

Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 1

Obligatorio 2

Santiago Toscanini (232566)

Nahuel Biladóniga (211138)

Sofia Tejerina (209239)

Entregado como requisito de la materia Diseño de
Aplicaciones 1

25 de junio de 2020

Resumen

Para la materia Diseño de Aplicaciones 1, en el primer obligatorio se planteó la elaboración de un sistema que permita definir ciertas palabras (o combinaciones) que indican sentimientos positivo o negativos, y otras palabras que representan a las entidades. El sistema debía ser capaz de realizar un análisis básico de frases ingresadas, basándose en los datos de sentimientos y entidades que tiene guardados.

Ahora se debe agregar una nueva funcionalidad al sistema, esta versión del sistema permite el registro de autores, con alta, baja y modificación de los mismos, las frases tendrán un autor y habrá una nueva alarma centrada en los autores.

Se cambió la forma de persistencia de los datos, pasándose a usar una base de datos, de esta manera, cada que se ejecute la aplicación se contará con datos cargados con el último estado guardado antes de cerrar la aplicación.

El objetivo de este proyecto es aplicar los conocimientos adquiridos en clase, Clean code, TDD, patrones GRASP, los principios SOLID y la persistencia de datos usando Entity Framework.

Índice general

1. Descripción General	2
2. Descripción y justificación de diseño	4
2.1. Diagrama de paquetes	4
2.2. Diagrama de clases	5
2.3. Diagrama de interacción	7
2.4. Estructura de base de datos	9
2.5. Suposiciones	9
2.6. Descripción de diseño	10
2.6.1. Diseño actual	10
2.6.2. Asignación de Responsabilidades	15
3. Pruebas	19
3.1. Cobertura de pruebas unitarias	19
Bibliografía	20

1. Descripción General

El sistema permite el ingreso de datos (entidades, sentimientos positivos, negativos y autores) para poder realizar distintos análisis de las frases.

Las funcionalidades con las que contaba la primer parte del sistema eran las siguientes:

- Alta de sentimientos positivos y negativos, estos se podrán visualizar.
- Baja de sentimientos positivos y negativos, siempre y cuando no se encuentren en alguna frase.
- Alta de distintas entidades, estas se podrán visualizar.
- Alta de frases, estas se podrán visualizar.
- Realizar el análisis de las frases ingresadas para determinar que tan positiva o negativa es respecto a la entidad de la que se habla.
- Creación de alarmas que, dado un tiempo (en horas o días) hacia atrás, una entidad a ser detectada, y un tipo de sentimiento a buscar (positivo o negativo) y una cantidad de sentimientos, evalúe si debe activarse, estas se podrán visualizar, así como también su estado.

Actualmente se le agregaron nuevas funcionalidades, tales como:

- Alta de autores, estos se podrán visualizar.
- Baja de autores con sus respectivas frases.
- Modificación de autores.
- Las frases ahora contienen un autor, además de la entidad y los sentimientos.
- Reporte de autores, que podrán ser ordenados por el usuario según los siguientes criterios: porcentaje de frases negativas, porcentaje de frases positivas, cantidad de entidades mencionadas por cada autor, promedio de frases diario de cada autor.

- Consulta de una gráfica de barras con los 10 primeros autores de la lista y su porcentaje de frases negativa, porcentaje de frases positivas, cantidad total de entidades mencionadas o promedio de frases diario. La gráfica dependerá de que criterio el usuario elija para ordenar a los autores.
- Persistencia de datos del sistema, al iniciar la aplicación se contará con todos los datos cargados con el último estado guardado.

Al ingresar a la aplicación se encontrará con el menú principal, a la izquierda tendrá las opciones para acceder a las distintas funcionalidades.

La lógica del programa fue desarrollada utilizando TDD (desarrollo guiado por pruebas), siguiendo las buenas prácticas propuestas por Clean Code, haciendo uso de los patrones de diseño GRASP y los principios SOLID. La persistencia de datos se realizó utilizando Entity Framework, para poder trabajar con la base de datos utilizando objetos del dominio.

Aplicar SOLID Y GRASP como nos hubiera gustado no fue posible, tuvimos que romper algunas de sus reglas debido a que el programa es muy chico y seguirlas al pie de la letra requería hacer más compleja la lógica de forma innecesaria. Sobre todo fue difícil seguir el principio de responsabilidad única, un ejemplo de esto es el "Controlador" el cuál idealmente no debería tener lógica; su tarea es pasar los datos a las clases correspondientes que los van a utilizar, sin embargo en nuestro caso, aconsejados por nuestros docentes, decidimos que era contraproducente crear clases que cumplan la funcionalidad de "Servicios" que se encargaran de esa lógica que actualmente se encuentra aquí.

Para esta entrega procuramos seguir las buenas prácticas recomendadas por el libro "Clean Code", debido a que en la primera versión del sistema fue donde estuvimos más en falta, también intentamos arreglar aquellos errores que nos fueron marcados en la entrega anterior, no fue posible arreglar todos por falta de tiempo, nos quedaron todavía métodos que reciben muchos parámetros, algo que Clean Code toma como erróneo.

