Una compañía de autobuses tiene modelados sus vehículos mediante la siguiente clase:

```
abstract class  Bus{
Bus(String id, long km, double gas); // fijamos sus atributos al construirlo
String getId(); // obtén matricula del bus
long getKM(); // obtén los kms recorridos por el bus
double getGas(); // obtén los litros de gasolina consumidos por el bus
}
```

La empresa dispone de diferentes subtipos de buses modelados como subclases de la clase Bus. Y que además con el tiempo pueden aparecer nuevos tipos de autobuses, y de forma similar, desaparecer algunos de los subtipos existentes.

El equipo de programación de la Empresa decide aplicar patrones de diseño que faciliten estos cambios. Centrándonos en la tarea de crear objetos de las subclases de Bus a partir de un tipo identificado por un dato (por ejemplo un String: “tipo1”, “tipo2”, etc.) se pide:

1.- Indica el patrón de diseño que deberíamos utilizar y justifica por qué crees que se ajusta a este problema. Si el patrón utilizado tiene varias versiones justifica cual utilizarías **(1 punto)**

En este caso necesitamos un patrón de tipo Creacional, que nos permita en función de un dato crear objetos de un tipo u otro. Por tanto un patrón Factory Method, en su versión parametrizada, es el que mejor nos permite separa la creación del autobús en concreto, de cómo se utiliza a posteriori. En este caso descartamos utilizar un patrón Abstract Factory ya que sólo hay un tipo de producto.

Los conductores de la empresa se representan mediante objetos de esta segunda clase:

```
abstract class  Conductor{
Conductor(String nif, String nombre ); // fijamos sus atributos al construirlo
String getNif(); // obtén NIF del conductor
String getNombre(); // obtén el nombre del conductor
}
```

De nuevo suponemos que hay diferentes tipos de conductores modelados como subclases de Conductor, y se decide utilizar el mismo patrón que se empleó en el apartado 1 en este caso aplicado a los conductores.

2.- Justifica sí esta decisión sigue siendo válida en el caso de que para cada subtipo de Bus solamente pudiera tener asociado un **único** subtipo concreto de Conductor. En caso de no ser válido el patrón anterior justifica que otro patrón si lo sería. **(2 puntos)**

De nuevo seguimos necesitando un patrón creacional por cada tipo de producto. Podríamos pensar en un Factory Method para los buses y otro Factory Method para los conductores, pero el resolverlo por separado nos llevaría a que se rompieran las correspondencias entre el tipo de bus y el tipo de conductor. Así que el patrón a utilizar es Abstract Factory, ya que tenemos dos tipos de productos: buses y conductores; y por cada familia (o subtipo): normal, supra, microbús… sólo hay un subtipo de bus y conductor compatibles.

La compañía nos solicita ampliar su sistema de información, de forma que se puedan agrupar libremente sus vehículos dando lugar a objetos MetaBus que puedan representar tanto un conjunto de autobuses, un solo autobús o incluso mezcla de autobuses y otras agrupaciones (otros objetos MetaBus previamente definidos). Pero nos piden que los objetos MetaBus presenten la misma interfaz que un simple Bus, es decir:

```
Interface MetaBus{
String getId(); // matricula del bus o nombre que se da a un grupo de autobuses
long getKM(); // Kms totales recorridos por el bus o buses
double getGas(); // litros totales de combustible consumidos por el bus o buses
}
```

