



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

**LABORATORIO 3
PARADIGMAS DE PROGRAMACIÓN**

Gabriel Gaete L.

Profesores: Roberto González
Daniel Gacitúa
Víctor Flores

Fecha de Entrega: 10 de Agosto del 2018

Santiago de Chile

1 - 2018

TABLA DE CONTENIDO

TABLA DE FIGURAS	2
CAPÍTULO 1. INTRODUCCIÓN	3
CAPÍTULO 2. MARCO TEÓRICO	4
2.1 PARADIGMA ORIENTADO A OBJETOS	4
2.2 CLASES	4
2.3 OBJETO.....	4
CAPÍTULO 3. DESCRIPCIÓN DEL PROBLEMA	5
3.1 ANÁLISIS.....	5
CAPÍTULO 4. DESCRIPCIÓN DE LA SOLUCIÓN.....	6
CAPÍTULO 5. RESULTADOS	12
CAPÍTULO 6. CONCLUSIONES.....	15
CAPÍTULO 7. REFERENCIAS.....	16

TABLA DE FIGURAS

Figura 1 Diagrama de Clases	6
Figura 2 Representación de un mensaje	7
Figura 3 Método para intersectar mensajes del usuario.....	8
Figura 4 Método que permite al Chatbot dar la bienvenida al Usuario.	9
Figura 5 Código fuente de la clase Log.....	9
Figura 6 Código fuente de la clase Main.....	10
Figura 7 Implementación instrucción !rate	10
Figura 8 Implementación instrucción !saveLog.....	10
Figura 9 Implementación instrucción !beginDialog seed	11
Figura 10 Implementación !endDialog.....	11
Figura 11 Ejemplo de beginDialog	12
Figura 12 Ejemplo de intercambio de mensajes	12
Figura 13 Ejemplo de !saveLog.....	13
Figura 14 Registro dentro del log	13
Figura 15 Ejemplo de !endDialog y !exit.....	14

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 segundo laboratorio del curso “*Paradimas de programación*”. En esta oportunidad, se hará uso del paradigma de orientación a objetos bajo el lenguaje de programación *Java*.

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*.

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*).

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

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.

CAPÍTULO 3. DESCRIPCIÓN DEL PROBLEMA

Se solicita crear un programa en el lenguaje de programación *Java* 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 *log* o historial de conversaciones, y un usuario. El programa debe ser capaz de realizar las siguientes funcionalidades:

1. **!beginDialog 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. **!endDialog:** Ésta funcionalidad termina la conversación con el chatbot, entregando éste último un mensaje de despedida.
3. **!saveLog:** Escribe en un archivo de texto plano el log completo de la conversación actual (incluyendo marcas de tiempo e información relevante del chatbot). El archivo será guardado en el directorio en que se ejecuta el programa y su formato del nombre de archivo se basará en la marca de tiempo en la que se envió el comando. Ej: 2018-05-30_15-45.log
4. **!rate notaChatbot notaUsuario:** Permite evaluar el desempeño del chatbot (y del usuario) tras haber terminado una conversación. Las evaluaciones serán entre 1 y 5, o 0 si no se puede determinar. Dichos puntajes deberán almacenarse con su marca de tiempo.

Estas funcionalidades tienen un formato definido para su implementación y desarrollo, trabajan con estructuras específicas (principalmente listas). Se tiene la libertad de definir e implementar éstas estructuras 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 5 clases (Usuario, Chatbot, Log, Mensaje y Chat). De esta forma, un diagrama inicial de clases sería como muestra la siguiente figura:

CAPÍTULO 4. DESCRIPCIÓN DE LA SOLUCIÓN

Para llevar a cabo este programa, se han utilizado 6 clases propias y un conjunto de clases de Java, principalmente *ArrayList* y *Scanner*, para lo que es el manejo de las respuestas que es capaz de entregar el chatbot. La clase principal del programa corresponde Chat, 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 dicho el Usuario, es decir, si es que el Usuario ha ingresado una instrucción especial, o de lo contrario, sólo quiere intercambiar un mensaje con el bot.

Debido al gran tamaño, se ha decidido intencionalmente no incluir el código fuente dentro de éste documento.

