UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática

PATRONES DE DISEÑO PARA DESARROLLOS FLEXIBLES DE GUIs

ALEJANDRO HERNÁNDEZ FERRERO

Dirigido por: JOSÉ MANUEL CUADRA TRONCOSO

Curso: 2016-2017: 2ª Convocatoria

PATRONES DE DISEÑO PARA DESARROLLOS FLEXIBLES DE GUIs

Proyecto de Fin de Grado de modalidad oferta general

Realizado por: Alejandro Hernández Ferrero

Dirigido por: José Manuel Cuadra Troncoso

Tribunal calificador

Presidente: D/Dª.

Secretario: D/Dª.

Vocal: D/Dª.

Fecha de lectura y defensa:

Calificación:

Resumen

Este proyecto se centra en el desarrollo de una librería que proporcione soporte para el diseño de interfaces gráficas de usuario, tanto desde la vista de codificación como desde la vista de diseño de un IDE. Esta librería permite diseñar las GUIs modularmente en fragmentos e integrarlos posteriormente, de forma que se favorece la reutilización, y sobre todo hace posible diseñar únicamente el fragmento de GUI correspondiente a las novedades añadidas cuando se extienda una clase para la que ya se haya diseñado una GUI, así como reutilizar las GUIs de los objetos de los que se compone una clase para formar su GUI.

Ofrece tres formas de integrar estos fragmentos: en una sola página, en pestañas y en vista de árbol; así como mecanismos que pueden servir para simplificar la tarea de diseño de GUIs, tales como envolver automáticamente una GUI en una ventana con botones para aceptar o cancelar su contenido y un sistema de paso de mensajes entre GUIs.

En este proyecto también se elabora un programa de prueba que utiliza todos los aspectos de la librería y demuestra su correcto funcionamiento.

Palabras clave

Swing, GUI, interfaz gráfica de usuario, librería, extensible, reutilización, ensamblado, vista de árbol, pestañas, ventana, componente, herencia, IDE, GUI builder, vista de diseño, paleta.

Abstract

This project focuses on the development of a library that provides support for designing graphical user interfaces, both in source and design views. This library allows designing modular GUIs by automatically assembling them, which favours reuse, and specially makes possible designing only the part of GUI corresponding to the new code that extends a class with a previously designed GUI, as well as reusing the GUIs from the objects that compose a class to form its GUI.

It offers three modes of assembling these parts: in a single page, in tabs and in tree view; as well as mechanisms that can serve to simplify the task of GUI design, such as a wrapper of GUIs in a window with buttons to accept or cancel its content and a message passing system.

In this project a test program is also developed to ensure the correct performance of the library.

Keywords

Swing, GUI, graphical user interface, library, extensible, reuse, assembly, tree view, tabs, window, component, inheritance, IDE, GUI builder, design view, palette.

Índice general

Nomenclatura

FPS Imágenes Por Segundo, página 34

GUI Interfaz Gráfica de Usuario, página 1

IDE Entorno de Desarrollo Integrado, página 3

POO Programación Orientada a Objetos, página 1

Índice de figuras

Índice de tablas

Capítulo 1

Introducción general y objetivos

La evolución tan vertiginosa de la informática es debida en gran medida a su capacidad para automatizarse a sí misma. Cada paso dado no solo facilita la creación de aplicaciones comerciales sino que hace posibles desarrollos más ambiciosos, lo que se traduce en un progreso exponencial. La finalidad de este proyecto es ésta, aunque de una forma mucho más modesta: llevar los fundamentos de la filosofía orientada a objetos a la creación de interfaces gráficas de usuario y de esta forma automatizar y simplificar el proceso.

1.1. Motivación y objetivos

La programación orientada a objetos (POO) es el paradigma dominante en la actualidad, sin embargo, algunos de los principios de su filosofía no son llevados al diseño de interfaces gráficas de usuario (GUIs). Si se desea diseñar GUIs para editar un conjunto de clases con antepasados comunes, no existe un mecanismo de herencia que permita que cada una obtenga el fragmento de GUI correspondiente al antepasado común. Por otro lado, si posteriormente se requiere ampliar una de estas clases, un lenguaje orientado a objetos ofrece la gran ventaja de poder extender esta clase con tan solo la nueva ampliación, evitando así la duplicidad del código y favoreciendo la modularidad y reusabilidad; sin embargo no sucede así al intentar ampliar su GUI asociada, ya que si bien es posible crear una subclase de esta GUI, para extenderla sería necesario conocer y acceder a componentes internos, lo que además de ser tedioso atenta contra principios básicos de diseño como el encapsulamiento, el bajo acoplamiento y la alta cohesión. El propósito principal de este proyecto es desarrollar una librería que ofrezca un mecanismo de extensión de GUIs que permita resolver las situaciones anteriores de forma orientada a objetos.

Este mecanismo podrá usarse también de forma más general para acoplar fragmentos de GUIs, lo que permite crear fragmentos independientes que pueden mostrarse por separado o incrustados en otra GUI, e incluso ser reutilizados en todas las GUIs que lo necesiten.

Además, la librería dispondrá de funcionalidades que facilitarán y automatizarán en gran medida

el proceso de creación de GUIs. Dispondrá de un sistema de paso de mensajes que permitirá la comunicación entre GUIs, y de un mecanismo que permitirá envolver cualquier GUI en una ventana predefinida con botones para aceptar o cancelar la edición realizada en ella.

En resumen, se extraen los siguientes objetivos:

1. Permitir crear GUIs para una jerarquía de clases sin duplicar las partes comunes.
2. Poder extender una GUI sin modificarla.
3. Posibilitar la integración de GUIs de objetos contenidos dentro de una clase en la GUI de la clase contenedora.
4. Proporcionar la capacidad de diseñar las GUIs modularmente, pudiéndose ensamblar cualquier combinación de módulos en una única GUI y ser reutilizados para formar otras.
5. Facilitar un sistema de paso de mensajes entre GUIs.
6. Ofrecer un mecanismo que envuelva automáticamente una GUI en una ventana con botones para aceptar o cancelar la edición, de forma que cuando se pulse uno de estos botones se redirija este evento a todos los componentes que contiene.

1.2. Estructura de la memoria

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

1. Introducción general y objetivos
2. Fundamentos teóricos
3. Análisis y diseño
4. Implementación
5. Pruebas
6. Conclusiones y trabajos futuros

En este primer capítulo se han introducido las demandas que motivan el surgimiento de este proyecto. En el segundo capítulo se exponen las nociones básicas de informática necesarias para comprender el resto de la memoria. En el tercero se plantean los requisitos y la estrategia seguida para resolverlos, y en el cuarto se describe cómo llevar a la práctica esa estrategia. En el quinto se presenta el programa de prueba creado y se muestran posibles casos de prueba que se pueden realizar con él. Finalmente en el sexto se evalúan los objetivos conseguidos y los posibles objetivos futuros. Además se incluyen anexos que guían de forma práctica en cómo instalar y usar la librería en NetBeans.

Capítulo 2

Fundamentos teóricos

En este capítulo se introducen las bases técnicas sobre las que se asienta el proyecto, que si bien son comunes a casi la totalidad de los proyectos informáticos, son de vital importancia en éste.

En primer lugar se definen los conceptos clave del paradigma orientado a objetos, que son usados ampliamente a lo largo de este proyecto y cuya filosofía, además de ser usada para la implementación del propio proyecto, es la que se persigue aportar al proceso de diseño de interfaces gráficas de usuario. Posteriormente se introducen los patrones de diseño, imprescindibles para la creación de software de calidad, y a cuya creación para el caso de diseño de GUIs está dedicado este proyecto. A continuación se explora brevemente Swing, que es la librería gráfica de Java y la usada como base en este proyecto. Finalmente se describen los IDEs y el caso particular de NetBeans, para el que se proporciona mayor integración de la librearía creada.

2.1. Programación orientada a objetos

Es un paradigma de programación en el que se modela un problema como abstracciones de las entidades del mundo real. Estas abstracciones constan de unos atributos que definen su estado y de unos métodos que permiten modificar ese estado. Esta es la gran diferencia respecto a la programación estructurada, en la que los datos y las funciones se encuentran separados y sin relación, y a la cual ha ido desplazando desde mediados de los ochenta hasta convertirse en el paradigma de programación dominante. Al principio se modificaron los lenguajes existentes para hacerlos orientados a objetos, pero como no fueron diseñados para tal propósito se producían problemas de compatibilidad y mantenimiento, por lo que posteriormente se crearon lenguajes nuevos con los principios de este paradigma, cuyo máximo exponente es Java.

Para describir los principios de este paradigma surge una nueva terminología, cuyos conceptos principales se definen a continuación [Barnes and Kölling, 2007, Weiss, 2013].

2.1.1. Clase

Es la abstracción de una entidad, en la que se definen los atributos y los métodos de los que consta un tipo de objeto. Se puede instanciar para crear múltiples objetos de una clase, que tendrán la misma estructura en cuanto a métodos y atributos pero diferente valor en éstos. Puede verse como un tipo de objeto del mundo real, por ejemplo un coche, que puede tener atributos como velocidad y objetos como ruedas, y métodos como conducir que modificará el valor de velocidad y operará sobre las ruedas.

2.1.2. Objeto

Instancia concreta de una clase. Sus atributos pueden tener un valor distinto al de otros objetos de la misma clase y representan el estado del objeto concreto en cada momento. Siguiendo el ejemplo anterior un objeto sería un coche concreto, aunque todos los coches tienen el atributo velocidad, tendrá un valor particular en cada uno, y unos objetos ruedas concretos.

2.1.3. Métodos

Operaciones que define un objeto y que permiten acceder a su estado y manipularlo. Por ejemplo acelerar un coche llamaría a métodos de objetos internos como girar las ruedas y como resultado se produciría un cambio en su atributo velocidad.

2.1.4. Atributos

Datos que define una clase y que formarán el estado de cada objeto. En general este estado debe ser interno y modificarse solo a través de los métodos del objeto. Por ejemplo, la velocidad de un coche.

2.1.5. Abstracción

Definir una entidad basándose únicamente en las ideas y cualidades en las que se basa, descartando los detalles particulares. En el ejemplo usado anteriormente, para definir la clase coche se han considerado solo las características esenciales con las que cuenta un coche, obviándose detalles particulares que pueda tener un coche en concreto.

2.1.6. Visibilidad

Permite controlar el alcance de las clases que pueden ver y modificar un atributo o método de un objeto. Se consideran tres tipos de visibilidad:

- Pública: todas las clases.

- Protegida: las clases que se encuentren en el mismo paquete o hereden de ella.
- Privada: solo la propia clase.

2.1.7. Encapsulamiento

Consiste en mostrar al exterior únicamente las operaciones que puede hacer y ocultar cómo las hace internamente. Siguiendo el paralelismo con el mundo real, el coche dispone de un pedal para acelerar y mantiene oculto en su interior todo lo que hace para conseguirlo.

2.1.8. Herencia

Mecanismo que permite crear clases nuevas a partir de otras existentes, de las que se obtiene todo su comportamiento, que puede modificarse sobreescribiendo los métodos heredados o ampliarse creando nuevos. De esta forma se puede crear una jerarquía de clases en la que en cada nivel se definen especializaciones del nivel anterior, lo que permite mantener en un único lugar las características comunes a varias clases y en consecuencia evitar la duplicidad del código y facilitar su mantenimiento. Por ejemplo se podría crear la clase vehículo en la que se definan todos los atributos y operaciones comunes a cualquier tipo de vehículo, como la velocidad, y clases como coche y avión que hereden de ella y extiendan y redefinan tan solo sus particularidades, como por ejemplo modificar la operación mover en el caso del avión para que sea tridimensional.

2.1.9. Polimorfismo

Permite que métodos con el mismo nombre en diferentes clases puedan ser llamados genéricamente sin discernir de qué clase se trata en cada caso. En el mundo real esto se correspondería con estar dentro de una cabina desde la que no se distinga de qué tipo de vehículo se trata pero poder arrancarlo igualmente porque todos disponen del mismo método para ello.

2.2. Patrones de diseño

Describen guías para solucionar tipos de problemas que se presentan frecuentemente en el diseño del software. Un patrón debe resolver el problema de forma efectiva y ser reutilizable en circunstancias variadas, lo que constituye una garantía de la calidad de la solución y favorece el aprendizaje y la difusión de buenas prácticas.

La idea surgió en el contexto de la arquitectura de la necesidad de formalizar el conocimiento práctico. Según su autor, *"Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez"*. Posteriormente, con el auge de la

programación orientada a objetos, se adoptó este concepto para la arquitectura del software debido a la dificultad para instruir a los programadores en este nuevo paradigma.

Los principales patrones seguidos en la realización de este proyecto son [Gamma et al., 2005, Larman, 2003]:

Builder: Separa la construcción de objetos complejos, centralizándola en un único punto.

Singleton: Crea un mecanismo de acceso global a clases en las que solo se requiera una instancia. Útil para clases que controlan el acceso a un recurso único o para clases que deban estar disponibles para todos los objetos del programa, lo que permite a todos ellos poder acceder a la misma instancia sin tener que arrastrar la referencia por toda la aplicación.

Model View Controller: Separa la lógica de la aplicación de la interfaz de usuario. Para ello la arquitectura del sistema se divide en 3 partes, el modelo que implementa la lógica de negocio, la vista que presenta los datos en la interfaz gráfica de usuario, y el controlador que sirve de intermediario entre ellos. De esta forma es posible realizar cambios en la vista de la aplicación sin afectar a su implementación y por lo tanto trabajar en las dos partes paralelamente.

Composite: Construye objetos complejos a partir de otros más simples recursivamente formando una estructura de árbol, y de forma que se permita tratar a los objetos individuales y a las composiciones de objetos uniformemente.

Command: Encapsula una operación en un objeto, de forma que se puedan invocar uniformemente.

Observer: Notifica el cambio de estado de un objeto determinado a todos los objetos interesados, de forma que éstos no tienen que estar preguntando continuamente sobre su estado.

Template Method: Define pasos de un algoritmo en métodos que pueden ser modificados por subclases sin cambiar toda la estructura del algoritmo.

2.3. Interfaces gráficas de usuario

Una interfaz gráfica de usuario (GUI) es la alternativa moderna a la E/S por terminal que permitía a un programa comunicarse con el usuario. Para proporcionar la entrada se utilizan elementos gráficos como listas de selección, botones, casillas de verificación y campos de texto; y la salida se puede realizar escribiendo en campos de texto o dibujando gráficos.

En sus orígenes Java contaba con una biblioteca de componentes gráficos llamada AWT. Esta biblioteca utilizaba los componentes nativos de cada sistema operativo, lo que iba en contra de la filosofía portable de Java y además ocasionaba graves problemas de rendimiento debido a los elevados tiempos de respuesta en la comunicación con el sistema operativo. Por estas razones la biblioteca no tuvo éxito y fue extendida con Swing. En Swing los componentes son mostrados y controlados

por código Java y no requieren reservar recursos nativos del sistema operativo, razón por la que son llamados componentes ligeros, y de esta forma reducen el tiempo de respuesta y son independientes de la plataforma.

AWT/Swing está organizado siguiendo una jerarquía de clases, cuya estructura principal se muestra en la figura 2.1, en la que los componentes pesados de AWT se muestran en gris y los componentes ligeros de Swing en azul; y sus elementos clave se describen a continuación.

Component: Representa algo que tiene una posición, un tamaño, que se puede dibujar en la pantalla y que recibe sucesos de entrada.

Container: Representa un componente que puede contener otros componentes.

JFrame: Ventana que tiene borde y que puede tener barra de menú. Una aplicación con interfaz gráfica debe tener esta clase o una subclase de ésta como contenedor más externo.

JDialog: Ventana utilizada para mostrar diálogos.

JPanel: Es la implementación más simple de un contenedor y se usa para agrupar un conjunto de componentes con un cierto orden.

Como puede apreciarse, esta estructura se ha formado aplicando el patrón Composite (ver sección 2.2) y por lo tanto permite insertar unos componentes dentro de otros recursivamente para formar una GUI. Para distribuir los componentes dentro de un contenedor se puede elegir un layout manager, que se encarga automáticamente de colocar cada componente y repartir el espacio entre ellos [Walrath et al., 2004].

2.4. Entornos de desarrollo integrado

Aunque la aparición de los lenguajes de alto nivel ya supuso una gran automatización y simplificación en la creación de aplicaciones informáticas, esta tarea sigue resultando tediosa y problemática de realizar escribiendo en un archivo de texto.

Por esta razón surgieron los entornos de desarrollo integrado (IDEs), herramientas que pretenden asistir al programador en esta tarea en todos los aspectos posibles, como detectando errores léxicos y sintácticos en tiempo real, permitiendo depurar el código ejecutándose paso a paso, gestión automática de la compilación, e infinidad de utilidades más.

Sin embargo, crear una GUI de esta forma sigue requiriendo demasiado esfuerzo, por lo que los IDEs han incorporado también herramientas que permiten diseñar GUIs de forma gráfica, ahorrando gran parte de complejidad y esfuerzo. Ya que el diseño de GUIs es en cierto modo una tarea artística, para estas clases los IDEs permiten, además de la vista del código típica, una vista de diseño en la que se dispone de un lienzo y una paleta. En la paleta el diseñador puede elegir entre cualquier tipo

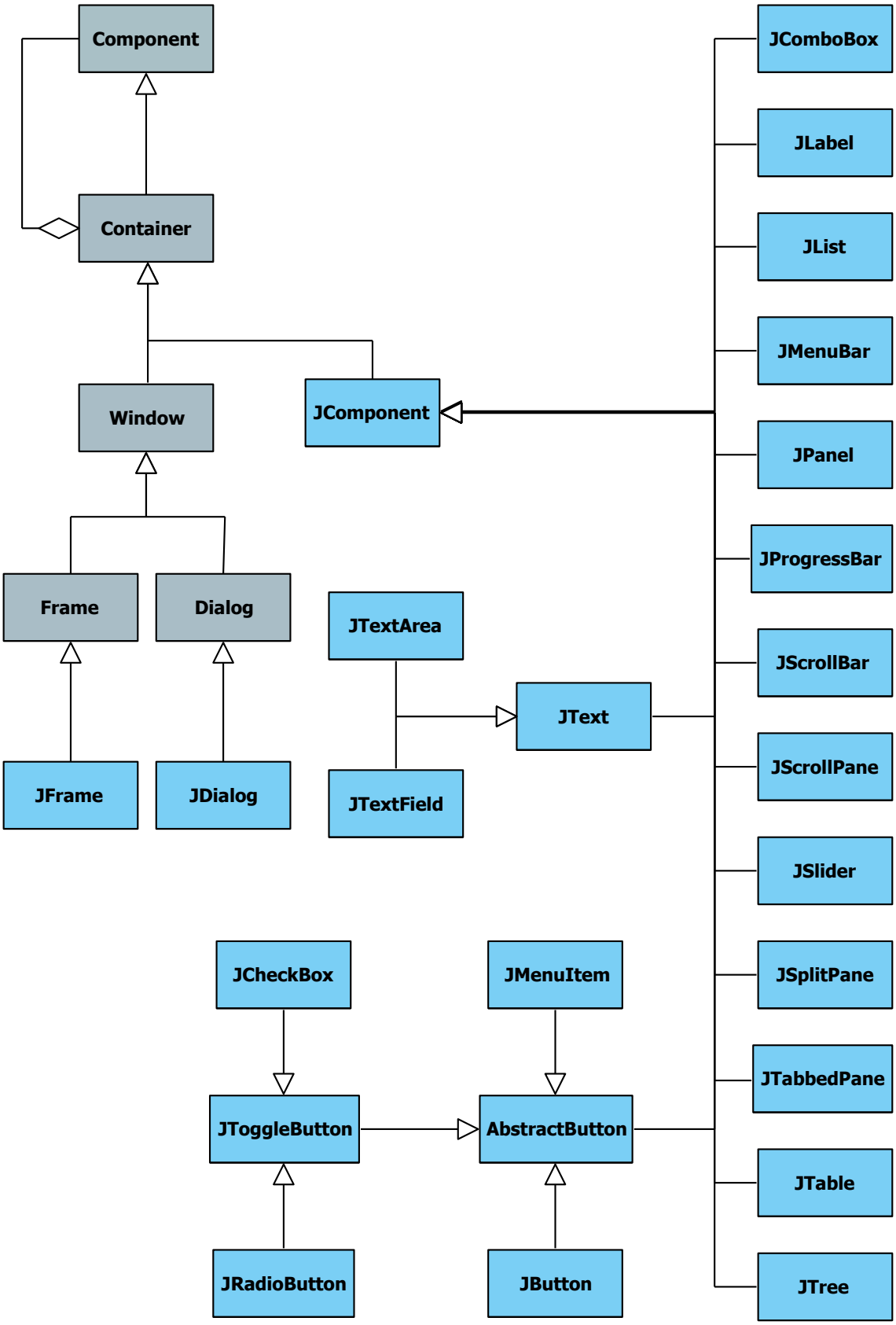


Figura 2.1: Estructura principal de Swing.

de componente disponible, arrastrarlo al lienzo, y en él colocarlo dentro del contenedor que desee. De esta forma, además de simplificarse el proceso, se puede ver en tiempo real el aspecto de la GUI y tomar decisiones creativas en consecuencia. En la figura 2.2 se muestra la vista de diseño del IDE NetBeans.

