EVALUACION NO CONTINUA

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 1. (3 puntos) (Solo en NO Continua)

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:

(a) 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.

(b) Para las siguientes operaciones, indica la clase más apropiada para contenerla, el nombre del método, argumentos, y tipo de resultado.

1. Subscribirse a los cambios de una página web
2. Dar de baja un mecanismo de notificación de un cliente

(c) Implementa, en pseudocódigo o código java, el método que sirve para avisar, cuando cambia una página, a los interesados en dicho cambio.

(d) ¿Qué patrón o patrones de diseño has utilizado?

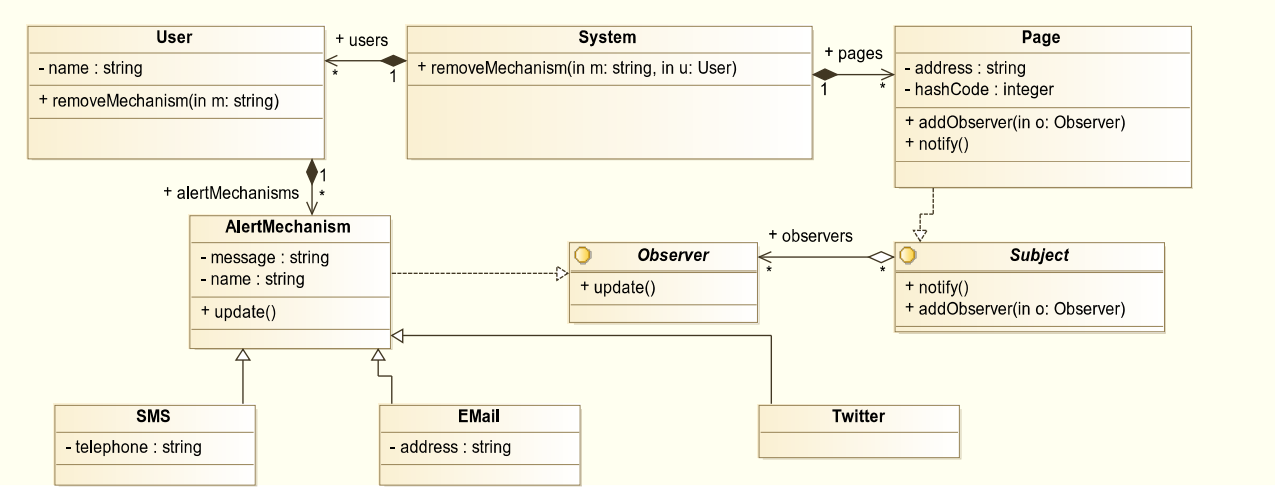
Nota: No es necesario detallar como es el proceso de descarga de una página web, por ejemplo se puede asumir que existe una clase URL que ya tiene implementada dicha funcionalidad. De forma similar, tampoco es necesario detallar el sistema de envío de notificaciones, será suficiente con un método send()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

A)



B)
1) Page::addObserver(o:Observer):void
2) System::removeMechanism(m:string, u:User):void y User::removeMechanism(m:string):void

C) (Solo en Evaluación NO continua)

```
void Page::notify(){
    for (Observer o:observers)
        o.update();
 }
```

D) Patrón Observer
Nota: Es el patrón Observer, aunque no existan las interfaces Observer o Subject.

Apartado 2. (2.5 puntos). (NO Continua)

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)]
```

Respuesta (b):

Se utiliza el patrón de diseño Composite, de forma que: nuestro interfaz Combatiente corresponde al Component del patrón, nuestra clase Soldado es la clase Leaf del patrón, y nuestra clase Comando es la clase Composite del patrón.

