Grado en Ingeniería Informática

Trabajo Final de Grado

Plugin de monitorización de datos de una base de datos en el sistema de monitorización

Autor:
Justo Molina Centelles

Tutor académico:
Oscar Belmonte Fernández

Fecha de lectura: 19 de julio de 2019
Curso académico 2018/2019

Resumen

Plugin de monitorización de datos de una base de datos en un sistema de monitorización, es el título que se le da al presente documento, el cual registra y describe el Trabajo de Final de Grado, realizado en el *Instituto de Tecnología Cerámica*, en torno a un proyecto que consiste, tal y como se refleja en el título, en desarrollar un plugin que permita recopilar datos de diferentes sistemas gestores de bases de datos en la plataforma de *IIoT* (Internet Industrial para las Cosas) y *Big Data*, denominada *Nexus Integra*.

La aplicación se desarrolla como un servicio Ciente-Servidor, en la que los usuarios introducen las consultas que obtienen los datos a monitorizar en el cliente, en lenguaje SQL. A su vez, el servidor ejecuta las consultas introducidas cíclicamente y almacena los resultados obtenidos en la plataforma anteriormente mencionada. El proyecto ha sido desarrollado en entorno .NET Framework 4.5, siendo implementado en lenguaje C# y almacenando sus datos en un sistema gestor de bases de datos tipo *MySQL*.

Palabras clave

monitorización, historización, base de datos, .NET, C#, Big Data

Keywords

monitoring, historicization, database, .NET, C#, Big Data

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

Para abordar la memoria que registra el presente proyecto, se comenzará por describir el contexto en el que se ha generado, que en este caso es el *Instituto de Tecnología Cerámica* (*ITC*) y, a su vez, se definirán los objetivos y motivaciones que lo han impulsado.

El proyecto se ha realizado en el Área de Procesos Industriales del *Instituto de Tecnología Cerámica* (*ITC*). El *ITC* es un centro de investigación cerámico, nacido de un convenio entre la *Universitat Jaume I* y la *Asociación de Investigación de las Industrias Cerámicas* (*AICE*). Su fundación se llevó a cabo con el objetivo de dar respuesta a las necesidades y requisitos de la industria del ámbito cerámico [3].

El área de Procesos Industriales donde ha sido realizado el proyecto, es un área multidisciplinar formada por Ingenieros Químicos, Técnicos en Instrumentación e Ingenieros Informáticos, dedicados a desarrollar proyectos de automatización, control y optimización de procesos industriales en general y, de forma más concreta, especializándose en el ámbito cerámico.

Con la aparición de la conocida como Industria 4.0 [2], en el *ITC* se están llevando a cabo diversos proyectos que promueven la transformación de la industria cerámica, integrando el conocido como Internet de las cosas (IoT) al nivel industrial, llamado *IIoT*. El proyecto que abarca este Trabajo de Final de Grado se encuentra dentro de uno de estos proyectos, denominado *CEBRA+*, centrado en el desarrollo de acciones de transformación hacia la Industria 4.0 en la cadena de valor del sector de fabricación de baldosas cerámicas, el cual es financiado por el *Instituto Valenciano de Competitividad Empresarial* (*IVACE*).

Este proyecto, enmarcado dentro del proyecto *CEBRA+*, se centra en el desarrollo de una aplicación de software que permita la monitorización e historización de valores almacenados en diferentes bases de datos, en la plataforma de integración de información industrial, denominada *Nexus Integra*. El proyecto *CEBRA+*, mencionado anteriormente, hace uso de la planta de fabricación de baldosas cerámicas de la empresa *Colorker*, ubicada en Xilxes, como piloto demostrativo de Industria Cerámica 4.0. En esta misma fábrica ya se ha desplegado la plataforma

Nexus Integra; en ella se están monitorizando los datos de parte de los equipos de producción, los cuales se comunican con el protocolo *Modbus*.

Ya que la plataforma *Nexus Integra* no está implementada para tener como origen de datos una base de datos *Structured Query Language* (SQL), se necesita un plugin para la plataforma, que permita la recolección y el almacenado de datos, provenientes de bases de datos con lenguaje SQL. Esto es debido a que numerosos equipos y sensores de la planta no utilizan el protocolo *Modbus*. No obstante, sí que almacenan su información en diversas bases de datos, por lo que es necesario desarrollar un plugin que pueda monitorizar estos equipos de producción y sensores dispuestos en las líneas de producción.

Hasta aquí un resumen de los aspectos fundamentales que contextualizan el proyecto llevado a cabo y su función en la empresa. A continuación se destacarán los objetivos en los que se centra este trabajo.

1.2. Objetivos del proyecto

Con este proyecto se pretende, como objetivo principal, dar una solución a la necesidad de poder monitorizar e historizar los datos almacenados en: bases de datos, equipos industriales u otras fuentes (por ejemplo, el coste del material), en la plataforma *Nexus Integra*. Para llevarlo a cabo, se ha desarrollado la presente aplicación, en la que se podrá especificar el origen de los datos, la consulta que se va a realizar y el tag (o nombre de las variables de los datos) que se le va a asignar en la plataforma *Nexus Integra*.

La aplicación deberá cumplir los siguientes objetivos:

- Una interfaz gráfica o formulario para que los usuarios administren la aplicación de forma sencilla.
- Que la misma interfaz gráfica almacene los datos, que indican cómo obtener los datos a monitorizar y almacenar.
- La curva de aprendizaje de la aplicación deberá ser sencilla para usuarios con conocimientos del lenguaje de bases de datos y el entorno de *Microsoft Windows*.
- Tener un servicio que monitorizará e historizará los datos especificados en la interfaz gráfica en tiempo y forma deseados.
- Que el servicio de monitorización e historización sea ejecutado en segundo plano en un servidor con disponibilidad plena, es decir, que se mantenga en continuo funcionamiento.

Una vez finalizado el proyecto, la monitorización e historización de los datos permitirá, entre otras cosas:

- Visualizar a tiempo real el funcionamiento, rendimiento, eficiencia y consumo de las líneas de producción.

- Realizar informes de costes de producción.
- Encontrar correlaciones entre datos de la producción que ayuden a la optimización del proceso.

Por último, a nivel personal, con este Proyecto de Final de Grado se pretende demostrar las habilidades adquiridas durante el grado de Ingeniería Informática.

1.3. Estado del Arte

En la actualidad existen numerosas plataformas que permiten la gestión de *Industrial Internet of Things* (*IIoT*) y *Big Data*, obteniendo los datos de múltiples orígenes. No obstante, la plataforma con la que se ha trabajado (por cuestiones externas a este proyecto), al comienzo del proyecto, no integra como origen de datos bases de datos SQL y planean desarrollar esta funcionalidad, a largo plazo, para obtener datos de este origen.

1.4. Estructura de la memoria

Tras esta introducción, el cuerpo de esta memoria se organiza, tal y como viene reflejado en el índice de contenidos, con un apartado en primer lugar, en el que se describe el proyecto, se especifican la funcionalidad y el alcance, se valoran los riesgos y las restricciones, y se mencionan los recursos necesarios para llevarlo a cabo. A continuación, le sigue un apartado en el que se detalla la metodología y la planificación que se han utilizado, incluyendo también una estimación de los recursos y costes, y el seguimiento del proyecto. Seguidamente, se ha elaborado un apartado de análisis del sistema y de la interfaz que, junto con el apartado continuo de implementación y pruebas, constituyen el grueso de esta memoria. Finalmente, se han incluido, a modo de reflexión, unas conclusiones junto a una reflexión personal y, tras ello, las referencias bibliográficas. Paralelamente, el apartado de los anexos, colocado al final de la memoria, incluye una documentación detallada del Proyecto Biblioteca, y algunas figuras que se han considerado relevantes para completar la explicación del proyecto.

Capítulo 2

Descripción del proyecto

2.1. Descripción General

A continuación se describe el proyecto en sus principales rasgos y características.

La finalidad de este proyecto es desarrollar un servicio que permita monitorizar e historizar datos obtenidos de diferentes Sistemas Gestores de Bases de Datos (SGBD), con una programación previa. La programación de los datos a obtener, se realizará desde una aplicación, que en un principio se desarrollará para escritorio, pudiendo en un futuro, transformarse a una aplicación web. Para programar los datos a monitorizar, será necesario escribir la consulta en lenguaje SQL e indicar el período[1] y el lapso[2] de tiempo en el que se ejecutará.

El proyecto se ha realizado en C#[7] con el framework .NET[5], tal y como la empresa *Colorker* lo demanda. Esto es debido a que en la empresa prefieren utilizar servidores con el sistema operativo *Microsoft Windows* y, a su vez, todo el resto de programas de gestión lo han diseñado con dicho entorno, .NET. Además, uno de los requisitos que se especificó cuando se eligió la plataforma *Nexus Integra* para realizar el proyecto, fue que la plataforma empleada para la monitorización e historización de los datos, estuviera construida en entorno .NET.

Los SGBD a los que se desea conectar son de diferentes tipos: MySQL[21] y Microsoft SQL Server[8], además, se planea en el futuro tener una Oracle SQL[20]. Por lo que la aplicación debe poder operar con todos estos SGBD.

Con el fin de abordar este proyecto, se ha decidido crear una aplicación Cliente-Servidor, dado que es un servicio que debe proporcionarse con la mayor disponibilidad posible, además de ser utilizado por diferentes clientes, desde diferentes estaciones de trabajo. A continuación se detallan algunos rasgos de la parte cliente y de la parte servidor del proyecto.

[1] El intervalo de tiempo entre cada ejecución de monitorización e historización de los datos.
[2] Instantes de inicio y fin de ejecución de monitorización e historización de los datos.

2.1.1. Cliente

Respecto a la interfaz gráfica del cliente, se planea realizarla inicialmente con Windows Forms, siendo también posible que, más adelante, se cambie a una interfaz web.

Así mismo, el cliente será una aplicación ejecutable compilada como .exe que se podrá ejecutar desde cualquier equipo que tenga acceso al servidor de almacenaje de consultas y a los servidores a los que se realizan las consultas.

2.1.2. Servidor

El servidor estará instalado como servicio en uno de los servidores de la empresa. Se decidió, por tanto, que el servicio se instalaría en el servidor donde la empresa tiene instalada la plataforma *Nexus Integra* y el SGBD MySQL, ya que es el que se adquirió y se está utilizando para el proyecto *CEBRA+*.

La parte servidor de esta aplicación almacenará sus datos en un SGBD MySQL que es el que se encuentra en el mismo servidor donde se desplegará el servicio.

2.2. Funcionalidad

En este apartado se especifican las funcionalidades que debe cumplir cada parte de la aplicación.

- **Cliente**
  - La parte cliente de la aplicación permitirá: por un lado, crear, modificar y eliminar consultas programadas para recolectar datos; por otro lado, crear y eliminar las variables en las que se monitorizan e historizan los datos.
- **Servidor**
  - La parte servidor de la aplicación debe monitorizar las consultas programadas y almacenar sus datos en la plataforma *Nexus Integra*.

2.3. Alcance

Se puede abordar el alcance de este proyecto desde diversas perspectivas. En primer lugar, hay que tener en cuenta que para el *ITC*, que aborda el proyecto *CEBRA+*, se pretende conseguir recopilar datos para analizarlos, con la finalidad de mejorar la productividad en las líneas de producción en la industria cerámica. La información obtenida de la recopilación y al análisis debe ser actualizada, íntegra, coherente, relevante, exacta, accesible y de confianza.

Por otro lado, el alcance funcional, será poseer una interfaz amigable que permitirá a los usuarios recopilar datos de las BD en la plataforma *Nexus Integra*. Además, deberá ejecutarse como un servicio con la máxima disponibilidad posible.

En cuanto al alcance organizativo, la aplicación será utilizada por trabajadores de distintas áreas, así como jefes de sección o algunos miembros del departamento comercial; será administrado por los administradores del departamento informático de la empresa y mantenido por los desarrolladores del *ITC*.

Respecto al alcance informático, se trata de una aplicación muy específica, únicamente utilizada para este proyecto por el momento; no se ha encontrado otra aplicación que cubra la necesidad requerida.

Por último, cabría destacar que, pese a que este proyecto vaya orientado a la industria cerámica, ello no excluye que pueda ser empleado en otras industrias o en distintos sectores.

2.4. Restricciones y Riesgos

Como restricción técnica, puede destacarse que los recursos utilizados por la aplicación no deben de ser excesivos, para así no bloquear el resto de servicios que se ejecutan en el servidor de Colorker, como el SGDB *MySQL* o la plataforma *Nexus Integra*.

Por otro lado, como restricción podrá tenerse en cuenta que, por un lado, el proyecto se ha desarrollado principalmente de manera individual, bajo la supervisión del único ingeniero informático del Área de Procesos Industriales del *ITC*, cuya especialización no es el lenguaje C# ni la gestión de BD. Asimismo, tampoco se ha contado con la supervisión de un tutor universitario durante el periodo de desarrollo de la aplicación.

En cuanto a los riesgos que harían que este proyecto no se completase serían: que el software requiera demasiados recursos o que no recopile los datos con fiabilidad; pues ello implicaría que no se podrían obtener los datos necesarios para el proyecto *CEBRA+*. Por otro lado, también sería un riesgo que que la plataforma *Nexus Integra* incorporara en su plataforma la obtención de datos de orígenes SQL, de modo que el proyecto ya no sería necesario.

2.5. Tecnologías y herramientas utilizadas

Durante la realización de este proyecto se han utilizado las siguientes tecnologías y herramientas para su desarrollo:

- .NET Framework 4.5: Utilizado para obtener soluciones pre-codificadas para requerimientos comunes, como la creación de un servicio.
- Lenguaje C#: Utilizado en la implementación del proyecto.

- Lenguaje SQL: Se ha utilizado para implementar partes del proyecto y probar su funcionamiento.
- Softwares utilizados:
  - *MySQL*: SGBD ha sido útil para el almacenaje de la información de la aplicación y ejecución de pruebas.
  - *Microsoft SQL Server*: SGBD empleado en la ejecución de pruebas.
  - *Oracle SQL Developer*: SGBD usado para la ejecución de pruebas.
  - *Microsoft Visual Studio 2017 Professional*: Utilizado para implementar la aplicación [6].
  - *Visual Paradigm 15.2 CE*: Utilizado para generar los diagramas de: casos de uso, clases y entidad relación [3].
  - *Team Foundation Server*: Se ha empleado con el fin de realizar la gestión de versiones y parte de su planificación futura.
  - *Microsoft Project*: Utilizado en la planificación del proyecto y generar los diagramas de Gantt.

[3]Las dimensiones de algunas de las imágenes que se incluyen en los anexos, tienen un tamaño demasiado reducido de letra, por lo que no las permite leer correctamente; esto es debido a que la aplicación donde se diseñan, las muestra con ese formato.

Capítulo 3

Planificación del proyecto

3.1. Metodología

A continuación, se expondrá la metodología usada para desarrollar el proyecto. Cabe destacar, que la empresa donde se desarrolla el proyecto y al comienzo del mismo, no posee ninguna política metodológica ni gestor de versiones. Por tanto, la metodología a usar queda en manos del desarrollador.

Debido a la urgencia de una primera versión funcional del proyecto y, tras valorar diferentes metodologías para aplicar en el mismo se determina que la opción más adecuada para la planificación, es una metodología predictiva, haciendo uso de diagramas de Gantt. Esto se debe a que no hay un gran número de tareas que se deben llevar a cabo para hacer una versión funcional y, al mismo tiempo, permite programar fácilmente nuevas tareas de mejoras y optimizaciones.

3.2. Planificación

Para realizar la planificación se ha hecho una lista que compile las tareas primordiales para llevar el proyecto a un punto funcional. Se tiene en cuenta que tras la realización de estas tareas, será necesario realizar más para la mejora del proyecto.

En la planificación se han estimado una duración del coste de cada tarea. Las fechas de comienzo y fin de cada tarea, se han calculado con la duración estimada y teniendo en cuenta que el proyecto se inició el 18/06/2018. Paralelamente, se tiene en cuenta que el horario laboral en el que se desarrolla es de 5 horas de lunes a viernes, a excepción de los meses de julio y agosto que el horario se reduce a 4 horas y 22 minutos y hay un periodo vacacional de tres semanas en agosto. Al finalizar el proyecto, también se analizará la desviación de la fecha de finalización de cada tarea.

Teniendo en cuenta que se utiliza una metodología predictiva, en la siguiente tabla, 3.1, se detallan las tareas a realizar, las fases que las engloban, los hitos, las dependencias entre ellas,

sus estimaciones de duración y las fechas de realización. Cabe aclarar que las filas marcadas en negrita hacen referencia a las diferentes fases del proyecto, mientras que las tareas que estas engloban, aparecen a continuación tabuladas, dentro de la segunda columna (Nombre de Tarea). Respecto a los hitos, estos aparecen indicados con su propio nombre, dentro de la cuarta columna (Trabajo Previsto). En la tercera columna (Predeceroras) aparece el identificador de la tarea (indicado en la primera columna) que se debe ejecutar previamente, separado por el caracter ";".

Id.	**Nombre de Tarea**	***Predecesoras***	***Trabajo previsto***	***Comienzo previsto***	***Fin previsto***
1	**Diseño y planificación del proyecto**		**90 horas**	**18/06/18**	**19/09/18**
2	Diseño del proyecto DBNexusBiblioteca		50 horas	18/06/18	02/07/18
3	Diseño de la Interfaz	2	10 horas	02/07/18	05/07/18
4	Diseño de la Base de Datos	2	10 horas	03/07/18	06/07/18
5	Diseño del proyecto DBNexusCliente	3	10 horas	17/09/18	19/09/18
6	Diseño del proyecto DBNexusServicio	4	10 horas	04/09/18	06/09/18
7	**Implementación del proyecto DBNexusBiblioteca**		**55 horas**	**06/07/18**	**31/07/18**
8	Implementación de los Modelos de DBNexusBiblioteca	2;3	15 horas	06/07/18	12/07/18
9	Implementación de los Conectores con las DB de DBNexusBiblioteca	2;8	15 horas	16/07/18	19/07/18
10	Implementación del Conector con la DB de almacenamiento de datos del proyecto	9;4	25 horas	24/07/18	31/07/18
11	Completada la implementación de DBNexusBibliotea	7	Hito		31/07/18
12	Implementación de la Base de Datos	4	15 horas	03/08/18	29/08/18
13	**Implementación del proyecto DBNexusCliente**		**35 horas**	**29/08/18**	**26/09/18**
14	Prototipado de la interfaz	3	10 horas	29/08/18	31/08/18
15	Implementación de DBNexusCliente	7;5	25 horas	19/09/18	26/09/18
16	Completada la implementación de DBNexusCliente	13	Hito		26/09/18
17	**Implementación del proyecto DBNexusServicio**		**25 horas**	**06/09/18**	**13/09/18**
18	Implementación de DBNexusServicio	7;6	25 horas	06/09/18	13/09/18
19	Completada la implementación de DBNexusCliente	17	Hito		13/09/18
20	**Pruebas y documentación del proyecto DBNexusBiblioteca**		**90 horas**	**12/07/18**	**12/10/18**
21	Implementación de Pruebas Unitarias de los Modelos de DBNexusBiblioteca	8	10 horas	12/07/18	16/07/18
22	Implementación de Pruebas Unitarias de los Conectores de DBNexusBiblioteca	9	10 horas	19/07/18	24/07/18
23	Implementación de Pruebas Unitarias del Conector con la DB de almacenamiento de datos del proyecto	10	10 horas	31/07/18	03/08/18
24	Realización de pruebas manuales del Conector con la DB de almacenamiento de datos del proyecto	10;23	10 horas	31/08/18	04/09/18
25	Documentación del proyecto DBNexusBibliteca	7	50 horas	28/09/18	12/10/18
26	**Pruebas y documentación del proyecto DBNeusCliente**		**10 horas**	**26/09/18**	**28/09/18**
27	Prueba manual del proyecto DBNeusCliente	24;13	10 horas	26/09/18	28/09/18
28	**Pruebas y documentación del proyecto DBNeusServicio**		**10 horas**	**13/09/18**	**17/09/18**
29	Prueba manual del proyecto DBNeusServicio	17;24	10 horas	13/09/18	17/09/18

Cuadro 3.1: Tabla con las tareas planificadas y sus detalles

Una vez planificadas todas las tareas y sus costes de ejecución, se puede ver en las figuras 3.1 y 3.2 el diagrama de Gantt correspondiente a la planificación, donde aparecen todas las tareas a realizar. En la figura B.1 de los anexos, se muestra el diagrama de planificación en una única figura.

Como se puede observar en el diagrama de Gantt, se ha planificado cada proyecto secuencialmente, es decir, se inicia diseñando, implementando y realizando las pruebas del proyecto DBNexusBiblioteca, se continua con la base de datos antes de realizar el DBNexusCliente y se finaliza con el DBNexusServicio. Respecto los proyectos DBNexusCliente y DBNexusServicio, al ser independientes, es irrelevante cuál de ellos se desarrolle primero pero, ya que el cliente ejecuta tareas más sencillas sobre la base de datos, se ha decidido implementar este primero y cerciorarse que los conectores con la base de datos funcionan como se espera. Además, a pesar que la documentación del código esté programada como última tarea, durante la ejecución del proyecto se irán documentando las partes más confusas o criticas del desarrollo y, durante esa tarea se ampliará y corregirá esta documentación.

