Software Design Patterns

CAT 1: Design Principles and Analysis Patterns (evaluated on 50 points)

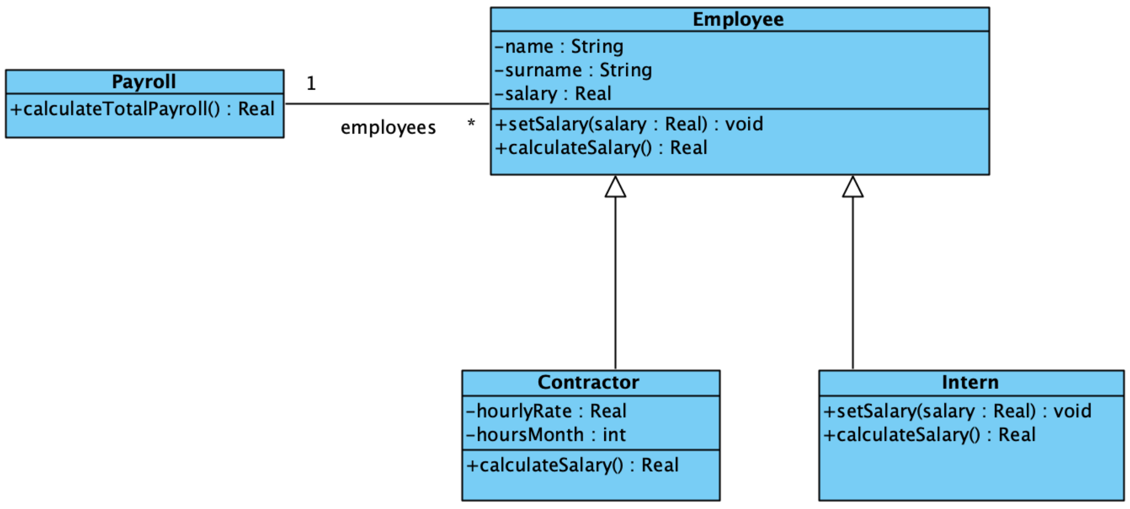
Question 1 (20 points)

Statement

We have developed a system for personnel management of a company. For each employee we will store their first name, last name and salary. In our company we have three types of employees: regular, interns and contractors. Regular workers receive their fixed salary at the end of the month, interns do not receive payment and contractor workers receive payment according to hours worked.

Our system also has a class that is responsible for calculating the total pay of all employees. To do this, it has a list of all employees and a *calculateTotalPayroll* function that goes through the list and adds up the salaries of all employees

Below is a possible model that represents the personnel management system described above:


Payroll
+calculateTotalPayroll() : Real
1
employees
*
Employee
-name : String
-surname : String
-salary : Real
+setSalary(salary : Real) : void
+calculateSalary() : Real
Contractor
-hourlyRate : Real
-hoursMonth : int
+calculateSalary() : Real
Intern
+setSalary(salary : Real) : void
+calculateSalary() : Real

NOTE: The constructors of the different classes have been omitted from the diagram.

In this model, *Employee* represents each of the company's workers. This class has two special employee subtypes *Intern* and *Contractor* that represent interns and freelancers respectively. Finally, *Payroll* manages the list of company workers and offers the calculateTotalPayroll operation that calculates the company's total personnel cost.

Integrity constraints:

- An employee is identified by first name and last name.

