

1.- (1,5 puntos) Se está diseñando una aplicación para gestionar una colección de monedas que se va a denominar **Numismatik**. Esta aplicación permite gestionar las distintas monedas de euro de la colección por medio de la clase **EuroCoin**:

```
public class EuroCoin {
    private String description;
    private int year;
    private String country;
    private double value;

    public EuroCoin(String description, int year,
                    String country, double value) {
        this.description = description; this.year = year;
        this.country = country; this.value = value;
    }
    // getters
    public String getDescription() { return description; }
    public int getYear() { return year; }
    public String getCountry() { return country; }
    public double getValue() { return value; }
}
```



España 5 céntimos
Catedral Santiago
2015

La colección de monedas vendrá representada por una simple lista de monedas:

```
public class Numismatik {
    private ArrayList<EuroCoin> myCollection;

    // ...
}
```

Sin embargo, los diseñadores de la aplicación se han dado cuenta de las limitaciones de tener únicamente una lista donde estén todas las monedas a la hora de organizar la colección.

Por esa razón, el diseño de la aplicación debe permitir a los usuarios agrupar sus monedas de forma flexible. Cada grupo de monedas (tipo de dato **Group**) debe disponer al menos de las operaciones:

```
String getDescription(); // genera una cadena con la descripción de la
moneda o las monedas del grupo.
double getValue(); // genera el valor de la moneda o la suma de valores de
todas las monedas del grupo.
```

El diseño debe facilitar las operaciones para añadir a un grupo tanto monedas sueltas, como otros grupos de monedas previamente definidos. De igual forma se podrán eliminar de un grupo tanto monedas sueltas como los grupos previamente añadidos.

Se pide:

Selecciona el patrón más adecuado para esta tarea. **Justifica tu respuesta.** Indica su tipo, y si el patrón utilizado tiene varias versiones justifica cual utilizarías. Indica también que papel toma cada clase en dicho patrón.

En este caso el patrón seleccionado sería un **patrón estructural**, ya que nos solicitan un mecanismo para organizar las monedas en distintas agrupaciones de manera jerárquica. Por esta razón el más apropiado de los vistos en la asignatura es el patrón **Composite** (no hemos visto más que una versión de él así que no hay que especificarlo). Este patrón es adecuado porque nos permite manipular de forma uniforme tanto objetos Hoja (las **EuroCoins**), como objetos **Compuesto** (las agrupaciones de **EuroCoins**). El cambio implicaría además que la aplicación pasaría a manejar una colección de objetos que implementen una interface **Componente** que implementarían tanto **EuroCoin** como la clase **Compuesto**.

2.- (1,5 puntos). La aplicación **Numismatik** dispone de acceso a una casa de subastas para comprar y vender monedas en la nube pública de Internet.

```
public class Numismatik {
    // ...
    AuctionHouse ah; // auction house manager

    public Numismatik() {
        ah = new AuctionHouse();
    }
    // User buy/sell coins requests
    void sellCoin(EuroCoin coin) {
        // sell a coin in the Cloud Auction House
        ah.sellCoin(coin);
    }

    void buyCoin(EuroCoin coin) {
        // buy a coin in the Cloud Auction House
        ah.buyCoin(coin);
    }
}
```

Para ello dispone de la clase **AuctionHouse** que se encargará de las operaciones de venta y compra de monedas en la casa de subastas en la nube.

```
public class AuctionHouse {

    /**
     * buy a coin in the Cloud auction house
     * @param coin the coin
     */
    public void buyCoin(EuroCoin coin) { /* connect to the Cloud and buy */ }

    /**
     * sell a coin from the Cloud auction house
     * @param coin
     */
    public void sellCoin(EuroCoin coin) { /* connect to the Cloud and sell */ }
}
```

Debido al contrato con la casa de subastas, resulta muy costoso lanzar cada operación de venta/compra de moneda en el instante en que la genera el usuario. En su lugar se desea almacenar todas estas peticiones y al final de la semana lanzarlas todas mediante un nuevo método **Numismatik.launchOperations()**. De esta manera si el usuario se arrepiente de una operación podría cancelarla antes de que acabe la semana.

Se pide:

Selecciona el patrón más adecuado para esta tarea. **Justifica tu respuesta.** Indica su tipo, y si el patrón utilizado tiene varias versiones justifica cual utilizarías. Indica también que papel toma cada clase en dicho patrón.

