Informe Final "Contrataciones de Servicios"

Introducción al comportamiento del Sistema

En la ventana principal conviven la gestión de Abonados y de facturación. En primera se agregan los Abonados (a partir de nombre y DNI único y el tipo de persona), los domicilios con sus respectivas contrataciones (Internet100 o Internet500) y finalmente los servicios que tendrán (TV/Cable, teléfono y celular). Para que se pueda agregar a abonado alguna prestación, es necesario que se lo seleccione a uno en la primera la lista, ya que el sistema requiere saber a qué cliente se le debe agregar el domicilio. En el caso del servicio es lo mismo, pero con la dirección. Luego está la parte de facturación (accesible a través del botón ubicado en la parte superior de la pantalla con el mismo nombre). Aquí están los abonados agregados, y las facturas pendientes o pagadas que tenga. Las facturas se pueden pagar eligiendo el tipo de pago y las ya pagadas se pueden imprimir (mostrará por pantalla el detalle). Además están los botones "*Generar reporte de AFIP*", que mostrará en una segunda ventana las facturas de todos los abonados y que bloqueará temporalmente la posibilidad de agregar un nuevo Abonado, y "*Pasar mes*" que emitirá una factura a todos los abonados que cumplan con la condición de tener por lo menos un domicilio y actualizará los estados de los cliente a morosos, sin contratación o con contratación según correspondan.

Patrones usados

MVC: dividimos el proyecto en tres parte independientes. El modelo, dónde está el estado de la aplicación, la vista, donde se muestra la información y Controlador que maneja el flujo entre las dos capas anteriores.

Observer-Observable: usado para la comunicación entre Controlador y Modelo (clase Sistema). El sistema se extiende de Observable y en el constructor del Controlador, se inicializa y se le agregar el Observer. En el método "update", el controlador recibe todos los cambios de estado del Modelo.

State: usando para el manejo de estados en el abonado, más precisamente en el hijo "*PersonaFisica*". Este objeto tiene una referencia de tipo State, inicializada como "*SinContratacionState*" y a partir del pago, emisión de facturas o nuevas contrataciones irá cambiando a "*ConContratacionState*" cuando adquiera por lo menos uno o "MorosoState" si tiene más de dos facturas sin pagar.

Emulación de “Visita AFIP”

Consta de una clase llama “*VisitaAFIP”* que se extiende de “*Thread*” y que es inicializado en “realizarReporteAFIP()” (método de “*Sistema*”) y ejecutado a través el método “Start()”. “*VisitaAFIP*” recibe una interfaz “*IReporteAFIP*”, que implementa “*Sistema*” y que usará para notificar de la finalización del hilo y una referencia a “*GestionAbonados*” que es donde se encuentra el método de clonación de las facturas (que también recibe un “IResporteAFIP”, para ir notificando de los mensajes que se deben mostrar por pantalla), ejecutado en ”run()”.

En “*GestionAbonados*” hay dos métodos importantes que implementan un semáforo. Uno es “*agregarNuevoAbonado*(...)” y otro “*clonarFacturas*(...)”. La razón es que a pesar de que ambos son ejecutados por distintos hilos, no pueden correr ambos al mismo tiempo. Entonces la idea es que al entrar uno, el otro quede bloqueado a la espera de que el semáforo sea liberado. Sin embargo hay una diferencia en cuanto a la espera en sí. Si AFIP quiere hacer un reporte mientras se está agregando un nuevo abonado, éste quedará esperando en “*semaforo.acquire()*”. Pero en el caso del abonado, esto no puede ocurrir debido a que la ventana quedaría congelada. Así que se hace uso de *“semaforo.tryAcquire()*”, que en caso de que no esté disponible, devolverá un false. Aprovechando esto, el método simplemente termina.

A medida que el reporte se vaya realizando (en pararlo al funcionamiento del programa principal), se usará el “*IReporteAFIP*” para que la clase “*Sistema*” vaya mandando mensajes con el nombre del abonado y la cantidad de facturas que se clonaron, ya que es esta la que tiene la comunicación con “Controlado” y a su vez con “Vista”.

