Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 1
Obligatorio 1

Santiago Toscanini (232566)
Nahuel Biladóniga (211138)
Sofia Tejerina (209239)

Entregado como requisito de la materia Diseño de aplicaciones 1

21 de mayo de 2020

Resumen

Para la materia Diseño de Aplicaciones 1, en el primer obligatorio se plantea 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 debe ser capaz de realizar un análisis básico de frases ingresadas, basándose en los datos de sentimientos y entidades que tiene guardados.

El objetivo de este proyecto es aplicar los conocimientos adquiridos en clase, Clean code y TDD.

Índice general

1. Descripción General

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

Las funcionalidades con las que cuenta el sistema son 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.

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 y siguiendo las buenas prácticas propuestas por Clean Code. Al ser nuestra primera vez desarrollando con TDD nos resultó difícil, sobre todo al comienzo y más sobre el final, cuando ya casi todo estaba hecho y debíamos agregar algún detalle o cambio al código que ya teníamos creado. Notamos que las pruebas que hacíamos eran poco explicitas, a medida que íbamos generando la lógica se requería que fuéramos complementándolas con pruebas más detalladas de lo qué debía hacer la función que buscábamos crear.

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.

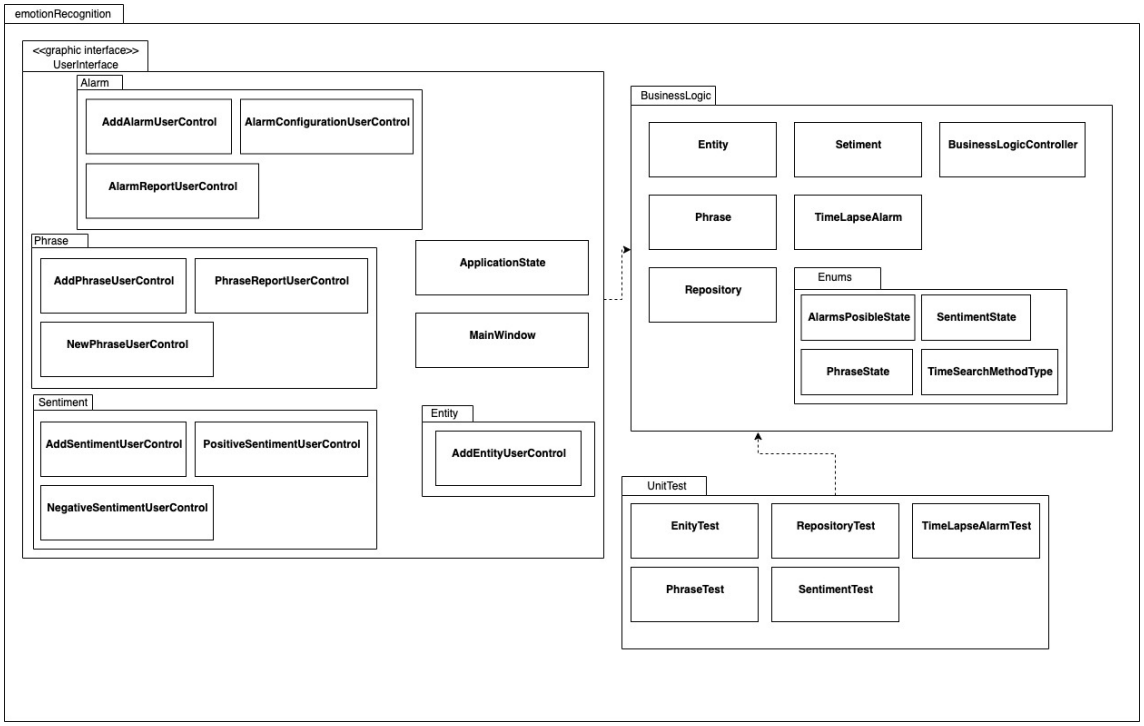


Figura 2.1: Diagrama de paquetes

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 una instancia, recibe una instancia o devuelve una instancia de una ClaseB, esa ClaseA tiene una dependencia con la clase B 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 esta 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.

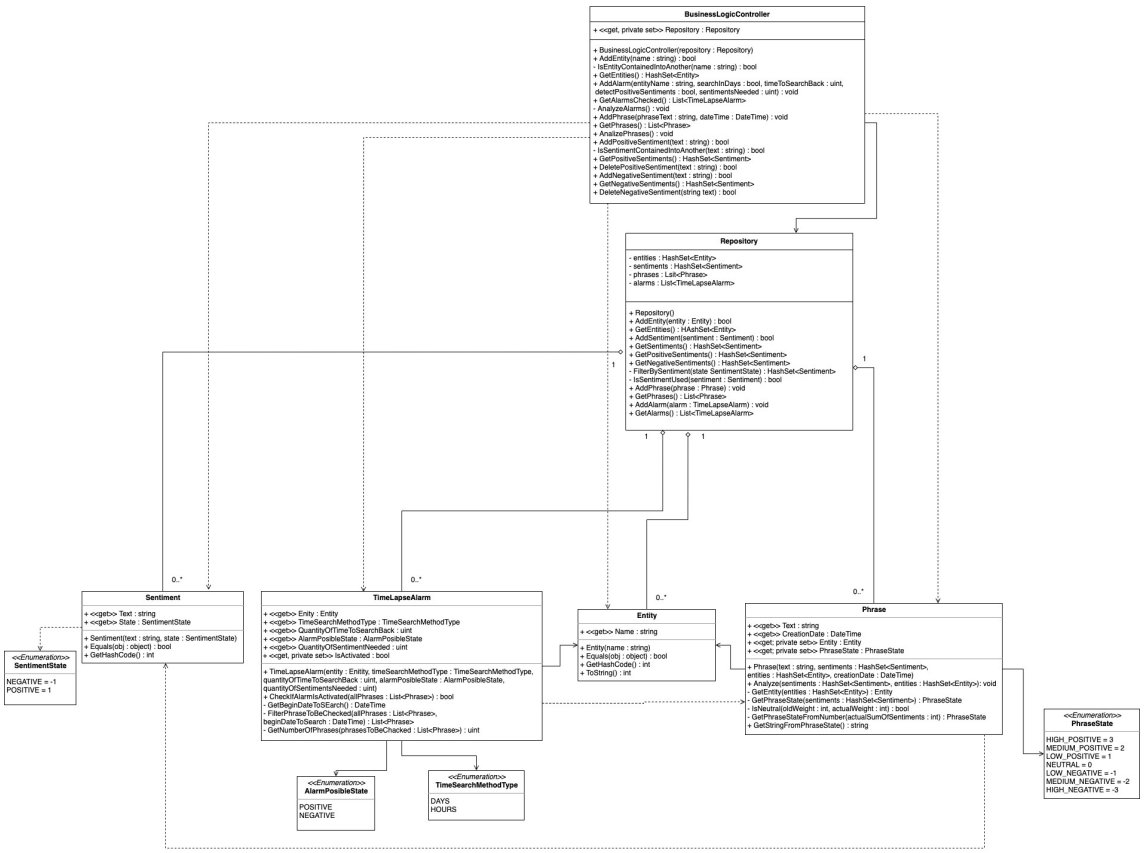


Figura 2.2: Diagrama de clases BusinessLogic

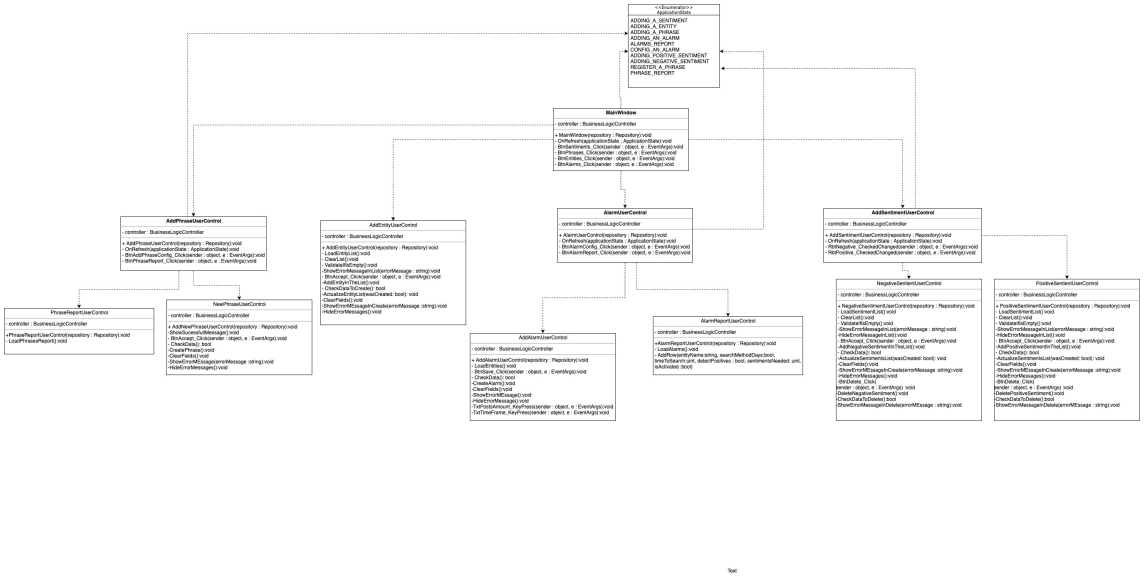


Figura 2.3: Diagrama de clases UserInterface

2.3. 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 ciertos 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, y al ingreso de una alarma.

