Obligatorio 2 - Diseño de Aplicaciones I

Autores: Matías Lijtenstein (231070) - Irina Trocki (203608)

Índice

Link a repositorio:

https://github.com/ORT-DA1/203608_231070

Instrucciones y costo de instalación:

En la carpeta raíz de nuestro repositorio se encuentran dos carpetas fundamentales para la instalación de la aplicación: *release* y *Base de Datos.*
Accediendo al archivo *Interface* de la carpeta release se ejecuta la app.

Para establecer la conexión a la base, pusimos algunos strings de conexión a lo largo de nuestro código:
En Interface/App.config y Repository/App.config

```
  <connectionStrings>
    <add name="PasswordManagerDBContext"
connectionString="Server=.\SQLEXPRESS;Database=PasswordManagerDB;Integrated
Security=True" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

Para la ejecución de Unit Testing de la base de datos en RepositoryTest/App.config

```
 <connectionStrings>
    <add name="PasswordManagerDBContext"
connectionString="Server=.\SQLEXPRESS;Database=PasswordManagerDB;Initial Catalog=
PasswordManagerTestDB;Integrated Security=True" providerName="System.Data.SqlClient"
/>
  </connectionStrings>
```

En la carpeta raíz de nuestro obligatorio creamos una carpeta llamada *Base de Datos*. Allí se encuentran tanto el backup de la base de datos vacía como el de los datos de prueba. Para hacer un restore, hay que establecer una conexion entre SQL Server y el archivo .bak.

Clave maestra de ingreso a la app para la base de datos con datos de prueba: **masterKey123**

Descripción general del trabajo y el sistema

En este obligatorio se implementó un Gestor de Contraseñas, con algunas funcionalidades básicas, apto para un solo usuario. La aplicación desarrollada puede almacenar datos de contraseñas junto a sus respectivos nombres de usuario para los diferentes sitios que se deseen, así como también los datos de tarjetas de crédito. A su vez, tanto a las contraseñas como a las tarjetas se les puede asignar una categoría. A la hora de crear una contraseña, el sistema brinda ciertas sugerencias con respecto a la misma. Por ejemplo si es segura, si es única o si ya aparece en algún Data Breach.
El sistema también brinda la posibilidad de chequear Data Breaches. Esto puede ser de dos maneras distintas: a través del ingreso manual de un texto o a través de un archivo txt seleccionado del filesystem. Por otro lado, se permite ver un histórico de Data Breaches, donde se muestra la fecha en el cual fue realizado cada uno, junto a sus resultados correspondientes. Además se puede ver un reporte de fortaleza de contraseñas según cierto criterio establecido.
En nuestra implementación, a nivel de interfaz gráfica, una vez que se accede al sistema ingresando la contraseña maestra correcta, la aplicación consta de una ventana principal donde se muestran todas las funcionalidades posibles.
Cabe destacar que el sistema es persistente. Es decir que al cerrar el programa, los datos ingresados anteriormente quedan guardados en una base de datos, y no se pierden.

En cuanto a bugs y complicaciones que hemos tenido, un gran punto en el que enfrentamos una dificultad a lo largo del desarrollo de este proyecto fue el scaling y auto-size de las diferentes ventanas. Durante muchos días nos vimos obligados a probar con diferentes recursos y buscar información para resolver un problema que surgió al momento de trabajar con la interfaz. El problema era que los UserControl no se adaptaban adecuadamente al tamaño del panel, ni siquiera si les asignabamos tamaños idénticos. Esto lo resolvimos cambiando la propiedad de *AutoScaleMode* de todos los UserControl a *Inherit* y manteniendo esta misma propiedad en la ventana principal en *Font*.

Descripcion y justificacion de diseño

Explicación de los mecanismos generales y descripción de las principales decisiones de diseño tomadas.

Uso de Git

Desde el comienzo del obligatorio intentamos utilizar git flow de la mejor manera y establecer criterios en común para el uso de la herramienta.
A lo largo del trabajo tomamos la decisión de abrir ramas desde la rama develop, para cada una de las clases necesarias y siempre aclarando el propósito que tendría la rama. Optamos por adoptar un criterio común para el nombramiento de las mismas. Ej: feature_Account, feature_User, entre otras.

En cuanto al trabajo en cada rama, intentamos no hacer un solo commit por rama, sino que hacer varios commits con su mensaje correspondiente y cuando terminamos de trabajar en cada rama, hicimos un pull request con un merge a develop.

Entendemos que pudo haber sido más ordenado crear ramas por funcionalidad y no por clase, ya que tal vez hacíamos commits con demasiadas modificaciones. Sin embargo, fue la forma en la que nos sentimos cómodos trabajando.

Luego, a medida que íbamos encontrando bugs o errores, en vez de volver a abrir una rama ya mergeada con develop, creamos ramas nuevas, con el objetivo de solucionar esos errores. Asimismo, fuimos creando ramas para algunos métodos que también vimos que faltaban en ciertas clases. Entendiendo que git flow no recomienda volver a trabajar en ramas que ya habían sido previamente cerradas y mergeadas en otra rama.

De todos modos decidimos no eliminar las ramas que fueron creadas, sino dejarlas en el repositorio.

Por último, cuando ya teníamos el trabajo terminado y no debimos hacer más cambios, hicimos merge de la rama develop con la rama master. De modo que la rama master contenga solo la versión final del trabajo con el tag indicado en la rúbrica.

Diagramas de interacción

Para la elección de los diagramas, decidimos optar por tres casos de uso que reflejaran aspectos muy diversos de la app. Si bien en un principio nos costó pensar en toda la comunicación entre las diversas clases, nos adaptamos rápidamente y a nuestro parecer, supimos reflejar el funcionamiento general de dichos casos de uso.
Los casos de uso elegidos para los diagramas de secuencia fueron:

- Agregar una cuenta - Imagen 8 del Anexo
- Chequear un data breach a partir de texto plano - Imagen 9 del Anexo
- Modificar la master key - Imagen 10 del Anexo

Cabe destacar que en lo que respecta a los objetos de las clases de la interfaz de usuario, y toda la interacción entre los diferentes paneles y user controls lo agrupamos y generalizamos bajo el nombre de "Interfaz", ya que nuestro objetivo fue enfocarnos en el dominio y su interacción en general con el resto de la aplicación.

Paquetes

Ver diagrama de paquetes en Anexo.

Para el correcto desarrollo de nuestro Gestor, optamos por dividir el diseño en cinco grandes namespaces, intentando lograr más cohesión al momento de trabajar, y logrando separar la información y el acceso a la misma.

Nuestra división fue la siguiente:

- **PasswordManager**
    - Implementamos todas las funcionalidades a nivel de dominio (creación de clases, manejo de listas de objetos, implementación de métodos, etc)
    - Dentro de este namespace, creamos varias carpetas con el objetivo de organizar mejor las clases y poder agrupar aquellas que fueron creadas con finalidades en común:
        - *Exceptions*: se encuentran las implementaciones de todas las excepciones utilizadas.
        - *FDataBreaches*: contiene todas las clases relacionadas a la implementación de la lógica de la funcionalidad del chequeo de data breaches. Al separar las responsabilidades en más clases, decidimos que esta sería la mejor manera de mantenerlas agrupadas.
        - *FPasswordGenerator*: Al igual que el caso anterior, contiene todas las clases respectivas a la funcionalidad de la generación de contraseña. Más adelante se explicará en detalle que para esta segunda entrega se tomó la decisión de implementar la generación de contraseñas de una manera completamente diferente, que resultó en varias clases que se relacionan entre sí.
- **PasswordManagerTests**
    - Implementamos el Unit Testing realizado con MSTest para poder trabajar con TDD.
- **Interface**
    - Implementamos toda la interfaz de usuario utilizando Windows Forms.
- **Repository**
    - Paquete utilizado para la conexión con la base de datos utilizando Entity Framework (code first). En este proyecto implementamos además las clases Data Access para acceder a cada uno de los datos de la base.
- **RepositoryTests**
    - Unit testing de la base de datos con conexión a una base de datos de prueba.

PasswordManagerTests (pruebas unitarias)

Para el completo desarrollo del obligatorio, trabajamos utilizando TDD, por lo que el desarrollo de las pruebas fue de vital importancia. Si bien en un principio nos costó entender cómo lograr generar las pruebas adecuadas, en poco tiempo pudimos adaptarnos al mecanismo y acostumbrarnos al desarrollo guiado por las mismas. Para cada clase que nos parecía pertinente tener en nuestro dominio, comenzamos creando una clase de pruebas.
Desarrollamos pruebas en un orden determinado, probando desde las cosas más macro y generales, como la instanciación de una clase o la validación de las properties de la misma, llegando a probar hasta los detalles más mínimos, logrando de esta manera completar el desarrollo completo de la clase a partir de todos los detalles probados.
Cabe mencionar que en el resultado final de nuestro obligatorio, no tenemos una clase de prueba para la totalidad de las clases del dominio, ya que varias de estas clases surgieron en la

etapa de refactor, en un intento de extraer características comunes o separar responsabilidades, pero manteniendo las pruebas unificadas en un solo lugar.
En algunas clases de pruebas, como ser el caso de la clase DataChecker o DataBreachCheck (que se mencionarán más adelante) decidimos tener datos inicializados en un método TestInitialize, ya que el chequeo de sus métodos debe darse a partir de una comparación con datos existentes. Sin embargo, en el caso de las pruebas de dominio, en la mayor parte de las clases de prueba se realizaron inicializando los datos necesarios para cada prueba en particular.

PasswordManager (dominio)

Ver diagrama de clases de PasswordManager en Anexo.

En una primera instancia, al enfrentarnos con la propuesta de este trabajo, lo primero que se nos vino a la cabeza, naturalmente, fue la necesidad de:

- Una clase para los pares usuario - contraseña (Account)
- Una clase para las tarjetas de credito (CreditCard)
- Una clase para las categorías (Category)
- Una clase para el usuario de la aplicación. (User)

De todas formas, sabíamos que si bien estas clases eran necesarias, no serían suficientes. Sin duda, también deberíamos buscar alguna manera de implementar funcionalidades como la de los Data Breaches, la fortaleza de contraseñas, las sugerencias para contraseñas, así como también, intentar llevar el diseño de nuestra aplicación a uno más extensible y reusable para el futuro.

Category

La clase **Category** representa cada categoría, con su respectivo nombre como atributo. Además, en ella se valida el nombre de ingresado para las categorías, haciendo uso de los métodos de la clase Validator.
En lo que respecta a la validación de categorías, decidimos que no se puede agregar categorías con el mismo nombre (case insensitive).

