Obligatorio 1 - Diseño de aplicaciones

Gastón Landeira - 238473
Matías González - 219329
Iñaki Etchegaray - 241072

Profesor: Darío Alejandro Macchi Heins
de Tecnologías: Balduccio - Anduano

Universidad ORT Uruguay

ÍNDICE

El Proyecto

Vista General

La aplicación es un gestor de contraseñas y tarjetas de crédito con funcionalidades que permiten a varios usuarios entrar y poder compartir contraseñas, ver la seguridad de las mismas e incluso generarlas si lo desea. La idea es realizar una aplicación con código profesional por detrás, aplicando técnicas de código que aseguren su escalabilidad en el futuro.

La solución se compone de tres sub-proyectos:

- Domain, donde yace la lógica del proyecto.
- Domain Tests, donde se realizan los tests de la lógica del proyecto.
- User Interface, donde encontramos los componentes visuales de la aplicación.

Cada uno de estos proyectos tienen su estructura interna que será mencionada más detalladamente en el documento.

Antes de comenzar, una lista de los bugs conocidos del proyecto:

- Hay un bug conocido a la hora de modificar contraseñas breacheadas que en algunos casos muy específicos que no logramos detallar exactamente en lugar de modificar la contraseña se crea una nueva.

Notas generales del proyecto:

- Nuestro proyecto de clases de negocio posee un archivo Program.cs, que constituye la entrada del proyecto Domain. Quisimos quitarlo ya que la entrada verdadera es por el Program de User Interface, pero hacer esto no permitía que corriera el programa.

Clean Code

Aplicamos técnicas de Clean Code a nuestro código. Para asegurarnos la consistencia nos definimos una lista de Standards de Codificación:

Nombres

- Para atributos públicos usamos auto-properties o properties y comienzan con mayúsculas.
- Para atributos privados comienzan con un “_”.
- Para variables locales y parámetros comienzan con minúscula.
- Si la variable posee más de una palabra de nombre, usar camelCase.
- Uso abierto de constantes const o readonly.
- Las clases comienzan con mayúsculas.
- Para Interfaces comienzan con I.
- Utilización de Paquetes para funcionalidades que se encapsulan solas.

- No tenerle miedo a los nombres de métodos muy largos.
- Ser lo más descriptivos con los nombres de tests que se permita.

Formato:

- El largo de una línea no puede ser tan larga que no se pueda leer toda la función en una pantalla.
- Aceptar a lo sumo 3 parámetros, no más.
- Evitar los comentarios lo más posible, al menos que hayan cosas que aclarar.
- Atributos los más arriba de la clase posible.
- Funciones abajo en orden de más abstracta a menos abstracta.
- Auto-properties en una línea.
- Separación de un enter entre funciones.
- Indentar con 4 espacios o un tab entre brackets abiertos.
- El primer y último bracket deben tener sus propias líneas.
- Separar operadores con un solo espacio.
- El código debe de estar en Inglés.

Sufijos para los tipos de ventanas:

- NombreWindow para ventanas grandes.
- NombreModal para los modals.
- NombreController para los UserController.

Prefijos para los elementos:

- Boton: btnNombre
- Textbox: txtbxNombre
- Label: lblNombre
- Panel: pnlNombre
- DataGridView: grdvwNombre
- ComboBox: cmbbxNombre
- ListView: lstvwNombre
- Number: numNombre
- Checkbox: chkbxNombre
- FlowLayoutPanel: fwlytNombre
- Chart: chrtNombre

En el caso de la interfaz, no pudimos cumplir que los métodos comienzan con mayúscula ya que son auto-generados por Windows Forms.

GitHub y TDD

Para el versionado del proyecto utilizamos GitHub y nos basamos en el framework de trabajo Git Flow ya que se ajusta bien a la técnica de diseño TDD.

Nuestro repositorio tiene una rama principal main, la cual está destinada a versiones release del proyecto. De la rama main nosotros creamos una rama develop, en donde creamos distintos branches para realizar las features propuestas. Además, nos propusimos a nombrar las branches con nombres en secuencia de donde surgían. Por ejemplo, si alguien trabajaba en la interfaz de usuario para las contraseñas, esa branch tendría el nombre: “develop-UserInterface-Passwords”, esto hace que sea más fácil seguir de donde surge la branch y a donde tendria que mergear.

El Git Flow también nos ayudó a separar las responsabilidades de cada Usuario a distintos branches, para que puedan trabajar tranquilos sin preocuparse por conflictos constantes.

