UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingenieria Informatica

Implementación del Patrón de Diseño de Reusabilidad de GUIs.

Realizado por: ALBERTO EIRIZ LOPEZ

Dirigido por: JOSÉ MANUEL CUADRA TRONCOSO

Curso: 2021-2022: Convocatoria Extraordinaria

Implementación del Patrón de Diseño de Reusabilidad de GUIs.

Proyecto de Fin de Grado de modalidad oferta general

Realizado por: Alberto Eiriz Lopez

Dirigido por: José Manuel Cuadra Troncoso

Tribunal calificador

Presidente: D/Dª.

Secretario: D/Dª.

Vocal: D/Dª.

Fecha de lectura y defensa: 12 de Diciembre de 2022

Calificación:

Agradecimientos

Me gustaria dedicar este trabajo a dos personas en especial, a mi padre, que siempre quiso disfrutar de este momento y nunca pudo, y a mi muy querida suegra, quien un dia me recordo todo lo que podia llegar a hacer y aun no habia hecho.

Quiero dar las gracias tambien a mi mujer y a mi hija por haberme dado durante todos estos años el tiempo y las ganas que necesitaba para poder llegar hasta aqui, este proyecto es vuestro tambien.

Y finalmente quiero dedicar este trabajo a mi madre, por haberlo aguantado todo hasta ahora.

Os lo agradezco de todo corazon.

Resumen

La reutilizacion de codigo es a dia de hoy uno de los factores fundamentales a tener en cuenta a la hora de desarrollar sistemas informaticos. Su pertinencia supone una notable economia de tiempo, esfuerzo y costes de ingenieria, y promueve una mayor facilidad de mantenimiento, extensibilidad y escalabilidad.

En el departamento de Inteligencia Artificial de la Escuela Tecnica Superior de Ingenieria Informatica de la UNED, se encontraron con el problema de la reutilizacion de codigo a la hora de diseñar las interfaces graficas de usuario (GUIs: Graphical User Interfaces) para CyBerSim, un software de simulacion y control de conductas en robots autonomos que estaban desarrollando. Llegaron a la conclusion de que era preciso diseñar una biblioteca de clases que les facilitara la funcionalidad necesaria para poder integrar GUIs que habian elaborado con anterioridad en nuevas interfaces que simplemente ampliaban la edicion de las anteriores en unos pocos parametros. De esta forma evitaban duplicar codigo y componentes graficos, y posibilitaban una rapida generacion de las interfaces graficas en posteriores desarrollos de la aplicacion. Diseñaron una biblioteca en C++, el lenguaje de programacion en el que se estaba escribiendo CyBerSym, usando la libreria grafica Qt-4 y utilizandola y aplicandola a traves del software de diseño de GUIs QtDesigner. A raiz de este desarrollo se extrajeron las pautas esenciales para articular un patron de diseño de reusabilidad de GUIs mediante el cual poder formalizar la solucion aplicada al problema que ellos habian encontrado, de forma que otros pudiesen utilizarla en sus diseños.

El objetivo de este trabajo es realizar una descripcion de dicho patron y ofrecer una implementacion del mismo en forma de libreria de clases, asi como el desarrollo de un modulo de extension para integrar dicha libreria en un entorno de desarrollo integrado (IDE: Integrated Development Environment). La implementacion se realizara utilizando el lenguaje de programacion Java y su libreria grafica Swing. El IDE que se utilizara sera NetBeans y su tecnologia de JavaBeans, de tal forma que podamos integrar un modulo con la libreria en el entorno de desarrollo para poder utilizar sus clases como un elemento mas de diseño. Finalmente se elaborara un programa de prueba mediante el cual mostrar la versatilidad de las GUIs reusables. Para ello se implementara una pequeña aplicacion para la edicion de un Catalogo de Instrumentos de Medicion en la que se pueda comprobar el funcionamiento de los diferentes mecanismos de reusabilidad articulados a traves de la libreria.

Palabras clave Reutilizacion|GUI |Patron|Biblioteca|Modulo|IDE

Abstract

Code reusability is nowadays one of the fundamental factors to consider when developing computer systems. It´s pertinence is a significant economy of time, effort and engineering costs, and boosts greater ease of maintenance, extensibility and scalability.

In the department of Artificial Intelligence of the Higher Technical School of Computer Engineering of the UNED, they found the problem of code reusability when designing the graphical user interfaces (GUIs) for CyBerSym, a simulation and behavior control software in autonomous robots that they were developing. They concluded that it was required to design a class library that will provide them with the necessary functionality to be able to integrate GUIs that they had already elaborated before in new interfaces that simply expanded the edition of the previous one in a few parameters. In that way they avoided duplicating code and graphic components, and made a faster generation of graphic interfaces on further developments of the application. They designed a library in C++, the programming language in wich CyBerSym was being written, using the Qt-4 graphic library and using it and applying it through the GUIs design software QtDesigner. As a result of this development, the essential guidelines were extracted to articulate a GUIs reusability Design Pattern, whereby they could formalize the solution applied to the problem they had found in a way that others could use it in their designs.

The objective of this work is to make a description of this pattern and offer an implementation of it in the form of class library, as well as de development of an extension module to integrate this library into an integrated development environment (IDE). The implementation will be carried out using the Java programming language and its graphical library Swing. The IDE that will be used will be NetBeans and it´s JavaBeans technology, so that we can integrate a module with the library into the development environment to be able to use its clasess as one more element of design. Finally, a test program will be developed through which to show the versatility of reusable GUIs. To do this, a small application for the edition of a Catalog of Measurement Instruments will be implemented, with this we will be able to check the behaviour of the different mechanisms of articulated reusability through the library.

Key words Reusability|GUI|Pattern|Module|Library|IDE

Contents

Nomenclature

API Application Programming Interface, page 77

BackEnd Parte de la programacion que se encarga de desarrollar todo lo que no vemos como usuarios dentro de una aplicacion o pagina web, page 78

BDD Behaviour Drive Development (Desarrollo guiado por el comportamiento), page 78

FTL FreeMarker Template Language, page 74

GoF The Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, autores del libro: Design Patterns: Elements of Reusable Object-Oriented Software, page 11

GRASP objected oriented design General Responsability Assignment Sortware Patterns (patrones generales de software para asignacion de responsabilidades en diseños orientados a objetos), page 15

IDE Integrated Development Environment (Entorno de Desarrollo Integrado), page 38

JAR Joint Application Requirement (Requerimiento de Aplicacion Conjunta), page 56

Matisse Modulo de construccion de interfaces graficas de Netbeans, page 78

MVC Modelo Vista Controlador, page 20

MVC Patron Modelo Vista Controlador, page 38

POO Programacion Orientada a Objetos, page 10

SOA Service Oriented Arquitecture, page 31

SOLID Single responsability-Open-closed-Liskov substitution -Interface Segregation-Dependency inversion, page 17

TDD Test-Driven Development (Desarrollo guiado por pruebas), page 78

List of Figures

Índice de tablas

Chapter 1

Introducción general y objetivos

En este capitulo ofrecemos una vision general del contexto teorico en el que desarrollaremos el proyecto, expondremos los principales motivos que promueven su desarrollo y los distintos objetivos a alcanzar a lo largo de todo el proyecto. Finalmente mostraremos la estructura de los distintos capitulos en la que hemos organizado este documento.

1.1. Introduccion.

La reutilizacion de codigo ha sido y es uno de los pilares basicos en el desarrollo de los sistemas informaticos actuales. Todas las arquitecturas de los sistemas orientados a objetos se fundamentan en esa idea, y muchas de las tecnicas y conceptos que se manejan en este paradigma de programacion tienen la reutilizacion de codigo como eje fundamental: la herencia de clases, la composicion de objetos, los enlaces dinamicos, las bibliotecas de software, el polimorfismo, las interfaces, la abstraccion, la encapsulacion, todas son tecnicas que evitan la redundancia de codigo y facilitan su comparticion y reutilizacion.

Uno de los mecanismos de reutilizacion mas ampliamente utilizados en la actualidad en el desarrollo de software es el de la aplicacion de patrones. Los patrones basicamente son descripciones de problemas que suceden una y otra vez, junto a la aplicacion de la solucion a dicho problema, todo planteado dentro de un determinado contexto. Dichos patrones, una vez muestran la eficacia de la solucion que ofrecen, se documentan y pasan a formar parte del catalogo de patrones que, en el ambito del software, los desarrolladores utilizan a su conveniencia cuando se enfrentan a problemas similares a los descritos por el patron. Existen infinidad de patrones que se aplican a lo largo de todo el ciclo de vida del software, desde el analisis y la especificacion de requisitos, pasando por el diseño de objetos, de la arquitectura del software, las pruebas o la interfaz de usuario. Todos ellos se articulan en torno al concepto de reutilizacion, de forma que se puedan aprovechar tanto el codigo ya elaborado, como las estructuras arquitectonicas o de diseño y facilitar de esa manera la labor de los desarrolladores, incrementando su productividad y la calidad del codigo, y reduciendo el tiempo

de desarrollo de sus proyectos.

En el departamento de Inteligencia Artificial de la Escuela Tecnica Superior de Ingenieria Informatica de la UNED, se plantearon utilizar esta tecnica para el diseño de la interfaz grafica de usuario (gui) de CyBeRSim, un software de simulacion y control de conductas en robots autonomos que estaban desarrollando. Se encontraron con la necesidad de elaborar una biblioteca que les permitiera reutilizar las interfaces graficas que habian diseñado para poder aplicarlas a nuevas extensiones y desarrollos del proyecto sin tener que duplicar codigo. Este planteamiento surgio a raiz de una serie de necesidades a las que se enfrentaron. Por una parte el problema de como crear una jerarquia de guis, diseñadas generalmente mediante un diseñador grafico, para editar una jerarquia de clases de manera que no se duplicaran los componentes graficos (widgets) ni el codigo. Por otra parte el problema de integrar distintas guis en una gui general, pero conservando la opcion de utilizar cada una de ellas individualmente cuando fuera necesario. Ademas, todo el proceso debia resultar transparente tanto para usuarios como para programadores.

Diseñaron la biblioteca en C++, el lenguaje de programacion en el que se estaba escribiendo CyBerSym, usando la libreria grafica Qt-4 y utilizandola y aplicandola a traves del software de diseño de GUIs QtDesigner. De esta manera consiguieron integrar los componentes de la biblioteca en el entorno de diseño grafico para que los diseñadores de guis pudiesen utilizarlos como si se tratara de un elemento mas de diseño, de esta manera podrian elaborar diseños reutilizando otros realizados previamente o integrando partes de otros realizados en desarrollos paralelos. Por otra parte, les facilitaron a los programadores de aplicaciones las interfaces necesarias para poder usar estas guis en sus programas y poder integrarlas en guis generales o utilizarlas individualmente segun las necesidades de la aplicacion.

A raiz de este desarrollo se extrajeron las pautas esenciales para articular un patron de diseño de reusabilidad de GUIs mediante el cual poder formalizar la solucion aplicada al problema que ellos habian encontrado, de forma que otros pudiesen utilizarla en sus diseños.

Este patron de diseño es el que nosotros desarrollaremos. Realizaremos una descripcion de dicho patron en base a los esquemas de los distintos patrones de diseño existentes, y ofreceremos la implementacion en lenguaje Java de dicho patron. Elaboraremos ademas un modulo para integrar los componentes desarrollados en un entorno de desarrollo integrado donde podamos incorporarlos al editor grafico de dicho entorno. Asimismo, elaboraremos un programa de prueba en base a una aplicacion en la que se implementara una interfaz grafica de usuario para la edicion de un Catalogo de Instrumentos de Medicion mediante el cual podamos poner en valor los mecanismos de reutilizacion del patron y su utilidad.

1.2. Motivación y objetivos

El objetivo principal de este PFG es realizar una implementacion del Patron de Diseño de Reusabilidad de Guis, para lo que previamente realizaremos una descripcion de dicho patron siguiendo el es-

quema ofrecido por [Gamma et al., 1994] de forma que podamos llegar a definir con mayor precision el problema que describe el patron y las posibilidades de las que disponemos a la hora de aplicar la solucion que nos ofrece.

La implementacion del patron se realizara en forma de biblioteca de clases, de forma que pueda ser utilizada en cualquier aplicacion desarrollada en lenguaje Java e integrada en el catalogo de bibliotecas del entorno de desarrollo escogido para realizar el diseño de la interfaz grafica de la aplicacion. La descripcion del patron establecera las pautas a seguir a la hora de realizar el diseño de las clases que conformaran la biblioteca.

La intencion de nuestra propuesta es mejorar y facilitar la tarea de los programadores a la hora de desarrollar las interfaces graficas de usario, tanto como agilizar el proceso de integracion de dichas interfaces en el desarrollo de nuevas aplicaciones. Para ello implementaremos diferentes tipos de funcionalidad.

Por un lado, le facilitaremos a los desarrolladorer de aplicaciones una interfaz publica mediante la cual puedan crear dialogos nuevos y añadir uno en otros para conformar las guis que vayan a utilizar, de manera que puedan crear una jerarquia de guis con las que editar las clases que hayan generado en el desarrollo de la aplicacion. Podran integrar distintos tipos de gui, anidando y añadiendo unas en otras, e incluso incluir las que les facilite el diseñador de guis para poder generar la interfaz grafica que precisen sin necesidad de duplicar codigo.

Por otra parte, desarrollaremos un plugin para NetBeans de forma que la biblioteca se añada al catalogo de bibliotecas del entorno y el diseñador o el desarrollador de aplicaciones puedan utilizarla con total libertad como si se tratara de una libreria mas. Se creara tambien como parte del modulo los templates o plantillas de los diversos tipos de gui para poder instanciarlos directamente desde el asistente del menu de NetBeans y crear el dialogo desde cero. Igualmente, como parte del mismo modulo, generaremos las instancias de los distintos tipos de gui como widgets de la paleta componentes de la vista de diseño del editor grafico. Mediante estos widgets, el diseñador de guis podra simplemente arrastrar y soltar el componente que desee en el panel de diseño a la hora de elaborar las diferentes guis.

Todas estas funcionalidades seran totalmente transparentes tanto para el desarrollador de aplicaciones como para el diseñador de guis, pero tambien deberan serlo para el usuario de la aplicacion, de forma que este no encuentre ninguna distincion entre la interfaz grafica de una aplicacion desarrollada con guis reusables respecto a cualquier otra. Para ello todos los procesos de cancelacion, validacion y cualquier funcionalidad activada por el usuario, debe propagarse por toda la estructura de guis anidadas como si en realidad esta no existiera.

Finalmente, desarrollaremos una pequeña aplicacion para la edicion de una parte del Catalogo de Instrumentos de Medicion para la marca de metrologia Mitutoyo utilizando los mecanismos implementados mediante la libreria. Esta pequeña aplicacion nos servira como programa de pruebas y como muestra de la versatilidad de la biblioteca y su utilidad.

Como parte del desarrollo del proyecto en su aspecto mas teorico, intentaremos dar respuesta a

diversas preguntas como: ¿en que consiste la reutilizacion de codigo?, ¿cuales son sus mecanismos?, ¿que tipos de patrones existen?, ¿que es un patron de diseño?, ¿porque son tan importantes?. Y formularemos el esquema del patron que pretendemos implementar.

1.3. Estructura de la memoria

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

1. Introducción general y objetivos: que resume el proyecto y expone los principales objetivos a conseguir asi como los motivos para realizar este proyecto.

2. Estado del Arte: donde ofrecemos una vision del contexto teorico en el que vamos a enmarcar todo el desarrollo del proyecto. Introduciremos el concepto de reutilizacion, y expondremos los diferentes mecanismos de reutilizacion existentes. Mostraremos una panoramica sobre los distintos patrones de software existentes y definiremos lo que es un patron de diseño. Finalmente formularemos un esquema para la descripcion del Patron de Reusibilidad de Guis.

3. Analisis: en este apartado realizaremos distintos analisis previos a la especificacion de requisitos con los que elaboraremos los distintos casos de uso aplicables a la libreria que pretendemos desarrollar.

4. Diseño: en este capitulo ofreceremos una perspectiva sistematica del proceso de diseño a la hora de extraer las clases que conformaran la libreria. Propondremos un modelo de diseño en base a la combinacion de diversos patrones de diseño y ofreceremos una vision del entorno de diseño sobre el que pretendemos articular un modulo con nuestra biblioteca. Igualmente ofreceremos un boceto de los elementos que conformaran la implementacion de dicho modulo.

5. Implementacion: donde ofreceremos una vision detallada de todo lo que tenga que ver con la implementacion tanto de la biblioteca como del plugin de NetBeans. Mostraremos las distintas arquitecturas realizadas para la implementacion y repasaremos las clases que la conforman y los metodos utilizados para su articulacion. Igualmente repasaremos los archivos que conformaran el modulo del IDE.

6. Experimentacion y pruebas: en este capitulo mostraremos la estructura del programa de pruebas elaborado para demostrar la viabilidad de la biblioteca implementada. Haremos un repaso de los distintos tipos de pruebas que se han realizado al desarrollar el programa y detallaremos algunos de los mecanismos implementados como formula particular del desarrollo. Se mostrara tambien las pruebas realizadas respecto a la integracion del modulo elaborado para NetBeans.

7. Conclusiones y trabajos futuros: finalmente en este capitulo se muestran las conclusiones, comentando la consecucion o no de los distintos objetivos planteados, asi como lo aprendido a lo largo de todo el trabajo y las posibles mejoras aplicables a futuro.

Chapter 2

Estado del Arte

En este capitulo se presentara un concepto clave en el desarrollo actual de software como es el de la reutilizacion. Revisaremos las formas de reutilizar el conocimiento que se ha ido adquiriendo sobre las distintas facetas involucradas en la creacion de software. Introduciremos el concepto de patron de software, concretamente el de patron de diseño. Veremos los elementos que conforman un patron de diseño, los mecanismos de reutilizacion que utilizan y como se estructuran. Repasaremos los distintos patrones de diseño que introdujeron The Gang of Four en el diseño de sistemas orientados a objetos y los principios GRASP de diseño de clases, ya que muchos de ellos los utilizaremos en el modelo de diseño y la implementacion del patron de reusabilidad de guis. Mostraremos la forma en que los patrones de diseño pueden mejorar la calidad y la reusabilidad de un producto software mediante un ejemplo clasico como es el del patron MVC (Modelo/Vista/Controlador), y elaboraremos un boceto del patron que pretendemos implementar. Finalmente, ofreceremos una panoramica de la amplia y extendida variedad de patrones de software existentes, introducidos en todo el ciclo del desarrollo de software, y que denotan la relevancia que ha adquirido la aplicacion de patrones en el ambito de la programacion.

2.1. La reutilizacion de codigo.

La estrategia de resolucion de problemas, en muchas de las facetas del ser humano, es determinar en primera instancia si este ya ha sido resuelto antes, y en caso contrario, buscar analogias con otros problemas ya resueltos y adaptar su solucion al problema al que nos enfrentamos. Cuando ninguna de las opciones anteriores nos es viable, acudimos a nuestra capacidad de analisis y a nuestras habilidades para enfrentar el problema. [Prieto-Diaz, 1993]Tomar como referencia el trabajo realizado anteriormente en el dominio del problema que tenemos que afrontar suele ser un acto comun y generalizado que realizamos de manera informal o intuitiva, aunque a veces tiene un caracter mas procedimental, siguiendo pautas establecidas por la literatura precedente entorno a la problematica y que siempre nos facilita la labor de la reutilizacion.

En la ingenieria de software la reutilizacion de codigo ha sido de un tiempo a esta parte un factor esencial a la hora de desarrollar productos informaticos, incrementando en gran medida la productividad y la calidad del software producido. Su utilizacion se fundamenta en el desarrollo de sistemas usando elementos preexistentes en lugar de tener que elaborarlos de cero, utilizando conocimiento, procesos, metodologias o componentes ya existentes para adaptarlo a una nueva necesidad.[Frakes and Terry, 1996]

Desde sus inicios, la idea de la reutilizacion de software[1] ha sido interpretada como un valioso mecanismo para superar la crisis del software: reducir el tiempo de desarrollo, incrementar la productividad de los desarrolladores, reducir la densidad de errores y por lo tanto mejorar la calidad del software. En un principio se propuso como una libreria de componentes de codigo fuente y tecnicas automatizadas para su adaptacion a los distintos desarrollos. Esta idea fue el germen del concepto que luego se denominaria *factoria de software*[2] donde se establecieron un conjunto de procedimientos bien definidos, mediante los cuales los programadores reutilizaban segmentos de codigo de desarrollos anteriores. Posteriormente, se comenzo a aplicar la reutilizacion formalmente en el ambito de distintas organizaciones, y se iniciaron distintas investigaciones de caracter academico en relacion a la reusabilidad del codigo, consensuando la necesidad de extender la reutilizacion a diferentes niveles de abstraccion, mas alla de considerarlo tan solo desde la perspectiva del codigo fuente. A dia de hoy la reutilizacion de codigo se viene aplicando a cualquier producto del ciclo de vida del software y a todos los aspectos y recursos que se encuentran vinculados a su proceso de desarrollo.

2.1.1. Las categorias del conocimiento reusable.

Existen muchos tipos de propuestas en relacion a la categorizacion del conocimiento reusable, pero es costumbre categorizar el trabajo de reutilizacion de software basandose en *que* esta siendo reutilizado (el *objeto de reutilizacion*) y en el *como,* el *metodo de reutilizacion*[Mili, 1995], ambas estrechamente relacionadas pero que distinguen las diferentes interpretaciones del ambito de las categorias que pueden establecerse.

2.1.1.1. El Objeto de reutilizacion

En lo que respecta al objeto de reutilizacion, por lo general, la mayor parte de las clasificaciones se basan en uno, o en una combinacion, de los siguientes factores:

- etapa del desarrollo en el cual se produce o usa el conocimiento, desde la especificacion de requisitos, pasando por el diseño de clases, el diseño arquitectonico, la implementacion, las pruebas, etc...

[1] Planteada por McIlroy en 1968 en su conferencia "Mass produced software components"[D.McIlroy, 1968]
[2] Termino acuñado por SDC (System Development Corp) en 1974

- nivel de abstraccion, si se trata de reusar abstracciones sobre alguna de las facetas del desarrollo que luego se apliquen al codigo concreto o si se trata de reutilizar directamente codigo ya implementado con anterioridad. Por lo general se interpreta que a mayor nivel de abstraccion, mayor capacidad de reutilizacion.

- naturaleza del conocimiento, si se trata de artefactos concretos o habilidades aplicadas al desarrollo. En el caso de los artefactos, se identificaron cuatro tipos de artefactos reusables [Jones, 1984]: *data reuse* (reutilizacion de datos), *architectures reuse* (reutilizacion de arquitecturas), *design reuse* (reutilizacion de diseños) y *program reuse* (reutilizacion de programas).

Pero por lo general, se pueden identificar diez objetos que se pueden manejar para la reutilizacion a la hora de afrontar un proyecto de software [Frakes and Terry, 1996] :

Datos.
Codigo fuente.
Arquitecturas.
Diseño.
Documentacion.
Templates.
Interfaces de usuario.
Planes.
Requisitos.
Pruebas.

Estos aspectos abarcan todo el ciclo de desarrollo de un proyecto de software y suponen una amplia influencia del concepto de reutilizacion, mas alla del ambito exclusivo de la generacion de codigo.

2.1.1.2. El metodo de reutilizacion

Por otro lado, en relacion al *metodo de reutilizacion*, existen diferentes propuestas entre las que se pueden distinguir las siguientes tecnicas:

- La reutilizacion por composicion o composicional, que utiliza componentes (de grano fino) existentes en colecciones o bibliotecas como bloques constructivos. P.ej. el empleo de funciones o subrutinas.

- Reutilizacion de modulos o componentes software, elementos de mayor entidad que las funciones que proveen servicios a otros modulos o componentes del sistema sin llegar a considerarse un sistema completo. P.ej. los paquetes de Ada o la Programacion Orientada a Objetos.

- Reutilizacion de templates o meta-componentes. Un template es un esqueleto de un artefacto, compuesto de una parte fija independiente y una parte que debe ser definida por el desarrollador

y concretada para el producto final. P.ej. templates de documentacion y de pruebas que definen el conjunto de casos de prueba y de documentos para el artefacto.

- La ingenieria de dominio, aplicada a familias o lineas de productos, consistiendo en agrupar, organizar y almacenar la experiencia obtenida en la construccion de sistemas sobre un dominio particular en forma de recursos reutilizables. Es un concepto clave en la reutilizacion sistematica.

- Frameworks orientados a objetos. Son aplicaciones semicompletas, implementadas generalmente como una jerarquia de clases. Una vez que el framework ha sido desarrollado, pueden derivarse de el multiples aplicaciones.

- La reutilizacion generativa es un proceso que consiste en la codificacion de conocimiento sobre el dominio y sobre la construccion de sistemas en ese dominio en un generador de aplicaciones especifico. Los nuevos sistemas son creados escribiendo especificaciones en un lenguaje propio del dominio. El generador traduce las especificaciones en codigo para el nuevo sistema en un lenguaje objetivo. P.ej. los analizadores lex y yacc son generadores de codigo.

- Reutilizacion basada en componentes. Este tipo de reutilizacion de componentes se encuentra ligada a un modelo de componentes particular como CORBA/CCM/EJB, dirigida a una plataforma de componentes concreta. Los componentes reutilizables pueden ser recuperados de un repositorio y compartidos entre productos en una familia de productos o como componentes Open Source Software o COSTS (Commodity off-the-shelf).

- Los repositorios o bibliotecas de reutilizacion, que constan de almacenes de recursos reutilizables, herramientas para la busqueda de recursos, y utilidades para la gestion del cambio y el aseguramiento de la calidad. Pueden ser genericos o especificos de un dominio y pueden combinarse con otras propuestas para la reutilizacion.

Existen infinidad de formulas para categorizar y aplicar el conocimiento reusable disponible, asi como multiples facetas en las que la reutilizacion se ha venido aplicando, incluido el paradigma de la Programacion Orientada a Objetos (POO) , que se fundamenta precisamente en este concepto. En nuestro caso, la faceta que vamos a contemplar mas detenidamente es el aspecto de la reutilizacion en el diseño, concretamente en el diseño de clases de la POO, donde el desarrollo del concepto de patron ha tenido una enorme repercusion, identificando aspectos similares de soluciones aplicadas a problemas de diseño y abstrayendolas para configurar un catalogo de patrones de diseño que puedan ser aplicados en el desarrollo de sistemas orientados a objetos de forma que sean sistemas mas flexibles y adaptables y tambien mas reutilizables.

2.2. Patrones de diseño.

La idea de elaborar y categorizar un catalogo de elementos reusables aplicables al diseño de sistemas software surgio a raiz de la publicacion del libro *Design Patterns: Elements of Reusable Object-Oriented Software*[Gamma et al., 1994] elaborado por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, a los que se denomino The Gang of Four (GoF) . En el se exponen distintos patrones de diseño de uso extendido entre especialistas en programacion de objetos que realizaban diseños con mucha mayor agilidad que otros, simplemente reutilizando soluciones que ya habian utilizado en el desarrollo de programas anteriores. Los autores recogen 23 diseños comunmente utilizados hasta entonces, y que a dia de hoy son de uso comun en el desarrollo de sistemas orientados a objetos, y los documentan detalladamente en forma de *patrones de diseño*, aunque el concepto de patron y su uso ya se venia utilizando con anterioridad.

2.2.1. Que es un patron de diseño.

En toda la literatura en relacion a los patrones, tanto de diseño como de cualquier otro tipo, se hace mencion de la aportacion que hizo en 1979 el arquitecto Chistopher Alexander en su libro *The Timeless Way of Building* [Alexander, 1979]donde propone y plantea una nueva teoria de la arquitectura, y del diseño en general, basandose en la comprension y configuracion de patrones de diseño. Aunque salio publicado con posterioridad, es esencialmente la introduccion de *A Pattern Language* [Alexander et al., 1977] y *The Oregon Experiment* [Alexander, 1975], donde se inspiro Kent Beck en los años ochenta para concebir los primeros patrones de software junto a Ward Cunningham [Beck and Cunnigham, 1987]. Tres de los libros posteriores mas destacados que aplicaron las ideas de Alexander a la programacion fueron [A.Salingaros,]:

1. James Coplien and Douglas Schmidt (Editors), Pattern Languages of Program Design (Reading, Massachusetts: Addison-Wesley, 1995).

2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns (Reading, Massachusetts: Addison-Wesley, 1995).

3. Richard Gabriel, Patterns of Software (New York: Oxford University Press, 1996). Con prologo de Christopher Alexander.

Como se puede observar, el libro de The Gang of Four es uno de los libros destacados que se vieron influenciados por las ideas de Alexander en relacion al lenguaje de patrones. Aunque Alexander se referia en todo momento a patrones aplicados a ciudades y edificios, los conceptos eran perfectamente validos para patrones de diseño orientados a objetos.

Christopher Alexander define el termino patron de la siguiente manera:

" Each pattern is a three-part rule., which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.
The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tell us how to create that thing and when we must create it.
It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing" [Alexander, 1979]

Segun Alexander, "cada patron describe un problema que ocurre una y otra vez en nuestro entorno, asi como la solucion a ese problema, de tal modo que se pueda aplicar esta solucion un millon de veces, sin hacer lo mismo dos veces" [Alexander et al., 1977]. Estos conceptos los aplican the GoF como patrones de diseño a sistemas orientados a objetos, definiendolos como "descripciones de clases y objetos relacionados que estan particularizados para resolver un problema de diseño general en un determinado contexto" [Gamma et al., 1994].

Los patrones de diseño estan concebidos como abstracciones que identifican los aspectos clave de una estructura de diseño comun, extraidos de problemas conocidos y tratados recurrentemente en los sistemas orientados a objetos, en base a soluciones aplicadas con exito. La aplicacion de estos patrones fomenta la eleccion de alternativas de diseño que hacen que un sistema sea reutilizable, y a evitar aquellas que dificultan dicha reutilizacion.