Account

La clase **Account** la modelamos para representar el par usuario-contraseña. En ella almacenamos, utilizando properties, los datos respectivos a las contraseñas almacenadas en el sistema, es decir:

- Nombre de usuario
- Contraseña
- Sitio
- Notas adicionales
- Última fecha de modificación
- Categoría

Además, a cada **Account** agregamos un atributo de clasificación por color. Como bien sabemos, uno de los requerimientos funcionales del gestor consistía en brindar un reporte de fortaleza de las contraseñas guardadas en el sistema. Desde un principio en nuestro grupo de trabajo estuvimos de acuerdo que sería conveniente mantener registro de la clasificación por color de cada contraseña. Notemos que, si no mantuviéramos un registro de esto, nos veríamos obligados a clasificar cada una de las contraseñas del sistema según su fortaleza para cada vez que el usuario desee obtener un reporte. Nosotros, por lo contrario, para evitar esta ineficiencia, decidimos que cada vez que se agregue o se modifique una contraseña, lo ideal sería que nuestro código guarde la fortaleza de esta clave, por si eventualmente se pide un reporte. Es por esto que la clase Account también cuenta con un atributo de *clasificación por color*. Dicha clasificación es de tipo ColorClassification, que será explicada más adelante.
Por su parte, para lograr que las contraseñas se clasifiquen en el rango de colores mencionados que se encargan de asignar un ColorClassification al atributo correspondiente de dicha clase cada vez que se agrega o modifica una contraseña. Dichos métodos se encuentran en la clase ColorClassificator, de la cual hablaremos más adelante.

Cabe destacar también que la última *fecha de modificación* no es un dato del account que lo ingrese el usuario del Gestor, sino que lo asigna automáticamente nuestro sistema cada vez que se modifica o crea una instancia de Account. Para hacerlo, utilizamos la librería externa DateTime.

Además, en muchas de las properties de esta clase se introdujeron validaciones para validar los atributos, como ser el largo de la contraseña, el largo del usuario, entre otros. Para ello, se utilizaron los métodos de la clase estática Validator.

Por último, la clase tiene dos override de métodos heredados de Object: ToString y Equals.
El override de *ToString* lo hicimos para que retorne un string con los datos de las cuentas como se deben mostrar en la ventana de resultados de Data Breaches:

"[Nombre de categoría] [Sitio] [Nombre de usuario]"

El override de *Equals* lo realizamos para introducir el criterio de comparación de dos cuentas, es decir, que si dos cuentas tienen mismo sitio y nombre de usuario, corresponden a la misma. Este método fue utilizado luego para imposibilitar la creación de más de una contraseña para cuentas de igual sitio y usuario.

CreditCard

La clase **CreditCard** la modelamos para representar una tarjeta de crédito. En ella se almacenan los datos que describen a cada tarjeta, que se desean guardar en el sistema:

- Nombre
- Compañía
- Número
- Código de seguridad
- Fecha de expiración
- Categoría

Esta clase también tiene incluye validaciones en las properties donde se llama métodos de Validator. Además, se redefinen en ella los métodos ToString y Equals, con los mismos propósitos que se hizo en Account. En este caso, el criterio de comparación para el override de Equals es el número de la tarjeta. El método ToString, por su parte, retornará un string de la forma: "[Nombre de la tarjeta] [Compañía] [Número]".

En cuanto a los métodos de validación decidimos que el año de la fecha de vencimiento debe estar en el rango de 1000 a 9999, o sea, tener 4 dígitos. El mes de vencimiento debe estar entre 1 y 12.

Por otro lado, es importante destacar que la fecha de expiración la separamos en mes y año por separado, como dos enteros. Entendemos que quizás a largo plazo esto pueda no resultar ser tan buena práctica, pero a los efectos de nuestro obligatorio y teniendo en cuenta que en la realidad las tarjetas de crédito contienen únicamente mes y año, nos pareció la mejor manera de implementarlo.

DataUnit

Dado que en la teoría, los datos de contraseñas y tarjetas de crédito son simplemente unidades de datos a guardar, que tienen en común una categoría que los clasifica y permite agruparlos, decidimos que las clases de *Account* y *CreditCard* podrían heredar de una clase común (en este caso *DataUnit*) que tenga una categoría. De esta forma, si eventualmente se quisiera tomar todos los datos guardados en el sistema, sin importar a qué tipo de datos corresponden, podría utilizarse el polimorfismo y tratarlos a todos como instancias de la clase *DataUnit*.

En conclusión, en lugar de asignar un atributo de categoría a las clases de CreditCard y Account, las mismas heredan de la clase DataUnit. Si bien esto quizás no tiene mucha utilidad en este obligatorio, lo hicimos porque podría resultar útil en un futuro, para poder tratar a las tarjetas de crédito y los pares usuario-contraseña como un mismo tipo de objeto.

User

Uno de los grandes cambios para esta segunda entrega fue la utilización de la clase User. Comprendimos que en una primer instancia le atribuimos gran parte de las funcionalidades del sistema a esta clase y también, era esta la que almacenaba las listas de datos. Para esta segunda instancia, si bien mantuvimos la clase User, le quitamos mucha responsabilidad.
La intención de la clase User es representar al Usuario de la app, que en nuestro caso, será uno solo. La misma almacena:

- La clave maestra de ingreso a la aplicación
- Un bool indicando el estado del usuario frente a la app
    - Optamos por utilizar un bool *Status* que lleva registro de si el usuario está ingresado en el sistema o no, es decir, si puso la clave maestra correcta y tiene acceso a los datos o no.

La clase User tiene dos métodos de sign in y sign out que ingresan y cierran sesión, cambiando el bool guardado en Status, e inhabilitando al usuario el acceso a los datos. Además, también existe un método de modificación de la contraseña maestra.

DataChecker

La responsabilidad de esta clase es chequear la información previo a agregarla a la base de datos. Es decir que tiene métodos que se encargan de comparar la entidad que se quiere agregar con las entidades que ya están dentro de la base de datos.
Por ejemplo. Si se quiere agregar una contraseña, el método UniqueAccountCheck revisa si ya existe alguna contraseña para dicho sitio y usuario.
Lo mismo si se quiere agregar una categoría. El método UniqueCategoryCheck debe verificar si ya existe la categoría que se quiere agregar.
Lo mismo con las tarjetas. El método UniqueCreditCardCheck verifica que no exista una tarjeta con el mismo número.

Modificator

Esta clase contiene todos los métodos que intentan modificar las categorías, las contraseñas y las tarjetas, chequeando que los datos ingresados sean válidos y que no existan elementos iguales ya registrados en el sistema. El método TryModifyCategory llama a los métodos de DataChecker que validan la información de las categorías.

En este sentido, también contiene los métodos TryModifyAccount y TryModifyCreditCard, que de la misma manera llaman a sus método correspondientes en DataChecker.

Enum ColorClassification

Creamos una estructura de **Enum** llamado ***ColorClassification*** para los 5 colores posibles de clasificación, facilitando así la asignación de valores al atributo de clasificación por color de Account.

ColorClassificator

Al observar la implementación de la funcionalidad del **reporte de fortalezas** de la primera entrega, nos dimos cuenta que los métodos respectivos a la misma estaban muy dispersos por el código, en clases que ya de por sí tenían mucha responsabilidad. En un intento de unificar estos métodos en un solo lugar y separar responsabilidad, creamos una clase estática ColorClassificator que los reúne.

El método FilterBy, que antes teníamos definido en la clase User, se encarga de devolver una lista con todas las contraseñas pertenecientes a un color de fortaleza, que será pasado por parámetro.
El método ClassifyColor recibe por parámetro un string, que será la contraseña ingresada y dependiendo su fortaleza devolverá el color correspondiente, que será alguno de los definidos en el enum ColorClassification.
El método PasswordStrengthCount recibe por parámetro la lista de accounts y devolverá un array de enteros. Este array tendrá tantas posiciones como colores diferentes hay en el

enum(en este caso 5). Y cada posición tendrá como valor la cantidad de contraseñas que tengan dicho nivel de fortaleza.
Cada vez que el usuario desee ver el reporte de fortalezas de contraseñas, se invoca a este método.

Validator

La clase Validator es una clase estática que tiene como responsabilidad única validar. Esta clase es estática, ya que entendimos que no iba a ser necesario crear instancias de la misma, y porque desde varios lugares del código precisamos acceder a sus métodos de forma sensilla.
Esta clase tiene distintos métodos que se encargan de que los datos ingresados por el usuario sean válidos. En la primera entrega, teníamos un método distinto para cada uno de los atributos que teníamos que validar.
Intentamos generalizar ciertos métodos que eran muy similares, contemplando sus diferencias con los parámetros.
Por ejemplo, implementamos un método ValidarString que recibe por parámetro el string a validar, y un par de enteros que contendrá el número mínimo y máximo de caracteres que puede contener, que sirve para sustituir todos aquellos métodos de validación de largos de strings del primer obligatorio.
Además, esta clase contiene algunos métodos más especificos, como ser ValidateNumber, el cual se encarga que el número de la tarjeta ingresado sea un número de 16 dígitos separado por espacios.
También contiene métodos encargados de validar la fecha de vencimiento de las tarjetas, y el código de seguridad de las mismas.

Clase PasswordAnalysis

En la segunda entrega una de las funcionalidades que se pedía desarrollar era la de sugerencia de mejoras de contraseñas. Para ello, creamos una clase *PasswordAnalysis* que, tal como lo indica su nombre, se encarga de hacer todo el análisis solicitado para la contraseña. Para los chequeos requeridos, precisamos contar con las listas de data breaches y accounts almacenadas en el sistema, por lo que dicha clase cuenta con un constructor por parámetro que las recibe y las setea como atributos privados de la misma.

Además, esta clase tiene tres atributos booleanos públicos: *DataBreach, Duplicated, Secure.* Los mismos varían para cada contraseña analizada, registrando si la misma aparece en data breaches, es una contraseña ya existente y es segura.

PasswordAnalysis tiene un solo método público llamado *RunAnalysis* que recibe por parámetro un string y cambia los flags mencionados anteriormente.
Decidimos que RunAnalysis sea un método independiente y no sea invocado al instanciar la clase porque de esta manera, siempre y cuando las listas del sistema no varíen, se puede utilizar la misma inicialización de un objeto de PasswordAnalysis para realizar el análisis de más de una contraseña.
Luego de llamar al método RunAnalysis, para acceder a los resultados es necesario obtener el valor final de los atributos *DataBreach, Duplicated* y *Secure.*

Generación de contraseñas - FPasswordGenerator

Ver diagrama de clases en Anexo.
Uno de los aspectos que nos habían quedado pendientes en la primera entrega del producto tenía que ver con el método de generación de contraseñas. Si bien el mismo funcionaba a la perfección, era muy largo, tenía muchas responsabilidades y no cumplía con ninguno de los lineamientos de clean code. Es por eso que para esta segunda entrega decidimos mejorarlo, intentando aplicar patrones y lograr que el código no solo quede más limpio, sino también más fácil de cambiar en casos futuros.

Clase PasswordRequirement

Para sustituir los flags booleanos que se pasaban por parámetro indicando los requerimientos para la generación de la contraseña, introducimos una clase abstracta llamada *PasswordRequirement.* Decidimos que dicha clase fuera abstracta ya que queríamos poder generar un contrato común para nuestros requerimientos para lograr aplicar polimorfismo y tratarlos como objetos de un mismo tipo, y a su vez enseguida descartamos la opción de hacer una interfaz porque nos interesaba poder implementar ciertos métodos que eran comunes para todos los requerimientos.
Esta clase almacena dos atributos: una lista de enteros (*AsciiNumbers*) y una lista de pares de enteros (*AsciiRanges*).
