Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 1
Obligatorio 2

Santiago Alpuy 247034

Repositorio GIT:
https://github.com/ORT-DA1/Obligatorio1_SantiagoAlpuy

Entregado como requisito de la materia Diseño de Aplicaciones 1

Junio 2020

Índice general

1. Descripción General del Trabajo y del Sistema **2**
1.1. Funcionalidades 2
1.2. Componentes del Sistema 5
1.2.1. UserInterface 5
1.2.2. BusinessLogic 6
1.2.3. Tests 6
1.3. Estado del Entregable 7

2. Descripción y Justificación del Diseño **9**
2.1. Diagramas de Paquetes 9
2.2. Diagramas de Clases 10
2.3. Diagramas de Interacción 12
2.4. Modelo de Tablas de la Estructura de la Base de Datos 14
2.5. Descripción de Decisiones de Diseño 15

3. Cobertura de Pruebas Unitarias **19**

4. Instalación **20**

1. Descripción General del Trabajo y del Sistema

Está aplicación tiene como principal objetivo analizar frases que son ingresadas por un usuario, así como también ver los diferentes tipos de alarmas que se activan luego de superar cierta cantidad de frases que cumplen ciertas características.

1.1. Funcionalidades

- El sistema es capaz de analizar frases. Una frase puede ser positiva, negativa o neutra. Será neutra en caso de que luego de analizarla, no se encuentre un sentimiento en el sistema asociado a dicha frase, o si hay varios sentimientos de distinto tipo. En caso contrario será positiva o negativa. Una frase tendrá asociada una entidad únicamente en el caso de que dicha entidad se encuentre entre las entidades del sistema.
- Cuando se agrega o se elimina un sentimiento de cualquier tipo, o una entidad, se analizan todas las frases, pues podría ser que frases ya agregadas al sistema, tengan alguno de dichos atributos actualizados.
- Cuando se agrega un sentimiento de cualquier tipo, una entidad, una frase, se verifica si se encendieron las alarmas agregadas al sistema, pues podría ser el caso de que una alarma se active debido a esta adición.
- Cuando se elimina un sentimiento o una entidad, se verifica si se apagaron alarmas en el sistema, pues podría ser el caso que una alarma activa debía su activación a que tenía cierto sentimiento o cierta entidad que luego fue eliminada.
- Tanto en la pantalla correspondiente al sentimiento positivo como también en el negativo, se podrá ver una lista que contiene todos los sentimientos agregados de su tipo correspondiente.
- El usuario podrá agregar sentimientos tanto positivos como negativos. La unicidad de los sentimientos está dada principalmente por si es positivo o negativo, y luego por su descripción. La unicidad de la descripción no discrimina variaciones de mayúsculas, minúsculas, o espacios al comienzo y final de la misma.

No se podrá agregar sentimientos cuya descripción sea vacía, o contenga números. Esto último es una decisión que tome personalmente. El sistema avisara si no se cumplen las condiciones para agregar un sentimiento.

- El usuario podrá eliminar sentimientos de cualquier tipo.
- El la pantalla de 'Gestionar Autores', se podrá dar de alta a usuarios. Para ello, hay que proveer un nombre de usuario único, un nombre, un apellido, y una fecha de nacimiento. El nombre de usuario esta compuesto de hasta 10 caracteres alfanuméricos. El nombre y el apellido son caracteres alfabéticos de hasta 15 caracteres. La fecha de nacimiento no puede corresponder a alguien cuya edad sea inferior a 13 años, y superior a 100.
- En la pantalla de 'Gestionar Autores', también se pueden observar todos los autores agregados al sistema, con sus respectivos atributos. Para eliminar un autor, se le debe seleccionar de dicho reporte, y presionar en el botón 'Eliminar'.
- En la pantalla de 'Gestionar Autores', también se puede acceder a la opción 'Ir a Edición de Autores', en la cual dado un autor existente, seleccionado de un componente combobox, se le pueden editar sus atributos. Aquí se rigen las mismas pautas que para la creación de un autor, pero a destacar, no puede modificarse el nombre de usuario de un autor, al de un usuario ya ingresado (exceptuando a el mismo).
- Cuando se elimina un autor, se eliminan todas las frases que ha creado.
- En la pantalla de Gestión de Entidades, se podrá ver una lista que contiene todas las entidades agregadas al sistema.
- El usuario podrá agregar entidades. La unicidad de las entidades está dada por su nombre. Esta no discrimina variaciones de mayúsculas, minúsculas, o espacios al comienzo y final de la descripción. No se podrá agregar entidades cuyo nombre este vacío. El sistema avisara si no se cumplen las condiciones para agregar una entidad.
- El usuario podrá eliminar entidades.
- El usuario podrá agregar frases. Para ello deberá agregar una descripción en el campo correspondiente, seleccionar una fecha, y elegir un autor dentro de los autores ya ingresados al sistema. La fecha no puede ser superior a la fecha actual, así como también no puede ser inferior a un año desde la fecha actual. No se pueden agregar frases cuya descripción sea vacía. El sistema avisara si no se cumplen las condiciones para agregar una frase.
- Cuando el usuario agrega una frase al sistema, se le sugerirá al usuario ir a ver las alarmas activas. El usuario puede aceptar o no esta sugerencia, con el fin de garantizar la heurística de Nielsen de el control y libertad del usuario.
- Existen dos tipos de alarmas:

    - **Alarmas del Tipo A**: Se activan si hay una cierta cantidad de frases prefijadas que tienen una cierta categoría (positiva o negativa), una cierta entidad, y dichas frases se encuentran en un rango de fechas acotado superiormente por la fecha actual del encendido de la alarma, y acotado inferiormente por dicha fecha, menos el tiempo definido en campos de dicha alarma que denotan días u horas. Para crear este tipo de alarma, debe presionar el botón 'Configuración de Alarmas A'.
    - **Alarmas del Tipo B**: Se activan si uno o más autores publican una cierta cantidad de frases prefijadas cada uno, que tienen una cierta categoría (positiva o negativa), y dichas frases se encuentran en un rango de fechas acotado superiormente por la fecha actual del encendido de la alarma, y acotado inferiormente por dicha fecha, menos el tiempo definido en campos de dicha alarma que denotan días u horas. Para crear este tipo de alarma, debe presionar el botón 'Configuración de Alarmas B'.

