Esame di Ingegneria del Software

Gemma Vaggelli

6348717

Elaborato di Ingegneria del Software

1 Intento

Il Model View Controller è un design pattern largamente utilizzato nello sviluppo del codice. L'MVC divide un'applicazione in tre parti interconnesse tra di loro, al fine di separare la rappresentazione interna delle informazioni, dai modi in cui l'informazione è rappresentata e accettata dall'utente. In questo programma si utilizza il Model-View-Controller (MVC) al fine di separare l'interfaccia utente dal modello di dati (temperatura) e dal codice utilizzato per collegare i due. L'intento è quello di realizzare un termostato secondo il pattern MVC. Il termostato è composto da tre scale: Kelvin, Farenheit, Celsius; un display e due bottoni adibiti uno all' incremento e l'altro al decremento. L'obiettivo da raggiungere è che quando si modifica la scala di riferimento, a seguito di innalzamenti o abbassamenti della temperatura, ciò avvenga in maniera concorde al sistema di misurazione selezionato, non permettendo di oltrepassare la temperatura minima di -10 gradi Celsius e quella massima di 70 gradi Celsius.

All'interno dell'elaborato, l'MVC è cosi ripartito:

- Model- TemperatureModel
- View - View
- Controller - i pattern Strategy e Decorator

