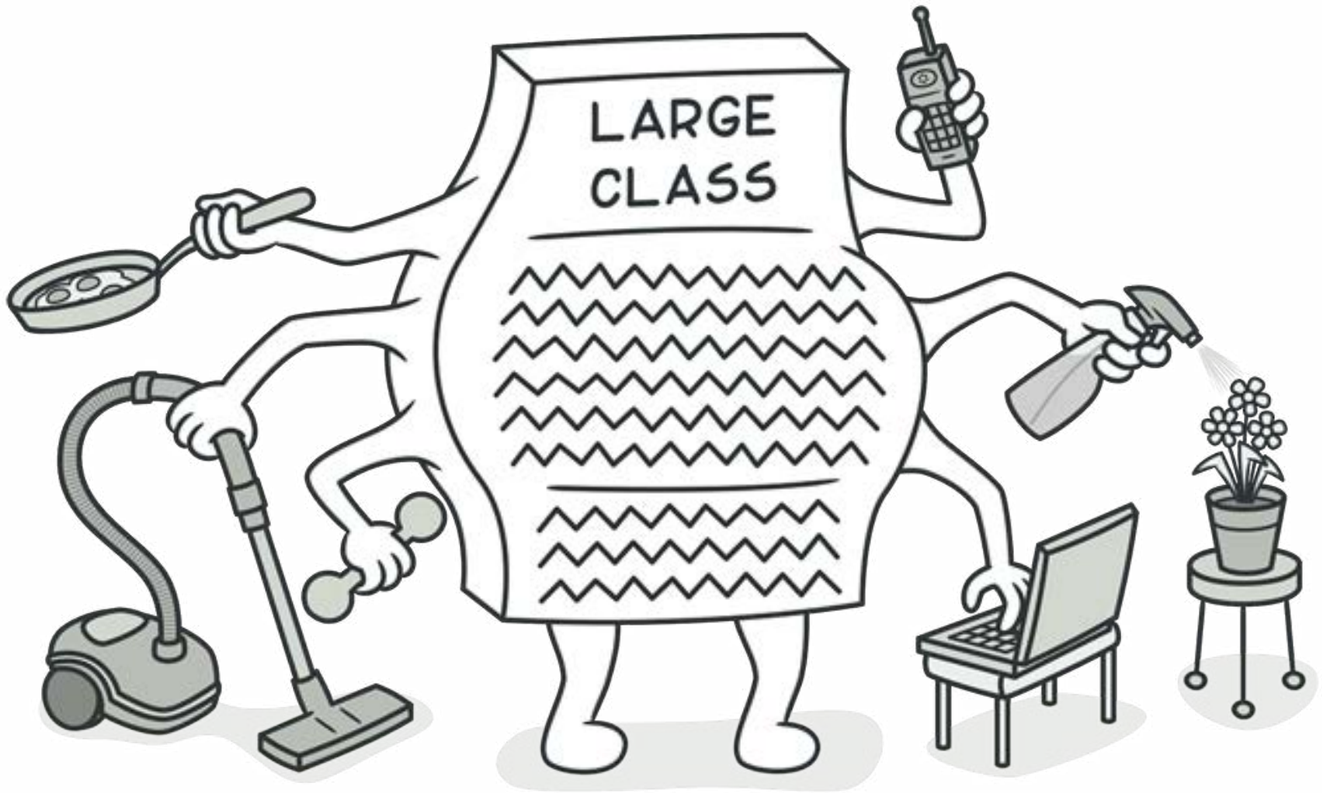
```
public void createReservation(Date start, Date end, Guest guest, Room room) {
    // Logic using start, end, guest, and room
}
```

```
public class ReservationRequest {
    private Date start, end;
    private Guest guest;
    private Room room;
    // ...
}

public void createReservation(ReservationRequest request) {
    // ...
}
```

> Group related parameters into objects.

Large Class



2. Large Class - Problem

- **Too many responsibilities**
- Hard to understand/modify

```
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
}
```

The `Employee` class manages personal details, pay calculations, and data storage.

2. Large Class - Solution

Extract Class: Split responsibilities

```
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
    }
}
```

> Each class is more focused, readable, and maintainable.

2. Large Class: Techniques

- **Extract Class**
- **Extract Subclass or Interface**

> Keep each class specialized.

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
}
```

Large Class: Extract Class

Refactored 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
}
```

Extract Subclass/Interface

Problem Code

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

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

> Each subclass handles its own variation.

Extract Subclass/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
    }
}
```

Extract Subclass/Interface

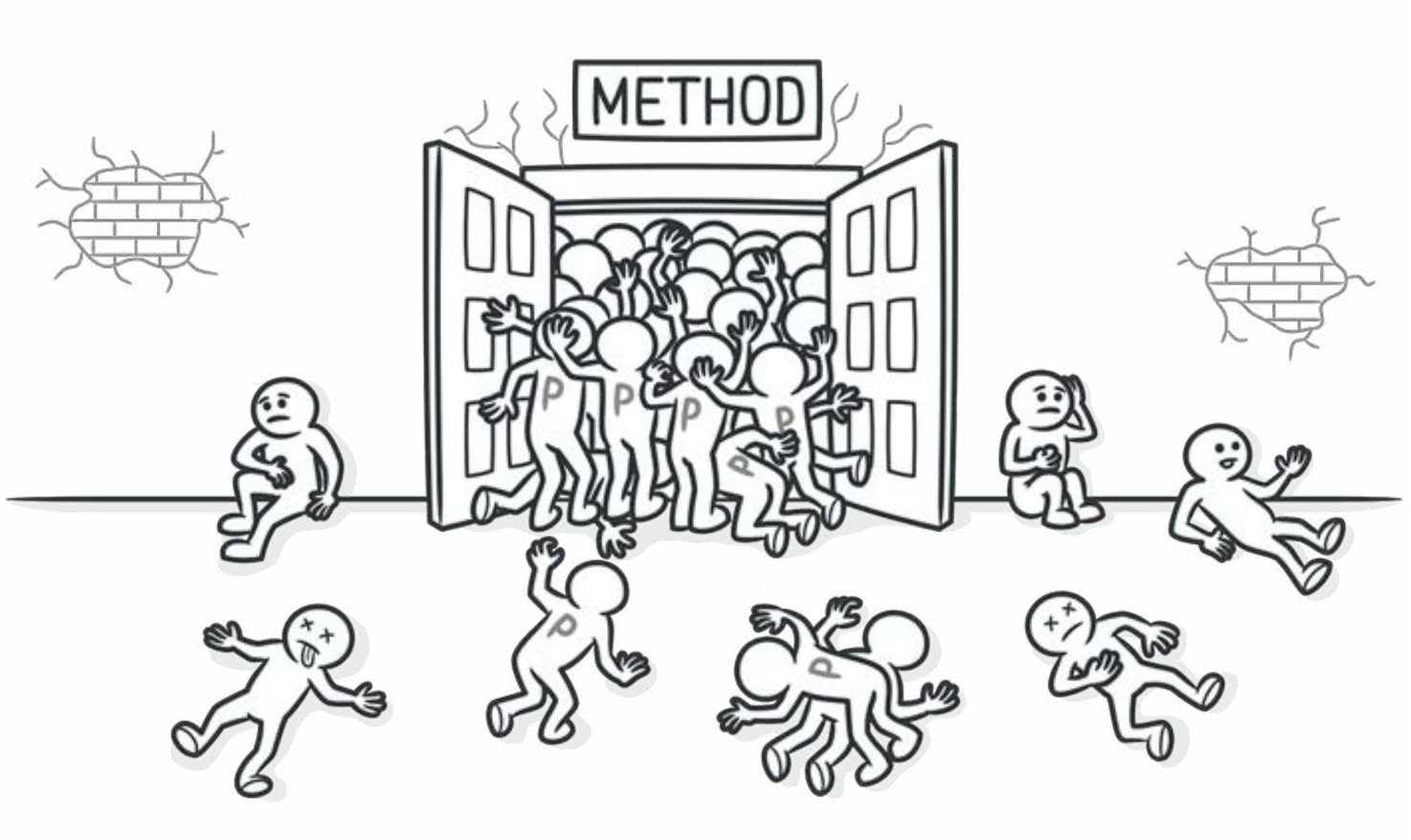
Refactored Code (Extract Interface)

