2019

Herramientas de productividad HDP115

UNIDAD III: HERRAMIENTAS PARA ANÁLISIS Y DISEÑO

BLADIMIR DIAZ CAMPOS

UNIVERSIDAD DE EL SALVADOR | Facultad de Ingeniería y Arquitectura, Escuela de Sistemas Informáticos

Contenido

1. Diseño orientado a objetos

No existe en realidad una frontera bien definida entre la fase de Análisis y la fase de Diseño Orientado a Objetos. Como ya se mencionó, se trata en realidad de un proceso iterativo de afinamiento que se detiene cuando se cuenta con especificaciones completas de requerimientos y diseño.

También se ha mencionado que el modelo conceptual es una primera aproximación a lo que será el diseño de clases del sistema. Esta aproximación deberá ser completada, en primer lugar, con la identificación de los atributos de las clases que aún no hayan sido identificados, tanto aquellos que sean propios del negocio, como aquellos que sean de control, identificados a lo largo del proceso.

Adicionalmente, el modelo deberá ser completado con los atributos de las clases, que hasta ahora podrían haber sido advertidos por el analista en forma intuitiva, aunque no hayan sido formalmente declarados en el modelo de clases.

1.2 Diagrama de secuencia

Para identificar las operaciones de las clases, es necesario acudir a los Diagramas de secuencia, que son un tipo de diagrama de interacción. Al igual que los casos de uso, describen el flujo de eventos que ocurren entre los actores y el sistema, pero revelando las responsabilidades de cada clase en la recepción, procesamiento y respuesta que se le dará a dicha solicitud.

Simbología

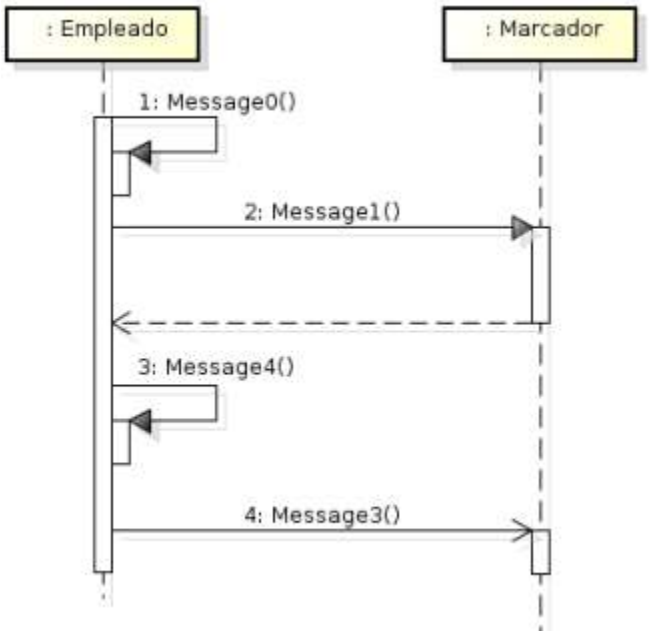
Objeto

Un objeto es una instancia de una clase, con un ciclo de vida propio e independiente de otros objetos, aunque estos fueran instancias de la misma clase. La representación de un objeto es un rectángulo con el nombre del objeto. A menos que

se le quiera dar un nombre específico, los objetos se distinguen únicamente por el nombre de la clase antecedido de dos puntos (:).

Mensaje

Un mensaje es una petición o respuesta que fluye entre los actores y los objetos, o entre los objetos. Existen dos tipos de mensajes:



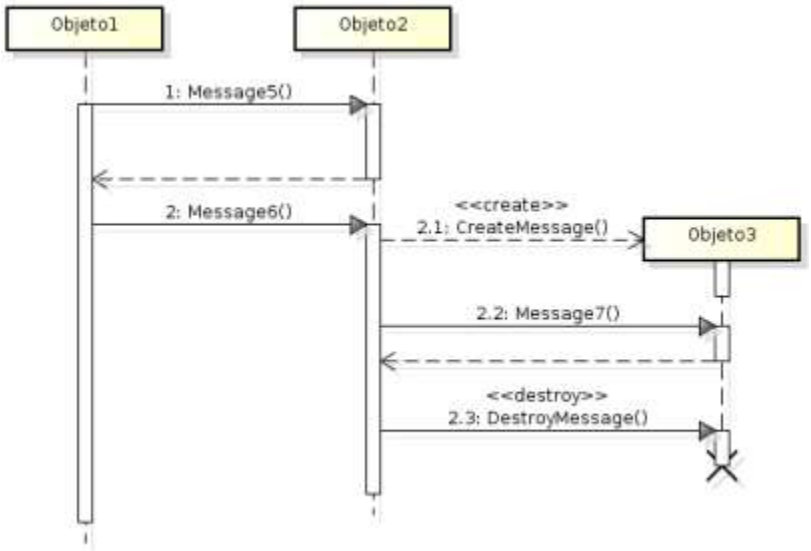
a) **Síncronos**: Se corresponden con llamadas a métodos del objeto que recibe el mensaje. El objeto que envía el mensaje queda "bloqueado" hasta que termina la llamada. Este tipo de mensajes se representa con una flecha continua de punta llena.
b) **Asíncronos**: Terminan inmediatamente y crean un nuevo hilo de ejecución dentro de la secuencia. Se representan con flechas continua de punta abierta.

Los mensajes de retorno se representan por flechas discontinuas.

Línea de vida

La línea de vida de un objeto representa el estado del objeto.

