Elizcom s.a.s
copyright

ISBN: 978-958-8801-76-6

Principios de Programación Orientada a Objetos • Sonia Jaramillo Valbuena • Sergio Augusto Cardona Torres

Elizcom s.a.s
copyright

2018

Principios de PROGRAMACIÓN Orientada a Objetos

Sonia Jaramillo Valbuena
Sergio Augusto Cardona Torres

2018

Principios de programación orientada a objetos

Sonia Jaramillo Valbuena
Sergio Augusto Cardona Torres

Armenia - Quindío

Principios de programación orientada a objetos

Sonia Jaramillo Valbuena
Adscrito al
Programa de Ingeniería de Sistemas y Computación
Facultad de Ingeniería
Universidad del Quindío

Sergio Augusto Cardona Torres
Adscrito al
Programa de Ingeniería de Sistemas y Computación
Facultad de Ingeniería
Universidad del Quindío

Principios de programación orientada a objetos

No está permitida la reproducción total o parcial
de esta obra, ni su tratamiento o transmisión por
cualquier método sin autorización escrita del editor.

Derechos reservados

ISBN: 978-958-8801-76-6

ELIZCOM SAS
Armenia, Quindío – Colombia
2018

Contenido

PREFACIO

Este libro ofrece a los lectores un primer acercamiento a la programación orientada a objetos. Este paradigma se fundamenta en la construcción de aplicaciones basadas en objetos que están interrelacionados, cada uno de los cuales, tiene una determinada identidad, un estado y un comportamiento.

La unidad I, denominada Fundamentos de programación orientada a objetos, en primer lugar, hace una introducción a conceptos básicos de programación y algoritmia tales como tipos de datos primitivos, tipos de datos por valor y por referencia, variables (declaración, inicialización, nomenclatura), constantes, comentarios, operadores (aritméticos, relacionales y lógicos), casting, métodos (con/ sin retorno, con/sin parámetros), llamado de métodos y paso de argumentos. Luego de esta parte introductoria se aborda el tema de la programación orientada a objetos. Aquí se describen los pasos para resolver un problema y se retoman los conceptos de creación e invocación de métodos, expresiones, tipos de datos. Además, se incorporan temas como instanciación, visibilidad, relaciones entre clases, documentación javadoc, pruebas y paquetes. En este capítulo se interactua con el ambiente de desarrollo Eclipse para la contrucción de interfaces gráficas. La unidad II, aborda el tema de expresiones y estructuras de decisión. Se analizan las estructuras if y switch. Además, conocen nuevos elementos de modelado tales como constantes y tipos enumerados. La unidad III, analiza el tema de estructuras contenedoras de tamaño fijo y de tamaño variable.

Este libro retoma algunos elementos del Proyecto Cupi2, de la Universidad de los Andes [1]. Este proyecto se consolida como una solución integral al problema de enseñar y aprender a programar. Entre ellos se destacan: contrato de un método, separación de la lógica y la interfaz, mediante el uso de paquetes. El tema de interfaz gráfica se trabaja desde la primera unidad. Las interfaces se construyen mediante JavaFx.

El código construido dentro del libro retoma algunos ejemplos y definiciones presentadas por lo autores en libros previos tales como [2][3][4]. Además, incorpora el uso de buenas prácticas de programación. Entre ellas se destaca: usar un único retorno por método, construir métodos cortos (definiendo claramente en el contrato del método), realizar una codificación legible y entendible por cualquier programador, utilizar llaves en las estructuras de decisión y ciclos, declarar valores literales como constantes, reutilizar código (modularizar) y usar los patrones de recorrido: parcial, total y doble recorrido [1][2][3]**.**

1. Fundamentos de Programación orientada a objetos

Objetivos Pedagógicos

Al final de este nivel el lector estará en capacidad de:

- Conocer los principios básicos de la Programación Orientada a Objetos
- Identificar y seguir las etapas para la solución de un problema
- Resolver un problema haciendo uso de un programa de computador
- Manejar los conceptos de creación e invocación de métodos, expresiones, tipos de datos e instanciación.
- Interactuar con el ambiente de desarrollo Eclipse
- Crear interfaces gráficas sencillas mediante Java FX

1. FUNDAMENTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS

1.1. Introducción

La programación orientada a objetos, POO, es un paradigma de programación que pretende simular el mundo real haciendo uso de objetos. Este paradigma le da la posibilidad al programador de construir una aplicación basado en abstracciones de alto nivel, a las que se les denomina objetos, siendos éstos cercanos a la realidad de la gente. Este tipo de lenguajes proporciona importantes características, que permiten mejorar la calidad del software, tales como herencia, polimorfismo, encapsulado, cohesión y abstracción [2][3][4].

Este capítulo se estructura de la siguiente forma. En primer lugar, presenta conceptos básicos del lenguaje, entre ellos: identificadores, tipos de datos, operadores, expresiones y casting. Luego incorpora conceptos referentes a Programación orientada a objetos, tales como clases, métodos, diagramas de clases, separación de la lógica e interfaz.

1.1. Conceptos introductorios del lenguaje Java

1.1.1. Identificadores

Todos los datos e instrucciones de un programa son almacenados en la memoria. Para poder tener acceso a las diferentes posiciones de memoria, es necesario hacer uso de identificadores. Un identificador es un nombre que puede darse a una variable, constante (variable con un valor inmutable durante la ejecución de un programa) o a cualquier otro elemento que necesite nombrarse. Existen diversas reglas para el uso de identificadores:

- Tener en cuenta la sensibilidad al tamaño, es decir, existe distinción entre mayúscula y minúscula, por ejemplo, suma es diferente de sumA.
- No puede contener espacios en blanco. Si se quiere escribir cantidad hijos, se escribiría cantidadHijos.
- Debe iniciar con letra Java (A-Z, a-z, $ y _)
- Puede contener dígitos (0-9).
- Si está formado por más de una palabra, las palabras adicionales iniciarán con mayúscula, cantidadHijos, edadPromedioPersonas.
- Si es una constante se escribe en maýuscula
- No usar caracteres propios del castellano (tilde, ni la letra ñ) [2][3][4].

Ejemplos de identificadores válidos son:

varX PI MAX_NUM dirección

PI y MAX_NUM son constantes.

1.1.2.Tipos de datos primitivos

En Java, a cada variable o constante se le debe asociar un tipo de dato. El tipo define el rango de valores que puede tomar mientras el programa se encuentra en ejecución. Los tipos de datos primitivos permitidos por Java son los enteros (**byte**, **short**, **int**, **long**), reales (**float**, **double**), tipos de dato caracter (**char**) y booleanos (**boolean**) [2][3][4].

Los **enteros** representan un intervalo de los números enteros (sin considerar decimales), ver Tabla 1.

Nombre	Rango de valores	Tamaño en bits	Declaración
byte	-128 a 127	8	**byte** var1=4;
short	-32768 a 32767	16	**short** var2
int	-2147483648 a 2147483647	32	**int** var3;
long	-9223372036854775808 a 9223372036854775807	64	**long** var4;

Tabla 1 Tipos de datos enteros

Algunos ejemplos de variables que tienen naturaleza de tipo entero son: la edad de una persona, la cantidad de hijos de una persona, la cantidad de equipos en una sala de computadores. Aunque existe 4 diferentes tipos de datos enteros (**byte** , **short**, **int** y **long** durante este libro por sencillez solo se utilizará únicamente tipo de dato int.

En Java, cuando se crea una variable de tipo de dato numérico el valor por defecto con el que se inicializa es 0. El lector debe tener precaución cuando trabaja con lenguajes tales como C++, dado que no ocurre lo mismo. Se sugiere, por facilidad de migración a otros lenguajes, que cuando declare una variable la inicialice con 0. Para inicializar una variable se hace uso del operador igual (=), que permite cambiar fijar el valor de la variable al valor deseado [3][4].

Ejemplo:

```
double promedio=0;
```

Los **reales** aceptan rangos de valores con parte entera y decimal, ver Tabla 2. Los datos soportados son float y double. Para los ejemplos de este libro solo se utiliza tipo de dato double. Este tipo de dato permite desarrollar completamente los objetivos del curso sin necesidad de estar haciendo conversiones forzosas, lo que le agregaría mayor dificultad al lector [2][3][4].

Nombre	Rango de valores	Tamaño en bits	Declaración
float	3,4E-38 a 3,4E38	32	**float** num1 = 2.7182f;
double	1.7E-308 a 1.7E308	64	**double** num2 = 2.125d;

Tabla 2 Tipos de datos reales

El tipo de dato **caracter**, usa la codificación Unicode. Esta codificación permite usar los caracteres de todos los idiomas. Se caracteriza porque sus primero 127 caracteres corresponden a los mismos caracteres del código ASCII. Un tipo de dato caracter corresponde a un único carácter. El valor de inicialización debe ir entre comillas simples[5][6].

Nombre	Rango de valores	Tamaño en bits	Declaración
char	0 a 65536 (Caracteres alfanuméricos)	16	**char** letra = 'a';

Tabla 3 Tipo de dato caracter

En Java las variables de tipo char no tienen una inicialización por defecto (contrario a lo que ocurre en C++, que se inicializa con el carácter nulo), por lo que debe tener la precaución de inicializarla. Tenga en cuenta que a un carácter se le puede asignar un numero entero (su código ASCII-unicode) [2][3][4].
char letra = '0';
char letra = 0;

Las instrucciones: **char letra = 70;** **System.out.println(""+letra);** Imprimen: F

Tabla 4 Impresión de un caracter mediante la instrucción System

Algunas de las secuencias de escape (caracteres especiales usados con frecuencia), se presentan en la

Tabla 5.

'\n'	Salto de línea
'\t'	Permite un salto al siguiente tope de tabulación
'\b'	Retrocede el cursor en un espacio
'\'	Se usa para mostrar el carácter '
'\"'	Se usa para escribir una comilla doble (“)
'\''	Se usa para escribir una comilla simple (')
'\\'	Se usa para mostrar el carácter back-slash (\)

Tabla 5 Secuencias de escape

Un dato de tipo booleano se caracteriza por tomar únicamente dos valores `true` o `false`. Su correcta declaración e inicialización se muestra en la Tabla 6.

Nombre	Rango de valores	Declaración e inicialización
boolean	**true**, **false**	**boolean** bandera = **false;**

Tabla 6 Tipo de dato boolean

1.1.3.Constantes

Las constantes pueden usarse para dos objetivos: representar valores inmutables o para representar el dominio de un atributos. La Tabla 7 ilustra cómo se declara una constante [1] [2][3][4].

```
public final static double PI  = 1416;
public final static double IVA = 0.18;
```

Representan valores inmutables

```
public final static int CARRO  = 1;
public final static int MOTO   = 2;
```

Representan el dominio de un atributo

Tabla 7 Descripción uso de constantes

La instrucción final impide que las constantes sean modificadas durante la ejecución del programa.

1.1.4.Tipos de datos por referencia

Java cuenta con dos tipos de datos: valor y referencia. Cuando se declara una variable que corresponde a un tipo de dato primitivo, se está haciendo referencia a una variable por valor, que almacena su contenido en una estructura denominada pila (stack). Los tipos de datos por referencia, corresponden a instancias de clases y utilizan un área de memoria llamada montículo o heap. Una clase es una plantilla para crear objeto y un objeto es una instancia de una clase. Un tipo de dato primitivo puede trabajarse a través de clases envoltorio. Las clases envoltorio permiten manipular un tipo de dato primitivo como un objeto [2][3][8]. Los 8 tipos de datos primitivos con su correspondiente clase envoltorio se muestran en la Tabla 8.

byte - Byte
short - Short
int - Integer
long - Long
float - Float
double - Double
char - Character
boolean - Boolean

Tabla 8 Tipo de dato primitivo con su correspondiente clase envoltorio

El uso de clases envoltorio permite hacer uso de funcionalidades adicionales que fueron implementados en Java en cada clase, por ejemplo:

Clase	Screenshot de algunos de los métodos implementados en Java en clases envoltorio
Integer La clase **Integer** encierra un dato primitivo (tipo **int)** en un objeto. Esta clase tiene métodos entre ellos: convertir un **int** en un String y un String en **int** Integer dato=**new Integer(5);**	byteValue() : byte - Integer compareTo(Integer arg0) : int - Integer doubleValue() : double - Integer equals(Object arg0) : boolean - Integer floatValue() : float - Integer getClass() : Class<?> - Object hashCode() : int - Integer intValue() : int - Integer longValue() : long - Integer notify() : void - Object notifyAll() : void - Object shortValue() : short - Integer
Double La clase **Double** encierra un dato primitivo (tipo **double)** en un objeto. Esta clase tiene métodos entre ellos: convertir un **double** en un String y un String en **double** Double dato=**new Double(5.0);**	equals(Object obj) : boolean - Double floatValue() : float - Double getClass() : Class<?> - Object hashCode() : int - Double intValue() : int - Double isInfinite() : boolean - Double isNaN() : boolean - Double longValue() : long - Double notify() : void - Object notifyAll() : void - Object shortValue() : short - Double
Character Encierra un tipo de dato primitivo de tipo char Character c=**new Character('A');**	isLetter(char ch) : boolean - Character isLetter(int codePoint) : boolean - Character isLetterOrDigit(char ch) : boolean - Character isLetterOrDigit(int codePoint) : boolean - Character isLowerCase(char ch) : boolean - Character isLowerCase(int codePoint) : boolean - Character isLowSurrogate(char ch) : boolean - Character isMirrored(char ch) : boolean - Character isMirrored(int codePoint) : boolean - Character isSpace(char ch) : boolean - Character

Tabla 9 Clases envoltorio que se usarán en este libro

La clase String permite expresar un conjunto de caracteres (cadena). Es de anotar, que *String no es un tipo de dato primitivo*. Para inicializar una cadena se hace uso de comillas dobles. **Ejemplo**
String direccion= "Carrera 12 #23 10";
String nombre= "Maria Jose";

La clase String consta de variados métodos, algunos de ellos, para extracción de caracteres, comparación y concatenación.

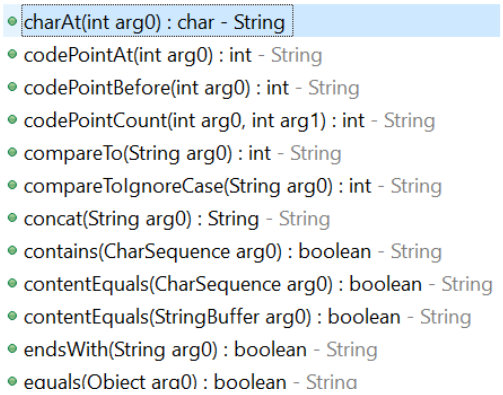


Figura 1 Screenshot de algunos de los métodos de la clase String

Las cadenas se pueden concatenar. Concatenar es unir dos cadenas, en Java esto se logra con el operador +, tal como se muestra en el siguiente ejemplo:

```
String direccion= "Carrera 12 #23 10";
String nombre= "Maria Jose";
String concatenado = dirección + nombre;
```

A un String se le puede concatenar cualquier tipo de dato primitivo

```
String nombre= "Maria Jose";
int edad= 20;
String resultado = nombre  +  " tiene " + edad +  " años ";
```

Actividad 1. Identificar por cada identificador el tipo de dato correspondiente

1) marca	( ) **int**
2) capacidadLibras	( ) **String**
3) modelo	( ) **String**
4) serie	( ) **String**
5) tipoProducto	( ) **double**
6) precio	( ) **int**
7) isbn	( ) **boolean**
8) activo	( ) **String**

1.1.5.Expresiones (Operadores aritméticos, lógicos, relacionales, promoción automática y casting)

Una expresión se compone de variables, constantes y/o operadores.

A. Operadores aritméticos

Este tipo de operadores permite realizar operaciones. Los operadores son: + (suma), - (resta), * (multiplicación), / (división), % (módulo o residuo de la división entera).

Se debe tener en cuenta que en Java entero dividido entre entero da como resultado otro entero. Por ejemplo:

```
int    a      = 20;
int    b      = 3;
double res = ( a + b ) / 3;
```

El programa debería imprimir 7.66666666667 pero da 7. Si lo que se desea es conservar estos decimales se pueden aplicar las siguientes 3 soluciones:

```
double res = (( a + b )* 1.0) / 3;
```

```
double auxiliar  = a + b;
double res = auxiliar / 3;
```

```
double res = ( a + b ) / 3.0;
```

B. Operadores de incremento y decremento

Java posee el operador incremental unario ++, y el operador decremental unario --, para incrementar o decrementar el valor de una variable en 1, respectivamente. La Tabla 10 presenta cada uno de estos operadores.

Operadores	Significado
`x++`	Incrementa x después de cualquier operación con ella (post-incremento)
`++x`	Incrementa x antes de cualquier operación con ella (pre-incremento)
`x--`	Decrementa x después de cualquier operación con ella (post-decremento)
`--x`	Decrementa x antes de cualquier operación con ella (pre-decremento)

Tabla 10 Operadores unarios

Las siguientes expresiones son equivalentes

Modo normal	Modo con operador
`x=x+1;`	`x++;`
`s=s-1;`	`s--;`

Los operadores pueden estar antes o después del operando. Por ejemplo:

`s = s+1;` es igual a:

`++s;`

`s++;`

Hay que tener precaución al usar esos operadores dentro de una expresión. Si operador incremento o decremento precede a su operando, java realiza operaciones de incremento o decremento antes de usar el valor del operando. Por el contrario, si el operador está luego del operando, java usa el valor del operador antes de incrementarlo o decrementarlo [2][3][4][9].

A continuación, se muestra un ejemplo junto con su equivalencia y resultado final.

Expresión	Expresión equivalente	Valor final
x = 11; x++;	x=11; x = x+1;	x = 12;
x = 12; y = ++x;	x=12; x = x+1; y = x;	y = 13;
x = 10; y = x++;	x=10; y = x; x=x+1;	y = 10;
x = 10; y = 6 * (++x);	x=10; x = x+1; y = 6 *x;	y = 66;
x = 10; y = 6 * (x++);	x=10; y = 6 *x; x = x+1;	y = 60;

Cuando se construyen expresiones es importe tener en consideración, la prioridad de operadores. La prioridad define el orden en el que se ejecutan las operaciones.

() Paréntesis +, - Operadores de signo (PRECAUCION: no hace referencia a suma y resta) ++, -- *, /, % +, - Suma y resta <, >, <=, >=, ==, != ! &&, || =

Tabla 11 Prioridad de operadores

Si se tiene una expresión, que contienen operadores con la misma prioridad, Java evalúa la expresión de izquierda a derecha. Si hay paréntesis lo que está dentro se evalúa primero. Si hay paréntesis internos, lo que está en su interior se evalúa primero que lo que está en los externos[1]. Ejemplo:

x = (2 + 3) * 8

Por precedencia de operadores, se efectúa primero (2+3) y por último se realiza el producto, por lo tanto:

x = (2 + 3) * 8

x = (5) * 8

x = 40

A continuación se muestra un segundo caso para la evaluación de la expresión.

x = (2+5+5) + (4+1)

Según la precedencia de los operadores, se debe evaluar primero las expresiones con paréntesis, estas son (2+5+5) y (4+1), en el caso de (3+5+5), como se tiene el mismo operador (+) dentro de la expresión, estos operandos, se evalúan de izquierda a derecha, por lo tanto la forma en que es evaluada la expresión es:

x = (2+5+5) + (2+1)

x = (7+5) + (3)

x = 12 + 3

x = 15

[1] Ejemplo explicativo de prioridad de operadores basado en material del profesor Julian Estebán Gutiérrez, de la Universidad del Quindío

La siguiente expresión involucra una expresión más compleja:

z = 1 + (10+6*6+7) + (18/2) / 3 / 1 + 2

Inicialmente, se resolverá las expresiones que se encuentran dentro de los paréntesis, se deben resolver las operaciones que tengan la mayor precedencia.

z = 1 + (10 + 6 * 6 + 1) + (18 / 2) / 3 / 1 + 2

z = 1 + (10 + 36 + 1) + 9 /3 / 1 + 2

Se observa que la expresión (3)/3/1, tiene tres operadores con idéntica precedencia, por lo tanto estos se evalúan de izquierda a derecha.

z = 1 + 47 + 3 + 2

Por lo tanto el valor final de y es z =53

Cuando se tenga una potencia y el exponente involucre una operación ésta debe encerrarse entre paréntesis:

$X^{1/2}$ = Math.pow (X, 1/2.0)

Recuerde que debe ponerse 2.0 porque 1/2=0 (entero dividido entre entero, da entero), mientras que 1/(2.0)=0.5

Si se tiene una raíz esta debe expresarse en forma de potencia. Ejemplo:

$\sqrt[n]{4}$ = Math.pow (4, 1.0/n);

Actividad 2 Construir expresiones que involucren únicamente los operadores aritméticos y el operador igual [2][3].

1. Construir las expresiones para los siguientes enunciados:

a. Se desea crear una aplicación que permita incrementar el salario de un empleado en un 10%

Variables	Expresión
Entrada: salario Salida: salarioIncrementado	double salarioIncrementado=salario + salario*0.1

b. Si se sabe el porcentaje de incremento del salario y el nuevo salario informar cuál era el salario previo

Variables	Expresión
Entrada: Salida:	

c. Si se sabe el salario incrementado y el salario anterior informar cual fue el porcentaje de incremento.

Variables	Expresión
Entrada: Salida:	

2. Construir las expresiones referentes al tetraedro.
a. Si se sabe el volumen de un tetraedro regular, calcular el lado. Recuerde que el volumen del tetraedro = $\frac{\sqrt{2} \times a^3}{12}$

Variables	Expresión
Entrada: a Salida: tetraedro	double tetraedro=(Math.sqrt(2) * Math.pow(a, 3))/12;

b. Si se sabe el volumen calcular a.

Variables	Expresión
Entrada: Salida:	

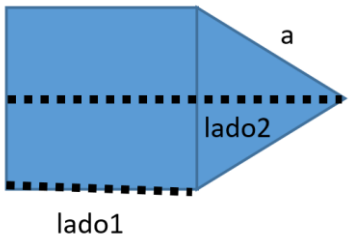
3. Construir las expresiones referentes a la Esfera.
a. Si se sabe el volumen de una esfera calcular el radio. Recuerde que volumenEsfera = 4/3 π r^3

Variables	Expresión
Entrada: Salida:	

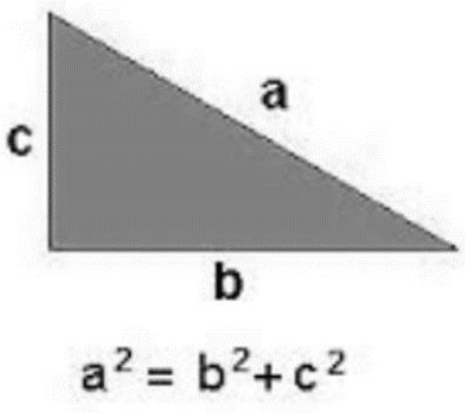
b. Si se sabe el radio calcular el volumen de la esfera

Variables	Expresión
Entrada: Salida:	

4. Una finca tiene la siguiente forma, hallar el perímetro de toda la finca y el área, si se conocen las longitudes que se encuentran punteadas.



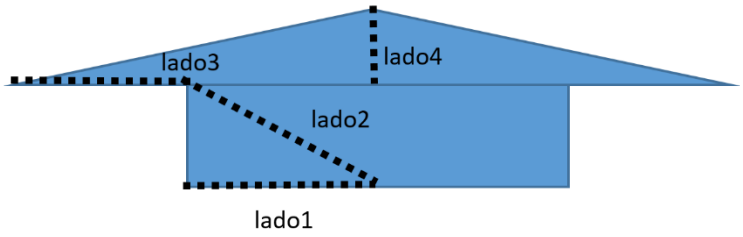
Tenga en cuenta que debe usar Pitágoras



Variables	Expresión perímetro de la finca
Entrada: Salida:	

Variables	Expresión área de la finca
Entrada: Salida:	

5. Calcular el área de la casa si se conocen las longitudes punteadas



Variables	Expresión área de la casa
Entrada: Salida:	

6. Obtener el número de cerámicas cuadradas para cubrir un salón rectangular. El usuario debe ingresar el largo y ancho del espacio a cubrir, y la longitud de cada cerámica.

Variables	Expresión numero ceramicas
Entrada: Salida:	

7. Se vende un vehículo perdiendo un porcentaje sobre el precio de compra. Hallar el precio de venta del mismo si se sabe que costó X pesos.

Variables	Expresión precio venta del vehiculo
Entrada: Salida:	

C. Promoción automática y casting

Cuando calcula una expresión que involucre operandos de tipo **byte** o **short,** Java efectúa una promoción a **int**. También se debe considerar que si hay un operador **long,** la expresión completa se promociona a este tipo de dato. Si un operador es float, entonces toda la expresión completa se promociona a **float**. Igual ocurre si es **double** [2][3][4][11]. Por ejemplo si se tiene:

```
byte a = 10, b = 40, j = 100;

int  d = a * b / d;
```

El resultado de a * b se calcula en **entero** y no en **byte** , esto ocurre porque tal operación puede salirse del rango de los **byte**. Observe el siguiente ejemplo:

```
byte b = 2, resultado;

resultado = b * 6; // No se puede efectuar porque un entero no se puede asignar aun byte.
```

Si se intenta multiplicar un byte b por 6 que es un entero el resultado será un entero, por tal razón este valor no podría asignarse a un byte. Si se requiere que el resultado se almacene en un byte, será necesario realizar una conversión explícita (casting):

```
byte b = 2, resultado;

resultado = (byte)( b * 6 );
```

D. Operadores lógicos

Estos operadores trabajan sobre operandos de tipo lógico para producir un resultado lógico (false o true). La Tabla 12 presenta los diferentes tipos de operadores lógicos.

Operador	Significado
&&	AND
||	OR
!	NOT

Tabla 12 Operadores lógicos

El operador AND (&&) da como resultado **true** si los dos operandos son verdaderos. De lo contrario da como resultado false.

Operando 1	Operador	Operando 2	Salida
false	&&	false	false
false	&&	true	false
true	&&	false	false
true	&&	true	true

Tabla 13 Operador AND

El operador OR (||) da como resultado verdadero cuando al menos uno de sus operandos es verdadero.

Operando 1	Operador	Operando 2	Salida
false	||	false	false
false	||	true	true
true	||	false	true
true	||	true	true

Tabla 14 Operador OR

Si la variables está en false, el operador negación cambia el valor de la variable a **true.** Si está en true lo vuelve false.

E. Operadores relacionales

Estos operadores dan como resultado un valor de tipo lógico (true o false). Los operadores lógicos son:

== (igualdad)
!= (diferente)
> (mayor que)
< (menor que)
>= (mayor o igual)
<= (menor o igual)

Tabla 15 Operadores relacionales

Actividad 3 Uso de operadores lógicos y relaciones

Escriba las expresiones correspondientes al enunciado:

a) Si es mayor de edad y salario excede 10350000

```
edad >= 18 && salario > 10350000
```

b) Si sexo es Masculino y edad es igual a 25.

```
sexo == 'F' &&  edad == 25

sexo.equals("femenino") &&  edad == 25
```

c) Si número es par y número es divisible por 100.

_____

e) Si celular está encendido y compartir internet está activo.

_____

1.2. Java como lenguaje de programación orientado a objetos

El mundo real está lleno de objetos, todo objeto tiene unas propiedades y un comportamiento. Java se basa en la construcción y manipulación de objetos. Cualquier concepto que se desee implementar en un programa Java debe ser encapsulado en una clase. En Java las aplicaciones son compiladas típicamente en bytecode, un código "neutro" que se ejecuta sobre una "máquina hipotética o virtual" denominada **Java Virtual Machine** (**JVM**), siendo la JVM quien interpreta el código para convertirlo al código particular de la unidad central de procesamiento utilizada. Esto implica que la salida de un compilador Java no es un código ejecutable, sino código binario diseñado para ser ejecutado por la JVM [1] [2][3][11].

1.3. Utilizando un lenguaje de Programación Orientada a Objetos para resolver un problema

Cuando una empresa tiene un problema que puede ser resuelto mediante el uso de un computador, es necesario contactar a un equipo de desarrollo. Éste debe determinar qué desea el cliente para poder dar una solución de calidad, que satisfaga sus necesidades.

Especificación del problema

El primer paso que debe seguir el equipo de desarrollo para poder dar solución a un problema es entenderlo. Esta etapa se denominada análisis. La salida de esta etapa se denomina especificación del problema.

La etapa de análisis considera varios aspectos, a saber:

- Identificar los requisitos funcionales, es decir, lo que el cliente espera que haga el programa. El programador debe tener una inmensa habilidad para interactuar con el cliente, pues son pocas las ocasiones en las que el

cliente sabe con exactitud las funcionalidades que el sistema debe proveer. El programador debe estar en capacidad de reconocer requisitos incompletos, contradictorios o ambiguos.

- Entender el mundo del problema, es decir, las reglas y estructura de la empresa dentro de la cual va a funcionar el software.
- Identificar los requisitos no funcionales, todas aquellas restricciones de los servicios o funciones que el sistema debe ofrecer por ejemplo rendimiento, cotas de tiempo, interfaz de usuario, factores humanos, documentación, consideraciones de hardware, características de ejecución, cuestiones de calidad, ambiente físico [1][2][3].

El programador debe adicionalmente haber contemplado la viabilidad del sistema, es decir, haber analizado el conjunto de necesidades para lograr proponer una solución en el plazo deseado que contemple restricciones económicas, técnicas, legales y operativas. El artefacto más importante que surge de la etapa de análisis se conoce como especificación de requisitos software (ERS). En la actualidad existen diversas modelos o formas para definir y documentar requisitos, todas ellas se encaminan hacia el mismo objetivo.

El proceso de solución del problema

Para resolver el problema el programador además de la etapa de análisis debe seguir las etapas de diseño y construcción de la solución. El proceso de diseño permite describir los aspectos del sistema a construir antes de que inicie el proceso de fabricación. En esta etapa se contemplan los requisitos identificados en el análisis y da como resultado un plano del software que incluye “la estructura de la solución, sus partes y relaciones” [1]. En esta etapa se fomenta la calidad del proyecto, pues permite plasmar con precisión los requerimientos del cliente [2][3][4].

La siguiente etapa es el proceso de construcción de la solución, ésta considera la creación del código fuente, el cual debe ser compilado para producir un código ejecutable, que posteriormente será instalado en el computador del usuario para que este pueda utilizarlo. El programador debe asegurarse de que el sistema funcione de acuerdo a los requisitos obtenidos del análisis. Es importante que se realice la documentación del código, la realización del manual de usuario, y posiblemente de un manual técnico que facilite realizar mantenimientos futuro y ampliaciones al sistema. La solución proporcionada al cliente debe ser evaluada través de las siguientes dimensiones:

- Evaluación operacional: analiza criterios como facilidad de uso, tiempo de respuesta, reportes en los que se muestra la información
- Impacto organizacional: eficiencia en el desempeño de los empleados, velocidad en el flujo de información, beneficios en finanzas.
- Evaluación del proceso de desarrollo, para valorar las herramientas y métodos seguidos durante el desarrollo del proyecto, al igual que la concordancia entre el esfuerzo y tiempo invertidos con el presupuesto destinado y criterios administrativos que se hayan contemplado.

Finalmente, la realización de pruebas permite verificar si el software entregado al cliente funciona correctamente y si cumple con los requisitos funcionales especificados. En caso de detectar falencias es necesario realizar las correcciones que garanticen su buen funcionamiento.

Es importante resaltar que el proceso de resolver un problema mediante el uso de un computador conduce a la creación de un programa y a la ejecución del mismo. Para construirlo es necesario hacer uso de un lenguaje de programación. Hoy en día existen muchos lenguajes de programación pero la tendencia actual es el uso de lenguaje orientados a objetos. La programación orientada a objetos o POO es un paradigma de programación que pretende simular el mundo real haciendo uso de objetos. A continuación se dan algunos conceptos introductorios, antes de iniciar con la construcción de casos de estudio [2][3][4].

Cuando una empresa requiere una aplicación, lo primero que hace es contactar a un equipo de desarrollo. Este escucha y analiza las necesidades del cliente, y sigue una serie de etapas (análisis, diseño e implementación) para poder construir un producto que satisfaga lo solicitado. En la etapa de análisis se determinan los requisitos funcionales, es decir, lo que se espera que el programa haga, el mundo del problema (contexto en el que se desarrolla el problema) y los requisitos no funcionales (restricciones tales como hardware, distribución geográfica y factores humanos). La fase de diseño considera las partes y diferentes relacionales que componen la solución. La fase de construcción, permite, partiendo del diseño, expresar la solución mediante algoritmos y construir el código fuente. Un algoritmo es un conjunto de pasos finitos y lógicos para resolver un problema, que consta de

pasos estrictamente descritos y acciones precisas. El archivo que ejecuta el cliente debe pasar por una etapa de pruebas para validar que si cumpla con los requisitos solicitados inicialmente [1]. La Figura 2 ilustra el proceso seguido:

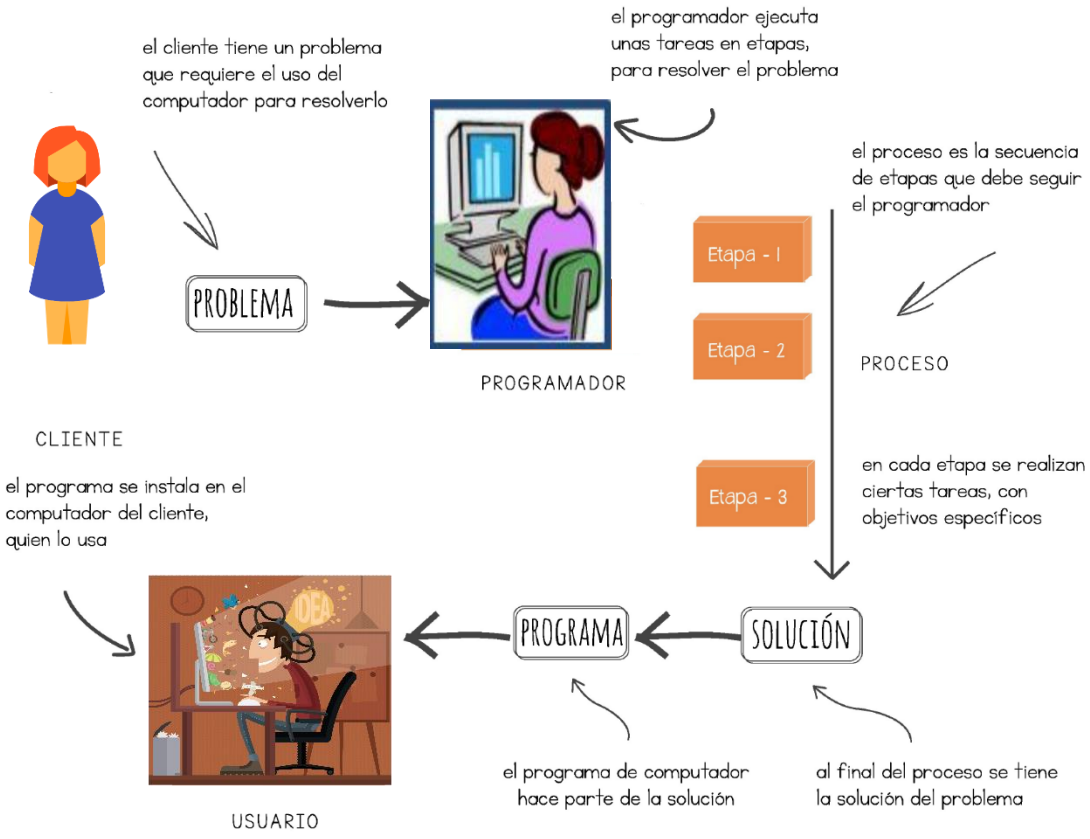


Figura 2 Proceso de solución de un problema, basado en [1]

Dentro de los elementos que forman parte de la solución de un problema, ver Figura 3, se contempla la construcción de un código fuente (para el caso de este libro, un archivo Java®). Este código luego debe compilarse, para dar lugar a un archivo de construcción (con extensión .class) que contiene el bytecode o código binario generado acorde a la arquitectura del equipo. Finalmente, la máquina virtual de java, interpreta el bytecode, para poder desplegar la aplicación. Es importante, hacer uso de datos de prueba para verificar la validez de los resultados que arroja el programa. El programador puede valerse de herramientas para automatizar las pruebas de los programas realizados en Java, tales como JUnit [1][2][3][4].

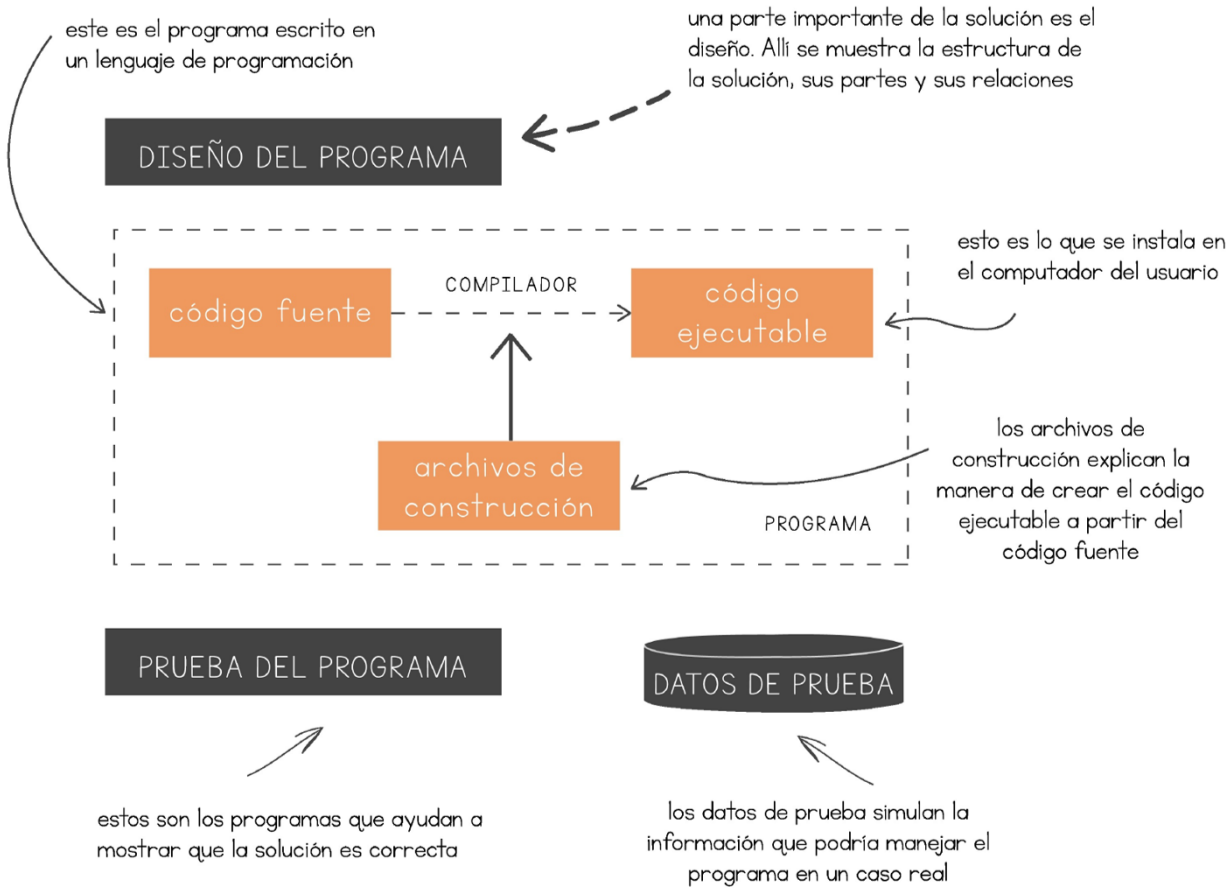


Figura 3 Elementos que forman parte de la solución, tomado de [1]

1.4. Objetos

"Un objeto es un elemento de un programa que almacena cierta información (estado del objeto), realiza algunas acciones (comportamiento del objeto), según sus responsabilidades y posee una única identidad en el sistema. Un objeto es un ejemplar de una clase" [2].

1.5. Clase

El mundo real está lleno de objetos (concretos o abstractos), por ejemplo una persona, un carro, un triángulo. Los cuales se pueden agrupar y abstraer en una clase. Una clase es una plantilla para fabricar objetos, por lo tanto, un objeto es una instancia de una clase. Las instancias de las clases (objetos) se comunican enviándose mensajes.

Un programa está compuesto por una o varias clases. Una clase define atributos, es decir, propiedades o características. También define un comportamiento. Un comportamiento es algo que un objeto puede realizar [2][3][4].

[2] Julián Esteban Gutiérrez. Material de apoyo al curso presencial de Paradigma Orientado a Objetos. 2011

1.6. Atributos

A las propiedades o características se les da el nombre de atributos. Un atributo se declara dentro de una clase, Una clase puede contener muchos o ningún atributo. El nombre de un atributo, por convención inicia con letra minúscula. Cada atributo tiene un nombre y un tipo de dato asociado. Los tipos de datos fueron definidos previamente en la sección 1.1.2.Tipos de datos primitivos. Sin embargo, el usuario podrá crear nuevos tipos dependiendo de sus necesidades [2][3][4].

Actividad 4. Resueva los siguientes puntos:

1. Para las siguientes clases identifique de qué tipo son los atributos mostrados. Escriba a la derecha dentro del paréntesis el número apropiado y la declaración correcta.

a) Lote

Atributo	Tipo	Declaración en Java
1) direccion 2) fichaCatastral 3) area	(3) `double` ( ) String ( ) String	`double area;`

b) Articulo

Atributo	Tipo	Declaración en Java
1) nombre 2) codigoBarras 3) cantidadExistencias 4) precio	( ) `double` ( ) **String** ( ) `int` ( ) `int`	

c) Persona

Atributo	Tipo	Declaración en Java
1) nombre 2) apellido 3) edad 4) correoElectronico 5) fechaNacimiento (dd-mm-aaaa) 6) direccion	( ) `int` ( ) String ( ) String ( ) String ( ) String ( ) String	

d) Puerta

Atributo	Tipo	Declaración en Java
1) Ancho 2) Alto 3) Color 4) Material	( ) **double** ( ) String ( ) String ( ) **double**	

2. Dados los siguientes objetos identifique los atributos correctos.

a) Estudiante

- [ ] nombre, dirección, grado, notaPromedio
- [ ] nombre, dirección, salario, fechaIngresoEmpresa, fechaNacimiento
- [ ] nombreProducto, codigoProducto, cantidadExistencias
- [ ] nombre, dirección, cantidadHijos

b) Juguete

- [ ] talla, color, material, tipo
- [ ] tipoJuguete, idJuguete, marcaJuguete, precio
- [ ] modelo, numeroMotor, placa
- [ ] numero, modelo, tonoDeTimbre, cámara, numTeclas, listadoContactos

1.7. Comportamiento

Una operación o comportamiento es algo que un objeto puede hacer. El comportamiento se define en la clase, haciendo uso de uso de métodos, pero es el objeto quien lo efectúa [2][3][4]. Algunos ejemplos se presentan a continuación:

Clase Triangulo: **Atributos**: base y altura **Métodos**: calcularArea y calcularPerimetro.

Clase Ceramica: **Atributos**: largo, ancho, color **Métodos**: obtenerAreaBaldosa().

Clase Licuadora: **Atributos**: marca, numeroSerie, capacidadLibras, modelo. **Métodos**: licuarNivel1, procesarAlimentos, partirHielo.

Observe que el nombre de los métodos inicia en en verbo y en minúscula.

Actividad 5. Para cada una de las clases indicadas especifique los atributos y los métodos.

ZAPATO	**ATRIBUTOS**				
	1.	2.	3.	4.	5.
	6.	7.	8.	9.	10.
	MÉTODOS				
	1.	2.	3.	4.	5.
	6.	7.	8.	9.	10.

Para la clase cepilloCabello:

C E P I L L O	**ATRIBUTOS**				
	1.	2.	3.	4.	5.
	6.	7.	8.	9.	10.
	MÉTODOS				
	1.	2.	3.	4.	5.
	6.	7.	8.	9.	10.

Es importante aclarar la diferencia entre Clase y objeto. Una clase es simplemente una plantilla para construir objetos y un objeto es una materialización (instancia o descripción completa de una clase). En lenguaje UML, una clase se representa mediante un rectángulo dividido en 3 secciones. La primera, es el nombre de la clase, la segunda corresponde a los atributos, y la tercera, son los métodos u operaciones [2][3][4]. Tal como puede verse a continuación:

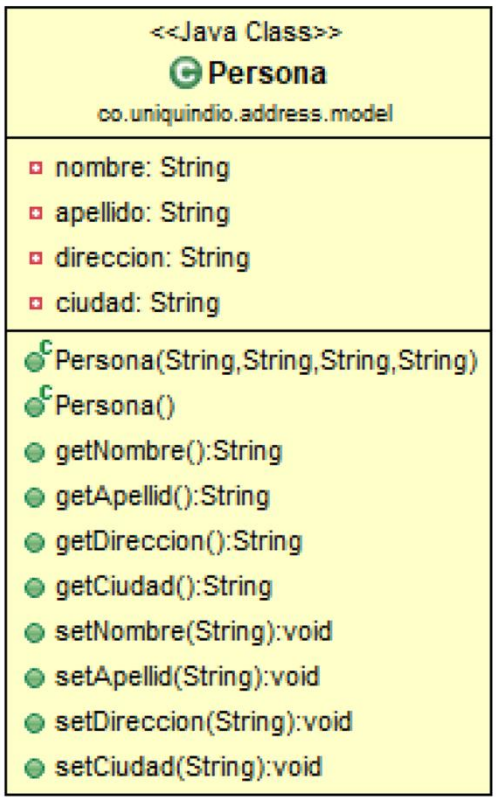


Figura 4 Clase Persona

Luego de construir la clase es posible instanciarla. Por ejemplo para la clase Persona del ejemplo anterior se puede crear un objeto de la siguiente forma:

```
Persona miPersona1;     //miPersona1 es una referencia a un objeto de  tipo Persona
miPersona = new Persona();
miPersona.setNombre("Luis");  //método para fijar el atributo nombre
miPersona.setApellido("Rodriguez"); //método para fijar el atributo apellido
miPersona.setDireccion("Calle 3#23"); //método para fijar el atributo direccion
miPersona.setCiudad("Armenia"); //método para fijar el atributo ciudad
```

La primera línea, permite realizar la declaración del objeto y, la segunda, permite reservar memoria (instanciar). Las demás líneas permiten inicializar los atributos de la clase. Observe que para reservar memoria, acorde al tipo de objeto que se desea crear, se hace uso del operador new. La instrucción para reservar memoria puede modificarse si el programador construye un método constructor dentro de la clase.

```
Persona miPersona1;     //miPersona1 es una referencia a un objeto de  tipo Persona
miPersona = new Persona(("Luis", "Rodriguez","Calle 3#23","Armenia");
```

1.8. Paquetes

Un paquete es una carpeta que contiene grupos de clases relacionadas. Por ejemplo, si se crea una interfaz gráfica esta clase se guardará en un paquete llamado view. Si se construye una clase del mundo del problema, en donde se desarrolle la lógica, irá en el paquete model. Los paquetes son una importante herramienta, dado que permiten trabajar por capas, reutilizar código, manejar diferentes niveles de acceso y agrupar con base en características comunes. Por costumbre el nombre del paquete inicia con un dominio de nivel superior referente al nombre de la organización y dominio de la misma, enumerado en orden inverso. Todos los caracteres correspondientes al nombre del paquete deben ir en minúscula [2][3][4]. En general, a lo largo de este libro se trabajarán 3 paquetes. El primero contiene la aplicación principal, el segundo contiene la lógica y el tercero, la interfaz gráfica y clases controladoras. Estas últimas se encargan de recibir la información de las vistas y de enviar dicha información al modelo, para su actualización. El uso de estos paquetes se da en concordancia con la implementación del modelo vista-controlador.

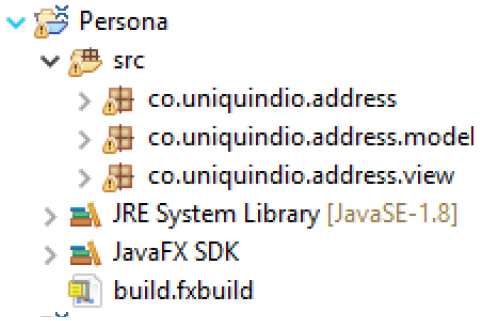


Figura 5 Paquetes del Proyecto Persona

El modelo vista controlador, es un patrón de arquitectura de software en el que se separa la lógica de la vista. Este modelo se compone de 3 elementos: controlador, vista y modelo. La vista se encarga de mostrar el modelo en la pantalla y de proveer un protocolo para pasar informacion desde y hacia el modelo. El controlador se encarga de coordinar la lógica (modelo) y vistas[3]. La estructura del modelo vista controlador puede verse en Figura 6.

[3] Modelo Vista controlador Reenskaug & Coplien, 2009

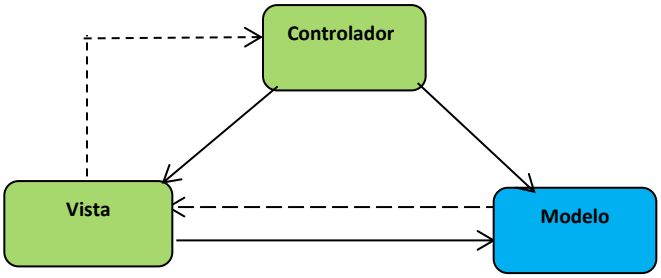


Figura 6 Modelo vista controlador. Las líneas completas indican la existencia de una asociación, las punteadas representan una asociación indirecta

1.9. Instalacion de Eclipse

El entorno en el cual se digitan todas las aplicaciones de este libro es Eclipse. Este IDE fue desarrollado por IBM.

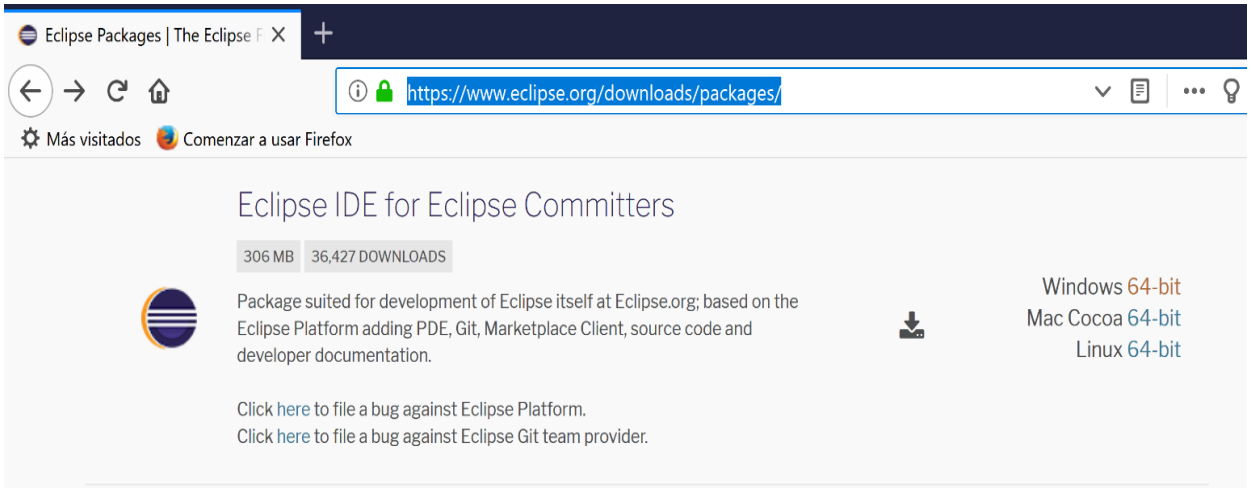


Figura 7 Página de descarga de Eclipse

Descomprima el archivo y ejecute Eclipse. En el navegador de Windows digite la siguiente dirección https://marketplace.eclipse.org/category/free-tagging/javafx. Allí podrá encontrar una serie de plugins para Eclipse que le permitirán trabajar con Java FX. Elija e(fx)clipse, ver Figura 8.

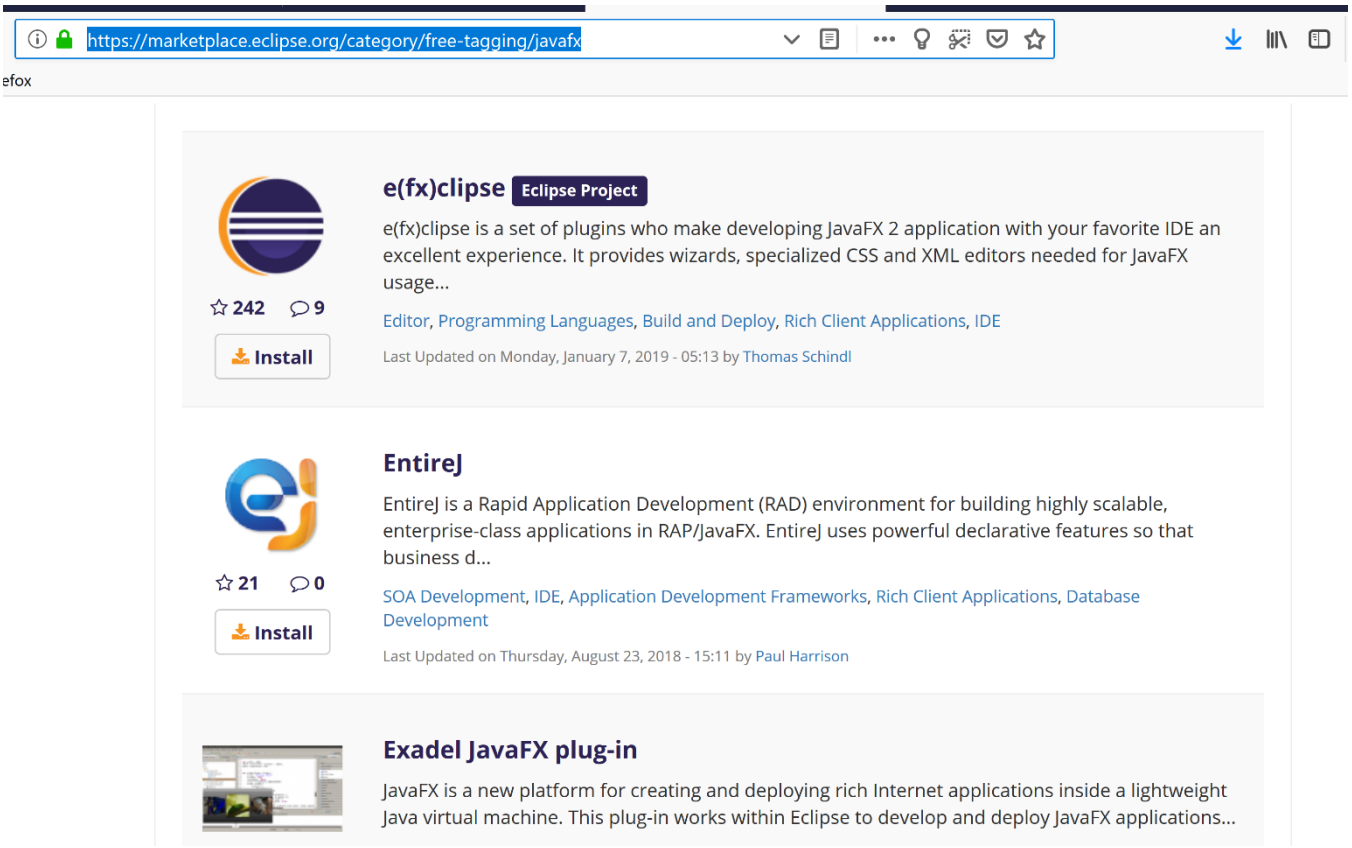


Figura 8 JavaFX plugin

En paralelo abra la ventana de eclipse y la de Marketplace. Presione el botón de instalación del Plugin y proceda a arrastrar el puntero hacia el IDE. El proceso de instalación iniciará inmediatamente.

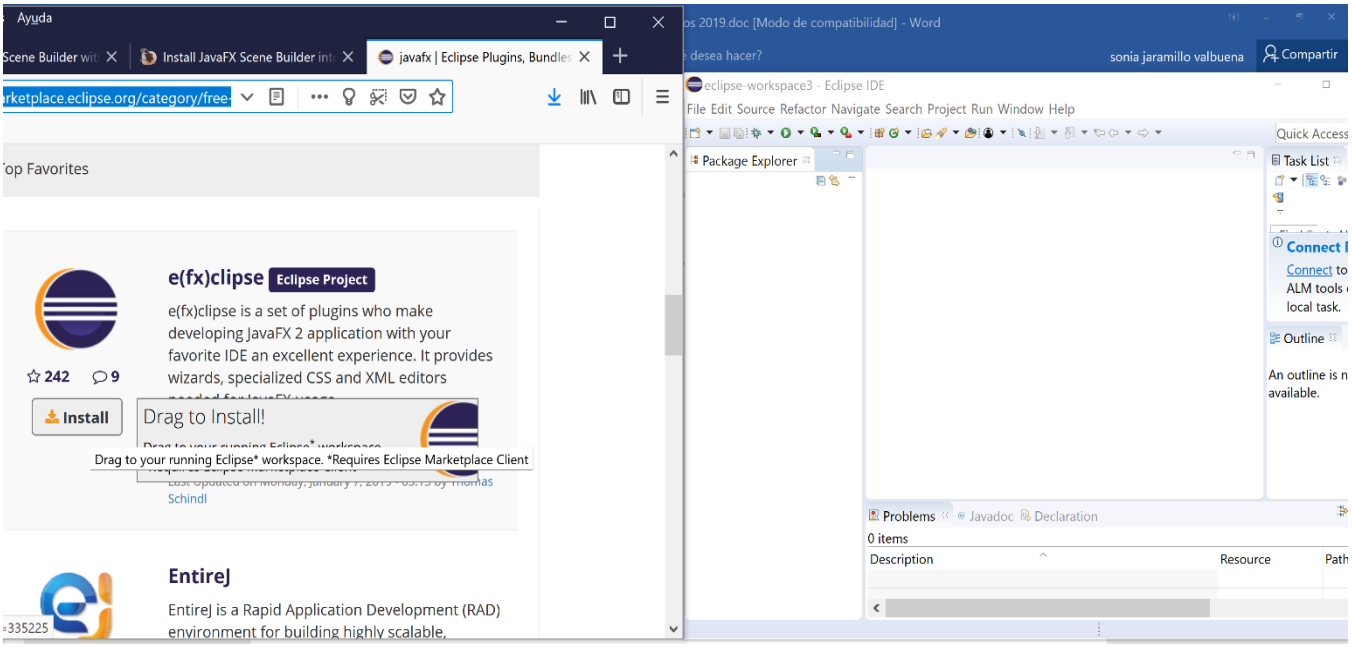


Figura 9 Instalación plugin mediante MarketPlace

Si el proceso de instalación no inicia, en Eclipse, de clic en Eclipse Marketplace y busque el plugin. De clic en instalar y acepte los términos de la licencia.

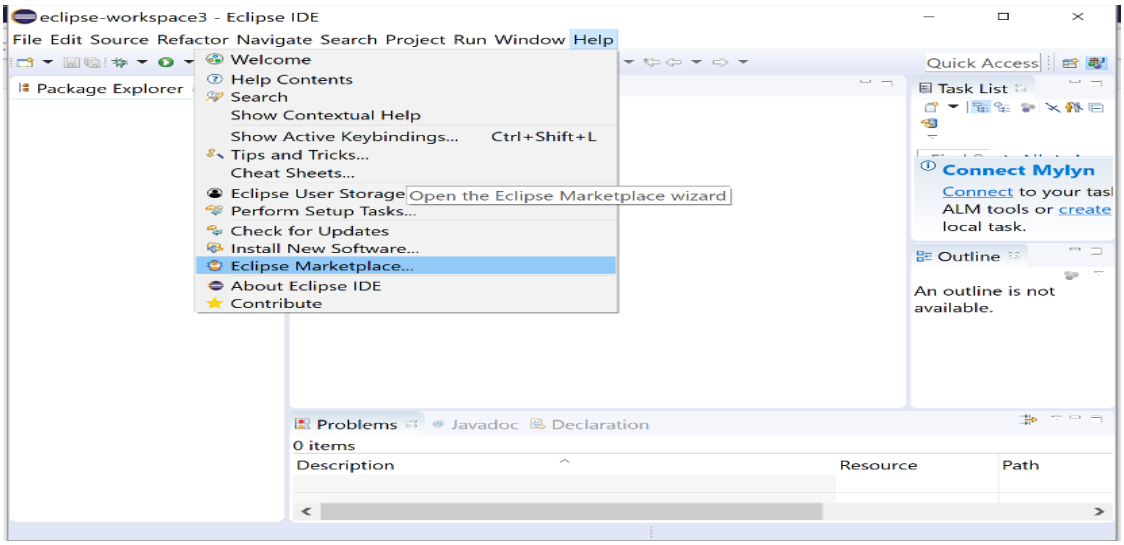


Figura 10 Búsqueda del plugin en Eclipse MarketPlace

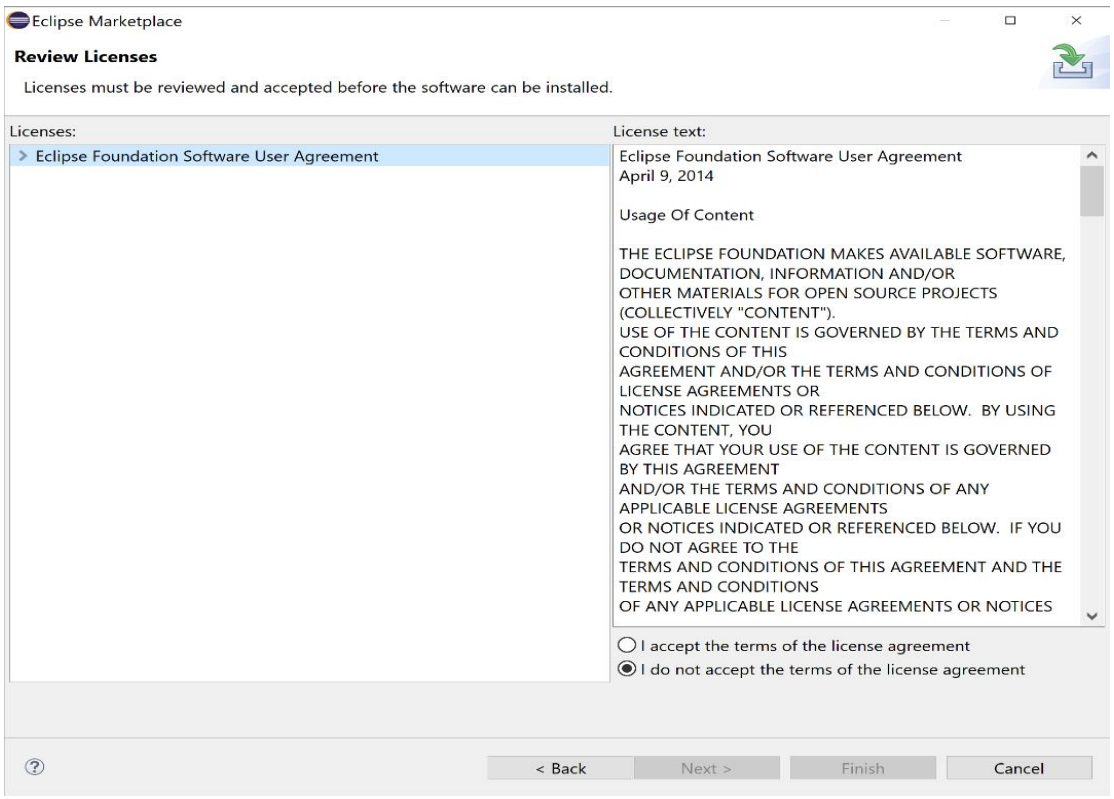


Figura 11 Aceptar términos de la licencia

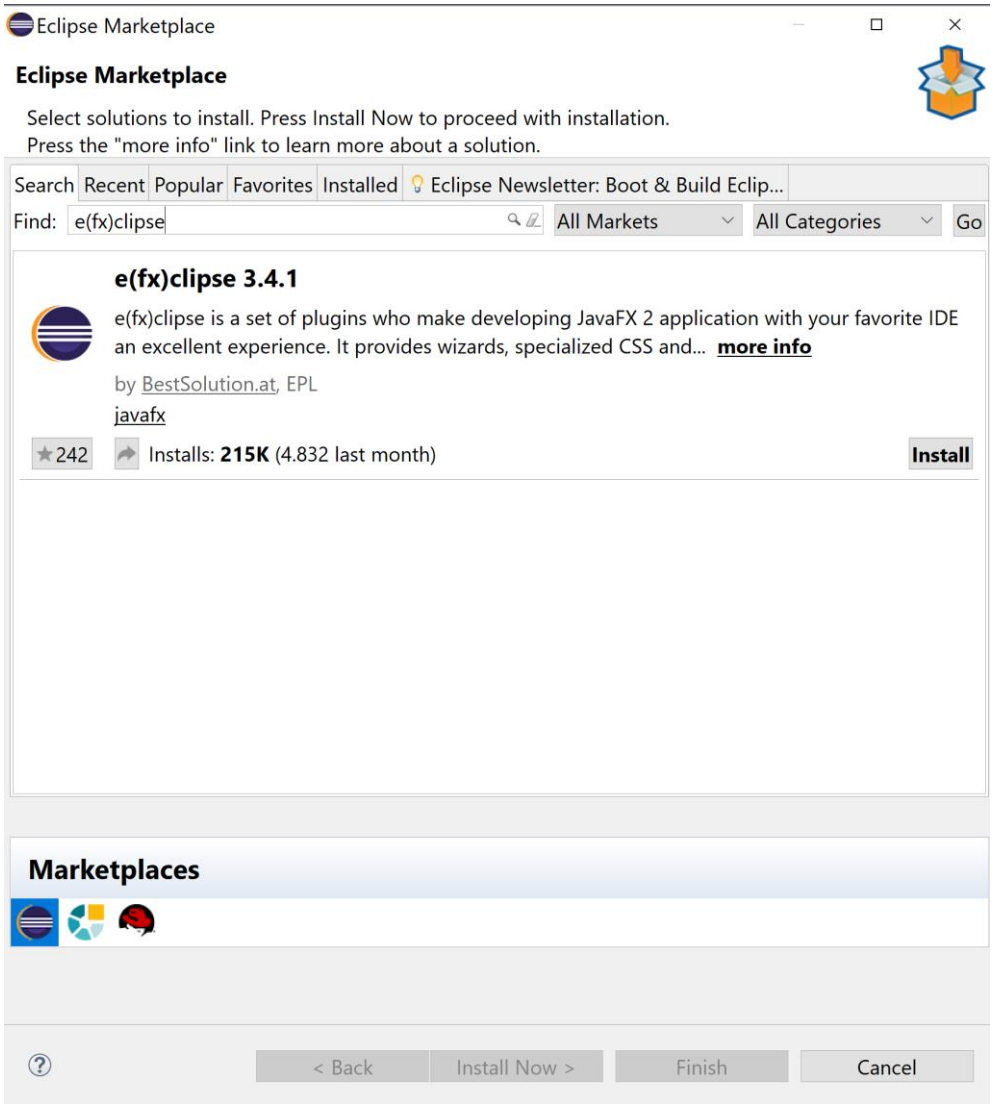


Figura 12 Búsqueda en Eclipse MarketPlace de un plugin

Descargue e instale JavaFX Scene Builder, de la página http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-1x-archive-2199384.html. Elija la versión apropiada para su sistema operativo.

JavaFX Scene Builder 2.0 Related Downloads

You must accept the Oracle BSD to download this software.

Thank you for accepting the Oracle BSD; you may now download this software.

Product / File Description	File Size	Download
Windows 32/64 bit (msi)	56.1 MB	javafx_scenebuilder-2_0-windows.msi
Mac OS X (dmg)	68.6 MB	javafx_scenebuilder-2_0-macosx-universal.dmg
Linux 32-bit (deb)	61.5 MB	javafx_scenebuilder-2_0-linux-i586.deb
Linux 32-bit (tar.gz)	61.8 MB	javafx_scenebuilder-2_0-linux-i586.tar.gz
Linux 64-bit (tar.gz)	60.7 MB	javafx_scenebuilder-2_0-linux-x64.tar.gz
Linux 64-bit (deb)	60.5 MB	javafx_scenebuilder-2_0-linux-x64.deb
JavaFX Scene Builder 2.0 Samples	0.3 MB	javafx_scenebuilder_samples-2_0.zip
JavaFX Scene Builder 2.0 Kit API Documentation	1.2 MB	javafx_scenebuilder_kit_javadoc-2_0.zip
JavaFX Scene Builder 2.0 Kit Samples	68 KB	javafx_scenebuilder_kit_samples-2_0.zip

Back to top

Figura 13 Seleccción del JavaFX SceneBuilder

Ejecute el archivo de instalación

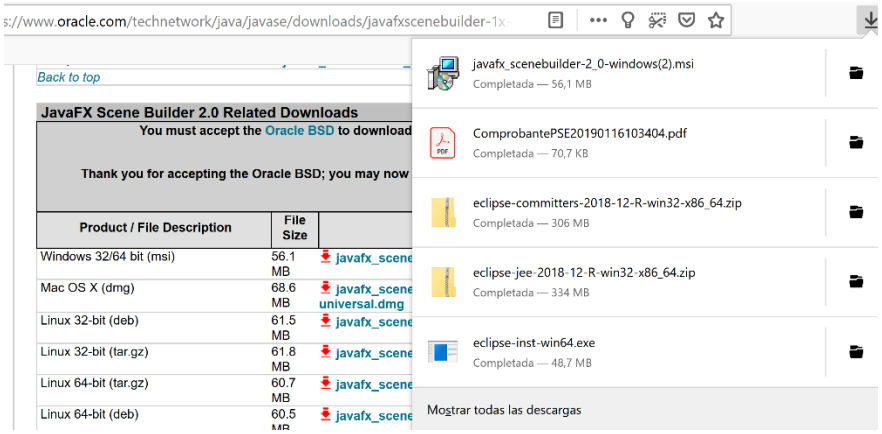


Figura 14 Identificación del archivo de instalación

De clic en aceptar

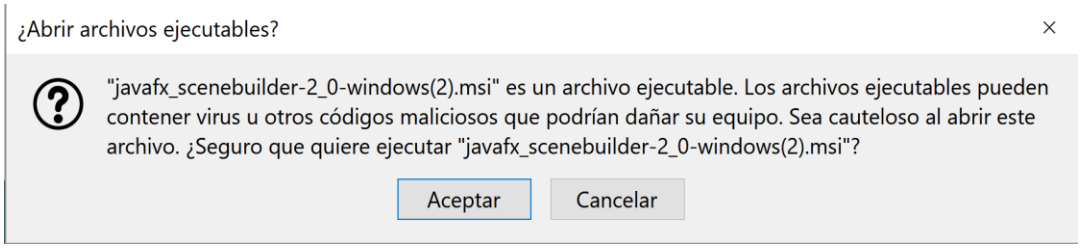


Figura 15 Ejecución del instalador

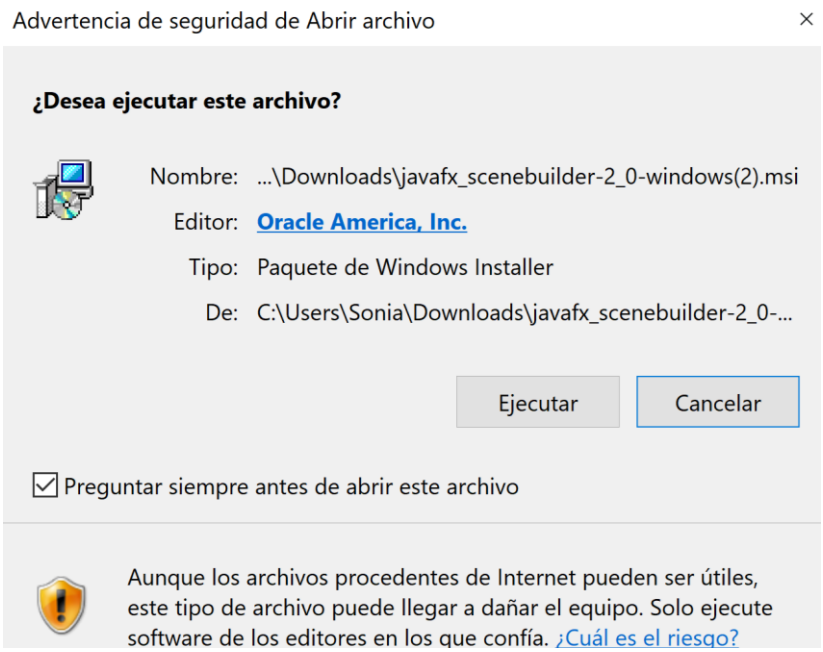


Figura 16 Archivo msi

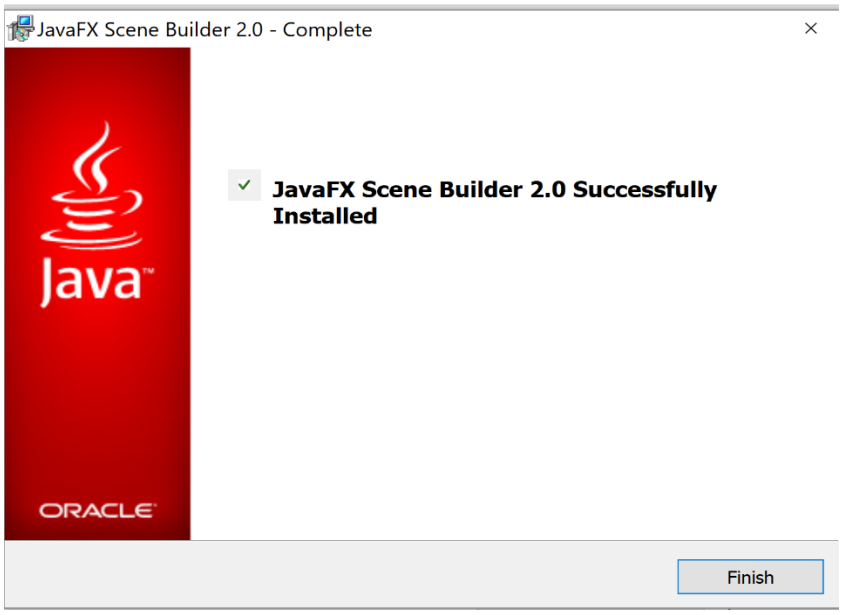


Figura 17 Finalizando el proceso de instalación

Ingrese a preferencias y especifique la ruta en donde quedó instalado el SceneBuilder. La ruta es: :\Program Files (x86)\Oracle\JavaFX Scene Builder 2.0\JavaFX Scene Builder 2.0.exe

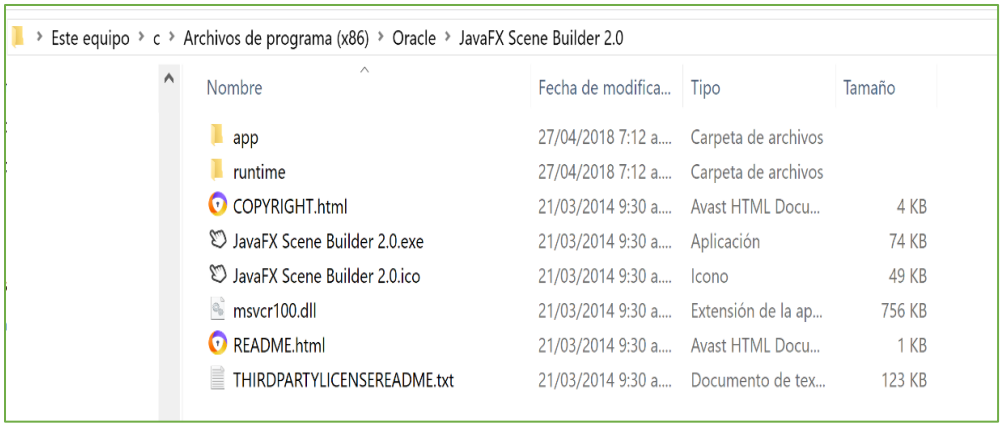


Figura 18 Ruta donde queda instalado Scene Builder

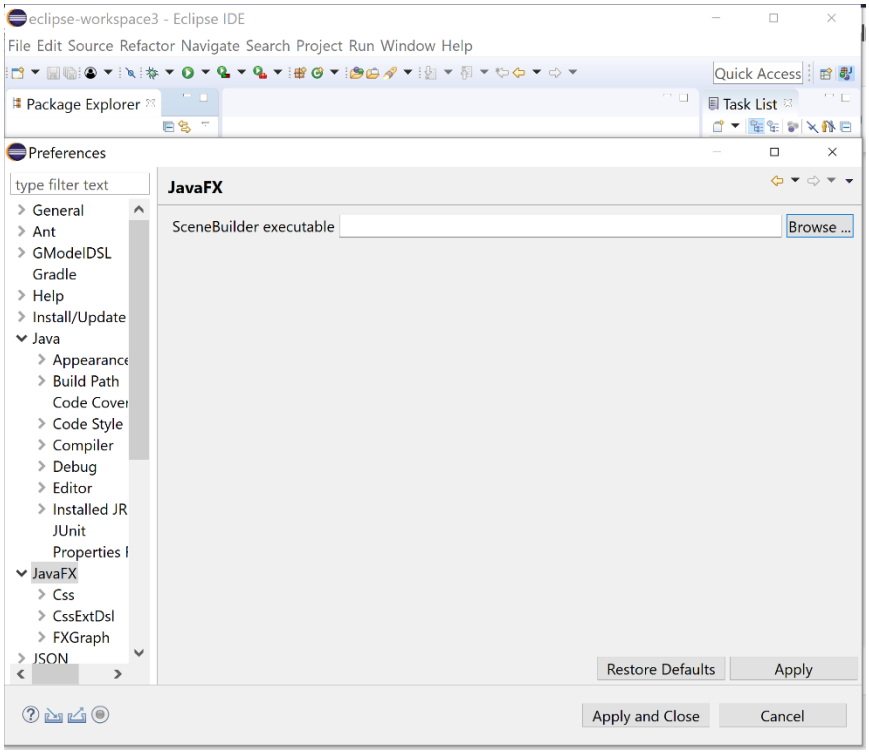


Figura 19 Configuración de la opción preferencias en Eclipse

También descargue el archivo Scene Builder Executable Jar de https://gluonhq.com/products/scene-builder/, y ubíquelo en la carpeta plugin de Eclipse

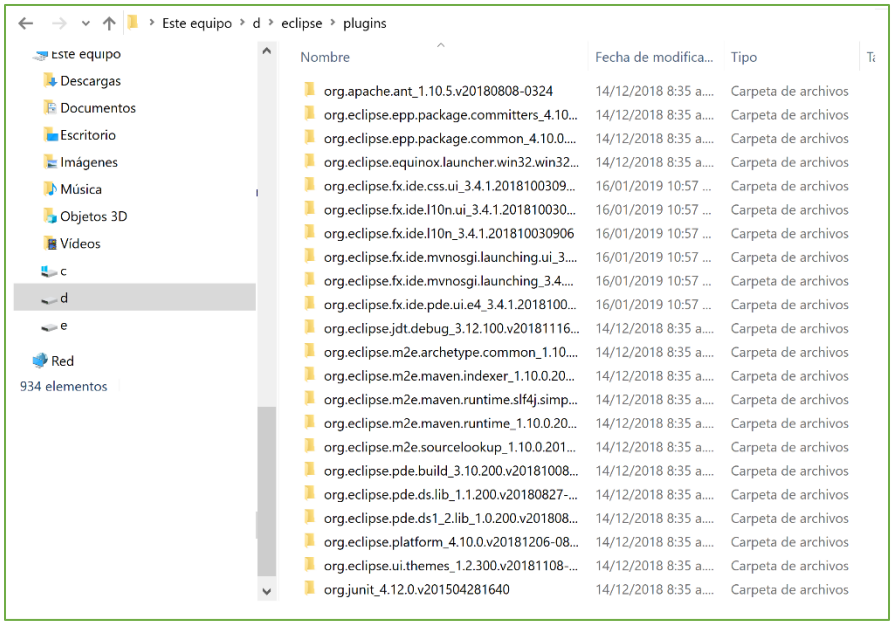


Figura 20 Jar correspondiente al Scene Builder

1.10. Caso de estudio 1 Unidad I: El Empleado

Se desea crear una aplicación para registrar la información de un empleado. El empleado tiene un codigo, nombre, direccion y ciudad de residencia. Se debe permitir crear un nuevo empleado y devolver el nombre del empleado.

