Boletín de ejercicios

1. Sean las siguientes dos clases Java

```
abstract class B {
    abstract public void metodo1 ();
}

abstract class A {
    public void necesitoUnB() {
           B ob = ???      //crear un B
           ob.metodo1();
    }
}
```

La clase `A` tiene el método `necesitoUnB()` que debe crear un objeto de alguna subclase de `B` y asignarlo a la variable `ob`, pero `A` no debe contener referencias explícitas a las subclases de `B`. Completa el código de este método aplicando alguno de los patrones vistos en clase y explica cómo se consigue crear la instancia de la subclase. Primero debes indicar una solución sin metaclases y luego otra con metaclases.

Solución 1.

Aplicaríamos el patrón *Método Factoría* sin metaclases

```
class B1 extends B {
    public void metodo1 ();
}
abstract class A {
    abstract public B crearUnB();
    public void necesitoUnB() {
           B ob = crearUnB()     //crear un B
           ob.metodo1();
    }
}
class A1 extends A {
    public B crearUnB() {
           return new B1();
    }
}
```

Solución 2.

Aplicaríamos el patrón *Método Factoría* con una clase *Factoría*.

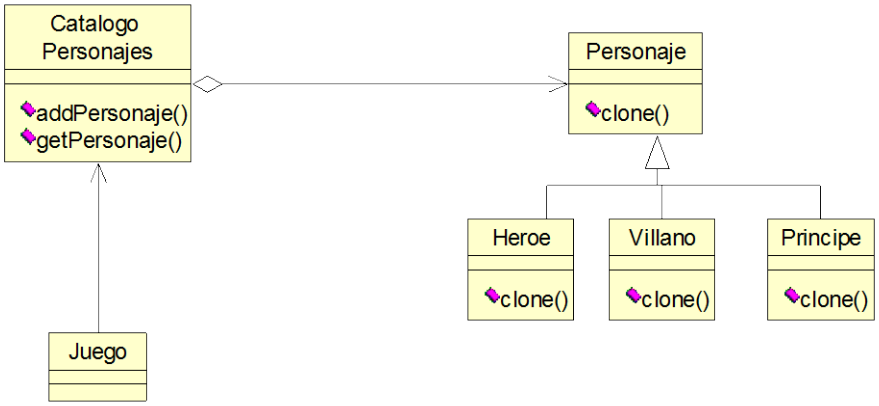
```
class B1 extends B {
    public void metodo1 ();
}


abstract class A {
    abstract public B crearUnB();
```

```
        public void necesitoUnB() {
               B ob = Factoria.getInstancia().crearUnB()
               ob.metodo1();
        }
    }
    class Factoría {
        // sería un Singleton
        public B crearUnB(String nombre) {
               return (B)Class.forName(nombre).newInstance()
        }
    }
```

2. Supuesto que se esté desarrollando un juego interactivo que permite al usuario interactuar con personajes que juegan ciertos roles. Se desea incorporar al juego una facilidad para crear nuevos personajes que se añaden al conjunto de personajes predefinidos. En el juego, todos los personajes serán instancias de un pequeño conjunto de clases tales como `Heroe`, `Villano`, `Principe` o `Monstruo`. Cada clase tiene una serie de atributos como `nombre`, `imagen`, `altura`, `peso`, `inteligencia`, `habilidades`, etc. y según sus valores, una instancia de la clase representa a un personaje u otro, por ejemplo podemos tener los personajes príncipe bobo o un príncipe listo, o monstruo bueno o monstruo malo. Diseña una solución basada en patrones que permita al usuario crear nuevos personajes y seleccionar para cada sesión del juego personajes de una colección de personajes creados. Escribe el código de los métodos del catálogo de personajes que permiten añadir un nuevo personaje a la colección y seleccionar un personaje para jugar.

Solución

Se utiliza el patrón Prototype para mantener un catálogo de instancias prototípicas que se crean a partir de los personajes predefinidos. El catálogo puede implementarse como un singleton.