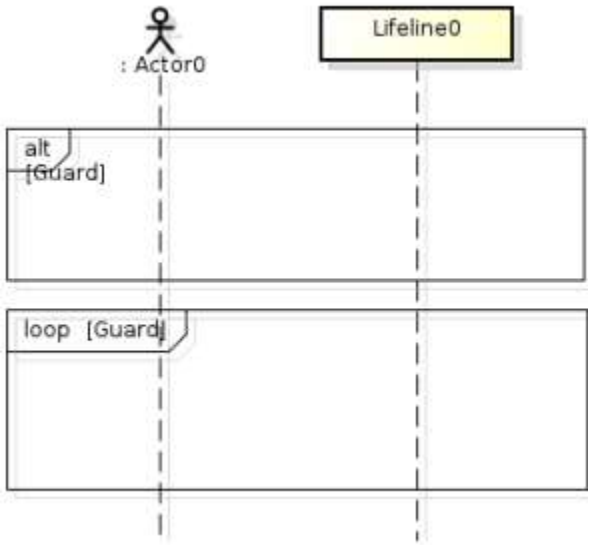
Si el objeto se encuentra activo, se representa por una línea gruesa sin relleno (barra vertical). Si el objeto se encuentra inactivo se representa por una línea punteada.



Usualmente el tiempo de actividad de un objeto está asociado al tiempo que dura un mensaje. En el ejemplo que se muestra, se realiza un mensaje de creación del objeto3, y la secuencia finaliza con la eliminación del objeto, que se denota por una X.

Fragmentos

Los fragmentos permiten definir bloques de mensajes que describen un comportamiento especial de un conjunto de mensajes: ciclos repetitivos (de mensajes), opcionales, alternativos, etc.



En la tabla 1 se definen los fragmentos más importantes usados en los diagramas de secuencia.

Tabla 1. Tipos de fragmentos más comunes en los diagramas de secuencia

OPERADOR	SIGNIFICADO
alt	Indica que el fragmento del diagrama es un flujo alternativo de eventos
loop	Indica que se trata de un bucle de interacción. Es decir, que el flujo de eventos se repite en un ciclo.
opt	Indica que el fragmento del diagrama es opcional.
par	Indica que el fragmento del diagrama incluye varios hilos de interacción.

1.3 Enfoques de diseño

Antes de la aparición de la arquitectura Web, los diagramas de secuencia jugaban un papel fundamental para alcanzar un diseño más eficiente.

Con la evolución del desarrollo orientado a objetos, los desarrolladores fueron dándose cuenta de algunos problemas relacionados al diseño, principalmente la duplicidad de código. Fue así

como aparecieron los *Patrones Generales de Software para Asignación de Reponsabilidad (GRASP,* acrónimo de su nombre en inglés*: object-oriented design General Responsability Assignment Software Patterns)*, especialmente los principios de Alta cohesión y Bajo acoplamiento.

En general, la alta cohesión se refiere a que las características de la clase sean coherentes con su definición. Es decir, que sus atributos y operaciones estén relacionadas con la clase.

El bajo acoplamiento se refiere a que las clases estén lo menos relacionadas posible para que en caso de requerirse una modificación estructural, tenga el menor impacto posible. Esto era especialmente importante si se considera que tanto las interfaces de las aplicaciones, la lógica del negocio y el acceso a los datos, estaban programados usualmente como clases.

Supongamos que deseamos programar un formulario para el registro de usuarios de una aplicación de escritorio, en este caso Java Swing. Nos centraremos únicamente en ese requerimiento.

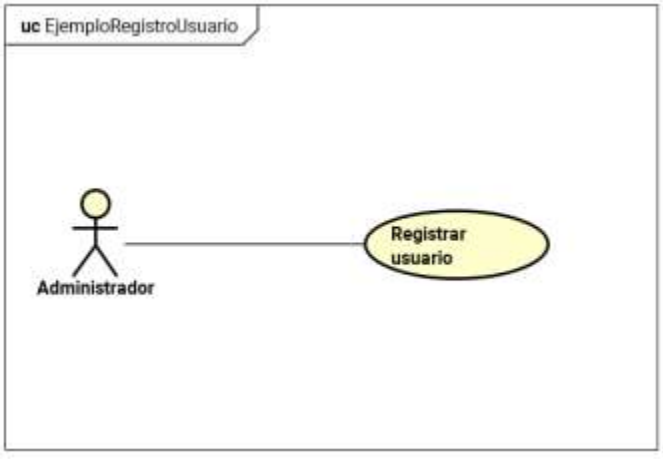


Figura 1: Modelo de casos de uso el registro de usuarios de una aplicación de escritorio.

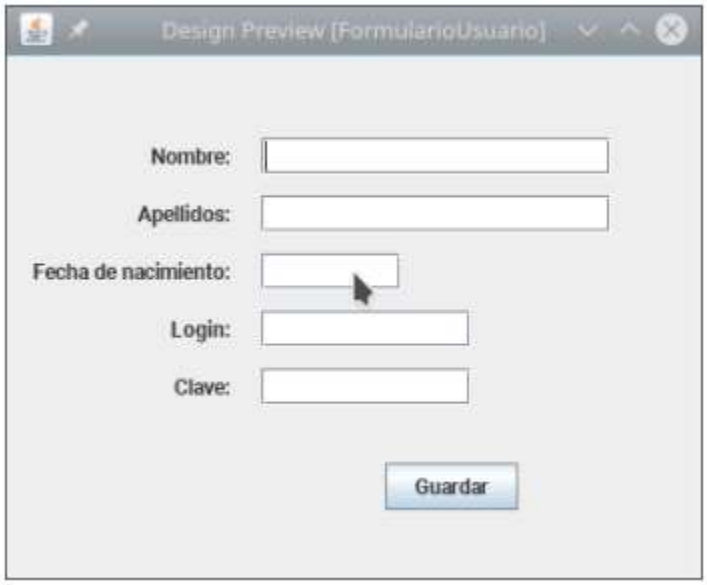


Figura 2: Prototipo correspondiente al caso de uso Registrar usuario.

