Universidad Autónoma del Estado de México
Centro Universitario UAEM Zumpango

Ingeniería en computación

Desarrollo de un punto de venta en Java

TESIS

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÓN

PRESENTA:

Daniel Hernández Velasco

ASESOR:

Dr. Asdrúbal López Chau

Zumpango, Estado de México, Noviembre 2019

Resumen

Actualmente, las pequeñas, medianas y grandes empresas tienen la necesidad de llevar una contabilidad minuciosa, así como de un mayor control sobre sus procesos internos. Esto sucede también para el establecimiento de comida rápida "Pizzas Davos". En el presente trabajo se propone una solución tecnológica para llevar el control de ventas del establecimiento antes mencionado, ya que previo al desarrollo de la propuesta, este control se realizaba de manera manual, generando problemas en el seguimiento de sus clientes y de tipo tributario.

El proyecto es un sistema tipo punto de venta, desarrollado en el lenguaje de programación Java, que mantiene conexiones a un servidor de bases de datos local (MySQL), el cual es administrado por el sistema de gestión de servicios XAMPP. Todas las tecnologías utilizadas para el uso y desarrollo del proyecto son de código abierto, y permiten ejecutar el sistema en cualquier plataforma realizando las configuraciones pertinentes.

De igual manera se presentan diagramas de clase y de entidad relación, descripciones detalladas de cada sección del sistema, las interfaces gráficas de usuario con explicación de los componentes que las integran, y pruebas de funcionamiento del sistema.

Abstract

Currently, small, medium and large companies have the need to keep a thorough accounting, as well as greater control over their internal processes. This also happens for the fast food establishment "Pizzas Davos". In the present work, a technological solution is proposed to control the sales of the aforementioned establishment, since prior to the development of the proposal, this control was carried out manually, generating problems in the follow-up of its clients and of a tax type.

The project is a point-of-sale system, developed in the Java programming language, which maintains connections to a local database server (MySQL), that is managed by the XAMPP service management system. All the technologies used for the use and development of the project are open source, and allow the system to be run on any platform, making the relevant configurations.

In the same way, class and relationship diagrams are presented, detailed descriptions of each section of the system, graphical user interfaces with explanation of the components that integrate them, and system performance tests.

Dedicatoria

El presente documento está dedicado a mis padres por siempre brindarme su apoyo y ayuda incondicional, así como a todas aquellas personas que apostaban a que mi vida sería el fracaso y la decepción familiar.

Agradecimientos

Quiero expresar mi agradecimiento a la Universidad Autónoma del Estado de México y al Centro Universitario UAEM Zumpango, por abrirme sus puertas y formarme como ingeniero.
A mi asesor, Dr. Asdrúbal López Chau por sus constantes recordatorios sobre terminar la tesis, su guía y sus consejos.
A mis revisores, M. en C. Manuel Almeida Vázquez y M. en C. Edith Cristina Herrera Luna, por las observaciones y sugerencias que enriquecieron considerablemente el presente documento.
A todos mis profesores, por todos los conocimientos y apoyo compartido a lo largo de mi formación y estancia académica.
A mis amigos por siempre permanecer unidos y llegar hasta donde nos encontramos el día de hoy.

Contenido

Lista de figuras

Lista de tablas

Capítulo 1

INTRODUCCIÓN

Actualmente, las pequeñas, medianas y grandes empresas tienen la necesidad de llevar una contabilidad minuciosa, así como de un mayor control en los procesos internos. Este caso aplica de igual manera para la pequeña empresa Pizzas Davos, que se encuentra localizada en el municipio de Nextlalpan, Estado de México. Es por lo anterior, que la microempresa Pizzas Davos se encuentra en la necesidad de incorporar en su plan de trabajo un sistema que permita una mayor eficiencia en la administración y flujo de procesos.

1.1 Planteamiento del problema

Pizzas Davos lleva actualmente sus procesos administrativos y de contabilidad de una manera manual y desordenada, es decir, sus empleados anotan en una libreta las ventas, pedidos y compra de insumos. Esto genera problemas de seguimiento de clientes y de tipo tributario.

La problemática actual de Pizzas Davos no es exclusiva, ya se ha detectado que otros negocios cercanos también tienen situaciones similares en cuanto a aspectos administrativos y de contabilidad. En esta tesis se resuelve parte de la problemática mencionada anteriormente, por lo que se propone la gestión semiautomática de los procesos de venta y de clientes. Desde el punto de vista técnico, la problemática

a atender es la obtención de requerimientos del cliente (Pizzas Davos), el diseño de un sistema informático personalizado que satisfaga esos requerimientos, así como la implementación para lograr un sistema funcional para el cliente.

1.2 Objetivos

1.2.1 Objetivo general

Diseñar e implementar en el lenguaje de programación Java un sistema informático personalizado para realizar las funciones básicas de punto de venta para un negocio de pizzas.

1.2.2 Objetivos específicos

Los objetivos específicos de esta tesis son los siguientes:

1. Realizar una comparativa sobre los programas comerciales para punto de venta.
2. Obtener los requerimientos del negocio para el cual se implementa el sistema.
3. Diseñar el sistema informático de punto de venta con un enfoque orientado a objetos.
4. Realizar la implementación en lenguaje Java de acuerdo al diseño generado en el objetivo anterior.
5. Realizar pruebas de funcionamiento al sistema para verificar que cumple con los requisitos de diseño.

1.3 Hipótesis

Es factible el desarrollo de un sistema informático tipo punto de venta para el negocio de comida rápida Pizzas Davos, ubicado en el municipio de San Francisco Molonco,

Nextlalpan en el Estado de México. El sistema a desarrollar puede automatizar las tareas de ventas, logística y administración de usuarios.

1.4 Justificación

Debido a las problemáticas previamente planteadas, es de vital importancia para la empresa llevar a cabo una optimización en los procesos administrativos, con el fin de proporcionar un mejor servicio a los clientes que acuden a la micro empresa. De igual manera, con la implementación del sistema es posible lograr que los procesos internos administrativos sean más eficientes y simples.

1.5 Alcance del proyecto

El proyecto presente puede realizar los siguientes procesos:

- Administrar información de clientes usando una base de datos relacional.
- Administrar información de productos mediante una interfaz gráfica
- Generar ventas usando una interfaz gráfica.
- Proporcionar un recibo de compra para su impresión.

Capítulo 2

Tecnologías relacionadas

2.1 Lenguaje de Programación Java

Historia

En 1991 Sun Microsystems patrocinó un proyecto interno llamado Green, en el cual se desarrollaba un nuevo lenguaje de programación basado en C++ a cargo de James Gosling, quien lo llamó Oak (roble en español). Esto debido a un roble que tenía a la vista desde su ventana en las oficinas de Sun, pero posteriormente se descubre que ya existía un lenguaje con el mismo nombre. Cuando Gosling junto con su equipo visitan una cafetería local, sugieren asignarle el nombre de Java, que es una variedad de café [6]. De este modo es como se le dio nombre a Java, un lenguaje de programación orientado a objetos ampliamente usado en todo el mundo, y en la mayoría de los dispositivos móviles, los que tienen instalado SO Android.

2.1.1 Generalidades

Los programas en Java constan de diversas piezas, las cuales son llamadas clases, que a su vez contienen métodos y atributos, que en conjunto realizan tareas y pueden devolver información . Java cuenta con una amplia variedad de estas piezas que realizan ciertas tareas, y proporcionan facilidades al programador para el desarrollo de sistemas

software, además, el programador puede crear sus propias piezas de software y así ampliar más las bibliotecas de Java. Java cuenta con un entorno de ejecución, logrando que los programas realizados en dicho lenguaje puedan ser ejecutados en cualquier sistema operativo, con lo anterior, deducimos que Java es portable. Esta portabilidad se logra gracias a un componente llamado JVM (Java Virtual Machine, o máquina virtual de Java), que se instala en el sistema operativo anfitrión y que permite la ejecución de código creado para la JVM.

Existe una controversia desde hace varios años sobre si Java es –además de un lenguaje de programación– un entorno de ejecución, una plataforma o ambas. De acuerdo al tutorial oficial de la empresa Oracle (https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html), "La tecnología Java es ambos un lenguaje de programación y una plataforma". Varios desarrolladores alrededor del mundo han debatido sobre esto (https://stackoverflow.com/questions/19817793/why-we-are-calling-java-is-a-programming-language-and-also-platform), argumentando que Java es un lenguaje de programación, y que la plataforma está conformada por la JVM, es decir, que son cosas diferentes. Para propósitos prácticos de este trabajo, Java es un lenguaje de programación a partir del cual se puede generar un código máquina (bytecodes) capaz de ejecutarse en una plataforma independiente del hardware (JVM).

2.1.2 Fases de un programa en Java

Usualmente los programas realizados en Java pasan por cinco fases:

1. **Fase 1:** Creación del Programa (Edición). Consiste en realizar la edición de un archivo con extensión. Java, que contiene el código fuente. Esto generalmente se realiza utilizando un editor de texto, o bien, un entorno de desarrollo integrado (IDE). Algunos IDES para Java son Eclipse y Netbeans, siendo estos los más utilizados [9], [3].

2. **Fase 2:** Compilación del Programa. En esta fase, el programador, hace uso del compilador de Java, proporcionado por Oracle, en el cual un archivo con la extensión .java es analizado por dicho compilador para detectar los posibles errores que el programador no corrigió o no vió, repitiendo este proceso hasta que el código fuente esté correctamente creado, posteriormente, con el código limpio de errores, el compilador lo traduce en códigos de bytes (bytecodes) los cuales son usados en la fase de ejecución. El resultado de la compilación son archivos ".class" identificados con el mismo nombre que el archivo .java compilado. Opcionalmente, se pueden empaquetar los archivos .class en un archivo único ejecutable, que tiene extensión .jar

3. **Fase 3:** Carga del Programa. En este punto la JVM toma los archivos .class o .jar para cargar el programa en la memoria principal antes de su ejecución, es decir, obtiene las instrucciones en bytes que almacenan los .class y los carga en la RAM. Esto se realiza de manera independiente al sistema operativo.

4. **Fase 4:** Verificación del Programa. La JVM analiza los códigos de bytes para confirmar que son válidos y no violen las restricciones de seguridad que Java ha implementado. Cabe mencionar que dichas restricciones ayudan para asegurar que los programas que llegan a través de la red no dañen nuestros archivos y/o el sistema.

5. **Fase 5:** Ejecución del Programa. En ésta última fase la JVM ejecuta cada uno de los códigos de bytes que ya se encuentran cargados en la memoria principal, para que los programas creados en Java funcionen correctamente, analizando los códigos de bytes a medida que se van interpretando.

2.1.3 Estructura general de un programa Java

Generalmente todos los programas realizados en Java (el código fuente) constan de una estructura general, que permite analizar y comprender la funcionalidad de los segmentos

de código realizado. En la Figura 2.1, se presenta un programa Java que imprime un mensaje en la pantalla. Los elementos de este programa son los siguientes:

```
1   // Paquete al que pertenece el archivo
2   package ejemplo;
3   /*
4       Definiciones de clases, la principal, debe
5       de llamarse exactamente igual que el archivo
6   */
7   public class Main{
8       public static void main(String[] args){
9           System.out.println("Tesis Daniel Hernández Velasco");
10      }
11  }
```

Figura 2.1: Programa que imprime texto en pantalla

1. package o paquete, que hace referencia a la carpeta o directorio en la cual se encuentra alojado(s) uno o más archivos Java. Esto ayuda a agrupar los archivos y para evitar que existan duplicados de sí mismos dentro de el mismo directorio.

2. Definición de la clase principal, la cual debe de llamarse exactamente igual que el archivo y debe poseer un modificador de acceso.

3. Método main o método principal, el cual funciona como punto de entrada a nuestro programa, gracias a este método "especial" es como comienza la ejecución de un programa Java.

2.1.4 Variables en Java

Tipos primitivos de datos en Java

El lenguaje de Java cuenta con ocho tipos de datos primitivos, los cuales se enlistan y describen a continuación:

1. **boolean:** Este tipo de dato sólo puede tomar dos valores, true(verdadero) o false (falso) y cuenta con un espacio en memoria de 1 byte.

2. **char:** Dispone de 2 bytes para almacenar exclusivamente un caracter.

3. **byte:** Permite almacenar números enteros con un rango de -128 a 127, contando con un espacio en memoria de 1 byte.

4. **short:** Este tipo de dato guarda números enteros de -32768 a 2147483647, contando con un espacio en memoria de 2 bytes.

5. **int:** Almacena números enteros en un rango de -2,147,483,648 a 83,647, contando con un espacio en memoria de 4 bytes.

6. **long:** Es capaz de almacenar números enteros entre -9,223,372,036,854,775,808 y 9,223,372,036,854,775,807, contando con un espacio en memoria de 4 bytes.

7. **float:** Este tipo de dato es capaz de almacenar números reales, es decir, números de punto flotante que comprende de -3.402823×10^{38} a -1.401298×10^{-45} y de 1.401298×10^{-45} a 3.402823×10^{38}, contando con un espacio en memoria de 4 bytes.

8. **double:** Tipo de dato que puede almacenar números reales de $-1.79769313486232\times10^{38}$ a $-4.94065645841247\times10^{-324}$ y de $4.94065645841247\times10^{-324}$ a $1.79769313486232\times10^{308}$, contando con un espacio en memoria de 8 bytes.

Declaración e inicialización de variables en Java

La declaración, así como la inicialización de variables en Java es relativamente sencilla. Para la declaración de una variable, basta con definir el tipo de dato que manejará la variable, seguido de un identificador terminando con punto y coma. Las variables también pueden ser referencias a objetos de clases pertenecientes a la API de Java o a clases creadas por el programador [10] [13].

```java
// Paquete al que pertenece el archivo
package calculadora;
/*
    Definiciones de clases, la principal, debe
    de llamarse exactamente igual que el archivo
*/
public class Main{
    //Método main
    public static void main(String[] args){
        /* Declaración de tres variables llamadas "x","y" y "z"
           cáda una de ellas de tipo entero
           Sintaxis: tipoDato identificador;*/
        int x;
        int y;
        int z;
        /* Inicialización de variables, asignando el valor 1,2,3 a
           x,y,z respectivamente.
           Sintaxis: identificador = valorAsignar; */
        x=1;
        y=2;
        z=3;
        // Impresión de los valores de las variables en pantalla
        System.out.println("Valor de x: "+x);
        System.out.println("Valor de y: "+y);
        System.out.println("Valor de z: "+z);
    }
}
```

Figura 2.2: Programa para declaración y asignación de variables

En el caso de la inicialización, se comienza con el nombre de la variable (identificador) seguido del signo igual (=) y posteriormente, el valor que se le desea asignar a dicha variable cabe mencionar que ese valor, debe ser acorde al tipo de dato que se está manejando, en la Figura 2.1 se muestra un ejemplo de lo anterior. Cuando se declaran variables es necesario decidir el tipo de dato a usar y el nombre de la variable [16] [2].

En la Figura 2.3 se ejemplifica otra manera de declarar e inicializar variables, la cual es básicamente la combinación de las dos formas antes mencionadas. Como se puede observar en las líneas 13 a 21 de la 2.3 es posible declarar e inicializar las variables al mismo tiempo, en la línea 22 se puede observar que también es posible declarar un conjunto de variables en una sola línea de código, pero todas las variables deben de ser estrictamente del mismo tipo de dato y posteriormente deben de ser inicializadas una a una.

```java
// Paquete al que pertenece el archivo
package calculadora;
/*
    Definiciones de clases, la principal, debe
    de llamarse exactamente igual que el archivo
*/
public class Main{
    //Método main
    public static void main(String[] args){
        /* Declaración de tres variables llamadas "x","y" y "z"
           cáda una de ellas de tipo entero
           Sintaxis: tipoDato identificador;*/
        int x;
        int y;
        int z;
        /* Inicialización de variables, asignando el valor 1,2,3 a
           x,y,z respectivamente.
           Sintaxis: identificador = valorAsignar; */
        x=1;
        y=2;
        z=3;
        float a,b,c;
        a=1.1;
        b=1.2;
        c=1.3;
        // Impresión de los valores de las variables en pantalla
        System.out.printf("Valor de x: %d\nValor de y: %d\nValor de z: %d",x,y,z);
        System.out.printf("Valor de a: %f\nValor de b: %f\nValor de c: %d",a,b,c);
    }
}
```

Figura 2.3: Programa para declaración e inicialización de variables

2.1.5 Operadores en Java

En Java es posible hacer operaciones con variables (en su mayoría del mismo tipo de dato) y para hacerlo se cuenta con diversos operadores, los cuales se organizan en las siguientes categorías: aritméticos, asignación, lógicos, relacionales, operadores de incremento o decremento y operadores a nivel de bits [15] [10].

Aritméticos

Estos operadores realizan procesos básicos como la suma, resta multiplicación y división. En la tabla 2.1 se ejemplifican y describen cada uno.

Asignación

Como su nombre lo indica, estos operadores permiten conferir valores a las variables, en la tabla 2.2 se ejemplifican y describen cada uno de ellos.

Operador	Uso	Descripción
+	var1+var2	Suma las variables “var1” y “var2”
-	var1-var2	Resta las variables “var1” y “var2”
*	var1*var2	Multiplica las variables “var1” y “var2”
/	var1/var2	Divide la variable “var1” entre la variable “var2”
%	var1%var2	Obtiene el residuo al dividir la variable “var2” entre la variable “var2”

Tabla 2.1: Operadores Aritméticos

Operador	Uso y sentencia equivalente	Descripción
=	var1 = var2	Se asigna el valor de la variable “var2” a la variable “var1”
+=	var1 += var2 var1 = var1 + var2	Se asigna el valor resultante de la suma entre el valor actual de “var1” y el valor de “var2” y se asigna a la variable “var1”
*=	var1 *= var2 var1 = var1 * var2	Se asigna el valor resultante de la multiplicación entre el valor actual de “var1” y el valor de “var2” y se asigna a la variable “var1”
/=	var1 /= var2 var1 = var1 / var2	Se asigna el valor resultante de la división del valor actual de “var1” entre el valor de “var2” y se asigna a la variable “var1”
%=	var1 %= var2 var1 = var1 % var2	Se asigna el residuo de la división del valor actual de “var1” entre el valor de “var2” y se asigna a la variable “var1”

Tabla 2.2: Operadores de asignación

Incremento y decremento

El operador de incremento (++) aumenta el valor de su operando en una unidad, y el operador decremento (–) lo disminuye en una unidad [17]. En la tabla 2.3 se ejemplifican cada uno de ellos.

Operador	Uso

++	i=1;
	i++;
	El valor de i es 2

–	i=1; i–; El valor de i es 0

Tabla 2.3: Operadores de incremento y decremento

Existe una variante para este tipo de operadores, en el cual el operador precede al operando, en la tabla 2.4 se describe su funcionamiento.

Funcionamiento de operadores de incremento y decremento como prefijo o sufijo	
Prefijo	Sufijo
El operador se encuentra antes del operando,en este caso, primero se incrementa o decrementa el valor del operando y posteriormente se utiliza.	El operador se encuentra después del operando, en este caso, primero se usa el valor del operando y posteriormente se incrementa o decrementa su valor.

Tabla 2.4: Funcionalidad de operadores de incremento y decremento

Operadores Relacionales

Los operadores relacionales son útiles y parte esencial en las sentencias de control, tanto las selectivas como las iterativas, (IF, FOR, WHILE) ya que ayudan al programa en la toma de decisiones. Estos operadores son capaces de hacer comparaciones entre operandos (variables) y en todos los casos devuelve un valor booleano (true o false). En la tabla 2.5 se enlistan y describen cada uno de dichos operadores.

Operador	Uso	Descripción
<	var1 <var2	Compara si el valor de la variable "var1" es menor que el valor de la variable "var2", sí lo es, devuelve un true, de lo contrario devuelve false
>	var1 >var2	Compara si el valor de la variable "var1" es mayor que el valor de la variable "var2", sí lo es, devuelve un true, de lo contrario devuelve false
==	var1 == var2	Compara si el valor de la variable "var1" es igual que el valor de la variable "var2", sí lo es, devuelve un true, de lo contrario devuelve false
<=	var1 <= var2	Compara si el valor de la variable "var1" es menor o igual que el valor de la variable "var2", sí lo es, devuelve un true, de lo contrario devuelve false
>=	var1 >= var2	Compara si el valor de la variable "var1" es mayor o igual que el valor de la variable "var2", sí lo es, devuelve un true, de lo contrario devuelve false
!=	var1 != var2	Compara si el valor de la variable "var1" es diferente que el valor de la variable "var2", sí lo es, devuelve un true, de lo contrario devuelve false

Tabla 2.5: Operadores relacionales

2.1.6 Clases

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos [14]. Una clase es una forma de abstraer objetos o situaciones de la vida real.

Declaración

Para declarar una clase, es necesario introducir la palabra reservada class seguido de un identificador que da nombre a la clase, por buenas prácticas de programación, dicho identificador debe comenzar con mayúscula. Después del identificador, procede el cuerpo de la clase que contiene las variables y métodos miembro de la clase las cuales están contenidas y delimitadas entre llaves, en la Figura 2.4 se describe con código la declaración de una clase.

```
1  class Main{
2      /*
3          Cuerpo de la clase donde
4          incluye constructores, variables
5          y métodos
6      */
7  }
```

Figura 2.4: Declaración de una clase en Java

Niveles de acceso

Los niveles de acceso son palabras reservadas (Modificadores de acceso) que preceden a las declaraciones de clases, métodos y variables. Dichas palabras permiten crear un cierto nivel de restricción sobre los componentes de la clase, es decir, permite a los componentes, ser o no visibles por otras clases. Java cuenta con cuatro niveles de acceso: public, private, default y protected (público, privado, predeterminado y protegido). El uso es estos modificadores de acceso es opcional y puede omitirse, sí el nivel de acceso es omitido entonces se dice que la declaración tiene una accesibilidad por defecto (default accesibility), lo que significa que es accesible por cualquier otra clase dentro del mismo paquete [24] [1] [30]. En la Figura 2.5 se muestra un ejemplo del uso de estos modificadores de acceso.

1. **public:** Todas las declaraciones a las que le preceda son accesibles desde cualquier clase, aunque esta última no se encuentre en el mismo paquete.

2. **protected:** Todas las declaraciones a las que le preceda son accesibles para cualquier clase y/o subclase que se encuentren en el mismo paquete.

3. **private:** Todas las declaraciones a las que le preceda son accesibles exclusivamente dentro de la clase y ninguna clase externa puede acceder a ellas.

