**Software Design Patterns**

Continuous Assessment Activity 1 – CAT1

Nicolas D’Alessandro Calderon

**Design Principles and Analysis Patterns**

Question 1

1. The proposed design does not satisfy the OCP principle.

To satisfy the OCP principle the classes should be open for extension but closed for modification, and the problem with this approach is that if we were to add a new type of employee that also doesn’t have a salary, we may need to modify the class Payroll.

In the Payroll class, the method calculateTotalPayroll() there’s a type check: if(!employee instanceof Intern). This is intended to check that Intern employees, that do not have a salary, are not included in the payroll calculation. But for example, if we must add a “volunteer” employee type that is also not having a salary, we will need to add a new check in the Payroll class:

// Example of code modification needed if we add a new employee subtype

if(!(employee instanceof Intern || employee instanceof Volunteer)) {

    totalSalary += employee.calculateSalary();

}

In summary, the Payroll class shouldn’t be modified every time we introduce a new type of employee. A better OCP approach will be to have each employee type class defining its own calculateSalary()method calculation logic, avoiding adding type checks in the Payroll class.

1. The proposed design is not fully satisfying the DRY principle.

This principle says that each piece of logic should appear only once, and as we can observe in our code, the logic to calculate the payroll is being repeated in multiple classes.

Also, the Intern class sets the salary to 0 via super(name, surname, 0) and then this logic is duplicated in the setSalary method when explicitly setting this.salary = 0.

Additionally, the Contractor class has a separate calculation method but also follows a repetitive logic when setting the salary to 0 via super(name, surname, 0).

In summary, we have multiple places to force salary = 0 for an Intern and the Contractor salary calculation following a repetitive pattern. This means that some pieces of logic are repeated and if any business rule changes, we will need to update multiple parts of the code. So, since we don’t have a clean way of avoiding duplicating the checks and the salary calculation, we can say that the DRY principle is not being satisfied.

1. The proposed design does not fully satisfy the High Cohesion principle.

The high cohesion principle states that every class should have a unique and clear responsibility and in our code the Payroll class in managing multiple logics.

The reason is that the Payroll class has to know about the specific implementation of the different employee types, assigning multiple responsibilities to this class and reducing cohesion.

1. To address all the issues mentioned, I will propose this design:

// Base employee class

public abstract class Employee {

    private String name;

    private String surname;

    public Employee(String name, String surname) {

        this.name = name;

        this.surname = surname;

    }

    public abstract double calculateSalary();

}

// Regular employees have a fix salary

public class RegularEmployee extends Employee {

    private double salary;

    public RegularEmployee(String name, String surname, double salary) {

        super(name, surname);

        this.salary = salary;

    }

    public void updateSalary(double newSalary) {

        this.salary = newSalary;

    }

    @Override

    public double calculateSalary() {

        return salary;

    }

}

// Interns do not have salary

public class Intern extends Employee {

    public Intern(String name, String surname) {

        super(name, surname);

    }

    @Override

    public double calculateSalary() {

        return 0;

    }

}

// Contractors with salary calculation based on hours worked

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;

    }

    @Override

    public double calculateSalary() {

        return hourlyRate \* hoursMonth;

    }

}

// Payroll only manages employees

public class Payroll {

    private List<Employee> employees;

    public Payroll() {

        employees = new ArrayList<>();

    }

    public void addEmployee(Employee employee) {

        employees.add(employee);

    }

    public double calculateTotalPayroll() {

        double totalSalary = 0;

        for (Employee employee : employees) {

            totalSalary += employee.calculateSalary(); // Polymorphism

        }

        return totalSalary;

    }

}

* In the proposed design, the Payroll class has only one responsibility, managing employees. Also, each subtype of employee handles its own salary calculation without affecting the rest of the system. **High Cohesion**
* If we create a new employee subtype, we don’t need to modify the current existing classes. **OCP**
* We don’t repeat checks for intern and there’s no repeated logic for salary. **DRY**
* Additionally, we improved the method to handle salary change for regular employees and we added the override decorator to make sure that each employee subtype salary calculation method correctly overrides the method from the Employee parent class, making the code easier to read and understand.

UML

A diagram of a computer

AI-generated content may be incorrect.

Question 2

1. The initial design does not satisfy the LSP principle.

The LSP principle states that if we replace one class type with another, the system should still work properly, meaning that if a subclass changes the system’s behavior, then the LSP principle is not being satisfied.

