Facultad de Ingeniería
Universidad Nacional de Mar del Plata

Sistema de Contratación de Servicios de Seguridad

Trabajo grupal final de la materia Programación III de la Facultad de Ingeniería de la Universidad de Mar del Plata

30/04/2023

—

Integrantes:

Bustamante, Joaquín Mendez

Juliano, Francisco

Romanazzi, Bautista

Vallone, Franco

Introducción

El proyecto consiste en el análisis y creación de un sistema para gestionar contrataciones de servicios de monitoreo y seguridad.

Requerimientos

Requerimientos del sistema

1. Agregar abonados al sistema
2. Quitar abonados del sistema
3. Distinguir entre abonados físicos y jurídicos
4. Clonar abonados físicos.
5. Dos abonados no pueden tener el mismo DNI
6. Agregar contratos a un abonado.
7. Quitar contratos de un abonado.
8. No permitir agregar un contrato con el domicilio o identificador de uno ya existente.
9. Agregar servicios adicionales a un contrato (cámaras, botones antipánico y móviles)
10. Facturar abonados con descuentos por medio de pago (Cheque, efectivo y tarjeta)
11. Clonar facturas.
12. Generar un reporte de todas las facturas de un abonado.
13. Distinguir entre contratos de vivienda y comercio.
14. Calcular el precio de un contrato por promoción (Dorada, platino y sin promoción)
15. Agregar técnicos
16. Quitar técnicos
17. Asignar técnicos a abonados
18. Manejar cambios de estado para personas físicas, Moroso, Con Contrataciones y Sin Contrataciones
19. Avanzar el tiempo
20. Generar facturas para todos los abonados al avanzar el mes
21. Gestionar el sistema a través de una interfaz gráfica

22. Persistir el sistema al finalizar la jornada

Requerimientos del código

1. Usar el lenguaje de programación Java.
2. Utilizar el paradigma de la programación orientado a objetos.
3. Aplicar la metodología de diseño por contrato.
4. Utilizar una arquitectura de tres capas.
5. Aplicar el patrón Double-Dispatch para calcular descuentos por promociones
6. Aplicar el patrón Decorator al facturar por medio de pago.
7. Aplicar el patrón Observer - Observable para notificar cambios en los servicios técnicos.
8. Aplicar el patrón State para el estado de los abonados físicos

Uso de patrones de diseño

Para implementar el sistema de manera eficiente, se han aplicado ciertos patrones de diseño que permiten estructurar el código de forma clara y eficiente. A continuación, se describen los patrones utilizados y el por qué se han utilizado:

Patrón Decorator

El patrón decorator permite extender la funcionalidad de un objeto. Se utilizó para aplicar descuentos por medio de pago al facturar un abonado. Las dos funciones principales que extiende es el cálculo del pago con el descuento aplicado, y la creación de una factura con el descuento aplicado.

Luego, se consideró la mejor opción abstraer el uso del decorator de la funcionalidad del sistema, ya que no es el mejor patrón para un uso directo. Para esto, los decoradores se crean solamente en el factory:

```
public static IFactura getFactura(String concepto, double subtotal, double
valorNeto, String medioDePago, LocalDate fecha) {
        IFactura respuesta = null;
        IFactura encapsulado = new Factura(concepto, subtotal, valorNeto,
```

```
fecha);

        if (medioDePago.equalsIgnoreCase("EFECTIVO"))
            respuesta = new PagoEfectivoDecorator(encapsulado);
        else if (medioDePago.equalsIgnoreCase("CHEQUE"))
            respuesta = new PagoChequeDecorator(encapsulado);
        else if (medioDePago.equalsIgnoreCase("TARJETA"))
            respuesta = new PagoTarjetaCreditoDecorator(encapsulado);
        else
            respuesta = encapsulado;

        return respuesta;
    }
```

De esta manera también se logra un desacoplamiento de la interfaz con la implementación.

Patrón Factory

El Patrón Factory permite encapsular la creación de objetos en una clase separada, lo que facilita la creación de objetos complejos y evita la duplicación de código, es por esto que se ha elegido implementar este patrón para implementar la creación de abonados, contratos y facturas. Las clases que lo implementan serían AbonadoFactory, ContratoFactory y FacturaFactory.