4. **default:** Todas las declaraciones que no posean un modificador de acceso, Java le asigna el default por lo que son visibles para todas las demas clases que se encuentren en el mismo paquete.

```
1   public class Main{
2       public int edad;
3       private String path;
4       public void metodo1(){
5           //código
6       }
7       public void metodo2(){
8           // Código
9       }
10  }
```

Figura 2.5: Modificadores de acceso

2.1.7 Miembros de clase

Variables miembro de objeto

Al crear objetos cada uno tiene sus propios datos o variables, es decir, cada objeto posee sus propios valores que son independientes de otros objetos que sean creados a partir de la misma clase. En la Figura 2.6 se muestra un ejemplo de las variables miembro de objeto, así como la salida del programa respectivamente.

```java
public class Persona {
    // Variables Objeto
    private String nombre, apPat,apMat;
    // Constructor clase Persona
    public Persona(String nombre, String apPat,String apMat){
        this.nombre = nombre;
        this.apPat=apPat;
        this.apMat=apMat;
    }
    // Métodos get para acceder a las variables objeto
    public String getNombre(){
        return nombre;
    }
    public String getApPat(){
        return apPat;
    }
    public String getApMat(){
        return apMat;
    }
    // Método main
    public static void main(String[] args){
        // Declaración de objetos de la clase persona
        Persona per1 = new Persona("Daniel","Hernández","Velasco");
        Persona per2 = new Persona("Brenda","Mendoza","Meza");
        // Impresión en pantalla de las variables objeto de cada instancia creada
        System.out.println("Objeto Persona 1: ");
        System.out.printf("\t%s %s %s.\n",per1.getNombre(),per1.getApPat(),per1.getApMat());
        System.out.println("Objeto Persona 2: ");
        System.out.printf("\t%s %s %s.\n",per2.getNombre(),per2.getApPat(),per1.getApMat());
    }
}
```

(a) Código

```
C:\Users\Danny\Desktop>javac Persona.java

C:\Users\Danny\Desktop>java Persona

Objeto Persona1:
        Daniel Hernandez Velasco

Objeto Persona2:
        Brenda Mendoza Meza

C:\Users\Danny\Desktop>_
```

(b) Ejecución de código

Figura 2.6: Código Variables objeto

Variables miembro de clase

Este tipo de variables, a diferencia de las variables miembro de objeto, son precedidas por la palabra reservada static y son inicializadas con valores por defecto de cualquier tipo de dato. Los valores de este tipo de variables son compartidos por todos los objetos creados a partir de la misma clase, por lo tanto, si son modificadas los cambios se aplican para todos los objetos creados. Una característica peculiar de este tipo de variables es que también se puede acceder a ellas haciendo referencia con el nombre de la clase sin la necesidad de crear un objeto para hacerlo, debido a que, al ser variables de una clase, se

comparten para todos los objetos creados a partir de ella. En la Figura 2.7 se muestra un ejemplo de las variables miembro de clase y la salida del programa respectivamente.

```java
public class Persona{
    private static int edad=15; // Variable miembro de clase
    private String nombre; // Variable miembro de objeto
    // Constructor
    public Persona(String nom){
        nombre=nom;
    }
    // Método para obtener la variable miembro de objeto
    public String getNombre(){
        return nombre;
    }
    // Método main
    public static void main(String[] args){
        // Declaración de objetos de la clase persona
        Persona p1 = new Persona("Daniel");
        Persona p2 = new Persona("Brenda");
        // Acceso de la variable miembro de objeto y miembro de clase
        System.out.println("Persona 1: ");
        System.out.printf("\t%s %d",p1.getNombre(),Persona.edad);
        System.out.println("Persona 2: ");
        System.out.printf("\t%s %d",p2.getNombre(),Persona.edad);
        // Modificción de la variable de clase
        Persona.edad =23;
        // Acceso de la variable miembro de objeto y miembro de clase
        System.out.println("Persona 1: ");
        System.out.printf("\t%s %d",p1.getNombre(),Persona.edad);
        System.out.println("Persona 2: ");
        System.out.printf("\t%s %d",p2.getNombre(),Persona.edad);
    }
}
```

(a) Código

```
C:\Users\Danny\Desktop>javac Persona.java

C:\Users\Danny\Desktop>java Persona
Persona 1:
        Daniel 15
Persona 2:
        Brenda 15

Persona 1:
        Daniel 20
Persona 2:
        Brenda 20
C:\Users\Danny\Desktop>
```

(b) Ejecución de código

Figura 2.7: Código variables miembro de clase.

Variables final

Son variables declaradas dentro de la clase, las cuales no pueden cambiar su valor durante toda la ejecución del programa. A este tipo de variables le precede la palabra reservada final. Las variables miembros de objeto y miembro de clase pueden ser declaradas como final [14].

Métodos de objeto

Los métodos son bloques de código definidos dentro de una clase y poseen un comportamiento en específico [14]. Este tipo de métodos sólo son accesibles a través de un objeto haciendo uso del operador punto (.).

Métodos de clase

A diferencia de los métodos de objeto, no es necesario hacer uso de una instancia para poder acceder a este tipo de métodos, ya que sólo basta con hacer referencia con el nombre de la clase.
En la Figura 2.8 se ejemplifica el uso de métodos de objeto y clase, así como la salida del programa.

```java
public class Calculadora{
    // Método de clase para obtener la suma de dos números
    public static double suma(double a, double b){
        return a+b;
    }
    /* Método de clase para obtener la multiplicación
       de dos números */
    public double multiplicacion(double a, double b){
       return a*b;
    }
    // Método de objeto para obtener la resta de dos números
    public double resta(double a, double b){
        return a-b;
    }
    // Método de objeto para obtener el módulo de dos numeros
    public double modulo(double a, double b){
        return a%b;
    }
    public static void main(String[] args){
        // Objeto de la clase calculadora
        Calculadora calc = new Calculadora();
        // Acceso a métodos de clase
        double suma = Calculadora.suma(2,3);
        double mult = Calculadora.multiplicacion(2,3);
        // Acceso a métodos de objeto
        double resta = calc.reta(2,3);
        double modulo = calc.modulo(2,3);
        System.out.printf("\n Suma: %f\n Multiplicación: %f\n",suma,mult);
        System.out.printf(" Resta: %f\n Modulo: %f\n",resta,modulo);
    }
```

(a) Código

```
C:\Users\Danny\Desktop>java Calculadora

 Suma: 5.000000
 Multiplicacion: 6.000000
 Resta: -1.000000
 Modulo: 2.000000

C:\Users\Danny\Desktop>
```

(b) Ejecución de código

Figura 2.8: Métodos miembro de objeto y de clase

2.1.8 Sobrecarga de métodos

Históricamente, en los programas de computadoras, los nombres de los métodos eran únicos, de este modo el compilador podría identificar qué método ha sido invocado con sólo revisar su nombre [20]. En Java es posible definir dos o más métodos dentro de una misma clase y que comparten el mismo nombre, siempre y cuando la declaración de sus parámetros sean diferentes [26].

A esta situación, se le conoce como sobrecarga de métodos. Cuando un método sobrecargado es invocado, Java usa el tipo de dato o el número de parámetros para discernir qué método sobrecargado es el que se ha mandado a llamar, dicho de otra

forma, Java sólo distingue métodos sobrecargados a través del número o tipos de argumentos; aunque es posible pensar que los métodos sobrecargados podrían tener diferentes tipos de retorno, éste último es insuficiente para poder diferenciarlos [7].

2.1.9 Herencia

Antes de abordar el tema de sobreescritura de métodos, es necesario comprender un poco sobre la herencia y polimorfismo.

La herencia es una de las bases de la programación orientada a objetos, ya que permite clasificaciones jerárquicas [26] [11]. Es una forma de reutilización de software en la que se crea una nueva clase absorbiendo los miembros de una clase existente, y se mejoran con nuevas capacidades, o con modificaciones en las capacidades ya existente [6]. Haciendo uso de ésta potente característica, es posible crear una clase general que nos permite abstraer aspectos en común de distintas clases. En la Figura 2.9 se ejemplifica un caso de herencia.

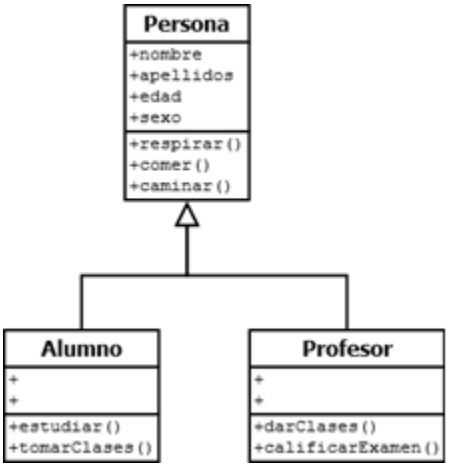


Figura 2.9: Diagrama de clases Herencia

En la figura anterior, contamos con tres clases, una superclase (Clase padre) y dos subclases (Clases hijas), siendo Persona la clase padre y, Alumno y Profesor las clases hijas. La clase persona es la generalización de un ser humano, tanto un alumno como un profesor, son personas; por lo tanto, tienen aspectos en común, como un nombre, apellidos, edad y sexo, además, todas ellas realizan acciones como caminar, comer y respirar.

Todos estos aspectos los tiene cualquier persona, por lo que son abstraídos en una sola representación de ellas, la clase Persona. Posteriormente las clases hijas están asociadas por una flecha ascendente a la clase padre, esto nos indica que dichas clases extienden de Persona. De este modo las clases hijas heredan las características y acciones de una persona.

En código, una clase hija se le puede reconocer debido a que la forma de declararse tiene una variación. En su declaración, después del identificador de la clase, se encuentra la palabra reservada "extends" y posteriormente el identificador de la clase padre. En la Figura 2.10 se muestra el ejemplo anterior en código Java. Se pueden observar las tres clases del ejemplo y una clase principal, tal como en el diagrama de clases (Figura 2.9), las clases hijas no tienen atributos; debido a que, al extender de la clase padre, todos los atributos y métodos de esta le son heredados y no es necesario volver a declarar dichos atributos. Para acceder a los métodos y atributos heredados sólo basta con hacer referencia a ellos como si hubiesen sido declarados dentro de la misma clase hija.

Por lo anterior se demuestra el funcionamiento de esta potente característica, cabe mencionar que en Java no existe la herencia múltiple, es decir, una clase hija puede tener única y exclusivamente una sola clase padre.

```java
package Tesis;
public class Persona{
    protected String nombre;
    protected String apellidos;
    protected int edad;
    protected char sexo;
    protected void respirar(){
        System.out.println("Respirando");
    }
    protected void comer(){
        System.out.println("Comiendo");
    }
    protected void caminar(){
        System.out.println("Caminando");
    }
}
```

(a) Clase Persona

```java
package Tesis;
public class Profesor extends Persona{
    protected void impartirClases(){
        System.out.println("Impartiendo clase");
    }
    protected void calificarExamen(){
        System.out.println("Impartiendo clase");
    }
}
```

(b) Clase Profesor

```java
package Tesis;
public class Alumno extends Persona{
    protected void estudiar(){
        System.out.println("Estudiando");
    }
    protected void tomarClases(){
        System.out.println("En clases");
    }
}
```

(c) Clase Alumno

```java
package Tesis;
public class Main{
    public static void main(String[] args){
        Alumno alu = new Alumno();
        // Atributos heredados de alumno
        alu.nombre = "Daniel";
        alu.apellidos = "Hernández Velasco";
        alu.sexo = 'M';
        System.out.println("Datos del alumno: ");
        System.out.printf("Nombre: %s\nApellidos: %s\nSexo: %c",
            alu.nombre,alu.apellidos,alu.sexo);
        // Métodos heredados de alumno
        alu.respirar();
        alu.comer();
        alu.caminar();
    }
}
```

(d) Clase Main

```
C:\Users\Daniel\Desktop\Código tesis>java Tesis/Main
Datos del alumno:
Nombre: Daniel
Apellidos: Hernandez Velasco
Sexo: H
Respirando
Comiendo
Caminando

C:\Users\Daniel\Desktop\Código tesis>
```

(e) Ejecución de código

Figura 2.10: Código de herencia

2.1.10 Polimorfismo

El polimorfismo nos permite "programar en forma general", en vez de "programar en forma específica". En especial, nos permite escribir programas que procesen objetos que compartan la misma superclase en una jerarquía de clases, como si todos fueran objetos de la superclase; esto puede simplificar la programación [6] [11].

En POO (Programación Orientada a Objetos), el polimorfismo permite que diferentes objetos respondan de modo distinto al mismo mensaje; adquiere su máxima potencia

cuando se utiliza en unión de la herencia [18]. A continuación, en la Figura 2.11 se muestra un ejemplo en código del polimorfismo.

```java
package Tesis;
public class Animal{
    public void dormir(){
        System.out.println("Animal Durmiendo");
    }
    public void vagar(){
        System.out.println("Animal Vagando");
    }
}
```

(a) Clase Animal

```java
package Tesis;
public class Lobo extends Animal{
    @Override
    public void vagar(){
        System.out.println("Los lobos vagan en manada");
    }
}
```

(b) Clase Lobo

```java
package Tesis;
public class Perro extends Animal{
    @Override
    public void vagar(){
        System.out.println("Los Perros vagan solos");
    }
}
```

(c) Clase Perro

```java
package Tesis;
public class Main{
    public static void main(String[] args){
        Animal animal =  new Perro();
        animal.vagar();
        animal = new Lobo();
        animal.vagar();
    }
}
```

(d) Clase Main

```
C:\Users\Cachubis\Desktop>java Tesis/Main
Los Perros vagan solos
Los lobos vagan en manada

C:\Users\Cachubis\Desktop>
```

(e) Ejecución de código

Figura 2.11: Código de polimorfismo

En el ejemplo de la figura anterior, haciendo uso de herencia; se cuenta con 4 clases, una clase llamada Animal (Clase padre), una llamada Perro (Clase hija), una más llamada Lobo (Clase hija), y por último una clase principal llamada Main.
Como es de esperarse, la clase animal es la abstracción de dos especies, el lobo y un perro con dos métodos en común, dormir y vagar. Todos los animales suelen dormir de la misma forma, sin embargo, cuando andan vagando en algún lugar suelen hacerlo de manera muy particular; a diferencia del perro que vaga solo, los lobos siempre vagan en manada. Es por lo anterior que las clases Lobo y Perro, sobrescriben el método vagar para hacerlo a su propia manera dando paso al polimorfismo. En la clase Main se observa que creamos un objeto que hace referencia a la clase Perro y se asigna a una variable de tipo Animal; esto quiere decir que nuestro objeto perro se va a comportar como un objeto de su clase padre, manteniendo su propia definición del método heredado y sobrescrito, para que cuando sea haga una llamada al método, este se haga de acuerdo

a su propia definición.

2.1.11 Interfaces gráficas de usuario con Java (GUI)

Una interfaz gráfica de usuario, tamien conocida como GUI (Graphical User Interface), es una capa de software que permite la comunicación de un usuario con un sistema a través de componentes gráficos, como lo pueden ser imágenes, botones, ventanas. Java cuenta con un conjunto de librerias que permiten crear aplicaciones de este tipo. Para el presente proyecto se ocupan las librerias Swing para los componentes gráficos y awt para el manejo de eventos. Tódos los componentes son realmente clases con variables y métodos que al pasar por el compilador crea los componentes a los que representan.

Componentes gráficos de la librería Swing

La librería Swing consta de diversos componentes, dentro de los componentes más comunes y que se encuentran frecuentemente en una interfaz de usuario se encuentran JFrame, JLabel, JButon, JTextField, JTextArea, JComboBox, mismas que a continuación se describen de manera general.

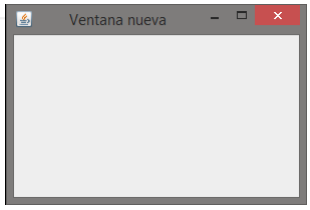
- **JFrame:** Esta clase representa una ventana, misma que a su ves puede contener distintos tipos de componentes que se explican más adelante. Al crear una aplicación de interfaz gráfica, usualmente, la clase JFrame debe de ser la superclase para la mayor parte de aplicaciones que cuenten con interfaz gráfica, ya que como se ha mencionado antes, tiene la capacidad de alojar diferentes componentes siendo por esta razón conocido como un contenedor. Al crear una clase que extienda de JFrame es posible acceder a todos sus métodos.

  Los más comunes son: **setTitle**(), el cual recibe como parámetro una cadena de caracteres que es mostrada en la barra de título de la ventana; **setSize**() el cual recibe como parámetros valores que representan el ancho y alto de la ventana mismos que son tomados para definir las dimensiones de la ventana; **setDefaultCloseOperation**()

el cual recibe como parámetro un valor constante que se encuentra dentro de la misma clase JFrame(EXIT_ON_CLOSE,DISPOSE_ON_CLOSE y DO_NOTHING_ON_CLOSE) que le indica el comportamiento de la ventana cuando se realiza clic en el botón de cerrar ubicado en la parte superior derecha de la ventana; **add**(), el cual recibe como parámetro cualquier componente que sea capaz de alojarse dentro de la ventana; y por último **setVisible**(), el cual tecibe un valor booleano el cual le indica a la ventana si es visible o no al usuario. Desde luego la clase JFrame posee más métodos que permiten darle una mayor funcionalidad y personalizacón a la ventana [12]. En la Figura 2.12 se muestra la clase para la creación de una ventana simple.

```
1  import javax.swing.JFrame;
2  public class Ventana extends JFrame{
3      public Ventana(){
4          setTitle("Ventana nueva");
5          setSize(300,200);
6          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7          setVisible(true);
8      }
9      public static void main(String[] args){
10         Ventana v = new Ventana();
11     }
12
13 }
```

(a) Clase ventana



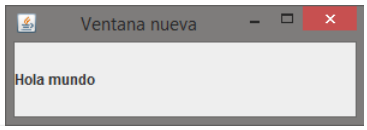
(b) Ejecución de código

Figura 2.12: Creación de una ventana simple

- **JLabel:** Esta clase representa una etiqueta, la cual en una GUI ayuda a mostrar información al usuario, usualmente únicamente se muestra una sola linea de texto. Debido a que el objetivo de este componente es de carácter informativo, los métodos más utilizados para dicho elemento son: **setText**(), el cual recibe como parámetro una cadena de caracteres, misma que es vista en el componente; y **getText**(), el cual devuelve la cadena de caracteres que contiene el componente, haciendo uso del código mostrado en la Figura 2.13 se agrega el componente JLabel con un saludo [12].

```java
import javax.swing.JFrame;
import javax.swing.JLabel;
public class Ventana extends JFrame{
    private JLabel etiqueta1;
    public Ventana(){
        setTitle("Ventana nueva");
        setSize(300,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        initComponents();
        setVisible(true);
    }
    public static void main(String[] args){
        Ventana v = new Ventana();
    }
    private void initComponents(){
        etiqueta1 = new JLabel();
        etiqueta1.setText("Hola mundo");
        add(etiqueta1);
    }
}
```

(a) Clase Ventana



(b) Ejecución de código

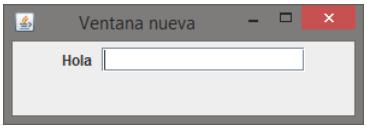
Figura 2.13: Ventana con JLabel

- **JTextField:** Esta clase representa una caja de texto, en la cual el usuario puede ingresar información. Los métodos más usados de este componente son: **getText**(), mismo que devuelve la información contenida dentro de este elemento; **setText**() el cual recibe una cadena de caracteres que ha de observarse dentro de dicho componente; y **setEditable**() el cual recibe valores booleanos que indican si el componente puede ser editado o no por el usuario.Haciendo uso del código mostrado anteriormente, se incluye un componente **JTextField** en la Figura 2.14. Cabe mensionar que para poder visualizar más de un componente en la ventana, es necesario crear un gestor de distribución tambien llamado layout el cual se explica más adelante, para fines prácticos y poder visualizar los componentes se crea un FlowLayout [12].

```java
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import java.awt.FlowLayout;
public class Ventana extends JFrame{
    private JLabel etiqueta1;
    private JTextField textField1;
    public Ventana(){
        setTitle("Ventana nueva");
        setSize(300,100);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        initComponents();
        setVisible(true);
    }
    public static void main(String[] args){
        Ventana v = new Ventana();
    }
    private void initComponents(){

        etiqueta1 = new JLabel();
        textField1 = new JTextField("",15);
        etiqueta1.setText("Hola ");
        add(etiqueta1);
        add(textField1);
    }
}
```

(a) Clase Ventana



(b) Ejecución de código

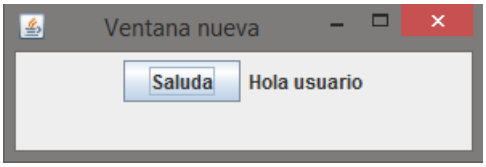
Figura 2.14: Ventana con JTextField

- **JButon:** Esta clase representa un botón, el cual sirve para realizar acciones o ciertas tareas dentro de un programa. Para que este componente tenga funcionalidad es necesario agregarle un evento, tema que se explica más adelante, para fines prácticos, se le asigna un evento con el método **addActionListener**() como se muestra en la Figura 2.15, y su tarea es mostrar un saludo en la ventana.

```java
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.FlowLayout;
import java.awt.event.*;
public class Ventana extends JFrame{
    private JLabel etiqueta1;
    private JButton boton;
    public Ventana(){
        setTitle("Ventana nueva");
        setSize(300,100);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        initComponents();
        setVisible(true);
    }
    public static void main(String[] args){
        Ventana v = new Ventana();
    }
    private void initComponents(){
        etiqueta1 = new JLabel();
        boton = new JButton("Saluda");
        boton.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                etiqueta1.setText("Hola usuario");
            }
        });

        add(boton);
        add(etiqueta1);
    }
}
```

(a) Clase Ventana



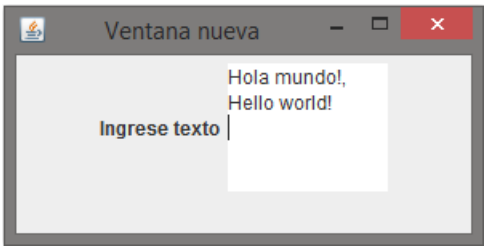
(b) Ejecución de código

Figura 2.15: Ventana con JButton

- **JTextArea:** Esta clase a diferencia de **JTextArea** y **JLabel**, permite ingresar o mostrar un mayor número de líneas de texto, algunos métodos que posee son: **getText**(), **setText**() y **setEditable**() los cuales realizan las mismas operaciones que en elementos antes vistos;**setLineWrap**() el cual recibe un valor booleano que indica si se ajustan las líneas al tamaño del componente o no; y **setWrapStyleWord**() el cual recibe un valor booleano que indica si se usan limites de palabra para al ajustar las líneas de texto. En la figura 2.16 se muestra el uso de dicho componente [12].

```java
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextArea;
import java.awt.FlowLayout;
import java.awt.event.*;
public class Ventana extends JFrame{
    private JLabel etiqueta1;
    private JTextArea areaTexto;
    public Ventana(){
        setTitle("Ventana nueva");
        setSize(300,150);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        initComponents();
        setVisible(true);
    }
    public static void main(String[] args){
        Ventana v = new Ventana();
    }
    private void initComponents(){
        etiqueta1 = new JLabel("Ingrese texto");
        areaTexto = new JTextArea("",5,8);
        areaTexto.setLineWrap(true);
        areaTexto.setWrapStyleWord(true);
        add(etiqueta1);
        add(areaTexto);
    }
}
```

(a) Clase Ventana



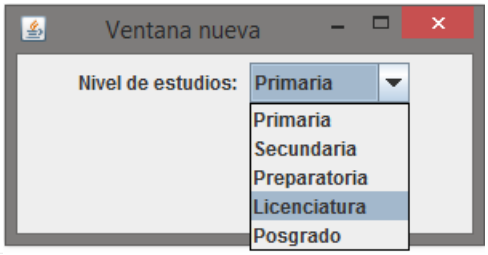
(b) Ejecución de código

Figura 2.16: Ventana con JTextArea

- **JComboBox:** Esta clase representa una lista desplegable, este componente contiene elementos y permite al usuario elegir uno de ellos, esto permite ahorrar espacio en una ventana y hace a una aplicación más elegante. Este componente tiene diversos métodos para poder manipularlo, los más comunes son: **addItem**() el cual recibe un elemento de cualquier tipo de dato, usualmente son cadenas de caracteres [12]; **getSelectedItem**() el cual retorna elemento que ha sido seleccionado de la lista. Generalmente este componente es inicializado con un arreglo de elementos pasado como parámetro al crear el objeto. En la Figura 2.17 se muestra el uso de dicho componente.

```java
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import java.awt.FlowLayout;
import java.awt.event.*;
public class Ventana extends JFrame{
    private JLabel etiqueta1;
    private JComboBox combo;
    public Ventana(){
        setTitle("Ventana nueva");
        setSize(300,150);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        initComponents();
        setVisible(true);
    }
    public static void main(String[] args){
        Ventana v = new Ventana();
    }
    private void initComponents(){
        etiqueta1 = new JLabel("Nivel de estudios: ");
        String[] items ={"Primaria","Secundaria",
            "Preparatoria","Licenciatura","Posgrado"};
        combo = new JComboBox(items);
        add(etiqueta1);
        add(combo);
    }
}
```

(a) Clase Ventana



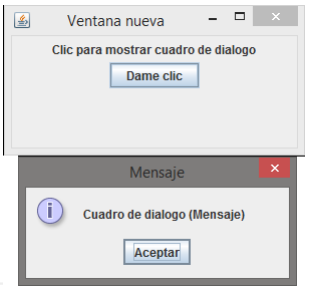
(b) Ejecución de código

Figura 2.17: Ventana con JComboBox

- **JOptionPane:** Es un tipo de ventana también conocido como cuadro de dialogo que, a diferencia de un JFrame, esta contiene ciertas restricciones en cuanto a su uso. El tiempo de vida de un cuadro de dialogo es menor al de un JFrame, es decir, permanece visible al usuario lo suficiente para realizar una tarea específica, además este tipo de ventanas tienen un formato específico que los identifica. JOptionPane cuenta con tres tipos de ventanas: Diálogo de mensaje, de entrada y de confirmación. El cuadro de diálogo de mensaje, usualmente se usa para proporcionar información al usuario. En la Figura 2.18 se muestra el código para crear un cuadro de dialogo de mensaje [12].

```java
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JButton;
import java.awt.FlowLayout;
import java.awt.event.*;
public class Ventana extends JFrame {
    private JLabel etiqueta1;
    private JButton butt;
    public Ventana() {
        setTitle("Ventana nueva");
        setSize(300, 150);
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        initComponents();
        setVisible(true);
    }
    Run | Debug
    public static void main(String[] args) {
        Ventana v = new Ventana();
    }
    private void initComponents() {
        etiqueta1 = new JLabel("Clic para mostrar cuadro de dialogo");
        butt = new JButton("Dame clic");
        butt.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null, "Cuadro de dialogo (Mensaje)");
            }
        });
        add(etiqueta1);
        add(butt);
    }
}
```

(a) Clase Ventana



(b) Ejecución de código

Figura 2.18: Uso de JOptionPane

2.2 Generalidades de las Bases de datos relacionales

2.2.1 Conceptos

Es demasiado común confundir la definición de los datos y asumir que es lo mismo que la información. Es por lo anterior que antes de definir el concepto de una base de datos, aclarar qué son los datos y la información con un enfoque a las bases de datos.