Se usó un “sleep”, para simular un tiempo de trabajo ya que la ejecución normal se hace en un tiempo muy reducido.

Paquete “Modelo”

En el paquete modelo se encuentra la clase “*Sistema*”, que se extiende de **observable**, y que es la que responde a las consultas del “*Controlador*” (observer). Dentro se encuentran dos atributos, “*GestionAbonados*” que usa para ceder la responsabilidad de gestionar la colección de clientes y “*IOAbonado*” donde se le asigna la tarea de persistir el estado de las clases. Además implementa “*IEnviarMensaje*”, donde se hace uso del objeto “EstadoMensaje” para notificar de los cambios de estado al Observer y “*IReporteAFIP*” donde están los métodos que usará el “VisitaAFIP” (hilo que genera los reportes) para enviar los mensajes que finalmente se mostrarán por pantalla.

Clase “*GestionAbonados*” : Contiene, como se dijo anteriormente, la colección de abonados, un atributo de tipo “*Semaphore*” que asegura que agregar Abonados y realizar reporte no se ejecuten al mismo tiempo y métodos como: “*agregarNuevoAbonado*”, donde además se verifica que no haya otro con un mismo DNI, “*agregarContratacion*”, comprueba la repetición pero con la

dirección, "*agregarNuevoServicio*", "*avanzarMes*", que es el encargado de emitir las facturas a todos los usuarios, y "*clonarFacturas*", donde hace el reporte.

Clase "Abonado" : Clase raíz del modelo y que contiene las "contrataciones" y "facturas" que se encuentran relacionadas mediante los mismos atributos del abonado, tiene la funcionalidad de poder generar una clonación de todas sus facturas, y comportamientos distintos en las clases que heredan de Abonado, éstas son:

- Persona Física
- Persona Jurídica

Clase Persona Física: Extiende de Abonado, implementa el patrón state, donde variarán sus acciones en base a 3 comportamientos que enumeramos aquí:

1. MorosoState: Cuando existen 2 facturas o más no pagas, se bloquean los acciones de dar de baja una contratación y los pagos de facturas atrasadas son con recargo.
2. SinContratacionState: No posee contrataciones por lo tanto no podrá pagar facturas ni realizar bajas de contratación.
3. ConContratacionState: Puede contratar nuevos servicios, darse de baja y pagar facturas

Clase Persona Jurídica: Extiende de Abonado, su comportamiento es único en todo el ámbito, cuando se genera una factura de una persona jurídica, aquél tendrá descuentos en el pago de la factura.

Clase IEnviarMensaje: Interfaz que implementa sistema y gestión de abonados dónde se desarrollaron comunicaciones entre la vista y el modelo, a través de la controladora

Clase Sistema: Esta clase es la contenedora del resto de las instancias de otros objetos y posee los métodos de persistencia y despersistencia de los datos. Implementa IEnviarMensaje para controlar el flujo de acciones provenientes de controlador en la aplicación.

Desarrollo: Se comenzó por el diseño del patrón State y vinculación con la clase "Persona Física", se modificaron los atributos de abonado para que contenga facturas e implementación de los métodos pertenecientes a la colección de factura "I_Coleccion_Factura", agregando los requisitos de facturas impagas y pagas. Se cambió el comportamiento de "Medio de Pago" para que lo adopte abonado, ya que antes era adoptado inmediatamente sobre la factura. En contratación, se ajustó el Hashcode para usarlo como ID único de cada contratación. Luego, se agregó la clase nueva "VisitaAFIP" que debía ser synchronized con el método de "dar de alta Usuario", es decir, no podía generar la clonación de las facturas si mientras se daba de alta un usuario.