2.2.2. Mecanismos de reutilizacion de los sistemas orientados a objetos.

La misma naturaleza de los sistemas orientados a objetos esta enfocada a la reutilizacion, pero la aplicacion de los patrones de diseño nos muestra que para que esa reutilizacion sea efectiva, deben aplicarse en el proceso del diseño de clases una serie de mecanismos de reutilizacion de los que dispone para conseguir la flexibilidad y adaptabilidad necesaria.

2.2.2.1. Herencia y composicion.

La *herencia de clases* y la *composicion de objetos* son dos de las tecnicas mas comunes para reutilizar funcionalidad en los sistemas orientados a objetos.

La *herencia de clases* no es mas que un mecanismo para extender la funcionalidad de una aplicacion reutilizando la funcionalidad de las clases padres. Permite definir una implementacion en terminos de otra heredando la mayoria de funcionalidad que precisa de clases ya existentes. Este

mecanismo no reutiliza tan solo la implementacion de la clase padre, tambien es capaz de definir interfaces identicas para crear familias completas de objetos. El polimorfismo se basa en este mecanismo de herencia de interfaces, y muchos de los patrones de diseño ofrecen distintos modos de asociar una interfaz con su implementacion de manera que resulte transparente. "Programar para interfaces, no para implementaciones" reduce las dependencias de implementacion entre subsistemas, favoreciendo la reutilizacion de sistemas.

A esta forma de reutilizacion se la denomina *reutilizacion de caja blanca*, en el sentido de que con la herencia, las interioridades de las clases padres suelen hacerse visibles a las subclases.

La *composicion de objetos* es un mecanismo mediante el cual, la nueva funcionalidad se obtiene ensamblando o componiendo objetos para obtener funcionalidad mas compleja, y requiere que los objetos a componer tengan interfaces bien definidas. De esta manera, se evita romper con la encapsulacion, cosa que ocurre con la herencia, ya que las dependencias de implementacion entre la clase padre y las subclases provocan que cualquier cambio en la primera obliga a cambiar las segundas, cosa que limita la flexibilidad y la reutilizacion.

A este estilo de reutilizacion se la denomina de *caja negra*, ya que los detalles internos de los objetos no son visibles, cada objeto aparece solo como una caja negra de la que no se conoce su implementacion.

Se puede añadir nueva funcionalidad componiendo los objetos existentes de otra forma en vez de definir subclases nuevas de otras ya existentes, si bien es cierto que un uso intensivo de la composicion puede derivar en diseños mas dificiles de entender.

Por lo comun ambos mecanismos suelen trabajar juntos, y aunque suele utilizarse con mas asiduidad la herencia, suele ser preferible la composicion de objetos por favorecer diseños mas reutilizables y mas simples.

2.2.2.2. La Delegacion.

Con la *delegacion*, dos objetos son los encargados de tratar una peticion: el objeto receptor delega operaciones en su objeto delegado. En lugar de hacer que una clase A sea una subclase de otra B, la clase A puede reutilizar el comportamiento de B guardando una instancia de esta en una variable y delegando en ella el comportamiento especifico de B. La clase A contiene a B y en lugar de heredar las operaciones de B, reenvia las peticiones a la instancia de B que posee.

La ventaja principal de la delegacion es que hace facil combinar comportamientos en tiempo de ejecucion y cambiar la manera en que estos se combinan. A puede hacerse C en tiempo de ejecucion simplemente cambiando su instancia de B por una instancia de C, siempre que B y C tengan el mismo tipo.

El inconveniente mas comun de la delegacion, derivado de los inconvenientes de la composicion, es que el software dinamico y altamente parametrizado es mas dificil de entender que el estatico. Por eso la delegacion es una buen mecanismo a utilizar siempre y cuando simplifique mas de lo que

complique.

La delegacion es un ejemplo extremo de composicion de objetos y es una buena muestra de como siempre se puede sustituir la herencia por la composicion de objetos como mecanismo de reutilizacion de codigo.

2.2.3. Estructura de un patron de diseño.

Segun [Gamma et al., 1994], un patron de diseño puede describirse mediante diferentes formatos, pero consta de cuatro elementos esenciales:

El nombre del patron que permite describir en pocas palabras el problema de diseño, las soluciones y sus consecuencias. Se trata de crear un nombre que cree una abstraccion sobre el patron y lo defina en todos sus ambitos con solo mencionarlo, generando un vocabulario con el que poder definirlo. Acertar con un buen nombre es esencial si queremos pensar en el diseño y transmitir a otros la forma en que queremos construir nuestro sistema.

El problema describe el contexto en el que podemos aplicar el patron. Expone el ambito del problema al que el patron puede darle solucion, a veces en funcion de una serie de requisitos bajo los que tiene sentido aplicarlo, otras describiendo problemas de diseño concretos o estructuras que son sintomaticas de diseños rigidos que no promueven la reutilizacion.

La solucion describe los elementos que constituyen el diseño, sus relaciones, responsibilidades y colaboraciones. No es un diseño o una implementacion concreta, sino una plantilla con una disposicion general de elementos de diseño que resuelve la descripcion abstracta del problema que define el patron.

Las consecuencias son los resultados, las ventajas e inconvenientes de aplicar el patron. Estas consecuencias son esenciales para evaluar alternativas de diseño, comprender los costes y beneficios de aplicar el patron, y barajar las opciones de diseño que pueden crearse. La flexibilidad, extensibilidad y portabilidad de un sistema dependeran en gran medida de las consecuencias de aplicar el patron al diseño, y por lo tanto a su capacidad de reusabilidad.

Otros autores adoptan un esquema elemental de tres partes [Buschmann et al., 1996]:

Context situacion de diseño que da lugar a un problema de diseño. El contexto extiende la dicotomia plana problema-solucion describiendo situaciones en las que el problema ocurre.

Problem conjunto de fuerzas que surgen repetidamente en el contexto y que pueden complementarse o contravenirse entre si.

Solution configuracion necesaria para equilibrar las fuerzas, en la arquitectura de software, cada solucion incluye dos aspetos:

- cada patron especifica una estructura estatica, una configuracion espacial de elementos que se relacionan entre si.
- cada patron especifica un comportamiento dinamico en tiempo de ejecucion, determinando como colaboran, se comunican y organizan el trabajo los participantes del patron.

Este esquema captura la esencia de un patron independientemente de su dominio y subyace en la descripcion de patrones que realizan muchos otros autores, entre ellos:
[Alexander et al., 1977, Coplien, 1994, Meszaros, 1994]

2.2.4. Los patrones GoF.

Los patrones descritos por the GoF, asi como los patrones GRASP, son fundamentales a la hora de interpretar el diseño realizado para la implementacion del patron de reusabilidad de guis, de forma que realizaremos una breve exposicion y clasificacion de dichos patrones.

Abstract Factory (Factoria Abstracta) Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre si, sin especificar sus clases concretas.

Adapter (Adaptador) Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podria, convirtiendo la interfaz en otra distinta que se adapta a la nueva clase.

Bridge (Puente) Desacopla una abstraccion de su implementacion, de manera que ambas puedan modificarse de forma independiente.

Builder (Constructor) Separa la construccion de un objeto complejo de su representacion, de forma que el mismo proceso de construccion pueda crear diferentes representaciones.

Chain of Responsability (Cadena de Responsabilidad) Permite dar a mas de un objeto la posibilidad de responder a una peticion, creando una cadena con los objetos receptores y pasando la peticion a traves de la cadena hasta que esta sea tratada por algun objeto.

Command (Orden) Encapsula una peticion en un objeto, permitiendo asi parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.

Composite (Compuesto) Combina objetos en estructuras de arbol para representar jerarquias de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Decorator (Decorador) Añade dinamicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Facade (Fachada) Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea mas facil de usar.

Factory Method (Metodo de Fabricacion) Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan que clase instanciar. Permite que una clase delegue en sus subclases la creacion de objetos.

Flyweight (Peso Ligero) Usa el compartimiento para permitir un gran numero de objetos de grano fino de forma eficiente.

Interpreter (Interprete) Dado un lenguaje, define una representacion de su gramatica junto con un interprete que usa dicha representacion para interpretar sentencias del lenguaje.

Iterator (Iterador) Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representacion interna.

Mediator (Mediador) Define un objeto que encapsula la forma en que interactuan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explicitamente, y permite variar la interaccion entre ellos de forma independiente.

Memento (Recuerdo) Representa y externaliza el estado interno de un objeto sin violar la encapsulacion, de forma que este puede volver a dicho estado mas tarde.

Observer (Observador) Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualizan automaticamente todos los objetos que dependen de el.

Prototype (Prototipo) Especifica los tipos de objetos a crear por medio de una instancia prototipica, y crea nuevos objetos copiando de este prototipo.

Proxy (Apoderado) Proporciona un sustituto o representante de otro objeto para controlar el acceso a este.

Singleton (Unico) Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella.

State (Estado) Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecera que cambia la clase del objeto.

Strategy (Estrategia) Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que una algoritmo varie independientemente de los clientes que lo usan.

Template Method (Metodo Plantilla) Define en una operacion el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

Visitor (Visitante) Representa una operacion sobre los elementos de una estructura de objetos. Permite definir una nueva operacion sin cambiar las clases de los elementos sobre los que opera.

La clasificacion que establecieron the Gang of Four de toda esta serie de patrones sigue dos criterios [Gamma et al., 1994]:

1. el PROPOSITO, que refleja lo que hace un patron. Los patrones pueden tener distintos propositos :
   - los patrones *de creacion* tienen que ver con el proceso de creacion de objetos.
   - los patrones *estructurales* tratan con la composicion de clases u objetos.
   - lo patrones *de comportamiento* caracterizan el modo en que las clases y objetos interactuan y se reparten la responsabilidad.
2. el AMBITO, que especifica si el patron se aplica principalmente a clases o a objetos. Los patrones de clases se ocupan de las relaciones entra las clases y sus subclases, son relaciones de herencia, por tanto estaticas. Los patrones de objetos tratan con las relaciones entre objetos, que pueden cambiarse en tiempo de ejecucion y son mas dinamicas.

		Proposito		
Ambito		**De Creacion**	**Estructurales**	**De Comportamiento**
	Clase	Factory Method	Adapter (de clases)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (de objetos) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Table 2.1: Cuadro de clasificacion de Patrones de Diseño

2.2.5. Patrones GRASP.

Otra familia de patrones muy representativos y que se utilizan con suma asiduidad en el diseño de sistemas orientados a objetos son los patrones GRASP (General Responsability Assignment Software

Patterns). Estos patrones junto a los principios SOLID , estan considerados principios generales basicos de diseño de clases, asignando responsabilidades entre los objetos de manera que se genere un sistema que sea facilmente extendible, adaptable y de facil mantenimiento, es decir que cumpla con los requisitos adecuados para la reutilizacion.

Estos principios o patrones basicos de diseño de objetos son:

Experto en Informacion La responsabilidad de crear un objeto debe recaer siempre sobre la clase que conoce toda la informacion necesaria para crearlo. Es un principio general de asignacion de responsabilidades, favorece el encapsulamiento y la distribucion de comportamiento entre las clases que contienen la informacion requerida. Todo esto favorece la alta cohesion y el bajo acoplamiento.

Creador Este patron guia la asignacion de responsabilidades relacionadas con la creacion de objetos, establece quien debe ser el responsable de la creacion de nuevos objetos o clases. La nueva instancia debera ser creada por la clase que:

- Tiene los datos de inicializacion del objeto a crear
- Utiliza estrechamente objetos de la clase a instanciar.
- Almacena o registra instancias de la clase.
- Contiene o agrega objetos de la clase.

Al usar este patron se establece una relacion de visibilidad entre la clase creada y la clase creador, por lo que la forma de determinar la relacion creador-creacion puede generar diseños con mayor encapsulacion, un acoplamiento mas bajo entre clases, y una mayor facilidad de mantenimiento y reutilizacion.

Controlador Este patron funciona como intermediario entre una determinada interfaz y el algoritmo que la implementa, es responsable de recibir o manejar eventos del sistema y de intermediar entre la logica de negocios y la capa de presentacion. Normalmente delega en otros objetos el trabajo que se necesita hacer, su unica funcion es la de coordinar y controlar la actividad. Utilizar este patron aumenta el potencial de reutilizacion de interfaces y de la logica de aplicacion.

Alta Cohesion La cohesion es la medida de la fuerza con la que las responsabilidades asignadas a un elemento estan relacionadas. Un elemento con responsabilidades altamente relacionadas y que no hace una gran cantidad de trabajo tiene alta cohesion. Por el contrario, una clase con baja cohesion hace muchas cosas no relacionadas o hace demasiado trabajo, generalmente este tipo de clases son dificiles de entender, de reutilizar y de mantener, y ademas suelen ser sumamente delicadas ya que se ven constantemente afectadas por los cambios. Al utilizar clases altamente cohesivas se incrementa la claridad, se facilita la comprension del diseño y se

simplifican el mantenimiento y las posibles mejoras del sistema. Ademas, las clases cohesivas se pueden utilizar para propositos muy especificos ya que su funcionalidad se encuentra altamente relacionada, lo que incrementa la capacidad de reutilizacion.

Bajo Acoplamiento El acoplamiento es la medida de la fuerza con que un elemento esta conectado a otro. Un elemento con bajo acoplamiento no depende de demasiados otros elementos, una clase con alto acoplamiento se relaciona con muchas otras clases lo que provoca que los cambios en unas clases fuerzan cambios en sus clases relacionadas, y son ademas dificiles de reutilizar ya que su uso requiere la presencia de las clases de las que depende. Aunque no existe una medida de cuando el acoplamiento es demasiado alto, siempre dependera del grado de problemas que aumentar el acoplamiento pueda llegar a dar. Por lo general es siempre recomendable asignar responsabilidades de manera que el acoplamiento permanezca bajo, lo que supondra que el sistema creado tiene un conjunto de objetos conectados adecuadamente entre si y que realizan las tareas en colaboracion.

La cohesion y el acoplamiento son las dos caras de la misma moneda, un mala cohesion causa generalmente un mal acoplamiento y viceversa. Ambos aspectos de diseño confluyen y promueven otro principio basico, la modularidad, que es la propiedad de un sistema que se ha descompuesto en un conjunto de modulos cohesivos y debilmente acoplados.

Polimorfismo El polimorfismo es un principio fundamental para diseñar como se organiza un sistema para gestionar variaciones. Cuando los comportamientos relacionados varian segun el tipo, las responsabilidades deben asignarse a los tipos para los que varia el comportamiento. Utilizando operaciones polimorficas se pueden gestionar y extender facilmente nuevas variaciones, y las nuevas implementaciones se pueden introducir sin afectar al resto de elementos.

Fabricacion Pura La fabricacion pura se da en clases que no representan un ente u objeto real del dominio del problema, sino que se ha creado con la intencion de disminuir el acoplamiento, aumentar la cohesion y potenciar la reutilizacion. Tales clases son una fabricacion de la imaginacion que idealmente soportan alta cohesion y bajo acoplamiento, de manera que el diseño de la fabricacion es muy limpio o puro, de ahi que se trate de una fabricacion pura. Se trata de clases creadas a raiz de la descomposicion del comportamiento, a las que se les asigna responsabilidades agrupando comportamientos o algoritmos, sin que esten relacionados con un concepto de la representacion del dominio del mundo real. Facilita la generacion de clases mas cohesivas, menos acopladas y con mas potencial de reutilizacion.

Indireccion Se trata de asignar la responsabilidad a un objeto intermedio que medie entre otros componentes o servicios de manera que no se acoplen directamente, el intermediario crea una *indireccion* entre los otros componentes.

Variaciones Protegidas Es el principio fundamental de protegerse del cambio, de tal forma que

todo lo que preveamos en un analisis previo que es susceptible de modificaciones, lo envolvamos con una interfaz estable alrededor para protegerla del cambio. Es un principio fundamental que motiva la mayoria de los mecanismos y patrones en la programacion y el diseño destinados a proporcionar flexibilidad y proteccion frente a las variaciones, por ejemplo, motiva la encapsulacion de datos, interfaces, polimorfismo, indireccion y otros estandares, asi como principios fundamentales como el Principio de Sustitucion de Liskov [3], la Ley de Demeter [4] o el principio Open/Closed [5], todos ellos destinados a proveer mayor flexibilidad aplicando el principio de variaciones protegidas.

2.3. El Modelo Vista Controlador.

Vamos a introducir una breve presentacion de un patron de diseño que pertenece mas al ambito de los patrones de diseño de arquitectura de software, pero que se repite en mucha de la literatura dedicada al estudio de patrones y al que haremos referencia en el desarrollo de la implementacion del patron de reusabilidad de guis. El patron en cuestion es la triada Modelo/Vista/Controlador (MVC) usada para construir interfaces de usuario en Smalltalk-80.

El desarrollo de interfaces de usuario ha supuesto siempre un reto para los programadores ya que siempre que se necesita extender la funcionalidad de una aplicacion, los menus deben modificarse para que se pueda tener acceso a las nuevas funciones y las interfaces de usuario deben poder adaptarse constantemente. Un sistema, por ejemplo, a menudo tiene que ser portado a otra plataforma con un estandar diferente de apariencia, incluso actualizar a un sistema nuevo de ventanas puede implicar muchos cambios en el codigo. En resumen, la interfaz de usuario de un sistema de larga vida es un objetivo en constante cambio.

Construir un sistema con la flexibilidad requerida sera costoso y propenso a errores si la interfaz de usuario esta estrechamente relacionada con el nucleo funcional. Esto puede resultar en el desarrollo y el mantenimiento de varios sistemas de software sustancialmente diferentes, uno por cada implementacion de la interfaz de usuario. Los cambios siguientes se extenderan sobre muchos modulos. Por lo tanto cuando se desarrolla un sistema software de este tipo, hay que considerar dos aspectos:

- Los cambios en la interfaz de usuario deberian ser faciles, y posibles en tiempo de ejecucion.
- Adaptar o portar la interfaz de usuario no deberia tener ningun impacto en el codigo del nucleo funcional de la aplicacion.

[3] "Si para cada objeto O1 de tipo S hay un objeto O2 de tipo T tal que para todos los programas P definidos en terminos de T, el comportamiento de P no varia cuanso se sustituye O1 por O2, entonces S es un subtipo de T". [Liskov, 1988]

[4] *No Hable con Extraños*. Hay que evitar crear diseños que recorren largos caminos de la estructura de objetos y envian mensajes, (hablan) con objetos distantes (extraños).

[5] Una entidad de software deberia estar abierta a la extension pero cerrada a la modificacion [Meyer, 1997]

Para solucionar este problema, se suele dividir la aplicacion en tres areas: procesamiento, salida y entrada:

- El componente *modelo* encapsula los datos y la funcionalidad del nucleo. El modelo es independiente de representaciones de salida especificas o de comportamiento de entrada.

- El componente *vista* dispone la informacion para visualizar al usuario. Una vista obtiene los datos que muestra del modelo. Puede haber multiples vistas del modelo.

- Cada vista tiene asociada un componente *controlador*. Los controladores reciben entrada generalmente a eventos que denotan el movimiento del mouse, la activacion de los botones del mouse o la entrada de teclado. Los eventos se traducen a solicitudes de servicio, que se envian al modelo o a la vista. El usuario interactua con el sistema unicamente a traves de controladores.

La separacion del modelo, la vista y el controlador permite multiples vistas del mismo modelo. Si el usuario cambia el modelo a traves del controlador de una vista, todas las demas vistas que dependen de estos datos deben reflejar el cambio. Para lograr esto, el modelo notifica todas las vistas cada vez que cambian sus datos. Las vistas a su vez recupera nuevos datos del modelo y actualizan la informacion que muestran. Esta solucion permite cambiar un subsistema de la aplicacion sin causar efectos importantes a otros subsistemas. Puedes cambiar por ejemplo de una interfaz grafica de usuario a una no grafica sin modificar el modelo subyacente. Tambien puedes añadir soporte para un nuevo dispositivo de entrada sin afectar la disposicion de la informacion o el nucleo funcional. Todas las versiones del software pueden operar bajo el mismo subsistema de modelo independientemente de la apariencia que tenga.

El patron MVC usa otros patrones de diseño, tales como Factory Method para especificar la clase controlador predeterminada de una vista, y el Decorator para añadir capacidad de desplazamiento a una vista, pero las principales relaciones se dan entre los patrones Observer, Composite y Strategy. Esto denota lo que se evidencia en todos los patrones de diseño que es la estrecha relacion que se establece entre ellos y como se complementan, ademas refleja como se llegan a conformar verdaderos sistemas de patrones que abarcan todo el desarrollo del software.

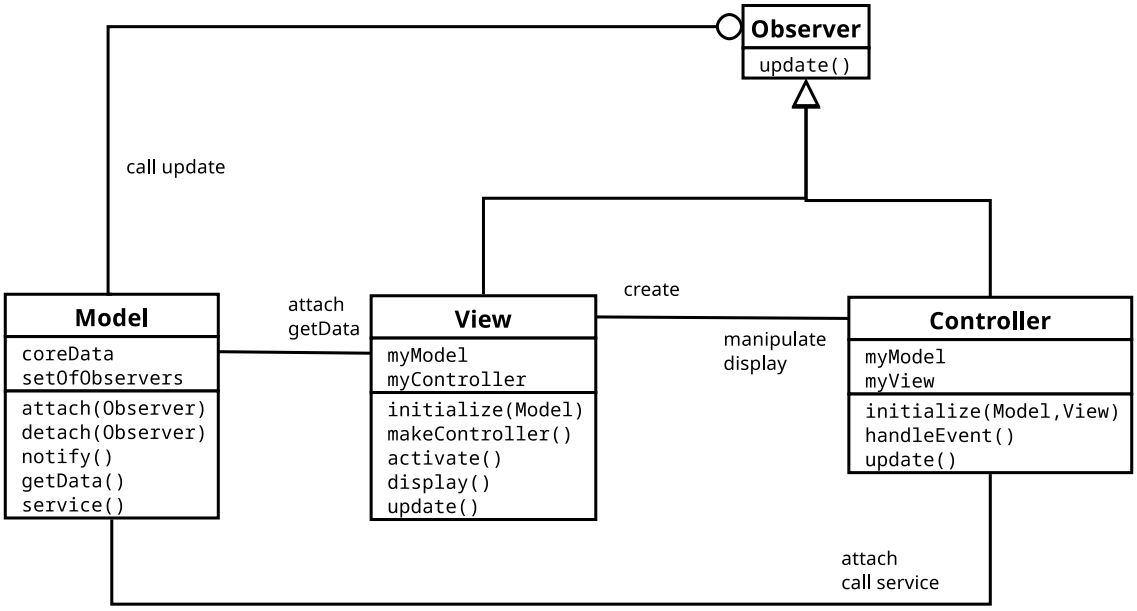


Figure 2.1: Esquema Modelo Vista Controlador[Buschmann et al., 1996]

2.4. El patron de reusabilidad de Guis.

En este apartado ofreceremos a modo de introduccion, una presentacion del patron de reusabilidad de guis siguiendo el esquema ofrecido por The GoF [Gamma et al., 1994]. No pretende ser una documentacion elaborada, sino tan solo una descripcion del patron para tomar como punto de inicio desde el que realizar su implementacion.

nombre Patron de Reusabilidad de GUIS (GUIS Reuse Pattern)

proposito permite integrar guis previamente diseñadas y añadir componentes para generar nuevas guis ampliadas, o utilizar los objetos de una gui general como guis independientes.

motivacion al elaborar la interfaz grafica de usuario de una aplicacion podemos encontrarnos con que para editar una clase derivada, tenemos que elaborar una gui que tan solo se diferencia en un par de campos de la editada para la clase base. Es decir, nos encontramos con que tenemos que duplicar todo el codigo utilizado para elaborar la gui de la clase base, y añadirle luego tan solo un par de campos mas para obtener asi la nueva gui de la clase derivada. Esto se traduce en una explosion exponencial de duplicacion de codigo si tenemos que elaborar las guis para toda la jerarquia de clases de una aplicacion.

Una adaptacion adecuada al problema nos llevaria a implementar una jerarquia de guis paralelas a la jerarquia de clases de la aplicacion que pudiese adaptarse al incremento de campos en las clases derivadas y que generara la gui en funcion de la necesidad de la clase a editar. Tratariamos entonces cada gui como una composicion de las guis previas mas los componentes

de clase en el arbol de jerarquias, obteniendo una jerarquia de guis paralela.

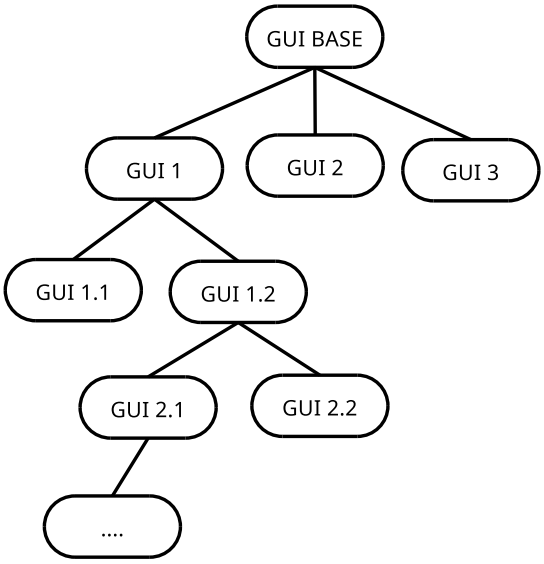


Figure 2.2: Esquema jerarquia de guis

Podriamos tambien realizar el diseño de distintas guis que representen componentes diseñados y cuya composicion conformen la gui general de una clase determinada. Diferentes composiciones con distintas guis de componente daran lugar a guis generales distintas para clases distintas, que a su vez pueden combinarse con unas u otras y configurar guis determinadas en funcion de la necesidad de cada clase.

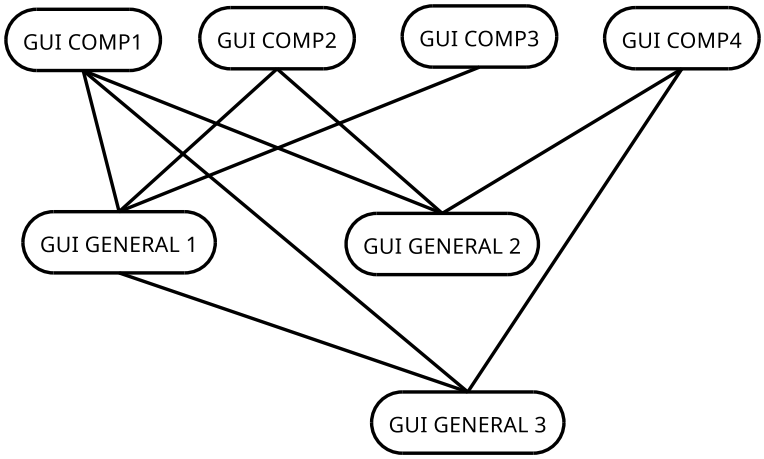


Figure 2.3: Esquema combinacion de guis

En definitiva, se trata de estructurar una composicion recursiva de guis utilizando una variante del patron composite, de manera que se puedan integrar distintos tipos de diseños en estructuras de diseño grafico distinto, en funcion de la composicion que se quiera realizar. Las guis

se comportaran como componentes graficos compuestos, integrables y adaptables a la adicion de cualquier otro componente.

aplicabilidad usese el patron de reusabilidad de guis cuando:

- se pretenda crear una jerarquia de interfaces graficas para editar una jerarquia de clases sin tener que duplicar codigo ni componentes graficos.
- se pretenda integrar interfaces graficas de objetos asociados en una gui general pero que se pueda disponer de dichas interfaces independientemente para conformar otras guis.

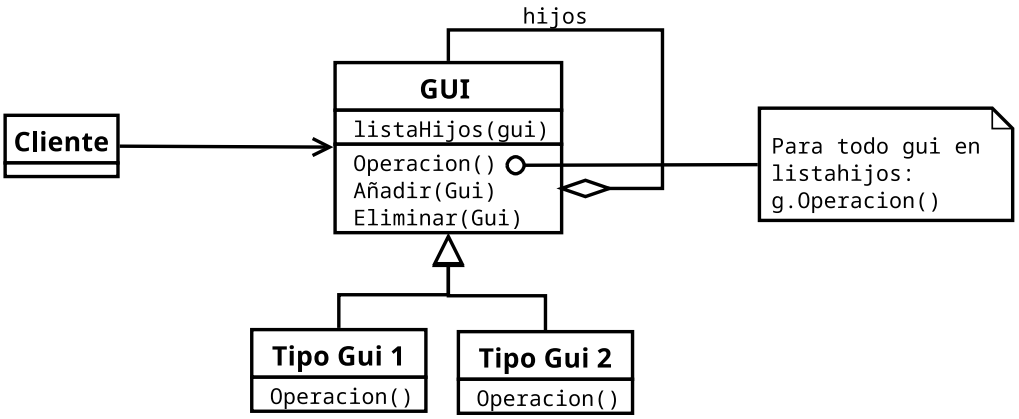


Figure 2.4: Esquema del patron de reusabilidad de Guis

estructura

participantes

- Gui
  - declara la interfaz para acceder a las guis de la composicion.
  - implementa una estructura de datos para almacenar todas las guis hijas y el comportamiento comun a todas ellas.
  - declara una interfaz para acceder y gestionar las guis hijas.
  - el diseño recursivo hace que la gui pueda convertirse en gui hija de otra gui mas general.
- Gui Hija
  - representa guis diseñadas que albergan diferentes tipos de componentes
  - implementa el comportamiento particular del tipo de gui hija definido
  - contiene los comportamientos primitivos de cada componente con los que tiene comunicacion directa

    - una gui hija puede ser tambien una gui compuesta de otras guis en la estructura recursiva
  - Cliente
    - maneja los objetos gui de la composicion a traves de la interfaz de la gui mas general.

colaboraciones los clientes usan la interfaz que les proporciona la gui para interactuar con los objetos gui hijas y con los objetos que las componen. Si el componente es una gui hija, las peticiones son dirigidas a las primitivas de los componentes , si es una gui compuesta, las peticiones se distribuyen a las guis hijas quienes las direccionaran al componente al que va dirigida la peticion.

consecuencias el patron de reusabilidad de guis

- establece una jerarquia de guis formadas por guis compuestas y guis hijas. Las guis hijas se componen de objetos primitivos y/o de composiciones de guis que tambien pueden estar formadas por objetos primitivos o de una composicion de guis, y asi de forma recurrente. Toda gui puede estar formada o por una serie de componentes o por una serie de guis.
- los clientes trataran la gui general de forma transparente, de manera que los componentes primitivos y las guis compuestas se manejaran de manera uniforme. No tendran constancia en ningun momento de que la gui que manejan esta compuesta de otras guis previamente diseñadas. De esta forma el codigo del cliente se simplifica, ya que no es necesario controlar el direccionamiento de las peticiones, ya que este lo hace internamente el patron.
- se pueden añadir nuevos diseños de guis incorporandolas como subclases de tipos de guis, estas se adaptaran a la interfaz y podran integrarse con otros tipos de guis sin necesidad de cambiar el codigo del cliente.
- nos obliga a cumplir las restricciones de tipo para poder manejar las guis e incorporar nuevos diseños de guis. Tendremos que realizar estas comprobaciones en tiempo de ejecucion.

implementacion a la hora de enfrentar la implementacion del patron de reusabilidad de guis, al fundamentarse en el patron composite deberemos tener en cuenta toda la problematica de la implementacion aplicada a este patron.[Gamma et al., 1994]

En la estructura de diseño del patron que hemos ofrecido, hemos introducido las operaciones de gestion de los hijos en la gui que sera la raiz de la jerarquia de clases, esto nos ofrece transparencia para tratar a todos los componentes de manera uniforme, pero supone que tendremos problemas al intentar añadir o eliminar guis cuando no existan guis hijas. La delegacion de peticiones entre guis supone establecer conexiones entre guis de manera recursiva, de forma que se pueda alcanzar cualquier componente desde cualquier otro en cualquier otra

gui.

