Obligatorio 1 - Diseño de aplicaciones 1

Agustín Hernandorena (233361)
Joaquín Lamela (233375)

Descripción general del trabajo y del sistema

El análisis de sentimiento, es una potente herramienta, que nos permite detectar cómo las personas se expresan respecto a marcas, personalidades públicas, productos, etc.

Esta técnica, consiste en procesar los textos que las personas publican (en redes sociales por ejemplo), con el fin de determinar si el sentimiento hacia una de las entidades es positivo o negativo.

Para realizar este análisis de sentimientos, se desarrolló una aplicación de escritorio (aplicación de Windows Form), utilizando el lenguaje C#, que permite:

1. Registrar y eliminar sentimientos: En esta sección del sistema, se permite al usuario, registrar palabras o una combinación de ellas. Además, deberá indicar, si se trata de un sentimiento negativo o positivo.
   Por último, el sistema muestra una lista con los sentimientos registrados, en donde el usuario podrá eliminar el que desee.

2. Registrar entidades: Se permite al usuario registrar entidades, y se muestra una lista de ellas, en donde, el usuario podrá eliminar la que desee.

3. Ingresar comentario: Se permite al usuario ingresar un comentario, con su correspondiente fecha (ésta no podrá ser menor a un año desde la fecha del sistema ni posterior a la fecha del mismo). Luego del ingreso del comentario, el sistema evalúa con qué entidad está relacionado. También, mediante un análisis se determina el tipo de comentario que se realizó (positivo, negativo o neutro). Un comentario positivo es aquel que tiene al menos un sentimiento positivo y ningún sentimiento negativo, mientras que un comentario negativo es aquel que tiene al menos un sentimiento negativo y ningún sentimiento positivo. Por último está el comentario neutro, que se da cuando hay al menos un sentimiento negativo y un sentimiento positivo, o cuando no hay entidades asociadas a dicho comentario.
   Por otra parte, al ingresar un comentario, se evalúan las alarmas registradas, porque es posible, que a partir de este comentario, se active alguna de las que fueron configuradas previamente.

4. Configurar alarma: A través de esta funcionalidad, se le permite al usuario crear un alarma asociada a una entidad que podrá ser seleccionada a partir de un listado.
   Por otra parte, también el usuario debe definir el tipo de alarma, el cual puede ser positivo o negativo. Esto referenciado con la cantidad de post necesarios para activar, ya que el usuario debe especificar la cantidad de comentarios para que una alarma se active, siendo esta asociada a la entidad y el tipo de alarma seleccionada (ya que nos vamos a fijar únicamente en las frases que sean del tipo que la alarma tiene).
   También dentro de esta sección el usuario debe especificar el plazo del tiempo en el cual la alarma se debe activar. Esto se chequea cada vez que se ingresa un comentario, con el tiempo especificado (horas o días respectivamente) con la fecha actual del sistema.

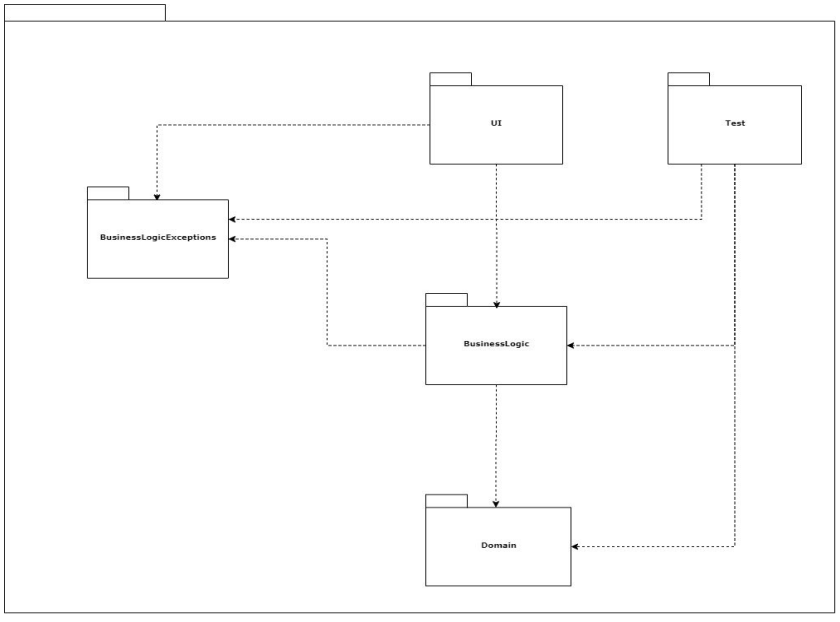
Es decir si se ingresó una alarma asociada a la entidad Coca Cola, con el tipo positivo, la cantidad de post igual a uno, y con un plazo de diez días, si se ingresa el comentario “Me gusta la Coca Cola” con fecha 11/05/20, el sistema verificará 10 días hacía atrás con la fecha actual del PC que se está utilizando determinando si la alarma se activa o no.

5. Reporte de análisis: Permite al usuario visualizar mediante una grilla, un reporte de las frases que fueron analizadas, para ello, para cada frase se muestra: el texto que contiene, la entidad relacionada a ella, la fecha de registro y el tipo (si es positiva, negativa o neutra).

6. Reporte de alarmas: Permite visualizar mediante un listado, un reporte de las alarmas generadas, en donde para cada una de ellas se indica: la entidad asociada, el tipo (positiva o negativa) y por último, si está activa o no.

Una vez especificadas las funcionalidades que nuestro sistema provee al usuario, nos centraremos en describir cómo es la solución que diseñamos.
Es por eso, que en una primera instancia, pensamos en cómo nuestro sistema se divide en agrupaciones lógicas, y la dependencia que existen entre estas. Para modelar estos conceptos, realizamos un diagrama de paquetes, una estructura que nos permite visualizar cómo nuestro sistema está dividido en agrupaciones lógicas y las dependencias entre esas agrupaciones.

Diagrama de paquetes



En particular nuestra solución cuenta de 5 paquetes, siendo ellos UI, Test, BusinessLogicExceptions, BusinessLogic, Domain. Esto a partir de las decisiones de diseños tomadas, las cuales se explicarán más adelante.

En primera instancia, comenzaremos a desarrollar acerca del paquete UI. Este paquete, está compuesto por un único Windows Form, el cual es la ventana de menú principal. También dicho paquete, está compuesto por diferentes controles de usuarios que son utilizados por la ventana de menú principal para brindarle al usuario las diferentes funcionalidades.

La UI, es la interfaz gráfica, presenta el sistema al usuario, le comunica la información y captura la información del usuario en un proceso. Las peticiones que realiza el usuario, deben ser procesadas por otra capa, que al consultar por los datos almacenados en el sistema, podrá dar una respuesta a esta petición. El paquete al que nos referimos, es llamado *BusinessLogic* (reglas de negocio), que es quien determina cómo se crean, almacenan y cambian los datos. A raíz de esto, se puede decir claramente que UI depende de BusinessLogic, lo cual determina que el paquete UI no compile sí no tiene el paquete BusinessLogic.

También, debemos pensar en que el usuario no siempre hará un uso correcto de la aplicación, es decir, puede introducir datos incorrectos, dejar campos sin completar, entre otros. A partir de esto, debemos diseñar una solución que esté preparada para afrontar estos flujos alternativos, y en particular, a considerar la manera en que en estos casos se debe cortar la ejecución del sistema, y mostrar al usuario un mensaje que indique el error que está ocurriendo. Es por esto, que el paquete UI tiene una dependencia de las excepciones de las reglas de negocio (*BusinessLogicException*), es decir, sabe cuándo lanzar una excepción y que mensaje mostrar al usuario gracias a la información que obtiene de BusinessLogicException.

Este paquete, está compuesto por elementos que se encargan del manejo de excepciones, es decir, de controlar los errores ocasionados durante la ejecución del sistema.
Ante la ocurrencia de un error, el sistema haciendo uso de estos elementos, reacciona cortando el flujo de ejecución y ejecutando un fragmento de código, que permite mostrar al usuario un mensaje de error.

Como nombramos anteriormente tenemos el paquete de las reglas de negocio, el cual está compuesto por diferentes manejadores. Nuestra solución está compuesta por seis manejadores, y dentro de cada uno de ellos se implementa cuáles serán las reglas de negocio adoptadas por el sistema.

Centrándonos en el paquete BusinessLogic, el mismo no tendría sentido si no tuviera una dependencia con el paquete Dominio, ya que claramente las reglas de negocio necesitan y utilizan clases del dominio para determinar cuáles son los comportamientos que deberían haber dentro del sistema.

También, las reglas de negocio dependen de las excepciones de las reglas de negocio, ya que es necesario que ante la ocurrencia de un error en ejecución, se capturen y se le notifique al usuario qué fue lo que ocurrió. Es por esto que necesariamente las reglas de negocio utilizan al menos una de las clases que se encuentran dentro de las excepciones.

