



UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática

## **REUTILIZACIÓN DE INTERFACES GRÁFICAS DE USUARIO**

ALBERTO BAUSÁ CANO

Dirigido por: JOSÉ MANUEL CUADRA TRONCOSO

Curso: 2016-2017 (2ª convocatoria - septiembre)





## **REUTILIZACIÓN DE INTERFACES GRÁFICAS DE USUARIO**

Proyecto de Fin de Grado de modalidad general

Realizado por: **Alberto Bausá Cano**

Dirigido por: **José Manuel Cuadra Troncoso**

Tribunal calificador

Presidente: D/D<sup>a</sup>.

Secretario: D/D<sup>a</sup>.

Vocal: D/D<sup>a</sup>.

Fecha de lectura y defensa: 2 de octubre, 2017

Calificación:



# Agradecimientos

Quisiera dedicar este proyecto a mi familia, que siempre ha entendido mi forma de ser, y ha estado ahí apoyándome, especialmente en los malos momentos.

Igualmente, a Maggie, quien me ha ayudado continuamente, aguantando a veces enfados o momentos de estrés, y comprendiendo las exigencias que demandaba lo que estaba haciendo. En definitiva, acompañándome durante esta última etapa de mi vida.

Por supuesto, a mis amigos, con los que siempre he tenido momentos para divertirme, relajarme y distraerme, y con los que a partir de ahora pasaré muchos más de esos momentos.

Tampoco quiero olvidar algunos compañeros conocidos y encontrados a lo largo de estos años de estudio. Puesto que, pese al carácter supuestamente individual o más aislado que tiene una universidad a distancia, como es la UNED, han hecho que nunca me haya sentido solo en el camino, pues eran personas con las mismas inquietudes e intereses que yo, de las que he aprendido bastante, y me han ayudado, y espero que yo a ellos, a progresar en la travesía emprendida.

Por último, quería hacer mención al profesorado de la UNED, pese a que, como en todo, siempre hay mejores y peores, en general siempre han sido un gran apoyo, un punto de referencia que seguir para saber hacia donde avanzar. Y especialmente, quería nombrar a mi director de proyecto, Jose Manuel, pues ha tenido una gran implicación y colaboración durante el desarrollo de mi proyecto, sabiéndome guiar de forma justa, y atendiéndome siempre que lo he necesitado, incluso, y es algo a destacar, estando de vacaciones.

A todos ellos, gracias por hacerme llegar hasta aquí, cerrando esta etapa, y comenzando una nueva, con grandes motivaciones y muchas cosas por hacer, profesional, y personalmente.



# Resumen

En el pasado, la creación de software siempre fue una tarea farragosa y compleja, que requería al menos un editor de texto, para escribir el código fuente de la aplicación, y un compilador o intérprete, que realizase los pasos necesarios para poder ejecutarla, partiendo del conjunto de ficheros que componían el programa. Con el tiempo, las exigencias de los usuarios que utilizaban esos programas crecían, y con ellas, la necesidad de disponer de entornos más amigables que facilitasen su uso. Así surgieron las primeras interfaces gráficas de usuario.

La introducción paulatina de la llamada 'vista de Diseño' de interfaces gráficas, donde los programadores podían construir dichas interfaces de manera visual, facilitó mucho su labor, pues les permitía un desarrollo del software más rápido, con mayor calidad, y menos propenso a errores. Este modo de construir software tiene una relación directa con un patrón, o arquitectura de diseño, conocido como *Modelo-Vista-Controlador*, el cuál separa los datos de la aplicación, la interfaz de usuario, y la lógica de control, en tres componentes claramente diferenciados. Siguiendo esta filosofía, se pueden discernir de forma clara los dos roles principales en todo desarrollo software: el diseñador de interfaces gráficas, y el programador de aplicaciones.

**Este proyecto busca proporcionar apoyo a ambos, desarrollando una librería que permita la reutilización de interfaces gráficas de usuario.** De esta forma, el diseñador, siguiendo unas directrices mínimas, y de forma transparente, podrá construir los diálogos que desee, modificando su apariencia y comportamiento a elección. Una vez los complete, estará en disposición de proporcionárselos al programador. El desarrollador de aplicaciones, partiendo de los diálogos contruidos por el diseñador, podrá combinarlos de diferentes formas, insertando unos dentro de otros, aplicando ligeras recomendaciones o restricciones que aseguren la compatibilidad entre los mismos.

La librería se encargará del resto de acciones necesarias en el proceso, para que todo funcione correctamente. Se ocupará de redimensionar los tamaños para los diálogos agregados, en caso de ser necesario, ajustándose al tamaño del mayor de ellos. Además, orquestará un sistema de paso de mensajes, el cuál actuará a nivel de diálogo, y no de componente interno. Para ello, comunicará todos los diálogos entre si bidireccionalmente, de tal forma que cualquiera de los diálogos podrá enviar un mensaje al resto de ellos, y también recibir un mensaje desde cualquier otro. Por último, se ocupará de arbitrar la implementación de un mecanismo de recursividad para las acciones de validación, guardado y finalización de los diálogos. Dicho mecanismo permitirá que, de cara al usuario, la edición de los datos de un diálogo se perciba como una transacción atómica. Es decir, los datos modificados en cada diálogo de la jerarquía se guardarán o limpiarán en un solo paso.

## Palabras clave

Interfaces gráficas de usuario | Librería | Reusabilidad | Java | Framework Swing | NetBeans

# Abstract

In the past, software development used to be a tedious and complex task, which required, at least, a text editor, in order to write the application source code, and a compiler or interpreter, which was in charge of make necessary steps so as to be able to execute it, starting from the set of files which composed the program. Over time, users' demands who harnessed these programs grew, and with them, the need to have more friendly enviroments which made easier their use. That way, first graphical user interfaces arose.

The gradual introduction of the called 'Design view' of graphical interfaces, where programmers could build those interfaces in a visual way, eased much their labour, because let them do a faster software development, with more quality, and less error prone. This manner of building software is directly related with a pattern, or design architecture, known as *Model-View-Controller*, which splits up application data, user interface, and logic of control, in three clearly differentiated components. Following this philosophy, it can be manifestly discerned the two main roles in every single software development: the graphical interfaces designer, and the application programmer.

**This project aims to provide support** to both of them, **developing a library that allows graphical user interfaces reuse**. This way, the designer, following some minimal guidelines, and transparently, will be able to build the dialogs he wants, editing their apparence and behaviour by choice. Once he completes them, he will be able to provide them to the programmer. The application developer, starting from dialogs built by the designer, will be able to combine them in different ways, inserting them ones within each other, applying slight suggestions or constraints in order to ensure compatibility between them.

The library will take care of the rest of necessary actions along the process, so as to everything works correctly. It will be responsible for make resizing for aggregated dialogs, if necessary, adjusting it to the size of the largest one. In addition, it will orchestrate a message sending system, which will act at dialog tier, but not at internal component tier. For that, it will communicate all dialogs between them bidirectionally, in such a way that any dialog will be able to send a message to the rest of them, and will be able to receive a message from any other. Finally, it will take care of settle the implementation of a recursive mecanism for dialogs' validating, saving or conclusion actions. Such a mecanism will allow that, towards the user, the modification of the dialog data is perceived as an atomic transaction. In other words, the modiflicated data per each dialog in the hierarchy will be saved or cleaned in one single step.

## Keywords

Graphical user interfaces | Library | Reusability | Java | Swing framework | NetBeans



# Índice general

<b>1. Introducción general y objetivos</b>	<b>1</b>
1.1. Introducción . . . . .	1
1.2. Motivación . . . . .	4
1.3. Objetivos . . . . .	5
1.4. Estructura de la memoria . . . . .	7
<b>2. Fundamentos teóricos</b>	<b>11</b>
2.1. Programación Orientada a Objetos . . . . .	12
2.1.1. Herencia . . . . .	12
2.1.2. Polimorfismo . . . . .	13
2.1.3. Sobrecarga . . . . .	14
2.1.4. Métodos abstractos . . . . .	14
2.1.5. Reutilización de código . . . . .	15
2.1.5.1. Librería . . . . .	15
2.2. Patrones de diseño . . . . .	16
2.2.1. Patrón <i>Builder</i> . . . . .	16
2.2.2. Patrón <i>Composite</i> . . . . .	17
2.2.3. Patrón <i>Observer</i> . . . . .	18
2.2.4. Patrón <i>Singleton</i> . . . . .	18
2.2.5. Patrón de <i>Indirección</i> . . . . .	19
2.2.6. Patrón de <i>Reusabilidad</i> . . . . .	19
2.3. Interfaces Gráficas de Usuario . . . . .	19
2.3.1. Arquitectura MVC . . . . .	20
2.4. Infraestructura tecnológica . . . . .	22
2.4.1. Java . . . . .	22
2.4.2. NetBeans . . . . .	23

<b>3. Análisis y diseño del sistema</b>	<b>25</b>
3.1. Estimación de costes . . . . .	26
3.1.1. Librería de software . . . . .	26
3.1.2. Programa de prueba . . . . .	27
3.2. Calendarización . . . . .	27
3.3. Requisitos del sistema . . . . .	29
3.4. Casos de uso . . . . .	33
3.4.1. Casos de uso del programador . . . . .	33
3.4.2. Casos de uso del diseñador . . . . .	36
3.4.3. Casos de uso del usuario . . . . .	39
3.5. Estructura y organización . . . . .	40
<b>4. Implementación</b>	<b>43</b>
4.1. Detalles generales de implementación . . . . .	44
4.2. <code>IMessageObserver</code> . . . . .	47
4.2.1. Método <i>getExternVal</i> . . . . .	47
4.3. <code>IRecursiveActions</code> . . . . .	48
4.3.1. Método <i>validateThis</i> . . . . .	48
4.3.2. Método <i>saveThis</i> . . . . .	48
4.3.3. Método <i>cleanThis</i> . . . . .	48
4.4. <code>JIAExtensibleDialog</code> . . . . .	49
4.4.1. Método <i>setUpDialog</i> . . . . .	50
4.4.2. Método <i>addExtensibleChild</i> . . . . .	50
4.4.3. Método <i>addExtensibleChildrenList</i> . . . . .	50
4.4.4. Método <i>linkListeners</i> . . . . .	51
4.4.5. Método <i>validar</i> . . . . .	51
4.4.6. Método <i>salvar</i> . . . . .	52
4.4.7. Método <i>limpiar</i> . . . . .	53
4.4.8. Método <i>cambiaVal</i> . . . . .	53
4.4.9. Método <i>resizeThis</i> . . . . .	53
4.5. <code>JIASimpleDialog</code> . . . . .	54
4.5.1. Método <i>setUpDialog</i> . . . . .	54
4.5.2. Método <i>addExtensibleChild</i> . . . . .	54
4.5.3. Método <i>addExtensibleChildrenList</i> . . . . .	55
4.5.4. Método <i>addWithConstraints</i> . . . . .	55
4.5.5. Método <i>resizeThis</i> . . . . .	56

4.6.	JITabDialog . . . . .	56
4.6.1.	Método <i>setUpDialog</i> . . . . .	57
4.6.2.	Método <i>addExtensibleChild</i> . . . . .	57
4.6.3.	Método <i>addExtensibleChildrenList</i> . . . . .	58
4.6.4.	Método <i>generateInternalPane</i> . . . . .	58
4.6.5.	Método <i>resizeThis</i> . . . . .	59
4.7.	JITreeViewDialog . . . . .	60
4.7.1.	Método <i>setUpDialog</i> . . . . .	61
4.7.2.	Método <i>addExtensibleChild</i> . . . . .	61
4.7.3.	Método <i>addExtensibleChildrenList</i> . . . . .	62
4.7.4.	Método <i>generateInternalNode</i> . . . . .	63
4.7.5.	Método <i>addDialogsToContent</i> . . . . .	64
4.7.6.	Método <i>checkRepeatedName</i> . . . . .	64
4.7.7.	Método <i>resizeThis</i> . . . . .	64
4.8.	JIAFactory . . . . .	66
4.8.1.	Método <i>createDialog</i> . . . . .	66
4.8.2.	Método <i>createInstance</i> . . . . .	66
4.8.3.	Método <i>getInstance</i> . . . . .	66
4.8.4.	Método <i>clone</i> . . . . .	67
4.9.	JIAUtils . . . . .	67
4.9.1.	Método <i>createWithTrivialImplementation</i> . . . . .	68
4.9.2.	Método <i>addMinimalStructure</i> . . . . .	69
4.9.3.	Método <i>resetButtons</i> . . . . .	70
4.9.4.	Método <i>generateTabbedPane</i> . . . . .	71
4.9.5.	Método <i>generateSplitPane</i> . . . . .	71
4.9.6.	Método <i>setPreferredSizeByType</i> . . . . .	72
4.9.7.	Método <i>wrapExtensibleDialog</i> . . . . .	72
<b>5.</b>	<b>Experimentación y pruebas</b>	<b>73</b>
5.1.	Variedad de tipos de diálogos . . . . .	73
5.2.	Posibilidades de integración y combinación de diálogos . . . . .	76
5.3.	Edición de datos mediante transacción atómica . . . . .	77
5.4.	Mecanismo de paso de mensajes . . . . .	78
<b>6.</b>	<b>Conclusiones y trabajos futuros</b>	<b>81</b>
6.1.	Conclusiones . . . . .	81
6.2.	Trabajos futuros . . . . .	83

<b>A. Manual de instalación del entorno</b>	<b>87</b>
A.1. Importar la librería . . . . .	87
A.2. Instalar el módulo NBM . . . . .	89
A.2.1. Uso del módulo para la creación de interfaces gráficas . . . . .	92
A.3. Importar los <i>JavaBeans</i> en la Paleta . . . . .	93
<b>B. Manual de uso del programador</b>	<b>97</b>
B.1. Papel y objetivo de uso . . . . .	97
B.2. Forma de interacción con los diálogos . . . . .	99
B.3. Limitaciones y sugerencias de uso . . . . .	100
B.3.1. Inserción en diálogos de tipo Simple creados por el diseñador . . . . .	100
B.3.2. Creación de diálogos vacíos ajenos a la factoría . . . . .	101
B.3.3. Error al exceder el número de hijos . . . . .	101
B.4. Ampliaciones disponibles . . . . .	102
B.5. Ejemplos de utilización . . . . .	105
B.5.1. Diálogos simples . . . . .	105
B.5.2. Diálogos de pestañas . . . . .	106
B.5.3. Diálogos con vista de árbol . . . . .	106
<b>C. Manual de uso del diseñador</b>	<b>111</b>
C.1. Papel y objetivo de uso . . . . .	111
C.2. Forma de interacción con los diálogos . . . . .	113
C.3. Limitaciones y sugerencias de uso . . . . .	114
C.3.1. Variedad de tipos de diálogos disponibles . . . . .	114
C.3.2. Creación de diálogos a partir de plantillas sin paquete asociado . . . . .	115
C.3.3. Implementación de los métodos de compatibilidad con la librería . . . . .	116
C.3.4. Uso de los <i>JavaBeans</i> . . . . .	118
C.3.4.1. Inserción de <i>beans</i> como diálogos hijos . . . . .	119
C.4. Ampliaciones disponibles . . . . .	120
C.5. Ejemplos de utilización . . . . .	121
C.5.1. Creación de diálogos a partir de plantillas . . . . .	121
C.5.2. Inserción de diálogos desde la Paleta . . . . .	121
<b>D. Manual del programa de pruebas</b>	<b>125</b>
D.1. Estructura y características . . . . .	125
D.2. Flujos de ejecución . . . . .	129
D.2.1. Interacción con el rol de 'Triaje' . . . . .	129

---

D.2.2. Interacción con el rol de 'Facultativo' . . . . .	135
D.3. Ejemplos de uso de la librería . . . . .	135
D.3.1. Variedad en la inserción de diálogos reutilizados . . . . .	137
D.3.2. Paso de mensajes entre diálogos independientes . . . . .	139



# Nomenclatura

COCOMO	COConstructive COst MOdel (Modelo Constructivo de Costes),	página 25
GRASP	General Responsibility Assignment Software Patterns (Patrones Generales de Software para Asignación de Responsabilidades),	página 19
GUI	Graphical User Interface (Interfaz Gráfica de Usuario),	página 19
IDE	Integrated Development Enviroment (Entorno de Desarrollo Integrado),	página 2
JAR	Java ARchive (Archivo Java),	página 23
JDK	Java Development Kit (Kit de Desarrollo Java),	página 87
JLS	Java Language Specification (Especificación del Lenguaje Java),	página 115
LOC	Lines Of Code (Líneas De Código),	página 26
MIC	Modelo-Interfaz-Controlador,	página 21
MVC	Modelo-Vista-Controlador,	página 11
NBM	NetBeans Module (Módulo NetBeans),	página 23
NUI	Natural User Interface (Interfaz Natural de Usuario),	página 20
PF	Punto de Función,	página 26
POO	Programación Orientada a Objetos,	página 11
SGH	Software de Gestión Hospitalaria,	página 125
SLOC	Source Lines Of Code (Líneas De Código Fuente),	página 26
UML	Unified Modeling Language (Lenguaje Unificado de Modelado),	página 8
VUI	Voice User Interface (Interfaz mediante Voz de Usuario),	página 20

WIMP Windows, Icons, Mouse, Pointer (Ventanas, Iconos, Ratón, Puntero), página 20

WORA Write Once, Run Anywhere (Escribir una vez, ejecutar en cualquier parte), página 22



# Índice de figuras

1.1. Vista de codificación de un IDE . . . . .	3
1.2. Vista de diseño de un IDE . . . . .	3
1.3. Esquema general del planteamiento del proyecto . . . . .	5
2.1. Herencia representada como un árbol jerárquico . . . . .	13
2.2. Arquitectura MVC pura . . . . .	21
2.3. Arquitectura MVC Cocoa (Apple) . . . . .	21
3.1. Diagrama de Gantt del proyecto . . . . .	28
3.2. Diagrama de casos de uso para el Programador . . . . .	33
3.3. Diagrama de casos de uso para el Diseñador . . . . .	36
3.4. Diagrama de casos de uso para el Usuario . . . . .	39
3.5. Diagrama de clases de la librería . . . . .	41
5.1. Ejemplo de uso de diálogo de tipo Simple . . . . .	74
5.2. Ejemplo de uso de diálogo de tipo Pestañas . . . . .	75
5.3. Ejemplo de uso de diálogo de tipo Vista de árbol . . . . .	75
5.4. Pantalla de datos bancarios de un paciente . . . . .	76
A.1. Pantalla inicial de NetBeans . . . . .	88
A.2. Menú para agregar dependencias . . . . .	88
A.3. Librería añadida correctamente . . . . .	89
A.4. Menú de herramientas NetBeans . . . . .	90
A.5. Ventana de Plugins en NetBeans . . . . .	90
A.6. Ventana de Plugins con uno nuevo . . . . .	91
A.7. Mensaje de aviso sobre la validación del módulo . . . . .	91
A.8. Ventana de Archivo Nuevo en NetBeans . . . . .	92
A.9. Diálogo con Vista de Árbol creado con el template . . . . .	93
A.10. Acceso al 'Administrador de paleta' . . . . .	94

A.11. Administrador de paleta de NetBeans . . . . .	94
A.12. Selección de beans a añadir a la paleta . . . . .	95
A.13. <i>JavaBeans</i> añadidos correctamente en la paleta . . . . .	95
B.1. Esquema básico del programador de aplicaciones . . . . .	98
B.2. Excepción al añadir un 3er diálogo a uno de tipo Simple . . . . .	102
B.3. Diálogo simple vacío . . . . .	107
B.4. Diálogo simple con un hijo . . . . .	107
B.5. Diálogo simple con dos hijos . . . . .	107
B.6. Diálogo de pestañas vacío . . . . .	108
B.7. Diálogo de pestañas con inserción sencilla . . . . .	108
B.8. Diálogo de pestañas con inserción de lista . . . . .	108
B.9. Diálogo con vista de árbol vacío . . . . .	109
B.10. Diálogo con vista de árbol con inserción sencilla . . . . .	109
B.11. Diálogo con vista de árbol con inserción de lista . . . . .	109
C.1. Esquema básico del diseñador de interfaces gráficas . . . . .	112
C.2. Error producido al crear el diálogo en la raíz de paquetes . . . . .	115
C.3. Excepción que se lanza fruto de la implementación trivial . . . . .	116
C.4. Mecanismo recursivo lanzado por la librería . . . . .	117
C.5. <i>Beans</i> disponibles en la Paleta de NetBeans . . . . .	117
C.6. Ventana de creación de un 'Archivo nuevo' . . . . .	122
C.7. Diálogo con vista de árbol creado con una plantilla . . . . .	122
C.8. Ejemplo sección C.5.2, diálogo de pestañas inicial . . . . .	123
C.9. Ejemplo sección C.5.2, añadiendo <i>bean</i> de diálogo simple . . . . .	124
C.10. Ejemplo sección C.5.2, <i>bean</i> simple añadido . . . . .	124
D.1. Estructura de la aplicación de prueba . . . . .	126
D.2. Pantalla de inicio de sesión . . . . .	129
D.3. Menú principal Triaje . . . . .	130
D.4. Factura médica para un paciente . . . . .	132
D.5. Resumen del paciente, con los datos básicos . . . . .	132
D.6. Modificación de los datos generales . . . . .	133
D.7. Modificación de los datos personales . . . . .	133
D.8. Modificación de los datos clínicos . . . . .	134
D.9. Modificación de los datos bancarios . . . . .	134
D.10. Menú principal facultativo . . . . .	136

D.11. Mensaje de confirmación de atención a un paciente . . . . .	136
D.12. Ficha de paciente, con el resumen y los datos separados . . . . .	137
D.13. Diálogo de DatosBancarios, insertado en un diálogo Simple . . . . .	138
D.14. Diálogo de DatosBancarios, insertado en un diálogo de Pestañas . . . . .	138
D.15. Diálogo de DatosBancarios, insertado en un diálogo con Vista de árbol . . . . .	139
D.16. Caso A. Estado inicial, antes de recibir el mensaje . . . . .	140
D.17. Caso A. Estado inicial, antes de enviar el mensaje . . . . .	140
D.18. Caso A. Estado posterior, después de enviar el mensaje . . . . .	141
D.19. Caso A. Estado posterior, después de recibir el mensaje . . . . .	141
D.20. Caso B. Estado inicial, antes de recibir el mensaje . . . . .	141
D.21. Caso B. Estado inicial, antes de enviar el mensaje . . . . .	142
D.22. Caso B. Estado posterior, después de enviar el mensaje . . . . .	142
D.23. Caso B. Estado posterior, después de recibir el mensaje . . . . .	142



# Capítulo 1

## Introducción general y objetivos

En este capítulo se relatarán, de manera general y global, los objetivos y propósitos del proyecto, sin entrar prácticamente en detalles de análisis, diseño o implementación, lo cuál se dejará para más adelante. Se presentará como una primera toma de contacto con el trabajo realizado.

En primera instancia, se esbozará una breve introducción, con ciertos tintes históricos, donde se comentará el proceso de construcción tradicional de software, y la aparición de las primeras interfaces gráficas de usuario, las cuales, junto con la vista de Diseño que las acompañó, supuso un auténtico punto de disrupción en el sector informático de desarrollo.

En dicho relato, llegaremos hasta el punto actual del *estado del arte* del desarrollo software, el cuál se sustenta, entre otros, en dos roles principales, el del programador, y el del desarrollador de interfaces gráficas. Lo que vendría a ser, traído a términos más actuales, las partes *front-end* y *back-end* del diseño de aplicaciones. En otras palabras, la capa de presentación del programa al usuario, y la capa de acceso a los datos del programa, respectivamente.

Partiendo de ese estado, se comenzará a hablar de la motivación original y fundamental de la librería, que no es otra que la reutilización de interfaces gráficas, lo cual da nombre al proyecto.

Después de desarrollar brevemente esa idea, se procederá, a continuación, a listar y explicar los objetivos básicos que persigue la librería, en su concepción, diseño, construcción y uso.

Por último, se cerrará el capítulo exponiendo la estructura general de la memoria, para lo cuál se enumerarán los capítulos y anexos, haciendo una pequeña reseña de cada uno, con el fin de dar una idea primordial y primeriza al lector, de los contenidos que se encontrará.

### 1.1. Introducción

Tradicionalmente, la programación siempre ha sido una misión ardua y laboriosa, donde se requería al menos un editor de texto, para crear el código fuente, y un traductor, bien fuese un compilador o bien fuese un intérprete, el cuál se encargase de reunir el conjunto de ficheros que componían la

aplicación, y ejecutarlos.

Según iban pasando los años, las exigencias de los usuarios de dichas aplicaciones crecían, pues cada vez se requería poder hacer más cosas, y de una forma más rápida y cómoda. Con sus exigencias, aumentaba también la necesidad de disponer de entornos de uso más amigables e intuitivos, que facilitasen el manejo de los programas. Así pues, surgieron las interfaces gráficas de usuario, las cuales no eran más que un conjunto de ventanas y menús que permitían al usuario interactuar con el software de una manera más sencilla y satisfactoria.

Con esas interfaces gráficas, comenzaron a aparecer también los primeros programas conocidos como IDEs, que facilitaban el trabajo del programador que quería desarrollar este tipo de software. Estos programas introdujeron la posibilidad de disponer de una *vista de Diseño* para esas interfaces, que permitía el planteamiento y construcción de las mismas mediante una vía mucho más visual e intuitiva, en comparación al código fuente a secas, el cual en ocasiones podía llegar a ser realmente complejo o enrevesado.

De esta forma, se tenía un escenario donde los IDEs ofrecían dos perspectivas o formas de enfocar el código fuente del desarrollo de un programa:

- Por un lado, la *vista de Codificación*, de la cual se muestra un ejemplo en la figura 1.1. En ella, el programador o desarrollador de aplicaciones podía acceder de manera directa al código fuente, pudiendo consultarlo o modificarlo. La forma usual de presentar esta vista era mediante un documento de texto, coloreando la sintaxis del lenguaje en uso, con el fin de aumentar la legibilidad del código, facilitando la tarea.
- Por otro lado, la *vista de Diseño*, de la cual se muestra un ejemplo en la figura 1.2. La cuál posibilitaba que los diseñadores creasen las interfaces, simplemente colocando y modificando gráficamente sus componentes, mientras que el IDE, en segundo plano, se encargaba de generar el código necesario. En ella, normalmente, se tiene una sección donde se muestra el estado actual de la interfaz, de forma visual, y una zona desde la cuál se pueden seleccionar los componentes gráficos disponibles, permitiendo elegir de forma sencilla cuál utilizar, de entre ventanas, botones, etiquetas, cajas de texto, etc.

Esta segunda opción supuso un enorme avance, pues proporcionó a los desarrolladores formas mucho más rápidas de construir las aplicaciones, con menos esfuerzo, y consiguiendo además productos finales de mayor calidad general.

Esta forma de desarrollo de software, con la clásica *vista de Codificación*, para el código fuente, y la nueva *vista de Diseño*, para las interfaces gráficas, tiene mucho que ver con una arquitectura de diseño de aplicaciones conocida como 'Modelo-Vista-Controlador', que aquí se reseña brevemente, y de la cuál se hablará de forma más extensa en la sección 2.3.1. Acorde a la filosofía de dicho patrón, todo programa se divide en 3 componentes claramente diferenciados:

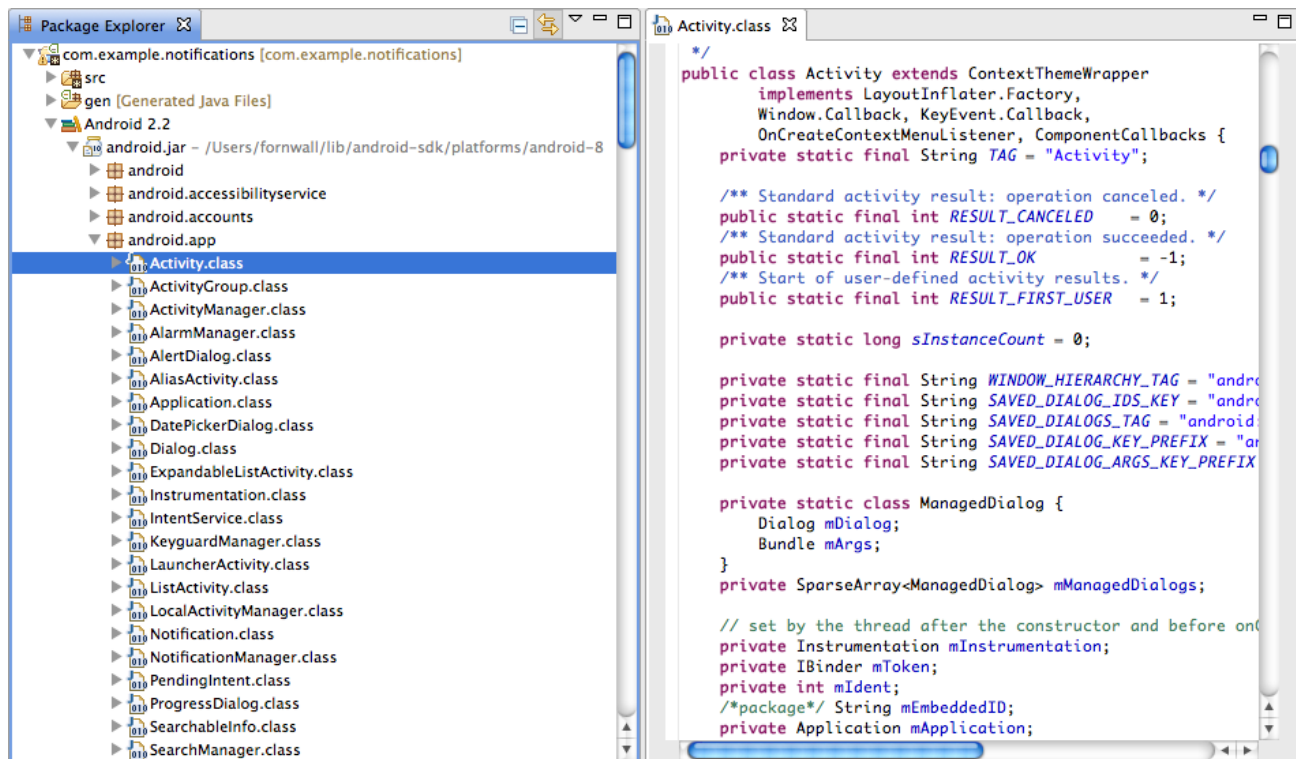


Figura 1.1: Vista de codificación de un IDE

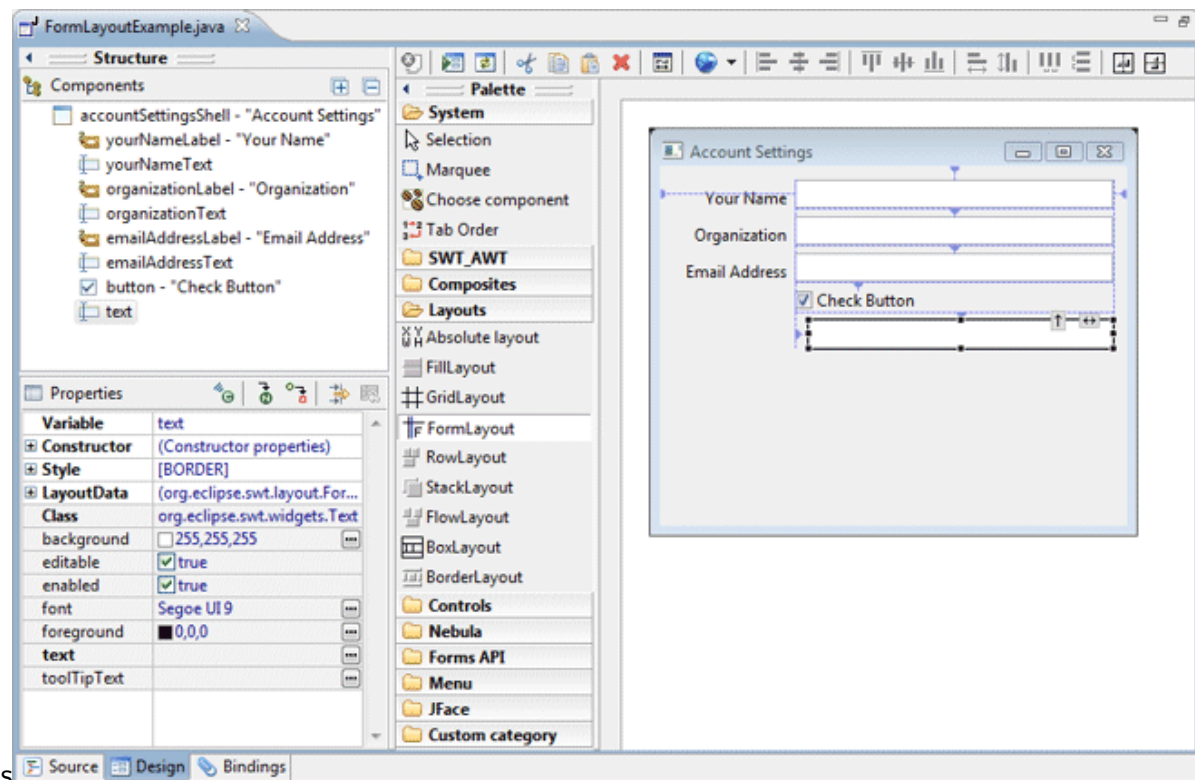


Figura 1.2: Vista de diseño de un IDE

- La Vista, que será la interfaz de usuario, compuesta por las diferentes pantallas y ventanas.
- El Modelo, los datos e información que manejar y representa la aplicación.
- El Controlador, el cuál albergará toda la lógica que comunique la vista con el controlador, en ambas direcciones, y se encargará de conducir el flujo de ejecución del programa.

Siguiendo esa línea de acontecimientos, y tras esta breve introducción, se llega al punto actual, en el cuál se da cabida a la concepción del desarrollo del proyecto y librería que nos ocupan.

## 1.2. Motivación

La idea original del proyecto actual, nace motivada por el pensamiento de mejorar o simplificar, de alguna forma, el proceso de desarrollo de software, y ayudar a las personas que se ocupan de alguno de los roles encargados de elaborarlo, suministrando soporte dentro del esquema de construcción de aplicaciones expuesto anteriormente.

Este proyecto, por tanto, busca proporcionar apoyo y facilidades a los desarrolladores, brindando un refuerzo enfocado tanto a los programadores de las aplicaciones, como a los diseñadores o constructores de las interfaces gráficas de usuario. Es decir, pretende ser útil en ambos lados del proceso de desarrollo, abarcando un espectro más general.

Para ello, se plantea la construcción de la herramienta fundamental en tal encomienda, una librería de código, que permita la reutilización, por parte del programador, de interfaces gráficas de usuario que hayan sido creadas por el diseñador.

Esta propuesta podría ayudar en dos escenarios distintos, que aquí se diferencian, aunque en la práctica puedan aparecer combinados o entrelazados:

- El primero, es la posibilidad de crear una jerarquía de interfaces gráficas, construidas por el diseñador, que permitan modificar a su vez una jerarquía de clases de un modelo, generadas quizá por un programador, existiendo entre ambas jerarquías una colección de relaciones de correspondencia mutua. En otras palabras, cada entidad del modelo debería tener su equivalente aproximado en una interfaz de la jerarquía opuesta, considerando siempre la posible existencia de clases que no pertenezcan directamente al dominio del problema, como clases de apoyo, o con funcionalidades auxiliares, las cuales no tendrán, ni tendría sentido que tuvieran, un equivalente gráfico.
- El segundo, es la existencia de un marco en el que se quisieran integrar interfaces gráficas con objetos asociados en una GUI general, pero conservando la posibilidad de mostrar cada diálogo individualmente en caso de ser necesario. Es decir, sirva como ejemplo, un caso en el que



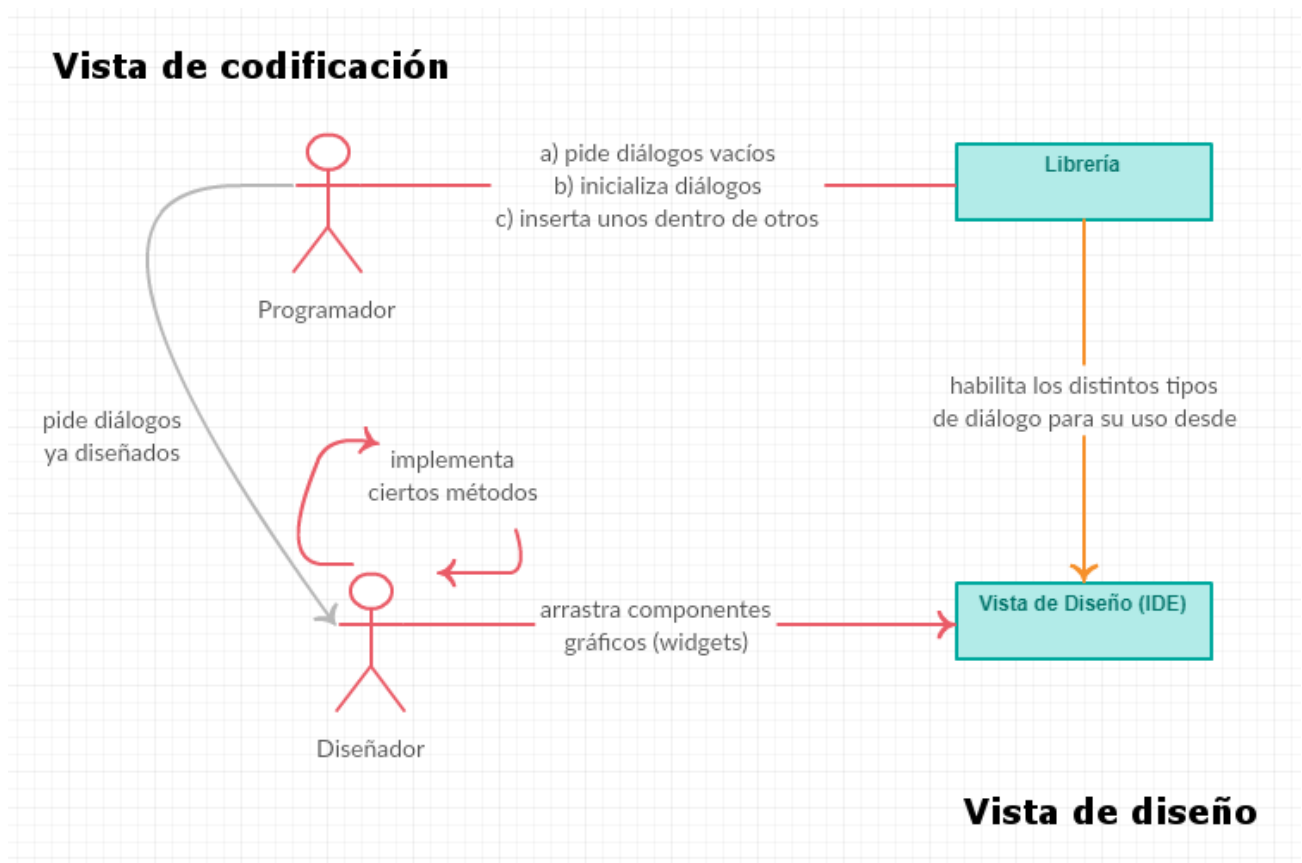


Figura 1.3: Esquema general del planteamiento del proyecto

se tuvieran diálogos de un tamaño pequeño o mediano, los cuales se insertasen y mostrasen, combinados de alguna forma, en un diálogo de tamaño mayor. En dicha circunstancia, a pesar de tener el diálogo ya formando parte de otro, se debería seguir teniendo siempre la oportunidad de seguir presentando, de forma aislada, esos mismos diálogos integrados.

Para ambas circunstancias, se debería evitar duplicar código o componentes gráficos, y, además, el proceso debería ser transparente para los usuarios, sean éstos programadores o diseñadores:

### 1.3. Objetivos

En esta sección se explicarán los propósitos que persigue la librería y su construcción, yendo punto por punto hasta recorrer las metas principales. Para ello, se presenta la figura 1.3, que encuadra el contexto funcional en el que tendrá que actuar la librería desarrollada.

En ella se pueden distinguir los dos roles principales comentados anteriormente, los cuales intervienen en el desarrollo de una aplicación: el programador, o desarrollador de aplicaciones, y el diseñador, o constructor de interfaces gráficas de usuario.

A partir de aquí, se pueden distinguir varios objetivos de los que la librería tendrá que encargarse.

- Por un lado, desde el *punto de vista del desarrollador*, todas las tareas se realizarán desde la vista de codificación del IDE correspondiente, en este caso NetBeans. La librería deberá ofrecerle al programador la posibilidad de:
  - Pedir diálogos vacíos, de tal forma que tenga un espacio en el cuál comenzar a combinar otros diálogos de la forma que necesite para su aplicación.
  - Pedir diálogos al diseñador, los cuáles ya estén contruidos, y tengan una apariencia y comportamiento definidos. Este no es un objetivo perseguido directamente por la librería, pero si que es la librería la que establece unas pautas mínimas que permitirán que los diálogos del diseñador puedan ser utilizados por el programador, y formen parte de la compatibilidad global de reusabilidad.  
Por este motivo se incluye aquí, de forma indirecta.
  - Inicializar los diálogos, para que tengan preparadas sus características de reusabilidad. El programador podrá inicializar tanto los diálogos vacíos pedidos a la librería, como aquellos pedidos al diseñador, pues formarán todos parte de la misma jerarquía de diálogos reusables. Es decir, heredarán todos propiedades de un mismo tipo de diálogo común, que será la raíz de la jerarquía mencionada.
  - Integrar diálogos hijos dentro de diálogos padres. Este punto es fundamental dentro de la librería. Con él, se pretende que el programador tenga disponibles diferentes opciones a la hora de mezclar diferentes tipos de diálogos, y que pueda hacerlo de maneras diversas.
- Por otro lado, desde el *punto de vista del diseñador*, sus labores las desempeñará en la vista de diseño del entorno de desarrollo. Por tanto, es importante resaltar que la librería deberá ofrecer la alternativa de habilitar sus distintos tipos de diálogos para que puedan ser usados desde esa vista de diseño. Así, el diseñador deberá ser capaz de:
  - Arrastrar componentes gráficos hacia su zona de trabajo, pudiendo ser estos componentes los distintos tipos de diálogo con los que trabaja la librería. Podrá operar sobre ellos, añadiéndoles otros componentes, y personalizando su aspecto y estilo de la forma que desee. En este sentido, trabajará como si se tratase de cualquier otro componente del framework gráfico, sin restricciones que limiten su operación.
  - Además, deberá implementar ciertos métodos básicos para el funcionamiento del proceso a desarrollar. Sin su redefinición, dichos métodos no actuarían correctamente, desembocando en un comportamiento no deseado de la librería y los diálogos que maneja.

Otro aspecto importante que se debe señalar, el cuál se trata de una faceta transversal, es el de que la librería deberá articular un sistema de comunicación entre sus diálogos.

Deberá ser posible siempre que éstos se encuentren dentro de la misma jerarquía, entendiendo esta jerarquía como el hecho de que los diálogos se pueden insertar unos dentro de otros, existiendo relaciones de parentesco entre padres e hijos, y que dichos hijos, a su vez, podrán tener más descendientes, de forma sucesiva. De esta forma, se puede imaginar una especie de estructura de árbol, donde su raíz será el diálogo principal.

Así pues, la librería deberá permitir que los diálogos puedan transmitirse información entre ellos, aún cuando dicha conducta no estuviera prevista en la concepción de cada diálogo por separado, cuando se construyesen de forma individual.

## **1.4. Estructura de la memoria**

La memoria de esta proyecto se estructura en los siguientes capítulos:

1. Introducción general y objetivos
2. Fundamentos teóricos
3. Análisis y diseño del sistema
4. Implementación
5. Experimentación y pruebas
6. Conclusiones y trabajos futuros

### **Introducción general y objetivos**

Este es un capítulo introductorio, que se encarga de proporcionar una primera toma de contacto al lector que se dispone a leer la memoria. En él se trata de establecer el marco de trabajo sobre los cuáles se actuará a posteriori, desarrollándolos en el resto de capítulos.

Pretende servir para que el lector atraído por el proyecto realice una inmersión preliminar en sus contenidos, familiarizándose con ellos.

Se espera que ayude también a establecer claramente la idea que pretende desarrollar este trabajo, y por así decirlo convenza a todos aquellos interesados, en continuar la lectura y entendimiento del resto del documento.

## **Fundamentos teóricos**

Este capítulo versará sobre todos aquellos elementos utilizados a lo largo del proyecto, que merezcan una especial dedicación o atención, ofreciendo una explicación clara y concisa sobre ellos. Esos elementos irán desde conceptos teóricos, metodologías, procesos de desarrollo, a tecnologías empleadas en la construcción de la librería, pasando por características propias de lenguajes o herramientas fundamentales en el progreso general.

Se realizará una introducción que sitúe el contexto del proyecto, y auxilie al lector no experimentado en la materia, a ubicarse y poder entender con mayor precisión, de aquí en adelante, los temas tratados en el texto.