Por esa razón la librería de este proyecto se diseña para ser también compatible con la vista de diseño del IDE y que sus componentes puedan ser usados desde la paleta como cualquier otro componente de forma transparente. En concreto, la librería está pensada para ser usada únicamente en el IDE NetBeans y por lo tanto solo se centra en ofrecer compatibilidad con éste, para el que se diseña además un plugin que la integra automáticamente en él (ver anexo A).

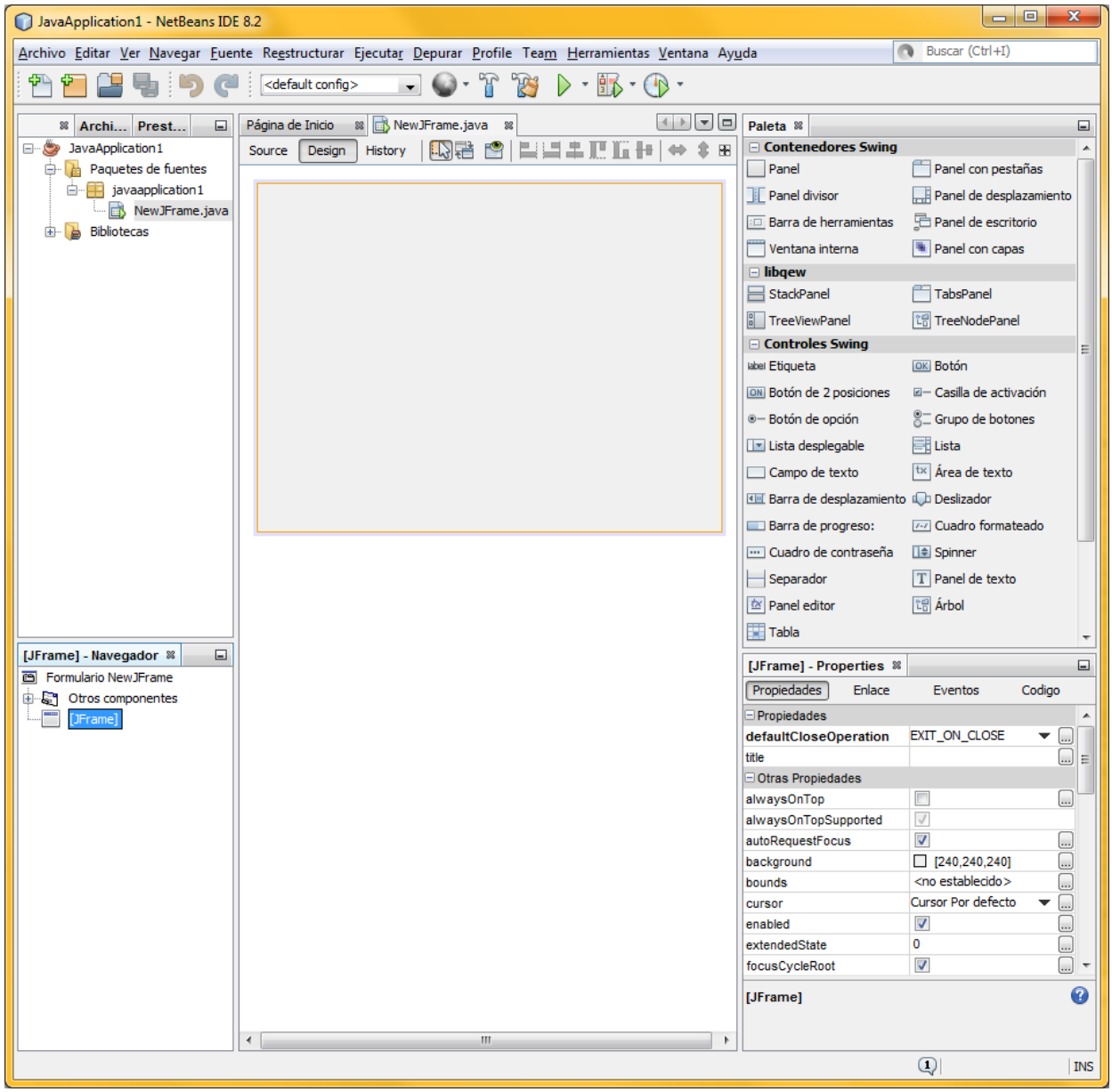


Figura 2.2: Vista de diseño en NetBeans.

Capítulo 3

Análisis y diseño

Este capítulo se corresponde con las fases de análisis y diseño de ingeniería de software en este proyecto. En primer lugar se enumeran los requisitos que deberán cumplirse, a continuación se describen brevemente los casos de uso típicos que se esperan de la librería, posteriormente se detalla la planificación realizada para la ejecución del proyecto, y finalmente se muestra la arquitectura diseñada como resultado.

3.1. Requisitos

Debido a que se trata de un proyecto académico, no ha sido necesario recabar requisitos de un cliente sino que ya venían estipulados en la guía didáctica. No obstante, no se han tomado directamente sino que se han modificado en algunos casos según se ha creído más conveniente (en la sección 6.1 se detallan los cambios realizados).

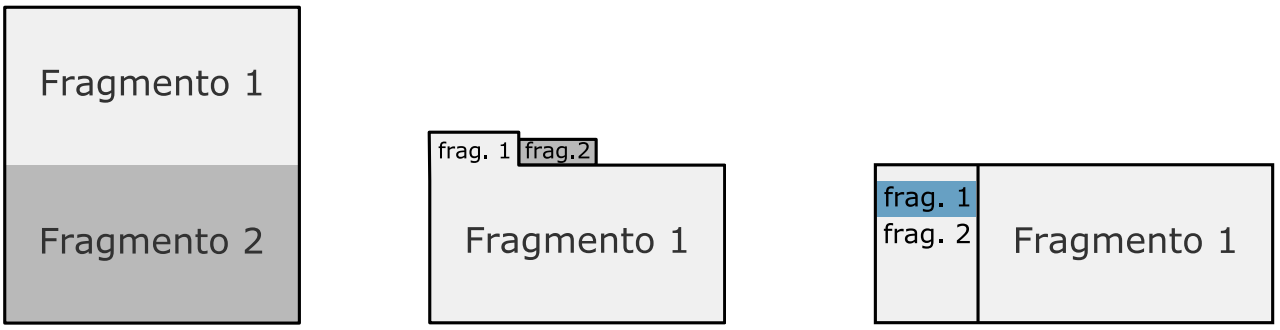
1. La librería será implementada en forma de biblioteca de clases usando el lenguaje Java y la librería gráfica Swing.

2. Se ofrecerán tres tipos diferentes de GUIs extensibles dependiendo del modo en el que se desee que acople los fragmentos, que se muestran en la figura 3.1:

   a) En una sola página: los fragmentos son acoplados verticalmente en la GUI, expandiéndose horizontalmente si tienen menor anchura que algún otro.

   b) En pestañas: se muestra una barra con los nombres de cada fragmento que permite elegir el fragmento a mostrar en cada momento.

   c) En vista de árbol: la GUI resultante consta de dos partes, la de la izquierda es permanente y muestra la jerarquía de fragmentos mediante sus nombres y permite al usuario navegar por ella y elegir uno, que es mostrado entonces en la parte de la derecha.



(a) En una sola página. (b) En pestañas. (c) En vista de árbol.

Figura 3.1: Tipos de GUIs extensibles.

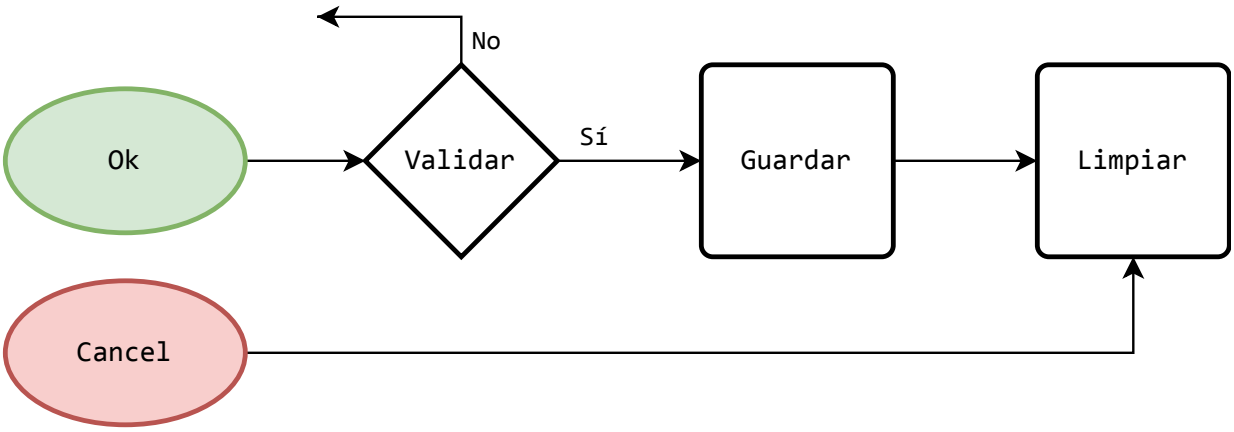


Figura 3.2: Diagrama de eventos que desencadena cada botón.

3. Estas GUIs extensibles deben poder usarse también como cualquier otro contenedor de Swing para diseñar GUIs, pudiendo añadirles cualquier componente y ser añadidos a cualquier contenedor de forma transparente.

4. También deben ser compatibles con la vista de diseño del IDE, apareciendo en la paleta y pudiéndose arrastrar al lienzo y usarse como cualquier otro componente de forma transparente.

5. Se debe mantener la estética de los fragmentos acoplados, preservando la distribución de los componentes y su comportamiento frente al redimensionamiento y sin acumular márgenes.

6. Se proporcionará un mecanismo que permitirá envolver cualquier GUI en una ventana con botones para aceptar o cancelar la edición de los datos. Estos botones desencadenarán una serie de eventos que se muestran en la figura 3.2 y serán propagados automáticamente a todos los componentes de la ventana, pudiendo el programador especificar las acciones que se realizarán cuando se reciban.

7. Estas acciones serán realizadas de forma atómica. No se desencadenará el siguiente evento hasta que el anterior no se haya resuelto en todos los componentes de la ventana, es decir, no se guardará nada hasta que no se hayan realizado todas las validaciones en todos los componentes

satisfactoriamente, y si una validación da resultado negativo se abortará el proceso sin realizar ningún guardado.

8. Debe detectarse si la GUI no entra en la pantalla y mostrar un mensaje de error en tal caso.

9. Se pondrá a disposición del programador un sistema de paso de mensajes con el fin de facilitar la comunicación entre GUIs independientes. En concreto, se dispondrá de dos modos de paso de mensajes:

   a) Síncrono: el mensaje será enviado directamente a todos los listeners que estén registrados para recibir el tipo de mensaje, por lo que será recibido al instante sin quedar el mensaje guardado en memoria.

   b) Asíncrono: el mensaje es almacenado en memoria hasta que se borre y se podrá acceder a él durante ese periodo.

10. Se debe crear un programa de prueba con interfaz gráfica para editar atributos de una jerarquía de clases, de forma que se diseñe para cada clase una GUI que edite solo los atributos nuevos que define la clase, y se acople esa GUI con la de la superclase para editar también los heredados. Debe haber al menos 2 clases que hereden de una común para mostrar la reutilización de la GUI de la superclase en ambas. Además debe usarse el mecanismo de paso de mensajes para sincronizar componentes pertenecientes a distintas GUIs.

3.2. Casos de uso

Existen dos casos de uso principales dependiendo del rol del programador que use la librería:

- El diseñador de la GUI tendrá disponibles en su paleta los componentes de la librería y los podrá arrastrar al lienzo para insertarlos en otros componentes y podrá insertarles componentes y editar sus propiedades según crea conveniente como con cualquier otro componente de Swing utilizando únicamente la vista de diseño.

- El programador de aplicaciones podrá usar las clases de la librería para ensamblar fragmentos de GUIs ya diseñadas en una ventana con botones de aceptación y cancelación de la edición de los datos y escribir el código para validar, guardar y limpiar los datos de la GUI cuando se pulse el botón correspondiente, así como usar el mecanismo de paso de mensajes para comunicar las GUIs que desee. Además, al igual que cualquier componente de Swing, los componentes de la librería ofrecen métodos para que el programador pueda diseñar la GUI completamente desde la vista de codificación si así lo desea.

El diagrama de la figura 3.3 muestra mediante lenguaje natural un caso de uso típico de la librería.

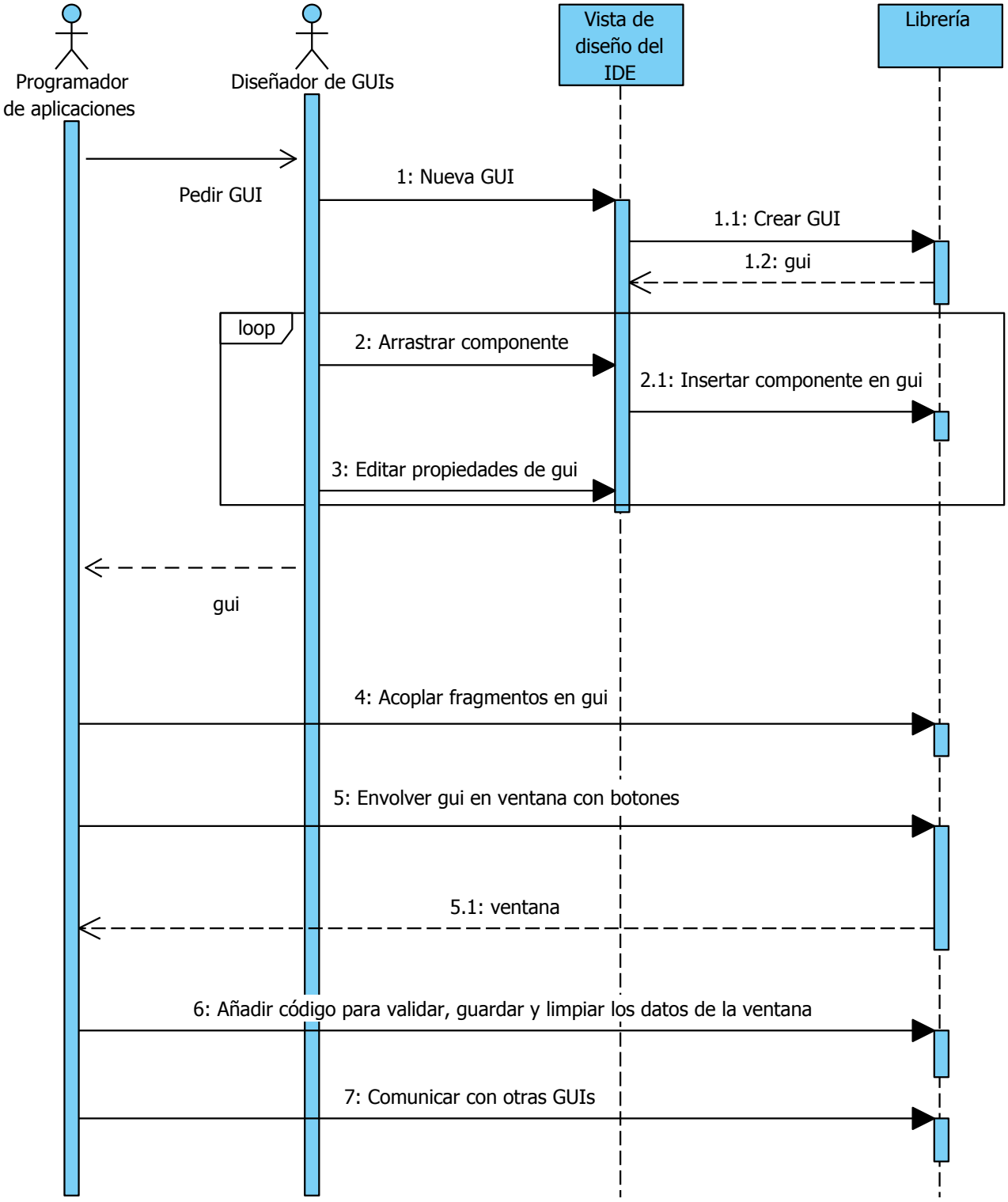


Figura 3.3: Caso de uso típico de la librería.

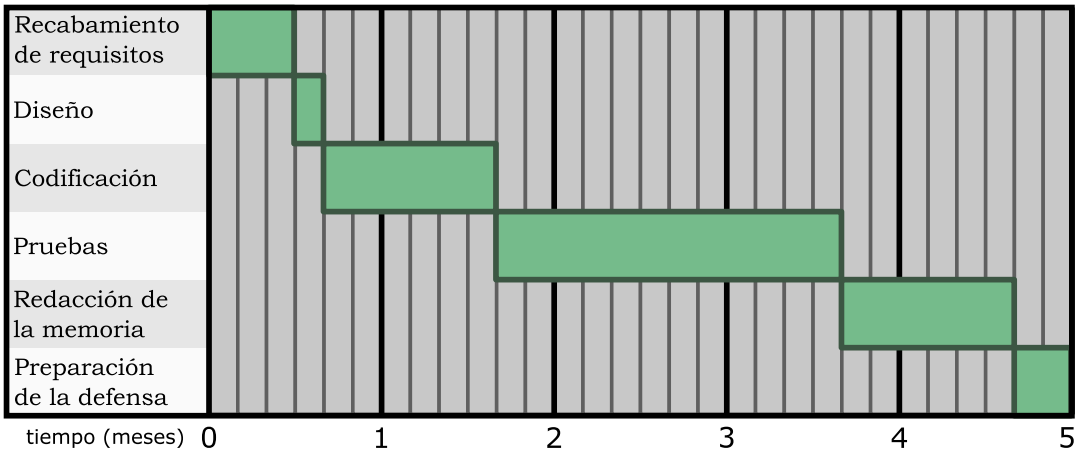


Figura 3.4: Cronograma.