```
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
    }
}
```

Long Parameter List



3. Long Parameter List

– **Too many parameters** → confusing & error-prone

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

3. Solution to Long Parameter List

Objects to group parameters or **replace params with method calls**.

```
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...
}
```

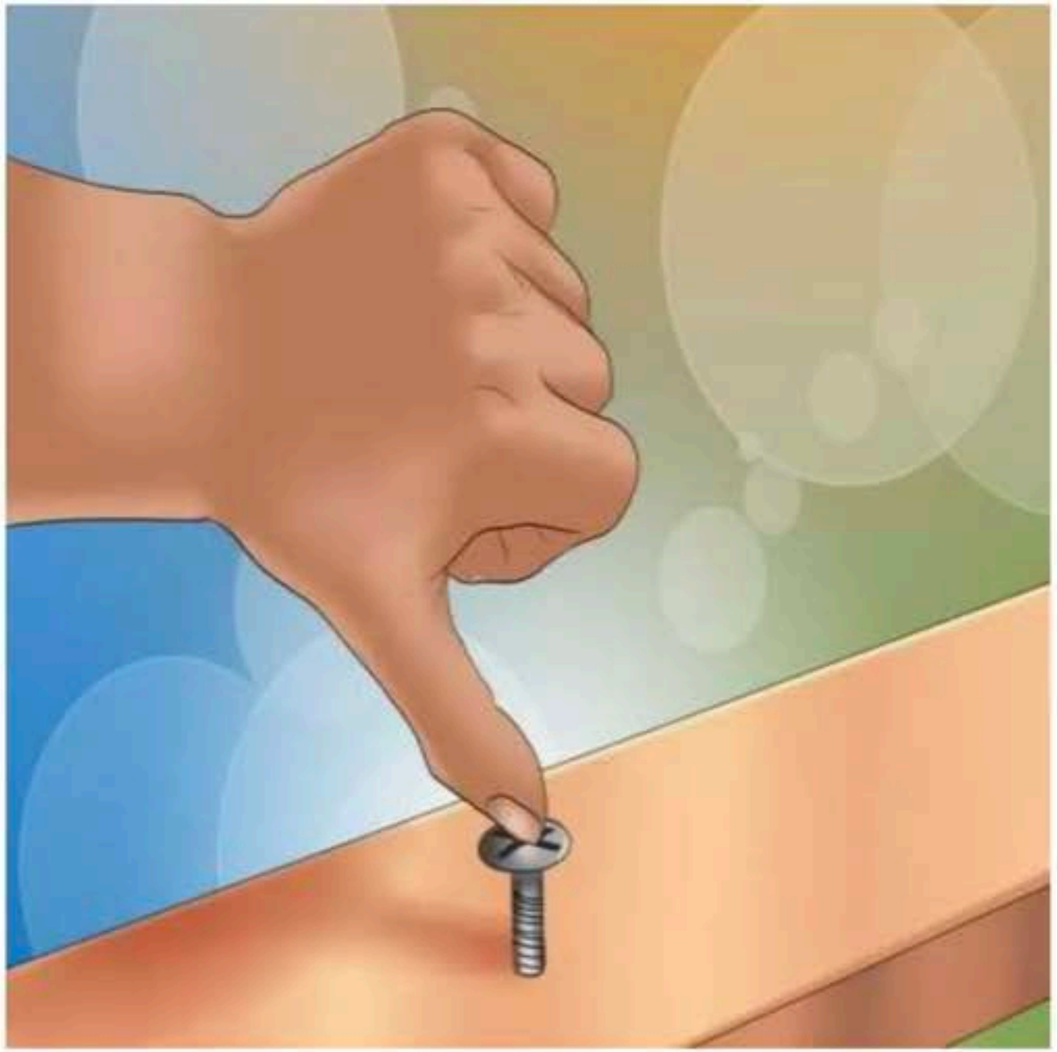
> Simplifies signatures, clarifies intent.

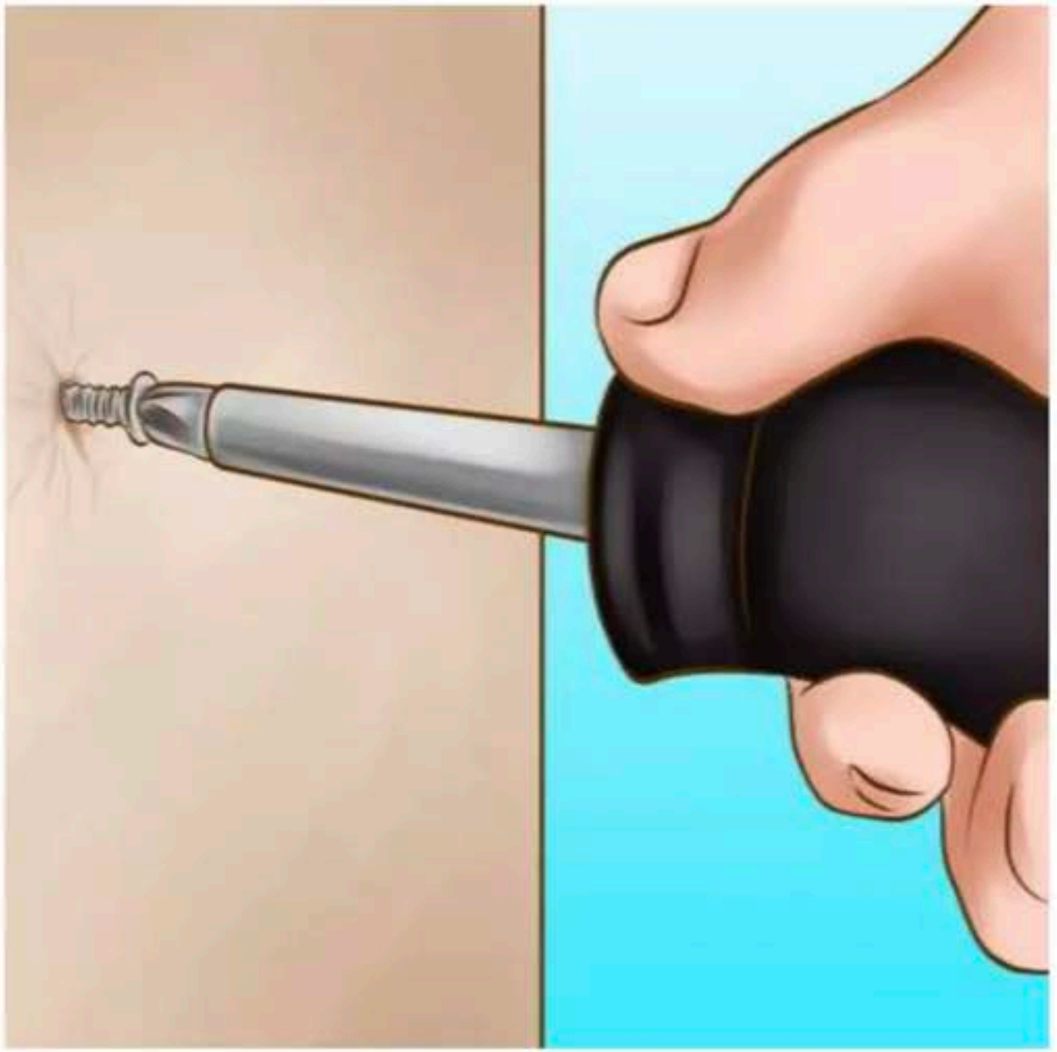
3. Alternative: Replace Parameters with Method Calls

– Retrieve needed data within the method instead of passing everything in.

