Documentación desafío “Sistema de facturación” Mercap

INFORMACION:

Realizado por Nicolás Garilli.

Lenguaje Utilizado: Java.

– Utilice openjdk-19 para realizar el desafío.

INTRODUCCIÓN:

El motivo de este documento es transmitir cómo fui modelando las entidades para llevar a cabo el enunciado, y explicar porque tome las decisiones que tome a la hora de modelar el mismo.

A la hora de empezar, comencé modelando entidades simples que no tengan muchas dependencias de otras entidades como lo es Destination. Me hubiese gustado realizarlo en base a TDD, pero no me sentía capaz de realizar el enunciado en tiempo y forma mediante este método, así que utilice TDD al inicio y luego fui realizando tests unitarios para comprobar que todo funcionara correctamente.

Destination:

Destination es el destino al cual el cliente realiza la llamada. Como el costo de las llamadas nacionales e internacionales varían según el precio del destino al que se realice la llamada, me pareció coherente que cada Destino sepa su costo por minuto de llamada.

CallDetails:

CallDetails lo utilizo para darle todos los elementos que una llamada va a necesitar para calcular su costo, tales como la duración de minutos, el destino, y la hora en la que se realizó la llamada.

CallDetails tambien resulta util a la hora de mostrar una factura de un Cliente, dando la

posibilidad de mostrar toda la información de los llamados que este realizó, junto con el monto de su factura mensual.

ICallStrategy:

Para resolver la lógica entre las llamadas Locales y las Nacionales/Internacionales, se me ocurrió resolverlo mediante el uso de un Strategy. De esta manera, el Cliente solo utiliza un único método “realizarLlamado(CallDetails, CallType)” El cual recibe por parámetro que tipo de llamado va a realizar, una Local, o Nacional/Internacional, Y los callDetails necesarios para poder realizar esa llamada.
De esta Manera, el cliente no requiere de dos métodos distintos para cada tipo de llamada que quiera realizar.

LocalCallStrategy/NationalOrInternationalStrategy:

A la hora de leer el enunciado, pude distinguir entre dos posibles escenarios, uno en el cual el Cliente quiere realizar una llamada Local, la cual tiene su propia lógica para ser calculada, y el caso del que él mismo quiera realizar una llamada Nacional o Internacional.
Como las llamadas Nacionales e Internacionales, ambas necesitan del precio al destino al que están llamando para calcular su costo, no vi la necesidad de realizar dos clases distintas para cada una, ya que ambas lo calculan de la misma manera.

LocalCallStrategy:

Para las llamadas locales, que tienen un costo distinto dependiendo el dia y el horario en el que se realice la llamada, no me gusto la idea de que sea el trabajo de las mismas llamadas saber el día en el que se realizó la llamada para calcularse, por lo cual la lógica del dia y la hora la delegue a un Utility llamado CallHelper.

NationalOrInternationalCallStrategy:

Según mi comprensión del enunciado, las llamadas nacionales e internacionales no tienen en cuenta el día y la hora en el cual se realice la llamada para calcular su costo,

sino que lo calcula en base a la duración de la llamada y el costo del destino al que están llamando.

CallHelper:

Clase estática que ayuda a delegar las tareas como ver en que dia y horario se realizó una llamada para ver su costo.
Originalmente, esta clase la iba a llamar LocalCallHelper, ya que la iba a utilizar únicamente en las llamadas locales para calcular su costo, ya que las nacionales/internacionales pueden calcularlo sin la necesidad de esta.
Sin embargo, a la hora de que cada CallStrategy calcule su costo, me di cuenta que ambos realizan la cuenta: (duracionDeLaLlamada * costoPorMinuto), únicamente que la local calcula su costo por minuto de una forma, y la nacional e internacional de otra. Para no repetir la misma cuenta en ambas clases, a pesar de que es únicamente una multiplicación, me pareció adecuado delegarlo al Utility y así no repetir ese código.
Por otro lado, quizá me hubiera gustado la posibilidad de delegar esa cuenta en una superclase, y que ambas llamadas la hereden, pero dado a que ambas implementan una interfaz, en la cual sus métodos son abstractos por defecto, me pareció mejor opción delegarlo al Utility y que ambas clases lo llamen cuando lo necesiten.

Client:

El cliente fue un gran dilema a la hora de modelar cómo realiza llamados.
Inicialmente se me ocurrió que tenga un método por cada tipo de llamada que quiera realizar. Si este fuese a realizar una llamada local, este usaria el metodo llamadaLocal(params) el cual recibirá por parámetro los datos necesarios para realizar una llamada local tales como el dia y horario de la llamada. Pero debería realizar lo mismo para cada tipo de llamada que vaya a poder realizar.
Para realizarlo de una manera más polimórfica, se me ocurrio que el mismo tenga un unico metodo realizarLlamado(params) el cual recibe por parámetro el tipo de llamada (CallType) que va a realizar y los detalles de la misma (CallDetails).
CallType es un ENUM, el cual puede ser LOCAL o NACIONALOINTERNACIONAL.
Dependiendo del CallType, se setea el tipo de Strategy que va a utilizar.

InvoiceSystem:

La utilizo como mi main del sistema de facturación, es el encargado de imprimir en pantalla, por cada cliente que esté en el sistema, su costo base, sus costos adicionales por cada llamada que realizó, los detalles de esas llamadas realizadas (Tales como su duración y destino), el monto total que debe cada cliente, y también el monto total de facturación de todos los clientes.

A tener en cuenta:

Me llamo la atención que el enunciado informaba que la facturación sea realizada mensualmente, pero no consideré necesario implementar dicha lógica en mi implementación, ya que considero que correr este sistema mensualmente, sería tarea de un Scheduler.