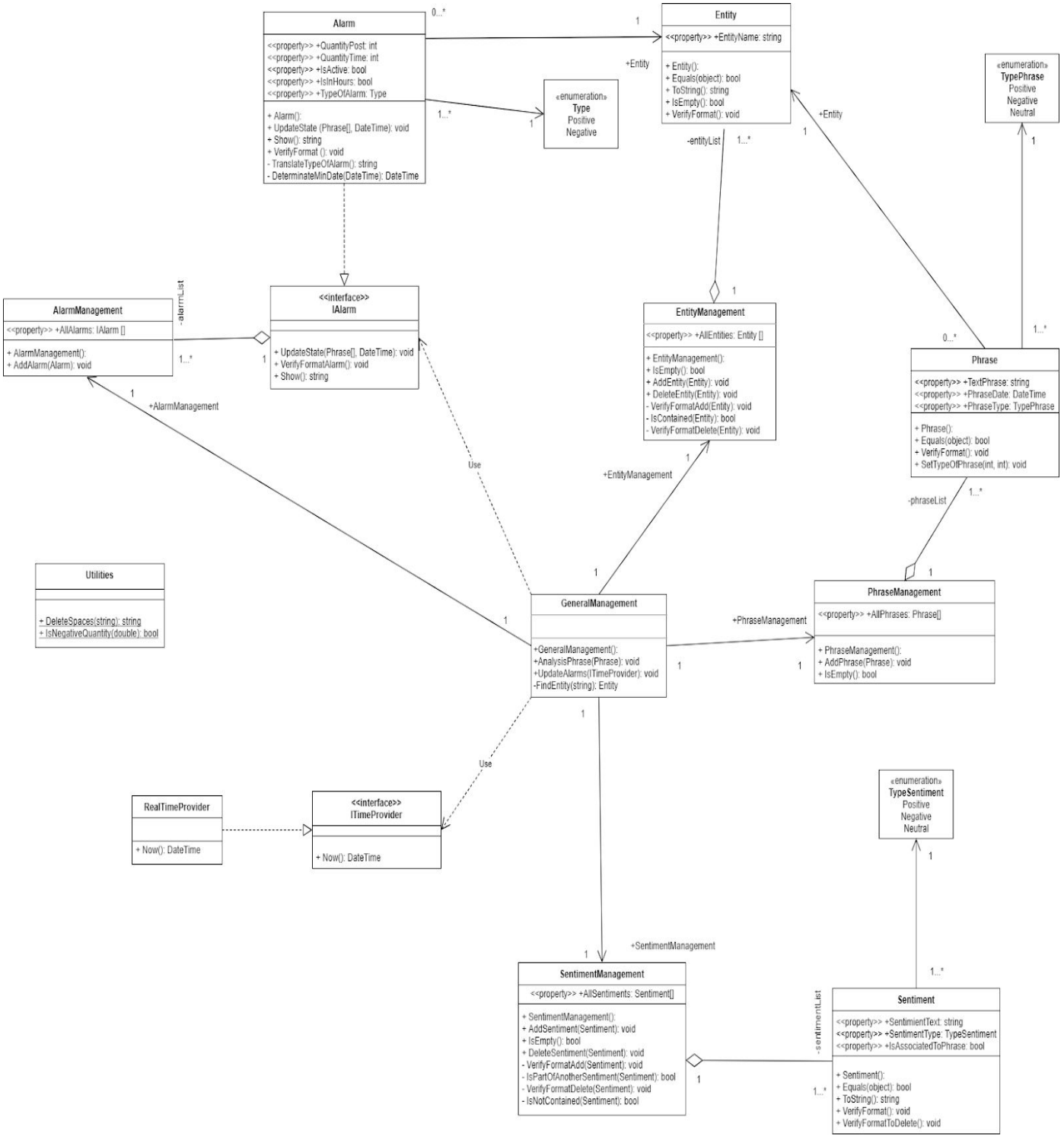
Otro paquete que se encuentra dentro de nuestra solución, es el paquete de Test, el mismo tiene una dependencia de las reglas de negocio, del dominio y de las excepciones de las reglas de negocio. Esto debido a que dentro del paquete Test, están todos los casos de prueba, debido a que los test son el proceso de ejecutar el software con el objetivo de encontrar defectos. La dependencia del paquete Test con los otros tres paquetes se nota con facilidad, ya que para que se lograra introducir código en dichos paquetes lo primero que se realizaron fueron los Test, esto siguiendo la metodología TDD.

Por último, se tiene el paquete dominio, del cual dependen el paquete Test, BL, UI. En él, se encuentran las entidades que maneja nuestro sistema, por lo cual es la parte fundamental del mismo, ya que sin él no habría sentido alguno en la aplicación. Pero a su vez el paquete dominio depende de las excepciones de las reglas de negocio, ya que son necesarias para saber si el formato de un objeto es válido, entre otras características.

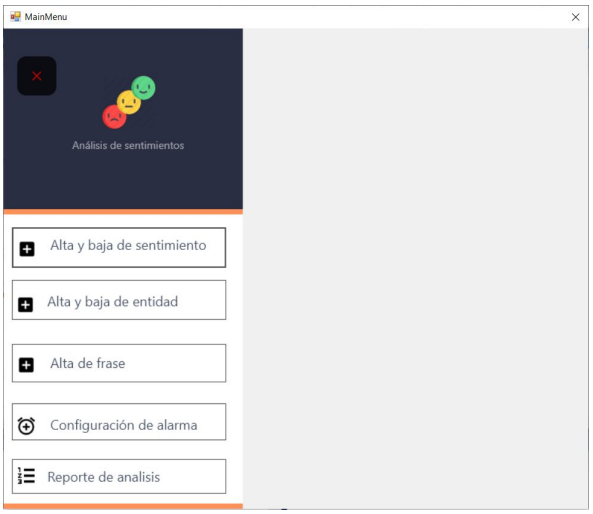
A su vez, cada uno de estos paquetes está compuesto por elementos, en este caso, clases.

Con el fin de crear una estructura que incluya las clases del sistema, sus atributos, operaciones (o métodos) y las relaciones entre los objetos, realizamos el siguiente diagrama de clases.

Diagrama de clases



En particular comenzaremos a detallar el funcionamiento básico de la aplicación. Todo comienza cuando el usuario inicia la aplicación. A partir de este momento, el mismo visualiza la ventana principal de la aplicación la cual posee diferentes funcionalidades. Observándose de la siguiente forma:



Siendo así que una vez observándose la ventana principal, se puede notar que el menú se encuentra del lado izquierdo, tomando la decisión de diseño que lo que se utilizaría para mostrar las diferentes funcionalidades que el sistema posee serían controles de usuarios totalmente independientes. Es así que el espacio que se encuentra vacío en el lado derecho del menú, se trata de un panel en el cual se despliega cada uno de los controles de usuario creados, una vez que se hace click en algunos de los botones.
En particular se cree que la mejor opción era utilizar diversos UserControl para que a los usuarios no se le abrieran nuevas ventanas, de forma que se mantuviera todo en una misma ventana. Es decir que cuando un usuario hiciera clic en algunos de los botones se mostrará directamente, sin necesidad de que al usuario se le abriera una nueva ventana.
La virtud principal consiste en expandir aplicaciones y generar una interfaz de usuario enriquecida, más rica y atractiva.

Otra característica orientada a la interfaz, fue la necesidad de decidir que el tamaño de la misma fuese fija y que el usuario no pudiera ni maximizar ni minimizar. Destacándose que es una característica arriesgada, por el impacto que le puede producir al usuario. En particular está decisión está basada en que la aplicación no tenga defectos gráfico. Con defectos gráficos, lo que se logra es causar un grado de insatisfacción al usuario al utilizar la aplicación, lo cual puede llevar a que utilice a la competencia por características de este estilo. De manera que los defectos que se podrían encontrar si se permitieran las características anteriormente nombradas, son cómo por ejemplo:

1. Imágenes pixeladas por maximizar o minimizar.
2. Ubicaciones erróneas de objetos de la interfaz por maximizar o minimizar.
3. Espacios en blanco por los tamaños de los objetos.

Por otra parte, para vincular la forma en que la interfaz se asocia con el dominio y las reglas de negocio, comenzaremos a detallar cómo cada uno de los User Control se vincula con dichas reglas. Es por esto que cada uno de los controles de usuario, poseen un objeto del tipo GeneralManagement. Este tipo de objeto, lo que nos permite es conectar a la interfaz de usuario con lógica de negocios, y hacer uso de las operaciones que ella presenta.

Es por esto, que como se podrá notar en su nombre posee el término “Management”. Haciendo énfasis en esto debemos especificar que una de las decisiones tomadas fue evitar el uso de una única clase conocida como “System” o “Sistema” en español. Por lo cual para cada una de las entidades que se encuentran en Dominio, se le realizó un manejador. Esta decisión fue tomada, en lo que expone el autor Robert C. Martín en su libro Clean Code, en la sección de Responsabilidad Única del capítulo 10[1].

