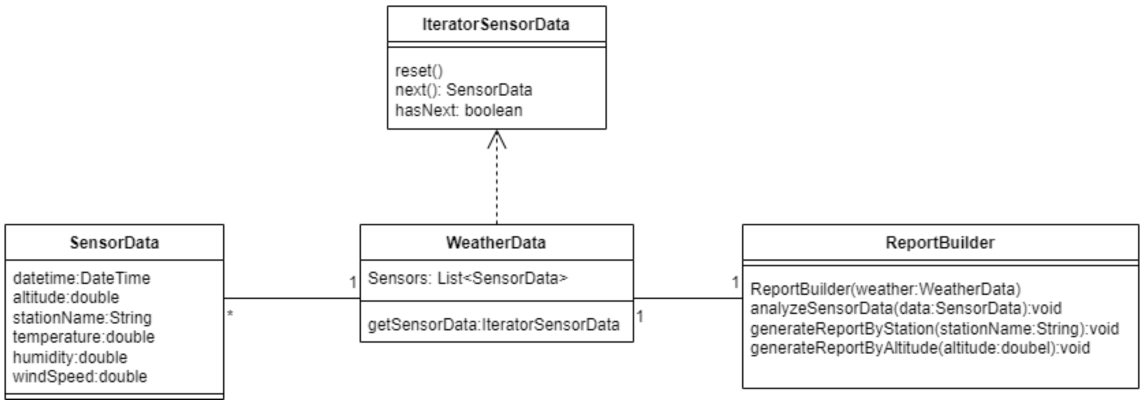
Name: Daniel Jesus del Rio Cerpa.

Exercise 1

a) I consider that the most suitable pattern for this exercise is the Iterator. This design pattern allows us to loop through a collection of elements without exposing their underlying representation. In this case, the Weather class is the one that would have the list of elements with the different Sensors, so we will create an independent object called iterator that allows us to go through the objects and extract information.

b)


IteratorSensorData
reset()
next(): SensorData
hasNext: boolean
SensorData
datetime:DateTime
altitude:double
stationName:String
temperature:double
humidity:double
windSpeed:double
WeatherData
Sensors: List<SensorData>
getSensorData:IteratorSensorData
ReportBuilder
ReportBuilder(weather:WeatherData)
analyzeSensorData(data:SensorData):void
generateReportByStation(stationName:String):void
generateReportByAltitude(altitude:doubel):void
1
*
1
1

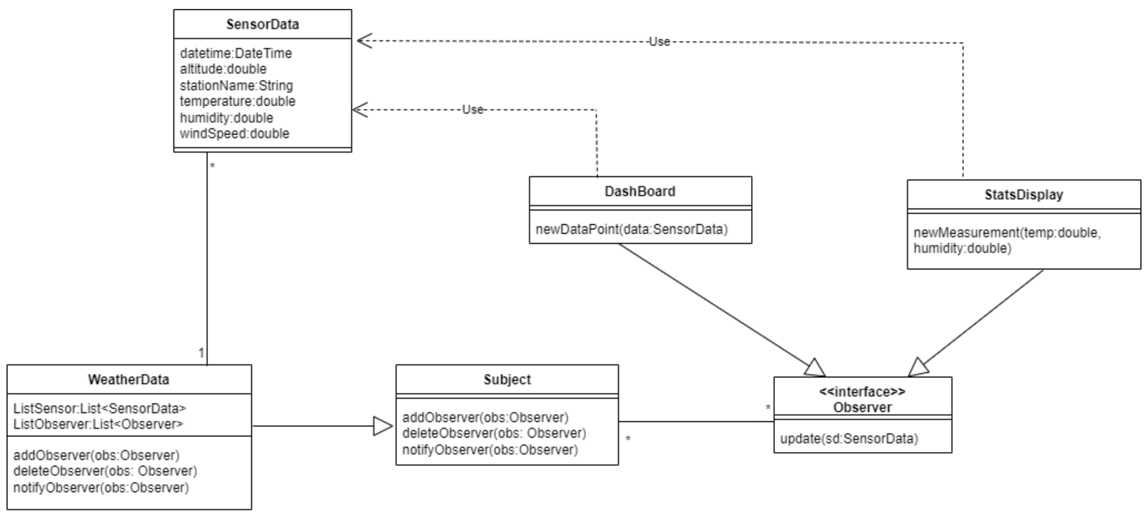
c)

```
class WeatherData{
        List<SensorData> sensorDataList;
        Iterator<SensorData> getSensorData() {
                return sensorDataList.iterator();
  }
}
class ReportBuilder{
        void analyzeSensorData(SensorData data){
        //Method for analyzing Data and generate a Report
        }
        //Method to traverse data by name
        void generateReportByStation(String stationName) {
            Iterator<SensorData> iterator = weather.getSensorData();
            while (iterator.hasNext()) {
              SensorData data = iterator.next();
              if (data.getStationName().equals(stationName)) {
                    analyzeSensorData(data);
                  }
               }
            }
        Method to traverse data by Altitude
        void generateReportByAltitude(double altitude) {
               Iterator<SensorData> iterator = weather.getSensorData();
               while (iterator.hasNext()) {
                  SensorData data = iterator.next();
                  if (data.getAltitude() > altitude) {
                     analyzeSensorData(data);
                  }
               }
            }
        }
}
```

