Hoja 4 de Ejercicios

Patrones de Diseño

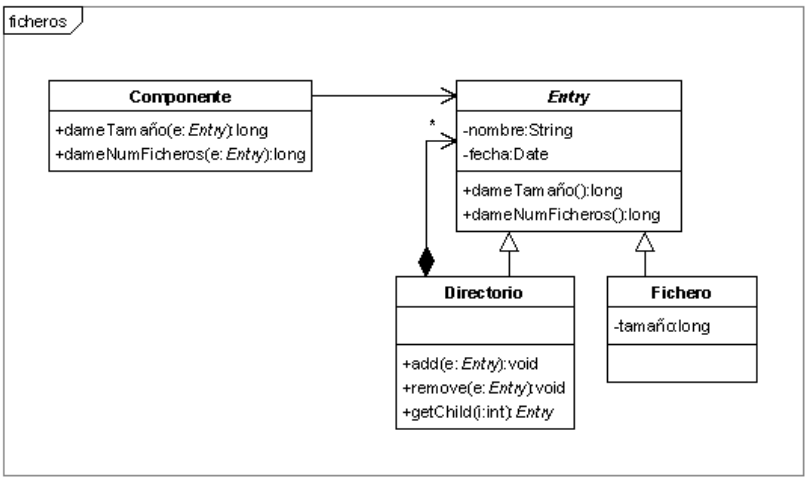
Inicio:
Duración:
Entrega:

1) Queremos desarrollar un componente para la gestión de ficheros que permita agruparlos en directorios y subdirectorios. Ficheros, directorios y subdirectorios tienen un nombre y una fecha de última modificación. Además, los ficheros ocupan un número de KBs. El componente debe permitir calcular el tamaño de directorios, subdirectorios y ficheros, así como calcular cuántos ficheros contiene un directorio o un subdirectorio. El tamaño de un directorio es igual al tamaño de los ficheros y subdirectorios que contiene. Lo mismo se aplica para el número de ficheros.

   Nos han comunicado que el componente se integrará en diversos sistemas, por lo que es requisito indispensable que sea seguro y no permita realizar operaciones incorrectas.

   Utilizando patrones de diseño, especifica el diagrama de clases del componente. Comenta cómo has reflejado en tu diagrama el requisito de seguridad.

SOLUCIÓN:

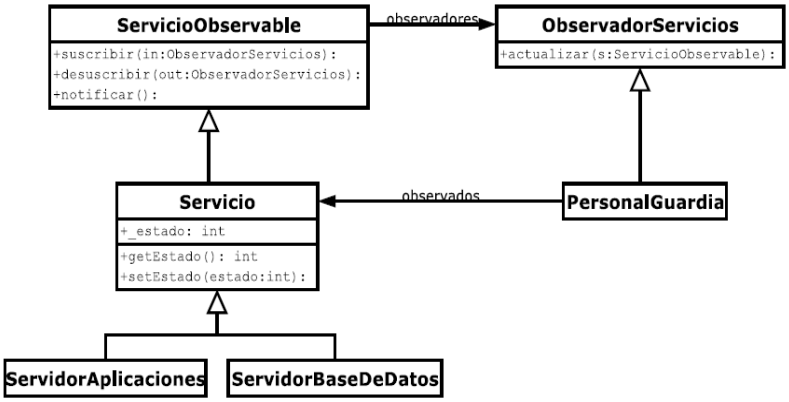


El requisito de seguridad se puede tratar de dos formas:
1. Los métodos add, remove y getChild se definen en la clase Directorio, de manera que no pueden invocarse sobre la clase Fichero. De ese modo se evita que un fichero incluya otros ficheros o directorios. El problema es que el tratamiento de directorios y ficheros desde la clase Componente no sería homogéneo.
2. Los métodos add, remove y getChild se definen en la clase Entry con un comportamiento por defecto (getChild devuelve null; add y remove lanzan excepción). Luego se sobrescriben en la clase Directorio para implementar el comportamiento adecuado. Así se tiene un tratamiento transparente de Ficheros y Componentes.

