UNIVERSITÀ DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Simulazione trasferimento dati di un centro meteorologico

Jacopo Sabatino
Giugno 2021

Contents

1 Introduzione

1.1 Dominio dell'elaborato

Un piccolo centro meteorologico monitora il meteo di una città attraverso più stazioni meteorologiche posizionate in punti strategici. I dati raccolti dalle stazioni vengono inviati al centro, il quale li rielabora per ottenere uno stato generale del meteo della città.
I parametri monitorati dal centro sono:

- **Pressione atmosferica**
- **Velocità del vento**
- **Direzione del vento**
- **Precipitazioni**
- **Temperatura**
- **Umidità**

Il centro meteorologico offre anche un servizio agli edifici comunali della città, consegnando un dispositivo che possa mostrare temperatura e umidità, interna ed esterna all'edificio in cui è posizionato.
I dispositivi prelevano temperatura e umidità esterna direttamente dal centro, mentre temperatura e umidità interna sono raccolte da un termoigrometo. Ogni dispositivo può avere un solo termoigrometro associato, il quale sarà posizionato all'interno dell'edificio. Il termoigrometro è fornito dal centro insieme al dispositivo.

1.2 Motivazione e contenuti

L'obiettivo di questo elaborato è riuscire a simulare la raccolta e la gestione simultanea di dati da parte di un centro meteorologico. Il centro dovrà essere anche capace di ridistribuire i dati raccolti a chi ne fa richiesta, cioè ai dispositivi messi a disposizione degli edifici comunali. La raccolta dati da parte delle stazioni è realizzata grazie ad un sensore per ognuno dei parametri di interesse:

- **Barometro:** per la pressione atmosferica
- **Anemometro:** per la velocità del vento
- **Banderuola:** per la direzione del vento
- **Pluviometro:** per le precipitazioni
- **Termometro:** per la temperatura
- **Igrometro:** per l'umidità

Ogni termoigrometro associato ad un dispositivo degli edifici comunali avrà invece solo un termometro ed un igrometro.
L'elaborato si pone quindi come obiettivo anche la simulazione dei dati che i singoli sensori potrebbero raccogliere in un vero ambiente.
Le stazione, i termoigrometri e il centro meteorologico sono dotati di un trasmettitore che gli permette di inviare i dati a chi è interessato.

1.3 Requisiti funzionali

Il sistema che si viene a creare deve dare possibilità di accesso a tre diversi tipi di attori:

- **Tecnico:** può sostituire i sensori di una stazione o un termoigrometro. Inoltre può sostituire i trasmettitori delle stazioni, dei termoigrometri e del centro meteorologico.
- **DipendenteCentro:** si occupa della gestione e dell'osservazione dati delle stazioni.
- **PersonaEdificioComunale:** dal dispositivo in dotazione, osserva temperatura e umidità raccolte dal centro meteorologico e dal termoigrometro associato al dispositivo.

Segue lo schema dei casi d'uso del sistema da parte dei tre attori:

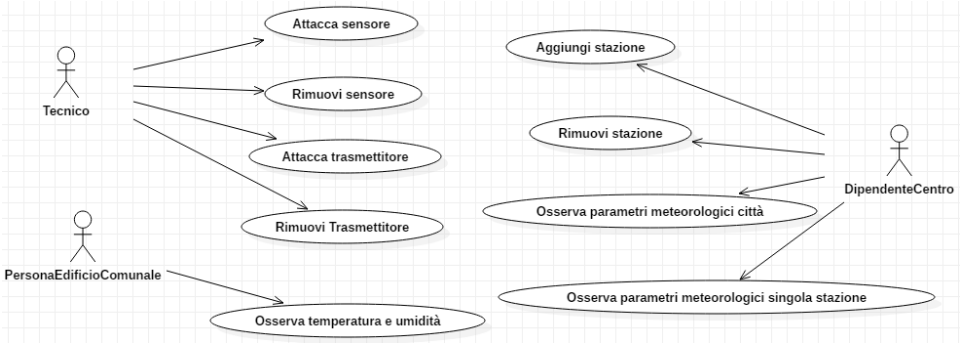


Figura 1: Diagramma dei casi d'uso

2 Progettazione

2.1 Diagramma delle classi UML

Di seguito è riportato nella sua interezza il diagramma delle classi dell'elaborato:

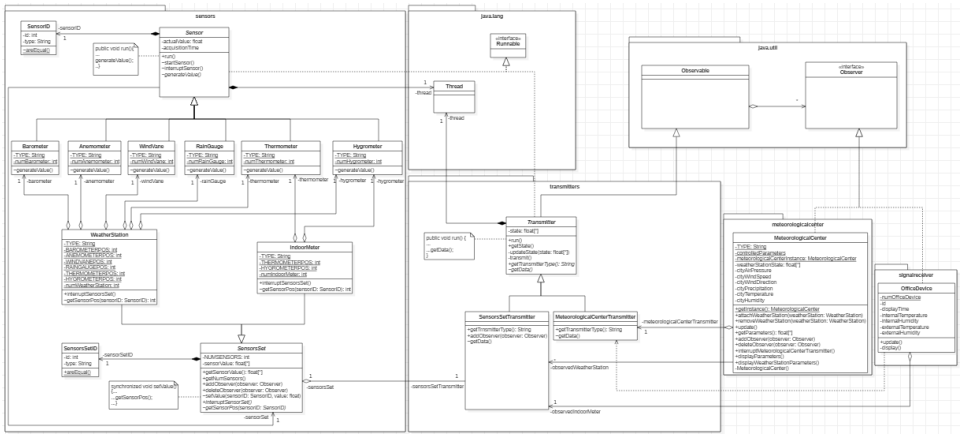


Figura 2: Diagramma delle classi

Nel seguente diagramma: le classi e i metodi scritti in corsivo sono astratti; gli attributi e i metodi sottolineati sono statici; gli attributi sottolineati e maiuscoli sono costanti della classe (static final in java).
Per una maggiore chiarezza riporto anche i dettagli di ogni package presente:

java.lang: package java utile in questo elaborato per l'interfaccia ***Runnable*** e la classe **Thread**.

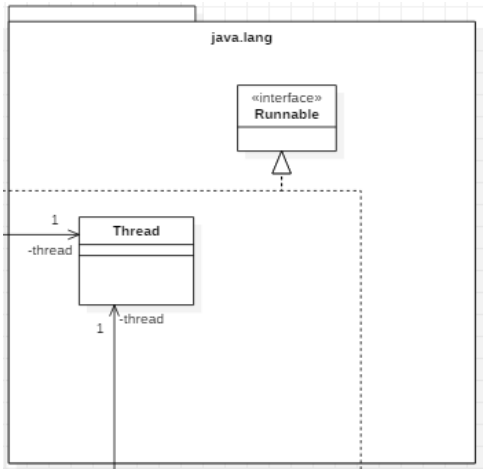


Figura 3: Diagramma delle classi del package java.lang

java.util: package java utile in questo elaborato per l'interfaccia ***Observer*** e la classe **Observable**.

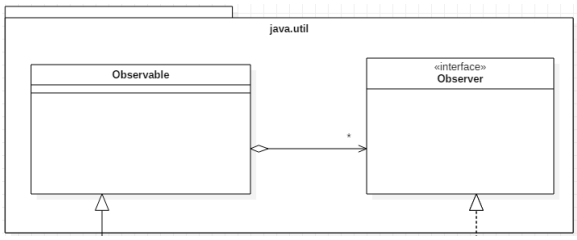


Figura 4: Diagramma delle classi del package java.util

signalreceiver:

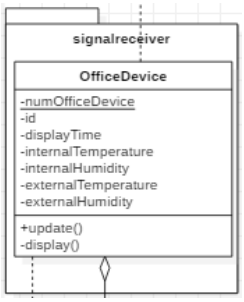


Figura 5: Diagramma delle classi del package signalreceiver

meteorologicalcenter:

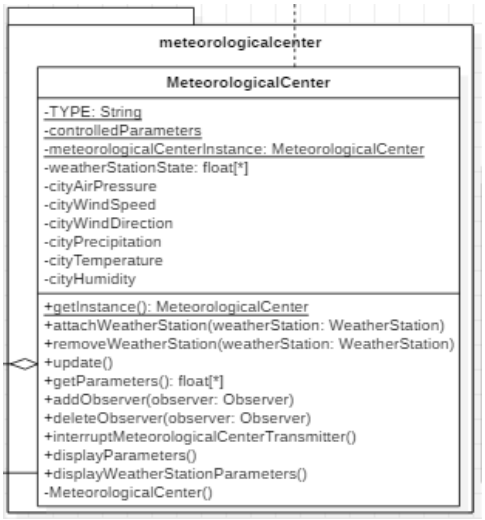


Figura 6: Diagramma delle classi del package meteorologicalcenter

sensors:

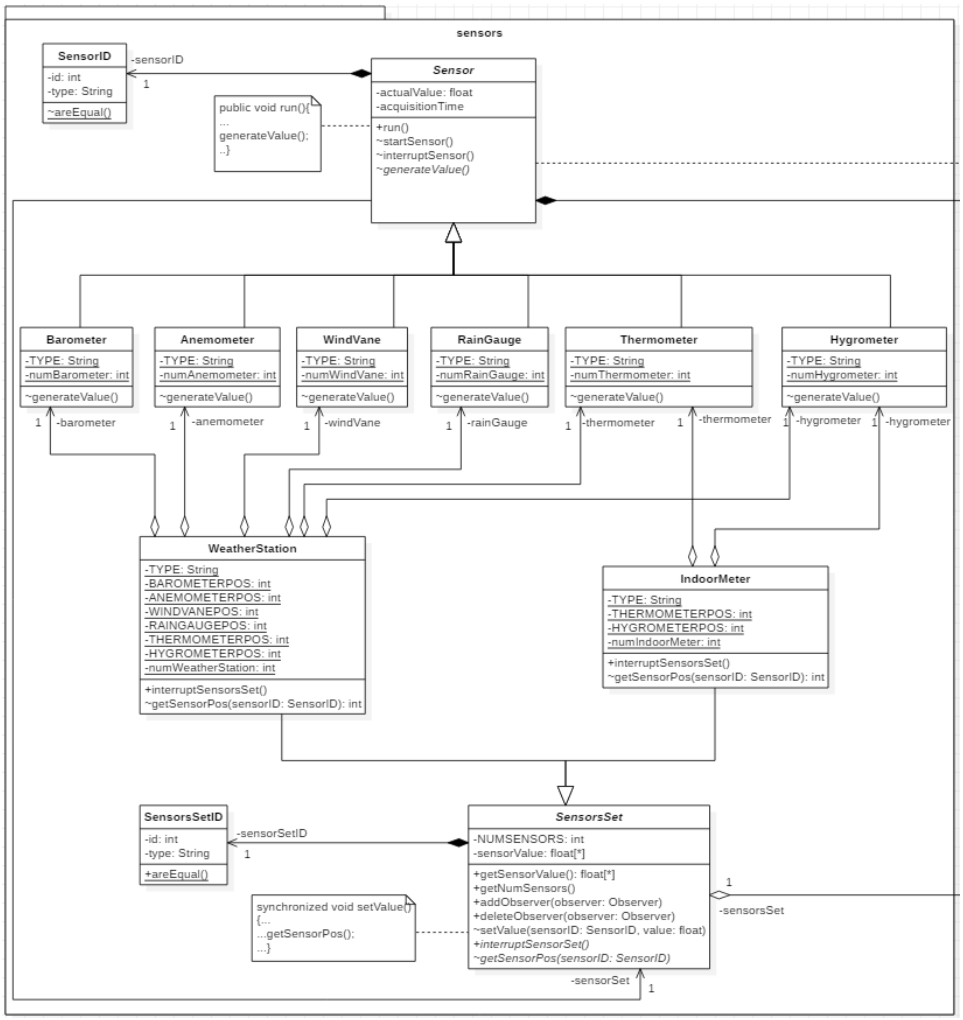


Figura 7: Diagramma delle classi del package sensors

Le note collegate a ***Sensor*** e ***SensorsSet*** servono a far capire che quelle classi hanno un metodo con il design pattern *template*. All'interno della nota si mostra il metodo *template* e il metodo astratto che lo rende tale.

transmitters:

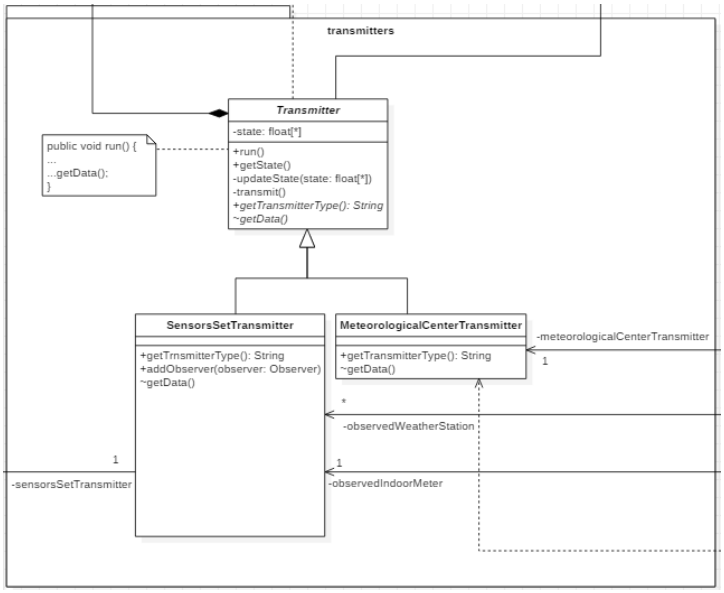


Figura 8: Diagramma delle classi del package transmitters

Anche in questo caso, la nota collegata alla classe ***Transmitter*** serve a far capire che al suo interno c'è un metodo con il design pattern *template*.

2.2 Mockups

Come visto nei casi d'uso, gli attori che si interfacciano al sistema sono 3: il **Tecnico**, il **DipendeteCentro** e la **PersonaEdificioComunale**.
Osservando i lori casi d'uso è stato ritenuto opportuno realizzare solamente il mockup dell'interfaccia fra **DipendeteCentro** e il sistema. Questa decisione è dovuta all'assenza di una vera e propria interfaccia fra gli attori esclusi ed il sistema.
Di seguito il mockup dell'interfaccia per **DipendeteCentro**:

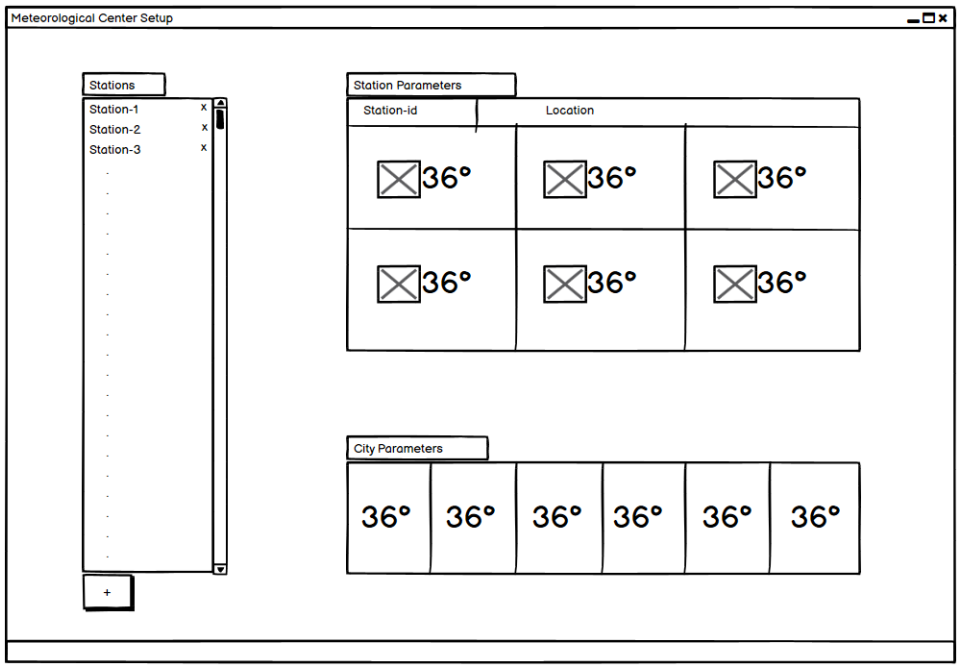


Figura 9: Mockup interfaccia **DipendeteCentro**

Sulla sinistra è presente una lista delle stazioni attualmente osservate. Ogni stazione è dotata di un tasto ”x” che permette la rimozione dalla lista. La rimozione di una stazione deve essere confermata dall'utente attraverso un *alert box*.
Il tasto ”+” permette di aggiungere una stazione non osservata alla lista di quelle osservate. Il passaggio appena citato viene realizzato in un'altra finestra che permette di visualizzare le stazioni non osservate e aggiungerle alla pagina principale attraverso un tasto ”Add”. Una volta aggiunta la stazione la finestra si chiude automaticamente.
In alto a destra è presente uno spazio dedicato a mostrare i parametri di una singola stazione. In questo spazio è visibile: il codice identificativo della stazione, la sua posizione in città e i dati che sta raccogliendo. Nella griglia dei dati, ogni parametro è dotato di un'immagine che lo rappresenta, del suo valore attuale e dell'unità di misura. Per cambiare la stazione osservata basta selezionare quella che interessa dalla lista delle stazioni osservate.
In basso a destra è mostrata la media dei dati meteorologici raccolti dalle stazioni cosicché si possa avere un'idea generale del meteo in città. Tutti i dati mostrati nell'interfaccia sono in tempo reale, cioè cambiano ad ogni aggiornamento di una stazione.

