



Centro Integrado de Formación Profesional

AVILÉS

Principado de Asturias

UNIDAD 1: CREACIÓN DE INTERFACES DE USUARIO

DESARROLLO DE INTERFACES

2º CURSO

C.F.G.S. DESARROLLO DE APLICACIONES MULTIPLATAFORMA

ÍNDICE

ÍNDICE	1
UNIDAD 1: CREACIÓN DE INTERFACES DE USUARIO	3
1.1. Arquitectura de las aplicaciones gráficas	3
1.1.1. MVC (Model-View-Controller). Modelo Vista Controlador	3
1.1.2. MVP (Model-View-Presenter). Modelo-Vista-Presentador	4
1.1.3. MVVM (Model-View-ViewModel). Modelo-Vista-VistaModelo.....	5
1.2. Herramientas de edición de interfaces.....	6
1.2.1. Área de diseño	9
1.2.2. Paleta de componentes	10
1.2.3. Editor de propiedades	11
1.2.4. Edición de código	12
1.3. Bibliotecas de componentes nativas y multiplataforma.....	13
1.3.1. Swing.....	13
1.3.2. SWT.....	14
1.3.3. JavaFX.....	14
1.3.4. Qt.....	15
1.3.5. OpenGL.....	15
1.3.6. DirectX.....	16
1.3.7. TKinter	16
1.3.8. Electron	17
1.3.9. GTK+	17

1.4. Sistemas de control de versiones	19
1.4.1. Tipos de sistemas de control de versiones.....	19
1.4.2. Introducción a Git.....	21
1.4.3. Instalación de Git.....	22
1.4.4. Comandos Git	24
1.4.5. Sincronización con repositorio remoto.....	26
1.5. Programación Orientada a Objetos.....	27
1.5.1. Clases.....	27
1.5.2. Propiedades	28
1.5.3. Métodos.....	29
1.5.4. Bibliotecas de clases	31
1.6. Componentes: características y campo de aplicación	34
1.6.1. Contenedores. Diálogos modales y no modales	35
1.6.2. Gestión de componentes de la interfaz	39
1.6.3. Ubicación y alineamiento de componentes	40
1.6.4. Propiedades de los componentes	42
1.6.5. Enlace de componentes a orígenes de datos.....	48
1.7. Controles de usuario. Bibliotecas de controles	60
1.8. Asociación de acciones a eventos.....	66
1.8.1. Programación de eventos	66
1.8.2. Escuchadores.....	66
ÍNDICE DE FIGURAS.....	73
BIBLIOGRAFÍA – WEBGRAFÍA	74

UNIDAD 1: CREACIÓN DE INTERFACES DE USUARIO

En la actualidad, existen multitud de ejemplos de aplicaciones informáticas en las cuales las interfaces de usuario correspondientes ayudan a la vida cotidiana de los seres humanos que las utilizan. Naturalmente, desde un ordenador se pueden proporcionar múltiples tipos de aplicaciones, pero también se puede pensar en un cajero automático el cual también requiere una interactividad con el usuario. Además, se pueden encontrar otros ejemplos que requieren interacción con el usuario como las televisiones inteligentes, las cuales proporcionan una serie de operaciones, una consola de videojuegos y por supuesto, tabletas y teléfonos móviles. Las interfaces de usuario están en continua evolución, por lo que en esta unidad se pretenden conocer aquellas herramientas que permiten su diseño y desarrollo.

1.1. ARQUITECTURA DE LAS APLICACIONES GRÁFICAS

Desde las primeras aplicaciones con interfaces de usuario hasta la actualidad, su arquitectura ha ido evolucionando desde una propiamente centralizada (todo en el mismo host) hasta una distribuida en la que se reparte el sistema de información en varias máquinas, clientes y servidores, aprovechando de esta forma las capacidad gráficas y de cómputo de los equipos cliente. Lo más habitual actualmente es el desarrollo de aplicaciones en n-capas siendo el caso base $n=3$, es decir, sistemas de información en tres capas: interfaz de usuario, lógica de negocio y capa de datos.

En el diseño de aplicaciones con entornos gráficos de usuarios, se suelen seguir una serie de patrones de diseño que se basan en esa arquitectura de n-capas. A continuación, se enumeran y detallan los más conocidos.

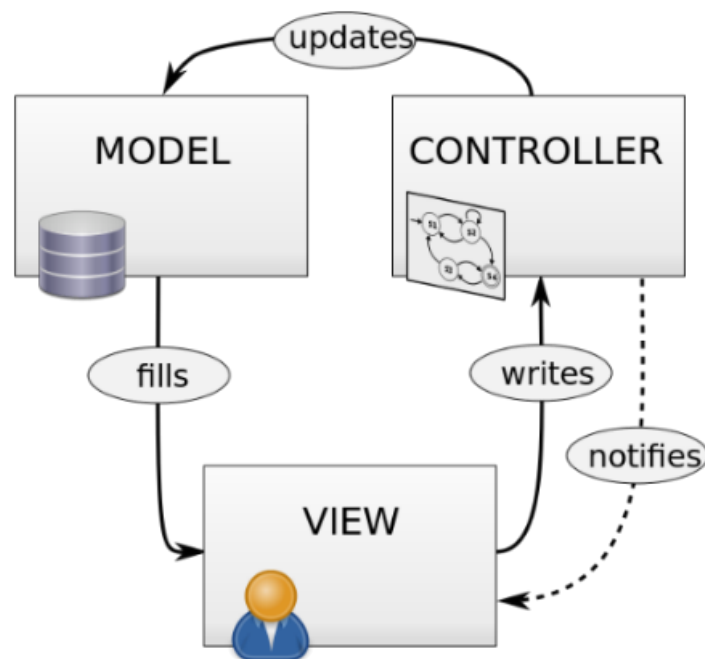
1.1.1. MVC (MODEL-VIEW-CONTROLLER). MODELO VISTA CONTROLADOR

Mediante este patrón se consiguen separar los datos (modelo) de la interfaz de usuario (vista), de manera que los cambios en la interfaz no afecten a los datos y viceversa, es decir, que estos puedan ser cambiados sin afectar a la interfaz de usuario.

Se divide en:

- **Modelo:** Es la parte del código responsable de contactar con los datos de la aplicación. Es un intermediario entre el origen de datos y la vista. Almacena y lee los datos además de asegurar su consistencia y validación. Contiene, por tanto, toda la lógica de negocio de la aplicación. Gracias a la separación de responsabilidades, el código llega a estar más estructurado además de ser más flexible a modificaciones.
- **Vista:** Es la parte de código responsable de la presentación al usuario de los datos obtenidos del modelo. La vista no contiene ninguna lógica de negocio, solo se encarga de mostrar los datos, lo que facilita realizar cambios posteriores en la aplicación. Una vista puede cambiar el modelo solo si la modificación conlleva un cambio en la forma en la que se muestran los datos. De hecho, si se necesita realizar un cambio en el interfaz de usuario, solo impactará en la vista, pero no en la lógica de negocio.

- Controlador: Define la forma en que la interfaz reacciona a la entrada del usuario. Dependiendo de las acciones del usuario, actualiza el modelo y refresca la vista. También puede transferir el control a otro controlador. Además, permite el acceso a ciertas partes de la aplicación solo a usuarios autorizados.

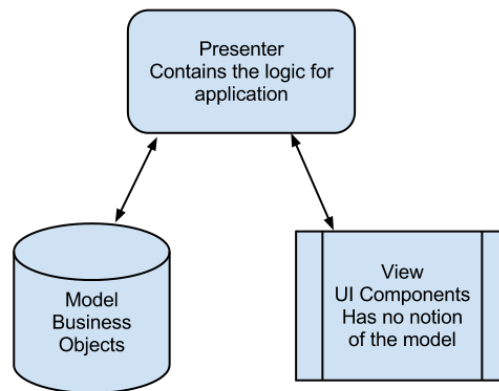


1. MVC

1.1.2. MVP (MODEL-VIEW-PRESENTER). MODELO-VISTA-PRESENTADOR

Este patrón se basa en dividir y estructurar la arquitectura de las interfaces de usuario en capas. Se diferencia del MVC en que separa aún más la lógica de presentación de forma que facilita las pruebas unitarias y la modularidad. Consta de los siguientes elementos:

- Modelo: Tiene las mismas funciones que en MVC.
- Vista: En el diseño MVP es la parte del código responsable de la presentación de los datos que proporciona el **presenter** o presentador al usuario. Cuando el usuario interactúa con la interfaz, se lanza la lógica contenida en el presenter o presentador.
- Presentador: Enlaza la vista y el modelo. Contiene la lógica de negocio de la aplicación y maneja las interacciones del usuario. No tiene acceso a la vista, lo que implica que no accede directamente a sus controles; lo único que puede hacer es pedir que se refresque la vista. En función de las acciones lanzadas por el usuario, lee los datos requeridos a través del modelo, realiza los cálculos necesarios y pasa el resultado a la vista. El presenter también maneja autorización y autenticación del usuario, permitiendo solo a usuarios autorizados acceder a partes específicas de la aplicación.

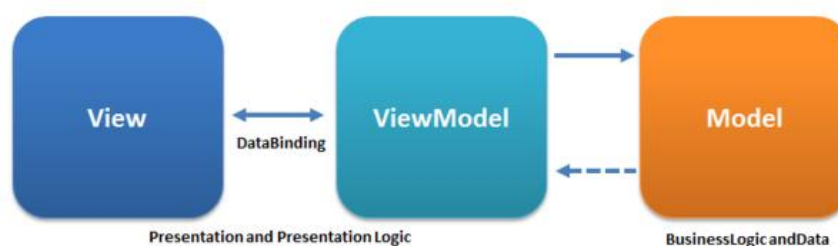


2. MVP

1.1.3. MVVM (MODEL-VIEW-VIEWMODEL). MODELO-VISTA-VISTAMODELO

Este patrón pretende facilitar la creación de las distintas vistas de la aplicación usando la división de responsabilidades en tres capas diferentes: vista, vista-modelo y modelo. La idea de este patrón se centra primeramente en la vista utilizando un patrón Observador (Observer) el cual está evaluando los datos cambiantes en la capa vista-modelo y respondiendo a ellos mediante mecanismos de enlace de datos (binding). Gracias a esta estrategia de enlace de datos, se minimiza su lógica considerablemente, el código está más estructurado y abierto a modificaciones y realizar pruebas es más sencillo. Está dividido en los siguientes elementos:

- Modelo (Model). Responsable de la lógica de negocios. Se encarga, entre otras operaciones, del procesado, almacenamiento, modificación y entrega de los datos esperados al ViewModel.
- Vista (View). La capa de la vista es la responsable de la presentación de los datos, estado del sistema y las operaciones habituales en la interfaz de usuario, además de la inicialización y enlazado del ViewModel a sus elementos. Es aquí donde se ajusta la interacción con el usuario.
- ViewModel. Encargado de entregar datos del modelo a la vista y manejar las acciones de usuario. Es importante mencionar que proporciona los flujos de datos que llegan a la vista. Se podría visualizar como un puente entre modelo y vista. Cuando el modelo cambia, el ViewModel proporciona la actualización de los datos a los componentes de la vista a través de mecanismos llamados LiveData (variables observables). Por último, reseñar que una vista puede contener distintos ViewModels.



3. MVVM

1.2. HERRAMIENTAS DE EDICIÓN DE INTERFACES

En el mundo del desarrollo, lo habitual es valerse de algún entorno integrado de desarrollo de software que facilite esta labor mediante algún sistema de ventanas, en el que se incluyen diferentes regiones rectangulares, llamadas "ventanas" cuya parte central es un conjunto de herramientas (toolkit). A este tipo de herramientas se le llama IDE (Integrated Development Environment) y provee de procedimientos que permiten incluir los componentes de las bibliotecas de componentes gráficos y añadirlos de un modo sencillo e intuitivo en la aplicación que estés desarrollando. También sirven como medio de entrada de las acciones del usuario.

A continuación, se listan varios de los IDE más utilizados en la actualidad:

- [Microsoft Visual Studio](https://visualstudio.microsoft.com/es/vs/community/). Es un entorno para el desarrollo de aplicaciones en entornos de escritorio, web y dispositivos móviles con la biblioteca .NET framework de Microsoft. Permite el uso de diferentes lenguajes de programación como C++, C#, F# o Visual Basic. En la actualidad se pueden crear aplicaciones para Windows, aplicaciones web, para móvil y RIA (Rich Internet Applications). Dispone de una versión Community gratuita que se puede conseguir en el siguiente enlace:
<https://visualstudio.microsoft.com/es/vs/community/>
- [NetBeans](#). IDE distribuido por Oracle bajo licencia GNU GPL. Está desarrollado en Java, aunque permite crear aplicaciones en diferentes lenguajes, Java, C++, PHP, Ajax, Python y otras.
- [Eclipse](#). IDE creado inicialmente por IBM ahora es desarrollado por la fundación Eclipse. Se distribuye bajo la Licencia Pública Eclipse, que, aunque libre, es incompatible con la licencia GNU GPL. Tiene como característica más destacable su ligereza ya que se basa en módulos, partiendo de una base muy sencilla con un editor de texto con resaltado de sintaxis, compilador, un módulo de pruebas unitarias con JUnit, control de versiones con CVS, integración con Ant, asistentes (wizards) para creación de proyectos, clases, tests, etc. y refactorización. Si se precisan funcionalidades más allá, éstas se incluirán como módulos aparte que van completando el IDE.
- [JDeveloper](#). Es un entorno de desarrollo integrado desarrollado por Oracle Corporation para los lenguajes Java, HTML, XML, SQL, PL/SQL, Javascript, PHP, Oracle ADF, UML y otros. Las primeras versiones de 1998 estaban basadas en el entorno JBuilder de Borland, pero desde la versión 9i de 2001 está basado en Java, no estando ya relacionado con el código anterior de JBuilder. Tras la adquisición de Sun Microsystems por parte de Oracle, está cada vez en más desuso, ya que NetBeans ofrece mayores opciones.
- [IntelliJ IDEA Community](#). Versión de código abierto del IDE de JetBrains, especializado en Java, pero también con soporte para otros lenguajes a través de complementos.
- [PyCharm](#). IDE de JetBrains enfocado en Python. Dispone también de una versión Community.

Por otra parte, existen editores que no llegan a la categoría de integrados, pero sí que pueden considerarse entornos de desarrollo ya que, además de reconocer la sintaxis propia de muchísimos lenguajes de programación (siendo en prácticamente todos los casos ampliables a otros), permiten la adición de plugins o extensiones para trabajar con lenguajes, compiladores, depuradores, sistemas de control de versiones, FTP, etc. Son los **editores avanzados** y a continuación se enumeran los más populares.

- **Sublime Text** (<https://www.sublimetext.com/>). Editor de código fuente que dispone de una versión de evaluación plenamente funcional sin fecha de caducidad, aunque el sistema recuerda que se debe obtener una licencia completa.
- **Visual Studio Code** (<https://code.visualstudio.com/>). Editor de código fuente de Microsoft. Al igual que el anterior, basado en paquetes o plugins. En este caso es gratuito y de código abierto, aunque la descarga oficial está bajo software privativo incluyendo características propias de Microsoft.

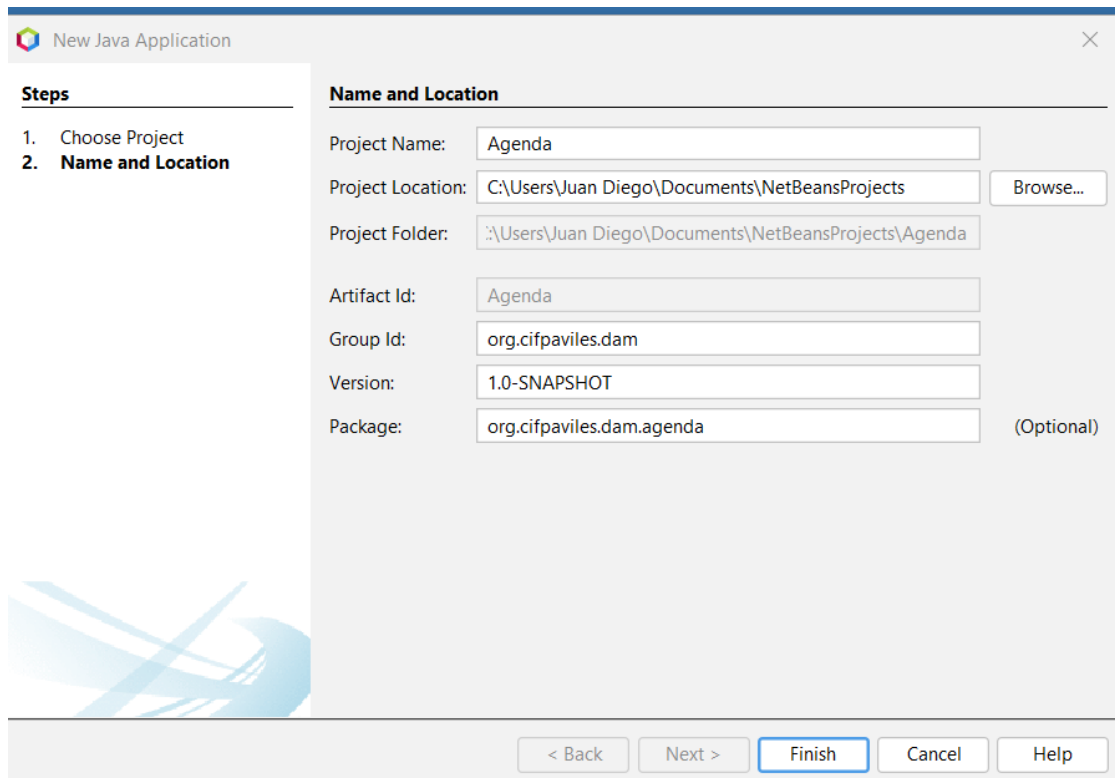
Para desarrollar los contenidos relacionados con Java, en este caso se ha elegido NetBeans como entorno de desarrollo integrado. NetBeans permite crear aplicaciones tanto de escritorio, como web como para dispositivos móviles y además se distribuye bajo licencia GNU GPL. Es multiplataforma e incluye varios lenguajes de programación.

Una de sus principales ventajas es que resuelve por sí mismo el tema de la colocación de los componentes en una interfaz gráfica, aspecto de cierta complejidad a la hora de programar, de forma que el desarrollador o desarrolladora sólo tiene que colocar los controles usando el ratón y el IDE se encarga de programarlo. También permite la inclusión de componentes creados por otros desarrolladores.

La última versión a fecha de este texto es la 19, la cual se puede conseguir en este enlace:
<https://netbeans.apache.org/download/nb19/>

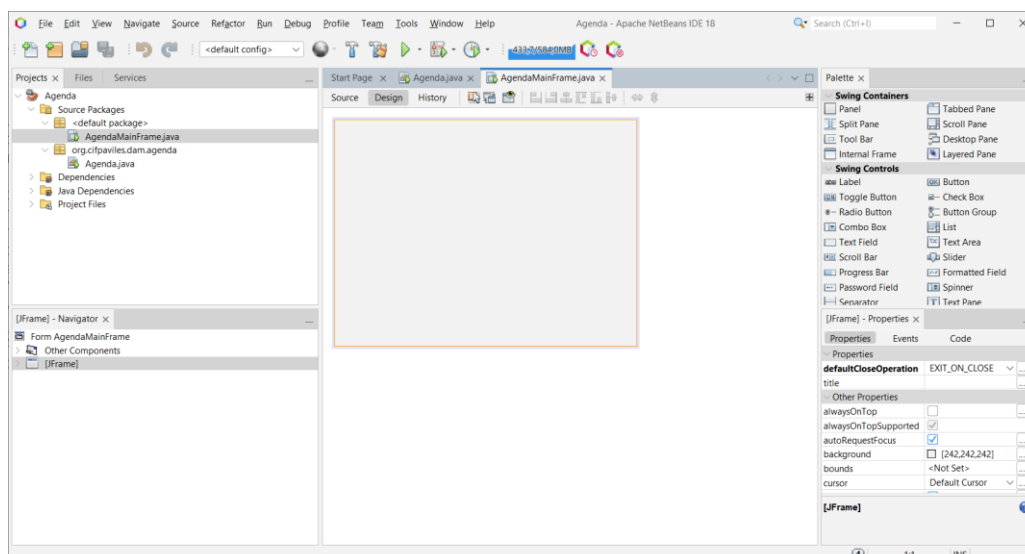
Como puede verse, hay versiones para distintos sistemas operativos, una independiente de plataforma y las fuentes. Por ejemplo, si se va a trabajar con Windows, lo ideal es descargar la versión correspondiente. Una vez que se tiene el entorno instalado y funcionando, como ejemplo, se va a comenzar creando un proyecto llamado **agenda** que pueda utilizar las clases incluidas en la biblioteca Swing y utilice como herramienta de automatización [Apache Maven](#). Para ello sólo hay que seleccionar como tipo de proyecto **Java Application** dentro de la categoría **Java with Maven**.

A continuación, se rellenan los datos principales del proyecto, como su nombre, la localización de los archivos en disco tal como se ven en la siguiente imagen



4. Proyecto de ejemplo NetBeans

A partir de esta operación, se creará un nuevo proyecto con una sola clase dentro del paquete agenda. Hasta el momento esta aplicación no es una aplicación gráfica. Si se hace clic con el botón derecho sobre el paquete agenda y se selecciona **New File>Swing GUI Forms>JFrame Form** se estará creando una nueva ventana para la aplicación. Como nombre, se le puede dar **AgendaMainFrame**. Una vez añadido, el entorno de desarrollo quedará de forma similar a la de la imagen.

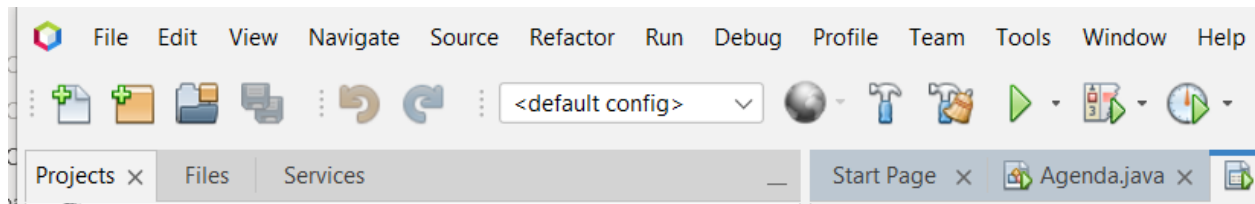


5. Primer formulario Swing.

1.2.1. ÁREA DE DISEÑO

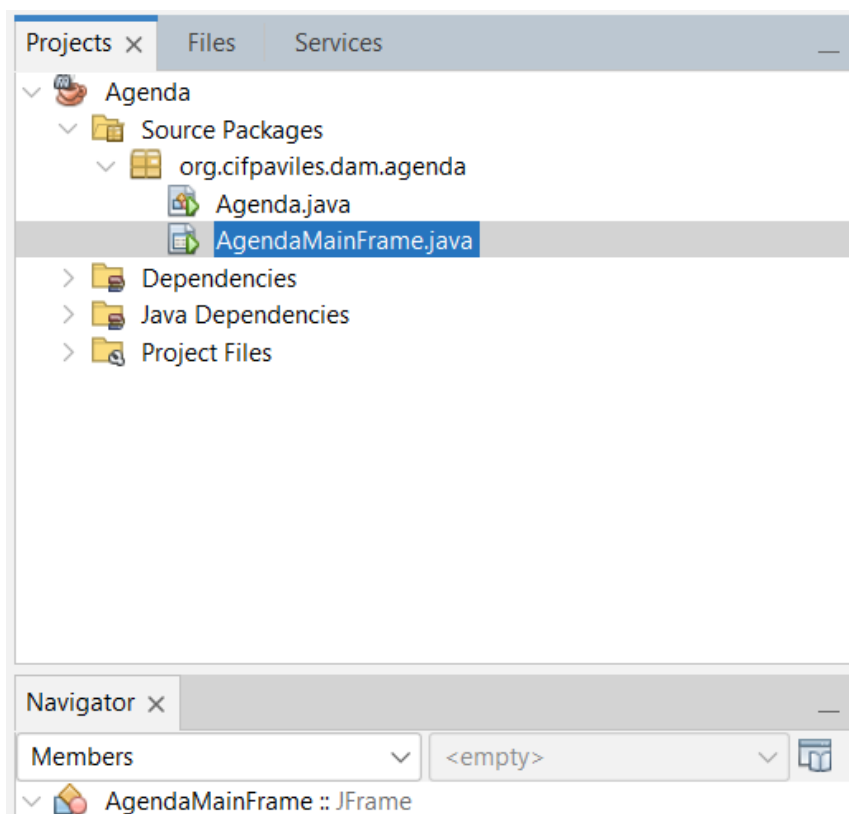
En la figura anterior puede visualizarse una disposición por defecto NetBeans en un proyecto Swing. A continuación, se muestran imágenes de las distintas áreas de dicho interfaz de usuario.

