Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 1
Obligatorio 2

Felipe Najson (232863)
Santiago Topolansky (228360)

Entregado como requisito de la materia Diseño de Aplicaciones 1

24 de junio de 2020

Declaraciones de autoría

Nosotros, Felipe Najson y Santiago Topolansky, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos Diseño de Aplicaciones 1;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Índice general

1. Introducción

Análisis de Sentimientos es una aplicación para la detección de emociones implícitas en comentarios textuales realizados en redes sociales u otros medios de expresión. Esta clase de herramientas suele utilizarse en los estudios sobre marketing para interpretar la opinión pública sobre determinada marca, y la reacción de los clientes hacia distintas acciones de una empresa.

En esta oportunidad, el proyecto se orientó a desarrollar la tecnología de análisis de frases, permitiendo identificar los sentimientos asociados a las mismas. En cuanto a la entrada de datos, por ahora la aplicación recibe de forma manual las frases a analizar, así como todo el resto de la información pertinente (Entidades, Sentimientos y Autores de cada frase).

Además, mediante la definición de alarmas orientadas a entidades o autores, el cliente será notificado cuando se alcance un determinado número de comentarios para cierto autor o entidad en un rango de tiempo establecido.

2. Descripción general del trabajo y del sistema

El sistema desarrollado cumple con la siguiente especificación funcional:

1. **Entidades**: El sistema dispone de una interfaz gráfica con una pantalla dedicada al ingreso de las marcas/entidades de las que interesa capturar información. Es posible consultar y eliminar las entidades desde la sección "ELEMENTOS DEL SISTEMA".

2. **Sentimientos**: El sistema permite el ingreso de sentimientos, solicitando como campo obligatorio su tipo (positivo/negativo).
   Es posible consultar y eliminar sentimientos desde la sección "ELEMENTOS DEL SISTEMA".

3. **Autores**: El sistema dispone de una sección dedicada al registro de autores para las frases del sistema, se ingresan de forma manual, y requieren el ingreso de la fecha en que se registró la frase. Cabe destacar que para el ingreso de una frase al sistema es obligatorio tener un autor asociado a esa frase.

   Es posible consultar el listado completo de autores desde la sección de "AUTORES".
   Además en esta ventana también es posible ordenar a los autores según alguno de los siguientes criterios:

   a) Porcentaje de frases positivas o negativas sobre el total de frases que generó cada uno de los autores.

   b) Cantidad de entidades mencionadas en las frases de cada auto

   c) Promedio diario de frases de cada autor

4. **Frases**: El sistema provee una sección dedicada a la "captura" de las frases que contienen información de relevancia de las entidades del sistema. Se ingresan de forma manual, y requieren el ingreso de la fecha en que se registró la frase así como el autor responsable.
   Es posible consultar el listado completo de frases desde la sección de "ELEMENTOS DEL SISTEMA".

5. **Análisis**: La funcionalidad central del sistema radica en el algoritmo de análisis de las frases que ingresan. Las frases/comentarios se analizan en búsqueda de alguna de las entidades y sentimientos del sistema, asociándoles una categoría (positiva, negativa o neutra).
   Según las reglas de negocio definidas, si la frase contiene más de una entidad, se asocia el resultado a la primer entidad identificada. Además, si contiene más de un sentimiento de cada tipo, o no contiene ningún sentimiento, inmediatamente se cataloga como comentario neutro.
   Es posible consultar el listado de todos los resultados de análisis del sistema en la sección de "ANÁLISIS".

6. **Alarmas**: El sistema cuenta con una sección dedicada a la configuración de alarmas, en este momento el sistema cuenta con dos configuraciones de alarmas diferentes, una centrada en los autores y otra centrada en las entidades. Se notifica al cliente en caso de que se alcance un determinado número de posteos (positivos/negativos) asociados a una entidad o autor del sistema, para una determinada ventana de tiempo que es dinámica (X cantidad de días/horas hacia atrás en el momento de evaluación).

   Todas las alarmas del sistema se re-analizan cada vez que entra una nueva frase al sistema, más allá de que la frase esté o no relacionada a la alarma. El sistema notifica mediante el notificación cuando alguna alarma ha sido activada.
   Es posible consultar el listado completo de alarmas configuradas en la sección de "ALARMAS DEL SISTEMA".

Al momento de la entrega todos los bugs detectados han sido corregidos. No tenemos conocimiento de ninguna funcionalidad que no funcione correctamente según lo especificado.

3. Estructura del Sistema

3.1. Organización del Proyecto

La solución esta dividida en 5 paquetes:

- **Domain**: Contiene las clases centrales que componen el sistema, y responden a las reglas de negocio que determinan cómo la información es creada, almacenada y modificada.
- **Logic**: En este paquete se realiza la gestión de la información. Cada clase de Logic es un controlador que responde a la entrada del usuario y realiza interacciones en los objetos del modelo de datos. El controlador recibe la entrada, la valida y luego pasa la entrada al modelo.
- **Persistence**: El paquete de persistencia contiene por un lado la configuración de los parametros y relaciones entre tablas de la base de datos y por otro lado clases que permiten agregar, quitar o consultar en la base de datos, estas clases de persistencia guarda los elementos del sistema de forma permanentemente en la base de datos.
- **Tests**: Es la parte del sistema que contiene las pruebas unitarias de las clases del paquete Domain, Logic y Persistence controlando el correcto funcionamiento de las mismas.
- **UI**: Es la parte que sirve de interacción con los usuarios. Es responsable de recolectar los datos de entrada del usuario, que pueden ser de variadas formas, y validarlos y transformarlos, ajustándolos a las especificaciones que demanda el backend para poder procesarlos. También se encarga de reportar errores de parte del usuario previo al ingreso de la información del sistema. Intenta ser llamativo y permitir un manejo ágil de las funcionalidades.

Aquí se representan las relaciones entre los diferentes paquetes de la solución:

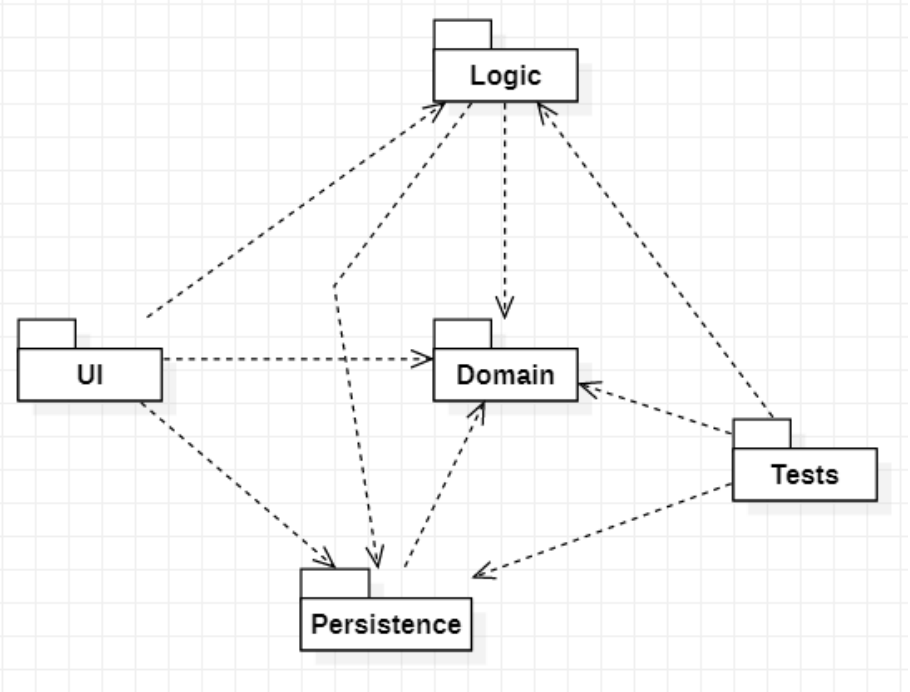


Figura 3.1: Diagrama de Paquetes

4. Descripción y justificación de diseño

4.1. Mejoras de diseño en Versión 2.0

4.1.1. Clases No-Anémicas

En la primera versión de la aplicación las clases del sistema se diseñaron siguiendo un principio denominado [1]"Clases Anémicas" según el cuál las clases no debían contenter lógica alguna, más allá de los métodos básicos de construcción y comparación.
En una etapa avanzada del proyecto nos informamos mejor al respecto y llegamos a la conclusión de que no era deseable seguir esa estrategia de diseño, ya que se contradecía con los principios fundamentales del paradigma de la [4] **Programación Orientada a Objetos**, y por limitaciones de plazos lo dejamos pendiente de cambio para la siguiente versión.