Di seguito l'interfaccia utente per l'aggiunta di una stazione e l'*alert box* di conferma per la rimozione:

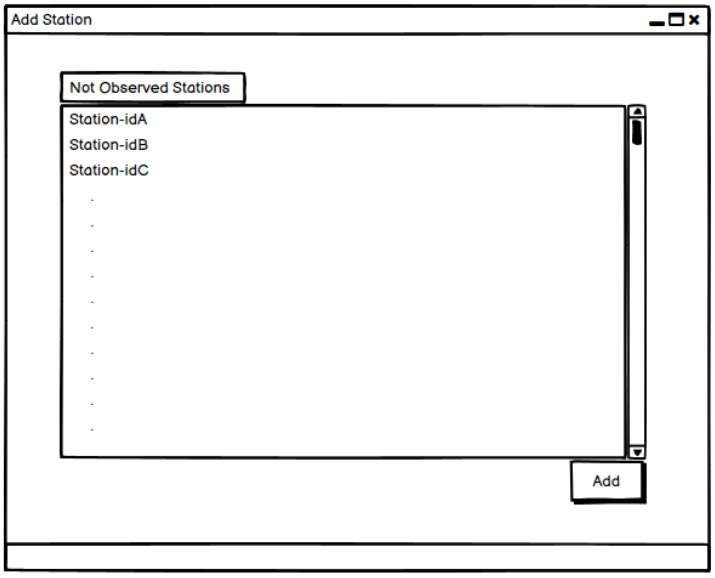


Figura 10: Mockup interfaccia per aggiungere una stazione

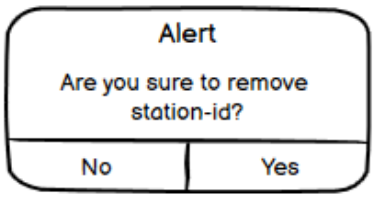


Figura 11: Mockup *alert box* per rimozione stazione

3 Implementazione

3.1 Package sensors

3.1.1 Sensor

Sensor è una classe astratta da cui si può derivare qualsiasi tipo di sensore reale.
Per simulare il funzionamento di un sensore, la classe ***Sensor*** è stata dotata di: un attributo **acquisitionTime** che indica l'intervallo di attesa per la generazione di un valore; un metodo astratto ***generateValue()*** che si occupa di generare il nuovo valore. L'inizializzazione dell'attributo **acquisitionTime** e la definizione del metodo ***generateValue()*** sono lasciate alla classe derivata.
La costante generazione di valori è resa possibile da un oggetto di tipo **Thread** assegnato a ***Sensor*** per composizione. L'oggetto **Thread** è istanziato alla costruzione del sensore e il parametro di tipo ***Runnable*** in ingresso è il sensore stesso. Quindi ***Sensor*** implementa ***Runnable*** e definisce il metodo **run()**.
Il metodo **run()** è stato realizzato usando il design pattern *template*. Un metodo realizzato con il design pattern *template* ha al suo interno almeno una chiamata ad un metodo astratto della classe. Nel caso di **run()** il metodo astratto è ***generateValue()***. Generato il valore, **run()** lo invia all'array della classe ***SensorsSet*** a cui ***Sensor*** è associato.
La classe ***Sensor*** ha per composizione un oggetto di tipo **SensorID**, utilizzato da identificativo per il sensore stesso.

```
abstract class Sensor implements Runnable{
    private SensorID sensorID;
    private SensorsSet sensorsSet;
    private Thread thread;
    private float actualValue;
    private int acquisitionTime;

    public void run() {
        try{
            while(true){
                if (sensorsSet != null)
                    sensorsSet.setValue(getSensorID(), actualValue);
                Thread.sleep(acquisitionTime);
                generateValue();
            }
        }catch(InterruptedException ignored) {}
    }


    Sensor(int acquisitionTime, int id, String type){
        this.sensorID = new SensorID(id, type);
        this.sensorsSet = null;
        this.acquisitionTime = acquisitionTime;
        thread = new Thread( target: this);
    }
```

Figura 12: Snippet classe ***Sensor***

3.1.2 SensorID

SensorID è una classe le cui istanze servono da identificativi per i sensori. I suoi attributi sono:

- **id:** intero che rappresenta il codice identificativo di quel sensore
- **type:** stringa che rappresenta il tipo di sensore

Il metodo statico **areEqual()** controlla se due oggetti **SensorID** corrispondono attraverso un confronto fra attributi.

```
class SensorID {
    private int id;
    private String type;

    SensorID(int id, String type) {
        this.id = id;
        this.type = type;
    }

    int getId() { return this.id; }

    String getType() { return this.type; }

    static boolean areEqual(SensorID a, SensorID b) {
        return ((a.getId() == b.getId()) && (a.getType().equals(b.getType())));
    }
}
```

Figura 13: Codice classe **SensorID**

3.1.3 Barometer

Barometer è una classe derivata da ***Sensor***.
Tutte le classi derivate da ***Sensor*** hanno: un attributo costante di tipo stringa che rappresenta il tipo di sensore; un attributo statico di tipo intero che conta il numero di sensori istanziati di quel tipo. Questi due attributi servono come parametri al costruttore del **SensorID**.
Ogni classe derivata da ***Sensor*** deve definire il metodo **generateValue()**.
Nella simulazione di questo elaborato, ogni classe derivata da ***Sensor*** ha un suo **acquisitionTime** che sarà fisso per ogni oggetto da lei istanziato. Quindi tutti i sensori dello stesso tipo avranno ugual tempo di attesa per la generazione di un nuovo valore. In realtà, un sensore di un certo tipo acquisisce dati secondo un intervallo di tempo o all'occorrenza di specifici eventi.

```java
public class Barometer extends Sensor {
    private static final String type = "Barometer";
    private static int numBarometer = 0;

    public Barometer() {
        super( acquisitionTime: 6000, numBarometer, type);
        numBarometer++;
    }

    void generateValue(){super.setActualValue((float)(Math.random() * 65+ 980));}

    public static String getType() { return type; }
}
```

Figura 14: Codice classe **Barometer**

3.1.4 Anemometer

```java
void generateValue(){
    float value;
    if (firstTime) {
        value = (float) (Math.random() * 150);
        firstTime = false;
    }
    else {
        do
            value = super.getActualValue() + ((float) ((Math.random() - 0.5) * 3));
        while (value < 0 || value > 150);
    }
    super.setActualValue(value);
}
```

Figura 15: Codice metodo **generateValue()** della classe **Anemometer**

3.1.5 WindVane

```java
void generateValue(){
    float value;
    if(firstTime) {
        value = (float)(Math.random() * 360);
        firstTime = false;
    }
    else {
        value = super.getActualValue() + ((float) ((Math.random() - 0.5) * 6));
        if (value < 0)
            value = value + 360;
        else if (value > 360)
            value = value + 360;
    }
    super.setActualValue(value);
}
```

Figura 16: Codice metodo **generateValue()** della classe **WindVane**

3.1.6 RainGauge

```
void generateValue() {super.setActualValue((float) (Math.random() * 26));}
```

Figura 17: Codice metodo **generateValue()** della classe **RainGauge**

3.1.7 Thermometer

```
void generateValue(){
    float value;
    if (firstTime) {
        value = (float)((Math.random()*45 - 5));
        firstTime = false;
    }
    else
        value = super.getActualValue() + ((float)((Math.random() - 0.5)* 4));
    super.setActualValue(value);
}
```

Figura 18: Codice metodo **generateValue()** della classe **Thermometer**

3.1.8 Hygrometer

```
void generateValue(){
    float value;
    if (firsTime) {
        value = (float)((Math.random()* 100));
        firsTime = false;
    }
    else {
        do
            value = super.getActualValue() + ((float) ((Math.random() - 0.5) * 8));
        while (value < 0 || value > 100);
    }
    super.setActualValue(value);
}
```

Figura 19: Codice metodo **generateValue()** della classe **Hygrometer**

3.1.9 SensorsSet

SensorsSet è una classe astratta contenente un array associato ad un insieme di sensori. I sensori associati al ***SensorsSet*** sono registrati nelle classi da lei derivate. Questa realizzazione ha permesso alle classi derivate di avere un insieme di sensori ben definito sia in numero che in tipo ma avere anche dei metodi che valgono per tutti questi insiemi.
Ogni ***SensorsSet*** è dotato di un trasmettitore che ha il compito di inviare i dati a chi necessita.
SensorsSet sincronizza l'accesso dei sensori e del trasmettitore, i quali rispettivamente inviano e prelevano dati . La sincronizzazione segue il principio del *Monitor*, secondo il quale, la classe ha dei metodi che se invocati limitano l'accesso ad un solo **Thread**. Tutti i **Thread** che accedono all'array sono messi in attesa (**wait()**) finché il ***SensorsSet*** non dispone di tutti i sensori di cui necessita.
L'inserimento dati nell'array da parte dei sensori avviene tramite l'invocazione del metodo **setValue()**, mentre **getSensorsValue()** permette al trasmettitore di prelevare l'array di valori. Il metodo **setValue()** è realizzato utilizzando il design pattern *template* in quanto la funzione **getSensorPos()** al suo interno è solo dichiarata, lasciando la definizione alle classi derivate.
Ogni ***SensorsSet*** ha un oggetto **SensorsSetID** il quale serve da identificativo.

```
synchronized void setValue(SensorID sensorID, float value) throws InterruptedException {
    while (!full)
        wait();
    int pos = getSensorPos(sensorID);
    if (pos >= 0 && pos <sensorValue.length){
        sensorValue[pos] = value;
    }
    else
        System.out.println("Error!!! An external sensor wants to update the "+sensorsSetID.getType());
}

public synchronized float[] getSensorsValue() throws InterruptedException {
    while (!full)
        wait();
    return Arrays.copyOf(sensorValue, sensorValue.length);
}
```

Figura 20: Snippet classe ***SensorsSet***

3.1.10 SensorsSetID

SensorsSetID è una classe le cui istanze servono da identificativi per oggetti di tipo ***SensorsSet***. Cosi come la classe **SensorID**, i suoi attributi sono:

- **id:** intero che rappresenta il codice identificativo del ***SensorsSet***
- **type:** stringa che rappresenta il tipo del ***SensorsSet***

Il metodo statico **areEqual()** controlla se due oggetti **SensorsSetID** corrispondono attraverso un confronto fra gli attributi.