1. Menú principal y barra de herramientas.



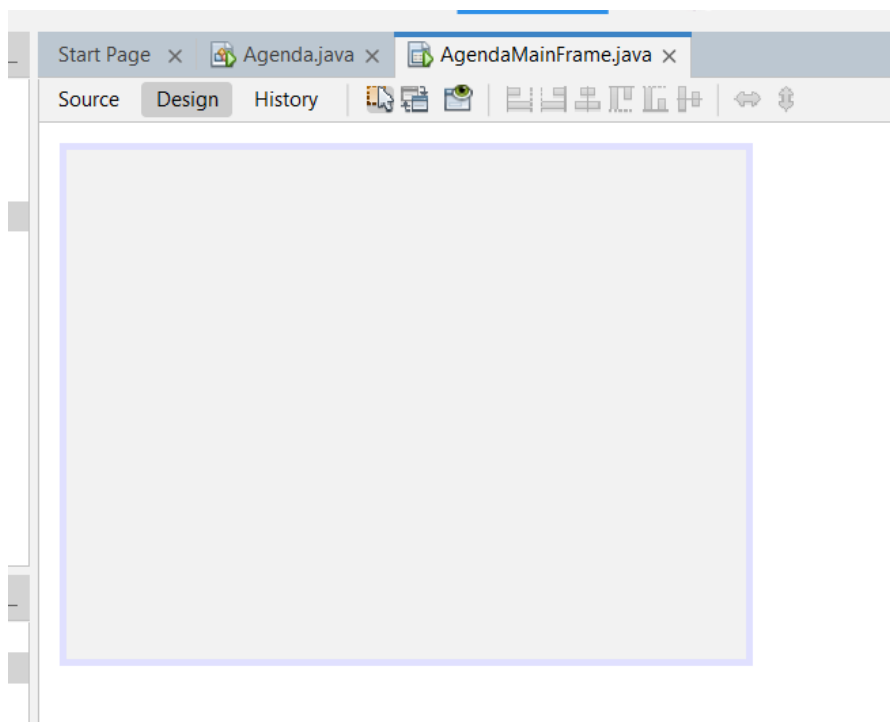
6. Menú principal y barra de herramientas

2. Estructura jerárquica de los archivos del proyecto. Dependiendo de su tipología, los ficheros se agruparán de una u otra forma. En un proyecto Java, los archivos se organizan en estructuras denominadas paquetes.



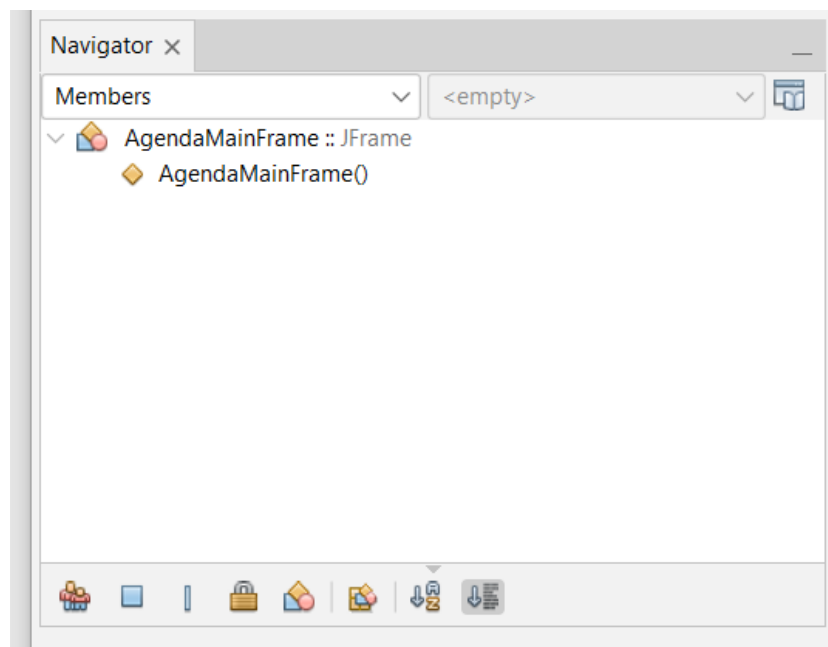
7. Archivos de un proyecto

3. Área principal de trabajo. Dependiendo del tipo de fichero, en caso de tener asociado un elemento de interfaz de usuario, dará la posibilidad de elegir entre vista de código o de diseño.



8. Área principal

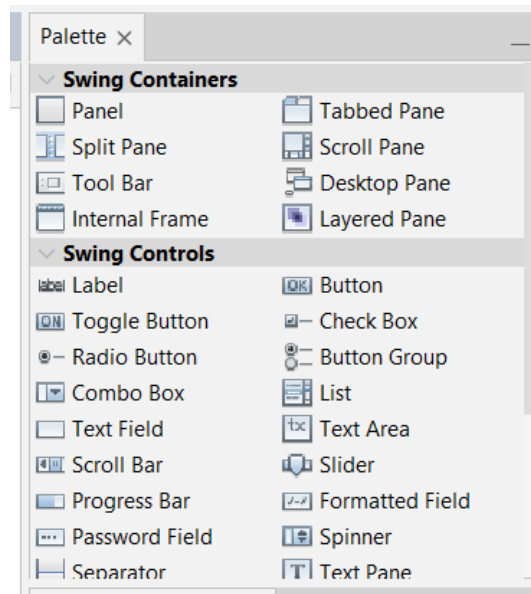
4. Propiedades y métodos de la clase seleccionada.



9. Navegador de propiedades y métodos

1.2.2. PALETA DE COMPONENTES

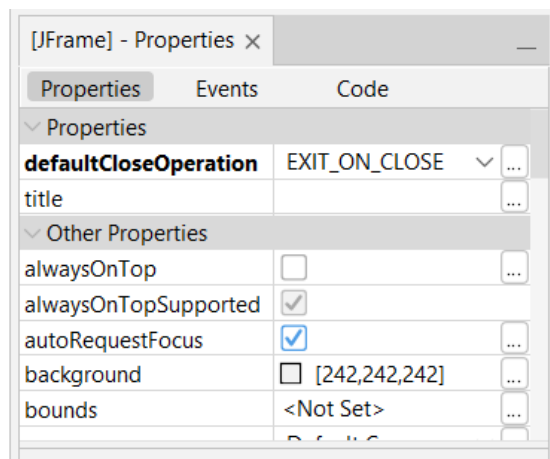
Los elementos que están relacionados directamente con la interfaz de usuario presentan ventanas adicionales que se incorporan al área de trabajo. Entre estas se puede destacar la paleta de componentes y la de propiedades del objeto que se verá en el siguiente apartado.



10. Paleta de componentes

1.2.3. EDITOR DE PROPIEDADES

Justo debajo de la paleta de componentes, se encuentra el editor de propiedades. Básicamente, muestra y permite editar las propiedades del componente visual que esté seleccionado. En este ejemplo, las propiedades de la imagen se refieren al JFrame creado anteriormente.



11. Editor de propiedades

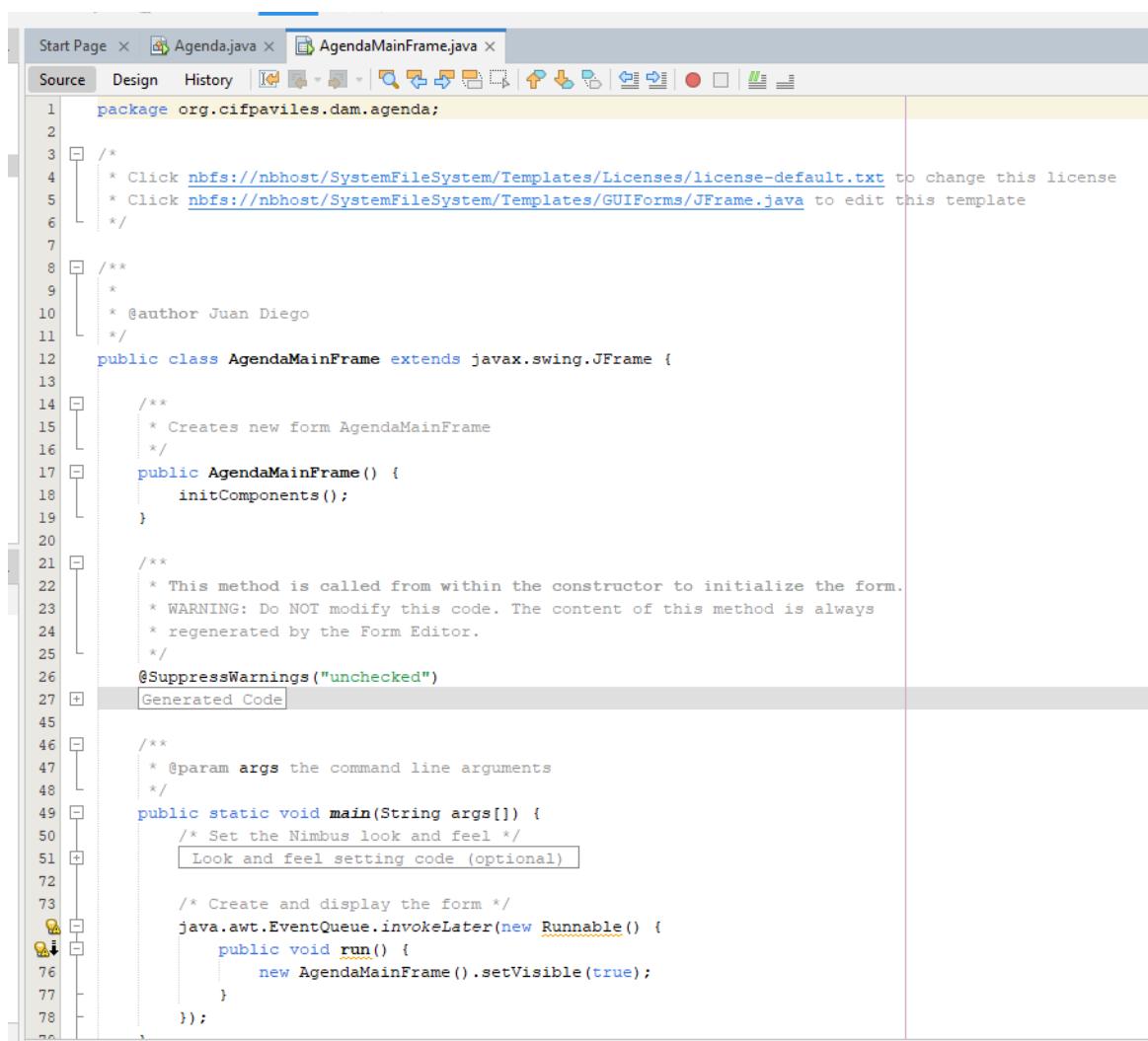
Uno de los ejemplos de la potencia de este editor consiste en ofrecer funcionalidades relacionadas con la propiedad que se quiere modificar. Por ejemplo, si se quiere modificar un color de cualquier control, aparecerá una paleta de colores al seleccionar el botón con los tres puntos a la derecha de la propiedad.

Como se puede ver en la imagen, este panel clasifica las distintas propiedades de un control en tres pestañas diferentes: Properties, Events y Code. Además de las propiedades como tal, en la pestaña Events se pueden consultar y codificar los distintos eventos que soporta el componente insertado. La pestaña de Código o Code permite, entre otras cosas, asignar un identificador para el elemento dentro del código.

1.2.4. EDICIÓN DE CÓDIGO

Como ya se vio en el apartado de área de diseño, al incluir un componente en un proyecto NetBeans, dentro del área principal, se puede ver dicho control en dos vistas: diseño y código. Resulta muy interesante consultar la vista de código de un control recién insertado para observar cómo el IDE autogenera la codificación del control en el lenguaje subyacente.

En la figura a continuación se puede ver la codificación de **AgendaMainFrame** como una clase que hereda de **javax.swing.JFrame** y su método **main** como punto de arranque de la aplicación.



```
1 package org.cifpaviles.dam.agenda;
2
3
4 /*
5  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
6  * Click nbfs://nbhost/SystemFileSystem/Templates/GuiForms/JFrame.java to edit this template
7  */
8
9 /**
10  *
11  * @author Juan Diego
12  */
13 public class AgendaMainFrame extends javax.swing.JFrame {
14
15     /**
16      * Creates new form AgendaMainFrame
17      */
18     public AgendaMainFrame() {
19         initComponents();
20     }
21
22     /**
23      * This method is called from within the constructor to initialize the form.
24      * WARNING: Do NOT modify this code. The content of this method is always
25      * regenerated by the Form Editor.
26      */
27     @SuppressWarnings("unchecked")
28     // Generated Code
29
30
31     /**
32      * @param args the command line arguments
33      */
34     public static void main(String args[]) {
35         /* Set the Nimbus look and feel */
36         Look and feel setting code (optional)
37
38         /* Create and display the form */
39         java.awt.EventQueue.invokeLater(new Runnable() {
40             public void run() {
41                 new AgendaMainFrame().setVisible(true);
42             }
43         });
44     }
45 }
```

12. Vista de código del JFrame

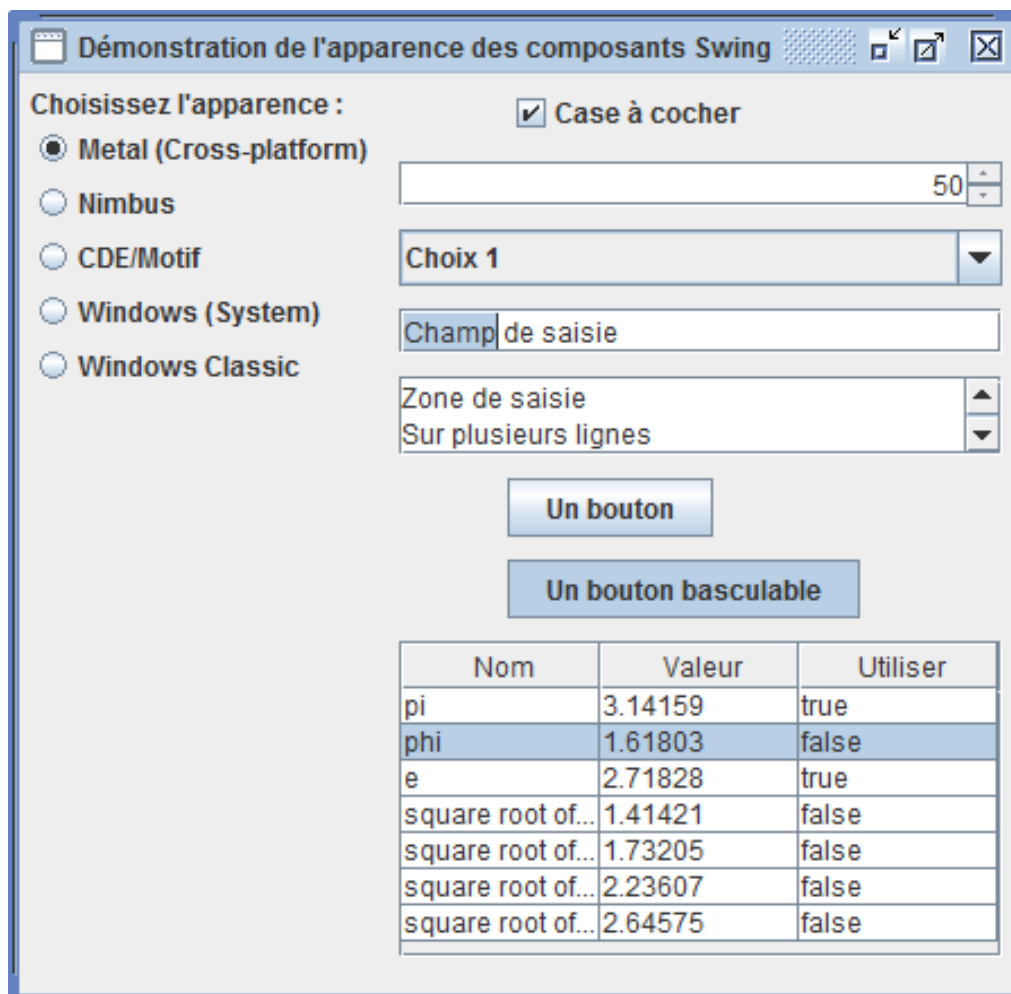
1.3. BIBLIOTECAS DE COMPONENTES NATIVAS Y MULTIPLATAFORMA

Existen multitud de bibliotecas orientadas al diseño de interfaces, algunas de ellas ligadas al sistema operativo anfitrión y otras propias del lenguaje de programación utilizado. A continuación, se van a enumerar algunas de las más utilizadas y, por ende, de interés para el desarrollador.

1.3.1. SWING

Biblioteca de Java creada a partir de otra anterior y ya obsoleta, AWT (Abstract Window Toolkit). Swing forma parte de la Java Foundation Classes (JFC) y busca resolver las deficiencias encontradas en AWT. En sus inicios se consideró que era lenta y repleta de errores, lo que provocó que IBM desarrollara una propia y específica del IDE Eclipse llamada [SWT](#) (Standard Widget Toolkit).

Swing, al contrario que AWT, presenta controles con gran funcionalidad apropiados para aplicaciones ricas. En cualquier caso, a pesar de las mejoras, aún presenta carencias en algunas áreas como el filtrado y organización de datos en controles de tipo tabla y árbol.

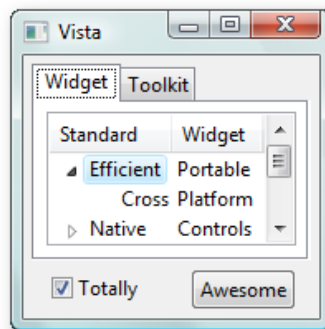


13. Interfaz de usuario basada en Swing

1.2.2. SWT

La biblioteca SWT, Standard Widget Toolkit, fue creada por la compañía IBM para el entorno de desarrollo Eclipse mejorando la versión existente de Swing en ese momento. Se puede considerar la tercera generación de bibliotecas de diseño de interfaces Java después de AWT y Swing.

Cabe destacar que no guarda una estrecha relación con Swing o AWT. Define una biblioteca de bajo nivel haciendo uso de widget nativos de la plataforma donde se ejecuta a través de [JNI](#) (Java Native Interface). Esto hace que las interfaces basadas en SWT no puedan ejecutarse en todas las plataformas. Uno de los inconvenientes de SWT es que la API proporcionada por la biblioteca es algo confusa en su uso y poco intuitiva para desarrolladores con poca experiencia sobre la misma.



14. Ejemplo de interfaz en SWT

Para poder utilizar SWT en Eclipse se puede seguir el tutorial de la URL a continuación:
<https://www.eclipse.org/swt/eclipse.php>

Es muy interesante la utilización de WindowBuilder en Eclipse, la cual permite trabajar en modo diseño, no solo en SWT, sino en otras bibliotecas como GWT o XWT, tal como se indica en el siguiente enlace:

<https://www.aluracursos.com/blog/interfaces-graficas-con-eclipse-windowbuilder>

El uso de WindowBuilder se ilustra en el siguiente clip:
<https://www.youtube.com/watch?v=bBa30KBIIIRI>

1.2.3. JAVAFX

[JavaFX](#) es la última biblioteca desarrollada por Java Oracle es una plataforma de código abierto la cual busca ser el estándar de facto. Enfocada en el desarrollo de aplicaciones [RIA](#) permite su uso en distintos dispositivos y plataformas. Las aplicaciones RIA buscan obtener el mismo funcionamiento en apariencia y funcionalidad que las aplicaciones tradicionales de escritorio buscando una mejor experiencia de usuario. El proceso de instalación de JavaFX en distintos IDEs y editores se muestra en la documentación oficial del Framework:
<https://openjfx.io/openjfx-docs/>

Es conveniente, además, la instalación de una herramienta que permita diseñar los interfaces de forma gráfica. Para ello existe la herramienta [JavaFX Scene Builder](#). En el siguiente tutorial se explica cómo puede integrarse con NetBeans:

<https://blog.idrsolutions.com/how-to-create-a-javafx-gui-using-scene-builder-in-netbeans/>

1.2.4. QT

[Qt](#) es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario, aunque también permite el desarrollo de programas sin interfaz gráfica como herramientas de consola y servidores.

Qt es utilizada en [KDE](#), [Konqueror](#), etc. Creado y desarrollado con C++ de forma nativa, usa una fuerte programación orientada a objetos, aunque permite usar otros lenguajes de programación. Su API cuenta con acceso a bases de datos, así como el uso de XML, gestión de hilos, soporte en red, manipulación de archivos, etc.

Para su uso con C++ y Visual Studio, QT proporciona varios ejemplos de uso muy sencillos de implementar:

<https://doc.qt.io/qtvstools/qtvstools-qt-widgets-application.html>

Para Python, existe una biblioteca de enlace con Qt llamada [PyQt](#) que permite crear aplicaciones de escritorio. Las definiciones de vistas se pueden realizar mediante archivos .ui, que son generados por la herramienta de diseño [Qt Designer](#) o directamente en código Python.

[Qt Jambi](#) es un empaquetado para Java de la biblioteca nativa Qt escrita en C/C++. Utiliza bibliotecas nativas, por lo que no es apta para todas las plataformas. Qt es muy potente y con gran aceptación y uso en la comunidad de desarrollo de software libre considerándose un estándar de facto para C++. Dispone de una gran cantidad de componentes prediseñados GUI y una API sencilla de usar.

1.2.5. OPENGL

[OpenGL](#) (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que producen gráficos 2D y 3D. Esta biblioteca contiene alrededor de 250 funciones que permiten especificar objetos y operaciones geométricas para definir elementos como puntos, líneas y polígonos los cuales sirven para la creación de gráficos en tres dimensiones.

