# Práctica 5

*Genericidad, Colecciones, Lambdas y Patrones de Diseño*

**Inicio: Semana del 17 de abril.**
**Duración: 2 semanas.**
**Entrega: Hasta el 5 de mayo a las 23:55, excepto grupos del lunes, que entregan el 8 de mayo a las 23:55.**
**Peso de la práctica: 30%**

El objetivo de esta práctica es ejercitar conceptos de orientación más avanzados, como son

- *Diseño de clases genéricas, y altamente reutilizables*
- *Uso de la librería de colecciones Java.*
- *Empleo de patrones de diseño*
- *Uso de expresiones lambda*

---

## Apartado 0. Introducción

En esta práctica crearemos un gestor personal de tareas, que nos permita conocer y gestionar tanto el tiempo estimado como el dedicado a cada una de ellas.

Las tareas tienen un nombre, que será único, y lo usaremos como identificador, y además podrán tener un conjunto de subtareas. Cada tarea proporcionará información sobre el tiempo dedicado y el estimado, directamente, o a través de subtareas. Las subtareas son simplemente tareas que forman parte de otra tarea.

Una tarea podrá estar contenida en, a lo sumo, otra tarea. El sistema de tareas es jerárquico, y no se permitirá que se formen ciclos de tareas. Este sistema de tareas se puede ver por lo tanto como múltiples árboles de tareas.

Para poder tener una visión global, existirá una clase especial, Tasks, implementando el patrón Singleton, que contendrá todas las tareas, y permitirá buscarlas por nombre.

Para poder interactuar con el gestor de tareas, crearemos un sistema genérico para consolas interactivas, programado mediante expresiones lambda, y que podría ser reutilizado de forma sencilla en otras aplicaciones.

---

## Apartado 1. Patrón Observer (2,5 puntos).

El patrón Observer nos permite suscribirnos a cambios en un objeto, de forma que otros objetos puedan reaccionar a dichos cambios. En esta práctica utilizaremos este patrón, entre otros motivos, para poder actualizar los tiempos estimados y totales de las tareas.

La librería estándar de Java implementa una clase Observable, que implementa este patrón, sin embargo no lo utilizaremos directamente, al no disponer de parámetros genéricos. En su lugar usaremos las siguientes interfaces, que nos permiten configurar el tipo de dato de la propiedad observada:

```
public interface ObservableProperty<V> {
	V getValue();
	void addObserver(PropertyObserver<V> o);
	void removeObserver(PropertyObserver<V> o);
}

public interface PropertyObserver<V> {
	void propertyChanged(ObservableProperty<V> property, V oldValue);
}
```

En esta práctica las propiedades representan el tiempo en minutos, usando números enteros. Por conveniencia crearemos una interfaz específica, que permita además ajustar los tiempos. El ajuste puede ser tanto positivo como negativo. En el caso de ajustes negativos, el valor resultante de la propiedad ha de ser igual o mayor a cero, en caso contrario se generará la excepción no controlada *IllegalArgumentException*.

```
public interface AdjustableTime extends ObservableProperty<Integer>{
	void incrementTime(int inc);
}
```

En este apartado implementaremos una clase abstracta, DefaultObservableProperty<V>, que implementará todos los métodos de

ObservableProperty<V>, y un método protegido setValue(V newValue), que actualizará el valor, y si ha cambiado notificará el cambio a todos los observadores registrados. Aunque para esta práctica no se usarán propiedades con valor *null*, esta clase abstracta ha de implementarse de tal forma que soporte dichos valores.

Una propiedad Observable también puede a su vez ser Observador. Este es el caso de los valores de tiempo de las tareas. Para facilitar la implementación, crearemos una clase que extienda DefaultObservableProperty<Integer> e implemente AdjustableTime. Esta clase permitirá añadir o eliminar propiedades observables, de forma que al añadirlas se sume su valor, al eliminarlas se reste, y cuando la propiedad añadida cambie, ajuste el valor total de la propiedad observadora.

Este apartado incluirá un main para demostrar el funcionamiento de las clases implementadas.

## Apartado 2. Tareas (2,5 puntos).

Gracias a las clases creadas en el apartado anterior, podemos implementar de forma sencilla las dos propiedades principales de las tareas, el tiempo estimado, y el tiempo calculado, delegando en ellas la suscripción a los cambios en los tiempos de las subtareas, y la notificación a los observadores.

Tal como avanzamos en la introducción el singleton *Tasks* tendrá todas las tareas creadas organizadas por nombre. Tasks también debería tener algún método para buscar tareas por nombre, sin importar si usamos mayúsculas o minúsculas.

Para crear las tareas utilizaremos un método de factoría en Tasks, *Task newTask(String taskName)*. Este método creará la tarea con el nombre indicado, y producirá la excepción *IllegalArgumentException*, si ya existe una tarea con dicho nombre. El nombre de las tareas no podrá cambiar, y se registrarán automáticamente en el Singleton Tasks al crearse.

También utilizaremos Tasks para simplificar la depuración, al observar los cambios en los tiempos de todas las tareas, y mostrar por pantalla las actualizaciones que se produzcan. Para simplificar no se podrán destruir tareas.

Las tareas implementarán los siguientes métodos, siendo *Task* una tarea:

```
String getName();
boolean addTask(Task t);
boolean removeTask(Task t);
Set<Task> getTasks();
boolean containsTask(Task t);
Task getParent();
void setParent(Task parent);
AdjustableTime getEstimated();
AdjustableTime getDedicated();
```

El método containsTask indica si la tarea está incluida en esta tarea, directa, o indirectamente.

El método addTask devolverá false si la tarea t ya estaba contenida, directa o indirectamente, o true si la ha añadido a su lista de subtareas. Se producirá la excepción *IllegalArgumentException* en caso de que la tarea t no esté contenida y ya tenga otra tarea padre.

