UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Laboratorio 4, Multiparadigma

Integrantes: Aldo Castillo

Curso: Paradigmas de Programación

Profesor: Roberto González

21 de Septiembre de 2020

Tabla de contenidos

1. Introducción

1.1. Motivación

El principal motivo por el cual se realiza este proyecto es desafiar los conocimientos sobre distintos lenguajes y el poder implementarlos para lograr llegar al final del proyecto de manera satisfactoria. A su vez, otra motivación es el hecho de crear algo que podrá ser usado por un usuario, logrando así un producto final funcional.

1.2. Descripción del problema

En esta ocasión enfrentaremos la problemática que impone la abstracción de Git (Figura 1) abarcado desde el punto de vista del ramo Paradigmas de programación, tal como dice el nombre del ramo, se verán distintos paradigmas (en distintos lenguajes) ejemplificados en esta problemática, algunos de estos son: Paradigma Funcional (scheme – R6RS), Paradigma Lógico (Prolog), Paradigma Orientado a Objetos (Java) entre otros. A final de semestre se espera poder complementar principalmente estos tres paradigmas (y lenguajes) y así tener una app totalmente funcional con interfaz gráfica aplicada con *C#*

Para esta cuarta instancia, abordaremos el problema con un lenguaje Multiparadigma, *C#*. con el cual deberemos implementar ciertas funciones que se detallaran más adelante. El principal paradigma del lenguaje *C#* es el Paradigma orientado a Eventos, este consta en llamar funciones cuando el usuario haga una acción (apretar un botón, deslizar una barra, elegir una opción, etc.). Con esto se debe solucionar la problemática de la abstracción de Git, su representación y trabajo interno.

Al ser un lenguaje Multiparadigma, se puede reutilizar el código antes hecho en Paradigma Orientado a Objeto en Java con algunas adaptaciones.

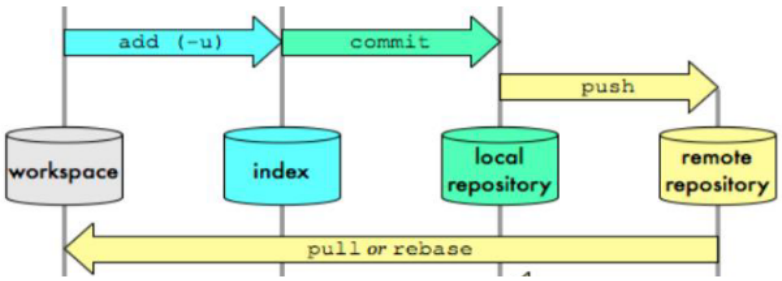


Figura 1: Abstracción de Git.

1.3. Solución propuesta

Para abordar esta problemática, la solución de dividió en varias partes:

1. Clase Repositorio
2. Clase Zones
3. Clase Work Space
4. Clase
5. Clase Local Repository
6. Clase Remote Repository
7. Clase Commit
8. Clase Archivo
9. Funciones complementarias
10. Distintas vistas del GUI

Estas se detallarán más adelante.

1.4. Objetivo general y alcances del proyecto

1.4.1. Objetivo general

El objetivo general de esta cuarta parte del proyecto es generar una aplicación totalmente funcional, la cual podrá realizar todo lo necesario para representar el uso de Git además de generar una interfaz grafica amigable y entendible.

1.4.2. Objetivos específicos

1. Crear y representar de manera adecuada la Clase Repositorio.
2. Crear y representar de manera adecuada la Clase Zones.
3. Crear y representar de manera adecuada la Clase Work Space.
4. Crear y representar de manera adecuada la Clase Index.
5. Crear y representar de manera adecuada la Clase Local Repository.
6. Crear y representar de manera adecuada la Clase Remote Repository.
7. Crear y representar de manera adecuada la Clase Commit.
8. Crear y representar de manera adecuada la Clase Archivo.
9. Realizar funciones Init, Add, Commit, Push, Pull y Status, además de sus funciones auxiliares.
10. Diseñar y crear las vistas del GUI para cada Clase

1.4.3. Alcances

En este informe se quiere dar a conocer el funcionamiento de esta cuarta parte del proyecto, mostrando sus fundamentos teóricos, métodos y herramientas ocupadas. En cuanto al laboratorio se espera realizar una abstracción funcional con los estándares de validación necesarios.