Su ámbito de utilización es muy amplio, desde la creación de videojuegos hasta simuladores de vuelo en ingeniería aeroespacial pasando por el Diseño Asistido por Ordenador (CAD).



15. OpenGL

Existen implementaciones de OpenGL para distintos lenguajes de programación y/o Frameworks. Por ejemplo, para .NET, una de las implementaciones posibles es Silk.NET.

En este repositorio pueden encontrarse distintos ejemplos de uso para OpenGL sobre C#:
<https://github.com/dotnet/Silk.NET/tree/main/examples/CSharp>

Para poder llevar a cabo estos ejemplos, basta con crear un proyecto básico de escritorio y añadir las referencias necesarias mediante el gestor de paquetes NuGet.

1.2.6. DIRECTX

DirectX es una colección de APIs desarrolladas para facilitar las tareas relacionadas con sistemas multimedia, especialmente la programación de juegos y vídeo dentro del sistema operativo Windows.

La última versión a la fecha de este documento es la número 12, la cual incluye optimizaciones específicas para videojuegos aprovechando las ventajas de las arquitecturas de procesamiento de varios núcleos.

Alguna de las APIs más relevantes de DirectX son:

- Direct Graphics. Sirve para dibujar imágenes en 2D y en 3D.
- Direct3D. Se utiliza para el procesamiento y generación de gráficos en 3D.
- DirectCompute: Se utiliza para manejar cientos o miles de hilos de procesamiento, especialmente en procesadores de núcleos masivos.

1.2.7. TKINTER

Tkinter es un módulo incorporado en Python que proporciona herramientas para crear interfaces gráficas de usuario. Es simple y una opción muy popular y ampliamente utilizada para desarrollar aplicaciones de escritorio con Python. Tkinter no utiliza XML o JSON para definir vistas; todo se realiza directamente en código Python.

Para diseñar interfaces de forma gráfica, existen herramientas web como [Visual TK](#) que permiten obtener el código Python correspondiente a una interfaz sencilla.

1.2.8. ELECTRON

[Electron](#) es un Framework que permite crear aplicaciones de escritorio multiplataforma utilizando tecnologías web, como HTML, CSS y JavaScript. Es ampliamente utilizado para desarrollar aplicaciones de escritorio de código abierto como Visual Studio Code, Slack y Discord. Electron se puede usar con una serie de bibliotecas que, a priori, son para desarrollo web como:

- [Vue.js](#): Framework de JavaScript ampliamente utilizado para crear interfaces de usuario interactivas y dinámicas. Se puede combinar Vue.js con Electron para construir la interfaz de usuario de tu aplicación de escritorio.
- [React](#): Framework de JavaScript muy popular que proporciona una biblioteca de componentes reutilizables que facilita la creación de interfaces interactivas. Además, existen bibliotecas de componentes como [Material-UI](#) que implementa los principios de diseño del sistema Material Design de Google.
- [Angular](#): Framework de JavaScript desarrollado por Google que se puede utilizar junto con Electron para construir aplicaciones de escritorio. Angular proporciona una estructura sólida y una amplia gama de características para desarrollar aplicaciones complejas.
- [jQuery](#): Aunque no es un Framework completo, jQuery es una biblioteca de JavaScript muy utilizada para simplificar la manipulación del DOM y realizar tareas comunes en el desarrollo web. Se puede combinar con Electron para construir aplicaciones de escritorio.
- [Bootstrap](#): Framework de CSS y JavaScript que ofrece una variedad de componentes y estilos prediseñados para facilitar el diseño y la creación de interfaces de usuario responsivas. Se puede utilizar junto con Electron para mejorar la apariencia y el diseño de una aplicación de escritorio.
- [Bulma](#): Similar a Bootstrap, Bulma es otro Framework de CSS que proporciona estilos y componentes para crear interfaces de usuario modernas y receptivas. Es otra opción para mejorar la apariencia de tu aplicación de escritorio con Electron.

Otro Framework similar a Electron es [NW.js](#) el cual permite crear aplicaciones de escritorio multiplataforma utilizando tecnologías web.

1.2.9. GTK+

[GTK+](#) es un conjunto de bibliotecas multiplataforma para desarrollar interfaces gráficas de usuario bajo el estándar de software libre. Dichas bibliotecas usan un modelo de desarrollo basado en objetos. Con GTK+ se ha desarrollado la interfaz GNOME de Linux. Aunque nativamente se usa con C++, se han desarrollado enlaces para distintos lenguajes de programación como JavaScript, Python, Rust, Scheme o Vala (un lenguaje similar a C#).

Para Java, aunque se han desarrollado varios enlaces, se han ido quedando por el camino cesando su desarrollo. Uno de los que sigue vigente es [Java-GTK](#), biblioteca que usa [JNA](#) (Java Native Access) para acceder a bibliotecas GTK.

Dentro del proyecto [Mono](#), el cual corresponde a una implementación libre de .NET, se encuentra [GtkSharp](#), que sirve como enlace para desarrollar aplicaciones GTK en este Framework.

1.4. SISTEMAS DE CONTROL DE VERSIONES

Los sistemas de control de versiones (VCS – Version Control System) también llamados de control de código fuente (SCM – Source Content Management) son herramientas que permiten gestionar, compartir y mantener un historial de cambios de un código fuente. Es un sistema que permite seguir con precisión la evolución del contenido de los archivos. Este tipo de sistema se emplea mucho en el desarrollo informático, pero no está limitado a este, ya que cualquier actividad que utiliza archivos legibles podrá ser seguida por un VCS. Por ejemplo, los sistemas Wiki (como Wikipedia) utilizan un VCS. También es posible citar el conocido sitio de asistencia informática Stack Overflow, que sigue las versiones de las preguntas y respuestas.

El seguimiento de los archivos está encaminado en la protección de los cambios. En efecto, cuando se termina de editar un archivo, se indica al VCS que ha terminado el trabajo y la razón para hacerlo. Las modificaciones efectuadas en el archivo son registradas por el sistema.

A partir de ahí, se añade una nueva línea (conocida como revisión) en su historial. El término revisión (llamado también commit en inglés) define un conjunto de modificaciones que son registradas por el sistema.

Los beneficios de ese sistema son múltiples y van mucho más allá del seguimiento puro y simple de un proyecto.

1.4.1. TIPOS DE SISTEMAS DE CONTROL DE VERSIONES

Según la arquitectura de los sistemas de gestión de control de versiones, se pueden distinguir los siguientes tipos de sistemas, lo que además permite ver cómo han ido evolucionando con el tiempo:

- **Sistemas de gestión de versiones locales.** Los sistemas de almacenamiento local son sistemas donde todos los usuarios comparten el mismo sistema de archivos. El sistema conserva los archivos originales y todas las modificaciones efectuadas a continuación. De esta forma, se puede recuperar los archivos en cualquier momento que se haya utilizado el VCS. Los sistemas de almacenamiento local aparecieron en 1972 mediante SCCS (Source Code Control System). En 1982, se publica GNU RCS (Revision Control System) el cual aporta, por su parte, una importante mejora de rendimiento en comparación con SCCS gracias a otra forma de almacenar los archivos. En efecto, RCS almacena las últimas versiones de los archivos y la diferencia con el archivo anterior, lo que permite tener la última versión de los archivos con mayor rapidez. Este tipo de solución no permite a los colaboradores trabajar juntos si no utilizan el mismo sistema de archivos. La colaboración no es fácil y no permite ninguna movilidad. Con estos sistemas solo se mantiene la versión de los archivos, cada uno de los cuales posee su propia historia de versión. Después de haber sido muy utilizado en el mundo del libre, RCS poco a poco fue sustituido por CVS.

- Sistemas de gestión de versiones centralizados. En este tipo de sistema, todos los datos del seguimiento de versión se almacenan en un servidor. A partir de estos sistemas, el administrador del VCS puede limitar los privilegios de algunos usuarios. Entre el software de gestión de versiones centralizado, se puede considerar CVS (Concurrent Versions System) y Subversion (o SVN). CVS apareció en 1990. Fue el primero en utilizar un sistema centralizado. Tradicionalmente se ha usado mucho en el mundo del software libre y se substituyó por Subversion en 2000. Este último nació del deseo de la empresa CollabNet de mejorar algunos aspectos de CVS. Los puntos de mejora más significativos fueron:
 - Los commits son atómicos. Un commit corresponde a una serie de modificaciones. A partir de Subversion, los commits afectan a varios archivos.
 - Los archivos renombrados o movidos conservan su histórico. Con los antiguos sistemas, un movimiento o cambio de nombre del archivo equivalía a decir “El archivo ya no tiene nada que ver con lo que era”. Estos sistemas tenían, sin embargo, una importante limitación: la dependencia de los clientes conectados al servidor. En efecto, si no se puede acceder al servidor, es imposible guardar los cambios o recuperar algo. Esta limitación exige que los administradores garanticen una alta disponibilidad del servidor de versiones para no detener a los desarrolladores.

Otro sistema centralizado que da soporte a un sistema de control de versiones centralizado es Microsoft Team Foundation Server (TFS) el cual se utiliza fundamentalmente con proyectos .NET desarrollados con Visual Studio.

- Sistemas de gestión descentralizados. Este tipo de sistema es el más reciente. Responde en primer lugar al principal defecto de los sistemas centralizados: la dependencia de los clientes del servidor. En concreto, cuando se utiliza un sistema descentralizado y deseamos trabajar sobre un proyecto contenido en el servidor, se clona en primer lugar este proyecto en el equipo local. Esta etapa de clonación copia íntegramente el proyecto y su histórico localmente. Esto permite al desarrollador ser autónomo, no requiere estar conectado a una red para guardar sus cambios y consultar el histórico. Por supuesto, se necesita estar conectado al servidor para recibir las actualizaciones del proyecto o compartir sus modificaciones. Este tipo de sistemas se conoce como DVCS (Decentralized VCS) y comenzó a usarse más o menos por 1997. El 20 de abril de 2005, Linus Torvalds anuncia la salida de un nuevo sistema de gestión de versiones distribuido: Git. El desarrollo de Git fue muy rápido, ya que la comunidad necesitaba un nuevo VCS para el mes de julio de 2005. Cuando Linus comenzó a trabajar en Git, definió una serie de necesidades a las que el nuevo sistema debería responder:
 - Un sistema libre: no se quiso ser dependiente de un sistema propietario y se quería disminuir las tensiones en el seno de la comunidad del proyecto Linux.
 - Un sistema eficaz: porque el proyecto Linux exigía mucho de fusiones de ramas y creaciones de parches de actualización, se quería un sistema eficaz para no penalizar el proyecto debido a fusiones difíciles.
 - Un sistema distribuido para conservar las ventajas de este tipo de sistema.
 - Un sistema compatible con los protocolos existentes: la compatibilidad de un nuevo sistema con los protocolos HTTP, FTP y SSH aceleraría el desarrollo y la

compatibilidad del nuevo sistema. Además, un servidor que emulaba el CVS fue integrado en Git para facilitar el trabajo de los desarrolladores que usan CVS.

- Un sistema adaptado a proyectos importantes: Git fue diseñado para poder gestionar proyectos importantes con muy buenos resultados.
- Un sistema basado en hashes: un hash (o etiqueta) es un valor (a menudo representado en forma hexadecimal). Se calcula a partir de otro valor, como una cadena de caracteres o un archivo. Dos archivos con un contenido diferente tendrán una etiqueta diferente (salvo en algunos casos muy raros). Este sistema de archivos basados en los hashes permitirá a Git detectar la menor modificación en un archivo e identificar un archivo a partir de su contenido.

Git ha evolucionado y seguido ganando popularidad. Esta popularidad se debe también a un servicio en línea nacido en 2008: GitHub. Es una plataforma que ofrece la posibilidad de almacenar sus proyectos en forma de repositorios remotos en los servidores de la empresa. Este servicio ofrece una funcionalidad para trabajar en equipo y colaborar eficazmente entre desarrolladores. El uso de GitHub para los proyectos libres (y sin restricción de acceso) es gratuito. En cambio, cuando una empresa desea utilizar GitHub para proyectos privados, está obligada a pagar el servicio. Con el mismo tipo de servicio, existe Bitbucket, basado en otro modelo de negocio diferente al de GitHub.

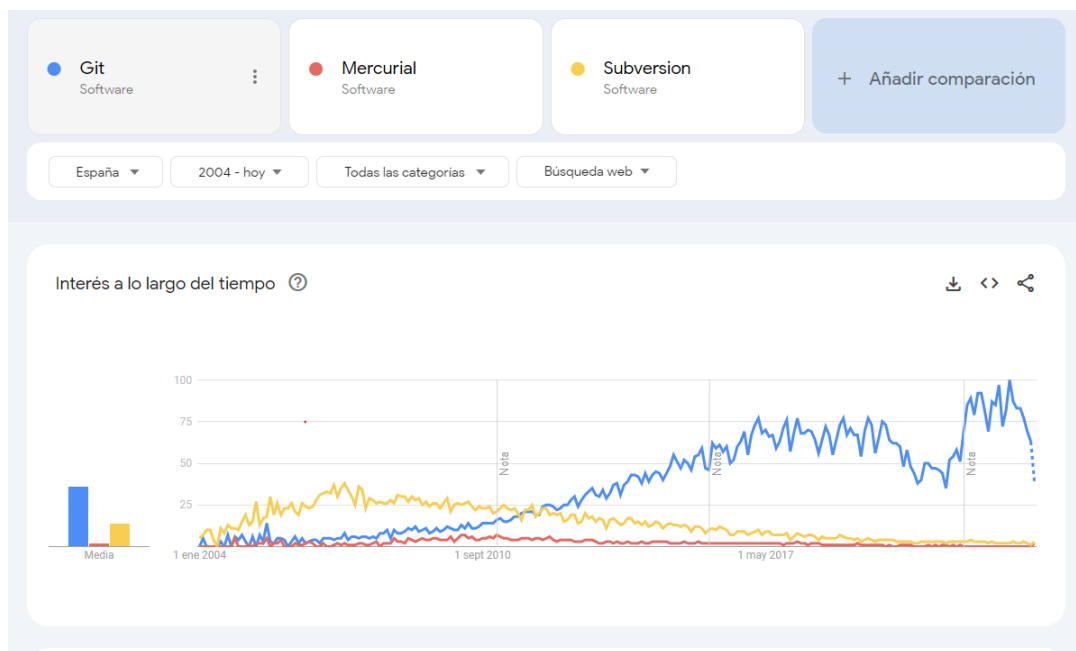
1.4.2. INTRODUCCIÓN A GIT

Git es un sistema libre de gestión de versiones, lo que implica que no puede haber limitaciones contractuales sobre su uso. Sea cual sea el número de colaboradores, de proyectos o actualizaciones, Git siempre será gratuito.

Git posee también todos los beneficios derivados de los sistemas de gestión de versiones descentralizados. Es decir, que es posible trabajar en modo desconectado con su repositorio manteniendo las características avanzadas de colaboración cuando se está conectado.

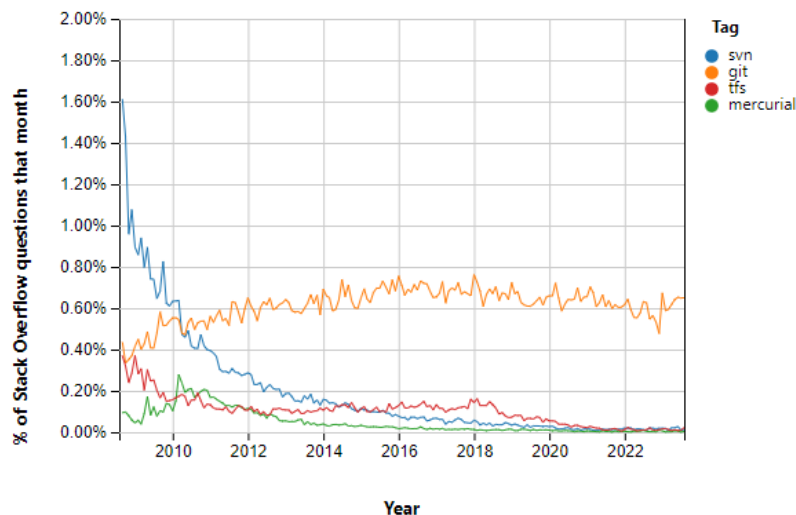
Git también tiene otra ventaja importante: su rendimiento. Git incorpora muchas herramientas internas para mejorar su rendimiento (en particular para el uso de las ramas).

Una última ventaja muy importante: Git es actualmente el sistema más popular y gana en popularidad cada año. Es difícil contar con estadísticas precisas sobre los sistemas de gestión de versiones instalados y usados en las empresas. Sin embargo, es posible juzgar la popularidad de algunos términos utilizados en búsquedas en Google gracias a Google Trends. Por consiguiente, es posible tener una idea del volumen de búsquedas de determinadas palabras clave. He aquí el gráfico comparativo de Git, Mercurial y Apache Subversion.



16. Comparativa de CVS por tendencia (Google Trends).

Stackoverflow también ofrece una herramienta para comparar la popularidad de diferentes términos. La siguiente imagen muestra los resultados de la comparación para Git, Mercurial, Subversion (SVN) y Team Foundation Server (TFS). Los resultados se expresan en el eje de ordenadas (y) y como porcentaje de preguntas para cada mes entre 2009 y 2023.



17. Comparativa de consultas en Stack Overflow

1.4.3. INSTALACIÓN DE GIT

Git no fue diseñado originalmente para funcionar en Windows. Para usar Git eficazmente en Windows, hay que hacerlo a través de una interfaz de línea de comandos que simule un entorno UNIX. Esta interfaz se llama Cygwin, lo desarrolló RedHat y viene directamente con el instalador de Git.

Para descargar el ejecutable de instalación, hay que ir al sitio oficial: <https://git-scm.com>. Una vez más, usando la información enviada por el navegador, se ofrece la descarga del ejecutable que corresponde al sistema operativo deseado:



18. Descarga de Git.

El proceso de instalación contiene elementos interesantes como, por ejemplo, con qué editor se va a trabajar con Git (una opción interesante sería Visual Studio Code), la rama inicial del proyecto (máster es la opción por defecto dejando que Git “decida”), qué línea de comando se va a usar (en este caso, también interesa que sea la de Windows, cmd.exe).

En el proceso de instalación también se permite configurar la gestión de los saltos de línea. Hay que saber que los sistemas UNIX y los sistemas Windows no utilizan los mismos caracteres para realizar un salto de línea. En efecto, Windows usa el modo CRLF (Carriage Return Line Feed), es decir, que, en un archivo de texto en bruto, Windows inserta entre dos líneas un retorno de carro y luego un salto de línea. Los sistemas UNIX utilizan el modo LF, es decir, que solamente usan un salto de línea entre dos líneas. Git permite realizar ciertas acciones sobre los saltos de línea. Si un usuario que trabaja en sistemas UNIX crea un archivo y lo añade al repositorio, después de que un usuario de Windows abra el archivo, este se percibirá como si hubiera sido totalmente modificado, ya que cada salto de línea habrá sido convertido por Windows. Git propondrá varios modos para permitir que los diferentes sistemas operen entre sí con él.

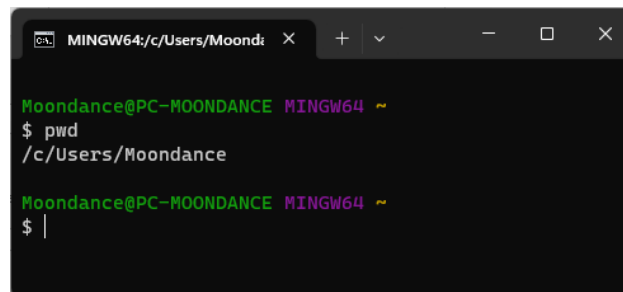
El primer modo es el modo predeterminado. Cuando Git grabe los archivos en el sistema, Windows convertirá los saltos de línea UNIX (LF) en saltos de línea Windows (CRLF). Git hará lo contrario cuando grabe el contenido de los archivos en el repositorio. Este es el modo recomendado para los usuarios de Windows.

En el segundo modo, Git no modifica los saltos de línea de los archivos que se graban en el sistema de archivos. A la inversa, cuando el usuario haga commit de los archivos en el repositorio, los saltos de línea de tipo Windows se convertirán en saltos de línea UNIX.

El último método no efectúa ninguna conversión de salto de línea. Normalmente, si todos los usuarios de Git utilizan el mismo sistema operativo, este tipo no plantea ningún problema en particular. Esto es lo mismo que prescindir de muchas posibilidades de restringir el uso del

repositorio a un solo tipo de sistema. Por esta razón se desaconseja encarecidamente el uso de esta opción cuando se administra un proyecto desarrollado en sistemas Windows y Unix.

Una vez que Git se instala, se crean dos accesos directos en la lista de programas. Un acceso directo llamado Git Bash redirige a una interfaz de línea de comandos gestionada por Cygwin. Otro llamado Git GUI lleva a una interfaz que permite utilizar Git. En los próximos ejemplos, se usará Git Bash, que permite realizar muchas más acciones y que también permite comprender mejor el funcionamiento de Git. Después de pulsar el acceso directo Git Bash, aparece una interfaz de línea de comandos. Esta interfaz permite usar Git, así como un gran número de comandos bash. Por ejemplo, cuando se utiliza el comando `pwd`, que en UNIX permite obtener la ruta actual, la interfaz muestra la siguiente salida:



```
MINGW64:/c/Users/Moondance
Moondance@PC-MOONDANCE MINGW64 ~
$ pwd
/c/Users/Moondance
Moondance@PC-MOONDANCE MINGW64 ~
$ |
```

19. Comando `pwd` en Git Bash

Hay que tener en cuenta que la sintaxis en este caso es UNIX y, como puede verse en el lanzamiento de la orden, las rutas también siguen ese formato en el que los directorios se dividen por barras invertidas en lugar de contrabarras como en Windows.

Es interesante consultar el sitio de [Cygwin](http://cygwin.com) para ver sus particularidades y la potencia de esta herramienta, pero lo primero que hay que hacer cuando se ha instalado una utilidad como git, es comprobar su versión, en este caso, lanzando la siguiente orden:

```
git --version
```

Esta orden se puede lanzar en el bash de Cygwin, pero si se ha configurado para poder usar la línea de comandos de Windows, también se puede ejecutar en el Símbolo de sistema e incluso en el Windows Powershell.

1.4.4. COMANDOS GIT

Git propone una serie de comandos que se dividen en dos categorías:

- Los comandos de porcelana (porcelain command en inglés). Comandos de alto nivel. Son los que se utilizan a diario y para los que Git propone una serie de controles. Por ejemplo, `git commit` y `git add` son comandos de porcelana.
- Los comandos de fontanería (plumbing commands en inglés). Comandos de bajo nivel. Manipulan directamente la información que constituye el núcleo de Git. Estos comandos pueden ser destructivos para el repositorio y suelen ser aplicados por usuarios que dominan Git.