1. **Datos:** El término datos hace referencia a los hechos brutos registrados en la base de datos [4]. En otras palabras, los datos son la parte atómica de la información. Los datos por sí solos no poseen un sentido, son sólo números, letras, símbolos o palabras que "no significan nada", carecen de sentido.

2. **Información:** La información consiste en datos procesados, organizados de una forma que es útil para tomar decisiones [4]. La información es el conjunto de

datos que organizados y aplicados en cierto contexto son capaces de aportar conocimiento hasta un cierto punto.

2.2.2 ¿Qué es una base de datos relacional?

Una base de datos es un conjunto de datos persistentes que es utilizado por los sistemas de aplicación de alguna empresa dada. Se trata de una colección de archivos relacionados los cuales no son independientes entre sí, en dichos archivos se encuentra almacenada la representación abstracta del dominio del problema, es decir la forma lógica y física de cómo son vistos los datos externamente [5] [28].
Un sistema de base de datos es una colección de datos interrelacionados y un conjunto de programas que permiten a los usuarios acceder y modificar dichos datos [27] [21].
El término bases de datos, comenzó a acuñarse en los años sesenta. En esta época la información era representada y almacenada en archivos de texto plano, los cuales no estaban de ninguna forma relacionados entre sí, por lo anterior existieron problemas en la integridad y redundancia de datos. [28]
Las bases de datos cuentan con diversas características, mismas que deben cumplirse para ser consideradas como tal. Algunas de estas características se describen en la tabla 2.6.

2.2.3 Vista de datos

Un mayor propósito de los sistemas de bases de datos, es proveer al usuario un punto de vista abstracto de los datos, es decir, el sistema oculta ciertos detalles de cómo son almacenados los datos [27].

Abstracción de datos

El objetivo de usar sistemas de bases de datos es que esta pueda recuperar datos eficientemente. Usualmente, el tipo de usuarios que utilizan los sistemas de bases de datos son usuarios finales que no tienen mucha experiencia en el desarrollo de estos

Característica:	Definición
Desempeño	Se debe de asegurar un óptimo tiempo de respuesta en la comunicación, así como permitir un acceso simultaneo.
Mínima redundancia	Eliminar la redundancia de datos dentro de la base de datos, siempre y cuando no propicie mayor complejidad o disminución en su desempeño.
Capacidad de acceso	Una base de datos debe de ser capaz de responder en tiempos adecuados a cualquier tipo de petición que involucre los datos que contienen. Esto depende de la organización física de los datos dentro de la base de datos.
Integridad	Esta característica se refiere a la veracidad de la información alojada en la base de datos, es decir, que no sean destruidos ni modificados de forma anómala.
Seguridad	Es la capacidad que tiene la base de datos para proteger la información contra perdida total o parcial, ya sea por fallos del sistema o por accesos accidentales y malintencionados.
Privacidad	Es la capacidad de la base de datos para restringir información a personas que no poseen permisos para obtener acceso a ella.

Tabla 2.6: Características de una base de datos

sistemas, por lo que los desarrolladores ocultan el procesamiento de los datos a dichos usuarios, obteniendo los siguientes niveles de abstracción:

1. **Nivel físico:** Este es el nivel más bajo de abstracción, en el cual se muestra y describe la manera de cómo están almacenados los datos dentro de la base de datos, que son básicamente estructuras complejas de bajo nivel.

2. **Nivel lógico:** Es un nivel posterior al nivel físico, el cual describe que datos se encuentran almacenados, así como la relación en entre sí. En otras palabras, describe toda la base de datos.

3. **Nivel de visualización:** Es el nivel más alto de abstracción que describe sólo una parte de la base de datos como tal. Esto hace más fácil el entendimiento de

los datos que se encuentran almacenados dentro de la base de datos, así como la interacción de los usuarios con el sistema.

2.2.4 Modelo E-R (Entidad-Relación)

El modelo entidad-relación, ayuda a prototipar y esquematizar una base de datos, dicho modelo posee algunos objetos que ayudan a realizarlo apropiadamente. Estos objetos son conocidos como entidades, y relaciones entre dichos objetos. Este modelo es un ejemplo de lo que se denomina modelo semántico [4] [29].

Las bases de datos pueden ser representadas gráficamente haciendo uso de un diagrama de entidad relación, actualmente existen muchos modelos para realizarlos, la manera más común de hacerlo es usando el modelado de lenguaje unificado (UML, por sus siglas en inglés) [21] [29].

En la realización del diagrama ER se deben de contemplar los esquemas simbólicos mostrados en la Figura 2.19 A groso modo, una entidad es una "cosa" u "objeto" del mundo real y es distinguible de otros objetos, por ejemplo, las personas, animales, productos, autos, etc. Una entidad es un tipo de objeto definido en base a la agregación

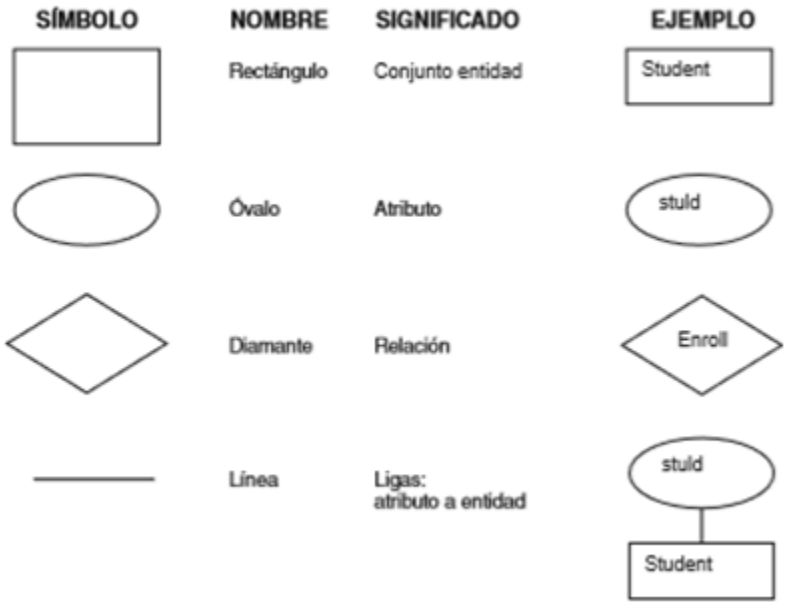


Figura 2.19: Esquemas simbólicos diagrama E-R [4]

de una serie de atributos, es decir, se considera a una entidad como un objeto real o abstracto que forma parte del sistema o problema. Dicha entidad cumple con las siguientes propiedades [22] [28]:

1. Tiene existencia por sí mismo, es decir, la entidad existe como un elemento que interviene en el comportamiento global del sistema

2. Es distinguible del resto de entidades que intervienen en el sistema

Existen dos tipos de entidades dentro de las bases de datos en el modelo entidad relación, las entidades débiles y entidades fuertes [28].
Las entidades débiles poseen dos tipos de debilidades:

1. **Debilidad por identificación:** Una entidad débil por identificación no puede ser identificada a menos que se identifique una entidad fuerte por cuya existencia se presenta dicha debilidad

2. **Debilidad por existencia:** Una entidad débil por existencia puede ser identificada sin la necesidad de la entidad fuerte por la cual existe.

Como ya se ha mencionado antes una entidad pose uno o más atributos, dichos atributos representan ciertas características que identifican y hacen única a dicha entidad, por ejemplo, se tiene una entidad denominada “persona”, dicha persona posee una edad, una estura, un género, y todas estas características son atributos simples con valores atómicos, es decir, no pueden descomponerse en datos más pequeños. Existe otro tipo de atributo llamado compuesto, el cual puede descomponerse en atributos más específicos, siguiendo el ejemplo de la persona, posee una dirección de domicilio, dicha dirección se puede descomponer en calle, barrio o colonia, municipio, estado, país y código postal.

Por otro lado, se pueden encontrar atributos derivados, estos atributos no son almacenados dentro de la base de datos ya que son atributos en los que su valor se puede calcular en cualquier momento. En la Figura 2.20 se muestra un ejemplo de cómo deben ser representados los atributos en el diagrama E-R.

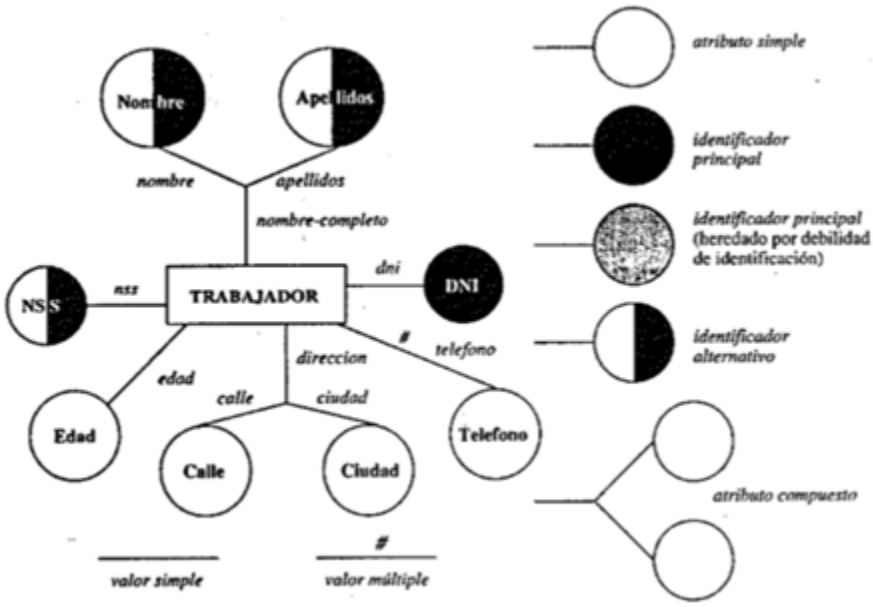


Figura 2.20: Representación de atributos en diagrama E-R [22]

Nótese que en la Figura 2.20 hay tres tipos de atributos más llamados identificadores. Este tipo de atributos funcionan como un mecanismo que evita ambigüedades dentro de la base de datos, es decir, ayuda a distinguir una entidad de otra. El identificador principal toma valores únicos; el segundo tipo denominado identificador alternativo, en algún punto estos pueden realizar la función del identificador principal, es por dicha razón por las que se considera una alternativa para identificar las entidades pero no son tan fuertes como el identificador principal. El último tipo es heredado, es un atributo que forma parte de una entidad débil, es decir, es un atributo identificador que se encuentra en otra entidad, sirve para representar una interrelación débil por identificación entre ellas.

2.2.5 Cardinalidad

Una relación, es una asociación entre varias entidades, es decir, una forma de describir como interactúan diferentes entidades. Existen ciertas restricciones en cuanto a relaciones se trata, una de ellas se le denomina cardinalidad .

La cardinalidad de una relación es el número de entidades a las que otra entidad puede mapear bajo dicha relación. Sean X y Y conjuntos de entidades y R una relación binaria de X a Y. Si no hubiera restricciones de cardinalidad sobre R, entonces cualquier número de entidades en X podría relacionarse con cualquier número de entidades en Y [4].
De lo anterior es posible discernir cuatro tipos de cardinalidad teniendo un conjunto de entidades A y otro conjunto de entidades B:

1. **Relación uno a uno:** Este tipo de relación permite realizar una sola asociación de cada entidad del conjunto A con una del conjunto B y viceversa [19].

2. **Relación uno a muchos:** Este tipo de relación permite realizar más de una asociación de una entidad perteneciente al conjunto A con entidades del conjunto B, sin embargo, una entidad del conjunto B sólo puede asociarse con una sola entidad del conjunto A [19].

3. **Relación muchos a uno:** Este tipo de relación existe cuando cada entidad del conjunto A se asocia a una sola entidad del conjunto B [19].

4. **Relación muchos a muchos:** Este tipo de relación existe cuando cada entidad del conjunto A, puede asociarse a más de una entidad del conjunto B, y cada entidad del conjunto B puede asociarse a más de una entidad del conjunto A [19].

En el diagrama E-R las relaciones anteriores se representan de cinco formas diferentes, para este trabajo se usa la más común, misma que se muestra en la Figura 2.21 siendo de arriba hacia abajo uno a muchos, uno a uno y muchos a muchos.

2.2.6 Sistemas Gestores de Bases de Datos (SGBD)

Un sistema gestor de bases de datos es una capa de software necesaria para crear, manipular y recuperar datos desde una base de datos [23].
Los SGBD desempeñan diversas acciones fundamentales como lo es la seguridad de acceso a los datos, almacenamiento e integridad de los datos, recuperación de datos

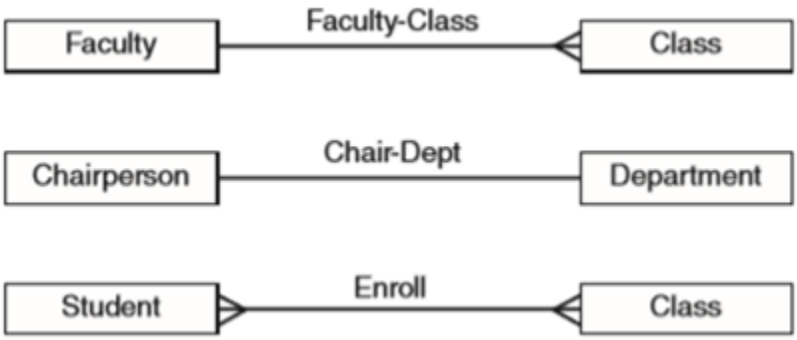


Figura 2.21: Representación simbólica de relaciones diagrama E-R [4]

debido a errores físicos y/o lógicos, control de concurrencia en el acceso a los datos, así como proporcionar la mayor eficiencia de respuesta a peticiones que realizan los usuarios a la base de datos. Los SGBD relacionales son capaces de soportar la arquitectura que se plantea en la abstracción de datos del punto 2.2.3. Al usar dicha arquitectura las bases de datos aportan una mayor independencia de los datos.

El nivel lógico se conforma por tablas, mismas que almacenarán los datos que el usuario final desea que permanezcan dentro de la base de datos. Estas tablas son diseñadas por el administrador de la base de datos haciendo uso del lenguaje SQL.

El nivel físico las tablas se representan en archivos, esta representación la manipula el SGBD por lo que dicha representación no suele coincidir con la del nivel lógico, sin embargo, los registros de la tabla siempre coinciden con las líneas del archivo. El nivel externo es el nivel con el que los usuarios interactúan, se encarga de ayudar al usuario a percibir los datos. En la tabla 2.7 se enlistan algunos sistemas gestores de código abierto y de licencia.

Sistemas Gestores de Bases de Datos	
Open Source	Pago
MySQL	Oracle
PostgreSQL	Windows SQL Server
SQLite	

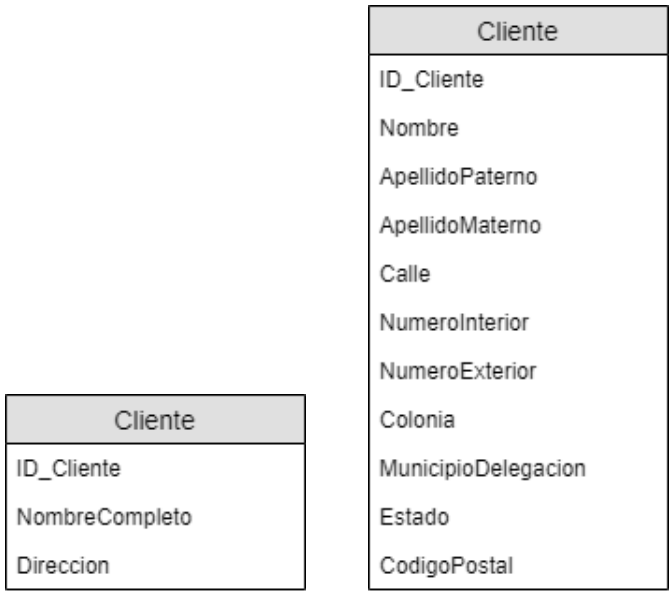
Tabla 2.7: SGBD Open Source y pago

2.2.7 Normalización de bases de datos

Al aplicar la normalización en una base de datos, nos permite eliminar redundancias e incoherencias minimizando de forma considerable la ineficiencia de la base de datos.

Primera forma normal

La primera forma normal indica: *"Una relación R satisface la primera forma normal sí, y sólo sí, todos los dominios subyacentes de la relación R contienen valores atómicos"* [22]. Cuando se hace referencia a la atomicidad de los datos significa que los atributos de las entidades deben de ser lo más simples posibles, por ejemplo, en la Figura 2.22 se tiene la entidad sin la primera forma normal en el inciso a y la misma entidad aplicando la primera forma normal en el inciso b.



(a) Entidad sin primera forma normal

(b) Entidad con primera forma normal

Figura 2.22: Primera forma normal

Como se puede observar en el inciso a solo se tienen tres atributos, el ID_Cliente, NombreCompleto y Dirección; los dos últimos son atributos que se componen de más de un elemento por lo que aplicando la 1FN se pueden descomponer en partes más

simples: NombreCompleto puede ser descompuesto en tres atributos más(nombre, apellido materno y apellido paterno) y Direccion puede descomponerse en siete atributos más(calle, número interior, número exterior, colonia, municipio o delegación, estado y código postal) dando como resultado la entidad del inciso b. Al aplicar esta normalización permite que la entidad sea más flexible al manipularse como por ejemplo, relaciones a otras entidades, filtrados en la búsqueda de registros, etc.

Segunda Forma Normal

La segunda forma normal indica: *"Una relación R satisface la segunda forma normal sí, y sólo sí, satisface la primera forma normal y cada atributo de la relación depende funcionalmente de forma completa de la clave primaria de esa relación"* [22]. Para que una entidad cumpla con la segunda forma normal debe de cumplir antes con la primera forma normal explicada anteriormente, donde cada atributo depende de manera funcional de una única clave primaria, esto evita inconsistencias que puedan afectar la integridad de los datos.

Tercera Forma Normal

La tercera forma normal indica: *"Una relación R satisface la tercera forma normal sí, y sólo sí, satisface la segunda forma normal y cada atributo no primo de la relación no depende funcionalmente de la forma transitiva de la clave primaria de esta relación, es decir, no pueden existir dependencias entre los atributos que no forman parte de la clave primaria de la relación R"* [22]. Para que una entidad cumpla con la tercera forma normal debe de cumplir con la segunda forma normal y los atributos que no formen parte de la clave primaria tampoco pueden depender de otros atributos que no sean identificadores.

2.2.8 El lenguaje SQL

SQL (Structured Query Languaje) es el lenguaje estándar para trabajar con bases de datos relacionales y es soportado por la mayoría de los productos en el mercado de los sistemas gestores de bases de datos. SQL consta de siete partes

1. **Lenguaje de definición de datos:** El DDL por sus siglas en inglés, provee comandos para definir, modificar y eliminar esquemas de relaciones [27].

2. **Lenguaje de manipulación de datos:** El DML por sus siglas en inglés, provee de habilidades de consulta de información proveniente de la base de datos, así como de poder insertar, eliminar y modificar tuplas en la base de datos [27].

3. **Integridad:** En el DDL se incluyen comandos para especificar restricciones de integridad que los datos almacenados en la base de datos deben de satisfacer, así como impedir que actualizaciones infrinjan las restricciones de integridad [27].

4. **Definición de vistas:** En el DDL incluye comandos para definir vistas.

5. **Control de transacciones:** SQL contiene comandos para especificar el comienzo y final de transacciones [27].

6. **SQL embebido y dinámico:** Define cómo declaraciones de SQL pueden ser embebidas con propósitos generales dentro de lenguajes de programación como C/C++, Java, Python, etc [27].

7. **Autorización:** El DDL de SQL incluye comandos para especificar derechos de acceso a relaciones y vistas [27].

2.2.9 Tipos de datos básicos

Al igual que otros lenguajes de programación, SQL cuenta con varios tipos de datos, de los cuales los más comunes son char(n) el cual almacena una cadena de caracteres de tamaño fijo igual a n, varchar(n) que almacena cadenas de caracteres de tamaño

variable no mayor a n, int, smallint, numeric(p,d) donde almacena un numero flotante de 'p' dígitos y 'd' decimales tal que d ya se encuentra incluido en p, real, float, entre otros. Sin embargo, cada tipo de dato ya incluye un valor especial, dicho valor es el null (valor nulo) con el fin de que ciertos atributos de una entidad puedan no existir dentro de un registro en la base de datos.

2.2.10 Definición de tablas

Para crear tablas haciendo uso de SQL, se proporciona un comando para realizarlo, dicho comando es create table nombreDeTabla() [8]. El comando anterior, crea el esquema de una tabla, en la Figura 2.23 se muestra un ejemplo del comando anterior. Como se puede observar, en la línea uno del código se respeta la sintaxis

```
1  create table Cliente(
2      idCliente int(4) not null auto_increment,
3      nombreCliente varchar(40) not null,
4      apellidoPaterno varchar(30) not null,
5      apellidoMaterno varchar(30) not null,
6      fechaNacimiento datetime,
7      primary key(idCliente)
8  );
```

Figura 2.23: Comando SQL para crear tablas

dada anteriormente, dentro de los paréntesis se incluyen n parámetros mismos que representan los atributos que ha de contener la tabla. En la línea dos del código, se define un atributo llamado idCliente, en su estructura se visualiza:
El tipo de dato int(4), el cual indica que dicho atributo es de tipo entero con cuatro dígitos; not null hace mención a que el atributo no puede tomar valores nulos; y por último una directiva auto_increment el cual permite que con cada registro que ha de ingresar en la tabla se incrementa en uno automáticamente dicho atributo. Para los atributos siguientes (nombreCliente, apellidoPaterno, apellidoMaterno), se sigue un patrón en el cual se indica el tipo de dato y la directiva not null, misma que se ha

explicado anteriormente.

El siguiente atributo denominado fechaNacimiento posee un tipo de dato especial (datetime) el cual es capaz de almacenar una fecha y hora. Finalmente se encuentra una regla de integridad, primary key, la cual indica que el atributo idCliente ha de usarse como llave primaria, la cual permite crear relaciones con otras tablas.

2.2.11 Relaciones entre tablas

Las relaciones entre las tablas se dan cuando en una de ellas se comparten identificadores principales de otras, esta relación debe indicarse cuando se crean las tablas haciendo uso de la regla de integridad foreign key y references, haciendo uso de la tabla cliente (Figura 2.23) y la tabla ventas (Figura 2.24) se explica el funcionamiento de dicha regla de integridad.

```
10  create table Venta(
11      idVenta int(8) not null auto_increment,
12      idCliente int(4) not null,
13      fechaVenta datetime,
14      primary key(idVenta),
15      foreign key(idCliente) references Cliente
16  );
```

Figura 2.24: Tabla Venta para crear relación con tabla Cliente

En la tabla de la figura 2.24 se tiene una estructura como se ha visto en el apartado 2.2.8, sin embargo, dentro de ella ha de crearse una relación con la tabla Cliente, esto se hace con la regla foreign key que recibe como parámetro el atributo que pertenece a la tabla con la que se realiza la asociación, cabe mencionar que el nombre y tipo de dato de dicho atributo debe de ser exactamente igual en ambas tablas y por último la directiva references seguida del nombre de la tabla con la que se desea crear la relación.