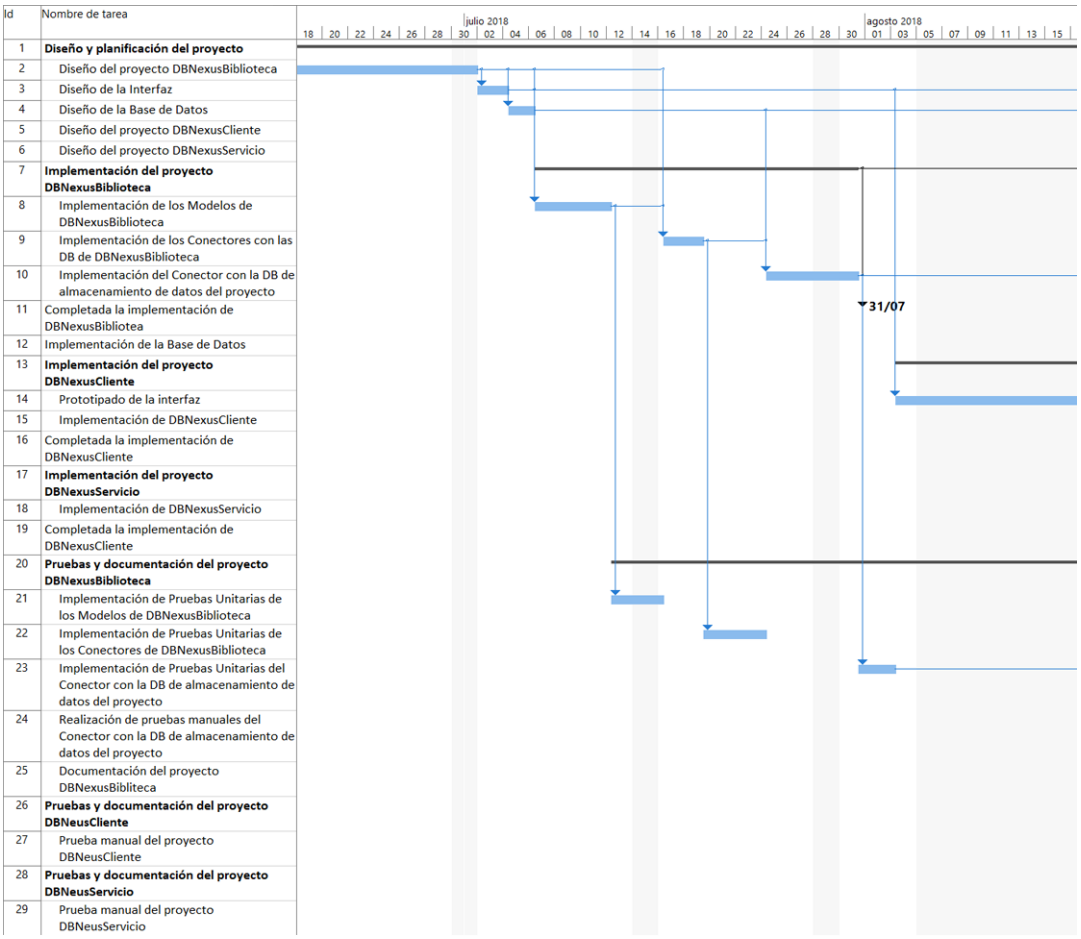

Figura 3.1: Diagrama de Gantt de la planificación del Proyecto del 18/06/2018 al 15/08/18

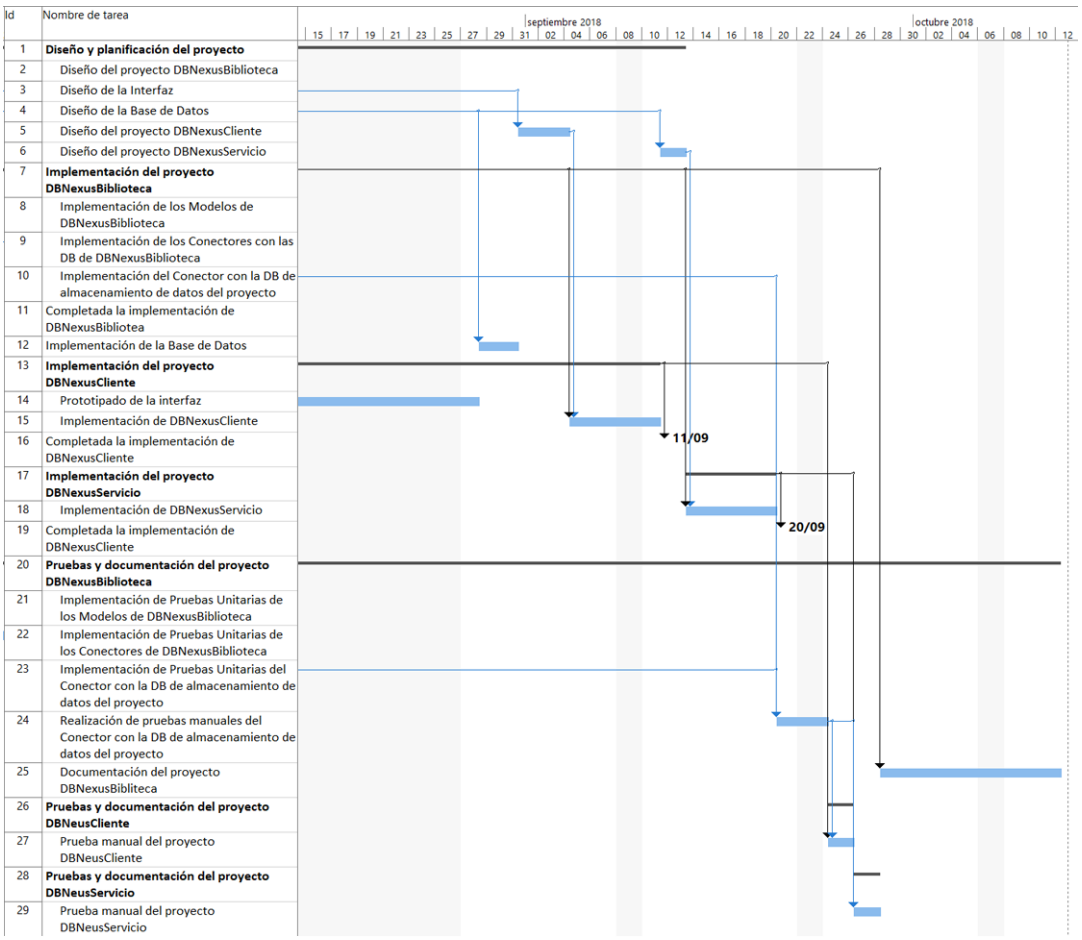

Figura 3.2: Diagrama de Gantt de la planificación del Proyecto del 15/08/18 al 12/10/2018

Con la planificación propuesta, se han estimado unas 330 horas de trabajo, por lo que, teniendo en cuenta que el proyecto inició el 18/06/2018 y el periodo laboral anteriormente mencionado, se estima tener una versión funcional del proyecto para el 12/10/2018. Cabe destacar, que esta versión funcional no cumplirá todos los requisitos de la aplicación, no obstante permitirá iniciar la recolección de datos.

3.3. Estimación de recursos y costes del proyecto

En esta sección se hará una estimación en la que se especificarán qué recursos y herramientas serán necesarios para realizar el proyecto. Para conocer los costes temporales, nos basaremos en la planificación, desarrollada en el apartado anterior, para su estimación.

Como en todo proyecto profesional, hay que prever que pueden darse ciertos retrasos, debido a causas internas (error en el diseño en la planificación) o externas (por ejemplo, una baja laboral) del proyecto. Por ello, a las horas totales estimadas del proyecto, se le añadirá un incremento del 10 %. Por otro lado, hay que tener en cuenta que la planificación del apartado anterior no tiene en cuenta las mejoras que se aplicarán una vez realizada la versión funcional del proyecto. Para ello, se estima que serán necesarias otras 300 horas aproximadamente. De este modo, el coste total de las horas del proyecto es el resultado de las 330 planificadas inicialmente, sumadas a las 300 estimadas para las mejoras, más el incremento del 5 %; todo ello suma exactamente 661,5 horas, las cuales se redondean a un total de 660.

Teniendo este cálculo en cuenta, si se estima que el coste de un desarrollador es de aproximadamente de 20€/hora, el coste económico del proyecto sumaría 13.200 €.

3.4. Seguimiento del proyecto

Durante el desarrollo de este proyecto, la ejecución de las tareas ha variado respecto a la planificación, pues se han retrasado algunas tares y otras se han priorizado. En este apartado, se expondrán las principales desviaciones respecto a la planificación y cómo ha afectado esto al desarrollo del proyecto.

Para estimar de forma apropiada el transcurso de las tareas, se han compilado el coste de trabajo y las fechas de comienzo y fin de las tareas, junto con su variación respecto a lo planificado. Estos datos se encuentran en la tabla 3.2. Cabe destacar que las distintas fases del proyecto se marcan en negrita; a su vez, las tareas que estas engloban, se indican a continuación, tabuladas, dentro de la segunda columna (Nombre de Tarea). Los hitos aparecen indicados con su propio nombre, en la cuarta columna (Trabajo). En la tercera columna (Predeceroras) aparece el identificador de la tarea (indicado en la primera columna) que se debe ejecutar previamente, separado por el caracter ";". Las tres columnas de variación (Variación de trabajo, Variación de comienzo y Variación de fin), indican el retraso de los valores de esta tabla, respecto a la tabla 3.1 de planificación.

Al llevar, durante la realización del proyecto, un seguimiento de la realización de las tareas, es posible conocer el retraso respecto a la planificación en el día a día. En el diagrama obtenido de las planificaciones y el desarrollo real de las tareas, es posible hacerse una idea de en cuales la planificación ha fallado realmente. Este diagrama se encuentra en las figuras 3.3, 3.4, 3.5 y 3.6. En la figura B.2 de los anexos, se muestra el diagrama en una única figura.

Es necesario tener en cuenta que, en este diagrama de Gantt, las fechas en las que estaban planificadas la tareas se marcan con una linea gris en la misma fila que la tarea, mientras que,

Id.	Nombre de Tarea	Predecesoras	Trabajo	Variación de trabajo	Comienzo	Variación de comienzo	Fin	Variación de fin
1	**Diseño y planificación del proyecto**		**92 horas**	**2 horas**	**18/06/18**	**0 días**	**20/09/18**	**0,5 días**
2	Diseño del proyecto DBNexusBiblioteca		48 horas	-2 horas	18/06/18	0 días	29/06/18	-0,25 días
3	Diseño de la Interfaz	2	8 horas	-2 horas	29/06/18	-0,25 días	03/07/18	-1,29 días
4	Diseño de la Base de Datos	2	15 horas	5 horas	03/07/18	-0,25 días	06/07/18	0,13 días
5	Diseño del proyecto DBNexusCliente	3	6 horas	-4 horas	06/09/18	-4,38 días	07/09/18	-4,88 días
6	Diseño del proyecto DBNexusServicio	4	15 horas	5 horas	17/09/18	5,5 días	20/09/18	6,13 días
7	**Implementación del proyecto DBNexusBiblioteca**		**63 horas**	**8 horas**	**06/07/18**	**0,13 días**	**02/08/18**	**1,13 días**
8	Implementación de los Modelos de DBNexusBiblioteca	2;3	18 horas	3 horas	06/07/18	0,13 días	13/07/18	0,5 días
9	Implementación de los Conectores con las DB de DBNexusBiblioteca	2;8	13 horas	-2 horas	17/07/18	0,75 días	20/07/18	0,5 días
10	Implementación del Conector con la DB de almacenamiento de datos del proyecto	9;4	32 horas	7 horas	24/07/18	0,25 días	02/08/18	1,13 días
11	Completada la implementación de DBNexusBibliotea	7	Hito				02/08/18	1,13 días
12	Implementación de la Base de Datos	4	19 horas	4 horas	31/08/18	3,25 días	06/09/18	3,75 días
13	**Implementación del proyecto DBNexusCliente**		**42 horas**	**7 horas**	**28/08/18**	**-0,38 días**	**17/09/18**	**-4,5 días**
14	Prototipado de la interfaz	3	14 horas	4 horas	28/08/18	-0,38 días	31/08/18	0,13 días
15	Implementación de DBNexusCliente	7;5	28 horas	3 horas	07/09/18	-4,88 días	17/09/18	-4,5 días
16	Completada la implementación de DBNexusCliente	13	Hito				17/09/18	-4,5 días
17	**Implementación del proyecto DBNexusServicio**		**22 horas**	**-3 horas**	**20/09/18**	**6,13 días**	**26/09/18**	**5,75 días**
18	Implementación de DBNexusServicio	7;6	22 horas	-3 horas	20/09/18	6,13 días	26/09/18	5,75 días
19	Completada la implementación de DBNexusCliente	17	Hito				26/09/18	5,75 días
20	**Pruebas y documentación del proyecto DBNexusBiblioteca**		**121 horas**	**31 horas**	**13/07/18**	**0,5 días**	**28/11/18**	**21,08 días**
21	Implementación de Pruebas Unitarias de los Modelos de DBNexusBiblioteca	8	12 horas	2 horas	13/07/18	0,5 días	17/07/18	0,75 días
22	Implementación de Pruebas Unitarias de los Conectores de DBNexusBiblioteca	9	8 horas	-2 horas	20/07/18	0,5 días	24/07/18	0,25 días
23	Implementación de Pruebas Unitarias del Conector con la DB de almacenamiento de datos del proyecto	10	13 horas	3 horas	02/08/18	1,13 días	28/08/18	1,5 días
24	Realización de pruebas manuales del Conector con la DB de almacenamiento de datos del proyecto	10;23	18 horas	8 horas	26/09/18	11,38 días	02/10/18	12,38 días
25	Documentación del proyecto DBNexusBibliteca	7	70 horas	20 horas	09/11/18	18,58 días	28/11/18	21,08 días
26	**Pruebas y documentación del proyecto DBNeusCliente**		**4 horas**	**-6 horas**	**02/10/18**	**2,38 días**	**02/10/18**	**1,63 días**
27	Prueba manual del proyecto DBNeusCliente	24;13	4 horas	-6 horas	02/10/18	2,38 días	02/10/18	1,63 días
28	**Pruebas y documentación del proyecto DBNeusServicio**		**6 horas**	**-4 horas**	**02/10/18**	**8,5 días**	**04/10/18**	**8 días**
29	Prueba manual del proyecto DBNeusServicio	17;24	6 horas	-4 horas	02/10/18	8,5 días	04/10/18	8 días
30	**Mejoras de la aplicación**		**256 horas**	**256 horas**	**04/10/18**		**17/01/19**	
31	Mejora en DBNexusServicio - Añadida hora de última ejecución	17	8 horas	8 horas	04/10/18		05/10/18	
32	Mejora en DBNexusBiblioteca - Consultas para obtener las consultas y variables mejoradas	31	16 horas	16 horas	05/10/18		10/10/18	
33	Mejora en DBNexusBiblioteca - Consulta obtención de consultas por ciclo mejorada	32	16 horas	16 horas	10/10/18		16/10/18	
34	Mejora Diseño de Base de Datos - BDs de consulta, añadidas a la BD	33	24 horas	24 horas	16/10/18		22/10/18	
35	Mejora DBNexusBiblioteca - Adaptación a la nueva DB	34	24 horas	24 horas	22/10/18		29/10/18	
36	Mejora DBNexusBiblioteca - Añadida Funcionalidad de Parametrización		32 horas	32 horas	29/11/18		07/12/18	
37	Mejora DBNexusBiblioteca - Conectores se obtiene de una mochila	36	40 horas	40 horas	07/12/18		19/12/18	
38	Mejora DBNexusBiblioteca - Usuario y contraseña añadia a la BD	37	32 horas	32 horas	19/12/18		27/12/18	
39	Mejora DBNexusBiblioteca - Errores recurrentes detienen la consulta y almacenan el error en la BD	38	16 horas	16 horas	27/12/18		03/01/19	
40	Documentación del proyecto DBNexusBibliteca actualizada	39	48 horas	48 horas	04/01/19		17/01/19	

Cuadro 3.2: Tabla con los costes y los periodos de ejecución de las tareas, contiene también la variación respecto a la planificación

las fechas donde realmente se realizó la tarea están marcadas con una linea azul.

Como se puede observar en el diagrama, las primeras semanas, a pesar de no cumplirse la planificación completamente, el retraso se encuentra dentro de lo esperado; de hecho, hasta la vuelta de las vacaciones de agosto no empieza a hacerse evidente el retraso en la planificación.

La primera medida que se toma para evitar retrasar más la entrega de la primera versión funcional, es implementar sólo las pruebas más criticas, además de generar únicamente las pruebas automáticas y retrasar las manuales hasta después de implementar el cliente y el servicio. Del mismo modo, también se descuida la documentación que, al no urgir para el funcionamiento de la aplicación, se desplaza hasta tener el cliente y el servicio desarrollados.

Pese a los retrasos acumulados, gracias a las decisiones tomadas, el retraso de la primera versión es sólo de seis días hábiles; por ello, se considera que, para ser una planificación diseñada por el programador, se ha ajustado suficientemente a lo planificado. No obstante, hay que tener en cuenta que la realización de la documentación sí que ha sufrido cambios severos, ya que no únicamente ha excedido en 20 horas a lo planificado, sino que, al comenzarse con un retraso de 19 días, no ha sido finalizada hasta 21 días después de lo planificado, cosa que supone un gran error de planificación.

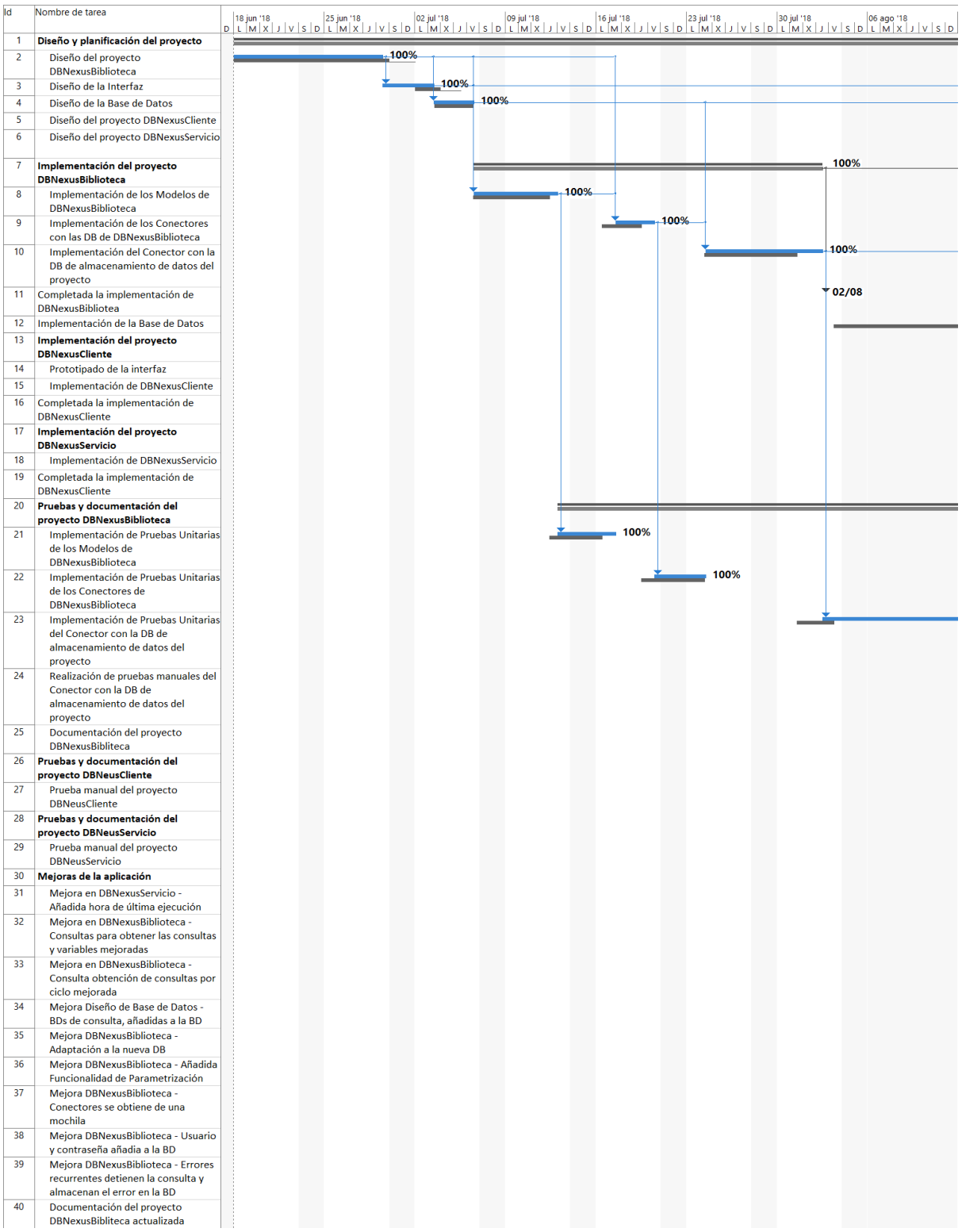

Figura 3.3: Diagrama de Gantt del transcurso del Proyecto del 18/06/18 al 12/07/18

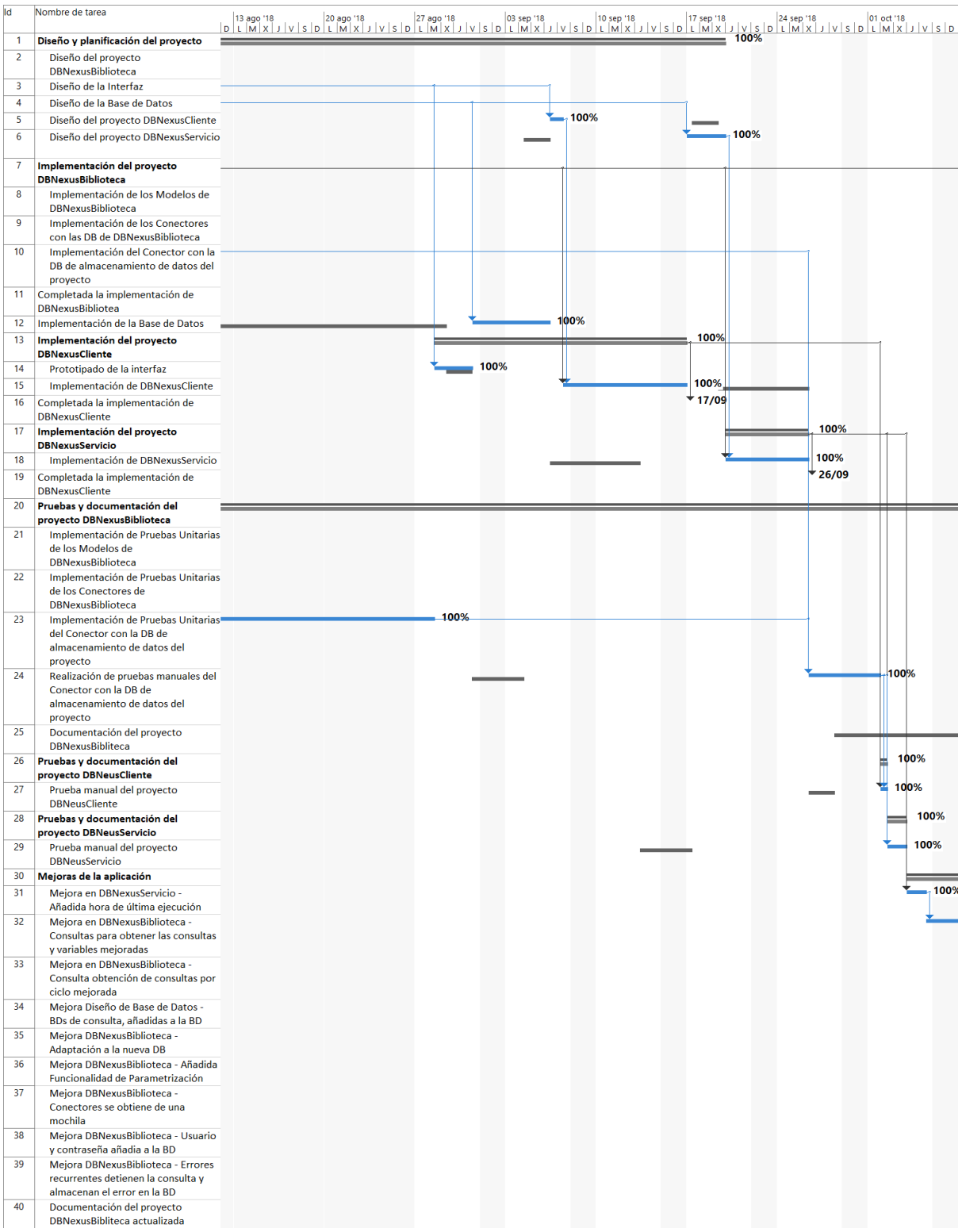

Figura 3.4: Diagrama de Gantt del transcurso del Proyecto del 12/07/18 al 07/10/18

Figura 3.5: Diagrama de Gantt del transcurso del Proyecto del 07/08/18 al 02/12/18

Figura 3.6: Diagrama de Gantt del transcurso del Proyecto del 24/11/18 al 19/01/19

En cuanto a las horas dedicadas al proyecto, las tareas que se habían planificado, en las que se pretendían realizar en 330 horas, han sido necesarias 369 en total (dos semanas más) para realizarse. Asimismo, se han incorporado 256 horas más para realizar mejoras, por tanto, finalmente se han empleado 625 horas para finalizar el proyecto.

Capítulo 4

Análisis y diseño del sistema

4.1. Análisis del sistema

Para comenzar el análisis del sistema, se empieza buscando los actores que participan y los casos de uso de la aplicación.

