ENSF 607 - Assignment 4

Jeremy Sugimoto

30232526

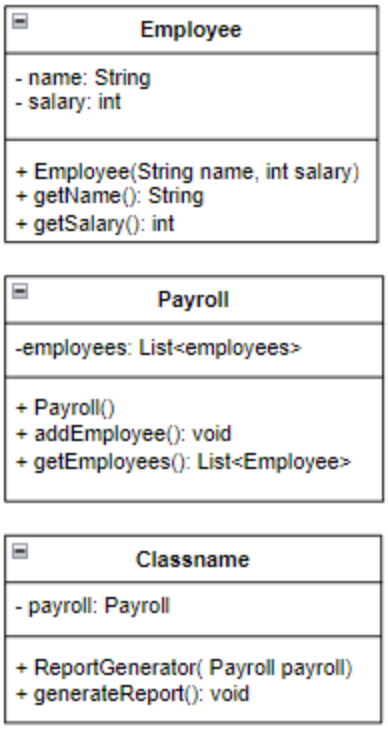
Exercise - 1: For each principle provide a class diagram (10 Marks)

Exercise - 2: For each principle provide the supporting code example (20 Marks)

Single-responsibility Principle (SRP)

A class should have only one reason to change.

Class Diagram:



Code Example:

```
public class Employee {
    private String name;
    private int salary;

    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }

    // Getters and setters

    public String getName() {
        return name;
    }

    public int getSalary() {
        return salary;
    }
}

public class Payroll {
    private List<Employee> employees;

    public Payroll() {
        employees = new ArrayList<>();
    }

    public void addEmployee(Employee employee) {
        employees.add(employee);
    }

    // Other payroll-related methods

    public List<Employee> getEmployees() {
        return employees;
    }
}

public class ReportGenerator {
    private Payroll payroll;

    public ReportGenerator(Payroll payroll) {
        this.payroll = payroll;
    }

    // Other methods

    public void generateReport() {
```

```
        // Code for generating the report
    }
}
```

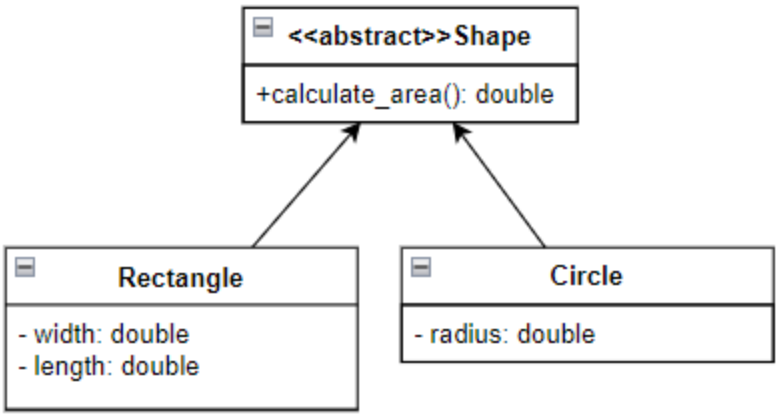
The code example above demonstrates the Single-responsibility Principle (SRP) in Java. The `Employee` class is responsible for storing employee information, the `Payroll` class is responsible for managing the employees' payroll, and the `ReportGenerator` class is responsible for generating reports based on the payroll data.

By separating the responsibilities into different classes, each class can be independently modified without affecting the others. This allows for better maintainability and flexibility in the system.

Open-Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Class Diagram:



Code Example:

```
public abstract class Shape {
    public abstract double calculateArea();
}

public class Rectangle extends Shape {
    private double width;
```

```
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double calculateArea() {
        return width * height;
    }
}

public class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double calculateArea() {
        return 3.14 * radius * radius;
    }
}
```

The `Shape` class is an abstract class with an abstract `calculateArea()` method. The `Rectangle` and `Circle` classes extend the `Shape` class and provide their own implementations of the `calculateArea()` method.