## **Análisis y diseño del sistema**

En este punto existe una correspondencia directa con dos fases típicas de todo desarrollo de software, como son las fases de Análisis y Diseño del sistema bajo construcción.

Es por ello, que en este capítulo se hará uso de herramientas propias de ingeniería del software, como es el lenguaje de modelado UML, para representar algunos tipos de diagramas. Entre ellos, estarán los diagramas de casos de uso del sistema, los cuáles además, serán ampliamente detallados de forma textual.

## **Implementación**

Aquí se tratará de realizar un esclarecimiento general sobre el código desarrollado, dando aclaraciones lo más descriptivas posibles acerca de las clases que componen el sistema.

Para ello, se comenzará mostrando un diagrama de clases minucioso del sistema, donde se podrá apreciar la estructura organizativa general de la librería.

Asimismo, se comentarán los métodos que componen la interfaz pública de las clases. En el caso de los métodos ajenos a dicha interfaz pública, es decir, aquellos que sean protegidos o privados, se explicarán en los casos donde sea relevante hacerlo, debido a la transcendencia o repercusión que tengan dichos métodos en el conjunto general del flujo de ejecución.

## **Experimentación y pruebas**

En este apartado se hará hincapié en resaltar los experimentos y pruebas a los que se ha visto sometido el sistema construido, tanto la librería, como los módulos auxiliares. Se describirá de forma somera, debido a que ya existe un manual propio, el programa de pruebas pedido y realizado. También se mostrará de forma clara la fiabilidad e integridad del conjunto, así como de las partes que lo componen.

## **Conclusiones y trabajos futuros**

Este capítulo final servirá como cierre al cuerpo principal del documento, y en él se manifestarán las conclusiones obtenidas a lo largo de todo el proyecto.

Asimismo, se comentarán algunas extensiones o ampliaciones posibles a realizar, como parte de un proceso de mejora del sistema, más concretamente de la librería. Se explicarán brevemente los fundamentos de esas expansiones, y una posible vía de aplicación, sin entrar en demasiados detalles.

Además de eso, se elaboran y presentan al final del documento, en forma de anexos, una serie de manuales de diversa índole:

1. Manual de instalación.
2. Manual de uso del programador.
3. Manual de uso del diseñador.
4. Manual del programa de prueba.

### **Manual de instalación**

Tratará de detallar los pormenores relativos a la instalación y configuración iniciales de la librería, así como del módulo auxiliar asociado.

### **Manual de uso del programador**

Se ocupará de explicar el papel del programador, o desarrollador de aplicaciones, que tenga intención de usar la librería, resaltando sus objetivos de uso, las alternativas que tiene disponibles, así como ligeras limitaciones que sería recomendable tener en cuenta.

### **Manual de uso del diseñador**

Expondrá la función principal que debe cumplir el diseñador de interfaces gráficas que se dispone a realizar un uso de la librería, resaltando las opciones que tiene disponibles, así como mostrando una visión global y específica de su situación dentro del desarrollo de un proyecto que pretenda hacer uso de la librería.

### **Manual del programa de prueba**

Abarcará el trabajo realizado con el programa de pruebas, comentando y ejemplificando su estructura, flujo de operación, así como particularidades específicas, mostrando además ejemplos concretos de uso de la librería en partes señaladas de la aplicación, con el fin de mostrar las posibilidades que ofrece la librería.



## Capítulo 2

# Fundamentos teóricos

Este capítulo pretende realizar una introducción, eminentemente teórica, a muchos de los conceptos, mecanismos, tecnologías y herramientas de los que se habla a lo largo del documento, los cuáles han sido utilizados además, en mayor o menor proporción, y de forma más o menos directa, en el desarrollo del proyecto y la construcción del producto final objetivo del mismo, la librería.

Se elaborará un repaso a todos los elementos utilizados, ofreciendo una descripción acerca de los mismos, con el fin de brindar un contexto adecuado para el proyecto, que permita ubicarse y comprender mejor el conjunto al lector no avezado en la materia.

Se intentará que las explicaciones no sean excesivamente prolijas, pues el fin del capítulo es más introductorio que didáctico. Además, se desea dar una organización a los contenidos, de tal forma que comiencen por aquellos más puramente teóricos o generales, hasta pasar a los términos más prácticos y tangibles, como son las herramientas y tecnologías utilizadas.

Así pues, empieza con un apartado dedicado a hablar sobre la programación, más concretamente la programación orientada a objetos, POO, resumiendo brevemente los puntos históricos más importantes, y pasando a cubrir su descripción general. Dentro de esta sección se consideran también algunas nociones básicas propias de la POO, como son los conceptos de herencia y polimorfismo, entre otros, pasando por un punto clave en este proyecto, como es el de reusabilidad de código, característica muy importante dentro de la POO, y que ofrece la idea fundamental tratada en el documento, el significado de librería software.

El capítulo continúa hablando de otra idea recurrente a lo largo del trabajo, los patrones de diseño software, sobre los cuáles se dará en primer lugar una visión global, relatando su idea general, así como explicando después, con mayor profundidad, alguno de los utilizados: el patrón *composite*, el patrón *listener* u *observer*, y el patrón *builder*, entre otros.

Seguidamente, se habla sobre las interfaces gráficas de usuario, y dentro de ellas, una pieza significativa dentro de su desarrollo, el modelo de construcción de aplicaciones basado en la arquitectura MVC, el cuál se ilustrará visualmente, dado que su naturaleza se presta a ello.

Se pasa luego a una parte menos teórica, al hablar acerca de la infraestructura tecnológica que sustenta el proyecto. En este apartado encontramos a Java, y dentro de la sección, Swing, el framework gráfico utilizado como engranaje a lo largo del proyecto, y *JavaBeans*, una tecnología que define un modelo de componentes para la construcción de aplicaciones. Para finalizar, se cierra el capítulo con una breve reseña sobre NetBeans, comentando algo de sus módulos y *templates*.

## 2.1. Programación Orientada a Objetos

La programación orientada a objetos (Edgar (2002)) nace a finales de los años 60, con el lenguaje de programación *Simula*, considerado el primer lenguaje dentro de este paradigma de programación, dándole origen al mismo (Cmgustavo (2004)). En este primer acercamiento, ya se incluían conceptos fundamentales como los de 'clase' u 'objeto'. Más tarde, se fueron desarrollando lenguajes sucesivos, como *Smalltalk*, y *C++*, el cuál fue una de las causas principales de que este modelo se fuese convirtiendo en el predominante hacia mediados de los años 80.

Esa influencia se fue consolidando con el surgimiento de las primeras interfaces gráficas, para las que la orientación a objetos tenía una predisposición natural. Y más tarde, con la aparición de Java, uno de los lenguajes de programación más utilizados a lo largo de la historia, y aún en la actualidad, donde mantiene su posición como una de las primeras alternativas.

La POO es fundamentalmente un paradigma de programación, es decir, una propuesta de desarrollo, que trata de alterar la forma de procesar la información, en relación a como se hacía tradicionalmente en paradigmas anteriores. Para ello, propone un modelo donde la base fundamental es la utilización de 'objetos' como elementos funcionales básicos, en la construcción de la solución a un problema. Un 'objeto', dentro de la POO, vendría a ser algún hecho o entidad del mundo real, modelable en la aplicación, que tiene una serie de propiedades que definen su estado, y puede operar a través de una serie de métodos, que representan su comportamiento.

Además, estos objetos se agrupan en 'clases', cada una de las cuales representa prototipos con los que crear (*instanciar*) dichos objetos. Estas clases permiten agrupar todas las características y conductas comunes a los objetos de un mismo tipo. Por ello, se considera que cada objeto es como un ejemplar de la clase que representa el tipo al que pertenece.

A continuación se presentan algunos de los conceptos más interesantes que engloba la POO.

### 2.1.1. Herencia

Es uno de los mecanismos más utilizados dentro de la POO, y permite alcanzar algunos objetivos deseables, como son la reusabilidad o la extensibilidad del código desarrollado (Vialfa (2017)). Permite crear nuevas clases hijas a partir de otras ya existentes, denominadas padres. El término 'herencia' en



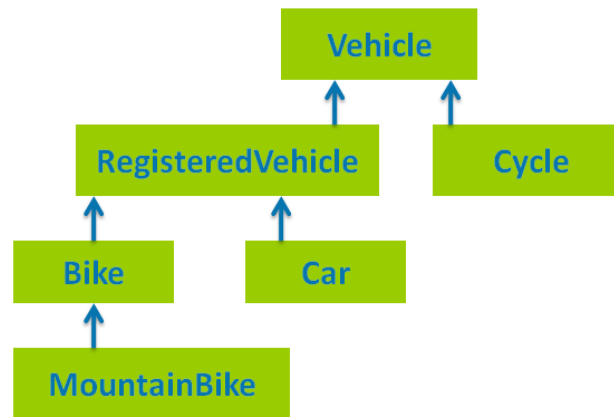


Figura 2.1: Herencia representada como un árbol jerárquico

si mismo, está basado en el hecho de que los hijos creados reciben todas las propiedades y métodos de su clase padre. La principal ventaja de esto, es que dada una base ya construida, se reutiliza, no teniendo que rehacerla de cero, y se puede ampliar con nuevas propiedades o funciones.

Esta relación de parentesco entre padre e hijo puede inducir a pensar en una estructura jerárquica, representada con forma de árbol, como la mostrada en la figura 2.1. En ella existen 3 tipos de nodos: la raíz de la jerarquía (*Vehicle*), clase padre de todas las demás, las clases intermedias (*Bike*), padres e hijas respectivamente de otras clases, y las hojas del árbol (*Car*), que serían las que heredarían todas las propiedades de sus ancestros.

### 2.1.2. Polimorfismo

Estrechamente relacionado con el concepto de herencia, aparece el de polimorfismo (Álvarez (2014)), el cuál es una propiedad de algunos lenguajes de programación por la que los objetos de una clase pueden comportarse de formas diferentes, en función del contexto, y por tanto responder de maneras diversas antes los mismos mensajes recibidos.

Esto se puede considerar como una relajación del sistema de tipos de un lenguaje, pues permite que ciertos objetos se comporten como la clase de un tipo de objeto determinado, pero también, en circunstancias diferentes, como las clases que derivan de ella. Es decir, si tenemos tres clases: A, B y C, donde B y C derivan (heredan) de A, una referencia a un objeto del tipo A, puede comportarse como esa clase A, pero también, dependiendo de la situación, como un objeto de las clases B o C.

Por lo expuesto anteriormente, se puede concluir que en realidad, el polimorfismo se refiere al comportamiento de un objeto, pero no tanto a su pertenencia a una u otra clase dentro de la jerarquía. Es decir, a la forma en que es capaz de responder ante los mensajes que recibe.

### 2.1.3. Sobrecarga

Gracias precisamente al polimorfismo, emerge también el concepto de sobrecarga. Según se menciona en José Cerrada (2000), la sobrecarga no es más que una de las posibilidades de manifestación que tiene el polimorfismo. Dicha sobrecarga puede darse en dos escenarios diferentes, propiciando la existencia de al menos dos tipos claramente diferenciados:

- Sobrecarga de operadores. Se produce cuando los operadores de un lenguaje adquieren múltiples formas, que les permiten actuar de formas diferentes frente a tipos de datos distintos. Un ejemplo claro lo tenemos en la mayoría de lenguajes, donde los operadores aritméticos (+, -, \* y /, entre otros), están sobrecargados, actuando de forma diferente si los operandos son números enteros, que si son números reales, o matrices. El operador se mantiene, pero su comportamiento varía.
- Sobrecarga de funciones. En este caso, los elementos sobrecargados son las funciones o métodos que contienen las clases. En muchos lenguajes no se permite tener métodos con la misma signatura (compuesta generalmente por el tipo de retorno, el identificador del método, y los nombres de los parámetros y sus tipos), pero si se permite tener una función con el mismo identificador que otra, siempre que tengan distintos argumentos. Es decir, siempre que la cantidad o el tipo de sus parámetros sea diferente. En este caso, se dice que dicha función está sobrecargada, pues en función de los argumentos con los que se invoque, tendrá un comportamiento u otro.

### 2.1.4. Métodos abstractos

En programación, y más concretamente en POO, acotando al lenguaje Java como ejemplo, un método abstracto es aquel que tiene una declaración, pero no una implementación. Es decir, es un método para el que se ha proporcionado la signatura, pero no el cuerpo del método en si.

Típicamente, un método se establece como abstracto, cuando en el momento de su declaración, o bien se desconoce, o bien no se quiere proporcionar la implementación que tendrá. Esto se puede hacer, por ejemplo, en caso de querer delegar la implementación del método en otras clases. De esta forma, si suponemos una jerarquía de clases que utilicen el mecanismo de herencia, podríamos declarar un método como abstracto en la clase padre, obligando a todos sus descendientes, es decir, a todas sus clases hijas, a proveer una implementación.

Por otro lado, conviene señalar que, si una clase contiene un método abstracto (y basta con uno solo), se convierte automáticamente en una clase abstracta, y por tanto en su declaración se debe especificar dicha condición. Por el contrario, también existe la posibilidad de que una clase sea declarada como abstracta, aún sin contener ningún método abstracto en su cuerpo.

### 2.1.5. Reutilización de código

Este es un concepto intrínseco a la programación orientada a objetos, pues su planteamiento propicia una mejor y más sencilla reusabilidad.

Citando a Dibujon (2007), encontramos la siguiente definición:

*“La reutilización de código se refiere al comportamiento y a las técnicas que garantizan que una parte o la totalidad de un programa informático existente se pueda emplear en la construcción de otro programa. De esta forma se aprovecha el trabajo anterior, se economiza tiempo, y se reduce la redundancia.”*

Por tanto, como vemos en dicha descripción, el objetivo que persigue la también conocida como reusabilidad, no es otro que el de poder volver a utilizar aquellos fragmentos de código, componentes, módulos, o subsistemas enteros, extraídos de una aplicación determinada, en nuevas y diferentes aplicaciones, pudiendo hacer uso de la totalidad, o al menos de parte de dicho código, en ocasiones siendo necesarias ligeras modificaciones. Todo con el fin último de poder ahorrar tiempo, costes, o en muchas ocasiones, ambos.

#### 2.1.5.1. Librería

A colación de la idea anterior, que mencionaba la posibilidad de reutilizar fragmentos de código, clases, módulos, e incluso sistema enteros, en nuevas aplicaciones, asoma la necesidad de buscar una forma de organizar o de categorizar todo ese código, con el fin de facilitar su búsqueda, y por tanto, su uso posterior.

Surge así el término librería, también conocida como 'biblioteca' de software. Una librería (K.lee (2002)) no es más que un conjunto de funcionalidades algorítmicas, empaquetadas o agrupadas de algún modo, que ofrecen una interfaz pública bien definida para su uso y aprovechamiento. Es decir, permiten ser utilizadas por cualquier aplicación que las invoque, aunque no tenga relación alguna con ella, ni con otras aplicaciones que también utilicen la misma librería.

En esencia, el comportamiento de un programa normal, y de una librería, no difiere demasiado, pudiendo tener un contenido muy semejante en cuanto a estructura. Si principal diferencia reside en su independencia. Mientras que una aplicación se puede ejecutar de forma autónoma, y cumplir algún propósito específico, una librería no puede hacerlo, puesto que siempre sirve como un medio para otro programa, no siendo un fin en si misma.

Como ejemplos de librerías muy extendidas y de uso popular en las aplicaciones modernas, tenemos las que ofrecen sistemas operativos como Windows, Linux, o macOS, donde se brinda, en forma de estas librerías, una serie de implementaciones de los servicios que expone el sistema, y que pueden disponer las aplicaciones desarrolladas para esos sistemas.

## 2.2. Patrones de diseño

Un patrón de diseño, entendido de forma simple, no es más que un par 'problema/solución', al que se le asigna un nombre, con el fin de identificarlo, y que contiene consejos acerca de cómo aplicarlo en diversas situaciones. Citando a Larman (2003), podemos definir un patrón como:

*“Una descripción de un problema y la solución, a la que se da un nombre, y que se puede aplicar a nuevos contextos; idealmente, proporciona consejos sobre el modo de aplicarlo en varias circunstancias, y considera los puntos fuertes y compromisos.”*

Es decir, no solo proporciona una explicación del problema a resolver, y propone una solución, sino que también reúne una serie de sugerencias acerca de cómo adaptarlo a una nueva circunstancia concreta, así como señala las fortalezas y debilidades reconocidas para dicho patrón.

El término sugiere, intencionalmente, algo repetitivo, debido a que es precisamente en ese punto donde reside su interés. El objetivo de un patrón de diseño no es plantear ideas completamente nuevas, o soluciones innovadoras, sino proporcionar conocimiento, en forma de principios y técnicas, acerca de un problema bien conocido, al que se le ha aplicado una solución demostradamente válida en varias situaciones, por lo que se prevé igualmente útil en casos similares en el futuro.

### 2.2.1. Patrón *Builder*

El patrón *Builder*, conocido en español como patrón Constructor (no confundir con el componente 'constructor' de Java), basa su funcionamiento, tal y como se comenta en Juárez (2011), en permitir la creación de un objeto complejo, a partir del ensamblado de partes individuales más pequeñas, y más simples, las cuáles contribuyen al total. Es por tanto un patrón 'creacional', pues permite resolver un problema asociado a la creación de instancias.

Además, otra de sus ventajas es que aglutina todo el proceso de construcción en un solo punto, por lo que dicho proceso en si mismo, puede tener un comportamiento dinámico, creando variaciones diferentes del mismo objeto, en función de los parámetros que reciba.

Los actores principales en este patrón son 4: el *Producto*, que representa el objeto del cual se va a crear la instancia, el *Builder*, que declara una interfaz para la creación de los 'Productos', el *ConcreteBuilder*, que implementa la interfaz 'Builder', de tal forma que reúne las partes necesarias para la construcción del objeto de una forma determinada, y el *Director*, el cuál utiliza objetos genéricos del tipo 'Builder', que, en realidad, serán alguna de las implementaciones de 'ConcreteBuilder' existentes, para crear el 'Producto' final, es decir, el objeto complejo deseado.

De esta forma, la invocación del proceso de construcción podrá realizarse siempre de la misma forma, sobre un 'Builder' abstracto, mientras será el 'ConcreteBuilder' que lo esté implementando en la práctica, el que determine la manera específica en que ese objeto es construido.

Estrechamente relacionado con el patrón *Builder*, tenemos el patrón *Factory*, también un patrón creacional, y con un comportamiento similar. De hecho, en ocasiones pueden surgir dudas acerca de cuál utilizar, por lo que conviene resaltar sus diferencias fundamentales:

- El patrón *Builder* encaja mejor cuando estamos hablando de la creación siempre del mismo elemento, pero el cuál puede tener características o propiedades que varían de una a otra forma de construirlo. Por ejemplo, si hablamos de crear un objeto que puede presentar diferentes formas, o para el que pueden existir diferentes tipos.  
Por otro lado, el patrón *Factory* es un buen candidato si estamos hablando de una familia de elementos diferentes, pero relacionados. Por ejemplo, cuando hablamos de la construcción de un vehículo, varía completamente la forma de construirlo si es un coche, una moto, o un avión.
- Por lo comentado en el punto anterior, es frecuente que el patrón *Builder* generalmente cree objetos que sigan a su vez el patrón *Composite* (sección 2.2.2), debido a que en última instancia, todos los elementos creados sean del mismo tipo genérico. Esto es algo bastante habitual, pues constantemente se deben mezclar o complementar diversos patrones entre sí para resolver un problema más grande que aquel que resuelve uno de los patrones por sí solo. En el caso del patrón *Factory*, esto no ocurre, pues puede crear objetos completamente diferentes, salvo porque tienen alguna clase de dependencia entre ellos.
- Generalmente, el patrón *Builder* entiende la creación del elemento que maneja, paso a paso, siendo el último de los pasos el devolver el 'Producto' ya construido. Por el contrario, para el patrón *Factory*, la construcción del producto implica el retorno inmediato del mismo al cliente que lo ha invocado, en un solo paso.

### 2.2.2. Patrón *Composite*

El patrón *Composite*, también conocido como patrón Compuesto, posibilita, valga la redundancia, la composición de objetos complejos, a partir de otros más sencillos, utilizando para ello una estrategia de creación recursiva, y resultando en una estructura con forma de árbol.

Todos los objetos que forman parte del objeto compuesto comparten la misma interfaz, y por tanto, pueden ser tratados externamente de la misma manera. Esto habilita jerarquías de objetos tan rebuscadas como se desee, puesto que allí donde se quiera usar el objeto compuesto, podrá emplearse de igual manera cualquiera de los objetos que lo componen, sin necesidad de cambios. Además, permite la inserción sencilla de nuevos componentes en la jerarquía.

Para explicarlo más claramente, supongamos el siguiente esquema, donde tenemos:

- *Componente*. Es la entidad que declara la interfaz común para todos los objetos de la jerarquía, y en consecuencia, implementa el comportamiento por defecto general para todas las clases.

- *Compuesto*. Define la conducta de los objetos compuestos y almacena sus hijos.
- *Hoja*. Se ocupa de especificar la actuación de los objetos primitivos del objeto compuesto.
- *Cliente*. Es el encargado de manipular los objetos compuestos a través de la interfaz que ofrece el 'Componente', así pues, es el sujeto final que hace uso de este patrón.

### 2.2.3. Patrón *Observer*

El patrón *Observer* (TakuyaMurata (2003)), llamado también Observador, sirve para definir una dependencia 'uno-a-muchos' entre diversos objetos. Dicha dependencia provoca que, cuando uno de los objetos (el sujeto) cambia su estado, el resto de objetos dependientes (los observadores) son notificados de ese cambio, y pueden actuar en consecuencia. Para ello, el sujeto no conoce ni necesita conocer la existencia de los observadores, sino que simplemente se limita a notificar la variación de estado que ha sufrido. Se trata de un patrón de 'comportamiento', puesto que se encarga de asignar responsabilidades y funcionalidades a las entidades.

Es un patrón muy oportuno cuando se precisa que un conjunto de clases sea consistente, es decir, coherentes entre ellas y los datos que manejan, sin que estas clases tengan una relación directa, lo que tiene la ventaja de reducir el acoplamiento interno del grupo. El hecho de mantener un nivel de acoplamiento bajo entre las clases, es un propósito que siempre debe perseguir cualquier implementación software, pues, en caso de producirse modificaciones en alguna de ellas en el futuro, la repercusión debería tratarse que fuese siempre mínima. Es decir, ayuda a mejorar la mantenibilidad y reutilización del software generado.

Este patrón tiene un papel fundamental en la arquitectura MVC comentada en el punto 2.3.1, así como en casi todos los frameworks de construcción de interfaces gráficas en general. Normalmente, la idea que se aplica es la de que el usuario, a través de los componentes de la vista, provoca acciones que ocasionan algún tipo de evento, siendo de esta forma el sujeto que notifica su propia condición. Mientras que los observadores, u observador, mejor dicho en este caso, sería el controlador que está a la espera, o escucha, de dichos eventos, para desencadenar alguna respuesta del programa.

### 2.2.4. Patrón *Singleton*

Se trata de un patrón que se podría decir que maneja un único concepto, el de restringir la creación de instancias de una misma clase, limitando las posibilidades a una sola. Es decir, una clase que implemente este patrón únicamente podrá tener 1 instancia de sí misma, como máximo, o ninguna, en el caso de que no se instancie nunca un objeto de ese tipo. De esta forma, lo que se consigue es que una clase solo tenga un punto de acceso, en caso de que así se precise.

La forma de conseguir esto es, en primer lugar, privatizando el acceso de su constructor, de forma que no pueda ser invocado para crear nuevas instancias. Por otro lado, se ofrece un método de clase (estático), el cuál es además público, lo que permite un acceso global, cuyo funcionamiento interno está basado en lo siguiente: si no existe ya una instancia de la clase, la crea, la almacena internamente y la devuelve como retorno; si ya existe una instancia creada anteriormente, la devuelve.

Los campos de aplicación más propicios para este patrón son aquellas situaciones en las que se dispone de un recurso único (por ejemplo, un fichero abierto en modo de acceso exclusivo), y se quiere que el acceso a dicho recurso esté controlado por la clase que implementa el *singleton*.

### 2.2.5. Patrón de *Indirección*

Este es un patrón bastante sencillo, que como muy bien se explica en Larman (2003), basa su mecanismo en delegar tareas o cometidos. La idea es que una responsabilidad sea asignada a un intermediario, el cual intercede entre dos objetos o componentes, de manera que éstos no se acoplan de forma directa. Es decir, se crea un nivel de 'indirección' entre las dos entidades.

El intermediario es el encargado de recibir mensajes desde una de las entidades, o enviar mensajes hacia la otra entidad, de forma bidireccional, de tal forma que crea un canal de comunicación directo entre componentes no acoplados directamente, lo cual es su principal ventaja. Se puede decir que es un patrón que sirve de base a muchos otros que aplican el mismo concepto, como puede ser el patrón *Adaptador*, o incluso el mismo patrón *Observador* explicado anteriormente.

Forma parte de un conjunto de patrones o principios de diseño conocidos como GRASP, los cuales se pueden considerar 'buenas prácticas', cuya aplicación es altamente recomendable en el proceso de diseño software.

### 2.2.6. Patrón de *Reusabilidad*

Aunque no es un patrón en si mismo, se considera como tal, siendo en realidad un conjunto de técnicas y procedimientos propios de los lenguajes orientados a objetos.

Se trata de un concepto más propio de la librería desarrollada, que un planteamiento teórico general, aunque las nociones usadas son en efecto comunes al paradigma de orientación a objetos.

## 2.3. Interfaces Gráficas de Usuario

Las primeras interfaces gráficas de usuario (GUIs) datan de hace aproximadamente 5 décadas, y sus orígenes las sitúan en el llamado 'proyecto de Aumento del Intelecto Humano' (Lynxblack (2015)), de la universidad de Standford, en la década de los 60.

Ya por aquel entonces se desarrolló una primera aproximación, con un cursor de ratón, y varias ventanas que trabajaban con hipertexto. Era la piedra angular del llamado paradigma WIMP(Derksen (2004)), que más tarde se iría perfilando a lo largo de los años, hasta convertirse en el aspecto estándar de los sistemas operativos de escritorio actuales.

Hoy en día, una interfaz gráfica de usuario es la capa de cualquier programa informático, que actúa como mediador entre el usuario y dicho programa. Generalmente utiliza un conjunto de elementos gráficos para presentar la información, y ofrecer las operaciones posibles que se pueden realizar sobre ella. Su principal objetivo es el de ofrecer un entorno visual, amigable y sencillo al usuario, en su proceso de comunicación con la aplicación y el ordenador.

En los últimos años, han ido surgiendo diversas variaciones de estas interfaces gráficas de usuario, entre las que destacan sobre todo las conocidas como NUI(Poleydee (2009)), que se caracterizan por no usar ningún tipo de dispositivo tecnológico como entrada para la interfaz, en contraposición a lo que se venía haciendo con el ratón, teclado, lápiz táctil, etc. En su lugar, se basan en una interacción directa del usuario con las manos, o bien las yemas de los dedos. Incluso también cabe mencionar, por su reciente mejora, a las VUI(Kitsonk (2005)), fundamentadas en el control por voz.

### 2.3.1. Arquitectura MVC

Con un papel vital en la construcción de interfaces gráficas, se debe hablar aquí acerca del patrón MVC (Anonymous (2004)). Considerado tanto un patrón, como un modelo de construcción de aplicaciones, como una arquitectura de desarrollo software, MVC basa su filosofía en la división de todo programa informático en 3 secciones claramente diferenciadas:

- Modelo. Representa los datos que maneja la aplicación, así como las operaciones de consulta o modificación que se realizan sobre ellos. Se encarga de comunicar a la 'Vista' aquella información que se le solicita en cada momento, y las peticiones de operación sobre los datos le llegan por parte del 'Controlador':
- Controlador. Intermediario entre la 'Vista' y el 'Modelo'. Se ocupa de recibir los eventos que llegan por parte del usuario, y realizar las solicitudes pertinentes al modelo.
- Vista. Se encarga de darle una presentación visual al 'Modelo', con una configuración adecuada que facilite la interacción. Por tanto, precisa que dicho 'Modelo' le envíe la información que necesita mostrar al usuario.

El explicado aquí es el que se conoce como MVC *puro*, en su concepción original. Se puede apreciar mejor de forma gráfica en la figura 2.2.



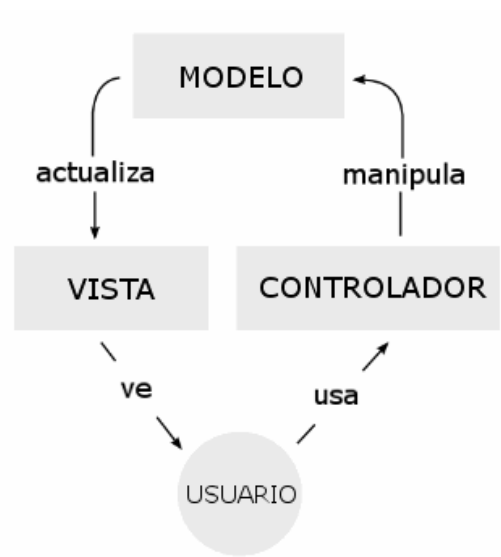


Figura 2.2: Arquitectura MVC pura

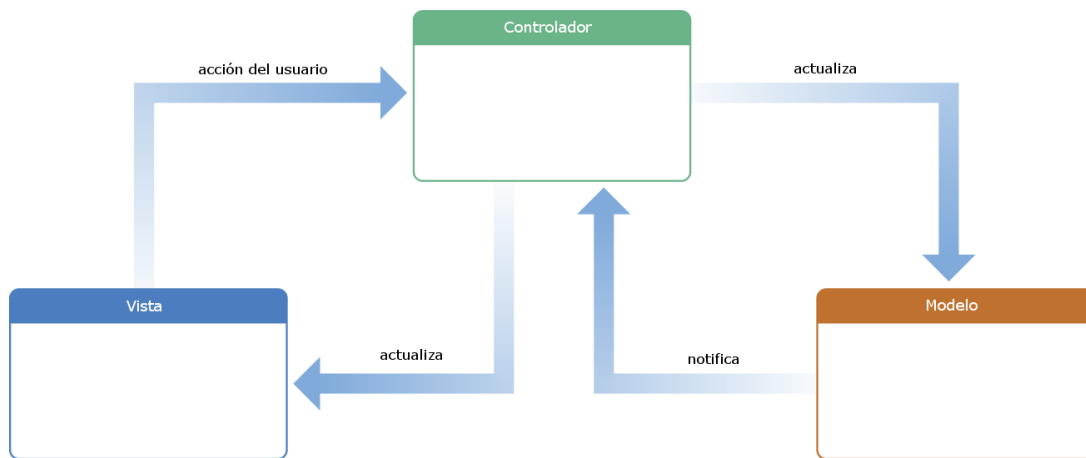


Figura 2.3: Arquitectura MVC Cocoa (Apple)

Sin embargo, existen ligeras modificaciones, algunas incluso más utilizadas que el original, como por ejemplo la presentada por Apple, para su framework de desarrollo de aplicaciones para Mac OS X: Cocoa (Apple (2015)). Dicha variación se muestra en la figura 2.3.

En ella se puede apreciar algún cambio respecto al MVC puro. Por ejemplo, en esta alteración, la 'Vista' no tiene un acceso directo al 'Modelo', sino que es el 'Controlador' el que se encarga de transmitir los datos desde el 'Modelo' a la 'Vista'. Por ello también se le conoce como MIC.

## 2.4. Infraestructura tecnológica

Se reseñarán aquí muy brevemente algunos conceptos relacionados con las tecnologías y herramientas utilizadas durante el desarrollo y progreso del proyecto. Algunas son bien conocidas, mientras que otras son algo más específicas, pero necesarias dentro del proyecto realizado.

### 2.4.1. Java

Se trata de un lenguaje de programación orientado a objetos de propósito general. Es decir, tiene la intención de posibilitar el desarrollo de cualquier tipo de aplicación. Es conocido por el famoso lema WORA, el cuál viene a recalcar la voluntad de sus creadores de que el código desarrollado una vez pueda ser ejecutado en cualquier sitio, de forma independiente de la plataforma, evitando tener que recompilar el programa para cada una de ellas.

Su sintaxis deriva directamente del lenguaje de programación C++, pero a diferencia de este, Java fue definido desde un principio con una clara orientación a objetos. Esto quiere decir que prácticamente todo en Java es un objeto, salvo algunas excepciones bien conocidas, como los tipos primitivos de datos. Además, posee otras características, como son el soporte por defecto para trabajo en red o el soporte para programación concurrente y multihilo.

Es actualmente uno de los lenguajes de programación más conocidos y utilizados a nivel mundial.

### Swing

Es un framework, también llamado 'biblioteca gráfica' que sigue la arquitectura MVC, para Java. Permite el desarrollo de aplicaciones con interfaces gráficas de usuario.

Ofrece un conjunto de componentes ya contruidos, que el usuario puede utilizar, personalizándolos, e incluso extendiendo el conjunto con componentes propios. Además, tiene la ventaja de que es independiente de la plataforma, al igual que Java.

### JavaBeans

Se trata de un modelo de componentes, originalmente desarrollado por Sun Microsystems, mantenido actualmente por Oracle (Oracle (2011)), que permite la construcción de aplicaciones Java.

La especificación de JavaBeans (Oracle (1997)) los define como:

*“Componentes de software reutilizables que se puedan manipular visualmente en una herramienta de construcción.”*

Para funcionar como *beans*, deben cumplir una serie de requisitos, entre los que se encuentran la necesidad de disponer de: constructor sin argumentos, atributos con acceso privado, propiedades

accesibles mediante métodos *get* y *set* públicos, y deben ser 'serializables'. Esto último quiere decir que el objeto puede transformarse en un flujo de bytes, para persistirlo, y más tarde recuperar el objeto a partir de dichos bytes almacenados en disco.

### 2.4.2. NetBeans

Es un entorno de desarrollo integrado, cuyo objetivo original era el lenguaje Java, aunque más tarde se extendió para soportar la compatibilidad con otros lenguajes.

Su ideal se basa en un proyecto de código abierto, con una gran comunidad de usuarios que hacen que siga en constante crecimiento. Fue fundado originalmente por Sun Microsystems, en el año 2000, y actualmente Oracle, compañía que compró y administra Sun Microsystems desde entonces, se ocupa de proporcionar apoyo en su desarrollo.

#### Módulo NBM

Un módulo NBM es típicamente un archivo JAR, es decir, un conjunto de recursos de código y metadatos empaquetados, que contiene cierta metainformación almacenada en su manifiesto, y posee la extensión de archivo NBM (.nbm).

Puede ser usado en cualquier aplicación desarrollada sobre la plataforma NetBeans, permitiendo una fácil instalación y configuración, actuando como un 'plugin' a través el menú de herramientas del IDE, en este caso, NetBeans.

#### *Templates*

Un *template* es esencialmente una plantilla, propia del entorno NetBeans, que posibilita su reutilización por parte del usuario o programador que utilice el IDE. Se pueden crear y utilizar básicamente dos tipos de plantillas:

- Plantillas de código (Oracle (2014a)). Consisten en fragmentos de código reusables y fácilmente invocables o replicables, generalmente a través de atajos de teclado. Un ejemplo es la creación de la sentencia para mostrar una cadena de caracteres por pantalla (*System.out.print*), en Java, a la cuál se puede recurrir escribiendo en un archivo del código fuente simplemente la cadena '*sout*' y pulsando la tecla de tabulación a continuación, de tal forma que se genera la sentencia en el lugar indicado.
- Plantillas de archivo (Oracle (2014b)). Similares al caso anterior, en esta ocasión permiten la creación de archivos nuevos, basados en plantillas, de tal manera que se pueden crear archivos de tipos determinados (por ejemplo, ficheros HTML) que ya contengan un código por defecto, pudiendo partir de ahí para ampliarlo y seguir desarrollando.



## Capítulo 3

# Análisis y diseño del sistema

En este capítulo se realizará un recorrido por las fases de Análisis y Diseño, propias de cualquier proceso de desarrollo que, como el que nos ocupa, aplique técnicas de ingeniería del software.

Comenzaremos hablando sobre dos fases previas al análisis, pero muy importantes en cualquier tipo de proyecto, no solo software, como son la estimación y la calendarización.

En el primer caso, nos centraremos en desarrollar un modelo de estimación de costes empírico conocido como COCOMO, gracias al cuál podremos conocer algunos datos interesantes acerca de los costes temporales o económicos del proyecto. Se realizará una aproximación general a dicho modelo, debido a su enorme versatilidad y la precisión que puede llegar a alcanzar.

En el segundo caso, mostraremos una calendarización global del proyecto, sin entrar en un nivel de detalle muy prolijo. Para ello, utilizaremos un diagrama de Gantt, en el que se mostrará un gráfico bidimensional que entrecruza por un lado, las tareas realizadas a lo largo del proyecto, con los períodos temporales en los que se fueron realizando dicha tareas, además de señalar de una forma especial los hitos alcanzados en el tiempo.

A continuación, iniciaremos la fase de análisis pasando a detallar los requisitos del proyecto, deteniéndonos y explicando punto por punto, pues serán la base sobre la que se estructurará el resto de fases, comenzando por el diseño.

Después de eso, concluimos el análisis especificando los casos de uso del sistema, tanto de forma gráfica con varios diagramas de casos de uso, para lo que emplearemos UML, como textualmente, narrando caso por caso los flujos principales. Se hablará de todos los casos de uso para todos los roles que toman parte y se ven involucrados en el proceso de construcción de una aplicación que use la librería, desde el programador y el diseñador, hasta el propio usuario que utilice el software generado, con el fin de resaltar la separación de los mismos, y diferenciar claramente sus actividades.

Por último, pasaremos a la fase de diseño, la cuál se desarrollará elaborando e interpretando un diagrama de clases, también usando UML. En él, mostraremos de forma minuciosa cada una de las clases del sistema, las relaciones entre ellas, así como sus elementos internos.

## 3.1. Estimación de costes

Cualquier gestión de un buen proyecto software, debe comenzar con un conjunto de actividades que se conocen como planificación del proyecto. Antes de proceder con las fases propias de ingeniería del software, como puedan ser el análisis o el diseño, se debe realizar una estimación del trabajo a realizar y de los recursos que se requerirán a lo largo del desarrollo. Se debe recalcar que dicha estimación nunca será una ciencia exacta, debido a la gran cantidad de variables que entran en juego a lo largo del proceso, todavía más cuánto más se dilate éste en el tiempo.

El modelo presentado a continuación, conocido como COCOMO, utiliza fórmulas empíricas como funciones basadas en LOC/SLOC o PF, para predecir el esfuerzo necesario. El modelo COCOMO es (Nopuomas (2005)):

*“Un modelo matemático de base empírica utilizado para estimación de costos de software. Incluye tres submodelos, cada uno ofrece un nivel de detalle y aproximación, cada vez mayor, a medida que avanza el proceso de desarrollo del software: básico, intermedio y detallado.*

*Este modelo fue desarrollado por Barry W. Boehm a finales de los años 70 y comienzos de los 80, exponiéndolo detalladamente en su libro "Software Engineering Economics" (Prentice-Hall, 1981)."*

Basado en este modelo, se exponen unos resultados resumidos acerca del trabajo realizado, no solo para el caso de la librería de código construida, sino también para el caso del programa de prueba, como parte del proyecto. Se omite el módulo de *templates*, debido al poco impacto que supone.

### 3.1.1. Librería de software

Para el caso de la librería, nos encontramos con los siguientes datos estimados:

- **Lenguaje utilizado** (porcentualmente): Java (100 %).
- **SLOC**: 658
- **Esfuerzo estimado de desarrollo** (persona/mes): 1,55
- **Esfuerzo estimado de desarrollo** (persona/año): 0,13
- **Coste de desarrollo total estimado**: 17.400\$ (~14.600€)

Como se puede apreciar, y más en comparación con el siguiente punto, la librería no ha supuesto un esfuerzo considerable en relación al producto generado. Eso es precisamente porque la mayor parte del trabajo ha recaído en partes menos tangibles de forma directa, como es especialmente el análisis del sistema a construir, y el diseño de la estructura general de componentes para su articulación.

### 3.1.2. Programa de prueba

En relación al programa de prueba desarrollado, obtenemos los siguientes resultados:

- **Lenguaje utilizado** (porcentualmente): Java (100 %).
- **SLOC**: 2.251
- **Esfuerzo estimado de desarrollo** (persona/mes): 5,63
- **Esfuerzo estimado de desarrollo** (persona/año): 0,47
- **Coste de desarrollo total estimado**: 63.300\$ (~53.000€)

En esta ocasión, la estimación arroja un esfuerzo mayor, vista la dimensión final del trabajo producido. Sin embargo, al contrario que en la librería, la mayor parte recayó en la implementación y construcción del programa, siendo las fases de análisis y diseño previas mucho más reducidas.

Si hacemos un cálculo rápido del total estimado del proyecto, podemos apreciar unos números como: el coste económico estimado (67.600€), el número de líneas de código necesarias (2.909), o el esfuerzo necesario en persona/mes y persona/año, respectivamente (7,18/0,6).

## 3.2. Calendarización

Además de la estimación, otro paso importante, previo al comienzo de la partes más directamente relacionadas con la ingeniería de cualquier desarrollo software, es el del cómputo global del tiempo previsto para el proyecto. Esta fase se puede articular a través de un calendario de proyecto que defina las tareas más importantes, repartidas a lo largo de intervalos temporales, con las dependencias existentes entre ellas, así como los acontecimientos más significativos en el proceso, los hitos.

Es lo que se conoce como *calendarización* del proyecto. Se puede organizar alrededor de un cronograma, también conocido como *diagrama de Gantt*. En la figura 3.1 se muestra el cronograma concreto desarrollado para este proyecto, con un nivel de detalle macroscópico, pues pretende proporcionar una visión global de conjunto.

En primer lugar, se puede ver como el proyecto comienza con la lectura y análisis del material inicial que está disponible, etapa que se solapa a su vez, una semana más tarde, con el comienzo de la investigación y adquisición de conocimientos paralela, necesaria para aprender nuevas tecnologías desconocidas, como es el caso de Swing, y los módulos NBM, entre otros. Una vez finalizada la lectura inicial del material, se procede a realizar el primer planteamiento del sistema que se quiere construir, para lo cuál entran en juego las fases de análisis de requisitos, en primera instancia, y del diseño de la estructura y los módulos necesarios, por otro lado.

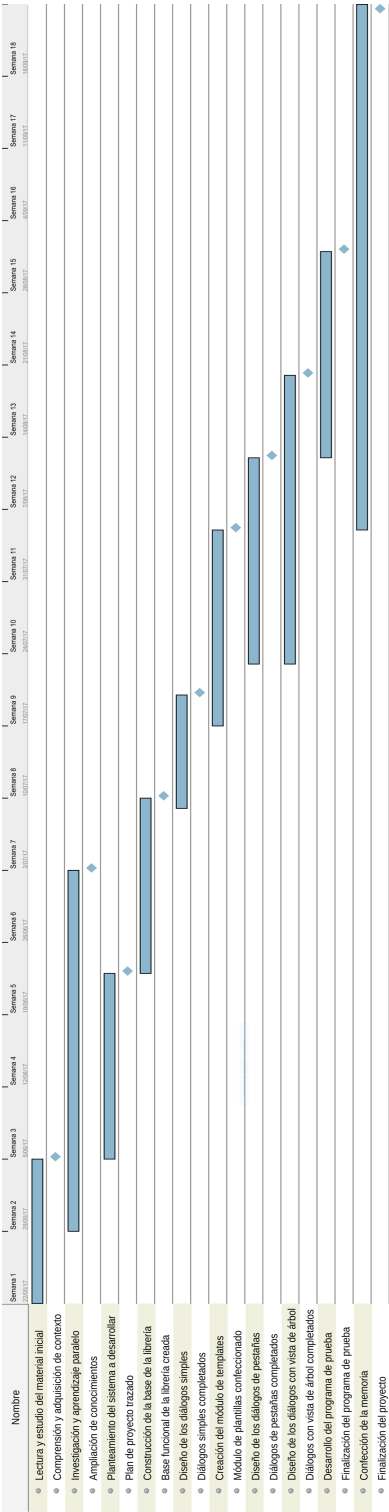


Figura 3.1: Diagrama de Gantt del proyecto



Después de ello, y una vez se ha trazado un plan inicial a seguir, se comienza la construcción de la base estructural y funcional para la librería, donde se empieza a perfilar la factoría de diálogos. En paralelo, como se puede apreciar, continúa durante algunos días la adquisición de conocimiento. Casi a la par que se finaliza la base funcional, se comienza a dar forma a los tipos de diálogos más sencillos, los diálogos de tipo simple, para empezar a tener unidades con las que trabajar.

