

EVALUACION NO CONTINUA

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 1. (3 puntos)

Se desea crear un servicio automático de notificación de cambios en la web. El sistema admite distintos mecanismos para enviar los avisos de cambio, como e-mail, twitter, o SMS.

De cada usuario se tendrá su nombre, y una lista de mecanismos de notificación registrados. El usuario le asigna un nombre único a cada mecanismo registrado, además de los datos propios de cada tipo de mecanismo:

- Para el e-mail indica la dirección de correo y el texto que se utilizará para el asunto del mensaje.
- En el caso de Twitter se indicará la cadena de 140 caracteres que se enviará.
- En el caso de SMS se indicará la cadena de 140 caracteres que se enviará, y un número de teléfono, que también será una cadena de caracteres.

Los usuarios expresan su interés en una página web indicando la dirección web, y el mecanismo, de entre los registrados por el usuario, que quiere utilizar para recibir los avisos.

Todas las páginas se comprueban una vez al día, y para ahorrar recursos, el sistema agrupará a todos los usuarios interesados en seguir una misma página. El sistema de detección de cambios se basa en calcular un código hash del contenido de la página, y compararlo con el código hash calculado la vez anterior. Asumiremos que si hay un fallo en la descarga, el código hash será -1.

Se pide:

- Representa en UML las clases y relaciones del sistema anterior. En el diagrama no es necesario incluir los métodos ni los constructores. Tampoco se incluirá la interfaz de usuario.
- Para las siguientes operaciones, indica la clase más apropiada para contenerla, el nombre del método, argumentos, y tipo de resultado.
 - Subscribirse a los cambios de una página web
 - Dar de baja un mecanismo de notificación de un cliente
- Implementa, en pseudocódigo o código java, el método que sirve para avisar, cuando cambia una página, a los interesados en dicho cambio.
- ¿Qué patrón o patrones de diseño has utilizado?

Nota: No es necesario detallar como es el proceso de descarga de una página web, por ejemplo se puede asumir que existe una clase URL que ya tiene implementada dicha funcionalidad. De forma similar, tampoco es necesario detallar el sistema de envío de notificaciones, será suficiente con un método send() o similar en la clase o clases que apropiadas.

EVALUACION NO CONTINUA

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 2. (2.5 puntos)

Como parte de una simulación de juegos de guerra se desea definir la interfaz `Combatiente` implementada por las clases `Soldado` y `Comando`. Los comandos reciben un nombre al crearse vacíos, y se le pueden ir añadiendo soldados u otros comandos con las siguientes limitaciones. Si se intenta añadir otro combatiente (sea soldado o comando) a un comando que ya pertenezca a él directamente (es decir, no a través de otro comando), no se añadirá ni cambiará nada en el comando. Tampoco cambiará nada si se intenta añadir un comando a sí mismo, ni cuando se le intenten añadir objetos no creados, ni cuando se le intenten añadir otros comandos que ya estén incluidos en él ya sea directa o indirectamente (a través de otros comandos), o en los que él esté incluido ya sea directa o indirectamente. Ver ejemplos de estas restricciones en el código dado. Cada soldado se configura con un valor entero que representa su fuerza. La fuerza de un comando se calcula sumando la fuerza de todos los soldados incluidos directa o indirectamente en el comando, pero sin sumar más de una vez la fuerza de un mismo soldado.

Se pide:

(a) utilizando el patrón de diseño más adecuado, implementa la interfaz `Combatiente` y las clases `Soldado` y `Comando` para que el siguiente método `main` produzca la salida indicada abajo, teniendo en cuenta también el código dado para los métodos `toString()` de ambas clases; y

(b) indica qué patrón de diseño has empleado y establece la relación entre las clases e interfaces de tu implementación y las de dicho patrón.

```
// Método dado para la clase Comando
@Override public String toString() { return "C" + getNombre() + ":" + getMiembros(); }
// Método dado para la clase Soldado
@Override public String toString() { return "S(" + getFuerza() + ")"; }
```

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Combatiente[] s = {new Soldado(10), new Soldado(11), new Soldado(12), new Soldado(13) };
        Combatiente[] c = { null, new Comando("1"), new Comando("2"), new Comando("3") };

        s[1].add(s[2]); // no se añade nada a un soldado
        c[1].add(c[0]).add(s[2]); // añade solo s[2], pero no c[0] que es null
        c[1].add(s[2]).add(s[0]); // añade solo s[0], pero no s[2] (no lo repite)
        c[1].add( c[3].add(s[3]) ); // OK a c[1] le añade c[3] que lleva dentro s[3]
        c[2].add( c[1] ); // OK añade c[1] a c[4], y después c[4] a c[2]