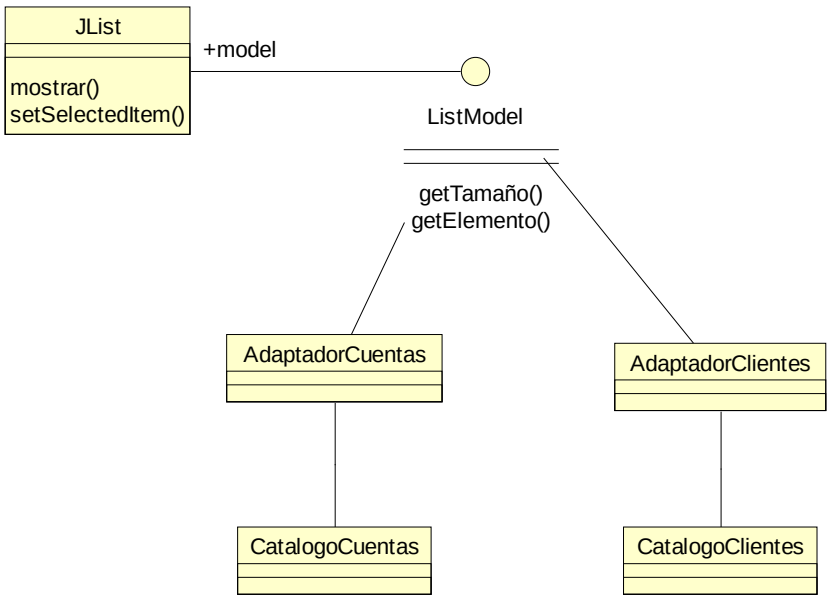



```
class CatalogoPersonajes {
   public static CatalogoPersonajes getInstancia() {
      if (unicaInstancia == null)
                  unicaInstancia = new CatalogoPersonajes();
      return unicaInstancia;
   }
   private static CatalogoPersonajes unicaInstancia = null;
   private CatalogoPersonajes() {personajes= new HashTable();}
```

```
    private HashTable personajes;
    public void addPersonaje(String nom, Personaje p) {
            personajes.put(nom, p);
       }
    public Personaje getPersonaje(String nombre) {
            Personaje p = (Personaje) personajes.get(nombre);
            return (Personaje) p.clone();
       }
}
```

3. La librería Swing de Java incluye la clase `JList` para presentar una lista de objetos (el texto que se visualiza es determinado por el método `toString` de la clase de los objetos) y permitir al usuario seleccionar uno de ellos. El diseño de esta clase es un ejemplo de utilización del patrón `Adaptador` para conseguir una clase reutilizable; se consigue que `JList` sea independiente de la fuente (o modelo) de datos. Muestra mediante un diagrama de clases el diseño que permitiría a una clase como `JList` visualizar diferentes listas de items, como por ejemplo un catálogo de clientes (por ejemplo, se mostraría su NIF) o un catálogo de cuentas (por ejemplo, se mostraría el código de cuenta). Para cada clase o interfaz del diagrama indica los métodos y atributos que son significativos. Nótese que la clase `JList` necesita disponer de información como el número de ítems a visualizar y el objeto seleccionado. Los adaptadores almacenan la lista que se visualiza como una instancia de `Vector`.

Solución.



4. Sea un conjunto de clases que permiten la creación y envío de mensajes de correo electrónico y que entre otras incluye clases que representan el cuerpo del mensaje, los anexos, la cabecera, el mensaje, la firma digital y una clase encargada de enviar el mensaje. El código cliente debe interactuar con instancias de todas estas clases para el manejo de los mensajes, por lo que debe conocer en qué orden se crean esas instancias; cómo colaboran esas instancias para obtener la funcionalidad deseada y cuáles son las

relaciones entre las clases. Idea una solución basada en algún patrón tal que se reduzcan las dependencias del código cliente con esas clases y se reduzca la complejidad de dicho código cliente para crear y enviar mensajes. Dibuja el diagrama de clases que refleje la solución e indica qué patrón has utilizado.

Solución.

