

# Obligatorio 1 - Diseño de Aplicaciones I

## **Autores:**

Matías Lijtenstein - 231070

Irina Trocki - 203608

## Índice

<b>Índice</b>	<b>1</b>
<b>Descripción general del trabajo y el sistema</b>	<b>2</b>
<b>Descripción y justificación de diseño</b>	<b>2</b>
Diagramas de paquetes	2
Diagramas de clase	3
PasswordManager	3
PasswordManager.Exceptions	4
Interface	5
Explicación de los mecanismos generales y descripción de las principales decisiones de diseño tomadas.	6
Uso de Git	6
Paquetes	7
PasswordManagerTests (pruebas unitarias)	7
PasswordManager (dominio)	7
Category	8
Account	8
CreditCard	9
DataUnit	9
User	10
Data Breaches	11
Exceptions	11
Interface (Interfaz de usuario)	12
<b>Pruebas</b>	<b>14</b>
Cobertura de pruebas unitarias	14
Link a vídeo de youtube con pruebas	15

## **Link a repositorio:**

[https://github.com/ORT-DA1/203608\\_231070](https://github.com/ORT-DA1/203608_231070)

# 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.

El sistema también brinda la posibilidad de chequear Data Breaches y 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.

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 asignábamos 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*.

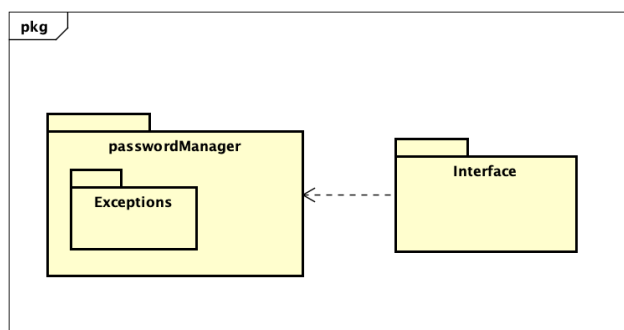
Además, nos quedó un bug sin solucionar por temas de tiempo:

Uno de ellos es la validación del formato al ingresar una fecha para las tarjetas de credito.

Nuestro programa está diseñado para aceptar fechas en el formato mm/yyyy pero esto no lo validamos al momento de tomar el string de la fecha.

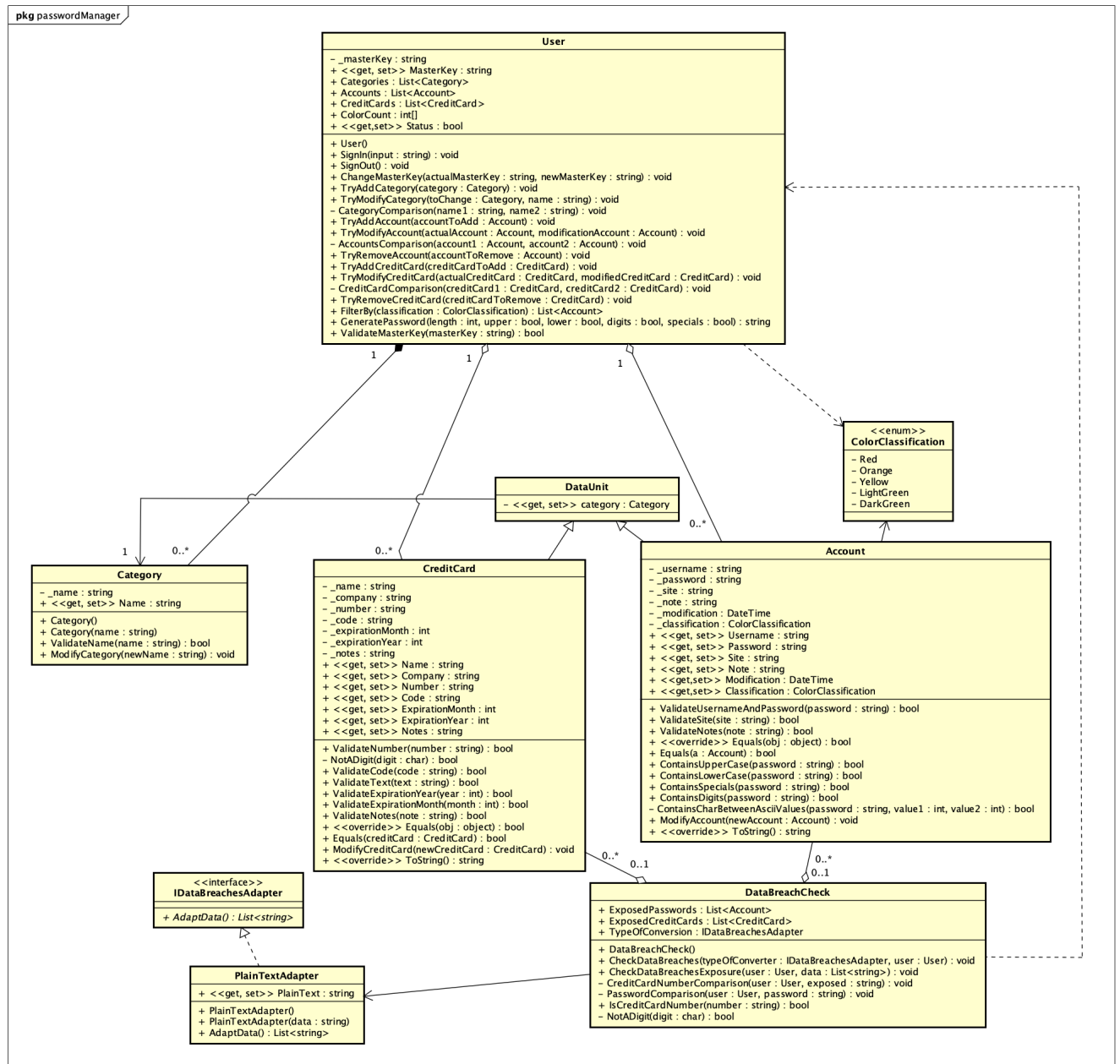
## Descripcion y justificacion de diseño