Para entender un poco mejor estas dos listas, recordemos que cada uno de los requerimientos para la contraseña (mayúsculas, minúsculas, dígitos y/o caracteres especiales) se corresponden con uno o varios rango/s de números de la tabla ASCII. Por lo tanto, decidimos almacenar en *AsciiNumbers* la lista de todos los números correspondientes al requerimiento en cuestión y en *AsciiRanges* los límites de rangos de números.
Al heredar de *PasswordRequirement*, es necesario implementar un método llamado *AsciiRangeInitialization*, ya que el mismo es abstracto en la clase padre. La idea de dicho método es setear la lista de rangos de la tabla Ascii para el requerimiento creado. De esta forma, logramos que si en un futuro se desea agregar otro tipo de requerimiento que contemple otro conjunto de rangos numéricos de la tabla, esto pueda implementarse sin necesidad de realizar muchos cambios en el sistema.
La lista *AsciiNumbers* por su parte, se carga de forma automática a partir de los rangos inicializados para cada requerimiento. Para ello se implementó un método llamado *ChargeAsciiList* que es llamado al crear instancias del requerimiento.

Como dijimos anteriormente, hay ciertos métodos que nos interesaba que todos los requerimientos pudieran tener, y resultaba innecesario implementarlos múltiples veces, por lo que se encuentran en PasswordRequirement. Estos son:

- *GenerateCharacter* → Genera un caracter random que cumpla con el requerimiento que lo pide.
- *ContainsRequirement* → Recibe un string por parámetro y chequea si dicho string contiene al menos un caracter que cumpla con el requerimiento al que respecta.

Clase *PasswordGenerator*

Esta clase es efectivamente la encargada de la generación de la contraseña. La misma recibe una lista de PasswordRequirement con las instancias de las clases de los requerimientos que heredan de PasswordRequirement y un entero correspondiente al largo de la contraseña, según lo indique el usuario. Además, se guardan como atributos un string que se va modificando, que corresponde a la contraseña generada, y una lista de enteros llamada *AsciiNumbers* que contiene todos los números de los rangos numéricos de la tabla ascii de todos los requerimientos seleccionados para la contraseña.

La manera de generar una contraseña es inicializar una instancia de esta clase, pasandole la lista de requerimientos y el largo por parámetro, y luego, acceder al atributo Password para obtener la contraseña generada.

El funcionamiento interno de la clase consta de:

- Un método de validación de los requerimientos solicitados.
- Un método de generación de contraseña en el cual se llama a dos métodos extraídos:
    - AddMandatoryRequirementsToPassword → agrega en primer lugar el minimo necesario de caracteres para asegurar que la contraseña generada cumpla con los requerimientos solicitados
    - CompletePassword → completa el string a partir que fue agregado el mínimo previamente mencionado, hasta alcanzar el largo establecido, tomando caracteres random que cumplan con cualquiera de los requerimientos posibles.

Clase *AsciiRangeRandomNumber*

Creamos una clase AsciiRangeRandomNumber que tiene como responsabilidad seleccionar un número random dentro de una lista de números que se corresponden a la tabla Ascii.

Decidimos aislar esta funcionalidad e implementarla en una clase aparte porque, no solo queríamos separar la responsabilidad para mejorar nuestro código, sino que también la precisamos utilizar en más de un lugar.
De esta manera es que intentamos aplicar otro de los principios de GRASP llamado alta cohesión. Donde intentamos asignar responsabilidades en clases pequeñas, con el objetivo de mantener la cohesión alta. Si antes teníamos la generación de contraseña solo en un método, ahora el código es mucho más eficiente y entendible gracias a los principios de GRASP aplicados.

Esta clase es utilizada en el método *GenerateCharacter* de la clase PasswordRequirements. El mismo le pasa la lista de AsciiNumbers y recibe un número random de todos ellos. Recordemos que todos los requerimientos cuentan con un rango diferente de números de la tabla ascii, y hasta en casos como los caracteres especiales los rangos numéricos no son de corrido, por lo que esta clase nos facilitó mucho la implementación.

Por otro lado, también acudimos a AsciiRangeRandomNumber desde la clase PasswordGenerator para completar la contraseña con caracteres random, una vez que nos aseguramos que la misma cumple mínimamente con los requerimientos solicitados.

Data Breaches

Ver ampliación de diagrama de clases de carpeta FDataBreaches en Anexo.

Resta hablar sobre cómo llevamos a cabo la funcionalidad de chequeo de Data Breaches. Para la implementación de la misma, optamos en una primera instancia por abstraernos del formato en que nos fueran entregados los datos en este caso, y elegir un formato genérico a chequear. Decidimos que el chequeo de data breaches se daría a partir de una lista de strings, los cuales se compararían uno a uno con los datos guardados en el sistema. Creamos entonces una clase genérica ***DataBreachCheck*** que guarda una lista de Account's y una lista de CreditCard's, correspondientes a aquellos datos que fueron expuestos.

Sin embargo, en nuestra app, notamos que la información a chequear para los data breaches se puede ingresar en diferentes formatos. Para ello aplicamos el polimorfismo, uno de los principios de GRASP que aprendimos en las clases teóricas. Ya que este principio nos permite dejar nuestro sistema abierto a implementaciones nuevas que aún no existan. Por lo tanto, precisaríamos una manera de transformar la información y adaptarla a una lista de strings, que sea apta para nuestro chequeo.
Afortunadamente, desde la primer entrega, creamos una interfaz ***IDataBreachesAdapter*** que establezca un contrato común para poder adaptar cualquier tipo de entrada que se quiera, pensando que en un fúturo, si se optara por ingresar el texto expuesto en varios formatos, se precisaría simplemente hacer una nueva implementación de esta interfaz. De esta forma también tuvimos en cuenta el principio OCP de SOLID. Ya que el formato para chequear los data breaches queda abierto para su extensión, sin necesidad de modificar el código.

Implementamos esta interfaz en dos casos entonces:

- ***PlainTextAdapter*** → para la adaptación de texto plano a lista de strings, tomando como separación las nuevas líneas.
- ***TxtFileAdapter*** → para la adaptación de archivos txt a lista de strings, tomando como separación '\t'.

Para introducir la funcionalidad del histórico de data breaches, nos vimos obligados a realizar ciertos cambios, no muy significativos.
En primer lugar, agregamos a la clase DataBreachCheck un atributo que guarda la fecha de tipo DateTime en que se realizó el data breach. Esta fecha se setea de forma automática en el constructor de la clase.
Además, esa misma clase la volvimos persistente en nuestra base de datos. Para ello fue necesario modelar una clase llamada DataBreachLines que guarda un string. Entonces, en lugar de guardar los data breaches en listas de strings, debimos convertirlos a listas de DataBreachLines, para que las tablas de la base de datos se generan correctamente.

Cabe mencionar que el criterio de comparación utilizado para establecer si una contraseña mostrada en el histórico de data breaches fue modificada es si la fecha de modificación del Account es posterior a la fecha de realización del Data Breach. Este punto fue muy cuestionado a lo largo del trabajo, porque en nuestro caso, la fecha de modificación de los accounts varía por cualquier tipo de modificación en los mismos, y no necesariamente al cambiar el string de la contraseña. Es decir, si en un account se modifica el nombre de usuario, pero la contraseña se mantiene, la fecha de modificación cambia. Esto nos llevó a pensar por mucho tiempo que el criterio no debía ser por fechas, sino por comparación del string de la contraseña.
Por otro lado, en la interfaz de usuario, al seleccionar la opción de modificar un account, todos los campos del mismo se autocompleta con los datos, menos el de la contraseña, que aparece vacío, forzando de esta manera al usuario a modificarla. Si el usuario quisiera que la contraseña no se modifique, debería intencionalmente ingresar la misma que ya estaba. Con el objetivo de demostrar gráficamente lo explicado anteriormente, se adjuntará una imagen de la interfaz de usuario al elegir modificar una cuenta (imagen 11 del Anexo).

Por esta forma de mostrar el panel de modificación, decidimos guiarnos por las fechas, ya que lo más lógico sería no ingresar nuevamente la misma contraseña.

Como reflexión final acerca de los data breaches, creemos que desde una primera instancia tomamos muy buenas decisiones de diseño para esta funcionalidad, lo que nos permitió ahorrarnos mucho trabajo en la segunda entrega y hacer cambios muy sencillos.

Comprobamos de forma práctica todo lo estudiado en la clase teórica sobre la importancia del diseño de la aplicación y los resultados positivos que se obtienen al tener una aplicación diseñada siguiendo con las reglas y patrones establecidos.

Exceptions

Ver diagrama de clases de Exceptions en Anexo.

Para el manejo de errores y excepciones, hemos creado una carpeta ***Exceptions***, que se encuentra en el paquete donde tenemos el dominio. Esta carpeta fue creada con el propósito de mantener un orden y tener todas nuestras excepciones guardadas en un mismo lugar.

Hemos intentado seguir un mismo criterio a la hora de crear excepciones. Entendimos que para capturar las excepciones lanzadas con el try and catch, lo más eficiente sería capturar una excepcion general ***InvalidCreditCardException*** (en este caso con tarjeta de crédito) y luego excepciones más específicas que vayan a heredar de esta excepción general mencionada anteriormente. Por ejemplo, ***InvalidCreditCardCodeException*** o ***InvalidCreditCardNumberExcpetion.***
Cada una de nuestras Excepciones específicas, es lanzada con su mensaje correspondiente. Esto ayuda a que cuando el usuario comete un error o realiza algo que no es esperado por nuestro programa, este pueda darle feedback más acertado y atinado sobre cómo debe ingresarse un dato o porque se cometió este error.
Este criterio para las excepciones fue manejado de igual manera con las contraseñas.

Por otro lado, también fueron creadas excepciones para cuando la masterkey ingresada es incorrecta, o para cuando ningún checkbox es seleccionado a la hora de generar una contraseña.

Interface (Interfaz de usuario)

Ver diagrama de clases en Anexo.

Con respecto a la ventanas, cuando se ejecuta la aplicación, se muestra una ventana donde el usuario puede hacer login. Al hacerlo correctamente, se muestra la ventana principal, desde la cual se pueden ver una serie de botones que refieren cada uno a las distintas funcionalidades del gestor de contraseñas.
Estas dos ventanas mencionadas son las únicas que decidimos modelar como *forms* en nuestro proyecto. Todo el resto de las funcionalidades fueron creadas como clases que heredan de *UserControl*. De esta manera, la ventana principal contiene un panel, en el cual se agregan y van cambiando los distintos UserControls creados.
Al menos que el usuario cierre su sesión, siempre se va a mantener en esa misma ventana principal, donde el panel mostrará los diferentes user control dependiendo lo que quiera ver el usuario.
Cabe destacar que al implementar las ventanas, surgió la duda de cómo modificar el contenido mostrado en el panel de la ventana principal desde las diferentes clases creadas. Para hacerlo, optamos por tener guardado dicho panel como atributo privado de las clases que lo necesitaran. Para ello, creamos constructores por parámetro de las mismas que reciben la instancia del panel.

