2023

# Trabajo Práctico Programación III

PARTE 1

RAMIRO CARDELLI, JOSEFINA FRASCA, NICOLAS HERRERA, JOAQUIN MARTINEZ

## Introducción

El objetivo de este trabajo consiste en programar un sistema para gestionar contrataciones de servicio de monitoreo y seguridad, así como también la gestión de abonados y facturación.

La implementación del mismo, ha requerido la puesta en práctica de los conocimientos adquiridos en relación a la programación orientada a objetos brindados por la cátedra. Entre ellos se destacan: la herencia, el polimorfismo, el principio de Liskov, el uso de patrones (Singleton, Factory,, Decorator, Double Dispatching), el uso de excepciones, etc.

## Desarrollo del programa

En primera instancia, los participantes analizamos y planteamos la estructura del programa en formato de papel, en razón de que luego nuestro programa tuviera una base sólida, que nos librará de tener que rehacerlo por posibles errores. Es decir, implícitamente definimos el contrato de nuestro programa, donde establecimos el conjunto de precondiciones, postcondiciones e invariantes de cada método, las excepciones que podrían lanzarse, los asserts en el proceso de desarrollo, etc.

Una vez con el contrato definido, se procedió a realizar la implementación del programa secuencialmente, siguiendo cada una de las etapas del proceso de desarrollo.

## Aspectos del programa

Entre los aspectos más relevantes del programa, podemos encontrar la implementación de tres clases principales: la clase Factura, Abonado y Contratación. Dichas clases son el núcleo del programa y moldean la gestión del sistema.

Algo que nos pareció muy importante, fue la correcta creación de dichas clases; las cuales se crean permitiendo que en un futuro quieran agregarse otros tipos extendidos de las mismas.

En particular, la clase Factura para crearse necesita tener un único Abonado y una lista de Contrataciones, así como estar decorada (encapsulada) por un método de pago. Por lo tanto, optamos por utilizar el patrón Factory para su

creación, ya que cada una está estrictamente ligada a un método de pago que la decora.

En adición a lo solicitado en el trabajo, creímos oportuno no limitar la creación de la factura a un solo tipo, en caso de que en un futuro quieran agregarse otras facturas extendidas de las mismas (por ejemplo: factura A,B,C,etc). Por lo tanto creamos un método protected dentro del factory el cuál crea la factura según el tipo de factura recibido como parámetro para luego ser decorada. Dicho método en la actual implementación, solo contempla que el parámetro recibido sea null (sería como definir a la factura en su estado predeterminado). La ventaja resultante es poder crear únicamente facturas decoradas por su tipo de pago y que además el tipo de facturas pueda ser expandido a través de una extensión del factory donde se redefina el método protected que crea la factura según su tipo.

Otro aspecto por fuera de las especificaciones del trabajo que creímos fundamental para nuestro sistema fue el de guardar aquellos abonados que no cuentan con una contratación. De esta forma la empresa tiene los medios de contactar con aquellos clientes potenciales. Para poder implementar esta idea decidimos que la creación de un abonado siempre inserta a este mismo en la lista de abonados sin contratación y luego cuando quiera crearse una factura se buscan y se retiran los abonados de esa misma lista. Este método de creación tiene su fundamento en evitar tener instancias en la capa de presentación. Por más de que en esta etapa se tenga un main, intentamos que se asimilará lo más posible a una interfaz en la que solamente se interactúa con sistema con parámetros que puedan ser ingresados por teclado.

## Dificultades encontradas y sus soluciones

Como todo desarrollo de programa, se han encontrado dificultades a medida que se realizaba su implementación.

Una de ellas fue la maquetación del programa. Un requerimiento fue trabajar respetando la arquitectura de 3 capas, lo cual precisó de una organización clave de cada aspecto del programa para poder respetar esto correctamente. Teníamos claro que en la capa de presentación sólo debería tener un método main que compruebe el correcto funcionamiento del sistema con todos los casos posibles. Al principio, no estábamos seguros de cómo diferenciar la capa de negocio de la de modelo, hasta

que finalmente optamos que la capa de modelo se encargue de tener una lista de facturas y una lista de abonados sin factura, y que sólo puedan retornar un iterator de la lista y agregar y eliminar elementos. Mientras que la capa de negocio, se encarga de tener todas las clases

Otra de las problemáticas con las que nos topamos fue la creación y el diseño de las facturas, ya que no sabíamos cómo distribuir las responsabilidades entre las clases facturas y personas y cómo implementar en ellas el decorator para no limitar la expansión de tipos de pago y tipos de factura. En un principio no sabíamos si tener un tipo de persona y dos tipos de factura (jurídica y física), pero luego determinamos que la clase abonado es quién debe extenderse en estas 2 clases. Al hacerlo, nos hallamos en el dilema de cómo hacer para que la factura calcule su valor dependiendo el tipo de persona que contenía en su atributo. Finalmente optamos por aplicar el patrón “double dispatch” que según el tipo de persona contenida en factura ejecutase el método correspondiente para calcular el valor de la factura. Este patrón trae aparejado una limitación a la hora de la expansión de clases (en nuestro caso se limita la expansión del lado de la persona), pero para nuestro sistema esto no tiene ninguna repercusión negativa, ya que comercialmente siempre existieron personas físicas y jurídicas, por lo cuál no creemos que sea necesario expandir estos en un futuro (a no ser que cambiase el paradigma comercial, lo que de igual manera implicaría rehacer todo el programa). Por lo tanto logramos aplicar este patrón permitiendo expandir los tipos de facturas, pero no los tipos de personas.

Asimismo, el manejo de excepciones fue un tema de discusión. Debimos ponernos de acuerdo sobre qué métodos podrían/deberían lanzar excepciones, cuales otros debían establecer precondiciones claras en el contrato y cuáles debían utilizar ambas. Cabe destacar que todos los métodos (salvo los getters y setters) y constructores, poseen un contrato con sus precondiciones (condiciones que deben cumplirse al invocar al método) y sus postcondiciones (condiciones que deben ser ciertas luego de la ejecución del método). De esta manera, se logra mejorar la calidad y la confiabilidad del software, facilitar el mantenimiento y mejorar la legibilidad del código para los desarrolladores. También se han utilizado asertos durante la etapa de desarrollo del programa para verificar que ciertas condiciones sean verdaderas en puntos específicos del programa. Lo anteriormente mencionado puede contemplarse en el Javadoc, una herramienta en la que se documentan

descripciones de clases, métodos, variables y otros elementos del código, así como información sobre sus parámetros y valores de retorno.

## Conclusión

Como cierre de esta primera instancia de entrega creemos haber desarrollado un sistema con una estructura remarcablemente sólida y fundamentalmente expandible. A su vez tenemos como objetivo para entregas futuras mejorar la capa de presentación y de modelo, además de aumentar su peso en el programa.