### Diagramas de paquetes

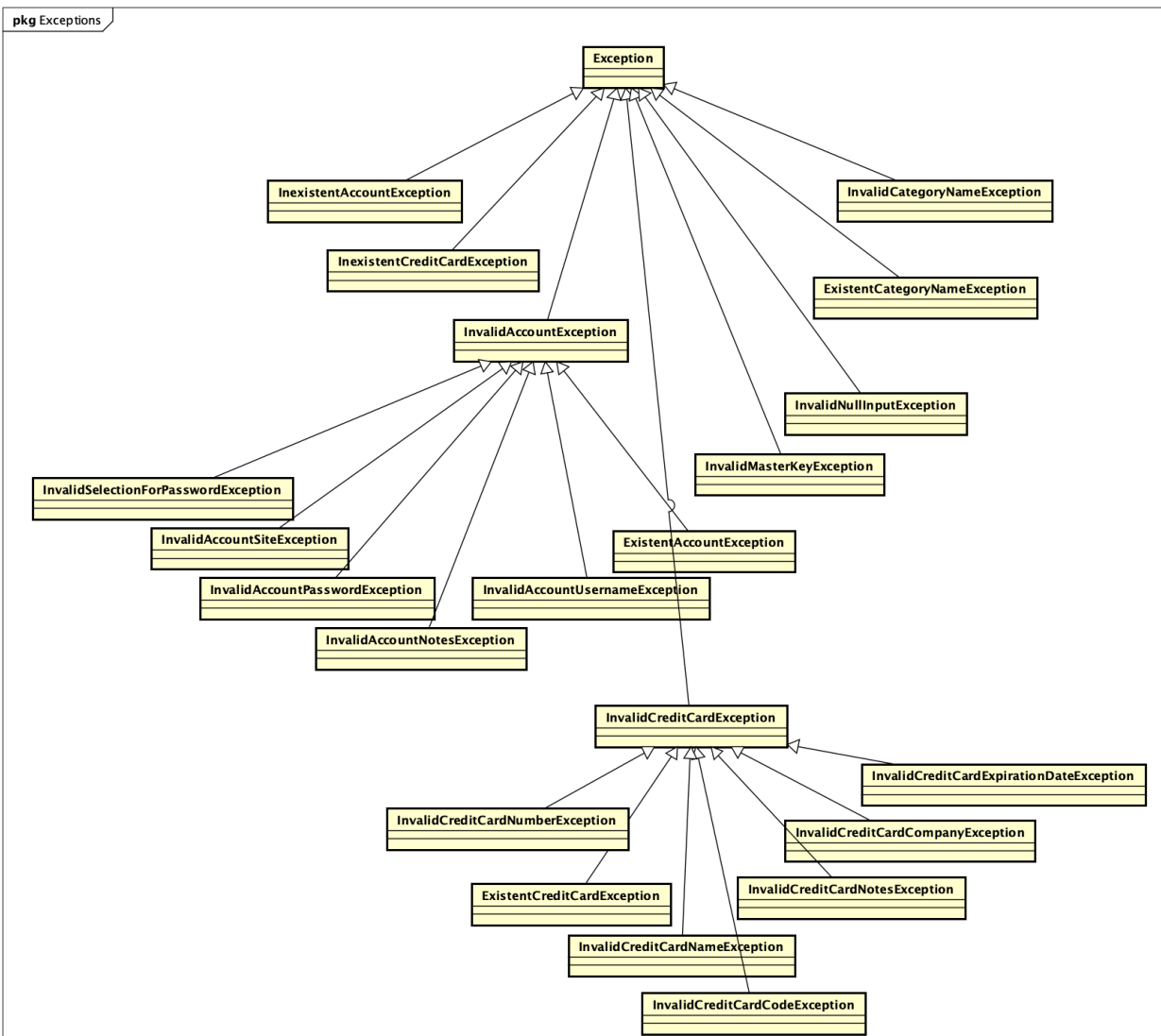


# Diagramas de clase

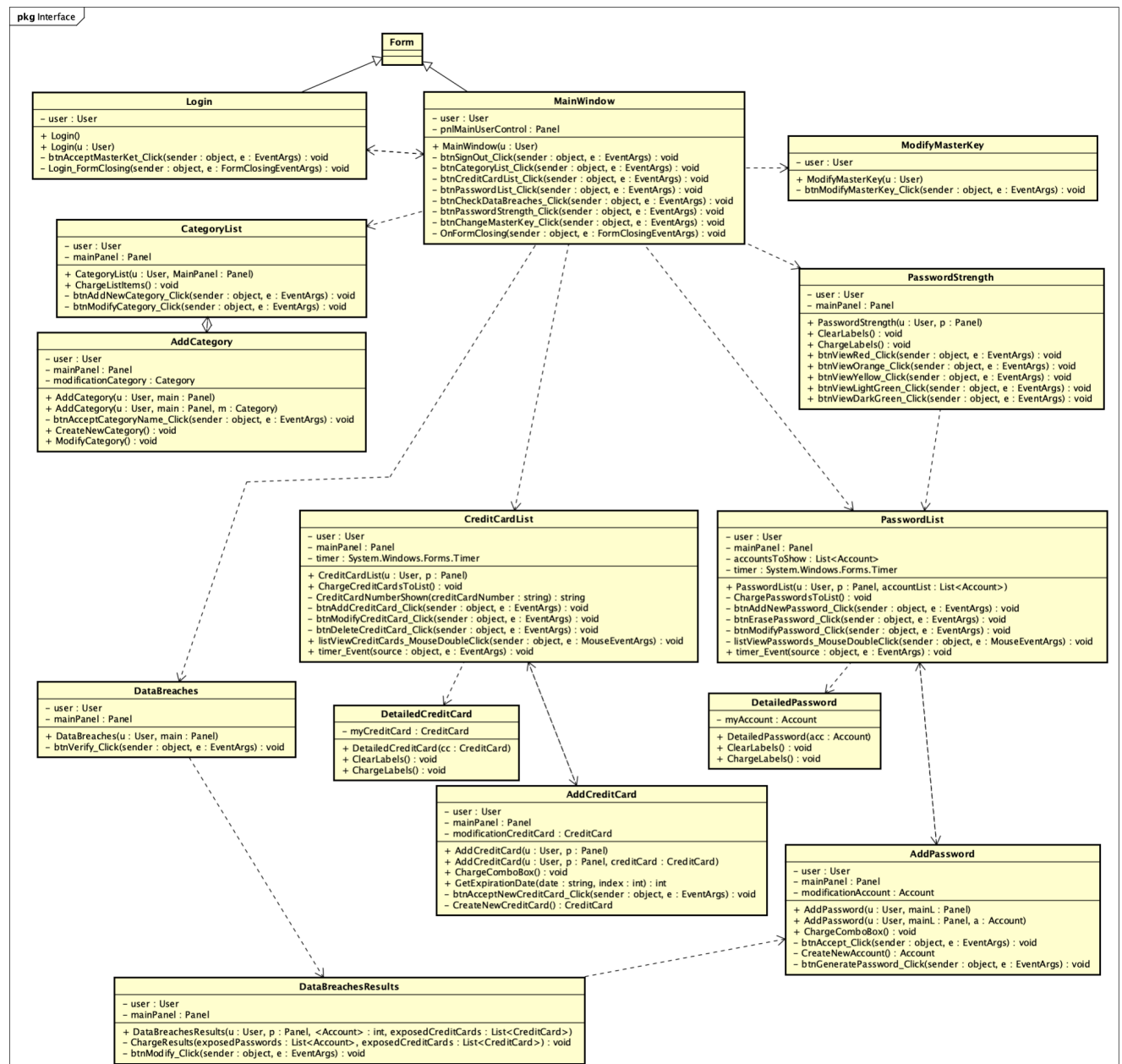
## PasswordManager



## PasswordManager.Exceptions



## Interface



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

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

### Uso de Git

Con respecto al uso y manejo de la herramienta git, se nos dificultó al principio agarrarle un poco la mano. Sin embargo, a medida que fue avanzando el obligatorio fuimos internalizando un poco más la herramienta e intentando utilizar git flow de la mejor manera.

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, una vez que creíamos que habíamos terminado de desarrollar la clase correspondiente a la rama en la que estábamos trabajando, hacíamos un último commit para asegurar que todo quedó registrado y un pull request con un merge a develop.

Cabe destacar que, en una primera instancia, si observamos con atención, creamos una rama llamada "feature" que no resulta del todo claro su uso. Esto sucedió por

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.