3.3. Planificación

Una vez analizados los requisitos se procede a una primera planificación en la que se asigna un tiempo de referencia para cada tarea, y una vez concluido el proyecto se utilizan los datos reales para estimar el esfuerzo requerido y compararlo con la estimación realizada antes de conocerlos.

3.3.1. Calendarización del proyecto

La duración máxima del proyecto está limitada por el plazo para la defensa del mismo, que es en torno a 5 meses. Por lo tanto, se considerará este tiempo y se dividirá según la regla del 40-20-40 [Pressman, 2010, pág. 626] entre análisis y diseño (2 meses), codificación (1 mes) y pruebas (2 meses). Dentro de esta distribución, las tareas específicas de este proyecto como la escritura de esta memoria y la defensa se han contabilizado dentro de la parte de análisis y diseño, y la creación de la aplicación de prueba en la de pruebas. Por otra parte el recabamiento de requisitos en este proyecto se correspondería con leer la guía didáctica y entender qué se pide. En la figura 3.4 se muestra un cronograma con la planificación de las tareas del proyecto.

3.3.2. Estimación de costes

Una vez concluido el proyecto se ha contabilizado el número total de líneas de código para estimar los recursos reales requeridos usando el modelo COCOMO [Boehm, 1981] mediante la herramienta SLOCCount y los resultados se muestran en la tabla 3.1. Aunque se puede observar que la discrepancia con la planificación original es grande (el esfuerzo es más del doble), si se analizan los datos relativos a únicamente la librería y se considera que una aplicación de prueba no debería requerir más esfuerzo que la propia librería se puede concluir que las estimaciones son coherentes y que el sobrecoste es debido a una aplicación de prueba demasiado elaborada.

	Librería	Aplicación de prueba	Total
Líneas de código	1174	4160	5334
Esfuerzo estimado (personas-mes)	2,84	10,72	13,92
Tiempo estimado (meses)	3,72	6,16	6,8
Número medio estimado de desarrolladores	0,76	1,74	2,05
Coste estimado	31974 $	120695 $	156692 $

Tabla 3.1: Resultados del modelo COCOMO.

3.4. Diseño

La arquitectura de la librería diseñada para satisfacer los anteriores requisitos se muestra en el diagrama de clases simplificado de la figura 3.5, en el que solo se muestran los métodos públicos que define cada clase y se omiten sus parámetros, tipos de retorno y los constructores por motivos de espacio; y a continuación se describe la finalidad de cada una de estas clases.

3.4.1. Extensible

Interfaz base para todos los componentes gráficos de la librería que permiten acoplar fragmentos de GUI. Su mayor ventaja es que permite incrustar fragmentos en la GUI de forma genérica sin tener que preocuparse de si está envuelta en una ventana con botones.

3.4.2. ExtensiblePanel

Clase base para las GUIs extensibles de la librería. Implementa las funcionalidades comunes a todos los tipos de GUIs extensibles como la propagación de los eventos de validar, guardar y limpiar, e insertar el código personalizado que se desee ejecutar cada vez que se produzcan estos eventos.

3.4.3. StackPanel

GUI extensible en una sola página (requisito 2*a*). Permite apilar fragmentos de GUI de forma vertical y distribuye la altura total de la GUI entre cada fragmento de forma proporcional a la altura original de cada uno, incluso si se redimensiona (requisito 5); es decir, si la GUI aumenta su altura cada fragmento recibirá mayor parte de este aumento cuanto más alto fuera originalmente respecto a los demás. Horizontalmente todos los fragmentos se expanden hasta rellenar el tamaño de la GUI. Por motivos de diseño esta clase no permite modificar su layout y considerará como fragmento a acoplar cualquier componente que se le añada, por lo que si se desea añadirle componentes de forma convencional y modificar su layout se deberá insertarle un contenedor (bien añadiéndolo normalmente o mediante la propiedad content) e insertar los componentes en ese contenedor.

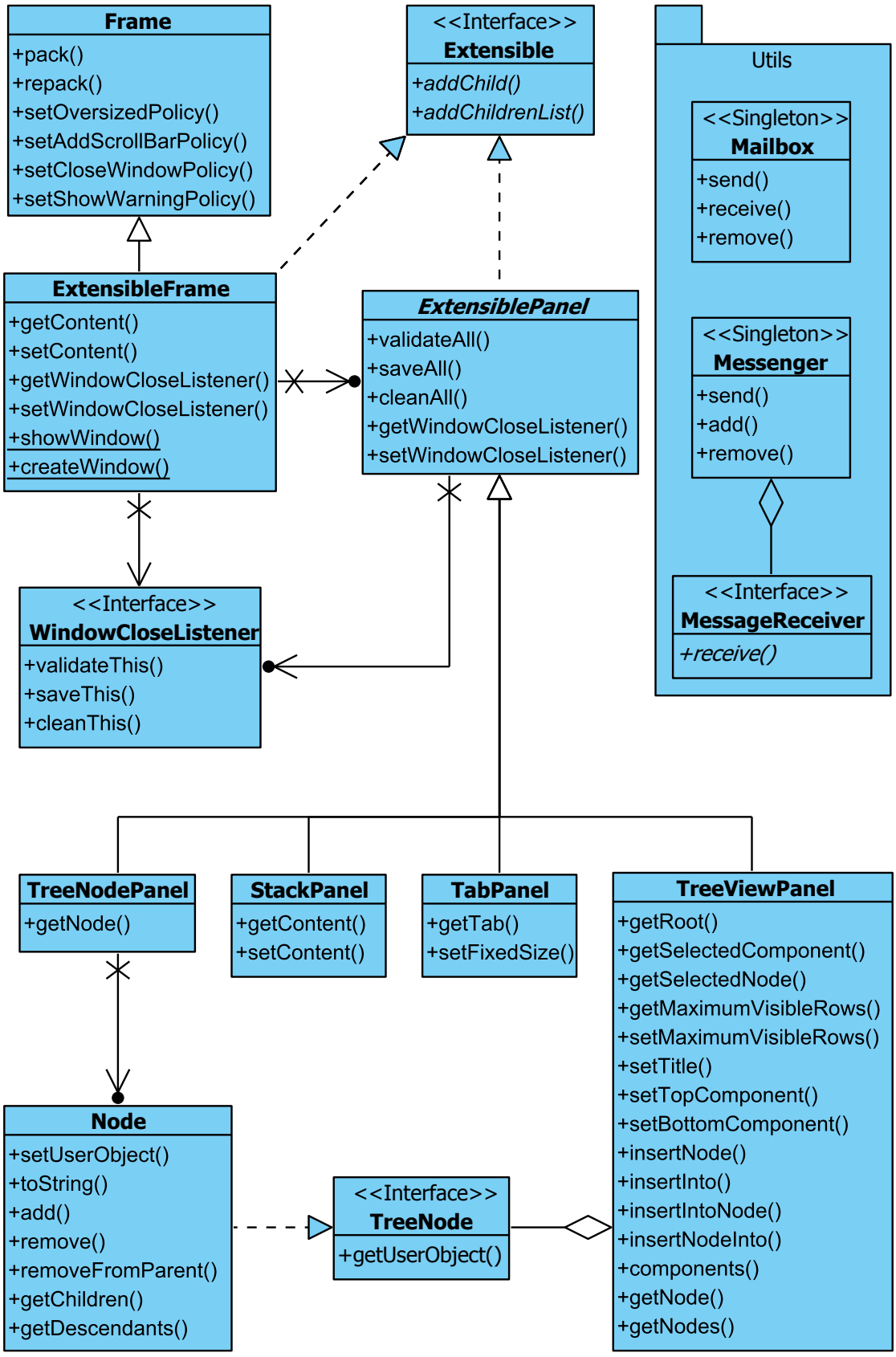


Figura 3.5: Diagrama de clases de la librería.

3.4.4. TabsPanel

GUI extensible en pestañas (requisito 2*b*). Permite elegir el fragmento a mostrar en cada momento mediante una barra superior que contiene los nombres de todos los fragmentos disponibles. Swing ya proporciona una clase con esta funcionalidad por lo que se podría usar directamente, sin embargo la librería ofrecerá algunas funcionalidades extra. En la librería el nombre de cada pestaña se tomará automáticamente del nombre del fragmento que contiene. Además se modificará el modo en el que la GUI se ajusta al contenido de las pestañas: en Swing el tamaño de la GUI se ajusta al de la pestaña más grande y no cambia al seleccionar las demás, no permitiendo al usuario redimensionar ninguna pestaña por debajo del tamaño mínimo de la más grande; mientras que en la librería el tamaño de la GUI se ajustará al de la pestaña seleccionada y se autoajustará cada vez que se cambie de pestaña. De esta forma se respetan las dimensiones de las GUIs acopladas y por lo tanto su estética (requisito 5), pero debe tenerse en cuenta que también hará cambiar de tamaño la ventana dinámicamente sin respetar el redimensionamiento que pueda haber hecho el usuario, por lo que si no se desea tal efecto secundario se puede desactivar este modo y usar el de Swing mediante la propiedad *fixedSize*.

3.4.5. TreeViewPanel

GUI extensible en vista de árbol (requisito 2*c*). Permite navegar por una jerarquía de fragmentos en la parte izquierda y seleccionar una de ellas, que se mostrará en la parte derecha. Aunque no forma parte de los requerimentos, permitirá modificar la jerarquía de nodos de forma dinámica, pudiendo insertar y eliminar nodos de la jerarquía en tiempo de ejecución. El tamaño de la GUI se autoajustará para adaptarse a su contenido cada vez que un nodo cambie, se seleccione otro nodo, o se expandan, colapsen, inserten o eliminen nodos en la jerarquía. Aunque permite cualquier *MutableTreeNode* de Swing como nodo, debido a que a esta interfaz le falta un método para obtener su contenido, solo será visible el contenido de nodos que implementen la interfaz de la librería creada al respecto: *TreeNode*, o bien sean subclases de la clase de Swing *DefaultMutableTreeNode*. Además se han diseñado dos clases listas para ser usadas como nodos: *Node* y *TreeNodePanel*, cuya finalidad se describe a continuación.

3.4.6. Node

Permite envolver un componente en un nodo para ser parte de la jerarquía de un *TreeViewPanel*. Se ofrece como opción principal para crear la jerarquía de nodos de forma rápida, sencilla y transparente, ya que además de toda la funcionalidad otorgada por *DefaultMutableTreeNode*, se encarga de automáticamente mostrar y actualizar su nombre en la jerarquía de nodos a partir del nombre del componente que contiene, así como de avisar al *TreeViewPanel* que lo contenga cuando se le inserten o eliminen nodos para que se actualice la GUI, lo que libera al programador de todas estas complejas y oscuras tareas.

3.4.7. TreeNodePanel

Su objetivo es proporcionar un mecanismo para construir la jerarquía de nodos con sus GUIs asociadas completamente desde la vista de diseño de un IDE. Puede usarse como un *JPanel* normal para diseñar la GUI correspondiente al nodo al que representa, pero los componentes que implementen *MutableTreeNode* o sean instancias de esta propia clase no se añadirán de forma visual sino que se añadirán como nodos hijo, pudiendo crear de esta forma una jerarquía con cualquier nivel de profundidad. Una vez creada, puede añadirse directamente a un *TreeViewPanel*, a otro nodo como hija mediante su nodo interno, o incluso ser envuelta en una ventana con botones, en cuyo caso se mostrará su contenido con vista de árbol automáticamente.

3.4.8. Frame

Ventana que proporciona un control automático de su tamaño. Se encarga de establecer su tamaño mínimo y máximo automáticamente a partir del de su contenido. Además detecta si su tamaño excede al de la pantalla y permite mostrar un mensaje de error informando de este hecho (requisito 8), añadir barras de scroll o cerrar la ventana, según se desee más conveniente.

3.4.9. ExtensibleFrame

Ventana que envuelve con botones de aceptar y cancelar cualquier tipo de GUI y se encarga de propagar los eventos que desencadenan estos botones (ver figura 3.2) a todos los componentes que contiene (requisito 6). También permite acoplar fragmentos de GUI, que serán incrustados en la GUI extensible establecida como contenido o en una sola página si no se estableció ninguna.

3.4.10. WindowCloseListener

Interfaz que permite al programador especificar las acciones que desea que se lleven a cabo para validar, guardar y limpiar el contenido del componente cuando se reciba el evento correspondiente (requisito 6). Para ello, el programador tan solo debe implementar esta interfaz en los componentes que desee y si forman parte de una *ExtensibleFrame* el código será asociado automáticamente con los botones de la ventana. En las GUIs extensibles también se ofrece la alternativa de especificar las acciones pasándoles un objeto que implemente esta interfaz mediante *setWindowCloseListener*, lo cual puede ser beneficioso en muchos escenarios como por ejemplo para reutilizar el objeto o recibirlo externamente. Ambos métodos pueden ser usados simultáneamente y se garantiza que en cada componente siempre se realizarán las acciones pasadas como objeto posteriormente a las implementadas como interfaz en el componente; esto es particularmente útil en el caso de las ventanas, ya que en este caso uno se ejecutaría antes de propagarse y otro al terminar de propagarse.

3.4.11. Mailbox

Mecanismo de paso de mensajes asíncrono (requisito 9*b*). Permite almacenar mensajes asociados a una clave y ser recuperados por otros objetos en cualquier momento mediante esa clave.

3.4.12. Messenger

Mecanismo de paso de mensajes síncrono (requisito 9*a*). Permite asociar listeners a una clave, que serán notificados siempre que se envíe un mensaje asociado a esa clave.

Capítulo 4

Implementación

Este capítulo se dedica a describir las soluciones seguidas para implementar las clases diseñadas en la sección 3.4 del capítulo anterior.

4.1. ExtensiblePanel

Todas las GUIs extensibles de la librería extenderán a *JPanel* de Swing. De esta forma serán compatibles transparentemente con Swing (requisito 3) y se heredarán todas sus funcionalidades, siendo solo necesario añadir las particularidades de cada GUI.

En esta clase se implementarán las funcionalidades compartidas por todas las GUIs extensibles, principalmente la propagación de eventos de validar, guardar y limpiar. Para propagar cada uno de estos eventos se sigue el mismo esquema algorítmico:

1. Si la propia clase implementa la interfaz *WindowCloseListener* se llama al método correspondiente.

2. Si se ha establecido un objeto mediante *setWindowCloseListener* se llama al método correspondiente del objeto.

3. Para cada componente hijo:

    a) Si es también un objeto de esta clase se ejecuta recursivamente este algoritmo en este hijo y se vuelve al paso 3 con el siguiente hijo.

    b) Si el componente implementa la interfaz *WindowCloseListener* se llama al método correspondiente.

    c) Si el componente es un contenedor se realiza el paso 3 para cada componente que contiene.

De esta forma, los métodos *validateAll*, *saveAll*, y *cleanAll* llamarán a los métodos *validateThis*, *saveThis* y *cleanThis*, respectivamente, de todos los componentes que contiene la GUI que contengan o implementen la interfaz *WindowCloseListener*.

4.2. StackPanel

Para distribuir los fragmentos verticalmente tan solo será necesario configurar un layout de Swing (ver A Visual Guide to Layout Managers). En principio podría bastar con crear un layout de tipo *BoxLayout* con orientación en el eje y, y no sería necesario hacer nada más; sin embargo este layout no se comporta de acuerdo a los requisitos frente a redimensionados en algunas ocasiones, ya que distribuye el nuevo espacio solo entre los componentes expansibles y no entre los fragmentos proporcionalmente, y además tampoco los expande horizontalmente, por lo que se ha optado por un layout más sofisticado como *GridBagLayout*.

Este layout usa un sistema de pesos para expandir los fragmentos en el espacio por lo que si a cada fragmento se le otorga un peso vertical proporcional a su altura, y un peso horizontal positivo que los expanda horizontalmente se obtendrá el comportamiento deseado. Hay que tener en cuenta que un fragmento podría cambiar de tamaño dinámicamente por lo que será necesario actualizar el valor de su peso siempre que esto ocurra. Para ello se sobrescribirá el método *invalidate* de la GUI, que es llamado siempre que cambie el contenido de la GUI, y en él se recalcularán los pesos.

En la figura 5.3 de la sección 5.2.1 se muestra un ejemplo de GUI de esta clase.

4.3. TabsPanel

Swing ya contiene una GUI con pestañas por lo que se podría extender esta clase añadiéndole las funcionalidades que le falten, sin embargo la clase base *ExtensiblePanel* ya extiende a *JPanel* y Java no permite la herencia múltiple por lo que con el fin de tener todas las GUIs de la librería como subclases de ésta se optará por otra estrategia.

Se encapsulará la clase *JTabbedPane* de Swing (ver How to Use Tabbed Panes) y esta clase actuará tan solo como un envoltorio redireccionando los componentes que se le inserten a esta clase interna y añadiendo alguna funcionalidad extra en el proceso.

El nombre que se muestre en las pestañas será en todo momento el del componente que contiene, incluso si se cambia dinámicamente en tiempo de ejecución. Para ello se añadirá un listener al componente que escuchará cambios en su propiedad nombre y cuando ésta cambie se actualizará el título de la pestaña.

Para conseguir que el tamaño de la GUI se adapte dinámicamente al de la pestaña seleccionada en todo momento será necesario modificar los métodos que devuelven el tamaño de la GUI, ya que *JTabbedPane* sigue la política de establecer para todas las pestañas el tamaño de la más grande.

Tan solo sería necesario devolver el tamaño del componente seleccionado en ese momento y sumarle el tamaño de la barra de pestañas, sin embargo el tamaño de esta barra dependerá del *Look and Feel (ver* How to Set the Look and Feel) utilizado por lo que no se puede saber de antemano. Una solución rápida sería calcular el tamaño de la GUI con una pestaña vacía, con lo que se obtendría la altura de esta barra; sin embargo, la anchura de la GUI dependerá del componente seleccionado en ese momento y dependiendo de esta anchura la barra puede necesitar distinto número de filas para mostrarse, con lo que cambiaría su altura, por lo tanto será necesario recalcular su altura para cada pestaña. Pero como se ha indicado antes *JTabbedPane* no tiene en cuenta el tamaño de la pestaña actual, por lo tanto será necesario "engañar" a esta clase: se envolverá cada pestaña en una clase que dirá que su tamaño es 0 si no está seleccionada, y altura 0 y anchura real del componente si lo está. De esta forma *JTabbedPane* tendrá en todo momento como tamaño el de la barra de pestañas y la GUI podrá calcular su tamaño real sumándoselo al del componente seleccionado. La figura 5.4 de la sección 5.2.2 muestra este comportamiento.

Para dar la opción de desactivar el comportamiento anterior se crea la propiedad *fixedSize*, que si está establecida a *true* evita que se realicen todos esos cálculos y simplemente obtiene su tamaño del *JTabbedPane* interno.

4.4. TreeViewPanel

Para crear esta GUI se insertarán una serie de componentes en el *JPanel* base. Para dividir el espacio entre el componente seleccionado y la vista de árbol se usará un *JSplitPane* en modo horizontal (ver How to Use Split Panes).

