Code Refactoring and UML

Table of contents

1 Tutorial Sheet: Refactoring and Code Smells **1**
1.1 Multiple Choice Questions 1
1.2 Short Answer Questions 2
1.3 Long Answer Questions 2

1 Tutorial Sheet: Refactoring and Code Smells

1.1 Multiple Choice Questions

- 1. **Which of the following best describes 'Technical Debt'?**
  - A. The time taken to write technical documentation
  - B. The financial cost of purchasing new software tools
  - C. The implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer
  - D. The expenses related to hiring technical staff
- 2. **How can the 'Long Method' code smell be addressed?**
  - A. By adding more comments to explain the code
  - B. By breaking it down into smaller, more focused methods
  - C. By moving it to a different class
  - D. By rewriting the entire method from scratch
- 3. **What is the primary risk of the 'Switch Statements' smell in code?**
  - A. It makes the code easier to read
  - B. It can lead to code that is hard to modify and extend
  - C. It improves the performance of the code

- D. It helps in making the code more modular

- 4. **What is a 'Data Clump' in code smells?**

  - A. A cluster of data that always appears together but isn't organized into a structure
  - B. A collection of unrelated data types
  - C. A large amount of data processed by a single method
  - D. A group of variables that are frequently modified together

1.2 Short Answer Questions

1. **Explain how 'Feature Envy' can be identified and refactored in a codebase.**
2. **How can the 'Long Method' code smell negatively impact code quality, and what is a typical way to refactor it?**

1.3 Long Answer Questions

1. Identify the code smells in the InvoiceProcessor's printInvoices method and suggest improvements.

```
public class InvoiceProcessor {
    private List<Invoice> invoices;

    public InvoiceProcessor(List<Invoice> invoices) {
        this.invoices = invoices;
    }

    public void printInvoices() {
        for (int i = 0; i < invoices.size(); i++) {
            System.out.println("Invoice ID: " +
             ↪  invoices.get(i).getId());
            System.out.println("Customer: " +
             ↪  invoices.get(i).getCustomer().getName());
            System.out.println("Address: " +
             ↪  invoices.get(i).getCustomer().getAddress().getStreet()
                                + ", " +
                                 ↪  invoices.get(i).getCustomer().getAddress().getCity()
                                + ", " +
                                 ↪  invoices.get(i).getCustomer().getAddress().getZipCode()
```

```
            System.out.println("Total: " + invoices.get(i).getAmount());
            System.out.println("Due Date: " +
             ↪  invoices.get(i).getDueDate());
            System.out.println("-----------------------------------");
        }
    }
}

class Invoice {
    private String id;
    private Customer customer;
    private double amount;
    private String dueDate;

    // Constructor, getters, and setters
}

class Customer {
    private String name;
    private Address address;
    // Constructor, getters, and setters
}

class Address {
    private String street;
    private String city;
    private String zipCode;
    // Constructor, getters, and setters
}
```

2. How can you refactor the OrderCalculator class to improve the handling of product IDs and quantities?

```
public class OrderCalculator {
    public double calculateOrderTotal(int[] productIds, int[]
     ↪  quantities) {
        double total = 0;
        for (int i = 0; i < productIds.length; i++) {
            double price = getProductPrice(productIds[i]);
```

```
            total += price * quantities[i];
        }
        return total;
    }

    public double calculateDiscountedTotal(int[] productIds, int[]
     ↪ quantities, double discountRate) {
        double total = 0;
        for (int i = 0; i < productIds.length; i++) {
            double price = getProductPrice(productIds[i]);
            total += price * quantities[i];
        }
        return total - (total * discountRate);
    }

    private double getProductPrice(int productId) {
        // Returns the price based on the product ID
        return 0; // Simplified for this example
    }
}
```