Por de pronto, lo primero que hay que hacer después de instalar git es configurar el repositorio local. De hecho, hay dos parámetros básicos de configuración para el repositorio: dirección de correo del usuario y su nombre. Se hace lanzando las dos órdenes siguientes:

```
git config --global user.name "Nombre Apellido"
git config --global user.email email@dominio.extension
```

Para cualquier nuevo proyecto que se desee versionar, se requiere crear un nuevo repositorio; luego será en este depósito donde Git almacenará toda la información. Para esto, hay que dirigirse al directorio raíz del proyecto. Por ejemplo, si el proyecto es un sitio web simple, se puede suponer que el expediente que contendrá el proyecto se llamará www. En este caso, el directorio que va a contener el código del proyecto se llama repositorio. Hay que dirigirse a él y ejecutar el comando siguiente:

```
git init
```

Git confirma la creación del repositorio con el siguiente mensaje:
Initialized empty Git repository in C:/path/to/www/.git/

Esta acción creará un directorio llamado .git en la raíz del proyecto. En la mayoría de los exploradores de archivos, este directorio está oculto. Se puede usar la línea de comandos para ver su contenido con el comando ls -la de los sistemas Linux o Mac OS, y también en Windows utilizando la herramienta Cygwin.

De forma predeterminada, la creación de un repositorio crea automáticamente una rama master. Para cambiar el nombre de esta rama, es posible establecer la opción de configuración init.defaultBranch como en el ejemplo siguiente (dentro de otro directorio distinto al anterior):

```
git config --global init.defaultBranch main
touch README.md
git init
```

Lo que da como salida:
Initialized empty Git repository in path.to/www/main/.git/

Se añade un fichero README y se hace el primer commit:

```
git add README.md
git commit -m "Añadido README"
```

Como salida se obtendrá:
[main (root-commit) e7e85f8] Añadido README
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 README.md

El siguiente comando muestra la rama en la que se está:

```
git branch
```

De esta forma, se tiene un primer repositorio local con un fichero. Cada vez que se desee añadir un fichero o varios, simplemente basta con hacer un add y cuando ese lote de ficheros corresponda a un commit, lanzar el comando correspondiente. Por ejemplo, si se desea añadir todos los ficheros del directorio, se lanzaría:

```
git add *
```

1.4.5. SINCRONIZACIÓN CON REPOSITORIO REMOTO

Obviamente, las operaciones que se han realizado hasta ahora no salen del ámbito local de la máquina en la que se han realizado, pero la potencia de Git reside en su carácter distribuido. Ello implica que varios desarrolladores estén en sus repositorios locales enviando y obteniendo datos de un repositorio remoto. Existen varias soluciones que ofrecen repositorios remotos. Las más conocidas son [GitHub](#) y [BitBucket](#).

El proceso es sencillo: crear una cuenta, por ejemplo, en GitHub y dar de alta un nuevo repositorio.

A continuación, hay que añadir al repositorio local la URL del remoto. Se hace con la siguiente orden:

```
git remote add origin URL-del-repositorio-remoto
```

La subida de cambios al repositorio se realiza con el comando:

```
git push origin main
```

El sistema ahora pedirá usuario y contraseña de GitHub, pero al autenticarse, lanzará un error ya que desde mediados del 2021 no se permite el acceso con contraseña sino mediante un token personal. Para crear dicho token basta con seguir esta URL propia de GitHub:

<https://docs.github.com/es/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>

1.5. PROGRAMACIÓN ORIENTADA A OBJETOS

En este apartado se verán los principales conceptos relacionados con la Programación Orientada a Objetos. Se estudiarán los conceptos de clase, propiedades y métodos entre otros. Estas son las ideas más básicas que quien trabaja en POO debe comprender y manejar constantemente.

1.5.1. CLASES

El ser humano tiende a agrupar seres o cosas -objetos- con características similares en grupos -clases-. Así, aun cuando existen por ejemplo multitud de vasos diferentes, se puede reconocer un vaso en cuanto se ve, incluso aun cuando ese modelo concreto de vaso no se haya visto nunca. El concepto de vaso es una abstracción de algo real. Cada clase posee unas cualidades que la diferencian de las otras. Así, por ejemplo, los vegetales se diferencian de los minerales -entre otras muchas cosas- en que los primeros son seres vivos y los minerales no. De los animales se diferencian en que las plantas son capaces de sintetizar clorofila a partir de la luz solar y los animales no. Como vemos, el ser humano tiende, de un modo natural a clasificar los objetos del mundo que le rodean en clases.

Por ejemplo, en la clase felinos se tienen varias subclases (géneros en palabras de los biólogos): león, tigre, pantera, gato, etc. Cada una de estas subclases tiene características comunes (por ello se identifican a todos ellos como felinos) y características diferenciadoras (que permiten distinguir a un león de una pantera). Sin embargo, ni el león ni la pantera en abstracto existen, sino que lo que hay son leones y panteras particulares, pero se ha realizado una abstracción de esos rasgos comunes a todos los elementos de una clase para llegar al concepto de león, de pantera, o más generalmente, de felino. Se puede ver que las clases superiores son más generales que las inferiores y cómo, al ir bajando por este árbol, se va definiendo cada vez más (dotando de más cualidades) a las nuevas clases.

Hay cualidades que ciertas clases comparten con otras, pero no son exactamente iguales en las dos clases. Por ejemplo, la clase hombre, también deriva de la clase vertebrado, por lo que ambos poseen columna vertebral, sin embargo, mientras que en la clase hombre se halla en posición vertical, en la clase león la columna vertebral está en posición horizontal.

Una clase es un concepto en tiempo de compilación que representa un objeto en tiempo de ejecución. Por ejemplo, si se tiene una clase llamada Coche, un ejemplar particular de esta clase sería un Escarabajo Wolkswagen del 66. Estos ejemplares (Escarabajo Wolkswagen del 66) de una clase (Coche) son conocidas como objetos. Para definir una clase en Java, se necesitaría un código similar al siguiente:

```
public class Coche {  
    // Variables miembro  
    // Métodos miembro  
}
```

1.5.2. PROPIEDADES

Coche es ahora una clase vacía. Para poder utilizarla en un programa, es preciso añadir algunos campos a la clase. Un campo puede ser una variable miembro o una método miembro. Para declarar variables miembro, todo lo que hay que hacer es identificar la variable por su tipo y nombre en la definición de la clase, como se muestra a continuación:

```
public class Coche {  
    // Estas son variables miembro  
    String fabricante;  
    String modelo;  
    int anyo;  
    int plazas;  
}
```

En este ejemplo, se ha extendido la clase Coche con las variables de tipo cadena para el fabricante y el modelo, y variables enteras para el año en el que fue construido y el número de plazas que tiene. De esta definición de clase es posible crear ejemplares u objetos en tiempo de ejecución. Una de las ventajas principales de la programación orientada a objetos es la encapsulación. Encapsulación es la posibilidad de definir clases las cuales ocultan sus detalles de implementación a las otras clases, exponiendo solamente sus interfaces públicos a esas otras clases. El soporte para la encapsulación en Java viene dado por tres palabras clave: public, private y protected. Cuando se define una clase, estos modificadores de acceso a campos son usados para controlar quien tiene acceso al campo indicado en cada clase. Si se declara un campo como public, se está indicando que es enteramente accesible desde todas las demás clases. Continuando con el ejemplo del Coche, para declarar todos sus campos como públicos, se haría lo siguiente:

```
public class Coche {  
    public String fabricante;  
    public String modelo;  
    public int anyo;  
    public int plazas;  
}
```

Por supuesto, declarar todo como público no garantiza exactamente el objetivo de la encapsulación porque permite que todas las demás clases accedan directamente a las variables en la clase Coche. Considérese lo que pasaría si se necesitara crear un ejemplar de esta clase para un Mustang de mitad de 1964. Como el año solo puede contener valores enteros, debería ser cambiado al tipo float, de forma que pudiera contener 1964.5. Si el código en otras clases accede directamente al año, éste puede dar un error.

Para restringir el acceso a un campo, se usa la palabra clave private. Una clase no puede acceder a los campos privados de otra clase. Supóngase que la clase Coche se va a usar en una aplicación de venta de coches usados.

En este caso se debería definir Coche como sigue de forma que oculte el coste del coche a los potenciales compradores:

```
public class Coche {  
    public String fabricante;  
    public String modelo;  
    public int anyo;  
    public int plazas;  
    private float coste;  
}
```

Finalmente, la palabra clave `protected` se usa para indicar que los campos son accesibles dentro de la clase actual y todas las clases derivadas de ellas, pero no para otras clases. La cualidad de derivar una clase de otra será explicada posteriormente.

1.5.3. MÉTODOS

Además de las variables miembro, la mayoría de las clases tendrán también funciones miembro o métodos. Los métodos, al igual que las variables son campos, y por ello pueden ser controlados con los modificadores `public`, `private` y `protected`. Un método es declarado acorde con la siguiente sintaxis, en la cual los elementos encerrados entre corchetes “[...]” son opcionales:

```
[modificadores] tipo nombre{  
    //cuerpo del método  
}
```

Los modificadores de método se muestran en la siguiente tabla:

Modificador	Propósito
public	Accesible fuera de la clase en la que fue declarada.
protected	Accesible por la clase en la cual es declarado y por las subclases de esa clase
private	Accesible solo por la clase en la cual es declarado
static	Un método de la clase independiente de las instancias particulares de esa clase
abstract	No implementado en esta clase
final	No puede ser sobrescrito en las subclases
native	Una implementación dependiente de la plataforma del método en otro lenguaje, típicamente C o ensamblador.
synchronized	Usado para indicar un método crítico que bloqueará el objeto para prevenir la ejecución de otros métodos mientras el método sincronizado se ejecuta.

El tipo de una declaración de un método puede ser uno de los primitivos (por ejemplo, int, float, char), otra clase, o void. Un tipo void indica que en la llamada del método no se va a devolver nada.

Después de haberle dado nombre al método, se da una lista de excepciones que pueden ser lanzadas por el método. Si no hay excepciones que puedan ser lanzadas por el método, esta lista no es necesaria.

En el siguiente código, se puede ver un ejemplo de método añadido a la clase Coche:

```
public class Coche {  
    public String fabricante;  
    public String modelo;  
    public int anyo;  
    public int plazas;  
    public float CalcularPrecioVenta() {  
        return coste * 1.5;  
    }  
    private float coste;  
}
```

En este caso, la clase Coche tiene una función miembro pública, CalcularPrecioVenta. El método devuelve un float, y el cuerpo del método calcula este valor de retorno. Para calcular el precio de venta de un coche, la variable miembro privada coste se multiplica por 1.5, reflejando un incremento del 50% sobre la cantidad por la cual el coche fue adquirido.

Pero los métodos más comunes en POO en Java son los **getters** y los **setters**, los cuales tienen como misión, devolver o asignar valores a las variables miembro privadas. Si en la clase anterior, se declaran las variables miembro como privadas, se encapsularían en métodos get y set de la siguiente forma:

```
public class Coche {  
    private String fabricante;  
    private String modelo;  
    private int anyo;  
    private int plazas;  
    private float coste;  
    public String getFabricante() {  
        return fabricante;  
    }  
    public void setFabricante(String fabricante) {  
        this.fabricante = fabricante;  
    }  
    public String getModelo() {  
        return modelo;  
    }  
    public void setModelo(String modelo) {  
        this.modelo = modelo;  
    }  
}
```

```
public int getAnyo() {  
    return anyo;  
}  
public void setAnyo(int anyo) {  
    this.anyo = anyo;  
}  
public int getPlazas() {  
    return plazas;  
}  
public void setPlazas(int plazas) {  
    this.plazas = plazas;  
}  
public float getCoste() {  
    return coste;  
}  
public void setCoste(float coste) {  
    this.coste = coste;  
}  
}
```

En este caso, `this` representa a un puntero del mismo objeto con el que se está trabajando. Por otra parte, es interesante conocer que la encapsulación de las variables miembro se puede hacer de forma sencilla en cualquier entornos de desarrollo.

1.5.4. BIBLIOTECAS DE CLASES

Las bibliotecas de clases son agrupaciones funcionales de dichos elementos para proporcionar una API (Application Programming Interface). A partir de ahí, cada lenguaje de programación o Framework tiene su estructura interna para proporcionar las clases necesarias al personal de desarrollo. La meta principal de estas bibliotecas es organizar y ordenar las clases y otros elementos de una aplicación. El principio es el mismo que en la vida corriente: Si se dispone de un gran armario en casa, es obvio que la instalación de estanterías en este armario facilitará la organización y la búsqueda de los objetos guardados en relación con un almacenamiento en desorden. Por ejemplo, en Java las clases se agrupan en lo que se conoce como paquetes. En .NET, el equivalente serían los espacios de nombres y en Python u otros lenguajes, los módulos.

La organización de las clases en paquetes conlleva las ventajas siguientes:

- Facilidad para encontrar y utilizar de nuevo un elemento.
- Limitación de los riesgos de conflictos de nombres.
- Creación de una nueva visibilidad (la visibilidad package) además de las estándares (private, protected, public). Cuando un elemento (una clase, una variable o un método) no tiene modificadores de visibilidad, entonces este elemento solo es visible por los presentes en el mismo paquete.

Lo primero que hay que hacer cuando se quiere crear un paquete es darle un nombre. Hay que elegirlo con cuidado para permitir que el paquete cumpla totalmente con su papel. Debe ser único y representativo de los elementos almacenados en su interior.

Para asegurar la unicidad del nombre de un paquete, por convención se utiliza el nombre de dominio de la empresa invirtiendo el orden de los elementos como primera parte para el nombre del paquete. Por ejemplo, si el nombre del dominio es cifpaviles.pa, la primera parte del nombre del paquete será pa.cifpaviles. Si el nombre del dominio contiene caracteres prohibidos para los nombres de paquetes, se sustituyen por el carácter _ (guion bajo). De este modo, para el nombre de dominio cifp-aviles.pa, la primera parte del nombre del paquete será pa.cifp-aviles. Esta primera parte del nombre permite garantizar la unicidad de los elementos respecto al exterior de la empresa. La parte siguiente del nombre del paquete asegurará la unicidad de los elementos en el interior de la aplicación. Hay que elegir nombres claros y representativos de los elementos del paquete. Por ejemplo, en una aplicación se pueden tener dos clases Cliente, una que represente a una persona que hizo un pedido a la empresa y otra que represente a un cliente en el sentido informático del término (cliente - servidor). Se podrían emplear los siguientes nombres de paquete para albergar estas dos clases. pa.cifp-aviles.conta para la clase que representa al cliente físico y pa.cifp-aviles.conta.red para la clase que representa un cliente de software. Por convención, los nombres de los paquetes se escriben en minúsculas. Además, la declaración del paquete se hace obligatoriamente en la primera línea del archivo.

Todos los elementos alojados en este archivo de código fuente formarán parte de este paquete. Si no se indica ninguna información relativa al paquete en el archivo de código fuente, entonces se considerará que los elementos definidos en este archivo forman parte del paquete por defecto, que no tiene ningún nombre. Conviene ser prudente con este paquete por defecto, ya que las clases definidas en él no son accesibles desde un paquete con nombre. Entre otros, este es el motivo por el que se recomienda ubicar siempre una clase en un paquete con nombre.

El uso de los paquetes impone también una organización específica a la hora de grabar en disco los archivos de código fuente y los archivos compilados. Los directorios donde se graban estos archivos deben respetar los nombres utilizados para los paquetes. Cada componente del nombre de paquete debe corresponder a un subdirectorio. Los archivos de código fuente y los compilados pueden almacenarse en distintas ramas de árbol.

1.5.4.1. Clases abstractas

A veces se desea declarar una clase y no definir todavía todos los métodos que pertenecen a ella. Por ejemplo, supóngase una clase llamada **Mamifero** y que incluya en ella un método llamado **MarcaTerritorio**. Sin embargo, a priori no se sabe cómo definir MarcaTerritorio ya que es diferente para cada tipo de mamífero. Por supuesto, se puede plantear manejar esto con subclases derivadas de mamífero, como Perro o Humano ¿pero que código hay que poner en la función MarcaTerritorio de la clase Mamifero?

En Java, se puede declarar la función MarcaTerritorio de Mamifero como un método abstracto. Hacerlo así permite declarar el método sin escribir el código para él en esa clase. Sin embargo, es posible escribir código para el método en la subclase. Si un método es declarado como abstracto, entonces la clase ha de ser declarada también como abstracta. Para Mamifero y sus subclases, esto se traduciría en código de la siguiente forma:

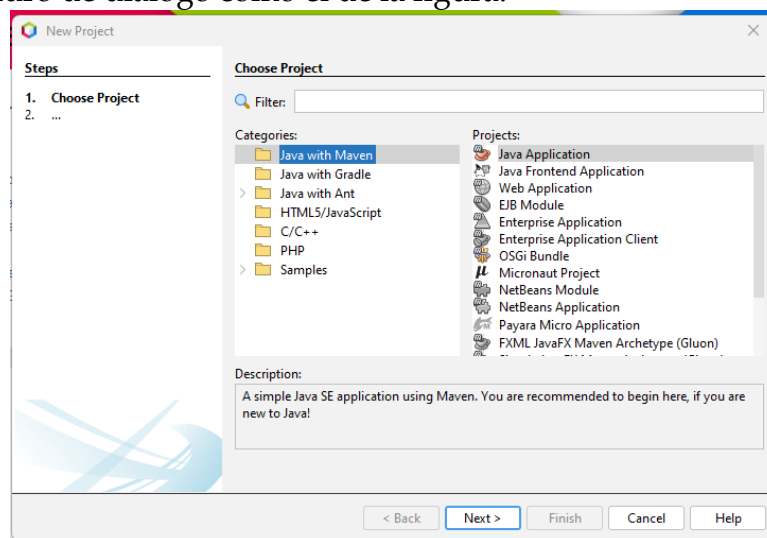
```
abstract class Mamifero {  
    abstract void MarcaTerritorio();  
}  
  
public class Humano extends Mamifero {  
    public void MarcaTerritorio() {  
    }  
}  
  
public class MiembroTribuUrbana extends Mamifero {  
    public void MarcaTerritorio() {  
        // Marca territorio con grafiti  
    }  
}  
  
public class Perro extends Mamifero {  
    public void MarcaTerritorio() {  
        // Marca territorio pues... como saben marcar los perros. :)  
    }  
}
```

Con las declaraciones precedentes, la clase Mamifero no contiene código para MarcaTerritorio. La clase **Humano** podría marcar su territorio construyendo una cerca alrededor de él, mientras que la clase **MiembroTribuUrbana** podría contener código para marcar territorio con un spray de pintar grafitis. La clase perro marcaría territorio levantando la pata y haciendo lo que hacen los perros para marcar territorio.

Nota: Un método que es privado no puede ser también declarado como abstracto. Puesto que un método privado no puede ser sobrecargado en una subclase, un método privado abstracto podría no ser utilizable.

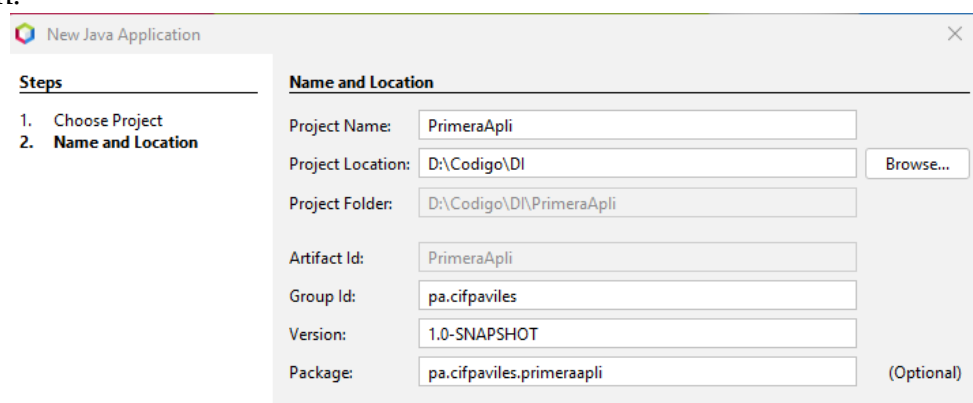
1.6. COMPONENTES: CARACTERÍSTICAS Y CAMPO DE APLICACIÓN

Para afrontar este apartado y los siguientes, se va a trabajar con el Framework Swing de Java utilizando como entorno de desarrollo NetBeans. Para los ejemplos de este texto se utilizará Apache NetBeans 19 aunque se pueden realizar perfectamente sobre versiones anteriores. Por otra parte, estas versiones nuevas de NetBeans están en inglés, por lo que las opciones elegidas estarán en ese idioma. Es fácil deducir cuáles serían en castellano en caso de estar usando versiones anteriores que soporten el español (hasta la 8.2). En primer lugar, para crear una aplicación Java que soporte Swing, hay que crear un nuevo proyecto en File >> New Project. Se mostrará un cuadro de diálogo como el de la figura:



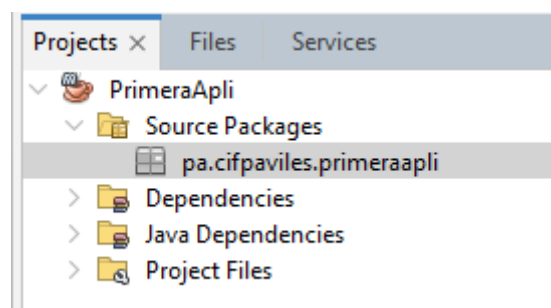
20. Nuevo proyecto Java

Se pueden ver diversas opciones de crear una aplicación Java, que es lo que interesa para este ejemplo. Tanto Maven como Gradle y Ant, son herramientas de construcción automática de software y su misión es automatizar el proceso de despliegue llevando a cabo las operaciones necesarias para obtener un empaquetado, es decir, en este caso, un ejecutable Java. Como puede verse en la imagen, NetBeans recomienda Maven para principiantes y resulta la opción más apropiada, básicamente porque Ant es más antigua y porque Gradle está más orientada a web y desarrollo móvil además de soportar lenguajes como Groovy; Maven está más centrada en Java. A partir de esta elección, se lanza un asistente en el que se pueden ajustar los datos de la aplicación.



21. Primera aplicación Java

En la imagen pueden verse los datos principales a modificar: Nombre del proyecto, ubicación, identificador de grupo (la nomenclatura de la organización para paquetes y clases) y el nombre del paquete, el cual por defecto sigue las normas de nombrado que se vieron en la sección correspondiente. Una vez terminado, se muestra la estructura de proyecto de la nueva aplicación junto con una clase principal, la cual de primeras se va a eliminar para poder partir de un proyecto completamente limpio. El borrado es muy simple, basta con acceder a la clase (PrimeraAppli.java en el ejemplo) y con el menú contextual seleccionar **Delete**. En el cuadro de diálogo que aparece es muy recomendable hacer un borrado seguro marcando “Safety delete (with usage search)” y pulsando **Refactor**. Este tipo de borrado no solo elimina el fichero sino todas las referencias a la clase que pueda haber en otras partes del proyecto, minimizando la aparición de errores posteriores de compilación. Una vez borrada, el árbol de proyecto debería quedar más o menos así:



22. Árbol de proyectos de primera aplicación.

1.6.1. CONTENEDORES. DIÁLOGOS MODALES Y NO MODALES

Los contenedores son componentes que permiten almacenar, alojar o contener otros elementos gráficos. Java Swing provee algunos contenedores útiles para diferentes casos; así cuando se diseña una ventana, se puede decidir de qué manera se van a presentar sus elementos, cómo serán alojados y de qué forma se presentan al usuario.

El primer contenedor que se va a crear es el principal que requiere cualquier aplicación de escritorio y que equivaldría a una ventana principal. Se conoce como **JFrame** y es el elemento que se añadirá a continuación al proyecto que ya se ha comenzado como ejemplo. En el apartado correspondiente a las herramientas de edición, ya se vio cómo crear un nuevo **JFrame**. Otra forma de hacerlo consiste en abrir el menú contextual sobre el paquete en el árbol de proyectos y elegir **New > JFrame Form**. Como nombre, se le da **FormPrincipal**.

