Universidad Nacional de Mar del Plata 

Trabajo Práctico

Asignatura: Programación 3

Año: 2023

Lugar: Facultad de Ingeniería

Integrantes:

Basualdo, Juan Ignacio

Juares, Alan

Trinitario, Bruno

Olave, Juan Ignacio

Informe

Introducción:

En este trabajo se pretende desarrollar un sistema de facturación empresarial para trabajar con contrataciones de servicio de monitoreo y seguridad, con clientes.

Utilizando los patrones de diseño vistos en clase, tales como Singleton, Decorator, Observer/Observable, State etc.

También se nos propuso el reto de crear una interfaz gráfica la cual “visibiliza” las capas de abstracción dentro de nuestros software.

Soluciones elegidas:

La clase empresa es la encargada de agregar sus respectivos abonados al sistema o quitarlos en su defecto, crea/remueve sus facturas. También crea sus contrataciones y trabaja con los domicilios ya sea designando a su respectivo abonado o quitándole al mismo.

Se crean domicilios, abonados, servicios, promos y contrataciones. A la contratación se le asigna un domicilio e implementa la lista de servicios con su respectiva promoción, en caso de tenerlas.

A cada abonado se le asignan domicilios y sus correspondientes contrataciones.

Encapsulamos el abonado según su tipo de pago, inicialmente su tipo de pago será siempre “SinPago” y luego a partir de la instancia obtenida en la clase factory (en el momento que

pague el abonado) se lo envía a factura, la cual cuenta con las siguientes funcionalidades: imprimir un factura, clonar una factura.

Las excepciones con las que tratamos son:

- ContratacionInvalidaException: esta excepción se realiza cuando no se puede crear la contracción debido a la asignación de un domicilio ya vinculado.
- DomicilioExistenteException: es lanzada cuando el domicilio ya se encuentra agregado a la lista de domicilios del abonado.
- DomicilioInexistenteException: es creada en la situación en la que se solicita eliminar un domicilio de la lista de domicilios del abonado pero este no existe.
- AbonadoInexistenteException: es lanzada cuando se quiere eliminar un determinado abonado de la lista que contiene abonados y este no existe en ella.
- FacturaInexistenteException: es creada cuando se quiere eliminar una factura de la lista de facturas y no está en la lista.
- FactoryInvalidoException: esta excepción se lanzará cuando no existe el tipo String de pago para realizar la instancia del objeto.
- ReparacionYaSolicitadaException: es creada cuando un usuario solicite servicio técnico pero no hay ninguno en la cola de espera de técnicos.

Las primeras 8 excepciones planteadas las desarrollamos debido a que existe la posibilidad de que el domicilio que ingrese sea incorrecto o no exista. Y al encontrarse en una lista, la cual puede contar con varios domicilios no es meramente responsabilidad de quien invoque el programa tener conocimiento de cada uno de sus domicilios (lo mismo aplica para abonado y factura).

La anteúltima excepción se realiza ya que puede mandarse un tipo String de pago inválido y al existir la posibilidad de que la clase Factory se extienda debido a la implementación de nuevos métodos de pagos aumenta la probabilidad de error.

Para modelar los diferentes comportamientos que puede tener una persona física creamos las clases Moroso, ConContratación y SinContratación. En ellas si el abonado en su lista de facturas tiene por lo menos una factura entonces se encuentra en el estado "ConContratación" o dependiendo de la cantidad de facturas que posee impagas cambia su estado a Moroso o no.

Patrón MVC:

Propuso una dificultad extra a la hora de diseñarlo e implementarlo ya que si bien no teníamos las prácticas bien entendidas, nos surgieron dudas tales como ¿dónde se coloca esto?, cómo se comunica la vista?, ¿Qué debe hacer el controlador respecto a la vista y el modelo?. Pero más allá de todas estas dudas y dificultades pudimos llegar a soluciones satisfactorias. Logramos comprender el patrón y su uso.

Esto se denota mucho en el método:
"public void actionPerformed(ActionEvent e)"
Este método recibe de la vista la acción de un evento producido por la interacción que otorga la vista y desde la misma dependiendo el tipo de evento se puede solicitar al modelo que actualice,consulte,inserte,cambie o elimine datos. La ventaja que nos dio este patrón es la de trabajar sin la dependencia de saber la implementación de parte del código entre las distintas partes del modelo permitiéndonos trabajar en paralelo.

Persistimos la clase Empresa en cascada para resguardar los datos en el sistema.

Simulamos el flujo de técnicos, permitiendo agregar los mismos y al solicitar la reparación,uno de los técnicos toma ese trabajo,poniendo en espera a los demás.Puede haber más un técnico reparando en simultáneo.

El paquete de servicios contiene los distintos tipos de servicios (botón, cámara o acompañamiento) y sus precios, los cuales se sumarán al precio base dependiendo de si el abonado tenga un domicilio de tipo vivienda o comercio y si tiene aplicada la promoción platino o dorada.

Las dificultades que se nos presentaron son:

- Le dábamos responsabilidades al domicilio de conocer su abonado,con la sumatoria del costo total del servicio con sus agregados y promos. Solventamos este problema delegando la responsabilidad a la clase Contratación de informar estos costos previamente mencionados.
- Mala implementación del doble dispatch generado por el problema anteriormente mencionado. Aplicamos double dispatch en la clase ‘domicilio’.
- No tuvimos en cuenta que el software desarrollado debe poseer cierto grado de escalabilidad
- Poder adecuar los patrones de diseño al código sin modificar el funcionamiento de este.

Cambios en la segunda parte:

Decidimos re-implementar el método decorator, para facilitar el uso del patrón MVC, nos aseguramos de que esta nueva implementación cumpla con los requerimientos previos de la parte anterior.
Decidimos eliminar los métodos que “simulaban” la vista, ya que por razones claras pasaron a ser parte de esta misma