Exercise 2

a) I consider that the most appropriate pattern for this case is Observer. This Pattern allows us to define a subscription mechanism to notify various objects about events that happen to the observed object. In this case it is very useful since we are going to have to enter and access different objects. So to implement it, we are going to use an observer interface that updates the object and a Subject class that will contain the add, delete and notify actions.

b)


SensorData
datetime:DateTime
altitude:double
stationName:String
temperature:double
humidity:double
windSpeed:double
Use
Use
DashBoard
newDataPoint(data:SensorData)
StatsDisplay
newMeasurement(temp:double, humidity:double)
*
1
WeatherData
ListSensor:List<SensorData>
ListObserver:List<Observer>
addObserver(obs:Observer)
deleteObserver(obs: Observer)
notifyObserver(obs:Observer)
Subject
addObserver(obs:Observer)
deleteObserver(obs: Observer)
notifyObserver(obs:Observer)
*
*
<<interface>>
Observer
update(sd:SensorData)

c)

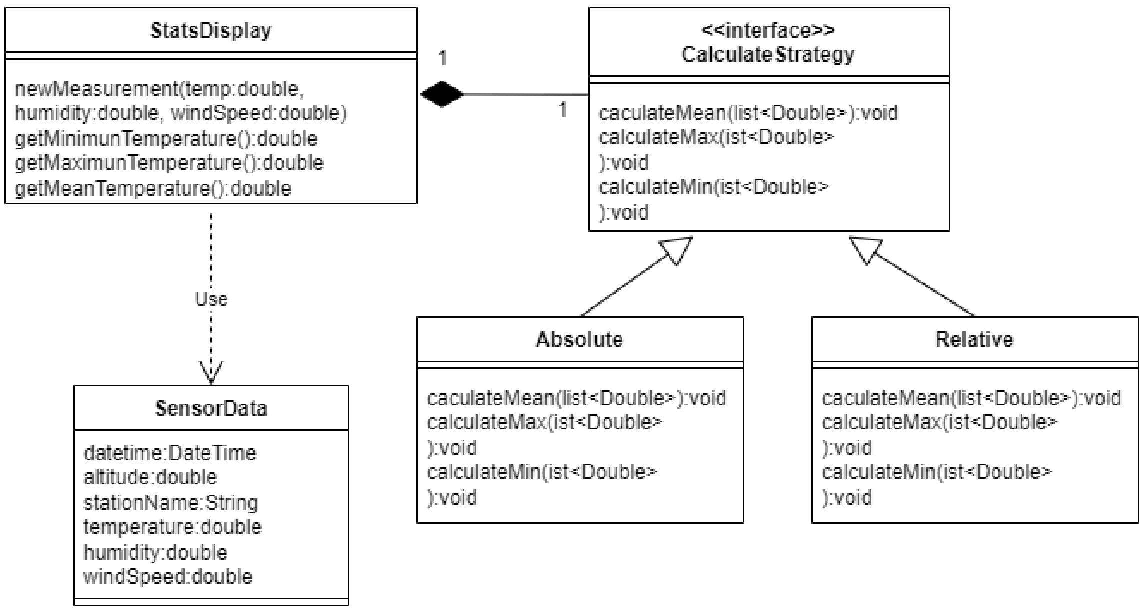
```
interface Observer {
   void update(SensorData sd);
}
Class Subject {
   void addObserver(Observer observer);
   void deleteObserver(Observer observer);
   void notifyObservers(SensorData sd);
}
class SensorData {
   DateTime datetime;
   double altitude;
   String stationName;
   double temperature;
   double humidity;
   double windSpeed;
}
class WeatherData implements Subject {
   List<SensorData> sensorDataList;
   List<Observer> observers;
           void addNewSensorData(SensorData sd) {
      sensorDataList.add(sd);
```

```
        notifyObservers(sd);
    }


    void addObserver(Observer observer) {
        observers.add(observer);
    }


    void deleteObserver(Observer observer) {
        observers.remove(observer);
    }
void notifyObservers(SensorData sd) {
        for (Observer observer : observers) {
            observer.update(sd);
        }
}
class Dashboard implements Observer {
            void update(SensorData sd){
            newDataPoint(sd);
}
}
class Stats Display implements Observer {
            void update(SensorData sd){
            newMeasurement(sd.getTemperature(), sd.getHumidity());
}
}
```