- No se puede agregar una Alarma del Tipo A que tenga una entidad vacía, o no tenga una categoría definida, o tenga una cantidad de posts necesarios para su activación igual a cero.

- No se puede agregar una Alarma del Tipo B que no tenga una categoría definida, o tenga una cantidad de posts necesarios para su activación igual a cero, o su cantidad de posts sea superior a mil.

- No se pueden agregar posts, días u horas que no sean números naturales, el sistema les impide hacerlo.

- En la pantalla de 'Configuración de Alarmas A', se podrá ver una lista que contiene a todas las alarmas del tipo A del sistema, con sus correspondientes atributos, y si están activas o no.

- En la pantalla de 'Configuración de Alarmas B', se podrá ver una lista que contiene a todas las alarmas del tipo B del sistema, con sus correspondientes atributos, y si están activas o no.

- En la pantalla de 'Reporte de Análisis de Frases', se podrá ver una lista de todas las frases del sistema con sus respectivos atributos. Dichas frases han sido analizadas.

- En la pantalla de 'Reporte de Alarmas Activas', se podrá seleccionar el tipo de alarma activa que quiere visualizar. Si elige una alarma del tipo B y hay alarmas de dicho tipo activas, al hacer doble click sobre una de las alarmas, se podrá ver el nombre de usuario, nombre y apellido de los autores que activaron efectivamente a dicha alarma.

- En la pantalla de 'Reporte de Autores' se podrán ver diferentes estadísticas de los autores, como la cantidad de frases positivas, la cantidad de frases negativas, la cantidad de entidades mencionadas, y el promedio diario de mensajes de cada uno. También el usuario podrá ordenar tanto de manera ascendente como descendente, al seleccionar el filtro correspondiente en esta ventana.

- En la pantalla de 'Información de Uso', aparece información relacionada al entendimiento del sistema. Por el momento, informa al usuario que son los tipos de alarmas del tipo A y del tipo B.
- **Persistencia**: Se cambio el paradigma de persistencia de información. Anteriormente se guardaba en listas, y el ciclo de vida de esa información terminaba cuando se cerraba la aplicación. Ahora, utilizamos EntityFramework para persistir.

1.2. Componentes del Sistema

El sistema esta compuesto por tres componentes principales:

- UserInterface
- BusinessLogic
- Tests

1.2.1. UserInterface

El componente UserInterface es el que concentra todo lo respectivo a la Interfaz de Usuario. Es un proyecto del tipo Windows Forms App y está compuesto por una ventana principal, llamada 'MainWindow.cs', y también por múltiples UserControls.

En la ventana principal se encontraran los botones que permitirán navegar entre las distintas funcionalidades del sistema. Dichos botones siempre persistiran a lo largo del ciclo de vida de la aplicación.

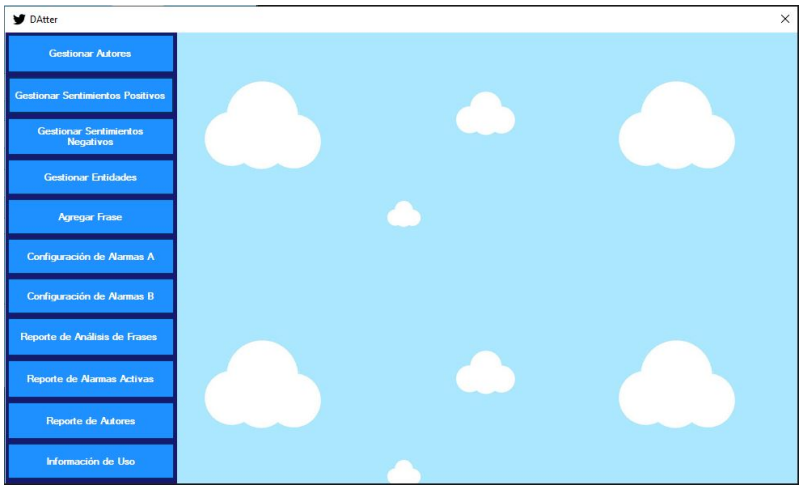

Figura 1.1: Ventana principal del sistema.

Los UserControls serán contenedores dentro de la ventana principal, específicamente a la derecha del contenedor en el cual se encuentran los botones, y servirán para que la aplicación tenga una única ventana, pues al hacer click en cada uno

de los botones de la ventana principal, se mostrara a su vez, cada uno de estos UserControls que contienen funcionalidades únicas.

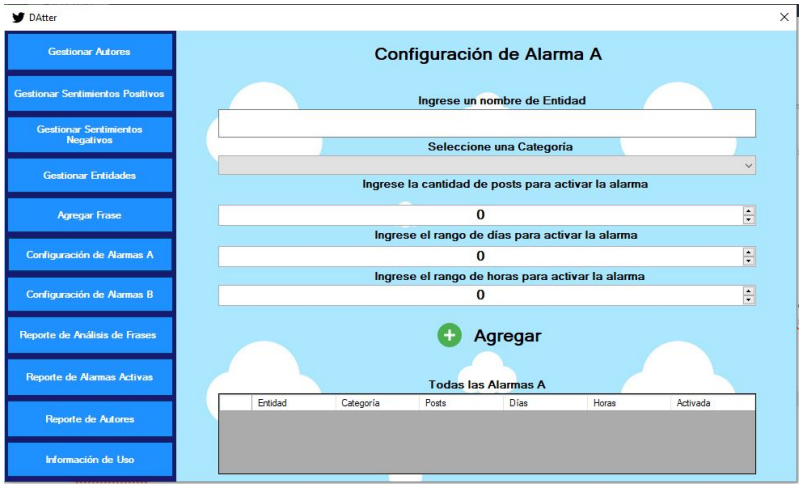

Figura 1.2: Un ejemplo de UserControl, el de sentimientos positivos.

1.2.2. BusinessLogic

El componente BusinessLogic es el que contiene toda la lógica del sistema. Dicha lógica podemos desensamblarla en las clases principales del sistema, refiriendome a los clases cuyos objetos interpretan a la realidad del problema, y tambien a otras clases que fueron añadidas debido a mejoras en cuanto a la arquitectura del sistema, y ultimamente, para la persistencia.

Hay subpaquetes dentro de BusinessLogic, como Controllers, IControllers, y DataAccess.