5. Las frases son analizadas luego del ingreso de una entidad, un sentimiento o una frase.

2.4. Descripción de diseño

2.4.1. Diseño actual

Para la implementación de este sistema creamos tres proyectos; un Class Library .NET Framework (BusinessLogic), un MSTest .NET Framework (UnitTest) y un WindowsForms .NET Framework (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 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 cuatro clases una para cada elemento esencial de nuestro sistema: "Entity", "Phrase", "Sentiment" y "TimeLapseAlarm".

Una clase "Repository" encargada de guardar los datos que se crean en el sistema: entidades, frases, alarmas y sentimientos. Actualmente también tiene métodos de lógica relacionada con sentimientos, como principal prioridad para la segunda versión del sistema se planea cambiar esto, dejando a "Repository" encargado de una única tarea: guardar y devolver los datos de la aplicación. Separar la lógica de esta clase, permite cambiarla fácilmente por una Base de datos sin afectar el funcionamiento del sistema.

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 "BusinessLogicController" tiene una instancia de "Repository", la cuál recibe como parámetro en el constructor. En la ventana principal de la interfaz de usuario creamos la instancia de "Repository" que utilizaremos en toda la aplicación, luego cada clase User Control creará una instancia de "BusinessLogicController" con este repository, para poder acceder a las funciones que necesita.

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, 1 y -1 que se utiliza para determinar que tan positiva, negativa o simplemente neutra es una frase que se ingresó, al analizar los sentimientos que contiene. "AlarmPosibleState" 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 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 simbolo 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 estandar 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	tvw
TabControl	tab
DataTimePicker	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 2.4: Estandar 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 2.5: Estandar para elementos de WindowsForms

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

2.4.2. Mejora del diseño

Luego de un análisis de mejoras para el diseño implementado llegamos a una mejor arquitectura que no se implemento por limitaciones de tiempo, pero se describirá a continuación, siendo posible implementarse para el segundo obligatorio.

Por un lado tendríamos las clases que modelan nuestro problema, estas mismas no contendrían lógica, esta ultima iría en uno o varios servicios que utilizan estas clases con el propósito que necesitamos, tendríamos un controlador que nos permita utilizar el backend de forma unificada y sencilla, abstrayendo totalmente la implementación por detrás que se utilice, el controlador tendría una instancia de un de repository que sería un singleton, y el controlador se comunicaría con el frontend mediante DTOs, para así enviar los datos listos para ser mostrados, no teniendo lógica alguna de enums, 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.

También se cambiarían todas las listas y sets por interfaces del tipo IEnumerable, esto combinado a que el repositorio fuera 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.

A su vez Alarm podría ser una interfaz, para de esta forma poder implementar distintos tipos de alarmas sin necesitar cambios en el resto de clases, únicamente en el servicio.

2.4.3. 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.

"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.

"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.

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

- Crear los objetos de tipo Repository.
- Almacena los datos de las frases, entidades, sentimientos y alarmas del sistema.
- Administrar los datos del repositorio (agregar y obtener los datos guardados, eliminar los sentimientos guardados que no han sido usados).

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

- Crear los objetos de tipo TimeLapseAlarm.
- Administra los datos de la alarma (entidad, tiempo que abarca, en que unidad de tiempo: dias 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.

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

- Crear los objetos de tipo BussinessLogicController.
- 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).

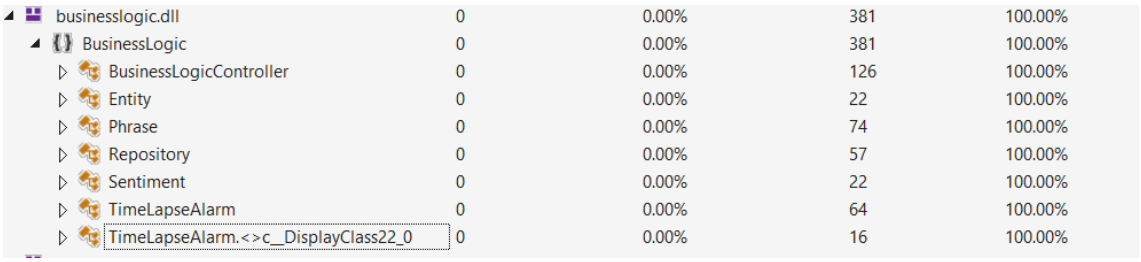
- Permite ingreso y consulta de frases al sistema, realiza también el análisis de las frases luego de ingresado un nuevo sentimiento o entidad.
- Permite agregar nuevos sentimientos, realizando las validaciones correspondientes para permitir el ingreso de dicho dato y devolviendo a la interfaz si se 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.

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. Queda pendiente, como fue mencionado en las mejoras de diseño, implementar este concepto también para el resto de los objetos (entidad, alarma, sentimiento y frase).

3. Pruebas

3.1. Cobertura de pruebas unitarias



businesslogic.dll	0	0.00%	381	100.00%
BusinessLogic	0	0.00%	381	100.00%
BusinessLogicController	0	0.00%	126	100.00%
Entity	0	0.00%	22	100.00%
Phrase	0	0.00%	74	100.00%
Repository	0	0.00%	57	100.00%
Sentiment	0	0.00%	22	100.00%
TimeLapseAlarm	0	0.00%	64	100.00%
TimeLapseAlarm.<>c__DisplayClass22_0	0	0.00%	16	100.00%

Figura 3.1: Cobertura de pruebas unitarias

Durante las pruebas unitarias, como fue mencionado previamente, se utilizó la metodología de TDD. Lo cual, nos permitió alcanzar un porcentaje de prueba superior al que hubiéramos tenido de no utilizar la metodología anterior mencionada; llegando a cubrir el 100 % del la lógica de negocio. La justificación de las mismas procede de que; permite definir las reglas de negocio que deberá tener la aplicación, así como sus posibles variantes y casos limites. De esta forma logramos generar una relación entre cada unit test con su correspondiente lógica asociada.