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

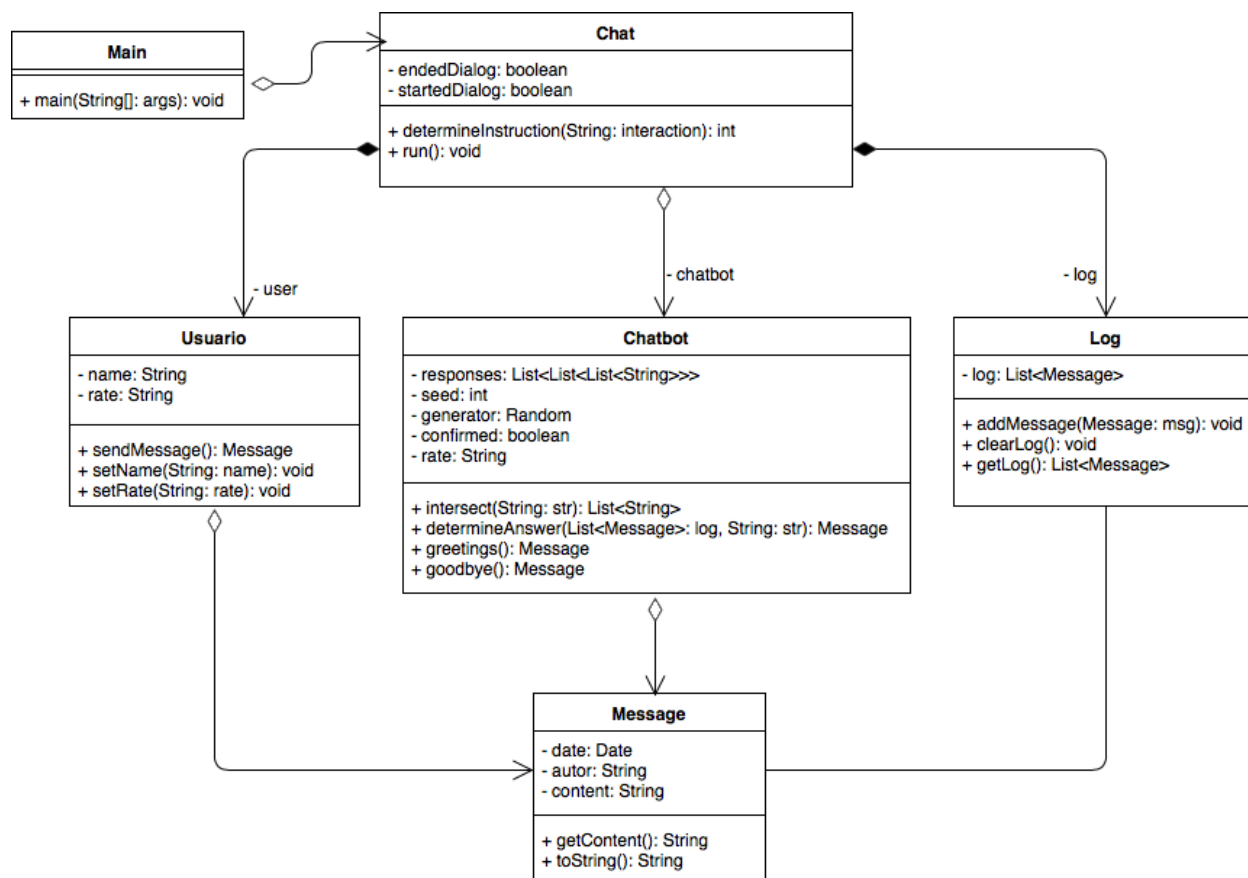


Figura 1 Diagrama de Clases

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*.

```
import java.util.*;
import java.lang.*;
import java.text.*;

public class Message{
    private Date date;
    private String autor;
    private String content;

    public Message(Date date, String autor, String content){
        this.date = date;
        this.autor = autor;
        this.content = content;
    }

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

    public String toString(){
        DateFormat df = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

        return "[" + df.format(this.date) + "]" + this.autor + ": " + this.content;
    }
}
```

Figura 2 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. El método más importante de esta clase es *sendMessage()*, la cual permite al usuario intercambiar mensajes dentro del chat. La siguiente figura, muestra el código fuente de la clase Usuario.