En el diseño aparece tambien una lista de guis hijas que puede incurrir en una penalizacion de espacio para cada gui hija no compuesta. Mas aun, para implementar un mecanismo de comunicacion, se deberia gestionar una lista de observadores de cada gui o en su defecto, establecer una referencia a la gui padre para poder generar una estructura en arbol de relaciones entre guis, para poder recorrerla cuando se quisiera establecer comunicacion entre las distintas guis. Aunque una comunicacion en ambos sentidos, donde cualquier gui pudiese comunicarse con cualquier otra supondria realizar continuas busquedas en anchura o en profundidad a lo largo o ancho de toda la estructura. Aqui intervendria la aplicacion del patron mediador implementado como singleton, que podria realizar la tarea de almacenar una unica lista de observadores donde se incluirian todas las guis que participan en la gui general y utilizarla para establecer la comunicacion. Esta version sera la que tomemos a la hora de realizar nuestra implementacion.

2.5. Otros patrones software.

Viendo mas de cerca los patrones que existen, podemos observar que cubren varios rangos de escala y abstraccion en todo el ambito del desarrollo de software. Algunos patrones ayudan a estructurar un sistema de software en subsistemas, otros cubren el refinamiento de subsistemas y componentes, o las relaciones entre ellos, otros ayudan a implementar aspectos de diseño particulares en un lenguaje de programacion especifico.

Hay una enorme variedad de patrones que intervienen en casi todas las facetas del ciclo de vida del software, desde el analisis de requisitos, hasta la fase de pruebas o el despliegue, pasando por la fase de implementacion donde existen una amplia variedad de patrones especificos de un dominio concreto o de interaccion entre componentes.

Ofrecemos ahora una panoramica general de la enorme variedad de patrones existentes aplicados al desarrollo del software en cualquier dominio. La ofrecemos como una muestra de la amplia extension que se ha venido haciendo del concepto de patron convirtiendose en un verdadero lenguaje de patrones con el que intentar estructurar los procesos del software dentro de una metodologia que consiga agilizar su desarrollo, teniendo siempre en cuenta la experiencia adquirida y la abstraccion alcanzada en muchos de esos procesos:

- Patrones de analisis [Fowler, 1996]:
  - Party
  - Organization hierarchies.
  - Organization structure
  - Accountability

        - Accoutability Knowledge level.
        - Party Type Generalizations
        - Hierarchic Accountability
        - Operating scopes.
        - Post
- Patrones de integracion de aplicaciones [Frantz and Corchuelo, 2007]:
    - Channels:
        - Point to point
        - Guaranted Delivery
        - Datatype
        - Invalid Message
    - Routers:
        - Content-Based router
        - Recipient-List
        - Message filter
    - Message transformations:
        - Message Trasnlator
        - Canonical Data Model
        - Content Enricher
        - Content Filter
    - Adapter Block:
        - Polling consumer
        - Messaging mapper
        - Code Block
- Patrones de interfaz de usuario [UIP, 2022]:
    - User Interface design patterns (UI Web):
        - Getting input:
            - Forms: Password Strength, Meter, Structured Format, Rule Builder, Keyboard, Shortcuts, Captcha, Drag and Drop, Inplace Editor, Preview, Expandable Input, Autosave, Input Prompt, Good Defaults, Fill in the Blanks, WYSIWYG, Input Feedback, Calendar Picker, Undo, Settings, Morphing, Controls, Fogiving Format.

        - ◇ Explaining the process: Wizard, Steps Left, Completeness meter, Inline Help Box
        - ◇ Community driven: Rate Content, Wiki, Vote to Promote, Flagging & Reporting, Pay to Promote.
    - Navigation:
        - ◇ Tabs: Navigation Tabs, Module Tabs.
        - ◇ Jumping in hierarchy: Breadcrumbs, Fat Footer, Notifications, Modal, Home Link, Shortcut, Dropdown.
        - ◇ Menus: Vertical Dropdown, Menu, Accordion Menu, Accordion Menu, Horizontal, Dropdown Menu
        - ◇ Content: Adaptable view, Article List, Pagination, Cards, Carousel, Progressive, Disclosure, Continuous, Scrolling, Archive, Event Calendar, Thumbnail, Favorites, Tagging, Categorization, Tag Cloud.
        - ◇ Gestures: Pull to refresh
    - Dealing with Data:
        - ◇ Tables: Alternating Row, Colors, Sort By Column, Table Filter.
        - ◇ Formatting data: Frequently Asked, Questions(FAQ), Dashboard, CopyBox
        - ◇ Images: Gallery, SlideShow, Image Zoom
        - ◇ Search: Autocomplete, Search Filters.
    - Onboarding:
        - ◇ Guidance: Coachmarks, Playthrough, Inline Hints, Walkthrough, Blank Slate, Guided Tour
        - ◇ Registration: Lazy Registration, Paywall, Account, Registration
    - Social:
        - ◇ Reputation: Collectible, Achievements, Testimonials, Leaderboard.
        - ◇ Social Interactions: Activity Stream, Auot-sharing, Friend List, Reaction, Chat, Follow, Invite Friends, Friend.
    - Miscellaneous:
        - ◇ Shopping: Product page, Coupon, Shopping Cart, Pricing table.
        - ◇ Increasing frequency: Tip A Friend.
- Persuasive Design Patterns (UX Web):
    - Cognition:
        - ◇ Loss Aversion: Status-Quo Bias, Optimism Bias, Framing, Decoy Effect, IKEA Effect, Loss Aversion, Endowment Effect, Sunk Cost Effect, Negativity Bias.

        - ◇ Other cognitive blases: Illusion of control, Set Completion, Present Bias, Delay discounting, Need for closure, Curiosity, Value attribution, Priming Effect, Peak-end rule, Choice Closure, Cashless Effect, Inaction Inertia, Effect, Temptation, Bundling.
        - ◇ Scarcity; Limited choice, Scarcity, Limited duration.
    - Game mechanics:
        - ◇ Gameplay design: Appropriate challenge, Storytelling, Intentional gaps, Levels, Periodic Events, Investment Loops, Hedonic, Adaptation, Self-Monitoring.
        - ◇ Fundamentals of rewards: Variable rewards, Fixed rewards, Shaping.
        - ◇ Gameplay rewards: Goal-Gradien Effect, Privileges, Praise, Unlock features, Delighters, Achievements, Appointment, Dynamic, Prolonged play.
    - Perception and memory:
        - ◇ Attention: Tunnelling, REduction, Isolation Effect, Picture Superiority Effect, Zeigarnik Effect.
        - ◇ Comprehension: Chunking, Recognition over, Recall, Anchoring, Serial Positioning Effect, Pattern recognition, Conceptual metaphor, Sequencing.
    - Feedback:
        - ◇ Timing: Kairos, Feedback loops, Tailoring, Trigger, Simulation, Fresh Start Effect.
    - Social:
        - ◇ Social blases: Authority, Liking, Role playing, Self-Expression, Reciprocation, Social proof, Positive mimicry, Cognitive Dissonance, Commitment & consistency, Reputation, Halo Effect, Nostalgia Effect, Competition, Autonomy, Retallation, Status, Noble Edge Effect.

- Patrones de pruebas [UTe, 2022]:
    - Patrones de aprobado/reprobado
        - Patron de prueba simple.
        - Patron de ruta de codigo
        - Patron de rango de parametros.
    - Patrones de prueba controlados por datos:
        - Patron de datos de prueba simple.
        - Patron de prueba de transformacion de datos.
    - Patrones de transacciones de datos:
        - Patron simple-data-I/O

        - Patron de restriccion de datos.
        - Patron de revertir.
    - Patrones de gestion de colecciones:
        - Patron de orden de coleccion.
        - Patron de enumeracion.
        - Patron de resticcion de coleccion.
        - Patron de indexacion de la coleccion.
    - Patrones de rendimiento:
        - Patron de prueba de rendimiento.
    - Patrones de proceso:
        - Patron de secuencia de proceso.
        - Patron de estado de proceso.
        - Patron de regla de proceso.
    - Patrones de simulacion:
        - Patron de simulacro de objeto.
        - Patron de simulacion de servicio.
        - Patron de simulacion de error de bit.
        - Patron de simulacion de componentes.
    - Patrones de subprocesamiento multiple:
        - El patron señalado.
        - Patron de resolucion de interbloqueo.
    - Patrones de prueba de estres:
        - Patron Bulk-Data-Stress-Test.
        - Patron de prueba de estres de recursos.tipificacion
        - Patron de prueba de carga.
    - Patrones de capas de presentacion:
        - Patron de prueba de estado de vista.
        - Patron de prueba de modelo de estados
- Patrones de usabilidad [Moreno and Sánchez-Segura, 2003]:
    - Different languages
    - Different access methods

- Alert
- Status indication
- Shortcuts (key and tasks)
- Form/field validation
- Undo
- Context-sensitive help
- Wizard
- Standard help
- Tour
- Workflow model
- History logging
- Provision of views
- User profile
- Cancel
- Multi-tasking
- Commands aggregation
- Action for multiple objects
- Reuse information

- Patrones arquitectonicos [Suárez and Gutiérrez, 2016]:
  - Patrones de modulos:
    - Patron de n-capas
  - Patrones de componentes y conectores:
    - Patron broker
    - Patron modelo vista controlador
    - Patron tuberias y filtros
    - Patron cliente-servidor
    - Patron punto a punto
    - Patron Arquitectura Orientada a Servicios (SOA)
    - Patron Publish-Suscribe
    - Patron Shared Data
  - Patrones de asignacion:

- Patron Map-reduce
- Patron multi niveles

Chapter 3

Analisis

En este capitulo se realizara una descripcion de los objetivos del proyecto, estableceremos la distincion de roles entre los diferentes actores con los que la libreria tendra que interactuar, haremos un analisis de los requisitos que la libreria debe cumplir para implementar el patron de reusabilidad y finalmente realizaremos un esbozo de los casos de uso que podremos encontrar a la hora de utilizar la biblioteca en los distintos entornos de desarrollo.

3.1. Analisis previo.

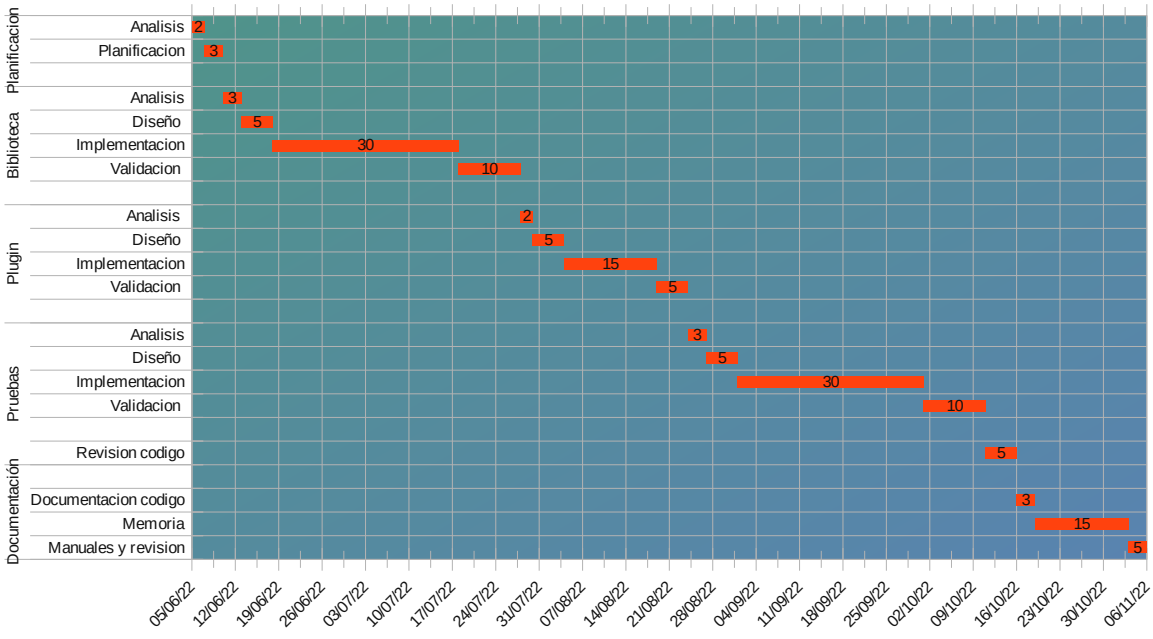
Todo proyecto software precisa de una analisis previo en relacion al tiempo que se espera invertir en la realizacion de dicho proyecto y los diversos costes de ingenieria, suministros, licencias y otros gastos derivados del desarrollo del producto informatico. En nuestro caso hemos elaborado una diagrama con la programacion temporal estimada para la ejecucion del proyecto, y un calculo estimado de los costes que supondria dicho desarrollo.

3.1.1. Planificacion temporal.

La planificacion temporal estimada se ha realizado en funcion de los distintos procesos de analisis, diseño, implementacion y validacion de cada una de las partes en que se divide la ejecucion del proyecto. En el diagrama que ofrecemos a continuacion se pueden observar los costes temporales estimados de cada una de las fases. En el diagrama se refleja la estimacion de coste temporal de cada proceso en cada fase, aunque su ejecucion se haya realizado en muchos casos de manera paralela y siguiendo los metodos iterativos del proceso unificado de desarrollo de software. La estimacion de cada una de las fases se ha realizado teniendo en consideracion margenes de posibles retrasos en el desarrollo de cada proceso.

CALENDARIO PLANIFICACION

Fase	Proceso	Fecha inicio	Duracion	Fecha fin
Planificacion	Analisis	05/06/22	2	07/06/22
	Planificacion	07/06/22	3	10/06/22
Biblioteca	Analisis	10/06/22	3	13/06/22
	Diseño	13/06/22	5	18/06/22
	Implementacion	18/06/22	30	18/07/22
	Validacion	18/07/22	10	28/07/22
Plugin	Analisis	28/07/22	2	30/07/22
	Diseño	30/07/22	5	04/08/22
	Implementacion	04/08/22	15	19/08/22
	Validacion	19/08/22	5	24/08/22
Pruebas	Analisis	24/08/22	3	27/08/22
	Diseño	27/08/22	5	01/09/22
	Implementacion	01/09/22	30	01/10/22
	Validacion	01/10/22	10	11/10/22
	Revision codigo	11/10/22	5	16/10/22
Documentación	Documentacion codigo	16/10/22	3	19/10/22
	Memoria	19/10/22	15	03/11/22
	Manuales y revision	03/11/22	5	08/11/22



Como se puede observar en el diagrama, la estimacion de coste temporal del proyecto es de 22 semanas, con una media por persona de 40 horas semanales, da un resultado estimado de 880 horas.

3.1.2. Analisis de costes.

En esta parte del analisis previos realizaremos un analisis de los costes

3.1.2.1. Costes de ingenieria.

En base a la estimacion del tiempo necesario para la ejecucion del proyecto, podemos realizar una valoracion inicial de los costes de ingenieria que supondria su elaboracion.

Tomando como referencia el salario medio de un ingeniero de software, concretamente 392 salarios de programador java, segun datos de [Tal, 2022] , el sueldo anual seria de 31500 €. De esta forma, el coste por hora del proyecto seria:

$$\underline{31500\ €\ /12\text{MESES} = 2625\ €}$$

$$\underline{2625\ €/\ 4\ \text{SEMANAS} = 656.25\ €}$$

$$\underline{656.25\ €/\ 40\ \text{HORAS} = 16.15\ €/\text{HORA}}.$$

La duracion estimada total del proyecto es de 22 semanas con una dedicacion media de 40 horas semanales, con lo que el coste salarial aproximado quedaria en:

$$\underline{880\ \text{HORAS X}\ 16.15\ €/\text{HORA} = 14212\ €}$$

Con lo que el coste de ingenieria total seria de 14212 €.

3.1.2.2. Costes de material.

Los costes de amortizacion del material utilizado en el desarrollo del proyecto se van a limitar al hardware empleado. En nuestro caso, se trata de un PC especificado en 5.1.1 cuyo coste de adquisicion fue de 1150 €. De esta manera, aplicando el coeficiente de amortizacion lineal maximo anual obtenemos:

$$\underline{1150\ €\ \text{X}\ 0.25 = 287.5\ €/\text{AÑO}}$$

con lo que el coste de amortizacion para el desarrollo del proyecto seria de:

$$287.5\ €/\text{AÑO}\ /\ 12\ \text{MESES} = 23.96\ €/\text{MES}$$

$$23.96\ €/\text{MES}\ /\ 4\ \text{SEMANAS} = 5.99\ €/\text{SEMANA}$$

$$5.99\ €/\text{SEMANA}\ \text{X}\ 22\ \text{SEMANAS} = 131.78\ €$$

Por lo tanto el coste de material del proyecto seria de 132 € aproximadamente.

3.1.2.3. Costes de suministros.

Se introduce una cantidad simbolica mensual de 5€ al mes como coste de suministros que pueden ser gastos de suministro electrico, de proveedores de internet, etc..

$$5€\ \text{X}\ 5.5\ \text{MESES} = 27.5\ €$$

3.1.2.4. Costes de licencia.

Los posibles costes sobre licencias de software que se podria aplicar queda reducido en nuestro caso a $0€$ porque todo el software que se utilizara en el desarrollo es de codigo libre bajo licencia GPL 3.0.

3.1.2.5. Factor multiplicativo

Respecto a otros posibles gastos derivados del desarrollo del proyecto, se aplica un incremento sobre la estimacion total de coste. Este factor multiplicativo aplicado como incremento de costes sera de un 10%, e incluiria otro tipo de gastos mas generales.

De esta manera, sumando todos los gastos obtenidos:

$$14212€ + 131.78€ + 27.5€ + 0€ = 14371.28€$$

Al que aplicandole el factor multiplicativo obtenemos:

$$14371.28€\ \text{X}\ 1.10\% = 15808.40€$$

Que seria la estimacion total de costes para el presupuesto inicial.

3.2. Analisis del proyecto.

Una vez realizado el analisis previo al desarrollo del proyecto, vamos a realizar una descripcion de los objetivos que queremos alcanzar con este proyecto y concretaremos en las diversas funcionalidades que queremos obtener en base al analisis de requisitos y de casos de uso que elaboremos.

3.2.1. Descripcion del proyecto

El objetivo principal de este proyecto es realizar una implementacion en lenguaje Java del *patron de reusabilidad de guis* esbozado en el capitulo anterior. Dicha implementacion se estructurara en forma de una biblioteca de clases software que proporcione la funcionalidad necesaria para poder hacer efectivo el patron en el entorno del desarrollo de la interfaz grafica de una aplicacion que desarrollaremos como programa de pruebas.

3.2.2. La distincion de roles: Actores y funcionalidad.

Para comenzar a esbozar la funcionalidad que debemos conseguir para implementar el patron, debemos tener en cuenta para quien esta destinada la biblioteca que queremos desarrollar.

A pesar de que el objetivo de nuestra biblioteca es basicamente facilitarle el trabajo a los desarrolladores, el destinatario final de cualquier interfaz grafica, la persona ultima para la que se diseña la gui es siempre el *Usuario* de la *Aplicacion*, que es quien se relaciona directamente con la interfaz e interactua con ella. Para este usuario, no debe haber ninguna diferencia entre las interfaces diseñadas con las guis reusables y las guis normales que utiliza en cualquier otro ambito o dominio. Es decir, la aplicacion del proceso de reutilizacion de guis que se elabore, ha de ser completamente transparente para el usuario final.

Las bibliotecas de software son en realidad codigo reutilizable que se puede introducir en un contexto de desarrollo para agilizar la codificacion, por lo tanto, los que van a utilizar realmente la implementacion del patron que vamos a elaborar seran los desarrolladores de software. En nuestro caso se trata de un contexto de desarrollo de guis, con lo que debemos tener en cuenta que quien va a utilizar con mas asiduidad el codigo de la libreria va a ser el encargado de diseñar y desarrollar el codigo de la interfaz grafica: El *Diseñador de Guis* (DG).

Este es uno de los roles principales que se distinguen dentro del patron *Modelo Vista Controlador*, concretamente el que se ocupa de la parte de la *Vista* segun dicho paradigma. El sera el encargado del desarrollo y diseño de toda la parte visual de la interfaz grafica y sera quien maneje mas intensivamente las clases de la libreria.

Por otra parte, los desarrolladores de software disponen a dia de hoy de numerosos entornos de desarrollo que favorecen y facilitan la tarea de la programacion, en nuestro caso, damos por supuesto que el diseñador va a utilizar un framework de desarrollo de interfaces graficas de usuario. Es por eso que uno de los requisitos fundamentales a la hora de implementar la biblioteca es que se puedan diseñar las guis reusables en el entorno de un editor grafico de guis, de tal forma que el Diseñador de Guis pueda hacer uso de ellas a traves de dicho entorno de una manera transparente, sin que exista distincion entre las guis reusables y cualquier otro widget de edicion de guis . En nuestro caso el entorno de desarrollo escogido es *NetBeans* (NB).

De esta forma uno de los actores principales que haran uso de la libreria sera el Diseñador de Guis a traves de NetBeans, que sera quien acoja las clases de la biblioteca para su uso. Para conseguir

esto, sera necesario elaborar un plugin para integrarlas en el IDE de manera que el Diseñador de Guis pueda disponer de los distintos tipos de dialogos que ofrece la libreria en la vista de diseño del entorno, y lo haga del mismo modo que dispone del resto de componentes de diseño.

Otro de los roles dentro del esquema MVC es el *Desarrollador* o *Programador de Aplicaciones* (PA) que centra su tarea en las partes del *Modelo* y el *Controlador*. El Programador de Aplicaciones es quien va a dotar de inteligencia a la *Vista,* dando respuesta a las acciones del usuario recopiladas a traves la interfaz grafica, manejando los datos que introduzca, haciendo que la interfaz pueda interactuar con el usuario y no sea solo un cascaron vacio de ventanas, paneles y widgets de edicion.

El Programador de Aplicaciones sera otro de los actores que utilizaran la biblioteca para poder integrar las guis reusables en la aplicacion, añadiendo dialogos hijos a dialogos padres y manejando tanto las guis que solicite directamente a la libreria, como los que le proporcione ya elaborados el Diseñador de Guis. Para ello, la libreria debera disponer de una interfaz publica lo mas reducida posible, para que se le facilite al Programador de Aplicaciones poder realizar estas funciones sin llegar comprometer el codigo de la implementacion.

3.2.3. Especificacion de actores.

En base al analisis anterior, podemos extraer y especificar los principales actores que intervendran en el modelo del dominio y en los casos de uso de la biblioteca que queremos elaborar. [1]

Los actores seleccionados son: Usuario A.1, Aplicacion A.2, Programador de Aplicaciones A.3, Diseñador de Guis A.4 y NetBeans A.5. Cada uno de ellos interactuan, de una manera mas o menos directa con las clases que desarrollaremos e implementaremos en la biblioteca. En el caso de Usuario, Aplicacion, cuya relacion con la libreria es mas indirecta, se han añadido como elementos que intervienen en la especificacion de requisitos y para el modelaje del dominio. En el caso de NetBeans, al tener que integrar la biblioteca en el entorno, se considera como parte activa y actor, ya que sera a traves de la vista de diseño del IDE como el diseñador de guis interactuara y se relacionara con las clases de la libreria .

3.2.4. Especificacion de requisitos.

En este apartado vamos a ir desgranando el conjunto de requisitos esbozados en el analisis anterior y que debemos tener en cuenta a la hora de realizar la implementacion del patron de reusabilidad.

3.2.4.1. Requisitos previos: RP

Como el problema de la reutilizacion de guis y la implementacion del patron ya fue afrontado por el departamento de IA de la ETSII de la UNED, a la hora de plantear el desarrollo del proyecto se

[1] El desarrollo de las especificaciones de cada uno de los actores se ha delegado al Anexo A de la memoria.

determinaron una serie de requisitos, que denominaremos requisitos previos, y que especifican una serie de requerimientos muy concretos a la hora de realizar la implementacion. Algunos de ellos son:

- los distintos tipos de guis seran: de navegacion simple, de navegacion mediante pestañas y de navegacion mediante vista de arbol.
- el numero de tipos de dialogos a implementar se limita a tres pero cabria la opcion de extenderlo a mas tipos utilizando algun mecanismo de facil implantacion.
- el nivel de anidacion del tipo simple sera de un maximo de dos. Este valor se tomo para estar seguros de que la gui no se saldria de la pantalla.
- debe existir un mecanismos de comunicacion entre guis para que se puedan establecer conexiones y comunicacion entre los diversos componentes que se introduzcan en las diferentes guis.
- se limitara al maximo la interfaz publica de la libreria: se debe proteger lo mas posible el funcionamiento interno de las clases de la biblioteca.
- se debe poder diseñar las guis en un editor grafico de guis.
- el proyecto generado sera de codigo abierto bajo licencia GPL.

Estos requisitos se han incorporado tanto a los requisitos funcionales como a los no funcionales del analisis y se identificaran durante su exposicion.

3.2.4.2. Requisitos funcionales: RF.

1. La libreria debe disponer de metodos con los que se puedan añadir unos dialogos a otros.[CU1 B.2, CU4 B.5]
2. La libreria debe disponer de metodos con los que se puedan añadir listas de dialogos a un dialogo padre.[CU1 B.2, CU4 B.5]
3. Se deben poder generar distintos tipos de guis. (En nuestro caso lo limitaremos a tres: simples, de navegacion en pestañas y de navegacion en arbol). [RP 3.2.4.1, CU3 B.4]
4. Cualquiera de las guis deben poderse anidar recursivamente.[CU1 B.2, CU4 B.5]
5. El numero de guis anidables teoricamente podra ser ilimitado. [RP 3.2.4.1]
6. Se limitara la anidacion en los dialogos de navegacion simple a dos. [RP 3.2.4.1]
7. La biblioteca se integrara en un entorno de desarrollo y diseño de interfaces graficas: NB. [CU5 B.6]

8. Las guis reusables deben poder utilizarse como un componente mas en la elaboracion de la interfaz grafica.[CU5 B.6]

9. La aceptacion de la edicion de la gui debe ser completamente transparente para el usuario de la aplicacion.[CU0 B.1]

10. El DG debe poder acceder y tratar las clases de la libreria de manera transparente, como si se tratara de las guis normales de la libreria grafica.[CU3 B.4, CU5 B.6]

11. Los dialogos deben aparecer en la paleta de componentes y en el asistente de plantillas de NB.[CU5 B.6]

12. El PA solo podra tener acceso a la interfaz publica de la libreria. [RP 3.2.4.1]

13. Los componentes de las guis deben tener un comportamiento adecuado a la hora de su redimensionamiento y recolocacion por acciones del usuario.[CU0 B.1]

14. Debe crearse un mecanismo de creacion de guis del tipo requerido como interfaz para el PA. [CU3 B.4]

15. Las guis solo podran crearse mediante este mecanismo y a traves del editor de guis (NB). [CU3 B.4]

16. Los componentes podran transmitirse mensajes entre ellos a traves de las distintas guis mediante un mecanismo de comunicacion. [RP 3.2.4.1]

17. La libreria puede forzar la implementacion de metodos para la aceptacion y validacion de datos de los widgets.[CU6 B.7]

18. El DG debe facilitar al PA los mecanismos necesarios para inicializar e integrar los dialogos que diseñe.[CU2 B.3]

19. El PA podra utilizar los dialogos diseñados por el DG como si se tratara en un dialogo normal.[CU2 B.3, CU3 B.4]

3.2.4.3. Requisitos no funcionales: RNF.

1. La biblioteca debera tener la libreria grafica Swing de Java como base para sus clases.

2. El entorno de desarrollo grafico sera el framework de NetBeans.

3. Se han de preservar las distribuciones de componentes a la hora de integrar las guis unas en otras.

4. Para proteger la funcionalidad de la libreria, su interfaz publica sera lo mas reducida posible. [RP 3.2.4.1]

5. Los tests de pruebas de la libreria se realizaran en base al diseño de una aplicacion totalmente funcional.

6. El proyecto generado sera de codigo abierto bajo licencia GPL.[RP 3.2.4.1]

3.2.5. Casos de Uso.

Una vez establecidos los distintos requisitos vinculados al desarrollo del proyecto, elaboraremos un diagrama 3.1de los posibles casos de uso que podemos establecer entre los diversos actores. Los distintos casos de uso se enfocan tanto en base a la funcionalidad que debe proporcionar la biblioteca, como respecto a las distintas relaciones que se establecen entre los actores y su repercusion en la implementacion.

