Práctica 8. **El Supermercado**

1 ENUNCIADO

Un nuevo cliente, SmartTPV, le pregunta si podría ayudarle a simular el sistema de cobro de un supermercado. SmartTPV es proveedor de cajas registradoras para una importante cadena de supermercados. Además de aportar las cajas registradoras, SmartTPV hace labores de consultoría aconsejando el número de cajas a instalar en la zona de cobro. SmartTPV utilizará el software para estudiar el número de cajas que deben estar abiertas en base al número de clientes que llegan a las colas de cada caja, manteniendo una buena calidad de servicio.

Las cajas registradoras que provee SmartTPV son de dos tipos:

- Caja de tipo A. Equipadas con lectores ópticos de códigos de barra de última tecnología y sensores de proximidad para el movimiento automático de la cinta transportadora de productos. Por cada producto el tiempo de cobro es de 2 a 10 segundos siguiendo una distribución equiprobable.
- Caja de tipo B. Equipadas con lectores ópticos de códigos de barra baratos que exigen lecturas más cercanas y pedales para movimiento de la cinta transportadora de productos. Por cada producto el tiempo de cobro es de 3 a 20 segundos siguiendo una distribución equiprobable.

La cadena de supermercados se dispone a abrir un nuevo supermercado, para el que SmartTPV aconseja instalar 10 cajas registradoras en la zona de cobro, las cinco primeras de tipo A, las cinco restantes de tipo B.

En base a su experiencia acumulada en otras instalaciones, los analistas de SmartTPV han comprobado:

- *Distribución de clientes en las cajas.* Si hay dos cajas abiertas el 100% de los clientes eligen la caja más vacía entre las abiertas. Pero que si hay tres o más cajas abiertas el 50% eligen la más vacía, el 30 % la segunda más vacía y el 10% la tercera. El 10% restante eligen aleatoriamente entre las cajas abiertas.
- *Distribución de llegada de clientes a la zona de cobro*. Los clientes del supermercado llegan a la zona de cobro con una distribución exponencial cuyo tiempo medio de llegada depende de múltiples variables: tamaño del supermercado, cantidad de clientes, etc...
- *Distribución de productos en el carrito*. Los clientes que pasan por las cajas llevan un carrito de la compra con productos. Cada cliente dispone de un único carrito. El número de productos de cada carrito sigue una distribución exponencial cuyo valor medio depende también de múltiples factores: inicio o final de mes, renta media del hogar en la zona, etc.
- *Distribución del precio de cada producto en el carrito*. Cada producto tiene un precio diferente que seguirá una distribución exponencial cuya media depende de muchos factores: vísperas de celebraciones (navidad, fin de año), número de productos de marca blanca que comercializa el supermercado, etc.

SmartTPV necesita simular el funcionamiento de las zonas de cobro que diseña. En la simulación se deberá fijar el tiempo media de llegada de clientes a la caja, el número medio de productos de cada carrito y el precio medio de cada producto en el carrito.

SmartTPV desea estudiar el funcionamiento de la zona de cobro abriendo y cerrando cajas durante la simulación. En concreto desea que a cada hora de simulación se muestren los tamaños de las colas y poder decidir abrir o cerrar cajas (sólo si no hay clientes a la espera en la caja, claro). Al finalizar la simulación se tiene que indicar, por cada caja: el tiempo medio de espera de los clientes, el tamaño máximo de la cola, el número de productos y la cifra de ventas que ha cobrado la caja. También se desea conocer el tiempo medio de espera en el supermercado, el total de clientes servidos, los productos cobrados y las ventas totales.

Aunque el trabajo le va a proporcionar poco dinero, tras escuchar esa descripción recuerda el simulador de la autopista que programó hace poco. La nueva aplicación parece muy similar por lo que parece que podría realizar tal proyecto en un plazo breve tiempo y con costes muy reducidos (lo que supone un apreciable margen de beneficio).

2 ENTREGAS

En todas las entregas se podrá configurar el tiempo de simulación. En todas las pruebas el tiempo de simulación se fijará en 3 horas (10.800 segundos).

