UNIVERSIDADE DA CORUÑA

Padín Torrente, Héctor
Teixeira Crespo, Jorge

Informe sobre la Práctica de Diseño (2020-2021)

UNIVERSIDADE DA CORUÑA

Padín Torrente, Héctor
Teixeira Crespo, Jorge

Ejercicio 1: Problema del Termostato

Principios de Diseño

Patrones SOLID utilizados para el desarrollo de la solución:

- **Principio de Sustitución de Liskov**: En nuestra solución, este principio de diseño puede observarse cumpliendo el principio de subcontratación de *Estado* y *EstadoConcreto*. La interfaz *Estado* define una serie de métodos que deben implementar los estados concretos con sus restricciones. Por ejemplo para el método *cambioEstado*, el estado concreto *Timer* lo redefine cumpliendo las restricciones iniciales del método y añadiendo una más que es que no se pueda cambiar directamente al modo *Program*. Por esto, cualquier objeto *Estado* podrá ser sustituído por un objeto *Timer*. Idem para el resto de estados concretos.

- **Principio de Inversión de la dependencia**: Este principio se ve reflejado mediante el atributo estado de la clase *Termostato*, este atributo es una interfaz *Estado* que define los métodos que deberán implementar todas las clases que extiendan de esta interfaz. Por tanto estamos evitando depender de un estado en concreto o tener que disponer de una instancia de todos ellos en el termostato. De esta manera cuando se hace un cambio de estado, este seguirá llamando al mismo método sin enterarse de si el estado ha cambiado o en qué modo se encuentra (*Manual*, *Off*, etc).

- **Principio abierto-cerrado**: Se utiliza por ejemplo en el método *nuevaTemperatura* de la interfaz *Estado* tenemos estados que actúan de diferente manera ante una actualización de la temperatura que se hace cada 5 minutos, el estado *Program* activa o desactiva la calefacción según la temperatura de consigna, el *Timer* se desactivará pasado un tiempo, el *Manual* solo actualiza el registro, etc. Por lo tanto desde la clase termostato, se utiliza la ligadura dinámica para llamar al método *nuevaTemperatura* del estado dinámico que contiene en ese momento la clase *Termostato*. Añadir un nuevo estado simplemente significaría añadir una nueva clase que extienda la interfaz *Estado* y que redefina sus métodos.

También se usaron como referencias otros principios de diseño que son inherentes a los anteriormente citados, como el principio de "*encapsula lo que varía*" (*Estado* y *Estado Concreto*), el principio de "*bajo acoplamiento*", "*Tell, don´t ask*" (por ejemplo el termostato nunca le pregunta al objeto estado qué estado es, simplemente le ordena hacer cosas) y otros principios más genéricos como el "*DRY*" o "KISS".

