Examen de la convocatoria de Septiembre de 2009

Bloque II. Problemas **5 puntos**

Se debe tener un mínimo del 40% de la puntuación de este bloque para poder aprobar el examen.

En este apartado se proponen ejercicios de programación que deben ser resueltos utilizando el lenguaje de programación **Java**. Se valorará que los ejercicios cumplan la funcionalidad requerida y que el código cumpla los **criterios de calidad** expresados en el tema 1 de la asignatura: extensibilidad, reutilización, principios de diseño modular, etc.

---

1. Queremos implementar una aplicación para la gestión del cobro en autopistas de peaje. Un usuario de una autopista accede a ella por un punto kilométrico y a la entrada se le entrega un ticket donde se registra el kilómetro de entrada y el precio por kilómetro de esa autopista. Cuando abandona la autopista se registra en el ticket el kilómetro de salida y se calcula la tarifa. Existe un tipo de ticket general para la mayor parte de vehículos y tickets especiales para autobuses y camiones.

   El cálculo de la tarifa de un ticket general aplica la fórmula básica *kilómetros recorridos x precio por kilómetro*. En el caso de los autobuses la tarifa se calcula como la fórmula básica multiplicada por el número de pasajeros del autobús.

   a) (**0,75 puntos**) Implementa las clases `Ticket` y `TicketAutobus`.

   b) (**0,25 puntos**) Explica los conceptos de polimorfismo y ligadura dinámica utilizando el código implementado en el apartado anterior.

   c) (**0,5 puntos**) Según el tipo de ticket es posible aplicar una reducción de la tarifa. Para los autobuses se aplica una reducción del 10% a partir de 100 kilómetros y en los camiones se aplica un 5% por cada 500 kilómetros. Aplica el patrón de diseño *Método Plantilla* en el método que calcula la tarifa para que contemple el porcentaje de reducción e indica los cambios necesarios en las clases `Ticket` y `TicketAutobus`.

2. Un pool es una estructura de datos que almacena objetos que van a ser utilizados en varias ocasiones dentro de una aplicación. Por ejemplo, para trabajar con bases de datos se crea un pool de conexiones cuyos objetos serán utilizados cada vez que se necesiten. De este modo se pueden reutilizar objetos ya creados y se evita la creación de objetos costosos.

   Inicialmente el pool se encuentra vacío y podemos añadir en cualquier momento nuevos objetos al pool. El cliente de un pool pide prestado un objeto para utilizarlo y lo devuelve al pool cuando ya no lo necesita. Por tanto, la funcionalidad que ofrece un pool es la siguiente:

   - Método para añadir un nuevo objeto al pool. Este objeto se guardará en la colección de objetos disponibles.
   - Método para pedir prestado el primer objeto disponible del pool. El objeto deja de pertenecer a la colección de objetos disponibles y pasa a la colección de prestados. Si no hay objetos disponibles, devuelve una referencia nula.
   - Método para devolver un objeto al pool. El objeto pasa de la colección de prestados a disponibles.

   a) (**0,5 puntos**) Implementa la clase `Pool` de manera que se garantice que todos los objetos que contiene sean compatibles con un tipo establecido en su creación (por ejemplo, objetos `ConexionBD` o compatibles).

   b) (**0,5 puntos**) Un pool automático es un tipo de pool que intenta recuperar los objetos prestados y no devueltos durante cierto tiempo. Cuando un cliente le pide prestado un objeto y no encuentra ninguno en la colección de disponibles, intenta devolver de entre los objetos prestados el primero que esté inactivo más de 5 segundos. Por tanto, es necesario poder consultar el tiempo de inactividad de los objetos almacenados en este tipo de pool. Implementa la clase `PoolAutomatico`.

3. Validadores y acumuladores.

   a) (**0,5 puntos**) Queremos disponer de objetos que sean capaces de validar cadenas de texto para comprobar si cumplen un formato. Por ejemplo, nos puede interesar saber si una cadena representa un valor entero o si cumple una expresión regular. Nótese que un objeto validador es capaz de realizar sólo un tipo de validación.

   - Define el tipo de datos `Validador`.
   - Implementa un tipo de validador que denominamos `ValidadorEnteros` que compruebe si una cadena es un entero mayor o igual que un valor dado. Por ejemplo, podremos crear un validador que compruebe los números mayores o iguales que 0, otro para los números mayores o iguales que 1000, etc.

     Nota: la clase `Integer` ofrece el método de clase `parseInt` que intenta convertir una cadena en entero. Si la conversión es incorrecta, este método lanza la excepción runtime `NumberFormatException`.

b) (**0,75 puntos**) Un acumulador es un objeto responsable de mantener una lista de números naturales (enteros no negativos) y de actualizar sus valores cuando sea necesario. Un objeto acumulador debe ofrecer la siguiente funcionalidad:

- Se construye a partir de una lista de números naturales, por ejemplo (1, 5, 7). Con esta lista se actualiza su propiedad *lista acumulada*. La lista acumulada puede tener cualquier tamaño, pero no podrá variar de tamaño una vez establecida en el constructor.
- Permite consultar la lista acumulada.
- Ofrece un método para *acumular* los valores de una nueva lista sobre la lista acumulada. Por ejemplo, si la lista acumulada del objeto es (1, 5, 7) y este método toma como parámetro la lista (3, 2, 1), sumará estos valores a la lista acumulada dando como resultado (4, 7, 8).

  Este método sólo podrá ejecutarse si el tamaño de la lista que obtiene como parámetro es igual al tamaño de la lista acumulada y los números que almacena la lista son naturales.

  Asimismo, el método no podrá cumplir su tarea (incumple su postcondición) si alguno de los elementos de la lista acumulada sobrepasa el límite superior de los números enteros. **Nota**: en Java se produce esta situación si el resultado de la suma de dos números naturales devuelve un número negativo.

Implementa la clase `Acumulador` aplicando los criterios de diseño estudiados en el tema 4 de la asignatura (pre y postcondiciones, diseño por contrato y excepciones).

c) (**0,75 puntos**) Queremos añadir a la clase `Acumulador` un método que actualice los valores de la *lista acumulada* aplicando una operación sobre ellos. Ejemplos de operación podrían ser, sumar 1 a cada valor o multiplicarlo por sí mismo. Implementa este método y la operación que sume 1 a todos los valores de la lista acumulada.

d) (**0,5 puntos**) Escribe un programa Java que haga uso del código implementado en los apartados a), b) y c). La funcionalidad del programa debe ser:

- Crea un objeto acumulador con una lista que contiene un único valor: (0).
- Recorre todos los argumentos del programa y comprueba si son números naturales. Para ello instancia un objeto `ValidadorEnteros` (apartado a) para la validación de enteros iguales o mayores que 0. Si la validación es correcta, se convertirá la cadena en entero utilizando el método `parseInt` de la clase `Integer` (véase apartado a). Una vez obtenido el número se crea una lista con ese único elemento y se acumula en el objeto acumulador.
- Si se supera el rango de los enteros en la acumulación o la validación de alguno de los argumentos del programa es incorrecta, se mostrará un mensaje de error al usuario y finaliza el programa.
- Si el procesamiento de los argumentos del programa ha sido correcto, aplicaremos el método y la operación implementados en el apartado c) para sumar 1 a todos los valores de la lista acumulada del acumulador.