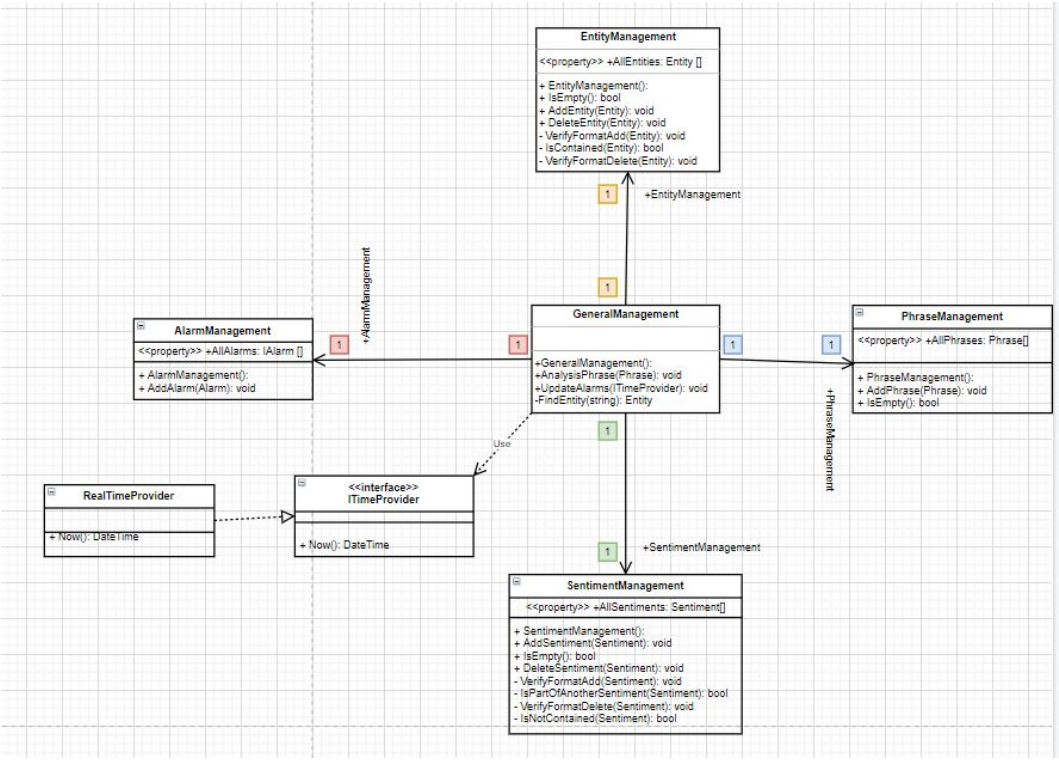
Martín nos dice que una clase debe tener uno y solo un motivo para cambiar. ¿A qué nos referimos con esto?

En particular sí se desarrollara una única clase Sistema, la cual tuviera múltiples listas y múltiples métodos públicos, según el principio que describió Martín, lo que sucedería es que dicha clase tendrá múltiples responsabilidades, describiendo dichas responsabilidades serian:

1. Se encargaría de verificar el formato de cada uno de los objetos que se crean.
2. Se encargaría de agregar a las diferentes listas los diferentes tipos de objetos creados.
3. Se encargaría de eliminar a los objetos de las diferentes listas.
4. Se encargaría de “tirar” excepciones una vez verificado el formato.

Entonces la justificación en la que nos basamos para no utilizar una única clase la cual es responsable de manejar todo, es que la misma tendría múltiples responsabilidades ya que tiene diferentes tipos de objetos asociados. Es por esto que decidimos crear para cada una de las clases del dominio un manejador, el cual la única responsabilidad que tiene, es manejar los objetos del tipo del objeto del Dominio.

Por último lo que también realizamos fue un manejador general, el cual se encarga de instanciar a los demás manejadores. Observándose esto en la figura que se presenta a continuación:



[1] Interpretado de: Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education. Página: 132-133. Página: 138.

Otra de las características consideradas fue que dentro de cada una de las reglas de negocio se tomaron diversas decisiones de cómo manejar dichas funcionalidades. En particular una de las decisiones tomadas, fue que cuando un usuario nos ingresa una frase, entidad o sentimiento con diversos espacios entre las palabras que la componen, dichos espacios se reducen a un único espacio. Un ejemplo de esto se da cuando el usuario nos ingresa un sentimiento de tipo positivo, con la forma “Me gusta”, se puede notar que dicho sentimiento entre las palabras que lo componen tienen más de un espacio, entonces lo que nuestro sistema hace es reducir dichos espacios, quedándonos con el sentimiento de la siguiente forma: “Me gusta”.
En este caso se arregla de esta forma, por una cuestión de estética y gramática, ya que no tiene sentido ingresar sentimientos/frases/entidades con espacios innecesarios.

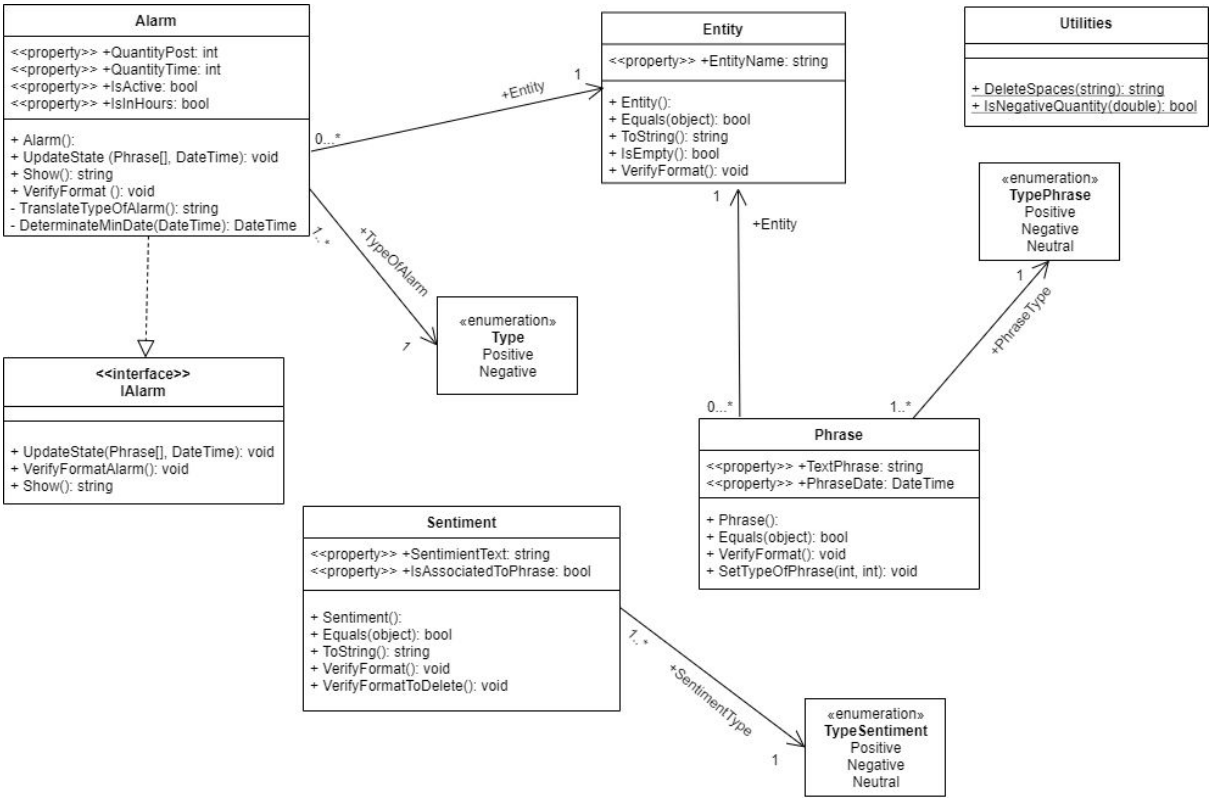
También otras de las decisiones que fueron tomadas, se relaciona con los sentimientos. En particular a la hora del registro de los mismos, un sentimiento no puede estar contenido en otro ya ingresado en el sistema. O que alguno de los sentimientos que se encuentran ingresados en el sistema no se encuentren contenidos dentro del sentimiento a ingresar. Por ejemplo: si un usuario intenta registrar el sentimiento negativo “No me gusta”, y dentro del sistema se encuentra registrado el sentimiento positivo “Me gusta”, el sistema no permite ingresar dicho sentimiento, ya que me gusta está contenido dentro de no me gusta. El otro ejemplo es el caso inverso, cuando queremos registrar el sentimiento positivo “Me gusta”, y dentro del sistema está registrado el sentimiento negativo “No me gusta”, el sistema no permite registrar dicho sentimiento ya que me gusta está contenido dentro de no me gusta. La idea principal de esta decisión es facilitar la implementación y la complejidad de aceptar o no un sentimiento que el sistema debe tomar.

También, con respecto a los sentimientos, con el fin de disminuir la complejidad, se tomó la decisión de que el sistema solo permita eliminar aquellos que no estén contenidos en ninguna frase, porque de lo contrario, al eliminar un sentimiento que esté asociado a una o más frases, se debería volver a analizar las frases asociadas a dicho sentimiento y eventualmente podrían cambiar su tipo.

Por otro lado, con respecto al ingreso de una entidad, únicamente verificamos que la entidad que se ingresa ya no esté ingresada en el sistema (para que no quede duplicada), pero no consideramos apropiado chequear que la entidad a ingresar no esté contenida en otra porque estaríamos acotando el uso de la aplicación, teniendo en cuenta en que hay efectivamente nombres de marcas que están contenidos dentro de otros.
Por ejemplo: si se registra la empresa Discovery Kids, debe poder registrarse posteriormente la empresa Discovery.
Si estuviéramos chequeando la condición de que no es posible registrar una entidad que esté contenida dentro de otra, en este caso Discovery, no podría registrarse en el sistema.