En la aplicación se pueden diferenciar dos actores:

- **Usuario**: El usuario será el encargado de crear y administrar las consultas del sistema y sus variables. También podrá probar las consultas y se realizará una verificación de su funcionamiento antes de almacenarlas y de que sean ejecutadas por el servicio. Para poder probar y comprobar las consultas, el cliente deberá tener conexión al servidor donde se realizará la consulta.
- **Servicio**: El servicio obtendrá las consultas que se deban ejecutar del SGBD MySQL, las ejecutará cuando corresponda y almacenará sus datos en los tags correspondientes del sistema *Nexus Integra*.

A continuación se exponen todos los casos de uso:

- **Crear Consulta**: Este será utilizado por el usuario para crear una consulta. Más adelante, en el apartado 4.2, se definirán las propiedades de cada consulta. Además, antes de almacenarla, se realizará un prueba de su ejecución en la que se comprobará que la consulta devuelva resultados válidos.
- **Ver Variables**: Mostrará todos los nombres de variables de la aplicación que serán igual a los tags del sistema *Nexus Integra* y permitirá borrar aquellas que no tengan consulta. Cada nombre de variable o tag se obtendrá del nombre de la columna de su consulta y en estos se almacenarán los valores obtenidos por la consulta.
- **Eliminar Variable sin consulta**: Se puede borrar una consulta y no su nombre de variable, para que otra consulta no escriba en los datos almacenados. Del mismo modo, también se puede querer borrar una variable del sistema *Nexus Integra* para que no

aparezca en el sistema, o para poder reutilizar su nombre sin los datos anteriores. Únicamente se permitirá borrar variables sin consulta o cuya consulta actual no esté utilizando la variable.

- **Ver Consultas**: Tanto el usuario como el servicio podrán listar las consultas almacenadas en el sistema para sus diferentes usos.
- **Modificar Consulta**: El usuario podrá modificar las propiedades de la consulta.
- **Eliminar Consulta**: El usuario podrá eliminar la consulta y sus variables.
- **Monitorizar Consulta**: El servicio podrá ejecutar la consulta, acorde a las restricciones de sus parámetros y realizar los pasos necesarios para la monitorizacion y/o historización, los cuales se describen a continuación:
  - **Buscar/Crear Tags**: Se encarga de buscar si los nombres de tags o variables existen en el plataforma *Nexus Integra* y, si existen, devuelve su identificador, en la plataforma; si no existen, los crea y devuelve su identificador.
  - **Monitorizar RealTime**: Si los parámetros de la consulta lo indican, se almacenará en el servicio de tiempo real (temporal) de la plataforma *Nexus Integra*, lo cual permitirá que aparezca en gráficas a tiempo real en el sistema; esta opción se podrá desactivar para minimizar el consumo en casos donde no se requiera.
  - **Monitorizar Historico**: En este caso, el valor obtenido por la consulta se almacenará siempre y, de forma permanente, en la plataforma *Nexus Integra*.

Para representar los diferentes actores de la aplicación, así como los casos de uso que necesitan realizar, se ha creado un diagrama de casos de uso, el cual se puede visualizar en la figura 4.1.

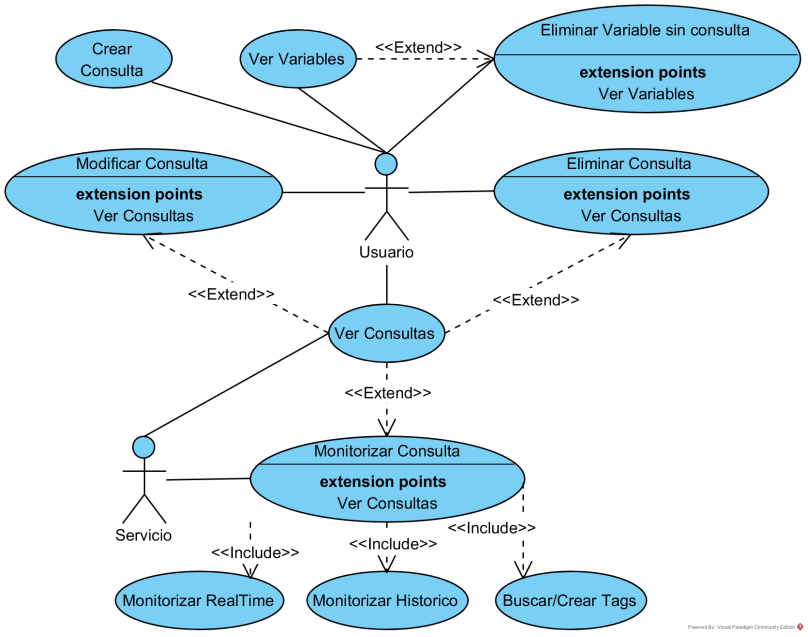

Figura 4.1: Diagrama de Casos de Uso del Proyecto

Al comenzar con el diseño de la aplicación, se ha decidido dividir en tres subproyectos, puesto que los dos actores necesitan aplicaciones diferentes. Para el usuario, una interfaz gráfica que le permita interactuar con las consultas almacenadas; para el servicio, sin embargo, no es necesario interfaz ni consola, únicamente un proceso ejecutándose en segundo plano. No obstante, ambos emplean los mismos objetos para trabajar con las consultas.

Los tres proyectos en los que se dividirá la aplicación serán:

- **DBNexusBiblioteca**: Este proyecto contendrá, por un lado, las variables de configuración que usarán los otros dos proyectos para su funcionamiento; por otro lado, los modelos, y por último, los conectores, para los diferentes SGBD que permitirán conectarse a las bases de datos en las que se realizarán, y en la que se almacenaran las consultas.

- **DBNexusCliente**: Este proyecto, que incluirá el proyecto DBNexusBiblioteca, también contendrá las ventanas de Windows Forms utilizadas por el cliente, para probar, añadir, modificar y eliminar consultas (a realizar por el servicio) y variables registradas en el sistema.

- **DBNexusServicio**: Este proyecto, que incluirá el proyecto DBNexusBiblioteca, también contendrá los procesos necesarios para realizar las consultas que correspondan periódicamente y almacenar sus resultados en la plataforma *Nexus Integra* conectándose a ella como un plugin.

En la figura 4.2 se puede visualizar cómo interaccionan estos tres proyectos y cómo almacenan o se nutren de los datos con la base de datos.

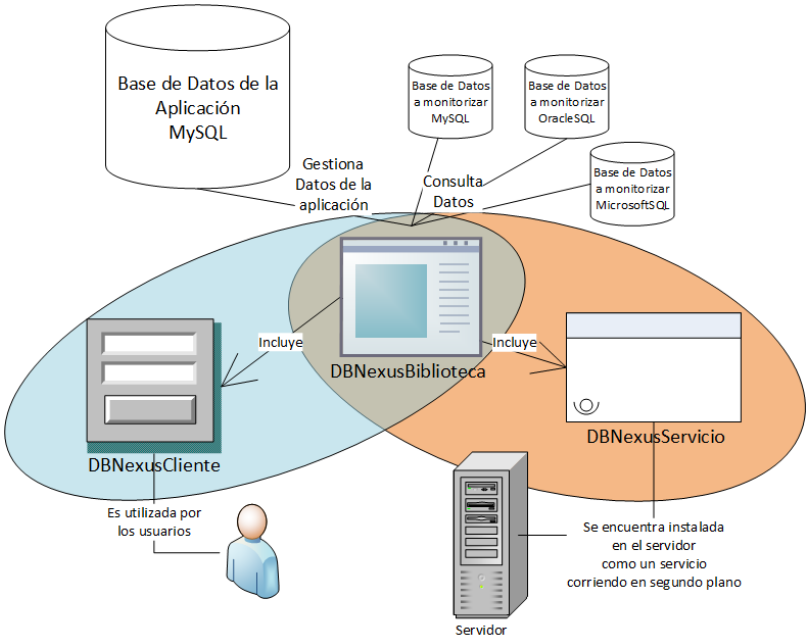

Figura 4.2: Interacción entre los proyectos que componen la aplicación y las bases de datos

4.2. Diseño de la arquitectura del sistema

4.2.1. Diseño de la arquitectura del sistema: DBNexusBiblioteca

En esta sección se explican los diferentes paquetes y clases que componen esta biblioteca y para qué serán utilizados posteriormente en los otros dos proyectos. Hay una explicación más detallada de cada clase de este paquete en la sección A.1 de los anexos.

En primer lugar, cabe destacar, dos clases utilizados por la mayoría de clases del proyecto: la clase *Constantes* y la clase *Log*. La clase *Constantes*, como su propio nombre indica, contiene valores constantes de propiedades de la aplicación (cargadas desde un fichero de configuración) o los conectores para conectarse a las bases de datos. Paralelamente, la clase *Log*, es usada para almacenar todos los mensajes de registro de la aplicación; esta clase utiliza una librería llamada log4net que permite filtrar, almacenar y/o enviar los mensajes de registro.

Por otro lado, los paquetes que engloban el resto de la biblioteca son dos: el paquete *Model* y el paquete *Conectors*. El paquete *Model* contiene los modelos necesarios para las otras dos aplicaciones. Con los requisitos de la aplicación, se ha determinado necesario crear las siguientes clases:

- *Consulta*: Contendrá los atributos de las consultas a ejecutar en las bases de datos y las variables propias de la consulta.
- *Variable*: Contendrá los atributos y el último valor de las variables de las consultas; además, un requisito para versiones posteriores del proyecto es poder realizar operaciones con la variable antes de almacenarla, por lo que posee una *Operacion*.
- *Operacion*: Esta clase contiene la forma de operar los datos y también la posibilidad de encolar otra operación para ejecutar varias sobre un dato. Asimismo, se ha creado un enumerado para indicar el tipo de operación de la que se trata.

En el paquete *Conectors* se pueden encontrar los diferentes conectores que usarán las aplicaciones para conectarse con las bases de datos y la plataforma de monitorización. Para ello, se crean, en este paquete, otros tres paquetes:

- *DB*: Este paquete contendrá los conectores para conectarse a los tres tipos de bases de datos requeridos, así como para conectarse a la base de datos donde la aplicación almacenará las consultas programadas. Para facilitar la creación de los conectores, este paquete cuenta con una clase fábrica utilizada para crear las conexiones. A fin de optimizar las conexiones, cada clase *DBConector*, al crearse, genera una colección atómica de un máximo definido en las *Constantes* y la llena de los conectores que utilizará para realizar sus consultas. En el momento de realizar una consulta, esta clase:
  1. Extrae un conector de la colección.
  2. Abre la conexión.
  3. Usa el conector.

4. Cierra la conexión.
5. Vuelve a introducir el conector en la colección.

- *DAO*: Este paquete contendrá la clase con las funciones necesarias para leer, añadir, editar y borrar las consultas que debe realizar la aplicación en la base de datos. Esta conexión será heredada del conector oportuno (en este caso, por requisitos, es el de *MySQL*), del paquete *DB*.

- *Nexus*: Este paquete contiene una única clase que utilizará las bibliotecas *CoreDataContracts* y *CoreServicesInterfaces*, proporcionadas por *Nexus Integra*, para crear tags (variables) en su plataforma y encapsular y enviar datos a su servicio de monitorización o de historización.

En la figura 4.3 se puede observar un diagrama minimizado de todas las clases del proyecto DBNexusBiblioteca; en la figura A.1 de los anexos aparece mas detallado.

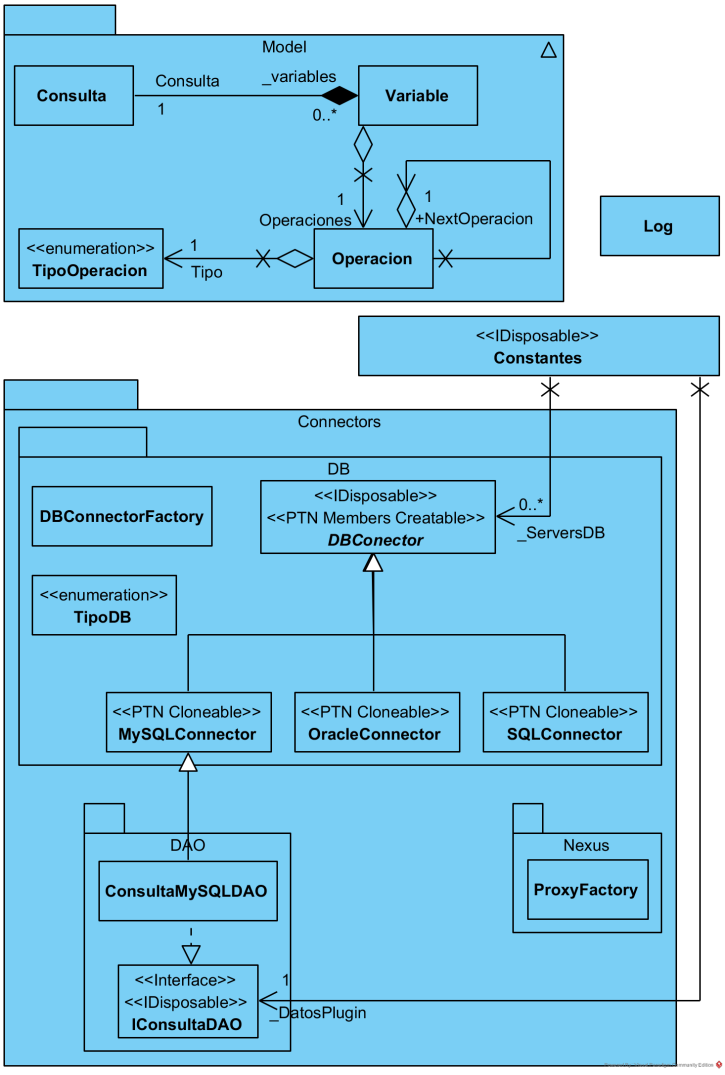

Figura 4.3: Diagrama de Clases de la Biblioteca

4.2.2. Diseño de la arquitectura del sistema: DBNexusCliente

En esta sección se detalla cómo se ha abordado el diseño de la arquitectura, de la parte del proyecto correspondiente al cliente, que lleva por nombre DBNexusCliente. Este proyecto incluye, como una de sus bibliotecas, el proyecto DBNexusBiblioteca que se expone en el apartado anterior, 4.2.1.

Para diseñar las interfaces de este proyecto, se hará uso de las bibliotecas que provee *Microsoft Windows* para crear formularios en sus ventanas, System.Windows.Forms. El presente proyecto incluye las interfaces que visualizará el cliente y que se desarrollan, más adelante, en la sección 4.3.

Se ha concluido que se pueden llevar a cabo todos los casos de uso del cliente con dos ventanas: una en la que muestre la lista de consultas almacenadas en el servidor, y otra en la que aparezcan los detalles de cada consulta; además, se hará uso de las ventanas de mensajes de alerta y confirmación, que se encuentran en la biblioteca System.Windows.Forms.

Lista de las Consultas

En la ventana de la lista de consultas almacenadas, se planea incluir dos botones con las funciones de actualizar la lista de consultas almacenadas y de crear nuevas consultas. El botón de crear una nueva consulta, abrirá la ventana de edición de consultas sin cumplimentar.

Paralelamente, al lado de cada fila, en la lista de consultas, se dispondrá de dos botones: uno para ejecutar la consulta una única vez, de forma inmediata y otro, para editar la consulta que abrirá la ventana de edición de consultas. La lista de las consultas aparecerá en una tabla, la cual se podrá ordenar por columnas con los siguientes campos a vista:

- **ID Sentencia**: Aquí se puede leer el nombre o identificador de la consulta.
- **Variables**: En esta columna se muestra la lista de variables que contiene la consulta, en una única línea y separadas por comas.
- **Base de Datos**: El identificador de la base de datos donde se ejecuta la consulta.
- **TActualización**: Periodo de ejecución de la consulta.
- **Realtime?**: Si la consulta es almacenada en el sistema de tiempo real de *Nexus Integra*.
- **Activo?**: Si la consulta está programada para ejecutarse en el servicio.
- **Ultima Ejecución**: La marca de tiempo de la última ejecución de la consulta.

También se considera que, para aquellas variables sin consulta asignada, se determinó necesario un desplegable y un botón para borrar la variable seleccionada. Por cuestiones de seguridad, al ejecutar esta acción aparece en pantalla una ventana de confirmación con un diálogo explicativo. Dicha ventana explica las consecuencias de esta acción y pregunta si se desea o no continuar, o si se desea borrar la variable, tanto del almacenamiento como del sistema *Nexus*

Integra, con el fin de reiniciar el nombre de variable. Si se selecciona esta última opción, aparece un nuevo mensaje de confirmación para asegurarnos, puesto que es la elección más crítica.

Por otro lado, hay que tener en cuenta que la interfaz no se debe bloquear mientras se realizan acciones en la interfaz, como la actualización de la lista de consultas o el borrado de las variables. Para estos casos, se utiliza el objeto *backgroundWorker*, proporcionado por las bibliotecas de formularios de *Microsoft Windows*, el cual permite ejecutar una función predefinida, en segundo plano.

Las únicas acciones en segundo plano en la ventana de listar las consultas son: actualizar la lista de consultas y borrar variables sin consulta asignada.

Creación, edición y borrado de una Consulta

En la ventana de la consulta almacenada, es posible crear una nueva consulta, editar una existente o borrarla. Al abrirse, la ventana recibirá como dato, o bien el identificador de la consulta que se va a editar o nada, en caso de que se desee crear una nueva consulta.

En esta ventana hay un formulario con todas las propiedades de la consulta modificable por el usuario: el identificador de la consulta (que tan sólo será modificable antes de crear la consulta), un desplegable con los servidores de datos disponibles, otro desplegable con las opciones de período de actualización (todas las opciones son múltiplos del tiempo de actualización almacenado como constante en el proyecto DBNexusBiblioteca), dos botones activables para indicar si se ejecuta la consulta en el servicio y si se almacena en el sistema de tiempo real, dos conjuntos de botón y selector de fecha para indicar las fechas de inicio y fin de la consulta y si se van a utilizar y, por último, un espacio para introducir texto donde se introducirá la sentencia SQL de la consulta que se desea almacenar.

Una vez introducidos los datos en una nueva consulta o editados algunos en una ya existente, no se permitirá guardar la consulta directamente, sino que es necesario probarla para así validar los datos que devuelve la consulta. El validador comprobará que:

- Todos los datos devueltos son en punto flotante (*double*) o transformables a punto flotante.
- Ninguno de los nombres de las columnas, que serán los nombres de variable o tag, se repita o se encuentre en uso como identificador de variable en otra consulta.
- El tiempo de ejecución de la consulta sea menor al periodo de ejecución de la consulta en el servicio. Si no, la consulta se acumulará en la cola de ejecución del servicio una y otra vez, siendo más lenta su ejecución que su encolamiento y pudiendo producir fallos.
- También verifica que la sentencia sólo responda una fila o, en caso de que haya más de una fila, comprueba si tiene un campo de marca de tiempo o si tiene campos de identificador de variable y ninguno de los nombres de variable generados es repetido.

Además de los campos indicados anteriormente, se utilizarán otros dos para controlar los datos que devolverá la sentencia introducida: una tabla con los resultados de la sentencia y una

lista con las variables de la consulta, tanto las utilizadas como las que se encuentran en desuso. Las variables en desuso de la lista, irán acompañadas de un botón que las permita borrar. Este botón de borrado creará los mismos mensajes de confirmación que al borrar una variable desde la ventana de lista de las consultas, expuesta en el apartado anterior.

En esta ventana se dispondrá de un botón para salir sin almacenar los cambios, el cual tendrá el mismo efecto que cerrar la ventana desde la cruceta. También se dispondrá de un botón para almacenar la consulta y, cuando se esté editando la consulta, también se dispondrá de uno para eliminarla. El botón de borrado hará que aparezca un diálogo de confirmación antes de su eliminación, dando también la opción de borrar todas las variables de la consulta, tanto del almacenamiento como de la plataforma *Nexus Integra* o de mantener las variables.

Para poder ejecutar todas estas funciones sin bloquear la interfaz, se crean dos disparadores de trabajos en segundo plano: uno para la ejecución de las pruebas, el cual cancela su ejecución si se realiza cualquier modificación mientras se ejecuta y otro para el borrado de las variables desde la lista.

Las tareas de almacenaje y borrado de la consulta se ejecutan en primer plano, ya que son críticas y cerrarán la ventana tras ejecutarse correctamente.

El diagrama de clases que describe este proyecto se muestra en la figura 4.4 y más detalladamente en la figura B.18 de los anexos.

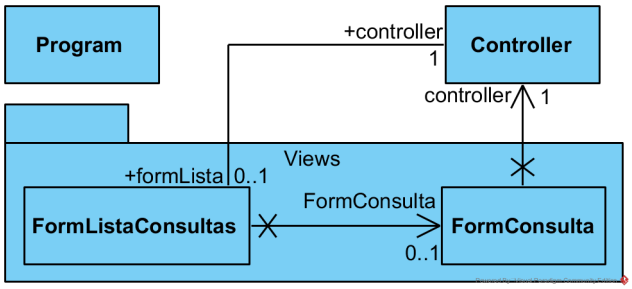

Figura 4.4: Diagrama de Clases del proyecto DBNexusCliente

4.2.3. Diseño de la arquitectura del sistema: DBNexusServicio

El servicio de monitorización será un programa compilado como .exe que se instalará en el servidor *Microsoft Windows* como servicio. Como medida de seguridad, este servicio será dependiente del servicio de base de datos *MySQL* y de varios servicios de la plataforma *Nexus Integra*, para que arranque después de ellos y se detenga si cualquiera se detiene.

Para este proyecto se ha hecho uso la plantilla de Servicio de Windows (.NET Framework) que incluye, entre otras, la biblioteca *System.ServiceProcess*, además de añadir posteriormente la biblioteca DBNexusBiblioteca del primer proyecto expuesto.

La plantilla crea por defecto dos clases: una denominada *Program*, para el arranque y otra, donde se indica cómo arrancar y parar el servicio, denominada *ServiceDBNexus*. Además, se

incluye otra clase *MonitorConsultas* en la que se definen los métodos del servicio.

La clase *Program* contiene la función de arranque *Main*, que arranca los servicios del proyecto, en este caso *ServiceDBNexus*. Del mismo modo, la clase *ServiceDBNexus* invoca el método *OnStart()* de la clase *MonitorConsultas*.

La clase *MonitorConsultas* contiene un diccionario de datos concurrente, que contiene una clave para cada ciclo (periodo mínimo entre ejecuciones de la misma consulta) pendiente de ejecución y, como valor, la colección de consultas que se deben ejecutar en ese ciclo. En este diccionario se irán encolando las consultas obtenidas por el método *GetConsultasCiclos*, de la clase *Conectors.DAO.ConsultaMySQLDAO* del proyecto DBNexusBiblioteca. Este método será llamado por el método *ProductorConsultas*, que borrará los ciclos ya pasados (en caso de error o embotellamiento), comparará que las consultas encoladas pendientes coincidan con las proporcionadas por la base de datos y sustituirá o encolará las nuevas colecciones de consultas.

Este método será ejecutado de forma repetitiva por un temporizador de la clase *System.Threading.Timer*, hasta que reciba la orden de detención. La obtención de consultas a ejecutar de la base de datos, se ejecutará cada *Constantes.TsActulizaConsultas* segundos (se ha considerado que 10 minutos es un tiempo adecuado). Puesto que la clase *Timer* no es muy exacta a la hora de repetir sus ejecuciones o se puede producir fallos de comunicación al obtener las consultas de la base de datos, se obtiene las consultas de los próximos *Constantes.TsActulizaConsultas*∗2,2 segundos, dando la posibilidad de obtener dos veces la misma consulta.