UNIVERSIDADE DA CORUÑA

Padín Torrente, Héctor
Teixeira Crespo, Jorge

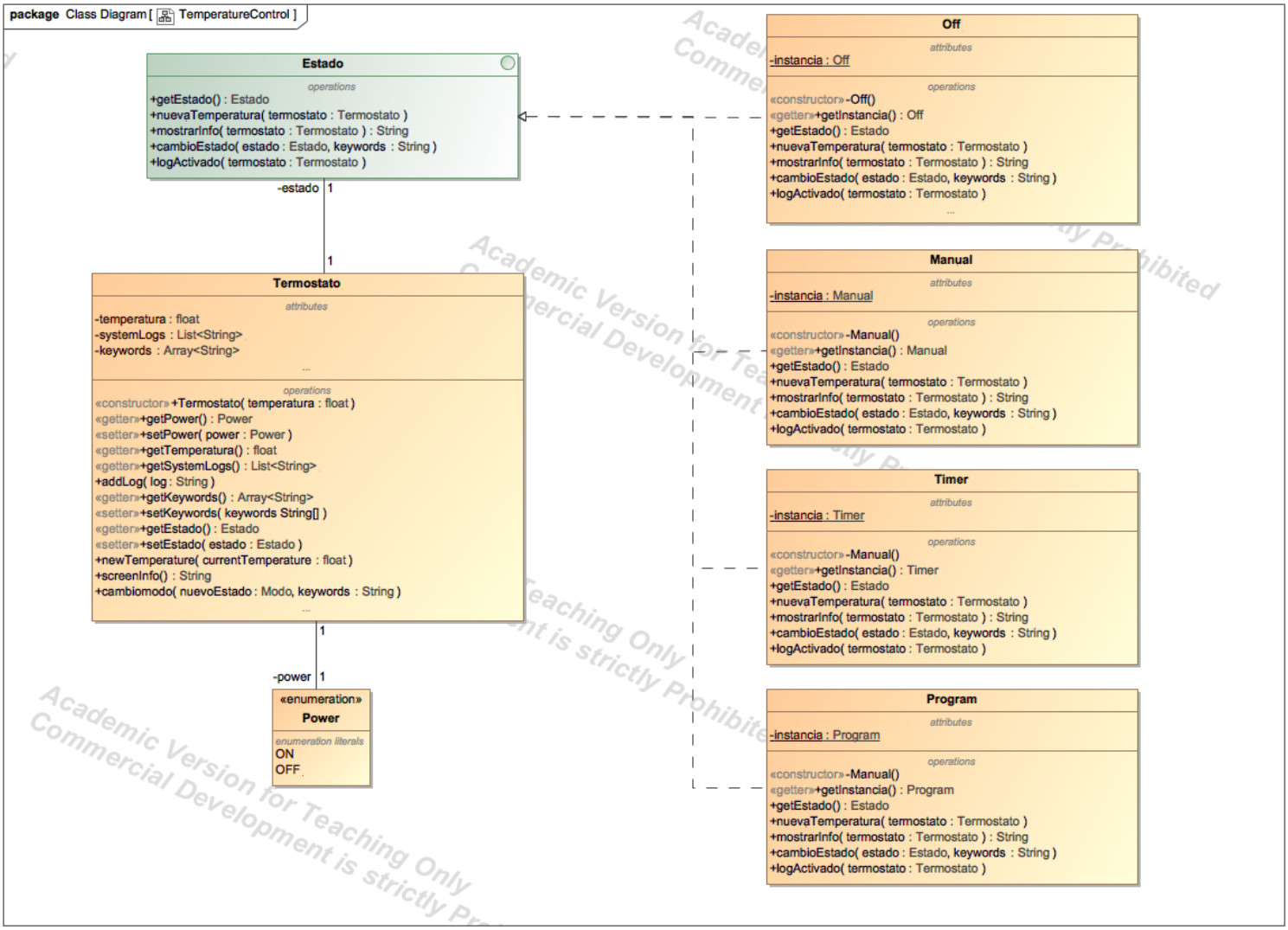
Patrones de Diseño

Para este ejercicio decidimos implementar el *Patrón Estado* y el *Patrón Instancia Única*.

Cada modo de funcionamiento del termostato representa un estado dentro de nuestro programa, puesto que estos actúan de diferente forma ante cambios de temperatura o mismo entre cambios del modo de funcionamiento, crean logs de diferente forma, etc. Por lo tanto nos pareció adecuado separar el comportamiento de cada "estado" o "modo" en clases diferenciadas que cumplen un contrato con su clase superior para poder cambiar dinámicamente entre estos sin preocuparse.

Estos estados pueden compartirse, por ejemplo, si en la facultad tenemos ocho termostatos, ambos estarían usando los mismos estados puesto que se representan un punto de acceso global para todos estos.

