UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática

Implementación del Patrón de Diseño de Reusabilidad de GUIs.

FRANCISCO JAVIER SANCHEZ LOZANO

Dirigido por: MANUEL ARIAS CALLEJA

Curso: 2020-2021: 2ª Convocatoria

Implementación del Patrón de Diseño de Reusabilidad de GUIs.

Proyecto de Fin de Grado de modalidad oferta general

Realizado por: Francisco Javier Sánchez Lozano

Dirigido por: Manuel Arias Calleja

Tribunal calificador

Presidente: D/Dª.

Secretario: D/Dª.

Vocal: D/Dª.

Fecha de lectura y defensa:

Calificación:

Agradecimientos

Quisiera dedicar este proyecto a todas las personas que con sus consejos y enseñanzas me ayudaron a crecer y ser una persona mas sabia , una mejor persona.

Resumen

El desarrollo del Software nunca ha sido una tarea sencilla. Se ha requerido al menos de un editor de texto, donde escribir el programa, y de un compilador o intérprete que diera los pasos necesarios con el conjunto de ficheros del programa para que éste se pudiera ejecutar. Además, la práctica totalidad de los programas hoy en día disponen de un Interfaz Grafica (GUI) , que consiste en un conjunto de ventanas y menús que permiten al usuario interactuar con el software y proporcionarle una experiencia lo más satisfactoria posible.

Actualmente la codificación de programas y la creación de las Interfaces graficas se pueden crear a través de los Entornos Integrados de desarrollo (IDE) en su vista de Codificación y en la vista de Diseño, respectivamente. Esto guarda una estrecha relación con el estilo de arquitectura de software denominado Modelo Vista Controlador (MVC), que separa los datos de usuario de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes diferentes. De esta forma, se distinguen dos roles principales: el diseñador de GUI y el desarrollador de aplicaciones.

Este proyecto tiene como principal objetivo facilitar el desarrollo de Software a través una Libreria que permita la reutilización de las interfaces graficas de usuario. La librería aporta la funcionalidad de una jerarquía de interfaces gráficas y los mecanismos para integrar interfaces gráficas de objetos asociados en una GUI general, pero conservando la posibilidad de mostrar cuadro individual cuando fuera necesario sin duplicación de código ni de componentes gráficos. El proceso es transparente para usuarios y programadores, tanto los que diseñan las GUIs como los que las usan al programar aplicaciones. Ademas también se aportan funcionalidades que permiten el paso de mensajes entre las GUI y la recursividad de las operaciones de validación de la edición, grabación de datos y finalización.

Con las funcionalidades que aporta la Libreria se espera conseguir que la tarea del desarrollo de Software sea un poco mas sencilla.

Palabras clave

Librería | Reusabilidad |Interfaces gráficas de usuario(GUI) | Java | Swing | NetBeans

Abstract

Software development has never been a simple task. At least a text editor has been required, where to write the program, and a compiler or interpreter that would take the necessary steps with the set of files of the program so that it could be executed. In addition, almost all programs today have a Graphical Interface (GUI), which consists of a set of windows and menus that allow the user to interact with the software and provide the most satisfactory experience possible.

Currently the coding of programs and the creation of the graphical interfaces can be created through the Integrated Development Environments (IDE) in its Coding view and in the Design view, respectively. This is closely related to the software architecture style called Model View Controller (MVC), which separates an application's user data, user interface, and control logic into three different components. In this way, two main roles are distinguished: the GUI designer and the application developer. This project aims to make the task of both roles easier.

The main objective of this project is to facilitate the development of Software through a Library that allows the reuse of graphical user interfaces. The library provides the functionality of a hierarchy of graphical interfaces and the mechanisms for integrating graphical interfaces of associated objects in a general GUI, but preserving the possibility of displaying individual tables when necessary without duplicating code or graphical components. The process is transparent to users and programmers, both those who design the GUIs and those who use them when programming applications. In addition, functionalities are also provided that allow the passage of messages between the GUIs and the recursion of the validation operations of the edition, data recording and completion.

With the functionalities that the Library provides, it is hoped to make the task of Software development a little easier.

Keywords

Library | Reusability | Graphical User Interfaces (GUI) | Java | Swing | NetBeans

Índice general

Nomenclatura

POO Programación Orientada a Objetos, página 55

Índice de figuras

Índice de tablas

Capítulo 1

Introducción general y objetivos

Este capitulo tiene como cometido explicar de una forma general la motivación y objetivos del proyecto. Se pretende dar una idea global del proceso de creación de la Librería que detalla esta memoria.

Primero se hará una breve introducción que sirva de contexto. En ella se comenta como funciona normalmente la tarea del desarrollo de Software y la importancia que tienen las GUI (interfaces graficas de Usuario) en muchos de los proyectos de desarrollo.

Después de esta corta introducción se pasa a explicar la motivación principal de este proyecto: las Interfaces graficas Reusables, la herramienta que debe ayudar tanto a programadores y diseñadores de GUI a hacer su tarea mas sencilla.

Una vez que se ha presentado la motivación del proyecto, seguidamente se pasa a explicar los objetivos básicos que pretende cumplir la librería.

Finalmente se comenta la estructura general de la memoria con la lista de capítulos y anexos, con una descripción corta de cada uno. Se pretende que sirva como una pequeña síntesis de los contenidos que aborda este documento.

1.1. Introducción

El desarrollo de software ha sido siempre una tarea tediosa y difícil que ha requerido al menos de un editor de texto, donde escribir el programa, y de un compilador o intérprete que diera los pasos necesarios con el conjunto de ficheros del programa para que éste se pudiera ejecutar.

Además, la práctica totalidad de los programas hoy en día disponen de un interfaz de gráfico, que consiste en un conjunto de ventanas y menús que permiten al usuario interactuar con el software y proporcionarle una experiencia lo más satisfactoria posible. Dadas las crecientes exigencias de los usuarios por disponer de interfaces de usuario amigables surgieron hace unos años distintos programas, conocidos como entornos de desarrollo (siglas IDEs en inglés), orientados a facilitar el trabajo al programador que debía desarrollar dicho software. **Figura 1.1**

En todos ellos se suelen distinguir dos vistas o perspectivas desde las cuales explorar y/o editar el código fuente del programa:

Vista de codificación: En la que el programador accede directamente al código fuente del programa, que es al fin y al cabo el código que el compilador o intérprete procesará para que se pueda ejecutar el programa. La típica presentación de esta vista es un documento de texto, con la sintaxis del lenguaje coloreada. **Figura 1.2**

Vista de diseño: En la que el programador puede construir de manera gráfica las distintas ventanas y menús que le aparecerán en el programa, en la que añadirá botones, cajas de texto, marcos con borde, etc., o sea, todo tipo de componentes gráficas que quiere que se muestren en algún momento en el programa. **Figura 1.3**

La introducción de la posibilidad de disponer de la vista de diseño en los IDEs fue un gran avance para los programadores y les permitió desarrollar las aplicaciones de forma más rápida, con menos esfuerzo, y ofrecer programas de mayor calidad. Guarda una estrecha relación con el estilo de arquitectura de software denominado Modelo Vista Controlador (MVC), que separa los datos de usuario de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes diferentes.

1. La Vista, que será la interfaz de usuario, compuesta por las diferentes pantallas y ventanas.

2. El Modelo, los datos e información que manejar y representa la aplicación.

3. El Controlador, que contiene la lógica que comunique la vista con el controlador.

El diseñador de GUI va a trabajar sobre la parte vista según el paradigma MVC, con lo que se encarga del desarrollo de toda la parte de la GUI. El desarrollador de aplicaciones se centra en las partes Modelo y Controlador según el esquema MVC.