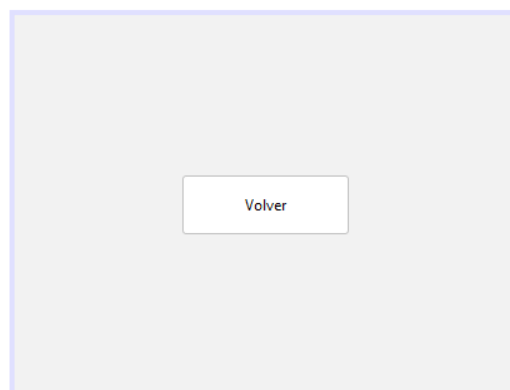
Dentro de ese **JFrame**, se va a colocar un botón de comando en el centro aproximadamente. Normalmente, a la derecha está la paleta de componentes en la cual se elegirá un **Button** dentro de la categoría **Swing Controls**. Se arrastra y se suelta al centro del formulario y se le hace un poco más grande para que resulte más vistoso. Cuando se tiene seleccionado un control, pueden verse sus propiedades en la paleta de componentes. La primera propiedad que debe cambiarse en cualquier control es el nombre de la variable, ya que luego será la forma de poder referenciarlo en el código. Por ejemplo, dado que es un botón que se pretende que abra un diálogo, se le puede nombrar como **btnDialogo**. El valor se cambia en **Code / Variable Name**.

Otra propiedad que es interesante modificar es el propio texto del botón (label), que en este caso será “Ir al diálogo”. Al final, el formulario debería quedar con una forma similar a la de la imagen:



23. JFrame con botón.

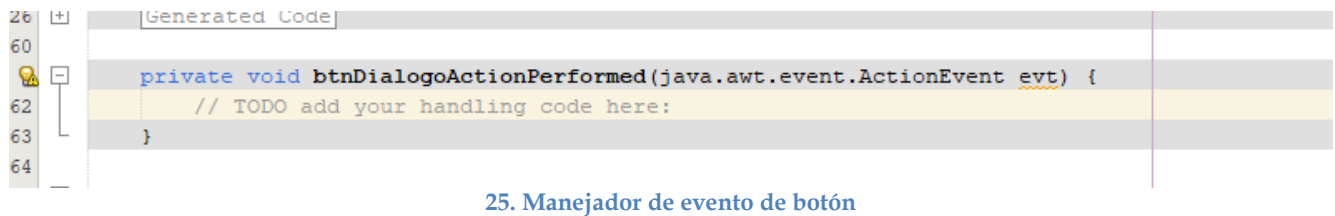
La idea en este caso es que con el botón se acceda a un nuevo componente contenedor, en este caso, JDialog. Este tipo de contenedores permite prácticamente la inclusión de los mismos componentes que un JFrame, aunque está más orientado a ser un cuadro de diálogo de interacción con el usuario, por ejemplo, en un asistente. El proceso de añadido de un JDialog es muy similar al de cualquier fichero que se añade al proyecto: Menú contextual sobre el paquete y se elige **New > Other > Swing GUI Forms > JDialog Form**. En este caso, se le puede dar como nombre **FormSecundario**. Dentro de este diálogo, ahora se repite el proceso introduciendo un nuevo botón que tenga como texto “Volver” y que se tenga de nombre de variable **btnVolver**.



24. Control JDialog

Ahora, lo que queda es programar la interacción entre ambos contenedores, es decir, que cuando se pulse el botón “Ir al diálogo” se abra el control correspondiente y al pulsar “Volver” se cierre. Para ello, se debe programar la secuencia de operaciones que la aplicación debe realizar al pulsar sobre cada botón, es decir, lo que sería un manejador de evento “clic” sobre un botón. Aunque ya se profundizará posteriormente sobre el manejo de eventos en Java y otras tecnologías, NetBeans facilita mucho esta tarea, ya que basta con hacer un doble clic sobre el botón en la vista de diseño para que muestre una vista de código la cual tiene el cursor justo donde se programa el manejo del evento.

Por ejemplo, si se hace doble clic sobre el botón del FormPrincipal, el cursor se pararía aproximadamente en la zona de la figura:



25. Manejador de evento de botón

Dentro de este método manejador, se va a introducir el siguiente código:

```
FormSecundario frmSecundario = new FormSecundario(this, true);  
frmSecundario.setVisible(true);
```

Lo que hace esta secuencia de comandos es:

1. Crear un nuevo objeto de la clase FormSecundario (JDialog). Es importante darse cuenta de que, **tanto el JFrame como el JDialog que se acaban de añadir al proyecto son clases, no objetos**. Es decir, en realidad son nuevas clases que van a contener una serie de controles u otros contenedores que tienen que ser materializadas como objetos cuando se necesiten. Esto, además, permite que se puedan crear tantos objetos idénticos como se desee de un formulario o diálogo concreto. Además, se le dan dos parámetros al constructor de clase: el elemento padre que en este caso es el formulario principal y que se representa por `this` ya que el código se lanza desde él mismo y si el diálogo es o no modal (`true` o `false`). Si es modal, cuando se abra, no se va a poder acceder dentro de esa aplicación más que a ese diálogo hasta que se cierre. Si no es modal, se puede dejar abierto y volver a otras ventanas.
2. Mostrar el diálogo recién creado mediante el método **setVisible** con un booleano que indica si se quiere mostrar u ocultar.

De forma similar, se va a programar el botón Volver dentro del diálogo. Al ser modal el diálogo, simplemente basta con ocultarlo para que se pueda volver al formulario principal. Por tanto, el código a introducir es muy sencillo:

```
this.setVisible(false);
```

Para probarlo, se necesita abrir el menú contextual en el formulario principal, ya que es el de “arranque” y seleccionar **Run**. ¡Ojo! Si se lanza la ejecución desde el botón correspondiente al proyecto, el sistema mostrará un error, ya que no se ha definido qué clase es la de punto de entrada. Para ello, lo ideal es ajustar dicha clase en el proyecto: **Menú contextual > Properties > Run > MainClass > Browse** y se selecciona el archivo FormPrincipal.java

En la ejecución se puede comprobar cómo se abre el diálogo y se cierra con la acción correspondiente.

Además de estos dos contenedores, Java Swing ofrece otros los cuales también tienen interés en el diseño de interfaces de usuario.

JPanel

Este contenedor es uno de los más simples. Permite la creación de paneles independientes donde se almacenan otros componentes, para de esta manera decidir qué elementos se alojan en cada panel y, dado el caso poder usar sus propiedades para ocultar, mover o delimitar secciones.

JScrollPane

Este contenedor permite vincular barras de scroll o desplazamiento en una aplicación. Puede ser utilizado tanto en paneles como en otros componentes como un JTextArea.

JSplitPane

Este componente permite la creación de un contenedor dividido en 2 secciones. Se usa en aplicaciones donde una sección presenta una lista de propiedades y otra sección presenta el elemento al que se le aplicamos dicha lista. Cada sección puede ser manipulada aparte pudiendo incluso redimensionar sus componentes.

JTabbedPane

Permite la creación de una pestañas en una ventana. Cada pestaña representa un contenedor independiente donde se pueden alojar paneles u otros elementos.

JDesktopPane

Este contenedor aloja componentes de tipo JInternalFrame, los cuales representan ventanas internas, permitiendo así crear ventanas dentro de una principal. En el momento de su creación se pueden manipular sus propiedades para definir si se quieren redimensionar, cerrar, ocultar etc. También es posible definir una posición inicial de cada ventana interna, sin embargo, después de presentadas, se pueden mover por toda la ventana principal donde se encuentran alojadas.

JToolBar

Este contenedor representa una barra de herramientas en la que se pueden alojar diferentes componentes que pudieran ser útiles. Esta barra de herramientas puede ser manipulada permitiendo cambiar su ubicación con tan solo arrastrarla al extremo que se desee o sacarla para que las opciones se encuentren como una ventana independiente.

1.6.2. GESTIÓN DE COMPONENTES DE LA INTERFAZ

Una vez que se han visto los componentes que son capaces de contener a otros (incluso contenedores), a continuación, se verán algunos de los componentes más comunes.

TextField, Button y Label

Para ilustrar la utilización de estos controles básicos, se va a crear un nuevo proyecto con el nombre DatosAlumnos. Se siguen los pasos a continuación:

1. Una vez creado el proyecto, eliminar la clase principal.
2. Añadir un JFrame. Su nombre será FrmInicio y como título se llamará “Datos de alumnos”.

Una vez creado el formulario, ahora se necesitará insertar varios controles para tener un aspecto similar al de la figura:



26. Ventana de datos de alumnos

Las propiedades de los controles se detallan en la siguiente tabla:

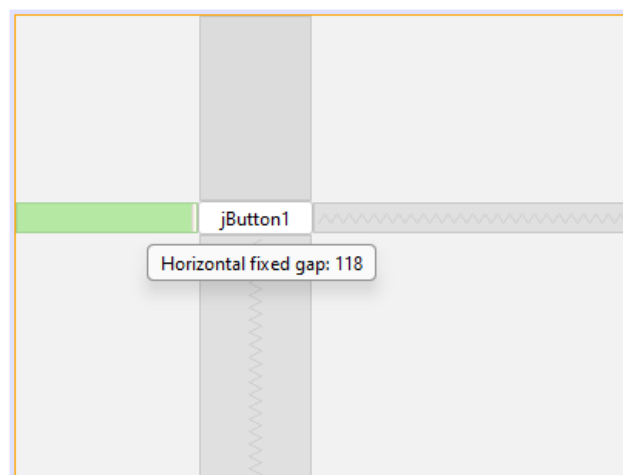
Tipo de control	Variable name	Text
JLabel	lblDescNALum	Nº alumnos
JLabel	lblNumAlum	lblNALum
JLabel	lblDatosAlum	Datos de alumnos
JLabel	lblNombre	Nombre
JLabel	lblApellidos	Apellidos
JLabel	lblCurso	Curso
JButton	btnAceptar	Aceptar
JButton	btnVerDatos	Ver datos
JTextField	txtNombre	
JTextField	txtApellidos	
JTextField	txtCurso	

El nombre de las etiquetas que sirven de leyenda a los distintos campos no es relevante, por lo que no es imprescindible cambiarlos, pero sí la que contiene el texto lblNAIum, ya que por código se va a modificar su texto.

En este formulario se pueden ver varios detalles. Por un lado, una tendencia en la nomenclatura a identificar cada control anteponiendo su tipo (lbl, btn, txt). Obviamente, no es obligatorio, pero sí muy útil para la codificación posterior. Por otra parte, una de las etiquetas tiene un tipo de letra más grande y en negrita. En el panel de propiedades del control hay un elemento llamado **font** que permite su cambio abriendo un cuadro de diálogo estándar de cambio de fuente.

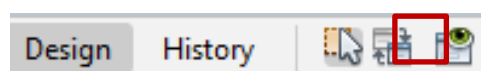
1.6.3. UBICACIÓN Y ALINEAMIENTO DE COMPONENTES

Uno de los aspectos más importantes en el diseño de cualquier interfaz de usuario, sea web, escritorio o para dispositivos móviles es la maquetación. En el diseño del anterior formulario pudo verse cómo NetBeans introduce una serie de líneas guía para maquetar los controles y alinearlos. Además de las líneas, también se pueden ver lo que se conoce como **gaps** (huecos en español), los cuales pueden ser fijos (**fixed**) o redimensionables (**resizable**). Para ilustrar esto, se puede crear un nuevo formulario e insertar un botón en cualquier parte. Por defecto, tanto el gap izquierdo como el superior (las distancias hacia los bordes correspondientes) son fijos mientras que el derecho y el inferior son redimensionables.



27. Gaps en un JFrame

Ello quiere decir que, si el formulario se amplía por la derecha por abajo, crece sin moverse el botón ni modificar su tamaño, manteniéndose anclado tanto por la parte superior como por la izquierda. Esto se puede ver sin necesidad de ejecutar el programa con la vista previa del diseño (**Preview Design**) que puede encontrarse en un botón a la derecha de la vista de diseño y el historial.



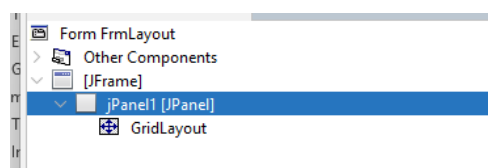
28. Preview Design

Es interesante hacer pruebas manteniendo fijos los distintos gaps. Por ejemplo, es posible fijar también el derecho con solo seleccionarlo y en el menú contextual seleccionar **Edit Layout Space**. Con solo desmarcar la casilla **Resizable** ya se fija también el control a la parte derecha, pero aún así, para que esto suceda, hay que seleccionar el botón y elegir **Auto Resizing > Horizontal** para que redimensione el control al cambiar el ancho del formulario. Con la misma operativa, se puede anclar a la parte superior y añadir el autodimensionado vertical para que el botón cambie su tamaño acorde con el del formulario.

Aunque ha servido para un primer diseño, tanto NetBeans como otros Frameworks de desarrollo de interfaz de usuario proporcionan mecanismos para maquetar y alinear controles. En Swing, uno de esos elementos es el conocido como **layout**, que traducido al español sería disposición. Precisamente eso, la disposición de los controles en un contenedor es lo que se gestiona mediante un layout en Swing.

Un control contenedor puede estar asociado a un layout por defecto, lo que, como ya se ha comentado, define la disposición de los controles en su interior. Si se selecciona el JFrame creado anteriormente y se abre su menú contextual, una de las opciones es **Set Layout**. En el formulario del ejemplo, el layout por defecto es “Free design”, es decir, diseño libre, lo que deriva en que no hay una disposición cerrada, sino que los controles se pueden colocar en cualquier parte del formulario.

Si se introduce otro elemento contenedor, por ejemplo, un JPanel, este también tiene su layout totalmente independiente del componente que lo contenga. Volviendo al ejemplo anterior, ahora se va a eliminar el botón y colocar un JPanel con un borde (esto ayudará a identificarlo). A este panel se le puede dar una disposición de tipo GridLayout mediante el menú contextual y **Set Layout > GridLayout**. Una vez hecho esto, parece que el panel desaparece, pero en realidad se ha ajustado a un tamaño mínimo. Se puede redimensionar sin problema, pero hay que tener cuidado con elegir el control y no su layout en el Navigator.



29. Panel y Layout en Navigator.

La variedad de layouts que ofrece Java Swing es muy diversa ofreciendo distintas posibilidades que cubren casi todas las necesidades de maquetación de controles. En la documentación oficial de Java puede consultarse una guía visual de los distintos layouts que es realmente útil para hacerse una idea de cada uno de ellos:

<https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

Volviendo al GridLayout que se ha ajustado en el panel de ejemplo, sus propiedades se pueden ajustar accediendo a él vía Navigator. Un GridLayout no deja de ser una disposición en cuadrícula, así que al seleccionarlo, podrán verse sus propiedades, que consisten en filas,

columnas y gaps horizontal y vertical. Se pueden ajustar 2 filas y 2 columnas y un espacio entre filas y entre columnas de 10 cada uno. Para probarlo, nada mejor que ubicar en su interior cuatro botones y observar cómo se van disponiendo según el layout elegido. El resultado sería similar al de la figura:



30. Panel con GridLayout y cuatro botones

Esta disposición, por ejemplo, podría ser interesante para interfaces que requieren botones grandes como los TPV.

1.6.4. PROPIEDADES DE LOS COMPONENTES

Anteriormente se presentó tanto el editor de propiedades como algunas de ellas en controles como cuadros de texto, etiquetas y botones. Resulta interesante ver cómo materializa Java Swing mediante NetBeans estos cambios de propiedades.

En el formulario anterior de ejemplo el cual estaba preparado para registrar datos de alumnos, se puede ver un ejemplo de lo citado en el anterior párrafo. Si se va a vista de código, entre otros elementos hay un método **private void initComponents()** en el cual puede verse cómo se inician todos los controles del formulario. Aquí resulta interesante pararse en el código de la imagen:

```
lblDatosAlum.setFont(new java.awt.Font("Segoe UI", 1, 18)); // NOI18N  
lblDatosAlum.setText("Datos de alumnos");
```

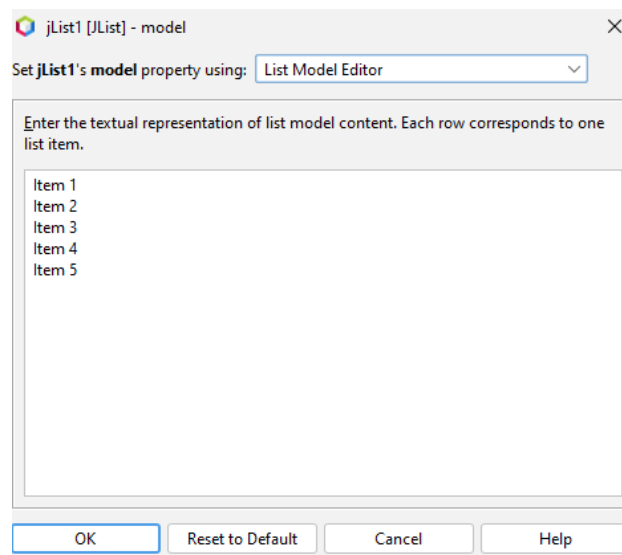
31. Ajuste de propiedad en etiqueta

Como puede verse en este código generado automáticamente por NetBeans, se ajustan tanto el tipo de letra como el texto de la etiqueta según los parámetros registrados en el entorno de desarrollo. Hay que destacar que las propiedades que aparecen en un control en cuanto se insertan mediante NetBeans son las que tiene por defecto y que solo se hará algún cambio en código si se producen modificaciones en ellas, como es el caso. También hay que recordar que esos cambios “visuales” tienen una traducción a código y que, si no se contara con el apoyo del entorno de desarrollo, sería preciso codificar al completo el diseño.

Para ilustrar el manejo de estas propiedades, se va a continuar introduciendo algunos de los elementos más comunes de uso en un interfaz de usuario.

Control JList

Un control JList contiene una lista de elementos, de los cuales el usuario puede hacer una selección de uno o varios. La propiedad **model** recoge el modelo subyacente a la propia lista el cual representa los elementos que aparecerán en ella. Se puede acceder en el cuadro de propiedades y asignarle elementos en tiempo de diseño pulsando en el botón con tres puntos que aparece a la derecha de la propiedad. A partir de ese momento se abre el editor de modelos el cual permite varias posibilidades de llenado de lista: desde el editor de modelo de lista, desde un valor de un componente existente o con código personalizado.



32. Editor de modelo de lista

Menús

El control **Menu Bar (JMenuBar)** proporciona una estructura de opciones de menús para un formulario. Para ilustrar la creación de un menú, se va a crear un nuevo JFrame llamado **FrmMenu**.

En el diseñador de formularios, se añade un control **Menu Bar** al formulario principal seleccionándolo en la paleta de controles. Está localizado dentro del grupo **Swing Menus**. Una vez insertado, puede verse en la parte superior izquierda conteniendo además unas opciones por defecto (File, Edit). Aunque se va a crear un menú nuevo, es posible aprovechar estas entradas modificando simplemente sus valores. De momento, a esta barra de menú se le va a dar el nombre de **mnuPrincipal**. Al pulsar sobre esta barra de menú, el entorno deja modificar los elementos contenidos en ella, File y Edit, que son del tipo Menu (JMenu). Una vez más, es muy útil acudir al Navigator donde se va viendo cómo se configura el árbol de controles una vez se modifica el menú.

A la primera opción que ahora se llama File, se le dará como texto “Opciones de Lista” y como name **mnuOpcionesLista**. Sobre este menú, se van a configurar las distintas entradas que en este caso se sustentan en controles de tipo **Menu Item (JMenuItem)**. La idea es tener la siguiente estructura:

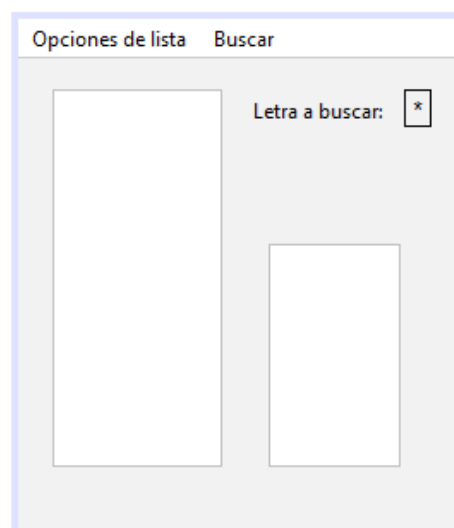
Añadir
Eliminar
----- (separador)
Rellenar

Hay que tener especial cuidado al arrastrar los Menu Item ya que deben soltarse justo encima del JMenuItem correspondiente (en este caso, Opciones de Lista). También se pueden añadir desde el Navigator con la opción **Add from Palette > Menu Item** desde el contextual del JMenuItem.

La entrada de menú Edit se va a etiquetar como Buscar y contendrá la siguiente estructura:
Buscar elemento (F8)
Buscar siguiente (F9)

En este caso, tanto F8 como F9 son atajos de teclado. Es fácil configurarlos, basta con hacer doble clic en el apartado **shortcut** que aparece justo a la derecha de la entrada. También aquí hay varias opciones y en este caso se irá a la más sencilla: **Key stroke editor**. Dentro de Virtual Key se puede elegir desde una gran variedad de teclas que representan posibles atajos además de combinarlas con Ctrl, Alt, Shift, etc. En realidad, estas teclas virtuales no dejan de ser constantes las cuales parametrizan estos posibles atajos. Para elegir tanto F8 como F9, se seleccionan VK_F8 y VK_F9 respectivamente.

Para probar las opciones de menú y algunas de las características de las listas, se va a configurar el formulario como se puede ver en la imagen:



33. Uso de menú y listas

La idea de esta aplicación de ejemplo es disponer de una lista con letras (lslLetras) en la parte izquierda las cuales se pueden añadir o eliminar y que se pueda buscar sobre ella mostrando en la lista de la derecha (lslPosiciones) la posición de la letra buscada, la cual aparecerá en una etiqueta (lblLetra) a continuación de la leyenda “Letra a buscar”.

En primer lugar, interesa que las opciones de “Buscar elemento” y “Buscar siguiente” no estén habilitadas si la lista de letras está vacía, ya que no tiene sentido buscar en ella. Para ello, se va a programar un manejador de evento de acceso a la entrada de menú “Buscar”. Esto se realiza acudiendo a la pestaña **Events** del JMenu correspondiente y en ella puede verse toda la lista de eventos soportados por este control. El que interesa para esta situación es **menuSelected**, ya que, en la misma selección de la entrada de menú, se controlará si la lista está o no vacía. Al pulsar en el desplegable, se propone un nombre de manejador de evento, el cual en este ejemplo podría ser **mnuBuscarMenuSelected**. Al pulsar sobre él, se crea el manejador y automáticamente se pasa a la vista de código para su diseño.