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 guíen 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 = Arrays.asList(Arrays.asList("Arica", "$32.200 pesos."), Arrays.asList("Iquique",
"$30.100 pesos"), Arrays.asList("Antofagasta", "$21.600 pesos"), Arrays.asList("Copiapó", "$15.000 pesos"),
Arrays.asList("La Serena", "$9.100 pesos"), Arrays.asList("Valparaíso", "$6.500 pesos"), Arrays.asList("Rancagua",
"$3.000 pesos"), Arrays.asList("Talca", "$6.500 pesos"), Arrays.asList("Concepción", "$13.900 pesos"),
Arrays.asList("Temuco", "$14.900 pesos"), Arrays.asList("Puerto Montt", "$19.900 pesos"),
Arrays.asList("Coyhaique", "$33.000 pesos"), Arrays.asList("Punta Arenas", "$15.000 pesos"),
Arrays.asList("Valdivia", "$17.900 pesos"));

    List<List<String>> positive = Arrays.asList(Arrays.asList("sí"), Arrays.asList("si"));

    List<String> splittedStr = Arrays.asList(str);

    for (List<String> city : cities) {
        if (str.toLowerCase().contains(city.get(0).toLowerCase())){
            return city;
        }
    }

    for (List<String> pos : positive){
        if (str.toLowerCase().contains(pos.get(0).toLowerCase())) {
            return pos;
        }
    }

    if (str.toLowerCase().compareTo("no") == 0){
        return Arrays.asList("no");
    }

    return null;
}

```

Figura 3 Método para intersectar mensajes del usuario

```

public Message greetings(){
    Date date = new Date();

    int position = (int) ((this.generator.nextDouble() * 20) % 2);
    String response = this.responses.get(this.seed).get(0).get(position);
    System.out.println("Chatbot [>]: " + response);

    Message message = new Message(date, "Chatbot", response);

    return message;
}

```

Figura 4 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 utilizar esta estructura. La siguiente figura muestra el código fuente de la clase Log.

```

import java.util.*;

public class Log{
    private List<Message> log;

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

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

    public void addMessage(Message msg){
        this.log.add(msg);
    }

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

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

```

Figura 5 Código fuente de la clase Log

Por un tema de orden y de buenas prácticas, la última clase corresponde a la clase *Main*, donde se instancia el laboratorio en general, y se ejecuta el chat, como muestra la Figura 6.

```

public class Main{
    public static void main(String[] args) {
        Chat chat = new Chat();

        chat.run();
    }
}

```

Figura 6 Código fuente de la clase Main

Por último, las siguientes figuras muestran los algoritmos creados para implementar las instrucciones listadas en el capítulo 3.

```

if (splitedString.length != 3){
    System.out.println("Sistema [!]: A la instrucción especificada le falta un parámetro. Por favor ingrese nuevamente.");
} else {
    DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
    Date dateToRate = new Date();
    String strDate = dateFormat.format(dateToRate);
    this.chatbot.setRate(strDate + splitedString[1]);
    this.user.setRate(strDate + splitedString[2]);
}

```

Figura 7 Implementación instrucción !rate

```

Date date = new Date();
DateFormat df = new SimpleDateFormat("dd-MM-yyyy_HH-mm-ss");
String fileName = df.format(date);
fileName = fileName + ".log";

try {
    PrintWriter writer = new PrintWriter(fileName, "UTF-8");
    for (Message messageInLog : this.log.getLog()){
        writer.write(messageInLog.toString() + "\n");
    }

    writer.close();

    System.out.println("Sistema [!]: Archivo log generado satisfactoriamente.");
}
catch (Exception e){
    e.printStackTrace();
}

```

Figura 8 Implementación instrucción !saveLog

```

if (splitedString.length == 1)
    this.chatbot = new Chatbot();
else
    this.chatbot = new Chatbot(Integer.parseInt(splitedString[1]));

finished = false;

System.out.println("Se ha iniciado correctamente el chat");

this.log.clearLog();
this.log.addMessage(new Message(new Date(), "Usuario", msg.getContent()));

Message greetings = this.chatbot.greetings();
this.log.addMessage(greetings);

System.out.print("Usuario [>]: ");
String name = sc.nextLine();
this.user.setName(name);