1.2.3. Tests

Debido a la obligatoriedad del uso del enfoque de programación TDD (Test-Driven-Development), podría decirse que en este componente se desarrollo la completitud del sistema. A través del ciclo 'Red-Green-Refactor', se fueron implementando poco a poco, todas las funcionalidades requeridas.

Para ello, se generaron clases de prueba, en la cual todos sus métodos de prueba verificaban los principios de pruebas unitarias 'FIRST', que indican que los tests deben ser rápidos, independientes, repetibles, auto-validados, y abarcativos.

1.3. Estado del Entregable

Se han implementado todas las funcionalidades solicitadas, sin excepción alguna, siguiendo el enfoque TDD y las buenas prácticas de programación que brinda Clean Code y el uso de patrones, así como también siguiendo las diez heurísticas de Jakov Nielsen para mejorar la usabilidad del usuario.

El único detalle es que no pude efectivamente implementar la relación Many-To-Many que enseñaron en el curso, y presente en el siguiente enlace:

https://www.entityframeworktutorial.net/code-first/configure-many-to-many-relationship-in-code-first.aspx

Quise modelar la relación AlertB con Author, para indicar que un autor puede estar asociado a mas de una alarma del tipo B, y que una alarma de este tipo puede estar asociada con más de un autor, para de esa manera, implementar la funcionalidad de mostrar los autores que encendieron una alarma de este tipo. Lo implemente tal cual enseñado en el curso, y consiguientemente, como aparece en el enlace anterior, pero cuando dados dos objetos **existentes**, agregaba a la lista de uno, elementos del otro, no se actualizaban, y durante tres días estuve buscando la razón y probando diversas estrategias para resolverlo.

Lamentablemente, tuve que tomar la consideración de cambiar de estrategia, y utilizar el enfoque provisto en el siguiente enlace, en el cual hay que agregar una clase intermedia cuyo nombre es la yuxtaposición del nombre de ambas clases, y que cada una de las clases originales apunten a esta nueva clase, en una relación One-To-Many, es decir, un autor tiene muchas instancias de esta nueva clase, y una alarma del tipo B tiene muchas instancias de la misma.

https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx

Esta decisión que he tomado, implica que tuve que acoplar en mayor medida a la persistencia en la lógica, ya que tuve que crear esa clase adicional.

De todas maneras, debido a este percance que tuve, estuve experimentando con diversos patrones, donde aprendí el Unit of Work, aunque finalmente lo descarte, y también aprendí sobre el patrón Data Access Object (DAO), que quise implementarlo para terminar definitivamente con el acoplamiento de la lógica con la persistencia, pero tuve problemas, por lo que lo descarte definitivamente.

Dichos problemas con el patrón DAO fueron que experimente una referencia circular entre lógica y persistencia, que luego los docentes me comentaron que podría ser resuelta utilizando Reflection, pero que estaba fuera del alcance del curso.

Otro detalle es que me hubiese gustado haber cambiado un detalle que arrastro desde el primer obligatorio. Tanto en la clase Phrase, como en la clase AlertA, se hace referencia a su asociación con una entidad a través del nombre de la misma, como un string, y no como un objeto de tipo Entity. Esto me parece algo malo, pero no he tenido tiempo para corregirlo, ya que son clases muy usadas, y el impacto de hacer un refactor de esta envergadura, es muy grande y costoso.

Más allá de esto, creo haber logrado un producto de buena calidad, donde mejore, a través de exhaustivas iteraciones en todo el código, aun más el alcance de los principios de Clean Code, donde aplique los principios GRASP y SOLID, y otros patrones interesantes como Repository, al cual cree su respectiva interfaz en esta oportunidad, y también cree la clase FactoryRepository que es la encargada de desacoplar el repositorio totalmente de los controladores.

2. Descripción y Justificación del Diseño

2.1. Diagramas de Paquetes

El sistema consta de tres paquetes principales y su funcionalidad es la descripta en la sección anterior. Estos paquetes son: UserInterface, BusinessLogic, y Tests. A su vez, dentro del paquete BusinessLogic, coexisten tres paquetes más: Controllers, IControllers, y DataAccess.

Los paquetes Controllers e IControllers fueron creados por un tema organizativo, ya que antiguamente estaban todas las clases mezcladas, fuera cual fuera su fin, y considere que por este motivo debía agruparlos en paquetes.

El paquete DataAccess fue creado por un tema conceptual. No me gusta que la persistencia este en la misma capa que en la persistencia, pero debido a que entity framework esta fuertemente acoplado a la logica en este caso, quise dejar constancia conceptual de este dilema.

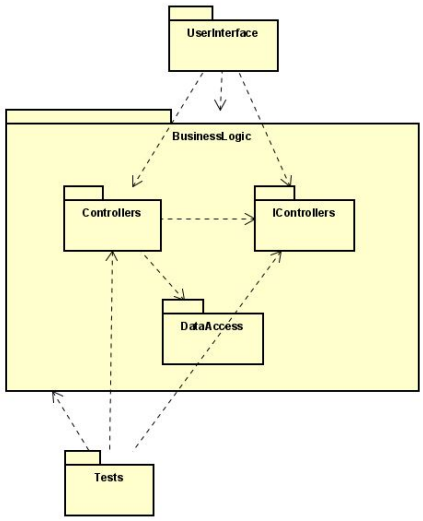

Figura 2.1: Diagrama de paquetes del sistema.

2.2. Diagramas de Clases

Este primer diagrama de clases contempla a todos los paquetes a excepción del de test, que por letra no estaba considerado en esta instancia. Es un diagrama muy grande, por lo que su adaptación a este documento puede no ser la mejor. Es por ello, que es recomendado ver este archivo de manera individual, encontrándose este en el archivo comprimido correspondiente a la entrega de este obligatorio.

Los demás archivos de imagen en esta sección también pueden encontrarse como adjuntos en el entregable.