2) El departamento de producción de una empresa informática tiene un sistema que asigna guardias a sus empleados sobre los servicios que ofrece. Existen distintos servicios (ej. BBDD, servidores) que tienen que monitorizarse, por lo que cuando uno de ellos se activa, las personas encargadas de la guardia de ese servicio reciben una notificación. Se pide:

a) Un diagrama de clases de una solución que facilite la notificación por parte de cada servicio al personal de guardia encargado de monitorizarlo. Suponed que el personal de guardia puede monitorizar más de un servicio.

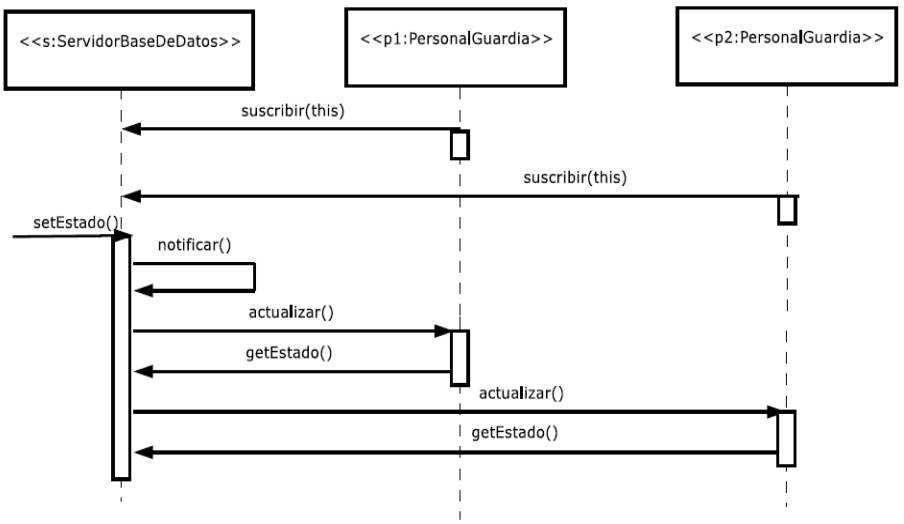
SOLUCIÓN:



Dado que cada personal de guardia puede monitorizar varios servicios, el sistema de actualización pasa información sobre el servicio concreto que ha modificado su estado utilizando para ello el parámetro de tipo "ServicioObservable" del método "actualizar".

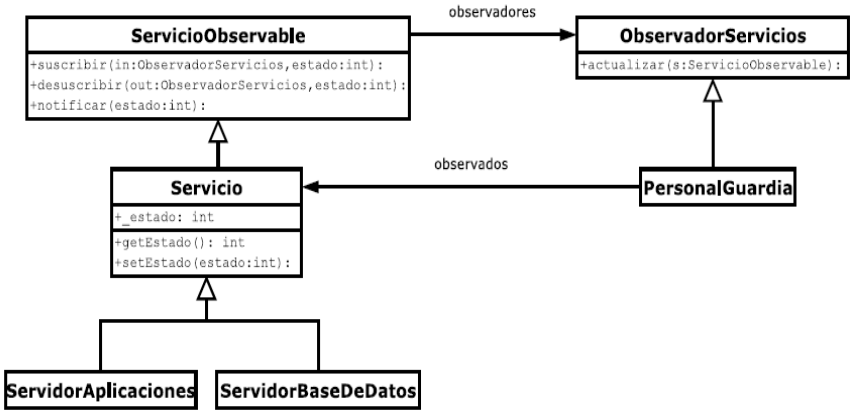
b) El diagrama de secuencia de dos miembros del personal que se suscriben a un servicio, y se enteran de la activación de dicho servicio.

