Documentación práctica Patrones de Diseño

Leopoldo A. Estévez Agra
David Rodríguez Bacelar

22 de diciembre de 2020

Ejercicio 1 : Control de la temperatura

- Principios SOLID utilizados

- **Principio de inversión de la dependencia**
  Este principio se puede observar en el método *screenInfo()* de la clase "Thermostat".

```
private ThermostatMode mode; //Interfaz
/*(...)*/
public String screenInfo() {
    String s = mode.getInfo(this);
    System.out.println(s);
    return s;
}
```

  Vemos como podemos ralizar la llamada al método, a pesar de no saber exactamente como está implementado.

- **Principio de responsabilidad única**

  Este es el principio fundamental del problema. Basándonos en él, buscamos quitar responsabilidades a la clase principal "Thermostat", delegando la lógica de los cambios de estado a los propios estados. Por ejemplo:

```
public void changeToOff() {
    String s = mode.changeToOff(this);
    System.out.println(s);
    log.addToLog(s);
}
```

- **Principio abierto-cerrado**

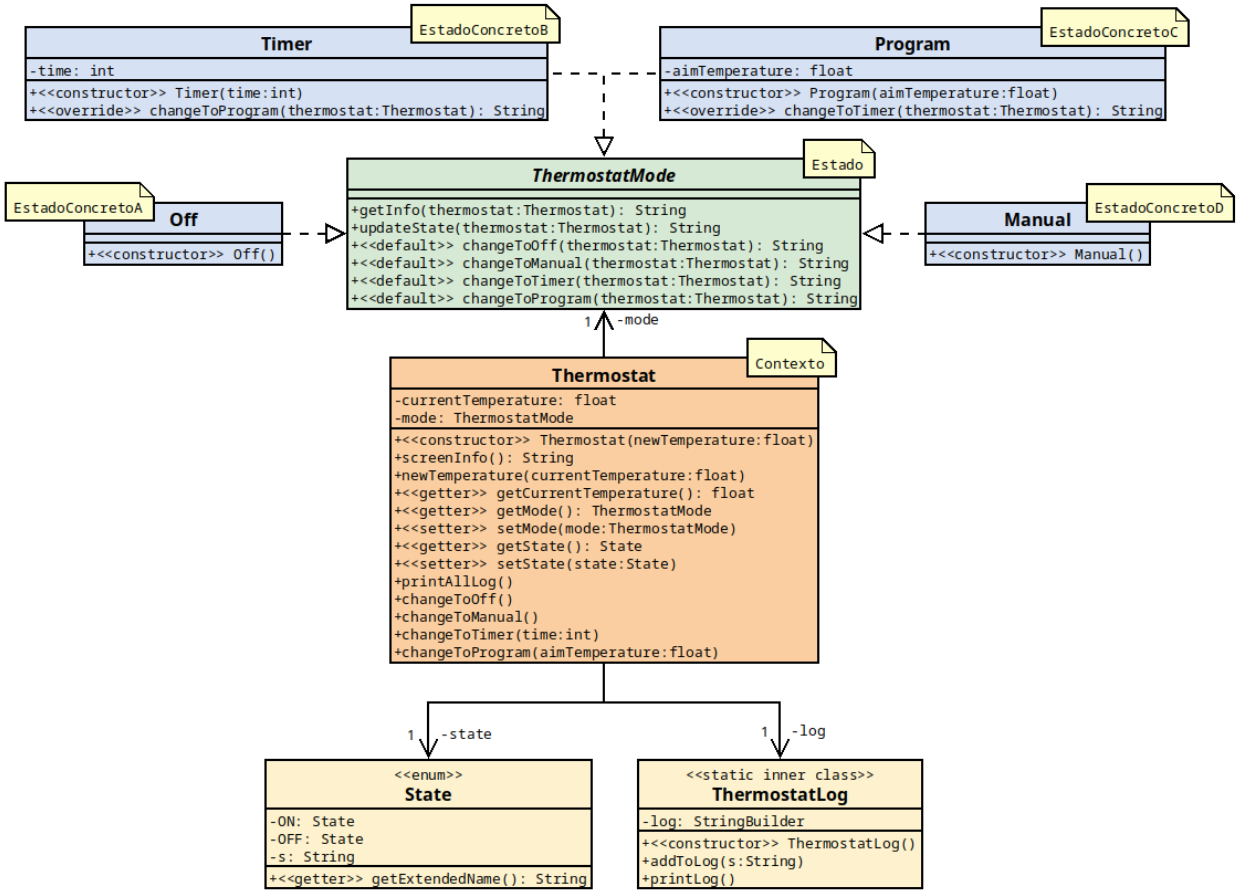
  Derivado del principio de inversión de la dependencia anterior, la utilización de la interfaz "ThermostatMode" nos permite abrir nuestro código para extenderlo (crear nuevos estados por ejemplo) pero cerrarlo ante la modificación (tienen que cumplirse las cabeceras declaradas en la interfaz)