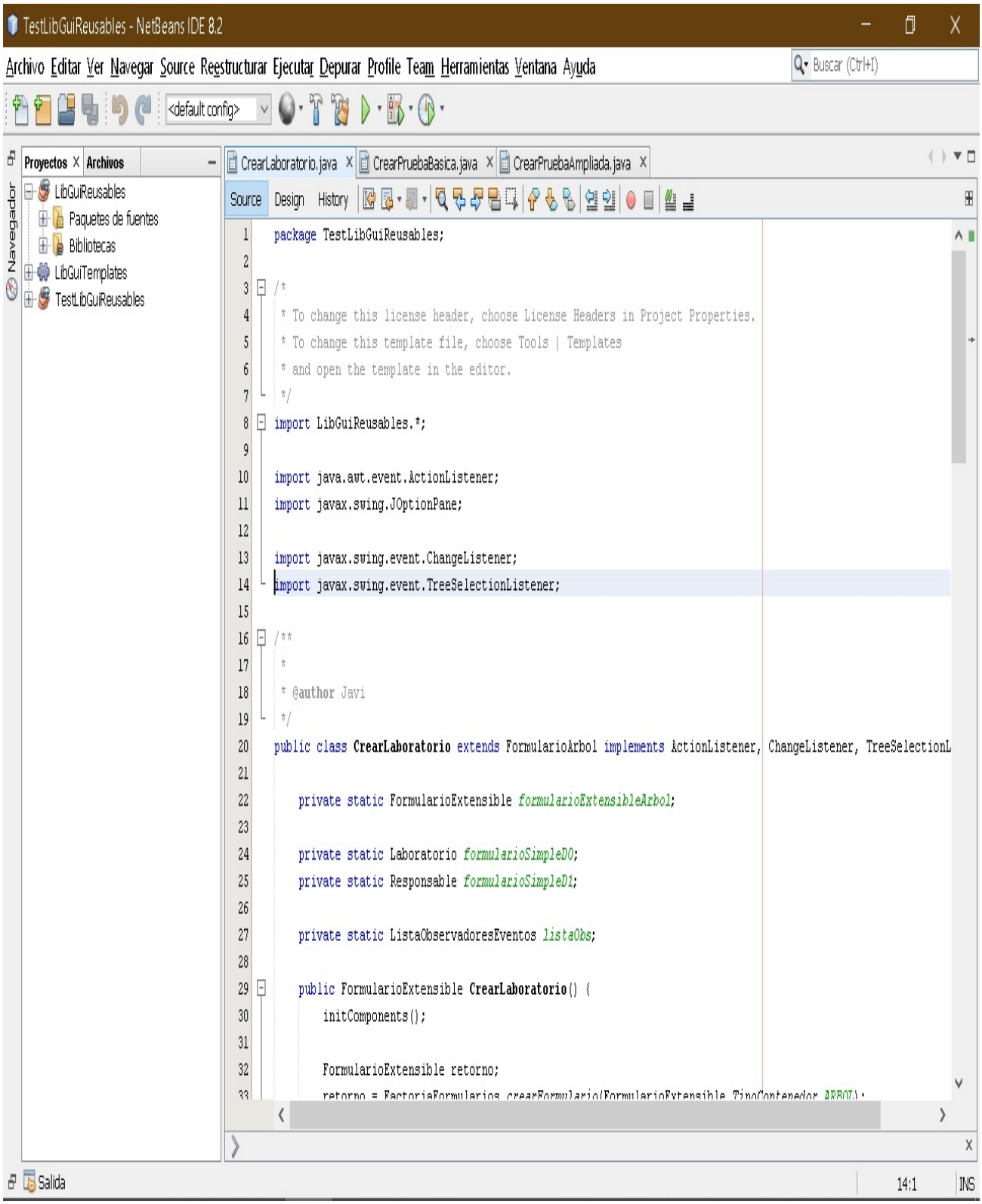

Figura 1.1: IDE

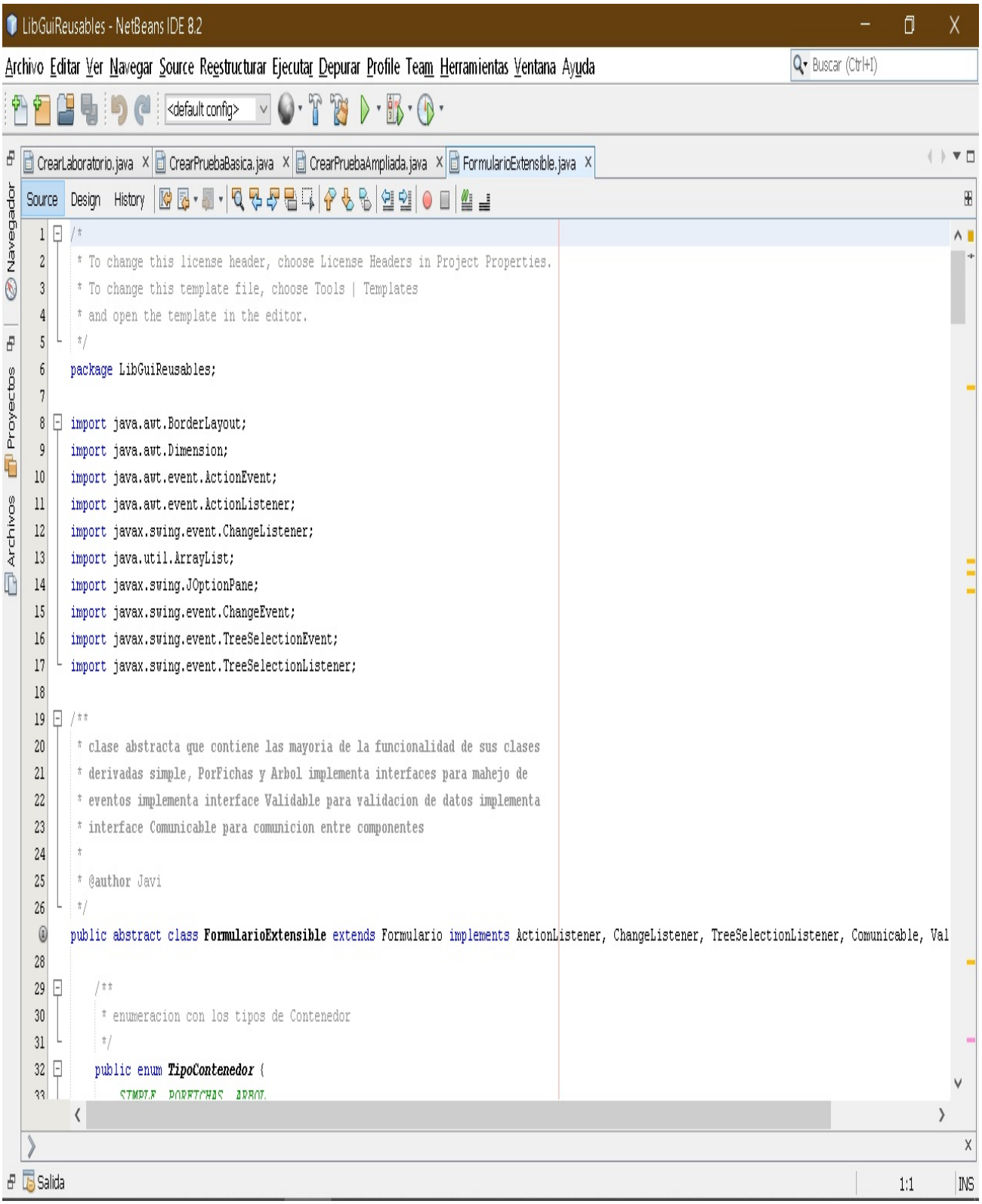

Figura 1.2: Vista codificación

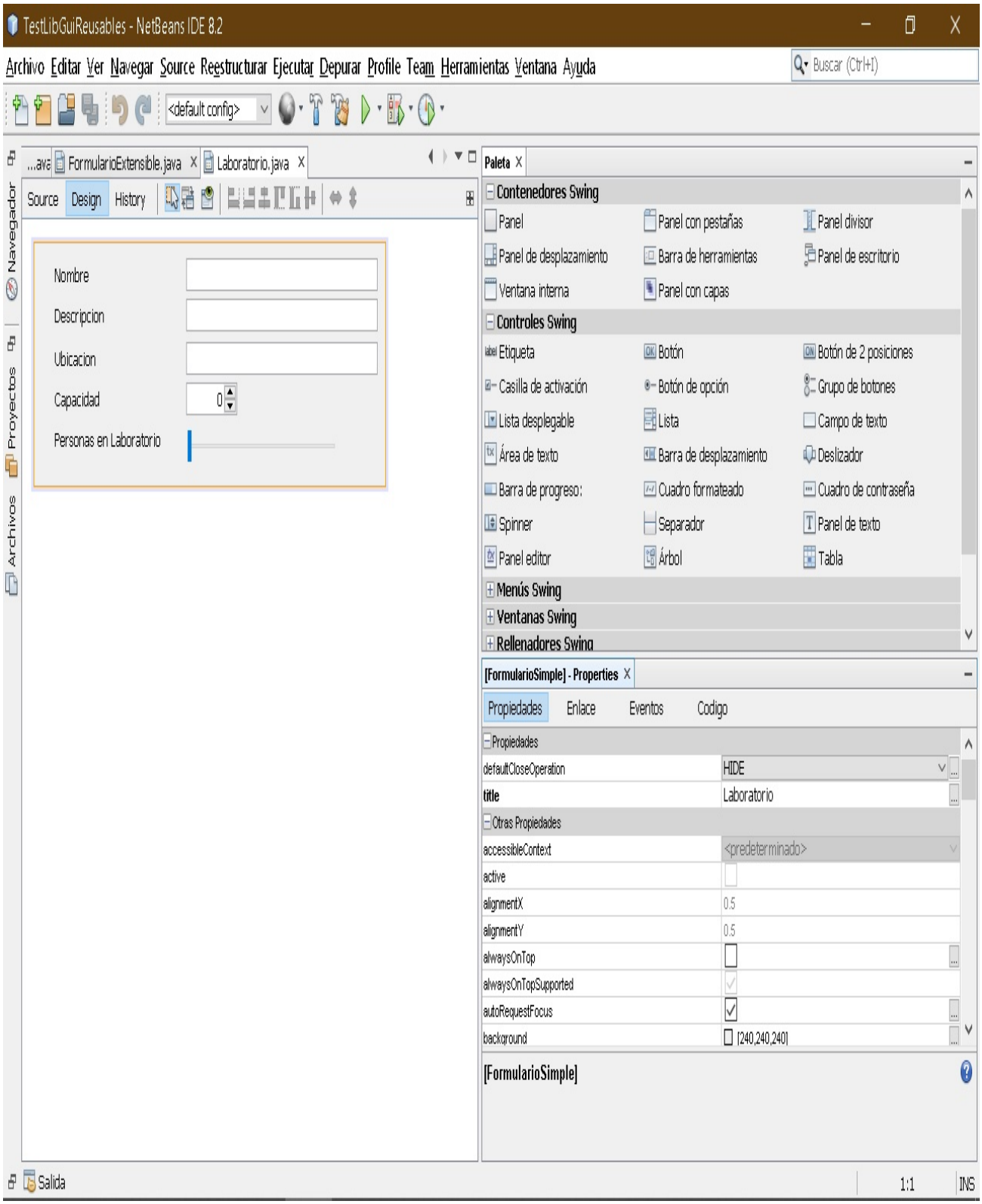

Figura 1.3: Vista diseño

Las GUI son un parte fundamental de la mayoria de proyectos de Software, crear alguna herramienta que facilite su reusabilidad puede hacer que la tarea de Desarrollo ocupe menos tiempo y no sea tan tediosa y difícil.

1.2. Motivación del proyecto

Después de alguna fase del desarrollo de un Proyecto de Software se llegó a la conclusión de que era necesario alguna herramienta que permitiera la reusabilidad de las interfaces gráficas (GUIs) para facilitar estas tareas.

Esta necesidad surgió en dos escenarios distintos, que pueden aparecer mezclados:

1. Crear una jerarquía de interfaces gráficas, generalmente diseñados con un diseñador gráfico, para editar una jerarquía de clases de manera que no se duplicara código ni componentes gráficos (widgets).

2. Integrar interfaces gráficas de objetos asociados en una GUI general, pero conservando la posibilidad de mostrar cada cuadro individual cuando fuera necesario, de nuevo sin duplicación de código ni de componentes gráficos. Además el proceso debería ser transparente para usuarios y programadores, tanto los que diseñan las GUIs como los que las usan al programar aplicaciones.

Esta necesidad con los escenarios comentados ha dado lugar a la motivación del proyecto, las Interfaces graficas Reusables. Para ello, se plantea la construcción de una librería que permita la reutilización, por parte del programador, de interfaces gráficas de usuario que hayan sido creadas por el diseñador.

1.3. Objetivos

El objetivo principal del proyecto es crear una librería con la jerarquía de interfaces graficas que integre interfaces graficas de objetos asociados, resolviéndolo en Java/Swing con el IDE Netbeans.

Desde el punto de vista del desarrollador de aplicaciones, éste pedirá diálogos vacíos a la librería, inicializará los diálogos que le proporcione la librería, y añadirá un diálogo hijo a un diálogo padre. Además, pedirá diálogos creados por el diseñador de aplicaciones. Todas estas tareas las realizará en la vista de codificación del IDE.

Desde el punto de vista del diseñador de GUIs, la librería debe permitir que sus distintos tipos de diálogos puedan ser usados en la vista de diseño del IDE. De esta forma, el diseñador de GUIs

arrastrará uno de esos diálogos a la hoja al tapiz de diseño del IDE y podrá trabajar sobre él añadiendo componentes y dándole el aspecto que desee.

Ademas la Libreria debe proporcionar un sistema de comunicacion de mensajes entre dialogos. Dado que se van a añadir diálogos hijos a un padre, y a su vez los hijos pueden tener hijos, al final podemos imaginar que va a haber como una especie de árbol donde la raíz es el diálogo principal. Es importante señalar que en el árbol que abarca a toda la estructura de diálogos del diálogo principal puede haber paso de mensajes entre cada par de diálogos del árbol, y además ese paso de mensajes ha de ser posible realizarlo en las dos direcciones.

1.4. Estructura de la memoria

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

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

Introducción general y objetivos

Este es un capítulo introductorio, que se encarga de proporcionar una primera toma de contacto al lector que se dispone a leer la memoria.

Fundamentos teóricos

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

Análisis y diseño del sistema

Se corresponde con dos fases típicas de todo desarrollo de software, como son las fases de Análisis y Diseño del sistema bajo construcción. Se usara el lenguaje de modelado UML, para representar algunos tipos de diagramas. (casos de uso, clases, interacción)

Implementación

Aquí se hace una explicación general del código desarrollado, describiendo las clases que componen el sistema y su funcionamiento.

Experimentación y pruebas

En este apartado se comentan los experimentos y pruebas a los que se ha visto sometido la librería. Se detalla el funcionamiento del programa de pruebas pedido y realizado.

Capítulo 2

Fundamentos teoricos

Aqui va la descripcion del capitulo

2.1. Programación Orientada a Objetos

La Programación Orientada a es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas. Está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo, y encapsulamiento. Su uso se popularizó a principios de la década de 1990. Actualmente son muchos los lenguajes de programación que soportan la orientación a objetos.

El término de Programación Orientada a Objetos indica más una forma de diseño y una metodología de desarrollo de software que un lenguaje de programación, ya que en realidad se puede aplicar el Diseño Orientado a Objetos, a cualquier tipo de lenguaje de programación.

Se puede hablar de Programación Orientada a Objetos cuando se reúnen las características de: abstracción, encapsulación, herencia y polimorfismo; y los conceptos básicos que las forman: objetos, mensajes, clases, instancias y métodos.

2.1.1. Conceptos Básicos de Programación Orientada a Objetos

1. Un **objeto** es una encapsulación abstracta de información, junto con los métodos o procedimientos para manipularla. Un objeto contiene operaciones que definen su comportamiento y variables que definen su estado entre las llamadas a las operaciones.

2. Una **clase** equivale a la generalización o abstracción de un tipo específico de objetos.

3. Un **mensaje** representa una acción a tomar por un determinado objeto.

4. Una **instancia** es la concrección de una clase.

5. Un **método** consiste en la implementación en una clase de un protocolo de respuesta a los mensajes dirigidos a los objetos de la misma. La respuesta a tales mensajes puede incluir el envío por el método de mensajes al propio objeto y aun a otros, también como el cambio del estado interno del objeto.

2.1.2. Caracteristicas de Programación Orientada a Objetos

2.1.2.1. Abstracción

Son las características especificas de un objeto, aquellas que lo distinguen de los demás tipos de objetos y que logran definir límites conceptuales respecto a quien está haciendo dicha abstracción del objeto. Se enfoca en la visión externa de un objeto, separa el comportamiento específico de un objeto, a esta división que realiza se le conoce como la barrera de abstracción, la cuál se consigue aplicando el principio de mínimo compromiso; que se refiere al proceso por el cuál la interfaz de un objeto muestra su comportamiento específico y nada más, absolutamente nada más.

Una interfaz de objeto permite crear código con el cuál se específica que métodos serán implementados por una clase sin necesidad de definir que harán estos métodos, dichos métodos deben ser públicos.

Las clases abstractas, como su nombre lo indica, son algo abstracto, no representan algo específico y las podemos usar para crear otras clases. No pueden ser instanciadas, por lo que no podemos crear nuevos objetos con ellas.

2.1.2.2. Encapsulación

Se refiere a la capacidad de agrupar y condensar en un entorno con límites bien definidos distintos elementos. La encapsulación se encarga de mantener ocultos los procesos internos que necesita para hacer lo que sea que haga, dándole al programador acceso sólo a lo que necesita.

Típicamente, el encapsulamiento es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos. Entonces, los detalles de la puesta en práctica pueden cambiar en cualquier tiempo sin afectar otras partes del programa.

2.1.2.3. Herencia

Se aplica sobre las clases. O sea, de alguna forma las clases pueden tener descendencia, y ésta heredará algunas características de las clases "padres". Si disponemos las clases con un formato de árbol genealógico, tenderemos lo que se denomina una estructura jerarquizada de clases.

La POO promueve en gran medida que las relaciones entre objetos se basen en construcciones jerárquicas. Esto es, las clases pueden heredar diferencialmente de otras clases (denominadas "superclases") determinadas características, mientras que, a la vez, pueden definir las suyas propias. Tales clases pasan, así, a denominarse "subclases" de aquéllas.

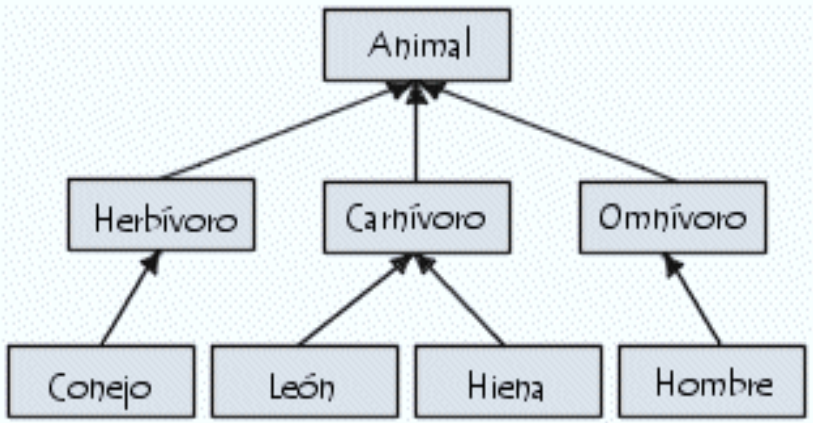

Figura 2.1: Ejemplo Herencia

La herencia se implementa mediante un mecanismo que se denomina derivación de clases: las superclases pasan a llamarse clases base, mientras que las subclases se constituyen en clases derivadas. El mecanismo de herencia está fuertemente relacionado con la **reutilización del código** en POO. Una clase derivada posibilita, el fácil uso de código ya creado en cualquiera de las clases base ya existentes.

