Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Diseño de Aplicaciones 1

Obligatorio 2

Klaus Gugelmeier Nro. Est. 247174

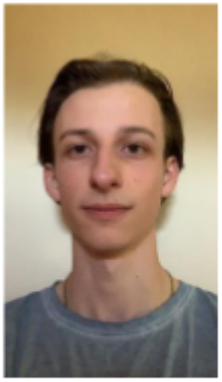
Facundo Sentena Nro. Est. 251661

Grupo M4A

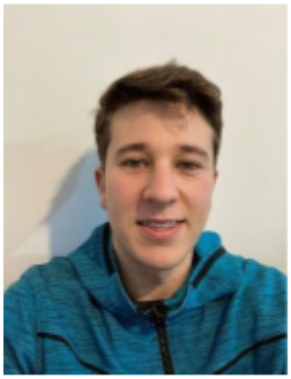
Docente: Mario Souto

Estudiantes

Nro. Estudiante	247174
Nombre:	Klaus
Apellido:	Gugelmeier
Grupo / Turno:	M4A



Nro. Estudiante	251661
Nombre:	Facundo
Apellido:	Sentena
Grupo / Turno:	M4A



Contenido

II. Descripción general del trabajo

A. Descripción

Para este obligatorio se partió de la implementación de un gestor de contraseñas y tarjetas de crédito realizada para la primera entrega del curso.
A dicha aplicación se le pedían una serie de nuevos requisitos y funcionalidades a implementar tales como:

1) La persistencia de todos los datos almacenados por el gestor en una base de datos relacional, a modo de que todos los datos ingresados se persistieran de manera transaccional y automática, sin necesidad de presionar ningún botón de guardado.

2) Registro histórico de "Data Breaches" a diferencia de la implementación anterior, donde se realizaba un chequeo de contraseñas y tarjetas filtradas en un determinado momento y se mostraba al usuario el resultado de el mismo. En esta instancia se pide la posibilidad de registrar, identificar y persistir en la base de datos cada uno de los chequeos realizados a modo de "históricos", permitiendo al usuario luego acceder a una lista de estos, seleccionar uno y poder ver el resultado original de cada Data Breach. Por este motivo se cambió el nombre de las clases referentes a esta funcionalidad a "Filtración" dejando de ser "BuscadorDeFiltraciones" ya que este nombre denotaba que solo se refería a una acción en particular y no a un objeto almacenable.

Como complemento a esto también se pedía la funcionalidad de informar al usuario por cada contraseña del chequeo si esta fue modificada desde su aparición y de no haberlo sido, dar la opción a este de modificarla.

3) Se pidió también la posibilidad de importar Data Breaches, esta vez desde archivos de texto, con un formato preestablecido. Sin deshabilitar la opción de ingresar estos manualmente a partir de una caja de texto.

4) Finalmente se pide que tanto al ingresar, como editar una contraseña se muestre un informe de la misma detallando:

- No aparece en un *data breach* conocido. El sistema deberá chequear si el texto de la contraseña ya apareció en un *data breach* ingresado por el usuario.

-No es una contraseña duplicada. El sistema verifica que el mismo usuario no haya guardado el mismo texto en otra contraseña.

- La contraseña es segura. El sistema verifica que la contraseña pertenece a las categorías Verde Claro o Verde Oscuro, tal como se definieron para el primer obligatorio.

Para esta entrega además se tiene en cuenta que se deben favorecer los patrones GRASP y SOLID vistos en el curso , respetando Clean Code, el uso de TDD y la metodología GitFlow.

Enlace al repositorio de GitHub: https://github.com/ORT-DA1/251661-247174

B. Puntualizaciones

Sobre Data breach: La aplicación permite ingresar Data breaches vacíos, solo se aceptan data breaches provenientes de archivos ".txt" y desde el text field de la interfaz.

A nuestro modo de entender un Data breach se interpreta como una filtración de las contraseñas y tarjetas que el sistema posee al momento del chequeo. Es decir si se ingresa un Data breach con un texto "123456" y luego (posteriormente) se ingresa una contraseña con texto "123456" el sistema no va a determinar que esa contraseña se filtró, por razones de que al momento del chequeo de filtraciones dicha contraseña no existía en el sistema, por lo tanto, por razones lógicas, el Data breach de ese instante no puede tener conocimiento sobre contraseñas que van a ser ingresadas en un futuro, pues no existiría como tal el concepto de "filtración".

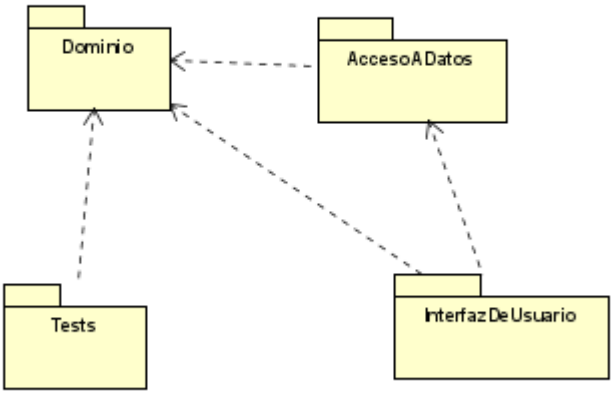
Se presenta un detalle que se encuentra en "editar Tarjeta de crédito", dicho detalle consiste en que, si se desea editar una tarjeta, se deben rellenar todos los campos (aunque se usen datos antiguos de la tarjeta) para que el sistema permite reasignar dichos datos.

En cuanto al versionado y la aplicación de la metodología GitFlow: Cabe destacar que se siguió la metodología, respetando el no realizar commits directos en develop, creando ramas asociadas a features que son posteriormente mergeadas con develop utilizando el - - no – ff de git para no perder todos los commits de la rama mergeada.