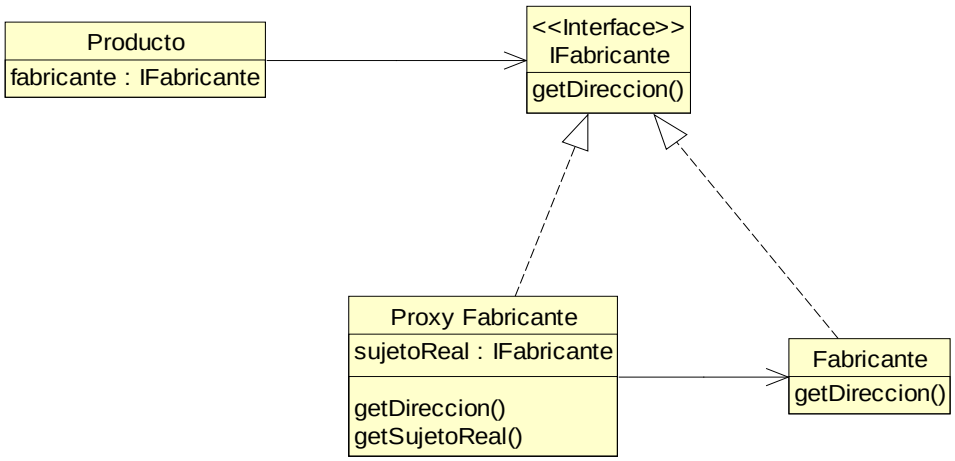
Habría que utilizar el patrón *Fachada*. Se crea una clase `FachadaEmail` que proporciona la interfaz necesaria para acceder al subsistema formado por clases como `Header`, `Message`, `DigitalSignature`, `SendMessage`, etc.

5. Explica cómo se utiliza el patrón *Proxy* para implementar una materialización perezosa de instancias almacenadas en una base de datos.

Solución.

(Ver fotocopia entregada en clase con diagrama de clases que mostraba un "proxy virtual Fabricante", extraída del libro de Larman, página 522)

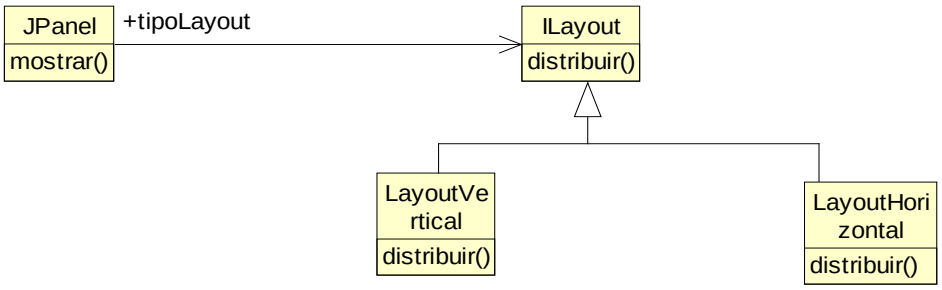
Cuando se carga un objeto desde la base de datos, por ejemplo una instancia de `Producto`, no se cargan todos sus campos, sino que para algún campo, normalmente por razones de rendimiento, se difiere la carga hasta que es necesario en el momento que se referencia, para lo cual se aplica el patrón *Proxy*.



6. Supuesto que se está construyendo una librería de clases para representar componentes GUI, se ha decidido que en vez de que un programador defina la posición de los componentes GUI (*Button*, *List*, *Dialog*,..) sobre una ventana, se incluyan manejadores de disposición de componentes (*layout manager*), cada uno de los cuales distribuye un conjunto dado de componentes gráficos de acuerdo a algún esquema de distribución: horizontalmente, verticalmente, en varias filas, en forma de una matriz, etcétera. Debe ser posible cambiar en tiempo de ejecución la distribución elegida inicialmente. Supuesto que la clase `JPanel` es la que representa a un contenedor de componentes gráficos, diseña una solución para introducir en la librería los manejadores de disposición. Dibuja el diagrama de clases que refleje la solución e indica qué patrón has utilizado.