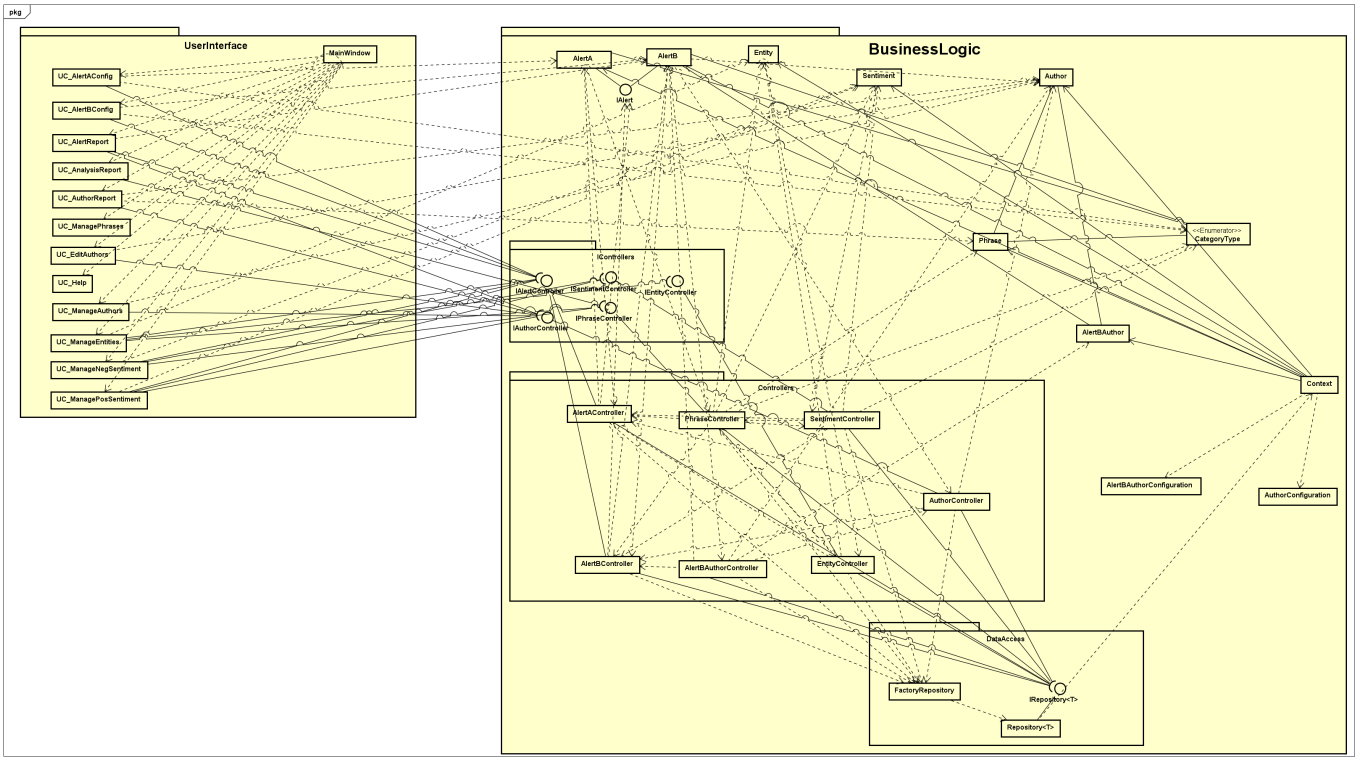

Figura 2.2: Diagrama de clases total del sistema.

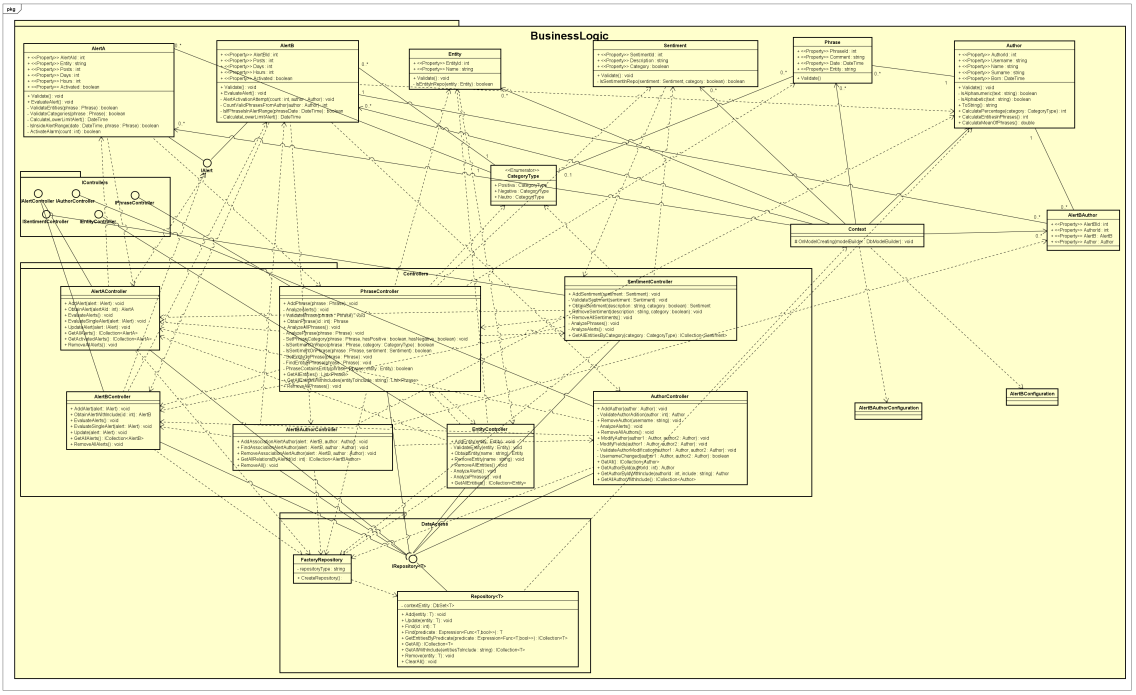

Figura 2.3: Diagrama de clases del paquete BusinessLogic.