Utilizamos la técnica de diseño TDD para desarrollar el dominio. Es decir, nos ajustamos a realizar las clases de negocio creando primero sus pruebas unitarias, luego implementarlas en la lógica, y finalmente realizar el refactoring de código.

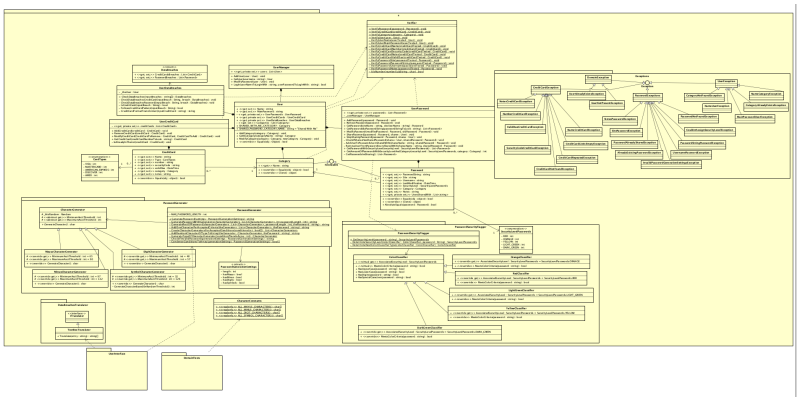
Durante el comienzo del proyecto se siguió este proceso. Pero, los commits eran pusheados luego de hacer varias pruebas e implementar numerosas partes del dominio. Esto causó que cuando volvimos a ver los commits para ver una prueba en específico, no logramos ver en secuencia la realización de estas pruebas e implementaciones. Luego de ver esto, decidimos hacer un commit por stage de TDD. Es decir, realizamos la prueba y dábamos commit, luego su implementación y commit, finalmente si se realizaba refactor un último commit para cerrar el ciclo. Esto mejoró mucho la documentación de los commits ya que proporcionaba más información sobre la secuencia de lo que se realizó.

Link al respositorio del proyecto:
https://github.com/ORT-DA1/238473_219329_241072/tree/main

Diseño

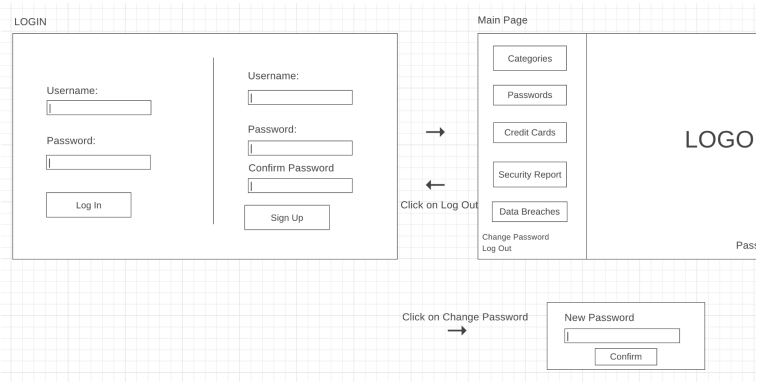
Para lograr comunicar bien el proyecto y mejorar la experiencia de desarrollo del mismo, lo primero que hicimos fue crear un diagrama UML de las clases de negocio. Por supuesto que este diagrama fue cambiando a lo largo del camino, pero nos definió las ideas fundamentales de cómo el proyecto iba a ser organizado por dentro.

Este diagrama UML cubre todas las clases de negocio y sus interacciones, y además las clases de Interfaz y su interacción con las clases de negocio.



El diagrama UML se encuentra en el proyecto junto a este PDF.

Además, decidimos también realizar un Wireframe de la interfaz del usuario para poder previamente decidir cómo sería la composición general de la aplicación visualmente antes de empezar a hacerla. Esto generó discusión entre los integrantes del grupo que, en caso de tener que deshacer algo, utilizando esta herramienta es tan fácil como borrar los componentes y reorganizarlos. Mucho más barato que hacerlo en el camino con código ya escrito.



Notar que el Wireframe puede tener algunas diferencias con el resultado final, ya que era un rough de la interfaz para asistir en el desarrollo. Pero las ideas principales se mantienen.