El método removeTask elimina la tarea t, que podría estar contenida directa o indirectamente. Devuelve true si la ha eliminado o false en caso contrario. Al eliminar la tarea t, esta se quedará sin tarea padre.

Al añadir o eliminar subtareas se han de ajustar los tiempos de la tarea contenedora, subscribiendose además a los posibles cambios en dichas subtareas. Los cambios en los tiempos se propagan hasta la raíz del árbol de tareas afectado. Esto debería ser sencillo gracias a las clases implementadas en el apartado anterior.

El método setParent cambia la tarea padre, eliminando la tarea de la tarea padre actual, y añadiéndola a la nueva tarea padre, si parent no es null. Este método producirá la excepción *IllegalArgumentException* en caso de que parent esté contenida en la tarea. Como consecuencia del cambio en la jerarquía, se han de ajustar automáticamente los tiempos en las tareas afectadas.

El método getTasks() devuelve la lista de subtareas directas de la tarea. Con el fin de evitar modificaciones no controladas este método devolverá un conjunto no modificable.

En este apartado, además de implementar Task y Tasks, se debe desarrollar un main para demostrar el funcionamiento de las clases implementadas.

## Apartado 3. Consola genérica (1,5 puntos).

Crearemos una clase TextConsole, con un método run() que ejecute el bucle principal de la consola.

En dicho bucle, la consola espera una línea con la forma "comando arg1 arg2 … argn". Si el comando es conocido, lo ejecutará pasando los argumentos a dicho comando. El comando podría fallar, en cuyo caso se mostrará el mensaje de error producido. Si el comando no existe, se mostrará el listado con todos los comandos disponibles. El método run() terminará cuando se introduzca una línea en blanco.

Para modelar el comportamiento de la consola, se podrán definir comandos con el método addCommands(String name, Function op). Function está definido como:

```
@FunctionalInterface
public interface Function {
    void execute(String ...args) throws IllegalArgumentException;
}
```

La excepción IllegalArgumentException tendrá el mensaje de error en caso de que los argumentos sean erróneos, o la operación no se pueda aplicar en este momento por algún motivo.

## Apartado 4. Uniendo todo (2,5 puntos).

En este apartado extenderemos TextConsole, para definir y modificar tareas. La consola tiene como estado la tarea actual, aunque podría no haber ninguna, así como los minutos que han pasado en la tarea actual. Para facilitar la simulación, contaremos los segundos como minutos.

Para ello crearemos los siguientes comandos:

a) `start taskName`
   El comando start creará la tarea con el nombre, taskName, indicado, si no existe. Además fijará la tarea como tarea actual, y detendrá la tarea anterior.
b) `stop`
   Detiene la tarea actual, pasando a no tener ninguna tarea actual. Al detener la tarea, se añade a tiempo dedicado los minutos (segundos en la simulación) que han pasado desde el comando start.
c) `addEstimate minutos`
   Añade los minutos indicados al tiempo estimado de la tarea actual. Minutos puede ser negativo, en cuyo caso el tiempo se restará.
d) `spend minutos`
   Similar al comando anterior, para ajustar el tiempo dedicado. Al igual que con estimate, se admitirán correcciones usando tiempos negativos.
e) `parent [parentTask]`
   Cambia la tarea padre de la tarea actual. Si no se indica la tarea padre, equivaldrá a eliminar la tarea de su tarea padre, mediante setParent(null)
f) `list`
   Lista en orden alfabético todas las tareas.
g) `status [taskName]`
   Imprime la información de la tarea indicada, su nombre, tarea padre, y tiempos. Si se omite el nombre, se mostrará la tarea actual.

Se ha de mostrar un mensaje de error al detectar acciones o argumentos erróneos, evitando ejecutar acciones que produzcan algún tipo de error o excepción.
Se aconseja implementar los comandos como métodos privados de la consola. Estos se pueden añadir a los comandos de la consola mediante una expresión lambda, o una referencia al método que lo implementa.

## Apartado 5. Diagrama de clases y explicación del diseño (1 punto)

Debes entregar un diagrama de clases que muestre el diseño efectuado. No olvides que el objetivo del diagrama es explicar el diseño con un nivel de abstracción adecuado, no es "Java en dibujos". De esta manera, debes obviar constructores, getters y setters, y debes representar las colecciones, arrays y referencias usadas como asociaciones del tipo más adecuado. No olvides incluir las interfaces que has usado, y las relaciones de implementación entre clases e interfaces. Incluye una pequeña explicación del diseño, así como de las decisiones que has tomado.

## Apartado 6. Resumen de tareas (Opcional, 1 punto)

Las tareas forman un bosque, donde hay varios árboles de tareas, sin embargo no tenemos ninguna forma sencilla de calcular los totales de todas las tareas.

Para solucionar este problema, añadiremos dos propiedades al singleton Tasks, estimatedTotal() y dedicatedTotal(), con los tiempos totales. Además añadiremos dos métodos, addRoot(Task) y removeRoot(Task), que serán usados desde las tareas para añadir o eliminar la raíz, cuando cambie la propiedad "parent" de la tarea, o al construirse sin tarea padre.

Tasks también observará estas dos propiedades, para mostrar el valor global cada vez que cambie.

## Normas de Entrega:

- Se deberá entregar
  - un directorio **src** con el código Java de cada apartado, incluidas las pruebas JUnit.
  - un directorio **doc** con la documentación generada
  - un archivo PDF con una breve justificación de las decisiones que se hayan tomado en el desarrollo de la práctica, los problemas principales que se han abordado y cómo se han resuelto.
- Se debe entregar un único fichero ZIP con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2261, entregarían el fichero: GR2261_MarisaPedro.zip.