Este formulario será en sí mismo una clase, que llamaremos FormularioUsuario.java., cuya estructura podemos describir como se muestra en la figura 2.

El diagrama de clases correspondiente al problema, será como se muestra en la figura 3

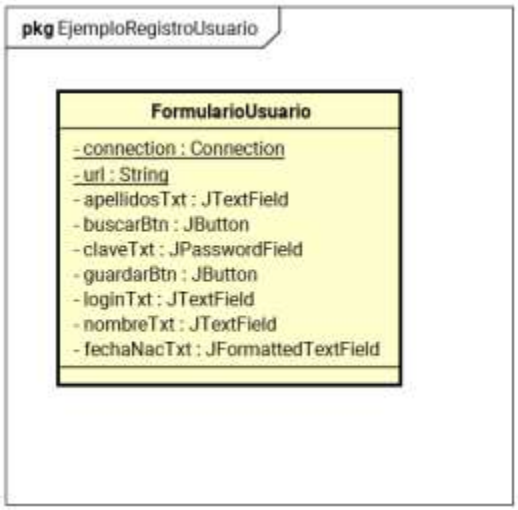


Figura 3: Diagrama de clases para el registro de usuarios.

El diagrama de secuencia, que resultaría bastante sencillo e intuitivo, se muestra en la figura 4.

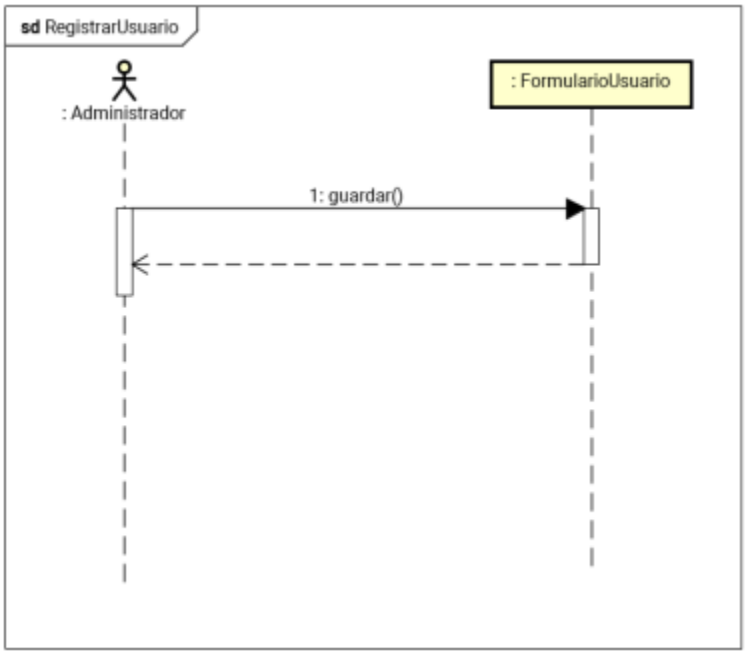


Figura 4: Diagrama de secuencia Registrar usuario.

Evidentemente el evento que dispararía una respuesta del sistema sería cuando el usuario haga clic en el botón Guardar. El diagrama de clases quedaría como se muestra en la figura 5.

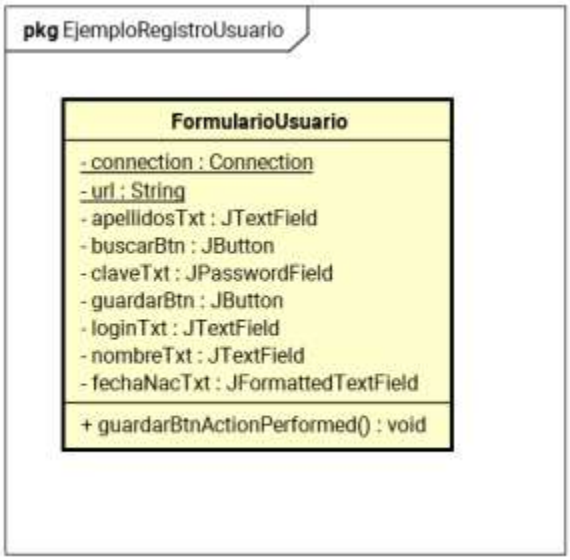


Figura 5: Diagrama de clases final para el registro de usuarios.

Al programar este método, usamos los valores capturados en los controles del formulario para enviar los valores a la consulta y ejecutarla, como se muestra en la lista de código 1.

Lista de código 1. Operación que se ejecuta al realizarse el evento guardar.

```
private void guardarBtnActionPerformed(java.awt.event.ActionEvent evt) {

try {

int idUsuario;

PreparedStatement preparedStatement = connection.prepareStatement("INSERT INTO usuario(login, clave, nombre, apellidos, fecha_nacimiento) VALUES (?,?,?,?,?) RETURNING id_usuario");

int i = 1;

preparedStatement.setString(i++, this.loginTxt.getText());

preparedStatement.setString(i++, this.claveTxt.getText());

preparedStatement.setString(i++, this.nombreTxt.getText());

preparedStatement.setString(i++, this.apellidosTxt.getText());

preparedStatement.setDate(i++, new java.sql.Date(((Date) this.fechaNacTxt.getValue()).getTime()));

ResultSet rs = preparedStatement.executeQuery();

if (rs.next()) {

idUsuario = rs.getInt(1);

JOptionPane.showMessageDialog(this, "El usuario fue registrado con el ID: " + idUsuario); }

} catch (SQLException ex) {

Logger.getLogger(FormularioUsuario.class.getName()).log(Level.SEVERE, null, ex);

}

}
```

Supongamos ahora que, además de guardar los datos de los usuarios, la pantalla también deberá servir para que se consulte a través del Login si ya existe ese usuario en el sistema. Para ello tendríamos que modificar la interfaz, incorporando un botón Buscar, como se muestra en la figura 6.