Anteriormente, todas las clases de nuestra interfaz tenían un atributo privado con la instancia de la clase User correspondiente a la aplicación. Ahora que nuestro sistema es persistente y la clase User ya no se encarga mas de manejar listas con los datos, no precisamos más de dicho atributo. Cada una de nuestras clases de la interfaz que precisa acceder a algún data access, tiene definido como atributo privado un IDataAccess<entidad>. Este atributo tendrá la instancia del data access creada en el singleton. Se accede a través del DataAccessManager e invocando al método GetDataAccess, dependiendo al data access que se quiera acceder.
De esta forma, se puede acceder a la información de la base de datos cuando sea necesario.

Algunas clases que heredan de UserControl tienen dos constructores por parámetro posibles. Algunas de ellas serían, por ejemplo, la ventana de agregar contraseñas o la ventana de agregar tarjetas. ¿A qué se debe la existencia de estos dos constructores? Si nos fijamos con atención, mirando el ejemplo de la clase de AddPassword uno de ellos tiene un Account y el otro no. Se adjuntará una imagen de dicho ejemplo en el Anexo.

Esto lo hicimos para poder reutilizar la ventana si se quisiera modificar o agregar contraseñas, de forma indistinta. De esta forma, si el UserControl se crea con el primer constructor de la foto, es decir, no se le pasa una cuenta para modificar, el sistema interpreta que se pretende crear una cuenta nueva. En el otro caso, se procede a modificar la cuenta recibida por parámetro.

A la hora de querer modificar una contraseña, al usuario se le abre una ventana con los datos de la contraseña que quiere modificar, ya precargados, de modo que solo tenga que modificar lo que precise, sin tener que volver a ingresar el resto de datos. Sin embargo, cuando ocurre eso, aparece la primera categoría registrada en la lista, y no precisamente la categoría asociada a la contraseña que se quiere modificar. Si bien no es un error, creemos que se podría mejorar.

Por otro lado, creemos que es muy conveniente que la interfaz de usuario le pueda dar un feedback al usuario en caso de que este ingrese algún dato invalido, en un formato no aceptado o simplemente quiera hacer algo que la aplicación no permite. Para ello, nuestros user control tienen una label que en un comienzo se encuentra vacía. Al momento en que el usuario cometa algún error, aparecerá el texto de la excepción lanzada por el sistema en rojo en la etiqueta previamente mencionada. De esta forma el usuario entiende que fue lo que hizo mal. Mismo las sugerencias que brinda el sistema acerca de la contraseña que se quiere registrar, se muestran a través de labels coloridos en pantalla. Esto permite que nuestro programa sea lo más amigable posible, y le permita al usuario tener una buena experiencia.

Notemos que uno de los requerimientos de este obligatorio consiste en mostrar durante 30 segundos los detalles completos de las contraseñas o tarjetas de crédito guardadas en el sistema a medida que se seleccionen de los respectivos listados. Sin embargo, la selección de estos datos en un listado por parte del usuario también puede hacerse con el propósito de proceder a eliminar o modificar el dato seleccionado. Aquí nos encontramos con un dilema, ya que, si cada vez que se seleccione una tarjeta o contraseña se ve el detalle de los datos, el usuario se vería obligado a esperar 30 segundos para poder hacer modificaciones de algún tipo. Por esta razón, tomamos la decisión de diferenciarlo con un único click y un doble click. Cuando se presiona en el listado de contraseñas o de tarjetas una fila con doble click, se muestra el detalle de los datos de la selección durante 30 segundos. De lo contrario, cuando esto se hace con un solo click, el ítem queda seleccionado para poder modificarlo o eliminarlo.

Repository

Ver diagrama de clases en Anexo.

Para tener conexión con una base de datos y lograr que nuestro sistema sea persistente, fue creado este proyecto Repository.
Principalmente tenemos implementado una clase llamada PasswordManagerDBContext que hereda de la clase DbContext. Este es el contexto que usamos para poder lograr la persistencia a la base de datos. En el constructor definimos un name que en nuestro caso es igual al nombre de la clase, y es el que incluimos en el connection string, utilizado en el app.config de Repository y de Interface.

```
public PasswordManagerDBContext() : base("name=PasswordManagerDBContext") { }
```

También tenemos definido en este contexto todas las entidades que vamos a persistir en nuestra base de datos. Es decir que por cada entidad que tengamos definida a través de las

migraciones, se creara una tabla en nuestra base de datos. Cuando modificamos nuestro contexto, debimos agregar una migración(add-migration) y luego actualizar nuestra base de datos para que estos cambios se apliquen en la misma(update-database).

En este proyecto también están implementados los DataAccess para cada una de las entidades que conforman nuestro sistema.

Para los DataAccess fue implementada una interface llamada IDataAccess. Esta interface es implementada por cada uno de los DataAccess. En este contrato se definen todos los métodos necesarios para agregar, modificar, obtener o borrar datos de nuestra base de datos.
Estos métodos son los siguientes:

T Get(T entity); → Devuelve la entidad que se necesita.

IEnumerable<T> GetAll() → Devuelve todas las entidades en formato de lista

void Add(T entity); → Agrega la entidad que se pasa por parametro a la base de datos

void Clear() → Borra todos los datos que haya en la tabla de la base de datos.

void Delete(T entity) → Borra la entidad que se pasa por parametro de la base de datos

void Modify(T entity) → modifica la entidad con los datos de la entidad que se pasa por parametro.

Para poder implementar esto de la mejor manera, decidimos utilizar un patrón de diseño llamado “Singleton”. Creamos una clase llamada DataAccessManager que será precisamente nuestro singleton mencionado anteriormente. Esta clase tiene como atributo privado una instancia de cada uno de los DataAccess.
La idea de aplicar este patrón es tener que crear una única instancia de cada uno de los data access, y no tener que estar creando nuevos objetos cada vez que se lo quiera utilizar.

La clase tiene métodos públicos Get, para cada uno de los data access, lo que permite acceder a dichas instancias desde donde sea necesario.

En el proyecto de la interfaz, las clases que necesitan utilizar algún data access, podrán acceder al mismo ya que tendrán como atributo la instancia del objeto.

Por ejemplo, para acceder al DataAccessCategory en alguna clase del proyecto de la interfaz, definimos lo siguiente:
private IDataAccess<Category> dataAccessCategory = DataAccessManager.GetDataAccessCategory();

Los data access que definimos son los siguientes DataAccessCategory, DataAccessAccount, DataAccesCreditCard, DataAccessUser, DataAccessDataBreaches. Cabe destacar que cada uno de estos data access manejara la persistencia de las entidades que su nombre lo indica.

Cabe aclarar que en DataAccessUser, la persistencia que manejamos es la de la masterkey, ya que al ser una aplicación monousuario, en la tabla de User nunca tendremos más de una tupla. En esta tabla habrá una tupla si ya hay una masterkey, sino la tabla estará vacía indicando que no hay ningún usuario registrado.

Repository Tests

En este paquete, fueron implementadas las pruebas unitarias que chequean el correcto funcionamiento de los data access. Por cada Data access creado en Repository, fue creada una clase en este paquete con sus respectivas pruebas.

Para la realización de estas pruebas, se decidió trabajar sobre una base de datos nueva, para no modificar la base de datos con la que se estaba trabajando anteriormente. Para ello, se instaló el paquete de Entity Framework en este nuevo proyecto. Además fue necesario establecer un nuevo connection string en el archivo app.config, definiendo en el initial catalog el nombre de la “base de datos”.
Es importante aclarar que cada una de estas clases de prueba, consta de un Test initialize que se encarga de que la base de datos esté vacía para cada prueba. Para ello se declaró un método SetUp(), en el cual se invoca al método Clear por cada data access que se vaya a utilizar. Estas pruebas nos ayudaron a confirmar el correcto funcionamiento de los métodos implementados en los Data Access.

Modelo de Tablas de la Base de Datos

Para realizar el modelo de tablas de la base de datos, abrimos nuestra base de datos desde el management studio. Luego hicimos click derecho en la carpeta Database diagrams y elegimos la opción New Database Diagrams. Allí elegimos todas las tablas de nuestra base de datos excepto por la tabla de migrations que creímos no era necesaria. Luego se nos generó el diagrama y quedó como se puede apreciar en la imagen del anexo.

Pruebas

Cobertura de pruebas unitarias

Con respecto a la cobertura de los tests, no hemos tenido muchos inconvenientes. El porcentaje total de cobertura que obtuvimos fue de un 85,41%. Esto significa que solo un 14,59% de nuestro código no quedó cubierto por los tests.

Si bien el porcentaje de cobertura no es muy alto y es aún menor que el de la primera entrega, esto tiene una explicación lógica.

Si observamos la captura de pantalla adjunta en la imagen 13 del Anexo de este documento, se puede notar que el porcentaje de cobertura del proyecto Repository es de 55,89%, siendo muy

inferior al del resto de los proyectos. Esto se explica con el porcentaje de cobertura de las migrations a la base de datos que es de 0,66%. Esto se debe a que no se les hizo tests a las migrations.
Este porcentaje lógicamente afecta al porcentaje de cobertura total de nuestro código, y debido a esto es que se encuentra alrededor de 85%.
En el resto de los proyectos logramos una cobertura bastante alta logrando superar el 90% de porcentaje de cobertura.

En la siguiente captura de pantalla (imagen 14 del Anexo) se puede observar como casi todos los métodos del dominio fueron testeados con nuestras pruebas unitarias. En el caso de la clase TxtFileAdapter, que es la clase que se encarga de adaptar un archivo txt al formato de datos que utiliza la clase de chequeo de data breaches el porcentaje es un poco menor. Esto se debe a que no pudimos mockear un file system con un archivo txt en su interior y por lo tanto, no supimos cómo hacer para chequear la correctitud del path recibido por la clase TxtFileAdapter. Sin embargo, testeamos que el texto obtenido de los archivos se adapte correctamente y también hicimos pruebas para paths erróneos, haciendo que nuestras pruebas cayeran en exceptions. De esta forma, intentamos hacer la mayor parte de la clase con TDD, pero nos faltó probar el caso en el que se recibe un path válido.

En la imagen 15 se puede observar el porcentaje de cobertura de nuestro proyecto Repository mas detalladamente. Como podemos ver todos los data access tienen una cobertura de 100%, menos el DataAccessManager.
Este DataAccessManager es el singleton que maneja las instancias de todo el resto de los data access. Y al tener todo el resto de los Data access testeados, entendimos que no hubiese sido necesario hacerle tests al Manager. Además de que observamos que las pruebas para un singleton pueden ser complicadas.

Donde sí hemos tenido algunas complicaciones fue con el tema de las excepciones, ya que no pudimos alcanzar un porcentaje superior al 50%. (Ver captura en anexo)

Si bien entendemos que este porcentaje es bastante pequeño, los motivos por los cuales esto ocurre son entendibles. Para cada una de las excepciones, nosotros hemos hecho tests, verificando que funcionen bien y que el programa las lance cuando sea necesario. Esto lo hicimos utilizado el siguiente comando: [ExpectedException(typeof(NombreDeException))]
Por otro lado cada excepción fue implementada con un mensaje, que tiene un texto donde se explica el motivo de dicha excepción. Entonces lo que nos faltó fue realizar tests donde se analice solamente al mensaje lanzado por la excepción. Sin embargo entendemos que fue así por la manera en la que nosotros implementamos las excepciones.
