Ejercicios: Patrones de diseño

Semana del 4 de mayo

1) La clase "Printer" simula una impresora, y tiene tres métodos que permiten a un usuario iniciar un trabajo, imprimir páginas y terminarlo. Utilizando ***patrones de diseño***, **y sin modificar la clase Printer**, completa programa para que sea posible crear impresoras de uso exclusivo. Estas impresoras no permiten imprimir a un usuario si ya hay otro que la está utilizando (es decir, si ya ha llamado al método "startJob"). La liberación de la impresora se produce cuando el usuario llama al método "endJob". ¿Qué patrón (o patrones) de diseño has usado?

```
class PrinterUtil {
	public static IPrinter makeExclusive(Printer p) {

		…… // Completar

	}
}

interface IPrinter {  //Completar
	……

}

class Printer implements IPrinter{
	public void startJob(String header, String user) {
		System.out.println("User "+user+" Printing Header:: "+header);
	}
	public void printPage(String page, String user) {
		System.out.println("User "+user+" Printing Page:: "+page);
	}
	public void endJob(String footer, String user) {
		System.out.println("User "+user+" Printing Footer:: "+footer);
	}
}

// Completar
…
…

public class Main {
	public static void main(String[] args) {
		Printer p = new Printer();
		IPrinter ap = PrinterUtil.makeExclusive(p);

		ap.startJob("inicio", "juan");
		ap.startJob("otro inicio", "eduardo"); // no imprime, la tiene juan
		ap.printPage("página 1", "juan");
		ap.printPage("otra pagina 1", "eduardo"); // no imprime, la tiene juan
		ap.endJob("final", "juan");               // juan libera la impresora
		ap.startJob("a ver si esta vez...", "eduardo"); // ahora sí
		ap.endJob("fin", "eduardo");
	}
}
```

```
Salida esperada:

User juan Printing Header:: inicio
User juan Printing Page:: página 1
User juan Printing Footer:: final
User eduardo Printing Header:: a ver si esta vez...
User eduardo Printing Footer:: fin
```