Los principales actores con los que la biblioteca va a interactuar seran el diseñador de guis y el programador de aplicaciones. Los casos de uso que quedan enmarcados dentro del ambito de la biblioteca son aplicables a la distinta funcionalidad que debera gestionar la libreria para permitir el uso de las guis reusables a estos dos actores. Hemos distinguido tambien el entorno de diseño de guis como otro ambito distinto en el que la biblioteca tendra que desenvolverse.

El caso de uso de la edicion de guis, aunque no esta dentro del dominio de la libreria, afectan a su implementacion ya que las guis reusables tendran que manejarse con los eventos que active el usuario en la gui, asi como con todos los procesos de edicion, validacion y aceptacion que manejan todas las interfaces graficas. Algo similar sucede con el de pedir dialogos diseñados ya que, se trata de un requisito necesario para que el programador de aplicaciones pueda acceder a los dialogos diseñados por el diseñador de guis.

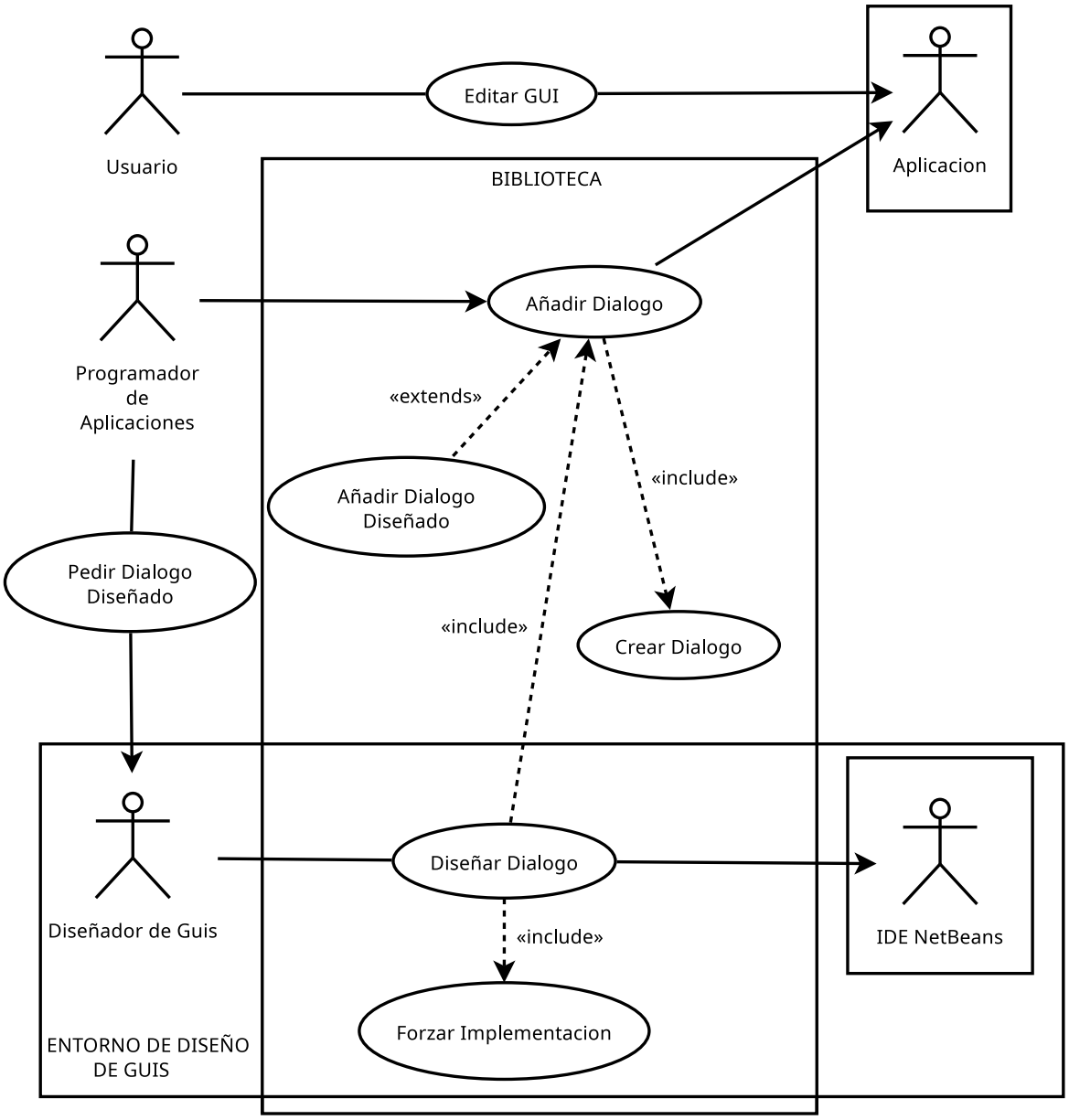


Figure 3.1: Diagrama de Casos de Uso

3.2.5.1. Especificacion de Casos de Uso.

El listado y la especificacion[2] de los casos de uso extraidos del diagrama anterior son:

CU0 Editar GUI. B.1

CU1 Añadir dialogo diseñado. B.2

CU2 Pedir dialogo diseñado. B.3

CU3 Crear dialogo. B.4

[2]La especificacion de los casos de uso se han delegado al Anexo B.

CU4 Añadir dialogo. B.5

CU5 Diseñar dialogo. B.6

CU6 Forzar implementacion. B.7

Chapter 4

Diseño

En este capitulo ofreceremos una vision paulatina del proceso de diseño de las clases de la libreria. Estableceremos en primer termino las clases conceptuales que utilizaremos en el diseño, elaboraremos un modelo del dominio en el que realizaremos el diseño. Propondemos un modelo de diseño en base a la aplicacion de diversos patrones que interactuan entre si y que utilizaremos a la hora de realizar la implementacion, y ofreceremos las diversas decisiones de diseño que se han tomado para llegar al modelo propuesto. Por ultimo, presentaremos el entorno de desarrollo integrado que utilizaremos para desarrollar la biblioteca y sobre el que modelaremos la estructura de un modulo con el que podamos integrar nuestra libreria en el IDE.

4.1. Modelando la biblioteca.

En esta seccion vamos a extraer del analisis realizado en el capitulo anterior los primeros esbozos de lo que sera el diagrama de clases que conformaran la biblioteca.

4.1.1. Artefactos del dominio: clases conceptuales.

Del analisis de los diferentes actores y requisitos y de la especificacion de los distintos casos de uso, podemos extraer un conjunto de clases conceptuales con las que podemos comenzar a modelar los elementos del dominio en el que se va a desenvolver la biblioteca.

Segun el analisis del capitulo anterior distinguimos las siguientes clases conceptuales:

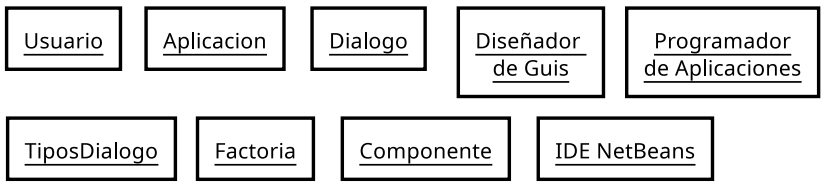


Figure 4.1: Clases conceptuales del dominio de la biblioteca

- USUARIO: es quien se encarga de interactuar con la biblioteca a traves de la interfaz grafica de la Aplicacion. Es quien realiza la edicion de la gui introduciendo los datos solicitados por el sistema y activando los distintos eventos que las guis reusables tendran que manejar.

- APLICACION: su funcion es gestionar el conjunto de datos y eventos generados por el usuario. El programador de aplicaciones le facilitara una interfaz grafica adecuada para establecer esa relacion con el usuario. En la elaboracion y el uso de esa interfaz grafica es donde se utilizaran las funcionalidades que proporcionen las clases de la biblioteca.

- DIALOGO: es el elemento central del desarrollo ya que son las piezas con las que el diseñador de guis y el programador de aplicaciones construiran la gui de aplicacion. Estas piezas una vez diseñadas tienen que poder manejarse con total libertad, utilizarlas individualmente, insertar unas en otras de forma recursiva y deben funcionar en todo momento como cualquier otro componente de la libreria grafica de Java. Conseguir reutilizar dialogos construidos previamente es el objetivo central del desarrollo de la biblioteca, asi que esta es una clase basica, ya que practicamente todos los otros elementos del modelo del dominio estan vinculados a ella de una manera mas o menos directa.

- DISEÑADOR DE GUIS: diseña los dialogos utilizando el IDE de NetBeans. En el proceso de diseño se incluyen la creacion y adicion de dialogos en otros dialogos de forma recursiva, asi como la creacion y adicion de componentes en esos dialogos. Tambien en ese proceso se incluye la implementacion de los metodos necesarios para poder articular los componentes de las guis reusables dentro de la gui de aplicacion.

- PROGRAMADOR DE APLICACIONES: es quien realizara la arquitectura para la gestion del conjunto de datos y eventos recaudados por la interfaz grafica. Por lo tanto, es quien va a integrar en la aplicacion las guis reusables que se hayan generado y las dotara de inteligencia. Para construir esa gui de aplicacion el programador de aplicaciones manejara tanto dialogos previamente diseñados por el diseñador de guis, como dialogos huecos con los que pueda envolver otros dialogos y componentes y los ira añadiendo a la gui. El objetivo es que el programador disponga de diferentes tipos de dialogos, y que pueda integrar unos en otros para conformar una gui de aplicacion especifica combinando diseños ya elaborados previamente.

- FACTORIA: esta clase conceptual se ha incluido por la necesidad de elaborar un mecanismo de creacion de dialogos que aislase el proceso de creacion de guis. Mediante esta factoria el programador podra acceder a los tres tipos de dialogos requeridos.

- TIPOSDIALOGO: hemos incluido este artefacto conceptual para representar los tipos de dialogo que se podran crear y manejar con la libreria. Representan en todo caso las distintas subclases de dialogo que la biblioteca tendra que articular. Este elemento se relaciona por un lado con la factoria, que precisa conocer el tipo de dialogo que debe crear para poder crearlo,

y por otro lado con los dialogos ya que estos deben seleccionar el tipo de dialogo que quieren ser.

- COMPONENTE: este elemento se introduce porque son partes integrantes de la interfaz grafica y los dialogos los utilizan para la composicion de guis reusables. Ellos mismos se pueden interpretar como componentes complejos, compuestos por amalgamas de otros dialogos y otros componentes.

- IDE NETBEANS: finalmente, introducimos el entorno de desarrollo en el diseño porque es un elemento activo en la interaccion de los desarrolladores con la libreria. La integracion de la libreria en el IDE permitira el acceso del diseñador a las clases de la biblioteca y facilitara el diseño con las guis reusables. El modulo a desarrollar para integrar la biblioteca en el entorno hace que este elemento sea necesario en el modelo del dominio.

4.1.2. Modelo del Dominio.

Una vez extraidos los distintos elementos conceptuales que vamos a utilizar, vamos a establecer las relaciones y asociaciones que se establecen entre ellos.

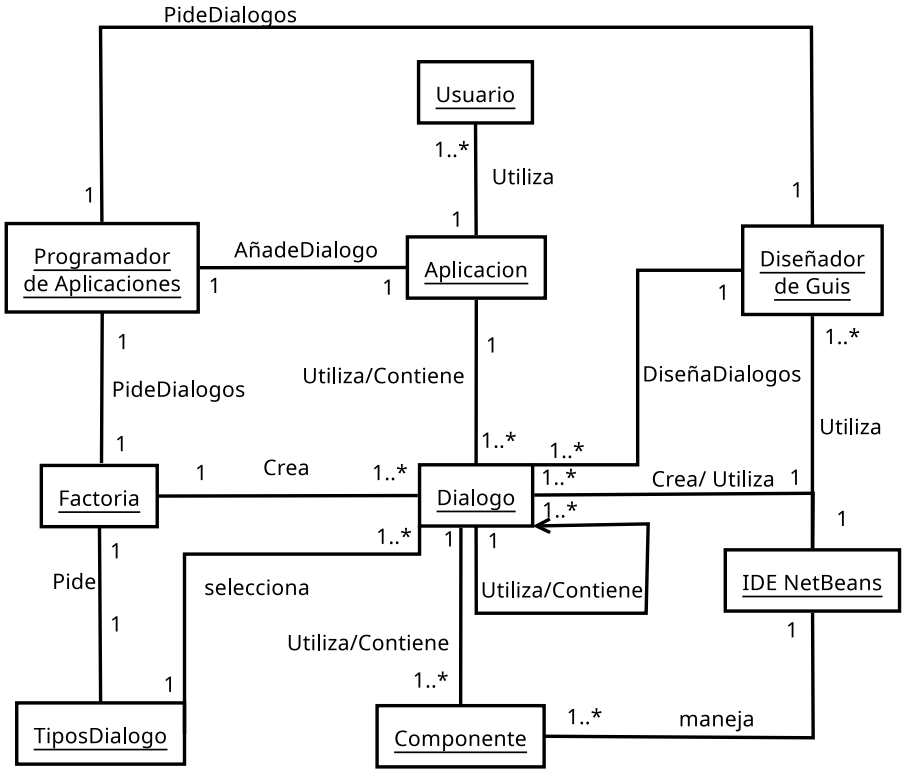


Figure 4.2: Modelo del dominio de la libreria.

En este diagrama del modelo del dominio podemos ver los vinculos que se generan entre las distintas clases. Se puede observar claramente como eje central del modelo la clase Dialogo. A

los objetos dialogo les hemos dado la capacidad de poder utilizar y contener otros dialogos, para plasmar asi la recursion necesaria para poder integrar unas guis en otras, parte esencial para conseguir articular la reusabilidad que buscamos.

Los creadores de dialogos seran por un lado la factoria, que debe saber el tipo de dialogo que el programador de aplicaciones quiere crear; por otro lado, el IDE que debe poder crear tambien distintos tipos de dialogos dentro del entorno grafico para que el diseñador de guis pueda realizar su tarea. Tanto la factoria como el IDE deben conocer que dialogo crear, uno mediante la insercion de algun parametro de tipo, el otro a traves de un mecanismo de seleccion en la implementacion del modulo.

La clase conceptual introducida con el nombre de TiposDialogo representa los tres tipos de dialogos que se exige en los requerimientos: simple, de navegacion por pestañas y en estructura de arbol. El creador debe tener constancia de ellos para su creacion, y el dialogo se creara en base a la seleccion de tipo que el diseñador o el programador de aplicaciones realicen a traves de el. El elemento dialogo no existe sin esa seleccion de tipo, con lo que podria plantearse la idea de concebir dialogo como una interfaz que se implemente en las clases determinadas por los distintos tipos de dialogo, o definirla como una clase que deja que sean sus subclases las que especifiquen los objetos que crea, en este caso un tipo distinto de dialogo.

La clase componente se ha introducido en el modelo por ser parte integrante tanto de los dialogos como de la mecanica de diseño del entorno grafico. Los dialogos son en realidad componentes complejos que combinan un puñado de otros componentes y conforman una gui completa y perfectamente funcional, posteriormente esa gui podra ser reutilizada e integrada en otras guis. En el modelo, dialogo y componente se han discriminado como conceptos en clases separadas ya que la biblioteca trabajara a nivel de guis no de componentes, aunque estos formen parte esencial de su composicion. Los componentes que se utilicen perteneceran a la libreria grafica Swing por lo que la clase dialogo tendra que heredar de una clase de esa libreria para poder acceder a los componentes graficos que precisa.

Por lo que respecta a los elementos usuario y aplicacion, su incorporacion al modelo es necesaria por ser ambos los destinatarios ultimos del proceso de desarrollo que van a realizar tanto el programador de aplicaciones como el diseñador de guis. Mucha de la funcionalidad interna de la biblioteca deriva de la necesidad de hacer transparente la edicion de las guis al usuario, con lo que la respuesta a eventos generados por este debe adaptarse en la aplicacion a los mecanismos implementados en la libreria para hacer las guis reusables.

4.1.3. Modelo de Diseño.

Teniendo en consideracion los requisitos y el analisis del dominio que hemos efectuado anteriormente, vamos a proceder a realizar el diseño de las clases que conformaran la base para la implementacion de la biblioteca. Para ello hemos aplicando distintos patrones de diseño a la hora de

elaborar el diagrama de clases.

El diseño de las clases de la libreria en base a la aplicacion de patrones es un proceso que se realiza automaticamente y de una forma casi instintiva, aqui hemos intentado desgranar en lo posible ese proceso para destacar la forma en que se conjuntan y aplican los distintos patrones a la hora de de implementar el patron de reusabilidad.

4.1.3.1. Guis compuestas.

La dinamica de reusabilidad de guis supone la composicion de dialogos que puedan anidarse unos en otros de tal forma que un dialogo diseñado previamente, pueda insertarse dentro de otro y ese proceso pueda realizarse recursivamente. Un grafico de estados que refleja el proceso seria el siguiente.

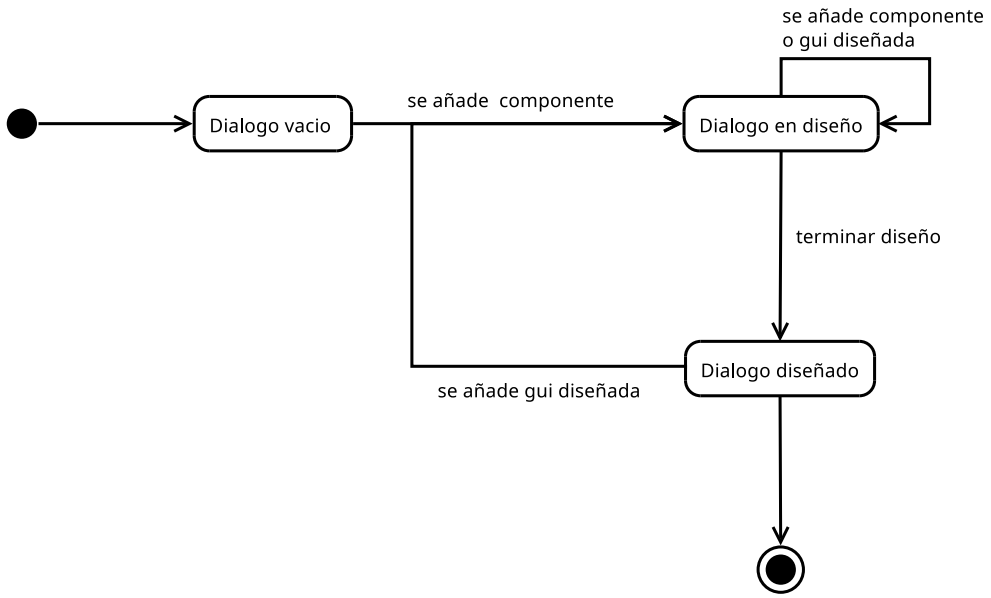


Figure 4.3: Grafico de estados de insercion de guis

Los dialogos seran entonces estructuras elaboradas con una serie de componentes y utilizadas directamente, o guis compuestas mediante la insercion de otras guis ya diseñadas, con su conjunto de componentes, sean estos componentes basicos de la libreria Swing o dialogos diseñados previamente, con lo que obtenemos una composicion recurrente. Esto se resume en que debemos poder tratar los dialogos como estructuras atomicas o estructuras compuestas, indistintamente. Y este es un problema que podemos gestionar con el el patron Composite.

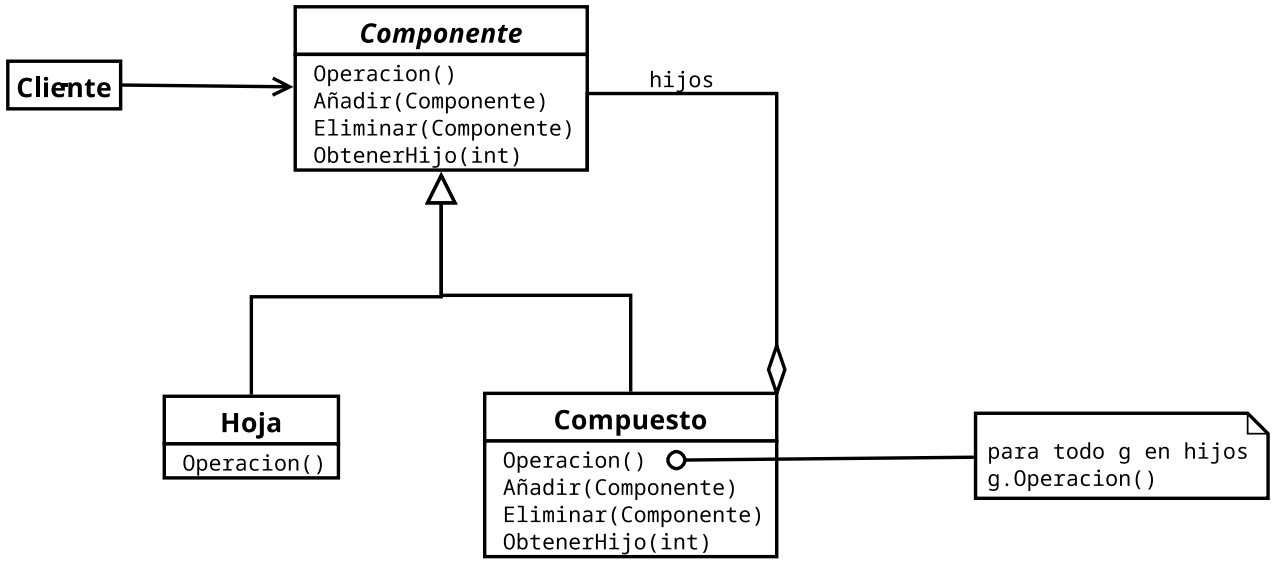


Figure 4.4: Patron Composite[Gamma et al., 1994]

Los elementos esenciales de este patron serian:

- *Componente:* declara la interfaz de los objetos de la composicion. Implementa el comportamiento comun a todas las clases. Declara mecanismos para acceder a los componentes hijos y gestionarlos.
- *Hoja:* representa los objetos basicos de la composicion y define su comportamiento. No tiene hijos.
- *Compuesto:* representa los objetos compuestos y define su comportamiento. Almacena componentes hijos e implementa las operaciones relacionadas con ellos.
- *Cliente:* manipula los objetos basicos y compuestos a traves de la interfaz Componente.

La clave de este patron es una clase abstracta, en nuestro caso se trataria de la clase Dialogo, que representaria tanto a los dialogos sencillos como a los dialogos que los contienen. Esta clase concentraria toda la funcionalidad comun a los distintos tipos de dialogo, asi como la que comparten todas las estructuras compuestas como todas las operaciones de gestion de hijos . En nuestro caso vamos a conjuntar en la clase Dialogo tanto la clase Componente como la Compuesto del esquema del patron, permitiendo asi que la clase Dialogo sea en realidad un conjunto de subdialogos definidos por los distintos tipos de gui implementados en subclases polimorficamente. Esto nos permitira construir dialogos formados por muchos subdialogos que se ensamblen de una manera flexible y sencilla, y podremos tratar a todos los componentes de una manera uniforme.[1]

Aplicando el Composite de esta manera, incurrimos en una serie de problematicas como es la inclusion en la clase base de la lista de hijos del elemento Compuesto, lo que puede hacer que resulte

[1]Una variacion similar, conjuntando Componente y Compuesto se implemento en la clase Vista original del Modelo/Vista/Controlador de Smalltalk[.Krasner and Pope, 1988]

redundante en las construcciones que no sean compuestas ya que estas nunca tendran hijos. En este caso interpretamos que las estructuras no tendran un numero tan elevado de hijos como para que el coste espacial no pueda llegar a asumirse. Tambien nos encontraremos con que la declaracion de las operaciones sobre los hijos no tienen mucho sentido aplicadas a los objetos no compuestos. En todo caso, tenemos que tener en cuenta que los dialogos no compuestos que se creen deben tener la posibilidad de convertirse en objeto compuesto, ya que en el proceso de reutilizacion de guis se le pueden agregar otros dialogos para generar una nueva interfaz, con lo que deben poder manejar las operaciones de los posibles hijos que se le adjudiquen. En nuestro caso y mediante esta aplicacion del patron, los objetos hoja siempre pueden disponer de las estructuras de datos y las operaciones necesarias para articular la composicion y poder ser asi reutilizados.

Por otra parte, para que la clase pueda integrarse en la libreria Swing de Java y disponer asi de todos los recursos graficos que esta biblioteca ofrece, es preciso que herede de alguna de sus clases. De esta forma el esquema del patron Composite aplicado a nuestra desarrollo tendria la siguiente estructura.

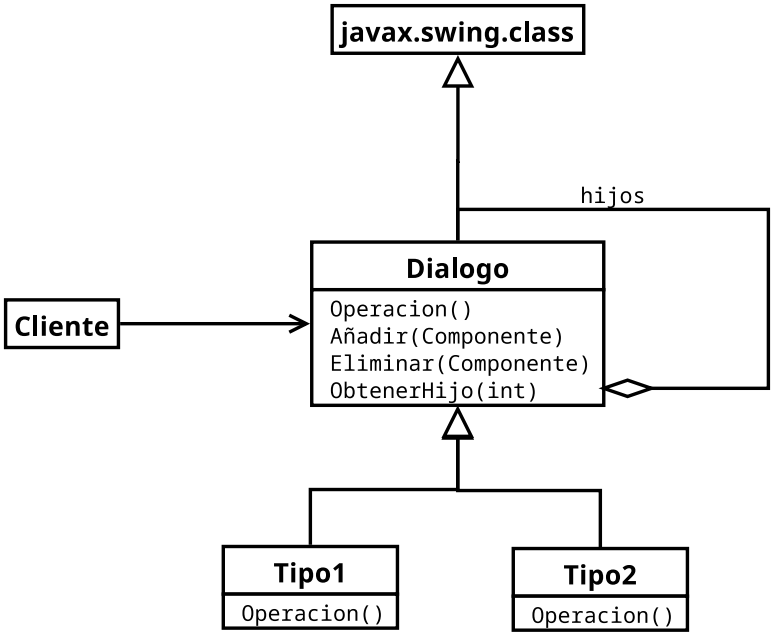


Figure 4.5: Esquema patron Composite aplicado.

El uso de este patron nos facilita cumplir con los requisitos de transparencia respecto al diseñador de guis [RF 10] y proporcina Variaciones Protegidas, de manera que no les afecte si el objeto con el que trabajan es un dialogo sencillo o compuesto.

4.1.3.2. Factoria, Controlador y Creador.

Teniendo en cuenta que para realizar su tarea el programador de aplicaciones puede solicitar a la libreria guis vacias, la responsabilidad de gestionar esas solicitudes debe recaer en una clase especifica.

La clase Factoria esta enfocada a controlar la fabricacion de instancias de la clase dialogo, por lo tanto el patron controlador la señala como responsable de gestionar un evento de sistema basico de la libreria como es la creacion de guis. Ademas, en la especificacion de requisitos nos piden implementar un mecanismo aislado de creacion de guis [RF14] para intentar proteger en lo posible el funcionamiento de la libreria [RNF 4] con lo que al aislar este mecanismo en la factoria, aplicamos tambien el patron de Variaciones Protegidas y Bajo Acoplamiento.

Por otra parte, hemos delegado en la factoria la responsabilidad de la creacion de instancias de los distintos tipos de gui, por lo tanto el patron GRASP Creador la señala como responsable, pero la clase se creo tambien para extraer un mecanismo aparte en el esquema del dominio, con lo que podemos enmarcarla dentro de un patron Fabricacion Pura, en concreto un Factory Method, y especificamente una Factoria simple, que se ciñe mas a lo que queremos representar, y utilizaremos una de sus variantes al pasarle al metodo de fabricacion el parametro de tipo de gui a crear.

Como se puede observar, a la hora de realizar el diseño se tienen en cuenta un conjunto de patrones que en realidad se encuentran sumamente relacionados. Aqui podriamos incluir tambien del patron de Indireccion, ya que la factoria seria la intermediaria entre la clase Dialogo y el cliente, que en este caso seria el Programador de Aplicaciones. Con esto cabe destacar que la interaccion de patrones es constante en el desarrollo del diseño, y en muchos casos la aplicacion de uno u otro es casi siempre una cuestion de la interpretacion que se haga del problema.

Un esquema basico de la secuencia de ordenes en el proceso de creacion de guis de la clase Factoria seria el siguiente.

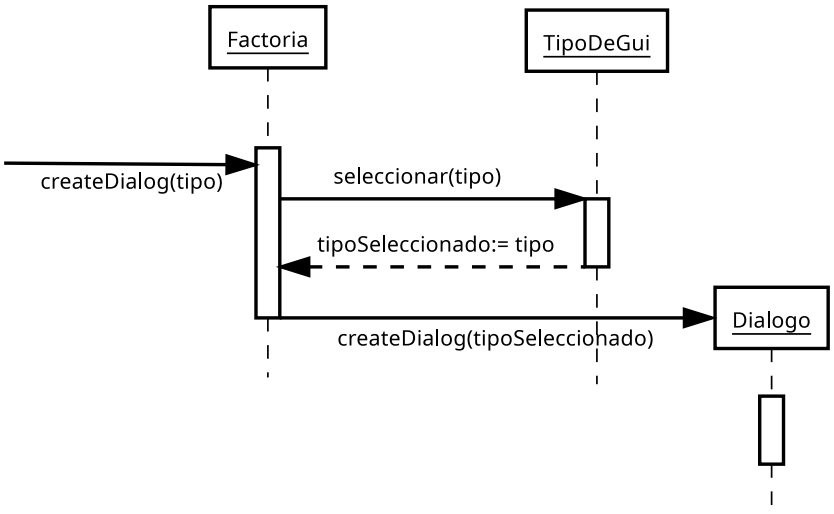


Figure 4.6: Diagrama de secuencia creacion de gui

Como vemos en el esquema, el parametro tipo selecciona en la clase TipoDeGui el tipo de dialogo a crear y posteriormente crea el dialogo adecuado. Toda esta secuencia de acciones se realiza internamente dentro del metodo de la factoria mediante Polimorfismo.

4.1.3.3. Polimorfismo de tipo.

Para ir definiendo las clases que vamos a necesitar para la implementacion, tenemos que desdoblar la clase conceptual TipoDeGui que habiamos creado, en las tres clases de guis que vamos a derivar.

