Obligatorio 2 - 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) la cual la persistencia es en una base de datos, 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). Además de haber previamente seleccionado un autor del listado de autores registrados, como él autor originario de la frase.
   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 configurar una alarma. En particular el usuario antes de comenzar a configurar una alarma deberá seleccionar el formato de la alarma, es decir si se trata de una alarma de entidad o una alarma de autor. Comenzando con la alarma de entidad, siendo así que al usuario se le permite crear una 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.

   Por otro lado se tiene las alarmas asociadas al autor, siendo así qué se le permite crear una alarma en la cual se debe especificar únicamente 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, 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 ingresa una alarma con tipo positivo, la cantidad de post igual a uno, y con un plazo de diez días, si no hay usuarios registrados en el sistema no va a ser posible activar nunca a esta alarma debido a que para ingresar frases debe haber al menos un usuario registrado. Suponiendo la existencia de un autor si se ingresa el comentario “Me gusta la Coca Cola” con fecha 20/06/20, 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. Mostrándose en él reporte, sí la alarma se encuentra activa o no. Siendo que sí se encuentra activa, se muestran además los autores que cumplan con la condición para la activación de la alarma, ya que cada uno de ellos activaron la alarma. Siendo de ejemplo sí hay dos autores registrados dentro del sistema, y se configuró previamente la alarma anteriormente mencionado. Una vez que el primer autor ingresa una frase (de tipo positiva dentro de los diez días que se encuentra en el plazo de la alarma), la alarma se activa y únicamente aparece el primer autor (ya que el segundo autor no tiene frases ingresadas en el sistema). Luego si él segundo autor ingresa una frase positiva dentro del plazo establecido en la alarma, la alarma estará activa y contendrá a ambos autores dentro del reporte.

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.

7. Mantenimiento de autores: Se permite al usuario del sistema registrar diferentes autores dentro del sistema. Brindándole la posibilidad de dar de alta, baja y modificación de los autores que han sido creados al sistema. Siendo así que una vez que el usuario se encuentra dentro de la interfaz de mantenimiento de autores, tiene las opciones anteriormente mencionadas.
    a. Alta: Se le permite al usuario registrar un nuevo autor, en el cual se debe ingresar el nombre de usuario del autor (con un máximo permitido de 10 caracteres), además del nombre y apellido del mismo (con un máximo permitido de 15 caracteres para cada uno respectivamente), además de la fecha de nacimiento en la cual la edad del autor debe estar entre 13 a 100 años.

    b. Baja: Se muestra el listado de autores registrados en el sistema, permitiéndole al usuario seleccionar cualquiera de los autores registrados, para luego eliminarlo. Siendo que una vez eliminado el autor, también se eliminan las respectivas frases que dicho autor haya comentado dentro del sistema, quedando únicamente en él sistema aquellas frases con autores registrados dentro del sistema.

    c. Modificación: Al igual que en la baja, se muestra el listado de autores registrados en el sistema, permitiéndole al usuario seleccionar un autor para modificar los datos del mismo. Siendo así que una vez seleccionado el autor, y habiendo clickeado para modificar, los datos de dicho usuario se cargan para poder ser modificados correctamente. Una vez confirmado los cambios de modificación, los datos se guardan correctamente, quedando dentro del sistema con los datos modificados.

8. Reporte de autores: Se permite al usuario, generar un reporte mostrándose como un listado de todos los autores ingresados en el sistema, ordenados en forma ascendente o descendente por alguno de los siguientes criterios que el usuario podrá seleccionar:

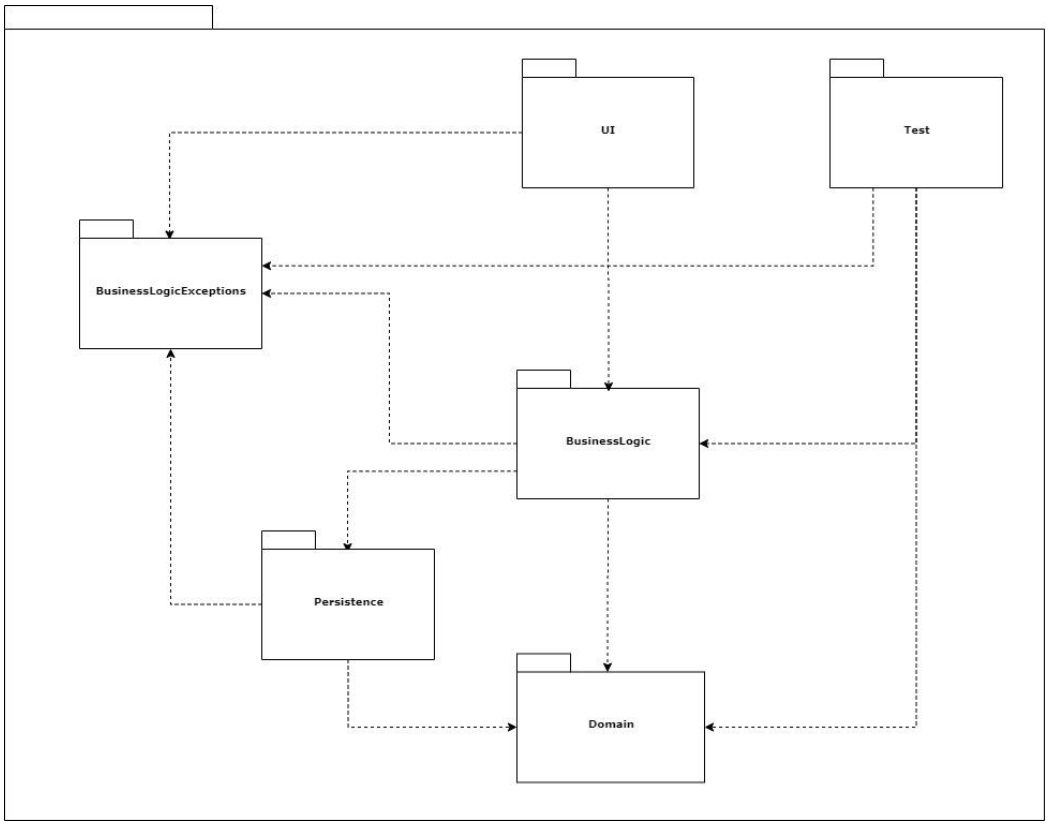
    a. Porcentaje de frases positivas o negativas sobre el total de frases que generó cada uno de los autores. Siendo que el porcentaje de dichas frases, se obtiene sumando todas las frases positivas o negativas generadas por el autor, dividido entre la cantidad de frases totales del autor.

    b. Cantidad de entidades mencionadas en las frases de cada autor. Siendo este el número de entidades distintas que el autor ha nombrado en todas sus frases.

    c. Promedio diario de frases de cada autor. Obteniéndose a partir de dividir el total de frases del autor, entre la cantidad de días que pasaron desde la primera frase del autor que aparece en el sistema (la que tiene la fecha más antigua, independientemente de la fecha de creación en el sistema).

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 la primera versión de la aplicación, la información se almacenaba en memoria, a diferencia de esta nueva versión, en donde se requiere que los datos del sistema sean persistidos en una base de datos, y de esta forma, la siguiente vez que se ejecute la aplicación se comenzará con dichos datos cargados con el último estado guardado antes de cerrar la aplicación.

Para satisfacer este nuevo requerimiento, consideramos necesario la creación de un nuevo paquete llamado *Persistence,* el cual está compuesto por diferentes clases que se encargan de almacenar la información.

Con la creación de este paquete, se generan nuevas dependencias, que hasta la versión anterior de la aplicación no estaban presentes.
Las reglas de negocio, que solo dependían del paquete Dominio y el de las excepciones de las reglas de negocio, ahora también dependen del paquete *Persistence*, porque es necesario que cada clase de las reglas de negocio contengan un objeto de la persistencia, que permita guardar y obtener objetos en el almacenamiento persistente.

