TÍTULO
DBCASE 4.0

TRABAJO FIN DE GRADO
CURSO 2023-2024

AUTOR
IVÁN PISONERO DÍAZ
JAVIER DE VICENTE VÁZQUEZ

DIRECTOR
FERNANDO SÁENZ PÉREZ

GRADO EN INGENIERÍA DEL SOFTWARE
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

DEDICATORIA

A todas las personas que nos hemos
cruzado durante este viaje

AGRADECIMIENTOS

Quiero agradecer a mi familia y a mis amigos por su apoyo, en especial a mi madre.

Iván

Quiero agradecer a mis padres, a todos mis amigos y amigas y sobre todo a Victoria por estar siempre ahí aguantándome y soportando mis quejas.

Javier

También queremos agradecer a Fernando por su tiempo invertido en este proyecto y por haber respondido siempre al momento todas las dudas que nos han ido surgiendo.

Resumen

DBCASE 4.0

El proyecto detallado a continuación corresponde con el Trabajo de Fin de Grado de los alumnos Iván Pisonero Díaz y Javier de Vicente Vázquez durante el curso 2023/2024, bajo la tutela del profesor Fernando Sáenz Pérez. DBCASE 4.0 es la última iteración de un proyecto comenzado en 2008 y que es una aplicación de escritorio basada en Java que permite el diseño y la implementación de bases de datos relacionales SQL. Tiene una interfaz gráfica intuitiva que permite el diseño de diagramas conceptuales y la generación posterior de un modelo lógico y un modelo físico el cual se puede implementar en una BBDD mediante el conector que tiene integrada la aplicación. En esta memoria se explica detalladamente todo el trabajo realizado para mejorar la aplicación, así como multitud de diagramas UML incluidos en forma de anexo, que junto a la documentación generada en versiones anteriores servirán para ayudar a futuros alumnos a seguir evolucionando y mejorando esta aplicación.

Palabras clave

Java, Bases de datos relacionales, SQL, Modelo Entidad-Relación, Modelo Vista-Controlador, Comando, Refactorización, Maven, Patrones de diseño.

ABSTRACT

DBCASE 4.0

The project detailed below corresponds to the Bachelor Thesis of students Iván Pisonero Díaz and Javier de Vicente Vázquez during the academic year 2023/2024, under the supervision of Professor Fernando Sáenz Pérez. DBCASE 4.0 is the latest iteration of a project started in 2008, which is a Java-based desktop application that allows for the design and implementation of relational SQL databases. It features an intuitive graphical interface for designing conceptual diagrams and subsequently generating a logical model and a physical model that can be implemented in a database using the integrated connector. This report explains in detail all the work done to improve the application, as well as numerous UML diagrams included as annexes, which along with the documentation generated in previous versions will help future students continue to evolve and enhance this application.

Keywords

Java, relational databases, SQL, Entity-Relationship Model, Model View-Controller, Command, Refactoring, Maven, Design Patterns

Índice de contenidos

Índice de figuras

Capítulo 1 - Introducción

DBCASE 4.0 es una aplicación de escritorio enfocada en el diseño e implementación de bases de datos relacionales de SQL. Mediante el diseño de un esquema conceptual genera el modelo lógico y el modelo físico permitiendo también la implementación de este último.

Una base de datos relacional es un tipo de base de datos que organiza la información en tablas relacionadas entre sí. Está basada en el modelo relacional propuesto por Edgar F. Codd en la década de 1970. En este modelo, los datos se almacenan en tablas con filas y columnas. Cada fila representa una entidad específica y cada columna representa un atributo de esa entidad. Las relaciones entre las entidades se establecen mediante la coincidencia de valores en columnas específicas (claves primarias y claves foráneas). En una base de datos relacional, la integridad de los datos se mantiene mediante restricciones como las claves primarias y foráneas, que garantizan la coherencia y la consistencia de los datos. Además, utiliza un lenguaje de consulta estructurado (SQL) para realizar operaciones como insertar, actualizar, eliminar y consultar datos.

En la Facultad de Informática, en los diferentes grados que ofrece, las bases de datos relacionales son usadas en multitud de asignaturas como en *Bases de Datos, Ampliación de Bases de Datos, ingeniería del Software, etc.* Aunque sobre todo este trabajo será útil para los alumnos de *Bases de Datos* ya que es en la asignatura donde se presentan los fundamentos del diseño de bases de datos y donde los alumnos empiezan a diseñar sus primeros diagramas Entidad-Relación y aprenden la sintaxis básica de Oracle SQL

1.1 Motivación

Nuestra principal motivación a la hora de elegir este tema y no otro fue expandir nuestros conocimientos en el diseño de las bases de datos relaciones y aumentar nuestra experiencia en el diseño e implementación de aplicaciones de escritorio basas en Java.

Por otro lado, al ser un TFG iterativo teníamos el reto de continuar lo que otros hicieron años anteriores, empezando en el 2008 con la idea de Yolanda García Ruiz y continuado por Fernando Sáenz Pérez, nuestro tutor. En el 2008 el proyecto se llamaba DBDT (Database Design Tool), que en 2009 cambió el nombre a DBCase (Database Computer-Aided Software Engineering) siendo este la cuarta iteración en versión de escritorio ya que en el año 2020 el proyecto se dividió en la versión de escritorio basada en Java y la versión web basada en HTML, CSS y JavaScript.

Esta herramienta ofrece una interfaz gráfica intuitiva e interactiva que permite a los usuarios diseñar una base de datos relacional mediante un esquema conceptual que sigue el modelo Entidad-Relación. Además, a partir del esquema conceptual creado, muestra las traducciones correspondientes a los modelos lógico y físico. Esto permite corroborar la corrección del diseño del esquema conceptual. También puede generar el código asociado a la creación de cada tabla en un formato compatible con diferentes sistemas de gestión de bases de datos (SGDB), como Oracle, MySQL o Microsoft Access.

Aunque esta herramienta está diseñada para usarse en entornos académicos, ya que facilita a los estudiantes la comprensión de la teoría de bases de datos en casos prácticos, su utilidad no se limita a este ámbito. La capacidad de generar código SQL permite la ejecución de scripts reales en diversos SGDB, lo que la hace valiosa tanto para entornos educativos como profesionales.

1.2 Objetivos

Las tareas más prioritarias de este proyecto son, en primer lugar, las relativas a la corrección de su funcionamiento y, en segundo, a las ampliaciones de funcionalidad para dejarlo en el mismo nivel que DBCASEweb (en particular, añadir las agregaciones y las cardinalidades con dos notaciones 1-N y mín-máx). Finalmente se abordarían las tareas de extensión (análisis de rendimiento, normalización/desnormalización y reingeniería).

1. Revisar la generación del esquema relacional. Hay errores al menos en las claves de las relaciones 1-N. DBCASEweb debería hacerlo bien, pero podría tener aún errores.
2. Terminar la implementación de las agregaciones.
3. Optimización del diseño: análisis de rendimiento (volúmenes, frecuencias, operaciones...).
4. Normalización/Desnormalización.
5. Reingeniería del diseño físico/relacional.
6. Las propiedades marcadas no se ven bien con el fondo oscuro.
7. Si la relación tiene dos o más roles diferentes hacia una misma entidad no le da un nombre diferente a cada campo, sino que los repite.
8. Añadir líneas de alineación para ayudar en la ubicación de los elementos en el diagrama E-R.
9. Permitir desplazar el diagrama pulsando y arrastrando.
10. No aparece la flecha en una relación recursiva con cardinalidad 1.
11. Permitir usar el botón secundario del ratón sobre los arcos para editar sus propiedades.

12. Cuadros de diálogo Yes/No. Permitir usar las teclas Y y N (cuidado en otros idiomas: en español S y N, en ruso: D y N, en francés O ...).
13. Guardar estado de la aplicación (último archivo abierto, idioma, zoom, paneles abiertos, conexiones, ... En general, todo lo que se puede configurar o tener visualizado Cuando se abre de nuevo la aplicación es generalmente para trabajar con el último archivo). Resuelto, excepto el zoom.
14. Help -> Abrir el manual de usuario.
15. Al hacer doble clic sobre un objeto se debe abrir un cuadro que permita modificarlo (prácticamente igual que el que se usó para su creación).
16. Barra de menús: añadir View (Design Panel Zoom (para permitir seleccionar un % como en Word, por ejemplo), Graph Panel Zoom, Graph Panel (para mostrarlo o no), Domain Panel (para mostrarlo o no), Event Panel (para mostrarlo o no)).
17. Añadir barra de iconos (Abrir, Guardar, Cerrar, Imprimir...).
18. Poder usar los cursores para mover los objetos.
19. Permitir guardar e imprimir el registro de la validación.
20. Añadir elementos de menú para todas las acciones. En particular para las acciones del menú contextual (Insertar entidad, ...) y para los botones de la pestaña Code Generation.
21. Nuevo menú Edición: Copiar, Cortar, Pegar, Deshacer y Rehacer. Atajos: Ctrl-C, Ctrl-V, Ctrl-X, Ctrl-Z y Ctrl-Y.
22. Proporcionar ayudas para el dibujo: cuadrícula (mostrar, definir, alinear objetos a la cuadrícula), alinear (arriba, abajo, izda., dcha, centro horizontal, vertical), tirar líneas de referencia (como en visual studio .net), disponer en horizontal, disponer en vertical. Añadir un cuadro de herramientas (parecido a Word), Girar (izda., dcha, 90º, 180º).

23. Hacer zoom en los apartados del esquema lógico y físico. Se podrían cambiar los tamaños de letra usando la rueda del ratón (también se podrían añadir entradas en el menú Ver).
24. Arrastrar y soltar los elementos del panel de la izquierda de diseño del esquema conceptual al lienzo de dibujo.
25. Añadir PostgreSQL, SQL Server y DB2 a los posibles SGBD.
26. Reorganización automática del grafo. Es decir, recolocar automáticamente los objetos gráficos.
27. Permitir definir unicidad sobre más de un atributo, por ejemplo, seleccionando todos los atributos que sean superclave y con el botón derecho seleccionarlo.
28. Restricciones de integridad de tabla como predicado lógico (condición a cumplir) expresado en SQL. Por ejemplo, a+b=10 para el siguiente esquema:
29. Especificar el nivel de zoom al panel del esquema E/R (pequeño, el de la izquierda) y del panel de diseño (grande, el de la derecha). Añadir opciones en la barra de menús (Nivel de zoom y Mostrar/ocultar paneles).
30. Añadir el atajo de teclado Ctrl+Rueda de ratón para el zoom (manteniendo también el que ya existe solo con la rueda de ratón).

Cuando empezamos a revisar el código nos dimos cuenta de que para poder llevar a cabo todas las tareas previamente mencionadas primero teníamos que hacer una refactorización global del código con el fin de hacer más claro, accesible, mantenible, actualizable y eficiente aplicando todos los conocimientos sobre ingeniería del software aprendidos durante el grado. Así que tras hablarlo con nuestro tutor decidimos que la tarea prioritaria de este proyecto iba a ser la refactorización del código y una vez concluida comenzaríamos con los objetivos previamente mencionados en este punto.

Al momento de escribir esta memoria se completó muy satisfactoriamente el objetivo prioritario de la refactorización. Se ha conseguido que el código sea mucho más legible pero lo más importante, mucho más mantenible y actualizable, fragmentando clases que tenían más de cinco mil líneas a unas mucho más manejables, se han creado factorías, contextos, comandos, etc. Por otro lado se ha hecho una reorganización de la estructura de carpetas, se ha migrado el proyecto a *Maven* para poder añadir nuevas librerías de una manera mucho más fácil y se ha trabajado en algunos de los objetivos marcados por nuestro tutor consiguiendo algunos y dejando otros comentados debido a que no se logró completarlos. En la sección **8.1** exponemos una lista completa de los objetivos cumplidos.

1.3 Plan de trabajo

En esta sección se expone el plan de trabajo a seguir para la consecución de los objetivos indicados anteriormente. Lo primero fue contactar con Fernando Sáenz Pérez para tener una primera toma de contactor con él, durante esta reunión nos explicó las bases del proyecto y como teníamos que continuar. Una vez teníamos el código empezamos a analizarlo y comprenderlo y propusimos lo descrito en la sección **1.2**. Lo primero que se hico fue migrar el proyecto a Maven para facilitar la importación de librerías y poder actualizarlas, lo siguiente fue la estandarización de los nombres de las clases, la ordenación de los recursos gráficos y de los paquetes de idiomas. Una vez terminadas estas tareas comenzó refactorización del código, explicada en detalle el **Capítulo 4**. Mientras sucedía esta refactorización también se fue avanzando con la lista de objetivos explicada anteriormente y con la escritura de esta memoria, aunque como está escrito más adelante, este trabajo fue muy lento y costoso.

Exponemos en la ilustración 1 un diagrama de Gantt de nuestro proyecto.

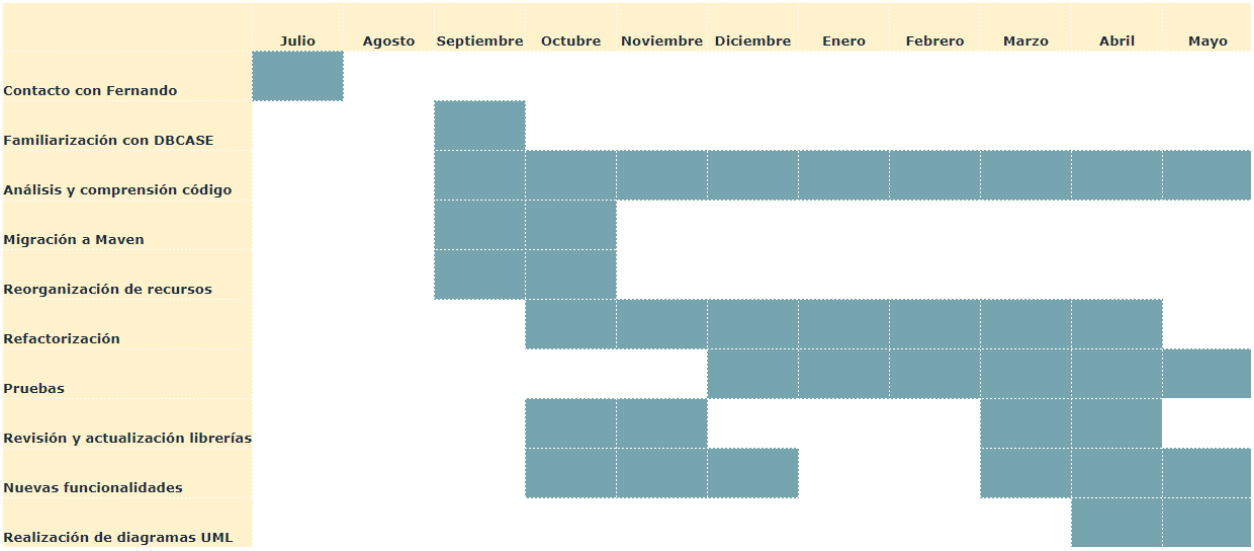

Ilustración 1: Diagrama de Gantt del proyecto

1.4 Estructura de la memoria

A lo largo de esta memoria describimos todas las tareas acometidas. Exponemos primero, en el **Capítulo 2**, la realización de diagramas UML, cuya exposición total se pueden ver en los anexos de este documento. Describimos a continuación, en el **Capítulo 3**, todo lo relativo a la migración a Maven y a la reorganización de los recursos. El capítulo más extenso de esta memoria es el **Capítulo 4** en el que se explica en veintidós apartados todo el proceso de refactorización del código. A continuación, siguen tres capítulos, **Capítulo 5**, **Capítulo 6** y **Capítulo 7**, describiendo otras tareas que se hicieron para terminar con las conclusiones, **Capítulo 8**, en las que damos nuestra opinión sobre el resultado obtenido y explicamos posibles cambios de cara al futuro que mejorarían la aplicación.

Capítulo 2 - Modelo y diagramas UML

Usando todo lo que aprendimos acerca de la aplicación (y también descubriendo muchas cosas nuevas sobre ella), creamos diagramas de clases UML para todos los paquetes y sub-paquetes de la aplicación, de esta forma dotamos al proyecto de una buena herramienta para facilitar la comprensión del código y del funcionamiento interno de la aplicación, así como para apoyar futuras ampliaciones y futuros desarrollos, dada la importancia bien conocida de la documentación en estos ámbitos. Cabe destacar que estos diagramas forman parte de un proyecto de modelado UML en la herramienta ***IBM RSAD***, que permite la modificación / ampliación de cualquiera de los diagramas fácilmente, así como la creación de nuevos elementos (diagramas, clases UML…). Usamos la transformación Java a UML que ofrece IBM RSAD para transformar las clases Java a clases UML, que además nos generó las relaciones de pertenencia-a-paquete y generalización.

Para la mayoría de los paquetes hay un único diagrama de clase, pero en algunos casos, como los servicios, hay varios porque en un diagrama se consolidaba demasiada información y eso lo hacía más difícil de interpretar. En los diagramas no aparecen los parámetros ni los tipos de retornos de las funciones, y en ocasiones tampoco los atributos de la clase, porque hacía el diagrama demasiado grande, pero eso se puede cambiar fácilmente en IBM RSAD, porque la información sí que está contenida en las clases UML. Para las clases que no pertenecen al paquete al que corresponde el diagrama, pero que aparecen en el diagrama, las hemos representado únicamente con el nombre, y es en el diagrama correspondiente al paquete al que pertenecen donde aparecen con toda la información.

Además, consideramos que era una buena idea usar diagramas de secuencia para mostrar cómo interactúan las clases y las capas de la aplicación en un caso de

uso específico, con el objetivo de mostrar con todo detalle y con la máxima claridad posible qué es lo que hace la aplicación en esa operación. Elegimos el caso de uso correspondiente a editar el campo 'Not null' de un atributo. Creamos entonces los diagramas de secuencia que representan las interacciones entre las clases, divididos por capas. Cabe destacar que el diagrama de secuencia de tratarContexto también interviene en este caso de uso.

Exponemos a lo largo del texto los diagramas realizados, introduciendo algunos a lo largo de los capítulos y exponiendo el resto en los apéndices de este documento. Y esperamos que esta documentación generada, junto con la aportada por nuestros compañeros de años anteriores, ayude y facilite lo máximo posible la ampliación de la aplicación, así como su mantenimiento.

Capítulo 3 - Maven y reorganización de recursos

Maven es una herramienta de gestión de proyectos de software desarrollada por Apache Software Foundation. Su principal función es ayudar en la construcción y administración de proyectos Java, Utiliza un archivo de configuración llamado *POM (Project Object Model)* que describe cómo se organiza el proyecto, sus dependencias, configuraciones de compilación, pruebas y distribución. A partir de este archivo *POM*, Maven automatiza tareas comunes en el ciclo de vida de desarrollo del software, como la compilación del código fuente, la gestión de dependencias, la ejecución de pruebas, la generación de documentación y la distribución del software.

En el ámbito de este proyecto cada librería usada fue añadida al *POM* actualizado a la última versión compatible existente. Esto fue una tarea que llevó más o menos hasta el mes de noviembre. Esto se debió principalmente a que al ser un proyecto iterativo que comenzó en el año 2008, el *core* de las librerías de renderizado usa versiones *Alpha* que cuentan con muchas vulnerabilidades. Se intentó actualizarlas pero no fue posible ya que las nuevas versiones cambian la sintaxis de los métodos por lo que obligaba a hacer una reingeniería completa del renderizado del esquema conceptual. Lo que sí se hizo fue actualizar el resto de las librerías y dejar comentado en el *POM* las librerías que tienen vulnerabilidades para que si es necesario en el futuro realizar la reingeniería previamente comentada.

A largo de todo el desarrollo del proyecto hemos mantenido actualizadas todas las librerías que eran posibles, para ello en GitHub activamos una funcionalidad llamada "*Dependabot alerts*" que una vez por semana escanea todas las dependencias del repositorio y manda por correo electrónico un informe con las vulnerabilidades encontradas, su código de referencia, una puntuación del uno al diez sobre la gravedad y si es posible, la versión a la que actualizarla. Por otro lado Maven permite la

instalación y creación de ejecutables de forma automática simplemente ejecutando el proyecto con el comando "*install*". En este ejecutable se encuentras embebidas todas las dependencias del proyecto.

La migración a Maven se hizo con la idea de que cuando el proyecto siga evolucionando sea mucho más fácil buscar y añadir nuevas librerías, ya que de la forma en la que estaba era necesario descargar e importar manualmente las 23 librerías con las que cuenta DBCASE 4.0. A la par que se migraba a Maven también se realizó una organización de las clases del proyecto, reorganizando los recursos en forma de imágenes, traducciones, etc. Previamente se encontraban "desperdigados" y no había unidad en cuanto a los nombres de estos. En esta etapa también se cambió la forma de importación de las imágenes creando una clase llamada ImagesPath con todos los Strings que contenían las ubicaciones para poder importarlas más fácilmente en vez de tener que introducir el String con la ruta manualmente que vez que fuese necesario. Esto mismo se hizo para los paquetes de idiomas, los paquetes de temas y otros recursos genéricos para los cuales se crearon las clases LanguagesPath, ThemesPath y Otros, respectivamente. Una vez completada esta primera refactorización nos pusimos con lo que sería la parte más importante de este proyecto: la refactorización completa del código.

Capítulo 4 - Refactorización del código

Con el objetivo de facilitar lo máximo posible que la aplicación DBCASE pueda seguir creciendo durante muchos años, decidimos refactorizar varias partes del código de la aplicación. Buscamos con ello simplificar y promover la ampliación de DBCASE, así como su mantenimiento.

Nos gustaría destacar que tuvimos siempre como una de nuestras prioridades que el código desarrollado durante tantos años por nuestros compañeros fuese perfectamente compatible con estos cambios. Nos esforzamos mucho por comprender su código a fondo y por respetar e impulsar el gran resultado de su trabajo.

4.1 Herencia para las vistas

La herencia en las vistas había sido implementada por nuestros compañeros de años anteriores. Había una clase Parent_GUI que estaba destinada a ser la clase de la que todas las vistas heredaban. No obstante, analizando el código, observamos que no se estaba aprovechando todo el potencial de la herencia en la gestión de las vistas, debido a las razones que exponemos a continuación:

- El controlador no trataba con la clase padre, sino que tenía una referencia directa para cada tipo de vista utilizada. Esto llevaba a que el controlador estuviese demasiado acoplado con las vistas específicas, de forma que llamaba a métodos que no estaban especificados en Parent_GUI.
- Las vistas tenían, entre ellas, modos distintos de funcionar y de comunicarse, que en la mayoría de los casos podían ser adaptados a un patrón único entre todas ellas.
- Muchas vistas no heredaban de Parent_GUI.

Cabe hacer una mención especial a GUIPrincipal, clase que representa a la vista principal de la que nacen todo el resto de ellas, en esta clase había mucha funcionalidad desarrollada en el controlador. Esta clase era capital para la aplicación y adaptarla al nuevo modo de trabajar con vistas sería una tarea muy costoso y complejo. Por lo que decidimos dejar que el controlador siguiese pudiendo importar la clase GUIPrincipal y obtener una referencia directa a ella a través de FactoriaGUI (que antes obtenía el controlador de sí mismo porque era él el que creaba la instancia y tenía la referencia a ella). Además, no vimos problema en dejar la referencia a las vistas About, Manual y Galería, ya que son vistas muy específicas que difieren en buena medida del resto.

4.2 FactoriaGUI

Anteriormente, el controlador guardaba una referencia directa para cada vista de la aplicación. Era el controlador el encargado de instanciar cada vista, para lo cual usaba en el constructor una llamada al método iniciaFrames() y era también el encargado de guardar una referencia a cada vista. Esto provocaba un fuerte acoplamiento entre las vistas y el controlador, de forma que éste último tenía que "conocer" cada vista en específico y, como comentamos en el apartado 4.1 en muchas ocasiones se llamaban a métodos de los frames que no figuraban en la clase Parent_GUI, y en ocasiones algún frame no heredaba de esta clase. Las ampliaciones se volvían así más complicadas, así como el mantenimiento del código, que requería de mucho análisis para determinar qué se debía hacer. Pensamos que, usando el patrón Factoría, podríamos revertir hasta cierto punto esta situación y dotar a la aplicación de una forma más sólida de gestionar las vistas.