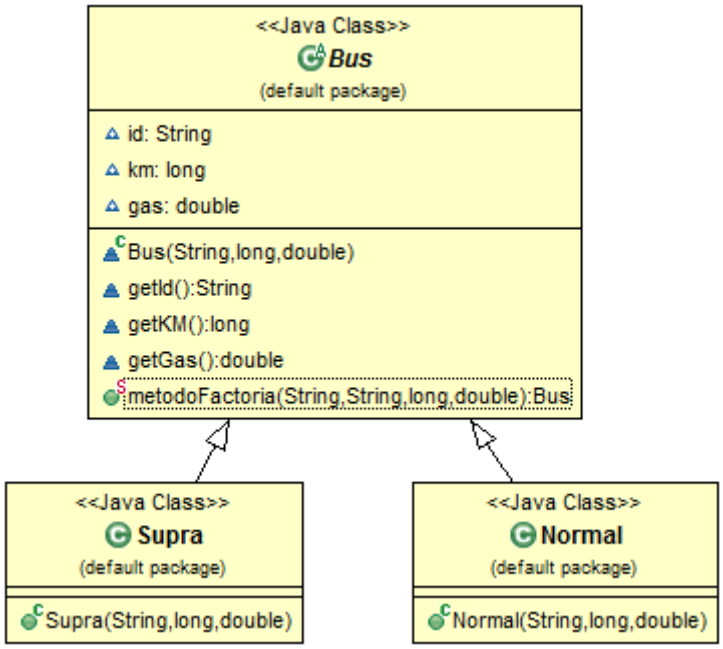
Para ello:

3.- Indica el patrón de diseño que deberíamos utilizar y justifica por qué crees que se ajusta a este problema. **(1 puntos)**

Nos están pidiendo que podamos agrupar de forma jerárquica los autobuses, pudiendo combinar tanto autobuses individuales como agrupaciones previas de autobuses. En definitiva necesitamos un patrón estructural Composite para representar esta estructura arborescente de objetos simples (buses) y compuestos (agrupaciones de buses).

4.- Define la jerarquía de objetos necesaria para facilitar la creación de buses en función de su tipo. Describe qué función tiene cada nueva Interfaz/Clase nueva, e implementa las operaciones de cada una de ellas. **(2 puntos)**

Añadimos en la clase Bus un método estático metodoFactoria() que recibe el tipo de bus a crear y devuelve un objeto de la subclase correcta. Además recibe los argumentos necesarios para crear el Bus.



```
public static Bus metodoFactoria (String tipo, String id, long km, double gas){
		switch(tipo){
		case "supra":
			return new Supra(tipo, km, gas);

		case "normal":
			return new Normal(tipo, km, gas);

		default:
			return null;
		}
}
```