In the original design, the Intern class breaks this rule by doing two different things:

1. Throwing an exception in calculateSalary() because any code expecting a number from this operation will make the system fail. This means the Intern class can be substituted for a regular Employee without changing how the system works, hence breaking the LSP principle:

public double calculateSalary() {

    throw new UnsupportedOperationException("Interns do not have salary");

  }

As we mentioned many times, the Payroll must manually exclude interns with the already explained check. So, this is a clear violation of the LSP principle since if Interns behave like any other Employee we wouldn’t need this check.

1. In our design, the Intern class is now returning 0 without throwing errors. So, now all the subclasses can be used the same way without breaking the code satisfying the LSP principle.

public double calculateSalary() {

        return 0;

    }

1. The clearest example of violation of the LSP principle is found in Intern class in the method calculateSalary() when throwing an exception. This prevents the Intern class from being used as a normal Employee, breaking the logic whenever the system assumes all employees can return a valid salary:

public double calculateSalary() {

    throw new UnsupportedOperationException("Interns do not have salary");

  }

So, if we replace a RegularEmployee with an Intern this will cause an error:

List<Employee> employees = new ArrayList<>();

employees.add(new RegularEmployee("Nico", "Dalessandro", 3000));

employees.add(new Intern("Juan", "Perez"));

// This will throw an exception breaking the LSP principle

for (Employee employee : employees) {

        double salary = employee.calculateSalary();

        System.out.println("Processing salary: " + salary);

}

For addressing this issue, we have modified the calculateSalary() method to return 0 instead of throwing an exception, making our Intern class available to be used interchangeably with other Employee types and making the Payroll available to process all employees without need of modification or extra checks:

public class Intern extends Employee {

@Override

public double calculateSalary() {

return 0;

}

}

1. Violating the LSP principle may have many consequences such as **unexpected errors or crashes at runtime** since the code expects some behavior from a base class but gets a different behavior from a subclass, as happened in our example from previous exercises.

Also, not following this principle may lead to **more manual checks and code complexity** as it happens in the instance type check for interns forced in the payroll class.

Finally, violating the LSP principle will result in **code that is hard to extend in the future**, as in our example, where we highlight that adding another employee type such as volunteers will require the modification of the payroll class.

Question 3

1. To design the proposed system, I will use the following patterns:

|  |  |
| --- | --- |
| **Pattern** | **Justification** |
| **Quantity Pattern** | The statement mentions that each physiological parameter is measured in a specific unit (bpm, mmHg, °C, etc.), so, **using the Quantity Pattern will help to ensure that the values are stored with their units, avoiding mistakes and calculation errors**. |
| **Range Pattern** | The statement mentions that each device has a valid range (blood pressure between 90-140 mmHg, etc.), so, using the Range **Pattern will help to ensure that the values remain within the acceptable limits avoiding incorrect measurements**. |

In summary:

* **For measuring physiological parameters,** we will use *Quantity pattern* for associate values with units
* **Because devices operate in a predefined range**, we will use the *Range pattern* to establish the limits for each device.

1. Static AnalysisDiagram

A diagram of a diagram

AI-generated content may be incorrect.

**System constraints achieved with this design:**

* A MedicalDevice is linked to one PhysiologicalParameter to ensure devices measure a specific parameter.
* A MedicalDevice has a defined Range setting the operational limits and avoiding incorrect readings.
* Each PhysiologicalParameter is measured in a specific unit to maintain the unit consistency.

Question 4

1. To design the proposed system, I will use the following patterns:

|  |  |
| --- | --- |
| **Pattern** | **Justification** |
| **Historical Association** | The statement mentions that the measurements are recorded over time, meaning that we will need to store the past values. So, **using the Historical Association pattern will help to ensure that we maintain the corresponding history of measurements with its timestamp and the serial number** and to be able to answer questions like what was the heart rate at 10:30 AM, etc. |
| **Quantity Pattern** | The statement mentions that each measurement has a specific unit (75 bpm, 120 mmHg, 36°C, etc.), so, **using the Quantity Pattern will help to ensure that the values are stored with their units, avoiding mistakes and calculation errors**. |

In summary:

* **For storing historical record of the measurements,** we will use the Historical Association pattern allowing us to keep track them.
* **For storing measurements with its corresponding unit,** we will use *Quantity pattern* for associate values with units

1. Static AnalysisDiagram

**A diagram of a diagram

Description automatically generatedA screenshot of a computer

Description automatically generated**

**System constraints achieved with this design:**

* A Measurement is connected to a specific MedicalDeviceInstance by the serial number to ensure traceability.
* A Measurement has a timestamp to avoid duplicates
* Each Measurement is stored with a Quantity, ensuring consistency in measurement units.