Profundizando, este paquete, hace referencia al guardado de los datos del sistema. Es decir cómo se guardan y se obtienen los datos ya ingresados. Básicamente, una capa de persistencia encapsula el comportamiento necesario para mantener los objetos, es decir: leer, escribir y borrar objetos en el almacenamiento persistente.

El hecho de que las reglas de negocio dependen del paquete de persistencia, viola el **principio de inversión de dependencia (DIP)**, en donde se indica que las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones (de interfaces bien definidas).

En nuestra aplicación, decidimos no aplicar este principio, para no agregar dificultad en el diseño, sabiendo de que no van a haber versiones siguientes, en donde cambie la forma en que se almacena la información.

Sin embargo, una posible solución, que llevaría a aplicar este principio, y sería muy oportuno implementarlo en caso de que exista la posibilidad de que cambie la forma de almacenar en futuras versiones, es agregar una abstracción (*IRepository*) para representar los servicios que deben brindar los repositorios, haciendo que la lógica de negocios dependa de la abstracción y no de una implementación concreta. Al realizar esto, lo que se logra es invertir la dependencia, ya que ahora sería *Persistence* quien depende de *BusinessLogic* (y no al revés).

Si se hubiese implementado de esa forma, y el dia de mañana, cambia la forma en que se almacena la información, simplemente habría que agregar un nuevo objeto que implemente la interfaz IRepository (y por tanto las operaciones), pero no sería necesario realizar modificaciones a nivel de las clases de las reglas de negocio, ya que estas únicamente dependen de la abstracción, y no de implementaciones concretas.

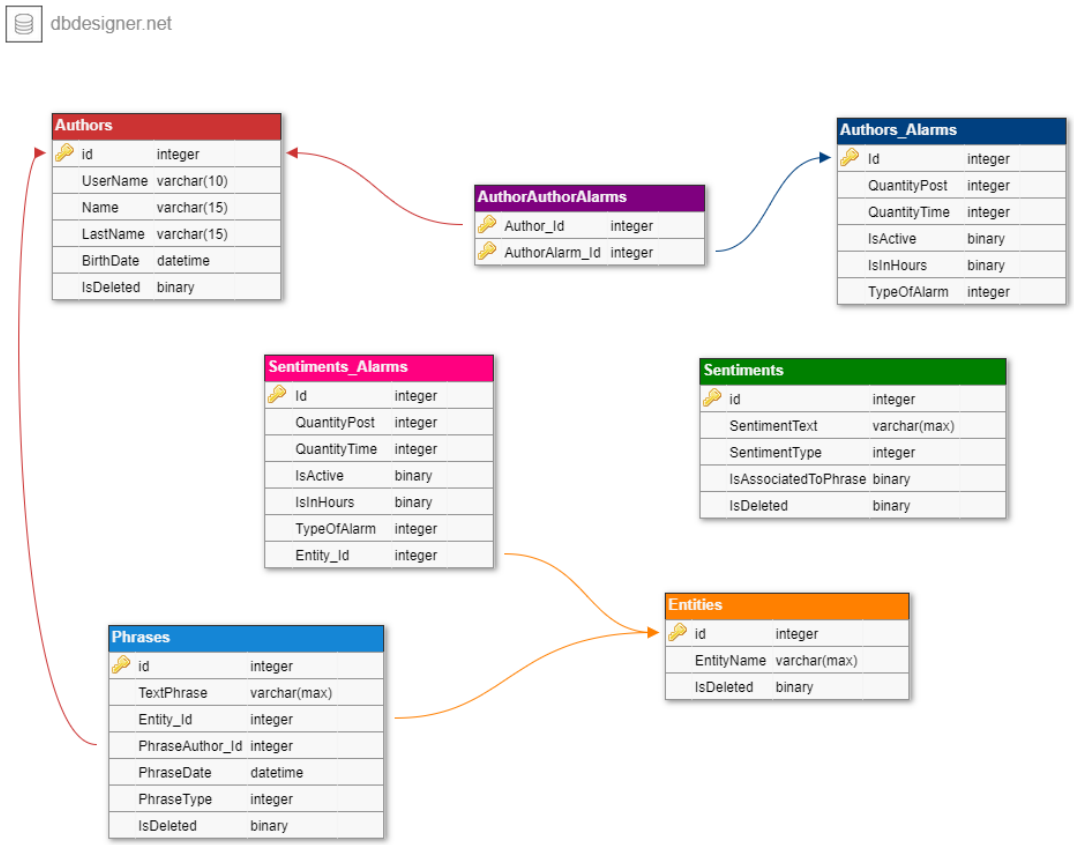
Dentro del paquete Persistence observamos que el mismo depende de las excepciones reglas de negocio, esto debido a que cuando uno quiere a acceder a datos que se encuentran alojados en algún almacenamiento persistente, se puede producir fallas a la hora de obtenerlos. De forma que debemos pensar que exista la posibilidad de fallas en el almacenamiento. 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 Persistence 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.

Considerando que el diseño debe contemplar el modelado de una solución de persistencia adecuada para el problema utilizando Entity Framework (Code First), a continuación, se detallará la solución implementada.

Solución de persistencia

En la siguiente figura, se observa un modelo de tablas de la solución de persistencia, utilizada al desarrollar la aplicación. Este modelo, fue desarrollado en una aplicación externa, pero se verificó que el mismo coincida con el autogenerado por Microsoft SQL Server Management Studio.

Modelo de tablas



La solución consta de 7 tablas, en donde cada una de ellas representa a una entidad del dominio, excepto la llamada *AuthorAuthorAlarms* que representa a la relación existente entre los autores y la nueva alarma lanzada en esta versión (alarma centrada en los autores). Como la relación entre estas entidades es N a N, se crea la tabla antes mencionada, que nos permite obtener dado un autor todas las alarmas en las que participa, y de forma similar, podemos obtener dada una alarma, todos sus participantes.

Las restantes tablas representan a las entidades del dominio, es por esto que tenemos una tabla *Authors,* que contiene todos los datos de los autores, y como clave primaria un ID autoincremental.

Luego, la tabla *Phrases* contiene la información de las frases, y posee una clave foránea a la tabla *Authors* y otra a la tabla *Entities* (este campo es nulleable, porque es posible que en la frase no se mencione a una entidad), permitiendo de esta forma, modelar que cada frase tiene un autor y posiblemente una entidad asociada.

Las tablas *Entities* y *Sentiments* contiene información acerca de las entidades y sentimientos respectivamente, y no poseen claves foráneas hacia otras entidades.

Por último, falta mencionar las dos restantes tablas que hacen referencia a las alarmas, una de ellas para las alarmas basadas en entidades, y la restante para las basadas en autores.

Si bien manejamos la posibilidad de utilizar una única tabla para representar los dos tipos de alarmas, notamos que implicaba un aumento en la complejidad, incluso habría que cambiar la interface *IAlarm* por una clase abstracta, considerando esto nos inclinamos por tener dos tablas independientes, una para cada tipo de alarma, que nos brinda algunas ventajas como: traer fácilmente las alarmas del tipo que se desee, sin tener que implementar una lógica muy complicada.

Otro aspecto a destacar, es que, tal como se observa en el diagrama, es que las entidades: *Authors*, *Sentiments*, *Entities* y *Phrases*, tiene un atributo llamado *IsDeleted* que representa si el objeto está borrado o no. Este atributo está presente porque decidimos hacer un borrado lógico en lugar de físico. La eliminación lógica es aquella que ocurre al activar una marca de "eliminado". Decidimos hacer este tipo de borrado, principalmente por la razón de que al eliminar un registro no debemos preocuparnos por la eliminación en cascada a través de otras tablas en la base de datos que hacen referencia a la fila que se está eliminado (por la existencia de una clave foránea hacia el registro que se elimina), además en caso de que se elimine erróneamente un registro, se puede recuperar fácilmente, colocando la marca *IsDeleted* en false.