El código que debe quedar en esa porción es el siguiente:

```
private void mnuBuscarMenuSelected(javax.swing.event.MenuEvent evt)
{
    if (lslLetras.getModel().getSize() == 0){
        mnuBuscarElem.setEnabled(false);
        mnuBuscarSig.setEnabled(false);
    }
    else
        mnuBuscarElem.setEnabled(true);
}
```

El botón de “Buscar siguiente” se activará únicamente si se ha buscado previamente, de ahí que en caso de haber elementos en la lista no se active por defecto. En realidad, solo hay que codificar la parte de dentro de las llaves ya que la otra la proporciona automáticamente el IDE. Para saber si la lista está vacía, hay que acudir a su modelo subyacente, el cual es del tipo ListModel<T> donde T es el tipo de datos de los elementos contenidos en la lista (en este caso, String). Si su tamaño distinto de cero es que en la lista hay algún elemento, por lo que se condicionará el habilitado del menú a esa condición.

En la carga del Frame también hay que deshabilitar ambos botones, aunque incondicionalmente ya que se da por hecho que no habrá ningún elemento en la lista. El código siguiente, por tanto, se podría introducir en su manejador de evento WindowOpened:

```
mnuBuscarElem.setEnabled(false);
mnuBuscarSig.setEnabled(false);
```

El siguiente paso es codificar la opción de menú de “Buscar elemento”, por lo que hay que programar su evento, pero ¿cuál de ellos? NetBeans es capaz de ayudar en esta decisión; con solo hacer un doble clic sobre la opción de menú, el IDE pasa a vista de código y se ubica en el manejador de evento creado, que en este caso es un ActionPerformed.

Antes de programar la lógica de dicho evento, hay que declarar una serie de variables que se utilizarán en este formulario. Son las correspondientes a la letra a buscar, su posición y la primera posición encontrada. El código siguiente debe ir en la clase ya que son variables propias de ella.

```
private char letraBuscar;  
private short posicion;  
private short primeraPosicion;
```

Además, hay que añadir una serie de imports para elementos que son necesarios en la codificación posterior.

```
import javax.swing.DefaultListModel;  
import javax.swing.JOptionPane;  
import javax.swing.text.Position;
```

El código del manejador de evento quedaría así:

```
private void mnuBuscarElemActionPerformed(java.awt.event.ActionEvent evt)  
{  
    // Se pide la letra a buscar  
    letraBuscar = JOptionPane.showInputDialog("Letra a buscar: ", "Buscar en lista");  
    // Solo se realiza la búsqueda si es un carácter y además una letra  
    // Para eso se usa una expresión regular  
    if (letraBuscar.length() == 1 && letraBuscar.matches("[A-Z a-z]"))  
    {  
        // Se ajusta la etiqueta con la letra introducida por el usuario en  
mayúsculas  
        this.lblLetra.setText(letraBuscar.toUpperCase());  
        DefaultListModel lstPosicionesModel = new DefaultListModel();  
        // Limpiado de lista  
        this.lstPosiciones.setModel(lstPosicionesModel);  
        // Se busca la letra en la lista avanzando desde arriba  
        posicion = this.lstLetras.getNextMatch(letraBuscar, 0,  
Position.Bias.Forward);  
        // Si se ha devuelto una posición válida, la posición se añade a la lista  
correspondiente  
        // Además, se selecciona la letra en la lista original y se habilita "Buscar  
siguiente"  
        if (posicion != -1){  
            lstPosicionesModel.addElement(posicion);  
            this.lstLetras.setSelectedIndex(posicion);  
            this.mnuBuscarSig.setEnabled(true);  
            this.lstLetras.ensureIndexIsVisible(posicion);  
            this.primerPosicion = posicion;  
        }  
        else  
        {  
            JOptionPane.showMessageDialog(this, "La letra no está en la lista",  
"Búsqueda en lista", JOptionPane.OK_OPTION);  
        }  
    }  
}
```


Este código tiene un buen puñado de líneas de interés. En primer lugar, la letra se pide mediante un cuadro de mensaje de entrada que se encuentra en la clase **JOptionPane**, la cual se usará posteriormente para mostrar otro cuadro de mensaje informativo.

Antes de añadir la letra, hay que evaluar si lo que se ha introducido por teclado tiene longitud uno y si tiene el formato de una letra de la A hasta la Z independientemente de si es minúscula o mayúscula. Para eso, se usa el método **matches**, el cual permite evaluar si una cadena cumple con una expresión regular. Las [Regular Expressions](#) (RegEx) constituyen en sí mismas un lenguaje declarativo orientado a la búsqueda de patrones de texto. En este caso, el patrón [A-Za-z] quiere evaluar si solo se ha introducido una letra del alfabeto.

Para realizar el limpiado de la lista de posiciones, lo que se hace es crear un nuevo **DefaultListModel**, que es la clase que utilizan las listas como modelo subyacente para su contenido. Se inicializa y se asocia a la lista con el método **setModel**. Después, se busca la letra en la lista hacia delante y se devuelve el índice en el que está comenzando desde la posición 0 mediante el método **getNextMatch**. Se evalúa si la posición es distinta de -1, lo que indica que se ha encontrado en la lista y, de ser así, se añade a la lista de posiciones y se selecciona la letra encontrada en la lista correspondiente. Hay que reseñar que añadir el elemento se hace al modelo subyacente (mediante **addElement**) y no a la lista en sí.

A continuación, se repite el proceso, pero codificando ahora “Rellenar”. La misión de esta opción de menú es la de llenar la lista con 100 letras al azar de forma automática para poder tener buen material de uso de los comandos de búsqueda.

```
private void mnuRellenarActionPerformed(java.awt.event.ActionEvent evt)
{
    // Se establece un nuevo modelo para la lista de letras
    DefaultListModel modelLetras = new DefaultListModel();
    this.lstLetras.setModel(modelLetras);
    // Límites inferior y superior de los códigos ASCII de las letras A a la Z
    short min = 65;
    short max = 90;
    int aleatorio;
    // Se generan 100 números aleatorios para convertirlos en letras.
    // Se usa el método Math.random(). Cada letra se añade al modelo subyacente de la
    lista.
    for (int i=0; i<100; i++)
    {
        aleatorio = (int)Math.floor(Math.random() * (max - min + 1) + min);
        modelLetras.addElement((char)aleatorio);
    }
    // Se limpia la etiqueta de la letra a buscar, se vacía lista de posiciones y se
    habilita buscar
    this.lblLetra.setText("");
    this.lstPosiciones.setModel(new DefaultListModel());
    this.mnuBuscarElem.setEnabled(true);
}
```


Este último código introduce como novedad la generación de números aleatorios y contiene una curiosidad: el número aleatorio generado entre 65 y 90 se convierte a su correspondiente letra del código ASCII con solo hacer una conversión a char de dicho número. De esta forma, se añade el elemento al modelo de la lista durante 100 iteraciones consiguiendo un conjunto de 100 letras al azar.

Ahora toca programar el botón de buscar siguiente, el cual se habrá activado con la primera búsqueda.

```
private void mnuBuscarSigActionPerformed(java.awt.event.ActionEvent evt)
{
    // Se empieza a buscar a partir de la posición anterior + 1
    posicion = this.lstLetras.getNextMatch(letraBuscar, posicion+1,
Position.Bias.Forward);
    // Se obtiene el modelo subyacente para trabajar con él
    DefaultListModel modelPosiciones =
(DefaultListModel)this.lstPosiciones.getModel();
    // Se guardó antes la primera posición para comprobar si se ha llegado a ella.
    // Ello se debe a que el método de búsqueda es cíclico y vuelve a empezar.
    // Si se termina la lista, se muestra el total
    if (posicion == primeraPosicion)
    {
        modelPosiciones.addElement("Total: " + modelPosiciones.getSize());
        this.mnuBuscarSig.setEnabled(false);
    }
    else if (posicion != -1)
    {
        modelPosiciones.addElement(posicion);
        this.lstLetras.ensureIndexIsVisible(posicion);
        this.lstLetras.setSelectedIndex(posicion);
    }
}
```

En los comentarios se explican algunas de las partes de este código.

1.6.5. ENLACE DE COMPONENTES A ORÍGENES DE DATOS

El acceso a datos es una de las principales tareas involucradas en el desarrollo de aplicaciones informáticas, lo cual hay que tener en cuenta debido a su especial relevancia a la hora de diseñar e implementar interfaces con acceso a datos de forma que su visualización y manipulación sea lo más práctica y funcional posible.

Con Java es posible manipular datos mediante la interfaz JDBC. Primeramente, se va a ver cómo usarla con un SGBD como [PostgreSQL](#). Se trata de un Sistema de Gestión de Base de Datos Relacional de código abierto que tiene un uso muy sencillo, además de ser muy potente y de utilizar un dialecto SQL muy similar al estándar.

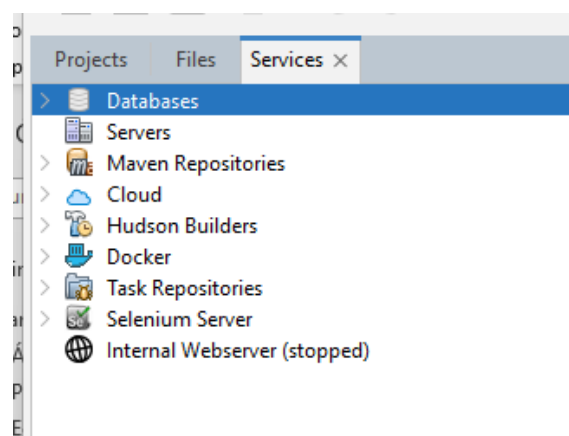
En este caso se realizará una conexión con un servidor local de PostgreSQL, ya que la conexión a uno remoto requiere una configuración posterior. Además, hay que tener en cuenta que una conexión a un servidor SQL remoto debe hacerse dentro de una misma red, nunca exponer a aquel a Internet abriendo su puerto correspondiente (en Postgres, 5432), ya que es una práctica insegura. Por tanto, todo ejemplo de acceso a datos mediante aplicaciones de escritorio que se trate en este módulo se tiene que ver como una arquitectura cliente-servidor, pero en una red interna, es decir, tanto sistema de base de datos como aplicaciones cliente estarán en una misma LAN no expuesta al exterior. Aunque es una arquitectura en desuso debido a que actualmente se utilizan más las aplicaciones web, todavía muchas empresas tienen aplicaciones internas que hacen uso de ella. Esto que no quiere decir que no haya aplicaciones de escritorio conectadas a bases de datos; se hace, pero a través principalmente de alguna capa intermedia o middleware que ofrece servicios de acceso a datos sin llegar directamente a la base de datos subyacente. Son arquitecturas orientadas a servicios siendo las más extendidas actualmente las que utilizan microservicios.

Volviendo al SGBD, para poder realizar los siguientes supuestos, es preciso instalar PostgreSQL en la misma máquina sobre la que se trabaja con NetBeans. Se puede descargar de la siguiente URL: <https://www.postgresql.org/download/>

En el proceso de instalación, hay un apartado en el que se solicita una contraseña; se trata de la del usuario postgres, el cual ejerce como administrador del sistema. Es importante recordarla para poder utilizarla posteriormente. También reseñar que lo que se instala es el servidor y que su manejo se realiza principalmente en línea de comando. Aunque hay utilidades como PGAdmin que permiten la administración de las bases de datos de este servidor, en este caso se verá cómo acceder a esa base de datos desde NetBeans.

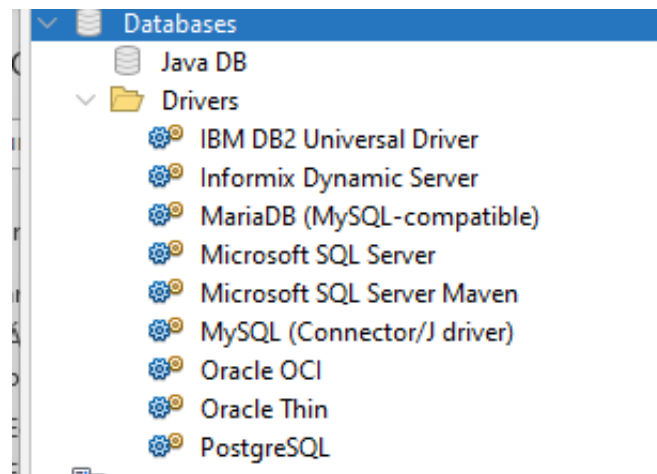
1.6.5.1. Servidor de bases de datos en NetBeans

Dentro del IDE NetBeans existe un gestor de servicios externos que resulta muy útil para trabajar, entre otras cosas, con un sistema gestor de bases de datos. Todas estas utilidades se encuentran en la pestaña Servicios (Services) que aparece en el panel que habitualmente se ubica a la izquierda del entorno junto con el de Proyectos y Ficheros.



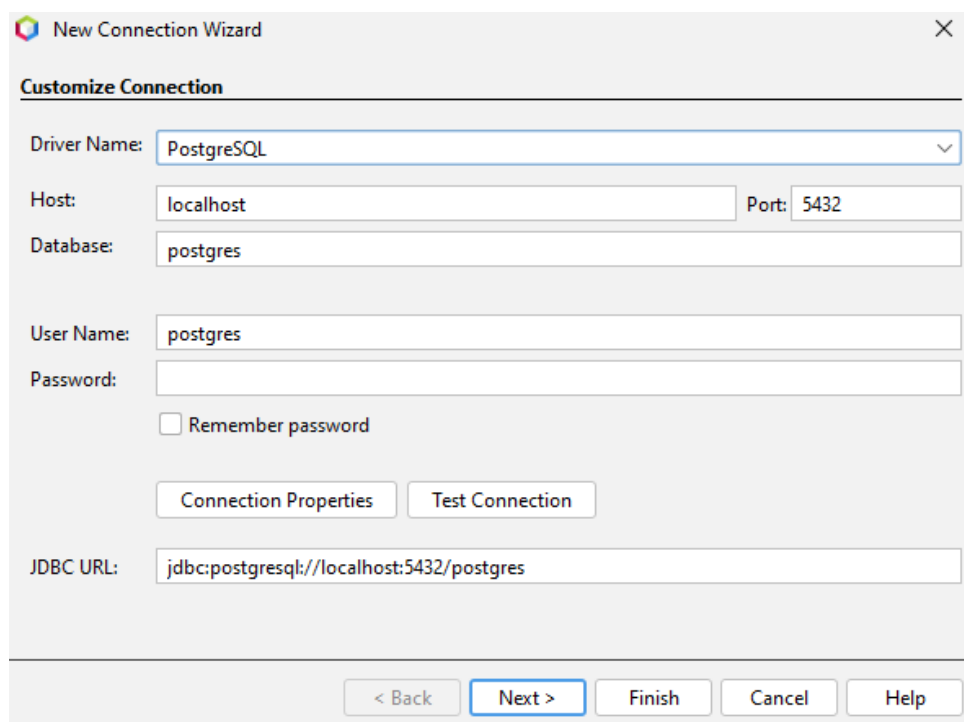
34. Pestaña de Servicios

Como puede verse por los grupos contenidos en ella, la variedad resulta muy interesante en un primer vistazo: bases de datos, servidores, repositorios Maven, nube, Hudson (sistema de integración continua), Docker, repositorios de tareas, servidor Selenium (entorno de pruebas para aplicaciones web) y el servidor web interno. En este apartado, se va a tratar el de bases de datos contenido en Databases. Al desplegarla, se puede ver el contenido de la imagen:



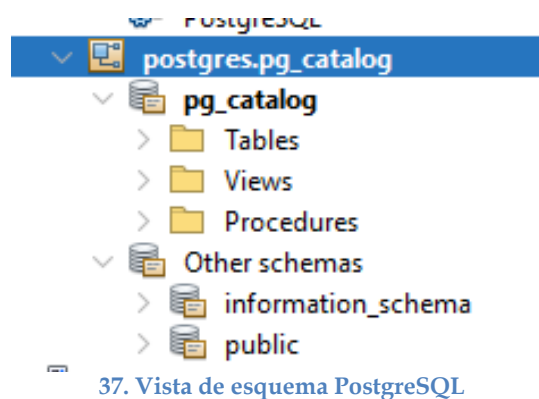
35. Drivers JDBC en NetBeans

La lista muestra los drivers de conexión a base de datos que están disponibles en el entorno. Para este ejemplo, se va a usar el de PostgreSQL. La conexión se realiza accediendo a él con el menú contextual y seleccionando **Connect Using** lo que provoca la aparición de un diálogo como el de la figura siguiente.



36. Conexión a PostgreSQL desde NetBeans

Este diálogo prácticamente da todo el trabajo hecho si la conexión se va a realizar a un servidor local, que es el caso de este ejemplo. Por tanto, basta simplemente con introducir la contraseña del usuario postgres (la que se configuró en la instalación) y pulsar “Test Connection” para comprobar que todo es correcto. Si es así, aparecerá un mensaje “Connection Succeeded” y se podrá pulsar Next sin mayor problema. Si la conexión no funciona, al ir al siguiente paso del asistente, mostrará el mismo error que en Test Connection y no dejará configurarla. A continuación, se indica el esquema con el que se va a trabajar, es decir, qué conjunto de tablas u otros objetos de bases de datos se va a visualizar (p.e. pg_catalog) y para finalizar, un nombre identificativo (p.e. postgres.esquema). Una vez configurado, NetBeans proporcionará una vista de árbol con el servidor y el esquema seleccionado (aunque se pueden ver otros esquemas en la parte inferior).



De esta forma, se pueden ver las tablas del modelo, vistas y procedimientos. Además, es una potente herramienta de administración la cual, siempre que se tengan permisos, proporciona la posibilidad de crear tablas, modificar registros, insertarlos y, en definitiva, ejecutar cualquier comando SQL.

1.6.5.2. El componente JTable

El componente JTable, tal como se describe en la documentación de Java Swing sirve para mostrar y editar tablas con celdas en dos dimensiones. En definitiva, es lo que se conoce como cuadrícula o rejilla y representa uno de los elementos más utilizados para visualizar datos.

De forma similar a como se vio con las listas, un JTable muestra la información de un objeto **TableModel** que representa al modelo subyacente de datos que componen dicha tabla. Si se consulta la propiedad con el entorno de desarrollo se puede ver cómo se configura este modelo. Su elemento principal es la columna, la cual llevará un identificador numérico, un título, el tipo de datos contenido en las celdas correspondientes a esa columna (tipos primitivos de Java: cadenas, booleanos, enteros, etc.) y si es o no editable. Pero en este caso, lo que interesa es crear un TableModel JDBC, es decir, asociar ese modelo a un objeto de una base de datos a través de JDBC. Lo que se explica a continuación se puede resumir en el uso de metadatos de tabla para construir un TableModel Swing a partir de ella. JDBC proporciona un medio abstracto para acceder a las bases de datos, es decir, no depende de la base de datos subyacente en el trabajo con Java ya que el interfaz que ofrece JDBC es común; la única diferencia es la forma en la que

JDBC consigue la conexión a la base de datos, lo que normalmente consiste en proporcionar variables para:

- Una clase controladora, la cual proporciona implementaciones de varios interfaces `java.sql`
- Una URL con la que conectarse a la base de datos. Esta se pudo ver en la conexión establecida con PostgreSQL dentro del apartado JDBC URL. La fontanería para conseguir dicha conexión conlleva el uso de sockets, que son elementos programáticos para comunicaciones a nivel bajo, aunque no necesariamente ya que podría darse el caso de que la base de datos estuviera integrada en la misma JVM que la aplicación que hace uso de ella.
- Un usuario (opcional).
- Una contraseña (opcional).

Con esto se conforma un objeto **Connection** con el que se puede comenzar a enviar comandos CRUD (Create, Read, Update, Delete) sobre las tablas o consultar la base de datos creando Statements (sentencias) contra la conexión. También se puede usar la conexión para obtener metadatos de la base de datos como qué clase de características soporta, la longitud de ciertas cadenas, etc. Lo más importante para el supuesto que se muestra aquí es conseguir qué tablas hay en la base de datos, qué columnas tienen y sus tipos de datos.

Así que, dada una conexión y el nombre de una tabla de una base de datos, se puede construir una representación en Java de sus contenidos con dos consultas. La primera obtiene los metadatos de las columnas de la tabla y construye arrays con sus nombres y tipos. Esto puede mapearse razonablemente bien a clases Java, al menos para cualquier tipo de datos que se desee admitir. La segunda consulta, obtiene todos los datos de la tabla. Para cada columna, extrae su valor y lo coloca en una matriz bidimensional, la cual representa los contenidos completos de la tabla.

Con el material de estas dos consultas, ya se tiene lo suficiente para poder utilizar los métodos abstractos de un [AbstractTableModel](#), que como su propio nombre indica, es la clase abstracta de la que debe heredar cualquier TableModel de un JTable:

- `getRowCount()` es la longitud de la matriz de contenidos que se ha creado.
- `getColumnCount()` es 0 si no hay contenido o la longitud del primer elemento la matriz de contenidos, siendo este en sí mismo un array ya que el de contenidos es bidimensional.
- `getValueAt()` es el valor en `contenidos[fila][columna]`

`AbstractTableModel` tiene implementaciones completamente triviales de `getColumnClass()` y `getColumnName()` de forma que la primera devuelve un `Object.class` y la segunda devuelve "A", "B", "C", etc. Conservar los metadatos de la columna con la primera consulta también permite implementaciones más útiles de estos métodos. Para llevar esto a la práctica, se va a crear una clase `JDBCTableModel` que herede de `AbstractTableModel` permitiendo ser poblada por información obtenida de una base de datos.