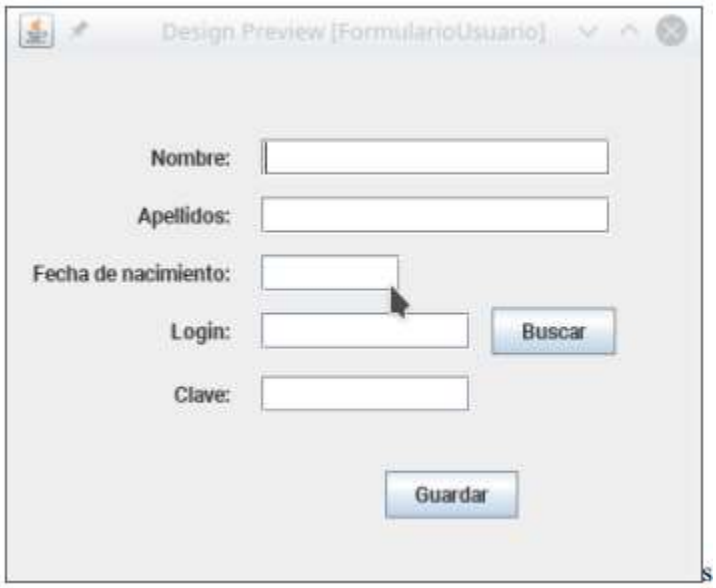


Figura 6: Prototipo modificado del formulario de registro de usuario.

El diagrama de secuencia sufriría modificaciones, quedando como se muestra en la figura 7.

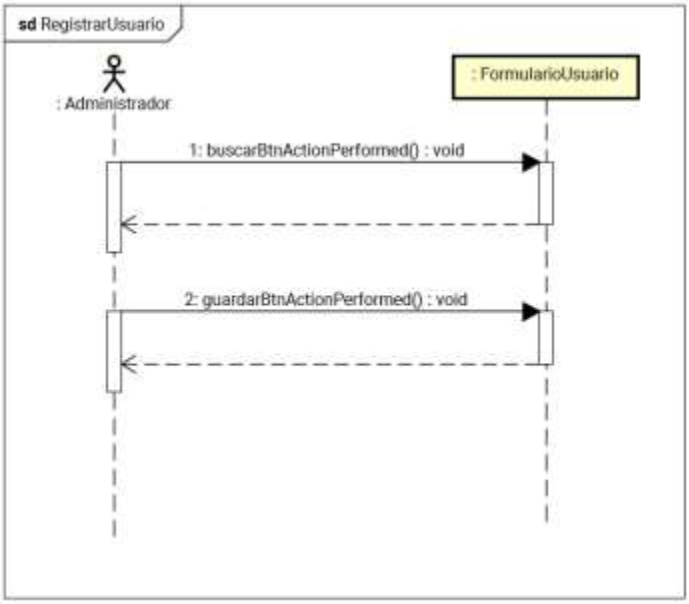


Figura 7: Diagrama de secuencia modificado.

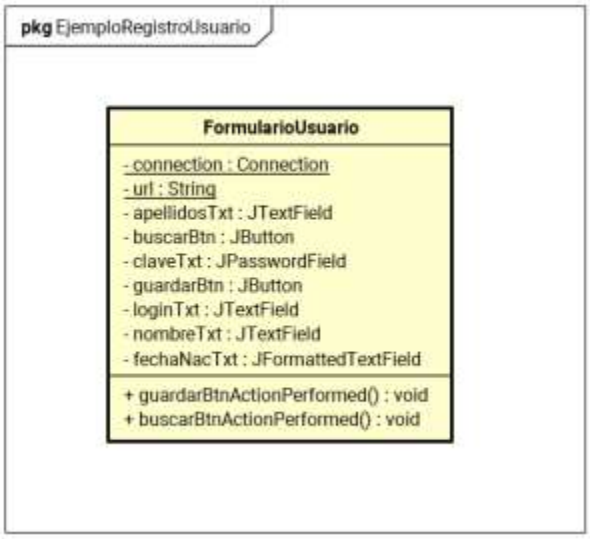


Figura 8: Diagrama de clases modificado.

El código de la operación del botón Buscar sería como se muestra en el listado de código 2

Lista de código 2. Código de la operación buscar de la clase FormularioUsuario.java

```
private void buscarBtnActionPerformed(java.awt.event.ActionEvent evt) {
try {
PreparedStatement preparedStatement = connection.prepareStatement("SELECT nombre,
apellidos, clave, login, fecha_nacimiento FROM usuario WHERE login = ?");
int i = 1;
preparedStatement.setString(i++, this.loginTxt.getText());
ResultSet rs = preparedStatement.executeQuery();
if (rs.next()) {
this.nombreTxt.setText(rs.getString(1));
this.apellidosTxt.setText(rs.getString(2));
this.claveTxt.setText(rs.getString(3));
this.loginTxt.setText(rs.getString(4));
this.fechaNacTxt.setValue(rs.getDate(5));
}else{
JOptionPane.showMessageDialog(this, "No existe un usuario con el login " +
this.loginTxt.getText());
}
} catch (SQLException ex) {
Logger.getLogger(FormularioUsuario.class.getName()).log(Level.SEVERE, null, ex);
}
}
```

Ahora supongamos que se nos pide una interfaz, únicamente para consultar los datos de algún usuario. Sin embargo, a diferencia del formulario de registro, no queremos mostrar todos los datos.