Sobre reportes de la contraseña ingresada: Los reportes que indican:
-Si la contraseña ya fue ingresada por el usuario
- Si esta se encuentra en un Data breach conocido
-El nivel de fortaleza de la contraseña ingresada.
Se muestran cuando el usuario agrega una contraseña o la edita desde la pantalla de ver contraseñas, pero no en los submenús de edición presentes en la pantalla de ver fortalezas por contraseña ni en la de los Data breaches debido a que en estas pantallas se busca una edición rápida y sencilla de la contraseña (además de el hecho de que seria redundante informar al usuario por ejemplo que una contraseña cambio de color de fortaleza si este decidió cambiarla desde la pantalla de contraseñas de ese mismo color).

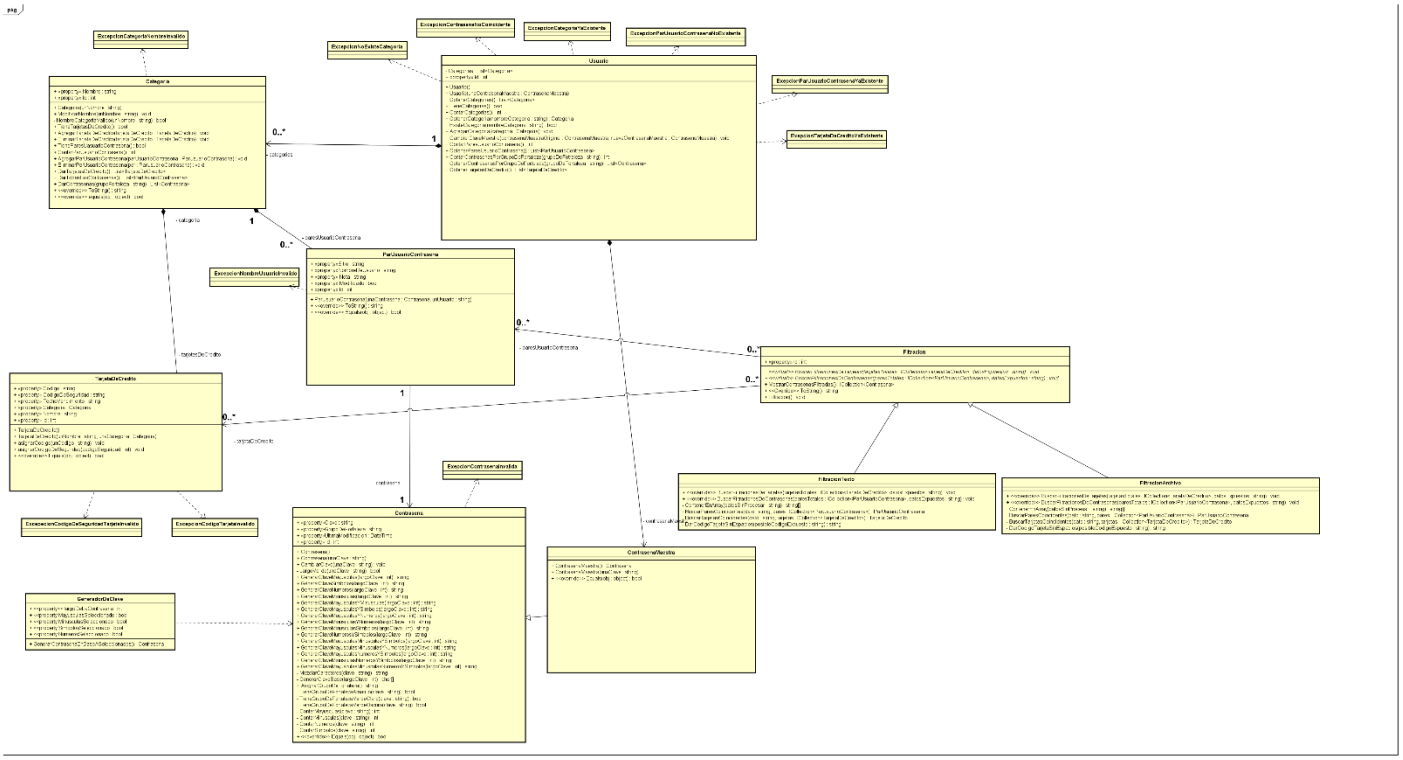
III. Descripción y justificación de diseño

A. Diagramas de paquetes



B. Diagrama de clases

Dominio:

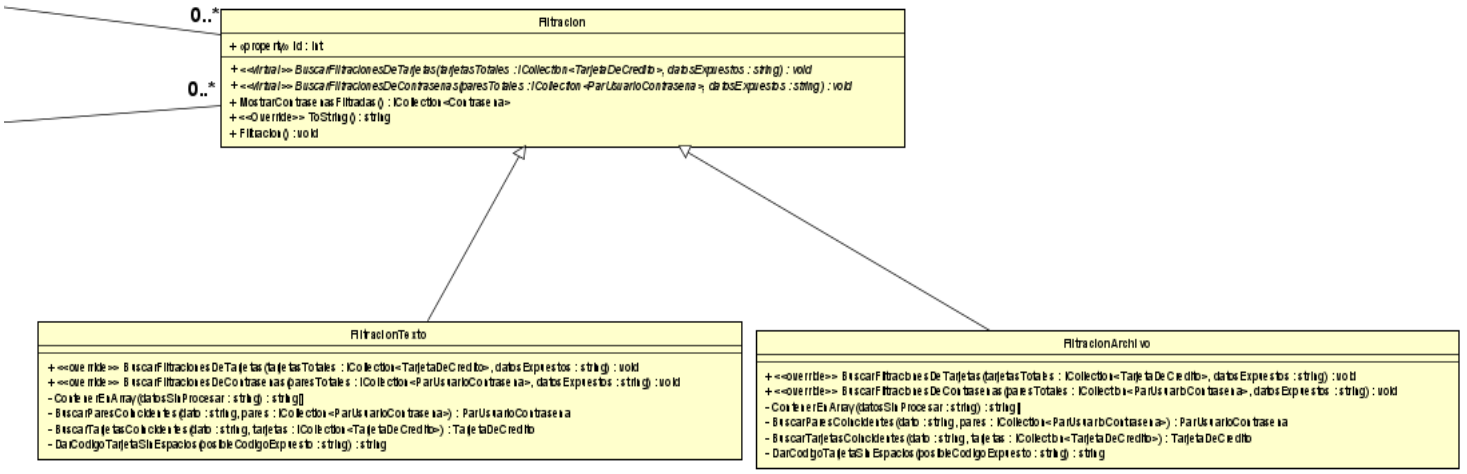


En sí, una de las metas principales durante la realización de esta entrega fue la de tratar de cambiar lo menos que se pudiera el dominio y toda la lógica del negocio, sin embargo, debido a los nuevos requisitos, fueron necesarios algunos cambios tales como:

- El agregado de una property “Id” a cada clase del dominio, a modo de facilitar el mapeo con las entidades de la base de datos.

- La sustitución de la interfaz ”IBuscadorDeFiltraciones” por una clase “Filtración” de la cual heredan “FiltraciónTexto” y “FiltracionArchivo”. La razón por la cual se optó por un polimorfismo basado en herencia y no en implementación de una interfaz fue debido a que en el repositorio de filtraciones (Requerido ahora debido a la necesidad de guardar históricos) se requería instanciar los objetos “filtración” , cosa que no se podía realizar con interfaces y la única alternativa posible a esto era realizar RTTI , aspecto que viola Clean Code.

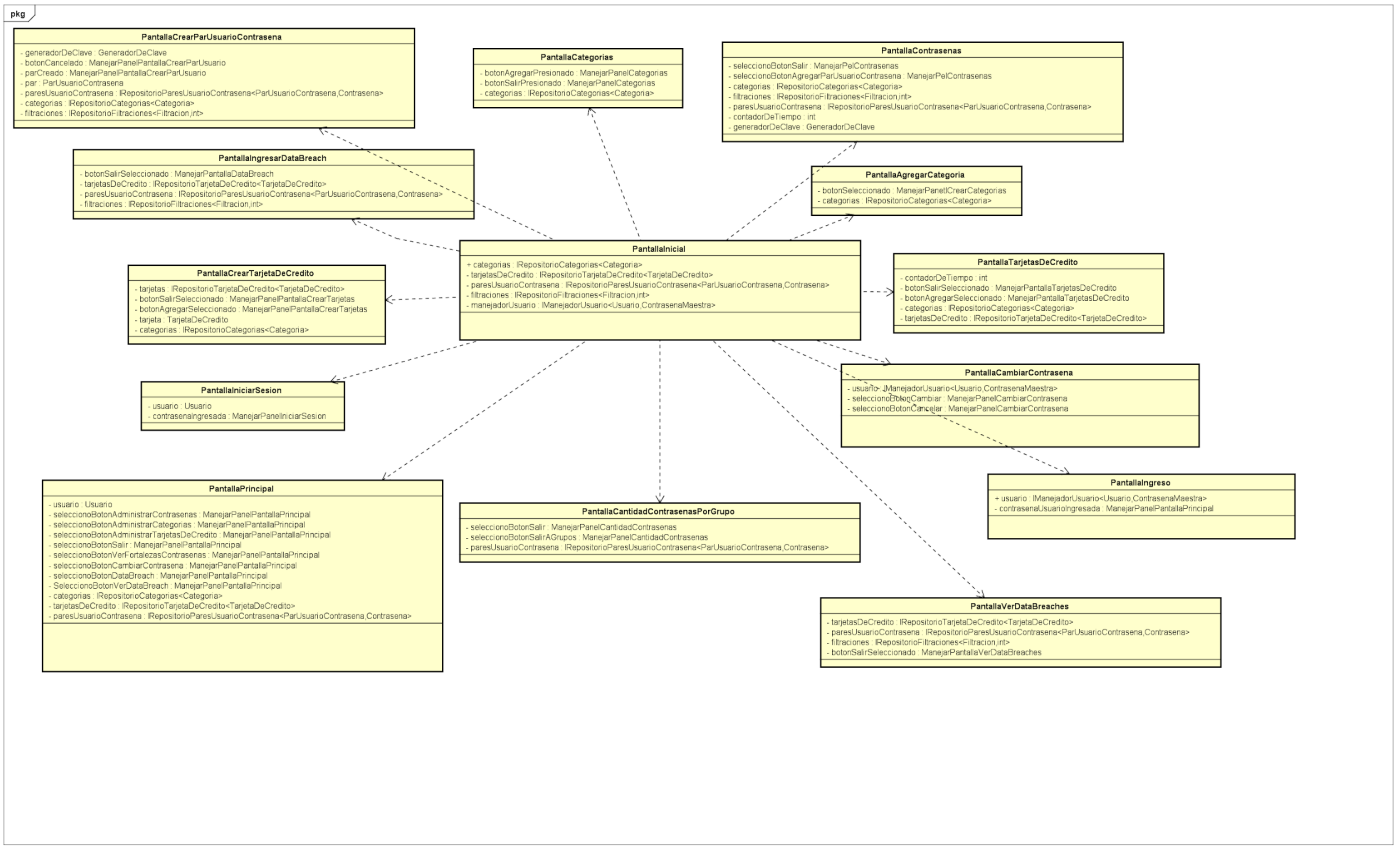
Explicación con UML : Herencia en base a la clase filtración



-En cuanto a las clase “ContrasenaMaestra” decidimos no deshacernos de esta clase, ni de su herencia en base a la clase “Contrasena” debido a que nos facilitó el hecho de mapear dicha clase con su respectiva entidad en la base de datos , utilizándose exclusivamente para el usuario principal de la aplicación.