3.1.11 WeatherStation

WeatherStation è una classe derivata da ***SensorsSet*** che aggrega sei sensori, uno per ogni tipo presente nell'elaborato. Ogni tipo di sensore ha una sua posizione predefinita nell'array della classe padre ***SensorsSet***. **WeatherStation** inoltre ha:

- l'attributo stringa costante (static final) **type** che indica il tipo di ***SensorsSet***
- un attributo intero statico che tiene conto del numero di **WeatherStation** istanziate
- un attributo intero costante che riporta il numero di sensori contenuti in **WeatherStation**

```
public class WeatherStation extends SensorsSet {
    private static final String type = "WeatherStation";
    private static final int barometerPos = 0;
    private static final int anemometerPos = 1;
    private static final int windVanePos = 2;
    private static final int rainGaugePos = 3;
    private static final int thermometerPos = 4;
    private static final int hygrometerPos = 5;
    private static final int maxDevice = 6;
    private static int numWeatherStation = 0;
    private Barometer barometer;
    private Anemometer anemometer;
    private WindVane windVane;
    private RainGauge rainGauge;
    private Thermometer thermometer;
    private Hygrometer hygrometer;

    public WeatherStation(){
        super(maxDevice, numWeatherStation, type);
        numWeatherStation++;
        barometer = null;
        anemometer = null;
        windVane = null;
        rainGauge = null;
        thermometer = null;
        hygrometer = null;
    }
```

Figura 21: Snippet classe **WeatherStation**

3.1.12 IndoorMeter

IndoorMeter è una classe derivata da ***SensorsSet*** che aggrega un termometro ed un igrometro. Il resto dell'implementazione segue quella di **WeatherStation**.
Le istanze di questa classe corrispondono ai termoigrometri per interni di cui si parla nel **Dominio dell'elaborato**.

```
public class IndoorMeter extends SensorsSet {
    private static final String type = "IndoorMeter";
    private static final int thermometerPos = 0;
    private static final int hygrometerPos = 1;
    private static final int maxDevice = 2;
    private static int numIndoorMeter = 0;
    private Thermometer thermometer;
    private Hygrometer hygrometer;

    public IndoorMeter(){
        super(maxDevice, numIndoorMeter, type);
        numIndoorMeter++;
        thermometer = null;
        hygrometer = null;
    }
```

Figura 22: Snippet classe **IndoorMeter**

3.2 Package java.lang

Da questo *package* base fornito da java è stato utilizzato:

- la classe **Thread** che ha permesso la simulazione di sensori e trasmettitori
- l'interfaccia ***Runnable*** che costringe le classi implementanti a definire la funzione **run**(), utile agli oggetti **Thread**

3.3 Package java.util

Da questo *package* base fornito da java è stato utilizzato:

- la classe **Observable** che implementa metodi atti a rendere la classe osservabile da più oggetti di tipo ***Observer***. Tali metodi possono essere utilizzati in modo efficace dalle classi derivate da **Observable**
- l'interfaccia ***Observer*** che costringe le classi implementanti a definire la funzione **update**(), la quale aggiorna lo stato dell'oggetto istanziato al variare dell'**Observable** osservato

Le due classi appena viste sono realizzate seguendo la struttura del design pattern *Observer*. Tutte le realizzazioni del design pattern *Observer* nell'elaborato (centro meteorologico che osserva stazioni e device da ufficio che osserva centro meteorologico e termoigrometro) sono in modalità *PULL*.

3.4 Package transmitters

3.4.1 Transmitter

Transmitter è una classe astratta derivata da **Observable**. ***Transmitter*** ha il compito di prelevare l'array di valori della classe a cui è aggregata e inviarlo agli oggetti ***Observer*** che la osservano. L'array viene prelevato grazie al metodo ***getData()*** la cui definizione è lasciata alle classi derivate. Ad essere aggregata non è direttamente ***Transmitter***, ma la sua classe derivata.
Anche in ***Transmitter*** abbiamo associato per composizione un oggetto **Thread** per simulare un processo in continua esecuzione. L'oggetto passato al **Thread** al momento della costruzione è proprio un ***Transmitter***, quindi la classe implementa l'interfaccia ***Runnable***.
Il metodo **updateState()** aggiorna lo stato del ***Transmitter*** con l'array prelevato da ***getData()*** e notifica gli ***Observer*** se lo stato ha subito variazioni.

```
public abstract class Transmitter extends Observable implements Runnable{
    private float[] state;
    private Thread thread;


    Transmitter(int stateLength) {
        state = new float[stateLength];
        thread = new Thread( target: this);
    }

    @Override
    public void run() {
        try {
            while(true)
                updateState(getData());
        }catch(InterruptedException ignored) {
        }
    }

    private void updateState(float[] state) throws InterruptedException {
        for (int i=0; i<this.state.length; i++) {
            if (this.state[i] != (state[i])) {
                this.state[i] = state[i];
                setChanged();
            }
        }
        if (hasChanged())
            transmit();
        else
            Thread.sleep( millis: 1);
    }
```

Figura 23: Snippet classe ***Transmitter***

3.4.2 SensorsSetTransmitter

SensorsSetTransmitter è una classe derivata da ***Transmitter*** che può essere aggregata da un ***SensorsSet***.
Il metodo **getData()** ereditato da ***Transmitter*** è definito restituendo l'array del ***SensorsSet*** a cui il **SensorsSetTransmitter** è aggregato.
SensorsSetTransmitter sovrascrive il metodo **addObserver()** di **Observable** in modo che, se il ***SensorsSet*** a cui è aggregato è un **IndoorMeter**, allora il numero di ***Observer*** consentiti sia uguale ad 1.

```
float[] getData() throws InterruptedException{
    if (sensorsSet == null)
        return getState();
    else
        return sensorsSet.getSensorsValue();
}

@Override
public void addObserver(Observer observer) {
    if (sensorsSet.getSensorsSetID().getType().equals("IndoorMeter")){
        if (countObservers() == 0)
            super.addObserver(observer);
        else
            System.out.println("This IndoorMeter already has an Observer");}
    else
        super.addObserver(observer);
}
```

Figura 24: Snippet classe **SensorsSetTransmitter**

3.4.3 MeteorologicalCenterTransmitter

MeteorologicalCenterTransmitter è una classe derivata da ***Transmitter*** che può essere aggregata dall'istanza del centro meteorologico.
Il metodo **getData()** ereditato da ***Transmitter*** è definito restituendo tutti i parametri del centro meteorologico sotto forma di array.

```
float[] getData() throws  InterruptedException{
    return  MeteorologicalCenter.getInstance().getParameters();
}
```

Figura 25: Metodo **getData()** della classe **MeteorologicalCenterTransmitter**

3.5 Package meteorologicalcenter

3.5.1 MeteorologicalCenter

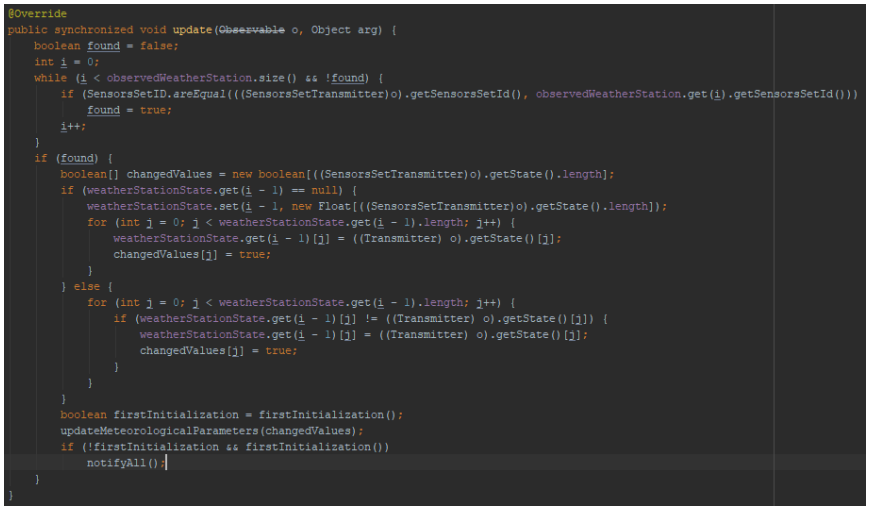
MeteorologicalCenter è una classe che implementa l'interfaccia ***Observer***.
Nel **Dominio dell'elaborato** è specificata la presenza di un unico centro meteorologico, per cui è stato deciso di implementare quest'ultimo come *Singleton*.
Da una classe implementata come *Singleton* si può istanziare un solo oggetto. Per ottenere questa specifica, la classe è stata dotata di un costruttore privato invocabile solo dal metodo pubblico **getInstance()**. Il metodo **getInstance()** restituisce l'istanza **MeteorologicalCenter** se già creata, altrimenti, prima la crea attraveso il costruttore, e poi la restituisce.
Il centro meteorologico ha un attributo per ognuno dei parametri meteorologici di interesse e una lista per gli array di valori delle stazioni che osserva.
Il centro non osserva direttamente le stazioni, ma osserva i trasmettitori da queste aggregate.
Il centro meteorologico ha un trasmettitore che invia i sei parametri controllati a chi ne fa richiesta.
Quando una stazione osservata si aggiorna, notifica il centro invocando il suo **update()**. Durante l'**update()**, il centro aggiorna l'array di valori corrispondente alla stazione chiamante nella lista delle stazioni osservate. Dopodiché i parametri della città sono aggiornati facendo una media fra tutti gli array della lista.

```
public class MeteorologicalCenter implements Observer {
    private static final String type = "MeteorologicalCenter";
    private static int controlledParameters;
    private static MeteorologicalCenter meteorologicalCenterInstance;
    private MeteorologicalCenterTransmitter meteorologicalCenterTransmitter;
    private ArrayList<SensorsSetTransmitter> observedWeatherStation;
    private ArrayList<Float[]> weatherStationState;
    private Float cityAirPressure;
    private Float cityWindSpeed;
    private Float cityWindDirection;
    private Float cityPrecipitation;
    private Float cityTemperature;
    private Float cityHumidity;

    public static MeteorologicalCenter getInstance() {
        if (meteorologicalCenterInstance == null)
            meteorologicalCenterInstance = new MeteorologicalCenter();
        return meteorologicalCenterInstance;
    }

    private MeteorologicalCenter() {
        controlledParameters = 6;
        observedWeatherStation = new ArrayList<>();
        weatherStationState = new ArrayList<>();
        meteorologicalCenterTransmitter = new MeteorologicalCenterTransmitter();
        meteorologicalCenterTransmitter.startTransmitter();
    }
```

