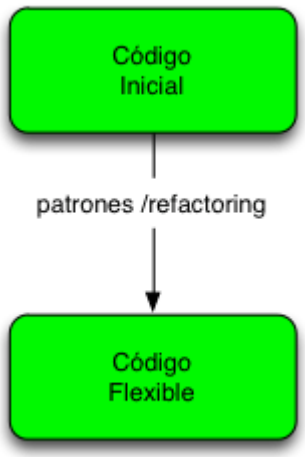
En la anterior entrada hemos terminado comparando un diseño flexible con un diseño rígido. Sin embargo el diseño que a priori era rígido tenía sus ventajas ya que era mas facil trabajar con él.

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

Comparado con el diseño que teníamos que era mas flexible pero a la vez mas enrevesado de trabajar.

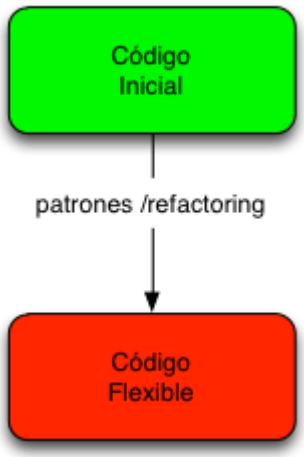
```
Motor motorA= new Motor(100,6);
Coche c= new Coche(100,6,"A");
System.out.println(c.getMotor().getCilindros());
```

Esto se debe a que en muchos casos los diseñadores se centran en convertir el código inicial en un código que sea mas flexible y que se pueda usar en situaciones varias.

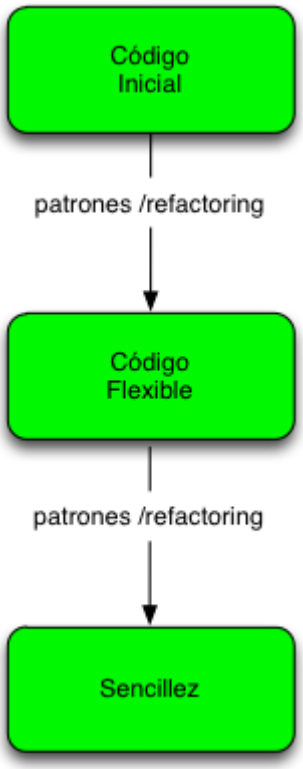


Ahora bien que nuestro código sea flexible ¿Es realmente el objetivo que queremos

alcanzar? . La respuesta parece ser bastante obvia y debiera ser SI. Sin embargo no hay que ir muy lejos para recordar tecnologías Java en las que la flexibilidad es grande (Caso de Swing , o de EJB 1.0 y EJB 2.0) y son tecnologías que no han tenido el éxito deseado. Por lo tanto parece claro que con diseñar una solución flexible no es suficiente.



Para que un diseño que propongamos tenga éxito ha de ser flexible pero a la vez ha de ser sencillo de utilizar.



¿Como podemos convertir el código actual en algo sencillo de manejar? .Deberemos hacer uso de dos conceptos . En primer lugar del concepto de encapsulación . En estos momentos el Motor es demasiado visible a nivel de código

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

Vamos a modificar el código de la clase Coche y encapsular al máximo el Motor

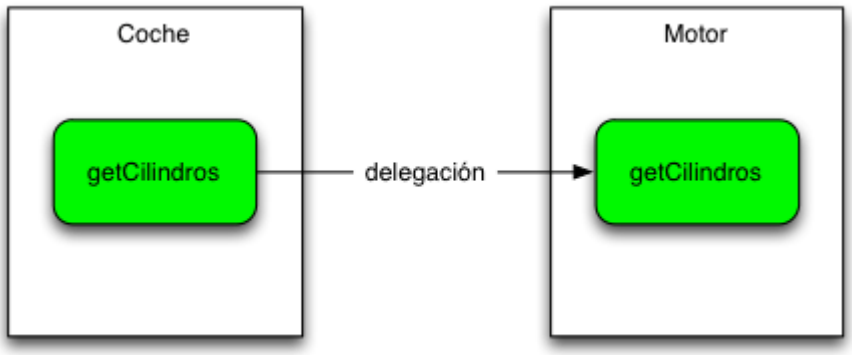
```
public class Coche {
```

```
private String marca;
 private Motor motor;

 public String getMarca() {
 return marca;
 }

public void setMarca(String marca) {
 this.marca = marca;
 }
 public Coche(int potencia ,int cilindros,String marca) {
 super();
 this.motor =new Motor(potencia,cilindros);
 this.marca=marca;
 }
}
```

Hemos eliminado el Motor en el código del constructor y hemos eliminado también los métodos set/get que hacían referencia a este. De esta forma el Motor no es visible desde el programa principal.

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

Lamentablemente este hecho nos hace imposible acceder a los cilindros y mostrarlos por pantalla . Para solventar este problema usaremos el concepto de delegación y publicaremos nuevos métodos set/get para cilindros y para potencia a nivel de la clase Coche que deleguen en el Motor.



Vamos a ver el nuevo código de la clase Coche

```
public class Coche {

private String marca;
 private Motor motor;

 public String getMarca() {
 return marca;
 }

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

public Coche(int potencia,int cilindros,String marca) {
 super();
 this.motor = new Motor(cilindros,potencia);
 this.marca=marca;
 }

public int getCilindros() {
```

```
 return motor.getCilindros();
 }

public void setCilindros(int cilindros) {
 motor.setCilindros(cilindros);
 }

public int getPotencia() {
 return motor.getPotencia();
 }

public void setPotencia(int potencia) {
 motor.setPotencia(potencia);
 }

}
```

Ahora si que podemos acceder a los cilindros de una forma sencilla y hemos conseguido encapsular el concepto de Motor de tal manera que para el programador que usa el Coche no exista.

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

Hemos conseguido las dos cosas un diseño flexible pero a la vez sencillo de manejar. Ese ha de ser nuestro objetivo a la hora de construir soluciones.