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

LABORATORIO 4

PARADIGMAS DE PROGRAMACIÓN

Gabriel Gaete L.

Profesores: Roberto González
Daniel Gacitúa
Víctor Flores

Fecha de Entrega:7 de Septiembre del 2018

Santiago de Chile
1 - 2018

TABLA DE CONTENIDO

TABLA DE FIGURAS

CAPÍTULO 1. INTRODUCCIÓN

Un *chatbot* o un bot conversacional es *"un robot capaz de hablar e interactuar imitando el comportamiento humano, ya sea oral o por escrito, respondiendo a las preguntas y reclamos de los usuarios."* (Herrero, s.f.) En otras palabras, es un programa que permite simular una conversación con una persona, entregando respuestas automáticas a entradas hechas por un usuario. Esta conversación, habitualmente se establece a través de texto, sin embargo, hay modelos que disponen de una interfaz multimedia. También se han desarrollado chatbots que utilizan conversores de texto a sonido, dotando de mayor realismo a la interacción con el usuario.

El presente informe tiene por objetivo principal ser una referencia a una línea de pensamiento y a un contexto de desarrollo del código fuente que lo acompaña para la presentación del cuarto laboratorio del curso *"Paradigmas de programación"*. En esta oportunidad, se hará uso del paradigma de orientación a objetos y dirigida por eventos bajo el lenguaje de programación *C#*.

El paradigma de orientación a objetos (*Object-oriented Programming)* corresponde a un paradigma basado en el concepto de "objetos" que contienen información, conocida comúnmente como *atributos,* y que también son capaces de realizar ciertas acciones u operaciones que son propias de este objeto, las cuales se conocen como *métodos.*

La programación dirigida por eventos es un paradigma de programación en el cual el flujo de la ejecución de un programa está dirigida por *eventos,* por ejemplo, una acción de un usuario, como un click, el presionar una tecla, etc. La principal diferencia con los paradigmas vistos anteriormente, radica en que los programas dirigidos por eventos se encuentran en "espera" de que un evento suceda para actuar. Esto hace que los programas sean del tipo "reactivo".

El problema a resolver en este laboratorio es el desarrollo de un *chatbot*, con el cual se deberá mantener una conversación básica, protocolar, en la que exista un flujo conversacional coherente. El contexto para este chatbot será una venta de pasajes a capitales regionales dentro de Chile.

La solución implementada trabaja en base a un conjunto de clases que definen el comportamiento de los elementos que interactúan en el programa, tales como el Chatbot, un usuario, el chat propiamente tal, los mensajes intercambiados, y el registro de mensajes que se han intercambiado (es decir, un *log*). Estas clases son las que componen el Back-End del programa, manejando datos e interactuando a través de eventos que son manejados por la librería GTK# con su interfaz gráfica o Front-End.

Se establece entonces, como objetivo en este laboratorio, el aplicar y demostrar los conocimientos adquiridos en cátedra con respecto tanto a lo que es el paradigma de programación orientado a objetos como dirigido por eventos, utilizando el lenguaje de progamación *C#*.

CAPÍTULO 2. MARCO TEÓRICO

2.1 PARADIGMA ORIENTADO A OBJETOS

El paradigma orientado a objetos es una forma de construir, diseñar e implementar programas basados en comunidades de objetos que combinan estados, comportamientos e identidades. El estado está compuesto por datos, también llamados atributos; el comportamiento está definido por una serie de acciones que se realizan y son los llamados métodos, mientras que la identidad es lo que diferencia a un objeto del resto de los objetos dentro de esta comunidad.

Los programas bajo este paradigma se basan en la relación y la colaboración que se pueda establecer entre objetos. La comunicación entre los diferentes objetos se da mediante "mensajes" entre ellos. Estos mensajes son llevados a cabo mediante el llamado a métodos de un cierto objeto.

2.2 CLASES

Es una definición de un objeto. Es un contenedor de uno o más datos, junto con las propiedades que lo manipulan llamados *"métodos".* En otras palabras, se puede decir que una clase es el modelo a través del cual se generarán los objetos dentro del programa. Como este modelo, que describe las propiedades es un conjunto de variables (atributos) y métodos que pueden ser utilizados como objetos en cualquier punto del programa, es que las clases son uno de los pilares fundamentales de la programación bajo el paradigma de orientación a objetos.

2.3 OBJETO

Es una unidad compacta, que se encuentra en tiempo de ejecución y que realiza las tareas propias de un programa. Los objetos en programación se utilizan para modelar objetos o entidades del mundo real (el objeto perro, gato, o tienda, por ejemplo). Un objeto es, en otras palabras, la representación dentro de un programa de un concepto, y contiene toda la información necesaria para abstraerlo: datos que permiten describir sus atributos y operaciones que pueden realizarse sobre los mismos.

De forma rápida, podría decirse también que un objeto es una *instancia* de una clase.

2.4 EVENTO

Es una acción o acontecimiento reconocido por el programa, a menudo originado de forma asíncrona en un contexto externo al código, pero siendo este quien los maneja.