3.2. Casos de prueba

3.2.1. Análisis de frases

Prueba	Datos de Prueba	Resultado Esperado	Resultado Obtenido	Resultado
Ingreso Frase	Frases: 'Me gusta Netflix'-19/05/2020-19:00:00	Frase Ingresada	Ver: 4.1 4.2	OK
Ingresar Frase vacía	Frases: ''-19/05/2020-19:00:00	Frase no puede ser vacía	Ver: 4.3	OK
Frase con entidad no registrada	Frases: 'Me gusta Netflix' 19/05/2020-19:00:00	Entidad analizada debe ser vacía	Ver: 4.4	OK
Frase con entidades múltiples y registradas	Frases: 'Me gusta Netflix y HBO'-19/05/2020-19:00:00 Entidades: 'Netflix' 'HBO'	Análisis Entidad:'Netflix'	Ver: 4.5	OK
Frase con sentimiento positivo	Frases: 'Me gusta Netflix'-19/05/2020-19:00:00 Entidades: 'Netflix' Sentimientos: 'gusta'	Análisis Entidad:'Netflix' Sentimiento:'Poco Positivo:	Ver: 4.6	OK
Frase con 2 sentimiento positivos	Frase: 'Netflix es impresionante y divertido'-19/05/2020-19:00:00 Entidades: 'Netflix' Sentimientos: 'impresionante' 'divertido'	Análisis Entidad:'Netflix' Sentimiento:'Medianamente Positivo'	Ver: 4.7	OK
Frase con 3 sentimientos positivos	Frase: 'Netflix es impresionante,divertido y emocionante'-19/05/2020-19:00:00 Entidades: 'Netflix' Sentimientos: 'impresionante' 'divertido' 'emocionante'	Análisis Entidad:'Netflix' Sentimiento:'Altamente Positivo'	Ver: 4.8	OK
Frase con sentimiento negativo	Frase: 'Netflix es aburrido'-19/05/2020-19:00:00 Entidades: 'Netflix' Sentimientos: 'aburrido'	Análisis Entidad:'Netflix' Sentimiento:'Poco Negativo'	Ver: 4.9	OK
Frase con 2 sentimientos negativos	Frase: 'Netflix es aburrido y malo'-19/5/2020-19:00:00 Entidades: 'Netflix' Sentimientos: 'aburrido' 'malo'	Análisis Entidad:'Netflix' Sentimiento:'Medianamente Negativo'	Ver: 4.10	OK
Frase con 3 sentimientos negativos	Frase: 'Neflix es aburrido,malo y caro'-19/5/2020-19:00:00 Entidades 'Netflix' Sentimientos: 'aburrido' 'malo' 'caro'	Análisis Entidad:'Netlix' Sentimiento:'Altamente Positivo'	Ver: 4.11	OK
Frase con sentimiento repetido	Frase: 'Netflix es bueno y muy bueno'-19/5/2020-19:00:00 Entidades 'Neflix' Sentimientos: 'bueno'	Análisis Entidad:'Netflix' Sentimiento:'Poco Positivo'	Ver: 4.12	OK
Frase con entidad cortada	Frase: 'Hola, todo bien?'-19/5/2020-19:00:00 Entidades: 'ola'	Análisis Entidad:'ola'	Ver: 4.13	OK

3.2.2. Generación de alerta

Prueba	Datos de Prueba	Resultado Esperado	Resultado Obtenido	Resultado
Alarma activada con frase	Sentimientos: 'Me gusta' 'Amo' 'Lo mejor' Entidades: 'Burger King' Frases: 'Me gusta Burger King' -16/05/2020 'Amo Burger King' - 18/05/2020 'Burger King es lo mejor' - 19/05/2020 Alarma: Si en los últimos 10 días hay 3 comentarios positivos sobre Burger King se activa	Se activa la alarma	Ver 4.14	OK
Alarma no se activa por falta de frases	Sentimientos: 'ok' 'divertido' Entidades: 'Valorant' Frases: 'Valorant es divertido'-21/05/2020-10:00:00 'Valorant es ok'-20/05/2020-10:00:00 Alarma: Si en los últimos 2 días hay 3 comentarios positivos sobre Valorant se activa	No se activa la alarma	Ver 4.15	OK
Alarma no se activa por falta de sentimientos	Sentimientos: 'muy bueno' 'emocionante' 'aburrido' Entidades: 'Netflix' Frases: 'Netflix es muy bueno'-20/05/2020-10:00:00 'Netflix es aburrido'-20/05/2020-10:00:00 'Netflix es emocionante'-20/05/2020-10:00:00 Alarma: Si en los últimos 2 días hay 3 comentarios positivos sobre Netflix se activa	No se activa la alarma	Ver 4.16	OK
Alarma no se activa porque no existe entidad	Sentimientos: 'divertido' Frases: 'Netflix es divertido'-20/05/2020-10:00:00 Alarmas: Si en los últimos 2 días hay 1 comentario positivo sobre Netflix se activa	No se activa la alarma	Ver 4.17	OK
Alarma no se activa porque la frase es neutra	Sentimientos: 'divertido' 'caro' Entidad: 'Netflix' Frase: 'Netflix es divertido pero caro'-20/05/2020-10:00:00 Alarma: Si en los últimos 2 días hay 1 comentario positivo sobre Netflix se activa	No se activa la alarma	Ver 4.18	OK
Alarmas múltiples	Sentimientos: 'divertido' 'emocionante' Entidad: 'Netflix' Frase: 'Netflix es divertido'-20/05/2020-10:00:00 'Netflix es emocionante'-20/05/2020-10:00:00 Alarma: Si en los últimos 2 días hay 1 comentario positivo sobre Netflix se activa Si en los últimos 2 días hay 2 comentario positivo sobre Netflix se activa	Se activan las dos alarmas	Ver 4.19	OK
Alarma con valores fuera de rango	Entidad: 'Netflix'Alarma: Si en los últimos 999999999 días hay 999999999 comentarios sobre Netflix	No se permite crear la alarma	Ver 4.20	OK