En la parte izquierda iría un *JTree*, que mostraría la jerarquía de nodos, envuelto en un *JScroll-Pane*, que permite visualizarlo cuando sea demasiado grande desplazándose mediante barras de scroll (ver How to Use Trees, How to Use Scroll Panes); pero se ha considerado que probablemente en algunas ocasiones sea necesario personalizar esta vista, por ejemplo para añadirle un título arriba o botones para modificar los nodos abajo, por lo que se ha creído conveniente permitir al programador insertar los componentes que desee en estas partes. Para ello, en vez de insertarse el árbol en la parte izquierda, se insertará un *JPanel* con *BorderLayout* (ver How to Use BorderLayout), y el árbol en el centro de este *JPanel*, y se crearán métodos que permitirán al programador especificar qué componentes desea establecer en la parte de arriba y de abajo de este *JPanel*: *setTopComponent* y *setBottomComponent*. Debe tenerse en cuenta que solo se puede establecer un componente en cada posición, pero si se necesita insertar más tan solo habrá que agruparlos en un *JPanel* y establecer este *JPanel*, pudiendo de esta forma personalizar la GUI tanto como se desee. Además para el caso típico del título se ofrece el método *setTitle* que se encarga de establecer los componentes adecuados en la parte superior a partir de tan solo una cadena de caracteres.

En la parte derecha se insertará el componente correspondiente al nodo seleccionado, y cada vez que se cambie de selección se cambiará por el nuevo. Para ello se añadirá un TreeSelectionListener

al árbol, que será avisado cada vez que cambie la selección (ver How to Write a Tree Selection Listener).

La GUI se encargará automáticamente de generar estos nodos en el árbol a partir de los componentes insertados de forma convencional, mostrando como nombre del nodo el nombre del componente, pero de esta forma es imposible definir la estructura del árbol, por lo que para tal propósito se proporcionan métodos que permiten insertar nodos. Estos nodos pueden ser cualquier clase que implemente la interfaz *MutableTreeNode* de Swing, no obstante, debido a que a esta interfaz le falta un método para obtener su contenido, se ha definido una interfaz que la amplía con el método faltante, *TreeNode*, y solo será visible el contenido de nodos que la implementen o bien sean subclases de *DefaultMutableTreeNode* de Swing.

Para facilitar este proceso se proporciona la clase *Node*, que ya implementa esta interfaz y libera al programador de esta tarea, que solo tendrá que pasarle el componente en el constructor para crear cada nodo; y entonces insertar unos nodos dentro de otros para crear la estructura que desee, para finalmente insertar el nodo raíz de esta estructura en la GUI.

Aunque con lo expuesto anteriormente ya se cumplirían los objetivos básicos del proyecto, se implementará esta GUI de forma que permita modificar el árbol de nodos dinámicamente en tiempo de ejecución. Para ello será necesario añadir un modelo al árbol, y un *TreeModelListener* al modelo (ver How to Write a Tree Model Listener), al que se avisará cada vez que cambie algo en el árbol para recalcular sus dimensiones y actualizar la GUI.

Cuando un nodo sea borrado debe controlarse también que no sea el que está seleccionado, para en tal caso seleccionar otro. Se intentará siempre seleccionar el anterior pero si no tiene anterior porque era el primero se intentará el siguiente. Si era el único nodo del árbol y por lo tanto quedará vacío, se seleccionará la raíz del árbol, que está oculta y cuyo contenido es un *JPanel* vacío.

Debe tenerse en cuenta que para que la GUI sea avisada de estos eventos por el modelo, el propio modelo debe tener constancia de ellos. La GUI puede conocer cuando se insertan y eliminan nodos a través de ella pero no cuando se hace a través de los mismos nodos. Los nodos proporcionados en la librería (*Node* y *TreeNodePanel*) ya se encargan de comunicarse adecuadamente con la GUI cuando son modificados pero si se usan otras clases de nodos deberán modificarse a través de los métodos de la GUI para que ésta se actualice correctamente.

Siempre que haya espacio suficiente en pantalla se intentará mostrar el árbol entero hasta un número límite de nodos que puede personalizar el diseñador mediante la propiedad *maximumVisibleRows*. Por lo tanto se debe reajustar el tamaño de la GUI cada vez que se expanda un nodo, lo cual se consigue añadiendo un *TreeExpansionListener* al árbol (ver How to Write a Tree Expansion Listener).

Puede verse un ejemplo de funcionamiento de esta clase en la figura 5.5 de la sección 5.2.5.

4.5. Node

Se extenderá la implementación base realizada en Swing, *DefaultMutableTreeNode*, con mecanismos de detección automática de cambio en los nodos.

Para que la GUI perciba cambios en los nodos es necesario que el modelo tenga constancia de ellos, sin embargo, el modelo está asociado a la GUI y no a los nodos, ya que los nodos son independientes, y por lo tanto no existe una conexión directa.

Para crear esta conexión se aprovechará la circunstancia de que la raíz del árbol sí está ligada permanentemente a la GUI, por lo que podrá tener acceso al modelo. De esta forma los nodos pueden buscar la raíz ascendentemente y si se trata de la raíz de la GUI acceder al modelo a través de ella. Por el contrario, si no se trata de esta raíz significa que el nodo no pertenece a la GUI en ese momento por lo que tampoco hay que notificar a ningún modelo, ya que en el momento en el que esta estructura de nodos pase a pertenecer al árbol de la GUI será porque se inserte en un nodo que sí pertenece, y que por lo tanto sí notificará al modelo.

Una vez disponible dicha conexión, se podrán sobrescribir los métodos para añadir y borrar hijos, para que lo hagan a través del modelo si es posible.

Además el nombre que mostrará el nodo será adoptado en todo momento de el del componente que contiene, incluso si este nombre cambia en tiempo de ejecución. Para ello se le añadirá un listener al componente que avisará cada vez que cambie su propiedad nombre para que el nodo notifique a la GUI de que el nodo cambió y se actualice.

4.6. TreeNodePanel

Aunque esta GUI no forma parte de los tipos de GUIs extensibles requeridos originalmente, podría considerarse como una GUI con vista de árbol aún sin envolver en la propia vista de árbol con el fin de poder ser extendida con más GUIs de este tipo, de forma que cuando se muestre en una ventana con botones, la estructura de nodos de todas estas GUIs sea mostrada en un árbol común. En la sección 5.2.4 se muestra un ejemplo práctico de esta situación.

Por lo tanto, al igual que las demás GUIs extensibles de la librería, esta clase extenderá a *ExtensiblePanel*. En este caso el *JPanel* que hereda no necesita ser modificado y será sobre el que se diseñe la GUI correspondiente. Además contendrá un nodo, que a su vez contiene esta GUI, para ser usado como representante de la GUI en el árbol.

Como esta clase está pensada para construir la jerarquía de nodos desde la vista de diseño, debe poder distinguir dentro del mismo método los componentes que se insertan para formar parte de la GUI y los que se insertan como GUIs hijas. Esta distinción se hará mediante esta propia clase: los componentes que sean también de esta clase se insertarán como hijos y todos los demás de forma convencional en la GUI. De esta forma el diseñador puede crear la GUI normalmente y establecer GUIs hijas tan solo arrastrándolas dentro de ésta.

Una vez que se desee envolver esta GUI en vista de árbol podrá insertarse normalmente sin tratar con el nodo, ya que *TreeViewPanel* ya se encarga de detectar cuando se le añade este tipo de GUI e inserta su nodo automáticamente. Sin embargo no es necesario hacer esto, ya que si esta GUI se envuelve directamente en una ventana con botones, se envolverá también en vista de árbol automáticamente. De esta forma se puede mantener el árbol de nodos expansible hasta el final.

4.7. Frame

Esta clase extenderá a *JFrame* de Swing aportando mayor gestión del tamaño.

Por una parte se encargará de limitar automáticamente el tamaño mínimo y máximo al que el usuario la puede redimensionar. Estos tamaños se pueden obtener directamente del panel contenedor de la ventana, ya que el layout calcula el tamaño mínimo y máximo que puede tener el contenedor del que se encarga. Por lo tanto tan solo será necesario sumarle el tamaño que ocupa el marco exterior de la ventana a estos tamaños para obtener el tamaño correspondiente a la ventana.

Por otra parte debe detectar cuando la ventana no entra en la pantalla, para lo cual simplemente se comparará el tamaño de la pantalla con el tamaño mínimo de la ventana cada vez que éste cambie. Se permitirá al programador especificar si desea que se añadan barras de scroll, que se muestre un mensaje de error y que se cierre la ventana siempre que se produzca esta situación mediante el método *setOversizedPolicy*. También detectará si se sale de la pantalla al compactarse, en cuyo caso se reposicionará para ser mostrada al completo. La sección 5.2.3 demuestra este funcionamiento.

4.8. ExtensibleFrame

Esta ventana se compondrá de 2 partes: un panel con 2 botones para aceptar y cancelar, y el contenido.

Se usará *BorderLayout* para distribuir estas partes, de forma que el panel con botones quedará en la parte inferior ocupando solo el espacio necesario, y el resto del espacio de la ventana será para el contenido.

Este contenido será una GUI extensible de la librería, que puede ser establecido tanto al crear la ventana como posteriormente. Se permite también establecer como contenido cualquier componente, pero si no se trata de una GUI extensible será insertado envuelto en una GUI extensible en una sola página.

El objetivo de esta restricción es doble: por un lado se tiene acceso a los métodos de estas GUIs que propagan los eventos que desencadenan estos botones y por el otro se puede seguir usando la GUI transparentemente sin necesidad de distinguir si está envuelta en botones o no, ya que la ventana podrá redirigir los fragmentos que se le inserten a la GUI que contiene.

Para desencadenar los eventos de los botones no se seguirá estrictamente el esquema de la figura

3.2. El botón cancelar en su lugar solo cerrará la ventana, y el de aceptar cerrará la ventana sin propagar el evento de limpiar (solo si ha validado con éxito y por lo tanto ha llegado a ese punto). La tarea de limpiar será responsabilidad de un listener que estará pendiente de cuándo se cierra la ventana, por lo que al final siempre se sucederán los eventos de acuerdo al esquema. Sin embargo de esta forma se concentra el código común de ambos botones y sobre todo se realizará la limpieza en las GUIs sin importar que la ventana no se cierre mediante estos botones.

4.9. Mailbox

Esta clase encapsula un *HashMap* en el que delega sus operaciones. En él se podrán guardar y recuperar pares de clave-valor en el que el valor sería el mensaje, con lo que cualquier GUI podrá consultar el mensaje enviado tan solo conociendo su clave. Cualquier objeto podrá también borrar el mensaje conociendo su clave.

Su mayor ventaja respecto a un *HashMap* básico es que es accesible de forma global. Esto se realiza aplicando el patrón Singleton (ver sección 2.2) y de esta forma se puede acceder desde cualquier GUI a la misma instancia sin necesidad de tener que pasarle la referencia a todas ellas.

4.10. Messenger

Esta clase también encapsula un *HashMap* y es accesible globalmente, pero en este caso es usado para relacionar cada clave con la lista de listeneres registrados en ella.

Los listeneres podrán vincularse o desvincularse con cualquier número de claves de forma independiente. Para enviar un mensaje se recorrerá secuencialmente la lista de listeners asociada a su clave, en consecuencia los destinatarios de un mismo mensaje dependen del instante en el que se mande y pueden ser cambiantes en el tiempo; además si en el momento en el que se envía no hay listeners vinculados el mensaje se perderá y no será recibido por listeners registrados posteriormente.

En cuanto a la estructura de datos para lista de listeners, una lista convencional permitiría ser recorrida en un tiempo lineal respecto a su tamaño pero no permitiría borrar elementos en tiempo constante, por otro lado un *HashSet* sí realiza borrados en tiempo constante y además evita duplicados pero requiere más tiempo para recorrer los listeners ya que debe recorrer el array interno en el que se dispersan los elementos en busca de posiciones que los contengan. *LinkedHashSet* utiliza conjuntamente estas estructuras para ofrecer las ventajas de cada una en cuanto a rendimiento a costa de requerir un mayor espacio. Con el fin de permitir operaciones de borrado frecuentes en grandes conjuntos de listeners sin resentir el rendimiento del sistema se ha optado por esta última clase.

Capítulo 5

Pruebas

Con el fin de realizar pruebas y sobre todo de poder mostrar el funcionamiento de algo tan abstracto como una librería de una forma más visual se ha creado un programa de prueba. Esta aplicación usará intensivamente la librería en todos sus ámbitos para demostrar que se ajusta a todos los requisitos y se comporta de forma robusta y fiable.

En este capítulo se describirá primero este programa para luego identificar posibles experimentos que se pueden realizar en él.

5.1. El programa de prueba

No se creará un típico programa que tan solo se dedique a comprobar mecánicamente que se cumplen ciertas condiciones para un conjunto de escenarios sino que se creará una aplicación real con sentido, similar a las aplicaciones que demandarían el uso de esta librería, con el objetivo de mostrar que se comporta adecuadamente en un contexto real y de presentar su funcionamiento de una forma más atractiva y amigable.

5.1.1. Descripción

El programa creado se trata de un juego basado en el popular juego de la serpiente (ver La serpiente (videojuego)). Este juego consiste en una serpiente orientada por el jugador que alarga su cola cada vez que come y que debe evitar chocar con obstáculos mientras avanza, incluida su propia cola. Estos obstáculos, a diferencia de en el juego original, estarán vivos e intentarán morder a la serpiente por el lugar más cercano. Todos estos elementos serán figuras geométricas: la comida serán círculos y los elementos vivos (serpiente y enemigos) serán polígonos.

El juego comenzará cuando se pulse una tecla de dirección, momento desde el que la serpiente se moverá ininterrumpidamente pudiendo el jugador tan solo modificar su dirección, hasta que choque contra los bordes de la ventana, contra su cola, o con algún enemigo; instante en el que se terminará

la partida mostrando la puntuación y se reiniciará el juego. Entre tanto la serpiente deberá sortear los enemigos para pasar por encima de la comida, lo que le proporcionará un polígono extra de cola.

El objetivo del juego es conseguir que la serpiente sea lo más grande posible mientras se desplaza sin chocarse con nada.

5.1.2. Características

En esta sección se resumen las principales características implementadas en el juego.

1. Detección de colisiones. En el caso de que se produzcan, los cuerpos involucrados rebotarán cambiando su trayectoria y velocidad según la velocidad y el tamaño de cada cuerpo y el ángulo de choque, aplicando las mismas leyes físicas del mundo real.

2. La velocidad del juego será relativa al tiempo real y no a la de la máquina. Es decir, si el sistema se ejecuta a menor velocidad de la normal se mostrarán menos imágenes por segundo pero la velocidad del juego no será alterada.

3. Se podrá pausar y reiniciar el juego mediante un botón en la barra superior.

4. Algunos elementos móviles tienen una rotación constante y permanente mientras que otros giran según la dirección que lleven. La serpiente girará por defecto según su dirección pero rotará mientras se mantenga pulsada la tecla espacio. En el caso de choque dejarán de rotar en ambos casos durante el tiempo que estén rebotando.

5. Las propiedades de todos los elementos de los que se compone el juego son editables mediante la interfaz gráfica de usuario.

6. La comida aparecerá de una en una a intervalos constantes hasta un máximo de instancias simultáneas determinado, ambos parámetros son configurables también por el usuario.

7. Los enemigos serán generados aleatoriamente cada vez que se ejecute el programa, siguiendo una distribución normal acotada entre valores deseables para cada atributo.

8. Estos parámetros estadísticos para generar los enemigos también son configurables para el usuario desde la interfaz gráfica de usuario.

9. El usuario podrá también añadir y eliminar los enemigos que desee, e incluso generar un número arbitrario de ellos haciendo uso del generador.

10. Los enemigos serán posicionados aleatoriamente en cada nuevo juego sin superponer unos encima de otros y sin salirse de la pantalla, actuando de forma robusta en el caso de que no entren todos en la pantalla.

11. Para identificar a cada enemigo en las GUIs, serán nombrados de acuerdo a su número de lados y color. Como el color puede ser establecido por el usuario entre un numero indefinido de opciones, se le dará el nombre del color estándar perceptualmente más parecido [Sharma et al., 2004].

12. Los enemigos serán lo suficientemente inteligentes como para seguir la trayectoria más corta al fragmento de serpiente más cercano, pero no para evitar chocarse unos con otros.

13. Los enemigos no podrán realizar giros completos instantáneos sino que tienen una velocidad máxima de giro que los obliga a girar paulatinamente en el tiempo hasta el ángulo deseado.

14. Se permite mantener cualquier número de instancias de la misma GUI abiertas simultáneamente, y todas las instancias estarán sincronizadas por lo que un cambio en una se reflejará en todas las demás e incluso en las que se abran posteriormente mientras se mantenga alguna abierta

5.1.3. Arquitectura

El motor del juego se trata de un hilo que periódicamente actualiza y dibuja cada elemento. Estos elementos se definen genéricamente mediante una interfaz para añadir un nivel de indirección entre el motor del juego y los elementos que lo componen.

Aunque lo ideal para un juego sería utilizar una librería gráfica especializada como OpenGL, ya que se trata de un programa de prueba para una librería basada en Swing, se usará también Swing. El juego por lo tanto usará como ventana la clase *Frame* (ver sección 4.7) que proporciona la librería y dibujará cada elemento en su panel interno.

La estructura podría considerarse también similar a la de Swing: cada elemento puede estar compuesto de otros elementos formando una jerarquía en la que actualizar o dibujar un elemento superior propaga recursivamente estas acciones hasta los elementos base.

Los distintos tipos de figuras geométricas se definen en distintos niveles de abstracción mediante relaciones de herencia que permiten mantener los atributos y funcionalidades comunes agrupados en superclases.

Además cada tipo de figura puede contener un número indefinido de figuras de ese tipo. De esta forma no se duplican los atributos en cada figura sino que todas las figuras que comparten estos atributos acceden al único lugar en el que se definen. Esto es especialmente útil en el caso de la comida, ya que todos los trozos de comida son iguales.

La arquitectura del sistema resultante se muestra en la figura 5.1, en la que solo se muestran las clases fundamentales y se omiten sus detalles.