La clase *MonitorConsultas* también es la que contiene el método consumidor de las consultas, *ConsumConsDoWork*. Como en este caso es necesario que la ejecución sea lo más cercana al inicio del ciclo, se usa un *System.ComponentModel.BackgroundWorker* con un bucle activo cuya primera tarea sea dormir tantos milisegundos como quedan, hasta el inicio del siguiente ciclo. Al despertarse, comprueba el ciclo actual, intenta obtener y borrar la colección de consultas del diccionario concurrente. Si la colección extraída tiene alguna consulta por ejecutar, crea otro *BackgroundWorker* encargado de ejecutar la colección de consultas. Tras delegar la ejecución de las consultas, si no se ha detenido la ejecución del servicio vuelve a ejecutar el bucle desde el comienzo, por lo que se duerme hasta el inicio del siguiente ciclo. Asimismo, para corroborar el buen funcionamiento de este método, se utiliza uno de los métodos de la clase *BackgroundWorker*, que permite indicar el progreso del hilo y, antes de finalizar el bucle, se indica que el progreso a avanzado al ciclo actual. El método que se ejecutará, en un hilo diferente, al cambiar el progreso, únicamente escribe en el registro que ha finalizado el ciclo indicado.

Para la toma de datos y su muestreo, es necesario tomar la muestra en el instante requerido y ser lo más preciso posible. Por ello, el inicio de la toma de datos se ha programado, con un hilo en segundo plano, con un bucle infinito (hasta que otro hilo lo detiene), cuyo único objetivo es crear otros hilos, encargados de obtener los datos y almacenarlos en la plataforma de monitorización. Para que el proceso despierte en el periodo exacto en el que se tratan de monitorízar los datos, al principio del bucle, se pone a dormir al proceso el tiempo restante hasta el inicio del siguiente ciclo y, antes de finalizar el bucle, también se le duermen 10 milisegundos (si no, puede llegar a ejecutar el bucle más de una vez por ciclo). Para la ejecución del método *ProductorConsultas*, no es necesaria tanta precisión, por esa razón se utiliza la clase *Timer*, que tiene un error de desfase entre sus periodos de ejecución.

Los dos métodos que están ejecutándose repetidamente en el servicio *ProductorConsultas* y *ConsumConsDoWork*, inician su ejecución con el método *OnStart* y, del mismo modo, cesan su ejecución con el método *OnStop*.

Para obtener los valores de las consultas y almacenarlos en la plataforma de monitorización, el hilo consumidor que ejecuta el método *ConsumConsDoWork*, contiene un bucle asíncrono definido con método *ForEach* de la clase *System.Threading.Tasks.Parallel* que permite, dada una colección (en este caso de consultas), crear un hilo para cada elemento de la colección y ejecutar una función para cada uno. El método *ForEach* también permite indicar un número de hilos simultáneos máximos; como máximo de hilos simultáneos, se ha utilizado el parámetro *Constantes.ConexNexusSimult*, ya que así se limita el número de hilos que intentarán establecer conexión con la plataforma simultáneamente al número definido en los parámetros y no es necesaria ninguna otra medida de control. Cada hilo creado por el bucle, ejecutará el método *EjecutarConsulta* y esperará a que finalice, pues como respuesta, este método indica si la ejecución ha tenido éxito o si se ha producido algún error.

El método *EjecutarConsulta* se encarga de ejecutar las consultas y almacenarlas según su configuración, además de controlar errores, grabar el estado de ejecución en los registros y, en caso de un error recurrente en una consulta, detenerla y registrar el motivo de error en una tabla específica de la base de datos.

Para ilustrar el funcionamiento de este fragmento del proyecto, se hace referencia a la figura 4.5 donde se encuentra su diagrama de clases. Para un diagrama más detallado, se puede consultar la figura B.19 de los anexos.

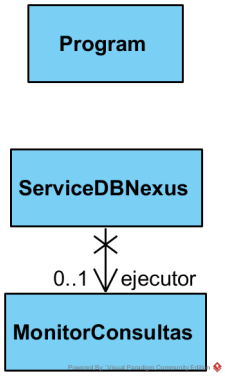

Figura 4.5: Diagrama de Clases del proyecto DBNexusServicio

4.2.4. Diseño de la base de datos

Este diseño de base de datos está enfocado a minimizar los datos almacenados, que serán los descritos en el modelo y el coste de las consultas. En un principio se determinaron tres tablas:

- *plug_conexdb*: Almacena las bases de datos en las que se realizarán las consultas, así como sus cadenas de conexión y la autentificación necesaria para conectarse.
- *plug_consulta*: Almacena todos los datos de las consultas, a excepción de las variables que

contienen; la columna *conexDbId* hace referencia a la clave primaria de una fila de la tabla *plug_conexdb*.

- *plug_variables*: Almacena las variables de las consultas y sus datos, ; la columna *plugConsID* puede ser nula (en el caso de que la consulta haya sido borrada y se desee mantener la variable) o hacer referencia a una clave primaria de la tabla *plug_consulta*.

Tanto en la tabla de consultas como en la de variables, se ha incluido el *AgentID* que sirve para identificar el origen de datos dentro de la plataforma Nexus. Incluyendo este identificador en la tabla, se puede separar, en la misma base de datos, consultas que en la plataforma provengan de orígenes diferentes. Si, por ejemplo, se tienen unas consultas diseñadas por el departamento de contabilidad y no se desea que sean visibles por otro departamento, se crea un origen de datos diferente en la plataforma y se indica su *AgentID* en el fichero de configuración de la interfaz gráfica. Cabe tener en cuenta que los tags o nombres de variables son únicos por origen de datos, por tanto, para mantener la integridad del sistema, se determina que la combinación de *AgentID* y *nomTag* de la tabla de variables debe ser única. Del mismo modo, también se ha aplicado a los campos *AgentID* y *nomCons* de la tabla de las consultas, ya que, como en cada interfaz gráfica sólo se listan las consultas del agente para el que está configurado, no llevará a confusión.

Para identificar cada fila se ha determinado una clave primaria numérica, ya que a la hora de relacionarlas son menos costosas. Además, se han incluido índices únicos de tipo árbol, en las columnas por las que el proyecto filtra al consultar. Los índices creados son de un conjunto de dos valores *AgentID* y *nomCons* en la tabla de las consultas (*plug_consulta*), y *AgentID* y *nomTag* en la tabla de las variables (*plug_variables*).

Asimismo, posteriormente, para llevar un registro a largo plazo de los fallos, se incluyó otra tabla llamada *plug_errores_consulta* que almacena errores en la ejecución de la consulta y posee una columna que hace referencia a la consulta que produce el error de la tabla *plug_consulta*.

Por otro lado, para reducir la cantidad de código SQL en el proyecto, se crean dos vistas con los datos que se esperan para cumplimentar los modelos. Las vistas son:

- *plug_cons_var*: Con los datos de las consultas.
- *plug_var*: Con los datos de las variables.

También se han incluido reglas de integridad para los campos que así lo requieran, como doble medida de seguridad, ya que el proyecto cliente también verifica que los campos introducidos sean válidos. Por nombrar algunas de las validaciones de integridad de la tabla *plug_consulta*, se comprueba que el *tAct* (tiempo de actualizado) sea mayor a cero y que la *fechFin* (instante en el que se detiene la ejecución de la consulta) no sea menor a *fechaIni* (fecha en la que la consulta inicia su ejecución).

Por último, por motivos de seguridad, se ha decidido crear un usuario en la base de datos de almacenamiento, específico para la aplicación, que únicamente tenga acceso a las tablas y vistas descritas. Tendrá permisos de lectura, adición, edición y borrado en todas ellas a excepción de la tabla *plug_conexdb*, donde sólo tendrá permisos de lectura.

En la figura 4.6, se muestra el diagrama entidad relación de la base de datos del proyecto. Asimismo se puede encontrar el diseño lógico de la aplicación en la sección A.2 de los anexos.

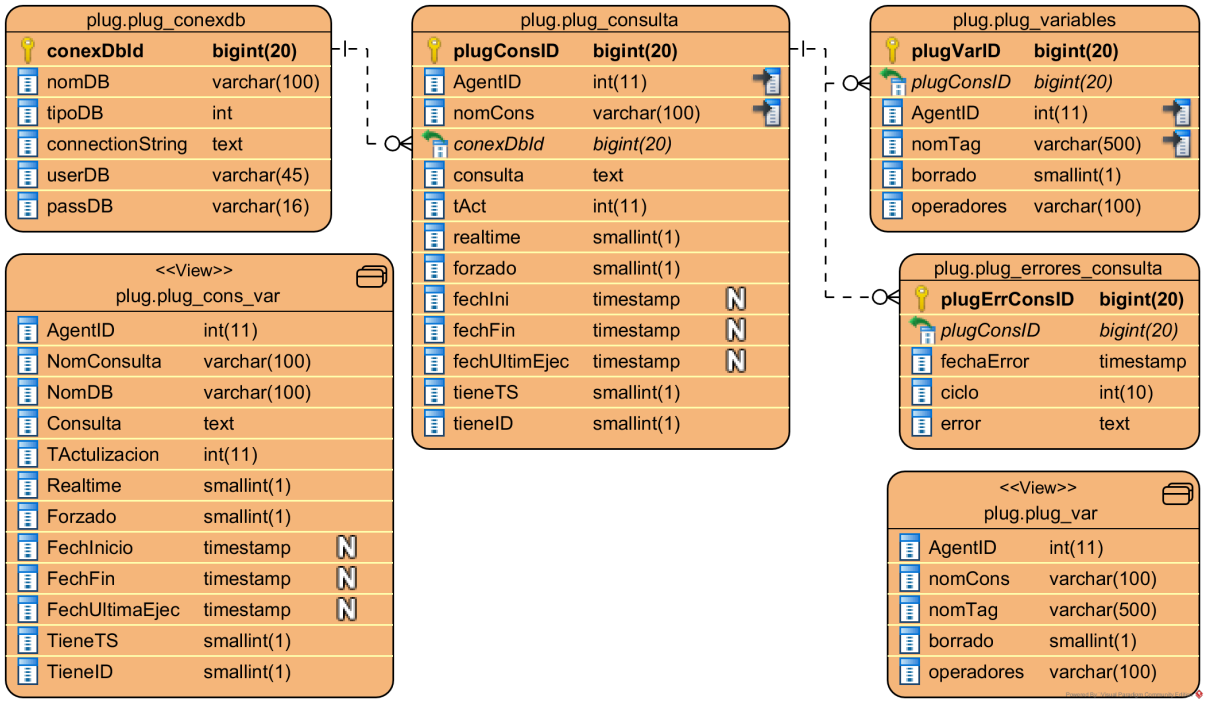

Figura 4.6: Diagrama Entidad Relación de la Base de Datos donde se almacenarán los datos del Proyecto

4.3. Diseño de la interfaz

Para el proyecto actual, sólo se han necesitado diseñar dos interfaces gráficas para el proyecto DBNexusCliente, a continuación se presentarán las interfaces.

Dado que *Microfort Visual Studio* posee una interfaz amigable para el diseño de formularios en ventanas (se selecciona el elemento y se arrastra a la ventana para incluirlo y posicionarlo) se ha decidido realizar el prototipado de la interfaz en el mismo programa, sin darle funcionalidad a los elementos hasta confirmar el prototipo [6].

Los prototipos se han diseñando con el objetivo de ser claros y de ayudar al usuario al manejo de la aplicación. Finalmente, los prototipos aceptados son los que se pueden observar en las figuras 4.7 y 4.8.

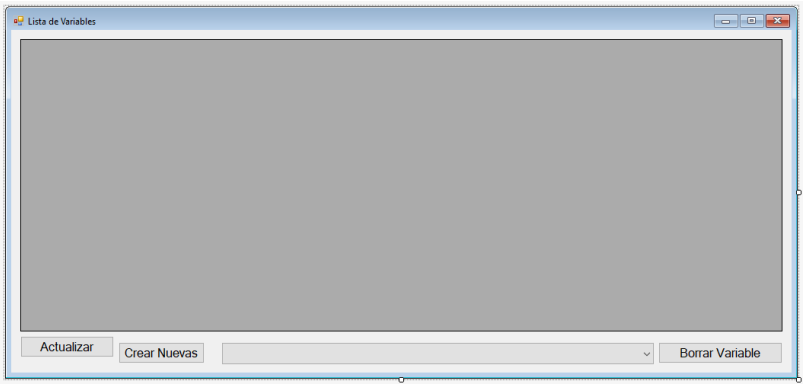

Figura 4.7: Diseño de la interfaz de la ventana de la lista de consultas

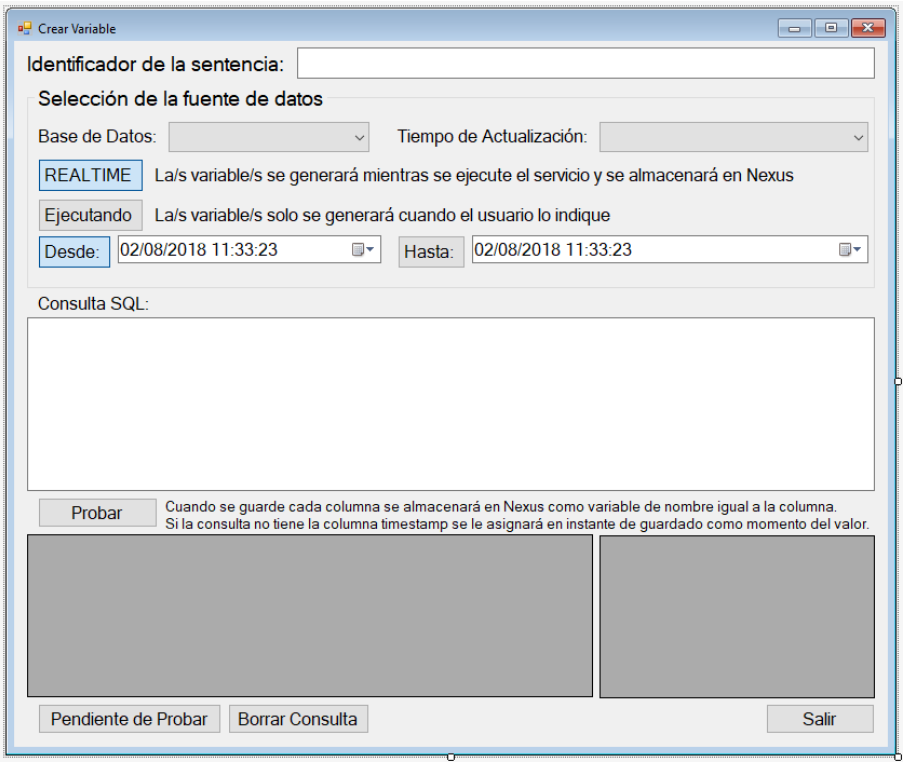

Figura 4.8: Diseño de la interfaz de la ventana para añadir o editar consultas

Una vez implementados los prototipos e introducidas consultas en la aplicación, las interfaces diseñadas lucirían como se muestra en las figuras 4.9 y 4.10.

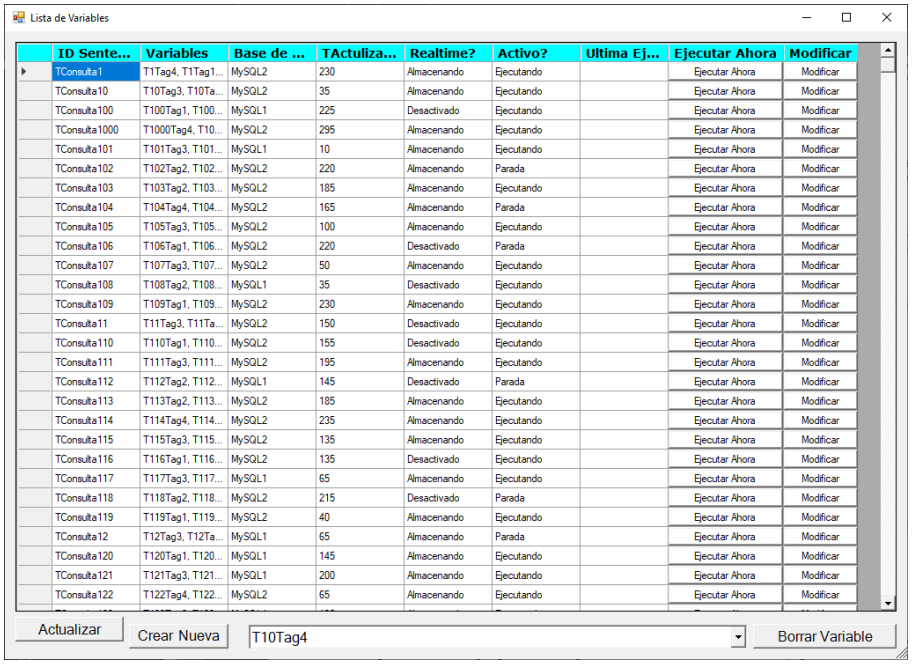

Figura 4.9: Muestra cómo se ven los datos de las consultas en la ventana de la lista de consultas

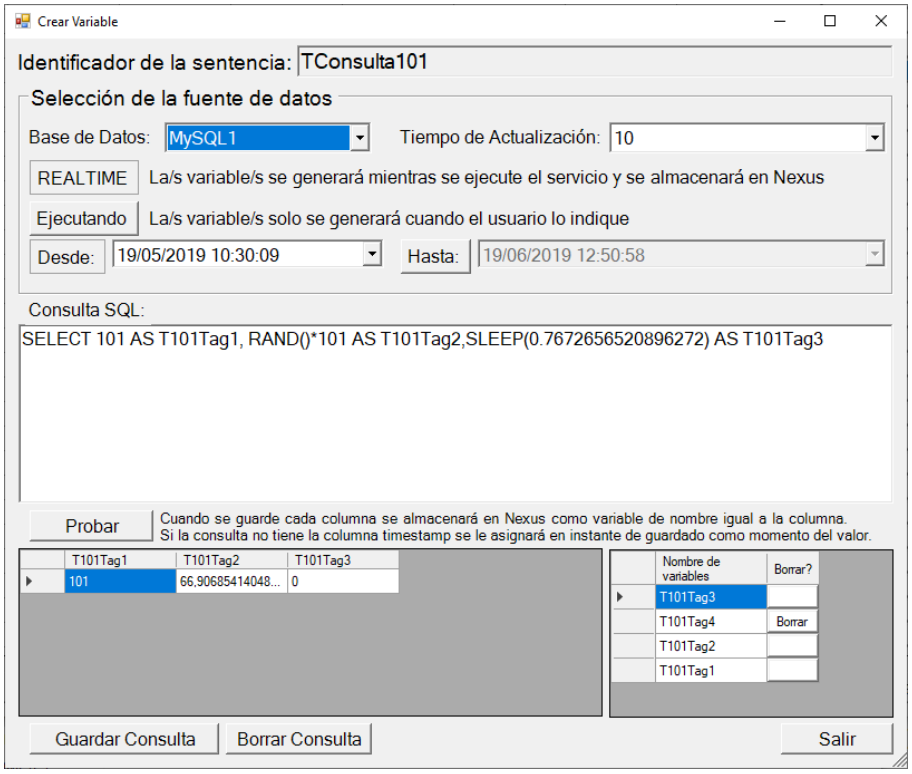

Figura 4.10: Muestra cómo se ven los datos de una consulta en la ventana para añadir o editar consultas

Además de estos prototipos, también se dispondrán de mensajes de confirmación, alerta y error, mostrados en las clásicas ventanas de *Microsoft Windows*. Para este fin, se puede visualizar un ejemplo de estas en las figuras 4.11 y 4.12.

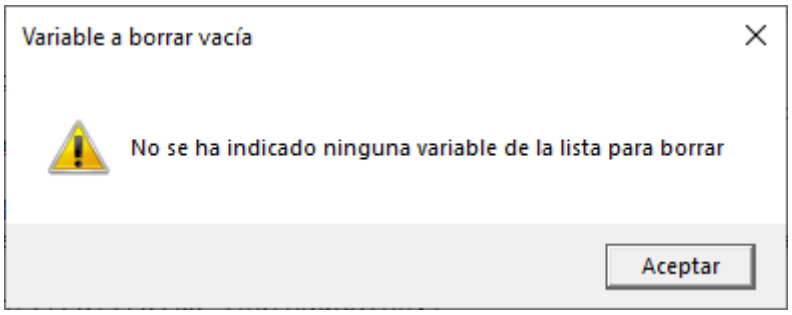

Figura 4.11: Ventana de alerta de *Microsoft Windows* que aparece al pulsar el botón 'Borrar Variable' sin seleccionar una del desplegable

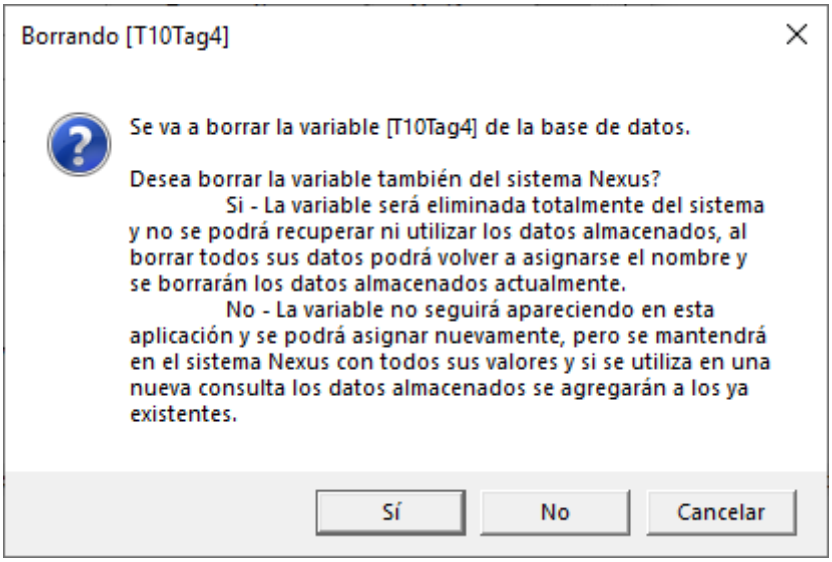

Figura 4.12: Ventana de confirmación de *Microsoft Windows* que aparece al intentar borrar una variable

Capítulo 5

Implementación y pruebas

Al comienzo de este proyecto, en la empresa donde se contextualiza, el *ITC*, no se disponía de un servicio de control de versiones de código fuente, como *Team Foundation Server* (*TFS*) o *GIT*. Sin embargo, durante el desarrollo del proyecto, se ha desplegado un servicio *TFS* con el fin de tener un control de versiones para los diferentes softwares que se desarrollan en la empresa.

Desde la implementación del servicio *TFS*, este proyecto ha hecho uso de él. Por tanto, únicamente se dispone de un control de versiones desde el final de las pruebas del servicio, una vez se finalizadas las pruebas de la biblioteca y del cliente. De todos modos, al incorporar el proyecto al *TFS*, se ha configurado para que, en cada actualización de versión, se comprueben las pruebas unitarias realizadas sobre el proyecto biblioteca. Además, se han creado las ramas que se consideran oportunas para el proyecto: para versiones desarrollo (llamada *develop*), para versiones pruebas de usuarios (llamada *user*) y para producción (llamada *master*).

5.1. Detalles de implementación

5.1.1. Implementación de la base de datos

Como se expuso anteriormente, en el apartado 2 (Descripción del proyecto), la aplicación almacenará sus consultas programadas en una base de datos que, por requisito del cliente, debe ser MySQL. MySQL es el sistema gestor de bases de datos relacional más popular de código abierto; fue desarrollado por *Oracle Corporation* y se distribuye bajo dos tipos de licencias: licencia pública general (*GNU*) y licencia comercial [21]. En este proyecto se usará una licencia *GNU* en el MySQL. Los softwares utilizados para conectarse y editar la base de datos son *MySQL Workbench* y *SQLyog*.