Link al wireframe completo:
https://wireframe.cc/pro/pp/336489a28439365

Soporte Multiusuario

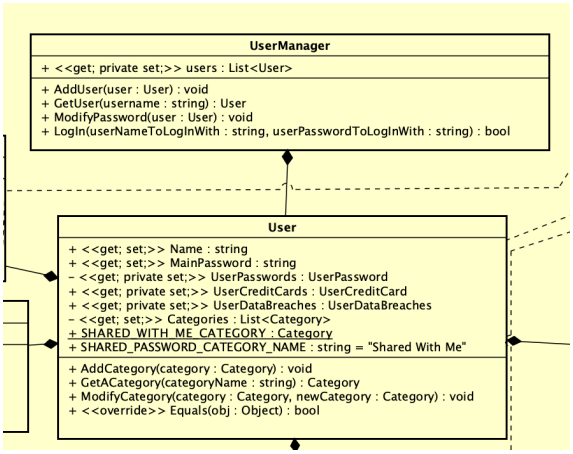
Al tener que diseñar el soporte multiusuario, identificamos que:

- La aplicación debe de tener una lista de usuarios registrados en memoria identificados por su nombre.
- Debe haber una noción de sesión, lo cual implica un log in y un log out.
- Implica la existencia de un perfil básico, con un nombre identificador y una contraseña. Esto implica un sign up y una edición de contraseña.

A base de estas necesidades definimos:

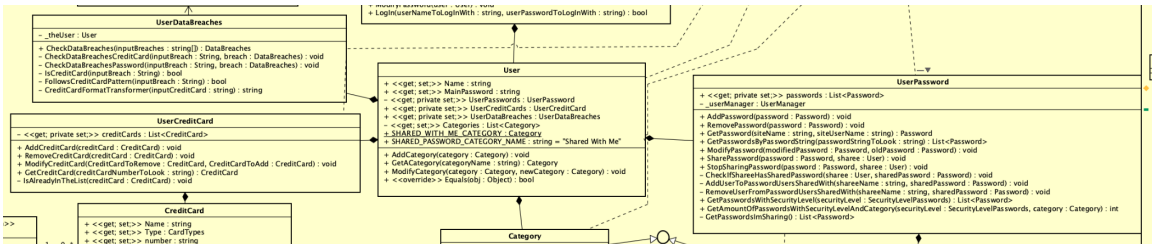
- La clase UserManager, cuya responsabilidad es contener a todos los usuarios existentes con su información, todo lo que implique acceso a los usuarios.
- La clase User la cual representa al usuario en la aplicación. Esta clase es el centro de toda la aplicación y el acceso a la mayoría de la funcionalidad.

Esto en el UML se manifiesta de la siguiente manera:



Como se ve en el UML, la presencia de sesión se da en el atributo LoggedUser de UserManager. Los métodos de UserManager están relacionados con obtener o modificar usuarios, el login y operaciones que requieren de una visión global de los usuarios registrados.

El User se encarga del registro de su categorías y el guardado de las mismas, también se va a encargar de todo lo relacionado a las contraseñas y tarjetas de crédito. Esto causó que la clase User fuera una clase “monolítica”. Es decir, era una clase con muchas responsabilidades y además muy larga para que sea fácil de leer. Para solucionar esto, dividimos las responsabilidades de la clase User en otras clases como se muestra en el siguiente diagrama: (UserPassword,UserCreditCard y UserDataBreaches)



Lo que hacen estas clases y sus responsabilidades estará cubierto más adelante en el documento.

La interfaz interactúa con UserManager de la siguiente forma:

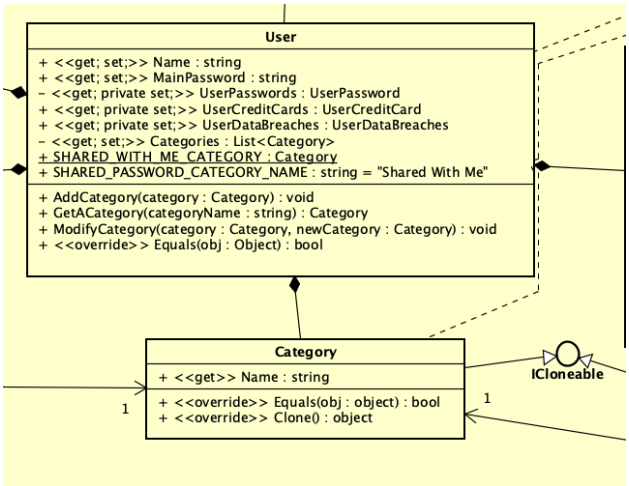
- Al darle al botón de Log In, corre la función Log In.
- Al hacer un sign up, accede a la función de agregar un usuario.
- Al tocar Change Password, accede a la función de cambio de contraseña de un usuario.