4. Anexo

4.1. Análisis de casos de Prueba

4.1.1. Frases

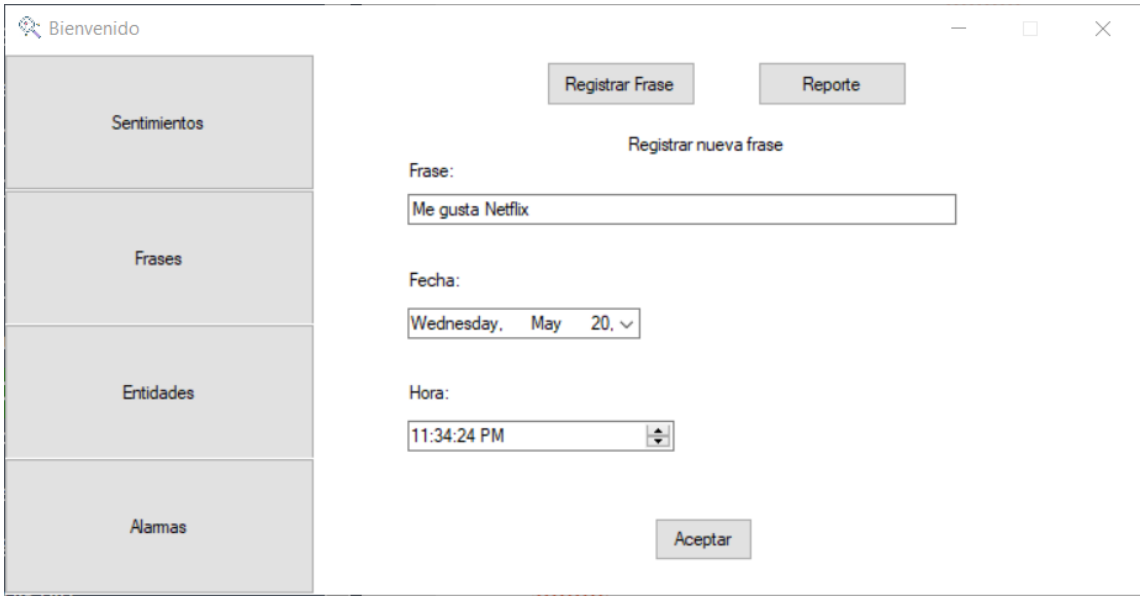


Figura 4.1: Ingreso frase 1

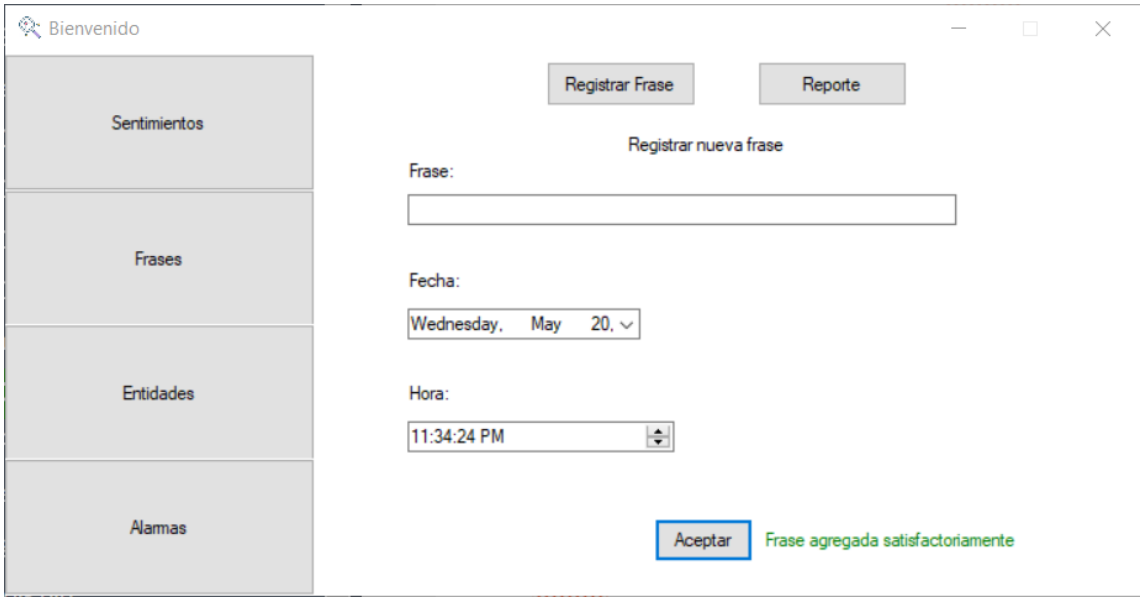


Figura 4.2: Ingreso frase 2

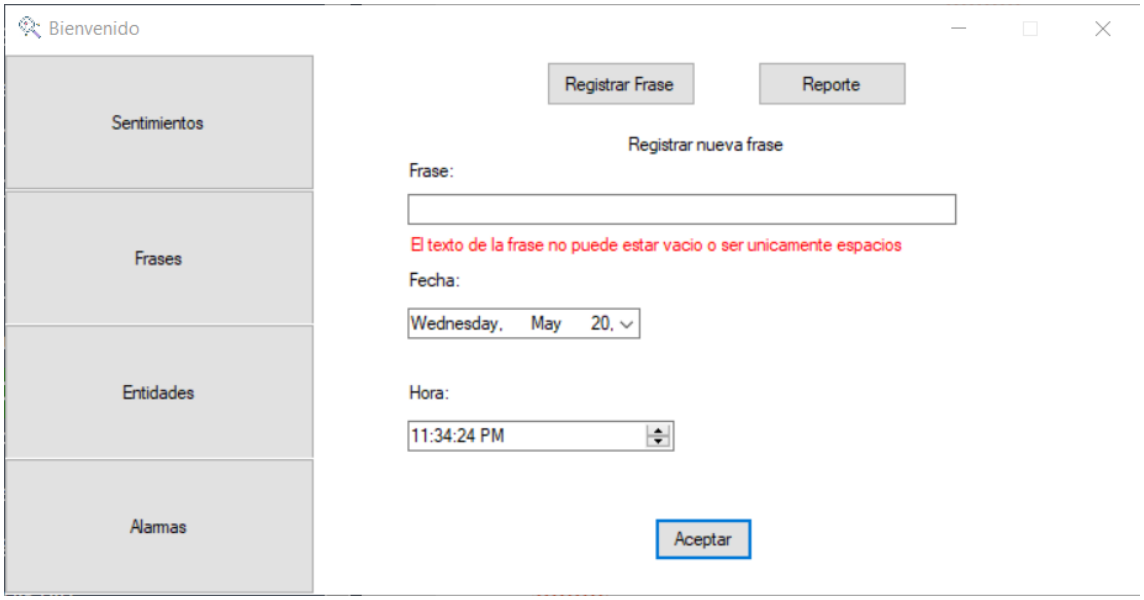


Figura 4.3: Ingreso Frase vacía

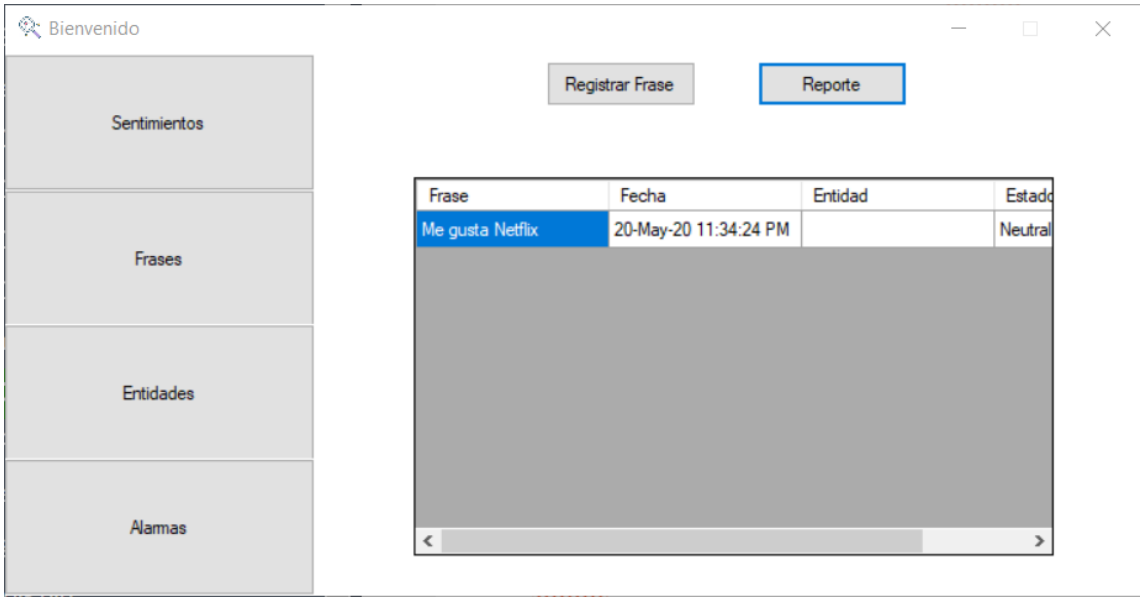


Figura 4.4: Frase con entidad no registrada