Exercise 3

a) I think the most appropriate pattern in this case would be the Strategy Pattern. This pattern allows us to separate by family of algorithms and separate them into specific details according to the type of implementation. In this case, we can see that we need different types of data depending on what scientists need.

b)


StatsDisplay
newMeasurement(temp:double, humidity:double, windSpeed:double)
getMinimunTemperature():double
getMaximunTemperature():double
getMeanTemperature():double
1
1
<<interface>> CalculateStrategy
caculateMean(list<Double>):void
calculateMax(ist<Double>):void
calculateMin(ist<Double>):void
Use
SensorData
datetime:DateTime
altitude:double
stationName:String
temperature:double
humidity:double
windSpeed:double
Absolute
caculateMean(list<Double>):void
calculateMax(ist<Double>):void
calculateMin(ist<Double>):void
Relative
caculateMean(list<Double>):void
calculateMax(ist<Double>):void
calculateMin(ist<Double>):void

c)

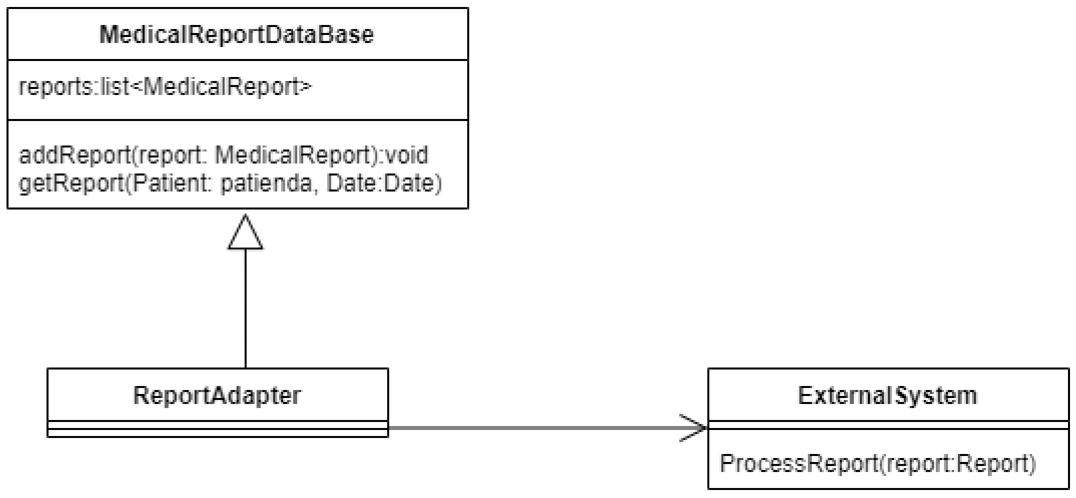
```
interface CalculateStrategy {
    double calculateMean(List<Double> temperatures);
     double calculateMax(List<Double> temperatures);
     double calculateMin(List<Double> temperatures);
  }
class Absolute implements CalculateStrategy {


   private double calculateMin(List<Double> temperatures) {
      if (temperatures.isEmpty()) {
         return 0.0;
      }


      double min = temperatures.get(0);
      for (double temperature : temperatures) {
         if (temperature < min) {
            min = temperature;
         }
      }


      return min;
    }

    private double calculateMax(List<Double> temperatures) {
        if (temperatures.isEmpty()) {
            return 0.0;
        }

        double max = temperatures.get(0);
        for (double temperature : temperatures) {
            if (temperature > max) {
                max = temperature;
            }
        }

        return max;
    }

    private double calculateMean(List<Double> temperatures) {
        if (temperatures.isEmpty()) {
            return 0.0;
        }

        double sum = 0.0;
        for (double temperature : temperatures) {
            sum += temperature;
        }

        return sum / temperatures.size();
    }
}
```

```
class Relative implements CalculateStrategy {