2.2.12 Eliminación y modificación de tablas

SQL proporciona comandos que permiten la administración de las tablas, dichos comandos permiten modificar y/o eliminar partes internas de la tabla o toda la tabla como tal. Para eliminar una tabla completamente de la base de datos se usa el comando drop table nombreTabla. Por otro lado, se cuenta con el comando delete from nombreTabla el cual, de acuerdo con una condición dada, es posible eliminar uno o más registros de la tabla. Para actualizar una tabla se puede acceder al comando alter table permite cambiar los tipos de datos de cada atributo de la tabla, agregar atributos o eliminarlos de la tabla [8].

2.2.13 Control y manipulación de los datos

El DML de SQL permite al administrador de la base de datos, controlar y manipular los datos. Los comandos existentes en el DML son: select, update, insert, delete. El comando select es usado para la recuperación de información proveniente de la base de datos en la Figura 2.25 se ejemplifica el uso de esta sentencia haciendo uso de la tabla Clientes creada en la Figura 2.23.

```
18  -- Obtener información específica de la tabla Cliente
19  select nombreCliente, fechaNacimiento from Cliente where idCliente='3';
20  -- Obtener toda la información de la  tabla Cliente
21  select * from Cliente;
```

Figura 2.25: Uso de sentencia select

En la línea 19 de la Figura 2.25, se obtienen los datos contenidos en las columnas (atributos) nombreCliente y fechaNacimiento para todos los registros de la tabla en el que su idCliente sea igual a 3. En la línea 21 se obtiene la información de todas las columnas de la tabla clientey al no haber una condición también devuelve cada uno de los registros existentes dentro de la tabla.

El comando update permite realizar cambios en los registros que se encuentran almacenados dentro de la tabla, cabe mencionar que dicho comando permite realizar

dichas actualizaciones tabla por tabla. En la Figura 2.26 se ejemplifica el uso de esta sentencia [8].

```
23  -- modificar un registro específico de la tabla Cliente
24  update Cliente set nombreCliente='Daniel' where idCliente='9';
25  -- modificar tódos los registros de la tabla Cliente
26  update Cliente set apellidoMaterno = 'Velasco';
27  -- modificar un conjunto de registos de la tabla Cliente
28  update Cliente set apellidoPaterno = 'Hernández' where nombreCliente='Daniel';
```

Figura 2.26: Uso de sentencia update

En la línea 24 de la figura anterior, se hace uso del comando update para modificar un registro específico de la tabla cliente, para este caso en el registro que contiene el idCliente igual a nueve, se le cambia el valor de la columna nombreCliente por el de 'Daniel'. En la línea 26 se modifica el valor correspondiente a la columna apellidoMaterno por el de 'Velasco' a todos y cada uno de los registros existentes en la tabla Cliente.Por último, en la línea 28 se camba el valor correspondiente a la columna apellidoPaterno de la tabla Cliente a todos aquellos registros que posean valores igual a 'Daniel' en la columna correspondiente a nombreCliente.

El comando insert es usado para la creación de nuevos registros dentro de una tabla, al igual que el comando update, insert sólo puede usarse en una tabla a la vez, en la Figura 2.27 se ejemplifica el uso de la sentencia [8].

```
30  -- Insertar un registro en la tabla Cliente
31  insert into Cliente(nombreCliente,apellidoMaterno,apellidoPaterno) values(
32      'Evelin','López','Ramírez'
33  );
```

Figura 2.27: Uso de sentencia insert

En las líneas 31 a 33 de la figura anterior se da un ejemplo claro del uso de insert, para este caso de realiza la inserción de un nuevo registro dentro de la tabla clientes, el nuevo registro contiene los valores 'Evelin', 'López' y 'Ramírez' a las columnas nombreCliente, apellidoPaterno y apellidoMaterno respectivamente; cabe mencionar que en la lina 31 entre paréntesis se debe de especificar el nombre de las columnas con los valores a

asignary los parámetros de values se pasan los valores que tendrá el registro en dichas columnas. El comando delete es usado para eliminar registros existentes en una tabla, en la Figura 2.28 se ejemplifica el uso de dicha sentencia [8].

```
35   -- Eliminar un registro específico de la tabla Cliente
36   delete from Cliente where idCliente='9';
37   -- Eliminar tódos los registros de la tabla Cliente
38   delete from Cliente;
```

Figura 2.28: Uso de sentencia delete

En la línea 36 de la Figura anterior, se muestra el uso de la sentencia delete para eliminar un registro específico de la tabla Cliente, en este caso se borra el registro que contiene el idCliente igual a 9. En la línea 38 se eliminan todos y cada uno de los registros que habitan en la tabla.

Clausula JOIN

Al realizar consultas en una base de datos, en ocasiones se presenta la necesidad de obtener información de más de una tabla a la vez de a cuerdo con ciertos criterios sujetos a las necesidades del problema a solucionar; dichas tablas deben de encontrarse relacionadas con al menos alguna de las demás. Para fines explicativos de estaclausula se proponen las tablas mostradas en la Figura 2.29. La cláusula **JOIN** permite ejecutar este tipo de consultas que resultarían más complejas de realizar con la estructura de sentencia **SELECT** presentada anteriormente en la sección 2.2.13 realizando la unión de más de una tabla [32]. Existen diversos tipos de **JOIN**. Para el presente proyecto se presentan los tres más usados:

- **INNER JOIN:** Esta cláusula, devuelve los registros que tienen coincidencias en todas las tablas involucradas en el join, si en algun caso no hay una asociación entre dichos registros, ninguno de ellos aparece en el resultado de la consulta [31]. En ley de conjuntos, esta operación es un equivalente a la intersecciόm, la Figura 2.30 muestra un ejemplo de uso de esta cláusula.

```
MariaDB [tesis]> select * from Cliente;
+-------------+------------+-------------+
| id_cliente  | nombre     | telefono    |
+-------------+------------+-------------+
|           1 | daniel     | 5574722394  |
|           2 | brenda     | 5540123690  |
|           3 | juan       | 5585213697  |
|           4 | yadira     | 5514120300  |
|           5 | marcos     | 5579584210  |
|           6 | karen      | 5510806512  |
|           7 | ismael     | 5540123654  |
|           8 | Janeth     | 5541723980  |
|           9 | Monserrath | 7714958632  |
+-------------+------------+-------------+
9 rows in set (0.00 sec)
```

(a) Tabla Cliente

```
MariaDB [tesis]> select * from vuelo;
+----------+---------+---------------+
| id_vuelo | destino | id_claseVuelo |
+----------+---------+---------------+
|        1 | Londres |             3 |
|        2 | México  |             2 |
|        3 | España  |             1 |
|        4 | Dubai   |             3 |
|        5 | Arfica  |             1 |
|        6 | Chile   |             1 |
|        7 | Brasil  |             2 |
|        8 | Canadá  |             2 |
+----------+---------+---------------+
8 rows in set (0.00 sec)
```

(b) Tabla Vuelo

```
+-----------+----------+------------+------------+
| id_reserv | id_vuelo | id_cliente | fecha      |
+-----------+----------+------------+------------+
|         1 |        1 |          7 | 1995-06-22 |
|         2 |        2 |          4 | 1995-05-01 |
|         3 |        2 |          2 | 1995-05-15 |
|         4 |        7 |          1 | 1995-06-22 |
|         5 |        3 |          2 | 1995-06-22 |
|         6 |        3 |          7 | 1995-02-01 |
|         7 |        5 |          4 | 1995-06-22 |
|         8 |        1 |          1 | 1995-07-02 |
|         9 |        1 |          2 | 1995-07-02 |
|        10 |        2 |          6 | 1995-08-22 |
|        11 |        2 |          1 | 1995-08-22 |
|        12 |        1 |          2 | 1995-08-22 |
|        13 |        5 |          5 | 1995-08-21 |
+-----------+----------+------------+------------+
```

(c) Tabla Reservaciones

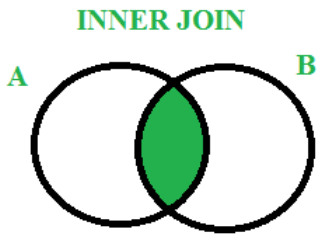
```
MariaDB [tesis]> select * from clasevuelo;
+---------------+--------+---------------+
| id_claseVuelo | costo  | descripcion   |
+---------------+--------+---------------+
|             1 | 105.00 | Comercial     |
|             2 | 200.95 | Primera clase |
|             3 | 999.00 | Privado       |
+---------------+--------+---------------+
3 rows in set (0.00 sec)
```

(d) Tabla ClaseVuelo

Figura 2.29: Base de datos Aerolínea

```
MariaDB [tesis]> select Reservaciones.id_cliente,
    -> Cliente.nombre, Reservaciones.id_vuelo
    -> from Cliente inner join Reservaciones
    -> on (Cliente.id_cliente=Reservaciones.id_cliente)
    -> order by Reservaciones.id_reserv;
+------------+--------+----------+
| id_cliente | nombre | id_vuelo |
+------------+--------+----------+
|          7 | ismael |        1 |
|          4 | yadira |        2 |
|          2 | brenda |        2 |
|          1 | daniel |        7 |
|          2 | brenda |        3 |
|          7 | ismael |        3 |
|          4 | yadira |        5 |
|          1 | daniel |        1 |
|          2 | brenda |        1 |
|          6 | karen  |        2 |
|          1 | daniel |        2 |
|          2 | brenda |        1 |
|          5 | marcos |        5 |
|          7 | ismael |        5 |
|          2 | brenda |        1 |
|          3 | juan   |        3 |
+------------+--------+----------+
16 rows in set (0.00 sec)
```

(a) Query INNER JOIN



(b) Ley de conjuntos INNER JOIN

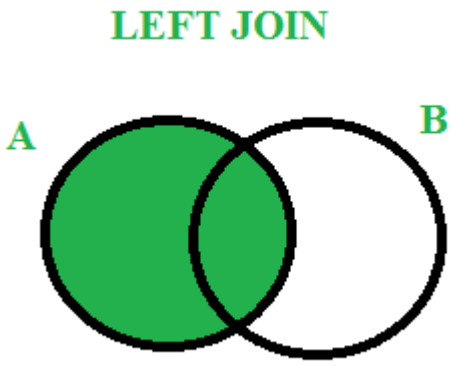
Figura 2.30: Uso de **INNER JOIN**

Al aplicar inner join a las tablas Cliente y Reservaciones como se muestra en la Figura 2.30, podemos acceder a la información de ambas tablas y como se ha hecho mención anteriormente regresa todos los registro que se encuentren presentes en ambas tablas. Para este ejemplo, se obtienen los valores de las columnas id_cliente y id_vuelo de la tabla Reservaciones así como el valor de la columna nombre correspondiente a la tabla Cliente, desde la intersección de la unión de la tabla Cliente y Reservaciones; y retorna los registros cuando el valor de la tabla Cliente en la columna id_cliente sea igual al valor de la tabla Reservaciones en la columna id_cliente; y como resultado final, todos los registros encontrados los ordena de acuerdo al valor de la tabla Reservaciones en la columna id_reservaciones. Nótese que en este resultado no se encuentran los usuarios Janeth y Monserrath, esto es debido a que aún no existe una relación de dichos registros con los registros de de la tabla Reservaciones.

- **LEFT JOIN:** Tomando en cuenta dos tablas (A,B), que tienen registros relacionados entre sí, **LEFT JOIN** regresa todos los registros de la tabla A que se encuentren asociados o no, con los registros de la tabla B, siendo valores nulos paro todos aquellos donde no encuentre la relación. En ley de conjuntos, es el equivalente a tener la unión de A con la intersección de A y B, es decir, A∪(A∩B) [33]. En la Figura 2.31 se muestra el uso de este tipo de join.

```
MariaDB [tesis]> select Cliente.id_cliente,
    -> Cliente.nombre,Reservaciones.id_vuelo
    -> from Cliente left join Reservaciones
    -> on
    -> (Cliente.id_cliente=Reservaciones.id_cliente)
    -> order by Cliente.id_cliente;
+------------+-----------+----------+
| id_cliente | nombre    | id_vuelo |
+------------+-----------+----------+
|          1 | daniel    |        7 |
|          1 | daniel    |        1 |
|          1 | daniel    |        2 |
|          2 | brenda    |        2 |
|          2 | brenda    |        3 |
|          2 | brenda    |        1 |
|          2 | brenda    |        1 |
|          2 | brenda    |        1 |
|          3 | juan      |        3 |
|          4 | yadira    |        2 |
|          4 | yadira    |        5 |
|          5 | marcos    |        5 |
|          6 | karen     |        2 |
|          7 | ismael    |        1 |
|          7 | ismael    |        3 |
|          7 | ismael    |        5 |
|          8 | Janeth    |     NULL |
|          9 | Monserrath|     NULL |
+------------+-----------+----------+
18 rows in set (0.00 sec)
```

(a) Query LEFT JOIN



(b) Ley de conjuntos LEFT OIN

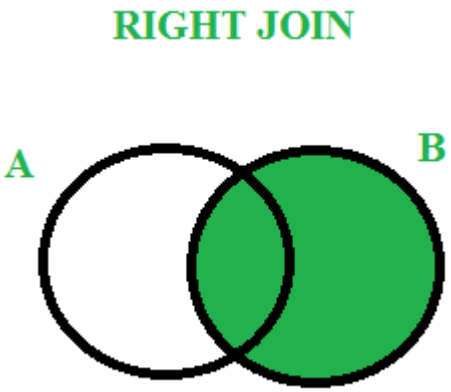
Figura 2.31: Uso de **LEFT JOIN**

En el caso de la Figura 2.31, al aplicar **LEFT JOIN** a las tablas Cliente y Reservaciones, se pueden acceder la la información persistente en ellas. En la sentencia mostrada, se obtienen los valores de las columnas id_cliente y nombre de la tabla Cliente, así como del valor de la columna id_vuelo de la tabla Reservaciones desde la unión de la tabla Cliente con la intersección de las tablas Cliente y Reservaciones. Como se puede apreciar, regresa todos los registros de la tabla Cliente que se encuentran o no asociados con la tabla Reservaciones, a diferencia de **INNER JOIN**, aquí si se muestran los usuarios Janeth y Monserrath y en el campo id_vuelo se les asigna el valor nulo, ya que no poseen ningun vuelo.

- **RIGHT JOIN:** Este tipo de join tiene el mismo principio que **LEFT JOIN**, sólo que en lugar de devolver todos los registros de la tabla A, devuelve los registros de la tabla B. En ley de conjuntos es el equivalente a tener la unión de B con la intersección de A y B, es decir, B∪(A∩B) [34]. En la Figura 2.32 se muestra el uso de este tipo de join. Para la Figura 2.32, al aplicar **RIGTH JOIN** a las tablas Vuelos y reservaciones, es posible acceder a los registros de dichas tablas. En la sentencia mostrada, se obtienen los calores de las columnas id_vuelo y destino

```
MariaDB [tesis]> select vuelo.id_vuelo,
    -> vuelo.destino, Reservaciones.id_reserv
    -> from Reservaciones
    -> right join Vuelo
    -> on
    -> (Reservaciones.id_vuelo=Vuelo.id_vuelo);
+----------+---------+-----------+
| id_vuelo | destino | id_reserv |
+----------+---------+-----------+
|        1 | Londres |         1 |
|        1 | Londres |         8 |
|        1 | Londres |         9 |
|        1 | Londres |        12 |
|        1 | Londres |        15 |
|        2 | México  |         2 |
|        2 | México  |         3 |
|        2 | México  |        10 |
|        2 | México  |        11 |
|        3 | España  |         5 |
|        3 | España  |         6 |
|        3 | España  |        16 |
|        4 | Dubai   |      NULL |
|        5 | Arfica  |         7 |
|        5 | Arfica  |        13 |
|        5 | Arfica  |        14 |
|        6 | Chile   |      NULL |
|        7 | Brasil  |         4 |
|        8 | Canadá  |      NULL |
|        9 | Japon   |      NULL |
|       10 | USA     |      NULL |
+----------+---------+-----------+
21 rows in set (0.00 sec)
```

(a) Query RIGHT JOIN



(b) Ley de conjuntos RIGHT JOIN

Figura 2.32: Uso de **RIGHT JOIN**

pertenecientes a la tabla Vuelo, así como de los valores en la columna id_reserv perteneciente a la tabla Reservaciones desede la unión de la tabla Reservaciones, con la intersección de ambas tablas. Como se puede apreciar, regresa tódos los registros de la tabla Vuelo que se encuentran o no asociados con la tabla Reservaciones, en el caso de los vuelos con destino a Dubai, Chile, Canadá, Japón y USA no se encuentran asociados a ningun registro de la tabla Reservaciones, por lo que en este campo se ve la presencia de valores nulos.

2.2.14 Entornos de Desarrollo Integrado (IDE)

Los entornos de desarrollo integrado (IDE, por sus siglas en inglés) es una aplicación usada para crear software, los IDE ofrecen soporte para uno o varios lenguajes de programación, así como de diferentes herramientas que le permiten al programador crear software de manera más sencilla y eficiente.
Actualmente existen diversos entornos de desarrollo con múltiples propósitos, los más comunes se encuentran:

1. NetBeans
2. Eclipse
3. Visual Studio
4. Aptana Studio
5. Xcode

Durante el desarrollo del presente proyecto, se elige el IDE NetBeans debido a que soporta el lenguaje de programación Java, cuenta con una vista de diseño en la cual se pueden crear vistas gráficas del software a desarrollar y arrastrar dentro de ellas componentes gráficos desde la paleta de componentes, una vista jerárquica de la estructuración del proyecto, una potente herramienta para depurar; permitiendo un desarrollo y planeación más eficiente en el proceso de desarrollo. Por otro lado, NetBeans cuenta con un catálogo de plugins, mismos que pueden ser descargados e instalados para extender la productividad y alcances propios de NetBeans. En la Figura 2.33 se muestra la interfaz gráfica del entorno de desarrollo de NetBeans.

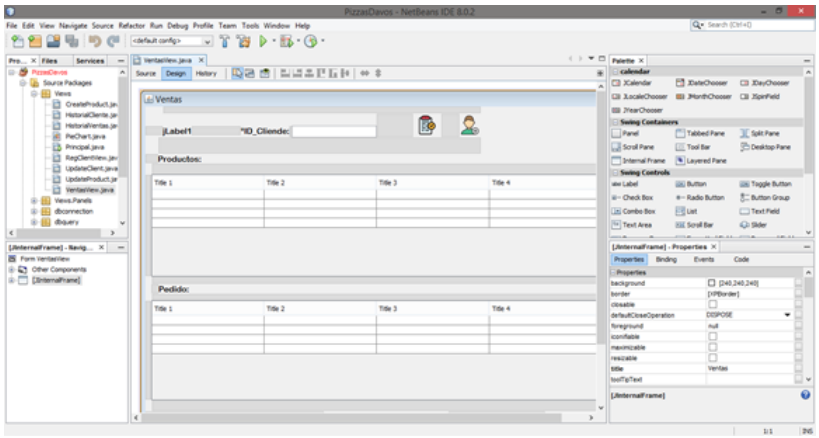


Figura 2.33: Vista de un proyecto en NetBeans

2.3 Modelos de desarrollo de sofware

Los modelos de desarrollo de software (o simplemente modelos de desarrollo) son formas de trabajo planeadas con el objetivo de organizar, administrar y desarrollar proyectos de manera colaborativa entre distintos equipos en tiempos considerablemente cortos. Actualmente existen muchos tipos de modelos y formas de trabajo por lo que las empresas comienzan a adaptar estas "nuevas" tendencias en sus ideologías. Dependiendo de las necesidades de cada empresa o corporativo, se elige un modelo para desarrollo de software que se adapte. Sin embargo, las empresas usualmente adoptan dos o más formas de trabajo tomando las mejores características de cada una y combinándolas.

A continuación, se presentan tres de los modelos comúnmente utilizados en la industria de desarrollo en México.

2.3.1 Modelo en cascada (*Waterfall*)

Hasta hace algunos años gran parte de las empresas y corporativos trabajaban en el desarrollo de software basados en este modelo. *Waterfall* es un modelo de desarrollo secuencial que en teoría "funciona" bien cuando los requerimientos de las necesidades de la empresa son perfectamente visibles y definidos, además de que se esperan pocos

cambios durante el desarrollo del proyecto [25]. Es por lo anterior que este modelo es poco eficiente ya que el entregable se libera al final de todo el proceso y la interacción del cliente durante el desarrollo es casi nulo, sólo interactúa al comienzo y al final del proceso de desarrollo.

Una de las desventajas más notables de este modelo es que existen tiempos muertos para los diferentes equipos involucrados en el desarrollo del entregable, ya que se necesita que el equipo en turno finalice el proceso actual para que miembros de otros equipos continúen realizando los procesos correspondientes. En la Figura 2.34 se muestra el diagrama del modelo, cuyas etapas se decriben a continuación.

Figura 2.34: Modelo Waterfall

Definición de los requerimientos: En este punto se establece comunicación entre el cliente que solicita el entregable y los analistas; este paso es primordial ya que en esta reunión es la única en la que el cliente interactúa en el desarrollo del proyecto y se describen a muy alto nivel las necesidades, alcances, reglas de negocio y diseño de las interfaces gráficas de usuario .

Diseño: Con base en los requerimientos planteados entre el cliente y analistas se diseña el entregable en el ámbito físico y lógico por parte de analistas, arquitectos de software, desarrolladores y los demás miembros del equipo que se encuentren involucrados en el proceso del desarrollo del proyecto. El diseño debe apegarse estrictamente en los requerimientos y reglas de negocio planteados.

Implementación y pruebas de unidades: Una vez concluido el diseño se comienza por desarrollar el entregable en secciones y cuando se encuentran finalizadas se hacen pruebas sobre piezas específicas, cabe mencionar que la solución no está

completamente integrada ya que algunos procesos dependen de otros.

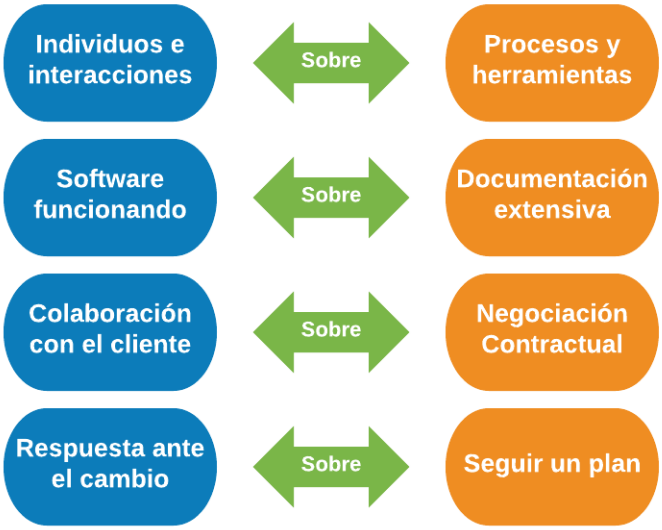
Integración y prueba del sistema: Cuando se termina con el desarrollo de todas las secciones del apartado anterior se comienza por "ensamblar" todos esos módulos generados para incorporar la funcionalidad del sistema completo, realizando las pruebas necesarias para comprobar que todo el entregable se encuentre estable para liberarlo al cliente. Una vez que el proyecto se encuentra óptimo para liberar se hace la entrega al cliente.

Mantenimiento: En este punto se le proporciona soporte al sistema del cliente atendiendo errores, agregando funcionalidades o modificando las actuales siendo un tanto complejos de llevar a cabo. Sí se requiere llevar a cabo la integración de nuevas funcionalidades en el entregable se retorna a la definición de los requerimientos o diseño pasando por todo el flujo visto hasta ahora, pero si el cliente decide modificar aspectos de lo ya desarrollado existen dos escenarios posibles: En el mejor caso donde las modificaciones son muy puntuales y sencillas de realizar, como por ejemplo cambios de estilos, validaciones, corrección de errores,etc. sólo se retorna a la implementación e integración o diseño dependiendo de la modificación a realizar. En el peor caso donde las modificaciones requieren de un cambio en la lógica de la solución se retorna a la redefinición de requerimientos y los flujos consecuentes ya que requiere una mayor cantidad de esfuerzo por parte de todos los equipos involucrados en el desarrollo del entregable debido a que este tipo de cambios causan una reingeniería en todo o gran parte del sistema.