La única Excepción a la que no le hicimos ninguna prueba es a InvalidNullInputException, esta excepción es lanzada cuando se quiere agregar una tarjeta o una contraseña sin tener una categoría asignada. Esta excepción es capturada desde la interfaz y no desde el dominio, debido a esto no fue posible testearla y aparece con el 0% de cobertura.

Concluimos que nuestro nivel de cobertura con las pruebas unitarias es alto, y se vio perjudicado por el porcentaje de nuestras excepciones y por las migrations. Si la cobertura de nuestras excepciones fuese mayor, también habría sido mayor el porcentaje total de nuestro código. Para eso, deberíamos haber chequeado que cada excepción retorne el mensaje correcto.

Anexo

Imagen 1 - Diagrama de paquetes

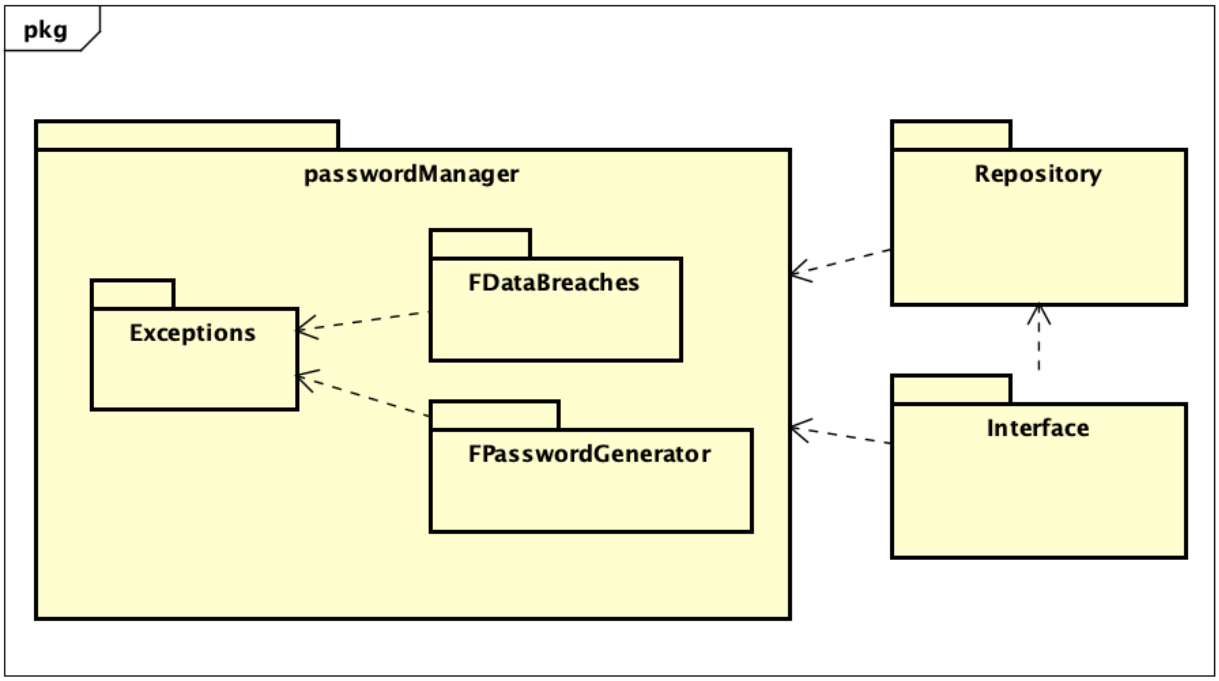


Imagen 2 - Diagrama de clases: PasswordManager

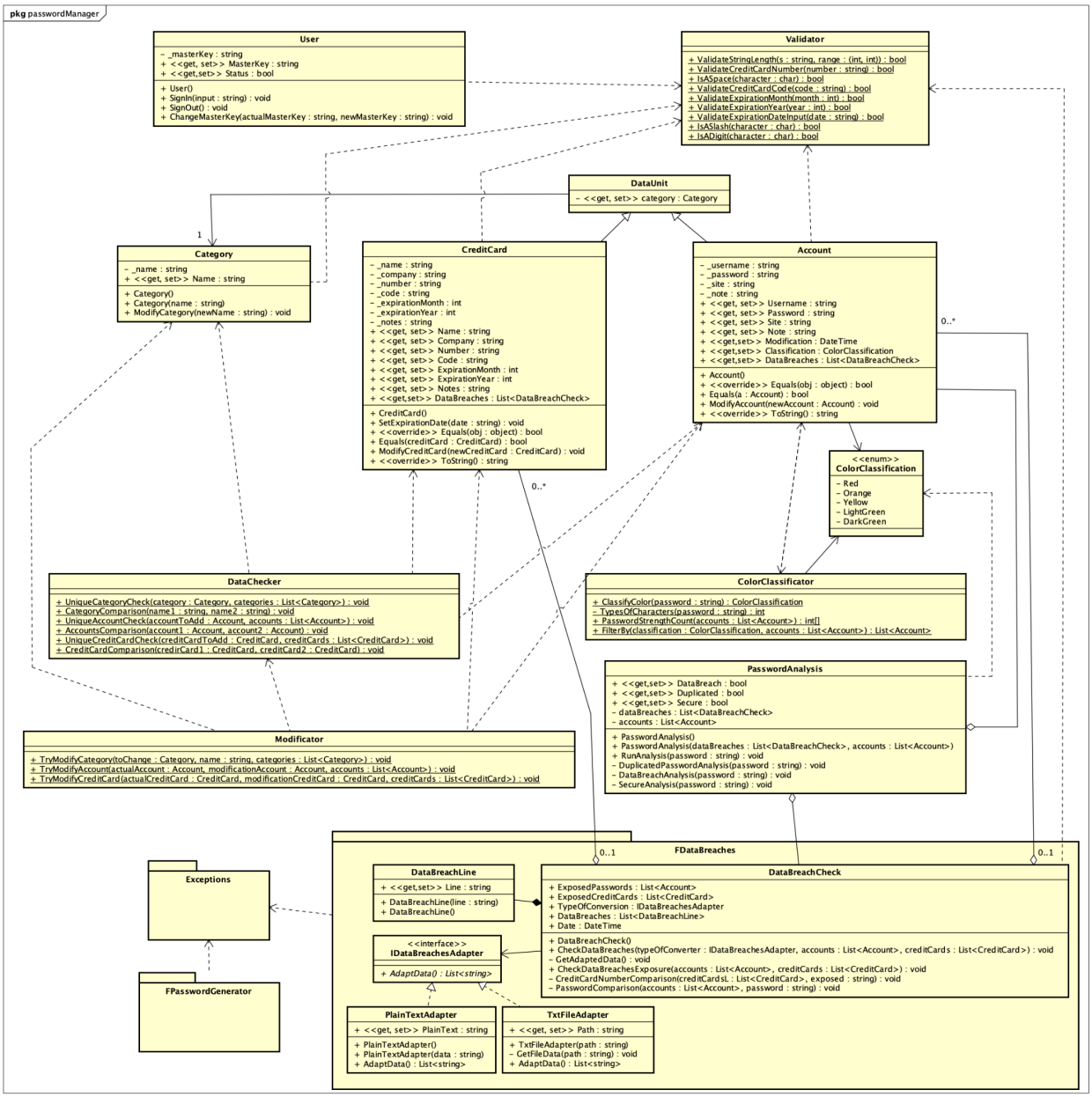


Imagen 3 - Diagrama de clases: PasswordManager.Exceptions

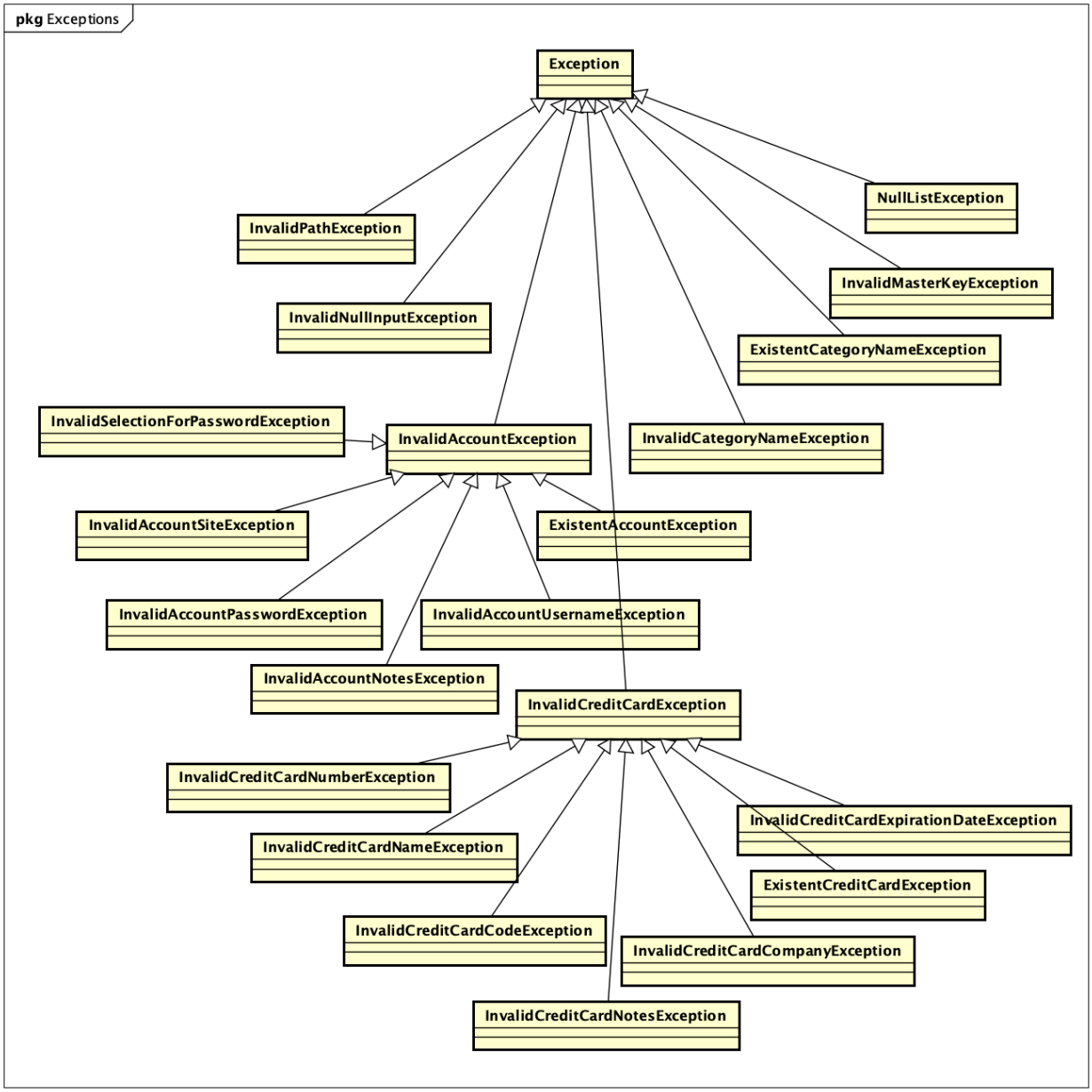


Imagen 4 - Diagrama de clases: PasswordManager.FPasswordGenerator

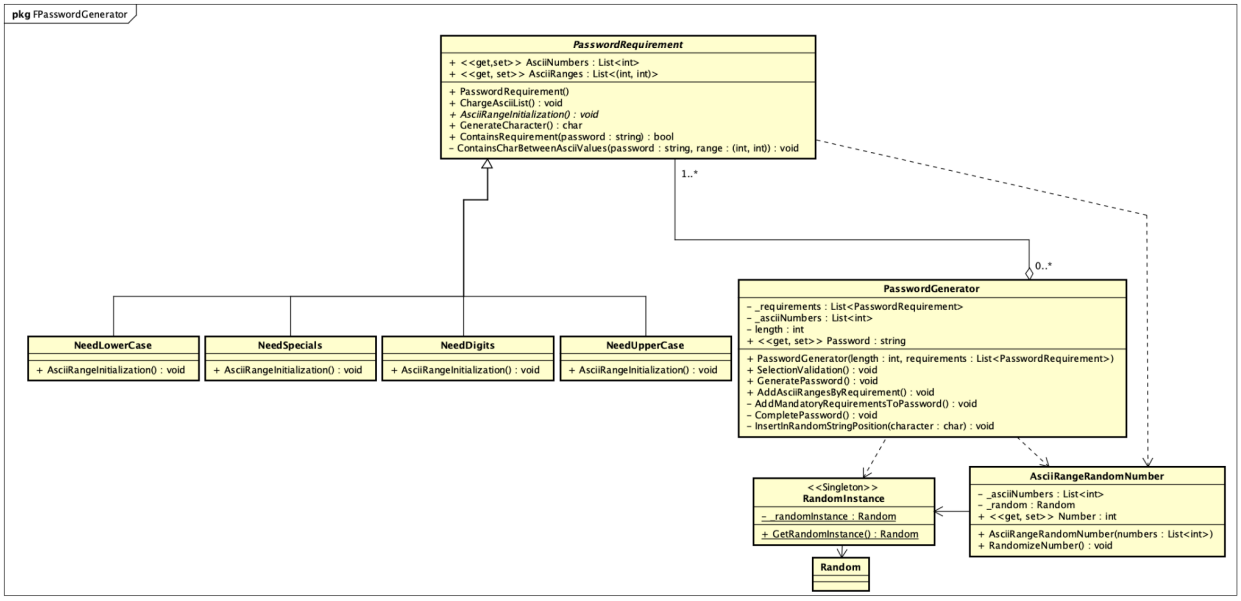


Imagen 5 - Diagrama de clases: PasswordManager.FDataBreaches

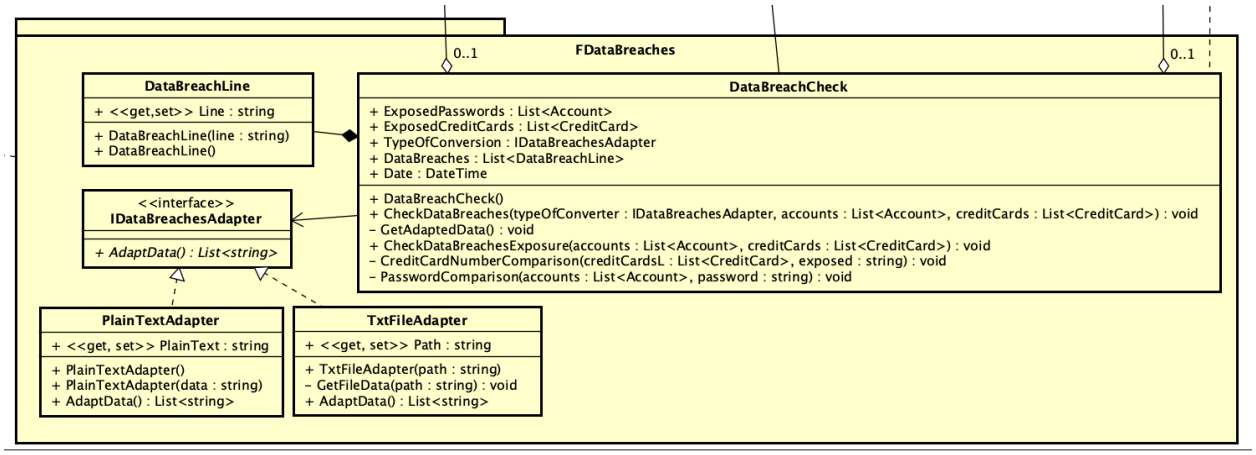
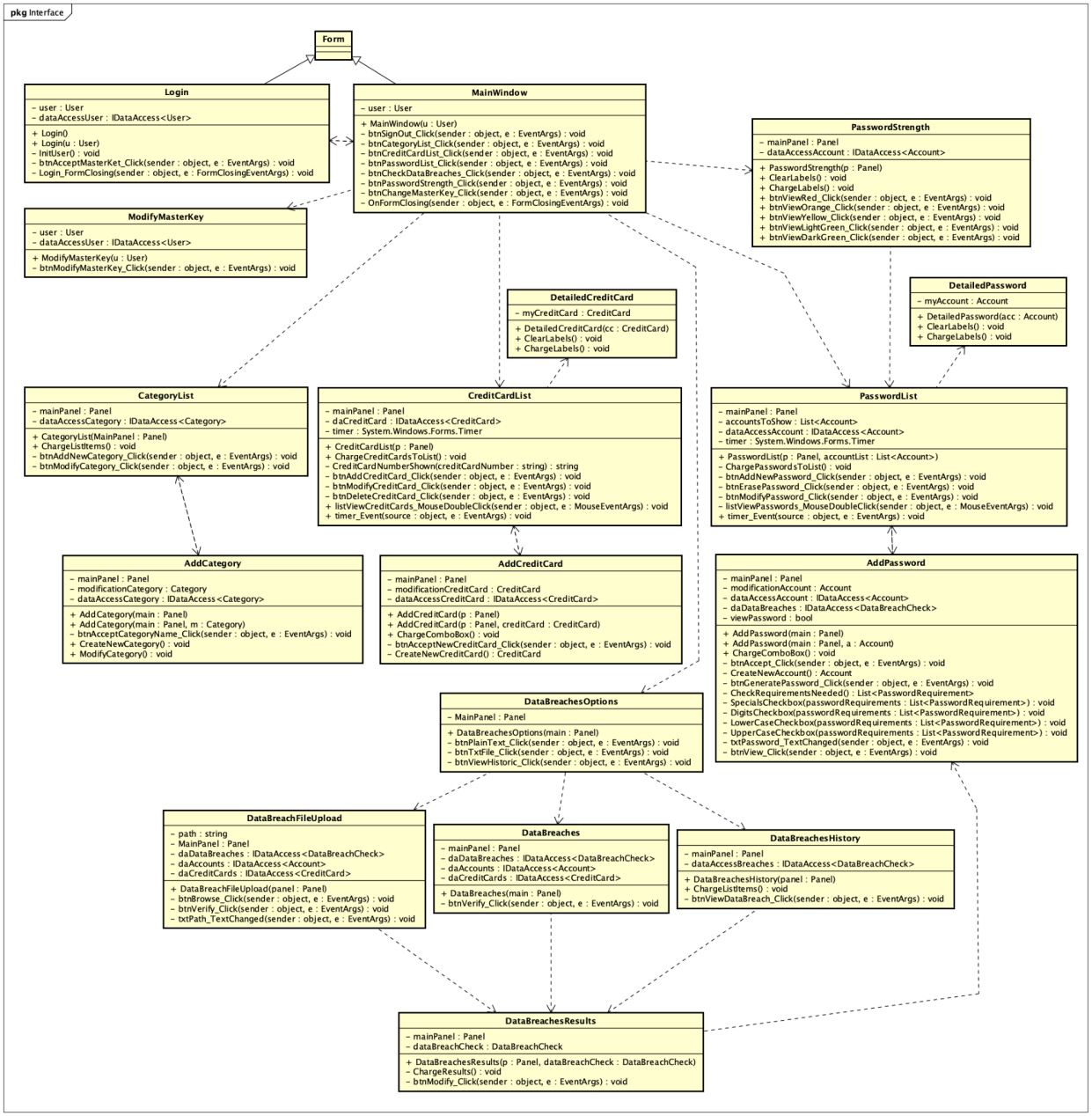


Imagen 6 - Diagrama de clases: Interface



Aclaración: todas las clases del diagrama que no aparecen como herederas de Form, heredan de la clase UserControl.