La gráfica de barras con las estadísticas de los autores, si bien es parte del reporte de estos, no lo pusimos en la misma pantalla porque la pantalla de nuestro sistema es pequeña, al intentar poner el reporte y la gráfica juntas se sobrecargaba de información, por lo que optamos por poner la gráfica en otra pantalla que se dedica exclusivamente a mostrar las gráficas y no la tabla.

El repositorio se encuentra en la URL:
<https://github.com/ORT-DA1/Biladoniga-Tejerina-Toscanini>

2. Descripción y justificación de diseño

2.1. Diagrama de paquetes

El diagrama de paquetes tiene como objetivo dar una visión más clara del sistema, organizando y agrupando los elementos de lógica, interfaz y test, detallando las relaciones de dependencia entre ellos.

Los paquetes son una agrupación de elementos; clases o componentes, a su vez también puede contener otros paquetes que contendrán alguno de los elementos anteriores.

Los paquetes se representan con el dibujo de una "carpeta", las relaciones de dependencia con una flecha punteada y los elementos que contiene cada paquete con un cuadrado con su respectivo nombre.

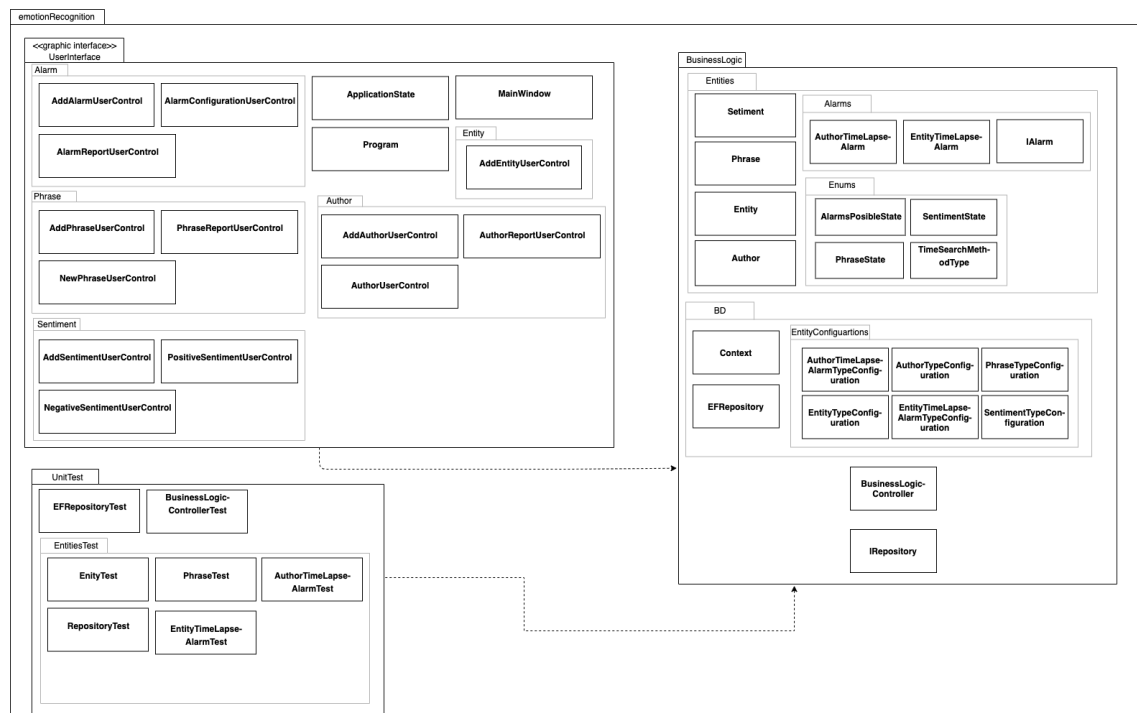


Figura 2.1: Diagrama de paquetes

Como se puede ver en el diagrama nuestro sistema esta separado en 3 grandes paquetes: `UserInterface`, `BusinessLogic` y `UnitTest`. Cada paquete de estos corresponde a un nivel diferente del sistema; `UserInterface` almacena todo lo relacionado con la interfaz de usuario, pantallas y formularios, `BusinessLogic` de la lógica, persistencia de datos, validaciones y análisis, `UnitTest` de los tests unitarios que hicimos para guiarnos en el desarrollo de la lógica de negocio.

Nos centraremos ahora en el paquete de `BusinessLogic`, dentro de este podemos encontrar dos paquetes más, uno encargado de la base de datos, con las configuraciones para cada entidad de la base de datos, la clase contexto que nos permite comunicarnos con la base de datos, el repositorio que se encarga del almacenamiento de datos (en este caso `EFRepository` se encarga de guardarlos en la base de datos). El otro paquete, se encarga de las clases de nuestro sistema y los enumeradores. Tenemos también la clase `BusinessLogicController` encargada de la comunicación con la interfaz de usuario, brindándole los datos que necesita y obteniendo los datos que el usuario ingresa.

2.2. Diagrama de clases

Los diagramas de clases son uno de los tipos de diagramas más útiles en UML, ya que muestran claramente la estructura de un sistema al modelar sus clases, atributos, operaciones y relaciones entre objetos.