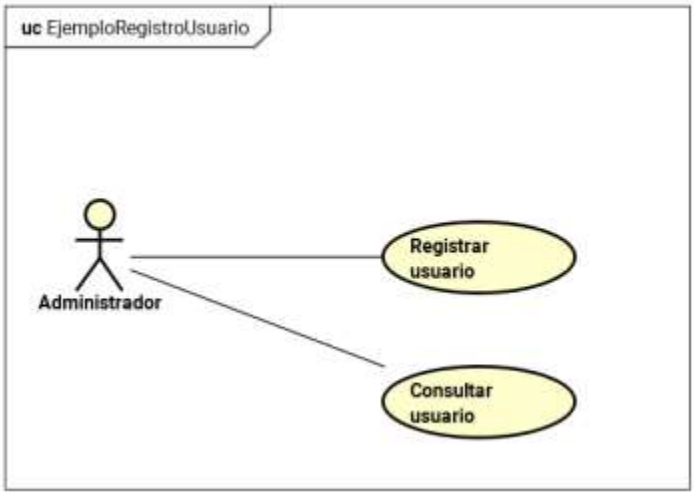


Figura 9: Modelo de casos de uso que incorpora el caso de uso Consultar usuario.

Figura 10: Prototipo del caso de uso Consultar usuario.

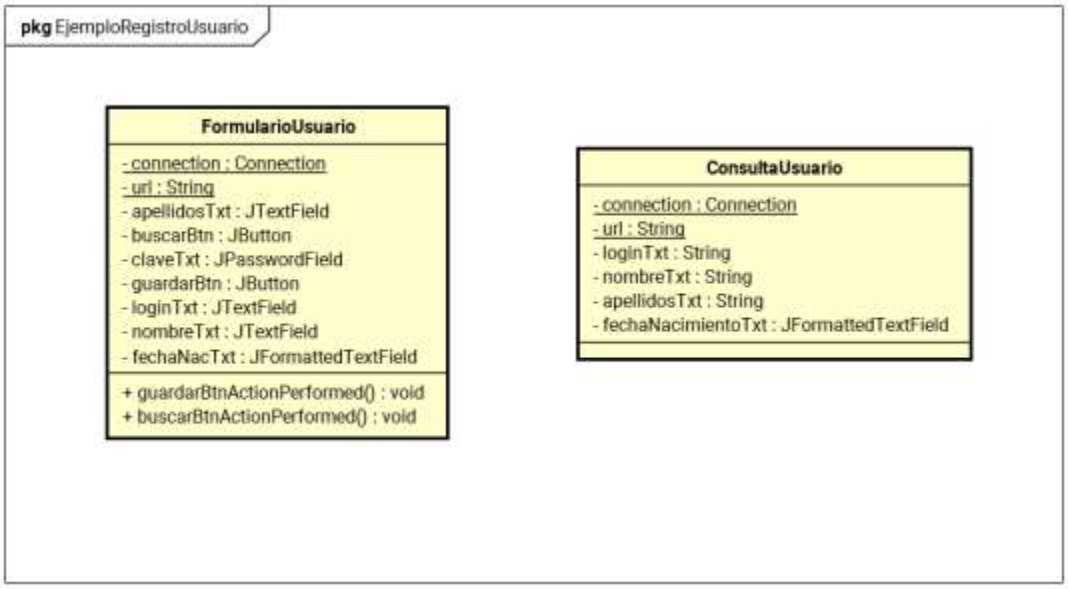


Figura 11: Diagrama de clases para el registro y consulta de usuarios.

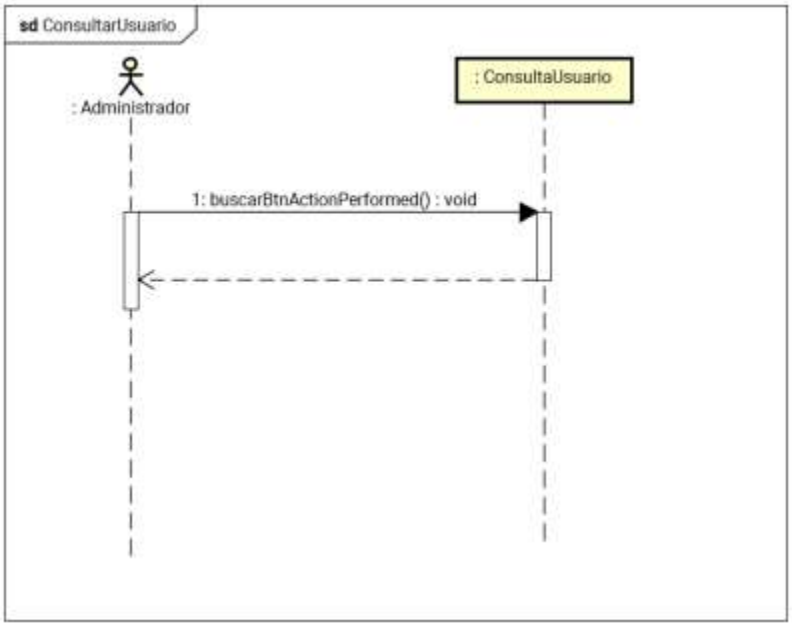


Figura 12: Diagrama de secuencia para el caso de uso Consultar usuario.

