**CÂU 1:**

a.

Abstract class:

Class works as a blueprint for an object (real-world entity ) that defines all the attributes and functionality of that particular object. An abstract class is a class with the partial definition of an object that means all the methods or attributes are not defined yet ( or not properly ) and because of this partial implementation of an object there is no use in creating this object that is why the object creation of an abstract class is not possible.

abstract class A

{

void m1();

}

b.

// abstract class A

abstract class A

{

// this is an abstract class because the method in this class is not defined

// so object creation for this class is not possible

public void m1();

}

// class B is a concrete class ( concrete class provides the full implementation for an object )

class B extends A

{

// because class B extends A ( inheritance ) defines the method m1 in this class

public void m1()

{

// provide the functionality

// functionality

}

// another method m2 for class B

public void m2()

{

// functionality

}

}

// end of the class

**Câu 2:**

A screenshot of a computer program

Description automatically generated

public class Zombie {

private int health;

private int strength;

public int getHealth() {

return health;

}

public void setHealth(int health) {

this.health = health;

}

public int getStrength() {

return strength;

}

public void setStrength(int strength) {

this.strength = strength;

}

}

public class Combat {

public int computeDamage(Zombie zombie, Tree target) {

// Logic to compute damage based on zombie's strength and target's properties

return zombie.getStrength() - target.getDefense();

}

}

public class Graphic {

private GIF gif;

public void draw() {

// Logic to render the GIF on screen

}

}

**Câu 3:**

**D**ependency Inversion: A class should depend on abstractions rather than concretions, meaning that it should rely on interfaces or abstract classes rather than concrete classes.

a. Explanation of Violation of the Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules but rather on abstractions. In the provided code, the MySocialNetwork class directly depends on the My\_HCMIU\_Email class, which is a concrete implementation. This creates a tight coupling between the MySocialNetwork class and the My\_HCMIU\_Email class, making the code less flexible and harder to maintain or extend. If we wanted to change the login service or add a new one, we would have to modify the MySocialNetwork class.

The key issues are:

Direct Dependency: MySocialNetwork depends directly on My\_HCMIU\_Email, a specific implementation.

Lack of Abstraction: There is no abstraction (like an interface) that defines the login service behavior. Both classes depend on concrete implementations instead of abstractions.

b. Rewritten Code to Conform with the Principle

To conform with the Dependency Inversion Principle, we need to introduce an abstraction for the login service. This can be done using an interface that both the My\_HCMIU\_Email class and any future login service classes can implement. The MySocialNetwork class will then depend on this interface instead of a concrete class.

// Define an interface for the login service

public interface ILoginService {

boolean login(String username, String password);

}

// Implement the interface in My\_HCMIU\_Email class

public class My\_HCMIU\_Email implements ILoginService {

@Override

public boolean login(String username, String password) {

// Implementation of login with student ID

/\* ………..\*/

return true; // Assuming login is successful for demonstration purposes

}

}

// Modify MySocialNetwork class to use the ILoginService interface

public class MySocialNetwork {

private ILoginService loginService;

// Setter to inject the login service

public void setLoginService(ILoginService loginService) {

this.loginService = loginService;

}

// Login method that uses the injected login service

public boolean login(String username, String password) {

return loginService.login(username, password);

}

}

// Example usage

public class Main {

public static void main(String[] args) {

MySocialNetwork socialNetwork = new MySocialNetwork();

My\_HCMIU\_Email emailService = new My\_HCMIU\_Email();

// Injecting the login service

socialNetwork.setLoginService(emailService);

// Attempt to login

boolean success = socialNetwork.login("username", "password");

System.out.println("Login successful: " + success);

}

}

1. **SOLID**

An acronym for five principles of OOP that help developers design software that is easy to understand, maintain, and extend. Those principles are:

**S**ingle Responsibility: A class should have only one reason to change, meaning that it should have only one job or responsibility.

* Một class chỉ nên xử lí một công việc duy nhất, mình sẽ tách nhỏ những phần của project thành từng class, functions

**O**pen-Close: A class should be open for extension, but closed for modification, meaning that it should allow adding new features without changing the existing code.

* Khi thêm chương trình hoặc chức năng cho class thì ta không nên chỉnh sửa class đó mà ta nên tạo ra class khác và kế thừa class cũ.

**L**iskov Substitution: A subclass should be able to replace its superclass without affecting the functionality of the program, meaning that the subclass should follow the contract of the superclass.

**I**nterface Segregation: A class should not depend on methods that it does not use, meaning that it should have multiple specific interfaces rather than one general interface.

**D**ependency Inversion: A class should depend on abstractions rather than concretions, meaning that it should rely on interfaces or abstract classes rather than concrete classes.