    private finalNumbers int n;


    Relative (int n) {
        this.n = n;
    }
    private double calculateMin(List<Double> temperatures) {
        return calculateRelativeStat(temperatures, Double.MAX_VALUE, (current, candidate) ->
current > candidate);
    }


    private double calculateMax(List<Double> temperatures) {
        return calculateRelativeStat(temperatures, Double.MIN_VALUE, (current, candidate) ->
current < candidate);
    }


    private double calculateRelativeMean(List<Double> temperatures) {
        if (temperatures.size() < n) {
            return calculateMean(temperatures);
        } else {
            List<Double> lastNValues = temperatures.subList(temperatures.size() - n,
temperatures.size());
            return calculateMean(lastNValues);
        }
    }


    private double calculateMean(List<Double> temperatures) {
        if (temperatures.isEmpty()) {
            return 0.0;
        }


        double sum = 0.0;
        for (double temperature : temperatures) {
            sum += temperature;
        }

        return sum / temperatures.size();
    }

    private double calculateRelativeStat(List<Double> temperatures, double initialValue,
BiPredicate<Double, Double> comparison) {
        if (temperatures.isEmpty()) {
            return 0.0;
        }

        double stat = initialValue;
        int size = Math.min(n, temperatures.size());
        for (int i = temperatures.size() - size; i < temperatures.size(); i++) {
            double temperature = temperatures.get(i);
            if (comparison.test(stat, temperature)) {
                stat = temperature;
            }
        }

        return stat;
    }
}
```

Exercise 4

a) I consider that the most appropriate pattern for this case is the Adapter. This is because the adapter pattern allows us to collaborate with objects that are inaccessible due to their interface. As can be seen in the statement, the interfaces are incompatible, since both generate Reports in different formats, so we need to create the adapter class so that they are compatible.

b)


MedicalReportDataBase
reports:list<MedicalReport>
addReport(report: MedicalReport):void
getReport(Patient: patienda, Date:Date)
ReportAdapter
ExternalSystem
ProcessReport(report:Report)