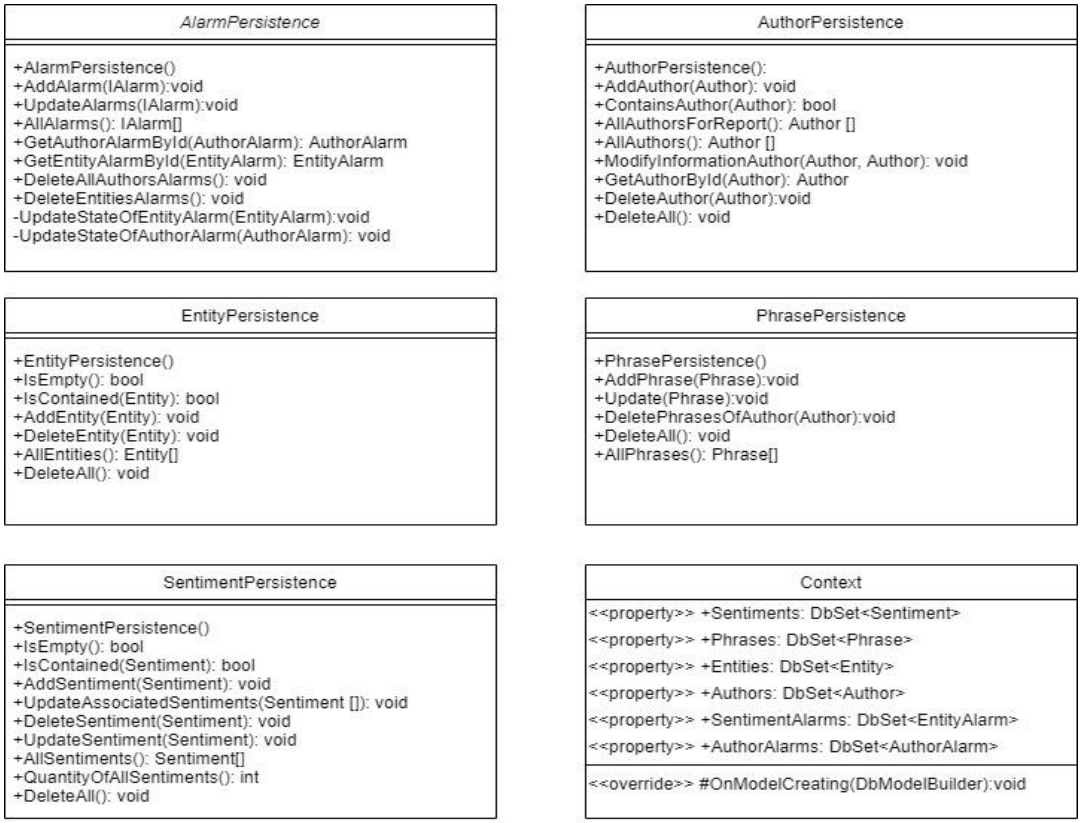
Para explicar el motivo de la creación de cada una de las clases del paquete *Persistence* utilizaremos el siguiente diagrama de clases.

Diagrama de clases de *Persistence*

En la siguiente figura, se observan las clases presentes en el paquete *Persistence*. Tal como se observa, contamos con una clase en este paquete para cada controlador presente en *BusinessLogic*. La razón por la cual decidimos realizarlo de esta forma, y no tener una única clase en *Persistence*, que se encargue de realizar todas las operaciones asociadas a la persistencia de los datos (independientemente del tipo de objeto que se quiera consultar), es que se obtiene una alta cohesión en cada clase de *Persistence,* ya que cada una de ellas realizará consultas a la base de datos acerca de objetos de un tipo específico, lo que implica que haya una alta relación entre todas las operaciones de cada clase, logrando un enfoque único a su propósito.

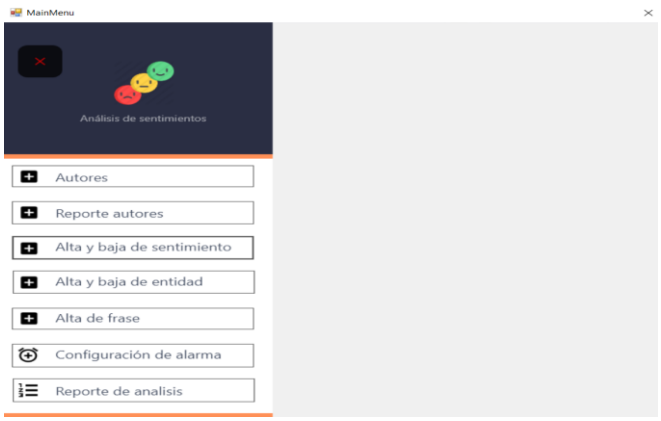
Implementando la solución de esta forma, también se aplica el Principio de Responsabilidad Única, en donde se indica que cada clase debe tener un único motivo para cambiar.

Si se hubiera implementado con una única clase en Persistence, sucedería lo contrario, ya que la misma realizaía consultas con objetos de diferentes tipos, lo que tiene como resultado una baja cohesión, ya que la relación entre las distintas operaciones de la clase sería muy baja, además de que se estaría violando el Principio de Responsabilidad Única, ya que ante un cambio de un requerimiento del sistema, lo más probable que esta clase tenga varios motivos para cambiar (por la baja relación entre sus funcionalidades).



Decisiones de diseño y principales patrones utilizados

En particular comenzaremos a detallar el funcionamiento de las nuevas funcionalidades agregadas a partir de la primera versión. Como se mencionaba anteriormente, se trata de una aplicación de escritorio la cual cuenta con una interfaz gráfica, la cual es muy similar a la primera versión lanzada, siendo que los únicos aspectos que se agregan son nuevas funcionalidades además de la persistencia de los datos. Observándose de la siguiente forma:



Las decisiones de diseño que fueron consideradas en particular para la interfaz y su visualización, se encuentran enfocadas únicamente en las nuevas funcionalidades. Siendo que nuevamente se consideró que la mejor opción era seguir utilizando diversos UserControl. Esto, para que a los usuarios no se le abrieran nuevas ventanas, haciendo que el usuario se pierda a la hora de utilizar la interfaz. 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.

Uno de los puntos claves, fue en la sección Autores, dónde se tomó la decisión que la alta, baja y modificación ocurriera todo en una misma ventana. Siendo así que el proceso para ingresar un usuario es iniciar la aplicación, hacer clic sobre el botón autores, y allí se desplegará la interfaz para la creación de autores, donde deberá ingresar los datos del mismo y clickear sobre el botón agregar. Una vez agregados, dicho usuario se carga en la lista de autores registrados, visualizandose debajo de dicha lista, los botones “Eliminar” y “Modificación”. Por lo cual para eliminar únicamente lo que se debe realizar, es seleccionar un autor de la lista de autores, y clickear el botón eliminar, lo que conlleva a que dentro del sistema se elimine al autor, además de las lista de frases que ha escrito dicho autor y las alarmas en las cuales participó. Sobre esto se especificará más adelante.
Por último para la modificación del autor, la decisión que se tomó, es que una vez seleccionado el autor de la lista de autores registrados, los datos de dicho autor se carguen en los campos de datos, donde el usuario pueda modificarlos, y luego de clickear sobre él botón modificar se confirmarán los cambios, y se modificarán los datos de dicho autor seleccionado.

La segunda gran decisión de diseño de interfaz fue tomada en el reporte de autores, dónde se consideró que era mejor tener una grilla con dos columnas. Es decir una columna dedicada únicamente a los datos de autor, y la otra columna dedicada únicamente al valor obtenido del tipo de reporte seleccionado. En lugar de tener múltiples columnas las cuales se separasen de la siguiente forma: nombre de usuario, nombre y apellido, el valor del campo elegido por el usuario para ordenar los autores (porcentaje de frases positivas/negativas, cantidad de entidades mencionadas, o promedio diario de frases).

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?