Las relaciones entre objetos que aquí se observan son las siguientes (ordenadas desde la más débil a la más fuerte):

- Dependencias: Cuando una ClaseA crea, recibe o devuelve una instancia de una ClaseB, esa ClaseA tiene una dependencia con la ClaseB y se representa con una flecha punteada.
- Asociación: Cuando una ClaseA se construye a partir de una ClaseB, entonces la ClaseA tiene una asociación con la ClaseB. Se representa con una flecha.
- Agregación: La agregación es un tipo de asociación que indica que una clase es parte de otra clase, pero destrucción del compuesto no conlleva a la destrucción de los componentes. Por ejemplo la ClaseA tiene una lista de elementos de la ClaseB pero esta lista puede estar vacía sin afectar que la contiene. Se representa con una flecha que en el extremo tiene un rombo blanco, y a diferencia de las anteriores la ClaseA es la que tiene el rombo.

2.3. Diagrama de interacción

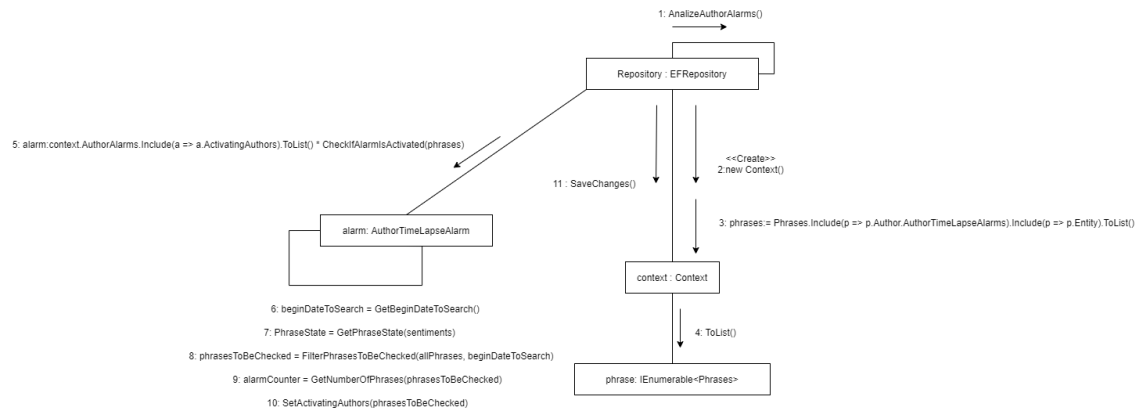


Figura 2.4: Diagrama de comunicación de análisis de alarma de autores

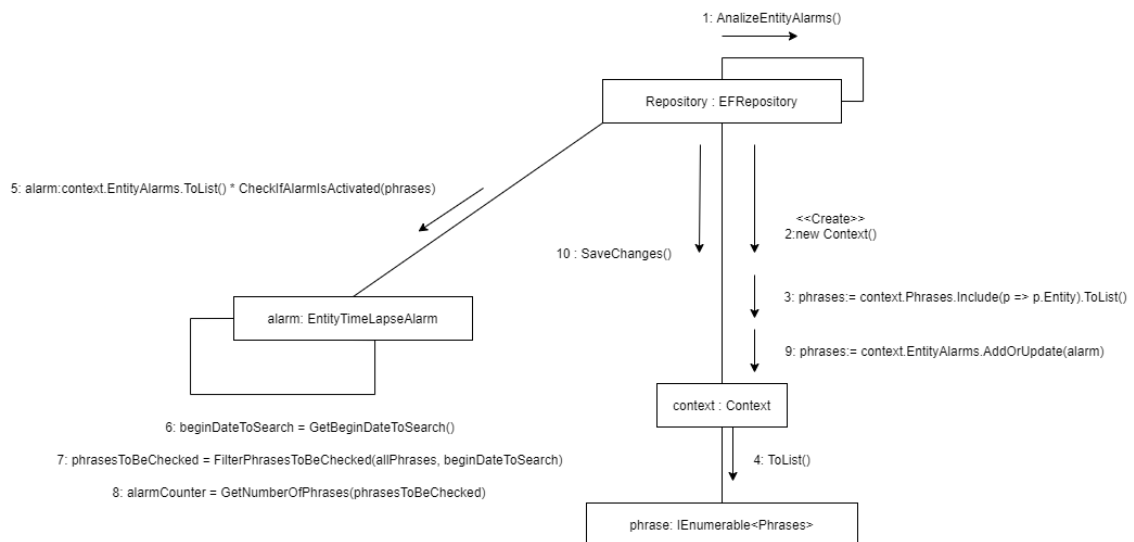


Figura 2.5: Diagrama de comunicación de análisis de alarma de entidades

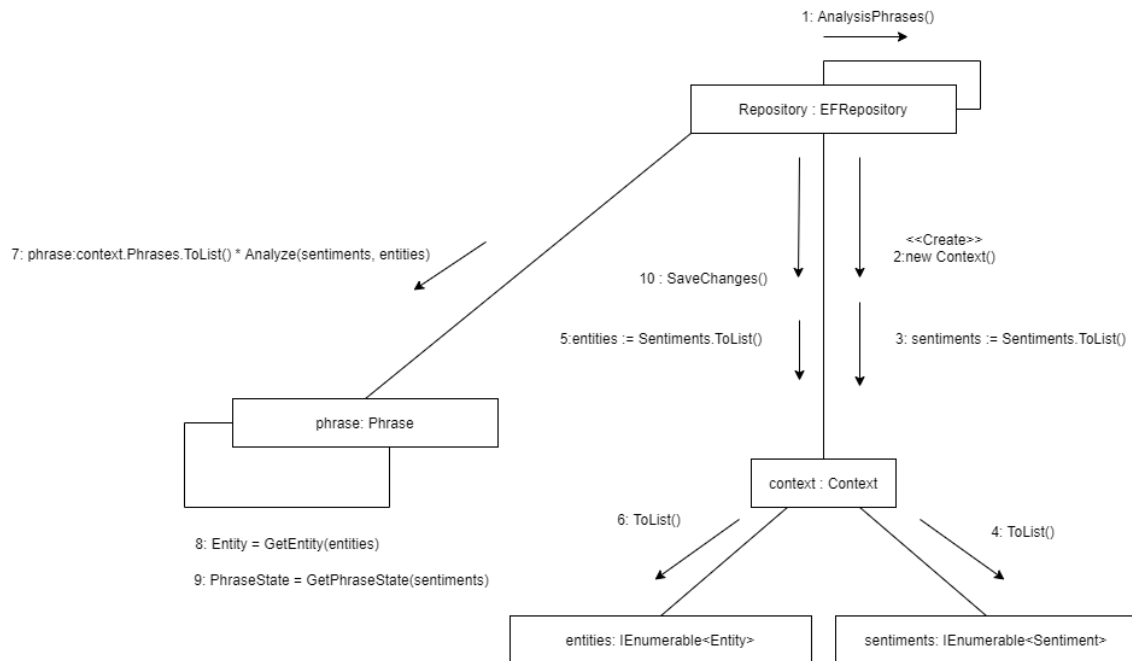