Acto seguido, y aún con los diálogos simples por completar, se procede a iniciar el subproyecto para el módulo de NetBeans, necesario para desarrollar los *templates* que más tarde serán aprovechables para realizar las pruebas de una forma más cómoda. Una vez se ha hecho un progreso con dicho módulo, y ya se tienen las plantillas para los primeros tipos de diálogos, los simples, arranca la construcción en paralelo de los dos tipos de diálogos restantes, los diálogos de pestañas, así como los de vista de árbol.

Ambas tareas se realizan de forma concurrente, debido a la naturaleza similar de algunas de sus operaciones, como las de inserción de nuevos diálogos hijos, lo que permite aprovechar la experiencia lograda en uno de los tipos, para aplicarla inmediatamente después al otro tipo.

Poco después de este punto, cuando se está acabando de implementar los diálogos de tipo pestañas, más sencillos de elaborar que los de vista de árbol (de ahí que requieran un período de tiempo menor), se inicia la construcción final del programa de pruebas, aunque previamente ya se fueran perfilando algunas de sus partes y detalles en fases anteriores.

Se menciona por último la tarea de confeccionar la memoria, es decir, el documento que recoge todo el contenido y trabajo desarrollado durante el proyecto. Aunque se mencione en este punto, cabe destacar que su producción es un trabajo a largo plazo, que abarca buena parte de la segunda mitad de la longitud total del proyecto, tiempo durante el cuál, si bien no se comenzó a escribir el documento definitivo como tal, si se realizaron una gran cantidad de anotaciones acerca del desarrollo de la librería, y el programa de prueba, que más tarde se incorporaron al documento final.

### 3.3. Requisitos del sistema

Después de la etapa de planificación inicial, que engloba entre otras tareas, como se ha visto, la estimación de costes, y la perspectiva temporal del proyecto, se comienza con la fase de análisis en si misma. Esta se inicia haciendo un repaso íntegro a los requisitos del sistema a desarrollar.

Se debe precisar en este punto, que los requisitos expuestos a continuación han sido extraídos como una copia textual parcial del documento de especificaciones proporcionado junto con el material del proyecto. Se ha decidido hacer así, con el fin de mantener la máxima fidelidad posible a dichos requerimientos, sin cambiar o malinterpretar ninguno de los puntos, debido a una posible transcripción errónea. La reproducción del citado documento comienza, por tanto, en este punto, y forman parte de él todos los elementos enumerados a continuación.

Los **requisitos del sistema** son:

1. Los tipos de GUIs, ya comentados en los ejemplos, serán los siguientes:
  - a) GUIs de una sola página, que denominaremos simples, en las que se integran como máximo dos GUIs.
  - b) GUIs con navegación mediante pestañas.
  - c) GUIs con navegación mediante vista de árbol.
2. Cualquier tipo de GUI se debe poder anidar cualquier otro, aunque ciertas combinaciones no tengan mucho sentido. El nivel de anidamiento de GUIs, salvo para la simple, debe ser teóricamente ilimitado.
3. El proceso debe ser transparente para:
  - a) El usuario. La edición de una GUI debe ser considerada como una transacción atómica, por lo que lo editado en cada una de su páginas debe ser aceptado o rechazado con una sola acción, una sola pulsación de botón del usuario al finalizar la edición completa.
  - b) El diseñador de GUIs. Debe poder tratar estas GUIs como las normales de la librería gráfica que se use, salvo, quizás, algunos pocos métodos añadidos para la reutilización y teniendo en cuenta que si existe un mecanismo previo de extensión de GUIs en la librería gráfica debería ser explícitamente anulado para que no entre en conflicto el mecanismo de reusabilidad aquí expuesto.
  - c) El programador de aplicaciones. Sólo debe tener acceso a la interfaz pública de la jerarquía de GUIs reusables, que, además de la propia de las GUIs normales de la librería gráfica, debe constar de métodos para añadir GUIs, listas de GUIs relacionadas y para la inicialización propia de los GUIs reusables, aparte de la que puedan tener como GUIs normales. Además debe contar un procedimiento sencillo para el ensamblado de basado en métodos polimórficos, de manera que cada clase a editar pueda ensamblar convenientemente su propio GUI reusable, ya sea por invocación de los mismos métodos en las clases de niveles más básicos de su jerarquía y/o por invocación de métodos con igual propósito en los objetos asociados.
4. Se debe mantener en lo posible la estética de las GUIs diseñadas. Esto tiene que ver principalmente con el redimensionamiento de ventanas, ya sea por necesidades propias de la integración de GUIs y/o por acciones del usuario. Esta cuestión se resuelve mediante la distribución (layout) de los componentes gráficos que forman la GUI. Esta distribución no es una mera colocación sino también un procedimiento de recolocación y redimensionamiento, bastante sofisticado,

que poseen los componentes gráficos, especialmente los diseñados para ser contenedores de componentes, en realidad una distribución es un objeto asociado al componente. El requisito estético se concreta en que se debe diseñar el proceso de reutilización de manera que se preserven las distribuciones al integrar GUIs pero sin que se acumulen los márgenes externos de las GUIs al anidar, ya que eso va reduciendo el espacio útil de la ventana.

5. Debe existir un mecanismo para crear GUIs de los tres tipos requeridos en el punto 1, con la estructura mínima: botones para aceptar la edición y rechazarla, contenedor apropiado con distribución y vista de árbol para el último de los tipos requeridos. Estas GUIs las usa el programador de aplicaciones para envolver GUIs sin botones. Es preferible que se diseñen las GUIs sin botones, salvo excepciones, para no tener el problema de qué hacer con ellos al integrar GUIs.
6. Comunicación entre GUIs integradas, hay que arbitrar un sistema de paso de mensajes entre GUIs.
7. Deseamos proteger lo más posible el funcionamiento interno de las clases de la biblioteca, nueva referencia a una interfaz pública reducida, y poder forzar al diseñador de GUIs la implementación de ciertos métodos, imprescindibles para el funcionamiento del proceso a implementar, de manera que su ausencia sea detectada en tiempo de compilación.
8. Se usará polimorfismo en lugar de reflexión, introspección sobre clases, siempre que sea posible. En algunos casos, donde no se pueda hacer de otra forma, se usará la introspección.
9. Se debe poder diseñar las GUIs reusables en un editor gráfico de GUIs, tal y como se diseñan los GUIs usuales. Esto es una parte de la transparencia requerida para el diseñador de GUIs, pero la separamos aparte por que añade complejidad extra al diseño del proceso de reutilización. De hecho el patrón exacto que describiremos como solución del problema de la reusabilidad es posible que no pueda ser implementado para ciertos diseñadores gráficos, o tenga una solución muy compleja, el problema proviene de la última parte del requisito 4. Si el alumno, investigando el funcionamiento del editor gráfico, llega a la conclusión de que no es posible la implementación exacta, debe ofrecer un razonamiento que apoye dicha conclusión e implementar una solución de compromiso.
10. La implementación del proyecto se estructurará en una forma de biblioteca de clases, usando el lenguaje de programación Java y la librería gráfica Swing. La biblioteca se empaquetará en un archivo de tipo jar.
11. Se debe elaborar un programa de prueba, según el siguiente esquema mínimo:

- a) Los campos (variable) en una clase no tendrán acceso público, sino a través de métodos accesorios, este es un requisito general para todo el proyecto.
  - b) Crear una clase, p. ej. A, con al menos cinco campos. Derivar de A dos clases, A1 y A2. A1 añade uno o dos campos más a la clase base. A2 añade al menos cinco.
  - c) Diseñar GUIs reusables para editar todos los campos de las clases A, A1 y A2. Diseñarlas sin botones, para que posteriormente sean envueltas en GUIs reusables con estructura mínima, y darles una distribución adecuada. Usar cierta variedad de componentes gráficos en las GUIs.
  - d) Crear tres clases B, C, D con al menos cinco campos cada una y de manera un objeto de clase A1, otro de clase A2 y otro de clase C estén agregados en uno de clase B, y uno de clase D esté agregado en uno de clase C.
  - e) Diseñar GUIs reusables para editar los campos de las clases B, C y D. El de B debe contener vista de árbol y botones, los otros no, darles una distribución adecuada. Usar cierta variedad de componentes gráficos en las GUIs. Mediante la comunicación de GUIs requerida en el punto 6, el cambio en un determinado componente gráfico de la GUI de D debe provocar un cambio en uno de B, p. ej. al deslizar un JSlider en la GUI de D que cambie un número en un JSpinner en la de B.
  - f) Crear un aplicación con interfaz gráfica en la que se vayan mostrando, y se puedan editar, las GUIs completas de A1, en una sola página (simple), de A2, con pestañas, y de B, con vista de árbol en la que se pueda acceder a todos las GUIs de sus objetos agregados y los agregados a sus agregados. Las GUIs deben ser ensambladas siguiendo el procedimiento a desarrollar comentado en el requisito 3(c). La edición debe ser real, o sea, con actualización de campos en los objetos al aceptar la edición el usuario, previa validación de la coherencia de los datos.
  - g) Las GUIs ensamblados deben tener un buen comportamiento frente a redimensionamientos realizados por el usuario.
12. El proyecto debe estar debidamente documentado usando un sistema estándar de documentación, como pueden ser Javadoc o Doxygen.
13. El software producido debe ser de código abierto, bajo la licencia GPL.
14. Se recomienda al alumno usar la convención de nombres estándar de Java.

Además de estos, en el citado documento se mencionan ciertos patrones de diseño que es conveniente conocer y seguir, puesto que serán de necesaria aplicación en algunos puntos del trabajo. Entre ellos, el documento destaca: *observer*, *builder*, *composite*, *indirección*, y el propio mecanismo de reusabilidad, al que también denomina patrón.

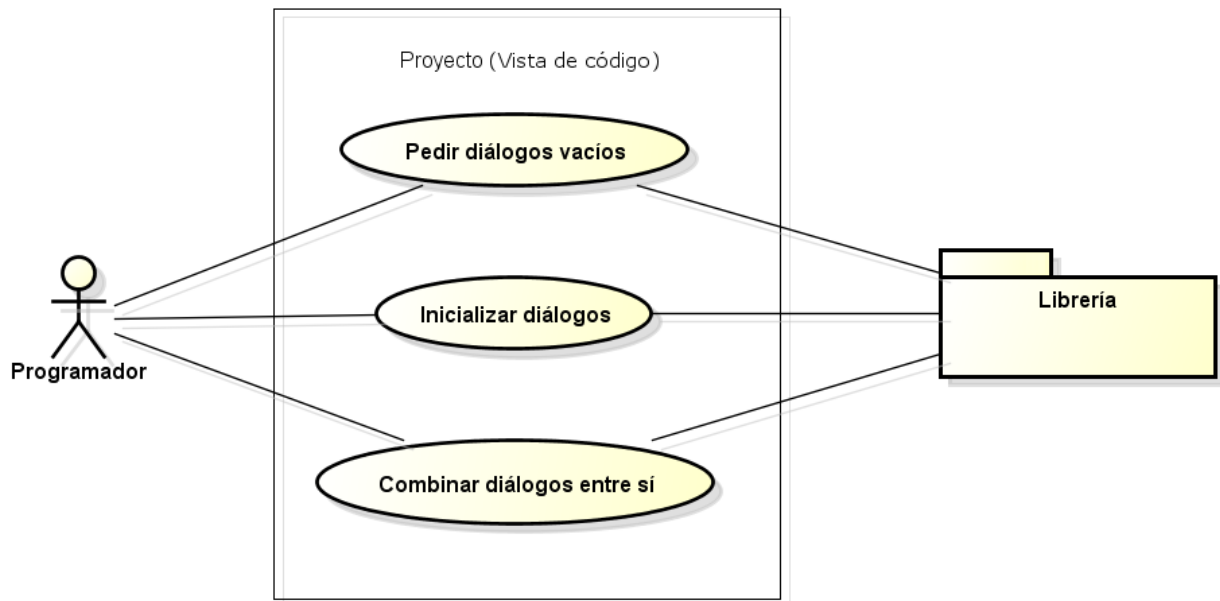


Figura 3.2: Diagrama de casos de uso para el Programador

## 3.4. Casos de uso

De forma posterior a la planificación y al estudio de los requisitos realizados anteriormente, se puede proseguir la etapa de análisis pasando a detallar los casos de uso del sistema.

En conjunto, podemos considerar un total de tres casos de uso del sistema principales, con actores bien definidos, y un cuarto caso necesario para la consistencia e integridad de la aplicación desarrollada, pero sin un actor tan evidente como los otros, por lo que se incluye como parte del tercer caso.

### 3.4.1. Casos de uso del programador

Se muestra el diagrama de casos de uso para el programador en la figura 3.2.

En dicho diagrama se pueden observar 3 casos de uso, que se pasan a detallar a continuación:

#### Caso de uso 1: 'Pedir diálogos vacíos'

- Actor principal: Programador.
- Precondiciones: Se ha creado un proyecto que importe la librería como dependencia, y la utilice.
- Escenario principal (*flujo básico de ejecución*):

1. El caso de uso comienza cuando el Programador desea crear un nuevo diálogo vacío, para lo cuál invoca el método de creación de diálogos de la librería, con los parámetros deseados: tipo, nombre, título, ventana.
  2. La Librería inicia entonces el proceso de creación de diálogo, a través de su factoría, generando una nueva instancia del tipo de diálogo indicado.
  3. La Librería, a través de su factoría, proporciona una implementación trivial para los métodos que más tarde deberá redefinir el Diseñador.
  4. La Librería, a través de su factoría, crea la estructura mínima de componentes para el diálogo (la cuál depende del tipo).
  5. La Librería, a través de su factoría, asigna un tamaño predeterminado al diálogo creado, una vez tiene su estructura.
  6. La Librería, a través de su factoría, envuelve el diálogo construido en una ventana predefinida.
  7. La Librería devuelve el diálogo generado al Programador, y finaliza el caso de uso.
- Extensiones (*flujos alternativos de ejecución*):
    - 2-3a. Se produce algún error durante la creación de la instancia del diálogo, o durante la asignación de la implementación trivial para los métodos:
      - 1. La Librería detiene el proceso de creación.
      - 2. La Librería devuelve un valor nulo como respuesta a la invocación del método, y finaliza el caso de uso.
    - 6a. La Librería recibe como parámetro para la ventana un valor distinto de nulo:
      - 1. La Librería, a través de su factoría, envuelve el diálogo construido con la ventana pasada como parámetro.
      - 2. La Librería devuelve el diálogo generado al Programador, y finaliza el caso de uso.

### **Caso de uso 2: 'Inicializar diálogos'**

- Actor principal: Programador.
- Precondiciones: Se ha instanciado y se tiene el objeto que representa a un diálogo.
- Escenario principal (*flujo básico de ejecución*):
  1. El caso de uso comienza cuando el Programador desea inicializar uno de los diálogos reusables para utilizarlo, para lo cuál invoca su método de inicialización.

2. La Librería, a través del diálogo reusable a inicializar, comienza el proceso de inicialización específico de cada tipo de diálogo.
3. La Librería, una vez se ha realizado la inicialización propia para un tipo de diálogo, realiza la inicialización genérica para todos los tipos.
4. La Librería prepara las propiedades internas del diálogo de forma adecuada, así como inicializa las colecciones que utiliza: la lista de hijos, y la lista de *listeners*.
5. La Librería devuelve una respuesta indicando el éxito de la operación, y finaliza el caso de uso.

■ Extensiones (*flujos alternativos de ejecución*):

- 2-5a. Alguno de los pasos de inicialización falla durante el proceso:
  - 1. La Librería devuelve una respuesta negativa, indicando un fracaso en la ejecución, y finaliza el caso de uso.
- 5b. El diálogo a inicializar es uno construido por el Diseñador, y éste ha proporcionado un método adicional de inicialización para los componentes gráficos:
  - 1. El Programador, una vez ha inicializado la parte reusable del diálogo a través de la Librería, invoca el método de inicialización facilitado por el Diseñador.
  - 2. Se realiza la inicialización de la parte gráfica (propia de Swing) del diálogo, de la forma que haya indicado el Diseñador, y finaliza el caso de uso.

### Caso de uso 3: 'Combinar diálogos entre sí'

- Actor principal: Programador.
- Precondiciones: Se ha generado e inicializado un diálogo.
- Escenario principal (*flujo básico de ejecución*):
  1. El caso de uso comienza cuando el Programador desea combinar dos o más diálogos entre sí, para lo cuál invoca el método apropiado en la Librería, a través del diálogo.
  2. La Librería inicia, a través del diálogo, el proceso de inserción de un diálogo en otro.
  3. La Librería realiza la parte de inserción genérica, que corresponde a la parte reusable del diálogo: asignación de un ancestro y actualización de las colecciones de hijos y de 'escuchadores'.
  4. La Librería pasa a realizar la parte de inserción específica del tipo de diálogo, la cuál consiste en la colocación gráfica de los componentes de forma propicia, así como del redimensionamiento de la jerarquía de diálogos, en caso de ser necesario.

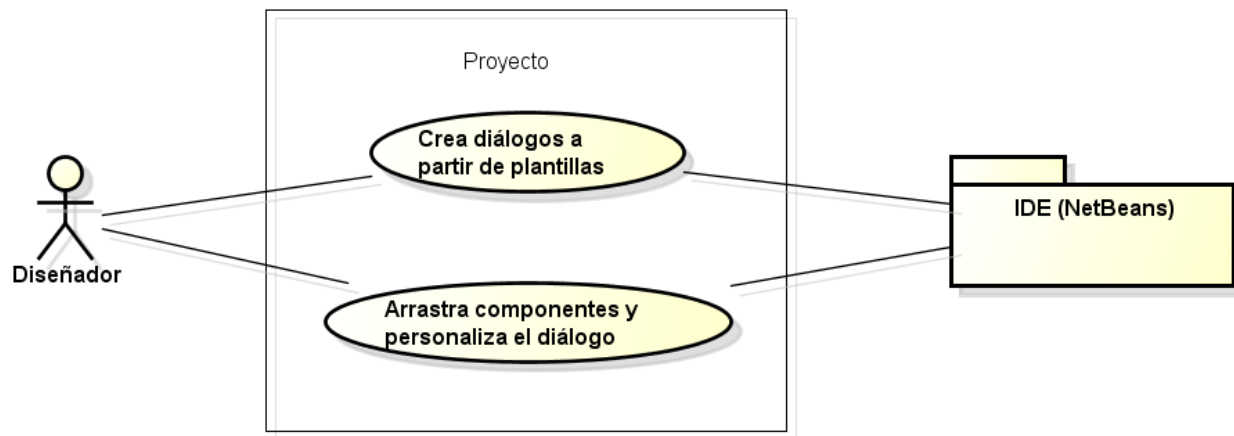


Figura 3.3: Diagrama de casos de uso para el Diseñador

5. La Librería termina el proceso de inserción, y finaliza el caso de uso.

■ Extensiones (*flujos alternativos de ejecución*):

- 2a. La Librería recibe entidades vacías o no válidas como diálogos para insertar:
  - 1a. Los argumentos recibidos están vacíos o no son conflictivos:
    - ◇ 1. La Librería no realiza ninguna inserción, ni modifica la jerarquía de diálogos actual, y finaliza el caso de uso.
  - 1b. Los argumentos recibidos son conflictivos, y provocan algún tipo de fallo concreto detectado por la Librería:
    - ◇ 1. La Librería lanza una excepción indicativa del error producido, y finaliza el caso de uso.
- 2b. El Programador desea insertar una colección de diálogos relacionados, en lugar de una sola entidad:
  - 1. La Librería repite los pasos 2-4 hasta que se han insertado todos los diálogos de la colección, y finaliza el caso de uso.

### 3.4.2. Casos de uso del diseñador

Se muestra el diagrama de casos de uso para el diseñador en la figura 3.3.

En dicho diagrama se pueden observar 2 casos de uso, que se pasan a detallar a continuación:



**Caso de uso 1: 'Crear diálogos a partir de plantillas'**

- Actor principal: Diseñador.
- Precondiciones: Se ha creado un proyecto que importe la librería como dependencia, y la utilice.
- Escenario principal (*flujo básico de ejecución*):
  1. El Diseñador desea crear un nuevo diálogo sobre el que empezar a trabajar, para lo que utiliza las herramientas de creación de nuevos ficheros del IDE, seleccionando el tipo de archivo que desea.
  2. El IDE (NetBeans) abre una ventana para configurar las opciones de creación del nuevo diálogo.
  3. El Diseñador proporciona un nombre para el nuevo diálogo, una ubicación, y termina su parte de la operación.
  4. El IDE, a partir de los datos facilitados, genera dos nuevos ficheros correspondientes al diálogo del tipo seleccionado, para lo cuál utiliza la plantilla disponible. Los nuevos ficheros son el .java y .form propios de Swing, del nuevo diálogo.
  5. El IDE abre el nuevo diálogo en pantalla, desde la vista de Diseño, para que el Diseñador pueda empezar a editarlo, y finaliza el caso de uso.
- Extensiones (*flujos alternativos de ejecución*):
  - 2-3a. El Diseñador decide cancelar la operación:
    - 1. El Diseñador cierra la ventana de configuración de parámetros del nuevo diálogo.
    - 2. El IDE no realiza ningún cambio ni crea ningún nuevo fichero, y finaliza el caso de uso.

**Caso de uso 2: 'Arrastrar componentes y personalizar el diálogo'**

- Actor principal: Diseñador.
- Precondiciones: Se ha creado un nuevo diálogo a partir de una plantilla.
- Escenario principal (*flujo básico de ejecución*):
  1. El Diseñador desea comenzar a editar el diálogo creado, para lo cuál abre la vista de Diseño del diálogo a modificar.

2. El IDE, en la vista de Diseño, le ofrece al Diseñador una serie de componentes, agrupados por categorías, entre los que se encuentran los propios tipos de diálogos reusables de la librería.
3. El Diseñador comienza a personalizar el diálogo principal, arrastrando componentes y añadiéndoselos, dándole una estructura adecuada.
4. El Diseñador termina de construir el cuadro de diálogo, y continúa con el protocolo que le impone la librería: redefine los métodos para validar, guardar y finalizar el diálogo. Una vez lo hace, finaliza el caso de uso

■ Extensiones (*flujos alternativos de ejecución*):

- 1a. El Diseñador elige construir el diálogo a partir de uno vacío, en lugar de utilizar una plantilla:
  - 1. El Diseñador, previo al resto de pasos, le da la estructura adecuada al diálogo, según su tipo.
  - 2. Crear y sitúa los componentes convenientes, asigna el contenedor apropiado para el tipo de diálogo, y asigna sus propiedades de la forma oportuna.
  - 3. El Diseñador puede continuar entonces, una vez tiene el diálogo preparado para ser reusable, a partir del paso 1 del flujo principal.
- 3a. El Diseñador desea elegir los nombres para las etiquetas de las pestañas, o de las entradas del árbol, si está diseñando un diálogo de Pestañas, o de Vista de árbol, respectivamente:
  - 1. El Diseñador confecciona los nombre para las etiquetas de forma adecuada, y continúa con el resto de la personalización.
  - 2. Cuando termina de diseñar el cuadro de diálogo, prosigue con el paso 4 del flujo básico.
- 4a. El Diseñador desea que algún atributo, del diálogo o su modelo, pueda ser modificable desde el exterior:
  - 1. El Diseñador redefine el método de recepción de valores externos de la librería, para adaptarlo a su diálogo, y finaliza el caso de uso.
- 4b. El Diseñador desea que su diálogo emita un valor para posibles observadores:
  - 1. El Diseñador crea o redefine el método correspondiente que realiza tal operación, utilizando un identificador y un valor que conoce previamente y comparte con el Programador.

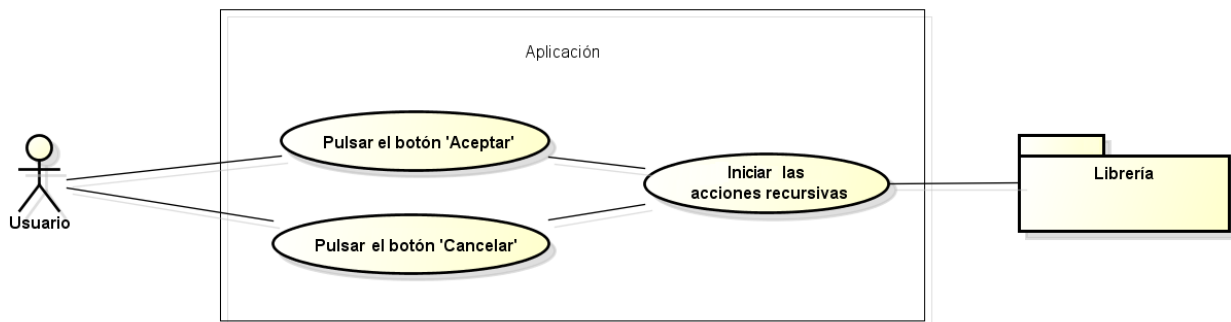


Figura 3.4: Diagrama de casos de uso para el Usuario

### 3.4.3. Casos de uso del usuario

Se muestra en la figura 3.4 el diagrama de casos de uso para el usuario de una aplicación que haya sido desarrollada utilizando la librería.

En dicho diagrama se pueden observar 2 casos de uso, que se pasan a detallar a continuación:

#### Caso de uso 1: 'Pulsar el botón 'Aceptar''

- Actor principal: Usuario.
- Precondiciones: Se ha iniciado la aplicación, y mostrado al Usuario alguno de los diálogos reusables.
- Escenario principal (*flujo básico de ejecución*):
  1. El Usuario desea confirmar los datos introducidos en el diálogo de la aplicación, o en alguno de los diálogos anidados de este, y para ello, pulsa el botón de 'Aceptar'.
  2. La Librería inicia el mecanismo recursivo para atender la petición del Usuario. Comienza validando los datos introducidos en el diálogo, y en toda su jerarquía, de forma recursiva.
  3. La Librería procede después a guardar en el modelo toda la información editada en el diálogo y en aquellos con los que tiene relación de descendencia o ascendencia..
  4. La Librería, por último, realiza las tareas de finalización recursivas en todos los diálogos, y finaliza el caso de uso.
- Extensiones (*flujos alternativos de ejecución*):
  - 2a. La Librería detecta que los datos introducidos no son correctos, y la validación fracasa:
    - 1. La Librería emite un mensaje de advertencia para avisar al Usuario.

- 2. El Usuario lee el mensaje de advertencia y lo cierra.
- 3. La Librería muestra de nuevo el diálogo a editar, dando al Usuario una nueva oportunidad de introducir los datos.
- 4. El Usuario vuelve a introducir los datos modificados, y pulsa de nuevo el botón de 'Aceptar'. El flujo continúa desde el paso 2 del escenario principal.

### Caso de uso 2: 'Pulsar el botón 'Cancelar''

- Actor principal: Usuario.
- Precondiciones: Se ha iniciado la aplicación, y mostrado al Usuario alguno de los diálogos reusables.
- Escenario principal (*flujo básico de ejecución*):
  1. El Usuario desea rechazar las modificaciones realizadas, para lo cuál pulsa el botón de 'Cancelar'.
  2. La Librería inicia el mecanismo recursivo para atender la petición del usuario.
  3. La Librería realiza las tareas de finalización especificadas para los diálogos, recursivamente, y finaliza el caso de uso.

## 3.5. Estructura y organización

Después de las fases de planificación y análisis, conviene comenzar ahora a hablar acerca de la fase de diseño, durante la cuál se perfilará el modelo que simboliza el sistema a construir en las fases posteriores, y para ello nos centraremos en la fase de diseño arquitectónico.

Para mostrar dicho modelo, se elige como forma más usual de representación la de componer un diagrama de clases de diseño, que enfatice la arquitectura global del sistema, con las clases existentes, así como las relaciones efectivas entre ellas.

Dicho diagrama se muestra en la figura 3.5.

A continuación, se procede a explicar el diagrama a nivel macroscópico, con la intención de dar una visión global de conjunto de la librería. Para ello, se enumeran y comentan brevemente los componentes de la misma, ocho en total:

- *IMessagesObserver*. Interfaz que declara el código, en este caso un único método, que puede implementar el diálogo que quiera utilizar el mecanismo de paso de mensajes, para enviar y recibir mensajes externos, desde otros diálogos. Es decir, permite que los diálogos reusables la implementen para habilitar el paso de mensajes en la librería. Se le da una implementación

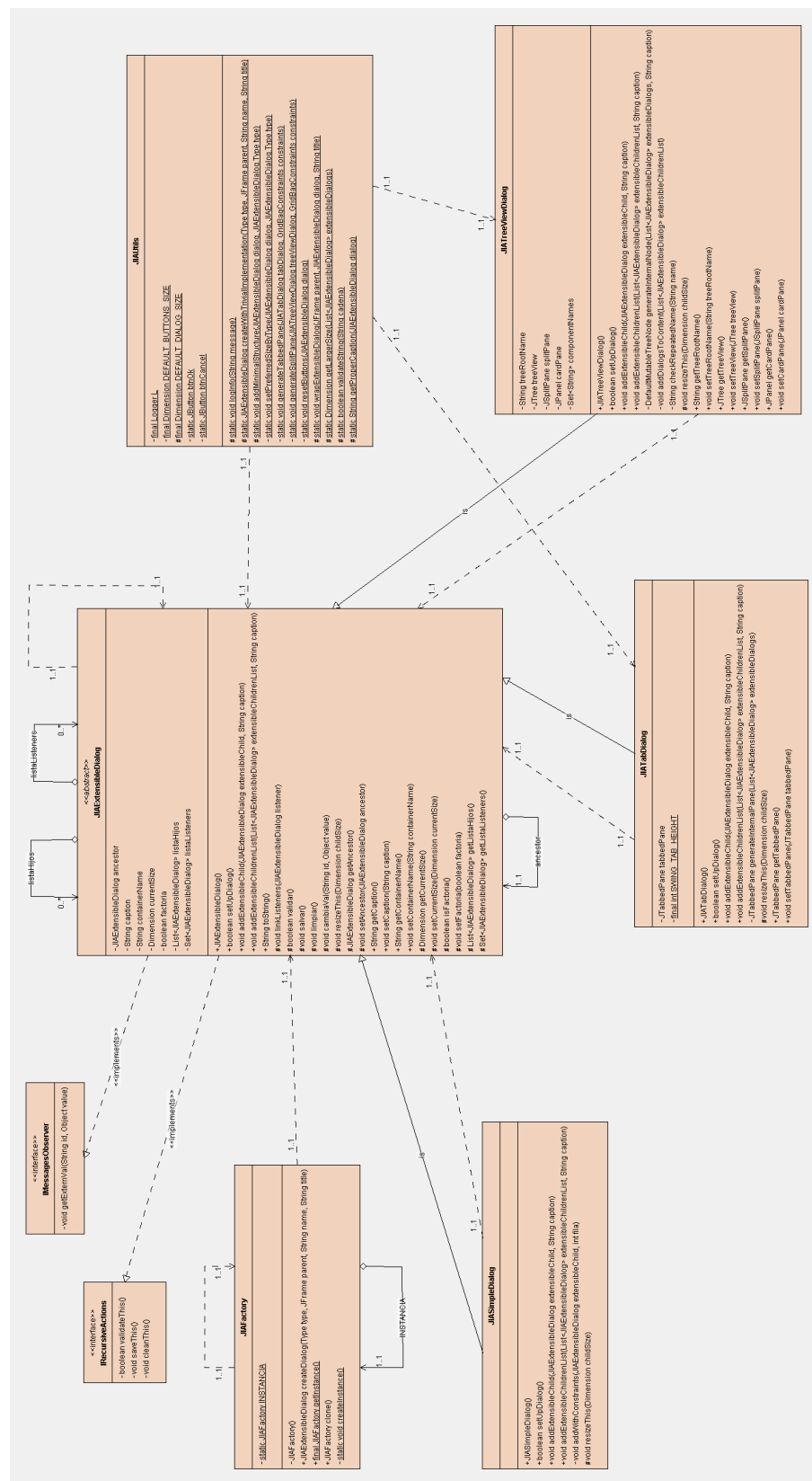


Figura 3.5: Diagrama de clases de la librería

por defecto, la cual lanza una excepción avisando de que se debe redefinir. De esta forma se convierte un error en tiempo de compilación, como la falta de implementación para un método heredado, en un error en tiempo de ejecución, con el lanzamiento de la excepción.

- *IRecursiveActions*. Interfaz que define los métodos recursivos de validación, guardado de datos y finalización, que se ejecutan cuando son pulsados los botones de 'Aceptar' o 'Cancelar' del diálogo principal. De la misma forma que antes, sus métodos tienen una implementación por defecto, la cual lanza una excepción avisando de que se deben redefinir.
- *JIAExtensibleDialog*. Diálogo principal, que hereda de JPanel para conectar con Swing, y sirve como base al resto de tipos de diálogos de la jerarquía de la librería: JIASimpleDialog, JIATabDialog y JIATreeViewDialog. Se crea como clase abstracta para poder delegar la implementación de alguno de sus métodos a las clases hijas de la jerarquía, así como también para evitar que se pueda instanciar de manera directa sin implementar sus métodos genéricos, como por ejemplo 'resizeThis', que cambian según el tipo de diálogo.
- *JIASimpleDialog*. Clase que actúa como diálogo de tipo Simple, y hereda de JIAExtensibleDialog, la raíz de la jerarquía. Puede contener hasta un máximo de 2 diálogos, aunque también admite uno solo. Dichos diálogos tendrán una disposición vertical, lo que quiere decir, que cuando se añadan al diálogo Simple, estarán uno encima del otro.
- *JIATabDialog*. Clase que actúa como diálogo de tipo Pestañas, y hereda de JIAExtensibleDialog, la raíz de la jerarquía. Cada diálogo anidado que contiene se sitúa en una pestaña diferente, existiendo como máximo uno por pestaña, y con un número arbitrario de niveles de anidamiento.
- *JIATreeViewDialog*. Clase que actúa como diálogo de tipo Vista de árbol, y hereda de JIAExtensibleDialog, la raíz de la jerarquía. Cada diálogo anidado que contiene se sitúa en una entrada diferente de la vista de árbol, con la posibilidad de que exista anidamiento de entradas con un número arbitrario de niveles, teniendo un diálogo por entrada.
- *JIAFactory*. Factoría de la librería, que permite generar y obtener diálogos de estructura mínima. Es decir, diálogos completamente vacíos, excepto porque tienen: botones, un contenedor y una vista de árbol de navegación en el caso de JIATreeViewDialog. Se hace uso del patrón Singleton, asegurando una instancia única, y por tanto un solo punto de acceso.  
La sigla JIA viene de 'Java Inteligencia Artificial', en alusión al Departamento para el que se desarrolla el proyecto. Por ello, todas las clases principales de la librería tendrán dicho prefijo.
- *JIAUtils*. Clase utilitaria creada con el fin de reducir la complejidad de la factoría, manteniendo en aquella solo lo esencial. Provee funcionalidades comunes y auxiliares para varios casos, todo de forma estática y con acceso protegido.

# Capítulo 4

## Implementación

Este capítulo se sumerge de lleno en la fase de implementación y construcción del sistema objeto del proyecto: la librería.

Se comenzará señalando algunos detalles generales acerca de la implementación, como toma de contacto inicial, y con el fin de proporcionar algo de contexto transversal a la librería, en lo relativo a esta fase y a cómo se ha ido progresando en su desarrollo.

A continuación, se pasarán a detallar las clases que componen el sistema en su totalidad, a saber:

- *IMessageObserver.*
- *IRecursiveActions.*
- *JIAExtensibleDialog.*
- *JIASimpleDialog.*
- *JIATabDialog.*
- *JIATreeViewDialog.*
- *JIAFactory.*
- *JIAUtils.*

Para cada una de ellas, se realizará una explicación minuciosa de su situación y cometido dentro del entorno global en el que se desenvuelven. Posteriormente, para cada clase se desgranará, uno a uno, cada uno de sus métodos más importantes, haciendo una descripción precisa y cuidadosa.

Por tanto, descontando la sección inicial para hablar sobre los detalles generales de implementación, el capítulo se organizará en torno a las clases que componen la librería, dedicando una sección por cada una de ellas, y una subsección por cada uno de sus métodos más relevantes.

## 4.1. Detalles generales de implementación

### Clase utilitaria de funcionalidades

Gran parte de las funcionalidades comunes y auxiliares empleadas a lo largo de la librería se han refactorizado en una clase utilitaria llamada *JIAUtils*. Esta decisión se ha tomado con el fin de extraer, en cierta manera, la complejidad de alguna de las clases, para que éstas se puedan centrar exclusivamente en su propósito principal, aumentando su cohesión.

Además, se ha diseñado para que todos sus métodos tengan un acceso estático, lo que evita tener que lidiar con instancias múltiples de dicha clase, y con un nivel de visibilidad protegido, lo que permite que su alcance sea global dentro de la librería y esté limitado a ella.

Por último, otra ventaja es la de prevenir la duplicidad de código, pues en lugar de tener código repetido en diferentes clases, se tiene todo centralizado en un único punto, y lo único reiterativo son las invocaciones a dicho código. De esta manera, se aumenta en gran medida la mantenibilidad, puesto que en caso de requerirse modificaciones, estas se focalizan en un solo lugar.

### Empleo del patrón *Singleton*

Se ha decidido diseñar la librería de tal forma que aplique un patrón *Singleton*, explicado en la sección 2.2.4. De esta forma, lo que se consigue es tener una sola instancia, es decir, un solo punto de acceso a la factoría, pues interesa que sea así, debido a que es una entidad única dentro del sistema que queremos construir.

Además, se ha implementado de tal forma que el acceso a dicha instancia esté sincronizado, lo que evita problemas en caso de que la aplicación a desarrollar plantease un escenario con ejecución multihilo. Por otro lado, se ha redefinido el método *clone* heredado de la clase *Object*, con el fin de que no se puedan realizar réplicas de la instancia de la factoría a través de ese medio.

### Uso de un patrón *Builder* no puro

Si atendemos a la explicación del patrón *Builder* realizada en la sección 2.2.1, veremos que en la implementación de la factoría no estamos aplicando el patrón de forma estricta, sino que se ha realizado una simplificación. Se puede comprobar como mantiene la esencia y la razón de ser del patrón, pues permite ofrecer una interfaz común para la creación de los diálogos, sea cuál sea el tipo concreto que queremos generar. Es decir, siempre se crean los diálogos de la misma forma.

Pero, si nos fijamos en el funcionamiento interno, vemos que no estamos separando claramente los procesos de creación para cada tipo en diferentes entidades, como sugiere el patrón. En su lugar, se mantiene toda la lógica en la misma entidad, y se hace una separación menos agresiva, a nivel de métodos privados, que serán invocados condicionalmente acorde al tipo pedido en la llamada.



La justificación de esta elección es básicamente ganar en eficiencia y reducir la complejidad. Debido a la simpleza relativa del proceso en ese punto, y a los pocos tipos diferentes que hay, se ha creído más conveniente el mantener una sola clase que agrupe todos los procesos, y separar éstos a un nivel inferior, mediante funciones diferentes, y no clases.

### Restricciones que imponen los *JavaBeans*

Para tratar este punto, conviene comenzar recordando el requisito 9 enumerado durante la fase de análisis. En él, como se puede ver, se indica la necesidad de que el diseñador pueda utilizar las GUIs reusables de la librería en el editor gráfico del IDE, tal y como se diseñan el resto de GUIs. Esto tiene una implicación directa, y es que para satisfacer ese requerimiento, es menester que los tipos de diálogos reusables aparezcan y puedan ser usados desde la Paleta de NetBeans.

Para poder añadirlos a dicha Paleta, los diálogos deben comportarse como *JavaBeans* (sección 2.4.1). Eso implica algunas limitaciones a la hora de implementarlos:

- Por un lado, si se desea que los componentes internos de los diálogos, como por ejemplo el *TabbedPane* de los diálogos de Pestañas, puedan aparecer como una propiedad del diálogo modificable por el diseñador, sus métodos de acceso y modificación deben ser públicos. Por tanto, así se ha hecho con todos aquellos campos de los diálogos que precisen ser tratados como propiedades del diálogo en la vista de diseño, a la hora de implementarlos.
- Por otro lado, las necesidades de los *beans* tienen una implicación directa sobre las clases que deban comportarse como tal, y es que éstas no pueden ser clases abstractas. Por tanto, la estrategia forzada que se ha seguido es la de usar una única clase abstracta, que es la clase raíz de la jerarquía, *JIAExtensibleDialog*, y que de ella extiendan 3 clases no abstractas, como son las que representan a cada tipo de diálogo.
- Debido a lo anterior, se ha tenido que modificar la naturaleza de los métodos *validateThis*, *saveThis*, *cleanThis*, y *getExternVal*. Deberían ser métodos abstractos, pero en lugar de eso, se les ha tenido que dar una implementación por defecto, la cual lanza una excepción avisando de que se debe redefinir. De esta forma se convierte un error en tiempo de compilación, como la falta de implementación para un método heredado (no admisible en un *bean*), en un error en tiempo de ejecución, con el lanzamiento de la excepción. Esta implementación se ha articulado directamente a través de métodos 'default' en las interfaces que los declaran.

### Colección de 'listeners', en comparación a *EventListenerList*

Es bueno señalar una decisión relativa a la forma de implementación del mecanismo de paso de mensajes. Dicho mecanismo se ha implementado mediante una colección de 'observadores' por cada

diálogo, de tal forma que cada uno tenga un conjunto de referencias a todos los demás.

Una alternativa es la de aprovechar el campo *listenerList*, heredado de la clase *JComponent*, y añadir en él los observadores. Como se señala en Trashgod (2010), la mayor ventaja de tomar esta decisión sería el poder manejar varios tipos de observadores de forma simultánea. Debido a que no existe esa necesidad, pues solo hay un tipo de observador, los diálogos reusables (*JIAExtensibleDialog*), se optó por no adoptar esa medida, manteniendo la colección genérica, la cuál además tiene una implementación más limpia y sencilla.

### Elección de implementación para los métodos de inserción

Se proporciona aquí una justificación para la vía elegida a la hora de implementar los métodos de inserción en los casos de los diálogos de Pestañas y Vista de árbol. Primeramente, repasamos de forma breve lo que hacen dichos métodos, pues su explicación se ampliará en las secciones correspondientes posteriores:

- Caso A: *addExtensibleChild*, adición de GUIs. Se ha determinado que este método añada los diálogos tal cual, como un todo (holísticamente), a otro diálogo, sin tener en cuenta la lista de hijos que estos diálogos a añadir puedan tener. Añade los hijos de uno en uno.
- Caso B: *addExtensibleChildrenList*, adición de lista de GUIs relacionadas. Se ha definido para que permita añadir una lista de diálogos, y, en caso de que alguno de los diálogos de esa lista tenga hijos, estos hijos se añadirán a su vez creando un nuevo nivel inferior de anidamiento, y lo mismo de forma sucesiva para nietos, bis-nietos, y el resto de potenciales descendientes, creando los niveles de anidamiento necesarios, teóricamente de forma ilimitada, cumpliendo así el requisito 2 de las especificaciones.

La forma seleccionada para esta implementación se ha basado en 2 motivos:

- En primer lugar, si el método del caso B hiciese lo mismo que el del caso A, el primero carecería de sentido alguno, pues se podría realizar la misma acción del B con invocaciones sucesivas al A. Esto haría redundante y por tanto innecesario el método *addExtensibleChildrenList*.
- En segundo lugar, tomando esta determinación, se le da mayor flexibilidad al programador a la hora de combinar y mostrar los diálogos: por un lado, con A, no se crean varios niveles de anidamiento, pero se conservan las partes Swing del diálogo definidas por el diseñador. Por otro lado, con B, ignora la parte Swing del diseñador, al centrarse solo en la jerarquía de descendientes, pero a cambio si que permite crear una subordinación de niveles de anidación.

De esta forma quedan cubiertas todas las opciones. Si se desea insertar varios diálogos usando la vía A, se pueden realizar llamadas sucesivas. Por el contrario, si lo que se quiere es añadir un solo

diálogo, pero de la forma indicada en B, se puede pasar como parámetro una lista que contenga un solo elemento, y ya se encargará la librería de generar la jerarquía necesaria.

### Otros detalles a comentar

A lo largo del desarrollo, se ha seguido un estilo de codificación acorde a la convención de nombres de Java (Oracle (1999)), lo que aumenta la legibilidad del código para aquellos familiarizados con otros proyectos desarrollados en dicho lenguaje.

Además, en algunos puntos muy concretos se ha diseñado un manejo de errores basado en excepciones, como el comentado antes para la implementación predeterminada de los métodos en las interfaces, o también, como se comentará más tarde, en los diálogos simples, por ejemplo, cuando se intentan añadir más de 2 hijos al mismo diálogo, momento en el cuál se dispara una excepción advirtiendo de tal condición errónea.

Luego se podrían destacar algunas elecciones bien fundamentadas, para las estructuras de datos usadas en las clases. Como muestra, se mencionará la utilización de una colección de tipo conjunto (*Set*) en el caso de los listeners de cada diálogo. Se ha decidido así, debido a la falta de sentido que tendría tener un mismo observador repetido varias veces en dicha colección, pues se le enviarían varios mensajes de forma innecesaria.

