

# **Trabajo Práctico Final 2023**

**Integrantes:**

Florio Francisco

Frangolini Matias

Iarritu Pedro

Porfilio Bernardo

## **Planteo del algoritmo**

Se comenzó realizando un diagrama de las clases que se iban a necesitar para llevar a cabo el funcionamiento del sistema, con el objetivo de tener una mejor perspectiva del problema, incluyendo sus métodos y atributos, estableciendo relaciones de herencia, agregación y composición entre las mismas. También se listaron las precondiciones y postcondiciones de cada método.

## **Cálculo del precio de contratación (Double Dispatch)**

Para calcular el precio de cada contratación se crearon dos subclases de la superclase Contratación, las cuales modificaron el valor base del precio de la misma, finalmente para obtener el precio total se le sumó al valor base el precio de agregado de otros servicios (monitoreo de alarmas, botón antipánico, móvil de acompañamiento) los cuales fueron representados como atributos de tipo entero y booleano en la superclase.

Finalmente se aplicaron las promociones correspondientes (dorada y platino), mediante el Patrón Double Dispatch, para evitar el uso de numerosos if y depositar el trabajo de aplicar la promoción en el receptor (contratación), creando dos interfaces, una Promocionable, la cual fue implementada por la clase Contratación y la otra IAplicaPromo implementada por las clases PromoDorada y PromoPlateada.

Cada promoción fue obligada a implementar el método aplicarPromocion (ya que el mismo fue definido por su interfaz), que poseía como parámetro a un elemento promocionable, el cual tenía dos métodos, uno para aplicarse la promo dorada y otro para la platino (también definidos en su interfaz). De esta forma al llevarse a cabo el método aplicarPromocion, el encargado de generar el descuento fue la contratación.

## **Cálculo pago total de servicios (Patrón Decorator)**

Se creó una clase Factura que implementa una interfaz llamada IFactura, esta clase posee un atributo de tipo Cliente y otro que representa el precio total de todas las contrataciones del cliente. Para obtener este precio total se delega su cálculo al cliente, el cual en un método suma el precio de cada contratación, teniendo en cuenta sus descuentos por promociones y en el caso de ser un cliente jurídico sobreescribiéndolo y realizando un descuento de 50% a partir de la tercera contratación. Como el precio total se verá afectado por el medio de pago, es decir será decorado dependiendo este, creamos una super clase llamada DecoratorMedioDePago que también implementará IFactura, la cual encapsulará un elemento de tipo IFactura (en este caso una factura). Esta super clase será extendida por otras tres, las cuales representarán cada medio de pago, sobreescribiendo el método que obtiene el pago total de un cliente, aplicando al mismo su respectivo descuento o incremento.

## **Creación de Facturas (Patrón Factory)**

Se creó un cliente en el main (de un determinado tipo) y posteriormente con el fin de obtener un encapsulamiento en la creación del objeto, fue pasado como parámetro junto a un string que representa el medio de pago al método getFactura de un objeto de tipo FacturaFactory. Dentro del método que genera una factura creamos la misma y luego dependiendo del string ingresado el decorador del medio de pago, el cual en su constructor incorpora su encapsulado (en este caso la factura recientemente creada).

## Clonación de factura

La interfaz `IFactura` se extenderá de `cloneable` y redefinirá el método `clone` indicando que puede propagar la excepción `CloneNotSupportedException`. Como la clase `DecoratorMedioDePago` implementa esta interfaz y la clonación no depende del medio de pago, se realizó la sobreescritura del método `clone` en la super clase, la cual retornará el clonado de su encapsulado, además de poseer la posibilidad de propagar la excepción. Para clonar el encapsulado, el cual es una factura, en la clase `factura` también sobreescribirá el método `clone`, donde se realizará una clonación profunda de su cliente (el cual también se extenderá de `cloneable`), para evitar la duplicación de referencia a un objeto y que al modificar sus atributos (si posee) se modificará también en el clon.

La clase cliente sobreescribirá el método `clone`, aplicando clonación profunda a sus dos atributos de tipo lista (ya que son objetos) clonando inicialmente las listas, limpiandolas y luego cada elemento por separado (la clase a la que pertenecen estos elementos implementará `cloneable` y tendrá su método para clonarse) y poseerá la posibilidad de lanzar la excepción. La subclase `ClienteJuridico` extendida de `Cliente` no puede ser clonada nunca, por ende sobreescribirá el método `clone` de `Cliente`, arrojando siempre la excepción `CloneNotSupportedException`, junto a un cartel que indica el tipo de excepción.

Las clonaciones profundas realizadas se implementaron en los casos que hay relación de composición entre clases y así evitar la doble referencia.

La excepción `CloneNotSupportedException` será atrapada en un bloque `catch` en el `main`.

## Excepciones, Condiciones y Aserciones

Una vez establecidas todas las precondiciones para poder llevar a cabo el método y postcondiciones para describir el resultado obtenido, implementamos el uso de aserciones con el fin de comprobar que se cumplan, verificando los valores de las variables. Con esto logramos, en tiempo de programación, analizar en base al contrato de cada método y en cada clase cuáles serían las excepciones a lanzar, para que habiéndose cumplido las precondiciones, el comportamiento del método fuera el correcto y se puedan cumplir las postcondiciones, controlando los errores y

permitiéndonos tratarlos y solucionarlos en un espacio distinto dependiendo de las características de cada excepción.

Actualmente el sistema posee las siguientes excepciones específicas:

- ContratacionInvalidaException
- DomicilioInvalidoException
- DomicilioNoPerteneceAClienteException
- DomicilioNuloException
- DomicilioYaExisteException
- ClienteInvalidoException
- FactoryInvalidoException
- FacturaInvalidaException
- MetodoDePagoInvalidoException

Si bien todas las excepciones son lanzadas en clases, métodos y por motivos distintos, hasta el momento serán propagadas hasta el método main, ya que son errores del ingreso de datos del usuario, que podrán ser atrapados y solucionados al momento de tener una ventana con la que se decida interactuar, lo que permitirá el reingreso de los datos o la búsqueda de otra solución.