Como ya se ha mencionado en el apartado 4.2.4 (Diseño de la base de datos), los campos que relaciona las tablas entre sí son campos numéricos. También se han añadido índices en los campos más utilizados por la aplicación para realizar búsquedas.

Asimismo, para mantener la integridad de la base de datos, se ha considerado oportuno crear disparadores para controlar todos los casos posibles (los ejecutados por la aplicación y también la edición directa en la base de datos). Se han creado disparadores para las siguientes situaciones:

- Antes de cada inserción en la tabla *plug_variables*.
- Antes de cada actualización en la tabla *plug_variables*.
- Después de la actualización de la tabla *plug_consulta*.

Como medida de seguridad, al almacenar las contraseñas de las bases de datos a las que la aplicación se conecta, en la tabla *plug_conexdb*, se codifica la contraseña con una encriptación *AES*[1]. Como clave para la encriptación *AES*, se usa el *MD5*[2], de la cadena resultante de concatenar las cadenas: de conexión, el nombre de usuario y una clave precompartida con la aplicación (la cual, no se encuentra en la base de datos). De este modo, sólo la aplicación debe poder desencriptar la contraseña para conectarse a la base de datos y, si se modifica la cadena de conexión sin recodificar la contraseña, no es posible obtenerla.

Para facilitar la inserción de consultas y variables en la base de datos de la aplicación y no tener que incluir largas transacciones SQL, se crearon seis procedimientos que utilizará la aplicación. Se han determinado necesarios los siguientes procedimientos:

- *plug_insert_plug_cons_var*: Inserta una consulta, con sus variables y operaciones.
- *plug_update_plug_cons_var*: Edita una consulta, con sus variables y operaciones.
- *plug_delete_plug_cons_var*: Borra una consulta y, dependiendo de sus parámetros, también sus variables (sólo de la aplicación o también de la plataforma de monitorización). Sería más recomendable borrar la variable de la plataforma desde la biblioteca que Nexus ha proporcionado pero, al no disponer de esta función, se tiene que hacer directamente sobre la base de datos.
- *plug_delete_plug_var*: Borra una variable, sólo de la aplicación o también de la plataforma de monitorización.
- *plug_update_plug_cons_fejec*: Modifica la fecha de última ejecución de todas las consultas pasadas por parámetro.
- *plug_reg_err_plug_cons*: Registra un error recurrente y para la consulta.

Por último, con el fin de obtener las consultas, junto a su próximo ciclo de ejecución y su periodicidad, se ejecuta la siguiente consulta que, en su primer campo, obtiene el próximo ciclo de ejecución y en el segundo, el número de ciclos en volver a ejecutarse. La consulta es:

[1] *AES*: El *Advanced Encryption Standard* es un algoritmo de cifrado por bloques que utiliza una clave para encriptar los datos y únicamente es desencriptable con la misma clave [4].

[2] *MD5*: El *Message-Digest Algorithm 5* es un algoritmo de reducción criptográfico de 128 bits utilizado para determinar la autenticidad de los datos, ya que un mismo bloque de datos siempre obtendrá la misma firma única [23].

```
SELECT CAST(IF(pcv.FechInicio,
     ROUND(
       UNIX_TIMESTAMP(NOW()) / @tAct +
       MOD(pcv.nCicRep -
           ROUND(pcv.nCicRep -
             MOD(cicloIni, pcv.nCicRep)) +
           ROUND(pcv.nCicRep -
             MOD(ROUND(UNIX_TIMESTAMP(NOW()) / @tAct),
               pcv.nCicRep)),
         pcv.nCicRep)
     ),
     ROUND(
       UNIX_TIMESTAMP(NOW()) / @tAct +
       MOD(pcv.nCicRep -
           MOD(ROUND(UNIX_TIMESTAMP(NOW()) / @tAct),
             pcv.nCicRep),
         pcv.nCicRep)
     )
   ) AS UNSIGNED) AS 'ciclo',
   pcv.nCicRep,
   pcv.AgentID,
   pcv.NomConsulta,
   pcv.NomDB,
   pcv.Consulta,
   pcv.TActulizacion,
   pcv.Realtime,
   pcv.Forzado,
   pcv.FechInicio,
   pcv.FechFin,
   pcv.FechUltimaEjec,
   pcv.TieneTS,
   pcv.TieneID,
   pcv.Tags
 FROM (
     SELECT pcv.*,
         CAST(ROUND(pcv.TActulizacion / @tAct) AS UNSIGNED) AS 'nCicRep',
         CAST(
           ROUND(UNIX_TIMESTAMP(pcv.FechInicio) / @tAct)
         AS UNSIGNED) AS 'cicloIni'
       FROM plug_cons_var pcv
       WHERE @AgentID = pcv.AgentID AND
         pcv.Forzado = 0 AND
         IF(pcv.FechInicio, pcv.FechInicio <= NOW(), true) AND
         IF(pcv.FechFin, pcv.FechFin >= NOW(), true)
   ) pcv
 HAVING ciclo <= @fin
 ORDER BY ciclo, nCicRep;
```

Donde los parámetros con una `@` son variables locales que contienen:

- `@AgentID`: Identificador del origen de datos de la plataforma de monitorización.
- `@tAct`: Tiempo de un ciclo en segundos.
- `@fin`: Ciclo hasta el que se va a obtener.

5.1.2. Lenguaje y técnicas utilizadas para implementar la aplicación

Como ya se mencionó, en el apartado 2 (Descripción del proyecto), el lenguaje utilizado para la implementación de la aplicación desarrollada en este proyecto, por requisito del cliente, será C#[7] del el framework .NET[5]. .NET es un framework desarrollado por Microsoft, que aúna una gran cantidad de bibliotecas, con el fin de ser una herramienta de desarrollo para todo tipo de productos y que utiliza C# como lenguaje de programación. C# es un lenguaje de programación orientado a objetos desarrollado por Microsoft derivado del C/C++.

Como software para el desarrollo, se ha utilizado *Visual Studio*, ya que, por un lado, es el programa que posee la empresa y, a su vez, , es el recomendado por *Microsoft*; por otro lado, integra un administrador de paquetes (*NuGet*) y también posee integrados sistemas de control de versiones como el *GIT* o *Team Foundation Server* (*TFS*).

Como técnicas de programación a destacar en la implementación de cada proyecto de esta aplicación, cabe nombrar los siguientes:

- **Proyecto DBNexusBiblioteca**
  - Los conectores para cada tipo de base de datos heredan de una clase abstracta donde se definen todos los métodos y variables comunes.
  - Cada clase conector posee una mochila de conexiones, de la que extrae la conexión al utilizarla y la devuelve cuando ya no la necesita.
  - Para obtener los diferentes tipos de conectores existe una clase fábrica, a la que se le indica el tipo de base de datos, junto con la cadena de conexión, usuario y contraseña; esta fábrica devuelve el conector específico para cada tipo de base de datos.
  - El conector para almacenar los datos de la ejecución de las consultas, hereda del conector específico de la base de datos usada (MySQL), e implementa una interfaz donde se especifican todos los métodos utilizados por la aplicación para administrar las consultas a ejecutar, en la base de datos. Esto es así para que, en un futuro, si se emplea otra clase de gestos de base de datos, sea más sencillo implementar.
- **Proyecto DBNexusCliente**
  - Posee hilos en segundo plano para evitar que las funciones en ejecución bloqueen la interfaz.
- **Proyecto DBNexusServicio**
  - Utiliza un sistema productor-consumidor para almacenar las consultas a ejecutar.

- El productor se ejecuta cada cierto tiempo y consulta las consultas que se ejecutarán a continuación, las coloca en colecciones por ciclo y, por último, introduce las colecciones en un diccionario compartido, utilizando en el ciclo de ejecución como clave.
- El consumidor, cada ciclo consulta el diccionario compartido, obtiene las consultas del ciclo y ejecuta cada consulta en un hilo diferente.
- Con el fin de no crear excesivos hilos ejecutándose simultáneamente, los hilos que crea el consumidor para ejecutar cada ciclo, se introducen en una misma bolsa de hilos, que únicamente permite un máximo de hilos en ejecución simultáneamente; cuando uno de ellos finaliza, inicia la ejecución de otro hilo aleatorio de la bolsa.

Para implementar los diferentes proyectos, se ha hecho uso de los siguientes paquetes a destacar:

- **Proyecto DBNexusBiblioteca**
  - System.Collections.Concurrent: Posee colecciones concurrentes para ser utilizadas simultáneamente por diferentes hilos [9].
  - log4net: Es una herramienta para registrar mensajes en varios destinos de salida, configurados en un fichero de configuración. El fichero de configuración es modificable una vez compilada la aplicación [1].
  - System.Data.SqlClient, MySQL.Data y Oracle.ManagedDataAccess: Los conectores utilizados para conectar con las diferentes bases de datos, *MicrosoftSQL*, *MySQL* y *OracleSQL* [11] [19] [22].
  - CoreDataContracts y CoreServicesInterfaces: Proporcionadas por *Nexus Integra*, para interactuar con su plataforma de monitorización e historización.
- **Proyecto DBNexusCliente**
  - System.Windows.Forms y System.Windows.Forms.DataVisualization: Paquetes que contienen todas las clases necesarias para crear la interfaz de formularios en ventanas de *Windows* [15].
  - System.ComponentModel: Paquete utilizado para crear los hilos en segundo plano que ejecutarán tareas de la aplicación sin bloquear la interfaz [10].
- **Proyecto DBNexusServicio**
  - System.ServiceProcess y System.ServiceModel: Contienen las clases necesarias para compilar un servicio de *Windows* [13] [12].
  - System.Timers.Timer: Usado para ejecutar el productor cada cierto tiempo [17].
  - System.Threading y System.Threading.Tasks: Paquetes que contienen las clases utilizadas para administrar los hilos, tanto el consumidor que se ejecuta cada ciclo, como las bolsas de hilos que este crea [14] [16].

A excepción de los paquetes proporcionadas por *Nexus Integra*, todos se encuentran en el administrador de paquetes *NuGet* de *Windows* o forman parte del paquete System [18].

5.2. Verificación y validación

En esta sección se exponen las diferentes pruebas que se han llevado a cabo, a fin de verificar y validar el buen funcionamiento del proyecto.

Debido a que la empresa no se dedica exclusivamente al desarrollo de software, sino que sólo se dedica una sección de un área, no existe ninguna política que especifique las pruebas a realizar para la verificación y validación de los proyectos. Por ello, se han ido realizando las pruebas que el programador ha considerado oportunas durante la realización del proyecto, teniendo en cuenta el poco tiempo que se ha destinado para la realización de las mismas, por necesidades de la empresa.

Como el software de compilación *Visual Studio* permite marcar partes del código con marcas como TRACE o DEBUG para que, durante su compilación, se incluyan o ignoren, se ha utilizado de la siguiente forma:

- Cuando la marca DEBUG se encuentra activa, no se envían los datos recopilados a la plataforma nexus.
- Cuando la marca TRACE se encuentra activa, de forma independiente a la marca DEBUG, se escriben en el registro (con ayuda de la clase Log del proyecto biblioteca) todos los pasos que realiza la aplicación, como la ejecución de las consultas a las bases de datos o el inicio de un ciclo en el servicio.

Ya que estos fragmentos de código no son compilados en la aplicación de producción, no restaran rendimiento a la aplicación.

Al constar este proyecto de tres subproyectos, se expondrán las pruebas de verificación y validación realizadas en cada uno, de forma individual.

5.2.1. Pruebas realizadas en el proyecto DBNexusBiblioteca

Ya que el proyecto no es posible ejecutarlo por sí mismo e interactuar con él mediante consola o interfaz, se han desarrollado pruebas automatizadas. Para ello, se ha creado un proyecto paralelo con un conjunto de pruebas unitarias y de integración.

En las pruebas unitarias se ha probado que todas las clases del paquete modelo actúan e interactuan como se espera de ellas. Es decir, cuando se crea una clase con un conjunto de datos, se comprueba que esos datos son los que componen la clase.

Por otro lado, como pruebas de integración, se han programado: por un lado, pruebas simples de lectura de datos de diferentes bases de datos, sabiendo el resultado que se debía obtener; por otro lado, la creación, lectura, edición y borrado de consultas y sus variables en la base de datos de almacenamiento del proyecto.

5.2.2. Pruebas realizadas en el proyecto DBNexusCliente

Independientemente de las pruebas de la biblioteca DBNexusBiblioteca, incluida en este proyecto, al tratarse de una interfaz gráfica, sólo ha sido posible realizar pruebas manuales, ya que las pruebas automatizadas sobre una interfaz gráfica, sin experiencia previa, requieren de una cantidad de tiempo de la que no se disponía.

Las pruebas realizadas han sido de aceptación, para comprobar, tanto que los requisitos demandados por el cliente se cumplían, como que la interfaz fuese comprensible para los usuarios. Estas pruebas han sido realizadas principalmente por el propio programador del proyecto, siendo algunas otras realizadas por otros compañeros del área de trabajo y con el visto bueno del cliente.

5.2.3. Pruebas realizadas en el proyecto DBNexusServicio

Ya que este servicio sólo interactua con los datos de la base de datos y, anteriormente, se ha verificado en las pruebas de DBNexusBiblioteca que los datos son leídos correctamente, la única prueba que queda por realizar, es verificar que las consultas son ejecutadas en tiempo y forma adecuados.

Para ello, se utilizará un proyecto compilado con la marca TRACE activada; con él se comprobará que las consultas ejecutas en cada ciclo, son las configuradas en la base de datos. Además, comprobará que los valores resultantes de la ejecución de las consultas, son los esperados para esos instantes.

Para la realización de esta prueba se han creado dos tablas: una con datos en permanente cambio y otra, que con ayuda de un disparador, guarda cada modificación de los datos. Para finalizar se comprueba manualmente que los datos y ejecuciones del registro coincidan con los esperados.

También se ha creído oportuno someter el servicio a una prueba de estrés. Esta, se ha llevado a cabo en el equipo donde se ha desarrollado el proyecto, con las siguientes especificaciones: una CPU Intel Core i5-7500 @ 3.4GHz, 8GB de RAM y *Windows 10* como sistema operativo. Para la prueba de estrés, se han creado 1000 consultas de forma aleatoria, de las cuales se ejecutan entre 20 y 60 por ciclo, a bases de datos de tipo MySQL y MicrosoftSQL. Se han incluido consultas con el campo SLEEP, con el objetivo de que la base de datos tarde más de un ciclo en responder y ver cómo reacciona el servicio. Como máximo, la consulta permanecerá dormida un ciclo.

Ya que el servicio crea en paralelo los procesos de ejecución de consultas en cada ciclo y la ejecución de las consultas del ciclo, no se ha detectado ningún retraso en las consultas que debían ejecutarse en menos de un ciclo. Esto es debido a que no se asigna cada ejecución a un hilo fijo, sino que las ejecuciones se van llevando a cabo en hilos independientes, igual que la forma de obtener las conexiones a las bases de datos. Por ello, a pesar de que varias consultas tarden varios ciclos en ejecutarse, las consultas más rápidas se ejecutan sin retraso antes de los 2 segundos del ciclo.

Gracias a esta prueba de estrés, se ha concluido que, con 25 conexiones para cada base de datos, son suficientes para una media de 40 ejecuciones por ciclo. Bajo estas circunstancias, la aplicación, tiene un consumo de CPU que se encuentra por debajo del $0,8\,\%$ en sus picos y que nunca llega a tener más de 60 subprocesos simultáneos.

En la figura 5.1 se pueden visualizar los resultados obtenidos del muestreo del servicio en ejecución, durante 5 minutos [6].

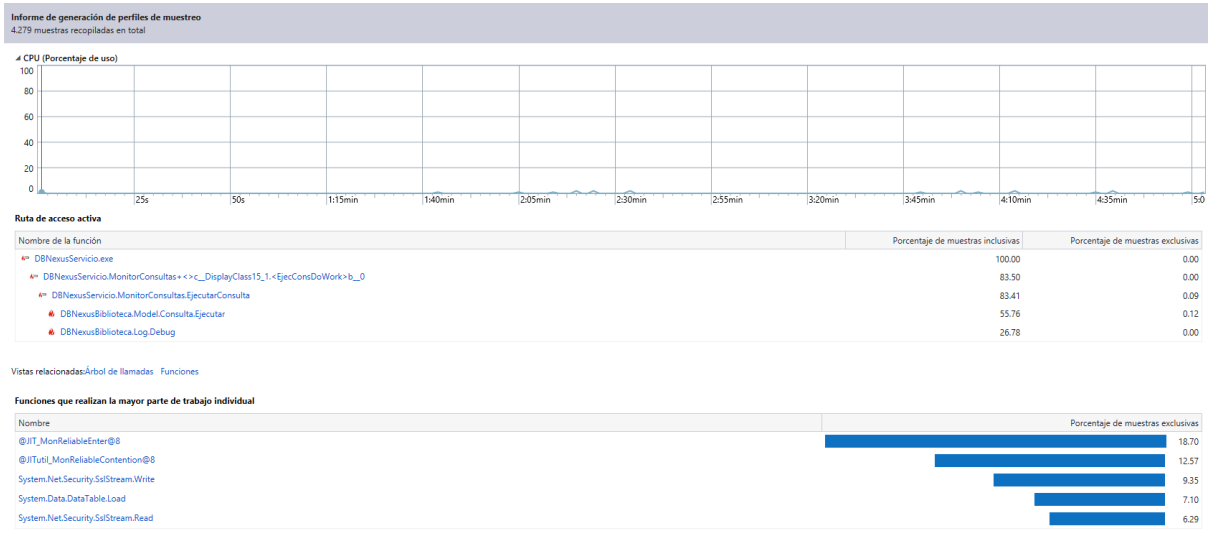

Figura 5.1: Informe generado por el *Visual Studio* resultante del muestreo del servicio en ejecución

Capítulo 6

Conclusiones

Haber realizado este proyecto en un entorno laboral independiente de la universidad, cuando tenía claro, durante su desarrollo, que iba a ser el proyecto para realizar el Trabajo de Fin de Grado, ha supuesto un reto, a la par que una oportunidad de aprendizaje. Previamente a cursar los estudios del grado y durante los mismos, he ido acumulando varias experiencias laborales en diversos ámbitos de la informática. No obstante, esta ha sido la experiencia en la que he tenido la oportunidad de administrar, de forma mucho más autónoma, un proyecto por mí mismo. Gracias a ello y a los estudios realizados, creo haber obtenido un mayor conocimiento sobre cómo plantear y dimensionar proyectos de estas características.

Haber tenido que prever todos los posibles problemas en el desarrollo de la aplicación, ha sido un objetivo ambicioso. Sin embargo, considero que, esto en particular, me ha ayudado a ser más consciente de lo necesaria que es una buena organización. Valorándolo desde cierta perspectiva, hay algunos aspectos durante el desarrollo de la aplicación que, de cara a la realización de futuros proyectos, habría que reconsiderar, a fin de mejorar y no cometer los mismos errores. En el apartado 6.1 de continuidad del proyecto, se comentará alguno.

Por otro lado, en comparación de las expectativas iniciales con los resultados obtenidos, estos las superan considerablemente, de forma positiva. Por las demandas que le urgían a la empresa de recopilar los datos, fue requerido que el proyecto fuera funcional, con agilidad y eficacia, en el menor tiempo posible. Por ello, la primera versión (implantada en producción) del proyecto, fue completada, a falta de muchas mejoras, durante las once primeras semanas de trabajo.

Valorándolo en retrospectiva, cuando inicié el proyecto, la mayor preocupación que tenía en mente era que la aplicación no consumiera excesivos recursos; considero que ese objetivo ha sido realmente alcanzado. Asimismo, cumple los objetivos especificados al inicio del proyecto.

Finalmente, considero oportuno añadir que la experiencia que ha supuesto cursar el Grado de Ingeniería informática, desde un punto de vista personal, ha sido grata. Mayoritariamente he focalizado un interés más notable hacia los campos de la seguridad informática, los sistemas inteligentes y las tecnologías emergentes. Tanto la organización general del plan de estudios, la impartición de algunas materias en inglés, la experiencia de colaborar en una empresa durante la realización de las prácticas y la opción que ofrece la universidad de realizarlas con una beca

Erasmus Plus en el extranjero, constituyen las bases de un programa sólido y actualizado, con el objetivo de formar a futuros trabajadores competentes.

6.1. Continuidad del proyecto

Uno de los planes más ambiciosos a la hora de continuar este proyecto, es el cambio de la interfaz del cliente a un servicio web.

Como mejora más inmediata, se resalta la necesidad de que el servicio tenga un canal de comunicación, con el fin de indicarle cuándo han sido modificadas las consultas y las obtenga nuevamente del almacenamiento.

Otra de las mejoras que se pretende realizar, a nivel global, en todos los proyectos de software de la empresa, es implementar actualizaciones automáticas con la gestión de versiones de TFS.

Como error de sobredimensionamiento del proyecto, se ha determinado que la parte para realizar cálculos con los valores obtenidos antes de almacenarlos en la plataforma, no es necesario que se desarrolle, puesto que estas operaciones se pueden realizar en el código SQL escrito en la aplicación o en la propia plataforma *Nexus Integra*.

Por último, destacar que también queda pendiente finalizar el desarrollo de la historización de un conjunto de valores, con marcas de tiempo, desde las bases de datos.

Bibliografía

[1] Apache Software Foundation. Apache log4net - apachelog4net: Home - apache log4net. https://logging.apache.org/log4net/. [Consulta: 17 de Junio de 2019].

[2] José Luis Del Val Román. Industria 4.0: la transformación digital de la industria. http://coddii.org/wp-content/uploads/2016/10/Informe-CODDII-Industria-4.0.pdf, 2016. [Consulta: 10 de Abril de 2018].

[3] Instituto de Tecnología Cerámica. Sobre itc. http://www.itc.uji.es/sobreITC. [Consulta: 10 de Abril de 2019].

[4] Chih-Chung Lu and Shau-Yin Tseng. Integrated design of aes (advanced encryption standard) encrypter and decrypter. In *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 277–285. IEEE, 2002.

[5] Microsoft. Documentación de .net. https://docs.microsoft.com/es-es/dotnet/. [Consulta: 17 de Junio de 2019].

[6] Microsoft. Documentación de visual studio — microsoft docs. https://docs.microsoft.com/es-es/visualstudio/?view=vs-2017. [Consulta: 17 de Abril de 2019].

[7] Microsoft. Guía de c. https://docs.microsoft.com/es-es/dotnet/csharp/. [Consulta: 17 de Junio de 2019].

[8] Microsoft. Microsoft sql documentation - sql server — microsoft docs. https://docs.microsoft.com/en-us/sql/. [Consulta: 16 de Junio de 2019].

[9] Microsoft. System.collections.concurrent namespace — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.collections.concurrent?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[10] Microsoft. System.componentmodel namespace — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.componentmodel?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[11] Microsoft. System.data.sqlclient namespace — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.data.sqlclient?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[12] Microsoft. System.servicemodel namespace — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.servicemodel?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[13] Microsoft. System.serviceprocess namespace — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.serviceprocess?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[14] Microsoft. System.threading namespace — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.threading?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[15] Microsoft. System.windows.forms namespace — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.windows.forms?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[16] Microsoft. Task class (system.threading.tasks) — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.threading.tasks.task?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[17] Microsoft. Timer class (system.timers) — microsoft docs. https://docs.microsoft.com/es-es/dotnet/api/system.timers.timer?view=netframework-4.5. [Consulta: 17 de Junio de 2019].

[18] Microsoft. What is nuget and what does it do? — microsoft docs. https://docs.microsoft.com/en-us/nuget/what-is-nuget. [Consulta: 17 de Junio de 2019].