Si es cierto que a nivel de funcionalidades no implementa ninguna funcionalidad distinta a la de la contraseña “normal” pero no encontramos que agregue ninguna complicación innecesaria al código e incluso si mañana se pidiera que la contraseña del usuario general tuviera que implementar algún nuevo requisito específico, el cambio seria mucho mas sencillo de realizar.

Interfaz:



Como se muestra en el diagrama la clase, al igual que en la entrega pasada la clase PantallaInicial mantiene una dependencia con el resto de las ventanas ya que actúa como manejador(a nivel de UI) de todas estas pantallas instanciándolas y pasandoles los respectivos repositorios por parámetro.

Los repositorios y la interfaz interactúan de la siguiente manera, suponiendo la clase “pantalla genérica” como una de las pantallas, la cual solo accede a los repositorios que necesita. (En este caso se muestra el acceso con asociaciones a varios repositorios, pero en la realidad esto no ocurre.)

Acceso a datos:

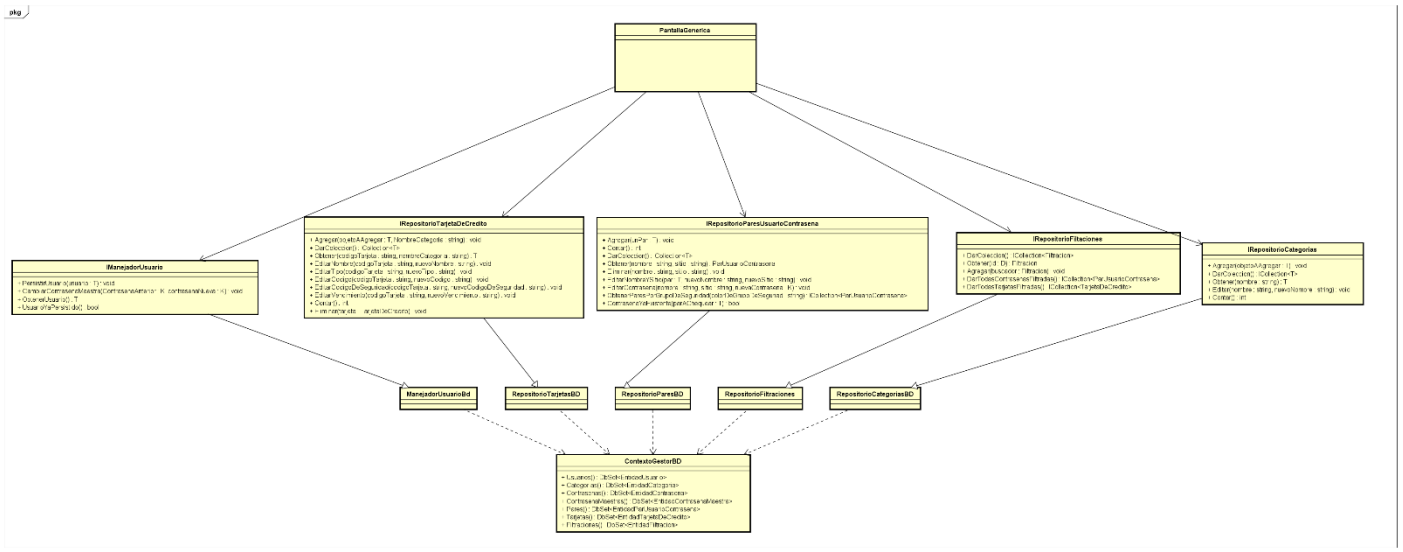
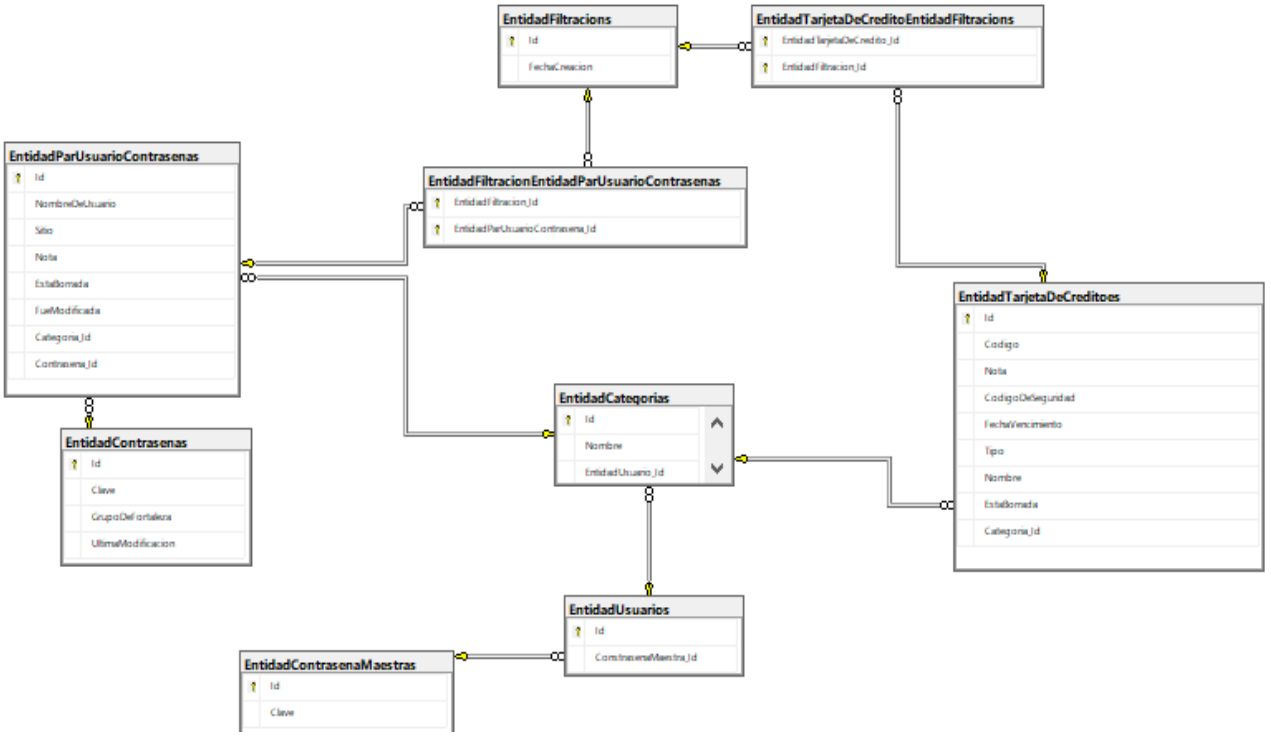