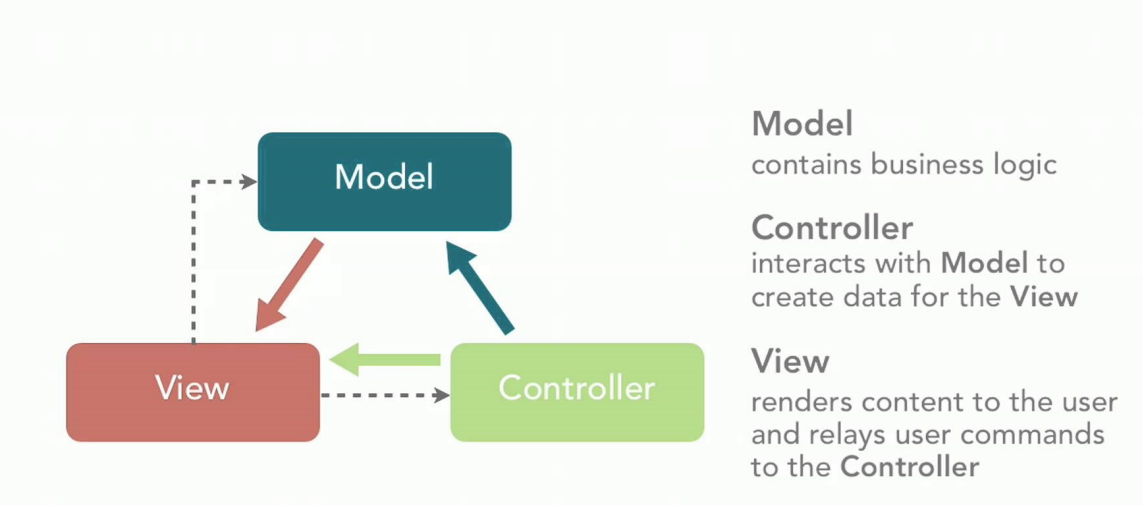

Figura 1: Schema MVC

2 Documentazione

2.1 Class Diagram

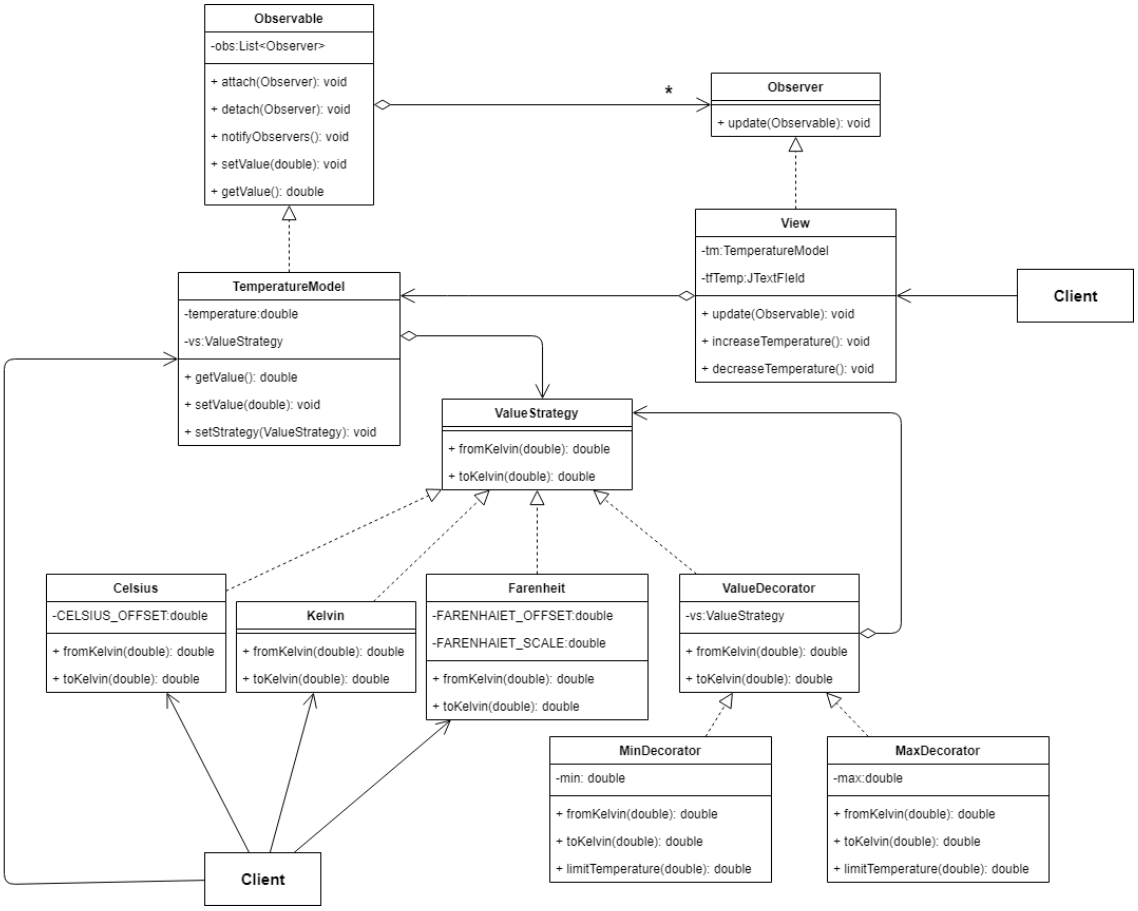

Figura 2: Class Diagram

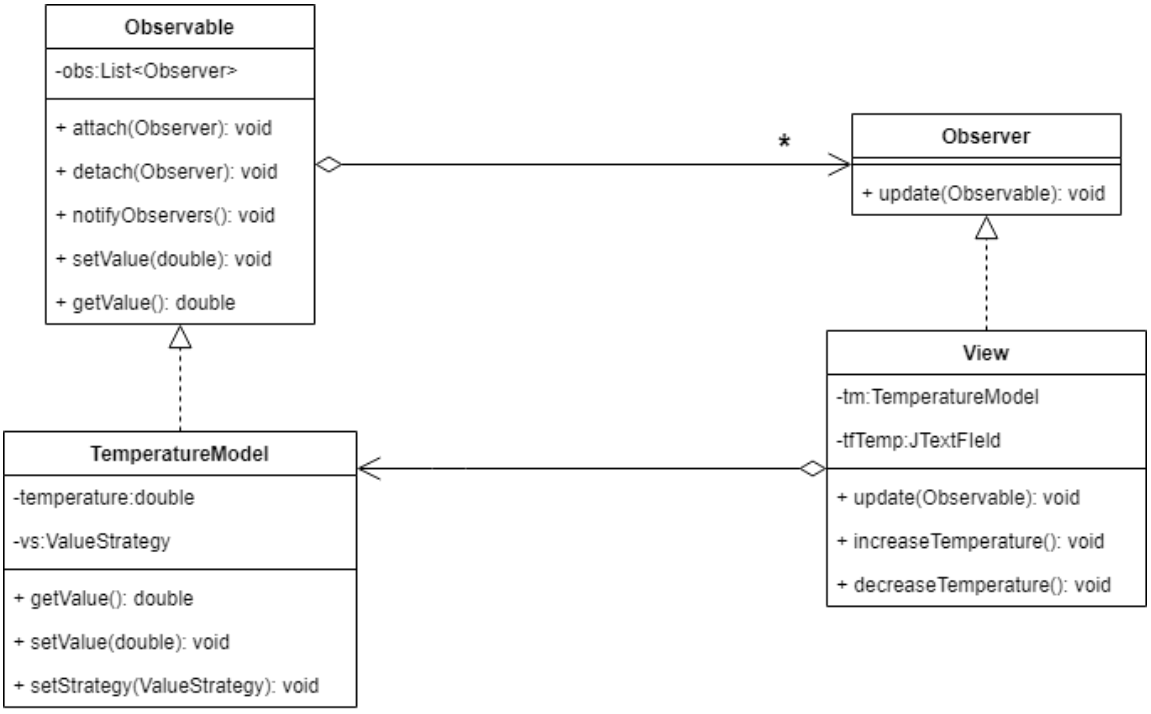

Figura 3: Observer Pattern Class Diagram

Il pattern Observer è formato da uno o più Observer che vengono registrati per gestire un evento che potrebbe essere generato dall'oggetto "osservato", che può essere chiamato Subject o Observable. Oltre all'Observer esiste il concrete Observer, che si differenzia dal primo in quanto implementa direttamente le azioni da compiere in risposta ad un cambiamento di stato del Concrete Subject. Il pattern può essere implementato in modo push o pull. Nel modello push, il Subject invia agli Observer tutti i dati riguardanti il suo stato, di cui essi necessitano. Il Subject deve fare assunzioni sulle informazioni utili ai vari Observer. Nel modello pull, più flessibile del precedente, dopo avere ricevuto una notifica, sono gli Observer a richiedere al Subject le informazioni di cui necessitano. Nel nostro caso, le classi TemperatureModel e View interagiscono tra loro attraverso il suddetto pattern. Il TemperatureModel fa da Observable concreto e quando aggiorna il suo stato, notifica la View (Observer concreto) che si ridisegna.

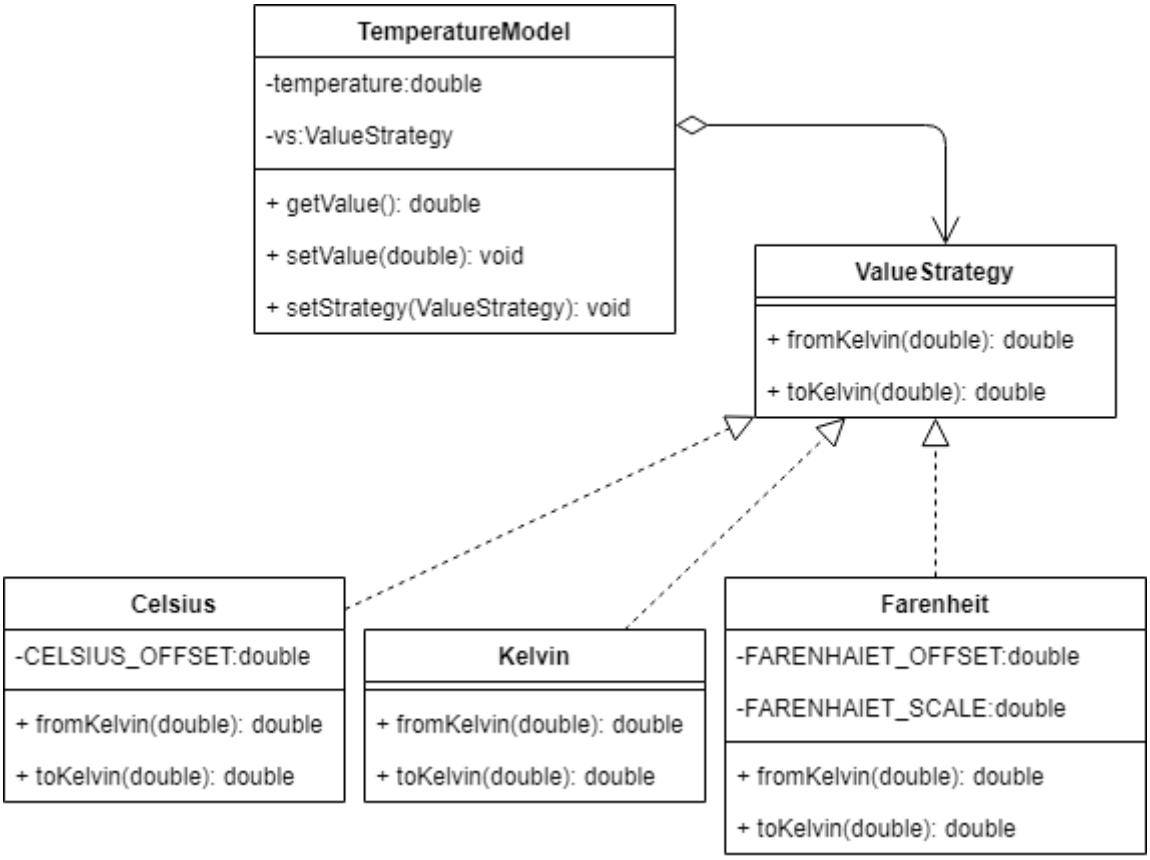

Figura 4: Strategy Pattern Class Diagram

L'obiettivo del pattern Strategy è isolare un algoritmo all'interno di un oggetto, in maniera tale da risultare utile in quelle situazioni dove sia necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione. Questo pattern prevede che gli algoritmi siano intercambiabili tra loro, in base ad una specificata condizione, in modalità trasparente al Client che ne fa uso. In altre parole, data una famiglia di algoritmi che implementa una certa funzionalità, come può essere ad esempio un algoritmo di visita oppure di ordinamento, essi dovranno esportare sempre la medesima interfaccia, così il Client dell'algoritmo non dovrà fare nessuna assunzione su quale sia la strategia concreta istanziata in un particolare istante. Dover presentare sia gradi Fahrenheit che gradi Celsius nella stessa applicazione, creando così un modello per ciascuno, è poco efficiente. L'utilizzo di un design pattern Strategy è più adatto. La ValueStrategy è realizzata come un' interfaccia che definisce la firma dei metodi di conversione e CelsiusStrategy, KelvinStrategy e FarenheitStrategy la

implementano definendo il modo in cui vengono effettuate le conversioni.

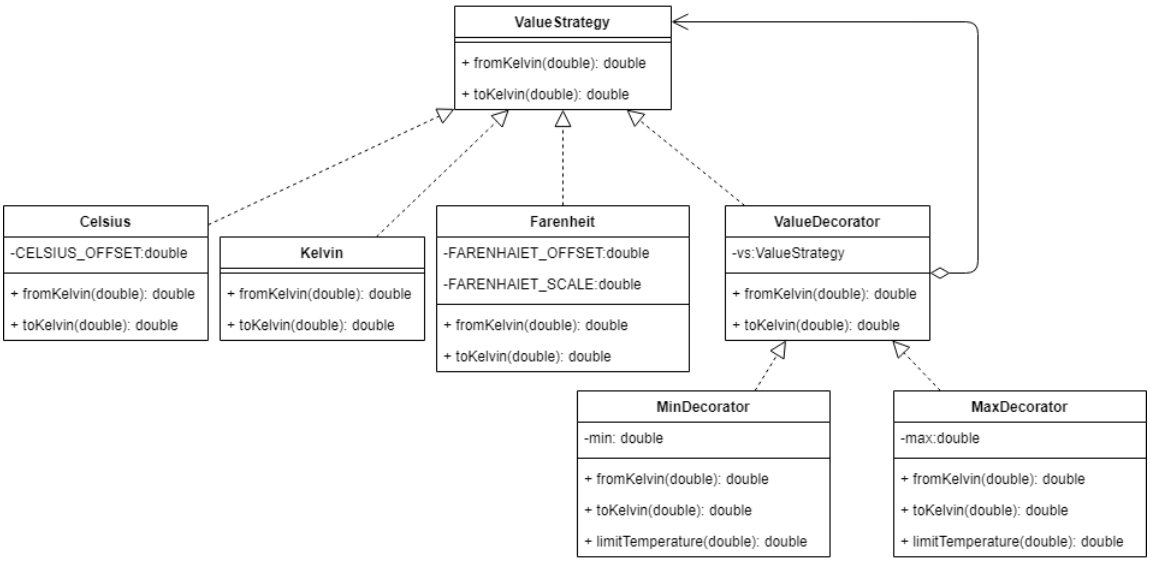

Figura 5: Decorator Pattern Class Diagram

Il pattern Decorator viene utilizzato per fornire funzionalità aggiuntive a un oggetto di qualche tipo. Il concetto base di un pattern Decorator è che "avvolge" l'oggetto decorato e appare al Client esattamente come l'oggetto avvolto. Ciò significa che il decoratore implementa la stessa interfaccia dell'oggetto che decora. Si può pensare a un decoratore come a un guscio attorno all'oggetto decorato. Qualsiasi messaggio inviato da un Client all'oggetto viene invece catturato dal decoratore. Il decoratore può applicare alcune azioni e quindi passare il messaggio ricevuto all'oggetto decorato. Probabilmente quell'oggetto restituisce un valore al decoratore che può nuovamente applicare un'azione a quel risultato, inviando infine il risultato al Client originale. Per il Client, il decoratore è invisibile. In questo programma, il Decorator consente di impostare le temperature minime e massime.Sono stati creati due decoratori, MinDecorator e MaxDecorator, che controllano che la temperatura, impostata o letta, non superi un limite minimo o massimo.

