Problemas de pruebas unitarias

Para no alargar demasiado los enunciados, en algunos casos usaremos "Java abreviado", con el que mostraremos solamente lo esencial de cada clase de cara a poder definir pruebas.
Únicamente se mostrará lo siguiente de las clases:

- En vez de usar llaves para delimitar bloques se usará indentación (*à la python*)
- parámetros del constructor (si éste los necesita)
- métodos que implementa

Problemas simples (clases 'independientes')

Problema 1. Clase *Dictionary*

Utilizaremos la clase `Dictionary` para implementar un diccionario en el que cada palabra (entrada al diccionario) pueda tener distintos significados o definiciones.
Así, consideraremos las siguientes definiciones:

```
1 interface Dictionary {
2   void defineWord(String word, String definition);
3   List<String> getDefinitions(String word)
4     throws NotDefinedException;
5 }
6
7 public class DictionaryImpl implements Dictionary {
8   ...
9 }
```

con la siguiente especificación:

- *defineWord* añade una definición a las ya existentes de una palabra, o bien la añade como una nueva entrada (la palabra junto con su definición), si ésta aún no existe en el diccionario.
- *getDefinitions* obtiene la lista de definiciones asociadas a una palabra. En caso de que la palabra no esté incluida en el diccionario lanza la excepción *NotDefinedException*.

Definid una o más clases de prueba para la clase *DictionaryImpl*, teniendo en cuenta todos los posibles casos de test, a fin de preparar un juego de pruebas unitarias suficientemente exhaustivo.
¿Qué otra excepción convendría definir?

Aspectos de Java: *Excepciones*

- Definición de una excepción propia:

```
public class NotDefinedException extends Exception {
    public NotDefinedException(String message){
        super(message);
    }
}
```

- Lanzamiento explícito de una excepción: instrucción `throw`:

```
throw new NotDefinedException("No incluida en
                                    diccionario");
```

- Declaración de que un método puede lanzar una excepción: cláusula `throws` en la cabecera del método.
- La clase invocadora de este método deberá manejar la/s excepción/nes declarada/s utilizando el bloque `try-catch`.
  Otra opción, **que es la que utilizaremos en el código de test**, es **relanzar** esa excepción, incorporando esta misma cláusula `throws` en la cabecera del método invocador, y de ese modo traspasar la responsabilidad de manejarla al nivel superior.

```
public List<String> getDefinitions(String word) throws
          NotDefinedException { . . . }
```

Problema 2. Clase *Merger*

Considerad la clase Merger que contiene un método para mezclar dos listas de enteros ordenadas crecientemente y sin números repetidos, en una única lista también ordenada y que tampoco contiene repetidos.

```
1 public class Merger
2   public List<Integer> mergeSorted(
3     List<Integer> list1,
4     List<Integer> list2)
5       throws IllegalArgumentException...
```

La excepción se lanza cuando alguno de los argumentos no cumple con la precondición de estar ordenado crecientemente y no tener elementos repetidos.

Definid pruebas unitarias para comprobar el funcionamiento correcto de esta clase.

Aspectos de Java

- Para obtener listas con los elementos deseados puede usarse el método estático *`asList`* de la clase *`Arrays`*. Por ejemplo: `List<Integer> aListExample = Arrays.asList(1, 2, 3, 4, 5);`
- Para obtener una lista vacía, podéis usar el método estático *`emptyList`* de la clase *`Collections`*: `Collections.emptyList()`

Problema 3. Clase *`Receipt`* básica

Se dispone de la clase *Receipt* (recibo) que tiene los siguientes métodos:

```
1 public class Receipt
2   public void addLine(BigDecimal pricePerUnit,
3                       int numUnits)
4     throws IsClosedException
5   public void addTaxes(BigDecimal percent)
6     throws IsClosedException
7   public BigDecimal getTotal()...
8 }
```

Las especificaciones son las siguientes:

- *`addLine`* añade una línea al recibo utilizando dos argumentos: *`numUnits`*, número de unidades y *`pricePerUnit`*, precio unitario.
- *`addTaxes`* añade al recibo los impuestos correspondientes a un tanto por ciento *`percent`* y cierra el recibo.
- Las operaciones *`addLine`* y *`addTaxes`* lanzan *`IsClosedException`* si se aplican sobre un recibo ya cerrado.
- *`getTotal`* devuelve el total del recibo y puede llamarse en cualquier momento.