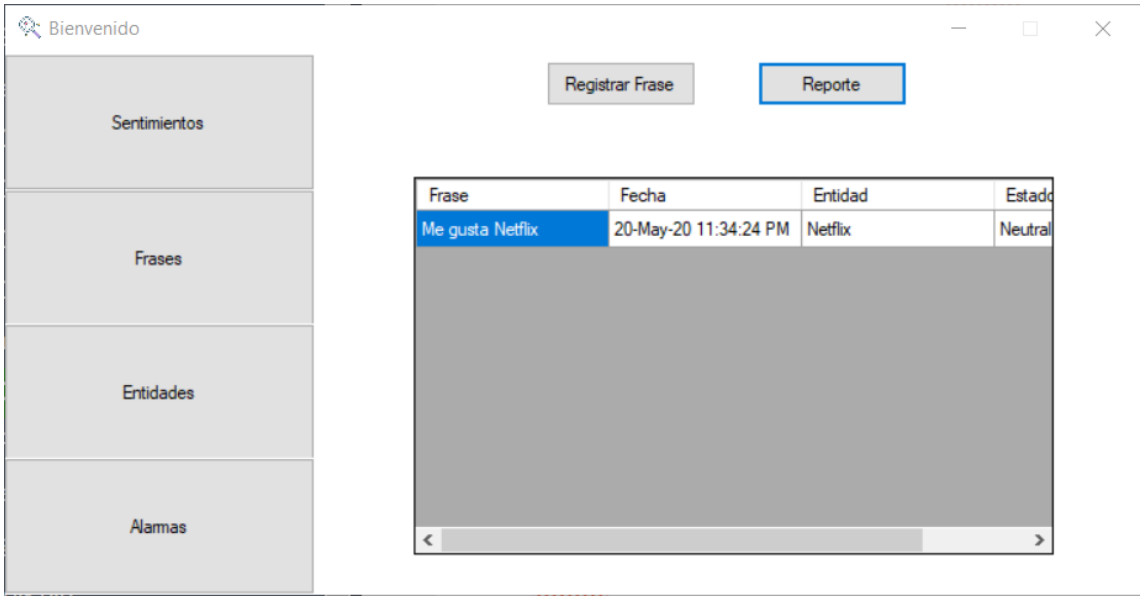


Figura 4.5: Frase con entidades múltiples y registradas

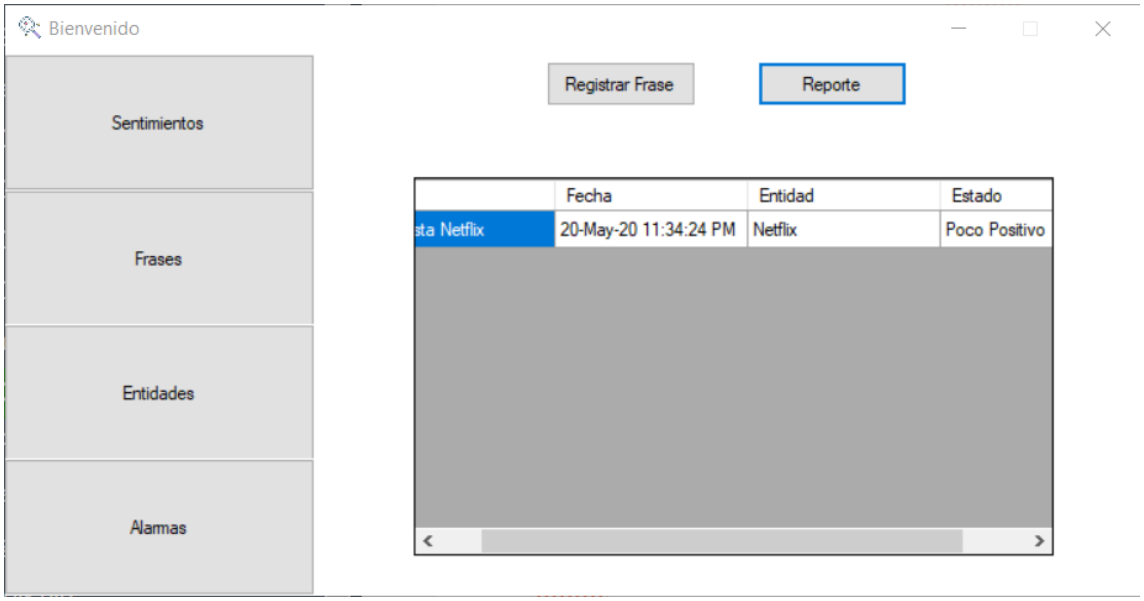


Figura 4.6: Frase con sentimiento positivo

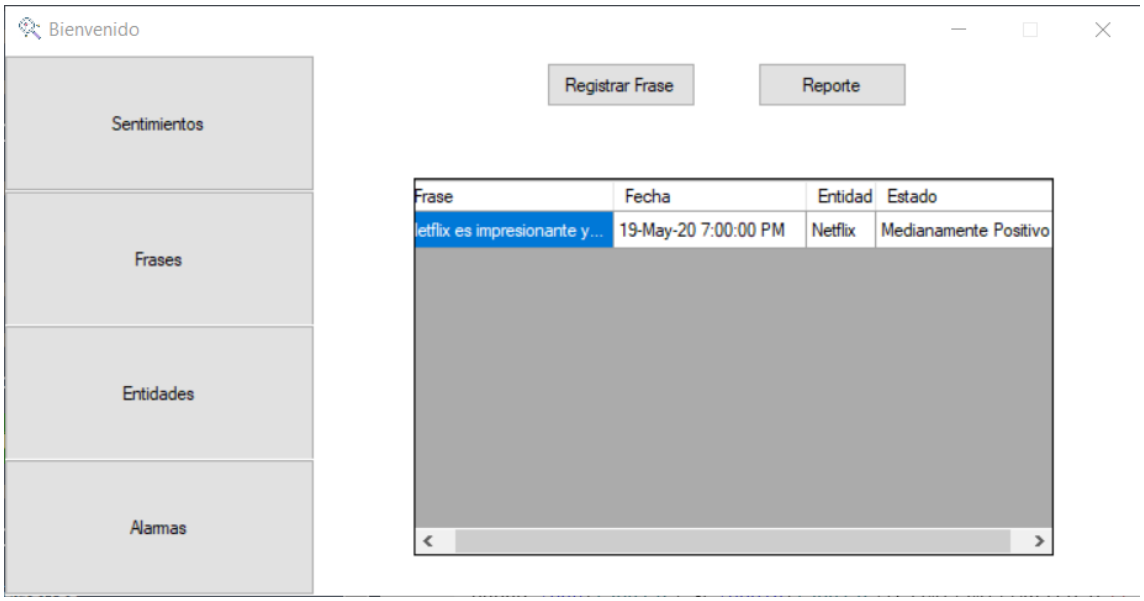


Figura 4.7: Frase con 2 sentimientos positivos

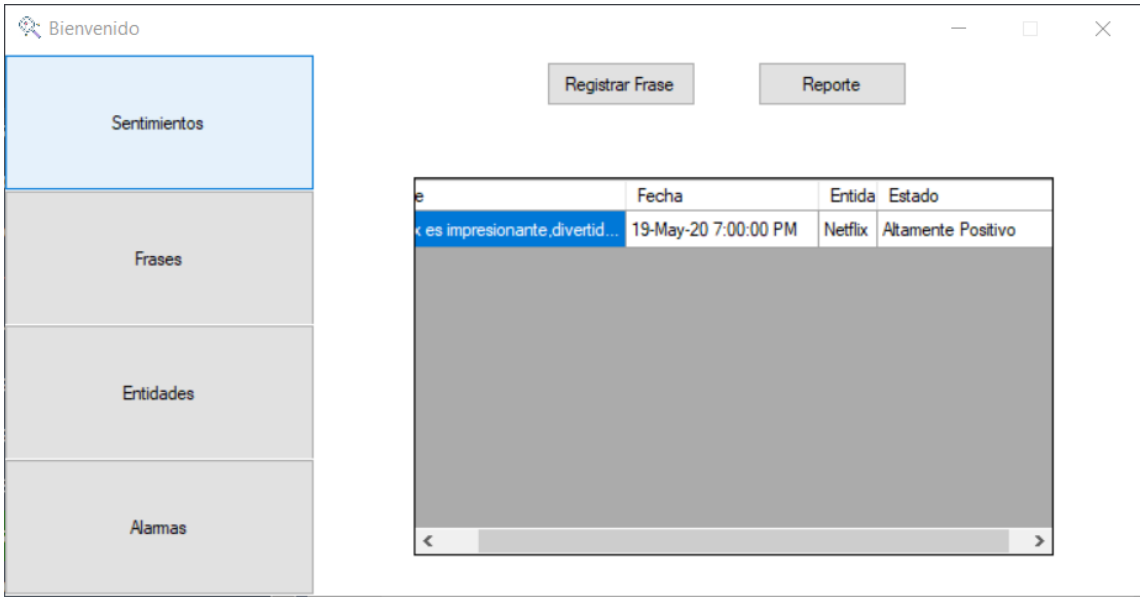


Figura 4.8: Frase con 3 sentimientos positivos

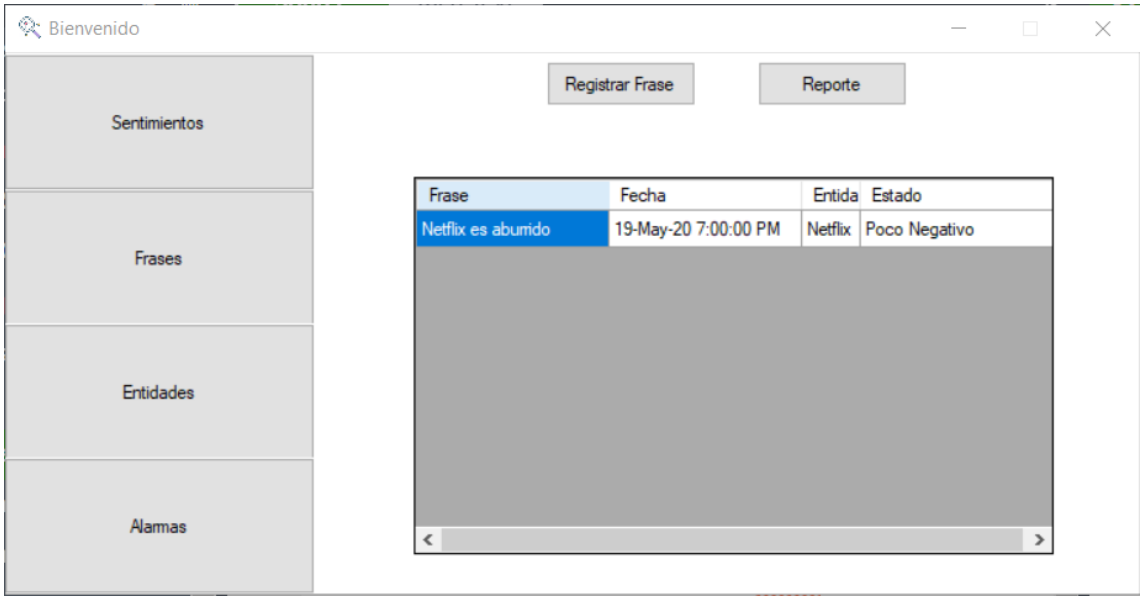