- Patrón de diseño utilizado

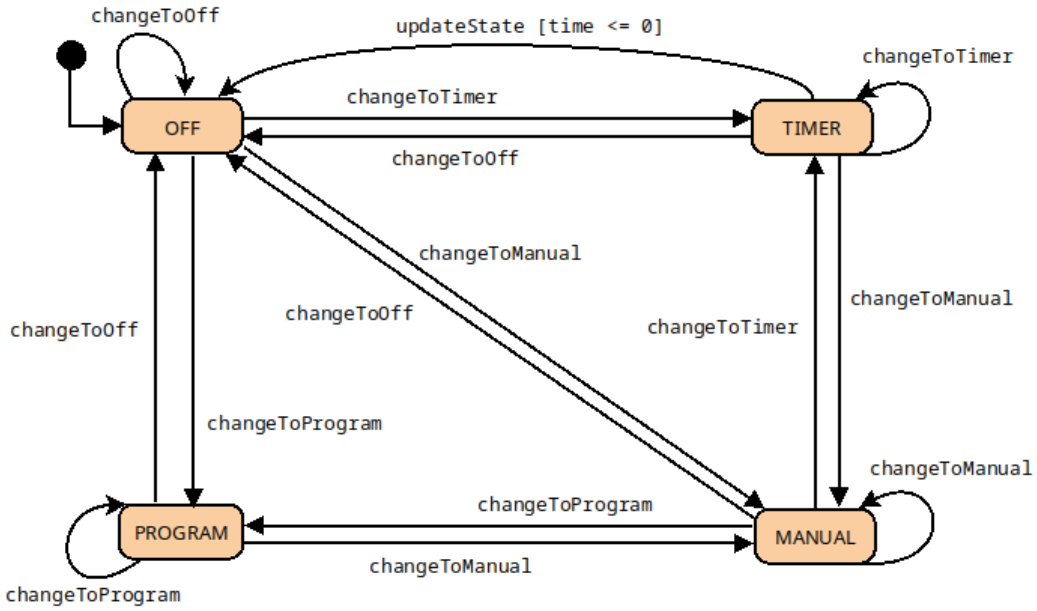
- **Patrón Estado:**

  El problema nos presenta un termostato el cual tiene un estado interno que puede ir cambiando; por lo tanto -y a su vez para cumplir los principios SOLID- decidimos utilizar este patrón, evitando crear una clase "Thermostat" con demasiadas responsbilidades y muy cerrada a extensión.

- **Diagrama de Clases:**



- **Diagrama de Estados:**



Nota: Decidimos no especificar las acciones que se realiza en cada cambio de estado sobre la calefacción debido a que el diagrama resultaba demasaido intrincado y menos entendible.

Ejercicio 2 : Gestión de equipos de trabajo

- Principios SOLID utilizados:

- **Principio de inversión de la dependencia**

  Este principio, de forma similar al ejercicio anterior, se observa en las clases "Team" y "Project", concretamente en la declaración de los atributos "workParts" y "projectTeams" respectivamente.

```
private List<WorkPart> workParts = new ArrayList<>();
/*(...)*/
private List<Team> projectTeams = new ArrayList<>();
```

  Esto nos permite el cambio en cualquier momento de la implementeción de nuestra lista no dependiendo así de una concreción de la misma.

- **Principio de responsabilidad única**

  Para cumplir este principio y aumentar la cohesión del código, se buscó hacer las clases lo más simples y entendibles posible delegando resposabilidades entre ellas.
  Esto se puede ver por ejemplo en los métodos *getHours()* o *getCost()* de la clase "Team", los cuales delegan continuamente a los trabajadores la responsabilidad de devolver sus horas totales trabajadas y el dinero generado.

```
/*Team class*/
public float getCost() {
    float totalCost = 0;
    for ( WorkPart w : workParts )
        totalCost += w.getCost();
    return totalCost;
}
/*(...)*/
/*Worker Class*/
public float getCost() {
    return workedHours * hourCost;
}
```

- **Principio abierto-cerrado**

  Este principo se ve de forma muy clara a través de la utilización de la interfaz "WorkPart".
  Si, por ejemplo, imaginemos que queremos añadir otro tipo de trabajador y la forma de calcular su sueldo es distinta, al heredar de dicha clase "WorkPart", tendrá que implementar obligatoriamente todos los métodos (de la manera que él quiera), pero no será necesario cambiar el resto de clases.