## 4.2. IMessageObserver

Esta clase actúa como una interfaz que implementa la raíz de la jerarquía de diálogos, *JIAExtensibleDialog*, pues de esa manera, todos los diálogos reusables, al extender de ella, heredarán también sus propiedades, implementando esta interfaz. Su función es la de declarar el código que deben redefinir los diálogos que quieran utilizar el mecanismo de paso de mensajes de la librería.

### 4.2.1. Método *getExternVal*

Habilita la recepción de un mensaje desde otro diálogo externo.

Se ha decidido darle una implementación por defecto, la cuál lanza una excepción avisando precisamente de la falta de implementación. Es decir, se convierte un error en tiempo de compilación, como sería la ausencia de implementación para un método, en un error en tiempo de ejecución, con el lanzamiento de la excepción, que avisa de la necesidad de redefinir el método. La redefinición puede ser trivial, es decir, vacía, si el diseñador no tiene intención de usar el mecanismo de paso de mensajes para ese diálogo.

Toma como parámetros un *String*, como identificador del mensaje, y un valor genérico de tipo *Object*, como contenido del mensaje a recibir. No devuelve ningún valor. Al tratarse de la declaración

e implementación trivial del método, no hace nada más que lanzar la excepción comentada. Su redefinición posterior en las clases que lo hereden es la que le dará un sentido de uso.

## 4.3. IRecursiveActions

Interfaz que declara los métodos recursivos de validación, guardado de datos y finalización de los diálogos reusables, que se ejecutan cuando el usuario pulsa los botones de 'Aceptar', o de 'Cancelar' que hay en cada diálogo principal.

De la misma forma que en el caso de la interfaz anterior, *IMessagesObserver*, la única clase que en este caso implementa realmente esta interfaz es el padre de todos diálogos los diálogos reusables, *JIAExtensibleDialog*. Se da así la situación, con la misma intención de antes, de que absolutamente todos los diálogos de la jerarquía hereden de forma automática sus métodos, y deban darles una redefinición adecuada específica.

### 4.3.1. Método *validateThis*

Valida el diálogo que lo implementa, y solo ese, de la forma que lo determine el diseñador.

No recibe ningún parámetro como entrada, pero si que devuelve un valor booleano como retorno. Este valor será verdadero si la validación, para ese diálogo, es exitosa. En caso contrario, devolverá un valor falso. La redefinición puede ser trivial, si así lo considera el diseñador, de tal forma que puede hacer que el método simplemente devuelve verdadero por defecto, no afectando en la validación global de la jerarquía de diálogos, pues siempre superaría su prueba.

### 4.3.2. Método *saveThis*

Guarda los cambios realizados para este diálogo, de forma que se persistan adecuadamente.

No recibe ningún parámetro de entrada, y tampoco devuelve ningún valor como retorno. La redefinición puede ser trivial, si así lo considera el diseñador, de tal forma que podría optar por darle simplemente una implementación vacía, que no hiciera nada. De esta forma, el método siempre sería invocado como parte del mecanismo recursivo que dispara la librería, pero no afectaría en ningún aspecto al flujo de ejecución de la aplicación.

### 4.3.3. Método *cleanThis*

Limpia y se ocupa de las tareas de finalización en el cierre de un diálogo. Se puede invocar tanto como último paso al pulsar 'Aceptar', como directamente el único paso al pulsar 'Cancelar'.

No recibe ningún parámetro de entrada, y tampoco devuelve ningún valor como retorno. La redefinición puede ser trivial, si así lo considera el diseñador, de tal forma que podría optar por darle simplemente una implementación vacía, que no hiciera nada. De esta forma, el método siempre sería invocado como parte del mecanismo recursivo que dispara la librería, pero no afectaría en ningún aspecto al flujo de ejecución de la aplicación.

## 4.4. JIAExtensibleDialog

Esta clase representa el diálogo principal, que hereda de JPanel para conectar con Swing, y sirve como base al resto de tipos de diálogos de la librería: JIASimpleDialog, JIATabDialog y JIATreeViewDialog. Se crea como clase abstracta para poder delegar la implementación de alguno de sus métodos a las clases hijas de la jerarquía, así como también para evitar que se pueda instanciar de manera directa sin implementar sus métodos genéricos, los cuáles cambian según el tipo de diálogo.

Además, engloba las propiedades comunes que debe poseer todo diálogo reusable. Estas propiedades, simplemente enumeradas con una escueta explicación, serían:

- *'ancestor'*, representa el padre de cada diálogo, por lo que es en realidad una referencia de tipo *JIAExtensibleDialog* que apunta a dicho padre.
- *'caption'*, representa el nombre usado como *String*, en los diálogos de pestañas y de árbol, para mostrar en el encabezado de cada pestaña, o como nombre de cada entrada en la vista de árbol, respectivamente. En los diálogos simples no tiene uso.
- *'containerName'*, representa el nombre interno del objeto como *String* que simboliza el diálogo, para poder utilizarlo en caso de querer referenciarlo en algún punto del código.
- *'currentSize'*, representa las dimensiones del diálogo, empleando para ello un objeto de tipo *Dimension*. Es utilizado durante el proceso de redimensionamiento.
- *'factoria'*, se trata de un marcador de tipo *boolean*, utilizado durante el proceso de redimensionamiento, que indica si el diálogo en cuestión ha sido creado por la factoría. Por tanto, solo se le asignará un valor verdadero durante su construcción desde la factoría.
- *'listaHijos'*, colección de hijos que contiene el diálogo. Es un objeto de tipo *List<JIAExtensibleDialog>*, es decir, una lista Java de objetos de tipo diálogo reusable. Esta es la forma que tiene cada diálogo de implementar el patrón *Composite*, explicado en la sección 2.2.2. Es decir, cada diálogo es una entidad compuesta por una colección, una lista en este caso, de elementos de su mismo tipo. De esta forma, se pueden componer estructuras complejas, a la par que permite un tratamiento sencillo de cada diálogo, tratando todos como la misma interfaz de acceso.

- *'listaListeners'*, colección de 'escuchadores' que contiene el diálogo, es decir, de diálogos observadores de los eventos que se produzcan en este. Es un campo de tipo *Set<JIAExtensibleDialog>*.

Seguidamente, se pasan a comentar sus métodos más relevantes, entre los que destacan.

#### 4.4.1. Método *setUpDialog*

Realiza las inicializaciones propias y comunes a todo diálogo reusable, entre las cuales, destacan: inicializar las colecciones de datos para los hijos y los observadores del diálogo, establecer literales adecuados como nombres para los *caption* o *containerName*, y establecer el valor adecuado para el tamaño de la variable *currentSize*, basándose para ello en el valor *preferredSize* que tenga el diálogo.

No recibe ningún argumento de entrada, y devuelve un valor de tipo *boolean* como retorno. Este valor indica el éxito o el fracaso de la operación de inicialización realizada. Para ello, comprueba que todos los campos internos del diálogo no sean nulos, a excepción del campo *ancestor*, para el cuál realiza precisamente la comprobación contraria, pues debe ser inicialmente nulo, al no tener todavía un padre asignado. En caso de cumplirse esas condiciones, devuelve verdadero, pero si falla una sola de ellas, devolverá falso.

#### 4.4.2. Método *addExtensibleChild*

Añade el diálogo pasado como parámetro al diálogo sobre el cuál se invoca. Por un lado, tendrá una parte común a todos los tipos de diálogo, mientras que por el otro, se completará con la parte específica que defina cada tipo de diálogo de forma individual.

Esa parte común consistirá básicamente en 3 pasos separados:

1. Sumar el nuevo diálogo hijo a la lista de hijos del padre.
2. Asociar las colecciones de observadores de la forma adecuada, para lo cuál existe un método concreto explicado posteriormente: *linkListeners*.
3. Y asignar el diálogo actual como padre del diálogo hijo añadido.

El método recibirá 2 parámetros: el diálogo hijo a insertar, de tipo *JIAExtensibleDialog*, y un *'caption'* opcional, de tipo *String*, que no es usado en la parte común descrita aquí, pero si en la parte específica de algunos de los tipos de diálogos. Por otro lado, no devuelve ningún valor como retorno.

#### 4.4.3. Método *addExtensibleChildrenList*

Añade la lista de diálogos relacionados, pasados como parámetro, al diálogo sobre el cuál se invoca. Por un lado, tendrá una parte común a todos los tipos de diálogo, mientras que por el

otro, se completará con la parte específica que defina cada tipo de diálogo de forma individual. Es decir, actúa de la misma forma que el método para un solo diálogo comentado anteriormente, pero aplicando el proceso secuencialmente para cada uno de los diálogos de la lista.

La parte común ya se comentó en el método *addExtensibleChild*, y debido a que lo que hace este segundo método es, básicamente, recorrer la lista de diálogos e invocar a *addExtensibleChild*, se elude repetir la misma explicación aquí.

El método recibe 2 parámetros: la lista de diálogos hijos relacionados que se quieren insertar, de tipo *List<JIAExtensibleDialog>*, y un 'caption' opcional, de tipo *String*, que no es usado en la parte común descrita aquí, pero si en la parte específica de algunos de los tipos de diálogos. Por otro lado, no devuelve ningún valor como retorno.

#### 4.4.4. Método *linkListeners*

Vincula los diferentes observadores entre sí, de cara a construir la estructura para el paso de mensajes. El algoritmo es el siguiente: en primer lugar, se construye una 'lista superior' de listeners, formada por el padre más todos sus listeners, y en segundo lugar se construye una 'lista inferior' de listeners, formada por el hijo más todos sus listeners.

A continuación, se recorre la lista inferior, y para cada uno de sus elementos se añaden todos los diálogos de la lista superior, de forma íntegra. Después, se recorre la lista superior, y para cada uno de sus elementos se repite la operación, añadiendo en este caso cada uno de los diálogos de la lista inferior. De esta manera, quedan todos los diálogos enlazados con todos, formando una estructura equivalente a un grafo completo (en un grafo completo, todos sus nodos tiene una interconexión plena con el resto, existiendo  $n * (n - 1) / 2$  enlaces: Zifra (2006)).

Este método recibe un único parámetro: el diálogo hijo que se está añadiendo en el padre, de tipo *JIAExtensibleDialog*, es decir, el nuevo observador que se va a asociar. Por otro lado, no retorna ningún valor de vuelta.

#### 4.4.5. Método *validar*

Método que utiliza la librería para disparar el proceso de validación recursiva para todos los diálogos de la jerarquía. No recibe ningún argumento, y como retorno devuelve un valor de tipo *boolean* que indica el éxito o fracaso de la validación. Es llamado desde el diálogo principal de la jerarquía, cuando es pulsado el botón de 'Aceptar' de dicho diálogo, lo cual desencadena la invocación de este método, y el inicio del mecanismo de recursividad asociado.

Su funcionamiento es el siguiente: establece al inicio un indicador booleano de validación, inicialmente como verdadero. Después, comienza a recorrer la lista de diálogos hijos del diálogo sobre el cuál se invoca, y por cada uno de ellos, lo que hace es llamarse de forma recursiva, asignando el

resultado de la llamada al valor del indicador comentado anteriormente. Además, para la finalización del bucle que recorre los diálogos establece dos condiciones: primero, que siga habiendo diálogos que recorrer, y segundo, que el indicador de validación siga siendo verdadero.

Una vez finaliza el bucle, lo que hace es devolver el valor booleano basado en las siguiente condición: si el indicador local de validación es verdadero, invoca el método *validateThis* de ese diálogo, y asigna como retorno el resultado de esa llamada. Si el indicador es falso, directamente se devuelve un resultado falso como retorno del método.

De esta manera, el método solo tiene dos flujos de ejecución posibles:

- Por un lado, el indicador de validación puede ser verdadero al final del método, lo que indica que todo ha ido correctamente, y el método devuelve como resultado final 'verdadero'.
- Por otro lado, el indicador, en alguna de las llamadas recursivas, puede asignarse como falso. Por tanto, suceda en el punto en que suceda, este valor se propagará de forma ascendente en la secuencia de llamadas recursivas, rompiendo la ejecución de todos los bucles a su paso (una de las condiciones del bucle es que el indicador se mantenga verdadero). En ese caso, el valor falso llega al nivel superior de llamadas, y el método devuelve como resultado definitivo falso.

Se destaca aquí que el método se ha declarado como *final* por un motivo bien claro, y es que este mecanismo de recursividad es un sistema bien establecido y funcional, cuyo funcionamiento no interesa que pueda ser modificado en diálogos que lo hereden. De esta forma, todos los diálogos pueden aprovechar el mecanismo, evitando que puedan redefinirlo.

#### 4.4.6. Método *salvar*

Dispara el proceso de guardado de cambios realizados para todos los diálogos de la jerarquía. No recibe ningún parámetro, ni tampoco devuelve ningún valor de retorno. Es invocado desde el diálogo principal, en caso de que el proceso de validación recursivo que le precede haya resultado exitoso. En caso de fallar la validación, el guardado no se inicia en ningún caso.

Su funcionamiento es simple: hace un recorrido por la lista de hijos, y por cada uno, se invoca de forma recursiva. Una vez termina de recorrer la lista en una de las llamadas, hace una invocación al método *saveThis()* propio que tiene, guardando los datos para sí mismo, como diálogo. En otras palabras, lo que hace el método es un proceso de recursividad descendente por la jerarquía de diálogos, hasta llegar a las hojas (aquellos diálogos sin hijos), y una vez ahí, se inicia la vuelta recursiva ascendente, guardando los datos para cada uno de los diálogos, en sentido inverso.

De la misma forma que en el caso anterior, se ha declarado como *final* por un motivo bien claro, y es que este mecanismo de recursividad es un sistema bien establecido y funcional, cuyo funcionamiento no interesa que pueda ser modificado en diálogos que lo hereden. De esta forma, todos los diálogos pueden aprovechar el mecanismo, evitando que puedan redefinirlo.



#### 4.4.7. Método *limpiar*

Dispara el proceso de cierre y finalización para todos los diálogos de la jerarquía. No recibe ningún parámetro, ni tampoco devuelve ningún valor de retorno. Es invocado desde el diálogo principal, en caso de que el proceso de validación y guardado de datos se hayan realizado con éxito. En caso de fallar el proceso en algún punto previo, el limpiado y finalización no dan comienzo en ningún caso.

Su funcionamiento es simple: hace un recorrido por la lista de hijos, y por cada uno, se invoca de forma recursiva. Una vez termina de recorrer la lista en una de las llamadas, hace una invocación al método *cleanThis()* propio que tiene, limpiando, cerrando y finalizando lo relativo a sí mismo, como diálogo. En otras palabras, lo que hace el método es un proceso de recursividad descendente por la jerarquía de diálogos, hasta llegar a las hojas (aquellos diálogos sin hijos), y una vez ahí, se inicia la vuelta recursiva ascendente, finalizando cada uno de los diálogos, en sentido inverso.

De la misma forma que en los casos anteriores, se ha declarado como *final* por un motivo bien claro, y es que este mecanismo de recursividad es un sistema bien establecido y funcional, cuyo funcionamiento no interesa que pueda ser modificado en diálogos que lo hereden. De esta forma, todos los diálogos pueden aprovechar el mecanismo, evitando que puedan redefinirlo.

#### 4.4.8. Método *cambiaVal*

Habilita y realiza el envío de un mensaje a otro diálogo externo. No devuelve ningún valor como retorno, pero si recibe dos parámetros de entrada: un identificador, de tipo *String*, referido al mensaje enviado, y un valor genérico, de tipo *Object*, el cuál es el enviado en el mensaje, como contenido.

El algoritmo es simple: recorrer la lista de observadores del diálogo, invocando para cada uno de ellos su método de recepción de mensajes. Es decir, realizando llamadas a los métodos *getExternVal* de cada uno de sus observadores, y pasando como argumentos en esas llamadas, el identificador y el valor que conforman el mensaje que se quiere transmitir.

De esta forma, cualquier diálogo de la jerarquía que quiera enviarle un mensaje a cualquier otro, tan solo tendrá que realizar una llamada interna a este método, en el punto en el que desee, pasándole el identificador y el valor adecuados al destinatario que espera que reciba el mensaje. Ese identificador y el tipo del valor esperado deben ser datos conocidos en la aplicación.

#### 4.4.9. Método *resizeThis*

Calcula el nuevo tamaño para el diálogo padre, si es necesario, en función del tamaño del diálogo hijo que se está insertando. Recibe como argumentos un valor de tipo *Dimension*, que representa el tamaño del hijo que se quiere añadir. No devuelve ningún valor como retorno.

En esta clase, el método es declarado como método abstracto, con el fin de forzar o delegar su

implementación en cada uno de los diálogos disponibles en la jerarquía de diálogos reusables. Así, cada tipo de diálogo redefinirá este método, manteniendo una interfaz común para todos, pero en cada caso adaptándolo a las necesidades y particularidades específicas de ese tipo.

## 4.5. JIASimpleDialog

Clase que representa el diálogo de tipo Simple. Extiende de *JIAExtensibleDialog*, la raíz de la jerarquía, por lo que hereda todos sus métodos y propiedades. No utiliza ningún campo interno, pues le basta con los heredados, por lo que no hay nada que comentar a ese respecto en este punto.

Sus métodos más destacados son los que se presentan a continuación.

### 4.5.1. Método *setUpDialog*

Este método redefine y por tanto sobrescribe al de su clase padre, aunque para este tipo de diálogo, su implementación es trivial. No recibe ningún parámetro de entrada, y devuelve un valor de tipo *boolean* como retorno. Su cometido es simplemente realizar una invocación del método de la clase padre, utilizando para ello la referencia 'super', aludiendo al nivel superior de jerarquía.

Se añade por completud y para mantener la homogeneidad con el resto de clases.

### 4.5.2. Método *addExtensibleChild*

Añade el diálogo hijo al diálogo padre sobre el cuál se invoca. Recibe como entrada dos parámetros: por un lado, el diálogo hijo a insertar, de tipo *JIAExtensibleDialog*, y por el otro, un 'caption', de tipo *String*, que en el caso del diálogo simple carece de utilidad.

Para el caso de *JIASimpleDialog*, en primer lugar se realiza una comprobación de tamaño, para asegurarse de que el diálogo no tenga más de 2 diálogos insertados como máximo, pues es el límite impuesto por la especificación de la librería. Así pues, si el diálogo ya contiene 2 hijos, se lanza una excepción en tiempo de ejecución indicando tal condición de fallo.

A continuación, se añade el diálogo hijo al padre, respetando el layout utilizado (*GridBagLayout*), para lo cuál se emplea un método privado que será explicado posteriormente: *addWithConstraints*. Por último, en caso de ser necesario, se realiza un redimensionamiento adecuado, aumentando el tamaño para el diálogo y ventana principales. Este último paso invoca a su vez otro método para realizar todo el proceso: *resizeThis*.

### 4.5.3. Método *addExtensibleChildrenList*

Añade la lista de diálogos relacionados, pasados como parámetro, al diálogo sobre el cuál se invoca. Recibe como entrada dos parámetros: por un lado, la lista de diálogo hijos relacionados a insertar, de tipo *List<JIAExtensibleDialog>*, y por el otro, un 'caption', de tipo *String*, que en el caso del diálogo simple carece de utilidad. Su funcionamiento se ve limitado a dos acciones.

En primer lugar, realiza una comprobación de espacio, comparando dos variables, el número de hijos que tiene actualmente el diálogo padre, y el número de elementos que tiene la lista de hijos a agregar. Si entre la suma de ambos da una cantidad mayor de 2, quiere decir que el intento de insertar los hijos tendría como consecuencia un diálogo con 3 o más hijos añadidos, algo no permitido por la librería. Por tanto, en ese caso, se lanzaría una excepción indicando la condición de error.

En segundo lugar, debido a la carencia de utilidad real de este método para el caso de los diálogos simples, pues no tiene ninguna característica distintiva o de utilidad ya desde su especificación, su actividad se ciñe a recorrer la lista de diálogos hijos asociados pasada como parámetro, y por cada, uno, invocar el método de inserción de un diálogo individual (*JIAExtensibleDialog*), el cuál realiza el proceso explicado anteriormente.

### 4.5.4. Método *addWithConstraints*

Este método permite añadir gráficamente, como un componente, el diálogo hijo al diálogo padre, durante el proceso de inserción de uno dentro de otro. Además, respeta el 'layout' definido para el diálogo, que en esta ocasión se trata de *GridBagLayout*. Recibe 2 parámetros: el diálogo hijo a añadir, de tipo *JIAExtensibleDialog*, y un entero (*int*) que indica el número de fila en la cuál se debe añadir el diálogo (primera o segunda).

Establece las restricciones adecuadas para el proceso de adición, entre las que cabe destacar: posición del diálogo, en la primera columna y en la fila indicada por el parámetro. Anchura del diálogo de 2 columnas, frente a la anchura para los botones, que es de 1 columna cada uno. Redistribución de peso horizontal y vertical para el diálogo al 100 %, de tal forma que se expanda en ambas direcciones cuando el diálogo padre en el que está contenido crezca también.

Dado que todos los diálogos reusables heredan de *JIAExtensibleDialog* y éste a su vez hereda de *JPanel*, que es en si mismo un contenedor, basta con usar el método '*add()*' propio de *JPanel*. El diálogo añadido se situará en la primera o segunda fila, dependiendo si el diálogo padre ya contenía o no un hijo. El orden siempre será descendente para los nuevos diálogos.

#### 4.5.5. Método *resizeThis*

Calcula el nuevo tamaño para el diálogo padre, si es necesario, en función del tamaño del diálogo hijo. Recibe como parámetro el tamaño del hijo que se quiere añadir, como un objeto de tipo *Dimension*. No devuelve ningún valor como retorno. En cada uno de los hijos, se define como un método *final*, pues no interesa que se sobrescriba a posteriori en ningún caso su implementación, debido a que ésta ya es cerrada y está bien definida para todos los diálogos de un mismo tipo.

Dado que la disposición en los *JIASimpleDialog* es vertical, formando una columna, el tamaño a comparar será el de la anchura del nuevo diálogo hijo a añadir. En caso de que la anchura del diálogo hijo sea mayor que la del padre, la nueva anchura del padre pasará a ser la del diálogo hijo, es decir, la máxima de ambos. Las alturas de los dos, padre e hijo, se sumarán, pues los diálogos hijos se situarán uno encima del otro, verticalmente, por lo que se debe aumentar el espacio disponible para la nueva suma total de sus alturas. En caso de que la anchura del hijo sea menor o igual que la anchura del diálogo padre, no será necesario modificar ésta, siendo solo necesario modificar la altura, sumando ambas, la del diálogo padre y el diálogo hijo.

Además, el método, previo a toda la ejecución, realiza una comprobación para verificar si el diálogo padre en el que se intenta insertar el hijo, es o no un diálogo creado por la factoría. En caso de no serlo, se lanza una excepción indicando tal circunstancia, pues el programador solo tiene permitido insertar otros diálogos dentro de los de tipo simple, cuando estos simples sean los creados por la factoría. No son válidos como padres para este caso los contruidos por el diseñador.

### 4.6. JIATabDialog

Clase que actúa como diálogo de tipo Pestañas, y hereda de *JIAExtensibleDialog*, la raíz de la jerarquía. Cada diálogo que contiene se sitúa en una pestaña diferente, como máximo uno por pestaña, pero sin una restricción del número de pestañas totales, por lo que técnicamente, puede contener un número ilimitado de diálogos. Tiene un par de campos internos propios:

- *'tabbedPane'*, panel de pestañas interno que utiliza el diálogo para colocar los componentes que contiene. En la factoría se crea uno y se asigna al diálogo de tipo Pestañas que se está construyendo. En el caso de que no sea creado por la factoría, se debe inicializar esta propiedad correctamente, siendo el panel que se asigna al *JIATabDialog* el mismo que se le añade como componente Swing a la hora de mostrarlo gráficamente. Es de tipo *JTabbedPane*.
- *'SWING\_TAB\_HEIGHT'*, se trata de una constante, de tipo entero, y con acceso estático, que simboliza la altura aproximada que tiene una pestaña añadida por Swing en el panel interno. Se usa para tenerla en cuenta y sumarla a la altura final del diálogo redimensionado, cuando se hace el proceso de redimensionamiento.

Nos encontramos con seis métodos a destacar.

#### 4.6.1. Método *setUpDialog*

Método que redefine y sobreescribe el heredado por su clase padre. No recibe parámetros de entrada, y devuelve un valor de tipo *boolean* indicando el éxito o fracaso de la inicialización. En este caso, ese valor se calcula en dos pasos: por un lado, realiza una llamada al método de la clase padre, para hacer en primer lugar las inicializaciones comunes, propias de todos los tipos de diálogos, y el valor devuelto es la primera condición. Por otro lado, comprueba que el panel interno del diálogo no sea nulo, porque de serlo, fallará el comportamiento general del diálogo, para añadir más hijos o realizar cualquier otra acción; esta es la segunda condición.

Con ambas condiciones, se elabora una respuesta lógica con el operador *and*, cuyo valor es verdadero si ambas se cumplen, y es falso si una de las dos, o ambas, fallan.

#### 4.6.2. Método *addExtensibleChild*

Añade el diálogo pasado como parámetro al diálogo sobre el cuál se invoca. Por un lado, tendrá una parte común a todos los tipos de diálogo, que será la ya explicada en la clase padre para este método. Recibe como entrada dos parámetros: por un lado, el diálogo hijo a insertar, de tipo *JIAExtensibleDialog*, y por el otro, un 'caption', de tipo *String*.

Para el caso del diálogo de Pestañas, crea una nueva pestaña, como contenedor, por cada diálogo reusable que se vaya a añadir, estableciendo como título de la pestaña el *caption* pasado como parámetro. Además, llama a un método de redimensionamiento, que, en caso de ser necesario, redimensionará los diálogos adecuadamente.

El proceso cuenta al principio con una verificación de nulidad, pues si el diálogo hijo pasado como parámetro es nulo, el método directamente finaliza sin hacer nada. En caso contrario, se pasa a iniciar el curso normal de acontecimientos, realizando una llamada al método de la clase padre, para las tareas comunes a todos los diálogos, ya comentadas con anterioridad.

A continuación, calcula el nombre de la pestaña más adecuado a asignar, utilizando para ello una escala de prioridades, de mayor a menor: el 'caption' pasado como parámetro, el 'caption' que ya pudiera tener el diálogo de antes, y en último lugar, el nombre de la clase del diálogo. Para los 3 elementos realiza una comprobación de que no sean nulos, y además no estén vacíos, en cuyo caso serán candidatos válidos como nombre. Comenzando por el de máxima prioridad, el primero que cumpla las condiciones será el nuevo nombre de la pestaña, y por defecto, en caso de no cumplir los dos primeros casos, se asignará como nombre el nombre de clase.

Después de eso, hace la inserción del componente propia de Swing, utilizando el método que dispone el panel interno para añadir una nueva pestaña, cuyo contenido será el diálogo hijo, y

cuyo nombre será el explicado anteriormente. Por último, realiza una invocación del proceso de redimensionamiento, *resizeThis*, pues en caso de ser necesario, se modificarán los tamaños a lo largo de la jerarquía de diálogos de forma oportuna.

### 4.6.3. Método *addExtensibleChildrenList*

Añade la lista de diálogos pasados como parámetro al diálogo sobre el cuál se invoca. Por un lado, tendrá una parte común a todos los tipos de diálogo, que será la misma que en el método individual (*addExtensibleChild*), pero aplicada para cada uno de los diálogos de la lista. Recibe como entrada dos parámetros: por un lado, la lista de diálogo hijos relacionados a insertar, de tipo *List<JIAExtensibleDialog>*, y por el otro, un 'caption', de tipo *String*.

Para el caso del diálogo de Pestañas, este método permite añadir una lista de diálogos, y en caso de que alguno de ellos tenga hijos, estos hijos serán también añadidos en una pestaña anidada de nivel inferior, sucediendo así de forma reiterativa mientras continúen existiendo hijos que añadir, construyendo una jerarquía.

En primer lugar se hace la parte común para cada diálogo de la lista. A continuación, se crea una nueva pestaña en el diálogo al que se están añadiendo los hijos, y en esa pestaña se añade como componente principal un nuevo *tabbedPane* interno, que contendrá una pestaña anidada por cada diálogo de la lista pasada como parámetro. En caso de que algunos de los hijos tenga a su vez hijos, se llama de forma recursiva al método para crear la estructura interna de ese hijo (con sus hijos) antes de añadirlo de forma efectiva al diálogo padre. Esto se realiza a través de un invocación a otro método: *generateInternalPane*.

Por último, se obtiene un tamaño formado por la altura y anchura máximas de entre todos los diálogos de la lista, y se utiliza para invocar el redimensionamiento que se ocupará de hacer los cambios necesarios en la jerarquía.

### 4.6.4. Método *generateInternalPane*

Método recursivo que, dada una lista de diálogos, los inserta en un panel de pestañas y devuelve dicho panel. En caso de que alguno de los diálogos de la lista tenga hijos, construye la estructura correspondiente, con nuevos niveles de anidamiento, de forma sucesiva a lo largo de la jerarquía.

Recibe como parámetro una lista de diálogos reusables a organizar, de tipo *List<JIAExtensibleDialog>*. Devuelve el panel interno, de tipo *JTabbedPane*, resultado de todo el proceso recursivo. Dicho panel contendrá toda la estructura generada, con los niveles de anidamiento internos correspondientes. Pero al ser un único elemento, se podrá añadir en una sola pestaña como resultado de la invocación del método de inserción.

El método comienza creando un nuevo objeto para el panel interno a devolver, situando las pestañas en la parte superior, y aplicándole una política de *scroll*, lo cual quiere decir, que en caso de que el panel tenga tantas pestañas como para no encajar en pantalla, el desplazamiento hacia las no visibles se realizará mediante una barra de desplazamiento horizontal, a izquierda y derecha.

Continúa haciendo un recorrido por la lista de diálogos pasada como parámetro, y para cada uno, comprueba si tiene hijos:

- Si los tiene, realiza una llamada recursiva pasándole la lista de hijos del diálogo de la lista original en cuestión, para que genere su estructura interna. El resultado de la invocación recursiva, lo añade como una pestaña al panel interno que está construyendo en ese nivel.
- En caso de no tener hijos, simplemente añade el diálogo de forma normal, creando una nueva pestaña para contenerlo.

Finalmente, devuelve el panel interno creado inicialmente, y completado posteriormente, resultado de todas las posibles construcciones internas recursivas que se hayan ido produciendo.

#### 4.6.5. Método *resizeThis*

Calcula el nuevo tamaño para el diálogo padre, si es necesario, en función del tamaño del diálogo hijo. Recibe como parámetro el tamaño del hijo que se quiere añadir, como un objeto de tipo *Dimension*. No devuelve ningún valor como retorno.

El algoritmo se describe a continuación, separando los puntos clave para mayor claridad, así como distinguiendo las posibles ramas que puede tomar debido a condiciones de ejecución.

- Comienza creando una variable *finalSize* para almacenar el tamaño final del diálogo. Inicialmente le asigna como valor el tamaño del diálogo padre, siendo la anchura la del padre, y para la altura, realiza una comprobación:
  - Si el diálogo padre es uno de la factoría, a su altura habrá que restarle el tamaño de los botones de edición, que debe ser tenido en cuenta. Así, la altura resultante de la resta será la que se asigne a la variable *finalSize* inicialmente.
  - En caso contrario, si no es de factoría, la altura asignada será exactamente la misma que la del padre, sin modificarla.
- Después, modifica esa variable *finalSize* en dos vías: por un lado, comparando la altura final asignada inicialmente de la forma comentada antes, y la altura del hijo que se pretende añadir, y asignando el mayor valor entre ambos como nueva altura para *finalSize*. Hace lo propio con la anchura, asignando el valor máximo entre la anchura del diálogo hijo, y la anchura asignada inicialmente a la variable *finalSize*.

- Seguidamente evalúa si el diálogo padre en el que se está insertando el hijo tiene o no un ancestro. Es decir, si se ha alcanzado ya el diálogo superior de esa jerarquía. Esto abre los dos caminos de ejecución principales del proceso:
  - Por un lado, si el diálogo padre tiene algún antecesor, es decir, si el valor de ese campo en su caso no es nulo, hace lo siguiente.

Comprueba si el hijo ha modificado, aumentándolo, el tamaño final inicialmente creado (*finalSize*). En caso de ser así, asigna el nuevo tamaño al diálogo padre, y realiza una llamada recursiva a su ancestro, pasándole como tamaño en el parámetro el nuevo tamaño del padre. En caso de no ser así, esa rama de ejecución finaliza.
  - Por otro lado, si se ha llegado al diálogo de nivel superior, se comprueba, igual que antes, si el diálogo hijo insertado ha provocado que sea necesario aumentar el tamaño del padre. En caso de no ser así, la ejecución del método termina. En caso contrario, se pasa a comprobar otra condición:
    - Si el diálogo superior no es un diálogo principal, es decir, no ha sido creado por la factoría, es porque se trata de algún diálogo intermedio creado por el diseñador, que más tarde se agregará a alguna jerarquía de diálogos. Es decir, será un futuro diálogo intermedio. En ese caso, se le asigna el tamaño final calculado anteriormente, y termina.
    - Si el diálogo superior es un diálogo principal, de la factoría, entonces se asigna el tamaño final calculado a ese diálogo principal, pero además, se asigna un nuevo tamaño mínimo para la envoltura del diálogo, es decir, para la ventana de la aplicación. Ese tamaño mínimo asignado será el mismo que el tamaño final calculado, asignado a su vez al diálogo principal.

## 4.7. JIATreeViewDialog

Clase que actúa como diálogo de tipo Vista de árbol, y hereda de `JIAExtensibleDialog`, raíz de la jerarquía. Cada una de las referencias a los diálogos que contiene se sitúa en una entrada diferente de la vista de árbol, con la posibilidad de que exista anidamiento de entradas con un número arbitrario de niveles. Permite como máximo un diálogo por entrada. Los diálogos en si mismos, estarán ubicados en el panel de contenido con distribución de 'tarjetas'.

Alberga 5 campos internos específicos para su tipo:

- `'treeRootName'`, de tipo `String`. Nombre de la raíz de la vista de árbol situada en la parte izquierda del diálogo.



- *'treeView'*, de tipo *JTree*. Vista de árbol del diálogo, la cual contiene la colección de elementos del diálogo dispuestos jerárquicamente.
- *'splitPane'*, de tipo *JSplitPane*. Panel divisor del diálogo, que cuenta con dos componentes: a la izquierda contendrá la vista de árbol, y a la derecha el panel apilado.
- *'cardPane'*, de tipo *JPanel*. Panel de contenido del diálogo. Contiene una colección con todos los diálogos hijos del diálogo principal, y permite mostrarlos de forma selectiva e individual, acorde al nodo seleccionado en la vista de árbol. Tiene establecido un *'layout'* de tipo *Card-Layout* con el fin de facilitar este comportamiento.
- *'componentNames'*, de tipo *Set<String>*. Conjunto de nombres de los componentes añadidos al panel de contenido del diálogo, guardados con el fin de gestionar de forma sencilla las referencias a todos los diálogos añadidos.

Pasamos a comentar los 7 métodos principales que contiene.

#### 4.7.1. Método *setUpDialog*

Método que redefine y sobreescribe el heredado por su clase padre. No recibe parámetros de entrada, y devuelve un valor de tipo boolean indicando el éxito o fracaso de la inicialización.

En este caso, ese valor se calcula en dos 4 pasos: por un lado, realiza una llamada al método de la clase padre, para hacer en primer lugar las inicializaciones comunes, propias de todos los tipos de diálogos, y el valor devuelto es la primera condición. Por otro lado, comprueba que el panel divisor interno del diálogo no sea nulo, porque de serlo, fallará el comportamiento general del diálogo, para añadir más hijos o realizar cualquier otra acción; esta es la segunda condición. En tercer lugar, comprueba la nulidad de la vista de árbol (*treeView*), y por último, hace lo propio con el panel de contenido, el panel de 'tarjetas' (*cardPane*).

Con esas cuatro condiciones, se elabora una respuesta lógica con el operador *and*, combinándolas, cuyo valor es verdadero si todas se cumplen, y es falso si alguna de las cuatro, o todas, fallan.

Además de eso, un paso previo que realiza al retorno del valor lógico, es el de asignar el nombre para la raíz de la vista de árbol. Para ello, en primer lugar comprueba el campo interno *treeRootName*, y en caso de que sea válido (es decir, no nulo y no vacío), se lo asigna. Sino, utiliza por defecto el nombre de la clase del diálogo.

#### 4.7.2. Método *addExtensibleChild*

Añade el diálogo pasado como parámetro al diálogo sobre el cuál se invoca. Por un lado, tendrá una parte común a todos los tipos de diálogo, que será la ya explicada en la clase padre para

este método. Recibe como entrada dos parámetros: por un lado, el diálogo hijo a insertar, de tipo *JIAExtensibleDialog*, y por el otro, un 'caption', de tipo *String*.

Para el caso del diálogo con Vista de árbol, crea una nueva entrada en el árbol (un nuevo nodo) para el diálogo que se va a añadir, estableciendo como nombre de la entrada el *caption* pasado como parámetro, o en su defecto, obteniendo el *caption* que pueda tener el propio diálogo, y en último caso, el nombre de clase del diálogo, por defecto. También añade el diálogo, junto con el mismo *caption* utilizado para la entrada en el árbol, al panel de contenido de la derecha.

Por último, realiza una invocación del proceso de redimensionamiento, *resizeThis*, pues en caso de ser necesario, se modificarán los tamaños a lo largo de la jerarquía de diálogos de forma oportuna.

El proceso cuenta al principio con una verificación de nulidad, pues si el diálogo hijo pasado como parámetro es nulo, el método directamente finaliza sin hacer nada. En caso contrario, se pasa a iniciar el curso normal de acontecimientos, realizando una llamada al método de la clase padre, para las tareas comunes a todos los diálogos, ya comentadas con anterioridad.

A continuación, calcula el nombre más adecuado a asignar para la entrada del árbol, utilizando para ello una escala de prioridades, de mayor a menor: el 'caption' pasado como parámetro, el 'caption' que ya pudiera tener el diálogo de antes, y en último lugar, el nombre de la clase del diálogo. Para los 3 elementos realiza una comprobación de que no sean nulos, y además no estén vacíos, en cuyo caso serán candidatos válidos como nombre. Comenzando por el de máxima prioridad, el primero que cumpla las condiciones será el nuevo nombre de la entrada de la vista de árbol, y por defecto, en caso de no cumplir los dos primeros casos, se asignará como nombre el nombre de clase.

Después de eso, hace la inserción del componente propia de Swing, pero en esta ocasión en dos localizaciones diferentes:

- Por un lado, añade la entrada en la vista de árbol, insertando simplemente un nodo, cuyo nombre será el explicado anteriormente.
- Por otro lado, añade el diálogo como objeto al panel de contenido, con una inserción estándar de *JPanel*, pasando como parámetros el diálogo, y el nombre referido antes, que debe ser exactamente el mismo que se ha utilizado al añadir el nombre al árbol.

### 4.7.3. Método *addExtensibleChildrenList*

Añade la lista de diálogos pasados como parámetro al diálogo sobre el cuál se invoca. Por un lado, tendrá una parte común a todos los tipos de diálogo, que será la misma que en el método individual (*addExtensibleChild*), pero aplicada para cada uno de los diálogos de la lista. Recibe como entrada dos parámetros: por un lado, la lista de diálogo hijos relacionados a insertar, de tipo *List<JIAExtensibleDialog>*, y por el otro, un 'caption', de tipo *String*.

Para el caso del diálogo con Vista de árbol, este método permite añadir una lista de diálogos, y en caso de que alguno de ellos tenga hijos, estos hijos serán también añadidos en una entrada del árbol anidada, de nivel inferior, sucediendo así de forma reiterativa mientras continúen existiendo hijos que añadir, construyendo una jerarquía.

En primer lugar se hace la parte común para cada diálogo de la lista. A continuación, crea una nueva entrada (nodo) en el árbol, el cuál es "rellenado" con la lista de diálogos hijos pasada como parámetro. Para ello, se invoca un método auxiliar: *generateInternalNode*. Una vez generado el nodo correctamente, se añade al nodo raíz de la vista de árbol. Después de eso, se añade la lista de diálogos, yendo uno por uno, al panel de contenido de la parte derecha del diálogo. Para esta parte final, se invoca otro método auxiliar privado: *addDialogsToContent*.

Por último, se obtiene un tamaño formado por la altura y anchura máximas de entre todos los diálogos de la lista, y se utiliza para invocar el redimensionamiento que se ocupará de hacer los cambios necesarios en la jerarquía: *resizeThis*.

#### 4.7.4. Método *generateInternalNode*

Método recursivo que se encarga de construir la estructura interna del nodo que actúa como nueva entrada en el árbol. Para ello, utiliza la lista de hijos, y las subsecuentes listas de dichos hijos, en caso de que alguno de ellos tenga, para construir recursivamente la jerarquía de nodos, teniendo como nodo raíz el que será añadido como nueva entrada en el árbol.

Recibe como parámetros: por un lado, la lista de diálogos hijos a partir de la cuál formar la estructura, de tipo *List<JIAExtensibleDialog>*, y por otro lado, el 'caption' que se aplicará como nombre para la entrada del nuevo nodo que engloba la lista de esa invocación. Devuelve como retorno el nodo del árbol construido, con su estructura interna perfectamente definida, debido a la construcción recursiva realizada.

El proceso empieza creando un nuevo nodo raíz para esa iteración: *rootNode*, al que se le asigna el 'caption' del parámetros como elemento. A continuación, se comienza a recorrer la lista de diálogos, y por cada uno, se hace una evaluación condicional:

- Si el diálogo no tiene hijos, se añade tal cuál al *rootNode*.
- Si el diálogo tiene al menos un hijo, se realiza una llamada recursiva sobre el mismo método, pasando como parámetros: su lista de hijos, y el 'caption' más adecuado en ese caso, por orden de prioridad (un 'caption' que ya tuviera asignado el diálogo, recuperado mediante el método *getCaption*, o bien el nombre de clase del diálogo). El resultado de esa llamada recursiva se añade el nodo raíz creado al inicio: *rootNode*.

El método finaliza devolviendo el nodo creado (*rootNode*), lo cual, tras sucesivas llamadas recursivas, acabará construyendo la totalidad de la estructura jerárquica a devolver para la llamada inicial.

#### 4.7.5. Método *addDialogsToContent*

Se encarga de añadir la lista de diálogos al panel de contenido del árbol. Recibe como parámetro la lista de diálogos a añadir, de tipo *List<JIAExtensibleDialog>*. No devuelve ningún valor.

Comienza haciendo un recorrido por la lista de diálogos hijos a añadir, y por cada uno:

- Si tiene hijos, hace una llamada recursiva con la lista de sus hijos.
- Si no tiene hijos, calcula el nombre adecuado para el diálogo, y lo añade al panel de tarjetas, junto con el nombre evaluado anteriormente.

Es importante que el nombre con el que se añade al panel de contenido, sea exactamente el mismo que el usado para añadirlo a la vista de árbol, pues dicho nombre actúa como identificador único. Debido a esto último, entra en escena el siguiente método, que precisamente se encarga de asegurar dicha unicidad.

#### 4.7.6. Método *checkRepeatedName*

Método recursivo que comprueba si el nombre con el que se va a añadir el diálogo está repetido, y, en tal caso, lo modifica con el fin de que no lo esté, básicamente añadiendo un asterisco (\*) al final. De esta forma, con varios diálogos repetidos, tendríamos: *nombre*, *nombre\**, *nombre\*\**, y así sucesivamente. Recibe como parámetro el nombre a comprobar, de tipo *String*, y devuelve como retorno el nombre adecuado con el que añadir el diálogo, que puede ser o no el mismo que el del parámetro de entrada.

La única acción que realiza este método es la de comprobar si el nombre proporcionado ya existe en la jerarquía del diálogo, mirando si está contenido dentro de la colección *componentNames*:

- En caso de no existir, devuelve el parámetro directamente como respuesta.
- En caso de estar repetido, hace una llamada recursiva, añadiendo al parámetro el carácter '\*', y recomprobando en dicha invocación su validez. Así, hasta que se encuentre un identificador no repetido.

#### 4.7.7. Método *resizeThis*

Calcula el nuevo tamaño para el diálogo padre, si es necesario, en función del tamaño del diálogo hijo. Recibe como parámetro el tamaño del hijo que se quiere añadir, como un objeto de tipo *Dimension*. No devuelve ningún valor como retorno.

El algoritmo se describe a continuación, separando los puntos clave para mayor claridad, así como distinguiendo las posibles ramas que puede tomar debido a condiciones de ejecución.

- Comienza creando una variable *finalSize* para almacenar el tamaño final del diálogo. Inicialmente le asigna como valor el tamaño del diálogo padre, siendo la altura la resultante de la siguiente comprobación:
  - Si el diálogo padre es uno de la factoría, a su altura habrá que restarle el tamaño de los botones de edición, que debe ser tenido en cuenta. Así, la altura resultante de la resta será la que se asigne a la variable *finalSize* inicialmente
  - En caso contrario, si no es de factoría, la altura asignada será exactamente la misma que la del padre, sin modificarla.
- Y siendo la anchura la resultante de la resta:  
$$anchuraOriginalPadre - anchuraVistaArbol - anchuraSeparador$$

Es decir, la anchura a comparar con la del hijo, no será la del diálogo de vista de árbol completo, sino solo la de su panel de contenido, pues es donde se ubicará el hijo.