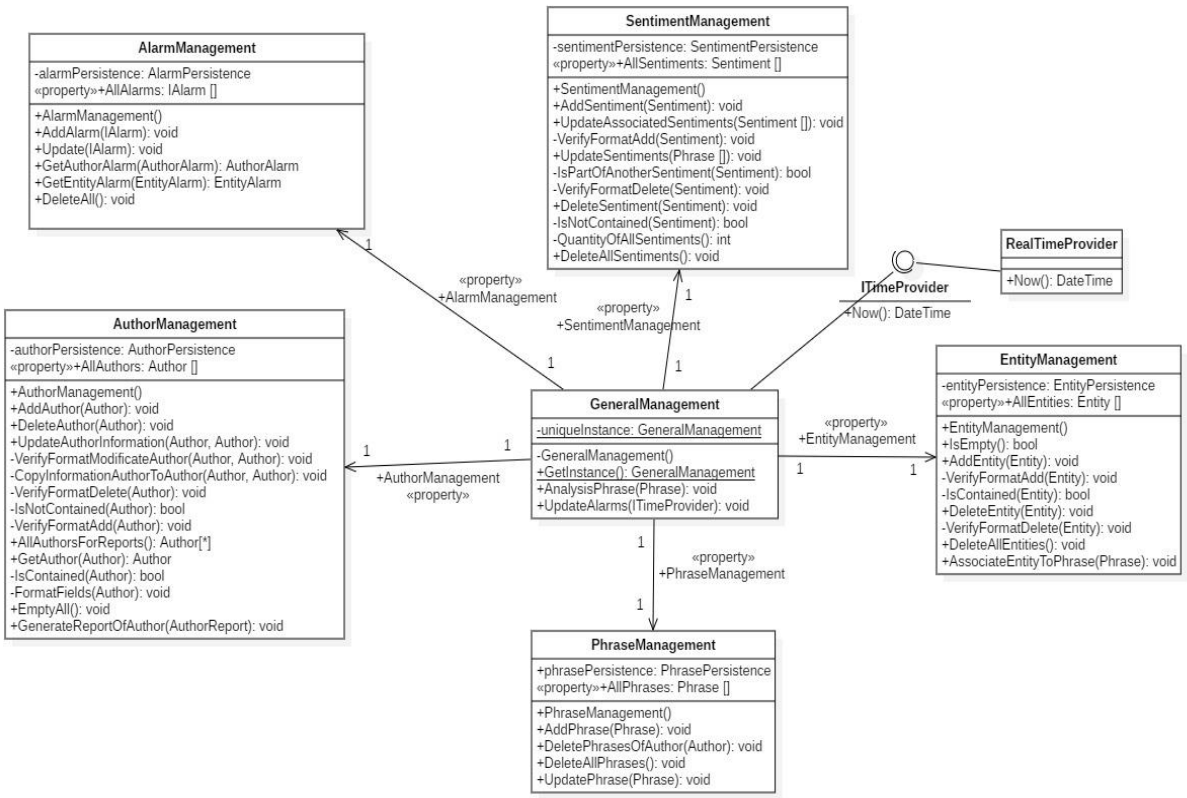
[1] Interpretado de: Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education. Página: 132-133. Página: 138.

En particular sí se desarrollara una única clase Sistema, la cual tuviera múltiples persistencias 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 tablas los diferentes tipos de objetos creados.
3. Se encargaría de eliminar a los objetos de las diferentes tablas.
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. Pero es importante garantizar que solo exista una instancia de dicho manejador general, de manera que para asegurar que solo exista una instancia de la clase *GeneralManagement* y un punto global de acceso a ella, decidimos aplicar el patrón de diseño **Singleton**. Consideramos importante realizarlo ya que solo queremos que haya una única instancia de esta clase, porque de lo contrario, si tuviéramos varias, la información estaría distribuida en las diferentes instancias, y cuando hagamos una consulta utilizando una de ellas, solo va a tener en cuenta cierta parte de la información (la que tiene esa instancia) y no la información total que hay en el sistema.
Observándose esto en la figura que se presenta a continuación:

Diagrama de clases de BusinessLogic



Además el logro que conlleva dividir una clase sistema en diferentes subsistemas, lleva al cumplimiento del **GRASP Controller**. En particular, en nuestra solución se tomó la decisión de que haya un único controlador general, el cual la responsabilidad que tiene es la instancia de los diferentes manejadores. Estos, son pequeños subsistemas, los cuales constituyen diferentes controladores, y cada uno de estos, cumplen con qué la única responsabilidad que tienen es el manejo de los objetos del tipo del objeto del Dominio, los cuales están claramente identificados dentro del nombre del controlador (siendo así que en un comienzo le llamamos manejadores, pero él termino correcto son controladores).

Otro punto importante a destacar del patrón, es que nos dice que la lógica de negocios debe estar separada de la capa de presentación, esto para aumentar la reutilización de código y a la vez tener un mayor control, lo cual se cumple en su totalidad dentro del proyecto, en el cual el dominio está dividido en su totalidad de las reglas de negocio, donde se encuentra una dependencia en que las reglas de negocio dependen del dominio.

También en consecuencia de la subdivisión de diferentes controladores, nos lleva al cumplimiento de dos patrones diferentes. Siendo estos, el patrón **GRASP de Alta cohesión** y también el patrón **GRASP de Bajo acoplamiento**.

- Alta cohesión: Es la medida en la que un módulo de un sistema tiene una sola responsabilidad. Por ende, un módulo con alta cohesión será aquel que guarde una alta relación entre sus funcionalidades, manteniendo el enfoque a su único propósito. Nótese cómo este principio se relaciona de manera casi directa con el principio de responsabilidad única desarrollado anteriormente.
  Lo cual nos lleva a decir, que sí hubiera una única clase la cual se conociera como “System”, lo que sucedería es que dicha clase tendría múltiples responsabilidades como mencionamos anteriormente, pero además tendría una baja cohesión ya que no tendría una única responsabilidad, debido a que no tendría una alta relación entre sus funcionalidades.
  Lo que nos lleva a decir que cada uno de los “manejadores” tiene una alta cohesión, ya que la responsabilidad que tienen siempre está orientada al mismo tipo de objeto del dominio, es decir no tiene múltiples responsabilidades con objetos del dominio. Cumpliendo así el GRASP de alta cohesión, debido a la subdivisión en pequeños “System´s”.

- Bajo acoplamiento: Siendo el acoplamiento la relación que se guardan entre los módulos de un sistema y la dependencia entre ellos. El bajo acoplamiento dentro de un sistema indica que los módulos no conocen o conocen muy poco del funcionamiento interno de otros módulos, evitando la fuerte dependencia entre ellos. Lo que también nos lleva a relacionarlo con el principio **SOLID abierto/cerrado**. En particular dentro de nuestra solución se cumple este patrón, ya que se tiene el controlador general, el cual no conoce el funcionamiento interno de los diferente controladores, únicamente los instancia. Además de que los diferentes controladores, tampoco tienen conocimiento del funcionamiento interno de la persistencia, esto debido a que cada controlador tiene una clase persistencia asociada.
  Lo que convella a que el acoplamiento sea mínimo, debido a que cada controlador tiene un único “controlador de persistencia asociado”, y los diferentes controladores no conocen ni al controlador de persistencia, ni al controlador de una entidad del dominio distinta al que está asociado. Conlleva a que haya elementos que no se vean afectado por el cambio

en otros, además hace una lectura más sencilla al leer de manera aislada dicho código y son elementos más fáciles de reutilizar.

Observando como beneficios dentro del código, la legibilidad del mismo ya que si se requiere analizar una determinada funcionalidad, se observará por como está diseñada la solución que dicha funcionalidad estará enfocada a uno o muy pocos módulos del proyecto. Otro beneficio que se logra siguiendo estos patrones, es la mantenibilidad del código, ya que si se requiere hacer un cambio orientado a una única funcionalidad, afectará únicamente a un módulo, esto debido a la alta cohesión y bajo acoplamiento.

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 nuevas decisiones tomadas, fue que cuando un usuario está registrando un autor, 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 autor con nombre, con el siguiente formato "Juan Pedro", se puede notar que dicho nombre entre las palabras que lo componen tiene más de un espacio, entonces lo que nuestro sistema hace es reducir dichos espacios, quedándonos con el nombre de la siguiente forma: "Juan Pedro".
En este caso se arregla de esta forma, por una cuestión de estética y gramática, ya que no tiene sentido ingresar autores con espacios innecesarios.