Figura 2.6: Diagrama de comunicación de análisis de frases

2.4. Estructura de base de datos

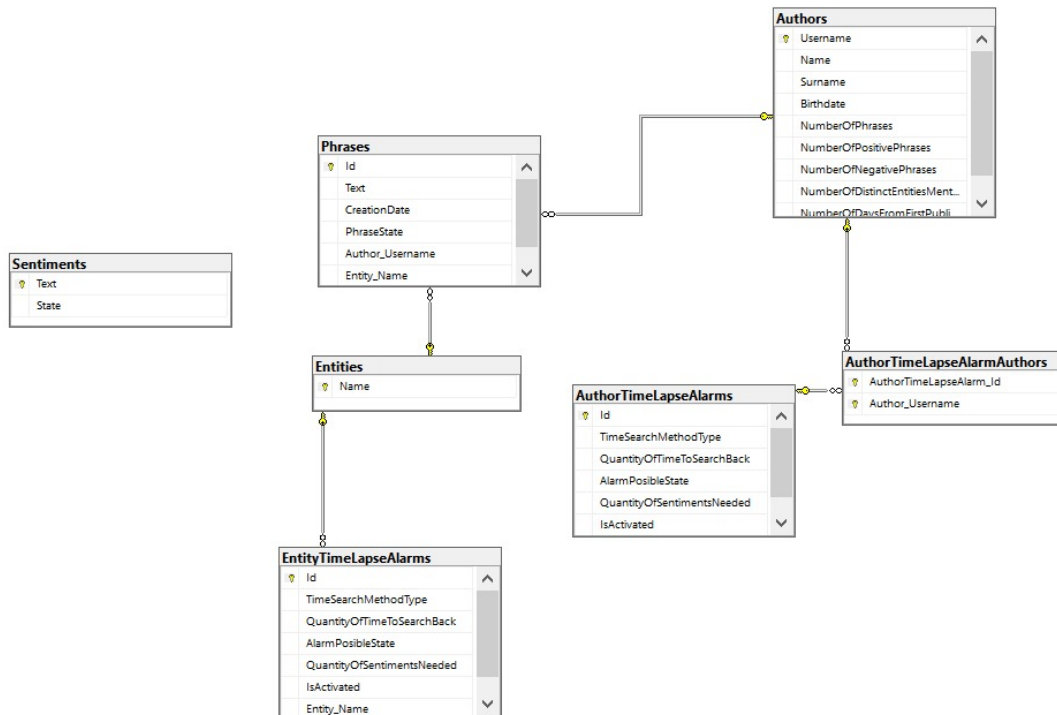


Figura 2.7: Estructura de base de datos

2.5. Suposiciones

Para realizar el diseño del obligatorio nos basamos en la letra del mismo y en las preguntas que se fueron realizando en el foro, ciertas dudas que surgieron no estaban en ninguno de los dos lugares, en nuestro lugar realizamos los siguientes supuestos:

1. Si hay mas de un sentimiento igual en una misma frase lo tomaremos en cuenta una sola vez, por ejemplo si tenemos una entidad "Chocolate" y un sentimiento "rico" que es positivo, la frase "el chocolate es rico, muy rico", la tomaríamos como poco positiva, dado que se registra una única vez el sentimiento "rico".
2. No tomamos en cuenta que una entidad pueda estar en la lista de sentimientos y viceversa, esto no fue consultado en el foro y tampoco se especifica en la letra, por lo que no lo tomamos en cuenta, somos conscientes de que si se ingresa una frase como "ORT" y tenemos registrado a "ORT" como sentimiento y entidad, nos dirá que detecta la entidad y el sentimiento, el principal problema de esto es no poder diferenciar cuando se habla del sentimiento y cuando de la entidad.