## Paquetes

Para el correcto desarrollo de nuestro Gestor, optamos por dividir el diseño en tres 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 una carpeta llamada "Exceptions", donde se encuentran las implementaciones de todas las excepciones utilizadas.
- **PasswordManagerTests**
  - Implementamos el Unit Testing realizado con MSTest para poder trabajar con TDD.
- **Interface**
  - Implementamos toda la interfaz de usuario utilizando Windows Forms.

## PasswordManagerTests (pruebas unitarias)

Obviaremos el desarrollo y la explicación de las clases de pruebas, puesto a que solo cabe aclarar que cada clase se corresponde con una de nuestras clases del dominio. En el caso de la clase User (que se mencionará más adelante) definimos dos clases de prueba, ya que en una de ellas inicializamos datos con TestInitialize y en la otra no. Decidimos tener datos inicializados también en la clase de pruebas de los Data Breaches, ya que el chequeo de los mismos debe darse a partir de una comparación con datos existentes. Todas las demás clases de prueba, exceptuando las dos mencionadas, se realizaron inicializando los datos necesarios para cada prueba en particular.

## PasswordManager (dominio)

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 sistema en general, donde manejar las listas de las instancias de las clases anteriormente mencionadas. (User)

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

Tal como mencionamos anteriormente, comenzamos con la creación de tres clases que nos resultaron "obvias": Account, CreditCard (para una tarjeta de crédito) y Category (para representar una categoría).

## Category

La clase **Category** representa cada categoría, con su respectivo nombre como atributo. Además, en ella se incluyen los métodos de validación y modificación de categorías. 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*. Ahora bien, la gran duda entonces es: ¿de qué tipo es esta clasificación?

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.

Por su parte, para lograr que las contraseñas se clasifiquen en el rango de colores mencionados, fue necesario implementar también ciertos métodos en Account, que se encargan de asignar un ColorClassification al atributo correspondiente de dicha clase, cada vez que se agrega o modifica una contraseña.

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, esta clase también cuenta con diversos métodos de validación requeridos para el sistema, como ser, el largo de la contraseña, el largo del usuario, entre otros.