También otras de las decisiones que fueron tomadas, nuevamente se relaciona con autor. En particular a la hora del registro de los mismos, siendo así que el nombre de usuario del autor, no puede estar registrado. Es decir el nombre de usuario de un autor es único, ya que no va a haber dos autores, con el mismo nombre de usuario.
Por ejemplo: si un usuario intenta registrar el autor con nombre "Joaquín" y apellido "Lamela", pero con nombre de usuario "josami", y dentro del sistema se encuentra registrado el autor con nombre "Ricardo", apellido "Gutman" y nombre de usuario "josami", el sistema no permite el registro del autor con nombre "Joaquín", ya que el nombre de usuario que se escogió para dicho autor, ya existe dentro del sistema. Mientras que sí se intenta registrar nuevamente al autor con nombre "Joaquin", apellido "Lamela" y nombre de usuario "josami", y dentro del sistema se encuentra registrado el autor con nombre "Ricardo", apellido "Gutman" y nombre de usuario "JoSaMi", tampoco el sistema nos permitirá registrar al primer autor, siendo así qué no se permite nombre de usuarios iguales, sin importar mayúsculas ni minúsculas.

Nuevamente enfocándonos sobre el autor, otro punto importante que involucra tanto a las reglas de negocio como a la interfaz gráfica, está relacionado con las validaciones de los campos de datos del autor. En particular se optó de que el maximo de caracteres permitido en el nombre de usuario sea de 10 caracteres, mientras que el nombre y apellido, el máximo permitido es de 15 caracteres. Además, de que también se optó porque la fecha mínima para el ingreso de una fecha de nacimiento sea de al menos 13 años, mientras que la fecha de nacimiento máxima soportada por el sistema es hasta 100 años, con respecto a la fecha actual del sistema en ambos casos. De igual manera, esta validación se realizó también en la interfaz, de manera que el usuario pueda seleccionar únicamente fechas válidas, lo que conlleva a que no haya errores de este tipo. Siendo que si el usuario consiguiera burlar la seguridad de la interfaz, de igual manera está controlado en las reglas de negocio.
Además de que el nombre de usuario puede ser caracteres alfanuméricos, lo cual fue controlado de manera que únicamente se puedan escribir letras y números. No menos importante, también se realizó la verificación de que cuando se ingrese el nombre o apellido sean únicamente

caracteres alfabéticos, siendo así que no se permiten nombres o apellidos con caracteres especiales ni números.

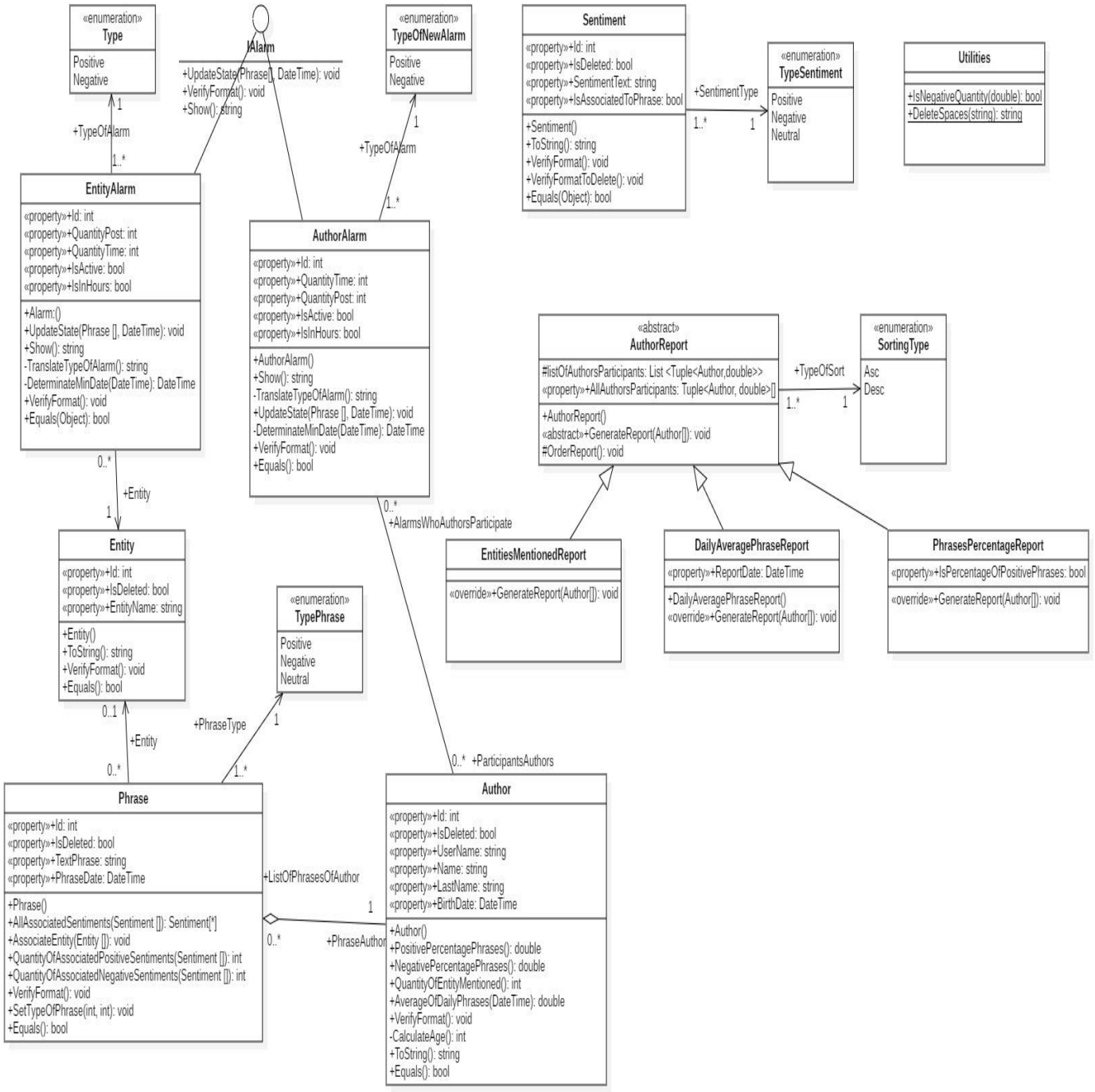
Por otra parte, un punto ajeno a los autores, pero a la vez relacionado con ellos, es que las frases, a partir de esta nueva versión de la aplicación, tienen asociado un autor que fue quién las escribió y desarrolló. Es decir, el usuario ingresa una frase, además de la fecha correspondiente de creación de la misma, y el usuario selecciona un autor de la lista de autores registrados en el sistema, lo que hace que cuando se agrega una frase, la misma tenga un autor, pero además como mencionamos anteriormente la clase “Author”, ahora tiene una lista de frases, las cuales son las que el autor escribió.Esto facilita los cálculos a la hora de realizar los reportes de autor.

Un último punto importante a destacar, es orientado a las alarmas. Con la introducción del nuevo tipo de alarmas, asociadas a los autores, se introdujeron nuevas restricciones. Siendo una de estas que la cantidad máxima de post debe ser menor o igual 1000, lo cual fue validado en las reglas de negocio, siendo que no se permite agregar una nueva alarma que supere dicha cantidad de post. Esto únicamente asociado a la nuevas alarmas.

Continuando con lo mencionado anteriormente en el diagrama de paquetes, y el funcionamiento general de la aplicación, además de las decisiones de diseño tomadas, encontramos prudente diagramar el dominio. Ya que como se visualiza en el diagrama de paquetes, las reglas de negocio tienen una dependencia con respecto al dominio.