3. Una entidad o un sentimiento puede estar contenido dentro de otra palabra "normal", por ejemplo, la entidad "ola" sería detectada dentro de la frase "¿Hola, cómo estas?"
4. Las alarmas son analizadas luego del ingreso de una frase, alarma.
5. Las frases son analizadas luego del ingreso de una entidad, un sentimiento o una frase.
6. Asumimos que para el correcto funcionamiento del sistema, quien lo vaya a utilizar deberá tener correctamente configurado el connectionString en el archivo UserInterface.exe.config que se encuentra en la misma carpeta que el ejecutable. Para el caso de esta entrega utilizamos como valor para el Data Source: localhost/sqlexpress, que apunta a la maquina actual, a una base de datos sqlexpress, pero esto podría cambiarse luego para utilizar otra base de datos, y el ORM se encargaría de que todo siga funcinando.

2.6. Descripción de diseño

2.6.1. Diseño actual

Para la implementación de este sistema creamos tres proyectos; un Class Library .NET Framework EntityFramework (BusinessLogic), un MSTest .NET Framework EntityFramework (UnitTest) y un WindowsForms .NET Framework EntityFramework (UserInterface).

Dentro de UserInterface tenemos una clase UserControl por cada pantalla y una clase Formulario que es la ventana principal que contiene el menú de opciones que permiten acceder a las funcionalidades.

Para cambiar entre las pantallas creamos un enum "ApplicationState".

En general en UnitTest tenemos una clase de Test para cada Clase dentro de BusinessLogic, donde fuimos creando los test que nos permitieron ir creando la lógica de todo el sistema.

Creamos seis clases una para cada elemento esencial de nuestro sistema: "Entity", "Sentiment", "Author", "Phrase", "EntityTimeLapseAlarm" (anteriormente llamada "TimeLapseAlarm") y "AuthorTimeLapseAlarm". Estas mismas clases son usadas como entidades por el ORM.

Tomamos la decisión de crear una interfaz "IAlarm", esta contiene el comportamiento base que debe tener una alarma en nuestro sistema, hasta ahora ese es su análisis. Pero en un futuro se le podría agregar un nuevo comportamiento a las alarmas, y de esta forma nos aseguraremos de que todas las implementaciones lo cumplan, también mantenemos la consistencia entre las alarmas; ya que cada alarma

nueva que se decida agregar al sistema deberá cumplir con el comportamiento básico.

Con la interfaz tratamos de seguir el principio de "Liskov substitution", por lo cual si necesitamos evaluar por alguna razón todos los tipos de alarmas, no necesitaremos ir preguntando por el tipo, debido a que, aunque tengan distinta implementación, todas las alarmas son del tipo "IAlarma". También nos permite seguir el patrón GRASP "Polimorfismo".

Una interfaz "IRepository", que define el comportamiento de las clases "Repository" de nuestro sistema, estas son las encargadas de guardar los datos que se crean en el sistema: entidades, autores, frases, alarmas y sentimientos. Además de eso se encargaran de realizar el análisis de frases, alarmas y autores debido a que los mismos hacen cambios a las entidades guardadas y se deben almacenar.

Intentamos seguir con todas nuestras clases el principio de "Single responsibility", para eso implementamos cambios en el diseño y la arquitectura de nuestro sistema, arquitectura que habíamos definido para la primera versión pero por falta de tiempo no se había logrado implementar. Esta nueva arquitectura se basaba en el uso de controladores para comunicar la interfaz con la lógica (punto que veremos más adelante) y de esa forma que el repositorio se encargara únicamente de almacenamiento. A su vez para esto tendríamos que implementar "Servicios" encargados de la lógica correspondiente que se debiera hacer antes de almacenar un objeto; lo ideal sería un servicio para cada concepto, es decir por ejemplo un "Servicio Alarma" que se encargara de recibir la alarma, crearla, analizarla y guardarla. Esto también sigue patrón GRASP de Indirección, porque los Servicios se encargarían de "mediar" entre el controlador y el repositorio, y el patrón "Creador", donde cada servicio se encarga de la creación de su respectivo elemento del dominio. Por el porte del proyecto, se nos aconsejó no realizar esta última parte de la implementación del nuevo diseño, por esa razón nuestro repositorio no cumple con el principio, además de almacenar los datos ingresados y brindar los datos solicitados, se encarga de la creación y análisis. El resto de nuestras clases, si siguen firmemente el principio de responsabilidad única.

Las interfaces que creamos en nuestro sistema cumplen con el principio de Segregación de la Interfaz, es decir que son específicas para un tipo de comportamiento, definen lo necesario, que se va a utilizar, no tiene métodos o funciones forzadas.