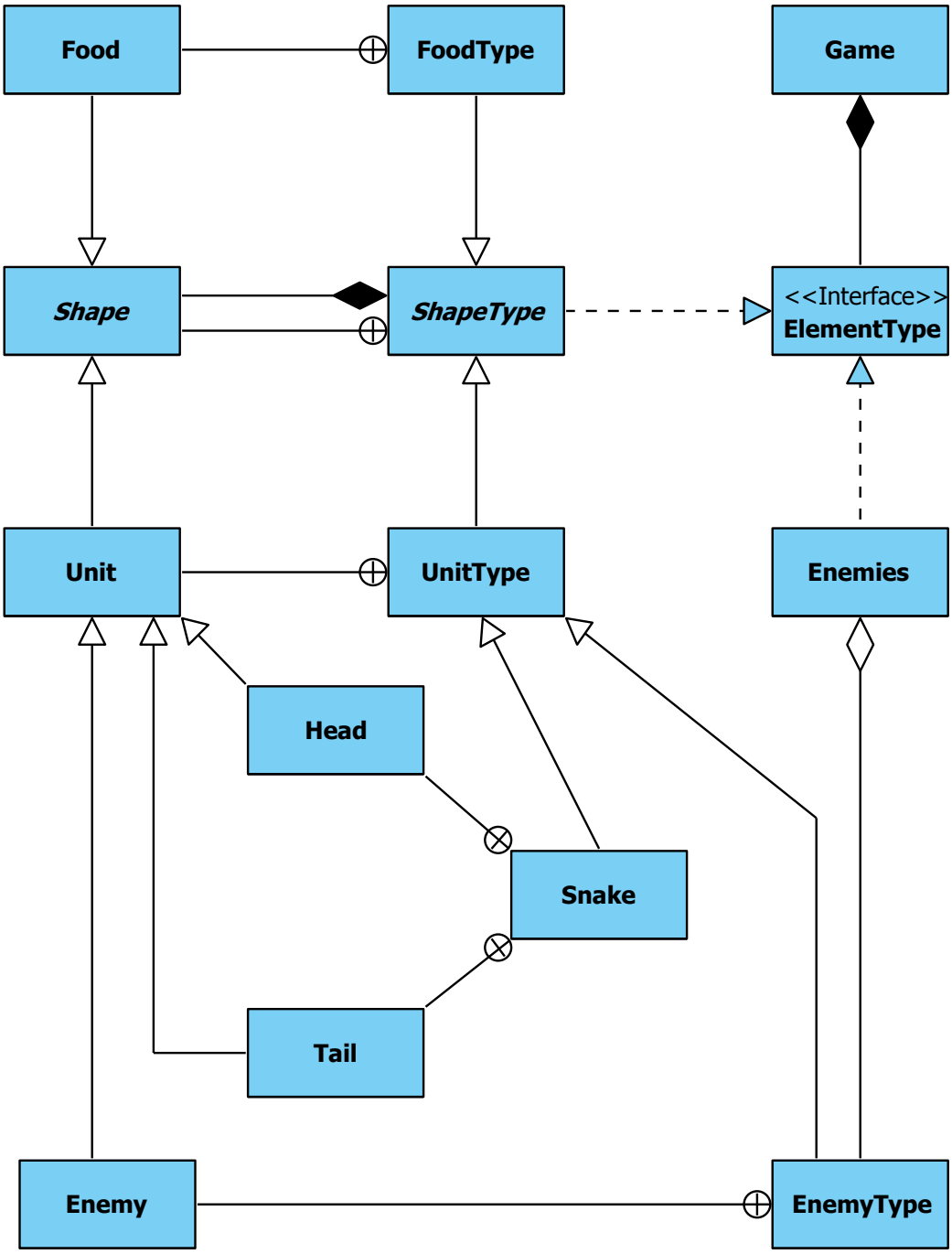


Figura 5.1: Arquitectura del programa de prueba.

5.1.4. Propiedades configurables por el usuario

Por último en esta sección se enumeran los atributos editables de cada clase mediante la interfaz gráfica de usuario.

5.1.4.1. ShapeType

Fill enabled: si la figura tiene relleno.

Fill color: color del relleno.

Border enabled: si la figura tiene borde.

Border size: anchura del borde, en píxeles.

Border color: color del borde.

Radius: radio de la figura, en píxeles.

5.1.4.2. FoodType

Maximum: número máximo de trozos de comida que pueden existir simultáneamente.

Spawn time: tiempo que tarda en aparecer el siguiente trozo de comida (siempre que no se supere el máximo), en segundos.

5.1.4.3. UnitType

Speed: velocidad en píxeles por segundo a la que se mueve.

Sides: número de lados del polígono.

Rotate enabled: si rota continuamente o si por el contrario gira según su dirección (ver característica 4).

Angular speed: velocidad de rotación, en revoluciones por segundo.

Clockwise: si gira en sentido horario.

Counterclockwise: si gira en sentido antihorario.

Initial angle: rotación inicial que tiene el polígono, en grados.

5.1.4.4. Snake

Tail sides: número de lados de los polígonos que forman la cola de la serpiente.

Tail initial angle: rotación inicial de los polígonos que forman la cola, en grados.

5.1.4.5. EnemyType

Turning speed: velocidad máxima a la que puede girar para cambiar de dirección, en revoluciones por segundo.

Copies: número de instancias de este tipo a crear.

5.1.4.6. Game

Background: color de fondo.

FPS: número máximo de imágenes por segundo al que se ejecutará el juego.

FPS show: si se muestran los FPS en la pantalla.

Resolution: tamaño de la ventana del juego, en píxeles.

5.1.4.7. Enemies

Esta clase permite configurar, además de los atributos de todos los enemigos, los parámetros estadísticos usados para generar valores aleatorios para cada uno de los atributos de los enemigos nuevos que se creen. Dependiendo del tipo de cada atributo se requieren distintos parámetros para generar sus valores:

- Para atributos booleanos se debe generar o bien verdadero o bien falso, y solo será necesaria la probabilidad de ser cierto. Por ejemplo para el sentido de rotación.
- Para atributos que requieran valores con distribución uniforme en un rango tan solo se necesitan los límites del rango. Por ejemplo para el tono del color.
- Para atributos que requieran valores con distribución normal en un rango, además de los límites del mismo, es necesaria la media de la distribución y la desviación típica. Por ejemplo para el tamaño.

Sin embargo, para obtener un resultado más compacto en la GUI, se mostrarán en una tabla común y dependiendo del atributo se usarán unas columnas u otras. A continuación se detalla el significado de cada columna para cada tipo de atributo.

Mean/probability: valor medio en torno al que se generan los números aleatorios para distribuciones normales o probabilidad de ser verdadero para atributos booleanos.

Std dev: desviación típica para valores generados con distribución normal. Define cómo de dispersos estarán los valores respecto al punto medio. En concreto, implica que sobre un 68 % de los números generados tenderá a estar a una distancia de la media menor que la desviación típica, un 95 % a menos de 2 desviaciones típicas, y un 98,7 % a menos de 3 desviaciones típicas.

Listado de código 5.1: Herencia de GUIs

```
1  public class A {
2
3    public ExtensiblePanel getGUI() {
4      return new GUI_A();
5    }
6
7  }
8
9  public class B extends A {
10
11   @Override
12   public ExtensiblePanel getGUI() {
13     ExtensiblePanel guiPadre = super.getGUI();
14     guiPadre.addChild(new GUI_B());
15     return guiPadre;
16   }
17
18 }
```

Min: límite inferior del rango.

Max: límite superior del rango.

5.2. Casos de prueba

5.2.1. Herencia de GUIs en una jerarquía de clases

El acoplamiento de GUIs de la librería permite extender una clase con GUI asociada teniendo tan solo que diseñar la parte con la que se desea ampliarla. De esta forma se puede seguir un proceso paralelo al de la POO en el que cada subclase es responsable solo de los atributos nuevos que define y de la parte de GUI que los edita, evitando así la duplicidad del código y favoreciendo la modularidad y la cohesión de las clases, principios imprescindibles en la programación moderna.

Esto puede conseguirse fácilmente siguiendo el esquema mostrado por el código 5.1, en el que GUI_A y GUI_B son las GUIs que editan los atributos definidos en las clases A y B respectivamente. Nótese que como las GUIs devueltas pueden ser extendidas por varias subclases diferentes en cada llamada se crea una nueva instancia por lo que la GUI final obtenida se debería almacenar en una variable en vez de utilizar este método cada vez que deba mostrarse.

En el programa de prueba, esta situación se produce en las GUIs que editan los atributos de la serpiente, la comida y los enemigos, cuyas clases forman parte de una jerarquía común en la que se encuentran dispersos sus atributos, tal y como se muestra en la figura 5.2.

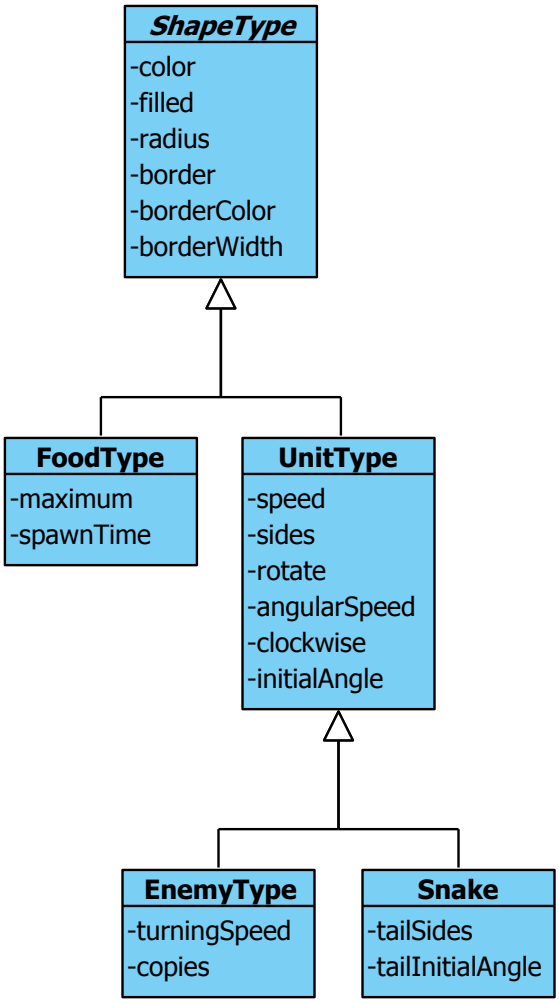


Figura 5.2: Jerarquía de clases editables en el programa de prueba.

Para cada una de estas clases se diseña una GUI que se encarga de editar solo los atributos definidos en dicha clase (no los heredados) que se hereda en las subclases siguiendo el procedimiento expuesto anteriormente, con lo que se consiguen reutilizar los fragmentos de GUI comunes.

En la figura 5.3 se muestran los resultados para el caso de la GUI de la comida, en el que además se puede apreciar cómo el fragmento más estrecho se expande hasta tener la misma anchura que el más ancho y si se redimensiona esta GUI se puede observar cómo escala en mayor proporción el fragmento que originalmente era más alto (se cumple el requisito $2a$). El resto de casos se pueden inferir del programa de prueba examinando las partes comunes de cada GUI.

5.2.2. Redimensionamiento

Puede observarse en todas las GUIs del programa de prueba cómo al redimensionar la ventana la GUI se adapta a su contenido y cómo se impide hacerla más pequeña que su tamaño mínimo. Esto se aprecia especialmente en el caso de GUIs con pestañas, en el que dependiendo de la pestaña mostrada se permite hacer la ventana más pequeña si el contenido de ésta es menor; en la figura 5.4 se muestra un ejemplo.

5.2.3. Detección y tratamiento de GUIs más grandes que la pantalla

Para probar cómo la librería cumple el requisito 8 se pueden seguir los siguientes pasos en el programa de prueba, mostrados en la figura 5.5a:

1. Abrir la ventana Options.
2. Modificar el valor de Resolution de forma que al menos una de las dos dimensiones sea mayor que la de la pantalla.
3. Click en Ok para guardar los cambios.

Como resultado se obtiene una GUI que no entra en la pantalla y la librería al detectarlo muestra un mensaje de error y envuelve el contenido en barras de scroll para permitir visualizarlo completamente (figura 5.5b).

Para conseguirlo, la ventana principal del juego, que extiende a *Frame* de la librería, tan solo ha tenido que especificar la política que desea que se siga cuando esto suceda, en este caso añadir barras de scroll y mostrar advertencia pero no cerrar la ventana, mediante el método siguiente:

```
1 setOversizedPolicy(boolean addScrollBar, boolean showWarning,
    boolean closeWindow)
```

Si posteriormente se vuelve a establecer el tamaño del juego para que entre en la pantalla, se observa que desaparecen las barras de scroll.

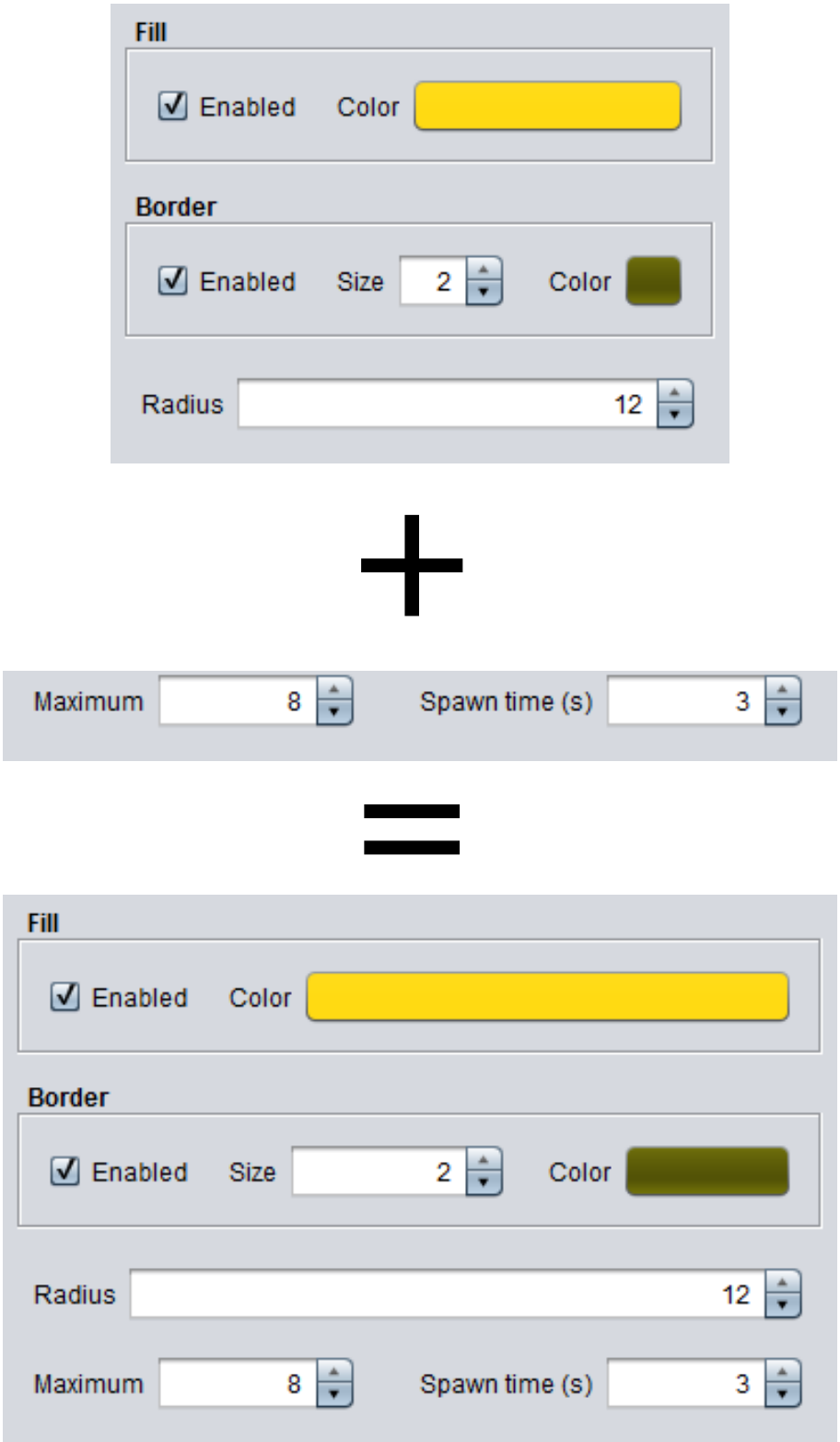


Figura 5.3: Ejemplo de herencia de GUIs.

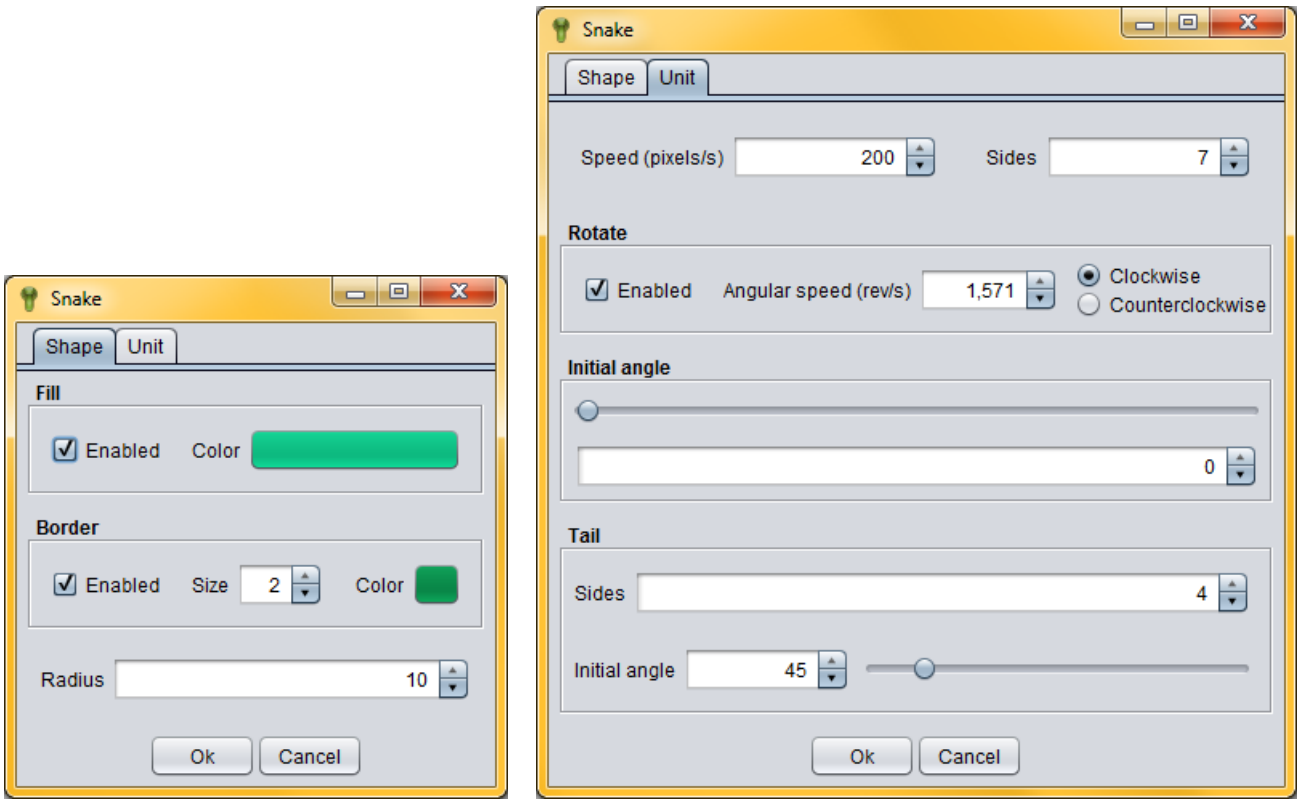
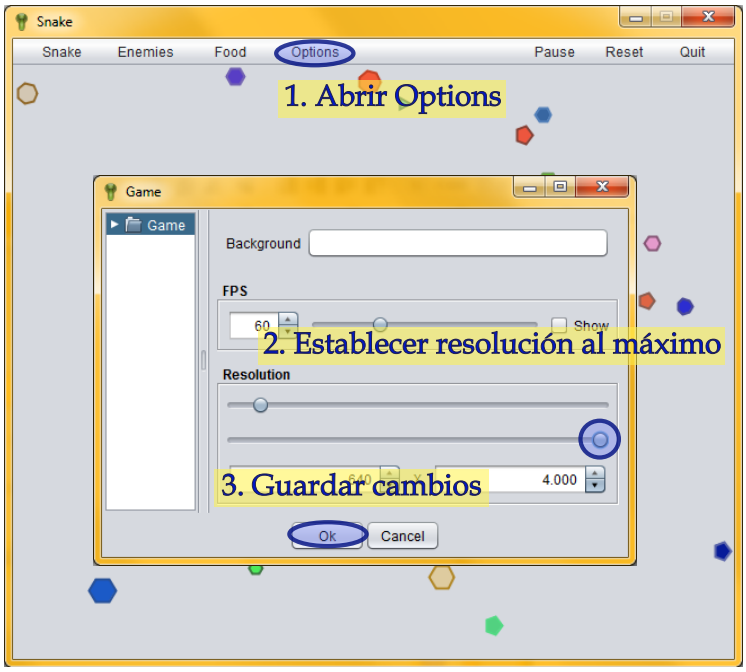
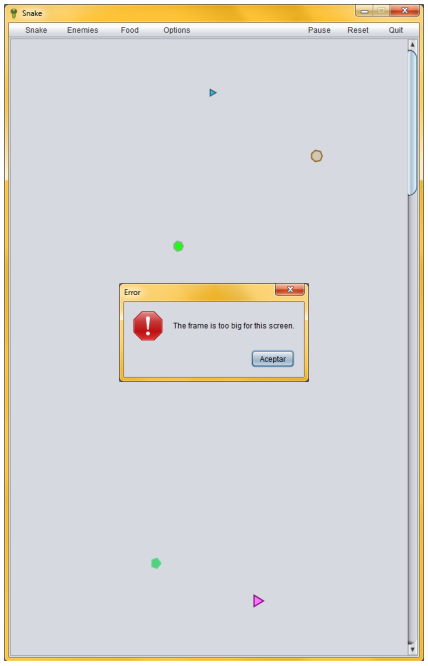


Figura 5.4: Diferentes tamaños mínimos de la ventana según la pestaña.



