Assignment 1

Supriya Cirimoni

1. Describe the principle of information hiding as it applies to software design?

Ans: The information hiding principle is a powerful tool and fundamental concept for software engineering designers who want to improve the complexity, maintainability, and extensibility of modern software systems. This principle states that modules or components should conceal their internal details and only expose well-defined interfaces to other modules. According to this, a software module should reveal only those aspects of its internal implementation that are required for other modules to use it. In other words, a module should keep its implementation details hidden from other modules and only provide an interface describing how to use it. This enables modules to be developed and modified independently without affecting the system's overall behavior.

1. Describe the roles of the three sections of CRC (class responsibility collaborator) cards?

Ans: CRC cards are used in object-oriented software design. They are divided into three categories:

Class, Responsibility, Collaboration. The class section contains superclass and subclass information to refer to groups of similar objects or people. This is the name of the object or concept represented by the class. The Responsibilities section outlines the responsibilities of the class. These are functions that classes are responsible for. The Collaborate section lists other classes that collaborate with this class to accomplish that task. Collaborators can be external systems or libraries as well as other classes within the same system.

1. Consider the following class implementations in java. Which OO SOLID design principle is violated? Explain your answer. Also, how would you change the code to improve the design?

Ans: The OO SOLID design principle violated in this code is the Single Responsibility Principle (SRP).

The TransportationDevice class has multiple responsibilities such as storing the name, speed, and engine of a transportation device and starting the engine. This violates the SRP, which states that a class should change for only one reason. In this case, if any of the responsibilities change, the TransportationDevice class must be modified.

To improve the design, we can divide the responsibilities of the TransportationDevice class into smaller, more focused classes. For example, we can create a separate class to handle the engine and its functions, and another class to handle the name and speed of the transportation device.

Changes made in the code to improve design:

class TransportationDevice

{

String name;

String getName() { ... }

void setName(String n) { ... }

double speed;

double getSpeed() { ... }

void setSpeed(double d) { ... }

}

class Engine {

void start() { ... }

}

class EngineAbsentException extends Exception { ... }

class EngineStarter {

void startEngine(Engine engine) throws EngineAbsentException {

if (engine == null) {

throw new EngineAbsentException();

}

engine.start();

}

}

class Car extends TransportationDevice

{

Engine engine;

EngineStarter engineStarter = new EngineStarter();

void setEngine(Engine e) { engine = e; }

void startEngine() throws EngineAbsentException {

engineStarter.startEngine(engine);

}

}

class Bicycle extends TransportationDevice

{

void startEngine() throws EngineAbsentException {

throw new EngineAbsentException();

}

}

1. Consider the following code representing employees of some company. We have a Supervisor class which represent the person who supervises the employees. Next, we have two types of employees - Grade1 and Grade2. Both types of employees works and they need a daily lunch break to eat.

Ans: It could be argued that the Supervisor violates the Open/Closed principle because adding a new type of employee, would necessitate modifying the Supervisor class to handle the new type. To address this, we could change the Supervisor class to use a collection of IEmployee objects instead of a single employee object, and then use the supervise method to iterate through the collection, calling the work method on each employee. This allows us to add new employee types without having to change the Supervisor class.

Because robots do not require lunch breaks, we can create a new class Robot that implements the IEmployee interface but does not include the eat method.

Changes in the code:

public abstract class Employee {

public abstract void work();

}

abstract class EmployeeWithLunchBreak extends Employee {

public abstract void eat();

}

abstract class Robot extends Employee {

// no lunch break needed

}

class Grade1Employee extends EmployeeWithLunchBreak {

public void work() {

// ....working

}

public void eat() {

// ...... eating in lunch break

}

}

class Grade2Employee extends EmployeeWithLunchBreak {

public void work() {

//.... working more

}

public void eat() {

//.... eating in lunch break

}

}

class RobotWorker extends Robot {

public void work() {

//.... working without lunch break

}

}

class Supervisor {

Employee employee;

public void setEmployee(Employee e) {

employee = e;

}

public void supervise() {

employee.work();

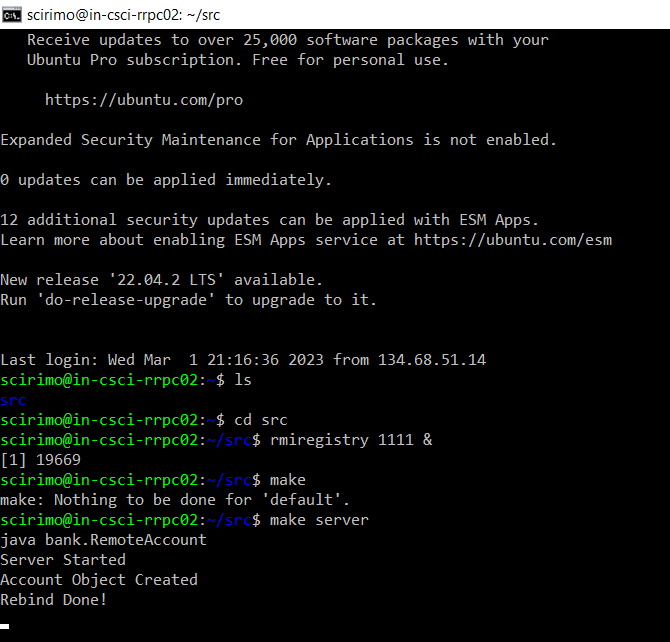
}

} Main {

}

1. Attached are the screenshots of the results.

Server side



Client side