El resultado de esto fue la FactoriaGUI, una nueva clase que se encarga de la instanciación y gestión de las referencias a las vistas. Esta nueva clase ofrece un método getGUI, que devuelve un objeto de tipo ParentGUI. Este método recibe tres parámetros,

el primero de ellos es el enum TC cuya vista asociada se quiere obtener; el segundo es un parámetro de tipo Object que contiene los datos que se quieran pasar a la vista a la hora de la creación / activación; y por último un tipo boolean que, si es cierto, fuerza la destrucción del frame almacenado (si existiese) y crea uno nuevo desde cero.

Para la gestión de las referencias, la clase FactoriaGUI usa un objeto de tipo Map<TC, Parent_GUI>, que asocia un TC con, como máximo, una vista.

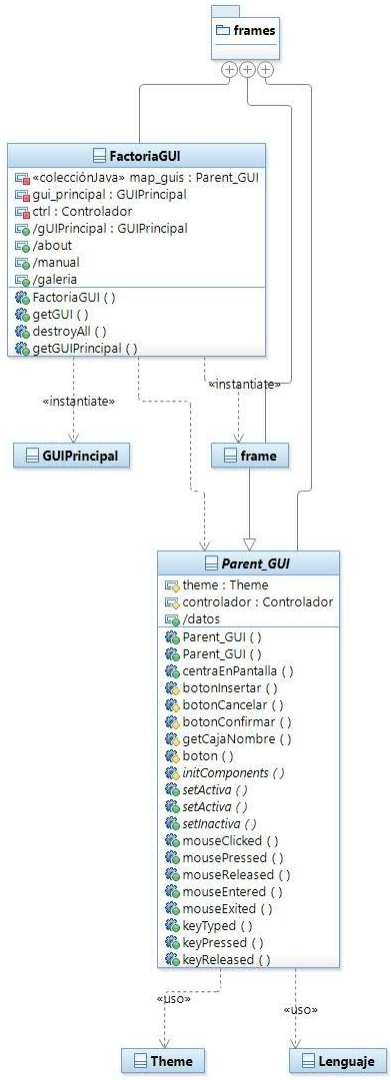

Ilustración 2: Diagrama de clase paquete vista.frames

Cabe destacar que surgió un problema complicado a la hora de determinar qué mensajes deberían ser asociados a vistas en la implementación del método getGUI. El problema surgía porque, a la hora de obtener una referencia a una vista, anteriormente se usaba el método del controlador específico para esa vista. Ahora bien, ahora esto no era así, y para ello lo más adecuado sería utilizar el mensaje asociado a la situación que se estuviese tratando, que podría ser, en general, de tres tipos según nuestros compañeros lo definieron: Controlador_, GUI_ o Servicios_. Esto quería decir que, o bien asociábamos en general a tres mensajes distintos una misma vista, o bien elegíamos algún criterio para "traducir" de alguna forma estos mensajes entre sí, de forma que en la FactoríaGUI solo hubiese ciertos mensajes.

Optamos por la segunda opción por dos razones: la primera opción hacía que la FactoríaGUI creciese de tamaño y oscurecía hasta cierto punto la función para la que esta clase había sido diseñada; y, por otra parte, traducir estos mensajes a uno unificado era un buen paso para que se pudiese dividir en un futuro el enumerado TC en varios, unos utilizados por los servicios, otros por las vistas y otros por el controlador. Esto posibilitaría además el uso de constructores para enumerados, porque al estar divididos todos los enumerados de una clase tendrían los mismos atributos (ej. un valor booleano de éxito en el caso de los servicios).

Siguiendo esta solución, decidimos que la FactoriaGUI se encargase de los mensajes Controlador_, con la única excepción de las vistas dedicadas a la gestión del workspace que no tenían mensajes de este tipo asociados.

4.3 FactoriaTCCtrl

Para llevar a cabo ese proceso de traducción que era necesario tal y como comentamos en la sección anterior, creamos una nueva clase FactoriaTCCtrl que se encargaría de esa asociación entre mensajes. Más tarde, a la hora de diseñar la función

tratarContexto que explicaremos en el apartado 4.9, esta clase resultó de gran utilidad para realizar también la traducción de los mensajes que provenían de la capa de negocio.

En la ilustración 3 podemos ver un diagrama de clase del paquete controlador, en el que se puede ver la relación del controlador con esta clase y con la factoriaGUI.

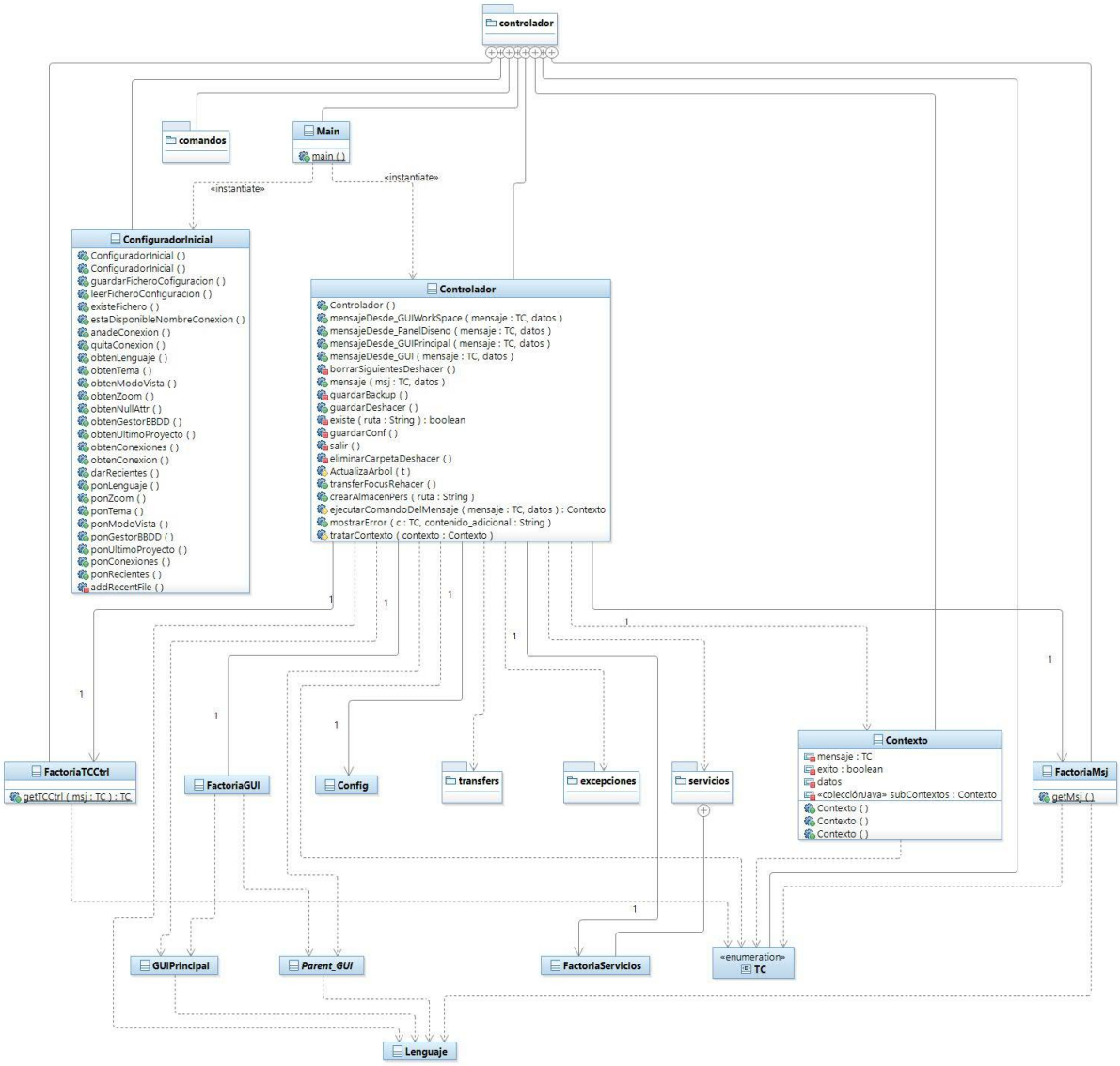

Ilustración 3: Diagrama de clase paquete controlador

4.4 Referencias a listas

Las listas de elementos, ya sean de entidades, de relaciones, etc. son utilizadas en muchos puntos de la aplicación. Anteriormente, la forma de trabajar con ellas consistía principalmente en tener una referencia local desde cada clase que vaya a utilizar en algún punto alguna de esas listas, o bien en tomar una de estas listas de otras clases, como ocurría por ejemplo con la GUIPrincipal, que usaban otras vistas para obtener estas listas de elementos.

Con esta forma de trabajar, el controlador recibía mensajes específicos de cada vista para que se le actualizasen las listas que iba a utilizar. El controlador tomaba la referencia específica para cada vista y le actualizaba las listas que le habían pedido. Además, había mensajes como 'SE_ListarEntidades_HECHO', para los que el controlador, al recibirlos, directamente actualizaba las referencias específicas de cada vista que lo necesitase.

Consideramos que esto hacía que el código fuese menos mantenible, y además era una estructura que facilitaba la aparición de errores y de referencias desactualizadas.

Como solución, dado que en la mayoría de los casos se tenía que enviar un mensaje al controlador para que actualizase la referencia específica antes de tratar con ella, decidimos unificar todos estos casos en uno solo, en el que se le envía un mensaje al controlador del tipo "ObtenerListaDeEntidades", y el controlador devuelve la lista, en este caso de entidades, actualizada.

Con esto conseguimos eliminar código repetido y hacer más fácil la inclusión de nuevas vistas, al no tener que cambiar el controlador en absoluto ni tener que crear mensajes nuevos para trabajar con las listas cuando se decida crear una nueva vista o funcionalidad. Esto, además, nos permitió que todo siguiese funcionando igual mientras

respetábamos el nuevo modelo de comunicación controlador-vistas descrito en FactoriaGUI y Herencia para las vistas.

4.5 Comandos

Hemos utilizado el patrón de diseño de la arquitectura multicapa 'Comando', con el objetivo de encapsular las funcionalidades más complejas que desarrollaba el controlador cuando recibía un mensaje para lograr un código más fácilmente comprensible, manejable y mantenible. Además, toda la estructura creada para implementar este patrón permite usar un mismo comando en varias situaciones, evitando así la repetición del código.

- Se delega en la factoría de comandos la asociación de un mensaje a un comando.

- El método ejecutarComandoDelMensaje que está implementado en el controlador. Llama a la factoría de comandos con el mensaje que el controlador ha recibido, y ejecuta el comando si este existe, pasándole los datos que el controlador ha recibido (en caso de no encontrarse un comando, el método lanza una excepción).

- Todos los comandos heredan de la clase abstracta Comando, situada en el paquete Controlador. Esta clase tiene un único atributo, de tipo Controlador, y un constructor con un único parámetro de tipo Controlador que se asigna al atributo previamente mencionado en el cuerpo del constructor. Esta clase tiene además un método abstracto, ejecutar, que recibe un parámetro de tipo Object con los datos que se quieren pasar al comando para su ejecución. Además, la clase cuenta con tres métodos más: getFactoriaServicios(),

tratarContexto(Contexto) y ActualizaArbol(Transfer). Estos métodos, todos ellos protected, llaman a los métodos de mismo nombre del controlador.

- Dentro del paquete comandos encontraremos todos los comandos que se han implementado. En cuanto a la organización de este paquete, destacamos que esta subdividido en varios paquetes: agregación, atributo, dominio, entidad y relación, y un paquete más, dedicado a las funcionalidades desarrolladas para las vistas. Es dentro del paquete comandos donde podemos también encontrar la factoría de comandos. Exponemos en esta sección, en la ilustración 4, el diagrama de clase del paquete comandos.

En la ilustración 5 presentamos un diagrama de secuencia que expone la estructura básica del método ejecutar de un comando que interactúa con los servicios (los comandos del paquete vista son los únicos que no interactúan con servicios y tienen funcionalidad más específica que es difícilmente generalizable). Exponemos también en la ilustración 4 un diagrama de clase del nuevo paquete comandos.

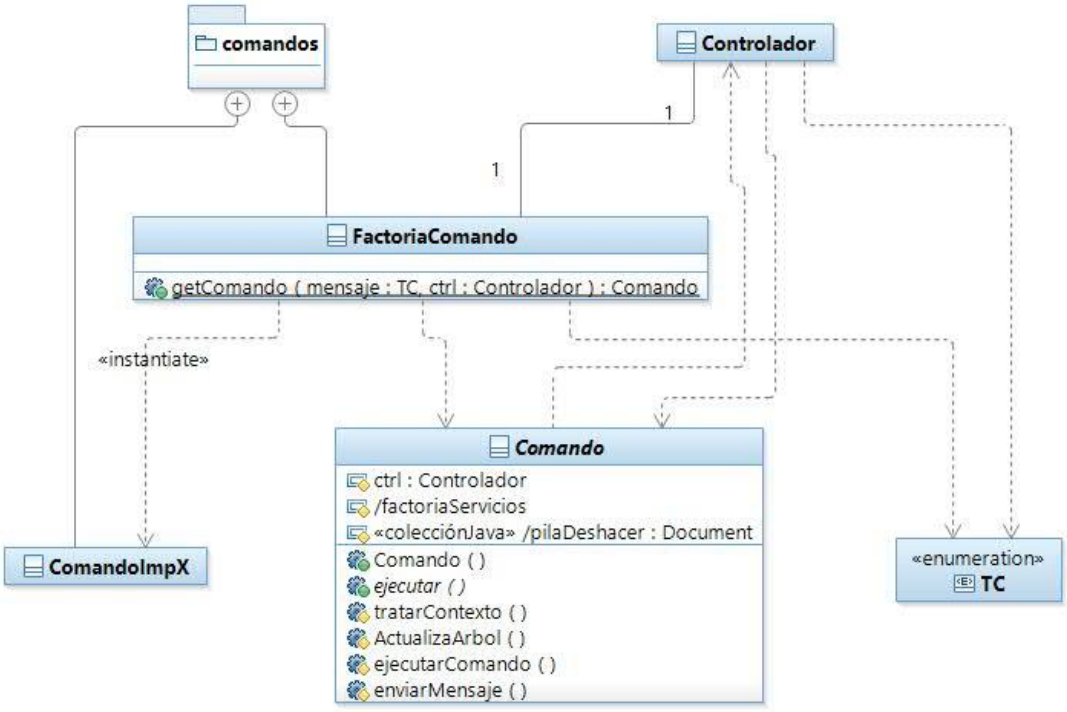

Ilustración 4: Diagrama de clase paquete comando

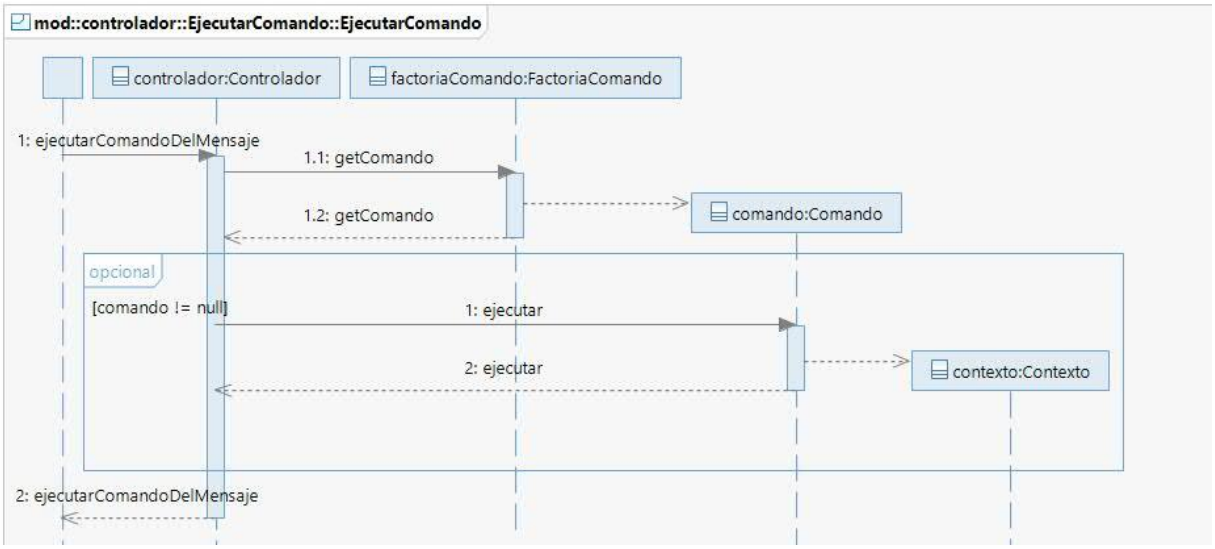

Ilustración 5: Diagrama de secuencia ejecutar comando

La implementación de comandos ha permitido reutilizar código en varias ocasiones, y ha acabado desembocando en la inclusión de varias funcionalidades simplemente por medio de eliminar algunas acciones y sustituirlas por un comando.

Esto ocurrió por ejemplo en el caso de insertar entidades débiles, en el que anteriormente se asignaban entidades a relaciones sin tener en cuenta todas las comprobaciones que se hacen en el caso específico que ya había para ello. Por medio de la creación del comando dedicado a asignar entidad a relación, se pudo reutilizar esta funcionalidad simplemente usando el comando y con una única línea de código, que además añadía las comprobaciones que se hacían en otros casos.

4.6 Factoría de servicios

Anteriormente, el controlador tenía referencia directa a todos los servicios. Con el objetivo de dividir las responsabilidades que antes tenía el controlador, hicimos una factoría de servicios, que sirve ahora como el punto desde el que tanto el controlador como los comandos se comunican con los servicios. Además, anteriormente la instanciación de todos los servicios se hacía desde el propio controlador; ahora es la factoría de servicios la que se encarga de esta tarea.

Cabe destacar que el método getFactoriaServicios() del controlador está definido con el modificador protected, lo que blinda a la aplicación contra posibles rupturas de la arquitectura, ya que ninguna vista podrá obtener una referencia a la factoría de servicios y estará entonces forzada a comunicarse con el controlador y no con la capa de negocio.

En el diagrama de clase que mostramos en la ilustración 6 se puede ver reflejado lo que explicamos. Cabe destacar que el diagrama de clase no está completo debido al gran tamaño que adquiría un solo diagrama para el paquete servicios. No obstante,

se puede ver en la ilustración 8 la relación entre factoría de servicios y la clase GeneradorEsquema. También se puede consultar la ilustración 3 para ver la relación entre el controlador y esta factoría.

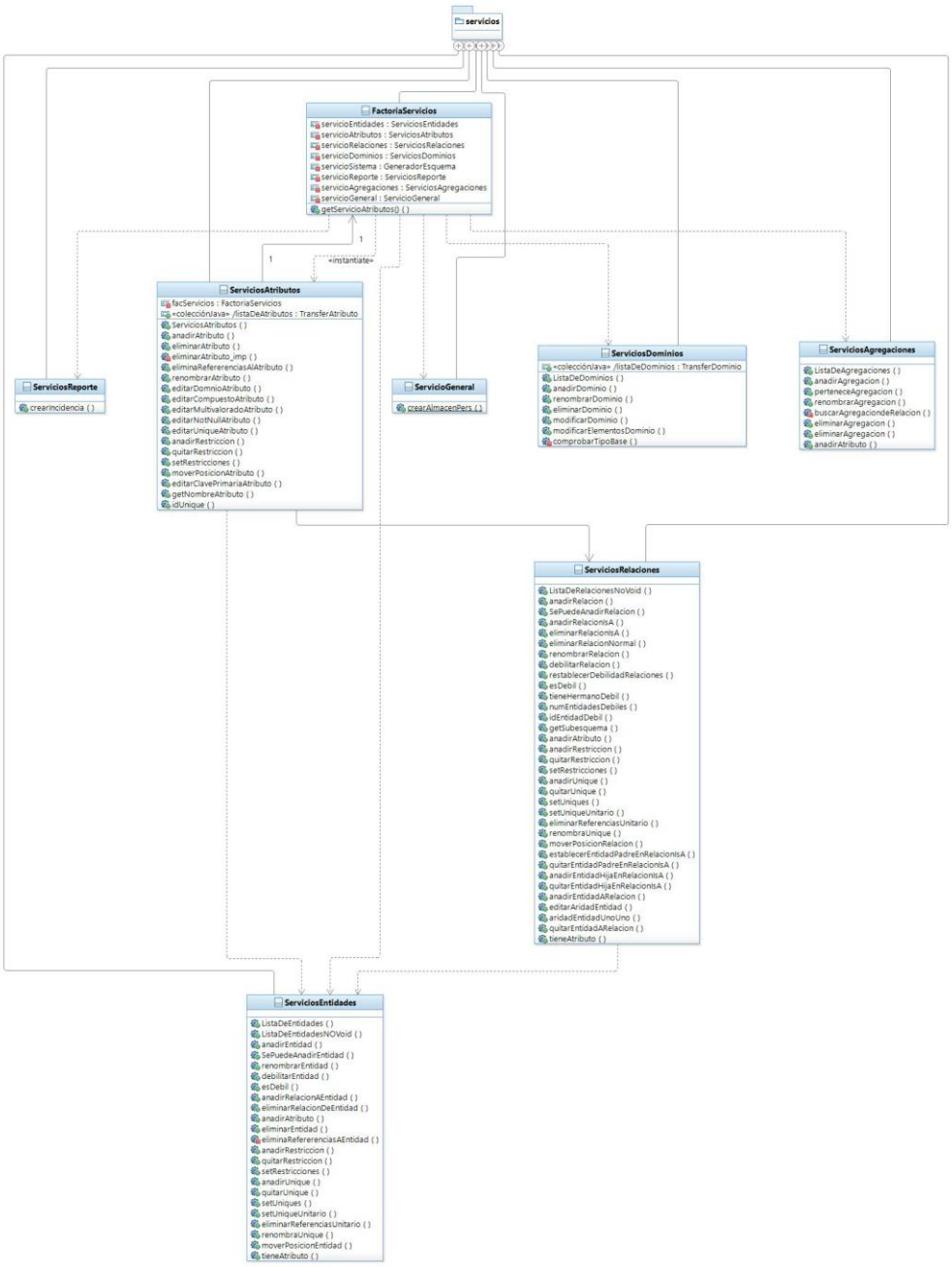

Ilustración 6: Diagrama de clase paquete servicios (parcial)

4.7 Relación controlador - servicios

Ya hemos comentado en el punto dedicado a la factoría de servicios que decidimos hacer algunos cambios en cuanto a la forma que tenía el controlador de acceder y comunicarse con los servicios.

Por otra parte, en cuanto a los servicios, observamos que todos ellos tenían una referencia a un objeto de tipo Controlador. Se usaba esta referencia para enviar mensajes a este controlador usando los métodos mensaje_DesdeSE, mensaje_DesdeSA, mensaje_DesdeSR, mensaje_DesdeSAG, mensaje_DesdeSD y mensaje desdeSS. Estos métodos del controlador tenían como objetivo recibir mensajes desde un servicio en específico (mensaje_DesdeSE estaba dedicado a mensajes provenientes del servicio de entidades etc.…), y responder de forma específica a cada mensaje.

Analizando el código y valorando esta relación entre los servicios y el controlador, concluimos que podíamos hacer algunos cambios para lograr revertir o subsanar estas situaciones que se estaban presentando conforme la aplicación crecía (y se seguirían presentando conforme la aplicación creciese):

- Mantener esta forma de comunicación servicios - controlador era difícil. La cantidad de código que requería en el controlador era muy grande.
- Las ampliaciones de código se complicaban, al tener que analizar el código de otros casos específicos para ver qué se tenía que hacer y qué no para la funcionalidad nueva que se implementaba.
- El controlador tenía un tamaño muy grande.
- Muchas de las funcionalidades que se implementaban en los métodos mensajeDesde… para cada mensaje eran muy parecidas entre sí, y en la mayoría de los casos se llamaban a los mismos métodos. En consecuencia,

había código repetido que también complicaba la mantenibilidad y la escalabilidad.

- Los servicios eran dependientes del controlador que se le asignaba y de los métodos que él ofreciese.