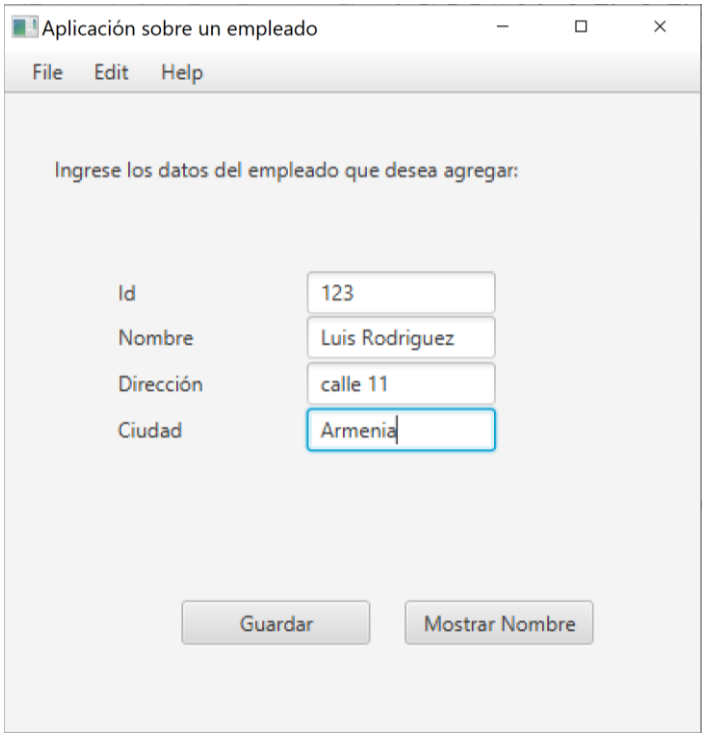


Figura 21 Interfaz aplicación del empleado

1.10.1. Comprensión del problema

a) Requisitos funcionales

NOMBRE	R1 – Crear un nuevo empleado
RESUMEN	Permite crear un nuevo estudiante
ENTRADAS	
id, nombre, direccion y ciudad	
RESULTADOS	
Un nuevo empleado ha sido creado	

NOMBRE	R3 – Devolver el nombre del empleado
RESUMEN	Permite devolver el nombre del empleado
ENTRADAS	
Ninguna	
RESULTADOS	
Él nombre del empleado	

b) El modelo del mundo del problema

Son varias las actividades que se deben realizar para construir el modelo del mundo del problema.

Identificar las entidades o clases. Para identificarlas, lo primero que debe hacer es resaltar los nombres o sustantivos. Normalmente se convierten en clases o atributos.

“Se desea crear una aplicación para manejar la información de un empleado. El empleado tiene un id, nombre, direccion y ciudad de residencia.

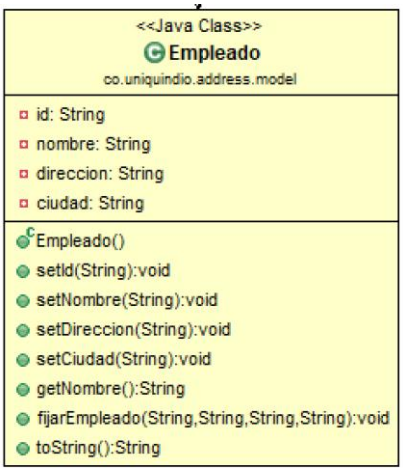
Se debe permitir crear un nuevo empleado y devolver el nombre del empleado.

ENTIDAD DEL MUNDO	DESCRIPCIÓN
Empleado	Es la entidad más importante del mundo del problema.

Se descarta nombre, id, direccion y ciudad de residencia puesto que son posibles atributos de la clase Empleado. Todos ellos son cadenas de caracteres.

Identificar los métodos. Para ello se deben resaltar los verbos. El nombre de los métodos debe ser un identificador válido en Java, éste debe ser mnemotécnico, iniciar en verbo y en minúscula. Si está compuesto por dos palabras, entonces la primera palabra irá en minúscula y la inicial de la segunda en mayúscula.

IDENTIFICACIÓN Y MODELAMIENTO DE ATRIBUTOS PARA EMPLEADO



ATRIBUTO	VALORES POSIBLES	Tipo de dato
id	Cadena de caracteres	String
nombre	Cadena de caracteres	String
direccion	Cadena de caracteres	String
ciudad	Cadena de caracteres	String

IDENTIFICACIÓN DE MÉTODOS

(para crear un empleado se debe fijar cada uno de los atributos id, nombre, direccion y ciudad)
setId
setNombre
setDireccion
setCiudad

(para devolver el nombre del empleado se requiere un método accesor – get)
getNombre()

Nota: Todo método lleva paréntesis. Dentro de los paréntesis pueden o no ir parámetros. Un parámetro es todo aquello que se necesita para resolver un problema y que no pertenece al objeto. Los parámetros están relacionados con las variables de entrada que se identificaron cuando se obtuvieron los requisitos funcionales. Por ahora, dado que se está introduciendo apenas el tema de identificación de métodos, se omitirán los parámetros, pues lo importante es determinar las operaciones que el objeto realiza [1][2][3].

3. Las Relaciones entre las clases del mundo

En este punto es importante familiarizarse con el concepto de diagrama de clases. "Un diagrama de clases describe los tipos de objetos que hay en el sistema y las diversas clases de relaciones estáticas que existen entre ellos. Hay dos tipos principales de relaciones estáticas: las asociaciones y los subtipos (un profesor es un empleado)." [4] Por ahora se se hará uso de asociaciones.

Es importante identificar las relaciones entre las clases para luego proceder a asignarle un nombre. Para indicar que existe una relación se hará uso de una línea con flechas en uno de sus extremos. El nombre de la asociación debe dejar claro que una entidad utiliza a otra clase como parte de sus atributos o características. Se debe tener en cuenta que entre dos clases puede existir más de una relación [2][3][4].

Antes de continuar el lector debe tener claro, que las clases se guardarán en carpetas especiales llamadas paquetes, dependiendo del tipo de clase que se vaya a definir. Si se crea una interfaz gráfica esta clase se almacenará en un paquete llamado view. Por el contrario, si se crea una clase del mundo real, que contenga la parte lógica se almacenará en un paquete llamado model.

A continuación se construirá un diagrama de clases que permita visualizar las relaciones de los elementos involucrados en el mundo del problema, con lo cual se ve claramente un modelo conceptual.

Métodos identificados	Nombre en el diagrama de clases
setId	setId(String id)
setNombre	setNombre(String nombre)
setDireccion	setDireccion(String direccion)
setCiudad	setCiudad(String ciudad)
getNombre	getNombre()

Observe que cada uno de los métodos set que se identificaron previamente, llevan un parámetro, que tiene el mismo nombre de la variable que se desea inicializar. Por ejemplo, si se requiere fijar el id, entonces el método setId debe llevar un parámetro, que en nuestro caso se llama también id. Lo mismo ocurre con el método setNombre, el cual tiene un parámetro llamado nombre. Por sencillez, es importante que tenga claro que todo método set lleva un parámetro, que será el mismo nombre de la variable que se desea inicializar. Es de anotar, que el nombre que se le de al parámetro puede ser diferente, lo realmente importante es que exista una coincidencia en tipo de dato, más no en nombre. No obstante, en este material el parámetro de cada método set llevará el mismo nombre de la variable que pretende inicializar [2][3][4].

[4]Fowler, Martin y Scott, Kendall

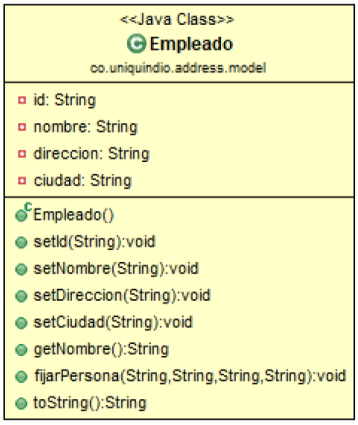


Figura 22 Clase Empleado

Como se explicó con anterioridad, cada método set tiene un parámetro. Al mirar la interfaz de este caso de estudio se puede observar que para poder crear el estudiante, el usuario debe ingresar cuatro datos, a saber: id, nombre, direccion y ciudad antes de que se presione el botón guardar. Esta información debe capturarse para poderla utilizar posteriormente. Tenga en cuenta que crear el empleado no solo implica reservarle un espacio en la memoria sino también almacenar la información que el usuario digita. Si la información no se almacena todo se perdería. Ahora bien, se dijo previamente que se requería de los métodos: setId(String id), setNombre(String nombre), setDireccion(String direccion), setCiudad (String ciudad)y getNombre(). Cada uno de estos métodos se ejecuta dependiendo del botón que el usuario presione, tal como se muestra en el siguiente gráfico.

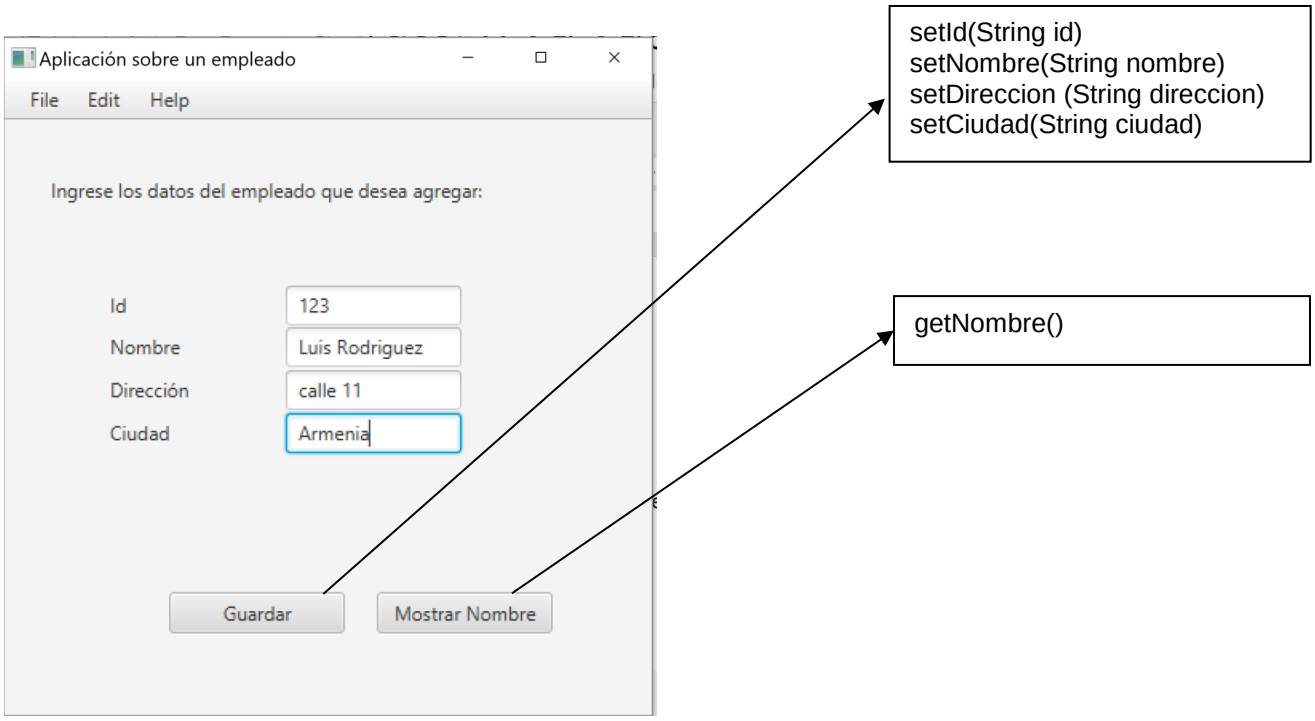


Figura 23 Descripción métodos a invocar cuándo se presiona el botón

- Cuando se presiona el botón guardar se debe reservar memoria al empleado (si aun no se ha hecho) y posteriormente capturar el id, el nombre, direccion y ciudad, para iniciar el empleado creado. Se debe guardar el código que el usuario digita en la ventana, este es el motivo por el cual el método setId tiene un parámetro, es decir, lo que el usuario ingresa en la ventana se le debe hacer llegar a la lógica. Igual ocurre con los otros 3 atributos.

- Si se presiona el botón “Mostrar nombre”, es porque previamente ya se ha ingresado la información, por ello no se requiere que el usuario ingrese información adicional a la que ya se encuentra almacenada. Esto permite concluir que el método getNombre no lleva parámetros.

1.10.2. Métodos

Los métodos permiten definir el comportamiento de los objetos que se crean con una clase. Un ***método*** es un conjunto de instrucciones que permiten resolver un problema particular. Está formado por un tipo de acceso, un tipo de retorno, un nombre, unos parámetros (si son necesarios) y un listado de instrucciones.

La estructura general de un método es la siguiente:

```
tipoDeAcceso tipoRetorno nombreDelMetodo( parámetros ){
   listado de instrucciones
}
```

El tipo de acceso indica la visibilidad del método para las clases externas, existen 4 tipos de accesos, a saber:

- private: Indica que el método solo puede ser utilizado en el interior de la clase en la que se encuentra definido
- public: Permite que el método sea utilizado desde cualquier clase.
- protected: Indica que el método sólo puede accederse desde su mismo paquete y desde cualquier clase que extienda (es decir que herede) la clase en la cual se encuentra, sin importar si está o no en el mismo paquete [2][3][4] [12][14]. Es de anotar que la herencia no se tratará en libro.

Si no se especifica un modificador se asume el nivel de acceso por defecto, el cual consiste en que el método puede ser accedido únicamente por las clases que pertenecen al mismo paquete. Un paquete puede verse como una carpeta a través de la cual es posible agrupar clases con características comunes.

La siguiente tabla resume los modificadores:

	La misma clase	**Otra clase del mismo paquete**	**Subclase de otro paquete**	**Otra clase de otro paquete**
private	X			
public	X	X	X	X
protected	X	X	X	
default	X	X		

Tabla 16 Modificadores de acceso

Es de anotar que los métodos construidos en este material llevarán el modificador public y los atributos private.

Por otra parte, el nombre del método debe ser un identificador válido en Java, éste debe ser mnemotécnico y tener correspondencia con lo que hace. También debe iniciar en verbo y en minúscula. Si está compuesto por dos palabras, entonces la primera palabra irá en minúscula y la inicial de la segunda en mayúscula.

El retorno indica el tipo del resultado [2][3][4]. Pueden considerarse dos opciones:

1. Sin Retorno (void). Es decir, el método no devuelve ningún valor. Estos métodos son útiles, dentro de las clases de la lógica, cuando se pretende modificar el valor de los atributos de una clase [1] y dentro de las clases de la interfaz, cuando se quiere mostrar mensajes o resultados. Si el método no devuelve un valor,

se debe poner void como tipo de retorno y no puede utilizarse return dentro de la implementación del método.

2. Con retorno: Muchos de los métodos que se utilizan en la vida cotidiana, retornan valores. Los métodos pueden retornar valores de tipos primitivos, pueden retornar objetos, pueden retornar arreglos entre otros. El retorno indica el tipo del resultado, por ejemplo en el caso del método getNombre() el retorno sería **String**, es decir, se devolvería un String que corresponde al nombre del empleado. Si hay retorno debe especificarse claramente el tipo de dato devuelto y es obligatorio el uso de la sentencia return para realizar dicha tarea.

Los parámetros son los datos necesarios para resolver el problema y que no pertenecen al objeto. Es importante distinguir los términos parámetro y argumento. Un parámetro es una variable definida por un método y que recibe un valor cuando se llama a ese método. Un argumento es un valor que se pasa a un método cuando éste es invocado"[5]. Dentro de un método pueden ir múltiples instrucciones, ya sea declaraciones de variables locales, expresiones, estructuras de decisión o repetitivas (estas últimas se tratarán en unidades siguientes)[1] [2][3][4].

Una operación debe ser simple y cohesiva, por lo cual es importante que use al menos uno de los atributos definidos en la clase.

Dependiendo de la función que realice un método, éste puede clasificarse en uno de los siguientes tipos:

- Métodos modificadores: Son los encargados de cambiar el estado del objeto
- Métodos analizadores: permiten calcular información apoyándose en la información con la que cuentan los objetos de la clase
- Métodos constructores: Su objetivo es inicializar los atributos de un objeto en el momento mismo de su creación.
- Métodos accesores o getter's: Permiten devolver el valor de un campo privado, es decir, un atributo de un objeto.

Cada método debe tener claramente definidos sus compromisos y responsabilidades (es decir, lo que va a hacer) [1][2]. No se recomienda la creación de métodos demasiado extensos que "realicen de todo". En su lugar, deben crearse múltiples métodos y luego integrarlos. Lo anterior permite obtener un código mejor estructurado y reutilizable, además de que, facilita la detección de errores.

1.10.3. Cómo construir un método

La construcción de un método incluye los pasos descritos en [2][3][4], que se muestran a continuación:

- Identificar el tipo de retorno
- Definir la signatura del método. La signatura de un método está compuesta por el nombre del método, los tipos de los parámetros y el orden de éstos). Se debe tener claro que los modificadores usados y el tipo devuelto no forman parte de la signatura del método.
- Agregar las instrucciones dentro de él.

Para ilustrar lo anterior se procede a retomar el caso de estudio 1:

Se desea crear una aplicación para manejar la información de un empleado. El empleado tiene un id, nombre, direccion y ciudad de residencia. Se debe permitir crear un nuevo empleado y devolver el nombre del empleado.

[5] Gómez, Carlos Eduardo y Gutiérrez, Julián Esteban

El diagrama de clases construido fue:

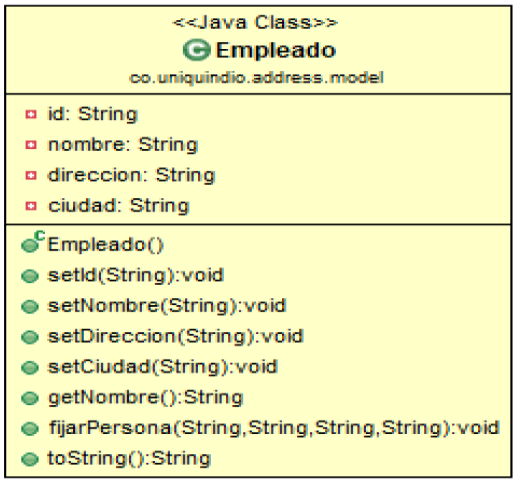


Figura 24 *Diagrama de clases, única clase del paquete view*

En primer lugar se inicia con la contrucción de los métodos modificadores o setter. Según este diagrama de clases, se debe construir uno por cada atributo. Los pasos para contruir un método set se presentan a continuación:
- No llevan retorno, es decir, se debe poner void.
- Llevan un parámetro, al cual se le pondrá el mismo nombre del atributo que se pretende inicializar.
- En su interior llevan una única línea: `this`.atributo=parámetro. Esto significa que el valor que llega por parámetro, y que proviene de la interfaz gráfica (es decir, es lo que el usuario ingresa), será usado para inicializar el atributo de la clase. `this`.atributo hace referencia al atributo de la clase y atributo al parámetro. Se llaman igual pero son dos variables diferentes.

La siguiente tabla ilustra los métodos set para los atributos de la clase Estudiante

```
public void setId(String id) {
	this.id = id;
}

public void setNombre(String nombre) {
	this.nombre = nombre;
}

public void setDireccion(String direccion) {
	this.direccion = direccion;
}

public void setCiudad(String ciudad) {
	this.ciudad = ciudad;
}
```

Finalmente se construyen los métodos get. Para su elaboración se siguen las siguientes reglas:
- Llevan retorno, del mismo tipo del atributo al que se le está haciendo el get.
- No llevan parámetros
- En su interior llevan la instrucción: return atributo;

get para nombre

```
String getNombre(  )
{
   return nombre;
}
```

Luego de haber construido los métodos se procede a integrar todo en una clase.

Programa 1 Clase Empleado

```
package co.uniquindio.address.model;
/**
 * Clase para crear un empleado
 * @author Sonia Jaramillo Valbuena
 * @author Sergio Augusto Cardona
 */
public class Empleado {
/**
 * Es el id del estudiante
 */
private String id;
/**
 * Es el nombre del estudiante
 */
private String nombre;
/**
 * Es la direccion del estudiante
 */
private String direccion;
/**
 * Es la ciudad de residencia del estudiante del estudiante
 */
private String ciudad;
/**
 * Metodo constructor, en este caso puede omitirse su construccion
 */
public Empleado()
{}
/**
 * Metodo modificador
 * @param id, es el id del empleado
 */
public void setId(String id) {
      this.id = id;
}
/**
 * Metodo modificador
 * @param nombre, Es el nombre del empleado
 */
public void setNombre(String nombre) {
      this.nombre = nombre;
}
/**
 * Metodo modificador
```

Programa 1 Clase Empleado

```
 * @param direccion, la direccion del empleado
 */
public void setDireccion(String direccion) {
	this.direccion = direccion;
}
/**
 * Metodo modificador
 * @param ciudad, la ciudad de residencia
 */
public void setCiudad(String ciudad) {
	this.ciudad = ciudad;
}
/**
 * Metodo accesor
 * @return el nombre del estudiante
 */
public String getNombre() {
	return nombre;
}

/**
 * Es metodo inicia todo el objeto existente con los valores recibidos
 * @param id Es el id del empleado
 * @param nombre Es el nombre del empleado
 * @param direccion Es la direccion del empleado
 * @param ciudad Es la ciudad del empleado
 */
public void fijarPersona(String id, String nombre, String direccion, String ciudad)
{
	this.id = id;
	this.nombre = nombre;
	this.direccion = direccion;
	this.ciudad = ciudad;
}

/**
 * Devuelve la representacion en String del empleado
 */
public String toString()
{
return id+"  "+nombre +"   "+direccion+"   "+ciudad;
}


}
```

1.10.4. Declarar variables en Java

El sitio donde sea declare una variable determina donde puede ser utilizada. Las posibles ubicaciones se describen a continuación:

- El primer lugar donde se puede declarar una variable es dentro de la clase, pero fuera de los métodos. Este tipo de variables se conocen como atributos de la clase y pueden usarse dentro de cualquier método contenido en la clase.

- El segundo lugar es al interior de cualquier método. Se les denomina variables locales y solo pueden utilizarse en instrucciones que estén contenidas en el método en donde se declaran.

- En tercer lugar se pueden definir como parámetros de un método, en donde luego de recibir valor, se pueden manipular como una variable local en ese método.

Se puede concluir entonces que una variable local sólo puede utilizarse dentro del método donde se declara, es decir, solo allí está viva y es desconocida por otros métodos[2][3].

1.10.5. Envío de mensajes o llamado de métodos

Cuando se envía un mensaje a un objeto, se le está ordenando que ejecute un método, el cual debió definirse previamente en la clase con la cual fue creado dicho objeto. Para mandar mensajes a los objetos se utiliza el operador punto. En general para llamar un método, lo que se hace es poner la referencia al objeto, seguido del operador punto, luego el nombre del método, y finalmente, dentro de paréntesis se pasan los correspondientes argumentos (tantos argumentos como parámetros se hayan puesto al momento de crear el método). Es importante que previamente a la invocación del método haya reservado memoria a través de la instrucción **new**, esto con el objetivo de evitar que se genere la excepción NullPointerException. NullPointerException aparece cuando se intenta acceder a métodos en una referencia que se encuentra nula, es decir a la cual no se le ha reservado memoria a través de new. En tal caso la llamada al método no es permitida [2][3][4].

A continuación se muestran las instrucciones necesarias para poder invocar los métodos y mostrar en pantalla los resultados que estos arrojan.

- Se reserva memoria

```
Empleado miEmpleado=new Empleado();
```

- Se lee la información de los campos de texto y se declaran tantas variables como métodos con retorno se hayan hecho.

La pantalla que se creó fue la siguiente. En ella hay campos de texto (TextFields, Buttons y Labels). Un Label es un componente que permite mostrar un texto estático, como es el caso de “Ingrese los datos del empleado que desea agregar”.

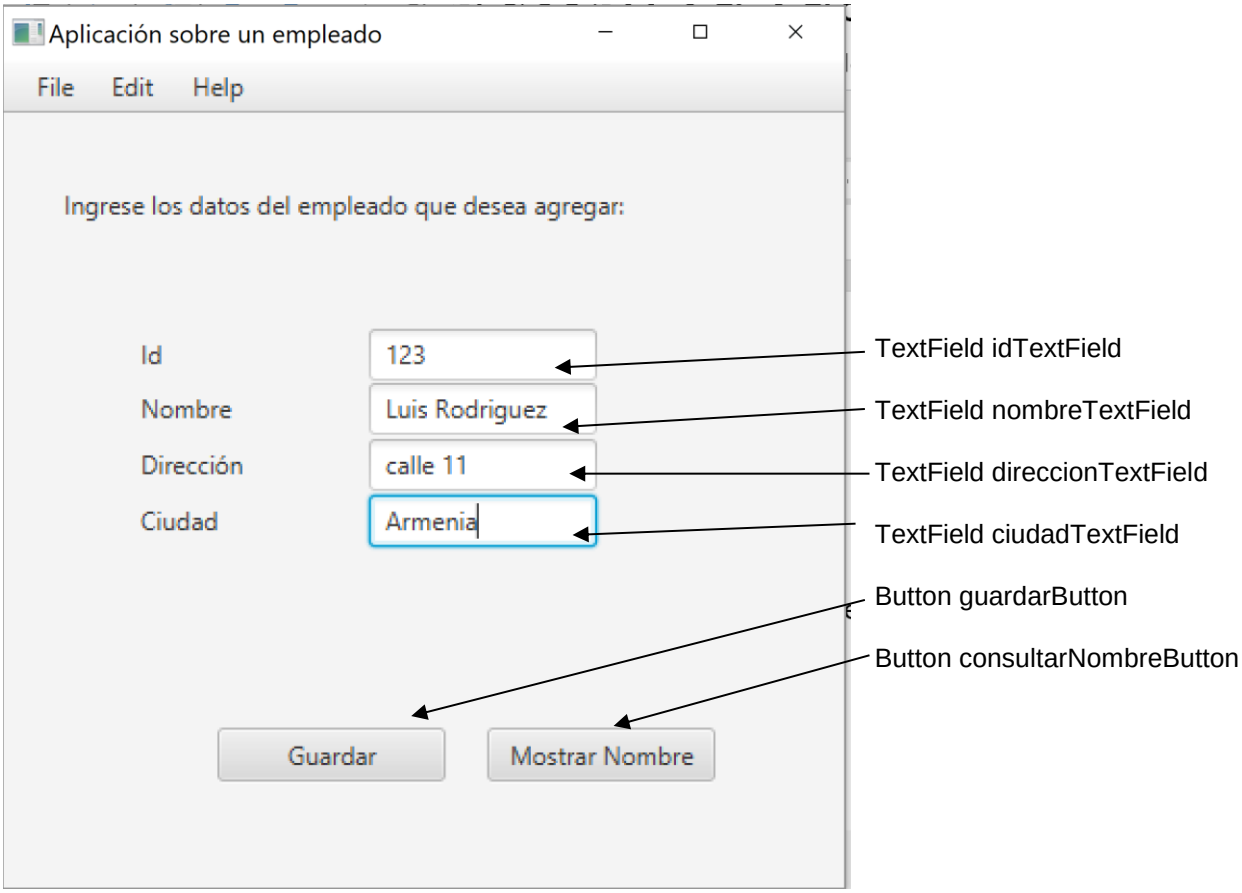


Figura 25 Botones y campos de texto

Para que los métodos funcionen es necesario tomar la información de los campos de texto de la interfaz. Esto será realizado mediante una clase que se llamará clase “Controladora”. En dicha clase hay una referencia a la clase Principal. La clase Principal, que extiende de Application, es la única que se comunicará directamente con la clase principal del mundo de la lógica, es decir, con Empleado (ver la siguiente figura).

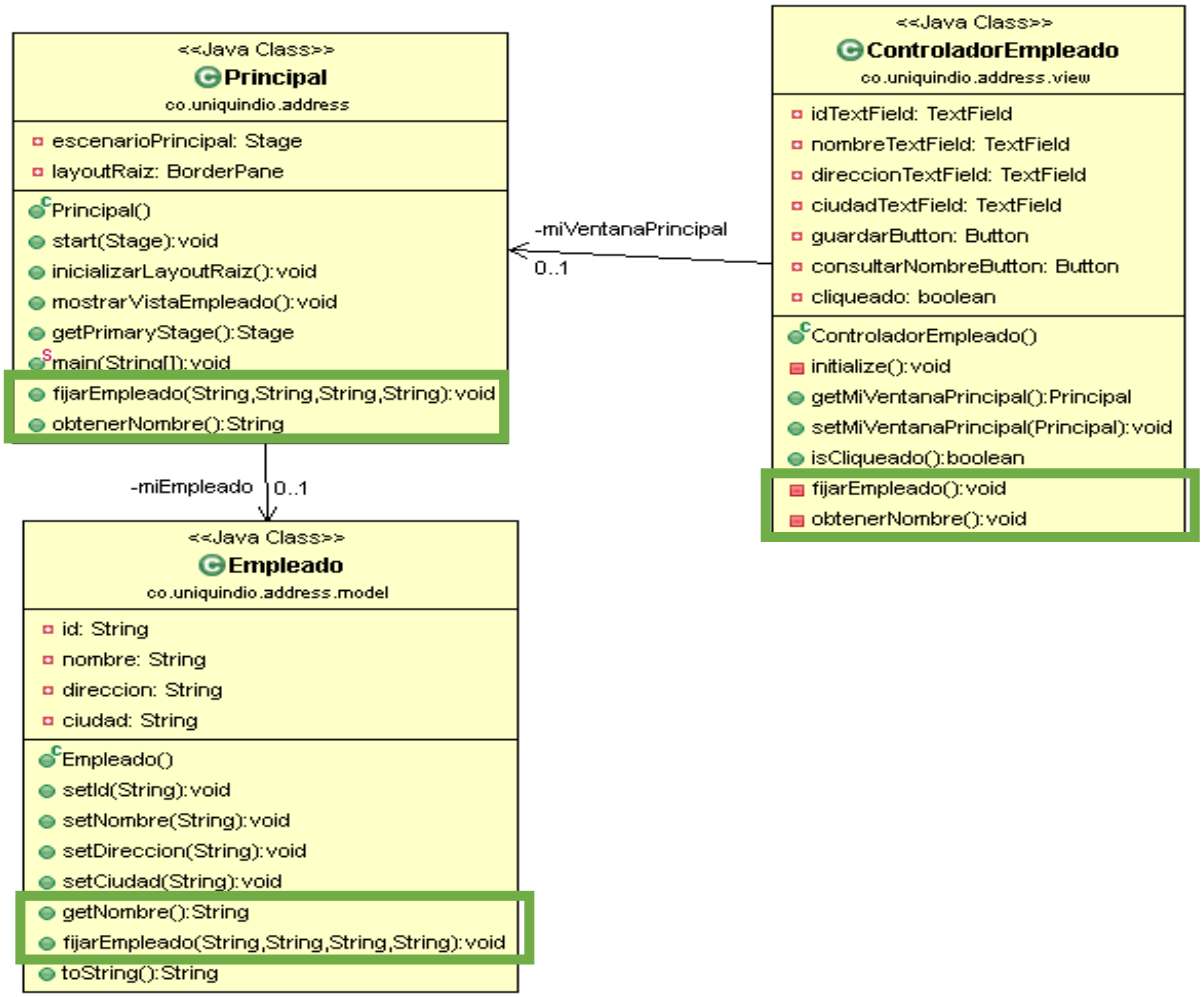


Figura 26 Diagrama de clases, relación entre la vista, la lógica y la clase controladora

La clase Principal crea el controlador, mediante las instrucciones:
ControladorEmpleado miControlador = loader.getController();
miControlador.setMiVentanaPrincipal(**this**);
Además, implementa dos métodos que son de interés, acorde a la interfaz contruída. Ellos son: fijarEmpleado (relacionada con el botón Guardar) y obtenerNombre (relacionado con mostrarNombre). El método fijarEmpleado tiene 4 parametros: id, nombre, direccion y ciudad. Esta información debe capturarse de la interfaz gráfica, mediante la clase controladora. La información que obtiene la clase controladora, es enviada a la clase principal del mundo del problema cuando se efectúa la instrucción miEmpleado.fijarPersona(id, nombre, direccion, ciudad). Observe que la clase Empleado del paquete view tiene también el método fijar empleado y getNombre.

```
//controlador
String id=idTextField.getText();
String nombre=nombreTextField.getText();
String direccion=direccionTextField.getText();
String ciudad=ciudadTextField.getText();
miVentanaPrincipal.inicializarEmpleado(id, nombre, direccion, ciudad);
```

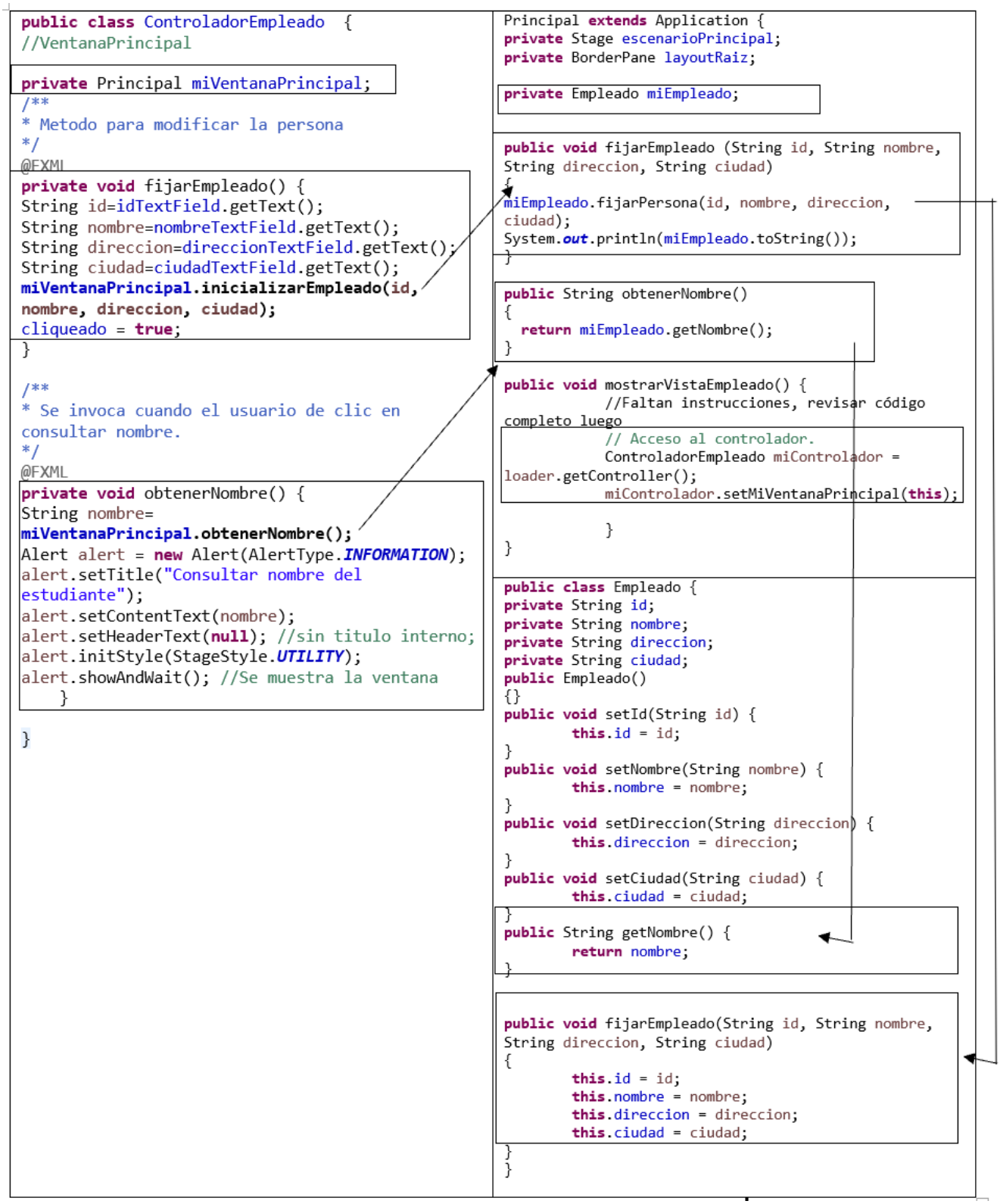


Figura 27 Relación entre las diferentes clases del proyecto

1.10.6. Documentación javadoc

"Cada programador debe aprender a escribir la especificación de su programa (o sea, la documentación), antes de escribir el programa"[6]. Documentar permite entender el programa a medida que crece y también, identificar posibles fuentes de error. Javadoc es una herramienta creada para tal fin. Está pensado para lograr que la arquitectura de la solución sea mucho más comprensible, es decir, su formato común hace que los comentarios escritos sean más comprensibles por otro programador [2][3][4].

Una adecuada documentación requiere agregar comentarios a todas las clases y métodos que componen el programa. Un comentario debe contener la información suficiente para que otras personas puedan entender lo que se ha realizado y el porqué de lo que se ha realizado [2][3][4].

Javadoc permite generar un conjunto de páginas web a partir de los archivos de código que contengan los delimitadores /**… */.

Cuando se documenta se debe expresar lo que no es evidente. Es decir, se debe explicar por qué se hacen las cosas, en lugar de repetir lo que se hace. Cuando se empiece la construcción del software es importante explicar:

- Objetivo de cada clase o paquete
- La función de cada método y uso esperado
- Explicar por qué se declaran las variables
- Exponer el funcionamiento del algoritmo que se está utilizando y las limitaciones del mismo
- Posibles mejoras que podrían realizarse

Se debe poner documentación javadoc en los siguientes casos:

- Al inicio de cada clase
- A cada atributo
- Al inicio de cada método

La escritura de un comentario Javadoc debe tener la siguiente sintaxis: iniciar por "/**" y finalizar con "*/":

```
/**
* Descripción clara y breve de lo que hace.
* @etiqueta texto para la etiqueta
*/
```

Una descripción de las posibles etiquetas a utilizar para documentar una clase se muestra a continuación:

@author	Nombre del autor
@version	Información sobre la versión y fecha
@see	Referencia con otras clases y métodos

En la documentación de los métodos y constructores se deben usar como mínimo las siguientes etiquetas:

[6] Universidad de Costa Rica – ECCI - 2008

Etiqueta	Seguida de…	Descripción
@param	Nombre del parámetro	Descripción, uso y valores válidos
@return	Si el método no es void	Descripción de lo que se debe devolver
@exception ó @throws	Nombre de la excepción	Excepciones que pueden surgir durante la ejecución

A continuación se ilustra el uso de algunas etiquetas Javadoc para documentar el código:

- Al inicio de la clase

```
/**
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * $Id$
 * Universidad del Quindío(Armenia- Colombia)
 * Programa  de Ingeniería de Sistemas y Computación
 * Licenciado bajo el esquema Academic Free License
 * Fundamentos de Algoritmia
 * Ejercicio: Empleado
 * @author Sonia Jaramillo Valbuena
 * @author Sergio Augusto Cardona
 * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 */
/**
 * Clase que representa un Empleado
 */

public class Empleado {
}
```

- Por cada método

```
/**
 * Es metodo inicia todo el objeto existente con los valores recibidos
 * @param id Es el id del empleado
 * @param nombre Es el nombre del empleado
 * @param direccion Es la direccion del empleado
 * @param ciudad Es la ciudad del empleado
 */
public void fijarPersona(String id, String nombre, String direccion, String ciudad)
{
	this.id = id;
	this.nombre = nombre;
	this.direccion = direccion;
	this.ciudad = ciudad;
}
```

Luego de finalizar la escritura de la documentación es posible generar un archivo html con la documentación. Los pasos para ello son:

Crear una carpeta para la documentación llamada docs. Esto se hace ubicándose sobre el proyecto y seleccionando new Folder. Dentro de docs cree carpeta specs.

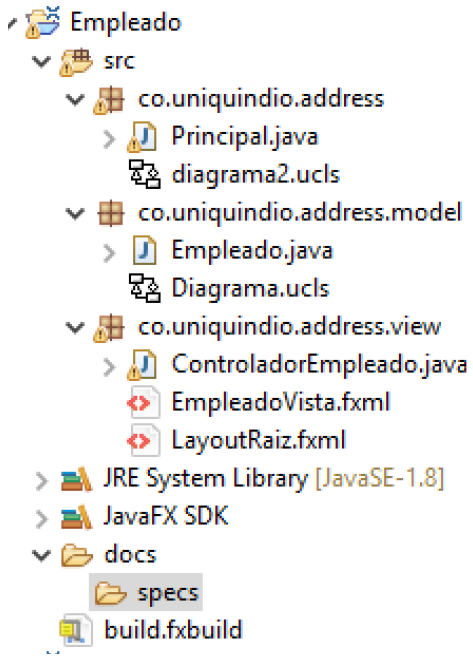


Figura 28 Carpetas para almacenar los archivos referentes a documentación

Luego se debe dar clic en la opción Project de la barra de menú principal y seleccionar la opción Generar Javadoc.

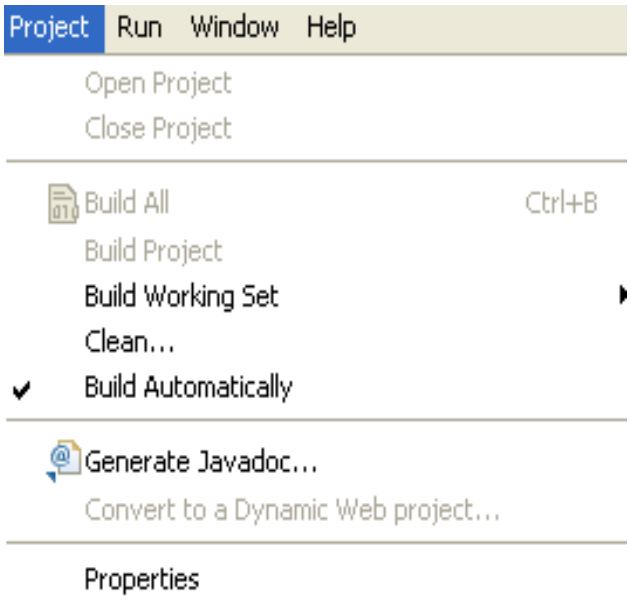


Figura 29 Generación de Javadoc

Luego deber verificar que en Javadoc Command esté la ruta de javadoc.exe. Es decir, C:\Program Files\Java\jdk1.8.0\bin\javadoc.exe. Active la Use Standard Doclet. Aquí debe estar seleccionada la carpeta docs del proyecto. Por último dar clic en Finish

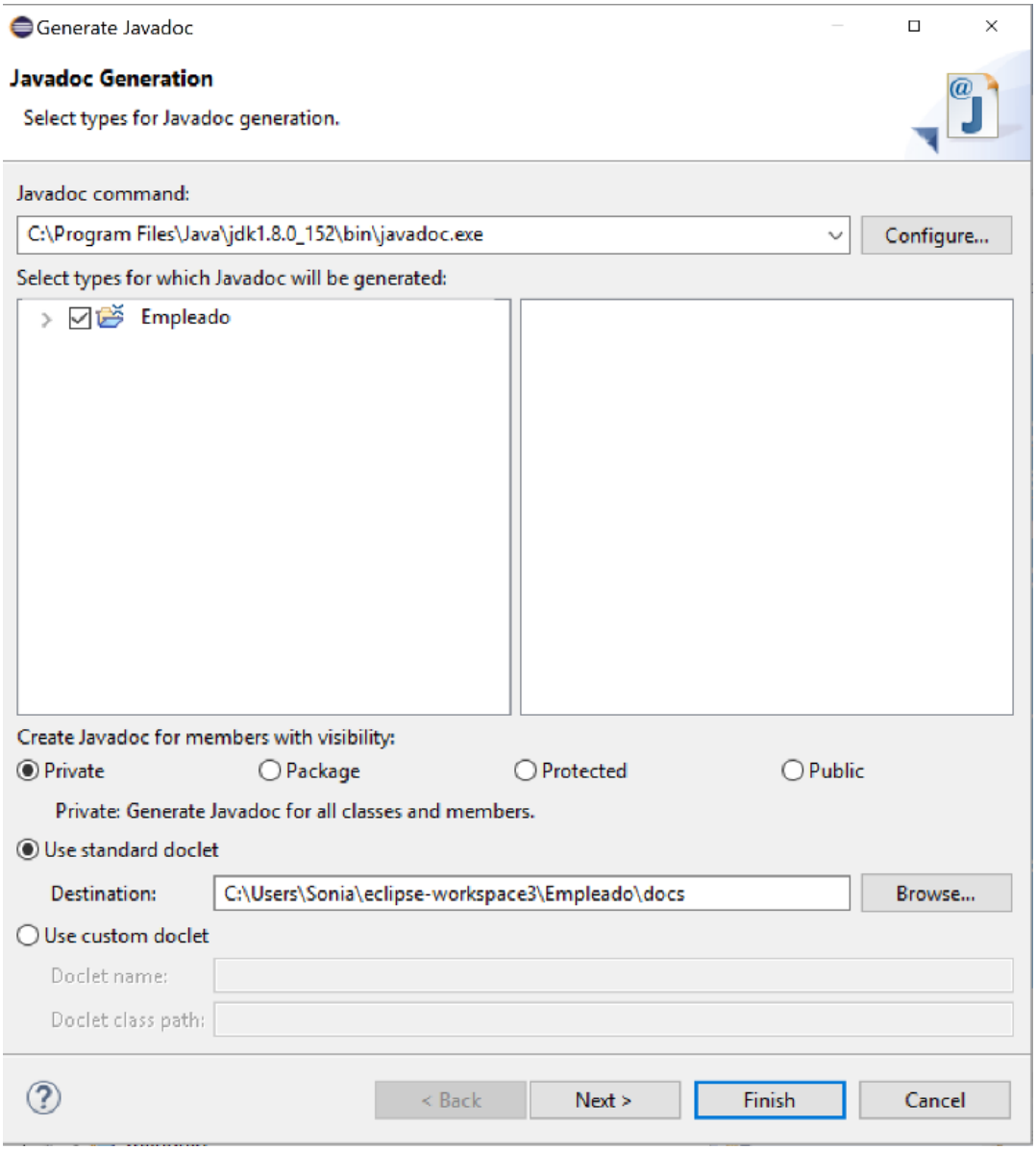


Figura 30 Destino de la documentación javadoc

1.10.7. Creación de un primer Proyecto en Eclipse con Java FX

Para crear un proyecto de clic en File → New→ Java→ Other→ Java FX Project.

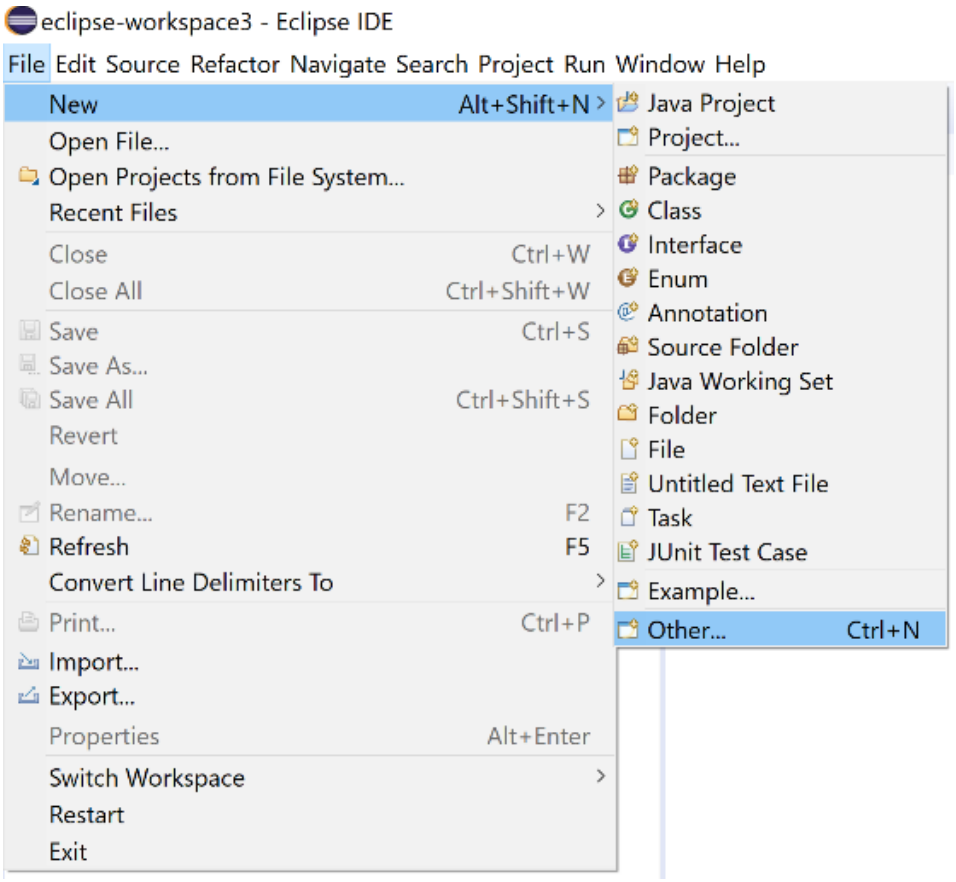


Figura 31 Crear un nuevo proyecto

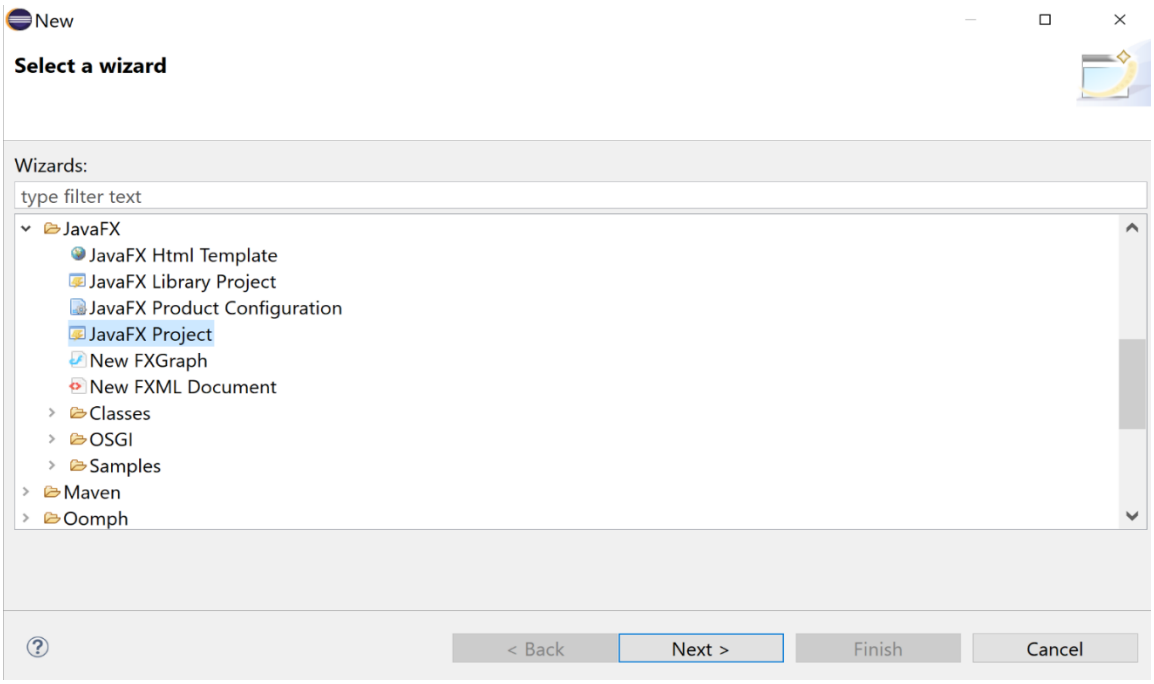


Figura 32 Selección JavaFX project

A continuación escriba el nombre del proyecto en project name

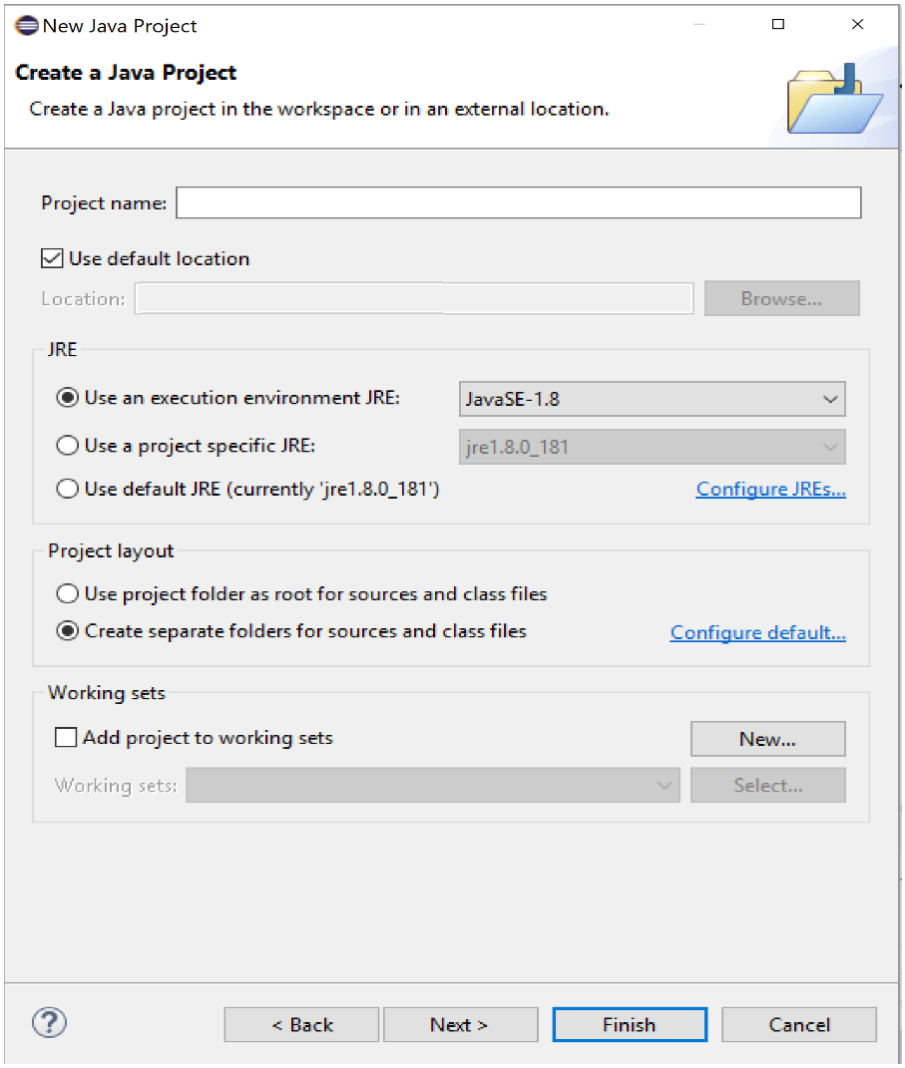


Figura 33 Dar nombre al proyecto

Luego active la opción Desktop.

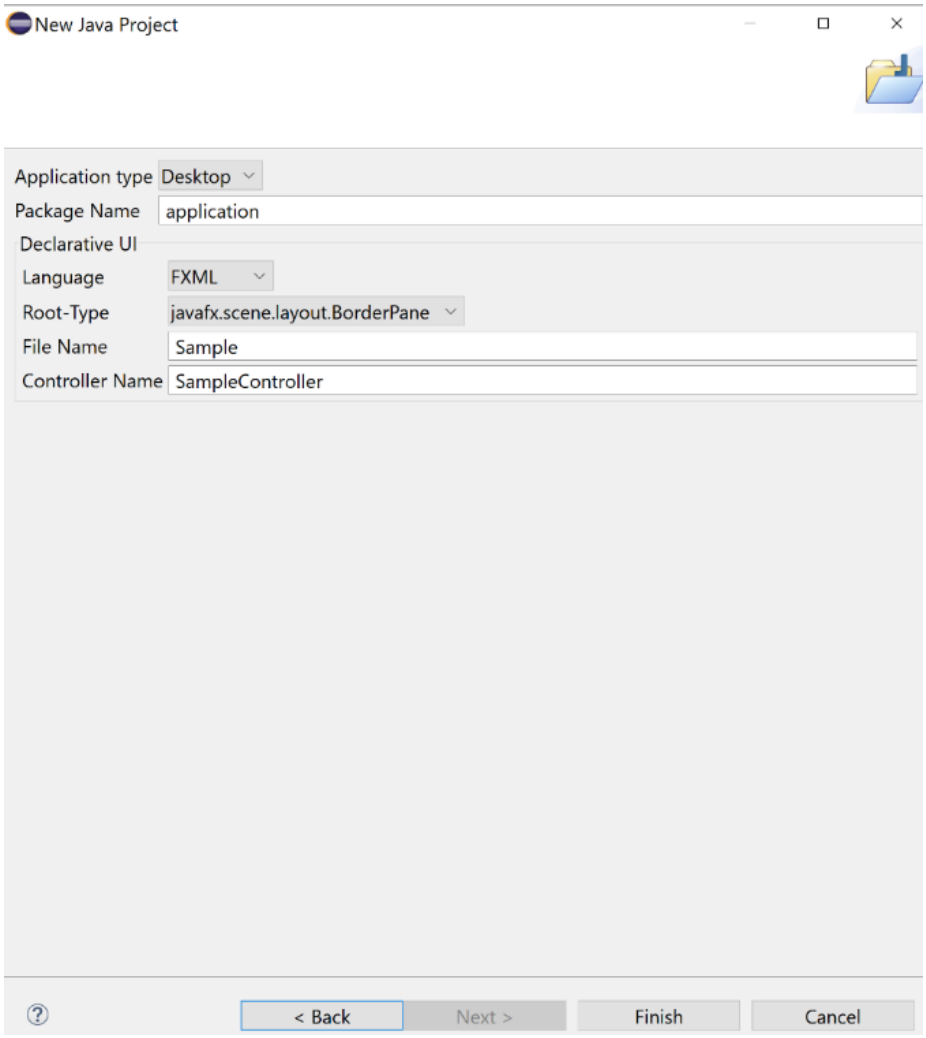


Figura 34 Selección opción Aplicación de escritorio

La estructura inicial del proyecto creado se presenta en la Figura 35.

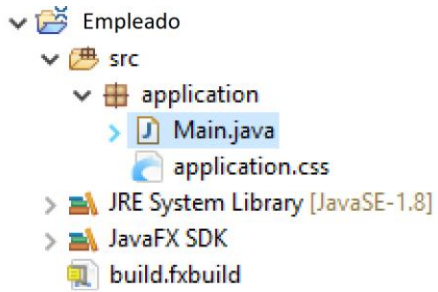


Figura 35 Paquetes del proyecto Empleado

Con el mouse de clic sobre el proyecto → *src*→ *New*→ *Package* . Construya los siguientes paquetes

co.uniquindio.address - contendrá la *aplicación principal* (C)

co.uniquindio.address.model - contendrá las clases del modelo (M)
co.uniquindio.address.view - contendrá las vistas (V) y los controladores

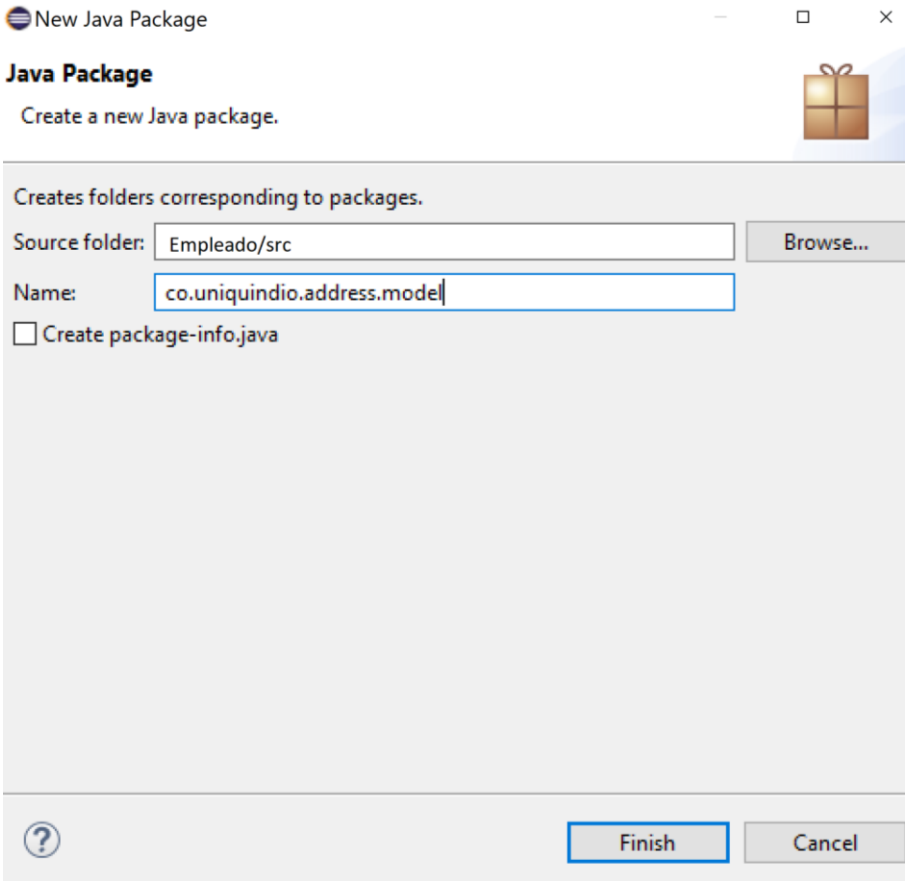


Figura 36 Pasos para crear un nuevo paquete

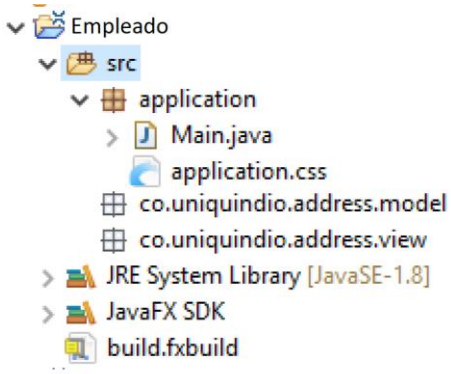


Figura 37 Estructura del proyecto luego de crear los paquetes

Dentro del paquete modelo construya la clase Persona. Para ubiquese en el paquete model, de clic derecho New→ Class

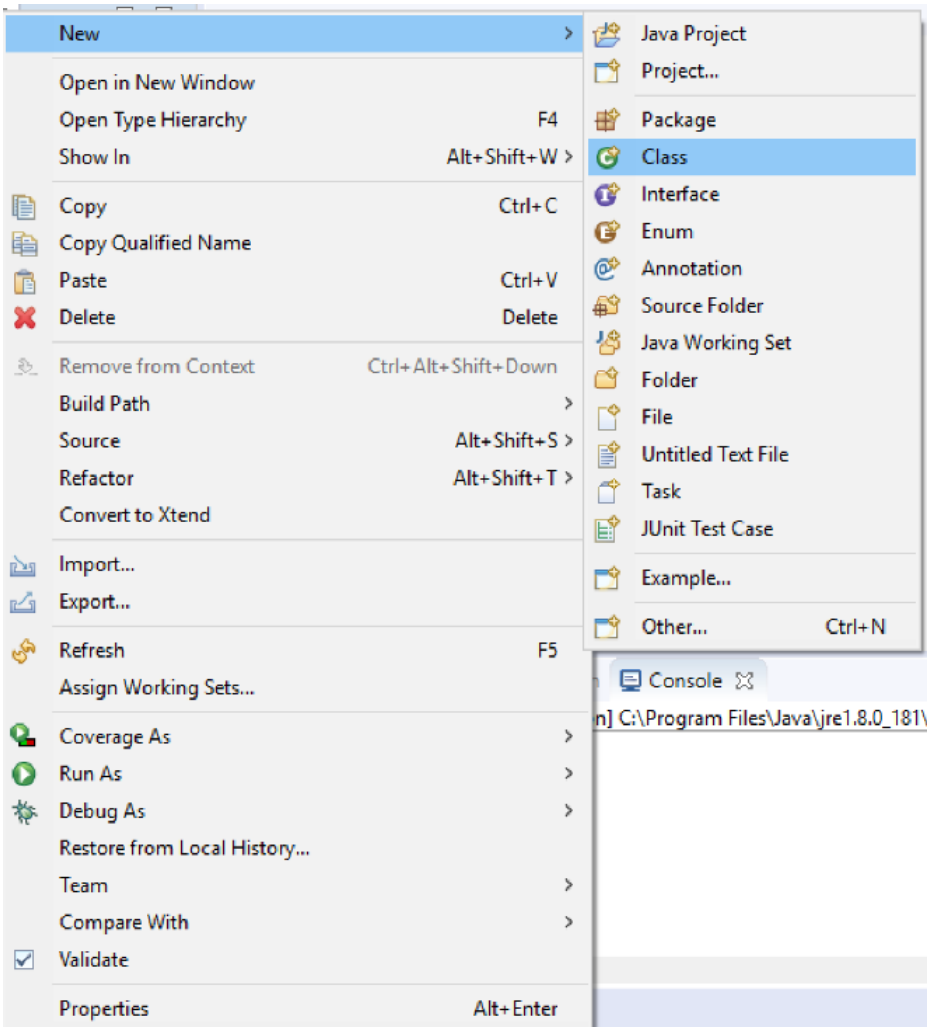


Figura 38 Creación de una clase

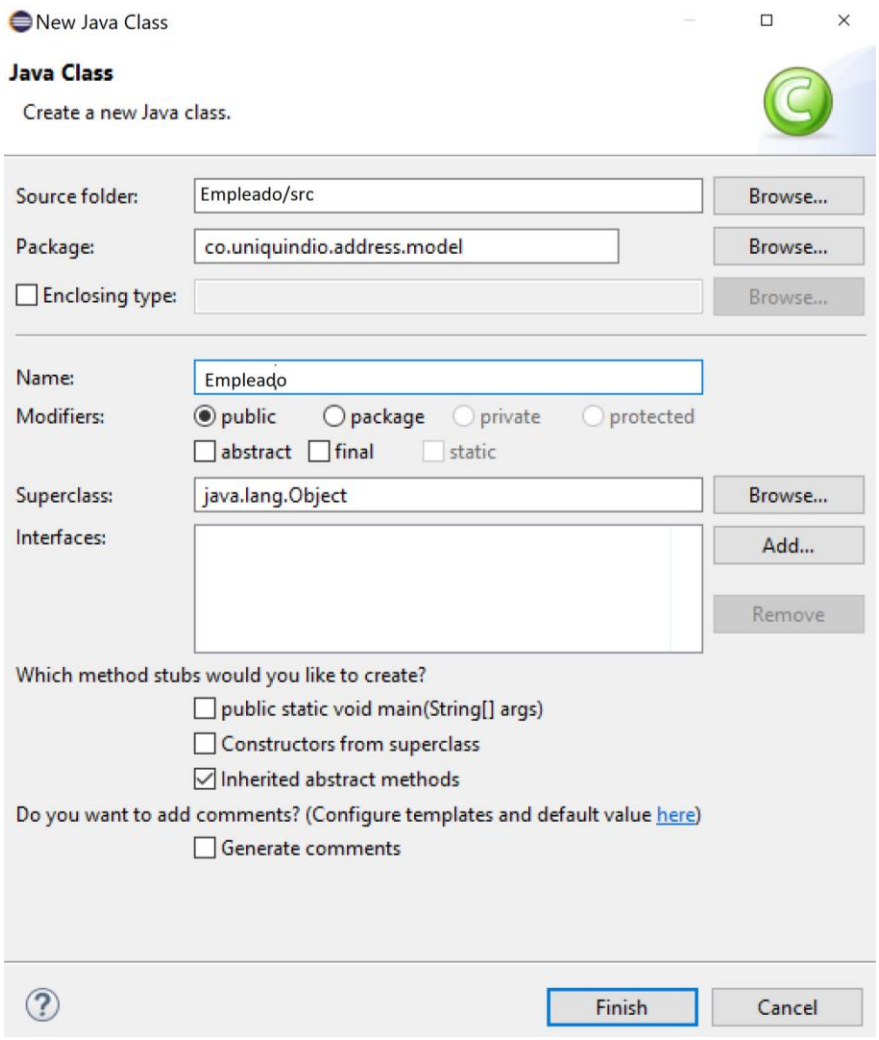


Figura 39 Dando nombre a la clase

Programa 2 Agregando atributos a la clase Empleado

```
package co.uniquindio.address.model;
/**
 * Clase para crear un empleado
 * @author Sonia
 *
 */
public class Empleado {
/**
 * Es el id del estudiante
 */
private String id;
/**
 * Es el nombre del estudiante
 */
private String nombre;
/**
 * Es la direccion del estudiante
```

Programa 2 Agregando atributos a la clase Empleado

```
 */
private String direccion;
/**
 * Es la ciudad de residencia del estudiante del estudiante
 */
private String ciudad;
```

Ahora agregue los métodos set y get necesarios, para ello de clic derecho en Source y luego seleccione Generate Getters and Setters.

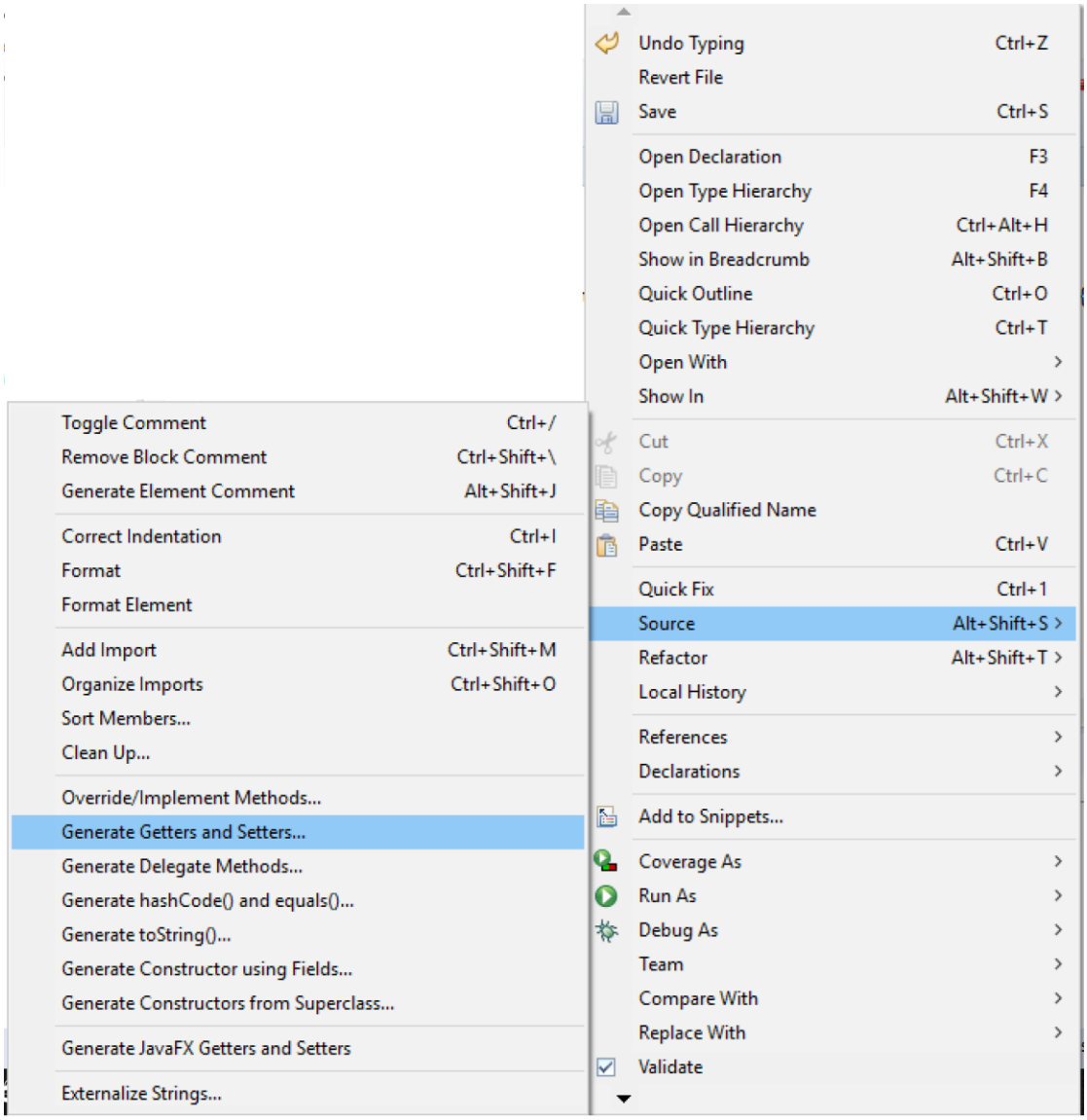


Figura 40 Generar métodos get y set

Finalmente agregue el siguiente constructor. Esta línea es opcional.

```
public Empleado()
{}
```

Este es el código resultante:

Programa 3 *Clase Empleado completa*

```
package co.uniquindio.address.model;
/**
 * Clase para crear un empleado
 * @author Sonia
 *
 */
public class Empleado {
/**
 * Es el id del estudiante
 */
private String id;
/**
 * Es el nombre del estudiante
 */
private String nombre;
/**
 * Es la direccion del estudiante
 */
private String direccion;
/**
 * Es la ciudad de residencia del estudiante del estudiante
 */
private String ciudad;
/**
 * Metodo constructor, en este caso puede omitirse su construccion
 */
public Empleado()
{}
/**
 * Metodo modificador
 * @param id, es el id del empleado
 */
public void setId(String id) {
	this.id = id;
}
/**
 * Metodo modificador
 * @param nombre, Es el nombre del empleado
 */
public void setNombre(String nombre) {
	this.nombre = nombre;
}
/**
 * Metodo modificador
 * @param direccion, la direccion del empleado
 */
public void setDireccion(String direccion) {
	this.direccion = direccion;
}
```

Programa 3 *Clase Empleado completa*

```
/**
 * Metodo modificador
 * @param ciudad, la ciudad de residencia
 */
public void setCiudad(String ciudad) {
	this.ciudad = ciudad;
}
/**
 * Metodo accesor
 * @return el nombre del estudiante
 */
public String getNombre() {
	return nombre;
}

/**
 * Es metodo inicia todo el objeto existente con los valores recibidos
 * @param id Es el id del empleado
 * @param nombre Es el nombre del empleado
 * @param direccion Es la direccion del empleado
 * @param ciudad Es la ciudad del empleado
 */
public void fijarEmpleado (String id, String nombre, String direccion, String ciudad)
{
	this.id = id;
	this.nombre = nombre;
	this.direccion = direccion;
	this.ciudad = ciudad;
}

/**
 * Devuelve la representacion en String del empleado
 */
public String toString()
{
return id+"  "+nombre +"   "+direccion+"   "+ciudad;
}
}
```

Ahora se procede a crear la vista. Para ello de clic en New → Other→ JavaFX → FXML → New FXML Document y dele el nombre de PersonaVista.

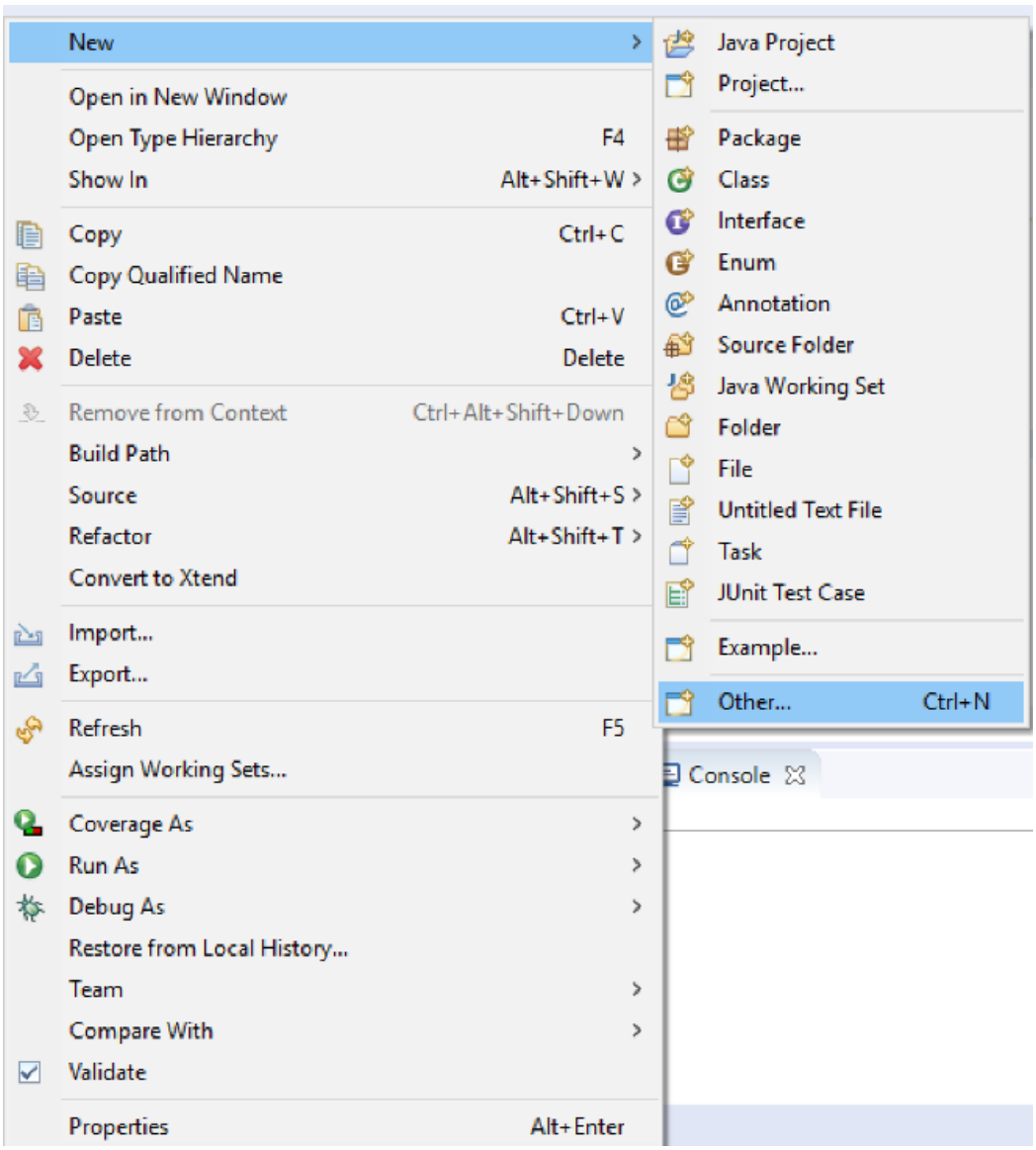


Figura 41 Pasos para construir un nuevo elemento

Elija AnchorPane.