(a) Pasos.



(b) Resultado.

Figura 5.5: Detección y tratamiento de GUIs más grandes que la pantalla.

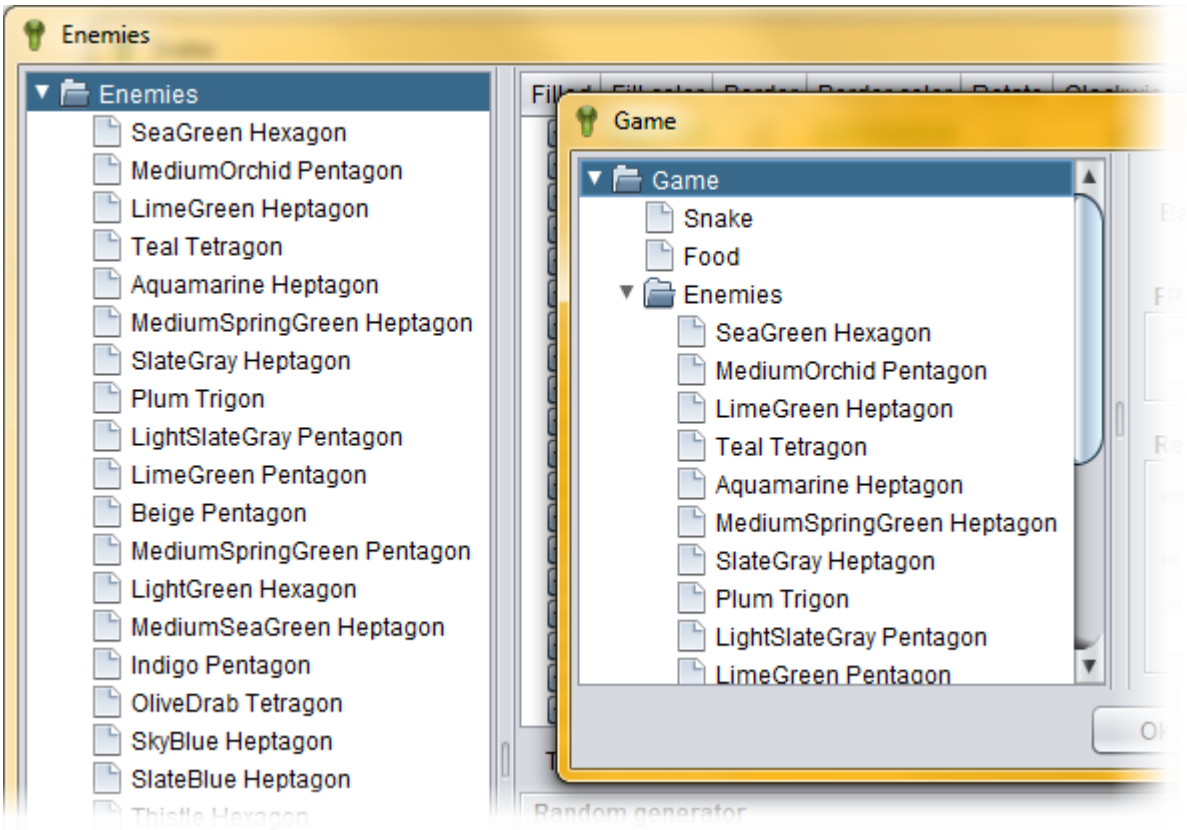


Figura 5.6: Fusión de GUIs con vista de árbol.

También se puede apreciar que solo se añade la barra de scroll necesaria, pero si la otra dimensión ahora se establece cerca del límite, la barra de scroll hará que se sobrepase este límite y la librería detecta correctamente esta situación excepcional añadiendo también la otra barra de scroll.

Además en cualquier caso detecta si la ventana se sale de la pantalla y la reposiciona para que se muestre entera (esto no se aplica cuando es el usuario el que la mueve fuera de la pantalla para otorgarle esa posibilidad).

5.2.4. Fusión de GUIs con vista de árbol

Las GUIs individuales accesibles en el menú (Snake, Enemies y Food) son también mostradas en la vista de árbol de Options, que contiene todas las GUIs del programa. Sin embargo, la GUI Enemies ya era una GUI con vista de árbol por lo que se produciría la situación poco deseable de tener una vista de árbol anidada en otra.

No obstante, tal y como se puede comprobar en la figura 5.6, esto no sucede así sino que se ha fusionado el árbol de Enemies con el de Options, manteniendo la semántica jerárquica de las GUIs pero dentro de la misma vista de árbol; y todo ello de forma transparente sin tener que añadir lógica adicional en el programa de prueba, tal y como se puede apreciar en el listado de código 5.2, en el que se muestra de forma simplificada la creación de estas GUIs en los botones del menú.

Listado de código 5.2: Creación de GUIs en los botones del menú.

```
WindowCloseListener listener = new WindowCloseListener() {

  @Override
  public boolean validateThis() {
    return true;
  }

  @Override
  public void saveThis() {
    game.reset();
  }

  @Override
  public void cleanThis() {}

};

buttonEnemies.addActionListener(new ActionListener() {

  @Override
  public void actionPerformed(ActionEvent e) {
    ExtensibleFrame.showWindow(enemies.getUI()).
       setWindowCloseListener(listener);
  }

});

buttonOptions.addActionListener(new ActionListener() {

  @Override
  public void actionPerformed(ActionEvent e) {
    TreeNodePanel gameui = new GameUI(game);
    gameui.addChild(snake.getUI());
    gameui.addChild(food.getUI());
    gameui.addChild(enemies.getUI());
    gameui.setName("Game");
    ExtensibleFrame.showWindow(gameui).setWindowCloseListener(
       listener);
  }

});
```

Esto se consigue manteniendo la GUI sin envolver en la vista de árbol mediante *TreeNodePanel*, tal y como se especificó en la sección 4.6. También podría modificarse la clase *TreeViewPanel* para que cuando se le añadiera una clase de su mismo tipo insertara los nodos que contiene en vez de la GUI entera anidada, sin embargo se debe descartar esta solución debido a que es posible que en algún caso se desee anidar las GUIs y no fusionarlas.

De esta prueba también se puede inferir la utilidad y simplicidad del mecanismo que permite envolver una GUI en una ventana con botones, que se puede realizar en una sola línea, tal y como se muestra en el listado de código 5.2. En este código también queda de manifiesto lo conveniente que puede resultar disponer del método *setWindowCloseListener* para aplicar el patrón Command y de esta forma añadir acciones personalizadas que se ejecutarán en último lugar para cada evento, en este caso reiniciar el juego después de que se guarden datos nuevos.

5.2.5. Paso de mensajes

Aunque lo óptimo sería tener una instancia de cada GUI disponible en todo momento y hacerla visible e invisible para mostrarla y ocultarla, minimizando de esta forma el tiempo de respuesta, con el fin de mostrar el mecanismo de paso de mensajes, en su lugar se sacrifica este tiempo de respuesta creando una nueva instancia cada vez que se muestre una GUI para permitir mostrar múltiples copias de la misma GUI simultáneamente pulsando el mismo botón del menú reiteradamente. Estas copias son instancias diferentes de la misma clase por lo que cada una tendrá una copia local de sus atributos, sin embargo, mediante un mecanismo que usará el paso de mensajes de la librería se mantendrán sincronizadas de modo que un cambio en una se reflejará en las demás. Esto incluye también a las GUIs que se encuentran en la vista de árbol de Options.

Además en la GUI Enemies se ha creado una tabla que resume los atributos de todos los enemigos; esta tabla también estará sincronizada con la GUI de cada enemigo, además de con otras tablas en otras GUIs paralelas y las GUIs de los enemigos que contiene, por extensión de lo expuesto en el párrafo anterior.

Para comprobarlo se puede seguir un procedimiento similar al descrito a continuación, cuyos pasos se ilustran en la figura 5.5 (se omite parte de la GUI para ahorrar espacio).

1. Click en el botón Enemies de la barra superior del programa de prueba para abrir la GUI.

2. En la vista de nodos de la izquierda de la GUI abierta, click en la flecha anterior al nodo Enemies o doble click en el mismo para expandirlo. Si hay espacio suficiente en pantalla la GUI se reajusta para mostrar el nombre completo de los nuevos nodos.

3. Modificar algún atributo en la tabla de enemigos, por ejemplo el color. Si se cambia el color o el número de lados además cambiará el nombre del nodo correspondiente y la GUI se reajustará si es necesario para mostrarlo completo.

4. Click en el nodo correspondiente al enemigo cambiado (están en el mismo orden que en la tabla). Se mostrará su GUI y se puede comprobar que el atributo ha sido cambiado también aquí.

5. Nuevamente click en el botón Enemies para abrir una segunda GUI y expandir sus nodos. Se observa que en esta nueva GUI ya aparece el atributo modificado.

6. Seleccionar en la tabla de la nueva GUI la fila modificada anteriormente.

7. Click en el botón Delete. Se elimina tanto la fila de la tabla como el nodo asociado, en ambas GUIs. Además en la primera GUI, que se tenía seleccionado ese nodo, al ser borrado se seleccionará el anterior automáticamente.

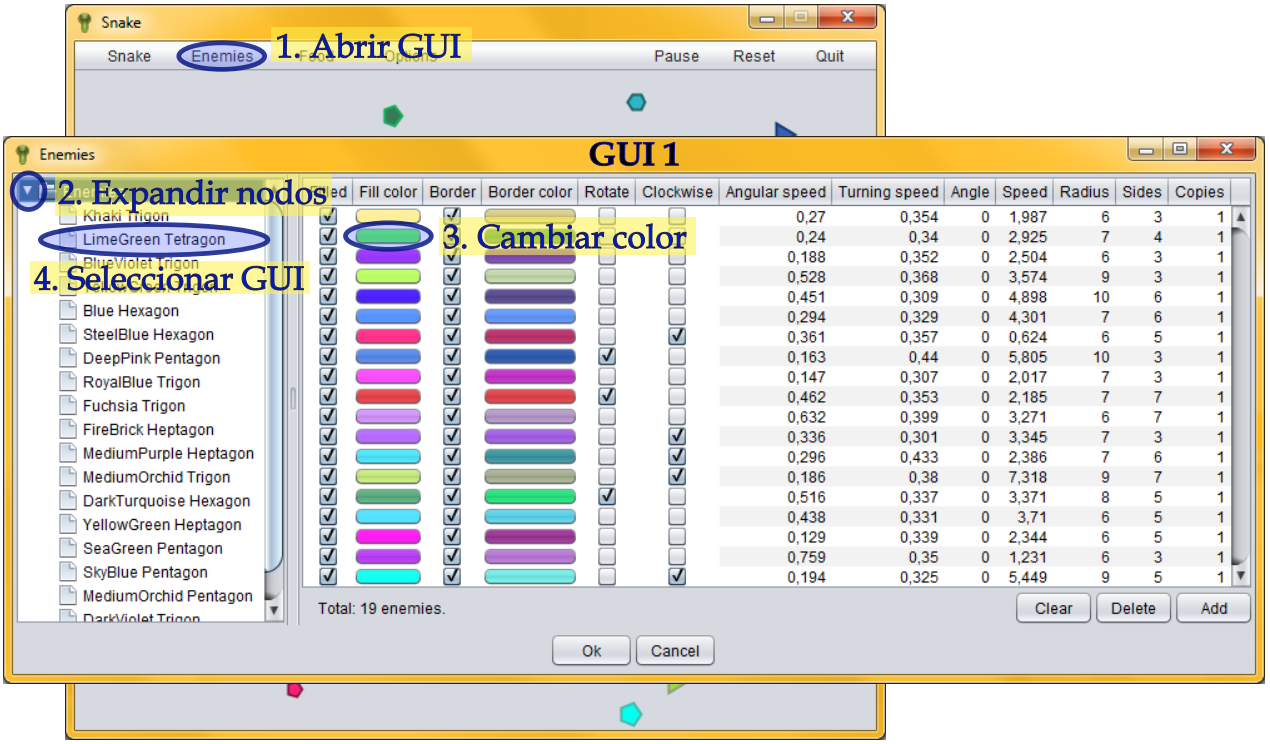
8. Utilizar los botones Clear, Delete, Add o Generate para eliminar todos los enemigos, borrar los seleccionados, añadir uno nuevo o generar un conjunto de ellos, respectivamente, para comprobar que la GUI permite modificar su árbol de nodos dinámicamente y que estás acciones están también sincronizadas entre las GUIs. También se pueden modificar los parámetros de la tabla inferior (se puede observar que estos parámetros en cambio no están sincronizados) para personalizar los valores de los atributos de los enemigos nuevos creados (ver sección 5.1.4.7).

9. Cerrar una de las GUIs mediante el botón Cancel para descartar los cambios realizados.

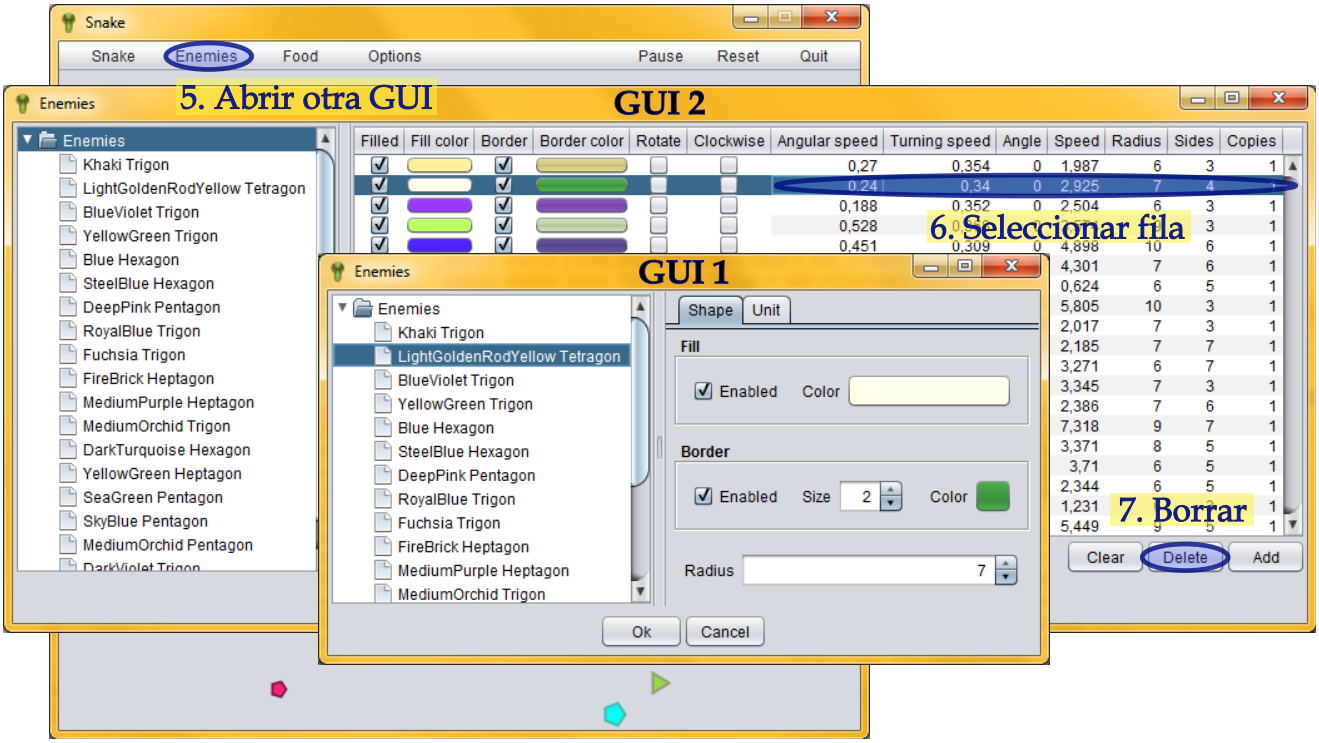
10. Abrir de nuevo otra GUI Enemies para comprobar que descartar los cambios en una no rompe la sincronización mientras siga habiendo otras instancias abiertas.

11. Cerrar las dos instancias mediante el botón Cancel.

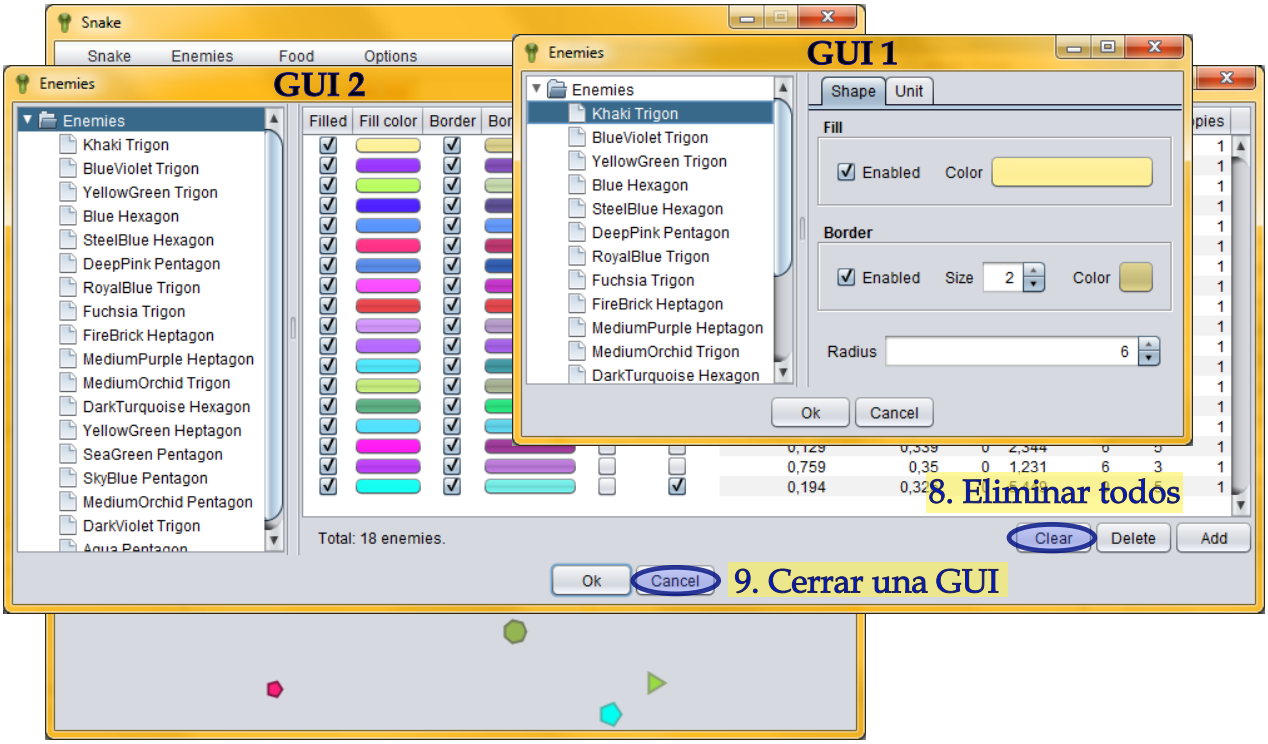
12. Abrir una nueva GUI pulsando el botón Enemies del menú. A diferencia de lo ocurrido en el paso anterior, esta vez no se perciben los cambios realizados anteriormente debido a que se descartaron previamente en todas las instancias sincronizadas.