Con estas dos medidas se conformará el tamaño de *finalSize*.
- Después, se pasa a comparar el *finalSize* calculado anteriormente con el tamaño del diálogo hijo que se pretende añadir. En cualquier caso, si el tamaño del hijo es igual o menor, es decir, cabe dentro del panel de contenido del diálogo padre, el algoritmo termina.
- Si, por el contrario, el hijo tiene una altura o anchura mayores, se pasa a evaluar otra condición:
  - Por un lado, si el diálogo padre tiene algún antecesor, es decir, si el valor de ese campo en su caso no es nulo, hace lo siguiente. Asigna el nuevo tamaño (*finalSize*) al diálogo padre, y realiza una llamada recursiva a su ancestro, pasándole como tamaño en el parámetro el nuevo tamaño del padre.
  - Por otro lado, si se ha llegado al diálogo de nivel superior, se pasa a comprobar otra condición:
    - Si el diálogo superior no es un diálogo principal, es decir, no ha sido creado por la factoría, es porque se trata de algún diálogo intermedio creado por el diseñador, que más tarde se agregará a alguna jerarquía de diálogos. Es decir, será un futuro diálogo intermedio. En ese caso, se le asigna el tamaño final calculado anteriormente, y termina.
    - Si el diálogo superior es un diálogo principal, de la factoría, entonces se asigna el tamaño final calculado a ese diálogo principal (*finalSize*), pero además, se asigna un nuevo tamaño mínimo para la envoltura del diálogo, es decir, para la ventana de la aplicación. Ese tamaño mínimo asignado será el mismo que el tamaño final calculado, asignado a su vez al diálogo principal.

## 4.8. JIAFactory

Factoría de la librería. Permite crear y obtener diálogos de estructura mínima. Es decir, botones, un contenedor y una vista de árbol para navegación en el caso de *JIATreeViewDialog*. La sigla JIA viene simplemente de 'Java Inteligencia Artificial', en alusión al departamento. Por ello, todas las clases principales de la librería tendrán dicho prefijo. Se hace uso del patrón *Singleton* para la factoría, explicando en la sección 2.2.4, asegurando así una instancia única.

El único campo interno que contiene es '*INSTANCIA*', necesario para la implementación del *Singleton*. Seguidamente se exponen sus métodos más relevantes.

### 4.8.1. Método *createDialog*

Crea y devuelve un diálogo del tipo indicado. Recibe 4 parámetros:

- '*type*', el tipo de diálogo que se quiere crear.
- '*parent*', la ventana principal que envuelve el diálogo principal creado.
- '*name*', el nombre del objeto interno del diálogo que se le quiera asignar, se corresponde con el campo *containerName* que posee todo diálogo reusable.
- '*title*', el título de la ventana principal de la aplicación.

Devuelve el diálogo creado, o un valor nulo en caso de que no se haya creado correctamente.

Su funcionamiento es simple, debido a que hace uso de una clase auxiliar, *JIAUtils*, que realiza toda la carga de trabajo. En este método únicamente se invoca el método que desencadena el resto de funcionalidades necesarias, se recoge el valor devuelto, y es ese diálogo precisamente, el creado, el que a su vez devuelve el método.

### 4.8.2. Método *createInstance*

Si no existe una instancia de la factoría, la crea. En caso de ya existir, no hace absolutamente nada. Se trata de un método con acceso interno sincronizado a nivel de la clase, con el fin de evitar problemas en el caso de ejecución concurrente o multi-hilo.

Ni recibe parámetros ni tampoco devuelve ningún valor.

### 4.8.3. Método *getInstance*

Crea, en caso de no existir, una instancia de la factoría, y la devuelve. Internamente lo que hace es una comprobación para ver si el campo *INSTANCIA* es nulo, y en caso de serlo, hace una llamada al método referido en la sección 4.8.2, para crear una nueva instancia.

No recibe parámetros, y el valor devuelto es precisamente la instancia de la factoría, creada en esa invocación o existente de forma previa.

Se define con un acceso global y estático, pues así lo requiere la implementación del patrón, con el fin de que las clases externas que quieran obtener una instancia puedan hacerlo desde cualquier localización, e invocando a la propia clase, que devuelve su instancia única.

#### 4.8.4. Método *clone*

Sobrescribe el método *clone()* de la clase *Object*, lanzando una excepción con el fin de indicar que la instancia del *Singleton* de *JIAFactory* no es clonable. De esta forma, si se intenta invocar el método, el mensaje de la excepción avisará de tal restricción.

Se aplica esta medida, pues el fin mismo del patrón *Singleton* es el de tener una instancia exclusiva para una clase, es decir, un único punto de acceso. Carece de sentido, por tanto, que se pueda clonar la instancia de dicha clase.

### 4.9. JIAUtils

Clase utilitaria creada con el fin de reducir la complejidad de la librería en alguna de sus clases, como en los casos de la factoría (*JIAFactory*) o del diálogo raíz de la jerarquía de reusabilidad (*JIAExtensibleDialog*). Ofrece funcionalidades comunes y auxiliares para varios casos, todo de forma estática y con acceso protegido. Es decir, tiene alcance dentro de la totalidad de la librería, desde cualquier punto, pero no es alcanzable desde fuera de ella.

Contiene 5 campos internos:

- *'L'*, *logger* estándar de Java, de tipo *Logger*, utilizado con el fin de mostrar mensajes de traza con diferentes niveles de prioridad. Según el nivel establecido, el *logger* mostrará unos mensajes u otros por pantalla. Los niveles son inclusivos, de tal forma que si está establecido un nivel, se mostrarán los mensajes de ese nivel y los mensajes de todos los niveles inferiores, a saber:
  - *ALL* : Se trazan todos los mensajes, a cualquier nivel.
  - *FINEST* : Mensajes de información de máximo detalle, útil para localizar errores (depuración nivel 3) .
  - *FINER* : Mensajes de información más detallada, útil para localizar errores (depuración nivel 2) .
  - *FINE* : Mensajes de información detallada, útil para localizar errores (depuración nivel 1)

- *CONFIG* : Mensajes de configuración inicial del programa en el arranque.
  - *INFO* : Mensajes de información, para seguir la ejecución del programa.
  - *WARNING*: Mensajes de error peligroso, que tienen previsto un mecanismo de recuperación.
  - *SEVERE* : Mensajes de error catastrófico, que provocan la terminación del programa.
  - *OFF* : No se genera ninguna traza.
- 
- *'DEFAULT\_BUTTONS\_SIZE'*, tamaño por defecto para los botones de 'Aceptar' y 'Cancelar'.
  - *'DEFAULT\_DIALOG\_SIZE'*, tamaño por defecto para los diálogos creados por la factoría, si el programador no proporciona otro.
  - *'btnOk'*, botón 'Aceptar' del diálogo principal, el cual se resetea en cada generación de diálogo.
  - *'btnCancel'*, botón 'Cancelar' del diálogo principal, el cual se resetea en cada generación de diálogo.

Además, posee un bloque de inicialización estático (como el resto de la clase), con el fin de asignar el nivel de prioridad para los mensajes de log, haciéndolo de esta manera parametrizable. El valor por defecto es el del nivel de *'INFO'*.

Presenta un total de 7 métodos, enumerados y explicados a continuación.

#### 4.9.1. Método *createWithTrivialImplementation*

Crea y devuelve un diálogo del tipo indicado por el parámetro, proporcionando una implementación trivial para sus métodos. Este es el diálogo invocado por la factoría cuando el programador pide generar un nuevo diálogo. Recibe 4 parámetros:

- *'type'*, el tipo de diálogo que se quiere crear.
- *'parent'*, la ventana principal que envuelve el diálogo principal creado.
- *'name'*, el nombre del objeto interno del diálogo que se le quiera asignar, se corresponde con el campo *containerName* que posee todo diálogo reusable.
- *'title'*, el título de la ventana principal de la aplicación.

Devuelve el diálogo creado, o un valor nulo en caso de que no se haya creado correctamente.

El algoritmo comienza creando una variable que contendrá la referencia al objeto diálogo creado. A continuación, hace un *switch*, para el tipo recibido como parámetro. Si el tipo es *'SIMPLE'*, se



crea una nueva instancia de un diálogo simple (*JIASimpleDialog*), con una implementación trivial para los métodos que debe redefinir. Es decir, los cuerpos de los métodos estarán vacíos (*saveThis*, *cleanThis*, *getExternVal*), excepto en el que devuelve un valor lógico (*validateThis*), el cuál devolverá por defecto siempre verdadero.

De forma similar se resolverán los casos para los tipos '*TAB*' y '*TREE*', creando diálogos de tipo Pestañas y Vista de árbol, respectivamente. Después de crear las instancias, el siguiente paso será asignar el *containerName* pasado como parámetro.

Por último, el algoritmo finaliza con sendas llamadas a métodos para añadir la estructura mínima del diálogo (*addMinimalStructure*) y para envolverlo en una ventana de una forma oportuna (*wrapExtensibleDialog*). Luego, simplemente devuelve el diálogo generado tras todo el proceso a través de los métodos comentados.

#### 4.9.2. Método *addMinimalStructure*

Este método contiene la lógica que establece la disposición más adecuada para el diálogo que se está creando. Coloca los botones de 'Aceptar' y 'Cancelar' en la segunda (caso A) o tercera (caso B) fila de un *GridBagLayout*, dejando la primera (caso A) o las dos primeras (caso B) filas disponibles para colocar los posibles componentes que puedan contener los distintos tipos de diálogos. El caso A será el de los diálogos de Pestañas y con Vista de árbol, y el caso B será el del diálogo Simple.

En el caso del simple, podrá contener como máximo otros 2 diálogos reusables. En el caso del diálogo de pestañas, contendrá un *tabbedPane* interno, un panel de pestañas, donde irán sus componentes. Y en el caso del diálogo con vista de árbol, contendrá un *splitPane*, un panel divisor, donde el componente izquierdo será la vista de árbol, y el componente derecho será el panel de contenido, con una distribución *CardLayout*.

Recibe 2 parámetros: el diálogo para el cuál se va a construir la estructura mínima, de tipo *JIAExtensibleDialog*, y el tipo de dicho diálogo, de tipo *JIAExtensibleDialog.Type* (el enumerado interno de esa clase). No devuelve ningún valor como retorno.

El proceso comienza creando algunas variables necesarias durante el procedimiento, como por ejemplo un entero que indica el número de filas que tendrá el diálogo final, para su 'layout'. A continuación, invoca un método (*resetButtons*) para preparar los botones de edición para su uso en el diálogo. Además, establece como verdadero el indicador de '*factoria*' para ese diálogo, así como le asigna el *layout* necesario para el desarrollo (*GridBagLayout*).

Después de todo ello, realiza una evaluación condicional que abre dos posible caminos:

- Si el número de filas del diálogo es 2, significa que será de tipo pestañas o vista de árbol. Por lo tanto, lo que se hace es generar la estructura para el panel de pestañas (*generateTabbedPane*), o bien para el panel divisor (*generateSplitPane*), según el tipo de diálogo respectivo.

Dicha estructura generada se inserta en el diálogo en la primera fila y primera columna.

- Si el número de filas del diálogo es 3, significa que será de tipo simple. Por lo tanto, no hay ninguna estructura interna que crear, ni ningún componente que añadir, salvo los botones de edición, que se situarán en la segunda (tab y tree) o tercera (simple) fila, y en la primera columna para el caso de 'Aceptar', o la segunda columna para el caso de 'Cancelar'.

Una vez superado ese paso, el algoritmo finaliza dándole un tamaño por defecto adecuado al diálogo que se acaba de generar y modelar, para lo cuál se vale del método *setPreferredSizeByType*.

### 4.9.3. Método *resetButtons*

Método utilizado para resetear los botones de edición para cada nuevo diálogo que se vaya a crear. Recibe como parámetro el diálogo al que vincular los botones, de tipo *JIAExtensibleDialog*, y no devuelve nada como retorno.

El algoritmo comienza creando nuevas instancias para las referencias a los objetos que representan los botones (*btnOk* y *btnCancel*). Una vez creadas las instancias, les asigna un tamaño mínimo adecuado, indicado por la constante *DEFAULT\_BUTTONS\_SIZE*.

Continúa agregando un listener a cada uno de los botones, de modo que respondan a la pulsación sobre los mismos cuando ésta se produzca:

- Para el botón de 'Aceptar', se crea una clase anónima que se ejecuta al recibir el evento del botón, y tiene la siguiente secuencia de pasos: primero, intenta validar recursivamente el diálogo, para lo cuál invoca su método *validar*, explicado en la sección 4.4.5:
  - Si la validación tiene éxito, se invocan sucesivamente los métodos para *salvar* (4.4.6) los datos, y para *limpiar* (4.4.7) el diálogo y realizar las tareas de finalización, ambos también recursivos. El algoritmo termina invocando el método de la ventana para que provoca que se cierre y libere sus recursos (*dispose*). Además, si se trata de la última ventana de la aplicación, al cerrarse la ventana, la aplicación también finaliza.
  - Si la validación falla, se muestra un mensaje de advertencia al usuario, notificándole de dicha circunstancia. Cuando el usuario acepta dicho mensaje, se vuelve a mostrar el diálogo para poder continuar la edición, rectificando los datos necesarios.
- Para el botón 'Cancelar' se crea también una clase anónima que se ejecuta al recibir el evento del botón. En esta ocasión el flujo es mucho más sencillo, pues simplemente invoca el método para *limpiar* el diálogo, y su jerarquía de forma recursiva. Posteriormente, el algoritmo termina invocando el método de la ventana para que provoca que se cierre y libere sus recursos (*dispose*). Además, si se trata de la última ventana de la aplicación, al cerrarse la ventana, la aplicación también finaliza.

#### 4.9.4. Método *generateTabbedPane*

Crea un panel de pestañas por defecto, estableciendo la colocación de las pestañas en la parte superior, y la política de disposición de pestañas como *SCROLL\_TAB\_LAYOUT*, lo que quiere decir, que en el caso de que la cantidad de pestañas sea tan grande que no sea posible mostrarlas todas horizontalmente en el espacio disponible de la pantalla, se mostrarán solo las primeras y se habilitarán unos botones de dirección para desplazarse a izquierda y derecha, navegando por el resto.

Recibe dos parámetros. El primero, de tipo *JIAExtensibleDialog*, es el diálogo en el cuál se va a insertar el panel de pestañas. El segundo de tipo *GridBagConstraints*, representa las restricciones con las que se añadirá el panel al diálogo, definidas ya por el método que invoca a éste: *addMinimalStructure*. No devuelve ningún valor de retorno.

El método comienza creando un nuevo panel de pestañas con las características comentadas en el primer párrafo, para más tarde, añadirse al diálogo pasado como parámetro de dos formas: por un lado, como un componente gráfico de Swing, colocándolo correctamente, y por otro, asignando la instancia creada del panel a la propiedad interna *tabbedPane* del diálogo. De tal forma que el diálogo no solo contenga visualmente el panel como un componente más, sino que además tenga una referencia interna a él, para poder usarlo.

#### 4.9.5. Método *generateSplitPane*

Crea un panel divisor por defecto, con una vista de árbol a su izquierda, y un panel de tarjetas a su derecha. Ambos componentes, a izquierda y derecha, se envolverán en un *JScrollPane*, con el fin de mantenerlos visibles, sin necesidad de que ocupen más espacio. La vista de árbol estará vacía por defecto y su raíz no será visible.

Recibe dos parámetros. El primero, de tipo *JIAExtensibleDialog*, es el diálogo en el cuál se va a insertar el panel divisor que se genere. El segundo de tipo *GridBagConstraints*, representa las restricciones con las que se añadirá el panel al diálogo, definidas ya por el método que invoca a éste: *addMinimalStructure*. No devuelve ningún valor de retorno.

El algoritmo inicia su ejecución creando el panel de tarjetas (un *JPanel* con distribución *Card-Layout*), donde se localizará el contenido en forma de diálogos. Se le asigna por defecto un tamaño mínimo, con el objetivo de que no se pueda redimensionar por debajo de ese tamaño, al mover la barra divisoria.

Después, se pasa a construir la vista de árbol, de tipo *JTree*, dándole una serie de características concretas: modo de selección simple para los nodos, mayor altura para las filas de cada entrada del árbol, un nodo raíz por defecto de nombre "(default)", y lo más importante, se crea una clase anónima como listener del árbol, de tipo *TreeSelectionEvent*, la cual contiene la lógica necesaria para que se muestre el diálogo adecuado en el panel de contenido, cuando se pulsa una entrada concreta

de la vista de árbol.

Se continúa creando el panel divisor, de tipo *JSplitPane*, con una orientación horizontal, estableciendo como su componente izquierdo la vista de árbol, y como su componente derecho el panel de contenido, con la distribución de 'tarjetas' (*CardLayout*). Además, se fija una posición inicial para la línea divisoria, y se aumenta su tamaño respecto al predeterminado.

Termina el proceso añadiendo el panel divisor creado, con sus componentes izquierdo y derecho anteriormente explicados, al diálogo de vista de árbol pasado como parámetro. Además, también se enlazan correctamente las propiedades de dicho diálogo con vista de árbol, de tal forma que tenga referencias internas a los objetos que representan al panel de contenido, al panel divisor, y a la vista de árbol, contruidos anteriormente, y añadidos de forma gráfica al diálogo como componentes.

#### 4.9.6. Método *setPreferredSizeByType*

Recibe un diálogo, y según su tipo, le asigna un tamaño preferido por defecto. Los parámetros de entrada son, por un lado, el diálogo de tipo *JIAExtensibleDialog*, y por otro, el tipo, definido como un enumerado *JIAExtensibleDialog.Type*. No devuelve ningún valor como retorno del método.

El funcionamiento interno es sencillo. Simplemente evalúa la condición que representa el tipo, de tal forma que dependiendo cuál sea, asigna un tamaño u otro al diálogo, teniendo como base siempre la constante *DEFAULT\_DIALOG\_SIZE*.

#### 4.9.7. Método *wrapExtensibleDialog*

Crea o utiliza una envoltura para el diálogo principal generado en la factoría. Recibe 3 parámetros: una envoltura personalizada, de tipo *JFrame*, el diálogo a envolver, de tipo *JIAExtensibleDialog*, y por último un título, de tipo *String*. No devuelve ningún valor como retorno.

Comienza creando una variable *wrapper* para contener la referencia al objeto de la ventana. A continuación, evalúa una condición lógica, comprobando si la ventana pasada como parámetro es nula (lo cuál está contemplado) o no lo es. Si es nula, crea una nueva ventana por defecto, y si no lo es, asigna esa instancia del parámetro al *wrapper*.

Por último, le asigna una serie de características a la ventana definida anteriormente: la operación por defecto cuando se cierra se establece en *DISPOSE*, su tamaño mínimo se establece como el mismo que el tamaño preferido del diálogo principal que contiene, se asigna un título para la ventana utilizando el parámetro proporcionado, o en su defecto, el título que ya contuviera la ventana, si es una personalizada, o, en última instancia, el nombre de la clase del diálogo, de forma predeterminada, y también establece el foco por defecto al mostrar la ventana en el botón de 'Aceptar', por comodidad.

Finaliza añadiendo el diálogo principal a la ventana, y haciéndola visible.

# Capítulo 5

## Experimentación y pruebas

Este capítulo abordará una fase esencial de todo proceso de ingeniería software, la etapa de experimentación, o de pruebas. En ella, se evalúa el software bajo construcción en base a una serie de ensayos y procesos de validación y verificación, con el fin de comprobar y estudiar su fiabilidad, robustez y consistencia, así como su adecuación a diferentes situaciones.

En el caso que nos ocupa, dichas pruebas se han articulado por medio de una aplicación software estándar de Java, donde se ha utilizado la librería, a través de sus posibles modos de uso, para examinar cada una de las funcionalidades que ofrece.

Además de ello, también se ha hecho uso del módulo NBM aportado como complemento a la librería, el cuál es ciertamente provechoso en lo referente a la creación y construcción de los diálogos de la aplicación, pues resuelve gran parte del trabajo en la generación de estas GUIs. Provee una estructura predefinida en forma de plantillas, así como una serie de propiedades de los diálogos, asignadas inicialmente en valores apropiados para su uso.

Se dividirá en 4 secciones, correspondientes a los principales puntos fuertes de la librería, como son los diferentes tipos de diálogos que ofrece para su uso, el proceso de combinación e inserción de unos diálogos dentro de otros, permitiendo su reutilización en nuevas estructuras y jerarquías, el sistema recursivo para las acciones de edición del usuario, mediante un solo paso (transacción atómica), o el mecanismo de paso de mensajes, que permite la comunicación entre diálogos ajenos entre si, e inicialmente independientes.

### 5.1. Variedad de tipos de diálogos

La librería, como se ha comentado en ocasiones anteriores, ofrece los siguientes tipos de diálogos:

- GUIs de una sola página, denominados simples, en los que se añaden como máximo dos diálogos.
- GUIs con sistema de navegación mediante pestañas.

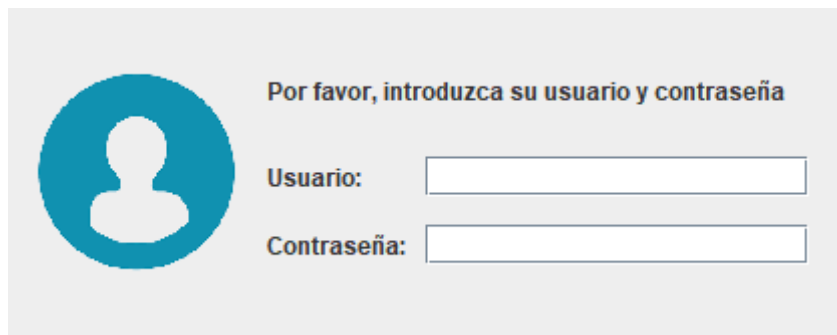


Figura 5.1: Ejemplo de uso de diálogo de tipo Simple

- GUIs con navegación mediante vista de árbol jerárquica.

En el programa de prueba desarrollado, se pueden encontrar diversas situaciones donde se usa alguno de los diferentes tipos de diálogo, aprovechando al máximo las características que ofrece la librería.

Como ejemplo en el que se emplea un diálogo de tipo simple, tenemos:

- Pantalla de inicio de sesión de la aplicación (*LoginVC*). Es la pantalla inicial que se muestra nada más arrancar el programa. Utiliza un diálogo simple, con un *layout* determinado, para disponer sus elementos de una forma ordenada. Mostrada en la figura 5.1.

Como ejemplo en el que se emplea un diálogo de tipo pestañas, tenemos:

- Pantalla de resumen global del paciente (*resumen*). Se trata de una pantalla creada 'ad hoc' para la aplicación, por parte del programador, pero no ideada inicialmente por el diseñador de interfaces. En ella, se utilizan varios diálogos construidos por el diseñador, colocándolos en diferentes pestañas del diálogo, y aprovechando así el modo de exploración de contenido que ofrece. Mostrada en la figura 5.2.

Como ejemplo en el que se emplea un diálogo de tipo vista de árbol, tenemos:

- Pantalla de la ficha de un paciente concreto (*fichaPaciente*). Creada también por el programador en tiempo de ejecución, pero no existente como entidad completa en la arquitectura estática de la aplicación. Se emplea la opción de navegación mediante un árbol de entradas jerárquicamente organizado. Mostrada en la figura 5.3.

Se pueden ver la pluralidad de tipos que ofrece la librería, e incluso, dada su naturaleza y diseño, se podrían plantear nuevos tipos, como extensiones adicionales, en un futuro, si así lo requiriesen sus usuarios. Algunas ideas relativas se comentan en el siguiente capítulo, en la sección 6.2, cuando se hace referencia a posibles ampliaciones.

Resumen Edición de datos

Tiburcio Rodriguez, Tristán

Código SNS | 900000001

Estado

Urgencia 0%

Figura 5.2: Ejemplo de uso de diálogo de tipo Pestañas

Ficha de paciente

- Resumen
- Información
  - General
  - Personal
  - Clínica
  - Bancaria

Datos clínicos

Doctor de cabecera:

Medicación actual:

☐ Reanimación Cardio Pulmonar (RCP)

Alergias conocidas

Prioridad triaje

0 1 2 3 4 5 6 7 8 9 10

Figura 5.3: Ejemplo de uso de diálogo de tipo Vista de árbol

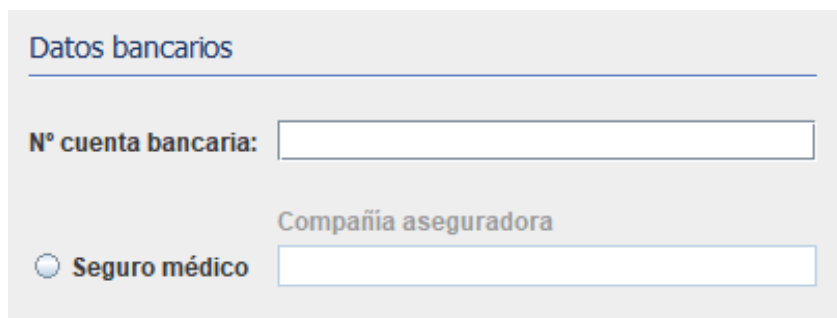


Figura 5.4: Pantalla de datos bancarios de un paciente

## 5.2. Posibilidades de integración y combinación de diálogos

En lo referente a la combinación de diálogos y las facultades que dispone la librería para ello, se ha explicado y contado con detalle a lo largo del capítulo 4, haciendo mención a cómo es posible combinar cada diálogo con cualquier otro, o con un conjunto de otros diálogos, dependiendo del tipo de diálogo del que se trate.

Como ejemplo global de esas posibilidades, y de la forma en que se ha usado en nuestro programa de pruebas, vamos a exponer la pantalla utilizada para mostrar información financiera del paciente (*DatosBancariosVC*). Se trata de una pantalla ideada y construida por el diseñador de interfaces. Se muestra en la figura D.9.

Dicho diálogo ha sido integrado en hasta 3 ocasiones distintas, una por cada tipo de diálogo de la librería, tomando ventaja así, de las enormes alternativas de composición que ofrece la librería:

- Por un lado, podemos verla añadida como parte de un diálogo simple, formando parte de la *factura* médica del paciente. Figura D.13.
- Por otro lado, también se ha utilizado como parte de la pantalla de modificación de datos del paciente, añadida en una pestaña más de un diálogo de tipo Pestañas. Figura D.14.
- En último lugar, se ha empleado en la *ficha de paciente*, accesible para el perfil de 'facultativo'. En esta ocasión, se trata como una entrada más del árbol, bajo la rama de información del paciente, junto a todas las demás. Figura D.15.

En todos los casos, se puede apreciar cómo la librería se encarga de recolocar los componentes del diálogo, y de ajustar su tamaño, expandiéndolo si es necesario, pero manteniendo su estilo visual.

Asimismo, no solo expande el propio diálogo reutilizado, sino también el diálogo padre en el cuál se está insertando, si es necesario. Esto quiere decir, que, en el caso de que intentásemos agregar un diálogo hijo a un diálogo padre, siendo el tamaño del primero superior al del segundo, la librería tomaría el control de la situación, aumentando el tamaño del padre lo que fuese indispensable.



### 5.3. Edición de datos mediante transacción atómica

Una de las características que posee la librería, es la de realizar la edición de información de sus diálogos de forma indivisible, es decir, en un solo paso. Esto es lo que se conoce como 'transacción atómica', propiedad mediante la cuál la librería habilita que, cuando el usuario decide aceptar o rechazar los cambios realizados, todo el proceso se produce en una sola acción.

Por tanto, si el usuario pulsa el botón para aprobar las modificaciones de datos hechas en cualquiera de los diálogos de una misma jerarquía, la librería se encargará de recorrer, diálogo a diálogo, dicha jerarquía, para, en primer lugar, validar cada uno de ellos de la forma en que haya definido el diseñador. A continuación, y solo si la validación de todos los diálogos ha resultado exitosa, procede a realizar el guardado de datos, y posteriormente, las tareas de finalización y limpieza de los diálogos.

Por otro lado, si el usuario pulsa el botón de rechazar los cambios, la librería se encargará también de ejecutar las labores necesarias de terminación y limpieza de los diálogos relacionados en la jerarquía, con el fin de atender la solicitud.

Ambos procesos se producen de forma recursiva, lo que quiere decir, que la librería va recorriendo el árbol de diálogos, uno a uno, para aplicar sobre cada uno de ellos las acciones que haya definido individualmente el diseñador para ese diálogo. Comienza desde la parte superior de la jerarquía, descendiendo hasta llegar a los diálogos 'hojas', que no tienen descendientes, y empieza a ascender de nuevo, ejecutando, para cada uno, las acciones apropiadas, hasta llegar de nuevo a la raíz.

En nuestro programa de prueba, esto se puede comprobar en muchos de los diálogos diseñados. Cogemos como ejemplo la pantalla de *DatosClinicosVC*, la cuál tiene implementadas las acciones recursivas mostradas en el listado 5.1.

- En primer lugar, podemos ver la acción de validación del diálogo, que para este diálogo solo comprueba que exista un doctor asignado (línea 3) y que el indicador del nivel de triaje esté comprendido entre valores admitidos por la aplicación, es decir, entre 0 y 10, ambos incluidos. Si se cumplen ambas condiciones, la validación devolverá un valor verdadero, indicando éxito, para este diálogo concreto.
- En segundo lugar, tenemos las sentencias para el guardado de los datos, que guardarán los valores que es seguro que tendrán algún valor asignado, como los dos campos comprobados anteriormente (doctor y nivel de triaje), además del indicador RCP, puesto que este último, en caso de que el usuario no haga nada, tendrá el valor almacenado en la base de datos, seleccionado o no, y por tanto siempre se podrá recoger su valor verdadero o falso. Además de ello, verifica si existe un valor no nulo y no vacío para los campos de 'medicación' y 'alergias', y en caso de ser así, también guarda los datos en el modelo de la forma oportuna.
- Por último, vemos que en esta ocasión, la acción para la limpieza y finalización tiene una

implementación trivial, es decir, vacía, puesto que el diseñador ha considerado adecuado no aprovechar dicha función, o no le ha sido necesaria.

Ejemplos como éste, hay repartidos a lo largo de la aplicación, para tareas de validación, guardado y finalización de otros diálogos, definiéndolos en cada caso de forma específica y adaptada al diálogo en cuestión, y aprovechando el sistema de edición de datos que ofrece la librería. Para ello, en ningún momento el diseñador tiene que preocuparse de nada, así como tampoco tiene que ocuparse de disparar la recursividad intrínseca a estas acciones, puesto que para ello ya existe la librería.

## 5.4. Mecanismo de paso de mensajes

En esta última sección se comentará otra de las fortalezas de la librería, como es el hecho de habilitar un mecanismo de comunicación entre los diálogos de la misma jerarquía, aún cuando originalmente éstos no estuvieran ideados para transmitirse información.

La librería orquesta un mecanismo de paso de mensajes, que conecta todos los diálogos entre sí, de forma bidireccional al cargarlos en memoria durante la ejecución. Por tanto, permite que cualquiera de esos diálogos envíe mensajes a cualquier otro, sea el que sea, y también permite que ese diálogo reciba mensajes procedentes de cualquier otro.

Esta alternativa ya se comenta con mayor profundidad durante el anexo correspondiente al manual de uso de la librería, concretamente en la sección D.3.2. Por ello, en este apartado nos limitaremos a referenciarlo adecuadamente, recordando por encima su funcionamiento.

Los dos ejemplos de paso de mensajes de los que se habla en el manual son:

- Por un lado, el que denomina como caso A, mostrado en las figuras D.16, D.17, D.18 y D.19. En este primer caso, tenemos un mensaje que es enviado desde la pantalla de *Datos Generales* del paciente. El mensaje se origina en el componente combobox que muestra y permite seleccionar el estado del paciente. El diálogo encargado de recibir el mensaje, o mejor dicho, apto para ello, es el de *Resumen* del paciente. Más concretamente, el componente específico que se ve afectado es el cuadro que muestra la imagen (icono) del estado actual, así como el *tooltip* asociado, que aparece al pasar el ratón por encima de ella.
- Por otro lado, el nombrado como caso B, representado en las figuras D.20, D.21, D.22 y D.23. En esta ocasión, el mensaje se origina en el componente JSlider (barra deslizadora) del diálogo *Datos Clínicos* del paciente, el cuál emite el mensaje en todas direcciones, es decir, con una emisión 'broadcast'. Por otro lado, la parte receptora es, una vez más, la pantalla de *Resumen* del paciente, aunque en esta ocasión el componente relacionado es la barra de progreso vertical, que muestra el nivel de urgencia del paciente en función de su prioridad de triaje, utilizando además un código de colores según la gravedad.

## Listado de código 5.1: Acciones recursivas definidas para los Datos Clínicos

```
1
2 public boolean validateThis() {
3     return Utils.validateString(tfDoctor.getText())
4         && sliderTriage.getValue() >= 0
5         && sliderTriage.getValue() <= 10;
6 }
7
8 public void saveThis() {
9     this.model.setAssignedDoctor(tfDoctor.getText());
10    this.model.setRcp(cbRCP.isSelected());
11    this.model.setTriagePriority(sliderTriage.getValue());
12
13    if(Utils.validateString(tfMedication.getText()))
14        this.model.setCurrentMedication(tfMedication.getText());
15
16    if(Utils.validateString(taAllergies.getText()))
17        this.model.setAllergiesByLine(taAllergies.getText());
18 }
19
20 public void cleanThis() { }
```



## Capítulo 6

# Conclusiones y trabajos futuros

Este capítulo pretende servir como broche final al proyecto, exponiendo las conclusiones alcanzadas tras su realización. Se recordarán de forma breve los objetivos y las metas marcadas para el proyecto en sus inicios, y se irá comentando como se ha logrado cada uno de ellos a través del trabajo realizado con la librería y los demás apartados desarrollados.

Por último, se finalizará con algunas ideas de futuro, pensamientos respecto a posibles ampliaciones que poder hacer para el proyecto, más adelante. En definitiva, mejoras de cara a la librería, con el fin de hacerla aún más eficiente, aumentar sus posibilidades, y desarrollar las funcionalidades y alternativas que ofrece.

### 6.1. Conclusiones

Se recuperarán en este apartado, de manera general, los objetos perseguidos por el proyecto, e iremos viendo de qué manera se han ido cumpliendo, acorde a las características y funcionalidades construidas. Podemos empezar enfocando algunos puntos claramente diferenciados para los dos roles principales que harán uso de la librería: el programador de aplicaciones, y el diseñador de interfaces gráficas de usuario. Más tarde, continuaremos con algunos objetivos más globales y transversales, verificando a su vez su realización y adecuación.

En el caso del desarrollador, la librería pretendía proporcionarle las siguientes posibilidades:

- Solicitud de diálogos vacíos, de tal forma que disponga de un espacio sobre el cuál comenzar a combinar e insertar los diálogos del diseñador, de la manera que precise el programador. Esto se ha logrado mediante el desarrollo de una factoría de diálogos, la cuál permite generar y obtener diálogos del tipo indicado, con una estructura mínima: botones de edición para aceptar o rechazar las modificaciones realizadas, contenedor apropiado al tipo de diálogo, con una distribución asociada, y en el caso de los diálogos de tipo vista de árbol, además, cuenta

con un componente interno para manejar esa vista jerárquica de árbol.

- Mecanismo de inicialización de diálogos, tanto los propios de la factoría, como los del diseñador. En este caso, este logro se articula a través de un método general, conocido como *setUpDialog*, el cuál permite poner a punto las propiedades necesarias de reusabilidad, de cara a asegurar la compatibilidad total del diálogo dentro de la jerarquía de diálogos reusables. El método se define ya en la clase padre de toda la jerarquía, *JIAExtensibleDialog*, de tal forma que cualquier diálogo que pertenezca a la jerarquía, y por tanto extienda ese diálogo padre, lo heredará, y podrá incluso, si lo ve necesario, redefinirlo.
- Engranaje de combinación e inserción de diálogos entre sí. Uno de los puntos principales de la librería. Conseguido a través de sendos métodos: *addExtensibleChild* y *addExtensibleChildrenList*. Estos métodos, de la misma forma que en el caso anterior, están presentes ya en la raíz de la jerarquía de reusabilidad, definiendo una implementación genérica para todas las necesidades comunes a este tipo de operación. Después, cada tipo de diálogo concreto los sobrescribe, adaptándolos a sus casos particulares. La funcionalidad implementada consigue insertar, colocar (y recolocar en caso de ser necesario), así como redimensionar, los diálogos implicados y sus componentes de la forma propicia.

Por otro lado, el diseñador de interfaces también tenía una serie de necesidades específicas:

- Posibilidad de arrastrar componentes gráficos hacia su zona de trabajo, pudiendo ser éstos alguno de los propios tipos de diálogos de la librería. Es decir, la librería debía ofrecer sus diálogos en el constructor de interfaces elegido, en este caso el de NetBeans, para que el diseñador pudiera usarlos como cualquier otro elemento del framework gráfico empleado. Esto se ha conseguido de dos formas. En primer lugar, conectando de alguna forma la jerarquía de diálogos reusables con los componentes de Swing, mediante la relación de herencia existente entre la raíz de la jerarquía (*JIAExtensibleDialog*) y el componente *JPanel*, nativo de Swing. De esta forma, se posibilita que los diálogos reusables, en última instancia componentes de tipo *JPanel*, puedan ser tratados y usados como componentes Swing estándar. En segundo lugar, haciendo aparecer en la Paleta de NetBeans los tipos de diálogos en si mismos, como *beans*, para lo cuál se han tenido que construir con unas directrices mínimas y adecuadas, que les habiliten como componentes de esas características.
- Obligación de implementar ciertos métodos con el fin de conseguir la ansiada compatibilidad como diálogos cien por cien reusables. Como se puede ver, más que una necesidad del diseñador, es una imposición de la librería, con el fin de facilitar el entendimiento entre los diálogos contruidos por el diseñador, y los del resto de la librería. Logrado a través del uso de métodos *default*, en las interfaces que declaran esas funcionalidades,

las cuáles más tarden heredan y redefinen los diálogos. La estrategia seguida, ha sido la de escribir una implementación que provoque el lanzamiento de una excepción, cuando dichos métodos son invocados y ejecutados. Esa excepción avisa con un mensaje, al diseñador, de la necesidad de redefinir esos métodos, con lo que se consigue la exigencia requerida.

En último lugar, existían otra serie de metas no asociadas estrictamente con ninguno de los roles, o bien transversales a ambos, que se detallan a continuación:

- Sistema de comunicación entre los diálogos asociados en la misma jerarquía.  
Objetivo alcanzado mediante el mecanismo de paso de mensajes implementado en la librería, el cuál se encarga de enlazar de forma bidireccional cada par de diálogos, permitiendo así que todos puedan enviar y recibir mensajes de cualquiera de los otros.
- Edición de datos de forma atómica, de cara al usuario final.  
Logrado a través del sistema habilitado por la librería, para ejecutar las acciones de aceptación o rechazo de las modificaciones, en cada uno de los diálogos de la jerarquía, como una transacción atómica, es decir, en un solo paso.

## 6.2. Trabajos futuros

Aquí se señalarán muy brevemente algunas propuestas de cambio a futuro. Ideas que podrían mejorar la librería en alguna de sus facetas, bien sea aumentando su eficiencia, o bien añadiendo nuevas funcionalidades o características no contempladas en su concepción y diseños originales.

- En primer lugar, podemos comenzar hablando de algo que podría resultar como una necesidad bastante obvia: disponer de más y diferentes tipos de diálogos. La librería ofrece 3 tipos básicos, suficientemente variados, pero un diseñador experimentado podría verse en la necesidad de requerir alguna clase de diálogo no contemplada actualmente en la organización establecida. Por ejemplo, un diálogo cuyo sistema de navegación por el contenido fuese a través de menús, o bien un diálogo que permitiese realizar búsquedas indexadas de contenido, a través de un componente concreto, como una caja de texto, o un combobox editable.  
Para ello, se debería realizar una ampliación de la librería, creando la clase que representase el nuevo tipo de diálogo deseado, y haciendo que éste extendiese alguno de los tipos ya existentes en la jerarquía de diálogos reusables. Bien heredando las propiedades del diálogo genérico, el padre de la jerarquía, o bien incluso, como subtipo de alguno de los tipos ya definidos.
- Por otro lado, se podría realizar una mejora de la eficiencia del sistema de comunicación entre diálogos. Para ello, podríamos modificar el mecanismo de paso de mensajes actualmente establecido, con el fin de conseguir una reducción de costes en memoria, en tiempo de ejecución,

o incluso en ambos.

Actualmente, el mecanismo conserva una lista almacenada en cada uno de los diálogos que pertenecen a la jerarquía, siendo éstas, listas con contenido prácticamente idéntico, y por tanto, replicadas y redundantes entre sí. Una alternativa es la de establecer una lista única, común a toda una misma jerarquía, almacenada quizá, en la raíz de dicha jerarquía, el diálogo principal. De esta forma, esa lista sería la que iría añadiendo referencias a todos los diálogos existentes, para, más tarde, ser invocada desde cualquiera de los diálogos de la jerarquía, incluido el propio diálogo principal, con el fin de emitir un mensaje de difusión global.

- Por último, se podría plantear un mecanismo de reutilización de instancias de los diálogos, con el objetivo final de reducir el gasto de memoria de una aplicación que requiriese una gran cantidad de diálogos iguales para funcionar.

Podría darse el caso, como ejemplo, del desarrollo de una aplicación de gestión bancaria, que contase con una pantalla resumen de los datos de un cliente, y que dicha pantalla fuese igual para todos los clientes, solo difiriendo los valores de los datos mostrados, y para cambiar de un cliente a otro, bastaría con realizar una pulsación de ratón, o bien una búsqueda por nombre. En ese caso, aunque externamente de cara al usuario, existirían tantas pantallas diferentes como clientes, a nivel interno, se crearía únicamente una instancia del diálogo a reutilizar. Para cada cambio entre clientes, en lugar de eliminar el diálogo del cliente anterior, generar el diálogo del cliente nuevo, y cargar sus datos, lo que se haría es, directamente, limpiar el diálogo, borrando los datos actuales, y cargando los valores del nuevo cliente seleccionado.



# Bibliografía

- Álvarez, M. A. (2014). Polimorfismo, concepto. <https://desarrolloweb.com/articulos/polimorfismo-programacion-orientada-objetos-concepto.html>.
- Anonymous (2004). Arquitectura mvc. <https://es.wikipedia.org/wiki/Modelo-vista-controlador>.
- Anonymous (2007). Zk framework. [https://es.wikipedia.org/wiki/ZK\\_Framework](https://es.wikipedia.org/wiki/ZK_Framework).
- Apple, I. (2015). Mvc cocoa. <https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>.
- Cmgustavo, a. (2004). Simula (lenguaje de programación). <https://es.wikipedia.org/wiki/Simula>.
- Derksen, B. (2004). Paradigma wimp. [https://en.wikipedia.org/wiki/WIMP\\_\(computing\)](https://en.wikipedia.org/wiki/WIMP_(computing)).
- Dibujon, a. (2007). Reutilización de código, concepto. [https://es.wikipedia.org/wiki/Reutilización\\_de\\_código](https://es.wikipedia.org/wiki/Reutilización_de_código).
- Edgar (2002). Poo. [https://es.wikipedia.org/wiki/Programación\\_orientada\\_a\\_objetos](https://es.wikipedia.org/wiki/Programación_orientada_a_objetos).
- José Cerrada, Manuel Collado, o. (2000). *Introducción a la Ingeniería del Software*. Editorial Centro de estudios Ramón Areces, S.A.
- Juarez, M. (2011). Patrón builder. <http://migranitodejava.blogspot.com.es/2011/05/builder.html>.
- Kitsonk, a. (2005). Voice user interface. [https://en.wikipedia.org/wiki/Voice\\_user\\_interface](https://en.wikipedia.org/wiki/Voice_user_interface).
- K.lee, a. (2002). Librería, concepto. [https://en.wikipedia.org/wiki/Library\\_\(computing\)](https://en.wikipedia.org/wiki/Library_(computing)).
- Larman, C. (2003). *UML y patrones*. Prentice Hall.