[19] Oracle Corporation. Apache log4net - apachelog4net: Home - apache log4net. https://dev.mysql.com/doc/connector-net/en/connector-net-tutorials-sql-command.html. [Consulta: 17 de Junio de 2019].

[20] Oracle Corporation. Database sql reference - contents. https://docs.oracle.com/cd/B19306_01/server.102/b14200/toc.htm. [Consulta: 16 de Junio de 2019].

[21] Oracle Corporationâ. Mysql :: Mysql documentation. https://dev.mysql.com/doc/. [Consulta: 16 de Junio de 2019].

[22] Oracle Corporation. Using odp.net client provider in a simple application. https://docs.oracle.com/database/121/ODPNT/intro005.htm#ODPNT148. [Consulta: 17 de Junio de 2019].

[23] Ronald Rivest. The md5 message-digest algorithm. Technical report, 1992.

Anexo A

Descripción detallada del proyecto

A.1. DBNexusBiblioteca

En esta sección se detallará todas las clases del proyecto biblioteca que se incluirá en los otros dos proyectos.

En la figura A.1 podemos observar un diagrama minimizado de todas las clases del proyecto DBNexusBiblioteca.

Clase *Constantes*

En esta clase estática se almacenarán parámetros de funcionamiento para las otras dos aplicaciones.

Algunos de los parámetros obtenidos se extraen directamente del fichero de configuración del proyecto, situado en su misma carpeta, denominado App.config. En este fichero con formato XML 1.0 se define una sección *appSetings* donde se encuentran los valores de algunas de las variables que se pueden obtener utilizando el comando de C#:

ConfigurationManager.AppSettings["nombre_de_la_variable"].toString()

Otras serán obtenidas por el constructor estático de esta clase en el momento de iniciar la aplicación.

Las variables que contiene la clase son:

- *TimeStamp*: Cadena de caracteres que identifica una columna resultante de una *Consulta* como la marca de tiempo.
- *IdentificadorVariable*: Cadena de caracteres que identifica una columna de una *Consulta* como identificador de variable.

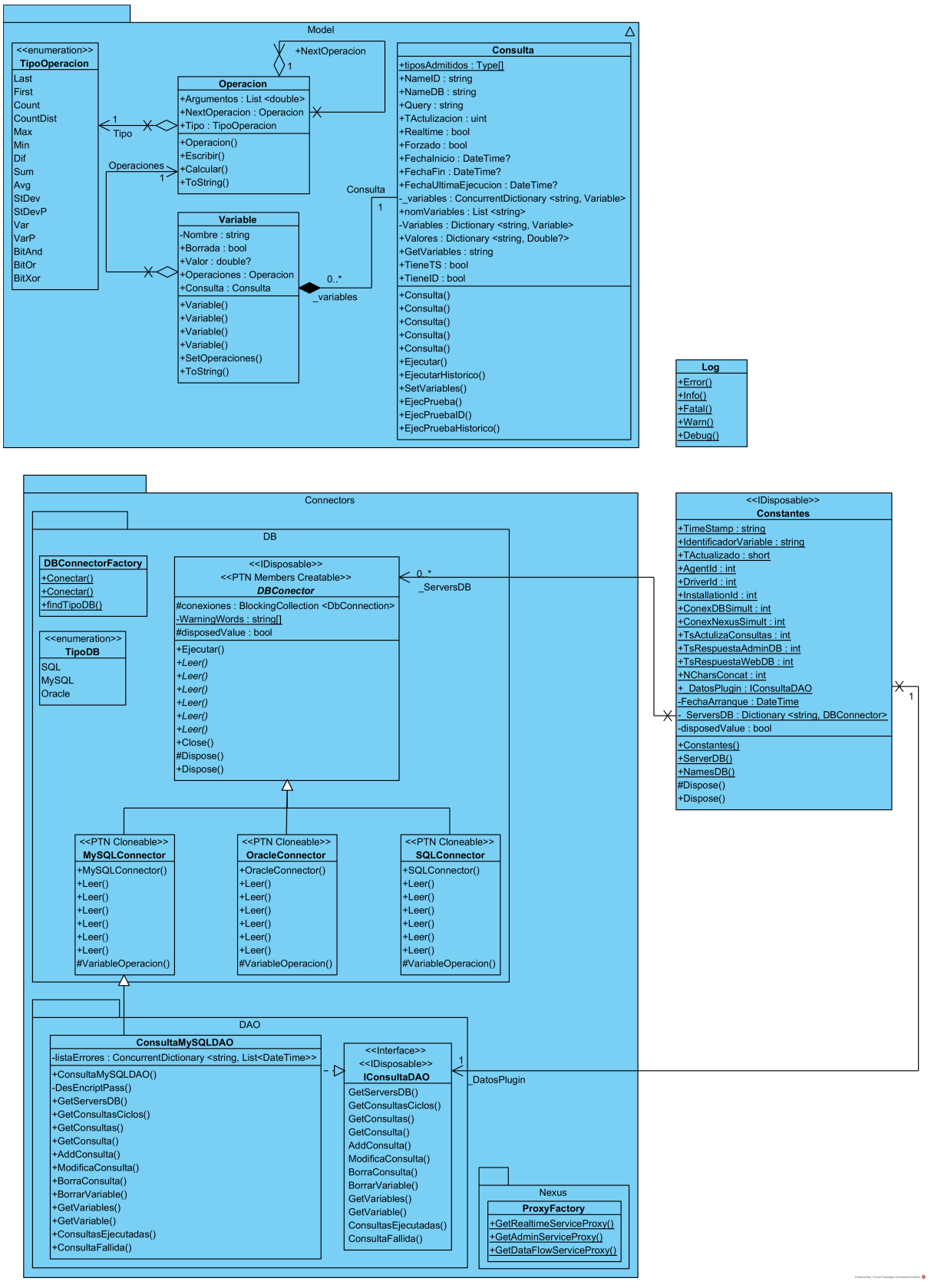

Figura A.1: Diagrama de Clases Minimizado de la Biblioteca

- *TActualizado*: Número entero que indica el tiempo(seg) mínimo entre monitorizaciones en cada *Consulta*.
- *AgenteId*: Número entero que indica el identificador del agente de *Nexus Integra* donde almacenar los valores.
- *DriverId*: Número entero que indica el identificador del driver de *Nexus Integra* con el que almacenar los valores.
- *InstallationId*: Número entero que indica el identificador de la instalación de *Nexus Integra* donde almacenar los valores.
- *ConexionDBSimult*: Número entero que indica el número máximo de conexiones simultáneas con cada base de datos.
- *ConexNexusSimult*: Número entero que indica el número máximo de conexiones simultáneas con el servicio de sistema y monitorización a tiempo real e histórico de *Nexus Integra*.
- *TsActulizaConsultas*: Número entero que indica el intervalo de tiempo(seg) entre recargas de las *Consultas* a monitorizar por el servicio.
- *TsRespuestaAdminDB*: Número entero que indica el intervalo de tiempo(seg), que la aplicación se mantendrá esperando la respuesta del servidor de almacenamiento de consultas, para funciones de administración.
- *TsRespuestaWebDB*: Número entero que indica el intervalo de tiempo(seg) que la aplicación se mantendrá esperando la respuesta del servidor de almacenamiento de consultas, para funciones web. Esta variable, actualmente, se encuentra en desuso y se utilizará cuando se realice la implementación web como tiempo máximo de respuesta para las peticiones de la web.
- *NCharsConcat*: Número entero que indica el número de caracteres por columna máximo, necesario para el servidor de datos en el que se almacenan las consultas. Esta variable es necesaria porque, cuando se obtiene una consulta en el servidor de almacenamiento, también devuelve en una columna (tipo cadena de caracteres) sus variables separadas por comas. Se utiliza esto en sustitución a un tipo Array, ya que, en el SGDB MySQL, no existen; ello nos permitiría cadenas de datos más grandes. Tras haber realizado unas pruebas, se ha concluido que, con 40960 caracteres son suficientes y, ya que esta configuración únicamentese realiza en las conexiones de almacenamiento de la aplicación, no supondrá una sobrecarga excesiva para la base de datos ni para el tamaño de la comunicación.
- *FechaEpoch*: Marca de tiempo en el día *01/01/1970*, valor muy utilizado para calcular el ciclo en el que se ejecuta cada *Consulta*.
- *_DatosPlugin*: Objeto utilizado para almacenar/recuperar cada *Consulta* y *Variable* que el plugin ejecutará.
- *FechaArranque*: Almacena la fecha de arranque del servicio. Más adelante, se utilizarápara saber si una consulta que no se está ejecutando correctamente esta fallando desde su última ejecución o desde que se inició el servicio; para detenerla cuando lleve cierto tiempo de ejecución, ejecutándose erróneamente.

- *_ServersDB*: Esta variable contendrá un diccionario con un nombre identificador como clave y, como valor, una clase heredada de la clase abstracta *Connectors.DB.DBConnector*, que contendrá los conectores a la base de datos identificada.

Cuando se ejecuta cualquier aplicación que contenga esta biblioteca, al ser esta una clase estática con un constructor estático, se construirá antes de lanzar el hilo principal de la aplicación. En este constructor se utilizará la conexión con la base de datos de almacenamiento de la variable *_DatosPlugin*, para obtener las cadenas de conexión, así como los usuarios y las contraseñas de las bases de datos a las que se va a conectar y almacenarlas en el diccionario privado *_ServersDB*, ya como conectores, junto con su nombre de conexión.

Para obtener los conectores almacenados en el diccionario *_ServersDB*, se llamará a la función *ServersDB(nameDB : String)* indicando como parámetro del nombre identificador de la conexión, cosa que devolverá una **copia de la conexión del diccionario**.

Para obtener la lista de los identificadores disponibles se utilizará la función *NamesDB()* que devolverá la lista de las claves del diccionario *_ServersDB*.

Se puede visualizar el diagrama de esta clase en la figura B.3 situada en el Anexo B.

Clase *Log*

Esta es un clase estática con métodos encargados de registrar los mensajes de la aplicación y, a su vez, registrarlos en la biblioteca log4net con diferente nivel de alerta. En el apartado *log4net* del fichero de configuración App.config se encuentra la definición de cómo tratar los diferentes niveles de mensaje que se emiten desde la aplicación.

Se utiliza esta biblioteca porqué el sistema *Nexus Integra* también la incorpora, por tanto, sería más sencillo en caso de realizar un registro de eventos común. También cabe destacar la versatilidad de configuraciones que se pueden realizar con esta biblioteca. Esta biblioteca permite tratar cada tipo de alerta (informacion, debug, alerta, error,...) de una forma diferente, pudiendo: almacenarse en un fichero, escribirlo por consola o enviarlo a un servidor de registros.

Para facilitar el uso de esta biblioteca se definen cinco funciones estáticas para el registro de eventos, que únicamente tendrán como parámetro el mensaje emitido por el evento. Las funciones son:

- *Error*: Método para indicar un mensaje de error en el registro.
- *Info*: Método para indicar un mensaje informativo en el registro.
- *Fatal*: Método para indicar un mensaje de error fatal en el registro.
- *Debug*: Método para indicar un mensaje de desarrollo en el registro.
- *Warn*: Método para indicar un mensaje de alerta en el registro.

El diagrama de esta clase se puede visualizar en la figura B.4 situada en el Anexo B.

Paquete *Modelo*

El paquete *Modelo* contendrá los modelos de las consultas y variables utilizados por la aplicación.

Para tener una idea general del paquete *Modelo* se puede visualizar su diagrama de clases en la figura A.2.

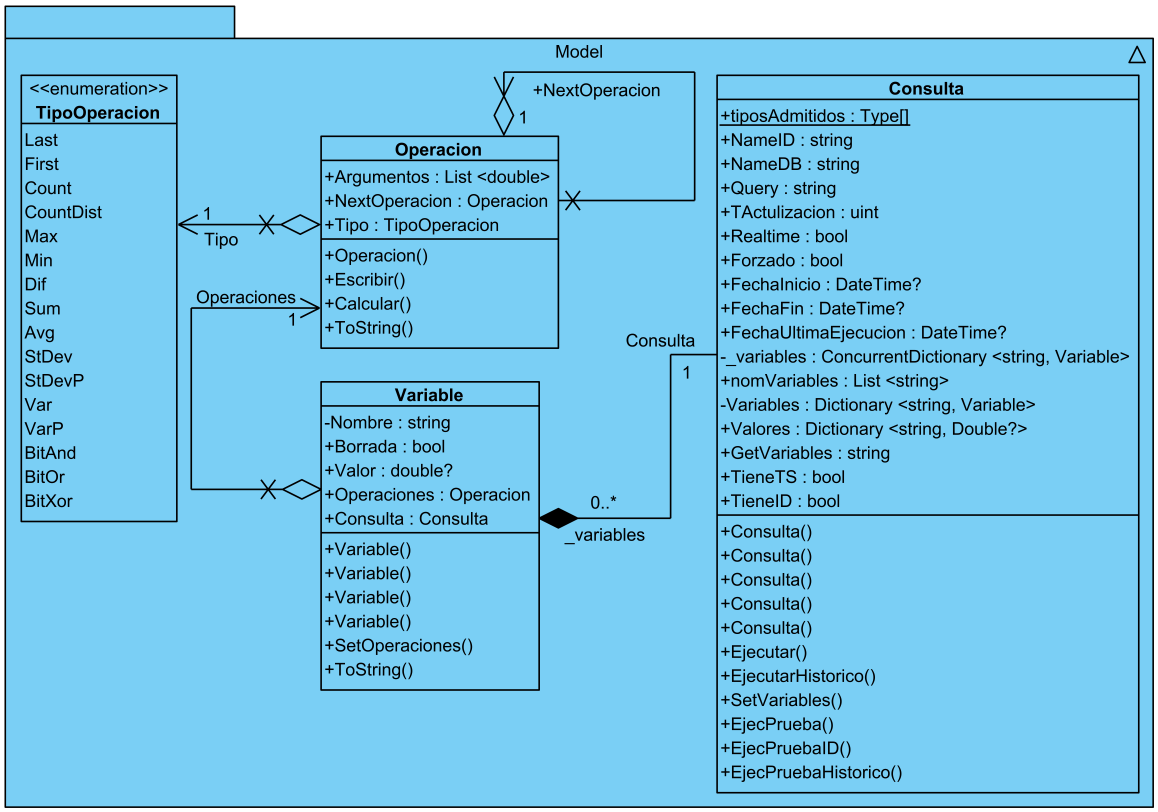

Figura A.2: Diagrama de Clases Minimizado del paquete Modelo de la Biblioteca

Clase *Modelo.Consulta*

La clase Consulta forma parte del Modelo del programa y se utiliza para gestionar y almacenar todos los valores y funcionalidades que puede tener cada consulta de las que se quiera tratar, para monitorizar sus valores o cualquier otra opción de las que permite la aplicación.

Los parámetros que se necesitan para definir una consulta son:

- *tiposAdmitidos*: Es una colección estática y de sólo lectura, que contiene los tipos de datos que puede devolver una consulta SQL, ya que el sistema *Nexus Integra* únicamente permite almacenar *Doubles* y la devolución de las consultas sólamente deben permitir valores

transformables a *Double* para almacenar en *Nexus Integra*. Los tipos son: *Boolean*, *Byte*, *Char*, *DateTime*, *Decimal*, *Double*, *Int16*, *Int32*, *Int64*, *SByte*, *Single*, *UInt16*, *UInt32* y *UInt64*.

- *NameID*: Es en nombre identificador con el que se almacena la consulta.
- *NameDB*: Es el nombre del identificador de la base de datos donde se ejecutará la consulta.
- *Query*: Es la cadena de la consulta de la que se desea leer los datos, escrita en SQL.
- *TActulizacion*: Es el periodo de tiempo en segundos que indica cada cuanto se almacenan los datos de la consulta en *Nexus Integra*. Este valor debe ser múltiplo de *Constantes.TActualizado*, ya que el ejecutor de consultas se ejecutará con esa frecuencia.
- *Realtime*: Este booleano indica si esta consulta debe ser almacenada en el RealTime de *Nexus Integra*. En caso de *true*, cuando se almacene la consulta en tiempo real también se almacenará en el servicio de RealTime de *Nexus Integra*. En caso de *false* nunca se almacena el valor de la consulta en el RealTime de *Nexus Integra*.
- *Forzado*: Este booleano indica si la consulta debe almacenarse periódicamente en el tiempo indicado en *Consulta.TActulizacion*. En caso de *true* la consulta se almacena sólo bajo demanda. En caso de *false* la consulta se ejecuta periódicamente cada *Consulta.TActulizacion* segundos.
- *FechaInicio*: Fecha desde la que se inicia la consulta, sirve tanto para indicar cuándo se desea que inicie una consulta, como para indicar a qué hora se desea que inicie una consulta periódica. En caso de introducir *null*, significa que no hay fecha de inicio.
- *FechaFin*: Fecha a partir de la cual no se almacenará la consulta. En caso de introducir *null* significa que no hay fecha de finalización.
- *FechaUltimaEjecucion*: Fecha de la última vez que la consulta fue ejecutada. En caso de obtener *null* significa que no ha sido ejecutada.
- *_variables*: Diccionario donde se almacenan las variables con el tipo *Variable* que componen la consulta con su nombre identificador como clave.
- *Variables*: Copia del diccionario donde se almacenan las variables de el tipo *Variable* de las que se compone la consulta, con su nombre identificador como clave.
- *Valores*: Diccionario con los nombres de las variables de la consulta que no han sido borradas y su valor.
- *GetVariables*: Devuelve un *String* con las variables y sus atributos concatenados, en el formato necesario para almacenarlo en la base de datos de almacenamiento de las consultas.
- *TieneTS*: Indica si la consulta tiene alguna columna con el identificador de marca de tiempo.
- *TieneID*: Indica si la consulta tiene alguna columna con el identificador de marca de identificador de variable, se utiliza para saber si la consulta está parametrizada.

Para esta clase se han definido cinco constructores para diferentes necesidades:

- *Consulta()*: Constructor vacío donde se inicializan los valores que no dependen del usuario y los diccionarios.

- *Consulta(string nameID, string nameDB, string query, uint tActulizacion = 60, bool realtime = true, bool forzado = false, DateTime? fechaInicio = null, DateTime? fechaFin = null)*: Constructor de la clase *Consulta*, utilizado para almacenar los datos de una consulta antes de ser comprobada o almacenada. Todos los parámetros de esta función son los necesarios para definir una nueva consulta por el usuario; algunos de estos parámetros también tienen, como valores por defecto, los más comunes (como *fechaInicio* y *fechaFin* a *null*).

- *Consulta(string nameID, string nameDB, string query, uint tActulizacion, bool realtime, bool forzado, DateTime? fechaInicio, DateTime? fechaFin, DateTime? fechaUltimaEjecucion, bool tieneTS, bool tieneID, Dictionary<string, Variable> variables)*: Constructor utilizado para almacenar los datos de una consulta obtenida de la base de datos donde se almacenan y, que aparte de los parámetros que definen la consulta, están los parámetros de control: fechaUltimaEjecucion, tieneTS, tieneID y variables.

- *Consulta(Consulta con)*: Constructor de la clase *Consulta*, utilizado para copiar una consulta de otra.

- *Consulta(Consulta con, string nameDB, string query, uint tActulizacion, bool realtime, bool forzado, DateTime? fechaInicio, DateTime? fechaFin, bool tieneTS = false, bool tieneID = false)*: Constructor de la clase *Consulta*, utilizado para copiar una consulta de otra realizando cambios sobre esta.

Los métodos definidos para esta clase serán:

- *Dictionary<string, Double?> Ejecutar(out bool resultado, out string error)*: Método empleado para ejecutar la consulta que acumula los valores en las respectivas variables tipo Variable en _variables. Además de devolver un diccionario con el nombre de la variable como clave y como valor, el valor de esta variable en la consulta de tipo *Double?*, también tiene dos parámetros de salida donde indica si ha podido obtener resultado y la cadena de error en caso de que ocurra un error.

- *void SetVariables(List<Variable> variables)*: Método usado para añadir una lista de variables tipo *Variable* a la consulta.

- *DataTable EjecPrueba(out Dictionary<string, bool> variables, out bool TieneTS, out bool TieneID)*: Método utilizadopara probar la consulta. Devuelve la tabla producida por la ejecución de la consulta para poder ver un ejemplo de salida, además nos proporciona más información con sus parámetros de salida, como: un boleano que indica si la consulta tiene una columna marcada como marca de tiempo, un boleano que indica si la consulta tiene una columna marcada como identificador y un diccionario que contiene como clave los nombres de las variables de la consulta y, como valor contiene *True* para las variables utilizadas en la ejecución y *False* en caso contrario. Asimismo, en caso de que TieneTS o TieneID sea cierto también ejecuta las pruebas de su buen funcionamiento (EjecPruebaID y EjecPruebaHistorico).

- *void EjecPruebaID(DataTable data)*: Método que se utiliza para probar que la consulta, almacenada en este objeto tipo *Consulta*, es válida cuando tiene columnas de identificación. Las columnas de identificación son parámetros que se encolarán al tag de la variable de identificación. Este método no ejecuta la consulta ya que se le pasa la tabla de datos de la ejecución resultante de EjecPrueba.
- *Dictionary<DateTime,Dictionary<string, Double?>> EjecutarHistorico(DateTime? fI = null, DateTime? fF = null)*: Método que se utiliza para ejecutar la consulta de forma histórica. Aquellas consultas que tengan una marca de tiempo definida con *Constantes.TimeStamp* pueden utilizar este método para devolver resultados de consultas agrupándolos en periodos de tiempo. Para agrupar los distintos valores se hará uso dela operación definida en sus *Variable*. Este método pertenece a una funcionalidad que se planteó en el diseño, pero que no se implementará hasta futuras versiones.

Podemos ver el diagrama de clases en la figura B.5 situada en el Anexo B.

Clase *Modelo.Variable*

La clase Variable forma parte del Modelo del programa; sirvepara almacenar todos los valores y funcionalidades de cada uno de los valores de la consulta.

Sus parámetros son:

- *Nombre*: Nombre del tag de la variable.
- *Borrada*: Define si la variable ha pertenecido a esta consulta, pero no está en uso en este momento.
- *Operaciones*: Define las operaciones que se realizarán sobre la variable, definidas como clase *Operacion*.
- *Valor*: Almacena el valor de la variable la última vez que se realizó la consulta.
- *Consulta*: Almacena la consulta a la que pertenece la variable para poder recuperar la consulta en base a una de sus variables.

Para inicializar la clase se han definido los siguientes constructores:

- *Variable()*: Constructor vacío para utilizar si se necesita crear una variable y añadirle los valores a los parámetros más adelante.
- *Variable(string nombre, bool borrada = false, string operaciones = null)*: constructor de la variable con sus parámetros donde no se indica la consulta a la que pertenece.
- *Variable(Consulta consulta, string nombre, bool borrada = false, string operaciones = "")*: constructor de la variable donde se le indica la consulta a la que pertenece, además de sus parámetros.

- *Variable(Consulta consulta, Variable var)*: constructor de la variable donde se indica la consulta a la que pertenece y se extraen los valores de los parámetros de la variable facilitada.

Esta clase también incluye otros dos métodos para trabajar con ella:

- *new string ToString()*: Cadena que define la variable actual y sus parámetros, utilizada para almacenar la definición de la variable en el sistema de almacenamiento; también se lee en este formato.
- *void SetOperaciones (string operaciones)*: Método que se utilizapara definir las operaciones en una variable mediante una cadena que define la operación.

Se puede visualizar el diagrama de clases en la figura B.6 situada en el Anexo B.

Clase *Modelo.Operacion*

Esta clase es creada por una petición posterior, en la quese solicitaba que se pudieran operar los valores obtenidos, antes de almacenarse en el sistema *Nexus Integra*. Pese a que no se ha acabado de implementar para este proyecto, aun así se explica su funcionamiento.