Definiremos tres clases: Simple, de navegacion mediante Pestañas y de navegacion en Arbol, para especificar los distintos tipos de guis que va a gestionar la libreria. Estas clases son tipos de dialogo, sus comportamientos difieren en la forma de mostrar al usuario los componentes y de integrar las guis, de modo que asignaremos la responsabilidad de la variacion de comportamiento a cada uno de los tipos y los estructuraremos como subclases de la clase Dialogo. Mediante Polimorfismo conseguiremos la flexibilidad necesaria para obtener dialogos con unas caracteristicas particulares, determinadas en cada una de las subclases, y unas comunes que se incluiran en la clase base. Estas subclases actuaran como hijos dentro del patron Composite esbozado para la clase Dialogo, funcionaran como variantes de tipo pero no deben ser instanciables directamente, sus constructores deben ser metodos protegidos , la factoria se encargara de discriminar el tipo e instanciar los dialogos que precise el programador de aplicaciones.

Interpretando las subclases como variantes de comportamiento del dialogo, podemos reconocer la aplicacion del patron Strategy si pensamos que los tipos de guis son una familia de algoritmos que se implementan utilizando la interfaz proporcionada por la clase Dialogo. Podemos decir que cada subclase es una estrategia de diseño distinta que se aplica a traves de la clase base facilitando una eleccion de implementacion que obliga al desarrollador a conocer las distintas formas en las que puede implementarse el dialogo y como pedirlas a la factoria.

El acceso a los tres tipos de guis estara restringido a una visibilidad interna entre clases. Solo el diseñador de guis a traves del IDE, podra derivar clases de los distintos tipos de guis. Una vez integremos el modulo con la biblioteca en el entorno, podra diseñar dialogos nuevos heredando de la jerarquia de clases de la libreria.

Esta restriccion se impone aplicando el patron de Variaciones Protegidas, que es una premisa basica en todo el desarrollo de la biblioteca: preservar la funcionalidad de la libreria reduciendo su interfaz al minimo [RNF 4].

El esquema del diseño de las clases esenciales de la biblioteca quedaria asi:

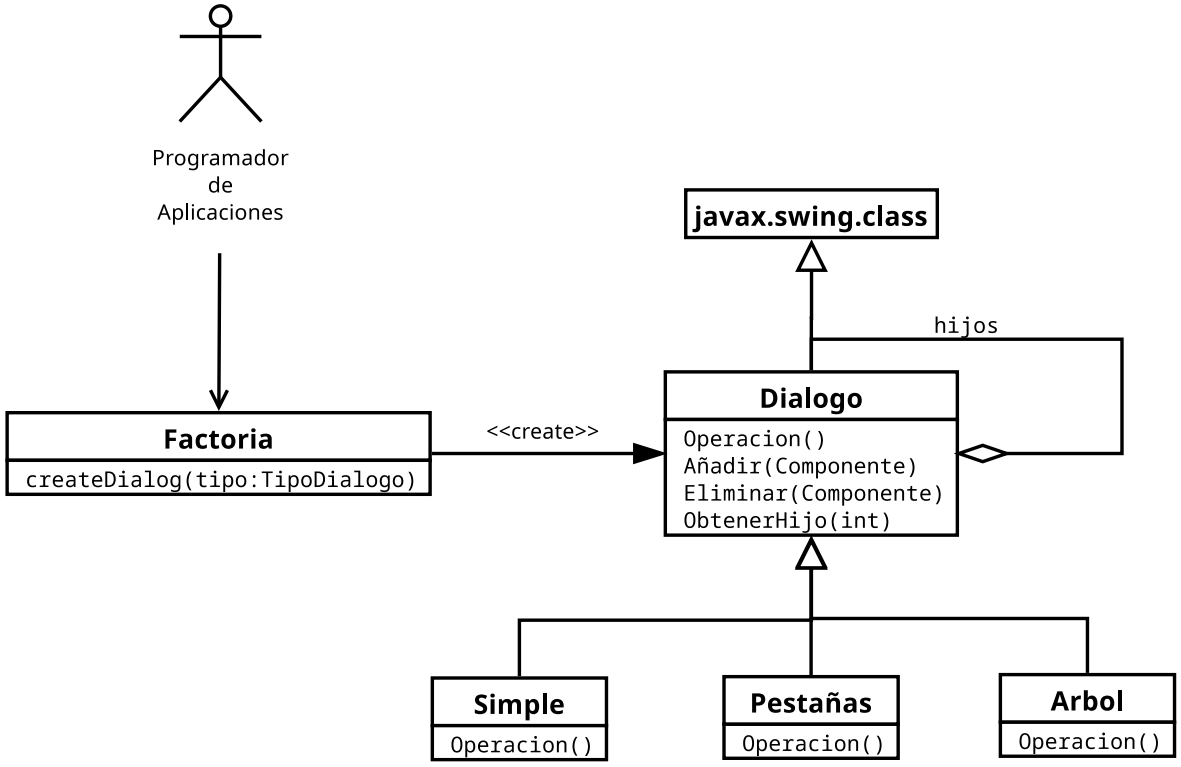


Figure 4.7: Esquema clases de diseño

4.1.3.4. Guis que se observan.

Uno de los requerimientos exigidos en el analisis de requisitos es la creacion de un mecanismo de paso de mensajes entre guis que permitan la comunicacion entre los distintos componentes de la gui [RF 16]. Para implementar este mecanismo hay acudir al patron Observer y aplicarlo entre las distintas guis reusables que se podran generar. Para ello hay que establecer el nivel al que debemos aplicar el patron, ya que en realidad no sabemos que componentes van a conformar cada una de las guis, ni entre cuales se va a establecer la comunicacion. Por lo tanto, el mecanismo debe construirse a nivel de dialogo. Las distintas guis deben ser emisoras y receptoras de mensajes, todas las guis se observaran unas a otras y se notificaran mediante broadcast cualquier cambio que consideren relevante. El componente en cuestion y el valor modificado se deberan pasar como parametros, ya que las distintas guis no saben ni de quien viene ni a quien va dirigido el mensaje, para de esa manera poder controlar y saber a quien encauzar el valor de cambio. En este sentido podemos entender que se aplica aqui el patron de Indireccion, haciendo que sean las propias guis las que encaminen las notificaciones de y hacia los componentes adecuados.

El diseñador de guis debera por lo tanto implementar en la gui que diseñe un metodo concreto que realice estas funciones de identificacion y encauzamiento al componente con el que quiera que se establezca la comunicacion.

4.1.3.5. Incorporando un mediador.

Durante el desarrollo del proyecto y por cuestiones de implementacion que se especifican mas adelante, se creyo oportuno incorporar al diseño una clase que funcionara de intermediaria, para articular la implementacion del paso de mensajes entre guis, asi como para centralizar el control y encapsular aun mas si cabe la funcionalidad de la clase base. Para su desarrollo se tomo como referencia el patron Mediator.

Esta clase funciona como gestora de los metodos y las estructuras de datos necesarias para implementar el patron observer entre las distintas guis, y se encarga de transmitir notificaciones de cambio a todos los componentes que estan escuchando a la espera de algun evento. El patron observer se aplica en la biblioteca a nivel de gui, con lo que el mecanismo de comunicacion se realiza entre guis y seran estas las que encaminen las notificaciones de y hacia los distintos componentes. A traves del mediador las guis envian y reciben las peticiones y las dirigen a sus objetivos sin necesidad de tener referencias entre ellas ya que delegan en el mediador esa responsabilidad, lo que es una forma de aplicar el patron de indireccion.

El mediador tambien funciona como centro de control de edicion ya que concentra toda la funcionalidad de la clase base en relacion a la aceptacion, validacion y manipulacion de datos que no se fuerza a implementar al diseñador de guis.

De esta manera, añadiriamos el mediador al esquema del diseño y quedaria de la siguiente forma:

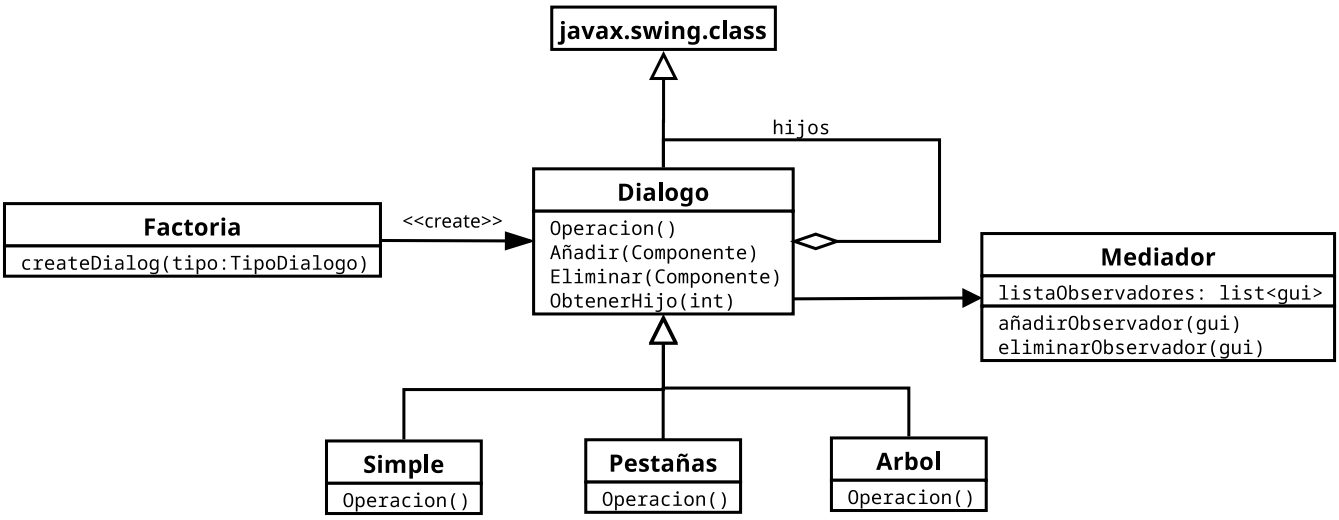


Figure 4.8: Diagrama de clases de diseño de la biblioteca.

4.2. Modelando el plugin para NetBeans.

Para modelar el tipo de extension que queremos desarrollar para incorporar nuestra biblioteca a la plataforma del IDE de NetBeans, debemos primero saber que elementos son los que necesitamos agregar y de que manera queremos hacerlo.

4.2.1. Que es NetBeans.

NetBeans es un entorno de desarrollo integrado (IDE) libre y de codigo abierto que permite desarrollar aplicaciones web, aplicaciones moviles y aplicaciones de escritorio. Admite el desarrollo de aplicaciones en diferentes tipos de lenguajes, incluidos Java, HTML5, PHP y C++. Brinda soporte para todo el ciclo de desarrollo completo de una aplicacion, desde la creacion del proyecto, hasta la depuracion, gestion de versiones, implementacion, documentacion y despliegue. El IDE, al estar escrito completamente en Java y utilizar la maquina virtual de Java, puede ejecutarse en cualquier sistema operativo: Windows, Linux, Mac OS X y otros.

Una de las caracteristicas mas destacables del entorno es que funciona sobre una plataforma de cliente enriquecido (Rich Client Platform). Este termino, en las arquitecturas cliente-servidor, se utiliza cuando el tratamiento de los datos se realiza principalmente en el lado del cliente, y son definidas como aplicaciones que son extensibles a traves de extensiones y modulos. De esta manera, en NetBeans disponemos de un entorno extensible al que se le pueden añadir caracteristicas, servicios o funciones especificas mediante modulos que se ejecutan desde el sistema principal e interactuan por medio de API´ s. Esta caracteristica nos servira para incorporar al entorno las funcionalidades que queremos incorporar a su editor grafico mediante el modulo con las clases de nuestra biblioteca.

4.2.2. Estructura de un modulo NetBeans.

La estructura de un modulo NebBeans es un archivo JAR formado por un conjunto de archivos entre los que se encuentran las clases con la funcionalidad y una serie de archivos que lo identifican como modulo en el IDE. Ademas un archivo XML de configuracion que es el primer archivo que lee el sistema de modulos, y que anuncia el modulo a la plataforma. Basicamente un modulo añade funcionalidad agregando entradas al sistema de archivos del sistema. Examina su archivo manifiesto y busca entradas que le digan que hacer con el contenido del JAR. Su estructura suele ser:

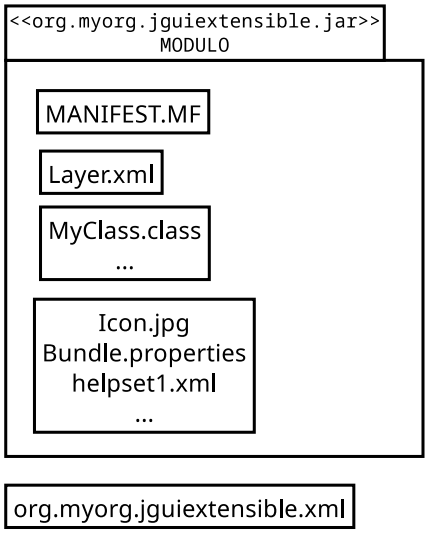


Figure 4.9: Arquitectura de un modulo NetBeans

La funcion de los distintos archivos es la siguiente:

MANIFEST.MF es una descripcion textual del modulo y de su entorno, y el primer archivo que el sistema de modules lee cuando carga un modulo. NetBeans lo reconocera como modulo si el archivo de manifiesto dispone del unico atributo obligatorio, el OpenIDE-Module. El archivo manifiesto suele contener ademas otro tipo de informacion:

- Identificacion del modulo.
- Descripcion del modulo
- Informaciones de versiones sobre el modulo.
- Informacion sobre las dependencias del modulo.
- Entradas de classpath requeridas por el modulo.
- Un paquete de localizacion.
- Secciones de manifest especificas de NetBeans
- etc...

XMLLayers los archivos XML de capa son usados para instalar datos y objetos en el sistema de archivos del sistema. Su uso principal es colocar clases Java en los ficheros del sistema que tienen funciones especiales como registro de servicios o funciones de menu.

Bundle.properties es un archivo de propiedades, donde se guardan distintos parametros de la configuracion del paquete en formato clave-valor que utilizara NetBeans para configurar el modulo en el entorno.

MyClass.class la clase o clases que formen parte de la extension y que ampliaran las funcionalidades del IDE integrandose en su sistema de archivos en tiempo de ejecucion.

Icon.jpg es un archivo o archivos de imagen con el que se representara el objeto que añadiremos al entorno, p.ej. icono de plantilla de archivo, un nuevo componente del menu, de la paleta de componentes, etc...

helpset1.html archivo en formato html que sirve de apoyo al artefacto que queramos integrar definiendo su funcionalidad, realizando una descripcion del modulo o del elemento del modulo a integrar, etc.

org.myorg.jguiextensible.xml archivo externo de configuracion con el que el modulo es declarado al sistema de modulos de la plataforma. El modulo se cargara de acuerdo a la informacion almacenada en este archivo de configuracion: nombre, version y la localizacion del modulo estan definidas en el archivo, asi como otras

Tanto en este caso como en el de los dos anterioes, el nombre del archivo no es preceptivo. Igualmente, esta es una estructura basica y elemental para la elaboracion del modulo, lo que no supone que su arquitectura no se pueda ampliar en funcion de la extension que se pretenda desarrollar.

4.2.3. Diseñando un modulo para NetBeans.

El principal objetivo de la elaboracion del modulo es que cualquier diseñador de guis pueda disponer de los mecanismos implementados en la biblioteca a traves del editor grafico del IDE. Para eso deberiamos conseguir que el modulo integrara nuestra biblioteca con el catalogo de librerias disponibles en el entorno de NetBeans, de manera que sus funcionalidades esten en todo momento disponibles para el desarrollador.

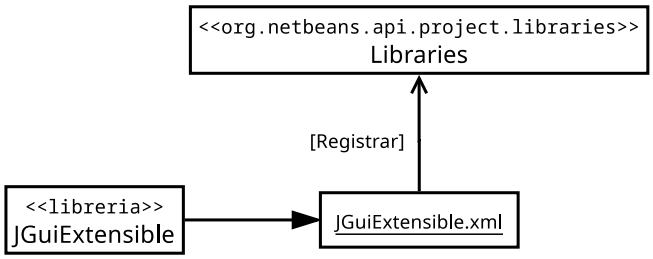


Figure 4.10: Esquema insercion libreria

Por otra parte, para disponer de las plantillas de los distintos tipos de gui en el asistente del IDE habra que crear un template de cada uno de los tipos de forma que cuando se llame a la plantilla se genere por una parte el panel de diseño donde añadir los componentes a la gui, por otra se genere el esquema de la clase con el codigo y los metodos necesarios para implementar una clase derivada de la gui seleccionada. Estos templates tendran que registrarse en la api de NetBeans para incorporarlos al asistente de creacion de archivos.

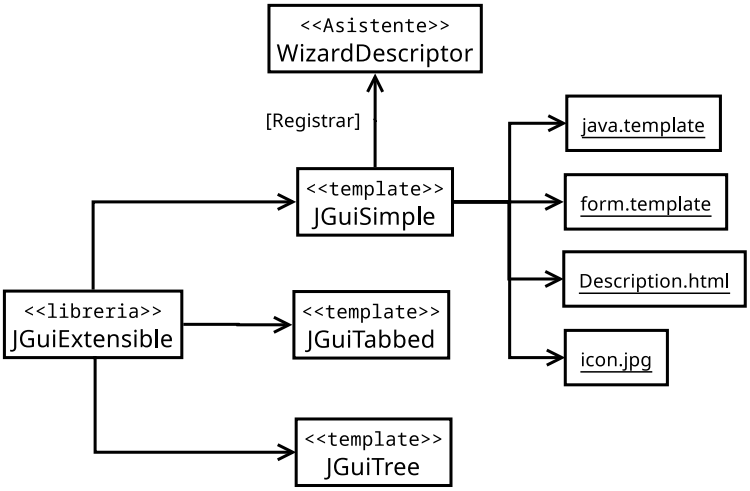


Figure 4.11: Esquema registro templates

Tambien se tendra que incorporar a la paleta de componentes del editor grafico los distintos tipos de gui, de manera que se cree una nueva seccion llamada JGuiExtensible donde dispongamos de los dialogos como items de paleta con los que trabajar en el panel de diseño. Igualmente, los distintos items de paleta generados tendran que registrarse en el fichero FormDesignerPalette del sistema de archivos del sistema para integrarlos en la paleta.

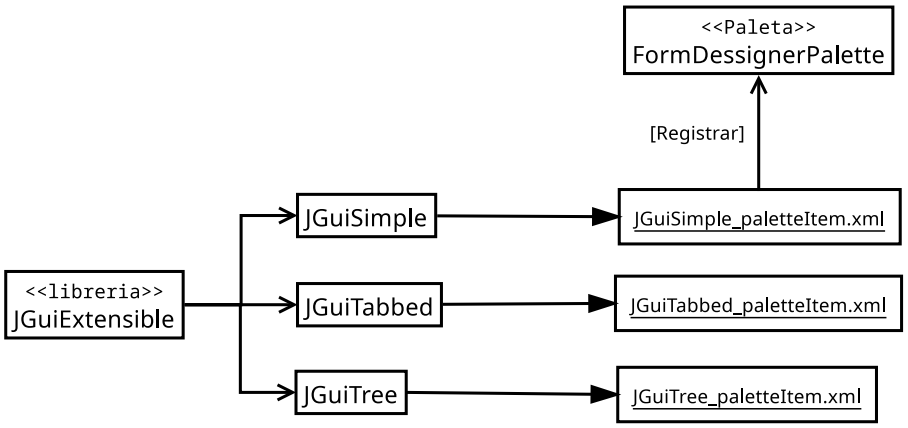


Figure 4.12: Esquema registro items de paleta

Todas estas funcionalidades se registraran a traves del archivo de capa del modulo, mediante archivos XML que funcionan como descriptores de recursos para el sistema de archivos del sistema.

Chapter 5

Implementacion

En este capitulo ofreceremos una vision detallada de todo lo que tiene que ver con la implementacion del patron. Primero expondremos las caracteristicas del hardware y el software utilizado para la elaboracion del proyecto. Seguidamente ofreceremos el diagrama de la arquitectura de clases de la biblioteca y especificaremos su estructura, haciendo un repaso de las clases que la conforman y las funcionalidades que implementan. Finalmente mostraremos el esquema de la arquitectura del plugin desarrollado para NetBeans y los archivos que lo conforman.

5.1. Equipo de desarrollo.

En este apartado vamos a mostrar las caracteristicas del equipo sobre el que se ha realizado todo el desarrollo de la biblioteca, asi como el modulo de NetBeans y el programa de prueba.

5.1.1. Hardware.

Las caracteristicas fisicas del equipo son:

- Placa base: ASRock H61M-VS.
- Procesador: 4 × Intel® Core™ i5-2500K CPU @ 3.30GHz.
- Memoria RAM: 7.7 GiB of RAM DDR3.
- Memoria:
  HDD: ST500DM002-1BC142 de 500GB
  SSD: Samsung SSD 860 EVO 250GB
- Tarjeta grafica: NVIDIA GK208B GeForce GT 710.

5.1.2. Software.

Los distintos tipos de software utilizado son:

5.1.2.1. Sistema operativo.

- Manjaro Linux Versión de KDE Plasma: 5.25.5
- Versión de KDE Frameworks: 5.97.0
- Versión de Qt: 5.15.5
- Versión del kernel: 5.18.19-3-MANJARO (64 bits)
- Plataforma gráfica: X11
- Procesador gráfico: NV106

5.1.2.2. Software de programacion.

- Apache NetBeans IDE 12.5 para el diseño y la codificacion.
- Java: 18.0.2 ; Openjdk version "18.0.2" 2022-07-19; OpenJDK 64-Bit Server VM 18.0.2+0; Runtime: OpenJDK Runtime Environment 18.0.2+0.
- Github y OneDrive para el control y gestion de versiones.

5.1.2.3. Sortware de edicion.

- Lyx vs. 2.3.6.1 para la creacion de este documento.
- KBibTex vs. 0.9.2 para la insercion de bibliografia.
- Diagram Editor Dia vs. 0.97.3 para la creacion de diagramas.
- Gwenview vs. 22.08.1 para la gestion de imagenes.

5.2. Arquitectura de la biblioteca.

Vamos ahora a exponer la jerarquia de clases establecida para la implementacion del patron de reusabilidad. Se mostrara el diagrama de clases de la biblioteca y se detallaran las funcionalidades y algoritmos mas relevantes, asi como las interrelaciones establecidas entre las distintas clases, sus metodos y sus interfaces. Se iran tambien aclarando las diferentes decisiones de implementacion que se han ido tomando a lo largo del desarrollo, sus causas y sus justificaciones.

5.2.1. Diagrama de clases.

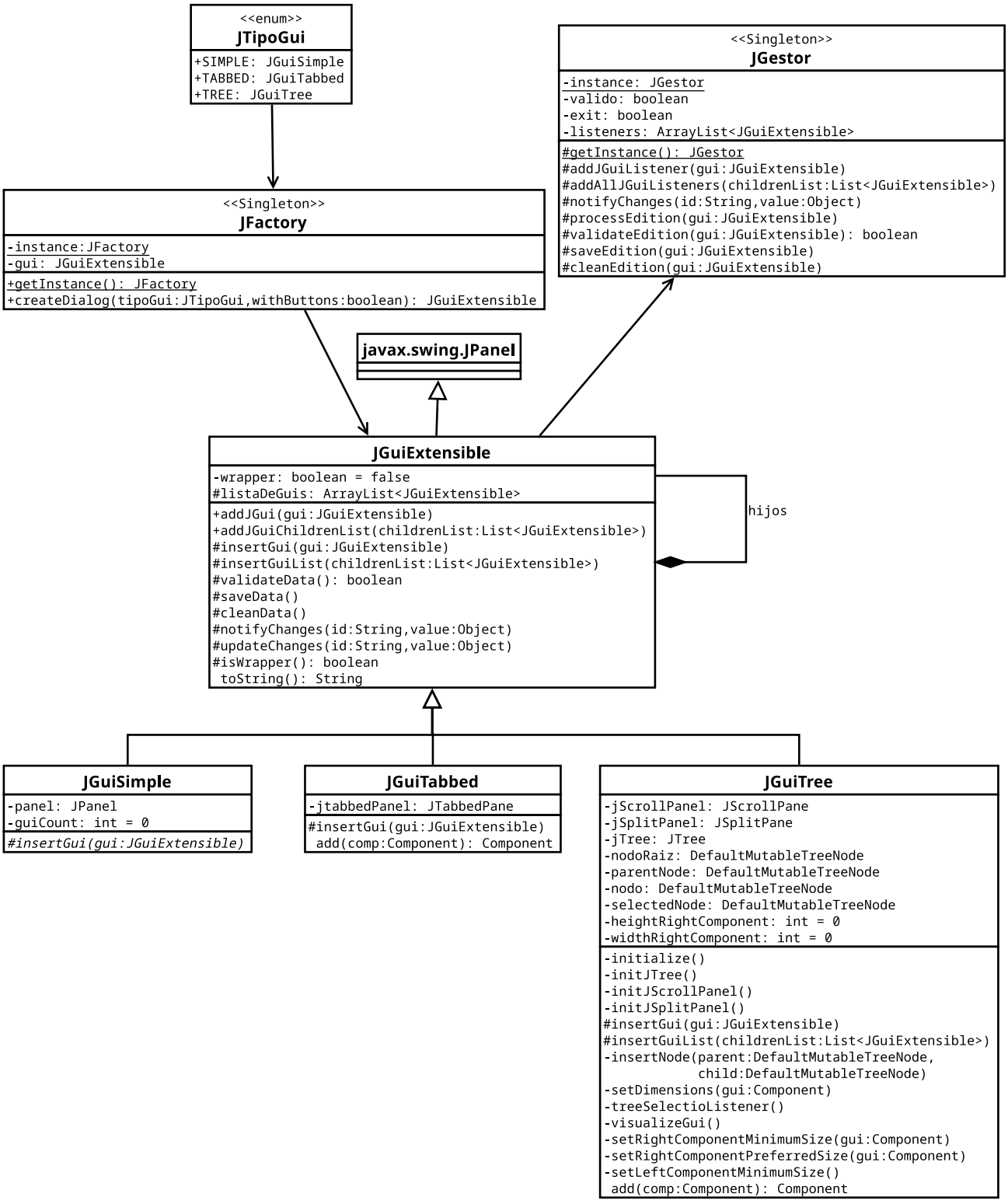


Figure 5.1: Diagrama de clases de la biblioteca.

5.2.2. Estructura de clases.

En este apartado vamos a especificar las funcionalidades de los distintos metodos implementados en cada una de las clases. Se comenta la funcionalidad global de cada clase y sus atributos, y posteriormente se repasan cada uno de los metodos describiendo la funcionalidad que abarcan y las relaciones que establecen con otros metodos.

La denominacion de las clases siguen cierta pauta establecida en la implementacion de la libreria en C++. La J del inicio de cada clase y de los distintos metodos se refiere al lenguaje en el que estan escritas, el lenguaje Java.

5.2.2.1. JFactory.

Como ya se comento durante la fase de analisis y diseño, esta clase se corresponde con la factoria que utilizara el programador de aplicaciones para crear guis vacias. Es el componente encargado de crear cada uno de los tres tipos de guis posibles. Hemos considerado que solo es necesaria la creacion de una instancia de la factoria, con lo que la clase se ha implementado utilizando un patron Singleton, de modo que se puede acceder directamente a la instancia de clase mediante su metodo estatico y a traves de ella al metodo de creacion de guis. Los atributos de la clase son la gui que se va a crear y la instancia que va a guardar la clase de si misma.

getInstance

Metodo estatico que devuelve la unica instancia de la factoria. Primero se comprueba si ya esta creada, si lo esta se devuelve la instancia creada anteriormente, sino se crea y se guarda en una variable estatica. El metodo esta sincronizado para evitar que se puedan crear dos instancias por dos hilos distintos que comprueban y crean la instancia a la vez.

createDialog (tipoGui, withButtons)

Es el metodo de creacion de guis. Los parametros que el programador debe pasarle son el tipo de gui a crear (JTipoGui) y un booleano para determinar si quiere que el dialogo generado disponga de los botones de aceptacion de edicion, Ok y Cancel. Si la gui se crea con botones, se modifica el layout de la gui y se inserta un panel con los botones al final de pagina de la ventana. Durante la creacion, al tratarse de un dialogo vacio, se establece la gui como wrapper o envoltorio, con lo que no tendra que recorrerse a la hora de buscar widgets de edicion que validar.

panelBtns(gui)

Metodo interno que devuelve el panel con los botones de edicion para insertar en la nueva gui. Los botones estan enlazados con la gestion de la edicion de guis reusables, uno para iniciar el

procesamiento de la edicion (OK) y el otro para limpiar los datos introducidos y salir de la edicion (Cancel).

5.2.2.2. JTipoGui

Esta clase es una enumeracion en el que se definen los distintos tipos de gui que se pueden crear y como deben ser nombradas: SIMPLE, TABBED o TREE.

5.2.2.3. JGuiExtensible.

Esta es la clase base de la jerarquia de guis reusables. La clase se ha denominado asi por tratarse precisamente de guis que pueden ser extendidas con otras guis. Los distintos tipos de guis son subclases de esta y heredan de ella, con lo que utilizaran sus metodos publicos y sobreescribiran polimorficamente los que afecten a la implementacion de cada una en particular. JGuiExtensible hereda de javax.swing.JPanel, la clase JPanel de la libreria Swing, con lo que dispone de toda la funcionalidad que esta clase le ofrece, y del mismo modo lo hace el resto de la jerarquia de guis.