Siendo este compuesto por entidades que nos permiten modelar el problema a resolver, para esto, consideramos en el paquetes las clases: Entity, EntityAlarm, AuthorAlarm, Sentiment, Phrase, Author, AuthorReport (la cual es una clase abstracta, cuyo motivo de creación se detallará más adelante), DailyAverageReport, EntitiesMentionedReport, PhrasesPercentageReport, Utilities, una interface IAlarm cuyo propósito se explicará más adelante, y cinco enumerados que nos permiten manejar los tipos de alarmas, sentimientos, frases y los reportes de autores.
Las clases presentes en este paquete, así como las relaciones que existen entre ellas, se pueden visualizar en el siguiente diagrama de clases.

Diagrama de clases de Domain



Manteniendo el diseño que se había realizado en la primera versión cabe destacar que 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.

Siendo así que a continuación explicaremos el motivo de la creación de la clase abstracta “AuthorReport” y sus tres clases derivadas.

Una de las nuevas funcionalidades lanzadas en esta versión es el reporte de autores.

Cuando empezamos a diseñar la solución para implementar esta nueva funcionalidad, nos cuestionamos acerca de que a qué entidad teníamos que asignarle la responsabilidad de realizar los cálculos que permitan generar el reporte.
Entonces, aplicando el **GRASP Experto,** consideramos realizar estos cálculos en la clase *Author*, ya que es quien cuenta con la información necesaria para cumplir la responsabilidad. Realizando la implementación de esta forma, se mantiene el encapsulamiento, ya que los objetos utilizan su propia información para resolver tareas, además de que se logra distribuir el comportamiento entre las clases que tienen la información requerida.

En este caso, la información necesaria para cada autor es: la cantidad de frases positivas y negativas que posteó, la cantidad de entidades mencionadas, y el promedio diario de frases. Todos estos datos los podemos obtener dentro de la clase *Author,* ya que la misma tiene una lista de frases.

Decidimos aplicar el **Open–closed principle**, que indica que las entidades de software (clases, módulos, funciones, etc), deberían ser abiertas para la extensión pero cerradas para la modificación, es decir que en este caso, se deberían poder agregar nuevos reportes de autores facilmente (sin tener que modificar el código existente).

La forma de llevar adelante esto, es mediante la abstracción, por lo que, para aplicar este principio, considerando que en total tenemos cuatro reportes diferentes, en donde se realizan cálculos diferentes, debemos aplicar el **GRASP Polimorfismo,** que indica que cuando el comportamiento varía dependiendo el tipo, debemos asignar responsabilidades a los tipos cuyo comportamiento varía a través de operaciones polimórficas.

En suma, y aplicando los patrones mencionados, decidimos realizar una clase abstracta llamada *AuthorReport*, que tiene un método abstracto que genera el reporte, el cual es sobreescrito con diferentes implementaciones por sus subclases: *DailyAveragePhraseReport*, *EntitiesMentionedReport* y *PhrasesPercentageReport.*
Finalmente, el método de *AuthorManagement* que genera el reporte, invoca al método de la clase abstracta pasando un AuthorReport, y ahí se produce el polimorfismo antes mencionado.

De una forma similar a los reportes, diseñamos la solución para el manejo de las alarmas. En la primera versión de la aplicación, se aplicó el **Open–closed principle,** porque se mencionaba que para la siguiente versión se agregarían nuevos tipos de alarma, en ese sentido el sistema estaba preparado para afrontar nuevos tipos de alarmas, sin tener que hacer grandes modificaciones en el código, ya que aplicando el **GRASP Polimorfismo,** se realizó una interface *IAlarm*, que era implementada por la clase *EntityAlarm*.
Para esta segunda versión, lo que se realizó fue crear otra clase llamada *AuthorAlarm* que también implementa la interface *IAlarm,* y por lo tanto, la operación que actualiza el estado de una alarma.

Diagramas de secuencia

Para explicar de forma más clara las principales funcionalidades del sistema, utilizaremos diagramas de secuencia, un tipo de diagrama que nos permite modelar la interacción entre objetos en un sistema y visualizar cómo estos intercambian mensajes en un orden determinado.
Las funcionalidades que consideramos representar son: actualizar una alarma, analizar una frase (determinando su tipo y asociando la entidad), y eliminar un autor (lo que también implica la eliminación de las frases escritas por ese autor).

Actualizar alarma

El mecanismo para actualizar una alarma se puede observar en el diagrama número uno. Este proceso comienza cuando un autor ingresa una frase, porque es en ese momento que se tienen que evaluar todas las alarmas con el fin de verificar si se cumple o no la condición de activación. En el controlador general se recorren todas las alarmas (objetos *IAlarm*), y se invoca para cada una de ellas el método que actualiza la alarma. Este método, tiene un comportamiento polimórfico, y dependiendo del tipo concreto que implemente la interface *IAlarm* (*EntityAlarm* o *AuthorAlarm*), tendrá un comportamiento diferente. El comportamiento en cada uno de los casos se puede apreciar en más detalle en el diagrama uno. Finalmente, luego de modificar el estado de una alarma, es necesario impactar estos cambios en la base de datos, por lo que se invoca a un método del controlador de alarma, el cual invocará a un método de la persistencia encargado de esta funcionalidad.

Analizar frase

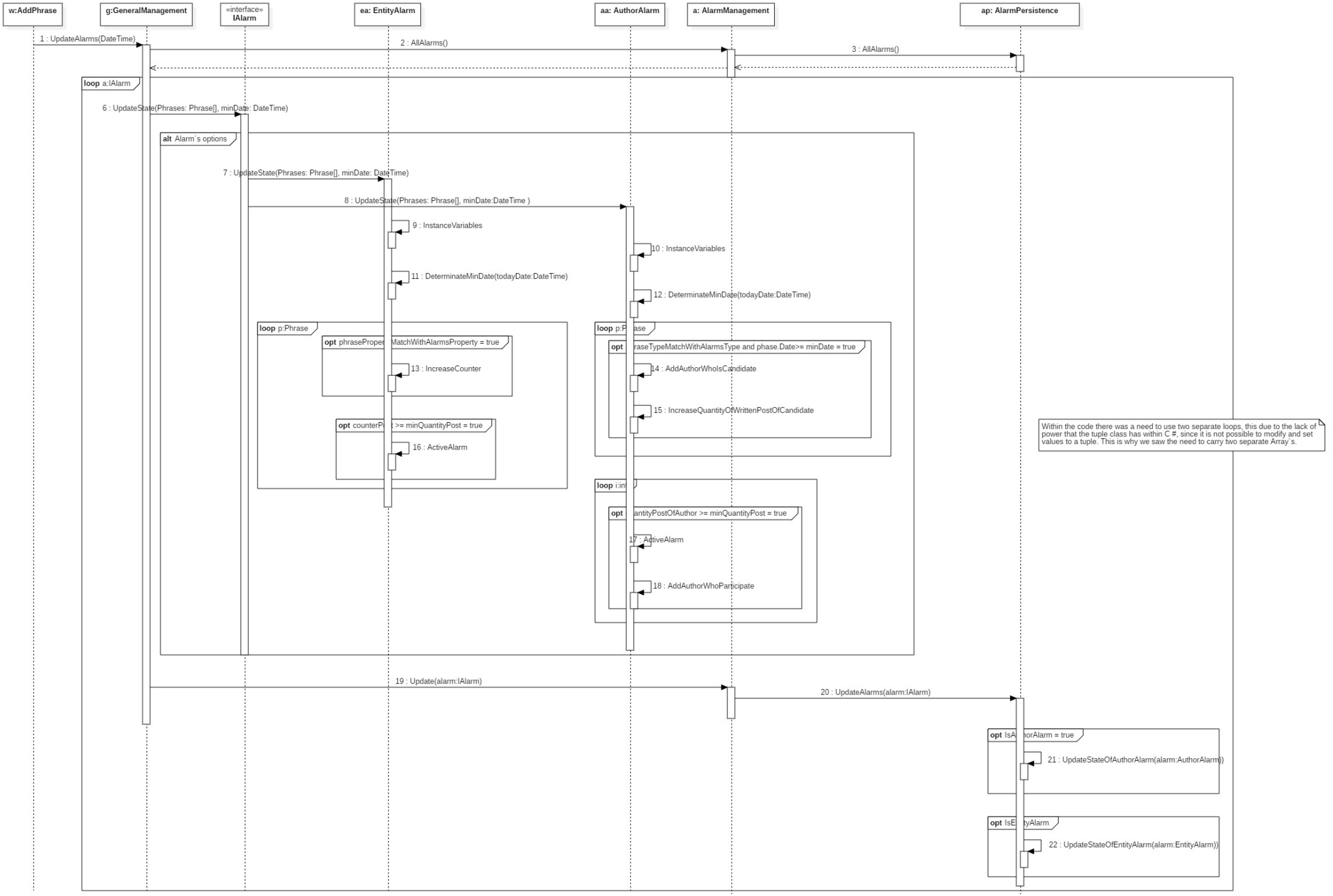
El proceso de analizar una frase sucede cuando un autor registra una de ellas en el sistema. En primer lugar lo que realizamos es agregar la frase a la persistencia directamente sin analizar (únicamente con su texto). Luego se realiza específicamente el análisis de la frase, para esto, en primer lugar, se determinan los sentimientos asociados a esa frase, luego a partir de esa lista, se determinan la cantidad de ellos que son positivos y los que son negativos, luego se actualiza el estado de esos sentimientos indicando que estos ahora están asociados a una frase (y esto se debe impactar en la persistencia también). Luego, se asocia la entidad a la frase (en la clase frase y a partir de la lista de entidades del sistema). Posteriormente, con la cantidad de sentimientos positivos y negativos asociados, se invoca a un método de la clase Phrase, que a partir de esos dos valores cataloga a la frase como: positiva, negativa o neutra.
Por último, se actualiza la frase en la base de datos, con los nuevos atributos que se identificaron en este proceso (tipo de la frase y entidad).
Para ver en detalle el funcionamiento de este proceso, ver el diagrama número dos.