Patrón Double-Dispatch

Se ha utilizado el patrón Double-Dispatch para implementar las promociones que ofrece la empresa a sus servicios. Este patrón permite elegir la acción correcta a realizar al momento de ejecución, la decisión de qué método se va a ejecutar se hace a partir del objeto receptor del mensaje. Para ello se ha creado la clase abstracta Promocion que implementa la interfaz IPromocion, a su vez también se crearon las clases PromocionDorada, PromocionPlatino y SinPromocion que son clases concretas que extienden de Promocion.

Patrón Singleton

Cuando creamos la clase Sistema, utilizamos el patrón Singleton para asegurarnos de que sólo exista una instancia de la clase en todo momento. De esta manera, evitamos la creación innecesaria de otras instancias.

Dentro de la clase Sistema, se encuentra el ArrayList con todos los Abonados donde podremos realizar diferentes operaciones con ellos.

Patrón Observer - Observable

Se usó el patrón observer-observable para notificar cambios en los servicios técnicos. Esto permite a la vista actualizar el progreso del técnico de manera continua y con bajo acoplamiento.

Patrón State

Se utilizó el patrón state para manejar los cambios de estado para los abonados físicos.

Implementación de herramientas

Excepciones

Las excepciones son situaciones que no pueden ser manejadas por el sistema. Se lanzan excepciones en los siguientes casos:

- Al agregar un abonado al sistema con DNI repetido
- Al agregar un contrato al sistema con domicilio repetido.
- Al facturar un abonado que no tiene contratos.
- Al intentar operar sobre un abonado que no existe.

Estas excepciones deben ser manejadas apropiadamente por el programa que lo utiliza, mostrando mensajes de error en una interfaz de usuario por ejemplo.

Interfaces

Las interfaces son utilizadas para mejorar la abstracción y el polimorfismo, permiten la definición de un contrato entre las clases que las implementan y la creación de código más reutilizable. Lo utilizamos para obtener más flexibilidad en las clases promoción, abonado, servicio alarma, sistema y en los decorator.

Serialización

La serialización es un mecanismo de java que permite persistir el estado de los objetos. Se implementó la persistencia a través de archivo binario para el singleton de Sistema.

Al implementar la serialización en un Singleton, hay algunas consideraciones adicionales a tener en cuenta debido a la naturaleza única de este. Fue necesario crear un DTO especial solo para la persistencia. Al persistir, se crea el DTO a partir del estado del sistema, y se almacena en archivo binario. Al depersistir, se crea el DTO a partir de los datos del archivo, y se instancia el sistema con sus atributos.

Esto permite una serialización fácil de las clases usadas por este; solo es necesario que implementen la interfaz Serializable, y java hace el resto.

DTO y DAO

Se utilizó el patrón DTO para el transporte de información desde la vista al controlador. En la vista, se capturan los datos ingresados por el usuario, y se almacenan en un DTO. Luego, el controlador recibe el DTO y lo procesa para interactuar con el modelo. Esto ayuda a

tener una separación clara de responsabilidades: La vista no crea nuevas entidades, solo pasa información.

Concurrencia

Se utilizó la concurrencia y los recursos compartidos para simular las visitas de los técnicos a los abonados.

El técnico es el recurso compartido; solo puede visitar a un abonado a la vez. Esto se implementó con wait y notify:

```
    public synchronized void asignarAbonado(IAbonado abonado) throws
InterruptedException {
        while (!disponible) {
            wait();
        }

        this.disponible = false;
        notifyAll();
    }

    public synchronized void liberar() {
        this.disponible = true;
        notifyAll();
    }
```

De esta manera, si algún abonado requiere la visita de un técnico que ya está trabajando, debe esperar a que termine.