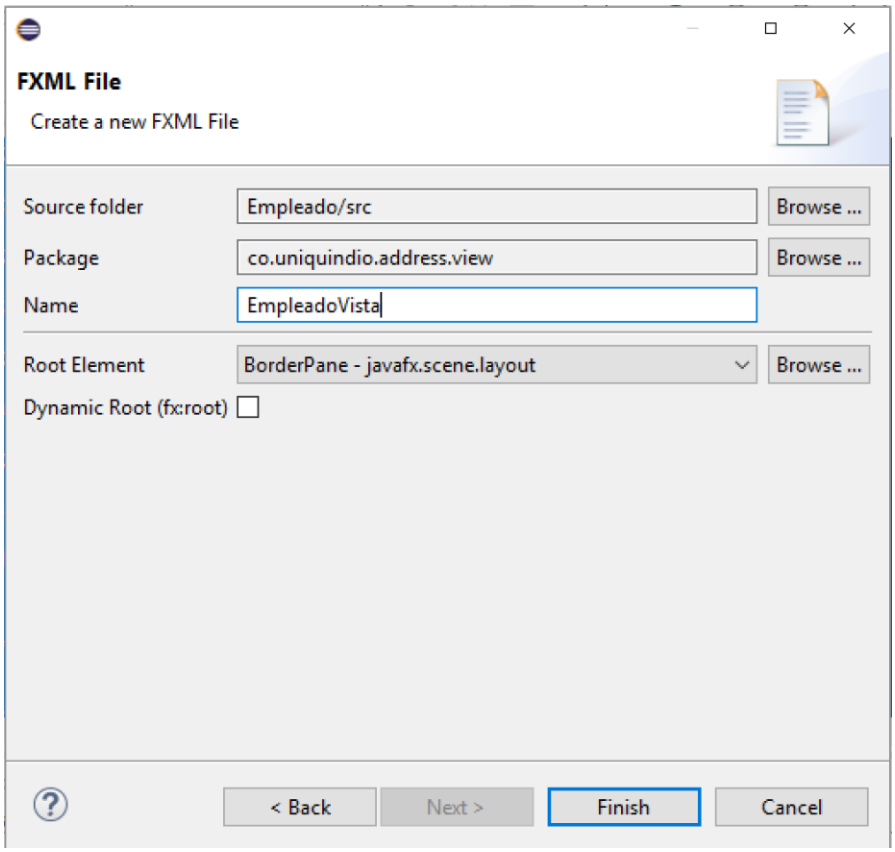


Figura 42 Configuración del archivo FXML

De clic derecho sobre el archivo generado y seleccione abrir con SceneBuilder.

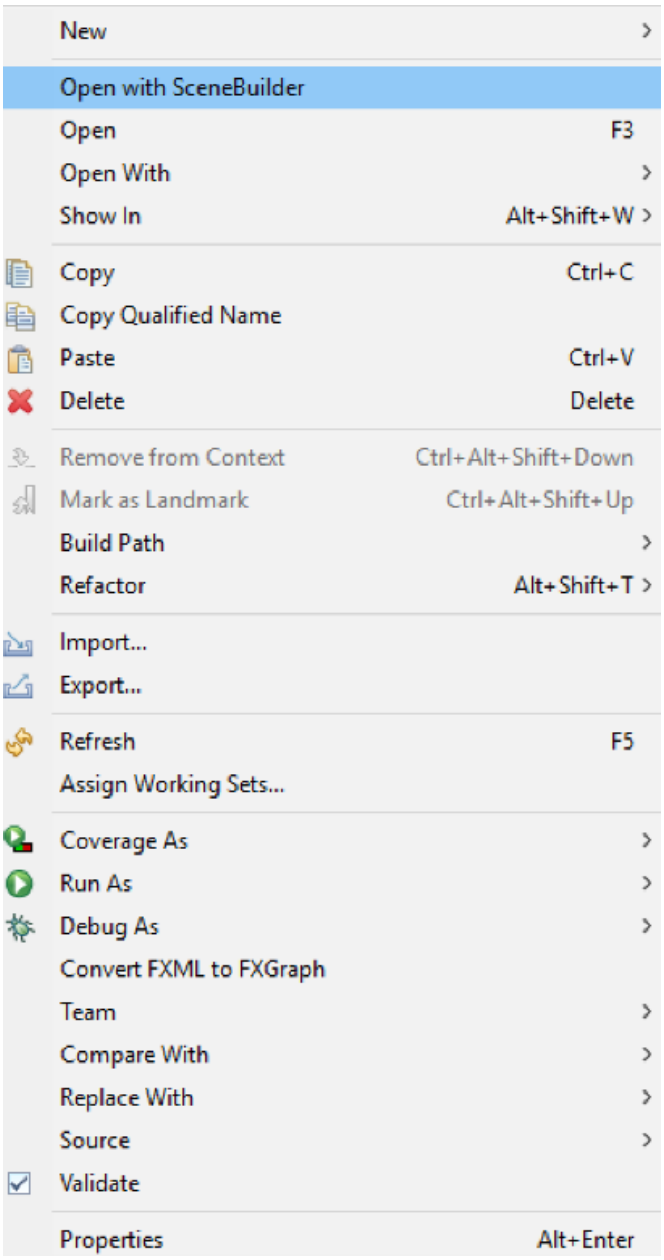


Figura 43 Abrir con el Scene Builder

De clic al lado izquierdo sobre AnchorPane. En la pantalla aparecerá una cruz → de clic sobre ella y con el mouse amplie el cuadro.

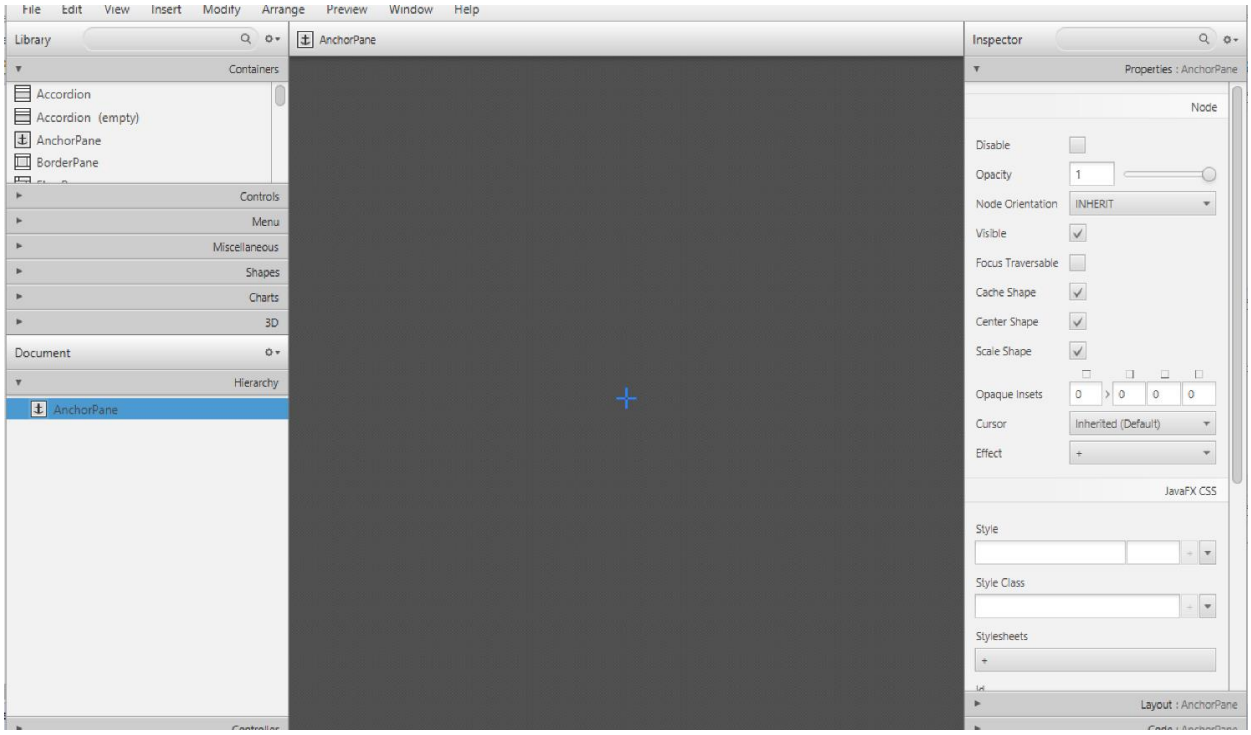


Figura 44 *AnchorPane*

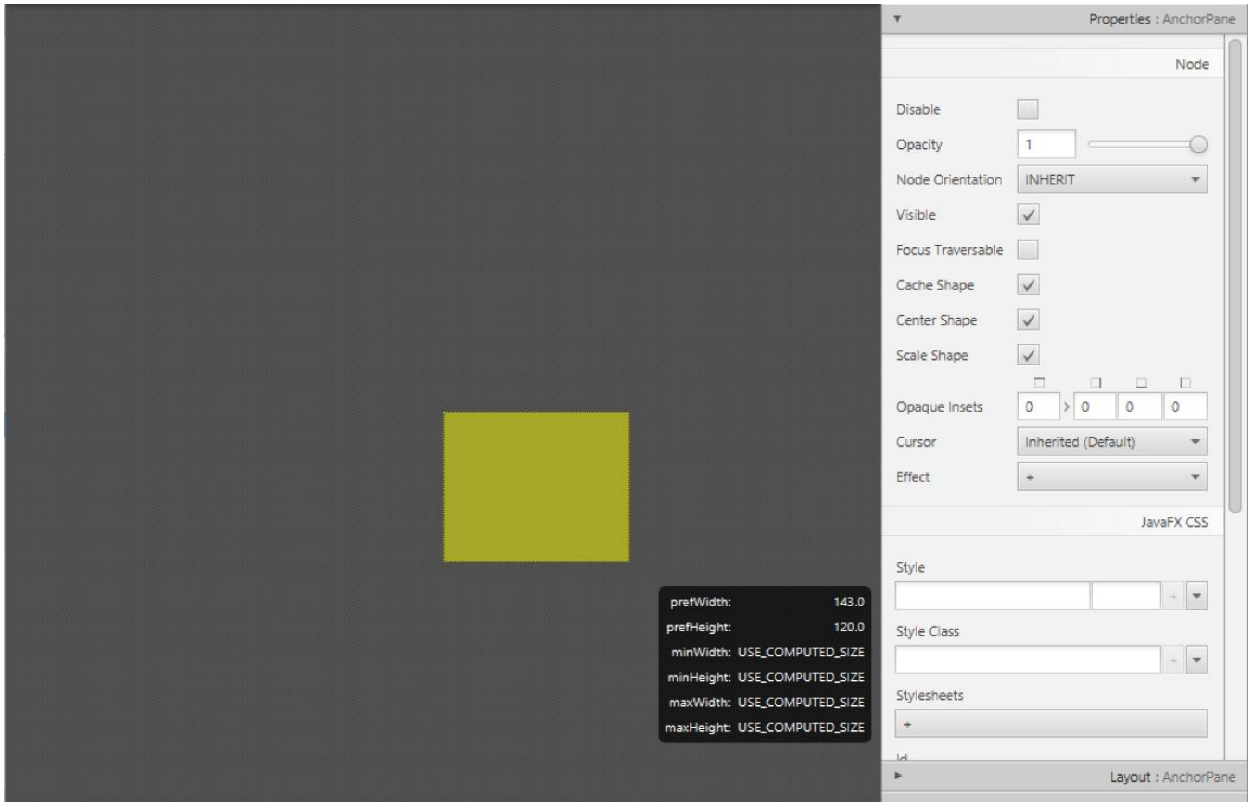


Figura 45 Ampliación del AnchorPane

Observe que al lado derecho, está el menú Inspector. Allí se despliegan 3 opciones: Properties, Layout y Code. Mediante la opción Layout podrá también, ajustar el tamaño de la forma, ver Figura 46.

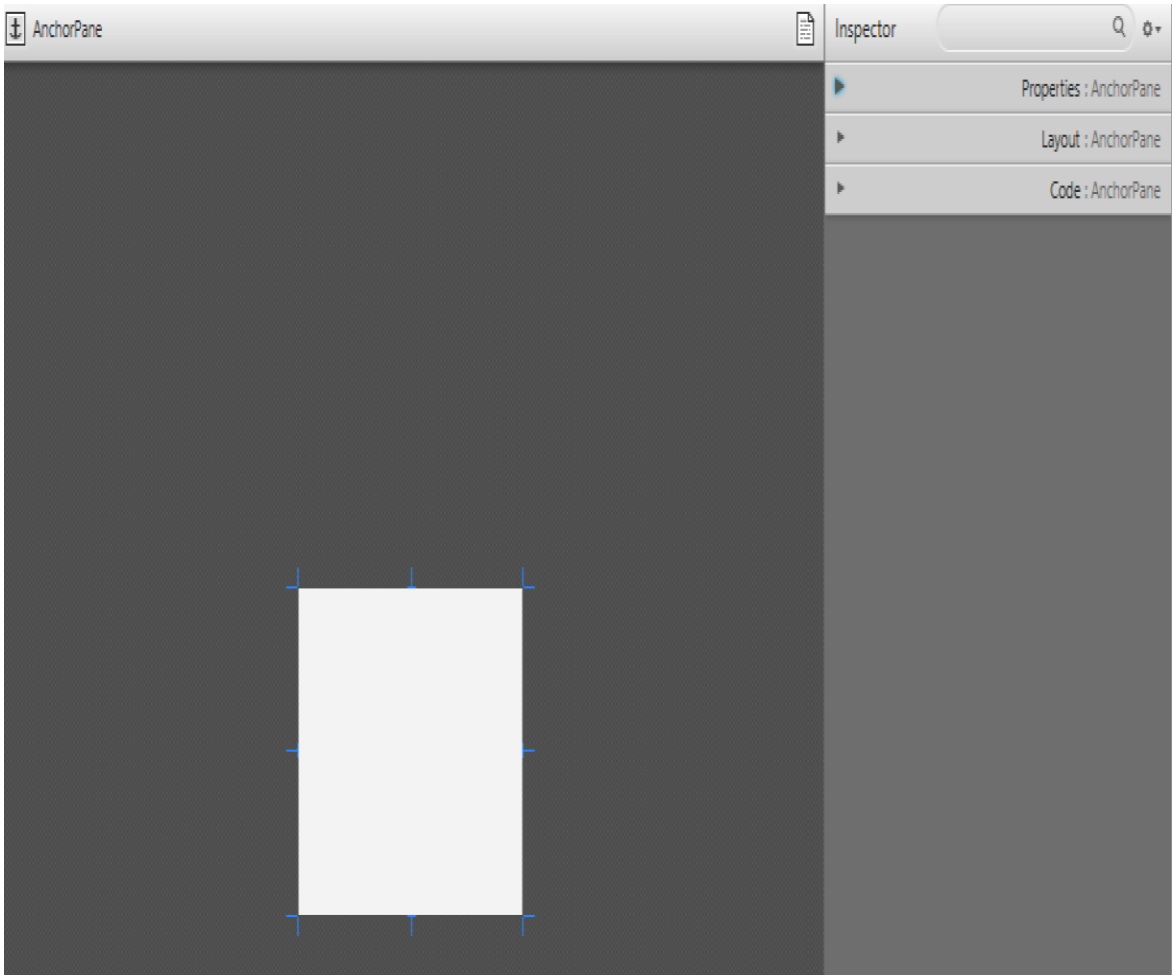


Figura 46 Inspector

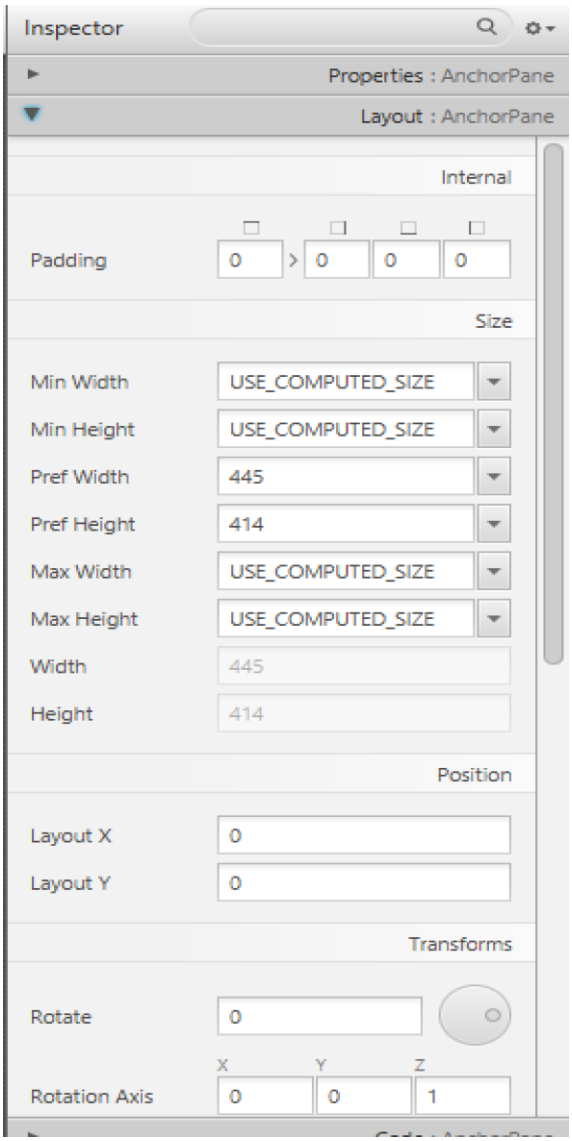


Figura 47 Configuración del tamaño de la forma

Agregue una etiqueta al contenedor. Para ello puede buscar el componente al lado izquierdo, junto a la opción Library. De clic sobre el componente encontrado y arrástrelo, ver Figura 48 y Figura 49.

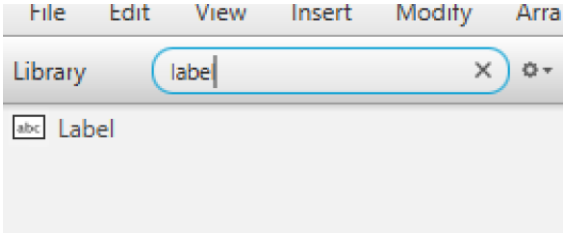


Figura 48 Búscar un componente

Figura 49 Agregar una etiqueta

Modifique el texto dando clic sobre el componente, agregue el texto, tal como se muestra a continuación .

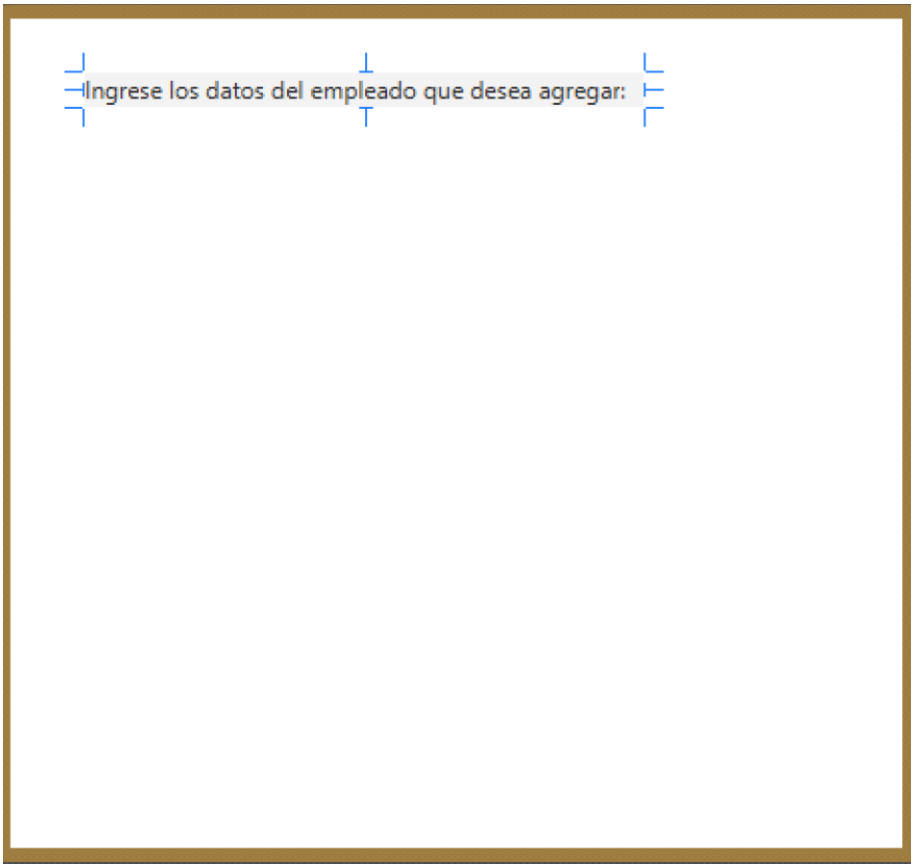


Figura 50 Cambiar el texto de una etiqueta

Ahora agregue un GridPane. Puede hacer uso de la opción Library para buscar el componente.

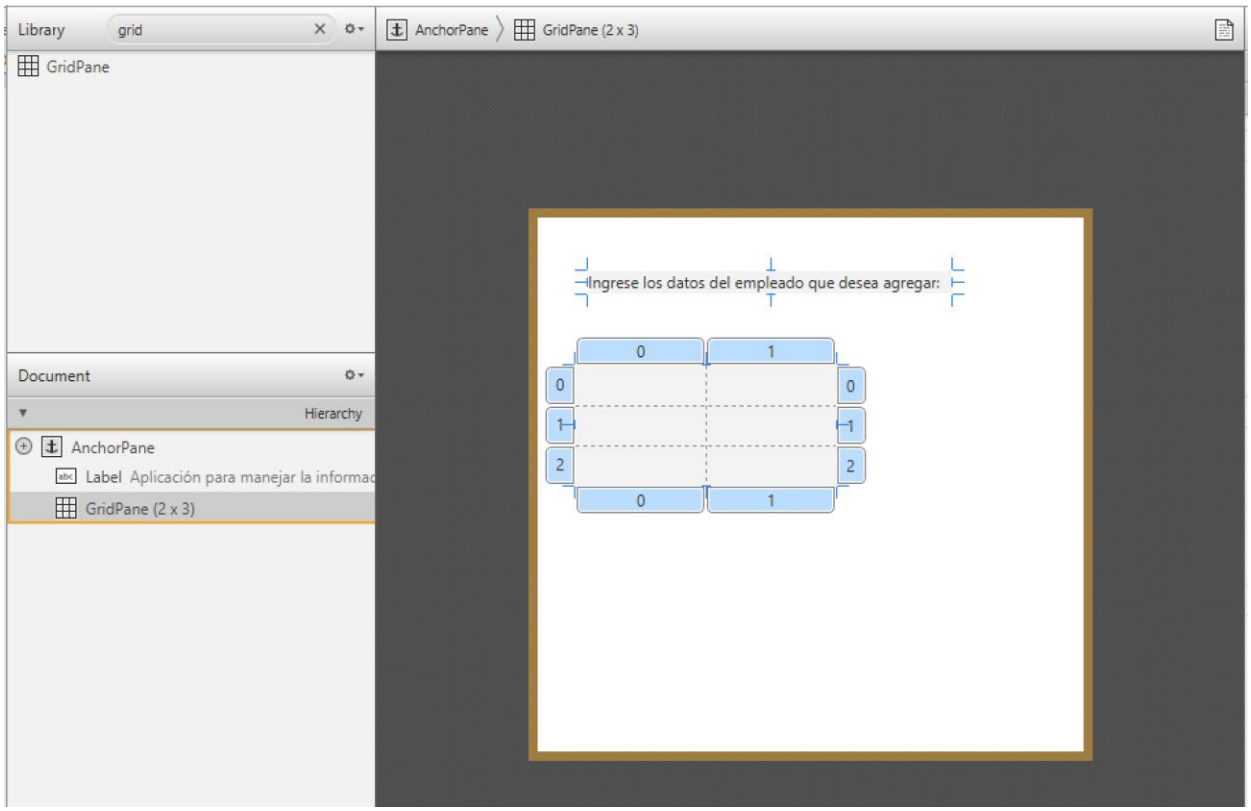


Figura 51 Agregar un GridPane

La tabla tendrá tantas filas como atributos tenga el empleado. Recuerde que los atributos son id, nombre, direccion, ciudad. Para agregar una fila debe dar clic sobre una fila hasta que se ponga amarilla y luego elegir, adicionar una Fila.

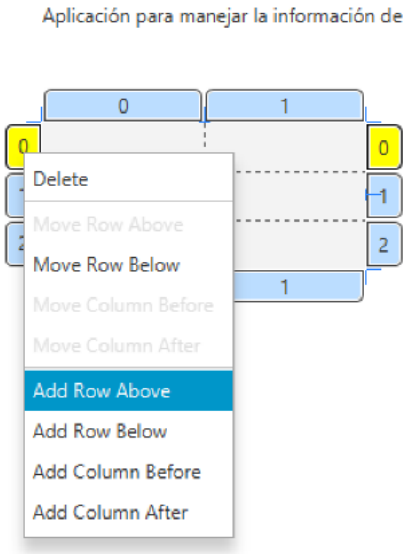


Figura 52 Adicionar una fila

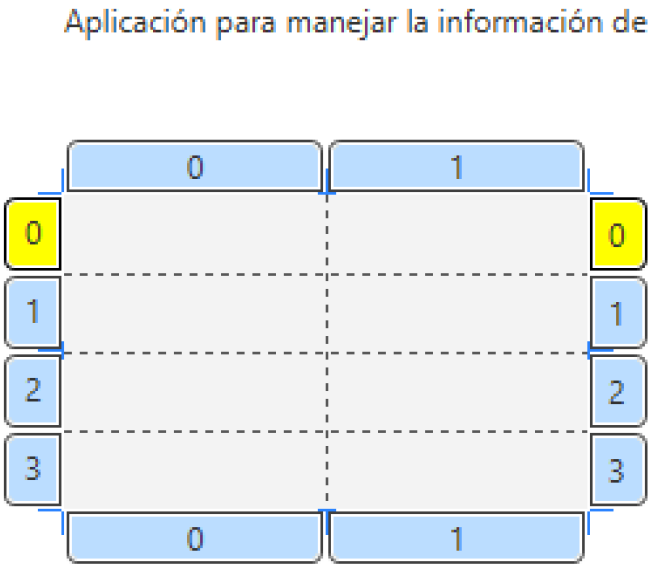


Figura 53 Cantidad de filas del GridPane

Ahora agregue etiquetas a la izquierda de la tabla y Textfield a la derecha. Recuerde buscar Label y Textfield, en el menú de la izquierda. Modifique el texto en las etiquetas de la izquierda, acorde a los atributos que se desean solicitar, ver Figura 54.

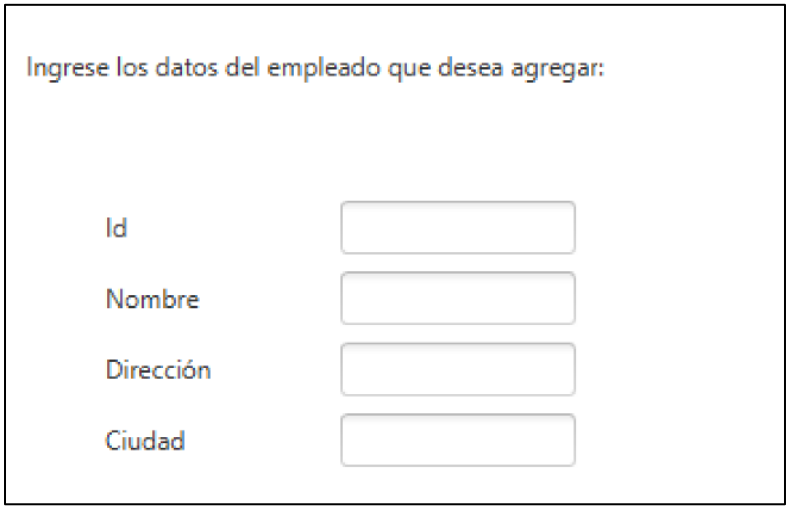


Figura 54 Elementos a agregar en el GridPane

Agregue dos botones al contenedor, el primero con el texto Guardar y el segundo con Mostrar nombre.

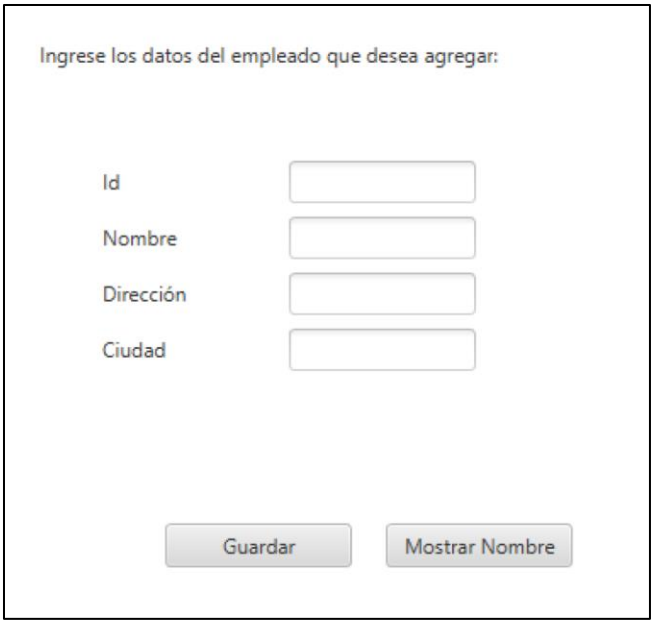


Figura 55 Interfaz con todos los elementos gráficos requeridos

Seleccione los 2 botones y agrúpelos en un panel vertical, tal como se muestra en Figura 56.

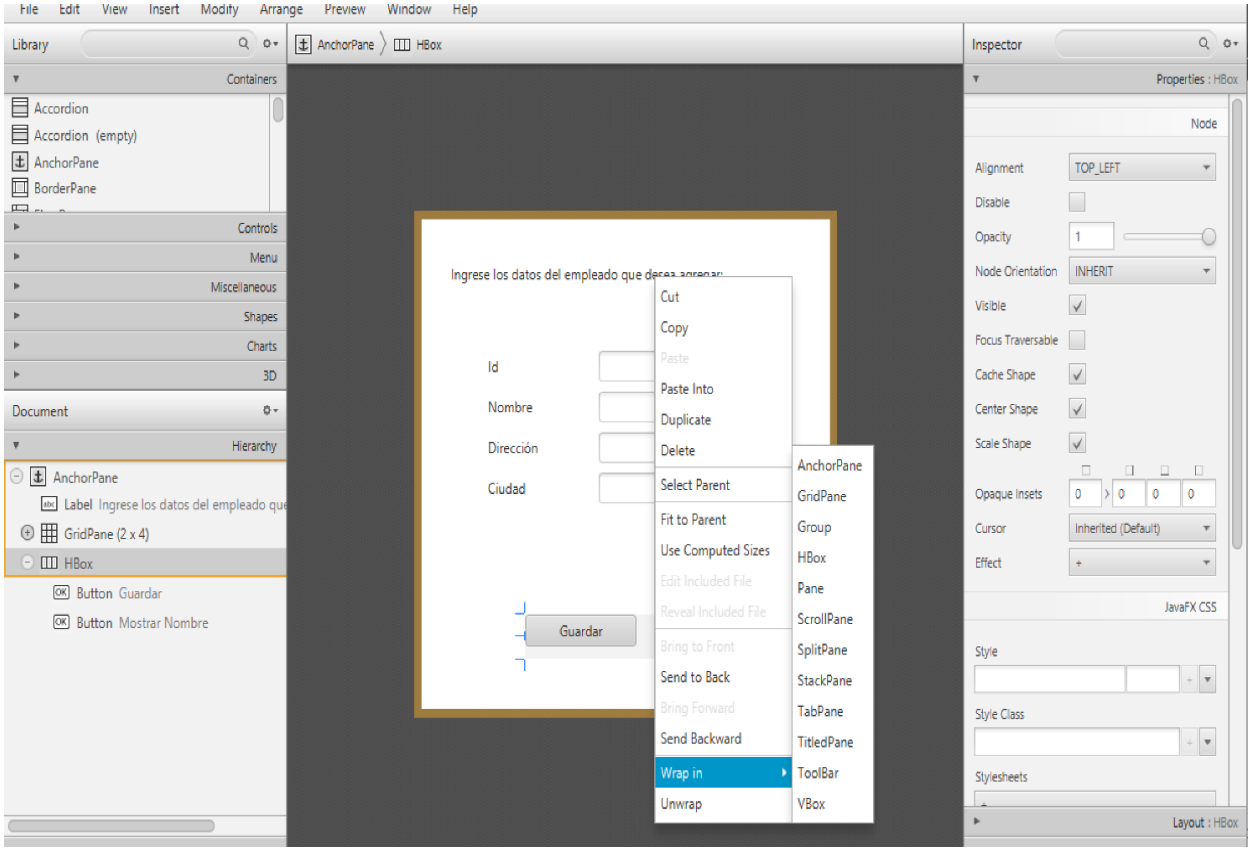


Figura 56 Agrupar en un HBOX

A la izquierda de la pantalla, de clic sobre GridPane. Esto provocará que a la derecha de la pantalla aparezca una configuración de anclaje. Modifique el valor de la derecha a 80. Ancle también el botón.

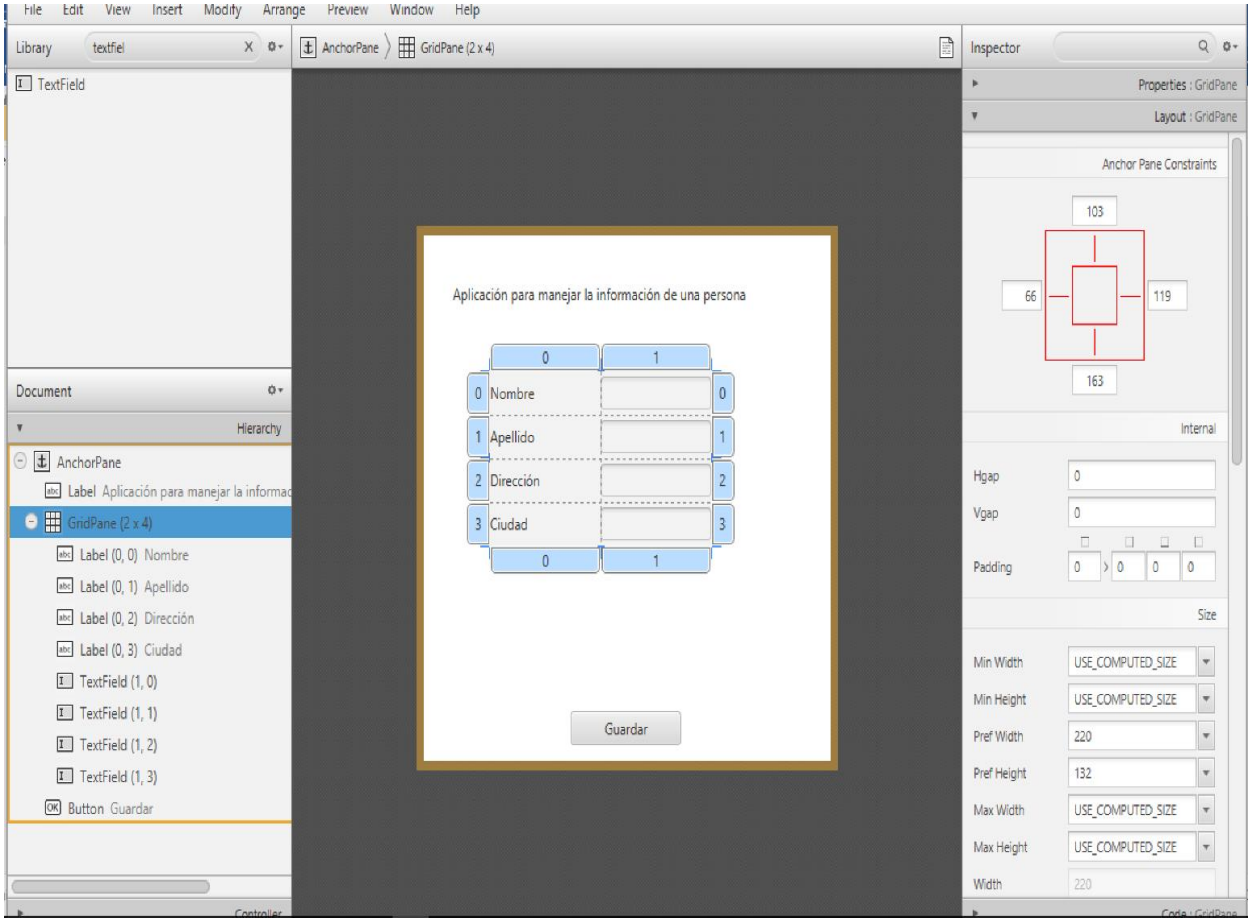


Figura 57 Anclar componentes

El tamaño de la tabla puede ser modificado de la esquina superior izquierda o de la esquina inferior derecha de la misma. Puede anclar el Hbox, mediante el menú Layout.

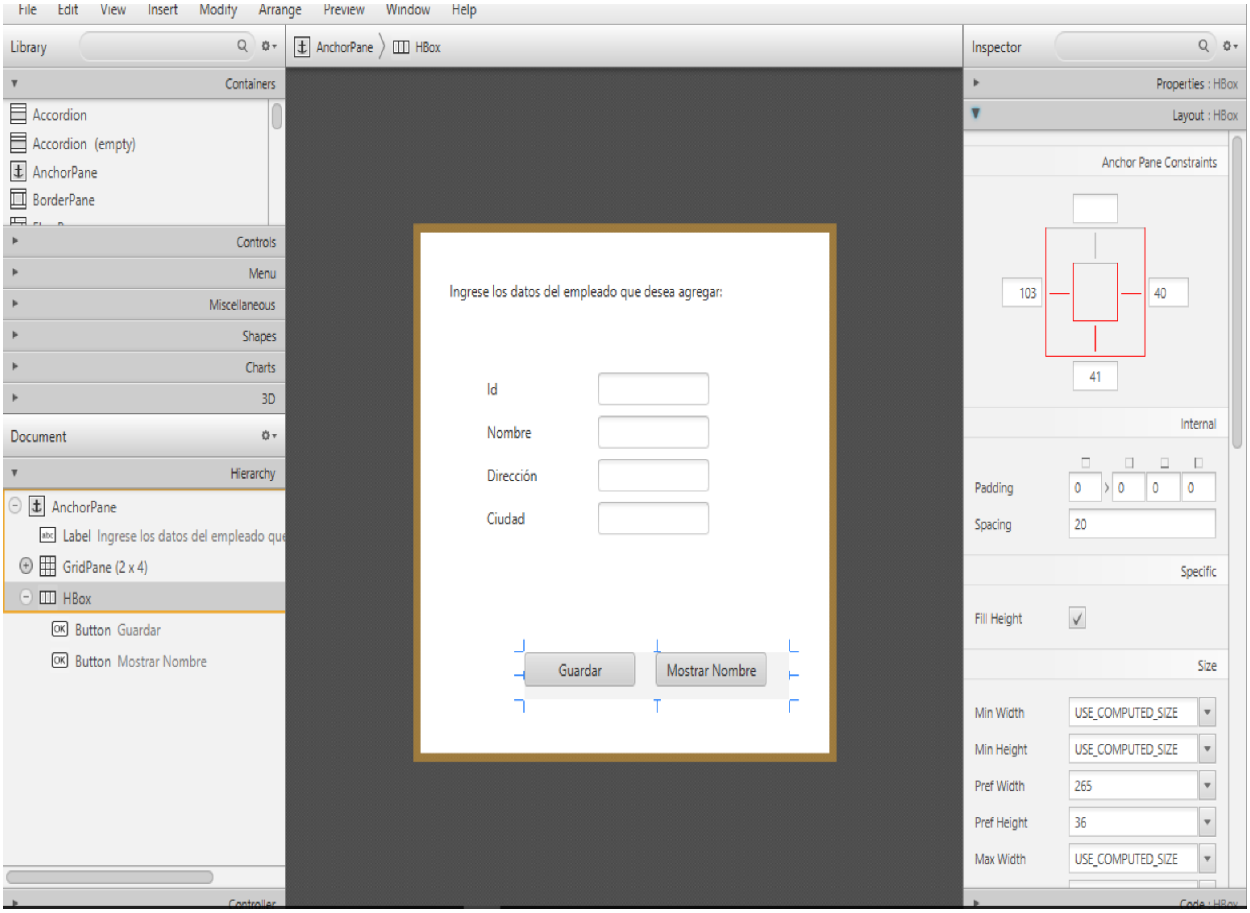


Figura 58 Anclado del HBox

Una vista previa de la pantalla puede verse mediante Preview → Show Preview View.

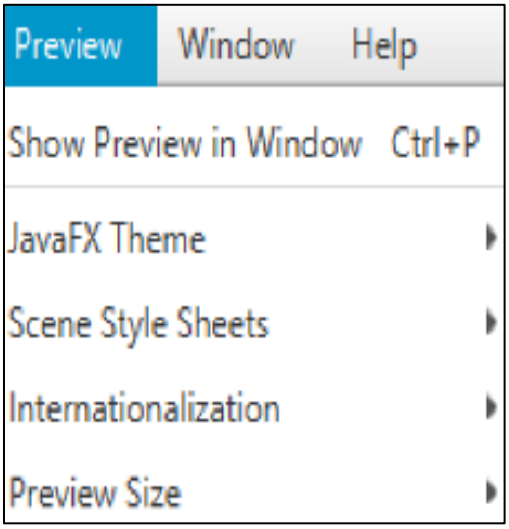


Figura 59 Vista Previa

Finalmente de clic en File → Save para guardar el archivo.

Ahora cree otro archivo FXML en el paquete view, nómbrelo LayoutRaiz.fxml. Elija *BorderPane* como layout.

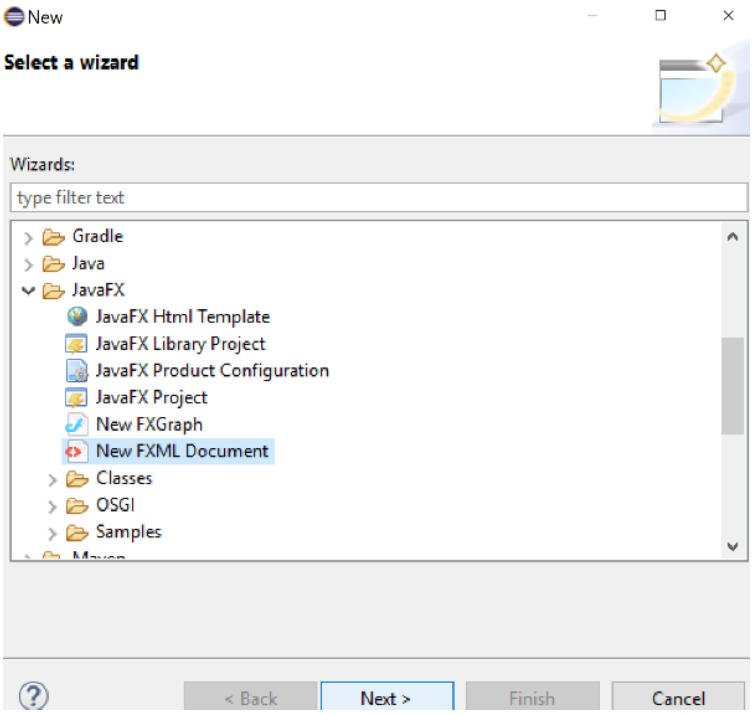


Figura 60 Nuevo archivo FXML

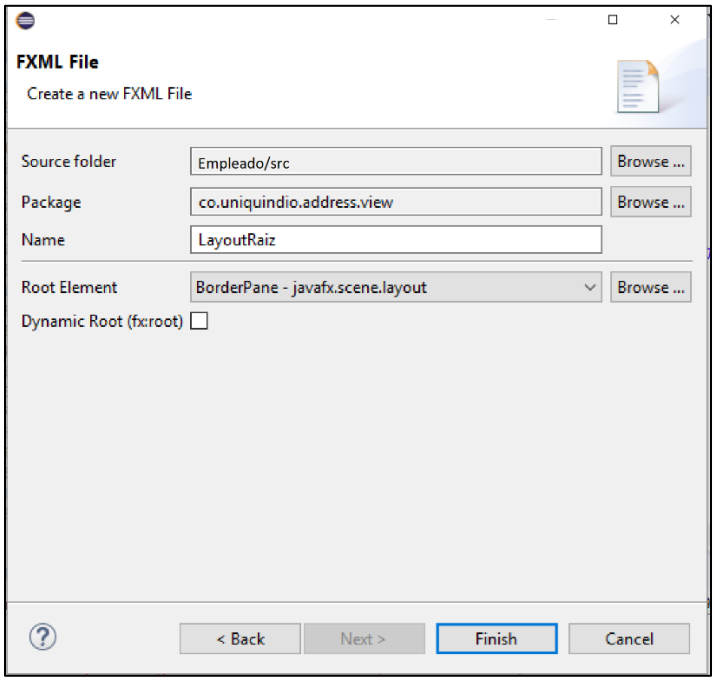


Figura 61 Dando nombre al archivo

El código generado se presenta en continuación:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.BorderPane?>

<BorderPane xmlns:fx="http://javafx.com/fxml/1">
      <!-- TODO Add Nodes -->
</BorderPane>
```

Abra el archivo LayoutRaiz.fxml con el Scene Builder.

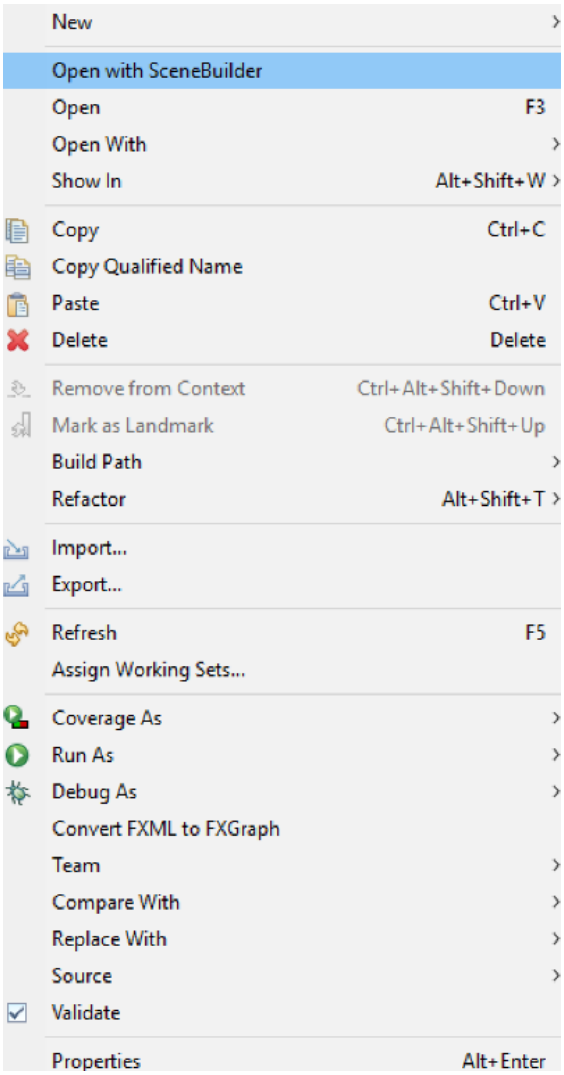


Figura 62 Abrir el archivo con SceneBuilder

Modifique el tamaño del *BorderPane* para que quede con el mismo tamaño de la ventana construida anteriormente, use las propiedades *Pref Width* y *Pref Height*, en la opción layout, ubicada a la derecha.

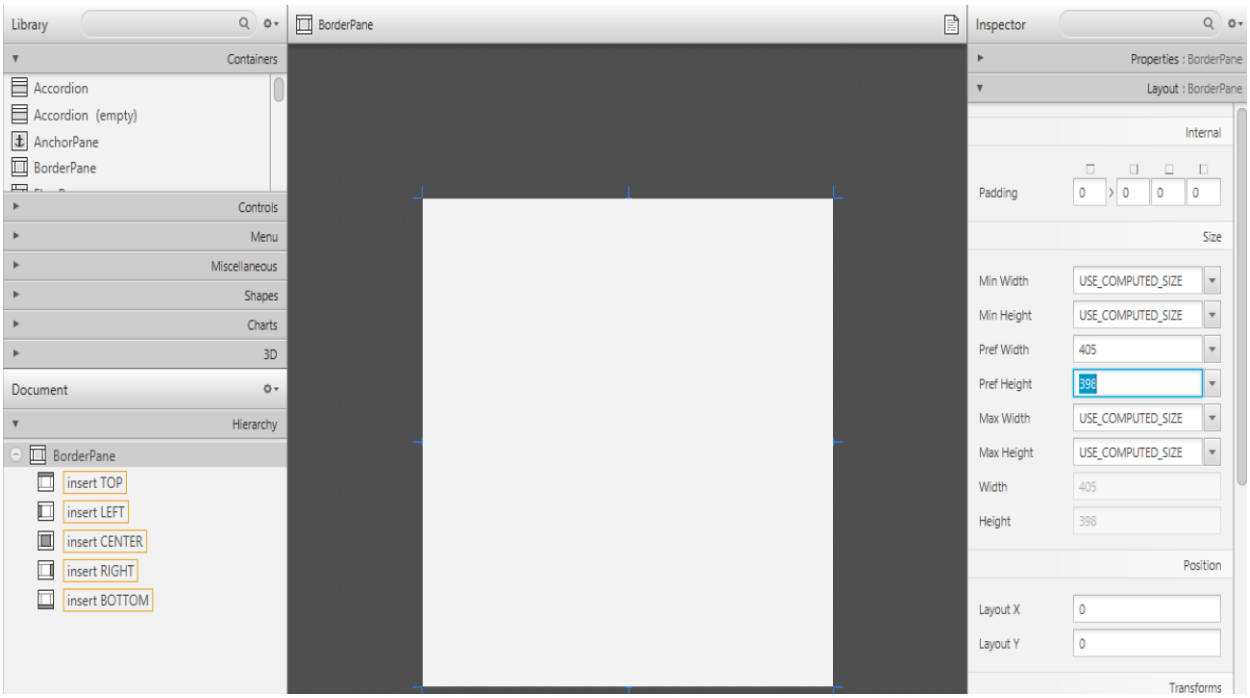


Figura 63 Configurar ancho y alto

Agregue un MenuBar al contendor, en la parte superior *del* BorderPane.

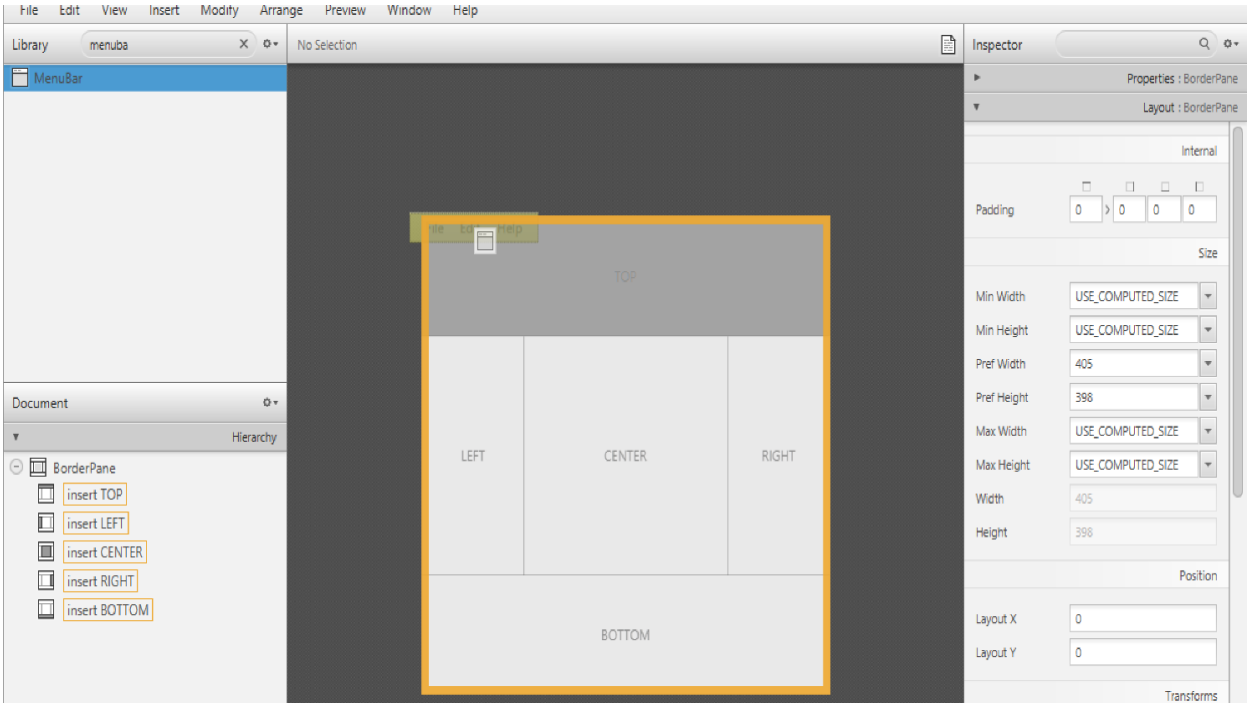


Figura 64 MenuBar

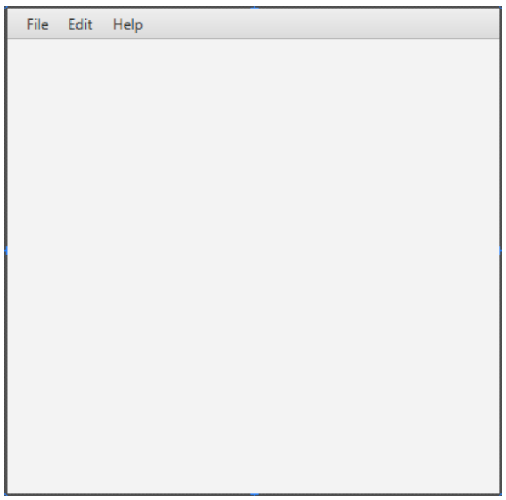


Figura 65 Opciones por defecto del MenuBar

Proceda a crear la clase principal. Para ello, de clic derecho en el proyecto, elija *New→ Other →JavaFX→ classes →JavaFX Main Class*.

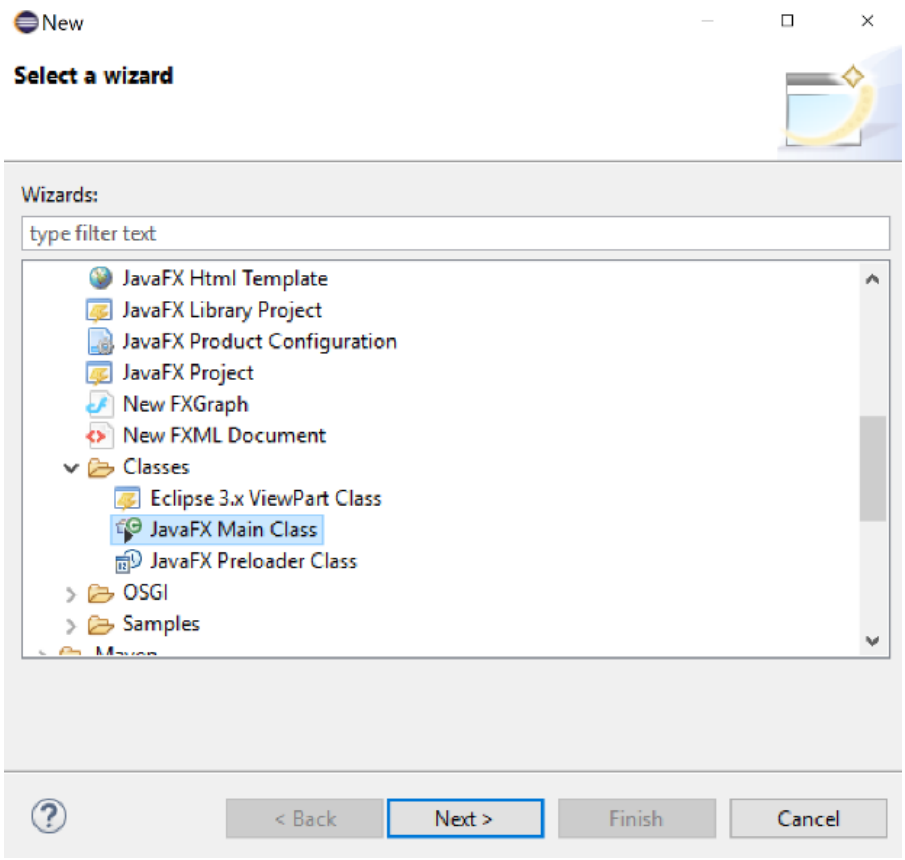


Figura 66 Clase principal

Dele nombre Principal y seleccione el paquete address.

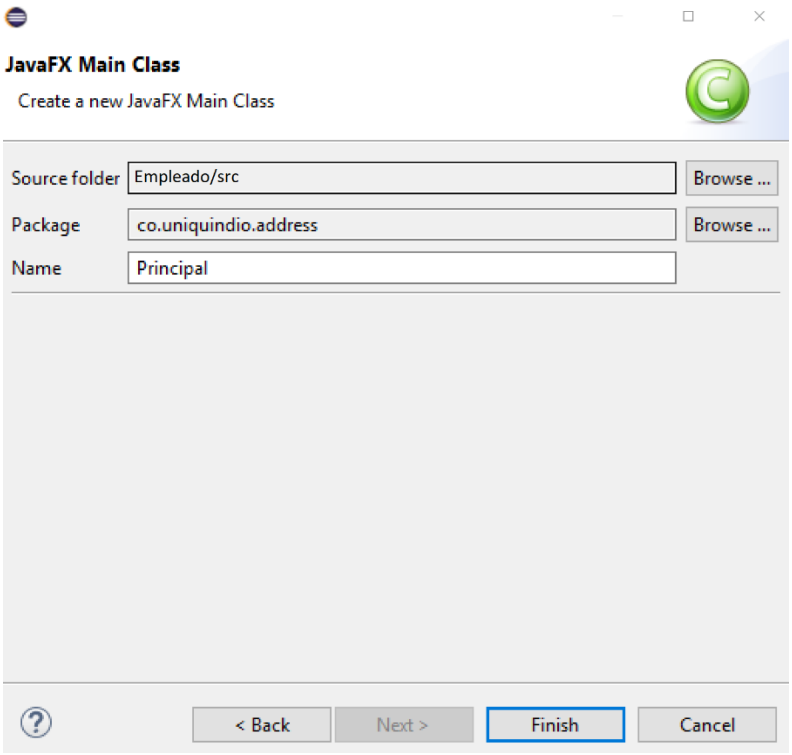


Figura 67 Clase Principal

El código generado se presenta a continuación:

Programa 4 Aplicación Principal

```
package co.uniquindio.address;

import javafx.application.Application;
import javafx.stage.Stage;

public class Principal extends Application {

	@Override
	public void start(Stage primaryStage) {

	}

public static void main(String[] args) {
	 launch(args);
    }
}
```

Observe que Principal extiende de Application. Hay dos métodos, start y main. El main es el que lanza la aplicación. El método start es invocado automáticamente cuando la aplicación es ejecutada. Observe que tiene un parámetro primaryStage, de tipo Stage.

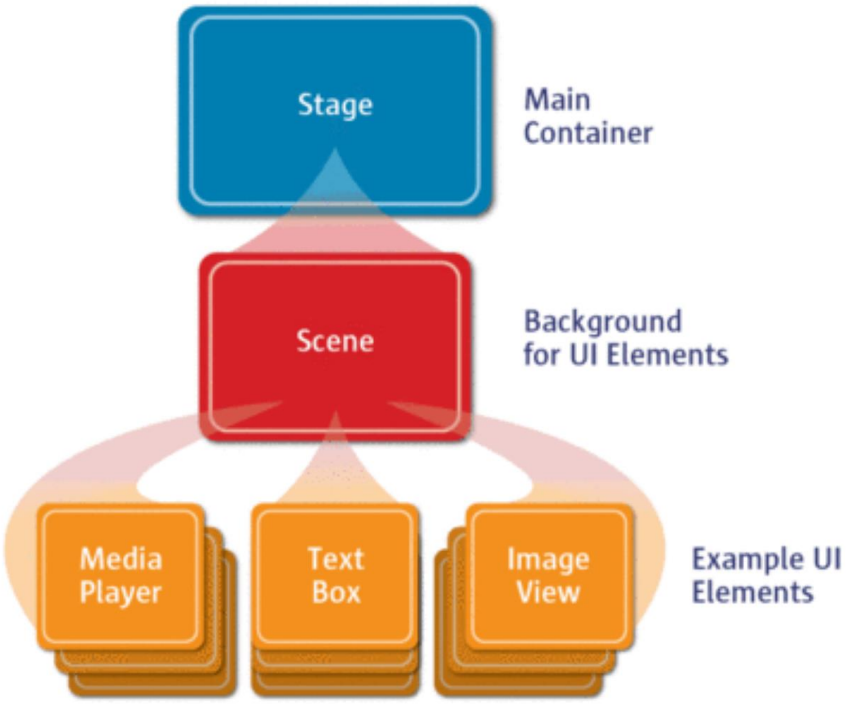


Figura 68 Java FX, *fuente* http://www.oracle.com

Programa 5 Clase Principal completa

```
package co.uniquindio.address;
import java.io.IOException;
import co.uniquindio.address.model.Empleado;
import co.uniquindio.address.view.ControladorEmpleado;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Principal extends Application {

private Stage escenarioPrincipal;
    private BorderPane layoutRaiz;
    private Empleado miEmpleado;
    @Override
    public void start(Stage primaryStage) {
      miEmpleado=new Empleado();
       this.escenarioPrincipal = primaryStage;
       this.escenarioPrincipal.setTitle("Aplicación sobre un empleado");
       inicializarLayoutRaiz();
```

Programa 5 Clase Principal completa

```
        mostrarVistaPersona();
    }

    /**
     * Inicializa el layout raiz
     */
    public void inicializarLayoutRaiz() {
        try {
            // Carga el root layout desde un archivo xml
            FXMLLoader cargador = new FXMLLoader();
            cargador.setLocation(Principal.class.getResource("view/LayoutRaiz.fxml"));
            layoutRaiz = (BorderPane) cargador.load();
            // Muestra la escena que contiene el RootLayout
            Scene scene = new Scene(layoutRaiz);
            escenarioPrincipal.setScene(scene);
            escenarioPrincipal.show();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    /**
     * Shows the person overview inside the root layout.
     */
    public void mostrarVistaPersona() {
        try {
            // Carga la vista de la persona.
            FXMLLoader loader = new FXMLLoader();
            loader.setLocation(Principal.class.getResource("view/EmpleadoVista.fxml"));
            AnchorPane vistaPersona = (AnchorPane) loader.load();

            // Fija la vista de la person en el centro del root layout.
            layoutRaiz.setCenter(vistaPersona);
            // Acceso al controlador.
            ControladorEmpleado miControlador = loader.getController();
            miControlador.setMiVentanaPrincipal(this);

        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }



    /**
     *
     * @return
     */
    public Stage getPrimaryStage() {
        return escenarioPrincipal;
    }

    public static void main(String[] args) {
```

Programa 5 Clase Principal completa

```
        launch(args);
    }

    public void inicializarEmpleado(String id, String nombre, String direccion, String
ciudad)
    {
      miEmpleado.fijarEmpleado(id, nombre, direccion, ciudad);
      System.out.println(miEmpleado.toString());
    }

    public String obtenerNombre()
    {
      return miEmpleado.getNombre();
    }

}
```

Ahora cree la clase ControladorEmpleado, para ello ubiquese en el paquete Vista y de clic derecho →new → clase.

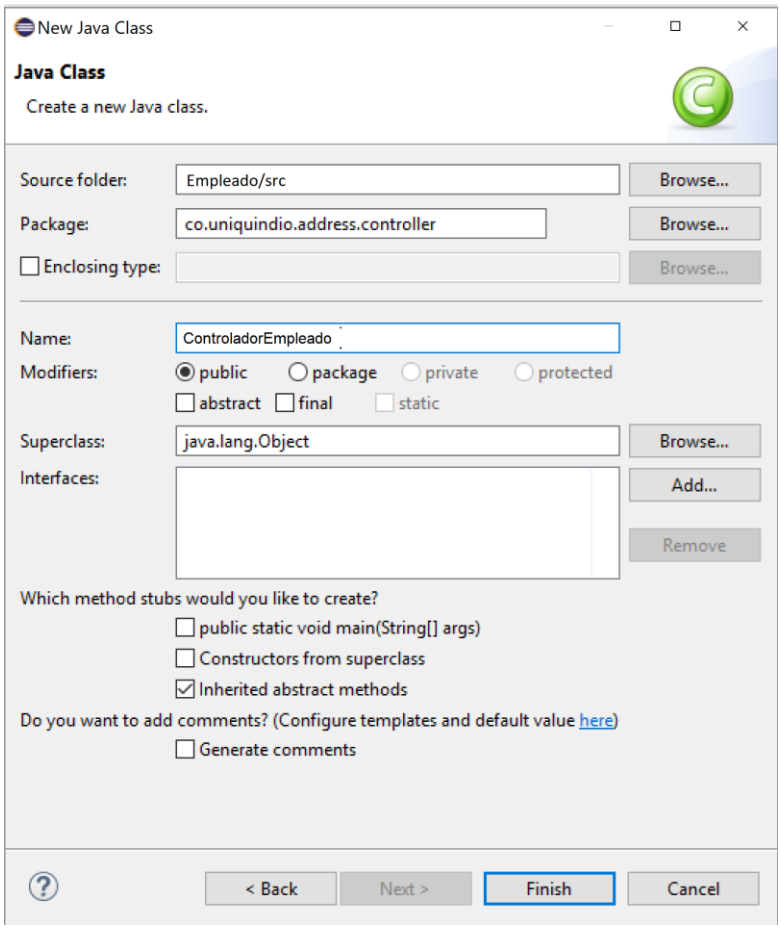


Figura 69 Clase ControladorEmpleado

Agregue el siguiente código a la clase controlador.

Programa 6 ControladorEmpleado

```
package co.uniquindio.address.view;
import java.awt.Button;

import co.uniquindio.address.Principal;
import co.uniquindio.address.model.Empleado;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;


/**
 *Permite editar los datos de la persona
 *
 * @author Sonia Jaramillo
 */
public class ControladorEmpleado {

    @FXML
    private TextField idTextField;
    @FXML
    private TextField nombreTextField;
    @FXML
    private TextField direccionTextField;
    @FXML
    private TextField ciudadTextField;
    @FXML
    private Button guardarButton;
    @FXML
    private Button consultarNombreButton;
    //para manejar el boton
    private boolean cliqueado = false;


    //VentanaPrincipal
    private Principal miVentanaPrincipal;

    /**
     * Inicializa la clase contenedor.
     */
    @FXML
    private void initialize() {
    }

    /**
     * Metodo modificador
     *
     * @param dialogo
```

Programa 6 ControladorEmpleado

```
     */


    public Principal getMiVentanaPrincipal() {
            return miVentanaPrincipal;
      }

      public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
            this.miVentanaPrincipal = miVentanaPrincipal;
      }

      /**
     * Retorna verdadero si el usuario dio clic sobre el boton
     *
     * @return
     */
    public boolean isCliqueado() {
        return cliqueado;
    }

    /**
     * Metodo para modificar la persona
     */
    @FXML
    private void fijarPersona() {
    String id=idTextField.getText();
    String nombre=nombreTextField.getText();
    String direccion=direccionTextField.getText();
    String ciudad=ciudadTextField.getText();
    miVentanaPrincipal.fijarEmpleado(id, nombre, direccion, ciudad);    cliqueado = true;
    }

    /**
     * Se invoca cuando el usuario de clic en consultar nombre.
     */
    @FXML
    private void obtenerNombre() {
        String nombre= miVentanaPrincipal.obtenerNombre();
        Alert alert = new Alert(AlertType.INFORMATION);
         alert.setTitle("Consultar nombre del estudiante");
        alert.setContentText(nombre);
        alert.setHeaderText(null); //sin titulo interno;
        alert.initStyle(StageStyle.UTILITY);
        alert.showAndWait(); //Se muestra la ventana
    }


}
```

A continuación debe vincular la vista al controlador. Para ello abra VistaEmpleado.fxml en el SceneBuilder y en la sección Controller, ubicada al lado izquierdo, seleccione ControladorEmpleado. Además, seleccione cada uno de los campos de texto y configure su identidad, mediante el campo fx:id. Por ejemplo, para el primer textField la identidad es idTextField.

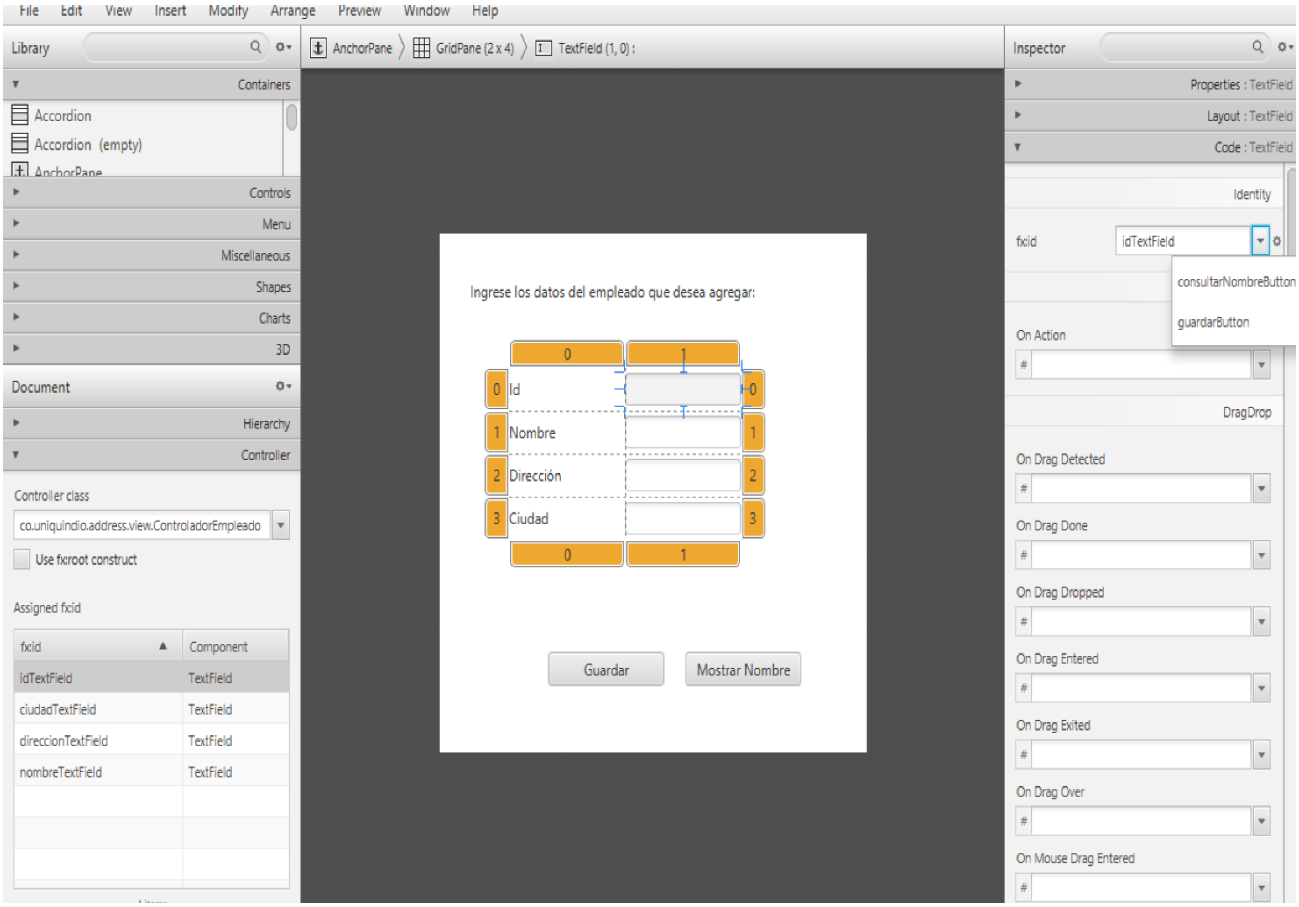


Figura 70 Asignar identidad a cada elemento

Luego de configurar todos los 4 campos debe visualizar lo siguiente, en el Controlador, a la izquierda de la pantalla.

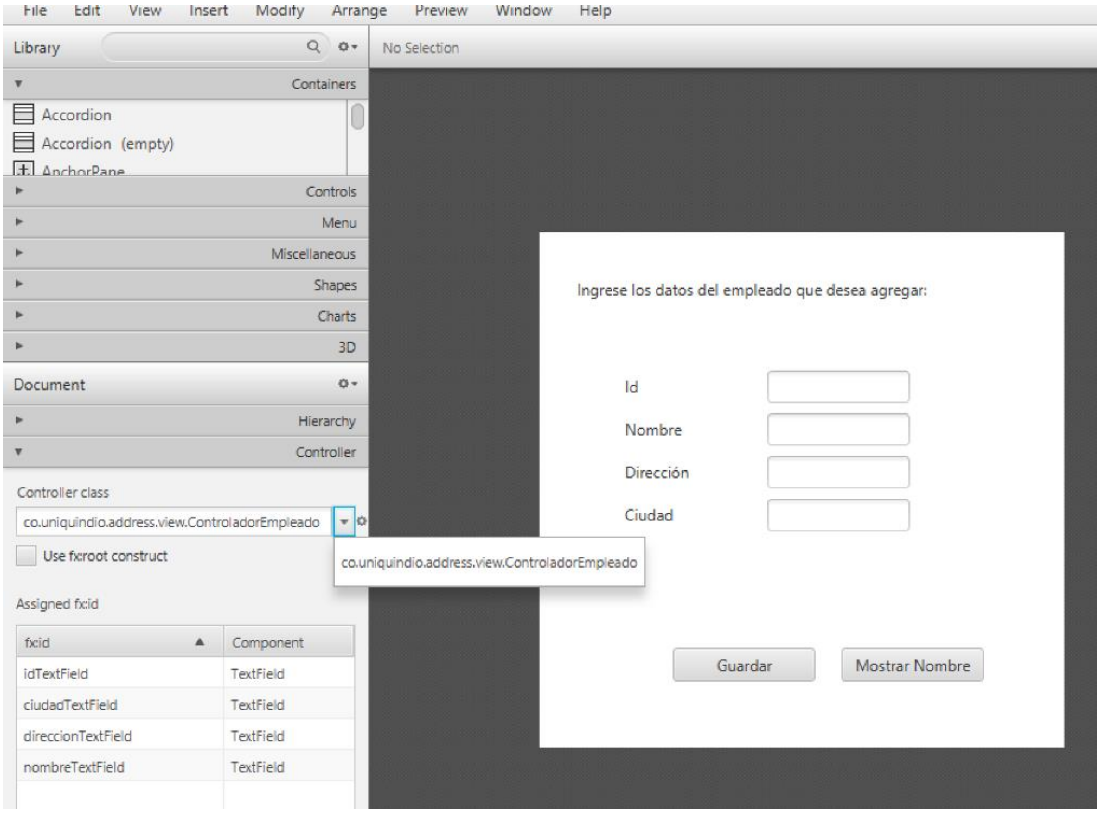


Figura 71 Controlador

Inactive use Use fx: root construct

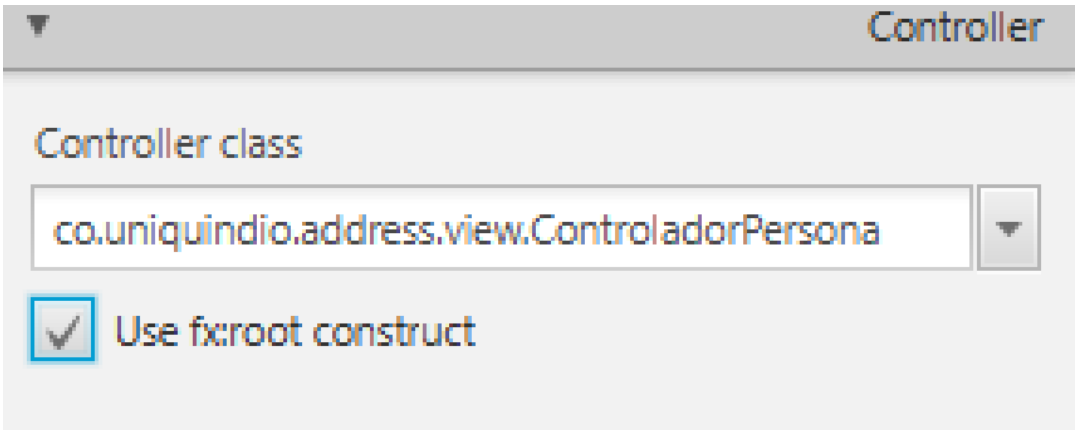


Figura 72 Inactivar la opción root construct

Finalmente configure las acciones de los botones, mediante la opcion on Action. Elija fijarPersona y obtenerNombre.

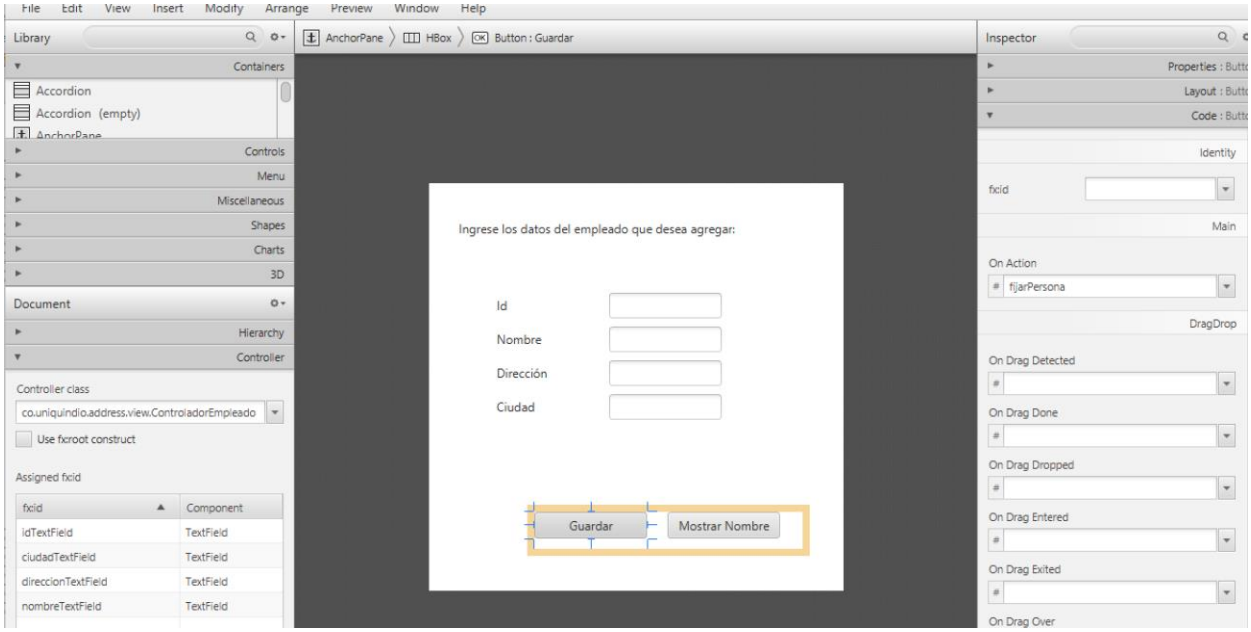


Figura 73 Configurar las acciones en los botones

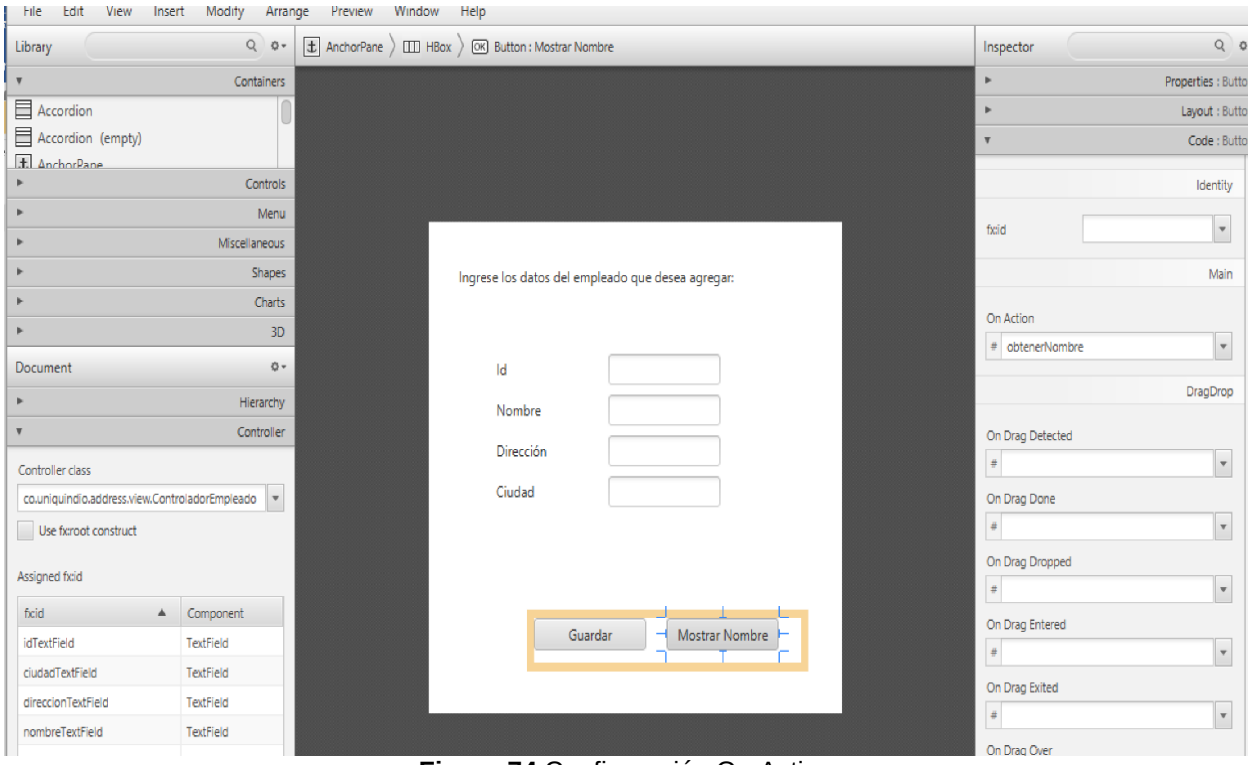


Figura 74 Configuración On Action

Nuevamente de clic en file y guardar. En Eclipse puede observar que el código generado es:

Programa 7 EmpleadoVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>

<AnchorPane prefHeight="398.0" prefWidth="405.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.address.view.ControladorEmpleado">
   <children>
      <Label layoutX="29.0" layoutY="36.0" prefHeight="17.0" prefWidth="279.0"
text="Ingrese los datos del empleado que desea agregar:" />
      <GridPane layoutX="66.0" layoutY="103.0" prefHeight="132.0" prefWidth="220.0"
AnchorPane.bottomAnchor="163.0" AnchorPane.leftAnchor="66.0"
AnchorPane.rightAnchor="119.0" AnchorPane.topAnchor="103.0">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Label text="Id" />
            <Label text="Nombre" GridPane.rowIndex="1" />
            <Label text="Dirección" GridPane.rowIndex="2" />
            <Label text="Ciudad" GridPane.rowIndex="3" />
            <TextField fx:id="idTextField" GridPane.columnIndex="1" />
            <TextField fx:id="nombreTextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
            <TextField fx:id="direccionTextField" GridPane.columnIndex="1"
GridPane.rowIndex="2" />
            <TextField fx:id="ciudadTextField" GridPane.columnIndex="1"
GridPane.rowIndex="3" />
         </children>
      </GridPane>
      <HBox layoutX="203.0" layoutY="321.0" prefHeight="36.0" prefWidth="265.0"
spacing="20.0" AnchorPane.bottomAnchor="41.0" AnchorPane.leftAnchor="103.0"
AnchorPane.rightAnchor="40.0">
         <children>
            <Button mnemonicParsing="false" onAction="#fijarEmpleado" prefHeight="26.0"
prefWidth="110.0" text="Guardar" />
            <Button mnemonicParsing="false" onAction="#obtenerNombre" prefHeight="26.0"
prefWidth="110.0" text="Mostrar Nombre" />
         </children>
      </HBox>
   </children> </AnchorPane>
```