Imagen 7 - Diagrama de clases: Repository

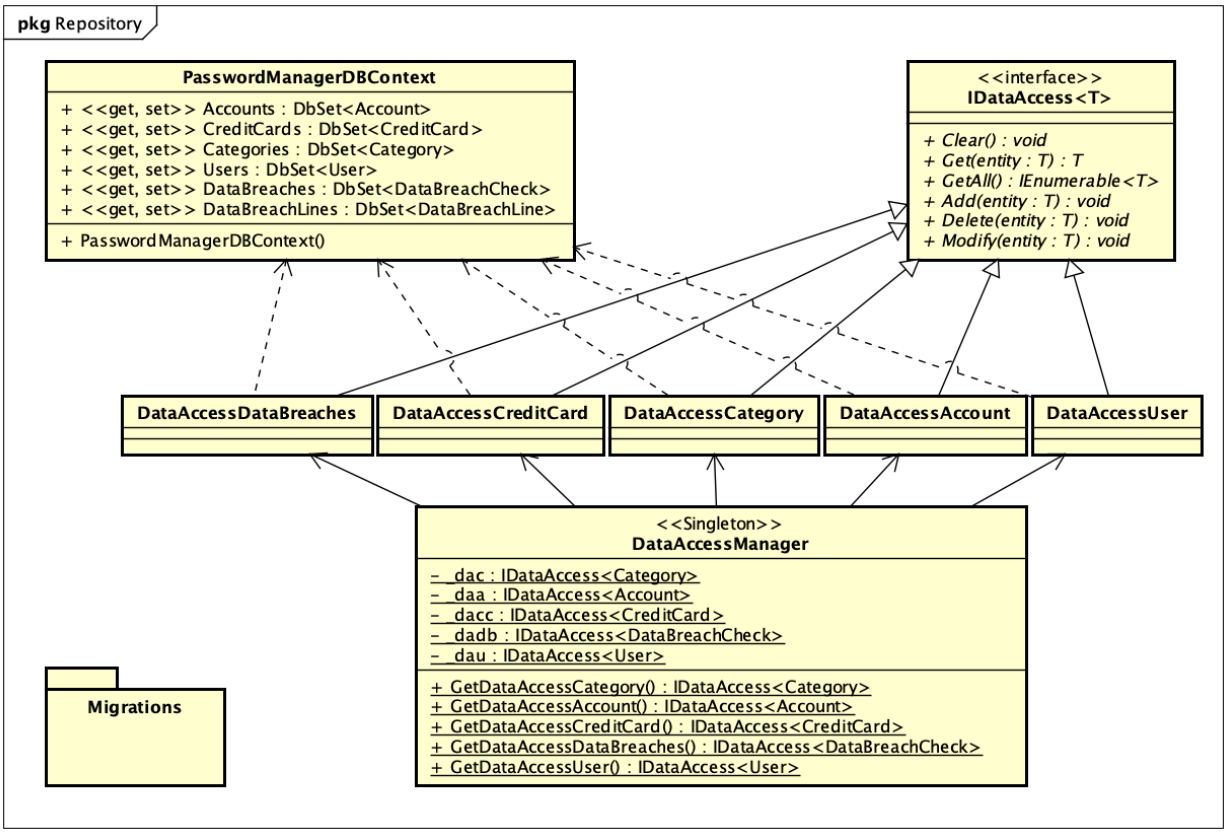


Imagen 8 - Diagrama de secuencia: agregar una cuenta

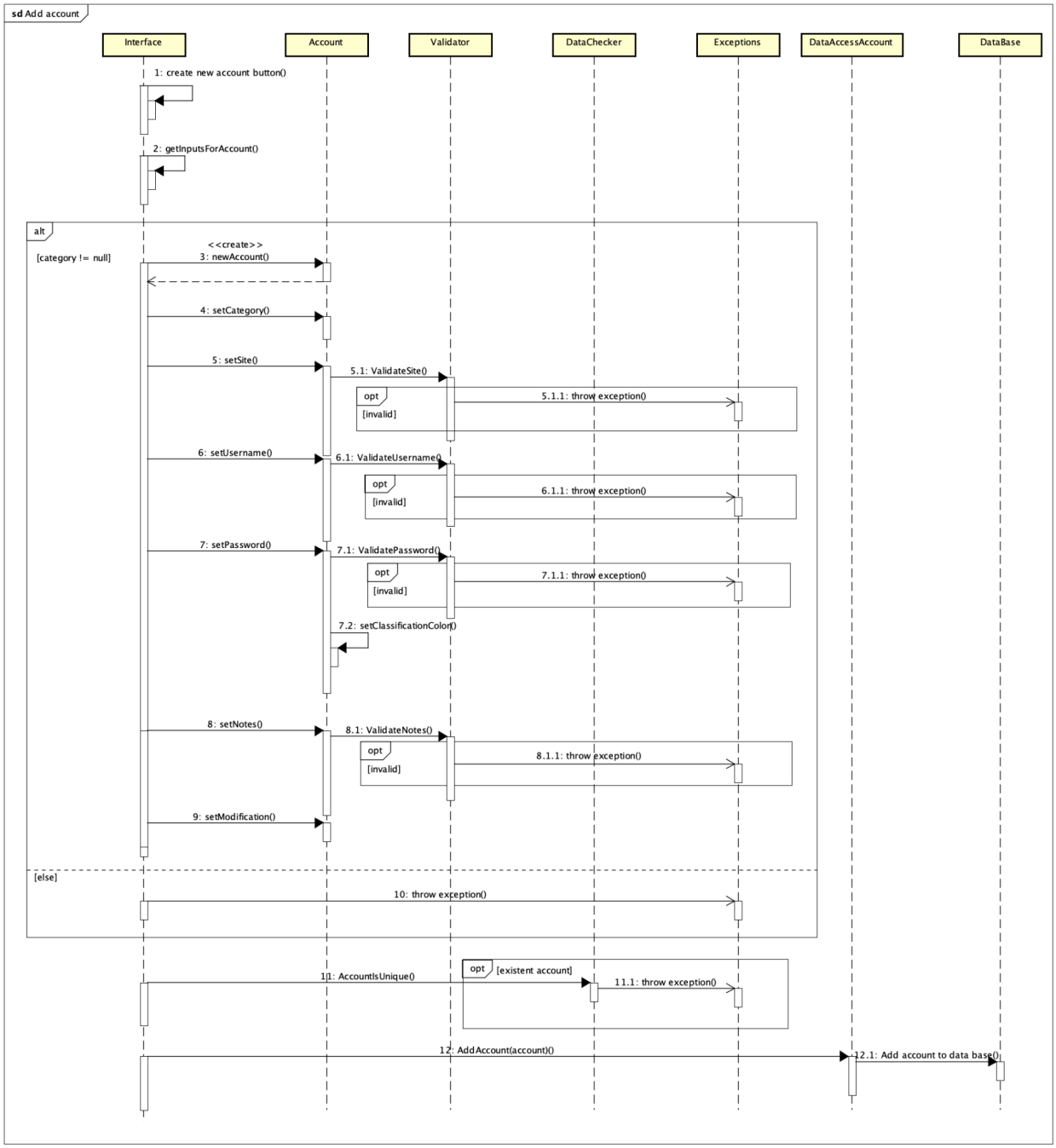


Imagen 9 - Diagrama de secuencia: chequeo de data breaches con texto plano

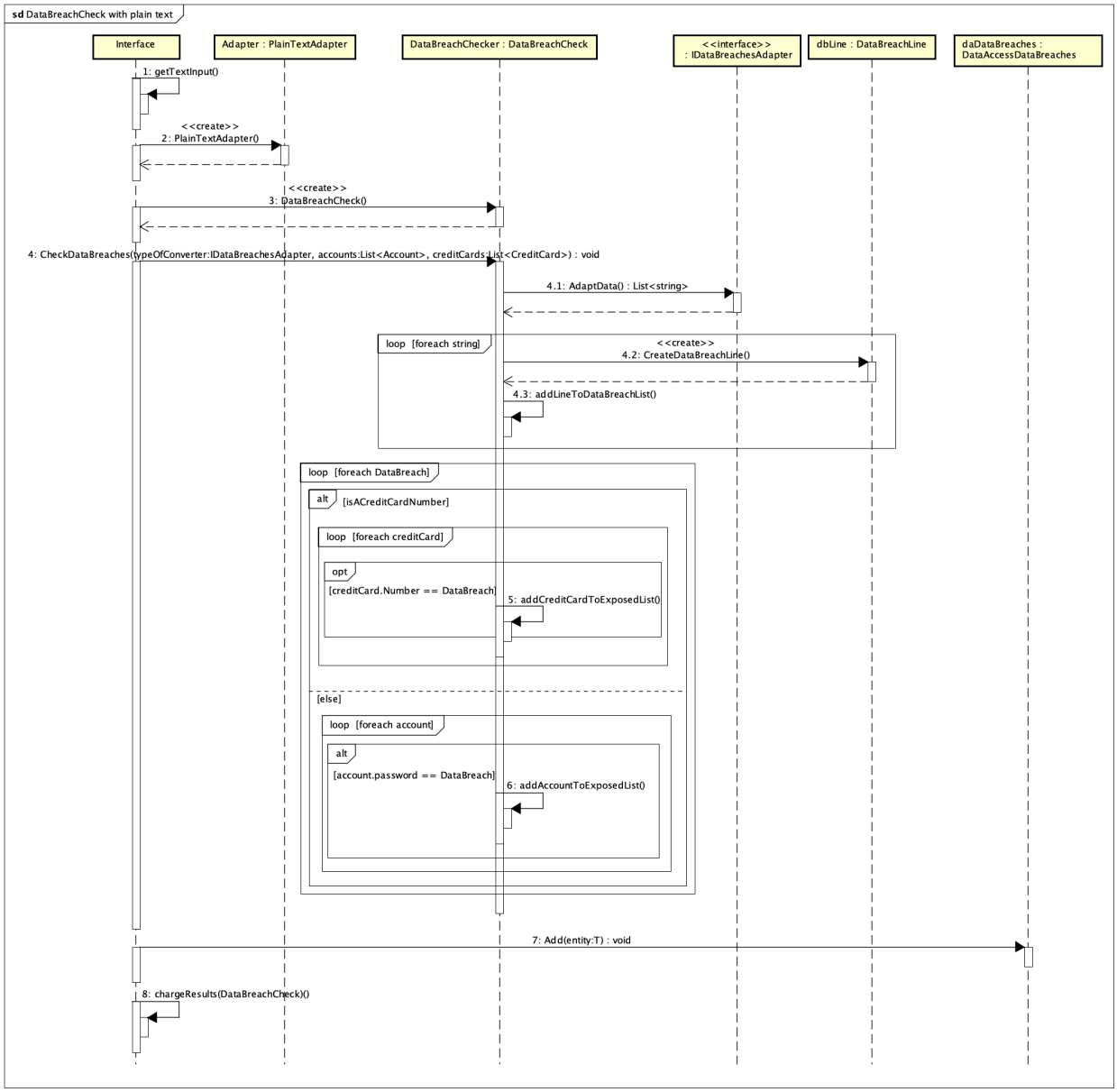


Imagen 10 - Diagrama de secuencia: modificación de master key

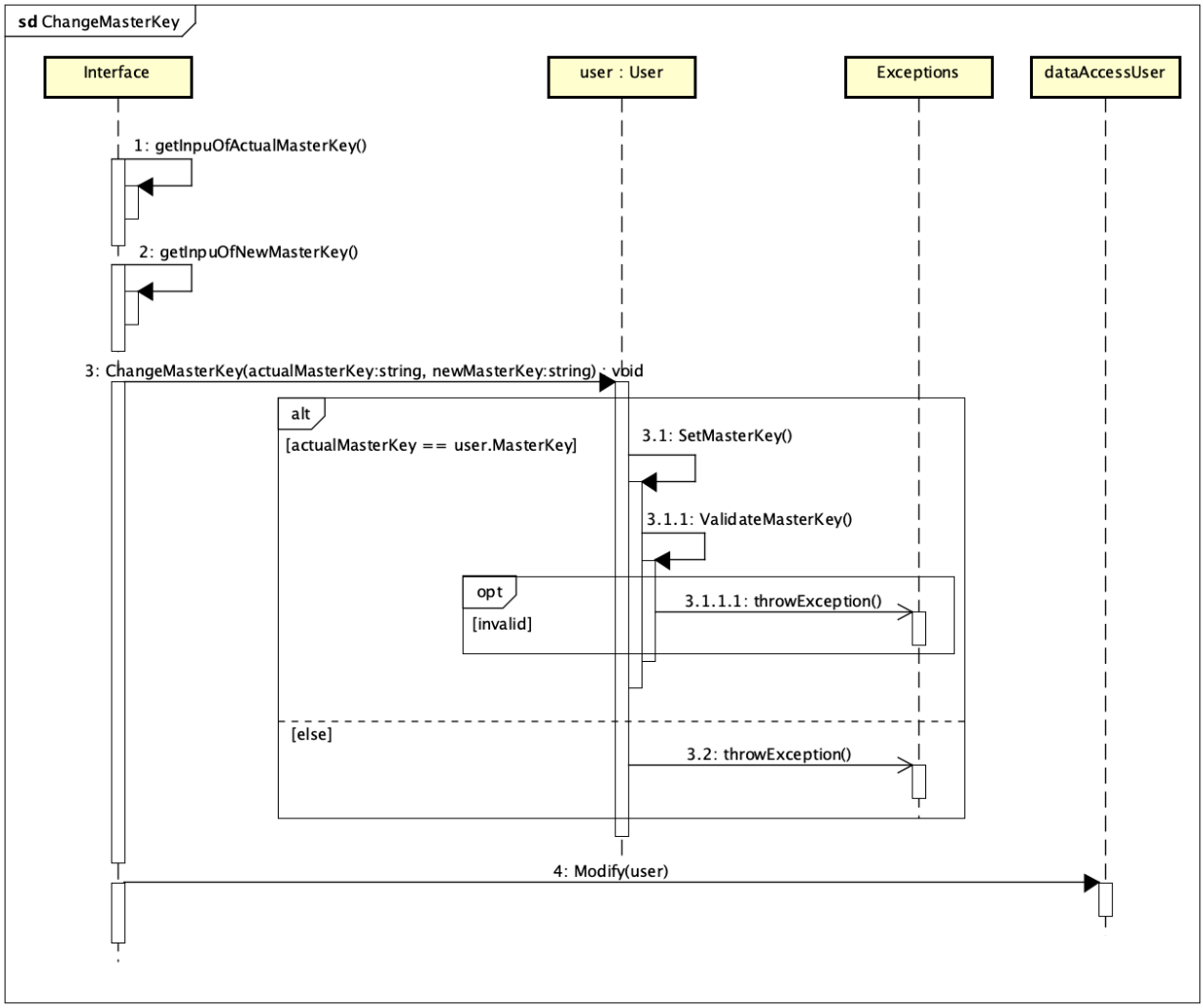


Imagen 11 - UI modificación de contraseñas

Imagen 12 - Dos constructores por parámetros en User Controls de alta/modificación

```
public partial class AddPassword : UserControl
{
    private Account modificationAccount;
    private Panel mainPanel;
    private IDataAccess<Account> dataAccessAccount = DataAccessManager.GetDataAccessAccount();
    private IDataAccess<DataBreachCheck> daDataBreaches = DataAccessManager.GetDataAccessDataBreaches();
    private bool viewPassword;

    1 reference | iritrocki, 2 days ago | 1 author, 2 changes
    public AddPassword(Panel main)
    {
        InitializeComponent();
        txtPassword.PasswordChar = '*';
        lblError.Text = "";
        lblDataBreaches.Text = "";
        lblDuplicated.Text = "";
        lblSecure.Text = "";
        this.viewPassword = false;
        this.mainPanel = main;
        ChargeComboBox();
    }

    2 references | iritrocki, 2 days ago | 1 author, 2 changes
    public AddPassword(Account a, Panel main)
    {
        InitializeComponent();
        this.modificationAccount = a;
        txtPassword.PasswordChar = '*';
        lblError.Text = "";
        lblDataBreaches.Text = "";
        lblDuplicated.Text = "";
        lblSecure.Text = "";
        this.viewPassword = false;
        this.mainPanel = main;
        txtSite.Text = string.Format("{0}", a.Site);
        txtNotes.Text = string.Format("{0}", a.Note);
        txtUsername.Text = string.Format("{0}", a.Username);
        ChargeComboBox();
    }
```

Imagen 13 - Code Coverage Results

Code Coverage Results

mikael_MIKAEL 2021-06-16 16_47_00.cover

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
mikael_MIKAEL 2021-06-16 1...	585	14,59 %	3424	85,41 %
passwordmanager.exe	52	5,57 %	881	94,43 %
passwordManager	8	0,94 %	841	99,06 %
passwordManager.Exc...	44	52,38 %	40	47,62 %
passwordmanagertest.dll	57	3,70 %	1482	96,30 %
passwordManagerTest	57	3,70 %	1482	96,30 %
repository.dll	476	44,11 %	603	55,89 %
Repository	27	4,31 %	600	95,69 %
Repository.Migrations	449	99,34 %	3	0,66 %
repositorytest.dll	0	0,00 %	458	100,00 %
RepositoryTest	0	0,00 %	458	100,00 %

Imagen 14 - Code Coverage Results PasswordManager