Un último punto importante que involucra tanto a las reglas de negocio como a la interfaz gráfica, está relacionado con las frases. En particular se optó porque la fecha mínima para el ingreso de una frase que un usuario puede seleccionar es únicamente un año hacía atras, basicamente se considera esto con respecto a la fecha actual de nuestro sistema, solamente con un año menos. Por otra parte la fecha máxima que puede seleccionar un usuario es la actual del sistema.

A partir de lo mencionado anteriormente en el diagrama de paquetes, las reglas de negocio tienen una dependencia con respecto al dominio.
Este, está compuesto por entidades que nos permiten modelar el problema a resolver, para esto, consideramos en el paquetes las clases: Entity, Alarm, Sentiment, Phrase, una interface IAlarm cuyo propósito se explicará más adelante, y tres enumerados que nos permiten manejar los tipos de alarmas, sentimientos y frases.
Las clases presentes en este paquete, así como las relaciones que existen entre ellas, se pueden visualizar en el siguiente diagrama de clases.



En cuanto al modelado de las clases que componen a este paquete, consideramos apropiado seguir un modelo de dominio no anémico, porque creemos necesario que los objetos del dominio deben tener cierta lógica, es decir, un objeto debería saber si su formato es el correcto, mostrar su estado, en el caso de las alarmas saber actualizar el estado.

Utilizando un modelo de dominio anémico, va en contra un poco de la idea básica del diseño orientado a objetos, el cual es combinar los datos y comportamientos en una única unidad, ya que en este modelo se tratan de representar los objetos del dominio pero sin comportamiento.[2]

De forma que en las clases de dominio no hubiese código duplicado, dicho paquete, cuenta con una clase llamada Utilites, la cual contiene métodos genéricos que son utilizados en las diferentes clases manejadores y algunos métodos en particular de las clases del dominio (como por ejemplo el método equals), por ello, preferimos colocar todos esos métodos como estáticos en una clase aparte, para no tener que escribirlos en cada clase en que vayan a ser utilizados, evitándose así el código duplicado.

[2] Fowler, M. (2003, noviembre 25). AnemicDomainModel. Recuperado 19 de mayo de 2020, de https://martinfowler.com/bliki/AnemicDomainModel.html

En los requerimientos del sistema, se indica que en el futuro podrán existir diferentes tipos de alarma. A raíz de esto, consideramos apropiado realizar una interface IAlarm, que es quien describe el comportamiento que toda alarma debe cumplir: actualizar su estado y mostrar su información.
El hecho de tener una interface para las alarmas, nos permite mediante polimorfismo, definir distintos comportamientos para un conjunto de métodos dependiendo de la clase sobre la que se realice la implementación.
En el futuro, podrá crearse otra clase de alarma, diferente a la actual, que implementará la interface IAlarm, y podrá definir su propio comportamiento con respecto a como actualizar su estado y la forma en que muestra su información.
Esto también nos permite diferenciar en la interfaz gráfica del usuario, la manera en que se visualiza la misma, dependiendo del tipo de alarma seleccionada, cómo se va a visualizar dentro de la interfaz.

Con respecto a los tests, la aplicación se desarrolló utilizando TDD (desarrollo guiado por pruebas), para ello consideramos apropiado realizar una clase de test para cada clase manejadora del paquete *BusinessLogic,* con el objetivo de que los tests que prueban operaciones de una determinada clase estén todos juntos, y que sean independientes de otros que prueban otros tipos de objetos.
En particular se tomó esta decisión para seguir las características que deben poseer las pruebas unitarias, conocido como principio FIRST, definido por el autor Robert Cecil Martín[3].
El principio FIRST, es el acrónimo de las cinco características que deben tener nuestros tests unitarios para ser considerados tests con calidad. Siendo las características:

1. F (fast-rápido): posibilidad de ejecutar un gran número de tests en cuestión de segundos.

2. I (independent-independiente): Todas las pruebas unitarias deben de ser independientes de las otras. En el momento que un test falla por el orden en el que se ha ejecutado, este test está mal desarrollado. El resultado no debe verse alterado ejecutando los tests en un orden y otro o incluso de forma independiente.

3. R (repeatable-repetible): El resultado de las pruebas debe ser el mismo independientemente del pc/servidor/terminal en el que se ejecute.

4. S (self-validating- auto evaluable): La ventaja de las pruebas automatizadas es que podemos ejecutarlas simplemente al pulsar un botón o incluso hacer que se ejecuten de forma automática tras otro proceso.

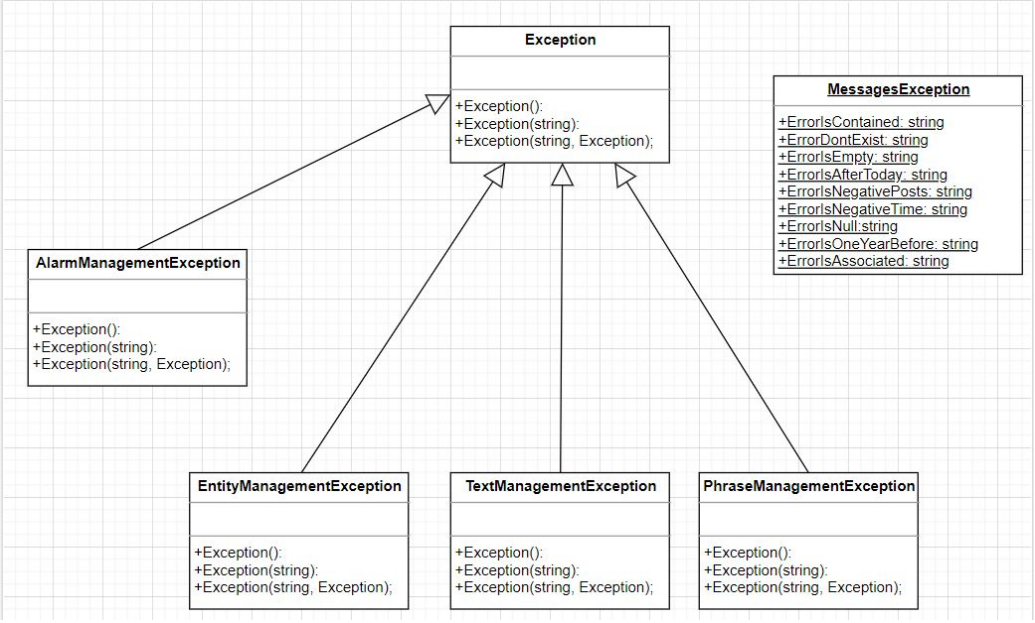
5. T (timely-oportuno): Las pruebas unitarias deben escribirse justo antes del código de producción que las hace pasar. Sí se escribe pruebas después del código de producción, se puede encontrar que el código de producción es difícil de probar.

[3] Interpretado de: Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education. Página: 132-133.

En cuanto al manejo de errores, preferimos utilizar excepciones frente a devolver códigos de error. Nuestra preferencia se basa en que si utilizamos códigos de error, el invocador debe procesar el error de forma inmediata, además de generarse una gran estructura de condicionales anidados, que hacen al código más complejo y difícil de entender.

En cambio, si utilizamos excepciones, el código de procesamiento del error se puede separar del código de ruta (de la función en sí), haciendo que el código sea más sencillo de comprender.

Para el manejo de excepciones, contamos con un paquete llamado *BusinessLogicException*, el cual está compuesto por cinco clases, en donde cuatro de ellas fueron creadas para lanzar excepciones relacionadas a: alarmas, entidades, frases y sentimientos, y la restante, está compuesta por variables estáticas que contienen mensajes de error que son utilizados a la hora de lanzar una excepción.



En primer lugar, decidimos crear una clase de excepción para cada clase manejadora del paquete *BusinessLogic* ya que consideramos que de esta forma, las excepciones quedan bien definidas para cada clase, y evitamos a la hora de realizar un catch, capturar una excepción genérica que nos impide saber realmente cuál fue la causa por la que se lanzó la excepción. En cambio si definimos una clase de excepción para cada clase manejadora, podremos capturar excepciones más específicas, y saber con mayor certeza, cuál fue la causa que originó el error en ejecución.

Decidimos crear una clase que contiene variables estáticas que representan el mensaje de error, pensando en que si en un futuro esos mensajes habría que cambiarlos por otros, simplemente habría que modificar en un solo lugar. Diferente sería, si no usamos constantes, en donde ante la ocurrencia de un cambio, sería necesario cambiar el valor en todos los lugares en que fue utilizado.

De forma que para organizar el trabajo con ramas usamos *GitFlow.* En este flujo de trabajo se utilizan dos ramas principales:

- Rama Master: Cualquier commit que pongamos en esta rama debe estar preparado para subir a producción.

- Rama Develop: Rama en la que está el código que conformará la siguiente versión planificada del proyecto.

Luego, contamos con ramas auxiliares: feature.

En nuestro caso, cada vez que implementamos un cambio en el sistema, abrimos una nueva rama *feature* a partir de *develop*. Luego de haber implementado todos los cambios en esa rama, se hace un merge de la rama sobre *develop*, donde se integrará con las demás funcionalidades.