Programa 8 LayoutRaiz.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>


<BorderPane prefHeight="398.0" prefWidth="405.0" xmlns:fx="http://javafx.com/fxml/1"
xmlns="http://javafx.com/javafx/8">
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem mnemonicParsing="false" text="Close" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Edit">
            <items>
              <MenuItem mnemonicParsing="false" text="Delete" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Help">
            <items>
              <MenuItem mnemonicParsing="false" text="About" />
            </items>
          </Menu>
        </menus>
      </MenuBar>
   </top>
</BorderPane>
```

Por último, ejecute la Aplicación, dando clic en New Run As→ Java Aplication, ver gráficas 74 y 75.

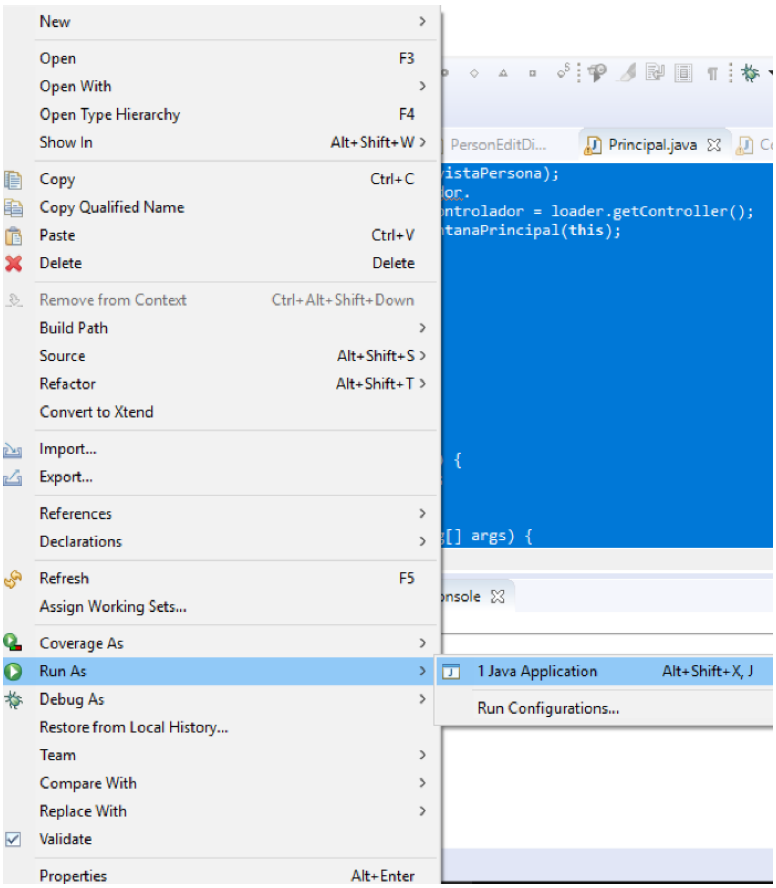


Figura 75 Ejecutar la aplicación

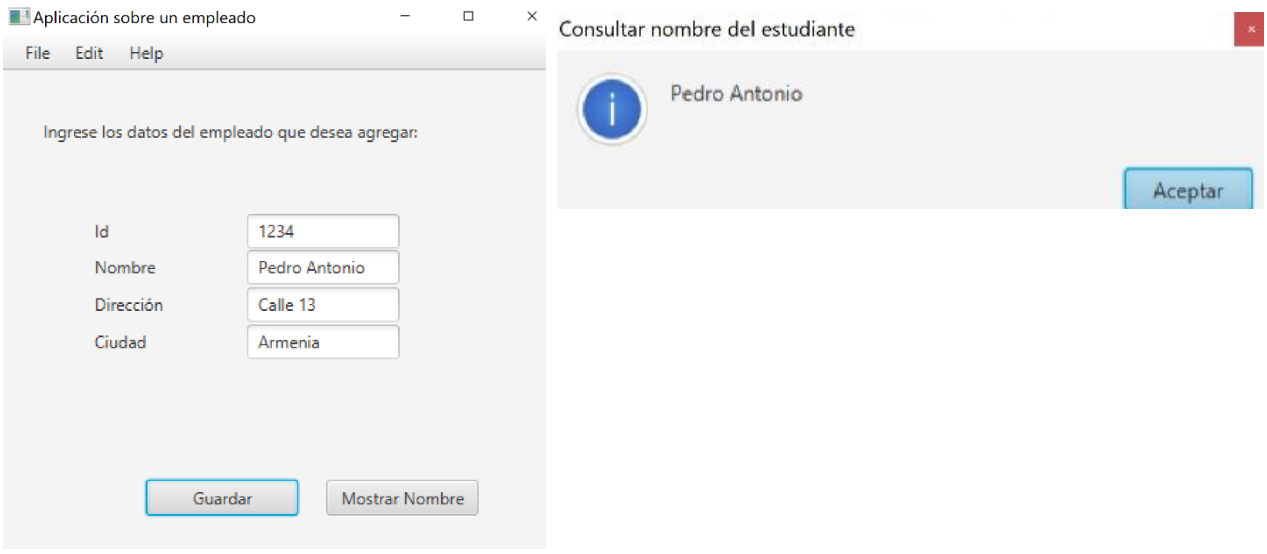


Figura 76 Aplicación en ejecución

1.11. Caso de estudio 2 Unidad I: El estudiante con notas

Se desea crear una aplicación para manejar la información de un estudiante. Un estudiante tiene los siguientes datos básicos: un código, un nombre y una fecha de nacimiento. Por cada estudiante se podrá registrar la información de dos asignaturas (cada una de ellas con dos notas parciales) Se debe permitir:
Registrar un estudiante con sus datos básicos
Asignar una asignatura a un estudiante, con sus respectivas notas parciales
Calcular la edad del estudiante
Calcular la nota definitiva de cada asignatura si se hace a través de un promedio ponderado.
Calcular la nota promedio del semestre, es decir, la nota que surge al promediar la nota definitiva de la asignatura 1 con la nota definitiva de la asignatura 2.

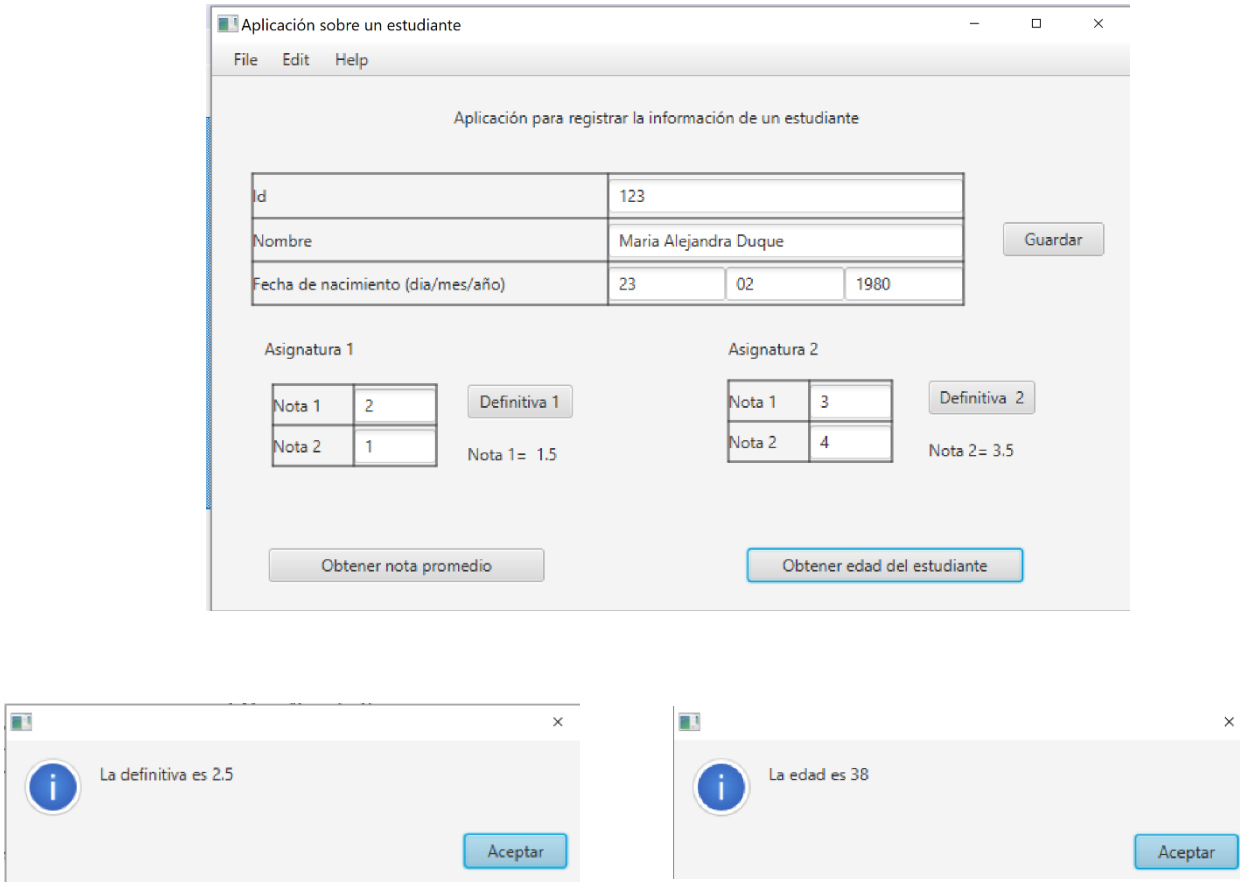


Figura 77 Interfaz de la aplicación Estudiante

1.11.1. Comprensión del problema

a) Requisitos funcionales

NOMBRE	R1 – Crear un nuevo estudiante
RESUMEN	Permite crear un nuevo estudiante
ENTRADAS	
id, nombre, fecha de nacimiento (dia, mes y año)	
RESULTADOS	
Un nuevo estudiante ha sido creado	

NOMBRE	R2 – Fijar asignatura 1
RESUMEN	Permite fijar una asignatura a un estudiante para ello se ingresan las dos notas
ENTRADAS	
nota1, nota2	
RESULTADOS	
Se fijó la asignatura 1 al estudiante	

NOMBRE	R3 – Fijar asignatura 2
RESUMEN	Permite fijar la asignatura 2 a un estudiante para ello se ingresan las dos notas
ENTRADAS	
nota1, nota2	
RESULTADOS	
Se fijó la asignatura 2 al estudiante	

NOMBRE	R4 – Calcular la nota definitiva para una asignatura
RESUMEN	Se calcula como el promedio ponderado de dos notas parciales
ENTRADAS	
Ninguno	
RESULTADOS	
La nota definitiva de la asignatura	

NOMBRE	R5 – Calcular la nota definitiva del semestre
RESUMEN	Permite calcular el promedio aritmético de las definitivas de las dos asignaturas
ENTRADAS	
Ninguna	
RESULTADOS	
La nota promedio	

NOMBRE	R6 – Calcular la edad del estudiante
RESUMEN	Permite calcular la edad del estudiante, para ello se resta la fecha actual de la fecha de nacimiento
ENTRADAS	
Ninguna	
RESULTADOS	
La edad	

b) El modelo del mundo del problema

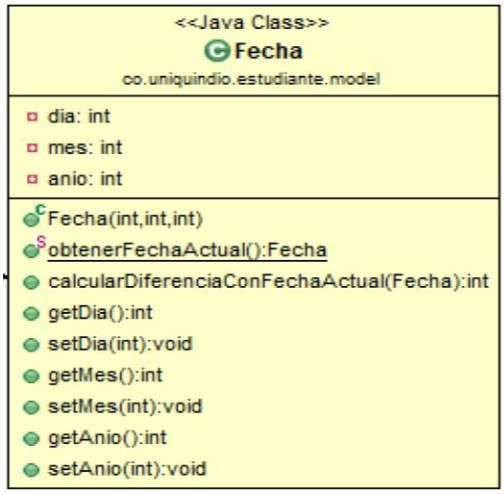
1. Identificar las entidades o clases
Las clases identificadas son las siguientes:

ENTIDAD DEL MUNDO	DESCRIPCIÓN
Estudiante	Es la entidad más importante del mundo del problema.
Fecha	Atributo del estudiante
Asignatura	Atributo del estudiante

Para identificar los métodos se deben resaltar los verbos

Se desea crear una aplicación para manejar la información de un estudiante. Un estudiante tiene un código, un nombre, un sexo, dos asignaturas registradas (cada una de ellas con código y dos notas parciales) y una fecha de nacimiento. A su vez, una fecha está formada por un día, un mes y un año.
Se debe permitir:
Calcular la edad del estudiante
Calcular la nota definitiva de cada asignatura si se hace a través de un promedio ponderado.
Calcular la nota promedio del semestre, es decir, la nota que surge al promediar la nota definitiva de la asignatura 1 con la nota definitiva de la asignatura 2.

IDENTIFICACIÓN Y MODELAMIENTO DE ATRIBUTOS PARA FECHA



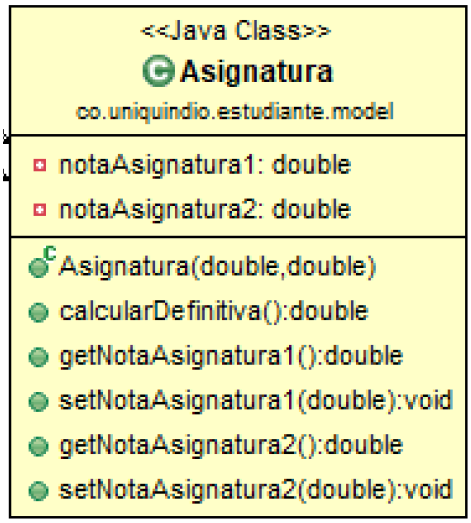
ATRIBUTO	VALORES POSIBLES	Tipo de dato
dia	enteros positivos mayores que 1 y menores de 32	int

mes anio	enteros mayores que 1 y menores que 13 enteros positivos	int int

IDENTIFICACIÓN DE MÉTODOS

```
(para crear una fecha se debe fijar cada uno de los atributos)
public void setDia(int dia)
public void setMes(int mes)
public void setAnio(int anio)

(se debe poder obtener la fecha actual)
public void obtenerFechaActual( )

(Permite restar a la fecha actual la fecha de nacimiento)
public int calcularDiferenciaConFechaActual( Fecha miFecha )

//Ademas de los métodos get para cada uno de los atributos
```

IDENTIFICACIÓN Y MODELAMIENTO DE ATRIBUTOS PARA ASIGNATURA

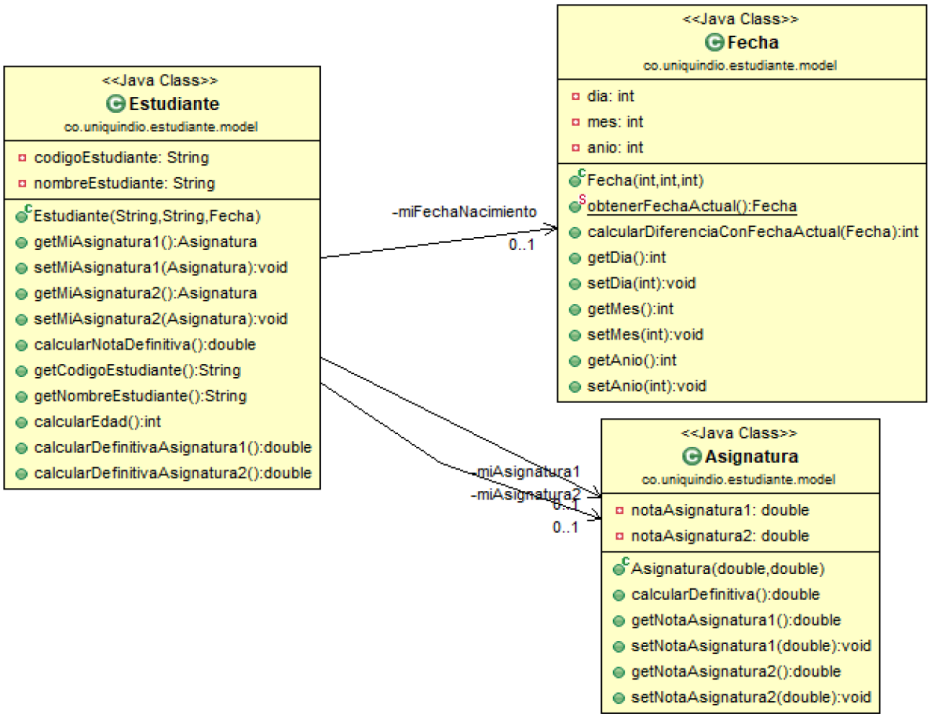


ATRIBUTO	VALORES POSIBLES	Tipo de dato
nota1 nota2	Un número real mayor que 0 y menor o igual a 5 Un número real mayor que 0 y menor o igual a 5	double double

IDENTIFICACIÓN DE MÉTODOS

```
(para crear una Asignatura se debe fijar cada uno de los atributos que el usuario conoce,
mediante el método constructor)
public Asignatura(double nota1, double nota2)

(para calcular la nota definitiva se requiere un método que calcule el promedio
aritmetico)
public double calcularDefinitiva()

//Además de los respectivos métodos get y set para cada uno de los atributos
```

IDENTIFICACIÓN Y MODELAMIENTO DE ATRIBUTOS PARA ESTUDIANTE



ATRIBUTO	VALORES POSIBLES	Tipo de dato
codigoEstudiante	Cadena de caracteres	String
nombreEstudiante	Cadena de caracteres	String
miFechaNacimiento	Referencia a objeto de tipo Fecha	**Fecha**
miAsignatura1	Referencia a objeto de tipo Asignatura	**Asignatura**
miAsignatura2	Referencia a objeto de tipo Asignatura	**Asignatura**

IDENTIFICACIÓN DE MÉTODOS MAS RELEVANTES

(para crear un estudiante se debe fijar cada uno de los atributos, se hará mediante un constructor)

```
public Estudiante (String codigo, String nombre, Fecha miFecha)
```

(para fijar cada asignatura se tiene un método set

```
public void setMiAsignatura1(Asignatura miAsignatura)
public void setMiAsignatura2(Asignatura miAsignatura)
```

(para calcular la nota definitiva se requiere un método que haga el promedio de las dos notas)

```
public double calcularNotaDefinitiva()
```

(para calcular la edad se requiere de un método que tome la fecha actual y la reste de la fecha de nacimiento)

```
public int calcularEdad()
```

(Se debe poder calcular la definitiva para cada asignatura)

```
public double calcularDefinitivaA1()
public double calcularDefinitivaA2()

//Faltan los respectivos métodos get.
```

El diagrama de clases correspondiente para este caso de estudio es el siguiente:

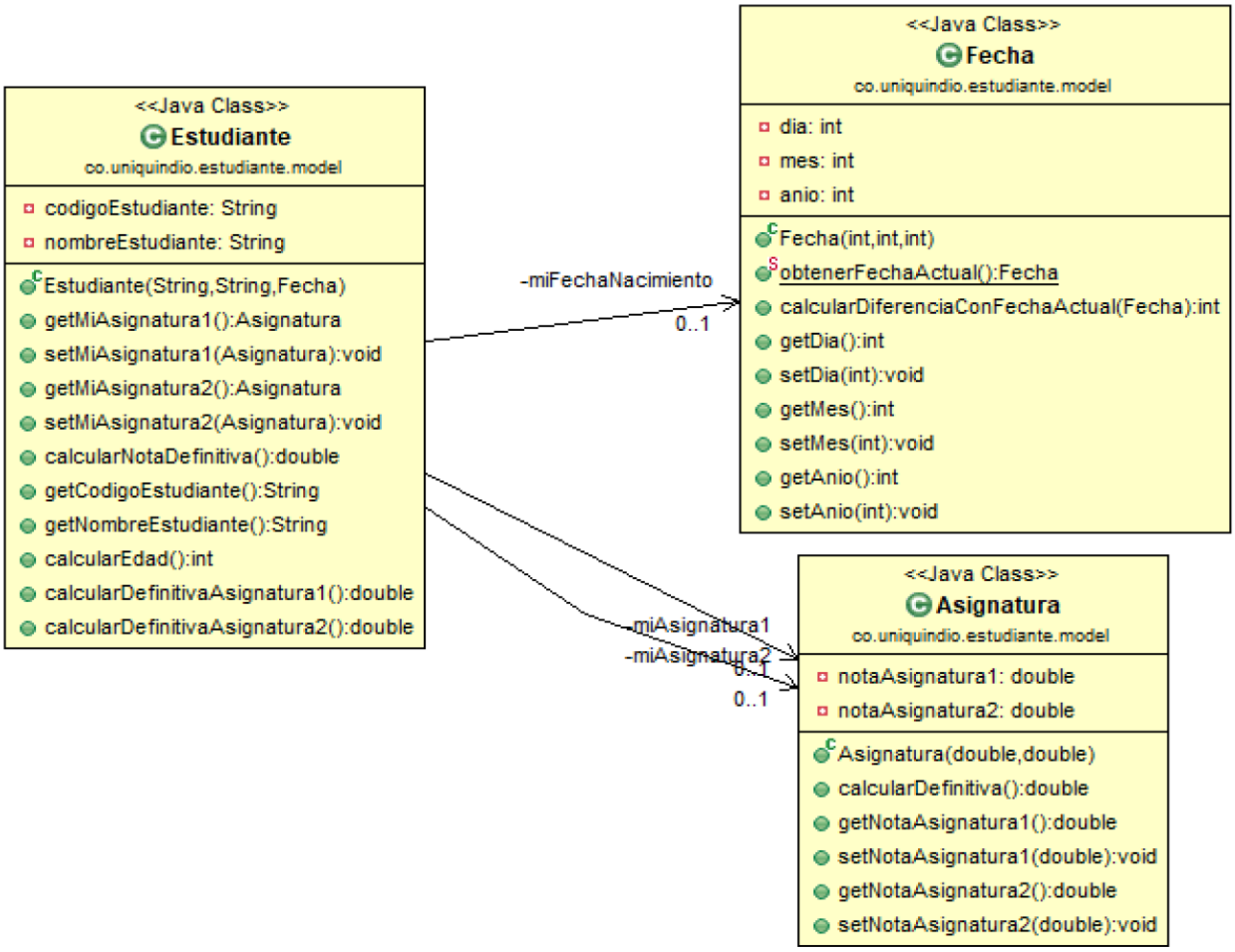


Figura 78 Diagrama completo con todas las clases de la lógica

La primera clase que se va a crear es la clase Fecha, dado que no depende de ninguna otra. Se inicia con la construcción del método inicializarFechaActual, en el que se crea una referencia a un objeto de tipo GregorianCalendar para luego inicializarlo con la fecha actual. El método calcularDiferenciaConFechaActual permite calcular la diferencia entre la fecha actual y una fecha ingresada por el usuario. La clase Fecha se presenta en el Programa 9.

Programa 9 Clase Fecha

```
package co.uniquindio.estudiante.model;

import java.util.Calendar;
import java.util.GregorianCalendar;
/**
 * Clase para representar una Fecha
 * @author sonia
 * @author sergio
 */
public class Fecha {
//Atributos de la clase
private int dia;
private int mes;
private int anio;
/**
 * Constructor de la clase Fecha
 * @param dia Es el dia
 * @param mes Es el mes
 * @param anio Es el anio
 */
public Fecha(int dia, int mes, int anio) {
	super();
	this.dia = dia;
	this.mes = mes;
	this.anio = anio;
}
/**
 * Devuelve la fecha actual
 * @return
 */
public static Fecha   obtenerFechaActual( )
{   Fecha actual;
    GregorianCalendar diaHoy = new GregorianCalendar( );
    int anio = diaHoy.get( Calendar.YEAR );
    int mes = diaHoy.get( Calendar.MONTH ) + 1;
    int dia = diaHoy.get( Calendar.DAY_OF_MONTH );
    actual=new Fecha(dia, mes,anio);
    return actual;
}
/**
 * Halla la diferencia entre dos fechas
 * @param miFecha Es la fecha que se resta
 * @return La cantidad de dias
 */
public int calcularDiferenciaConFechaActual( Fecha miFecha )
{
int diferenciaEnMeses = 0;
diferenciaEnMeses  =  ( miFecha.getMes() - mes )+12 * ( miFecha.getAnio() - anio ) ;
//En caso de que el día no sea mayor se debe restar un mes. Es decir, no //se tiene la
cantidad de días suficientes para poder sumar un mes más.
if(dia >= miFecha.getDia()  )
	diferenciaEnMeses --;
return diferenciaEnMeses/12 ;
```

Programa 9 Clase Fecha

```
}
/**
 * Metodo accesor
 * @return dia
 */
public int getDia() {
	return dia;
}
/**
 * Metodo modificador
 * @param dia
 */
public void setDia(int dia) {
	this.dia = dia;
}
/**
 * Metodo accesor
 * @return mes
 */
public int getMes() {
	return mes;
}
/**
 * Metodo modificador
 * @param mes
 */
public void setMes(int mes) {
	this.mes = mes;
}
/**
 * Metodo accesor
 * @return anio
 */
public int getAnio() {
	return anio;
}
/**
 * Metodo modificador
 * @param anio
 */
public void setAnio(int anio) {
	this.anio = anio;
}

}
```

En la clase Asignatura, ver ***Programa 10***, se implementan diferentes métodos, entre ellos el constructor de la clase, que tiene 2 parametros, correspondientes a las notas. El método calcularDefinitiva() calcula el promedio aritmético de las notas ingresadas.

Programa 10 Clase Asignatura

```java
package co.uniquindio.estudiante.model;

/**
 * @version 1.0
 * @author  Sonia Jaramillo Valbuena
 * @author  Sergio A. Cardona
 *
 * Maneja la informacion referente a la asignatura
 *
 */

public class Asignatura {
private double notaAsignatura1;
private double notaAsignatura2;
package co.uniquindio.estudiante.model;

/**
 * @version 1.0
 * @author  Sonia Jaramillo Valbuena
 * @author  Sergio A. Cardona
 *
 * Maneja la informacion referente a la asignatura
 *
 */

public class Asignatura {
/**
 * Se declaran los atributos
 */
private double notaAsignatura1;
private double notaAsignatura2;

/**
 * Metodo constructor de la clase Asignatura
 * @param codigoAsignatura
 * @param notaAsignatura
 */
public Asignatura( double notaAsignatura1,double notaAsignatura2)
{
	this.notaAsignatura1=notaAsignatura1;
	this.notaAsignatura2=notaAsignatura2;
}

/**
 * Calcula la nota definitiva
 * @return la definitiva
 */
public double calcularDefinitiva()
{
	double resultado=(notaAsignatura1+notaAsignatura2)/2;
	return resultado;
}
/**
```

Programa 10 Clase Asignatura

```
 * Devuelve la nota de la  asignatura 1
 * @return notaAsignatura1
 */
public double getNotaAsignatura1() {
	return notaAsignatura1;
}
/**
 * Metodo modificador
 * @param notaAsignatura1
 */
public void setNotaAsignatura1(double notaAsignatura1) {
	this.notaAsignatura1 = notaAsignatura1;
}
/**
 * Metodo accesor
 * @return notaAsignatura2
 */
public double getNotaAsignatura2() {
	return notaAsignatura2;
}
/**
 * Metodo modificador
 * @param notaAsignatura2
 */
public void setNotaAsignatura2(double notaAsignatura2) {
	this.notaAsignatura2 = notaAsignatura2;
}

}
```

Finalmente en la clase Estudiante, ver Programa 11, se implementa el metodo calcularNotaDefinitiva(), que retorna el promedio de las definitivas de ambas asignaturas. También se tiene el método calcularEdad, que calcula la diferencia entre 2 fechas. Para ello, en primer lugar se obtiene la fecha actual. Finalmente, a la fecha de nacimiento le resta la fecha actual.

Programa 11 Clase Estudiante

```
package co.uniquindio.estudiante.model;


/**
 * @version 1.0
 * @author  Sonia Jaramillo Valbuena
 * @author  Sergio A. Cardona
 *
 * Esta es la clase principal del mundo
 *
 */
public class Estudiante {
//Se definen los atributos
private String codigoEstudiante;
```

Programa 11 Clase Estudiante

```
private String nombreEstudiante;
private Fecha miFechaNacimiento;
private Asignatura miAsignatura1, miAsignatura2;

/**
 * Constructor del la clase Estudiante
 * @param codigoEstudiante codigo del estudiante solo puede tomar valores numericos
 * @param nombreEstudiante solo puede tener letras
 * @param Fecha miFechaNacimiento
 */
public Estudiante(String codigoEstudiante, String nombreEstudiante,Fecha
miFechaNacimiento)
 {this.codigoEstudiante=codigoEstudiante;
 this.nombreEstudiante=nombreEstudiante;
  this.miFechaNacimiento=miFechaNacimiento;
 }

public Asignatura getMiAsignatura1() {
	return miAsignatura1;
}

public void setMiAsignatura1(Asignatura miAsignatura1) {
	this.miAsignatura1 = miAsignatura1;
}

public Asignatura getMiAsignatura2() {
	return miAsignatura2;
}

public void setMiAsignatura2(Asignatura miAsignatura2) {
	this.miAsignatura2 = miAsignatura2;
}

/**
 * Permite calcular la nota definitiva
 * @return double valor con la nota definitiva
 */
public double calcularNotaDefinitiva()
{
double
definitiva=(miAsignatura1.calcularDefinitiva()+miAsignatura2.calcularDefinitiva())/2;
	return definitiva;
}


/**
 * Metodo accesor
 * @return String el codigo del Estudiante
 */
public String getCodigoEstudiante() {
	return codigoEstudiante;
}

/**
```

Programa 11 Clase Estudiante

```
 * Metodo accesor
 * @return String el nombre del estudiante
 */

public String getNombreEstudiante() {
      return nombreEstudiante;
}
/**
 * Calcula la edad del estudiante
 * @return la edad del estudiante
 */
public int calcularEdad()
{
Fecha actual= Fecha.obtenerFechaActual();
int edad= miFechaNacimiento.calcularDiferenciaConFechaActual(actual);
return edad;
}
/**
 * Calcula la definitiva de la asignatura 1
 * @return la definitiva
 */
public double calcularDefinitivaAsignatura1()
{
return miAsignatura1.calcularDefinitiva();
}
/**
 * Calcula la definitiva de la asignatura 2
 * @return la definitiva
 */
public double calcularDefinitivaAsignatura2()
{
return miAsignatura2.calcularDefinitiva();
}
}
```

En el Programa 12 se presenta la clase Principal. En primer lugar se declaran los atributos, escenarioPrincipal y layoutRaiz. El método inicializarLayoutRaiz() permite especificar la ruta del archivo LayoutRaiz.fxml y asignar ese layout a la escena. El método mostrarVistaEstudiante() permite cargar el archivo FXML y cargar el controlador del estudiante. Los métodos crearEstudiante, setMiAsignatura1, setMiAsignatura2, calcularEdad, calcularDefinitivaAsignatura1, calcularDefinitivaAsignatura2 y calcularNotaDefinitiva permiten la interacción con la clase principal del mundo del problema.

Programa 12 Clase Principal

```
package co.uniquindio.estudiante;
import java.io.IOException;
import co.uniquindio.estudiante.model.Asignatura;
import co.uniquindio.estudiante.model.Estudiante;
import co.uniquindio.estudiante.model.Fecha;
import co.uniquindio.estudiante.view.ControladorEstudiante;
import javafx.application.Application;
```

Programa 12 Clase Principal

```
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Principal extends Application {
/**
 * Se declaran los atributos
 */
    private Stage escenarioPrincipal;
    private BorderPane layoutRaiz;
  /**
   * Se declara una referencia a la clase principal del mundo de la lógica
   */
    private Estudiante miEstudiante;

  /**
   * Metodo start
   */
    @Override
    public void start(Stage primaryStage) {

        this.escenarioPrincipal = primaryStage;
        this.escenarioPrincipal.setTitle("Aplicación sobre un estudiante");
        inicializarLayoutRaiz();
         mostrarVistaEstudiante();
    }

    /**
     * Inicializa el layout raiz
     */
    public void inicializarLayoutRaiz() {
        try {
            // Carga el root layout desde un archivo xml
            FXMLLoader cargador = new FXMLLoader();
            cargador.setLocation(Principal.class.getResource("view/LayoutRaiz.fxml"));
            layoutRaiz = (BorderPane) cargador.load();
            // Muestra la escena que contiene el RootLayout
            Scene scene = new Scene(layoutRaiz);
            escenarioPrincipal.setScene(scene);
            escenarioPrincipal.show();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    /**
     * Muestra la vista del estudiante
     */
    public void mostrarVistaEstudiante() {
        try {
```

Programa 12 Clase Principal

```
            // Carga la vista de la persona.
            FXMLLoader loader = new FXMLLoader();
            loader.setLocation(Principal.class.getResource("view/EstudianteVista.fxml"));
            AnchorPane vistaPersona = (AnchorPane) loader.load();

            // Fija la vista de la person en el centro del root layout.
            layoutRaiz.setCenter(vistaPersona);
            // Acceso al controlador.
            ControladorEstudiante miControlador = loader.getController();
            miControlador.setMiVentanaPrincipal(this);

        } catch (IOException e) {
         System.out.println(e.getMessage());
        }
    }

/**
 * Devuelve el escenario principal
 * @return escenarioPrincipal
 */
    public Stage getPrimaryStage() {
        return escenarioPrincipal;
    }
/**
 * Metodo principal
 * @param args Los argumentos de la linea de comando
 */
    public static void main(String[] args) {
        launch(args);
    }
   /**
    * Muestra un mensaje
    * @param mensaje El mensaje
    * @param miA El tipo de alerta
    * @param titulo,  El titulo
    * @param cabecera, La cabecera
    * @param contenido, El contenido a mostrar
    * @param escenarioPrincipal, El escenario donde se muestra el mensaje
    */
    public static void mostrarMensaje(String mensaje, AlertType miA, String titulo,
String cabecera, String contenido, Stage escenarioPrincipal )
    {
        // Muestra el mensaje
        Alert alert = new Alert(miA);
        alert.initOwner(escenarioPrincipal);
        alert.setTitle(titulo);
        alert.setHeaderText(cabecera);
        alert.setContentText(contenido);
         alert.showAndWait();
    }
/**
 * Devuelve el escenario principal
 * @return el escenarioPrincipal
 */
```

Programa 12 Clase Principal

```
	public Stage getEscenarioPrincipal() {
		return escenarioPrincipal;
	}
/**
 * Permite fijar el escenario principal
 * @param escenarioPrincipal
 */
	public void setEscenarioPrincipal(Stage escenarioPrincipal) {
		this.escenarioPrincipal = escenarioPrincipal;
	}
	/**
	 * Permiten crear el estudiante
	 * @param id El id
	 * @param nombre El nombre del  estudiante
	 * @param miFechaNacimiento La fecha de nacimiento
	 */
	public void crearEstudiante(   String id, String nombre, Fecha miFechaNacimiento)
    {
      miEstudiante=new Estudiante(id, nombre, miFechaNacimiento);

    }
    /**
     * Permite fijar la asignatura 1
     * @param miAsignatura1
     */
    public void setMiAsignatura1(Asignatura miAsignatura1) {
      miEstudiante.setMiAsignatura1(miAsignatura1);
    }
    /**
     * Permite fijar la asignatura 2
     * @param miAsignatura2
     */
    public void setMiAsignatura2(Asignatura miAsignatura2) {
      miEstudiante.setMiAsignatura2(miAsignatura2);
    }
    /**
     * Permite calcular la nota definitiva
     * @return double valor con la nota definitiva
     */
    public double calcularNotaDefinitiva()
    {
      return miEstudiante.calcularNotaDefinitiva();
    }

    /**
     * Calcula la definitiva de la asignatura 2
     * @return la definitiva
     */
    public double calcularDefinitivaAsignatura1()
    {
    return miEstudiante.calcularDefinitivaAsignatura1();
    }

    /**
```

Programa 12 Clase Principal

```
     * Calcula la definitiva de la asignatura 2
     * @return la definitiva
     */
    public double calcularDefinitivaAsignatura2()
    {
      return miEstudiante.calcularDefinitivaAsignatura2();
    }
    /**
     * Calcula la edad del estudiante
     * @return la edad del estudiante
     */
    public int calcularEdad()
    {
      return miEstudiante.calcularEdad();
     }
 }
```

En el Programa 13 se declaran los elementos de los cuales se debe extraer información, es decir, los campos de texto idTextField, nombreTextField, diaTextField, mesTextField, anioTextField, nota1A1TextField, nota2A1TextField, nota1A2TextField y nota2A2TextField. El método crearEstudiante(), obtiene el texto de los Textfield mediante la instrucción getText, y a continuación se procede a crear una instancia de Fecha, llamada miFecha. Con toda la información obtenida se invoca el método crearEstudiante, mediante la instrucción miVentanaPrincipal.crearEstudiante(id, nombre, miFecha).

Programa 13 Controlador Estudiante

```
package co.uniquindio.estudiante.view;

import java.awt.Button;

import co.uniquindio.estudiante.Principal;
import co.uniquindio.estudiante.model.Asignatura;
import co.uniquindio.estudiante.model.Fecha;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;


/**
 *Permite editar los datos de la persona
 *
 * @author Sonia Jaramillo
 */
public class ControladorEstudiante {
 /**
  * Se declaran los atributos
```

Programa 13 Controlador Estudiante

```
  */
    @FXML
    private TextField idTextField;
    @FXML
    private TextField nombreTextField;
    @FXML
    private TextField diaTextField;
    @FXML
    private TextField mesTextField;
    @FXML
    private TextField anioTextField;
    @FXML
    private TextField nota1A1TextField;
    @FXML
    private TextField nota2A1TextField;
    @FXML
    private TextField nota1A2TextField;
    @FXML
    private TextField nota2A2TextField;
    @FXML
    private Button guardarEstudianteButton;
    @FXML
    private Button definitiva1Button;
    @FXML
    private Button definitiva2Button;
    @FXML
    private Button promedioButton;
    @FXML
    private Button edadButton;
    @FXML
    private Label  definitiva1Label;
    @FXML
    private Label  definitiva2Label;

/**
 * Se declara una referencia a la ventana principal
 */
    private Principal miVentanaPrincipal;

    /**
     * Inicializa la clase contenedor.
     */
    @FXML
    private void initialize() {
    }

  /**
   * Metodo accesor
   * @return miVentanaPrincipal;
   */


    public Principal getMiVentanaPrincipal() {
```

Programa 13 Controlador Estudiante

```
            return miVentanaPrincipal;
      }
/**
 * Metodo modificador
 * @param miVentanaPrincipal
 */
      public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
            this.miVentanaPrincipal = miVentanaPrincipal;
      }

    /**
     * Metodo para modifica el estudiante
     */
    @FXML
    private void crearEstudiante() {
    String id=idTextField.getText();
    String nombre=nombreTextField.getText();
   int dia=Integer.parseInt( diaTextField.getText());
   int mes=Integer.parseInt( mesTextField.getText());
   int anio=Integer.parseInt( anioTextField.getText());
   Fecha miFecha=new Fecha (dia, mes, anio);
    miVentanaPrincipal.crearEstudiante(id, nombre, miFecha);
   }
    /**
     * Metodo para capturar los datos de la asignatura 1
     */
    public void fijarAsignatura1 ()
    {
     double nota1A1= Double.parseDouble( nota1A1TextField.getText());
    double nota2A1= Double.parseDouble(nota2A1TextField.getText());
    Asignatura miA=new Asignatura(  nota1A1,nota2A1);
    miVentanaPrincipal.setMiAsignatura1(miA);
    definitiva1Label.setText("Nota 1=
"+miVentanaPrincipal.calcularDefinitivaAsignatura1());

     }
    /**
     * Metodo para capturar los datos de la asignatura 2
     */
    public void fijarAsignatura2()
    {
   double nota1A2= Double.parseDouble( nota1A2TextField.getText());
    double nota2A2= Double.parseDouble(nota2A2TextField.getText());
    Asignatura miA=new Asignatura(  nota1A2,nota2A2);
    miVentanaPrincipal.setMiAsignatura2(miA);
    definitiva2Label.setText("Nota 2=
"+miVentanaPrincipal.calcularDefinitivaAsignatura2());
    }
    /**
     * Permite calcular la definitiva
      */
    public void calcularNotaDefinitiva()
    {double nota=miVentanaPrincipal.calcularNotaDefinitiva();
```

Programa 13 Controlador Estudiante

```
      miVentanaPrincipal.mostrarMensaje("Definitiva ", AlertType.INFORMATION, "", "", "La
definitiva es "+nota, miVentanaPrincipal.getEscenarioPrincipal() );

    }
    /**
     * Se invoca cuando el usuario de clic en edad.
     */
    @FXML
    private void mostrarEdad() {
        int edad=miVentanaPrincipal.calcularEdad();
      miVentanaPrincipal.mostrarMensaje("Edad del estudiante", AlertType.INFORMATION, "",
"", "La edad es "+edad, miVentanaPrincipal.getEscenarioPrincipal() );
      }
}
```

El Programa 15 presenta el código fuente generado en el SceneBuilder para construir la siguiente interfaz.

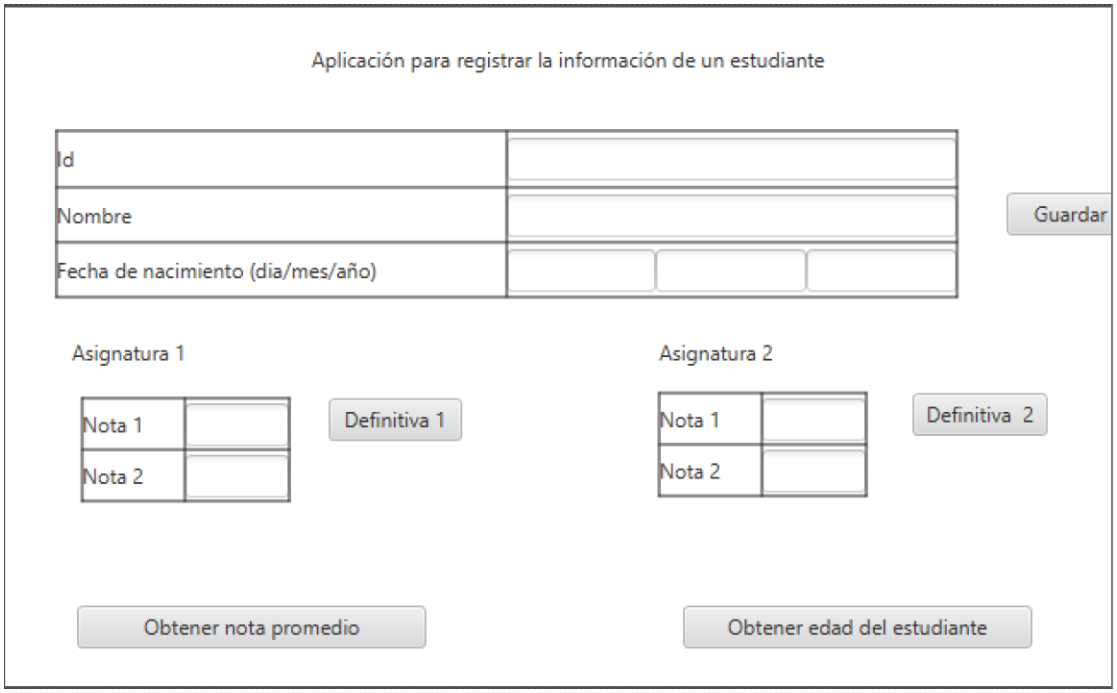


Figura 79 Pantalla para ingresar los datos del estudiante

Programa 14 EstudianteVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.shape.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>
```

Programa 14 EstudianteVista.fxml

```
<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-
Infinity" prefHeight="396.0" prefWidth="679.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.estudiante.view.ControladorEstudiante">
   <children>
      <Label layoutX="179.0" layoutY="22.0" prefHeight="17.0" prefWidth="397.0"
text="Aplicación para registrar la información de un estudiante" />
      <GridPane alignment="TOP_CENTER" cacheHint="QUALITY" gridLinesVisible="true"
layoutX="33.0" layoutY="72.0" prefHeight="97.0" prefWidth="523.0"
AnchorPane.bottomAnchor="227.0" AnchorPane.leftAnchor="30.0"
AnchorPane.rightAnchor="123.0" AnchorPane.topAnchor="72.0">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Label prefHeight="17.0" prefWidth="64.0" text="Id" textAlignment="JUSTIFY"
/>
            <Label text="Nombre" GridPane.rowIndex="1" />
            <Label prefHeight="17.0" prefWidth="219.0" text="Fecha de nacimiento
(dia/mes/año)" GridPane.rowIndex="2" />
            <TextField fx:id="idTextField" GridPane.columnIndex="1" />
            <TextField fx:id="nombreTextField" prefHeight="25.0" prefWidth="189.0"
GridPane.columnIndex="1" GridPane.rowIndex="1" />
            <GridPane prefHeight="50.0" prefWidth="215.0" GridPane.columnIndex="1"
GridPane.rowIndex="2">
              <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
                  <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0"
/>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
              </columnConstraints>
              <rowConstraints>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
              </rowConstraints>
               <children>
                  <TextField fx:id="anioTextField" GridPane.columnIndex="2" />
                  <TextField fx:id="diaTextField" />
                  <TextField fx:id="mesTextField" GridPane.columnIndex="1" />
               </children>
            </GridPane>
         </children>
      </GridPane>
      <Pane layoutX="31.0" layoutY="179.0" prefHeight="149.0" prefWidth="315.0"
AnchorPane.bottomAnchor="68.0" AnchorPane.leftAnchor="31.0"
AnchorPane.rightAnchor="333.0" AnchorPane.topAnchor="179.0">
         <children>
            <Label layoutX="8.0" layoutY="14.0" text="Asignatura 1" />
```

Programa 14 EstudianteVista.fxml

```
            <GridPane gridLinesVisible="true" layoutX="14.0" layoutY="49.0"
prefHeight="60.0" prefWidth="121.0">
              <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
              </columnConstraints>
              <rowConstraints>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
              </rowConstraints>
               <children>
                  <Label text="Nota 1" />
                  <Label text="Nota 2" GridPane.rowIndex="1" />
                  <TextField fx:id="nota1A1TextField" GridPane.columnIndex="1" />
                  <TextField fx:id="nota2A1TextField" prefHeight="25.0" prefWidth="72.0"
GridPane.columnIndex="1" GridPane.rowIndex="1" />
               </children>
            </GridPane>
            <Button layoutX="158.0" layoutY="49.0" mnemonicParsing="false"
onAction="#fijarAsignatura1" prefHeight="25.0" prefWidth="78.0" text="Definitiva 1" />
            <Label fx:id="definitiva1Label" layoutX="158.0" layoutY="92.0"
prefHeight="17.0" prefWidth="149.0" />
         </children>
      </Pane>
      <Pane layoutX="351.0" layoutY="181.0" prefHeight="149.0" prefWidth="350.0"
AnchorPane.bottomAnchor="68.0" AnchorPane.leftAnchor="350.0"
AnchorPane.rightAnchor="29.0" AnchorPane.topAnchor="179.0">
         <children>
            <Label layoutX="32.0" layoutY="14.0" text="Asignatura 2" />
            <GridPane gridLinesVisible="true" layoutX="32.0" layoutY="46.0"
prefHeight="60.0" prefWidth="121.0">
              <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
              </columnConstraints>
              <rowConstraints>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
              </rowConstraints>
               <children>
                  <Label text="Nota 1" />
                  <Label text="Nota 2" GridPane.rowIndex="1" />
                  <TextField fx:id="nota1A2TextField" GridPane.columnIndex="1" />
                  <TextField fx:id="nota2A2TextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
               </children>
            </GridPane>
            <Button layoutX="180.0" layoutY="46.0" mnemonicParsing="false"
onAction="#fijarAsignatura2" text="Definitiva  2" />
            <Label fx:id="definitiva2Label" layoutX="180.0" layoutY="89.0"
prefHeight="17.0" prefWidth="132.0" />
         </children>
      </Pane>
      <HBox layoutX="10.0" layoutY="336.0" spacing="11.0" />
```

Programa 14 EstudianteVista.fxml

```
      <HBox layoutX="42.0" layoutY="349.0" spacing="150.0" AnchorPane.bottomAnchor="22.0"
AnchorPane.leftAnchor="42.0" AnchorPane.rightAnchor="79.0" AnchorPane.topAnchor="349.0">
         <children>
            <Button layoutX="42.0" layoutY="349.0" mnemonicParsing="false"
onAction="#calcularNotaDefinitiva" prefHeight="25.0" prefWidth="204.0" text="Obtener nota
promedio" AnchorPane.leftAnchor="42.0" AnchorPane.topAnchor="349.0" />
            <Button layoutX="410.0" layoutY="349.0" mnemonicParsing="false"
onAction="#mostrarEdad" prefWidth="204.0" text="Obtener edad del estudiante" />
         </children>
      </HBox>
      <Button layoutX="582.0" layoutY="108.0" mnemonicParsing="false"
onAction="#crearEstudiante" prefHeight="25.0" prefWidth="78.0" text="Guardar"
AnchorPane.bottomAnchor="263.0" AnchorPane.leftAnchor="585.0"
AnchorPane.rightAnchor="19.0" AnchorPane.topAnchor="108.0" />
   </children>
</AnchorPane>
```

El Programa 15 presenta el código para generar el LayoutRaiz, sobre el se mostrará la vista del estudiante.

Figura 80 Layout Raiz

Programa 15 LayoutRaiz.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="420.0" prefWidth="679.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1">
```

Programa 15 LayoutRaiz.fxml

```
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem mnemonicParsing="false" text="Close" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Edit">
            <items>
              <MenuItem mnemonicParsing="false" text="Delete" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Help">
            <items>
              <MenuItem mnemonicParsing="false" text="About" />
            </items>
          </Menu>
        </menus>
      </MenuBar>
   </top>
</BorderPane>
```

Actividad 6. Minimercado

Construya una aplicación para un minimercado, el cual ofrece tres diferentes productos. Se sabe que cada producto tiene un código, una cantidad de existencias y un precio unitario.

Se debe permitir:
Agregar un producto a la compra.
Aumentar en 1 la cantidad de existencias adquiridas de un producto incluido en la compra
Indicar la totalidad de artículos incluidos en la compra
Calcular el total a pagar por cada tipo de producto
Calcular el total a pagar por el total de la compra

Debe manejar dos 2 clases, Minimercado y Compra. En primer lugar identifique por cada una de ellas los atributos y luego cada uno de los métodos que debe implementar. Elabore el diagrama de clases. Implemente todo en Eclipse, con sus correspondientes interfaces gráficas.

2. Estructuras de decisión

Objetivos Pedagógicos

Al final de este nivel el lector estará en capacidad de:

- Utilizar instrucciones condicionales simples y anidadas para dar solución a problemas que requieren la toma de decisiones.
- Utilizar constantes al momento de modelar las características de un objeto
- Utilizar expresiones dentro de métodos como un medio para modificar el estado de un objeto.

2. EXPRESIONES Y ESTRUCTURAS DE DECISION

En Java se manejan dos estructuras de selección if y switch.

2.1 Instrucción If

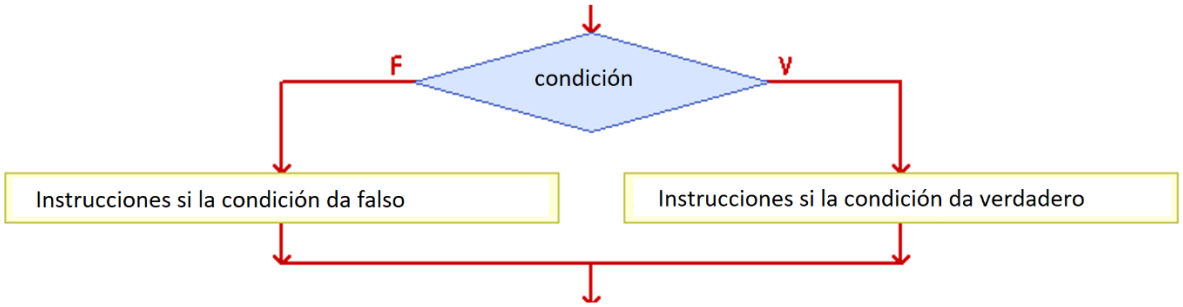
La instrucción if, evalúa la condición, si el resultado es verdadero ejecuta las acciones encerradas dentro de llaves (grupoDeAcciones1). El grupoDeAcciones1 puede ser simple o compuesto. Si la condición es falsa, no se ejecuta ninguna acción.

```
if( condición )
{
  grupoDeAcciones1
}
```

Por ejemplo, si se desea cambiar calcular la raíz cuadrada de un número se puede escribir:

```
double raiz=0;
if(numero>=0)
{
        raiz=Math.sqrt(numero);
}
```

Una variante de esta estructura de decisión es el if-else. En ella se evalúa la condición y si es verdadera se ejecutan las acciones encerradas dentro de llaves, en caso contrario se ejecuta grupoDeAcciones2 [7][8].



El código correspondiente se presenta a continuación:

```
if ( condición )
{
//grupoDeAcciones1
}

 else
{
//grupoDeAcciones2
}
```

En el siguiente ejemplo si el número es positivo o cero, se calcula la raíz cuadrada, de lo contrario la variable raíz se inicializa con cero.

```
double raiz=0;

if(numero>=0)

{          raiz=Math.sqrt(numero);

}

else

 {

    raiz=0;

 }
```

Algunas instrucciones sencillas pueden escribirse mediante un if in-line. El siguiente ejemplo ilustra su uso. Si numero es menor que 10, entonces se le asigna el valor de 6, de lo contrario se le asigna 23.

```
int numero=0;

numero=((numero<10)? 6:23);
```

Otra variante es el **if** anidado, que se caracteriza porque la instrucción que sigue a un else es un if (excepto la última que podría ser un único else). En un if anidado, las condiciones se evalúan de forma descendente, es decir, solo pasa a la siguiente, si la anterior no es verdadera. Cuando la condición es verdadera se efectúa el bloque de instrucciones correspondiente y se omite el resto de la estructura. Si ninguna se cumple se efectúa la acción del último else [2][3][4][9].

Las condiciones se evalúan de arriba hacia abajo pasando de una a otra si la anterior resulta falsa. En el momento que se encuentra una condición verdadera, se efectúa la acción correspondiente a dicha condición y se corta el resto de la estructura. Si todas las condiciones resultan falsas se efectúa la acción correspondiente al último else. No hay límite con respecto al número de estructuras de selección doble que pueden ponerse en cascada[2][3][4][10].

Cuando el if es anidado o en cascada las condiciones se examinan de arriba hacia abajo (forma descencente) pasando de una a otra si la anterior resulta falsa. Cuando se encuentre la condición verdadera, entonces se efectúa la acción correspondiente (cuerpo correspondiente a dicha condición) y no se continúa examinando el resto de la estructura. En caso de no encontrar una condición verdadera se ejecuta la acción correspondiente al último else [2][3][4][11].

2.2 Instrucción switch

La instrucción **switch**, permite elegir entre diferentes rutas, haciendo uso de una variable denominada selector. El selector es comparado con una serie de constantes. En caso de que el selector coincida con alguno de ellos, se efectua el bloque de instrucciones correspondiente a dicha constante. En caso de no coincidir con ninguno se efectua la acción por defecto (**default**), si es que esta existe. Esta instruccion funciona con los tipos de dato primitivos byte, short, int y char. Tambien con Strings, y las clases Byte, Short, Integer y Character. También funciona con tipos de datos enumerados (que se construyen mediante Enum) [2][3][4][12].

El **switch** tiene la siguiente sintaxis:

```
switch ( selector )
{
  case constante1 : acción A1
        break;

  case constante2 : acción A2
        break;

  case constante3 : acción A3
        break;
          .
          .
          .
  case constanteN : acción An
        break;

  default : acción por defecto
}
```

Cada uno de estos valores dentro del case “constante” debe ser único, es decir, no se pueden definir rangos. Observe que se hace uso de la sentencia break, para romper, es decir, para que no continue con el siguiente caso (salta fuera del switch).

2.1. Excepciones

Una excepción es un problema inesperado que surge durante la ejecución de un programa. Para realizar un tratamiento básico a una excepción se hace uso de las instrucciones try y catch. Dentro del try van las instrucciones que se espera se ejecuten si todo sale bien, pero que pueden producir la excepción. Pero si algo malo ocurre entonces se ejecuta el catch. Ambas instrucciones serán usadas en las clases de la interfaz. En la lógica o modelo se usan las instrucciones throw y throws. Throw permite lanzar una excepción y throws se usa para indicar que el método puede lanzar esa excepción [4].

```
try {
  // bloque de código sobre el que se captura el error
}
catch (ArithmeticException  e){
}
catch (Exception  e1){
// instrucciones para el tratamiento del problema
}
finally {
  // bloque de código que  se ejecuta siempre, haya o no
excepción
}
```

Un ejemplo de uso de excepciones se presenta a continuación:

Programa 16 Ejemplo de excepciones

```
//Esta clase se guarda en un archivo aparte

public class DivPorCeroException extends Exception {
    public MiExcepcion(String message) {
        super(message);
    }
}
```

```
public class Ensayo
{
private double a, b;
public Ensayo(double a, double b)
{
this.a=a;
this.b=a;
}
public double  calcularDivision() throws DivPorCeroException
{
double res;
if(b==0)
{
throw new DivPorCeroException("El divisor es cero");
}
res=a/b;
return res;
}
}
```

```
public class Principal
{
public void dividir()
{
double res=0;
try
{
res=miEnsayo. calcularDivision();
System.out.println("El resultado es "+res);
}
catch (DivPorCeroException  e)
{
System.out.println(e.getMessage());
}
}
}
```

La Figura 81 presenta las clases de Java para el manejo de excepciones.

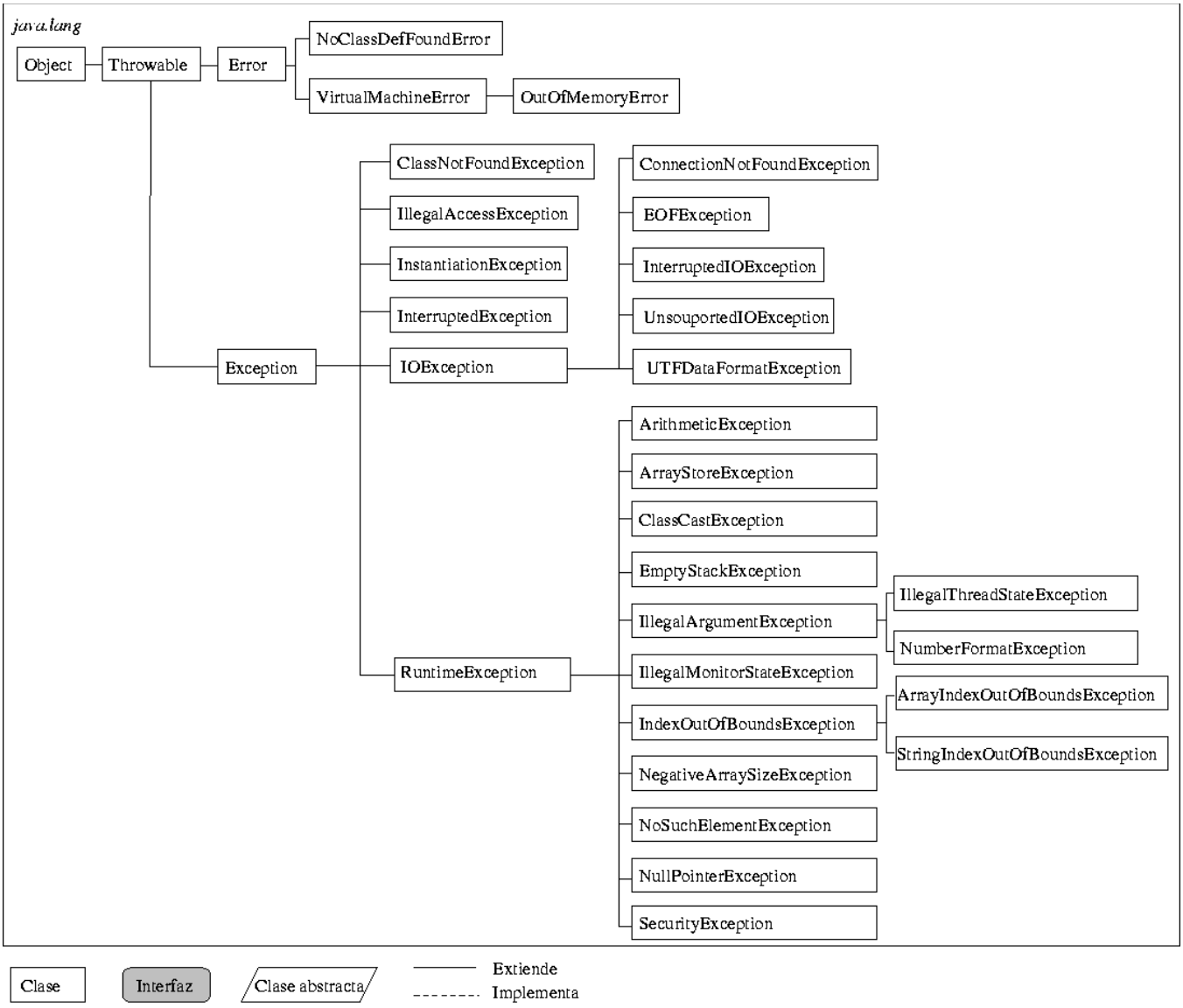


Figura 81 Excepciones en Java tomado de http://leo.ugr.es/J2ME/CLDC/transjava/node10.html

2.2. Caso de estudio 1 Unidad II: La Recta

Se requiere una aplicación para manejar la información de una recta. Una recta se define por 2 puntos. Cada uno de ellos tiene una coordenada en `x` y una coordenada en `y`. Se debe permitir fijar los puntos, mostrar la ecuación y obtener la pendiente.

La ecuación de la recta que pasa por los punto $P(x_1, y1)$ y $Q\ (x_2, y_2)$ es:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

La formula para calcular la pendiente de la recta m es

$m = (y_2 - y_1) / (x_2 - x_1)$

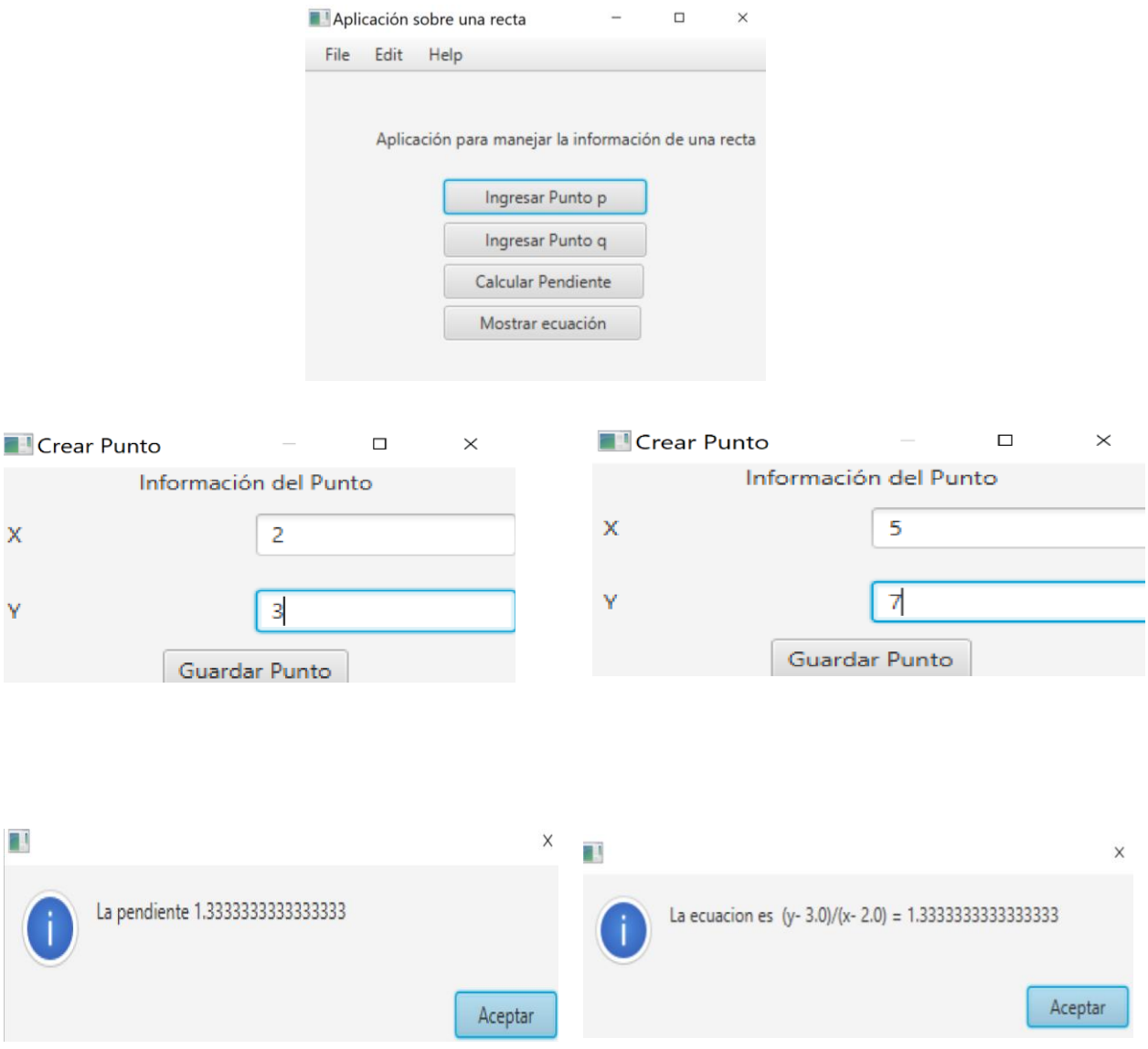


Figura 82 Interfaz aplicación del punto

2.2.1.Comprensión del problema

a) Requerimientos funcionales

NOMBRE	R1 – Inicializar punto p
RESUMEN	Se requiere recibir la información básica para crear un punto de la recta.
ENTRADAS	
El punto (con su respectiva x y y)	
RESULTADOS	
El punto ha sido creado	

NOMBRE	R2 – Inicializar punto q
RESUMEN	Se requiere recibir la información básica para crear un punto de la recta.
ENTRADAS	
El punto (con su respectiva x y y)	
RESULTADOS	
El punto ha sido creado	

NOMBRE	R2 – Calcular la pendiente de la recta
RESUMEN	Permitir calcular la pendiente de una recta, partiendo de la fórmula `m = ( y2 - y1 ) / ( x2 - x1 ).` Dado que ya se leyeron las coordenadas de los puntos `p` y `q` no es necesario ingresar más informacion
ENTRADAS	
Ninguna	
RESULTADOS	
Pendiente de la recta	

NOMBRE	R4 – Determinar la ecuación de la recta
RESUMEN	Permite determinar la ecuación de la recta. Ya se tiene un Punto p e igualmente la pendiente, con esta información es suficiente para obtener la ecuación de la recta.
ENTRADAS	
Ninguna	
RESULTADOS	
Ecuación de la recta	

b) El modelo del mundo del problema

Son cuatro las actividades que se deben realizar para construir el modelo del mundo.

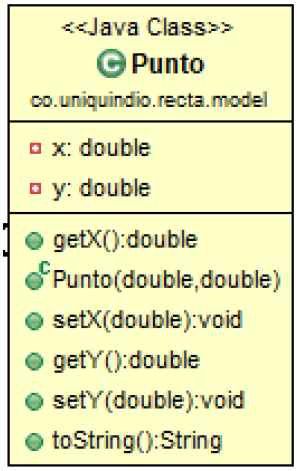
1. Identificar las entidades o clases. Lo primero que debe hacer es resaltar los nombres o sustantivos.

Se desea crear una Aplicación que permita crear una Recta. Esta figura está definida por 2 Puntos. Cada uno de ellos tiene una coordenada en x y una coordenada en y. La aplicación debe permitir cambiar los puntos, mostrar la ecuación y calcular la pendiente.

ENTIDAD	DESCRIPCIÓN
`Recta`	Es la entidad más importante del mundo del problema.
`Punto`	Forma parte de la Recta

2. Modelar las características e identificar el comportamiento

IDENTIFICACIÓN Y MODELAMIENTO DE ATRIBUTOS PARA PUNTO



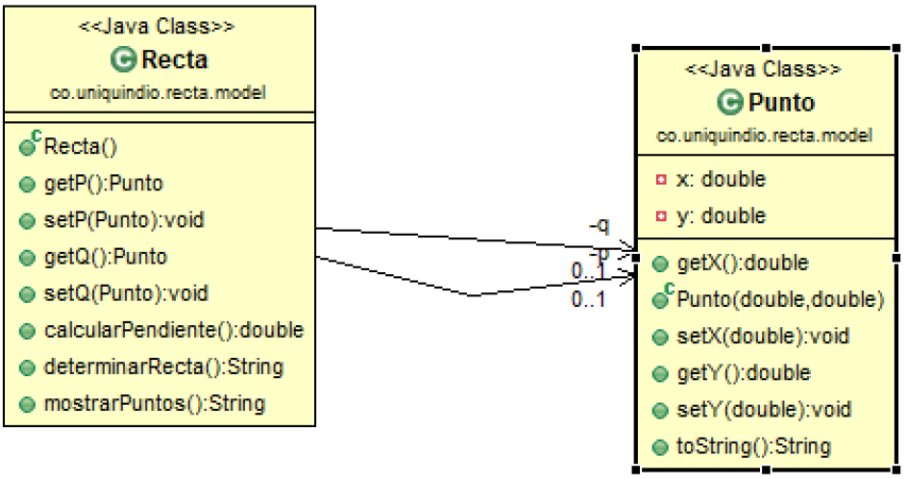
ATRIBUTO	VALORES POSIBLES	COMENTARIOS
`X`	Valor real	Recibe cualquier valor real, ya sea positivo o negativo
`Y`	Valor real	Recibe cualquier valor real, ya sea positivo o negativo

IDENTIFICACIÓN DE MÉTODOS

NOMBRE	DESCRIPCIÓN
`setX (double x)`	Inicializa el valor de la variable `x` de la clase Punto
`setY (double y)`	Inicializa el valor de la variable `y` de la clase Punto

`getX ()`	Devuelve el valor de `x` de la clase Punto
`getY ()`	Devuelve el valor de `y` de la clase Punto
`Punto (double x, double y)`	Constructor de la clase Punto

IDENTIFICACIÓN Y MODELAMIENTO DE ATRIBUTOS PARA RECTA



ATRIBUTO	VALORES POSIBLES	COMENTARIOS
`p`	Objeto de tipo Punto	Recibe un Punto. Recuerda que un Punto está formado por una coordenada en x y una coodenada en y.
`q`	Objeto de tipo Punto	Recibe un Punto

IDENTIFICACIÓN DE MÉTODOS

NOMBRE	DESCRIPCIÓN
`setP (punto miP)`	Permite fijar el punto p
`setQ (punto miQ)`	Permite fijar el punto q
`getM`	Devuelve el valor de la pendiente

`calcularPendiente()`	Se asigna valor a la pendiente
`determinarRecta()`	Devuelve un string con la fórmula de la recta

3. Las relaciones entre las clases del mundo

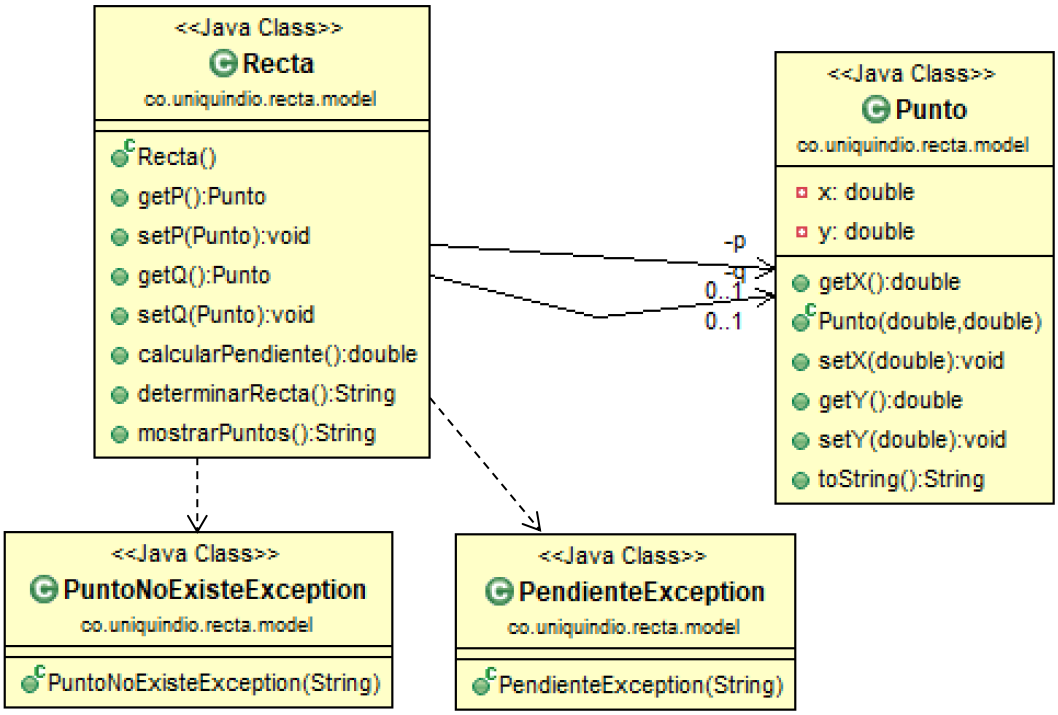


Figura 83 Diagrama de clases para el caso de estudios de la Recta

2.2.2.Elementos de un programa

La clase Punto, ver Programa 17, contiene los atributos de la clase x y y. Además, de un método constructor que permite inicializar los atributos de la clase. También están presentes los métodos get y set.

Programa 17 Clase Punto

```
package co.uniquindio.recta.model;

/**
 * @version 1.0
 * @author  Sonia Jaramillo Valbuena
 * @author  Sergio A. Cardona
 * Maneja la informacion referente al Punto
 */

public class Punto
{
/**
 * Atributos de la clase
 */


  private double x;
  private double y;


/**
 * método accesor
 * @return double, valor de la coordenada en x
 */
  public double getX()
  {
       return x;
  }

/**
 * método constructor
 */

public Punto(double x, double y) {
      super();
      this.x = x;
      this.y = y;
}



/**
 * Método modificador
 * @param x, variable double que tiene el
 *  valor de la coordenada en x
 */
```

Programa 17 Clase Punto

```
 public void setX(double x)
 {
   this.x = x;
 }

/**
 * Permite obtener el valor de y
 * @return double, coordenada en y
 */
 public double getY()
 {
   return y;
 }

/**
 * Método modificador
 * @param y, variable de tipo double
 */
 public void setY( double y )
 {
   this.y = y;
 }
/**
 * Devuelve un string con la representación del punto
 * @return Un string con la representación del punto
 */
 public String toString()
 {
       return "("+x+", "+y+")";
 }
}
```

La clase Recta implementada en Programa 18 tiene dos atributos de tipo punto, p y q respectivamente. Dentro de la clase Recta se contruyen, entre otros, 2 métodos para inicializar los puntos setP y setQ.

El método calcular pendiente lanza una excepción si alguno de los puntos no ha sido definido o si la diferencia entre las coordenadas en x entre ambos puntos es cero, dado que la división se indeterminaría. Para calcular la pendiente se usa la fórmula $m = (y_2 – y_1) / (x_2 - x_1)$

El método determinarRecta() devuelve la ecuación de la recta, siempre y cuando no se lance una excepción. La ecuación utilizada para obtener la ecuación es $y - y_1 = \frac{y_2 - y_1}{x_2 - x_1} (x - x_1)$. La ecuación se obtiene mediante concatenación de términos, la expresión usada para ello es ecuacion = "(y- " + p.getY() + ")/(x- " + p.getX() + ") = " + m;

Programa 18 Clase Recta

```
package co.uniquindio.recta.model;

public class Recta
{
  /**
   * Atributos de la recta
   */
  private Punto  p;
  private Punto  q;


/**
 * Metodo accesor
 * @return
 */
public Punto getP() {
	return p;
}

/**
 * Permite modificar el punto p
 * @param p Es el punto p
 */
public void setP(Punto p) {
	this.p = p;

}

/**
 * Metodo accesor
 * @return q
 */
public Punto getQ() {
	return q;
}

/**
 * Metodo modificador
 * @param q Es el segundo punto
 */
public void setQ(Punto q) {
	this.q = q;

}


/**
 * Método que permite calcular el valor de la pendiente
 */
 public double calcularPendiente() throws PendienteException, PuntoNoExisteException
 {
 double m=0;

 if(p==null)
```

Programa 18 Clase Recta

```
        {throw new PuntoNoExisteException("El punto p no ha sido definido ");}
 if(q==null)
        {throw new PuntoNoExisteException("El punto q no ha sido definido  ");}

 if( q.getX() - p.getX() != 0 )
   {
     m = ( q.getY() - p.getY() ) /  ( q.getX() - p.getX() );
   }
 else
 {
        throw new PendienteException("Pendiente indeterminada");
 }
 return m;
 }


/**
 * Permite calcular la ecuacion de la Recta
 * @return La ecuacion
 * @throws PuntoNoExisteException
 */
 public String determinarRecta() throws PendienteException, PuntoNoExisteException
 {  String ecuacion="";
       double m= calcularPendiente();
    if(m!=0)
       {ecuacion = "(y- "   + p.getY() + ")/(x- " + p.getX() + ") = "    + m;
       }
    return ecuacion;
 }

 public String mostrarPuntos()
 {
        return p.toString()+"   "+q.toString();
 }
}
```

La clase PendienteException cuyo código se encuentra en Programa 19 permite el manejo de una excepción personalizada, que se usa cuando el divisor es cero.

Programa 19 Clase PendienteException

```
package co.uniquindio.recta.model;
/**
 * Clase personalizada para la manejar la division por cero
 * @author sonia
 * @author sergio
 */
public class PendienteException extends Exception {
/**
 * Constructor de la clase
```

```
 * @param mensaje
 */
	public PendienteException(String mensaje) {
		super(mensaje);
}
}
```

La clase PuntoNoExisteException se usa para la lanzar una excepción si alguno de los puntos es null.

Programa 20 Clase PuntoNoExisteException

```
package co.uniquindio.recta.model;
/**
 * Clase para manejar la excepcion si un punto no ha sido definido
 * @author sonia
 * @author sergio
 */
public class PuntoNoExisteException extends Exception {
/**
 * Constructor de la clase
 * @param mensaje
 */
public PuntoNoExisteException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

El Programa 21 contiene el método main y la referencia a la clase principal del mundo del problema. Allí se implementan los métodos inicializarLayoutRaiz(), mostrarVistaRecta() y crearVentanaPunto(**int** opcion). El método mostrar vistarRecta permite, entre otras acciones, obtener el controlador para la recta, igual ocurre con el método crearVentanaPunto, que permite obtener el controlador para el punto. Los métodos setQ, setP, calcularPendiente() y determinarRecta() permiten la interaccion con la clase principal del la lógica, es decir, con Recta.