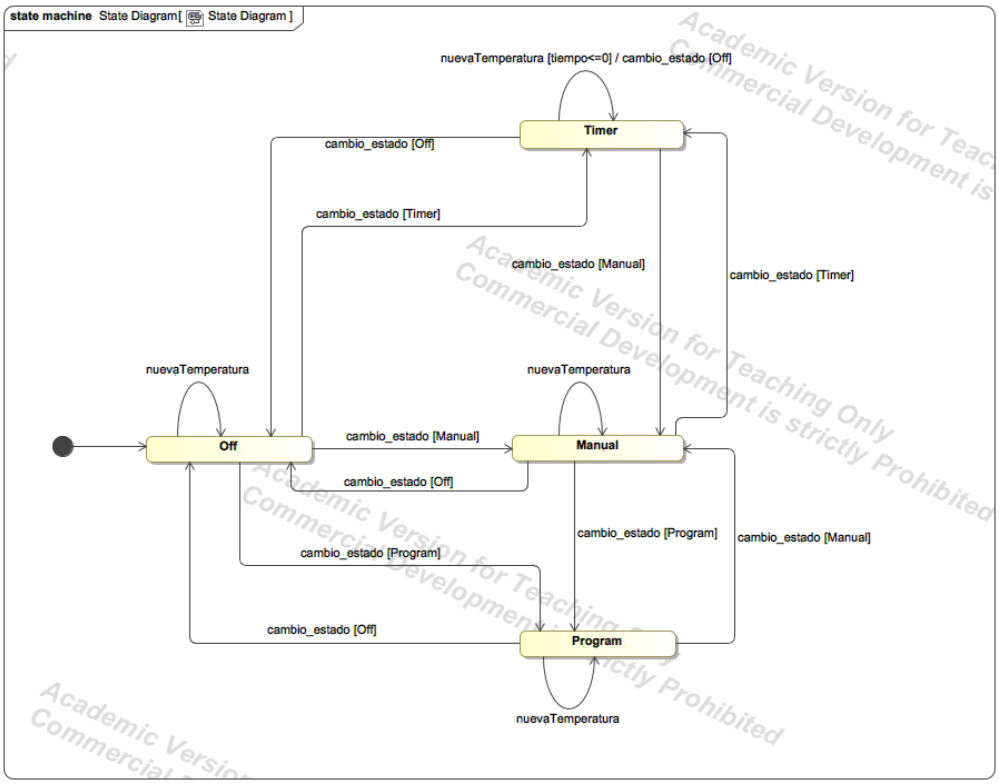
Diagrama de clases



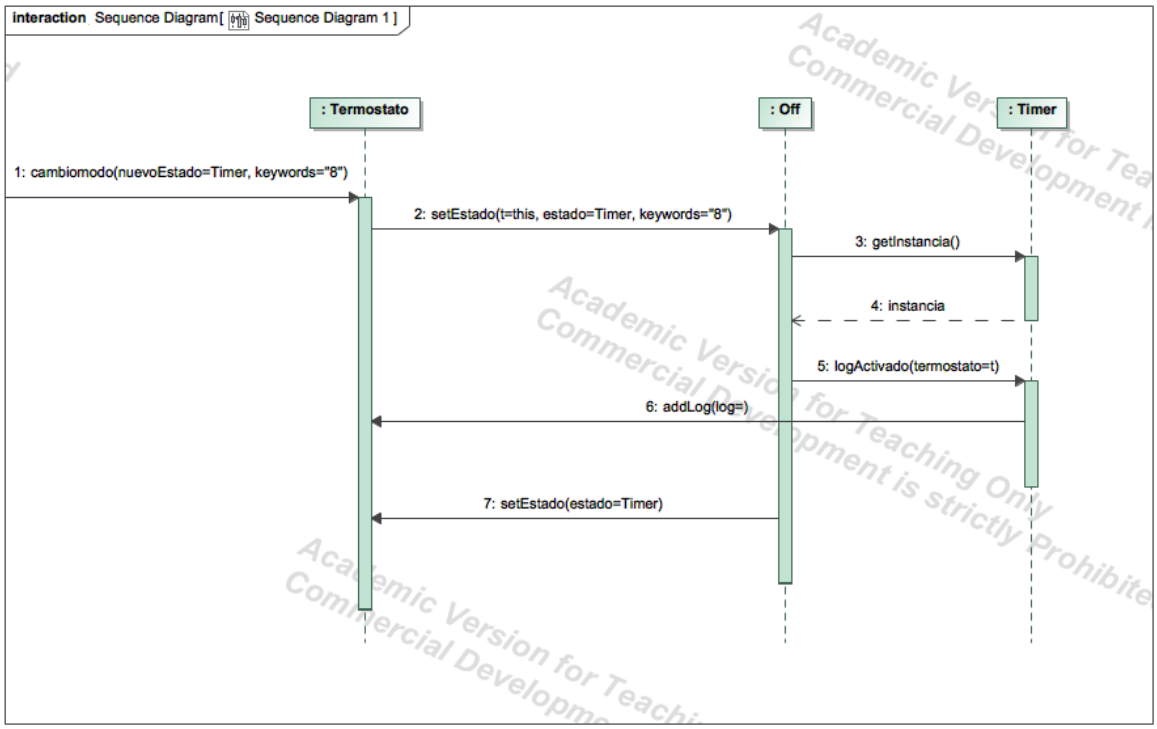
Padín Torrente, Héctor
Teixeira Crespo, Jorge

Diagramas Dinámicos

A continuación se muestra el diagrama de estados correspondiente al diseño de la solución. Con esto buscamos representar la capacidad de cambio de los diferentes estados entre sí.

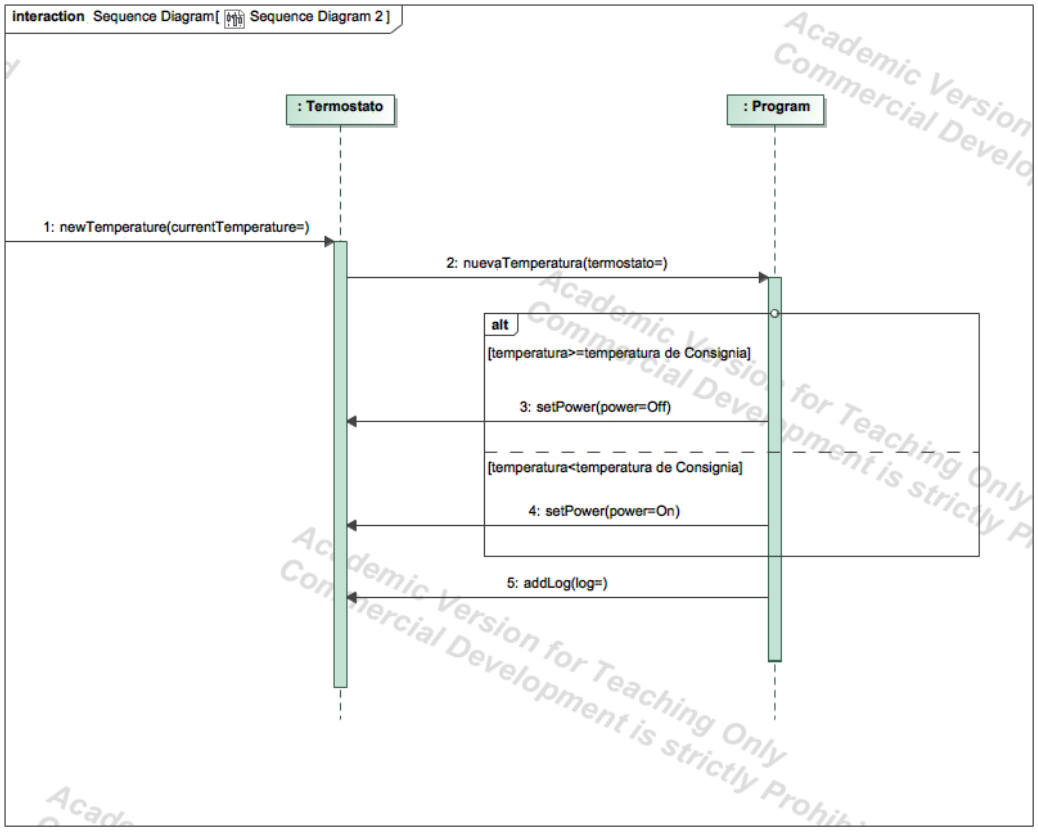


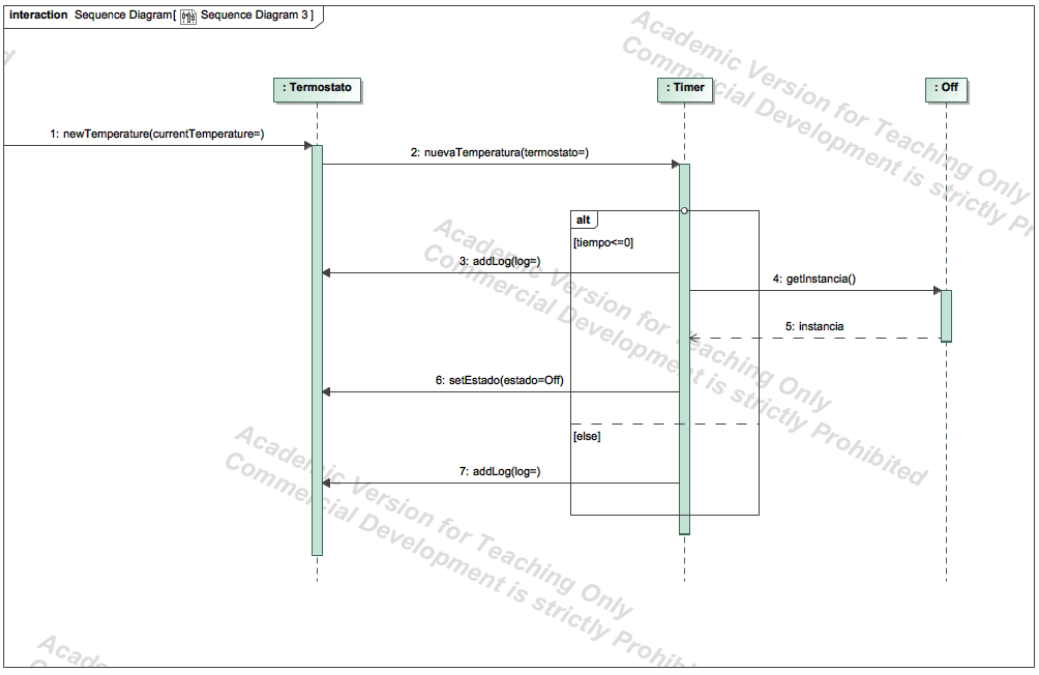
El siguiente diagrama dinámico representa un cambio de estado simple de modo *Off* a modo *Timer*.



Padín Torrente, Héctor
Teixeira Crespo, Jorge

A continuación mostramos el funcionamiento de actualizar la temperatura con el estado *Program* y *Timer*, puesto que el primero puede apagar o encender la calefacción según la temperatura del ambiente suba o baje con respecto a una temperatura de consigna introducida anteriormente y el segundo la activa durante un determinado tiempo.





Padín Torrente, Héctor
Teixeira Crespo, Jorge

Ejercicio 2: Gestión de equipos de trabajo

Principios de diseño

YAGNI (You aren't gonna' need it)

Como ejemplo del uso, revisando el código, hemos encontrado una función no utilizada y además que permitía modificar el *arraylist* de *teams* de manera peligrosa sin ninguna comprobación previa, con lo cual, como no lo íbamos a necesitar hemos decidido borrarla. El borrado está en el commit 3153194 .

DRY (Don't repeat yourself)

Para no repetir varias veces el mismo trozo de código que comprueba la pertenencia de un elemento de la plantilla a un proyecto, después de tenerlo repartido se refactorizó en un único método dentro de la clase Project que, dado un *WorkforceElement* devuelve true si puede trabajar en ese proyecto y false en caso contrario.

```
/**
* Checks if the given element is an allowed element * in the project.
*
* @param we the element to check
* @return <code>true</code> if is allowed, or * <code>false</code> otherwise
*/
public boolean hasWorkforceElement(WorkforceElement we){
        ...
}
```

KISS (Keep it simple, stupid)

Este principio fue básico para pensar el código del iterador creado. Después de varias vueltas intentando codificarlo, de manera no satisfactoria, se realizó un cambio en el planteamiento que finalmente consiguió mantener el código simple. Estábamos intentando hacer que el iterador, cada vez que se llamaba a su método *next()* realizara el cálculo de cuál sería el siguiente elemento del árbol. De esta manera teníamos que añadir varios atributos, llevar la cuenta de en qué elemento íbamos, y programar un complejo algoritmo. Al final, y de forma mucho más sencilla, al llamar al constructor del iterador se crea una cola con todos los elementos por los que se va a iterar, y con una simple llamada se consigue el siguiente. Véase:

Creación de la cola:

```
/**
*  Adds the given element to the queue, determines with
*  element has to be next and calls recursively with the
*  element.
*
*  @param we the element to be added to the queue
*/
```

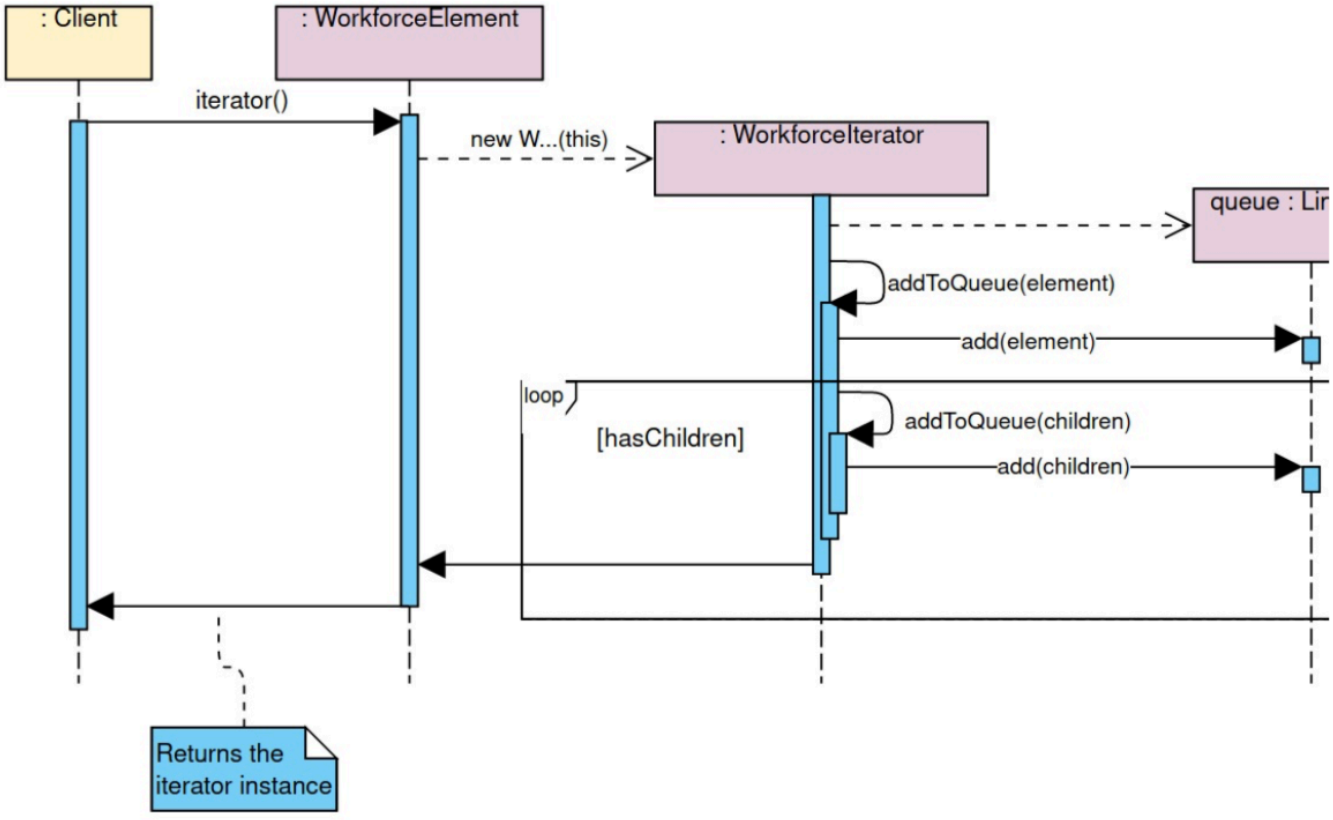
Padín Torrente, Héctor
Teixeira Crespo, Jorge

```
private void addToQueue(WorkforceElement we){
        queue.add(we);
        int i=0;
        while(we.getChild(i)!=null){
                addToQueue(we.getChild(i++));
        }
}
```

Llamada a *next()* simplificada:

```
@Override
public WorkforceElement next() { return queue.pop(); }
```

Diagrama creación iterador:



Padín Torrente, Héctor
Teixeira Crespo, Jorge

SOLID

Single Responsibility

Las clases cumplen una única responsabilidad.

- *Worker*: almacenar el coste del trabajo de un trabajador en un proyecto.
- *Team*: agrupar elementos.
- *Project*: almacenar el coste derivado de un proyecto.

Open/Close principle

Si quisiéramos cambiar el código para calcular el precio hora, y añadir 10 € de comisión por cada trabajador en un proyecto, el código estaría abierto a modificaciones pero cerrado para poder usarlo. Véase:

Código inicial:

```
@Override
public float getCost(Project project) {
        return getHours(project) * hourlyRate;
}
```

Código modificado:

```
@Override
public float getCost(Project project) {
        return getHours(project) * hourlyRate + 10f;
}
```

Y no tendríamos que realizar ninguna otra modificación en ninguna de las clases que lo utilizan, tan sol cambia el comportamiento del *Worker*.

Liskov substitution principle

La clase *WorkforceElement* cumple este principio con los elementos *Team* y *Worker*. Se pueden intercambiar sin suponer un cambio en la conducta del software.

Dependency inversion principle

Se cumple al utilizar la clase *List* o *Iterator*. Siempre utilizando la clase abstracta. También se procura utilizar siempre con los *WorkforceElement*. A excepción de que *Project* se relaciona con *Team* por seguridad y fidelidad al enunciado.

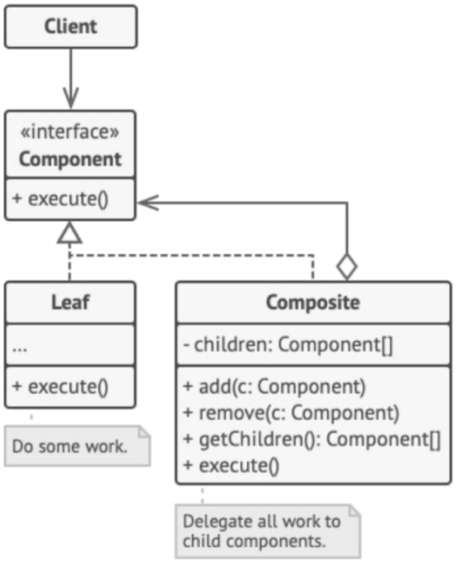
Padín Torrente, Héctor
Teixeira Crespo, Jorge

Patrones de diseño

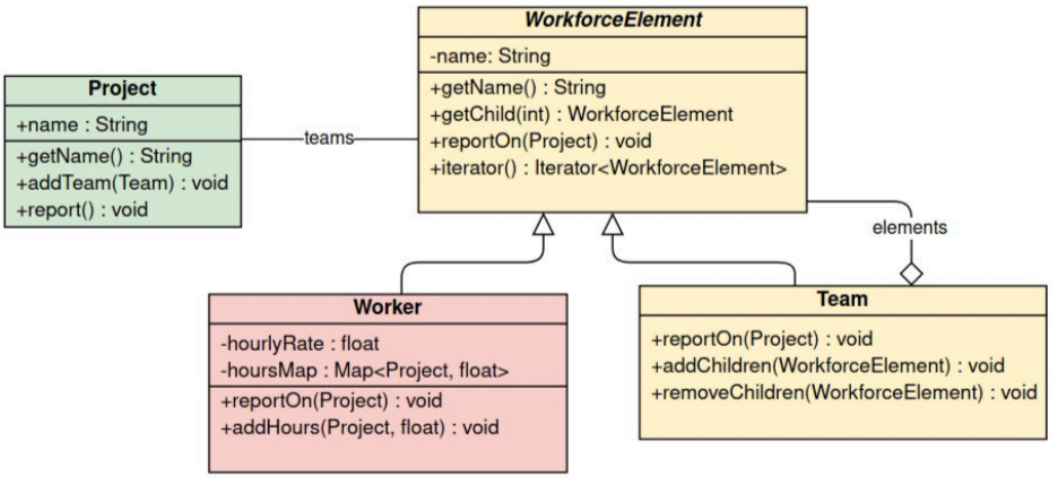
Patrón Composición

El patrón de diseño escogido para el ejercicio es Composición. El motivo por el cual no cabe duda es que la estructura que se pretende representar es similar a la de un árbol. En la que los nodos son equipos y trabajadores, más concretamente siendo los equipos los nodos y los trabajadores las hojas.

Modelo Composición



Diseño implementado al problema



Padín Torrente, Héctor
Teixeira Crespo, Jorge

Explicaciones

- El cliente es el proyecto, ya que es el que utiliza a los elementos del personal de la empresa, representados por instancias de *WorkforceElement* en forma de árbol.

- Para representar a la plantilla de la empresa se utiliza la clase abstracta *WorkforceElement*, que conforma cada uno de los nodos del árbol. Los cuales pueden ser de dos tipos:
    - Contenedores (Team): Contienen más elementos de la plantilla y delega la operación principal.
    - Hojas (*Worker*): Contienen la información necesaria para la operación principal *report()* del cliente.
        - hourlyRate: Contiene el salario/hora del trabajador.
        - hoursMap: Mapa clave-valor que contiene las horas (valor) trabajadas en cada proyecto (clave) por el trabajador.

- La operación común es *reportOn(Project)* que lo que hace es recorrer todos los elementos del árbol empezando por él elemento desde el que se llama, y realizar un informe/sumatorio de la totalidad del coste del proyecto. Para realizar esta operación se utilizan más métodos no reflejados en el diagrama anterior, omitidos para facilitar la comprensión, ya que estos son exclusivamente llamados desde el método *report* y sirven para hacer los cálculos. Estos métodos funcionan de la misma forma que *report*, en las hojas realizan cálculos y los contenedores delegan siempre el trabajo a todos sus hijos.
    - getHours(Project): Devuelve el total de horas invertidas por un trabajador en un proyecto dado, o en su defecto, si se llamase desde un equipo, la suma todas las horas invertidas por los miembros del mismo.
    - getCost(Project): Idem, multiplicado por el salario/hora de los trabajadores.

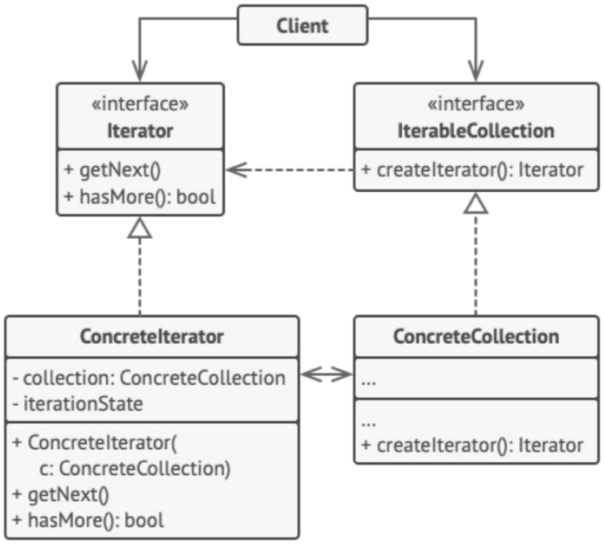
- **Anotación**: En el diagrama, aparece representado el cliente (Project) relacionado con *WorkforceElement*. En la práctica y por seguir el enunciado, esta relación es realmente con la clase *Team* ("Un proyecto se organiza en una jerarquía de equipos de trabajo."). Esto se realiza para asegurar que no hay trabajadores sueltos en los proyectos. Aun así, si así se desease, se podría cambiar la firma de la lista por *List*<*WorkforceElement*> en todas sus apariciones y se solventaría sin ninguna otra modificación.

Padín Torrente, Héctor
Teixeira Crespo, Jorge

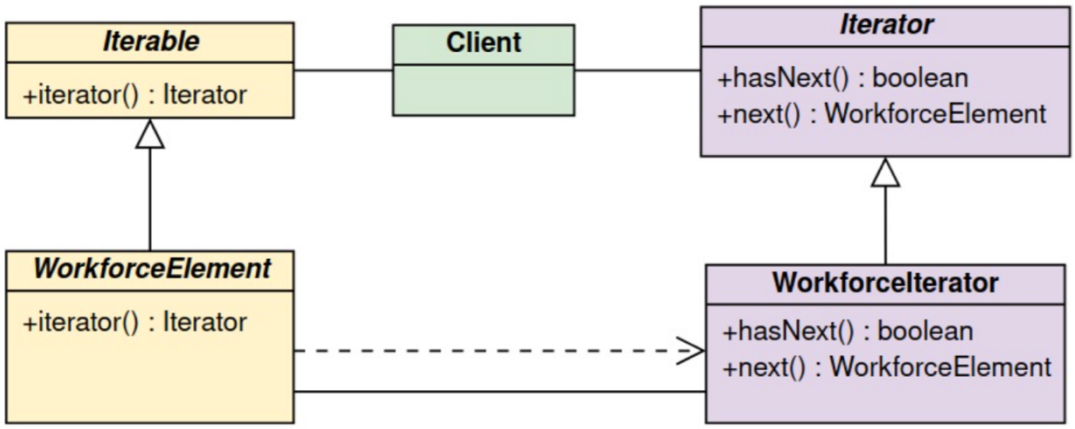
Patrón Iterador

El patrón iterador nos sirve en el software para recorrer la estructura del árbol de forma secuencial y sin tener en cuenta el árbol. Está implementado de forma que recorre el árbol desde el elemento desde el que se llama y de forma descendente, utilizando el algoritmo *preorder*.

Modelo Iterador



Implementación concreta

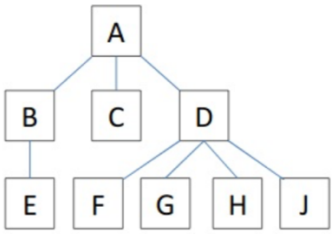


En este caso, al estar utilizando Java, solo tenemos que realizar codificar la clase *ConcreteIterator*, e implementar la interfaz *Iterable* en la colección (en nuestro caso el árbol empezado en *WorkforceElement*).

Padín Torrente, Héctor
Teixeira Crespo, Jorge

Algoritmo de recorrido preorder

El algoritmo utilizado para recorrer el árbol es *preorder*. Devuelve primero el elemento padre, de cada hijo siempre recorrido todos sus hijos antes de pasar a su hermano. Más sencillo con ejemplos:



- *Preorder* desde A: ABECDFGHJ
- *Preorder* desde D: DFGHJ
- *Preorder* desde C: C