(a) Pasos 1-4.

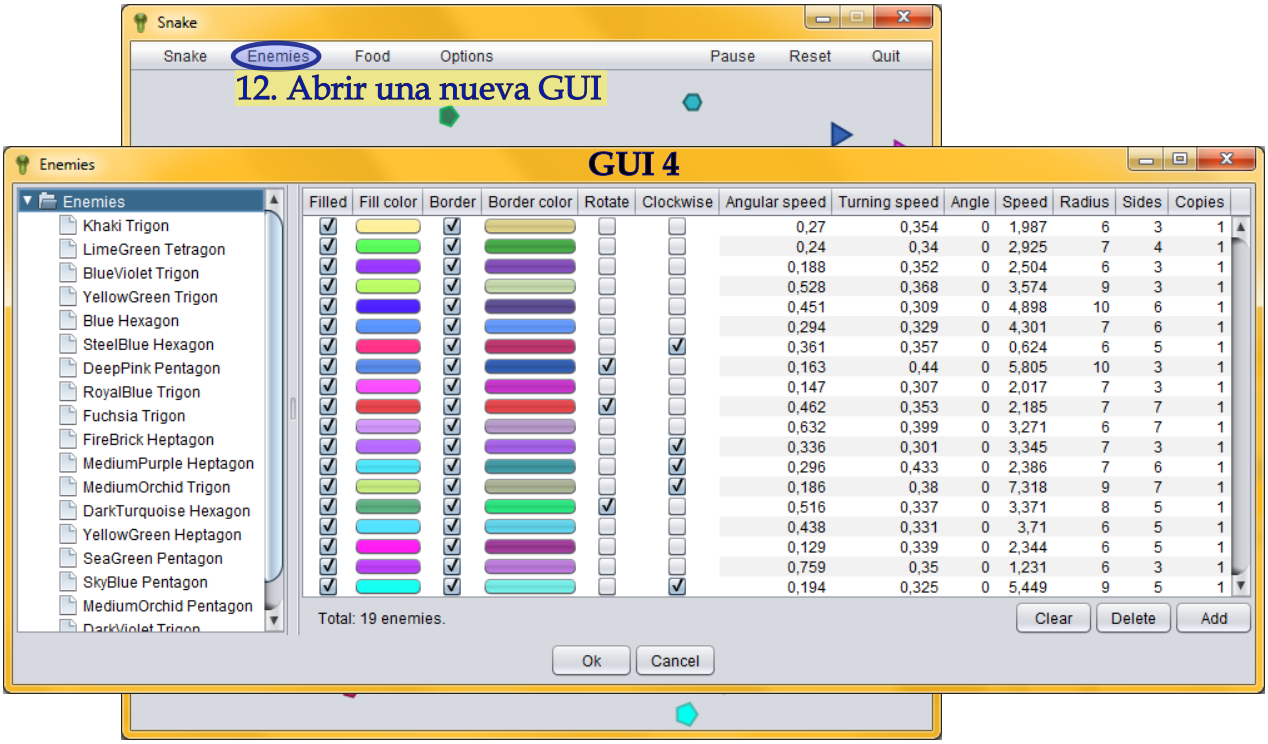


(b) Pasos 5-7.



(c) Pasos 8-9.

(d) Pasos 10-11.



(e) Paso 12.

Figura 5.5: Sincronización entre GUIs.

Capítulo 6

Conclusiones y trabajos futuros

6.1. Conclusiones

Tal y como se desprende de los capítulos anteriores, se han cumplido todos los objetivos:

- Objetivo 1: se pueden diseñar GUIs para una jerarquía de clases siguiendo un patrón similar al de la sección 5.2.1.
- Objetivo 2: todas las GUIs de la librería permiten ser extendidas, además de mediante un método creado específicamente para tal propósito: *addChild*, simplemente añadiéndole las extensiones de forma convencional mediante su método *add*, e incluso desde la vista de diseño arrastrándole componentes o GUIs enteras. También es posible extender GUIs que no pertenecen a la librería tan solo envolviéndolas en uno de los 3 tipos que ofrece la librería según el modo de extensión que se desee.
- Objetivo 3: en la sección 5.2.4 se demuestra mediante un ejemplo cómo aprovechar las GUIs de objetos contenidos en una clase para crear la GUI de la clase que lo contiene. Para ello tan solo es necesario extender la GUI de la clase contenedora con la GUI del objeto que contiene.
- Objetivo 4: es posible diseñar una GUI en módulos independientes y ensamblarlos en una única GUI insertando cada modulo en una GUI extensible de la librería.
- Objetivo 5: se proporciona un sistema de paso de mensajes tanto síncrono como asíncrono mediante las clases *Messenger* (sección 4.10) y *Mailbox* (sección 4.9) respectivamente.
- Objetivo 6: la clase *ExtensibleFrame* (sección 4.8) permite envolver una GUI en una ventana con botones para aceptar o cancelar la edición realizada y cuando se pulse uno de estos botones se usarán los métodos de *ExtensiblePanel* (sección 4.1) para buscar y ejecutar las acciones correspondientes en todos los componentes que contiene la ventana. El programador puede especificar estas acciones en cualquier componente tan solo implementando la interfaz

WindowCloseListener en él. En el listado de código 5.2 de la sección 5.2.4 se puede apreciar un ejemplo de uso de este mecanismo.

En algunos casos incluso se han sobrepasado los objetivos iniciales. Concretamente, se han modificado los requisitos originales de la siguiente forma:

- Requisito 5: aunque no forma parte de los requisitos originales, se han diseñado las GUIs en pestañas y en vista de árbol de manera que se autoajusten al fragmento mostrado en cada momento, con lo que se respetan los tamaños de cada fragmento.
- Requisito 6: en principio solo se pedía una factoría que envolviera GUIs en ventanas con botones. Sin embargo, siguiendo la filosofía orientada a objetos, se ha considerado más apropiado crear también una clase (*ExtensibleFrame*) que representa a esta ventana y puede ofrecer mayores funcionalidades que una ventana genérica. Además ofrece los métodos necesarios para ser usada como la factoría original.
- Requisito 8: originalmente se pedía limitar el número de GUIs a integrar en una sola página a dos para asegurarse de que la GUI resultante entra en la pantalla. En su lugar se ha considerado mejor solución detectar directamente si la GUI resultante entra en la pantalla, para lo que se ha creado una clase (*Frame*) que puede ser usada como ventana para cualquier GUI y de esta forma otorgar dicha funcionalidad a todas ellas.

Además se han implementado nuevas funcionalidades no especificadas en los requisitos:

- Se ha diseñado la GUI en vista de árbol de forma que permita modificar su estructura de nodos dinámicamente en tiempo de ejecución.
- Se han creado clases de soporte (*TreeNodePanel* y *Node*) para facilitar el proceso de creación de la estructura de nodos y se han implementado múltiples métodos extra en las clases de la librería que mejoran su usabilidad.
- Se ha creado un plugin para NetBeans que automatiza la instalación de la librería en dicha plataforma realizando las siguientes funciones:
  - Instalación de la librería en NetBeans.
  - Creación de plantillas para los componentes de la librería.
  - Instalación de las GUIs de la librería en la paleta de la vista de diseño.

6.2. Trabajos futuros

Si bien, como se indicó en la sección anterior, este proyecto sobrepasa los requisitos de calidad a un nivel académico, si se deseara llevarlo a un nivel profesional sería necesario mejorarlo en ciertas áreas para cumplir con unos estándares de calidad mínimos, como por ejemplo:

- Ofrecer también en las GUIs con vista de árbol la posibilidad de elegir, que ya proporcionan las GUIs en pestañas, si se desea no autoajustar la GUI al tamaño de cada fragmento sino ampliar su tamaño al del fragmento más grande y no cambiar el tamaño de la ventana cada vez que se cambie el fragmento a mostrar, lo que puede resultar molesto para el usuario.
- Si no se establece el modo anterior al menos tener en cuenta los redimensionamientos que haya hecho el usuario a la hora de cambiar de fragmento a mostrar, ya que puede ser frustrante para él que la GUI se compacte al tamaño que considera óptimo sin tener en cuenta sus preferencias.
- Crear más tipos de componentes.
- Optimizar el rendimiento y reducir el uso de memoria, ya que estos aspectos no han sido prioritarios al tratarse de un proyecto académico.
- Ampliar el plugin con más funcionalidades como por ejemplo una vista de diseño dedicada a construir la estructura de nodos, y exportarlo a las demás plataformas.

Bibliografía

A Visual Guide to Layout Managers. URL https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html.

D. J. Barnes and M. Kölling. *Programación orientada a objetos con Java: una introducción práctica usando BlueJ*. Pearson Educación, 2007.

B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2005.

How to Set the Look and Feel. URL https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html.

How to Use BorderLayout. URL https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html.

How to Use Scroll Panes. URL https://docs.oracle.com/javase/tutorial/uiswing/components/scrollpane.html.

How to Use Split Panes. URL https://docs.oracle.com/javase/tutorial/uiswing/components/splitpane.html.

How to Use Tabbed Panes. URL https://docs.oracle.com/javase/tutorial/uiswing/components/tabbedpane.html.

How to Use Trees. URL https://docs.oracle.com/javase/tutorial/uiswing/components/tree.html.

How to Write a Tree Expansion Listener. URL https://docs.oracle.com/javase/tutorial/uiswing/events/treeexpansionlistener.html.

How to Write a Tree Model Listener. URL https://docs.oracle.com/javase/tutorial/uiswing/events/treemodellistener.html.

How to Write a Tree Selection Listener. URL https://docs.oracle.com/javase/tutorial/uiswing/events/treeselectionlistener.html.

La serpiente (videojuego). URL https://es.wikipedia.org/wiki/La_serpiente_(videojuego).

C. Larman. *UML y patrones: una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Pearson Educación, 2003.

R. S. Pressman. *Ingeniería del Software. Un Enfoque Práctico*. McGraw-Hill, 7th edition, 2010.

G. Sharma, W. Wu, and E. N. Dalal. *The CIEDE2000 Color-Difference Formula: Implementation Notes, Supplementary Test Data, and Mathematical Observations*. 2004. URL http://www.ece.rochester.edu/~gsharma/ciede2000/ciede2000noteCRNA.pdf.

SLOCCount. URL https://www.dwheeler.com/sloccount.

K. Walrath, M. Campione, Huml A., and Zakhour S. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley, second edition, 2004.

M. A. Weiss. *Estructuras de datos en Java*. Pearson Educación, 7th edition, 2013.

Anexo A

Instalación en NetBeans

Para facilitar este proceso se ha creado un plugin que realiza automáticamente esta tarea. Por lo tanto el procedimiento se reduce a instalar el plugin en NetBeans, para lo cual se pueden seguir los siguientes pasos:

1. En el menú Herramientas seleccionar Plugins.
2. En la ventana que se abre seleccionar la pestaña Descargados.
3. Click en el botón Agregar Plugins...
4. En la ventana emergente abrir el plugin de la librería.
5. Click en el botón instalar.
6. Aceptar todos los pasos del asistente.

Una vez instalado satisfactoriamente:

- La librería aparecerá entre las librerías globales debidamente enlazada con su documentación y código fuente, y estará lista para ser usada en cualquier proyecto que la añada como dependencia mediante Agregar biblioteca...
- Las GUIs extensibles de la librería estarán disponibles en la paleta de la vista de diseño para todos los proyectos (figura A.1). Si se arrastran estos componentes en un proyecto que no ha declarado dependencias con la librería, se importará automáticamente no siendo necesario realizar el procedimiento del punto anterior.
- Al crear un nuevo archivo estarán disponibles como plantillas los componentes de la librería (figura A.2). En este caso sí es necesario que el proyecto tenga declarada la dependencia con la librería y de lo contrario se mostrará un error al crear el archivo.

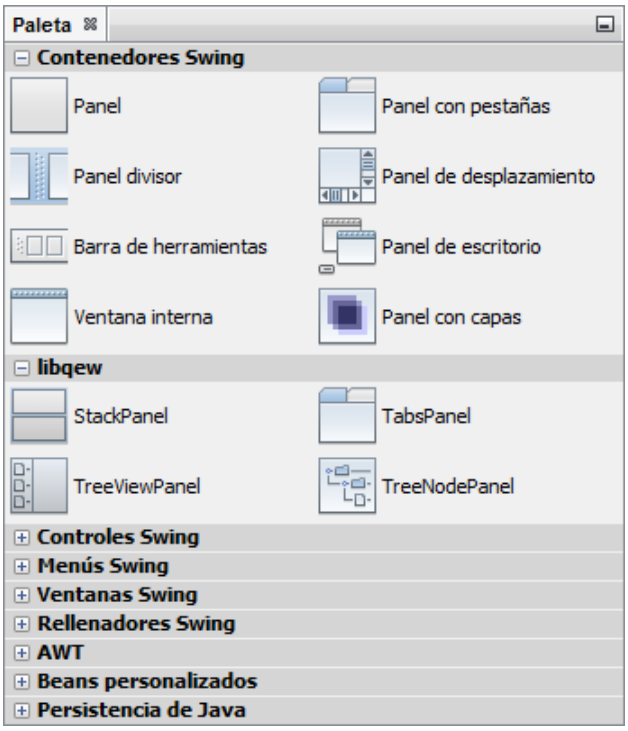


Figura A.1: Paleta con los componentes de la librería.

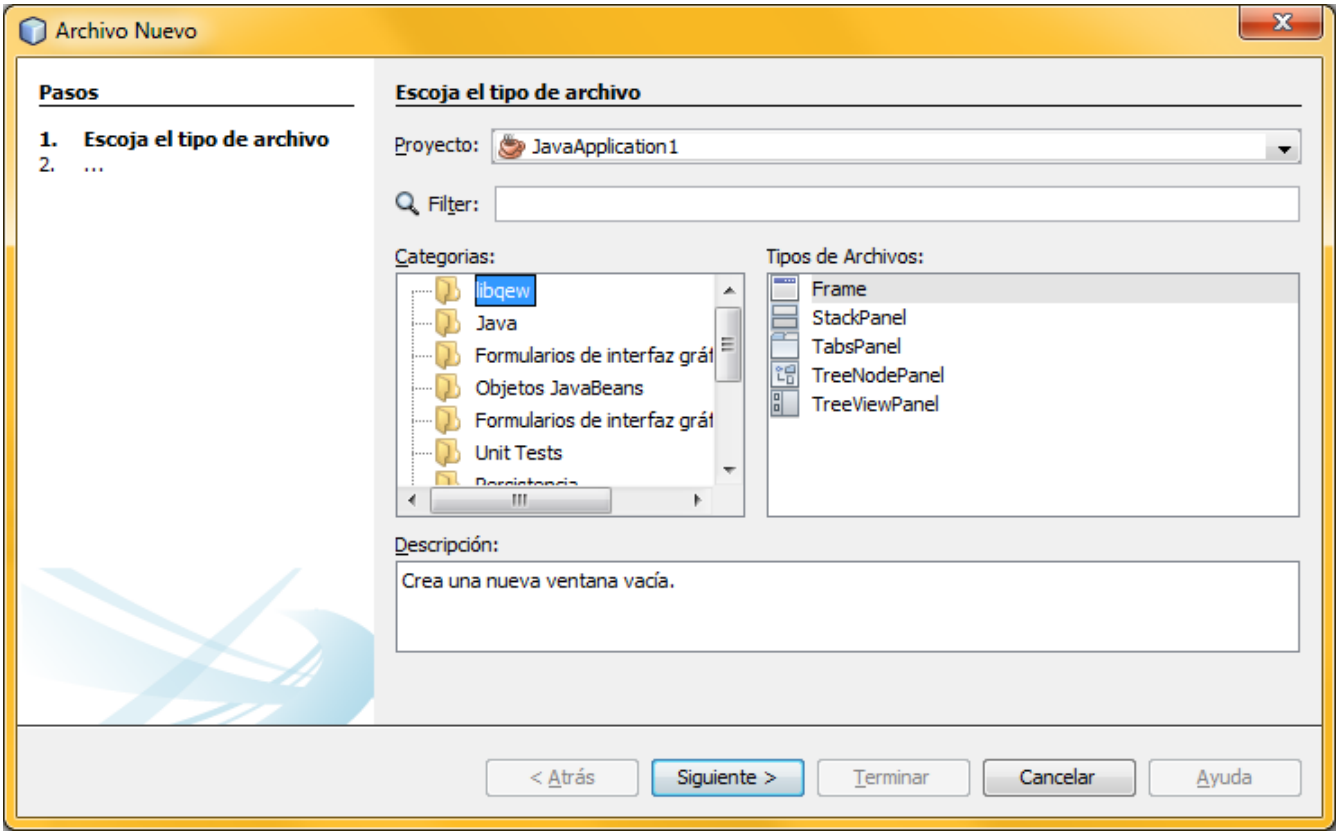


Figura A.2: Plantillas de la librería.

Anexo B

Un ejemplo de uso

Finalmente, en este anexo se mostrará paso por paso cómo usar la librería para crear interfaces gráficas de usuario. Aunque en el capítulo 5 ya se creó un programa de prueba que usa la librería, dicho programa se diseñó para que tuviera sentido y fuera similar a una aplicación real, lo que añade una gran cantidad de complejidad extra, y por lo tanto en este caso, con el objetivo de mostrar únicamente el uso de la librería, se creará un programa lo más básico posible que necesite sus principales funcionalidades aunque carezca de sentido real.

B.1. Descripción del programa

El programa constará de una ventana con botones y vista de árbol en la que se podrá replicar y anidar un panel base desde un panel de control localizado en otra ventana, mediante un sistema de paso de mensajes. Este panel base estará compuesto únicamente por una clave y un valor y podrá ser anidado de cualquiera de las 3 formas que ofrece la librería. Si se pulsa el botón aceptar se comprobará que no haya ningún valor sin rellenar y si no es así, se pondrá en rojo el panel correspondiente y no se cerrará la ventana.