CAPÍTULO 3. DESCRIPCIÓN DEL PROBLEMA

Se solicita crear un programa en el lenguaje de programación *C#* que simule un bot conversacional (chatbot). Para este laboratorio, el tema del chatbot será una venta de pasajes a capitales regionales de Chile. Este debe funcionar en base a ciertas clases/estructuras como lo son un chatbot, un usuario, y un *log* o historial de conversaciones. El programa debe ser poseer una interfaz gráfica desde la cual se puedan realizar las siguientes funcionalidades:

1. **Iniciar Chat (seed):** Esta funcionalidad permite iniciar el chatbot con un valor semilla (seed) para su personalidad. Si se omite el valor semilla, se cargará una personalidad predeterminada (default). Si ya se estaba ejecutando una conversación, se deberá terminar ésta y luego reiniciar la conversación con el nuevo chatbot.

2. **Terminar Chat:** Esta funcionalidad permite terminar la conversación. El chatbot entrega un mensaje de despedida y luego de unos segundos, la conversación se cierra. Al momento de terminar la conversación, se debe poder utilizar la funcionalidad Rate.

3. **Enviar Mensajes:** El programa debe permitir enviar mensaje al Robot en cada momento. El chat debe mostrar los mensajes enviados por el usuario y la respuesta del Chatbot. Este chat debe indicar la hora de los mensajes, quién envía el mensaje y el contenido del mensaje.

4. **Guardar Log:** Esta función permite guardar toda la interacción con el Chatbot desde que se inició el chat en un archivo de texto. El programa debe permitir escoger la ubicación y el nombre donde se guardará el archivo log.

5. **Rate:** Una vez se ha cerrado una conversación aparecerá una ventana en la que se le pedirá al usuario calificar su interacción con el chatbot. Se le debe indicar al usuario la escala en la que debe calificar o se puede calcular la nota en base a un pequeño cuestionario presentado al usuario. En base a las respuestas entregadas por el usuario, se calcula la nota.

6. **Cargar Log**: Esta función permite cargar una conversación almacenada con la función Guardar Log. Para esto el programa debe permitir escoger al usuario un archivo en su computador donde está el log. El programa debe ser capaz de identificar si el archivo escogido es un log o no. Una vez cargado el log, la ventana del chat debe cargarse con todas las interacciones entre el usuario y el chatbot. Posteriormente se puede reanudar esta conversación desde el momento en que se guardó.

El programa debe trabajar en base a un conjunto de clases para el desarrollo de la implementación del chatbot, las cuales son Chatbot, Usuario y Log. Además, se tiene la libertad de definir e implementar éstas clases acorde a las necesidades de la implementación deseada.

3.1 ANÁLISIS

El principal problema que se presenta en este laboratorio radica en cómo definir las clases necesarias para la implementación de la solución. A primera vista, son necesarias 4 clases (Usuario, Chatbot, Log y Mensaje), las cuales conformaran lo que es el Back-End de la solución. Posterior a la realización del Back-End, se debe escoger una librería gráfica para permitir el intercambio de mensajes entre el Usuario y el Chatbot. De esta forma, e incluyendo una clase para la ventana principal, una aproximación del diagrama de clases será como se muestra en la figura:

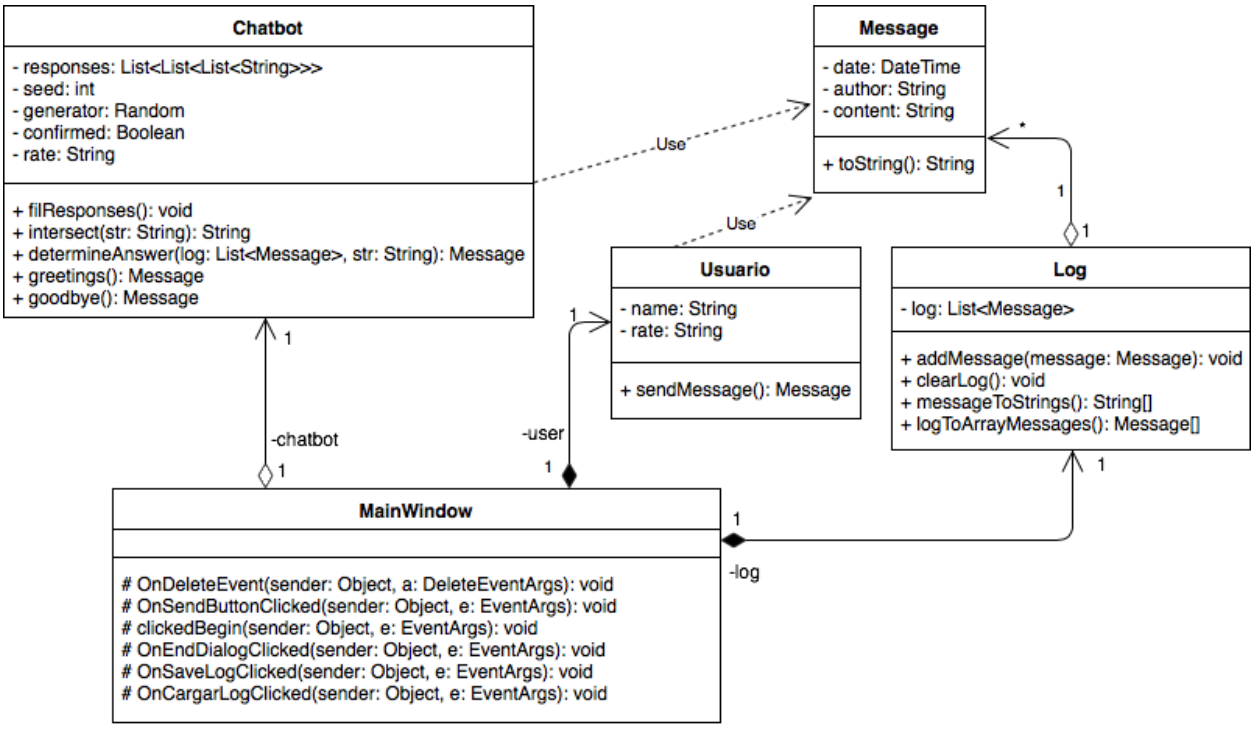

Figura 1 Diagrama de Clases inicial

CAPÍTULO 4. DESCRIPCIÓN DE LA SOLUCIÓN

Para llevar a cabo este programa, se han utilizado 4 clases propias y un conjunto de clases de C#, principalmente *List* y *MainWindow,* para lo que es el manejo de las respuestas que es capaz de entregar el chatbot y el uso de ventanas para la interfaz gráfica, respectivamente. La clase principal del programa corresponde a *MainWindow*, siendo aquí donde se instancian los demás objetos, como los participantes de este chat, con su respectivo historial de mensajes. En esta misma clase, es donde se realiza la distinción en lo que ha realizado el usuario, es decir, mediante eventos, se determina qué cosa quiere realizar, si escribir un log, si mandar un mensaje, etc.

La siguiente figura, permite reflejar las clases creadas y la relación que mantienen entre ellas. De esta forma, fue implementada la solución al problema.

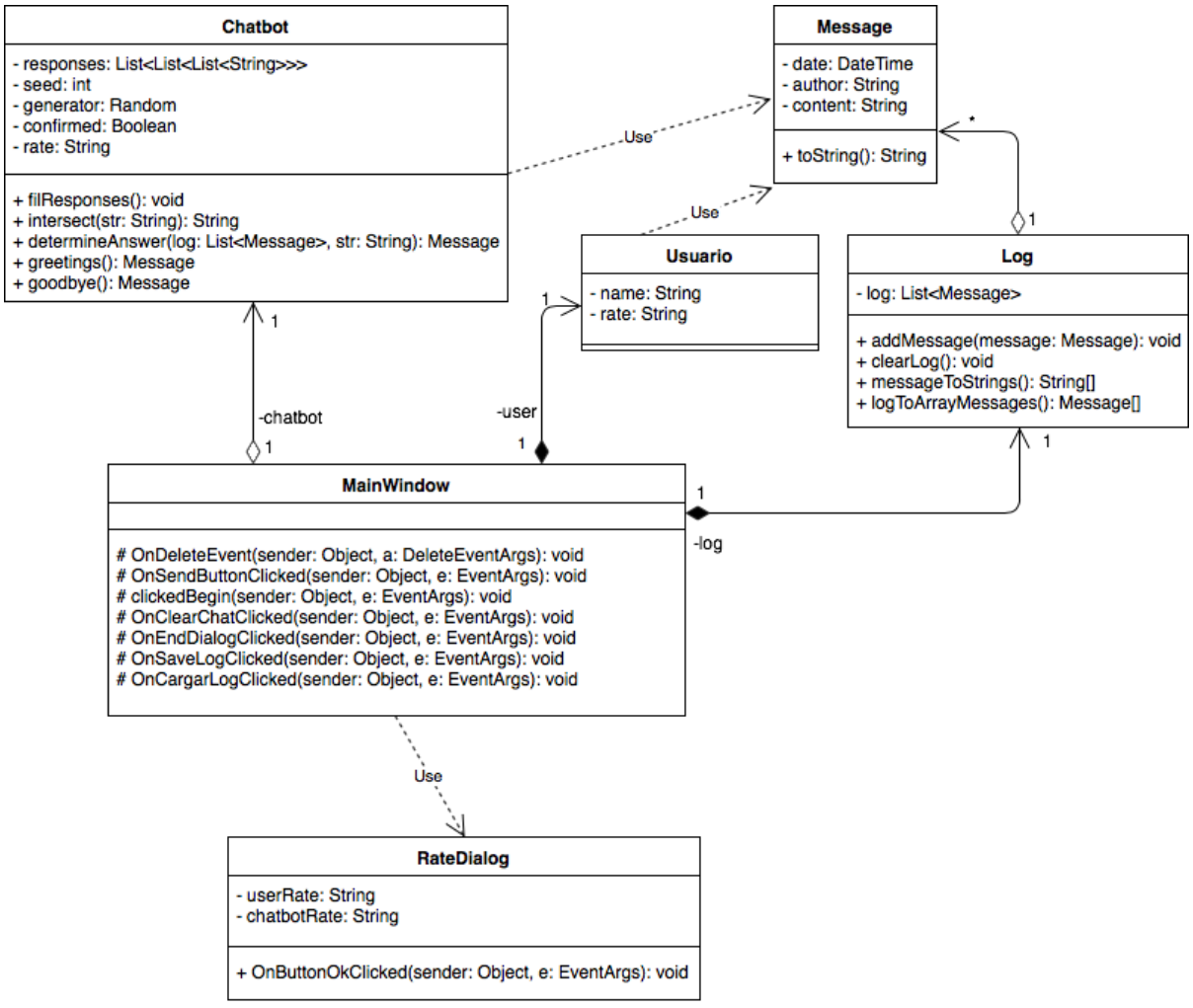

Figura 2 Diagrama de Clases final

Al comparar el diagrama de clases inicial con el final, se puede ver que tanto la clase Usuario como la clase MainWindow sufrieron modificaciones. Además de esto, se ha tenido la necesidad de crear una nueva clase, para la ventana o cuadro de diálogo que permite al usuario ingresar el rate. Inicialmente, se había previsto que la clase Usuario iba a contener un método que le permitiera enviar mensajes, cosa que finalmente quedó completamente bajo el control de la ventana principal. Por otra parte, se le agregó un evento a la ventana principal, que permite "limpiar" la pantalla, cuando se tiene demasiado texto, reiniciando el contenido.

Antes de entrar de lleno en lo que son las clases requeridas por el enunciado, se debe saber que tanto la clase Usuario, como la clase Chatbot y la clase Log, trabajan en base a la clase Mensaje (o *Message*). Esta clase *Message* permite mantener un mayor orden y claridad a la hora de almacenar los mensajes dentro del registro. En ésta clase *Message*, se establece un nombre o remitente del mensaje, una fecha de envío, y el contenido del mensaje. Su método más importante es *toString()*, el cual permite obtener los datos del mensaje siguiendo un patrón establecido. La siguiente figura, muestra el código fuente de la clase *Message*.

```
using System;
namespace ChatbotBackend
{
    public class Message
    {
        private DateTime date;
        private String author;
        private String content;

        public Message(DateTime date, String author, String content)
        {
            this.date = date;
            this.author = author;
            this.content = content;
        }

        public String getContent(){
            return this.content;
        }

        public String toString(){
            return date.ToString("[dd/MM/yyyy, H:mm:ss] ") + this.author + this.content + "\n";
        }
    }
}
```

Figura 3 Representación de un mensaje

Una vez realizada la representación de un mensaje, se debe determinar quiénes son los que interactúan directamente con los mensajes, es decir, quiénes son los que participan dentro de la conversación. Para este programa, se separó en dos clases, un usuario que desea comprar pasajes, y un chatbot, que hace de vendedor de pasajes. La clase usuario se utiliza principalmente para almacenar un nombre de usuario, y asignarle una nota de autoevaluación final.

A diferencia del Usuario, la clase Chatbot posee una mayor lógica detrás, puesto que no es sólo enviar mensajes al azar, sino que se debe lograr una correcta interpretación de los mensajes del usuario, para de esta forma, permitir una conversación lógica y con sentido. Para esto, se ha realizado un método dentro de la clase Chatbot que permite intersectar strings, con el fin de identificar palabras "clave" que guien al Chatbot a responder de una manera más asertiva. Dado que la clase es de una extensión considerable, las siguientes figuras muestran los métodos más importantes para la correcta implementación del chat.

```
public List<String> intersect(String str){
    List<List<String>> cities;
    List<List<String>> positive;
    List<String> splittedString;
    splittedString = new List<String>();
    positive = new List<List<String>>();
    cities = new List<List<String>>();
    splittedString.Add(str);
    positive.Add(new List<String>(new String[] { "sí"}));
    positive.Add(new List<String>(new String[] { "si"}));
    cities.Add(new List<String>(new String[] {"Arica", "$32.000 pesos."}));
    cities.Add(new List<String>(new String[] { "Iquique", "$30.100 pesos." }));
    cities.Add(new List<String>(new String[] { "Antofagasta", "$21.600 pesos." }));
    cities.Add(new List<String>(new String[] { "Copiapó", "$15.000 pesos." }));
    cities.Add(new List<String>(new String[] { "La Serena", "$9.100 pesos." }));
    cities.Add(new List<String>(new String[] { "Valparaíso", "$6.500 pesos." }));
    cities.Add(new List<String>(new String[] { "Rancagua", "$3.000 pesos." }));
    cities.Add(new List<String>(new String[] { "Concepción", "$13.900 pesos." }));
    cities.Add(new List<String>(new String[] { "Puerto Montt", "$19.900 pesos." }));
    cities.Add(new List<String>(new String[] { "Coyhaique", "$33.000 pesos." }));
    cities.Add(new List<String>(new String[] { "Punta Arenas", "$15.000 pesos." }));
    cities.Add(new List<String>(new String[] { "Valdivia", "$17.900 pesos." }));

    foreach (List<String> city in cities){
        if (str.ToLower().Contains(city[0].ToLower())){
            return city;
        }
    }

    foreach (List<String> pos in positive){
        if (str.ToLower().Contains(pos[0].ToLower())){
            return pos;
        }
    }

    if (str.ToLower().Contains("no")){
        return new List<String>(new String[] { "no" });
    }

    return null;
}
```

Figura 4 Método para intersectar mensajes del usuario

```
public Message greetings(){
    DateTime date = DateTime.Now;
    this.generator = new Random();
    int position = (int)((this.generator.NextDouble() * 20) % 2);
    String response = this.responses[this.seed][0][position];

    return new Message(date, "Chatbot [>]: ", response);
}
```

Figura 5 Método que permite al Chatbot dar la bienvenida al Usuario.

La clase Log es casi una copia de algunos métodos de la clase *List*, puesto que la manera de llevar un registro de mensajes, ha sido a través de una estructura *List*, sin embargo, la clase Log se ha creado con la finalidad de mantener una mayor claridad y estructuración a la hora de implementar el código. De todas formas, además de estos métodos, la clase Log cuenta con dos métodos importantes, que permiten transformar el Log, tanto en un arreglo de mensajes, como en un arreglo de strings. Esto se hace para que la escritura en un archivo .log sea más fácil de ser implementada. La siguiente figura muestra el código fuente de la clase Log.

```
using System;
using System.Collections.Generic;
using ChatbotBackend;
namespace ChatbotBackend
{
    public class Log
    {
        private List<Message> log;

        public Log(List<Message> log)
        {
            this.log = log;
        }

        public Log(){
            this.log = new List<Message>();
        }

        public void addMessage(Message message){
            this.log.Add(message);
        }

        public void clearLog(){
            this.log.Clear();
        }

        public List<Message> getLog(){
            return this.log;
        }

        public String[] messagesToStrings(){
            List<String> listOfStrings;
            listOfStrings = new List<String>();

            foreach (Message msg in this.log){
                listOfStrings.Add(msg.toString())
            }

            return listOfStrings.ToArray();
        }

        public Message[] logToArrayMessages(){
            return this.log.ToArray();
        }
    }
}
```

Figura 6 Código fuente de la clase Log

Por último, las siguientes figuras muestran los algoritmos creados para implementar las instrucciones listadas en el capítulo 3. Estos algoritmos son disparados por un evento ocurrido en la interfaz gráfica, por lo que están dentro de manejadores de eventos, pertenecientes a la clase *MainWindow*.

```
protected void OnSaveLogClicked(object sender, EventArgs e)
{
    if (this.log.getLog().Count != 0)
    {
        FileChooserDialog fcd = new FileChooserDialog("Guardar Historial", this, FileChooserAction.Save,
        "Seleccionar Directorio", ResponseType.Ok, "Cancelar", ResponseType.Close);
        fcd.SelectMultiple = false;
        fcd.CurrentName = "filename_historial";
        fcd.Run();
        String[] messages = this.log.messagesToStrings();

        if (fcd.Filename != null){
            File.WriteAllLines(fcd.Filename + ".log", messages);
            textview1.Buffer.Text += "Sistema [!]: Archivo log escrito satisfactoriamente.\n";
        }

        fcd.Destroy();
    }
    else
    {
        textview1.Buffer.Text += "Sistema [!]: El log no contiene mensajes. Por favor, inicie una conversación.\n";
    }
}
```

Figura 7 Implementación evento "Guardar Log"

```
protected void clickedBegin(object sender, EventArgs e)
{
    int seed = 0;
    this.log.clearLog();

    if (seedview.Buffer.Text != "" && int.TryParse(seedview.Buffer.Text, out seed))
    {
        this.chatbot = new Chatbot(seed);
        seedview.Buffer.Text = "";
    }
    else
    {
        this.chatbot = new Chatbot();
    }
    Message msg = this.chatbot.greetings();

    textview1.Buffer.Text += msg.toString();
    this.log.addMessage(msg);
}
```

Figura 8 Implementación evento "Iniciar chat (seed)".

En la figura 8, dado que dentro de los requisitos no se especifíca cómo ingresar una semilla, se ha introducido un cuadro de texto que permite ingresar un eventual número entero como semilla. En caso de no ingresar una entrada válida (número entero), se toma una semilla por defecto.

```
protected void OnEndDialogClicked(object sender, EventArgs e)
{
    if (this.chatbot != null)
    {
        Message msg = this.chatbot.goodbye();
        this.log.addMessage(msg);
        this.chatbot = null;
        textview1.Buffer.Text += msg.toString();

        RateDialog rateDialog = new RateDialog();
        rateDialog.Run();

        this.log.addMessage(new Message(DateTime.Now, "Sistema [!]: ", "Nota del Usuario: " + rateDialog.getUserRate()));
        this.log.addMessage(new Message(DateTime.Now, "Sistema [!]: ", "Nota del Chatbot: " + rateDialog.getChatbotRate()));

        textview1.Buffer.Text += "Sistema [!]: Nota del Usuario -> " + rateDialog.getUserRate() + "\n";
        textview1.Buffer.Text += "Sistema [!]: Nota del Chatbot -> " + rateDialog.getChatbotRate() + "\n";
        textview1.Buffer.Text += "Sistema [!]: Notas puestas de manera satisfactoria. Esto se verá reflejado en el log.\n";
    }
    else
    {
        textview1.Buffer.Text += "Sistema [!]: No se puede finalizar una conversación no iniciada!. Por favor, inicie una conversación.\n";
    }
}
```

Figura 9 Implementación evento "Terminar Chat" y Rate

```
protected void OnCargarLogClicked(object sender, EventArgs e)
{
    FileChooserDialog fcd = new FileChooserDialog("Leer Historial", this, FileChooserAction.Open,
    "Seleccionar Archivo", ResponseType.Ok, "Cancelar", ResponseType.Close);
    fcd.Run();

    if (fcd.Filename != null && fcd.Filename.Contains(".log"))
    {
        textview1.Buffer.Text += "Sistema [!]: Archivo log cargado satisfactoriamente.\n";
        String[] lines = File.ReadAllLines(fcd.Filename);
        foreach (String str in lines)
        {
            if (str != "\n")
                textview1.Buffer.Text += str + "\n";
        }
    }
    else
    {
        textview1.Buffer.Text += "Sistema [!]: No se ha proporcionado un archivo log.\n";
    }

    fcd.Destroy();
}
```

Figura 10 Implementación evento "Cargar Log"

Como se puede ver en la figura 10, la implementación de *"Cargar Log"*, a pesar de poder filtrar de buena manera los archivos .log, no permite continuar una conversación desde el punto que el log indica, por lo que se considera que la implementación de cargar el log ha quedado al debe.

Por último, la implementación de la clase *RateDialog* no es compleja algorítmicamente hablando, puesto que sólo se encarga de recibir datos mediante un combobox (a modo de formulario), en donde el usuario expresa su nota, como se puede ver en la siguiente figura.

```
using System;
namespace ChatbotFrontend
{
    public partial class RateDialog : Gtk.Dialog
    {
        String userRate;
        String chatbotRate;

        public RateDialog()
        {
            this.Build();
            this.userRate = "0";
            this.chatbotRate = "0";
        }

        protected void OnButtonOkClicked(object sender, EventArgs e)
        {
            this.userRate = comboboxUser.ActiveText;
            this.chatbotRate = comboboxChatbot.ActiveText;
        }

        public String getUserRate()
        {
            return this.userRate;
        }

        public String getChatbotRate()
        {
            return this.chatbotRate;
        }
    }
}
```

Figura 11 Implementación clase RateDialog

Por último, dado el largo de la funcionalidad *"Enviar Mensaje"*, se ha decidido intencionalmente no incluir el código fuente dentro de este informe.

CAPÍTULO 5. RESULTADOS

A pesar de tener complejidades algorítmicas que podrían inducir a una lenta carga del chatbot, los tiempos de ejecución de las instrucciones no presentan un tiempo de respuesta notable para el usuario, siendo la carga del archivo log lo que más tiempo demora.

La siguiente figura muestra el resultado presentado a la hora de utilizar el botón *"Iniciar Chat"*, en este caso, sin el uso de una semilla.

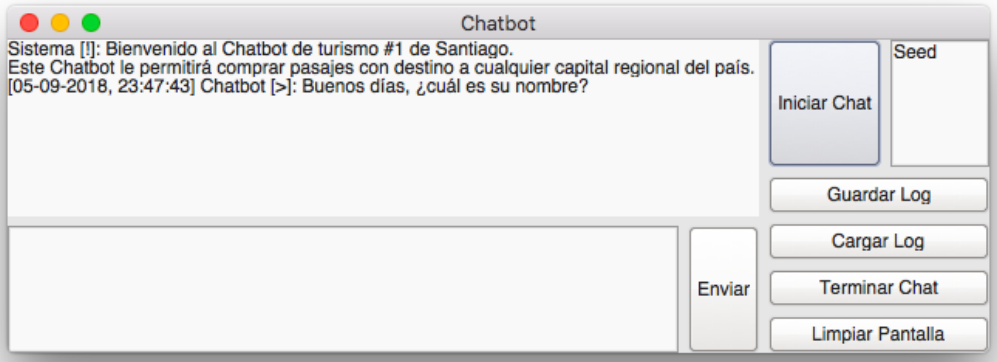

Figura 12 Ejemplo de inicio de chat

Por otra parte, a la hora de mantener una conversación entre usuario y chatbot, los tiempos de respuesta son relativamente rápidos. La siguiente imagen muestra el resultado de una conversación a través de su flujo esperado.

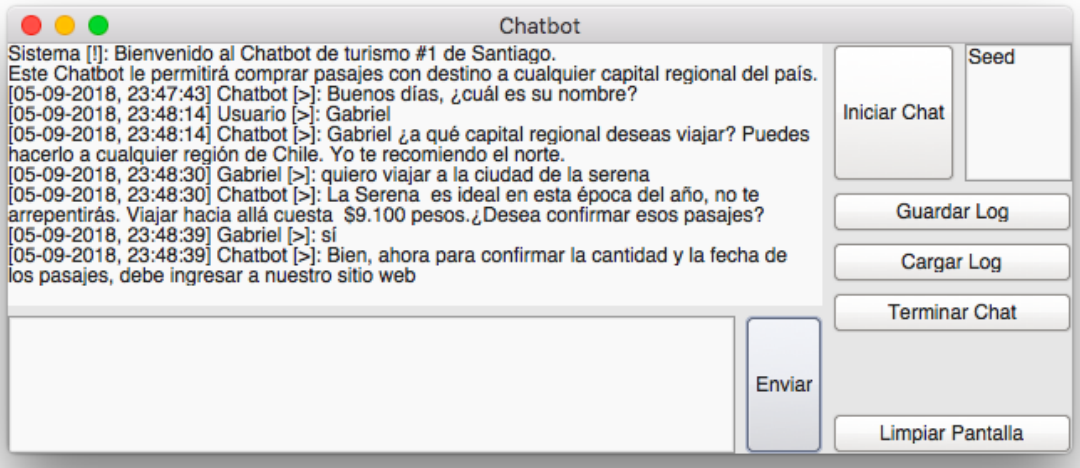

Figura 13 Ejemplo de intercambio de mensajes

Además de permitir el intercambio de mensajes entre usuario y chatbot, manteniendo un flujo normal y coherente de una conversación, el botón *Guardar Log* permite al usuario ingresar un directorio junto con un nombre de archivo donde guardar un registro de lo conversado hasta el momento, como ejemplifican las siguientes figuras.

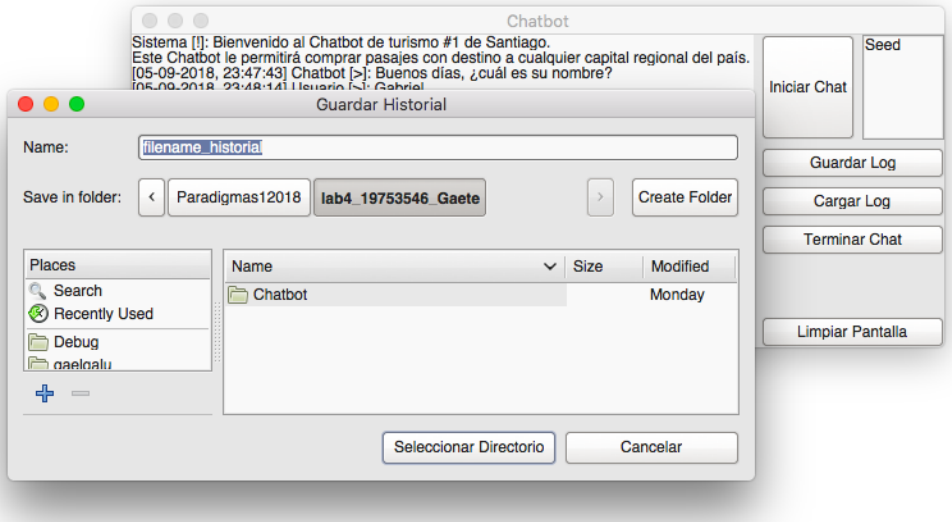

Figura 14 Ventanas creadas mediante evento "Guardar Log"

El archivo log generado en la figura 14, es registrado dentro de un archivo .log. Este archivo almacena la conversación que se lleva hasta el momento entre usuario y chatbot. Al guardar este registro, además de mantener el mensaje y el remitente, también se almacena la respectiva marca de tiempo, tanto para saber la fecha como la hora en que un mensaje fue enviado. La siguiente figura permite observar el resultado generado dentro del archivo, el cual se encuentra con el nombre y en el directorio especificados.

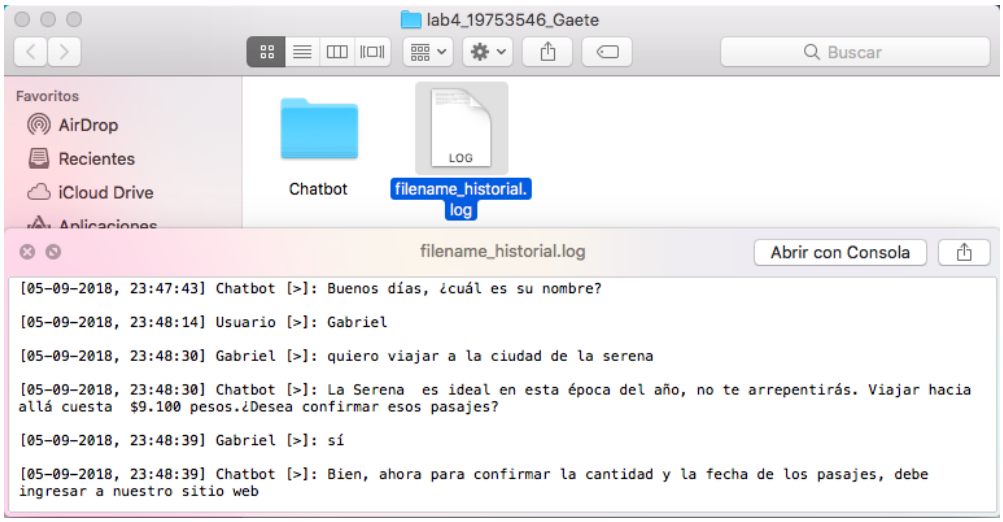

Figura 15 Registro dentro del log

Por último, para terminar la conversación con el chatbot, es necesario utilizar el botón *"Terminar Chat".* Sin embargo, debido a los requerimientos del programa, al presionar este botón, se abrirá la siguiente ventana.

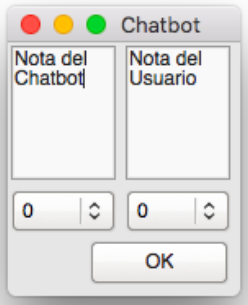

Figura 16 Ventana de rate

A través de esta ventana, es posible asignarle una nota tanto al chatbot como al usuario (auto evaluación). En caso de no ingresar una nota (mediante el uso de los combobox), se establece que la nota no ha podido ser determinada, quedando con una calificación "0".

En simultáneo, cuando se presiona el botón que permite terminar el chat, se puede ver un mensaje de despedida del chatbot. Cabe destacar, que el programa no se cierra al momento de terminar un chat. Esto se ha dejado a propósito, en caso de que otro usuario (o incluso el mismo), tengan la posibilidad tanto de comenzar una nueva conversación (tal vez con el uso de una semilla distinta), de cargar un log, o en su defecto, de guardar el log de la última conversación tenida. La siguiente figura retrata la despedida del chatbot.

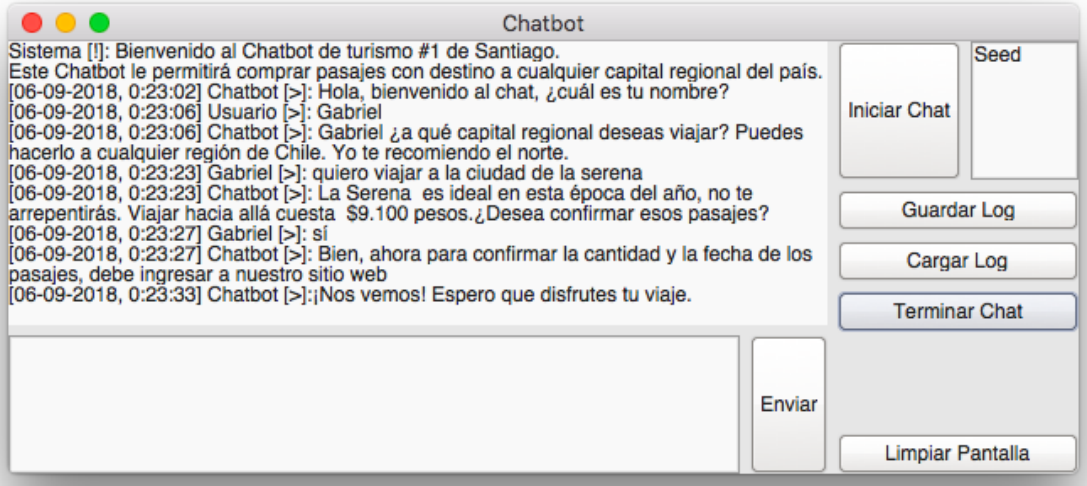

Figura 17 Despedida del chatbot

CAPÍTULO 6. CONCLUSIONES

El paradigma orientado a objetos permite reflejar de una mejor manera la realidad, además de agrupar bajo un mismo elemento no sólo la información, sino también sus respectivos comportamientos y características (funciones propias, métodos, cambios de estado, incluso atributos), lo cual produce un mayor orden a la hora de estructurar soluciones, en contraste con los paradigmas vistos anteriormente, que no poseen ese nivel de estructuración.

Sin embargo, una desventaja evidente del paradigma orientado a objetos, es que la interpretación de un objeto queda totalmente a criterio del programador, por lo que dos programadores pueden llegar a ideas totalmente diferentes acerca de un mismo objeto, requiriendo una documentación más extensa en comparación a los paradigmas vistos anteriormente, como lo pueden ser el funcional y el lógico.

A pesar de la desventaja mencionada, el paradigma orientado a objetos sigue siendo una herramienta que permite un fácil mantenimiento, dado lo sencillo que resulta modificar las clases existentes.

Al ser comparado con los paradigmas vistos anteriormente, para la realización de este proyecto resulta una mejor herramienta el paradigma orientado a objetos, debido a la facilidad que nos entrega de abstraer el problema y llevarlo a una solución estructurada. Además, la programación dirigida por eventos, facilita el uso de una interfaz gráfica, puesto que permite entender a las acciones de un usuario como una serie de eventos, los cuales cada uno posee cierta “respuesta”. A pesar de esto, una debilidad presentada durante la realización del laboratorio, es que la librería utilizada se encarga casi sola del manejo de los eventos y de asociarlos a sus respectivas funciones, lo cual, a pesar de que agiliza la programación, también le quita el control al mismo programador para trabajar. Esto se ve reflejado en el archivo con la declaración de widgets de la interfaz gráfica, el cual se crea sólo, además de actualizarse tras cada cambio en la misma, es decir, no es posible editar dicho código. Cabe mencionar, que esto es un problema de la librería, y no del paradigma en sí.

El lenguaje de programación C# entrega diversas y vastas herramientas para confeccionar soluciones, debido a esto, el tiempo de desarrollo para la creación del chatbot fue bastante reducido en comparación a laboratorios anteriores.

Finalmente, dado que se ha logrado desarrollar un *chatbot* o *bot conversacional* como ha sido requerido, de paso demostrando los conocimientos expuestos en la cátedra de la clase y se ha obtenido el resultado esperado, es posible concluir que se ha logrado cumplir con el objetivo de este laboratorio. Sin embargo, no ha sido posible completar del todo los requerimientos pedidos (**Cargar Log**), debido a la forma en que la clase fue estructurada se ha dificultado el proceso, siendo imposible implementar al 100% esta funcionalidad dentro de los plazos establecidos para el laboratorio.

CAPÍTULO 7. REFERENCIAS

Herrero, C. (s.f.). No son mis cookies. Recuperado el 21 de Abril de 2018, de No son mis cookies: http://nosinmiscookies.com/que-es-un-chatbot/