Eliminar autores

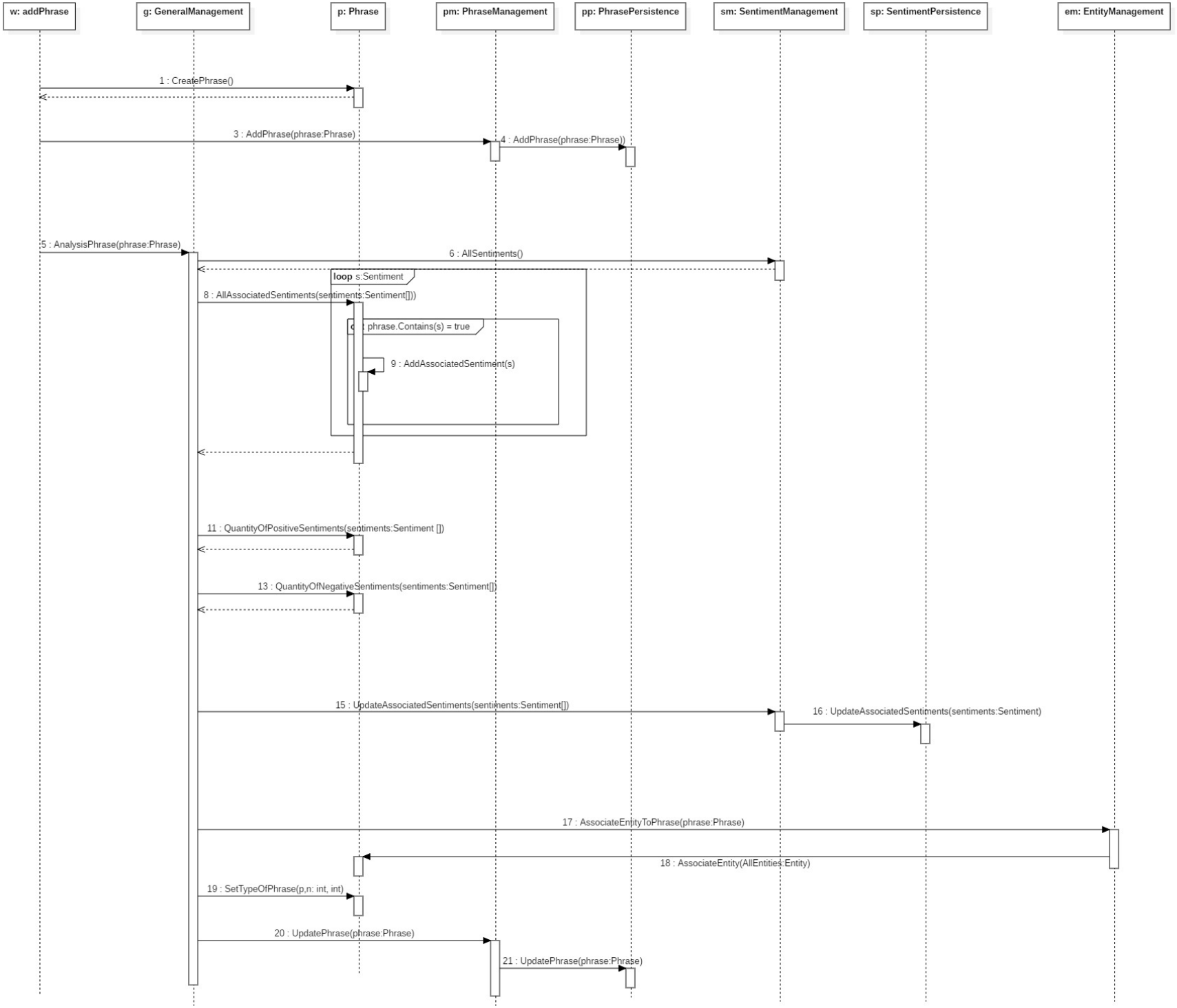
El formato en él qué ocurre la eliminación de los autores se puede observar en el diagrama número dos. Está funcionalidad se da cuando hay únicamente autores registrados dentro del sistema.
Siendo así que se comienza cuando un usuario se dirige a la ventana de Autores, allí él usuario podrá visualizar él campa para registrar nuevos autores, o modificar autores existentes o eliminarlos, a partir de la visualización de los mismos en una lista. Está lista la

particularidad que tiene, es que están contenidos en ella los usuarios registrados pero no eliminados del sistema, es decir aquellos que todavía pueden redactar frases. Por lo cual lo que sucede, es que el usuario selecciona a un usuario para eliminar y clickea sobre él botón eliminar. A partir de esto lo que sucede, se obtiene el controlador general del sistema, a partir de allí se le solicita a este que se obtenga el controlador de autores. Siendo que allí se invoca al método DeleteAuthor, al controlador de autores, lo cual produce que esté invoque a la persistencia para eliminar el autor seleccionado. Una vez finalizado está acción, se invoca al controlador de frases, donde se llama a que se eliminen las frases creadas por él autor recientemente eliminado, es así que dicho controlador llama a la persistencia para eliminar realmente las frases del sistema (ya que no tiene sentido tener frases que no tienen autor dentro del sistema). Siendo así que la última acción que se realiza, es actualizar los sentimientos, ya que al haber frases que se eliminaron puede haber algún sentimiento que no esté asociado a ninguna frase, lo cual produce que dicho sentimiento ahora se encuentre disponible para ser eliminado. Está ultima acción se hace llamando al controlador de sentimientos, dónde se le solicita que actualice los sentimientos, y esté luego de dicha acción, invoca a su persistencia de manera que se actualicen los datos de los sentimientos en la base de datos. Para lograr observar con más precisión se sugiere observar el diagrama de secuencia número tres, y además observar el código.