Programa 21 Clase Principal

```
package co.uniquindio.recta;
import java.io.IOException;

import co.uniquindio.recta.model.PuntoNoExisteException;
import co.uniquindio.recta.model.PendienteException;
import co.uniquindio.recta.model.Punto;
import co.uniquindio.recta.model.PuntoNoExisteException;
import co.uniquindio.recta.model.Recta;
import co.uniquindio.recta.view.ControladorPunto;
import co.uniquindio.recta.view.ControladorRecta;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
```

Programa 21 Clase Principal

```
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Modality;
import javafx.stage.Stage;

public class Principal extends Application {

      private Stage escenarioPrincipal;
    private BorderPane layoutRaiz;
    private Recta miRecta;
    @Override
    public void start(Stage primaryStage) {
        miRecta= new Recta();
        this.escenarioPrincipal = primaryStage;
        this.escenarioPrincipal.setTitle("Aplicación sobre una recta");
        inicializarLayoutRaiz();
         mostrarVistaRecta();
    }

    /**
     * Inicializa el layout raiz
     */
    public void inicializarLayoutRaiz() {
        try {
            // Carga el root layout desde un archivo xml
            FXMLLoader cargador = new FXMLLoader();
            cargador.setLocation(Principal.class.getResource("view/LayoutRaiz.fxml"));
            layoutRaiz = (BorderPane) cargador.load();
            // Muestra la escena que contiene el RootLayout
            Scene scene = new Scene(layoutRaiz);
            escenarioPrincipal.setScene(scene);
            escenarioPrincipal.show();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    /**
     * Muestra la vista Recta
     */
    public void mostrarVistaRecta() {
        try {
            // Carga la vista de la recta.
            FXMLLoader loader = new FXMLLoader();
            loader.setLocation(Principal.class.getResource("view/RectaVista.fxml"));
            AnchorPane vistaRecta = (AnchorPane) loader.load();

            // Fija la vista de la person en el centro del root layout.
            layoutRaiz.setCenter(vistaRecta);
            // Acceso al controlador.
            ControladorRecta miControlador = loader.getController();
```

Programa 21 Clase Principal

```
            miControlador.setMiVentanaPrincipal(this);

        } catch (IOException e) {
         System.out.println(e.getMessage());
        }
    }

      public  boolean crearVentanaPunto(int opcion) throws IOException
      {     try {
              FXMLLoader loader = new FXMLLoader();
         loader.setLocation(Principal.class.getResource("view/PuntoVista.fxml"));
         BorderPane vistaPunto = ( BorderPane) loader.load();

         Stage dialogStage = new Stage();
         dialogStage.setTitle("Crear Punto");
         dialogStage.initModality(Modality.WINDOW_MODAL);
         dialogStage.initOwner(escenarioPrincipal);
         Scene scene = new Scene(vistaPunto);
         dialogStage.setScene(scene);

      // Acceso al controlador.
         ControladorPunto miControlador = loader.getController();
         miControlador.inicializar(this,opcion, dialogStage);
         dialogStage.showAndWait();

         return miControlador.isOkClicked();
    } catch (IOException e) {
        e.printStackTrace();
        return false;
    }
      }
    /**
     *
     * @return
     */
    public Stage getPrimaryStage() {
        return escenarioPrincipal;
    }

    public static void main(String[] args) {
        launch(args);
    }
    /**
     * Metododo para imprimir un mensaje
     * @param mensaje
     * @param miA
     * @param titulo
     * @param cabecera
     * @param contenido Es el mensaje a mostrar
     * @param escenarioPrincipal Es el scenario sobre el que se muestra el mensaje
     */
    public static void mostrarMensaje(String mensaje, AlertType miA, String titulo,
String cabecera, String contenido, Stage escenarioPrincipal )
```

Programa 21 Clase Principal

```
    {
        // Muestra el dialogo
        Alert alert = new Alert(miA);
        alert.initOwner(escenarioPrincipal);
        alert.setTitle(titulo);
        alert.setHeaderText(cabecera);
        alert.setContentText(contenido);
         alert.showAndWait();
    }
/**
 * Devuelve el escenario
 * @return escenarioPrincipal
 */
      public Stage getEscenarioPrincipal() {
            return escenarioPrincipal;
      }
/**
 * Permite fijar el escenario
 * @param escenarioPrincipal
 */
      public void setEscenarioPrincipal(Stage escenarioPrincipal) {
            this.escenarioPrincipal = escenarioPrincipal;
      }
      /**
       * Metodo para modificar q
       * @param q
       */
      public void setQ(Punto q) {
            miRecta.setQ(q);
      }
      /**
       * Metodo para modificar p
       * @param p
       */
      public void setP(Punto p) {
            miRecta.setP(p);
      }
      /**
       * Metodo para calcular la pendiente
       * @return la pendiente
       * @throws PendienteException
       * @throws PuntoNoExisteException
       */
      public double calcularPendiente() throws PendienteException, PuntoNoExisteException
       {
            return miRecta.calcularPendiente();
       }
      /**
       * Metodo para obtener la recta
       * @return la representacion en String de la recta
       * @throws PendienteException
       * @throws PuntoNoExisteException
       */
```

Programa 21 Clase Principal

```
	public String determinarRecta() throws PendienteException, PuntoNoExisteException
	 {
		return miRecta.determinarRecta();
	 }

}
```

El ControladorPunto implementado en Programa 22 permite obtener la información de los campos de texto y crear los puntos. Las instrucciones requeridas para ello se presentan a continuación.

```
double x = Double.parseDouble(xTextField.getText());
double y = Double.parseDouble(yTextField.getText());
Punto punto = new Punto(x, y);
if (opcion == 0) {
	miVentanaPrincipal.setP(punto);
} else {
	miVentanaPrincipal.setQ(punto);
	}
```

Observe que la variable opción permite inicializar el punto apropiado, p ó q, dependiendo del botón que se presione.

Programa 22 ControladorPunto

```
package co.uniquindio.recta.view;

import java.awt.Button;



import co.uniquindio.recta.Principal;
import co.uniquindio.recta.model.Punto;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;

/**
 * Permite editar los datos de la recta
 *
 * @author Sonia Jaramillo
 */
public class ControladorPunto {
	/**
	 * Atributos de la clase
	 */
	@FXML
	private TextField xTextField;
```

Programa 22 ControladorPunto

```
	@FXML
	private TextField yTextField;
	@FXML
	private Button guardarButton;

	// VentanaPrincipal
	private Principal miVentanaPrincipal;
	private Stage dialogStage;
	private int opcion;
	private boolean okClicked = false;

	/**
	 * Inicializa la clase contenedor.
	 */
	@FXML
	private void initialize() {
	}

	/**
	 * Permite inicializar la ventana principal, y abrir la ventana correspondiente
	 * dependiendo de la opcion
	 *
	 * @param miVentanaPrincipal
	 * @param opcion             Toma valores de 0 y 1, 0 para p y 1 para q
	 * @param dialogStage        Es la stage sobre el que se muestra la info
	 */
	public void inicializar(Principal miVentanaPrincipal, int opcion, Stage
dialogStage) {
		this.miVentanaPrincipal = miVentanaPrincipal;
		this.opcion = opcion;
		this.dialogStage = dialogStage;
	}

	/**
	 * Metodo para crear el punto
	 */
	@FXML
	private void crearPunto() {
		double x = Double.parseDouble(xTextField.getText());
		double y = Double.parseDouble(yTextField.getText());
		Punto punto = new Punto(x, y);
		if (opcion == 0) {
			miVentanaPrincipal.setP(punto);
		} else {
			miVentanaPrincipal.setQ(punto);

		}

		okClicked = true;
		dialogStage.close();
	}

	/**
```

Programa 22 ControladorPunto

```
         * Retorna true si el boton es cliqueado
         *
         * @return
         */
        public boolean isOkClicked() {
              return okClicked;
        }
}
```

En el Programa 23 se presenta el controlador de la Recta. Esta clase se encargar de invocar métodos tales como mostrarPendiente() y mostrarEcuacion(), que están implementados en la clase Principal. Las instrucciones usadas para ello son respectivamente, pendiente = miVentanaPrincipal.calcularPendiente() y ecuacion = miVentanaPrincipal.determinarRecta();

Programa 23 ControladorRecta

```
package co.uniquindio.recta.view;

import java.awt.Button;
import java.io.IOException;

import co.uniquindio.recta.Principal;
import co.uniquindio.recta.model.PuntoNoExisteException;
import co.uniquindio.recta.model.PendienteException;
import co.uniquindio.recta.model.Punto;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Modality;
import javafx.stage.Stage;
import javafx.stage.StageStyle;

/**
 * Permite editar los datos de la recta
 *
 * @author Sonia Jaramillo
 */
public class ControladorRecta {
        /**
         * Atributos de la clase
         */
        @FXML
```

Programa 23 ControladorRecta

```
	private Button puntoPButton;
	@FXML
	private Button puntoQButton;
	@FXML
	private Button pendienteButton;
	@FXML
	private Button ecuacionButton;

	// VentanaPrincipal
	private Principal miVentanaPrincipal;

	/**
	 * Inicializa la clase contenedor.
	 */
	@FXML
	private void initialize() {
	}

	/**
	 * Metodo accesor
	 */

	public Principal getMiVentanaPrincipal() {
		return miVentanaPrincipal;
	}

	/**
	 * Permite fijar la ventanar
	 *
	 * @param miVentanaPrincipal
	 */
	public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
		this.miVentanaPrincipal = miVentanaPrincipal;
	}

	/**
	 * Permite crear el punto p
	 *
	 * @throws IOException
	 */
	@FXML
	public void crearPuntoP() throws IOException {
		miVentanaPrincipal.crearVentanaPunto(0);
	}

	/**
	 * Permite crear el punto q
	 *
	 * @throws IOException
	 */
	@FXML
	public void crearPuntoQ() throws IOException {
		miVentanaPrincipal.crearVentanaPunto(1);
```

Programa 23 ControladorRecta

```
	}

	/**
	 * Se invoca cuando el usuario de clic en pendiente
	 */
	@FXML
	private void mostrarPendiente() {
		double pendiente;
		try {
			pendiente = miVentanaPrincipal.calcularPendiente();
			miVentanaPrincipal.mostrarMensaje("Pendiente", AlertType.INFORMATION,
"", "", "La pendiente " + pendiente,
					miVentanaPrincipal.getEscenarioPrincipal());
		} catch (PendienteException e) {
			// TODO Auto-generated catch block
			miVentanaPrincipal.mostrarMensaje("ERROR", AlertType.ERROR, "", "",
e.getMessage(),
					miVentanaPrincipal.getEscenarioPrincipal());
		} catch (PuntoNoExisteException e) {
			// TODO Auto-generated catch block
			miVentanaPrincipal.mostrarMensaje("ERROR", AlertType.ERROR, "", "",
e.getMessage(),
					miVentanaPrincipal.getEscenarioPrincipal());

		}
	}

	/**
	 * Se invoca cuando el usuario de clic en ecuacion
	 */
	@FXML
	private void mostrarEcuacion() {
		String ecuacion;
		try {
			ecuacion = miVentanaPrincipal.determinarRecta();
			miVentanaPrincipal.mostrarMensaje("Ecuación", AlertType.INFORMATION,
"", "", "La ecuacion es  " + ecuacion,
					miVentanaPrincipal.getEscenarioPrincipal());
		} catch (PendienteException e) {
			// TODO Auto-generated catch block
			miVentanaPrincipal.mostrarMensaje("ERROR", AlertType.ERROR, "", "",
				e.getMessage() + "\nNo se puede obtener la ecuación ",
miVentanaPrincipal.getEscenarioPrincipal());

		} catch (PuntoNoExisteException e) {
			// TODO Auto-generated catch block
			miVentanaPrincipal.mostrarMensaje("ERROR", AlertType.ERROR, "", "",
e.getMessage(),
					miVentanaPrincipal.getEscenarioPrincipal());
		}
	}
```

Programa 23 ControladorRecta

```
}
```

El Programa 24 contiene el código necesario para la creación de la interfaz presentada en la *Figura 84*.

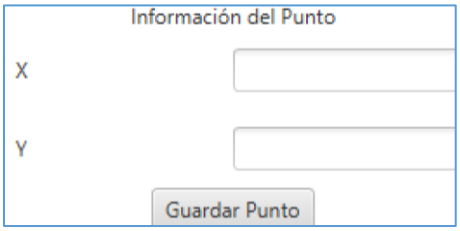


Figura 84 Interfaz para agregar un punto

Programa 24 Archivo PuntoVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="132.0" prefWidth="264.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.recta.view.ControladorPunto">
   <top>
      <Label text="Información del Punto" BorderPane.alignment="CENTER" />
   </top>
   <bottom>
      <Button mnemonicParsing="false" onAction="#crearPunto" text="Guardar Punto"
BorderPane.alignment="CENTER" />
   </bottom>
   <center>
      <GridPane prefHeight="99.0" prefWidth="259.0" BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Label prefHeight="17.0" prefWidth="45.0" text="  X">
               <GridPane.margin>
                  <Insets />
```

Programa 24 Archivo PuntoVista.fxml

```
               </GridPane.margin>
            </Label>
            <Label prefHeight="17.0" prefWidth="26.0" text="  Y" GridPane.rowIndex="1" />
            <TextField GridPane.columnIndex="1" fx:id="xTextField" />
            <TextField fx:id="yTextField" GridPane.columnIndex="1" GridPane.rowIndex="1"
/>
         </children>
      </GridPane>
   </center>
</BorderPane>
```

El Programa 25 presenta el código para la creación de la ventana correspondiente al menú principal de la aplicación, que se presenta en la Figura 85.

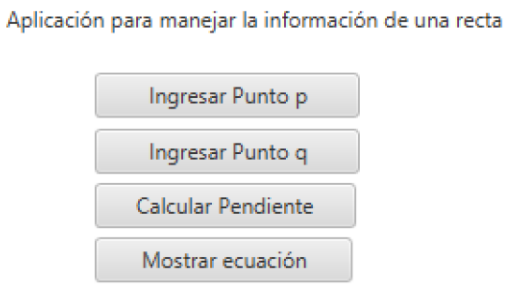


Figura 85 Menú principal de la aplicación.

Programa 25 RectaVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-
Infinity" prefHeight="255.0" prefWidth="359.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.recta.view.ControladorRecta">
   <children>
      <Label layoutX="53.0" layoutY="40.0" text="Aplicación para manejar la información
de una recta" />
      <GridPane layoutX="102.0" layoutY="75.0" prefHeight="121.0" prefWidth="195.0">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
```

Programa 25 RectaVista.fxml

```
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Button mnemonicParsing="false" onAction="#crearPuntoP" prefHeight="25.0"
prefWidth="147.0" text="Ingresar Punto p" />
            <Button mnemonicParsing="false" onAction="#crearPuntoQ" prefHeight="25.0"
prefWidth="147.0" text="Ingresar Punto q" GridPane.rowIndex="1" />
            <Button mnemonicParsing="false" onAction="#mostrarPendiente"
prefHeight="25.0" prefWidth="145.0" text="Calcular Pendiente" GridPane.rowIndex="2" />
            <Button mnemonicParsing="false" onAction="#mostrarEcuacion" prefHeight="25.0"
prefWidth="143.0" text="Mostrar ecuación" GridPane.rowIndex="3" />
         </children>
      </GridPane>
   </children>
</AnchorPane>
```

Finalmente se presenta el código para crear el LayoutRaiz, ver Figura 86

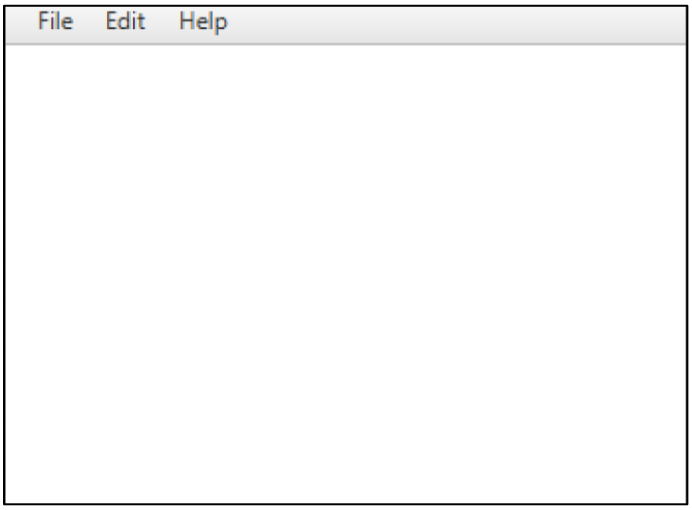


Figura 86 Layout Raiz

Programa 26 LayoutRaiz

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>
```

Programa 26 LayoutRaiz

```
<BorderPane prefHeight="259.0" prefWidth="342.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1">
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem mnemonicParsing="false" text="Close" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Edit">
            <items>
              <MenuItem mnemonicParsing="false" text="Delete" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Help">
            <items>
              <MenuItem mnemonicParsing="false" text="About" />
            </items>
          </Menu>
        </menus>
      </MenuBar>
   </top>
</BorderPane>
```

2.3. Caso de estudio 2 Unidad II: Juguetería

Se desea crear una Aplicación para manejar la información de una Tienda. En la tienda se venden juguetes, cada Juguete tiene un código, un nombre, un tipo (O->Femenino 1->Masculino 2-> Unisex) y un precio [2][3][4].

Se debe permitir agregar un nuevo juguete
Informar cuántos juguetes hay por cada tipo
Informar la cantidad total de juguetes
Informar el valor total de todos los juguetes que hay en la tienda
Disminuir existencias de un juguete
Informar el tipo del cuál hay más juguetes
Ordenar de menor a mayor la cantidad de existencias de juguetes por tipo. Ejemplo si hay 5 juguetes femeninos, 2 masculinos y 7 unisex, el ordenamiento quedaría como 2 masculino, 5 femenino y 7 unisex.
Agregar existencias de un juguete

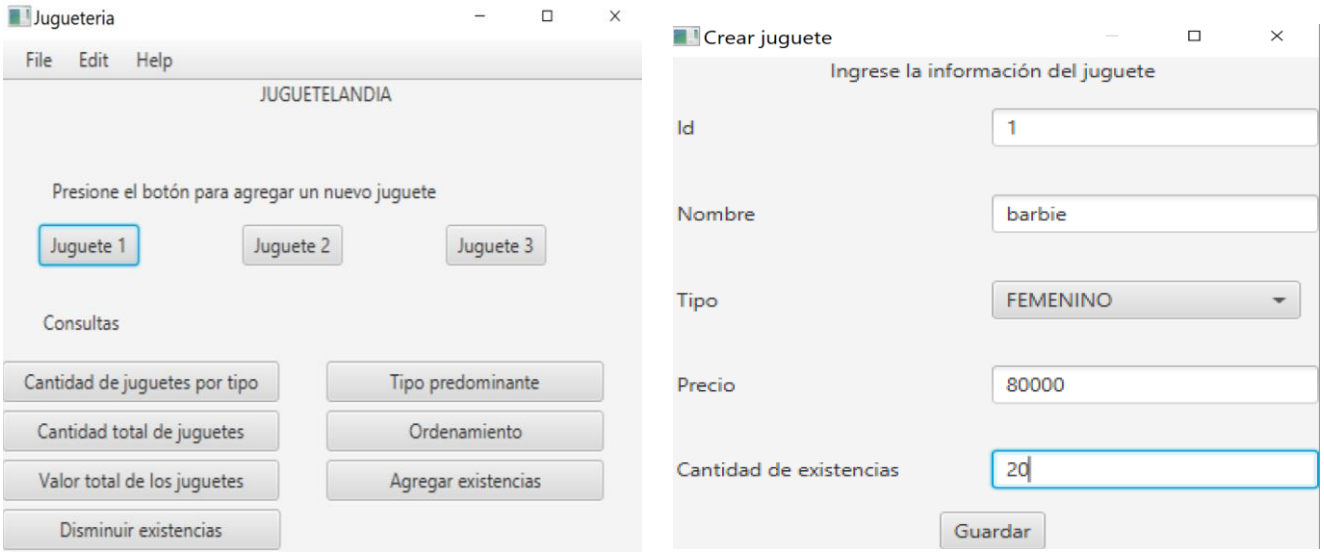


Figura 87 Interfaz aplicación Juguetelandia

2.3.1. Comprensión del problema

Tal como se planteó en el capítulo previo, lo primero que debe hacerse para dar solución a un problema es entender dicho problema, para ello es importante identificar los requisitos funcionales y los no funcionales, además del modelo del mundo del problema.

a) Requisitos funcionales

NOMBRE	R1 – Agregar un juguete
RESUMEN	Permite agregar un juguete
ENTRADAS	
Id, nombre, tipo, precio, cantidad de existencias	
RESULTADOS	
El juguete ha sido agregado	

NOMBRE	R2 – Informar la cantidad de juguetes que hay por tipo
RESUMEN	Permite contar la cantidad de juguetes que hay de un determinado tipo
ENTRADAS	
El tipo	
RESULTADOS	
La cantidad de juguetes por tipo	

NOMBRE	R3 – Informar la cantidad total de juguetes
RESUMEN	Permite calcular la cantidad total de juguetes que hay en la tienda
ENTRADAS	
Ninguna	
RESULTADOS	
La cantidad total del juguetes	

NOMBRE	R4 – Informar el valor total de todos juguetes
RESUMEN	Se suma el valor total de los juguetes y se tiene en cuenta la cantidad de existencias
ENTRADAS	
Ninguna	
RESULTADOS	
El valor total de los juguetes	

NOMBRE	R5 – Disminuir la cantidad de existencias de un juguete
RESUMEN	Se recibe la cantidad de elementos que se desea disminuir y se resta de la cantidad existente
ENTRADAS	
El id del juguete y la cantidad de existencias a disminuir	
RESULTADOS	
El valor promedio	

NOMBRE	R6 – Informar el tipo del cual hay más juguetes
RESUMEN	Se debe realizar una comparación entre los 3 tipos para determinar cuál es el que tiene mayor cantidad de juguetes. Los posibles salidas son O->Femenino, 1->Masculino y 2-> Unisex.
ENTRADAS	
Ninguna	
RESULTADOS	
El tipo del cual hay más juguetes	

NOMBRE	R7 – Ordenar la cantidad de existencias de juguetes por tipo
RESUMEN	Se debe ordenar de menor a mayor la cantidad de existencias de juguetes por tipo. Ejemplo, si hay 5 juguetes femeninos, 2 masculinos y 7 unisex, el ordenamiento quedaría como 2 masculino, 5 femenino y 7 unisex.
ENTRADAS	
Ninguna	
RESULTADOS	
La ordenación de acuerdo a la cantidad de existencias por tipo	

NOMBRE	R8 – Aumentar la cantidad de existencias de un juguetes
RESUMEN	Se recibe la cantidad de elementos que se desea aumentar y se suma a la cantidad existente de ese juguete.
ENTRADAS	
El id del juguete y la cantidad de existencias a disminuir	
RESULTADOS	
El valor promedio	

2.3.2.Especificación de la Interfaz de usuario

Una parte importante del diseño de la solución es la especificación de la interfaz de usuario. La siguiente figura muestra el diseño seleccionado para este caso de estudio:

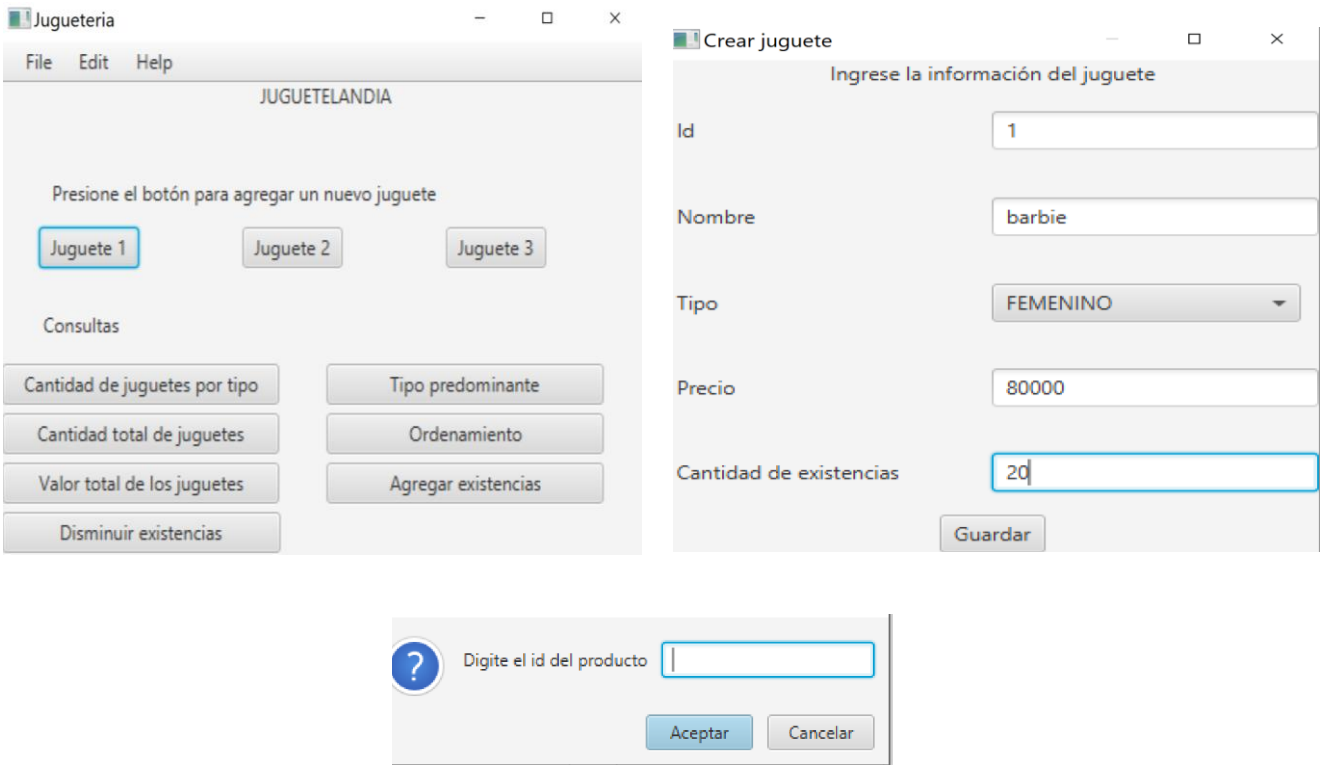


Figura 88 Interfaz aplicación Juguetelandia

2.3.3. Otros elementos de modelado (constantes para representar el dominio de un atributo y enumeraciones)

En el nivel anterior se introdujo el concepto de tipos de datos simples de datos (byte, short, int, long, float, double, char y **boolean**) y el tipo String. También se explicaron los diferentes tipos de operadores. Ahora se introducirá el concepto de constantes, a través de las cuales es posible definir datos que no cambiarán a lo largo de la ejecución de un programa.

La declaración de una constante incluye la palabra final. Se debe tener en cuenta que los nombres de las constantes se escriben en mayúscula y si están formados por más de una palabra se separan con guión bajo. Una constante puede usarse para definir el dominio de un atributo o para representar valores inmutables. A continuación se muestran algunos ejemplos del uso de constantes [2][3][4].

Las siguientes constantes representan valores inmutables

```
public final static double CONSTANTE_PI  = 3.1416;
public final static double IVA = 0.16;
```

Las siguientes constantes representan el dominio de un atributo

```
public final static int FEMENINO  = 1;
public final static int MASCULINO = 2;
```

El concepto de constantes aplicado al caso de estudio de la Juguetería:

```
public class Juguete {

//Constantes
public final static int FEMENINO=0;
public final static int MASCULINO=1;
public final static int UNISEX=2;

//Atributos
private String id;
private String nombre;
private int tipo;
private double precio;
private int cantidadExistencias;}
```

En la columna de la izquierda se puede observar:

-Se declara FEMENINO, para representar el primer valor posible del atributo tipo de juguete. Se le asigna un valor de cero.

-Se declara MASCULINO, para representar el segundo valor posible del atributo tipo de juguete. Se le asigna un valor de 1.

-Se declara UNISEX, para representar el tercer valor posible del atributo tipo de juguete. Se le asigna un valor de 2.

-Se declara el atributo “tipo”

Otra opción para especificar el tipo de juguete son las enumeraciones. Una enumeración puede verse como una clase especial, que restringue la creación de objetos a los que se define explícitamente dentro de la enumeración. Una enumeración no puede ser instanciada. El Programa 27 muestra como una enumeración se usa para este caso de estudio.

Programa 27 Enumeracion Tipo y declaración dentro de la clase Juguete

```
package co.uniquindio.jugueteria.model;

public enum Tipo {
	//Tipos de vehículos disponibles
	FEMENINO(0), MASCULINO (1), UNISEX(2);

	/**
	 * Atributo de la enumeraci[on
	 */
	private int numTipo;
/**
 * Metodo constructor
 * @param numTipo
 */
	private Tipo(int numTipo) {
		this.numTipo = numTipo;

	}
/**
 * Metodo accesor
 * @return
 */
	public int getNumTipo() {
		return numTipo;
	}
/**
 * Representacion en string del valor elegido
 */
	public String toString()
	{
		String tipo=" ";
		switch(numTipo)
		{case 0: tipo="FEMENINO";
				break;
		case 1:tipo="MASCULINO";
			break;
		case 2: tipo="UNISEX";
		}
	return tipo;
	}
	}
```

```
public class Juguete {
/**
 * Atributos de la clase
 */
private String id;
private String nombre;
private Tipo tipo;
private double precio;
private int cantidadExistencias;
}
```

Para efectuar una comparación con una enumeración se hace con el operador ==, ejemplo:

```
Tipo miTipo=Tipo.MASCULINO;
If(miTipo==Tipo.FEMENINO)
{
  //se ejecuta alguna acción
}
```

b) El modelo del mundo del problema

Las actividades que se deben realizar para construir el modelo del mundo son las siguientes:

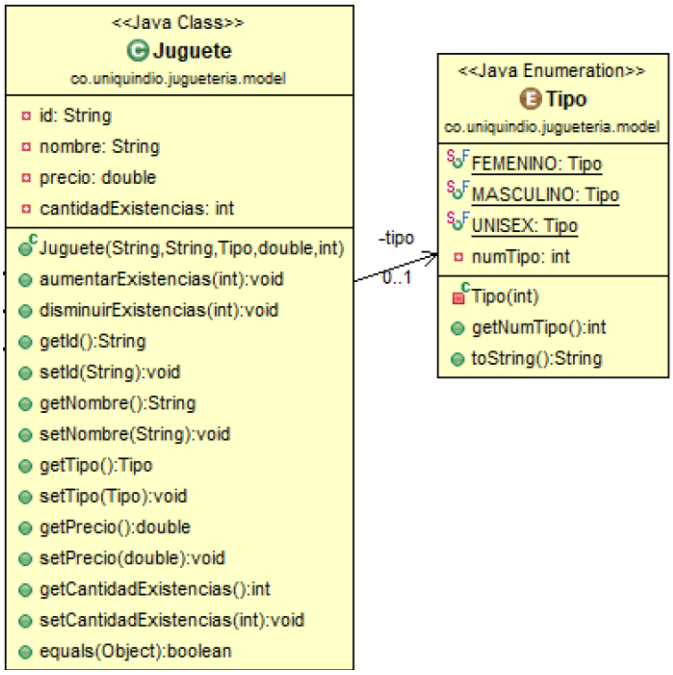
Identificar las entidades o clases.
Las clases identificadas son las siguientes:

ENTIDAD DEL MUNDO	DESCRIPCIÓN
Tienda	Es la entidad más importante del mundo del problema.
Juguete	Es un atributo de la tienda

Identificar los métodos

Los atributos y métodos identificados para cada una de las clases se muestran en las siguientes tablas.

IDENTIFICACIÓN Y MODELAMIENTO DE ATRIBUTOS PARA JUGUETE



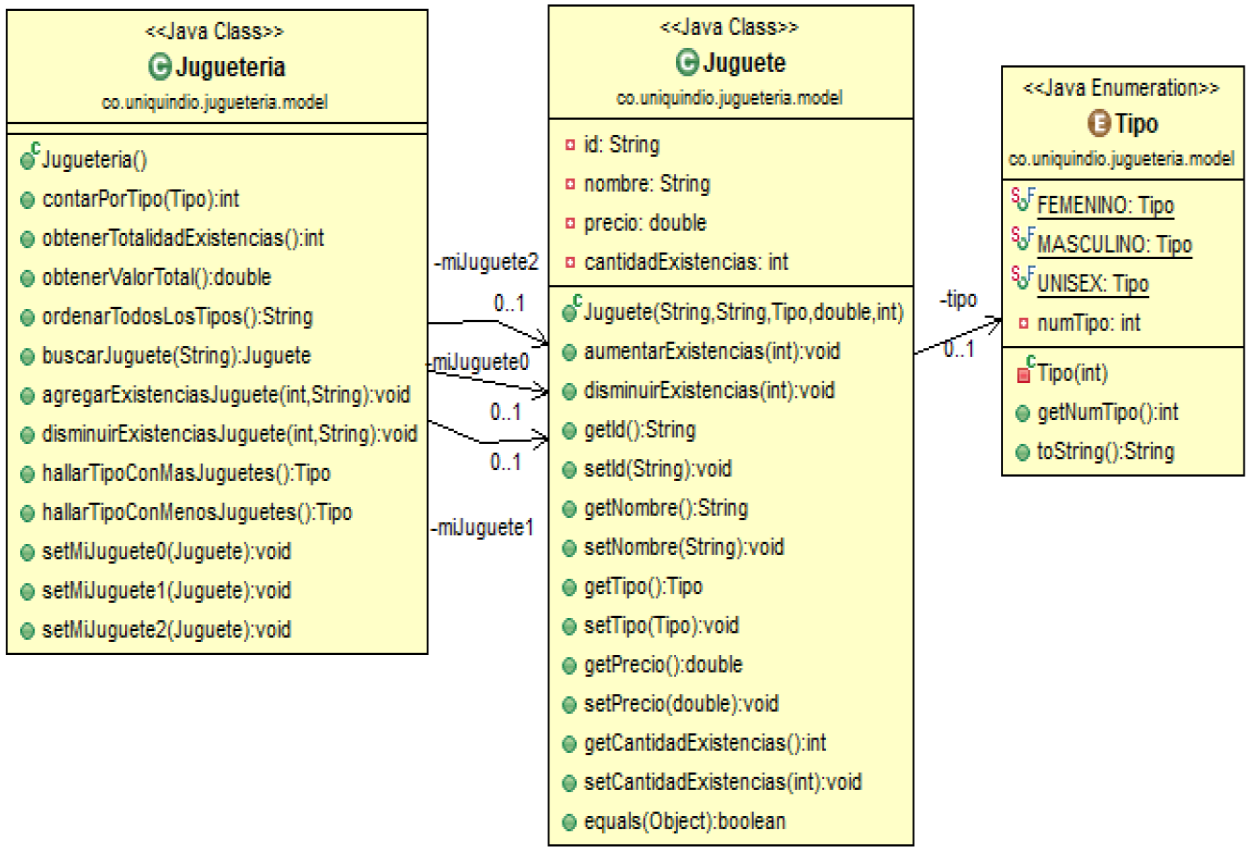
ATRIBUTO	VALORES POSIBLES	Tipo de dato
id	Cadena de caracteres	String
nombre	Cadena de caracteres	String

tipo precio cantidadExistencias	Enumeracion,valores FEMENINO(0), MASCULINO (1), UNISEX(2); Valor real positivo Valor entero positivo	Tipo double int

IDENTIFICACIÓN DE MÉTODOS

```
//Además de los métodos get y set que pueden visualizarse en el diagrama, se identifican -
//otros tales como
//Constructor de la clase juguete
public Juguete(String id, String nombre, Tipo tipo, double precio, int
cantidadExistencias)

//Metodo para aumentar la cantidad de existencias
public void aumentarExistencias(int cantidad)

//Metodo para disminuir la cantidad de existencias
public void disminuirExistencias(int cantidad)
```

IDENTIFICACIÓN Y MODELAMIENTO DE ATRIBUTOS PARA JUGUETERIA



ATRIBUTO	VALORES POSIBLES	Tipo de dato

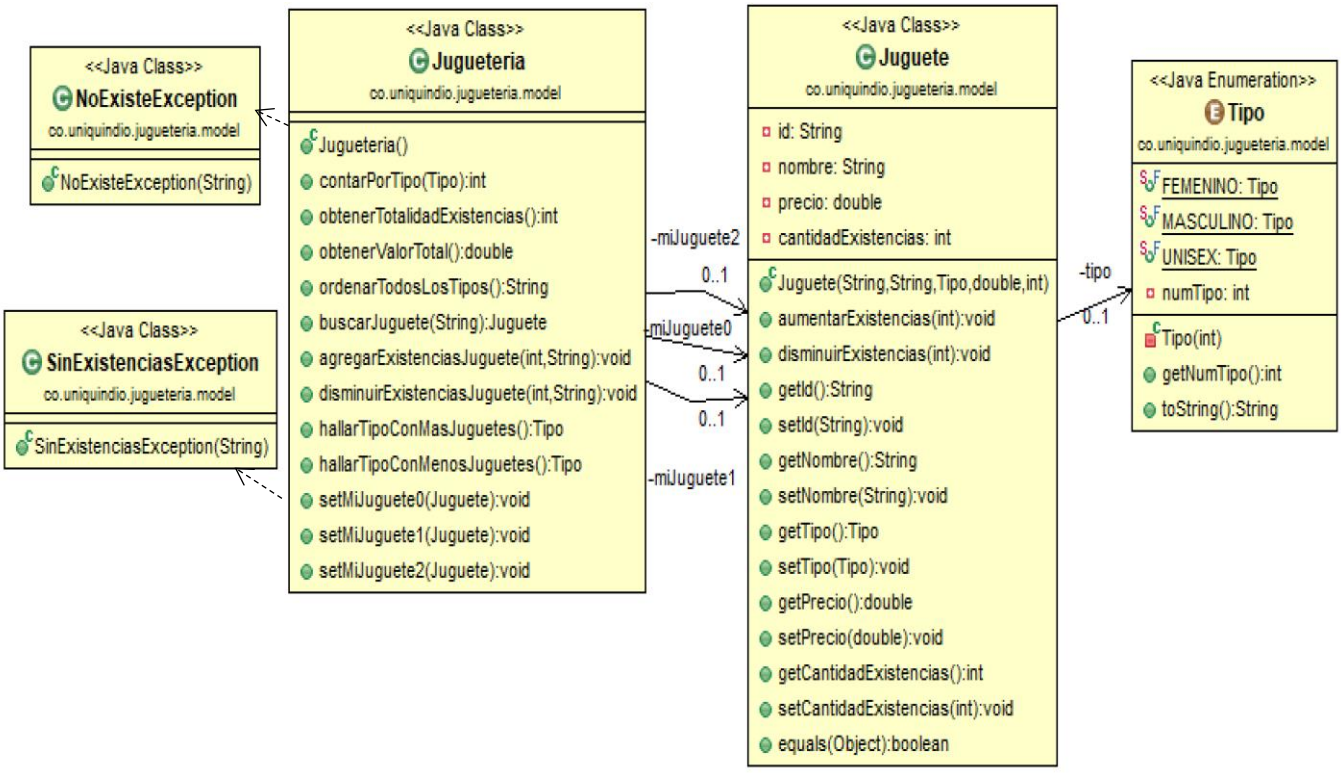
miJuguete1 miJuguete2 miJuguete3	Referencia a objeto de tipo juguete Referencia a objeto de tipo juguete Referencia a objeto de tipo juguete	Juguete Juguete Juguete

IDENTIFICACIÓN DE MÉTODOS

(para crear una jugueteria e inicializar variables se debe construir un constructor)

```
public Jugueteria()

El método setMiJuguete permite fijar el juguete
setMiJuguete1(Juguete miJuguete1)
setMiJuguete2(Juguete miJuguete2)

setMiJuguete3(Juguete miJuguete3)

//Devuelve la cantidad de juguetes por tipo

public int contarPorTipo(Tipo tipo)

//Obtiene la totalidad de existencias
public int obtenerTotalidadExistencias()

//Obtiene el valor total de los juguetes
public double obtenerValorTotal()

//Ordena los juguetes por tipo, de acuerdo a la cantidad de existencias
public String ordenarTodosLosTipos()

//Agrega existencias a un juguets
public void agregarExistenciasJuguete(int cantidad, String codigo)

//Disminuye existencias a un juguete
public void disminuirExistenciasJuguete(int cantidad, String codigo)

//Devuelve el tipo con mas juguets
public Tipo hallarTipoConMasJuguetes()
```

Las Relaciones entre las clases del mundo



La construcción de esta aplicación inicia con la clase Juguete, que se presenta en el Programa 28. Observe que los atributos del juguete son id, nombre, precio, cantidadExistencias y tipo. Tipo es una variable de tipo enumeración, que puede tomar uno de tres posibles valores FEMENINO(0), MASCULINO (1), UNISEX(2). Esta enumeración se implementa en el Programa 29. En la clase Juguete se implementa un método constructor. Además, del método disminuirExistencias, que tiene como parámetro **int** cantidad, si es posible restar esa cantidad se efectúa la operación mediante la instrucción cantidadExistencias-=cantidad, de lo contrario se lanza una excepción.

Programa 28 Clase Juguete

```
package co.uniquindio.jugueteria.model;
/**
 * Clase para construir un juguete
 * @author Sonia
 *
 */
public class Juguete {
/**
 * Atributos de la clase
 */
private String id;
```

Programa 28 Clase Juguete

```
private String nombre;
private Tipo tipo;
private double precio;
private int cantidadExistencias;
/**
 * Metodos constructor
 * @param id Es el id del jugute
 * @param nombre, Es el nombre del juguete
 * @param tipo, es el tipo del juguete
 * @param precio, es el precio del juguete
 * @param cantidadExistencias, es la cantidad de existencias del juguete
 */
public Juguete(String id, String nombre, Tipo tipo, double precio, int
cantidadExistencias) {
	this.id = id;
	this.nombre = nombre;
	this.tipo = tipo;
	this.precio = precio;
	this.cantidadExistencias = cantidadExistencias;
}
/**
 * Permite aumentar existencias
 * @param cantidad La cantidad a aumentar
 */
public void aumentarExistencias(int cantidad)
{
	cantidadExistencias+=cantidad;
}
/**
 * Permite disminuir existencias
 * @param cantidad La cantidad a disminuir
 */
public void disminuirExistencias(int cantidad) throws SinExistenciasException
{
	if(cantidadExistencias-cantidad>=0)
	 {cantidadExistencias-=cantidad;}
	else
	{
		throw new SinExistenciasException("No hay existencias del producto para
realizar la venta");
	}
}
/**
 * Metodo accesor
 * @return id
 */
public String getId() {
	return id;
}
/**
 * Metodo modificador
 * @param id
 */
public void setId(String id) {
```

Programa 28 Clase Juguete

```
		this.id = id;
	}
	/**
	 * Metodo accesor
	 * @return nombre
	 */
	public String getNombre() {
		return nombre;
	}
	/**
	 *  Metodo modificador
	 * @param nombre
	 */
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	/**
	 * Metodo accesor
	 * @return tipo
	 */
	public Tipo getTipo() {
		return tipo;
	}
	/**
	 *  Metodo modificador
	 * @param tipo
	 */
	public void setTipo(Tipo tipo) {
		this.tipo = tipo;
	}
	/**
	 * Metodo accesor
	 * @return precio
	 */
	public double getPrecio() {
		return precio;
	}
	/**
	 * Metodo modificador
	 * @param precio
	 */
	public void setPrecio(double precio) {
		this.precio = precio;
	}
	/**
	 * Metodo accesor
	 * @return cantidadExistencias;
	 */
	public int getCantidadExistencias() {
		return cantidadExistencias;
	}
	/**
	 * Metodo modificador
	 * @param cantidadExistencias
```

Programa 28 Clase Juguete

```
 */
public void setCantidadExistencias(int cantidadExistencias) {
	this.cantidadExistencias = cantidadExistencias;
}
/**
 * Sobreescribe el metodo equals
 */
public boolean equals(Object obj)
{
Juguete miJ;
boolean centinela=false;
if (this==obj)
{	centinela=true; }
else
	if(getClass()==obj.getClass())
	{  miJ=(Juguete)(obj);
     if(miJ.getId().equals(id))
	 { centinela=true;    }
  }
return centinela;
}


}
```

La enumeración Tipo, ver Programa 29, permite dejar explicitos los tipos que un juguete puede tener. En un momento dado, solo tendrá uno de ellos, es decir FEMENINO(0), MASCULINO (1) ó UNISEX(2).

Programa 29 Enumeracion Tipo

```
package co.uniquindio.jugueteria.model;

public enum Tipo {
	//Tipos de vehículos disponibles
	FEMENINO(0), MASCULINO (1), UNISEX(2);

	/**
	 * Atributo de la enumeraci[on
	 */
	private int numTipo;
/**
 * Metodo constructor
 * @param numTipo
 */
	private Tipo(int numTipo) {
		this.numTipo = numTipo;

	}
/**
 * Metodo accesor
 * @return
```

Programa 29 Enumeracion Tipo

```
 */
	public int getNumTipo() {
		return numTipo;
	}
/**
 * Representacion en string del valor elegido
 */
	public String toString()
	{
		String tipo=" ";
		switch(numTipo)
		{case 0: tipo="FEMENINO";
				break;
		case 1:tipo="MASCULINO";
			break;
		case 2: tipo="UNISEX";
		}
	return tipo;
	}
	}
```

Los siguientes 2 programas son usados por las clases de la lógica para el manejo de excepciones. La excepción SinExistenciasDelProductoException se lanza cuando se desea reducir la cantidad de existencias del producto, y al restar, la diferencia da negativa. NoExisteException se lanza cuando se busca un juguete por su código y no es encontrado.

Programa 30 SinExistenciasDelProductoException

```
package co.uniquindio.jugueteria.model;
/**
 * Excepcion para cuando no hay existencias
 * @author sonia
 * @author sergio
 */
public class SinExistenciasDelProductoException extends Exception {
/**
 * Constructor de la clase
 * @param mensaje El mensaje mostrado
 */
	public SinExistenciasDelProductoException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

Programa 31 NoExisteException

```
package co.uniquindio.jugueteria.model;

/**
 * Excepcion para cuando no hay existencias
 * @author sonia
 * @author sergio
 */
public class NoExisteException extends Exception {
/**
 * Constructor de la clase
 * @param mensaje
 */
public NoExisteException(String mensaje) {
            super(mensaje);
            // TODO Auto-generated constructor stub
      }
}
```

La clase Jugueteria, presente en Programa 32, es la clase principal del mundo del problema.

Programa 32 Jugueteria

```
package co.uniquindio.jugueteria.model;
/**
 * Clase para crear una jugueteria
 * @author Sonia
 *
 */
public class Jugueteria {
/**
 * Atributos de la clase
 */
private Juguete miJuguete0;
private Juguete miJuguete1;
private Juguete miJuguete2;


/**
 * Devuelve la cantidad de juguetes por tipo
 * @param tipo Es el tipo del juguete
 * @return la cantidad de juguetes
 */
public int contarPorTipo(Tipo tipo)
{
int contador=0;
if(miJuguete0.getTipo()==tipo)
{contador+=miJuguete0.getCantidadExistencias();
}
if(miJuguete1.getTipo()==tipo)
{contador+=miJuguete0.getCantidadExistencias();
}
```

Programa 32 Jugueteria

```
if(miJuguete2.getTipo()==tipo)
{contador+=miJuguete0.getCantidadExistencias();
}
 return contador;
}
/**
 * Obtiene la cantidad de existencias en la tienda
 * @return la cantidad de existencias
 */
public int obtenerTotalidadExistencias()
{
int
total=miJuguete0.getCantidadExistencias()+miJuguete1.getCantidadExistencias()+miJuguete2.
getCantidadExistencias();
return total;
}
/**
 * Obtiene el valor total de los juguetes que hay en la tienda
 * @return el valor total
 */
public double obtenerValorTotal()
{
double total=miJuguete0.getCantidadExistencias()*miJuguete0.getPrecio()
             +miJuguete0.getCantidadExistencias()*miJuguete1.getPrecio()+
             miJuguete2.getCantidadExistencias()*miJuguete2.getPrecio();
return total;
}
/**
 * Ordena de menor a mayor
 * @return un string con los tipos de menor a mayor
 */
public String ordenarTodosLosTipos()
{
Tipo tipoMasJuguetes, tipoMenosJuguetes, tipoIntermedio=Tipo.FEMENINO;
String salida="";
//Se obtiene la cantidad de juguetes por tipo
tipoMasJuguetes= hallarTipoConMasJuguetes();
tipoMenosJuguetes=hallarTipoConMenosJuguetes();
int
intermedio=Tipo.FEMENINO.getNumTipo()+Tipo.MASCULINO.getNumTipo()+Tipo.UNISEX.getNumTipo(
)-tipoMasJuguetes.getNumTipo()-tipoMenosJuguetes.getNumTipo();
//se procede a ordenar
if(intermedio==1)
{tipoIntermedio=Tipo.MASCULINO;}
else
      if(intermedio==2)
      {tipoIntermedio=Tipo.UNISEX;}
salida=" Ordenado de menor a mayor\n "+tipoMenosJuguetes.toString()+",
"+tipoIntermedio.toString()+", "+tipoMasJuguetes.toString();
return salida;
}
/**
 * Permite buscar un jugue con el id
 * @param id El id del juguete
```

Programa 32 Jugueteria

```
 * @return El juguete
 * @throws Exception
 */
public Juguete buscarJuguete(String id) throws NoExisteException
{
Juguete miJ=null;
if(miJuguete0.getId().equals(id))
{
miJ=miJuguete0;
}
else
if(miJuguete1.getId().equals(id))
{
miJ=miJuguete1;
}
else
if(miJuguete2.getId().equals(id))
{
miJ=miJuguete2;
}
if(miJ==null)
{
 throw new NoExisteException("El juguete no existe en la jugueteria");
}

return miJ;
}
/**
 * Permite agregar existencias al juguete
 * @param cantidad La cantidad de existencias
 * @param codigo El codigo dle juguete
 * @throws NoExisteException
 */
public void agregarExistenciasJuguete(int cantidad, String codigo) throws
NoExisteException
{
Juguete miJuguete= buscarJuguete( codigo) ;
miJuguete.aumentarExistencias(cantidad);
}
/**
 * Pemite disminuciones existencias al juguete
 * @param cantidad La cantidad de existencias
 * @param codigo El codigo dle juguete
 * @throws NoExisteException
 */
public void disminuirExistenciasJuguete(int cantidad, String codigo) throws
NoExisteException
{
Juguete miJuguete= buscarJuguete( codigo) ;
miJuguete.aumentarExistencias(cantidad);
}
public Tipo hallarTipoConMasJuguetes()
{  Tipo miTipo= Tipo.FEMENINO;
      int femenino, masculino, unisex, mayor;
```

Programa 32 Jugueteria

```
	femenino=contarPorTipo(Tipo.FEMENINO);
	masculino=contarPorTipo(Tipo.MASCULINO);
	unisex=contarPorTipo(Tipo.UNISEX);
	mayor=femenino;
	if(masculino>mayor)
	{
		mayor=masculino;
		 miTipo= Tipo.MASCULINO;
	}

	if(unisex>mayor)
	{
		mayor=unisex;
		 miTipo= Tipo.UNISEX;
	}
	return miTipo;
}
/**
 * Devuelve el tipo con menos juguetes
 * @return El tipo
 */
public Tipo hallarTipoConMenosJuguetes()
{  Tipo miTipo= Tipo.FEMENINO;
	int femenino, masculino, unisex, menor;
	femenino=contarPorTipo(Tipo.FEMENINO);
	menor=femenino;
	masculino=contarPorTipo(Tipo.MASCULINO);
	unisex=contarPorTipo(Tipo.UNISEX);
	if(masculino<menor)
	{
		menor=masculino;
		 miTipo= Tipo.MASCULINO;
	}

	if(unisex<menor)
	{
		menor=unisex;
		 miTipo= Tipo.UNISEX;
	}
	return miTipo;
}
public void setMiJuguete0(Juguete miJuguete0) {
	this.miJuguete0 = miJuguete0;
}
public void setMiJuguete1(Juguete miJuguete1) {
	this.miJuguete1 = miJuguete1;
}
public void setMiJuguete2(Juguete miJuguete2) {
	this.miJuguete2 = miJuguete2;
}
}
```

El Programa 33 tiene el código para general la ventana presente en la Figura 89.

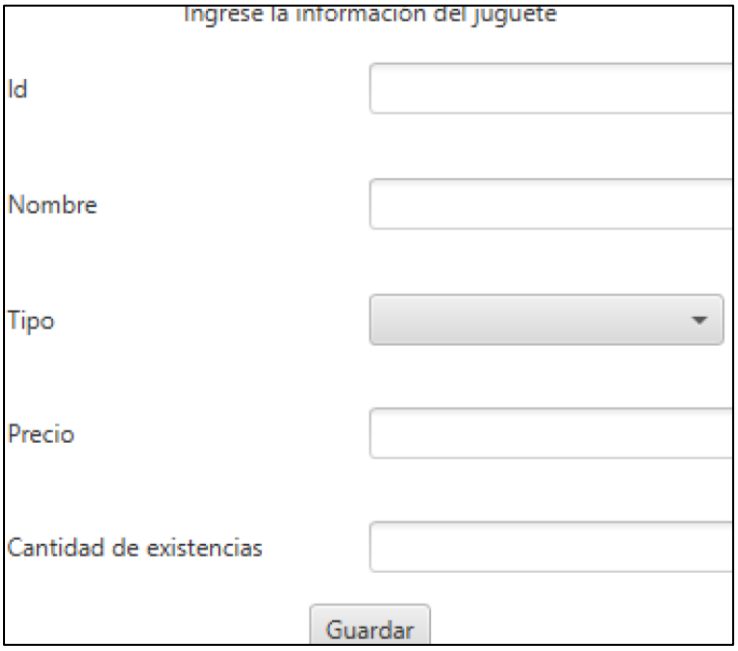


Figura 89 Agregar juguete

Programa 33 JugueteVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="320.0" prefWidth="364.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.jugueteria.view.ControladorJuguete">
   <bottom>
      <Button mnemonicParsing="false" onAction="#crearJuguete" text="Guardar"
BorderPane.alignment="CENTER" />
   </bottom>
   <center>
      <GridPane BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
```

Programa 33 JugueteVista.fxml

```
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Label text="  Id" />
            <Label text="  Nombre" GridPane.rowIndex="1" />
            <Label text="  Tipo" GridPane.rowIndex="2" />
            <Label text="  Precio" GridPane.rowIndex="3" />
            <Label text="  Cantidad de existencias" GridPane.rowIndex="4" />
            <TextField fx:id="idTextField" GridPane.columnIndex="1" />
            <TextField fx:id="nombreTextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
            <TextField fx:id="precioTextField" GridPane.columnIndex="1"
GridPane.rowIndex="3" />
            <TextField fx:id="cantidadTextField" GridPane.columnIndex="1"
GridPane.rowIndex="4" />
            <ComboBox fx:id="tipoComboBox" prefHeight="25.0" prefWidth="172.0"
GridPane.columnIndex="1" GridPane.rowIndex="2" />
         </children>
      </GridPane>
   </center>
   <top>
      <Label text="Ingrese la información del juguete" BorderPane.alignment="CENTER" />
   </top>
</BorderPane>
```

El Programa 34 contiene el código para la generar la ventana presente en Figura *90*.

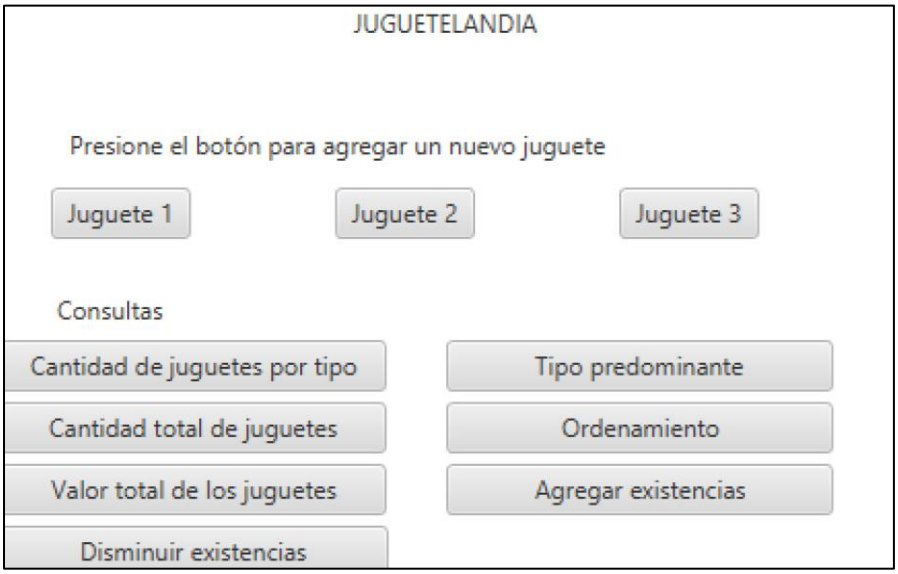


Figura 90 Menú de la aplicación

Programa 34 JugueteriaVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="270.0" prefWidth="434.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.jugueteria.view.ControladorJugueteria">
   <top>
      <Label text="JUGUETELANDIA" BorderPane.alignment="CENTER" />
   </top>
   <center>
      <AnchorPane prefHeight="201.0" prefWidth="413.0" BorderPane.alignment="CENTER">
         <children>
            <Pane layoutX="22.0" layoutY="14.0" prefHeight="97.0" prefWidth="370.0"
AnchorPane.bottomAnchor="31.0" AnchorPane.leftAnchor="22.0" AnchorPane.rightAnchor="42.0"
AnchorPane.topAnchor="14.0">
               <children>
                  <HBox layoutX="3.0" layoutY="57.0" spacing="70.0">
                     <children>
                        <Button layoutX="50.0" layoutY="52.0" mnemonicParsing="false"
onAction="#crearJuguete0" text="Juguete 1" />
                        <Button layoutX="153.0" layoutY="52.0" mnemonicParsing="false"
onAction="#crearJuguete1" text="Juguete 2" />
                        <Button layoutX="262.0" layoutY="52.0" mnemonicParsing="false"
onAction="#crearJuguete2" text="Juguete 3" />
                     </children>
                  </HBox>
               </children>
            </Pane>
            <Label layoutX="34.0" layoutY="42.0" prefHeight="17.0" prefWidth="307.0"
text="Presione el botón para agregar un nuevo juguete" AnchorPane.leftAnchor="34.0"
AnchorPane.topAnchor="42.0" />
            <Label layoutX="28.0" layoutY="119.0" prefHeight="23.0" prefWidth="179.0"
text="Consultas" />
         </children>
      </AnchorPane>
   </center>
   <bottom>
      <GridPane BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
```

Programa 34 JugueteriaVista.fxml

```
            <Button mnemonicParsing="false" onAction="#contarPorTipo" prefHeight="22.0"
prefWidth="188.0" text="Cantidad de juguetes por tipo" />
            <Button mnemonicParsing="false" onAction="#obtenerTotalidadExistencias"
prefHeight="22.0" prefWidth="188.0" text="Cantidad total de juguetes"
GridPane.rowIndex="1" />
            <Button mnemonicParsing="false" onAction="#obtenerValorTotal"
prefHeight="22.0" prefWidth="188.0" text="Valor total de los juguetes"
GridPane.rowIndex="2" />
            <Button mnemonicParsing="false" onAction="#hallarTipoConMasJuguetes"
prefHeight="22.0" prefWidth="188.0" text="Tipo predominante" GridPane.columnIndex="1" />
            <Button mnemonicParsing="false" onAction="#ordenarTodosLosTipos"
prefHeight="22.0" prefWidth="188.0" text="Ordenamiento" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
            <Button mnemonicParsing="false" onAction="#aumentarExistenciasJuguete"
prefHeight="25.0" prefWidth="188.0" text="Agregar existencias" GridPane.columnIndex="1"
GridPane.rowIndex="2" />
            <Button mnemonicParsing="false" onAction="#disminuirExistenciasJuguete"
prefHeight="25.0" prefWidth="189.0" text="Disminuir existencias" GridPane.rowIndex="3" />
         </children>
      </GridPane>
   </bottom>
</BorderPane>
```

La Figura 91 muestra la interfaz que se genera con el código presentado en el Programa 35 .

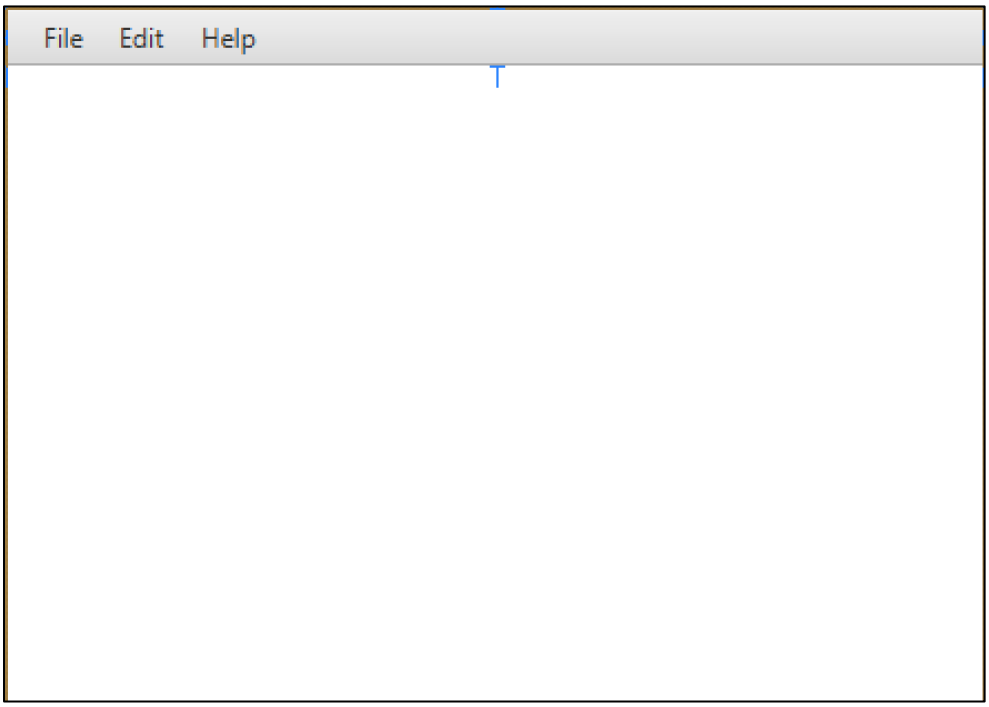


Figura 91 Layout raiz

Programa 35 LayoutRaiz.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="313.0" prefWidth="439.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1">
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem mnemonicParsing="false" text="Close" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Edit">
            <items>
              <MenuItem mnemonicParsing="false" text="Delete" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Help">
            <items>
              <MenuItem mnemonicParsing="false" text="About" />
            </items>
          </Menu>
        </menus>
      </MenuBar>
   </top>
</BorderPane>
```

Actividad 7. Actividad Estructuras de decisión

Construya aplicaciones completas para los siguientes enunciados

1. Crear una aplicación que informe el sitio que visitará un turista, si se sabe el mes en el que desea viajar, la edad del turista, la cantidad de días a permanecer en el territorio, y su presupuesto. El sitio asignado para el viaje depende de las estaciones, de la disponibilidad para trabajar y de otras condiciones que se presentan a continuación:
-Si el mes en el que el turista desea viajar corresponde a Invierno, hay dos opciones de viaje: Turkia o Gran Bretaña . Si su presupuesto es superior a 20 millones, el turista viajará a Gran Bretaña. Cuando el turista sea menor de edad, sin importar su presupuesto, viajará a Turkia.
- Cuando el mes corresponda a Verano, el turista, que debe ser mayor de edad, viajará a Dubai, siempre y cuando permanezca más de 10 días en el territorio y pueda trabajar como mínimo 5 días en Uber. De lo contrario irá a Cancún.
- Si el mes de viaje corresponde a Otoño, el turista deberá tener edad entre 40 y 65 años, y un presupuesto de 10 millones, viajará a Chile.

- Si el mes es Primavera, y el turista no puede trabajar, pero su presupuesto es superior a 22 millones, irá a Japón.
- Si el turista no cumple con ninguna condición previa, deberá viajar a Perú.

Informar el sitio que debe visitar el turista.
Tabla de equivalencia Mes - estación.
12, 1, 2 Invierno 3, 4, 5 Primavera 6, 7, 8 Verano 9, 10, 11 Otoño

2. Se desea crear una aplicación para informar el obsequio que recibirá una mujer en una tienda de ropa. Las condiciones para la entrega del obsequio son las siguientes:

- Si tiene tarjeta Platinum, un acumulado de puntos superior a 20000, sus compras anuales son inferiores a 4.5 millones y tiene la tarjeta desde hace 2 o más años, se entregará un bolso.
- Si su tarjeta es platinum, el acumulado de puntos es inferior o igual a 20000 recibirá una cosmetiquera.
- Si su tarjeta de puntos es Gold, posee la tarjeta desde hace 2 o más años, tiene edad superior a 45 años y acumulado de compras anuales superior a 10 millones, se le dará una pulsera.
- Si su tarjeta es Gold, su edad está entre 18 y 45 años y cumple años en el mes actual, se le regalará un Brazalete.
- Si tiene alguna de las tarjetas, y no cumplió ninguna de las condiciones anteriores, y su promedio de compras anuales es de 6 millones de pesos se le regalará una cosmetiquera.
- Si no tiene tarjeta pero su promedio de compras excede los 15 millones, se le regalará iphone 6s.
- Si no cumple ninguna de las condiciones anteriores no recibirá obsequio.

3. Estructuras contenedoras

Objetivos Pedagógicos

Al final de este nivel el lector estará en capacidad de:

- Utilizar estructuras contenedoras de tamaño fijo y variable para resolver problemas en los cuales es necesario almacenar una secuencia de elementos
- Utilizar ciclos para poder manipular las estructuras contenedoras fijas
- Construir interfaces gráficas que involucren la utilización de estructuras contenedoras

3. ESTRUCTURAS CONTENEDORAS DE TAMAÑO FIJO Y VARIABLE

3.1. Estructuras contenedoras de tamaño fijo

Los arreglos son un tipo de estructura contenedora que permite almacenar elementos del mismo tipo. Para poder hacer uso de este tipo de estructura, el arreglo primero debe declararse y luego se le debe reservar memoria (haciendo uso del operador new). Esto se hace indicando el tipo de dato y el nombre.

```
int [] misEdades;
```

Tipo de dato — nombre

para reservar memoria se debe utilizar el operador new tal como se muestra a continuación:

```
misEdades = new int[10];
```

Es posible efectuar a declaración y creación en una sola línea:

```
int   misEdades [] = new int [10];
```

ó

```
int[]misEdades  = new int [10];
```

Un arreglo que contiene tipos de datos primitivos puede inicializarse al momento de su declaración:

```
int[]misEdades  ={2,3,4,5};
```

Tipo de dato puede ser un tipo de dato primitivo o tipo clase, es decir, cualquier clase creada por el programador. Ejemplo:
Empleado[] miEmpleado;
miEmpleado= new Empleado[5];

Los arreglos pueden tener más de una dimensión, cuando su dimensión es 2 se les conoce como matrices.

Para poder desplazarse y obtener los elementos almacenados en dicha estructura, se requiere hacer uso de ciclos. Cada uno de los elementos contenidos en la estructura se designan haciendo uso de un subíndice, que inicia en cero. Por ejemplo si se tiene el arreglo misEdades, que contiene 4 elementos, el primer elemento corresponderá a misEdades [0], el segundo a misEdades [1], el tercero a misEdades [2] y finalmente el cuarto será misEdades [3]. Esto permite concluir que para poder tener acceso a un elemento, se debe especificar el nombre e índice respectivo.

En Java, cuando se declara un arreglo de un tipo de dato primitivo numérico, este automáticamente se inicializa con ceros. Si es boolean, se inicializa con false. Debe tener precaución porque esto no ocurre en lenguajes como C++ [2][3][4]. En tal caso será necesario hacer uso de una estructura repetitiva para fijar un valor por defecto, como se muestra a continuación:

```
int   misEdades [] = new int [10];
for(int i=0; i<misEdades.length; i++)
{
   misEdades[i]=0
}
```

Si el arreglo es de Strings automáticamente se inicializa con null.

3.2 Pila y Stack

En Java se diferenciar dos áreas, Heap **(Montículo)** y Stack (**Pila**). La primera de ellas almacena información global (instancias de las variables) y la segunda, almacena información de variables locales y retornos. La Figura 3 ilustra el comportamiento de las estructuras contenedoras en las diferentes áreas de memoria[7].

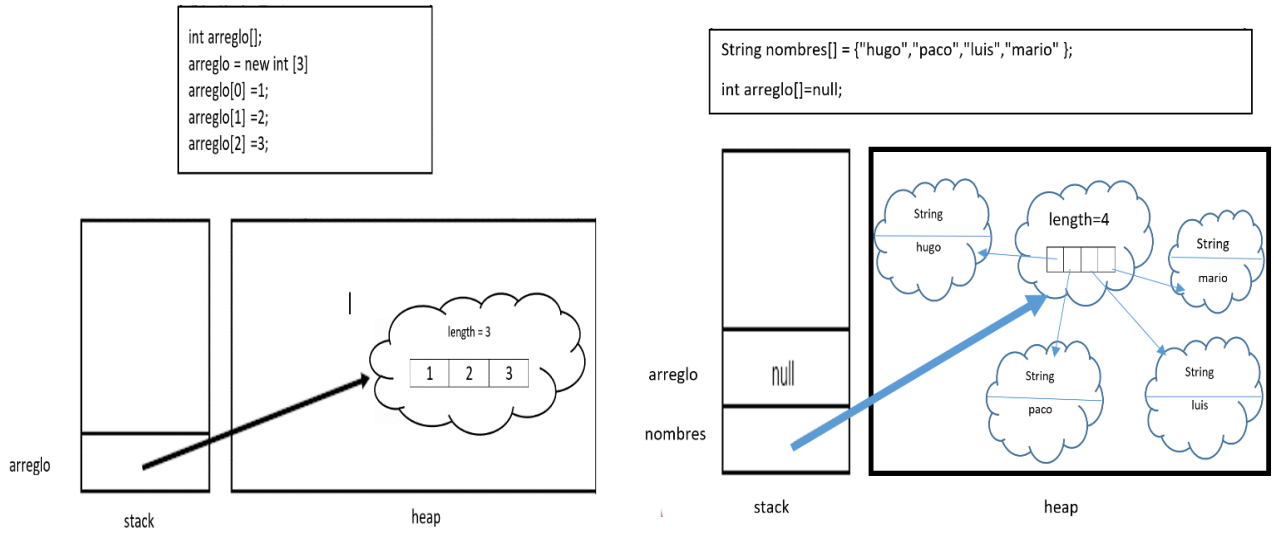


Figura 92 Stack y heap

3.3 Patrones para instrucciones repetitivas

Un patrón puede verse como una solución genérica para un problema. Para trabajar con estructuras contenedoras se se plantean 3 diferentes patrones[8]:

[7] Basado en los ejemplos creados por el Julián Esteban Gutierrez, también docente de la Universidad del Quindío, presentes en su material de clase Introducción a la Programación en Java2.

[8] Estos patrones están descritos en el libro Fundamentos de Programación. Aprendizaje activo Basado en Casos de Jorge Villalobos y Rubby Casallas

a) Patrón de recorrido Total

Un ejemplo de este tipo de patrón es el siguiente:

```
public double calcularSalarioPromedio ()
{
  double promedioSueldo = 0.0;

  for( int i = 0; i < misEmpleados.length ; i++)
  {
     promedioSueldo += misEmpleados[ i ].getSueldo();
  }
}
```

La variable length, devuelve la longitud del vector. Observe que el índice va desde 0 hasta misEmpleados.**length-1.** Si el índice se encuentra por fuera de este rango se genera una excepción.

b) Patrón de Recorrido Parcial

El recorrido se interrumpe cuando se cumple una condición. Un ejemplo de este tipo de patron se presenta a continuación, en el que el ciclo deja de ejecutarse cuando se encuentra un salario superior a 5000000.