2.1 Primera entrega. Una única caja con tráfico constante y productos constantes

La primera versión del simulador dispondrá de una zona de cobro con una única caja. El tráfico será constante: un carrito cada minuto. Cada carrito únicamente cargará un producto cuyo precio será de un euro.

Piense cuáles deben ser los resultados esperados tras tres horas de simulación. Escriba el código de los casos de prueba necesarios para verificar que el simulador hace lo que debe.

2.2 Segunda entrega. Una única caja con tráfico constante y carritos con número de productos variables y precios variables.

La segunda versión del simulador seguirá considerando una zona de cobro con solo una caja. El tráfico será constante: un carrito cada minuto. Sin embargo, tanto el número de productos como el precio de los mismos será variable.

Piense cuáles deben ser los resultados esperados tras tres horas de simulación para valores bajos, altos y medios de los parámetros. Escriba el código de los casos de prueba necesarios para verificar que el simulador hace lo que debe.

2.3 Tercera entrega. Una zona de cobro con diez cajas y tráfico variable.

La tercera versión del simulador añadirá a la zona de cobro las 10 cajas previstas. El tráfico será variable, al igual que el número de productos o el precio de los mismos será variable.

Piense cuáles deben ser los resultados esperados tras tres horas de simulación para valores bajos, altos y medios de los parámetros. Escriba el código de los casos de prueba necesarios para verificar que el simulador hace lo que debe.

2.4 Entrega Final. Una zona de cobro con diez cajas con tráfico variable y productos variables y posibilidad de abrir o cerrar cajas.

La última versión del simulador es el producto software final que entregará al cliente.

3 DESARROLLO

Para implementar el simulador del supermercado partirá del código del simulador de la autopista. En miAulario podrá encontrar una versión totalmente funcional del simulador de autopistas.

Al finalizar la práctica deberá proporcionar **un único proyecto de Netbeans con al menos cuatro clases ejecutoras**, una para cada entrega. Además, deberá acompañar el código de los pruebas automatizadas solicitadas, para ello tiene dos alternativas:

- ❏ Escribir nuevas clases ejecutoras usando la consola como elemento de visualización de los casos de prueba.
- ❏ Escribir clases de tests con ayuda de JUnit usando la ventana de tests de netbeans como elemento de visualización de los casos de prueba.

Para poder entregar el simulador de forma iterativa, una posible solución es escribir diferentes tipos de tráfico, zona de cobro, carritos y productos. De este modo para la primera entrega escribirá una clase que implementa un tráfico constante, una zona de cobro con solo una cabina, un carrito con solo un producto y un producto cuyo precio sea fijo. En la última entrega utilizará otras clases: el tráfico será variable, la zona de cobro tendrá 10 cabinas y permitirá abrir y cerrar cajas, los carritos tendrán un número variable de productos, etc.

Es usted libre para codificar las entregas cómo crea más conveniente. A continuación, a modo de consejo, se le ofrece una serie de pautas, patrones y prácticas que harán su código más mantenible, extensible y testable. La clave para no “emborronar” su código en cada nueva entrega es el polimorfismo, para ello será imprescindible que derive clases a partir de un supertipo del que heredar o implemente clases a partir de un interface.

3.1 Adecuar la autopista al supermercado

Lo primero que deberá hacer es modificar el código para adecuarlo al supermercado:

- ❏ La zona de cobro es muy similar al peaje.
- ❏ Las cajas son muy similares a las cabinas de la autopista.
- ❏ Los vehículos son similares a los carritos de la compra (incluso en lo referente a cómo llegan). Aunque un carrito contiene varios productos con diferentes precios.
- ❏ El resto de clases responden a la idea de simulación por eventos discretos, por lo que también serán muy similares

Es posible que sea necesario crear alguna nueva clase.

3.2 Separar la vista del negocio

Para implementar algunas de las pruebas que se proponen es imprescindible, además de mostrar los resultados en la consola, poder chequearlos en el código. Para ello es posible crear un objeto donde persistir los resultados. El método "estado" de la clase "Peaje" (ahora renombrada como "ZonaCobro") no debe "*escupir*" los resultados en la consola. En lugar de "*printear*" los resultados, el método debe devolverlos encapsulados en un objeto.