1.4.4. Metodologías y herramientas

Para la realización de este laboratorio se utilizaron los apuntes de clase, sumado a las librerías de *C#* para el manejo de distintos tipos de datos y funciones.

2. Desarrollo de la experiencia

2.1. Fundamentos teóricos

Para la realización de este laboratorio, se ocuparon diversos tipos de datos y funciones nativas de *C#*. Estas se detallarán a continuación.

2.1.1. Vistas

Las Vistas la forma en que se les llama a los Formularios de Windows, estos son los que ve e interactúan con el usuario. Haciendo uso del IDE Visual Studio 2019 se pueden usar de manera amigable, arrastrando botones y viendo una preview del GUI en todo momento (Figura 2).

Figura 2: Ejemplo de una vista.

2.1.2. Eventos

Los eventos son la forma de funcionar de esta parte del proyecto. Un evento se ejecuta cuando el usuario realiza alguna acción que este disponible en la vista correspondiente. Es decir, si el usuario aprieta un botón, desliza una barra, marca una opción o incluso si cierra la ventana actual esto hará que se ejecute un evento. Estos eventos son manejados por los EventHandler que marcan el comportamiento de esa acción.

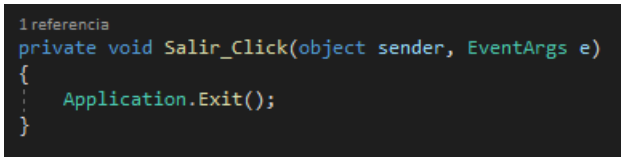


Figura 3: Ejemplo de un evento.

2.1.3. Clases

Como se mostró en los anteriores laboratorios, existen los TDA, una clase es algo muy similar a estos. Son abstracciones de lo que nosotros queremos crear. Estos tienen atributos y métodos.

Un atributo es una característica de la clase, , mientras que un método es una funcionalidad de esta.

Por ejemplo, la clase Auto. Esta posee como atributo:

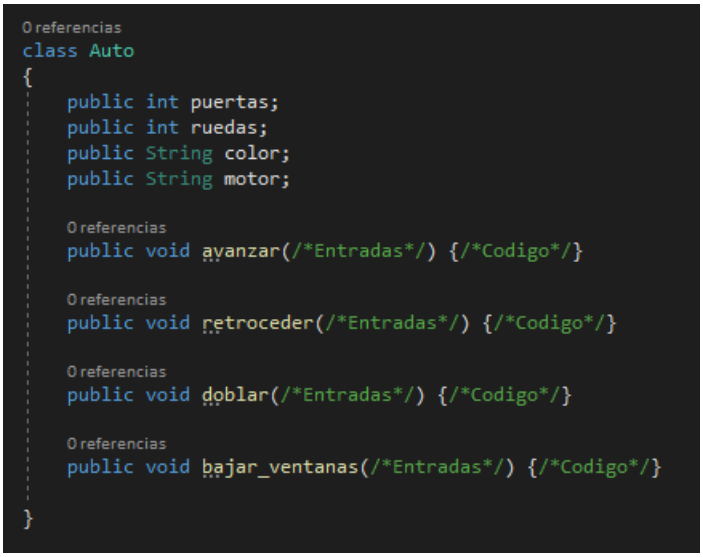
- Tiene puertas.
- Tiene ruedas.
- Tiene color.
- tiene motor.

Y como métodos:

- Avanzar.
- Retroceder.
- Doblar.
- Bajar ventanas.