Los atributos de clase son la lista de guis añadidas como hijos a cada una de las guis[1], un entero que determina el numero de guis insertadas y un booleano que establece si la gui es una gui vacia que funciona como wrapper o se trata de una gui diseñada.

addJGui (gui)

Añade una gui hija a la gui. Se incluye la nueva gui en la lista de hijos y en una estructura de datos de JGestor donde se almacenan los listeners para la implementacion del mecanismo de comunicacion entre guis. Se llama tambien al metodo interno polimorfico para la insercion de guis.

addJGuiChildrenList (childrenList)

Tiene la misma funcionalidad que el metodo anterior pero dedicado al tratamiento de listas de guis que se insertan al mismo nivel en la gui padre. De la misma forma se insertan como listeners a traves de JGestor y llama al metodo interno de insercion de guis.

insertJGui

Metodo interno que funciona polimorficamente para insertar una gui en otra.

insertJGuiList

Metodo interno que funciona polimorficamente para insertar listas de gui en otra.

[1]Ya se discutio con anterioridad los problemas de la inclusion de la lista de hijos en la clase base 4.1.3.1

isWrapper

Este metodo devuelve un booleano identificando si la gui es una gui diseñada, con lo que debe tener forzosamente implementados una serie de metodos, o si en realidad se trata de una gui vacia que funciona como envoltorio para insertar otras guis en dialogos con o sin botones.

setWrapper

Metodo para determinar que la gui que se va a crear es una gui vacia que funciona como envoltorio de otras guis. Estas guis, al no tener componentes de edicion, no necesitan ser recorridas cuando se realiza la validacion y confirmacion de los datos.

Los siguientes metodos corresponden a los metodos de edicion de gui, de obligada implementacion por parte del diseñador de guis. Si no se implementan en las guis que se estan diseñando, lanzan una UnsupportedOperationException() avisando de que el metodo debe ser implementado por el desarrollador. Esto es asi porque la forma de tratar los datos puede variar de una aplicacion a otra y depende en gran medida del diseño y la politica de tratamiento de datos que se quiera establecer. Cualquiera de los metodos de esta serie puede ser implementado en vacio para evitar el lanzamiento de la excepcion.

validateData

A traves de este metodo se realiza la validacion de los datos introducidos. Se pueden realizar todo tipo de comprobaciones que el diseñador considere oportunos, se puede comprobar si los datos entran dentro de un rango determinado, si faltan datos por introducir, si el tipo introducido es el adecuado, si se adecuan a una interfaz determinada, etc... La implementacion dependera de los widgets de edicion y la forma en que se quiera gestionar los datos. Este metodo devuelve un booleano confirmando que los datos son validos o no. Es llamado por validateEdition() en el JGestor, que realiza el recorrido recursivo por las guis reusables obteniendo de cada una la confirmacion.

saveData

Mediante este metodo se implementa el sistema escogido por el diseñador para guardar los datos de edicion. Puede implementar una interfaz hacia el Modelo de datos que se establezca en la aplicacion para posteriormente acceder a una base de datos para almacenarlos, o simplemente guardarlos en un archivo en el sistema anfitrion. La implementacion nuevamente dependera de como se diseñe la gestion de datos.

cleanData

Realiza una limpieza eliminando toda la serie de datos introducidos. La gestion de la limpieza de datos tambien puede implicar eliminar archivos, enviar directrices de eliminacion a la base de datos,

y puede motivar otra serie de operaciones que se necesiten realizar en funcion del tipo de aplicacion para la que se este implementando la interfaz grafica.

notifyChanges (id,value)

Metodo de notificacion de eventos. El componente que desee enviar un mensaje llamara a este metodo y le pasara un identificador de quien ha activado el evento y el valor que ha cambiado. Este metodo llamara a la clase JGestor para que gestione la notificacion.

updateChanges (id, value)

Esta funcionalidad forma parte de la implementacion del sistema de comunicacion entre guis incluido en los requerimientos del sistema [RF 16]. Tiene la particularidad de constituir parte de la implementacion del patron Observer, y se corresponde con el metodo que actualiza los cambios en el componente observador una vez se ha realizado la notificacion. NotifyChanges recorre recursivamente todas las guis oyentes de evento, y notifica que ha habido un cambio en un componente determinado. Mediante este metodo, la gui recoge la notificacion y modifica el componente adecuado con el valor determinado. Los parametros que se le pasan al metodo corresponden entonces a un identificador del componente que ha activado el evento y el valor que ha sido modificado. El diseñador debe implementar en las guis el componente o la serie de componentes que se ven afectados por el cambio en cuestion y transmitirles el valor de cambio.

5.2.2.4. JGuiSimple

Esta clase representa las guis de navegacion simple. Son una subclase de JGuiExtensible con lo que dispone de toda su funcionalidad y la de la clase JPanel de la que hereda. Tanto en esta clase como en el resto se ha intentado aplicar el estilo New Jersey [NJS, 1989], manteniendo la simplicidad de la implementacion al maximo, con lo que su funcionalidad basicamente se ha limitado a sobreescribir el metodo interno de insercion de guis. En el constructor se ha inizialido tan solo un jpanel sobre el que realizaremos la insercion y se ha modificado su layout para que las guis se integren adecuadamente al diseño de la ventana. El jpanel se utiliza para poder realizar inserciones recursivas de composiciones realizadas con esta clase.

insertJGui (gui)

Metodo interno para la insercion de guis. Se sobreescribe el metodo de la clase base para que la gui se inserte en el jpanel y para implementar el contador de guis. Recordemos que uno de los requerimientos era que no se pudiesen anidar mas de dos dialogos simples en la misma gui [RF 6]. Si se supera el limite se lanza una Excepcion y un mensaje de advertencia.

La insercion de listas de guis se realiza a traves de este metodo desde el metodo de insercion de listas de la clase base.

5.2.2.5. JGuiTabbed

Esta subclase de JGuiExtensible genera las guis de navegacion mediante pestañas. Al igual que la clase anterior, su implementacion se ha intentado simplificar al minimo, con lo que tan solo se ha implementado el metodo interno polimorfico para la insercion de guis. La construccion de estas guis se crean sobre un JTabbedPane al que se van añadiendo pestañas por cada gui que se inserta.

insertJGui (gui)

Metodo interno para la insercion de guis. Se sobreescribe el metodo de la clase base para que la insercion de gui se realice sobre el JTabbedPane. Cada gui inserta se correspondera con una pestaña del dialogo.

La insercion de listas de guis se realiza a traves de este metodo desde el metodo de insercion de listas de la clase base.

5.2.2.6. JGuiTree

Esta clase supone una implementacion bastante mas elaborada que las anteriores ya que el manejo de los nodos en el arbol de navegacion obliga a manejar bastantes mas objetos y realizar calculos de tamaño para poder visualizar adecuadamente las guis en la ventana. La adecuacion de las vistas en un panel dividido se establecen en funcion de los tamaños minimos y preferidos del objeto a insertar, con lo que se deberan adecuar las dimensiones de la vista a los distintos tamaños de las guis insertadas.

Los atributos que maneja esta clase son los siguientes:

- *jScrollPanel:* panel con barra de desplazamiento para alojar la vista de arbol
- *jSplitPanel:* panel con divisor para ubicar el jscrollpanel con la vista de arbol en la parte izquierda y la vista de la gui en la parte derecha del panel.
- *jTree:* arbol sobre el que se articulara la insercion de guis. Se agrega al jScrollPanel para disponer de barras de desplazamiento si el tamaño del arbol crece mas alla de las dimensiones de la ventana.
- *rootNode, parentNode, nodo, selectedNode:* nodos de la clase DefaultMutableTreeNode para la gestion de los nodos del arbol.
- *model:* objeto de la clase DefaultTreeModel representando el modelo de datos del arbol.
- *infoNode:* objeto donde se extrae la gui almacenada en los distintos nodos.

- *panelNode:* objeto que recoge el objeto de gui y se pasa al panel de visualizacion.
- *heightRightComponent, widthRightComponent:* integers que se utilizan para gestionar la altura y anchura maxima de la ventana al insertar las guis en el panel de visualizacion.

El nodo raiz, asi como los nodos padres de la listas de nodos se configuran con el primer objeto de gui introducido, de modo que el nodo en cuestion y el primer nodo de la lista señalan al mismo objeto.

initialize

Metodo de inicializacion donde se concentran el resto de metodos de inicializacion del arbol, el jscrollpanel y el jsplitpanel, asi como el nodo raiz y el modelo de datos del arbol.

initJTree

Inicia el arbol. Crea la instancia y establece una serie de parametros de visualizacion.

initJScrollPanel

Inicia el JScrollPanel. Crea la instancia e inserta la instancia del arbol en la vista del jscrollpanel.

initJSplitPanel

Inicia el jSplitPanel. Crea la instancia y establece una serie de parametros de visualizacion. Inserta el jscrollpanel en la parte izquierda de la vista.

insertJGui (gui)

Metodo interno polimorfico para la insercion de guis. Crea un nuevo nodo con la gui y la introduce en el arbol. Establece las dimensiones preferida y minima de las vistas izquierda y derecha en funcion del tamaño de la gui que se inserta. Configura el oyente de seleccion del arbol con el nuevo modelo de datos generado una vez insertada la nueva gui.

insertJGuiList

Metodo interno para la insercion de listas de guis reusables. Se establece la primera gui de la lista como nodo padre y el resto se insertan como hijas.

insertNode

Metodo para la insercion de nodos. Se insertan los nodos en el modelo de datos del arbol como padre e hijo. Si no se ha vinculado ningun objeto con el nodo raiz se establece con el objeto del primer hijo. Cualquier nodo padre siempre señala al objeto insertado en el primer hijo.

setDimensions

Metodo de concentracion para configurar las dimensiones minima y preferida de la parte izquierda y derecha de la vista del panel dividido.

setRightComponentMinimumSize (gui)

Se obtienen la altura y anchura minimas de la nueva gui y se comparan con las guis anteriores. La altura y anchura minimas se estableceran en funcion del tamaño minimo de cada una de las guis insertadas. Se trasladan al JSplitPanel las nuevas dimensiones minimas de la parte derecha de la vista.

setRightComponentPreferredSize (gui)

Se obtienen la altura y anchura preferidas de la nueva gui y se comparan con las guis anteriores. La altura y anchura preferidas se estableceran en funcion del tamaño preferido de cada una de las guis insertadas. Se trasladan al JSplitPanel las nuevas dimensiones preferidas de la parte derecha de la vista.

setLeftComponentMinimumSize

Se configura el tamaño minimo de la vista izquierda del panel dividido en funcion de la parte visible del jscrollpanel.

treeSelectionListener

Metodo para configurar el oyente de eventos y obtener la seleccion del usuario a la hora de elegir con el raton un elemento en la vista de arbol. Una vez detectado el evento, se llama a visualizeGui para visualizar el objeto de gui almacenado en el nodo en la parte derecha del panel dividido.

visualizeGui

Metodo para visualizar en el panel derecho de la vista el objeto de gui seleccionado. Se obtiene la direccion del nodo seleccionado por el usuario, se obtiene el objeto que almacena el nodo, que en nuestro caso es una gui y finalmente se pasa el objeto de gui a la parte derecha de la vista para visualizarlo.

5.2.2.7. JGestor.

Esta clase se encarga de gestionar la comunicacion entre guis y de concentrar la funcionalidad derivada de la edicion de guis. La inclusion de esta clase se gesto a la hora de implementar el patron observer para articular el paso de mensajes entre guis. La comunicacion se debia establecer

entre todas las guis, de forma que todas fueran emisoras y oyentes de evento, con lo que se decidio implementar en la clase base una variable de clase con la lista de todos los dialogos que conformaran la gui de aplicacion. Se decidio entonces extraer esa variable estatica e incluirla en una clase aparte configurada como un singleton de manera que almacenara la lista de guis e hiciera de mediador en la comunicacion entre ellas. Finalmente se decidio sacar de la clase base toda la funcionalidad que no tuviera que ver con la insercion de guis e incluirla en la clase mediadora. De esta manera se concentro en ella los metodos recursivos de validacion y aceptacion de la edicion de las guis.

Los atributos de la clase son la lista de oyentes, dos booleanos para la gestion de la validacion de datos y la instancia de singleton.

getInstance Metodo estatico que devuelve la unica instancia de la clase. Se comprueba primero si ya esta creada y si no lo esta se crea. El metodo esta sincronizado para evitar que dos hilos distintos puedan crear instancias paralelas del singleton.

addJGuiListeners (gui)

Añade una gui a la lista de guis oyentes de evento. Si la gui ya esta incluida en la lista se excluye de la adicion.

addJGuiListeners (childrenList)

Añade una lista de guis a la lista de oyentes.

notifyChanges (id, value)

Metodo para notificar a la lista de guis oyentes que ha ocurrido un evento. El metodo recorre la lista de oyentes y llama al metodo de actualizacion de cada gui que no sea un wrapper, pasandole el componente que ha activado el evento y el valor que ha sido modificado.

processEdition (gui)

Metodo que procesa la edicion de gui. Comprueba que los datos sean validos y si el proceso de validacion es positivo se activa una ventana de confirmacion para guardar los datos. Finalmente, mediante un switch funcional se gestionan los distintas opciones: limpia los datos introducidos si no quiere guardar los datos, los guarda si la opcion es si, y lo deja todo como esta si opta por la opcion de cancelar.

validateEdition (gui)

Este metodo realiza la validacion de los datos de cada gui. Llama al metodo validateData que el diseñador de guis ha implementado previamente y almacena el resultado de la validacion. Recorre

entonces todas sus guis hijas llamandose recursivamente para comprobar la validacion de cada una. Si alguna de las validaciones no es verdadera sale del bucle y el metodo devuelve falso, con lo que la validacion completa de la gui queda anulada.

saveEdition (gui)

Este metodo guarda los datos introducidos en los widgets del dialogo. LLama al metodo saveData que el diseñador de guis ha implementado previamente. Luego recorre la lista de guis hijas llamandose recursivamente en cada gui. La forma de almacenar los datos quedan relegados al metodo que implementa el diseñador de guis.

cleanEdition (gui)

Este metodo limpia los datos introducidos en cada uno de los componentes. Llama al metodo cleanData que el diseñador ha implementado previamente y recorre la lista de guis hija llamandose recursivamente.

La biblioteca facilita estos ultimos tres metodos implementando la recursividad necesaria para aplicar los procesos de edicion a las guis reusables, delegando en los metodos de obligada implementacion la tarea de validar, limpiar y guardar los datos de los distintos componentes de cada gui. Esta es la razon por la cual el diseñador debe implementar estos metodos, ya que solo el sabe cuales son los componentes que va a incluir en cada gui y como deben ser validados o guardados los datos.

5.3. Arquitectura del Plugin de Netbeans.

Vamos a mostrar y desarrollar ahora el diagrama de los distintos paquetes y archivos que conforman la extension realizada sobre NetBeans para integrar las clases de la libreria en el entorno, y concretamente en el entorno de diseño.

5.3.1. Diagrama del plugin.

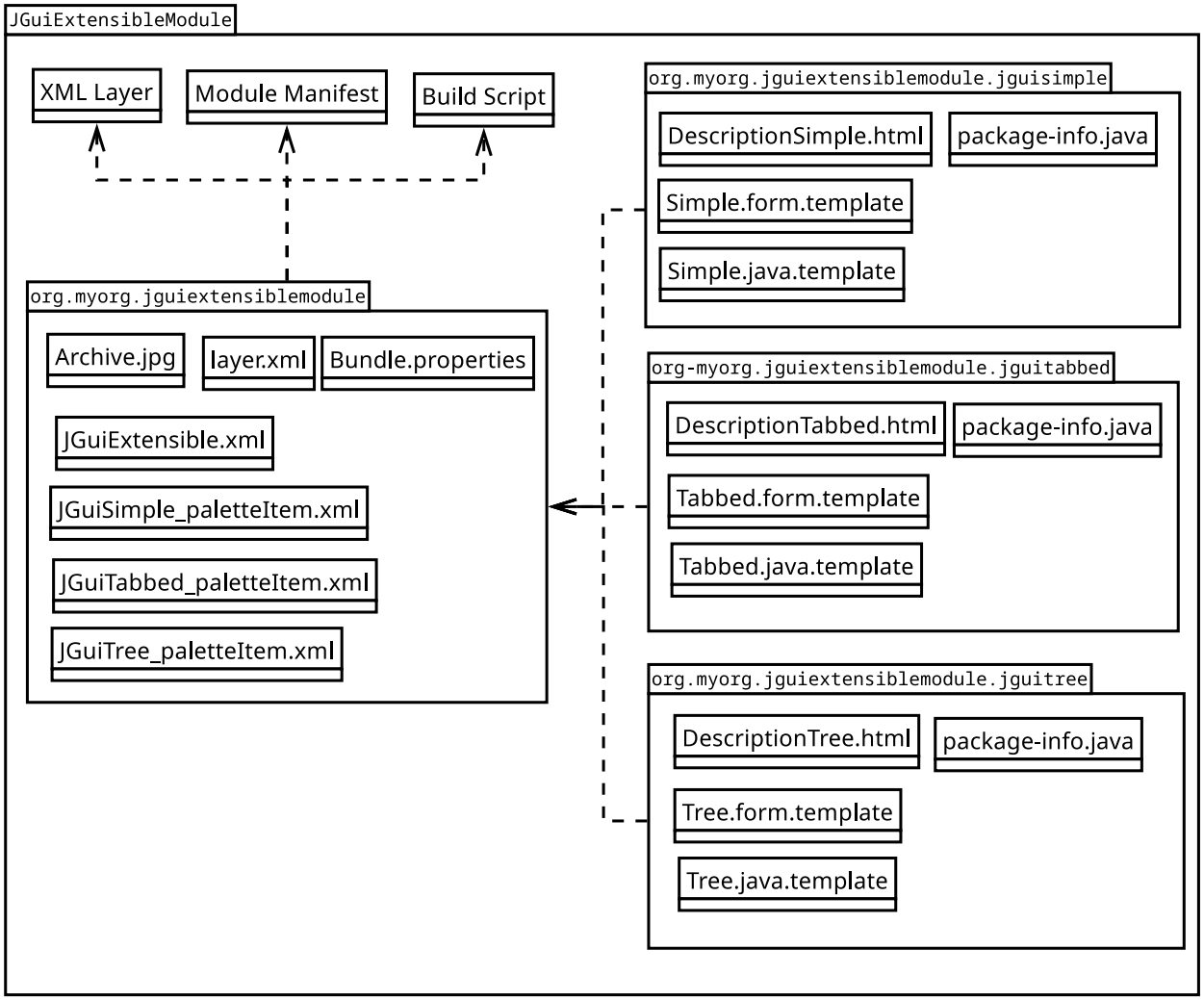


Figure 5.2: Esquema paquete extension NetBeans.

5.3.2. Estructura del plugin.

Desgranaremos ahora la composicion de los distintos paquetes que conforman el plugin de la extension JGuiExtensible para NetBeans.

La estructura de los paquetes destinados a las plantillas de los tres tipos de guis siguen una pauta de construccion similar con lo que solo se describira la composicion de uno de ellos.

5.3.2.1. org.myorg.jguiextensiblemodule.

En este paquete se encuentran los archivos necesarios para añadir la biblioteca al catalogo de librerias de NetBeans, asi como las referencias para añadir las clases como un item mas a la paleta de componentes de la vista de diseño del editor grafico.

archive.jpg es el archivo con el icono que representara las plantillas de las distintas guis en el menu de plantillas del wizard.

bundle.properties es el archivo donde se guardan los datos y configuraciones del paquete.

JGuiExtensible.xml archivo xml que funciona como descriptor de biblioteca Java SE para incorporar al archivo layer. xml. Localiza el paquete y añade al classpath el archivo JAR de la bibilioteca como recurso.

JGuiSimple_paletteItem.xml archivo xml que crea un componente de la paleta con la clase JGuiSimple extraido de la libreria JGuiExtensible obtenida como recurso del classpath.

JGuiTabbed_palettItem.xml idem que el archivo anterior pero la clase que utiliza para el componente es la clase JGuiTabbed.

JGuiTree_paletteItem.xml idem anteriores con JGuiTree.

layer.xml archivo de capa del modulo que define los archivos y las carpetas que se combinaran con el sistema de archivos del sistema y el resto de archivos de capa para configurar el IDE en tiempo de ejecucion. En nuestro claso incluyen las referencias a la carpeta FormDesignerPalette donde incluimos la carpeta JGuiExtensible con los tres items de la paleta de componentes y la inclusion en la carpeta Libraries en org-netbeans-api-project-libraries de la referencia a la biblioteca JGuiExtensible. Se incluyen tambien los templates para la creacion de las clases desde el asistente de plantillas.

5.3.2.2. org.myorg.jguiextensiblemodule.jguisimple.

En este paquete se incluyen los distintos archivos que se utilizaran para acceder a los templates y poder crear los distintos tipos de guis desde el asistente para la creacion de plantillas de NetBeans. Los archivos que lo componen son:

DescriptionSimple.html archivo html para insertar en el cuadro de dialogo del asistente de plantillas, una descripcion del nuevo archivo que se va a crear, en nuestro caso, un JGuiSimple.

Simple.form.template plantilla para generar el archivo de diseño para la interfaz grafica de la clase. Se utiliza para almacenar informacion sobre la colocacion de los componentes en la vista de diseño.

Simple.java.template plantilla para crear un esquema basico de la clase en la vista de codificacion del IDE. Se introducen los distintos metodos que el diseñador de guis debe implementar. Se utiliza FreeMarker Template Language como motor de secuencias de comandos para introducir datos en tiempo de ejecucion en el template.

package-info.java clase mediante la cual se registra el template en el sistema de archivos virtual de NetBeans. Se registran tanto la plantilla del archivo form como la plantilla de la clase, especificando el icono, los recursos asociados, la etiqueta del template, el motor de secuencias de comandos utilizado, el archivo de descripcion y el fichero donde se localizara el template.

El resto de paquetes con los otros tipos de gui, jguitabbed y jguitree, se componen de los mismos elementos y tienen la misma funcionalidad.

5.3.2.3. Otros archivos importantes.

A la hora de la creacion de un modulo de NetBeans, se generan automaticamente una serie de archivos relevantes que se utilizan internamente para localizar los recursos ofrecidos por el modulo, para identificarlo como modulo y para configurar las distintas dependencias y las clases del proyecto dentro del entorno. Los archivos mas relevantes son los siguientes:

XML-Layer archivo de capa del paquete. Crea un archivo de capa para registrar archivos y carpetas en el registro central del sistema. (Vease layer.xml en org.myorg.jguiextensiblemodule) 5.3.2.1

Module_manifest es una descripcion textual del modulo y su entorno. Especifica su nombre, layer.xml, paquete de localizacion, version de especificacion y otro tipo de informacion relacionada con el modulo. Es el primer archivo leido por el sistema de modulos cuando el modulo se carga.

Build_script script de compilacion llamado por el IDE. Se utiliza para añadir objetivos y tareas al proceso de compilacion o definir o modificar alguna tarea antes o despues de algun paso de dicho proceso. En nuestro caso solo se especifica el nombre del proyecto y se importa "nbproject/build-impl.xml" para la compilacion.

project.xml archivo de metadatos generado por el IDE para especificar rutas de clases y dependencias del modulo. En nuestro caso se especifica el nombre del modulo y las referencias a org.openide.util (Base Utilities API) y a org.netbeans.api.templates (File Templates), para incorporarlas como dependencias a la carpeta de librerias del proyecto.

El resto de archivos son de configuracion interna de la plataforma.

Chapter 6

Experimentacion y pruebas.

La realizacion de todo tipo de pruebas es fundamental en el desarrollo de cualquier proyecto, de forma que se pueda asegurar que el funcionamiento del codigo es el deseado. En nuestro caso el desarrollo de la libreria y la implementacion del plugin para NetBeans se realizaron de forma paralela a una aplicacion de prueba donde se fueron probando y consolidando todos los desarrollos realizados. A continuacion expondremos las distintas pruebas que se realizaron mediante la aplicacion de los distintos mecanismos implementados en la biblioteca y las modificaciones que se realizaron en dicha implementacion en base a los resultados que se fueron obteniendo.

6.1. Aplicacion de Prueba.

Para realizar las pruebas de las clases de la biblioteca se ha desarrollado una pequeña aplicacion enfocada a la clasificacion y registro de instrumentos de medicion de la marca Mitutoyo: diferentes tipos de calibres, micrometros, comparadores, etc. Cada instrumento tiene unas especificaciones y unas metricas generales que son comunes a todos y luego cada tipo dispone de registros concretos de su ambito de medicion que le son particulares. De esta manera, podemos aplicar la reusabilidad de guis reutilizando las que elaboremos para los instrumentos mas sencillos cuando diseñemos las de los que precisen de guis mas elaboradas para definir todas las caracteristicas necesarias para especificarlos. Asi por ejemplo, un micrometro se podra especificar con la integracion de un par de guis generales con las especificaciones y las dimensiones del instrumento, en cambio un micrometro de alturas o un micrometro de interiores precisaran de otras metricas distintas que tendran que especificarse en otras guis y una vez diseñadas añadirse a las guis generales e integrarse con ellas mediante la API de la biblioteca. De esta forma podremos ir viendo como se pueden utilizar las clases de la libreria en una aplicacion, y la utilidad que suponen los dialogos reutilizables para cualquier desarrollador de interfaces graficas.

Se ha intentado enfocar la estructura de la aplicacion siguiendo el patron MVC, aunque este no era el objetivo primordial ya que el desarrollo se ha concentrado en enfocar la aplicacion como

bateria de pruebas para la biblioteca y en aplicar el patron de reusabilidad en su interfaz grafica, sin entrar en las interioridades del tratamiento de los datos, el acceso a bases de datos u otras cuestiones relacionadas mas con el desarrollo BackEnd de la aplicacion .

Toda la codificacion se ha realizado en el IDE de NetBeans construyendo las diferentes guis mediante su diseñador grafico (Matisse), y utilizando los widgets de la paleta del diseñador con las clases de la biblioteca que se fue desarrollando e integrando en paralelo como modulo de NetBeans.

6.1.1. Desarrollo guiado por la aplicacion

El desarrollo de la aplicacion de prueba ha sido continua, progresiva y paralela al de las clases de la biblioteca. Se ha intentado ir adecuando la implementacion de la aplicacion a los desarrollos de la biblioteca de modo ciclico, refactorizando la implementacion de las clases de la libreria con los resultados de la integracion de estas en la aplicacion. De esta manera se ha conseguido un cierto grado de retroalimentacion e incremento en el desarrollo de la biblioteca, de modo que se fueran revelando a traves de la implementacion de la aplicacion posibles fallos o incompatibilidades en el codigo de la libreria.

La aplicacion ha sido el campo de pruebas para las clases de la libreria, es donde se ha ido verificando los distintas funcionalidades que se le exigian a los dialogos creados a partir de la libreria, adaptandolos a los requisitos preestablecidos y asegurandonos de que la aplicacion ofrecia en cada paso los comportamientos deseados. Desde ese punto de vista podriamos decir que la biblioteca siguio un desarrollo guiado por la aplicacion. [1]

6.1.1.1. Pruebas funcionales.

Para ir probando las funcionalidades[2] de la libreria se realizaron distintas pruebas unitarias[3]en el ambito del desarrollo de la aplicacion:

- Funciones relacionadas con la creacion de guis. Se realizaron distintas pruebas de los mecanismos de creacion de dialogos, tanto a traves del entorno de diseño grafico de NetBeans como mediante la interfaz publica de la libreria habilitada para ello. Conforme se añadian al desarrollo tipos de guis diferentes, se fueron haciendo pruebas de adaptacion al entorno grafico, y se fueron integrando a la interfaz, realizando pruebas de distintas combinaciones para la creacion de guis. Finalmente se decidio añadir a la interfaz un parametro para la seleccion de botones en la gui.

[1] En paralelo al concepto TDD(Test Driven Delopment) o BDD (Behaviour Driven Development)

[2] Las pruebas funcionales se encargan de comprobar que las distintas funcionalidades del software se comportan segun lo esperado

[3] Las pruebas unitarias son pruebas que se realizan para asegurar que una funcion determinada cumple su objetivo de manera aislada.

- Funciones relacionadas con el layout de las guis. Al integrar unas guis en otras, la disposicion de las guis hijas en el interior de la gui que la va a albergar supone un problema de distribucion de componentes de dificil solucion. Tuvo que realizarse toda una bateria de pruebas aplicando distintos tipos de layouts y disposicion de paneles para que se adecuaran en lo posible a la distribucion deseada para las guis reusables, sin que se vieran afectados ni los componentes ni el aspecto visual de la gui. Se decidio incluir la configuracion de un layout predeterminado, lo mas sencillo posible, en el constructor de las guis que lo precisaran.

- Funciones relacionadas con la integracion de unas guis en otras. La insercion de unas guis en otras dependen de un conjunto de metodos que interactuan polimorficamente y que se deben adecuar a cada tipo de gui. Las pruebas unitarias respecto a la integracion de guis suponen la comprobacion de que cada metodo inserte adecuadamente una gui en otra conforme a la implementecion realizada en cada uno de los tipos de gui. En el ambito de la aplicacion se hicieron numerosas baterias de combinaciones de insercion de guis, verificando que la integracion de las guis se realizara acorde a su tipo y en condiciones optimas para que tuviesen un resultado visual adecuado.

- Funciones relacionadas con el mecanismo de comunicacion entre guis. Se realizaron distintas pruebas para comprobar el mecanismo implementado para el paso de mensajes. Se verifico el alcance de los mensajes, la dinamica de transmision y recepcion, que la comunicacion se realizara en todo el ambito de la gui de aplicacion, etc..En base a las pruebas realizadas se realizaron distintas modificaciones como la implementacion de la clase JGestor para mediar en la transmision.

6.1.1.2. Pruebas de integracion.

Las pruebas de integracion [4] son necesarias en relacion a la adecuada composicion una vez se inserten las guis reusables unas en otras. Se fueron realizando diferentes tipos de composicion a la hora de desarrollar la aplicacion de forma que al final se pudiera comprobar que las distintas guis se integraban adecuadamente. Para ello se realizaron varios tipos de comprobaciones:

- Se comprobo que los layouts de los distintos tipos de guis integraran de forma solvente las distintas guis reusables que se añadieran y lo hicieran de manera que la distribucion de los distintos componentes de cada gui se adaptara a la composicion.

- Se aseguro que los metodos de validacion y tratamiento de la edicion funcionaran correctamente una vez se hubiera compuesto en su totalidad la gui de aplicacion, comprobando su adecuado funcionamento una vez integradas todas las guis.