Se puede acceder al repositorio en línea mediante el siguiente enlace:

- **https://github.com/ORT-DA1/233375-233361-Lamela-Hernandorena**

Utilizando esta metodología de trabajo, debemos mencionar que cometimos algunos errores, específicamente en los siguientes commits:

1. Adding the class Utilities because in the last commit don´t go commit: Este commit se realizó producto que en el commit previo, no se compilo la aplicacion, y por tanto, no se generaron archivos de compilación necesarios, para que cuando el otro desarrollador haga un pull de esa rama vea la nueva clase que se había agregado (Utilites).

2. Change tests for Entity Management: Este commit se realizó, puesto que en el commit previo, se cometió un error al subir los cambios, no quedando los tests para Entity Management, ocasionado que el sistema no compile.

3. Adding more test for equals method because CollectionAssert didn´t use: En este commit se prueba exclusivamente métodos Equals(), porque creímos que estos métodos ya estaban cubiertos al realizar tests que utilicen *CollectionAssert*, luego analizando, notamos que CollectionAsert no invoca a Equals(), por lo que tuvimos que probar los Equals() por separado para mantener el nivel de cobertura.

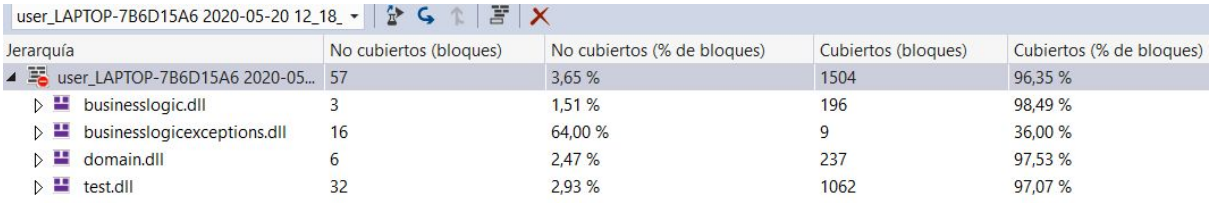
Coberturas de pruebas unitarias

La aplicación fue desarrollada siguiendo TDD (desarrollo guiado por pruebas). Esta práctica consiste en escribir antes la prueba que la funcionalidad. Para esto, en primer lugar, se escribe una prueba y se verifica que la nueva prueba falle. Luego, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito.

Utilizando esta práctica, se alcanza un porcentaje de cobertura de pruebas unitarias muy bueno. En el caso de nuestra aplicación, el porcentaje de coberturas de pruebas es de 98.49% para el paquete *BusinessLogic*, y 97.53 % para *Domain*.

La razón por la cual no llega a 100 % en el paquete Logic, es que no fueron probados, algunas secciones de métodos equals, específicamente en aquellos en que se recibe un objeto NULL por parámetro, o un objeto cuyo tipo no es igual al de la clase en que está el método equals.
En cuanto al paquete Domain, no fue probada la clase RealTimeProvider, porque simplemente contiene un método que nos da la fecha actual. Específicamente para realizar pruebas se debe utilizar la clase MockedDateTime, con el objetivo, de que la fecha y hora utilizada en las pruebas no depende la hora actual de la máquina, evitando el hecho de que las pruebas pasen a una cierta hora del día, y a otra hora no.

La evidencia del porcentaje de cobertura, se puede observar en la siguiente figura.


user_LAPTOP-7B6D15A6 2020-05-20 12_18_

Jerarquía	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% de bloques)
user_LAPTOP-7B6D15A6 2020-05...	57	3,65 %	1504	96,35 %
businesslogic.dll	3	1,51 %	196	98,49 %
businesslogicexceptions.dll	16	64,00 %	9	36,00 %
domain.dll	6	2,47 %	237	97,53 %
test.dll	32	2,93 %	1062	97,07 %

Casos de prueba

Hay que tomar en cuenta, como ya se ha hablado anteriormente, que el objetivo de nuestro sistema está basado en el análisis de sentimientos de las frases. Es por esto que las dos grandes funcionalidades que posee, son el registro de las frases y el reporte de alarmas generadas. Es por esto que para comenzar hablaremos en dos secciones diferentes acerca del registro y del reporte de alarmas.

En particular el registro de las frases, engloba tener previamente registrado o asociado dentro del sistema al menos un sentimiento (sin importar el tipo del mismo) y una entidad, para así poder hacerle el análisis correspondiente a dicha frase. Esto debido a que si no hay previamente registrados sentimientos ni entidades, la frase queda de tipo neutral y no asociada a ninguna entidad.

Es por esto que se puede decir que es una de las funcionalidades claves dentro del comportamiento del sistema, ya que sin un correcto análisis de las mismas, no habría un análisis de sentimiento ni se generarían reportes de alerta. De manera que se trabajará con casos de prueba para probar y verificar el correcto comportamiento de dicha funcionalidad.
Esto nos permitirá asegurar un sistema correcto, eficiente e íntegro.

Especificando acerca de cómo será el formato de los casos de prueba, se realizarán distintas tablas, en las cuales se especificara el accionar del sistema respecto a los distintos comportamientos que puede haber. Es decir frente a las entradas que nos ponen los usuarios, determinar si se tratan de datos válidos o inválidos, si corresponde seguir un curso normal o un curso alternativo, dependiendo del ingreso del usuario.

En general, un caso de prueba especifica el comportamiento de un sistema (o de una parte del mismo). En el caso de prueba se ejecutan un conjunto de acciones con la finalidad de obtener un resultado observable y analizable.
Esto para comprobar el comportamiento del sistema, dependiendo la entrada ingresada y la salida obtenida, contra los requerimientos solicitados.

Un último punto importante a destacar, es que hay pruebas las cuales se encuentran simplificadas por cómo fue implementado el sistema, por ejemplo: el ingreso de la fecha de una frase, está simplificada en las reglas de negocio y también en la UI, ya que desde las reglas se controla que la fecha se encuentre un año menos a la fecha actual y la fecha actual (es decir si estamos en la fecha 17/05/20, el usuario puede ingresar una frase con fecha dentro del rango 17/05/19-17/05/20). Está caracteristica también se implemento en la interfaz, ya que las fechas disponibles qué le aparecen al usuario para seleccionar es la fecha actual menos un año, hasta la fecha actual del día.

Insertar frase

En particular esta sección se basa en el ingreso de la frase. El ingreso de una frase conlleva a que luego de agregar dicha frase al sistema, el mismo le realiza un análisis correspondiente el cual determina entidad y tipo de frase (positiva, neutra o negativa). Luego de realizar el análisis de la frase, lo que nuestro sistema verifica es la generación de alguna alerta, esto sí previamente al ingreso de la frase existe ya una alarma configurada. Pero particularmente de la generación de alarmas se desarrollará en la próxima sección.

Escenario	**Nombre**	**Curso alternativo**
Escenario 1	Ok	
Escenario 2	Texto inválido	C.A 2.1
Escenario 3	Fecha invalida	C.A 3.1

Caso de prueba	**Escenario**	**Texto**	**Fecha**	**Resultado esperado**	**Resultado obtenido**
1.1	1	V	V	Sistema registra la frase	OK
2.1	2	V	NV	El calendario no permite seleccionar fecha fuera de rango	OK
3.1	2	NV	V	Mensaje de error, no se registra la frase	OK

Condición	Clase válida	Clase inválida
Frase	String no vacío	String vacío
Fecha	Fecha comprendida entre fecha actual y un año hacia atrás	Fecha fuera de rango, es decir, menor a un año atrás, o superior a la fecha actual

Antes de realizar las siguientes pruebas, se registraron en el sistema las entidades: *Coca Cola* y *Pepsi*, y los sentimientos: *Me gusta*, *me encanta, odio*, *detesto*, *lo peor*.
La fecha en que se realizan estas pruebas es: 17/05/2020

CP	Input	Resultado esperado	Resultado obtenido
1.1	“Me gusta Coca Cola”	Frase ingresada, positiva para *Coca Cola.*	Ok
1.2	“Pepsi es lo peor”	Frase ingresada, negativa para *Pepsi.*	Ok
1.3	“Coca Cola”	Frase ingresada, neutra para *Coca Cola*.	Ok
1.4	“Me gusta la Coca Cola y la Pepsi”	Frase ingresada, positiva para *Coca Cola.*	Ok
1.5	“Me gusta y a la vez detesto la Pepsi”	Frase ingresada, neutra para *Pepsi.*	Ok
1.6	“Me gusta y me encanta la Coca Cola.”	Frase ingresada, positiva para Coca Cola.	Ok
1.7	“Odio, detesto, es lo peor la Coca Cola, pero igual me gusta”	Frase ingresada, neutra para Coca Cola.	Ok
1.8	“PEPSI y Coca Cola, son lo peor, pero me encanta, y me gusta.”	Frase ingresada, neutra para *Pepsi*.	Ok
1.9	“Coca Cola y pepsi, son lo peor, pero me encanta, y me gusta.”	Frase ingresada, neutra para *Coca Cola*.	Ok
2.1	“”	Frase no ingresada, el sistema muestra error que el texto no puede ser vacío.	Ok
2.2	“ “	Frase no ingresada, el sistema muestra error que el texto no puede ser vacío.	Ok
3.1	“Me gusta Coca Cola.” Fecha: 17/05/2020	Frase ingresada, positiva para *Coca Cola*, con fecha: 17/05/2020	Ok
3.2	“Me gusta Coca Cola.” Fecha: 16/05/2019	La interfaz no permite seleccionar esta fecha (está fuera de rango).	Ok
3.3	“Me gusta Coca Cola.“ Fecha: 18/06/2020	La interfaz no permite seleccionar esta fecha (está fuera de rango).	Ok

Generación de alertas

Cómo mencionamos anteriormente, se puso foco en dos secciones. En particular nos centraremos en la generación de alertas, es decir el registro de una frase conlleva y puede desatar en la activación de una alarma, lo cual se conoce como una alerta. En particular como mencionamos anteriormente, luego de realizado un registro, lo que nuestro sistema realiza es una verificación de la activación de cada una de las alarmas que se hayan configurado previamente al ingreso de dicha frase.

