Análisis y diseño de software **Pareja Nº 4**
Informe - Práctica 5 Luis Cárabe
Fecha: 29/4/2017 (luis.carabe@estudiante.uam.es)
Profesor: Miguel Ángel Mora David García
Grupo 2201 (david.garcia03@estudiante.uam.es)

Introducción

En este informe explicamos las decisiones de diseño más relevantes y necesarias para la implementación de los distintos ejercicios de los que se compone la *Practica 5* de la asignatura de *Análisis y diseño de software*.

Apartado 1

En este apartado hemos implementado las clases e interfaces necesarias para poder crear un patrón *Observer*. Las clases e interfaces creadas son las siguientes:

1. **ObservableProperty:** Interfaz con los métodos básicos necesarios para que una propiedad con un valor sea observable por un observador.

2. **PropertyObserver:** Interfaz con el único método necesario para implementar un observador. Este método será una rutina que se ejecute cuando el valor al que está suscrito cambia.

3. **AdjustableTime:** Interfaz que extiende (1) y que utiliza valores de tipo entero en lugar de valores de tipo genérico, añadiendo un método que incrementa esos valores.

4. **DefaultObservableProperty:** Se trata de una clase abstracta que implementa (1). Esta compuesta de un atributo valor (valor de la propiedad) y una colección de los observadores suscritos a ella. Agrega un método donde se puede actualizar el valor de la propiedad y avisar a sus observadores de que ha cambiado.

A continuación explicamos la clase **ObservableObserver** donde implementamos una propiedad que a su vez puede observar a otras y cambiar con sus modificaciones (extiende (4) e implementa (2) y (3)).

Esta clase se basa en los métodos **add** y **remove** en la que nos suscribimos o desuscribirnos a una propiedad. Cuando añadimos o eliminamos una propiedad incrementamos (o decrementamos) el valor de nuestra propiedad, además a través de estos métodos avisamos a los observadores que nos vigilan, donde cambiamos sus valores si es necesario a partir de los métodos **incrementTime** y **propertyChanged** que se llaman unos a otros hasta que han modificado todo el grafo de observadores afectados por el cambio.

Nota: En este apartado no nos hemos preocupado de la formación de bucles o de duplicados en las propiedades o propiedades observadas por dos observadores distintos. Estos detalles se trataran más adelante en la implementación de las *Tasks*.

En el tester *Test1.java* simplemente hemos creado una serie de observadores-observables y los hemos ido añadiendo y removiendo para ver las modificaciones en sus valores (ya que estos dos métodos realizan llamadas a todos los demás implementados).

Apartado 2

Las decisiones de diseño de este apartado han sido bastante simples. Por un lado la clase **Task** tiene unos atributos de tiempos, para los que hemos aprovechado la clase **ObservableObserver** creada en el apartado anterior. A parte de la referencia al Task padre y el nombre cabe destacar el conjunto de subtareas, en nuestro caso implementado como un *LinkedHashSet* ya que no nos importa un orden especial las ordenamos por orden de llegada.

La implementación de los distintos métodos ya viene bastante detallada en el enunciado de la práctica. Solo cabe destacar que en los métodos **addTask** y **removeTask** actualizamos la suscripción de los tiempos a los de sus subtareas, así como cuando cambiamos el nodo padre. Con el método **equals** y **hashSet** hemos determinado que dos tareas son iguales si y sólo si tienen el mismo nombre (sin importar mayúsculas o minúsculas).

En cuanto a la clase **Tasks** implementada con el patrón Singleton hemos utilizado un *TreeSet* para el conjunto de tareas, ya que en esta ocasión sí importa el orden (alfabético), teniendo por tanto que implementar *Comparable<Task>* en la clase **Task**.

Para hacer más ligera la codificacion de los testers de este apartado los hemos dividio en dos partes, la primera (*Test2_ Task.java*) profundiza más en el funcionamiento de los métodos de la clase **Task** y su uso individual. El segundo (*Test2_ Tasks.java*) testea la administración de tareas desde el patrón *Singleton* **Tasks**. Además se han creado unos métodos **toString** para mejorar la visibilidad de los resultados de los tester.

Apartado 3

En este apartado hemos implementado la clase **TextConsole** como un patrón *Singleton*. Para los comandos hemos creado una clase privada **Command** con la que poder trabajar (estos contienen el nombre del comando y su operación correspondiente). Los métodos de esta clase son triviales salvo el método **run**. En él ejecutamos un bucle que lee lineas del teclado hasta que llegue a una línea vacía. En el bucle de lecturas troceamos el comando, comprobamos que existe (en caso contrario devolvemos la lista de comandos disponibles) y ejecutamos la operación correspondiente al comando (que ha debido de ser definida previamente mediante una expresión lambda).

Para probar el correcto funcionamiento de la consola hemos creado el programa *Test3.java* donde creamos unos cuantos comandos simples añadiéndoles funcionalidad con expresiones lambda (implementando la interfaz **Function**).