Figura 4.9: Frase con sentimiento negativo

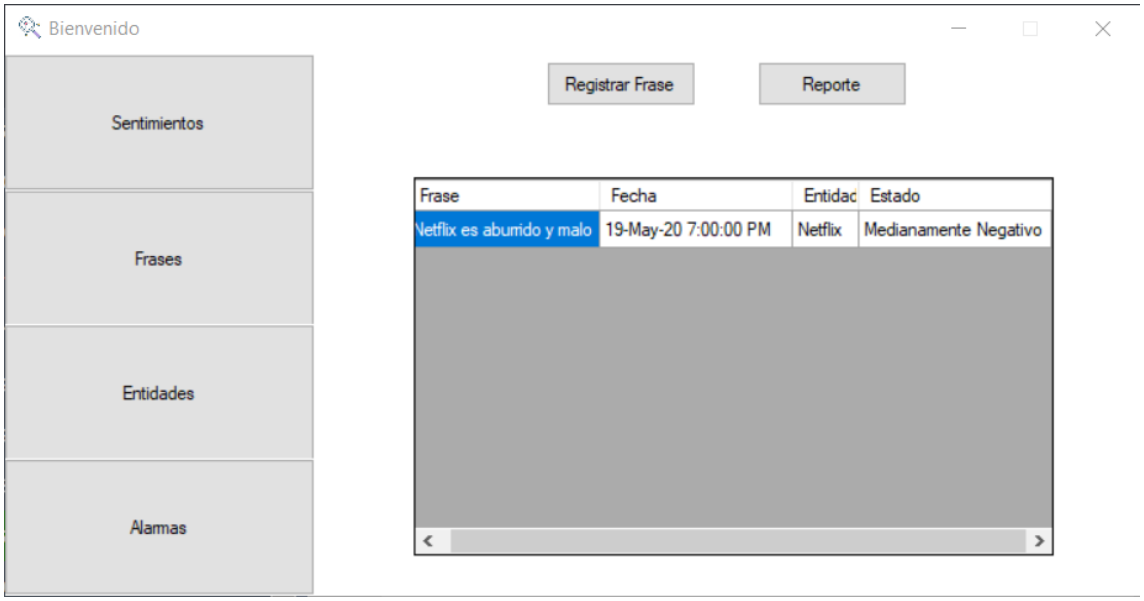


Figura 4.10: Frase con 2 sentimientos negativos

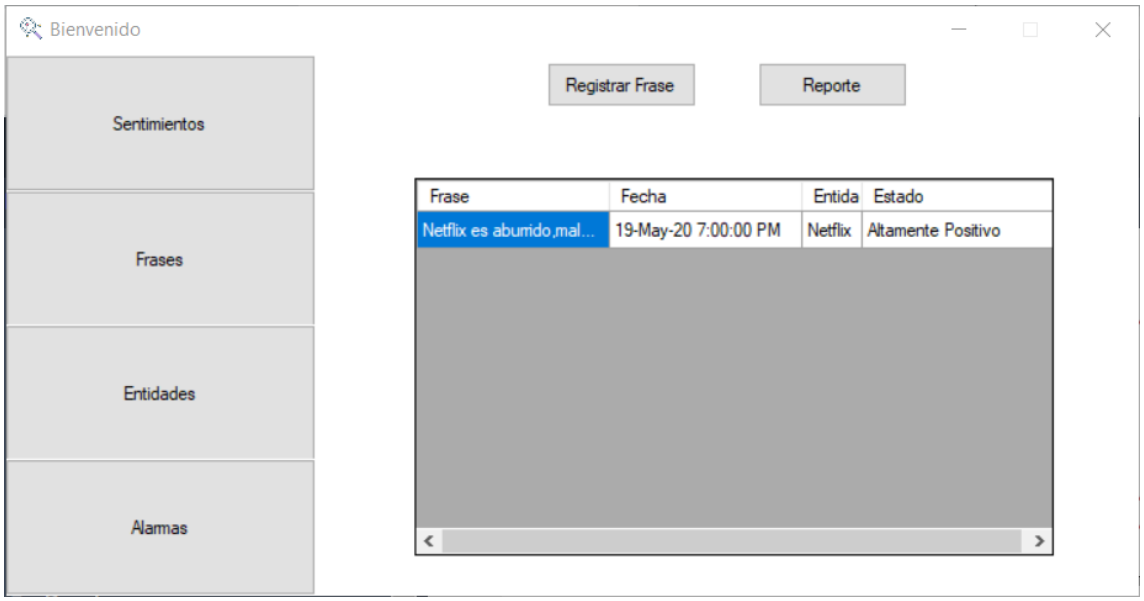


Figura 4.11: Frase con 3 sentimientos negativos

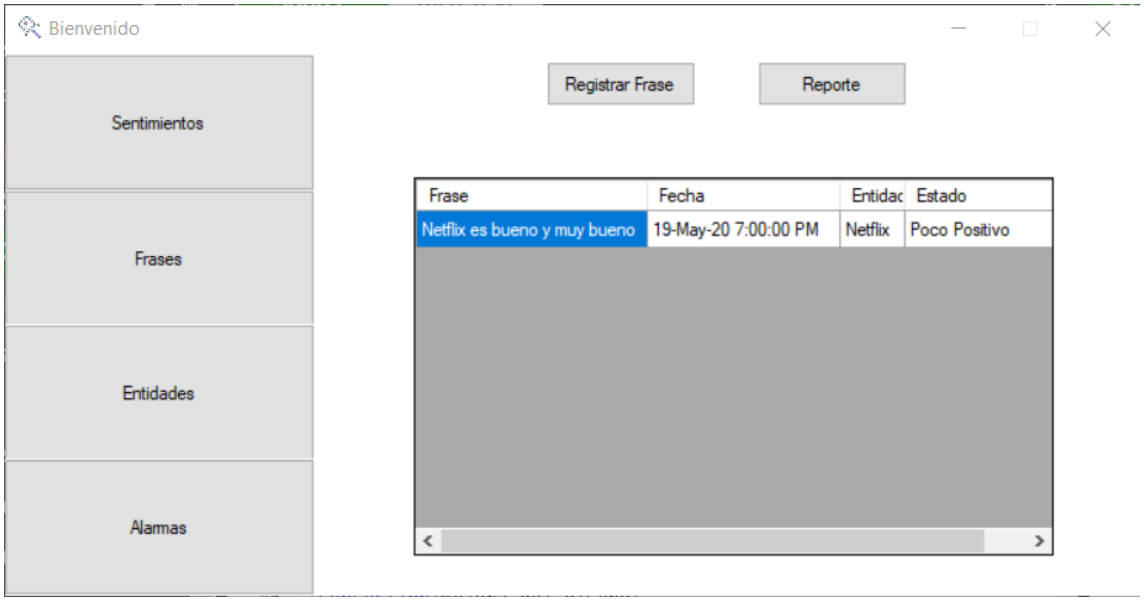


Figura 4.12: Frase con sentimiento repetido

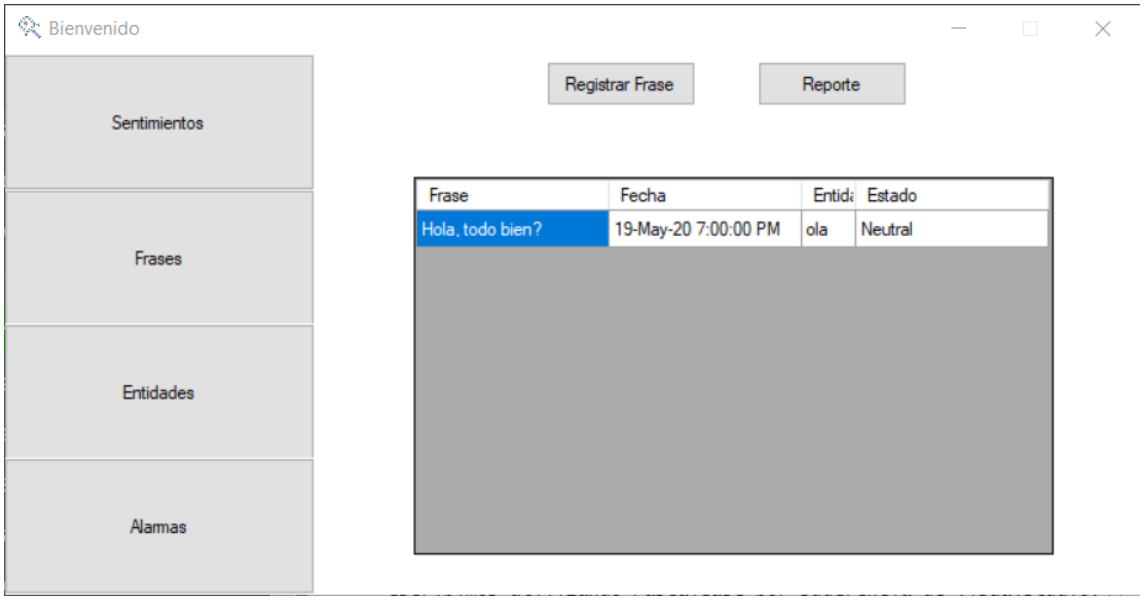


