

ACADEMÍA JAVA XIDERAL MONTERREY, N.L

SEMANA 1 - EJERCICIO 3: INYECCIÓN DE DEPENDENCIAS

**Realizado por:
Luis Miguel Sánchez Flores**

PROBLEMÁTICA:

Imagina un restaurante que se ha convertido muy popular y exitoso gracias a su amplio menú de comida deliciosa que todo el mundo saborea. El restaurante es tan popular que siempre está lleno de gente durante todo el día.

Si bien el restaurante es decente, todavía no ha recibido una estrella Michelin.

Eso se debe al tiempo de espera del que se queja la mayoría de los clientes, afirmando que la comida tarda demasiado en salir.

Aunque los chefs del restaurante son estupendos sirviendo platos deliciosos, **se vuelven torpes a la hora de buscar y obtener las recetas, tardando bastante tiempo en el proceso.**

Fue entonces cuando al dueño del restaurante se le ocurrió una pregunta:

¿hay alguna forma de que los cocineros puedan delegar la búsqueda de recetas a alguien o algo, y centrarse mejor en preparar la comida?

¿Podría eso mitigar o incluso resolver el problema del tiempo de espera del que se quejan los clientes?



SOLUCIÓN:

En la programación habitual, los objetos crean y administran sus propias dependencias. Esta dependencia puede ser muy problemática de modificar, ya que puede estar muy integrada con el objeto que depende de ella, lo que puede provocar en un código laborioso y más propenso a errores.

Se puede evitar esta problemática mediante la implementación de patrón de diseño conocido como la **Inyección de Dependencias**, en donde las dependencias requeridas (recetas) son proporcionadas de forma externa (por el asistente) a los objetos (chefs), en lugar de permitir que los propios objetos las generen.

Mediante la inyección de Dependencias se facilita el cambio de dependencias que vayan requiriendo los objetos sin afectar los demás componentes del programa, mejorando su mantenimiento y verificabilidad.

De esta forma, el asistente puede proporcionar las recetas apropiadas a los chefs mientras que ellos se concentran en preparar los platos, agilizando el servicio y mitigando el problema de las esperas.

Se empezó con el desarrollo de la interfaz llamada **Receta** que servirá como base para las recetas especializadas. En ella se incluye la función para preparar la receta por un determinado chef:

```
1 package injeccion_dependencias;
2
3 public interface Receta {
4
5     void preparar(Chef chef);
6 }
7
```

Luego se creó una clase **Chef** en el que se proporcionará la receta que necesite a través de una inyección mediante un método setter (**setReceta**):

```
3 public class Chef {
4
5     private String nombre;
6
7     public Chef(String nombre) {
8         this.nombre = nombre;
9     }
10
11     // objeto Receta que pueda requerir el chef:
12     private Receta receta;
13
14     // INYECCIÓN DE DEPENDENCIA MEDIANTE SETTER:
15     public void setReceta(Receta receta) {
16         this.receta = receta;
17     }
18
19     // Preparar la comida con la receta proporcionada al chef:
20     public void prepararComida()
21     {
22         // Intenta preparar la comida:
23         try {
24             receta.preparar(this);
25
26             // Si no tiene una receta con la que trabajar, envía un mensaje de error:
27         } catch (NullPointerException e) {
28             System.err.println("El chef " + nombre + " no tiene una receta con que trabajar...");
29         }
30     }
31 }
```

Se desarrollaron varias recetas especializadas como la receta de **Hamburguesa, Pizza y Tacos**. Cada una de las clases implementa la interfaz de Receta, por lo que integran el método preparar() como se muestra a continuación:

```
3 public class Hamburguesa implements Receta{
4
5     private String tipo;
6
7     public Hamburguesa(String tipo) {
8         this.tipo = tipo;
9     }
10
11     // Prepara la hamburguesa...
12     @Override
13     public void preparar(Chef chef) {
14         System.out.println("El chef " + chef.getNombre() + " prepara la hamburguesa tipo " + tipo + " ....");
15     }
16 }
17 }
```

```
2
3 public class Pizza implements Receta{
4
5     private String tipo;
6
7     public Pizza(String tipo) {
8         this.tipo = tipo;
9     }
10
11     // Prepara la pizza...
12     @Override
13     public void preparar(Chef chef) {
14         System.out.println("El chef " + chef.getNombre() + " prepara la pizza " + tipo + " ....");
15     }
16 }
17 }
```

```
2
3 public class Tacos implements Receta{
4
5     private String tipo;
6
7     public Tacos(String tipo) {
8         this.tipo = tipo;
9     }
10
11     // Prepara la orden de tacos...
12     @Override
13     public void preparar(Chef chef) {
14         System.out.println("El chef " + chef.getNombre() + " prepara los tacos de " + tipo + " ....");
15     }
16 }
17 }
```