A continuación se muestra unas capturas del funcionamiento de la terminal:

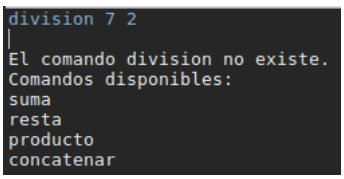


Figura 1: Paso de un comando inexistente

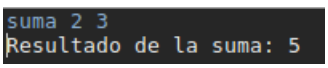


Figura 2: Comando *suma*

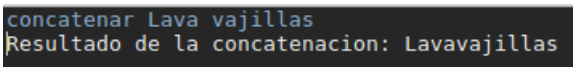


Figura 3: Comando *concatenar*

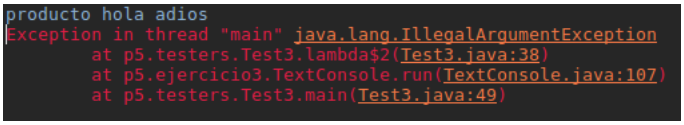


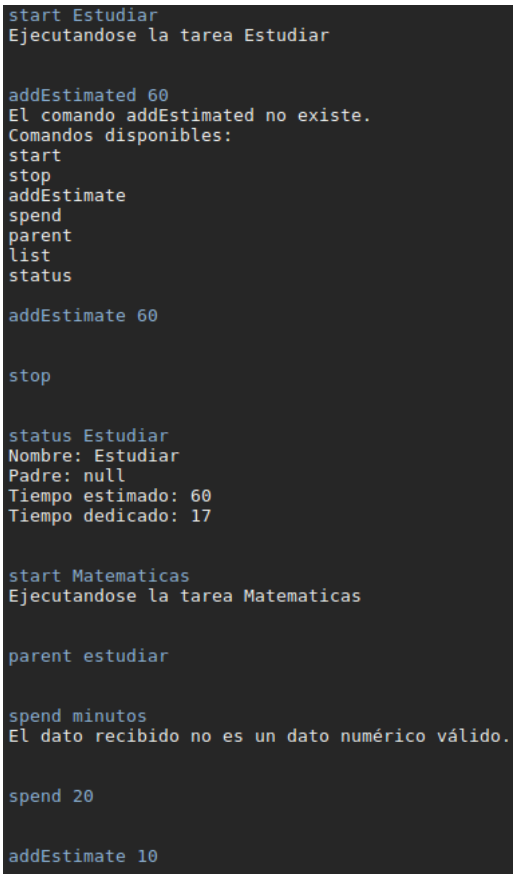
Figura 4: Paso de parámetros inadecuados

Apartado 4

En este apartado hemos creado una clase *TaskConsole* que extiende la clase **TextConsole** añadiendo tres parámetros nuevos. Estos son una tarea actual (ejecutándose en este momento), y unos *long* para medir el tiempo de comienzo de la tarea y el tiempo de fin.

Para implementar los comandos pedidos en el enunciado hemos creado un método **loadCommands** que crea todos ellos mediante expresiones lambda. Para su ejecución se ha hecho un control de parámetros de entrada minucioso para evitar que se lancen excepciones en la ejecución de las funciones de la clase **Task**, ya que la excepción *IllegalArgumentException* no es capturable, generando en su lugar mensajes de error en la pantalla de la consola.

Para la comprobación del correcto funcionamiento de la consola hemos considerado oportuno adjuntar la siguiente salida de la terminal obtenida tras interactuar con distintos comandos:



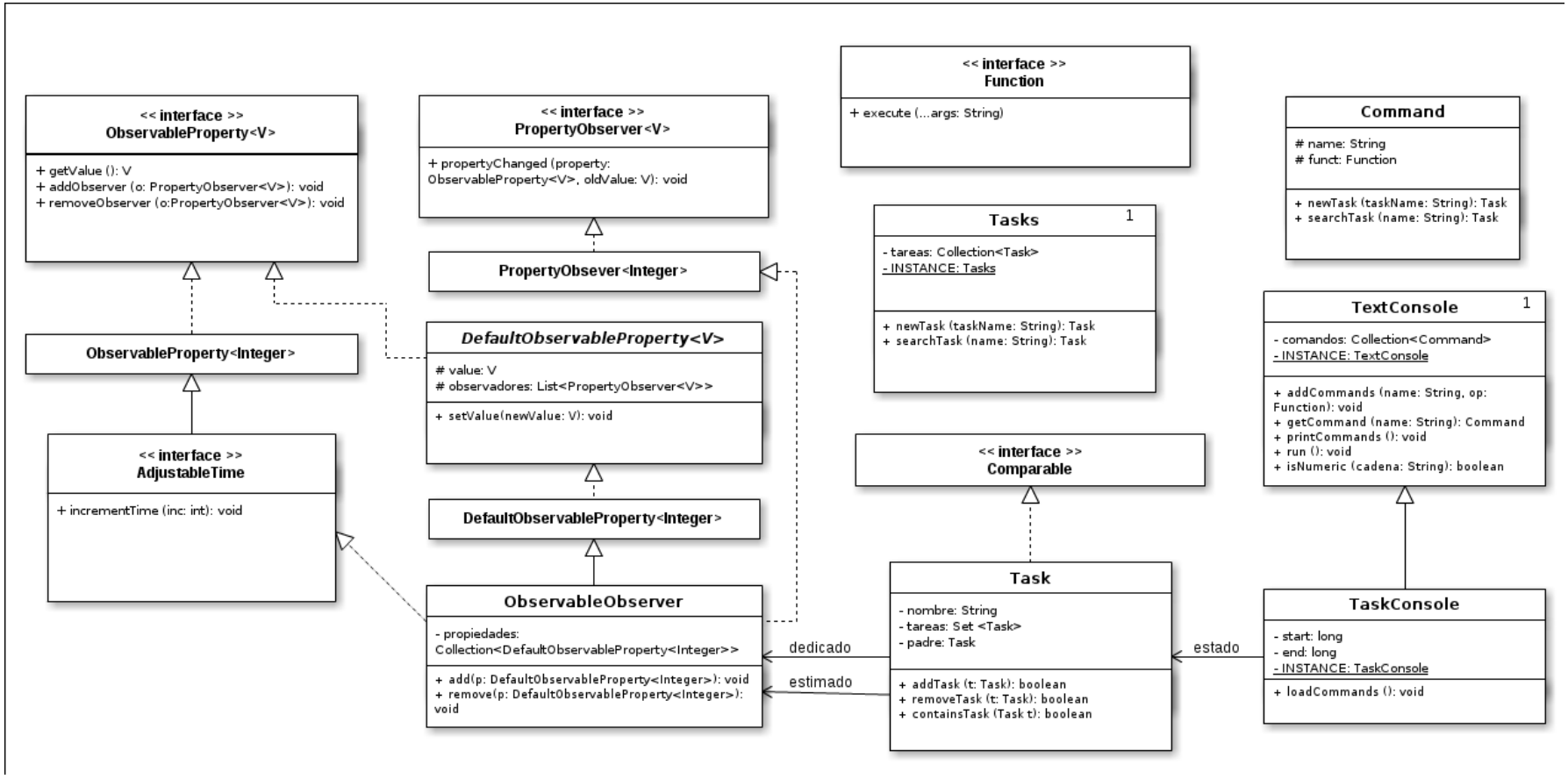
```
stop


status matematicas
Nombre: Matematicas
Padre: Estudiar [68,70]
Tiempo estimado: 10
Tiempo dedicado: 51


status estudiar
Nombre: Estudiar
Padre: null
Tiempo estimado: 70
Tiempo dedicado: 68


stop
Ninguna tarea ejecutandose actualmente.


start Calculo
Ejecutandose la tarea Calculo


parent matematicas


stop calculo
No se esperan argumentos para este comando.


start Algebra
Tarea Calculo detenida.
Ejecutandose la tarea Algebra


parent matematicas


addEstimate 10
```

```
start estudiar
Tarea Algebra detenida.
Ejecutandose la tarea estudiar


status
Nombre: Estudiar
Padre: null
Tiempo estimado: 80
Tiempo dedicado: 127


parent Probabilidad
El padre especificado no existe. Use el comando list para ver las tareas creadas.


parent Calculo
El padre especificado pertenece a las subtareas de la tarea actual.


stop


list
[Algebra [34,10], Calculo [25,0], Estudiar [159,80], Matematicas [110,20]]
```

Apartado 6 - opcional

En un proyecto distinto hemos añadido las funcionalidades pedidas en el ejercicio 6, incluyendo otro tester *Test6.java* en el que hemos probado su funcionamiento.

Diagrama de clases:


<< interface >>
ObservableProperty<V>
+ getValue (): V
+ addObserver (o: PropertyObserver<V>): void
+ removeObserver (o:PropertyObserver<V>): void
<< interface >>
PropertyObserver<V>
+ propertyChanged (property: ObservableProperty<V>, oldValue: V): void
<< interface >>
Function
+ execute (...args: String)
Command
name: String
funct: Function
+ newTask (taskName: String): Task
+ searchTask (name: String): Task
PropertyObsever<Integer>
Tasks
1
- tareas: Collection<Task>
- INSTANCE: Tasks
+ newTask (taskName: String): Task
+ searchTask (name: String): Task
TextConsole
1
- comandos: Collection<Command>
- INSTANCE: TextConsole
+ addCommands (name: String, op: Function): void
+ getCommand (name: String): Command
+ printCommands (): void
+ run (): void
+ isNumeric (cadena: String): boolean
ObservableProperty<Integer>
DefaultObservableProperty<V>
value: V
observadores: List<PropertyObserver<V>>
+ setValue(newValue: V): void
<< interface >>
AdjustableTime
+ incrementTime (inc: int): void
<< interface >>
Comparable
DefaultObservableProperty<Integer>
ObservableObserver
- propiedades: Collection<DefaultObservableProperty<Integer>>
+ add(p: DefaultObservableProperty<Integer>): void
+ remove(p: DefaultObservableProperty<Integer>): void
dedicado
estimado
Task
- nombre: String
- tareas: Set <Task>
- padre: Task
+ addTask (t: Task): boolean
+ removeTask (t: Task): boolean
+ containsTask (Task t): boolean
estado
TaskConsole
- start: long
- end: long
- INSTANCE: TaskConsole
+ loadCommands (): void