En esta versión, se trasladaron las responsabilidades propias de cada clase a ellas mismas (métodos de gestión interna). Por ejemplo, la clase **Analysis** ahora es capaz de ejecutar el análisis por sí mismo por medio del método **ExecuteAnalysis**, ahí contenido. Anteriormente, en la V1, el análisis se ejecutaba desde un manejador externo llamado AnalysisLogic.

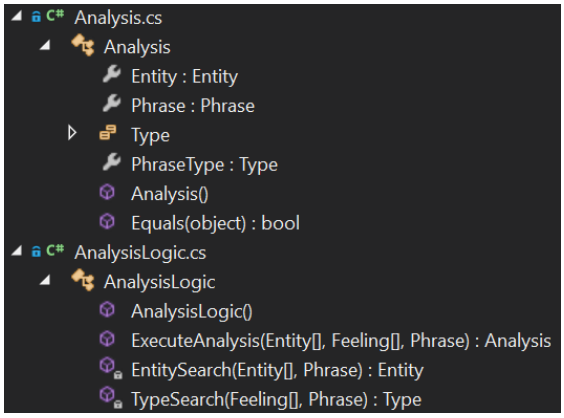


Figura 4.1: Antes: Clase Anémica

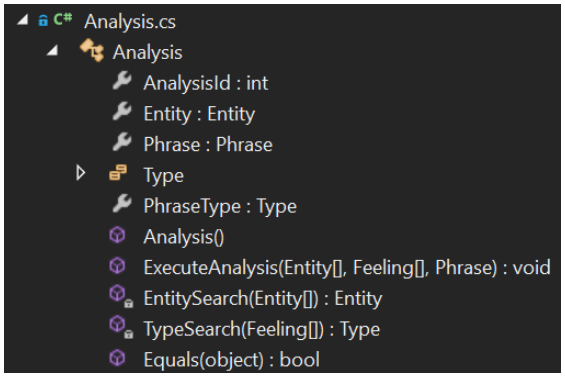


Figura 4.2: Después: Clase con responsabilidad

4.1.2. Fraccionamiento de clase FeelingAnalyzer

Otra mejora en la organización de nuestras clases respecto a la primera versión del sistema fue el fraccionamiento de la clase **FeelingAnalyzer**. Esta clase reunía muchas responsabilidades distintas relacionadas a la persistencia de la aplicación (gestionando las listas de cada tipo del dominio en memoria) y a la lógica de negocio de la misma (conteniendo métodos con algoritmos claves para el funcionamiento del sistema como la activación de Alarmas).

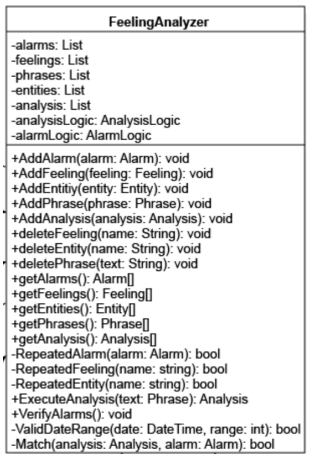


Figura 4.3: Antes: Clase FeelingAnalyzer

Como se verá en la sección 4.3 en mayor detalle, en pos de aplicar algunos de los patrones de diseño comentados en el curso, se dividió esta gran clase, en varios **"Sub Sistemas"**, cada uno de ellos especializado en la gestión de las reglas de negocio de las distintas clases del Dominio. Denominamos a cada subsistema según el formato [**NombreClase**]**Logic** y los agrupamos juntos en el paquete Logic.

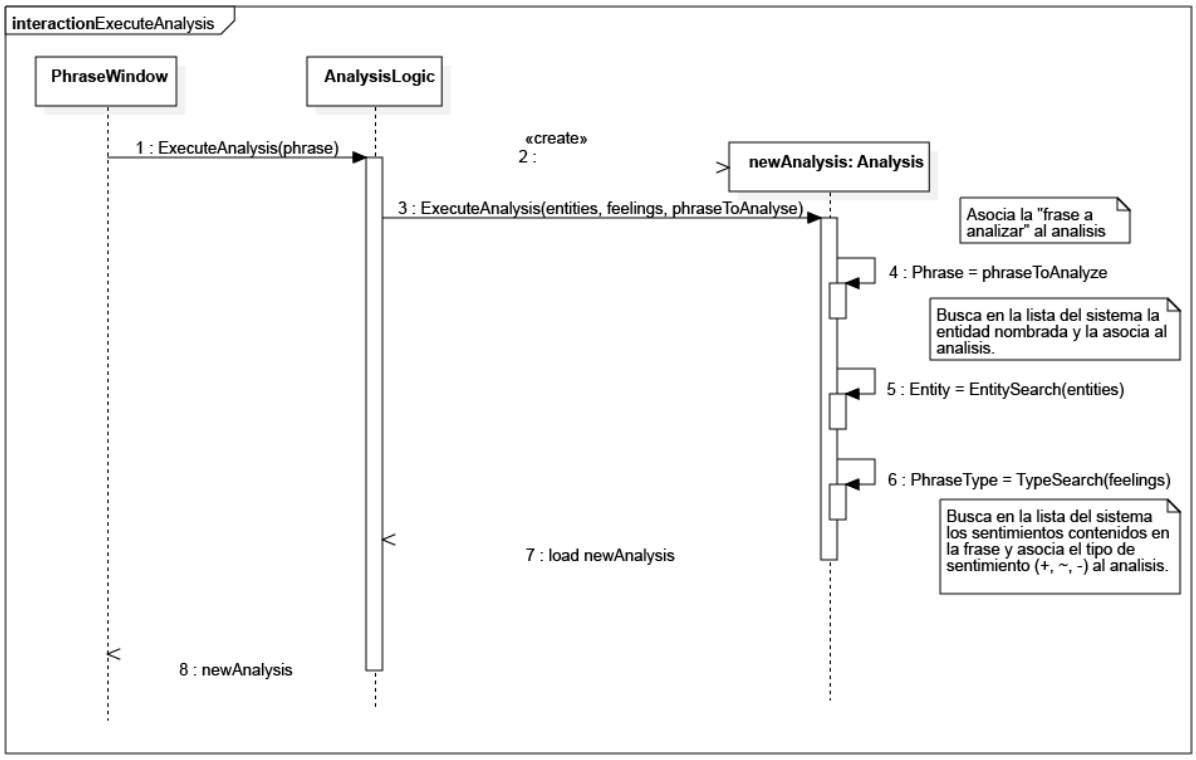
Como ya se explicó en la sección 3.1, también se migró toda la lógica relacionada a la persistencia (antes en Listas nativas en memoria), a un nuevo paquete dedicado especialamente al guardado y gestión de los datos, llamado **Persistence**.