En consecuencia, decidimos usar los patrones de diseño y las funcionalidades que describimos a continuación en las secciones **4.8**, **4.9** y **4.10** para tratar de revertir estas situaciones

4.8 Contexto

A la hora de diseñar soluciones para estas situaciones, pensamos directamente en el patrón de diseño de la arquitectura multicapa "Contexto". Los servicios no tenían por qué usar los métodos del controlador para devolver mensajes y datos, podían devolver un objeto de tipo Contexto como retorno de sus métodos, que contuviese la información necesaria. De esta forma, y junto a la clase Config que describimos a continuación, los servicios ya no necesitaban para nada tener una referencia al controlador. Así, son completamente independientes del controlador (y, en general, de cualquier objeto que llame a sus métodos).

En cuanto a la clase contexto, ha sido diseñada e implementada con tres atributos. El primero de ellos es un tipo boolean que indica si la operación que se pretendía llevar a cabo ha sido exitosa o no; en segundo lugar, un enumerado de tipo TC (el mensaje que previamente enviaban los servicios directamente al controlador); y en tercer lugar un objeto de tipo Object donde se almacenan datos. Se han implementado métodos getters y setters para todos estos atributos.

4.9 TratarContexto

Al estudiar el código, después de analizar muchos casos de uso vimos que las acciones que llevaba a cabo el controlador cuando recibía un mensaje de los servicios podían corresponder a un patrón específico: que en el caso de operaciones fallidas se mostraba un mensaje de error y que en el caso de operaciones exitosas se hacían una serie de operaciones que, en la gran mayoría de casos, siempre eran las mismas. Con esto surgió la idea de tratar de crear una función que se dedicase a tratar estas situaciones en las que se ha llamado a la capa de negocio y se tiene una información que se quiere tratar.

Esto produjo la necesidad de tener un patrón, una norma común que tenían que cumplir los datos devueltos por la mayoría de las operaciones de los servicios (anotación) para que su tratamiento pudiese ser lo más uniforme y general posible. De cara al diseño de esta función, identificamos esos procedimientos que se hacían en casos de éxito y que exponemos a continuación:

- Llamar a guardarDeshacer() y asignación de la variable del controlador auxDeshacer a true. Aparecían comentadas asignaciones de las variables ultimoMensaje (de tipo TC) y ultimosDatos (de tipo Object) al mensaje recibido por el controlador y los datos recibidos. Además, se hacían unas comprobaciones para actualizar los componentes de la interfaz gráfica dedicados a la funcionalidad Deshacer / Rehacer. Todas estas funciones están dedicadas a la funcionalidad Deshacer / Rehacer.

- Casting del objeto de tipo Object recibido por el controlador a un objeto de un tipo específico dependiendo del caso. Por ejemplo, al recibir SE_AnadirAtributoAEntidad_HECHO se hacía un casting a Vector<Transfer> mientras que al recibir SE_RenombrarEntidad_HECHO se hacía un casting a TransferEntidad.

- Enviar un mensaje específico, y distinto del recibido, a GUIPrincipal con los datos recibidos.

- Desactivar el frame específico de la acción si lo hay.

- LLamar al método ActualizaArbol (que llama al método de mismo nombre de GUIPrincipal) con un objeto de tipo Transfer.

Haciendo uso del patrón Contexto, de esta información, y de la FactoriaTCCtrl para poder traducir mensajes con los que comunicarnos con las vistas, implementamos una función tratarContexto que está destinada a reunir todas estas acciones y cuyo diagrama de secuencia UML exponemos en la ilustración 7.

Como comentaremos en el apartado 4.21, fue necesario añadir una forma de tratar subcontextos derivados de una operación o contexto principal, explicamos esto con más detalle en dicho apartado.

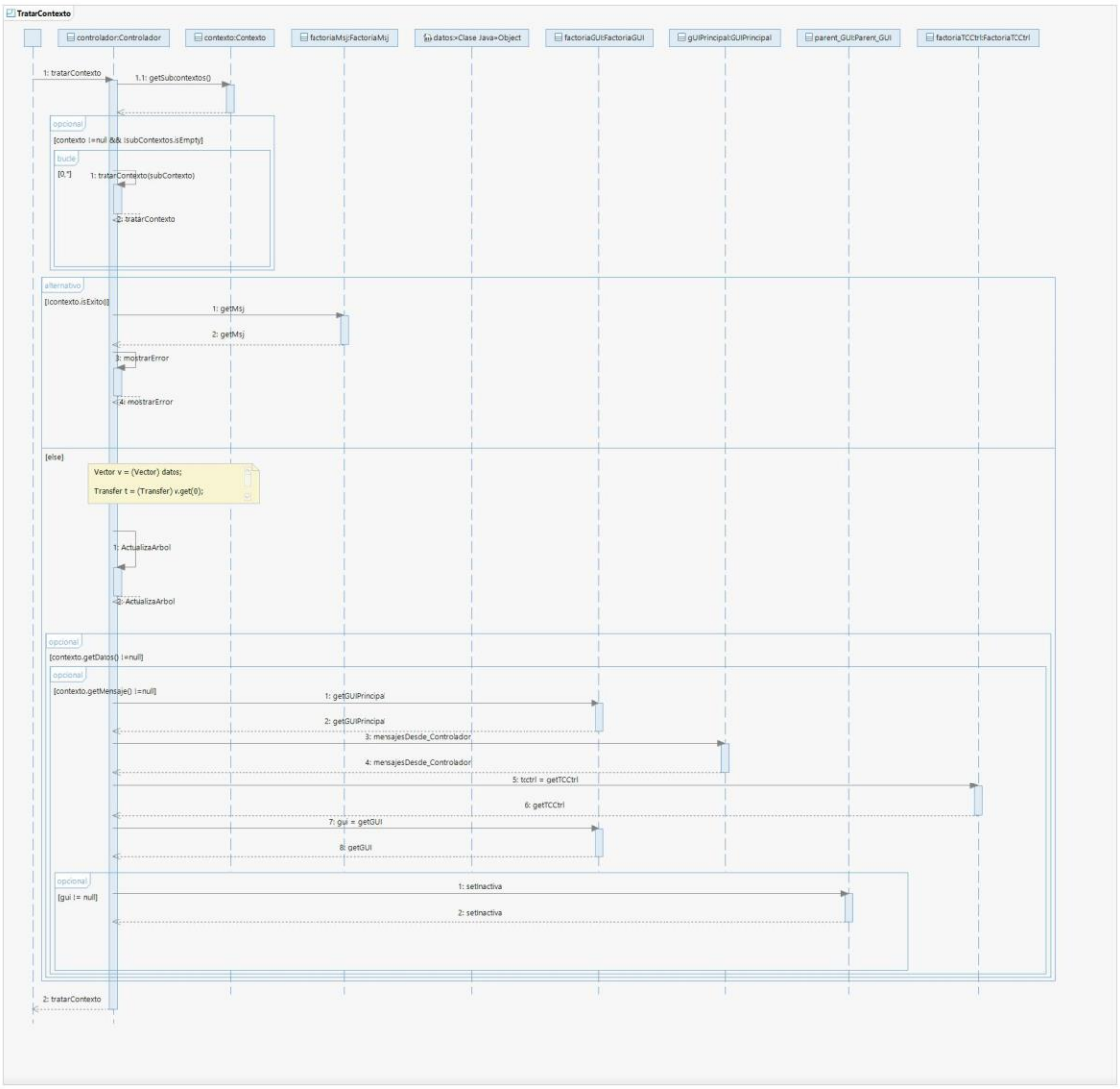

Ilustración 7: Diagrama de secuencia de tratarContexto

Nos gustaría destacar que tuvimos especial cuidado para que esta función tratarContexto y este cambio en la forma de tratar y estructurar la información devuelta por los servicios fuese compatible con la funcionalidad Deshacer / Rehacer que había sido desarrollada por nuestros compañeros. Al basarse parte de esta funcionalidad en el último mensaje recibido y los datos asociados, se consigue el mismo comportamiento al almacenar los datos de la misma forma, y al poder ser estos analizados de igual forma

por el método del controlador destinado a esta funcionalidad. Esto, claro, en el caso de que se decida seguir desarrollando dicha funcionalidad como está implementada en el método dedicado a ella en el controlador, que no es la forma en la que la aplicación actualmente lleva a cabo esta función. La forma en la que la aplicación lleva a cabo esta función actualmente también es totalmente compatible con los cambios, dado que se basa en el guardado de los documentos intermedios según se van produciendo cambios, funcionalidad en la que no se ha cambiado nada y para la que los cambios no afectan.

4.10 FactoriaMsj

Anteriormente, a la hora de mostrar un mensaje de error, se procedía de la siguiente forma: el controlador recibía un mensaje que indicaba que una acción en específico había acabado en error, y entonces, el controlador decidía qué mensaje de error mostrar, de los establecidos en la clase Lenguaje del paquete vista.

Para dividir la responsabilidad en estas acciones, decidimos hacer una factoría de mensajes, que se encargase de la asociación mensaje recibido por el controlador - cadena informativa. Esta nueva factoría se encuentra en el subpaquete 'factorias' del paquete 'controlador. Cabe destacar que esta factoría también se encarga de llamar al método text() de la clase Lenguaje, de forma que lo que devuelve no es un entero con el cual obtener la cadena haciendo Lenguaje.text() sino que directamente devuelve la cadena, encapsulando estas especificidades de implementación para que no se tenga que lidiar con ellas desde fuera. Con esta nueva forma de tratar los mensajes de error, se ha conseguido eliminar gran cantidad de código repetido.

4.11 Config

En ocasiones, desde algún punto de la aplicación era necesario obtener la variable path y la variable isNullAttrs, ambas presentes en el controlador. Decidimos mover estos atributos a una nueva clase Config, para que, si alguna clase necesitaba esa información, no tuviese que ser dependiente de todo el controlador sino solo de esta clase Config que contiene esa información relevante relativa a la configuración de la aplicación.

4.12 Transfer atributo

Anteriormente, Transferatributo tenía una referencia al controlador. La razón por la que se estaba usando esta referencia era que, a la hora de determinar si un atributo es nullable o no, se tenía en cuenta el valor de isNullAttrs, un atributo que anteriormente se encontraba en el controlador y que indica si se permiten atributos nullables. Consideramos que, debido a la propia naturaleza del patrón transfer, y a que había una forma de cambiar esta situación para establecer una división de responsabilidades más clara, lo mejor era cambiar esto. Decidimos usar la ya mencionada clase Config del paquete controlador, desplazando este atributo a dicha clase. También decidimos hacer que este atributo pasase a ser estático, siguiendo lo ya comentado en el apartado **4.11**. De esta forma, el transfer ya no necesita la referencia al controlador (que ha sido eliminada).

4.13 Arquitectura

Una de las primeras cosas que vimos a la hora de analizar el código es que la arquitectura de la aplicación era una arquitectura multicapa. La división principal de los ficheros fuentes eran los paquetes vista, controlador, servicios... Nuestros compañeros

de años anteriores desarrollaron la aplicación siguiendo esta arquitectura, y queda claro, viendo el código, que toda la aplicación está desarrollada desde esta perspectiva multicapa. No obstante, encontramos algunos sitios en los que se había roto la arquitectura. Los analizamos de forma específica y valoramos por qué se había roto la arquitectura.

En algunos casos, se daba la situación en la que desde un punto de la aplicación que no pertenecía a la capa de negocio ni a la de persistencia, se creaban objetos DAO para obtener listas de elementos, que eran usadas para alguna funcionalidad en específico. Fue útil para estos casos lo que desarrollamos en la sección 4.4, y usamos esta forma de obtener listas para quitar estas comunicaciones con la capa de persistencia. Esto ocurría en los siguientes casos:

- Mensaje modificar entidad recibido por el controlador (código ahora presente en el comando ComandoModificarEntidad)
- Action listener del submenú añadir atributo en la clase MyMenu (paquete vista.componentes)
- Método getListaTransfers en la clase AddTransfersPanel (paquete vista.componentes.GUIPanels)
- Mensaje editar elemento recibido por el controlador (código ahora presente en el comando ComandoEditarElemento)
- Método modificar atributo recibido por el controlador (código ahora presente en el comando ComandoModificarAtributo)

Por otra parte, en los objetos DAO, a la hora de acceder al almacén persistente en la función *dameDoc*, se podían producir varias excepciones, que se capturaban en un catch y se manejaban usando la clase JOptionPane para mostrar un diálogo informando del error. Para cambiar esto, dado que la capa de persistencia no debería encargarse de mostrar el error en la vista, decidimos hacer uso de las excepciones de

Java. Creamos una nueva clase ExceptionAp, que hereda de Exception. El DAO ahora lanza esta excepción con un enumerado asociado a ella (en el propio constructor de ExceptionAp), y esta excepción es capturada por el controlador (después de ser relanzada por los servicios al ser anotados con throws ExceptionAp y no manejar la excepción), donde se traduce el enumerado al lenguaje actual de la aplicación haciendo uso de FactoriaMsj, y por último desde el controlador se delega en la GUIPrincipal el acto de mostrar ese mensaje de error.

Para ello, todos los métodos del controlador y de los comandos que se comunican con los servicios ahora están equipados con un try-catch, capaz de capturar la excepción y llamar a un nuevo método, *mostrarError*, que llama al método que se encarga de ello en GUIPrincipal. Este cambio permitió además implementar fácilmente una forma de notificar errores no solo en dameDoc sino en guardaDoc, donde antes no se notificaba directamente por medio de la interfaz de usuario, sino que se llamaba al método printStackTrace de la excepción. Ahora, al captar una excepción, lanza una ExceptionAp de la misma forma que en el caso anterior, que se mostrará también, en la interfaz gráfica, de la misma forma.

Por último, en la clase GeneradorEsquema perteneciente a la capa de modelo, se hacía uso también de la clase JOptionPane para notificar diferentes situaciones, que en ocasiones se correspondían con errores y en otras eran mensajes informativos. Usamos la ya mencionada clase ExceptionAp para cambiar esta situación en cuanto a los errores.

No obstante, para que siguiesen apareciendo los mensajes informativos, dado que siempre se producían al final de la operación, movimos la funcionalidad para que se mandase la orden desde el controlador, a la vuelta de la llamada a la capa de negocio, si se ha confirmado que la operación ha sido exitosa. Se creó una función en

el controlador a la que se llama para mostrar un mensaje informativo y que delega en la GUIPrincipal los detalles acerca de cómo mostrarlo.

Además, en dicha clase GeneradorEsquema se había desarrollado una funcionalidad dedicada a la elección del fichero destino del esquema, si es que se elige uno. Se hacía uso de la clase MyFileChooser perteneciente al paquete vista, que interviene en dicho proceso de la elección del fichero. Esta clase GeneradorEsquema también hacía uso de la clase JOptionPane para mostrar algunos mensajes. Dado que pertenece a la capa de negocio, modificamos esta situación. Ahora el controlador delega en GUIPrincipal la forma de escoger el fichero, y obtiene la información necesaria (en este caso la ruta) para llevar a cabo la operación. Después, se comunica con la capa de negocio, y delega en la vista la forma de mostrar el mensaje acerca de la operación.

4.14 Separación del Main

El código dedicado a lanzar la aplicación y a inicializarla (la función main) estaba dentro del controlador. Dado que es bastante compleja y contiene varios procedimientos de los que, pensamos, convenía separar al controlador, y con objetivo de restar alguna funcionalidad más al controlador y de lograr una división de responsabilidades más amplia, trasladamos la función a una nueva clase Main que es la que ahora se encarga de ello.

4.15 Separación de algunas funciones del controlador a UtilsFunc

Había algunas funciones presentes en el controlador que eran usadas por varias partes de la aplicación y que no eran estricta responsabilidad del controlador, como algunos algoritmos para ordenar listas, particiones de vectores y otras utilidades.

Movimos estas funciones a una clase UtilsFunc en el paquete misc. Todas estas funciones eran estáticas y lo siguen siendo.

4.16 Relación entre validadorBD y GeneradorEsquema

En la aplicación hay dos clases que intervienen decisivamente en el proceso de generación del esquema lógico a partir del diagrama: validadorBD y GeneradorEsquema. La clase validadorBD interviene en el proceso de validación del esquema generado por la clase GeneradorEsquema. Es por ello por lo que las dos clases tienen una relación bastante importante.

La clase validadorBD heredaba de la clase GeneradorEsquema, lo que consideramos no era lo más adecuado debido a la relación que tenían porque, aunque el validador usase métodos presentes en el generador, estos métodos no eran parte de la funcionalidad del validador, y en definitiva el validador no era una forma de generador sino que era más bien un ayudante, y así es usado por el resto de la aplicación. Esta relación de herencia no influía más allá de la propia clase validador, porque el validador nunca era tratado como generador. Viéndolo así, cambiamos esta relación por una relación de dependencia: ahora el validador obtiene una referencia a un generador en su constructor, y de esta forma puede usar los métodos de él que necesita. Exponemos en la ilustración 8 un diagrama de clase de la capa de negocio dedicado a la clase GeneradorEsquema y a las clases que están en relación con ella.

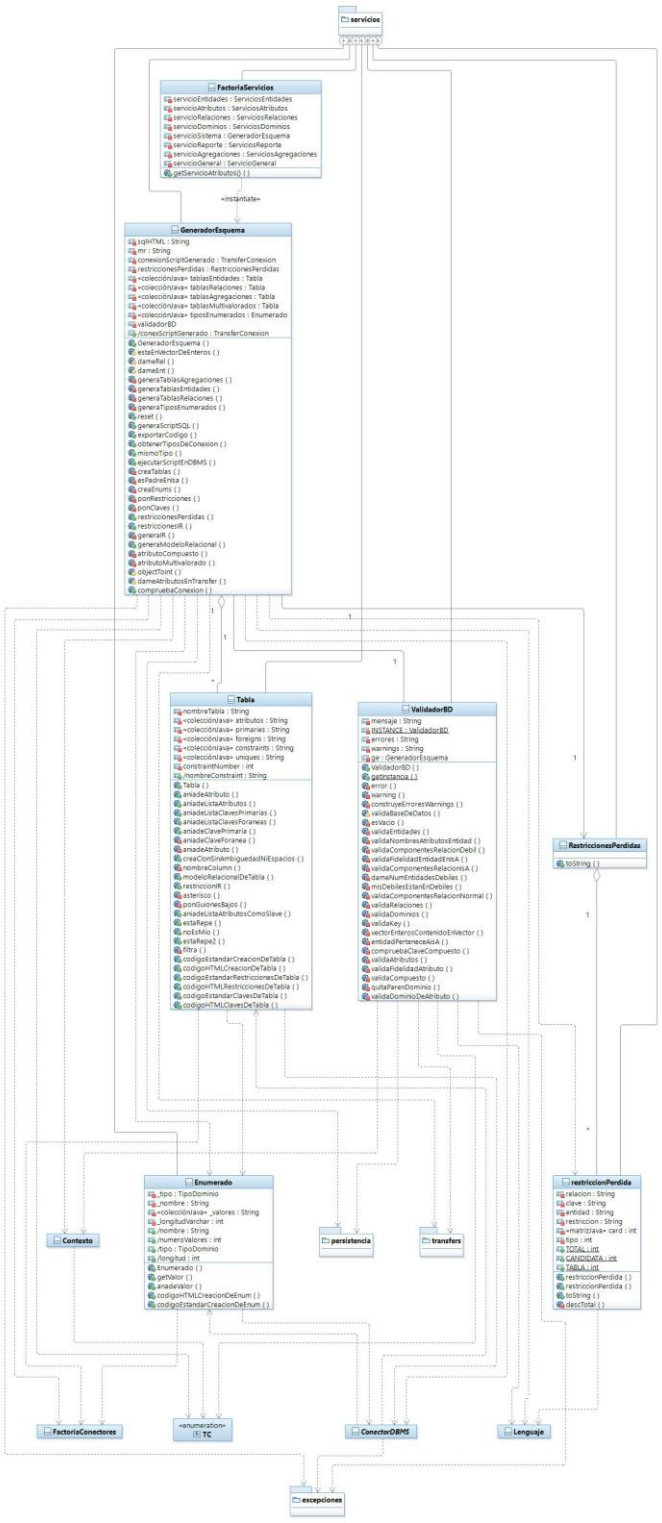

Ilustración 8: Diagrama de clase GeneradorEsquema

4.17 Creación de almacén persistente

De cara a crear ficheros, que es la forma de persistencia que usa actualmente DBCASE, anteriormente se usaba una función *creaFicheroXML()* que estaba presente en el controlador. Dado que se trata de una funcionalidad que debería concernir a la capa de persistencia, trasladamos la función a dicha capa.

Por otra parte, creamos un nuevo servicio que está a disposición del controlador para que se dé la orden de crear el almacén persistente, y es este servicio el que se comunica con la capa de persistencia.

4.18 Constructores y funciones de cargar y guardar documentos en DAOs

Anteriormente estaban replicadas en todos los DAO dos funciones: dameDoc y guardaDoc, dedicadas a la carga y almacenamiento de documentos. Como la funcionalidad era la misma, y como también los constructores se dedicaban a las mismas funciones, hicimos una clase abstracta DAO, en la que quedan implementadas ambas funciones y el constructor, y de la que ahora heredan el resto de los objetos DAO. De esta forma, el código repetido pudo ser eliminado y las funciones solo están presentes una vez en esta clase abstracta. Es en esta clase abstracta donde incluimos también la función dedicada a crear el almacén persistente, mencionada en el apartado **4.17**.

Se pueden ver en el diagrama de clase de la ilustración 9 las nuevas relaciones entre las clases del paquete de persistencia.

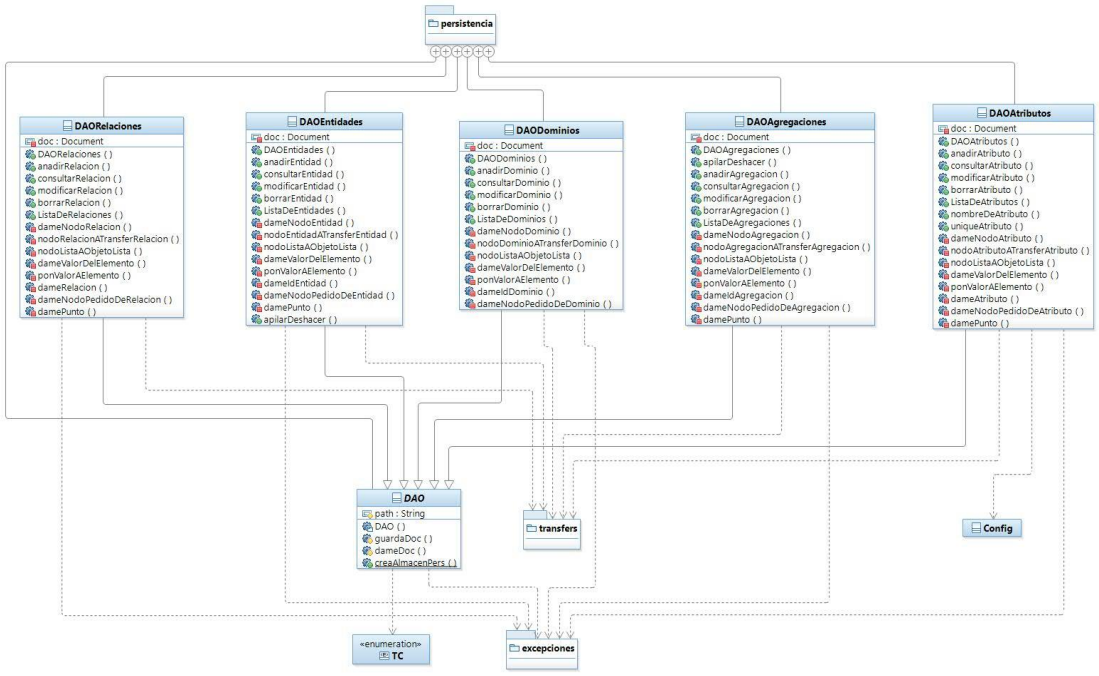

Ilustración 9: Diagrama de clase paquete persistencia

4.19 Método mensaje en controlador

La situación comentada en apartado 4.4 nos llevó a implementar un método en el controlador por el que éste recibiera mensajes, pero sin que estuviera dedicado a tratar mensajes provenientes de un sitio específico, como los métodos mensajeDesdeGUIPrincipal, los anteriores métodos mensajeDesdeServicioEntidades, etc. Creamos un método mensaje con este propósito, para poder implementar en él la recepción y tratamiento de mensajes que no estuviesen acotados a un sitio específico de la aplicación, sino que podían ser recibidos desde varios sitios.