Luego para obtener la persistencia con una BD se utiliza la clase "EFRepository" la cual implementa las operaciones definidas en "IRepository", operaciones básicas que deben cumplir los repositorios de nuestro sistema. Esto es útil debido a que para cambiar la forma de almacenamiento, como hicimos anteriormente al cambiar del almacenamiento local al uso de base de datos, lo único que deberíamos hacer es cambiar el tipo al crear la instancia del Repositorio en el Controlador y el sistema seguirá funcionando de la misma forma, porque siempre estamos usando "IRepository" para referirnos a nuestra clase encargada del almacenamiento, sin preocuparnos el cómo almacena.

Como se ve, la implementación de una interfaz repositorio nos permite minimizar el impacto de un cambio en la forma de almacenamiento, como nos sugiere el patrón GRASP "Variaciones protegidas".

El principio de "Inversión de Dependencias" se puede ver claramente en cómo el controlador es el único en instanciar al repositorio, luego para referirse al repositorio que se este utilizando se utiliza la interfaz "IRepository". Como también puede verse en los constructores que reciben por parámetro las instancias de las clases a almacenar internamente y no las instancian dentro, por lo tanto nuestro código depende de la abstracción, logrando un bajo acoplamiento.

El uso de un "Repositorio" de por sí, sigue el patrón de "Fabricación pura" de GRASP, es una clase que no representa un concepto del dominio que fomenta la alta cohesión y el bajo acoplamiento, además claro de su capacidad de reutilización (gracias al uso de la interfaz).

Para comunicar la interfaz de usuario con la lógica de negocio creamos una clase "BusinessLogicController", esta se encarga de brindarle a las clases de la interfaz lo que necesitan de la lógica limitándole el acceso a esta. La ventaja de mantener separada la lógica de la interfaz es, que si en un futuro se decide usar esta lógica para otro tipo de aplicación (aplicación web, mobile, etc) se podría fácilmente exponer las funciones del controller por medio de una API y la lógica no se vería afectada. Se podría decir que el controller es la única puerta de comunicación a toda la lógica y limita que información se puede obtener de esta.

El controlador, principalmente cumple con el patrón GRASP de su mismo nombre, que resuelve el problema de manejar los eventos de entrada y salida del sistema.

El "BusinessLogicController" tiene una variable de "IRepository", la cuál es almacenada en el constructor del controlador. En la ventana principal de la interfaz de usuario creamos la instancia de "BusinessLogicController" que utilizaremos en toda la aplicación. Por esa razón, con solo hacer un cambio en controller basta para que se utilice una nueva forma de almacenamiento y el resto del sistema no se vea afectado.

Dentro de "BusinessLogic" también tenemos un paquete con los enums que utilizamos en las distintas clases de lógica. Los enums son una buena herramienta para evitar errores en el código y mantener un estándar, aportan claridad y lo hacen más escalable. Por ello, tenemos un "SentimentState" para definir el estado de un sentimiento, cada estado, positivo y negativo, tiene un valor numérico asociado, 3, 2, 1 para altamente positivo, medianamente positivo y poco positivo respectivamente, 0 para neutral y -1, -2, -3 que se utilizan para poco negativo, medianamente negativo y altamente negativo respectivamente, que se utilizan para determinar que tan positiva, negativa o simplemente neutra es una frase que se ingresó, al analizar los sentimientos que contiene. "AlarmPossibleState" para determinar el estado que valida la alarma; si valida el ingreso de frases positivas o negativas, respecto a una cierta entidad. También para las alarmas tenemos "TimeSearchMethodType" para

saber si va a estar evaluando en días o en horas hacia atrás. Para las frases tenemos "PhraseState", similar al "SentimentState" para los sentimientos, ayuda a determinar si una frase es positiva, negativa o neutra, y si es alguna de las primeras dos opciones en qué grado, también con un número asociado para ayudarnos a determinar qué estado tiene.

Para las listas y sets se utilizaran interfaces del tipo IEnumerable, esto combinado a que el repositorio es una interfaz dejaría muy simple el hecho de un cambio en el repositorio, ya sea por otro repositorio local con distintas implementaciones internas, como así una base de datos, leer datos de una cola de mensajes, etc.

Con esta arquitectura generamos una división total entre la vista y los modelos, dejando abierta la implementación en un futuro de otra interfaz para el usuario.

Para la codificación utilizamos el estándar propuesto en Microsoft Docs para .NET:

- Properties, firmas de métodos, nombres de clases y enums, constantes: con PascalCase
- Variables, atributos y parámetros: con camelCase
- Interfaces: con PascalCase y un "I" al comienzo. Esto se debe a que en C# implementamos y extendemos de una clase con el mismo símbolo reservado ":", a diferencia de Java por ejemplo que diferencia "implements" de "extends". Debido a que utilizamos TDD para la creación de este proyecto, y con este no llegamos a la necesidad de utilizar Interfaces, no tenemos ningún ejemplo de esto en nuestro proyecto.
- Valores de los enumerados: con SCREAMING_SNAKE_CASE

Para los elementos del proyecto en WindowsForms utilizamos el estándar propuesto en el libro "ASP.NET Developer's JumpStart" escrito por Paul D. Sheriff, Ken Getz :