2.3.2 Desarrollo ágil de software (*Agile*)

Las metodologías de desarrollo ágil de software surgen a partir de que grandes empresas detectan que trabajar con las metodologías tradicionales como *waterfall* retrasaba la liberación del entregable al cliente, además de que solía ser de mala calidad. En el año 2001 se reunieron los directores ejecutivos (CEO, siglas de Chief Executive Officer) y profesionales de software pertenecientes a diferentes e importantes empresas de software

para crear el *Agile manifest*. Más que un modelo, *agile* es una ideología que no indica cómo hacer los procesos de desarrollo, sino que proporciona valores y principios para mejorar el desarrollo de software. En la Figura 2.35 se muestran los aspectos que *agile* pondera. Del lado izquierdo se encuentran los aspectos de mayor valor, mientras que a la derecha los que tienen un menor valor, pero sin ser excluidos.



Agile.png

Figura 2.35: Valores *agile*

Principios de *Agile*

De acuerdo con el manifiesto, *agile* se rige bajo los siguientes principios:

1. La mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.

2. Se debe de aceptar que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

3. Se debe entregar software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.

4. Los responsables de negocio y los desarrolladores trabajan en conjunto de forma cotidiana durante todo el proyecto.

5. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.

6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.

7. El software funcionando es la medida principal de progreso.

8. Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.

9. La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.

10. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.

11. Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.

12. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

2.3.3 Modelo *Scrum*

El modelo de trabajo *scrum* es un modelo *agile* que posee un conjunto de prácticas y roles, en la Figura 2.36 se muestra un esquema del modelo. Este modelo no sólo se limita al desarrollo de software sino que también puede ser implementado en otros tipos de proyectos. En este modelo existen 3 roles:

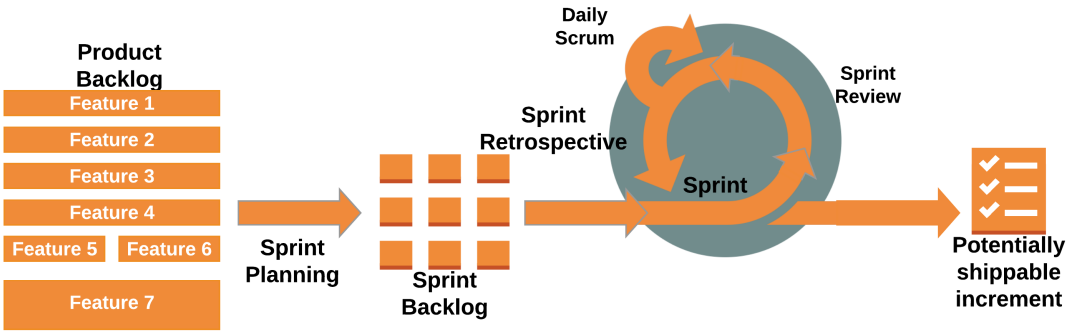


Figura 2.36: Modelo *Scrum* [25]

- ***Product owner***: El *product owner* debe de poseer autoridad para poder realizar la toma de decisiones sobre lo que se debe realizar durante el proyecto ya que uno de sus deberes es priorizar las tareas que el equipo debe de realizar, debe de tener disponibilidad de tiempo, el conocimiento de las necesidades del cliente y del entregable que se llevará a cabo ya que debe entender los requerimientos del cliente y traducirlos al equipo de trabajo además de que debe de resolver las dudas e inquietudes del equipo.

- ***Scrum master***: El *scrum master* sirve al *product owner* y al equipo de trabajo, sus principales funciones es el de guiar al equipo durante el desarrollo del proyecto, ayuda al equipo de trabajo a resolver situaciones que le impidan continuar o realizar sus actividades y debe de conocer ampliamente el proceso *scrum* ya que debe de facilitar las ceremonias de *scrum* que se revisan más adelante.

- **Equipo de desarrollo**: Es un equipo conformado de tres a nueve personas que debe de auto organizarse y responsabilizarse como equipo, debe de entregar un incremento del producto, participar en las ceremonias *scrum*, administrar el *sprint backlog*. El equipo puede solicitar capacitaciones, certificaciones o cursos que le permitan continuar o mejorar su productividad.

Sprint

El *sprint* es el alma de *scrum*, es un bloque de tiempo con una duración máxima de un mes. Al final de este periodo de tiempo se le debe de entregar al cliente una parte utilizable y potencialmente desplegable del proyecto final. Durante el *sprint* no se deben realizar cambios que puedan afectar a los objetivos establecidos, el alcance se puede renegociar con el *product owner*, cuando se termina un *sprint* inmediatamente da comienzo al siguiente *sprint*.

El *sprint* sólo puede ser cancelado por el *product owner* cuando el cliente hace un cambio importante en el proyecto y decide que el módulo con el que se está trabajando actualmente en el *sprint* ya no le es útil.

Artefactos de *Scrum*

Scrum cuenta con una serie de artefactos que permiten al equipo de desarrollo comenzar a trabajar sobre el proyecto.

- ***Product backlog***: El *product backlog* es la división del proyecto completo en pequeñas tareas que deben de ser priorizadas (algunas) exclusivamente por el *product owner* con base en las desiciones del cliente, a cada tarea se le denomina como elemento del *product backlog* (*Product Backlog Item*) y muchos de estos *items* pueden no estar priorizados. Este artefacto es dinámico ya que siempre está en constante cambio debido a que depende de las decisiones del cliente quien en cualquier momento puede quitar y/o agregar funcionalidades, o realizar un re priorizado de los *items* que deben liberarse antes que otras. En la mayoría de equipos de trabajo los PBI son documentos llamados histroias de usuario (*User Strories*).

- ***User Stories*** (Historias de usuario): Las historias de usuario es un requerimiento del entregable definido en un alto nivel que posee un valor agregado para el cliente y no contiene una descripción detallada. Existe una "plantilla" para realizar

correctamente una historia de usuario que corresponde a la siguiente estructura:

$$Como < rol > quiero < acción > paraque < valoragregado >$$

Un ejemplo de una historia de usuario sería el siguiente: *"Como usuario del punto de venta Pizzas Davos quiero almacenar la información de los clientes para evitar solicitar sus datos cuando realizan un pedido vía telefónica"*.

Cuando una historia de usuario es tomada del backlog para ser implementada en el *sprint* se debe de realizar una ceremonia para que el *product owner* resuelva dudas al equipo de desarrollo y es en este punto donde nacen los criterios de aceptación que complementan a las historias de usuario para que al desarrollador y el evaluador les sea mas sencillo entender lo que el cliente solicita, los criterios de aceptación no son fijos y pueden cambiar, son condiciones de satisfacción y son independientes, es decir, sólo pertenecen a una sola historia de usuario.

- ***Sprint backlog***: El *sprint backlog* es un conjunto de historias de usuario seleccionadas por el equipo desde el *product backlog* y que puede desglosar en tareas más pequeñas. El equipo decide el número de historias que puede realizar durante el *sprint*, en otras palabras el *sprint backlog* es una parte del *product backlog* desglosada en pequeñas partes. El equipo puede cambiar las tareas desglosadas sin perder de vista los objetivos del *sprint*.

Ceremonias de *Scrum*

Las ceremonias que se llevan a cabo en *scrum*, son reuniones con un propósito en específico, a continuación se listan las reuniones que *scrum* contempla:

- ***Sprint planning***: En esta ceremonia participan el equipo de desarrollo, *scrum master*, el *product owner* y personal de la empresa que se encuentre interesado en participar(*stakeholders*). Se lleva a cabo al comienzo de cada *sprint* y la duración de esta reunión debe de durar 8 horas máximo por un *sprint* de 1 mes; el equipo

toma del *product backlog* los puntos que puede realizar en el periodo del *sprint*, en este punto el *product owner* se encarga de priorizar los puntos elegidos, resolver dudas y establecer la meta del sprint. De igual manera, el equipo desglosa los puntos tomados del *product backlog* en tareas mucho más sencillas que puedan ser realizadas en en un día ideal o menos.

- ***Daily Scrum***: Es una reunión diaria con duración máxima de 15 minutos y de horario fijo en la que solamente participa el equipo de desarrollo y el *scrum master*. En esta reunión cada miembro del equipo debe de informar un estatus actual respondiendo a las preguntas ¿Qué hice ayer?, ¿Qué haré hoy? y ¿Tengo algún impedimento para realizar mi trabajo? y sirve para llevar una perspectiva común del progreso del trabajo. En *scrum* se propone un tablero con una estructura similar a la de kanban como se muestra en la Figura 2.37.

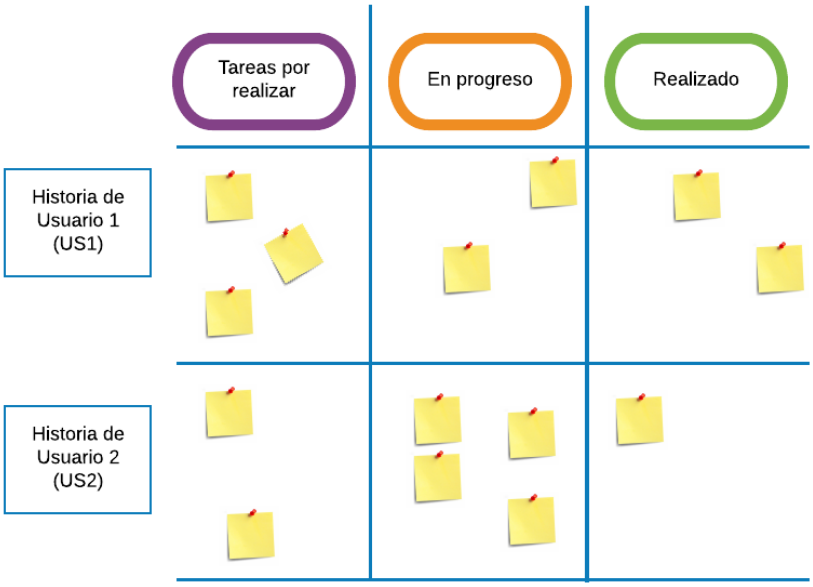


Figura 2.37: Tablero *scrum*

Este tablero puede contener más columnas de acuerdo a la organización del equipo, inicialmente *scrum* propone 3 columnas: *to do* (Tareas por realizar) en la cual se encuentran las tareas que el equipo tiene pendientes por realizar, *in progress*(En

progreso) donde viven todas las tareas que el equipo se encuentra realizando actualmente y por último *done*(Realizado) donde son colocadas todas las tareas que el equipo ha culminado.

- ***Sprint review***: En esta reunión participa el equipo de desarrollo, *product owner*, *scrum* y *stakeholders* en la que el equipo de trabajo muestra los objetivos logrados y tareas realizadas durante la ejecución del *sprint*, resuelven dudas de todos los participantes del *sprint review*, al finalizar reciben una retroalimentación y el *product owner* confirma si las tareas realizadas por el equipo están totalmente terminadas o no y se realiza una actualización del *product backlog*. Esta sesión se lleva a cabo al final de cada *sprint* y tiene una duración máxima de 4 horas.

- ***Retrospective***: En esta ceremonia participa el equipo de desarrollo y el *scrum master* aunque de igual manera puede incorporarse el *product owner*. Se lleva a cabo al final del *sprint review* y tiene como duración máxima de 3 horas por un *sprint* de un mes, aquí el equipo plantea puntos relacionados sobre el último *sprint* analizando actitudes, aptitudes y desempeño entre otros aspectos que permiten al equipo mejorar y crecer.

En esta tesis, el punto de venta fue desarrollado usando la ideología *agile*, aunque sin llevar un registro formal o generar la documentación correspondiente. Lo anterior debido a limitaciones de tiempo y falta de equipo de trabajo (el autor de este documento desempeñó todos los roles para el desarrollo del entregable).

2.4 Arquitectura de software

Actualmente existen muchos modelos de arquitectura de software y dependen de las características del negocio. El proyecto desarrollado en este trabajo se basa en el modelo vista controlador y modelo de dos capas, mismos que son explicados a continuación.

2.4.1 Arquitectura Modelo Vista Controlador

Esta arquitectura -como su nombre lo dice- se compone de tres secciones básicas (capas) el modelo, la vista y el controlador; cada una de esas capas tiene una función definida es por eso que se deben de ubicar elementos del software específicos dentro de ellas. A continuación se describe cada sección.

- **Modelo**: En esta capa se encuentra el acceso y la manipulación de datos, debe de contener toda o gran parte de la lógica de negocio aunque no es una regla vital y se puede delegar un poco de esta lógica a la vista sin que afecte la integridad del sistema.

- **Vista**: En esta capa se encuentra toda la interfaz gráfica con la que el usuario interactúa. Su función es la de digerir la información que proporciona el modelo y presentarla de forma comprensible para el usuario. Como ya se ha mencionado en la sección del modelo, esta capa puede contener lógica de negocio pero debe de ser muy simple, como por ejemplo validaciones de lo que puede o no realizar el usuario de manera gráfica, formato de datos, etc.

- **Controlador**: En esta capa se encarga de responder a eventos que ocurren en el sistema desde la capa de vista causados por el usuario como puede ser entrada de teclado, dar un clic en un botón, etc. y que pueden causar cambios tanto en la vista como en el modelo, es decir gestiona la información proveniente de la vista hacia el modelo así como el comportamiento de ambos.

2.4.2 Back End y Front End

El back end y front end son capas que ayudan a distinguir las funcionalidades de un sistema de software o hardware. Son una forma lógica de abstraer las secciones de un sistema respecto a la comunicación con el usuario y las operaciones que no puede manipular.

Back End

El *back end*, es una capa de software que se encarga de realizar los procesos internos de un sistema, estos procesos dependen de los estímulos e información proveniente del front end. Esta capa se mantiene oculta y aislada del usuario final ya que no necesita de su intervención para realizar sus tareas, es decir, es una forma lógica de agrupar y distinguir las secciones de un sistema que contienen la lógica de negocio y el acceso a datos.

Front End

El *front end* es una capa de software que se encuentra en contacto constante con el usuario final, se encarga de recolectar y abstraer información. Dicha información se pasa a la capa de *back end* y los procesos se lleven a cabo de acuerdo a las especificaciones de ésta ultima capa. En otras palabras, en este estrato se abstraen las secciones de un sistema que son presentadas a un usuario final y solo se encargan de la presentación de información obtenida del *back end*.

Capítulo 3

Desarrollo de proyecto

Introducción

En este capítulo se presenta la arquitectura del software desarrollado para un punto de venta, en específico para la microempresa Pizzas Davos. El sistema se encuentra diseñado en dos capas, a las que llamamos back-end y front-end. La primera capa se encuentra en constante comunicación y control de peticiones con la base de datos diseñada para almacenar diversos datos que serán explicados más adelante. La capa front-end es la encargada de presentar una interfaz gráfica al usuario.

Es importante mencionar que esta arquitectura presentada es válida para cualquier otro negocio, con sus respectivas adecuaciones.

3.1 Requerimientos del sistema

Para llevar a cabo el presente proyecto, primero es necesario plantear los requerimientos del sistema a desarrollar, es decir, las funcionalidades que se necesitan implementar para satisfacer las necesidades del cliente. Un punto importante a considerar es que los usuarios que han de interactuar con el sistema, sólo poseen conocimientos en la manipulación básica del sistema operativo Windows en su versión 7 o superior. Es por lo anterior que el sistema debe se ser lo más sencillo e intuitivo posible. Además de

esto último, no abordaremos más requerimientos no funcionales en este trabajo, como los relacionados a plataformas, lenguajes de programación específicos, desempeño del sistema, etc. debido principalmente a que el sistema puede ser migrado en el futuro por necesidades no conocidas en este momento.

Para lograr determinar los requerimientos, es importante observar y recabar información por parte del propietario sobre la metodología actual en los procesos internos de la microempresa y las problemáticas que necesitan atenderse con un mayor urgencia. Después de llevar a cabo lo anterior, se encontró que los requerimientos para el sistema son los que se enlistan a continuación:

1. Es necesario contar con un catálogo fácilmente actualizable de los productos que la microempresa ofrece a sus clientes. Entre los principales productos se encuentran pizzas de varios sabores y tamaños, así como refrescos.

2. El sistema debe de facilitar el levantamiento de pedidos, mediante una interfaz que permita buscar productos y elegir la cantidad de ellos así como modificar algunas de sus propiedades, como el tamaño.

3. Se requiere llevar un registro de las ventas diarias que realiza la microempresa.

4. Se necesita llevar un registro de los clientes frecuentes.

5. El cliente considera apropiado el poder visualizar el consumo histórico de sus clientes.

6. Se debe de poder consultar el total ventas en una determinada fecha o en un intervalo de días.

7. Se debe de poder generar e imprimir tickets de compra para los clientes y estos deben de contener información del producto o productos adquirido(s), cantidad, costo, costo total y un identificador del cliente al que pertenece.

Aunque en lo personal se observó la posibilidad de incluir más funcionalidades, como la generación de facturas, envío de mensajes de texto de confirmación al cliente, generación

de promociones basadas en las tendencias de las ventas, pago con tarjetas bancarias o de manera electrónica y seguimiento del pedido en tiempo real por parte del cliente, el cliente consideró que por el momento no era necesario.

3.2 Back-end

Una parte fundamental de la capa back-end es la base de datos, el sistema gestor de base de datos elegido para el presente proyecto es MySQL el cual es de código abierto permitiendo ahorrar los recursos financieros que la microempresa ha destinado para su desarrollo, además de que hay compatibilidad con el lenguaje de programación Java en el que se desarrolló el sistema. El servidor de la base de datos se encuentra de manera local instalado en el mismo equipo de cómputo donde se aloja el sistema, cabe mencionar que en un futuro esta base de datos será migrada a un servidor en la nube y adaptada para integrar nuevas conexiones con plataformas de dispositivos *mobile*; a continuación se presenta el diseño de esta última para el sistema desarrollado.

3.2.1 Estructura de la base de datos

La estructura de la base de datos para la micro empresa Pizzas Davos cuenta con tres tablas, que fueron previamente planeadas de acuerdo con las necesidades, requerimientos, entradas y salidas de información; en conjunto con el propietario del negocio. La primera tabla - llamada Clientes- debe de contener los registros de todos aquellos clientes frecuentes del establecimiento, con información de contacto y dirección para realizar pedidos y entregas a domicilio. La segunda tabla - llamada Compras- debe de contener registros de las compras realizadas por todos los usuarios registrados dentro de la tabla Clientes. La tercera tabla - a la que se nombró Productos- debe de tener registrados todos los productos que el establecimiento ofrece a sus clientes. En la figura 3.1 se muestra el diagrama E-R elaborado para describir la interacción interna de la base de datos. La interpretación de las relaciones entre las tablas de la figura 3.1 es la

siguiente: un cliente puede realizar una o muchas compras; cada compra la realiza un único cliente; una compra puede contener uno o muchos productos.

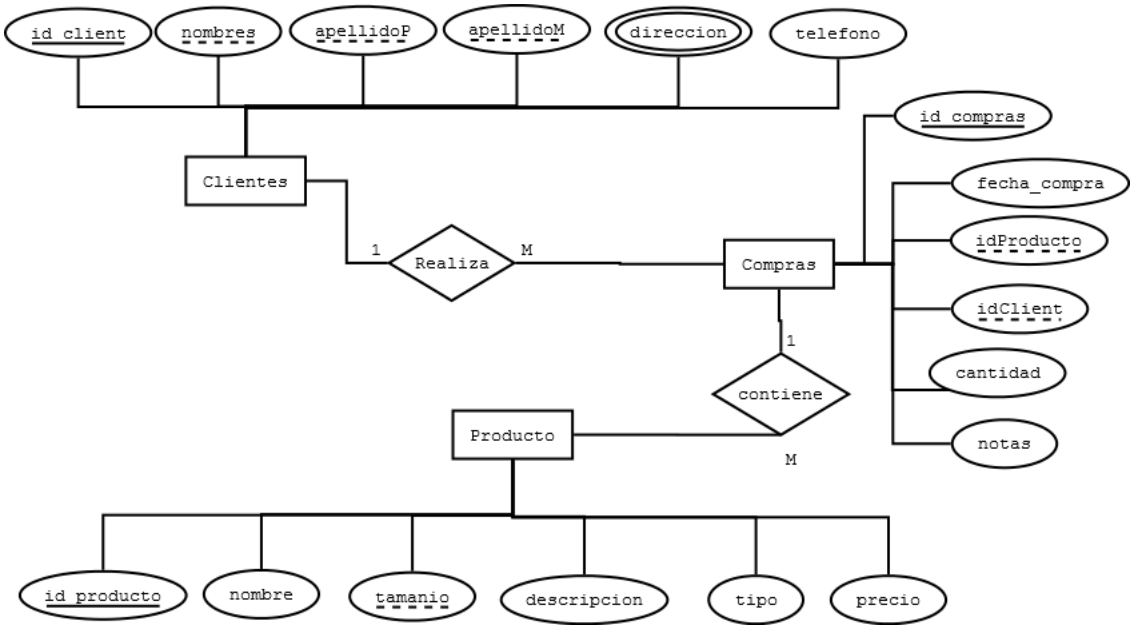


Figura 3.1: Diagrama E-R para la base de datos Pizzas Davos

3.2.2 Clases auxiliares en el flujo de información entre capas

Para facilitar el flujo de información entre ambas capas (back-end y front-end), se crearon tres clases en el lenguaje de programación Java, estas son las siguientes:

1. **ObjectClient:** Encargada de abstraer las características de un cliente.

2. **ObjectProduct:** Abstrae las características de un producto.

3. **ObjectSale:** Utilizada para abstraer las características de una venta.

Las tres clases mencionadas anteriormente permiten mantener una homogeneidad de la información al pasar esta de una capa a otra. La figura 3.2 muestra una descripción de cada clase.

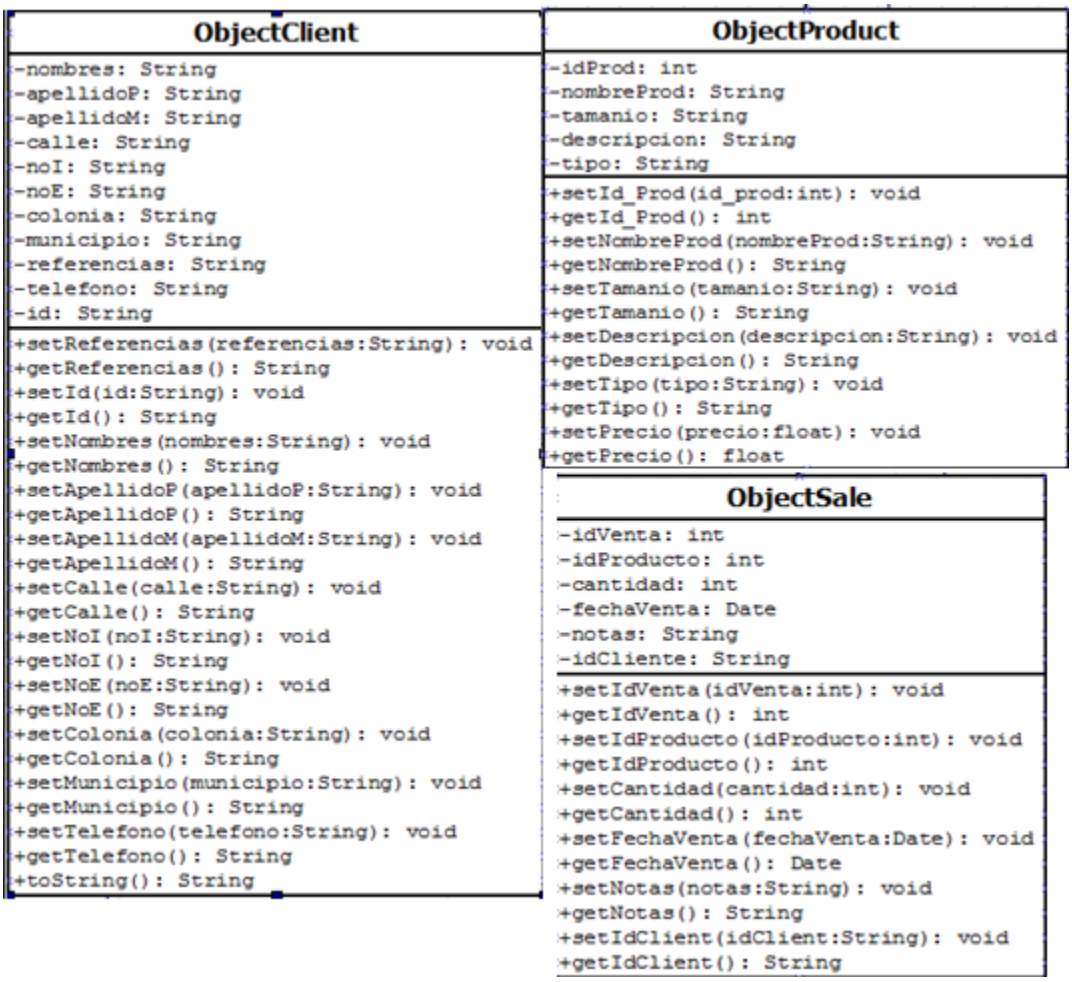


Figura 3.2: Definición de clases ObjectClient, ObjectProduct y ObjectSale

3.2.3 Clases Java para la comunicación con la base de datos

Como ya se ha mencionado anteriormente, el back-end es la capa que se encuentra en contacto directo con la base de datos (lo que separa la vista de los datos y la lógica de control, de la forma recomendada por el patrón modelo-vista-controlador (MVC)), por lo que es necesario contar con clases que permitan gestionar adecuadamente las peticiones que se realicen. En el sistema desarrollado, lo anterior se logró mediante las siguientes seis clases escritas en Java:

1. **DBConnection:** Se encarga de crear conexiones a la base de datos.