2.1.2.4. Polimorfismo

Esta propiedad, como su mismo nombre sugiere múltiples formas, se refiere a la posibilidad de acceder a un variado rango de funciones distintas a través del mismo interfaz. O sea, que, en la práctica, un mismo identificador puede tener distintas formas (distintos cuerpos de función, distintos comportamientos) dependiendo, en general, del contexto en el que se halle inserto. El polimorfismo se puede establecer mediante la **sobrecarga** de identificadores y operadores, la ligadura dinámica y las funciones virtuales. El término sobrecarga se refiere al uso del mismo identificador u operador en distintos contextos y con distintos significados.

2.1.3. Beneficios de Programación Orientada a Objetos

1. Reutilización del código.
2. Convierte cosas complejas en estructuras simples reproducibles.
3. Evita la duplicación de código.

4. Permite trabajar en equipo gracias al encapsulamiento ya que minimiza la posibilidad de duplicar funciones cuando varias personas trabajan sobre un mismo objeto al mismo tiempo.

5. Al estar la clase bien estructurada permite la corrección de errores en varios lugares del código.

6. Protege la información a través de la encapsulación, ya que solo se puede acceder a los datos del objeto a través de propiedades y métodos privados.

7. La abstracción nos permite construir sistemas más complejos y de una forma más sencilla y organizada.

2.2. Reusabilidad

Realmente es un conjunto de patrones y otras técnicas de la programación orientada a objeto que facilitan la reutilización de Sofware.

A traves de estas tecnicas de Reusabilidad se consiguen grandes beneficios:

1. **Confiabilidad**: el software de mayor calidad y fiabilidad.

2. **Eficiencia**: mejores algoritmos y estructuras de datos posibles.

3. **Ahorro de tiempo**: las aplicaciones se pueden construir más rápido.

4. **Disminución del esfuerzo de mantenimiento**: menor esfuerzo de mantenimiento que el desarrollador de aplicaciones tiene que realizar

5. **La consistencia**: la información se mantiene consistente y correcta

6. **Inversión**: permite ahorrar el coste de desarrollar un software similar desde cero.

2.3. Patrones de diseño

Es una solución general y reutilizable aplicable a diferentes problemas de diseño de software. Se trata de plantillas que identifican problemas en el sistema y proporcionan soluciones apropiadas a problemas generales a los que se han enfrentado los desarrolladores durante un largo periodo de tiempo, a través de prueba y error.Java es un lenguaje de programación basado en clases y orientado a objetos. Existen diversos índices de lenguajes de programación y dependiendo el que tomemos como referencia puede considerarse el lenguaje más popular, o uno de los 3 más populares que existen en el mundo.

En 1994, cuatro autores Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, a los que llamaron Gang of Four (GoF), publicaron un libro titulado Design Patterns, elementos de software orientado a objetos reutilizables. Con este trabajo se inició el concepto de patrón de diseño en el desarrollo de software y recoge 23 patrones de diseño comunes. Cada uno de ellos define la solución para resolver un determinado problema, facilitando además la reutilización del código fuente.

Los patrones de diseño ayudan a estar seguro de la validez de tu código, ya que son soluciones que funcionan y han sido probados por muchísimos desarrolladores siendo menos propensos a errores.

2.3.1. Tipos de patrones de diseño

Existen diversas maneras de agrupar los patrones de diseño. Quizá la más extendida es agruparlos según su propósito. En este caso tendríamos las siguientes categorías:

2.3.1.1. Patrones Creacionales

Como su nombre indica, estos patrones vienen a solucionar o facilitar las tareas de creación o instanciación de objetos. Estos patrones hacen hincapié en la encapsulación de la lógica de la instanciación, ocultando los detalles concretos de cada objeto y permitiéndonos trabajar con abstracciones.

Algunos de los patrones creacionales más conocidos son:

1. **Factory**: Desacoplar la lógica de creación de la lógica de negocio, evitando al cliente conocer detalles de la instanciación de los objetos de los que depende.

2. **Abstract Factory**: Nos provee una interfaz que delega la creación de una serie de objetos relacionados sin necesidad de especificar cuáles son las implementaciones concretas.

3. **Factory Method**: Expone un método de creación, delegando en las subclases la implementación de este método.

4. **Builder**: Separa la creación de un objeto complejo de su estructura, de tal forma que el mismo proceso de construcción nos puede servir para crear representaciones diferentes.

2.3.1.2. Patrones estructurales

Los patrones estructurales nos ayudan a definir la forma en la que los objetos se componen.

Los patrones estructurales más habituales son:

1. **Adapter**: Nos ayuda a definir una clase intermedia que sirve para que dos clases con diferentes interfaces puedan comunicarse. Esta clase actúa como mediador, haciendo que la clase A pueda ejecutar métodos de la clase B sin conocer detalles de su implementación. También se conoce como Wrapper.
2. **Decorator**: Permite añadir funcionalidad extra a un objeto (decora el objeto) sin modificar el comportamiento del resto de instancias.
3. **Facade**: Una fachada es un objeto que crea una interfaz simplificada para tratar con otra parte del código más compleja.
4. **Composite**: Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

2.3.1.3. Patrones comportamentales

Los patrones comportamentales nos ayudan a definir la forma en la que los objetos interactúan entre ellos.

Algunos de los más conocidos (por citar unos pocos) son:

1. **Command**: Son objetos que encapsulan una acción y los parámetros que necesitan para ejecutarse.
2. **Observer**: Los objetos son capaces de suscribirse a una serie de eventos que otro objeto va a emitir, y serán avisados cuando esto ocurra.
3. **Strategy**: Permite la selección del algoritmo que ejecuta cierta acción en tiempo de ejecución.
4. **Template Method**: Especifica el esqueleto de un algoritmo, permitiendo a las subclases definir cómo implementan el comportamiento real.

2.3.2. Patrones relevantes para el diseño de la Libreria

2.3.2.1. Patrón Factory Method

El patrón de diseño Factory Method nos permite la creación de un subtipo determinado por medio de una clase de Factoría, la cual oculta los detalles de creación del objeto.

El objeto real creados es enmascarado detrás de una interface común entre todos los objetos que pueden ser creado, con la finalidad de que estos pueden variar sin afectar la forma en que el cliente interactúa con ellos.

Es normal que un Factory pueda crear varios subtipos de una determinada interface y que todos los objetos concretos fabricados hagan una tarea similar pero con detalles de implementación diferentes.

La intención del Factory Method es tener una clase a la cual delegar la responsabilidad de la creación de los objetos, para que no sea el mismo programador el que decida que clase instanciará, si no que delegará esta responsabilidad al Factory confiando en que este le regresará la clase adecuada para trabajar.

2.3.2.2. Patrón Composite

El concepto básico del patrón Composite consiste en representar objetos simples y sus containers (o contenedores, también llamados colecciones en algunos lenguajes, o sea: grupos de objetos) en una clase abstracta de manera que puedan ser tratados uniformemente. Este tipo de estructura se conoce como jerarquía parte-todo (en inglés: part-whole hierarchy), en la que un objeto es siempre, o una parte de un todo, o un todo compuesto por varias partes.

El patrón Composite requiere mínimo de tres componentes :

1. Component: Generalmente es una interface o clase abstracta la cual tiene las operaciones mínimas que serán utilizadas, este componente deberá ser extendido por los otros dos componentes Leaf y Composite.

2. Leaf: El leaf u hoja representa la parte más simple o pequeña de toda la estructura y este extiende o hereda de Component.

3. Composite: los Composite tiene los métodos add y remove los cuales nos permiten agregar objetos de tipo Component, Sin embargo como hablamos anteriormente, el Componente es por lo general un Interface o Clase abstracta por lo que podremos agregamos objetos de tipo Composite o Leaf.

La finalidad del patrón Composite es, como en todos los patrones "GoF", mejorar la gestión de problemas de diseño recurrentes en la programación orientada a objetos. El resultado deseado es un software flexible, caracterizado por objetos fácilmente implementables, intercambiables, reutilizables y testeables. A tal efecto, el patrón de diseño Composite describe un método según el cual los objetos simples y complejos pueden ser tratados de la misma manera. De este modo, se puede crear una estructura de objetos fácilmente inteligible que permite al cliente un acceso altamente eficiente. Además, también se minimiza la probabilidad de error en el código.

2.3.2.3. Patrón Observer

El patrón de diseño Observer, es uno de los patrones de diseño de software más populares. Esta herramienta ofrece la posibilidad de definir una dependencia uno a uno entre dos o más objetos para transmitir todos los cambios de un objeto concreto de la forma más sencilla y rápida posible. Para conseguirlo, puede registrarse en un objeto (observado) cualquier otro objeto, que funcionará como observador. El primer objeto, también llamado sujeto, informa a los observadores registrados cada vez que es modificado.

El patrón Observer trabaja con dos tipos de actores: por un lado, el sujeto, es decir, el objeto cuyo estado quiere vigilarse a largo plazo. Por otro lado, están los objetos observadores, que han de ser informados de cualquier cambio en el sujeto. El patrón de diseño Observer se implementa sobre todo en aplicaciones basadas en componentes cuyo estado, por un lado, es muy observado por otros componentes y, por otro, es modificado regularmente.

Algunos de los casos de aplicación típicos de este patrón son las GUI (interfaces gráficas de usuario), que actúan como interfaz de comunicaciónde manejo sencillo entre los usuarios y el programa. Cada vez que se modifican los datos, estos deben actualizarse en todos los componentes de la GUI. Esta situación es perfecta para la aplicación de la estructura sujeto-observador del patrón Observer. Incluso los programas que trabajan con conjuntos de datos en formato visual (ya sean tablas clásicas o diagramas gráficos) pueden beneficiarse de la estructura de este patrón de diseño.

2.3.2.4. Patrón Indireccion

Problema

¿Dónde asignar responsabilidades para evitar/reducir el acoplamiento directo entre elementos y mejorar la reutilización?

Solución

Asigne la responsabilidad a un objeto que medie entre los elementos.

Forma parte de un conjunto de patrones o principios de diseño conocidos como GRASP. Se trata de asignar la responsabilidad a un objeto intermedio para mediar entre otros componentes o servicios, para que no estén directamente acoplados. El intermediario es el encargado de recibir mensajes desde una de las entidades, o enviar mensajes hacia la otra entidad, de forma bidireccional.

Sirve de base a muchos otros patrones que aplican el mismo concepto, como puede ser el patrón Adaptador, o incluso el mismo patrón Observador.

2.4. Interfaces graficas de Usuario

La GUI es una interfaz de usuario que permite a los usuarios comunicarse con el ordenador. Suele estar basada en la interacción a través del ratón y el teclado (aunque el control a través de gestos es cada vez más común): al mover el ratón, el puntero se desplaza en la pantalla. La señal del dispositivo se transmite al ordenador, que luego la traduce en un movimiento equivalente en la pantalla. Por ejemplo, si un usuario hace clic en un determinado icono de programa en el menú, se ejecuta la instrucción correspondiente y se abre el programa.

Una GUI combina el diseño visual y las funciones de programación. Por esto, ofrece botones, menús desplegables, campos de navegación, campos de búsqueda, iconos y widgets. Los desarrolladores deben tener siempre en cuenta la facilidad de uso.

La GUI suele ser un trabajo en conjunto entre desarrolladores y diseñadores que buscan la mejor manera de que el usuario pueda interactuar con el programa mediante elementos visuales fáciles de comprender.

2.4.1. Modelo–Vista–Controlador

Modelo-vista-controlador (MVC) es un patrón de arquitectura de software, que separa los datos y principalmente lo que es la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.

Modelo El modelo es un conjunto de clases que representan la información del mundo real que el sistema debe reflejar. Es la parte encargada de representar la lógica de negocio de una aplicación. Así, por ejemplo, un sistema de administración de datos geográficos tendrá un modelo que representara la altura, coordenadas de posición, distancia, etc. sin tomar en cuenta ni la forma en la que esa información va a ser mostrada ni los mecanismos que hacen que esos datos estén dentro del modelo, es decir, sin tener relación con ninguna otra entidad dentro de la aplicación.

Vista Las vistas son las encargadas de la representación de los datos, contenidos en el modelo, al usuario. La relación entre las vistas y el modelo son de muchas a uno, es decir cada vista se asocia a un modelo, pero pueden existir muchas vistas asociadas al mismo modelo.

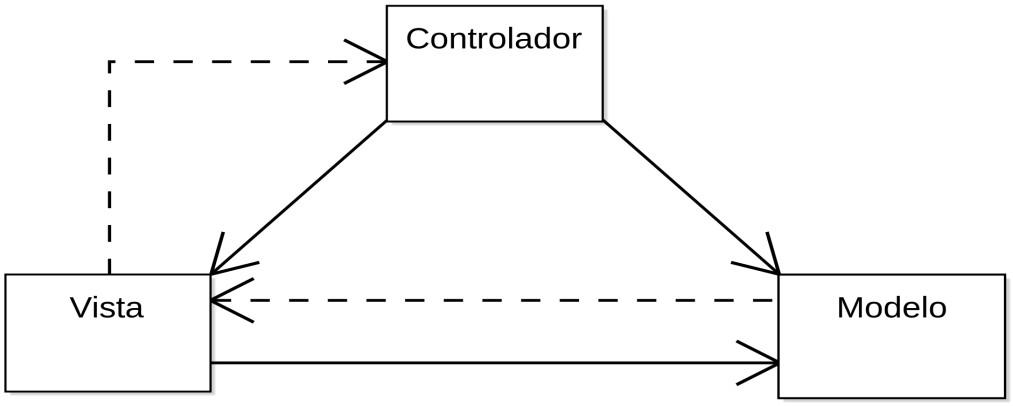

Figura 2.2: MVC

Controlador El controlador es el encargado de interpretar y dar sentido a las instrucciones que realiza el usuario, realizando actuaciones sobre el modelo. Si se realiza algún cambio, comienza a actuar, tanto si la modificación se produce en una vista o en el modelo. Interactúa con el Modelo a través de una referencia al propio Modelo.