Alta, Modificación y Listado de Categorías

En práctica las Categorías son simplemente un string, ya que se compone solamente de un nombre.

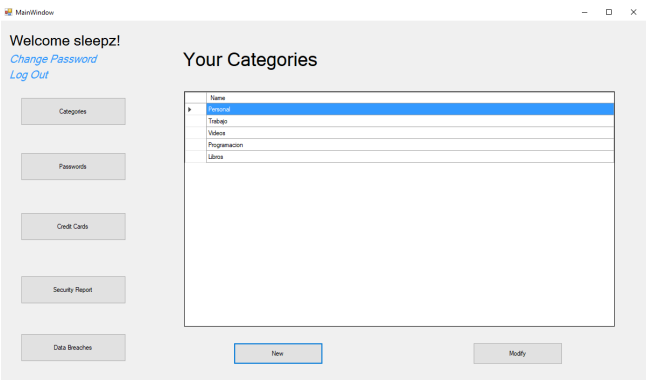
Sin embargo, nos parece que es un participante importante en el sistema y decidimos crear una clase Categoría que posee un nombre. Esto permite que la funcionalidad de Categoría sea mucho más escalable, ya que si llega a haber funcionalidad nueva ligada a Categoría, podemos agregarla en su clase.

Las categorías son un elemento presente en el listado de ambas tarjetas de créditos y las contraseñas. Por lo tanto nos pareció adecuado darle la responsabilidad a la clase User de guardar las categorías y de actuar sobre ellas como se ve en el diagrama:



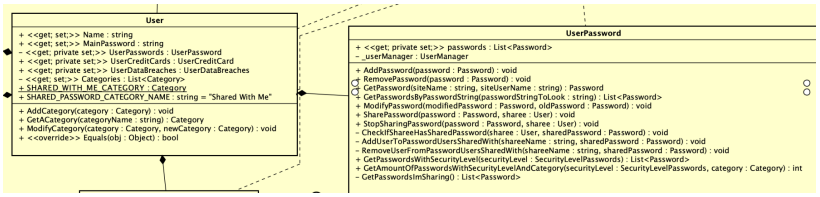
Además, las categorías son clonables al igual que passwords. Esto se debe a algunos momentos en el cual necesitábamos una copia sin referencia de las categorías para evitar cambios en la memoria no deseados.

Listamos las categorías con un DataGridView en Winforms que accede a la lista del usuario logueado. Los botones de Add y Modify interactúan directamente con las funciones de User respectivas.

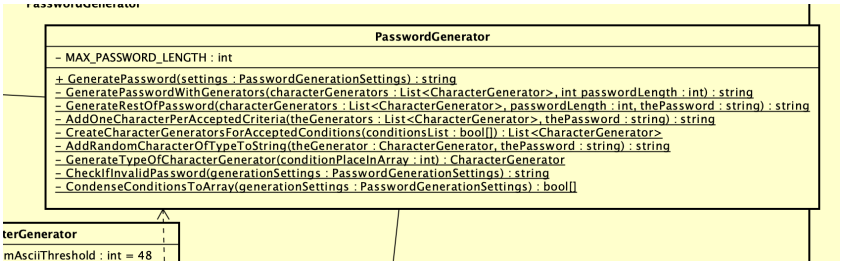


Alta, Baja, Modificación y Listado de Contraseñas

Para el manejo de sus contraseñas cada usuario tiene un atributo UserPasswords que guarda una lista con todas sus contraseñas y tiene acceso al UserManager. A su vez utilizamos una clase Password que almacena toda la información correspondiente a una contraseña(username, site, password, etc), entre ellos almacenamos una lista de strings con los nombres de los usuarios a los que la misma está siendo compartida en ese momento y un ENUM securityLevelPassword donde se guarda el color correspondiente al nivel de seguridad de esa contraseña. El acceso al UserManager desde UserPasswords se debe a que se necesita acceso a los demás usuarios para poder compartir/descompartir una contraseña.