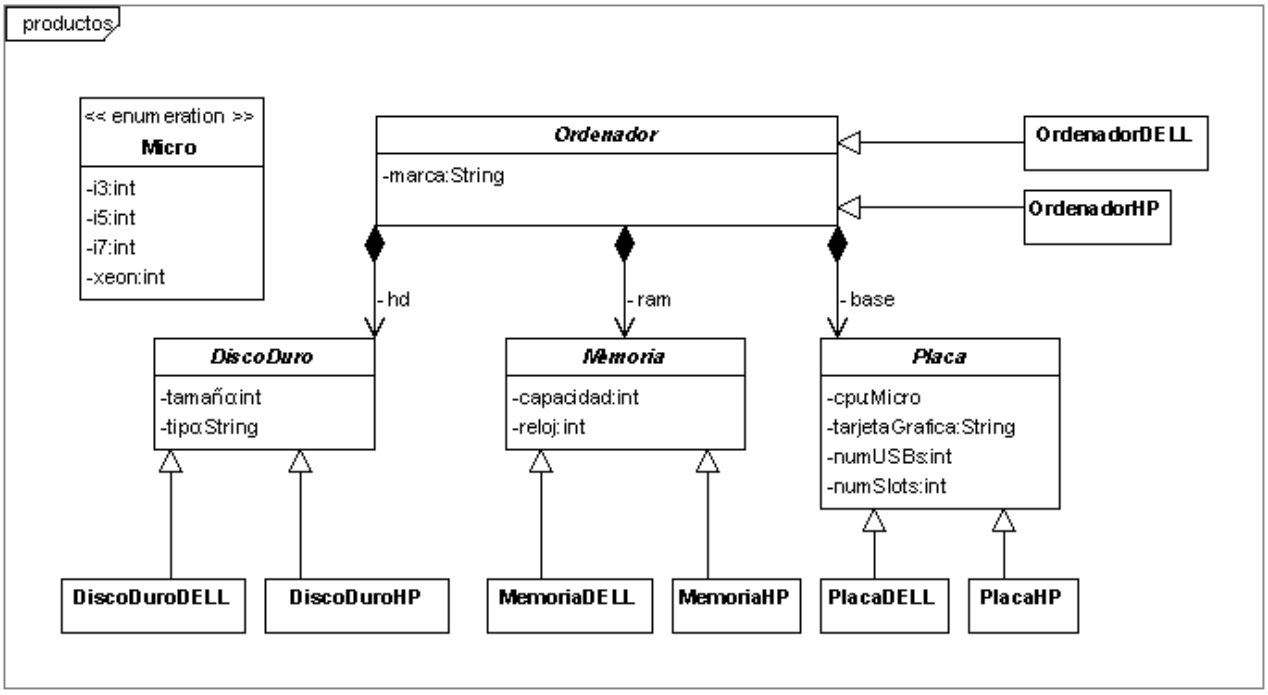
Solución: Se ha usado el patrón proxy

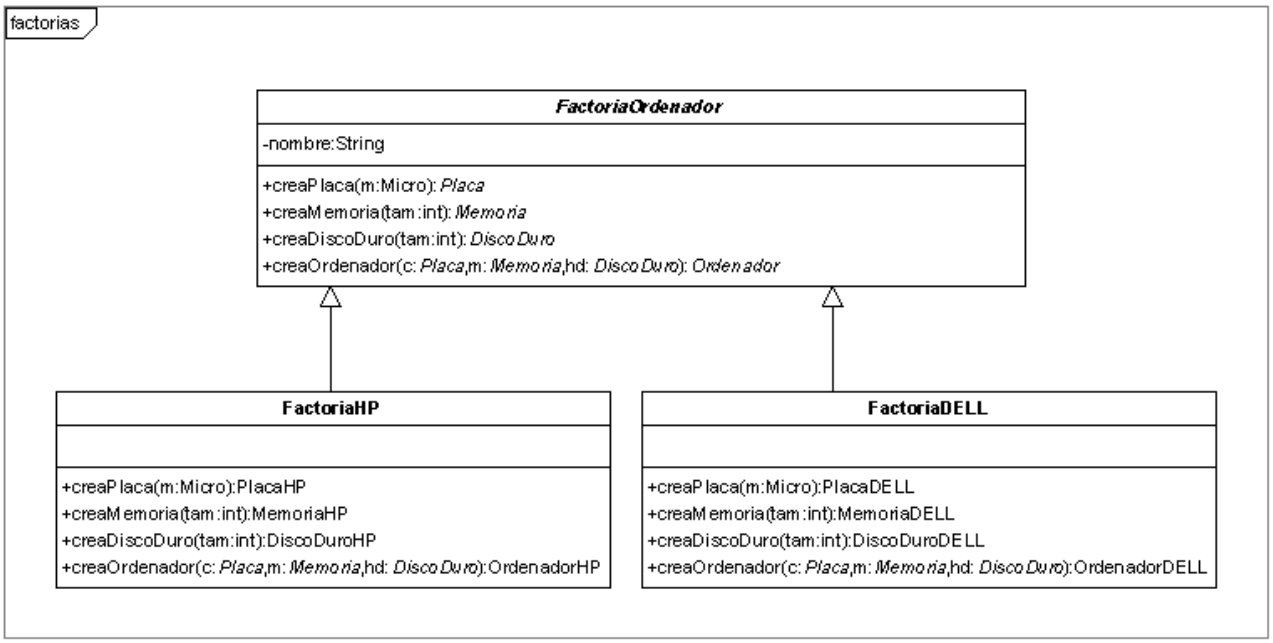
```
class PrinterUtil {
	public static IPrinter makeExclusive(Printer p) {
		return new PrinterProxy(p);
	}
}

interface IPrinter {  //Completar
	void startJob(String header, String user);
	void printPage(String page, String user);
	void endJob(String footer, String user);
}

class PrinterProxy implements IPrinter {
	private Printer printer;
	private String  usuario = null;

	public PrinterProxy(Printer p) {
		this.printer = p;
	}

	@Override
	public void startJob(String header, String user) {
		if (this.usuario == null) {
			this.usuario = user;
			this.printer.startJob(header, user);
		}
	}

	@Override
	public void printPage(String page, String user) {
		if (canPrint(user)) {
			this.printer.printPage(page, user);
		}
	}

	private boolean canPrint(String user) {
		return this.usuario!=null && this.usuario.equals(user);
	}

	@Override
	public void endJob(String footer, String user) {
		if (canPrint(user)) {
			this.printer.endJob(footer, user);
			this.usuario = null;
		}
	}

}

class Printer implements IPrinter{
	public void startJob(String header, String user) {
		System.out.println("User "+user+" Printing Header:: "+header);
	}
	public void printPage(String page, String user) {
		System.out.println("User "+user+" Printing Page:: "+page);
	}
	public void endJob(String footer, String user) {
		System.out.println("User "+user+" Printing Footer:: "+footer);
	}
}
```

2) Se quiere desarrollar una aplicación que modele la construcción de ordenadores a partir de componentes. Un ordenador está formado por una placa base, memoria RAM y disco duro. Una placa base está descrita por un tipo de microprocesador, un tipo de tarjeta gráfica, así como un número de puertos USBs y de expansión. Las memorias RAM se describirán por su capacidad en Mb y su frecuencia de reloj (MHz). Finalmente el disco duro tiene una capacidad (Gb) y un tipo.