```
public int encontrarSalario()
{
  boolean centinela = false;
  int pos;
//Se devuelve la posición en donde se encuentra a la primera persona con salario> 5000000
  for( int i = 0; i < misEmpleados.length && centinela != true ; i++)
  {
    if(misEmpleados [ i ] .getSueldo() > 5000000)
    {
      centinela = true;
      pos=i;
    }
  }
return pos;
}
```

Se sugiere al lector evitar el uso de sentencias tales como break **y** continue para construir un recorrido parcial**.** Es decir, sentencias que rompan el flujo secuencial de ejecución de la aplicación. La sentencia *break*, finaliza la ejecución del ciclo y salta a la primera línea del programa que se debe ejecutar tras el ciclo. Por su parte, *continue* finaliza la iteración actual y salta a la siguiente iteración de la estructura repetiva sin salir de ella [1]. En [2][3][4] se desaconseja el uso de ambas, dado que existe la posibilidad de que el programador primero plantee el ciclo y luego verifique rústicamente su funcionamiento mediante ensayo y error. La invitación a no usar estas instrucciones

sugiere entonces que es mejor pensar la condición o invariante para que el ciclo se ejecute (y no al contrario), y a continuación crear el cuerpo del ciclo. Esta técnica hace mucho más legible el programa y facilita su posterior verificación.

c) Patrón de doble recorrido

Con este patrón por cada uno de los elementos del arreglo se debe realizar un recorrido completo.

```
public int determinarMayorFrecuencia()
{
   int edad, contador = 0, mayorFrecuencia = 0, guardaContador = 0;
   for( int i = 0 ; i < misPersonas.length ; i++ )
   {
     edad     = misPersonas[ i ];
     contador = 0;
     for( int j = 0 ; j < misPersonas.length; j++ )
     {
       if( i != j )
       {
         if( edad == misPersonas [j].getEdad())
         {
           contador++;
         }
       }
      }

     if( contador > guardaContador )
     {
        guardaContador  = contador;
        mayorFrecuencia = i;
     }
  }
   return misPersonas [ mayorFrecuencia ].getEdad();
}
```

3.4 Clase Arrays

Esta clase implementada en Java, facilita el trabajo con estructuras contenedoras. Algunos de sus métodos se presentan a continuación.

Método	Descripción
static boolean **equals** (int[ ] a,int[ ] a2)	Devuelve true si los arreglos tienen los mismos elementos, en el mismo orden if(Arrays.equals(miArreglo1, miArreglo2))
static int[ ] **copyOf** (int[ ] original, int newLength);	Copia el arreglo original (trunca o rellena para que el resultado quede con el tamaño requerido). resultado = Arrays.copyOf(miArreglo1, 4);
static int[ ] **copyOfRange**(int[] original, int from, int to)	Copia el arreglo original, desde la posición inicial hasta la final
static void **fill** (int[ ] a, int val) static void **fill** (int[ ] a, int inicio, int final, int val)	Rellena con val cada elemento del arreglo de enteros. Arrays.fill(miArreglo1, 5); La segunda opción permite especificar el rango de posiciones en la que se va a efectuar el reemplazo
static String **toString**(int[] a)	Retorna la representación en String del contenido del arreglo

Tabla 17 Métodos de la clase Arrays [2][3][4]

Otro método importante que permite agilizar el trabajo con arreglos es el método arraycopy:

System.arraycopy(src, srcPos, dest, destPos, length);

Src es el arreglo original, srcPos es la posición desde donde se inicia a copiar, normalmente es cero.

Dest es el arreglo en donde se copia

destPos, posicion a partir de la cual se copia en el arregloo destino

length, posicion final hasta donde se copia.

Ejemplo:

int p1[] = {2,3,1,4,8};

int p2[] = {7,7,3,1,10};

System.arraycopy(p1, 0, a, 0, p1.length)

Para poder manipular los elementos contenidos en una estructura contenedora es necesario hacer uso de estructuras repetitivas. Es importante tener en cuenta que los elementos de un arreglo se designan con un subíndice que comienza en cero. Por ejemplo si se tiene el arreglo `miEmpleado` el cual contiene 4 elementos, el primer elemento corresponderá a `miEmpleado[0]`, el segundo a `miEmpleado[1]`, el tercero a `miEmpleado[2]` y finalmente el cuarto será `miEmpleado [3]`. Puede verse entonces, que para tener acceso a un elemento es necesario indicar su nombre y subíndice.

Ejemplo:

```
public double calcularPromedioSalario()
  {
     double promedioSueldo = 0;

     int i = 0;
     //estructuras repetitiva
     while( i < 4 )
     {
       promedioSueldo += miEmpleado[ i ].getSalario();//i toma valores entre 0 y 3
       //se suma el salario del empleado 0, del 1, del 2 y del empleado 3
       i++;
     }
   promedioSueldo= promedioSueldo/4;
  }
```

El **`for`**, el **`while`** y el **`do-while`** son estructuras repetitivas.
Todo bucle o ciclo consta de tres partes básicas, a saber:

- **Decisión**: aquí se evalúa la condición y, en caso de ser cierta, se ejecuta el ciclo.
- **Cuerpo del bucle**: son las instrucciones que se desea ejecutar repetidamente un cierto número de veces.
- **Salida del bucle**: condición que indica cuando termina de ejecutarse el ciclo, es decir ya no se harán más repeticiones.

Una forma de controlar un bucle es mediante una variable llamada **CONTADOR**, cuyo valor se incrementa o decrementa en una cantidad constante en cada repetición que se produzca.

También, en los bucles suele haber otro tipo de variables llamadas **ACUMULADOR**, cuya misión es almacenar una cantidad variable resultante de operaciones sucesivas y repetidas. Es como un contador, con la diferencia que el incremento/decremento es variable.

3.5 for

El ciclo for la primera vez ejecuta la parte de inicialización, la cual normalmente contiene una expresión de asignación dirigida hacia una variable de control (actúa como contador) del bucle. La expresión de inicialización se ejecuta únicamente una vez.

A continuación se examina la condición, que debe ser una expresión booleana. Normalmente se compara la variable de control con un valor específico. Si la comparación da verdadero, entonces se ejecuta el cuerpo del ciclo. De lo contrario, el bucle termina.

Es importante anotar que a continuación se ejecuta la parte de iteración, que suele ser una expresión que incrementa o decrementa la variable de control del ciclo. No se debe olvidar que en cada pasada del ciclo, se evalua nuevamente la expresión condicional, se ejecuta el cuerpo del bucle y a continuación la iteración. Esto se repetirá hasta que la expresión de control sea falsa [2][3][4].

```
for ( inicialización; condicion;  iteración)
{
  acción 1
  ........
  acción n
}
```

3.6 while

En este tipo de estructura, el cuerpo del bucle se repite MIENTRAS se cumple una determinada condición, que se especifica entre paréntesis.

La variable que cuenta las veces que el ciclo se ha ejecutado se llama contador.

Luego se llega al fin del while, lo cual significa que se han terminado las instrucciones del cuerpo del bucle: se debe, por lo tanto volver a evaluar la condición que tenía el "while" entre paréntesis para ver qué sucede ahora [2][3][4].

```
while( condición )
{
  acción 1
  ........
  acción n
}
```

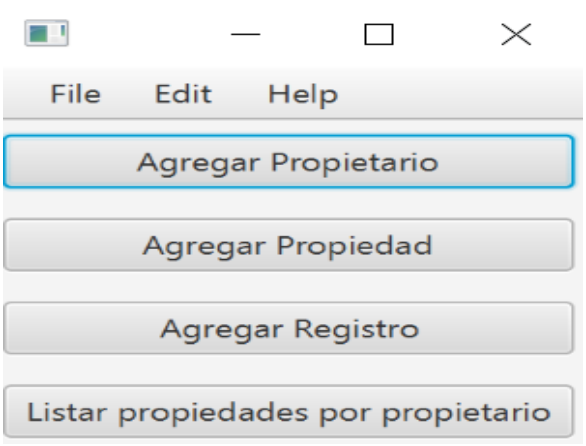
3.7 do-while

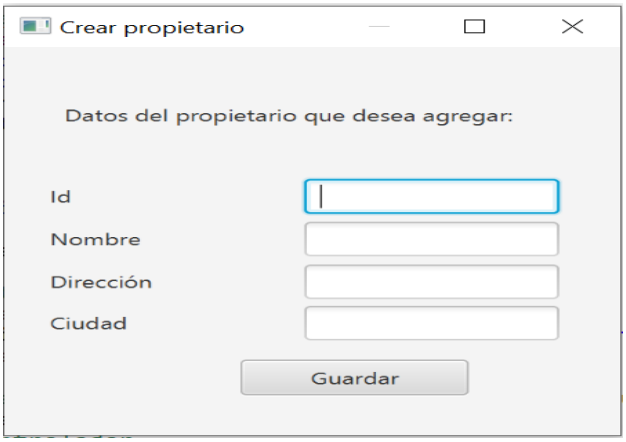
Con este tipo de estructura se logra que un bucle se ejecute AL MENOS UNA VEZ antes de comprobar la condición de repetición [2][3][4].

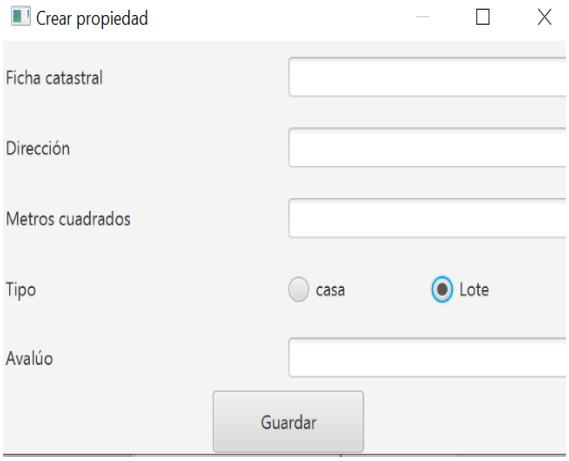
La estructura del **`do ... while`**, genéricamente, es:

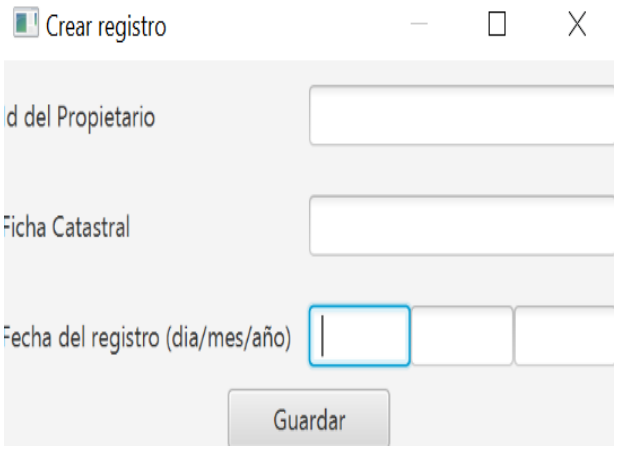
```
do
{
  acción 1
  ........
  acción n
} while( condición );
```

3.8 Caso de estudio 1Unidad III: Oficina de registro









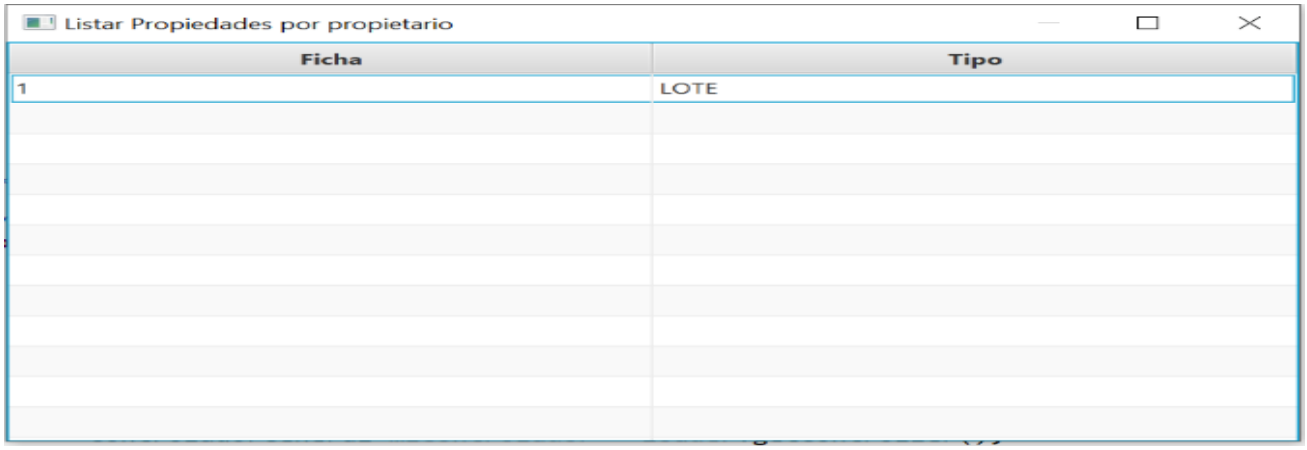


Figura 93 Interfaz Aplicación Registro

3.8.1 Comprensión del problema

a) Requisitos funcionales

NOMBRE	R1- Agregar propietario
RESUMEN	Permite agregar un nuevo propietario a la oficina de registro
ENTRADAS	
Id, nombre, dirección y ciudad	
RESULTADOS	
Un nuevo propietario se ha agregado a la oficina de registro	

NOMBRE	R2- Agregar una nueva propiedad
RESUMEN	Permite agregar una nueva propiedad a la oficina de registro
ENTRADAS	
Ficha catastral, dirección, metros cuadrados, tipo y avaluo	
RESULTADOS	
Una propiedad ha sido agregada a la oficina de registro	

NOMBRE	R3- Crear un registro
RESUMEN	Permite crear un nuevo registro
ENTRADAS	
Id del propietario, ficha catastral, Fecha del registro	
RESULTADOS	
Un nuevo registro ha sido creado	

NOMBRE	R4- Listar propiedades por propietario
RESUMEN	Devuelve un listado con todas las propiedades que tiene el propietario
ENTRADAS	
Id del propietario	
RESULTADOS	
Un listado con todas las propiedades que posee el propietario	

b) El modelo del mundo del problema

Las actividades que se deben realizar para construir el modelo del mundo son:

Identificar las entidades o clases.

ENTIDAD DEL MUNDO	DESCRIPCIÓN
OficinaDeRegistro	Es la entidad más importante del mundo del problema.
Propiedad	Es un atributo de la oficina de registro
Propietario	Es un atributo de la oficina de registro
Fecha	Es un atributo del Registro
Registro	Se compone por una fecha, un propietario y un propietario

El diagrama de clases correspondiente para este caso de estudio es el siguiente:

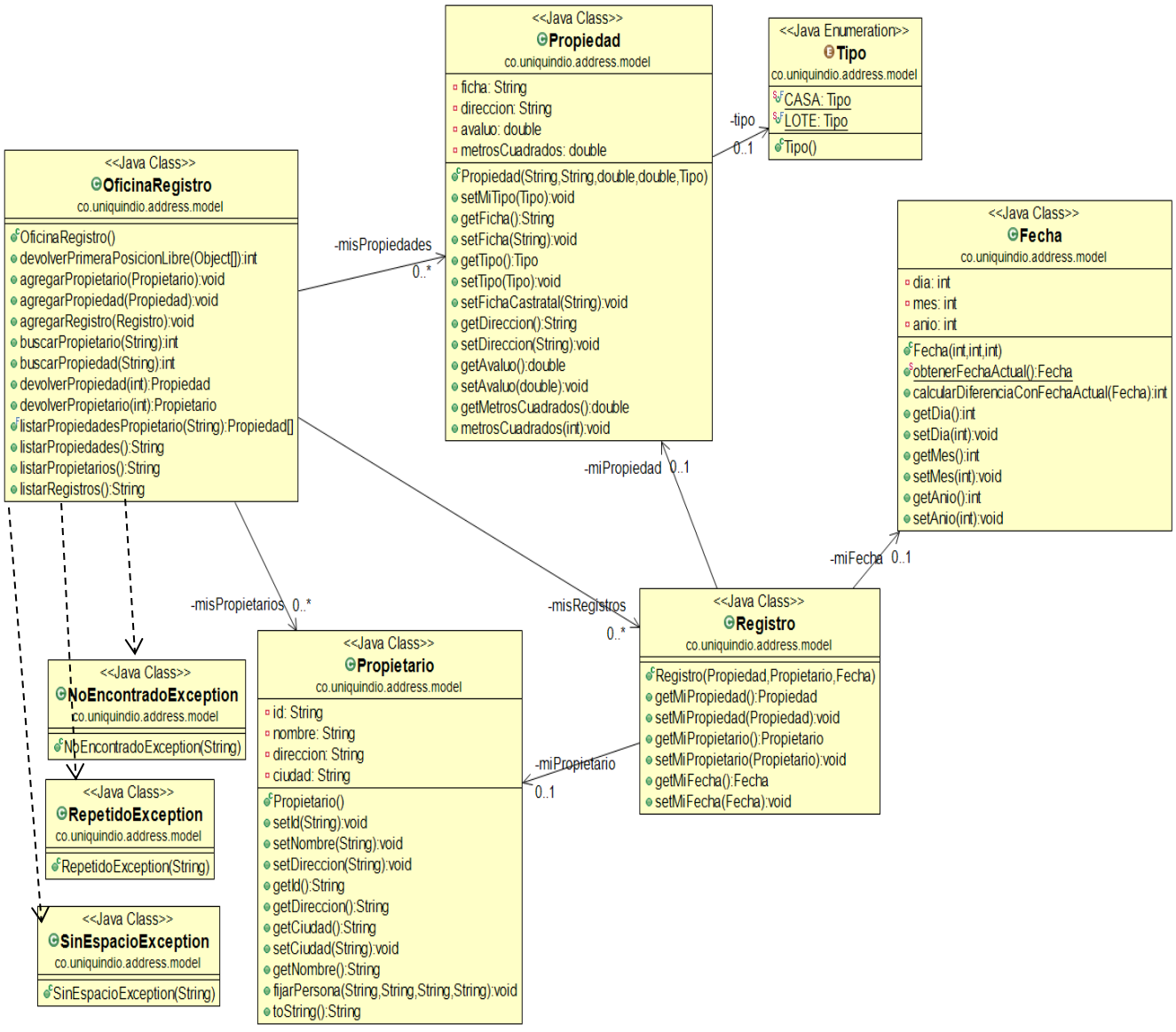


Figura 94 Diagrama de clases para el caso de estudio de la oficina de registro

La primera clase que se crea es la clase Fecha, ver Programa 36. En esta clase se declaran 3 atributos: dia, mes y año. Adémas se implementan métodos tales comos obtenerFechaActual y calcularDiferenciaConFechaActual. Para obtener la fecha actual se hace uso de la clase calendario gregoriano.

Programa 36 Fecha

```
package co.uniquindio.address.model;

import java.util.Calendar;
import java.util.GregorianCalendar;
/**
 * Clase para representar una fecha
 * @author sonia
 * @author sergio
 */
public class Fecha {
/**
 * Atributos de la clase
 */
private int dia;
private int mes;
private int anio;
/**
 * Constructor de la clase
 * @param dia Es el dia
 * @param mes Es el mes
 * @param anio Es el anio
 */
public Fecha(int dia, int mes, int anio) {
	super();
	this.dia = dia;
	this.mes = mes;
	this.anio = anio;
}
/**
 * Metodo para obtener la fecha actual
 * @return la fecha
 */
public static Fecha   obtenerFechaActual( )
{   Fecha actual;
    GregorianCalendar diaHoy = new GregorianCalendar( );
    int anio = diaHoy.get( Calendar.YEAR );
    int mes = diaHoy.get( Calendar.MONTH ) + 1;
    int dia = diaHoy.get( Calendar.DAY_OF_MONTH );
    actual=new Fecha(dia, mes,anio);
    return actual;
}


/**
 * Devuelve la diferencia en dias
 * @param miFecha La fecha a restar
 * @return los dias
 */
```

Programa 36 Fecha

```
public int calcularDiferenciaConFechaActual( Fecha miFecha )
{
int diferenciaEnMeses = 0;
diferenciaEnMeses  =  ( miFecha.getMes() - mes )+12 * ( miFecha.getAnio() - anio ) ;
//En caso de que el día no sea mayor se debe restar un mes. Es decir, no //se tiene la
cantidad de días suficientes para poder sumar un mes más.
if(dia >= miFecha.getDia()  )
      diferenciaEnMeses --;
return diferenciaEnMeses/12 ;
}
/**
 * Metodo accesor
 * @return dia
 */
public int getDia() {
      return dia;
}
/**
 * Metodo modificador
 * @param dia
 */
public void setDia(int dia) {
      this.dia = dia;
}
/**
 * Metodo accesor
 * @return mes
 */
public int getMes() {
      return mes;
}
/**
 * Metodo modificador
 * @param mes
 */
public void setMes(int mes) {
      this.mes = mes;
}
/**
 * Metodo accesor
 * @return anio
 */
public int getAnio() {
      return anio;
}
/**
 * Metodo modificador
 * @param anio
 */
public void setAnio(int anio) {
      this.anio = anio;
}

}
```

Programa 36 Fecha

La enumeracion Tipo, que se presenta en Programa 37, permite especificar el tipo de la propiedad, que puede ser casa o lote.

Programa 37 Tipo

```
package co.uniquindio.address.model;
/**
 * Enumeracion para fijar el tipo de la propiedad
 * @author sonia
 *
 */
public enum Tipo {
//posibles valores
	CASA, LOTE;

}
```

La clase Propiedad, implementada en Programa 38, tiene los métodos get y set para cada uno de los atributos declarados, a saber: String ficha, String dirección, double avaluo, double metrosCuadrados y Tipo tipo. Además tiene un método constructor para inicializar los atributos de la clase. El constructor tiene los siguientes parámetros: String fichaCastratal, String direccion, double avaluo, double metrosCuadrados, Tipo miTipo.

Programa 38 Propiedad

```
package co.uniquindio.address.model;

/**
 * clase para representar una propidad
 *
 * @author sonia
 * @author sergio
 */
public class Propiedad {
	/**
	 * Atributos de la propiedad
	 */
	private String ficha;
	private String direccion;
	private double avaluo;
	private double metrosCuadrados;
	private Tipo tipo;

	/**
	 * Constructor de la clase
	 *
	 * @param fichaCastratal  El id de la propiedad
	 * @param direccion       Es la direccion del inmueble
	 * @param avaluo          El avaluo
	 * @param metrosCuadrados Los metros cuadrados
```

Programa 38 Propiedad

```
         * @param miTipo          Es una enumeracion
         */
public Propiedad(String fichaCastratal, String direccion, double avaluo, double
metrosCuadrados, Tipo miTipo) {
              super();
              this.ficha = fichaCastratal;
              this.direccion = direccion;
              this.avaluo = avaluo;
              this.metrosCuadrados = metrosCuadrados;
              this.tipo = miTipo;
       }

       /**
        * Metodo modificador
        *
        * @param miTipo
        */
       public void setMiTipo(Tipo miTipo) {
              this.tipo = miTipo;
       }

       /**
        * Metodo accesor
        *
        * @return ficha
        */
       public String getFicha() {
              return ficha;
       }

       /**
        * Metodo modificador
        *
        * @param ficha
        */
       public void setFicha(String ficha) {
              this.ficha = ficha;
       }

       /**
        * Metodo accesor
        *
        * @return tipo
        */
       public Tipo getTipo() {
              return tipo;
       }

       /**
        * Metodo modificador
        *
        * @param tipo
        */
       public void setTipo(Tipo tipo) {
```

Programa 38 Propiedad

```
		this.tipo = tipo;
	}

	/**
	 * Metodo modificador
	 *
	 * @param fichaCastratal
	 */
	public void setFichaCastratal(String fichaCastratal) {
		this.ficha = fichaCastratal;
	}

	/**
	 * Metodo accesor
	 *
	 * @return direccion;
	 */
	public String getDireccion() {
		return direccion;
	}

	/**
	 * Metodo modificador
	 *
	 * @param direccion
	 */
	public void setDireccion(String direccion) {
		this.direccion = direccion;
	}

	/**
	 * Metodo accesor
	 *
	 * @return avaluo
	 */
	public double getAvaluo() {
		return avaluo;
	}

	/**
	 * Metodo modificador
	 *
	 * @param avaluo
	 */
	public void setAvaluo(double avaluo) {
		this.avaluo = avaluo;
	}

	/**
	 * Metodo accesor
	 *
	 * @return metrosCuadrados
	 */
	public double getMetrosCuadrados() {
```

Programa 38 Propiedad

```
		return metrosCuadrados;
	}

	/**
	 * Metodo modificador
	 *
	 * @param avaluo
	 */
	public void metrosCuadrados(int metrosCuadrados) {
		this.metrosCuadrados = metrosCuadrados;
	}

}
```

La clase Propietario, ver Programa 39, permite crear un nuevo propietario. Es una clase sencilla, que contiene métodos get y set para cada uno de los atributos. El método fijarPersona(String id, String nombre, String direccion, String ciudad) facilita el trabajo de inicializar los atributos de la clase. El método toString(), devuelve la representación en String del objeto.

Programa 39 Propietario

```
package co.uniquindio.address.model;

/**
 * Clase para crear un empleado
 *
 * @author Sonia
 *
 */
public class Propietario {
	/**
	 * Es el id del propietario
	 */
	private String id;
	/**
	 * Es el nombre del propietario
	 */
	private String nombre;
	/**
	 * Es la direccion del propietario
	 */
	private String direccion;
	/**
	 * Es la ciudad de residencia del propietario del propietario
	 */
	private String ciudad;

	/**
	 * Metodo constructor, en este caso puede omitirse su construccion
	 */
	public Propietario() {
	}
```

```java
	/**
	 * Metodo modificador
	 *
	 * @param id, es el id del empleado
	 */
	public void setId(String id) {
		this.id = id;
	}

	/**
	 * Metodo modificador
	 *
	 * @param nombre, Es el nombre del empleado
	 */
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	/**
	 * Metodo modificador
	 *
	 * @param direccion, la direccion del empleado
	 */
	public void setDireccion(String direccion) {
		this.direccion = direccion;
	}

	public String getId() {
		return id;
	}

	public String getDireccion() {
		return direccion;
	}

	public String getCiudad() {
		return ciudad;
	}

	/**
	 * Metodo modificador
	 *
	 * @param ciudad, la ciudad de residencia
	 */
	public void setCiudad(String ciudad) {
		this.ciudad = ciudad;
	}

	/**
	 * Metodo accesor
	 *
	 * @return el nombre del propietario
	 */
	public String getNombre() {
		return nombre;
```

```
	}

	/**
	 * Es metodo inicia todo el objeto existente con los valores recibidos
	 *
	 * @param id        Es el id del empleado
	 * @param nombre    Es el nombre del empleado
	 * @param direccion Es la direccion del empleado
	 * @param ciudad    Es la ciudad del empleado
	 */
	public void fijarPersona(String id, String nombre, String direccion, String ciudad)
{
		this.id = id;
		this.nombre = nombre;
		this.direccion = direccion;
		this.ciudad = ciudad;
	}

	/**
	 * Devuelve la representacion en String del empleado
	 */
	public String toString() {
		return id + "  " + nombre + "    " + direccion + "    " + ciudad;
	}

}
```

Las clases SinEspacioException, RepetidoException y NoEncontradoException permiten efectuar todo el tratamiento de excepciones durante la ejecución del programa.

Programa 40 SinEspacioException

```
package co.uniquindio.address.model;

/**
 * Clase para manejar la excepcion de sin espacio
 *
 * @author sonia
 * @author sergio
 */
public class SinEspacioException extends Exception {
	/**
	 * constructor de la clase
	 *
	 * @param mensaje El mensaje a presentar
	 */
	public SinEspacioException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

Programa 41 RepetidoException

```
package co.uniquindio.address.model;

/**
 * Clase para manejar la excepcion de elemento repetido
 *
 * @author sonia
 * @author sergio
 */
public class RepetidoException extends Exception {
	/**
	 * constructor de la clase
	 *
	 * @param mensaje El mensaje a presentar
	 */
	public RepetidoException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

Programa 42 NoEncontradoException

```
package co.uniquindio.address.model;
/**
 * Clase para manejar la excepcion de  elemento no encontrado
 * @author sonia
 * @author sergio
 */
public class NoEncontradoException extends Exception {
/**
* constructor de la clase
* @param mensaje El mensaje a presentar
*/
	public NoEncontradoException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

El Programa 43 presenta el código referente a la creación de un registro. El registro tiene 3 atributos: Propiedad miPropiedad, Propietario miPropietario y Fecha miFecha. El constructor de Registro recibe 3 parámetros, que permiten inicializar los atributos de la clase. La signatura de dicho método es: public Registro(Propiedad miPropiedad, Propietario miPropietario, Fecha miFecha)

Programa 43 Registro

```
package co.uniquindio.address.model;

/**
 * Clase para representar un registro
 *
 * @author sonia
 * @author sergio
 */
public class Registro {
	/**
	 * Atributos de la clase
	 */
	private Propiedad miPropiedad;
	private Propietario miPropietario;
	private Fecha miFecha;

	/**
	 * Constructor de la clase
	 *
	 * @param miPropiedad   La propiedad
	 * @param miPropietario El propietario
	 * @param miFecha       La fecha
	 */
	public Registro(Propiedad miPropiedad, Propietario miPropietario, Fecha miFecha) {
		super();
		this.miPropiedad = miPropiedad;
		this.miPropietario = miPropietario;
		System.out.println("Registro " + miPropiedad.getFicha());
		this.miFecha = miFecha;
	}

	/**
	 * Metodo accesor
	 *
	 * @return miPropiedad
	 */
	public Propiedad getMiPropiedad() {
		return miPropiedad;
	}

	/**
	 * Metodo para fijar la propiedad
	 *
	 * @param miPropiedad
	 */
	public void setMiPropiedad(Propiedad miPropiedad) {
		this.miPropiedad = miPropiedad;
	}

	/**
	 * Metodo accesor
	 *
	 * @return miPropietario
```

Programa 43 Registro

```
	 */
	public Propietario getMiPropietario() {
		return miPropietario;
	}

	/**
	 * Metodo para fijar el propietario
	 *
	 * @param miPropietario
	 */
	public void setMiPropietario(Propietario miPropietario) {
		this.miPropietario = miPropietario;
	}

	/**
	 * Metodo accesor
	 *
	 * @return miFecha
	 */
	public Fecha getMiFecha() {
		return miFecha;
	}

	/**
	 * Metodo para fijar la fecha
	 *
	 * @param miFecha
	 */
	public void setMiFecha(Fecha miFecha) {
		this.miFecha = miFecha;
	}

}
```

La oficina de registro implementada en Programa 44, tiene 3 atributos de tipo arreglo: Propiedad misPropiedades[], Propietario misPropietarios[] y Registro misRegistros[].
El método devolverPrimeraPosicionLibre(Object miArreglo[]), recibe un arreglo de objetos y devuelve la primera posición en donde encuentra null, es decir, es la primera posición libre.
El método agregarPropietario recibe un propietario y lo agrega al arreglo en la primera posición libre. Esta posición se halla con el método devolverPrimeraPosicionLibre.
El método buscarPropietario, recibe el id del propietario y devuelve la posición donde lo encuentra, sino está devuelve -1. El método listarPropiedadesPropietario(String id) recibe el id del propietario y devuelve todas las propiedades que este posee, para ello es necesario recorrer el listado de registros.

Programa 44 OficinaRegistro

```
package co.uniquindio.address.model;

import java.util.ArrayList;
import java.util.Arrays;

/*
 * Clase principal del mundo de la logica
 */
public class OficinaRegistro {
	/**
	 * Se declaran los atributos
	 */
	private Propiedad misPropiedades[];
	private Propietario misPropietarios[];
	private Registro misRegistros[];

	/**
	 * Constructor de la clase
	 */
	public OficinaRegistro() {
		misPropiedades = new Propiedad[10];
		misPropietarios = new Propietario[10];
		misRegistros = new Registro[10];
	}

	/**
	 * Devuelve la primera posicion libre, es decir, que es null
	 *
	 * @param miArreglo El arreglo a evaluar
	 * @return la posicion libre
	 */
	public int devolverPrimeraPosicionLibre(Object miArreglo[]) {
		int pos = -1;
		boolean centinela = false;
		for (int i = 0; i < miArreglo.length && centinela == false; i++) {

			if (miArreglo[i] == null) {
				centinela = true;
				pos = i;
			}

		}
		return pos;
	}

	/**
	 * Permite agregar un propietario
	 *
	 * @param miP El propietario
	 * @throws SinEspacioException si no hay espacio en el arreglo
	 * @throws RepetidoException   Si esta repetido
	 */
```

Programa 44 OficinaRegistro

```
public void agregarPropietario(Propietario miP) throws SinEspacioException,
RepetidoException {
		int pos = devolverPrimeraPosicionLibre(misPropietarios);
		int posPropietario = buscarPropietario(miP.getId());
		if (pos != -1 && posPropietario == -1) {
			misPropietarios[pos] = miP;

		}
		if (pos == -1) {
			throw new SinEspacioException("No hay posiciones libres");
		}
		if (posPropietario != -1) {
			throw new RepetidoException("El propietario ya está registrado");
		}

	}

	/**
	 * Permite agregar una propiedad
	 *
	 * @param miP La propiedad
	 * @throws SinEspacioException si no hay espacio en el arreglo
	 * @throws RepetidoException   Si esta repetido
	 */
	public void agregarPropiedad(Propiedad miP) throws SinEspacioException,
RepetidoException {
		int pos = devolverPrimeraPosicionLibre(misPropiedades);
		int posPropiedad = buscarPropiedad(miP.getFicha());
		if (pos != -1 && posPropiedad == -1) {
			misPropiedades[pos] = miP;

		}
		if (pos == -1) {
			throw new SinEspacioException("No hay posiciones libres");
		}
		if (posPropiedad != -1) {
			throw new RepetidoException("La propiedad ya está registrada");
		}

	}

	/**
	 * Permite agregar un regisro
	 *
	 * @param miR El registro
	 * @throws SinEspacioException No hay espacio en el arreglo
	 */
	public void agregarRegistro(Registro miR) throws SinEspacioException {
		int pos = devolverPrimeraPosicionLibre(misRegistros);

		if (pos != -1) {
			misRegistros[pos] = miR;

		} else {
```

Programa 44 OficinaRegistro

```
                    throw new SinEspacioException("No hay posicion libre para ubicar el
registro");
              }

       }

       /**
        * Buscar un propietario
        *
        * @param miP El id del propietario
        * @return La posicion donde lo encuentra o -1 si no está
        */
       public int buscarPropietario(String miP) {
              boolean centinela = false;
              int pos = -1;
              for (int i = 0; i < misPropietarios.length && centinela == false; i++) {
                    if (misPropietarios[i] != null &&
misPropietarios[i].getId().equals(miP)) {
                           pos = i;
                           centinela = true;
                    }
              }
              return pos;
       }

       /**
        * Busca una propiedad
        *
        * @param miP La propiedad
        * @return La posicion donde lo encuentra o -1 si no está
        */
       public int buscarPropiedad(String miP) {
              boolean centinela = false;
              int pos = -1;
              for (int i = 0; i < misPropiedades.length && centinela == false; i++) {
                    if (misPropiedades[i] != null &&
misPropiedades[i].getFicha().equals(miP)) {
                           pos = i;
                           centinela = true;
                    }
              }
              return pos;
       }

       /**
        * Devuelve la propiedad en la posicion indicada
        *
        * @param pos La posicion a devolver
        * @return La propiedad
        */
       public Propiedad devolverPropiedad(int pos) {
              return misPropiedades[pos];
       }
```

Programa 44 OficinaRegistro

```
	/**
	 * Devuelve el propietario en la propiedad indicada
	 *
	 * @param pos La posicion
	 * @return El propietario
	 */

	public Propietario devolverPropietario(int pos) {
		return misPropietarios[pos];
	}

	/**
	 * Devuelve las propiedades del propietario
	 *
	 * @param id El id del propietario
	 * @return un vector con las propiedades
	 */
	public final Propiedad[] listarPropiedadesPropietario(String id) {

		Propiedad misPropiedadesPropietario[] = new
Propiedad[misPropiedades.length];
		Propiedad misPropiedadesPropietario1[] = null;
		int i, j = 0;
		Registro miR;
		Propietario miPropietario;
		Propiedad miPropiedad;
		for (i = 0; i < misRegistros.length; i++) {
			miR = misRegistros[i];
			if (miR != null) {
				miPropietario = miR.getMiPropietario();

				miPropiedad = miR.getMiPropiedad();
				if (miPropietario != null && miPropietario.getId().equals(id))
{
					JOptionPane.showMessageDialog(null, "valor " + j);
					misPropiedadesPropietario[j] = miPropiedad;
					j++;
				}
			}
		}
		if (j > 0) {
			misPropiedadesPropietario1 = new Propiedad[j];
			System.arraycopy(misPropiedadesPropietario, 0,
misPropiedadesPropietario1, 0, j);
		}
		return misPropiedadesPropietario1;
	}

	/**
	 * Lista las propiedades
	 *
	 * @return La representacion en String del listado de propiedades
	 */
	public String listarPropiedades() {
```

Programa 44 OficinaRegistro

```
		String lista = "";
		for (int i = 0; i < misPropiedades.length; i++) {
			if (misPropiedades[i] != null) {
				lista += "Propiedades " + misPropiedades[i].getFicha() + "    ";
			}
		}
		return lista;
	}

	/**
	 * Lista los propietarios
	 *
	 * @return La representacion en String del listado de propietarios
	 */

	public String listarPropietarios() {
		String lista = "";
		for (int i = 0; i < misPropietarios.length; i++) {
			if (misPropietarios[i] != null) {
				lista += "Propietarios " + misPropietarios[i].getId() + "    ";
			}
		}
		return lista;
	}

	/**
	 * Lista los registros
	 *
	 * @return La representacion en String del listado de registros
	 */
	public String listarRegistros() {
		String lista = "";
		for (int i = 0; i < misRegistros.length; i++) {
			if (misRegistros[i] != null) {
				lista += "Registro:" +
misRegistros[i].getMiPropiedad().getFicha() + "    "
					+ misRegistros[i].getMiPropietario().getId() + "
";
			}
		}
		return lista;
	}

}
```

La clase Principal, presentada en Programa 45 y correspondiente a la vista, tiene la referencia a la oficina de registro, que es la clase principal del mundo del problema. Se implementan métodos para el registro de información entre ellos mostrarVistaListadoPropiedades(), mostrarVistaPropietario(), mostrarVistaRegistro() y mostrarVistaListadoPropiedades(). El método mostrarVistaGeneral() presenta el menú general de la aplicación

Programa 45 Principal

```
package co.uniquindio.address;
import java.io.IOException;
import java.util.Optional;
import co.uniquindio.address.model.OficinaRegistro;
import co.uniquindio.address.model.Propiedad;
import co.uniquindio.address.model.Propietario;
import co.uniquindio.address.model.Registro;
import co.uniquindio.address.model.RepetidoException;
import co.uniquindio.address.model.SinEspacioException;
import co.uniquindio.address.view.ControladorGeneral;
import co.uniquindio.address.view.ControladorPropiedad;
import co.uniquindio.address.view.ControladorPropietario;
import co.uniquindio.address.view.ControladorRegistro;
import co.uniquindio.address.view.ControladorListaVista;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.TextInputDialog;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.stage.Modality;
import javafx.stage.Stage;
/**
 * Clase principal de la interfaz
 * @author sonia
 * @author sergio
 */
public class Principal extends Application {
	/**
	 * Se declaran los atributos
	 */
	private Stage escenarioPrincipal;
	private BorderPane layoutRaiz;
	private OficinaRegistro miOficina;
	private Stage dialogStage;
	private Stage dialogStagePropiedad;
	private Stage dialogStageRegistro;

	@Override
	public void start(Stage primaryStage) {
		miOficina=new OficinaRegistro();
		this.escenarioPrincipal = primaryStage;
		this.escenarioPrincipal.setTitle("");
		inicializarLayoutRaiz();
		mostrarVistaGeneral();
	}

	/**
	 * Inicializa el layout raiz
	 */
	public void inicializarLayoutRaiz() {
```

Programa 45 Principal

```
            try {
                  // Carga el root layout desde un archivo xml
                  FXMLLoader cargador = new FXMLLoader();

      cargador.setLocation(Principal.class.getResource("view/LayoutRaiz.fxml"));
                  layoutRaiz = (BorderPane) cargador.load();

                  // Muestra la escena que contiene el RootLayout
                  Scene scene = new Scene(layoutRaiz);
                  escenarioPrincipal.setScene(scene);
                  escenarioPrincipal.show();
            } catch (IOException e) {
                  System.out.println(e.getMessage());
            }
      }
      /**
       * Permite mostrar la vista general
       */
      public void mostrarVistaGeneral() {
            try {
                  // Carga la vista de la persona.
                  FXMLLoader loader = new FXMLLoader();

      loader.setLocation(Principal.class.getResource("view/VistaGeneral.fxml"));
                  BorderPane vistaGeneral = (BorderPane) loader.load();

                  // Fija la vista de la person en el centro del root layout.
                  layoutRaiz.setCenter(vistaGeneral);
                  // Acceso al controlador.
                  ControladorGeneral miControlador = loader.getController();
                  miControlador.setMiVentanaPrincipal(this);


            } catch (IOException e) {
                  System.out.println(e.getMessage());
            }
      }

      /**
       * muestra el propietario
       */
      public void mostrarVistaPropietario() {
            try {
                  // Carga la vista de la persona.
                  FXMLLoader loader = new FXMLLoader();

      loader.setLocation(Principal.class.getResource("view/PropietarioVista.fxml"));
                  AnchorPane vistaPersona = (AnchorPane) loader.load();


                  // Crea el Stage.
                  dialogStage = new Stage();
                  dialogStage.setTitle("Crear propietario");
                  dialogStage.initModality(Modality.WINDOW_MODAL);
```

Programa 45 Principal

```
                dialogStage.initOwner(escenarioPrincipal);
                Scene scene = new Scene(vistaPersona);
                dialogStage.setScene(scene);

                // Acceso al controlador.
                ControladorPropietario miControlador = loader.getController();
                miControlador.setMiVentanaPrincipal(this);
                dialogStage.showAndWait();
        } catch (Exception e) {
                System.out.println(e.getMessage());
        }
    }
    /**
     * Muestra la propiedad
     */
    public void mostrarVistaPropiedad() {
        try {
                // Carga la vista de la persona.
                FXMLLoader loader = new FXMLLoader();

    loader.setLocation(Principal.class.getResource("view/PropiedadVista1.fxml"));
                BorderPane vistaPropiedad = (BorderPane) loader.load();


                // Crea el Stage.
                dialogStagePropiedad = new Stage();
                dialogStagePropiedad.setTitle("Crear propiedad");
                dialogStagePropiedad.initModality(Modality.WINDOW_MODAL);
                dialogStagePropiedad.initOwner(escenarioPrincipal);
                Scene scene = new Scene(vistaPropiedad);
                dialogStagePropiedad.setScene(scene);
                // Acceso al controlador.
                ControladorPropiedad miControlador = loader.getController();
                miControlador.setMiVentanaPrincipal(this);
                dialogStagePropiedad.showAndWait();
        } catch (IOException e) {
                System.out.println(e.getMessage() + e.getLocalizedMessage());
        }
    }
    /**
     * Muestra el registro
     */
    public void mostrarVistaRegistro() {
        try {
                // Carga la vista de la persona.
                FXMLLoader loader = new FXMLLoader();

    loader.setLocation(Principal.class.getResource("view/RegistroVista.fxml"));
                BorderPane vistaRegistro = (BorderPane) loader.load();

                // Crea el Stage.
                dialogStageRegistro  = new Stage();
                dialogStageRegistro.setTitle("Crear registro");
                dialogStageRegistro.initModality(Modality.WINDOW_MODAL);
```

Programa 45 Principal

```
                dialogStageRegistro.initOwner(escenarioPrincipal);
                Scene scene = new Scene(vistaRegistro);
                dialogStageRegistro.setScene(scene);

                // Acceso al controlador.
                ControladorRegistro miControlador = loader.getController();
                miControlador.setMiVentanaPrincipal(this);
                dialogStageRegistro.showAndWait();
            } catch (IOException e) {
                System.out.println(e.getMessage() + e.getLocalizedMessage());
            }
        }
        /**
         * Muestra el listado de propiedades
         */
        public void mostrarVistaListadoPropiedades() {
            try {
                // Carga la vista de la persona.
                FXMLLoader loader = new FXMLLoader();

        loader.setLocation(Principal.class.getResource("view/ListadoVista.fxml"));
                StackPane vistaRegistro = (StackPane) loader.load();

                // Crea el Stage.
                Stage dialogStage = new Stage();
                dialogStage.setTitle("Listar Propiedades por propietario");
                dialogStage.initModality(Modality.WINDOW_MODAL);
                dialogStage.initOwner(escenarioPrincipal);
                Scene scene = new Scene(vistaRegistro);
                dialogStage.setScene(scene);

                // Acceso al controlador.
                ControladorListaVista miControlador = loader.getController();
                miControlador.setMainApp(this);
                String id=leerMensaje("Ingrese el id del propietario");
                miControlador.actualizarTabla(id);
                dialogStage.showAndWait();
            } catch (Exception e) {
                System.out.println(e.getMessage() + e.getLocalizedMessage());
            }
        }

        /**
         *
         * @return el escenario principal
         */
        public Stage getPrimaryStage() {
            return escenarioPrincipal;
        }
        /**
         * Metodo principal
         * @param args
         */
        public static void main(String[] args) {
```

Programa 45 Principal

```
            launch(args);
      }
      /**
       * Metodo para agregar un propietario
       * @param miP
       * @throws SinEspacioException
       * @throws RepetidoException
       */
      public void agregarPropietario(Propietario miP) throws SinEspacioException,
RepetidoException
      {
            miOficina.agregarPropietario(miP);

      }
      /**
       * Metodo para agregar una propiedad
       * @param miP
       * @throws SinEspacioException
       * @throws RepetidoException
       */
      public void agregarPropiedad(Propiedad miP) throws SinEspacioException,
RepetidoException
      {
            miOficina.agregarPropiedad(miP);
      }

      /**
       * Metodo para agregar un registro
       * @param miR
       * @throws SinEspacioException
       * @throws RepetidoException
       */
      public void agregarRegistro(Registro miR) throws SinEspacioException,
RepetidoException
      {
            miOficina.agregarRegistro(miR);
      }

      /**
       * Metodo para mostrar un mensaja
       * @param mensaje
       * @param miA
       * @param titulo
       * @param cabecera
       * @param contenido Es el mensaje  mostrado
       * @param escenarioPrincipal En donde se muestra el mensaje
       */

      public static void mostrarMensaje(String mensaje, AlertType miA, String titulo,
String cabecera, String contenido, Stage escenarioPrincipal )
      {
            Alert alert = new Alert(miA);
            alert.initOwner(escenarioPrincipal);
            alert.setTitle(titulo);
```

Programa 45 Principal

```
            alert.setHeaderText(cabecera);
            alert.setContentText(contenido);
            alert.showAndWait();
      }
      /**
       * Permite leer un mensaje
       * @param mensaje El texto a mostrar
       * @return El mensaje leido
       */
      public static String leerMensaje(String mensaje )
      {
            String salida="";
            // Muestra el mensaje
            TextInputDialog miDialogo = new TextInputDialog("");
            miDialogo.setTitle("Ingreso de datos");
            miDialogo.setHeaderText("");
            miDialogo.setContentText(mensaje);

            // Forma tradicional de obtener la resuesta.
            Optional<String> result = miDialogo.showAndWait();
            if (result.isPresent()){
                  salida= result.get();
            }

            return salida;
      }
      /**
       * Permite buscar un propietario
       * @param miP el id
       * @return La posicion donde se encuentra o -1
       */
      public int buscarPropietario(String miP)
      {
            return miOficina.buscarPropietario(miP);
      }
      /**
       * Permite buscar una propiedad
       * @param miP la ficha
       * @return La posicion donde se encuentra o -1
       */
      public int buscarPropiedad(String miP)
      {
            return miOficina.buscarPropiedad(miP);
      }
      /**
       * Metodo accesor
       * @param pos Posicion deseada
       * @return La propiedad
       */
      public Propiedad devolverPropiedad(int pos)
      {

            return miOficina.devolverPropiedad(pos);
      }
```

Programa 45 Principal

```
	/**
	 * Devuelve un propitarior
	 * @param pos La posicion deseada
	 * @return El propietario
	 */
	public Propietario devolverPropietario(int pos)
	{

		return miOficina.devolverPropietario(pos);
	}

	/**
	 * Devuelve un listado de propietarios
	 * @param id El identificador del propietario
	 * @return Un vector con propiedades
	 */
	public Propiedad[] listarPropiedadesPropietario(String id)
	{
		return miOficina.listarPropiedadesPropietario(id);
	}
	/**
	 * Metodo accesor
	 * @return dialogStage
	 */
	public Stage getDialogStage() {
		return dialogStage;
	}
	/**
	 * Metodo modificador
	 * @param dialogStage
	 */
	public void setDialogStage(Stage dialogStage) {
		this.dialogStage = dialogStage;
	}
	/**
	 * Metodo accesor
	 * @return dialogStagePropiedad
	 */
	public Stage getDialogStagePropiedad() {
		return dialogStagePropiedad;
	}
	/**
	 * Metodo modificador
	 * @param dialogStagePropiedad
	 */
	public void setDialogStagePropiedad(Stage dialogStagePropiedad) {
		this.dialogStagePropiedad = dialogStagePropiedad;
	}
	/**
	 * Metodo accesor
	 * @return dialogStageRegistro
	 */
	public Stage getDialogStageRegistro() {
		return dialogStageRegistro;
```

Programa 45 Principal

```
	}
	/**
	 * Metodo modificador
	 * @param dialogStageRegistro
	 */
	public void setDialogStageRegistro(Stage dialogStageRegistro) {
		this.dialogStageRegistro = dialogStageRegistro;
	}


}
```

El ControladorPropietario, ver Programa 46, permite capturar y crear el propietario. Ello se hace en el método fijarPropietario(). La información para crear el propietario se captura de los campos de texto. Las instrucciones para crear el propietario son:

```
Propietario miP = new Propietario();
miP.setCiudad(ciudad);
miP.setDireccion(direccion);
miP.setId(id);
miP.setNombre(nombre);
miVentanaPrincipal.agregarPropietario(miP);
```

Programa 46 ControladorPropietario

```
package co.uniquindio.address.view;

import java.awt.Button;

import co.uniquindio.address.Principal;
import co.uniquindio.address.model.Propietario;
import co.uniquindio.address.model.RepetidoException;
import co.uniquindio.address.model.SinEspacioException;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;

/**
 * Permite editar los datos del propietario
 *
 * @author Sonia Jaramillo
 */
public class ControladorPropietario {

	@FXML
	private TextField idTextField;
	@FXML
	private TextField nombreTextField;
	@FXML
```

Programa 46 ControladorPropietario

```
	private TextField direccionTextField;
	@FXML
	private TextField ciudadTextField;

	// para manejar el boton
	private boolean cliqueado = false;

	// VentanaPrincipal
	private Principal miVentanaPrincipal;

	/**
	 * Inicializa la clase contenedor.
	 */
	@FXML
	private void initialize() {
	}

	/**
	 * Metodo modificador
	 *
	 * @param dialogo
	 */

	public Principal getMiVentanaPrincipal() {
		return miVentanaPrincipal;
	}

	public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
		this.miVentanaPrincipal = miVentanaPrincipal;
	}

	/**
	 * Retorna verdadero si el usuario dio clic sobre el boton
	 *
	 * @return
	 */
	public boolean isCliqueado() {
		return cliqueado;
	}

	/**
	 * Metodo para modificar propietario
	 */
	@FXML
	private void fijarPropietario() throws SinEspacioException, RepetidoException {
		String id = idTextField.getText();
		String nombre = nombreTextField.getText();
		String direccion = direccionTextField.getText();
		String ciudad = ciudadTextField.getText();
		Propietario miP = new Propietario();
		miP.setCiudad(ciudad);
		miP.setDireccion(direccion);
		miP.setId(id);
		miP.setNombre(nombre);
```

Programa 46 ControladorPropietario

```
            try {
                  miVentanaPrincipal.agregarPropietario(miP);
                  miVentanaPrincipal.getDialogStage().hide();
            } catch (SinEspacioException e) {
                  // TODO Auto-generated catch block
                  Principal.mostrarMensaje("Error", AlertType.ERROR, "", "",
e.getMessage(),
                              miVentanaPrincipal.getPrimaryStage());
            } catch (RepetidoException e) {
                  // TODO Auto-generated catch block
                  Principal.mostrarMensaje("Error", AlertType.ERROR, "", "",
e.getMessage(),
                              miVentanaPrincipal.getPrimaryStage());
            }
            cliqueado = true;
      }

}
```

El Programa 47 implementa el metodo fijarVivienda() que permite crear una nueva propiedad. El método initialize se encarga de agregar los botones de radio al grupo, para que se muevan en conjunto. Es decir, en un momento dado solo puede estar activo una de las opciones.

Programa 47 ControladorPropiedad

```
package co.uniquindio.address.view;

import java.awt.Button;
import java.net.URL;
import java.util.ArrayList;
import java.util.ResourceBundle;

import co.uniquindio.address.Principal;
import co.uniquindio.address.model.Propiedad;
import co.uniquindio.address.model.Propietario;
import co.uniquindio.address.model.RepetidoException;
import co.uniquindio.address.model.SinEspacioException;
import co.uniquindio.address.model.Tipo;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.RadioButton;
import javafx.scene.control.TextField;
import javafx.scene.control.ToggleGroup;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;

/**
 * Permite editar los datos de la propiedad
 *
```

Programa 47 ControladorPropiedad

```
 * @author Sonia Jaramillo
 */
public class ControladorPropiedad implements Initializable {
	/**
	 * Se declaran los atributos
	 */
	@FXML
	private RadioButton casa;
	@FXML
	private RadioButton lote;
	@FXML
	private TextField fichaTextField;
	@FXML
	private TextField direccionTextField;
	@FXML
	private TextField metrosCuadradosTextField;
	@FXML
	private TextField avaluoTextField;
	@FXML
	private Button guardarButton;
	@FXML
	private ToggleGroup group;

	// VentanaPrincipal
	private Principal miVentanaPrincipal;

	/**
	 * Metodo accesor
	 *
	 * @return miVentanaPrincipal
	 */

	public Principal getMiVentanaPrincipal() {
		return miVentanaPrincipal;
	}

	/**
	 * Metodo modificador
	 *
	 * @param miVentanaPrincipal
	 */
	public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
		this.miVentanaPrincipal = miVentanaPrincipal;
	}

	/**
	 * Metodo para agregar la propiedad
	 */
	@FXML
	private void fijarVivienda() {
		String ficha = fichaTextField.getText();
		double area = Double.parseDouble(metrosCuadradosTextField.getText());
		String direccion = direccionTextField.getText();
		double avaluo = Double.parseDouble(avaluoTextField.getText());
```

Programa 47 ControladorPropiedad

```
		Tipo miT = Tipo.CASA;
		RadioButton button = (RadioButton) group.getSelectedToggle();
		String radioBoton = button.getText();
		if (radioBoton.equals("Lote")) {
			miT = Tipo.LOTE;
		}

		Propiedad miP = new Propiedad(ficha, direccion, avaluo, area, miT);
		try {
			miVentanaPrincipal.agregarPropiedad(miP);
			miVentanaPrincipal.getDialogStagePropiedad().hide();
			;
		} catch (SinEspacioException e) {
			// TODO Auto-generated catch block
			Principal.mostrarMensaje("Error", AlertType.ERROR, "", "",
e.getMessage(),
					miVentanaPrincipal.getPrimaryStage());

		} catch (RepetidoException e) {
			// TODO Auto-generated catch block
			Principal.mostrarMensaje("Error", AlertType.ERROR, "", "",
e.getMessage(),
					miVentanaPrincipal.getPrimaryStage());

		}
	}

	/**
	 * Metodo para inicializar la ventana
	 */
	@Override
	public void initialize(URL arg0, ResourceBundle arg1) {
		// TODO Auto-generated method stub
		// Group
		group = new ToggleGroup();
		casa.setToggleGroup(group);
		lote.setToggleGroup(group);
	}

}
```

El controladorRegistro, ver Programa 48 , permite crear un nuevo Registro, para ello la propiedad y el propietario deben existir previamente en el sistema. El registro requiere, además de propiedad y propietario, que se le ingrese una fecha.

Programa 48 ControladorRegistro

```
package co.uniquindio.address.view;
import java.awt.Button;
import co.uniquindio.address.Principal;
import co.uniquindio.address.model.Fecha;
import co.uniquindio.address.model.NoEncontradoException;
import co.uniquindio.address.model.Propiedad;
```

Programa 48 ControladorRegistro

```
import co.uniquindio.address.model.Propietario;
import co.uniquindio.address.model.Registro;
import co.uniquindio.address.model.RepetidoException;
import co.uniquindio.address.model.SinEspacioException;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;

/**
 * Permite editar los datos del registro
 *
 * @author Sonia Jaramillo
 */
public class ControladorRegistro {
	/**
	 * Se declaran los atributos
	 */
	@FXML
	private TextField idTextField;
	@FXML
	private TextField fichaTextField;
	@FXML
	private TextField diaTextField;
	@FXML
	private TextField mesTextField;
	@FXML
	private TextField anioTextField;

	// VentanaPrincipal
	private Principal miVentanaPrincipal;

	/**
	 * Inicializa la clase contenedor.
	 */
	@FXML
	private void initialize() {
	}

	/**
	 * Metodo modificador
	 *
	 * @param dialogo
	 */

	public Principal getMiVentanaPrincipal() {
		return miVentanaPrincipal;
	}

	/**
	 * Metodo para fijar la ventana
```

Programa 48 ControladorRegistro

```
	 *
	 * @param miVentanaPrincipal
	 */
	public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
		this.miVentanaPrincipal = miVentanaPrincipal;
	}

	/**
	 * Metodo para agregar el registro
	 *
	 * @throws NoEncontradoException
	 */
	@FXML
	private void fijarRegistro() throws SinEspacioException, RepetidoException {
		try {
			Registro miR;
			String id = idTextField.getText();
			String ficha = fichaTextField.getText();
			int dia = Integer.parseInt(diaTextField.getText());
			int mes = Integer.parseInt(mesTextField.getText());
			int anio = Integer.parseInt(anioTextField.getText());
			Fecha miFecha = new Fecha(dia, mes, anio);
			int posPropiedad = miVentanaPrincipal.buscarPropiedad(ficha);
			int posPropietario = miVentanaPrincipal.buscarPropietario(id);
			Propiedad miPropiedad;
			Propietario miPropietario;
			if (posPropiedad == -1) {
				Principal.mostrarMensaje("Error", AlertType.ERROR, "", "",
"Propiedad no encontrada",
					miVentanaPrincipal.getPrimaryStage());
			}
			if (posPropietario == -1) {
				Principal.mostrarMensaje("Error", AlertType.ERROR, "", "",
"Propietario no encontrado",
					miVentanaPrincipal.getPrimaryStage());
			}

			if (posPropiedad != -1 && posPropietario != -1) {

				miPropiedad =
miVentanaPrincipal.devolverPropiedad(posPropiedad);
				miPropietario =
miVentanaPrincipal.devolverPropietario(posPropietario);

				miR = new Registro(miPropiedad, miPropietario, miFecha);
				miVentanaPrincipal.agregarRegistro(miR);
				miVentanaPrincipal.getDialogStageRegistro().hide();
				;
			}
		} catch (SinEspacioException e) {
			// TODO Auto-generated catch block
			Principal.mostrarMensaje("Error", AlertType.ERROR, "", "",
e.getMessage(),
				miVentanaPrincipal.getPrimaryStage());
```

Programa 48 ControladorRegistro

```
            } catch (RepetidoException e) {
                  // TODO Auto-generated catch block
                  Principal.mostrarMensaje("Error", AlertType.ERROR, "", "",
e.getMessage(),
                              miVentanaPrincipal.getPrimaryStage());
            }

      }

}
```

El Programa 49 se encarga de mostrar en un table view el listado de propiedades por propietario. Para facilitar el proceso de actualización de este componente gráfico, se implementa el método initialize, que se encarga de obtener los valores desde la lógica y de actualizar la tabla con la información obtenida de la misma.

```
public void initialize(URL location, ResourceBundle resources) {
            ficha.setCellValueFactory(new PropertyValueFactory<>("Ficha"));
            tipo.setCellValueFactory(new PropertyValueFactory<>("Tipo"));
            // adicionar datos a la tabla
            tbData.setItems(propiedadesLista);

      }
```

Programa 49 ControladorListaVista

```
package co.uniquindio.address.view;
import javafx.application.Platform;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Alert;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.ComboBox;
import java.io.FileNotFoundException;
import java.net.URL;
import java.text.ParseException;
import java.util.Arrays;
import java.util.List;
import java.util.ResourceBundle;
import java.util.Timer;
import java.util.TimerTask;
import javax.swing.text.Segment;
import co.uniquindio.address.Principal;
import co.uniquindio.address.model.Propiedad;
import co.uniquindio.address.model.Tipo;
```

Programa 49 ControladorListaVista

```
/**
 * Controlador para la lista
 *
 * @author sonia
 * @author sergio
 */

public class ControladorListaVista implements Initializable {
	/**
	 * se declaran los atributos
	 */
	@FXML
	private TableView<Propiedad> tbData;
	@FXML
	public TableColumn<Propiedad, String> ficha;
	@FXML
	public TableColumn<Propiedad, Tipo> tipo;
	// Referencia a la ventana principal
	private Principal miVentanaPrincipal;
	// add your data here from any source
	private ObservableList<Propiedad> propiedadesLista = FXCollections
			.observableArrayList(new Propiedad("1", "direccion", 12, 32,
Tipo.CASA));

	/**
	 * Metodo para inicializar
	 */
	@Override
	public void initialize(URL location, ResourceBundle resources) {
		ficha.setCellValueFactory(new PropertyValueFactory<>("Ficha"));
		tipo.setCellValueFactory(new PropertyValueFactory<>("Tipo"));
		// adicionar datos a la tabla
		tbData.setItems(propiedadesLista);

	}

	/**
	 * constructor. constructor es llamado antes de inicializar.
	 */
	public ControladorListaVista() {

	}

	/**
	 * se fija la ventana principal
	 *
	 * @param ventanaPrincipal
	 */
	public void setMainApp(Principal ventanaPrincipal) {
		this.miVentanaPrincipal = ventanaPrincipal;
	}

	public void actualizarTabla(String id) {
		tbData.getItems().clear();
```

Programa 49 ControladorListaVista

```
            Propiedad misP[] = miVentanaPrincipal.listarPropiedadesPropietario(id);
            if (miVentanaPrincipal != null && misP != null) {
                  for (int i = 0; i < misP.length; i++) {
                        if (misP[i] != null)
                              propiedadesLista.add(misP[i]);
                        // System.out.println("Ficha
"+propiedadesLista.get(i).getFicha());
                  }
            }
            tbData.setItems(propiedadesLista);
      }

}
```

El ConntroladorGeneral, ver Programa 50, se encarga de controlar el menú general de la aplicación.

Programa 50 ControladorGeneral

```
package co.uniquindio.address.view;
import java.awt.Button;
import co.uniquindio.address.Principal;
import co.uniquindio.address.model.Propietario;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;


/**
 *Es el controlador general
 *
 * @author Sonia Jaramillo
 */
public class ControladorGeneral {

      //VentanaPrincipal
      private Principal miVentanaPrincipal;

      /**
       * Inicializa la clase contenedor.
       */
      @FXML
      private void initialize() {
      }

      /**
       * Metodo accesor
       * @return la ventana principal
       */

      public Principal getMiVentanaPrincipal() {
```

Programa 50 ControladorGeneral

```
		return miVentanaPrincipal;
	}
	/**
	 * Metodo modificador
	 *
	 * @param la ventana principal
	 */

	public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
		this.miVentanaPrincipal = miVentanaPrincipal;
	}
	/**
	 * metodo para crear el propietario
	 */
	public void crearPropietario()
	{
		miVentanaPrincipal.mostrarVistaPropietario();

	}
	/**
	 * Metodo para crear la propiedad
	 */
	public void crearPropiedad()
	{

		miVentanaPrincipal.mostrarVistaPropiedad();
	}

	/**
	 * Metodo para crear el registro
	 */
	public void crearRegistro()
	{
		miVentanaPrincipal.mostrarVistaRegistro();
	}
	/**
	 * Metodo para listar
	 */
	public void listarPropiedadesPropietario()
	{
		miVentanaPrincipal.mostrarVistaListadoPropiedades();

	}


}
```

El Programa 51 tiene el código para crear la interfaz del propietario, ver Figura 95.

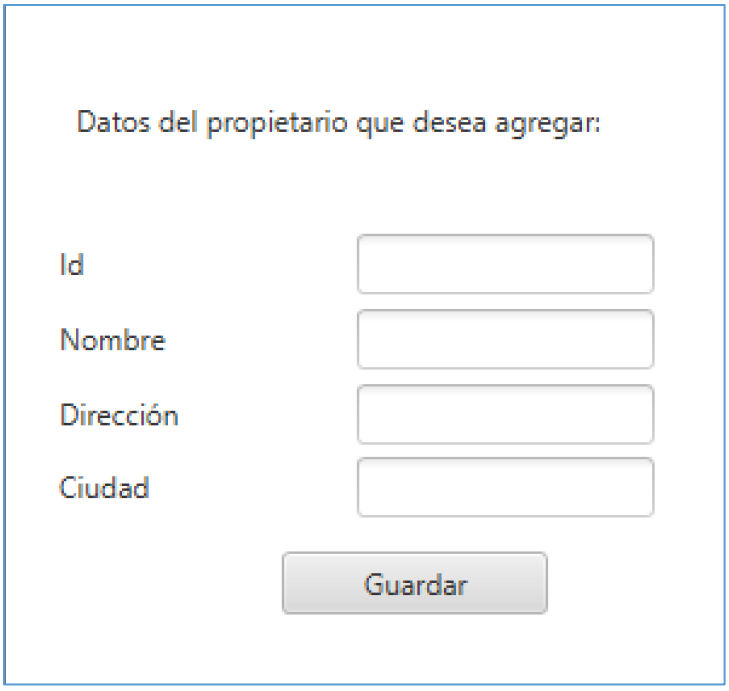


Figura 95 Ingreso del propietario

Programa 51 PropietarioVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>

<AnchorPane prefHeight="282.0" prefWidth="297.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.address.view.ControladorPropietario">
   <children>
      <Label layoutX="29.0" layoutY="36.0" prefHeight="27.0" prefWidth="246.0"
text="Datos del propietario que desea agregar:" />
      <GridPane layoutX="22.0" layoutY="93.0" prefHeight="122.0" prefWidth="246.0"
AnchorPane.bottomAnchor="67.0" AnchorPane.leftAnchor="22.0" AnchorPane.rightAnchor="29.0"
AnchorPane.topAnchor="93.0">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Label text="Id" />
```

Programa 51 PropietarioVista.fxml

```
            <Label text="Nombre" GridPane.rowIndex="1" />
            <Label text="Dirección" GridPane.rowIndex="2" />
            <Label text="Ciudad" GridPane.rowIndex="3" />
            <TextField fx:id="idTextField" prefHeight="25.0" prefWidth="144.0"
GridPane.columnIndex="1" />
            <TextField fx:id="nombreTextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
            <TextField fx:id="direccionTextField" GridPane.columnIndex="1"
GridPane.rowIndex="2" />
            <TextField fx:id="ciudadTextField" prefHeight="0.0" prefWidth="133.0"
GridPane.columnIndex="1" GridPane.rowIndex="3" />
         </children>
      </GridPane>
      <Button layoutX="114.0" layoutY="227.0" mnemonicParsing="false"
onAction="#fijarPropietario" prefHeight="26.0" prefWidth="110.0" text="Guardar"
AnchorPane.leftAnchor="114.0" AnchorPane.rightAnchor="73.0" AnchorPane.topAnchor="227.0"
/>
   </children>
</AnchorPane>
```

PropiedadVista1.fxml contiene las instrucciones para crear la interfaz presente en ***Figura 96***.

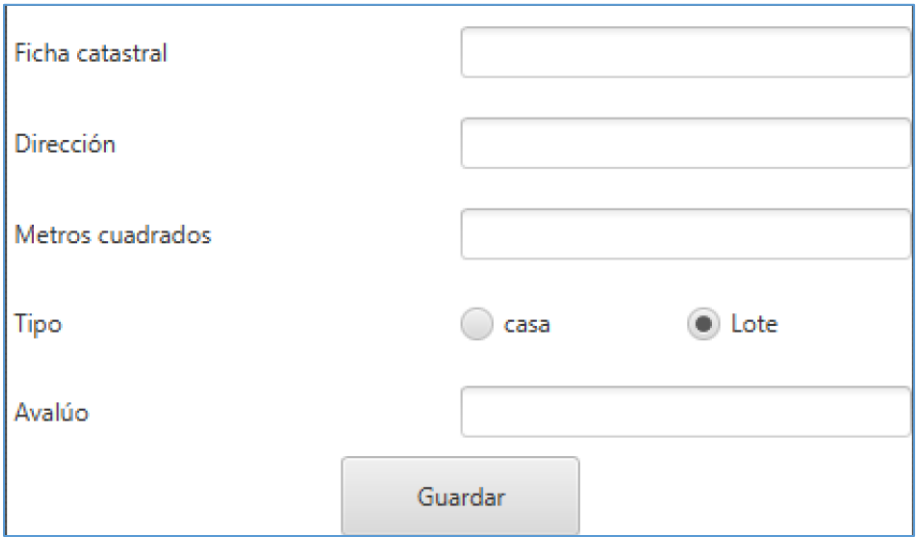


Figura 96 Ingreso de la propiedad

Programa 52 PropiedadVista1.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>
```

Programa 52 PropiedadVista1.fxml

```
<BorderPane prefHeight="257.0" prefWidth="439.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.address.view.ControladorPropiedad">
   <center>
      <GridPane prefHeight="416.0" prefWidth="708.0" BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Label text=" Ficha catastral" />
            <Label text=" Dirección" GridPane.rowIndex="1" />
            <Label text=" Avalúo" GridPane.rowIndex="4" />
            <Label text=" Metros cuadrados" GridPane.rowIndex="2" />
            <Label text=" Tipo" GridPane.rowIndex="3" />
            <GridPane prefHeight="102.0" prefWidth="358.0" GridPane.columnIndex="1"
GridPane.rowIndex="3">
              <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
              </columnConstraints>
              <rowConstraints>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
              </rowConstraints>
               <children>
                  <RadioButton fx:id="casa" mnemonicParsing="false" text="casa" />
                  <RadioButton fx:id="lote" mnemonicParsing="false" selected="true"
text="Lote" GridPane.columnIndex="1" />
               </children>
            </GridPane>
            <TextField fx:id="fichaTextField" GridPane.columnIndex="1" />
            <TextField fx:id="direccionTextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
            <TextField fx:id="metrosCuadradosTextField" GridPane.columnIndex="1"
GridPane.rowIndex="2" />
            <TextField fx:id="avaluoTextField" GridPane.columnIndex="1"
GridPane.rowIndex="4" />
         </children>
      </GridPane>
   </center>
   <bottom>
      <Button mnemonicParsing="false" onAction="#fijarVivienda" prefHeight="39.0"
prefWidth="116.0" text="Guardar" BorderPane.alignment="CENTER" />
   </bottom>
</BorderPane>
```

La interfaz presente en la Figura 97 se construye con el código fuente presente en el Programa 53.

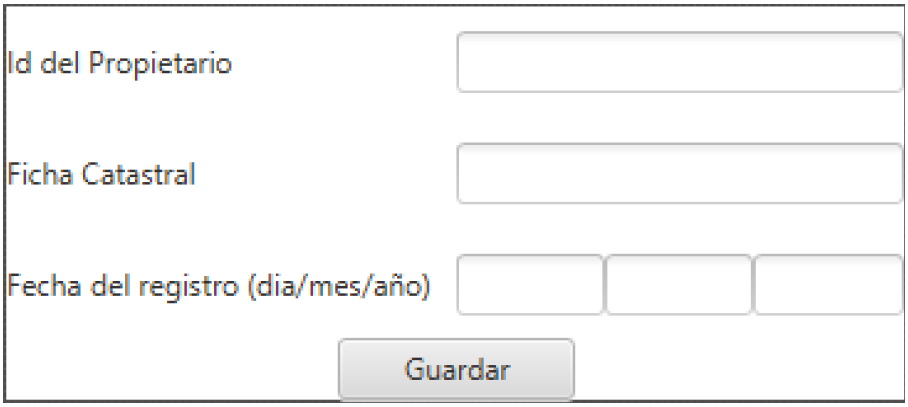


Figura 97 Ingreso del registro

Programa 53 RegistroVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="159.0" prefWidth="363.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.address.view.ControladorRegistro">
   <center>
      <GridPane prefHeight="370.0" prefWidth="509.0" BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Label text="Ficha Catastral" GridPane.rowIndex="1" />
            <Label text="Id del Propietario" />
            <Label text="Fecha del registro (dia/mes/año)" GridPane.rowIndex="2" />
            <GridPane GridPane.columnIndex="1" GridPane.rowIndex="2">
              <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
                  <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0"
/>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
              </columnConstraints>
              <rowConstraints>
```

Programa 53 RegistroVista.fxml

```
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
              </rowConstraints>
               <children>
                  <TextField fx:id="diaTextField" />
                  <TextField fx:id="anioTextField" GridPane.columnIndex="2" />
                  <TextField fx:id="mesTextField" GridPane.columnIndex="1" />
               </children>
            </GridPane>
            <TextField fx:id="idTextField" GridPane.columnIndex="1" />
            <TextField fx:id="fichaTextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
         </children>
      </GridPane>
   </center>
   <bottom>
      <Button mnemonicParsing="false" onAction="#fijarRegistro" prefHeight="13.0"
prefWidth="96.0" text="Guardar" BorderPane.alignment="CENTER" />
   </bottom>
</BorderPane>
```

El listado de propiedades por propietario se visualiza en un tableView, ver ***Figura 98***. El código fuente para crear este componente se presenta en Programa 54.

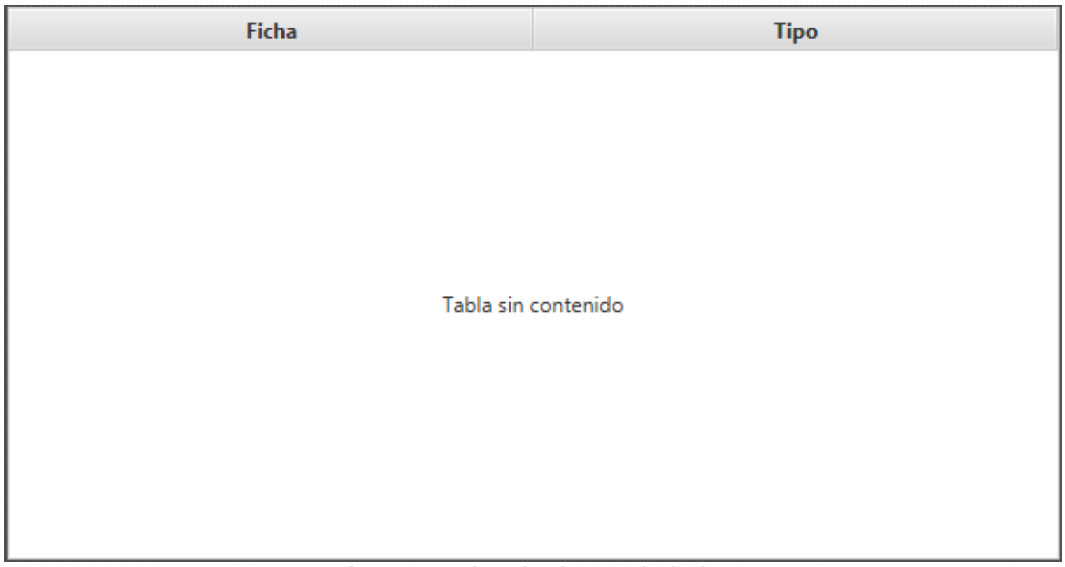


Figura 98 Listado de propiedades

Programa 54 ListadoVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.TableColumn?>
<?import javafx.scene.control.TableView?>
<?import javafx.scene.layout.StackPane?>
```

Programa 54 ListadoVista.fxml

```
<StackPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-
Infinity" prefHeight="315.0" prefWidth="600.0" xmlns="http://javafx.com/javafx/10.0.1"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.address.view.ControladorListaVista">
   <children>
      <TableView fx:id="tbData" prefHeight="200.0" prefWidth="200.0">
        <columns>
          <TableColumn fx:id="ficha" prefWidth="75.0" text="Ficha" />
          <TableColumn fx:id="tipo" prefWidth="75.0" text="Tipo" />
        </columns>
         <columnResizePolicy>
            <TableView fx:constant="CONSTRAINED_RESIZE_POLICY" />
         </columnResizePolicy>
      </TableView>
   </children>
</StackPane>
```

El menú principal se construye con el código fuente presente en Programa 55.