2.2 Sequence Diagram

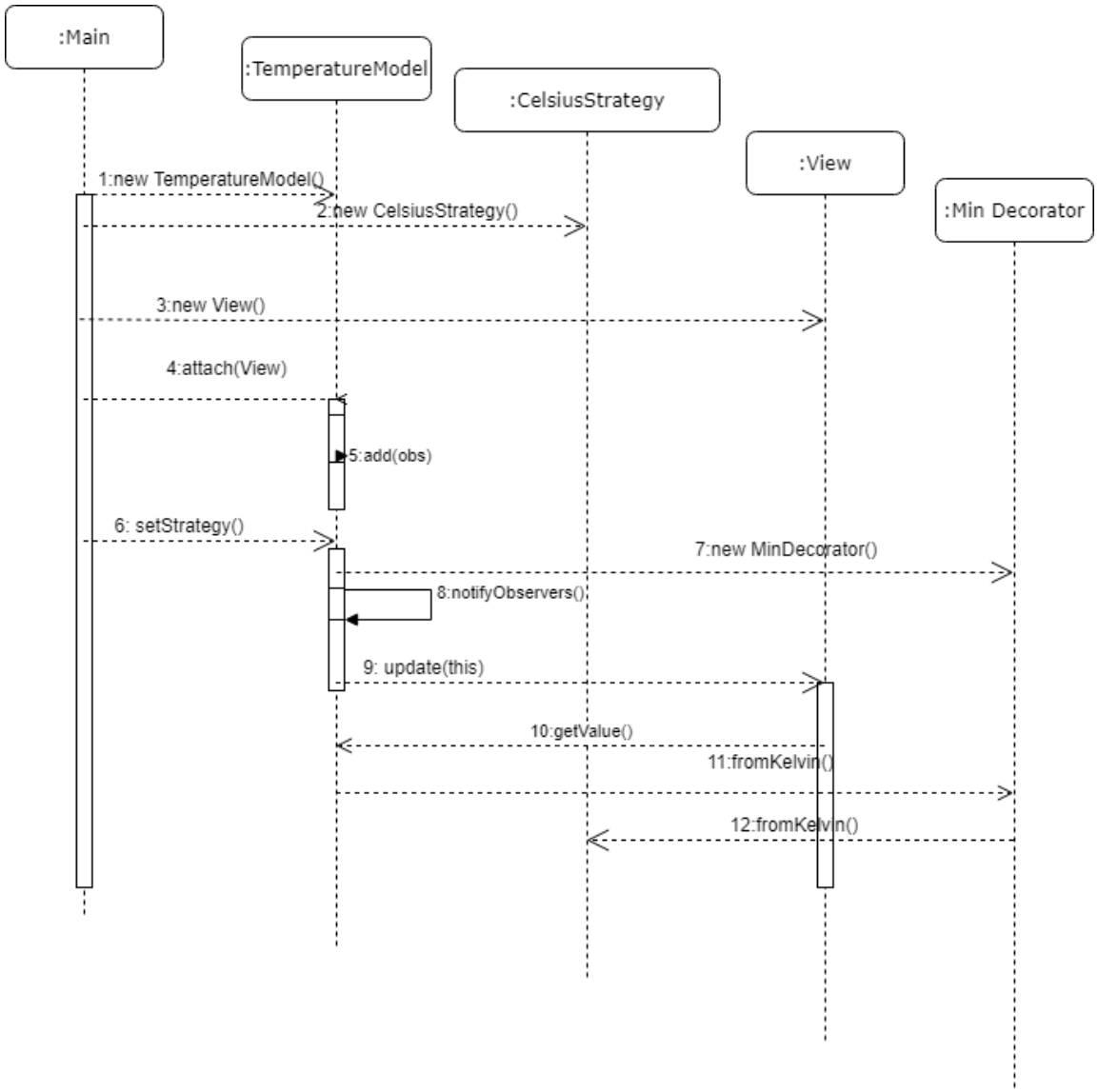

Figura 6: Sequence Diagram

3 Codice

Classe ValueStrategy

```
public interface ValueStrategy {
    public double fromKelvin(double temperature);

    public double toKelvin(double temperature);
}
```

Classe CelsiusStrategy

```
public class CelsiusStrategy implements ValueStrategy {
    private double CELSIUS_OFFSET = 273.15;

    @Override
    public double fromKelvin(double temperature) {
        return temperature - CELSIUS_OFFSET;
    }

    @Override
    public double toKelvin(double temperature) {
        return temperature + CELSIUS_OFFSET;
    }

}
```

Classe FarenheitStrategy

```
public class FarenheitStrategy implements ValueStrategy {
    private double FARENHEIT_OFFSET = 459.67;
    private double FARENHEIT_SCALE = ((double) 9 / 5);

    @Override
    public double fromKelvin(double temperature) {
        return (temperature * FARENHEIT_SCALE) -
           FARENHEIT_OFFSET;
    }

    @Override
    public double toKelvin(double temperature) {
        return (temperature + FARENHEIT_OFFSET) /
           FARENHEIT_SCALE;
    }
```

```
14
15 }
```

Classe KelvinStrategy

```
1  public class KelvinStrategy implements ValueStrategy {
2      public double fromKelvin(double temperature) {
3          return temperature;
4      }
5
6      public double toKelvin(double temperature) {
7          return temperature;
8      }
9  }
```

Classe ValueDecorator

```
1  public abstract class ValueDecorator implements ValueStrategy
       {
2      private ValueStrategy vs;
3
4      public ValueDecorator(ValueStrategy vs) {
5          this.vs = vs;
6
7      }
8
9      @Override
10     public double fromKelvin(double temperature) {
11         return vs.fromKelvin(temperature);
12     }
13
14     @Override
15     public double toKelvin(double temperature) {
16         return vs.toKelvin(temperature);
17     }
18
19 }
```

Classe MaxDecorator

```
1  public class MaxDecorator extends ValueDecorator {
2      private double max;
3
4      public MaxDecorator(ValueStrategy vs, double max) {
```

```
5           super(vs);
6           this.max = max;
7
8       }
9
10      @Override
11      public double fromKelvin(double temperature) {
12          return super.fromKelvin(limitTemperature(temperature))
                ;
13      }
14
15      @Override
16      public double toKelvin(double temperature) {
17          return limitTemperature(super.toKelvin(temperature));
18      }
19
20      private double limitTemperature(double temperature) {
21          if (temperature > max) {
22              return max;
23          } else {
24              return temperature;
25          }
26      }
27
28  }
```

Classe MinDecorator

```
1   public class MinDecorator extends ValueDecorator {
2       private double min;
3
4       public MinDecorator(ValueStrategy vs, double min) {
5           super(vs);
6           this.min = min;
7
8       }
9
10      @Override
11      public double fromKelvin(double temperature) {
12          return super.fromKelvin(limitTemperature(temperature))
                ;
13      }
```

```
14
15      @Override
16      public double toKelvin(double temperature) {
17          return limitTemperature(super.toKelvin(temperature));
18      }
19
20      private double limitTemperature(double temperature) {
21          if (temperature < min) {
22              return min;
23          } else {
24              return temperature;
25          }
26      }
27
28  }
```

Classe Observable

```
1   import java.util.ArrayList;
2
3   public abstract class Observable {
4       private ArrayList<Observer> obs = new ArrayList<>();
5
6       public void attach(Observer o) {
7           obs.add(o);
8
9       }
10
11      public void detach(Observer o) {
12          obs.remove(o);
13      }
14
15      public void notifyObservers() {
16          for (Observer o : obs) {
17              o.update(this);
18          }
19      }
20
21      public abstract void setValue(double temperature);
22
23      public abstract double getValue();
24  }
```