Para ello le propongo usar una clase similar a esta:

```
public class EstadoZonaCobro {
    public ArrayList<ResultadoCaja> resultados;
    //ToDo resto de resultados
    public EstadoZonaCobro() {
        resultados = new ArrayList<ResultadoCaja>();
    }
}
```

Modifique el método *estado()* de `ZonaCobro` para devolver un objeto de tipo `EstadoZonaCobro`:

```
public EstadoZonaCobro estado(){
//toDo
}
```

De esta forma, algunas pruebas pueden instanciar el simulador y verificar que los resultados obtenidos en la simulación son los que deben. Por ejemplo:

```
simulador.simular();
EstadoZonaCobro resultadoSimulacion = simulador.obtenerResultados();
if(resultadoSimulacion.TotalCarritos>900 && resultadoSimulacion.TotalCarritos<1100){
   System.out.println("OK");
} else {
   System.out.println("NOK!");
}
```

Recuerde que el producto software solicitado debe seguir mostrando los resultados por la consola. Puede crear una clase `VistaSimuladorConsola` con un método `mostrarEstado(EstadoZonaCobro resultado)` encargada de imprimir el objeto que encapsula los resultados en la consola. Puede añadir a esta clase todas las interacciones del usuario con la consola.

3.3 Inversión de Control.

Para implementar pruebas automatizadas es también imprescindible introducir en el simulador las condiciones iniciales de la prueba: el tipo de tráfico, el número de cajas en la zona de cobro, el número de productos en cada carrito o el precio de los mismos. Si estos parámetros de la simulación son introducidos por el usuario en la consola, no es posible automatizar la prueba.

Una posible solución para por “inyectar las dependencias”. Además esta técnica nos ofrece un mecanismo para lograr la “inversión de control” requerida. Observe el siguiente código:

```
// La ClaseA está fuertemente acoplada a la ClaseB
public class ClaseA {
    public void doSomething() {
        ClaseB objetoB = new ClaseB();
        // invoca algún método del objetoB
    }
}
public class ClaseEjecutora {
     public static void main(String[] args){
        ClaseA miObjetoA = new ClaseA();
        miObjetoA.doSomething();
    }
}
```

La clase ejecutora instancia un objeto de la ClaseA, al ejecutar el método doSomething(), el objeto de la clase A inevitablemente instanciará un objeto de la ClaseB. Desde la clase ejecutora no es posible evitarlo.

Ahora observe el siguiente código:

```
// La ClaseA está suavemente acoplada a la ClaseB
public class ClaseA {
   private ClaseB objetoB;
   public ClaseA(ClaseB objetoB){
      this.objetoB = objetoB
   }
    public void doSomething() {
        // invoca algún método del atributo objetoB
    }
}
public class ClaseEjecutora {
     public static void main(String[] args){
        ClaseB miObjetoB = new ClaseB();
        ClaseA miObjetoA = new ClaseA(miObjetoB);
        miObjetoA.doSomething();
    }
}
```

Ahora es la clase ejecutora la que decide la dependencia del objeto de la Clase A. Por supuesto siempre dentro de unas restricciones. La clase A no puede inyectar cualquier objeto, solo puede “inyectar” un objetos que cumplan el “contrato” que establece la clase B. Es decir, solo puede inyectar diferentes “tipos” de B. En el siguiente código se muestra el “poder” del polimorfismo.

```
public class ClaseBConSabor extends ClaseB {
   // sobreescribe algún método de la clase base ClaseB
}
public class ClaseEjecutora {
     public static void main(String[] args){
        ClaseB miObjetoBConDiferenteSabor = new ClaseBConSabor();
        ClaseA miObjetoA = new ClaseA(miObjetoBConDiferenteSabor);
        miObjetoA.doSomething();
    }
}
```

Desde la clase ejecutora es posible cambiar el comportamiento del algoritmo sin modificar ni la clase A, ni la clase B. El programador en lugar de modificar el código ya existente, escribe nuevo código en la "clase B con sabor". En la "Raíz de Composición" (Composition Root) de la aplicación deberá "ensamblar" las piezas. La "Raíz de Composición" es la clase ejecutora y las piezas son las instancias de las clases (objetos).

