Objetivo general del proyecto

El objetivo general de este proyecto es practicar varias etapas del desarrollo de una aplicación de software, desde el análisis hasta la construcción una aplicación funcional. A través del proyecto, los estudiantes pondrán en práctica todas las habilidades desarrolladas en el curso.

Objetivos específicos del proyecto

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

1. Diseño y documentación de interfaces gráficas basadas en el framework Swing.
2. Implementar interfaces gráficas
3. Diseño de sistemas basado en un framework existente.
4. Evaluar y mejorar el diseño del proyecto #2 con base en requisitos de evolución.

Instrucciones generales

Para este proyecto usted puede (debe) empezar a trabajar sobre su entrega para el proyecto #2, pueden hacerse cambios sobre el diseño original para acomodar mejor los requerimientos de este proyecto.

El proyecto DEBE desarrollarse en los mismos grupos del proyecto #2.

Entrega 1: Diseño e implementación de la interfaz gráfica

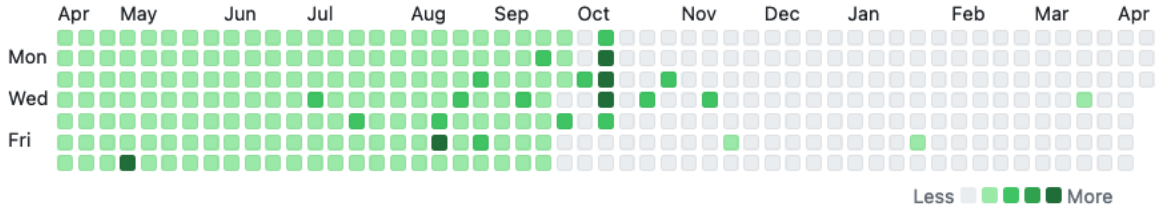
Modifique su programa para que la interfaz de usuario ya no sea una interfaz basada en consola y sea en cambio una interfaz gráfica. La interfaz idealmente debe estar basada en Swing, pero JavaFX también es una opción aceptable.

Primero, usted debe hacer un diseño preliminar de cómo se verá la interfaz gráfica de la aplicación. Este diseño puede ser dibujos digitales, fotografías de dibujos hechos a mano, *mockups* construidos con una herramienta especializada o no especializada, bocetos construidos como un programa Java sin funcionalidades, o cualquier otra técnica que usted considere conveniente.

Lo importante es que estos artefactos le permitan saber cómo será la interfaz que van a construir, sean útiles para guiar el diseño de la implementación y sirvan para planear su trabajo.

Restricciones adicionales

Debe haber una visualización de alto nivel que muestre gráficamente la cantidad de ventas realizadas a lo largo de un año. Un ejemplo basado en la matriz de actividades de GitHub se puede ver en la siguiente imagen (donde las casillas oscuras indican el valor total de las ventas en ese día), pero usted puede seleccionar cualquier otra representación que quiera.



Nuevo requerimiento: Pago 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 seleccionará una de las pasarelas de pago disponibles, digitará la información de la tarjeta y de su dueño, y procederá a realizar el cobro.

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 el pago, 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).

Entrega única: documentación e implementación

Actividades

1. Realice el diseño y construya un documento de diseño donde presente el diseño con las justificaciones para las decisiones clave que hayan tomado. El documento debe incluir por lo menos los siguientes elementos:
    a. Al menos un diagrama de clases que incluya todas las clases del sistema, incluyendo sus relaciones, atributos y métodos. Los diagramas deben cubrir tanto la interfaz como la parte del diseño dedicado a la lógica de dominio.
    b. Un diagrama de clases de alto nivel, que incluya todas las clases del sistema y sus relaciones.
    c. Un diagrama de clases de alto nivel de la interfaz, que muestre qué elementos se incluye y cómo se relacionan con los elementos del dominio.

Estos elementos NO son los únicos que debe incluir su documento. Con seguridad hay muchos más elementos que usted considerará relevantes sobre su diseño.

2. Implemente el sistema que diseñó. Tenga en cuenta que los detalles de la implementación deben ser coherentes tanto con el modelo de clases, como con los diagramas de secuencia que incluya dentro del documento de diseño.

No se evaluarán implementaciones que no tengan un documento de diseño actualizado que las acompañe.

Entrega

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

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:

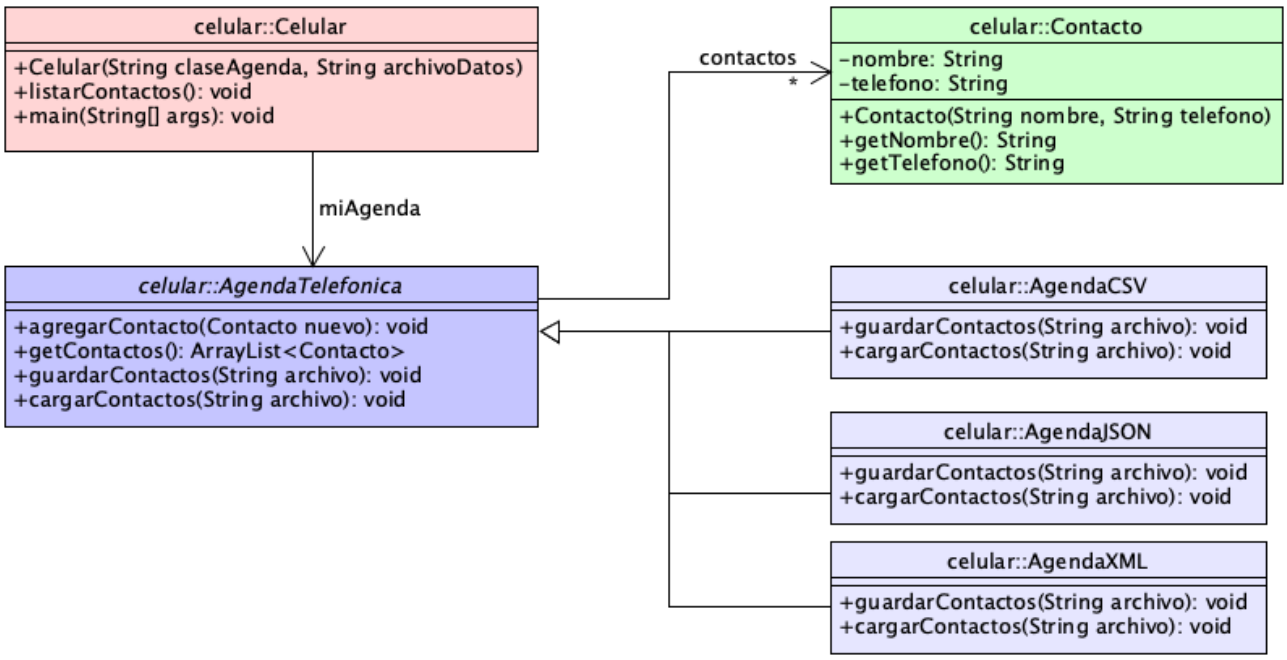
- El método estático `forName` de la clase `Class`, que nos permite cargar una clase a partir de su nombre (cadena de caracteres). Este método retorna un objeto de la clase `Class` que tiene toda la información de la clase de la que nos dieron el nombre.
- El método `getDeclaredConstructor(Object[] tipos)` 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

```
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;

   /**
```

```java
    * 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();
   }

}
```