```java
package ej2final2015.NO.CONTINUA;

public interface Combatiente {
    Combatiente add(Combatiente c);   // debe devolver Combatiente
    int getFuerza();                  // ver main
    Set<Soldado> getSoldados();       // ver main
    boolean contiene(Combatiente c);  // ver main
}

public class  Soldado implements Combatiente {
    private int fuerza; // debe ser prívate
    public Soldado(int fuerza) { this.fuerza = fuerza; }
    @Override public int getFuerza() { return this.fuerza; }

    @Override public Combatiente add(Combatiente c) { return this; }
    @Override public boolean contiene(Combatiente c) { return false; }
    @Override
    public Set<Soldado> getSoldados() {
        Set<Soldado> soldados = new HashSet<>();
        soldados.add(this);
        return soldados;
    }
}

public class Comando implements Combatiente {
    // para evitar repetidos usamos un Set
    // es mal diseño separar miembros soldados y miembros comandos
    private Set<Combatiente> miembros = new LinkedHashSet<>();

    private String nombre;  // debe ser prívate
    public Comando(String nombre) { this.nombre = nombre; }
    public String getNombre() { return this.nombre; }

    public Collection<Combatiente> getMiembros() {
       // return this.miembros; violaría la protección del objeto
       return Collections.unmodifiableSet(miembros);
    }
    @Override
    public Combatiente add(Combatiente c) {
        if ((c == null )) return this;  // ver enunciado
        if (this.contiene(c) || c.contiene(this)) return this; // ver enunciado
        // if (this.miembros.contains(c)) return this; es innecesario porque
        this.miembros.add(c); // al ser un Set evita repetidos automáticamente
        return this;
    }
    @Override
    public boolean contiene(Combatiente c) { profundiza recursivamente
        if (this == c) return true;  // la hoja soldado se contiene a si misma
        for (Combatiente miembro : this.miembros) // recorrer miembros a ver si
            if (miembro.contiene(c)) return true;  // lo contiene algun miembro
        return false;
    }
    @Override
    public Set<Soldado> getSoldados() { // recursivamente, sin duplicados
        Set<Soldado> soldados = new HashSet<>();
        for (Combatiente c : this.miembros) soldados.addAll( c.getSoldados() );
        return soldados;
    }
    @Override
    public int getFuerza() { // sumar todos los soldados, sin duplicados
        int sum = 0;
        for (Soldado s : this.getSoldados()) sum += s.getFuerza();
        return sum;
    }
}
```

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 parametriza 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:

a) Implementar las clases necesarias, y completar el siguiente programa produzca la salida de más abajo:

b) Indicar 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 long -> 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]
```

Solución a): Ver la solución de continua

Solución b):

```
import java.util.*;

class RegistroRepetido extends Exception {
	private Registro<?> r;

	public RegistroRepetido(Registro<?> r) { this.r = r; }
	public String toString() { return "Registro repetido: "+this.r; }
}

class Registro<T extends Comparable<T>> implements Comparable<Registro<T>>{
	private T data;

	public Registro(T d) { this.data = d; }

	public T getData() { return data; }

	@Override public String toString() { return this.data.toString(); }

	@Override public int compareTo(Registro<T> a) {
		return this.data.compareTo(a.data);
	}

}

class Almacen<T extends Comparable<T>> {
	private SortedSet<Registro<T>> registros;

	public Almacen(Comparator<T> ...ops) {
		this.registros = new TreeSet<Registro<T>>( new MultiComparator(ops) );
	}
	public void añade(Registro<T> r) throws RegistroRepetido {
		if (!this.registros.add(r)) throw new RegistroRepetido(r);
	}
	@Override public String toString() {
		return "Registros: "+this.registros;
	}

	class MultiComparator implements Comparator<Registro<T>> {
		private List<Comparator<T>> compar ;

		public MultiComparator(Comparator<T> ...ops) {
			this.compar = new ArrayList<Comparator<T>>(Arrays.asList(ops));
		}

		@Override
		public int compare(Registro<T> o1, Registro<T> o2) {
			for (Comparator<T> c: this.compar) {
				int result = c.compare(o1.getData(), o2.getData());
				if (result != 0) return result;
			}
			return o1.compareTo(o2);
		}

	}

}
```

EVALUACION CONTINUA

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 4.

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 dada 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.Sream
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]
```

Solución:

```
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import java.util.HashMap;

class ApplicableMap<K,V> extends HashMap<K,V> implements Function<K,V>, Predicate<K> {
      @Override
      public V apply(K in) {
            V value = this.get(in);
            return value;
      }

      @Override
      public boolean test(K in) {
            return this.containsKey(in);
      }
}
```