La aplicación debe considerar distintas marcas de ordenadores (DELL, HP), de tal manera que un ordenador de una determinada marca no puede llevar componentes de otras marcas.

Se pide:
a) Realizar un diagrama de clases, utilizando los patrones de diseño más adecuados. Realiza un diseño flexible, que permita añadir nuevas marcas de ordenadores, a la vez que facilite la construcción de ordenadores independientemente de la marca, en el código cliente. **Se utiliza el patrón abstract Factory.**



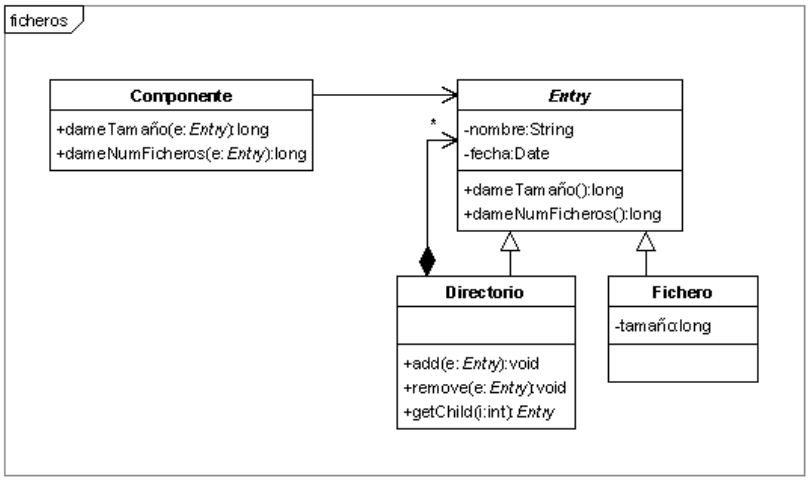


3) Queremos desarrollar un componente para la gestión de ficheros que permita agruparlos en directorios y subdirectorios. Ficheros, directorios y subdirectorios tienen un nombre y una fecha de última modificación. Además, los ficheros ocupan un número de KBs. El componente debe permitir calcular el tamaño 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:

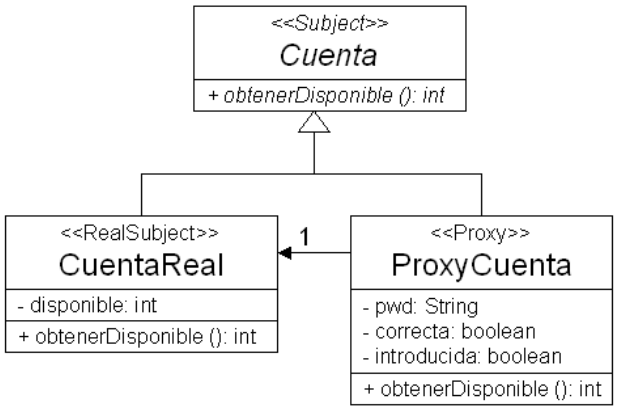


El requisito de seguridad se puede tratar de dos formas:
1. Los métodos add, remove y getChild se definen en la clase Directorio, de manera que no pueden invocarse sobre la clase Fichero. De ese modo se evita que un fichero incluya otros ficheros o directorios. El problema es que el tratamiento de directorios 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 Directorios.

4) 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.

Se pide:
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.
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:
a) Se utiliza el patrón proxy



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 { // decidimos usar una interfaz, ya que no hay atributos comunes
   int obtenerDisponible ();  // De manera más directa, podría usarse una clase abstracta
}

//-------------------------------------------------------
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());
   }
}
```

5) Se quiere construir una clase genérica `ConjuntoInfinito` para emular conjuntos ordenados, de tamaño potencialmente infinito. Este tipo de conjuntos no mantiene sus elementos en memoria (ya que sería imposible), sino que se configuran con una condición de pertenencia, que se utiliza para saber si un elemento pertenece al conjunto.

El constructor de `ConjuntoInfinito` recibe tres parámetros: la condición de pertenencia, y dos parámetros adicionales que permiten buscar miembros potenciales del conjunto. El segundo parámetro es el elemento inicial a partir del que se buscarán elementos, y el tercero es una expresión que permite obtener el siguiente elemento potencial del conjunto. Este siguiente elemento pertenecerá al conjunto sólo si cumple la condición de pertenencia.

Para facilitar el recorrido de los elementos de un conjunto infinito, se creará una clase `RecorreConjunto`. Adicionalmente debe ser posible convertir un `ConjuntoInfinito` a un conjunto ordenado estándar de Java (truncándolo hasta un límite máximo de elementos).

Se pide:
a) Implementar las clases e interfaces necesarias para que el siguiente código produzca la salida de más abajo.
b) ¿Qué patrón(es) de diseño has utilizado? Explica los roles de las clases, interfaces y métodos en los patrones usados.