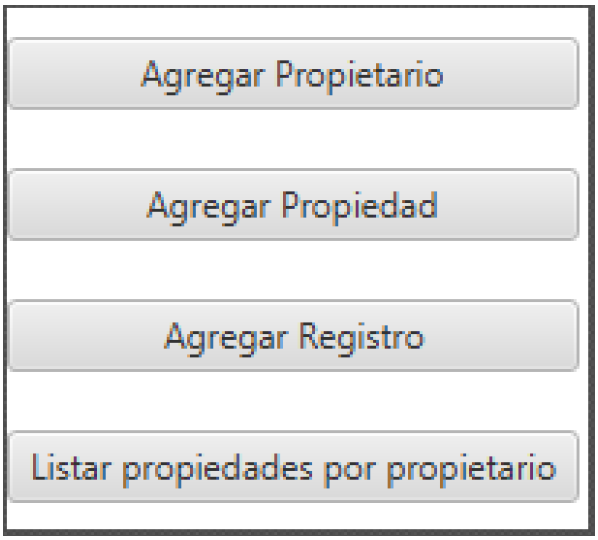


Figura 99 Menú general de la aplicación

Programa 55 VistaGeneral.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="179.0" prefWidth="200.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.address.view.ControladorGeneral">
```

Programa 55 VistaGeneral.fxml

```
   <center>
      <GridPane prefHeight="271.0" prefWidth="279.0" BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
            <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Button mnemonicParsing="false" onAction="#crearPropietario"
prefWidth="197.0" text="Agregar Propietario" />
            <Button mnemonicParsing="false" onAction="#crearPropiedad" prefWidth="197.0"
text="Agregar Propiedad" GridPane.rowIndex="1" />
            <Button mnemonicParsing="false" onAction="#crearRegistro" prefWidth="197.0"
text="Agregar Registro" GridPane.rowIndex="2" />
            <Button mnemonicParsing="false" onAction="#listarPropiedadesPropietario"
text="Listar propiedades por propietario" GridPane.rowIndex="3" />
         </children>
      </GridPane>
   </center>
</BorderPane>
```

El Programa 56 presente el layout raiz, que puede visualizarse en la Figura 100.

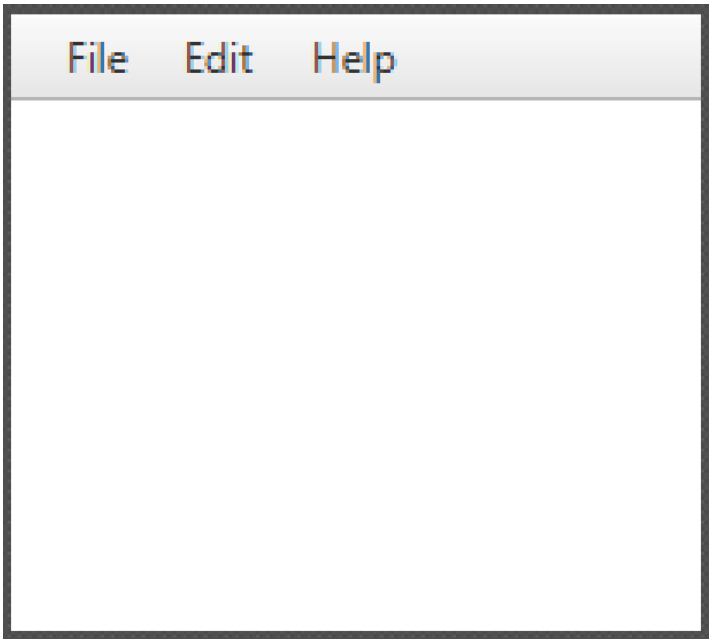


Figura 100 Layout Raiz

Programa 56 LayoutRaiz.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="179.0" prefWidth="200.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1">
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem mnemonicParsing="false" text="Close" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Edit">
            <items>
              <MenuItem mnemonicParsing="false" text="Delete" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Help">
            <items>
              <MenuItem mnemonicParsing="false" text="About" />
            </items>
          </Menu>
        </menus>
      </MenuBar>
   </top>
</BorderPane>
```

3.9 Gestión de cadenas

A diferencia de lenguaje C, Java no trata las cadenas como un conjunto de caracteres contiguos en memoria, sino como objetos. Un String es una clase final, lo cual significa que después de inicializada, los caracteres que la componen no pueden ser modificados. El recorrido de un String se realiza de forma similar al recorrido de un arreglo, la una diferencia radica en que debe usarse el método charAt [2][3][4].

Los constructores de la clase String son los siguientes:

1. Crea una cadena vacia.

```
String cadena = new String ("");
```

2. Se crea un **String** a partir de un array de caracteres.

```
char[] arreglo = { 'h', 'o', 'l', 'a' };
String cadena  = new String( arreglo );
```

3. Se crea un **String** a partir de un array de caracteres especificando la posición a partir de la cual se desea copiar y la cantidad de caracteres a copiar.

```
char[] arreglo = { 'h', 'o', 'l', 'a' };
String cadena  = new String( arreglo, 1, 2 );
```

4. Se crea un **String** con los mismos caracteres que otro objeto **String**

```
 String s1 = "Hola";
 String s2 = new String( s1 );
```

Sobre los Strings es posible realizar operaciones como concatenación, extracción de caracteres, comparación de cadenas, búsqueda dentro de las cadenas, cálculo de la longitud, particionamiento basado en un separador, obtención de subcadenas, entre otras.

a. Concatenación

Para unir dos cadenas6 Java proporciona el operador +. Ejemplo:

```
String saludo  = "Hola";
int    edad    = 20;
String saludo1 = saludo + "amigos" + "mi edad es : " + edad + "años";
```

b. Extracción de caracteres

Existen tres métodos que permiten lograr este objetivo:

1. **charAt():** permite extraer un solo carácter de la cadena. Su sintaxis es la siguiente:

```
char charAt( int posicion );
```

El siguiente ejemplo ilustra el uso de esta instrucción:

```
String cadena ="Hola a todos";
char letra=hallarLetraConMayorAscci(cadena);
System.out.println("La letra con mayor ASCCI es  "+letra);
```

```
public char hallarLetraConMayorAscci(String cadena)
      {
      char mayor=cadena.charAt(0);
      char caracter;
      for(int i=1; i<cadena.length(); i++)
      {   caracter=cadena.charAt(i);
          if(caracter>mayor)
```

```
            {
              mayor=caracter;
            }
      }
       return mayor;
      }
```

Imprime:
La letra con mayor ASCCI es t

2. **getChars():** Permite extraer más de un carácter. Su sintáxis es la siguiente:

```
char getChars(int posicioninicial, int posicionfinal,char[] destino, int destinoinicio);
```

Ejemplo:

```
char arreglo2[]=new char [2];
cadena.getChars(1, 3, arreglo2, 0);
System.out.println("Ejemplo de getChar " +Arrays.toString(arreglo2));
```

Imprime:
Ejemplo de getChar [o, l]

3. **toCharArray():** devuelve una array de caracteres a partir de un String. Su sintaxis es: char [] toCharArray();

```
char arreglo[]=cadena.toCharArray();
System.out.println(Arrays.toString(arreglo));
```

Imprime:
[H, o, l, a, , a, , t, o, d, o, s]

c. Comparación de cadenas

Para comparar cadenas se pueden usar los métodos equals y **equalsIgnoreCase**(). El primer método compara cadenas teniendo en cuenta si son mayúsculas o minúsculas. El segundo, ignora éste aspecto. La sintaxis es la siguiente:

```
boolean equals           ( Object cadena );
boolean equalsIgnoreCase ( Object cadena );
```

Tenga muy claro que el método equals y el **==** no realizan la misma tarea, ya que el primero compara los caracteres de ambas cadenas y el segundo las referencias de ambos objetos para determinar si se trata de una misma instancia.

```
String salida="";
if(!cadena.equals("Hola A todos"))
      {
      salida=("Son diferentes");
      }
```

El método **compareTo**() permite comparar cadenas para determinar cuál de ellas es menor alfabéticamente. Su forma general es:

```
int compareTo( String cadena );
```

Si el valor retornado es menor que cero entonces la cadena que llama al método es menor que cadena. Si es positivo entonces ocurre lo contrario. Si es igual a cero es porque son iguales.

Debe tener presente que éste método realiza distinción entre mayúsculas y minúsculas, por lo que las letras que empiezan con mayúsculas saldrán primero.

```
String salida="";
if(cadena.compareTo("Ana ")>0)
      {
       salida=(cadena +"es alfabeticamente mayor");
      }
```

d. Búsqueda en las cadenas

Para determinar si un carácter está presente en una cadena se puede hacer uso de dos métodos: El método **indexOf**() devuelve la primera ocurrencia de un carácter dentro de una cadena y el método **lastIndexOf**() retorna la última aparición del carácter. La sintaxis es la siguiente:

```
int indexOf     ( int caracter );
int lastIndexOf ( int carácter);
```

Un ejemplo de uso de esta instrucción se presenta a continuación:

```
cadena ="Hola a todos";
int posicionLetra=cadena.lastIndexOf('w');
System.out.println("La posicion de la letra es  "+posicionLetra);
```

El mensaje que se imprime es: La posicion de la letra es -1

e. Cálculo de la longitud

Para calcular la longitud de una cadena se utiliza el método **length**(). Por ejemplo si se tiene:

```
String cadena = "Hola";
int longitud=cadena.length(); // Debe retornar 4
```

f. Modificación de una cadena

Para reemplazar un carácter dentro de una cadena se debe usar el método replace. Su sintaxis es la siguiente:

```
String replace( char original, char reemplazo );
```

```
cadena ="Hola a todos";
cadena=cadena.replace('o','p');
System.out.println("La nueva cadena es "+cadena);
```

El mensaje que se imprime es: La nueva cadena es Hpla a tpdps

Los métodos **toLowerCase()** y **toUpperCase()** permiten convertir los caracteres de una cadena a minúscula o mayúscula respectivamente. La forma de estos dos métodos es la siguiente:

```
String toLowerCase();
String toUpperCase();
```

El siguiente ejemplo pasa una cadena a mayúscula.

```
cadena ="Hola a todos";
cadena= cadena.toUpperCase();
System.out.println("La nueva cadena es "+cadena);
```

El resultado que se imprime es: La nueva cadena es HOLA A TODOS

Para eliminar los espacios presentes en una cadena debe usarse el método **trim**(). Tal como se muestra a continuación:

```
cadena ="Hola a todos";
cadena= "       "+cadena+"                          ";
cadena = cadena.trim();
System.out.println("La longitud de la cadena es "+cadena.length());
```

Imprime:

La longitud de la cadena es 12

g. Particionamiento basado en un separador

Para particionar cadenas basadas en un separador, se puede utilizar Split o StringTokenizer. El método split particiona la cadena partiendo de las coincidencias con la expresión regular dada. El siguiente ejemplo ilustra el uso de split.

```
String cadena = "Hola:Es un ensayo";
String [] arreglo = cadena.split(":");
```

La invocación del método split retorna un arreglo de strings obtenido a partir de la expresión regular indicada.

```
arreglo[0]= "Hola";
arreglo[1]= "Es un ensayo";
```

Es posible hacer uso de patrones para realizar el particionamiento. Un patrón es una expresión compilada de una expresión regular. El patrón definido para el siguiente ejemplo es Pattern.*compile*("[,/ +]+"). Es decir, se buscan dentro de la frase los símbolos [, / +]. El más al final indica que la expresión regular se puede repetir una o más veces.

```
Pattern patron = Pattern.compile("[,/ +]+");
//La entrada del split es el patrón
String result[]= patron.split(linea);
```

La clase StringTokenizer permite particionar un String en tokens. Un token, o componente léxico, es un string que tiene significado consistente. Para reservarle memoria a la clase StringTokenizer se requiere el String, el

delimitador, y la bandera. Si esta última toma el valor de false, el delimitador servirá para separar los tokens (pero él por si mismo no será considerado uno). Si está en true, el delimitador, será considerado un token.

```
StringTokenizer tokens = new StringTokenizer(linea, "# ", false);
String arreglo[]= new String[tokens.countTokens()];
int contador=0;
while(tokens.hasMoreElements())
      {
        arreglo[contador]=tokens.nextToken();
        contador++;
      }
```

Los siguientes métodos pueden ser usados para particionar cadenas. Estos hacen uso de split y StringTokenizer.

```
/**
* Devuelve un arrreglo con los elementos del String enviado luego de particionar
* @param linea La linea a procesar, linea!=null
* @return Un arreglo con cada una de las palabras
*/
public static String[] particionar (String linea)
{
/*
*Un token es un componente que significa algo en un lenguaje de programación, ejemplo
identificador, palabras claves, signos, números
*/
StringTokenizer tokens = new StringTokenizer(linea, "# ", false);
String arreglo[]= new String[tokens.countTokens()];
int contador=0;
while(tokens.hasMoreElements())
{
  arreglo[contador]=tokens.nextToken();
  contador++;
}
return arreglo;
}


public static String[] particionarSplit (String linea)
{
Pattern patron = Pattern.compile("[,/ +]+");
//La entrada del split es el patrón
String result[]= patron.split(linea);
return result;
}
```

h. Subcadenas

Una subcadena es una porción de la cadena original. La instrucción substring permite la obtención de subcadenas, para ello se le debe indicar el rango de posiciones que se desea copiar.

```
String linea="amarillo azul# rojo verde#";
System.out.println("Una subcadena "+linea.substring(0, 4));
//Imprime:
//Una subcadena  amar
```

i. Verificar si una cadena inicia o finaliza con una subcadena

Las instrucciones startsWith y endsWith permiten verificar si una cadena inicia o finaliza con un determinado String. Para el siguiente ejemplo se verifica si se inicia o finaliza con la palabra ("ama"). Si es así se inicializa la variable centinela con el valor de true.

```
boolean centinela=false;
if(linea.startsWith("ama"))
	{
		centinela=true;
	}

if(linea.endsWith("ama"))
	{
		centinela=true;
	}
```

j. Obtención del valor primitivo del objeto indicado

Para obtener el valor primitivo se utiliza la instrucción **ValueOf**. A continuación se presenta un ejemplo de su uso:

```
String cadena ="Hola a todos ";
double dato=5;
cadena +=cadena.valueOf(dato);
System.out.println("el dato es "+cadena);
```

3.10 Caso de estudio 1Unidad III: Operaciones entre 2 conjuntos

Se desea crear una operación para realizar operaciones entre conjuntos. Se debe permitir:

Agregar los 2 conjuntos

Realizar la unión

Realizar la intersección

3.10.1 Comprensión del problema

a) Requisitos funcionales

NOMBRE	R1- Agregar un conjunto
RESUMEN	Permite agregar un nuevo conjunto
ENTRADAS	
Los elementos del conjunto, el numero de conjunto (0 ó 1)	
RESULTADOS	
Un nuevo conjunto ha sido agregado	

NOMBRE	R2-Unir 2 conjuntos
RESUMEN	Permite unir 2 conjuntos
ENTRADAS	
No se ingresa nada, dado que los conjuntos ya fueron ingresados y son atributos de la clase	
RESULTADOS	
Un conjunto	

NOMBRE	R3-Permite halla la intersección de 2 conjuntos
RESUMEN	
ENTRADAS	
No se ingresa nada, dado que los conjuntos ya fueron ingresados y son atributos de la clase	
RESULTADOS	
Un conjunto	

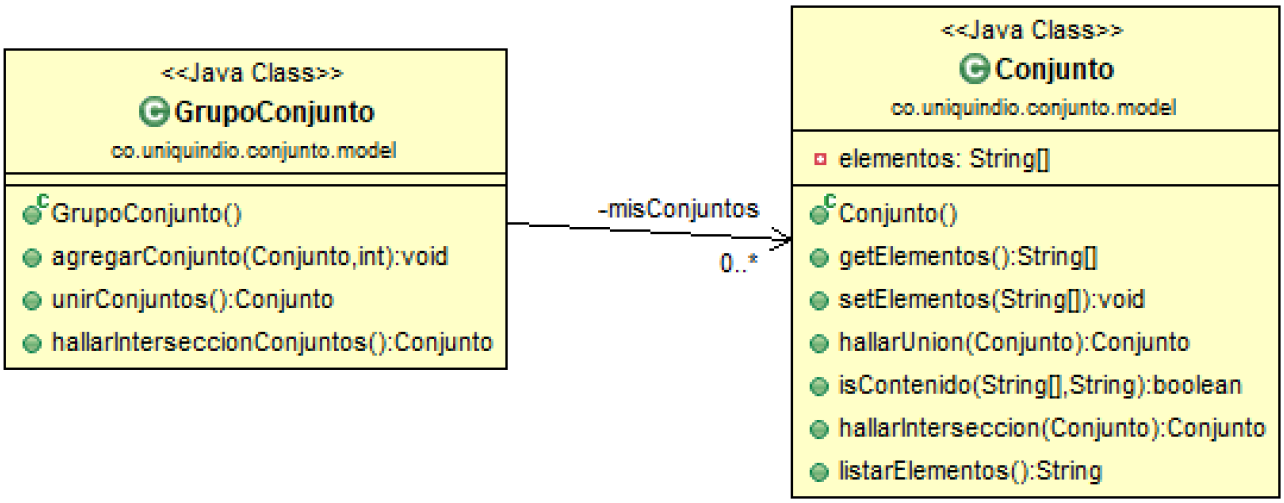
b) El modelo del mundo del problema

Las actividades que se deben realizar para construir el modelo del mundo son:

Identificar las entidades o clases.

ENTIDAD DEL MUNDO	DESCRIPCIÓN
GrupoConjunto	Es la entidad más importante del mundo del problema.
Conjunto	Es un atributo del grupo de conjuntos

El diagrama de clases correspondiente para este caso de estudio es el siguiente:



La clase Conjunto, presente en Programa 57 , permite representar un conjunto. El método hallarUnion tiene como parámetros a un conjunto. El primer paso que se realiza es copiar en el vector resultante todos los elementos del primer conjunto. Luego, se recorren todos los elementos del segundo conjunto y, por cada uno de sus elementos, se verifica que no este contenido en el conjunto resultante, si es así se agrega, de lo contrario se descarta.

El método hallarInterseccion recorrre el conjunto1 y por cada uno de sus elementos verifica si está contenido en el conjunto2, si es así, lo agrega al vector resultante. Esta operación requiere el uso de un índice (contador) que permita almacenar el elemento en la posición correcta.

Programa 57 Conjunto

```
package co.uniquindio.conjunto.model;

import java.util.Arrays;

/**
 * Clase para representar un conjunto
 *
 * @author sonia
 * @author sergio
 */
public class Conjunto {
	/**
	 * Los elementos del conjunto
	 */
	private String elementos[];

	/**
	 * Devuelve los elementos del conjunto
	 *
	 * @return elementos
	 */
	public String[] getElementos() {
```

Programa 57 Conjunto

```
		return elementos;
	}

	/**
	 * Metodo modificador
	 *
	 * @param elementos
	 */
	public void setElementos(String[] elementos) {
		this.elementos = elementos;

	}

	/**
	 * Halla la union de dos conjuntos
	 *
	 * @param conjunto1 El conjunto 1
	 * @param conjunto2 el conjunto 2
	 * @return La union de los dos conjuntos
	 */
	public Conjunto hallarUnion(Conjunto miConjunto) {
		String resultado[] = new String[elementos.length +
miConjunto.getElementos().length];
		int contador = 0;
		String dato = "";
		Conjunto miC;
		// se copia primero el primer vector
		for (int i = 0; i < elementos.length; i++) {
			resultado[contador] = elementos[i];
			contador++;
		}

		for (int i = 0; i < miConjunto.getElementos().length; i++) {
			dato = miConjunto.getElementos()[i];
			if (isContenido(resultado, dato) == false) {
				resultado[contador] = dato;
				contador++;
			}

		}
		// Se eliminan los espacios en null
		resultado = Arrays.copyOf(resultado, contador);
		miC = new Conjunto();
		miC.setElementos(resultado);
		return miC;
	}

	/**
	 * Verifica si dato está contenido en el arreglo
	 *
	 * @param conjunto El arreglo de elementos
	 * @param dato     El dato a buscar
	 * @return Un true si se encuentra en el arreglo, false en caso contrario
	 */
```

Programa 57 Conjunto

```
	public boolean isContenido(String conjunto[], String dato) {
		boolean centinela = false;
		for (int i = 0; i < conjunto.length && centinela == false; i++) {
			if (conjunto[i] != null && (conjunto[i].trim()).equals(dato.trim())) {
				centinela = true;
			}
		}

		return centinela;
	}

	/**
	 * Halla la interseccion de dos conjuntos
	 *
	 * @param conjunto1 El conjunto 1
	 * @param conjunto2 eE l conjunto 2
	 * @return La interseccion de los dos conjuntos
	 */
	public Conjunto hallarInterseccion(Conjunto miConjunto2) {
		String resultado[];
		int tamMaximo = elementos.length;
		int contador = 0;
		String dato = "";
		Conjunto miC = new Conjunto();
		// se halla el tamaño maximo del conjunto
		if (miConjunto2.getElementos().length > tamMaximo) {
			tamMaximo = miConjunto2.getElementos().length;
		}

		resultado = new String[tamMaximo];

		// se verifica si un elemento está en ambos conjuntos
		for (int i = 0; i < elementos.length; i++) {
			dato = elementos[i];
			if (isContenido(miConjunto2.getElementos(), dato)) {
				resultado[contador] = dato;
				contador++;
			}
		}
		// Se eliminan los espacios en null
		resultado = Arrays.copyOf(resultado, contador);
		miC.setElementos(resultado);

		return miC;
	}

	/**
	 * Devuelve el listado de los elementos
	 *
	 * @return la representacion en string del conjunto
	 */
	public String listarElementos() {
		String salida = "";
```

Programa 57 Conjunto

```
		for (int i = 0; i < elementos.length; i++) {
			salida += elementos[i] + "    ";
			if (i != 0 && i % 10 == 0) {
				salida += "\n";
			}
		}
		return salida;
	}

}
```

El Programa 58 permite ingresar los 2 conjuntos y tiene los métodos de unión e intersección. Los métodos se presentan a continuación:

```
public Conjunto unirConjuntos() {
		//al conjunto 1 se le une el conjunto 2
		Conjunto miC = misConjuntos[0].hallarUnion(misConjuntos[1]);
		return miC;
}


public Conjunto hallarInterseccionConjuntos() {
		//se halla la intersección del conjunto1 con el conjunto 2
		Conjunto miC = misConjuntos[0].hallarInterseccion(misConjuntos[1]);
		return miC;
}
```

Programa 58 GrupoConjuntos

```
package co.uniquindio.conjunto.model;

import java.util.Arrays;

/**
 * Clase principal del mundo del problema
 *
 * @author sonia
 * @author sergio
 */
public class GrupoConjunto {
	/**
	 * Atributos
	 */
	private Conjunto misConjuntos[];

	/**
	 * Constructor de la clase
	 */
	public GrupoConjunto() {
		misConjuntos = new Conjunto[2];
	}
```

Programa 58 GrupoConjuntos

```
	/**
	 * Permite agregar un conjunto
	 *
	 * @param miConjunto el conjunto
	 * @param pos        La posicion donde su ubica el conjunto
	 */
	public void agregarConjunto(Conjunto miConjunto, int pos) {
		misConjuntos[pos] = miConjunto;
	}

	/**
	 * Permite unir dos conjuntos
	 *
	 * @return un conjunto
	 */
	public Conjunto unirConjuntos() {
		Conjunto miC = misConjuntos[0].hallarUnion(misConjuntos[1]);
		return miC;
	}

	/**
	 * Halla la interseccion de 2 conjuntos
	 *
	 * @return un conjunto
	 */
	public Conjunto hallarInterseccionConjuntos() {
		Conjunto miC = misConjuntos[0].hallarInterseccion(misConjuntos[1]);
		return miC;
	}
}
```

La clase Principal, implementada en Programa 59, contiene la referencia a la clase principal del mundo. El método mostrarVistaGeneral() se encarga de visualizar la interfaz para ingresar los dos conjuntos y efectuar las operaciones de unión e intersección.

Programa 59 Clase `Principal`

```
package co.uniquindio.conjunto;
import java.io.IOException;
import java.util.Optional;
import javax.swing.JOptionPane;
import co.uniquindio.conjunto.model.Conjunto;
import co.uniquindio.conjunto.model.GrupoConjunto;
import co.uniquindio.conjunto.view.ControladorConjunto;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.TextInputDialog;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
```

Programa 59 Clase Principal

```
import javafx.stage.Modality;
import javafx.stage.Stage;

/**
 * Es la clase principal de la interfaz
 *
 * @author sonia
 * @author sergio
 */
public class Principal extends Application {
	/**
	 * Atributos de la clase
	 */
	private Stage escenarioPrincipal;
	private BorderPane layoutRaiz;
	private GrupoConjunto miGrupoConjunto;

	@Override
	public void start(Stage primaryStage) {
		miGrupoConjunto = new GrupoConjunto();
		this.escenarioPrincipal = primaryStage;
		this.escenarioPrincipal.setTitle("");
		inicializarLayoutRaiz();
		mostrarVistaGeneral();
	}

	/**
	 * Inicializa el layout raiz
	 */
	public void inicializarLayoutRaiz() {
		try {
			// Carga el root layout desde un archivo xml
			FXMLLoader cargador = new FXMLLoader();

	cargador.setLocation(Principal.class.getResource("view/LayoutRaiz.fxml"));
			layoutRaiz = (BorderPane) cargador.load();

			// Muestra la escena que contiene el RootLayout
			Scene scene = new Scene(layoutRaiz);
			escenarioPrincipal.setScene(scene);
			escenarioPrincipal.show();
		} catch (IOException e) {
			System.out.println(e.getMessage());
		}
	}

	public void mostrarVistaGeneral() {
		try {
			// Carga la vista generla
			FXMLLoader loader = new FXMLLoader();

	loader.setLocation(Principal.class.getResource("view/ConjuntoView.fxml"));
			BorderPane vistaGeneral = (BorderPane) loader.load();
```

Programa 59 Clase `Principal`

```
                    // Fija la vista general en el centro del root layout.
                    layoutRaiz.setCenter(vistaGeneral);
                    // Acceso al controlador.
                    ControladorConjunto miControlador = loader.getController();
                    miControlador.setMiVentanaPrincipal(this);
             } catch (IOException e) {
                    System.out.println(e.getMessage());
             }
      }

      /**
       * Devuelve el escenario principal
       *
       * @return
       */
      public Stage getPrimaryStage() {
            return escenarioPrincipal;
      }

      /**
       * Metodo principal
       *
       * @param args
       */
      public static void main(String[] args) {
            launch(args);
      }

      /**
       * Metodo para mostrar un mensaje
       *
       * @param mensaje
       * @param miA
       * @param titulo
       * @param cabecera
       * @param contenido          El mensaje
       * @param escenarioPrincipal El escenarioPrincipal donde se muestra el mensaje
       */
public static void mostrarMensaje(String mensaje, AlertType miA, String titulo, String
cabecera, String contenido,
                    Stage escenarioPrincipal) {
             Alert alert = new Alert(miA);
             alert.initOwner(escenarioPrincipal);
             alert.setTitle(titulo);
             alert.setHeaderText(cabecera);
             alert.setContentText(contenido);
             alert.showAndWait();
      }

      /**
       * Permite leer un mensaje
       *
       * @param mensaje
       * @return el mensaje leido
```

Programa 59 Clase `Principal`

```
	 */
	public static String leerMensaje(String mensaje) {
		String salida = "";
		// Muestra el mensaje
		TextInputDialog miDialogo = new TextInputDialog("");
		miDialogo.setTitle("Ingreso de datos");
		miDialogo.setHeaderText("");
		miDialogo.setContentText(mensaje);

		// Forma tradicional de obtener la resuesta.
		Optional<String> result = miDialogo.showAndWait();
		if (result.isPresent()) {
			salida = result.get();
		}
		return salida;
	}

	/**
	 * Permite agregar un conjunto en una posicion
	 *
	 * @param miConjunto
	 * @param pos        La posicion donde se agrega el conjuto
	 */
	public void agregarConjunto(Conjunto miConjunto, int pos) {
		miGrupoConjunto.agregarConjunto(miConjunto, pos);
	}

	/**
	 * Permite unir 2 conjuntos
	 *
	 * @return Un conjunto
	 */
	public Conjunto unirConjuntos() {
		return miGrupoConjunto.unirConjuntos();
	}

	/**
	 * Permite hallar la union de 2 conjuntos
	 *
	 * @return un conjunto
	 */
	public Conjunto hallarInterseccionConjuntos() {
		return miGrupoConjunto.unirConjuntos();
	}
}
```

La clase ControladorConjunto, ver Programa 60, se encarga de la lectura de los conjuntos y de implementar los métodos para las acciones de los botones. Los métodos guardarConjunto0() y guardarConjunto1() permiten ingresar los conjuntos. Observe que para obtener los elementos de los conjuntos se usa el método Split.
Los métodos hallarInterseccion y unirConjuntos() son usados desde la interfaz para configurar las acciones de los botones.

Programa 60 ControladorConjunto

```
package co.uniquindio.conjunto.view;
import java.awt.Button;
import java.net.URL;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.ResourceBundle;
import co.uniquindio.conjunto.Principal;
import co.uniquindio.conjunto.model.Conjunto;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.RadioButton;
import javafx.scene.control.TextField;
import javafx.scene.control.ToggleGroup;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;

/**
 * Permite editar los datos del conjunto
 *
 * @author Sonia Jaramillo
 */
public class ControladorConjunto implements Initializable {
	/**
	 * Se declaran los atributos de la clase
	 */
	@FXML
	private TextField conjunto1TextField;
	@FXML
	private TextField conjunto2TextField;
	// VentanaPrincipal
	private Principal miVentanaPrincipal;

	/**
	 * Metodo accesor
	 *
	 * @return miVentanaPrincipal;
	 */
	public Principal getMiVentanaPrincipal() {
		return miVentanaPrincipal;
	}

	/**
	 * Metodo modificador
	 */
	public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
		this.miVentanaPrincipal = miVentanaPrincipal;
	}

	@FXML
	private void inicializar() {
```

Programa 60 ControladorConjunto

```
	}

	@Override
	public void initialize(URL arg0, ResourceBundle arg1) {
		// TODO Auto-generated method stub
		// Group

	}

	/**
	 * Permite crear un nuevo conjunto, el primero
	 */
	public void guardarConjunto0() {
		Conjunto miC = new Conjunto();
		String e[] = conjunto1TextField.getText().split(",");
		miC.setElementos(e);
		miVentanaPrincipal.agregarConjunto(miC, 0);
	}

	/**
	 * Permite crear un nuevo conjunto, el segundo
	 */
	public void guardarConjunto1() {
		Conjunto miC = new Conjunto();
		String e[] = conjunto2TextField.getText().split(",");
		miC.setElementos(e);
		miVentanaPrincipal.agregarConjunto(miC, 1);
	}

	/**
	 * Permite unir dos conjuntos
	 */
	public void unirConjuntos() {
		Conjunto miC = miVentanaPrincipal.unirConjuntos();
		Principal.mostrarMensaje("Union", AlertType.INFORMATION, "", "La unión es:
", miC.listarElementos(),
				miVentanaPrincipal.getPrimaryStage());
	}

/**
* Permite hallar la interseccion de 2 conjuntos
*/
public void hallarInterseccion() {
Conjunto miC = miVentanaPrincipal.hallarInterseccionConjuntos();
Principal.mostrarMensaje("Union", AlertType.INFORMATION, "", "La interseccion es: ",
miC.listarElementos(),
				miVentanaPrincipal.getPrimaryStage());
	}

}
```

ConjuntoView.fxml, presente en Programa 61, permite construir la interfaz presente en Figura 101.

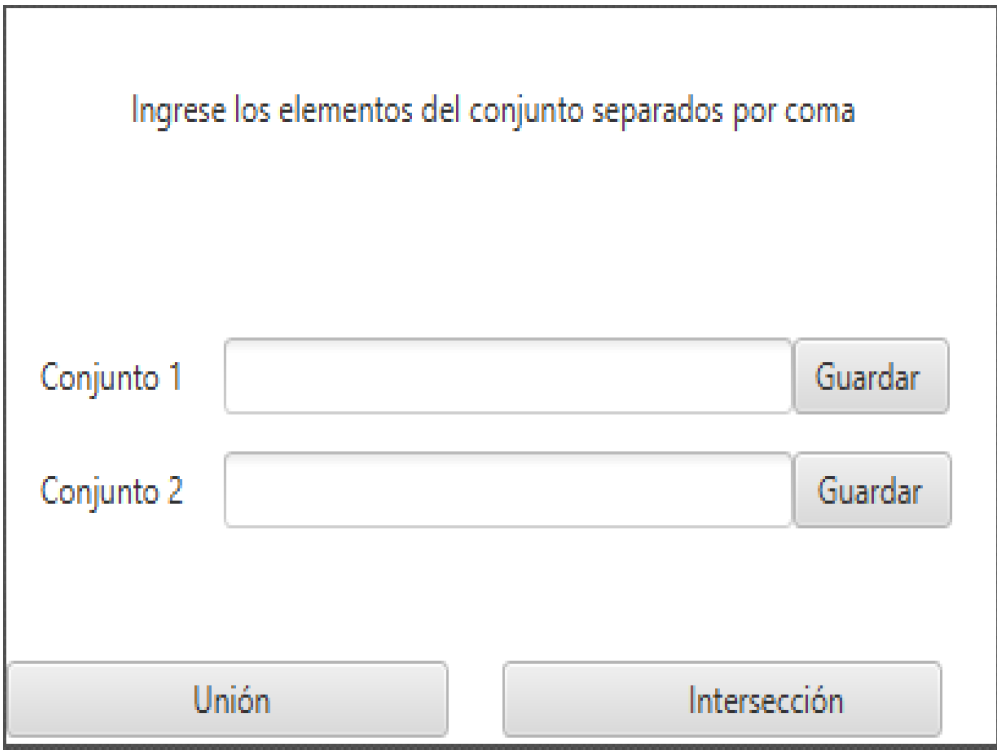


Figura 101 Agregar los 2 conjuntos y realizar operaciones de unión e intersección

Programa 61 ConjuntoView.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="239.0" prefWidth="405.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.conjunto.view.ControladorConjunto">
   <center>
      <AnchorPane prefHeight="190.0" prefWidth="407.0" BorderPane.alignment="CENTER">
         <children>
            <GridPane layoutX="14.0" layoutY="100.0" prefHeight="75.0" prefWidth="379.0">
              <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" maxWidth="175.0" minWidth="10.0"
prefWidth="78.0" />
                <ColumnConstraints hgrow="SOMETIMES" maxWidth="313.0" minWidth="10.0"
prefWidth="235.0" />
                  <ColumnConstraints hgrow="SOMETIMES" maxWidth="313.0" minWidth="10.0"
prefWidth="73.0" />
              </columnConstraints>
```

Programa 61 ConjuntoView.fxml

```
               <rowConstraints>
                 <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
                 <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
               </rowConstraints>
                <children>
                   <Label text="Conjunto 1" />
                   <Label text="Conjunto 2" GridPane.rowIndex="1" />
                   <TextField fx:id="conjunto1TextField" prefHeight="23.0"
prefWidth="230.0" GridPane.columnIndex="1" />
                   <TextField fx:id="conjunto2TextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
                   <Button mnemonicParsing="false" onAction="#guardarConjunto0"
prefHeight="25.0" prefWidth="64.0" text="Guardar " GridPane.columnIndex="2" />
                   <Button mnemonicParsing="false" onAction="#guardarConjunto1"
prefHeight="25.0" prefWidth="65.0" text="Guardar " GridPane.columnIndex="2"
GridPane.rowIndex="1" />
                </children>
            </GridPane>
            <Label layoutX="51.0" layoutY="24.0" text="Ingrese los elementos del conjunto
separados por coma" />
         </children>
      </AnchorPane>
   </center>
   <bottom>
      <GridPane BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Button mnemonicParsing="false" onAction="#unirConjuntos" prefHeight="25.0"
prefWidth="181.0" text=" Unión" />
            <Button mnemonicParsing="false" onAction="#hallarInterseccion"
prefHeight="25.0" prefWidth="180.0" text="           Intersección"
GridPane.columnIndex="1" />
         </children>
      </GridPane>
   </bottom>
</BorderPane>
```

Finalmente, LayoutRaiz.fxml presenta el código para la interfaz presente en la Figura 102.

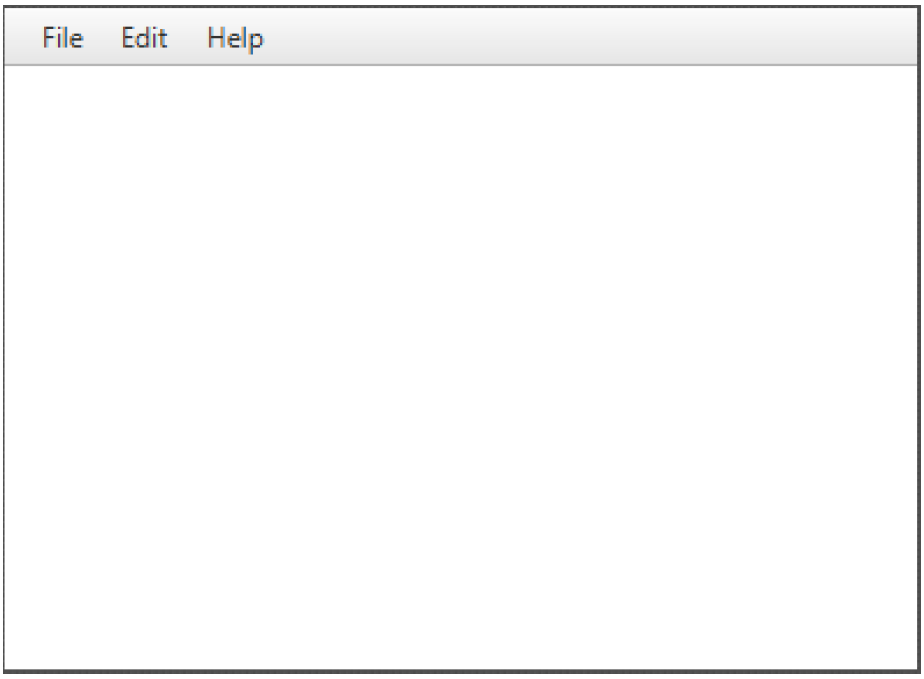


Figura 102 Layout raiz

Programa 62 LayoutRaiz.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="285.0" prefWidth="393.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1">
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem mnemonicParsing="false" text="Close" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Edit">
            <items>
              <MenuItem mnemonicParsing="false" text="Delete" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Help">
            <items>
              <MenuItem mnemonicParsing="false" text="About" />
            </items>
          </Menu>
```

Programa 62 LayoutRaiz.fxml

```
        </menus>
      </MenuBar>
   </top>
</BorderPane>
```

3.11 ArrayList

Un ArrayList es una estructura contenedoras de tamaño variable, muy útil para cuando no se conoce a priori el tamaño máximo de la estructura, pues crece o disminuye dinámicamente. Cuando se quieren representar ArrayList en el diagrama de clases, se pone la multiplicidad de 0…*, indicando que el tamaño de la estructura puede variar. Todo ArrayList en Java debe ser declarado y también se le debe reservar memoria. Tal como se muestra a continuación:

```
ArrayList <Persona> listaPersona ;

listaPersona=new ArrayList<Persona>();
```

Observe que cuando se le reservó memoria no fue necesario dejar explícita la cantidad de elementos que iba a almacenar la estructura. Java automáticamente arranca el ArrayList con 0 elementos dentro de él.

Algunos métodos de los métodos de la clase ArrayList se muestran a continuación:

size(): Devuelve el tamaño de la estructura, es decir, la cantidad de elementos que contiene

isEmpty(): Indica si hay o no elementos en el ArrayList.

add(elemento): permite insertar un nuevo elemento al final de la estructura contenedora.

set(posición, elemento): reemplaza el elemento que se hay en la posición indicada por uno nuevo.

contains(elemento): Si el elemento está contenido se devuelve true, de lo contrario false.

toArray(). Copia los elementos de la estructura a un arreglo de objetos.

add(posición, elemento): permite insertar un elemento en la posición indicada. “Si ya había un elemento en esa posición, el elemento que ya existe y todos los que se encuentran a su derecha se correrán una posición hacia la derecha”.

remove (posición): borra el elemento que está en la posición indicada, esto implica que los elementos que estaban a la derecha del elemento eliminado se correrán hacia la izquierda para ocupar el lugar eliminado. Esta operación hace que el tamaño de la estructura se reduzca en 1.

remove(elemento): en este caso se envía el objeto que se desea eliminar. Es importante aclarar, que si se crea un nuevo elemento con los datos del objeto que se desea eliminar no indica que sea el mismo elemento. Para que sean iguales deben ocupar la misma posición de memoria, es decir, deben ser la mismas instancia”[9].

[9] Programación orientada a Objetos - Sonia Jaramillo, Sergio Augusto Cardona, Leonardo Alonso Hernández

3.12 Caso de estudio 2 Unidad III: Grupo de estudiantes

Se desea crear una aplicación para registrar la información de una planilla de estudiantes. Un estudiante tiene un id, un nombre y 3 asignaturas registradas. Paradigma, introduccion y geometria. Cada una de estas asignaturas tiene un código dado por el sistema. Se debe permitir:
Agregar un nuevo estudiante
Agregar las 3 asignaturas al sistema
Agregar un nuevo registro de un estudiante a la planilla, que contenga notas y asignaturas cursadas.
Dado un estudiante, obtener la asignatura con mayor nota
Hallar la nota promedio de toda la planilla

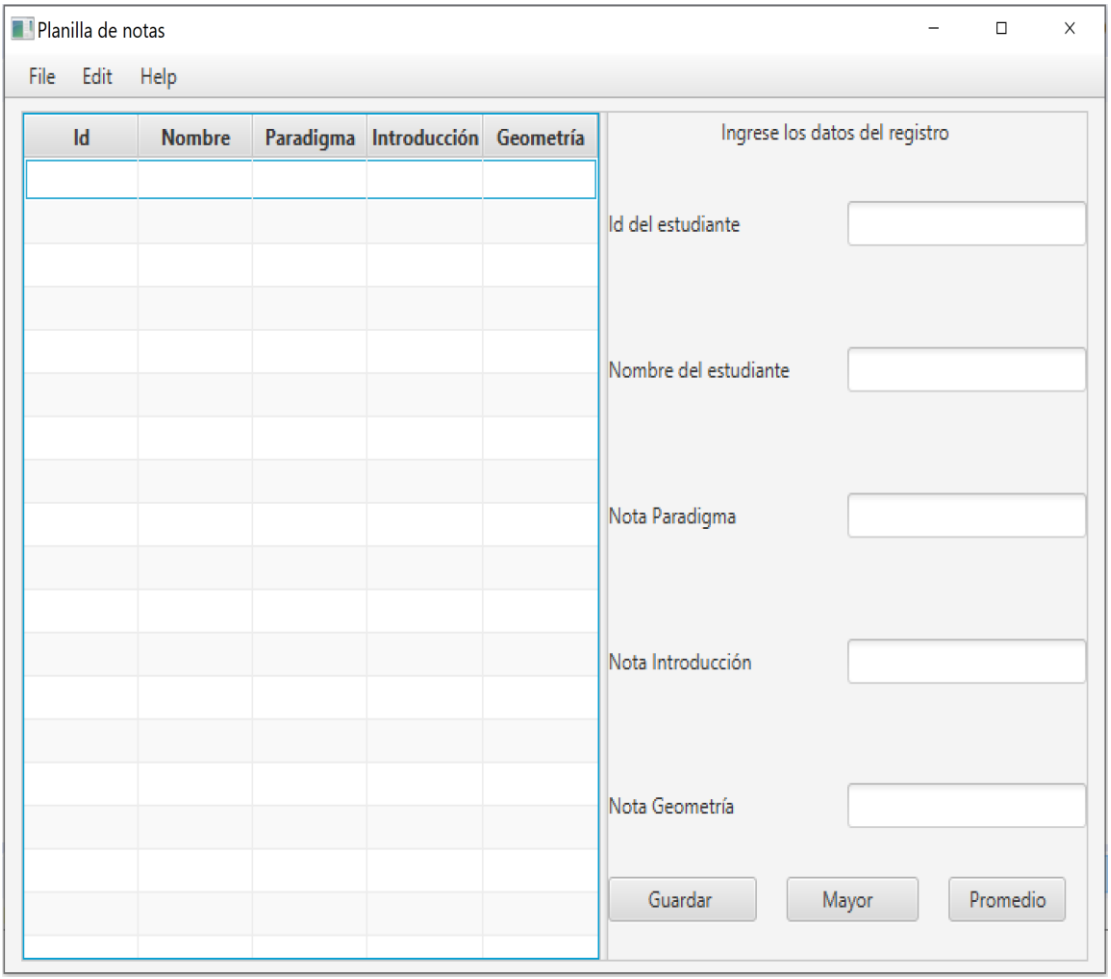


Figura 103 Interfaz planilla de estudiantes

3.12.1 Comprensión del problema

a) Requisitos funcionales

NOMBRE	R1- Agregar un estudiante al sistema
RESUMEN	Permite agregar un nuevo estudiante al sistema
ENTRADAS	
Codigo y nombre del estudiante	
RESULTADOS	
Un nuevo estudiante ha sido agregado	

NOMBRE	R2- Agregar una asignatura al sistema
RESUMEN	Permite agregar una nueva asignatura
ENTRADAS	
Codigo y nombre de la asignatura	
RESULTADOS	
Una nueva asignatura se agrega al sistema	

NOMBRE	R3-Crear un nuevo registro para la planilla
RESUMEN	Permite crear nuevo registro para la planilla
ENTRADAS	
Listado de notas, listado de asignaturas y el estudiante	
RESULTADOS	
Un nuevo registro ha sido agregado	

NOMBRE	R3-Obtener asignatura con mayor nota
RESUMEN	Permite obtener la asignatura de un estudiante dado, con mayor nota
ENTRADAS	
El id del estudiante	
RESULTADOS	
Una asignatura	

NOMBRE	R4-Hallar la nota promedio de la planilla
RESUMEN	Permite obtener la nota promedio de toda la planilla
ENTRADAS	
ninguna	
RESULTADOS	
El promedio	

b) El modelo del mundo del problema

Las actividades que se deben realizar para construir el modelo del mundo son:

Identificar las entidades o clases.

ENTIDAD DEL MUNDO	DESCRIPCIÓN
	Es la entidad más importante del mundo del problema.

El diagrama de clases correspondiente para este caso de estudio es el siguiente:

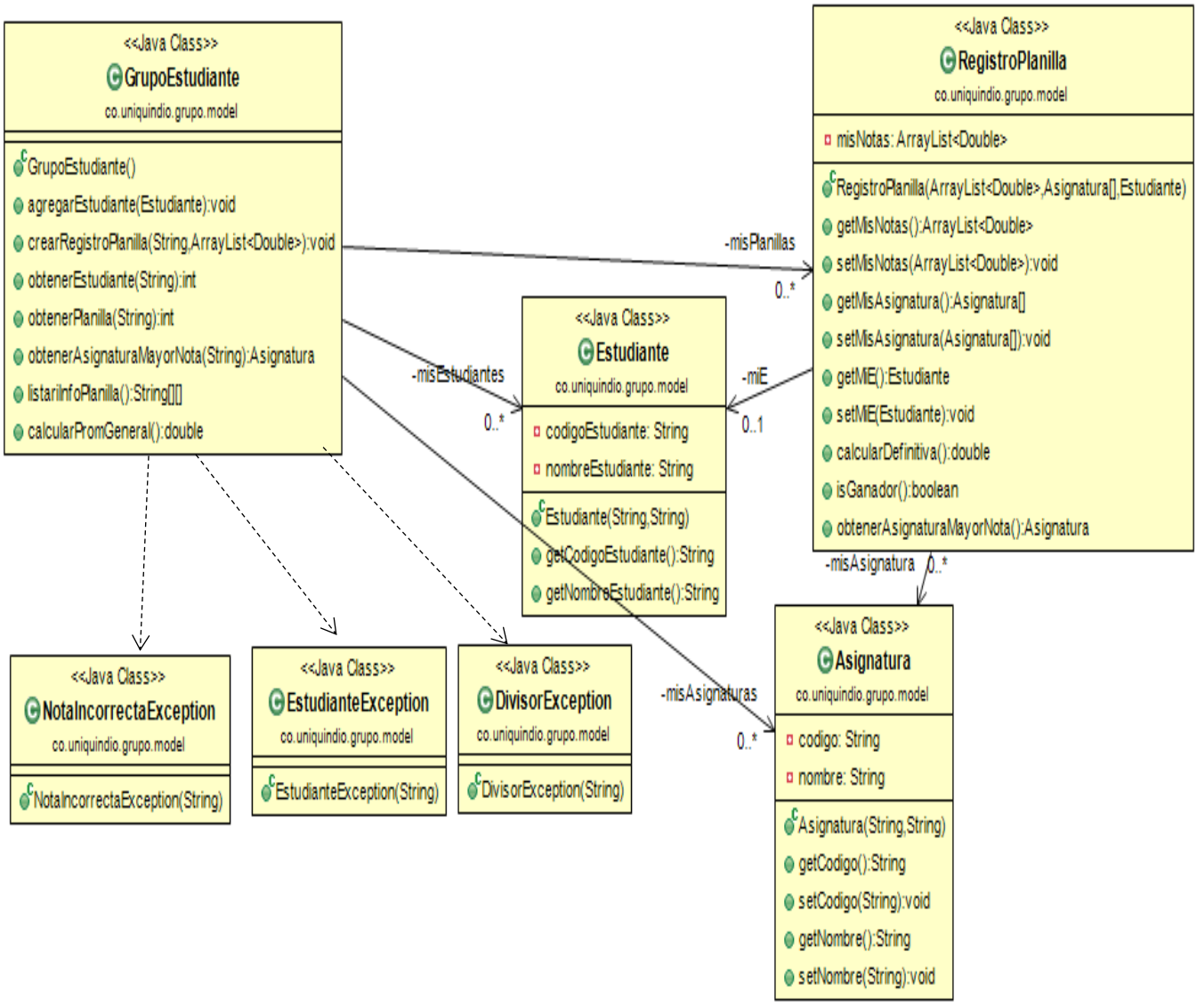


Figura 104 Diagrama de clases planilla de estudiantes

La primera clase en implementar se muestra en Programa 63 y es la clase Asignatura. Esta clase tiene 2 atributos String código y String nombre, además de los métodos get, set y constructor.

Programa 63 Asignatura

```
package co.uniquindio.grupo.model;

/**
 * Clase para representar una asignatura
 *
 * @author sonia
 * @author sergio
 */
public class Asignatura {
	/**
	 * Atributos de la asignatura
	 */
	private String codigo;
	private String nombre;

	/**
	 * Constructor de la asignatura
	 *
	 * @param codigo el codigo de la asignatura
	 * @param nombre El nombre de la asignatura
	 */
	public Asignatura(String codigo, String nombre) {
		super();
		this.codigo = codigo;
		this.nombre = nombre;
	}

	/**
	 * Metodo accesor
	 *
	 * @return codigo
	 */
	public String getCodigo() {
		return codigo;
	}

	/**
	 * Metodo modificador
	 *
	 * @param codigo El codigo
	 */
	public void setCodigo(String codigo) {
		this.codigo = codigo;
	}

	/**
	 * Metodo accesor
	 *
	 * @return nombre
```

Programa 63 Asignatura

```
	 */
	public String getNombre() {
		return nombre;
	}

	/**
	 * Metodo modificador
	 *
	 * @param nombre
	 */
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
}
```

La clase Estudiante, ver Programa 64, tiene 2 atributos: String codigoEstudiante y String nombreEstudiante, además del método constructor y los correspondientes métodos get y set, para los atributos.

Programa 64 Estudiante

```
package co.uniquindio.grupo.model;

/**
 * @version 1.0
 * @author Sonia Jaramillo Valbuena
 * @author Sergio A. Cardona
 *
 *         Esta es la clase principal del mundo
 *
 */
public class Estudiante {
//Se definen los atributos
	private String codigoEstudiante;
	private String nombreEstudiante;

	/**
	 * Constructor del la clase Estudiante
	 *
	 * @param codigoEstudiante codigo del estudiante solo puede tomar valores
	 *                         numericos
	 * @param nombreEstudiante solo puede tener letras
	 * @param Fecha            miFechaNacimiento
	 */
	public Estudiante(String codigoEstudiante, String nombreEstudiante) {
		this.codigoEstudiante = codigoEstudiante;
		this.nombreEstudiante = nombreEstudiante;

	}

	/**
	 * Metodo accesor
	 *
	 * @return String el codigo del Estudiante
	 */
```

Programa 64 Estudiante

```
	public String getCodigoEstudiante() {
		return codigoEstudiante;
	}

	/**
	 * Metodo accesor
	 *
	 * @return String el nombre del estudiante
	 */

	public String getNombreEstudiante() {
		return nombreEstudiante;
	}

}
```

Las clases NotaIncorrectaException, DivisorException y EstudianteException permite realizar todo el tratamiento de excepciones en la aplicación.

Programa 65 NotaIncorrectaException

```
package co.uniquindio.grupo.model;

/**
 * Excepcion si se ingresa una nota fuera del rando
 *
 * @author sonia
 * @author sergio
 */
public class NotaIncorrectaException extends Exception {
	/**
	 * constructor de la clase
	 *
	 * @param mensaje El mensaje a mostrar
	 */
	public NotaIncorrectaException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

Programa 66 DivisorException

```
package co.uniquindio.grupo.model;

/**
 * Clase para representar una excepcion
 *
 * @author sonia
 * @author sergio
 */
public class DivisorException extends Exception {
	/**
	 * Constructor de la clase
```

Programa 66 DivisorException

```
	 *
	 * @param mensaje El mensaje a mostrar
	 */
	public DivisorException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

Programa 67 EstudianteException

```
package co.uniquindio.grupo.model;

/**
 * Clase para construir una excepcion si el estudiante esta repetido
 *
 * @author sonia
 * @author sergio
 */
public class EstudianteException extends Exception {
	/**
	 * Constructor de la clase
	 *
	 * @param mensaje Es el mensaje a mostrar
	 */
	public EstudianteException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

El Programa 68 representa un nuevo registro para la planilla. Esta clase tiene tres atributos:

```
private ArrayList<Double> misNotas;
private Asignatura misAsignatura[];
private Estudiante miE;
```

El método calcularDefinitiva() recorre el ArrayList de notas y acumula en un variable cada una de ellas, para luego dividir por la cantidad de notas.
El método obtenerAsignaturaMayorNota() recorre todas las notas del arraylist misNotas, evalua cada una de ellas y la compara con una variable mayor. Esto le permite obtener el índice de la nota más alta.Con este índice se retorna la asignatura que está en el índice hallado.

Programa 68 RegistroPlanilla

```
package co.uniquindio.grupo.model;

import java.util.ArrayList;

/**
 * @version 1.0
```

Programa 68 RegistroPlanilla

```
 * @author Sonia Jaramillo Valbuena
 * @author Sergio A. Cardona
 *
 *         Maneja la informacion referente a la asignatura
 *
 */

public class RegistroPlanilla {
	/**
	 * Atributos de la clase
	 */
	private ArrayList<Double> misNotas;
	private Asignatura misAsignatura[];
	private Estudiante miE;

	/**
	 * Constructor de la clase
	 *
	 * @param misNotas, las notas del estudiante
	 * @param misAsignatura, las asignaturas
	 * @param miE, El estudiante
	 */
	public RegistroPlanilla(ArrayList<Double> misNotas, Asignatura[] misAsignatura,
Estudiante miE) {
		super();
		this.misNotas = misNotas;
		this.misAsignatura = misAsignatura;
		this.miE = miE;
	}

	/**
	 * Metodo accesor
	 *
	 * @return misNotas
	 */
	public ArrayList<Double> getMisNotas() {
		return misNotas;
	}

	/**
	 * Metodo modificador
	 *
	 * @param misNotas
	 */
	public void setMisNotas(ArrayList<Double> misNotas) {
		this.misNotas = misNotas;
	}

	/**
	 * Metodo accesor
	 *
	 * @return misAsignatura
	 */
	public Asignatura[] getMisAsignatura() {
```

Programa 68 RegistroPlanilla

```
		return misAsignatura;
	}

	/**
	 * Metodo modificador
	 *
	 * @param misAsignatura
	 */
	public void setMisAsignatura(Asignatura[] misAsignatura) {
		this.misAsignatura = misAsignatura;
	}

	/**
	 * Metodo accesor
	 *
	 * @return miE
	 */
	public Estudiante getMiE() {
		return miE;
	}

	/**
	 * Metodo modificador
	 *
	 * @param miE
	 */
	public void setMiE(Estudiante miE) {
		this.miE = miE;
	}

	/**
	 * permite calcular la definitiva
	 *
	 * @return
	 */
	public double calcularDefinitiva() {
		double prom = 0;
		for (int i = 0; i < misNotas.size(); i++) {
			prom += misNotas.get(i);
		}
		prom = prom / misNotas.size();
		return prom;
	}

	/**
	 * Informa si gano el semesntre
	 *
	 * @return true si gano, false en caso contrario
	 */
	public boolean isGanador() {
		boolean ganador = false;
		double def = calcularDefinitiva();
		if (def >= 3) {
			ganador = true;
```

Programa 68 RegistroPlanilla

```
                }
                return ganador;
        }

        /**
         * Devuelve la asignatura con mayor nota
         *
         * @return la asignatura
         */
        public Asignatura obtenerAsignaturaMayorNota() {
                Asignatura miA = null;
                double mayor = 0, notaActual;

                for (int i = 0; i < getMisNotas().size(); i++) {
                        notaActual = getMisNotas().get(i);
                        if (notaActual > mayor) {
                                mayor = notaActual;
                                miA = getMisAsignatura()[i];
                        }
                }
                return miA;
        }
}
```

El GrupoEstudiante, ver Programa 69, es la clase principal del mundo del problema. Esta clase tiene 3 atributos, a saber:

private Asignatura misAsignaturas[];
private ArrayList<Estudiante> misEstudiantes;
private ArrayList<RegistroPlanilla>misPlanillas;

El método agregarEstudiante(Estudiante miE) recibe un estudiante y lo agrega al ArrayList de estudiantes, siempre y cuando no haya sido agregado previamente, para lo cual invoca el método obtenerEstudiante. Si este método retorna -1, quiere decir que el Estudiante no existe y no ende, puede agregarse al listado.
El método crearRegistroPlanilla(String idE, ArrayList<Double>misNotas), recibe el id del estudiante y un listado de notas. El registro se agrega a misPlanillas, siempre y cuando no exista otro registro para ese estudiante. Se usa la instrucción add para agregar dicho registro.
El método obtenerAsignaturaMayorNota(String id), recibe el id del estudiante. Luego de lo cual verifica su existencia en el sistema. Si es así, obtiene la planilla correspondiente a dicho estudiante y mediante las instrucciones: miPlanilla=misPlanillas.get(pos) y miA=miPlanilla.obtenerAsignaturaMayorNota(), obtiene la asignatura con mayor nota.
El método calcularPromGeneral(), recorre todos los registros de la planilla y acumula la definitiva por estudiante en una variable. Finalmente, divide este valor por la cantidad de registros.

Programa 69 GrupoEstudiante

```
package co.uniquindio.grupo.model;
import java.util.ArrayList;
import java.util.Arrays;
import javax.swing.JOptionPane;

/**
```

Programa 69 GrupoEstudiante

```
 * Clase principal del mundo del problema
 * @author sonia
 * @author sergio
 */
public class GrupoEstudiante {
	/**
	 * Atributos de la clase
	 */
	private  Asignatura misAsignaturas[];
	private ArrayList<Estudiante> misEstudiantes;
	private ArrayList<RegistroPlanilla>misPlanillas;
	/**
	 * Metodo constructor
	 */
	public GrupoEstudiante()
	{
		Asignatura miA;
		misAsignaturas=new Asignatura[3];
		misEstudiantes=new ArrayList<Estudiante>(3);
		misPlanillas=new ArrayList<RegistroPlanilla>();
		ArrayList<String> listado=new ArrayList<String>();
		listado.add("Paradigma");
		listado.add("Introduccion");
		listado.add("Geometria");
		for(int i=0; i<3; i++)
		{miA= new Asignatura("10"+i, listado.get(i));
		misAsignaturas[i]=miA;
		}
	}
	/**
	 * Permite agregar un estudiante
	 * @param miE El estudiante
	 * @throws EstudianteException, si ya existe el estudiante
	 */
	public void agregarEstudiante(Estudiante miE) throws EstudianteException
	{
		int pos=obtenerEstudiante(miE.getCodigoEstudiante());
		if(pos==-1)
		{misEstudiantes.add(miE);
		}
		else
		{
			throw new EstudianteException("Ya existe el estudiante");
		}
	}
	/**
	 * Permite crear un nuevo registro
	 * @param idE, el id del estudiante
	 * @param misNotas, las notas
	 * @throws EstudianteException si ya existe un registro de este estudiante
	 */
	public void crearRegistroPlanilla(String idE, ArrayList<Double>misNotas)throws
EstudianteException
	{
```

Programa 69 GrupoEstudiante

```
            int pos=obtenerEstudiante(idE);
            RegistroPlanilla miPlanilla;
            Estudiante miEstudiante;
            if(pos!=-1)
            {
                  miEstudiante=misEstudiantes.get(pos);
                 miPlanilla=new RegistroPlanilla(misNotas, misAsignaturas,
miEstudiante);
                  misPlanillas.add(miPlanilla);
            }
      }
      /**
       * Permite obtener un estudiante
       * @param id El id del estudiante
       * @return la posicion donde se encuentra o -1
       */
      public int obtenerEstudiante(String id)
      {
            int pos=-1;
            boolean centinela=false;
            for(int i=0; i<misEstudiantes.size()&&centinela==false; i++)
            {
                  String id2=misEstudiantes.get(i).getCodigoEstudiante();

                  if(id.equals(id2))
                  {
                        pos=i; centinela=true;
                  }
            }
            return pos;
      }
      /**
       * Devuelve la planilla para el id correspondiente
       * @param id
       * @return la posicion dentro de la planilla
       */
      public int obtenerPlanilla(String id)
      {
            int pos=-1;
            boolean centinela=false;
            for(int i=0; i<misPlanillas.size()&&centinela==false; i++)
            {
                  Estudiante miE=misPlanillas.get(i).getMiE();

                  if(id.equals(miE.getCodigoEstudiante()))
                  {
                        pos=i; centinela=true;
                  }
            }
            return pos;
      }
      /**
       * Obtiene para el estudiante con el id solicitado a asignatura con mayor nota
       * @param id El id del estudiante
```

Programa 69 GrupoEstudiante

```
	 * @return la asignatura
	 * @throws EstudianteException
	 */
	public Asignatura obtenerAsignaturaMayorNota(String id) throws EstudianteException
	{
		Asignatura miA=null;
		int pos=obtenerPlanilla(id);
		RegistroPlanilla miPlanilla;
		double mayor=0, notaActual;
		if(pos==-1)
		{
			throw new EstudianteException("El estudiante no se encuentra en el
sistema");
		}
		if(pos!=-1)
		{
			miPlanilla=misPlanillas.get(pos);
			miA=miPlanilla.obtenerAsignaturaMayorNota();
		}
		return miA;
	}
	/**
	 * Devuelve un listado para actualizar la planilla
	 * @return un listado
	 */
	public String[][] listariInfoPlanilla()
	{
		String arreglo[][]= new String[misPlanillas.size()][5];
		RegistroPlanilla miR;
		String id, nombre, nota0, nota1, nota2;
		String salida="";
		for(int i=0; i< misPlanillas.size(); i++)
		{
			miR=misPlanillas.get(i);
			id=miR.getMiE().getCodigoEstudiante();
			nombre=miR.getMiE().getNombreEstudiante();
			nota0=""+miR.getMisNotas().get(0);
			nota1=""+miR.getMisNotas().get(1);
			nota2=""+miR.getMisNotas().get(2);
			arreglo[i][0]=id;
			arreglo[i][1]=nombre;
			arreglo[i][2]=nota0;
			arreglo[i][3]=nota1;
			arreglo[i][4]=nota2;
			salida+=id+" "+nombre+" "+nota0+" "+nota1+" "+nota2+"\n";
		}

		return arreglo;

	}
	/**
	 * Calcula el promedio general
	 * @return el promedio
	 * @throws DivisorException, si el divisor es cero
```

Programa 69 GrupoEstudiante

```
	 */
	public double calcularPromGeneral() throws DivisorException
	{
		double prom= 0;
		if(misPlanillas.size()==0)
		{
			throw new DivisorException("No hay registros en el sistema");
		}
		for(int i=0; i< misPlanillas.size(); i++)
		{
			prom+=misPlanillas.get(i).calcularDefinitiva();
		}
		prom=prom/misPlanillas.size();
		return prom;
	}
}
```

La clase Principal presente en la vista, ver Programa 70, se encarga de mostrar la vista para el estudiante y de interactuar con la clase principal del la lógica.

Programa 70 Principal

```
package co.uniquindio.grupo;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Optional;
import javax.swing.JOptionPane;
import co.uniquindio.grupo.model.RegistroPlanilla;
import co.uniquindio.grupo.model.Asignatura;
import co.uniquindio.grupo.model.DivisorException;
import co.uniquindio.grupo.model.Estudiante;
import co.uniquindio.grupo.model.EstudianteException;
import co.uniquindio.grupo.model.GrupoEstudiante;
import co.uniquindio.grupo.view.ControladorGeneral;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.TextInputDialog;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
/**
 * Ventana principal
 * @author sonia
 *
 */
public class Principal extends Application {
/**
 * Atributos de la clase
 */
	private Stage escenarioPrincipal;
```

Programa 70 Principal

```
    private BorderPane layoutRaiz;
    private GrupoEstudiante miGrupo;
    @Override
    public void start(Stage primaryStage) {
        miGrupo=new GrupoEstudiante();
        this.escenarioPrincipal = primaryStage;
        this.escenarioPrincipal.setTitle("Planilla de notas");
        inicializarLayoutRaiz();
         mostrarVistaEstudiante();
    }

/**
* Inicializa el layout raiz
 */
public void inicializarLayoutRaiz() {
        try {
            // Carga el root layout desde un archivo xml
            FXMLLoader cargador = new FXMLLoader();
            cargador.setLocation(Principal.class.getResource("view/LayoutRaiz.fxml"));
            layoutRaiz = (BorderPane) cargador.load();
            // Muestra la escena que contiene el RootLayout
            Scene scene = new Scene(layoutRaiz);
            escenarioPrincipal.setScene(scene);
            escenarioPrincipal.show();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

 /**
* Muestra la vista del estudiante
*/
    public void mostrarVistaEstudiante() {
        try {
            // Carga la vista del estudiante.
            FXMLLoader loader = new FXMLLoader();
            loader.setLocation(Principal.class.getResource("view/GeneralVista.fxml"));
            AnchorPane vistaEstudiante = (AnchorPane) loader.load();

            // Fija la vista de la person en el centro del root layout.
            layoutRaiz.setCenter(vistaEstudiante);
            // Acceso al controlador.
            ControladorGeneral miControlador = loader.getController();
            miControlador.setMiVentanaPrincipal(this);

        } catch (IOException e) {
         System.out.println(e.getMessage());
        }
    }

/**
 * Metodo accesor
 * @return  escenarioPrincipal
 */
```

Programa 70 Principal

```
    public Stage getPrimaryStage() {
        return escenarioPrincipal;
    }
/**
 * Metodo principal
 * @param args
 */
    public static void main(String[] args) {
        launch(args);
    }
 /**
  * Permite mostrar un mensaje
  * @param mensaje
  * @param miA
  * @param titulo
  * @param cabecera
  * @param contenido el mensaje
  * @param escenarioPrincipal
  */
    public static void mostrarMensaje(String mensaje, AlertType miA, String titulo,
String cabecera, String contenido, Stage escenarioPrincipal )
    {
        Alert alert = new Alert(miA);
        alert.initOwner(escenarioPrincipal);
        alert.setTitle(titulo);
        alert.setHeaderText(cabecera);
        alert.setContentText(contenido);
         alert.showAndWait();
    }
/**
 * Devuelve el escenario
 * @return escenarioPrincipal
 */
      public Stage getEscenarioPrincipal() {
            return escenarioPrincipal;
      }
/**
 * Metodo modificador
 * @param escenarioPrincipal
 */
      public void setEscenarioPrincipal(Stage escenarioPrincipal) {
            this.escenarioPrincipal = escenarioPrincipal;
      }
/**
 * Permite crear un registro
 * @param idE El id del estudiante
 * @param misNotas
 * @throws EstudianteException Si el estudiante ya tiene un registro
 */
      public void crearRegistroPlanilla(String idE, ArrayList<Double>misNotas) throws
EstudianteException
      {
            miGrupo.crearRegistroPlanilla(idE, misNotas);
      }
```

Programa 70 Principal

```
/**
 * Permite agregar un estudiante
 * @param miE El estudiante
 * @throws EstudianteException Si ya existe el estudiante
 */
      public void agregarEstudiante(Estudiante miE) throws EstudianteException
      {
            miGrupo.agregarEstudiante(miE);
      }
/**
 * Devuelve la informacion de la planilla
 * @return el listado
 */
      public String[][] listariInfoPlanilla()
      {
            return miGrupo.listariInfoPlanilla();
      }
/**
 * Devuelve la asignatura con mayor nota
 * @param id El id del estudiante
 * @return una asignatura
 * @throws EstudianteException
 */
      public Asignatura obtenerAsignaturaMayorNota(String id) throws EstudianteException
      {
            return miGrupo.obtenerAsignaturaMayorNota(id);
      }
/**
 * Calcula el promedio general
 * @return el promedio
 * @throws DivisorException
 */
      public double calcularPromGeneral() throws DivisorException
      {
            return miGrupo.calcularPromGeneral();
      }
/**
 * Permite leer un entero
 * @param mensaje
 * @return
 */
      public static int leerEntero(String mensaje)
    {
      return Integer.parseInt(leerMensaje(mensaje ));
    }
/**
 * Permite leer un mensaje
 * @param mensaje
 * @return el  texto leido
 */
       public static String leerMensaje(String mensaje )
          {
            String salida="";
             // Muestra el mensaje
```

Programa 70 Principal

```
            TextInputDialog miDialogo = new TextInputDialog("");
            miDialogo.setTitle("Ingreso de datos");
            miDialogo.setHeaderText("");
            miDialogo.setContentText(mensaje);

            // Forma tradicional de obtener la resuesta.
            Optional<String> result = miDialogo.showAndWait();
            if (result.isPresent()){
                salida= result.get();
            }
          return salida;
          }
}
```

La clase ControladorGeneral, ver Programa 71 , se encarga de ir actualizando el tableView a medida que se agregan datos al sistema. La actualización de la tabla se efectúa mediante un vector de Strings llamado data.
El método initialize(URL location, ResourceBundle resources) es el encargado de especificar con que valores se va a actualizar la tabla y de construir las columnas de la misma.
El método crearRegistro, se encarga de crear un nuevo registro para la planilla y de mantener el TableView actualizado, las instrucciones encargadas de ello son:

```
miVentanaPrincipal.crearRegistroPlanilla(id, notas);
tv.getItems().clear();
//Se actualiza data con la informacion de la logica
String data[][]=miVentanaPrincipal.listariInfoPlanilla();
this.data=data;
tv.getItems().addAll(Arrays.asList(data));
```

Programa 71 ControladorGeneral

```
package co.uniquindio.grupo.view;
import java.net.URL;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.ResourceBundle;
import co.uniquindio.grupo.Principal;
import co.uniquindio.grupo.model.Asignatura;
import co.uniquindio.grupo.model.DivisorException;
import co.uniquindio.grupo.model.Estudiante;
import co.uniquindio.grupo.model.EstudianteException;
import co.uniquindio.grupo.model.NotaIncorrectaException;
import co.uniquindio.grupo.model.RegistroPlanilla;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Alert;
```

Programa 71 ControladorGeneral

```
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.stage.StageStyle;
import javafx.util.Callback;


/**
 *Permite editar los datos de el estudiante
 *
 * @author Sonia Jaramillo
 */
public class ControladorGeneral implements Initializable {


    //VentanaPrincipal
    private Principal miVentanaPrincipal;
    String[][] data = new String[1][5];
    @FXML
    private TextField idTextField;
    @FXML
    private TextField nombreTextField;
    @FXML
    private TextField nota0TextField;
    @FXML
    private TextField nota1TextField;
    @FXML
    private TextField nota2TextField;

    @FXML
    private TableView<String[]> tv;
    @FXML
    public TableColumn<String[],String> id;
    @FXML
    public TableColumn<String[],String>nombre;
    @FXML
    public TableColumn<String[],String> nota1;
    @FXML
    public TableColumn<String[],String> nota2;
    @FXML
    public TableColumn<String[],String> nota3;


    private String style;
    @Override
    public void initialize(URL location, ResourceBundle resources) {
       id.setCellValueFactory(new Callback<TableColumn.CellDataFeatures<String[],
String>, ObservableValue<String>>() {
```

Programa 71 ControladorGeneral

```
            @Override
            public ObservableValue<String> call(TableColumn.CellDataFeatures<String[],
String> p) {
                String[] x = p.getValue();
                if (x != null && x.length>0) {
                    return new SimpleStringProperty(x[0]);
                } else {
                    return new SimpleStringProperty("<no name>");
                }
            }
        });

      nombre.setCellValueFactory(new Callback<TableColumn.CellDataFeatures<String[],
String>, ObservableValue<String>>() {
            @Override
            public ObservableValue<String> call(TableColumn.CellDataFeatures<String[],
String> p) {
                String[] x = p.getValue();
                if (x != null && x.length>1) {
                    return new SimpleStringProperty(x[1]);
                } else {
                    return new SimpleStringProperty("<no value>");
                }
            }
        });
      nota1.setCellValueFactory(new Callback<TableColumn.CellDataFeatures<String[],
String>, ObservableValue<String>>() {
          @Override
          public ObservableValue<String> call(TableColumn.CellDataFeatures<String[],
String> p) {
              String[] x = p.getValue();
              if (x != null && x.length>0) {
                  return new SimpleStringProperty(x[2]);
              } else {
                  return new SimpleStringProperty("<no name>");
              }
          }
      });

     nota2.setCellValueFactory(new Callback<TableColumn.CellDataFeatures<String[],
String>, ObservableValue<String>>() {
          @Override
          public ObservableValue<String> call(TableColumn.CellDataFeatures<String[],
String> p) {
              String[] x = p.getValue();
              if (x != null && x.length>1) {
                  return new SimpleStringProperty(x[3]);
              } else {
                  return new SimpleStringProperty("<no value>");
              }
          }
      });
    nota3.setCellValueFactory(new Callback<TableColumn.CellDataFeatures<String[],
String>, ObservableValue<String>>() {
```

Programa 71 ControladorGeneral

```
          @Override
          public ObservableValue<String> call(TableColumn.CellDataFeatures<String[],
String> p) {
              String[] x = p.getValue();
              if (x != null && x.length>0) {
                  return new SimpleStringProperty(x[4]);
              } else {
                  return new SimpleStringProperty("<no name>");
              }
          }
      });


        //Adicionar los datos a la tabla aqui

  // Add Data
     tv.getItems().addAll(Arrays.asList(data));
    }

    /**
     * Metodo modificador
     *
     * @param dialogo
     */


    public Principal getMiVentanaPrincipal() {
            return miVentanaPrincipal;
      }

      public void setMiVentanaPrincipal(Principal miVentanaPrincipal) {
            this.miVentanaPrincipal = miVentanaPrincipal;
      }


    /**
     * Metodo para modificar el estudiante
     */
    @FXML
    private void crearRegistro() {
   try {
   String id=idTextField.getText();
   String nombre=nombreTextField.getText();
   double nota0=Double.parseDouble(nota0TextField.getText());
   double nota1=Double.parseDouble(nota1TextField.getText());
   double nota2=Double.parseDouble(nota2TextField.getText());
   validarNota( nota0);
   validarNota( nota1);
   validarNota( nota2);
   Estudiante miE=new Estudiante(id, nombre);
   miVentanaPrincipal.agregarEstudiante(miE);
   ArrayList<Double> notas=new ArrayList<Double>();
   notas.add(nota0);
   notas.add(nota1);
```

Programa 71 ControladorGeneral

```
    notas.add(nota2);

        miVentanaPrincipal.crearRegistroPlanilla(id, notas);
        tv.getItems().clear();
        String data[][]=miVentanaPrincipal.listariInfoPlanilla();
        this.data=data;
          tv.getItems().addAll(Arrays.asList(data));
} catch (EstudianteException | NotaIncorrectaException e) {
        // TODO Auto-generated catch block
        miVentanaPrincipal.mostrarMensaje("Error: ", AlertType.ERROR, "",
"",e.getMessage(), miVentanaPrincipal.getPrimaryStage() );

}
 }
  /**
   * Permite validar los valores ingresados en la nota
   * @param nota0
   * @throws NotaIncorrectaException si se sale de 0-5
   */
   public void validarNota(double nota0) throws NotaIncorrectaException
   {
          if(!(nota0>=0&&nota0<=5))
          {
                throw new NotaIncorrectaException("Nota incorrecta");
          }

   }
 /**
  * Obtiene la asignatura de mayor nota
  */
   public void obtenerAsignaturaMayorNota()
       {
          String id=miVentanaPrincipal.leerMensaje("Digite el id del estudiante");
          Asignatura miA;
       try {
              miA = miVentanaPrincipal.obtenerAsignaturaMayorNota(id);
              Principal.mostrarMensaje(" ", AlertType.INFORMATION, "", "La asignatura con
mejor nota es: ",miA.getNombre(), miVentanaPrincipal.getPrimaryStage() );

       } catch (EstudianteException e) {
              // TODO Auto-generated catch block
              Principal.mostrarMensaje(" ", AlertType.ERROR, "", "","+e.getMessage(),
miVentanaPrincipal.getPrimaryStage() );
              }

       }
  /**
   * Calcula el promedio general
   */
   public void calcularPromGeneral()
       {
          double prom;
       try {
              prom = miVentanaPrincipal.calcularPromGeneral();
```

Programa 71 ControladorGeneral

```
             Principal.mostrarMensaje(" ", AlertType.INFORMATION, "", "","El promedio
general es: "+prom, miVentanaPrincipal.getPrimaryStage() );

      } catch (DivisorException e) {
            // TODO Auto-generated catch block
            Principal.mostrarMensaje(" ", AlertType.ERROR, "", "",""+e.getMessage(),
miVentanaPrincipal.getPrimaryStage() );

      }
      }

}
```

VistaGeneral.fxml, implementada en Programa 72, permite la construcción de la interfaz presente ***Figura 105***.

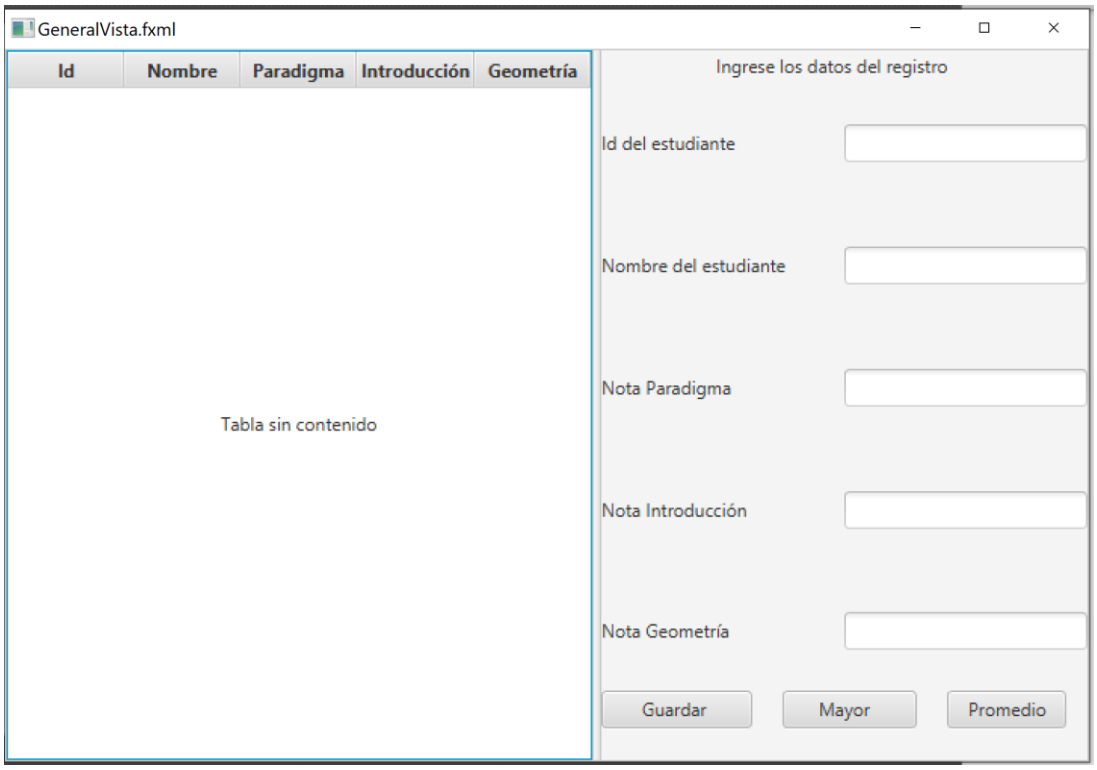


Figura 105 Interfaz de la aplicación para crear una planilla de notas

Programa 72 VistaGeneral.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.shape.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
```

Programa 72 VistaGeneral.fxml

```
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-
Infinity" prefHeight="472.0" prefWidth="718.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.grupo.view.ControladorGeneral">
   <children>
      <SplitPane dividerPositions="0.5454545454545454" layoutX="29.0" layoutY="46.0"
prefHeight="399.0" prefWidth="630.0" AnchorPane.bottomAnchor="0.0"
AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
        <items>
          <AnchorPane minHeight="0.0" minWidth="0.0" prefHeight="397.0"
prefWidth="334.0">
               <children>
                  <TableView fx:id="tv" prefHeight="397.0" prefWidth="340.0"
AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0"
AnchorPane.topAnchor="0.0">
                    <columns>
                      <TableColumn fx:id="id" prefWidth="75.0" text="Id" />
                      <TableColumn fx:id="nombre" prefWidth="75.0" text="Nombre" />
                        <TableColumn fx:id="nota1" prefWidth="75.0" text="Paradigma" />
                        <TableColumn fx:id="nota2" prefWidth="75.0" text="Introducción"
/>
                        <TableColumn fx:id="nota3" prefWidth="75.0" text="Geometría" />
                    </columns>
                     <columnResizePolicy>
                        <TableView fx:constant="CONSTRAINED_RESIZE_POLICY" />
                     </columnResizePolicy>
                  </TableView>
               </children>
            </AnchorPane>
          <AnchorPane minHeight="0.0" minWidth="0.0" prefHeight="397.0"
prefWidth="255.0">
               <children>
                  <BorderPane layoutX="7.0" layoutY="14.0" prefHeight="397.0"
prefWidth="282.0" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0"
AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
                     <center>
                        <GridPane BorderPane.alignment="CENTER">
                          <columnConstraints>
                            <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0"
prefWidth="100.0" />
                            <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0"
prefWidth="100.0" />
                          </columnConstraints>
                          <rowConstraints>
                            <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                            <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                            <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
```

Programa 72 VistaGeneral.fxml

```
                                 <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                                 <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                              </rowConstraints>
                              <children>
                                 <Label text="Id del estudiante" />
                                 <Label text="Nombre del estudiante" GridPane.rowIndex="1"
/>
                                 <Label text="Nota Paradigma" GridPane.rowIndex="2" />
                                 <Label text="Nota Introducción" GridPane.rowIndex="3" />
                                 <Label text="Nota Geometría" GridPane.rowIndex="4" />
                                 <TextField fx:id="idTextField" GridPane.columnIndex="1" />
                                 <TextField fx:id="nombreTextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
                                 <TextField fx:id="nota0TextField" GridPane.columnIndex="1"
GridPane.rowIndex="2" />
                                 <TextField fx:id="nota1TextField" GridPane.columnIndex="1"
GridPane.rowIndex="3" />
                                 <TextField fx:id="nota2TextField" GridPane.columnIndex="1"
GridPane.rowIndex="4" />
                              </children>
                           </GridPane>
                        </center>
                        <top>
                           <Label prefHeight="21.0" prefWidth="171.0" text="Ingrese los
datos del registro" BorderPane.alignment="CENTER" />
                        </top>
                        <bottom>
                           <HBox prefHeight="46.0" prefWidth="284.0" spacing="20.0"
BorderPane.alignment="CENTER">
                              <children>
                                 <Button mnemonicParsing="false" onAction="#crearRegistro"
prefHeight="25.0" prefWidth="100.0" text="Guardar " />
                                 <Button mnemonicParsing="false"
onAction="#obtenerAsignaturaMayorNota" prefHeight="25.0" prefWidth="89.0" text="Mayor  "
/>
                                 <Button mnemonicParsing="false"
onAction="#calcularPromGeneral" prefHeight="25.0" prefWidth="79.0" text="Promedio" />
                              </children>
                           </HBox>
                        </bottom>
                     </BorderPane>
                  </children>
               </AnchorPane>
            </items>
         </SplitPane>
      </children>
</AnchorPane>
```

El Programa 73 permite la construcción del Layout Raiz que se presenta en la *Figura 106.*

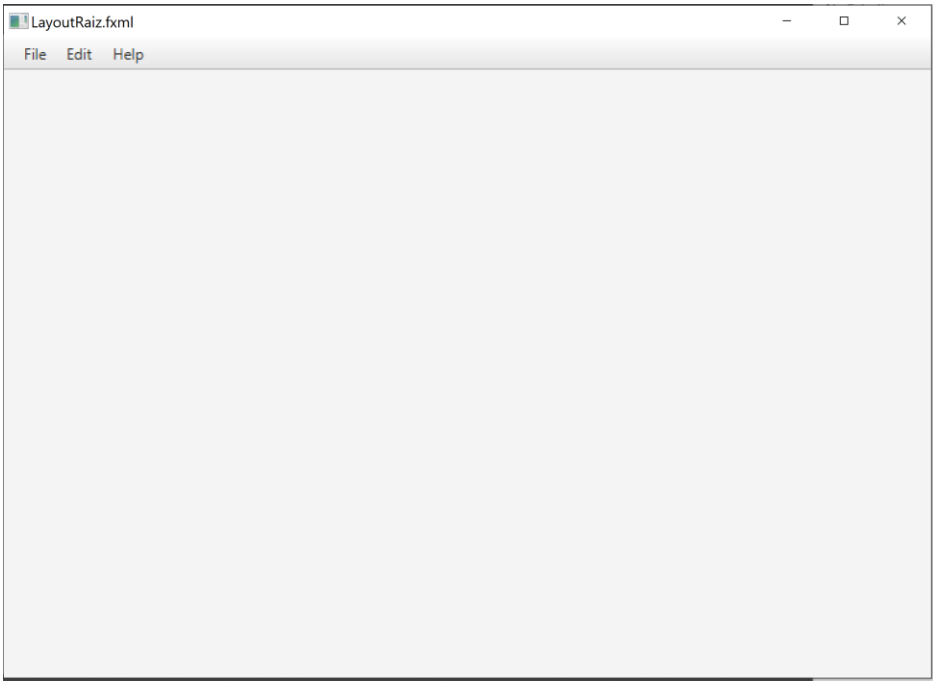


Figura 106 Layout Raiz

Programa 73 LayoutRaiz.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="510.0" prefWidth="740.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1">
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem mnemonicParsing="false" text="Close" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Edit">
            <items>
              <MenuItem mnemonicParsing="false" text="Delete" />
            </items>
          </Menu>
          <Menu mnemonicParsing="false" text="Help">
            <items>
```

Programa 73 LayoutRaiz.fxml

```
                <MenuItem mnemonicParsing="false" text="About" />
              </items>
            </Menu>
          </menus>
        </MenuBar>
    </top>
</BorderPane>
```

3.13 Arreglos bidimensionales

Un arreglo en Java es una estructura que permite agrupar elementos del mismo tipo. Los arreglos pueden tener una o varias dimensiones. En este último caso se les denomina arreglos multidimensionales.

Para hacer uso de un arreglo es necesario seguir los siguientes pasos:
- Declarar
- Reservar memoria
- Inicializar
- Utilizar

Es importante anotar, que en Java los arreglos multidimensionales son considerados como arreglos de arreglos.

La sintaxis para declarar esta estructura de este tipo es la siguiente:

Tipo [ ][ ]… [ ] nombre = new Tipo [dimensión1][dimensión2]…[dimensiónN];

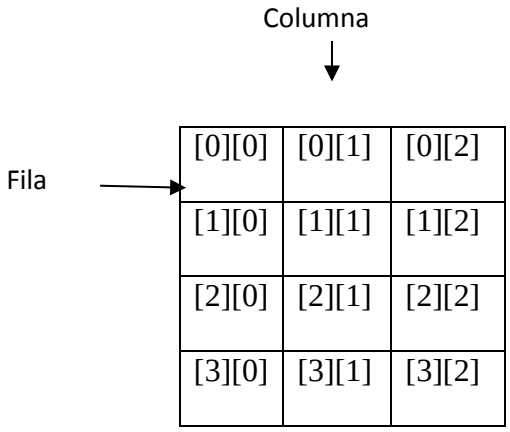
Como forma alternativa se puede utilizar la declaración:

Tipo [ ][ ] … [ ] nombre;
nombre = new Tipo [dimensión1][dimensión2]…[dimensiónN];

En “Tipo” puede ir cualquier tipo de dato primitivo. Además se permiten String y variables tipo Clase. A continuación ilustra el proceso de declaración e inicialización de un arreglo en dos dimensiones. Se resalta que los arreglos bidimensionales son conocidos también con el nombre de matrices.