En este caso necesitamos de un patrón de **Comportamiento** y en concreto **Command** (no tiene versiones). Lo que permitirá separar el instante en que se crea la orden de compra o venta de moneda, del instante en que realmente se ejecuta. La aplicación **Cliente** será nuestra clase **Numismatik** y **AuctionHouse** actuará como

Receptor. Finalmente, cada operación de **AuctionHouse** dará lugar a cada una de las clases que implementen la interface **ICommand**

3.- (2,75 puntos) Aplica el patrón elegido en la pregunta 2. Actualiza el diagrama de clases, describe qué función tiene cada nueva interfaz/clase. Parte de la clase **Numismatik** se presenta a continuación, completa las cajas de bordes discontinuos con el código que falta. **(2,75 puntos)**

```
public class Numismatik {
    //...
    AuctionHouse ah; // auction house manager

    [ ] operations; // List of add/remove
                    coins operations

    public Numismatik() {
        ah = new AuctionHouse();

        operations = new [ ] ;
    }

    public void sellCoin(EuroCoin coin) {
        // add "sell a coin" to operations
        [ ]
    }

    public void buyCoin(EuroCoin coin) {
        // add "buy a coin" to operations
        [ ]
    }

    public void launchOperations() {
        // launch buy/sell operations
        [ ]
    }
}
```

[Consulta la solución en el proyecto prototipo de Java](#)

4.- (1,5 puntos) Visto el éxito de la aplicación para gestionar colecciones de monedas, se plantea la posibilidad de integrar en la aplicación y en las colecciones además de monedas también sellos. Para ello desde otra aplicación denominada **Filatelik**, nos proporcionan la clase **Stamp** que utilizan para representar sus sellos de colección. La única restricción que nos imponen es que no podemos modificar dicha clase en **Numismatik**.

<pre> <<Java Class>> Stamp numismatik id: String productionDate: int region: String cost: double Stamp(String,int,String,double) getId():String getProductionDate():int getRegion():String getCost():double </pre>	<pre> public class Stamp { private String id; private int productionDate; private String region; private double cost; public Stamp(String id, int date, String region, double cost) { this.id = id; this.productionDate = date; this.region = region; this.cost = cost; } // getters public String getId() { return id; } public int getProductionDate() { return productionDate; } public String getRegion() { return region; } public double getCost() { return cost; } } </pre>
--	---

Con esta restricción decidimos actualizar el diseño de **Numismatik** y **EuroCoin** para que nuestra aplicación pueda manejar colecciones de monedas y sellos.

Se pide:

Selecciona el patrón más adecuado para esta tarea. Justifica tu respuesta. Indica su tipo, y si el patrón utilizado tiene varias versiones justifica cual utilizarías. Indica también que papel toma cada clase en dicho patrón.

En esta ocasión necesitamos un patrón de tipo **Estructural** y en concreto **Adapter**. En nuestra aplicación **Numismatik** (que actúa como **Cliente**) tenemos una clase de objetos **Stamp** a adaptar (la clase **Adaptable** del patrón) para que funcionen como objetos de tipo **EuroCoin** (que actúa como clase **Objetivo**). A priori podemos utilizar tanto la versión de **Adapter de clases** como **Adapter de objetos**.

5.- (2,75 puntos) Aplica el patrón elegido en la pregunta 4. Actualiza el diagrama de clases, describe qué función tiene cada nueva/modificada interfaz/clase e impleméntala. Parte de la clase **Numismatik** se presenta a continuación, haz los cambios necesarios en ella y completa en esta página el ejemplo de uso para que se ajuste al diseño del patrón y se pueda añadir tanto la moneda como el sello a la colección.

```
public class Numismatik {
    //...
    private ArrayList<EuroCoin> myCollection;

    // add coin to the collection
    void addToCollection(EuroCoin c) { myCollection.add(c); }
}

// Ejemplo de uso
Numismatik app = new Numismatik();
EuroCoin c = new EuroCoin("spanish 5cents coin", 2015, "Spain", 0.05);
Stamp s = new Stamp("Vader", 2017, "Spain", 5.00);
app.addToCollection( ); // add c to collection
app.addToCollection( ); // add s to collection
```



En el código de la solución os ofrecemos tanto la resolución utilizando el patrón en su versión de clases (basada en herencia), aprovechando que podemos modificar **Numismatik** y **EuroCoin** introducimos una interface **Item**, sino no sería posible al no disponer de herencia múltiple; como la versión de objetos (basada en composición).

[Consulta la solución en el proyecto prototipo de Java](#)