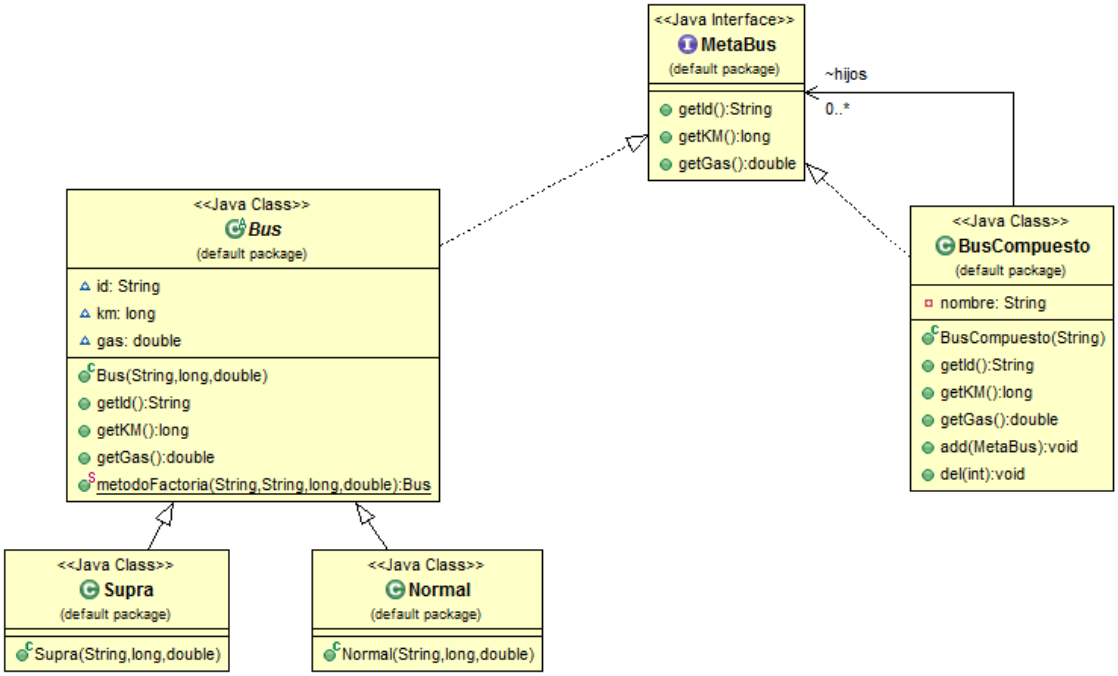
5.- Pon un ejemplo de uso del patrón anterior suponiendo que existen como subclases los buses "Normal" y "Supra". Crea para ello una clase "Cliente" con un método creaYMuestra(String tipo), que cree un bus de ese tipo y lo muestre por consola. **(1 punto)**

```
public class Cliente {

	public static void main(String[] args) {
		// crea y muestra un autobus supra
		creaYMuestra("supra");
	}

	private static void creaYMuestra(String tipo) {
		//  crea el bus utilizando el metodo factoria
		Bus b = Bus.metodoFactoria(tipo, "01", 123, 50);

		// imprime el bus independientemente del tipo que tenga
		System.out.printf("%s %d %f\n",
					b.getId(), b.getKM(), b.getGas());

	}
}
```

La compañía nos solicita ampliar su sistema de información, de forma que se puedan agrupar libremente sus vehículos dando lugar a objetos MetaBus que puedan representar tanto un conjunto de autobuses, un solo autobús o incluso mezcla de autobuses y otras agrupaciones (otros objetos MetaBus previamente definidos). Pero nos piden que los objetos MetaBus presenten la misma interfaz que un simple Bus, es decir:

```
Interface MetaBus{
String getId(); // matricula del bus o nombre que se da a un grupo de autobuses
long getKM(); // Kms totales recorridos por el bus o buses
double getGas(); // litros totales de combustible consumidos por el bus o buses
}
```

6.- Define la jerarquía de objetos necesaria para implementar los Metabuses. Describe qué función tiene cada nueva Interfaz/Clase nueva, e implementa las operaciones de cada una de ellas. **(3 puntos)**



La interface MetaBus engloba tanto buses simples (clase Bus), como agrupaciones de objetos MetaBus (clase BusCompuesto).

La Clase Bus pasa a implementar por tanto la interface MetaBus, no necesitamos cambiar nada de su implementación ya que dispone de antemano de los métodos requeridos de la interface.

La Clase BusCompuesto, dispone de un array de objetos MetaBus (sus hijos), e implementa los métodos de MetaBus (de acuerdo con los comentarios del enunciado). E incorpora los métodos para gestionar sus hijos (add() y del()).

La implementación de esta clase BusCompuesto sería la siguiente:

```
import java.util.ArrayList;


public class BusCompuesto implements MetaBus {

	private ArrayList<MetaBus> hijos;
	private String nombre;

	public BusCompuesto(String nombre) {
		this.nombre = nombre;
	}
```

```
	// operaciones interface MetaBus
	public String getId() {
		return nombre;
	}


	public long getKM() {
		long total = 0;
		for(MetaBus b: hijos)
			total += b.getKM();

		return total;
	}

	public double getGas() {
		double total = 0;
		for(MetaBus b: hijos)
			total += b.getKM();

		return total;
	}

	// operaciones de gestion de hijos
	public void add(MetaBus b){
		hijos.add(b);
	}

	public void del(int pos){
		hijos.remove(pos);
	}

}
```