  Vemos como, tanto si queremos añadir nuevas clases como si queremos añadir nuevas funcionalidades, la interfaz "WorkPart" nos facilita todo ello.

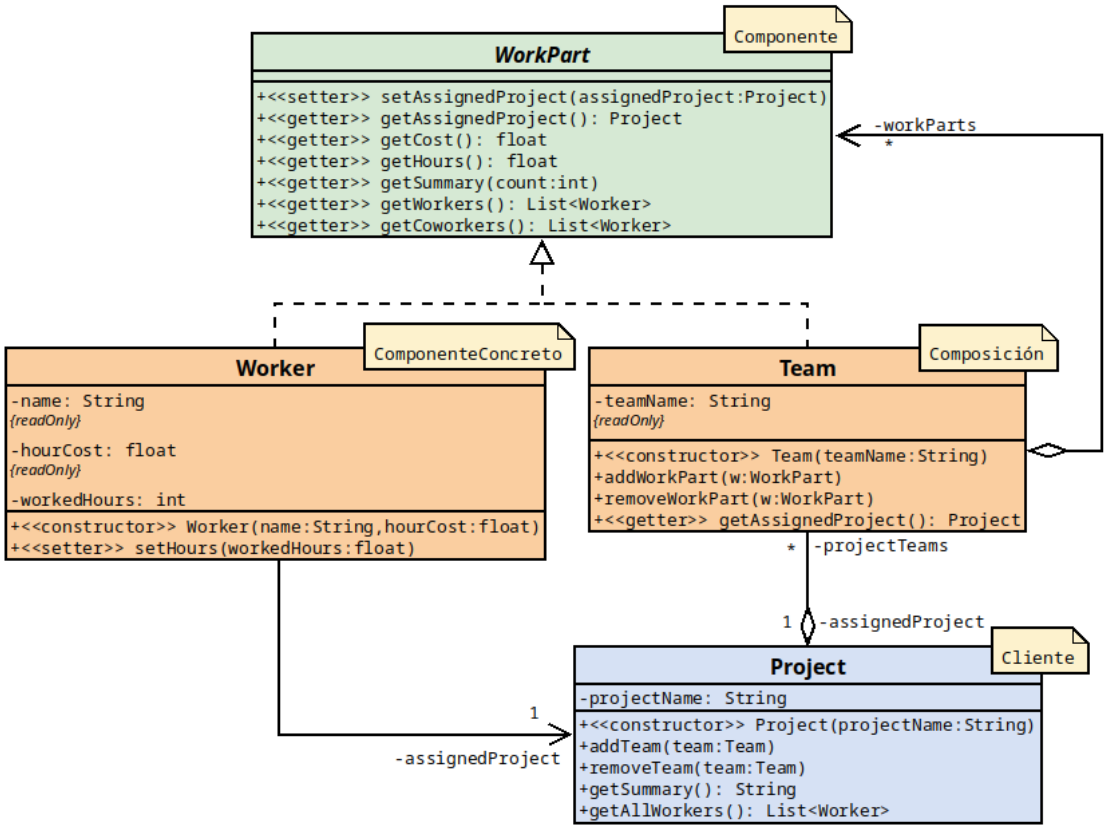
```
public interface WorkPart {
    //Obliga a todas las clases implementadoras a
       cumplir estas cabeceras
    void setAssignedProject(Project project);
    float getHours();
    float getCost();
    String getSummary(int count);
    /*(...)*/
}
```

- Patrón de diseño utilizado:

- **Patrón Composición:**

  Dado que el problema que propone el ejercicio tiene una estructura de árbol, en la que un equipo puede tener varios trabajadores asociados así como a su vez varios subequipos con la misma estructura, decidimos utilizar el patrón composición.
  Implementamos así en nuestro programa una intefaz que representa tanto a los equipos como a los trabajadores, declarando métodos utilizados por ambos.

- **Diagrama de Clases:**



- **Diagrama de Secuencia:**

  Tomamos como ejemplo para el diagrama un proyecto *P1* con un único equipo *T1* que tiene a su vez dos trabajadores *w1* y *w2*.