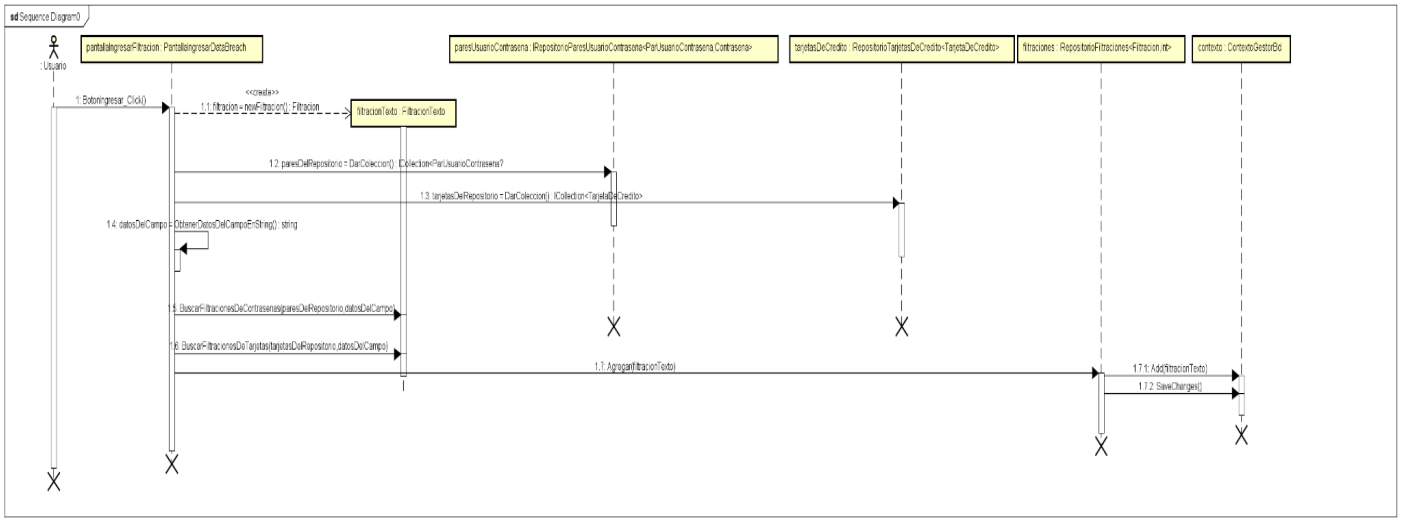
Diagrama de tablas:



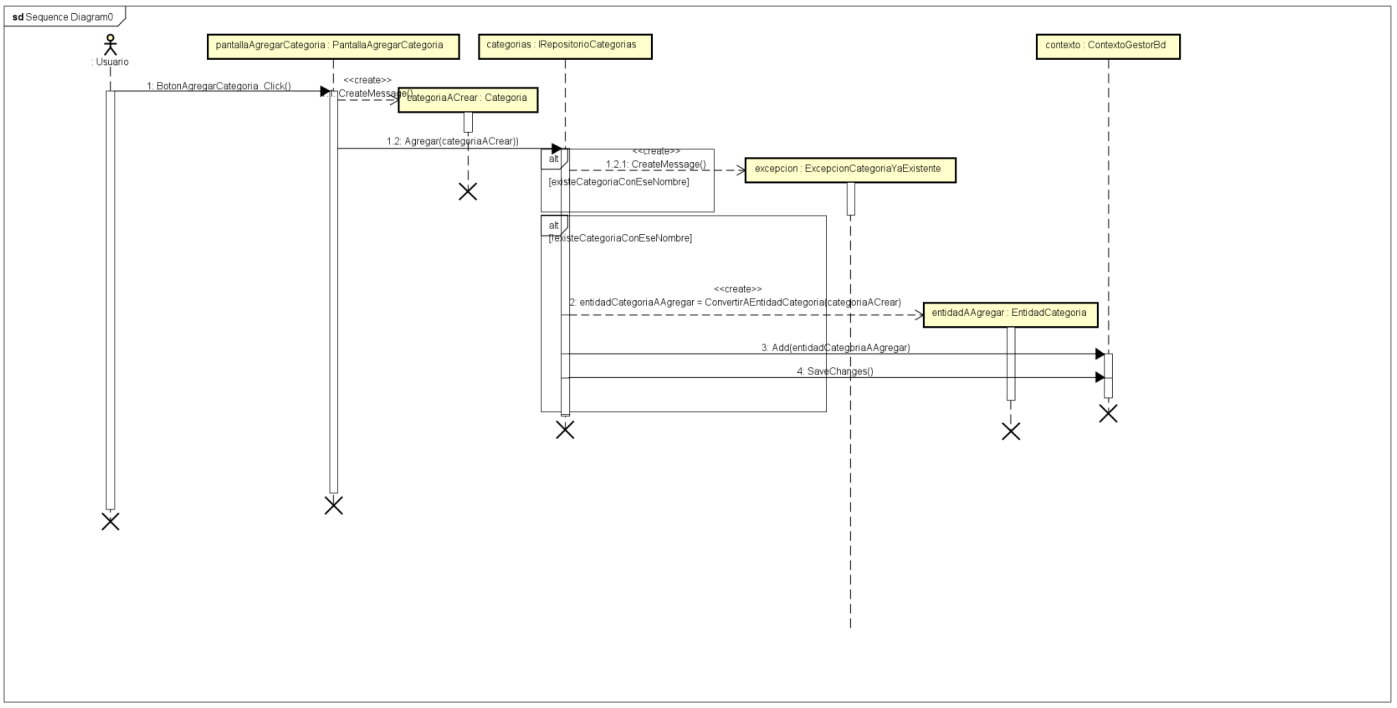
Diagramas de Interacción:

(fotos con mayor definición en anexo)

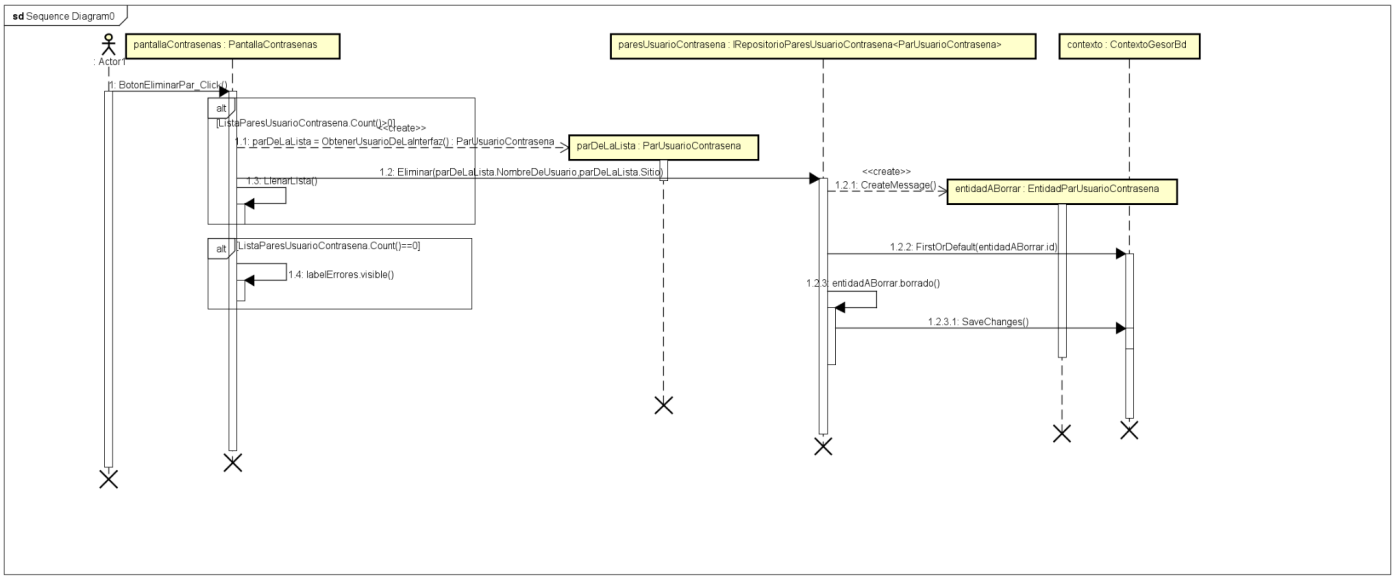
Agregar filtración:



Agregar Categoría:



Eliminar par Usuario-Contraseña:



C. descripción general del sistema

El sistema presenta una pantalla inicial de la cual se ingresa una contraseña maestra para acceder al menú, en este se encuentran distintas opciones:

- Administrar contraseñas (agregar, editar, eliminar y ver).
- Administrar categoría (agregar y modificar).
- Administrar tarjetas de crédito (agregar, editar, eliminar y ver).
- Ingresar data breaches.
- Ver los históricos de data breaches
- Ver fortaleza de contraseñas.
- Cambiar contraseña maestra.
- Un apartado con la cantidad de categorías, contraseñas y tarjetas de crédito.
- Un botón para salir de la aplicación.

D. Explicación de los mecanismos generales

Con respecto al obligatorio anterior, se realizaron varios cambios en los mecanismos generales de interacción entre las distintas clases, esto a modo de favorecer los patrones GRASP y SOLID vistos en el curso.

En primer lugar cabe recalcar que en la entrega anterior nuestro punto de conexión entre el dominio y la UI era la misma clase “Usuario”, clase que de alguna manera actuaba a modo de “fachada” del sistema . Esto nos generaba un nivel de acoplamiento altísimo, ya que se llamaban a los métodos de las clases del dominio desde la propia UI, se

generaban una suma de dependencias bastante grande a las clases del dominio debido a que se instanciaban regularmente en la propia UI, etc.

La manera que encontramos de solucionar este problema, sin perder ningún tipo de funcionalidad y favoreciendo los patrones mencionados anteriormente aún más fue recurrir al “repository pattern”, generando repositorios para las clases del dominio.

De alguna manera nos basamos en los GRASP “Nivel de indirección” y “Variación protegida” debido a que ahora el cliente (en este caso la UI) no interactúa directamente con una clase del dominio sino que lo hace con un repositorio, el cual ofrece un conjunto de operaciones, siendo el quien interactúa con la lógica de negocio, encapsulándola.

Más precisamente en cuanto a “Variación protegida” optamos porque cada repositorio implementara una interfaz IRepository específica al tipo de objeto del dominio que manejaba , ejemplo (IRepositorioTarjetaDeCredito).