Classe Observer

```
public interface Observer{
    public void update(Observable observable);
}
```

Classe TemperatureModel

```
public class TemperatureModel extends Observable {
    private double temperature;
    private ValueStrategy vs;

    public TemperatureModel() {
        this.temperature = 273.15;
    }

    @Override
    public double getValue() {
        return vs.fromKelvin(temperature);
    }

    @Override
    public void setValue(double temperature) {
        this.temperature = vs.toKelvin(temperature);
        notifyObservers();
    }

    public void setStrategy(ValueStrategy vs) {
        MinDecorator minD = new MinDecorator(vs, 263.15);
        MaxDecorator maxD = new MaxDecorator(minD, 343.15);
        this.vs = maxD;
        notifyObservers();
    }
}
```

Classe View

```
import javax.swing.*;
import java.awt.event.*;

public class View implements Observer {
    private TemperatureModel tm;
    private JTextField tfTemp;

```

```
8       public View(TemperatureModel tm, JTextField tfTemp,
            JButton bInc, JButton bDec) {
9           this.tm = tm;
10          this.tfTemp = tfTemp;
11          this.tfTemp.setText(String.format("%.1f", tm.getValue
                ()));
12
13          bInc.addActionListener(new ActionListener() {
14              @Override
15              public void actionPerformed(ActionEvent e) {
16                  increaseTemperature();
17              }
18          });
19          bDec.addActionListener(new ActionListener() {
20              @Override
21              public void actionPerformed(ActionEvent e) {
22                  decreaseTemperature();
23              }
24          });
25      }
26
27      @Override
28      public void update(Observable obs) {
29          tfTemp.setText(String.format("%.1f", obs.getValue()));
30      }
31
32      public void increaseTemperature() {
33          double currentTemperature = tm.getValue();
34          tm.setValue(currentTemperature + 1);
35      }
36
37      public void decreaseTemperature() {
38          double currentTemperature = tm.getValue();
39          tm.setValue(currentTemperature - 1);
40      }
41
42  }
```

Classe Main

```
1   import javax.swing.*;
2   import java.awt.event.*;
```

```
3  import java.awt.*;
4
5  public class Main {
6      public static void main(String args[]) {
7          JFrame f = new JFrame("Thermostat");
8          f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9          JTextField tfTemp = new JTextField();
10         JButton bInc = new JButton("Increase");
11         JButton bDec = new JButton("Decrease");
12         JButton bK = new JButton("Kelvin");
13         JButton bC = new JButton("Celsius");
14         JButton bF = new JButton("Farenheit");
15
16         tfTemp.setHorizontalAlignment(JTextField.CENTER);
17         tfTemp.setEditable(false);
18         tfTemp.setBounds(175, 100, 150, 20);
19         bK.setBounds(50, 200, 100, 50);
20         bC.setBounds(200, 200, 100, 50);
21         bF.setBounds(350, 200, 100, 50);
22         bInc.setBounds(100, 300, 100, 50);
23         bDec.setBounds(300, 300, 100, 50);
24         bInc.setBackground(Color.decode("#98fb98"));
25         bInc.setForeground(Color.BLACK);
26         tfTemp.setBackground(Color.WHITE);
27         bDec.setBackground(Color.decode("#ff6961"));
28         bDec.setForeground(Color.BLACK);
29
30         f.add(tfTemp);
31         f.add(bK);
32         f.add(bC);
33         f.add(bF);
34         f.add(bInc);
35         f.add(bDec);
36
37         f.setSize(500, 500);
38         f.setLayout(null);
39         f.setVisible(true);
40
41         TemperatureModel tm = new TemperatureModel();
42         tm.setStrategy(new CelsiusStrategy());
43         View v = new View(tm, tfTemp, bInc, bDec);
```

```
44          tm.attach(v);
45
46          bK.addActionListener(new ActionListener() {
47              @Override
48              public void actionPerformed(ActionEvent e) {
49                  tm.setStrategy(new KelvinStrategy());
50              }
51          });
52          bC.addActionListener(new ActionListener() {
53              @Override
54              public void actionPerformed(ActionEvent e) {
55                  tm.setStrategy(new CelsiusStrategy());
56              }
57          });
58          bF.addActionListener(new ActionListener() {
59              @Override
60              public void actionPerformed(ActionEvent e) {
61                  tm.setStrategy(new FarenheitStrategy());
62              }
63          });
64
65      }
66  }
```

4 Testing

Una volta avviato il programma, si apre il display del termostato inizializzato alla temperatura di 0 gradi Celsius.