4.2. Funcionalidades Clave

Para algunas de las funcionalidades más complejas e importantes del sistema decidimos utilizar diagramas UML de secuencia. De esta manera podemos reflejar el funcionamiento interno con el nivel de abstracción deseado.

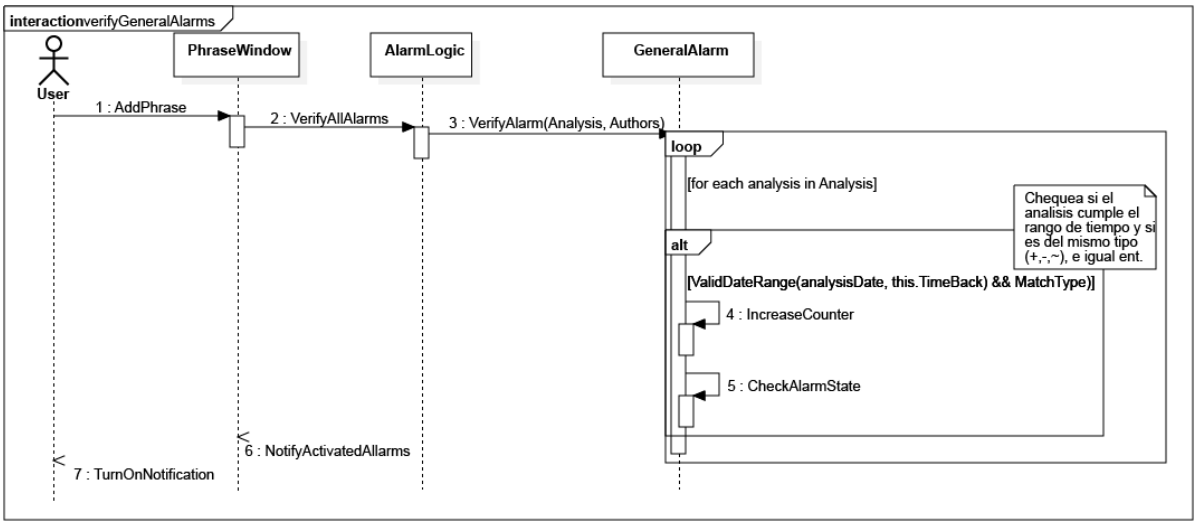
Ejecución de Análisis de Frase

Funcionalidad de análisis de una nueva frase ingresada al sistema.



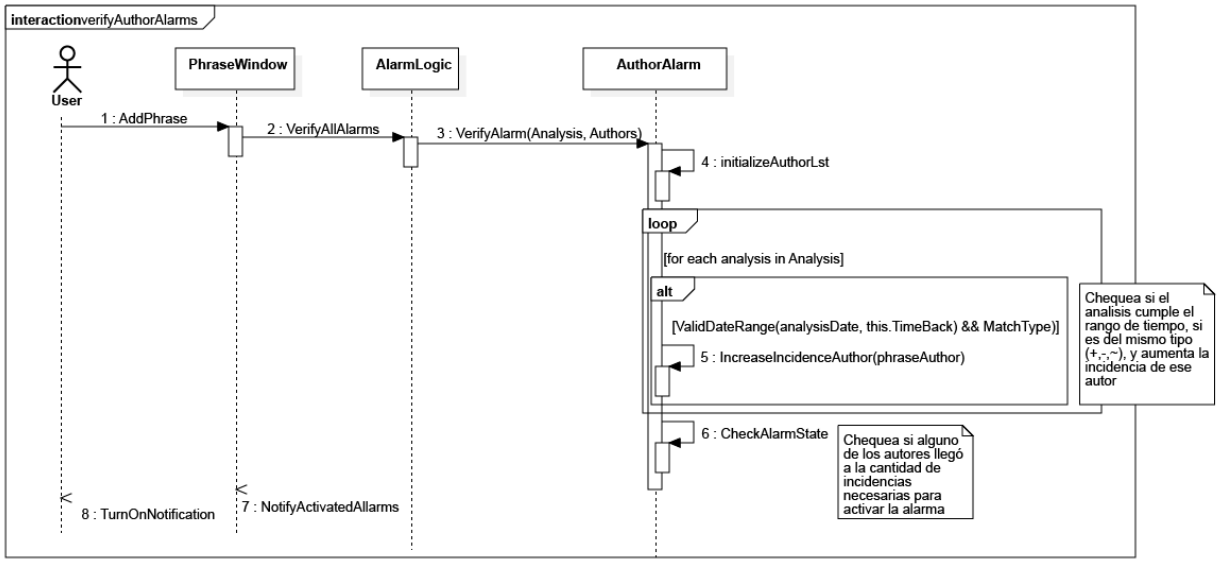
Activación de Alarmas Generales

Funcionalidad de chequeo de activación en las alarmas de tipo general (Orientadas a las entidades).



Activación de Alarmas de Autor

Funcionalidad de chequeo de activación en las alarmas de tipo Autor (Orientadas a la evaluación del comportamiento de autores).



4.3. Patrones de Diseño Aplicados

4.3.1. GRASP

Bajo Acoplamiento

Se trata de asignar responsabilidades a las clases de forma de mantener el acoplamiento entre ellas bajo. Esto se traduce en un menor riesgo a la hora de introducir cambios en una de las clases vinculadas.

Logramos mantener un bajo acoplamiento en nuestras clases de lógica de negocios (Paquete Logic), haciendo uso de la fragmentación de la clase FeelingAnalyzer en pequeños subSistemas.

Naturalmente, se sigue dando una interconexión lógica entre los mismos al utilizarse unos dentro de otros. Aún así las responsabilidades de cada uno están claramente diferenciadas según las clases que gestionan, y por eso residen en clases separadas.

También dividimos las responsabilidades de persistencia según el tipo de operación. Se dividió el paquete en tres clases: una responsable de la operación de los sub sistemas, otra responsable de limpiar la base de datos para las pruebas, y otra responsable de contener los algoritmos de listado de Autores, para los reportes en UI.

Alta Cohesión

La cohesión es una medida de tan fuertemente relacionadas están las responsabilidades de una clase.
En nuestro caso logramos una alta cohesión en todos nuestros paquetes.

- Paquete Domain: Cada clase del dominio tiene sus responsabilidades claramente asignadas, realizando únicamente tareas que le compete a si misma.
- Paquete Logic: Se dividió el paquete Logic en **una clase de lógica por clase de dominio**, justamente para lograr esta alta cohesión. Cada clase gestiona y realiza tareas de su respectiva entidad en el dominio.
- Paquete Persistence: Como se comentaba en el punto anterior, también se dividió el paquete persistence en tres clases, las cuales tienen claramente definida su finalidad (Manipular datos, limpiar base de datos y ordenar información).

En este paquete se realiza la gestión de la información. Cada clase de Logic es un controlador que responde a la entrada del usuario y realiza interacciones en los objetos del modelo de datos. El controlador recibe la entrada, la valida y luego pasa la entrada al modelo.

Controlador

El patrón Controlador recomienda tener un responsable claro de manejar los eventos externos al sistema.

Siguiendo este patrón fue que decidimos introducir un paquete nuevo **Persistence** (respecto a la versión anterior) exclusivamente dedicado a la manipulación de la base de datos, que se puede considerar como un servicio externo.

Esta clase coordina la actividad en la base de datos, cumpliendo así un rol de **bisagra** entre el interior de la aplicación y el servicio exterior.

Polimorfismo

Cuando el comportamiento varía dependiendo del tipo de dos clases emparentadas, este patrón ofrece una solución mediante la asignación de responsabilidad a los tipos cuyo comportamiento varía a través de operaciones polimórficas.