Paquete "io"

Se desarrollaron las clases de persistencia "IOAbonado", que realiza la persistencia de los abonados, y "IOAFIP", que realiza la persistencia de las facturas emitidas a todos los abonados. Estas clases implementan los comportamientos de las interfases IWriter y IReader. IWriter representa métodos de escritura de datos en archivos binarios. IReader representa métodos de lectura de datos en archivos binarios.

Paquete "Controlador"

La clase "*Controlador*" se divide en dos partes, "*actionPerformed(ActionEvent arg0)*" donde recibe todos los eventos de "*IVista*" y "*update(Observable arg0, Object arg1)*" donde recibirá y enviará, a través del **Patrón Observer-Observable**, todos los mensajes del modelo para mantener la vista actualizada. Este patrón nos permite notificar a "*Ventana*" cuando se produzcan cambios concretos en el estado del modelo. La comunicación se da por un atributo de clase "*IVista*" (la vista) y "*Sistema*" (el modelo). Ambas instancias se dan en el constructor, donde además "*Sistema*" agrega a "*Controlador*" como observer, respetando la responsabilidad de que sea el observador quien mantenga la vinculación.

Paquete "Vista"

Está compuesto por 2 clases concretas que se extiende de JFrame y 2 interfaces. La clase "*Ventana*", que implementa "*IVista*", recibe la notificación de cambios y envía a "*Controlador*" los gestos del usuario, haciendo uso de "ActionListener" (definido en "Controlador" y recibido a través del método "setActionListener(...)"). Todos los botones trabajan con este objeto para comunicarse con "*Controlador*", salvo "*btnAbonados_Panel*" y "*btnFacturacion_Panel*" que son de uso interno y que manejan los paneles, permitiendo que la gestión de abonados y de facturas estén en una misma ventana. Los "JList" deben pasar por "valueChanged(ListSelectionEvent arg0)" para ser convertidos a "ActionEvent" y que "ActionListener" pueda recibirlos correctamente. El resto de los métodos que implementa de "*IVista*", tienen como tarea mostrar la lista de Abonados, Contrataciones y/o Servicios, a partir de Iterators que reciben, indicar a "Controlador" que se ha escrito en "JTextField" o los "*JRadioButton*" que se han marcado. También tiene una instancia de tipo "*IVentanaAFIP*", que será la ventana encargada de mostrar los reportes de la AFIP. La idea es que reciba de "Controlador" la orden de abrir la ventana con el método "*generarReporteAFIP()*" (verificando que no esté ya creada), mostrar los mensajes en pantalla con "*actualizarReporteAFIP(String mensaje)*", terminar de mostrar el reporte "*terminarReporteAFIP(String mensaje)*" y finalmente cerrar dicha ventana con "*cerrarReporteAFIP()*" para además borrar la referencia del objeto y se pueda volver a abrir. Finalmente se encuentrar los metodos "*MostrarErrorEmergente(String mensaje)*" y "*imprimirFacturas(String mensaje)*" que reciben String y los muestran como ventanas emergentes. Algo importante que se tuvo que implementar fue el atributo de tipo booleano "*listaUsada*", que evita que en los métodos donde se actualizan las listas, no se llame continuamente a "valueChanged(...)". El problema es que cuando se borran todos los elementos de los "DefaultListModel" se envia a "Controlador" mensajes vacíos o erróneos haciendo que ventanas

emergentes se muestran entre otros problemas. Entonces mientras se hace la limpieza de la lista, el valor de este atributo pasa a ser true.

Luego está la ”*VentanaAFIP*”, que implementa “*IVentanaAFIP*”. Tiene la tarea de recibir los mensajes de la clonación de Facturas y mostrarlos por pantalla en un “*JTextPane*” y mandar a “*Controlador*” el mensaje de que se ha presionado el botón Salir. Estos mensajes pasan primero por "*Ventana*" debido a que se quiere respetar la independencia del **Patrón MVC**. De esta manera si el diseño de la Vista consiste en una sola ventana o en dos (como este caso), el "*Controlador*" no se vea afectado