Figura 4.13: Frase con entidad cortada

4.1.2. Alarmas

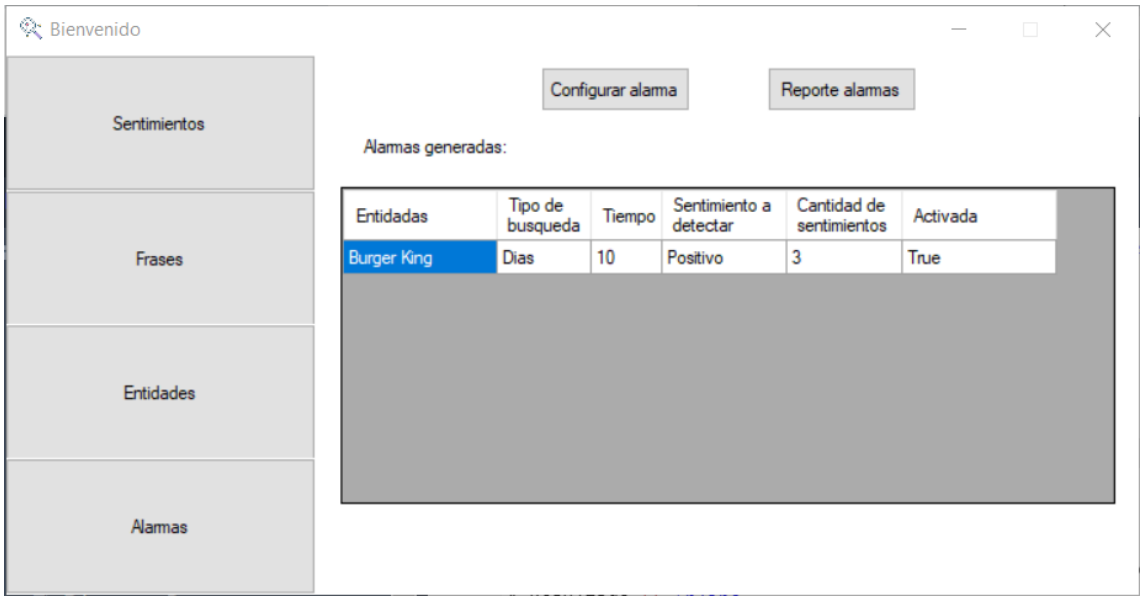


Figura 4.14: Alarma activada con frase

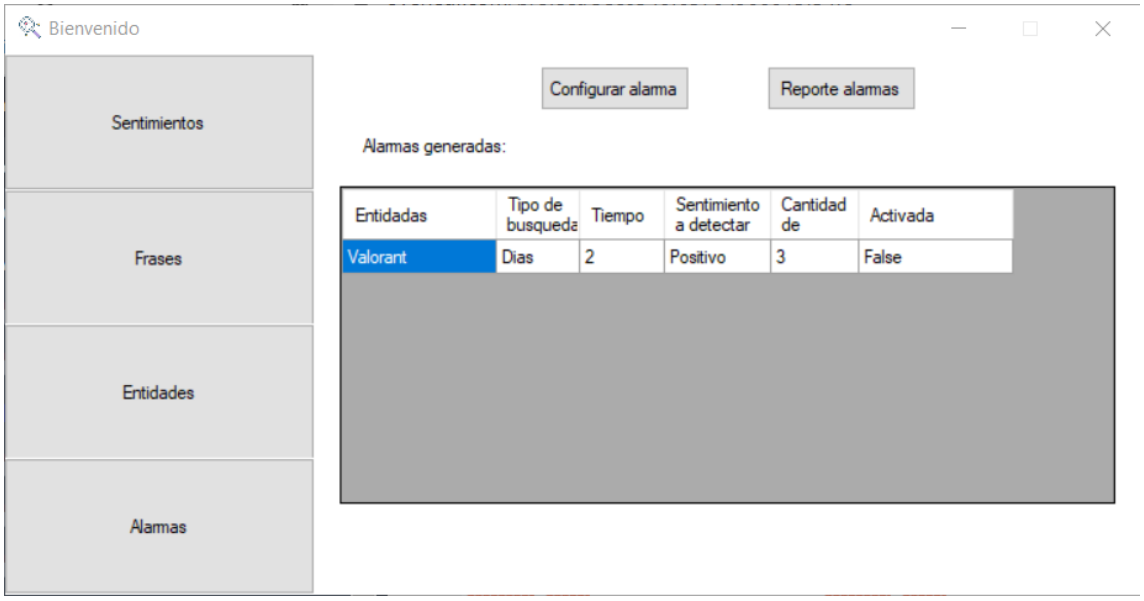


Figura 4.15: Alarma no se activa por falta de frases

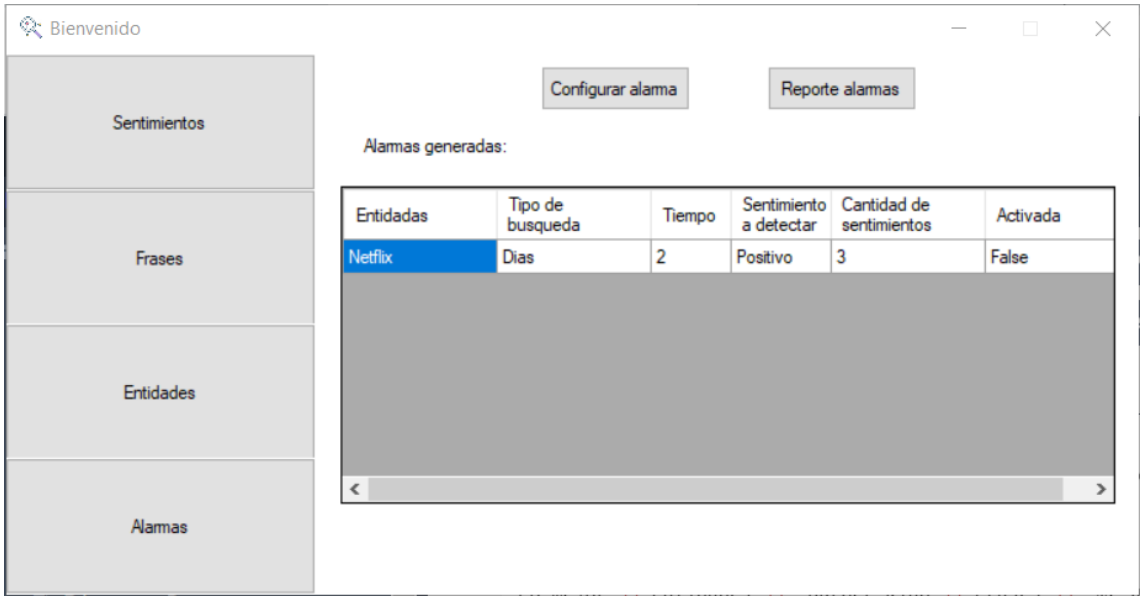


Figura 4.16: Alarma no se activa por falta de sentimientos

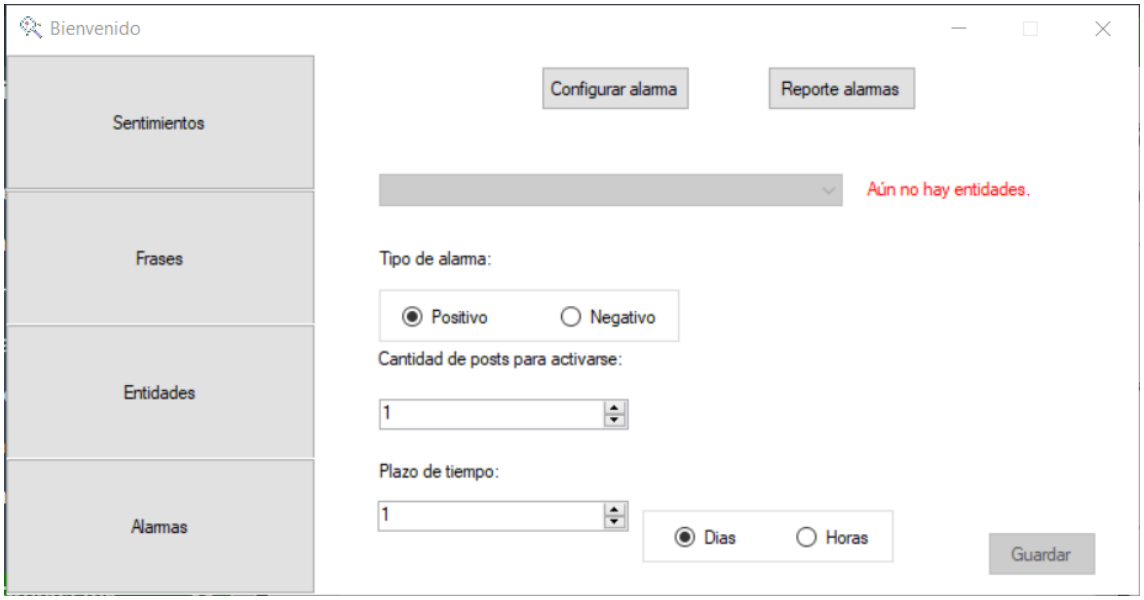