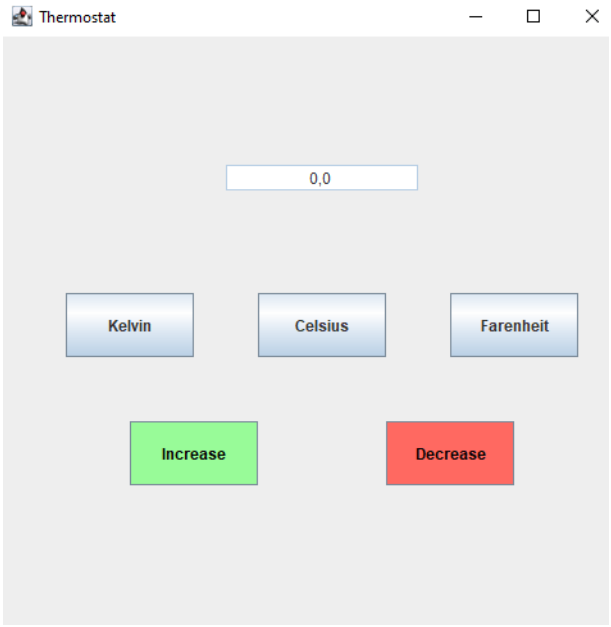

Figura 7: Output all' inizializzazione

Cliccando 4 volte il bottone 'Increase' ci si aspetterebbe che la temperatura aumenti di 4 gradi nella scala selezionata, in questo caso Celsius. Difatti, il termostato segna 4 gradi Celsius in più rispetto alla temperatura di partenza.

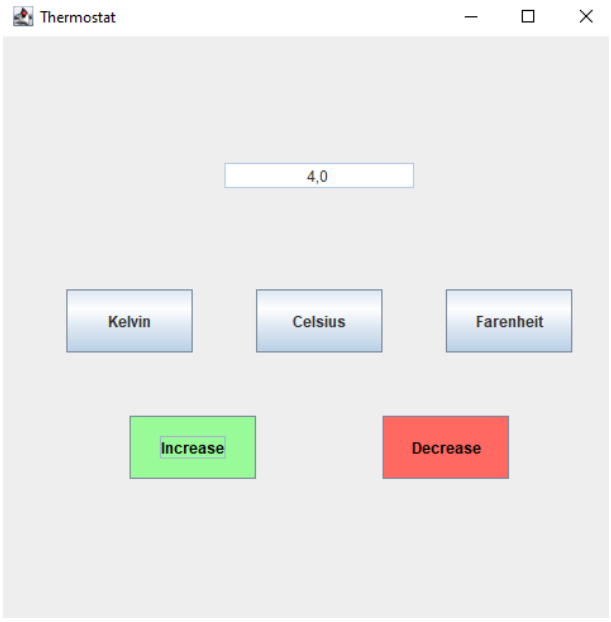

Figura 8: Risultato dopo aver incrementato 4 volte

Cliccando sul bottone 'Farenheit' ci si aspetterebbe che la temperatura venga convertita in Farenheit. Infatti applicando la formula di conversione 4 gradi Celsius vengono convertiti in 39,2 gradi Farennheit.

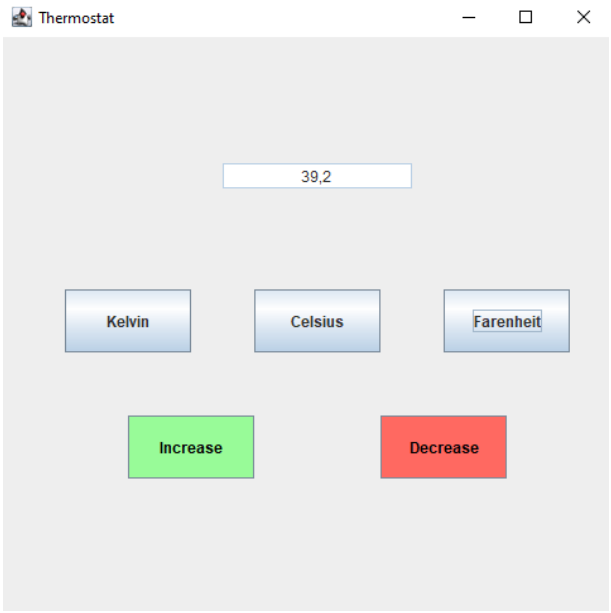

Figura 9: Conversione in gradi Farenheit

Cliccando sul bottone 'Decrease' due volte,ci si aspetterebbe che la temperatura diminuisca di 2 gradi nella scala selezionata, in questo caso Farenheit. Difatti, il termostato segna 2 gradi Farenheit in meno rispetto alla temperatura precedente.

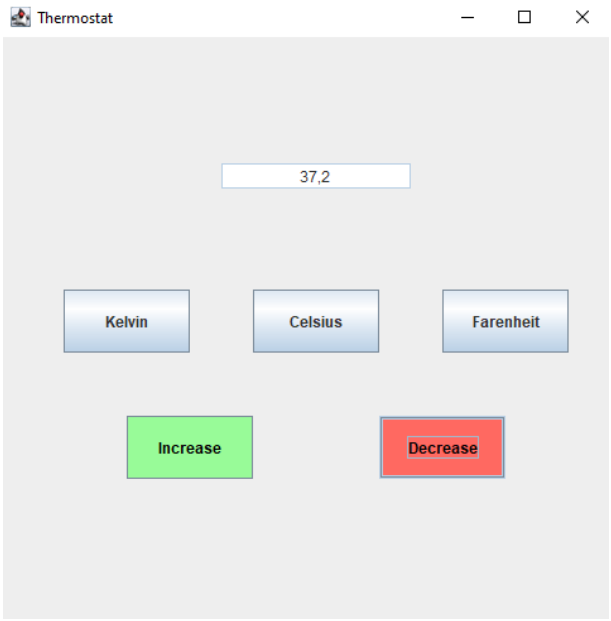

Figura 10: Risultato dopo aver decrementato 2 volte

Cliccando sul bottone 'Kelvin' ci si aspetterebbe che la temperatura venga convertita in Kelvin. Infatti applicando la formula di conversione 37,2 gradi Farenheit vengono convertiti in 276,0 gradi Kelvin.