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 diversos métodos de validación requeridos para la asignación de sus atributos. 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

## 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.

## User

Como mencionamos anteriormente, desde un principio resultó evidente que precisaríamos una clase que represente en general a nuestro sistema, donde guardar las listas de datos, las listas de categorías, poder implementar ciertas funcionalidades generales requeridas y por sobre todo, dar acceso a los datos únicamente a aquel que ingrese la clave maestra adecuada. A dicha clase le llamamos **User**. La misma tiene algunos atributos que son evidentes y otros que fueron necesarios a medida que implementamos el gestor. Estos son:

- La clave maestra de ingreso a la aplicación
- La lista de categorías registradas
- La lista de pares usuario-contraseña registrados
- La lista de tarjetas de crédito registrados
- 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.
- Un array de enteros llamado ColorCount, del cual hablaremos más adelante.

La clase User tiene diversos métodos respectivos a todas las funcionalidades del sistema. Entre ellos están los 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.

Por otro lado, esta clase reúne los métodos necesarios para el alta y modificación de categorías, así como también el alta, baja y modificación de tarjetas de crédito y contraseñas, tal como se especificó en los requerimientos.

Para implementar la funcionalidad de **reporte de fortalezas** de las contraseñas, hicimos uso de un atributo y un método en esta clase. Como mencionamos anteriormente, la clase User guarda un array de enteros llamado ColorCount. El mismo se actualiza cada vez que se agrega, se modifica o se elimina una contraseña del sistema. Básicamente, en dicho array se lleva registro de la cantidad de contraseñas que existen de cada color posible de clasificación. Pero, como bien sabemos, el reporte de fortalezas no consta solo de mostrar la cantidad de contraseñas que hay por color, sino que también debe brindar la opción de ver una lista de las contraseñas de cada uno de los colores desde donde poder modificarlas. Para lograr generar estas listas para cada color, creamos un método *FilterBy* que retorna la lista de cuentas filtrada para un solo color solicitado.

La clase User tiene también un método de generación de contraseñas. Es importante recalcar que el mismo podría haberse implementado de mejor manera, pero por cuestiones de tiempo no fue posible. Sin embargo, esperamos poder hacerlo para una segunda instancia.

Como comentario extra, se puede notar que, en diversas clases nos encontramos con varios métodos de validación que tienen características muy similares. Si bien no nos dio el tiempo de hacerlo, cabe destacar que nos hubiese gustado desarrollar una clase estática que reúna todos los métodos de validación.

## Data Breaches

Por último, 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 chequeado 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 ingresa en formato de strings. Por lo tanto, precisaríamos una manera de transformar esta información y adaptarla en una lista de strings, que sea apta para nuestro chequeo. Creamos una interfaz **IDataBreachesAdapter** que establezca un contrato común para poder adaptar cualquier tipo de entrada que se quiera, para que en un futuro, si se opta por ingresar el texto expuesto en otro formato, se precise simplemente hacer una nueva implementación de esta interfaz. Necesitamos entonces, crear una clase **PlainTextAdapter** que implementara dicha interfaz, para la adaptación de texto plano a lista de strings.

## Exceptions

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 credito) 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.

Entendemos también que han habido casos donde lo ideal hubiese sido crear una excepción cuando hay un error, pero que por temas de tiempo no hemos podido concretarlo.

## Interface (Interfaz de usuario)

Con respecto a las 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.

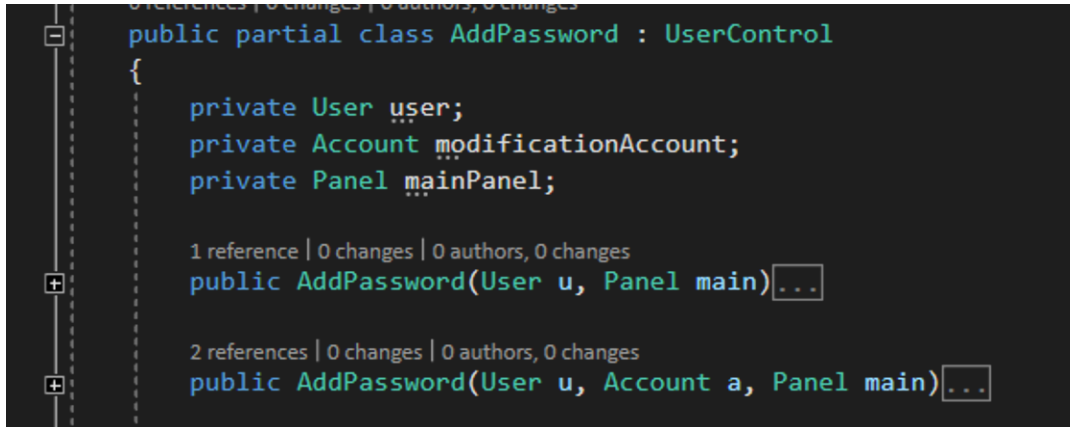
Estas dos ventanas mencionadas son las únicas que decidimos modelar como *forms* en nuestro proyecto. Todo el resto de las funcionalidades fueron creadas como clases que heredan de *UserControl*. De esta manera, la ventana principal contiene un panel, en el cual se agregan y van cambiando los distintos 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.