2.5. Entorno de Desarrollo de la Librería (Java/Swing/Netbeans)

2.5.1. Java

Java es un lenguaje de programación basado en clases y orientado a objetos. Existen diversos índices de lenguajes de programación y dependiendo el que tomemos como referencia puede considerarse el lenguaje más popular, o uno de los 3 más populares que existen en el mundo.

Su objetivo es, permitir que los desarrolladores de aplicaciones escriban el código una sola vez y que ese código pueda se ejecutado en cualquier lugar, esto es lo que denominan WORA (Write once, run anywhere? / Escribe una vez, corre en cualquier lugar).

Java ofrece muchas características atractivas:

1. Un lenguaje independiente de la plataforma.
2. Una amplia biblioteca estándar que facilita la codificación.
3. Puede crear una aplicación independiente completa utilizando Java.
4. Java admite la asignación automática de memoria y la desasignación (llamada recolección de basura).

5. Ofrece un gran rendimiento ya que Java admite multiproceso y concurrencia, lo que lo convierte en un lenguaje altamente interactivo y receptivo.

6. Seguro y simple.

2.5.2. Swing

El paquete Swing es parte de la JFC (Java Foundation Classes) en la plataforma Java. La JFC provee facilidades para ayudar a la gente a construir GUIs. Swing abarca componentes como botones, tablas, marcos, etc...

Posee las siguientes características principales:

1. Independencia de plataforma.

2. Extensibilidad: es una arquitectura altamente particionada: los usuarios pueden proveer sus propias implementaciones modificadas para sobrescribir las implementaciones por defecto. Se puede extender clases existentes proveyendo alternativas de implementación para elementos esenciales.

3. Personalizable: dado el modelo de representación programático del framework de Swing, el control permite representar diferentes estilos de apariencia.

2.5.3. Netbeans

Netbeans es un IDE (Integrated Development Environment) o entorno de desarrollo integrado, que es gratuito y de código abierto. Netbeans sirve para el desarrollo de aplicaciones web, corporativas, de escritorio y móviles que utilizan plataformas como Java y HTML5, entre otras.

NetBeans es un proyecto de código abierto de gran éxito con una gran base de usuarios, una comunidad en constante crecimiento.

La plataforma NetBeans permite que las aplicaciones sean desarrolladas a partir de un conjunto de componentes de software llamados módulos. Un módulo es un archivo Java que contiene clases de java escritas para interactuar con las APIs de NetBeans y un archivo especial (manifest file) que lo identifica como módulo. Las aplicaciones construidas a partir de módulos pueden ser extendidas agregándole nuevos módulos. Debido a que los módulos pueden ser desarrollados independientemente, las aplicaciones basadas en la plataforma NetBeans pueden ser extendidas fácilmente por otros desarrolladores de software.

Capítulo 3

Análisis y diseño del Sistema

Aquí va la descripción del capitulo

3.1. Requisitos del proyecto

A continuación se enumeran los requisitos del proyecto:

1. Los tipos de GUIs serán los siguientes:

a) GUIs de una sola página, que denominaremos simples, en las que se integran como máximo dos GUIs .

b) GUIs con navegación mediante pestañas.

c) GUIs con navegación mediante vista de árbol.

2. Cualquier tipo de GUI se debe poder anidar cualquier otro, aunque ciertas combinaciones no tengan mucho sentido. El nivel de anidamiento de GUIs, salvo para la simple, debe ser teóricamente ilimitado.

3. El proceso debe ser transparente para:

a) El usuario. La edición de una GUI debe ser considerada como una transacción atómica, por lo que lo editado en cada una de su páginas debe ser aceptado o rechazado con una sola acción, una sola pulsación de botón del usuario al finalizar la edición completa.

b) El diseñador de GUIs. Debe poder tratar estas GUIs como las normales de la librería gráfica que se use, salvo, quizás, algunos pocos métodos añadidos para la reutilización y teniendo en cuenta que si existe un mecanismo previo de extensión de GUIs en la librería gráfica debería ser explícitamente anulado para que no entre en conflicto el mecanismo de Reusabilidad aquí expuesto.

c) El programador de aplicaciones. Solo debe tener acceso a la interfaz pública de la jerarquía de GUIs reusables, que, además de la propia de las GUIs normales de la librería gráfica, debe constar de métodos para añadir GUIs, listas de GUIs relacionadas y para la inicialización propia de los GUIs reusables, aparte de la que puedan tener como GUIs normales. Además debe contar un procedimiento sencillo para el ensamblado de basado en métodos polimórficos, de manera que cada clase a editar pueda ensamblar convenientemente su propio GUI reusable, ya sea por invocación del los mismos métodos en las clases de niveles más básicos de su jerarquía y/o por invocación de métodos con igual propósito en los objetos asociados.

4. Se debe mantener en lo posible la estética de las GUIs diseñadas.

5. Debe existir un mecanismo para crear GUIs de los tres tipos requeridos en el punto 1, con la estructura mínima: botones para aceptar la edición y rechazarla, contenedor apropiado con distribución y vista de árbol para el último de los tipos requeridos. Estas GUIs las usa el programador de aplicaciones para envolver GUIs sin botones.

6. Comunicación entre GUIs integradas, hay que arbitrar un sistema de paso de mensajes entre GUIs.

7. Deseamos proteger lo más posible el funcionamiento interno de las clases de la biblioteca, nueva referencia a una interfaz pública reducida, y poder forzar al diseñador de GUIs la implementación de ciertos métodos, imprescindibles para el funcionamiento del proceso a implementar, de manera que su ausencia sea detectada en tiempo de compilación.

8. Se usará polimorfismo en lugar de reflexión, introspección sobre clases, siempre que sea posible.

9. Se debe poder diseñar las GUIs reusables en un editor gráfico de GUIs, tal y como se diseñan los GUIs usuales.

10. La implementación del proyecto se estructurará un forma de biblioteca de clases, usando el lenguaje de programación Java y la librería gráfica Swing. La biblioteca se empaquetará en un archivo de tipo jar.

11. Se debe elaborar un programa de prueba, según el siguiente esquema mínimo:

a) Los campos (variable) en una clase no tendrán acceso público, sino a través de métodos accesores, este es un requisito general para todo el proyecto.

b) Crear una clase, p. ej. A, con al menos cinco campos. Derivar de A dos clases, A1 y A2. A1 añade uno o dos campos más a la clase base. A2 añade al menos cinco.

c) Diseñar GUIs reusables para editar todos los campos de las clases A, A1 y A2. Diseñarlas sin botones, para que posteriormente sean envueltas en GUIs reusables con estructura mínima, y darles una distribución adecuada. Usar cierta variedad de componentes gráficos en las GUIs.

d) Crear tres clases B, C, D con al menos cinco campos cada una y de manera un objeto de clase A1, otro de clase A2 y otro de clase C estén agregados en uno de clase B, y uno de clase D esté agregado en uno de clase C.

e) Diseñar GUIs reusables para editar los campos de las clases B, C y D. El de B debe contener vista de árbol y botones, los otros no, darles una distribución adecuada. Usar cierta variedad de componentes gráficos en las GUIs. Mediante la comunicación de GUIs requerida en el punto 6, el cambio en un determinado componente gráfico de la GUI de D debe provocar un cambio en uno de B, p. ej. al deslizar un JSlider en la GUI de D que cambie un número en un JSpinner en la de B. 9

f) Crear un aplicación con interfaz gráfica en la que se vayan mostrando, y se puedan editar, las GUIs completas de A1, en una sola página (simple), de A2, con pestañas, y de B, con vista de árbol en la que se pueda acceder a todos las GUIs de sus objetos agregados y los agregados a sus agregados. Las GUIs deben ser ensambladas siguiendo el procedimiento a desarrollar comentado en el requisito 3(c). La edición debe ser real, o sea, con actualización de campos en los objetos al aceptar la edición el usuario, previa validación de la coherencia de los datos.

g) Las GUIs ensamblados deben tener un buen comportamiento frente a redimensionamientos realizados por el usuario.

12. El proyecto debe estar debidamente documentado usando un sistema estándar de documentación, como pueden ser Javadoc o Doxygen. 13. El software producido debe ser de código abierto, bajo la licencia GPL

3.2. Estimación de Costes y Tiempos

A continuación se detalla una posible estimación de los tiempos de la elaboración del proyecto.

Concepto	Porcentaje	Total Horas
memoria	25 %	135
proyecto	75 %	405
		540

Estimación del esfuerzo por Tareas

Tarea	Porcentaje	Total Horas
Análisis	15 %	60,75
Diseño	15 %	60,75
Codificación	35 %	141,75
Pruebas	35 %	141,75
		405

3.3. Casos de Uso

Existen dos roles principales que intervienen en el desarrollo de una aplicación: el desarrollador de aplicaciones y el diseñador de GUIs.

El desarrollador de aplicaciones interactuá con la librería pidiendo diálogos vacíos , inicializará los diálogos que le proporcione la librería, y añadirá un diálogo hijo a un diálogo padre. Además, pedirá diálogos creados por el diseñador de aplicaciones. Todas estas tareas las realizará en la vista de codificación del IDE.

El diseñador de GUIs interactuá con la vista diseño, la librería permite que sus distintos tipos de diálogos puedan ser usados en la vista de diseño del IDE. De esta forma, el diseñador de GUIs arrastrará uno de esos diálogos a la hoja al tapiz de diseño del IDE y podrá trabajar sobre él añadiendo componentes y dándole el aspecto que desee.