Cabe destacar que cada repositorio implementa una interfaz específica en base al tipo de dato que almacena, esto debido a que queríamos hacer que las interfaces fueran lo más específicas y adaptadas al tipo de objeto en concreto con el que se trabaja en ese repositorio. A modo de ejemplo:

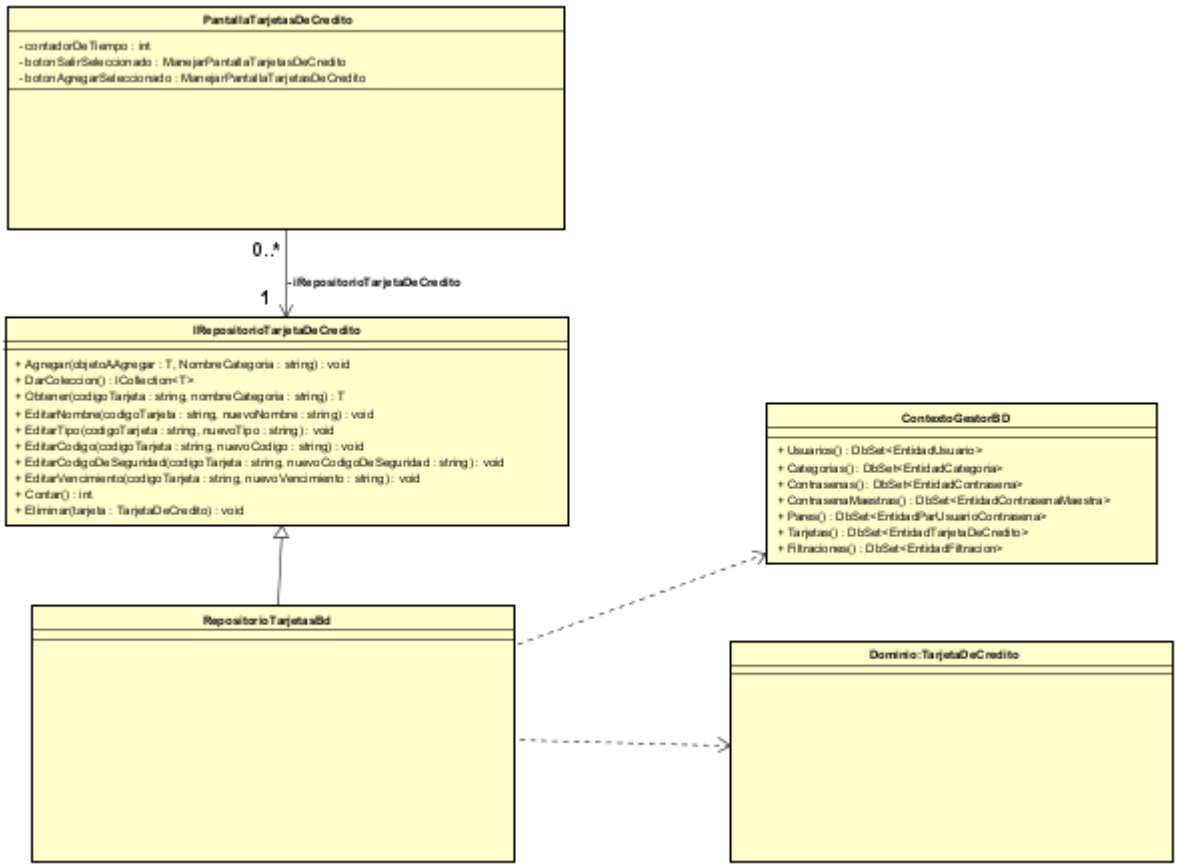
“Si yo implementara una interfaz común de repositorio para TarjetaDeCredito y Categoria . ¿De que me serviría que categoría implementara una operación “borrar”? si en la realidad del problema una categoría no se borra”.

Esto permite implementar de manera mucho más sencilla cambios a nivel de desde donde se obtienen o se guardan los datos del repositorio. Sin ir más lejos, la implementación de estos “IRepositorios” implementados en base a interfaces nos permitieron cambiar rápidamente la implementación anterior de guardar objetos en memoria a persistirlos en una base de datos simplemente creando una clase “RepositorioBd” que implementara dicha interfaz.

De alguna manera también se favoreció el GRASP de bajo acoplamiento ya que se logra desacoplar la interfaz de un tipo concreto como lo era la clase “Usuario” (tipo que además era factible que pudiese cambiar en cualquier momento) a un repositorio que implementa una interfaz donde cualquier cambio no implicaría más que la creación de otro repositorio que implementara la misma.

Aun así, nos consta que siguen quedando elementos de la lógica del negocio dispersos por algunas partes de la UI, por ejemplo, creaciones de instancias, referencias a los tipos, etc.

A continuación se mostrará un ejemplo de la explicación antes mencionada en un diagrama UML:



Posteriormente a implementar esta solución , nos dimos cuenta de que tal vez manejar una clase “Repositorio general” que contuviera todos los repositorios podría haber sido aun más correcto, ya que habría hecho que solo tuviéramos que pasarle a las ventanas un único parámetro en lugar de cada repositorio que necesite, haciendo que en caso de que una pantalla necesitara tener muchos repositorios , esta no violara Clean code teniendo que recibir más de 3 parámetros, por ejemplo.

A nuestra forma de verlo, la clase “Contrasena” continua teniendo una gran cantidad de responsabilidades , las cuales no se encuentran tan vinculadas entre sí, como la generación de la contraseña, la asignación de criterios de fortaleza , etc.

Recurrimos a crear una clase “GeneradorDeClave” cuya función no es otra que la de decidir mediante properties booleanas que método de la clase contraseña utilizar para generar una clave. La principal idea o fundamentación para la creación de esta clase fue desacoplar del dominio las pantallas vinculadas a la creación y edición de las contraseñas, además de quitarle a dichas pantallas la responsabilidad de decidir qué tipo de contraseña generar.

A modo de autocrítica, si bien implementamos “Fabricación pura” respecto la clase mencionada anteriormente(debido a que es un objeto que no existe como tal en la vida real, pero contribuye a favorecer otros GRASP) , podrían haberse creado más clases que contribuyeran a deslindar más responsabilidades de la clase Contrasena , favoreciendo el SRP.