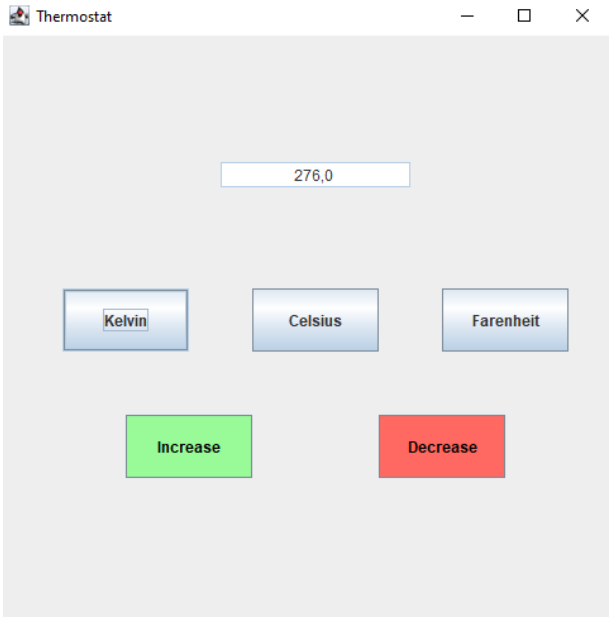

Figura 11: Conversione in gradi Kelvin

Se si raggiunge la temperatura di 70 gradi Celsius (o le rispettive conversioni nelle altre scale d riferimento) e si prova cliccare nuovamente sul bottone 'Increase', la temperatura non viene incrementata poiché è stato raggiunto il limite massimo.

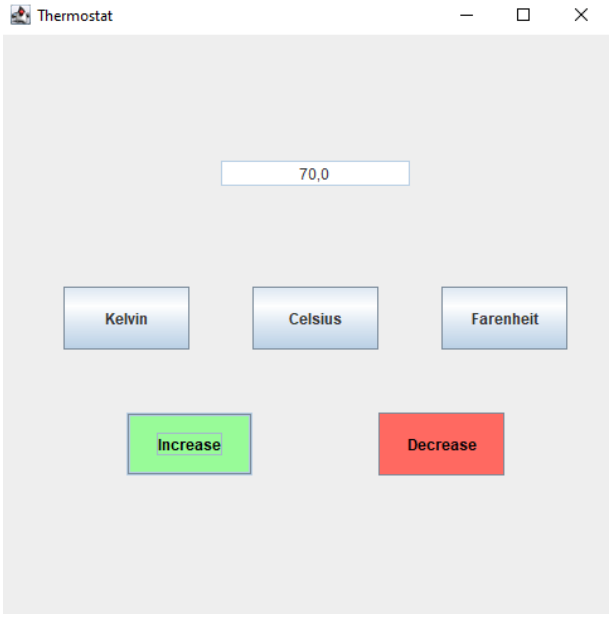

Figura 12: Limite massimo di temperatura dopo il tentato incremento

Se si raggiunge la temperatura di -10 gradi Celsius (o le rispettive conversioni nelle altre scale d riferimento) e si prova cliccare nuovamente sul bottone 'Decrease', la temperatura non viene decrementata poiché è stato raggiunto il limite minimo.

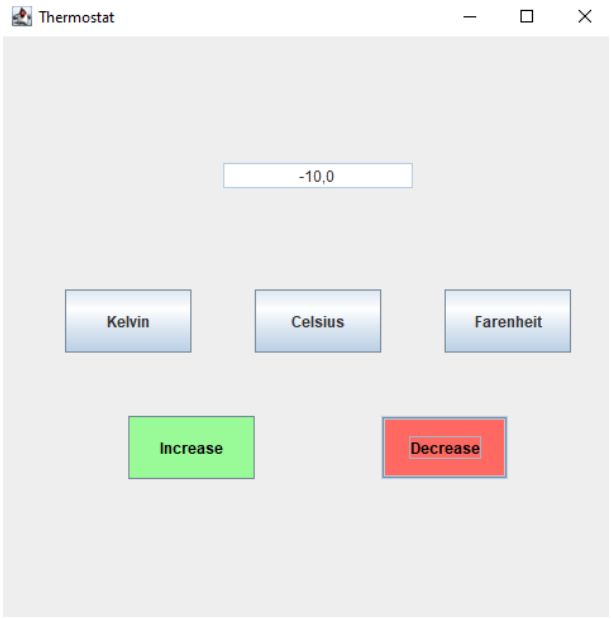

Figura 13: Limite minimo di temperatura dopo il tentato decremento