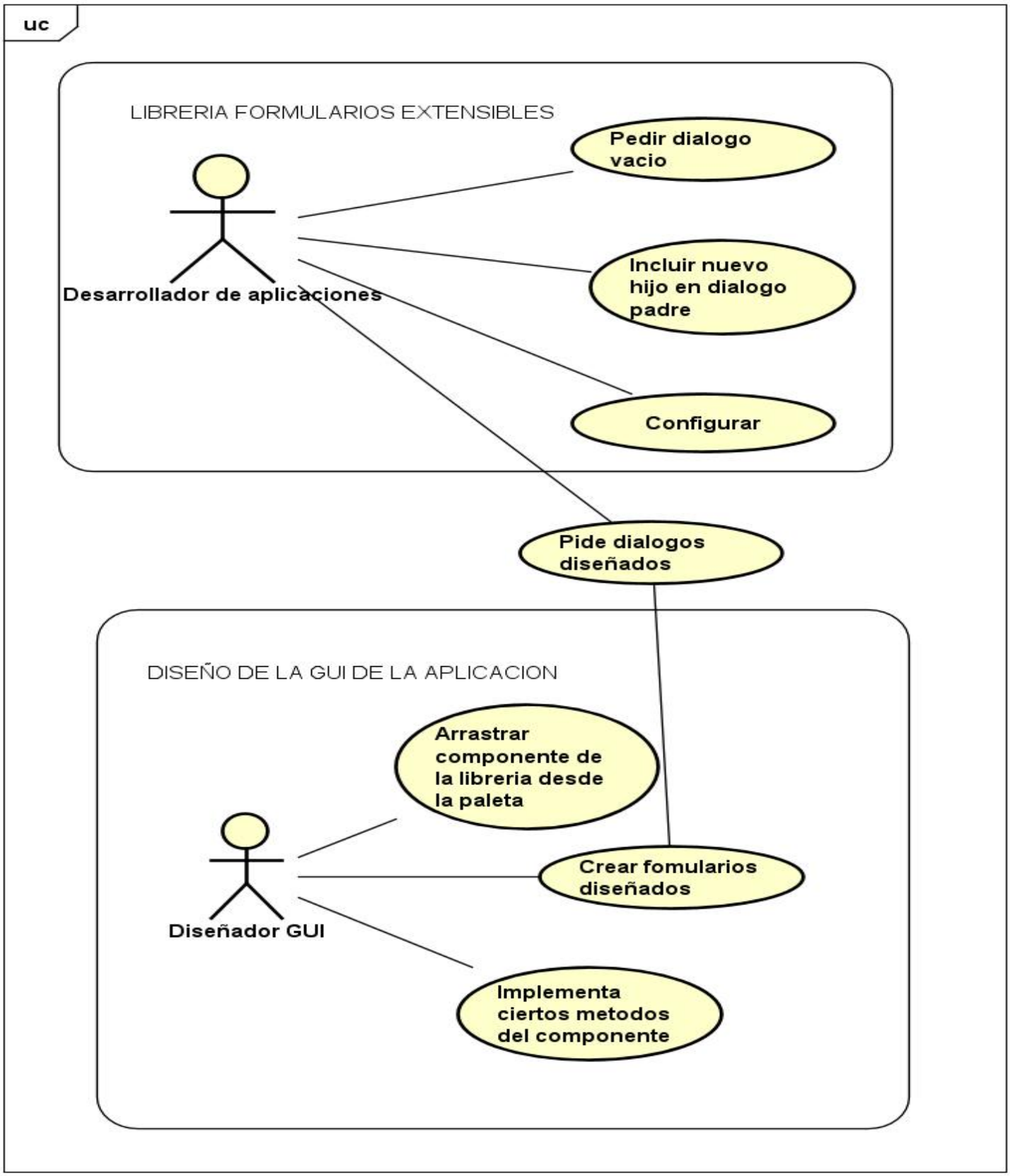

Figura 3.1: Esquema casos de Uso

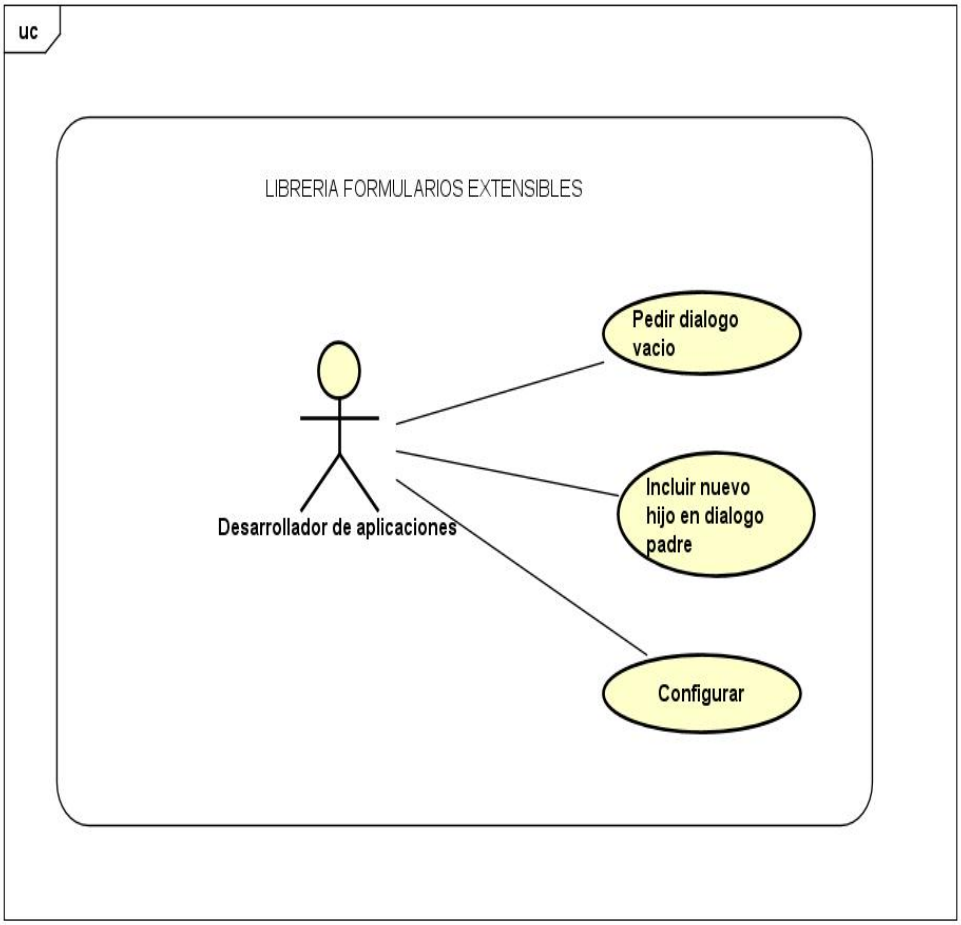

Figura 3.2: Librería Formulario Extensibles

3.3.1. Caso de Uso: LIBRERÍA FORMULARIOS EXTENSIBLES

Desde el punto de vista del desarrollador de aplicaciones, éste pedirá diálogos vacíos a la librería inicializará los diálogos que le proporcione la librería, y añadirá un diálogo hijo a un diálogo padre. Además, pedirá diálogos creados por el diseñador de aplicaciones. Todas estas tareas las realizará en la vista de codificación del IDE.

Evolución típica de los acontecimientos

1) Pide a la Librería un Formulario Vacío (Simple, Por Fichas, Árbol) => 2) Se devuelve una instancia del tipo de formulario

3) Pide añadir un hijo o una lista de hijos => 4) Se integran los Formularios hijos en el padre

5) Pide configurar el formulario => 6) Se ajustan los Formularios para visualizar

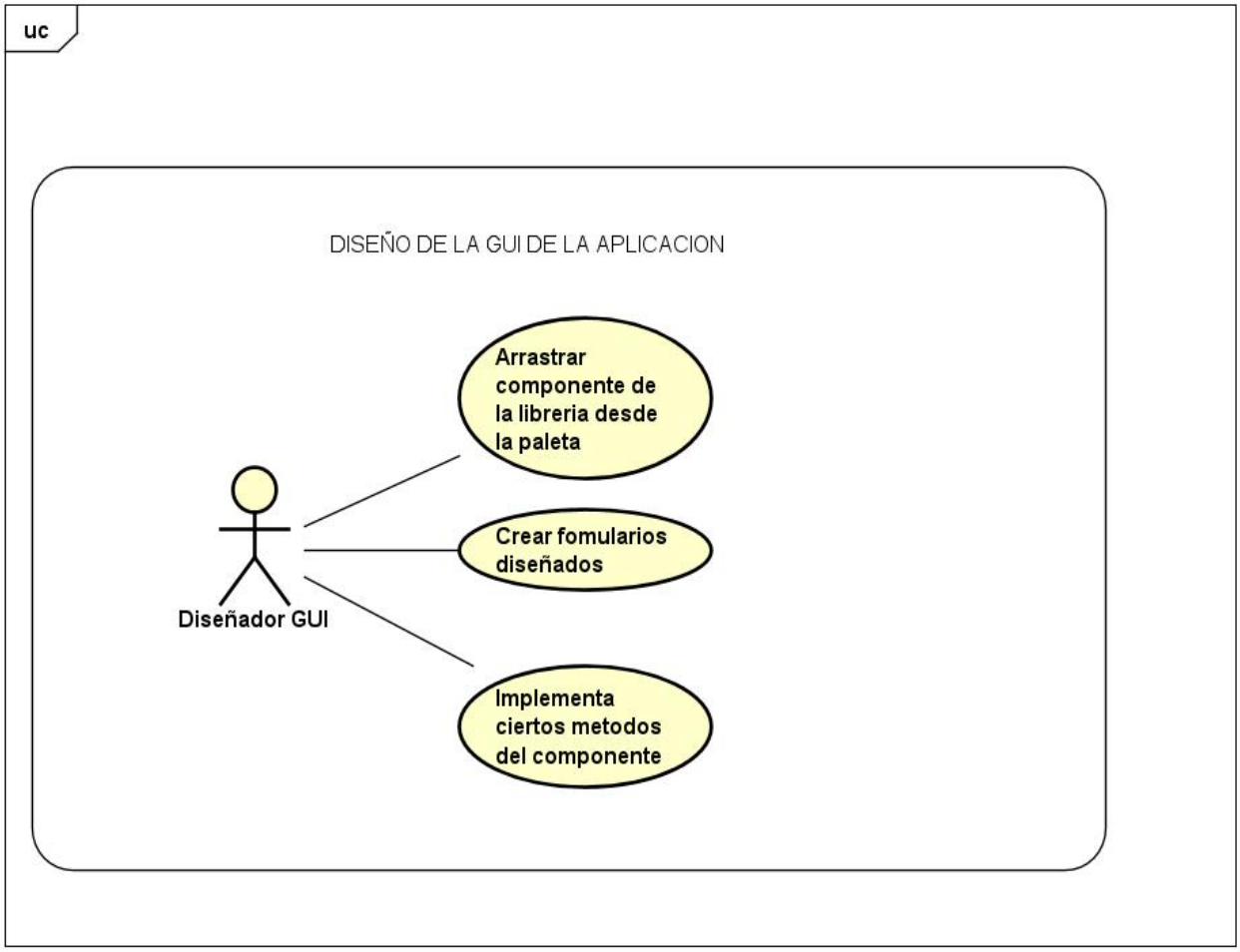

Figura 3.3: Diseño Gui

3.3.2. Caso de Uso: DISEÑO DE LA GUI DE LA APLICACIÓN

Desde el punto de vista del diseñador de GUIs, la librería debe permitir que sus distintos tipos de diálogos puedan ser usados en la vista de diseño del IDE. De esta forma, el diseñador de GUIs arrastrará uno de esos diálogos a la hoja al tapiz de diseño del IDE y podrá trabajar sobre él añadiendo componentes y dándole el aspecto que desee.

Evolución típica de los acontecimientos

1)En la vista diseño del IDE arrastra desde la paleta un icono de Formulario (Simple, Por Fichas, Árbol) => 2)La librería permite utilizar los Formularios Extensibles en vista diseño

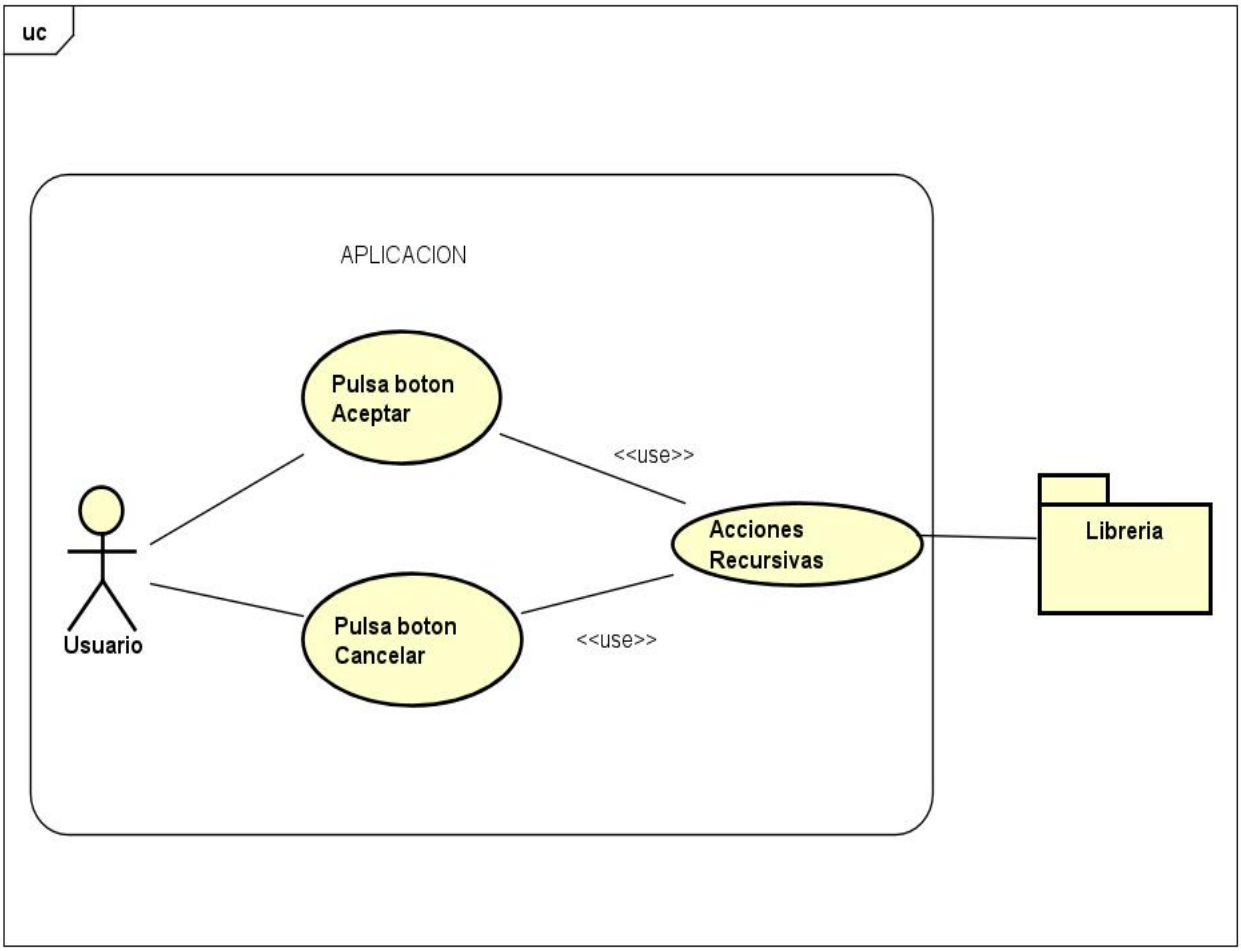

Figura 3.4: Acciones del Usuario

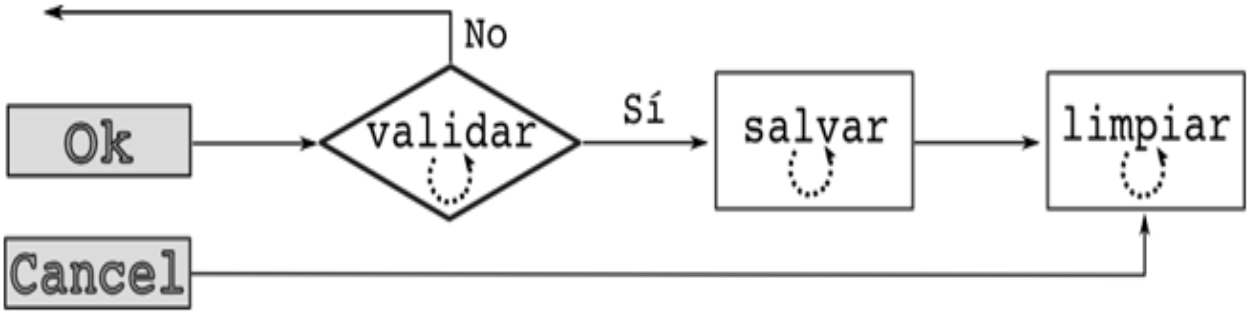

Figura 3.5: Esquema Botones

3.3.3. Caso de Uso: ACCIONES DEL USUARIO

Desde el punto de vista del Usuario, la librería inicia las acciones recursivas de validación , guardado o limpieza de los datos de los Formularios cuando se ha pulsado el Botón Aceptar o Botón Cancelar.

Evolución típica de los acontecimientos Botón Aceptar con Validación correcta

1)El usuario introduce cambios => 2)El usuario pulsa Botón Aceptar para confirmar los cambios realizados

3)Se realizan las tareas recursivas de Validación => 4)Se realizan las tareas recursivas de Guardado

5)Se realizan las tareas recursivas de Limpieza => 6)Se presenta mensaje de Operación Realizada.

Evolución típica de los acontecimientos Botón Aceptar con Validación incorrecta

1)El usuario introduce cambios => 2)El usuario pulsa Botón Aceptar para confirmar los cambios realizados

3)Se realizan las tareas recursivas de Validación => 4)Se presenta mensaje de Datos Incorrectos

Evolución típica de los acontecimientos Botón Cancelar

1)El usuario introduce cambios => 2)El usuario pulsa Botón Cancelar para rechazar los cambios realizados

3)Se realizan las tareas recursivas de Limpieza => 4)Se presenta mensaje de Operación Cancelada.