También lo podríamos haber implementado en los repositorios de bases de datos del obligatorio, creando una clase “Mapper” que fuera capaz de convertir objetos a entidades y viceversa. Por falta de tiempo y también por no querer comprometer la estabilidad del obligatorio es que no implementamos esta clase, sin embargo, nos consta que hubiese sido muy útil para favorecer el reuso de Código en los diferentes repositorios así como también aumentar la

cohesión entre los métodos de la misma evitando sobrecargar dichos repositorios con responsabilidades que no tenían tanto que ver con la esencia de la clase.

Cabe destacar, se mantuvieron las excepciones propias creadas en la entrega pasada que eran lanzadas cuando se daban casos no permitidos dentro de la lógica del negocio (valores inválidos, excepciones de no unicidad de un elemento, etc.). Dichas excepciones son “atrapadas” dentro de la lógica de la interfaz de usuario, a modo de evitar que la aplicación lance errores durante la interacción con el usuario. Todas las excepciones implementadas se encuentran dentro del paquete dominio.

En cuanto a los Data breaches: Si bien ya se explicó un poco qué decisión se tomó en cuanto a la implementación de una nueva funcionalidad de importación de Data breaches en el UML de arriba, queríamos profundizar un poco más sobre este tema, y sobre qué tan costoso fue para el equipo realizar este cambio, además de que en la letra se explicita se profundice sobre este tópico.

La realidad es que la implementación de este nuevo formato de ingreso de Data breaches no nos generó demasiadas complicaciones debido a que nuestra clase de ese momento “BuscadorDeFiltraciones” implementaba una interfaz y el hecho de agregar otro formato de ingreso solo suponía crear una nueva clase que implementara dicha interfaz sin modificar código existente.

De esta manera mantuvimos vigente el principio de SOLID de “Abierto y Cerrado” ya que el módulo asociado a la filtración es abierto a la extensión y cerrado al cambio. Es decir, permite añadir nuevo comportamiento sin necesidad de cambiar lo ya existente.

La complicación surgió a posteriori cuando debido al requerimiento de registrar y guardar cada filtración debimos cambiar esta interfaz a una clase como tal, haciendo que las clases concretas heredarán de esta para permitir al repositorio de filtraciones poder instanciar y guardar instancias de la clase sin recurrir a RTTI para preguntar el tipo de filtración recibida, como en el siguiente caso:

```
1 referencia
private EntidadFiltracion ConvertirAEntidadFiltracion(Filtracion filtracion)
{
    EntidadFiltracion entidadARetornar = new EntidadFiltracion();
    ICollection<ParUsuarioContrasena> paresAConvertir = filtracion.ParesDeLaFiltracion;
    ICollection<TarjetaDeCredito> tarjetasAConveritr = filtracion.TarjetasDeLaFiltracion;
```

Sin embargo, a efectos de polimorfismo y de los principios SOLID mencionados anteriormente, no generó consecuencias negativas.

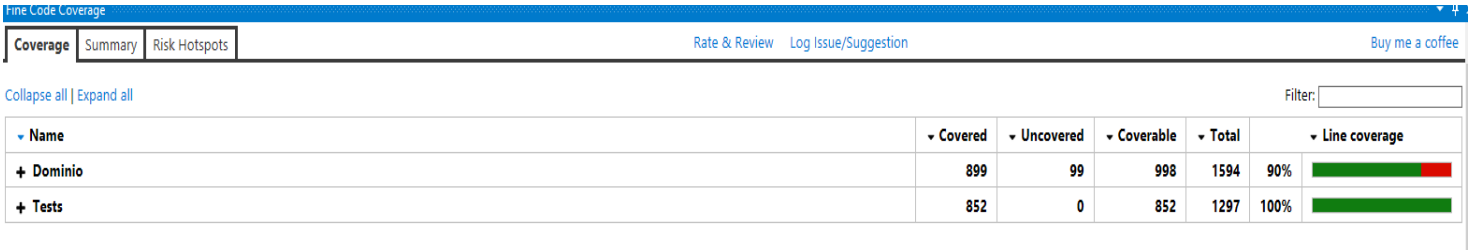
Para mantener la relación entre las contraseñas y tarjetas de crédito bien vinculada a nivel de la base de datos , pero evitando que al borrar una contraseña del sistema esta se borrara del histórico de la filtración, nuestras operaciones de borrado simplemente hacen un borrado lógico a nivel de base de datos, cambiando una property que posee la entidad , la cual indica su estado (borrado o no borrado).

IV. Cobertura de pruebas unitarias

A. Cobertura de líneas de código

Este obligatorio fue hecho desde un principio a través de la metodología TDD (Test Driven Development) enseñado en clase, por este motivo, la cobertura de pruebas unitarias dio un resultado aceptable del 90%.

La razón del porque el resto del código posee un nivel de cobertura más bajo radica en que no se testea por ejemplo la clase “GeneradorDeClave” ya que solo se testearían booleanos y métodos ya testeados de la clase “Contrasena”. En cuanto a la clase “Filtración” la misma no se testea debido a que sus métodos en su mayoría no poseen cuerpo o en caso de tenerlo estos son testeados en sus clases “hijas” puesto que actúa como si fuese una clase “generica” utilizada para favorecer el polimorfismo en el obligatorio.



V. Costo de instalación:

Primero que nada, para lograr correr este sistema adecuadamente, necesitamos tener instalado Sql Server Express dado a que es una parte fundamental del programa. Sin él, el sistema no logrará persistir los datos.
Luego, se debe levantar la base de datos a partir de un backup, que puede ser sin datos o con datos de prueba. En la carpeta Backups en el repositorio se guardarán los scripts con datos y sin datos con sus correspondientes backups para lograr levantar la base de datos con facilidad.

VI. Anexo

Imágenes en resolución completa :
https://drive.google.com/drive/folders/1TDXrxP9xAHuo8x3jnPDUtADRPW3xqUP5?usp=sharing