Luego, para iniciar un servicio técnico, se crea un Thread con el Runnable ServicioTecnico, que utiliza el recurso compartido:

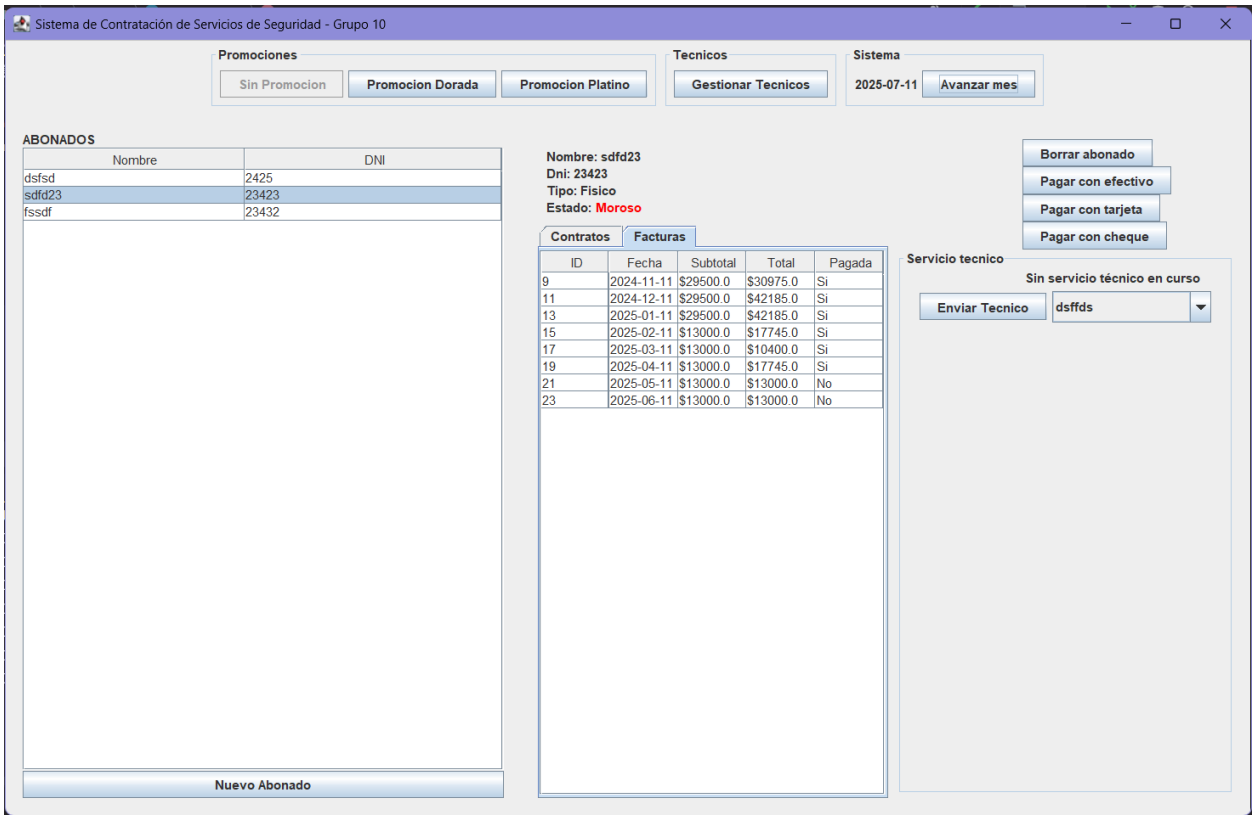
```
public void run() {
    try {
        this.tecnico.asignarAbonado(abonado);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    //función que simula el progreso del service.
    this.avanzar();
    this.tecnico.liberar();
}
```

Si el técnico no está disponible, espera hasta que lo esté.

Java Swing

Se utilizó la librería Java Swing para crear la interfaz gráfica.

