Objetivo general del proyecto

El objetivo general de este proyecto es analizar críticamente el diseño planteado e implementado en las entregas anteriores, desde la óptica de nuevos requerimientos que tengan que agregarse a una aplicación.

Objetivos específicos del proyecto

Durante el desarrollo de este proyecto se buscará el desarrollo de las siguientes habilidades:

1. Analizar críticamente un diseño.
2. Introducir nuevas funcionalidades en un diseño existente.
3. Diseñar bajo restricciones.
4. Diseñar e implementar un esquema de manejo de errores basado en excepciones
5. Implementar pruebas automáticas usando JUnit

Instrucciones generales

El proyecto debe desarrollarse en los mismos grupos del proyecto #2 y debe partir de lo que entregaron para el proyecto #2.

Tienen derecho a reevaluar todas las decisiones de diseño que tomaron en las entregas anteriores siempre y cuando las documenten.

Cambios en el proyecto

Para este proyecto, su aplicación tendrá que ser capaz de soportar los requerimientos adicionales y restricciones que se describirán a continuación. Debe seguir teniendo una interfaz gráfica para soportar todas las funcionalidades y tiene que seguir soportando todos los requerimientos que estaba soportando en las iteraciones anteriores del proyecto.

Emisión de facturas en pdf

De ahora en adelante al momento de la entrega de un vehículo y del pago se debe generar una factura en pdf. Tienen libertad sobre la organización y el contenido de la factura, pero debe ser necesariamente un archivo pdf que incluya la firma del administrador como una imagen.

Use alguna de las múltiples librerías existentes para trabajar con archivos pdf.

Características de los vehículos

De ahora en adelante la descripción de los vehículos debe incluir obligatoriamente las siguientes características, además de todas las que ustedes ya hubieran definido:

- Tipo de vehículo (automóvil, moto, atv, bicicleta, bicicleta eléctrica, patineta eléctrica, …)
- Porcentaje adicional de la prima del seguro (algunos vehículos son más riesgosos y requieren que se pague un valor adicional sobre los seguros).

Es importante que no se pidan características que no tengan sentido para ciertos vehículos (ej. el cilindraje de una patineta eléctrica).

Pagos con Tarjeta de Crédito

Buscaremos ahora que su aplicación esté cada vez más lista para recibir pagos con tarjeta de crédito: la aplicación debe incluir un mecanismo flexible para integrarse con pasarelas de pago (PayU, Paypal, etc.). Todas estas pasarelas ofrecen un servicio que recibe la información de una tarjeta de crédito (incluyendo la información del dueño) y la información de un pago que se debe hacer (monto, número de cuenta y número de transacción), y retornan un resultado que indica si el pago se realizó de forma exitosa o si hubo algún problema particular (ej. los datos de la tarjeta están equivocados, la tarjeta no tiene cupo suficiente, la tarjeta está reportada, etc.). En la realidad cada pasarela tiene pequeñas diferencias con este modelo, pero es imaginable un servicio abstracto de pago detrás del cuál pueda estar cualquier pasarela.

Su aplicación debe permitir ahora que se pueda hace el pago (simulado) con una tarjeta de crédito: en el momento del pago, el empleado de la sede seleccionará una de las pasarelas de pago disponibles, digitará la información de la tarjeta y de su dueño, y procederá a realizar el cobro del alquiler del vehículo. También debe poderse hacer el bloqueo del cupo en la tarjeta de crédito.

Su aplicación debe poder soportar con facilidad que se cambie la implementación de las pasarelas: debe haber un archivo de texto plano con la configuración de las pasarelas disponibles. Cada línea en ese archivo será el nombre de una clase Java que implemente una pasarela simulada. Las pasarelas simuladas deben simplemente crear un archivo de texto con la traza de las transacciones que se hayan realizado y el resultado de cada una.

Ayuda: al final de este documento se encuentra una ayuda sobre el método `forName()` de la clase `Class` que le será de mucha ayuda en este punto.

Por ejemplo, el archivo de configuración podría decir que hay 3 implementaciones de pasarelas llamadas `uniandes.dpoo.PayPal`, `uniandes.dpoo.Payu` y `uniandes.dpoo.Sire`. Entonces, cuando se vaya a hacer checkout, las opciones de pasarela que deben aparecerle al usuario serán PayPal, Payu y Sire. A medida que se vayan haciendo pagos, se irán registrando en tres archivos (PayPal.log, Payu.txt, y Sire.json). En este ejemplo los tres archivos son diferentes para ilustrar que cada implementación puede definir independientemente cómo hace el registro.

El valor de su diseño dependerá de qué tan fácil sea soportar una nueva pasarela (ej. Apple Pay).

Aplicación para clientes

Usted debe crear un nuevo programa con una interfaz gráfica que se ejecute independientemente del resto del sistema. Este nuevo programa será usado sólo por los clientes y sólo debe tener las siguientes funcionalidades:

- Registrar un usuario para un nuevo cliente con login y password.
- Consultar cuál es la disponibilidad de vehículos para una cierta sede en unas fechas determinadas.

- Reservar un vehículo. Si un cliente paga un alquiler a través de esta nueva aplicación apenas haga la reserva, tendrá derecho a un 10% de descuento. Después de haber recogido un vehículo, es posible que en el momento de la entrega el cliente tenga que pagar un excedente: esto debe hacerse a través de la aplicación de los empleados que han venido construyendo (no de la de clientes).

Esta nueva aplicación para clientes debe funcionar sobre los mismos datos que la otra aplicación. Para simplificar el problema, puede suponer que no se van a ejecutar las dos aplicaciones a la vez.

Cuando se abra la aplicación de clientes debe mostrar sólo dos opciones: crear un nuevo usuario, o autenticar un usuario. No debe haber ningún mecanismo para que el usuario indique qué archivos se deben cargar.

Sólo debe desarrollarse un proyecto (en Eclipse) que contenga los dos programas. No debe haber código repetido.

Pruebas automáticas

La implementación de su proyecto debe incluir pruebas automáticas para dos aspectos: pruebas de integración sobre las funcionalidades relacionadas con la creación de reservas y pruebas unitarias sobre las funcionalidades para la carga de archivos.

Las pruebas deben cubrir tanto los caminos correctos como los posibles caminos de excepción.

Las pruebas no deben incluir la interfaz de la aplicación.

Para las pruebas sobre la carga de archivos, intente maximizar la cobertura: intente que todas las líneas de código relacionadas con la carga de archivos queden probadas e intente evitar que se pruebe más de un componente (o clase) a la vez.

La implementación de las pruebas debe hacer parte del mismo proyecto en Eclipse, pero debe estar en una carpeta de fuentes diferente a las otras aplicaciones.

Entrega: Implementación y análisis del proceso

Esta entrega incluye tres elementos fundamentales:

1. La implementación completa del sistema. Esta implementación debe incluir archivos con datos para poder probar manualmente todas las funcionalidades de la aplicación.

2. El documento de diseño para el proyecto 3, teniendo en cuenta todas las recomendaciones dadas en clase para estos documentos e incluyendo el diseño de los posibles errores.

3. Un documento donde cada grupo analice el proceso completo que siguió durante los tres proyectos (Proyecto 1, Proyecto 2 y Proyecto 3). Con este documento se busca que ustedes reflexionen sobre el proceso de diseño en el contexto de su propia aplicación: ¿Qué cosas salieron bien y qué cosas salieron mal? ¿Qué decisiones resultaron acertadas y qué decisiones fueron problemáticas? ¿Qué tipo de problemas tuvieron durante el desarrollo de los proyectos y a qué se debieron? En este último punto, sería conveniente discutir aspectos como los problemas con las estimaciones del trabajo necesario debido al desconocimiento de la tecnología, los problemas realizando el análisis del dominio, o la dificultad de diseñar en un entorno incierto. La estructura de este documento es libre, pero será evaluada con respecto a su calidad general (está bien escrita, las ideas son claras, ilustra correctamente las situaciones con ejemplos puntuales), a la visión crítica que presenten

(qué tan críticos son con respecto a su propio trabajo, identifican las partes problemáticas del proceso, pero también identifican los aspectos positivos) y a qué tan útil sería este documento para alguien que fuera a realizar los mismos 3 proyectos.

Entrega

1. Dentro de la carpeta del proyecto debe crear una carpeta con el nombre "**Entrega Proyecto**" donde deben quedar todos los elementos correspondientes a esta entrega, incluyendo el documento de diseño, el documento de reflexión y el proyecto Eclipse con instrucciones para su ejecución. Incluya también archivos de prueba para poder correr las aplicaciones y tener datos con los que se pueda probar con facilidad. Incluya también los archivos fuente de los diagramas de sus documentos, así como imágenes que se puedan leer con facilidad.

2. Entregue un enlace al repositorio a través de Bloque Neón en la actividad designada como "**Proyecto 3 - Entrega 2**".

Ayuda: carga dinámica de clases

Hasta el momento, para crear una nueva instancia de una clase, usted ha escrito líneas como la siguiente:

```
Clase1 obj = new Clase2(parámetros);
```

Escribir esta línea implica lo siguiente:

- `Clase1` debe ser una superclase de `Clase2`, o deben ser la misma clase.
- Cuando usted escribió el código, usted sabía que el objeto `obj` iba a ser una instancia de la clase `Clase2`.

Hay algunos casos en los cuales el segundo punto no aplica porque no estamos seguros de cuál va a ser la clase exacta del objeto `obj`: sabemos que debería ser una subclase de `Clase1`, pero no sabemos exactamente cuál. En estos casos, necesitamos un mecanismo flexible que nos permita definir el nombre de la clase lo más tarde posible.

El siguiente fragmento de código resuelve el problema, usando tres métodos que posiblemente usted no ha usado hasta ahora.

```
Class clase = Class.forName(nombreClase);
SuperClase obj = (SuperClase) clase.getDeclaredConstructor(null).newInstance(null);
```

A continuación, describimos estos métodos:

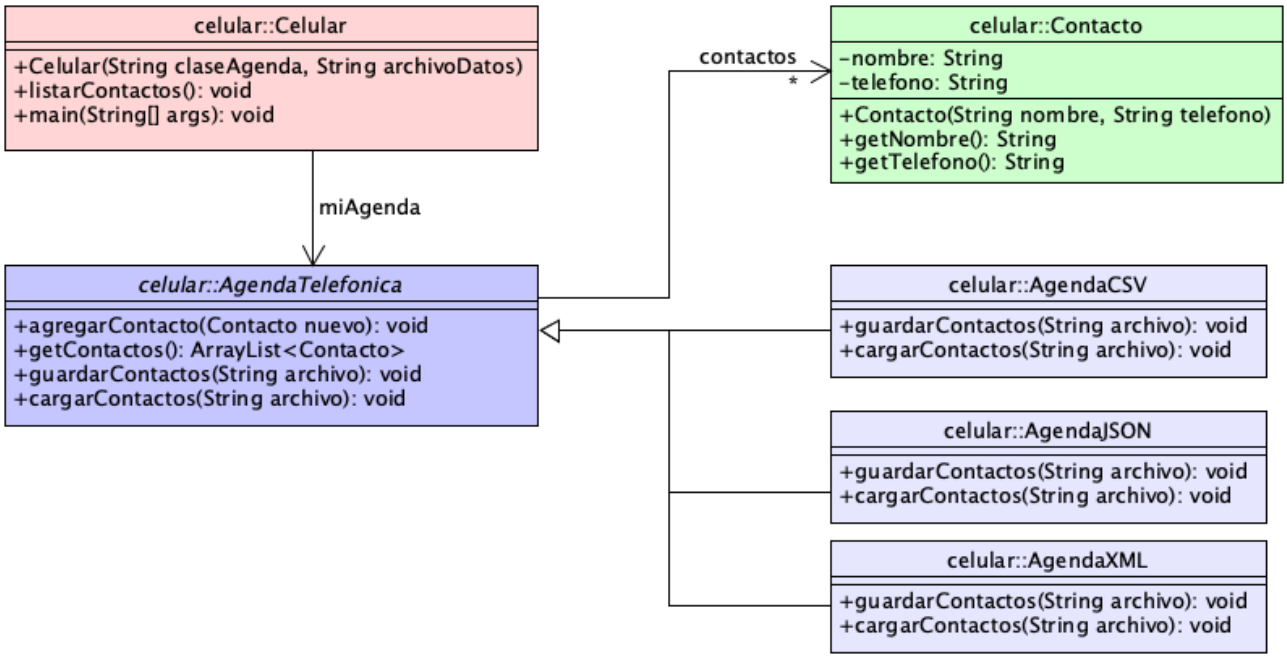
- El método estático `forName` de la clase `Class`, que nos permite cargar una clase a partir de su nombre (cadena de caracteres). Este método retorna un objeto de la clase `Class` que tiene toda la información de la clase de la que nos dieron el nombre.
- El método `getDeclaredConstructor(Object[] tipos)` de la clase `Class`, que nos permite obtener uno de los constructores definidos en la clase. El parámetro de este método indica los tipos de los parámetros del constructor que estamos buscando; si usamos `null`, significa que buscamos un constructor sin parámetros. Este método retorna algo de tipo `Constructor`.
- El método `newInstance(Object[] valores)` de la clase `Constructor`, que recibe una cantidad de valores e invoca el constructor con esos parámetros para construir un nuevo objeto; si se usa `null` como

parámetro, significa que no se están enviando parámetros porque el constructor no esperaba ningún parámetro. El método retorna algo de tipo `Object` al cual se le debe hacer casting para que se útil.

Veamos ahora el uso de este mecanismo en un caso concreto. Queremos tener un celular que tiene la posibilidad de guardar los datos de la agenda telefónica en diferentes tipos de archivos. Por ahora sabemos que los archivos podrían ser CSV, JSON o XML, pero en el futuro podría haber otros tipos. Además, no queremos que nuestro código defina explícitamente el tipo de archivo, por lo cual decidimos cargar dinámicamente la clase específica que implementará la agenda y se encargará de escribir y leer los datos de los contactos.

El siguiente diagrama de clases muestra las clases que hacen parte de la solución:

- La clase abstracta `AgendaTelefonica`, que tiene una colección de contactos. Los métodos `guardarContactos` y `cargarContactos` de esta clase son abstractos.
- Las clases `AgendaCSV`, `AgendaJSON` y `AgendaXML` que extienden `AgendaTelefonica` y le dan una implementación a los métodos faltantes.
- La clase `Celular`, que tiene una agenda: el celular no sabe la clase exacta de ese objeto; sólo sabe que pertenece a una subclase de `AgendaTelefonica`.



Las clases `Contacto`, `AgendaTelefonica`, `AgendaCSV`, `AgendaJSON` y `AgendaXML` no tienen nada que usted no haya visto ya y sea capaz de reconstruir. La clase `Celular` es la que tiene cosas particulares relacionadas con la carga dinámica de clases: estudie con cuidado el siguiente código y preste especial atención a estas dos cosas:

- En el método `main`, se le pregunta al usuario por el nombre de la clase que se usará para la agenda. El usuario debe digitar el nombre completo de la clase, incluyendo el nombre de los paquetes.
- En el constructor de la clase `Celular`, se carga la clase cuyo nombre llega como un `String`, se busca un constructor sin parámetros y se crea un objeto que sea una instancia de esa clase.

`Celular.java`

```java
public class Celular
{
   // La agenda telefónica de este celular: no sabemos a qué clase pertenece
   // exactamente
   // el objeto; sólo sabemos que pertenece a una subclase de AgendaTelefónica
   private AgendaTelefonica miAgenda;

   /**
    * Construye un nuevo celular e inicializa su agenda telefónica
    *
    * @param claseAgenda  El nombre completo de la clase que se usará para
    *                     construir la agenda
    * @param archivoDatos El nombre del archivo que continene la información de la
    *                     agenda
    */
   public Celular(String claseAgenda, String archivoDatos)
   {
        try
        {
            // 1. Dado el nombre completo (claseAgenda), encontramos un objeto de la clase
            // Class
            Class clase = Class.forName(claseAgenda);

            // 2. Le pedimos a la clase un constructor sin parámetros y luego lo usamos
            // para crear una nueva instancia de la clase
            miAgenda = (AgendaTelefonica) clase.getDeclaredConstructor(null).newInstance(null);

            // 3. Cargamos los contactos llamando al método abstracto de AgendaTelefonica:
            // la implementación que se ejecutará dependerá de la clase exacta que se haya
            // recibido como el parámetro claseAgenda
            miAgenda.cargarContactos(archivoDatos);
        }
        catch (IOException e)
        {
            System.out.println("Hubo un error de lectura");
        }
        catch (ClassNotFoundException e)
        {
            System.out.println("No existe la clase " + claseAgenda);
        }
        catch (Exception e)
        {
            System.out.println("Hubo otro error construyendo la agenda telefónica: " + e.getMessage());
            e.printStackTrace();
        }
   }

   /**
    * Le pide los contactos a la agenda e imprime el nombre de cada uno en la
    * consola
    */
   public void listarContactos()
   {
        miAgenda.getContactos().forEach(contacto ->
        {
            System.out.println(contacto.getNombre());
        });
   }

   public static void main(String[] args) throws IOException
   {
        System.out.println("Indique el nombre de la clase para la agenda telefónica del celular.");
        System.out.println("Si no teclea nada, será 'celular.AgendaCSV'");

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        String nombreClase = reader.readLine();
        if (nombreClase.length() == 0) // El usuario no tecleó nada
            nombreClase = "celular.AgendaCSV";

        // Crea un nuevo celular indicando que la agenda debe ser una instancia
        // de la clase que haya tecleado el usuario
```

```
        Celular miCelular = new Celular(nombreClase, "./data/datos.csv");
        // Celular miCelular = new Celular("celular.AgendaCSV", "./data/datos.csv");

        miCelular.listarContactos();
    }

}
```