Se creó la clase **Asistente** que actuará como el inyector de dependencias para el objeto **Chef** mediante el setter. Para esto, se implementa una función estática en el que se proporciona una receta de cierto tipo hacia el objeto Chef introducido como argumento. Las recetas son validadas mediante el uso de una **enumeración** que establece el listado de comidas que se incluyen en el menú del restaurante. En caso que se proporciona una receta invalidada, se rechaza mediante un error:

```
2
3 // Asistente (Inyector) que se encargará de proporcionar las dependencias (Recetas) que requieran
4 // los objetos (Chef)
5
6 public class Asistente {
7
8
9     static void entregarReceta(Chef chef, Comida comida, String tipo) {
10
11         // Si se pidieron tacos...
12         if(comida == Comida.TACO)
13             // ...entrega la receta correspondiente de Tacos
14             chef.setReceta(new Tacos(tipo));
15
16         // O una pizza...
17         else if(comida == Comida.PIZZA)
18             // ...entrega la receta correspondiente de Pizza
19             chef.setReceta(new Pizza(tipo));
20
21         // O una hamburguesa...
22         else if(comida == Comida.HAMBURGUESA)
23             // ...entrega la receta correspondiente de Pizza
24             chef.setReceta(new Hamburguesa(tipo));
25
26         // Si se recibió una orden invalida...
27         else {
28             // ...se debe rechazarla.
29             throw new UnsupportedOperationException("Esta receta no existe. Intenta con otra receta");
30         }
31     }
32 }
```

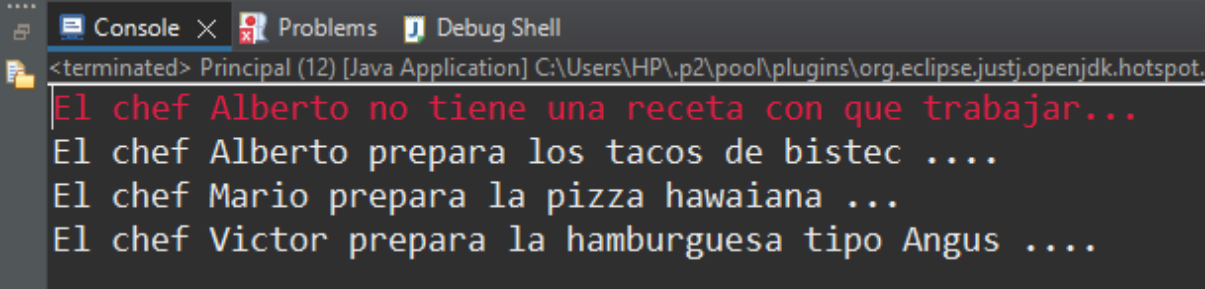
```
3 public enum Comida {
4     TACO, PIZZA, HAMBURGUESA
5 }
6 |
```

Se pone a prueba la inyección de dependencias mediante la función **main**, en el cual se establecen los objetos Chef del restaurante (Alberto, Mario y Victor), quienes reciben las recetas correspondientes por parte del nuevo asistente mediante la función **entregarReceta()**:

```
10 public class Principal {
11
12
13     public static void main(String[] args) {
14
15
16         // Entra el chef Alberto al restaurante...
17         Chef alberto = new Chef("Alberto");
18
19         // Sin ninguna receta proporcionada por el asistente, el chef Alberto
20         // no puede preparar la comida por el momento....
21         alberto.prepararComida();
22
23         /*
24          * Al recibir una orden de tacos de bistec, el asistente proporciona
25          * la receta de tacos hacia Alberto:
26          */
27         Asistente.entregarReceta(alberto, Comida.TACO, "bistec");
28
29
30         // Ahora, el chef Alberto puede preparar los tacos sin ningún problema.
31         alberto.prepararComida();
32
33         // Entra el chef Mario...
34         Chef mario = new Chef("Mario");
35
36         // Con una orden de pizza hawaiana, el asistente le entrega la receta
37
38         // Con una orden de pizza hawaiana, el asistente le entrega la receta
39         // de pizza hawaiana al chef Mario:
40         Asistente.entregarReceta(mario, Comida.PIZZA, "hawaiana");
41
42         // El chef Mario prepara la pizza...
43         mario.prepararComida();
44
45         // El chef Victor es el último en entrar:
46         Chef victor = new Chef("Victor");
47
48         // Con una orden de hamburguesa Angus pendiente, el asistente le entrega
49         // la receta apropiada al chef Victor
50         Asistente.entregarReceta(victor, Comida.HAMBURGUESA, "Angus");
51
52         // El chef victor prepara la hamburguesa con gusto...
53         victor.prepararComida();
54
55     }
56
57
58 }
59
```

El resultado demuestra que, al intentar preparar la comida desde el inicio, el chef Albertono tiene una receta a mano, lo que provoca el envío del mensaje de error que se muestra en la primera línea.

Una vez que se empieza a recibir las órdenes de los comensales y el asistente proporciona las recetas correspondientes hacia los chefs, pueden empezar a preparar las comidas de una forma más rápida y eficaz.



```
****
Console  Problems  Debug Shell
<terminated> Principal (12) [Java Application] C:\Users\HP\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.
El chef Alberto no tiene una receta con que trabajar...
El chef Alberto prepara los tacos de bistec ....
El chef Mario prepara la pizza hawaiana ...
El chef Victor prepara la hamburguesa tipo Angus ....
```