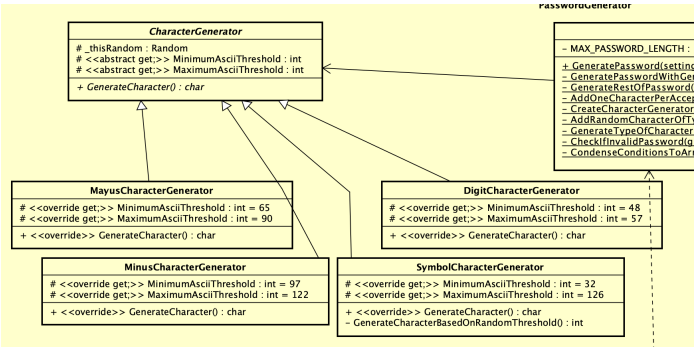


Para la generación de contraseña identificamos que tenía sentido tener una clase separada llamada PasswordGenerator que posee un único método público GeneratePassword, luego la UI podría utilizarla para generar la contraseña al apretar el botón de generar.



El generador nos trajo muchos desafíos de TDD y Clean Code. Los primeros problemas yacían en Clean Code. La cantidad de parámetros en un mismo método no respetaba la regla y, esto afectaba directamente a nuestro método GeneratePassword. Por esto, creamos un struct llamado PasswordGenerationSettings el cual se encarga de encapsular los settings posibles de generación de contraseñas, luego pasamos esto al método y logramos mantener Clean Code.

Seguido se nos presentó una posibilidad de polimorfismo, ya que teníamos presencia de switch statements y code smells. Para cada instancia de generar un char teníamos que generar alguno aleatorio de algún tipo (Mayus, Minus, Digit o Symbol), para eso se nos ocurrió iterar el “largo del string” cantidad de veces sobre un array generando un número aleatorio (que representa un lugar en el array) para decidir qué tipo de carácter generamos. Para ello, este array podría ser un array de CharacterGenerators, una clase de la cual heredan cuatro clases que implementan el método GenerateCharacter que nos devuelve un caracter del tipo necesitado:

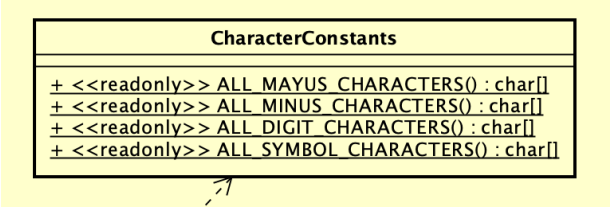


Luego tuvimos un problema con TDD: ¿Cómo probamos funciones con comportamiento randomico? Esto presenta un gran problema, ya que no podemos asegurarnos los mismos resultados dos veces para un test, algo que rompe con el concepto de una prueba Unitaria adecuada. Necesitábamos una manera de asegurarnos que un test no pueda dar bien en un caso y mal en otro.

Se nos ocurrieron dos maneras de resolver esto:

- Manipular el Random() en los tests.
- Que los Asserts se hagan sobre un conjunto exhaustivo de posibilidades.

Decidimos ir por la segunda por un tema de preferencia. Lo realizamos de la siguiente manera: creamos arrays readonly de chars que poseen todas las posibilidades para el tipo de carácter (const de strings habría funcionado perfectamente también).



Luego, en los Assert, nos fijamos que este char se encuentre en el array. Debido a que el array posee todos los caracteres posibles exhaustivamente (a responsabilidad del tester), si encontramos un error es, indudablemente, debido a la funcionalidad testeada y no al carácter randomico de la situación.

Una prueba para ver si genero un carácter mayús:

```
char characterGenerated = generator.GenerateCharacter();
Assert.IsTrue(
    CharacterConstants.ALL_MAYUS_CHARACTERS.Any(
        theCharacter => theCharacter == characterGenerated));
```

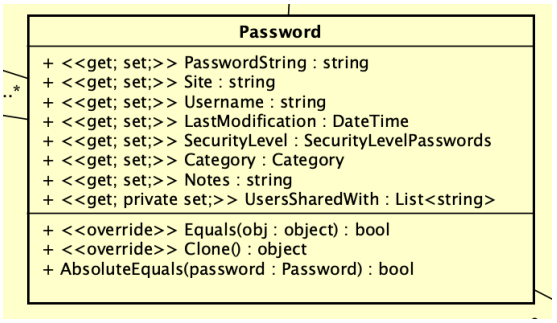
Por último queda la funcionalidad de compartir contraseñas. La discusión de implementar esta funcionalidad en una clase PasswordSharer se dio varias veces, pero al final decidimos que tenía más sentido que la funcionalidad se diera en el UserPassword por facilidad de acceso y debido a que la funcionalidad no escalaría más por fuera de UserPassword.