```
// 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...
}
```





Primitive Obsession

4. Primitive Obsession

- **Overuse of basic types** for complex tasks
- **Refactor**: “Replace Data Value with Object,” “Replace Array with Object”

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

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

> **Encapsulates behaviors (formatting, validation) inside the object.**

4. Primitive Obsession: Data Value

Refactored Code:

```
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());
    }
}
```

4. Primitive Obsession: Arrays

Problem:

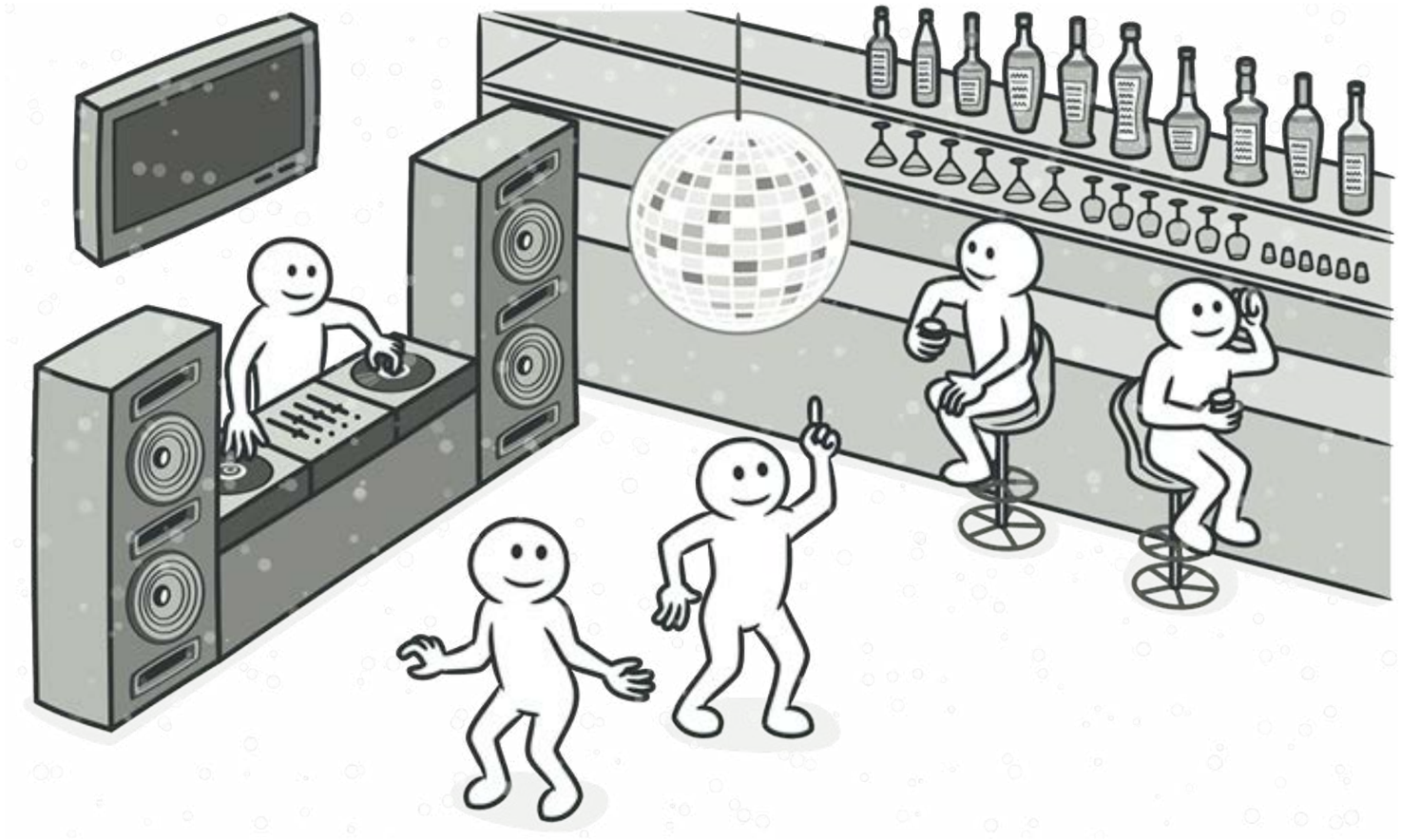
```
String[] userInfo; // [0]: name, [1]: phone, [2]: address
```

Refactor:

```
class User {
    private String name;
    private String phone;
    private String address;
    // ...
}
```

> Clearly defined fields → better readability.

Data Clumps



5. Data Clumps

- Groups of data always used together
- **Introduce Parameter Object** to bundle them

```
public void createCustomer(String firstName, String lastName, String
street, String city, String zip) { ... }
```

> **Replace with `Customer` and `Address` classes.**

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

5. Data Clumps: Parameter Object

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

class Address {
    String street, city, zip;
    // ...
}
```

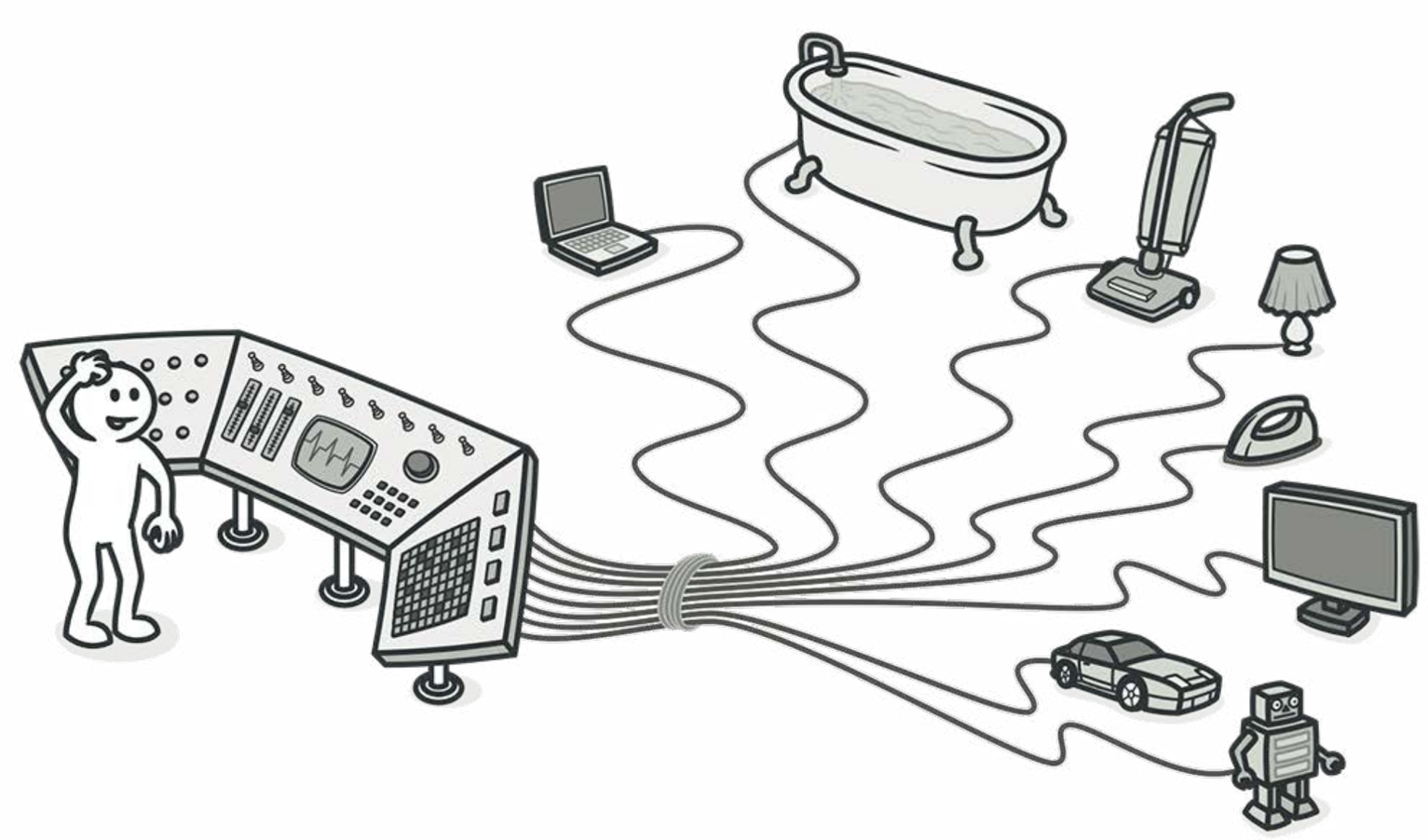
> Eliminates repetitive parameter lists, clarifies usage.

Object-Orientation Abusers

Object-Orientation Abusers

- Poor OOP design → complex, rigid code
- Often misuse inheritance/polymorphism
- Usually from misunderstanding OOP principles

Switch Statements



6. Switch Statements

- Over-reliance on switch/if-else for type-based logic
- Violates Open/Closed Principle (hard to extend)

```
public String makeSound(String animalType) {
    switch(animalType) {
        case "dog": return "Bark";
        case "cat": return "Meow";
        // ...
    }
}
```


E480
Backspace
Enter
Shift

6. Switch Statements - Solution

Replace with Polymorphism:

```
interface Animal { String makeSound(); }
class Dog implements Animal { public String makeSound() { return "Bark"; } }
class Cat implements Animal { public String makeSound() { return "Meow"; } }

public class AnimalSound {
    public String makeSound(Animal animal) {
        return animal.makeSound();
    }
}
```

> No need to modify `AnimalSound` when adding a new animal.

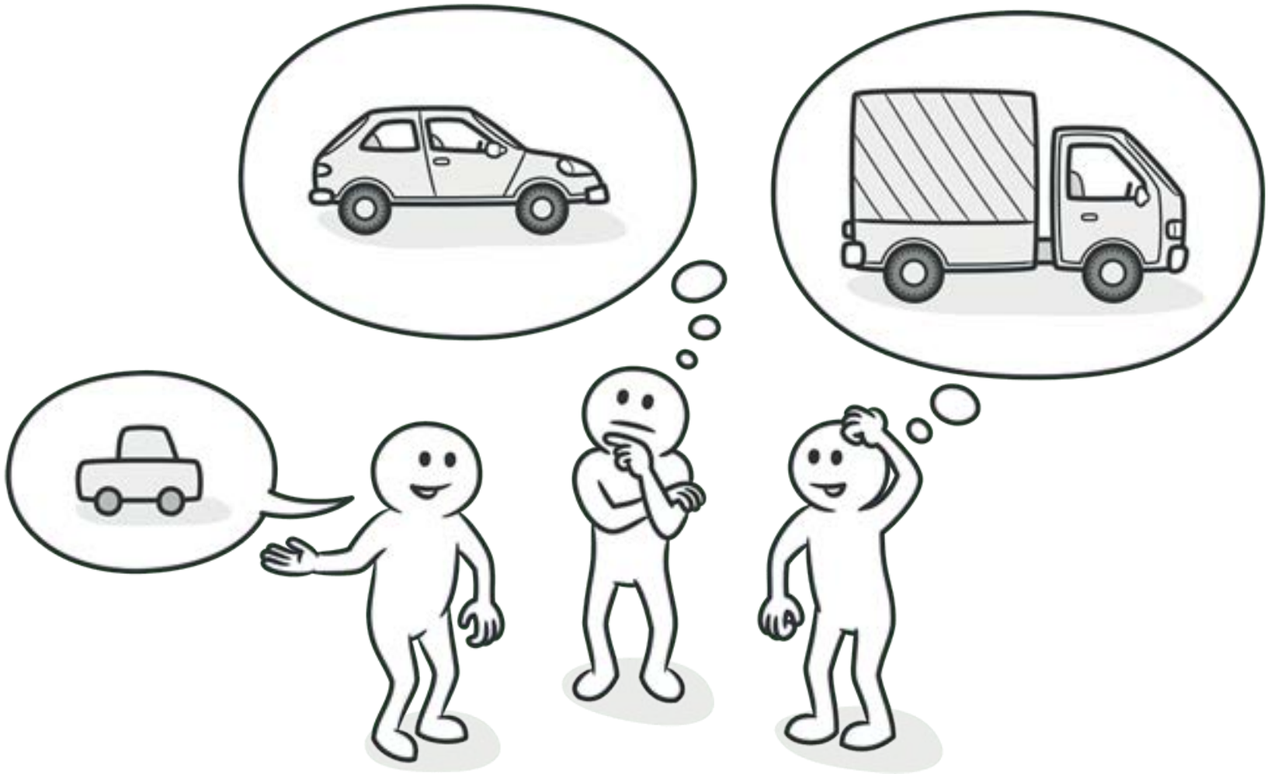
6. Techniques for Switch Statements

- **Replace Conditional with Polymorphism**
- **Replace Type Code with Subclasses**

```
abstract class Employee { abstract double calculatePay();
}
class CommissionedEmployee extends Employee { /* ... */ }
class HourlyEmployee extends Employee { /* ... */ }
```

> Each subclass implements its own behavior.

Alternative Classes with Different Interfaces



7. Alternative Classes with Different Interfaces

- Two classes do the same thing but with different method names
- **Problems**: Duplication, confusion, extra maintenance

```
class AudioPlayer { void playSound(String file) { ... }
}
class MusicPlayer { void startMusic(String track) { ...
} }
```

7. Solutions

Rename Methods or **Extract Superclass**:

```
abstract class Player { abstract void play(String
source); }
class AudioPlayer extends Player { void play(String
file){ ... } }
class MusicPlayer extends Player { void play(String
track){ ... } }
```

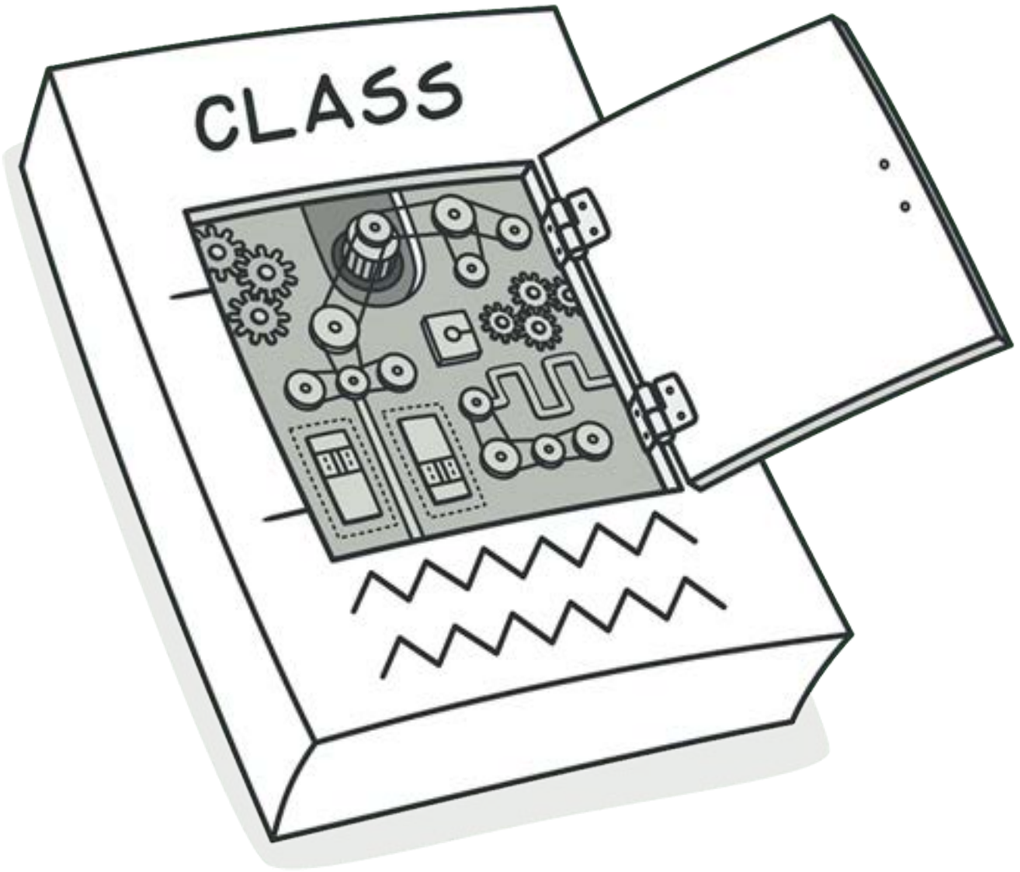
> Reduces duplication, unifies interfaces.

Change Preventers

Change Preventers

- Code requiring widespread edits for one change
- Makes modifications more expensive
- Tightly coupled or monolithic designs

8. Divergent Change



8. Divergent Change

- One class must change in multiple ways for different reasons
- Example: `ProductManager` handles add, display, order, etc.

```
public class ProductManager {
    public void addProduct(Product p) { ... }
    public void displayProduct(Product p) { ... }
    public void orderProduct(Product p) { ... }
}
```

> Each new product feature touches many unrelated methods.

8. Divergent Change - Solution

Extract Class or **Use Inheritance**:

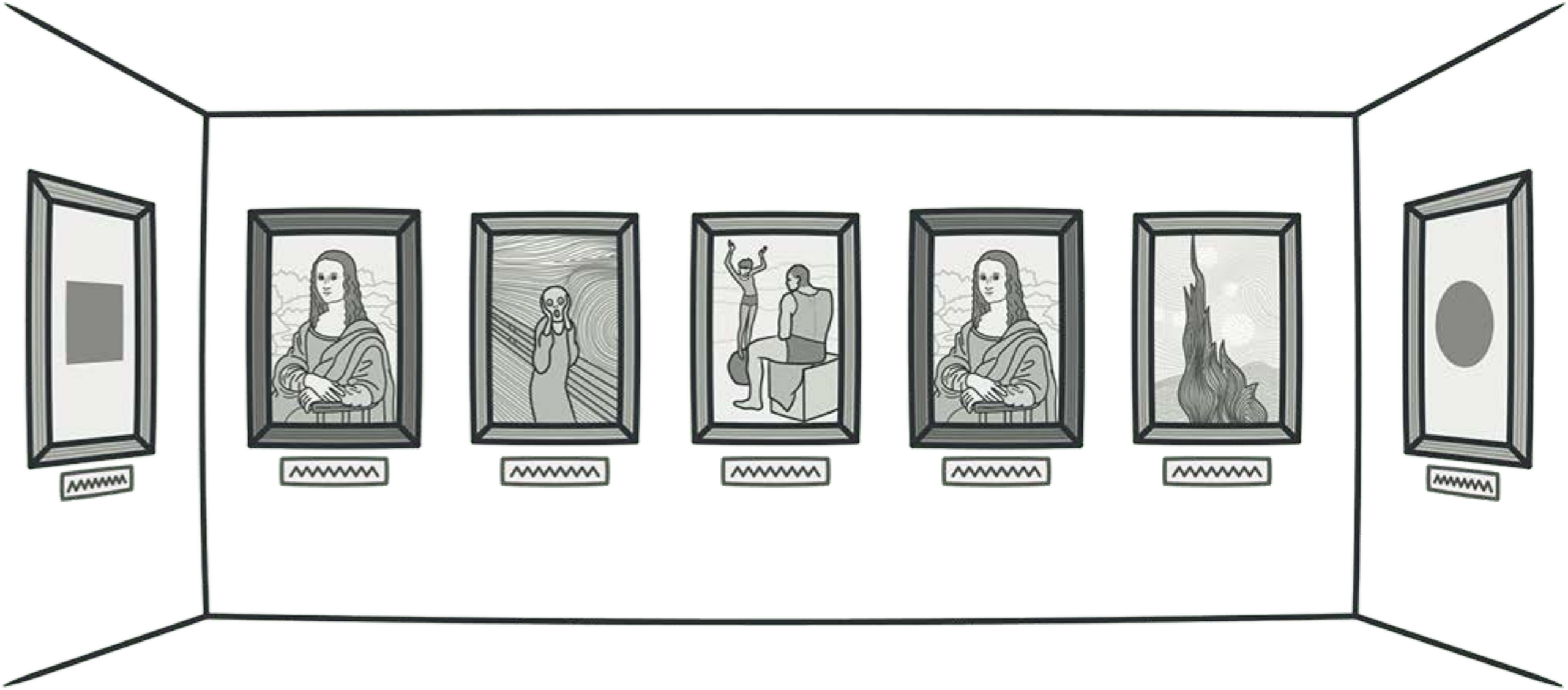
```
public class ProductDisplay {
    public void displayProduct(Product p){ ... }
}
public class ProductOrder {
    public void orderProduct(Product p){ ... }
}
```

> Separate responsibilities for better maintainability.

Dispensables

Dispensables

- Redundant code that clutters the base
- Examples: Duplicated code, pointless classes, unused methods



9. Duplicated Code

- Same logic in multiple places
- **Problems**: Harder to maintain, easy to break

```
// 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

Extract Method:

```
public double calculateTotalWithTax(double
total) {
    double tax = total * 0.05;
    return total + tax;
}
```

> One place to update if the logic changes.

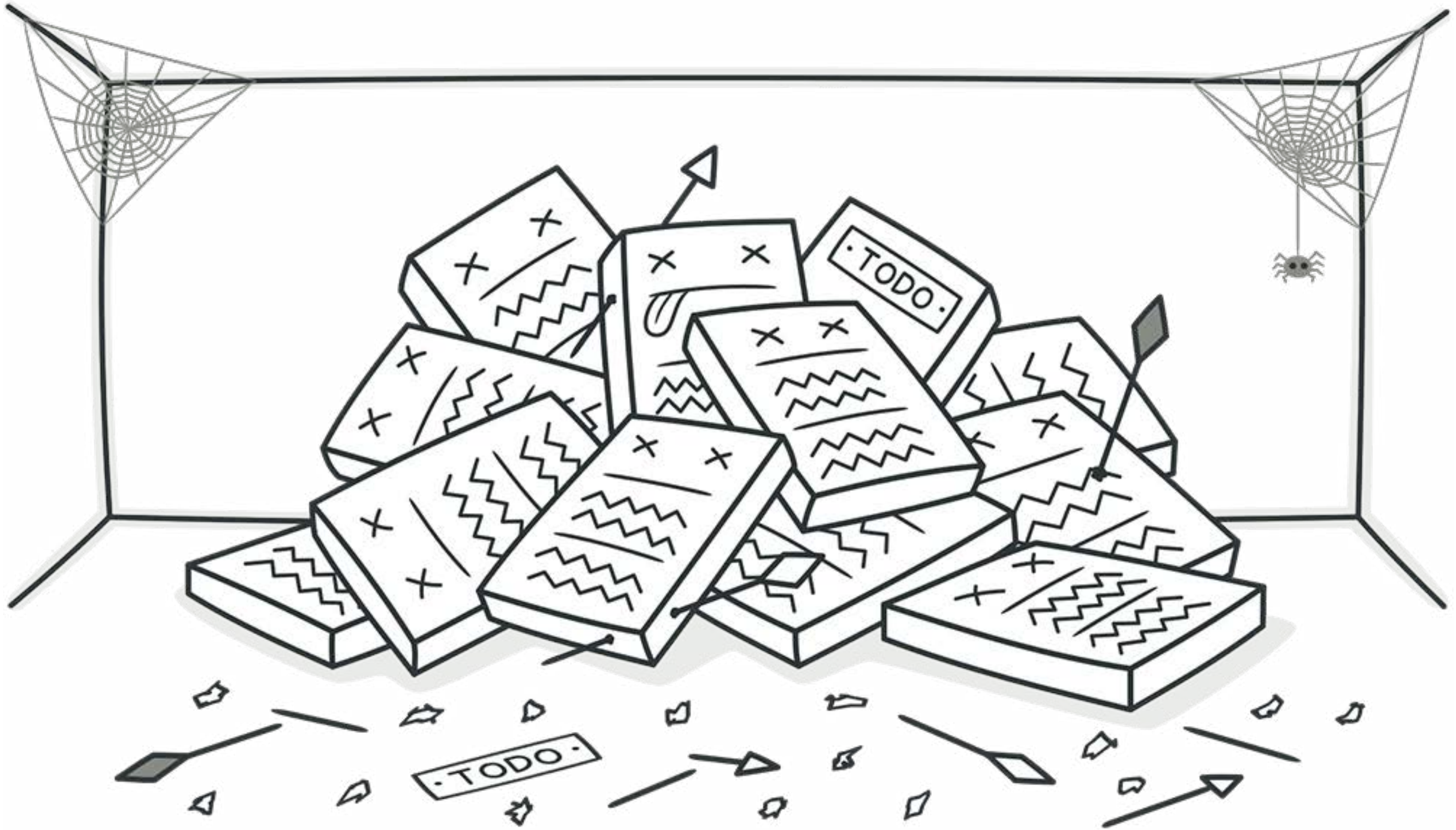
9. Identifying Duplicated Code

- Similar code blocks or loops
- **Pull Up Method/Field**: Move duplicates to a superclass.

```
class Dog { void eat() { ... } }
class Cat { void eat() { ... } }
```

Refactored:

```
class Animal { void eat() { ... } }
class Dog extends Animal { }
class Cat extends Animal { }
```


·TODO·
·TODO·

10. Comments

- Overusing comments to explain unintuitive code
- Often signals a deeper design issue

```
// This method adds two numbers
public int add(int a, int b) {
    return a + b; // sum
}
```

10. Comments - Solution

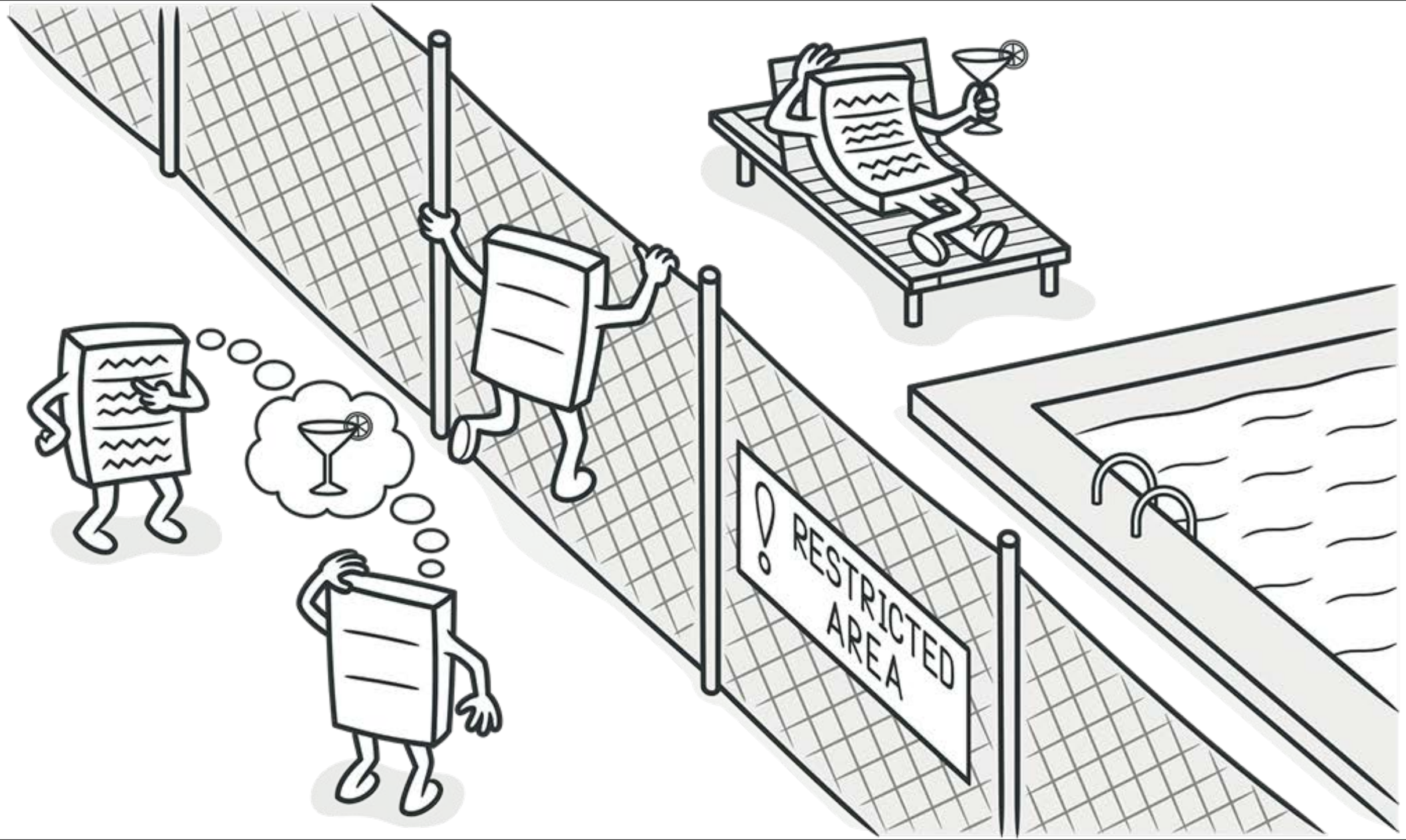
- **Extract Variable** or **Method** to clarify the code
- **Rename Methods** to be self-explanatory

> If code is clean, fewer inline comments are needed. Comments should explain *why*, not *what*.

Couplers

Couplers

- Excessive class dependencies or delegation
- Hard to maintain or modify
- Common forms: Feature Envy, Message Chains


RESTRICTED
AREA

11. Feature Envy

– A method that uses another class’s data more than its own