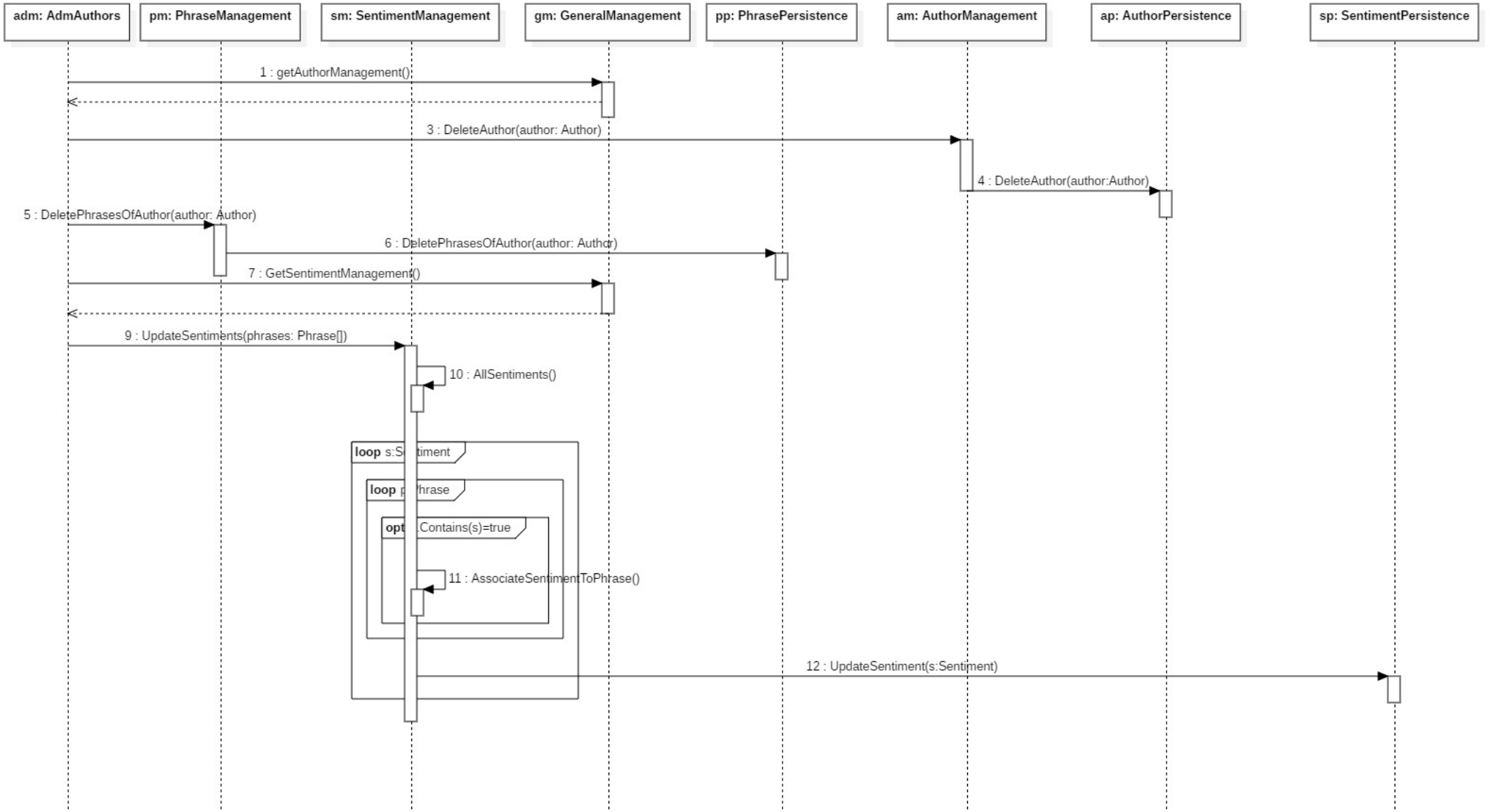
1. Actualizar alarma



2. Analizar frase



3. Eliminar autor



Pruebas unitarias

Con respecto a los tests, la nuevas funcionalidades se desarrollaron 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[2].
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.

Excepciones

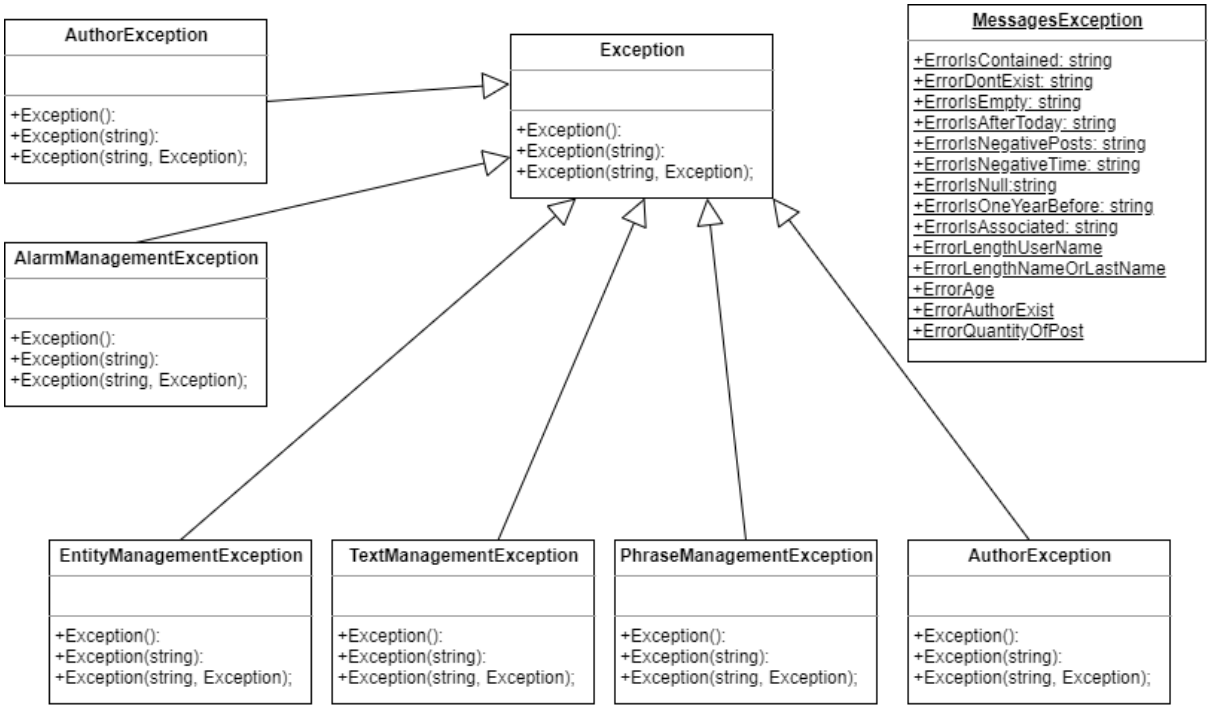
En cuanto al manejo de errores, se prefirió seguir utilizando las excepciones frente a devolver códigos de error. Básicamente ya que sí se utiliza códigos de error, él invocador debe procesar el error de forma inmediata, además de generarse una gran estructura de condicionales anidados, que hacen al 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.

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

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

En particular, para no generar mayor complejidad en el diseño de la aplicación se tomó la decisión de que la excepciones de la base de datos, estuvieran dentro del paquete llamado BusinessLogicException, para no generar un nuevo paquete que fuera DataBaseExcepction, justificandonos y basándonos en que las excepciones posibles de la base de datos, son excepciones de las reglas de negocio de la base de datos, por lo cual está relacionado con excepciones de las reglas de negocio.



Como mencionamos anteriormente el único agregado a la primera versión, es la clase AuthorException, y DataBaseException. Siendo así que se crearon para cada clase manejadora del paquete BusinessLogic, una clase de excepción. Ya 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.

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 98.95% para el paquete *BusinessLogic*, y 96.93 % 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. 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.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Agustin Hernandorena_DESKTOP...	169	3,96 %	4099	96,04 %
businesslogic.dll	3	1,05 %	283	98,95 %
businesslogicexceptions.dll	26	70,27 %	11	29,73 %
domain.dll	23	3,07 %	725	96,93 %
persistence.dll	70	8,10 %	794	91,90 %
test.dll	47	2,01 %	2286	97,99 %

Repositorio

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 abre un pull request para que el desarrollador que no trabajo en la funcionalidad pueda hacer una revisión de codigo, y posteriormente, si se aprueba el pull request 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