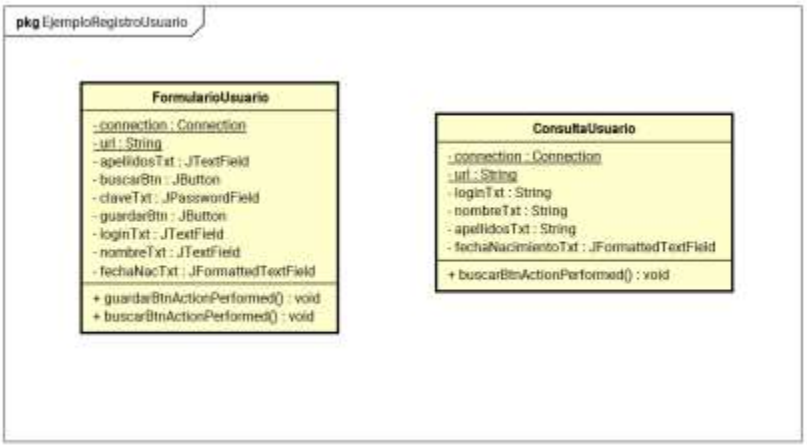


Figura 13:Diagrama de clases final para el registro y consulta de usuarios.

Lista de código 3. Operación buscar de la clase ConsultarUsuario.java

```
private void buscarBtnActionPerformed(java.awt.event.ActionEvent evt) {

try {

PreparedStatement preparedStatement = connection.prepareStatement("SELECT nombre, apellidos,

clave, login, fecha_nacimiento FROM usuario WHERE login = ?");

int i = 1;

preparedStatement.setString(i++, this.loginTxt.getText());

ResultSet rs = preparedStatement.executeQuery();

if (rs.next()) {

this.nombreTxt.setText(rs.getString(1));

this.apellidosTxt.setText(rs.getString(2));

this.loginTxt.setText(rs.getString(4));

this.fechaNacTxt.setValue(rs.getDate(5));

}else{
```

```
JOptionPane.showMessageDialog(this, "No existe un usuario con el login " + this.loginTxt.getText());

}

} catch (SQLException ex) {

Logger.getLogger(ConsultaUsuario.class.getName()).log(Level.SEVERE, null, ex);

}

}

}
```

Nótese que el código que corresponde a la persistencia de datos, en este caso la consulta; es el mismo que el escrito en el formulario de registro.

El problema inherente a la duplicidad de código es el impacto de los cambios. Supongamos que se nos pide agregar un atributo a la tabla de Usuario, ahora necesitamos almacenar la dirección. Se nos pide además que esto se muestre en ambas interfaces que ya hemos desarrollado.

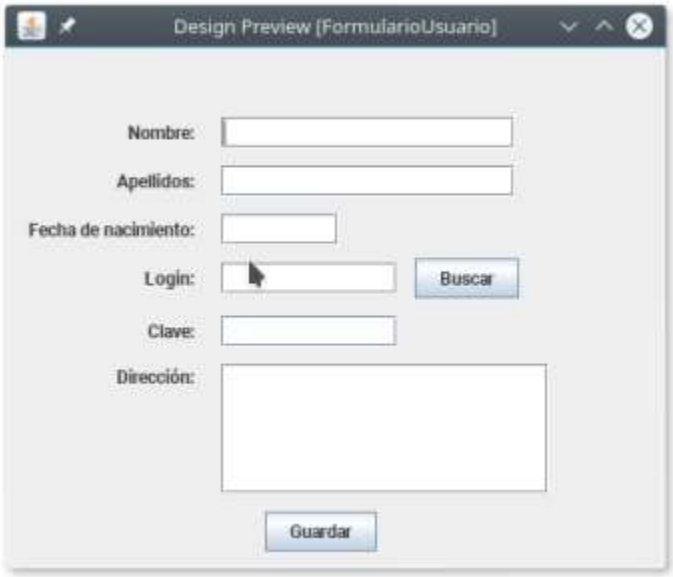


Figura 14:Formulario de registro de usuarios con el campo Dirección.

La modificación de las interfaces no solo supone agregar un JTextArea para capturar y mostrar la dirección del usuario, como se muestra en la figura 14. También supone modificar la sentencia SQL que realiza la consulta y la que realiza el almacenamiento.

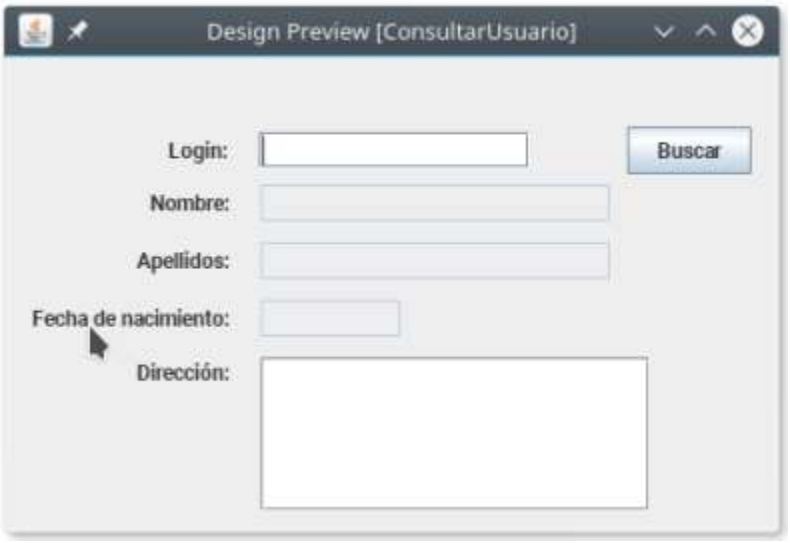


Figura 15: Interfaz de consulta de usuarios con el campo Dirección.