Message nameMessage = new Message(new Date(), "Usuario", name);

```

Figura 9 Implementación instrucción !beginDialog seed

```

finished = true;
this.startedDialog = false;
Message goodbye = this.chatbot.goodbye();
this.user.setName("Usuario");
this.log.addMessage(goodbye);
this.log.clearLog();

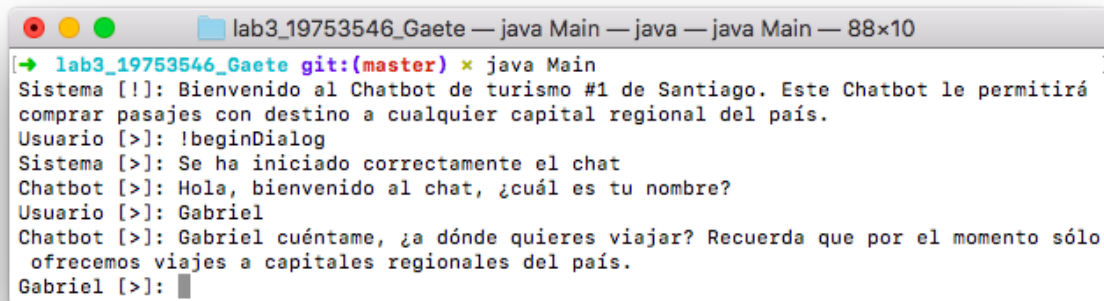
```

Figura 10 Implementación !endDialog

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 creación del archivo log el que más tiempo demora.

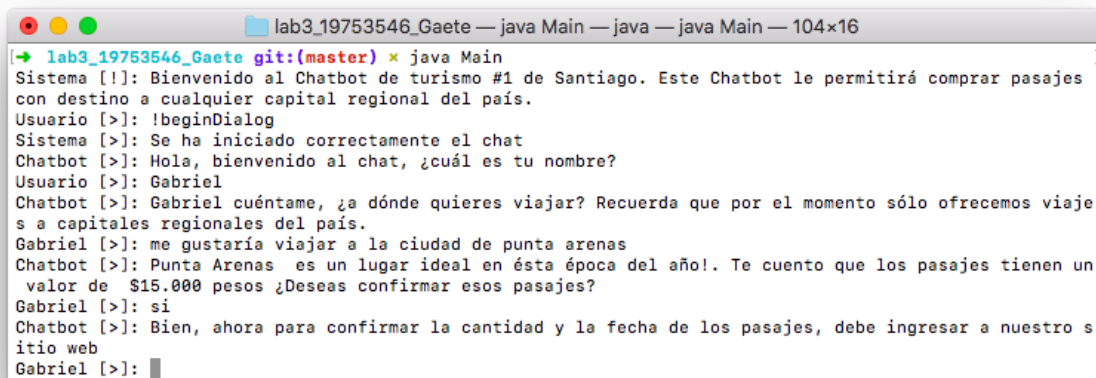
La siguiente figura muestra el resultado presentado a la hora de utilizar la instrucción `!beginDialog`, en este ejemplo, sin el uso de una semilla.