4.20 Comandos para entidades débiles

A la hora de insertar una entidad débil, anteriormente se procedía de la siguiente forma: al recibir el evento del clic en el botón de eliminar, se enviaba un mensaje al controlador, quien al recibirlo comprobaba si se podía insertar la entidad con el método correspondiente del servicio de entidades y a continuación llamaba al método comprobadaEntidad(boolean factibleEntidad) del frame de insertar entidad. Si factibleEntidad era cierto, el frame enviaba un nuevo mensaje al controlador para ver si se podía insertar la relación que identificaba a la entidad débil, y el controlador hacía el mismo proceso, llamando esta vez a comprobadaRelacion(boolean factibleRelacion), que se encargaba de realizar las operaciones de inserción si factibleRelacion era cierto.

La nueva herencia para las vistas y la eliminación de todas las referencias del controlador a vistas específicas imposibilitó este procedimiento. Además, consideramos que era mejor que la vista simplemente se encargase de mandar un solo mensaje al controlador y que fuese el controlador (en este caso un comando) quien controlase las condiciones que antes se controlaban desde la vista. Nacieron así los comandos insertar entidad débil e insertar relación débil, que se encargan de las funciones descritas.

Por otro lado, a la hora de modificar una entidad se puede debilitar una entidad ya creada. El proceso para editarla era el mismo que el usado para insertarla con la única diferencia que la propia entidad no se insertaba, solo la relación que la identificaba. Cambiamos esto haciendo uso de los mencionados comandos, gracias a los cuales la vista de modificar entidad solo manda el mensaje de modificar entidad al controlador, y este (en este caso los comandos de modificar elemento y de insertar relación débil) se encarga del resto. Eliminamos así además la repetición del código para esta funcionalidad, que estaba duplicado en las vistas de insertar entidad y modificar entidad.

4.21 Recursión en eliminación de subatributos

Con los cambios descritos surgió un problema a la hora de eliminar subatributos. En la aplicación se eliminan recursivamente todos los subatributos del atributo que se quiere eliminar, y por tanto en cada llamada recursiva antes había un mensaje enviado al controlador, para el atributo que se estaba eliminando.

Ahora los servicios no tienen referencia al controlador, luego había que adaptar esta situación. Se dividió entonces el procedimiento de la capa de negocio para eliminar un atributo en dos funciones, **eliminarAtributo,** la función que se estaba usando todo este tiempo y se sigue usando y **eliminarAtributo_imp**, una función privada a la que se ha movido la funcionalidad que habían implementado nuestros compañeros de años anteriores y que estaba en la función mencionada previamente, y sobre la que hemos hecho algunas adaptaciones.

La función **eliminarAtributo** hace la primera llamada recursiva llamando a la función **eliminarAtributo_imp**, que genera la recursión y que ahora devuelve una doble cola de contextos, resultado de todas las eliminaciones de atributos. La recursión es de la forma:

eliminarAtributo_imp(atributo):

si compuesto = doble cola que contiene el contexto de eliminar atributo y todos los contextos devueltos por llamar a eliminarAtributo_imp() con cada subatributo suyo.

si simple = doble cola que contiene el contexto de eliminar atributo.

Esta operación sugirió una modificación en la clase Contexto, que ahora necesitaba una forma de almacenar sub-contextos derivados de la operación principal. Se añadió así a la clase contexto, tras valorar diferentes soluciones, un vector de sub-contextos, destinado a almacenarlos. Esto permite almacenar correctamente toda la información del resultado de una operación de los servicios, incluso si supone el uso de

otros métodos de otros servicios o del mismo, y siempre devolviendo un único contexto con el resultado de la acción que se ha pedido hacer.

Para poder tratar este tipo de contextos se añadió una nueva funcionalidad a tratarContexto, que comprueba si el vector de sub-contextos del contexto existe y no está vacío, y en ese caso llama a tratarContexto con cada uno de ellos, abstrayendo además completamente al controlador de si la operación ha generado sub-contextos o no. Usando esto, la función eliminarAtributo toma el contexto principal, es decir el correspondiente a eliminar el atributo sobre el que se hizo la llamada en un principio, y mete en su vector de sub-contextos el resto de los contextos que estaban en la doble cola, desde el primero hasta el último. Como el contexto que hemos llamado principal es el último en añadirse, y como necesitábamos acceder tanto al principio de la cola (a la hora de extraer primero los contextos de los atributos que primero se han eliminado) como al final (para tomar el contexto principal, correspondiente al último atributo que se elimina), decidimos usar una doble cola. Por último, devuelve el contexto.

Por último, destacamos que los contextos se tratan en el orden en que se produjeron, es decir, primero se tratan los contextos de los sub-atributos en orden, porque se eliminan antes que el atributo padre, y luego el contexto del padre, para que las vistas puedan actualizar correctamente su contenido, logrando así el mismo comportamiento que tenía antes la aplicación a la hora de hacer esta operación, en la que se trataban los mensajes recibidos por el controlador en este mismo orden debido a la recursión.

4.22 EntidadYAridad y NodoEntidad

EntidadYAridad y NodoEntidad son dos clases importantes en la aplicación. Están destinadas a reunir la información acerca de una asociación relación – entidad, en el caso de EntidadYAridad, y acerca de un objeto EntidadYAridad y su entidad asociada,

en el caso de NodoEntidad. Analizando estas clases vimos que, aunque estaban en el paquete persistencia, no intervenían en la interacción con el almacén persistente, y no eran usadas por las clases del paquete de persistencia más que lo eran por los servicios o por otros componentes de la aplicación. Concluimos que, dada su función, eran más bien clases que pertenecían al paquete de transfers (paquete dentro de la estructura de la aplicación en el que se encuentran todos los objetos "transfer", que encapsulan la información sobre los objetos de la aplicación como agregaciones, atributos, etc.), porque eran usados por varias capas y estaban encargados de reunir información, y en ocasiones de alguna función dedicada a representar esa información, como el resto de transfers que utiliza la aplicación.

Capítulo 5 - Nuevas funcionalidades

Este capítulo está dedicado todas las funcionalidades añadidas a la aplicación una vez terminada la refactorización, esto incluye también aquellas que se empezaron pero no se terminaron cuyos avances están comentados en el código.

5.1 Flechas al cargar proyectos

Nuestros compañeros del año pasado comentaron en la memoria de su proyecto, en la sección relativa al trabajo futuro, que habían detectado un error: al realizar cualquier acción que requería cargar un nuevo proyecto con DBCASE, se perdían todas las restricciones de cardinalidad 1. Tras bastante análisis, vimos cómo lo que ocurría no era que se perdiesen las restricciones de cardinalidad 1 sino que, aunque estas siguiesen presentes, no se representaban correctamente en el diagrama. Lo que ocurría era que, a la hora de determinar si una unión relación – entidad, es decir, una línea, tenía una flecha, se tenía en cuenta una variable marcadaConCardinalidad, perteneciente a la clase EntidadYAridad que representaba esa unión entidad – relación. Esta variable marcadaConCardinalidad, de tipo boolean, se marca a true cuando se asocia una entidad a una relación, pero únicamente se usa a la hora de editar esa asociación para determinar si en el diálogo de edición debería aparecer el botón de cardinalidad marcado o no (si no se marca el botón se asocia cardinalidad principio rango: 0, final de rango: N a la entidad).

Lo que ocurría era que, dado que esta variable únicamente es utilizada para una vista, su valor no era almacenado en persistencia, y era inicializado a false. Así, cuando se determinaba si se debía pintar una flecha, se veía que ese valor era falso y se decidía en consecuencia que no se debía pintar una flecha. Para solucionarlo, decidimos quitar esta comprobación dado que la que hacía falta era la que se hacía después de la primera: determinar si el final del rango de la cardinalidad era igual a 1.

5.2 Renderizado de relaciones recursivas

A la hora de renderizar las relaciones en las que aparece una misma entidad dos o más veces, ocurría un problema que ya había sido advertido en buena medida por nuestros compañeros del año anterior, que habían indicado que ocurrían algunos comportamientos raros. Además, se puede ver en el punto 7 de la lista que se presenta en la sección **1.2**, cómo se hace una observación acerca de la forma en que se renderizan los roles en relaciones recursivas.

Vimos cómo, si la entidad estaba asociada en una aparición con una cardinalidad distinta a otra, únicamente aparecía una de ellas, como si ambas asociaciones fueran de la misma cardinalidad. Comprobamos que esto ocurría debido a cómo se recolectaban las clases EntidadyAridad asociadas a una relación a la hora de renderizarla. Lo que ocurría era que se usaba una función getEntidadYAridad(TransferEntidad) que, siempre, devolvía la primera aparición de un objeto EntidadyAridad cuya entidad era la entidad pasada como parámetro (ambos TransferEntidad tenían el mismo id de entidad almacenado), de forma que siempre se devolvía el mismo objeto EntidadYAridad, aunque hubiese varios asociadas a dicha entidad. El rol se renderiza desde otra función, luego no estaba afectado por este efecto. Se puede ver esta situación en el ejemplo número 4, en la ilustración 10.

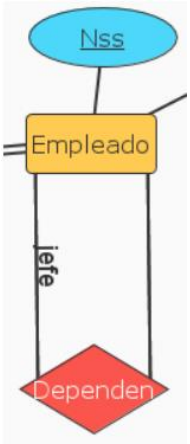

Ilustración 10: Renderizado de relación recursiva en DBCASE 3.0

Para solucionarlo, implementamos una nueva función en la clase EntidadYAridad que permite obtener la aparición número x de un objeto EntidadYAridad en el que la entidad es la indicada (y que, si no hay tal aparición, devuelve null). Ahora, desde el proceso de renderizado, en la clase CreaLineas, se llama a esta nueva función, usando como número de aparición una variable que ya estaba siendo utilizada por otras partes de esa misma función y que era justo lo que se necesitaba. De esta forma, dicho problema en el renderizado se solventó, como se puede ver en la ilustración 11, solucionando además el punto 7 de la lista que comentábamos al inicio de esta sección. Aunque, no obstante, identificamos un problema en el renderizado de los roles.

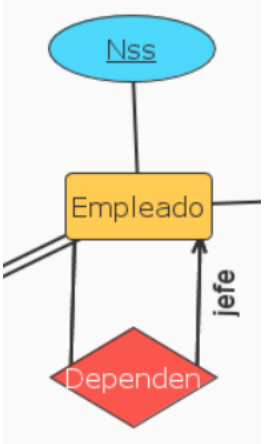

Ilustración 11: Renderizado de relación recursiva en DBCASE 4.0

En ocasiones, concretamente cuando la entidad en la relación recursiva está por debajo de la relación, el rol de las asociaciones se desplaza de una forma que hace parecer que es el rol de otra aparición (aunque sigue estando correctamente identificado en la aplicación). En otras ocasiones, aunque son más infrecuentes, ocurre al revés. Creemos que esto se puede deber a que el orden en el que se asocian las apariciones interviene en el reparto del espacio. Hicimos algunos intentos de solucionar este problema modificando las funciones de la clase EtiquetaSobreLinea.java (paquete vista.diagrama.lineas), que consistían principalmente en invertir el valor de una variable

parallelOffset en ciertas ocasiones, pero observamos finalmente que no se llegaba a un comportamiento uniforme.

5.3 Renombramiento de atributos unique

A la hora de renombrar un atributo que había sido marcado como unique, saltaba una excepción, que se debía a un problema a la hora de editar la tabla de uniques de la relación / entidad a la que pertenece el atributo renombrado. Este problema provocaba que no se actualizase correctamente dicha tabla de uniques. La excepción indicaba que se había tratado de hacer un casting que no era válido.

Desde la aplicación se trataba de hacer un casting a TransferAtributo de cada elemento de la lista de uniques que tenía la entidad / relación a la que pertenecía el atributo. Comprobamos que, en realidad, dicha lista no está compuesta por objetos de los que se pueda hacer un casting a TransferAtributo, sino que estaba compuesta por cadenas, objetos de tipo String, que indican el nombre del atributo presente en la tabla unique.

Se solucionó este problema usando el método toString y eliminando dicho casting, con lo cual la tabla de uniques ya se actualiza correctamente y no salta la excepción. Este problema provocaba que, en la construcción de algunos ejemplos, la lista de uniques en el archivo .xml del proyecto apareciese desactualizada. Debido a ello y a otra situación que comentamos en el **Capítulo 7**, introdujimos algunas modificaciones en los archivos de ejemplo que estaban en DBCASE.

5.4 Guardar como XML modelo lógico y físico

Se agregó una funcionalidad para guardar los modelos lógico y físico en formato XML. Anteriormente, el menú de guardar permitía guardar los modelos en formato de

texto plano (.txt), y el modelo físico también se podía guardar en formato SQL (.sql). Sin embargo, para mejorar la flexibilidad y la interoperabilidad, se implementó la capacidad de guardar ambos modelos como archivos XML. Para lograr esto, se agregó una nueva opción al menú de guardar y se creó un nuevo método para formatear la información de los modelos y escribirlos en archivos XML. Esto implicó trabajar en la estructura y el formato de los datos para garantizar una representación coherente y legible en el archivo XML resultante, lo que podría incluir la serialización de objetos o la conversión de datos estructurados en un formato XML válido. La segunda parte de la tarea consistía en poder cargar estos modelos en formato XML, pero no se completó.

5.5 Correcciones en traducción a modelo lógico y físico

Nuestro tutor nos llamó la atención sobre el hecho de que, en el ejemplo número cinco, algunas relaciones no se estaban traduciendo al modelo lógico con todos los atributos que debería, como se puede ver en la Ilustración 12.

Ilustración 12: Traducción errónea en Ejemplo 5

Tras analizar por qué estaba ocurriendo, vimos que se debía a lo siguiente. En la lista de las relaciones que hay en el proyecto que se tenga abierto, están incluidas las relaciones IsA, las que identifican a una entidad débil, y las de tipo normal. Los dos primeros tipos de relaciones tienen una influencia sobre los atributos que se le asocian a una entidad en la traducción. A la hora de traducir las relaciones, se recorría la lista de relaciones y se traducían dichas relaciones en el orden que estuviesen. Esto podía provocar que una entidad apareciese en una relación con menos atributos de los que al final acaba teniendo, cuando ya sí que se ha tratado una relación IsA / de identificación a la que pertenece. Resolvimos este problema haciendo que se tratasen primero las relaciones IsA / de identificación, y por último las relaciones de tipo normal, ya con las entidades en un estado estable.

No obstante, había una relación que se seguía traduciendo mal. Tras depurar el código vimos que, a la hora de realizar la traducción al modelo lógico, estaba interviniendo de una forma no esperada una función llamada filtrar, presente en la clase Tabla. Esta función se utiliza desde la traducción con el objetivo de separar los atributos que son clave primaria y los que no lo son. Para ello, se le dan dos vectores, uno con todos los atributos y otro con los que son clave primaria, y la función devuelve un vector con los elementos que están en el primero y no en el segundo. El problema que ocurría era que se podía dar el caso de que hubiese dos atributos de mismo nombre y dominio, y la función no tenía en cuenta ese caso, de forma que eliminaba del vector resultado todas las apariciones, en vez del número de apariciones correspondiente. Para solucionarlo, usamos un mapa, que asocia una cadena nombre-dominio-poseedor con un número de apariciones. De esta forma, consultando el mapa y restando uno en la entrada correspondiente cada vez que se filtra un atributo, solo se filtrará el número de veces que aparezca en el vector de claves primarias, dejando el resto de las apariciones en el vector original.

Podemos ver en la ilustración 13 cómo tras estos cambios, las relaciones que tenían errores en la traducción (relaciones en las que intervenían entidades hijas o entidades débiles) se traducen de forma correcta.

Debido a una situación que comentamos en el **Capítulo 8**, si hay atributos de mismo nombre y dominio en una tupla, en ocasiones la impresión de claves foráneas y la traducción al modelo físico genera resultados erróneos porque no se desambiguan correctamente dichos atributos, como ocurre en el ejemplo número cinco. Como comentamos en dicho capítulo, proponemos introducir un cambio en el proceso de desambiguación para solventar estos problemas.

Ilustración 13: Traducción de relaciones actual de Ejemplo 5

5.6 Ventana de editar cardinalidad y participación

Introdujimos algunos cambios en el frame dedicado a editar la cardinalidad y participación de una entidad en una relación (paquete vista.frames, clase

GUI_EditarCardinalidadEntidad.java) con el objetivo de que se actualizasen de forma correcta los botones según la entidad que se seleccionaba. La funcionalidad estaba ya presente en el código, pero en algunos casos no estaba funcionando. Tras depurar el código vimos que estaba ocurriendo algo similar a lo que comentamos en el apartado **5.1**, en ocasiones intervenían variables desactualizadas porque no se cargaban desde los archivos de persistencia. Cambiamos esto usando una función que ya estaba presente en la clase, llamada generaInicioFin, que se estaba usando en la mayoría de los casos, pero no en todos. Esta función se encarga de generar toda la salida que se necesita para determinar qué botones seleccionar. Después, tomamos el código que se usaba para actualizar los botones según dicha salida y lo pusimos en una función actualizarCampos, a la cual se llama ahora desde los tres sitios en los que se cambia el estado de los botones (inicialización, cambio de entidad y cambio de rol).

Por otra parte, también ocurría en ocasiones que la entidad que se tomaba no era la correcta. Vimos que se utilizaba el índice del nombre de la entidad en la lista como id de la entidad, lo cual no era correcto en todos los casos. Para cambiarlo, implementamos una pequeña función dedicada a buscar una entidad por su nombre.

De esta forma, en los casos de prueba que hemos llevado a cabo los contenidos de la ventana se actualizan de forma correcta.

De cara al panel de participación mínima-máxima, en el que se permite introducir números específicos, actualmente no se actualiza como sí lo hacen el resto de los componentes de la ventana. Dejamos comentado en el código fuente, en la ya introducida función actualizarCampos, una propuesta que se podría seguir si se desea implementar esta funcionalidad.

5.7 Otras funcionalidades

Se intentó seguir el trabajo de los compañeros de DBCASE 3.0 ya que estos habían comenzado los objetivos 22, 29 y 35, todos relacionados con opciones de zoom. Se

avanzó trabajo, pero no se pudo completar por problemas en el renderizado y quedó comentado dentro del código.

Capítulo 6 - Pruebas

Las pruebas fueron una pieza fundamental a lo largo de todo el proyecto. Su importancia es bien conocida para generar software de calidad. Las realizamos continuamente a lo largo de todo el proyecto, y nos ayudaron en gran medida a la hora de detectar errores y poder solucionarlos.

Era importante llevar a cabo pruebas exhaustivas que nos mostrasen que la aplicación seguía funcionando de la misma forma que lo hacía antes de los cambios introducidos. También era importante realizar pruebas para las nuevas correcciones y funciones, tratando de comprobar si funcionaban bien y si no provocaban que el resto de las funcionalidades no se llevasen a cabo de forma correcta.

Como comentaremos en el **Capítulo 7**, tuvimos una buena oportunidad para generar los ejemplos desde cero, lo que nos permitió generar casos de uso grandes en los que interactúan muchas partes de la aplicación. También decidimos, para algunos ejemplos, no generarlos desde cero, sino modificar uno para llegar a otro, tratando de realizar pruebas sobre las funcionalidades que no están dedicadas a la creación sino a la modificación (renombramiento, cambios en participación / cardinalidad, cambios en dominios, debilitar entidades, cambios en atributos, etc.) y eliminación. También era importante comprobar que la traducción a los modelos lógico y físico se realizaba como antes, y, en general, que se realizaba de forma correcta.

La reconstrucción de los ejemplos era una buena forma de comprobar casos de uso, pero teníamos que buscar que se realizasen el máximo de acciones posibles, para lo cual esta reconstrucción de los ejemplos no bastaba: era necesario diseñar más casos de uso buscando nuevas situaciones donde probar la aplicación.

Exponemos una lista en la que incluimos la mayoría de las acciones que llevamos a cabo en nuestras pruebas, tanto en las correspondientes a los ejemplos como en el resto de ellas:

- Creación de agregación en una relación, renombramiento de la agregación, eliminación de la agregación, creación de la misma agregación, eliminación de la relación.
- Creación de relación, asociación de entidades a relación, asociación de una misma entidad dos veces, renombramiento de relación, renombramiento de entidades, modificación de participación / cardinalidad / rol, eliminación de una entidad, desasociación de una entidad, eliminación de la relación.
- Creación de relación IsA, asignación entidad padre, asociación entidades hijas, cambio entidad padre, cambio entidad hija.
- Creación de atributo, modificar su dominio, su nombre, su condición de atributo multivalorado, unique, compuesto, clave primaria, renombrar atributo…
- Crear entidad normal, crear entidad débil, renombrar entidades.
- Eliminación de atributo con sub-atributos.
- Creación de elementos con nombres que ya existían, renombramiento de elementos en los que se introduce un nombre que ya existe.
- Creación de dominios con valores no válidos.
- Modificación a atributo no compuesto de un atributo compuesto con sub-atributos.
- Exportación de modelos.
- Deshacer / Rehacer.
- Asignación de valores no compatibles entre cardinalidad y participación.

- Creación de un nuevo dominio, asignación de dicho dominio a atributo existente, creación de atributo con nuevo dominio, renombramiento de nuevo atributo.
- Eliminación de entidades con varios atributos, y subatributos, y asociadas a relaciones.
- Inserción / eliminación de restricciones en la tabla de restricciones de un elemento.
- Inserción / eliminación de uniques en la tabla de uniques de un elemento.
- Carga / guardado de proyectos.
- Debilitar entidad a través de modificar entidad, debilitar entidad usando como nombre de relación uno ya existente, modificar una entidad débil y pasarla a entidad fuerte a través de modificar entidad.
- Modificación de varios campos en los cuadros de modificación de elementos.
- Traducción a modelo lógico y modelos físicos (se iban alternando los distintos tipos de SGBD aceptados) de los ejemplos, comprobación de su corrección. También se realizaron traducciones en muchos otros casos de uso y se evaluó su corrección.
- Uso y selección de conexiones con DBMS (DataBase Management System).

Hay acciones que se pueden llevar a cabo desde distintos puntos de la aplicación, en cuyo caso se llevó a cabo dicha acción desde cada punto que lo permitía.

Capítulo 7 - Cambios en archivos de ejemplo

Como comentamos en la sección **5.3**, en ocasiones la tabla de uniques no estaba actualizada correctamente en los archivos que generaba la aplicación. Vimos, por otra parte, que al partir de un archivo de ejemplo e intentar hacer una nueva asociación entre una entidad y una relación, saltaba una excepción. Esto, sin embargo, no ocurría con el ejemplo número 6. La excepción indicaba que se había tratado de insertar un nodo donde no estaba permitido, y más concretamente que no se había encontrado el padre del nodo a insertar en la entidad, cuyo nombre era RelatnList. Creemos que esto se pudo deber a que los primeros ejemplos fueron generados cuando la aplicación, por alguna razón, no generaba bien este nodo.

Pensamos que sería una buena oportunidad para tomar los ejemplos y generarlos desde cero, porque sería una tarea útil para realizar pruebas sobre nuestra aplicación y, además, porque podíamos comprobar si estos problemas que habían ocurrido reaparecían. De esta forma, podíamos cambiar en la carpeta de ejemplos los archivos anteriores por los nuevos que había generado DBCASE.

Además, decidimos introducir algunos cambios en los dominios de algunos atributos, en los ejemplos 1 y 4. Hicimos esto con el objetivo de mostrar nuevos dominios en los ejemplos.

Capítulo 8 - Conclusiones y trabajo futuro

Desde el principio este proyecto ha sido muy complejo y un reto para nosotros ya que nos encontramos con un código muy difícil de entender debido a las diferentes iteraciones que ha sufrido en las cuales los distintos grupos de alumnos añadieron diferentes funcionalidades sin atender a la unicidad del código. A lo largo de los años esto se fua agravando hasta el punto en que algunas clases tenían el nombre es español, otras en inglés y lo mismo pasó con los métodos y variables.