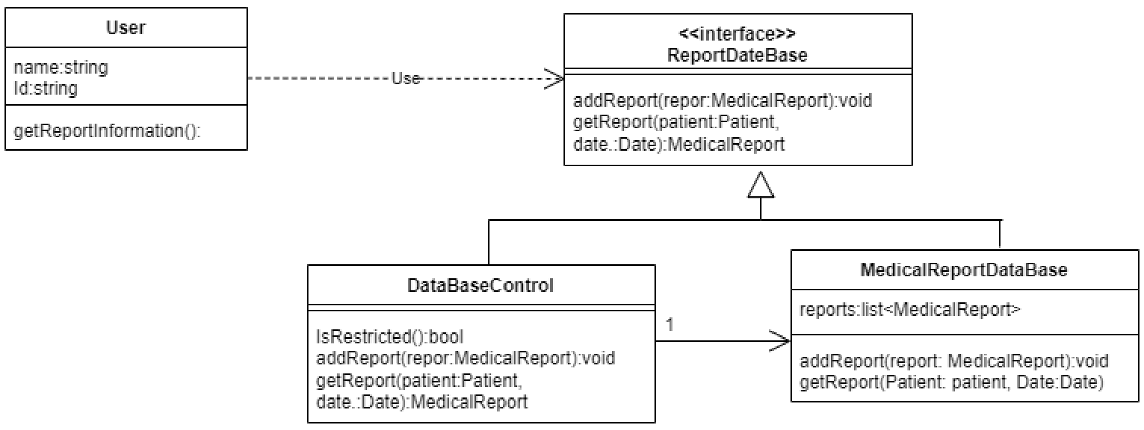
c)

```
public class ReportAdapter extends MedicalReportDataBase {
  private List<MedicalReport> reports;


  public ReportAdapter(List<MedicalReport> reports) {
    this.reports = reports;
  }


  public String getDate(int i) {
    return reports.get(i).getDate();
  }


  public String getDoctorNameAndSurname(int i) {
    return reports.get(i).getDoctor().getNameAndSurname();
  }


  public String getPatientNameAndSurname(int i) {
    return reports.get(i).getPatient().getNameAndSurname();
  }


  public String getReportText(int i) {
    return reports.get(i).getDiagnosis();
  }


  public void setDate(int i, String date) {
    reports.get(i).setDate(date);
  }


  public void setDoctorNameAndSurname(int i, String nameAndSurname) {
    reports.get(i).getDoctor().setNameAndSurname(nameAndSurname);
  }


  public void setPatientNameAndSurname(int i, String nameAndSurname) {
    reports.get(i).getPatient().setNameAndSurname(nameAndSurname);
  }


  public void setReportText(int i, String diagnosis) {
    reports.get(i).setDiagnosis(diagnosis);
  }
}
```

Exercise 5

a) I consider that the most appropriate pattern is the Proxy. It is a Design Pattern that allows us to control access to the original object, allowing us to do something before or after accessing it. I find it very useful in this case, since access is restricted to those users who have access to the database. So, we will create an interface from which the user will access, and then we will create a class to control accessibility to the database.

b)


User
name:string
Id:string
getReportInformation():
Use
<<interface>>
ReportDateBase
addReport(repor:MedicalReport):void
getReport(patient:Patient,
date.:Date):MedicalReport
DataBaseControl
IsRestricted():bool
addReport(repor:MedicalReport):void
getReport(patient:Patient,
date.:Date):MedicalReport
1
MedicalReportDataBase
reports:list<MedicalReport>
addReport(report: MedicalReport):void
getReport(Patient: patient, Date:Date)

c)

```
interface ReportDatabase {
  void addReport(MedicalReport report);
  MedicalReport getReport(Patient patient, Date date);
}
class MedicalReportDatabase implements ReportDatabase {
  private List<MedicalReport> reports = new ArrayList<>();
  public void addReport(MedicalReport report) {
    reports.add(report);
  }
  public MedicalReport getReport(Patient patient, Date date) {
    // Constructs a text string containing key information from the medical report
  }
}
class DataBaseControl implements ReportDatabase {
  private MedicalReportDatabase database;

  public DataBaseControl(MedicalReportDatabase database) {
    this.database = database;
  }

  public boolean isRestricted() {
    // Method that returns false if the user doesn't have permission and true if they do.
    }

    public void addReport(MedicalReport report) {
        if (isRestricted()) {
            System.out.println("No allowed to add medical report.");
        } else {
            database.addReport(report);
        }
    }

    public MedicalReport getReport(Patient patient, Date date) {
        if (isRestricted()) {
            System.out.println("No allowed to get medical report.");
            return null;
        } else {
            return database.getReport(patient, date);
        }
    }
}
```