- Lynxblack, a. (2015). Historia de las guis. <https://frikosfera.wordpress.com/2015/02/09/historia-de-las-interfaces-de-usuario-gui/>.
- Nopuomas, a. (2005). Modelo cocomo. <https://es.wikipedia.org/wiki/COCOMO>.
- Oracle, I. (1997). Javabeans specification. <http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/>.
- Oracle, I. (1999). Convención de nombres en java. <http://www.oracle.com/technetwork/java/codeconventions-135099.html>.
- Oracle, I. (2005). Import declarations. <http://docs.oracle.com/javase/specs/jls/se6/html/packages.html>.
- Oracle, I. (2011). Trail: Javabeans(tm). <http://ads.gigatux.nl/tutorial7/javabeans/index.html>.
- Oracle, I. (2014a). Netbeans code template module tutorial. <https://platform.netbeans.org/tutorials/nbm-code-template.html>.
- Oracle, I. (2014b). Netbeans file template module tutorial. <https://platform.netbeans.org/tutorials/nbm-filetemplates.html>.
- Poleydee, a. (2009). Natural user interface. [https://en.wikipedia.org/wiki/Natural\\_user\\_interface](https://en.wikipedia.org/wiki/Natural_user_interface).
- Potix, I. (2017). Página oficial zk. <https://www.zkoss.org>.
- TakuyaMurata, a. (2003). Patrón observer. [https://en.wikipedia.org/wiki/Observer\\_pattern](https://en.wikipedia.org/wiki/Observer_pattern).
- Trashgod, a. (2010). When to use eventlistenerlist instead of a general collection of listeners. <https://stackoverflow.com/questions/3240472/when-to-use-eventlistenerlist-instead-of-a-general-collection-of-listeners>.
- Vialfa, C. (2017). Herencia, concepto. <http://es.ccm.net/contents/411-poo-herencia>.
- Yearofthedragon, a. (2005). Información hardcodeada. [https://es.wikipedia.org/wiki/Hard\\_code](https://es.wikipedia.org/wiki/Hard_code).
- Zifra, a. (2006). Grafo completo, definición. [https://es.wikipedia.org/wiki/Grafo\\_completo](https://es.wikipedia.org/wiki/Grafo_completo).

# Anexo A

## Manual de instalación del entorno

En este manual se detallarán los pormenores relativos a la instalación y configuración iniciales de la librería, así como del módulo NetBeans asociado. La explicación se realizará basándose en el **entorno** que se enumera a continuación, y, aunque no se asegura una compatibilidad total fuera de dicho entorno, se puede suponer que previsiblemente todo funcionará de igual manera si se cuenta con versiones de software similares, anteriores o posteriores. Sus características son:

- Ubuntu 17.04 x86\_64
- JDK 1.8.0.144 (64 bit)
- NetBeans 8.2

Cabe destacar que el entorno sobre el que se ha desarrollado la librería es Windows 10 Pro x64, usando el JDK 1.8.0.121 (64 bit), por lo que también se puede asegurar su compatibilidad.

El manual comienza explicando cómo importar la librería desarrollada, para habilitar su uso en cualquier proyecto estándar de Java. A continuación, se comentan los pasos necesarios para instalar el módulo NBM, que contiene los *templates* necesarios para la construcción de las interfaces gráficas de usuario, que son la razón de ser de la librería. Por último, se abordará la forma de importar y mostrar los distintos tipos de diálogo que ofrece la librería en la Paleta de NetBeans, lo que abre la posibilidad de crear nuevos diálogos simplemente arrastrándolos desde la mencionada Paleta.

En resumen: importar la librería, instalar el módulo NBM, e importar los *JavaBeans* en la Paleta.

### A.1. Importar la librería

Como se ha comentado, será requisito indispensable tener instalado el IDE NetBeans. Además, será necesario un proyecto existente, por lo que en caso de no tenerlo, deberemos crear uno nuevo.

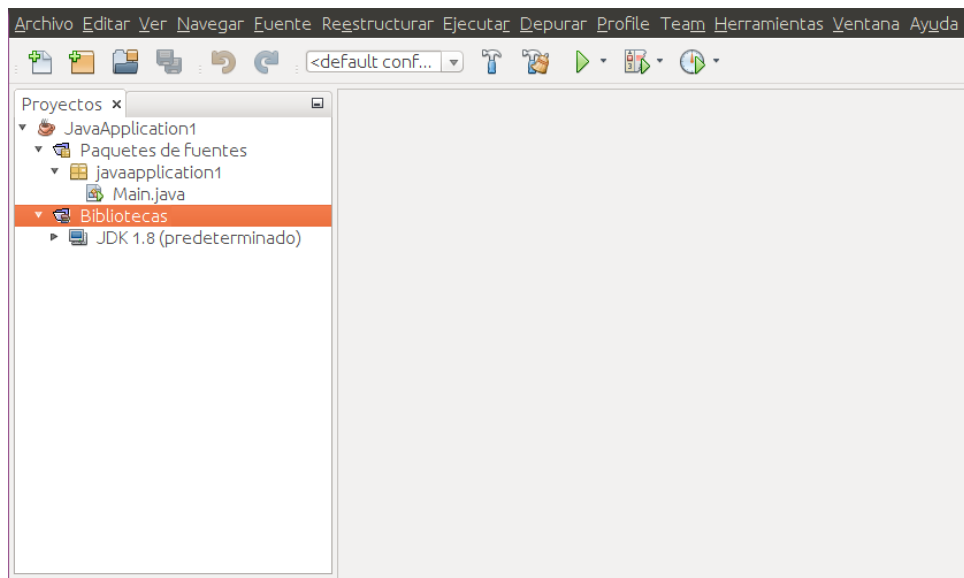


Figura A.1: Pantalla inicial de NetBeans

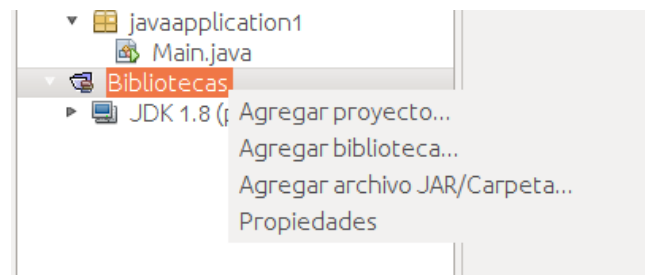


Figura A.2: Menú para agregar dependencias

Una vez abierto NetBeans, y teniendo el proyecto sobre el que deseamos importar la librería creado, nos encontraremos con algo como lo mostrado en la figura A.1.

Una vez aquí, dentro de la vista de Proyectos, en el nodo que dice Bibliotecas, deberemos hacer clic derecho y pulsar en 'Agregar archivo JAR/Carpeta' (figura A.2). Se abrirá un menú donde deberemos indicar la ruta del archivo que deseamos importar, que en nuestro caso será la librería proporcionada, cuyo nombre es 'pfg-uned-library.jar'.

Una vez agregada, podremos comprobar, si se encuentra dentro de las Bibliotecas del proyecto, que efectivamente se ha añadido correctamente, como se ve en la figura A.3.

A partir de entonces, podremos referenciar y utilizar el código y los componentes de la librería directamente desde el proyecto en el que la hemos importado, pudiendo aprovechar las funcionalidades que ofrece. Con esto bastaría para poder utilizar el trabajo desarrollado, aunque, sin embargo, existen dos formas más de facilitar el uso de la librería y el desarrollo de software y construcción de interfaces con ella. Para ello tenemos los *templates*, y los *beans*.

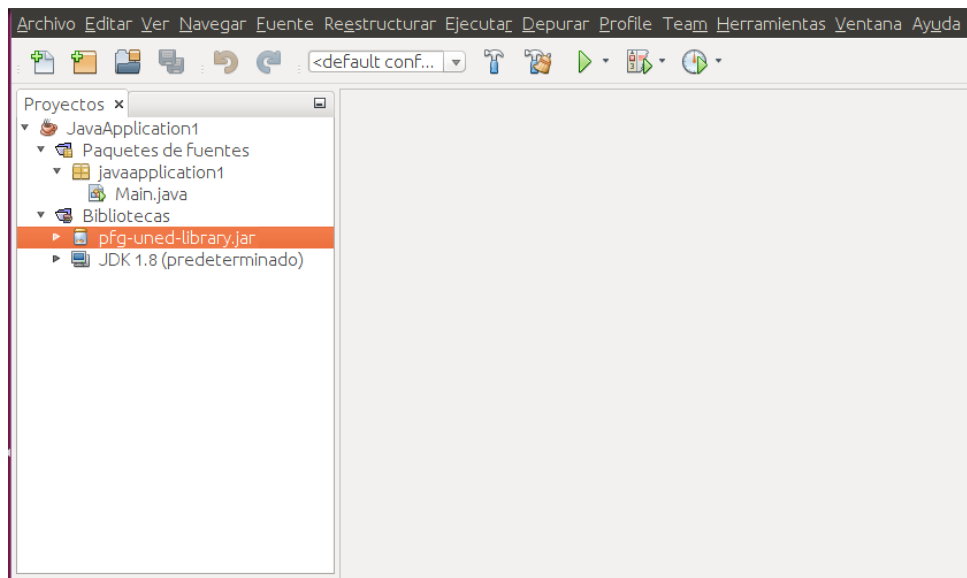


Figura A.3: Librería añadida correctamente a las dependencias del proyecto

## A.2. Instalar el módulo NBM

Los *templates* son meras plantillas para GUIs, que extienden una clase de alguno de los tipos de diálogo de la librería, y ofrecen un componente con una serie de características ya establecidas de antemano, lo que ahorra el trabajo de tener que construir dicho componente, y el diseñador puede pasar directamente a diseñar y construir el diálogo que desee. Básicamente disponemos de 3 templates diferentes, cada uno de los cuales corresponde a uno de los tipos de diálogo disponibles: Simple, con Pestañas, o con Vista de Árbol.

Para poder crear una de estas GUIs utilizando simplemente el menú contextual de Archivo Nuevo que ofrece NetBeans, debemos instalar dichas plantillas como un *plugin*, utilizando para ello el archivo de nombre 'templates.nbm', que se proporciona con el resto del material. Nos dirigiremos entonces al menú de Herramientas (figura A.4), y pulsaremos en el ítem Plugins.

Se nos abrirá una ventana (fig A.5) en la que deberemos navegar hasta la pestaña de Descargados, y una vez ahí, pulsar el botón situado en la esquina superior izquierda que reza 'Agregar Plugins...'. Dicho accionable nos abrirá otra nueva ventana en la que podremos buscar y seleccionar nuestro módulo NBM, cosa que haremos. Una vez seleccionado el fichero, la ventana de Plugins pasará a mostrar uno nuevo como disponible, y podremos hacer clic, con el nuevo plugin seleccionado, en el botón 'Instalar' de la esquina inferior izquierda, ahora habilitado (fig A.6).

Tras ello, podemos seguir el menú de instalación, donde simplemente tendremos que avanzar a través de las pantallas de configuración, aceptar la licencia de uso y finalmente darle a instalar. Es probable que cuando lo hagamos, nos aparezca una pantalla como la mostrada en la figura A.7, puesto que el módulo no está firmado digitalmente, por lo que NetBeans desconfía de él.

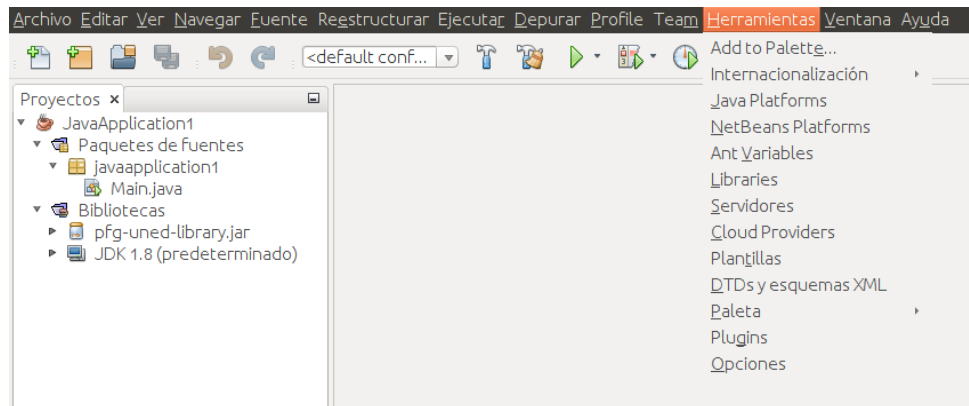


Figura A.4: Menú de herramientas NetBeans

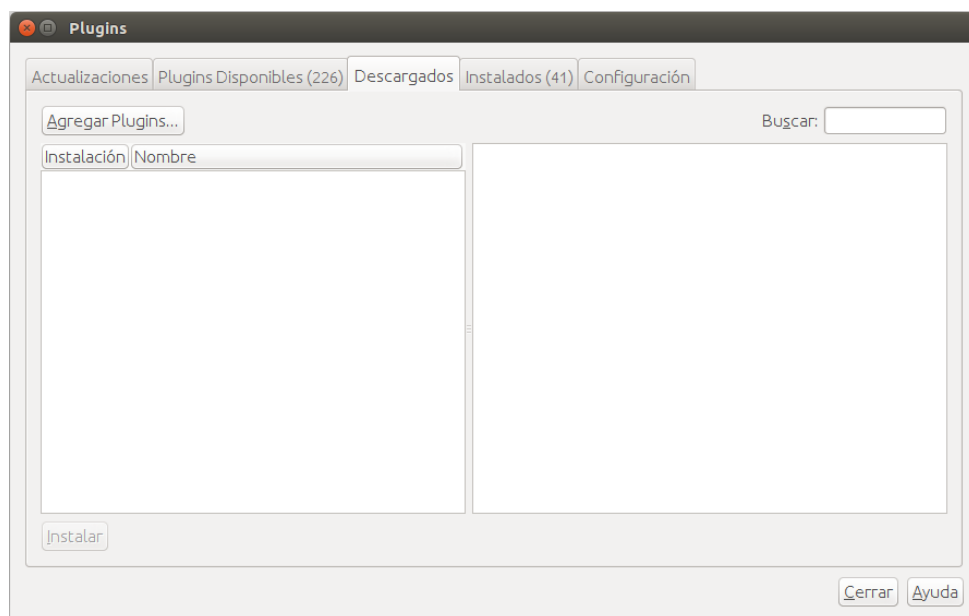


Figura A.5: Ventana de Plugins en NetBeans

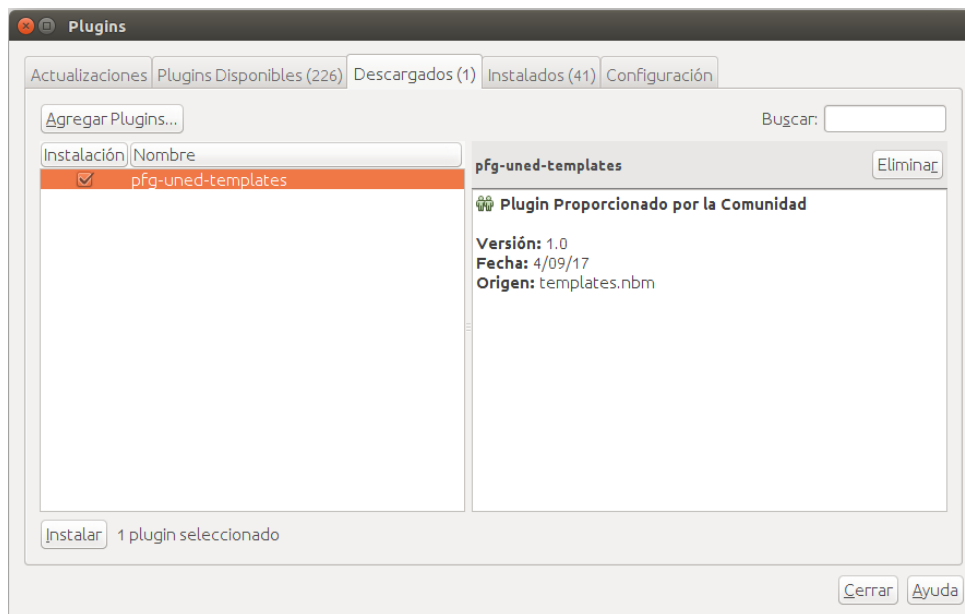


Figura A.6: Ventana de Plugins con uno nuevo

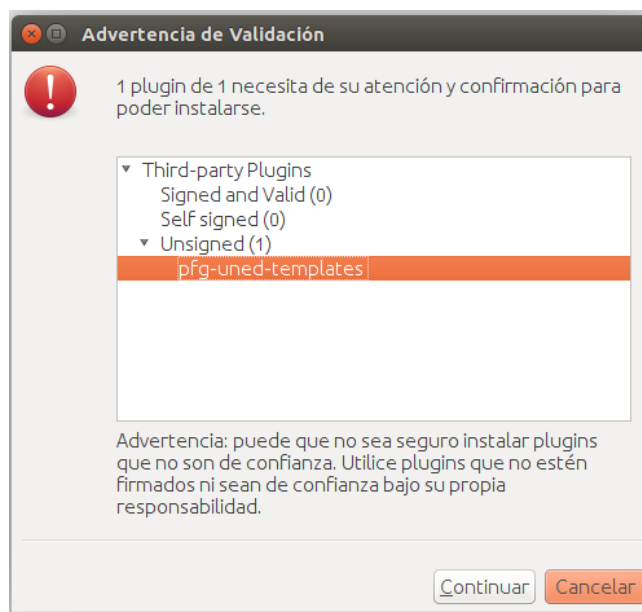


Figura A.7: Mensaje de aviso sobre la validación del módulo

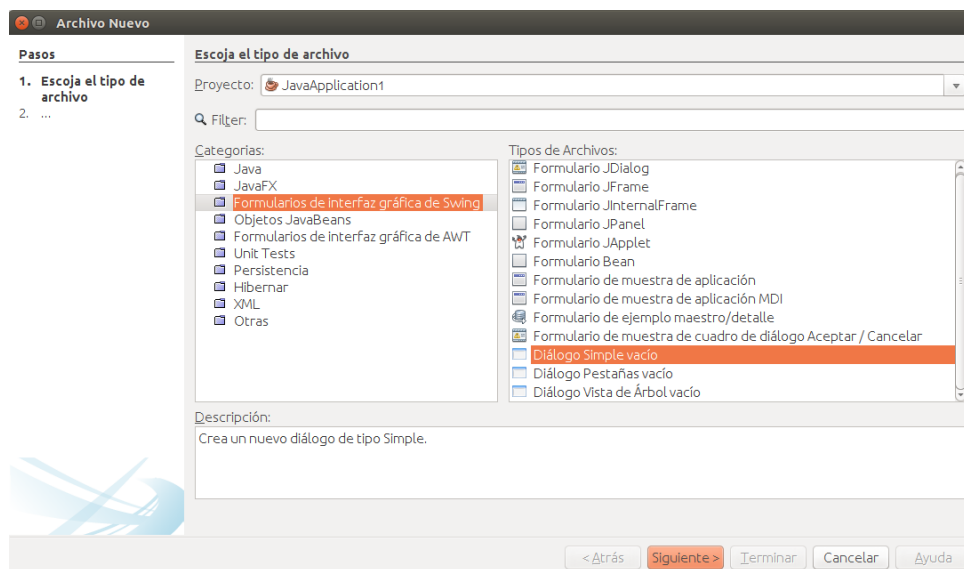


Figura A.8: Ventana de Archivo Nuevo en NetBeans

Basta con darle a Continuar para que la instalación del módulo finalice satisfactoriamente. En este momento, es recomendable, aunque no necesario, reiniciar el IDE, con el fin de asegurarse que todos los cambios se han aplicado correctamente.

Se puede comprobar, si miramos nuevamente en la ventana de Plugins, en la pestaña de Instalados, como el nuevo módulo ahora figura en la lista.

### A.2.1. Uso del módulo para la creación de interfaces gráficas

Una vez realizado el proceso de instalación, estamos en disposición de usar las plantillas. Para ello, la opción recomendada es seleccionar el paquete en el cuál queremos incluir el nuevo fichero, hacer clic derecho, y pulsar en 'Nuevo', y a continuación en 'Otro...'. Esto nos abrirá la ventana mostrada en la figura A.8. Una vez ahí, entramos en la categoría de 'Formularios de interfaz gráfica de Swing', puesto que es ahí donde se ha decidido incluir las plantillas. Dentro de dicha categoría veremos disponibles hasta 3 tipos de diálogos propios de la librería: *Diálogo <tipo> vacío*.

Simplemente deberemos seleccionar uno, y crear el fichero como si se tratase de cualquier otro archivo Java. Tras lo cual, nos aparecerá en la jerarquía de ficheros del proyecto como un nuevo archivo, abriéndose su vista de Diseño, como se muestra en la figura A.9.

**Aviso relativo a la ubicación de los nuevos ficheros** Es altamente recomendable que, al crear los diálogos nuevos, no se elija como ubicación la raíz de la jerarquía, sino que siempre se metan dentro de algún paquete o subpaquete de la misma. De lo contrario, se producirá un error debido a que NetBeans no puede leer el fichero de estructura creado, y por tanto no puede mostrar correctamente



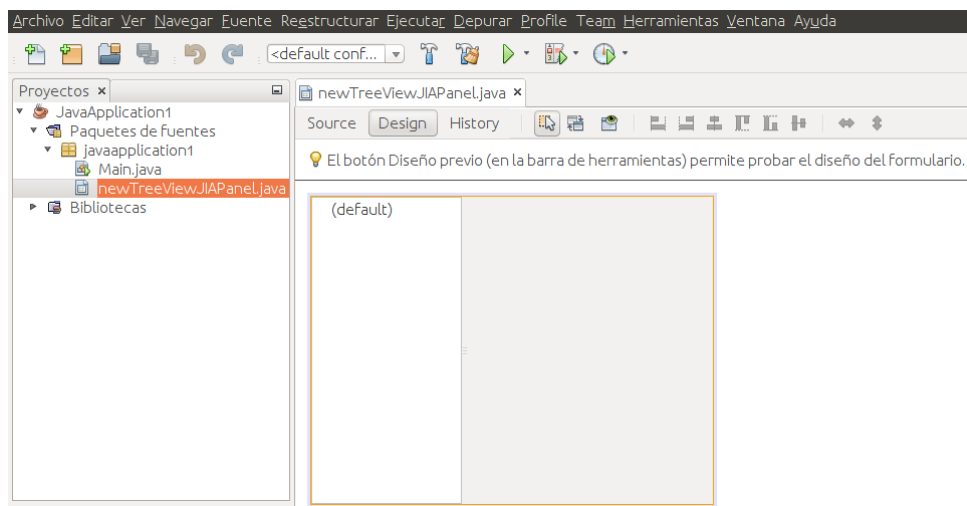


Figura A.9: Diálogo con Vista de Árbol creado con el template

el diálogo en la vista de Diseño, especialmente en el caso de los diálogos de pestañas y de árbol. En el caso del simple, al no tener propiedades ni componentes Swing internos, no se produce ningún error. Por tanto, si no se sigue esta recomendación, se debe tener en cuenta esta advertencia y asumir se que corre el riesgo de sufrir algún error.

### A.3. Importar los *JavaBeans* en la Paleta

El último paso, aunque opcional como el anterior, será el de hacer aparecer los JavaBeans de los tipos de diálogos en la Paleta de NetBeans. Para ello, una vez tenemos la librería importada en nuestro proyecto, tendremos que crear un diálogo cualquiera. Es válido tanto un diálogo propio de Swing, como uno de los mencionados anteriormente en los templates. Solo hace falta que se abra la vista de Diseño, para tener acceso a la Paleta. Una vez visible la Paleta, pinchamos con el clic derecho en cualquier zona dentro de la misma, y seleccionamos donde dice 'Administrador de paleta...', como se muestra en la figura A.10.

Esto nos abrirá el Administrador de Paleta que se muestra en la figura A.11. Una vez ahí, debemos seleccionar la primera opción en la parte superior derecha, que permite añadir componentes desde un archivo JAR. Pinchando en dicha opción, se abre otra ventana, donde debemos indicar el directorio donde tenemos ubicado el JAR de la librería, y al pulsar en siguiente, se nos muestra la lista de *beans* disponibles en ese JAR. Como vemos en la figura A.12, tenemos un total de 4, pero a nosotros nos interesan los 3 primeros, por lo que los seleccionamos, y avanzamos a la siguiente pantalla. Al final, nos dará la opción de elegir la categoría de la Paleta en la que queremos incluir el componente. En este caso, vamos a hacerlo en 'Contenedores Swing', puesto que nuestros diálogos son en realidad componentes de ese tipo.

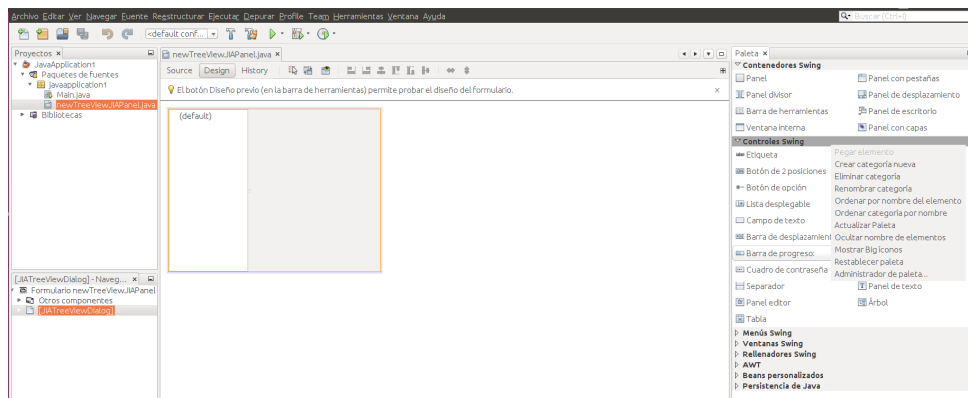


Figura A.10: Acceso al 'Administrador de paleta'



Figura A.11: Administrador de paleta de NetBeans

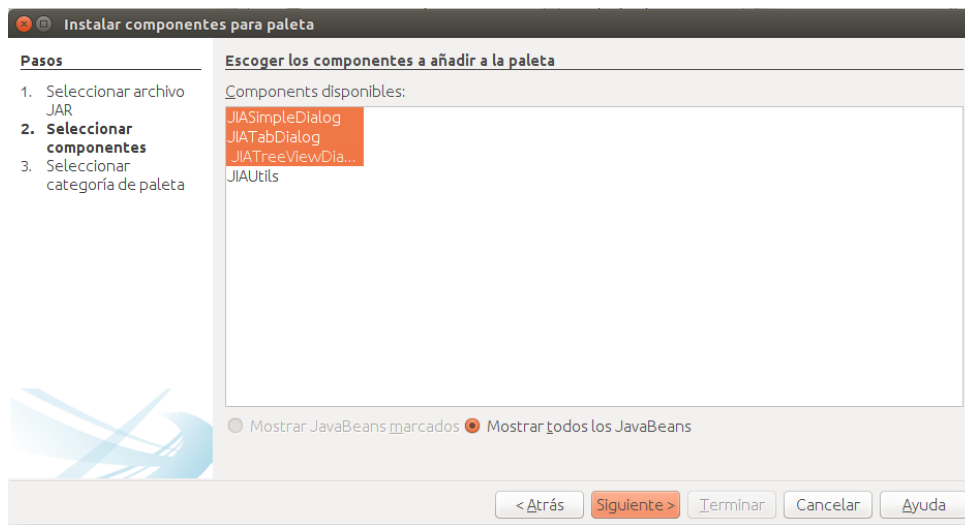
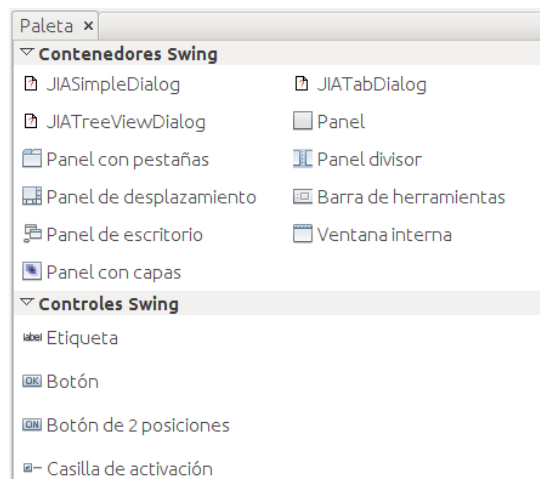


Figura A.12: Selección de beans a añadir a la paleta

Figura A.13: *JavaBeans* añadidos correctamente en la paleta

Una vez hecho todo el proceso, podremos ver que ahora los beans de nuestra librería se muestran en la Paleta del editor de interfaces gráficas de NetBeans (figura A.13), y para utilizar cualquiera de ellos, basta con arrastrarlos desde la Paleta, como un componente Swing más.

**Aviso relativo al uso de los *beans*** En este punto, se ha de hacer una advertencia sobre el uso de estos beans, a la hora de insertarlos dentro de otros diálogos de la librería. Debido a que no es la forma usual de introducir diálogos de la librería unos dentro de otros, se han de seguir unos pasos concretos, los cuales se indican claramente en el Manual de uso del diseñador, en el punto C.3.4. Se recomienda leer detenidamente dicho manual si no se desea que se produzcan errores inesperados durante el desarrollo del proyecto que haga uso de la librería.



## Anexo B

# Manual de uso del programador

Este manual se ocupa de explicar el papel del programador de aplicaciones que se dispone a usar la librería, así como reseñar las opciones que tiene disponibles y poner en contexto su función dentro del esquema general de uso de la librería.

Se comienza hablando del cometido que desempeña el programador dentro de un proyecto que haga uso de la librería, explicando sus atribuciones principales, así como las opciones básicas que tiene. A continuación, se comenta brevemente el uso que de los diálogos de la librería hace el programador, en comparación al uso que pueda hacer el diseñador de interfaces gráficas. Se señalan después las limitaciones o restricciones a las que se puede ver sometido el programador, debido a que aunque la librería es muy flexible y permite un uso transparente, existen ciertas acciones no permitidas, o no contempladas, que podrían provocar el fallo de la aplicación. Después de eso, se enumeran algunas opciones adicionales de las que dispone el programador, no contempladas en la idea original de desarrollo de la librería, pero añadidas a posteriori como ampliaciones de la misma. En último lugar, se muestran varios ejemplos de inserción usando la librería, así como su resultado, con el fin de ilustrar de forma más directa su puesta en práctica.

### B.1. Papel y objetivo de uso

En primera instancia, el rol del programador de aplicaciones es claro: **hacer uso de la librería para crear diálogos nuevos, combinando los ya existentes creados por el diseñador**. Ese es el principio en el que se basa todo el trabajo realizado, reutilizar interfaces gráficas de usuario.

Para ello, el programador tiene a su disposición la interfaz pública de la jerarquía de GUIs reusables de la librería, mostrada en la figura B.1, además de la interfaz pública de las GUIs normales propias de la librería gráfica usada, en este caso, Swing. Además, conoce otra información, como la relativa a de qué las clases de la librería derivan las GUIs diseñadas por el diseñador, y cómo invocar, si existe, el método de inicialización propio del diseñador para todo aquello no relacionado directamente con

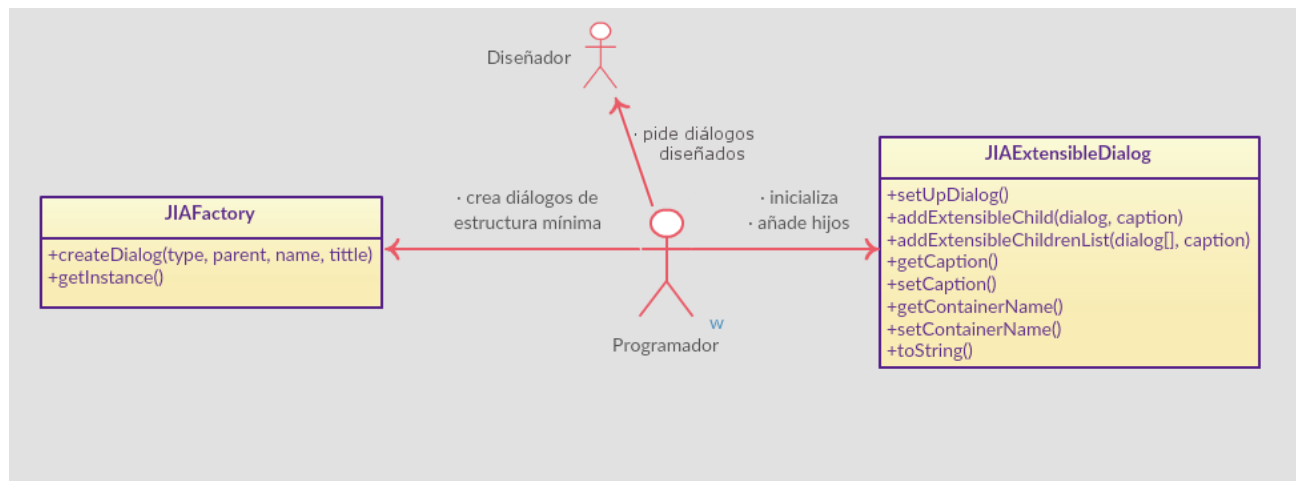


Figura B.1: Esquema básico del programador de aplicaciones

la librería, sino con el diálogo en sí mismo y sus componentes gráficos.

Su función encajaría dentro del desarrollo del *modelo* y de parte del *controlador*, ciñéndonos al patrón MVC (sección 2.3.1) que utiliza Swing, y que deberían seguir los proyectos que quieran hacer uso de la librería.

Se explican aquí, más detalladas, las funciones que aparecen en el diagrama:

- **JIAFactory:** es la factoría de diálogos vacíos.

- createDialog. Se utiliza para pedir a la factoría que genere y devuelva un diálogo de estructura mínima. El primer parámetro 'type' indica el tipo del diálogo que se desea generar (simple, pestañas o árbol), el tercer parámetro 'name' hace referencia al nombre interno que se le quiere asignar al diálogo. Los parámetros 'parent' y 'title', debido a que son ampliaciones realizadas a la librería, se comentarán con mayor detalle en la sección B.4 de este mismo anexo. Un diálogo de 'estructura mínima' es, básicamente, un diálogo completamente vacío, excepto porque dispone de los siguientes elementos:

- dos botones para 'aceptar' o 'rechazar' la edición realizada (acción atómica).
- un contenedor adecuado, con un *layout* asociado él.
- un componente de vista de árbol, en el caso de los diálogos de dicho tipo.

- getInstance. Utilizado para obtener una instancia de la factoría. Si se quieren invocar los métodos de la factoría y utilizar sus funcionalidades, antes se debe obtener su instancia mediante este método, debido a que se ha construido mediante un patrón *Singleton* (véase referencia a la sección 2.2.4).

- **JIAExtensibleDialog:** es la clase de diálogo padre para la jerarquía de tipos.

- setUpDialog. Permite realizar las inicializaciones propias del diálogo reusable.
- addExtensibleChild. Permite añadir un diálogo hijo pasado como parámetro al diálogo padre sobre el cuál es invocado.
- addExtensibleChildrenList. De una forma similar a la funcionalidad anterior, en este caso lo que se añade es una lista de diálogos hijos relacionados.
- getCaption y setCaption. Permiten obtener y establecer, respectivamente, el 'caption' del diálogo. Se amplía la información en la sección B.4.
- getContainerName y setContainerName. Permiten obtener y establecer, respectivamente, el 'containerName' del diálogo. A nivel interno de la librería, este containerName es exactamente el mismo campo que el parámetro 'name' del método `createDialog` de la factoría. Es decir, es el nombre interno que se usa para referenciar el objeto que representa el diálogo, en caso de ser necesario.
- toString. Utilizado para obtener una representación en forma de texto del diálogo. De forma predeterminada, se ha definido que devuelva el 'caption' del diálogo.

## B.2. Forma de interacción con los diálogos

En el caso del programador, su relación con los diálogos creados por la factoría será muy básica. Se ocupará de pedir diálogos vacíos, inicializarlos, y añadirles uno o más diálogos hijos, los cuáles serán diálogos contruidos por el diseñador, que tendrá a su entera disposición.

Referente a la interacción con los diálogos contruidos por el diseñador, también deberá inicializarlos antes de proceder a usarlos, para asegurar que funcionen de la manera esperada. Por otro lado, también tiene la posibilidad de añadir hijos dentro de aquellos creados por el diseñador, siempre que sean del tipo con Pestañas, o con Vista de árbol, pero no en los de tipo Simple. Esta limitación se ampliará en la sección B.3.

Estas tareas las realizará siempre desde la vista de Codificación del IDE, en este caso, NetBeans.

Por otro lado, conviene detallar con mayor precisión, aún sin entrar en detalles de implementación, las dos formas que ofrece la librería a la hora de insertar unos diálogos dentro de otros:

1. Método *addExtensibleChild*, recibe un único diálogo como parámetro.  
Permite añadir un diálogo de forma íntegra y holística, a otro diálogo, sin tener en cuenta los posibles hijos que a su vez pueda tener el diálogo que se está añadiendo.
2. Método *addExtensibleChildrenList*, recibe una lista de diálogos relacionados como parámetro.  
Permite añadir cada uno de los diálogos de la lista teniendo en cuenta sus potenciales hijos. De tal forma que, en caso de que alguno de los diálogos de la lista tenga hijos, éstos se añadirán

creando un nuevo nivel inferior de anidamiento. El proceso continúa de forma recursiva siempre que sigan existiendo diálogos que sigan teniendo más descendientes.

Esta decisión se basa en proporcionar más opciones al desarrollador a la hora de combinar los diálogos: por un lado, la primera vía no crea varios niveles de anidamiento, pero permite mostrar las partes Swing que ha creado el diseñador, y por otro lado, la segunda vía crea todos los niveles de anidamiento que existan en la jerarquía de la lista de diálogos relacionados, pero ignora las partes Swing de los diálogos cuando estos tienen hijos, centrándose solo en sus descendientes, que serán diálogos reusables de la librería.

Así, si queremos respetar el diseño original del diálogo, renunciando a tener varios niveles de anidamiento de la jerarquía que exista de diálogos reusables, podemos utilizar la primera opción para añadir un diálogo, y realizar varias invocaciones a la función en caso de querer añadir más. Y si queremos crear una jerarquía de niveles de anidamiento en base a las relaciones existentes entre los diálogos reusables, podemos utilizar la segunda alternativa, pasándole la lista de diálogos relacionados que queremos insertar, y, si solo quisiéramos insertar uno (porque de él descienda el resto de la jerarquía), siempre se podría mandar una lista de un único elemento. De esta forma, quedan cubiertas todas las opciones.

Existe una posible mejora en este apartado, referente a combinar ambas opciones, ofreciendo varios niveles de anidamiento y respetando a su vez el diseño original del diseñador. Se deja como posible ampliación de la librería en un futuro, sección 6.2 del último capítulo.

## **B.3. Limitaciones y sugerencias de uso**

En esta sección se detallarán tres limitaciones conocidas de la librería, que existen debido a su concepción y diseño, y los posibles problemas que éstas pueden originar si no se respetan y siguen las recomendaciones indicadas aquí.

### **B.3.1. Inserción en diálogos de tipo Simple creados por el diseñador**

Por un lado, tenemos una restricción relacionada con la inserción de diálogos por parte del programador. Para el programador, los diálogos construidos por el diseñador serán como una caja negra, y los tratará como tal, sin adentrarse en los pormenores de su implementación. Debido a ello, no deberá insertar otros diálogos, utilizando los métodos de adición explicados anteriormente en la sección B.2, dentro de aquellos diálogos del diseñador que sean del tipo Simple.

Esto sucede porque el programador, al desconocer el *layout* que haya podido aplicar el diseñador a su diálogo, no puede asegurar el correcto funcionamiento de ese diálogo Simple tras intentar insertar uno externo, debido a la implementación interna de dichas funcionalidades de adición comentadas,



que hacen un uso específico de *layouts* preestablecidos, los cuales pueden diferir de los utilizados por el diseñador.

En los diálogos creados por el diseñador que sean de tipo Pestañas o Vista de árbol, por el contrario, no se tiene esta limitación, y el programador puede añadir los diálogos que desee, y de la forma que desee. En este caso, los diálogos añadidos como hijos serán nuevas pestañas, cuando tengamos un diálogo del tipo Pestañas, o bien serán nuevas entradas en la vista de árbol, si se trata de un diálogo de dicho tipo. Como las pestañas o las entradas del árbol son independientes de los *layouts* asociados, no se genera ningún tipo de conflicto que pueda derivar en posibles problemas o fallos.

### B.3.2. Creación de diálogos vacíos ajenos a la factoría

Esta limitación tiene que ver con que el programador no debería crear por si mismo diálogos vacíos, sin botones, no generados por la factoría, si no quiere que falle el programa. Se debería limitar a pedir GUIs de estructura mínima a la factoría, y rellenarlas con los diálogos que haya construido el diseñador:

El origen de este problema viene por la propia naturaleza de los diálogos que genera la librería a través de su factoría, los cuales tienen una serie de propiedades concretas, y métodos asociados, que les habilitan y hacen compatibles con las funciones propia que ofrece la librería. De cualquier otra forma, teniendo diálogos vacíos creados *ad hoc*, e incluso si estos fueran de un tipo compatible con la librería, es decir, heredasen de la clase `JIAExtensibleDialog`, no funcionarían de la manera adecuada y esperada, debido a lo comentado anteriormente.

Quedarían, por así decirlo, excluidos de la compatibilidad con la librería, aunque dichos diálogos fuesen perfectamente válidos como componentes Swing en si mismos. Por tanto, se perdería la posibilidad de combinarlos con otros diálogos reusables, la posibilidad de enviar y recibir mensajes, las acciones recursivas de validación, salvado y limpieza de diálogos, y los servicios de redimensionamiento que ofrece la librería a la hora de añadir componentes reusables unos dentro de otros.

### B.3.3. Error al exceder el número de hijos

En el caso de los diálogos simples, ya desde su definición, el número máximo de hijos que pueden contener está limitado a 2, y la disposición que se ha elegido para ellos es vertical, es decir, uno está encima y el otro debajo, ambos por encima de los botones de aceptar y rechazar la edición.

Basándose en esto, un diálogo simple puede estar vacío, sin hijos y solo con los botones de edición, tener un solo hijo, o bien tener 2 hijos, siendo el primer diálogo el superior y el segundo el inferior. Lo que no permite la librería, sin embargo, es añadir un tercer hijo, ni a los lados, ni por encima, ni por debajo, de dos ya existentes, puesto que de entrada el método de adición no lo contempla.

```
Exception in thread "main" java.lang.RuntimeException:  
- Un diálogo Simple no puede tener más de 2 hijos.  
- Error tratando de añadir los diálogos, falta(n) 1 espacio(s).
```

Figura B.2: Excepción al añadir un 3er diálogo a uno de tipo Simple

En caso de tener ya 2 hijos en un diálogo Simple e intentar añadir un tercero, o bien intentar insertar directamente una lista con 3 diálogos, usando el método de adición de listas de diálogos relacionadas, la librería lanzaría una excepción en tiempo de ejecución del tipo *RuntimeException*, común de Java, con el mensaje mostrado en la figura B.2. Después de eso, el diálogo se mostraría igualmente y se permitiría que continuase la ejecución, pero el diálogo mostrado estaría vacío. Se hace así, para tener una forma más visual (con un diálogo vacío, cuando se está queriendo añadir algo en él), de comunicar el error, además de la propia excepción que aparecería por consola, y se podría incluso capturar y tratar si se quisiera.

Inicialmente, esta decisión de diseño se tomó con el fin de evitar que los diálogos se saliesen de la pantalla tras la inserción y el redimensionamiento implícitos, porque bastaría con que los diálogos tuviesen un tamaño medio o grande, para que, al añadir el tercero, ocupase totalmente, o incluso sobrepasase, la altura de resolución admitida en la mayoría de monitores convencionales.

## B.4. Ampliaciones disponibles

Como parte de la implementación de la librería, además de las características ideadas inicialmente en su concepción, se han añadido ciertas ampliaciones o extensiones, que la dotan de mayores funcionalidades y más opciones de control de cara a sus usuarios. En esta sección se comentarán brevemente aquellas que afectan directamente al programador, y que éste puede usar y aprovechar si así lo desea, pero siempre de forma opcional.

### ■ Personalización de la ventana

Se ha añadido un parámetro que originalmente no existía, en el método *createDialog* de la factoría. Se trata de 'parent', el cual representa un objeto de tipo *JFrame*, el cuál puede ser pasado al método con el fin de asignar una ventana personalizada para contener el diálogo principal de la jerarquía, es decir, el de más alto nivel.

El uso que de este elemento de tipo ventana hace la factoría, puede ejemplificarse en el código B.1. En él, se aprecia como, si el parámetro no es nulo, se utiliza como envoltura, y en caso contrario, se crea una ventana con unas propiedades predeterminadas. El programador tiene total libertad de utilizar o no esta extensión, y en caso de no querer usarla, bastaría con que en el lugar de dicho parámetro, se pasase un valor nulo, como se ha visto.

## Listado de código B.1: Código para el parámetro 'parent'

```
1
2 // Si 'parent' es distinto de null, se envuelve usando
3 // la ventana pasada como parámetro. Si no se proporciona
4 // ninguna ventana, se crea y utiliza una predeterminada.
5
6 JFrame wrapper = parent != null ? parent : new JFrame();
```

## Listado de código B.2: Código para el parámetro 'title'

```
1
2 // Se asigna el título de la ventana, mirando primero el
3 // parámetro 'title'. A continuación, el title del JFrame,
4 // y por último, el nombre de clase del diálogo, por defecto.
5
6 wrapper.setTitle(validateString(title) ? title :
7     validateString(wrapper.getTitle()) ? wrapper.getTitle()
8     : dialog.getClass().getSimpleName());
9
10 // NOTA: El método validateString() simplemente comprueba
11 // si la cadena es nula o está vacía, devolviendo false en
12 // ese caso, debido a que no es una cadena válida, y true en
13 // caso contrario, si la cadena no es nula, y no está vacía
```

Se tomó la decisión de incluir este comportamiento con el fin de dotar de mayor flexibilidad al programador a la hora de usar la librería, pudiendo utilizar sus propias ventanas personalizadas, creadas por él mismo, o incluso por el diseñador.

#### ■ Selección del título para la ventana

De la misma forma, se habilitó un nuevo parámetro 'title', de tipo String, en el método *createDialog* mencionado anteriormente, que permite pasar al método de creación un título para la ventana del diálogo principal. Podemos ver su lógica en el código B.2.

Como se puede apreciar, en primer lugar tiene preferencia el 'title' mencionado aquí, que se pasa como parámetro, y en caso de ser *válido*, se aplica como título a la ventana (wrapper). En los comentarios del código se puede apreciar esa definición de "válido".

En segundo lugar, se comprueba el título que ya pudiera tener la ventana en si misma. En el caso de la predeterminada de la factoría, el título estará vacío, pero si el programador, utilizando el parámetro 'parent' comentado anteriormente, ha decidido pasar una ventana personalizada a la factoría, y ésta tuviese un título válido, se conservaría.

Por último, en caso de no obtener un título adecuado de ninguna de las dos fuentes anteriores, se recurriría al nombre por defecto como título, que sería el nombre de la clase del diálogo principal que se va a insertar en esa ventana.

#### ■ Uso de los 'captions' según el tipo de diálogo

Se ha ampliado la interfaz pública de la librería, añadiendo un campo 'caption', de tipo String, para la clase *JIAExtensibleDialog*, es decir, los diálogos reusables de la librería lo heredan y pueden utilizarlo.

Dicho campo tiene diferentes usos según el tipo de diálogo para el que lo estemos evaluando, y se puede usar de dos formas diferentes, como parámetro en los métodos de inserción de diálogos, o como campo asignable y recuperable de cualquier diálogo reusable. El concepto como parámetro ya existía, y se amplió como campo, manteniendo el nombre, pues la finalidad es prácticamente idéntica. Como en ciertos escenarios puede llevar a una cierta confusión, se detalla minuciosamente su uso:

- En los diálogos de tipo Simple, dicho campo no se utiliza, pues no tiene cabida. Ni como parámetro ni como campo.
- En los diálogos de tipo Pestañas, el 'caption' como parámetro, se utiliza como el título para la pestaña en la que se está añadiendo el diálogo hijo, en el caso del método de inserción de un solo diálogo, o bien como título para la pestaña que sirve de contenedor para todas las demás pestañas, una para cada diálogo hijo de la lista, en el caso del método de inserción de una lista de diálogos relacionados. También se utiliza como campo, en el caso del método de inserción de una lista de diálogos, como título de la pestaña de cada uno de los diálogos de la lista.

- En los diálogos de tipo Vista de árbol, el 'caption' se utiliza de una forma muy parecida al caso anterior. Es decir, como parámetro, sería el nombre de la entrada, en la vista de árbol, del diálogo hijo (método inserción diálogo único), o bien sería el nombre de la entrada que engloba el resto de entradas, una por cada diálogo de la lista (método inserción lista de diálogos). Por otro lado, como campo, al igual que antes, en el caso de la inserción de la lista de diálogos, serviría como nombre para cada una de las entradas de cada uno de los diálogos de la lista.
- **Asignación para el nombre de la raíz de la vista de árbol**

Se ha añadido un campo 'treeRootName' a la clase JIATreeViewDialog, ampliando así su interfaz pública con métodos accesorios y modificadores. Es accesible como una propiedad mutable para el componente que representa dicho diálogo, a la hora de diseñarlo, y tanto el programador como el diseñador puede editar su valor. Su propósito es permitir cambiar el nombre que aparece como la raíz de la vista de árbol para los diálogos de ese tipo. Dicha raíz siempre existe, y se muestra, por lo que en caso de no asignarle un valor, tendrá uno por defecto, el nombre de la clase del diálogo.

La idea es la siguiente: el diseñador, desde la vista de Diseño, podrá editar el valor como una propiedad del componente del diálogo. Más tarde, cuando el programador instancie dicho diálogo y lo inicialice, se asignará el valor de la propiedad elegido por el diseñador, como nombre de la raíz. Opcionalmente, el programador tendrá la opción de, empleando el método modificador (setTreeRootName), volver a modificar ese valor por uno de su elección.

## B.5. Ejemplos de utilización

Con el fin de explicar más detallada y gráficamente los casos de **inserción de diálogos**, y de que el lector los pueda comprender con mayor facilidad, esta sección tratará de mostrar de forma visual los 2 métodos de inserción de diálogos, para los 3 casos de diálogo diferente que tenemos según su tipo. Se realizará a través de unos bosquejos sencillos, meramente ilustrativos, pero no técnicos, de cómo quedaría el resultado de aplicar las inserciones.

### B.5.1. Diálogos simples

En primer lugar se mostrará el caso de los diálogos simples. Se muestra en la figura B.3, el diálogo en su estado inicial recién generado por la factoría, el cuál estará vacío.

- **Método de inserción de diálogo único**

Si procedemos a insertar un diálogo en dicho diálogo vacío, obtenemos el resultado mostrado en la figura B.4. Si ya hubiera un hijo anterior, este sería el segundo, y se colocaría debajo.

- **Método de inserción de lista de diálogos**

Por otro lado, si lo que insertamos es una lista de hijos, de 2 en este caso, sobre el diálogo vacío inicial, tendríamos lo que se aprecia en la figura B.5. Como se puede apreciar, para el caso del diálogo simple, la inserción de dos diálogos en lista es equivalente a la inserción secuencial de dos diálogos sencillos, de forma individual. No será así para el resto de tipos de diálogo.

### **B.5.2. Diálogos de pestañas**

Como antes, se comienza mostrando un diálogo de pestañas vacío, el de la figura B.6. En este caso, su tamaño es mayor que el simple, pese a estar igualmente vacío. Esto es debido al espacio que representa el panel de pestañas, aún cuando todavía no tiene ninguna.

- **Método de inserción de diálogo único**

Si se añade un diálogo con inserción sencilla, pasaremos al escenario de la figura B.7.

- **Método de inserción de lista de diálogos**

Por otro lado, si insertamos una lista de hijos, en este caso, como ya se ha comentado anteriormente, se crea igualmente una única pestaña, pero dentro de ella se inserta un nuevo panel de pestañas, con una pestaña anidada por cada diálogo de la lista, y, si estos tuvieran a su vez hijos, se crearían más niveles de anidamiento, creando la jerarquía completa de diálogos reusables que existiese. El resultado se muestra en la figura B.8.

### **B.5.3. Diálogos con vista de árbol**

Empezamos mostrando el diálogo vacío, como en los casos anteriores. Figura B.9.

- **Método de inserción de diálogo único**

Insertamos un único hijo, creando una sola entrada en la vista, como se ve en la figura B.10.

- **Método de inserción de lista de diálogos**

Por último, existe la opción de insertar directamente una lista, siendo el comportamiento similar al del caso de las pestañas. Se crea una entrada de nivel superior, y dentro, anidadas, tantas entradas como diálogos tenga la lista. Mostrado en la figura B.11.



Figura B.3: Diálogo simple vacío



Figura B.4: Diálogo simple con un hijo



Figura B.5: Diálogo simple con dos hijos

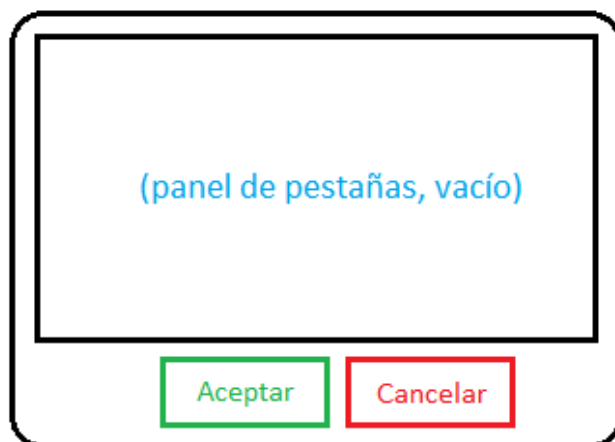


Figura B.6: Diálogo de pestañas vacío



Figura B.7: Diálogo de pestañas con inserción sencilla



Figura B.8: Diálogo de pestañas con inserción de lista



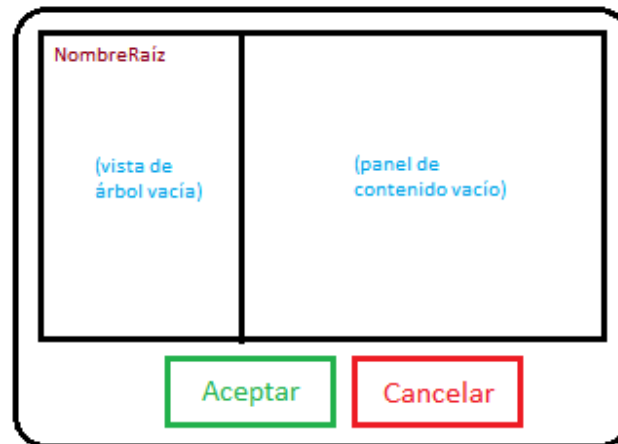


Figura B.9: Diálogo con vista de árbol vacío



Figura B.10: Diálogo con vista de árbol con inserción sencilla



Figura B.11: Diálogo con vista de árbol con inserción de lista



## Anexo C

# Manual de uso del diseñador

De la misma forma que se hizo para el programador, este manual se ocupa de desarrollar la función que cumple el diseñador de interfaces gráficas de usuario que pretende hacer uso de la librería, recalcar las opciones que le ofrece y mostrar una visión tanto global como específica dentro de su situación en el desarrollo de un proyecto que haga uso de la librería.

Se empieza comentando su papel dentro del contexto general, así como sus responsabilidades y alternativas disponibles. Después, se pasa a hablar sobre su relación con los diálogos reusables, en contraposición al uso que de ellos hace el programador, puesto que ambos manejan las mismas entidades, pero desde diferentes puntos de vista. Se comenta el buen y mal uso que puede hacer de la librería, en referencia a las acciones permitidas, y aquellas que pueden provocar fallos (y cómo solucionarlos). A continuación, se reseñan de forma breve alguna de las ampliaciones ya comentadas anteriormente para el programador, pero ofreciendo esta vez una perspectiva diferente, como es el caso del diseñador de diálogos. Por último, se muestran los dos escenarios de uso más generales a los que va a tener acceso el diseñador en su uso normal de la librería: la creación de nuevos diálogos usando *templates*, y la inserción de diálogos desde la Paleta de NetBeans, a través de los *JavaBeans*.

### C.1. Papel y objetivo de uso

Centrándonos en su uso fundamental, el rol del diseñador de interfaces gráficas de usuario (GUIs) es el siguiente: **crear y diseñar sus diálogos de forma que sean reusables, y tengan compatibilidad con las funciones que ofrece la librería**. Dicho requisito no supone un gran compromiso por su parte, puesto que la librería está pensada para un uso transparente, pero si es conveniente seguir unas pocas directrices para asegurar esa compatibilidad.

El contexto en el que se podría encajar al diseñador sería el mostrado en la figura C.1. Se puede apreciar como su interacción no es directamente con la librería, sino con el propio IDE, y su herramienta de construcción de GUIs. Es decir, se podría decir que su función va a ser trabajar



Figura C.1: Esquema básico del diseñador de interfaces gráficas

casi exclusivamente desde la vista de Diseño del IDE NetBeans. Por tanto, dentro del patrón MVC (sección 2.3.1), se puede cuadrar dentro de la construcción de la parte *vista* en su totalidad, aunque también puede desempeñar alguna función dentro de la capa del *controlador*.

Siguiendo la figura anterior, se pueden apreciar hasta 4 cometidos claramente separados:

- Crear nuevas GUIs, utilizando para ello plantillas accesibles desde el IDE, que se habrán puesto a disposición del diseñador durante la instalación del entorno, sección A.2. Estas plantillas, como se verá, extienden siempre de alguno de los diálogos reusables de la librería, por lo que los diálogos creados a partir de ellas, pertenecerán a esa misma jerarquía.
- Arrastrar componentes desde la Paleta de la herramienta de construcción y diseño de GUIs. El diseñador, una vez tiene el diálogo principal, que actúa como contenedor, podrá comenzar a arrastrar componentes desde la Paleta hacia el contenedor, personalizándolos a su gusto, y modificando sus propiedades y apariencia. Entre los componentes que puede seleccionar de la Paleta, estarán los propios diálogos reusables de la librería.
- Implementar ciertos métodos que debe redefinir en los diálogos reusables, con el fin de conseguir la citada compatibilidad con la librería. Dichos métodos se especificarán más adelante, en la sub-sección C.3.3. De no implementarlos, la aplicación será propensa a fallos.
- Por último, una vez ha creado los diálogos, una vez los ha rellenado con los componentes que ha elegido, de la forma que ha considerado oportuna, y redefinido y por tanto implementado

los métodos adecuados, habrá finalizado su labor, que es la de proporcionarle al programador diálogos ya contruídos o diseñados, que él pueda reutilizar y combinar.

Su tarea se limitará esencialmente a construir ventanas o diálogos, y rellenarlos de la forma que considere oportuna, arrastrando componentes desde la Paleta. El resultado de su trabajo deben ser interfaces de usuario amigables e intuitivas, que hagan al usuario sencilla la utilización del programa.

## C.2. Forma de interacción con los diálogos

Para el diseñador, los diálogos serán su cometido principal. Él será el encargado de crearlos, construirlos, darles la forma, comportamiento y apariencia adecuada, y después, dejarlos a disposición del desarrollador de la aplicación.

Los diálogos que diseñe no deberán tener botones de edición, para evitar el problema de que el programador tenga que decidir qué hacer con ellos a la hora de integrarlos en los diálogos de la factoría. Estos últimos, los de la factoría, serán los únicos que dispongan de botones de edición para aceptar o rechazar las modificaciones realizadas.

Desde el punto de vista del diseñador de GUIs, la librería debe permitir que sus distintos tipos de diálogo estén a disposición suya desde la vista de Diseño, de forma transparente. Así, el diseñador se limita a realizar una secuencia de los siguientes pasos:

1. Creación del diálogo principal para la ventana.

Para ello tiene dos opciones. O bien crea un nuevo diálogo utilizando una plantilla, o bien crea una nueva ventana vacía, sin ni siquiera un contenedor asociado, y arrastra el diálogo reusable completamente vacío desde la Paleta, empezando a modificarlo desde cero. O bien puede utilizar un *template* de los que tienes disponibles, eligiendo el tipo de diálogo que quiere crear, y dándole un nombre adecuado.

2. Modificación y personalización del mismo.

Una vez tiene creado el diálogo, puede comenzar a editarlo, y para ello, lo puede tratar de forma completamente transparente, como si se tratase de un componente de Swing más. Salvo quizá, por unos pocos métodos, comentados anteriormente, que tendrá que implementar para conseguir que sean diálogos compatibles con el patrón de reusabilidad, y con el resto de jerarquía de la librería.

En resumen, se puede deducir que el uso de la librería es muy simple para el diseñador, que se tiene que ceñir a conocer unos pocos detalles necesarios de la librería, con el fin de ajustarse a sus características, y el resto del trabajo, lo puede desempeñar como si estuviera diseñando componentes gráficos de Swing normales y corrientes.

## C.3. Limitaciones y sugerencias de uso

Una vez se ha planteado el rol que juega el diseñador, y explicado de forma breve su flujo de desarrollo a lo largo de un proyecto que haga uso de la librería, llega el momento de señalar las restricciones a las que se ve sometido, con el fin de adaptarse, mínimamente eso sí, a la librería, y que todo funcione correctamente.

### C.3.1. Variedad de tipos de diálogos disponibles

El diseñador, a la hora de definir el diálogo principal de la ventana de la aplicación, solo debería usar diálogos propios de la librería, es decir, diálogos reusables, no los propios de Swing. De otra forma, los diálogos que construyese quedarían excluidos, por así decirlo, de la librería, no formando parte, o no siendo compatibles, con el mecanismo de reutilización. Por tanto, no podría implementar sus métodos, ni tampoco aprovechar las funcionalidades que ofrece la librería.

Es decir, tiene a su disposición 3 tipos de diálogos base reusables, que puede modificar a su gusto.

- El diálogo simple, el más sencillo de todos, que no ofrece ninguna característica especial.
- El diálogo con pestañas, que permite el anidamiento de otros componentes, separados por pestañas, en un panel interno de pestañas. De tal forma que cada pestaña tendrá su propio contenido independiente, y al pulsar en cada una de ellas, se mostrará.
- El diálogo con vista de árbol, el cual tendrá dos partes claramente diferenciadas. Por un lado, a la izquierda tendrá una vista jerárquica de todo el contenido del diálogo (llamada vista de árbol), con entradas que se pueden distinguir según sus niveles de anidamiento, y, pinchando sobre cualquiera de las entradas de dicha vista y seleccionándola, se hará visible su contenido. Por otro lado, a la derecha tendrá el panel de contenido, por llamarlo de alguna manera, que es el lugar en el que se mostrarán los componentes comprendidos dentro de cada una de las entradas, cuando ésta sea seleccionada.

En caso de que el diseñador quisiese usar cualquier otro tipo de diálogo como principal, fuera de los 3 citados anteriormente, se debería realizar una ampliación de la librería, incluyendo el nuevo tipo deseado, y haciendo que dicho nuevo tipo de diálogo extendiese del diálogo raíz de la jerarquía: `JIAExtensibleDialog`, de tal forma que heredase todas sus propiedades y funcionalidades, permitiendo a mayores definir las suyas propias, según las necesidades del diseñador.

Por otro lado, como componentes propiamente internos de dicho diálogo principal, se puede encontrar cualquiera nativo de Swing, exceptuando aquellos que por defecto no son anidables, como los `JFrame`, `JDialog`, y cualquier ventana en general. E incluso también, se podrán utilizar a nivel interno los diálogos reusables de la librería, de cualquiera de los 3 tipos mencionados, realizando unos pequeños ajustes necesarios para su adecuada puesta en marcha, que se verán en el punto C.3.4.

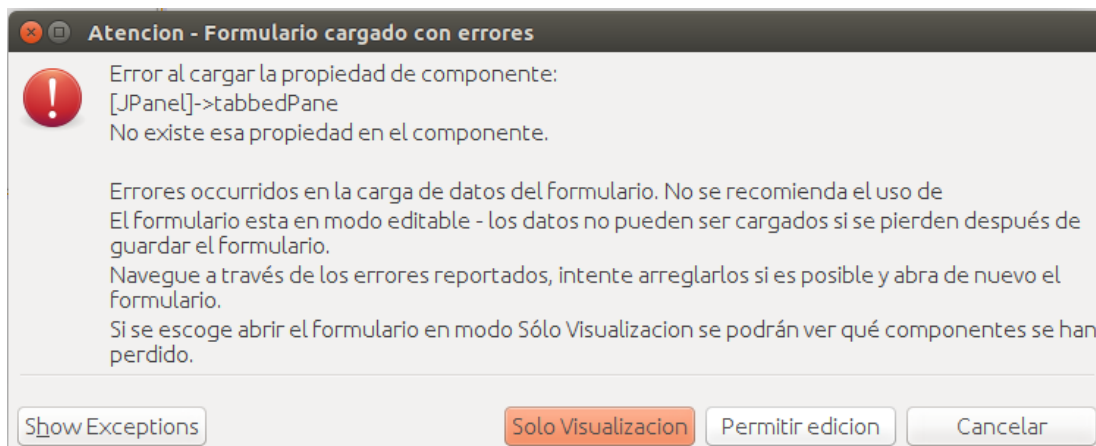


Figura C.2: Error producido al crear el diálogo en la raíz de paquetes

### C.3.2. Creación de diálogos a partir de plantillas sin paquete asociado

El diseñador, a la hora de utilizar los *templates* o plantillas que tiene disponibles, debe tener en cuenta una recomendación asociada a la ubicación de los nuevos ficheros de los diálogos que pretende crear. Si el diálogo que está creando es a partir de la plantilla para un diálogo simple, no pasará nada, puesto que dicho diálogo está vacío, solo tiene un contenedor, pero ningún componente interno ni propiedades asociadas.

Sin embargo, si los diálogos que está intentando crear son con plantillas del tipo 'diálogo con pestañas' o 'diálogo con vista de árbol', debe tener cuidado, pues estos diálogos tienen componentes internos expuestos como propiedades del diálogo, que pueden no cargarse correctamente si la ubicación del nuevo archivo es la del directorio raíz. Es decir, si el diálogo no tiene un paquete, de cualquier nivel, asociado. Si ese es el caso, al intentar abrir el diálogo en la vista de Diseño se mostrará un error como el de la figura C.2.

Dicho error se puede subsanar eliminando el diálogo y creándolo nuevamente en un paquete adecuado, o bien moviendo el diálogo actual desde el paquete raíz a un paquete interno seleccionado. Se pueden emplear para dicho fin las herramientas que ofrece el propio IDE, como por ejemplo, en el caso de NetBeans (y en la mayoría de IDEs), las herramientas de copiar y pegar archivos.

Además de ello, sería bueno no olvidar que debemos declarar el nuevo paquete dentro de la propia clase que representa el diálogo, utilizando la notación propia de Java: `package nombre_del_paquete;`, o de lo contrario, el compilador mostrará un error avisando de que el diálogo se encuentra en un paquete distinto al que tiene declarado. Recordemos que un archivo ubicado en el paquete raíz no tendrá ninguna declaración de paquete establecida en su clase.

De hecho, esta limitación tiene mucho que ver con la JLS, la cual recomienda no utilizar el paquete por defecto (la raíz) como ubicación para las clases, puesto que las clases o tipos declarados no serán accesibles. Así reza en Oracle (2005):

```
Exception in thread "main" java.lang.RuntimeException: Se debe redefinir la implementación de este método [validateThis].
```

Figura C.3: Excepción que se lanza fruto de la implementación trivial

*"It is a compile time error to import a type from the unnamed package."*

*Es un error en tiempo de compilación importar un tipo desde un paquete sin nombre.*

Es decir, las clases ubicadas en el paquete por defecto no serán directamente accesibles mediante la realización de un *import*, debido a que no existiría un nombre de paquete asociado para el que realizar dicho *import*.

### C.3.3. Implementación de los métodos de compatibilidad con la librería

Como se ha comentado previamente, todos los diálogos de la jerarquía de diálogos reusables de la librería tienen ciertos métodos, que son los que, además de otras cosas, dotan a los diálogos de compatibilidad con el mecanismo de reusabilidad.

Una vez que se ha diseñado el cuadro de diálogo, es labor del diseñador redefinir y darles una implementación a dichos métodos. De lo contrario, dichos métodos tienen ya una implementación trivial por defecto, consistente en lanzar una excepción para avisar precisamente de la necesidad de redefinirlos. La excepción es similar para todos, cambiando únicamente el nombre del método que falta por redefinir, como se muestra en la figura C.3.

Los métodos a redefinir son los siguientes:

- **validateThis()**

Se ocupa de la validación de los campos para el diálogo concreto para el que se implementa, y solo para ese. Si la validación tiene éxito, se ejecuta el salvado y limpieza de datos.

- **saveThis()**

Se ocupa de actualizar los campos de la clase editada a partir de los del diálogo.

- **cleanThis()**

Realiza tareas de finalización, limpieza, cierre de ficheros, etc, en caso de ser necesario.

- **getExternVal()**

Permite al diálogo recibir un mensaje concreto, asociado a un identificador, y con un objeto que representa un valor. Utilizado para el paso de mensajes.

Los 3 primeros métodos, para validar, salvar y limpiar los diálogos, forman en conjunto un sistema para cada diálogo, y a su vez, forman parte de un mecanismo recursivo que dispara la librería, representada en la figura C.4.



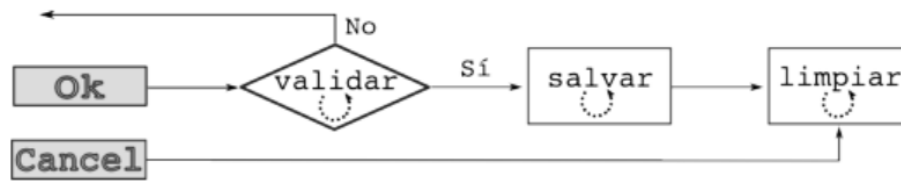


Figura C.4: Mecanismo recursivo lanzado por la librería

Listado de código C.1: Código del método 'cambiaVal'

```

1
2  protected void cambiaVal(final String id, final Object value) {
3      if(getListaListeners() != null)
4          getListaListeners().forEach((listener) -> { listener.
              getExternVal(id, value); });
5  }

```

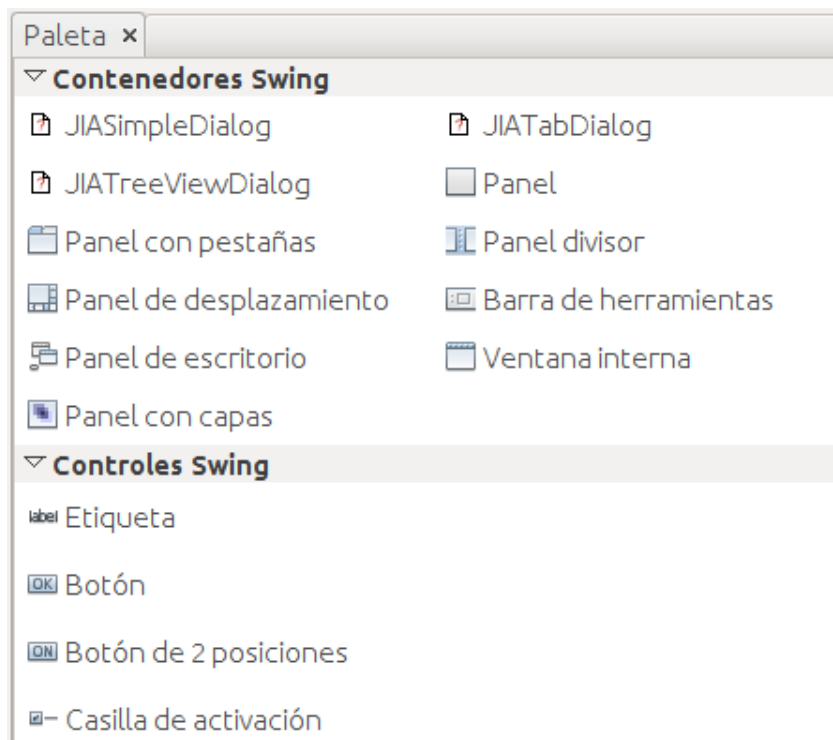


Figura C.5: Beans disponibles en la Paleta de NetBeans

Dicho mecanismo se activa cada vez que el usuario pulsa en uno de los botones de 'Aceptar' o 'Cancelar' de un diálogo principal. El flujo comienza desde ese diálogo principal, de nivel superior, el cuál va invocando recursivamente a todos sus hijos, y estos a su vez a los suyos, hasta llegar al final de la jerarquía de descendencia. Esta recursividad se produce para las 3 acciones por separado.

En el caso del botón de 'Aceptar', si la validación tiene éxito, se procede a ejecutar las otras dos acciones sucesivas, cada una también de forma recursiva para la jerarquía de diálogos. Por otro lado, para el botón de 'Cancelar', solo se ejecuta la acción de limpieza, sin validación ni guardado de los datos modificados en cada diálogo. Este proceso se produce de forma atómica para el usuario, es decir, a partir de una sola pulsación, o bien se produce la validación, salvado y limpieza, o bien no se realiza ninguna acción, en caso de un fracaso en la validación. Pero siempre en un solo paso.

Por otro lado, está el método de recepción de un mensaje. Si el diseñador desea que algún campo o alguna propiedad de su diálogo, o de los datos a editar, se pueda cambiar desde el exterior, debe redefinir el método de recepción de mensajes, de tal forma que se adapte a sus variables. Es decir, que se reciba un identificador esperado y conocido por todos los usuarios de la librería, y que el valor sea del tipo adecuado a la propiedad que se quiere modificar externamente.

Si además, también desea que su diálogo emita un valor para posibles observadores, puede emplear un método que ya implementa la clase `JIAExtensibleDialog`, y por tanto todos los diálogos reusables, conocido como 'cambiaVal', cuya implementación se muestra en el código C.1. Bastará con invocar dicho método con el identificador y el valor deseado, sabiendo que existe algún otro diálogo esperando esos parámetros exactos para recibir el mensaje.

### C.3.4. Uso de los *JavaBeans*

Ya se ha reseñado previamente la existencia, y puesta a disposición del diseñador, de los *beans* de cada uno de los tipos de diálogos. Estos 'beans' no son más que los diálogos reusables de la librería, desarrollados siguiendo unas directrices que les permiten actuar como *JavaBeans* (sección 2.4.1). Se muestran en la Paleta de la vista de Diseño, como se puede ver en la figura C.5.

Para poner en contexto la situación y las limitaciones que tienen su uso, conviene comenzar señalando las dos posibilidades que tiene el diseñador de interfaces, cuando va a construir un nuevo cuadro de diálogo reutilizable usando la librería.

- Por un lado, puede utilizar las plantillas, es decir, diálogos derivados de la librería, creados con el editor gráfico de NetBeans, los cuales ya incluyen contenedor, *layout* adecuado, y en el caso de los diálogos de tipo árbol, la vista de árbol correspondiente como componente gráfico añadido, además de como propiedad modificable. De esta forma, puede olvidarse de las cuestiones relacionadas con la librería, y pasar a diseñar directamente. Es la forma más recomendable de crear los diálogos.

- Por otro lado, puede partir de una ventana completamente vacía, y arrastrar el *bean* desde la Paleta, correspondiente al tipo de diálogo elegido. Es la forma más avanzada de diseñar un diálogo, y la más complicada a su vez.

Este *bean* no tendrá ningún elemento dentro, pero sí que tendrá una serie de propiedades correspondientes a los diálogos reusables, como por ejemplo la propiedad para el contenedor. El diseñador sabrá el tipo de contenedor más adecuado para cada tipo de diálogo, por lo que podrá arrastrarlo también desde la Paleta, colocarlo, darle el *layout* que considere conveniente, y enlazar dicho contenedor con la propiedad del diálogo reusable, para conectar con el mecanismo de la librería. De la misma forma, podrá editar cualquier otra propiedad del diálogo, así como también las que ofrece la librería para los diálogos reusables, además de la del contenedor.

Igualmente a como ocurría con los diálogos creados a partir de plantillas, deberá implementar ciertos métodos (validación, guardado, limpieza y recepción de mensajes), y a partir de ahí, podrá continuar trabajando como con cualquier otro diálogo normal de Swing.

#### C.3.4.1. Inserción de *beans* como diálogos hijos

A mayores de lo comentado anteriormente, cabe hacer una advertencia. El diseñador no debe utilizar los *beans* para insertarlos dentro de otros diálogos reusables (creados a partir de plantillas o de otros *beans*). Si lo hace simplemente arrastrando el componente, pero no hace nada más, dicho diálogo no formará parte del mecanismo de la librería, puesto que no se habrá insertado usando los métodos que ésta dispone ('addExtensibleChild' o 'addExtensibleChildrenList').

Para hacerlo correctamente, además de arrastrar el componente *bean*, deberá añadir cierto código manualmente, con el fin de completar correctamente la inserción, en cierta forma emulando el comportamiento que hace la propia librería. El código necesario se muestra en el listado C.2. Se debe añadir en el constructor del diálogo padre en el cuál se está insertando, justo después del método de inicialización de componentes que tienen todos los componentes de Swing: initComponents.

Aún existiendo la posibilidad, **esta opción no está recomendada**, y en caso de que, aún así, se quiera insertar un *bean* como hijo dentro de un diálogo reusable, sería preferible que fuese el programador, y no el diseñador, el que realizase dicha funcionalidad desde su código. Esto se debe a lo comentado tanto en el manual del programador, como aquí, en el del diseñador, de separar los roles, centrándose cada uno en una vista diferente del IDE, siendo la vista de Codificación, tarea del programador.

## C.4. Ampliaciones disponibles

De la misma forma que el programador tiene algunas opciones extra, respecto a la concepción original de la librería, a la hora de utilizarla, el diseñador también dispone de un par de mejoras de las que puede hacer uso si así lo desea.

### ■ Personalización de la ventana

Ampliación compartida con el programador, o mejor dicho, complementaria a dicha opción. Gracias al parámetro 'parent' explicado en la sección B.4, el programador tiene la alternativa de proporcionar una ventana que actúe como envoltura para el diálogo principal de la librería, a la hora de generarlo mediante el método 'createDialog' de la factoría de la librería.

Dicha ventana se usaría en lugar de la ventana por defecto que crearía la factoría, en caso de que se le pasase dicho parámetro con valor nulo. El diseñador puede colaborar en esta labor, construyendo y proporcionando esa ventana personalizada al programador, de tal forma que éste pueda pasarla al método de construcción del diálogo principal, devolviendo la librería el diálogo principal, contenido en la ventana.

Además, el diseñador podría hacer uso de algunas opciones que el diseñador tiene disponibles mediante el código, pero que establecería desde la vista de Diseño, en su lugar. Por ejemplo, hablamos aquí del título de la ventana personalizada, modificable como una propiedad del JFrame ('title').

### ■ Asignación para el nombre de la raíz de la vista de árbol

Mientras que el desarrollador de aplicaciones tiene disponible, mediante código, el método 'setTreeRootName' para modificar dicha característica de los diálogos con vista de árbol, el diseñador puede hacer lo propio a través de la propiedad 'treeRootName', accesible desde la vista de Diseño, para los diálogos de tipo vista de árbol. Modificando dicha propiedad, como se comentó en el manual del programador, ésta se asignará al campo del diálogo en el momento en el que el programador lo inicialice. Sin embargo, tiene también el propio programador la opción de volver a asignar dicha propiedad, con el método comentado, sobrescribiendo el posible valor que le haya dado el diseñador.

### ■ Otras propiedades modificables

De forma similar al anterior punto, el diseñador tiene también otra serie de propiedades de los diálogos reusables, las cuales se pueden modificar desde su vista de Diseño, asignándoles el valor deseado. Es el caso, por ejemplo, de la propiedad 'containerName' común a todos los diálogos, cuyo cometido ya se comentó. Asimismo, también tendría la propiedad 'caption', transversal a la jerarquía de diálogos reutilizables, pues es la raíz, JIAExtensibleDialog, quien la implementa. Ambas ya han sido detalladas en sendas secciones del manual del programador, por lo que no repetiremos su objetivo aquí.

## C.5. Ejemplos de utilización

Se busca en esta sección, el detallar más gráfica y ampliamente los dos ejemplos de uso principales con los que se encuentra el diseñador a la hora de interactuar con la librería. En su caso, al contrario que en el del programador, la interacción no es directa, pues no utiliza la interfaz pública de la librería, pero si debe tenerla en cuenta de cara a mantener los diálogos que construya dentro de la compatibilidad con el mecanismo de reutilización de la librería.

Se hablará en primer lugar de la creación de diálogos desde cero, usando *templates*, para pasar a continuación a comentar la forma de insertar *beans* desde la Paleta, que representen algún tipo de diálogo de los disponibles en la librería.

### C.5.1. Creación de diálogos a partir de plantillas

El diseñador, como se comentó en la sección C.3.4, tiene dos opciones a la hora de crear los diálogos principales. Por un lado, desde cero, con una ventana completamente vacía, donde tendría que hacer todos los pasos, y por otro lado, la que se mostrará aquí, mucho más sencilla, utilizando los *templates* que se proporcionan junto a la librería. Es conveniente recordar en este punto la sección C.3.2, para elegir la ubicación de paquete más adecuada para los nuevos diálogos.

En primer lugar, debe dirigirse, o bien al menú de NetBeans, pinchando en 'Archivo/Archivo nuevo', o bien, para asegurarse de no seleccionar una ubicación equivocada, puede pinchar, haciendo click derecho, en el paquete que desee de la jerarquía del proyecto, y a continuación darle a 'Nuevo'/'Otro...'. Al hacerlo, se abrirá la ventana mostrada en la figura C.6.

En ella aparecen disponibles, dentro de la categoría de 'Formularios de interfaz gráfica de Swing', los 3 tipos de diálogos de la librería. Cogiendo como ejemplo, el de la vista de árbol, podríamos seleccionarlo, dar a 'Siguiente', asignarle un nombre al nuevo diálogo y pinchar en 'Terminar', dando como resultado lo mostrado en la figura C.7. El nuevo diálogo se abriría por defecto en la vista de Diseño, y el diseñador de interfaces podría comenzar directamente su tarea.

### C.5.2. Inserción de diálogos desde la Paleta

Por otro lado, como se comentó en la sección C.3.4, se pueden arrastrar los *beans* disponibles desde la Paleta, para dos cosas. En primer lugar, hacia una ventana vacía, para empezar a diseñar el diálogo principal, prescindiendo de las plantillas. En segundo lugar, para insertar ese *bean* como diálogo hijo dentro de un diálogo reusable. Se mostrará este segundo caso.

Supongamos que partimos de un diálogo de pestañas ya creado, que actúa como diálogo principal. Estamos en la situación de la figura C.8.

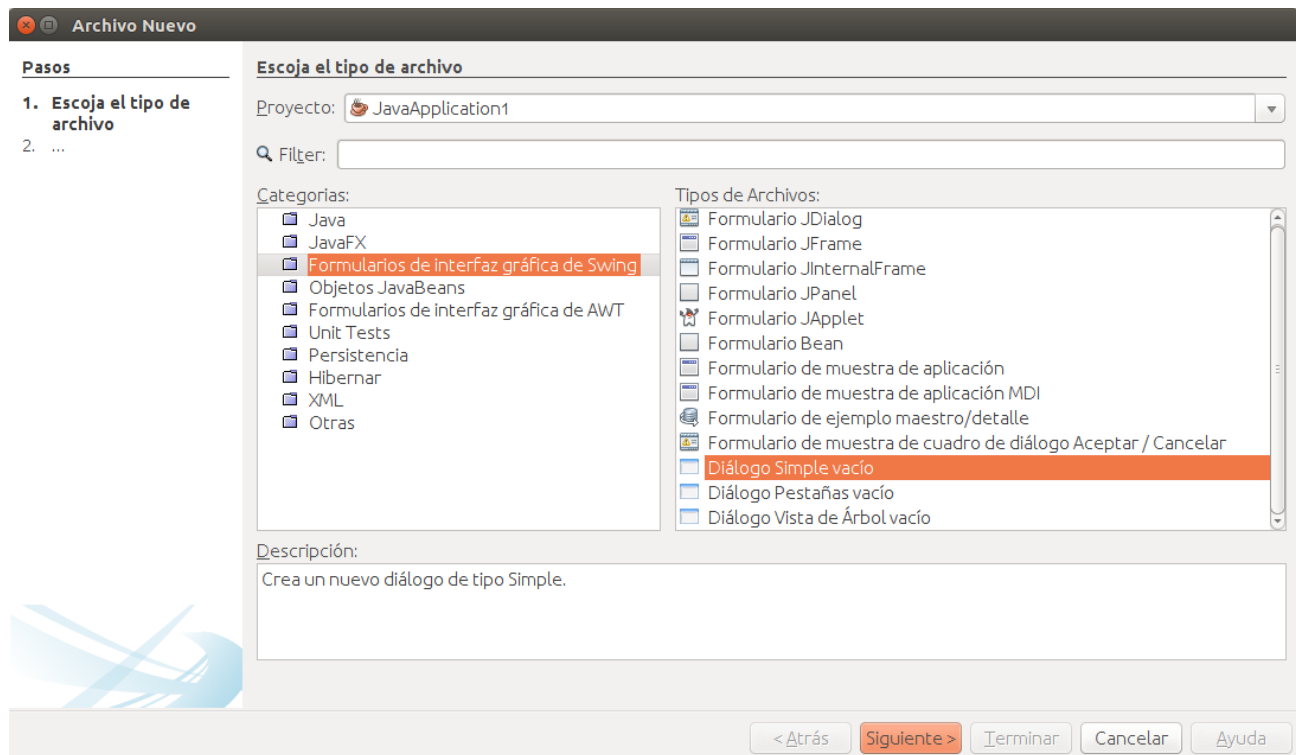


Figura C.6: Ventana de creación de un 'Archivo nuevo'

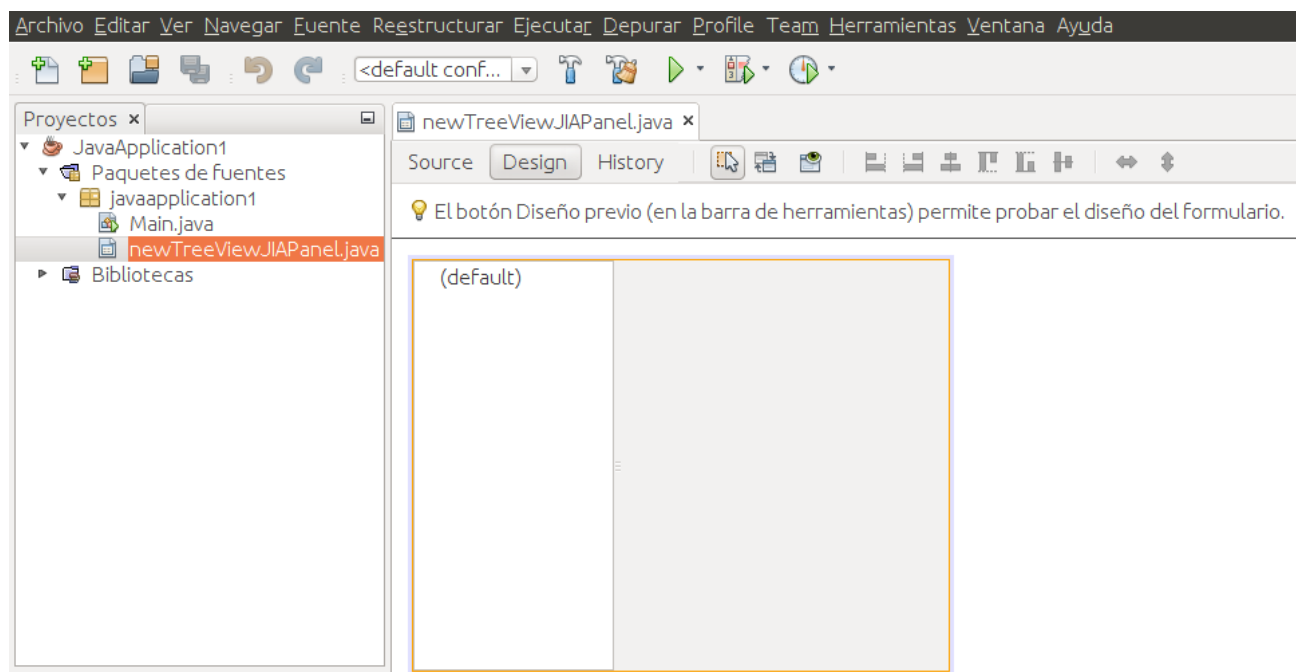


Figura C.7: Diálogo con vista de árbol creado con una plantilla

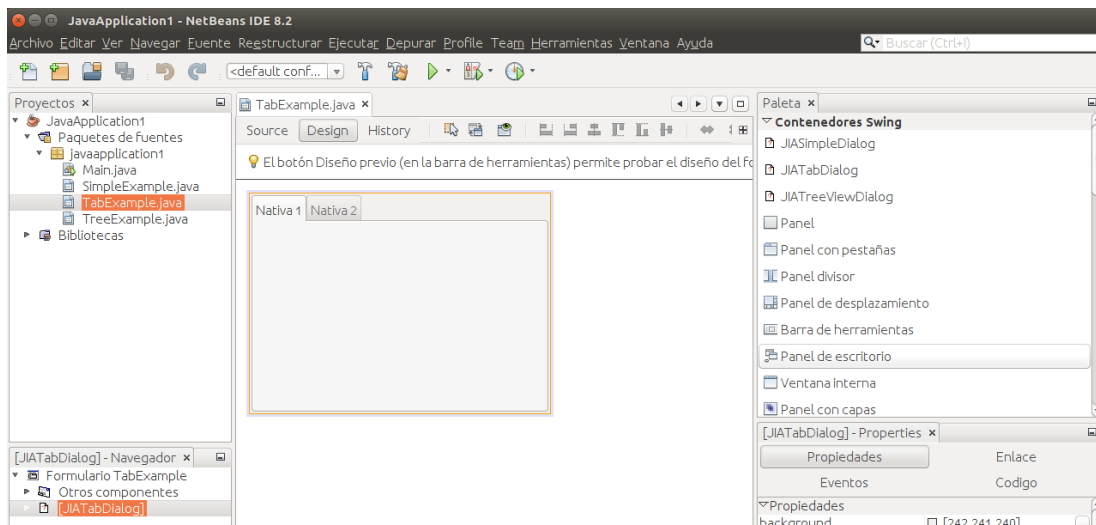


Figura C.8: Ejemplo sección C.5.2, diálogo de pestañas inicial

A continuación arrastramos un *bean* de diálogo de tipo Simple desde la Paleta, lo cuál se ejemplifica en la figura C.9. Se marca en rojo la situación de que se está en pleno proceso de “arrastre”. La zona roja recuadrada representa el *bean* simple que se está añadiendo. Una vez añadido, podemos expandir su tamaño.

Por último, una vez elegida la localización para el nuevo diálogo arrastrado desde la Paleta, podemos cambiar su tamaño, asignarle otro *layout*, e incluso añadirle componentes, como se ve en la figura C.10, al añadirle un botón, pudiendo tratarlo ya como un componente Swing más.

Una vez completado este proceso, se debe recordar que el *bean* aún no está listo para su compatibilidad con el mecanismo de reutilización de la librería, pues se debe seguir lo indicado en el punto C.3.4.1, añadiéndolo a la lista de hijos del padre, asociando los *listeners* de ambos, y estableciendo el padre en el campo adecuado.

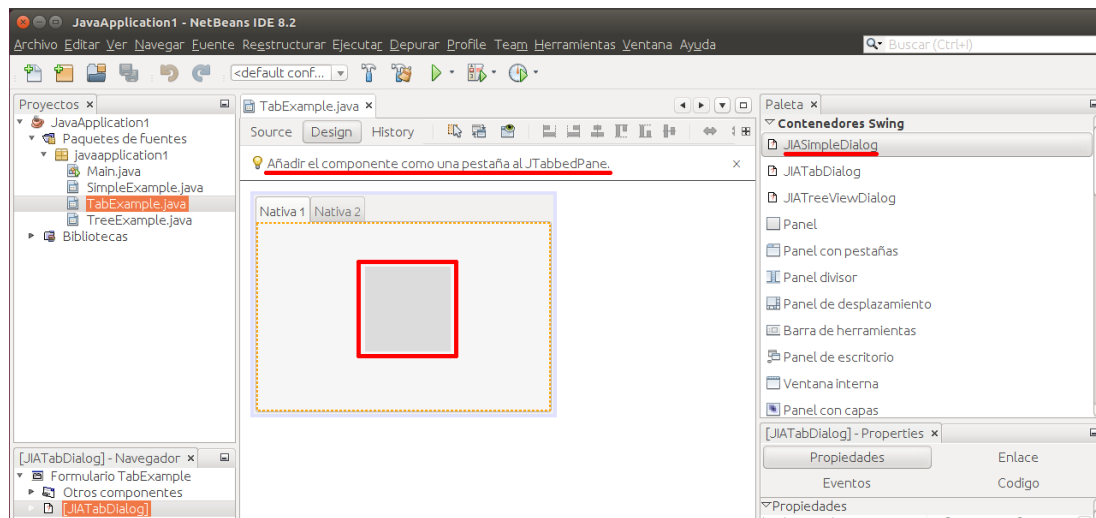


Figura C.9: Ejemplo sección C.5.2, añadiendo *bean* de diálogo simple

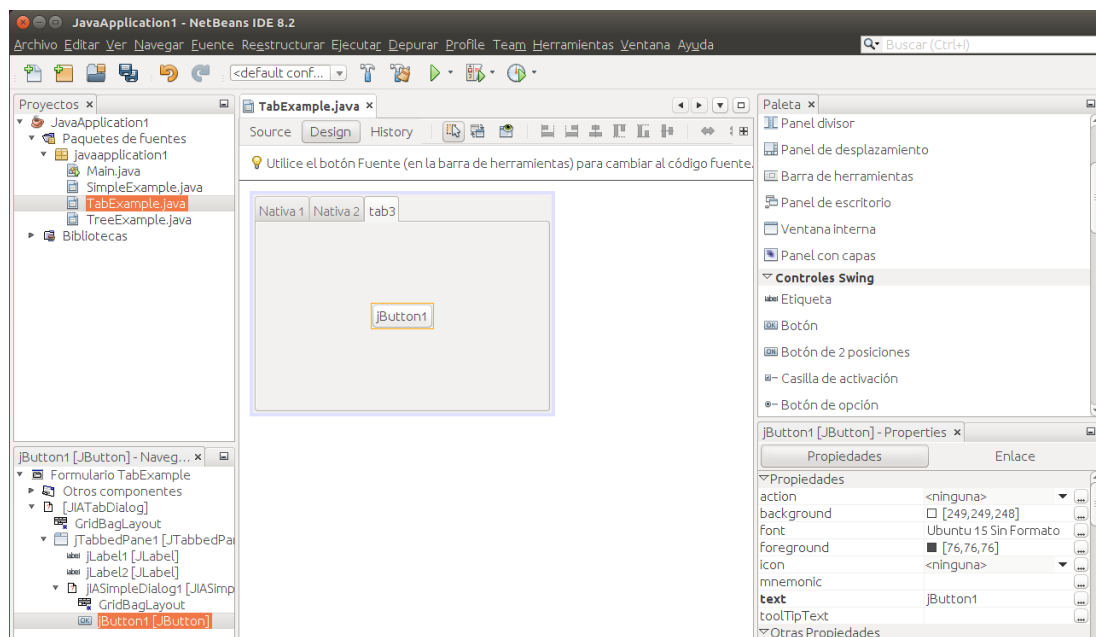


Figura C.10: Ejemplo sección C.5.2, *bean* simple añadido

Listado de código C.2: Código necesario si insertamos un *bean* desde la Paleta

```

1
2 if(beanInsertado != null) {
3     // añadimos el hijo a la lista
4     this.getListHijos().add(beanInsertado);
5     // asociamos los listeners de manera adecuada
6     this.linkListeners(beanInsertado);
7     // asociamos el padre con su hijo
8     beanInsertado.setAncestor(this);
9 }

```



## Anexo D

# Manual del programa de pruebas

En este manual se tratará de comentar y ejemplificar el trabajo realizado con el programa de pruebas, explicando su estructura, flujo de operación y particularidades, así como los ejemplos de uso de la librería que se han realizado con el fin de mostrar sus posibilidades.

Para esta aplicación, cuyo nombre de proyecto ha sido *pfg-uned-sgh*, se ha pensado en hacer una pequeña demostración de un software que asista a los profesionales sanitarios, en las gestiones administrativas y clínicas necesarias dentro de un hospital. Por supuesto, debido a su finalidad meramente demostrativa de uso de la librería, dicha aplicación es tan solo una ínfima parte de todo el posible proceso que podría tener un software real para estos propósitos, limitándose a dos roles de usuario, y una casuística particular donde puedan necesitar compartir información entre ambos, pudiendo reutilizar los diálogos. El sufijo SGH se ha escogido debido al objetivo del programa de pruebas, intentando darle una denominación adecuada y representativa.

Se comenzará reseñando brevemente las características principales de la aplicación, como ayuda para tener un mejor conocimiento global acerca de ella, así como su estructura general. Más tarde, se pasarán a detallar las diferentes pantallas que tiene disponibles a lo largo de la misma. Por último, se desglosarán algunas situaciones en las que se ha precisado de hacer uso de las alternativas y funcionalidades que provee la librería, entre otras cosas, para la inserción y combinación de diálogos, y para el paso de mensajes entre ellos.

### D.1. Estructura y características

Se iniciará esta sección mostrando la estructura general de la aplicación, en la figura D.1. Como se puede ver, el proyecto del programa de prueba se divide básicamente en 3 bloques: la aplicación en si misma, con la clase Main desde la que arrancará siempre la ejecución, un paquete donde se exportará el fichero de la base de datos persistente (y simulada), y el último, donde se ubicarán todas las imágenes usadas a lo largo de la aplicación.

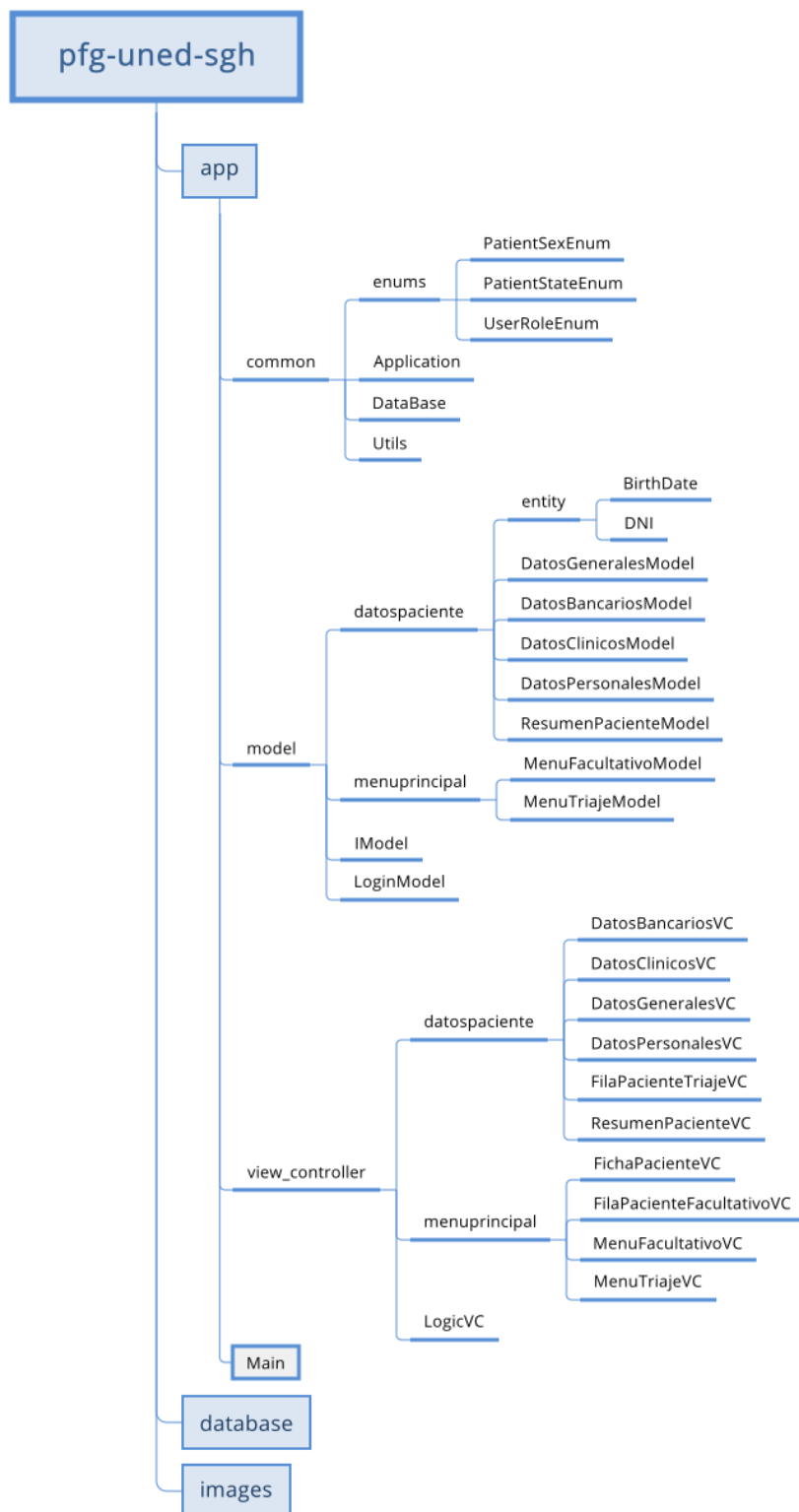


Figura D.1: Estructura de la aplicación de prueba

## ■ Paquete *app*

- Subpaquete *common*. Incluye todas las utilidades y clases transversales a la aplicación.
  - Tenemos por un lado un conjunto de enumerados, útiles para discernir el sexo de los pacientes, su estado hospitalario dentro de la aplicación, o los roles de usuario disponibles para realizar el inicio de sesión.
  - Por otro lado la clase 'Application', la cual implementa el patrón *Singleton* (sección 2.2.4) y es invocada desde el main del programa. En ella se lanza todo el flujo de ejecución del programa, desde la primera ventana hasta que se cierra la última.
  - Después nos encontramos con la clase 'DataBase', la cuál simula una fuente de datos para la aplicación, como una colección de roles de usuario, o de pacientes registrados, por poner algunos ejemplos. Toda la información está *hardcodeada* (Yearofthedragon (2005)), es decir, incrustada directamente en el código, para mantener la simplicidad.
  - Finalmente, tenemos la clase 'Utils', la cual engloba ciertas funcionalidades auxiliares utilizadas a lo largo de la aplicación: lectura y escritura del fichero externo de base de datos, métodos para centrar ventanas, funciones para limpiar o deshabilitar diálogos recursivamente (descendiendo por su jerarquía y haciendo lo propio con los descendientes), inicialización y/o generación de diálogos desde la factoría, o bien los propios del diseñador, etc.
- Subpaquete *model*. Contiene el **modelo** de la aplicación, según el MVC (sección 2.3.1).
  - Por un lado, tenemos la interfaz 'IModel', la cuál implementan todas las clases de la jerarquía del modelo. Su única función, como se puede ver en el código D.1, es la de extender la interfaz 'Serializable', de tal forma que todas las clases del modelo lo sean, de cara a la persistencia a fichero de la base de datos.
  - A continuación, tendríamos el modelo para la pantalla de inicio de sesión, la primera que aparecer al ejecutar la aplicación. Simplemente contiene campos para almacenar el usuario y contraseña introducidos, y nada más, puesto que como recomienda una buena práctica de programación, las clases del modelo de una aplicación no deberían albergar ningún tipo de lógica de negocio compleja.
  - Después tendríamos los datos del paciente, como una jerarquía donde la raíz es la clase 'DatosGeneralesModel', que incluye datos compartidos para todos los tipos, y de ella heredan el resto. Además, tendríamos el 'ResumenPacienteModel', el cual contiene una instancia de cada uno de los tipos de datos, para un paciente en concreto, de tal forma que se utiliza como representación de la información aglutinada de dicho paciente.

- Por último, vemos los modelos correspondientes a los dos menús de la aplicación. En ambos casos, cada uno contiene una colección de los pacientes registrados. Dicha colección es en realidad un mapa, donde la clave de cada entrada será el código SNS del paciente, y el valor de la entrada será el ResumenPaciente explicado anteriormente, el cuál contiene toda su información.
- Subpaquete *view\_controller*. En él están todas las clases relacionadas con la parte de **vista** y de **controlador** de la aplicación, la cuál aplica el patrón MVC (sección 2.3.1). Se ha decidido unificar ambas partes, vista y modelo, en un solo paquete, debido a las particularidades de Swing. Mientras que otros frameworks gráficos de programación, como pueda ser por ejemplo ZK (Potix (2017); Anonymous (2007)), permiten separar claramente la vista, del modelo, de los controladores, en Swing esa línea es más difusa, puesto que tanto el fichero para la vista (.form) como el del controlador (.java) están intrínsecamente relacionados, no pudiendo dividirlos en paquetes diferenciados. La única forma de acceder a uno y a otro es mediante las vistas, de Diseño, o bien de Codificación, del IDE correspondiente.  
El contenido de este subpaquete se detallará con mayor precisión en la sección D.2, a continuación, cuando hablemos de los flujos de ejecución que puede seguir la aplicación desde su arranque, todo ejemplificado con capturas de pantallas de la propia aplicación.

#### ■ Paquete *database*

Ubicación del directorio donde se almacena la base de datos persistente de la aplicación. El fichero guardado tiene la ruta: 'rutaDirectorioProyecto\src\database\dataBase.dat'.

Dicha base de datos, es fundamentalmente la instancia del objeto de la clase 'DataBase', el cuál se crea en la aplicación siempre que se inicia. Esa instancia, en primer lugar, se intenta leer desde el fichero, para ver si existe una base de datos previa que cargar, y en caso de no encontrarla, se crea una nueva. Más tarde, cuando se cierra la aplicación, siempre cerrando la ventana de inicio de sesión en último lugar, y antes de finalizar completamente, se persiste en el fichero y la ruta comentados anteriormente la instancia del objeto, por las posibles modificaciones que haya podido sufrir durante la ejecución del programa.

De esta forma, no solo tenemos una base de datos con información simulada, sino además hacemos que sea persistente entre diversas ejecuciones de la aplicación, si así lo deseamos.

#### ■ Paquete *images*

Se trata básicamente de un conjunto de imágenes utilizadas en la aplicación, separadas del paquete principal por mantener una organización más clara. Hay de varios tipos, desde un icono de paciente con diferentes colores, los cuales siguen un código acorde al nivel de prioridad de atención urgente del paciente, hasta efigies del estado del paciente.

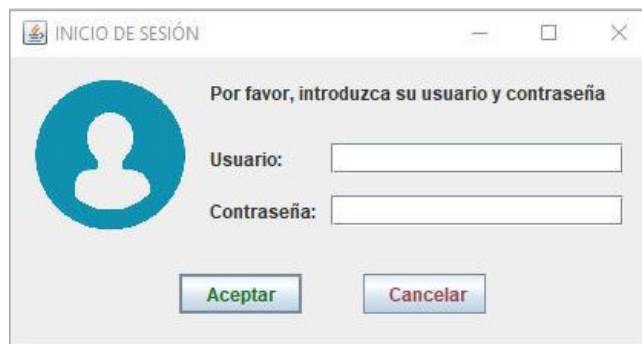


Figura D.2: Pantalla de inicio de sesión

## D.2. Flujos de ejecución

Una vez arranquemos la aplicación, siempre nos aparecerá la misma pantalla preliminar, que será la de inicio de sesión, mostrada en la figura D.2. En dicha pantalla deberemos introducir el usuario y contraseña del perfil de usuario con el que queremos iniciar sesión, lo cuál nos conducirá a diferentes pantallas sucesivas. En la base de datos únicamente existen dos usuarios registrados:

1. **Triage.** Representa el rol del usuario encargado de triar los pacientes que llegan al hospital. Su usuario será, literalmente *Triage*, pues la base de datos distingue mayúsculas y minúsculas. La contraseña deberemos dejarla vacía, porque así está establecido para mantener la sencillez.
2. **Facultativo.** Representa el rol de un usuario cualquiera que actúe como facultativo dentro del hospital, y sea el encargado de realizar la atención médica a los pacientes. Su usuario será *Facultativo*, una vez más, la primera en mayúscula. La contraseña deberemos dejarla vacía.

En base a esto, el flujo de ejecución se dividirá en dos menús principales diferentes.

### D.2.1. Interacción con el rol de 'Triage'

Si iniciamos sesión con el perfil de triador, se nos abrirá el menú principal de Triage, el cuál se muestra en la figura D.3. En él, como podemos apreciar, tenemos a la vista la lista de pacientes registrados en la aplicación. Cada uno de ellos ocupa una fila, y tenemos, de izquierda a derecha:

- Icono que indica el sexo del paciente: masculino, femenino, o desconocido, si no está especificado porque se desconoce, o simplemente no se ha indicado.
- Nombre completo del paciente, con sus apellidos.
- Botón para consultar los detalles de la factura médica.
- Botón para modificar los datos relativos al paciente.

## Listado de código D.1: Código de la interfaz 'IModel'

```
1  
2 public interface IModel extends Serializable {  
3     // vacío, no necesita nada  
4 }
```

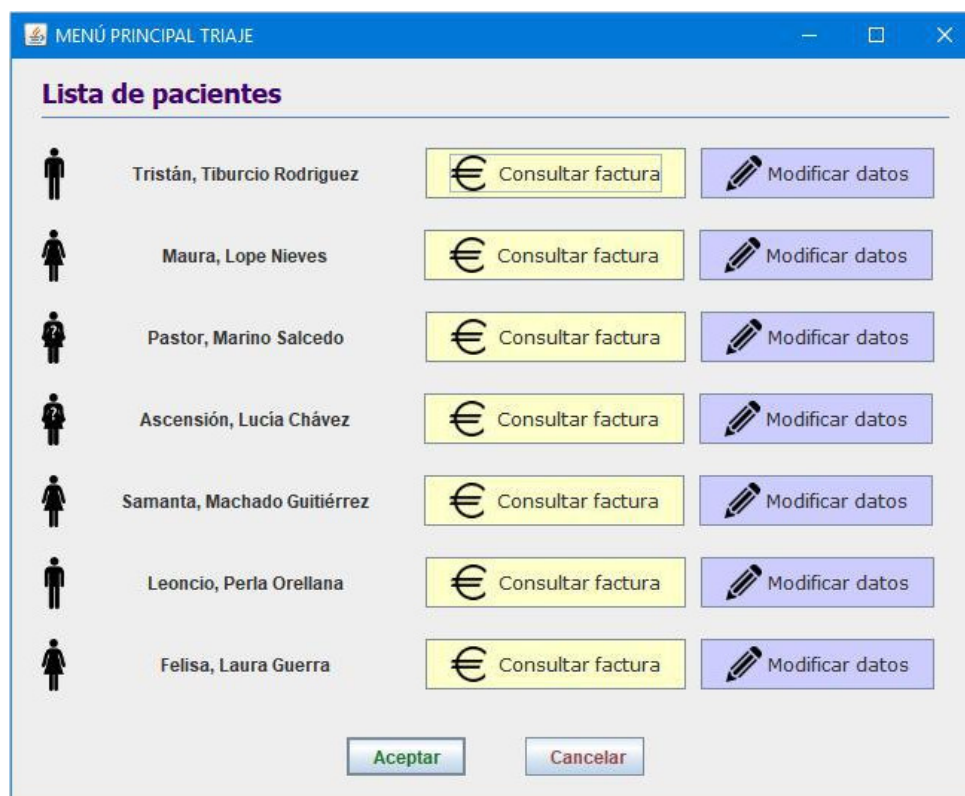


Figura D.3: Menú principal Triage

## Factura médica

Si decidimos pulsar en el botón de factura médica para uno de los pacientes, visualizaremos la pantalla de la figura D.4. En ella tenemos un resumen parcial de la información del paciente, donde veremos y podremos editar, datos generales, como el DNI, estado de atención hospitalaria actual o código SNS, junto a datos relativos a la información financiera del paciente, como su número de cuenta bancaria, así como el seguro médico, en caso de que tenga uno. Si no tiene seguro, la caja de texto de 'Compañía aseguradora' aparecerá deshabilitada, pero basta con marcar la casilla de 'Seguro médico', confirmando que tiene uno, para que se habilite el cuadro donde introducir la compañía con la que tiene dicho seguro.

Destacaremos aquí algo que comparten todas las pantallas diseñadas, y es la persistencia de los datos del modelo, no solo al finalizar la aplicación, como se explicó al final de la sección D.1 (paquete *database*). Sino también entra las diferentes pantallas y ventanas de la aplicación. De esta forma, si modificamos algún dato, por ejemplo, en la pantalla actual de la factura, podremos ver dichos cambios reflejados en otra parte de la aplicación donde se esté mostrando dicha información.

Esto es independiente del mecanismo de paso de mensajes que ofrece la librería, y usa la aplicación. Con el paso de mensajes, no se está guardando el dato en memoria, para recuperarlo más tarde desde otra parte, sino que se envía un mensaje, desde un diálogo, hasta otro cualquiera dentro de la misma jerarquía, con un identificador y un valor, lo que provoca el cambio de datos en el receptor del mensaje, no porque lo lea del modelo, sino porque recibe el mensaje con el identificador adecuado. Se ha creído conveniente resaltar aquí esta diferenciación.

## Resumen de información

Si en lugar de pulsar el botón de factura, pulsamos el de 'Modificar datos', en uno de los pacientes, se abrirá la ventana de la figura D.5. En ella podemos ver como se realiza una división entre dos pestañas, puesto que el diálogo principal de esa ventana se ha elegido que sea de tipo pestañas.

En la primera, veremos la información de la figura anteriormente comentada. Dicha información es transversal y bastante genérica, como puede ser el nombre completo del paciente (con el formato <apellidos, nombre>), un icono representando su estado, una barra que indica la urgencia de atención a dicho paciente, según su nivel de prioridad de triaje, o el código SNS, el cuál le identifica unívocamente a través de todo el sistema sanitario a nivel nacional.

En la segunda, tendremos la pestaña de edición de datos, la cual tiene anidadas hasta 4 subpestañas, mostradas en las figuras D.6, D.7, D.8, y D.9. Por orden, tendríamos la información de tipo: General, Personal, Clínica y Bancaria.

Como se ha comentado anteriormente, cualquiera de los datos introducidos que hayan pasado las validaciones necesarias, tras pulsar el botón de 'Aceptar', serán salvados como parte del modelo.

The screenshot shows a window titled "FACTURA MÉDICA" with a blue header bar. Below the header, there is a section titled "Datos generales" with a horizontal line separator. This section contains four input fields: "Nombre:" with the value "Tristán", "Apellidos:" with the value "Tiburcio Rodriguez", "DNI (8 dígitos + 1 letra):" with the value "10000001" and a dropdown menu showing "A", and "Código SNS:" with the value "900000001". Below these fields is a dropdown menu for "Estado:" with the value "Registrado". Below the "Datos generales" section is another section titled "Datos bancarios" with a horizontal line separator. This section contains two input fields: "Nº cuenta bancaria:" and "Compañía aseguradora:". Below the "Compañía aseguradora:" field is a radio button labeled "Seguro médico". At the bottom of the window are two buttons: "Aceptar" (green) and "Cancelar" (blue).

Figura D.4: Factura médica para un paciente

The screenshot shows a window titled "RESUMEN PACIENTE" with a blue header bar. Below the header, there are two tabs: "Resumen" (selected) and "Edición de datos". The "Resumen" tab displays the patient's name "Tiburcio Rodriguez, Tristán" and the "Código SNS | 900000001". Below this is a section titled "Estado" with a large blue icon of a person and a green checkmark. To the right of the "Estado" section is a vertical red bar representing the "Urgencia" level, with the value "90%06" written vertically. At the bottom of the window are two buttons: "Aceptar" (green) and "Cancelar" (blue).

Figura D.5: Resumen del paciente, con los datos básicos



RESUMEN PACIENTE

Resumen Edición de datos

General Personal Clínica Bancaria

Datos generales

Nombre: Tristán

Apellidos: Tiburcio Rodriguez

DNI (8 dígitos + 1 letra): 10000001 A

Código SNS: 900000001

Estado: Registrado

Aceptar Cancelar

Figura D.6: Modificación de los datos generales

RESUMEN PACIENTE

Resumen Edición de datos

General Personal Clínica Bancaria

Datos personales

Fecha de nacimiento (dd/mm/aaaa):

Sexo: Hombre

Dirección postal:

Correo electrónico:

Número de teléfono

Aceptar Cancelar

Figura D.7: Modificación de los datos personales

RESUMEN PACIENTE

Resumen Edición de datos

General Personal Clínica Bancaria

Datos clínicos

Doctor de cabecera:

Medicación actual:

☐ Reanimación Cardio Pulmonar (RCP)

Alergias conocidas

Prioridad triaje

☐ 0 1 2 3 4 5 6 7 8 9 10

Aceptar Cancelar

Figura D.8: Modificación de los datos clínicos

RESUMEN PACIENTE

Resumen Edición de datos

General Personal Clínica Bancaria

Datos bancarios

Nº cuenta bancaria:

Compañía aseguradora

☐ Seguro médico

Aceptar Cancelar

Figura D.9: Modificación de los datos bancarios

### D.2.2. Interacción con el rol de 'Facultativo'

La alternativa, en lugar de acceder con el rol de Triage, es iniciar sesión en la aplicación con un usuario con el rol de 'Facultativo'. Si lo hacemos, se mostrará el menú principal del facultativo, el cual encontramos en la figura D.10.

En él, de forma similar a como ocurría con el primer caso, tenemos una lista de pacientes, uno por cada fila, para los cuales tendremos, de izquierda a derecha:

- Botón de inicio de atención al paciente, si queremos asignarlo al facultativo en cuestión y comenzar su episodio, pasando a otra pantalla.

Cuando pulsamos en dicho botón, se abrirá el cuadro de confirmación que se muestra en la figura D.11, el cual nos permite ratificar si realmente queremos comenzar con el cuidado.

- Nombre completo del paciente, con el formato <nombre, apellidos>.
- Un icono que representa el estado del paciente, y que cambia de color según el nivel de triaje que se le haya asignado. Por defecto, con el nivel más bajo (nivel 0), todos los pacientes tendrán ese icono de color azul.

Si elegimos iniciar la atención del paciente, pasamos a la pantalla mostrada en la figura D.12, la cual representa la 'Ficha de paciente' para el paciente en cuestión.

Como vemos, en esta ocasión se trata de un diálogo con vista de árbol, cuya raíz se llama ficha de paciente, y donde tenemos dos entradas en el nivel superior:

- Por un lado, el resumen de paciente ya visto en el rol de Triage.
- Por otro, una segunda entrada, llamada 'Información detallada', que aglutina los distintos tipos de datos del paciente, con una entrada para cada uno de ellos, permitiendo el acceso y la modificación individual en cada tipo.

## D.3. Ejemplos de uso de la librería

En esta sección recalcaremos, señalando con más detalle, algunos ejemplos de uso de la librería, de todo lo visto anteriormente acerca de la aplicación. Se dividirá en dos subsecciones.

Por un lado, los ejemplos de reutilización de diálogos propiamente dichos, donde veremos como un mismo diálogo construido por el diseñador, ha sido combinado y empleado de formas diferentes y en diálogos diversos, por parte del programador, mostrando así el potencial del mecanismo de reusabilidad, punto central de la librería.

Por otro lado, mostraremos algún ejemplo de paso de mensajes entre diálogos pertenecientes a la misma jerarquía, es decir, contenidos en el mismo diálogo principal, bajo la misma ventana.

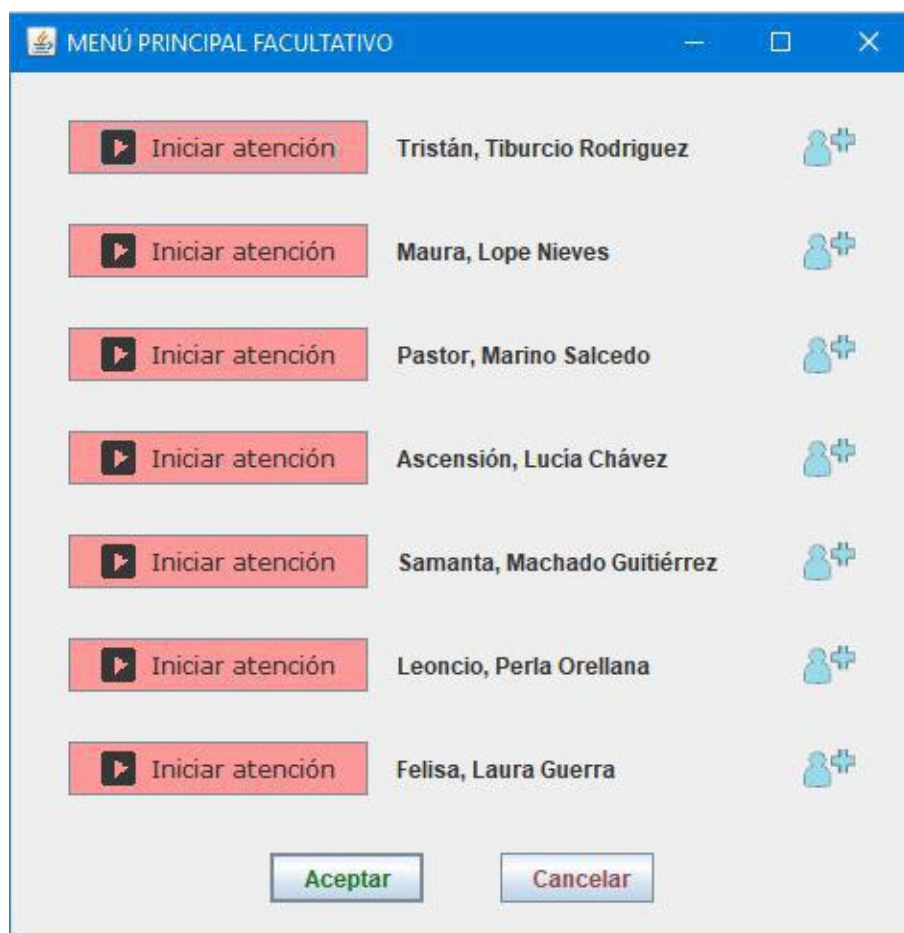


Figura D.10: Menú principal facultativo

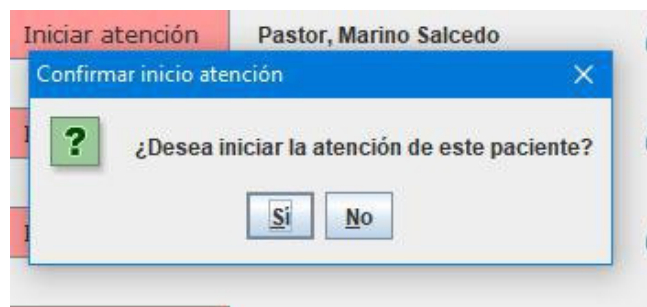


Figura D.11: Mensaje de confirmación de atención a un paciente

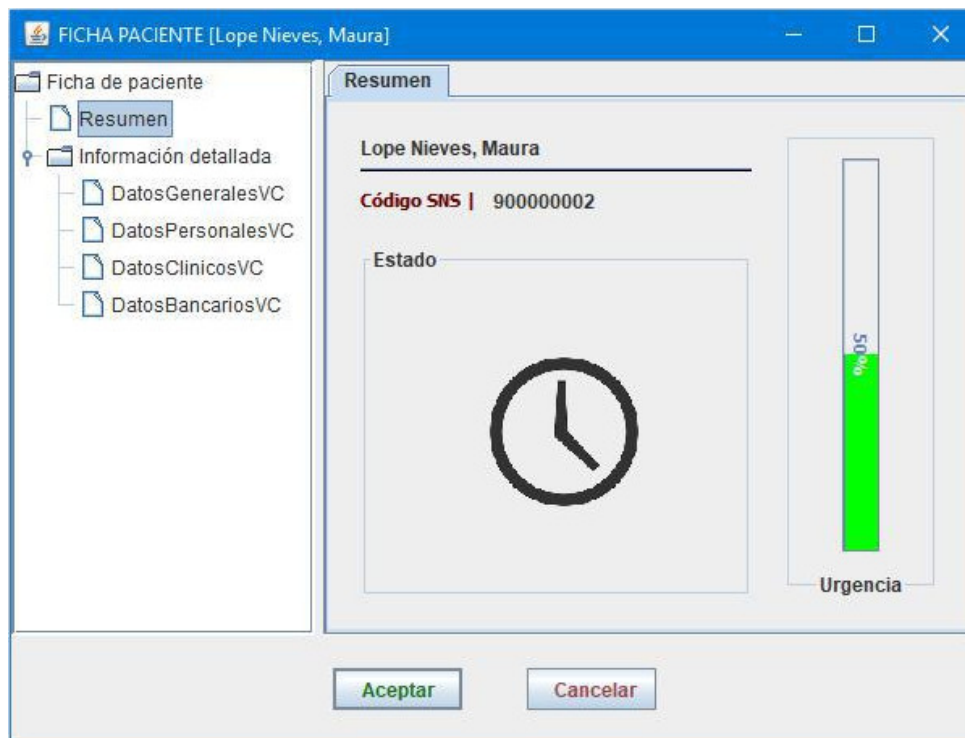


Figura D.12: Ficha de paciente, con el resumen y los datos separados

### D.3.1. Variedad en la inserción de diálogos reutilizados

Como ejemplo más claro de reutilización de diálogos, tenemos el caso de la pantalla que muestra los Datos Bancarios del paciente, la cuál ha sido empleada y mostrada hasta 3 veces, utilizando todos los tipos de posibles diálogos que ofrece la librería.

- En la figura D.13, podemos verlo insertado en la parte inferior de un diálogo Simple, formando parte de la 'Factura médica' de un paciente.
- En la figura D.14, podemos verlo añadido como una cuarta pestaña, independiente de las otras, anidada bajo la pestaña de 'Edición de datos', en el 'Resumen del paciente'.
- En la figura D.15, por último, podemos apreciar como se ha agregado a un diálogo de tipo Vista de árbol, bajo el epígrafe de 'Información detallada', mostrándose de forma aislada y pudiendo editarse individualmente, de la misma forma que las otras entradas.

En todos los casos, el diseñador ha tenido que diseñar un único diálogo, siguiendo unas pautas mínimas para asegurar su reusabilidad. Más tarde, el programador ha podido instanciarlo, inicializarlo y combinarlo en hasta 3, o más si así lo hubiera deseado, jerarquías de diálogos diferentes. Este es el verdadero potencial del mecanismo de reusabilidad desarrollado en la librería, además de otras funcionalidades, como la comentada a continuación.

The screenshot shows a window titled 'FACTURA MÉDICA'. It contains a 'Datos generales' section with fields for 'Nombre:' (Tristán), 'Apellidos:' (Tiburcio Rodríguez), 'DNI (8 dígitos + 1 letra):' (10000001 A), and 'Código SNS:' (900000001). There is also an 'Estado:' dropdown menu set to 'Registrado'. Below this is a section titled 'Datos bancarios' which is highlighted with a red border. This section contains a 'Nº cuenta bancaria:' field, a 'Compañía aseguradora' label, and a radio button labeled 'Seguro médico' next to another field. At the bottom are 'Aceptar' and 'Cancelar' buttons.

Figura D.13: Diálogo de DatosBancarios, insertado en un diálogo Simple

The screenshot shows a window titled 'RESUMEN PACIENTE'. It has two tabs: 'Resumen' and 'Edición de datos'. Under 'Edición de datos', there are sub-tabs: 'General', 'Personal', 'Clínica', and 'Bancaria'. The 'Bancaria' sub-tab is selected. The main area contains a section titled 'Datos bancarios' highlighted with a red border. This section includes a 'Nº cuenta bancaria:' field, a 'Compañía aseguradora' label, and a radio button labeled 'Seguro médico' next to another field. At the bottom are 'Aceptar' and 'Cancelar' buttons.

Figura D.14: Diálogo de DatosBancarios, insertado en un diálogo de Pestañas

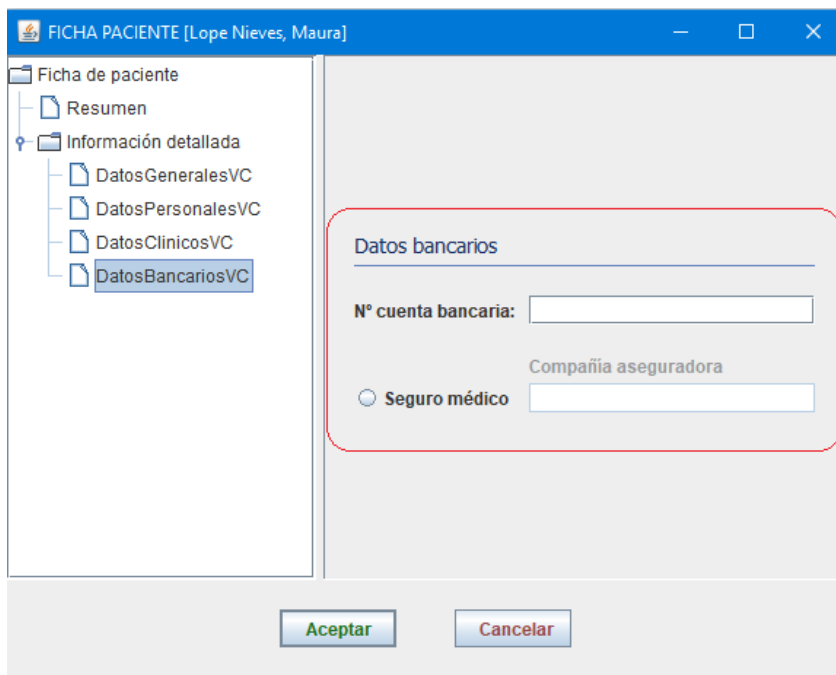


Figura D.15: Diálogo de DatosBancarios, insertado en un diálogo con Vista de árbol

### D.3.2. Paso de mensajes entre diálogos independientes

En lo referente al paso de mensajes, la librería tiene un par de casos de uso de dicho mecanismo.

- El primer caso (A), lo encontramos en la pantalla del 'Resumen del paciente', cuyo estado inicial se muestra en las figuras D.16 y D.17. Ahí se puede apreciar como, en la primera figura, tenemos un icono (y un *tooltip*) para el estado de REGISTRADO. A continuación, desde los Datos Generales del paciente, modificamos el valor de dicho estado, a través de un combobox (figura D.18), pasando a INGRESO. Es en este momento cuando se dispara el evento asociado al cambio de valor en el combobox, y se realiza el paso del mensaje, el cuál es recibido por el otro diálogo, provocando la situación de la figura D.19, donde, tanto el icono como el *tooltip* asociado, se han cambiado por los del nuevo estado asignado al paciente.

Esta misma conexión se podrá apreciar también en otros lugares donde el programador haya decidido reaprovechar esos mismos diálogos en otra jerarquía, como por ejemplo en la 'Ficha de paciente' (menú del facultativo). Por lo que si hacemos lo propio allí, se podrá apreciar el mismo entrelazamiento.

- El segundo caso (B), podemos comprobarlo con el mismo diálogo receptor del mensaje como protagonista, el 'Resumen del paciente', estado inicial en la figura D.20. Aunque en este caso, el diálogo que lo emite será el correspondiente a los Datos Clínicos, cuyo estado inicial se muestra en la figura D.21. Si modificamos el nivel de triaje, estableciéndolo por ejemplo en 8,

Resumen Edición de datos

Machado Gutiérrez, Samanta

Código SNS | 900000005

Estado

REGISTRADO

Urgencia %0

Figura D.16: Caso A. Estado inicial, antes de recibir el mensaje

Resumen Edición de datos

General Personal Clínica Bancaria

Datos generales

Nombre: Samanta Apellidos: Machado Gutiérrez

DNI (8 dígitos + 1 letra): 10000005 E Código SNS: 900000005

Estado: Registrado

Figura D.17: Caso A. Estado inicial, antes de enviar el mensaje

a través de la pantalla de Datos Clínicos (figura D.22), podremos ver como sucede el cambio automático, gracias al paso del mensaje, en la pantalla de la figura D.23.



The screenshot shows a software window titled 'Edición de datos' with a sub-tab 'General'. It contains a form for 'Datos generales' with the following fields:

Datos generales	
Nombre:	Samanta
Apellidos:	Machado Guitiérrez
DNI (8 dígitos + 1 letra):	10000005 E
Código SNS:	900000005
Estado:	Ingreso

Figura D.18: Caso A. Estado posterior, después de enviar el mensaje

The screenshot shows a 'Resumen' window for 'Machado Guitiérrez, Samanta'. It displays the 'Código SNS' as 900000005 and the 'Estado' as 'INGRESO', represented by a green icon of a person lying down. A vertical bar on the right indicates the 'Urgencia' level, which is currently at 0%.

Figura D.19: Caso A. Estado posterior, después de recibir el mensaje

The screenshot shows a 'Resumen' window for 'Lucía Chávez, Ascensión'. It displays the 'Código SNS' as 900000004 and the 'Estado' as 'INGRESO', represented by a green icon of a person lying down. A vertical bar on the right indicates the 'Urgencia' level, which is currently at 0%.

Figura D.20: Caso B. Estado inicial, antes de recibir el mensaje

Ficha de paciente

- Resumen
- Información detallada
  - DatosGeneralesVC
  - DatosPersonalesVC
  - DatosClínicosVC
  - DatosBancariosVC

**Datos clínicos**

Doctor de cabecera:

Medicación actual:

☐ Reanimación Cardio Pulmonar (RCP)

Alergias conocidas

Prioridad triaje

0 1 2 3 4 5 6 7 8 9 10

Figura D.21: Caso B. Estado inicial, antes de enviar el mensaje

Ficha de paciente

- Resumen
- Información detallada
  - DatosGeneralesVC
  - DatosPersonalesVC
  - DatosClínicosVC
  - DatosBancariosVC

**Datos clínicos**

Doctor de cabecera:

Medicación actual:

☐ Reanimación Cardio Pulmonar (RCP)

Alergias conocidas

Prioridad triaje

0 1 2 3 4 5 6 7 8 9 10

Figura D.22: Caso B. Estado posterior, después de enviar el mensaje

Ficha de paciente

- Resumen
- Información detallada
  - DatosGeneralesVC
  - DatosPersonalesVC
  - DatosClínicosVC
  - DatosBancariosVC

**Resumen**

Lucía Chávez, Ascensión

Código SNS | 900000004

Estado

Urgencia

Figura D.23: Caso B. Estado posterior, después de recibir el mensaje