Nuestros compañeros del año pasado en este mismo capítulo dijeron lo siguiente: "El hecho de que el proyecto se haya ido desarrollando a lo largo de diversos años supone dos cosas. La primera es que ha ido evolucionando de manera incremental, es decir, las funcionalidades se han ido añadiendo una tras otra teniendo en cuenta siempre el estado del proyecto en el pasado. La segunda es que han sido muchas las personas que han colaborado a la implementación de DBCASE. Por estos motivos aparecen complicaciones a la hora de la comprensión del código habituales en este tipo de proyectos. Es recomendable adaptarse a la manera en que está implementada la aplicación, utilizar una nomenclatura similar en el nombre de métodos o variables, que aclaren de un simple vistazo cuál es su función, para una mayor legibilidad por parte de futuros programadores" (Ibáñez Martínez & Derecho Prieto, 2022-2023)

Terminamos muy satisfechos con el resultado final, ya que la tarea más compleja que fue la refactorización del código se logró. La dificultad de esto no residía en hacer rehacer la lógica de la aplicación sino en hacerlo bloque por bloque y asegurarse que seguía funcionando perfectamente. En el mes de octubre nos encontramos con que la clase principal "Controller" tenía más de 5000 líneas de código, no existían factorías, ni comandos y el único patrón de diseño implementado era el Modelo Vista-Controlador. Al acabar la refactorización el proyecto consta de una clase "Main" mucho más

pequeña, factorías abstractas, factorías, comando, etc. Lo que permitirá a futuros alumnos poder evolucionar la aplicación de una forma mucho más rápida. Además, mientras trabajábamos en esta refactorización fuimos capaces de encontrar errores no identificados y corregirlos. Este proceso comenzó en octubre con la migración a Maven y la ordenación de recursos y culminó en el mes de mayo cuando las ultimas validaciones de la refactorización terminaron. A lo largo de estos meses el trabajo ha sido irregular con momentos en los que éramos capaces de avanzar rápidamente y otros en los que estábamos semanas e incluso meses sin ver avances lo cual en ciertos momentos llegó a ser frustrante pero poco a poco fuimos viendo a luz al final del túnel.

El proceso de desarrollo del proyecto ha sido un viaje lleno de desafíos y aprendizaje. Desde sus inicios, hemos enfrentado numerosos ensayos y errores, así como la realización de extensas pruebas tanto del código como de la propia aplicación. La curva de aprendizaje, inicialmente lenta, ha ido incrementándose gradualmente a medida que nos sumergíamos más en el proyecto. Una de las dificultades que hemos enfrentado ha sido la identificación de errores, un proceso que ha resultado ser costoso y a menudo confuso. Frecuentemente nos encontrábamos debatiendo si los problemas que encontrábamos eran originados por fallos en el código recientemente creado o si se trataba de errores arrastrados desde versiones anteriores del software. Esta incertidumbre añadía una capa adicional de complejidad a la resolución de problemas, haciendo que cada corrección fuera un desafío único.

Sin embargo, a pesar de los obstáculos, cada hito alcanzado ha sido motivo de gran satisfacción. Ver la evolución real del proyecto, con la incorporación de nuevas funcionalidades y la reducción de errores correspondiente, nos ha llenado de motivación y orgullo por el trabajo realizado. DBCASE, nuestra aplicación destinada a la UCM para su uso en la asignatura de *Bases de Datos* ha sido el centro de nuestro esfuerzo y dedicación a lo largo de este curso.

Al principio, nos enfrentamos a dificultades y tuvimos que reconstruir el repositorio dos veces en el mes de septiembre. Sin embargo, con el tiempo, hemos aprendido a utilizarlo de manera más efectiva y hemos descubierto todas las funciones que ofrece, como el "Dependabot alert" o los informes que proporciona para evaluar la calidad del código generado durante la refactorización.

De cara al futuro de DBCASE faltan por completar la mayoría de los objetivos propuestos para esta versión pero que debido a la refactorización explicada con detalle anteriormente no hemos podido implementar. A destacar:

- Actualización de librerías - El cambio más importante en nuestra opinión en la reingeniería del renderizado del diagrama y la consecuente actualización de las versiones de las librerías y evitar así una vulnerabilidad de 9,8/10 que permite la ejecución de código remoto.

- Añadir PostgreSQL, SQL Server y DB2 a los posibles SGBD – Esto haría a la aplicación mucho más versátil ya los tres son BBDD relacionales muy utilizadas a nivel profesional

- Reingeniería de bases de datos - Consiste en la capacidad de diseñar una base de datos a partir del esquema lógico o físico, incorporando las tablas o consultas necesarias. Posteriormente, se puede presentar el esquema conceptual correspondiente. Para abordar esta tarea, es esencial codificar un proceso sintáctico que interprete los datos introducidos en los paneles lógico o físico y genere los mensajes necesarios para la inserción de los elementos en el esquema conceptual. Además, es fundamental incorporar mensajes de errores o avisos para informar al usuario sobre posibles problemas o inconsistencias en el proceso de diseño de la base de datos.

- Consideramos conveniente el cambio de versión de Java a una superior como Java 11 o Java 17, cualquiera de las dos versiones supondría un cambio de calidad de vida ya que incluyen funciones de forma nativa que en Java 1.8 necesita de librerías externas. El motivo por el que no se hizo es que el proyecto tiene numerosas dependencias con clases internas de Java que quedan obsoletas en Java 9 y no dispusimos de tiempo suficiente para hacerlo.

- Desde el proceso de traducción al modelo lógico, se lleva a cabo un proceso de desambiguación para resolver posibles casos en los que dos atributos tengan el mismo nombre, dominio y poseedor en una misma tupla. No obstante, este proceso no modifica el nombre almacenado del atributo en la clase Tabla, sino que se lleva a cabo de cara a la impresión en el panel del modelo lógico. Por otra parte, a la hora de imprimir las restricciones de clave foránea, y a la hora de generar el modelo físico, se llevan a cabo otros procesos de desambiguación que son distintos. Si hay atributos de mismo nombre, dominio y poseedor en una tupla, en ocasiones la traducción al modelo físico genera resultados erróneos porque no desambigua correctamente dichos atributos, y no la hace de la misma forma que en el modelo lógico. Nos gustaría proponer un cambio en este aspecto, para lograr un proceso de desambiguación uniforme que modifique los nombres de los atributos en la clase Tabla al principio de la traducción al modelo lógico, y que de esa forma el modelo lógico, la función de imprimir restricciones de clave foránea, y el modelo físico accedan a los mismos nombres en un estado no ambiguo.

A nivel personal, consideramos que este proyecto ha sido una oportunidad invaluable para el crecimiento y el desarrollo profesional. Se ha asemejado mucho a la

experiencia de trabajar en un proyecto en una empresa, desde la planificación y distribución de tareas hasta la comprensión y adaptación de código ajeno. Además, hemos adquirido habilidades importantes en el uso de herramientas como Git/GitHub y Eclipse, así como descubierto nuevas posibilidades en el lenguaje de programación Java. Además, lo largo de este proyecto, hemos observado cómo nuestras habilidades han ido mejorando progresivamente, especialmente a medida que hemos recurrido con mayor frecuencia a las opciones de búsqueda de Eclipse e IntelliJ. Sin estas herramientas, habría sido prácticamente imposible trabajar en un proyecto con tantas líneas de código. Además, hemos experimentado un notable crecimiento en el uso y dominio de Java.

8.1 Objetivos cumplidos

Exponemos a continuación una lista con los objetivos cumplidos a lo largo del desarrollo del proyecto.

1. Proyecto cambiado a Maven para facilitar la importación, la gestión de las librerías de cara al futuro y la gestión de vulnerabilidades.
2. Rehacer la estructura de paquetes y rehacer la importación desde una carpeta RESOURCES.
3. Se ha solucionado el siguiente problema: No aparece la flecha en una relación recursiva con cardinalidad 1.
4. Creación de comandos para encapsular las funcionalidades más complejas que hace el controlador, y creación de una factoría para su uso.
5. Reorganización de la forma en que se obtienen listas de elementos desde vistas y controlador.
6. Resolución de rupturas de arquitectura.
7. Estandarización de los frames de la aplicación usando Parent_GUI.
8. Creación de una factoría para la gestión de los frames que usa la aplicación.
9. Se ha solucionado el siguiente problema: Al llevar a cabo cualquier acción que requiera apertura de ficheros, se pierden las restricciones de cardinalidad 1.
10. Se ha solucionado el siguiente problema: En una relación recursiva con una entidad, la entidad aparece siempre con la misma cardinalidad y participación en el diagrama.

11. Encapsulación de la gestión de los servicios en una factoría.
12. Implementación de una forma de traducir algunos enumerados en otros (FactoriaTCCtrl).
13. Uso del patrón Contexo para eliminar la referencia que los servicios mantenían hacia el controlador. Se eliminaron todos los métodos que estaban en el controlador, que tenían la función de responder a los mensajes que les enviaban los servicios directamente. Creación de una función (tratarContexto) destinada a reunir y generalizar todas las funcionalidades que llevaban a cabo estos métodos del controlador.
14. Creación de una factoría para mensajes de error.
15. Creación de clase Config para quitar al controlador la responsabilidad de almacenar algunas variables de configuración, y eliminación de las referencias que algunas clases mantenían del controlador con este objetivo (por ejemplo, DAOAtributos).
16. Separación a la clase Main y adaptación del código dedicado a inicializar la aplicación.
17. Separación de algunas funciones de uso general (ordenación de listas, partición de vectores...) a una clase específica para ellas.
18. Cambio y adaptación en la relación de las clases GeneradorEsquema y ValidadorBD.
19. Creación de una clase abstracta que implementa funcionalidad que usan todos los objetos DAO.
20. Traslado de la función dedicada a la creación del almacén persistente a la capa de persistencia.
21. Actualización de muchas librerías (pom.xml).
22. Guardar como XML modelo lógico y físico.
23. Implementación de excepciones y manejo de ellas.
24. Traslado a comandos, adaptación y generalización de código dedicado a la creación de entidades débiles.
25. Rediseño e implementación de contexto y tratarContexto para que permitan tratar sub-contextos derivados de una operación principal.
26. Traslado de las clases NodoEntidad y EntidadYAridad al paquete de transfers.
27. Recreación de archivos de ejemplos para solucionar problemas en su estructura.
28. Resolución de error que ocurría al renombrar un atributo unique.
29. Creación de diagramas de clase UML para todos los paquetes y sub-paquetes de la aplicación.

30. Creación de diagramas de secuencia para un caso de uso completo de la aplicación (editar el campo not-null de un atributo).
31. Creación de diagramas de secuencia para algunas nuevas interacciones (ejecutar un comando, activación de frame usando FactoriaGUI, tratar contexto).
32. Resolución de problemas en la traducción de ciertas relaciones al modelo lógico / físico (ver sección **5.5**).
33. Actualización de la ventana dedicada a editar cardinalidad y participación para que muestre los valores que se tienen almacenados según la entidad y rol que se elija.

Introduction

DBCASE 4.0 is a desktop application focused on the design and implementation of SQL relational databases. By designing a conceptual model, it generates the logical and physical models, also allowing for the implementation of the latter.

A relational database is a type of database that organizes information into related tables. It is based on the relational model proposed by Edgar F. Codd in the 1970s. In this model, data is stored in tables with rows and columns. Each row represents a specific entity, and each column represents an attribute of that entity. Relationships between entities are established by matching values in specific columns (primary keys and foreign keys). In a relational database, data integrity is maintained through constraints such as primary and foreign keys, which ensure the coherence and consistency of the data. Additionally, it uses a structured query language (SQL) to perform operations such as inserting, updating, deleting, and querying data.

At the Faculty of Informatics, in the different degree programs it offers, relational databases are used in a multitude of subjects such as Databases, Advanced Databases, Software Engineering, etc. However, this work will be especially useful for Database students since it is in the subject where the fundamentals of database design are presented and where students begin to design their first Entity-Relationship diagrams and learn the basic syntax of SQL and MySQL.

Motivation

Our main motivation in choosing this topic over others was to expand our knowledge in designing relational databases and to increase our experience in designing and implementing desktop applications based on Java.

Furthermore, as an iterative Final Degree Project (TFG), we had the challenge of continuing what others had done in previous years, starting in 2008 with the idea of Yolanda García Ruiz and continued by Fernando Sáenz Pérez, our supervisor. In 2008, the project was called DBDT (Database Design Tool), which in 2009 changed its name to DBCase (Database Computer-Aided Software Engineering), being the fourth iteration in desktop version as in the year 2020 the project was divided into the desktop version based on Java and the web version based on HTML, CSS, and JavaScript.

This tool offers an intuitive and interactive graphical interface that allows users to design the behavior of a database using Entity-Relationship diagrams. Additionally, based on the conceptual schema created, it displays the corresponding translations to logical and physical models. This allows verifying the correctness of the conceptual schema design. It can also generate the code associated with creating each table in a format compatible with different database management systems (DBMS), such as Oracle, MySQL, or Microsoft Access.

Although this tool is primarily designed for use in academic environments, where it facilitates students' understanding of database theory through practical cases, its usefulness is not limited to this area. The ability to generate SQL code allows the execution of real scripts in various DBMS, making it valuable for both educational and professional environments.

Goals

The most priority tasks of this project are, firstly, those related to correcting its functioning and, secondly, expanding its functionality to bring it up to the same level as DBCASEweb (in particular, adding aggregations and cardinalities with two notations 1-N and min-max). Finally, extension tasks would be addressed (performance analysis, normalization/denormalization, and reengineering).

1. Review the generation of the relational schema. There are errors at least in the keys of the 1-N relationships. DBCASEweb should do it correctly, but it could still have errors.
2. Finish the implementation of aggregations.
3. Optimization of design: performance analysis (volumes, frequencies, operations...).
4. Normalization/Denormalization.
5. Reengineering of the physical/relational design.
6. Marked properties are not well visible with dark backgrounds.
7. If the relationship has two or more different roles towards the same entity, it does not give a different name to each field but repeats them.
8. Add alignment lines to assist in placing elements in the E-R diagram.
9. Allow dragging the diagram by clicking and dragging.
10. The arrow does not appear in a recursive relationship with cardinality 1.
11. Allow using the right mouse button on the arcs to edit their properties.
12. Yes/No dialog boxes. Allow using the keys Y and N (careful in other languages: in Spanish S and N, in Russian: D and N, in French O...).
13. Save application state (last opened file, language, zoom, open panels, connections, etc. In general, everything that can be configured or displayed when the application is reopened is generally to work with the last file). Resolved, except for zoom.
14. Help -> Open user manual.

15. Double-clicking on an object should open a dialog box to modify it (practically the same as the one used for its creation).
16. Menus bar: add View (Design Panel Zoom (to allow selecting a % as in Word, for example), Graph Panel Zoom, Graph Panel (to show it or not), Domain Panel (to show it or not), Event Panel (to show it or not)).
17. Add toolbar (Open, Save, Close, Print,...) icons.
18. Able to use cursors to move objects.
19. Allow saving and printing the validation log.
20. Add menu items for all actions. In particular for actions from the context menu (Insert entity, ...) and for the buttons on the Code Generation tab.
21. New Edit menu: Copy, Cut, Paste, Undo, and Redo. Shortcuts: Ctrl-C, Ctrl-V, Ctrl-X, Ctrl-Z, and Ctrl-Y.
22. Provide drawing aids: grid (show, define, align objects to the grid), align (top, bottom, left, right, horizontal center, vertical center), draw reference lines (as in visual studio .net), arrange horizontally, arrange vertically. Add a toolbox (similar to Word), Rotate (left, right, 90º, 180º).
23. Zoom in on the logical and physical schema sections. Font sizes could be changed using the mouse wheel (entries could also be added in the View menu).
24. Drag and drop elements from the left panel of conceptual schema design to the drawing canvas.
25. Add PostgreSQL, SQL Server, and DB2 to possible DBMS.
26. Automatic graph rearrangement. That is, automatically rearrange graphic objects.
27. Allow defining uniqueness on more than one attribute, for example, by selecting all attributes that are a superkey and right-clicking on them.
28. Table integrity constraints as a logical predicate (condition to be met) expressed in SQL. For example, a+b=10 for the following schema:

29. Specify the zoom level to the E/R schema panel (small, the left one) and the design panel (large, the right one). Add options in the menu bar (Zoom level and Show/hide panels).
30. Add the keyboard shortcut Ctrl+Mouse Wheel for zooming (also keeping the one that already exists with just the mouse wheel).

When we started to review the code, we realized that in order to carry out all the previously mentioned tasks we first had to do a global refactoring of the code in order to make it more clear, accessible, maintainable, updatable and efficient applying all the knowledge about software engineering learned during the degree. So, after discussing it with our tutor we decided that the priority task of this project was going to be the refactoring of the code and once completed we would start with the objectives previously mentioned at this point.

At the time of writing this report, the priority objective of refactoring was completed very satisfactorily. The code has been made much more readable but more importantly, much more maintainable and updatable, fragmenting classes that had more than five thousand lines to much more manageable ones, factories, contexts, commands, etc. have been created. On the other hand, the folder structure has been reorganized, the project has been migrated to Maven to be able to add new libraries in a much easier way and we have worked on some of the objectives set by our tutor, achieving some of them and leaving others commented because they were not completed. In the section **"List of achieved goals"** we expose a complete list of the completed goals.

Work plan

In this section, the work plan to achieve the objectives mentioned above is outlined. The first step was to contact Fernando Sáenz Pérez to have an initial meeting with him. During this meeting, he explained the basics of the project and how we should proceed. Once we had access to the code, we began analyzing it and understanding

it and we proposed what is described in the previous section. The first action taken was migrating the project to Maven to simplify the importation of libraries and enable easier updates. Following this, we standardized the class names, organized the graphical resources, and sorted the language packages. Once these tasks were completed, we began the detailed process of code refactoring, as explained in **Chapter 4 - Code Refactoring**. While this refactoring process was underway, progress was also made, to a lesser extent, on the list of objectives mentioned earlier, as well as on the writing of this report.

A Gantt chart of our project is shown in Figure 1.

Document structure

Throughout this report we describe all the tasks undertaken. First, in **Chapter 2**, we describe the creation of UML diagrams, whose complete description can be found in the appendices of this document. Then, in **Chapter 3**, we describe everything related to the migration to Maven and the reorganization of resources. The longest chapter of this document is **Chapter 4**, which explains in twenty-two sections the whole process of refactoring the code. Then, three chapters follow, **Chapter 5**, **Chapter 6** and **Chapter 7**, describing other tasks that were done to finish with the conclusions, Chapter 8 (translation to English present in the section **"Conclusions and future work")**, in which we give our opinion about the result obtained and explain possible changes for the future that would improve the application.

Conclusions and future work

From the outset, this project has been very complex and challenging for us, as we encountered code that was difficult to understand due to the different iterations it had gone through. Over the years, this complexity escalated to the point where some classes had names in Spanish, others in English, and the same occurred with methods and variables.

"The fact that the project has been developed over several years implies two things. The first is that it has evolved incrementally, this means that many functionalities have been added one after the other, always considering the state of the project in the past. The second is that many people have collaborated in the implementation of DBCASE. For these reasons, complications could arise when it comes to understanding the code that are common in this type of project. It is advisable to adapt to the way in which the application is implemented, to use a similar nomenclature in the name of methods or variables, to clarify what their function is, for greater readability by 62 future programmers." (Ibáñez Martínez & Derecho Prieto, 2022-2023)

We are very satisfied with the result, as the most complex task, code refactoring, was achieved. The difficulty lay not in redoing the logic of the application, but in doing it block by block and ensuring that it still worked perfectly. In October, we found that the main "Controller" class had more than 5000 lines of code, there were no factories or commands, and the only design pattern implemented was the Model-View-Controller. After the refactoring, the project consists of a much smaller Main class, abstract factories, factories, commands, etc. This will allow future students to evolve the application much more quickly. Additionally, while working on this refactoring, we were able to identify and correct previously unidentified errors. This process began in October with the migration to Maven and the organization of resources and concluded in May when the final

validations of the refactoring were completed. Throughout these months, the work has been irregular, with moments where we were able to progress rapidly and others where we went weeks or even months without seeing progress, which at times was frustrating, but gradually we saw the light at the end of the tunnel.

The project development process has been a journey full of challenges and learning. From the beginning, we faced numerous trials and errors, as well as extensive testing of both the code and the application itself. The learning curve, initially slow, gradually increased as we delved deeper into the project. One of the difficulties we faced was identifying errors, a process that proved to be costly and often confusing. We frequently debated whether the problems we encountered originated from flaws in the newly created code or if they were issues inherited from previous versions of the software. This uncertainty added an additional layer of complexity to problem-solving, making each correction a unique challenge.

However, despite the obstacles, each milestone achieved has been a source of great satisfaction. Seeing the real evolution of the project, with the incorporation of new functionalities and the corresponding reduction of errors, has filled us with motivation and pride for the work done. DBCASE, our application destined for use at UCM in the Databases subject, has been the focus of our effort and dedication throughout this course.

Initially, we faced difficulties and had to rebuild the repository twice in September. However, over time, we learned to use it more effectively and discovered all the features it offers, such as the "Dependabot alert" or the reports it provides to assess the quality of the code generated during the refactoring.

Looking ahead to the future of DBCase, most of the proposed objectives for this version remain to be completed, but due to the detailed refactoring explained earlier, we have not been able to implement them. Key highlights include:

- Updating libraries: The most important change in our opinion is the reengineering of the rendering of the diagram and the consequent update of the versions of the libraries to avoid a vulnerability rating of 9.8/10 that allows remote code execution.

- Adding PostgreSQL, SQL Server, and DB2 to possible DBMS: This would make the application much more versatile as all three are widely used relational databases in professional settings.

- Database reengineering: This involves the ability to design a database from the logical or physical schema, incorporating the necessary tables or queries. Subsequently, the corresponding conceptual schema can be presented. To address this task, it is essential to code a syntactic process that interprets the data entered in the logical or physical panels and generates the necessary messages for the insertion of elements into the conceptual schema. Additionally, it is crucial to incorporate error or warning messages to inform the user about possible problems or inconsistencies in the database design process.

- We consider it convenient to upgrade the Java version to a higher one such as Java 11 or Java 17. Either of these versions would result in a better quality of life as they include native functions that in Java 1.8 require external libraries. The reason we didn't do so is that the project has numerous dependencies with internal Java classes that become obsolete in Java 9, and we didn't have enough time to do it.

- From the translation process to the logical model, a disambiguation process is carried out to resolve possible cases in which two attributes have the same name,

domain, and holder in the same tuple. However, this process does not modify the stored name of the attribute in the table class but is carried out for printing in the logical model panel. On the other hand, when printing foreign key constraints, and when generating the physical model, other disambiguation processes are carried out which are different. If there are attributes of the same name, domain, and holder in a tuple, sometimes the translation to the physical model generates erroneous results because it does not disambiguate these attributes correctly and does not do it in the same way as in the logical model. We would like to propose a change in this aspect, to achieve a uniform disambiguation process that modifies the attribute names in the table class at the beginning of the translation to the logical model, so that the logical model, the foreign key constraint printing function, and the physical model access the same names in an unambiguous state.

On a personal level, we consider this project to have been an invaluable opportunity for growth and professional development. It closely resembled the experience of working on a project in a company, from planning and task distribution to understanding and adapting to others' code. Additionally, we have acquired important skills in using tools such as Git/GitHub and Eclipse, as well as discovering new possibilities in the Java programming language. Throughout this project, we have observed how our skills have progressively improved, especially as we have turned more frequently to the search options in Eclipse and IntelliJ. Without these tools, it would have been practically impossible to work on a project with so many lines of code. Additionally, we have experienced a notable growth in the use and mastery of Java.

List of achieved goals

The following is a list of the objectives achieved during the development of the project.

1. Project changed to Maven to facilitate the import, the management of libraries for the future and the management of vulnerabilities.

2. Redo the package structure and redo the import from a RESOURCES folder.

3. Fixed the following problem: The arrow does not appear in a recursive relation with cardinality 1.

4. Created commands to encapsulate the more complex functionalities that the controller does and created a factory for their use.

5. Reorganization of the form in which lists of elements are obtained from views and controller.

6. Resolving architecture breaks.

7. Standardization of application frames using Parent_GUI.

8. Creation of a factory for the management of the frames used by the application.

9. The following problem has been solved: When carrying out any action that requires opening files, the cardinality 1 restrictions are lost.

10. The following problem has been solved: In a recursive relationship with an entity, the entity always appears with the same cardinality and participation in the diagram.

11. Encapsulation of the management of services in a factory.

12. Implementation of a way to translate some enumerated elements into others (FactoriaTCCtrl).

13. Use of the Context pattern to eliminate the reference that the services maintained towards the controller. All the methods in the controller, which had the function of responding to the messages sent to them directly by the services, were eliminated. Creation of a function (tratarContexto) to gather and generalize all the functionalities carried out by these controller methods.