2. **DBQuery:** Abstrae atributos y métodos que son utilizados por sus clases hijas

DBQueryActualiza, DBQueryConsulta, DBQueryElimina y DBQueryRegistro.

3. **DBQueryActualiza:** Se encarga de modificar de información en los registros de la base de datos.

4. **DBQueryConsulta:** Se encarga de obtener información de los registros en la base de datos.

5. **DBQueryElimina:** Se encarga de eliminar registros de la base de datos.

6. **DBQueryRegistro:** Se encarga de crear nuevos registros dentro de la base de datos.

En la figura 3.3 se muestra el diagrama de clase para **DBConnection**, esta cuenta con cinco variables, siendo cuatro de ellas cadenas con valores constantes (modificador *final* en Java), que sirven para acceder a la base de datos; y una de tipo **Connection**, que es un objeto que pertenece a la API de **JDBC** (Java Database Connectivity).

Cuando se crea una referencia a la clase **DBConnection** automáticamente realiza una conexión a la base de datos, dicha clase cuenta con dos métodos: **getConnection**(), la cual devuelve el estado de la conexión a la base de datos y **closeConnection**(), método que ayuda a cerrar la conexión del sistema a la base de datos. Es muy importante abrir la conexión a la base de datos antes de realizar alguna consulta a ella y cerrarla cuando no sea utilizada. De otra forma puede dar lugar a excepciones o problemas de inestabilidad del sistema por muchas conexiones activas simultáneamente.

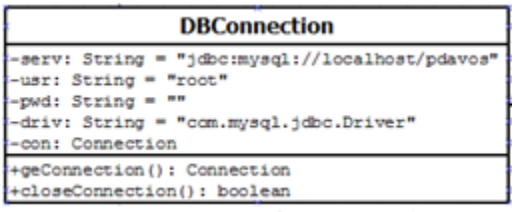


Figura 3.3: Definición de clase DBConnection

En la figura 3.4 se muestra el diagrama de la clase **DBQuery**, esta clase cuenta con cuatro variables, una de tipo **Connection** para obtener el estado de conexión que

devuelve el objeto que hace referencia a la clase **DBConnection** y dos más para poder ingresar y recuperar información de la base de datos. Dentro de la clase **DBQuery** se encuentran los métodos **splitID**() y **concatFullID**() que sirven para generar el ID del cliente, agregando el prefijo DAVOS, que hace referenca al establecimiento. El método **concatFullID**() recibe el id que la base de datos le asigna a un nuevo cliente y lo concatena a el **String** "DAVOS" que ha elegido la empresa para darle a la capa de front-end una mejor presentación; el método **splitID**() recibe un **String** siendo la concatenación de la cadena "DAVOS" y un número entero indicando el id en la base de datos y lo descompone en las partes ya mencionadas. En la figura 3.5 se muestra

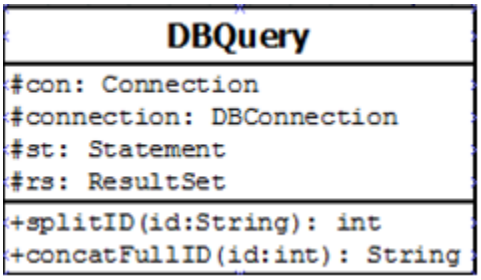


Figura 3.4: Definición de clase DBQuery

el diagrama de la clase **DBQueryActualiza**, dicha clase se encarga de actualizar los registros de la base de datos y hereda de la clase **DBQuery**. La clase cuenta con dos métodos: **updateClient**(), el cual actualiza los datos de un cliente en la base de datos recibiendo como parámetro una instancia de la clase **ObjectClient** misma que contiene los datos actualizados del cliente; de igual manera, el método **updateProduct**() realiza una función similar, pero con objetos de la clase **ObjectProduct**.

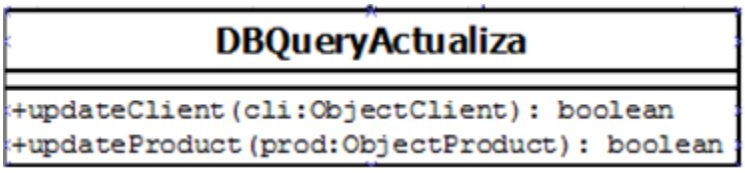


Figura 3.5: Definición de clase DBQueryActualiza

En la figura 3.6 se muestra el diagrama de la clase **DBQueryConsulta**, misma que se encarga de hacer consultas a la base de datos, es decir, su principal función es la

de recuperar datos; hereda de la clase **DBQuery**. La clase cuenta con nueve métodos para obtener cierta información, a continuación se presenta su funcionalidad.

- Método **getClientById**(), que recibe como parámetro el id del cliente y devuelve el registro contenido en la tabla **Cliente** de la base de datos que coincida con dicho id del cliente;

- Método **getProductById**(), que recibe como parámetro el id del producto y devuelve el registro contenido en la base de datos que coincida con el id del producto;

- Método **getClientByName**() que recibe como parámetros el nombre del cliente y de devuelve todos los registros existentes en la base de datos que coincidan y/o contengan en el campo 'nombres' con el parámetro dado;

- Método **getClientByFullName**(), que recibe como parámetros el nombre (o nombres), así como de los apellidos del cliente y devuelve el registro existente en la tabla Cliente que coincida con los parámetros dados;

- El método **getProductByNameTam**(), que recibe como parámetros el nombre y tamaño del producto; y retorna el registro existente en la base de datos que coincida con los parámetros dados;

- Método **getProductByName**(), que recibe como parámetro el nombre del producto y devuelve todos los registros existentes en la base de datos que coincidan con dichos parámetros;

- Método **getAllProducts**(), que devuelve todos los registros de la base de datos correspondientes a la tabla de productos;

- Método **getSaleHistory**(), que devuelve los registros de ventas que han sido realizadas en el mes y año que se especifican como parámetros de la función, es decir, devuelve todos los registros que se realizaron en un periodo mensual;

- Método **getClientHistory**(), que recibe como parámetro el id del cliente y devuelve todos los registros de ventas en la base de datos que estén asociados con el id de cliente, sirve como ayuda para observar patrones en las preferencias de consumo.

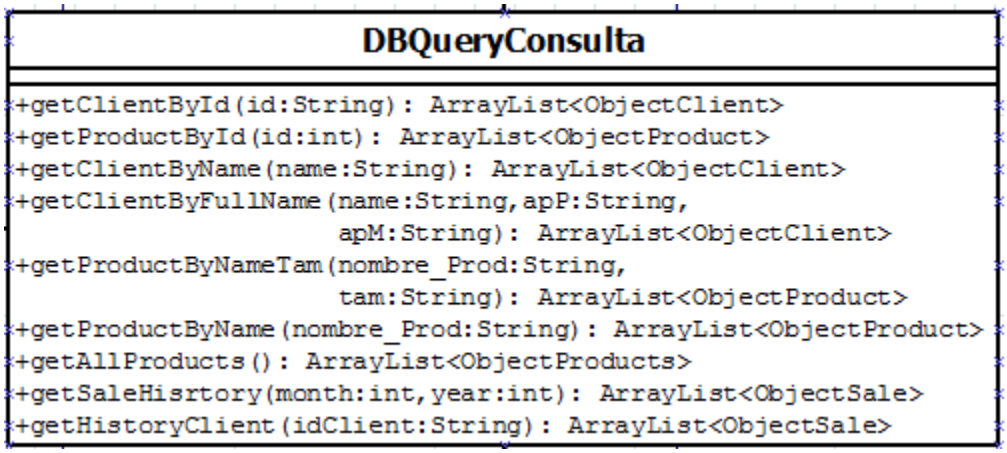


Figura 3.6: Definición de clase DBQueryConsulta

En la figura 3.7 se presenta el diagrama de la clase **DBQueryElimina**, misma que permite eliminar registros dentro de la base de datos y que hereda de la clase **DBQuery**. Esta clase cuenta con dos métodos: el método **deleteClient**(), que recibe como parámetro el id del cliente y elimina de la base de datos el registro que coincida con el parámetro dado en la tabla de clientes; el método **deleteProduct**(), recibe como parámetro el id del producto y de igual manera realiza la eliminación de un registro en la base de datos que coincida con el parámetro dado en la tabla de productos.

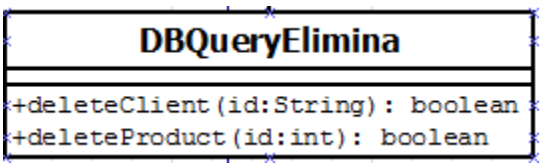


Figura 3.7: Definición de la clase DBQueryElimina

En la figura 3.8 se muestra el diagrama de la clase **DBQueryRegistro**, que permite guardar registros en la base de datos y también hereda de la clase **DBQuery.** Posee

tres métodos: el método **registerClient**(), que recibe como parámetro una instancia de la clase **ObjectClient**, (contiene toda la información de un cliente a registrar) para ser ingresada en la tabla **Cliente**; el método **createProduct**(), que recibe como parámetro una instancia de la clase **ObjectProduct**, la cual contiene toda la información de un producto a registrar, para ser ingresada a la tabla **Productos**; y por último, el método **generateVenta**(), que recibe como parámetro una instancia de la clase **ObjectSale**, la cual contiene toda la información de una venta para ser almacenada en la tabla **Ventas**.

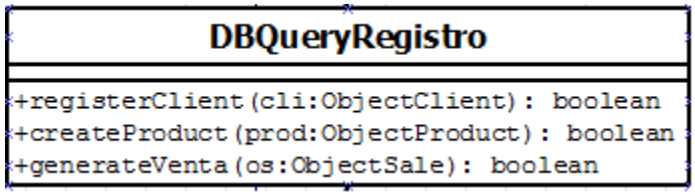


Figura 3.8: Definición de la clase DBQueryRegistra

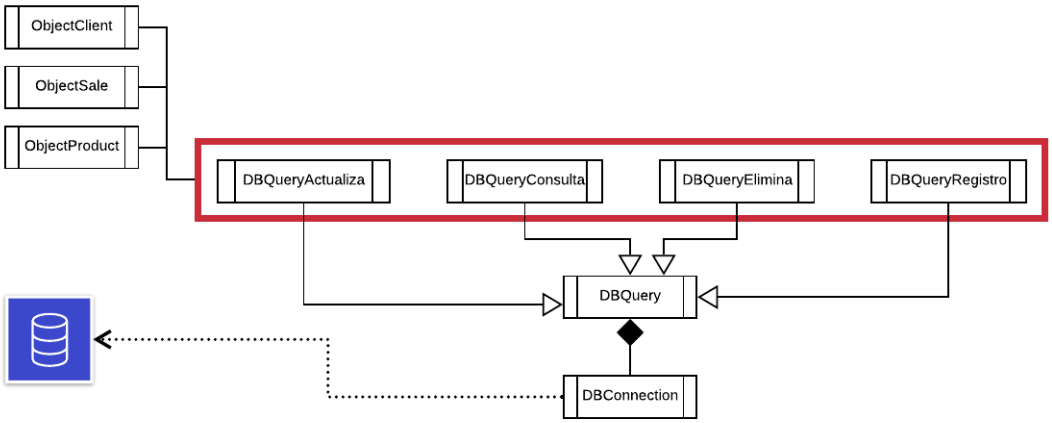


Figura 3.9: Diagrama de clases de la capa back-end

En la figura 3.9 se muestra en un diagrama de clases la forma en cómo se encuentra estructurado el proyecto en su parte lógica para gestión de los datos (Back-End). Como se observa en dicha figura, las clases **DBQueryActualiza**, **DBQueryConsulta**, **DBQueryRegistro** heredan de la clase **DBQuery**. La clase **DBQuery** se asocia con la clase **DBConnection**. Las clases auxiliar **ObjectProduct** y **ObjectClient**,

se asocian con las tres clases diseñadas para actualización, consulta y registro de información en la base de datos.

Por otra parte, la clase auxiliar **ObjectSale** se encuentra asociada sólo con las clases diseñadas para la consulta y registro de información en la base de datos. Estas asociaciones permiten restringir lo que el usuario puede hacer sobre la base de datos. El usuario únicamente puede actualizar, registrar y eliminar información de productos y clientes; y puede consultar y registrar las compras de los clientes. No puede borrar las ventas realizadas.

En resumen, las clases definidas hasta ahora son las que se mantienen en contacto con la base de datos y se encargan de realizar la transacción de la información entre la base de datos y el resto del sistema. Seguida de esta sección se encuentra una clase llamada **LogicProgram**, la cual se encarga de verificar todas las transacciones que se realizan entre el back-end y el front-end del sistema a pesar de que las transacciones finalicen de una manera correcta o incorrecta, los métodos de la clase **LogicProgram**, se encargan de presentar al usuario el estado final de las transacciones libres de excepciones. Esto permite evitar que el usuario presente dudas sobre la estabilidad del sistema. **LogicProgram** funciona como un canal de comunicación entre el back-end y el frontend. En la figura 3.10 se exhibe su diagrama de clase. La clase **LogicProgram**

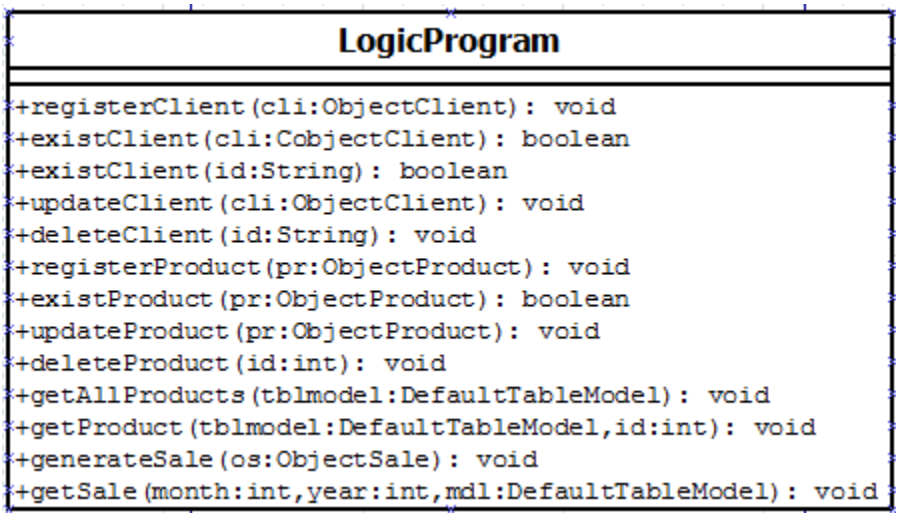


Figura 3.10: Definición de la clase LogicProgram

cuenta con trece métodos que ayudan a cumplir la función de avisar cuando una transacción ha sido realizada con éxito o sin él. A continuación se describe cada uno de ellos.

- El método **registerClient**(), recibe como parámetro una instancia de la clase **ObjectClient** la cual contiene la información para registrar un cliente nuevo en la base de datos, si el proceso ha resultado con éxito o no, despliega una ventana con una breve descripción de lo sucedido;

- El método **existClient**() recibe como parámetro una instancia de la clase **ObjectClient** la cual contiene información de un cliente sobre el cual se necesita saber de su existencia en la base de datos, devuelve un valor booleano **true** en caso de que exista dicho cliente o **false** en caso contrario, éste método sirve como auxiliar en la capa de front-end para asegurar que no existan registros duplicados de un mismo cliente. Cabe mencionar que la búsqueda se hace por medio del nombre y los apellidos del cliente. Se cuenta con un método más con el mismo nombre, **existClient**(), haciendo uso de la sobrecarga de métodos recibe el id de un cliente y devuelve **true** si hay algún registro dentro de la base de datos con dicho id o **false** en caso contrario.

- El método **updateClient**() recibe como parámetro una instancia de la clase **ObjectClient**, misma que contiene valores actualizados del registro de un cliente existente dentro de la base de datos. Si la transacción es realizada con éxito o no despliega una ventana con una breve descripción del estado final de la transacción.

- El método **deleteClient**() recibe como parámetro el id del cliente, elimina de la base de datos aquel registro de cliente que se encuentre asociado a dicho id, si la transacción es realizada con éxito o no despliega una ventana con una breve descripción del estado final de la transacción

- El método **registerProduct**() recibe como parámetro una instancia de la clase **ObjectProduct** la cual contiene información de un nuevo producto para ser

registrado en la base de datos, si la transacción es realizada con éxito o no despliega una ventana con una breve descripción del estado final de la transacción.

- El método **existProduct**() recibe como parámetro una instancia de la clase **ObjectProduct** la cual contiene información de un producto y verifica que dicho producto se encuentre o no dentro de la base de datos, devuelve un valor booleano **true** si dicho producto ha sido encontrado en la base de datos o **false** en caso contrario, éste método sirve como auxiliar en la capa de front-end para asegurar que no existan registros duplicados de un mismo producto, cabe mencionar que la búsqueda se hace por medio del nombre y el tamaño del producto.

- El método **updateProduct**() recibe como parámetro una instancia de la clase **ObjectProduct** el cual contiene información actualizada de algún producto, si la transacción se realiza con éxito o no, el método despliega una ventana con una breve descripción del estado final de la transacción.

- El método **deleteProduct**() recibe como parámetro el id asociado al registro de un producto, mismo que ha de ser eliminado, el método despliega una ventana con un breve resumen del estado final de la transacción.

- El método **getAllProducts**() recibe como parámetro una instancia del a clase **DefaultTableModel** que no es más que el modelo de un control **JTable** , componente de la librería **Swing** proveniente de la capa de front-end, en la cual se insertan todos los registros de productos encontrados en la base de datos, si la transacción se realiza sin éxito el método no agrega nada al modelo de la tabla y despliega un breve resumen del estado final de la transacción. En caso contrario, la tabla se llena con los datos de los productos.

- El método **getProduct**(), recibe como parámetros una instancia de la clase **DefaultTableModel** así como del id de producto, dicho método busca en la base de datos el registro asociado al id proporcionado anteriormente, en caso de hallar el producto, se agrega parte de la información al modelo de la tabla el

cual tiene un formato el cual se explicará mas adelante en apartado XXX de la sección de front-end. Si no se ha encontrado el producto, el método no agrega nada al modelo de la tabla y despliega una ventana con una breve descripción del estado final de la transacción. En caso contrario, la tabla de llena con los datos obtenidos.

- El método **generateSale**() recibe como parámetro una instancia de la clase **ObjectSale** la cual contiene información de una compra que ha realizado el cliente y se registra en la base de datos, si la transacción no es realizada con éxito, el método despliega una breve descripción del estado final de dicha transacción.

- El método **getSale**() ayuda a la capa de front-end a generar un reporte en modo de tabla de las ventas realizadas en un mes y año específico, para poder realizar lo anterior, recibe como parámetros el mes, el año y una instancia de la clase **DefaultTableModel** en el cual se agregan todos los registros de compras que ha generado el cliente que coincidan con los parámetros dados, si no se encuentran registros el método no agrega nada al modelo de la tabla.

Reestructurando el diagrama de la figura 3.9 y agregando la clase **LogicProgram**, el diagrama de clases para la capa de back-end se muestra en la figura 3.11.

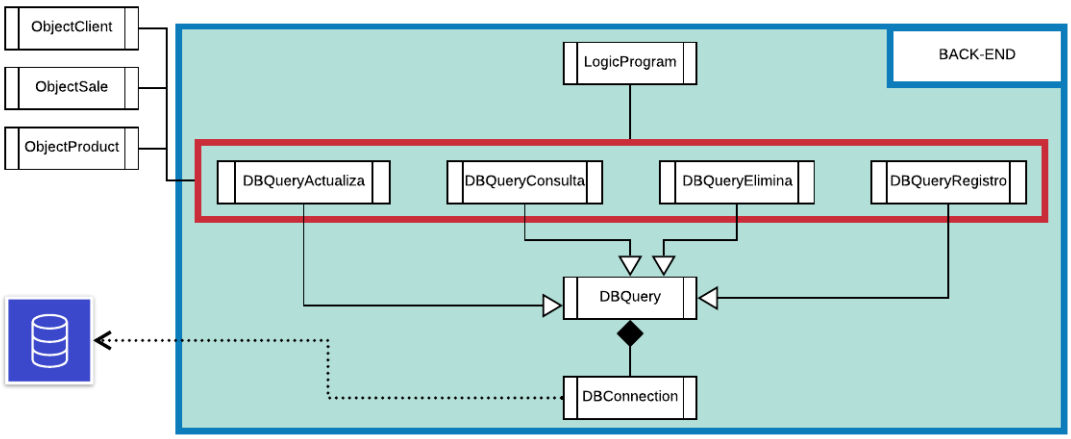


Figura 3.11: Diagrama de clases back-end

El back-end no puede ser utilizado directamente por el usuario, únicamente sirve para gestionar los datos y generar los resultados (o errores) de una manera que la segunda capa, el front-end, pueda usar para presentar gráficamente al usuario final. A continuación se presenta esta parte del sistema desarrollado.

3.3 Front-End

3.3.1 Vista gráfica

Para que un usuario pueda interactuar con el sistema de manera eficaz, simple y sencilla es necesario crear interfaces gráficas de usuario. Dichas interfaces son clases que extienden de la clase **JFrame**, **JInternalFrame** y **JPanel** pertenecientes a la biblioteca **Swing**.

Para el front-end, se generaron diez clases de Java. Cabe mencionar que dichas clases fueron realizadas con ayuda del IDE Netbeans, visto en el apartado 2.2.14.

La primera vista que se presenta al usuario cuando inicia el sistema es generada por la clase llamada **Principal**. Esta última ha de contener otras vistas dentro de ella, es decir, es un contenedor de ventanas. **Principal** hereda de la clase **JFrame** de **Swing**. En la figura 3.12 se muestra la estructura gráfica de la pantalla principal de la aplicación. Con propósitos explicativos, se han agregado las etiquetas para indicar la ubicación de la barra de menús y el escritorio principal.

La ventana principal contiene una barra de menú (de la clase **JMenuBar**) con tres categorías correspondientes a las tres tablas de la base de datos (INDICAR CUÁLES SON); cada menú contiene submenús que ofrecen al usuario una manera sencilla de realizar las operaciones permitidas. El segundo componente es un escritorio virtual (**JDesktopPane**), el cual sirve como contenedor para ventanas internas (**JInternalFrame**) que sólo pueden ser manipuladas dentro del dominio de dicho elemento.

Figura 3.12: Vista principal del sistema

La segunda clase, **VentasView** (figura 3.13), aparece siempre dentro del escritorio virtual, no puede maximizarse, minimizarse o cerrarse, ya que es una vista que tiene más persistencia respecto a las demás y sus dimensiones han sido definidas de manera predeterminada. El objetivo de esta vista es el de realizar ventas, aunque por ergonomía y eficiencia del proceso, posee accesos a otras vistas que más adelante serán explicadas. Dicha vista se compone de una etiqueta que muestra la fecha y hora actual, seguido de una sección para ingresar el id del cliente al que se le va a realizar la venta, seguido de dos botones, uno para consultar una gráfica del consumo personal calculado desde la base de datos y otro para desplegar la vista que permite registrar un nuevo usuario.

