Tutorial Week 3 Solutions

Table of contents

1 Multiple Choice Questions

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

Technical debt refers to the implied cost of choosing a quick and easy solution over a better approach that would take longer to implement. This debt accumulates over time and will need to be paid off through additional rework or refactoring in the future.

2. **B. By breaking it down into smaller, more focused methods**

The "Long Method" code smell can be addressed by refactoring the long method into smaller, more focused methods, each with a specific responsibility. This improves code readability, maintainability, and testability.

3. **B. It can lead to code that is hard to modify and extend**

The primary risk of the "Switch Statements" smell is that it can lead to code that is hard to modify and extend. As new cases are added, the switch statement becomes more complex and harder to maintain.

4. **A. A cluster of data that always appears together but isn't organized into a structure**

A "Data Clump" is a cluster of data (variables or fields) that always appear together but aren't organized into a proper data structure or class. This violates the principle of encapsulation and can make the code harder to maintain.

2 Short Answer Questions

1. **Feature Envy** can be identified when a method or class is excessively using data or methods from other classes, indicating that it may belong better in those other classes. To refactor, the relevant functionality can be moved to the class that contains the data or methods being used, promoting better encapsulation and cohesion.
2. The **Long Method** code smell can negatively impact code quality by making the code harder to read, understand, and maintain. Long methods often have multiple responsibilities, which violates the Single Responsibility Principle. A typical way to refactor a long method is to break it down into smaller, more focused methods, each with a single responsibility. This improves code readability, maintainability, and testability.

3 Long Answer Questions

1. The `printInvoices` method in the `InvoiceProcessor` class exhibits several code smells:

   - **Primitive Obsession**: The method uses primitive data types (e.g., `String`) for representing complex data structures like addresses.
   - **Long Method**: The method is too long and has multiple responsibilities, such as printing invoices and formatting addresses.
   - **Feature Envy**: The method accesses data from other classes (`Customer` and `Address`) extensively, indicating that some of its functionality might belong in those classes.
   - **Duplicated Code**: The code to format the address is duplicated for each invoice.

   Improvements:

   - Introduce separate classes to represent complex data structures like `Address`.
   - Break down the `printInvoices` method into smaller, more focused methods, each with a single responsibility (e.g., `printInvoice`, `formatAddress`).
   - Move address formatting logic to the `Address` class, promoting better encapsulation and cohesion.
   - Consider using a `StringBuilder` for string concatenation instead of repeated string operations.

2. The `OrderCalculator` class exhibits the following code smells:

- **Duplicated Code**: The logic for calculating the total is duplicated in the `calculateOrderTotal` and `calculateDiscountedTotal` methods.
- **Primitive Obsession**: The class uses primitive arrays (`int[]`) to represent product IDs and quantities, which can lead to errors and lacks expressiveness.

Refactoring:

- Extract the common logic for calculating the total into a separate method to eliminate duplication.
- Introduce a `Product` class to represent product information (ID, price, quantity) instead of using primitive arrays.
- Create a `ShoppingCart` class to manage the collection of products and their quantities.
- Move the discount calculation logic to the `ShoppingCart` class, promoting better encapsulation and cohesion.

Example refactored code:

```
public class Product {
    private int id;
    private double price;
    private int quantity;

    // Constructor, getters, and setters
}

public class ShoppingCart {
    private List<Product> products;

    public double calculateTotal() {
        double total = 0;
        for (Product product : products) {
            total += product.getPrice() * product.getQuantity();
        }
        return total;
    }

    public double calculateDiscountedTotal(double discountRate) {
        double total = calculateTotal();
        return total - (total * discountRate);
    }

    // Add methods to manage products in the cart
```

```
}

public class OrderCalculator {
    private ShoppingCart cart;

    public OrderCalculator(ShoppingCart cart) {
        this.cart = cart;
    }

    public double calculateOrderTotal() {
        return cart.calculateTotal();
    }

    public double calculateDiscountedTotal(double discountRate) {
        return cart.calculateDiscountedTotal(discountRate);
    }
}
```