Para este análisis de generación de alertas, se contemplaran casos en los cuales se vean o no afectas las alarmas ya previamente configuradas en el sistema, suponiendo que todas las frases ingresadas seguirán un curso normal (es decir, se lograron ingresar correctamente). Realizando esto para ver el cambio de comportamiento que tienen las alarmas cuando se activan o no.

Para realizar estos casos de prueba, previamente se registran en el sistema: el sentimiento positivo “me gusta” y el negativo “odio”, y las entidades: “Mc Donalds”, y “Burger King”.

Se crea una alarma para la entidad Mc Donalds, que tiene como condiciones de activación: 1 post negativo respecto a esta entidad en las últimas 2 horas.

Se crea una alarma para la entidad Burger King, que tiene como condiciones de activación: 2 posts positivos respecto a esta entidad en el último dia.

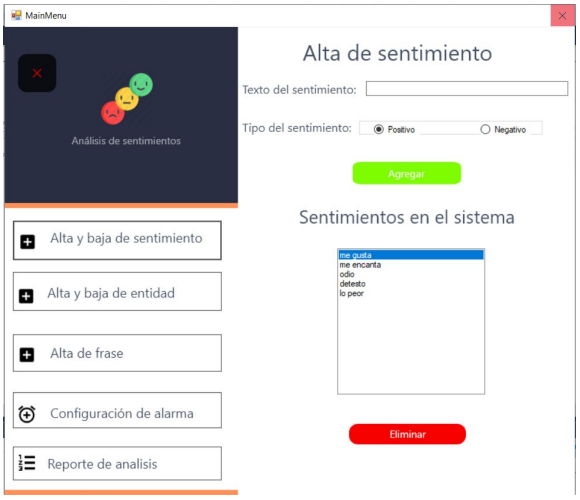
Caso de prueba	Frase	Resultado esperado	Resultado obtenido
1	“Odio Mc Donalds”. Fecha: 17/05/2020, 16:03	Se activa la alarma correspondiente a la entidad Mc Donalds.	Ok
2	“Me gusta Burger King”. Fecha: 17/05/2020, 18:10	Se desactiva la correspondiente a Mc Donalds.	Ok
3	“Me gusta demasiado Burger King” Fecha: 17/05/2020, 20:05	Se activa la alarma correspondiente a Burger King.	Ok
4	“Odio Mc Donalds y me gusta Burger King” Fecha: 18/05/2020, 21:05	Se desactiva la alarma correspondiente a Burger King.	Ok
5	“Mc Donalds”. Fecha: 18/05/2020, 21:30	No cambia el estado de las alarmas.	Ok
6	“Me gusta Burger King y Mc Donalds”. Fecha: 18/05/2020, 21:40	No cambia el estado de las alarmas.	Ok
7	“Me gusta mucho Buger King”. Fecha: 18/05/2020,	Se activa la alarma para Burger King.	Ok

	21:45		
8	“Coca Cola”. Fecha: 19/05/2020, 22:00	Se desactiva la alarma para Burger King.	Ok

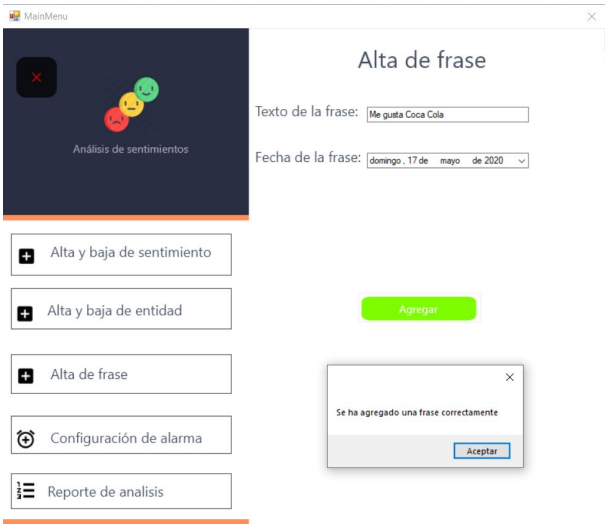
Anexo

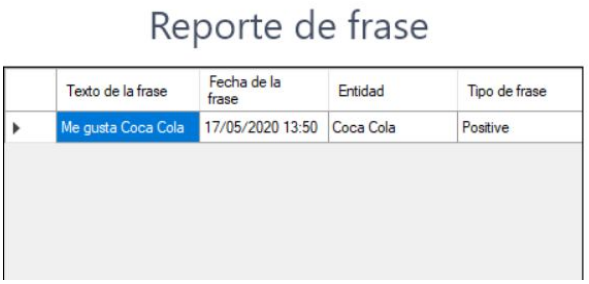
Ingreso de frases

Se asume ya ingresados los sentimientos y las entidades, mostrandose de la siguiente forma:

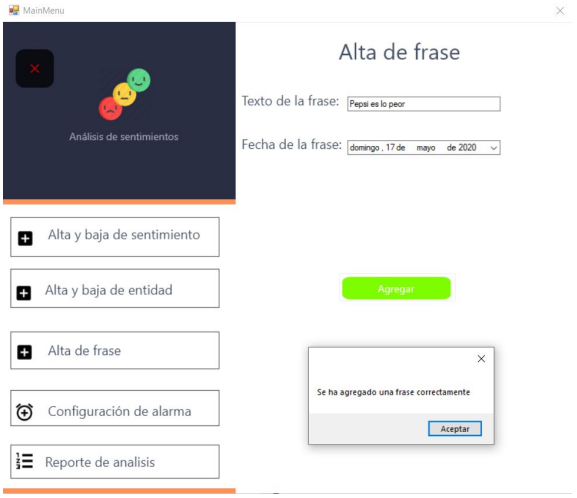


Caso de prueba 1.1:

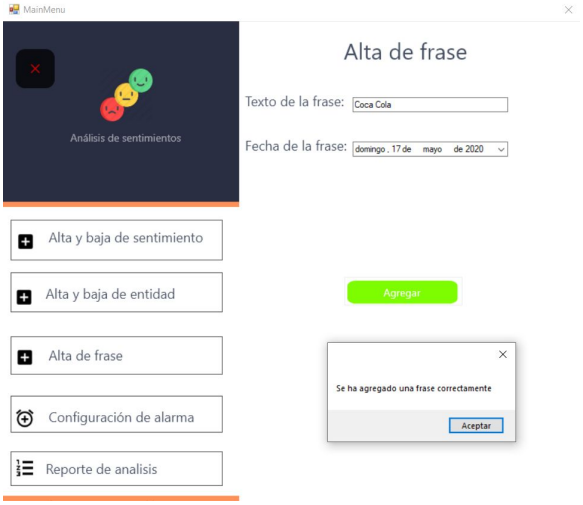




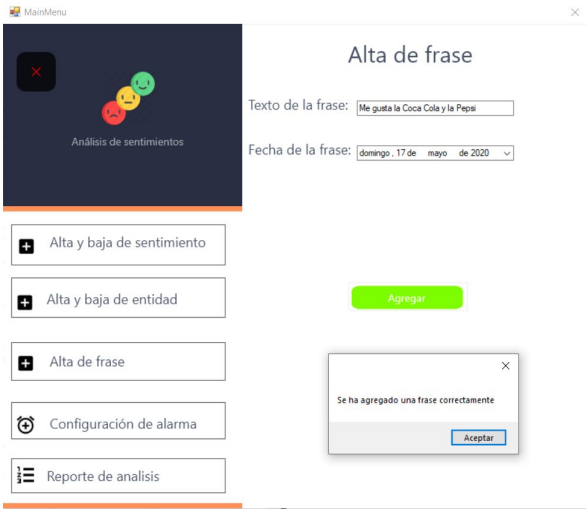
Caso de prueba 1.2



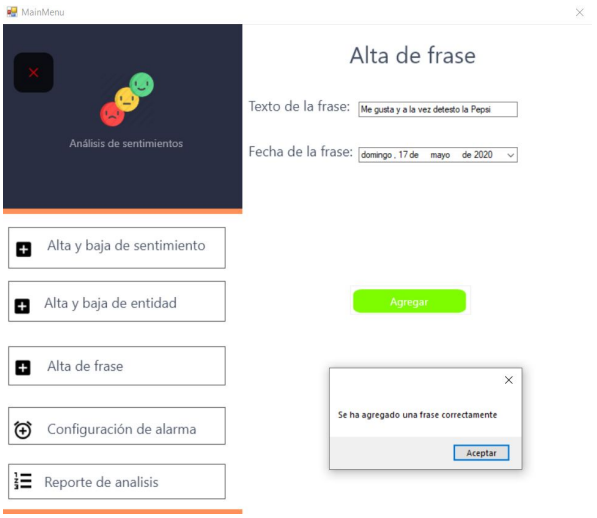
Caso de prueba 1.3:



Caso de prueba 1.4:

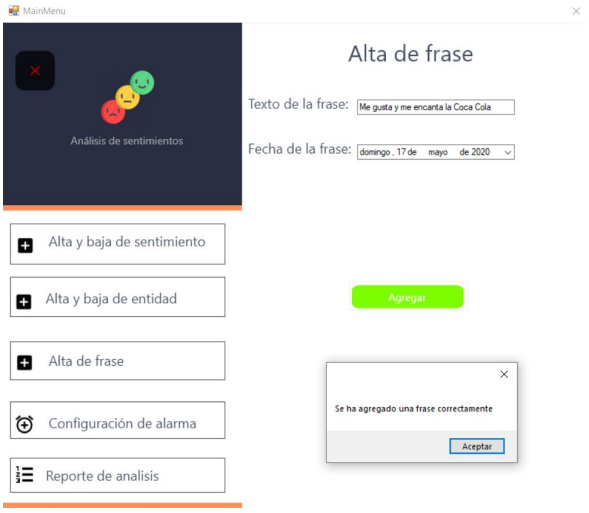


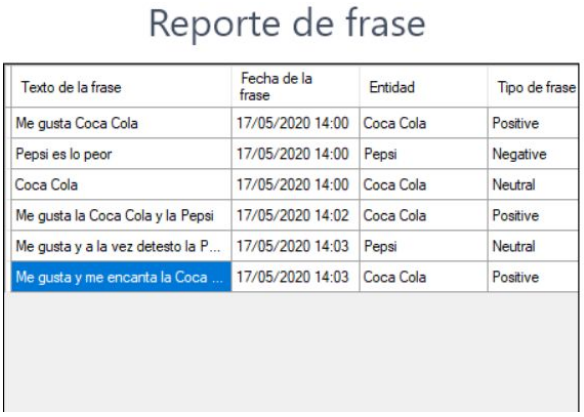
Caso de prueba 1.5:



Texto de la frase	Fecha de la frase	Entidad	Tipo de frase
Me gusta Coca Cola	17/05/2020 13:50	Coca Cola	Positive
Pepsi es lo peor	17/05/2020 13:51	Pepsi	Negative
Coca Cola	17/05/2020 13:52	Coca Cola	Neutral
Me gusta la Coca Cola y la Pepsi	17/05/2020 13:53	Coca Cola	Positive
Me gusta y a la vez detesto la ...	17/05/2020 13:56	Pepsi	Neutral

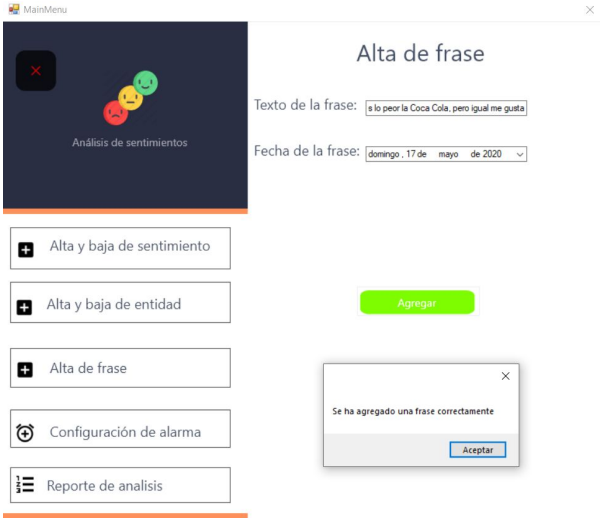
Caso de prueba 1.6:

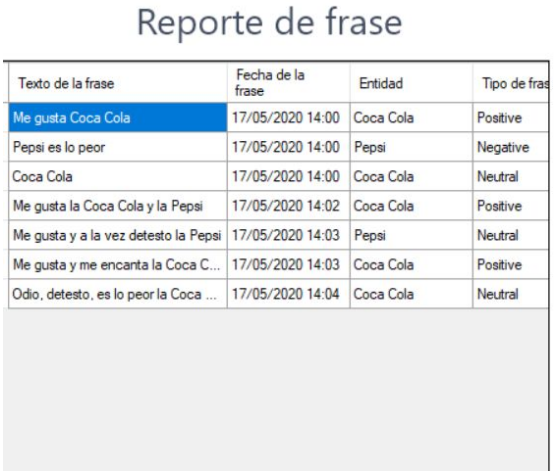




Texto de la frase	Fecha de la frase	Entidad	Tipo de frase
Me gusta Coca Cola	17/05/2020 14:00	Coca Cola	Positive
Pepsi es lo peor	17/05/2020 14:00	Pepsi	Negative
Coca Cola	17/05/2020 14:00	Coca Cola	Neutral
Me gusta la Coca Cola y la Pepsi	17/05/2020 14:02	Coca Cola	Positive
Me gusta y a la vez detesto la P...	17/05/2020 14:03	Pepsi	Neutral
Me gusta y me encanta la Coca ...	17/05/2020 14:03	Coca Cola	Positive

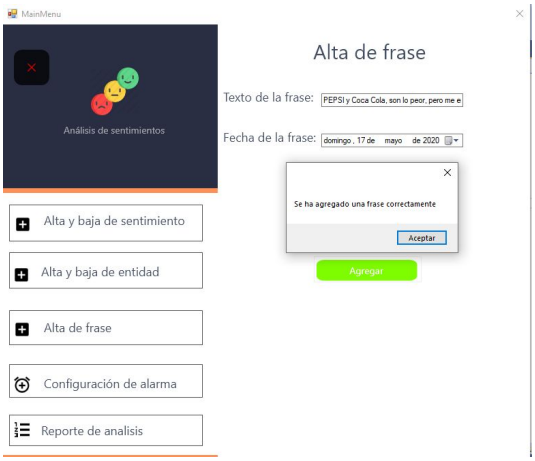
Caso de prueba 1.7:

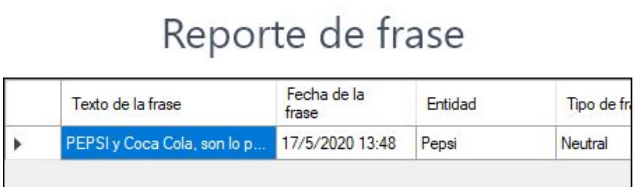




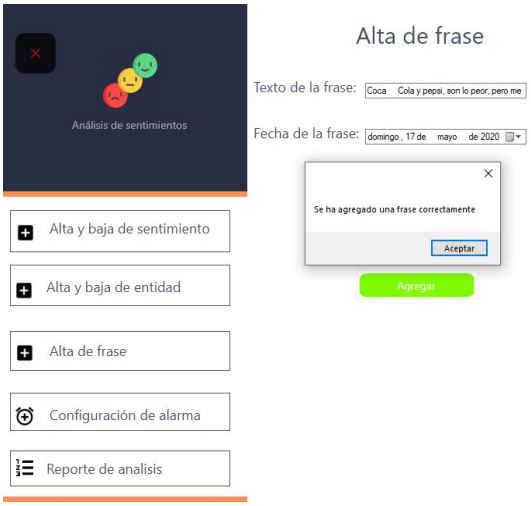
Texto de la frase	Fecha de la frase	Entidad	Tipo de fras
Me gusta Coca Cola	17/05/2020 14:00	Coca Cola	Positive
Pepsi es lo peor	17/05/2020 14:00	Pepsi	Negative
Coca Cola	17/05/2020 14:00	Coca Cola	Neutral
Me gusta la Coca Cola y la Pepsi	17/05/2020 14:02	Coca Cola	Positive
Me gusta y a la vez detesto la Pepsi	17/05/2020 14:03	Pepsi	Neutral
Me gusta y me encanta la Coca C...	17/05/2020 14:03	Coca Cola	Positive
Odio, detesto, es lo peor la Coca ...	17/05/2020 14:04	Coca Cola	Neutral

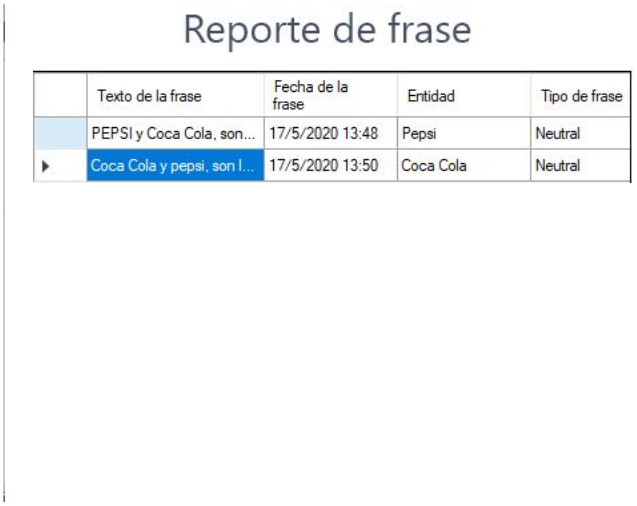
Caso de prueba 1.8:





Caso de prueba 1.9

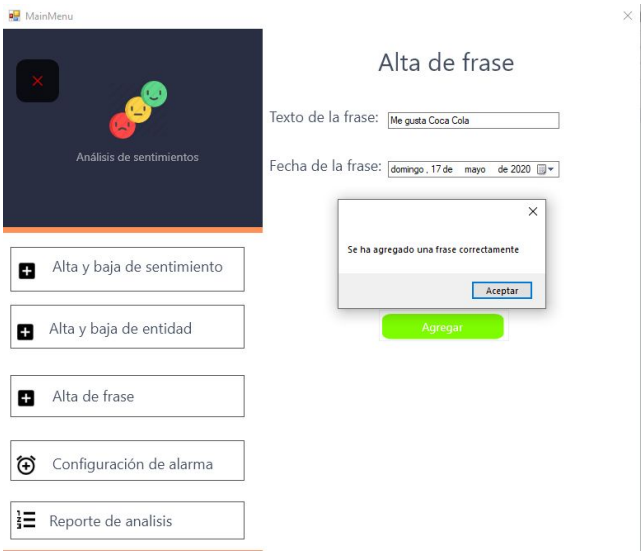




Caso de prueba 2.1

Caso de prueba 2.2

Caso de prueba 3.1



	Texto de la frase	Fecha de la frase	Entidad	Tipo de frase
	PEPSI y Coca Co...	17/5/2020 13:48	Pepsi	Neutral
	Coca Cola y peps...	17/5/2020 13:50	Coca Cola	Neutral
▶	Me gusta Coca C...	17/5/2020 13:55	Coca Cola	Positive

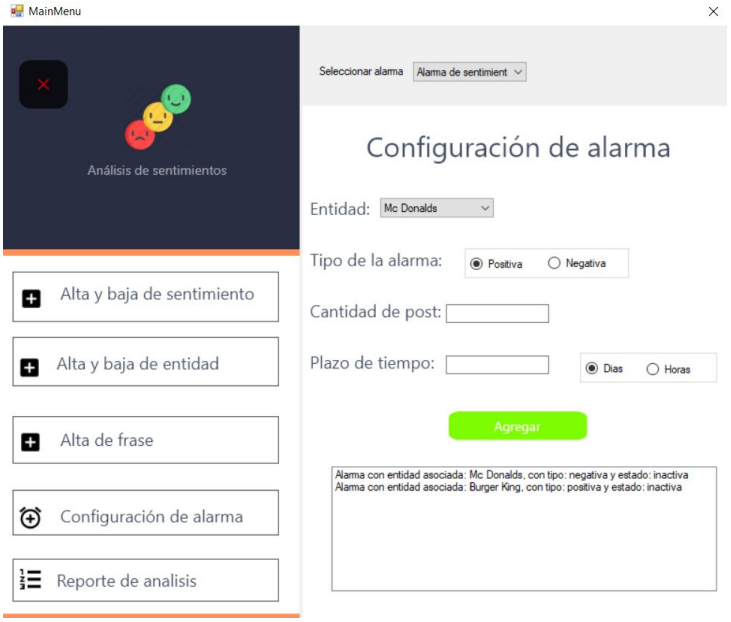
Caso de prueba 3.2 y 3.3
Para estos casos de prueba, se dificulta mostrar evidencia mediante capturas de pantalla, es por esto, que la realización de estas pruebas se pueden visualizar mediante el siguiente enlace:

https://www.youtube.com/watch?v=tnY-v-o-1tk

En estas pruebas se intenta ingresar una fecha fuera del rango establecido, tanto seleccionando del calendario, como intentando modificar la fecha directamente desde el teclado.

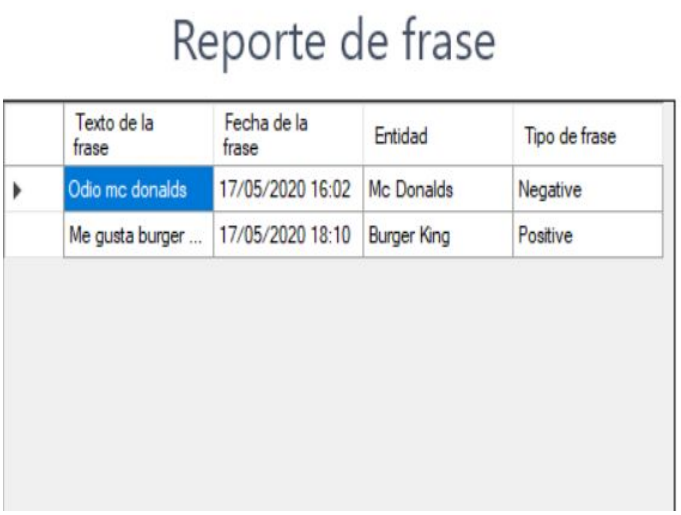
Generación de alertas

Creación de alarmas:

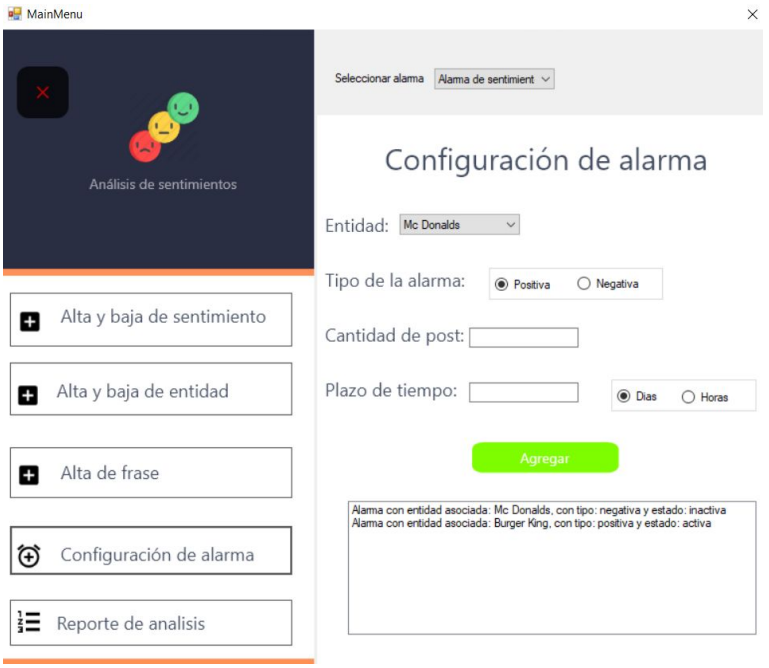


Caso de prueba 1:

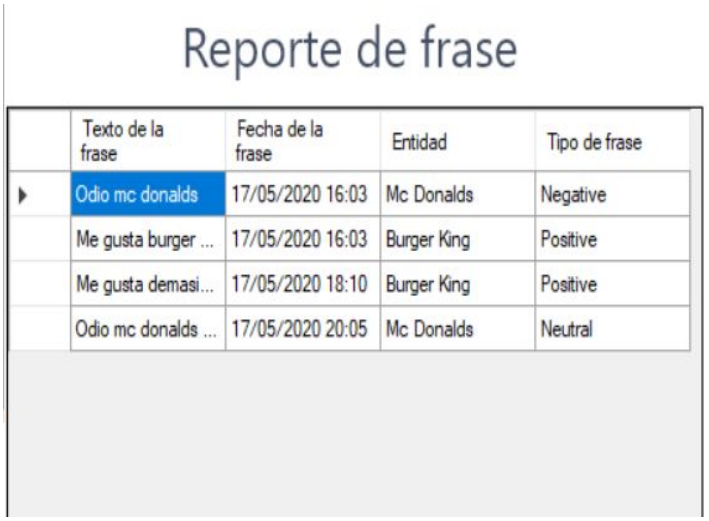
Caso de prueba 2:



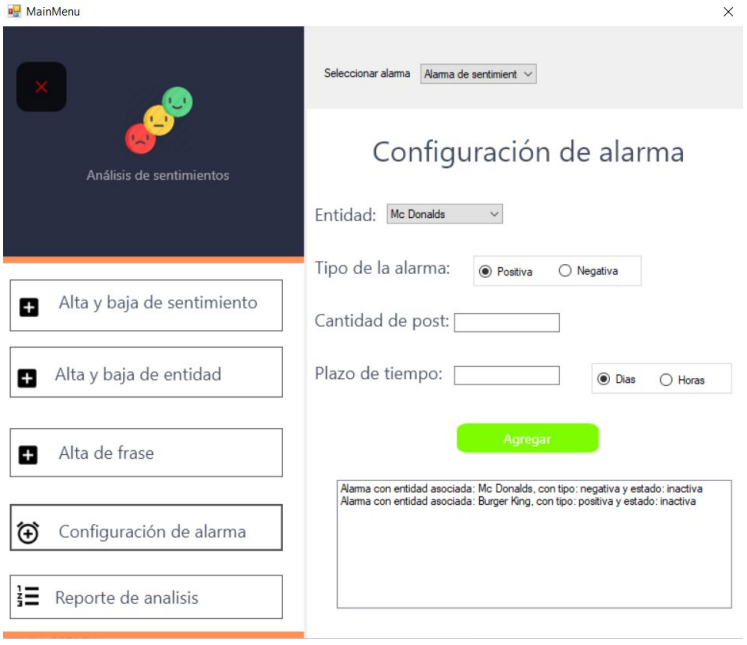
Caso de prueba 3:



Caso de prueba 4:



Caso de prueba 5:



Caso de prueba 6:

Caso de prueba 7:

Caso de prueba 8: