Informe de práctica de Diseño (2020-2021) Grupo 2.1 2020-21

Kevin Millán Canchapoma
Brais Cotelo Vilas

1. Control de temperatura

En este ejercicio usaremos principalmente 2 patrones de Diseño y los principios SOLID para desarrollar un código apropiado.

El primer patrón de Diseño que voy a tratar es el patrón de Estado, que permite al objeto termostato modificar su conducta en función su estado interno (del modo). Creamos una clase abstracta que encapsula el comportamiento de los estados del contexto esta clase será ModoTermostato que nos dará las cabeceras de distintas funciones necesarias para el ejercicio pero que se comportan diferente en función del contexto. También creamos estados concretos que serán clases que implementan el comportamiento asociado con un estado del contexto, estos estados concretos los veremos reflejados en las distintas clases que heredan de ModoTermostato, estas clases son: Off, Manual, Timer y Program. Además, creamos otro objeto que será el contexto, que mantiene una instancia de un estado concreto y delega en ella el funcionamiento dependiente del estado, nuestra clase Termostato realiza este rol, en él las funciones screenInfo y calentar se delega el funcionamiento en las clases estados concretos. Decidimos escoger este patrón porque permite a un usuario insertar más funciones y más estados aprovechando la ligadura dinámica y sin alterar el funcionamiento global del termostato.

En las implementaciones de los estados concretos y la clase termostato hacemos uso de otro patrón de Diseño, el patrón instancia única, esta nos asegura que una clase tiene solo una instancia y proporcionar un punto global de acceso a ella. Usamos una clase cuyo atributo es del tipo de la propia clase definido como estático y privado y no null (private static final Program instancia = new <Constructor> () ;) porque sabemos que se va a acceder a él y no sería útil porque no tratamos con una programación multihilo, el constructor se define privado para evitar que se creen mas instancias. Un método de lectura (getInstancia ()) estático nos permite acceder a dicho atributo. La función de este patrón es evitar crear instancias innecesarias que podrían complicar la lectura y la comprensión de nuestro código.

En nuestro código tenemos los estados concretos, que son una subclase de ModoTermostato, que en el paso de parámetros de la función setModo cumple el principio de sustitución de Liskov, ya que la clase derivada puede ser sustituibles por sus clases base, esto también es un ejemplo del Principio de la inversión de la dependencia ya que esta función depende de una abstracción y no de una concreción, así mismo ocurre en la función calentar en la clase termostato. No usamos el Principio de segregación de Interfaces ya que contemplamos pocas funciones abstractas y que estas no difieren tanto en su funcionalidad dentro de la clase Termostato. El Principio Abierto-Cerrado también está presente en nuestro código ya que como he dicho antes, estamos considerando la posibilidad de que nuestro termostato pueda extender sus funciones y sus estados sin que esté de mayores problemas

2. Gestión de equipos de trabajo

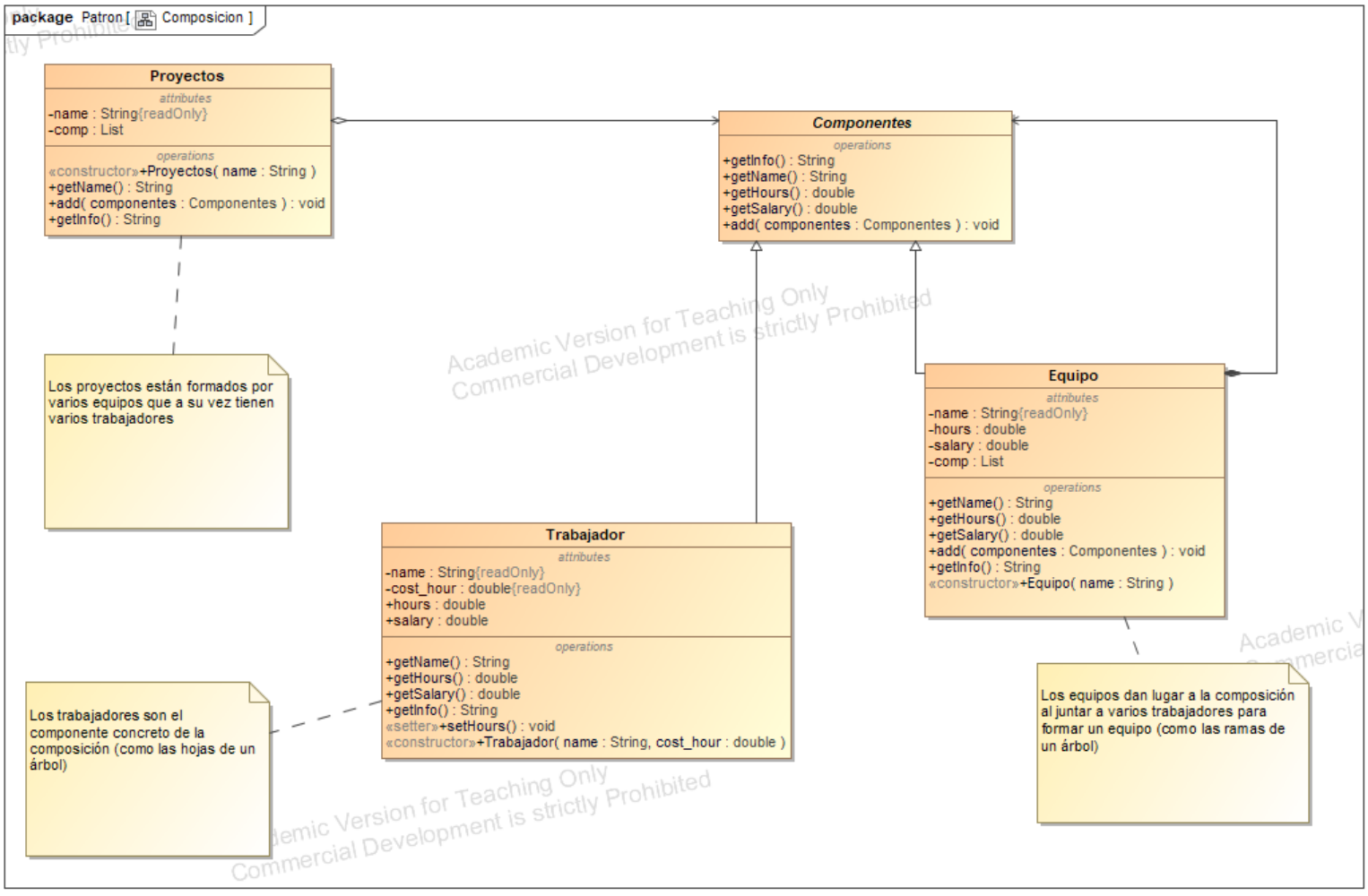
En este ejercicio usaremos principalmente 1 patrón de Diseño y los principios SOLID para desarrollar el código con un buen diseño.

Teniendo en cuenta el Principio de Sustitución de Liskov hacemos la sobreescritura de los métodos abstractos de la clase “Componentes” donde sea pertinente. Como “Trabajador” tiene una relación ES-UN con “Equipo” cuando nos pidan uno podríamos pasar el otro y el usuario no se daría cuenta.

Según el Principio de inversión de Dependencias debemos utilizar clases abstractas y no clases concretas. Las variables de tipo “List” están definidas como tal, pero le damos la implementación concreta de ArrayList.

En el código podemos ver que los métodos de la clase abstracta tienen una implementación por defecto (UnsupportedOperationException()) la cuál podemos cambiar en cada clase sobrescribiendo el método. Para las clases que no sobrescriban el método y hereden de la clase abstracta, el método tendrá la implementación por defecto.

El patrón de Diseño que utilizamos en el ejercicio es el patrón Composición. Este patrón nos permite crear una clase abstracta (Componentes) con los métodos que utilizan las clases “Trabajador”, “Equipo” y “Proyectos”, las cuales heredan de esta. Este patrón nos resulta muy útil a la hora de crear las listas de los equipos y los subequipos que participan en un proyecto ya que nos permite realizar llamadas a las funciones de “Trabajador” mediante el uso de la ligadura dinámica y así obtener la información de todo el equipo que participa en un proyecto en pocas líneas de código. Aquí podemos ver el diagrama de clases del ejercicio:



En el diagrama de comunicación siguiente, podemos ver como el método getInfo() es llamado por proyecto y este devuelve la información de los equipos y los trabajadores del proyecto (o de los trabajadores en caso de que no existan equipos):