3.4. Diagramas UML del Sistema

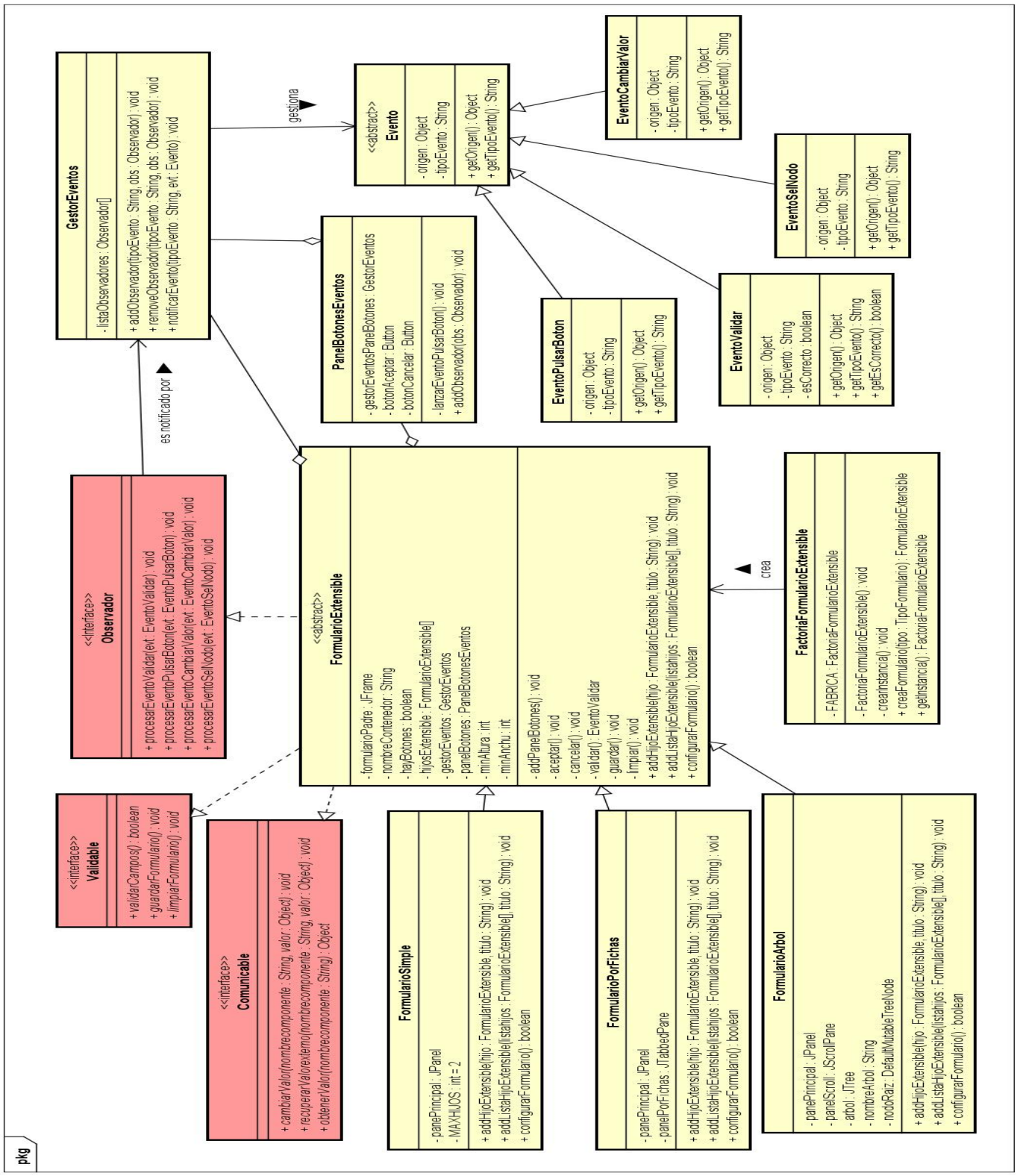

Figura 3.6: Diagrama Clases

3.4.1. Diagramas de Clases

La jerarquía de clases permite 3 tipos de Formulario: Simple, por Fichas y Árbol.

Lista de las clases que se visualizan:

- FormularioExtensible: Clase abstracta que es padre de los tipos de formulario de la jerarquía.
- FormularioSimple: Formulario básico de la jerarquía.
- FormularioPorFichas: Formulario con navegación por pestañas.
- FormularioArbol: Formulario con navegación con un esquema de árbol.
- FactoriaFormulariosExtensible: Permite la creación de formularios por tipo.
- PanelBotonesEventos: Clase que define un Panel que contiene los Botones Aceptar y Cancelar.
- GestorEventos: Clase que notifica Eventos a un Objeto Observador.
- Evento: Clase abstracta que es padre de los tipos de Evento de la jerarquía
- EventoCambioValor: Clase para representar un evento de cambio de valor en un Objeto.
- EventoPulsarBoton: Clase para representar un evento de botón pulsado.
- EventoSelNodo: Clase para representar un evento de selección de Nodo en un Árbol.
- EventoValidar: Clase para representar un evento de Validación.

Lista de las interfaces que se visualizan:

- Validable: Interface que define los métodos para implementar la validación de formularios.
- Comunicable: Interface que define los métodos para implementar la comunicación entre componentes de formularios.
- Observador: Interface que define los métodos que debe implementar un Objeto que observa eventos.

3.4.2. Diagramas de los métodos Recursivos para Validación, Guardado y Limpieza.

3.4.2.1. Botón Aceptar

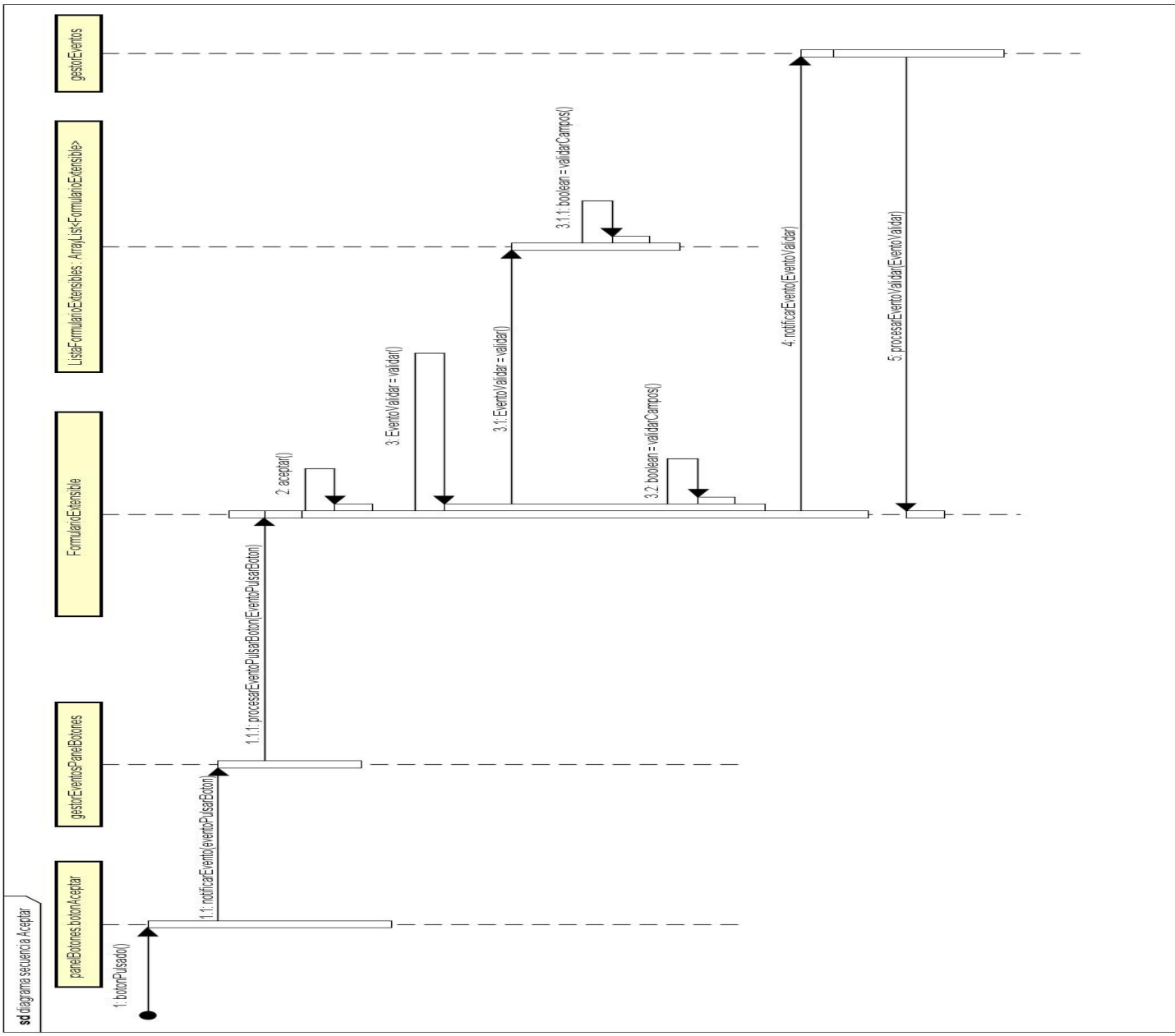

Figura 3.7: Diagrama Botón Aceptar

En este diagrama de secuencia se muestra el funcionamiento del Botón Aceptar. Se lanza un evento pulsar botón, que se notifica a través del gestor de Eventos. Esto provoca que se ejecute el método aceptar. Seguidamente se realiza la validación recursiva de los Formularios que puede lanzar un evento validar Correcto o Incorrecto.

3.4.2.2. Evento Validar Correcto

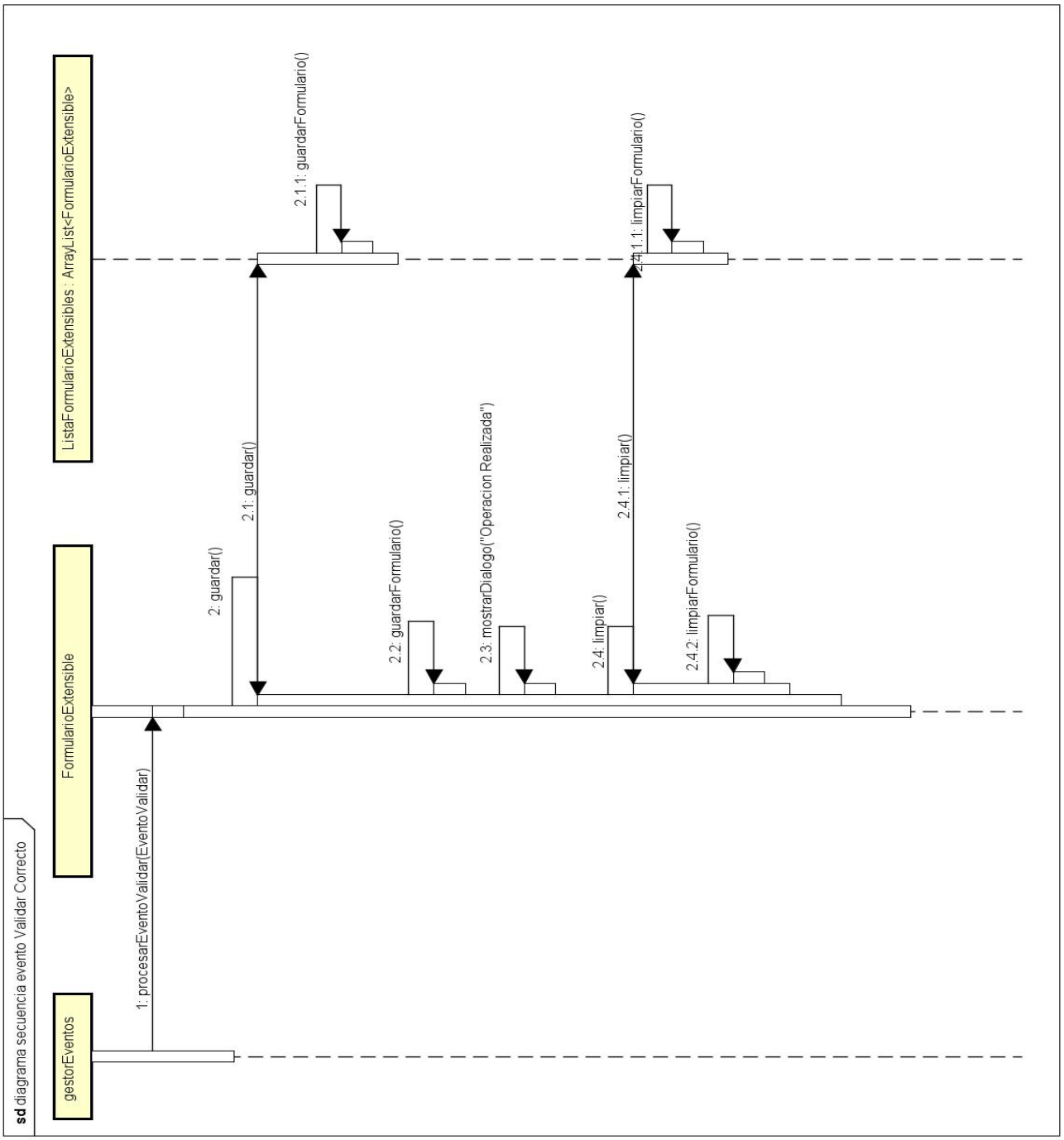

Figura 3.8: Evento Validar Correcto

En este diagrama de secuencia se muestra el funcionamiento del Evento Validar Correcto que se notifica a través del gestor de Eventos. Esto provoca que se realice las operaciones recursivas de guardado y seguidamente la limpieza de los Formularios. Se muestra un dialogo con un mensaje de Operación Realizada.