```
int[][] edad = new int[4][3];
```

En la línea anterior se puede observar que para reservar memoria se usó el operador new [2][3][4]. Además, que el arreglo tiene 4 filas y 3 columnas. Una vista conceptual de esta matriz, para poder desarrollar la lógica, es la siguiente:



En la memoria una matriz no se almacena de esta forma, pero esta visualización contribuye a una mejor comprensión del tema.

Dentro del arreglo bidimensional cada elemento tiene una posición ([0][0], [0][1], [0][2], [1][0]…[3][2]), compuesta por una fila y una columna. Partiendo de lo anterior, se puede concluir que para recorrer estas tipo de estructura se necesitan 2 índices. El primero, al que se denominará i, permite desplazarse por las filas. El segundo, llamado j, moverse por las columnas.

Los arreglos también pueden ser inicializados al momento de su declaración:

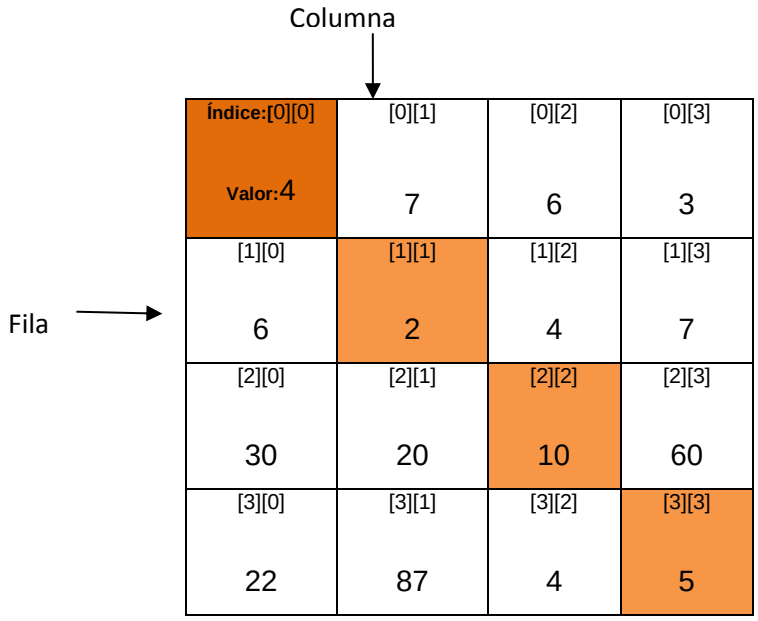
```
int edad[][]={{6,7,9},{3,4,6}};
```

Si esta inicialización no se efectúa, Java automáticamente llenará cada una de las posiciones del arreglo basándose en el tipo de dato, por ejemplo si el arreglo es numérico, por defecto se pondrán ceros, si es boolean false y si es una clase, se inicializa con null.
Si se requiere conocer el tamaño de la estructura bidimensional es necesario hacer uso de la propiedad length. Para el caso de las filas se utiliza edad.length y para las columnas edad[i].length.

A continuación se presenta una matriz de 4X4, correspondiente a la siguiente inicialización:

int edad[][]={{4,7,6,3},{6,2,4,7},{30,20,10,60},{22,87,4,5}};

En cada una de las celdas se ha puesto, en la parte superior el respectivo índice y en la inferior, el valor almacenado. Como se dijo con anterioridad el indice está formado por una fila, i, y una columna, j.



Se puede observar que en la posición [1][0] hay un 6 almacenado, en la [3][1] un 87. Es importante diferenciar la posición, de lo que se encuentra almacenado en la misma.

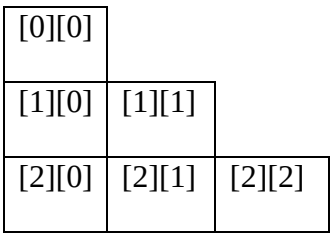
Los índices se ponen en cada una de las casillas para facilitar la identificación de aspectos que permitan generalizar, y por ende, desarrollar la algorítmica. Por ejemplo, si se desea promediar todos los elementos de la diagonal basta seleccionar aquellos ubicados en las posiciones en donde la i y la j son iguales: [0][0], [1][1], [2][2], ..

if(i==j)

Por el contrario, si lo que se desea es promediar los elementos de la primera fila, que corresponden a las posiciones [0][0], [0][1],[0][2],[0][3]: Se debería poner:

if(i==0)

En algunos casos puede ser necesario reservar diferentes tamaños para la segunda dimensión de cada elemento. Ejemplo:



Esto se logra declarando primero la cantidad de filas y luego asignado tamaño a cada una de las columnas.

```
int datos[][]=new int[3];
datos [0]=new int [1];
datos [1]=new int [2];
datos [2]=new int [3];
```

Si se desea recorrer una matriz será necesario el uso de dos ciclos for:

```
for(int i=0; i< datos.length;i++)
{
   for(int j=0; j< datos [i].length; j++)
   {

   }
 }
```

Para ilustrar el recorrido sobre arreglos bidimensionales se construirán una serie de métodos. El primero de ellos es `sumarElementos`, el cual suma los datos que se encuentran en la diagonal principal de la matriz. Este método tiene como parámetro un arreglo bidimensional. En su interior, se declara un acumulador que inicia en cero. Mediante los 2 ciclos for se recorre una a una de las posiciones de la matriz, preguntando si se está ubicado en la diagonal principal. Si es así, entonces el acumulador se incrementa en el valor que se encuentra en dicha posición.

```
public double sumarElementos (Persona  misPersonas[][])
{
double suma=0;
for(int i=0; i< misPersonas.length;i++)
{
   for(int j=0; j< misPersonas [i].length; j++)
   {
      if(i==j)
      {
          suma+= misPersonas [i][j].getEdad();
      }
   }
}
return suma;
}
```

El siguiente método genera la transpuesta de una matriz. En la matriz transpuesta el elemento a_{ji} de la matriz origen se convierte en el elemento a_{ij}. Ejemplo[10]:

$$A = \begin{pmatrix} 2 & 3 & 0 \\ 1 & 2 & 0 \\ 3 & 5 & 6 \end{pmatrix} \qquad A^t = \begin{pmatrix} 2 & 1 & 3 \\ 3 & 2 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$

En `hallarTranspuesta` se recibe como parámetro a la matriz original. Los dos for permiten recorrer esta matriz. Observe que mediante la instrucción transpuesta[j][i]=matriz[i][j] los elementos se guardan en transpuesta en la posición invertida.

[10] http://www.ditutor.com/matrices/matriz_traspuesta.html

```
public Persona[][] hallarTranspuesta(Persona matriz[][])
{
Persona transpuesta[][]=new int[matriz[0].length][matriz.length];
for(int i=0; i<matriz.length;i++)
 {
   for(int j=0; j<matriz[0].length; j++)
   {
    transpuesta[j][i]=matriz[i][j];
   }
 }
return transpuesta;
}
```

Por último, se tiene el método contarPalindromas, que recorre un arreglo bidimensional y halla la cantidad de palabras palíndromas. Una palíndroma se lee igual de izquierda a derecha, que de derecha a izquierda. Ejemplo: oso.

Para resolver este problema se ha construido previamente el método isPalindroma, que recibe una palabra y devuelve true en caso de que sea palíndroma. Para poder determinar dicha situación se construye un for cuyo límite va hasta la mitad de la palabra. Cuando la variable i toma el valor de cero compara la letra que está en dicha posición con la última letra de la cadena. Luego la letra que está en la posición uno con la penúltima y así sucesivamente. Si alguna de estas comparaciones falla es porque la palabra no es palíndroma.

```
public boolean isPalindoma(String palabra)
{
boolean centinela=true;
for(int i=0; i<palabra.length()/2;i++)
{
  if(palabra.charAt(i)!=palabra.charAt(palabra.length()-1-i))
  {
              return false;
  }
}
return centinela;
}
```

Por último, se procede a construir contarPalindromas. Este método recorre una a una las posiciones de la estructura bidimensional preguntando si la palabra allí contenida es palíndroma. Si es así el contador se incrementa en 1.

```
public int contarPalindromas(String palabras[][])
{
int contador=0;
for(int i=0; i<palabras.length;i++)
 {
   for(int j=0; j<palabras[0].length; j++)
   {
     if(isPalindoma(palabras[i][j]))
     {
      contador++;
     }
    }
}
return contador;
}
```

3.14 Propiedades JavaFX

JavaFX hace uso de propiedades. Estas pueden ser vistas como una clase tipo wraper, es decir, un envoltorio para encapturar los campos y adicionales propiedades, tales como informar si hay cambios, por ejemplo para actualizar componente gráficos. Existen propiedades para cada tipo de dato, String, Float, Double, Integer. Por cada una de ellas se tiene su correspondiente envoltorio, StringProperty, FloatProperty, DoubleProperty, IntegerProperty. Si se desea crear un envoltorio para un objeto tipo clase se hace uso de ObjectProperty<T> [13][14][15].

Para crear propiedades de lectura/escritura, se hace uso de métodos tales como SimpleIntegerProperty, SimpleObjectProperty, SimpleDoubleProperty, SimpleFloatProperty, entre otros. Las propiedades tiene implementaciones para los métodos set() y setValue() [16]. La siguiente porción de código ilustra el uso de propiedades dentro de la clase de la lógica VehiculoModel. Observe también el código correspondiente al controlador. Se declara **private final** ObservableList<VehiculoModel> vehiculoModels = FXCollections.*observableArrayList*( ), el cual permitirá mantener actualizado un TableView. Otras variables que se declaran son las siguientes:

```
@FXML
private TableView<VehiculoModel> tbData;
@FXML
public TableColumn<VehiculoModel, String> placa;
@FXML
public TableColumn<VehiculoModel, String> tipo;
@FXML
public TableColumn<VehiculoModel, Integer> modelo;
```

La actualización se logra mediante el método actualizarTabla().

El método initialize(URL location, ResourceBundle resources) permite especificar los lambda para poder actualizar los campos del TableView.

```
public void initialize(URL location, ResourceBundle resources) {
        tipoComboBox.getItems().removeAll(tipoComboBox.getItems());
        tipoComboBox.getItems().addAll("Carro", "Moto");
        tipoComboBox.getSelectionModel().select("Carro");
        placa.setCellValueFactory(new PropertyValueFactory<>("Placa"));
        tipo.setCellValueFactory(new PropertyValueFactory<>("Tipo"));
        modelo.setCellValueFactory(new PropertyValueFactory<>("Modelo"));
        tbData.setItems(vehiculoModels);
    }
```

```
public class VehiculoModel {

    private final SimpleStringProperty placa;
    private final SimpleIntegerProperty tipo;
    private  final SimpleIntegerProperty modelo;
    private  final ObjectProperty<Propietario> duenio;

    public VehiculoModel(String placa, int tipo, int modelo, Propietario miP) {
        this.placa = new SimpleStringProperty(placa);
        this.tipo = new SimpleIntegerProperty(tipo);
        this.modelo = new SimpleIntegerProperty(modelo);
        this.duenio=new SimpleObjectProperty<Propietario>(miP);
```

```
    }

    public String getPlaca() {
        return placa.get();
    }

    public void setPlaca(String placa) {
        this.placa.set(placa);
    }
    public void setPropietario(Propietario miP) {
        this.duenio.set(miP);
    }
    public Propietario getPropietario() {
        return duenio.get();
    }
    public String getTipo() {
        String tipo="Carro";
        if(this.tipo.get()==1)
        {tipo="Moto";};

       return tipo;
    }
    public int getTipoInt() {
        return tipo.get();
    }
    public void setTipo(int tipo) {
        this.tipo.set(tipo);
    }

    public int getModelo() {
        return modelo.get();
    }

    public void setModelo(int modelo) {
        this.modelo.set(modelo);
    }
}
```

```
public class VehiculoVistaController implements Initializable{
     @FXML
    private TableView<VehiculoModel> tbData;
    @FXML
    public TableColumn<VehiculoModel, String> placa;
    @FXML
    public TableColumn<VehiculoModel, String> tipo;
    @FXML
    public TableColumn<VehiculoModel, Integer> modelo;
    private  final ObservableList<VehiculoModel> vehiculoModels =
FXCollections.observableArrayList( );

    private String style;
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        tipoComboBox.getItems().removeAll(tipoComboBox.getItems());
        tipoComboBox.getItems().addAll("Carro", "Moto");
        tipoComboBox.getSelectionModel().select("Carro");
```

```
        placa.setCellValueFactory(new PropertyValueFactory<>("Placa"));
        tipo.setCellValueFactory(new PropertyValueFactory<>("Tipo"));
        modelo.setCellValueFactory(new PropertyValueFactory<>("Modelo"));
       tbData.setItems(vehiculoModels);
    }
@FXML
private Posicion handleNewVehiculo() {
       Posicion miP=new Posicion();
       Propietario miProp;
       VehiculoModel miV;
       String placa =placaTextField.getText();
       String id= propietarioIdTextField.getText();
       String nombre= propietarioNombreTextField.getText();
       int modelo= Integer.parseInt(modeloTextFieldl.getText());
       int tipo=tipoComboBox.getSelectionModel().getSelectedIndex();
       miProp=new Propietario(nombre, id);
       miV=new VehiculoModel(placa, tipo, modelo, miProp);
       miP=mainApp.agregarVehiculo(miV);
      style = mainApp.getMatrizBotones()[miP.getI()][miP.getJ()].getStyle();
       mainApp.getMatrizBotones()[miP.getI()][miP.getJ()].setStyle("-fx-background-color:
red;");
           actualizarTabla();
        mainApp.mostrarMensaje("Agregado  ", AlertType.INFORMATION, "", "","El vehiculo
se ubico correctamente en el parqueadero", mainApp.getPrimaryStage() );
       return miP;
    }
public void actualizarTabla()
    {
           tbData.getItems().clear();
           if(mainApp!=null)
           {   for(int i=0; i<mainApp.getMisVehiculos().size();i++)
             vehiculoModels.add(mainApp.getMisVehiculos().get(i));
           }
    }
}
```

3.15 Caso de estudio 4 Unidad III: Parqueadero

Se desea crear una aplicación para la administración de un parqueadero. El parqueadero está formado por 30 puestos. Los 12 primeros destinados a motos y los restantes a carros. Cada puesto tiene un número, un tipo (0→ Vehículo Particular, 1→ Moto) y un estado (0→ libre 1→ Ocupado). Para poder asignar un espacio dentro del parqueadero cuando un vehículo llega su propietario debe informar la placa del vehículo y el modelo, por su parte el administrador del sistema debe registrar la hora y fecha de ingreso manualmente o seleccionar la fecha actual del sistema.

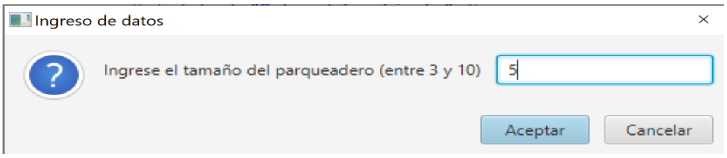


Figura 107 Interfaz para solicitar el tamaño del parqueadero

Figura 108 Interfaz del parqueadero

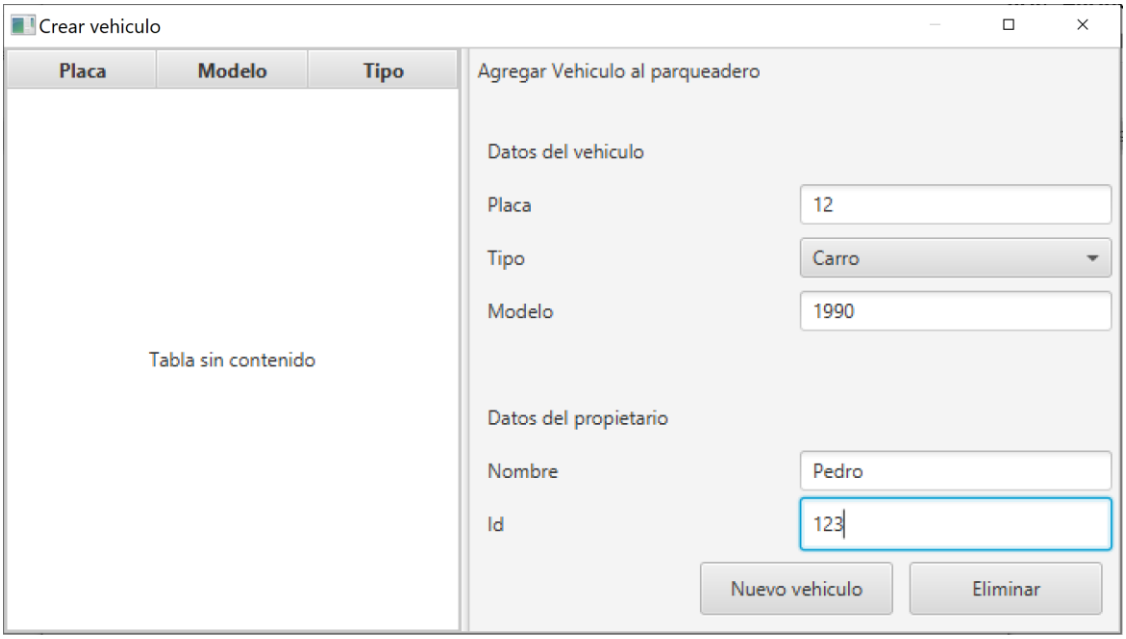


Figura 109 Interfaz del parqueadero para ingresar un nuevo vehiculo

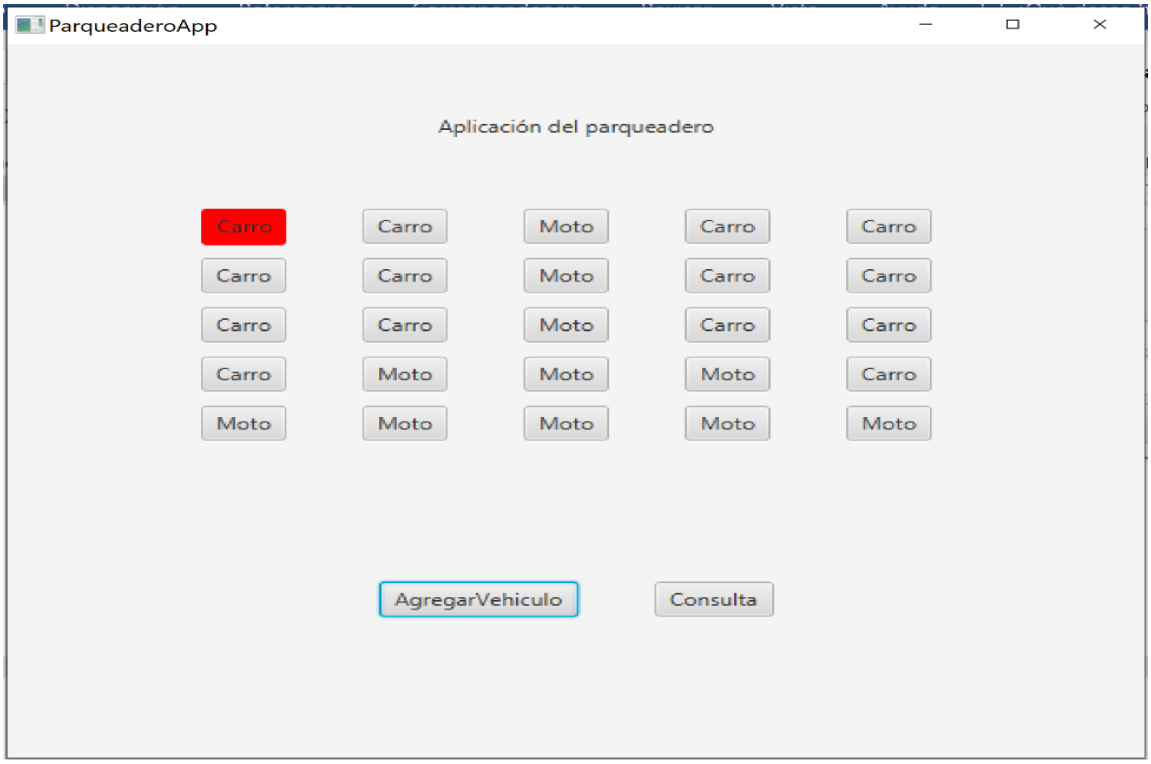


Figura 110 Interfaz del parqueadero luego de ingresar un vehiculo

3.15.1 Comprensión del problema

a) Requisitos funcionales

NOMBRE	R1 – Crear un propietario
RESUMEN	Permite agregar un nuevo usuario al parqueadero
ENTRADAS	
Nombre, id	
RESULTADOS	
Un nuevo propietario se registra en el parqueadero	

NOMBRE	R2 –.Crear un vehiculo
RESUMEN	Permite crear un nuevo vehiculo
ENTRADAS	
Placa, tipo, model, propietario	
RESULTADOS	
Un nuevo vehiculo se agrega al parqueadero	

NOMBRE	R3 – Crear un nuevo registro
RESUMEN	
ENTRADAS	
El vehiculo y el puesto que se le asigna. El puesto lo obtiene el sistema.	
RESULTADOS	
Un nuevo registro ha sido creado	

NOMBRE	R4 – Liberar el puesto ocupado por un vehiculo en el parqueadero
RESUMEN	
ENTRADAS	
La posición del vehiculo dentro del listado	
RESULTADOS	
El puesto ha sido liberaro	

b) El modelo del mundo del problema

Las actividades que se deben realizar para construir el modelo del mundo son:

Identificar las entidades o clases.

ENTIDAD DEL MUNDO	DESCRIPCIÓN
Parqueadero	Es la entidad más importante del mundo del problema.
Propietario	Es un atributo del parqueadero y del vehiculo
Puesto	Es un atributo del Parqueadero
VehículoModel	Es un atributo del registro y del parqueadero
Posicion	Es un atributo del puesto

El diagrama de clases correspondiente a este caso de estudio se presentan a continuación:

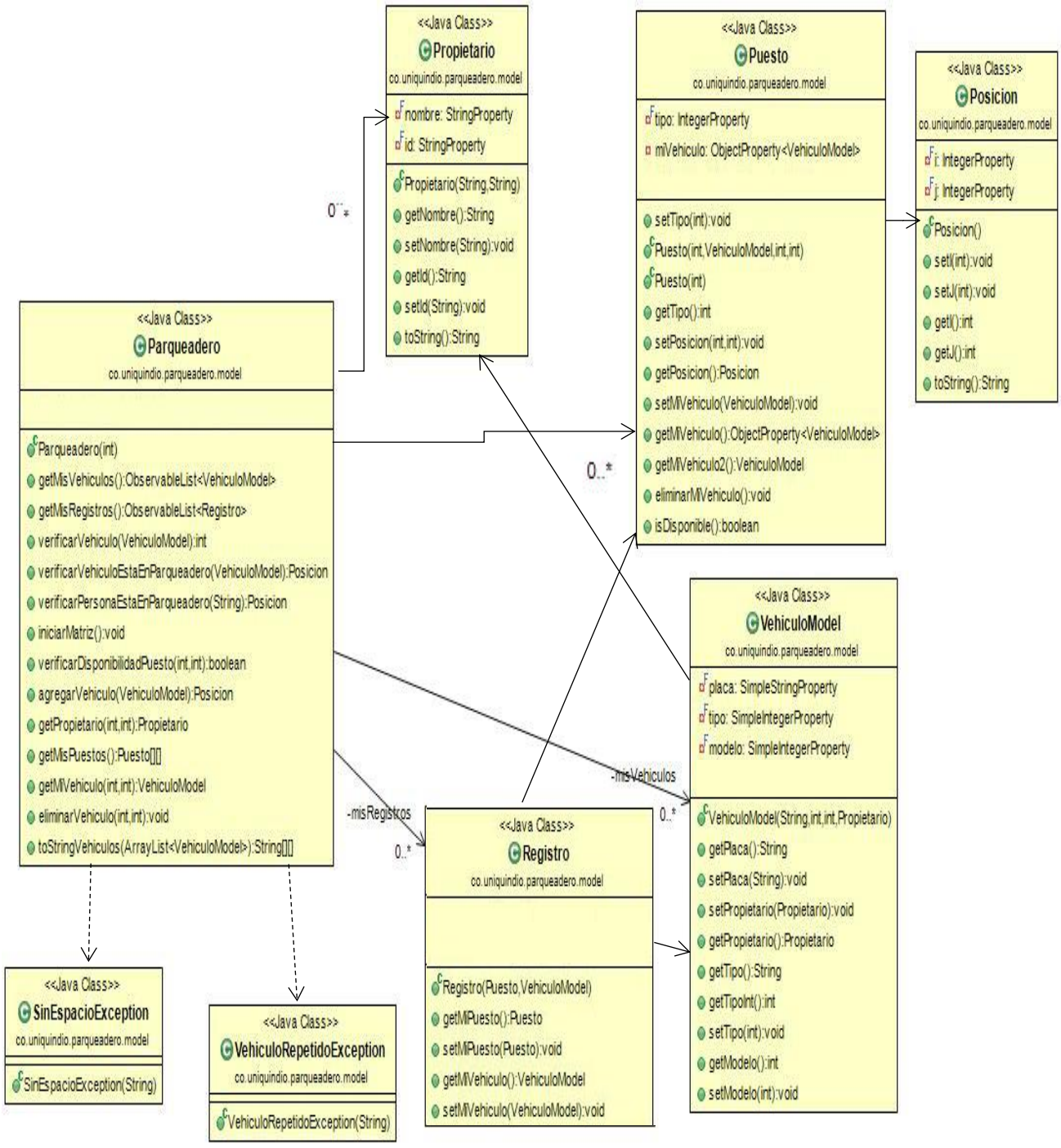


Figura 111 Diagrama de clases del parqueadero

La primera clase a construir es la clase Posicion, ver Programa 74. Esta clase tiene dos atributos i y j, ambos de tipo IntegerProperty. El metodo constructor permite inicializar las 2 variables de la clase.

Programa 74 Posicion

```
package co.uniquindio.parqueadero.model;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;

/**
 * Clase para representar una posicion
 *
 * @author sonia
 * @author sergio
 */
public class Posicion {
	/**
	 * Atributos de la clase
	 */
	private final IntegerProperty i, j;

	/**
	 * Constructor de la clase
	 */
	public Posicion() {
		this.i = new SimpleIntegerProperty();
		this.j = new SimpleIntegerProperty();
	}

	/**
	 * Metodo modificador
	 *
	 * @param i
	 */
	public void setI(int i) {
		this.i.set(i);
	}

	/**
	 * Metodo modificador
	 *
	 * @param j
	 */
	public void setJ(int j) {
		this.j.set(j);
	}

	/**
	 * Metodo accesor
	 *
	 * @return i.get()
	 */
	public int getI() {
```

Programa 74 Posicion

```
		return i.get();
	}

	/**
	 * Metodo accesor
	 *
	 * @return j.get()
	 */
	public int getJ() {
		return j.get();
	}

	/**
	 * Devuelve la representacion en string de la posicion
	 */
	public String toString() {
		return "(" + i + ", " + j + ")";
	}
}
```

La clase propietario permite representar un nuevo propietario. El propietario tiene los siguientes atributos:

```
private final StringProperty   nombre;
private final StringProperty id;
```

El constructor de propietario permite inicializar las variables de la clase, las instrucciones usadas para ello se presentan a continuación:

```
public Propietario(String nombre, String id)
   {
          this.nombre=new SimpleStringProperty();
          this.nombre.set(nombre);
          this.id=new SimpleStringProperty();
          this.id.set(id);
   }
```

Observe que los método get devuelven el valor de la propiedad:

```
/**
 * Metodo accesor
 * @return El nombre del propietario
 */
public String getNombre() {
      return nombre.get();
}
```

Programa 75 Propietario

```
package co.uniquindio.parqueadero.model;


import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

/**
```

Programa 75 Propietario

```
 * Clase que representa un Propietario
 * @author sonia
 * @author sergio
 */
public class Propietario
{
    //-------------------------------------------------------------------------
      //ATRIBUTOS
      //-------------------------------------------------------------------------
      private final StringProperty   nombre;
      private final StringProperty id;
      /**
       * Constructor de propietario
       * @param nombre El nombre del Propietario del vehiculo, nombre!=null
       * @param telefono El id del propietario, id!=null
       */
   public Propietario(String nombre, String id)
   {
          this.nombre=new SimpleStringProperty();
          this.nombre.set(nombre);
          this.id=new SimpleStringProperty();
          this.id.set(id);
   }


/**
 * Metodo accesor
 * @return El nombre del propietario
 */
public String getNombre() {
      return nombre.get();
}

/**
 * El nombre del propietario
 * @param nombre El nombre del Propietario del vehiculo, nombre!=null
 */
public void setNombre(String nombre) {
         this.nombre.set(nombre);
}

/**
 * Devuelve el id del propietario
 * @return El telefono
 */
public String getId() {
      return id.get();
}
/**
 * Permite fijar el id del propietario
 * @param id El telefono del propietario, telefono!=null
 */
public void setId(String id) {
         this.id.set(id);
```

Programa 75 Propietario

```
}
/**
 * Devuelve la representacion en String del propietario
 */
   public String toString()
   {
          return nombre + "   "+id;
   }

}
```

La clase VehiculoModel, ver Programa 76, tiene los siguientes atributos:

```
private final SimpleStringProperty placa;
private final SimpleIntegerProperty tipo;
private  final SimpleIntegerProperty modelo;
private  final ObjectProperty<Propietario> duenio;
```

Además cuenta con método constructor, métodos get y set.

Programa 76 VehiculoModel

```
package co.uniquindio.parqueadero.model;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.SimpleStringProperty;
/**
 * Clase para representar un vehiculo
 * @author sonia
 * @author sergio
 */
public class VehiculoModel {
/**
 * Atributos de la clase
 */
    private final SimpleStringProperty placa;
    private final SimpleIntegerProperty tipo;
    private  final SimpleIntegerProperty modelo;
    private  final ObjectProperty<Propietario> duenio;
/**
 * Constructor de la clase
 * @param placa La placa
 * @param tipo el tipo del vehiculo
 * @param modelo El modelo
 * @param miP El propietario
 */
    public VehiculoModel(String placa, int tipo, int modelo, Propietario miP) {
        this.placa = new SimpleStringProperty(placa);
        this.tipo = new SimpleIntegerProperty(tipo);
        this.modelo = new SimpleIntegerProperty(modelo);
        this.duenio=new SimpleObjectProperty<Propietario>(miP);
    }
/**
 * Metodo accesor
```

Programa 76 VehiculoModel

```
 * @return placa.get()
 */
    public String getPlaca() {
        return placa.get();
    }
/**
 * Metodo modificador
 * @param placa La placa
 */
    public void setPlaca(String placa) {
        this.placa.set(placa);
    }
  /**
   * Metodo modificador
   * @param miP El propietario
   */
    public void setPropietario(Propietario miP) {
        this.duenio.set(miP);
    }
  /**
   * Metodo accesor
   * @return duenio.get()
   */
    public Propietario getPropietario() {
        return duenio.get();
    }
 /**
  * Metodo para devover la representacion en String del tipo
  * @return el tipo
  */
    public String getTipo() {
        String tipo="Carro";
        if(this.tipo.get()==1)
        {tipo="Moto";};

       return tipo;
    }
  /**
   * Devuelve el tipo
   * @return tipo.get()
   */
    public int getTipoInt() {
        return tipo.get();
    }
  /**
   * Metodo modificador
   * @param tipo
   */
    public void setTipo(int tipo) {
        this.tipo.set(tipo);
    }
/**
 * Metodo accesor
 * @return modelo.get()
```

Programa 76 VehiculoModel

```
 */
    public int getModelo() {
        return modelo.get();
    }
/**
 * Metodo modificador
 * @param modelo
 */
    public void setModelo(int modelo) {
        this.modelo.set(modelo);
    }
}
```

La clase Puesto tiene un método constructor,y métodos tales como setPosicion(**int** i, **int** j), que recibe las coordenadas dentro de la matriz, y isDisponible(), que informa si el puesto tiene un vehiculo o está libre.

Programa 77 Puesto

```
package co.uniquindio.parqueadero.model;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleObjectProperty;


/**
 * Esta clase representa un puesto dentro del parqueadero
 * @author
 *
 */
public class Puesto
{

      //-----------------------------------------------------------------------
      //ATRIBUTOS
      //-----------------------------------------------------------------------
      private final IntegerProperty tipo;
      private ObjectProperty<VehiculoModel> miVehiculo;
      private ObjectProperty<Posicion> miPosicion;
/**
 * Metodo set
 * @param tipo
 */

      public void setTipo(int tipo) {
             this.tipo.set(tipo);
    }
/**
 * Constructor del puesto
 * @param tipo
 * @param miVehiculo
 * @param i la posicion en la fila
```

```
 * @param j a posicion en la  columna
 */
      public Puesto( int tipo, VehiculoModel miVehiculo,int  i,int  j) {
            this.tipo = new SimpleIntegerProperty(tipo);
            this.miVehiculo =new SimpleObjectProperty<VehiculoModel>();
            this.miVehiculo.set(miVehiculo);
            Posicion miP=new Posicion();
            miP.setI(i);
            miP.setJ(j);
            this.miPosicion =new SimpleObjectProperty<Posicion>();
            miPosicion.set(miP);
      }
/**
 * Constructor del puesto
 * @param tipo
 */
   public Puesto(int tipo) {
            super();
            this.tipo = new SimpleIntegerProperty(tipo);
            this.miVehiculo = null;
            this.miPosicion =new SimpleObjectProperty<Posicion>();
            miPosicion.set(new Posicion());
      }
/**
 * Metodo accesor
 * @return tipo.get()
 */
      public int getTipo() {
            return tipo.get();
      }
/**
 * Metodo para fijar la posicion
 * @param i la posicion en la fila
 * @param j a posicion en la  columna
 */
 public void setPosicion(int i, int j)
 {
       Posicion miP=new Posicion();
            miP.setI(i);
            miP.setJ(j);
            this.miPosicion =new SimpleObjectProperty<Posicion>();
            miPosicion.set(miP);

 }
 /**
  * Devuelve la posicion
  * @return miPosicion.get();
  */
public Posicion getPosicion()
{
return miPosicion.get();
}
 /**
 * Permite inicializar el vehiculo
 * @param miVehiculo El vehiculo a ubicar en el puesto.
```

```
 */
      public void setMiVehiculo(VehiculoModel miVehiculo) {
            this.miVehiculo =new SimpleObjectProperty<VehiculoModel>();
            this.miVehiculo.set(miVehiculo);
      }

/**
* Metodo accesor del vehiculo
* @return el vehiculo
 */
      public ObjectProperty<VehiculoModel> getMiVehiculo() {
            return miVehiculo;
      }
/**
 * Devuelve el vehiculo
 * @return miVehiculo.get()
 */
      public VehiculoModel getMiVehiculo2() {
            return miVehiculo.get();
      }
      /**
       * Permite eliminar el vehiculo
       */
      public void eliminarMiVehiculo() {
            this.miVehiculo = null;
      }
/**
 * Informa si esta disponible
 * @return un boolean, true si esta disponible
 */
    public boolean isDisponible()
    {
      boolean centinela=false;
      if(getMiVehiculo()==null)
      {
            centinela=true;
      }
      return centinela;
    }
 }
```

La clase Registro, ver Programa 78, permite crear un nuevo registro, especificando el puesto y el vehiculo, para ello se presenta el siguiente método constructor.

```
public Registro(Puesto miP, VehiculoModel miV) {
       this.miPuesto = new SimpleObjectProperty<Puesto>();
       this.miPuesto.set(miP);
       this.miVehiculo = new SimpleObjectProperty<VehiculoModel>();
       this.miVehiculo.set(miV);
}
```

Programa 78 Registro

```
package co.uniquindio.parqueadero.model;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;
/**
 * Permite crear un registro
 * @author sonia
 *
 */
public class Registro {
/**
 * Atributos de la clase
 */
private final ObjectProperty<Puesto> miPuesto;
private final ObjectProperty<VehiculoModel> miVehiculo;
/**
 * Constructor del registro
 * @param miP,el puesto
 * @param miV, el vehiculo
 */
public Registro(Puesto miP, VehiculoModel miV) {
	this.miPuesto = new SimpleObjectProperty<Puesto>();
	this.miPuesto.set(miP);
	this.miVehiculo = new SimpleObjectProperty<VehiculoModel>();
	this.miVehiculo.set(miV);
}
/**
 * Metodo accesor
 * @return miPuesto.get()
 */
public Puesto getMiPuesto() {
	return miPuesto.get();
}
/**
 * Metodo modificador
 * @param miP
 */
public void setMiPuesto(Puesto miP) {
	this.miPuesto.set(miP);
}
/**
 * Metodo accesor
 * @return miVehiculo.get()
 */
public VehiculoModel getMiVehiculo() {
	return miVehiculo.get();
}
/**
 * Metodo modificador
 * @param miV
 */
public void setMiVehiculo(VehiculoModel miV) {
	this.miVehiculo.set(miV);
}
```

Programa 78 Registro

```
}
```

La clase SinEspacioException, ver Programa 79,permite mostrar una excepción cuando el parqueadero está lleno, almenos, para el tipo de vehiculo que acaba de llegar.

Programa 79 SinEspacioException

```
package co.uniquindio.parqueadero.model;
/**
 * Clase para representar una excepcion cuando no hay espacio
 * @author sonia
 *
 */
public class SinEspacioException extends Exception {
/**
 * Constructor de la clase
 * @param mensaje El mensaje a mostrar
 */
public SinEspacioException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

El Programa 80 presenta la clase VehiculoRepetidoException, que se usa cuando se pretende parquear simultáneamente un mismo vehiculo varias veces.

Programa 80 VehiculoRepetidoException

```
package co.uniquindio.parqueadero.model;
/**
 * Clase para representar una excepcion
 * @author sonia
 * @author sergio
 */
public class VehiculoRepetidoException extends Exception {
/**
 * Constructor de la clase
 * @param mensaje
 */
public VehiculoRepetidoException(String mensaje) {
		super(mensaje);
		// TODO Auto-generated constructor stub
	}
}
```

La clase Principal, implementada en Programa 81, contiene la referencia a la clase principal del mundo de la lógica, es decir, Parqueadero miParqueadero, además, de una matriz de botones permite crear la interfaz del

parqueadero. El tamaño de la matriz es definido por el usuario. El método showVehiculoOverview() es el encargado de mostrar la vista para permitir el ingreso del vehiculo y de registrar la información del propietario.

Programa 81 Principal

```
package co.uniquindio.parqueadero;
import java.io.File;
import java.io.IOException;
import java.util.Optional;
import java.util.prefs.Preferences;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.fxml.FXMLLoader;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextInputDialog;
import javafx.scene.image.Image;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.layout.StackPane;
import javafx.stage.Modality;
import javafx.stage.Stage;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import co.uniquindio.parqueadero.model.Parqueadero;
import co.uniquindio.parqueadero.model.Posicion;
import co.uniquindio.parqueadero.model.Propietario;
import co.uniquindio.parqueadero.model.Puesto;
import co.uniquindio.parqueadero.model.SinEspacioException;
import co.uniquindio.parqueadero.model.VehiculoModel;
import co.uniquindio.parqueadero.model.VehiculoRepetidoException;
import co.uniquindio.parqueadero.view.RootLayoutController;
import co.uniquindio.parqueadero.view.VehiculoVistaController;

/**
 * Es la clase principal de la interfaz
 * @author sonia
 *
```

Programa 81 Principal

```
 */
public class Principal extends Application {
/**
 * Atributos de la clase
 */
    private Stage stagePrincipal;
    private Stage dialogStage;
    private BorderPane rootLayout;
    private Parqueadero miParqueadero;
    private Button matrizBotones[][];
    private Button consulta;
    private Button agregarVehiculo;

    /**
     * Constructor
     */
    public Principal() {
        }
  @Override
    public void start(Stage primaryStage) {
      int tamanio=leerEntero("Ingrese el tamaño del parqueadero (entre 3 y 10)");
      if(tamanio>2&&tamanio<=10)
      {
      matrizBotones =new Button[tamanio][tamanio];
      miParqueadero=new Parqueadero(tamanio);
      this.stagePrincipal = primaryStage;
        this.stagePrincipal.setTitle("ParqueaderoApp");
        inicializarLayoutRaiz();
        crearPanelesParqueadero();

    } else
                    try {
                          throw new Exception("El tamaño permitido es entre 3 y 10");
                    } catch (Exception e) {
                          // TODO Auto-generated catch block
                          System.out.println(e.getMessage());
                    }



    }
  /**
   * Devuelve la lista observable
   * @return miParqueadero.getMisVehiculos()
   */
    public final ObservableList<VehiculoModel> getMisVehiculos() {
      return miParqueadero.getMisVehiculos();
    }
  /**
   * Devuelve la matriz de botones
   * @return matrizBotones
   */
    public Button[][] getMatrizBotones() {
             return matrizBotones;
```

Programa 81 Principal

```
	}
/**
 * Permite fijar la matriz de botones
 * @param matrizBotones
 */
	public void setMatrizBotones(Button[][] matrizBotones) {
		this.matrizBotones = matrizBotones;
	}

/**
 * Inicializar el root layout
 */
    public void inicializarLayoutRaiz() {
        try {

            FXMLLoader loader = new FXMLLoader();
            loader.setLocation(Principal.class
                    .getResource("view/RootLayout.fxml"));
            rootLayout = (BorderPane) loader.load();
            Scene scene = new Scene(rootLayout);
            stagePrincipal.setScene(scene);
           RootLayoutController controller = loader.getController();
           controller.setMainApp(this);
            stagePrincipal.show();
            stagePrincipal.setX(0);
            stagePrincipal.setY(0);
        } catch (IOException e) {
            e.printStackTrace();
        }


    }

    /**
     * Muestra la vista del vehiculo
     */
    public void showVehiculoOverview() {
        try {
             FXMLLoader loader = new FXMLLoader();
            loader.setLocation(Principal.class.getResource("view/VehiculoVista.fxml"));
            AnchorPane personOverview = (AnchorPane) loader.load();
            VehiculoVistaController controller = loader.getController();
            controller.setMainApp(this);
           dialogStage = new Stage();
            dialogStage.setTitle("Crear vehiculo");
            dialogStage.initModality(Modality.WINDOW_MODAL);
            dialogStage.initOwner(stagePrincipal);
            Scene scene = new Scene(personOverview);
            dialogStage.setScene(scene);
             dialogStage.showAndWait();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Programa 81 Principal

```
/**
 * Se crea un manejador
 */
    final EventHandler<ActionEvent> myHandler = new EventHandler<ActionEvent>(){

            @Override
            public void handle(final ActionEvent event) {
             Button x = (Button) event.getSource();
             for(int i=0; i<matrizBotones.length; i++)
             {
                for(int j=0;j<matrizBotones[i].length; j++)
                    {
                   if (x.getId().equals(matrizBotones[i][j].getId()))
                   {
                    showVehiculoDetails(j,i);
                   }

                 }
             }

             if (x.getId().equals(consulta.getId()))
             { //System.out.println("consulta");
                }

             if (x.getId().equals(agregarVehiculo.getId()))
             {
                if(dialogStage==null)
                 {showVehiculoOverview();
                 }
                else
                { dialogStage.showAndWait();
                }

                }
        }};

/**
 * Permite crear los paneles del parqueadero
 */
    public void crearPanelesParqueadero()
    {
       matrizBotones =new
Button[miParqueadero.getMisPuestos().length][miParqueadero.getMisPuestos()[0].length];
             BorderPane raiz;
      raiz = new BorderPane();
        raiz.setLeft(new Label("                                    "));
        raiz.setRight(new Label("                                        "));
        Scene scene = new Scene(raiz);
      crearPanelSuperior(raiz);
             crearPanelCentro(raiz, miParqueadero.getMisPuestos().length);
             crearPanelnferior(raiz);
             stagePrincipal.setScene(scene);
             stagePrincipal.show();
```

Programa 81 Principal

```
    }

  /**
   * Permite crear el panel central
   * @param raiz
   * @param tamanio
   */
    public void crearPanelCentro(BorderPane raiz, int tamanio)
    {
      GridPane grid =new GridPane();
      grid.setVgap(10);
        grid.setHgap(40);
      raiz.setCenter(grid);

            for(int i=0; i<miParqueadero.getMisPuestos().length; i++)
            {
                  for(int j=0; j<miParqueadero.getMisPuestos()[i].length; j++)
                  {   Puesto miPuesto=miParqueadero.getMisPuestos()[i][j];
                        int tipo=miPuesto.getTipo();
                        String tipo1="Carro";
                        if(tipo==1)
                        {
                              tipo1="Moto";
                        }
                        matrizBotones[i][j]=new Button(tipo1);
                        matrizBotones[i][j].setId(""+i+","+j);
                        grid.add(matrizBotones[i][j], j, i);
                        matrizBotones[i][j].setOnAction(myHandler);

     }
}
            grid.setAlignment(Pos.CENTER);
               raiz.setCenter(grid);
    }
 /**
  * Permite crear el panel superior
  * @param raiz Es la raiz
  */
    public void crearPanelSuperior(BorderPane raiz)
    {

      StackPane contenedorSuperior=new StackPane();
      Label l=new Label("Aplicación del parqueadero");
      contenedorSuperior.getChildren().add(l);
        StackPane.setAlignment(l,Pos.CENTER);
        contenedorSuperior.setPadding(new Insets(50));
         raiz.setTop(contenedorSuperior);

      }
/**
 * Permite crear el panel superior
 * @param raiz
 */
    public void crearPanelnferior(BorderPane raiz)
```

Programa 81 Principal

```
    {
       StackPane sp = new StackPane();
       GridPane flow  = new GridPane();//Establecemos la orientación
       consulta=new Button("Consulta");
       agregarVehiculo=new Button("AgregarVehiculo");
       agregarVehiculo.setId("agregar");
       consulta.setId("agregar");
       agregarVehiculo.setOnAction(myHandler);
       consulta.setOnAction(myHandler);
       flow.setPadding(new Insets(0, 20, 10, 20));
       flow.add(agregarVehiculo, 0, 0);
         flow.add(consulta, 1, 0);
         flow.setVgap(20);
         flow.setHgap(40);
         // flow.getChildren().addAll(agregarVehiculo, consulta);
         flow.setAlignment(Pos.CENTER);
       sp.getChildren().add(flow);
          StackPane.setAlignment(flow,Pos.CENTER);
        flow.setPadding(new Insets(100));
        raiz.setBottom(sp);
   }
/**
 * Retorna el stage
 * @return  stagePrincipa
 */
    public Stage getPrimaryStage() {
        return stagePrincipal;
    }
/**
 * Metodo principal
 * @param args
 */
    public static void main(String[] args) {
        launch(args);
    }
  /**
   * Permite leer un mensaje
   * @param mensaje
   * @return el mensaje leido
   */
    public static String leerMensaje(String mensaje )
    {
       String salida="";
        // Muestra el mensaje
       TextInputDialog miDialogo = new TextInputDialog("");
       miDialogo.setTitle("Ingreso de datos");
       miDialogo.setHeaderText("");
       miDialogo.setContentText(mensaje);

       // Forma tradicional de obtener la resuesta.
       Optional<String> result = miDialogo.showAndWait();
       if (result.isPresent()){
           salida= result.get();
       }
```

Programa 81 Principal

```
    return salida;
    }
  /**
   * Permite mostrar un mensaje
   * @param mensaje
   * @param miA
   * @param titulo
   * @param cabecera
   * @param contenido El mensaje a mostrar
   * @param escenarioPrincipal
   */
    public static void mostrarMensaje(String mensaje, AlertType miA, String titulo,
String cabecera, String contenido, Stage escenarioPrincipal )
    {
        Alert alert = new Alert(miA);
        alert.initOwner(escenarioPrincipal);
        alert.setTitle(titulo);
        alert.setHeaderText(cabecera);
        alert.setContentText(contenido);
         alert.showAndWait();
    }
  /**
   * Permite leer un entero
   * @param mensaje
   * @return
   */
    public static int leerEntero(String mensaje)
    {
       return Integer.parseInt(leerMensaje(mensaje ));
    }

/**
 * Muestra los detalles del vehiculo en la posicion i j
 * @param i
 * @param j
 */
    private void showVehiculoDetails(int i, int j) {
      Puesto miP=miParqueadero.getMisPuestos()[j][i];
      if(miP.isDisponible()==false)
      {VehiculoModel miV=miP.getMiVehiculo2();
       if (miV != null&&miV.getPropietario()!=null) {
            String salida="";
            salida+="Placa: "+miV.getPlaca()+"\n";
            salida+="Propietario: "+miV.getPropietario().getNombre()+"\n";
            salida+="Modelo: "+(""+miV.getModelo())+"\n";
            salida+="Tipo: "+(""+miV.getTipo())+"\n";
            mostrarMensaje("Informacion del vehiculo: ", AlertType.INFORMATION,
"Información del vehiculo", "Información del vehiculo",salida, getPrimaryStage() );
          }
      }
      else
```

Programa 81 Principal

```
      {mostrarMensaje("Informacion del vehiculo: ", AlertType.INFORMATION, "Solo puede
seleccionar una celda ocupada", "", "Solo puede seleccionar una celda ocupada",
getPrimaryStage() );
      }
    }
/**
 * Metodo modificador
 * @param primaryStage
 */
    public void setPrimaryStage(Stage primaryStage) {
            this.stagePrincipal = primaryStage;
      }
/**
 * Permite agregar un vehiculo
 * @param miVehiculo
 * @return la posicion donde lo ubica
 * @throws SinEspacioException
 * @throws VehiculoRepetidoException
 */
      public Posicion agregarVehiculo(VehiculoModel miVehiculo) throws
SinEspacioException, VehiculoRepetidoException
    {
      return miParqueadero.agregarVehiculo(miVehiculo);
    }
/**
 * Permite eliminar un vehiculo
 * @param i
 * @param j
 */
      public void eliminarVehiculo(int i, int j)
       {
            miParqueadero.eliminarVehiculo(i, j);
       }
/**
 * Permite verificar si una persona esta en el parqueadero
 * @param id
 * @return La posicion
 */
      public Posicion verificarPersonaEstaEnParqueadero(String id)
      {

            return miParqueadero.verificarPersonaEstaEnParqueadero(id);
      }
/**
 * Devuelve el vehiculo en la posicion indicada
 * @param i
 * @param j
 * @return el vehiculo
 */
      public VehiculoModel getMiVehiculo (int i, int j )
       {
            return miParqueadero.getMiVehiculo(i, j);
       }
}
```

La clase VehiculoVistaController, implementado en Programa 82, se encarga de crear el tableView, en donde se listan los datos de los vehículos que ingresan al parqueadero. Este tableView se actualiza, bien sea cuando se agrega un nuevo vehiculo mediante el método handleNewVehiculo(), o cuando se elimina un vehiculo con el método handleDeleteVehiculo().
El método inicializar que se presenta a continuación, permite configurar los valores que tomarán las celdas del tableView, mediante el uso de operador lambda, aplicado a las columnas. Las líneas que realizan dicha operación se presentan subrayadas. Las expresiona lambda permiten escribir código fuente más corto y son una primera introducción al concepto de programación funcional, un paradigma que hace uso de funciones matemáticas.

```
public void initialize(URL location, ResourceBundle resources) {
        tipoComboBox.getItems().removeAll(tipoComboBox.getItems());
        tipoComboBox.getItems().addAll("Carro", "Moto");
        tipoComboBox.getSelectionModel().select("Carro");
        placa.setCellValueFactory(new PropertyValueFactory<>("Placa"));
        tipo.setCellValueFactory(new PropertyValueFactory<>("Tipo"));
        modelo.setCellValueFactory(new PropertyValueFactory<>("Modelo"));
        tbData.setItems(vehiculoModels);
    }
```

Programa 82 VehiculoVistaController

```
package co.uniquindio.parqueadero.view;

import javafx.application.Platform;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Alert;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.ComboBox;

import java.io.FileNotFoundException;
import java.net.URL;
import java.text.ParseException;
import java.util.Arrays;
import java.util.List;
import java.util.ResourceBundle;
import java.util.Timer;
import java.util.TimerTask;

import javax.swing.JOptionPane;
import javax.swing.text.Segment;

import co.uniquindio.parqueadero.Principal;
import co.uniquindio.parqueadero.model.Posicion;
import co.uniquindio.parqueadero.model.Propietario;
import co.uniquindio.parqueadero.model.SinEspacioException;
```

Programa 82 VehiculoVistaController

```
import co.uniquindio.parqueadero.model.VehiculoModel;
import co.uniquindio.parqueadero.model.VehiculoRepetidoException;

public class VehiculoVistaController implements Initializable {
	@FXML
	private TextField placaTextField;
	@FXML
	private TextField propietarioIdTextField;
	@FXML
	private TextField propietarioNombreTextField;
	@FXML
	private ComboBox tipoComboBox;
	@FXML
	private TextField modeloTextFieldl;

	@FXML
	private TableView<VehiculoModel> tbData;
	@FXML
	public TableColumn<VehiculoModel, String> placa;
	@FXML
	public TableColumn<VehiculoModel, String> tipo;
	@FXML
	public TableColumn<VehiculoModel, Integer> modelo;
	// add your data here from any source
	private final ObservableList<VehiculoModel> vehiculoModels = FXCollections.observableArrayList();
	private String style;
	private Principal mainApp;

	@Override
	public void initialize(URL location, ResourceBundle resources) {
		tipoComboBox.getItems().removeAll(tipoComboBox.getItems());
		tipoComboBox.getItems().addAll("Carro", "Moto");
		tipoComboBox.getSelectionModel().select("Carro");
		placa.setCellValueFactory(new PropertyValueFactory<>("Placa"));
		tipo.setCellValueFactory(new PropertyValueFactory<>("Tipo"));
		modelo.setCellValueFactory(new PropertyValueFactory<>("Modelo"));
		tbData.setItems(vehiculoModels);
	}

	.

	/**
	 * The constructor. The constructor is called before the initialize() method.
	 */
	public VehiculoVistaController() {

	}

	/**
	 * Is called by the main application to give a reference back to itself.
	 *
	 * @param mainApp
	 */
	public void setMainApp(Principal mainApp) {
		this.mainApp = mainApp;
```

Programa 82 VehiculoVistaController

```
	}

	/**
	 * se llama cuando se desea borrar un vehiculo
	 */
	@FXML
	private void handleDeleteVehiculo() {
		int selectedIndex = 0;
		selectedIndex = tbData.getSelectionModel().getSelectedIndex();
		if (selectedIndex >= 0) {
			{
				Propietario miP = tbData.getItems().get(selectedIndex).getPropietario();
				String id = miP.getId();
				Posicion miPos = mainApp.verificarPersonaEstaEnParqueadero(id);
				if (miPos != null) {
					mainApp.eliminarVehiculo(miPos.getJ(), miPos.getI());

	mainApp.getMatrizBotones()[miPos.getJ()][miPos.getI()].setStyle(style);
					tbData.getItems().remove(selectedIndex);
					actualizarTabla();
				}
			}
		} else {
			// Nothing selected.
			Alert alert = new Alert(AlertType.WARNING);
			alert.initOwner(mainApp.getPrimaryStage());
			alert.setTitle("Debe seleccionar una fila de la tabla");
			alert.setHeaderText("No seleccionó un vehículo");
			alert.setContentText("Por favor seleccione un vehiculo de la tabla.");
			alert.showAndWait();
		}
	}

	/**
	 * Se llama cuando se desea agregar un vehiculo
	 */
	@FXML
	private Posicion handleNewVehiculo() {
		Posicion miP = new Posicion();
		Propietario miProp;
		VehiculoModel miV;
		try {
			String placa = placaTextField.getText();
			String id = propietarioIdTextField.getText();
			String nombre = propietarioNombreTextField.getText();
			int modelo = Integer.parseInt(modeloTextFieldl.getText());
			int tipo = tipoComboBox.getSelectionModel().getSelectedIndex();
			miProp = new Propietario(nombre, id);
			miV = new VehiculoModel(placa, tipo, modelo, miProp);

			miP = mainApp.agregarVehiculo(miV);
			style = mainApp.getMatrizBotones()[miP.getI()][miP.getJ()].getStyle();
			mainApp.getMatrizBotones()[miP.getI()][miP.getJ()].setStyle("-fx-background-color:
red;");
			actualizarTabla();
```

Programa 82 VehiculoVistaController

```
                        mainApp.mostrarMensaje("Agregado  ", AlertType.INFORMATION, "", "",
                                        "El vehiculo se ubico correctamente en el parqueadero",
mainApp.getPrimaryStage());
                        placaTextField.setText("");
                        propietarioNombreTextField.setText("");
                        modeloTextFieldl.setText("");
                        propietarioIdTextField.setText("");

                } catch (SinEspacioException e) {
                        // TODO Auto-generated catch block
                        mainApp.mostrarMensaje("Error: ", AlertType.ERROR, "", "", e.getMessage(),
mainApp.getPrimaryStage());

                } catch (VehiculoRepetidoException e) {
                        // TODO Auto-generated catch block
                        mainApp.mostrarMensaje("Error: ", AlertType.ERROR, "", "", e.getMessage(),
mainApp.getPrimaryStage());

                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        mainApp.mostrarMensaje("Error: ", AlertType.ERROR, "", "", "Debe ingresar todos los
datos",
                                        mainApp.getPrimaryStage());

                }
                return miP;
        }

        public void actualizarTabla() {
                tbData.getItems().clear();
                if (mainApp != null) {
                        for (int i = 0; i < mainApp.getMisVehiculos().size(); i++)
                                vehiculoModels.add(mainApp.getMisVehiculos().get(i));
                }
        }

}
```

La clase RootLayoutController, ver Programa 83, se encarga de inicializar la ventana principal

Programa 83 RootLayoutController

```
package co.uniquindio.parqueadero.view;
import java.io.File;
import co.uniquindio.parqueadero.Principal;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.stage.FileChooser;

/**
* Clase pa
 * @author
 */
public class RootLayoutController {
```

```
/**
 * Referencia a la clase principal de la interfaz
 */
    private Principal ventanaPrincipal;

/**
 * Permite fijar la clase principal
 * @param mainApp
 */
    public void setMainApp(Principal mainApp) {
        this.ventanaPrincipal = mainApp;
    }
}
```

VehiculoVista.fxml se encarga de construir la siguiente interfaz presente en la ***Figura 112***

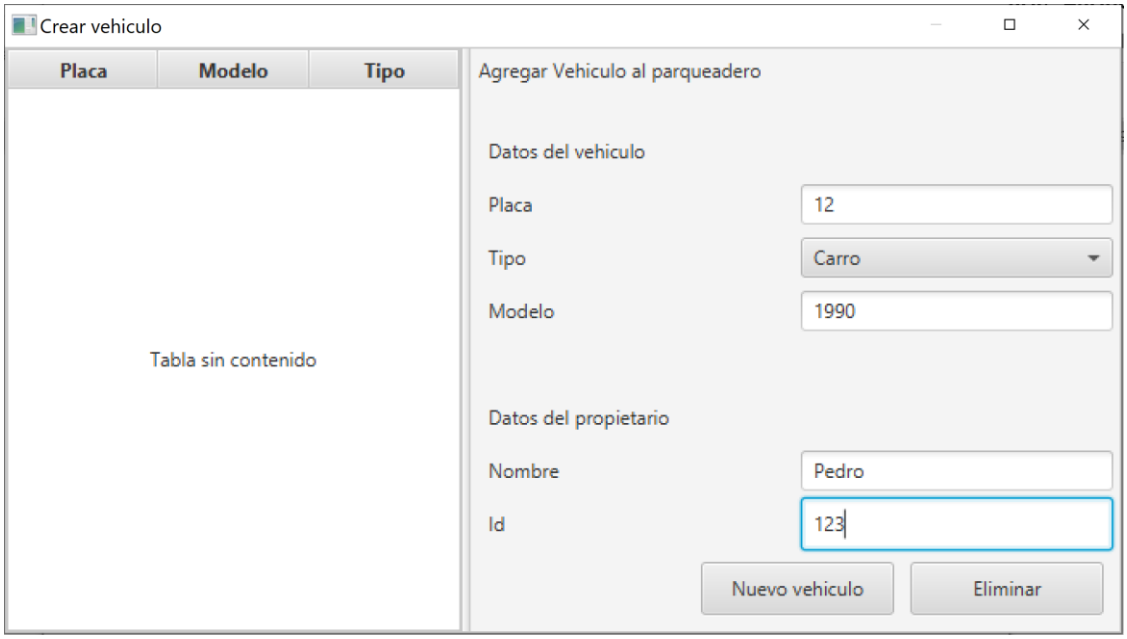


Figura 112 Ingreso de vehículos al parqueadero.

Programa 84 VehiculoVista.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>
```

Programa 84 VehiculoVista.fxml

```
<AnchorPane prefHeight="363.0" prefWidth="693.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.parqueadero.view.VehiculoVistaController">
   <children>
      <SplitPane dividerPositions="0.4126984126984127" layoutX="153.0" layoutY="70.0"
prefHeight="300.0" prefWidth="600.0" AnchorPane.bottomAnchor="0.0"
AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
        <items>
          <AnchorPane minHeight="0.0" minWidth="0.0" prefHeight="160.0"
prefWidth="100.0">
               <children>
                  <TableView fx:id="tbData" layoutY="7.0" prefHeight="361.0"
prefWidth="282.0" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0"
AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
                    <columns>
                      <TableColumn fx:id="placa" prefWidth="75.0" text="Placa" />
                      <TableColumn fx:id="modelo" prefWidth="75.0" text="Modelo" />
                        <TableColumn fx:id="tipo" prefWidth="75.0" text="Tipo" />
                    </columns>
                     <columnResizePolicy>
                        <TableView fx:constant="CONSTRAINED_RESIZE_POLICY" />
                     </columnResizePolicy>
                  </TableView>
               </children>
            </AnchorPane>
          <AnchorPane minHeight="0.0" minWidth="0.0" prefHeight="160.0" prefWidth="100.0"
styleClass="background">
               <children>
                  <Label layoutX="104.0" layoutY="51.0" styleClass="label-header"
text="Agregar Vehiculo al parqueadero" AnchorPane.leftAnchor="5.0"
AnchorPane.topAnchor="5.0" />
                  <GridPane layoutX="11.0" layoutY="47.0" prefHeight="264.0"
prefWidth="388.0" AnchorPane.leftAnchor="11.0" AnchorPane.rightAnchor="4.0"
AnchorPane.topAnchor="47.0">
                    <columnConstraints>
                      <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0"
prefWidth="100.0" />
                      <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0"
prefWidth="100.0" />
                    </columnConstraints>
                    <rowConstraints>
                        <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                      <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                        <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                        <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                        <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                        <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
```

Programa 84 VehiculoVista.fxml

```
                        <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                        <RowConstraints minHeight="10.0" prefHeight="30.0"
vgrow="SOMETIMES" />
                     </rowConstraints>
                      <children>
                        <Label text="Placa" GridPane.rowIndex="1" />
                        <Label text="Tipo" GridPane.rowIndex="2" />
                        <Label text="Modelo" GridPane.rowIndex="3" />
                        <Label prefHeight="17.0" prefWidth="295.0" text="Datos del
propietario" GridPane.rowIndex="5" />
                        <Label text="Nombre" GridPane.rowIndex="6" />
                        <Label text="Id" GridPane.rowIndex="7" />
                        <Label text="Datos del vehiculo" />
                        <TextField fx:id="placaTextField" GridPane.columnIndex="1"
GridPane.rowIndex="1" />
                        <TextField fx:id="modeloTextFieldl" GridPane.columnIndex="1"
GridPane.rowIndex="3" />
                        <TextField fx:id="propietarioNombreTextField"
GridPane.columnIndex="1" GridPane.rowIndex="6" />
                        <TextField fx:id="propietarioIdTextField" prefHeight="91.0"
prefWidth="199.0" GridPane.columnIndex="1" GridPane.rowIndex="7" />
                        <ComboBox fx:id="tipoComboBox" prefHeight="25.0"
prefWidth="195.0" GridPane.columnIndex="1" GridPane.rowIndex="2" />
                        </children>
                  </GridPane>
                  <ButtonBar layoutX="54.0" layoutY="250.0"
AnchorPane.bottomAnchor="10.0" AnchorPane.rightAnchor="10.0">
                    <buttons>
                      <Button mnemonicParsing="false" onAction="#handleNewVehiculo"
prefHeight="33.0" prefWidth="120.0" text="Nuevo vehiculo" />
                        <Button minWidth="66.0" mnemonicParsing="false"
onAction="#handleDeleteVehiculo" prefHeight="33.0" prefWidth="120.0" text="Eliminar" />
                    </buttons>
                  </ButtonBar>
               </children>
            </AnchorPane>
        </items>
      </SplitPane>
   </children>
</AnchorPane>
```

Finalmente el Programa 85, se encarga de crear el layout raiz, mostrado en la ***Figura 113*** *Layout Raiz*

Figura 113 Layout Raiz

Programa 85 RootLayout.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.input.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane prefHeight="400.0" prefWidth="600.0" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="co.uniquindio.parqueadero.view.RootLayoutController">
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
               <items>
                  <MenuItem mnemonicParsing="false" text="New">
                     <accelerator>
                        <KeyCodeCombination alt="UP" code="N" control="DOWN" meta="UP"
shift="UP" shortcut="UP" />
                     </accelerator>
                  </MenuItem>
                  <MenuItem mnemonicParsing="false" text="Open...">
                     <accelerator>
                        <KeyCodeCombination alt="UP" code="O" control="DOWN" meta="UP"
shift="UP" shortcut="UP" />
                     </accelerator>
                  </MenuItem>
                  <MenuItem mnemonicParsing="false" text="Save">
                     <accelerator>
                        <KeyCodeCombination alt="UP" code="S" control="DOWN" meta="UP"
shift="UP" shortcut="UP" />
```

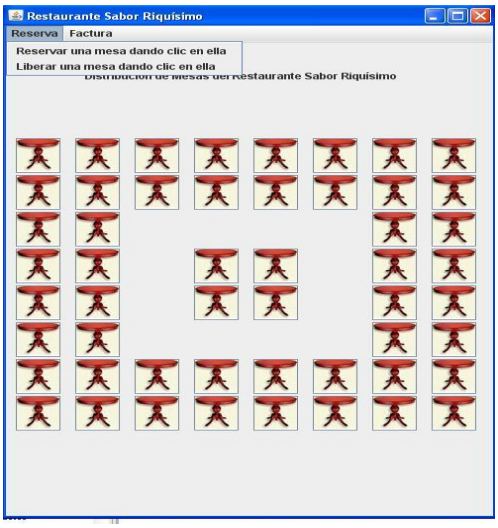
Programa 85 RootLayout.fxml

```
                    </accelerator>
                  </MenuItem>
                  <MenuItem mnemonicParsing="false" text="Save As..." />
                  <MenuItem mnemonicParsing="false" text="Exit" />
                </items>
            </Menu>
            <Menu mnemonicParsing="false" text="Edit">
              <items>
                <MenuItem mnemonicParsing="false" text="Delete" />
              </items>
            </Menu>
            <Menu mnemonicParsing="false" text="Help">
              <items>
                <MenuItem mnemonicParsing="false" text="About" />
              </items>
            </Menu>
          </menus>
        </MenuBar>
    </top>
</BorderPane>
```

Actividad 8. El restaurante

Construya la siguiente aplicación:

Se desea crear una Aplicación para manejar la información de un restaurante. En el restaurante hay mesas (en total 52), cada mesa tiene una fila y una columna que indican la ubicación dentro del restaurante. Igualmente la mesa tiene asignada una persona. Una persona posee a su vez, una cedula y un nombre. El restaurante también posee un listado de facturas. Es de anotar que máximo pueden expedirse 20 facturas.



Se debe permitir asignar una mesa a una persona, liberar una mesa, facturar una mesa. Es de anotar, que una factura tiene una persona asociada, un listado de platos(maximo 5), un iva y un total. Un plato posee un código y un precio [2][3][4].

4. BIBLIOGRAFÍA

[1] J. VILLALOBOS y R. CASALLAS, «Fundamentos de Programación – Aprendizaje Activo Basado en Casos. Edición digital, 2010.,» [En línea]. Available: https://cupi2.virtual.uniandes.edu.co/guia-estudiante-apo1

[2] JARAMILLO, CARDONA y CASTRO. Lógica de Programación en Java. Editorial ELIZCOM S.A.S, 2017

[3] JARAMILLO, CARDONA y TRIVIÑO. Aprendiendo a Programar en Java. Editorial ELIZCOM S.A.S, 2011

[4] JARAMILLO, CARDONA y VILLA. Introducción a las estructuras de datos en Java. Editorial ELIZCOM,

[5] 20080

GUPTA, Mala. Java 11 Quick Start. Packt Publishing, 2018.

[6] SCHILDTJ, Herbert. Java: A Beginner's Guide, Eighth Edition. Oracle Press, 2018

[7] SCHILDTJ, Herbert. Java: The Complete Reference, Tenth Edition. Oracle Press, 2017

[8] SANAULLA, Mohamed. Java 11 Cookbook: A definitive guide to learning the key concepts of modern application development, 2nd Edition 2nd Edition. Editorial Packt, 2018

[9] DEITEL y DEITEL. Java, como programar. Editorial Pearson, 2013

[10] SZNAJDLEDER, Pablo Augusto. El gran libro de Java a Fondo: Curso de Programación.3º Edición. Editorial Alfaomega, 2016.

[11] LILES, Karina R. Java Programming: A Beginner's Guide. Independently published, 2019

[12] DEBRAUWER. Patrones de diseño en Java. Los 23 modelos de diseño - 2ª edición. Editorial Cni, 2018

[13] TAMAN, Mohamed. JavaFX Essentials. Editorial Packt, 2015

[14] SCHILDT, H. Introducing JavaFX 8 Programming. Oracle Press, 2015

[15] LOWE, D. JavaFX For Dummies. For Dummies Series, 2015.

[16] CRINEV, S. Mastering JavaFX 10: Build advanced and visually stunning Java applications. Editorial Packt, 2018