14. Creation of an error message factory.

15. Creation of a Config class to remove from the controller the responsibility of storing some configuration variables, and elimination of the references that some classes maintained from the controller for this purpose (for example, DAOAtributos).

16. Separation to the Main class and adaptation of the code dedicated to initializing the application.

17. Separation of some general use functions (list sorting, vector partitioning...) to a specific class for them.

18. Change and adaptation of the relationship between the classes GeneradorEsquema and ValidadorBD.

19. Creation of an abstract class that implements functionality used by all the DAO objects.

20. Transfer of the function dedicated to the creation of the persistent store to the persistence layer.

21. Updating of many libraries (pom.xml).

22. Saving as XML logical and physical model.

23. Implementation of exceptions and exception handling.

24. Transfer to commands, adaptation and generalization of code dedicated to the creation of weak entities.

25. Redesign and implementation of Contexto and tratarContexto to allow the treatment of sub-contexts derived from a main operation.

26. Transfer of the classes NodeEntity and EntityYArity to the transfers package.

27. Recreation of example files to solve problems in their structure.

28. Resolution of an error that occurred when renaming a unique attribute.

29. Creation of UML class diagrams for all packages and sub-packages of the application.

30. Creation of sequence diagrams for a complete use case of the application (editing the not-null field of an attribute).

31. Creation of sequence diagrams for some new interactions (execute a command, frame activation using FactoriaGUI, tartarContexto).

32. Resolution of some problems in the translation of certain relations to logical physical model (see section **5.5**).

33. Updating of the frame dedicated to modifying cardinality and participation for it to show the stored values given a certain entity and role.

CONTRIBUCIONES PERSONALES

Iván Pisonero Díaz

Como se comenta a lo largo de esta memoria, ambos estudiantes propusieron realizar una refactorización del código. Tras concluir ambos en que sería bueno tratar de disminuir el tamaño del controlador y empezar a encapsular algunas de las funcionalidades más complejas en comandos, Iván comenzó a desarrollar la estructura que ahora tiene la aplicación en cuanto a los comandos.

Después, conforme se fue comprendiendo mejor la aplicación, Iván propuso e implementó el resto de los cambios que se describen en el capítulo dedicado a la refactorización. El proceso de la identificación de posibles cambios a realizar y su implementación fue un proceso gradual: nuevas ideas surgían conforme se avanzaba, y también conforme se avanzaba iban surgiendo problemas que no se habían tenido en cuenta y que era preciso solucionar y evaluar, esto era puesto en común y discutido por ambos en numerosas reuniones.

En línea con esto, y en general para conseguir un software de la mayor calidad posible, Iván realizó parte de las pruebas descritas en el capítulo 6, conforme se iba avanzando en los cambios propuestos y también una vez concluidos.

Iván se encargó de realizar los diagramas que se pueden ver a lo largo de este documento y en el modelo IBM RSAD.

Además, junto con Javier, evaluó con detalle el problema del renderizado de relaciones e implementó las soluciones descritas en las secciones 5.1 y 5.2. Esto llevó

bastante tiempo debido a que el renderizado, como se puede ver los diagramas de las ilustraciones 17 y 39, es un proceso bastante complejo que hace uso de librerías que no conocíamos y de una gran cantidad de clases. Familiarizarse con este proceso fue un reto, para el que también la realización de diagramas provisionales resultó de gran ayuda.

Dentro también del ámbito de la renderización, Iván realizó algunas tentativas para solucionar el problema con las etiquetas de los roles en las relaciones recursivas que comentamos en la sección 5.2. Como comentamos, no se llegó a una solución satisfactoria.

Iván realizó los cambios descritos en las secciones **5.5** y **5.6**.

Ambos estudiantes compararon también casos de uso y soluciones para llegar a lo descrito en la sección **5.3** y en el **capítulo 7**.

En esta memoria, Iván ha contribuido en los capítulos:

- **Capítulo 2 -** Modelo y diagramas UML
- **Capítulo 4 -** Refactorización del código
- **Capítulo 5 –** Nuevas funcionalidades
- **Capítulo 6 -** Pruebas
- **Capítulo 7**- Cambios en archivos de ejemplo
- Apéndices A y B.
- Maquetación y revisión

Javier de Vicente Vázquez

Una vez creado el repositorio de GitHub y haber analizado la lista de objetivos y el código, decidió entre ambos estudiantes que lo óptimo sería hacer una refactorización de la mayoría del código. Está fue la primera y más importante decisión tomada nada más comenzar el proyecto y ocuparía la mayor parte del tiempo.

La primera tarea real fue la migración del proyecto a Maven explicada en el **Capítulo 3**, e importar todas las librerías verificando compatibilidades e intento actualizarlas en la medida de lo posible. Posteriormente hizo reestructuración de carpetas y de recursos estandarizando nombres y creando las clases con los Strings para facilitar la importación de estos y se probó en detalle que estos cambios no rompían nada.

Una vez terminada esta migración junto con Iván empezaron a ver cómo proceder a la refactorización, aquí uno de los principales problemas fue que la estimación hecha infravaloró mucho el tiempo que se tomaría el hacerla, ya que los planes originales eran terminarla en enero, durante ese tiempo Iván se dedicaría al 100% a la migración y Javier a la lista de objetivos. Hasta el mes de diciembre intentó avanzar con la lista de objetivos marcados por el tutor (**Capítulo 5**).

Desde el principio avanzar con los objetivos era muy difícil por varios motivos, el primero el código era muy difícil de seguir y hacer cualquier cambio llevaba semanas, y el segundo, Iván trabajaba simultáneamente en la refactorización, por lo que cada cambio provocaba infinidad de conflictos en GitHub ya que mientras en una rama se trabajaba en los objetivos, en la otra se cambiaban las mismas clases y métodos continuamente.

Así que a partir de aquí y hasta el final del proyecto Javier estuvo trabajando en la refactorización **Capitulo 4** junto con Iván, ambos fueron trabajando simultáneamente en todas las pruebas necesarias para asegurar que los pasos dados en la refactorización no provocases errores y fallos en la aplicación, este trabajo se describe en el **Capítulo 6**.

Después ambos continuaron añadiendo funcionalidades descritas en la lista de objetivos. Ciertos objetivos como el **5.1** o el **5.2** llevaron mucho tiempo ya que el renderizado del diagrama E-R usa librerías que ninguno de los dos conocía y supusieron un reto. Por otro lado, también junto con Iván se resolvió el problema de renombramiento de atributos unique (**5.3**), y continuó trabajando en el resto de las funcionalidades añadidas, o cuyo trabajo se avanzó en este proyecto. Todo este trabajo está también descrito en el **Capítulo 5**. Por último ayudó a hacer los casos de ejemplo del **Capítulo 7** durante la fase de refactorización, para asegurar que todo funcionaba correctamente.

En esta memoria Javier ha contribuido en los capítulos:

- **Capítulo 1 -** Introducción
- **Capítulo 2 -** Modelo y diagramas UML
- **Capítulo 3 -** Maven y reorganización de recursos
- **Capítulo 5 -** Otros cambios
- **Capítulo 8 -** Conclusiones y trabajo futuro
- Resumen inicial
- Traducciones al inglés
- Maquetación y revisión

Bibliografía: Ibáñez Martínez, A., & Derecho Prieto, M. (2022-2023). *TFG DBCASE 3.0*. UCM.

APÉNDICES

Apéndice A - Diagramas UML de clase

Exponemos a continuación los diagramas de clase que realizamos. Nos gustaría destacar que, si estos son visualizados usando la herramienta IBM RSAD a partir del modelo, se podrán ver con un mayor grado de detalle, y se podrá mostrar en ellos toda la información contenida en las clases que, en estos diagramas que exponemos, en ocasiones han sido ocultados debido al gran tamaño que adquirían dichos diagramas.

Vista

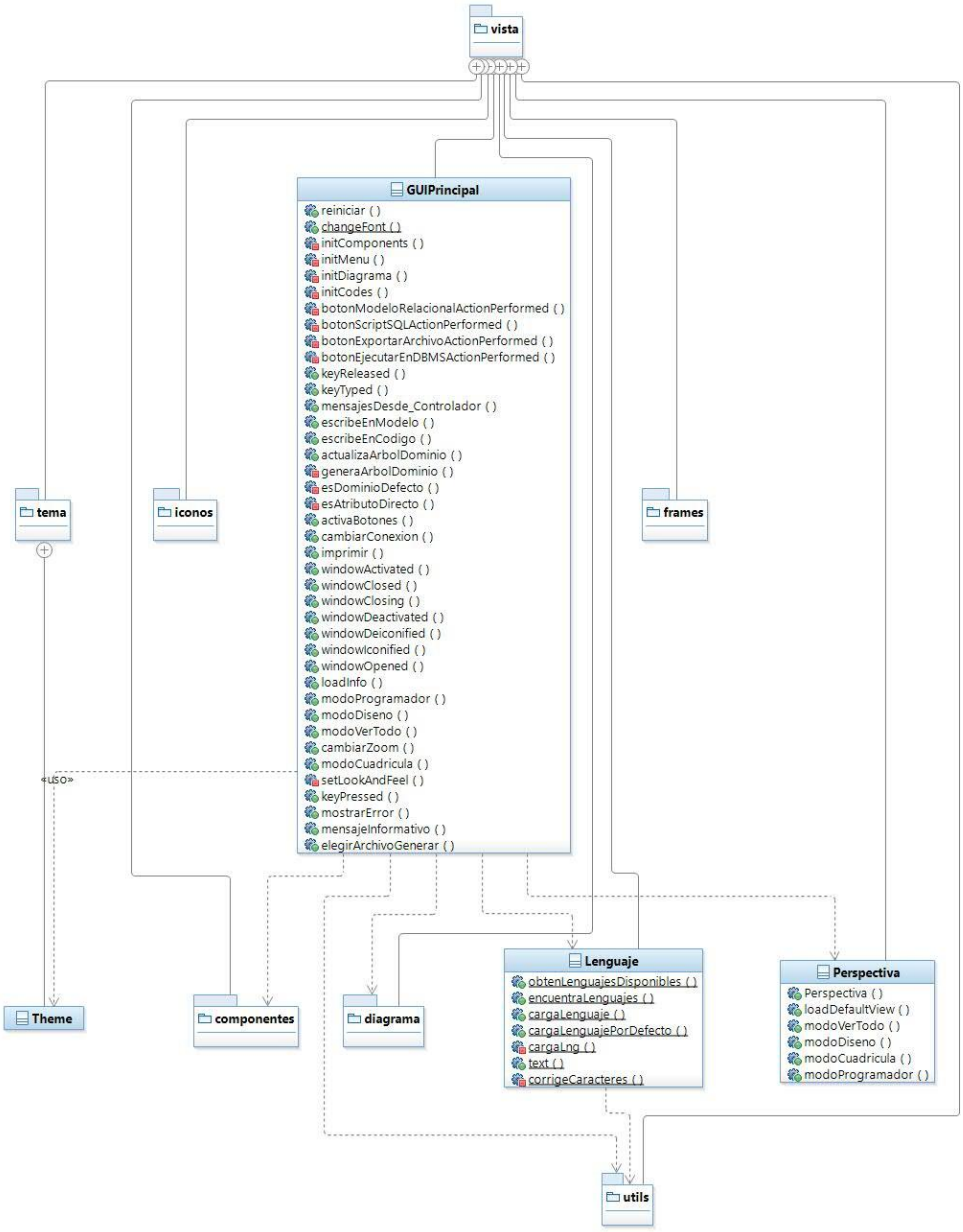

Ilustración 14: Diagrama de clase paquete vista

Componentes

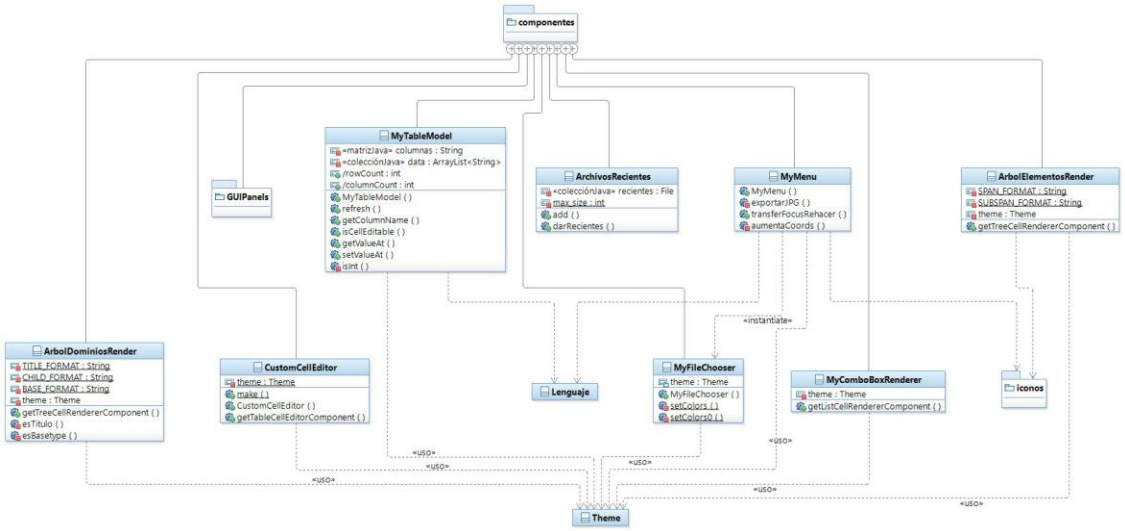

Ilustración 15: Diagrama de clase paquete vista.componentes

GUIPanels

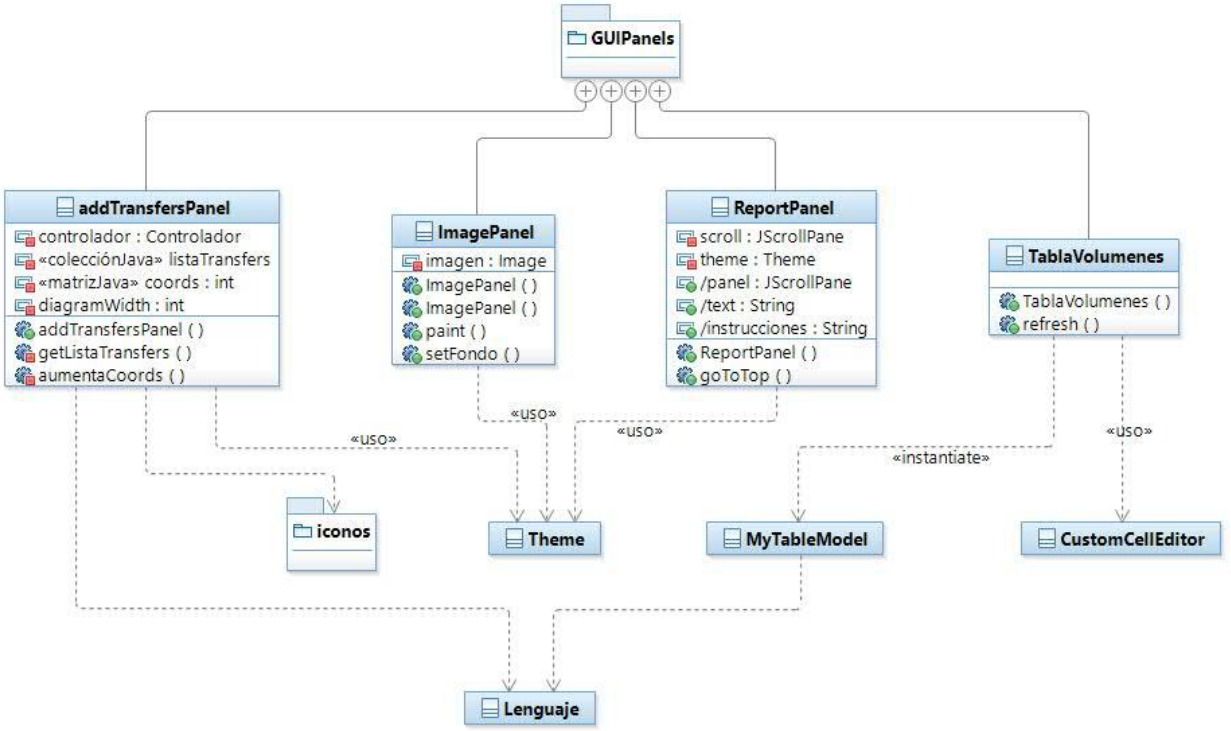

Ilustración 16: Diagrama de clase paquete vista.componentes.GUIPanels

Diagrama

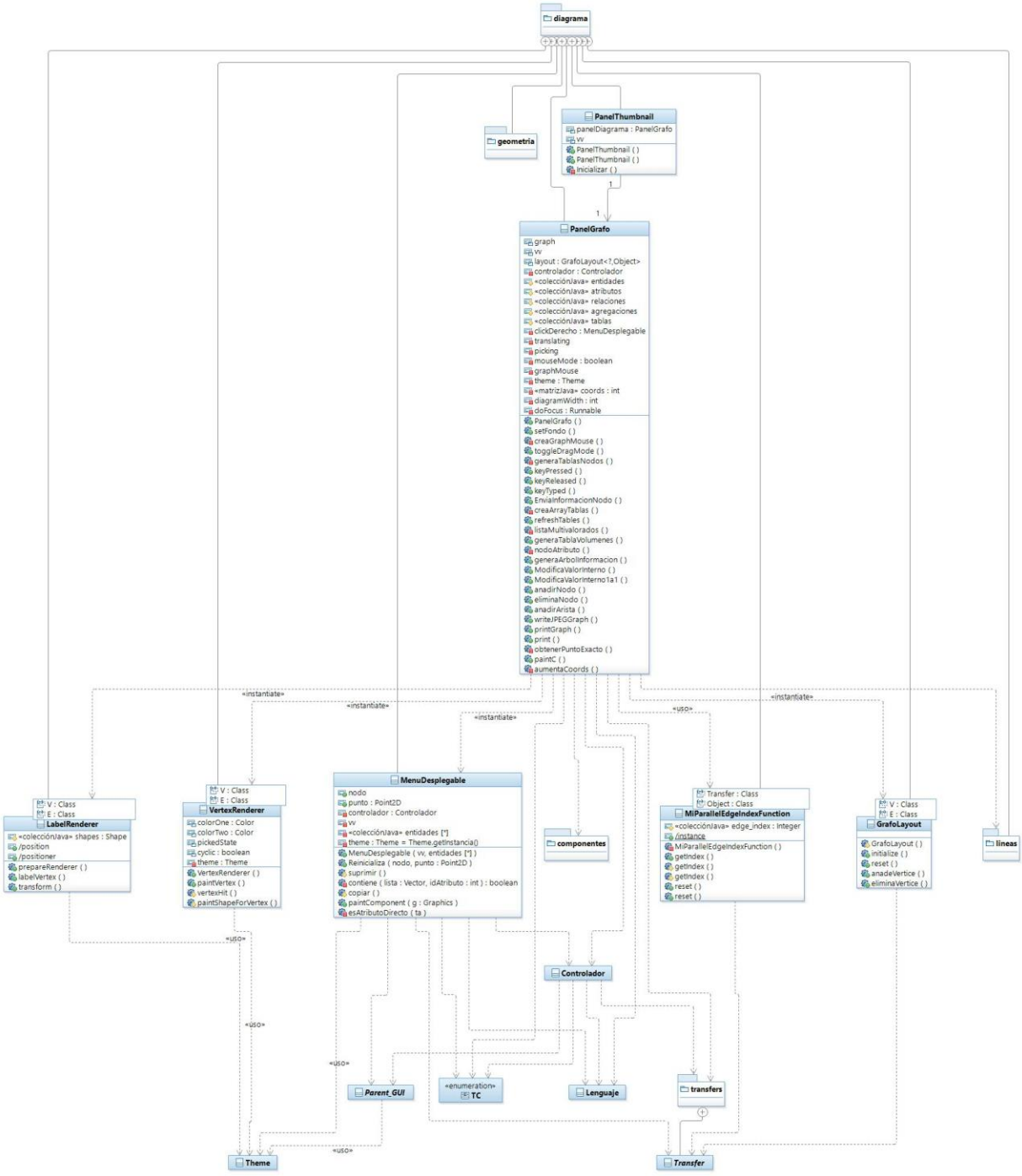

Ilustración 17: Diagrama de clase paquete vista.diagrama

Geometría

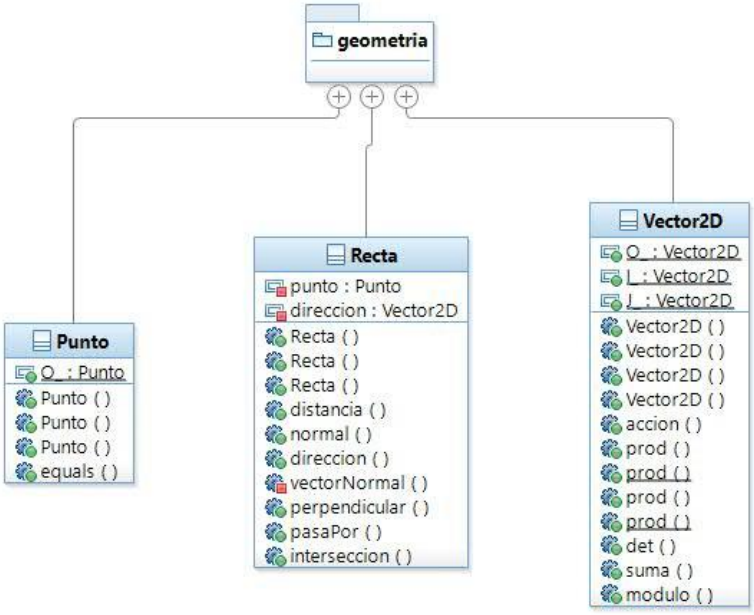

Ilustración 18: Diagrama de clase paquete vista.diagrama.geometria

Líneas

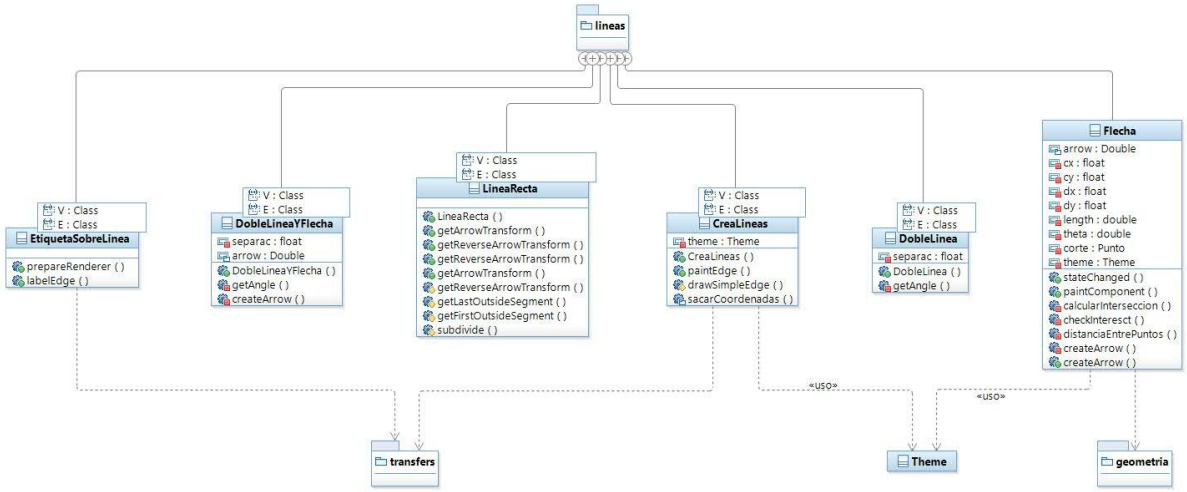

Ilustración 19: Diagrama de clase paquete vista.diagrama.lineas

Frames

Ver ilustración 2.

