Homework 5 - CSE 464
Bhavya Patel - ASU ID: 1225740997

Q1. a) Smelly code 1:

```
public static int x = 0;
```

The variable x is declared as public static, meaning it is accessible and modifiable from anywhere in the program. This violates the Object-Oriented principle of Encapsulation. And handleNumber modifies this global variable, causing common coupling.

Smelly code 2:

```
public static void handleNumber(String label, int num, boolean
flag, int mult, String p1, String p2)
```

The handleNumber method requires 6 different parameters, causing a long parameter list. A method requiring this many arguments is difficult to read and prone to errors like passing arguments in the wrong order.

Smelly code 3:
The main method contains hardcoded literals such as 10, 5, "Big", "Medium", "A", "B", 2, and 3. These are "magic values" with no context explaining why those specific numbers or strings are chosen. If these criteria need to change later, a developer must hunt through the logic to find every instance rather than changing a single named constant.

b) Suggested fix for the smelly code 1:
Remove the static keyword from the variable x and the handleNumber method. Move x into the main method as a local variable to enforce encapsulation. Alternatively, restructure new methods to return an integer result rather than modifying a void external variable.

Suggested fix for the smelly code 2:
The complex handleNumber method was refactored into two specialized methods doing separate tasks to eliminate the boolean control flag. To reduce the parameter list, compute string values locally to minimize the number of parameters.

Suggested fix for the smelly code 3:
Replace hardcoded numbers and strings (10, 5, 2, 3, "Big", "Medium", "Small", "Even", "Odd") with named constants to give semantic meaning and make future changes localized to one declaration.

c) i)

```
import java.util.*;
public class hw5modified {
    // Eliminate magic numbers with named constants
    private static final int THRESHOLD_BIG = 10;
    private static final int THRESHOLD_MED = 5;
```

```
    private static final int DEFAULT_MULTIPLIER = 2;
    private static final int DEFAULT_ADDEND = 3;
    private static final String LABEL_BIG = "Big";
    private static final String LABEL_MED = "Medium";
    private static final String LABEL_SMALL = "Small";
    private static final String LABEL_EVEN = "Even";
    private static final String LABEL_ODD = "Odd";

    // Constants for strings used in length calculation
    private static final String A = "A";
    private static final String B = "B";
    private static final String X = "X";
    private static final String Y = "Y";

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.println("Enter number:");
        int n = s.nextInt();
        // Variable x is now a local variable
        int x = 0;
        //Simplify parameter list
        int adjustmentAB = (A + B).length();

         if (n > THRESHOLD_BIG) {
             //Call specific method and uses named constant
             x = MultiplyHandler(LABEL_BIG, n, DEFAULT_MULTIPLIER,
adjustmentAB);
         }
         else if (n > THRESHOLD_MED) {
             x = AddHandler(LABEL_MED, n, DEFAULT_ADDEND,
adjustmentAB);
         }
         else {
             x = AddHandler(LABEL_SMALL, n, DEFAULT_ADDEND,
adjustmentAB);
        }

        int adjustmentXY = (P1_XY + P2_XY).length();
         if (n % 2 == 0) {
             x = MultiplyHandler(LABEL_EVEN, n,
DEFAULT_MULTIPLIER, adjustmentXY);
        } else {
             x = AddHandler(LABEL_ODD, n, DEFAULT_ADDEND,
adjustmentXY);
```

```
        }

        System.out.println("Final result: " + x);
    }

    //Helper method specifically for mutliplication logic and
returns value back to x
    public static int MultiplyHandler(String label, int num, int
multiplier, int adjustment) {
        System.out.println(label + " number!");
        int result = (num * multiplier) + adjustment;
        System.out.println("Intermediate result: " + result);
        return result;
    }

    //Helper method specifically for addition logic and returns
value back to x
    public static int AddHandler(String label, int num, int
addend, int adjustment) {
        System.out.println(label + " number!");
        int result = (num + addend) + adjustment;
        System.out.println("Intermediate result: " + result);
        return result;
    }
}
```

ii) https://drive.google.com/file/d/1bAifVCVRADACqaAysUPv-tgtZElvIY8S/view?usp=sharing

Q2. a) Cohesion issues: The Hospital Management System class has extremely low cohesion. It aggregates entirely unrelated responsibilities—such as patient demographics, financial transactions, medical lab results, and inventory management—into a single class. There is no logical relationship between restocking medication and registering a patient, other than that they happen in a hospital. Fields for patient, billing, lab, and inventory live in the same scope, so changes in any feature force edits to the same class, creating low cohesion

Testability: Testability is poor because you cannot isolate specific functionalities. To test the billing logic, you must instantiate the entire massive system object, potentially setting up unrelated state (like pharmacy inventory) just to run a payment test. If a bug exists in the lab reporting code, it could prevent the billing tests from compiling or running.