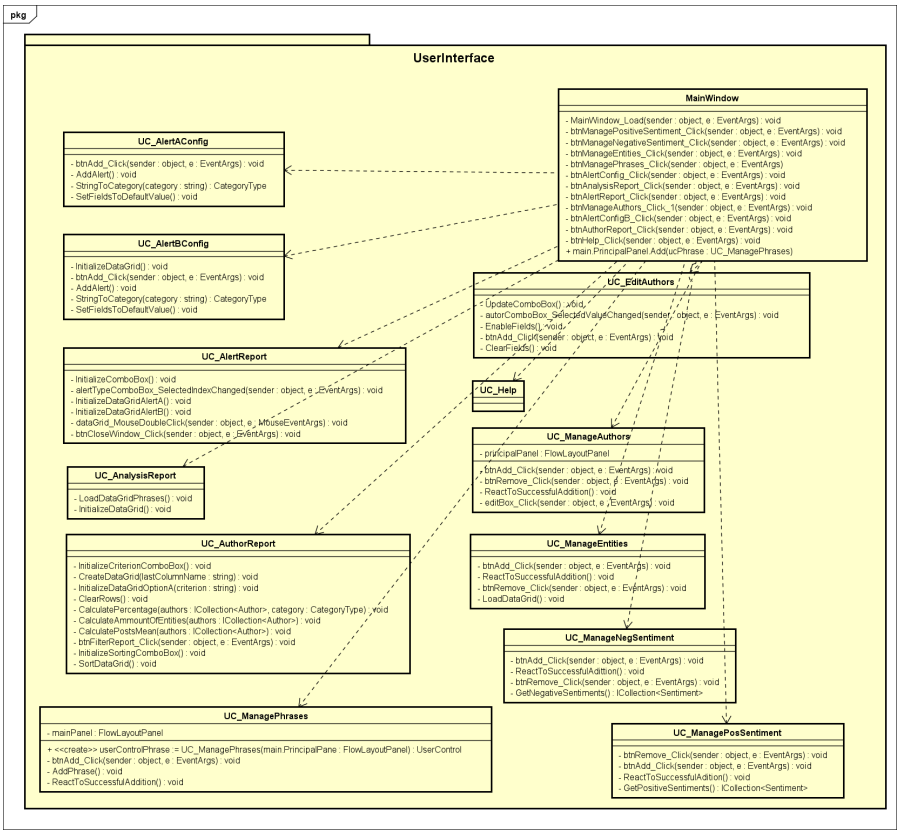

Figura 2.4: Diagrama de clases del paquete UserInterface.

2.3. Diagramas de Interacción

Para tener una perspectiva dinámica de la funcionalidad de agregar frases, realice diagramas de interacción de todo este proceso, que comprende las siguientes acciones:

- Desde el inicio, abrir la ventana de 'Agregar Frase'
- Agregar una frase a la base de datos
- Analizar una frase y actualizarla en la base de datos
- Encontrar todas las alarmas del sistema que se activan debido a la frase recientemente agregada

Dividí en varias imágenes por motivos de mejor lectura, pero comprenden un mismo hilo de ejecución, por lo que, quizás, actores creados en un diagrama, aparezcan en otra debido a que están instanciados.

Abrir Ventana 'Agregar Frase'

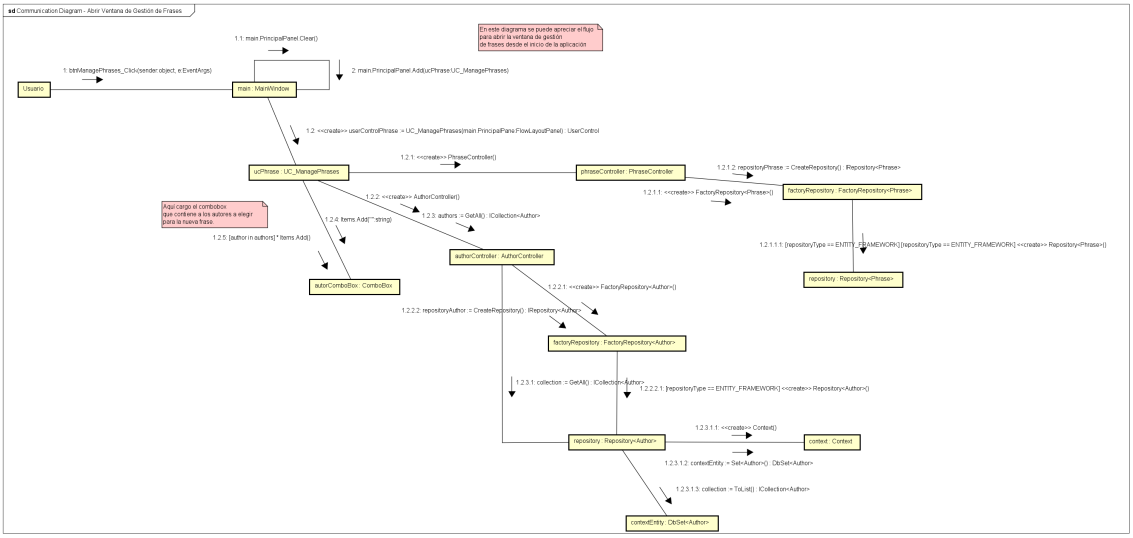

Figura 2.5: Proceso que se inicia cuando un usuario hace click en el botón 'Agregar Frase'.

Persistir Frase

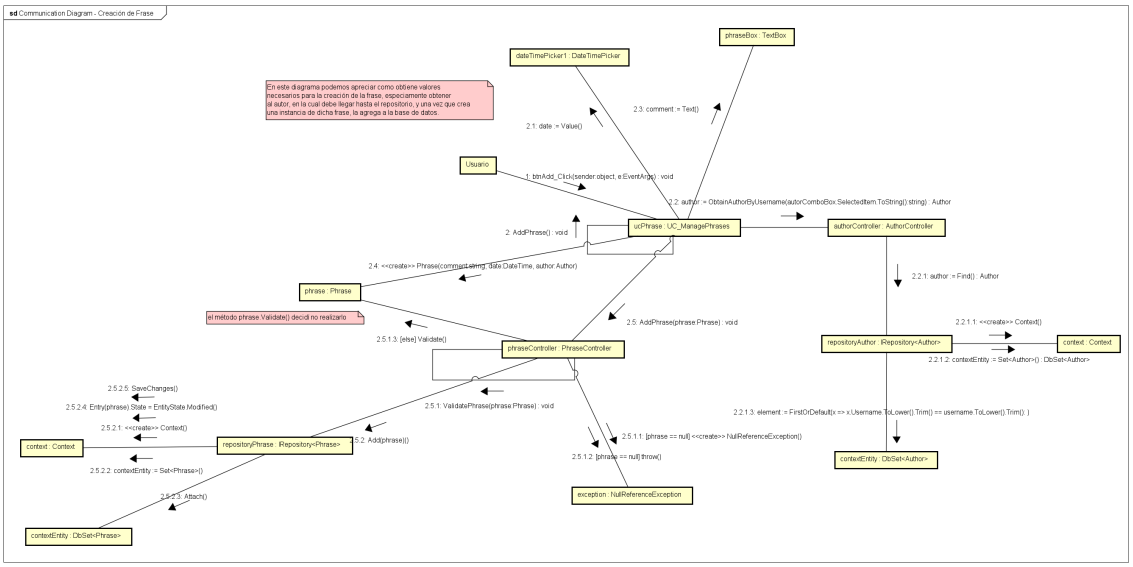

Figura 2.6: Proceso que agrega una frase a la base de datos.