Para efectos de este laboratorio, las clases usadas son:

- Repositorio
- Zones



```
0 referencias
class Auto
{
    public int puertas;
    public int ruedas;
    public String color;
    public String motor;

    0 referencias
    public void avanzar(/*Entradas*/) {/*Codigo*/}

    0 referencias
    public void retroceder(/*Entradas*/) {/*Codigo*/}

    0 referencias
    public void doblar(/*Entradas*/) {/*Codigo*/}

    0 referencias
    public void bajar_ventanas(/*Entradas*/) {/*Codigo*/}

}
```

Figura 4: Clase de ejemplo Auto.

- Work Space
- Index
- Local Repository
- Remote Repository
- Commit
- Archivo

2.1.4. Objetos

Un objeto nace desde una clase, tomando el ejemplo anterior se puede decir que es "un" Auto, el cual tiene los atributos y métodos anteriormente descritas pero con valores conocidos, osea:

- Tiene *4* puertas.
- Tiene *4* ruedas.
- Tiene color *Rojo*.
- tiene motor *Diesel*.

Pueden haber muchos objetos de una misma clase, todos con distintos valores dentro de los atributos descritos, mientras que todos compartirán los mismos métodos.

```
0 referencias
class Prueba
{
    0 referencias
    public void prueba()
    {
        Auto auto = new Auto(puertas: 4, ruedas: 4, color: "rojo", motor: "Diesel");
    }
}
```

Figura 5: Objeto de ejemplo Auto.

2.1.5. List

Tal como dice su nombre, son una lista dinámica de elementos. Estas pueden ser de elementos nativos de *C#* (String, int, Boolean, etc) o de Clases creadas para el contexto del proyecto (Clase Auto). Al ser una lista dinámica posee Métodos que nos ayudan en el manejo de esta (Add(), RemoveAt(), Count(), etc.).

```
Auto auto1 = new Auto(puertas: 4, ruedas: 4, color: "rojo", motor: "Diesel");

List<Auto> autos = new List<Auto>();
autos.Add(auto1); // Agrega el auto1 a la lista
autos.Count(); // Entrega el largo de la lista, en este caso 1
autos.RemoveAt(0); // Elimina el elemento en la posicion dada
```

Figura 6: Ejemplo List tipo Auto.

2.2. Desarrollo de la solución

Ya que se reutilizo el código de Java el diagrama de clases viene siendo muy parecido, solo que algunas funciones que se usaban para recibir datos desde consola, ahora no son necesarias por lo que se borraron. Ademas, gitPull y gitPush al ser funciones que solo cambian de lugar una lista de archivos no se necesito de implementación mas profunda.