Pseudocode:

```
public class Employee {
    private String name;
    private String surname;
    private double salary;

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

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public double calculateSalary() {
        return salary;
    }
}

public class Intern extends Employee {
    public Intern(String name, String surname) {
        super(name, surname, 0); // Interns do not have regular salary
    }

    public void setSalary(double salary) {
        this.salary = 0; // Interns do not have regular salary
    }

    public double calculateSalary() {
        throw new UnsupportedOperationException("Interns do not have salary");
    }
}

public class Contractor extends Employee {
    private double hourlyRate;
    private int hoursMonth;

    public Contractor(String name, String surname, double hourlyRate, int
hoursMonth) {
        super(name, surname, 0); // Contractors do not have regular salary
        this.hourlyRate = hourlyRate;
        this.hoursMonth = hoursMonth;
    }

    public double calculateSalary() {
        return hourlyRate * hoursMonth;
    }
}

public class Payroll {
    private List<Employee> employees;

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

    public double calculateTotalPayroll() {
        double totalSalary = 0;
        if(employees!=null) {
            for (Employee employee : employees) {
                if(!employee instanceof Intern) {
                    totalSalary += employee.calculateSalary();
                }
            }
        }
        return totalSalary;
    }
}
```

It is requested:

a) (5 points) Indicate whether this design satisfies Open/Closed Principle (OCP) and justify your answer.
b) (5 points) Indicate whether this design satisfies Don't Repeat Yourself (DRY) Principle and justify your answer.
c) (5 points) Indicate whether this design satisfies High Cohesion Principle and justify your answer.
d) (5 points) Provide the software analysis and design to ensure compliance with the principles mentioned in the previous sections (if they were not already satisfied). The design must include the resulting class diagram and all modified or newly added pseudocode.

Solution

a) The Open/Closed (OCP) design principle states that a class should be open for extension but closed to modification. In this case, the design breaks this principle because to add a new employee type, we'll potentially have to modify the Payroll class to validate the condition in the for loop of the *calculateTotalPayroll* function, to see if the new employee type supports the *calculateSalary* operation. In fact, by adding the Intern class, we've already broken the OCP principle.

b) The No-Repetition (DRY) design principle states that each piece of knowledge should have a unique and unambiguous representation in the system. In this case, the design complies with this principle since there is no repeated logic in the pseudocode.

c) The High Cohesion design principle states that classes should have a single responsibility, and that a class's attributes and methods should be related to that responsibility. In this case, the design does not adhere to this principle because the *Intern* and *Contractor* classes have an unused salary attribute.

d) To adhere to these design principles, we're going to separate the responsibility of storing an employee's first and last name from the responsibility of storing their salary as a fixed value, since this responsibility isn't common to all employee types.

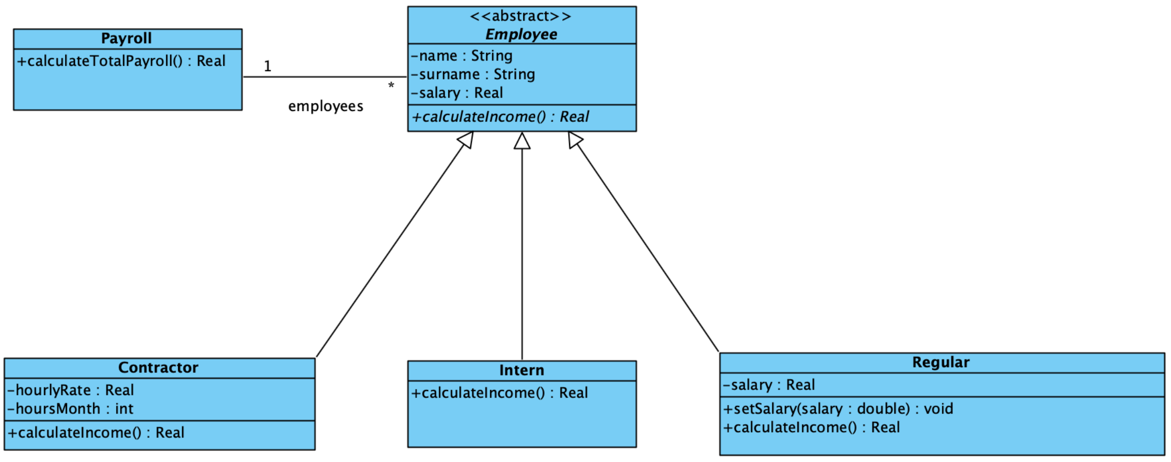
   To do this, we converted the *Employee* class into an abstract class with an abstract method for calculating an employee's income, which will be implemented by the child classes. We also removed the salary attribute from this class, since only permanent employees have a fixed salary and need to store this value. Others, such as self-employed workers, may receive other types of income. We changed the name of the method from *calculateSalary* to *calculateIncome* to emphasize this point.