En el dominio de nuestro problema, los dos tipos de alarma tienen un comportamiento similar aunque no exactamante igual. Las alarmas generales se orientan a alertar sobre comentarios respecto a una entidad particular, mientras que las alarmas de Autor, se orientan a evaluar el comportamiento de los autores.

Para resolver el problema utilizamos polimorfismo. Creamos una clase abstracta **Alarm** que contiene todos los atributos comunes de ambas alarmas, y requiere la sobreescitura del método **VerifyAlarm()** y **checkAlarm()**.

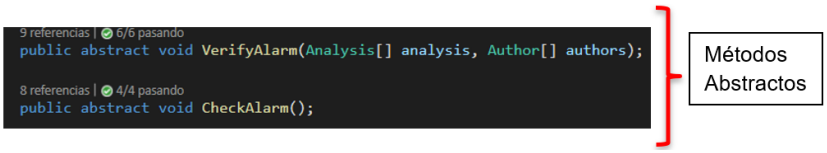


Figura 4.4: AuthorAlarm

Figura 4.5: GeneralAlarm

4.3.2. SOLID

Los principios [6] S.O.L.I.D refieren a un conjunto de 5 técnicas de diseño orientadas al desarrollo de calidad en [4]P.O.O. Aplicar estas estrategias favorece la mantenibilidad y escalabilidad del software. Fueron introducidas por Robert C. Martin en el año 2000.

- **S - Principio de Responsabilidad Única**: Este principio propone que las clases deberían tener una y solo una razón para cambiar. Se encuentra fuertemente vinculado a la alta cohesión y al bajo acoplamiento buscado en GRASP.

  En nuestro trabajo buscamos respetar el principio de responsabilidad única en la separación de sub sistemas del paquete Logic, y en la separación en clases con distintas responsabilidades en el paquete Persistence.

- **O - Abierto/Cerrado**: La premisa detrás de este principio es que se debería poder extender el funcionamiento de una clase sin modificarla (abierto al cambio, cerrado a la modificación).
  La forma por excelencia de lograr este cometido es mediante la implementación de clases abstractas e interfaces, que se pueden implementar múltiples veces para extender el comportamiento, sin necesidad de realizar cambios en las clases que las utilizan.
  Como explicamos en la sección 4.3.1, utilizamos polimorfismo para facilitar la modificación del comportamiento de las alarmas, a la vez de proteger a la aplicación de futuros cambios en estas clases.

- **L - Principio de sustitución de Liskov**: Todo método que use referencias a objetos de clases base, debe poder usar objetos de sus clases derivadas sin saberlo. No nos enfocamos en aplicar particularmente este principio, pero nuevamente la herencia en la clase alarma permite la utilización de las alarmas sin saber el tipo en tiempo de ejecución.

- **I - Principio de segregación de Interfaces**: El principio postula que es bueno tener varias interfaces especificas para cada cliente antes que una sola que provea un servicio general. La idea de trasfondo es pensar en el consumidor del servicio a la hora de diseñarlo.

  Esta idea fue la que nos motivó a separar la clase Repositorio en 3 clases distintas. La situación era que ciertos métodos de dicha clase, que permitían borrar todos los elementos de cada **DBSet**, solo se utilizaban en un consumidor (paquete Tests). Lo mismo pasaba con los métodos que listaban autores utilizando distintos criterios, que implicaban mucha lógica y solo se utilizaban en un **Form** específico del paquete UI. Los demás métodos se usaban para el funcionamiento de los sub sistemas en todos los demás casos de uso de la aplicación.
  Finalmente creamos las clases **Repository**, **Listing**, y **RepositoryCleaner**.

- **D - Principio de Inversión de Dependencia**: Para evitar la dependencia de los servicios externos, este principio indica que se debe utilizar clases contractuales que especifiquen el comportamiento deseado, y que los proveedores de servicio deban implementar. No hicimos ninguna implementación de este principio.

4.3.3. Clean Code

El código nos parece es uno de los aspectos mas importantes, representa el nivel de especificación final y contiene detalles que no pueden ser ignorados. Para mantener un código prolijo y limpio utilizamos algunas buenas practicas mencionadas en el libro "Clean Code" de Robert Martin.

Aquí las practicas que se aplicaron sobre el sistema:

- **Nombres**
  - Nombres nemotécnicos: La mayoría de las variables del sistema utilizan nombres nemotécnicos, excepto el paquete tests donde se realizan pruebas unitarias.
  - Desinformación: Se evito la desinformación de colecciones de objetos, algunas de las colecciones eran llamadas listas pero eran arrays.
  - Nombres pronunciables: Se tuvo en cuenta que los nombres de variables y métodos sean pronunciables
  - Utilización de constantes: Se utilizaron constates para algunas clases del sistema.
  - Nombres de Métodos: Siempre se incluye un verbo para nombrar a cada uno de los métodos y se trato de asignar nombres que describan la tarea a realizar. Todos los métodos comienzan con mayúscula según estándar del lenguaje.
- **Funciones**
  - Única Tarea: Un método realiza una única tarea.
  - Step-Down Rule: Las funciones se leen desde arriba hacia abajo.
  - Pocos parámetros: Se intento mantener funciones con pocos parámetros. Las únicas clases que reciben muchos para metros son las ventanas las cuales se pasan los subsistemas utilizados por las mismas.
- **Comentarios**
  - Código auto explicativo: Se intento hacer el código lo mas explicativo posible, evitando utilizar comentarios de no ser necesarios.
  - Clarificación: Para algunos métodos fue necesario clarificar el código utilizando comentarios objetivos y concisos.
  - Evitamos marcadores de posición / agrupadores.
- **Formato**
  - Formato vertical
    - Apertura vertical: Los métodos se separaron 1 linea verticalmente.
    - Declaración de atributos al comienzo.

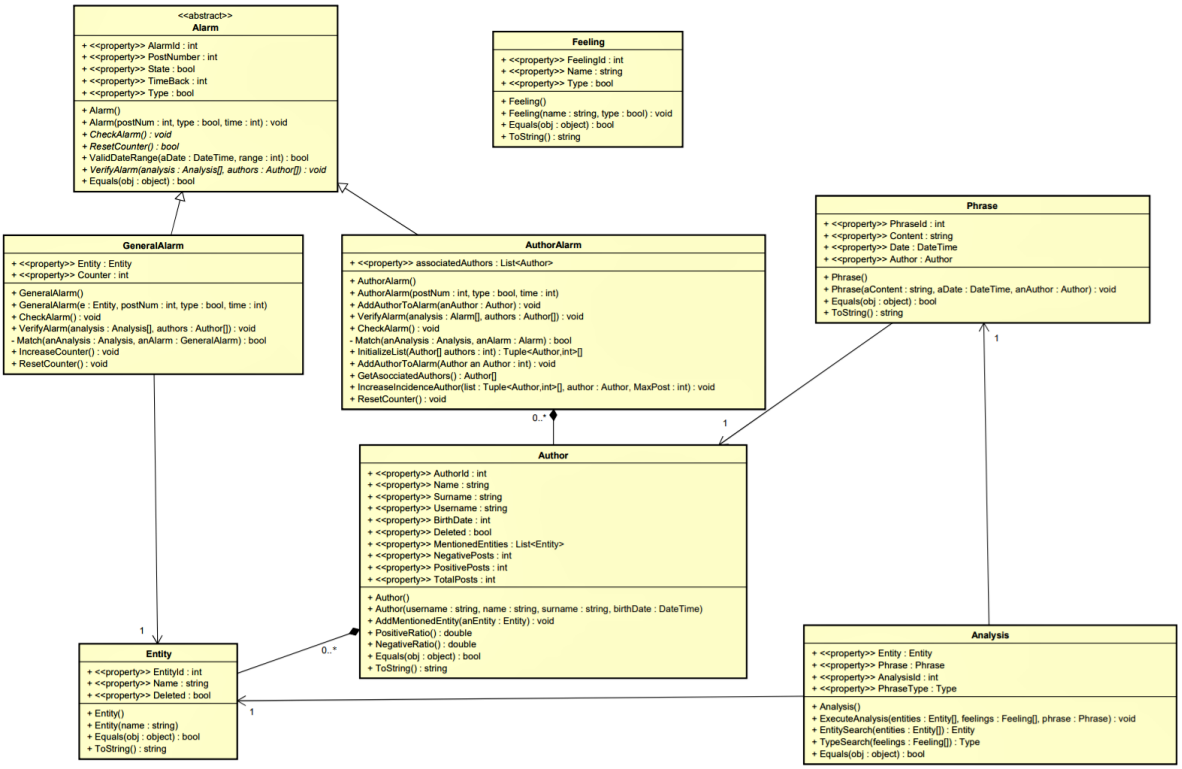
    - Ordenamiento de funciones por dependencia.
  - Formato horizontal
    - Largo de lineas: La mayoría de las lineas tienen un largo menor a 100 caracteres
    - Identación: Se utilizo una correcta identación de métodos y clases.
    - Llaves según del estándar del lenguaje