Definid una o más clases de prueba para la clase *Receipt*.

Problema 4. *Task Scheduling*

Se dispone de las siguientes clases que se utilizan en un sistema de planificación de tareas:

```
 1 interface Task {
 2   BigDecimal costInEuros();
 3   int durationInDays();
 4 }
 5
 6 public class Simple implements Task ...
 7   public Simple(BigDecimal euros, int days)
 8   public BigDecimal costInEuros()
 9   public int durationInDays()
10
11 public abstract class Composed implements Task...
12   public void addSubtask(Task subtask)
13   public BigDecimal costInEuros()
14
15 public class Sequential extends Composed
16                          implements Task ...
17   public int durationInDays()
18
19 public class Parallel extends Composed
20                        implements Task ...
21   public int durationInDays()
```

Las especificaciones son las siguientes:

- El coste y duración en una tarea simple son los que se indican en el constructor.
- El método *`addSubtask`* añade una subtarea a una tarea compuesta.
- En una tarea compuesta el coste es la suma de los costes de las subtareas (a obtener mediante el método *`costInEuros`*).
- En una tarea secuencial la duración es la suma de las duraciones de las subtareas (a obtener mediante el método *`durationInDays`*).
- En una tarea paralela la duración es el máximo de las duraciones de las subtareas (método *`durationInDays`*).

Definid pruebas unitarias para comprobar el funcionamiento correcto de estas clases.

Problemas utilizando Dobles

Problema 5. Clase *Receipt* searchable

El Problema 3 presentaba una clase *Receipt* que era poco real ya que, normalmente la información sobre precios y descripción de los productos reside en una base de datos, a la que acceder mediante el identificador del producto.
Por ello, la nueva versión, tendrá la forma:

```
1 public class Receipt
2   private ProductsDB = new ProductsDB();
3   public void addLine(String productID,
4                       int numUnits)
5     throws IsClosedException, DoesNotExistException
6   public void addTaxes(BigDecimal percent)
7     throws IsClosedException
8   public BigDecimal getTotal()
9 }
```

Lo que queremos es poder probar esta versión de la clase *Receipt* de forma independiente de la base de datos (podéis referenciarla *ProductDB*). Modificad y/o añadid lo que haga falta para poder probar la clase *Receipt* de forma independiente y desarrollar un conjunto de tests que prueben su funcionalidad.

La operación en la que estamos interesados de la base de datos es:

```
public BigDecimal getPrice(String productID)
        throws DoesNotExistException;
```

Ahora la nueva versión de *addLine* lanza también *DoesNotExistException* cuando se le pasa un identificador de producto que no existe en la BD.

Definid pruebas unitarias para la clase *Receipt* definida aquí.

Problema 6. `Forest-Fire Model`

El modelo FFM (Forest-Fire Model) simula la evolución de un incendio en un área determinada[1]. El modelo FFM consiste en una cuadrícula de celdas que pueden tener tres estados posibles:

- vacía
- ocupada por un árbol
- ardiendo

La evolución del sistema en el tiempo viene determinada por cuatro reglas que se ejecutan simultáneamente:

1. Una celda ardiendo se convierte en un espacio vacío
2. Un árbol arderá si al menos un árbol vecino suyo está ardiendo
3. Un árbol comienza a arder con probabilidad f, incluso si no tiene ningún vecino ardiendo
4. Un árbol brota en un espacio vacío con probabilidad p

Nosotros nos centraremos en la clase *Cell*, que representará cada una de las celdas de la cuadrícula. Cada celda sabe el estado en el que se encuentra y es responsable de gestionar las transiciones.