```
lab3_19753546_Gaete — java Main — java — java Main — 88x10
[→ lab3_19753546_Gaete git:(master) * java Main
Sistema [!]: Bienvenido al Chatbot de turismo #1 de Santiago. Este Chatbot le permitirá comprar pasajes con destino a cualquier capital regional del país.
Usuario [>]: !beginDialog
Sistema [>]: Se ha iniciado correctamente el chat
Chatbot [>]: Hola, bienvenido al chat, ¿cuál es tu nombre?
Usuario [>]: Gabriel
Chatbot [>]: Gabriel cuéntame, ¿a dónde quieres viajar? Recuerda que por el momento sólo ofrecemos viajes a capitales regionales del país.
Gabriel [>]:
```

Figura 11 Ejemplo de `beginDialog`

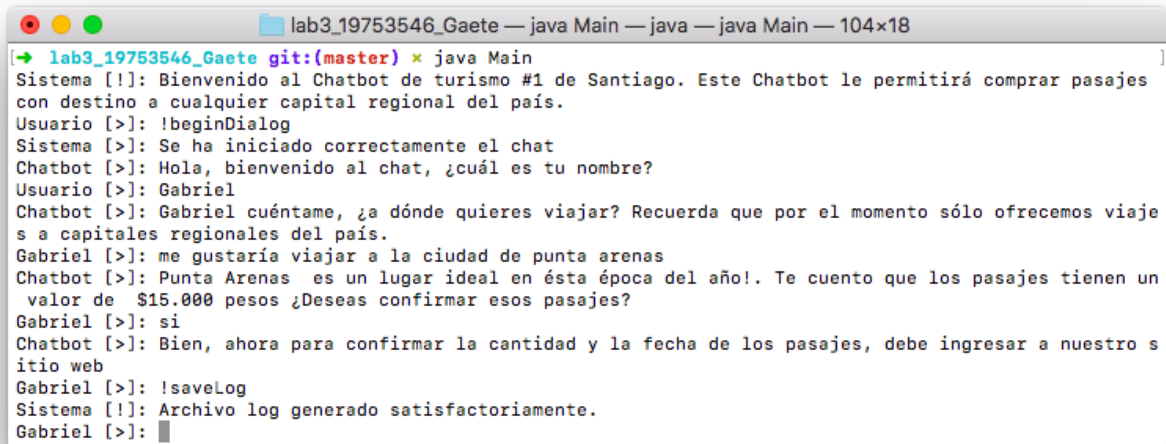
Por otra parte, a la hora de mantener una conversación entre usuario y chatbot, los tiempos de respuesta son relativamente rápidos. La siguiente imagen muestra el resultado de una conversación a través de su flujo esperado.



```
lab3_19753546_Gaete — java Main — java — java Main — 104x16
[→ lab3_19753546_Gaete git:(master) * java Main
Sistema [!]: Bienvenido al Chatbot de turismo #1 de Santiago. Este Chatbot le permitirá comprar pasajes con destino a cualquier capital regional del país.
Usuario [>]: !beginDialog
Sistema [>]: Se ha iniciado correctamente el chat
Chatbot [>]: Hola, bienvenido al chat, ¿cuál es tu nombre?
Usuario [>]: Gabriel
Chatbot [>]: Gabriel cuéntame, ¿a dónde quieres viajar? Recuerda que por el momento sólo ofrecemos viajes a capitales regionales del país.
Gabriel [>]: me gustaría viajar a la ciudad de punta arenas
Chatbot [>]: Punta Arenas es un lugar ideal en ésta época del año!. Te cuento que los pasajes tienen un valor de $15.000 pesos ¿Deseas confirmar esos pasajes?
Gabriel [>]: si
Chatbot [>]: Bien, ahora para confirmar la cantidad y la fecha de los pasajes, debe ingresar a nuestro sitio web
Gabriel [>]:
```

Figura 12 Ejemplo de intercambio de mensajes

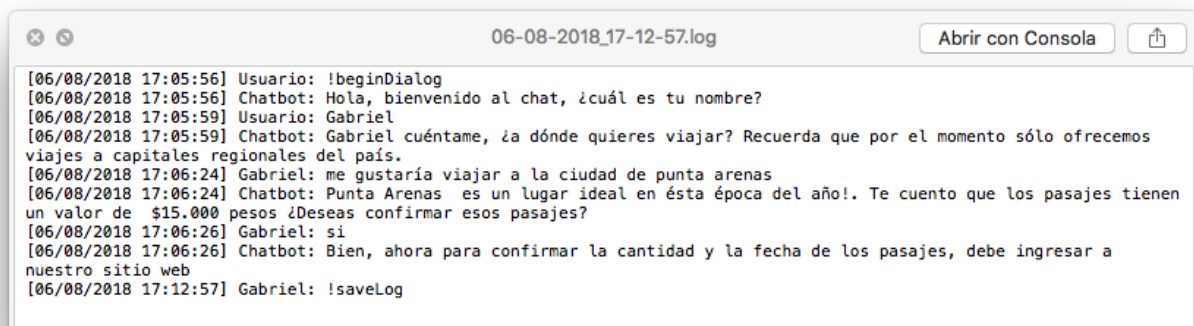
Además de permitir el intercambio de mensajes entre usuario y chatbot, manteniendo un flujo normal y coherente de una conversación, la instrucción **!saveLog** permite guardar un registro de lo conversado hasta el momento, como ejemplifican las siguientes figuras.