B.2. Creación del proyecto y el panel base

Se utilizará el IDE NetBeans y se supondrá que la librería ya se encuentra instalada en él como se indica en el anexo A. En primer lugar es necesario crear un nuevo proyecto y la GUI que servirá de panel base. Esto se realiza de forma convencional ya que no es necesaria la librería para ello, tal y como se muestra en la figura B.1.

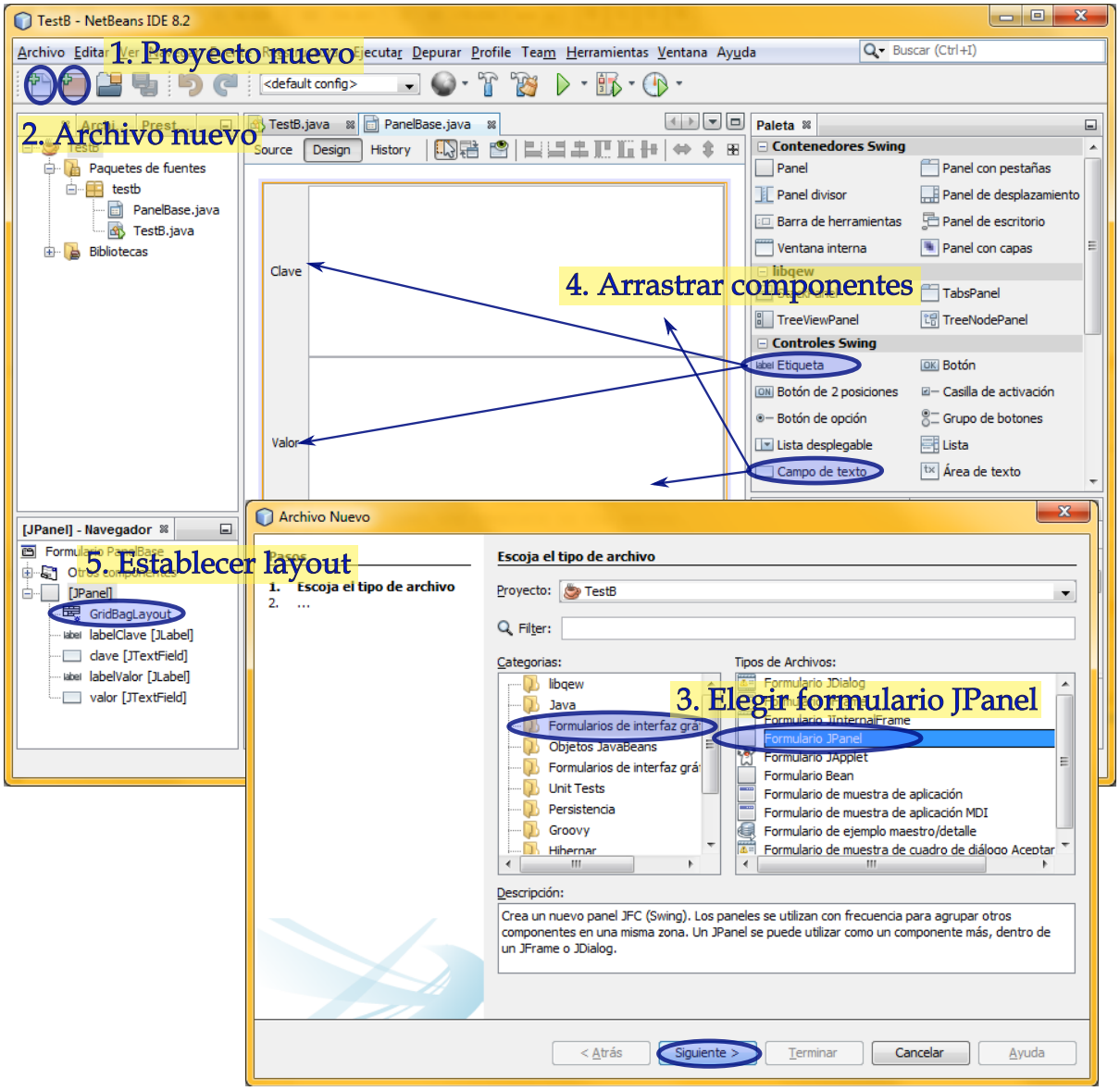


Figura B.1: Diseño del panel base.

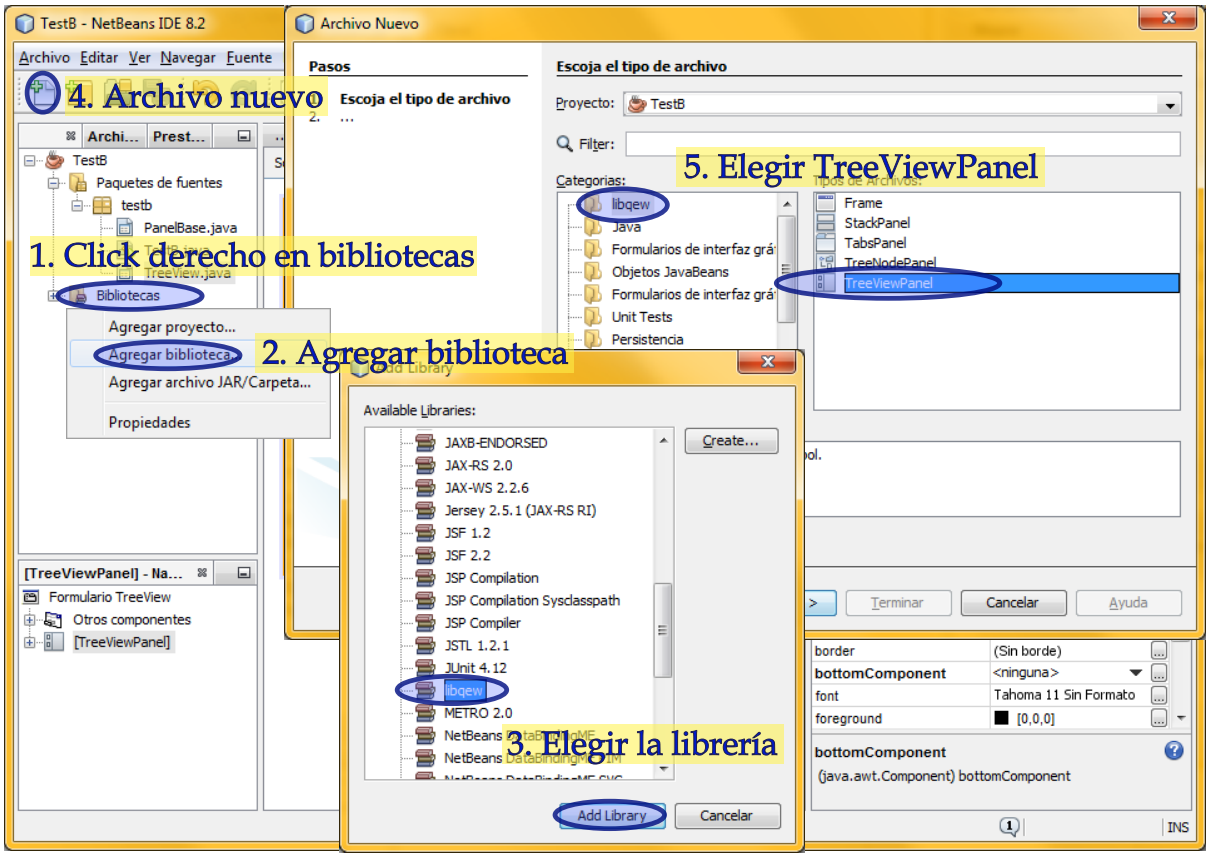


Figura B.2: Creación de una GUI con vista de árbol.

B.3. Creación de la GUI con vista de árbol y el panel de control

A continuación se crea la GUI con vista de árbol en la que se anidarán los paneles base. Como ya se va a usar una clase de la librería, es necesario añadirla al proyecto antes de crear la GUI, como se muestra en la figura B.2.

Una vez seguidos estos pasos, ya se tiene la GUI con vista de árbol vacía en el lienzo, por lo que ya se podrían arrastrarle componentes o GUIs enteras de forma convencional para diseñarla. Sin embargo en este ejemplo, para mostrar el potencial que ofrece la librería, en su lugar se diseñará un panel de control que permitirá insertarle las GUIs dinámicamente en tiempo de ejecución. Este panel de control constará de un menú que permita elegir el modo del que se anidará la nueva GUI, un campo de texto para escribir su nombre, y botones para elegir si insertar una nueva GUI o borrar el nodo seleccionado. Para ello se crea un nuevo archivo de tipo *Frame* de la librería y dentro del lienzo de esta nueva clase se arrastra un *StackPanel* desde la paleta, para posteriormente arrastrar los componentes necesarios dentro de él, que se encargará de apilarlos. La plantilla que crea archivos de la clase *Frame* ya crea un main que se encarga de establecer el *Look and Feel* y mostrar la ventana, por lo que si se generó una clase main al crear el proyecto debería borrarse y establecer ésta como

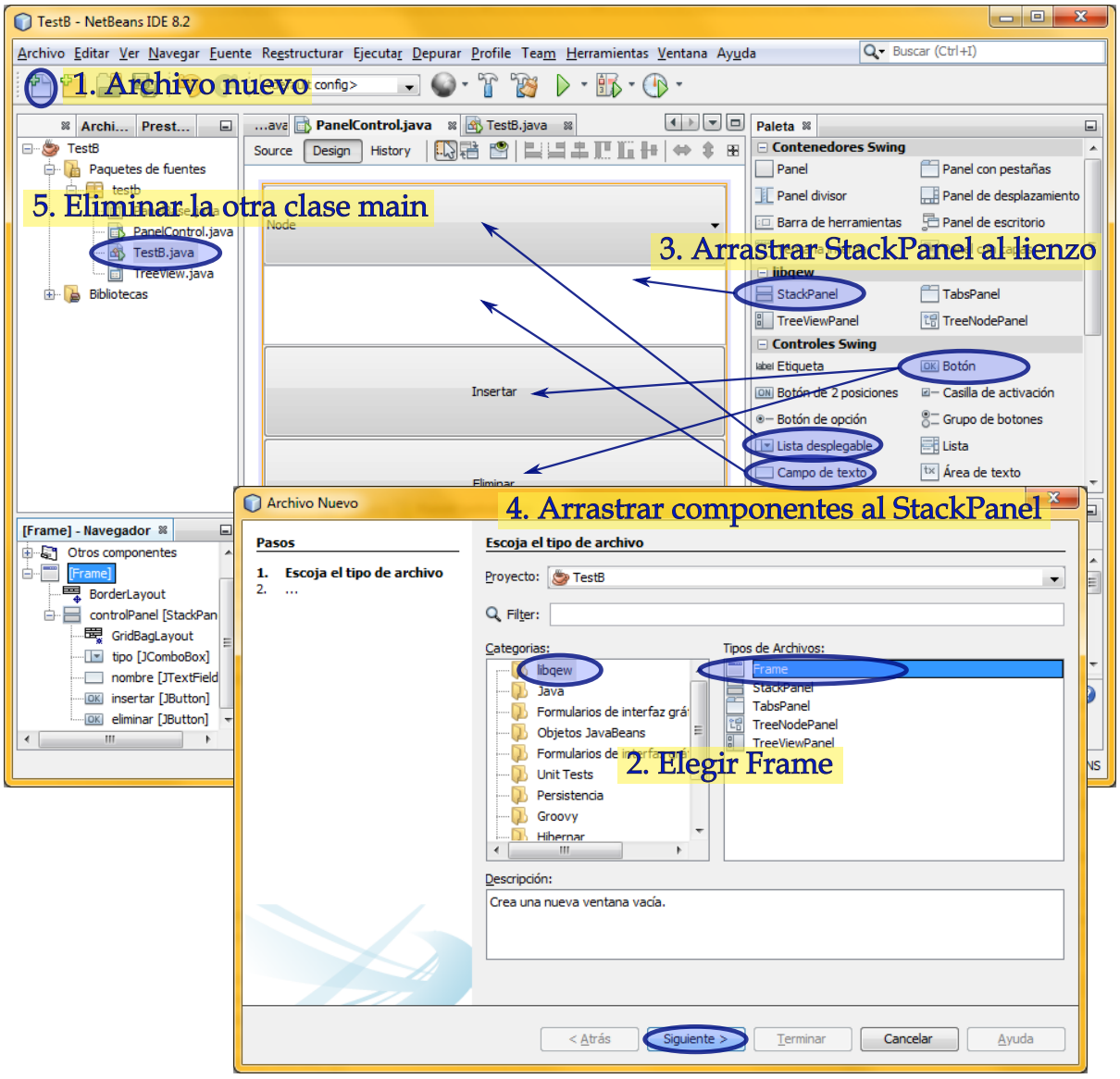


Figura B.3: Creación del panel de control.

la clase main del proyecto. En la figura B.3 se ilustra el procedimiento anterior.

B.4. Envoltura en una ventana con botones

Es necesario envolver la GUI con vista de árbol en una ventana para que pueda ser mostrada. Esta ventana tendrá además botones para aceptar o cancelar sus datos, por lo que en este caso se usará la clase *ExtensibleFrame* de la librería. Esta ventana será mostrada, junto con el panel de control, cuando se ejecute el programa, y cuando una de las 2 se cierre finalizará el programa. Por lo tanto, hay que añadir el código B.1 en el main creado en el panel de control, justo donde se crea y muestra la ventana del panel de control.

Listado de código B.1: Envoltura en una ventana con botones.

```
1 ExtensibleFrame.showWindow(new TreeView()).
     setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Listado de código B.2: Validación del contenido del panel base.

```
1  @Override
2  public boolean validateThis() {
3    if (valor.getText().isEmpty()) {
4      setBackground(Color.RED);
5      return false;
6    } else {
7      setBackground(Color.GREEN);
8      return true;
9    }
10 }
```

B.5. Validación del contenido

Cuando se pulse el botón de aceptar en la ventana de la GUI con vista de árbol se comprobará que en cada panel base ha sido rellenado el campo "Valor" y solo en ese caso se cerrará. Para conseguirlo tan solo es necesario implementar la interfaz *WindowCloseListener* de la librería en la clase del panel base, y comprobar en su método *validateThis* que ese campo no está vacío. Además se cambiará el color del panel para indicar el resultado. El método resultante se muestra en el listado de código B.2.

B.6. Enlace del campo clave del panel base con el nombre a mostrar

Se mostrará como nombre del nodo en la vista de árbol o como título de la pestaña, según corresponda, el texto del campo "Clave" de la GUI que contiene, incluso si se modifica este campo en tiempo de ejecución. Además esta conexión no se perderá aunque se envuelva la GUI en varias capas para extenderla.

La librería ya se encarga de enlazar de esta forma el nombre del componente con el nombre que muestra como nodo o pestaña, por lo que solo habrá que enlazar el campo de texto con el nombre del componente. Esto se realiza en el panel base por un lado sobreescribiendo el método *setName* para que actualice el campo de texto, y por otro añadiéndole un listener al campo de texto para que actualice el nombre cada vez que cambie su contenido, según muestra el listado B.3.

Listado de código B.3: Enlace del campo de texto con el nombre del componente.

```
1  @Override
2  public void setName(String name) {
3    clave.setText(name);
4  }
5
6  clave.getDocument().addDocumentListener(new DocumentListener() {
7
8    @Override
9    public void changedUpdate(DocumentEvent e) {
10     PanelBase.super.setName(clave.getText());
11     clave.grabFocus();
12   }
13
14   @Override
15   public void insertUpdate(DocumentEvent e) {
16     PanelBase.super.setName(clave.getText());
17     clave.grabFocus();
18   }
19
20   @Override
21   public void removeUpdate(DocumentEvent e) {
22     PanelBase.super.setName(clave.getText());
23     clave.grabFocus();
24   }
25 });
```

B.7. Paso de mensajes

Una vez creada la interfaz, ya solo resta añadirle la lógica a los botones para que inserten o eliminen GUIs. Estos botones y la GUI con vista de árbol se encuentran en ventanas independientes por lo que será necesario comunicarlos mediante un sistema de paso de mensajes.

En el panel de control, los botones enviarán un mensaje cada vez que se pulsen. Haciendo doble click en los botones en la vista de diseño se crea automáticamente el esqueleto para añadirles ese código, que se muestra en el listado B.4.

En la GUI con vista de árbol se registra un listener para cada uno de estos botones que ejecute las acciones correspondientes cuando reciba el mensaje. Cuando se reciba el mensaje "Eliminar" se borrará la GUI seleccionada en la vista de árbol, y cuando se reciba "Insertar" se creará y anidará un nuevo panel base en la GUI seleccionada. Si se anida como nodo simplemente será añadida como nodo hijo del nodo seleccionado en la jerarquía de nodos, mientras que si se anida como *StackPanel* o *TabsPanel*, si la GUI seleccionada es ya de ese tipo se extiende con la nueva GUI y si no se envuelve con la clase correspondiente para poder ser extendida de esta forma. Si no está seleccionada ninguna

Listado de código B.4: Envío de mensajes en los botones del panel de control.

```
1 private void insertarActionPerformed(ActionEvent evt) {
2   Messenger.sendMessage("Insertar", new String[]{(String) tipo.
      getSelectedItem(), nombre.getText()});
3 }
4
5 private void eliminarActionPerformed(ActionEvent evt) {
6   Messenger.sendMessage("Eliminar", null);
7 }
```

GUI, como ocurre al principio, los nodos serán insertados como hijos de la raíz del árbol pero el resultado no será visible en cualquiera de los otros 2 modos al no haber una GUI para extender, por lo que siempre que esté el árbol vacío se deberá empezar anidando una GUI como nodo. Además se establece como nombre de la GUI el que se escribió en el panel de control, con lo que podrá ser identificada en la vista de nodos o de pestañas. El listado B.5 recoge el código resultante.

Listado de código B.5: Recepción de los mensajes en la GUI con vista de árbol.

```
1  Messenger.addListener("Eliminar", new MessageReceiver() {
2    @Override
3    public void receive(Object key, Object message) {
4      MutableTreeNode seleccionado = getSelectedNode();
5      if (seleccionado != null) {
6        remove(seleccionado);
7      }
8    }
9  });
10
11 Messenger.addListener("Insertar", new MessageReceiver() {
12   @Override
13   public void receive(Object key, Object message) {
14     String[] mensaje = (String[]) message;
15     String tipo = mensaje[0];
16     PanelBase panelBase = new PanelBase();
17     panelBase.setName(mensaje[1]);
18     MutableTreeNode nodo = getSelectedNode();
19     if (nodo == null) {
20       nodo = getRoot();
21     }
22     switch (tipo) {
23       case "Node":
24         insertIntoNode(panelBase, nodo);
25         break;
26       case "StackPanel":
27         StackPanel stack;
28         if (getSelectedComponent() instanceof StackPanel) {
29           stack = (StackPanel) getSelectedComponent();
30         } else {
31           stack = new StackPanel(getSelectedComponent());
32         }
33         stack.addChild(panelBase);
34         nodo.setUserObject(stack);
35         break;
36       case "TabsPanel":
37         TabsPanel tabs;
38         if (getSelectedComponent() instanceof TabsPanel) {
39           tabs = (TabsPanel) getSelectedComponent();
40         } else {
41           tabs = new TabsPanel(getSelectedComponent());
42         }
43         tabs.addChild(panelBase);
44         nodo.setUserObject(tabs);
45         break;
46     }
47   }
48 });
```