SOLUCIÓN:



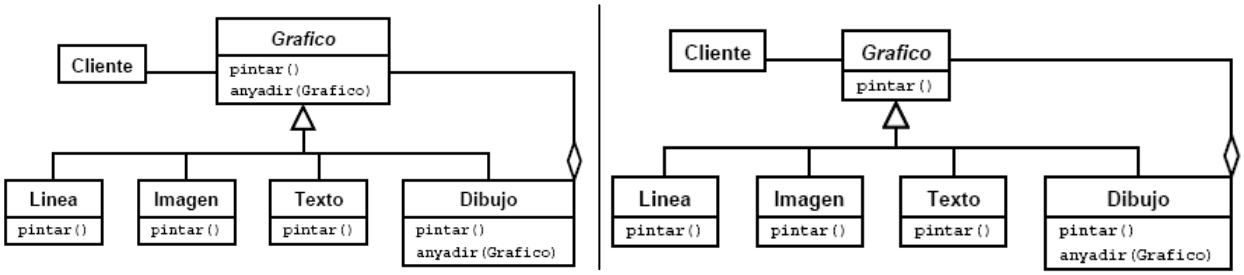
c) Modificar el diagrama de clases para que cada miembro del personal sólo reciba notificaciones sobre determinados tipos de eventos por los que previamente haya mostrado interés (ej. parada, activación...).

SOLUCIÓN:



La solución consiste en pasar a los métodos de la clase "ServicioObservable" el tipo de evento en el que se está interesado (parámetro "estado").

3) Partiendo de los diseños que muestran las siguientes figuras, se pide completar en cada caso el código de los métodos "anyadir" y "pintar" de la clase "Cliente". Se podrán hacer cuantas modificaciones en el diseño se consideren necesarias.



SOLUCIÓN:

Solución al diagrama de la izquierda:

```
class Cliente{
  private Grafico _grafico;
  void anyadir (Grafico nuevo){
    _grafico.anyadir(nuevo);
  }
  void pintar(){
    _grafico.pintar();
  }
}
```

Solución al diagrama de la derecha:

```
class Cliente{
  private Grafico _grafico;
  void anyadir (Grafico nuevo) throws ErrorOperacionAnyadirEnHojaException{
    Dibujo compuesto = _grafico.getComposite();
    if ( compuesto != null )
      compuesto.anyadir(nuevo);
    else
      throw new ErrorOperacionAnyadirEnHojaException();
  }
  void pintar(){
    _grafico.pintar();
```

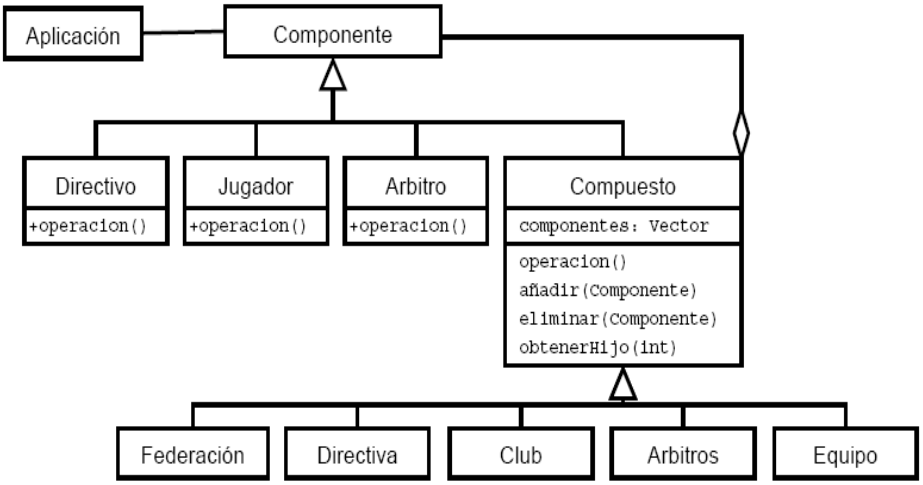
```
    }
}
```

4) La Federación Española de Canicas tiene un fichero con todas las personas que participan en sus actividades, como directivos, árbitros y jugadores, pero tiene todas las fichas mezcladas en el mismo cajón. En un momento de lucidez deciden informatizar el sistema organizando a todos los participantes en equipos arbitrales, clubs, equipos de jugadores y equipos directivos.

