**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 system, the following analysis patterns should be applied:

|  |  |
| --- | --- |
| **Pattern** | **Why is it needed?** |
| **Quantity Pattern** | Each physiological parameter is measured in a specific unit (e.g., bpm, mmHg, °C). This pattern ensures values are stored alongside their units, preventing confusion or calculation errors. This pattern is essential for representing the physiological measurements with their units (e.g., "Heart rate in beats per minute (bpm)"). As described in section 3.3 of the Patterns Catalogue, this pattern allows us to make the unit of measurement explicit, support different units, and avoid coupling our design to specific measurement representations. |
| **Range Pattern** | Each device operates within a valid range (e.g., blood pressure should be between 90-140 mmHg). This pattern ensures that values remain within acceptable limits, preventing incorrect measurements. This pattern is perfect for representing the operating measurement ranges of each device (e.g., CARDIOMAX measuring from 40 bpm to 200 bpm). As section 3.4 of the Patterns Catalogue explains, this pattern provides a structured way to represent ranges of values and encapsulates the semantics of ranges, such as whether values fall within them. |

* Devices must measure physiological parameters → ✅ We need Quantity to associate values with units.
* Devices must operate within a defined range → ✅ We need Range to establish valid limits for each device.
* These patterns ensure integrity and correctness in data management. These patterns work together perfectly for this medical device system where measuring physiological parameters within specific ranges is the core functionality.

1. Now, applying the identified patterns, we construct the UML diagram:

**UML Diagram**

A screenshot of a computer

Description automatically generated

**To maintain system integrity, the design ensures:**

* Each Device is linked to exactly one PhysiologicalParameter (ensuring that devices measure a specific parameter).
* Each Device has a defined Range, enforcing operational limits to avoid incorrect readings.
* Each PhysiologicalParameter is measured in exactly one Quantity, ensuring unit consistency.
* Each Range is associated with a Quantity, preventing mismatched unit types in comparisons.

**Explanation of the Static Analysis Diagram:**

This diagram applies the identified analysis patterns to model the medical

device specification system:

1. Device Class:
   * Represents the medical device type with attributes for name, description, and manufacturer code
   * Each device measures one physiological parameter
   * Each device has one operating range
2. PhysiologicalParameter Class:
   * Represents parameters like heart rate, blood pressure, or oxygen saturation
   * Linked to its measurement quantity (unit of measurement)
3. Quantity Pattern (Section 3.3):
   * Implemented as the Quantity class that contains:
     + A numeric value
     + A unit of measurement as a string
   * Provides operations for unit conversion and string representation
   * This follows the pattern described in the Patterns Catalogue that separates the measurement value from its unit
4. Range Pattern (Section 3.4):
   * Implemented as the Range class with:
     + Minimum and maximum values
     + Operations to check if a value is within the range
     + Operations to check if ranges overlap
   * This directly implements the Range pattern from the Patterns Catalogue
   * Used to represent the operating ranges of medical devices (e.g., 40-200 bpm)

**Integrity Constraints:**

* Each device must have exactly one physiological parameter it measures
* Each device must have exactly one operating range
* The operating range must use the same unit type as the physiological parameter being measured
* Range minimum values must be less than maximum values
* This design satisfies the requirements by:
* Capturing all required device information (name, description, unit of measurement)
* Modeling the physiological parameters that devices measure
* Using the Range pattern to represent operating ranges
* Using the Quantity pattern to represent measurements with their units
* The design is extensible and will accommodate new device types, parameters, and measurement units as the company's product line grows.

A diagram of a computer

Description automatically generated

Question 4

1. To properly design the system that records measurements taken by medical devices, we apply the following **analysis patterns**:

|  |  |
| --- | --- |
| **Pattern** | **Why is it needed?** |
| **Historical Association** | Measurements are recorded over time, meaning we need to **store past values** for each device instance. This pattern ensures that we maintain a **history of all recorded measurements**, including timestamps and serial numbers. This pattern is perfectly suited for recording measurements over time. Since we need to track when measurements were taken, this pattern helps us maintain the temporal aspect of device measurements, enabling queries like "what was the heart rate at 10:30 AM?" or "show all measurements from January 1, 2025." |
| **Quantity Pattern** | Each measurement consists of a **value with a unit** (e.g., "75 bpm", "120 mmHg"). This pattern ensures that **each recorded measurement is stored correctly with its associated unit**, avoiding inconsistencies.  This pattern is essential for representing measurement values with their respective units. Each measurement recorded by a device will need to store both a numeric value and its unit of measurement (e.g., 75 bpm, 120/80 mmHg). |

**Justification:**

Measurements are time-sensitive, so we need a history of all readings → Use Historical Association to track them.

Measurements include numerical values with units → Use Quantity to ensure proper handling of units.

These patterns together provide a robust foundation for recording, querying, and analyzing measurements taken by medical devices throughout time.

**System Structure:**

* **MedicalDeviceInstance** → Represents a **specific device instance** with a unique serial number.
* **Measurement** → Represents a **recorded measurement**, linked to a device instance and timestamped.
* **Quantity** → Stores **the recorded value with its unit** (e.g., "75 bpm").

**UML Diagram in Mermaid**

**A screenshot of a computer

Description automatically generated**

**A diagram of a diagram

Description automatically generated**

**To maintain system integrity, the design ensures:**

* **Each Measurement is linked to a specific MedicalDeviceInstance** via its **serial number**, ensuring traceability.
* **Each Measurement has a timestamp**, preventing duplicate or missing records.
* **Each Measurement is stored with a Quantity**, ensuring consistency in measurement units.