La funcionalidad se abarca en las funciones ligadas de UserPasswords y en una lista que agregamos en Password llamada UsersSharedWith. Esta lista posee los nombre de los

usuarios a los que esta contraseña está compartida. Esto da un fácil acceso a la información para su uso en la UI.

Hubo necesidad de hacer que Password implemente la interfaz de clonable ya que a la hora de compartir la contraseña a alguien se la agregamos a su lista con la categoría reservada “Shared With Me”, esto requería que no fuera la misma contraseña en memoria.

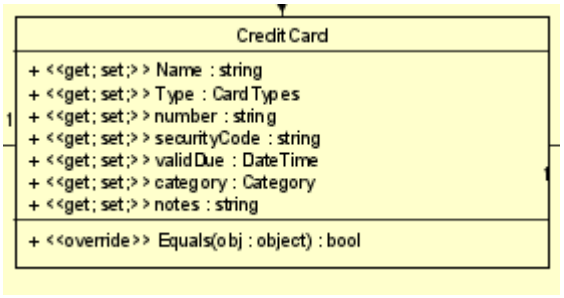
Luego los usuarios acceden a la funcionalidad mediante la selección de una contraseña en su listado y dándole a share, seleccionando al usuario que desea compartirla.



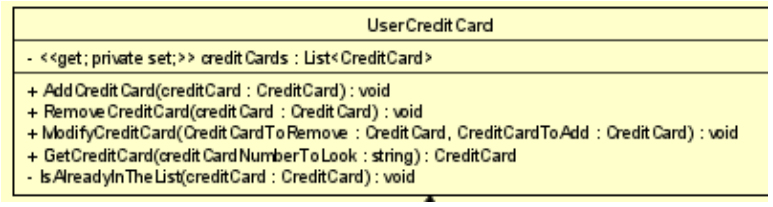
Alta, Baja, Modificación y Listado de Tarjetas de Crédito

Utilizamos la clase UserCreditCard para almacenar la lista de tarjetas de crédito(CreditCard) y sus métodos.

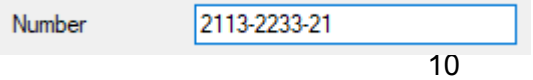
De lo más elemental a lo más general, los atributos de CreditCard son bastante estándar. Utilizamos string para los números(Number) y CVV(SecurityCode) porque, aunque sean números, en este contexto no tiene ningún sentido hacer cuentas con ellos. Para la fecha de vencimiento (ValidDue), si bien solo nos importa el mes y el año, manejamos DateTime. El beneficio fue que al no tener que crear métodos que limiten un string que representa una fecha, se ahorra tiempo y se gana en comodidad y claridad. Continuando, aplicamos un enum “CardType” para el tipo de tarjeta porque existe un número limitado de estos en el mundo real, así restringiendo posibles errores del usuario. Y el resto de los atributos es trivial.



Se explicara en conjunto a UserCreditCard y sus llamados desde la interfaz desde el punto de vista del usuario en el gestor. UserCreditCardController es la clase que se encarga de esta última. Al clickear en el botón “CreditCards” se despliega la lista en un dataGridView(una lista) que obtiene sus datos del get público de la lista de UserCreditCard. Solo se deja visible una parte de los datos almacenados de la tarjeta, se mencionara la funcionalidad de acceso a todos estos datos más adelante.



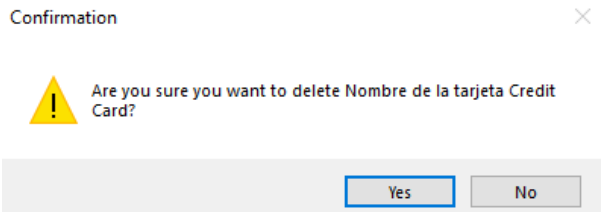
Para crear una nueva tarjeta se puede clickear en “New” que crea un modal llamado “NewOrModifyCreditCardModal” que despliega los input a llenar. Con respecto a los input, tanto Type como Category de la tarjeta se muestran con un combobox. El primero accede a la lista de “categories” del usuario mientras que el segundo al enum “CardTypes”. Además,



se crearon eventos y métodos para que el input de “numberCreditCard” sea interactivo al usuario mientras inserta los números, facilitando su uso. Lo último mencionable es que la fecha de caducidad de la tarjeta se recoge de un dataTimePicker, el cual su menú de selección fue removido y, está limitado al mes y año actual para prevenir confusiones del usuario y facilitar el uso. A la hora del formateo de datos usar componente permite guardarlos directamente en DateTime, formato que utilizamos.