> La técnica de "inyectar" las dependencias de las clases en el constructor, en vez de permitir que las propias clases instancien las dependencias dentro de sus métodos, se llama "**Inyección de Dependencias**".
>
> Cuando el código permite extender su funcionalidad (dotándole de nuevo comportamiento) sin modificar las clases ya existentes, sino escribiendo nuevas clases se dice que el código está cerrado a cambios pero abierto a la extensión (**Principio Open/Closed**, la "O" de SOLID).
>
> La capacidad del código de extender su comportamiento modificando únicamente la clase ejecutora, ensamblando diferentes "sabores" de las dependencias, se llama **"Inversión de Control"**.

Para dotar de "inversión de control" a nuestro código debemos aflorar las dependencias de una clase a otras clases en el constructor. Por ejemplo, la clase "Simulador" depende de "Reloj" y de "Controlador". En el constructor del simulador se inicializan atributos con instancias de las clases Reloj y Controlador a través del operador "new". Decimos entonces que "Simulador" está fuertemente acoplado a "Reloj" y a "Controlador".

```
public class Simulador {
    public int tSim; // tiempo de simulación
    public Reloj miReloj;
    public Controlador miCont;

    /**
     * Constructor del Simulador
     * Fija el timepo de simulación, crea un reloj a cero y un nuevo controlador
     * @param tSim tiempo de simulación
     */
    public Simulador(int tSim){
        this.tSim = tSim;
        miReloj = new Reloj();
        miReloj.aCero();
        miCont = new Controlador(miReloj);
    }
```

En lugar de ello, para dotar a nuestro código de "inversión de control" simplemente deberá "inyectar" estas dependencias en el constructor.

```
public class Simulador {

    private int tSim;
    private Reloj reloj;
```

```
    private ControladorSupermercado controlador;

    public Simulador(int tiempo, Reloj reloj, ControladorSupermercado controlador) {
        this.tSim = tiempo;
        this.reloj = reloj;
        this.controlador = controlador;
    }
```

El mismo tratamiento puede darse a la clase Controlador

```
public class ControladorSupermercado {

    public ZonaCobro miZona;
    public Trafico miTrafico;

    public ControladorSupermercado(Reloj reloj, Trafico trafico, ZonaCobro zona) {
        this.miZona = zona;
        this.miTrafico = trafico;
        this.miTrafico.proximaLLegada(reloj);
    }
```

ZonaCobro puede instanciar las cajas, de la misma forma que el “Peaje” instancia cabinas. No la modificaremos.

La primera entrega dispondrá de una zona de cobro con una única caja. El tráfico será constante: un carrito cada minuto. Cada carrito únicamente cargará un producto cuyo precio será de un euro. Para ello necesitará construir un tipo de Tráfico que en lugar de calcular el tiempo de llegada de los carritos de forma aleatoria lo haga de forma constante. Una posible solución es crear una nueva clase “TraficoFijo” que heredará de Tráfico. TraficoFijo sobreescribirá el método “`proximaLLegada()`” de la clase base.

```
public class TraficoFijo extends Trafico {
    public TraficoFijo(int tiempoMedioLLegada) {
        super(tiempoMedioLLegada);
    }
    @Override
    public void proximaLLegada(Reloj reloj){
        tLlegada = reloj.TiempoAhora()+this.tMedio;
    }
}
```

Para hacer funcionar este código deberá modificar el ámbito del atributo “tMedio” de la clase base para hacerlo accesible en la clase derivada.

De esta forma la “Raiz de Composición” del simulador puede ser algo así:

```
public static void main(String[] args){
        Reloj reloj = new Reloj();
        Trafico traficoCadaMinuto = new TraficoFijo(60);
        ZonaCobro zonaCobroConUnaCajaYCarritosConUnProducto = new ZonaCobroBasica(1);

        ControladorSupermercado controlador = new ControladorSupermercado(
                reloj,
                traficoCadaMinuto,
```

```
                zonaCobroConUnaCajaYCarritosConUnProducto);

        Simulador simulador = new Simulador(10800, reloj, controlador);
        simulador.simular();
        EstadoZonaCobro resultadoSimulacion = simulador.obtenerResultados();
        VistaConsolaSimulador vista = new VistaConsolaSimulador();
        vista.mostrar(resultadoSimulacion);
}
```

3.4 “Factoría” para crear los carritos.