Analizar Frase

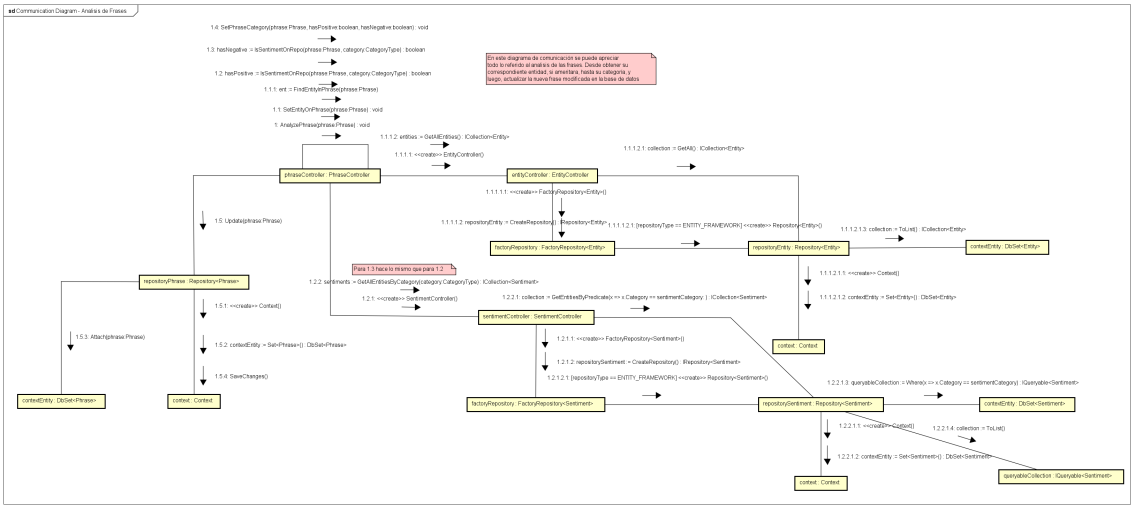

Figura 2.7: Proceso que analiza la frase recientemente agregada, y la actualiza en la base de datos.

Evaluar Alarmas

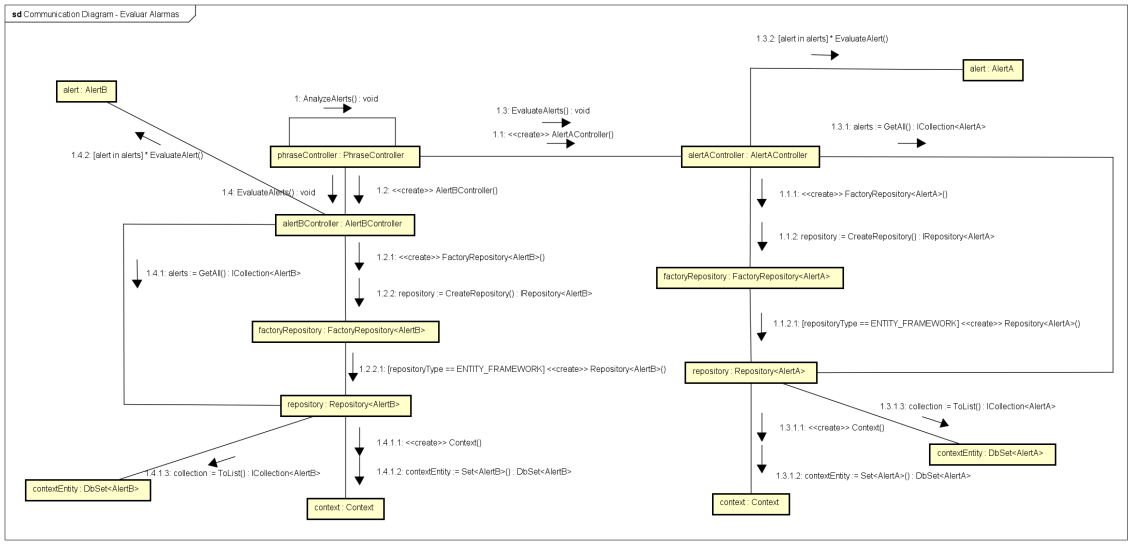

Figura 2.8: Proceso que evalúa todas las alarmas del sistema para verificar si se activan o no.

2.4. Modelo de Tablas de la Estructura de la Base de Datos

Para amoldar el sistema a EntityFramework, se tuvo que agregar un identificador a cada clase que quise mantener una referencia en la base de datos. Dicho identificador se define automáticamente por EntityFramework como Primary Key ya que sigue la nomenclatura de (nombre)Id.

Principalmente, en el sistema existe una relación One-To-Many entre Phrase y Author, es decir, una frase tiene un autor, y un autor puede tener múltiples clases. Es interesante agregar, que de eliminar un autor, se eliminaran todas sus frases, y todas las huellas que estas hayan realizado en el sistema.

Luego, como nombré en varias partes de esta documentación, tenemos una relación Many-To-Many entre Author y AlertB. Este tipo de relación no pude implementarlo como hubiera querido, entonces termine utilizando dos relaciones One-To-Many, donde se creo la relación intermedia como una clase en la lógica.

Es decir, una instancia de la clase Author tiene múltiples instancias de esta clase intermedia, y esta clase intermedia tiene una única instancia de Author. Además, una instancia de la clase AlertB, tiene multiples instancias de la clase intermedia, y esta clase intermedia tiene una única instancia de AlertB. Las Primary Key de la clase intermedia son Foreign Key que hacen referencia tanto a una instancia del tipo Author, como una instancia del tipo AlertB.