        System.out.println( c[2].contiene(c[3])); // imprime true
        c[2].add(c[3]); // no añade nada (ver println anterior)
        c[3].add(c[2]); // no añade nada (ver println anterior)

        c[3].add(s[0]).add(s[1]).add(s[2]); // OK todo

        System.out.println("Comando 1: " + c[1]);
        System.out.println("Comando 2: " + c[2]);

        System.out.println( c[2].getFuerza() ); // no suma dos veces la fuerza de s[0] y s[2]
        System.out.println( c[2].getSoldados() ); // no salen soldados repetidos
    }
}
```

Salida esperada:

```
true
Comando 1: C1:[S(12), S(10), C3:[S(13), S(10), S(11), S(12)]]
Comando 2: C2:[C1:[S(12), S(10), C3:[S(13), S(10), S(11), S(12)]]]
46
[S(12), S(10), S(13), S(11)]
```

EVALUACION NO CONTINUA

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 3. (3 puntos)

Se quiere realizar una clase genérica *Almacen* que almacene objetos de tipo *Registro*. La clase *Registro* será también genérica, y almacena valores que sean *comparables*. Un *Almacen* se inicializa en el constructor con un criterio primario de comparación, a través de un objeto *Comparator*. El *Almacen* debe almacenar los registros ordenados de acuerdo al criterio primario, y en caso de igualdad por el orden natural de los objetos almacenados en el registro (que son comparables). El *Almacen* no debe admitir registros repetidos, y debe lanzar una excepción si se intenta añadir un repetido.

Se pide:

- Implementar las clases necesarias, y completar el siguiente programa produzca la salida de más abajo:
- Describir usando Java, qué cambios habría que hacer en el programa del apartado (a) para que *Almacen* admita una secuencia de *Comparators* en el constructor en vez de uno solo. El *Almacen* debe almacenar los registros ordenados de acuerdo al primer criterio, en caso de empate, se usa el segundo criterio, y así sucesivamente hasta el último criterio de la secuencia, y si también resulta en igualdad se usaría el orden natural de los objetos almacenados en el registro.

```
// Un comparador de cadenas por longitud
class ComparaLongitud implements Comparator<String> {
    @Override public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
}

public class Main {
    public static void main(String[] args) {
        // 1er criterio: orden por longitud, 2º criterio: orden natural (alfabético)
        Almacen<String> alm = new Almacen<String>(new ComparaLongitud());
        try {
            alm.añade(new Registro<String>("Un registro"));
            alm.añade(new Registro<String>("Tres reg"));
            alm.añade(new Registro<String>("Dos regs")); // igual longi. -> orden natural
            alm.añade(new Registro<String>("Un registro")); // este registro está repetido

        } catch (RegistroRepetido e) {
            System.out.println("Error "+e);
        }
        System.out.println("Almacen = "+alm);
    }
}
```

Salida esperada:

Error Registro repetido: Un registro

Almacen = Registros: [Dos regs, Tres reg, Un registro]

EVALUACION NO CONTINUA

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 4. (1,5 puntos)

Los objetos de la clase `ApplicableMap` son mapas similares a los `java.util.HashMap` pero además pueden utilizarse con parámetro en algunas operaciones sobre streams como `map` y `filter`, respetando sus definiciones dadas en el API de `java.util.stream.Stream` (ver comentario en el código ejemplo de abajo).

Se pide: implementar la clase `ApplicableMap` para que el método `main` produzca la salida indicada abajo.

```
/* FRAGMENTO DEL API DE java.util.stream.Stream
Devuelve un stream con los resultados de aplicar a los elementos del
stream this la función dada como parámetro.
<R> Stream<R>    map(Function<? super T,? extends R> mapper)

Devuelve un stream con los elementos del stream this que
cumplen el predicado dado como parámetro.
<R> Stream<R>    filter(Predicate<? super T> mapper)
*/
public class Main {
    public static void main(String[] args) {
        ApplicableMap<Integer,Double> am = new ApplicableMap<>();
        am.put(1, 0.5); am.put(2, 1.0); am.put(4, 2.0); am.put(5, 2.5);
        System.out.println(am);

        Stream<Integer> stream = Stream.of(1,5,19,2,72);
        System.out.println(
            stream.filter(am) // no pasan el 19 ni el 72, por no estar en am
                .map(am)      // cada número se convierte a su valor asociado en am
                .collect(Collectors.toList())
        );
    }
}
```

Salida esperada:

```
{1=0.5, 2=1.0, 4=2.0, 5=2.5}
[0.5, 2.5, 1.0]
```