Software Design Patterns

Continuous Assessment Activity 1 – CAT1

Nicolas D'Alessandro Calderon

Design Principles and Analysis Patterns

Question 1

a) 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.

b) 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.

c) 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.

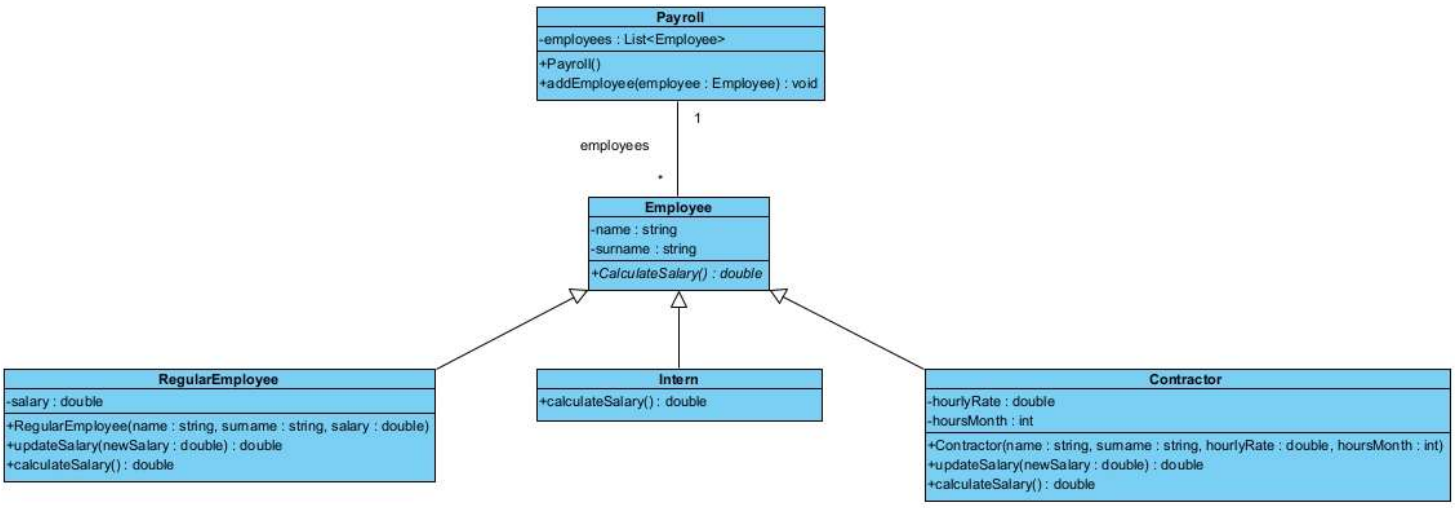
d) To address all the issues mentioned, I will propose this design:

```java
// 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;
    }
}
```

```java
 // 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



Question 2

a) 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.

b) 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;
    }
```

c) 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;
    }
}
```

d) 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

a) To design the proposed system, I will use the following patterns:

Pattern	Justification
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**.
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**.

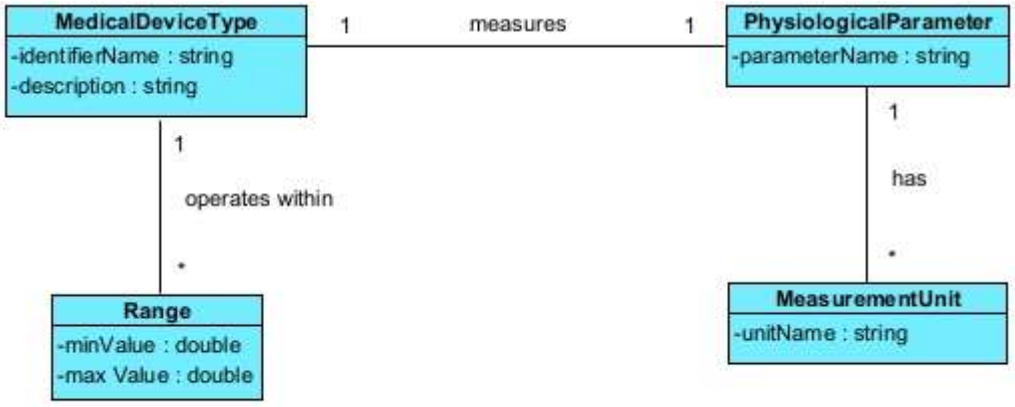
In summary:

- **Because devices operate in a predefined range**, we will use the *Range pattern* to establish the limits for each device.
- **For storing measurements with its corresponding unit,** we will use *Quantity pattern* for associate values with units.

Requirements:

1. `MedicalDeviceType` will represent the type of medical device like the "CARDIOMAX" and it will have the attributes `identifierName` (example "CARDIOMAX) and `description` (example "Device that measures hear rate").
2. `PhysiologicalParameter` will represent the physiological parameter that the device measure like "Heart Rate" and it will have the attribute `parameterName` (example "hear rate").
3. `MeasurementUnit` will represent the unit of measure the physiological parameter is being measured and will have the attribute `unitName` (example "bpm").
4. `Range` will represent the limits in which the device type can operate, and it will have the attribute `minValue` and `maxValue` (example 40-200)

b) Static Analysis Diagram



System constraints achieved with this design:

- A `MedicalDevice` is linked to one `PhysiologicalParameter` to ensure devices measure a specific parameter.
- Each `PhysiologicalParameter` is measured in a specific `MeasurementUnit` to maintain the unit consistency.
- A `MedicalDevice` has a defined `Range` setting the operational limits and avoiding incorrect readings.
- Each `PhysiologicalParameter` is measured in a specific `MeasurementUnit` to maintain the unit consistency.
- A `MeasurementUnit` can be the same in different `PhysiologicalParameter.`

Analysis Patterns applied in this design:

- **Quantity Pattern**: Represented when linking the physiological measurements with its unit by PhysiologicalParameter and MeasurementUnit
- **Range Pattern**: Represented by the Range class associated with each device and when defining the minValue and maxValue.

Example Implementation Code:

```
public class MeasurementUnit {
    private String unitName;

    public MeasurementUnit(String unitName) {
        this.unitName = unitName;
    }

    public String getUnitName() {
        return unitName;
    }

    public void setUnitName(String unitName) {
        this.unitName = unitName;
    }
}

public class PhysiologicalParameter {
    private String parameterName;
    private MeasurementUnit unit;

    public PhysiologicalParameter(String parameterName, MeasurementUnit unit) {
        this.parameterName = parameterName;
        this.unit = unit;
    }

    public String getParameterName() {
        return parameterName;
    }

    public MeasurementUnit getUnit() {
        return unit;
    }

    public void setUnit(MeasurementUnit unit) {
        this.unit = unit;
    }
}
```

```
public class Range {
    private double minValue;
    private double maxValue;

    public Range(double minValue, double maxValue) {
        if(minValue > maxValue) {
            throw new IllegalArgumentException("minValue must be <= maxValue");
        }
        this.minValue = minValue;
        this.maxValue = maxValue;
    }

    public double getMinValue() {
        return minValue;
    }

    public double getMaxValue() {
        return maxValue;
    }

    public boolean isWithinRange(double value) {
        return value >= minValue && value <= maxValue;
    }
}
public class MedicalDeviceType {
    private String identifierName;
    private String description;
    private PhysiologicalParameter parameter;
    private Range operatingRange;

    public MedicalDeviceType(String identifierName, String description,
PhysiologicalParameter parameter, Range operatingRange) {
        this.identifierName = identifierName;
        this.description = description;
        this.parameter = parameter;
        this.operatingRange = operatingRange;
    }

    public String getIdentifierName() {
        return identifierName;
    }

    public String getDescription() {
        return description;
    }
```

```
    public PhysiologicalParameter getParameter() {
        return parameter;
    }

    public Range getOperatingRange() {
        return operatingRange;
    }
}


public class Main {
    public static void main(String[] args) {

        // Measurement unit "bpm"
        MeasurementUnit bpmUnit = new MeasurementUnit("bpm");

        // Physiological parameter "Heart Rate" measured in "bpm"
        PhysiologicalParameter heartRate = new PhysiologicalParameter("Heart Rate",
bpmUnit);

        // Range: from 40 bpm to 200 bpm
        Range cardiomaxRange = new Range(40, 200);

        // Medical device type "CARDIOMAX"
        MedicalDeviceType cardiomax = new MedicalDeviceType(
                "CARDIOMAX",
                "Device that measures heart rate.",
                heartRate,
                cardiomaxRange
        );

        // Clear demonstration
        System.out.println("Device Type: " + cardiomax.getIdentifierName());
        System.out.println("Description: " + cardiomax.getDescription());
        System.out.println("Measures: " + cardiomax.getParameter().getParameterName());
        System.out.println("Unit: " + cardiomax.getParameter().getUnit().getUnitName());
        System.out.println("Operating Range: from " +
cardiomax.getOperatingRange().getMinValue() + " to " +
cardiomax.getOperatingRange().getMaxValue() + " " +
cardiomax.getParameter().getUnit().getUnitName());
    }
}
```

```
// Output of the code to demonstrate instantiation
Device Type: CARDIOMAX
Description: Device that measures heart rate.
Measures: Heart Rate
Unit: bpm
Operating Range: from 40.0 to 200.0 bpm
```

Question 4

a) 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 the heart rate at 10:30 AM was, 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

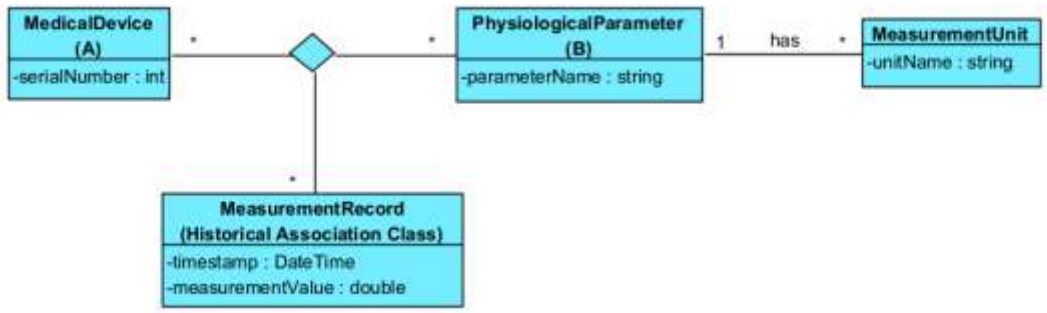
Requirements:

1. `MedicalDevice` will represent the physical device that is an instance of the MedicalDeviceType defined in exercise 3 and it will have a unique `serialNumber`. It can realize many measurements.
2. `PhysiologicalParameter` will represent the physiological parameter that the device measure like “Heart Rate” and it will have the attribute `parameterName` (example “hear rate”).

3. `MeasurementUnit` will represent the unit of measure the physiological parameter is being measured and will have the attribute `unitName` (example “bpm”).
4. `MeasurementRecord` will represent the historical association within the specific device type instance (example CARDIOMAX #1234) and the specific physiological parameter (example “Heart Rate”) at an exact time (example January 1, 2025, at 10:30 AM). It has the `Timestamp` attribute and the numerical value `measurementValue`).

By storing the MeasurementUnit and the values we will be applying the **Quantity** pattern, similar as we did in exercise 3. For applying the **Historical Association** pattern, we will use the learning material and the example solutions given as a reference, where we have *many examples of class A, class B and a class Date*. In our specific case the “Date” is not independent but an attribute within the historical class `MeasurementRecord.` This means we don’t need to represent the date as an independent class but as an attribute. So, we have a binary association within device and parameter with a historical class association that will store the date and the numerical value:

Medical Device (A) ↔ MeasurementRecord ↔ Physiological Parameter (B)

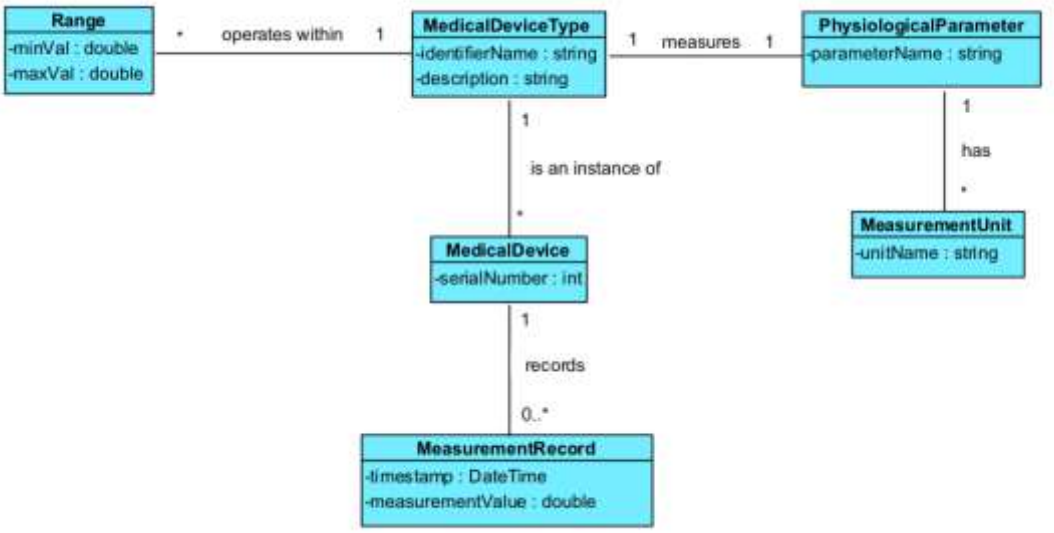


- **Class A**: *Medical Device* (CARDIOMAX #1234).
- **Class B**: *Physiological Parameter* (Heart Rate).
- **Historical Class**: *MeasurementRecord* that stores date time and measurement value (75 bpm, on January 1, 2025, at 10:30 AM)

System constraints achieved with this design:

- A `MeasurementRecord` is connected to a specific `MedicalDevice` instance by the serial number to ensure traceability.
- A `MeasurementRecord` has a timestamp to avoid duplicates
- Each `Measurement` is stored with a `PhysiologicalParameter`, that has it corresponding `MeasurementUnit` ensuring consistency in measurement units.

So, by adding this to the previous exercise class diagram we can obtain the following UML:



Example Implementation Code:

```
public class MeasurementUnit {
    private String unitName;

    public MeasurementUnit(String unitName) {
        this.unitName = unitName;
    }

    public String getUnitName() {
        return unitName;
    }
}

public class PhysiologicalParameter {
    private String parameterName;
    private MeasurementUnit unit;

    public PhysiologicalParameter(String parameterName, MeasurementUnit unit) {
        this.parameterName = parameterName;
        this.unit = unit;
    }

    public String getParameterName() {
        return parameterName;
    }

    public MeasurementUnit getUnit() {
        return unit;
    }
}

public class Range {
    private double minValue;
    private double maxValue;

    public Range(double minValue, double maxValue) {
        this.minValue = minValue;
        this.maxValue = maxValue;
    }

    public boolean isValid(double value) {
        return value >= minValue && value <= maxValue;
    }
}
```

```
public class MedicalDeviceType {
    private String identifierName;
    private String description;
    private PhysiologicalParameter parameter;
    private Range range;

    public MedicalDeviceType(String identifierName, String description,
PhysiologicalParameter parameter, Range range) {
        this.identifierName = identifierName;
        this.description = description;
        this.parameter = parameter;
        this.range = range;
    }

    public PhysiologicalParameter getParameter() {
        return parameter;
    }

    public Range getRange() {
        return range;
    }

    public String getIdentifierName() {
        return identifierName;
    }
}


public class MedicalDevice {
    private int serialNumber;
    private MedicalDeviceType deviceType;

    public MedicalDevice(int serialNumber, MedicalDeviceType deviceType) {
        this.serialNumber = serialNumber;
        this.deviceType = deviceType;
    }

    public int getSerialNumber() {
        return serialNumber;
    }

    public MedicalDeviceType getDeviceType() {
        return deviceType;
    }
}
```

```
import java.time.LocalDateTime;

public class MeasurementRecord {
    private LocalDateTime timestamp;
    private double measurementValue;
    private MedicalDevice device;

    public MeasurementRecord(LocalDateTime timestamp, double measurementValue,
MedicalDevice device) {
        this.timestamp = timestamp;
        this.measurementValue = measurementValue;
        this.device = device;
    }

    public String getSummary() {
        return "Device: " + device.getDeviceType().getIdentifierName() + " #" +
device.getSerialNumber() +
               "\nMeasured: " +
device.getDeviceType().getParameter().getParameterName() +
               "\nValue: " + measurementValue + " " +
device.getDeviceType().getParameter().getUnit().getUnitName() +
               "\nTimestamp: " + timestamp;
    }
}

import java.time.LocalDateTime;

public class Main {
    public static void main(String[] args) {

        // Unit (bpm)
        MeasurementUnit bpm = new MeasurementUnit("bpm");

        // Parameter (Heart Rate)
       PhysiologicalParameter heartRate = new PhysiologicalParameter("Heart Rate", bpm);

        // Valid Range (40 a 200 bpm)
        Range cardioRange = new Range(40, 200);

        // Device Type (CARDIOMAX)
        MedicalDeviceType cardiomaxType = new MedicalDeviceType(
            "CARDIOMAX",
            "Device measuring heart rate.",
            heartRate,
            cardioRange
        );
```

```
        // Specific device CARDIOMAX (#1234)
        MedicalDevice cardiomax1234 = new MedicalDevice(1234, cardiomaxType);

        // Historical Register (MeasurementRecord)
        MeasurementRecord record = new MeasurementRecord(
            LocalDateTime.of(2025, 1, 1, 10, 30),
            75.0,
            cardiomax1234
        );

        // Show Summary
        System.out.println(record.getSummary());
    }
}

// Output of the code to demonstrate instantiation
Device: CARDIOMAX #1234
Measured: Heart Rate
Value: 75.0 bpm
Timestamp: 2025-01-01T10:30
```