Sin más, a continuación dejo una imagen en la cual puede observarse todas las clases que quise que estuvieran presentes en la base de datos para así poder referenciarlas y trabajar con ellas.

AlertA	AlertB	AlertBAuthor	Author	Entity	Phrase	Sentiment
AlertAId (PK) Entity Category Posts Days Hours Activated	AlertBId (PK) Category Posts Days Hours Activated	AlertBId (PK) - (FK a AlertB) AuthorId (PK) - (FK a Author)	AuthorId (PK) Username Name Surname Born	EntityId (PK) Name	PhraseId (PK) Comment Date Entity Category AuthorId (FK a Author)	SentimentId (PK) Description Category

Figura 2.9: Descripción de relaciones generadas mediante el enfoque Code-First en la Base de Datos.

2.5. Descripción de Decisiones de Diseño

Utilice GIT, y más precisamente el método GITFLOW, que puede apreciarse en la imagen a continuación. Este enfoque tiene una rama master, una develop, y a partir de está última, ramas que se utilizan para agregar nuevas features, así como también para hacer refactors, que luego se fusionan con la rama develop. Cuando se tiene una versión estable, se impacta en la rama master.

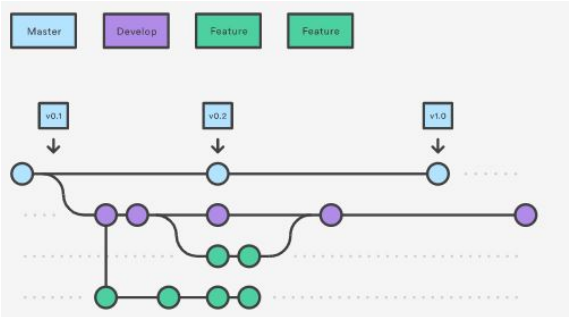

Figura 2.10: Método GITFLOW para trabajar con GIT.

Debo comentar que hasta la fecha 20/04/2020, solamente utilizaba la rama develop para trabajar, pero luego me di cuenta que no estaba aplicando GITFLOW correctamente, y enmendé el error, trabajando desde entonces sin realizar este error.

Adicionalmente, luego de recibir la retroalimentación del obligatorio número 1, empecé a utilizar las ramas que salen de develop, con la nomenclatura acción/nombreDeAccion, por ejemplo, feature/ActivateAlerts.

Para la interfaz de usuario, decidí ampararme en las diez heurísticas de Jakov Nielsen, para asegurar que los usuarios tengan la mejor experiencia posible con mi aplicación. Tener un diseño consistente, estandarizado, que no haga a los usuarios recordar decisiones, que les permita la aplicación recuperarse de errores, que haya mensajes que mantengan al tanto al usuario de que sucede en el sistema, entre otros.

Siguiendo aun con la interfaz de usuario, decidí que el sistema se encuentre en una única ventana. Para ello, tuve que hacer uso de los UserControls, que aparecen cuando se presiona en un botón indicado.

Anteriormente había creado excepciones especificas, tan especificas que solo con leer su nombre bastaría para darse cuenta de que error salio. Mi justificación era que si bien era muy difícil de mantener, beneficiaba mucho a que los programadores a simple vista, determinaran que estaba sucediendo.

La realidad es que termino siendo tan poco mantenible, que llegue a la conclusión de que mi decisión había sido muy mala, y opte por eliminar todas estas excepciones, y utilizar las excepciones nativas del lenguaje. Termino haciendo el código más limpio, mas mantenible, e hice que los mensajes que llegaban en las excepciones fueran muy descriptivos.

Las excepciones son enviadas con 'throw' desde las clases controlador, y son atrapadas en la interfaz de usuario, donde se determina que hacer cuando suceden.

En el primer obligatorio determine la existencia de controladores, cuya función es delegar trabajo a quien sepa resolverlo, por tanto, había creado dichos controladores que se encuentran en el paquete Controllers. A comienzos de este segundo obligatorio, me di cuenta que no estaban delegando tanto como a mi me gustara, entonces hice refactors muy importante con la intención de corregir esta problemática, y hacer que se cumpla el Single Responsability Principle, y también que los controladores tengan más cohesión.

Debido a estos refactors en los controladores, entre otras cosas, termine migrando métodos en todas las clases que validaban, a sus respectivas clases, siguiendo los consejos del patrón Expert, y haciendo lo propio con muchas otras clases.

En cuanto a la extensibilidad de las Alarmas, que había pensado erróneamente en el primer obligatorio, termine aplicando el patrón Factory, y creando una clase interfaz IAlert, que es implementada por los dos tipos actuales de alarmas, las clases AlertA y AlertB. Esto permite lograr la extensibilidad de las alarmas, sin modificar en gran demasía el código de fuente, apegándonos al Principio Abierto-Cerrado.

Para implementar la persistencia probé diferentes patrones. Probé el patrón Data Access Object (DAO), y me tope con la problemática de una referencia circular entre la capa de acceso de datos y la de lógica, que no supe resolver. Probé el patrón Unit of Work en conjunto con el patrón Repository, pero me encontré con un problema, ya que mis controladores hablan entre si en ciertos momentos, y al tener este patrón un único contexto, tenía problemas de acceso a objetos y termine dejándolo.

Finalmente, simplemente implemente el patrón Repository, clase que contiene la funcionalidad necesaria para interactuar con la base de datos, y dar de alta, baja, modificación a objetos, o devolver objetos de la misma.

Si bien EntityFramework esta altamente acoplado, en esta instancia, a la capa lógica, quise conceptualmente separar al repositorio de la lógica, dotándolo de su propio paquete dentro de BusinessLogic, llamado DataAccess. La forma de comunicarme con dicha clase Repository, es a través, exclusivamente, de los controladores.

Aprovechando que hablamos del repositorio, tuve una decisión arquitectural con respecto a este, ya que estaba ante la problemática de que si en el futuro se quisieran agregar otros tipos de repositorios, tendría que modificar la capa lógica, y eso, teniendo las herramientas, no está bien.

Para subsanar este problema, se creo una interfaz IRepository, que implementa la clase Repository. Además, se creo una clase FactoryRepository porque tenía el problema que desde los controladores, para obtener el repository, hacía **IRepository repository = new Repository**, por lo que aún los controladores estaban acoplados no solamente a la interfaz, sino también al repositorio.