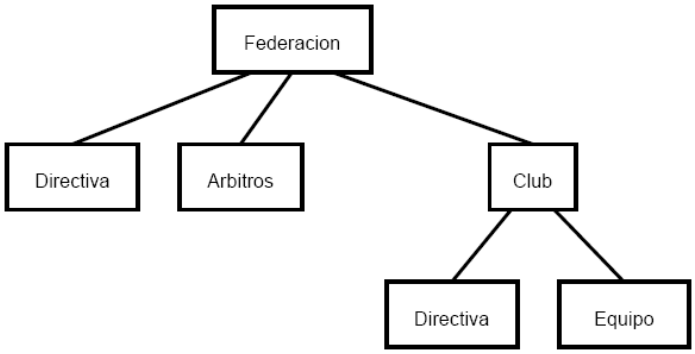
a) Utilizando alguno de los patrones conocidos, definir el diagrama UML que permita gestionar gestionar de manera uniforme las fichas de todos los componentes de la reestructuración. Utiliza como punto de partida las clases de la siguiente figura.

SOLUCIÓN:

Para el tratamiento uniforme de la estructura usamos el patrón Composite. La jerarquía de datos resultante es la siguiente: La Federación es un elemento compuesto que puede contener otros elementos compuestos, como equipos directivos, equipos arbitrales y clubs. A su vez, un club puede contener un equipo directivo y equipos de jugadores. Cualquier clase compuesta puede, además, contener elementos Hoja (personas en forma de jugadores, árbitros o directivos).



b) Escribir en Java una manera de controlar que la jerarquía de la siguiente figura es la única posible y que, por ejemplo, un equipo de jugadores no puede contener un club.



SOLUCIÓN:

Podemos añadir a cada clase un atributo que indique el nivel en la jerarquía, y con la funcionalidad mínima para que no permita añadir a su colección de objetos un objeto que tenga un nivel superior en la jerarquía. Dando a Federación el nivel más alto, el código para las clases compuestas (Compuesto, Federación, Directiva, Club, Árbitros y Equipo) podría quedar así:

```
public abstract class Compuesto{
   private int NIVEL = 3;
   int obtenerNivel() { return NIVEL; }
   // Comprobar si se puede añadir un compuesto a la colección
   boolean puedeAniadirse(Compuesto otro) {
      return obtenerNivel() < otro.obtenerNivel();
   }
}

public class Federacion extends Compuesto {
   private int NIVEL = 1; // nivel más alto
}

public class Club extends Compuesto {
   private int NIVEL = 2; // segundo nivel
}

public class Arbitros extends Compuesto {
   private int NIVEL = 2; // segundo nivel
}

...
```

Otra solución sería construir una clave que controle el nivel de todas las clases del sistema. Utilizando una tabla hash se puede asignar a cada clase compuesta un vector de las clases susceptibles de ser añadidas a la anterior, de modo que se cumpla la jerarquía de la figura. La función de control utilizaría la clase padre como clave de la tabla hash para obtener las posibles clases hija, comparándolas con la que se pretende añadir.

```
class Nivel{
   private HashMap _niveles;
   private Nivel() {
      _niveles = new HashMap();
      Class[] hijosFederacion = {
              Club.getClass(),
              Arbitros.getClass(),
              Directiva.getClass() };
      Class[] hijosClub = {
              Directiva.getClass(),
              Equipo.getClass() };
      _niveles.put(Federacion.getClass(), hijosFederacion);
      _niveles.put(Club.getClass(), hijosClub);
      _niveles.put(Arbitros.getClass(),null);
      _niveles.put(Directiva.getClass(),null);
      _niveles.put(Equipo.getClass(),null);
```