[4] Las pruebas de integracion son pruebas que se realizan para comprobar las interacciones entre distintos componentes independientes tras su integracion

- Se realizaron diversas comprobaciones en relacion a la estetica y a la respuesta "responsive" de la interfaz grafica una vez integradas las guis. El comportamiento de los componentes que las conformaban y la composicion de las guis debian tener una buena respuesta a las modificaciones realizadas por el usuario.
- Se realizaron paralelamente pruebas de integracion del modulo de NetBeans en el entorno de desarrollo y se fueron agregando los distintos tipos de gui a la paleta de diseño grafico, comprobando que la agregacion de guis resultara transparente y fiable dentro del entorno de diseño.

Todas las pruebas enumeradas y otras que tenian mas que ver con pruebas de usabilidad y de experiencia de usuario se realizaron todas en el ambito del desarrollo de la aplicacion, de manera que se consiguiera elaborar una interfaz grafica completamente funcional elaborada a la par del desarrollo de la biblioteca.

6.1.2. Estructura de la aplicacion.

La aplicacion desarrollada se ha estructurado de la siguiente manera:

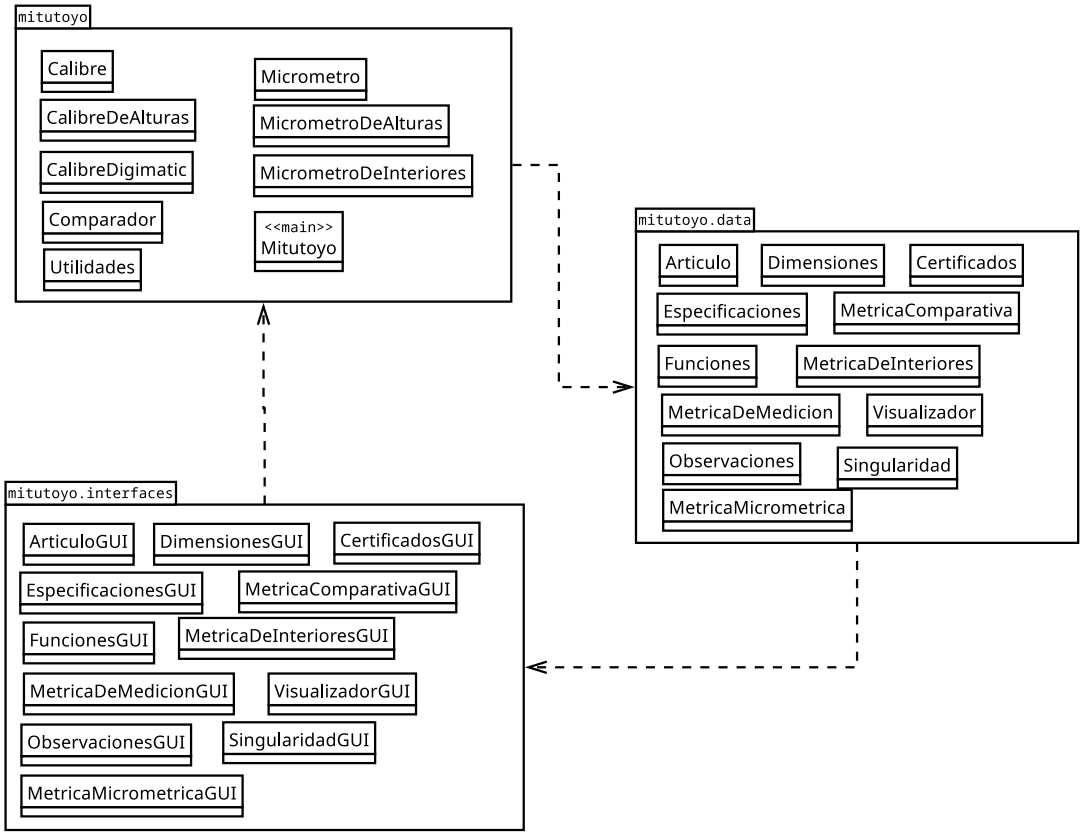


Figure 6.1: Paquetes de la Aplicacion Mitutoyo

El paquete mitutoyo es el paquete central de la aplicacion. En el se crean e integran las guis de

los diferentes tipos de instrumentos de medicion: calibres, micrometros, etc..., y es donde se elabora la jerarquia de clases que representaran cada una de las guis . Las distintas guis se las proporcionan mediante el metodo *createDialog* las diversas clases del paquete data. Cada clase es una estructura para manejar los datos que se introduzcan en la gui (Vease figura6.2) . En el metodo se realiza la llamada a su par clase-gui en el paquete de interfaces donde estan implementadas y diseñadas las distintas guis que representan cada estructura de datos.

6.1.2.1. Paquete mitutoyo.data.

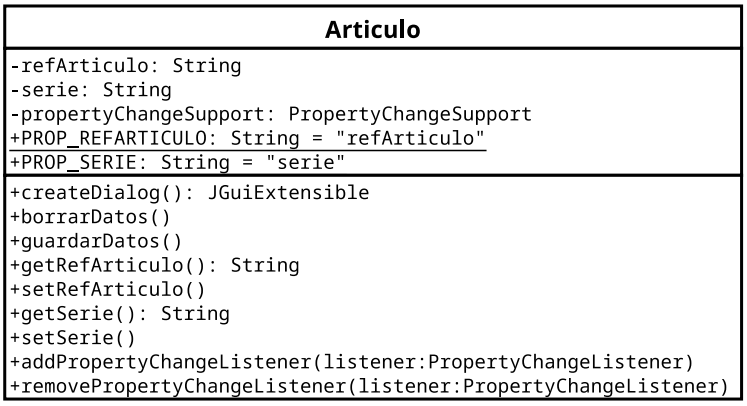


Figure 6.2: Ejemplo: clase Articulo del paquete data

Como podemos observar en el ejemplo, las clases del paquete data son estructuras de datos que siguen el standard de javaBean y se incluyen para el manejo de los datos introducidos en la interfaz grafica de la aplicacion. Los datos obtenidos de la edicion de la gui se almacenan en las variables del javaBean correspondiente y cuando el usuario lanza el evento aceptando guardar los datos, el objeto se guarda. En nuestro caso, los datos se almacenan en la carpeta *test* del programa de pruebas, en formato XML mediante el metodo *saveInXml* de la clase *Utilidades* en el paquete mitutoyo, pero se podria implementar la conexion con una Base de Datos o guardar en el sistema de ficheros anfitrion o utilizar cualquier otra forma de almacenamiento persistente.

En estas clases hemos implementado el metodo *createDialog* para crear la gui asociada a la clase de manera que si en algun momento quisieramos cambiar la interfaz grafica asociada al javaBean, solo tendriamos que cambiar la instancia de la gui en el metodo, sin tener que refactorizar codigo en la clase que utiliza la estructura de datos.

6.1.2.2. Paquete mitutoyo.interfaces.

En este paquete se incluyen las guis de cada una de las estructuras de datos mencionadas (vease fig.6.3). Cada gui ha sido diseñada mediante el editor grafico de diseño de NetBeans, utilizando las clases de la biblioteca y diseñando e implementando cada gui acorde a las exigencias de la libreria.

Se han utilizado cierta variedad de componentes para comprobar la versatilidad de las guis reusables, incluyendo la anidacion de otras guis reusables en el propio diseño.

Cabe destacar que se han implementado los metodos exigidos por la biblioteca: *validateData*, *saveData, cleanData* y *updateChanges* . Asimismo, se ha implementado en diversas clases el metodo *updateChanges* de manera que actualice el valor en distintos componentes en funcion del componente que lanzo el evento. El metodo se ha implementado mediante un switch funcional, de modo que se puede realizar extensiones de codigo, tanto respecto a los identificadores de activador de evento, como al numero de componentes que den respuesta al mismo evento en esa gui.

Los mecanismos ideados para realizar la validacion y salvaguarda de los datos son responsabilidad del diseñador de guis, en este caso se ha utilizado un metodo sobrecargado de la clase *Utilidades* para determinar si el campo se ha rellenado, si el valor se ha seleccionado o esta dentro de unos limites de rango. La comprobacion de que se ha introducido datos adecuados, en el caso particular de los datos numericos, se realiza utilizando el componente JFormattedTextField que presupone la insercion de enteros en una rango determinado.

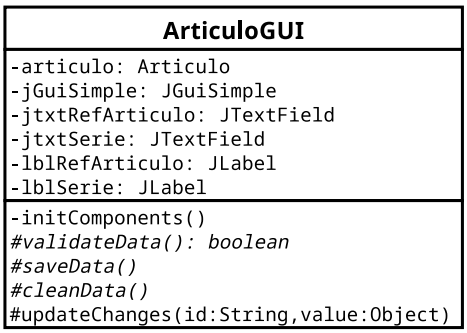


Figure 6.3: Ejemplo: clase ArticuloGUI del paquete interfaces

6.1.2.3. Paquete mitutoyo.

En este paquete se concentra la jerarquia de clases que representan cada uno de los distintos tipos de instrumentos de medicion del catalogo de la aplicacion. Cada clase posee un metodo *createDialog* que devuelve un objeto JGuiExtensible con la gui construida mediante guis reusables y que muestra la interfaz grafica de cada uno de los distintos instrumentos. La composicion de guis se realiza utilizando los metodos publicos de la interfaz de la libreria, integrando unas guis en otras, asi como creando distintos tipos de guis vacias para envolver dialogos diseñados que se van insertando conforme lo va exigiendo el diseño. El metodo devuelve finalmente un diseño para cada clase, que sera reutilizado en las subclases o en otras clases que deseen reutilizar la composicion de guis realizada para construir la suya.

La clase Mitutoyo es la clase main de la aplicacion, y es la que lanza la visualizacion de las distintas guis en cascada, creando la ventana en la que se va a insertar la gui diseñada. Dispone tambien de un metodo para cambiar el aspecto visual de las guis y poder ver la aplicacion con otros

looks and feels (Vease anexo c C).

Las ventanas y otros mecanismos utilizados en toda la aplicacion se almacenan en la clase *Utilidades* . Esta clase dispone de una bateria de metodos estaticos a los que se accede para realizar diferentes funciones:

mostrar(message) lanza una ventana con un mensaje de aviso.

crearFrame(gui) crea una ventana para albergar la gui que se ha construido.

validarCampoVacio(campo) comprueba que el campo no este vacio para validar la edicion

rango(comp,max,min) comprueba que el valor del componente se encuentre dentro de un rango de valores maximo y minimo.

matcher(comp,name,max,min,obj) comprueba que se hayan introducido los datos y se encuentren dentro de un rango determinado. Si no es asi lanza un mensaje de aviso.

matcher(comp,name,obj) metodo sobreescrito para validar que se hallan introducido los datos, tanto en componentes de texto como numericos, y que se haya realizado la seleccion en los componentes de JComboBox.

saveInXml(name,obj) guarda el objeto en un archivo XML en la carpeta *store* del proyecto.

Estas funciones se utilizan en el resto de clases de la aplicacion para realizar comprobaciones de edicion y salvaguarda de datos fundamentalmente.

En las diversas clases del paquete se han utilizado las diversas formas para la insercion de guis de que dispone la libreria con el fin de probar la interfaz para añadir dialogos y comprobar asi que las guis se insertaran adecuadamente de todas las maneras posibles, siguiendo las premisas esteticas y estructurales de los requirimientos, y que ademas tuviesen una buena respuesta en el ambito de la aplicacion.

6.2. Integracion en el entorno de diseño de NetBeans.

La integracion de las clases de la libreria en el editor grafico de NetBeans y su utilizacion de forma transparente en el diseño de una serie de interfaces graficas fue tambien objeto de la realizacion de pruebas durante el desarrollo de la aplicacion.

Las clases implementadas se fueron incorporando al plugin desarrollado para la integracion de la libreria en NetBeans, de manera que se pudiesen utilizar para la creacion y diseño de las gui de la aplicacion. Se utilizaron tanto las plantillas o templates del wizard, como los widgets incorporados a la paleta de componentes a la hora de ir desarrollando el diseño de las distintas guis, para insertarlas posteriormente incluyendolas en la gui de aplicacion. Una vez integrada la plantilla y el bean del widget como nuevo componente de la paleta, se realizaron las siguientes comprobaciones:

- Se comprobo que la libreria se incorporara sin problemas a la lista de librerias del entorno.
- Se comprobo que se crearan adecuadamente las Gui Forms de las clases de la libreria en la vista de diseño de NetBeans utilizando el gestor de plantillas.
- Se comprobo que los widgets de las guis se incorporaran adecuadamente desde la paleta de componentes mediante la funcion drag and drop del entorno.
- Se comprobo que los demas componentes de la paleta se insertaran adecuadamente al panel de diseño.
- Se comprobo que los metodos a implementar aparecieran en la vista de codificacion una vez se creara la clase desde el gestor de plantillas.

En definitiva, se realizaron diversas pruebas para comprobar que la integracion de la libreria en el entorno grafico fuera completamente transparente para el desarrollador, y que cada uno de los tipos de gui que se añadia, se comportara como un componente mas de la paleta o como un template mas, y que todas las funcionalidades se aplicaran de acuerdo a los requerimientos establecidos, lo que vendria a decir que la integracion del plugin de la libreria en la plataforma de NetBeans se habia realizado con exito.

Chapter 7

Conclusiones y trabajos futuros

En esta parte final de la memoria ofreceremos las conclusiones extraidas de la realizacion del proyecto y las posibles extensiones y mejoras que se le podrian aplicar a futuro.

7.1. Conclusiones

Para realizar la implementacion del Patron de Reusabilidad de Guis, que era el objetivo primordial de este proyecto, hemos realizado previamente una descripcion del patron en base a la estructura de los patrones de diseño estudiadas en la seccion 2.2.3 y al esquema sacado de [Gamma et al., 1994] seccion 1.3.

A traves del estudio de los patrones hemos comprendido la importancia del concepto de reutilizacion en el ambito de la ingenieria del software y hemos podido comprobar la amplia variedad de mecanismos de reutilizacion que se han ido introduciendo en todo el ambito del desarrollo de software, entre ellos el de los patrones de diseño.

Hemos ido aplicando distintos patrones a la hora de desarrollar el diseño de nuestra biblioteca, comprobando como los distintos patrones se combinan y colaboran entre si de diferentes maneras para conformar nuevas soluciones a los problemas de diseño que se nos han planteado.

Por otro lado, respecto a las cuestiones mas particulares de la implementacion, hemos facilitado al desarrollador de aplicaciones una interfaz publica reducida mediante la que poder crear cualquier tipo de dialogo vacio a traves de una factoria de guis. Utilizando los parametros de creacion de la factoria, el desarrollador puede elegir el tipo de dialogo que pretende crear y escoger si quiere que la gui tenga botones de aceptacion y cancelacion o no los tenga.

Tambien, a traves de esta interfaz publica a la que se puede acceder mediante la vista de codificacion del entorno de desarrollo, se han habilitado dos metodos a traves de los cuales el desarrollador podra insertar guis o listas de guis a la hora de generar la interfaz de la aplicacion. Estas guis podran ser tanto guis vacias que cree el desarrollador directamente, como guis previamente diseñadas por el diseñador de guis en la vista de diseño del IDE.

En lo que respecta al diseñador de guis, hemos facilitado un archivo .nbm mediante el cual poder instalar un modulo en el IDE de NetBeans con la libreria, los beans con los distintos tipos de gui para la paleta de diseño de componentes y los widgets para el asistente del menu de NetBeans. Mediante estos mecanismos, el diseñador podra diseñar las guis reusables de la misma forma que lo haria con cualquier otro tipo de componente.

Por otra parte, se han tenido que habilitar distintos metodos de obligada implementacion por parte del diseñador de guis para que se pudiera acceder a los componentes concretos de la interfaz grafica, ya que todo el desarrollo de la reutilizacion se ha realizado a nivel de gui no de componentes. De esta manera, hemos tenido que imponer la implementacion de ciertos metodos para poder validar, guardar o limpiar los datos de edicion, asi como notificar los cambios a los componentes concretos en el mecanismo de comunicacion entre guis.

Como se especifico en la introduccion, todos estos desarrollos se han realizado teniendo en cuenta que debian ser totalmente transparentes tanto para los desarrolladores como para el usuario de la aplicacion. Este ultimo objetivo se ha alcanzado habilitando los distintos mecanismos de recursion entre guis de forma que la aceptacion, modificacion o cancelacion de la interfaz grafica sea interpretada por parte del usuario como una transaccion atomica, es decir, la edicion se acepta o se rechaza con una sola accion.

Por ultimo hemos desarrollado un programa de pruebas mediante el cual poder mostrar la viabilidad de la implementacion del patron, en el cual se ha intentado amalgamar las distintas posibilidades de combinacion de guis que la implementacion ofrece, asi como mostrar las diferentes formas de articular tanto la validacion y salvaguarda de los datos, como los mecanismos de notificacion y comunicacion entre componentes.

7.2. Trabajos futuros

Todos los desarrollos que se han realizado a lo largo de este proyecto han tenido como finalidad ultima demostrar las posibilidades que ofrece la aplicacion del patron de reusabilidad a la hora de diseñar interfaces graficas de usuario. Su aplicacion facilita la reutilizacion y el diseño de nuevas interfaces graficas, de forma que se pueda evitar duplicar grandes cantidades de codigo.

En el caso concreto de nuestra implementacion, podriamos ofrecer a nuestro criterio algunas mejoras o nuevos desarrollos mediante los cuales hacer mas extensible los mecanismos implementados en la biblioteca.

El primer desarrollo seria el mas evidente, se trataria de ampliar el numero de tipos guis que ofrece la biblioteca, añadiendo nuevas subclases a la clase base con los distintos tipos de gui a generar, o incluso derivando nuevas subclases de las ya existentes, conformando toda una jerarquia de tipos de gui que los desarrolladores pudieran utilizar a voluntad.

Como desarrollo paralelo y extension a nuestra libreria, podriamos pensar en elaborar diferentes bibliotecas de dominios mas o menos concretos, en los que pudieramos disponer de componentes

reusables ya diseñados, con una estructura que pudiese ser aplicable a determinados contextos, de la misma manera en la que se utilizan los patrones de interfaces graficas de usuario aplicados a aplicaciones web o moviles. Estas bibliotecas, al estar conformadas mediante guis extensibles, podrian ser facilmente adaptadas a determinados dominios y tendrian la facultad de poder combinarse entre ellas sin ningun problema.

Un desarrollo mas especifico en relacion con la insercion de los dialogos en la vista de diseño seria el de implementar en NetBeans un mecanismo para que las guis elaboradas por el diseñador se pudiesen añadir automaticamente como beans a la paleta de componentes con cierta facilidad, a traves de un menu contextual por ejemplo. De esta forma, el diseñador podria disponer directamente en la paleta de las guis reusables que fuera diseñando para poder incorparlas con cierta inmediatez a sus nuevos diseños.

Finalmente, cabria la posibilidad de implementar una adaptacion del patron State de forma que se pudiesen generar guis determinadas para diferentes estados que se pudiesen generar. Se podrian asociar determinadas guis reusables a un objeto y dependiendo del estado de ese objeto, generar una gui especifica para cada estado. Al tener la capacidad de ser extensibles, se podrian integrar dinamicamente, ofreciendo la posibilidad de mostrar distintas combinaciones de guis dependiendo del estado en el que el objeto se encontrara.

Esta facultad de las guis reusables la harian sumamente util a la hora de construir y generar interfaces variables en tiempo de ejecucion.

Bibliography

[NJS, 1989] (1989). New Jersey Style or Worse_is_better. https://dreamsongs.com/Files/LispGoodNewsBadNews.pdf.

[UIP, 2022] (2022). Patrones de diseño de interfaz de usuario web. https://ui-patterns.com/patterns.

[UTe, 2022] (2022). Unit Test Patterns. https://www.codeproject.com/Articles/5772/Advanced-Unit-Test-Part-V-Unit-Test-Patterns.

[Tal, 2022] (2022). www.talent.com. https://es.talent.com/salary?job=programador+java.

[Alexander, 1975] Alexander, C. (1975). *The Oregon Experiment*. Oxford University Press.

[Alexander, 1979] Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

[Alexander et al., 1977] Alexander, C., S.Ishikawa, M.Silverstein, M.Jacobson, I.Fiksdahl-King, and S.Angel (1977). *A Pattern Language*. New York: Oxford University Press.

[A.Salingaros,] A.Salingaros, N. Some Notes on Chsristopher Alexander. https://applied.math.utsa.edu/~yxk833/Chris.text.html.

[Beck and Cunnigham, 1987] Beck, K. and Cunnigham, W. (1987). Using Pattern Languages for Object-Oriented Programs. (Report No CR 87-43).

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns.*, volume 1. John Wiley & Sons.

[Coplien, 1994] Coplien, J. (1994). A Generative Development-Process Pattern Language, Proceedings of PLOP. page 183–237.

[D.McIlroy, 1968] D.McIlroy (1968). Mass produced software components. Software Engineering Concepts and Techniques. page 88–98. NATO Conference on Software Engineering.

[Fowler, 1996] Fowler, M. (1996). *Analysis Patterns. Reusable Object Models*. Addison-Wesley Professional.

[Frakes and Terry, 1996] Frakes, W. and Terry, C. (1996). Software Reuse: Metrics and Models. ACM Computing Survey. 28(2).

[Frantz and Corchuelo, 2007] Frantz, R. and Corchuelo, R. (2007). Integración de Aplicaciones. In *Proceedings of the Zoco07 Workshop*, page 65–75.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Jones, 1984] Jones, T. (1984). Reusability in programming. A survey of the state of the art. 10(5):488–494.

[.Krasner and Pope, 1988] .Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, page 26–49.

[Liskov, 1988] Liskov, B. (1988). Data Abstraction and Hierarchy.

[Meszaros, 1994] Meszaros, G. (1994). Pattern: Halfobject + Prototocol (HOPPI, Proceedings of PLoP. page 129–132.

[Meyer, 1997] Meyer, B. (1988,1997). *Object-Oriented Software Construction*. Prentice-Hall.

[Mili, 1995] Mili, H. (1995). Reusing Software: Issues and Research Directions. IEEE Transactions of Software Engineering.

[Moreno and Sánchez-Segura, 2003] Moreno, A. M. and Sánchez-Segura, M. (2003). Patrones de Usabilidad: Mejora de la Usabilidad del Software desde el Momento Arquitectónico. In *JISBD*, page 117–126.

[Prieto-Diaz, 1993] Prieto-Diaz, R. (1993). Status Report: Software Reusability. IEEE Sortware.

[Suárez and Gutiérrez, 2016] Suárez, J. M. and Gutiérrez, L. E. (2016). Tipificación de dominios de requerimientos para la aplicación de patrones arquitectónicos. *Información tecnológica*, 27(4):193–202.

[Wheeler, 2004] Wheeler, D. A. (2004). SLOCCount User's Guide. `https://dwheeler.com/sloccount/sloccount.html`.

Anexo A

Especificacion de actores

En esta seccion desarrollamos la especificacion de los diversos actores hallados durante la fase de analisis del proyecto.

Figure A.1: Actor *Usuario* en el diagrama de casos de uso

[Usuario]

Actor	Usuario	**Identificador:** US
Descripción	Cliente de la aplicación que utilizara lal GUIs reusables a traves de su interfaz grafica.	
Características	Es el destinatario final de todo el proceso de desarrollo, tanto de la interfaz gráfica como de la aplicación en si. Su función es interactuar con la interfaz de una manera transparente y amigable, utilizando las GUIs reusables de la misma forma que utilizaria otro tipo de guis, sin encontrar ninguna diferencia entre ellas.	
Relación	Actores con que se relaciona: -Aplicacion: es quien va a hacer uso de ella directamente. -Con el resto de actores se relaciona de una manera indirecta y secundaria, ya que el resto de actores realizan sus funciones con el objetivo ultimo de satisfacer al usuario.	
Referencias	Utiliza la librería a través del uso de la interfaz gráfica de la aplicación. CU0, AP	

Figure A.2: Actor *Aplicacion* en el diagrama de casos de uso

[Aplicacion]

Actor	Aplicación	**Identificador:** AP
Descripción	Aplicación diseñada por el programador de aplicaciones donde se utilizaran las GUIs reusables en su interfaz de usuario.	
Características	Se trata de una jerarquía de clases vinculadas a un modelo de datos y a una interfaz de usuario diseñada mediante la librería JGuiExtensible. En la aplicación se generan diferentes tipos de GUIs vinculadas unas con otras a través del patrón de reusabilidad.	
Relación	Actores con que se relaciona: -Programador de Aplicaciones: el programador de aplicaciones utiliza todos los diálogos que le facilitan por un lado la librería, diálogos vacíos, y por otro el diseñador de GUIs, diálogos diseñados, para elaborar la interfaz de usuario de la aplicación.	
Referencias	Es el entorno de aplicación y pruebas de la librería. CU0, CU4, AP, US	

Figure A.3: Actor *Programador de Aplicaciones* en el diagrama de casos de uso

[Programador de Aplicaciones]

Actor	Programador de Aplicaciones	**Identificador:** PA
Descripción	Diseñador de las partes del modelo y el controlador en la arquitectura MVC.	
Características	Se encarga del diseño del acceso a los datos de la aplicación y la gestión de los eventos del usuario respecto a los datos del modelo. Solicita y utiliza las GUIs facilitadas por el diseñador de GUIs. Pide GUIs vacías a través de la interfaz publica de la librería JGuiExtensible. Envuelve otras GUIs y componentes con diálogos vacíos que solicita a través de la interfaz publica de la librería JGuiExtensible. Utiliza la vista de codificación del IDE.	
Relación	Actores con que se relaciona: -Diseñador de GUIs: le solicita GUIs ya diseñadas para incorporarlas al diseño de la aplicación. -Librería JGuiExtensible: le solicita a la librería diálogos vacíos para envolver GUIs. Utiliza sus métodos públicos para añadir GUIs hijas a los diálogos padre. -Aplicación: desarrolla todo el modelo de la aplicación e integra las interfaces de usuario facilitadas por el diseñador de GUIs.	
Referencias	Utiliza la librería directamente y a través de las GUIs diseñadas facilitadas por el diseñador de GUIs. CU1, CU2, CU4, DG, AP	

Figure A.4: Actor *Diseñador de Guis* en el diagrama de casos de uso

[Diseñador de Guis]

Actor	Diseñador de GUIs	**Identificador:** DG
Descripción	Diseña la parte de la vista del esquema de diseño ModeloVistaControlador.	
Características	Diseña GUIs reusables para el diseñador de aplicaciones mediante la librería JGuiExtensible. Diseña las interfaces gráficas de usuario a través de las herramientas facilitadas por el entorno gráfico. Utiliza la librería directamente a través del asistente de plantillas del IDE o mediante la paleta de componentes. Utiliza la vista de diseño del IDE.	
Relación	Actores con que se relaciona: -Programador de aplicaciones: le facilita las GUIs reusables ya diseñadas para incorporarlas al diseño de la aplicación. -IDE: pide diálogos vacíos y diseña GUIs utilizando las herramientas del editor gráfico entre las que se encuentran las GUIs reusables. Implementa en el proceso los métodos que la librería le obliga a completar.	
Referencias	Se relaciona con la librería a través del IDE. Es el único que puede derivar clases de los distintos tipos de diálogos de la librería. CU2, CU3, CU4, CU5, CU6, NB, PA	

Figure A.5: Actor *IDENetBeans* en el diagrama de casos de uso

[IDENetBeans]

Actor	IDE NetBeans	**Identificador:** NB
Descripción	Entorno de desarrollo en el que se integra la librería JGuiExtensible.	
Características	Es el entorno en el que se establecen las relaciones entre la librería y el diseñador de GUIs. La librería se integra en el IDE como un modulo o plugin de forma que el uso de esta sea transparente para el diseñador de GUIs.	
Relación	Actores con que se relaciona: -Diseñador de GUIs: el IDE le facilita los diálogos de las GUIs reusables a traves del asistente de plantillas y de la paleta de componentes. El diseñador de GUIs completa a traves del IDE los metodos que la librería le obliga a implementar. -Programador de Aplicaciones: puede utilizar su entorno para el desarrollo de la parte del modelo y controlador del patron MVC, pero el programador podria utilizar en otros entornos los dialogos del diseñador.	
Referencias	Es el entorno donde se elaborara tanto el diseño de las interfaces como el de la librería. CU3, CU5, DG, PA	

Anexo B

Especificacion de Casos de Uso.

En esta seccion vamos a desarrollar la especificacion cada uno de los casos de uso que aparecen en el diagrama 3.1 del capitulo 2 .

Figure B.1: CU0: EditarGUI

[CU0: Editar GUI]

Caso de Uso	Editar GUI	**Identificador:** CU0
Actores	US y AP	
Tipo	Tipo de caso de uso: primario	
Referencias		
Precondición	PA ha programado previamente la interfaz grafica de AP.	
Postcondición	El usuario introduce todos los datos que se le solicitan a través de la GUI. Todos los datos son validados y guardados.	
Descripción	US utiliza la interfaz diseñada por PA para AP.	
Resumen	US interactua con la aplicación a través de la GUI, diseñada con diálogos reusables e implementada por PA en el contexto de la aplicación. US edita las GUI reusables de la interfaz como si se tratara de una normal. Con solo apretar un boton al final de la edición, se aceptan o se rechazan todos los datos introducidos en todos los diálogos de la GUI reusable.	

Curso Normal

Nro.	**Ejecutor**	**Paso o Actividad**
1	US	US introduce los datos en los diferentes cuadros de diálogo de las guis de la interfaz
2	AP	AP comprueba que los datos sean validos.
3	AP	La aplicación acepta la edición y registra los datos introducidos.

Cursos Alternos

Nro.	**Descripción de acciones alternas**
1.1	US sobreescribe los datos introducidos anteriormente: la ultima edición prevalece y sera la que se registre.

Nro.	**Descripción de acciones alternas**
2.1	Los datos introducidos no son validos: AP lanza un mensaje de aviso y borra los datos erroneos. Vuelve al paso 1 del curso normal.