- **Manejo de Errores**
  - Usar excepciones en lugar de retornar códigos de error
  - Escribir primero try-catch

5. Paquete: Domain

5.1. Organización del paquete

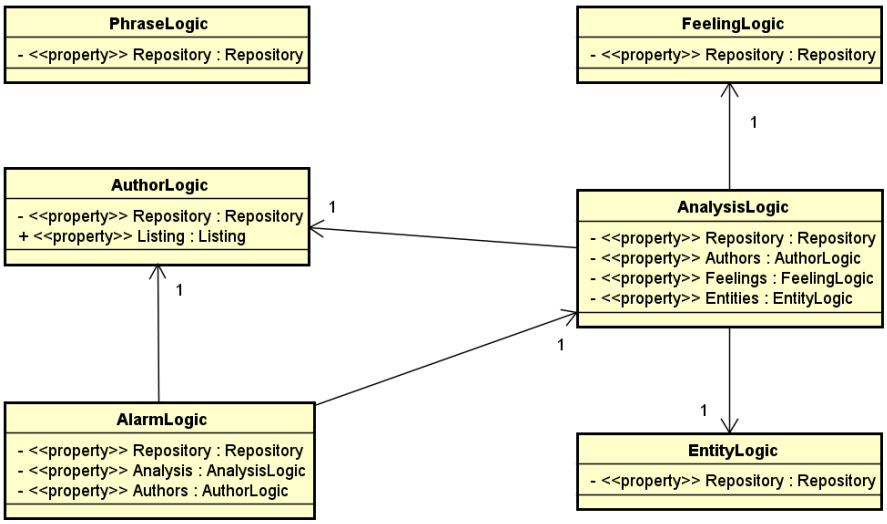
A continuación se presenta el diagrama de clases UML para el dominio, donde se denotan claramente los distintos tipos de relaciones entre las clases, con roles, cardinalidades y dependencias.



- **Asociación**:
    1. **Analysis - Phrase**: Los objetos Analysis poseen una instancia de Phrase, que no comparten con ninguna otra entidad.
    2. **Analysis - Entity**: Los objetos Analysis poseen una instancia de Entity, que no comparten con ninguna otra entidad.
    3. **GeneralAlarm - Entity**: Los objetos GeneralAlarm poseen una instancia de Entity, que no comparten con ninguna otra entidad.
    4. **Phrase - Author**: Los objetos Phrase poseen una instancia de Author, que no comparten con ninguna otra entidad.
- **Composición**:
    1. **AuthorAlarm - Author**: AuthorAlarm posee una lista de objetos Author, que no comparte con ninguna otra entidad del sistema.
    2. **Author - Entity**: Author posee una lista de objetos Entity, que no comparte con ninguna otra entidad del sistema.
- **Herencia**:
    1. **Alarm - GeneralAlarm**: GeneralAlarm hereda de la clase abstracta Alarm implementando todos sus metodos abstractos y agregando dos atributos nuevos: una instancia de Entity y un contador de tipo int.
    2. **Alarm - AuthorAlarm**: AuthorAlarm hereda de la clase abstracta Alarm implementando todos sus métodos abstractos y agregando un atributo nuevo: una lista de objetos Author.

6. Paquete: Logic

6.1. Organización del Paquete



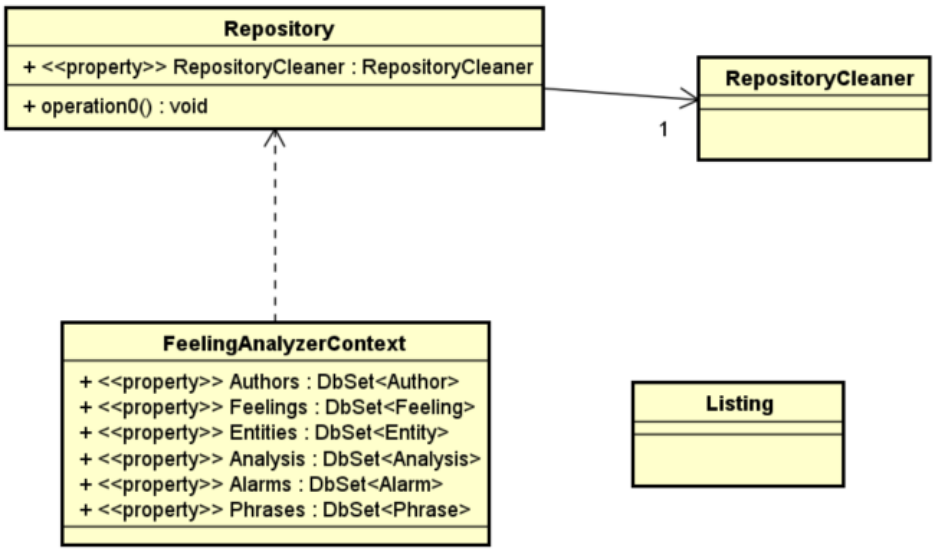
- **Asociación**:

  1. **AlarmLogic - AuthorLogic**: Los objetos AlarmLogic poseen una instancia de AuthorLogic, que no comparten con ninguna otra entidad.
  2. **AlarmLogic - AnalysisLogic**: Los objetos AlarmLogic poseen una instancia de AnalysisLogic, que no comparten con ninguna otra entidad.
  3. **AnalysisLogic - AuthorLogic**: Los objetos AnalysisLogic poseen una instancia de AuthorLogic, que no comparten con ninguna otra entidad.
  4. **AnalysisLogic - EntityLogic**: Los objetos AnalysisLogic poseen una instancia de EntityLogic, que no comparten con ninguna otra entidad.
  5. **AnalysisLogic - FeelingLogic**: Los objetos AnalysisLogic poseen una instancia de FeelingLogic, que no comparten con ninguna otra entidad.

NOTA: Se obviaron los métodos de las clases ya que no son relevantes para lo que se quiere expresar.

7. Paquete: Persistence

7.1. Organización del Paquete



- **Dependencias**

    1. **Repository - FeelingAnalyzerContext**: Repository interactúa brevemente con los objetos de FeelingAnalyzerContext creandolos y prestando servicios sin mantener ninguna referencia hacia los mismos.

- **Asociación**:

    1. **Repository - RepositoryCleaner**: Los objetos Repository poseen una instancia de RepositoryCleaner, que no comparten con ninguna otra entidad.