3.4.2.3. Evento Validar Incorrecto

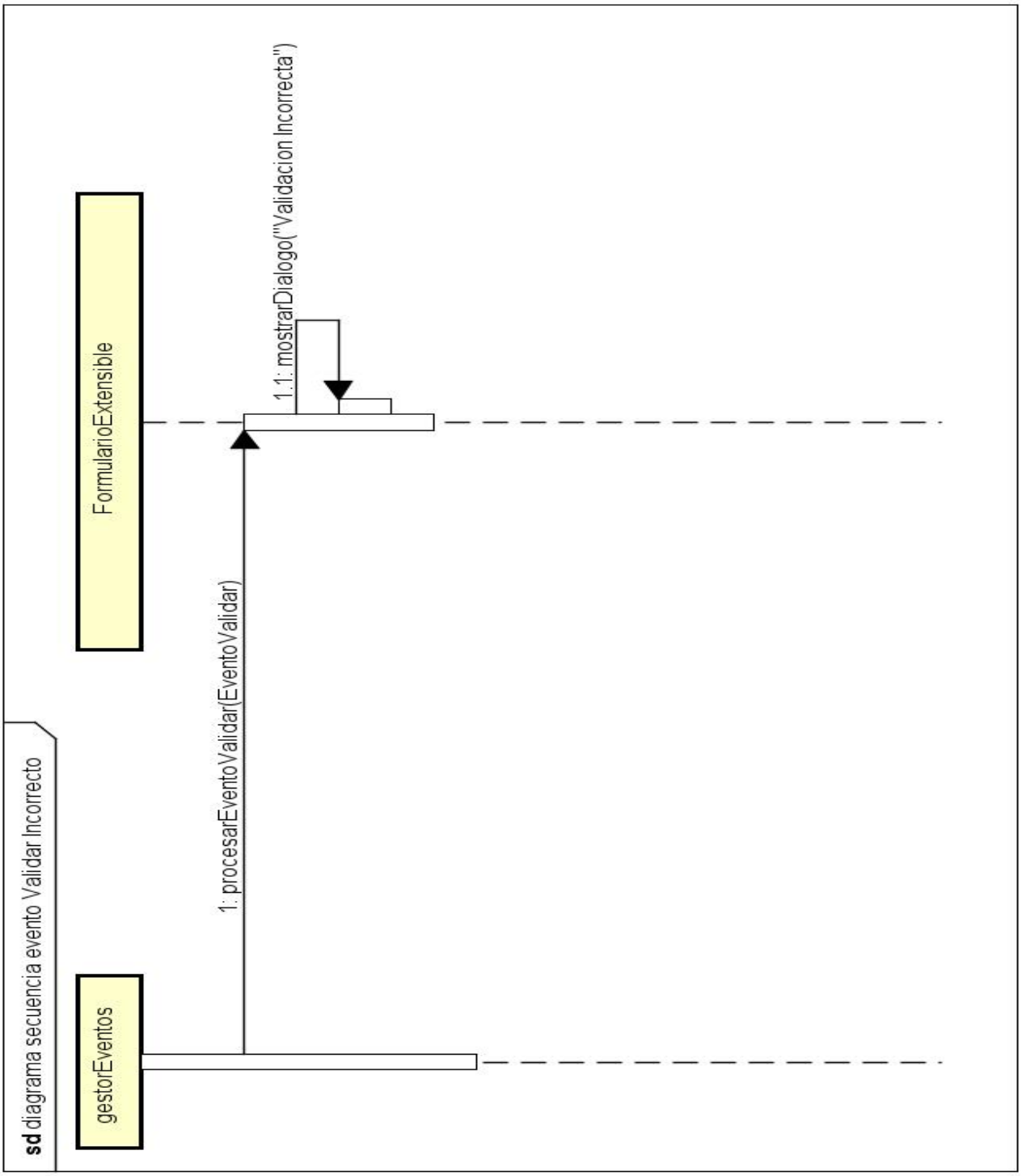

Figura 3.9: Evento Validar Incorrecto

En este diagrama de secuencia se muestra el funcionamiento del Evento Validar Incorrecto que se notifica a través del gestor de Eventos. Esto provoca que se muestre un dialogo con un mensaje de Validación Incorrecta.

3.4.2.4. Botón Cancelar

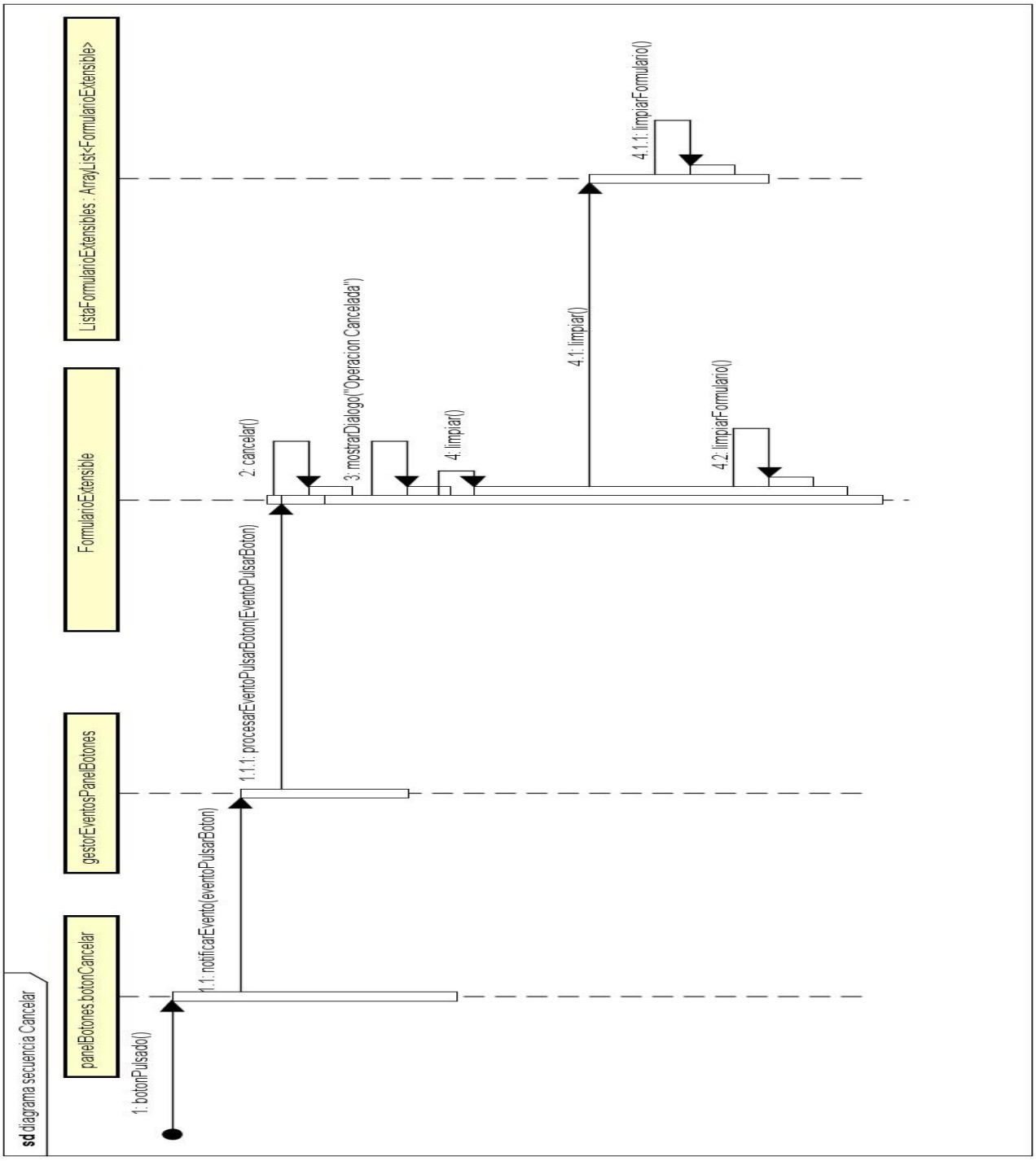

Figura 3.10: Botón Cancelar

En este diagrama de secuencia se muestra el funcionamiento del Botón Cancelar. Se lanza un evento pulsar botón, que se notifica a través del gestor de Eventos. Esto provoca que se ejecute el método cancelar. Seguidamente se realiza la operación recursiva de limpieza de los Formularios. Se muestra un dialogo con un mensaje de Operación Cancelada.

Capítulo 4

Implementación

Este capitulo aborda la fase de implementación y construcción de la Librería de GUI reusables.

Primero se hace una descripción general del código fuente de la librería explicando algunos aspectos generales de la implementación.

Seguidamente se comentaran las diferentes Clases que forman parte de la Librería, agrupadas por varios conceptos que son:

1) Interfaces

Comunicable

Validable

Observador

2) Manejo de Eventos

Evento

EventoPulsarBoton

EventoValidar

EventoCambiarValor

EventoSelNodo

GestorEventos

3) Formularios Extensibles

Formulario

FormularioExtensible

FormularioSimple

FormularioPorFichas

FormularioArbol

FactoriaFormularioExtensible

PanelBotonesEventos

Se explicara detalladamente la función de cada una de estas Clases junto con los métodos principales que contienen.

4.1. Aspectos generales del código fuente de la librería

4.2. Interfaces de la Librería

4.2.1. Comunicable

Esta clase es una interfaz que es implementada por la clase abstracta FormularioExtensible que es padre de los 3 tipos de Formularios (Simple, Por Fichas y Árbol) los cuales heredaran las propiedades y métodos de esta interfaz.

Define los métodos que deben declaran los formularios que quieran usar las comunicación por mensajes de la librería.

4.2.1.1. recuperarValorExterno

Este método permite cambiar valor a un componente de un Formulario Extensible desde otro Formulario Externo.

Toma como parámetros un String que es el identificador y un campo genérico Object. No devuelve ningún valor.

4.2.1.2. cambiarValor

Este método permite cambiar el valor a un componente de un Formulario Extensible.

Toma como parámetros un String que es el identificador y un campo genérico Object. No devuelve ningún valor.

4.2.1.3. obtenerValor

Este método permite recuperar el valor de un componente de un Formulario Extensible.

Toma como parámetros un String que es el identificador. Devuelve un campo genérico Object.

4.2.2. Validable

Esta clase es una interfaz que es implementada por la clase abstracta FormularioExtensible que es padre de los 3 tipos de Formularios (Simple, Por Fichas y Árbol) los cuales heredaran las propiedades y métodos de esta interfaz.

Define los métodos recursivos de validación, guardado y limpieza de datos que deben declaran los formularios reusables. Estos métodos se ejecutan cuando se pulsan los botones Aceptar o Cancelar.

4.2.2.1. validarCampos

Este método permite hacer las validaciones de un Formulario Extensible debe ser implementado obligatoriamente aunque su implementación puede ser trivial.

No toma ningún valor como parámetro. Debe devolver un booleano con valor cierto en caso de validación correcta.

4.2.2.2. guardarFormulario

Este método permite hacer las tareas de guardado de un Formulario Extensible, debe ser implementado obligatoriamente aunque su implementación puede ser trivial.

No toma ningún valor como parámetro. No devuelve ningún valor.

4.2.2.3. limpiarFormulario

Este método permite hacer las tareas de limpieza de un Formulario Extensible, debe ser implementado obligatoriamente aunque su implementación puede ser trivial.

No toma ningún valor como parámetro. No devuelve ningún valor.

4.2.3. Observador

Esta clase es una interfaz que es implementada por la clase abstracta FormularioExtensible que es padre de los 3 tipos de Formularios (Simple, Por Fichas y Árbol) los cuales heredaran las propiedades y métodos de esta interfaz.

Define los métodos que deben declaran los formularios que quieran usar la recepción y la gestión de Eventos.

4.2.3.1. procesarEventoValidar

Este método permite definir las tareas que hace un Formulario Extensible cuando recibe un Evento de tipo Validar. Normalmente esta asociado a la ejecución de la validación de un formulario al pulsar el botón Aceptar.

Toma como parámetro un EventoValidar. No devuelve ningún valor.

4.2.3.2. procesarEventoPulsarBoton

Este método permite definir las tareas que hace un Formulario Extensible cuando recibe un Evento de tipo Pulsar Botón. Normalmente esta asociado a la a la pulsación del botón Aceptar/Cancelar.

Toma como parámetro un EventoPulsarBoton. No devuelve ningún valor.

4.2.3.3. procesarEventoCambiarValor

Este método permite definir las tareas que hace un Formulario Extensible cuando recibe un Evento de tipo Cambiar Valor. Normalmente esta asociado al cambo de valor de un componente que se quiere notificar a otro componente.

Toma como parámetro un EventoCambiarValor. No devuelve ningún valor.

4.2.3.4. procesarEventoSelNodo

Este método permite definir las tareas que hace un Formulario Extensible cuando recibe un Evento de tipo Selección Nodo. Normalmente esta asociado a una navegación con vista de Árbol.

Toma como parámetro un EventoSelNodo. No devuelve ningún valor.

4.3. Clases para manejo de Eventos

4.3.1. Evento

Esta clase abstracta define la propiedades básicas de un evento. Las propiedades que define son tipoEvento de tipo String para diferenciar el tipo de evento y la propiedad Origen de tipo Object que sirve para guardar el objeto origen del evento.

4.3.2. EventoPulsarBoton

Esta clase derivada de Evento define la propiedades básicas de un evento de pulsación de Botón. Las propiedades que hereda son tipoEvento de tipo String para diferenciar el tipo de evento y la propiedad Origen de tipo Object que sirve para guardar el objeto origen del evento.

4.3.3. EventoValidar

Esta clase derivada de Evento define la propiedades de un evento de selección de nodo en una vista de navegación por Árbol. Las propiedades que hereda son tipoEvento de tipo String para diferenciar el tipo de evento y la propiedad Origen de tipo Object que sirve para guardar el objeto origen del evento. Define la propiedad Correcto de tipo bolean que permite almacenar si la validación es correcta o no.