Por último, para implementar algunas entregas es necesario modificar el tipo de carritos que la zona de cobro crea en el método “guardarCola”. En el código original de la “autopista” el carrito es creado por la zona de cobro, algo así:

```
public void guardarCola(int nCab, Reloj reloj){
        Caja caja = misCajas.get(nCab);
        Carrito carrito = new Carrito();
        carrito.LlenarCarritoConProductos();
        caja.encolarCarrito(reloj, carrito);
    }
```

La Zona de Cobro debe ser capaz de crear un nuevo carrito cuando así se determine. Para poder indicar en la “Raíz de Composición” del simulador qué tipo de carrito debe crear la zona de cobro puede hacer uso de uno de los patrones de diseño más habituales: el patrón Factory. Este patrón de construcción propone el diseño de clases a las que delegar la creación de instancias. En nuestro caso particular es posible crear dos clases que implementen un mismo contrato. Este podría ser un ejemplo de “contrato”:

```
public interface IFactoriaCarritos {
    Carrito Create();
}
```

Una clase (*FactoriaCarritosVariable*) creará carritos que se llenarán con un número variable de productos, otra clase (*FactoriaCarritosFija*) creará carritos con un número fijo de productos. La zona de cobro podrá recibir como dependencia en su constructor el contrato de dichas clases. de este modo en la “Raíz de Composición” podrá elegir, en función de la prueba que deseemos realizar, si inyectamos la factoría que crea carritos con un número variable de productos o con un número fijo. Por ejemplo:

```
public static void main(String[] args){
        Reloj reloj = new Reloj();
        Trafico traficoCada10segundos = new Trafico(10);
        FactoriaCarritosAleatorios factoria = new FactoriaCarritosAleatorios(10);
        ZonaCobro zonaCobroCon10CajasY10Productos = new ZonaCobroFija(10,factoria);
        ControladorSupermercado controlador = new ControladorSupermercado(
                reloj,
                traficoCada10segundos,
                zonaCobroCon10CajasY10Productos);
        Simulador simulador = new Simulador(10800, reloj, controlador);
        simulador.simular();
        EstadoZonaCobro resultadoSimulacion = simulador.obtenerResultados();
```

```
    }
```

4 MEMORIA

La memoria del proyecto debe ser un documento Word/PDF con las siguientes secciones:

- ❏ Introducción que explique la evolución/cambios realizados al proyecto de la autopista para llegar al del supermercado.
- ❏ Explicar la evolución del proyecto a lo largo de las 4 entregas empleando 4 diagramas de clase detallados (realizados con draw.io u otra herramienta similar) para explicar el diseño.
- ❏ Conclusiones sobre las ventajas del diseño separando la vista del negocio

5 CRITERIOS DE CORRECCIÓN

Para aprobar es necesario:

- ❏ Entregar el proyecto de Netbeans con las fuentes y la memoria, a través de MiAulario antes de la fecha límite (las 4 entregas se envían en una misma Tarea).
- ❏ El proyecto entregado debe ser compatible con el entorno del laboratorio, Java 8 y Netbeans 8.2 (Categories: Java, Projects: Java Application) , sin errores de formato, compilación o de descompresión.
- ❏ Código fuente con las 4 entregas solicitadas (evolución del código entregado de la práctica de la Autopista) funcionando y con las pruebas.
- ❏ Memoria del proyecto con el formato y las secciones solicitadas.

Se valorará el grado de aplicación de las directrices de diseño dadas y se tendrá en cuenta:

- ❏ La aplicación de los principios de POO en el proyecto.
- ❏ La forma de separar la vista del negocio.
- ❏ Cumplir con los principios de diseño: Inyección de dependencias, Inversión de control y principios SOLID.
- ❏ Seguir las recomendaciones sobre "código limpio" (clean code).