NOTA: Se obviaron los métodos de las clases ya que no son relevantes para lo que se quiere expresar.

7.2. Almacenamiento de Datos

7.2.1. Entity Framework

En esta nueva versión la aplicación cumple con la especificación de persistencia en base de datos. Para esto se utilizó el **ORM** "Entity Framework"[2], soportado por Microsoft en el desarrollo de aplicaciones .NET.

Entity framework nos habilitó a trabajar con datos usando las clases específicas del dominio, sin necesidad de enfocarnos en las estructuras de tablas de la base de datos subyacentes.

Utilizamos el enfoque **Code First**, lo que implica la priorización de la construcción del dominio de la aplicación frente al diseño de la estructura específica de tablas. Esto es posible ya que mediante un mapeo casi automático del dominio, EF es capaz de construir un modelo concreto de tablas sobre el que la aplicación puede trabajar.

Sin embargo, también fue necesario realizar ciertos ajustes específicos para indicar la forma exacta en que deseabamos que trabajara el modelo generado, mapeando ciertas características del dominio como relaciones y cardinalidades, restricciones de columnas, y gestión de jerarquias.

7.2.2. Fluent API

Para esto hicimos uso de Fluent API[3], una forma avanzada de especificar configuraciones del modelo, superior a las Data Annotations que se realizan directamente sobre las clases del dominio.

Fluent API se utiliza sobreescribiendo el método **OnModelCreating** del contexto. Allí especificamos el mapeo de las relaciones, esencial para indicar a EF que ciertos elementos ya existen en el contexto, en los casos de atributos de clase de otros tipos (también almacenados en la base).

Esto se hace especificando características de los DbSets de cada tipo, según las siguientes referencias

Método	Significado
.HasOptional	Puede estar relacionado a un atributo del otro tipo
.HasRequired	Debe estar relacionado a un atributo del otro tipo
.HasMany	Puede estar relacionados con muchos atributos del otro tipo
.WithMany	Que a su vez pueden estar relacionados a muchos atributos del primer tipo

Como se ve a continuación, también fue necesario mapear especialmente la jerarquía de alarmas, para indicar que se deseaba almacaner las clases derivadas (AuthorAlarm y GeneralAlarm) en dos tablas independientes, además de la tabla para la clase padre con los atributos comunes.

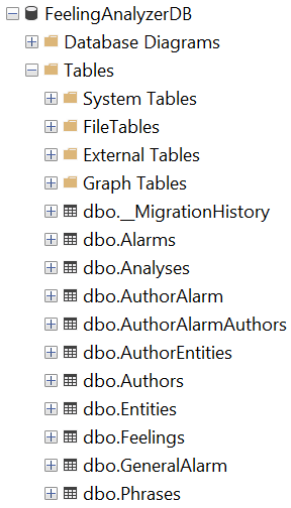
```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    //Inheritance
    modelBuilder.Entity<GeneralAlarm>().ToTable("GeneralAlarm");
    modelBuilder.Entity<AuthorAlarm>().ToTable("AuthorAlarm");

    //Relations
    modelBuilder.Entity<Analysis>()
        .HasOptional(e => e.Entity)
        .WithMany();

    modelBuilder.Entity<Analysis>()
        .HasOptional(p => p.Phrase)
        .WithMany();

    modelBuilder.Entity<Phrase>()
        .HasOptional(a => a.Author)
        .WithMany();

    modelBuilder.Entity<Author>()
    .HasMany<Entity>(e => e.MentionedEntities)
    .WithMany();

    modelBuilder.Entity<AuthorAlarm>()
    .HasMany<Author>(e => e.AssociatedAuthors)
    .WithMany();

    modelBuilder.Entity<GeneralAlarm>()
    .HasRequired(e => e.Entity)
    .WithMany();
}
```

Mapeo de herencia de alarmas

Mapeo de relaciones

7.2.3. Modelo de Tablas

Finalmente, el modelo de tablas generado con EF es el siguiente:



- Alarms
    1. AlarmID (PK, int, not null)
    2. PostNumber
    3. Type
    4. TimeBack
    5. State
- Analyses
    1. AnalyisisID (PK, int, not null)
    2. PhraseType
    3. Entity_EntityID (FK,int,null)
    4. Phrase_PhraseID (FK,int,null)
- AuthorAlarm
    1. AlarmID (PK, FK, int, not null)
- AuthorAlarmAuthors
    1. AuthorAlarm_AlarmID (PK, FK, int, not null)
    2. Author_AuthorID(PK, FK, int, not null)
- AuthorEntities
    1. Entity_EntityID (PK, FK, int, not null)
    2. Author_AuthorID(PK, FK, int, not null)
- Authors
    1. AuthorID (PK, int, not null)
    2. Username
    3. Name
    4. Surname
    5. BirthDate
    6. TotalPosts
    7. PositivePosts
    8. NegativePosts
    9. Deleted
- Entities
    1. EntityID (PK, int, not null)
    2. Name

3. Deleted

- Feelings
  1. FeelingID (PK, int, not null)
  2. Name
  3. Type
- GeneralAlarm
  1. AlarmID (PK, int, not null)
  2. Entity_EntityID
  3. Counter
- Phrases
  1. PhraseID (PK, int, not null)
  2. Content
  3. Date
  4. Author_AuthorID

7.2.4. Borrado Lógico

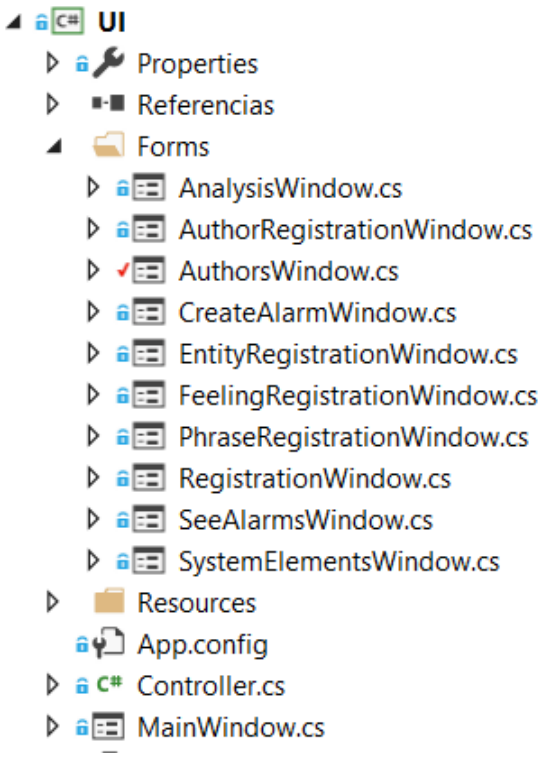
Para la eliminación de aquellas entidades que lo requieren (clases Entidad y Autor), optamos por la eliminación lógica. Utilizando un atributo de tipo booleano para indicar el estado deleted (True/False) de dicha entidad, "marcamos" a los elementos eliminados para así no considerarlos a la hora de listar, o buscar en el repositorio. En el caso de los autores, optamos por no permitir el reingreso de un nombre de usuario previamente utilizado por un usuario que haya sido eliminado. En el caso de las entidades, se pueden volver a ingresar al sistema las mismas entidades previamente eliminadas.

```
public void DeleteAuthor(Author anAuthor)
{
    using (FeelingAnalyzerContext ctx = new FeelingAnalyzerContext())
    {
        try
        {
            Author auth = ctx.Authors.Single(a => a.AuthorId == anAuthor.AuthorId);
            auth.Deleted = true;
            ctx.SaveChanges();
        }
        catch (Exception ex)
        {
            throw new ApplicationException("Error al eliminar autor", ex);
        }
    }
}
```

8. Paquete: UI

8.1. Organización del paquete

La estructura de ventanas elegida para el User Interface fue la siguiente:



Se dividió el proyecto en 3 partes principales de forma que sea sencillo identificar cada componente de la UI:

- **Ventana Principal**
  En primer lugar tenemos "MainWindow", la cual contiene el menú principal de la aplicación.
  Se estructura en: un panel lateral con cada una de las funciones que realiza el sistema, un panel de titulo de ventana para identificar la ventana en la que se encuentra y un panel principal el cual invocará a las ventanas secundarias.
  La estructura se muestra en la Figura 5.1.