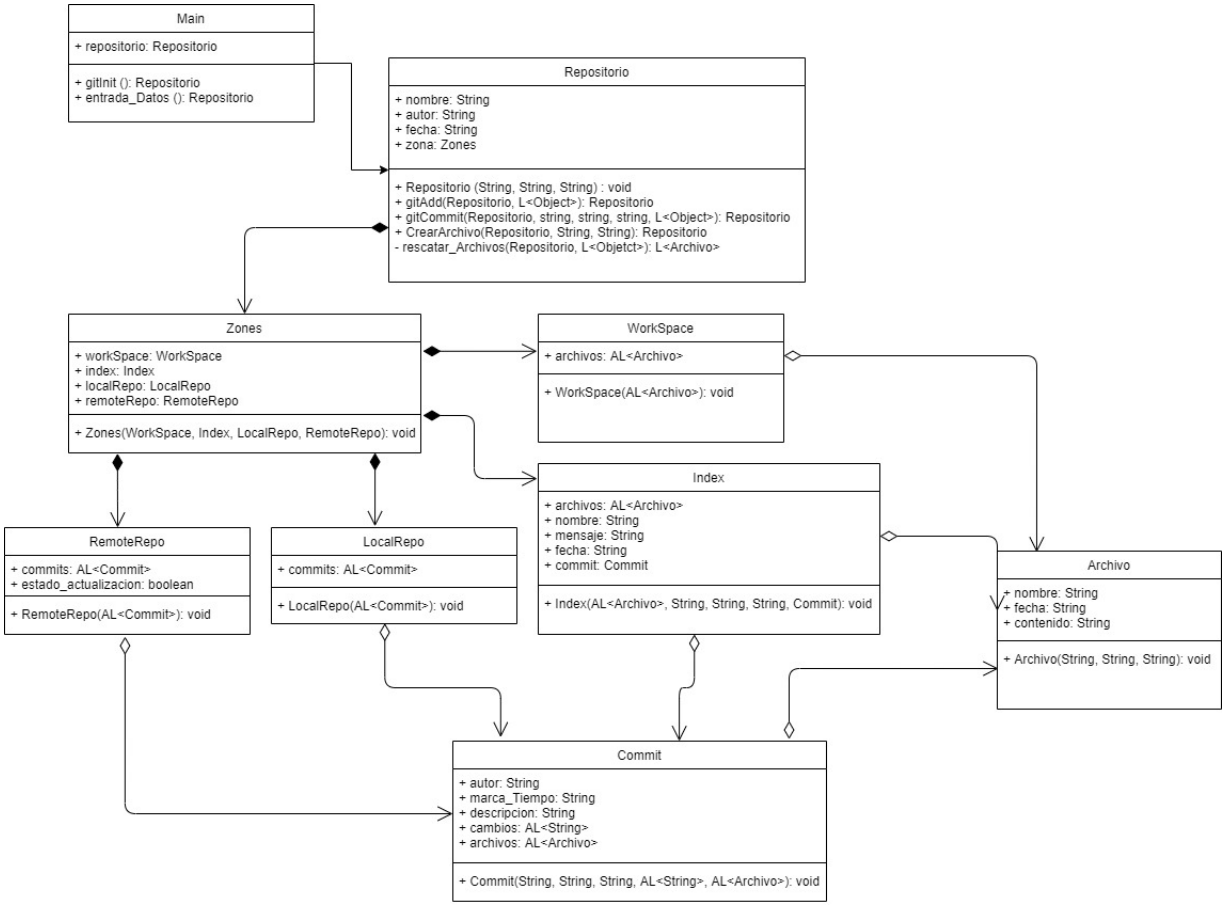


Figura 7: Diagrama de clases.

Se puede acceder a esta imagen en mejor tamaño Acá

Al dar inicio al programa se abrirá la vista1, la cual nos da la bienvenida al programa, tiene dos botones los cuales permiten salir e inicializar un repositorio. Al apretar el segundo botón se abre la vista2 que nos pregunta por el nombre del repositorio y su autor, si aceptamos y los datos son validos se abre la vista3, esta vista es la principal ya que nos permite ejecutar todas las funciones obligatorias que se detallaran mas adelante.

2.2.1. Vista 3

Vista principal del programa donde se accede a todas las funcionalidades del proyecto. Esta vista nos muestra los botones necesarios para acceder a todo, además de un botón de salida y dos etiquetas que nos dicen el nombre del repositorio y el autor de este (Figura 8).

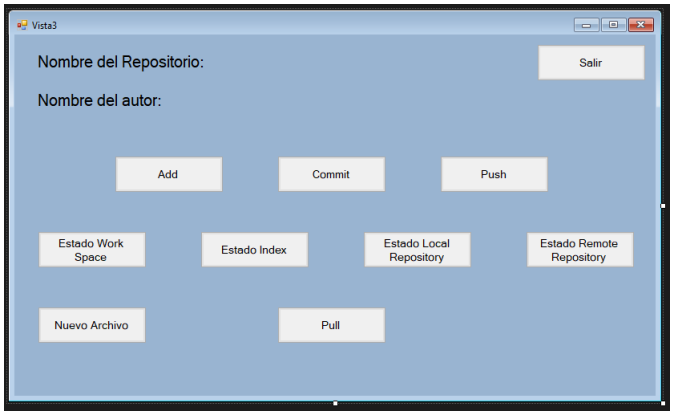


Figura 8: Ventana Vista3.