Esta clase se le asignará a cada clase *Variable* que se desee operar. Asimismo, se puede anidar en ella misma para asignar múltiples operaciones a una única variable. En esta clase también se asignará el tipo de operación que estará definida en la enumeración *Modelo.TipoOperacion* y los parámetros que se utilizan para operar, pudiendo ser valores de otras variables o valores estáticos.

Las variables de esta clase son:

- *TipoOperacion Tipo*: Valor del enumerado *Modelo.TipoOperacion* que indica que operación realizará.
- *List<Double> Argumentos*: Lista de argumentos a aplicar a la operación (varía su numero dependiendo de la operación).
- *Operacion NextOperacion*: Siguiente operación anidada, hay que tener en cuenta que las operaciones se ejecutan antes de pasar el valor a la siguiente operación, por lo que la primera operación en calcularse será la que se encuentra en la *Variable*.

Para esta clase se tendrá un único constructor que recibirá un string, en el que estarán definidas la propia operación y todas las que se encuentren anidadas dentro de la misma.

Como operaciones para esta clase se definenlas siguientes:

- *string Escribir(Func<string, TipoOperacion, string> func, string cadena)*: Método que devuelve una cadena de caracteres que define la operación actual con sus argumentos y sus anidadas.
- *double[] Calcular(double[] v)*: Método que aplica las operaciones definidas a un conjunto de valores y retorna el resultado.
- *new string ToString()*: Método que devuelve una cadena de caracteres que define a la operación actual.

Puede visualizarse el diagrama de clases en la figura B.7 situada en el Anexo B.

Enumeración *Modelo.TipoOperacion*

Este enumerado es utilizado para indicar el tipo de operación que se aplicará en cada operación, cosa que afectará al valor o valores de la *Variable*. La lista de operaciones que se plantean inicialmente son: *Last*, *First*, *Count*, *CountDist*, *Max*, *Min*, *Dif*, *Sum*, *Avg*, *StDev*, *StDevP*, *Var*, *VarP*, *BitAnd*, *BitOr* y *BitXor*.

Puede consultarse el diagrama de clases en la figura B.8 situada en el Anexo B.

Paquete *Connectors*

Con el fin de diferenciar los tipos de conexiones, se crean 3 sub paquetes:

- *DB*: Donde se incluirán los conectores para conectarse a los diferentes tipos de SGDB. Estos conectores se utilizarán tanto para ejecutar las *Consultas* como para almacenarlas.
- *DAO*: Aquí se encuentran las clases utilizadas para almacenar las *Consultas* y *Variables* en el servicio MySQL, para conectarse a este servicio de almacenamiento se utilizarán los conectores del paquete *Connectors.DB*.
- *Nexus*: En este paquete se incluirá la clase proporcionada por los técnicos de *Nexus Integra* para la creación y obtención de los identificadores de los tags y también para almacenar los valores obtenidos en sus diferentes servicios (realtime y historico).

Para hacerse a una idea general del paquete Connectors se puede visualizarsu diagrama de clases en la figura A.3.

Clase *Connectors.DB.DBConnectorFactory* y *Connectors.DB.DBConnectorFactory.TipoDB*

La clase *Connectors.DB.DBConnectorFactory* implementa una fabrica encargada de proveer el conector que corresponda según los datos proporcionados. Y todas sus funciones son estáticas.

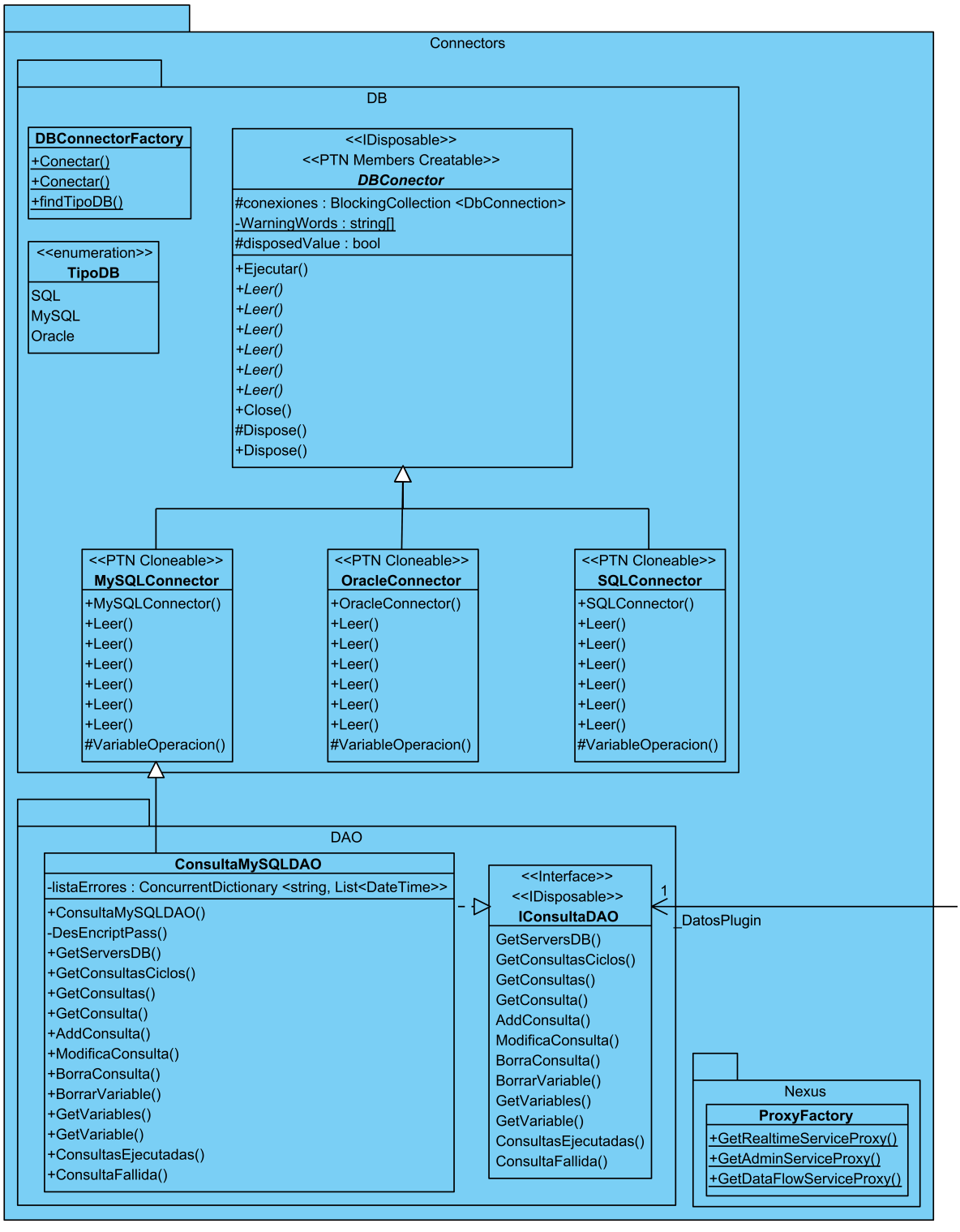

Figura A.3: Diagrama de Clases Minimizado del paquete Connectors de la Biblioteca

Para devolver los conectores utilizatres funciones, las cuales se describen a continuación:

- *DBConnector Conectar(TipoDB tipo, string conn, string userId = null, SecureString password = null)*: Encargada de seleccionar el tipo de SGBD entre los del enumerado *TipoDB* y, a su vez, de pasar el resto de parámetros al constructor de las clases *SQLConnector*, *MySQLConnector* y *OracleConnector* que corresponda y retornar el conector.
- *DBConnector Conectar(string tipo, string conn, string userId, SecureString password)*: Esta función es muy similar a la anterior pero, en este caso, el *tipo* es una cadena de caracteres que pasará por la función *findTipoDB* donde se seleccionará el *TipoDB* que le corresponde y llamará a la primera función (*Conectar*) para retornar el conector.
- *TipoDB findTipoDB(string tipo)*: Función encargada de seleccionar qué valor del enumerado *TipoDB* coincide con la cadena de valores para saber cual es el conector que se debe utilizar.

Los valores del enumerado *TipoDB* son los tipos de SGDB soportados por la aplicación y, en caso de incluir otro conector, habría que añadirlo a esta fábrica. Los tipos de SGBD son: *SQL*, *MySQL* y *Oracle*.

Se puede observar el diagrama de clases de la clase *DBConnectorFactory* en la figura B.9 y del enumerado *TipoDB* en la figura B.10, ambas situadas en el Anexo B.

Clase *Connectors.DB.DBConnector*

Esta clase abstracta define los métodos que van a tener que implementar todos los conectores para establecer las conexiones y realizar las diferentes funciones que requiere la aplicación.

Los parámetros comunes para todos los conectores que hereden de esta clase son los siguientes:

- *BlockingCollection<DbConnection> conexiones*: Cada conector tendrá una colección concurrente de sólo lectura, donde almacenará para cada *DbConnection* un número de copias de sí mismo, igual a *Constantes.ConexDBSimult*. Esta colección necesita ser creada porqué se requiere que, cuando se ejecuten las consultas, se realicen en paralelo y no se puede utilizar el mismo objeto *DbConnection* simultáneamente por más de un hilo, ya que aparecen problemas de concurrencia. Por ello, se crea una mochila que se llenará de conectores y cada vez que se necesite realizar una consulta, se extraerá un conector de la mochila. Una vez se acabe de utilizar el conector, se devolverá a la mochila. También se limitará el número de conexiones simultáneas con cada servidor para no sobrecargarlos. El número de conexiones simultáneas con cada base de datos se especifica el el fichero App.config.
- *string[] WarningWords*: Este vector de cadenas de caracteres estático y de sólo lectura, almacena cadenas de caracteres que no pueden estar en las consultas que va a ejecutar la aplicación. Se utilizará como medida de seguridad extra para que las consultas de esta aplicación no puedan editar las bases de datos a las que se conectan.

En esta clase hay dos métodos ya definidos para que sean heredados por los hijos de esta clase:

- *DataTable Ejecutar(out bool resultado, string query, Dictionary<string, Variable> variables = null, long? tAct = null, DateTime? fI = null, DateTime? fF = null, int timeout = 5)*: Método utilizado por los objetos de la clase *Consulta* para ejecutar las consultas en los conectores. Este método comprueba que la consulta no contenga ninguna cadena de caracteres prohibida, que la conexión se cierre al cerrar el objeto *DateTable* que devuelve y que el timeout de la consulta sean 5 segundos. Este método empleauno de los métodos abstractos que serán definidos posteriormente para ejecutar la sentencia en la base de datos.

- *string VariableOperacion(string var, Variable.Operacion.TipoOperacion op)*: Es un método no implementado que, junto con la clase *Modelo.Operacion*, permite aplicar una función a una variable de una consulta, antes de ejecutarse (ya que en operaciones como la suma de todos los elementos, hallar el máximo o el mínimo se puede delegar a la base de datos).

- *void Close()*: Método empleado para cerrar todas las conexiones almacenadas en la colección *conexiones*.

Asimismo, también se definen los métodos abstractos que se van a necesitar implementar en todos los conectores heredados:

- *DataTable Leer(out bool resultado, string query, Dictionary<string, Variable> variables, long tAct, DateTime? fI = null, DateTime? fF = null, CommandBehavior cb = CommandBehavior.CloseConnection, int timeout = 5)*: Tal y como sugiere su nombre, esta función y las del mismo nombre que la siguen, son utilizadas para ejecutar una sentencia en la base de datos de la conexión. Los parámetros se explican a continuación.

- *DataTable Leer(out bool resultado, string query, CommandBehavior cb = CommandBehavior.CloseConnection, int timeout = 5)*

- *DataTable Leer(out bool resultado, DbCommand select, CommandBehavior cb = CommandBehavior.CloseConnection, int timeout = 5)*

- *DataTable Leer(string query, Dictionary¡<string, Variable> variables, long tAct, DateTime? fI = null, DateTime? fF = null, CommandBehavior cb = CommandBehavior.CloseConnection, int timeout = 5)*

- *DataTable Leer(string query, CommandBehavior cb = CommandBehavior.CloseConnection, int timeout = 5)*

- *DataTable Leer(DbCommand select, CommandBehavior cb = CommandBehavior. CloseConnection, int timeout = 5)*

- *DataTable Leer(DbCommand select, CommandBehavior cb = CommandBehavior. CloseConnection, int timeout = 5)*

Entre los métodos anteriores, existen dos que no se utilizarán en la actual implementación del proyecto, puesto que se prevéen usar para la historización de datos almacenados en la base de datos y no de datos nuevos que se generan. Los métodos *DataTable Leer(out bool resultado, string query, Dictionary<string, Variable> variables, long tAct, DateTime? fI = null, DateTime? fF = null, CommandBehavior cb = CommandBehavior.CloseConnection, int timeout = 5)* y *DataTable Leer(string query, Dictionary¡<string, Variable> variables, long tAct, DateTime? fI = null, DateTime? fF = null, CommandBehavior cb = CommandBehavior.CloseConnection, int timeout = 5)* permiten obtener más de una fila por consulta; para ello se especifica qué columna indica la marca de tiempo de su fila. Esta marca de tiempo se define llamando a la columna como se ha indicado en *Constantes.TimeStamp* (en esta aplicación es "*TS*"por defecto).

Todas las funciones *Leer* retornan un objeto tipo *DataTable* que contiene, en formato tabla, la respuesta de la base de datos a la sentencia ejecutada. Los parámetros utilizadospor las funciones son prácticamente los mismos en todas las funciones, por lo que se explican todos juntos a continuación:

- *out bool resultado*: Es un valor boleano de salida que indica si la sentencia se ha ejecutado con éxito o si ha fallado por cualquier razón.
- *string query*: La cadena de caracteres que se ejecutará como sentencia en la base de datos.
- *DbCommand select*: Objeto que contiene una instrucción SQL o un procedimiento almacenado en la base de datos para ser ejecutado.
- *CommandBehavior cb*: Objeto que proporciona una descripción de los resultados esperados de la consulta y sus efectos en la base de datos. Por defecto, se le asigna *CommandBehavior.CloseConnection*, el cual cierra la conexión automáticamente cuando se cierra el objeto *DataReader* devuelto por la consulta.
- *int timeout*: Indica el tiempo que espera el servicio a obtener la respuesta de la base de datos. Por defecto es 5 segundos, dado que es la frecuencia mínima de ejecución de consultas, no obstante, este tiempo se adapta a cada consulta y es igual a la frecuencia de ejecución de la consulta; de este modo, una consulta que tarde más tiempo en responder que su frecuencia, no se acumulará creando un cuello de botella, porque se descartará como error.
- *int tAct*: Cuando se va a realizar la historización de diferentes resultados en el tiempo; el valor de *tAct* indica la frecuencia o tiempo de muestreo con que se historizan los resultados.
- *DateTime? fI*: Cuando se va a realizar la historización de una lista de resultados en el tiempo; el valor de *fI* indica el instante desde la que se historizan los resultados, si este valor es nulo se almacenarán todos los valores anteriores al instante de finalización indicado.
- *DateTime? fF*: Cuando se va a realizar la historización de diferentes resultados en el tiempo; el valor de *fF* indica el instante hasta el que se historizan los resultados; si este valor es nulo, se almacenarán todos los valores posteriores al instante de inicio indicado.

Se puede visualizar el diagrama de clases de la clase *Connectors.DB.DBConnector* en la figura B.11 situada en el Anexo B.

Clases *Connectors.DB.MySQLConnector*, *Connectors.DB.SQLConnector* y *Connectors.DB.OracleConnector*

Estas clases son heredadas de la clase *Connectors.DB.DBConnector* e implementan los métodos abstractos, anteriormente mencionados, con las particularidades de cada clase de conectores. En estas clases también se implementan los constructores a los que se les facilitan los siguientes parámetros:

- *string conn*: Cadena de caracteres utilizadapara conectarse a la base de datos.
- *string userId*: Cadena de caracteres con el nombre de usuario necesario para conectarse a la base de datos.
- *SecureString password*: Cadena de caracteres encriptada que contiene la contraseña del usuario que se utilizarápara realizar la conexión.

Así mismo, los constructores no crean una única conexión con la base de datos, sino que crean y almacenan tantas como quepan en su colección *conexiones*, cuya capacidad es igual a el número de conexiones máximas con cada base de datos, indicado en la variable *Contantes.ConexDBSimult*.

Se pueden consultarr los diagramas de clases, de las clases *Connectors.DB.MySQLConnector*, *Connectors.DB.SQLConnector* y *Connectors.DB.OracleConnector* en las figuras B.12, B.12 y B.13 sucesivamente situadas en el Anexo B.

Interfaz *Connectors.DAO.IConsultaDAO*

Esta interfaz declara todos los métodos necesarios para que la aplicación pueda gestionar la obtención, el almacenamiento y la administración de consultas, variables y conexiones a bases de datos. Dicha interfaz se crea para facilitar el cambio de base de datos, en caso de que fuese necesaria en un futuro.

Para ello, declara los siguientes métodos que deberán ser implementados en un conector de *MySQL* ya que, en este proyecto, la base de datos donde se almacenarán los datos es del tipo *MySQL*:

- *Dictionary<string, DBConnector> GetServersDB(string dBPass, int timeout = 30)*: Método utilizado por el constructor de la clase *Constantes* para obtener el diccionario de conectores almacenado en *_ServersDB*. Como parámetros necesita *dBPass*, la cadena con la que están codificadas las contraseñas en la base de datos.
- *Dictionary<string, Consulta> GetConsultas(out Dictionary<string, Variable> variables, bool soloRealtime = false, uint ciclo = 0, uint numPag = 0, uint? cantRow = null, List<string> orden = null, int timeout = 30)*: Método utilizado por el cliente para obtener la lista de consultas, pudiendo ser filtrada, ordenada y paginada con los parámetros del método. También devuelve un diccionario con las variables de todas las consultas devueltas por el método.

- *Consulta GetConsulta(string nameID, int timeout = 30)*: Método que obtiene una consulta mediante su identificador; si el tiempo de espera supera el del parámetro *timeout* en segundos, el método falla y devuelve *null*.

- *bool AddConsulta(Consulta consulta, out string error, int timeout = 30)*: Este método almacena una consulta en la base de datos y retorna verdadero en caso de que se almacene con éxito, en caso contrario, devuelve el error en el parámetro de salida *error*. Si el tiempo de espera supera el del parámetro *timeout* en segundos, el método falla y devuelve falso.

- *bool ModificaConsulta(Consulta consulta, out string error, int timeout = 30)*: Este método modifica una consulta almacenada en la base de datos y retorna verdadero en caso de que se modifique con éxito, en caso contrario, devuelve el error en el parámetro de salida *error*. Si el tiempo de espera supera el del parámetro *timeout* en segundos, el método falla y devuelve falso.

- *bool BorraConsulta(string nameIDConsulta, out string error, bool borrarVar = false, bool borrarNexus = false, int timeout = 30);*: Método que borra una consulta del almacenamiento y permite también borrar las variable asociadas a la consulta, del almacenamiento y del servicio *Nexus Integra* indistintamente. Retorna verdadero si ha tenido éxito, y falso, indicando el error en el parámetro *error*, en caso de que se de algún error.

- *bool BorrarVariable(List<string> nameTags, out string error, bool borrarNexus = false, int timeout = 30)*: Método que permite borrar una variable que no tenga consulta asociada o no se encuentre en uso en su consulta asociada; también permite indicar si se borrará también del servicio *Nexus Integra*.

- *Dictionary<string, Variable> GetVariables(string nomConsulta = null, bool noBorrada = false, int timeout = 30)*: Método que permite obtener todas las variables de una consulta, pudiendo filtrar por la que se encuentran en uso indicando verdadero en el parámetro *borrarNexus*.

- *Variable GetVariable(string nomTag, int timeout = 30)*: Método que obtiene una variable indicando su nombre.

- *bool ConsultasEjecutadas(HashSet<Consulta> consultas, DateTime? ultiEjec = null, int timeout = 30)*: Método que actualiza el valor del último instante de ejecución satisfactoria de las consultas pasadas en el parámetro *consultas*. Si no se le especifica *ultiEjec* utiliza la hora del sistema como instante de ejecución.

- *bool ConsultaFallida(Consulta consulta, uint ciclo, string error, out bool parar, int timeout = 30)*: Método que se ejecuta cada vez que una consulta falla, para detenerla en caso de que tenga demasiados errores acumulados. El método devuelve verdadero si no ha tenido ningún fallo y en el parámetro *parar* indica si se debe parar la consulta. Se ha determinado que la consulta se parará si falla 50 ejecuciones consecutivas.

- *Dictionary<uint, HashSet<Consulta>> GetConsultasCiclos(out bool resultado, int segHasta = 300, int timeout = 30)*: Método utilizadopor el servicio para obtener la consultas que se deben ejecutar los próximos n segundos, determinados en el parámetro *segHasta*. Este método devuelve un diccionario donde la clave es el ciclo de ejecución y el valor es una lista de las consultas que se van a ejecutar en ese ciclo.

Un ciclo se determina como el periodo de tiempo mínimo entre el que se va a ejecutar la misma consulta, que está definido en la constante que es *TActualizado* (5 segundos por defecto). Como ciclo 0, se define el que inicia el 01-01-1970 a las 00:00:00.000. Teniendo esto en cuenta, se puede saber en qué ciclo nos encontramos, contando los segundos hasta la fecha del ciclo 0 y dividiéndolo entre el *TActualizado*. También se puede saber cada cuantos ciclos se debe ejecutar una consulta, dividiendo el *TActulizacion* de la consulta por el *TActualizado* constante.

Se puede visualizar el diagrama de clases de la interfaz *Connectors.DAO.IConsultaDAO* en la figura B.15 situada en el Anexo B.

Clase *Connectors.DAO.ConsultaMySQLDAO*

Esta clase implementa la interfaz, *Connectors.DAO.IConsultaDAO*, definida en el apartado anterior. Dado que la base de datos donde se almacenan las consultas es del tipo *MySQL*, esta clase heredada de *Connectors.DB.MySQLConnector* e implementa los métodos de la interfaz, ayudándose de los métodos existentes en el conector.

Se puede consultar el diagrama de clases de la clase *Connectors.DAO.ConsultaMySQLDAO* en la figura B.16 situada en el Anexo B.

Paquete *Connectors.Nexus* y clase *ProxyFactory*

Este paquete incluye únicamente la clase *Clase ProxyFactory* que ha sido proporcionada por *Nexus Integra* junto con las bibliotecas *CoreDataContracts* y *CoreServicesInterfaces*, que necesita para su funcionamiento. Esta clase también obtiene su configuración del fichero App.config, exactamente del apartado *system.serviceModel*.

Para conectarse a su servicio y guardar los datos, esta clase posee las operaciones:

- *ServiceClient<ICoreAdminService> GetAdminServiceProxy()*: Método para obtener el *ServiceClient* que permite crear nuevos tags y/u obtener su identificador; permite indicarle como parámetro una lista de nombres de variables y devuelve una lista con sus identificadores.
- *ServiceClient<ICoreRealTimeService> GetRealtimeServiceProxy()*: Método para obtener el *ServiceClient* que permite enviar el valor actual de una variable, indicando el identificador de la variable y su valor para almacenarlo en el sistema de tiempo real, del servicio *Nexus Integra*.
- *ServiceClient<ICoreDataFlowService> GetDataFlowServiceProxy()*: Método para obtener el *ServiceClient* que permite enviar los valores de las variables al almacenamiento permanente del servicio *Nexus Integra*, acompañándolos del identificador de la variable, del instante de obtención del valor y del valor a almacenar.