Figura 26: Snippet classe **MeteorologicalCenter**



```java
@Override
public synchronized void update(Observable o, Object arg) {
    boolean found = false;
    int i = 0;
    while (i < observedWeatherStation.size() && !found) {
        if (SensorsSetID.areEqual(((SensorsSetTransmitter)o).getSensorsSetId(), observedWeatherStation.get(i).getSensorsSetId()))
            found = true;
        i++;
    }
    if (found) {
        boolean[] changedValues = new boolean[((SensorsSetTransmitter)o).getState().length];
        if (weatherStationState.get(i - 1) == null) {
            weatherStationState.set(i - 1, new Float[((SensorsSetTransmitter)o).getState().length]);
            for (int j = 0; j < weatherStationState.get(i - 1).length; j++) {
                weatherStationState.get(i - 1)[j] = ((Transmitter) o).getState()[j];
                changedValues[j] = true;
            }
        } else {
            for (int j = 0; j < weatherStationState.get(i - 1).length; j++) {
                if (weatherStationState.get(i - 1)[j] != ((Transmitter) o).getState()[j]) {
                    weatherStationState.get(i - 1)[j] = ((Transmitter) o).getState()[j];
                    changedValues[j] = true;
                }
            }
        }
        boolean firstInitialization = firstInitialization();
        updateMeteorologicalParameters(changedValues);
        if (!firstInitialization && firstInitialization())
            notifyAll();
    }
}
```

Figura 27: Metodo **update()** della classe **MeteorologicalCenter**

```java
private void updateMeteorologicalParameters(boolean[] changed) {
    for(int i=0; i<changed.length; i++) {
        if (changed[i]) {
            float sum = 0;
            int j = 0;
            for (Float[] values : weatherStationState) {
                if (values != null) {
                    sum = sum + values[i];
                    j++;
                }
            }
            updateParameter(i,  newValue: sum/j);
        }
    }
}

private void updateParameter(int pos, float newValue) {
    String sensorType = WeatherStation.getSensorType(pos);
    if (sensorType != null) {
        if (sensorType.equals(Barometer.getType()))
            cityAirPressure = newValue;
        else if (sensorType.equals(Anemometer.getType()))
            cityWindSpeed = newValue;
        else if (sensorType.equals(WindVane.getType()))
            cityWindDirection = newValue;
        else if (sensorType.equals(RainGauge.getType()))
            cityPrecipitation = newValue;
        else if (sensorType.equals(Thermometer.getType()))
            cityTemperature = newValue;
        else if (sensorType.equals(Hygrometer.getType()))
            cityHumidity = newValue;
    }
    else
        System.out.println("Error!!! The meteorological center doesn't follow this meteorological parameter.");
}
```

Figura 28: Metodi utili all'**update()** della classe **MeteorologicalCenter**

3.6 Package signalreceiver

3.6.1 OfficeDevice

OfficeDevice è una classe che implementa l'interfaccia ***Observer***.
Come detto nel **Dominio dell'elaborato**, ogni istanza di questa classe osserva il centro meteorologico ed un **IndoorMeter**. Anche in questo caso l'osservazione non è diretta a questi oggetti ma ai trasmettitori che questi aggregano.
Ogni **OfficeDevice** ha un attributo per ogni parametro di interesse: temperatura interna ed esterna, umidità interna ed esterna.
Ogni **OfficeDevice** ha un intervallo di tempo impostato alla costruzione che, se sorpassato, permette l'aggiornamento dei valori su schermo alla seguente invocazione di **update()**. Il dispositivo ha inoltre un attributo per memorizzare a quando risale l'ultimo aggiornamento.
OfficeDevice ha un attributo che tiene conto del numero delle istanze messe in vita. Questo attributo è usato per assegnare un **id** ad ogni istanza.
Il metodo **update()** distingue se è stato invocato dal centro meteorologico o dal termoigrometo in modo che possa aggiornare i parametri di conseguenza.

```
public class OfficeDevice implements Observer {
    private static int numOfficeDevice = 0;
    private int id;
    private int displayTime;
    private SensorsSetTransmitter observedIndoorMeter;
    private long lastUpdate;
    private boolean firstInternalParametersUpdate;
    private boolean firstExternalParametersUpdate;
    private float internalTemperature;
    private float internalHumidity;
    private float externalTemperature;
    private float externalHumidity;

    public OfficeDevice(int displayTime) {
        id = numOfficeDevice;
        this.displayTime = displayTime;
        numOfficeDevice++;
        firstInternalParametersUpdate = true;
        firstExternalParametersUpdate = true;
        MeteorologicalCenter.getInstance().addObserver(this);
    }
```

Figura 29: Snippet classe **OfficeDevice**

```java
@Override
public synchronized void update(Observable o, Object arg){
    if(((Transmitter) o).getTransmitterType().equals("IndoorMeter")) {
        updateInternalParameters(((SensorsSetTransmitter)o));
    }
    else if (((Transmitter) o).getTransmitterType().equals("MeteorologicalCenter")) {
        updateExternalParameters((MeteorologicalCenterTransmitter)o);
    }
    if (System.currentTimeMillis()-lastUpdate >= displayTime && !firstExternalParametersUpdate && !firstInternalParametersUpdate){
        // la funzione display che segue serve a controllare la correttezza del codice
        //display();
        lastUpdate = System.currentTimeMillis();
    }

}
```

Figura 30: Metodo **update()** della classe **OfficeDevice**

```java
private void updateInternalParameters(SensorsSetTransmitter sensorsSetTransmitter){
    internalTemperature = sensorsSetTransmitter.getState()[IndoorMeter.getDevicePos( deviceType: "Thermometer")];
    internalHumidity = sensorsSetTransmitter.getState()[IndoorMeter.getDevicePos( deviceType: "Hygrometer")];
    if(firstInternalParametersUpdate)
        firstInternalParametersUpdate = false;
}

private void updateExternalParameters(MeteorologicalCenterTransmitter meteorologicalCenterTransmitter) {
    externalTemperature = meteorologicalCenterTransmitter.getState()[WeatherStation.getDevicePos( deviceType: "Thermometer")];
    externalHumidity = meteorologicalCenterTransmitter.getState()[WeatherStation.getDevicePos( deviceType: "Hygrometer")];
    if(firstExternalParametersUpdate)
        firstExternalParametersUpdate = false;
}
```

Figura 31: Metodi utili all'**update()** della classe **OfficeDevice**

4 Testing ed Esecuzione

4.1 Test con JUnit

4.1.1 IndoorMeterTest

IndoorMeterTest è stata concepita principalmente per verificare l'effettivo funzionamento dei metodi per attaccare e rimuovere i sensori e il trasmettitore di un'istanza **IndoorMeter**.

```
@Test
public void detachSensorsSetTransmitterTest() {
    Assert.assertNotNull(indoorMeter.getSensorsSetTransmitter());
    indoorMeter.detachSensorsSetTransmitter();
    Assert.assertNull(indoorMeter.getSensorsSetTransmitter());
}

@Test
public void attachSensorsSetTransmitterTest() {
    indoorMeter.detachSensorsSetTransmitter();
    indoorMeter.attachSensorsSetTransmitter(new SensorsSetTransmitter(indoorMeter.getNumSensors()));
    Assert.assertNotNull(indoorMeter.getSensorsSetTransmitter());
}

@Test
public void attachThermometerTest() {
    Thermometer thermometer = new Thermometer();
    Assert.assertNull(indoorMeter.getThermometer());
    indoorMeter.attachThermometer(thermometer);
    Assert.assertEquals(indoorMeter.getThermometer(), thermometer);
}

@Test
public void detachThermometerTest() {
    Thermometer thermometer = new Thermometer();
    indoorMeter.attachThermometer(thermometer);
    indoorMeter.detachThermometer();
    Assert.assertNull(indoorMeter.getThermometer());
}

@Test
public void attachHygrometerTest() {
    Hygrometer hygrometer = new Hygrometer();
    Assert.assertNull(indoorMeter.getHygrometer());
    indoorMeter.attachHygrometer(hygrometer);
    Assert.assertEquals(indoorMeter.getHygrometer(), hygrometer);
}

@Test
public void detachHygrometerTest() {
    Hygrometer hygrometer = new Hygrometer();
    indoorMeter.attachHygrometer(hygrometer);
    indoorMeter.detachHygrometer();
    Assert.assertNull(indoorMeter.getHygrometer());
}
```