Al presionar el botón “Confirm” se llama al método de UserCreditCard “AddCreditCard” que se encarga de añadirla a la lista. Se cierra el modal y se refrescan los datos del datagridview para que estén a la par del dominio. Aquí recién se habilitan los botones de borrar “Delete” y modificar “Modify” (si se elimina el elemento y se vuelve a tener una lista vacía, vuelven a deshabilitarse).

El botón delete se encarga de llamar al método “RemoveCreditCard” de UserCreditCard después de una confirmación del usuario (imagen). En caso afirmativo elimina la tarjeta de crédito del sistema y refresca la lista.



El botón modify importa los datos de la tarjeta de crédito seleccionada y se muestran en el mismo modal mencionado en “New”. De esta manera el usuario puede modificar cualquier dato. Al presionar confirmar, se llama al método “ModifyCreditCard”, que elimina la tarjeta anterior e inserta su modificación. También se cierra el modal y se refresca la lista.

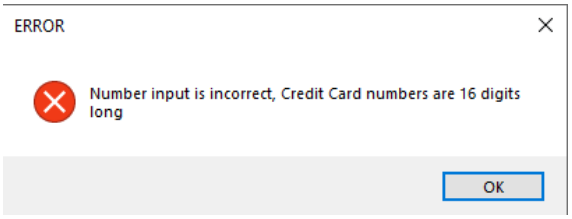
Es importante destacar que ambos métodos de añadir como modificar se hacen cargo de chequear si el número de la tarjeta es repetido, lo que no es admitido en el sistema.

La última funcionalidad de este segmento es la de obtener todos los datos de una tarjeta de crédito. Para esto el usuario solo debe hacer doble click en una celda del elemento deseado. De esta manera se crea el modal “CreditCardMoreInfoModal”. Este es muy similar al modal anterior con la diferencia que no se pueden modificar los datos (si se pueden copiar), además de que se le añade un timer a la ventana que permite su lectura durante treinta segundos. Al terminar este tiempo o al cerrar la ventana se efectúa lo segundo.

Se decidió en el equipo separar estas funcionalidades en dos modals para conseguir un código prolijo y no tener que enviar parámetros extras para poder distinguir su funcionalidad.

Los siguientes dos párrafos se aplican a todo el proyecto pero lo mencionaremos aquí.

Todos los modals y controllers muestran las excepciones del dominio, con su texto de error correspondiente, que pueden saltar al insertar o modificar. Una de las consecuencias de esto es no tener que controlar en la interfaz los campos de input. Por esta razón no dependemos de la interfaz, y se puede cambiar sin ningún problema. Ejemplo:



Todo esto es posible gracias a la clase Verifier. Se utiliza esta clase para chequear el input que vendrá de la interfaz y controlamos excepciones posibles antes de insertar en las listas.

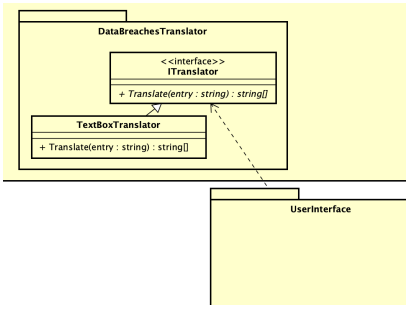
Detección de Data Breaches

Al pensar el diseño de los data breaches nos surgió un gran problema. Si el día de mañana llegara a cambiar el formato de entrada de los data breaches, el código presente debería de seguir funcionando sin tener que modificarlo.

Para ello, se nos ocurrió que la función que se encarga de detectar los data breaches (el que compara) esté separada de User en un UserDataBreaches que funciona similar a los UserPassword y UserCreditCards mencionados previamente. Este UserDataBreaches posee una función que toma un array de strings, el formato más genérico posible para esta funcionalidad.