VentasView cuenta con dos tablas, la primera que muestra todos los productos existentes dentro de la base de datos, la segunda se muestra vacía y sirve para ir agregando los productos que el cliente ha de solicitar para su actual compra, esta última tabla se va llenando seleccionando un producto de la primera tabla, haciendo clic derecho y seleccionando “agregar”, además de contar con un botón en la parte superior para actualizar la información de dicha tabla. En la segunda tabla de **VentasView** se pueden modificar dos parámetros, la cantidad de productos (el cual sólo acepta números enteros positivos); y el de observaciones el cual permite realizar notas sobre el

Figura 3.13: Clase VentasView

producto. Las notas son útiles para personalizar el pedido del cliente, por ejemplo, se puede agregar una nota para eliminar/agregar ingredientes de una pizza. De acuerdo con el número de productos y el precio que corresponde a cada uno de ellos, el sistema es capaz de calcular el total a pagar por parte del cliente, así como de imprimir una nota (*ticket*) con cierto formato el cual se describe más adelante. También permite quitar uno o más productos de los pedidos, haciendo clic derecho y seleccionando la opción correspondiente.

La siguiente clase, **RegClientView**, consiste en un formulario en el cual se solicita información personal de un cliente nuevo, misma que se manda al back-end para crear el registro del nuevo cliente. En la figura 3.14 se muestra el diseño de dicha interfaz gráfica.

La clase, **UpdateClient** (figura 3.15), se encarga de buscar registros específicos en la base de datos y mostrarlos en una tabla, en dicha tabla se puede seleccionar el registro y haciendo clic derecho se despliega un menú contextual del cual puede eliminarse o modificarse, si se ha de modificar el registro, se reutiliza la clase **RegClietView**,

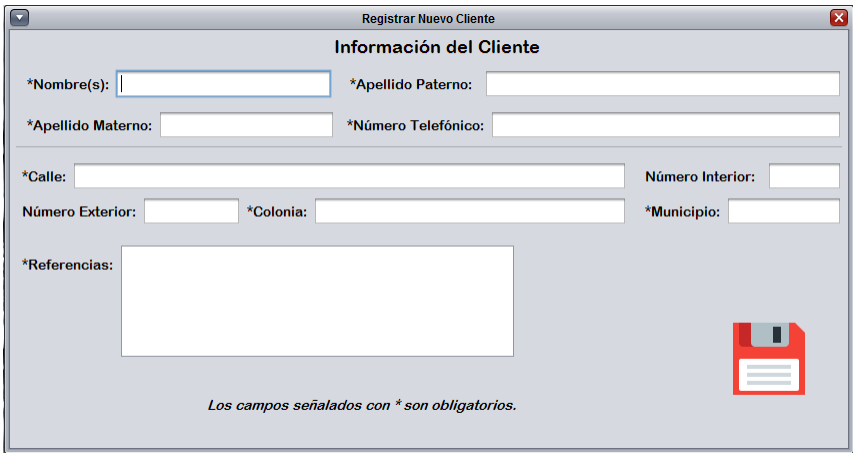


Figura 3.14: Clase RegClientView

llenando los campos con la información del cliente seleccionado y cambiando el título de la ventana como se muestra en la figura 3.16.

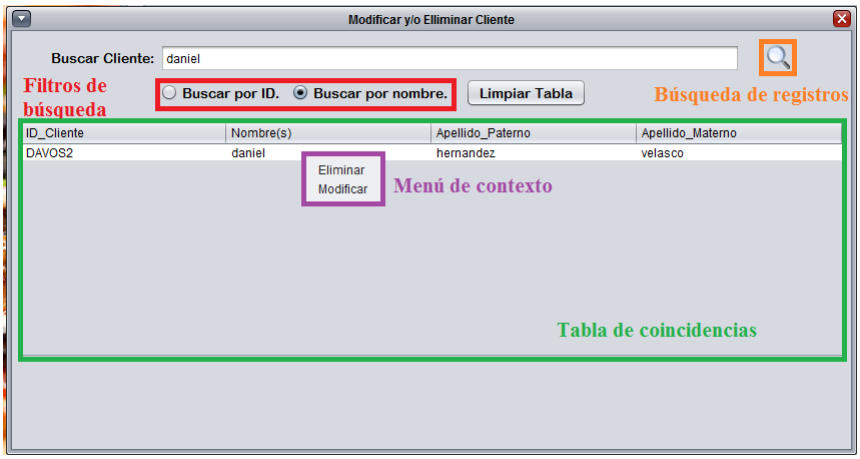


Figura 3.15: Clase UpdateClient

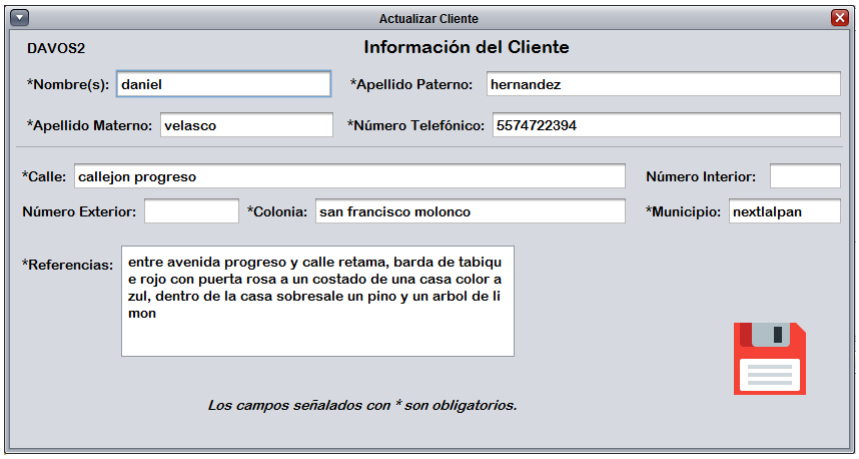


Figura 3.16: Clase RegClientview reutilizada para actualizar registros

La clase, **CreateProduct**, consiste en un formulario en el cual se le solicita al usuario información respecto al nuevo producto que ha de registrarse en la base de datos, misma que se envía al back-end para crear el nuevo registro del producto. En la figura 3.17 se muestra el diseño de dicha interfaz gráfica.

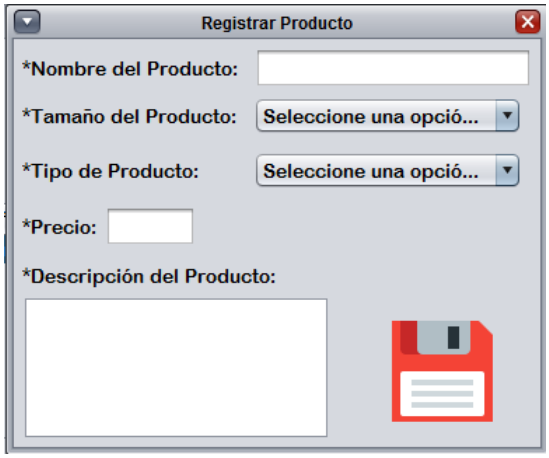


Figura 3.17: Clase CreateProduct

La clase **UpdateProduct**, se encarga de buscar registros específicos en la base de datos y mostrarlos en una tabla, en dicha tabla se puede seleccionar seleccionar el registro y haciendo clic derecho se despliega un menú el cual puede eliminar o modificar el registro según sea la selección del usuario, si el usuario elige modificar

la información de un registro se reutiliza la clase **CreateProduct**, llenando los campos con la información del producto seleccionado y cambiando el título de la ventana como se muestra en la figura 3.18.

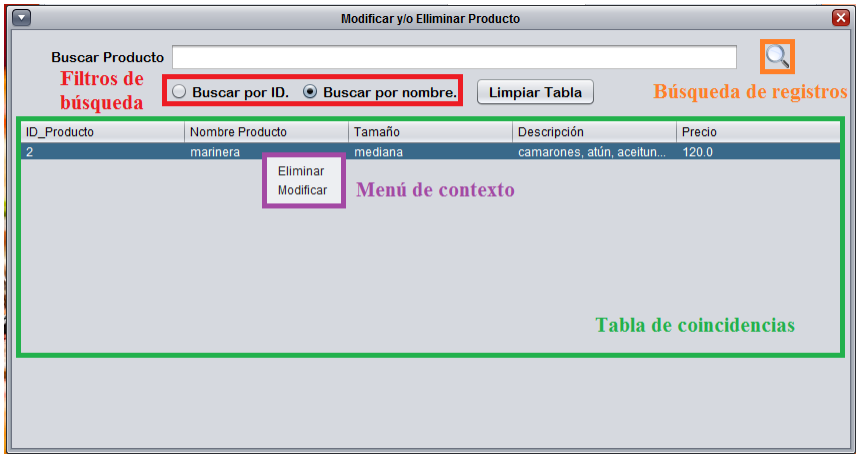


Figura 3.18: Clase UpdateProduct

La clase **HistorialVentas** se encarga de obtener el historial de ventas que la empresa ha realizado, mostrando la información en forma de tabla, el historial se obtiene de acuerdo con el mes y año en que fueron realizadas y calcula el total de ingresos que se generaron en el mes y año proporcionado. Como se puede observar en la figura 3.19, la vista consta de un filtro en el cual se selecciona el mes y año de la consulta, al ser modificados dichos valores el sistema automáticamente recupera la información de la base de datos mostrando los registros encontrados en la tabla.

Figura 3.19: Clase historial ventas

La clase **PieChart**, es una clase que extiende de **JPanel** y crea una gráfica de pastel que representa las tendencias en el consumo de un cliente en específico haciendo uso de la herramienta **JFreeChart**, con información extraída de la base de datos; integrando la gráfica elaborada dentro de un panel mismo que se puede obtener haciendo uso del método **getChartPanel**(). La clase **HistorialCliente** consta de una ventana que integra un Panel, mismo que se obtiene de la clase **PieChart**. De lo anterior, se crea el diagrama de clases de la figura 3.20 que muestra la forma en cómo se encuentra estructurado el proyecto en su parte lógica para la comunicación con el usuario (Front-End).

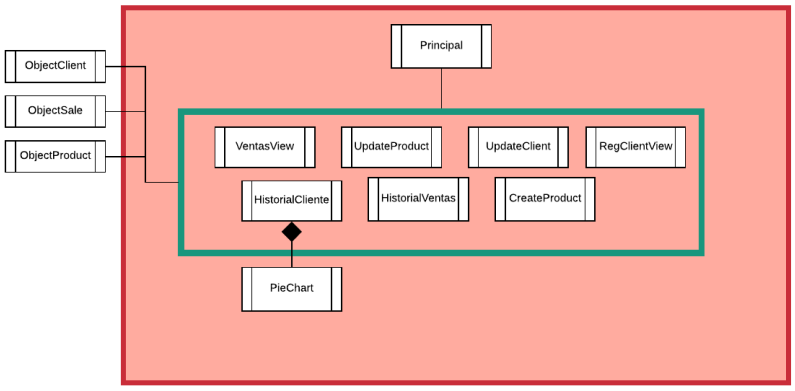


Figura 3.20: Diagrama de clases front-end

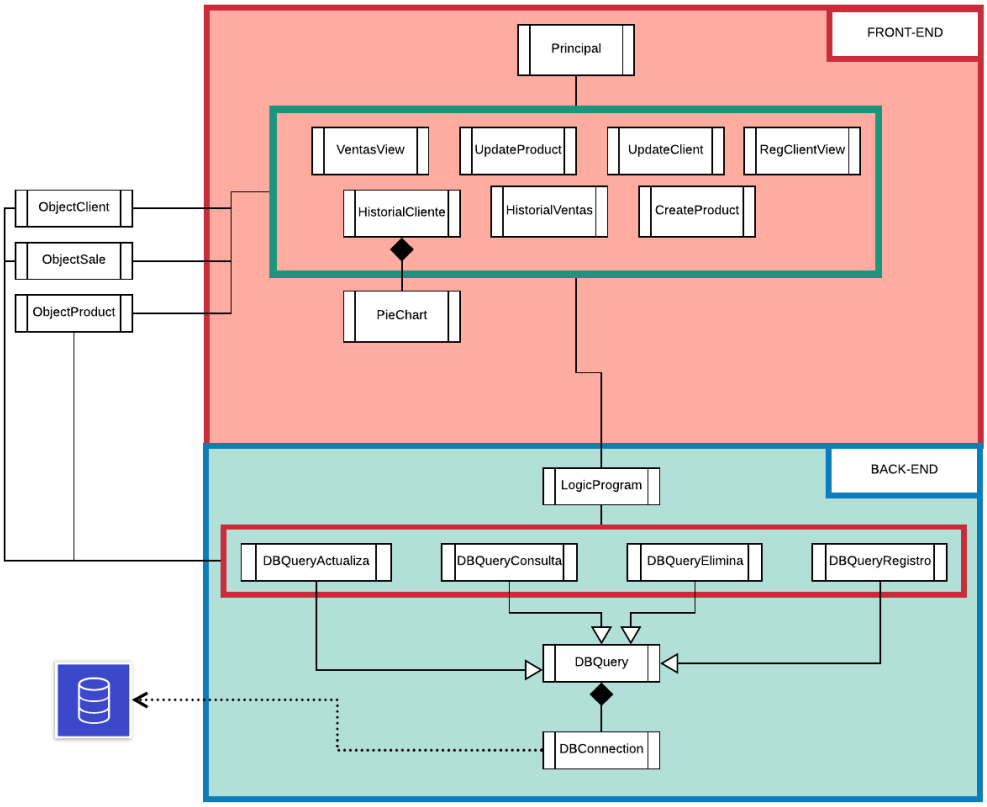


Figura 3.21: Arquitectura completa del sistema

3.4 Arquitectura del sistema

Para obtener una perspectiva completa respecto a la estructura del sistema propuesto en el presente trabajo, en la figura 3.21 se muestra la relación entre las capas back-end y front-end.

Dada la arquitectura de la figura 3.21, se garantiza la funcionalidad del sistema, permitiendo al desarrollador agregar o eliminar módulos de código, de acuerdo con las necesidades futuras de la microempresa.

3.5 Áreas de mejora y proyectos a mediano plazo

El sistema actual en su versión 1.0, aun cuenta con tareas por realizar, de las que destacan: crear conexiones a un servidor de base de datos en la nube, generar factura electrónica, pagos con tarjetas bancarias

3.6 Pruebas realizadas

A continuación se presentan algunas de las pruebas de funcionamiento que se realizaron al sistema para comprobar su correcto funcionamiento.

Agregar un producto Se pretende crear un nuevo producto haciendo uso del sistema, el cual tiene por nombre "hamburguesa de camarón", es para consumo individual de tipo hamburguesa con un costo de 80 pesos y su descripción es: "una hamburguesa con camarones". En la figura 3.22 se muestra la vista que es usada con el fin anterior logrando registrar dicho producto en la base de datos

Figura 3.22: Registro de Productos

Consultar Producto

Se necesita realizar tres consultas, la primera debe de mostrar todos los productos que se encuentren en la base de datos del sistema, la segunda debe de mostrar todos los productos que se encuentren en la base de datos del sistema que contengan un nombre en específico y la última debe de mostrar el producto que se encuentre en la base de datos del sistema que corresponda a un identificador de producto en específico. Para el primer caso, sólo basta con dar click en el boton con el icono de lupa o presionar la tecla de enter en la caja de texto sin ingresar informacion dentro de ella, tal como se muestra en la figura 3.23.

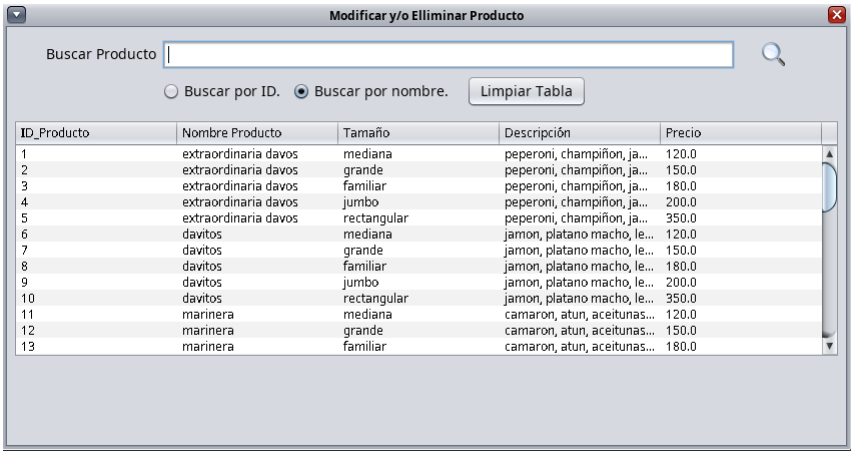


Figura 3.23: Consulta todos los productos

Para el segundo caso, es necesario ingresar un nombre o parte del nombre del producto que se desea buscar dentro de la base de datos, seleccionar la opción "Buscar por nombre" y dar click en el boton con el icono de lupa o presionar la tecla enter en la caja de texto, tal como se muestra en la figura 3.24.

Finalmente, para el tercer caso, es necesario ingresar el identificador del producto en particular que se desea buscar dentro de la base de datos del sistema, seleccionar la opción "Buscar por ID" y dar click en el boton con el icono de lupa o presionar la tecla enter en la caja de texto, tal como se muestra en la figura 3.25.

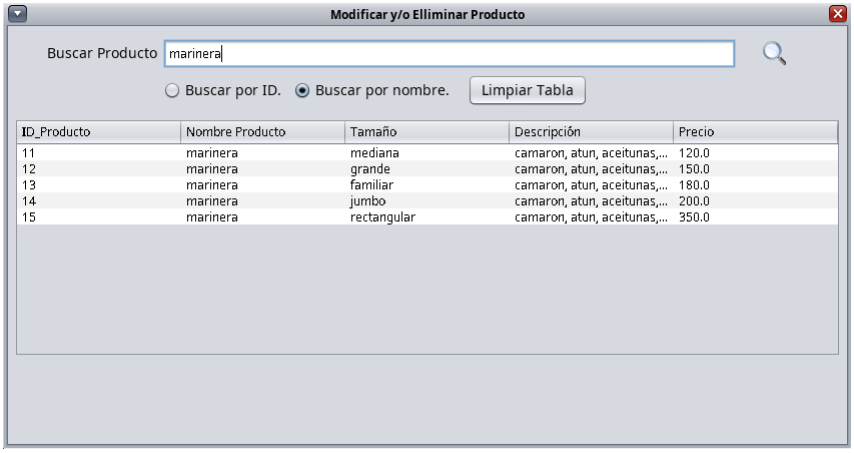


Figura 3.24: Consulta un producto por su nombre

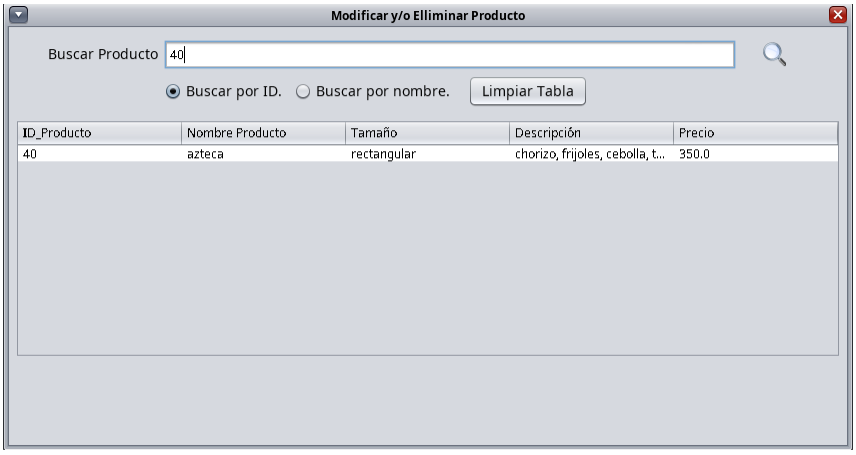


Figura 3.25: Consulta un producto por ID

Editar un producto

Se pretende editar la información de un producto en específico, por lo que se hace uso de cualquier tipo de consulta visto anteriormente, se selecciona el producto deseado en la tabla de resultados con click izquierdo, posteriormente se hace click derecho sobre el producto seleccionado para mostrar un pequeño menú, selecciona la opción editar y se abre una nueva ventana con la información de dicho producto. Se realizan las modificaciones necesarias en esta vista y se da click en el boton con el icono de un disquete para guardar los cambios, tal como se muestra en la figura 3.26

Eliminar un producto

Se pretende eliminar un registro en específico de la base de datos del sistema,

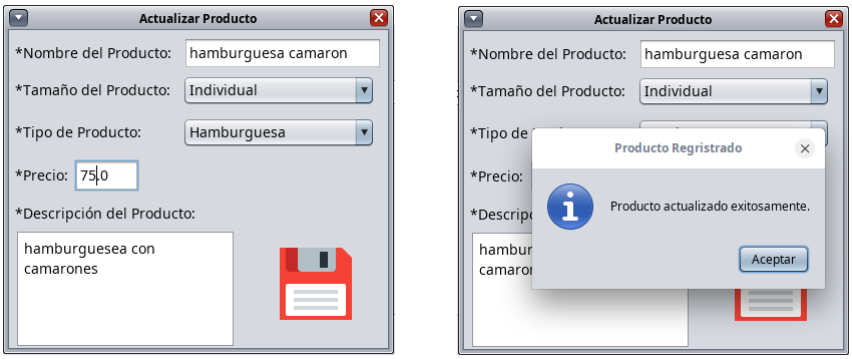


Figura 3.26: Edición de productos

para esto, se hace uso de cualquier tipo de consulta visto anteriormente, se selecciona el producto deseado en la tabla de resultados con click izquierdo, posteriormente se hace click derecho sobre el producto seleccionado para mostrar un pequeño menú, se selecciona la opción eliminar, que muestra un diálogo de confirmación, mismo que al dar clic en la opción si elimina el registro de ese producto, En la figura 3.27 se muestra claramente un ejemplo de esta operación.

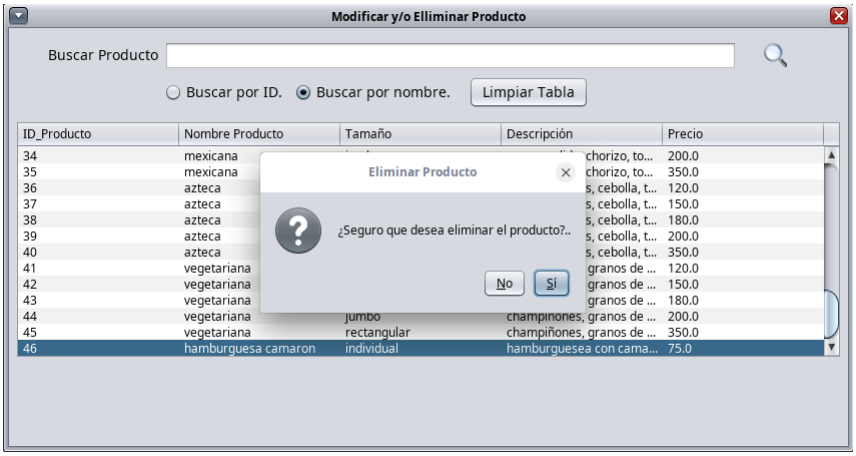


Figura 3.27: Eliminar un producto

Agregar cliente

Se necesita crear un nuevo registro dentro de la base de datos del sistema para almacenar un nuevo cliente, su nombre es Evelin López Juarez con número telefónico 5598360245 y domicilio en avenida insurjentes número interior 16, numero exterior 85

en Lomas de Sotelo, Benito Juarez. En las figuras 3.28 y 3.29 se muestra la vista que cumple con este objetivo, registrando exitosamente el nuevo cliente, mostrando en un cuadro de dialogo el identificador de cliente asignado por la base de datos del sistema, así como la información proporcionada anteriormente.