TABLE A.5 Prefixes to Use with WinForm Controls

Control	Prefix
Label	lbl
LinkLabel	lnk
Button	btn
TextBox	txt
MainMenu	mnu
CheckBox	chk
RadioButton	rdo
GroupBox	grp
PictureBox	pic
Panel	pnl
DataGrid	grd
ListBox	lst
CheckedListBox	clst
ComboBox	cbo
ListView	lvw
TreeView	twv
TabControl	tab
DateTimePicker	dtp
MonthCalendar	cal
HScrollBar	hscr
VScrollBar	vscr
Timer	tim
Splitter	spl
DomainUpDown	dup
NumericUpDown	nup
TrackBar	trk
ProgressBar	prg
RichTextBox	rtxt
ImageList	ilst
HelpProvider	hlp

Figura 2.8: Estándar para elementos de WindowsForms

TABLE A.5 Continued

Control	Prefix
ToolTip	tip
ContextMenu	cmnu
ToolBar	tbar
StatusBar	sbar
NotifyIcon	nic
OpenFileDialog	ofd
SaveFileDialog	sfd
FontDialog	fd
ColorDialog	cd
PrintDialog	pd
PrintPreviewDialog	ppd
PrintPreviewControl	ppc
ErrorProvider	errp
PrintDocument	pdoc
PageSetupDialog	psd
CrystalReportViewer	crv

Figura 2.9: Estándar para elementos de WindowsForms

Utilizamos estos prefijos antes del nombre del elemento, y lo escribimos utilizando PascalCase por sugerencia de VisualStudio.

2.6.2. Asignación de Responsabilidades

"Entity": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Crear los objetos de tipo Entity.
- Administrar los datos de la entidad (el nombre de la entidad).
- Distinguir entre dos entidades si son iguales o no.
- Pasar a texto la entidad.

"SOLID": Cumple con los principios S,O el resto no aplica

"GRASP": Experto, Alta Cohesión, Bajo Acoplamiento

"Sentiment": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Crear los objetos de tipo Sentiment.
- Administrar los datos del sentimiento (el texto y el estado del sentimiento).
- Distinguir entre dos sentimientos si son iguales o no.

"SOLID": Cumple con los principios S,O el resto no aplica

"GRASP": Experto, Alta Cohesión, Bajo Acoplamiento

"Author": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Crear los objetos de tipo Author.
- Administrar los datos del autor (nombre, apellido, nombre de usuario y fecha de nacimiento).
- Realizar el análisis del autor.

"SOLID": Cumple con los principios S,O el resto no aplica

"GRASP": Experto, Alta Cohesión, Bajo Acoplamiento

"Phrase": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Crear los objetos de tipo Phrase.
- Administrar los datos de la frase (entidad, estado, texto y fecha).
- Realizar el análisis de la frase.
- Obtener el Texto asociado al estado de la frase.

"SOLID": Cumple con los principios S,O el resto no aplica

"GRASP": Experto, Alta Cohesión, Bajo Acoplamiento

"IRepository": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Indicar el comportamiento de los repositorios que tendrán las clases que implementen esta interfaz.

"SOLID": Cumple con los principios S,I el resto no aplica

"GRASP": Variación Protegida

"EFRepository": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Crear los objetos de tipo EFRepository.
- Almacena los datos de las frases, entidades, autores, sentimientos y alarmas del sistema en la base de datos.
- Guardar los cambios en el repositorio de las entidades (frase, alarma y autores) los datos que fueron modificados durante el análisis.

"SOLID": Cumple con los principios S,O el resto no aplica

"GRASP": Experto, Creador, Alta Cohesión, Bajo Acoplamiento

"IAlarm": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Indicar el comportamiento de las alarmas a utilizar.

"SOLID": Cumple con los principios S,I el resto no aplica

"GRASP": Variación Protegida

"EntityTimeLapseAlarm": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Crear los objetos de tipo EntityTimeLapseAlarm.
- Administra los datos de la alarma (entidad, tiempo que abarca, en que unidad de tiempo: días u horas, naturaleza de la alarma: positiva o negativa, cantidad de posts necesarios y si está activada o no).
- Realizar análisis para ver si la alarma fue activada o no.

"SOLID": Cumple con los principios S,O el resto no aplica

"GRASP": Experto, Creador, Alta Cohesión, Bajo Acoplamiento

"AuthorTimeLapseAlarm": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Crear los objetos de tipo AuthorTimeLapseAlarm.
- Administra los datos de la alarma (tiempo que abarca, en que unidad de tiempo: días u horas, naturaleza de la alarma: positiva o negativa, cantidad de posts necesarios y si esta activada o no).
- Realizar análisis para ver si la alarma fue activada o no.