Iconos

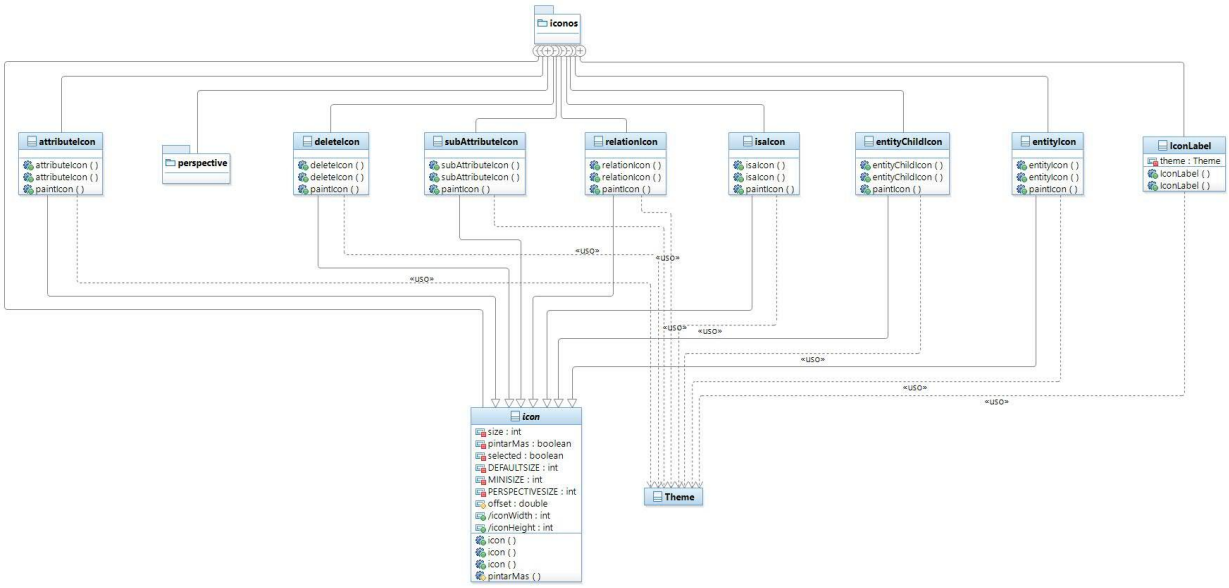

Ilustración 20: Diagrama de clase paquete vista.iconos

Perspectiva

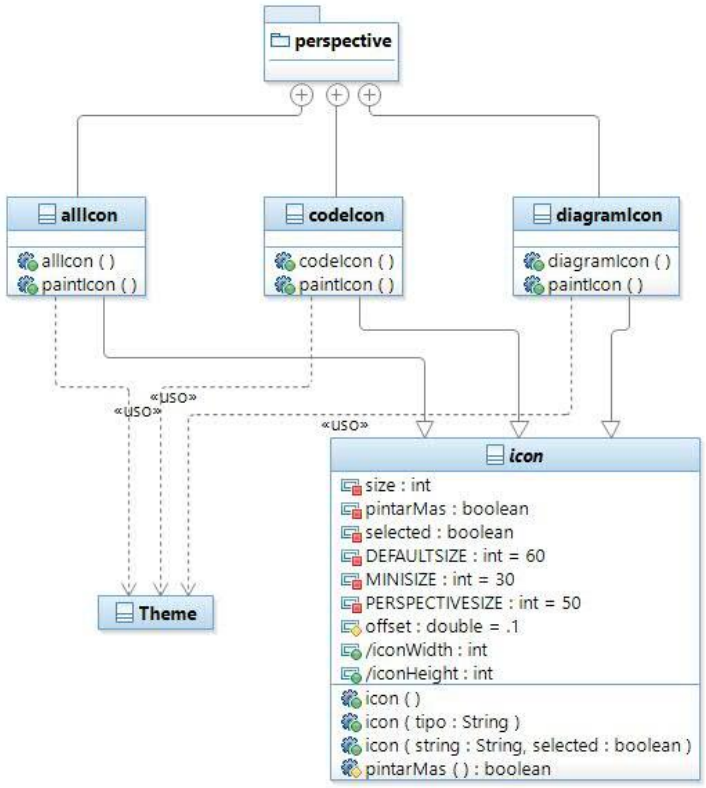

Ilustración 21: Diagrama de clase paquete vista.iconos.perspectiva

Tema

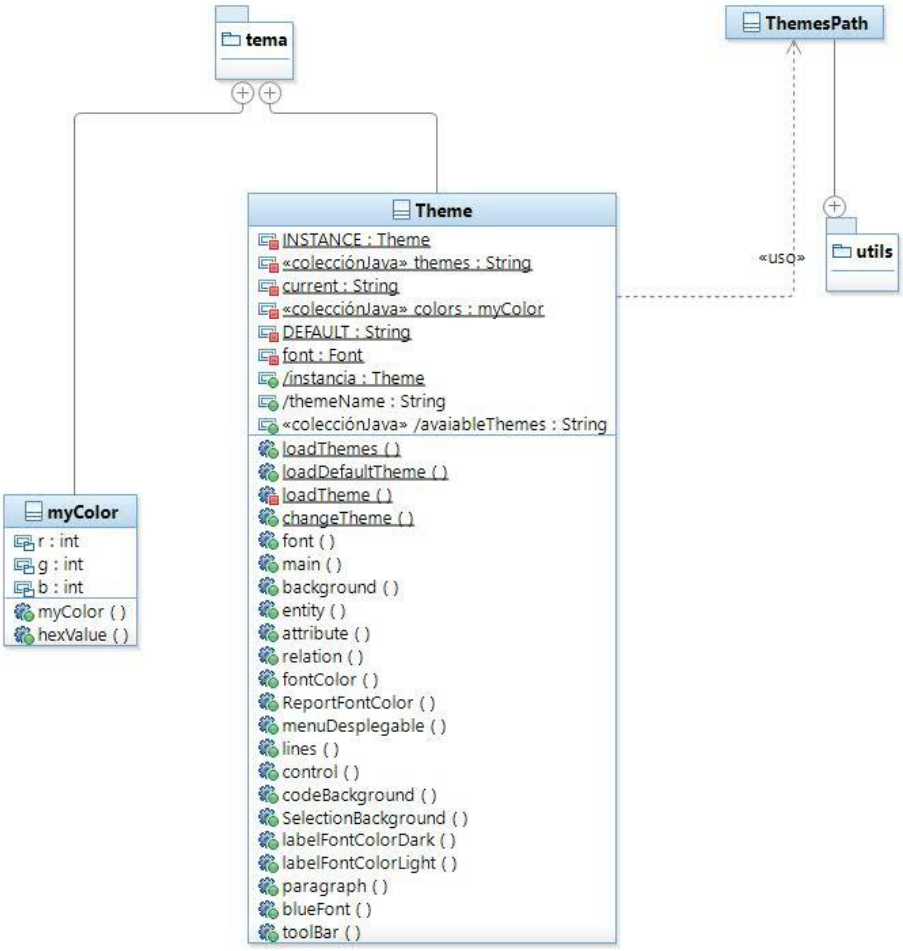

Ilustración 22: Diagrama de clase paquete vista.tema

Utils

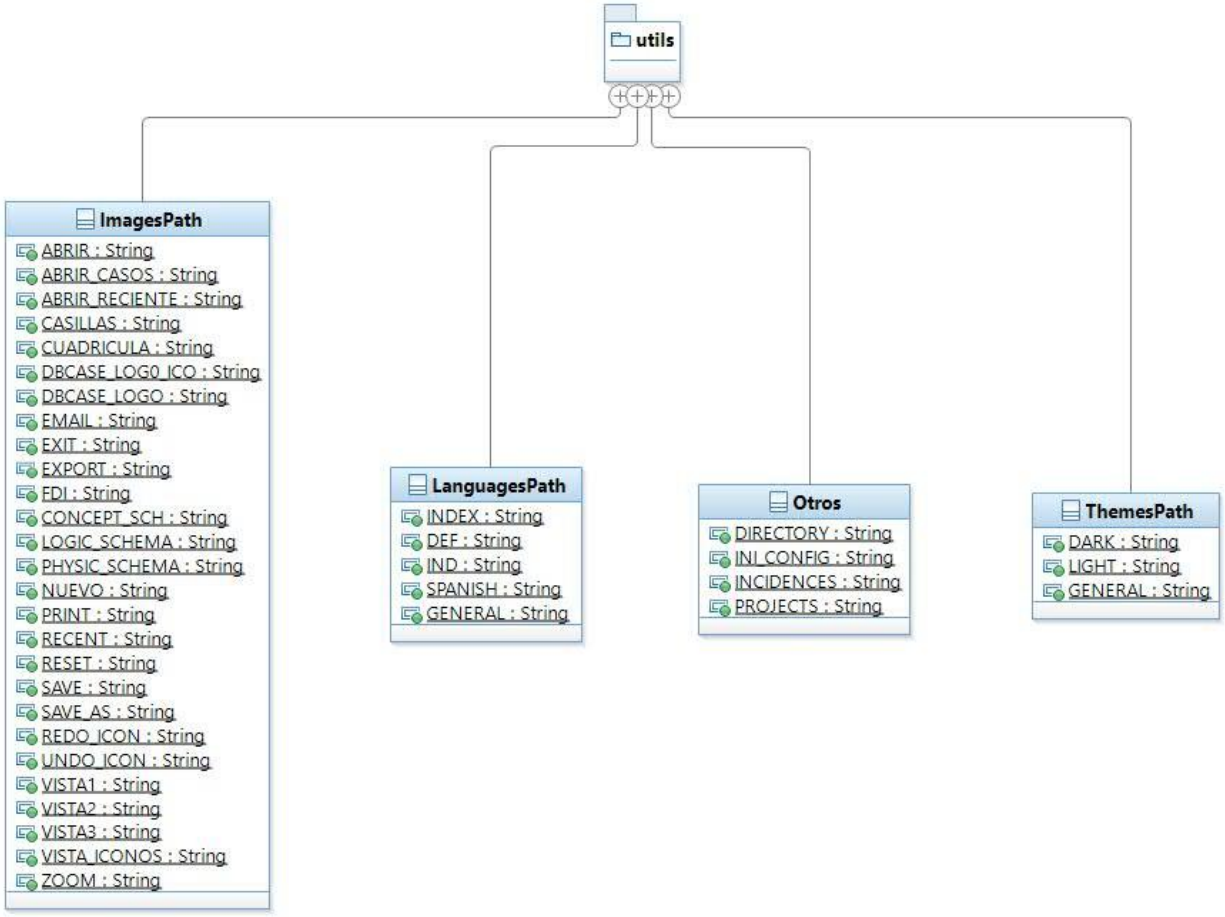

Ilustración 23: Diagrama de clase vista.utils

Controlador

Ver ilustración 3.

Comandos

Ver ilustración 4.

Agregación

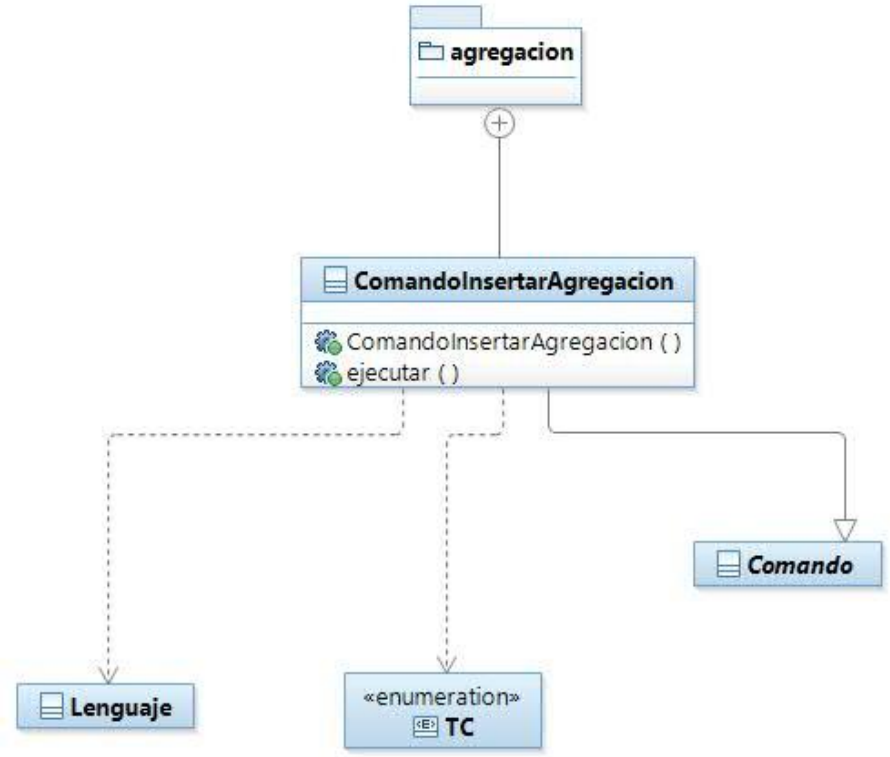

Ilustración 24: Diagrama de clase paquete controlador.comandos.agregacion

Atributo

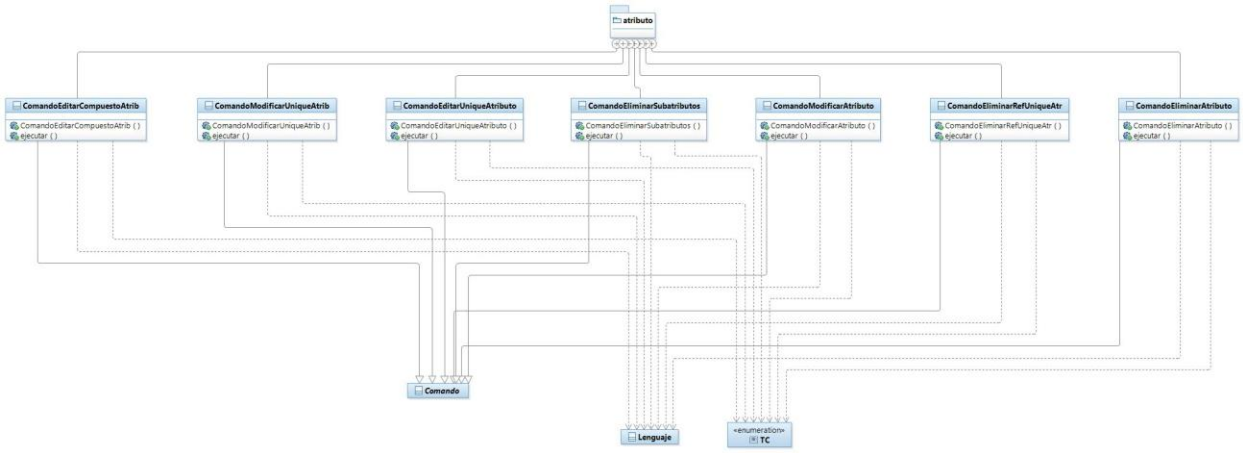

Ilustración 25: Diagrama de clase paquete controlador.comandos.atributo

Dominio

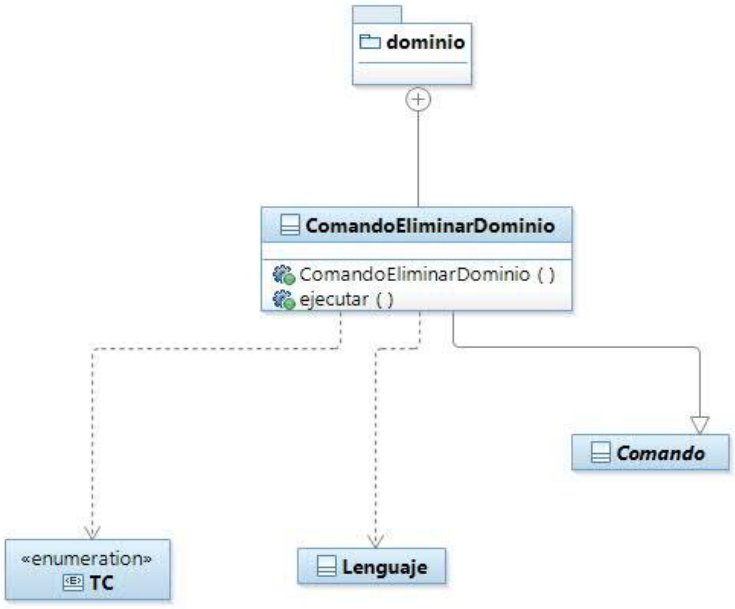

Ilustración 26: Diagrama de clase paquete controlador.comandos.dominio

Entidad

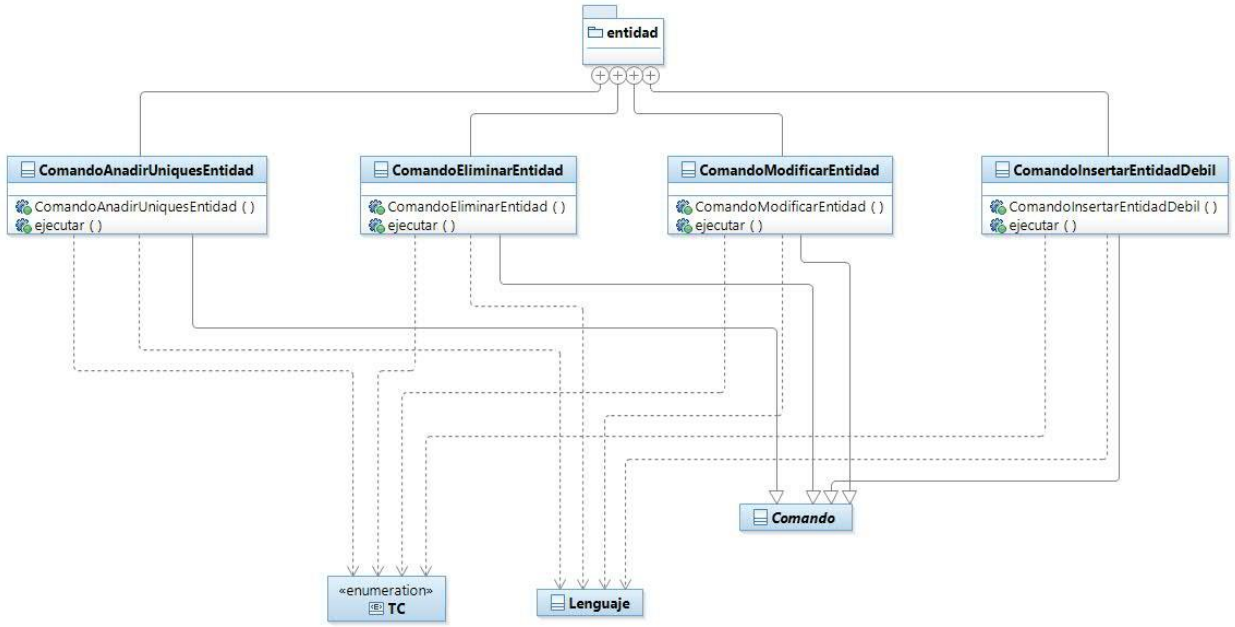

Ilustración 27: Diagrama de clase paquete controlador.comandos.entidad

Relación

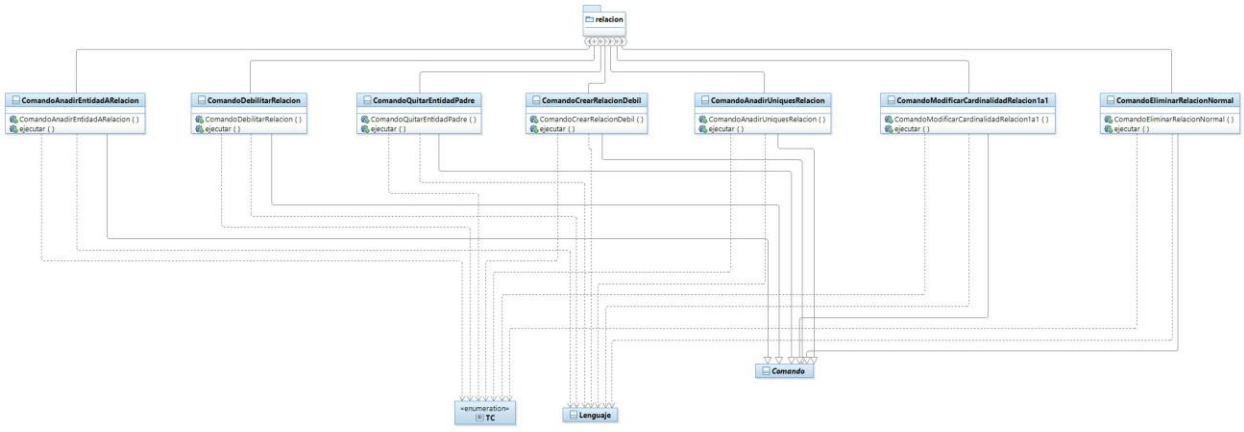

Ilustración 28: Diagrama de clase paquete controlador.comandos.relacion

Vistas

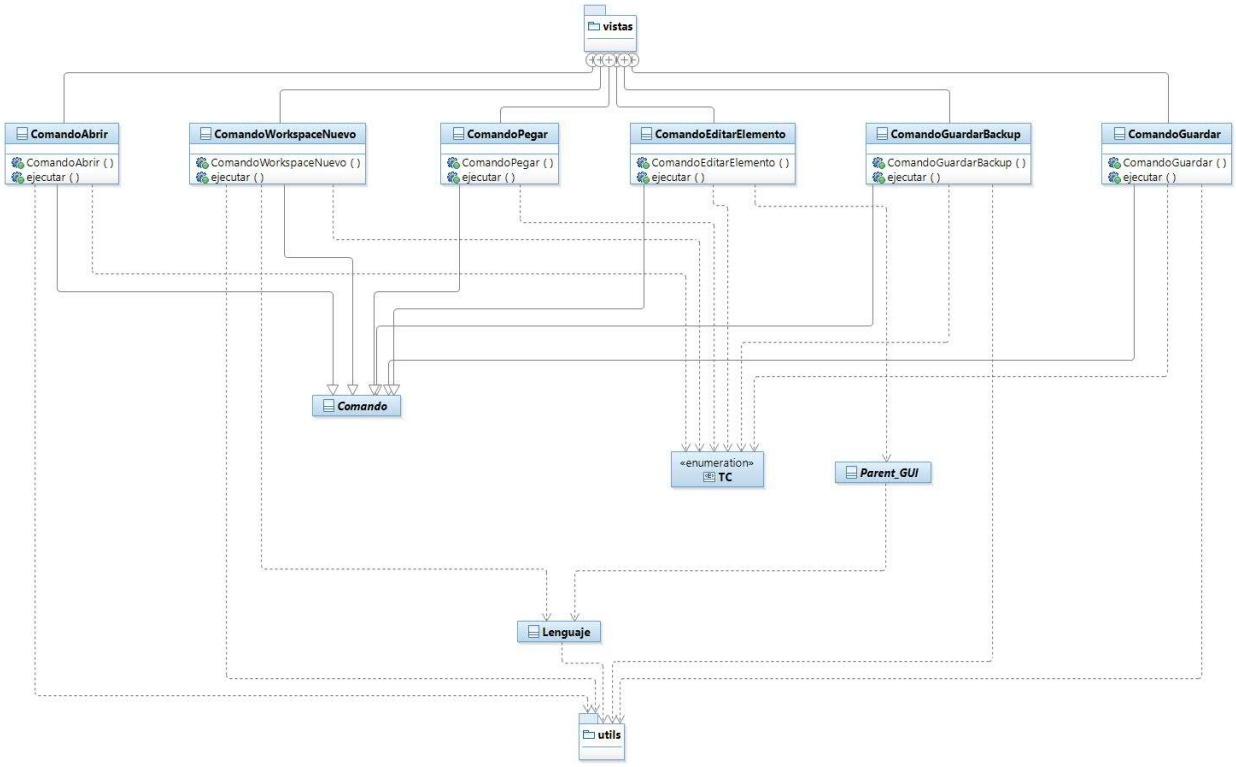

Ilustración 29: Diagrama de clase paquete controlador.comandos.vistas

Modelo

Servicios

Decidimos realizar varios diagramas de clase para este paquete, dado que en uno solo se condensaba demasiada información que hacía que el diagrama fuese menos útil.

En el diagrama que exponemos en la ilustración 6 mostramos la estructura del paquete a excepción de la clase GeneradorEsquema y otras clases relativas a la gestión del esquema.

En el diagrama que exponemos en la ilustración 8 mostramos todo lo relativo a la clase GeneradorEsquema y las clases relativas a la gestión del esquema, y su relación con otras clases.