   The new *Regular* class extends *Employee* and stores the fixed salary of a regular employee. The *Intern* and *Contractor* classes implement the calculateIncome method and store the values needed to calculate it. The *setSalary* method is removed from all child classes except for *Regular*, which is the only one that can modify the fixed salary.

Pseudocode:

```
public abstract class Employee {
    private String name;
    private String surname;

    protected Employee(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public abstract double calculateIncome();
}

public class Regular extends Employee {
    private double salary;

    public Regular(String name, String surname, double salary) {
        super(name, surname);
        this.salary = salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public double calculateIncome() {
        return salary;
    }
}

public class Intern extends Employee {
    public Intern(String name, String surname) {
        super(name, surname);
    }

    public double calculateIncome() {
        return 0;
    }
}

public class Contractor extends Employee {
    private double hourlyRate;
    private int hoursMonth;

    public Contractor(String name, String surname, double hourlyRate, int
hoursMonth) {
        super(name, surname);
        this.hourlyRate = hourlyRate;
        this.hoursMonth = hoursMonth;
    }
    public double calculateIncome() {
        return hourlyRate * hoursMonth;
    }
}

public class Payroll {
    private List<Employee> employees;

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

    public double calculateTotalPayroll() {
        double totalSalary = 0;
        if(employees!=null) {
            if(for (Employee employee : employees) {
                totalSalary += employee.calculateSalary();
            }
        }
        return totalSalary;
    }
}
```

Class diagram:


Payroll
+calculateTotalPayroll() : Real
1
*
employees
<<abstract>>
Employee
-name : String
-surname : String
-salary : Real
+calculateIncome() : Real
Contractor
-hourlyRate : Real
-hoursMonth : int
+calculateIncome() : Real
Intern
+calculateIncome() : Real
Regular
-salary : Real
+setSalary(salary : double) : void
+calculateIncome() : Real

Question 2 (10 points)

Statement

Design principles play a crucial role in software development, as adhering to them ensures the quality of the design. Among them is the Liskov Substitution Principle (LSP), which states that a subclass should be able to replace its base class without altering the behavior of the program. To understand this in depth, research its meaning and application using the course materials and references available in the classroom.

Once the concept is understood, answer the following questions:

a) (2.5 points) Does the Liskov Substitution Principle hold in the design of Exercise 1 before applying your changes? Justify your answer.
b) (2.5 points) Does the Liskov Substitution Principle hold in the design of Exercise 1 after applying your changes? Justify your answer.
c) (2.5 points) Provide a code example (before or after the changes in Exercise 1) that clearly shows why the Liskov Substitution Principle is violated, and explain why it does so.
d) (2.5 points) What are the consequences of violating this principle?

Solution

a) Before applying the changes, the **Liskov Substitution Principle** is not met in the design of exercise 1 since the Intern class cannot replace the *Employee* class without altering the program's behavior. The *Intern* class does not have a fixed salary, returning an exception, and therefore cannot replace the *Employee* class without altering its behavior. *Furthermore*, calling the *setSalary* method in the *Intern* or *Contractor* classes modifies the expected behavior of these classes. That is, we change the salary to a value, but immediately afterward, we can query it and it is not the value we assigned.

b) After applying the changes, the **Liskov Substitution Principle** is fulfilled in the design of exercise 1, since the *Intern* and *Contractor* classes can substitute the *Employee* class without altering the program's behavior. These two classes no longer implement the *setSalary* method, and the *calculateSalary* method behaves as expected in each case.