Code Coverage Results

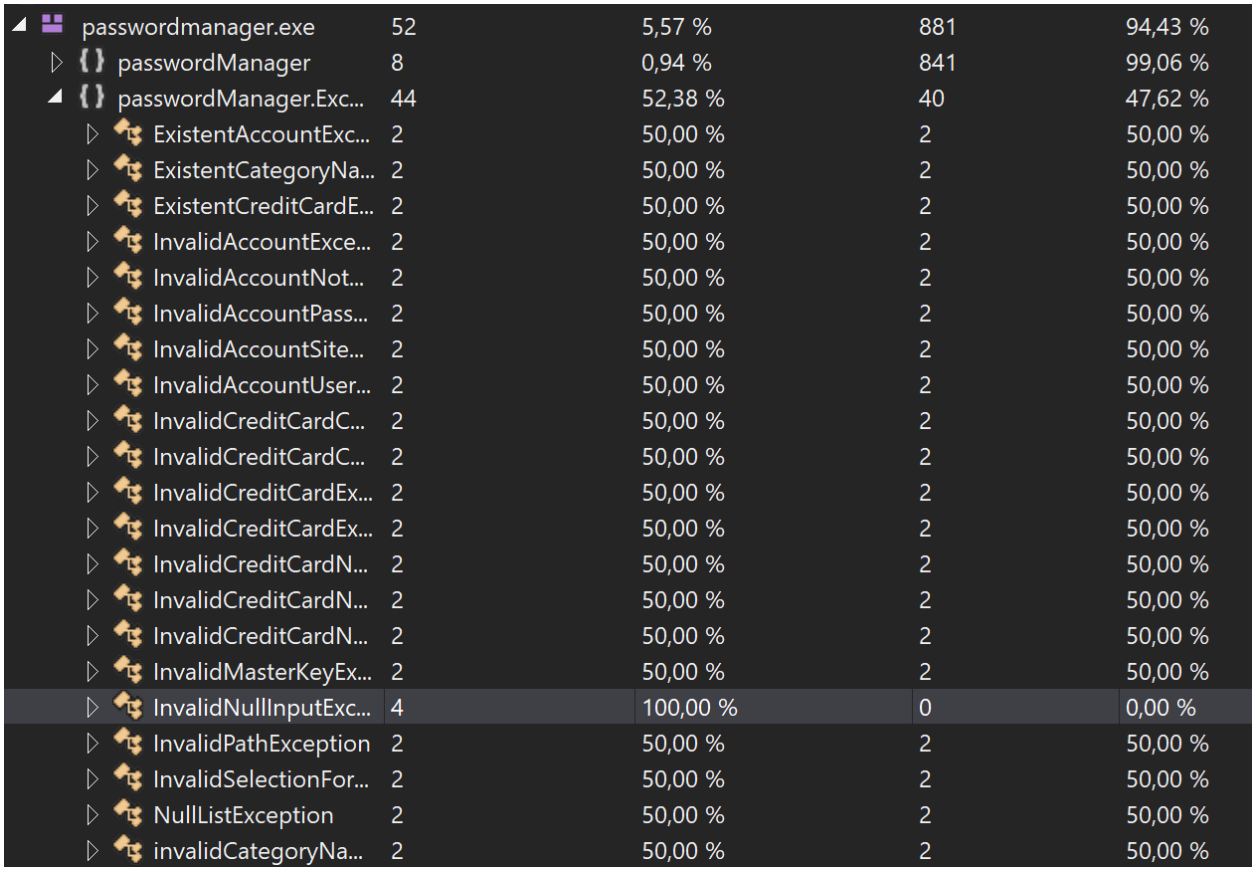
mikael_MIKAEL 2021-06-16 16_47_00.cover

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
passwordmanager.exe	52	5,57 %	881	94,43 %
passwordManager	8	0,94 %	841	99,06 %
Account	0	0,00 %	83	100,00 %
AsciiRangeRandom...	0	0,00 %	11	100,00 %
Category	0	0,00 %	17	100,00 %
ColorClassificator	0	0,00 %	69	100,00 %
CreditCard	0	0,00 %	104	100,00 %
DataBreachCheck	0	0,00 %	90	100,00 %
DataBreachLine	0	0,00 %	9	100,00 %
DataChecker	0	0,00 %	46	100,00 %
DataUnit	2	50,00 %	2	50,00 %
Modificator	0	0,00 %	43	100,00 %
NeedDigits	0	0,00 %	6	100,00 %
NeedLowerCase	0	0,00 %	6	100,00 %
NeedSpecials	0	0,00 %	15	100,00 %
NeedUpperCase	0	0,00 %	6	100,00 %
PasswordAnalysis	0	0,00 %	65	100,00 %
PasswordGenerator	0	0,00 %	61	100,00 %
PasswordRequirem...	0	0,00 %	54	100,00 %
PlainTextAdapter	0	0,00 %	13	100,00 %
Program	1	100,00 %	0	0,00 %
RandomInstance	0	0,00 %	5	100,00 %
TxtFileAdapter	4	21,05 %	15	78,95 %
User	0	0,00 %	33	100,00 %
Validator	1	1,12 %	88	98,88 %

Imagen 15 - Code Coverage Results Repository

repository.dll	476	44,11 %	603	55,89 %
Repository	27	4,31 %	600	95,69 %
DataAccessAccount	0	0,00 %	111	100,00 %
DataAccessCategory	0	0,00 %	81	100,00 %
DataAccessCreditC...	0	0,00 %	113	100,00 %
DataAccessDataBre...	0	0,00 %	200	100,00 %
DataAccessManager	27	100,00 %	0	0,00 %
DataAccessUser	0	0,00 %	81	100,00 %
PasswordManager...	0	0,00 %	14	100,00 %
Repository.Migrations	449	99,34 %	3	0,66 %

Imagen 16 - Code Coverage Results PasswordManager.Exceptions



passwordmanager.exe	52	5,57 %	881	94,43 %
passwordManager	8	0,94 %	841	99,06 %
passwordManager.Exc...	44	52,38 %	40	47,62 %
ExistentAccountExc...	2	50,00 %	2	50,00 %
ExistentCategoryNa...	2	50,00 %	2	50,00 %
ExistentCreditCardE...	2	50,00 %	2	50,00 %
InvalidAccountExce...	2	50,00 %	2	50,00 %
InvalidAccountNot...	2	50,00 %	2	50,00 %
InvalidAccountPass...	2	50,00 %	2	50,00 %
InvalidAccountSite...	2	50,00 %	2	50,00 %
InvalidAccountUser...	2	50,00 %	2	50,00 %
InvalidCreditCardC...	2	50,00 %	2	50,00 %
InvalidCreditCardC...	2	50,00 %	2	50,00 %
InvalidCreditCardEx...	2	50,00 %	2	50,00 %
InvalidCreditCardEx...	2	50,00 %	2	50,00 %
InvalidCreditCardN...	2	50,00 %	2	50,00 %
InvalidCreditCardN...	2	50,00 %	2	50,00 %
InvalidCreditCardN...	2	50,00 %	2	50,00 %
InvalidMasterKeyEx...	2	50,00 %	2	50,00 %
InvalidNullInputExc...	4	100,00 %	0	0,00 %
InvalidPathException	2	50,00 %	2	50,00 %
InvalidSelectionFor...	2	50,00 %	2	50,00 %
NullListException	2	50,00 %	2	50,00 %
invalidCategoryNa...	2	50,00 %	2	50,00 %

Imagen 17 - Modelo de tablas de la base de datos