Figura 4.17: Alarma no se activa porque no existe entidad

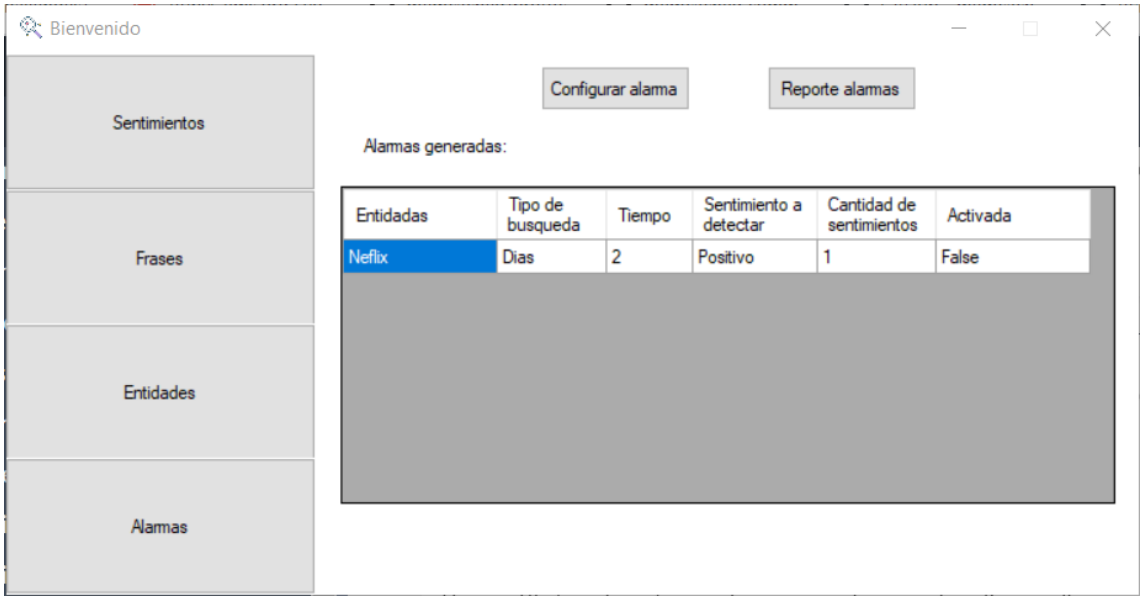


Figura 4.18: Alarma no se activa porque la frase es neutra

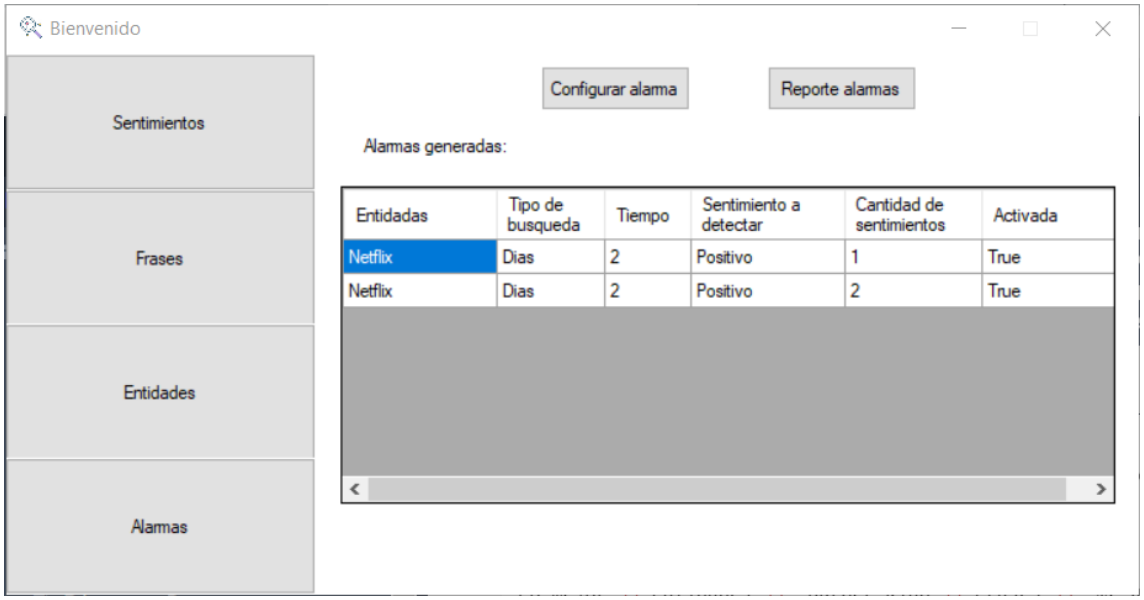


Figura 4.19: Alarmas múltiples

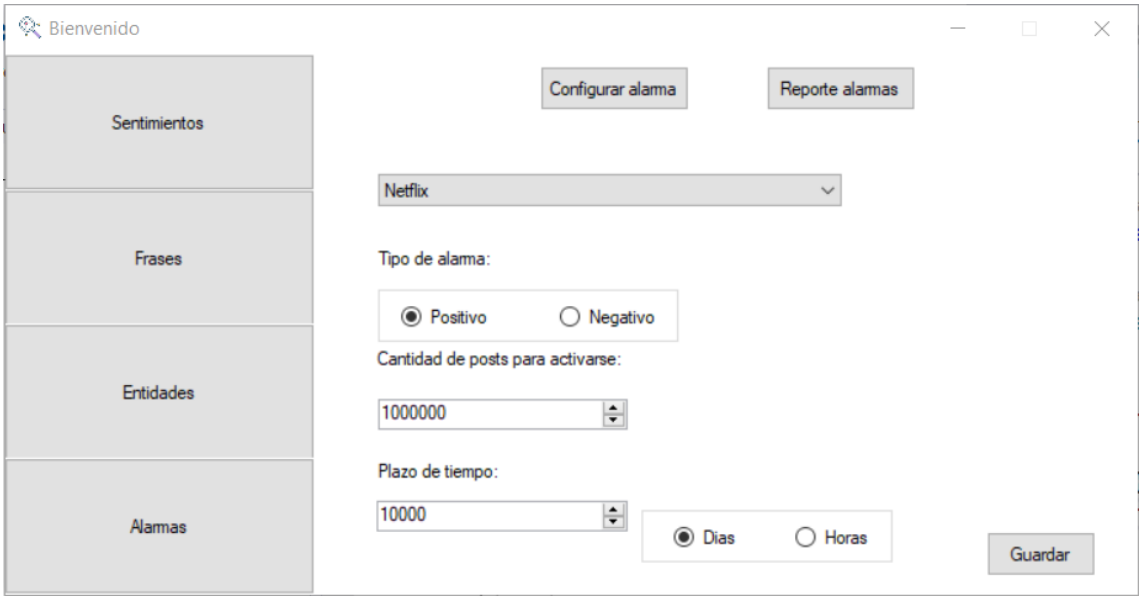


Figura 4.20: Alarma con valores fuera de rango

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