```
lab3_19753546_Gaete — java Main — java — java Main — 104x18
[→ lab3_19753546_Gaete git:(master) * java Main
Sistema [!]: Bienvenido al Chatbot de turismo #1 de Santiago. Este Chatbot le permitirá comprar pasajes
con destino a cualquier capital regional del país.
Usuario [>]: !beginDialog
Sistema [>]: Se ha iniciado correctamente el chat
Chatbot [>]: Hola, bienvenido al chat, ¿cuál es tu nombre?
Usuario [>]: Gabriel
Chatbot [>]: Gabriel cuéntame, ¿a dónde quieres viajar? Recuerda que por el momento sólo ofrecemos viaje
s a capitales regionales del país.
Gabriel [>]: me gustaría viajar a la ciudad de punta arenas
Chatbot [>]: Punta Arenas es un lugar ideal en ésta época del año!. Te cuento que los pasajes tienen un
valor de $15.000 pesos ¿Deseas confirmar esos pasajes?
Gabriel [>]: si
Chatbot [>]: Bien, ahora para confirmar la cantidad y la fecha de los pasajes, debe ingresar a nuestro s
itio web
Gabriel [>]: !saveLog
Sistema [!]: Archivo log generado satisfactoriamente.
Gabriel [>]:
```

Figura 13 Ejemplo de !saveLog

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. Nótese que además de mantener el mensaje y el remitente, también se almacena cada uno de estos con su respectiva marca de tiempo, tanto para saber la fecha en que se tuvo una conversación en específico, como la hora en que el mensaje fue enviado. La siguiente figura permite observar el resultado generado dentro del archivo.

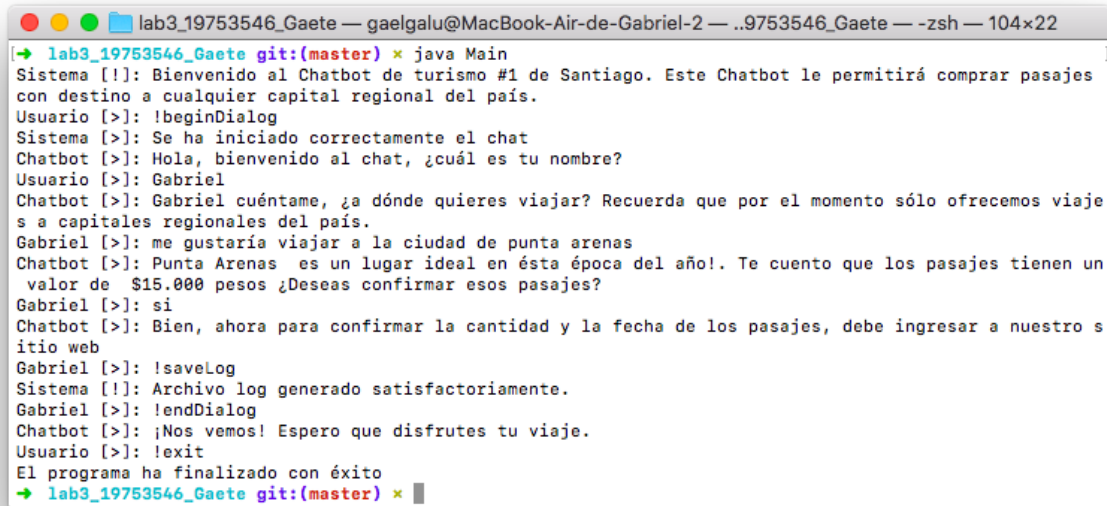


```
06-08-2018_17-12-57.log
[06/08/2018 17:05:56] Usuario: !beginDialog
[06/08/2018 17:05:56] Chatbot: Hola, bienvenido al chat, ¿cuál es tu nombre?
[06/08/2018 17:05:59] Usuario: Gabriel
[06/08/2018 17:05:59] Chatbot: Gabriel cuéntame, ¿a dónde quieres viajar? Recuerda que por el momento sólo ofrecemos
viajes a capitales regionales del país.
[06/08/2018 17:06:24] Gabriel: me gustaría viajar a la ciudad de punta arenas
[06/08/2018 17:06:24] Chatbot: Punta Arenas es un lugar ideal en ésta época del año!. Te cuento que los pasajes tienen
un valor de $15.000 pesos ¿Deseas confirmar esos pasajes?
[06/08/2018 17:06:26] Gabriel: si
[06/08/2018 17:06:26] Chatbot: Bien, ahora para confirmar la cantidad y la fecha de los pasajes, debe ingresar a
nuestro sitio web
[06/08/2018 17:12:57] Gabriel: !saveLog
```

Figura 14 Registro dentro del log

Por último, para terminar la conversación con el chatbot, es necesario utilizar la instrucción ***!endDialog***, sin embargo, el programa no se finaliza, pensando en que tal vez otro usuario (o el mismo si desea), pueda iniciar una nueva conversación con las instrucciones adecuadas.

Si se desea salir del programa, es decir, finalizar por completo la ejecución, se ha añadido una nueva instrucción, denominada ***!exit***, la cual permite esta acción. La siguiente figura muestra el resultado tanto de ***!endDialog*** como de ***!exit***.

A screenshot of a terminal window titled 'lab3_19753546_Gaete' with a subtitle 'gaelgalu@MacBook-Air-de-Gabriel-2 — .9753546_Gaete — zsh — 104x22'. The terminal shows a Java application running 'Main'. The chatbot's welcome message is: 'Bienvenido al Chatbot de turismo #1 de Santiago. Este Chatbot le permitirá comprar pasajes con destino a cualquier capital regional del país.' The user enters '!beginDialog'. The chatbot responds: 'Hola, bienvenido al chat, ¿cuál es tu nombre?'. The user enters 'Gabriel'. The chatbot asks: 'Gabriel cuéntame, ¿a dónde quieres viajar? Recuerda que por el momento sólo ofrecemos viajes a capitales regionales del país.' The user enters 'me gustaría viajar a la ciudad de punta arenas'. The chatbot responds: 'Punta Arenas es un lugar ideal en ésta época del año!. Te cuento que los pasajes tienen un valor de \$15.000 pesos ¿Deseas confirmar esos pasajes?'. The user enters 'si'. The chatbot asks: 'Bien, ahora para confirmar la cantidad y la fecha de los pasajes, debe ingresar a nuestro sitio web'. The user enters '!saveLog'. The chatbot responds: 'Archivo log generado satisfactoriamente.' The user enters '!endDialog'. The chatbot responds: '¡Nos vemos! Espero que disfrutes tu viaje.' The user enters '!exit'. The terminal then shows 'El programa ha finalizado con éxito' and returns to the shell prompt.

```
lab3_19753546_Gaete git:(master) ✕ java Main
Sistema [!]: Bienvenido al Chatbot de turismo #1 de Santiago. Este Chatbot le permitirá comprar pasajes
con destino a cualquier capital regional del país.
Usuario [>]: !beginDialog
Sistema [>]: Se ha iniciado correctamente el chat
Chatbot [>]: Hola, bienvenido al chat, ¿cuál es tu nombre?
Usuario [>]: Gabriel
Chatbot [>]: Gabriel cuéntame, ¿a dónde quieres viajar? Recuerda que por el momento sólo ofrecemos viaje
s a capitales regionales del país.
Gabriel [>]: me gustaría viajar a la ciudad de punta arenas
Chatbot [>]: Punta Arenas es un lugar ideal en ésta época del año!. Te cuento que los pasajes tienen un
valor de $15.000 pesos ¿Deseas confirmar esos pasajes?
Gabriel [>]: si
Chatbot [>]: Bien, ahora para confirmar la cantidad y la fecha de los pasajes, debe ingresar a nuestro s
itio web
Gabriel [>]: !saveLog
Sistema [!]: Archivo log generado satisfactoriamente.
Gabriel [>]: !endDialog
Chatbot [>]: ¡Nos vemos! Espero que disfrutes tu viaje.
Usuario [>]: !exit
El programa ha finalizado con éxito
lab3_19753546_Gaete git:(master) ✕
```

Figura 15 Ejemplo de ***!endDialog*** y ***!exit***

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 (funcional y lógico), 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.

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 (**loadLog**), debido a la forma en que la clase fue estructurada, se ha dificultado el proceso, siendo imposible concretar 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/>