```
public class Conjuntos {
  public static void main(String ... args ) {
    ConjuntoInfinito<Integer> pares = // Conjunto infinito de los números pares
                        new ConjuntoInfinito<>( x -> x % 2 == 0, // condición de pertenencia al conjunto
                                0, // elemento inicial (no necesariamente debe cumplir la condición)
                                n -> n + 1); // Calcula el próximo elemento a considerar
    RecorreConjunto<Integer> rec = pares.getRecorreConjunto();
    int num = 0;
    while ( num ++ < 20) { // imprime los 20 primeros elementos
        System.out.print( rec.elementoActual()+ " " ); // imprime el elemento actual
        rec.avanza(); // avanza al siguiente elemento
    }
    SortedSet<Integer> conjuntoNormalOrdenado = pares.toSet(5); // Convertir a un conjunto normal ordenado...
    System.out.println( conjuntoNormalOrdenado ); // ... que se trunca a 5 elementos
    System.out.println( pares.toStream(5).reduce(0, (x, y) -> x+y)); // obtiene stream de longitud 5 y suma
  }
}
```

Salida esperada:

```
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 [0, 2, 4, 6, 8]
20
```

Solución:
a) y b) Se ha usado el patrón iterator (roles en comentarios)

```
public class ConjuntoInfinito<T extends Comparable<T>> implements Predicate<T>, UnaryOperator<T>{
// ConcreteAggregate en patrón Iterator
	private Predicate<T> pertenencia;
	private T inicial;
	private UnaryOperator<T> siguiente;

	public ConjuntoInfinito(Predicate<T> pertenencia, T inicial, UnaryOperator<T> siguiente) {
		this.pertenencia = pertenencia;
		this.inicial = inicial;
		this.siguiente = siguiente;
	}

	public RecorreConjunto<T> getRecorreConjunto() {
		return new RecorreConjunto<T>(this);
	}

	@Override
	public boolean test(T element) {
		return this.pertenencia.test(element);
	}
```

```
	public T inicial() {
		return this.inicial;
	}

	@Override
	public T apply(T actual) {
		return this.siguiente.apply(actual);
	}

	public SortedSet<T> toSet(int i) {
		return this.toStream(i).
					collect(TreeSet::new, TreeSet::add, TreeSet::addAll);
	}

	public Stream<T> toStream(int i) {
		return Stream.iterate(this.inicial, this.siguiente).
				filter(this.pertenencia).
				limit(i);
	}
}

public class RecorreConjunto<T extends Comparable<T>> {  // clase ConcreteIterator en patrón Iterator
	private ConjuntoInfinito<T> subject;
	private T actual;

	public RecorreConjunto(ConjuntoInfinito<T> conjuntoInfinito) {
		this.subject = conjuntoInfinito;
		this.actual = conjuntoInfinito.inicial();
		while (!conjuntoInfinito.test(this.actual)) {
			this.avanza();
		}
	}

	public T elementoActual() {
		return this.actual;
	}

	public void avanza() {
		do {
			this.actual = this.subject.apply(this.actual);
		} while (! this.subject.test(this.actual));
	}

}
```

6) Usando el patrón de diseño “factory method”, crear una clase genérica AlmacenOrdenable que pueda configurarse con un criterio de ordenación (que por defecto será el orden natural). Crea un programa que configure la clase para que ordene por tamaño una colección de Strings.

```
public class AlmacenOrdenable<T extends Comparable<T>> {
	private List<T> datos = new ArrayList<>();

	public AlmacenOrdenable(T ...ts) {
		datos.addAll(Arrays.asList(ts));
		datos.sort(this.getCriterio());
	}

	public Comparator<? super T> getCriterio() {
		return Comparator.naturalOrder();
	}

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

public class Main {
	public static void main(String[] args) {
		AlmacenOrdenable<String> palabras =
				new AlmacenOrdenable<String>("esto", "es", "una", "prueba")
				{
					@Override
					public Comparator<? super String> getCriterio() {
						return (o1, o2) -> o1.length() - o2.length();
					}
				};
		System.out.println(palabras);
	}

}
Salida:
[es, una, esto, prueba]
```