"SOLID": Cumple con los principios S,O el resto no aplica

"GRASP": Experto, Alta Cohesión, Bajo Acoplamiento

"BusinessLogicController": Esta clase se encuentra dentro del paquete "BusinessLogic" y se encarga de:

- Crear los objetos de tipo BusinessLogicController.
- Administra la instancia de "Repository" que contiene, brindando seguridad, manteniendo separada la lógica y el almacenamiento de datos de las interfaces de usuario.
- Permite a la interfaz de usuario ingresar una nueva entidad al sistema, realizando las validaciones correspondientes para permitirle ser guardada, enviándole a una respuesta de si se guardó realmente o no a la interfaz, para que esta proceda a informarle al usuario.
- Permite obtener la lista de las entidades guardadas en el sistema.
- Permite ingresar nuevas alarmas al sistema y realizar los análisis de estas (luego de ingresar una frase, una alarma o antes de devolver la lista de alarmas cuando es solicitada).
- Permite ingreso y consulta de frases al sistema, realiza también el análisis de las frases luego de ingresado un nuevo sentimiento o entidad.
- Permite agregar nuevos sentimientos, realizando las validaciones correspondientes para permitir el ingreso de dicho dato y devolviendo a la interfaz si se guardó el dato o no, para que esta informe al usuario. También permite obtener los sentimientos guardados o eliminar aquellos que cumplan con la condición de no estar siendo utilizados por ninguna frase.

"SOLID": No cumple con S, cumple con O y el resto no aplican

"GRASP": Experto, Controlador, Alta Cohesión, Bajo Acoplamiento

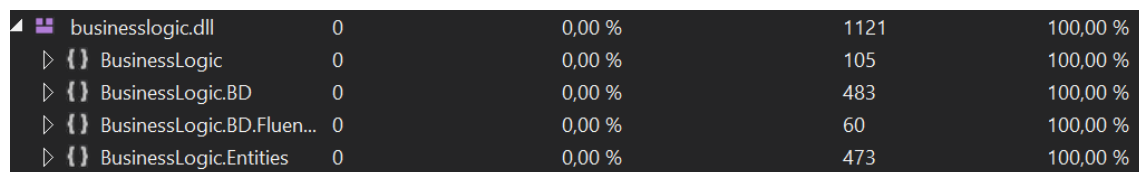
Los criterios usados para la asignación de responsabilidades fueron los siguientes:

Se tomo en cuenta la unicidad de responsabilidades de cada clase sugerida por Clean code, intentando cumplir con esto lo mejor posible y dejando un plan de mejora de la estructura a futuro que se acerque todavía más al objetivo. Para eso hicimos la separación entre las estructuras de datos y la lógica de negocio siendo utilizada en un controlador.

Como se menciona en la sección descripción general se aplicaron los patrones de diseño GRASP y SOLID con los cuales se determino las clases que se utilizarían, sus responsabilidades y la relación entre ellas.

3. Pruebas

3.1. Cobertura de pruebas unitarias



businesslogic.dll	0	0,00 %	1121	100,00 %
BusinessLogic	0	0,00 %	105	100,00 %
BusinessLogic.BD	0	0,00 %	483	100,00 %
BusinessLogic.BD.Fluen...	0	0,00 %	60	100,00 %
BusinessLogic.Entities	0	0,00 %	473	100,00 %

Figura 3.1: Cobertura de pruebas unitarias

Como fue mencionado anteriormente, el desarrollo del dominio del sistema fue realizado utilizando la metodología de "Desarrollo guiado por pruebas", este se basa en realizar los test unitarios y a partir de lo que estos prueban desarrollar el código mínimo y suficiente que los haga pasar, luego se realizan etapas de reestructuración de código.

El desarrollo utilizando esta técnica nos permite detectar fallas en el código de forma temprana, evitar tener código que no se utilice y tener el código testeado al terminar el desarrollo.

Los test dan respaldo al desarrollador que trabaja en el sistema para saber si sus cambios han afectado alguna otra funcionalidad, también sirven a modo de "documentación" de las funcionalidades del sistema; ya que no se centran en la implementación, el cómo, sino que dejan claro qué hace cada método que se esta testeando.

El desarrollo guiado asegura una alta cobertura de las pruebas unitarias y que se pasen el 100 % de los test realizados, debido a que no se pasa al siguiente test hasta que el código realizado logre pasar el actual, y el código para esto siempre debe ser el mínimo necesario. Se puede observar esto en el resultado de nuestras pruebas unitarias, al igual que la vez anterior el porcentaje de cobertura, de la lógica de negocio, se mantuvo en 100 %.

Por lo tanto, el uso de esta metodología de desarrollo nos permitió definir las reglas de negocio que debería tener nuestro sistema, sus posibles variantes y casos límites.

Bibliografía

- [1] Robert Martin. Clean Code. [Online]. Available: <https://github.com/SaikrishnaReddy1919/MyBooks/blob/master/%5BPROGRAMMING%5D%5BClean%20Code%20by%20Robert%20C%20Martin%5D.pdf>
- [2] Paul D. Sheriff, Ken Getz. ASP.NET Developer's JumpStart. [Online]. Available: https://books.google.com.uy/books?id=2NpMDIV_BmQC&lpg=PA599&dq=btn%2C%20pnl%2C%20txt&hl=es&pg=PA600#v=onepage&q&f=false
- [3] Entity Framework Microsoft Documentation. [Online]. Available: <https://docs.microsoft.com/en\protect\discretionary{\char\hyphenchar\font}\us/ef/>
- [4] Entity Framework.