El diagrama de clases la clase *ProxyFactory* se puede consultar en la figura B.17 situada en el Anexo B.

A.2. Diseño Lógico de la base de datos

El diseño lógico de la base de datos es el siguiente. Los valores en **negrita son claves primarias**, mientras que los valores en *cursiva son no nulos* y los que poseen un subrayado son índices.

PLUG_CONEXDB (***conexDbId***, *nomDB*, *tipoDB*, *connectionString*, *userDB*, *passDB*)

PLUG_CONSULTA (***plugConsID***, *AgentID*, *nomCons*, *conexDbId*, *consulta*, *tAct*, *realtime*, *forzado*, fechIni, fechFin, fechUltimEjec, *tieneTS*, *tieneID*)

	Nulos	Borrado	Edición
PLUG_CONSULTA → *conexDbId* ⇒ PLUG_CONEXDB	No	Restringido	Propagado

PLUG_VARIABLES (***plugVarID***, plugConsID, *AgentID*, *nomTag*, *borrado*, *operadores*)

	Nulos	Borrado	Edición
PLUG_VARIABLES → *plugConsID* ⇒ PLUG_CONSULTA	Si	Restringido	Propagado

PLUG_ERRORES_CONSULTA (***plugErrConsID***, plugConsID, *fechaError*, *ciclo*, *error*)

	Nulos	Borrado	Edición
PLUG_ERRORES_CONSULTA → *plugConsID* ⇒ PLUG_CONSULTA	Si	Restringido	Propagado

Anexo B

Figuras

B.1. Diagramas de Gantt

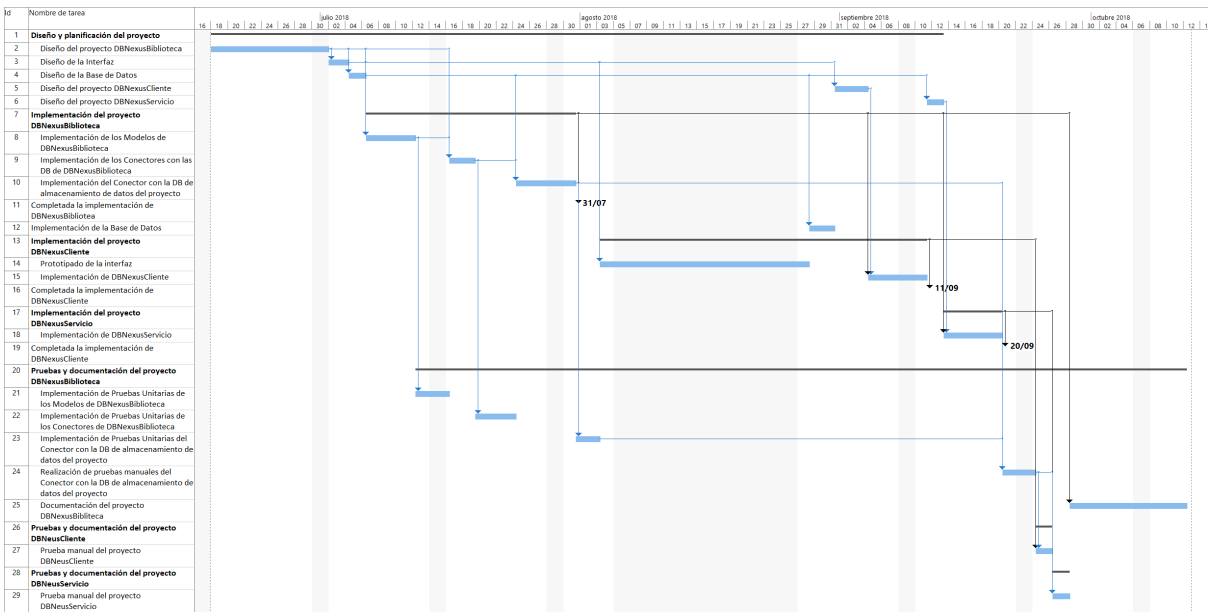

Figura B.1: Diagrama de Gantt de la planificación del Proyecto completo

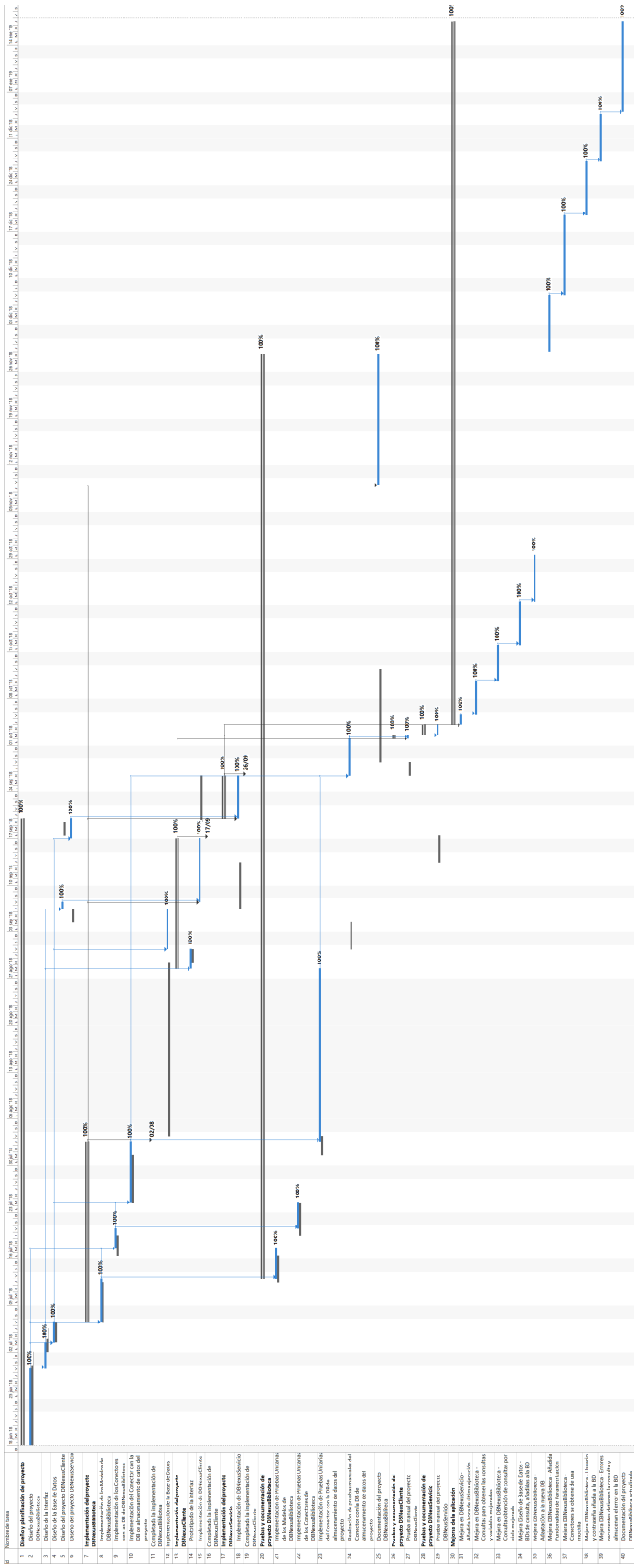

Figura B.2: Diagrama de Gantt del transcurso del Proyecto completo

B.2. Diagramas de Clases de la Biblioteca

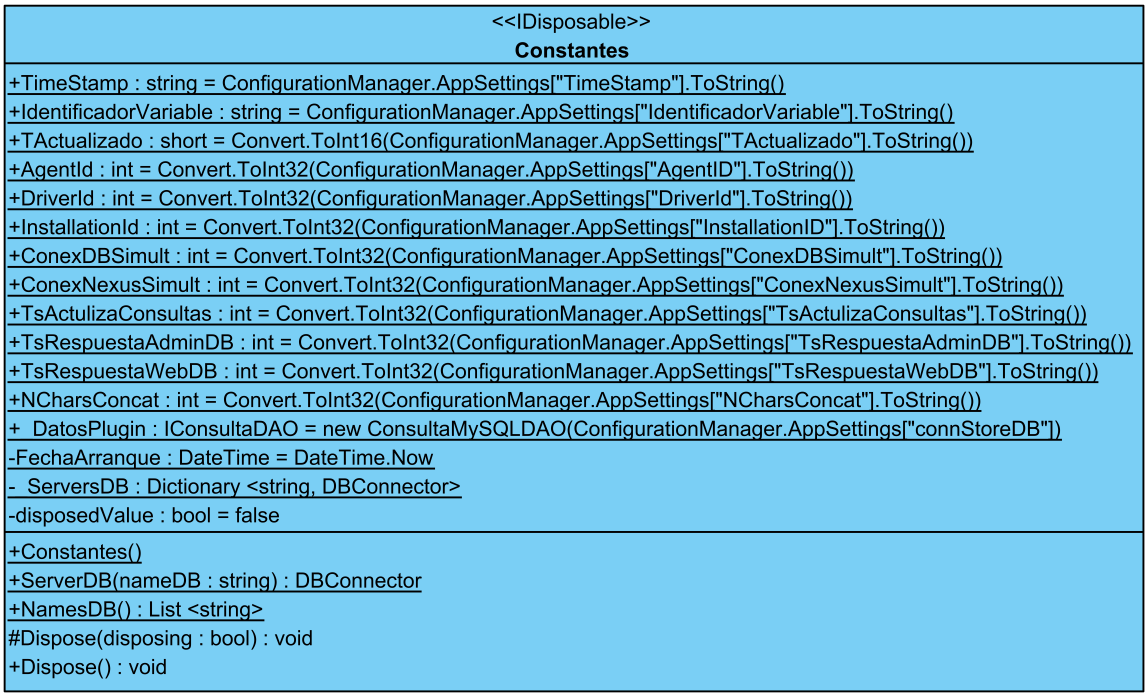

Figura B.3: Diagrama de Clases Extendido de la clase Constantes de la Biblioteca

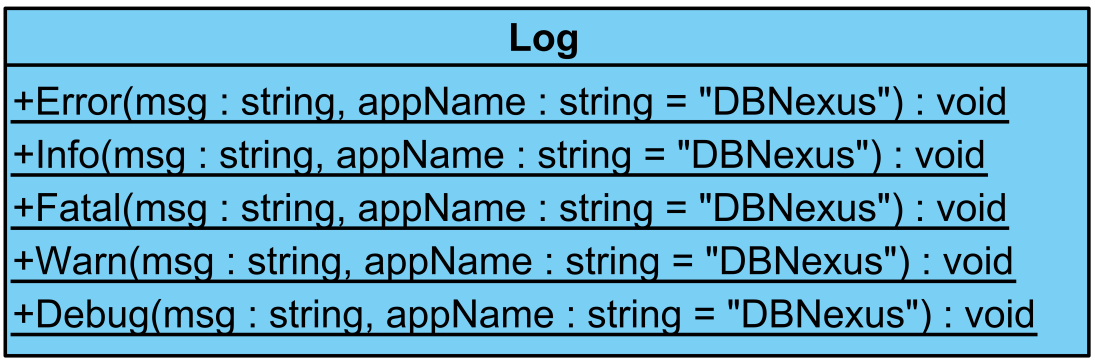

Figura B.4: Diagrama de Clases Extendido de la clase Log de la Biblioteca

Figura B.5: Diagrama de Clases Extendido de la clase Consulta del paquete Modelo de la Biblioteca

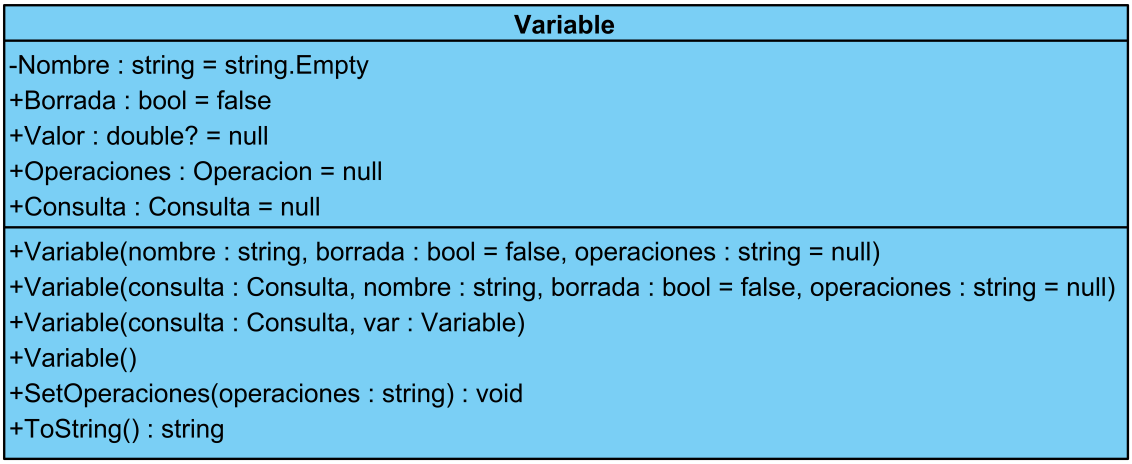

Figura B.6: Diagrama de Clases Extendido de la clase Variable del paquete Modelo de la Biblioteca

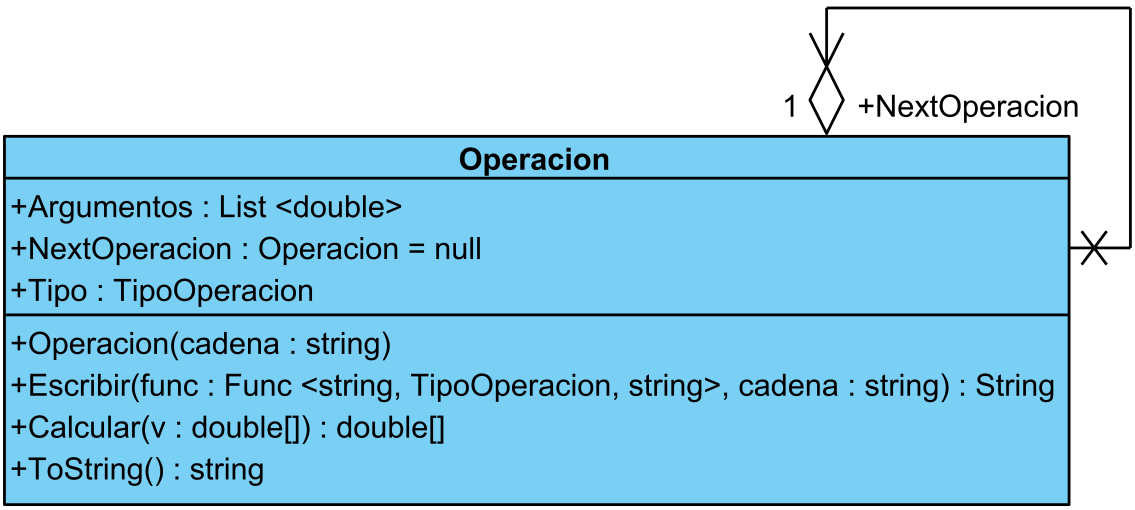

Figura B.7: Diagrama de Clases Extendido de la clase Operacion del paquete Modelo de la Biblioteca

Figura B.8: Diagrama de Clases Extendido de la enumeración TipoOperacion del paquete Modelo de la Biblioteca

Figura B.9: Diagrama de Clases Extendido de la clase DBConnectorFactory del paquete Connectors.Nexus de la Biblioteca

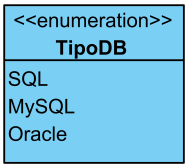

Figura B.10: Diagrama de Clases Extendido de la enumeración TipoDB del paquete Connectors.DB de la Biblioteca

Figura B.11: Diagrama de Clases Extendido de la clase DBConnector del paquete Connectors.DB de la Biblioteca

Figura B.12: Diagrama de Clases Extendido de la clase MySQLConnector del paquete Connectors.DB de la Biblioteca

Figura B.13: Diagrama de Clases Extendido de la clase OracleConnector del paquete Connectors.DB de la Biblioteca

Figura B.14: Diagrama de Clases Extendido de la clase SQLConnector del paquete Connectors.DB de la Biblioteca

Figura B.15: Diagrama de Clases Extendido de la interfaz IConsultaDAO del paquete Connectors.DAO de la Biblioteca

Figura B.16: Diagrama de Clases Extendido de la clase ConsultaMySQLDAO del paquete Connectors.DAO de la Biblioteca

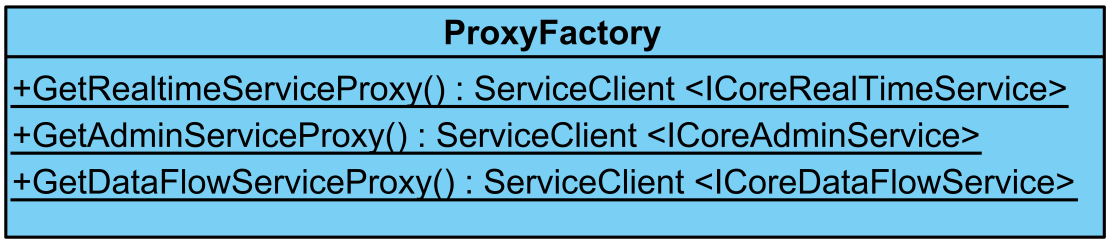

Figura B.17: Diagrama de Clases Extendido de la clase ProxyFactory del paquete Connectors.Nexus de la Biblioteca

B.3. Diagramas de Clases del Cliente

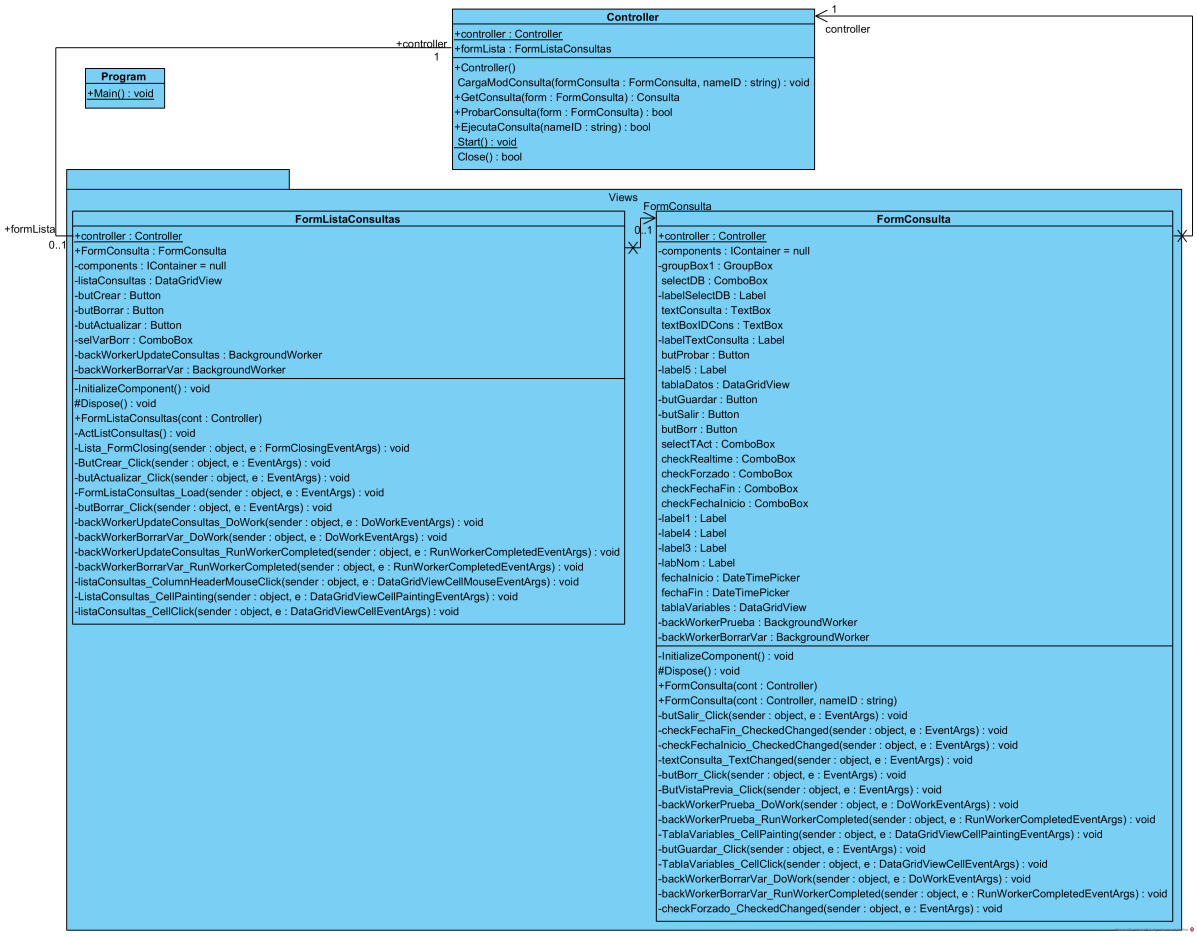

Figura B.18: Diagrama de Clases Extendido del proyecto DBNexusCliente

B.4. Diagramas de Clases del Servicio

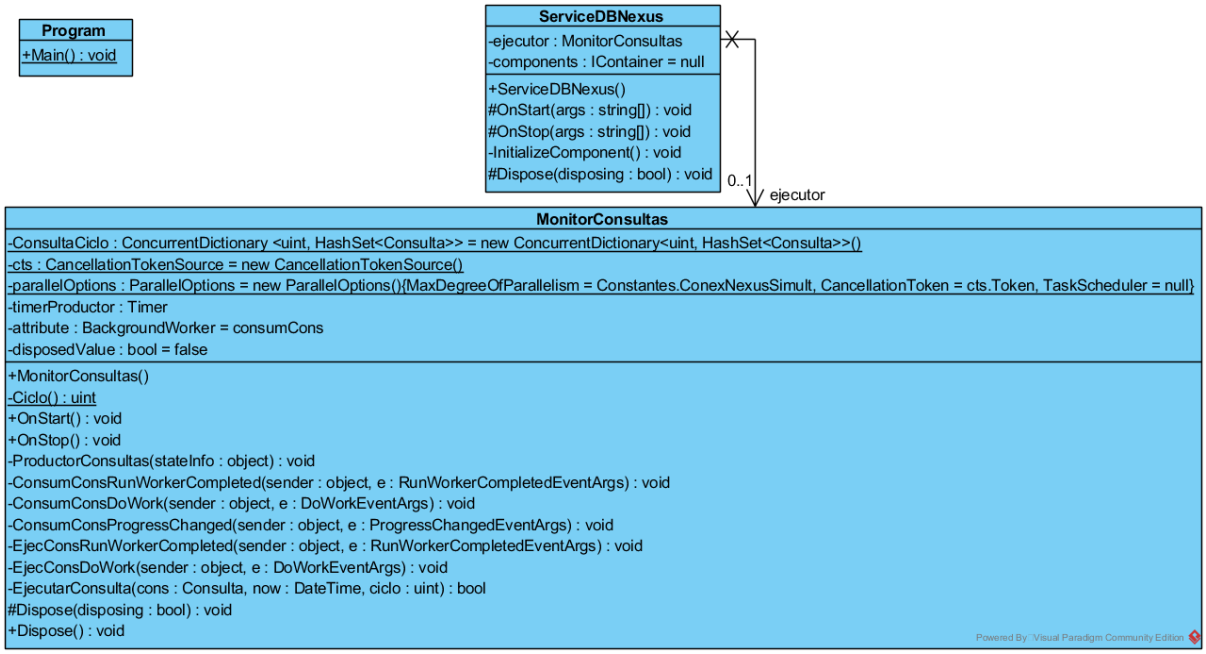

Figura B.19: Diagrama de Clases Extendido del proyecto DBNexusServicio