Programación Orientada a Objetos

Curso 2023/2024

Parcial 2

Alumno/a: _____ **GRUPO:**_____

Previo: para la gestión de fechas se utilizará la clase `LocalDate`. Las fechas pueden ser comparadas con los métodos `isAfter` e `isBefore`, que comprueban estrictamente si la fecha es posterior o anterior, respectivamente, al parámetro. Además, el método de clase `now()` retorna la fecha actual.

1. (**0,75 ptos**) La clase `Tarea` define ***objetos inmutables*** con dos propiedades: descripción (texto) y plazo (fecha).

a) Declara los atributos de la clase.

b) Dado que la propiedad `plazo` es **opcional**, declara los **constructores** de la clase. Una tarea no tiene plazo si la propiedad plazo tiene el valor `null`.

2. (**0,5 ptos**) Queremos introducir en la clase anterior dos nuevas propiedades:

a) tiene plazo (`tienePlazo`): propiedad booleana que indica si la tarea tiene definido un plazo de vencimiento.

b) está vencida (`isVencida`): propiedad booleana que indica si el plazo ha vencido, esto es, si es anterior a la fecha de hoy. Una tarea sin plazo nunca vence.

3. (**3 ptos**) Una **Agenda** gestiona las tareas utilizando dos propiedades: una lista con todas las tareas registradas y otra con todas las que han sido completadas. El constructor únicamente inicializa las colecciones.

```
public class Agenda {

    private LinkedList<Tarea> todas;
    private LinkedList<Tarea> completas;
}
```

a) Implementa un método de consulta **eficiente** de la propiedad `todas` que además proteja la colección almacenada en el atributo.

b) Introduce un método que calcule una lista con las tareas pendientes (`getPendientes`): no completadas y no vencidas.

c) Introduce un método, **sobrecarga** del método anterior, que calcule una lista ordenada con las tareas pendientes. El método recibe como parámetro el criterio de ordenación.

d) Implementa un criterio de ordenación de tareas basado en la propiedad `plazo`. La comparación de las fechas se hará en orden ascendente (orden natural de la clase `LocalDate`), esto es, las más antiguas las primeras. Nótese que `plazo` es una propiedad opcional y que dos tareas que no tienen plazo se consideran iguales y han de situarse las últimas, es decir, son mayores que cualquier tarea con plazo.

e) Implementa un método que retorne una colección con los distintos plazos de las tareas pendientes.

f) Implementa un método con visibilidad para las clases descendientes que elimine una tarea.

4. **(2 ptos)** Queremos ofrecer una implementación eficiente de una operación de consulta de las tareas cuyo plazo concluye en una fecha dada. Para ello introduce un *atributo de implementación* que almacene asociaciones entre una fecha y las tareas que tienen ese plazo de vencimiento.

   a) Indica la declaración del atributo.
   b) Implementa el método agregar tarea (`addTarea`), que recibe como parámetro una tarea que se quiere añadir a la agenda. Esta operación debe comprobar que no esté repetida. El método retorna un valor booleano que indica si ha sido insertada.
   c) Implementa el método agregar tareas (`addTareas`), que recibe varias tareas como argumento variable. El método devuelve una lista con las tareas que han sido añadidas.
   d) Implementa una operación de consulta de las tareas que tengan como plazo una fecha establecida como parámetro. Esta operación debe utilizar el atributo de implementación. En caso de que no esté la fecha en el mapa se retornará una colección vacía.

5. (**1 pto**) Queremos introducir un método que muestre una selección ordenada de *tareas destacadas*. Para ello, la implementación se apoya en el método que obtiene las tareas pendientes (`getPendientes`), ordenadas por plazo (criterio de ordenación). De esa lista se elegirán las tareas consideradas como "destacadas". El criterio para decidir si una tarea es destacada depende de los tipos de tareas (clases descendientes). Implementa este método aplicando el patrón del **método plantilla**:

   a) Indica qué cambios debe realizarse en la clase para poder introducir este método.
   b) Implementa el método.
   c) Indica qué cambios debería hacerse a la implementación del método para evitar que sea redefinido en las clases descendientes.

6. (**1 pto**) Implementa un tipo de Agenda llamada **AgendaPersonal** con una nueva propiedad: colección de tareas urgentes. Introduce como funcionalidad dos métodos para indicar que una tarea es urgente (`setUrgente`) o deja de ser urgente (`unsetUrgente`). Ambas operaciones devuelven un booleano si se ha realizado la acción. Implementa también el método de consulta para comprobar si una tarea (pasada como parámetro) es urgente (`isUrgente`), esto es, si se encuentra en la colección de tareas urgentes. Por otra parte, cuando se introduce una nueva tarea en la agenda se consideran urgente si tiene plazo. Por último, en esta clase se considera que una tarea es *destacada* si es urgente y el plazo concluye hoy.

7. (**0,75 ptos**) Introduce el método `clone` en la jerarquía de agendas siguiendo las recomendaciones de la asignatura. La copia de una agenda tiene las mismas tareas y las tareas urgentes siguen siéndolo en la copia. Indica los cambios necesarios en las clases y la implementación de los métodos. **Nota**: se puede utilizar el método de apoyo `copiaSuperfial` sin necesidad de ser declarado. Este método realiza una llamada al método `clone` de la clase `Object` que realiza la *copia superficial* del objeto.

8. (**0,5 ptos**) Implementa el método `toString` en la clase `AgendaPersonal`, suponiendo que la clase padre lo tiene implementado.

9. (**0,5 ptos**) Dada una lista de agendas (`LinkedList<Agenda> agendas`) que contiene objetos de toda la jerarquía, escribe un fragmento de código que muestre por la consola las tareas *urgentes* de las agendas personales.