Figure B.2: CU1: Añadir dialogo diseñado

[CU1: Añadir Dialogo Diseñado]

Caso de Uso	Añadir dialogo diseñado.	**Identificador:** CU1
Actores	PA y AP	
Tipo	Tipo de caso de uso: primario	
Referencias	CU2, CU3, CU4	
Precondición	DG tiene que haber diseñado previamente una o varias GUIs utilizando los diálogos reusables de la librería. PA tiene que solicitar previamente un dialogo diseñado a DG (CU2)	
Postcondición	Se genera un nuevo dialogo reusable y se incorpora a la interfaz de usuario de la aplicación.	
Descripción	PA añade en AP un dialogo diseñado por DG	
Resumen	El programador de aplicaciones añade un dialogo desarrollado previamente por el diseñador de GUIs a la jerarquia de clases de la interfaz de usuario de la aplicación.	

Curso Normal

Nro.	**Ejecutor**	**Paso o Actividad**
1	PA	PA solicita a DG un dialogo reusable previamente diseñado
2	PA	PA incorpora el dialogo a la interfaz de AP

Cursos Alternos

Nro.	**Descripción de acciones alternas**
1.1	No existe ningún dialogo diseñado. PA puede crear un dialogo vacio (CU3) y añadirlo a AP (CU4) y trabajar con otros componentes de la librería Swing.

Figure B.3: CU2: Pedir dialogo diseñado

[CU2: Pedir dialogo diseñado]

Caso de Uso	Pedir dialogo diseñado	**Identificador:** CU2
Actores	PA y DG	
Tipo	Tipo de caso de uso: primario	
Referencias	CU1, CU3	
Precondición	DG ha diseñado previamente el dialogo en el IDE. DG ha facilitado previamente a PA los métodos de inicializacion de las GUIs diseñadas.	
Postcondición	Se genera un dialogo diseñado. PA dispone de una GUI para incorporarla a la interfaz de AP.(CU1)	
Descripción	PA solicita a DG un dialogo diseñado.	
Resumen	PA solicita y utiliza, mediante los métodos facilitados por DG, el dialogo diseñado y generar una nueva GUI en la interfaz de AP. PA puede utilizarla directamente, incorporar el dialogo a otros ya diseñados o envolverla y añadir nuevos componentes para generar la nueva GUI.	

Curso Normal

Nro.	**Ejecutor**	**Paso o Actividad**
1	PA	PA inicializar y genera el dialogo diseñado por DG.
2	PA	Una vez disponibles, el diálogo se incorpora a la interfaz de AP (CU1)

Cursos Alternos

Nro.	**Descripción de acciones alternas**
2.1	Antes de incorporarlo a la aplicación, se le añaden otros diálogos diseñados (CU1) o se envuelven en otros (CU4)

Figure B.4: CU3: Crear dialogo

[CU3: Crear dialogo]

Caso de Uso	Crear dialogo	**Identificador:** CU3
Actores	PA, DG	
Tipo	Tipo de caso de uso: primario	
Referencias	CU4	
Precondición	El PA y el DG deben poder acceder a la biblioteca. PA sabe como acceder a la factoría y conoce cuales deben ser los argumentos que necesita introducir para crear uno de los tres tipos de dialogo. DG debe disponer de los diálogos a través de la vista de diseño del IDE	
Postcondición	El PA utiliza los diálogos vacíos creados para envolver otros diálogos, añadir componentes, generar nuevas GUIs e incorporarlas a la interfaz de AP. (CU4). El DG ha arrastrado el tipo de dialogo que necesita desde la paleta de componentes del IDE o lo ha creado desde el asistente de plantillas.	
Descripción	El PA o el DG crean un dialogo	
Resumen	El PA accede a la factoría de la librería y solicita un dialogo vacío. Debe introducir como argumentos el tipo de la factoría y si el dialogo debe llevar botones o no. La factoría creara un dialogo del tipo y forma indicados. El DG dispone de los distintos tipos de dialogo en la paleta de componentes y en el asistente de plantillas del IDE. Cuando lo necesite para su diseño puede crear el dialogo arrastrando el componente al panel o generandolo de inicio desde el asistente.	

Curso Normal PA

Nro.	**Ejecutor**	**Paso o Actividad**
1	PA	El PA introduce el tipo de dialogo vacío que requiere (SIMPLE, TABBED o TREE) en la factoría de la librería.
2	PA	El PA escoge el tipo de wrapper que necesita (con botones o sin botones)

Cursos Alternos PA

Nro.	**Descripción de acciones alternas**
1.1	El PA no introduce correctamente el tipo de dialogo: el sistema muestra un error de sintaxis en el método de la factoría.
2.1	El PA no introduce el booleano que representa el dialogo con o sin botones: el sistema muestra un error de sintaxis en el método de la factoría.

Figure B.5: CU4: Añadir dialogo

[CU4: Añadir dialogo]

Caso de Uso	Añadir dialogo	**Identificador:** CU4
Actores	PA y AP	
Tipo	Tipo de caso de uso: primario	
Referencias	CU1	
Precondición	El PA ha accedido a la factoría de la libreria y ha creado un dialogo vacio o ha pedido a DG un dialogo diseñado.	
Postcondición	El PA incorpora el dialogo a otros diálogos y a la interfaz de AP.	
Descripción	El PA genera una nueva GUI en la AP	
Resumen	El PA mediante la interfaz de la librería añade un nuevo dialogo a la AP envolviendo otros diálogos reusables u otros componentes, y así poder generar nuevas GUIs sin prácticamente tener que codificarlas. También puede añadir diálogos diseñados por DG (CU1) a otros diálogos anteriores para generar una nueva GUI para la AP	

Curso Normal

Nro.	**Ejecutor**	**Paso o Actividad**
1	PA	PA añade el dialogo creado en la AP.

Cursos Alternos

Nro.	**Descripción de acciones alternas**
1.1	PA añade el dialogo a otros diálogos creados previamente.
1.2	Repite el paso anterior tantas veces como sea necesario.
1.3	Vuelve al paso 1 del curso normal.

Figure B.6: CU5: Diseñar dialogo

[CU5: Diseñar dialogo]

Caso de Uso	Diseñar dialogo	**Identificador:** CU5
Actores	DG y NB	
Tipo	Tipo de caso de uso: primario	
Referencias	CU6	
Precondición	DG ha creado un dialogo reusable en el IDE.	
Postcondición	DG ha incorporado al diseño de la GUI distintos widgets entre los que pueden incluirse anidados otros diálogos de la librería.	
Descripción	DG utiliza los diálogos de la biblioteca a través de NB como componentes de diseño.	
Resumen	El DG realiza el diseño de una GUI completa, con todos los componentes de edición que necesite, entre los que se pueden incluir otras GUIs reusables. El DG a través del asistente de plantillas de NB o mediante su paleta de componentes añade los diálogos de la biblioteca al diseño de GUI que va a realizar. Los diálogos se agregan con total transparencia, como si se trataran de un componente mas de diseño de GUIs. Puede verse obligado a implementar ciertos métodos de la librería a la hora de realizar el diseño (CU6).	

Curso Normal

Nro.	**Ejecutor**	**Paso o Actividad**
1	DG	El DG incorpora al panel del diseño que realiza, un componente de la paleta de componentes del IDE. Entre ellos pueden encontrarse los distintos diálogos de la librería.
2	DG	El DG ejecuta el paso 1 tantas veces como sea necesario para realizar el diseño.

Cursos Alternos

Nro.	**Descripción de acciones alternas**
1.1	El DG incorpora un dialogo de la librería como panel de diseño mediante el asistente de plantillas. Vuelve al paso 2 del curso normal.

Nro.	**Descripción de acciones alternas**
1.1	El DG sobreescribe los distintos métodos que la biblioteca le obliga a implementar. (CU6).Vuelve al paso 2 del curso normal.

Figure B.7: CU6: Forzar implementacion

[CU6: Forzar implementacion]

Caso de Uso	**Forzar implementacion**	**Identificador:** CU6
Actores	DG y NB	
Tipo	Tipo de caso de uso: primario	
Referencias	CU5	
Precondición	DG diseña una GUI con las clases de la libreria en NB.	
Postcondición	El DG completa el diseño de la GUI (CU5) sobreescribiendo ciertos métodos que la librería le obliga a implementar.	
Descripción	El DG implementa ciertos métodos de la GUI	
Resumen	La biblioteca impone al DG que sobreescriba ciertos métodos para la validación y actualización de los datos de edición que no pueden implementarse en la propia librería. Estos métodos deben ser completados a la hora de diseñar la GUI. Por esta razón obliga a DG a sobreescribirlos mediante lanzamiento de excepciones.	

Curso Normal

Nro.	**Ejecutor**	**Paso o Actividad**
1	DG	El DG sobreescribe el método de la implementación de la librería en el diseño de GUI que esta realizando.
2	DG	El DG ejecuta el paso 1 tantas veces como sea necesario para completar el diseño.

Cursos Alternos

Nro.	**Descripción de acciones alternas**
1.1	El DG no implementa uno de los métodos señalados por la librería: el sistema lanza una excepción.

Anexo C

Manual de uso de la libreria.

C.1. Interfaz publica.

Como ya se ha comentado en diversas partes del documento, el desarrollador de aplicaciones dispone de varios metodos de acceso publico en la libreria mediante los que puede tanto crear dialogos vacios de cualquiera de los distintos tipos disponibles como añadir dialogos hijos o listas de dialogos hijos a una gui del tipo que sea.

C.1.1. Creacion de dialogos.

JFactory.getInstance(): para la creacion de guis, la libreria ofrece una clase-factoria mediante la cual poder crear cualquiera de los tipos de guis disponibles. La clase, denominada JFactory, esta implementada mediante el patron Singleton, por lo que solo existira una instancia de ella. Para acceder a dicha instancia se ha creado el metodo *getInstance()* que devuelve la instancia de la factoria, y sino se ha creado previamente, la crea.

createDialog (JTipoGui tipoGui, boolean withButtons): una vez conseguida la instancia de la factoria, el metodo para crear una instancia del tipo de gui escogido se denomina *createDialog(JTipoGui tipoGui, boolean withButtons*). Este metodo crea una instancia del tipo introducido en el primer parametro *tipoGui* , cuyas variantes estan determinadas por la clase Enum JTipoGui { SIMPLE, TABBED, TREE}, que determinan el tipo de gui a crear, simple, de navegacion mediante pestañas o mediante vista de arbol. El parametro withButtons es un booleano que determina si el dialogo se debe crear con los botones de edicion, Ok y Cancel (true) o sin ellos (false).

De esta manera, si el programador de aplicaciones desea crear un dialogo, primero debera obtener la instancia de la factoria y luego determinar su eleccion en los parametros del metodo createDialog(). Asi, por ejemplo, para crear un dialogo de vista en arbol con botones, la sentencia que debe introducir en el codigo seria:

JFactory.getInstance().createDialog(TREE, true);
si quisiera crear uno de navegacion mediante pestañas y sin botones de aceptacion de edicion seria: *JFactory.getInstance().createDialog(TABBED, false).*

C.1.2. Añadir dialogos.

addJGui(JGuiExtensible gui): mediante este metodo, el desarrollador podra añadir una gui hija de cualquiera de los tipos disponibles al dialogo con el que este trabajando. El parametro del metodo sera la gui hija que se agregara al dialogo sobre el cual se realiza la llamada al metodo.

addJGuiChildrenList (List<JGuiExtensible>_childrenList): este metodo funciona de manera similar al metodo anterior y permite introducir una lista de guis hijas, que se insertan al mismo nivel en la gui padre. El parametro del metodo es la lista de guis hijas que se agregaran al dialogo sobre el cual se realiza la llamada al metodo.

Existen toda otra serie de metodos de los que dispone el desarrollador y que forman parte de la interfaz publica de la clase JPanel de la biblioteca Swing que es de donde hereda toda la jerarquia de guis reusables. El desarrollador podra disponer de toda la API de esta clase para realizar las modificaciones que considere oportunas en las guis reusables para adaptarlas a sus desarrollos. En nuestro caso, destacaremos el metodo *setName()* que se utiliza con cierta asiduidad en el programa de pruebas para determinar los nombres de las ventanas o de las pestañas en el caso de los dialogos de navegacion mediante pestañas.

C.2. Instalacion del plugin de NetBeans.

La instalacion tanto de la libreria como de los templates para el asistente de NetBeans y los widgets para la paleta de componentes se instalan directamente a traves del archivo .nbm que se facilita con el resto del proyecto.[1]

Para instalar el modulo no hay mas que abrir el IDE NetBeans, y en el menu ->Tools -> Plugins. Una vez accedido a la ventana de Plugins, en la seccion Downloaded podremos ver un boton para añadir plugins (Add Plugins), accederemos a traves de el al sistema de ficheros del sistema y tendremos que navegar hasta donde tengamos descargado los archivos del proyecto y seleccionar el archivo *org.myorg.jguiextensiblemodule.nbm* . Una vez seleccionado, nos aparecera una ventana como esta:

[1] El mismo archivo se puede encontrar en el codigo fuente del modulo: JGuiExtensibleModule

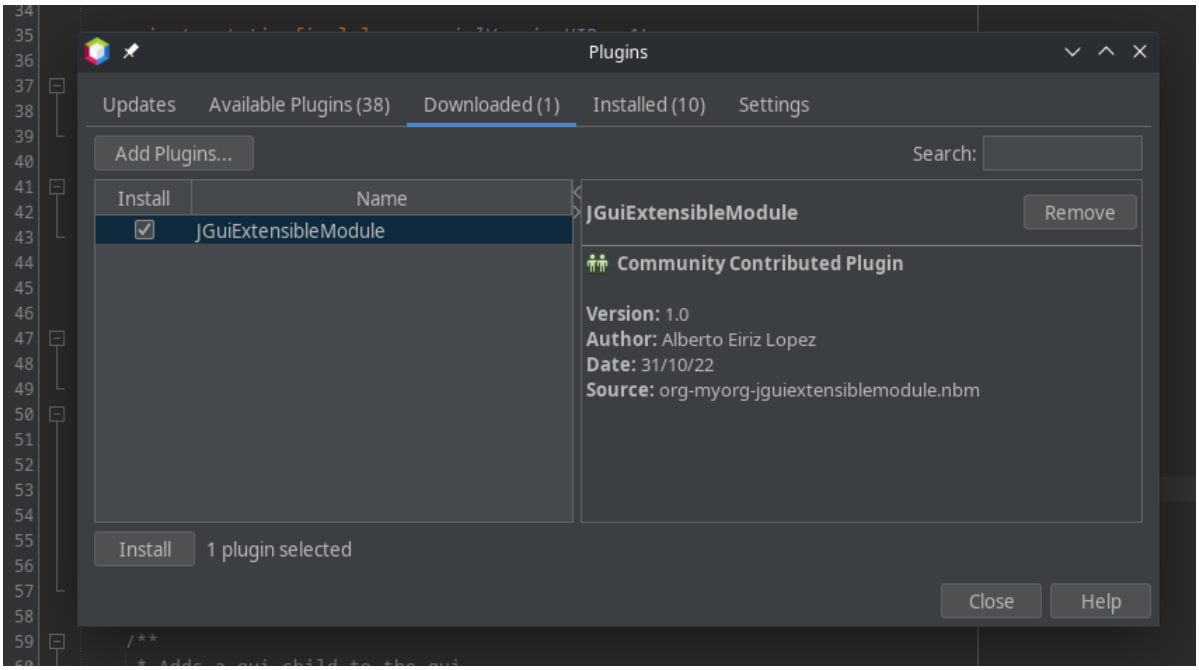


Figure C.1: Screenshot de la ventana para la instalacion del Plugin

Solo tendremos que darle a Install, confirmar confirmar el modulo que queremos instalar, y aceptar los derechos de la licencia GPL para terminar la instalacion. Finalmente, NetBeans nos lanzara un aviso de que se esta intentando instalar un modulo *Unsigned* es decir que no dispone de un certificado valido para NetBeans. La ventana que nos apareceria seria esta:

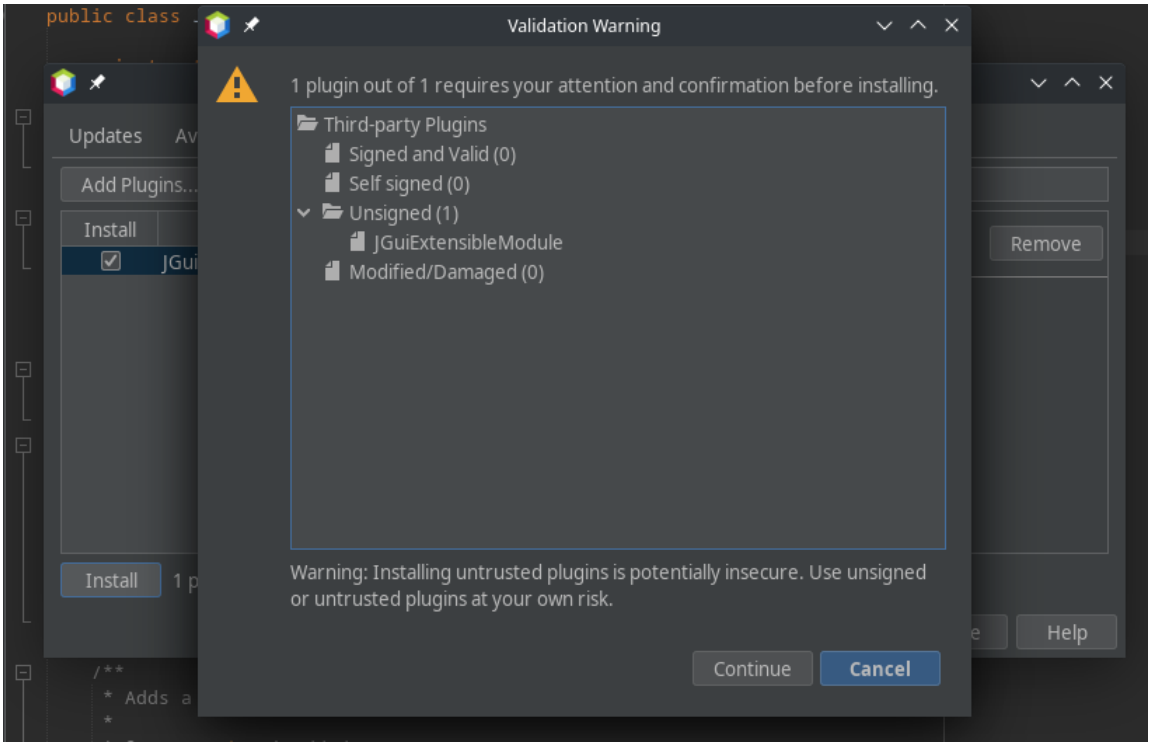


Figure C.2: Aviso de NetBeans de instalacion de modulo sin certificado

Deberemos simplemente continuar con la instalacion. El modulo esta diseñado para reiniciar el IDE cada vez que se instale o desinstale del entorno, y es recomendable hacerlo para no encontrarnos con problemas con la instalacion de la libreria y los componentes. Asi que nos pedira reiniciar el IDE, lo aceptaremos, y una vez reiniciado podremos disponer tanto de la biblioteca como de los templates en el menu de inicio y los widgets en la paleta de componentes de la vista de diseño.

Para comprobarlo solamente tenemos que ir a Tools -> Libraries y en el catalogo de bibliotecas disponibles en el entorno podremos ver nuestra libreria: JGuiExtensible. Para acceder a los templates tenemos que ir a New File -> Other y nos apareceran los tres tipos de guis disponibles: JGuiSimple, JGuiTabbed y JGuiTree. Tambien podremos acceder a estos templates a traves del menu contextual New -> Other -> Other.

Finalmente, si accedemos a la vista de diseño del editor grafico del entorno, podremos ver como entre las diferentes secciones de la paleta de componentes hay una denominada JGuiExtensible donde se encuentran los distintos beans de guis reusables: JGuiSimple, JGuiTabbed y JGuiTree. Estos elementos se pueden arrastrar de la paleta y soltar sobre el panel de diseño para trabajar directamente con ellos como si se tratara de un componente mas de diseño. Igualmente, a traves del menu contextual de la vista de diseño podremos añadir cualquiera de nuestras guis mediante Add from Palette -> JGuiExtensible -> y el tipo de gui que deseemos.

C.3. Look and Feel en la clase MitutoyoApp.

En la clase MitutoyoApp de la aplicacion de prueba, se ha introducido una funcionalidad añadida para acceder a los diferentes *looks & feels* que se suministran mediante la libreria *JTatoo-1.6.13.jar* . Esta libreria dispone de diferentes looks&feels que ofrecen distintos aspectos visuales a los dialogos diseñados en la aplicacion.

La funcion que introduce los distintos looks es la funcion estatica *setMyLookAndFeel()* . Para cambiar el aspecto visual de la aplicacion solo hay que modificar la variable estatica *LOOKANDFEEL,* que se pasa en la funcion al metodo setLookAndFeel de UIManager. En el programa esta variable se encuentra configurada como *"com.jtattoo.plaf.mcwin.McWinLookAndFeel"* , tan solo habria que cambiar el string de referencia a alguna de las distintas clases de looks and feels que aparecen al desplegar la libreria.

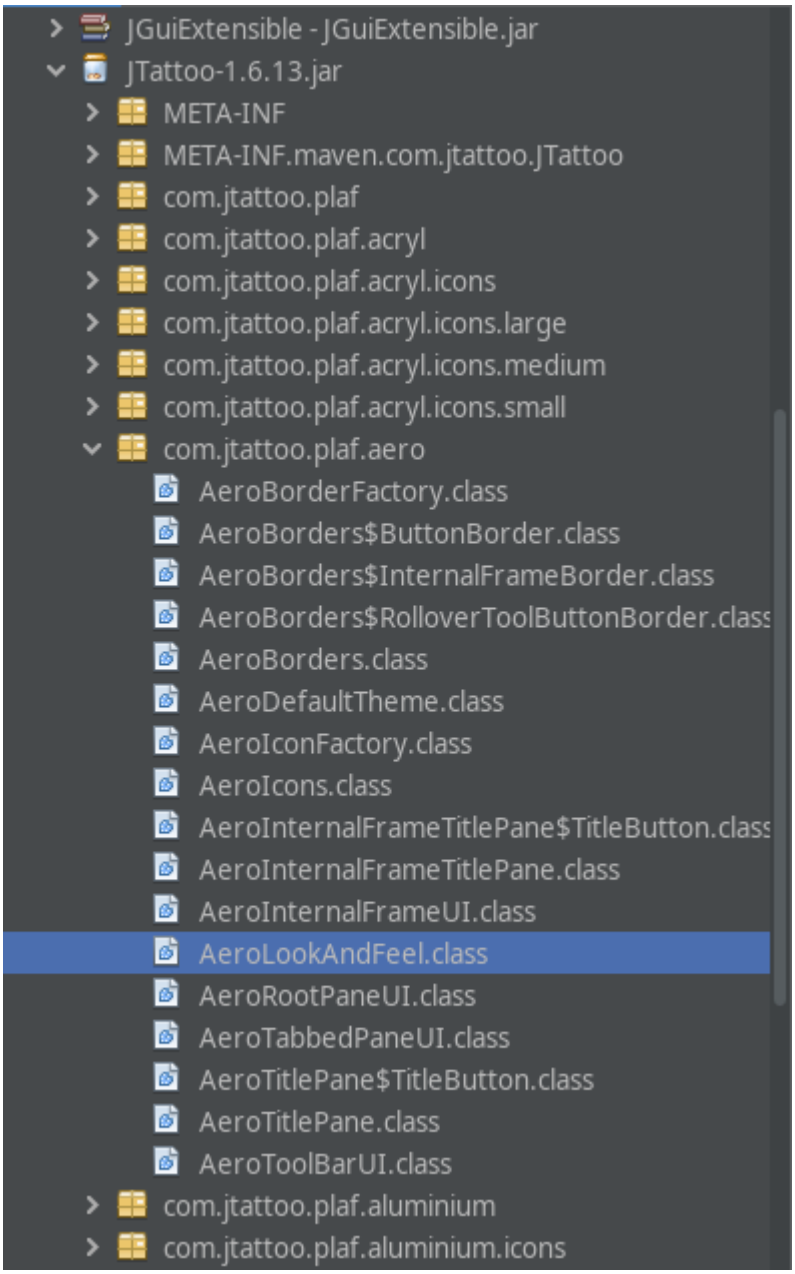


Figure C.3: Lista de paquetes de la libreria JTattoo-1.6.13

De esta manera, para modificar el aspecto de los dialogos al del paquete com.jtattoo.plaf.aero solo habria que modificar la variable de la siguiente manera:

$$LOOKANDFEEL = \text{“com.jtattoo.plaf.aero.AeroLookAndFeel”};$$

y de esta forma con cualquiera de las distintas clases de LookAndFeel que dispone la libreria.

Cabria mencionar que algunos de ellos quizas no funcionen de manera adecuada o cambien de alguna manera el funcionamiento normal de los dialogos, como por ejemplo los limites de minimos en el redimensionamiento u otros factores visuales mas evidentes.

Esta funcionalidad, a pesar de no tener mucha relevancia respecto a los objetivos intrinsecos del desarrollo del programa de pruebas, se ha introducido para enriquecer el aspecto de la aplicacion y hacerla algo mas atractiva de cara a su visualizacion.

Anexo D

Metricas y costes.

Para hacer una comparativa de los costes presupuestados durante el analisis previo 3.1, vamos utilizar una herramienta de metrica de software denominada SLOCCount [Wheeler, 2004] que nos proporcionara una serie de datos con los que poder realizar una comparativa de costes, una vez finalizado el proyecto, tanto economicos como de esfuerzo de desarrollo y de inversion de tiempo de tal manera que podamos tener una referencia para valorar las estimaciones realizadas.

Hay que tener en cuenta que la de SLOCCount sera una estimacion de minimos en un contexto de desarrollo optimo, siempre considerando que el desarrollo del proyecto se ha realizado por una sola persona, en una franja de tiempo muy distinta a la de los datos obtenidos, y aplicando un factor multiplicativo sobre gastos mucho mas amplio, mas del doble. Estas estimaciones no tienen caracter exhaustivo ni pretenden serlo. A la hora de realizar una valoracion real de los posibles costes de desarrollo de un proyecto de software, se deberian tener en consideracion muchas mas variables de las que se utilizan, y existirian muchas otras metricas que deberian ponerse en valor antes de concluir si los costes de un proyecto son asumibles o no.

SLOCCount es una suite de programas de codigo abierto desarrollados por David A. Wheeler para contar lineas de codigo fuente fisicas (SourceLinesOfCode) en sistemas de software potencialmente grandes. Sloccount puede contar lineas de codigo de un amplio numero de lenguajes, en nuestro caso, las lineas de codigo que tendra en cuenta seran basicamente lineas de codigo de archivos .java, eludiendo archivos xml y otro tipos de archivos que utiliza el IDE NetBeans para la gestion de proyectos. Por lo tanto, la metrica que se utilizara para la comparacion seran la cantidad de lineas de codigo escrito en Java tanto en lo que respecta al codigo de la biblioteca, como al modulo de NetBeans y al programa de prueba.

Por otra parte, a la hora de poner en valor los datos, tenemos que tener en consideracion una serie de factores:

- la estimacion de costes incluyen diseño, codificacion, testing y documentacion, asi como una tasa (overhead) de gastos generales (instalaciones, equipos, contabilidad, etc...) que se ha establecido en 2.4 como tasa representativa por defecto de los gastos generales sostenidos por

una empresa tipica de desarrollo de software [Wheeler, 2004]. En el caso de nuestro analisis, la tasa era de 1.1%, aunque contabilizamos una serie de gastos que aqui se podrian aplicar a dicha tasa.

- las unidades de esfuerzo se estiman en persona/año y persona/mes entre parentesis, y las unidades de tiempo estimado en años y meses entre parentesis.
- la formula de costes estimado es simplemente la cantidad de esfuerzo, multiplicada por el salario anual promedio y por el factor de gastos generales. (effort * personcost * overhead)
- el salario anual considerado ha sido el mismo que en el analisis de 31500 euros anuales[Tal, 2022].
- para otras consideraciones de la metrica como el modelo Basic COCOMO para las estimaciones de esfuerzo y tiempo vease [Wheeler, 2004]

Los datos obtenidos al correr el programa son:

```
SLOC    Directory       SLOC-by-Language (Sorted)
3240    MitutoyoApp     java=3240
340     JGuiExtensible  java=340
84      JGuiExtensibleModule java=84


Totals grouped by language (dominant language first):
java:          3664 (100.00%)




Total Physical Source Lines of Code (SLOC)                = 3,664
Development Effort Estimate, Person-Years (Person-Months) = 0.78 (9.38)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.49 (5.85)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 1.60
Total Estimated Cost to Develop                           = $ 59,116
 (average salary = $31,500/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to
redistribute it under certain conditions as specified by the GNU GPL license;
see the documentation for details.
Please credit this data as "generated using David A. Wheeler's 'SLOCCount'."
[a31r1z@a-3lpc Proyecto]$
```

Figure D.1: Estimacion de costes del proyecto con SLOCCount

Como se puede observar en la salida del programa, el numero total de lineas de codigo en Java es de 3664, el esfuerzo necesario en persona-meses es de 9.38, o 0.78 persona/año el coste de tiempo

estimado es de 5.85 meses, el numero de desarrolladores 1.60, y el coste de desarrollo estimado, teniendo en cuenta un salario anual medio de 31.500 y un factor de gasto de 2.40, es de 59.116.

Teniendo en cuenta que el coste de tiempo estimado es practicamente el mismo que el realizado en el analisis previo, hay que tener en cuenta que en SlocCount se consideran como numero medio de desarrolladores 1.6 con lo que a los costes de nuestro analisis habria que sumarle el de otro desarrollador. Por otro lado, el factor multiplicativo es casi el doble, aun sin incorporar los gastos de amortizacion y de suministros supone un margen bastante mas amplio que el realizado en nuestro presupuesto inicial.

En base a los datos obtenidos de SlocCount, deberiamos tener en cuenta la posibilidad de revisar la estimacion de costes aunque este programa se basa unicamente en la cantidad de lineas de codigo utilizado, y quizas esa no sea una manera muy adecuada para valorar programas desarrollados mediante la orientacion a objetos.