Code quality issues: The design violates the SRP of SOLID design. Code changes to one module (e.g., updating the Pharmacy logic) require modifying the same file used by Billing and

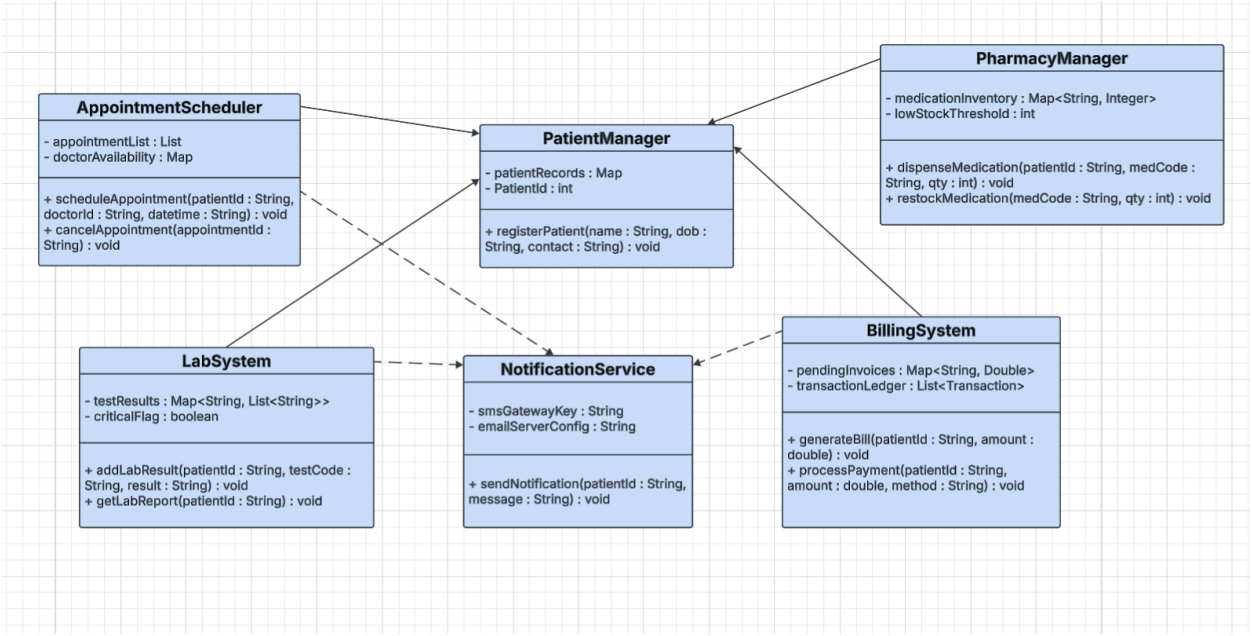
Appointments, increasing the risk of merge conflicts and regression bugs. The code is also difficult to read and maintain due to its size.

b) Improve cohesion: We split the class into dedicated classes based on functional domains: PatientManager, AppointmentScheduler, BillingSystem, LabSystem, PharmacyManager, and NotificationService. This ensures high cohesion as each class contains only the data and methods relevant to its specific purpose.

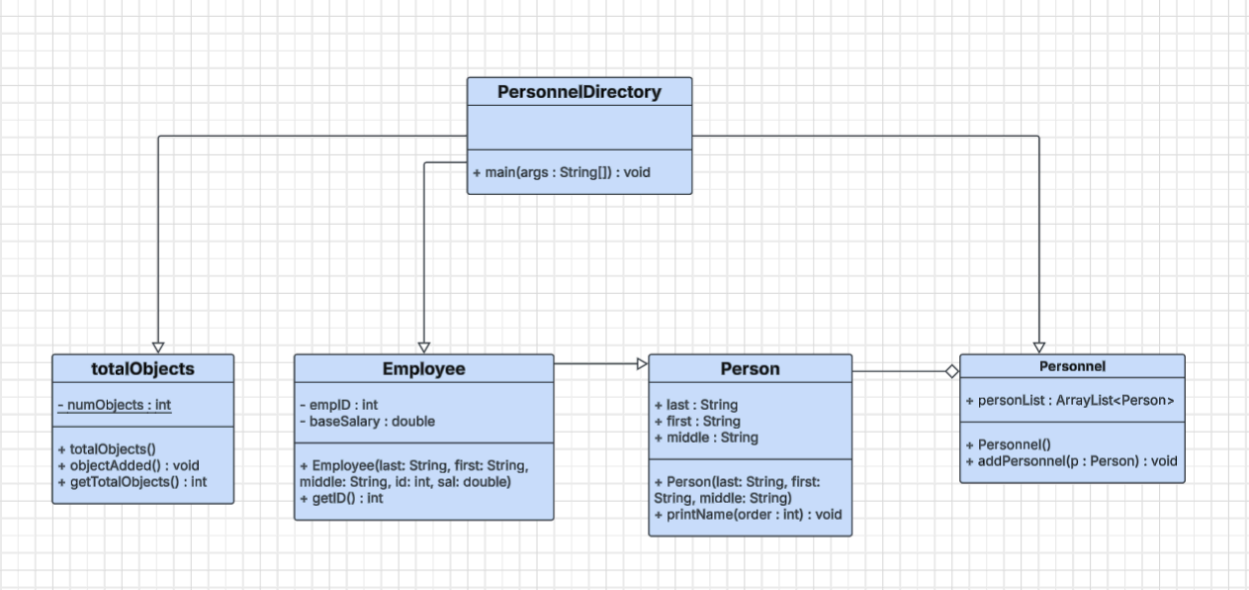
Improve testability: By decoupling the features into separate classes, we can write targeted unit tests for each class in isolation. For example, BillingSystem can be tested with mock payment data without needing to interact with or instantiate the LabSystem or PharmacyManager.

Improve code quality: The new codebase becomes modular, making it easier to understand, maintain, and extend. A developer can work on the PharmacyManager class without risking accidental breakage in the PatientManager logic, and the system becomes loosely coupled.

c)



Q3. a)



b)

Abstraction violation: UI/formatting logic is embedded in the domain model via Person.printName(int order), mixing I/O concerns with entity responsibilities.

Encapsulation violation: Person exposes name fields as public (public String first, middle, last), allowing uncontrolled external access and mutation.

Information hiding violation: Personnel exposes its internal collection as a public ArrayList personList, leaking representation and enabling external modules to traverse/modify the structure directly.

Content Coupling:

```
PersonnelDirectory:
for(int i =0; i <per.personList.size(); i++)
if( per.personList.get(i).first.equals(firstN) ...)
```

PersonnelDirectory accesses the personList field of Personnel directly and the first/last fields of Person directly. If Person changes its field names or Personnel changes from ArrayList to HashMap, PersonnelDirectory breaks.

Common Coupling:

```
totalObjects: private static int numObjects = 0;
```

totalObjects relies on a static variable. Static variables acts as global state. If another class instantiated totalObjects, it would reset or alter this shared global counter. This specifically happens in the constructor, numObjects=0 resets it globally every time an instance is created, which is a logic error caused by this coupling style.

Control Coupling:

```
Person: public void printName(int order)
PersonnelDirectory: per.personList.get(i).printName(order);
```

PersonnelDirectory requires callers to pass a control flag (order 0/1/2) to steer internal branching, forcing knowledge of internal control flow on the caller.

c)
Abstraction Fix: Remove I/O from domain objects by replacing printName(int order) with a pure method that returns a formatted name string or with toString(), and push printing/console concerns into PersonnelDirectory or a dedicated UI/service class.

Encapsulation Fix: Make Person fields private and provide necessary accessors or immutable construction, e.g., private final String first/middle/last with getters, preventing external mutation.

Information Hiding Fix: Make personList private and typed (private final List<Person>), and expose behavior-oriented methods like add(Person), size(), and findByName(first,last) instead of exposing the collection.

Content Coupling Fix: Make personList private in Personnel. Make name fields private in Person. Use methods like per.findPerson() or per.getAllPersonnel() so that the main program never directly accesses the specific list object or its fields.

Common Coupling Fix: Remove the totalObjects class and its static variable. Maintain the count of employees dynamically using the size() method of the ArrayList within the Personnel class. This removes the reliance on a global static state.

Control Coupling Fix: Remove the printName(int order) method. Instead, implement a toString() method in Person for a standard representation, or provide specific getters so the UI (PersonnelDirectory) can decide how to format the string itself. This decouples the data holder (Person) from the display logic.

Refactored Code Link:
https://drive.google.com/drive/folders/1kjFZxc6IJz40OllwEFHUR5EI_86tm1rS?usp=sharing