En el diagrama que exponemos en la ilustración 30 mostramos un diagrama de clase general que pretende mostrar la relación de los servicios con otras clases que no pertenecen al paquete.

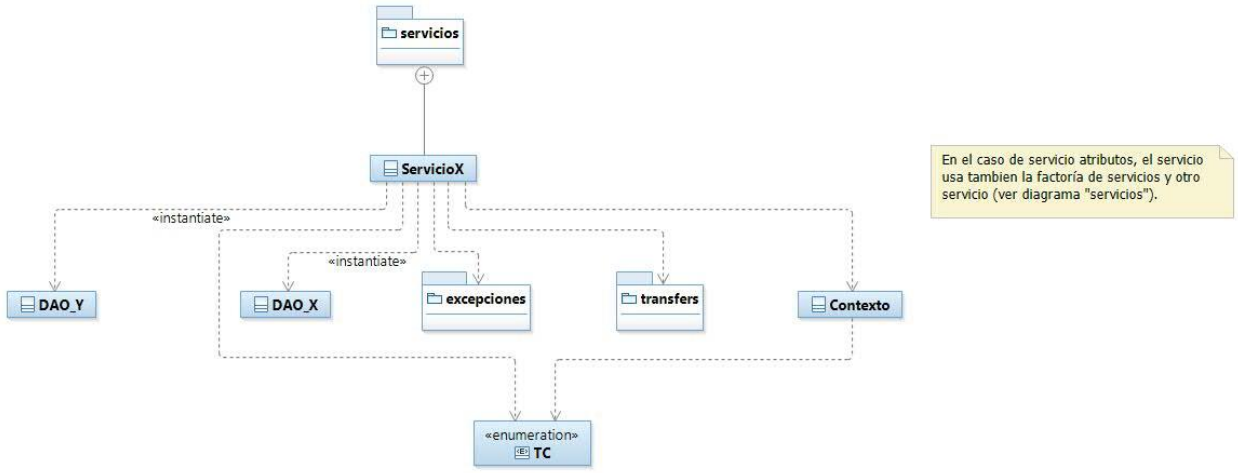

Ilustración 30: Diagrama de clase paquete modelo.servicios (parcial)

ConectorDBMS

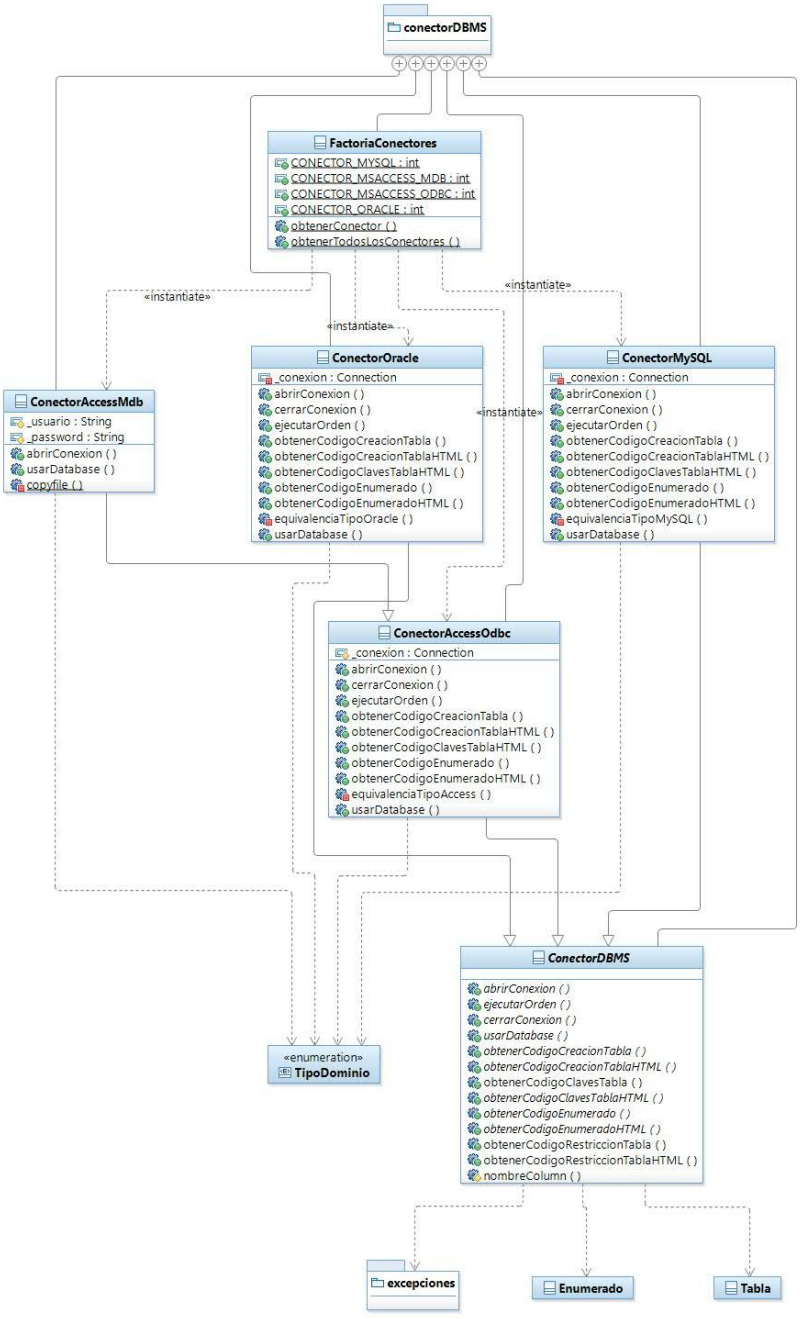

Ilustración 31: Diagrama de clase del paquete modelo.conectorDBMS

Transfers

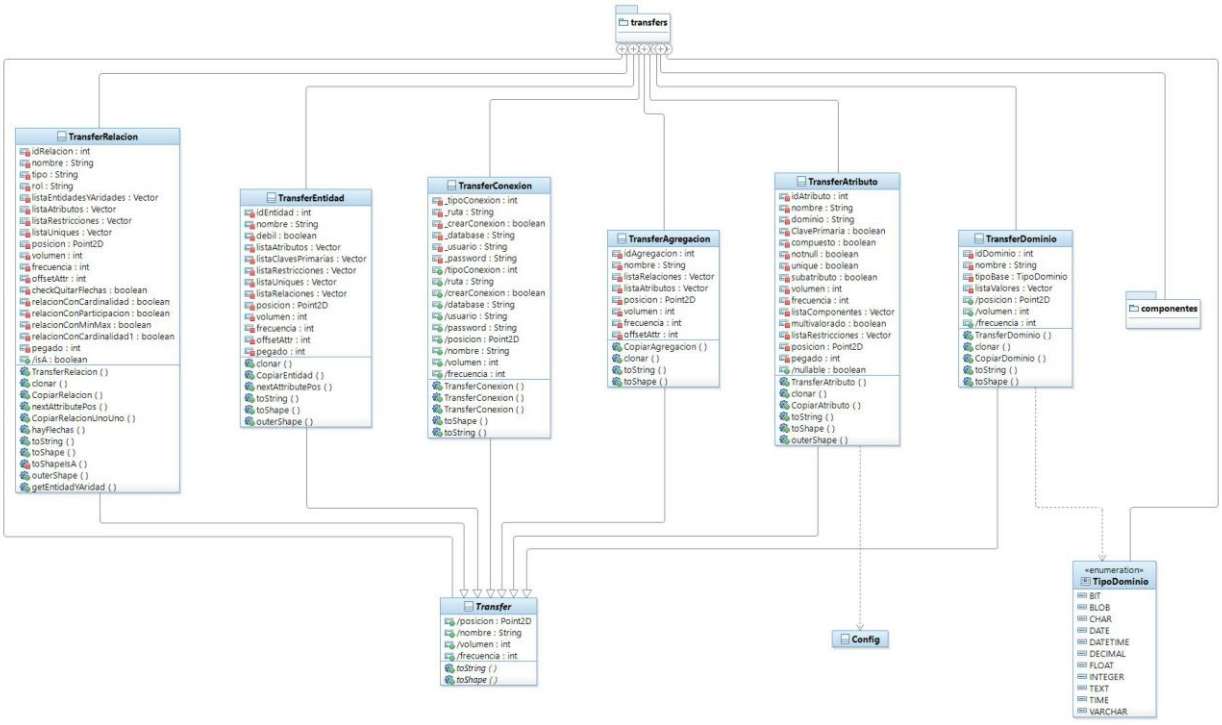

Ilustración 32: Diagrama de clase paquete modelo.transfers

Persistencia

Ver ilustración 9.

Excepciones

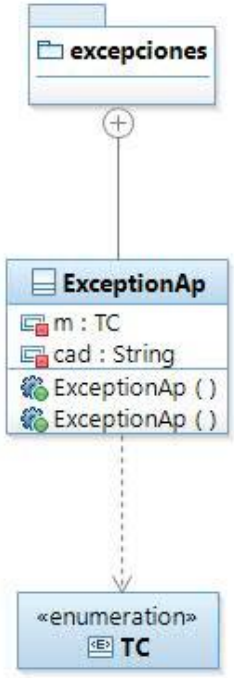

Ilustración 33: Diagrama de clase paquete excepciones

Misc

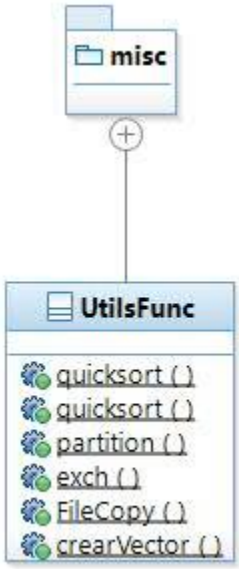

Ilustración 34: Diagrama de clase paquete misc

Apéndice B - Diagramas UML de secuencia

Exponemos a continuación los diagramas de secuencia que realizamos. Nos gustaría hacer notar lo mismo que comentamos al inicio del apéndice A: si estos diagramas son visualizados con la herramienta IBM RSAD, especialmente los que tienen un gran tamaño, se podrá ver con más detalle de forma más fácil.

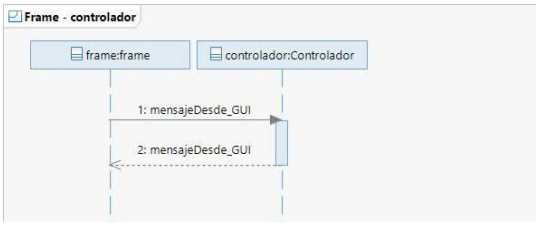

Ilustración 35: Diagrama de secuencia frame editar not-null atributo

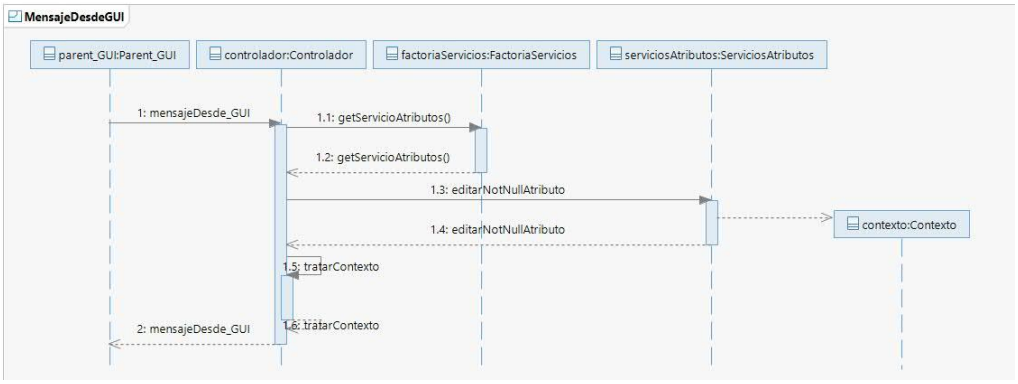

Ilustración 36: Diagrama de secuencia controlador editar not-null atributo

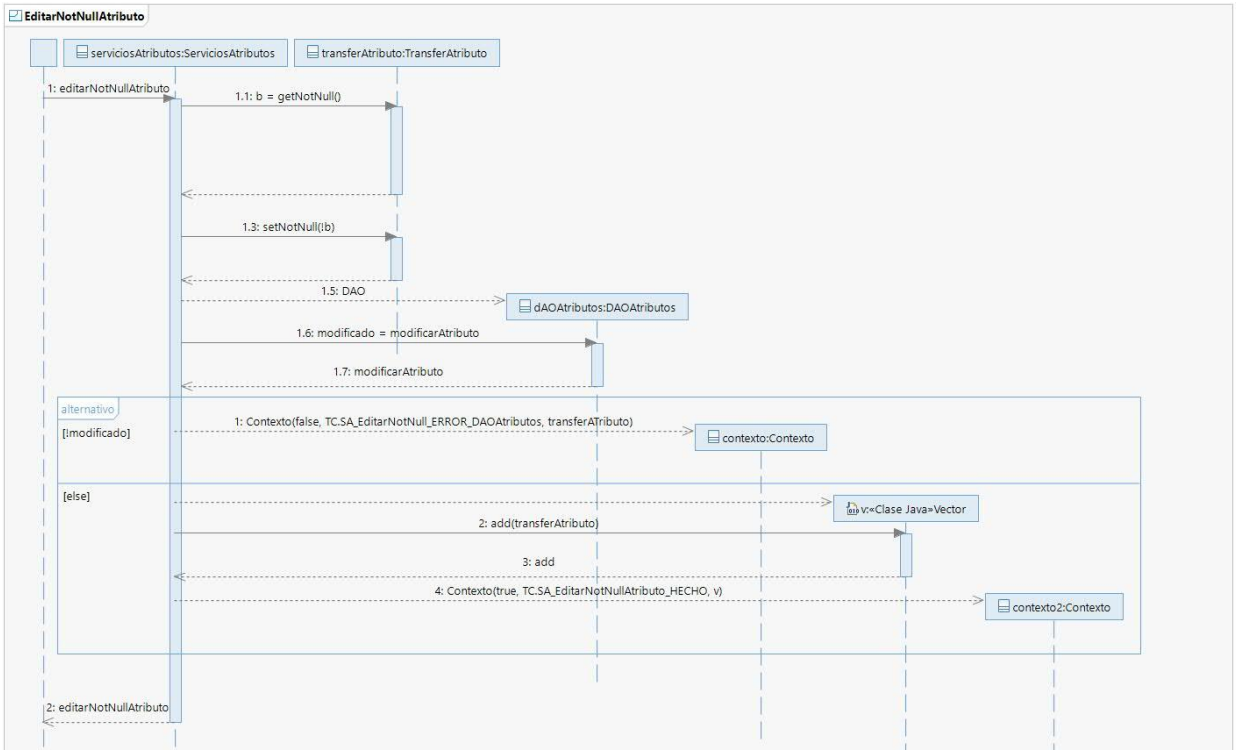

Ilustración 37: Diagrama de secuencia negocio editar not-null atributo

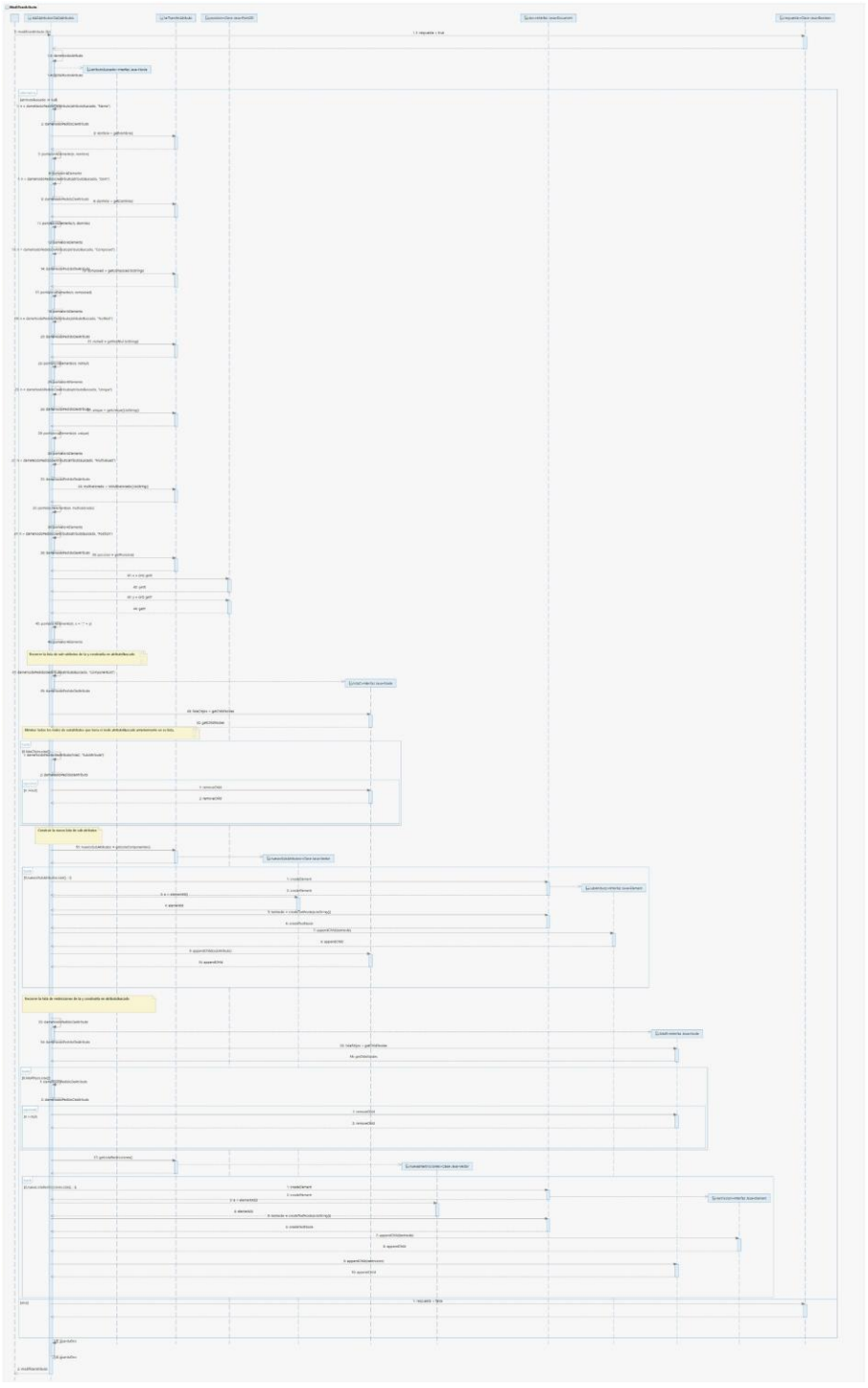

Ilustración 38: Diagrama de secuencia persistencia modificar atributo

Para ver el diagrama de secuencia de tratarContexto, ver ilustración 7.

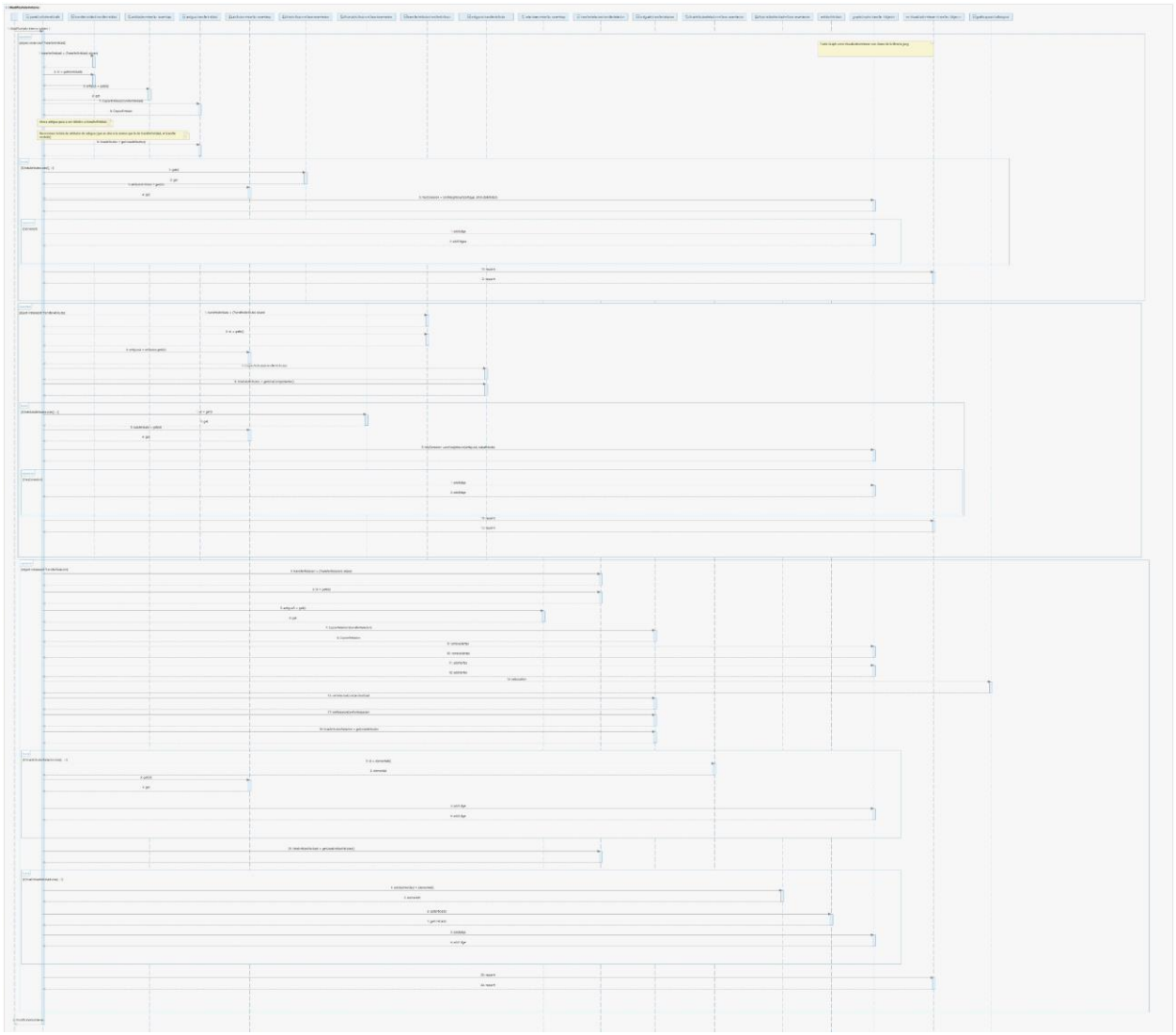

Ilustración 39: Diagrama de secuencia ModificarValorInterno (clase PanelGrafo, paquete vista.diagrama)

Para ver el diagrama de secuencia de ejecutar comando, ver ilustración 5.

Ilustración 40: Diagrama de secuencia comunicación GUIPrincipal – Controlador

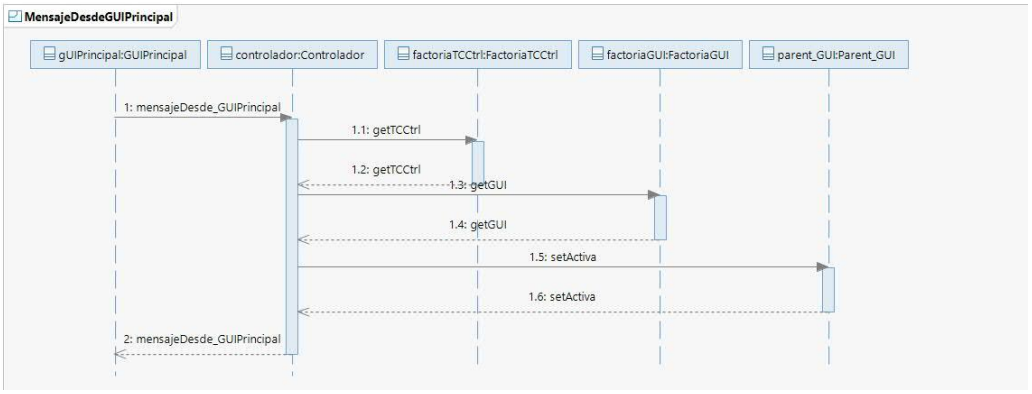

Ilustración 41: Diagrama de secuencia activación de frame por controlador