Solución.

Se utilizaría el patrón *Estrategia*



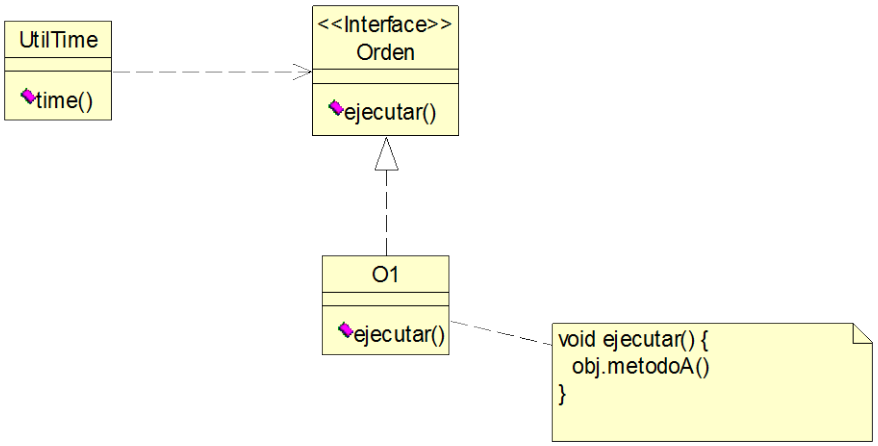
7. Se desea escribir una clase `UtilTime`, que no es abstracta, que incluya un método estático `time` que mide el tiempo que tarda un método cualquiera en ejecutarse. Parte del código de dicho método sería:

```
public static long time ( “parámetros”) {
    long t1 = System.currentTimeMillis();
     // falta aquí el código apropiado
    long t2 = System.currentTimeMillis();
    return t2 - t1
}
```

Diseña una solución basada en alguno de los patrones de diseño y completa el método: parámetros y código en la posición del comentario. Escribe un código cliente con un ejemplo de utilización del método `time()`.

Solución.

Utilizar el patrón *Command* que encapsula un mensaje




```
public static long time ( Orden o) {
      long t1 = System.currentTimeMillis();
      o.ejecutar()
```

```
      long t2 = System.currentTimeMillis();
      return t2 - t1
}
```

Código cliente: `Util.time (new O1() )`

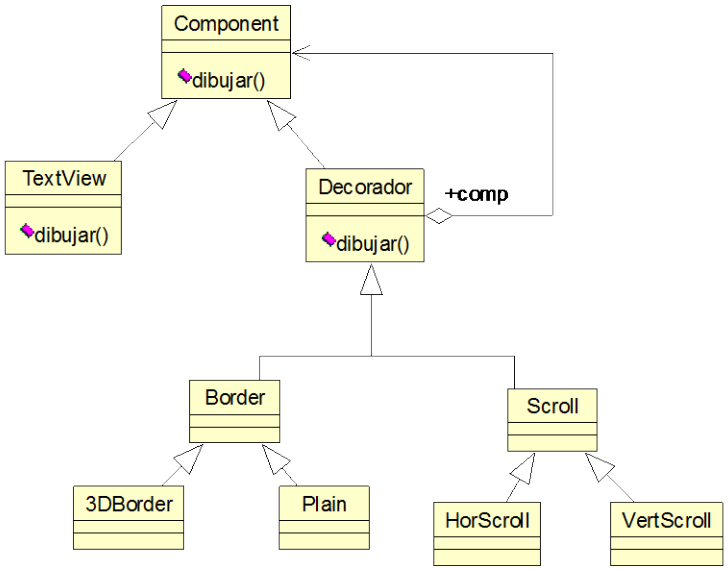
8. Sea una clase `TextView` que representa un componente GUI ventana de texto que es subclase de una clase `Component` raíz de la jerarquía de clases que representan componentes GUI. Queremos definir ventanas de texto con diferentes tipos de bordes (*Plain*, *3D*, *Fancy*) y barras de desplazamiento (horizontal, vertical). La clase `TextView` tiene un método dibujar entre otros

```
class TextView {
    public void mostrar() {
      // código para dibujar el objeto TextView }
}
```

Utiliza este ejemplo para:

a) Mostrar el diagrama de clases que resulta de aplicar el patrón *Decorador*.
b) Escribir el código de una clase decoradora: atributos, constructor y método mostrar, y señalar qué cambios es preciso realizar sobre la clase `TextView` al aplicar el patrón *Decorador*.
c) Describir cómo el patrón *Strategy* es una alternativa al patrón *Decorador*, mostrando el diagrama de clases y señalando los cambios que realizarías sobre el código dado de la clase `TextView`.
d) Escribir un código cliente Java que crea un objeto `TextView` con un borde 3D y una barra de desplazamiento horizontal, para el caso que se usan decoradores y para el caso que se usan estrategias.

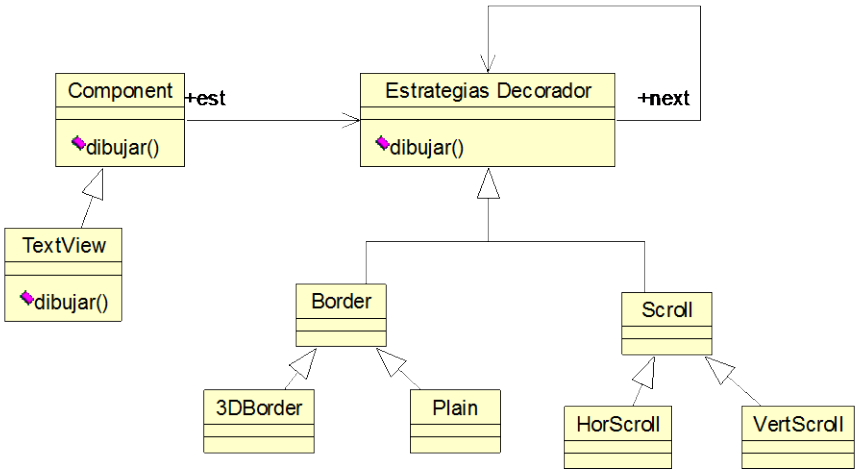
a)



b)
```
abstract class Decorator extends Component {
     Component comp;
     public Decorator(Component c) { comp = c;}
     public dibujar() {comp.dibujar();}
}

class Plain extends Border {
     //atributos y métodos adicionales
         public void dibujar() {
               super.dibujar();
               dibujarBordePlain();}
...
}
```

Añadir decoradores no requiere modificar la jerarquía de componentes visuales

c) Conviene usar el patrón *Estrategia* en vez de *Decorador* si la clase `Component` está muy "cargada".



El método dibujar de `TextView` debe incluir la invocación `est.dibujar()` para obtener la funcionalidad añadida por los decoradores.

Ahora los objetos decorados (`TextView`) tienen una referencia a los "decoradores", al revés de lo que sucede cuando usamos el patrón *Decorador*.

d) Con Decorador:

```
Component c = new 3DBorder (new VertScroll (new TextView))
```

Con Estrategia:

```
TextView tv = new TextView ();
tv.setEstrategia(new VertScroll(new 3DBorder()))
```

o si un Component contiene las estrategias como una agregación:

```
TextView tv = new TextView ();
tv.addEstrategia(new VertScroll());
tv.addEstrategia(new 3DBorder());
```

9. Supuesto que estamos desarrollando una aplicación financiera, se ha decidido emplear el patrón *Composite*, puesto que existen diferentes valores elementales (acciones, bonos, futuros, fondos,..) y valores compuestos (carteras de valores, cuentas,..). Describe cómo utilizar el patrón *Visitor* para realizar diferentes operaciones sobre un valor compuesto, tales como calcular su precio u obtener información fiscal anual ¿Qué beneficios se obtienen al aplicar el patrón *Visitor*?

La estructura de datos es definida por el patrón *Composite* y se define una jerarquía de *visitors* para aplicar operaciones sobre ella.