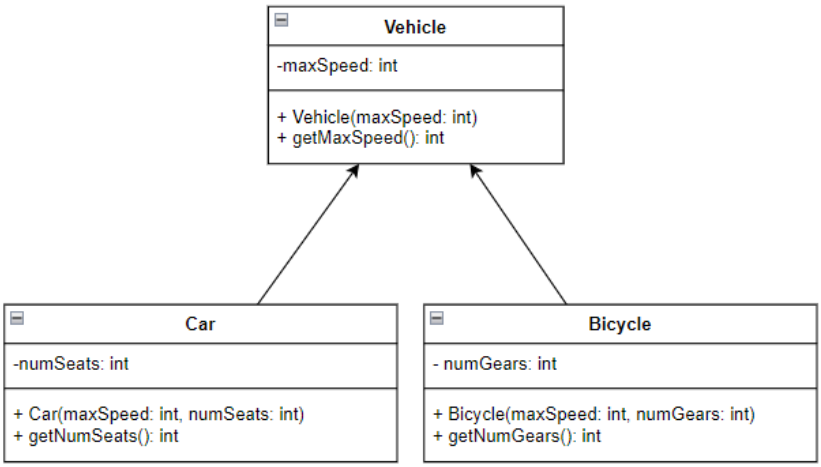
By adhering to the Open-Closed Principle, new shapes can be added to the system by creating new classes that extend the `Shape` class and implement the `calculateArea()` method. This allows for easy extensibility without modifying the existing code.

Please note that the code example is written in Java, as requested.

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

Class Diagram:



Code Example:

```
public class Vehicle {
    private int maxSpeed;

    public Vehicle(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }
}

public class Car extends Vehicle {
    private int numSeats;

    public Car(int maxSpeed, int numSeats) {
        super(maxSpeed);
        this.numSeats = numSeats;
    }

    public int getNumSeats() {
        return numSeats;
    }
}

public class Bicycle extends Vehicle {
    private int numGears;

    public Bicycle(int maxSpeed, int numGears) {
        super(maxSpeed);
        this.numGears = numGears;
    }
```

```
    public int getNumGears() {
        return numGears;
    }
}
```

The code example above demonstrates the Liskov Substitution Principle (LSP) in Java. The `Car` and `Bicycle` classes are substitutable for their base class `Vehicle`. They inherit the `maxSpeed` property from the `Vehicle` class and add their own specific properties.

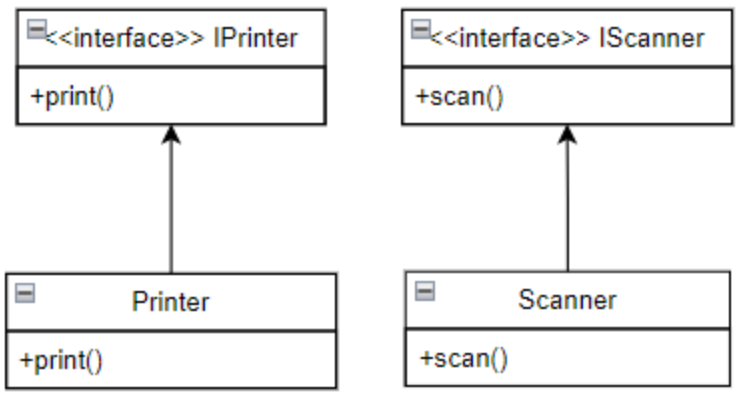
By adhering to the Liskov Substitution Principle, any code that expects a `Vehicle` object can also work correctly with a `Car` or `Bicycle` object. This principle ensures that subtypes can be used interchangeably with their base types without causing any unexpected behavior or breaking the system's functionality.

By applying the Liskov Substitution Principle, we can write more flexible and reusable code, and easily extend the system with new types of vehicles in the future.

Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use.

Class Diagram:



Code Example:

```
interface IPrinter {
    void print();
}
```

```
interface IScanner {
    void scan();
}

class Printer implements IPrinter {
    public void print() {
        // Code for printing
    }
}

class Scanner implements IScanner {
    public void scan() {
        // Code for scanning
    }
}
```

In this example, we have two interfaces: `IPrinter` and `IScanner`. The `Printer` class implements the `IPrinter` interface and provides the implementation for the `print()` method. The `Scanner` class implements the `IScanner` interface and provides the implementation for the `scan()` method.