El código completo de esta clase es el siguiente:

```
import javax.swing.table.*;
import java.sql.*;
import java.util.*;
/* TableModel inmutable construido
de los metadatos de una tabla en una BD JDBC
*/
public class JDBCTableModel extends AbstractTableModel {
    Object[][] contents;
    String[] columnNames;
    Class[] columnClasses;
    // El constructor por defecto requiere conexión y nombre de tabla
    public JDBCTableModel (Connection conn,
                           String tableName)
        throws SQLException {
        super();
        getTableContents (conn, tableName);
    }
    // Método de obtención de contenidos de tabla
    protected void getTableContents (Connection conn,
                                      String tableName)
        throws SQLException {
        // getMetaData: qué columnas existen y
        // de qué tipo (clase) son
        DatabaseMetaData meta = conn.getMetaData();
        System.out.println ("Metadatos obtenidos = " + meta);
        ResultSet results =
            meta.getColumns (null, null, tableName, null) ;
        System.out.println ("Obtenidos resultados de columna");
        ArrayList colNamesList = new ArrayList();
        ArrayList colClassesList = new ArrayList();
        while (results.next()) {
            colNamesList.add (results.getString ("COLUMN_NAME"));
            System.out.println ("Nombre: " +
                               results.getString ("COLUMN_NAME"));
            int dbType = results.getInt ("DATA_TYPE");
            switch (dbType) {
                case Types.INTEGER:
                    colClassesList.add (Integer.class);
                    break;
                case Types.FLOAT:
                    colClassesList.add (Float.class);
                    break;
                case Types.DOUBLE:
                case Types.REAL:
                    colClassesList.add (Double.class); break;
                case Types.DATE:
                case Types.TIME:
                case Types.TIMESTAMP:
```

```
        colClassesList.add (java.sql.Date.class); break;
    default:
        colClassesList.add (String.class); break;
};
System.out.println ("Tipo: " +
    results.getInt ("DATA_TYPE"));
}
columnNames = new String [colNamesList.size()];
colNamesList.toArray (columnNames);
columnClasses = new Class [colClassesList.size()];
colClassesList.toArray (columnClasses);
// Obtiene todos los datos de la tabla
// y los coloca en un array de contenidos
Statement statement = conn.createStatement ();
results = statement.executeQuery ("SELECT * FROM " +tableName);
ArrayList rowList = new ArrayList();
while (results.next()) {
    ArrayList cellList = new ArrayList();
    for (int i = 0; i<columnClasses.length; i++) {
        Object cellValue = null;
        if (columnClasses[i] == String.class)
            cellValue = results.getString (columnNames[i]);
        else if (columnClasses[i] == Integer.class)
            cellValue = Integer.valueOf(results.getInt (columnNames[i]));
        else if (columnClasses[i] == Float.class)
            cellValue = Float.valueOf(results.getInt (columnNames[i]));
        else if (columnClasses[i] == Double.class)
            cellValue = Double.valueOf(results.getDouble (columnNames[i]));
        else if (columnClasses[i] == java.sql.Date.class)
            cellValue = results.getDate (columnNames[i]);
        else
            System.out.println ("No puedo asignar " + columnNames[i]);
        cellList.add (cellValue);
    }
    Object[] cells = cellList.toArray();
    rowList.add (cells);
}
// Finalmente se crea el array bidimensional de contenidos
contents = new Object[rowList.size()] [];
for (int i=0; i<contents.length; i++)
    contents[i] = (Object []) rowList.get (i);
System.out.println ("Modelo creado con " +
    contents.length + " filas");
results.close();
statement.close();
}
// Métodos de AbstractTableModel
public int getRowCount() {
    return contents.length;
}
```



```
}  
public int getColumnCount() {  
    if (contents.length == 0)  
        return 0;  
    else  
        return contents[0].length;  
}  
public Object getValueAt (int row, int column) {  
    return contents [row][column];  
}  
  
// Métodos de sobrecarga para los cuales AbstractTableModel  
// tiene implementaciones  
public Class getColumnClass (int col) {  
    return columnClasses [col];  
}  
public String getColumnName (int col) {  
    return columnNames [col];  
}  
}
```

El constructor delega su trabajo real en **getTableContents()**, que es responsable de las dos consultas que se acaban de describir. Obtiene un objeto **DatabaseMetaData** de la conexión, desde el cual se pueden obtener los datos de las columnas con una llamada a **getColumns()**. Los argumentos de este método son el catálogo, esquema, nombre de tabla y nombre de columna; esta implementación ignora catálogos y esquema, aunque podría ser necesario si la base de datos es más compleja, de ahí que se incluya. **getColumns()** devuelve un **ResultSet**, que es un objeto de tipo cursor con el que se puede iterar para recorrer los resultados de una consulta, siendo uno de los elementos principales de trabajo con JDBC.

Obtener el nombre de columna es muy fácil: simplemente llamar a **getString("COLUMN_NAME")**. El tipo es un poco más interesante, desde la llamada a **getInt("DATA_TYPE")** que devuelve un entero representando una de las constantes de la clase **java.sql.Types**. Lo que se hace aquí es mapear los tipos básicos numéricos y los de cadena a sus correspondientes clases Java. **TIMESTAMP** es un concepto SQL que equivale a un punto concreto de tiempo (una fecha y hora), de forma que lo que se obtiene será un tipo **Date** de Java. Conociendo estos tipos, será mucho más fácil la llamada al método **getXXX()** correcto cuando se estén obteniendo los datos actuales de una tabla.

La segunda consulta es un simple **SELECT * FROM tableName**. Sin restricción **WHERE**, creará un **ResultSet** con cada fila de la tabla. Eso sí, hay que ser consciente de que, si la tabla en cuestión tiene millones de registros, el **TableModel** probablemente no quepa en memoria, así que hay que ser cuidadoso con su uso o buscar otras opciones en ese caso.

Una vez más, se itera sobre un **ResultSet**. Cada vez que **results.next()** devuelve **true**, lo que significa que hay otro resultado, se extraen todas las columnas conocidas de la consulta de

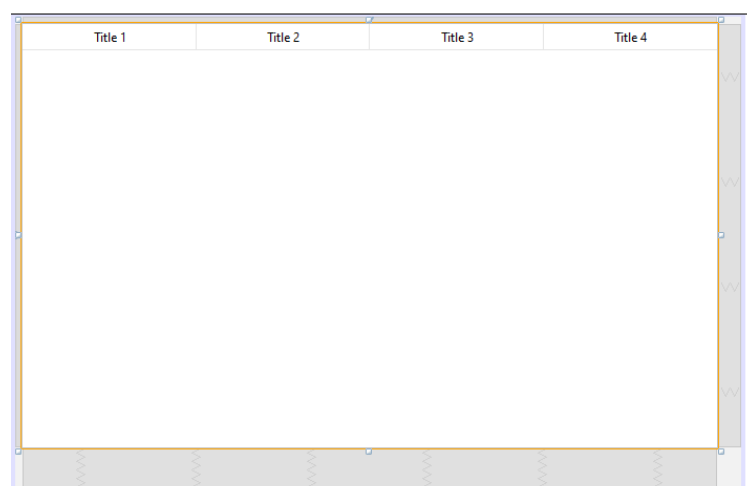
metadatos ya citada. Esto implica llamar a un método `getXXX()` y pasarle el nombre de columna, dando por hecho que se sabe qué corresponderá a esas tres X según la investigación anterior de cada columna. Ahora se puede continuar introduciendo los datos numéricos en su correspondiente clase envoltorio (Integer, Double, etc.) ya que esto trabaja mucho mejor con el sistema de renderizado basado en clases de los JTables. Esto se podría afinar usando un **TableCellRenderer** que aplique un formato a todos los datos de tipo Double de la tabla para que tengan un número concreto de decimales o mostrar las fechas con términos relativos como “Hoy” o “Hace 25 horas”.

Con las consultas realizadas, se convierten las **ArrayLists** a arrays reales ya que ofrecen búsquedas rápidas para los métodos `get`. Las implementaciones de los métodos del `AbstractTableModel` que se mencionaron antes, además de las implementaciones mejoradas de `getColumnClass()` y `getColumnName()` son usos triviales de `columnNames`, `columnClasses` y los arrays de contenido creados con este método.

Ahora toca probar todo esto. Para ello, se necesita un driver de conexión, nombre de usuario y contraseña. Por ejemplo, para conectarse con PostgreSQL los datos se podrían obtener de la conexión creada anteriormente en NetBeans consultando sus propiedades (con el menú contextual en la propia conexión). En este ejemplo, serían:

- Driver: `org.postgresql.Driver`
- URL: `jdbc:postgresql://localhost:5432/postgres`
- User: `postgres`
- Password: (la introducida en la instalación)

Para probar este modelo se va a crear un `JFrame` con una `JTable`. En la lógica de este componente, se va a crear programáticamente una nueva tabla en la base de datos. El modelo obtiene la conexión y el nombre de esta tabla y carga su contenido de la base de datos. Posteriormente, asocia el modelo de la `JTable` a esta tabla y muestra sus registros en la aplicación. El formulario tendrá un diseño muy sencillo ya que solo tiene como objetivo probar el enlace de control con base de datos.



38. JFrame con JTable

Dentro del JFrame se van a necesitar los siguientes import para el trabajo con la base de datos y otros elementos.

```
import java.io.File;
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import javax.swing.table.TableModel;
```

Para crear tanto tabla como registros en la base de datos, se usará el siguiente método al que se va a llamar posteriormente:

```
private static String createSampleTable (Connection conn)
throws SQLException {
    Statement statement = conn.createStatement();
    // Elimina la tabla si existe
    try {
        statement.execute ("DROP TABLE empleados");
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
    statement.execute ("CREATE TABLE empleados " +
        "(nombre CHAR(20), titulo CHAR(30), salario INT)");
    statement.execute ("INSERT INTO empleados VALUES " +
        "('Jill', 'CEO', 200000 )");
    statement.execute ("INSERT INTO empleados VALUES " +
        "('Bob', 'VP', 195000 )");
    statement.execute ("INSERT INTO empleados VALUES " +
        "('Omar', 'VP', 190000 )");
    statement.execute ("INSERT INTO empleados VALUES " +
        "('Amy', 'Ingeniero/a de Software', 50000 )");
    statement.execute ("INSERT INTO empleados VALUES " +
        "('Greg', 'Ingeniero/a de Software', 45000 )");
    statement.close();
    return "empleados";
}
```

A simple vista, se puede deducir que este método recoge una conexión a una base de datos y crea un objeto Statement con el que se van a lanzar sucesivas consultas al SGBD. La primera de ellas elimina la tabla empleados para crearla de nuevo posteriormente. Dado que en la eliminación se podría producir algún error, se envuelve dentro de un try..catch. Después se crea la tabla “empleados” y se insertan una serie de registros.

El siguiente método va a establecer ese modelo como TableModel en el JTable.

Su código es el siguiente:

```
private void setTableModel() {
    try {
        Properties testProps = new Properties();
        String ddriver = System.getProperty("jdbcTable.driver");
        String durl = System.getProperty("jdbcTable.url");
        String duser = System.getProperty("jdbcTable.user");
        String dpass = System.getProperty("jdbcTable.pass");

        if (ddriver != null) {
            testProps.setProperty("jdbcTable.driver", ddriver);
        }
        if (durl != null) {
            testProps.setProperty("jdbcTable.url", durl);
        }
        if (duser != null) {
            testProps.setProperty("jdbcTable.user", duser);
        }
        if (dpass != null) {
            testProps.setProperty("jdbcTable.pass", dpass);
        }
        try {
            testProps.load(new FileInputStream(
                new File("src/main/resources/jdbcTable.properties")));
        } catch (Exception e) {
        }
        System.out.println("Propiedades:");
        testProps.list(System.out);
        // Obtiene una conexión.
        // Además, reemplaza nulos con cadenas vacías para evitar errores.
        String url = testProps.getProperty("jdbcTable.url");
        url = ((url == null) ? "" : url);
        String user = testProps.getProperty("jdbcTable.user");
        user = ((user == null) ? "" : user);
        String pass = testProps.getProperty("jdbcTable.pass");
        pass = ((pass == null) ? "" : pass);
        try (Connection conn = DriverManager.getConnection(url, user, pass)) {
            // Crea la base de datos a usar
            String tableName = createSampleTable(conn);

            // Obtiene el modelo y se lo asocia al JTable
            TableModel mod = new JDBCTableModel(conn, tableName);
            tblJDBC.setModel(mod);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Antes de ver dónde ubicar la llamada a este método, hay que explicarlo, sobre todo teniendo en cuenta que se necesita hacer alguna operación más. En primer lugar, este método va a buscar los datos propios de la conexión en un fichero de propiedades. Este puede ser el propio de un proyecto (el cual normalmente se ubica en un `config.properties`) o uno creado para la ocasión. En este ejemplo, se crea un fichero en la ruta `src/main/resources/jdbctable.properties` (ojo, hay que crear el directorio `resources` además del fichero) y se ubican dentro las propiedades. Esto es muy útil ya que permite modificar estos parámetros sin tocar una línea de código. Además, se podría encriptar la contraseña ya que se trata de un archivo que al final va a poder verse en el despliegue de la aplicación. En este caso, no se hará para facilitar el trabajo.

En este caso, con una conexión a PostgreSQL, el archivo va a lucir de forma similar a la siguiente:

```
jdbctable.driver=org.postgresql.Driver  
jdbctable.url=jdbc:postgresql://localhost:5432/postgres  
jdbctable.user=postgres  
jdbctable.pass=(ContraseñaPostgres)
```

Es muy interesante ver que el método **createSampleTable** recibe una conexión, pero que no tiene ni idea de su tipo; es decir, puede ser una conexión a PostgreSQL, a SQL Server, a MySQL, a HSQL, incluso se puede establecer una conexión JDBC a un fichero o mediante un puente ODBC a un archivo de Excel. Estas conexiones a bases de datos se verán de forma superficial en este módulo ya que pertenecen al ámbito de “Acceso a Datos”. Una vez que se tienen los datos de la conexión, esta se crea con la siguiente línea:

```
Connection conn = DriverManager.getConnection(url, user, pass)
```

Es muy habitual envolver estas sentencias “delicadas” en bloques `try..catch` ya que son susceptibles de fallar (no funciona la conexión, está mal configurada, un ratón soltó el cable de red, un ratón ha soltado el cable de red, etc.). Una vez establecida la conexión, se llama al método creado anteriormente para crear la tabla de muestra (`createSampleTable`) y se inicializa el `TableModel` con la clase que se diseñó al principio de este apartado para, a continuación, asociarlo al control `JTable` (que en este caso se ha llamado `tblJDBC`):

```
TableModel mod = new JDBCTableModel(conn, tableName);  
tblJDBC.setModel(mod);
```

El constructor de la clase `JDBCTableModel` recoge la conexión y el nombre de la tabla. No necesita más para devolver un `TableModel` que le sirve de fuente de datos a la `JTable`. Gracias a esta implementación, se facilita mucho la tarea de enlazado de datos en un control de esta naturaleza. Por supuesto, es un método más y depende del estudiante en este caso (o del personal de desarrollo en el ámbito profesional) su utilización. Lo interesante, sobre todo, es ver aspectos importantes para poder trabajar con bases de datos enlazadas a componentes de un interfaz de usuario.

Finalmente, la llamada al método **setTableModel()** se puede introducir en el constructor del `Frame` justo después de **initComponents()**.

1.7. CONTROLES DE USUARIO. BIBLIOTECAS DE CONTROLES

Al mostrar datos en formularios en aplicaciones de escritorio, se pueden elegir controles existentes en el Cuadro de herramientas o bien, crear controles personalizados si la aplicación requiere funcionalidad que no está disponible en los controles estándar. Este sería el objetivo principal de los controles de usuario. En .NET, tal como se verá en unidades posteriores, existe una clase `UserControl` de la que se puede heredar para crear nuevos componentes o controles de usuario. En Java, existe un concepto del que poder apoyarse para crear este tipo de controles: componentes `JavaBean`.

Normalmente, un componente no deja de ser un conjunto de líneas de código que se encierran en una clase. La clave en este tipo de controles es que el IDE sea capaz de mostrar sus propiedades en el panel correspondiente y en eso consistirá precisamente el apoyarse en `JavaBeans`. Los componentes `Bean` no dejan de ser igualmente una clase; no es necesario escribir código adicional ni usar extensiones del lenguaje especiales para poder transformar algo en este tipo de componentes. De hecho, lo único que hay que hacer es modificar ligeramente la forma de llamar a los métodos, ya que esto es lo que le dirá al IDE si se trata de una propiedad, un evento o un método convencional.

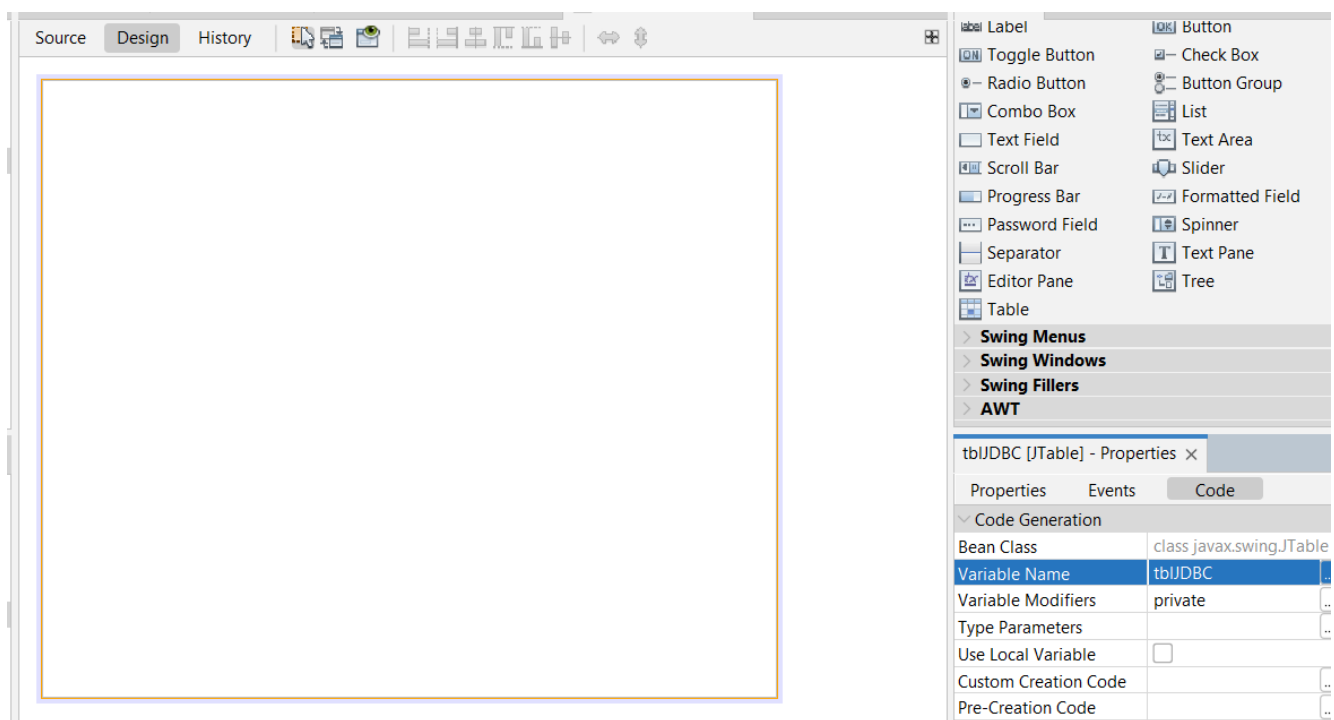
Para que los elementos de una clase sean reconocibles en un `Bean` se requiere que se sigan las normas de nombrado a continuación:

1. Para una propiedad llamada **xxx**, normalmente se crearán dos métodos **getXxx()** y **setXxx()**. La primera letra después de **get** o **set** se pondrá automáticamente en minúscula por las herramientas que examinen los métodos con el fin de generar el nombre de la propiedad. El tipo producido por el método **get** coincide con el del argumento del método **set**.
2. Para una propiedad de tipo booleano, se puede emplear la técnica anterior basada en **get** y **set**, pero se puede usar **is** en lugar de **get**.
3. Los métodos normales del componente `Bean` no se adaptan al convenio de denominación anterior, pero deben ser de tipo **public**.
4. Para los eventos, se usa la solución `Swing` basada en **listeners**, que se verán en el siguiente apartado. Ejemplos de estas denominaciones serían **addBounceListener(BounceListener)** y **removeBounceListener(BounceListener)** para gestionar un evento **BounceEvent**. En cualquier caso, la mayor parte de las veces, tanto los eventos como los listeners ya predefinidos servirán para satisfacer las necesidades del control.

Un ejemplo interesante se puede enlazar con el apartado anterior. Supóngase que se quiere encapsular como control esa `JDBCTable` que es capaz de enlazar con una tabla de un sistema gestor de base de datos. En primer lugar, habría que buscar un control que sirviera de contenedor a ese `JTable` ya que como tal no lo es. Uno de los contenedores vistos anteriormente que puede servir a este propósito es **JPanel**.

Por tanto, lo primero que habría que hacer en el proyecto es añadir un nuevo JPanel, lo que se hace de forma similar al agregado de un JFrame: menú contextual de proyecto, New / JPanel Form. Como el objetivo es crear un control de usuario que represente a una JTable con elementos JDBC, se le puede dar como nombre JDBCTable.

Ahora, en su interior, se puede colorar un JTable que se necesita que ocupe prácticamente todo el espacio del panel. Para que ocupe todo el panel y se pueda redimensionar acorde con el cambio en dimensiones del contenedor, se modificará el Layout del JPanel por un BorderLayout. También se puede borrar el contenido del TableModel subyacente. Como nombre de la tabla, se le puede asignar tblJDBC.



39. JTable incrustada en un JPanel

Ahora se puede aprovechar la anterior codificación con ligeras modificaciones. En primer lugar, añadir los import necesarios en la clase JPanel.

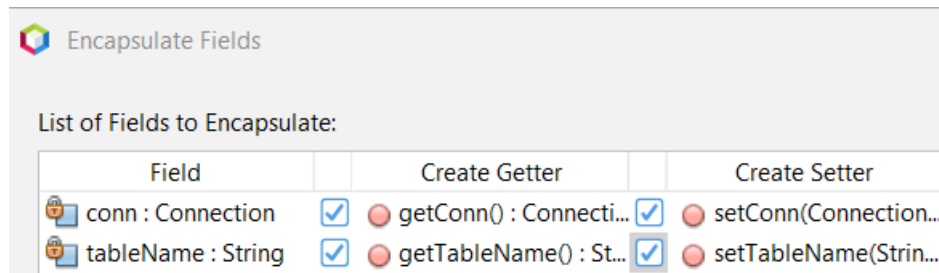
```
import java.sql.Connection;  
import java.sql.SQLException;  
import javax.swing.table.TableModel;
```

El control va a tener dos propiedades nuevas: la conexión y el nombre de la tabla. Por tanto, hay que crear ambas variables y colocarlas en la clase JDBCTable:

```
private Connection dbConnection;  
private String tableName;
```

Tal como se indica en la documentación, para crear un JavaBean, hay que envolver las variables que se necesiten como propiedad en getters y setters con el formato que se vio anteriormente. NetBeans (y prácticamente cualquier IDE) proporciona utilidades que ayudan a estos

menesteres. Es lo que se conoce como refactorización de código. Para este caso, lo interesante es conseguir ambos getters y setters de forma automática. El proceso es sencillo: ir a la variable que se necesite refactorizar, mostrar su menú contextual (botón derecho del ratón) e ir a **Refactor > Encapsulate fields**. Como puede verse en la imagen, el sistema da la posibilidad de encapsular toda variable privada que encuentra, no solo la seleccionada. Simplemente con marcar las deseadas y hacer clic en **Refactor**, repentinamente aparecerán nuevos métodos que cumplen con el cometido deseado.



40. Encapsulación de variables.

Además, por defecto el IDE proporciona los nombres de los métodos tal como se pide que sean para poder crear el Bean, así que, en este caso se podría decir que “miel sobre habichuelas”.

Antes de ver cómo quedan estos métodos, se va a presentar el método `setTableModel()` el cual ajustará el modelo al control `JTable` pero siempre que tanto conexión como nombre de tabla tengan un valor determinado.

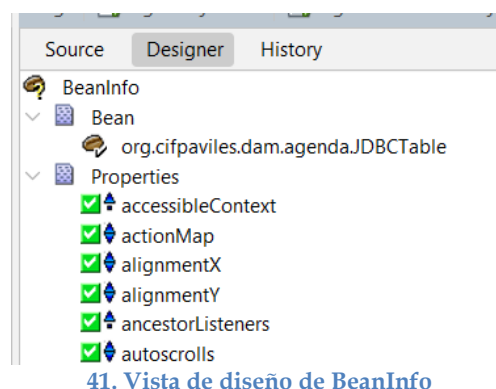
```
private void setTableModel() throws SQLException {  
    if (dbConnection != null && !"".equals(tableName))  
    {  
        TableModel mod = new JDBCTableModel(dbConnection, tableName);  
        tblJDBC.setModel(mod);  
    }  
}
```

Como puede verse, si no están los dos valores asignados, el método no hace nada ya que le falta uno de los elementos clave para establecer el modelo. Por otra parte, se pueden ver detalles importantes que, precisamente, ha propuesto el IDE a la hora de codificar. Primeramente, se incluye un `throws SQLException` ya que la implementación interna del `TableModel` necesita una conexión SQL y el sistema necesita poder gestionar dicha excepción. De esta forma, el método podría lanzar una excepción de ese tipo y es por lo que aparece en la propia definición de la función. Otro detalle curioso es cómo evalúa que la variable `tableName` no tiene valor. Para comparar cadenas, es recomendable usar el método `equals` que contiene la clase `String`, pero, además en este caso invierte los términos y pregunta si la cadena vacía es igual que el valor de la variable; esto último se hace para mejorar el rendimiento de la aplicación, aunque tenga una sintaxis un poco extraña.

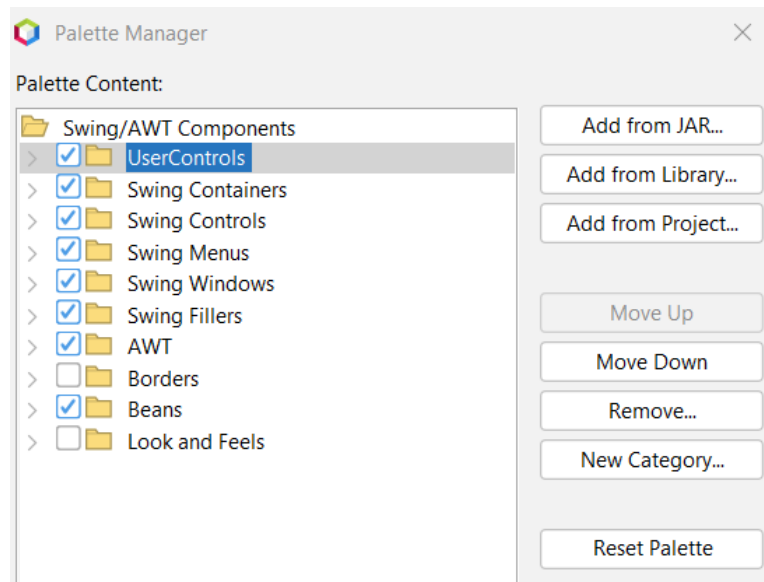
A este método se le llamará en cuanto se modifique alguna de las dos propiedades: Conn y TableName. Por tanto, los métodos quedarán finalmente así.