c) Below is a code example that clearly shows why the Liskov Substitution Principle is not met.Let's look at a couple of code examples before applying the changes to exercise 1 when the Liskov substitution principle was not fulfilled:

```
Employee employee = new Intern("Toni", "Oller");
employee.setSalary(1000);
double salary = employee.calculateSalary(); // throws
UnsupportedOperationException

Employee employee = new Contractor("Toni", "Oller", 2, 100);
employee.setSalary(1000);
double salary = employee.calculateSalary(); // 200
```

In the first case, we are creating an intern and assigning them a salary of €1,000. At this point, we would be violating the Liskov substitution principle, since we would expect that when querying for a salary that we could have correctly assigned, we would obtain that value as a result. However, when calculating the intern's salary, we obtain neither €1,000 nor €0. Instead, the method returns an unsupported operation exception, making it even more difficult to use the Intern class instead of its parent class, *Employee*.

In the second case, we create a contractor and assign them a salary of €1,000. In the next line, we query that object for their salary and obtain a result of €200. This breaks the expected behavior of the parent class, *Employee*, since if we assign a value to an attribute, we expect that value to be the one obtained if we subsequently query it in the object.

d) The consequences of not following the Liskov Substitution design principle are that subclasses cannot be used in place of the base class without altering the program's behavior. This can lead to runtime errors and a more complex and difficult-to-maintain design, as subclasses do not behave as expected.

Question 3 (10 points)

Statement

A company that manufactures medical equipment for hospitals and clinics wants to develop a system to record the specifications of the different types of medical devices it builds.

Each device has:

- An identifier name.
- A description that explains its function.
- A unit of measurement in which it operates.

For example, the device called “CARDIOMAX,” with the description “Digital heart rate monitor,” measures “Heart rate in beats per minute (bpm).”

Currently, the company manufactures devices that measure different physiological parameters such as heart rate, blood pressure (in mmHg), and oxygen saturation. For now, each device measures only one physiological parameter.

Each type of device, depending on its accuracy and technology, can operate within specific measurement values. These values must also be stored in our system.

For example, the “CARDIOMAX” device can measure heart rate from 40 bpm to 200 bpm.Se pide:

a) (2.5 puntos) Identificar qué patrones de análisis deben aplicarse para diseñar el sistema, justificando brevemente su utilidad.
b) (7.5 puntos) Proponer un diagrama estático de análisis, aplicando los patrones identificados y asegurando que se cumplan las restricciones de integridad del sistema.

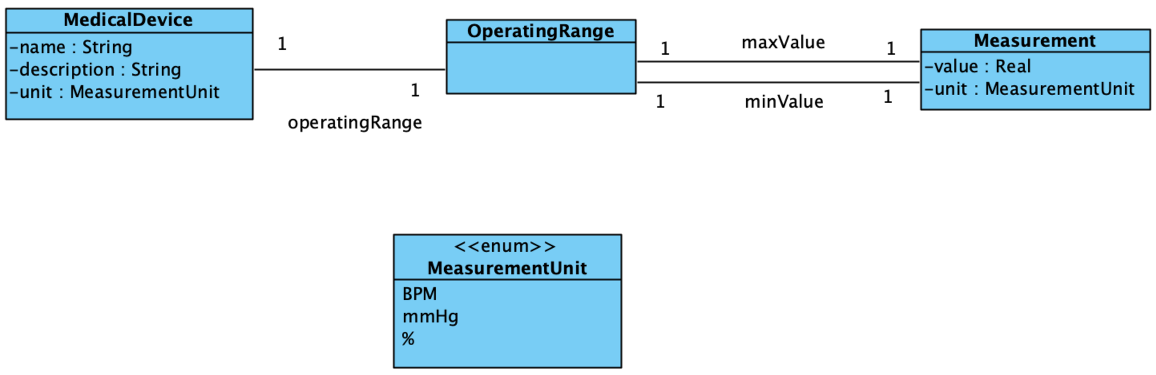
Note: If necessary, adjustments can be made to the patterns explained in the materials to adapt the solution to the specific requirements.