2.2.1.1 Botones "Status"

Son los encargados de mostrar el estado de una zona de trabajo (Work Space, Index, Local repository o Remote repository). Cada uno de estos llama una vista distinta que nos muestra el estado de cada uno. Los botones de Work Space e Index nos muestra los archivos de texto disponibles en estas zonas. Los botones de Local repository y Remote repository nos muestran los Commits guardados en estas zonas, además, al seleccionar un commit nos muestra los archivos que están contenidos en este.

Figura 9: Vista StatusWS.

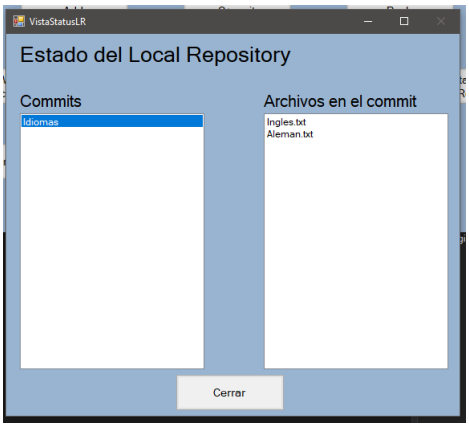


Figura 10: Vista StatusLR.

2.2.1.2 Botón "Nuevo Archivo"

Es el botón que le corresponde llamar a la vistaNuevoArchivo la cual nos pregunta por el nombre del archivo y su contenido. Al apretar el botón Çrear"se llama la función CrearArchivo de la clase Repositorio la cual se encarga de crear el objeto nuevo y darle los valores rescatados además de la marca de tiempo.

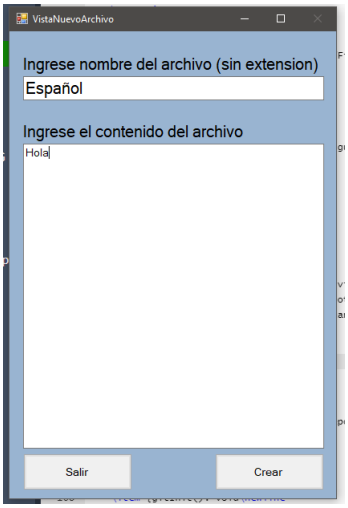


Figura 11: Ventana Nuevo Archivo.

2.2.1.3 Botón "Add"

Este botón llama a la vistaAdd, la cual nos pregunta que archivos deseamos agregar al Index, esto por medio de una CheckedListBox la cual nos permite marcar un Ticket en los archivos que queremos agregar, al marcar un archivo este se moverá a la derecha indicando que esta seleccionado. Al apretar el botón "Aceptar" se llama la función gitAdd de la clase Repositorio la cual agrega los archivos marcados al Index.

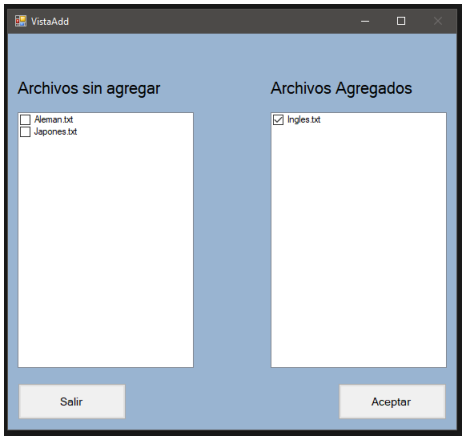


Figura 12: Ventana Add

2.2.1.4 Botón "Commit"

Este botón llama a la vistaCommit, la cual nos pregunta el nombre que deseamos ponerle al commit, el autor y un mensaje descriptivo de este. Al apretar el botón "Siguiente" nos preguntara que archivos deseamos agregar a este commit con el mismo sistema que en "Add". Cuando uno acepta por medio del botón correspondiente, se llama la función gitCommit de la clase Repositorio, la cual se encarga de generar el Commit y añadirlo a la lista de commits de Local Repository.

Ademas, al hacer un commit se cambia el estado del Remote repository a "No actualizado".

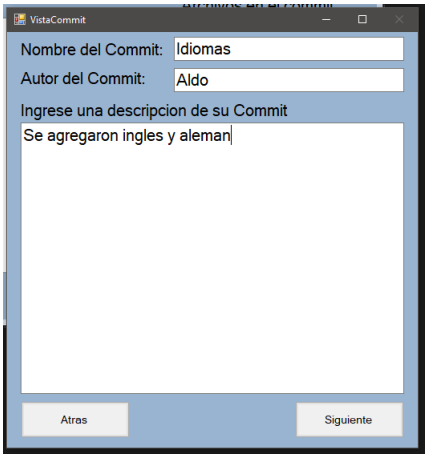


Figura 13: Vista gitCommit 1.

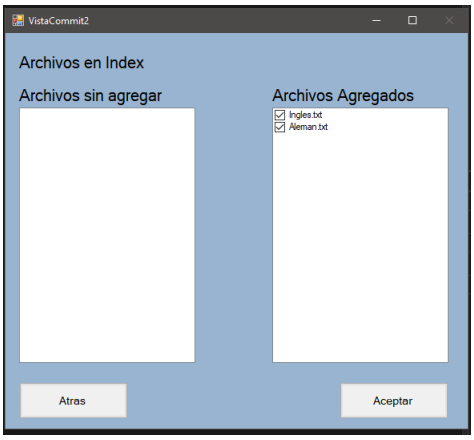


Figura 14: Vista gitCommit 2.

2.2.1.5 Botón "Push"

Este botón ejecuta dos lineas de código las cuales copia la lista de commits de Local repository al Remote repository, además actualiza el estado del mismo a "Actualizado". Al completar esto arroja un MessageBox anunciándolo.

2.2.1.6 Botón "Pull"

Al igual que el botón "Push", "Pull" mueve la lista de archivos del ultimo commit del remote repository al Work Space, Ademas compara estos para reemplazar aquellos que tengan el mismo nombre. Al completar esto arroja un MessageBox anunciándolo.

3. Análisis de resultados

Como se puede ver en el desarrollo de la solución se lograron todos los aspectos requeridos. Lo que resulto mas difícil fue el modificar la recepción de datos desde Java a C#, ya que hubo que cambiar el modo de pedir datos por consola, a rescatar los datos desde TextBox para asignar los a valores pedidos. Mas allá de eso, las ventanas y el manejo de eventos no presentaron mayor problema.

4. Conclusión

Según los resultados obtenidos se puede decir que se logro el cien por ciento de lo que se esperaba en un principio, además de que el plan inicial funcionó debido a la experiencia en el uso de java para distintas aplicaciones y las aproximadas 2 semanas de diseño del código.

Los objetivos específicos se completaron y fueron una guía para llevar a cabo el laboratorio, se realizaron las clases en orden de importancia para ir desde lo macro del proyecto a lo micro.

El conocimiento de lo que es y como funcionan las Clases fueron parte importante del desarrollo, ya que antes de tratar de hacer cualquier función se identificaron las relaciones correspondientes y las funciones que se necesitarían para lograr las funciones obligatorias, esto hizo que fueran relativamente fácil de programar.

Bibliografía

Advance, R. C. (2019). Mostrar la hora y fecha actual en tiempo real con c#, vb.net y winform. [Online] https://rjcodeadvance.com/como-mostrar-la-hora-y-fecha-actual-con-c-tipos-de-formato/.

Docs, M. (s/i). Enumerable.union method. [Online] https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.union?view=netcore-3.1.

Grepper (s/i). "convert ienumerable to list c#" code answer. [Online] https://www.codegrepper.com/code-examples/csharp/convert+ienumerable+to+list+c%23.

Net-informations (s/i). C# checked listbox control. [Online] http://csharp.net-informations.com/gui/cs-checkedlistbox.htm.