Figura 32: Snippet classe test **IndoorMeterTest**

4.1.2 MeteorologicalCenterTest

MeteorologicalCenterTest è stata realizzata per testare i metodi per attaccare e rimuovere le stazioni e il trasmettitore dal **MeteorologicalCenter**.

```
public class MeteorologicalCenterTest {
    @Test
    public void attachAndDetachWeatherStationTest() {
        WeatherStation weatherStation = new WeatherStation();
        Assert.assertEquals(MeteorologicalCenter.getInstance().getObservedWeatherStation().size(), actual: 0);
        MeteorologicalCenter.getInstance().attachWeatherStation(weatherStation);
        Assert.assertEquals(MeteorologicalCenter.getInstance().getObservedWeatherStation().size(), actual: 1);
        MeteorologicalCenter.getInstance().detachWeatherStation(weatherStation);
        Assert.assertEquals(MeteorologicalCenter.getInstance().getObservedWeatherStation().size(), actual: 0);
    }

    @Test
    public void attachAndDetachMeteorologicalCenterTransmitter() {
        Assert.assertNotNull(MeteorologicalCenter.getInstance().getMeteorologicalCenterTransmitter());
        MeteorologicalCenter.getInstance().detachMeteorologicalCenterTransmitter();
        Assert.assertNull(MeteorologicalCenter.getInstance().getMeteorologicalCenterTransmitter());
        MeteorologicalCenter.getInstance().attachMeteorologicalCenterTransmitter(new MeteorologicalCenterTransmitter());
        Assert.assertNotNull(MeteorologicalCenter.getInstance().getMeteorologicalCenterTransmitter());
    }
}
```

Figura 33: Classe test **MeteorologicalCenterTest**

4.1.3 MeteorologicalCenterTransmitterTest

MeteorologicalCenterTransmitterTest è stata realizzata per testare il metodo **getState()** della classe **MeteorologicalCenterTransmitter**.
Una volta invocato il metodo **getState()**, si verifica che ogni elemento dell'array rientri nel range di valori consentiti per quel parametro. Così facendo si verifica anche se la trasmissione dati da stazione a centro meteorologico si svolge con successo.

```
public class MeteorologicalCenterTransmitterTest {

    @Test
    public void getStateTest() throws InterruptedException{
        WeatherStation weatherStation = new WeatherStation();
        Assert.assertEquals(MeteorologicalCenter.getInstance().getObservedWeatherStation().size(), actual: 0);
        MeteorologicalCenter.getInstance().attachWeatherStation(weatherStation);
        Assert.assertEquals(MeteorologicalCenter.getInstance().getObservedWeatherStation().size(), actual: 1);
        Barometer barometer = new Barometer();
        weatherStation.attachBarometer(barometer);
        Anemometer anemometer = new Anemometer();
        weatherStation.attachAnemometer(anemometer);
        WindVane windVane = new WindVane();
        weatherStation.attachWindVane(windVane);
        RainGauge rainGauge = new RainGauge();
        weatherStation.attachRainGauge(rainGauge);
        Thermometer thermometer = new Thermometer();
        weatherStation.attachThermometer(thermometer);
        Hygrometer hygrometer = new Hygrometer();
        weatherStation.attachHygrometer(hygrometer);
        TimeUnit.SECONDS.sleep( timeout: 2);
        float[] temp = MeteorologicalCenter.getInstance().getMeteorologicalCenterTransmitter().getState();
        Assert.assertTrue( condition: temp[0]>=980 && temp[0]<=1045);
        Assert.assertTrue( condition: temp[1]>=0 && temp[1]<=150);
        Assert.assertTrue( condition: temp[2]>=0 && temp[2]<=360);
        Assert.assertTrue( condition: temp[3]>=0 && temp[3]<=26);
        Assert.assertTrue( condition: temp[4]>=-5 && temp[4]<=40);
        Assert.assertTrue( condition: temp[5]>=0 && temp[5]<=100);
    }
}
```

Figura 34: Classe test **MeteorologicalCenterTransmitterTest**

4.1.4 OfficeDeviceTest

OfficeDeviceTest è stata realizzata per testare i metodi per attaccare e rimuovere l'oggetto **IndoorMeter** da un **OfficeDevice**.

```
public class OfficeDeviceTest {

    @Test
    public void attachAndDetachIndoorMeterTest() {
        OfficeDevice officeDevice = new OfficeDevice( displayTime: 5000);
        Assert.assertNull(officeDevice.getObservedIndoorMeter());
        officeDevice.attachIndoorMeter(new IndoorMeter());
        Assert.assertNotNull(officeDevice.getObservedIndoorMeter());
        officeDevice.detachIndoorMeter();
        Assert.assertNull(officeDevice.getObservedIndoorMeter());
    }
}
```

Figura 35: Classe test **OfficeDeviceTest**

4.1.5 SensorsSetTransmitterTest

SensorsSetTransmitterTest è stata realizzata principalmente per verificare il funzionamento delle funzioni **getState**() e **addObserver**() della classe **SensorsSetTransmitter**.
Nel testare il metodo **getState**(), si verifica se ogni elemento dell'array rientra nel range di valori consentiti per quel parametro. Così facendo è stato anche verificato che il **generateValue**() delle classi derivate da ***Sensor*** sia correttamente implementato.

```
@Test
public void addObserverTest() {
    OfficeDevice officeDevice1 = new OfficeDevice( displayTime: 3000);
    OfficeDevice officeDevice2 = new OfficeDevice( displayTime: 500);
    IndoorMeter indoorMeter = new IndoorMeter();
    sensorsSetTransmitter = indoorMeter.getSensorsSetTransmitter();
    Assert.assertEquals(sensorsSetTransmitter.countObservers(), actual: 0);
    sensorsSetTransmitter.addObserver(officeDevice1);
    Assert.assertEquals(sensorsSetTransmitter.countObservers(), actual: 1);
    sensorsSetTransmitter.addObserver(officeDevice2);
    Assert.assertEquals(sensorsSetTransmitter.countObservers(), actual: 1);
}

@Test
public void getStateTest() throws  InterruptedException{
    WeatherStation weatherStation = new WeatherStation();
    IndoorMeter indoorMeter = new IndoorMeter();
    Barometer barometer = new Barometer();
    weatherStation.attachBarometer(barometer);
    Anemometer anemometer = new Anemometer();
    weatherStation.attachAnemometer(anemometer);
    WindVane windVane = new WindVane();
    weatherStation.attachWindVane(windVane);
    RainGauge rainGauge = new RainGauge();
    weatherStation.attachRainGauge(rainGauge);
    Thermometer thermometer = new Thermometer();
    weatherStation.attachThermometer(thermometer);
    Thermometer thermometer1 = new Thermometer();
    indoorMeter.attachThermometer(thermometer1);
    Hygrometer hygrometer = new Hygrometer();
    weatherStation.attachHygrometer(hygrometer);
    Hygrometer hygrometer1 = new Hygrometer();
    indoorMeter.attachHygrometer(hygrometer1);
    TimeUnit.SECONDS.sleep( timeout: 1);
    float[] temp = weatherStation.getSensorsSetTransmitter().getState();
    float[] temp1 = indoorMeter.getSensorsSetTransmitter().getState();
    Assert.assertTrue( condition: temp[0]>=980 && temp[0]<=1045);
    Assert.assertTrue( condition: temp[1]>=0 && temp[1]<=150);
    Assert.assertTrue( condition: temp[2]>=0 && temp[2]<=360);
    Assert.assertTrue( condition: temp[3]>=0 && temp[3]<=26);
    Assert.assertTrue( condition: temp[4]>=-5 && temp[4]<=40);
    Assert.assertTrue( condition: temp[5]>=0 && temp[5]<=100);
    Assert.assertTrue( condition: temp1[0]>=-5 && temp1[0]<=40);
    Assert.assertTrue( condition: temp1[1]>=0 && temp1[1]<=100);
}
```

Figura 36: Snippet classe test **SensorsSetTransmitterTest**

4.1.6 WeatherStationTest

Così come **IndoorMeterTest**, questa classe è stata realizzata principalmente per verificare l'effettivo funzionamento dei metodi per attaccare e rimuovere i sensori ed il trasmettitore da un'istanza di **WeatherStation**.

4.2 Test con classe Main

Per confermare l'effettivo funzionamento del codice dell'elaborato, è stata realizzata una classe **Main** contenuta nel package *main* della cartella *src*.
Nella classe è implementato un **main()** che verifica il funzionamento del codice nel caso in cui il centro meteorologico osservi tre stazioni (complete di tutti i sensori che necessitano) e invii i dati a due **OfficeDevice**.
Ogni **OfficeDevice** osserva un oggetto **IndoorMeter** (completo di tutti i sensori che gli servono).
Una volta che questi oggetti sono stati messi in vita, i **Thread** presenti lavorano per 15 secondi prima di essere interrotti.

Nell'elaborato è stato assegnato ad ogni sensore un tempo per la generazione di valori:

- **Barometro:** 6 secondi
- **Anemometro:** 4 secondi
- **Banderuola:** 4 secondi
- **Pluviometro:** 7 secondi
- **Termometro:** 8 secondi
- **Igrometro:** 10 secondi

Considerando i 15 secondi di lavoro dei **Thread**, si ottiene il seguente schema di generazione valori per una stazione:

	Barometro	Anemometro	Banderuola	Pluviometro	Termometro	Igrometro
0 s	✓	✓	✓	✓	✓	✓
1 s						
2 s						
3 s						
4 s		✓	✓			
5 s						
6 s	✓					
7 s				✓		
8 s		✓	✓		✓	
9 s						
10 s						✓
11 s						
12 s	✓	✓	✓			
13 s						
14 s				✓		
15 s						

Figura 37: Schema generazione valori per una stazione

Al secondo 0, il trasmettitore della stazione invierà l'array di valori solo quando tutti i sensori saranno connessi alla stazione stessa. Un sensore genera il suo primo valore nel momento in cui si attacca alla stazione.
Negli istanti di tempo in cui più di un sensore genera un valore, il numero di trasmissioni dati da parte del trasmettitore della stazione dipende dal momento in cui questo accede all'array di valori.
Prendiamo come esempio il secondo 12. Il trasmettitore può inviare l'array già dal momento in cui uno dei tre sensori ha inserito il nuovo valore. Quindi al secondo 12 il trasmettitore potrebbe effettuare da 1 a 3 trasmissioni dati. Il numero di trasmissioni dipende dall'ordine di accesso dei **Thread** alla stazione.