Solution

a) For this exercise, we will use the **Quantity pattern** to represent the different units of measurement in which medical devices operate. This pattern will allow us to represent not only the value but also the unit of measurement, as different devices will measure different physiological parameters and, therefore, different units. Even when measuring the same physiological parameter, different units of measurement may be used.

On the other hand, since we need to represent the minimum and maximum values within which a device can operate, we will use the **Range pattern**. This pattern will allow us to represent the minimum and maximum measurement values for each device.

b) Below is the static analysis diagram representing the requested system.



<u>Integrity constraints</u>

- The attribute name is the identifier of the MedicalDevice class.
- The attribute minValue must be less than maxValue.
- The attribute unit of a MedicalDevice must be equal to the unit attribute of its minimum and maximum values. In other words, the units of measurement must be the same for the MedicalDevice and its corresponding OperatingRange.

Question 4 (10 points)

Statement

Once the company has developed the system to record the specifications of the different types of medical devices, they now want to add the ability to record the measurements taken by the devices.

Each measurement taken by a medical device has:

- A unique identifier, e.g., the serial number, of the device instance that performed the measurement.
- A date and time when the measurement was taken.
- The recorded measurement.

For example, the device "CARDIOMAX" with serial number "1234" recorded a heart rate measurement of 75 bpm on January 1, 2025, at 10:30 AM.

It is requested:

a) (2.5 points) Identify which analysis patterns should be applied to design the system, briefly justifying their usefulness.
b) (7.5 points) Propose a static analysis diagram, applying the identified patterns and ensuring that the system's integrity constraints are met.

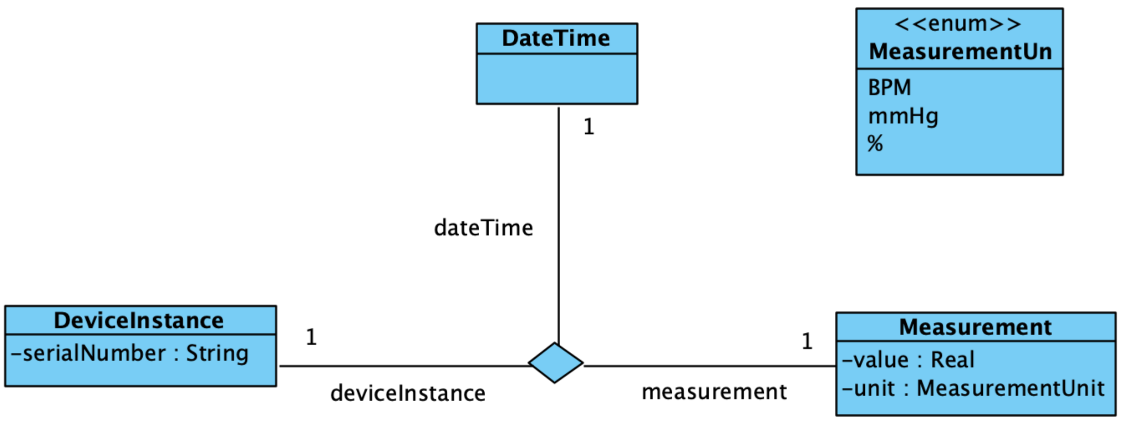
Note: If necessary, adjustments can be made to the patterns explained in the materials to adapt the solution to the specific requirements.

Solution

a) For this exercise, just like in the previous one, we will use the Quantity analysis pattern to represent the different units of measurement in which medical devices operate.

Additionally, we will use the Historical Association analysis pattern to represent the relationship between the measurements taken by medical devices and the date and time they were recorded.

b) Below is the static analysis diagram representing the requested system:



<u>Integrity constraints</u>

- *DeviceInstance* is identified by *serialNumber*.