La lista de código 4., muestra el cambio que se deberá realizar en el método buscar de la interfaz de consulta.

Lista de código 4. Operación buscar de la clase ConsultaUsuario.java modificada.

```
private void buscarBtnActionPerformed(java.awt.event.ActionEvent evt) {

try {

PreparedStatement preparedStatement = connection.prepareStatement("SELECT nombre, apellidos,

clave, login, fecha_nacimiento, direccion FROM usuario WHERE login = ?");

int i = 1;

preparedStatement.setString(i++, this.loginTxt.getText());

ResultSet rs = preparedStatement.executeQuery();
```

```
if (rs.next()) {

this.nombreTxt.setText(rs.getString(1));

this.apellidosTxt.setText(rs.getString(2));

this.loginTxt.setText(rs.getString(4));

this.fechaNacTxt.setValue(rs.getDate(5));

this.direccionTxt.setText(rs.getString(6));

}else{

JOptionPane.showMessageDialog(this, "No existe un usuario con el login " + this.loginTxt.getText());

}

} catch (SQLException ex) {

Logger.getLogger(ConsultaUsuario.class.getName()).log(Level.SEVERE, null, ex);

}

}

}
```

Este cambio también deberemos realizar en el formulario de registro. He aquí el problema de la no reutilización de código.

Para resolver esto, podemos separar las responsabilidades de las clases. Dejaremos las clases de formularios como objetos de frontera. Y crearemos una clase para que se encargue únicamente del acceso a los datos.

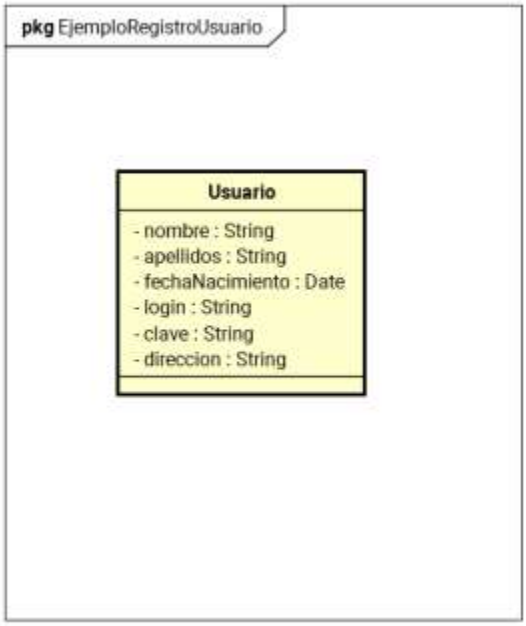


Figura 16:Modelo conceptual de clases.

Con este cambio, la secuencia se vería afectada y quedaría como se muestra en las figuras 17 y 18.

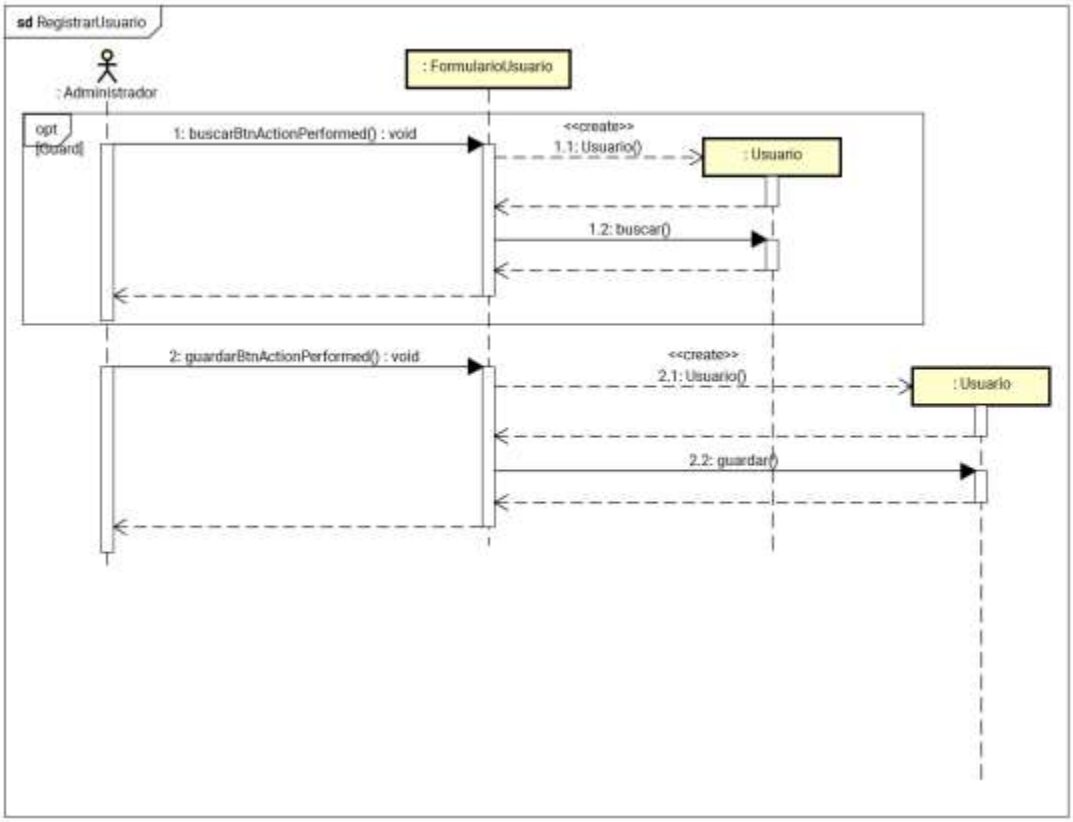


Figura 17:Diagrama de secuencia del caso de uso Registrar usuario, con clases de modelo.