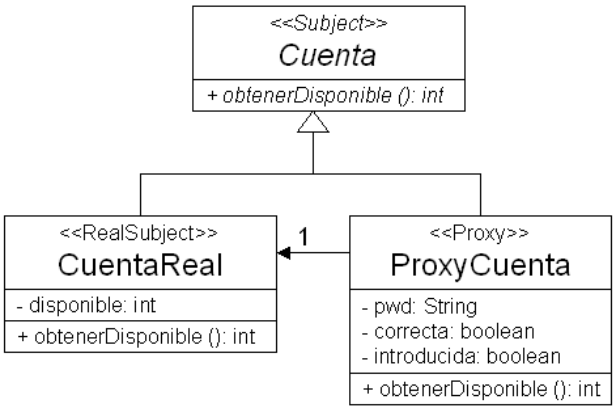
```
   }
   public boolean puedeAniadirse(Compuesto padre, Compuesto hijo){
      boolean ok = false;
      int indice = 0;
      Class[] hijos = (Class[]) _niveles.get(padre.getClass());
      while((indice < hijos.length) && (!ok))
         if (hijos[indice] == hijo.getClass())
            ok = true;

      return ok;
   }
}
```

5) En una aplicación bancaria, las cuentas de los clientes se han implementado como una clase "Cuenta" que define un atributo "disponible" de tipo int para almacenar el saldo disponible en la cuenta, y un método "obtenerDisponible" que devuelve el valor del atributo. Se quiere extender la aplicación para que sólo se pueda acceder a una cuenta si previamente se ha introducido correctamente cierta clave. La clave se debe introducir por teclado y, si es correcta, ya no se debe volver a pedir.

a) Utilizando patrones de diseño, especifica el diagrama de clases que modela lo expuesto en el enunciado. Señala el rol de las clases que participan en algún patrón.

SOLUCIÓN:



b) Implementa en Java el diagrama de clases definido en el apartado a), así como un ejemplo de uso por parte de una clase cliente.

SOLUCIÓN:

```
public interface Cuenta {
   int obtenerDisponible ();
}

//-----------------------------------------------------
public class CuentaReal implements Cuenta {
   private int disponible;
   public CuentaReal (int d) {
      disponible = d;
   }
   public int obtenerDisponible() {
      return disponible;
   }
}

//-----------------------------------------------------
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

```
class ProxyCuenta implements Cuenta {
   private CuentaReal cuenta;
   private String clave;
   private boolean correcta, intro;
   public ProxyCuenta (String psswd, CuentaReal c) {
      clave = psswd;
      cuenta = c;
      intro = false;
      correcta = false;
   }
   public int obtenerDisponible() {
      try {
         if (correcta) return cuenta.obtenerDisponible();
         else if (!intro) {
            intro = true;
            System.out.println("Introduce la clave:");
            BufferedReader entrada = new BufferedReader(
               new InputStreamReader(System.in));
            String claveIntroducida  = entrada.readLine();
            if (claveIntroducida.equals(clave)) {
               correcta = true;
               return cuenta.obtenerDisponible();
            }
            else {
               correcta = false;
              System.out.println("clave incorrecta");
               return -1;
           }
         }
         return -1;
      }
      catch (Exception e) { return -1; }
   }
}

//-----------------------------------------------------
public class Main {
   public static void main (String args[]) {
      CuentaReal c = new CuentaReal (100);
      ProxyCuenta pc = new ProxyCuenta("clave1", c);
      int d = pc.obtenerDisponible();
      System.out.println("Disponible = "+d);
      System.out.println("Disponible = "+pc.obtenerDisponible());
   }
}
```