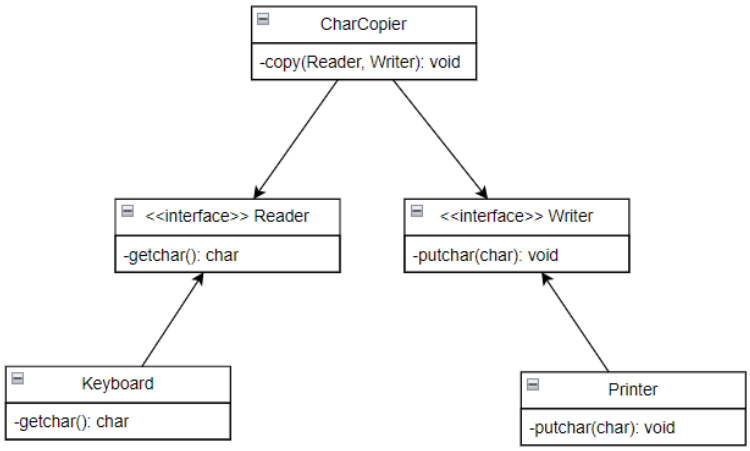
By separating the interfaces based on specific functionality, we ensure that clients only depend on the interfaces they actually use. This promotes code modularity, reduces coupling, and allows for easier maintenance and extension of the system.

Please note that the code example is written in Java, as requested.

Dependancy Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Class Diagram:




```
public interface Reader {
    char getchar();
}

public interface Writer {
    void putchar(char c);
}

class CharCopier {
    void copy(Reader reader, Writer writer) {
        int c;
        while ((c = reader.getchar()) != EOF) {
            writer.putchar(c);
        }
    }
}

public class Keyboard implements Reader {
    // Implementation of getchar()
}

public class Printer implements Writer {
    // Implementation of putchar(char c)
}
```

The code above demonstrates the Dependency Inversion Principle (DIP). The `CharCopier` class depends on abstractions (`Reader` and `Writer`) rather than concrete implementations (`Keyboard` and `Printer`). This allows for flexibility and decoupling between the `CharCopier` class and the specific implementations of `Reader` and `Writer`.

By depending on abstractions, the `CharCopier` class can be easily extended to work with different types of input sources (e.g., files, network streams) and output destinations (e.g., console, files) without modifying its implementation. This promotes code reusability, maintainability, and testability.

Exercise - 3: For each principle discussed provide an appropriate use case. Discuss, why you would apply this principle for the described use case compared to the other four. (30 Marks)

Single Responsibility Principle (SRP):

Use Case: Logging in an e-commerce application.

Reason for Applying SRP:

In this scenario, you might have a `Logger` class responsible solely for handling logging. Applying SRP to the logger class ensures that it focuses on one responsibility: logging. If you didn't apply SRP and mixed other responsibilities into the logger, it would be harder to maintain and could introduce errors when changes are made to other parts of the application.

Open/Closed Principle (OCP):

Use Case: Extending a payment processing system with new payment methods.

Reason for Applying OCP:

With OCP, you create a base `PaymentProcessor` class and derive specific payment method classes like `CreditCardPayment`, `PayPalPayment`, etc. The open/closed principle encourages extending the system without modifying existing code. You can add new payment methods without altering the core payment processing logic, making the code more maintainable.

Liskov Substitution Principle (LSP):

Use Case: Implementing various geometric shapes in a drawing application.

Reason for Applying LSP:

The LSP ensures that derived classes (e.g., `Rectangle`, `Circle`, `Triangle`) can be used interchangeably with the base class (`Shape`). In the drawing application, you can handle various shapes uniformly, making the code more flexible and understandable. Violating LSP would lead to unexpected behavior when switching between shapes.

Interface Segregation Principle (ISP):

Use Case: Defining interfaces for user roles in a content management system.

Reason for Applying ISP:

In the context of user roles, you may have interfaces like `AdminActions` and `UserActions`. Applying ISP, you ensure that classes only implement the methods relevant to their role. For instance, a regular user class shouldn't be forced to implement admin-specific methods, making the system more modular and less bloated.

Dependency Inversion Principle (DIP):

Use Case: Creating a messaging system where components can choose their messaging service.

Reason for Applying DIP:

DIP encourages abstraction and inversion of control. Components should depend on abstractions (e.g., `IMessagingService`) rather than concrete implementations (e.g., `EmailService` or `SMSService`). This allows components to choose their messaging service, promoting flexibility and testability.

Each SOLID principle has its specific use case, and the choice of which principle to apply depends on the context and the design goals. The principles can often be applied together to create a well-structured and maintainable software system. The choice to apply one principle over the others depends on the specific needs of the system and the desired benefits, such as maintainability, flexibility, or ease of extension.