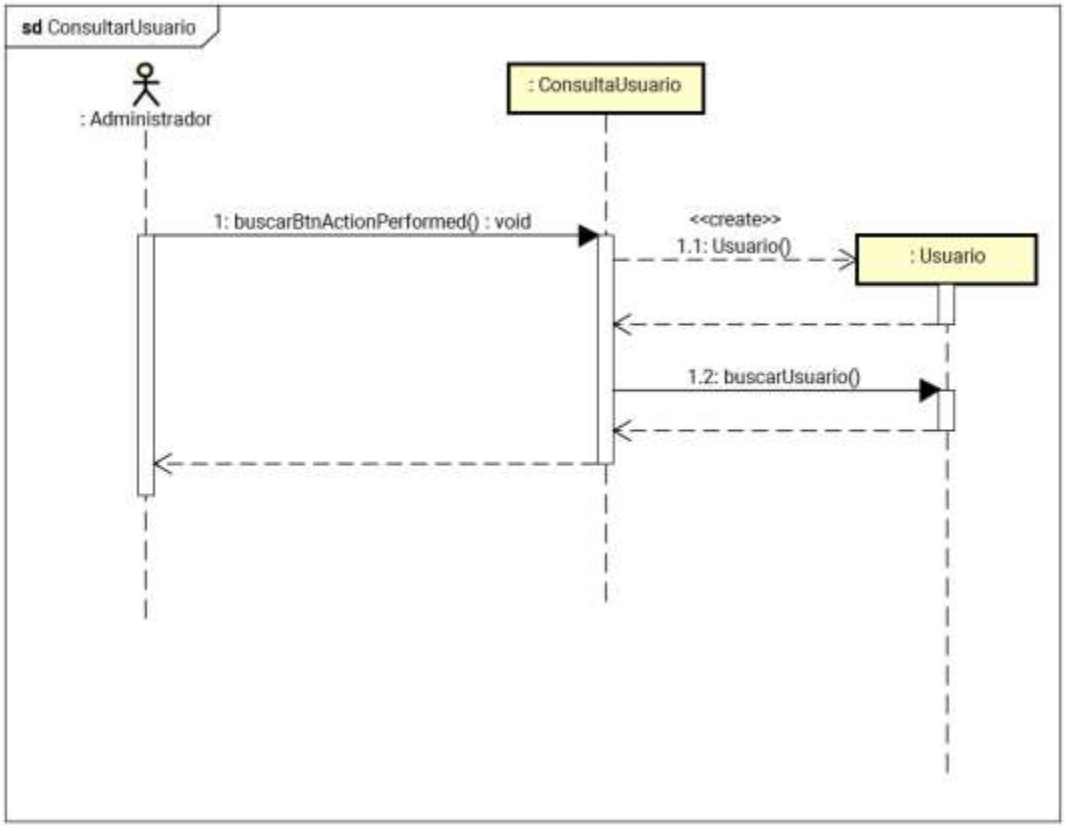


Figura 18:Diagrama de secuencia del caso de uso Consultar usuario, con clases de modelo.

Nótese que ahora, ambos flujos de evento invocan a la misma operación de la clase Usuario, por lo que, si fuera necesaria una modificación; solo tendríamos que realizarla una vez. El código del método buscar se muestra en la lista 5.

Lista de código 5. Operación buscar de la clase ConsultaUsuario.java modificada.

```
private void buscarBtnActionPerformed(java.awt.event.ActionEvent evt) {
Usuario usuario = new Usuario();
usuario.setLogin(this.loginTxt.getText());
usuario.buscarUsuario();
if (usuario.getIdUsuario() != null) {
this.nombreTxt.setText(usuario.getNombre());
```

```
this.apellidosTxt.setText(usuario.getApellidos());

this.loginTxt.setText(usuario.getLogin());

this.fechaNacTxt.setValue(usuario.getFechaNac());

this.direccionTxt.setText(usuario.getDireccion());

}else{

JOptionPane.showMessageDialog(this, "No existe un usuario con el login " + this.loginTxt.getText());

}

}
```

Lista de código 6. Operación buscarUsuario de la clase Usuario

```
public void buscarUsuario(){

try {

PreparedStatement preparedStatement = this.getConnection().prepareStatement("SELECT id_usuario,

nombre, apellidos, clave, login, fecha_nacimiento, direccion FROM usuario WHERE login = ?");

int i=1;

preparedStatement.setString(i++, this.login);

ResultSet rs = preparedStatement.executeQuery();

if(rs.next()){

this.setIdUsuario(rs.getInt(1));

this.setNombre(rs.getString(2));

this.setApellidos(rs.getString(3));
```

```
this.setClave(rs.getString(4));
this.setLogin(rs.getString(5));
this.setFechaNac(rs.getDate(6));
this.setDireccion(rs.getString(7));
}
} catch (SQLException ex) {
Logger.getLogger(Usuario.class.getName()).log(Level.SEVERE, null, ex);
}
}
```

Con esta modificación estructura, podemos notar que cuando se requiera un nuevo cambio en el modelo, solo será necesario modificar las funciones de persistencia una vez.