Ya que mencionamos el tema de los constructores, todas las clases (que heredan de Form y UserControl) de nuestra interfaz tienen un atributo privado con la instancia de la clase User correspondiente a la aplicación. Dicha instancia se inicializa al iniciar sesión por primera vez en la ventana de login, y se le pasa por parámetro a cada una de las ventanas que se crean.

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:



```

0 references | 0 changes | 0 authors, 0 changes
public partial class AddPassword : UserControl
{
    private User user;
    private Account modificationAccount;
    private Panel mainPanel;

    1 reference | 0 changes | 0 authors, 0 changes
    public AddPassword(User u, Panel main) ...
    2 references | 0 changes | 0 authors, 0 changes
    public AddPassword(User u, Account a, Panel main) ...

```

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 para un segundo obligatorio puede ser modificado tranquilamente.

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. 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.




## 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 95,29% , indicando un alto nivel de cobertura. Esto significa que solo un 4,71% de nuestro código no quedó cubierto por los tests.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
▸  iritrocki_IRINATROCKI5D88 2021-...	108	4.71%	2186	95.29%

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

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
▸  iritrocki_IRINATROCKI5D88 2021-...	108	4.71%	2186	95.29%
▸  passwordmanager.exe	52	6.58%	738	93.42%
▸ { } passwordManager	8	1.13%	698	98.87%
▸ { } passwordManager.Except...	44	52.38%	40	47.62%
▸  passwordmanagertest.dll	56	3.72%	1448	96.28%
▸ { } passwordManagerTest	56	3.72%	1448	96.28%

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 utilizando 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.

Si traducieramos este problema de las excepciones a porcentajes, podemos observar claramente que casi todas las excepciones de nuestro código tienen exactamente un 50% de cobertura, debido a que el método que retorna el mensaje de la excepción no fue testeado.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
iritrocki_IRINATROCKI5D88 2021-...	108	4.71%	2186	95.29%
passwordmanager.exe	52	6.58%	738	93.42%
{ } passwordManager	8	1.13%	698	98.87%
{ } passwordManager.Except...	44	52.38%	40	47.62%
ExistentAccountExce...	2	50.00%	2	50.00%
ExistentCategoryNam...	2	50.00%	2	50.00%
ExistentCreditCardExc...	2	50.00%	2	50.00%
InexistentAccountExc...	2	50.00%	2	50.00%
InexistentCreditCardE...	2	50.00%	2	50.00%
InvalidAccountExcept...	2	50.00%	2	50.00%
InvalidAccountNotes...	2	50.00%	2	50.00%
InvalidAccountPassw...	2	50.00%	2	50.00%
InvalidAccountSiteEx...	2	50.00%	2	50.00%
InvalidAccountUserna...	2	50.00%	2	50.00%
InvalidCreditCardCod...	2	50.00%	2	50.00%
InvalidCreditCardCo...	2	50.00%	2	50.00%
InvalidCreditCardExce...	2	50.00%	2	50.00%
InvalidCreditCardExpi...	2	50.00%	2	50.00%
InvalidCreditCardNa...	2	50.00%	2	50.00%
InvalidCreditCardNot...	2	50.00%	2	50.00%
InvalidCreditCardNu...	2	50.00%	2	50.00%
InvalidMasterKeyExce...	2	50.00%	2	50.00%
InvalidNullInputExce...	4	100.00%	0	0.00%
InvalidSelectionForPa...	2	50.00%	2	50.00%
invalidCategoryName...	2	50.00%	2	50.00%
passwordmanagertest.dll	56	3.72%	1448	96.28%

Concluimos que nuestro nivel de cobertura con las pruebas unitarias es bastante alto, y se vio perjudicado por el porcentaje de nuestras excepciones. 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.

Link a vídeo de youtube con pruebas

<https://youtu.be/I5IUTzCSwKs>