```
class ReportGenerator {
    public void generateReport(Data d) {
        // heavily manipulates d
    }
}
```

Fix: Move or extract the method into `Data`, or isolate the part that’s “envious.”

11. Feature Envy: Move Method

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

```
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
    }
}
```

> Align responsibilities with the data they operate on.

11. Feature Envy: Extract Method

If only part of the method suffers from feature envy, extract that part.

Problem Code:

```
class ReportGenerator {
    public void generateReport(Data data) {
        // ... some code working with ReportGenerator's own data ...
        System.out.println("Report Data: " + data.getFormattedData());
        // ... more code working with ReportGenerator's own data ...
    }
}

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

11. Feature Envy: Extract Method

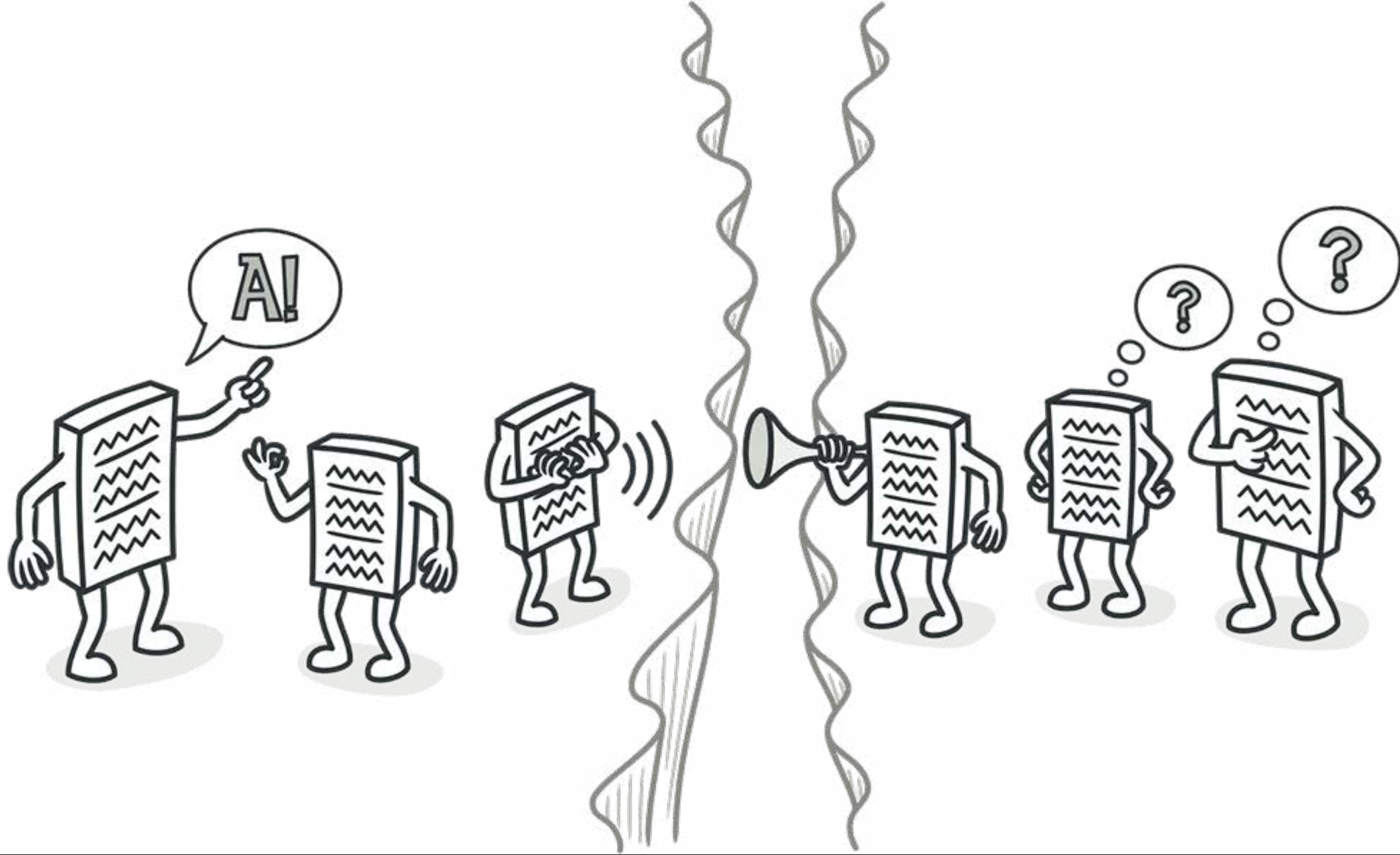
Refactored Code:

```
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 ...
}
```

> Isolate the portion that depends on `Data`.


AI!
?
?

12. Message Chains

- Long chains like `order.getCustomer().getAddress().getZipCode()`
- Tight coupling to object structure

Solution: **Hide Delegate** or **Extract Method**:

```
public class Order {
    public String getCustomerZipCode() {
        return getCustomer().getAddress().getZipCode();
    }
}
```

> Clients call `order.getCustomerZipCode()`, no longer needing to chain calls.

See you tomorrow!