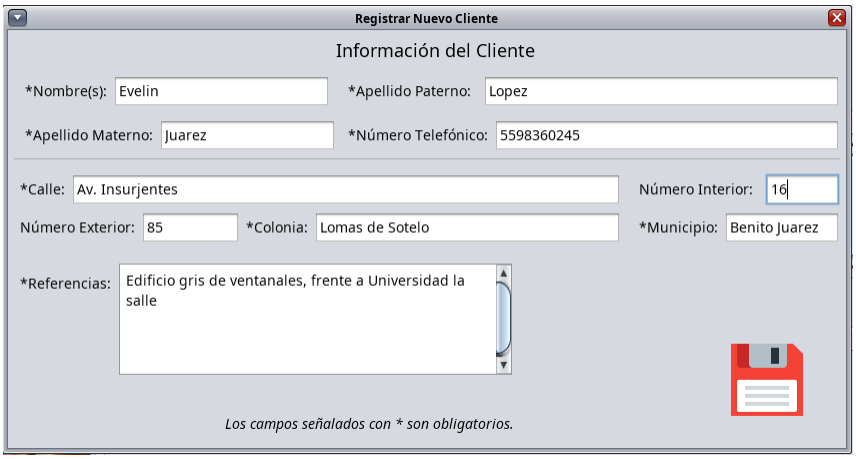


Figura 3.28: Agregar Cliente

Figura 3.29: Resultado de agregar un cliente

Consultar un cliente

Se necesita realizar tres consultas, la primera debe de mostrar todos los clientes que se encuentren en la base de datos del sistema, la segunda debe de mostrar todos los clientes que se encuentren en la base de datos del sistema que contengan un nombre en específico y la última debe de mostrar el cliente que se encuentre en la base de datos del sistema que corresponda a un identificador de cliente en específico. Para el primer caso, sólo basta con dar click en el boton con el icono de lupa o presionar la tecla de enter en la caja de texto sin ingresar informacion dentro de ella, tal como se muestra en la figura 3.30.

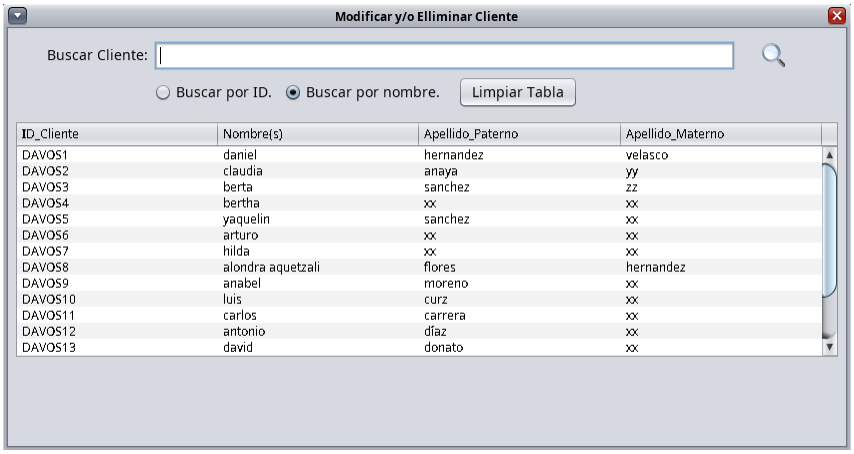


Figura 3.30: Consulta todos los clientes

Para el segundo caso, es necesario ingresar un nombre o parte del nombre del cliente que se desea buscar dentro de la base de datos, seleccionar la opción "Buscar por nombre" y dar click en el boton con el icono de lupa o presionar la tecla enter en la caja de texto, tal como se muestra en la figura 3.31.

Finalmente, para el tercer caso, es necesario ingresar el identificador del cliente en particular que se desea buscar dentro de la base de datos del sistema, seleccionar la opción "Buscar por ID" y dar click en el boton con el icono de lupa o presionar la tecla enter en la caja de texto, tal como se muestra en la figura 3.32.

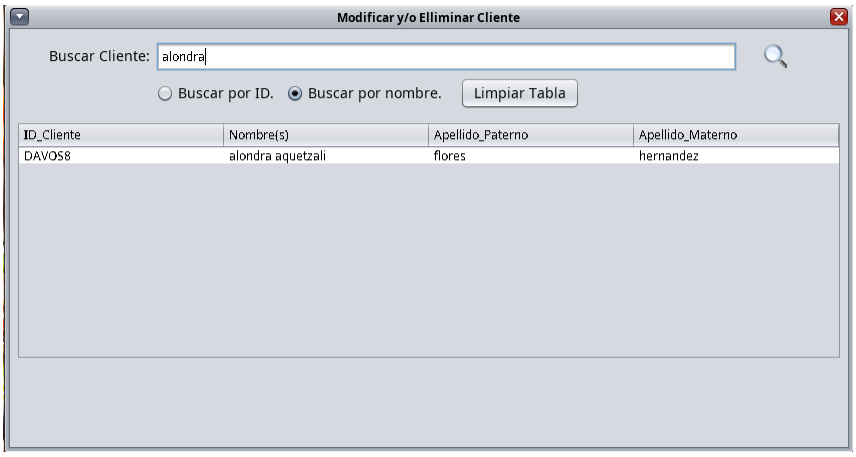


Figura 3.31: Consulta un cliente por su nombre

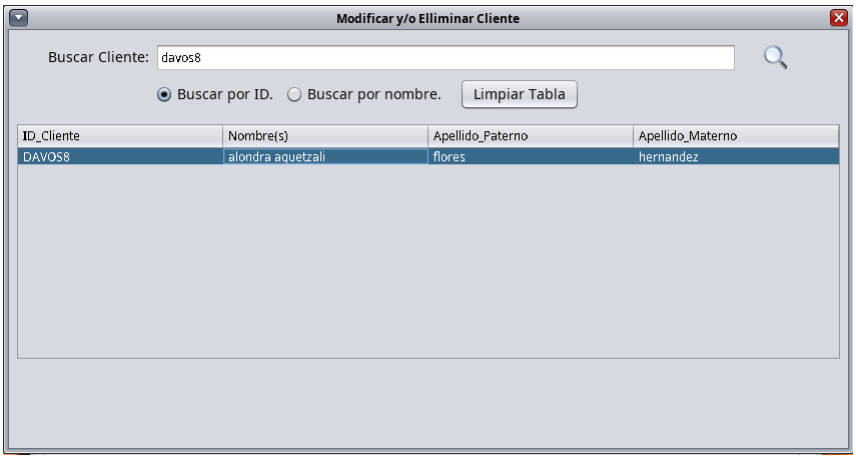


Figura 3.32: Consulta un cliente por su ID

Editar un cliente

Se pretende editar la información de un cliente en específico, por lo que se hace uso de cualquier tipo de consulta visto anteriormente, se selecciona el cliente deseado en la tabla de resultados con click izquierdo, posteriormente se hace click derecho sobre el cliente seleccionado para mostrar un pequeño menú, selecciona la opción editar y se abre una nueva ventana con la información de dicho cliente. Se realizan las modificaciones necesarias en esta vista y se da click en el boton con el icono de un disquete para guardar los cambios, tal como se muestra en la figura 3.33

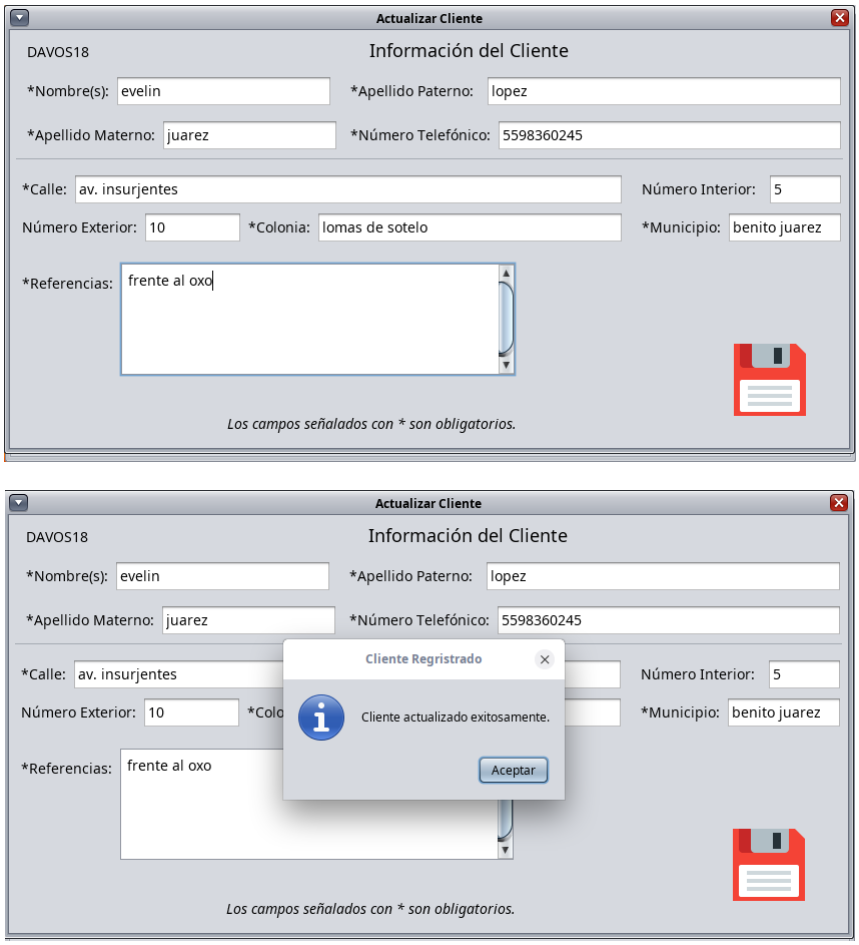


Figura 3.33: Editar un cliente

Eliminar un cliente

Se pretende eliminar un registro en específico de un cliente en la base de datos del sistema, para esto, se hace uso de cualquier tipo de consulta visto anteriormente, se selecciona el cliente deseado en la tabla de resultados con click izquierdo, posteriormente se hace click derecho sobre el producto seleccionado para mostrar un pequeño menú, se selecciona la opción eliminar, que muestra un diálogo de confirmación, mismo que al dar clic en la opción si elimina el registro de ese cliente, En la figura 3.34 se muestra claramente un ejemplo de esta operación.

Realizar una venta

Se pretende realizar una venta al usuario DAVOS1, el cual requiere de dos pizzas

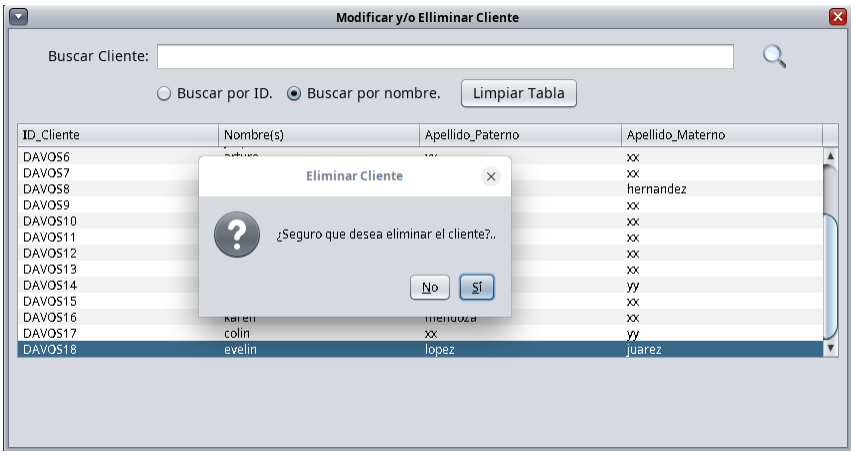


Figura 3.34: Eliminar un cliente

tamaño rectangular, la primera de sabor hawaiana y mexicana; la segunda de sabor marinera y azteca.

Para realizar esta operación, primero se ingresa el identificador del cliente que se encuentra realizando una compra, posteriormente se debe seleccionar de la tabla de productos, el producto deseado por el cliente, así como su tamaño y se agrega a la tabla de pedido haciendo click derecho sobre el producto seleccionado y eligiendo la opción agregar.

En la columna de observaciones se ingresan las peticiones del cliente, en este caso que la pizza de sabor hawaiana y mexicana deben de contener la mitad de sabor mexicana y azteca respectivamente. En las figuras 3.35 y 3.36 se muestran las operaciones anteriores.

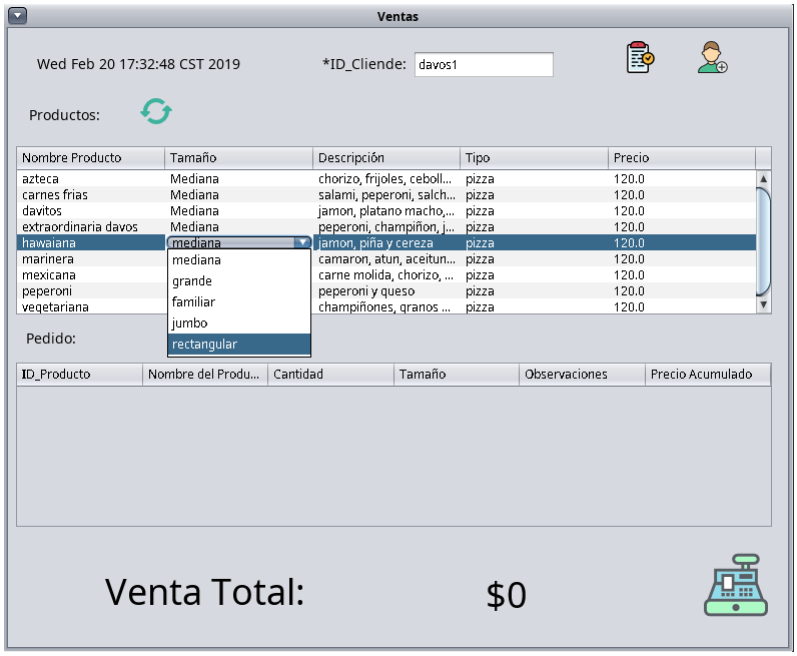


Figura 3.35: Realizar una Venta (Selección de producto)

Una vez confirmados los productos solicitados por el cliente, ese da click en el botón con el icono de caja registradora para cerrar la venta, mismo que despliega un cuadro de dialogo y proporciona un ticket de con la información de la compra, tal como se muestra en la figura 3.37.

Figura 3.36: Realizar una venta (modificación de pedido)

Consultar historial de cliente

Para consultar el historial de un cliente en específico, DAVOS1, en la vista de ventas, basta con ingresar el id del cliente y hacer click en el botón con el icono de una tabla de registros para mostrar una gráfica de pastel que muestra las tendencias en su consumo basado en las compras encontradas en la base de datos del sistema que se encuentren asociadas con el identificador de cliente, tal como se muestra en la figura 3.38. En el caso del cliente con identificador DAVOS1, su tendencia de consumo se inclina a los sabores marinera y hawaiana por igual.

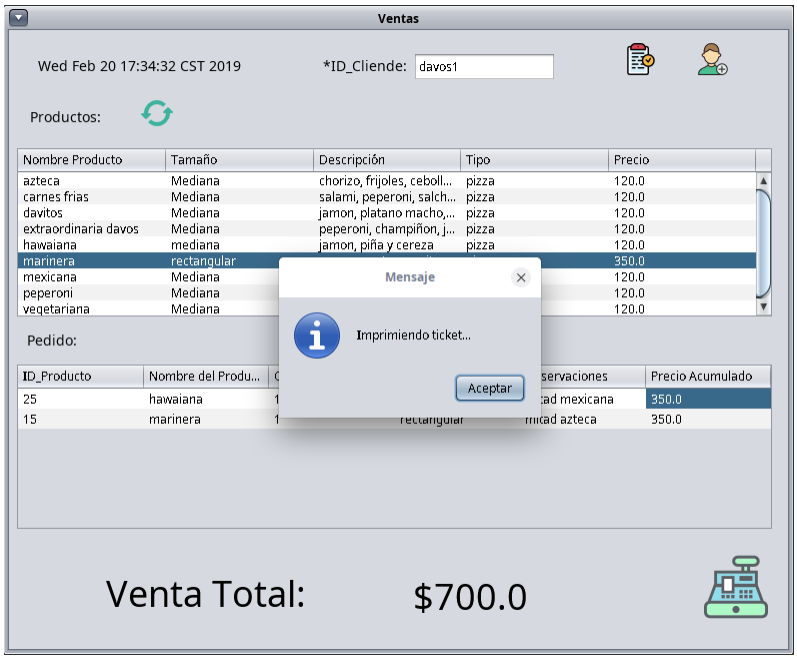


Figura 3.37: Realizar venta (Cierre de venta)

Figura 3.38: Consultar el historial de cliente

CONCLUSIONES

Uno de los procesos más comunes en todas las empresas, y en particular de las pequeñas y medianas (PyME), es llevar el control de ventas al menudeo. Muchas PyMes en México llevan un seguimiento de sus procesos administrativos y de contabilidad de una manera manual, es decir, sus empleados anotan en papel las ventas, pedidos y compra de insumos, o en el mejor de los casos usan hojas de cálculo para este seguimiento. Estas prácticas, aunque son funcionales, no son adecuadas actualmente.

En esta tesis, se ofrece una solución al problema de control de ventas para un pequeño negocio de Pizzas. Para ello, se desarrolló en Java un sistema informático que realiza las funciones básicas de un punto de venta. El sistema fue programado en el lenguaje de programación Java. Se puede afirmar que se cumplieron con todos los objetivos planteados al inicio de este trabajo.

A diferencia de las opciones comerciales para punto de venta, el sistema desarrollado puede ser escalado para soportar más funciones, gracias a la arquitectura usada, y al paradigma orientado a objetos empleado para su programación. Una de las funcionalidades diferenciadoras del sistema, es la de poder visualizar cuáles son los productos que más consume un cliente. Esta característica del sistema podría ser explotada más adelante por el dueño del negocio.

Aunque el sistema está personalizado para una empresa que se dedica a ventas de Pizzas, el sistema pueden ser adaptado para cualquier otro tipo empresa que requiera punto de ventas.

Durante el desarrollo del sistema de punto de venta, se encontraron varias

dificultades. La primera, y la más compleja, fue el entender las necesidades del negocio. Otra dificultad que se tuvo fue la de satisfacer al cliente con respecto a la interfaz gráfica. Aunque se le presentaron algunos prototipos de interfaces (conocidos como Mockups), al momento de presentarle el sistema el cliente decidió cambiar algunas interfaces. Para evitar o minimizar esto, es necesario tener pláticas con el cliente y con los posibles usuarios del sistema. Esto evitará que hayan muchos cambios drásticos cuando el sistema ya esté en etapas más avanzadas de desarrollo.

Uno de los problemas técnicos más relevantes fue la de utilizar la impresora de tickets con el sistema. Esto debido a que el sistema operativo donde se ejecuta (Linux Ubuntu) no reconocía la impresora. Se tuvo que descargar controladores genéricos y configurar manualmente el dispositivo. Se recomienda tener en cuenta esto desde el inicio, para poder elegir una impresora compatible con el sistema operativo y evitar configuraciones manuales.

En la parte de la programación del sistema, el manejo de tablas en Java Swing fue lo que más consumió tiempo de desarrollo. Se recomienda utilizar otra biblioteca más moderna, como JavaFx para facilitar el desarrollo de las interfaces gráficas.

Como trabajo futuro y mejoras al sistema desarrollado, se plantea la integración de facturación electrónica, notificaciones a los clientes mediante mensajes de texto, y la posibilidad de generar una aplicación para Android y sistema iOs para poder realizar las ventas desde un dispositivo móvil.

Referencias

[1] *Informaticos Generalitat Valenciana. Grupos a Y B. Temario Bloque Especifico Volumen I.e-book.* MAD-Eduforma.

[2] Abenza, P. *Comenzando a programar con JAVA*. Universidad Miguel Hernández, 2015.

[3] Belmonte Fernández, O., Granell Canut, C., and Erdozain Navarro, M. d. C. *Desarrollo de proyectos informáticos con tecnología Java*.

[4] Catherine, M. R. *Bases de datos*, primera ed. McGraw-Hill, México, D.F., 2009.

[5] Date, C. J. *Introducción a los sistemas de bases de datos*, séptima ed. Ra-Ma, México, 2001.

[6] Deitel, P. J., and M., D. H. *Cómo programar en Java*, séptima ed. Pearson Educación, 2008.

[7] DÉLÉCHAMP, F., and LAUGIÉ, H. *Java y Eclipse: Desarrolle una aplicación con Java y Eclipse*. Expert IT. Ediciones ENI, 2016.

[8] Dobson, R. *Beginning SQL Server 2005 Express Database Applications with Visual Basic Express and Visual Web Developer Express: From Novice to Professional*. Apress, 2006.

[9] Downey, A. B. *Pensando la computación como un científico (con Java)*. Los Polvorines: Universidad Nacional de General Sarmiento, 2012.

[10] Durán, F., Gutiérrez, F., and Pimentel, E. *Programación orientada a objetos con Java*. Thomson Paraninfo, 2007.

[11] Fernández, H., and Ediciones, E. *Programación orientada a objetos con Java*. ECOE EDICIONES, 2012.

[12] Fischer, P. *Introduction to Graphical User Interfaces with Java Swing*. Addison-Wesley, 2005.

[13] Gaona, A. *Introduccion Al Desarrollo de Programas Con Java*. Prensas de ciencias. Unam, 2007.

[14] García de Jalón, J., Rodríguez, J. I., Mingo, I., Imaz, A., Brazález, A., Larzabal, A., Calleja, J., and García, J. Aprenda Java como si estuviera en primero. Tech. rep., Universidad de Navarra, San Sebastián, 2000.

[15] Groussard, T. *Java Enterprise Edition: Desarrollo de aplicaciones web con JEE 6*. Recursos informáticos. Ediciones Eni, 2010.

[16] Horstmann, C. *Java Concepts for AP* Computer Science*, quinta edi ed. Jhon Wiley & Sons, Inc., 2008.

[17] Joyanes Aguilar, L. *Fundamentos generales de programación*, primera ed ed. McGraw-Hill, 2013.

[18] Joyanes Aguilar, L., and Martínez Zahonero, I. *Programación en Java 6: Algoritmos, programación orientada a objetos e interfaz gráfica de usuarios*, primera ed. McGraw-Hill, México, D.F., 2011.

[19] Kedar, S. *GUI and Database Management*. Technical Publications, 2009.

[20] Kendal, S. *Object Oriented Programing using Java*, frist edit ed. Simon Kendal & Ventus Publishing ApS, 2009.

[21] Limited ITIL Education, S. *Introduction to Database Systems*.

[22] Luque Ruíz, I., Gómez Nieto, M. Á., López Espinosa, E., and Cerruela García, G. *Bases de datos desde Chen hasta Codd*, primera ed. Ra-Ma, 2001.

[23] Millán, M. E. *Fundamentos de bases de datos*, primera ed ed. Progama Editorial Universidad del Valle, Cali, Colombia, 2012.

[24] Ortega Arjona, J. L. Notas de Introducción al Leguaje de Programación Java. Tech. rep., Departamento de Matemáticas Facultad dee Ciencias UNAM, 2004.

[25] Ramos, D., Noriega, R., Laínez, J., Durango, A., and Academy, I. *Curso de Ingeniería de Software: 2ª Edición*. CreateSpace Independent Publishing Platform, 2017.

[26] Schildt, H. *Java: The Complete Reference*, seventh ed ed. McGraw-Hill, 2007.

[27] Silberschatz, A., Kort, H. F., and Sudarsham, S. *Database System Concepts*, sixth edit ed. McGraw-Hill, New York, Unit States of America, 2006.

[28] Sumathi, S., and Esakkirajan, S. *Fundamentals of Relational Database Management Systems*. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2007.

[29] Thalheim, B. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer Berlin Heidelberg, 2013.

[30] Vicente, I., García, C., and Conte, V. *Manual imprescindible de Java 2 v5.0*. Manual imprescindible. Anaya Multimedia, 2005.

[31] W3Schools. Sql inner join keyword. https://www.w3schools.com/sql/sql_join_inner.asp, Consultado en Enero 2019.

[32] W3Schools. Sql joins. http:/www.w3schools.com/sql/sql_join.asp, Consultado en Enero 2019.

[33] W3Schools. Sql left join keyword. https://www.w3schools.com/sql/sql_join_left.asp, Consultado en Enero 2019.

[34] W3Schools. Sql right join keyword. https://www.w3schools.com/sql/sql_join_right.asp, Consultado en Enero 2019.