Figura 8.1: Estructura Ventana Principal

- **Ventanas Secundarias**
  Creamos una carpeta para organizar todas las ventanas secundarias del sistema, cada una de ellas cumple con una funcionalidad requerida de la aplicación, estas se invocarán desde la ventana principal.

  En esta nueva versión se agregaron dos nuevas ventanas secundarias al sistema:

  - AuthorRegistrationWindow
  - AuthorsWindow

- **Controller**
  Se incluye el controlador de la aplicación, como el punto de entrada al sistema. Desde aquí se inicia la aplicación.

9. Integración entre Paquetes

9.1. Interacción UI-Logic-Persistence

9.1.1. Manejo de sub sistemas

Se debió realizar un ajuste de los parámetros de construcción de las ventanas para adaptarlas a la nueva organización general de la aplicación y a los nuevos canales de acceso a métodos y datos, utilizando sub-sistemas en lugar de un solo FeelingAnalyzer individual, como se explicó en la sección 4.1.2.

Para ello, cuando se crea la Ventana Principal "MainWindow" primero se crea con ella una instancia única de Repository que se utiliza para inicializar los distintos subsistemas como se ve a continuación.

```
private void initializeSubSystems()
{
    systemRepo = new Repository();
    subSystemAuthor = new AuthorLogic(systemRepo);
    subSystemEntity = new EntityLogic(systemRepo);
    subSystemFeeling = new FeelingLogic(systemRepo);
    subSystemPhrase = new PhraseLogic(systemRepo);
    subSystemAnalysis = new AnalysisLogic(subSystemFeeling, subSystemEntity, systemRepo,subSystemAuthor);
    subSystemAlarm = new AlarmLogic(subSystemAnalysis, subSystemAuthor, systemRepo);
}
```

Figura 9.1: Inicialización de los subsistemas

Estas instancias únicas de los sub sistemas se pasan entre las ventanas a través de los constructores de cada una de ellas, recibiendo cada una lo único que requiere para ejecutar su funcionalidad. Esto aporta a las buenas prácticas de diseño, favoreciendo la **alta cohesión**.

Por otra parte, al momento de agregar un elemento al sistema capturamos primero todos los atributos ingresados por el usuario para verificar que esta información sea valida para su ingreso al sistema antes de crear el objeto. Esto nos ayuda a detectar fácilmente cual de los atributos es incorrecto al momento en que el usuario ingresa la información.

9.2. Manejo de errores

Para manejar campos de información incorrecta y excepciones, decidimos mostrar estos mensajes en pantalla con el componente “messageBox”.

Cabe remarcar que los errores producidos a través de reglas de negocio se encuentran en el backend y son lanzados mediante excepciones de tipo “ApplicationException” que se muestran a través de los “messageBox” con try/catch. Por otro lado, los errores para campos vacíos, cadenas de caracteres no admitidas, y otras validaciones de formatos en front, se validan directamente en la UI y se muestran también con “messageBox”.

```
private void btnRegisterFeeling_Click(object sender, EventArgs e)
{
    if (AreEmptyFields())
    {
        MessageBox.Show("No pueden haber campos vacíos");
    }
    else
    {
        try
        {
            Feeling f = new Feeling()
            {
                Name = txtName.Text,
                Type = rbtnPositive.Checked ? true : false
            };
            system.AddFeeling(f);
        }
        catch (ApplicationException ex)
        {
            MessageBox.Show(ex.Message);
        }

    }
        EmptyFields();
}
```

INFO INCORRECTA

EXCEPCIONES

Figura 9.2: Ejemplo Manejo de Errores

9.2.1. Errores de Entity Framework

La introducción de EF implica una especial atención al surgimiento de errores no controlados. Para controlar cualquier posible error inesperado en el acceso a la base de datos, ya sea por violación de restricciones o por indisponibilidad técnica de la base, se tomó como norma la utilización de **try-catch** para contener a cada **using**, cuando se utiliza el contexto para realizar alguna operación.

En caso de producirse alguna excepción general, automáticamente lanzamos una excepción personalizada, pero mantenemos la *inner exception* original para poder realizar un rastreo de la causa del error (aunque en esta versión no se produce ningún tipo de reporte de ejecución o log).

```
public List<Author> GetAllAuthors()
{
    try
    {
        using (FeelingAnalyzerContext ctx = new FeelingAnalyzerContext())
        {
            return ctx.Authors.Include("MentionedEntities").ToList();
        }
    }
    catch (Exception ex)
    {
        throw new ApplicationException("Error al obtener autores", ex);
    }

}
```

Figura 9.3: Ejemplo Manejo de Errores

10. Pruebas

10.1. Base de datos secundaria

Utilizamos una base de datos secundarias "FeelingAnalyzerDBTest" para realizar los tests del sistema, el principal motivo por lo que decidimos utilizar otra base para pruebas es para no modificar y sobrecargar la base de datos de la aplicación con datos erróneos y no reales.

Para comprobar que las pruebas funcionan de manera correcta, realizamos una "limpieza" a la base de datos cada vez que se iniciaba ([TestInitialize]) y se finalizaba ([TestCleanup]) una nueva prueba unitaria.

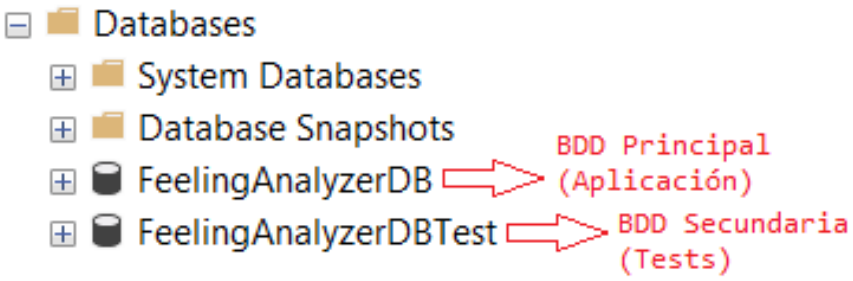


Figura 10.1: Bases de Datos del Sistema

10.2. Cobertura de Pruebas Unitarias

Estas pruebas consisten en aislar una parte del código y comprobar que funciona a la perfección. Son pequeños tests que validan el comportamiento de una funcionalidad particular de una clase y su lógica aislada.
Resultaron de gran ayuda, ya que con ellas pudimos detectar errores que hubieran sido mas difícil de detectar en fases más avanzadas del proyecto.

Se obtuvo un porcentaje de cobertura de un 95,70 % sobre los paquetes domain, logic y persistence.