- La clase guardará dos constantes (miembros estáticos) de tipo *double*, que serán las probabilidades de combustión espontánea y de nacimiento de un árbol. Por ejemplo:
  - `private static double PBURN = 0.00006;`
  - `private static double PGROWTH = 0.01;`
- El constructor de esta clase no tendrá parámetros, ya que las celdas estarán inicialmente vacías.
- La clase ofrecerá un método para conocer el estado en que se encuentra una celda:
  - `State getState()`
- Además, dispondremos de un método que hará que la celda cambie de estado según las reglas. Dicho método recibirá un parámetro adicional para indicar si alguna celda vecina está ardiendo:
  - `public void step(boolean hasBurningNeighbour)`

La clase, internamente, usa una instancia de la clase *java.util.Random* para generar números aleatorios para ambas probabilidades. Deberéis sustituirla (clase sustituta) para poder hacer las pruebas pertinentes.

Pista: Para hacer más sencillas las pruebas podéis decidir cómo os gustaría que el objeto que servirá para tomar las decisiones aleatorias

[1] Esta clase formó parte del enunciado de la segunda práctica de Programación 2 del curso 2011/2012.

se usará (básicamente decidir una interfaz con los métodos que os gustaría que tuviera para generar ambas probabilidades). Estos métodos servirán tanto para implementar la funcionalidad en la aplicación como para ser sustituidos en la clase que haga de doble en las pruebas.

Problema 7. `Apocalipsis`

Otro modelo similar al anterior sirve para simular la propagación de una enfermedad (sí, también lo podéis usar para modelar un apocalipsis zombi, poniendo la probabilidad de recuperación a 0).
El modelo, conocido como SIR (Susceptible-Infected-Recovered) es como sigue:

- Cada individuo puede estar en tres estados posibles: S, I, R
- Los individuos susceptibles pueden infectarse espontáneamente con probabilidad p
- Una vez infectados, tardan K días en pasar a recuperados
- La enfermedad genera inmunización, por lo que un individuo recuperado ya no vuelve a infectarse más

Basándonos en la estructura de la clase del Problema 6, definid pruebas para comprobar que el estado de los individuos evoluciona de la forma correcta. El método en cuestión será:

```
public void step();
```

Problema 8. `Clase Receipt printable`

Suponed que a la clase *`Receipt`* del Problema 5 le añadimos un método para imprimir el recibo, `printReceipt()`, con la siguiente cabecera:

```
void printReceipt() throws DoesNotExistException,
                           IsNotClosedException
```

donde *`IsNotClosedException`* se lanza si se intenta aplicar sobre un recibo sin cerrar.

Para ello, la clase *`Receipt`* depende de la clase *`ReceiptPrinter`*, la cual tiene cinco métodos:

- `public void init();`
- `public void addProduct(String description,`
  `int quantity,`
  `BigDecimal price);`
- `public void addTaxes(BigDecimal taxes);`

- `public void print(BigDecimal total);`
- `public String getOutput();`

Veamos el comportamiento de las operaciones (están muy simplificadas respecto a un caso real, ya que no consideramos alineamientos, etc, etc):

- *`init`* añade la cabecera del recibo que, simplificando, consiste en la cadena "Acme S.A." y un salto de línea.
- *`addProduct`* añade la línea de un producto. De cada producto se imprimen los tres campos pasados como argumento (descripción, cantidad y precio), transformados en `Strings` si es necesario, y separados por tabuladores. Por supuesto, se invocará tantas veces como líneas existentes en *`Receipt`*.
- *`addTaxes`* añade una línea con el texto "TAXES", un tabulador y el valor de los impuestos.
- *`print`* añade una línea con el texto "----------", y a continuación otra línea con el texto "TOTAL", un tabulador y el valor total, impuestos incluidos.
- *`getOutput`* retorna la cadena de texto resultante.

Para hacer los tests, sustituiréis la clase *`ReceiptPrinter`* por una doble, de manera que el test pueda comprobar lo que se ha impreso. La descripción y el precio de un producto se obtendrá de la base de datos (*`ProductsDB`*), a tratar convenientemente. Supondremos que *`ProductsDB`* dispone del siguiente método:

```
public ProductDTO getProduct(String productID)
          throws DoesNotExistException;
```

el cual devuelve una instancia de la clase `ProductDTO` siguiente:

```
1 public class ProductDTO {
2   private String productID;
3   private String description;
4   private BigDecimal price;
5
6   // getters and setters
7 }
```

para obtener toda la información necesaria sobre un producto.

Definid pruebas unitarias para la clase *`Receipt`* definida aquí.