```
public Connection getDbConnection() {  
    return dbConnection;  
}  
public void setDbConnection(Connection dbConnection) throws SQLException {  
    this.dbConnection = dbConnection;  
    setTableModel();  
}  
public String getTableName() {  
    return tableName;  
}  
public void setTableName(String tableName) throws SQLException {  
    this.tableName = tableName;  
    setTableModel();  
}
```

Lo más interesante ahora es poder usar este componente incluyéndolo en la paleta de controles. De primeras puede verse que cumple con los requisitos para ser convertido en Bean ya que tiene sus propiedades expuestas con getters / setters y un constructor por defecto. El disponer de un componente insertable en NetBeans pasa irremisiblemente por la existencia de una clase de tipo **BeanInfo** por cada componente a insertar. Este tipo de clase actúa como descriptor de las distintas propiedades de un componente cara a su inclusión en la paleta de componentes además de como un envoltorio de la clase sobre la que se basan. Para crearla, sencillamente hay que dirigirse a la propia clase de la cual se quiere crear esta BeanInfo y en el menú contextual seleccionar **BeanInfo Editor**. El sistema indicará que no existe e instará a su creación, operación que es preciso aceptar. A partir de ahí, se añadirá una nueva clase llamada **JDBCTableBeanInfo.java**. No se va a profundizar en este tipo de clases, aunque se enumerarán una serie de detalles con respecto a ellas. En primer lugar, hay que indicar que heredan de **java.beans.SimpleBeanInfo**, la cual según la [documentación oficial](#) consiste en una clase de soporte al desarrollador para generar clases **BeanInfo**. Por otra parte, NetBeans proporciona la posibilidad de visualizarlas en modo diseño, tal como puede verse en la figura:



Para poder hacer uso del control, simplemente hay que agregarlo a la paleta de componentes, lo que se hace mediante el menú **Tools > Palette > Swing/AWT Components** desde el cual sería aconsejable añadir una nueva categoría. En este caso se puede añadir una llamada `UserControls` y el diálogo quedará como en la imagen:



42. Agregado de componentes a paleta de controles

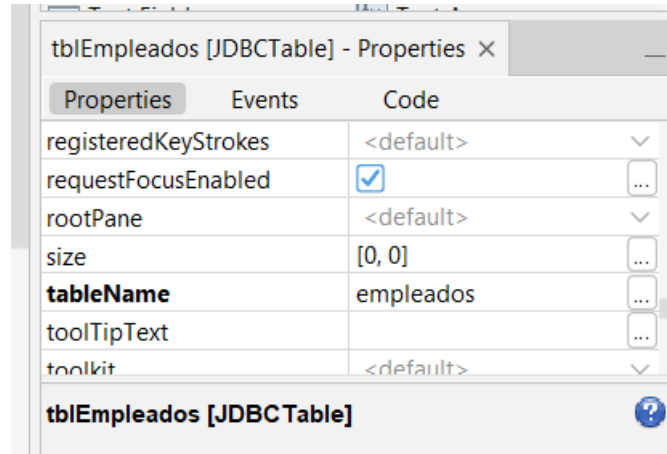
En este diálogo se propone el agregado desde tres fuentes:

- **Add from JAR.** Desde aquí se pueden añadir componentes que estén en un fichero JAR concreto dentro del sistema de archivos donde se está ejecutando NetBeans. Se usaría si se dispone de dicho JAR, aunque se puede hacer de forma indirecta desde el apartado siguiente.
- **Add from Library.** En este caso, se agrega desde una biblioteca reconocida por NetBeans. También se puede crear una nueva biblioteca desde un JAR como el que se puede añadir en el punto primero.
- **Add from Project.** Esta sería la mejor opción si se está diseñando un sistema de información que implique a varios proyectos también en fase de desarrollo. Puesto que en este ejemplo no se ha creado un proyecto específico que recoja los controles de usuario, se va a elegir esta posibilidad.

Independientemente de la fuente, el IDE reconocerá las posibles clases y las mostrará en una lista. Para este ejemplo, aparecerán dos: la clase `JDBCTable` y `JDBCTableInfo`; hay que añadir la primera y, gracias al `BeanInfo`, el IDE reconocerá sus propiedades.

En cualquier caso, un diseño elegante tendría que ubicar los componentes en una biblioteca propia. La operativa para diseñar este tipo de proyecto consiste en crear uno de tipo proyecto `Java Class Library`. La diferencia básicamente con una aplicación es que no tendrá una clase principal, ya que no está encaminado a una ejecución.

Una vez que aparece en el control, ya se puede insertar en un JFrame u otro componente contenedor y se puede ver cómo aparecen las nuevas propiedades en el panel correspondiente.



43. Propiedades del control de usuario.

La conexión podría introducirse también desde este panel, pero es mejor asignarla en el constructor de la clase del JFrame:

```
public FrTestUserControl() throws SQLException {  
    initComponents();  
    conn  
= DriverManager.getConnection("jdbc:postgresql://localhost:5432/postgres","postgres"  
,"contraseña");  
    this.tblEmpleados.setDbConnection(conn);  
}
```

1.8. ASOCIACIÓN DE ACCIONES A EVENTOS

La programación orientada a eventos es un paradigma en el que tanto estructura como ejecución de los programas se determinan por los sucesos ocurridos en el sistema, ya sean lanzados por el usuario o por él mismo. Para entenderla, se puede confrontar a lo que no es: mientras en la secuencial o estructurada se define cuál va a ser el flujo de programa, en la orientada a eventos, será la persona que interactúa con la aplicación la lo dirija. Aunque en la programación secuencial puede haber intervención de un agente externo, estas intervenciones ocurrirán cuando así se haya codificado y no en cualquier momento como cuando sucede un evento.

Al crear un programa con este paradigma, se definen los eventos que lo van a manejar y las acciones que producirán cada uno de ellos, lo que se conoce como manejador de evento. Al comenzar la ejecución de un programa se llevan a cabo las inicializaciones requeridas para el funcionamiento de la aplicación y esta quedará bloqueada esperando algún evento. Cuando alguno de ellos tenga lugar, la aplicación pasará a ejecutar el código de su manejador. Por ejemplo, si se pulsa un botón de play en un reproductor, el manejador será el que lance la visualización del vídeo por pantalla.

1.8.1. PROGRAMACIÓN DE EVENTOS

Como se vio en anteriores ejemplos, la programación de un manejador de evento en NetBeans es muy sencilla. Si se hace doble clic sobre el elemento, el sistema crea el método para el manejador de la acción por defecto, que, por ejemplo, en un botón sería el clic. Igualmente, en la pestaña Events dentro del panel de propiedades, se dispone de otra serie de eventos que pueden gestionarse del control elegido. El sistema que elige NetBeans es el de escuchadores o listeners que se verá en detalle en el apartado siguiente.

1.8.2. ESCUCHADORES

Un escuchador de eventos (event listener) es un mecanismo que permite reaccionar de forma asíncrona a ciertas circunstancias que suceden en una clase concreta. Es un sistema de comunicación muy habitual en Java donde un ejemplo de uso simple es la detección de pulsación de un botón. Para utilizarlos hay que seguir tres pasos:

1. Implementar la interfaz del listener
2. Registrar el listener en el objeto que genera el evento indicándole el objeto que los recogerá
3. Implementar los métodos callback correspondientes.

En esta clase de programación, a cada tipo de evento le corresponde una interfaz que debe implementar un objeto si quiere ser candidato a gestionar dicho evento. Para evitar la

proliferación de interfaces (ya muy numerosas), los eventos se agrupan en categorías. El nombre de estas interfaces siempre respeta la convención siguiente:

La primera parte del nombre representa la categoría de eventos que los objetos que implementan esta interfaz pueden gestionar. El nombre siempre termina en **Listener**.

Por ejemplo, está la interfaz [MouseEventListener](#), que corresponde a los eventos activados por los movimientos del ratón, o la interfaz [ActionListener](#), que corresponde a un clic en un botón. En cada una de estas interfaces se encuentran las firmas de los diferentes métodos asociados a cada evento.

```
public interface MouseEventListener extends EventListener
{
    void mouseDragged(MouseEvent e);
    void mouseMoved(MouseEvent e);
}
```

Cada uno de estos métodos recibe como argumento un objeto que representa el propio evento. Este objeto se crea automáticamente en el momento de la activación del evento; a continuación, se pasa como argumento al método manejador en el receptor de eventos. En general, contiene información adicional relativa al evento y es específico para cada tipo.

Es preciso crear clases que implementen estas interfaces. Desde este punto de vista, existen una multitud de posibilidades:

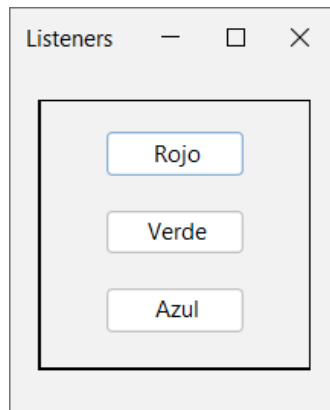
- ✓ Crear una clase «normal» que implemente la interfaz.
- ✓ Implementar la interfaz en una clase ya existente.
- ✓ Crear una clase interna que implemente la interfaz.
- ✓ Crear una clase interna anónima que implemente la interfaz.
- ✓ Crear eventualmente una expresión lambda si la interfaz es una interfaz funcional.

En algunos casos, quizá sea necesario no gestionar todos los eventos presentes en la interfaz. Sin embargo, es obligatorio escribir todos los métodos exigidos por la interfaz incluso si varios de ellos no contienen ningún código.

Para aclarar todo esto se va a ilustrar cada una de estas posibilidades con un pequeño ejemplo que va a permitir terminar correctamente la aplicación en el momento del cierre de la ventana principal al invocar al método **System.exit(0)**. Esta solución permite realizar verificaciones antes de detener la aplicación (copia de seguridad, mostrar un mensaje de confirmación, desconectar al usuario...). Es preciso, en este caso, modificar la propiedad **DefaultCloseOperation** de la ventana con el valor **DO_NOTHING_ON_CLOSE** para que no tenga una acción por defecto.

Se debe invocar este método durante la detección del cierre de la ventana. Para ello, se deben gestionar los eventos relacionados con la ventana y, en particular, el evento **windowClosing**, que se produce cuando el usuario cierra la ventana mediante el menú del sistema. La interfaz **WindowListener** está perfectamente adaptada para este tipo de trabajo.

Para ilustrar este ejemplo, primero se va a crear un JFrame que contenga un JPanel con borde y tres botones (recordar modificar la propiedad **DefaultCloseOperation**):



44. Ejemplo de uso de listeners

Dentro de esta clase, se incluirá un método `Mostrar()` que permita que la ventana sea visible:

```
public void Mostrar()
{
    this.setVisible(true);
}
```

Ahora se crea una clase `Main` que contenga un método homónimo para mostrar esta ventana.

```
import javax.swing.SwingUtilities;
public class Main {
    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(()->new FrmEventos().Mostrar());
    }
}
```

Hay que tener en cuenta que la interfaz gráfica está definida por una clase derivada de la clase `JFrame`. Por lo tanto, está completamente separada de la clase que contiene el método `main`. Si se ejecuta este código, la ventana aparece sin mayor problema, pero no se puede cerrar de ningún modo. Por tanto, habría que programar lo que va a suceder al cerrar la ventana. A continuación, se van a ver varias soluciones a la programación de este evento. Antes de nada, indicar que se utiliza un elemento nuevo consistente en un método `invokeLater` que aparece en la biblioteca `SwingUtilities`. No tiene la mayor relevancia, simplemente abre la aplicación, pero de una forma óptima en lo relacionado con concurrencia de procesos.

Usar una clase “normal” que implemente la interfaz `WindowListener`.

```
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class FrameListener implements WindowListener {
    @Override
    public void windowOpened(WindowEvent e) {
    }
    @Override
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
    @Override
    public void windowClosed(WindowEvent e) {
    }
    @Override
    public void windowIconified(WindowEvent e) {
    }
    @Override
    public void windowDeiconified(WindowEvent e) {
    }
    @Override
    public void windowActivated(WindowEvent e) {
    }
    @Override
    public void windowDeactivated(WindowEvent e) {
    }
}
```

Gracias a NetBeans, no es necesario copiar todo este código; se puede ir introduciendo gradualmente. Por ejemplo, se puede crear primero la clase vacía. Al introducir la cláusula “implements `WindowListener`”, NetBeans dará un aviso de que no encuentra la clase, pero propondrá, entre varias soluciones, añadir un import, que es lo que se va a hacer.

Pero, aun así, el sistema seguirá quejándose y tiene que ver con el hecho de que se le ha indicado que se va a implementar un interfaz, pero no se han introducido sus métodos. Una vez más, el propio IDE dará una posible solución como “Implement all abstract methods”, que es la que se solucionará. Cada uno de los métodos va a tener un throws relacionado con la no implementación; en este ejemplo se eliminarán y quedarán los métodos vacíos excepto el de `windowClosing`, que es el que interesa para este ejemplo.

Una vez creada la clase, el siguiente paso es asociarla con el `JFrame` correspondiente. Para realizar esta operación, se modificará el método `main` y además se utilizará un elemento nuevo: una llamada a un método anónimo por `SwingUtilities.invokeLater()`

Ahora el método main quedará como sigue:

```
public static void main(String[] args)
{
    SwingUtilities.invokeLater(()->{
        FrmEventos frm = new FrmEventos();
        // Se crea un objeto manejador de eventos
        FrameListener ev = new FrameListener();
        // Referencia a este objeto como receptor
        // de eventos para este Frame
        frm.addWindowListener(ev);
        frm.Mostrar();
    });
}
```

Implementar la interfaz en una clase ya existente

En esta solución, se va a delegar a la clase que representa la ventana la tarea de gestionar sus propios eventos al hacerle implementar la interfaz `WindowListener`. Basta con incluir el `extends` en el propio `JFrame` y repetir los pasos dados anteriormente con la clase externa.

La clase quedaría más o menos así_

```
public class FrmEventos extends javax.swing.JFrame implements WindowListener
```

Su constructor debería incluir una línea para asociar como `WindowListener` a la propia clase:

```
public FrmEventos() {
    initComponents();
    addWindowListener(this);
}
```

Y el método main volvería a tener la forma original:

```
SwingUtilities.invokeLater(()->new FrmEventos().Mostrar());
```

Con esta solución, el código se centraliza en una única clase. Si hay que gestionar varios eventos, esta clase va a contener una cantidad excesiva de métodos.

Crear una clase interna que implemente la interfaz

Esta solución es una mezcla de las dos anteriores, ya que se tiene una clase específica para la gestión de los eventos, pero definida en el interior de la clase que corresponde a la ventana. Por tanto, **hay que copiar el código de la clase `FrameListener` e introducirlo dentro del `FrmEventos` como clase anidada** y modificar el código del constructor del `JFrame` para que cree un objeto de ese tipo. Las clases anidadas o internas son un elemento permitido en Java y que puede ser útil en estos casos.

El código del constructor quedaría similar al siguiente:

```
public FrmEventos() {  
    initComponents();  
    // Creación de un objeto de la clase encargada  
    // de gestionar los eventos  
    FrameListener ev = new FrameListener();  
    // Referencia a este objeto  
    // como receptor de eventos para la ventana  
    addWindowListener(ev);  
}
```

Crear una clase interna anónima que implemente la interfaz

Esta solución es una ligera variante de la anterior, ya que se sigue teniendo una clase específica encargada de la gestión de los eventos, pero declarada ad-hoc para su uso. Se va a ubicar en el constructor, con lo que no se necesitará más código que el siguiente:

```
public FrmEventos() {  
    initComponents();  
    // Creación de un objeto de una clase anónima  
    // encargada de gestionar los eventos  
    addWindowListener(new WindowListener()  
    // Principio de la definición de la clase  
    {  
        public void windowActivated(WindowEvent arg0)  
        {  
        }  
        public void windowClosed(WindowEvent arg0)  
        {  
        }  
        public void windowClosing(WindowEvent arg0)  
        {  
            System.exit(0);  
        }  
        public void windowDeactivated(WindowEvent arg0)  
        {  
        }  
        public void windowDeiconified(WindowEvent arg0)  
        {  
        }  
        public void windowIconified(WindowEvent arg0)  
        {  
        }  
        public void windowOpened(WindowEvent arg0)  
        {  
        }  
    });  
}
```


El único inconveniente que presenta esta solución reside en la relativa complejidad de su sintaxis. También existe una problemática general que se le puede reprochar a todas estas soluciones: para un único método realmente útil, hay que escribir siete. Para evitar este código inútil, se puede trabajar con una clase que implemente ya la interfaz correcta y volver a definir únicamente los métodos que interesen.

Utilizar una clase XxxAdapter

```
public class FrameListener extends WindowAdapter {  
    public void windowClosing(WindowEvent arg0)  
    {  
        System.exit(0);  
    }  
}
```

En este caso se hereda de una clase que ya tiene una implementación de todos esos métodos, pero solo se sobrescribe el que se necesita. Ahora vuelve a servir el método utilizado anteriormente cuando se creó la clase en su forma original. De forma similar, se puede crear una clase interna o anidada y otra anónima para este WindowAdapter.

¿Pero cuál de todas estas formas es la que utiliza NetBeans? La respuesta se puede encontrar de forma sencilla; basta con acudir a la pestaña Events en el panel de propiedades y seleccionar el evento deseado (en este caso, windowClosing) para que el sistema proponga un manejador y se pueda ver su implementación.

En primer lugar, puede verse que crea un método manejador de este modo:

```
private void formWindowClosing(java.awt.event.WindowEvent evt)  
{  
    // TODO add your handling code here:  
}
```

Desplegando “Generated Code” es fácil encontrar la línea en la que se hace referencia a este método y, por ende, el sistema que usa NetBeans para asociar el evento:

```
addWindowListener(new java.awt.event.WindowAdapter() {  
    public void windowClosing(java.awt.event.WindowEvent evt) {  
        formWindowClosing(evt);  
    }  
});
```

La respuesta en este caso es sencilla: NetBeans usa un WindowAdapter creado como clase anónima con un método windowClosing que llama al manejador. Por tanto, se puede asumir que es la opción óptima en aquellos casos en los que haya más de un posible evento implicado.

ÍNDICE DE FIGURAS

1. MVC	4
2. MVP	5
3. MVVM.....	5
4. Proyecto de ejemplo NetBeans	8
5. Primer formulario Swing.....	8
6. Menú principal y barra de herramientas	9
7. Archivos de un proyecto	9
8. Área principal	10
9. Navegador de propiedades y métodos	10
10. Paleta de componentes	11
11. Editor de propiedades	11
12. Vista de código del JFrame	12
13. Interfaz de usuario basada en Swing.....	13
14. Ejemplo de interfaz en SWT.....	14
15. OpenGL.....	16
16. Comparativa de CVS por tendencia (Google Trends).	22
17. Comparativa de consultas en Stack Overflow	22
18. Descarga de Git.....	23
19. Comando pwd en Git Bash	24
20. Nuevo proyecto Java.....	34
21. Primera aplicación Java	34
22. Árbol de proyectos de primera aplicación.....	35
23. JFrame con botón.....	36
24. Control JDialog	36
25. Manejador de evento de botón.....	37
26. Ventana de datos de alumnos.....	39
27. Gaps en un JFrame	40
28. Preview Design.....	40
29. Panel y Layout en Navigator.	41
30. Panel con GridLayout y cuatro botones.....	42
31. Ajuste de propiedad en etiqueta	42
32. Editor de modelo de lista	43
33. Uso de menú y listas	44
34. Pestaña de Servicios	49
35. Drivers JDBC en NetBeans.....	50
36. Conexión a PostgreSQL desde NetBeans.....	50
37. Vista de esquema PostgreSQL.....	51
38. JFrame con JTable	56
39. JTable incrustada en un JPanel	61
40. Encapsulación de variables.	62
41. Vista de diseño de BeanInfo.....	63

42. Agregado de componentes a paleta de controles	64
43. Propiedades del control de usuario.	65
44. Ejemplo de uso de listeners.....	68

BIBLIOGRAFÍA - WEBGRAFÍA

Adamson, C. et al. (2005). *Swing Hacks*. Editorial O'Reilly.

Eckel, B. (2007). *Piensa en Java*. Editorial Pearson Prentice Hall.

Randolph, N. et al. (2010). *Professional Visual Studio 2010*. Editorial Wrox.

Vicente Carro, J.L. (2014). *Desarrollo de interfaces*. Editorial Garceta.

Groussard, T. et al. (2020). *Java 11. Los fundamentos del lenguaje Java*. Editorial Eni.

Dauzon, S. (2022). *Git - Controle la gestión de sus versiones (conceptos, utilización y casos prácticos)*. 2ª edición. Editorial Eni.

Zorrilla, M.E. et al. (2011). *Diseño y desarrollo de aplicaciones de bases de datos*.
<https://ocw.unican.es/pluginfile.php/1839/course/section/1447/Tema%205.pdf>

Cristian Henao (2013). *Contenedores Java Swing*.
<https://codejavu.blogspot.com/2013/10/contenedores-java-swing.html>

González, P. (2016). *UT1. Confección de interfaces de usuario. Java Swing*.
<https://youtu.be/3msU8OXyBB8?si=9ZrPHQ6pYZg3RNaO>

Trętowicz, M. et al. (2022). *MVC MVP and MVVM architecture pattern – Introduction*.
<https://itcraftapps.com/blog/mvc-mvp-and-mvvm-architecture-pattern-introduction/>