Jerarquía	No cubiertos (bloques)	No cubiertos (% de bloques)	Cubiertos (bloques)	Cubiertos (% de bloques)
felip_FELIPE-PC 2020-06-24 14_06_17.coverage	131	4,30 %	2918	95,70 %
domain.dll	27	6,22 %	407	93,78 %
Domain	27	6,22 %	407	93,78 %
Alarm	1	1,96 %	50	98,04 %
Analysis	6	6,25 %	90	93,75 %
Author	15	23,08 %	50	76,92 %
AuthorAlarm	1	1,32 %	75	98,68 %
Entity	1	4,35 %	22	95,65 %
Feeling	0	0,00 %	30	100,00 %
GeneralAlarm	2	3,39 %	57	96,61 %
Phrase	1	2,94 %	33	97,06 %
logic.dll	12	2,97 %	392	97,03 %
Logic	12	2,97 %	392	97,03 %
AlarmLogic	0	0,00 %	89	100,00 %
AnalysisLogic	0	0,00 %	35	100,00 %
AuthorLogic	8	5,56 %	136	94,44 %
EntityLogic	4	5,97 %	63	94,03 %
FeelingLogic	0	0,00 %	53	100,00 %
PhraseLogic	0	0,00 %	16	100,00 %
persistence.dll	78	7,78 %	925	92,22 %
Persistence	78	7,78 %	925	92,22 %
FeelingAnalyzerContext	0	0,00 %	95	100,00 %
Listing	8	3,48 %	222	96,52 %
Listing.custTypeAuthorAvgQuery	0	0,00 %	10	100,00 %
Listing.custTypeAuthorAvgRatio	1	4,55 %	21	95,45 %
Listing.custTypeAuthorEntities	1	4,55 %	21	95,45 %
Listing.custTypeAuthorNegRatio	1	4,55 %	21	95,45 %
Listing.custTypeAuthorPosRatio	1	4,55 %	21	95,45 %
Repository	47	9,42 %	452	90,58 %
RepositoryCleaner	19	23,46 %	62	76,54 %

Figura 10.2: Cobertura de Pruebas Unitarias - Visual Studio 2017

Las pruebas unitarias demuestran que la lógica y persistencia de la aplicación se encuentran estable y verificada, y que funcionará en la mayoría de los casos.

11. Repositorio GitHub

Adjuntamos el link al repositorio de GitHub donde se realizó el desarrollo de la aplicación. Este repositorio se encuentra dentro de la organización ORT-DA1.

`https://github.com/ORT-DA1/DA1-M5B-OBL1-228360-232863.git`

12. Conclusiones

12.1. Reflexión Final

Luego de más de tres meses dedicados al proyecto **Análisis de Sentimiento** consideramos que pudimos cumplir con los objetivos de la segunda entrega del obligatorio. Desarrollamos una aplicación de tipo WinForms utilizando el framework de .NET (lenguaje C#), e implementando TDD para la creación de las clases del Dominio. Además, tomamos decisiones de diseño bien fundadas lo que consideramos una evolución en la calidad del desarrollo, con respecto a otros proyectos de programación que habíamos realizado anteriormente.

Creemos que logramos un buen producto final, que cumple con los requerimientos especificados y en la mayor medida posible respeta las decisiones de diseño más importantes comentadas en el curso, como la organización del proyecto, la correcta asignación de las responsabilidades, y la aplicación de varios de los patrones estudiados.

Se intentó arribar a un producto final que permitiera su posterior mantenimiento y adaptabilidad a los cambios. También se consideraron ciertas sugerencias de los docentes para profesionalizar el proceso de trabajo. Por ejemplo, optamos por utilizar **GitFlow** como proceso de SCM para el manejo consistente del repositorio y el control de versiones. Realmente creemos que esta decisión nos ayudó a organizarnos más eficazmente y aunque implicó una curva de aprendizaje inicial mayor, nos introdujo a una gestión más profesional de los cambios y redujo la aparición de conflictos por trabajar en features separadas paralelamente. Siguiendo por la misma línea, tomamos como premisa el desarrollo completo en **idioma Inglés**, incluyendo comentarios y commits. También seguimos sus recomendaciones sobre la organización de la aplicación, creando 5 paquetes con responsabilidades claramente diferenciadas, como se explica en la sección 3.1: Domain, Logic, Test, Persistence, y UI.

12.2. Deuda Técnica

Aunque estamos conformes con el producto logrado, en las últimas semanas de desarrollo identificamos distintos puntos de potencial mejora en la calidad del diseño de nuestra aplicación. De forma consciente y estratégica hemos priorizado y optado por lograr una última versión funcional y aceptable desde todas las distintas perspectivas, así como una documentación sólida, aunque esto implicara dejar pendiente cierta deuda técnica que se detalla a continuación (para tomar en consideración en futuras versiones de la aplicación...):

- **IRepository**. Debido al temprano abordaje de la migración a base de datos, no tuvimos en consideración la importancia de mantener inalterada la versión anterior del repositorio en memoria, para facilitar la transición de un tipo de persistencia a la otra. En lugar de utilizar una interfaz, que hubiera habilitado una transición más fluida y segura, modificamos la clase pre-existente directamente para transicionar a repositorio en base de datos, dejando directamente vinculados los subsistemas a esta clase puntual.
  Se propone implementar una interfaz de Repositorio y mantener solamente una dependencia de los subsistemas hacia la interfaz, como en el diagrama siguiente:

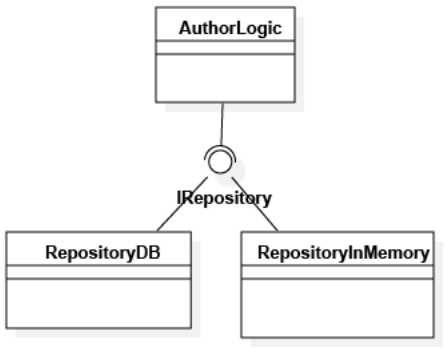


Figura 12.1: IRepository

- **Mejor resolución de los reportes de autores.** La forma en que se resolvieron los reportes visuales de los autores según los distintos criterios fue realizando consultas a la base de datos mediante [5]**LINQ** y **Querys puras**, que creaban estructuras de datos nuevas. Para poder recepcionar esas estructuras y ordenarlas según alguno de los parámetros en UI, utilizamos clases públicas denominadas **custTypeAuthor[CriterioSorting]**. Esto complejizó el paquete Persistence, y no respeta el patrón de Abierto al cambio, ya que es de difícil extensión si se quisiera añadir otro tipo de reporte en un futuro. Se podrían haber buscado alternativas de mejor diseño, como una interfaz **IReportAuthor**, que cada subclase de reporte implementara según sus propios parámetros, además de posiblemente alojarse fuera del paquete Peristence.

Figura 12.2: custom Types

- **Excepciones Particulares.** Otra mejora en el diseño que dejamos pendiente de implementar fue la utilización de excepciones personalizadas para las distintas clases del sistema. En todas las operaciones que validaban reglas de negocio se utilizaron excepciones de tipo "ApplicationException" para la captura de errores. Sin embargo, nos hubiera gustado lograr implementar excepciones dirigidas para cada regla de negocio. También se utilizaron estas excepciones para capturar los errores de Entity Framework, a pesar de que ese tipo de errores no entrarían en dicha categoría.

```
private bool ValidAge(DateTime birthDate)
{
    TimeSpan span = DateTime.Now - birthDate;
    DateTime age = DateTime.MinValue.Add(span); // MinValue is 1/1/1
    int years = age.Year - 1;

    if (years > MIN_AGE && years <= MAX_AGE)
    {
        return true;
    }
    else
    {
        throw new ApplicationException("El usuario debe tener entre 13 y 100 años");
    }

}
```

Figura 12.3: Application Exceptions

Bibliografía

[1] M. Fowler. Anemic Domain Model.

[2] Entity Framework Tutorial. Entity framework.

[3] Tutorialspoint. Fluent API.

[4] Wikipedia. Programacion Orientada a Objetos, conceptos fundamentales.

[5] tuprogramacion.com. ¿Qué es LINQ?

[6] M. Azevedo. S.O.L.I.D principles: what are they and why projects should use them.