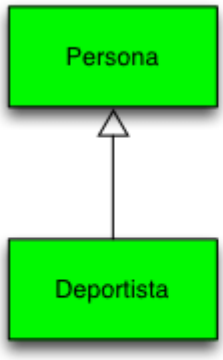
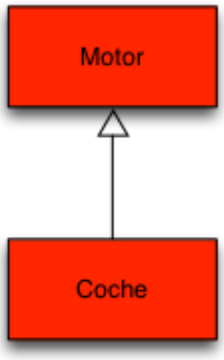
En el post anterior hemos visto como la herencia no siempre encaja de una forma natural y flexible con el diseño de nuestras aplicaciones .Una de las primeras preguntas que nos tenemos que hacer a la hora de relacionar dos clases a traves de la herencia es si ambas clases tienen una relación de categorización .Lo mas facil para resolver esto es preguntarse lo siguiente. ¿Es un Coche un Motor? si la respuesta es NO estamos ante una situación en la cual no deberíamos usar la herencia. En el otro caso que nos ocupa. ¿Es un Deportista una Persona? en este caso la respuesta es SI. Por lo tanto estamos ante una situación en la que la herencia puede encajar de forma mas natural.



Relacion de Categorización
(El deportista es una Persona)



El Coche no es un motor

Ahora bien una vez que tenemos claro que el Motor y el Coche no se deben relacionar a traves de Herencia vamos a ver el código fuente original de estas clases para luego poder refactorizarlo

```
public class Motor {

private int potencia;
 private int cilindros;
 public int getPotencia() {
 return potencia;
 }
```

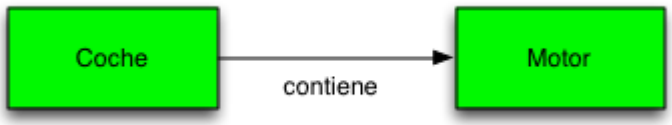
```
 public void setPotencia(int potencia) {
 this.potencia = potencia;
 }
 public int getCilindros() {
 return cilindros;
 }
 public void setCilindros(int cilindros) {
 this.cilindros = cilindros;
 }
 public Motor(int potencia, int cilindros) {
 super();
 this.potencia = potencia;
 this.cilindros = cilindros;
 }

}



public class Coche extends Motor {

private String marca;

public String getMarca() {
 return marca;
 }

public void setMarca(String marca) {
 this.marca = marca;
 }

public Coche(int potencia, int cilindros,String marca) {
```

```
 super(potencia, cilindros);
 this.marca=marca;

 }
}
```

Una vez tenemos definidas las dos clases vamos a ver como se comporta el programa Principal o Main.

```
public class Principal {

public static void main(String[] args) {

Coche c= new Coche(100,6,"A");
 System.out.println(c.getCilindros());

}

}
```

Vamos a refactorizar el código para construir una relación de composición entre las dos clases (El coche incluye el concepto de Motor).



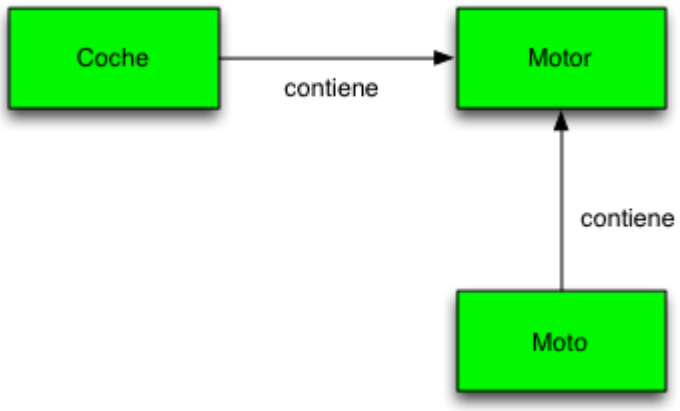
Ahora el código de nuestras clases queda de la siguiente forma ya que la clase Coche incluye el Motor.

```
public class Motor {

private int potencia;
 private int cilindros;

 public Motor(int potencia, int cilindros) {
 super();
 this.potencia = potencia;
 this.cilindros = cilindros;
 }
 public int getCilindros() {
 return cilindros;
 }
 public void setCilindros(int cilindros) {
 this.cilindros = cilindros;
 }
 public int getPotencia() {
 return potencia;
 }
 public void setPotencia(int potencia) {
 this.potencia = potencia;
 }
}
```

```
public class Coche {
```

```
private String marca;
 private Motor motor;

 public String getMarca() {
 return marca;
 }

public void setMarca(String marca) {
 this.marca = marca;
 }

public Motor getMotor() {
 return motor;
 }

public void setMotor(Motor motor) {
 this.motor = motor;
 }
 public Coche(Motor motor,String marca) {
 super();
 this.motor = motor;
 this.marca=marca;
 }
}
<pre>
```

Una vez refactorizado el código usando composición como concepto fundamental para la relación entre clases ganamos en flexibilidad y reutilización de código ya que por ejemplo la clase Moto podría apoyarse en la clase Motor.



.

Parece todo correcto pero si miramos a la nueva versión del programa principal que necesitamos para imprimir los cilindros nos encontraremos con lo siguiente:

```
public class Principal {

public static void main(String[] args) {

 Motor motorA= new Motor(100,6);
 Coche c= new Coche(motorA,"A");
 System.out.println(c.getMotor().getCilindros());

}
}
```

Es evidente que la linea 7 ha aumentado la complejidad a la hora de trabajar con nuestras clases . Hay situaciones en las que refactorizar el código nos puede llegar a situaciones mas flexibles .. pero también mas complejas. En el siguiente post trataremos esto.