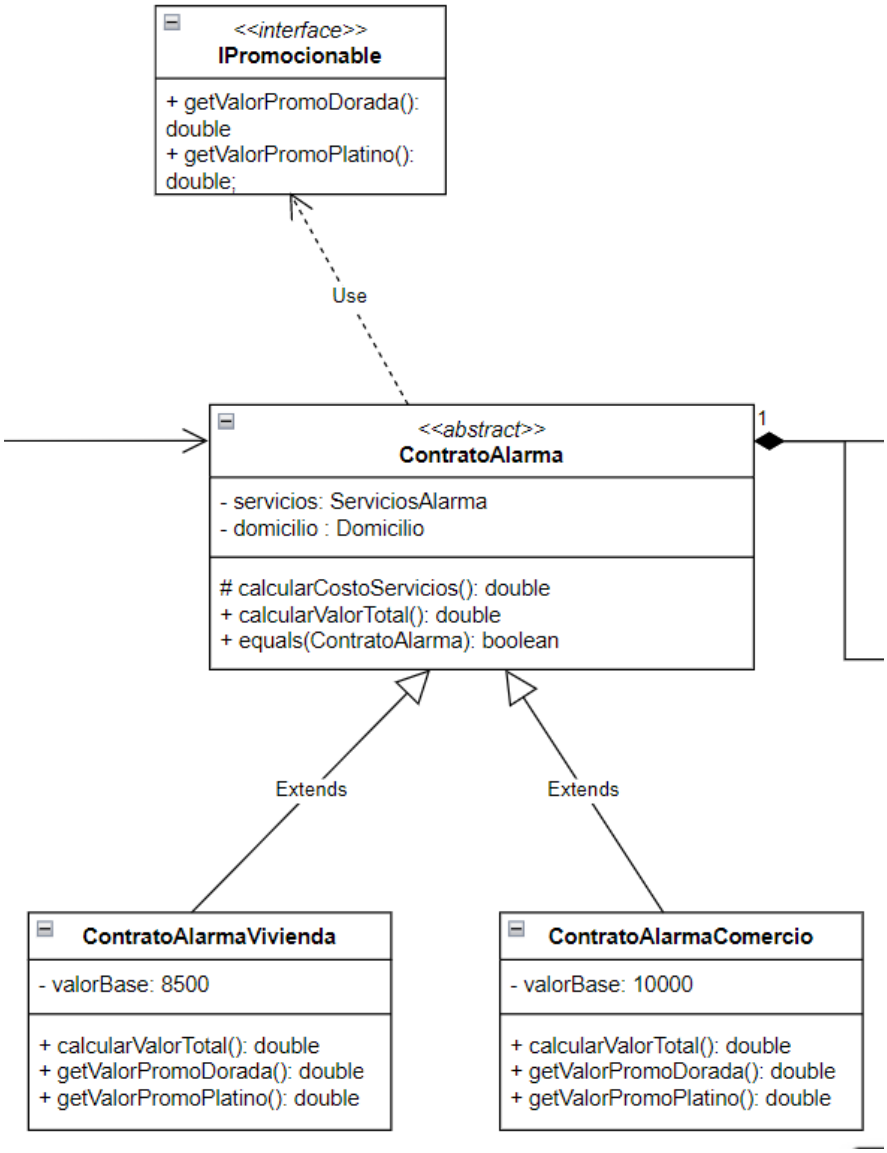
Swing es una librería basada en AWT que provee componentes comunes como botones, combobox, layouts etc. Esto da un alto grado de abstracción para facilitar la construcción de la vista.



Solución a problemas encontrados

Aplicación de double-dispatch para las promociones

En la primera iteración del diseño, se agregó el cálculo del descuento a cada subclase de Contrato, de la siguiente manera:



Sin embargo, esto tuvo el problema de que al aplicar el descuento era necesario discernir entre el tipo de promoción:

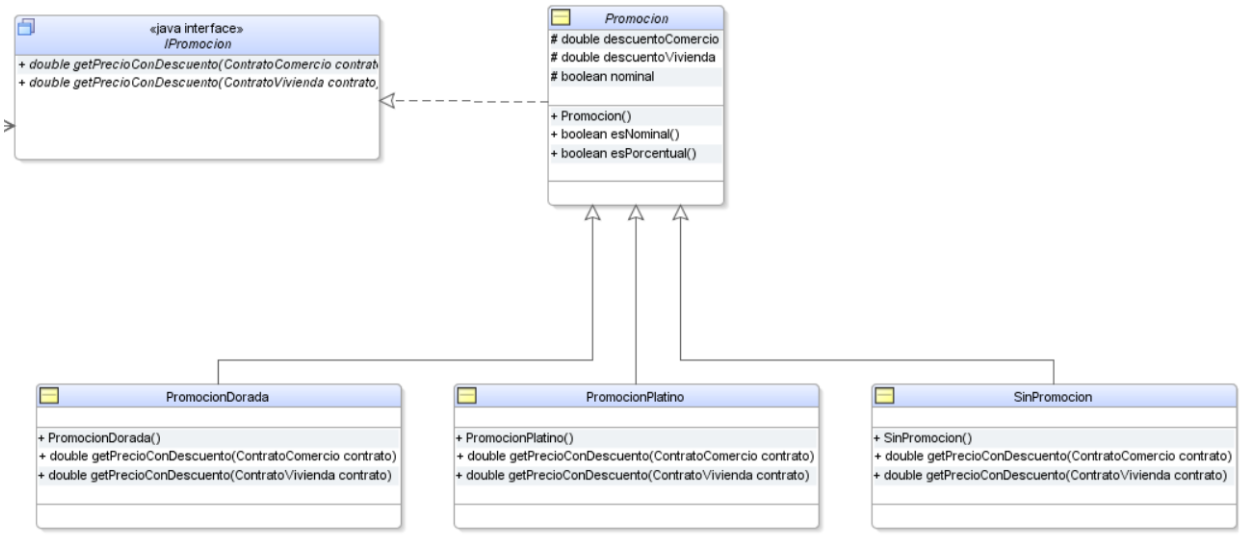
```
Abonado.java
public double obtenerValorContratos(Promo promo) {
	double total = 0;
	Iterator<ContratoAlarma> iterator = contratos.iterator();

	while (iterator.hasNext()) {
		ContratoAlarma contratoActual = iterator.next();
		if (promo.getEsPromoDorada()) {
			total += contratoActual.getValorPromoDorada();
		} else if (promo.getEsPromoPlatino()) {
			total += contratoActual.getValorPromoPlatino();
		} else {
			total += contratoActual.obtenerValorTotal();
		}
	}

	return total;
}
```

Esta versión no sería escalable si se agregan muchos tipos de contrato, ya que habría que agregar más condiciones por cada uno.

Para solucionar esto, se decidió aplicar el cálculo del descuento en la promoción en sí, aprovechando la sobrecarga de métodos para que cada promoción calcule el valor del contrato:



Entonces, desde el contrato se puede obtener el precio total de la siguiente manera:

```
ContratoVivienda.java
public double getPrecio(IPromocion promocionActual) {
    return promocionActual.getPrecioConDescuento(this);
}
```

Creación de la clase Factura

Inicialmente habíamos propuesto que la clase Factura recibiera en su constructor los parámetros necesarios para la creación de la misma como el abonado, el dni del abonado, el nombre del abonado, la cantidad de cámaras contratadas y varios parámetros adicionales. A continuación se deja una muestra de lo que serían los atributos de ésta clase que anteriormente pensábamos usar:

```
private final int cantCamaras;
private final int cantBotones;
private final int cantMoviles;
private final double precioTotal;
private final double precioSinDescuento;
private final int cantServiciosVivienda;
private final int cantServiciosComercio;
private final String dniAbonado;
private final String nombreAbonado;
private final int id;
```

Luego, a medida que íbamos implementando nuevas funcionalidades y patrones de diseño, nos vimos obligados a cambiar la implementación de la clase Factura, para que recibiera como parámetro un precio neto y el sub total como también un String concepto (este último es la unión de varios String que cada clase aporta para finalmente crear la factura con todos los detalles que se requieren). Debajo se deja parte del código actual:

```
Factura.java
public Factura(String concepto, double subtotal, double valorNeto) {
   this.valorNeto = valorNeto;
   this.concepto = concepto;
   this.subtotal = subtotal;
   this.id = numero++;
```

```
}

public String getDetalle() {
   StringBuilder detalle = new StringBuilder();

   detalle.append("FACTURA N°" + id + "\n\n");
   detalle.append(concepto + "\n");
   detalle.append("SUBTOTAL: $" + subtotal + "\n");
   detalle.append("TOTAL: $" + valorNeto + "\n");

   return detalle.toString();
}
```