Per valutare che sia rispettato lo schema di "Figura 37", sono state aggiunte delle linee di codice all'**update()** di **MeteorologicalCenter** che stampano ad ogni aggiornamento di una stazione il suo identificativo, i suoi parametri e, subito sotto, i parametri della città aggiornati. L'ordine di stampa dei parametri è: **Barometro**, **Anemometro**, **Banderuola**, **Pluviometro**, **Termometro** e **Igrometro**.
Segue il risultato di un'esecuzione:

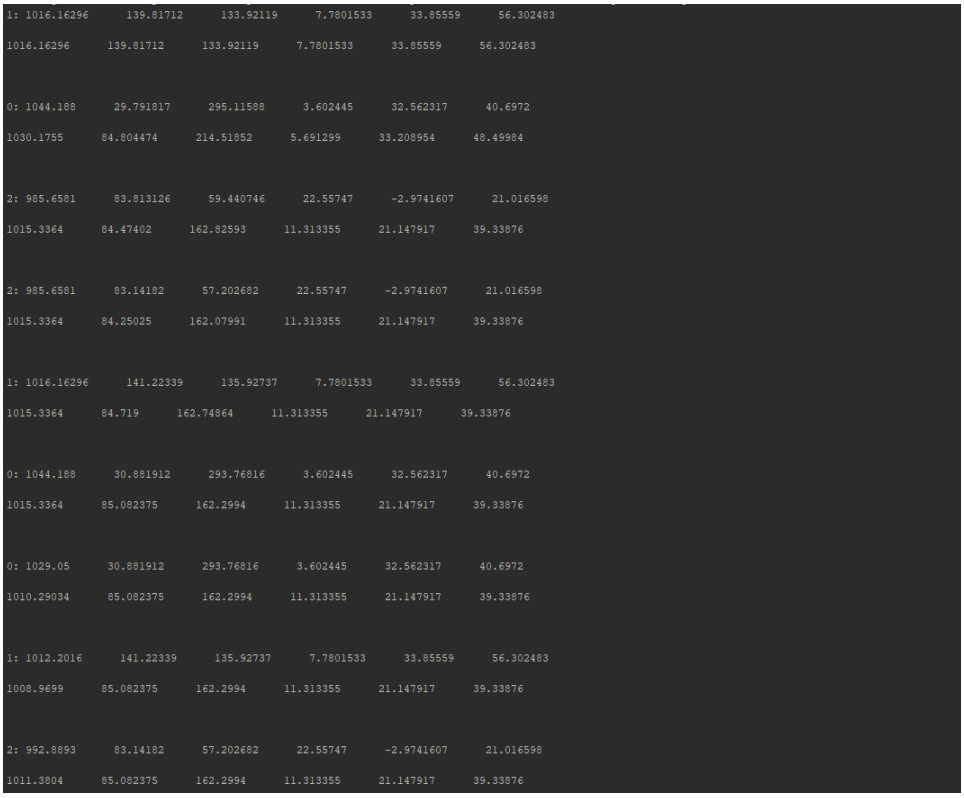


Figura 38: Stampa trasmissione valori da stazione a centro parte 1/4

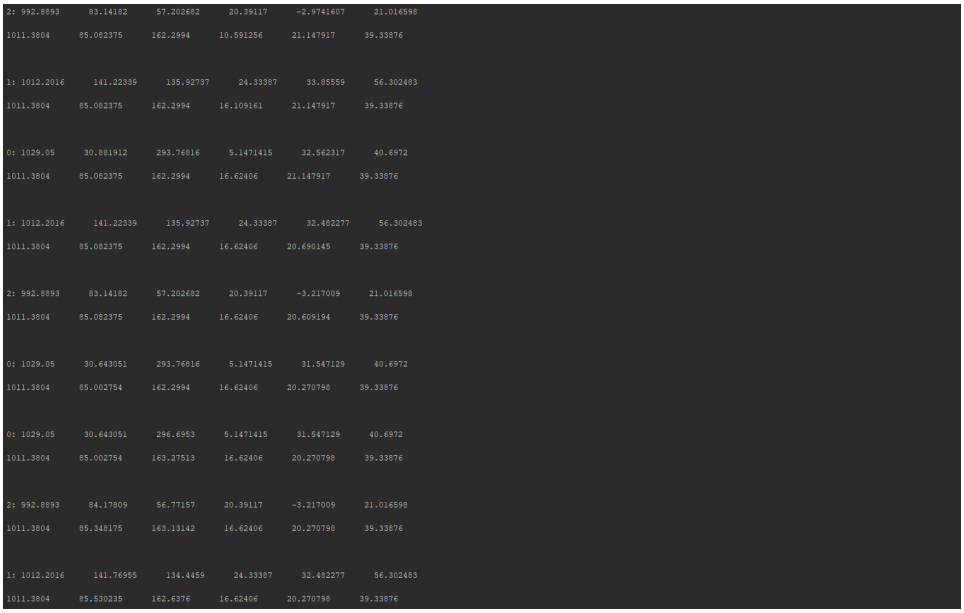


Figura 39: Stampa trasmissione valori da stazione a centro parte 2/4

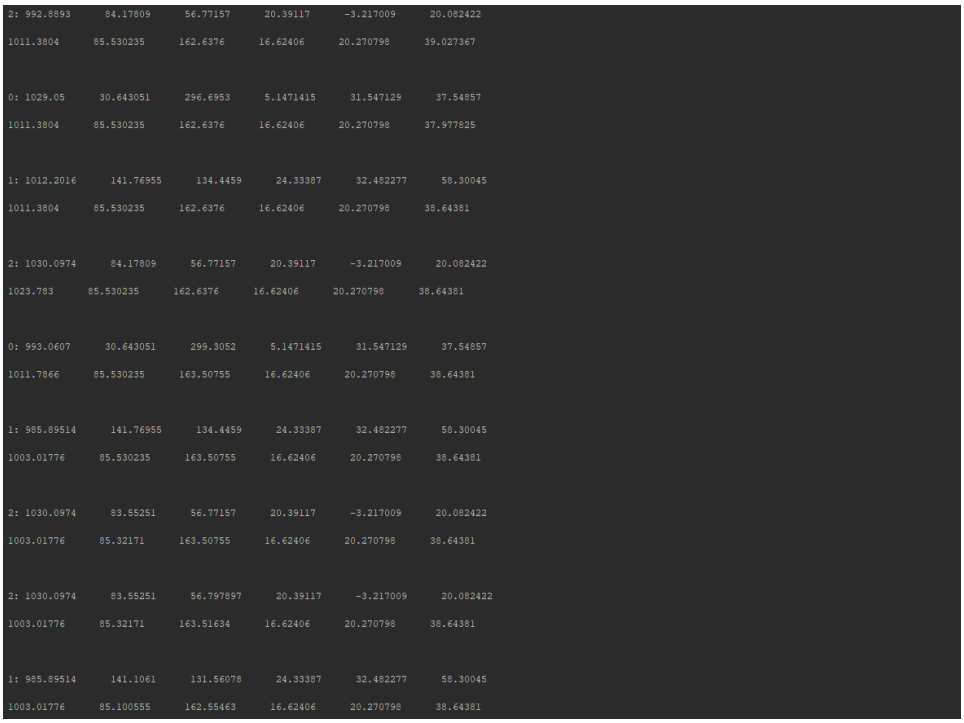


Figura 40: Stampa trasmissione valori da stazione a centro parte 3/4

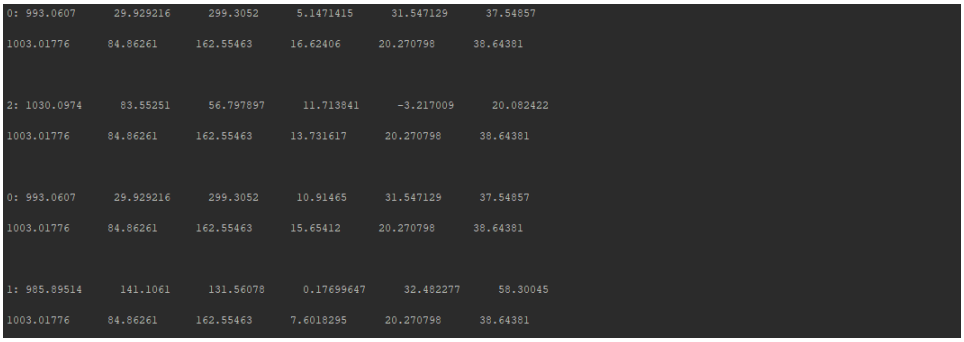


Figura 41: Stampa trasmissione valori da stazione a centro parte 4/4

É stata anche eseguita un'esecuzione del **main()** per controllare la validità dell'**update()** dell'**OfficeDevice**.
In questo caso è stato aggiunto all'**update()** una funzione **display()** che stampa i parametri dell'**OfficeDevice**. La funzione **display()** in realtà dovrebbe aggiornare i dati a schermo dell'**OfficeDevice**.
L'intervallo di aggiornamento dello schermo è 6 secondi per il primo device e 7 per il secondo. Quindi entrambi faranno 3 stampe: 0, 5 e 10 secondi il primo e 0, 6 e 12 il secondo. Per il primo device dovrebbe esserci una stampa dei valori anche al secondo 15, ma questa è assente perché l'esecuzione del **main()** finisce prima che l'**update()** sia chiamato.
Seguono i risultati per un'esecuzione:

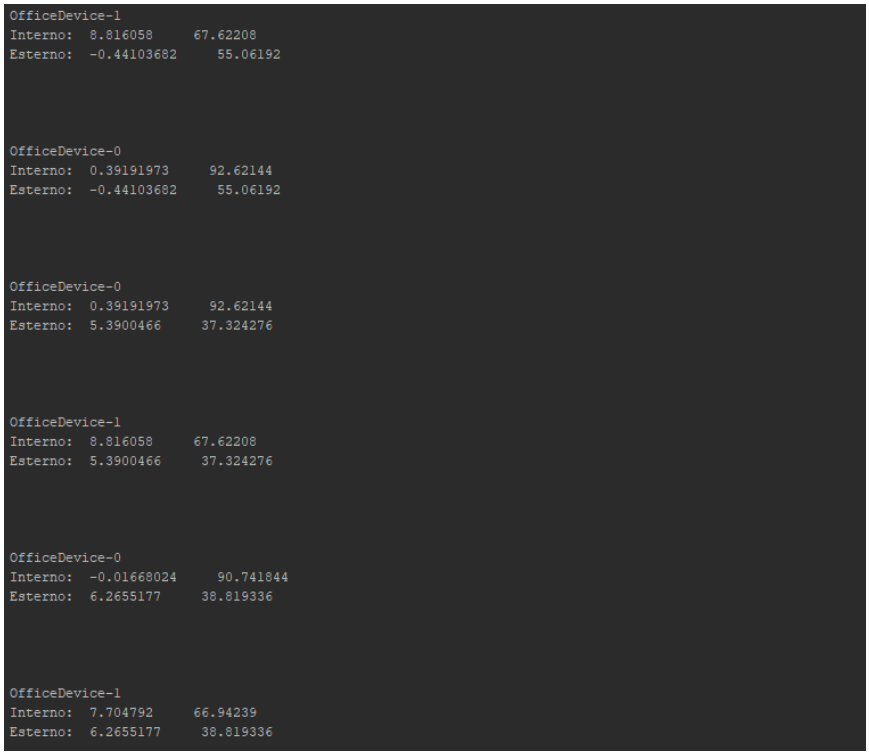


Figura 42: Stampa aggiornamento video dei device da ufficio

Nel caso si volesse replicare questi due tipi di test bisogna scommentare le parti interessate nei metodi **update()** sopra indicati.
Si consiglia di non tenere scommentate le parti di interesse in contemporanea, perché essendo il centro meteorologico e i device di ufficio oggetti distinti, potrebbero entrare in conflitto nel caso facciano richiesta di stampa contemporaneamente. Gli **OfficeDevice** visti nell'esempio prima, anche se sono oggetti distinti, non soffrono dello stesso problema perché il tempo di aggiornamento scelto per loro è diverso. Inoltre, un altro meccanismo che li tutela dal conflitto, è dato dal **notifyObservers()** del centro, che invoca **update()** in entrambi i dispositivi ma in tempi diversi. Quindi l'unica vera fonte di conflitto rimanente è quando i due termoigrometri aggiornano contemporaneamente i dispositivi.

Per avere un'idea più chiara di come sono in relazione tutte le istanze, è stato realizzato un diagramma simile al *Collaboration*. Nel diagramma sono rappresentate le istanze messe in vita e gli archi che le connettono. Il verso degli archi dipende da quale delle due istanze invoca i metodi dell'altra.
Le istanze aventi un **Thread** sono divise in gruppi colorati. I **Thread** sono sincronizzati fra loro solo se appartengono a gruppi dello stesso colore.

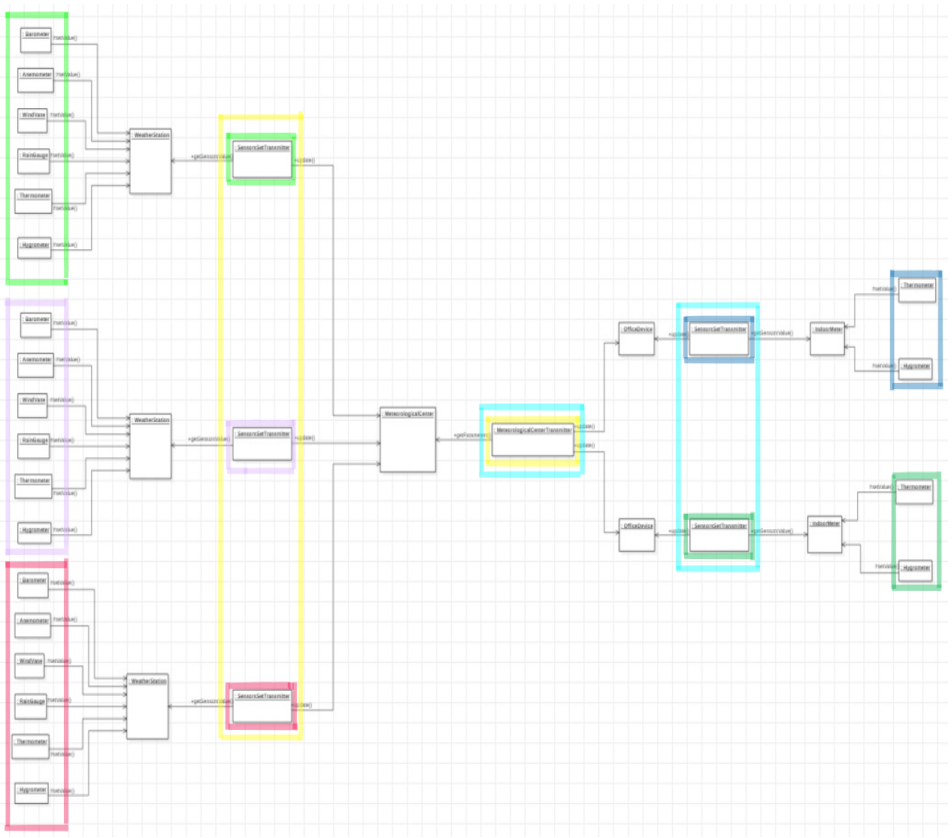


Figura 43: Diagramma sul rapporto fra istanze e sincronizzazione dei **Thread**

4.3 Sequence Diagram

É stato realizzato un *Sequence Diagram* che analizza il percorso dati dalla loro generazione fino agli schermi dei dispositivi da ufficio.
Nel diagramma sono state seguite due strade:

- i dati generati da un termometro vengono raccolti in un termoigrometro che poi li invia al device da ufficio
- i dati generati da un termometro vengono raccolti da una stazione, spediti al centro meteorologico e poi inviati allo stesso dispositivo da ufficio di cui sopra

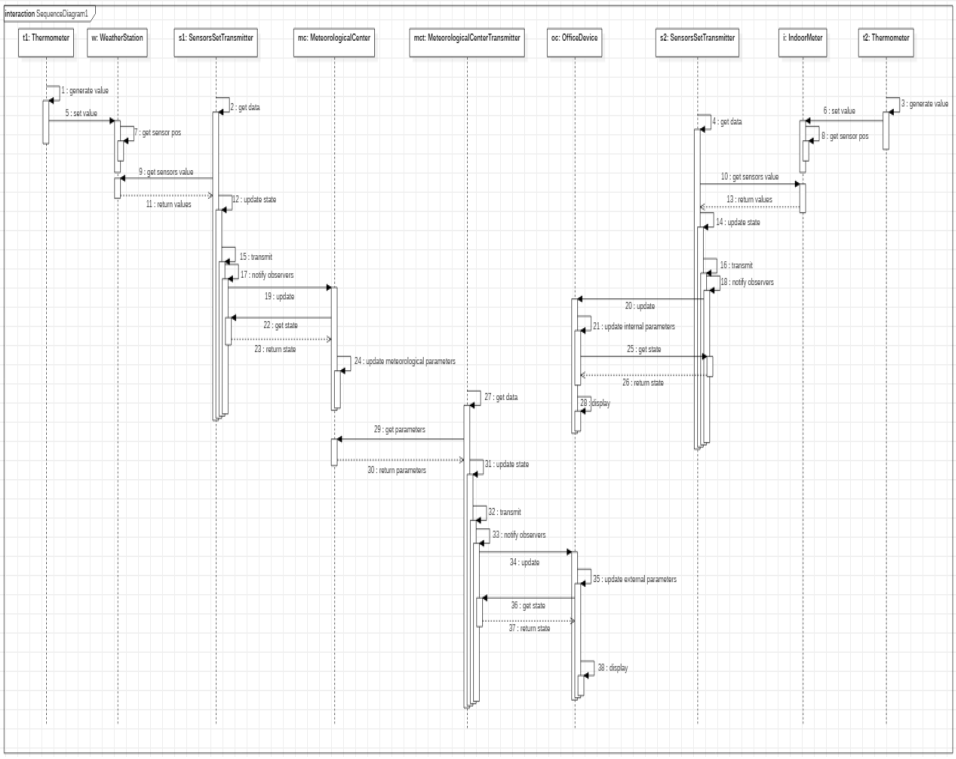


Figura 44: Sequence Diagram del percorso dati