Está clase FactoryRepository, es a quien consultan los controladores para que les devuelva el repositorio correcto. Está provista de una variable privada, que es utilizada para determinar que repositorio se debe devolver. Los controladores ejecutaran su método CreateRepository(), y lo recibirán como IRepository, es decir, no sabrán que tipo de repositorio es, pero no hay problema alguno. Con esto logro un salto de calidad importante.

Un detalle del cual estoy muy a gusto, es que implemente al repositorio como una clase genérica, ya que todas las consultas que se harían a la base de datos, serían básicamente las mismas. Se harán altas, bajas, modificaciones, consultas, y considero que implementarlo de otra manera, en este caso particular, generaría mucho a tener código duplicado, y alejar a mi aplicación de las buenas prácticas de código limpio.

Así como les dije lo anterior, también note que me saco muchísima flexibilidad haber implementado el repositorio de está manera, ya que dentro de los métodos, no podía hacer referencia a atributos particulares para cada relación, pero fue un trade-off que decidí pagar, pues fue una buena oportunidad de aprender sobre clases genéricas.

Como dije en la sección del estado del entregable, tuve un inconveniente a la hora de realizar la relación Many-To-Many, por lo que termine haciéndolo mediante otro enfoque al recomendado. Dicha decisión la tome luego de 3 días de investigar causas y probar diversas alternativas.

Está decisión, me llevo a tener que crear la clase AlertBAuthor, y un controlador que delega trabajo respecto a está clase.

Un tema que no me agrada mucho en mi aplicación, es que tengo dos controladores de alarmas, uno para cada una. He intentado encontrarle la vuelta y dejar solamente uno, pero debido a que utilizo un repositorio genérico, y como dije anteriormente, me quita flexibilidad, y no he podido hacerlo. El uso de genéricos me brindo beneficios por un lado, pero aquí me jugo en contra.

Y por último, para hablar un poco de las interfaces de los controladores, el objetivo de la creación de las mismas, fue desacoplar la interfaz de usuario de la lógica de negocios, tal cual describí anteriormente, pero relacionado a la interacción de la capa de negocios y la persistencia.

No obstante, me falto utilizar el Factory Method que utilice en la clase Factory-Repository, para que desde la interfaz de usuario no se haga, por ejemplo, **IEntityController entityController = new EntityController**, es decir, acoplar la interfaz de usuario tanto a la interfaz, como a su implementación.

3. Cobertura de Pruebas Unitarias

En total, para la culminación de mi trabajo, hice un total de 135 pruebas unitarias, 79 pruebas más que en el primer obligatorio, y todas de ellas pasando satisfactoriamente.

La cobertura del paquete BusinessLogic es del 98.98 %. Este número tan alto se debe a que utilice la metodología TDD. No logre el 100 % porque la herramienta utilizada para obtener el análisis de cobertura, en la clase Context, tuvo en cuenta unos getters que se generaron automáticamente a partir de los atributos DbSet.

Después, en la clase Author, tengo el método ToString, que implemente para que en un dataGrid de la interfaz de usuario, ponga en cierta columna al nombre de usuario del autor, y no caracteres ilegibles.
Por último, un par de ramas condicionales en otras clases no las recorro en las pruebas unitarias, pero es algo muy menor.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Santi_DESKTOP-K78SG46 2020-06-20 03_43_15.coverage	106	3.11%	3305	96.89%
businesslogic.dll	21	1.75%	1182	98.25%
{ } BusinessLogic	13	2.21%	574	97.79%
AlertA	1	0.98%	101	99.02%
AlertB	0	0.00%	94	100.00%
AlertBAuthor	0	0.00%	8	100.00%
AlertBAuthorConfiguration	0	0.00%	14	100.00%
Author	5	2.96%	164	97.04%
Author.<>c	0	0.00%	2	100.00%
Author.<>c__DisplayClass51_0	0	0.00%	2	100.00%
AuthorConfiguration	0	0.00%	15	100.00%
Context	7	28.00%	18	72.00%
Entity	0	0.00%	49	100.00%
Phrase	0	0.00%	40	100.00%
Sentiment	0	0.00%	65	100.00%
Sentiment.<>c	0	0.00%	2	100.00%
{ } BusinessLogic.Controllers	1	0.19%	539	99.81%
AlertAController	0	0.00%	46	100.00%
AlertBAuthorController	1	1.39%	71	98.61%
AlertBController	0	0.00%	37	100.00%
AlertBController.<>c	0	0.00%	2	100.00%
AlertBController.<>c__DisplayClass5_0	0	0.00%	2	100.00%
AuthorController	0	0.00%	117	100.00%
AuthorController.<>c__DisplayClass19_0	0	0.00%	2	100.00%
EntityController	0	0.00%	57	100.00%
PhraseController	0	0.00%	125	100.00%
SentimentController	0	0.00%	80	100.00%
{ } BusinessLogic.DataAccess	7	9.21%	69	90.79%
FactoryRepository<T>	0	0.00%	7	100.00%
Repository<T>	7	10.14%	62	89.86%
tests.dll	85	3.85%	2123	96.15%

Figura 3.1: Cobertura lograda mediante el uso de TDD.

4. Instalación

Incluido en el entregable del obligatorio, se encuentran los respaldos (.bak) y los scripts (.sql) de dos bases de datos.

Una de ellas está completamente vacía, y su nombre es **'EmptyDB'**. Por otro lado, tenemos a la base de datos restante, llamada **'DBwithInfo'**, que esta cargada con información como especificado en la letra del obligatorio.

Una vez restauradas las bases de datos, para seleccionar la base de datos apropiada, hay que seguir la siguiente ruta:

- Ir a la carpeta de la aplicación
- Ir a la carpeta 'UserInterface'
- Ir a la carpeta 'bin'
- Ir a la carpeta 'Release'

Una vez en está carpeta, hay que abrir en modo edición al archivo 'UserInterface.exe.config', donde se verá el connection string en el siguiente extracto en el mismo:

```
<connectionStrings>
  <add name="Context" connectionString="server=.\SQLEXPRESS; database=EmptyDB; Integrated Security=true"
       providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Figura 4.1: Connection String en el archivo UserInterface.exe.config

En donde aparece la propiedad 'database', hay que asignar el nombre de la base que se prefiera.

Por último, por defecto esta seleccionada la base de datos que esta cargada de información.