4.3.4. EventoCambiarValor

Esta clase derivada de Evento define la propiedades de un evento de cambio de valor en un Objeto. Las propiedades que hereda son tipoEvento de tipo String para diferenciar el tipo de evento y la propiedad Origen de tipo Object que sirve para guardar el objeto origen del evento.

4.3.5. EventoSelNodo

Esta clase derivada de Evento define la propiedades de un evento de selección de nodo en una vista de navegación por Árbol. Las propiedades que hereda son tipoEvento de tipo String para diferenciar el tipo de evento y la propiedad Origen de tipo Object que sirve para guardar el objeto origen del evento.

4.3.6. GestorEventos

Esta clase gestiona una lista de objetos que implementan la clase Observador a los que notifica Eventos. La responsabilidad de esta clase es la de servir de mediador entre los Eventos y los Observadores.

4.3.6.1. addObservador

Este método permite agregar un nuevo Observador a la lista de Observadores del Gestor de Eventos. El Observador se agrega para la notificación de un tipo de Evento.

Toma como parámetros un String para el tipo de Evento y un Observador a agregar. No devuelve ningún valor.

4.3.6.2. removeObservador

Este método permite quitar un Observador de la lista de Observadores del Gestor de Eventos. El Observador se puede quitar de todos los tipo de notificación o de un tipo de Evento.

Si se quiere quitar de la notificación de un tipo de de Evento toma como parámetros un String para el tipo de Evento y un Observador. No devuelve ningún valor.

Para quitar de la notificación de cualquier tipo de de Evento toma como parámetro un Observador. No devuelve ningún valor.

4.3.6.3. notificarEvento

Este método permite notificar un Evento de un tipo a la lista de Observadores del Gestor de Eventos. A cada Observador de ese tipo de Evento se le manda un mensaje de procesarEvento enviando como parámetro el Evento recibido.

Toma como parámetros un String para el tipo de Evento y un Evento a notificar. No devuelve ningún valor.

Listado de código 4.1: notificarEvento

```
public void notificarEvento(String TipoEvento, Evento evt) {
        ArrayList<Observador> listaObs = Observadores.get(
           TipoEvento);
        for (Observador obs : listaObs) {

            switch (TipoEvento) {
                case "Validar":
                    obs.procesarEventoValidar((EventoValidar) evt
                       );
                    break;
                case "PulsarBoton":
                    obs.procesarEventoPulsarBoton((
                       EventoPulsarBoton) evt);
                    break;
                case "CambiarValor":
                    obs.procesarEventoCambiarValor((
                       EventoCambiarValor) evt);
                    break;
                case "SelNodo":
                    obs.procesarEventoSelNodo((EventoSelNodo) evt
                       );
                    break;
                default:
                    throw new UnsupportedOperationException("Not␣
                       supported␣yet.");

            }

        }
    }
```

4.4. Clases de los Formularios Extensibles

4.4.1. Formulario

Esta clase abstracta es la base de los Formularios Extensibles y define la propiedades básicas de un Formulario. Las propiedades que define son formularioPadre de tipo JFrame para establecer el Formulario padre del Formulario actual.

4.4.2. FormularioExtensible

4.4.3. FormularioSimple

Esta clase representa el Formulario más sencillo de la jerarquia de Formularios Extensibles. Solo puede tener un máximo de 2 hijos. Implementa las interfaces Comunicable, Validable, Observador. Las propiedades que define son panelPrincipal de tipo JPanel que es el contenedor principal del Formulario y MAXHIJOS que es una constante con valor 2.

4.4.3.1. addHijoExtensible

Este método permite agregar un hijo de tipo FormularioExtensible al formulario actual. Se guarda la altura, anchura, titulo y se asigna como padre el formulario actual sino se supera el máximo de hijos permitidos. Cuando se supera el máximo de hijos lanza una Exception.

Toma como parámetros un FormularioExtensible y un String para el título del Formulario. No devuelve ningún valor.

4.4.3.2. addListaHijosExtensibles

Este método permite agregar una lista de hijos de tipo FormularioExtensible al formulario actual. Para cada FormularioExtensible de la lista se guarda la altura, anchura, titulo y se asigna como padre el formulario actual sino se supera el máximo de hijos permitidos. Cuando se supera el máximo de hijos lanza una Exception.

Toma como parámetros una lista de FormularioExtensible y un String para el título del Formulario. No devuelve ningún valor.

4.4.3.3. configurarFormulario

Este método permite configurar la visualizacion del formulario actual. Llama al metodo configurar-Formulario de la clase padre FormularioExtensible.

No toma ningún valor como parámetro. Devuelve un booleno para indicar si todo ha ido correctamente.

4.4.4. FormularioPorFichas

Esta clase representa el Formulario que permite una navegación de los Formularios Extensibles por pestañas. Implementa las interfaces Comunicable, Validable, Observador. Las propiedades que define son panelPrincipal de tipo JPanel que es el contenedor principal del Formulario y panelPorFichas de tipo JTabbedPane que representa la navegación por pestañas.

4.4.4.1. addHijoExtensible

Este método permite agregar un hijo de tipo FormularioExtensible al formulario actual. Se guarda la altura, anchura , titulo, se asigna como padre el formulario actual y se agrega una nueva pestaña al panelPorFichas con el Formulario hijo.

Toma como parámetros un FormularioExtensible y un String para el título del Formulario. No devuelve ningún valor.

4.4.4.2. addListaHijosExtensibles

Este método permite agregar una lista de hijos de tipo FormularioExtensible al formulario actual. Para cada FormularioExtensible de la lista se guarda la altura, anchura , titulo , se asigna como padre el formulario actual y se agrega una nueva pestaña al panelPorFichas con el Formulario hijo

Toma como parámetros una lista de FormularioExtensible y un String para el título del Formulario. No devuelve ningún valor.

4.4.4.3. configurarFormulario

Este método permite configurar la visualizacion del formulario actual. Llama al metodo configurarFormulario de la clase padre FormularioExtensible.

No toma ningún valor como parámetro. Devuelve un booleno para indicar si todo ha ido correctamente.

4.4.5. FormularioArbol

Esta clase representa el Formulario que permite una navegación de los Formularios Extensibles por pestañas. Implementa las interfaces Comunicable, Validable, Observador. Las propiedades que define

son panelPrincipal de tipo JPanel que es el contenedor principal del Formulario y panelPorFichas de tipo JTabbedPane que representa la navegación por pestañas.

4.4.5.1. addHijoExtensible

Este método permite agregar un hijo de tipo FormularioExtensible al formulario actual. Se guarda la altura, anchura , titulo, se asigna como padre el formulario actual y se agrega una nueva pestaña al panelPorFichas con el Formulario hijo.

Toma como parámetros un FormularioExtensible y un String para el título del Formulario. No devuelve ningún valor.

4.4.5.2. addListaHijosExtensibles

Este método permite agregar una lista de hijos de tipo FormularioExtensible al formulario actual. Para cada FormularioExtensible de la lista se guarda la altura, anchura , titulo , se asigna como padre el formulario actual y se agrega una nueva pestaña al panelPorFichas con el Formulario hijo

Toma como parámetros una lista de FormularioExtensible y un String para el título del Formulario. No devuelve ningún valor.

4.4.5.3. configurarFormulario

Este método permite configurar la visualizacion del formulario actual. Llama al metodo configurarFormulario de la clase padre FormularioExtensible.

No toma ningún valor como parámetro. Devuelve un booleno para indicar si todo ha ido correctamente.

4.4.6. FactoriaFormularioExtensible

Esta clase representa la factoria de Formularios Extensibles y usa el patrón Singleton que solo permite una instancia única de la clase y tiene un constructor privado . Las propiedades que define son FABRICA de tipo FabricaFormularioExtensible que es la instancia única de la clase.

4.4.6.1. crearFormulario

Este método permite crear un FormularioExtensible de un tipo.

Toma como parámetro un TipoFormulario(SIMPLE, PORFICHAS, ARBOL). Devuelve un FormularioExtensible del tipo de Formulario solicitado.

4.4.6.2. crearInstancia

Este método privado permite crear una instancia de la Factoria de Formularios Extensibles si no existía previamente.

No toma ningún valor como parámetro. No devuelve ningún valor.

4.4.6.3. getInstancia

Este método permite crear una instancia de la Factoria de Formularios Extensibles si no existía previamente.

No toma ningún valor como parámetro. Devuelve la instancia unica FABRICA de tipo FabricaFormularioExtensible.

4.4.7. PanelBotonesEventos

Esta clase representa un panel con los botones Aceptar y Cancelar, solo se muestra si se ha definido el Formulario Extensible con botones. Las propiedades que define son botonAceptar de tipo JButton que representa el botón que confirma las modificaciones, botonCancelar de tipo JButon que representa el botón que anula las modificaciones. Ademas gestorEventosPanelBotones de tipo GestorEventos que permite agregar un Observador al que notificar eventos de pulsación de botón.

4.4.7.1. addObservador

Este método permite agregar un Observador al que notificar Eventos de Pulsación de botón.

Toma como parámetro un Observador al que se va a notificar los Eventos de pulsación de botón. No devuelve ningún valor.

4.4.7.2. lanzarEventoPulsarBoton

Este método permite notificar eventos de pulsación de botón a los Observadores agregados al gestorEventosPanelBotones.

Toma como parámetro un EventoPulsarBoton que se va a notificar los Observadores de pulsación de botón. No devuelve ningún valor.

Capítulo 5

Experimentación

Este capitulo aborda la fase de implementación y construcción de la Librería de GUI reusables.

Primero se hace una descripción general del código fuente de la librería explicando algunos detalles generales de la implementación.

Seguidamente se comentaran las Clases que forman parte de la Librería, que son:

1. Comunicable
2. Validable
3. Formulario
4. FormularioExtensible
5. FormularioSimple
6. FormularioPorFichas
7. FormularioArbol
8. FactoriaFormularios
9. ListaObservadoresEventos
10. PanelBotones

Se explicara detalladamente la función de cada una de estas Clases junto con los métodos principales que contienen.

5.1. Descripción del código fuente de la aplicación de prueba

5.2. Funcionamiento de la aplicación de prueba

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

1. Introducción general y objetivos
2. <añadir los demás capítulos>
3. Conclusiones y trabajos futuros

<Comentar los capítulos>

Capítulo 6

Conclusiones y trabajos futuros

6.1. Conclusiones

<....>

6.2. Trabajos futuros

<....>

Capítulo 7

Ejemplos Referencias

<Intro ... >

7.1. Ejemplos Referencias

<Ejemplo biblio Eckel (2003); Gamma et al. (2005)>

<Ejemplo referencia a sección 1.4 y a subsección de otro capítulo, que en este caso es un anexo A.1.1. **Para que funcionen correctamente las referencias a otros capítulos al exportar a pdf, debe estar abierto el archivo maestro, abrir/editar el archivo a exportar desde él y exportar desde el maestro**.>

<Ejemplo referencia a figura 7.1>

<Ejemplo referencia a tabla 7.1>

<Ejemplo de nomenclatura (POO)>

<Ejemplo de listado de código, ver listado 7.1, lo mejor para utilizarlo es copiar el recuadro y cambiar el código contenido, así conservaremos las opciones que se han establecido, como: mostrar números de línea, quebrar líneas largas. etc. Para ver las opciones y poder modificarlas click derecho sobre el listado y elegir Configuración. Para cambiar el listado contenido copiar del IDE el trozo de código que se desee y pegarlo dentro del listado con Ctrl+May+v, en el menú de Lyx Editar->Pegado especial->Texto simple.>

1	2	3	4
aaaaaa	aaaaaa	aaaaaa	aaaaaa
aaaaaa	aaaaaa	aaaaaa	aaaaaa

Cuadro 7.1: Ejemplo de tabla

Figura 7.1: Ejemplo de figura con un pie largo para mostrar el uso del ítem del menú de Lyx Insertar->Título breve. La utilidad del título breve se aprecia en la lista de figuras, es ahí donde aparece, si no, aparecería todo el pie en la lista, también es conveniente para títulos de capítulos secciones y demás largo que ocupen más de una línea en la lista de índice correspondiente.(Para insertar salto de línea en pie o en enumeración Ctrl-Enter)
La figura o tabla hay que insertarla dentro de un flotante.
Siempre hay que Insertar->Etiqueta de la figura (fig:Ejemplo-de-figura), tabla etc, para referenciarla desde el texto.

Listado de código 7.1: Código de edit

```
1     public final boolean edit() {
2
3         ArrayList<String> campos = cargaCampos();
4
5         Scanner in = new Scanner( System.in );
6
7     /* La variable resp sera true si el usuario acepta la edicion
8      */
9
10    boolean resp = false;
11        int numero_total_opciones = campos.size() +
              opciones_no_campos;
12        System.out.println("\n" + toString()); // muestra el
              registro
13
14        editLoop: while (true) {
15            editMensaje();
16            System.out.println("Seleccione un numero (1-" +
                  numero_total_opciones + "): ");
17            int opcion;
18            if ( in.hasNextInt() ) {
19                opcion = in.nextInt();
20                in.nextLine();
21            } else {
22                System.out.println("\nOPCION NO VALIDA. POR FAVOR
                      , INTRODUZCA UN NUMERO.");
23                in.nextLine();
24                continue;
25            }
26            campos = procesaOpcion(campos, opcion, in);
27            if(salir) {
28                if(aceptar)
29                    resp = true;
30                salir = false;
31                aceptar = false;
32                break editLoop;
33            }
34
35        }
36        return resp;
37    }
```

Bibliografía

Eckel, B. (2003). *Thinking in Patterns: Problem-Solving Techniques using Java*. http://mindview.net/Books/TIPatterns/.

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

Anexo A

<Título Anexo A>

<...>

A.1. <Primera sección anexo>

A.1.1. <Primera subsección anexo>