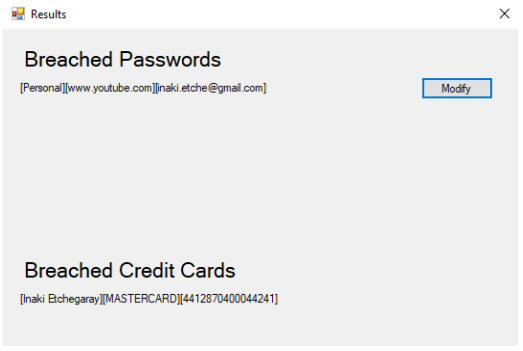
Luego habría una clase intermediaria que se encarga de tomar el formato entrante del texto y, de acuerdo a reglas establecidas, transforme esa entrada en una entrada aceptable para la función data breaches, es decir, un array de strings. Es como una especie de traductor para nuestro detector de datos breaches.

El día de mañana, basta con agregar una clase nueva que implemente las particularidades de traducción del formato nuevo y no habrá que cambiar nada de la clase encargada de los data breaches.



En este caso en concreto, nosotros tenemos una gran textbox en donde el usuario puede escribir contraseñas y tarjetas de crédito. Las limitaciones son que las tarjetas de crédito tienen que ser escritas con los espacios entre los numeros y los elementos separados por un enter. Al darle check nos muestra las coincidencias.

Luego la interfaz accede al traductor y le pasa este texto traducido al detector de data breaches para luego mostrarlo:



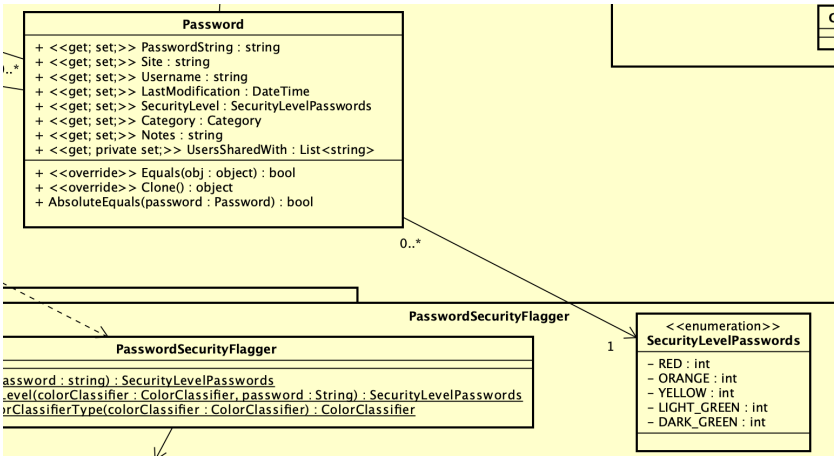
Reporte de Fortaleza de Contraseñas

Para el reporte de fortaleza de contraseñas se nos presentaron dos opciones de implementación:

- Calcular el nivel de seguridad cada vez que es necesario mostrarlo.
- Guardar el nivel de seguridad con la contraseña.

Optamos por la segunda, ya que nos parece un desperdicio de computo calcular el nivel de seguridad cada vez que se necesite. Además, nos pareció que es la opción más escalable para incorporar a la base de datos ya que podemos guardar esta clasificación.

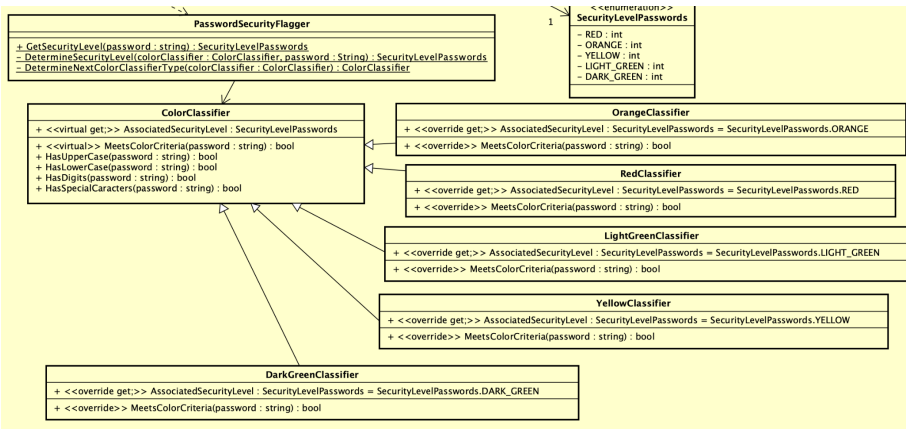
Para representar los niveles de seguridad, utilizamos un enum llamado SecurityLevelPasswords que encapsula a cada uno de los colores.



Luego tuvimos que pensar cómo vamos a realizar la clasificación de los strings. Para ello, identificamos que la funcionalidad podría abstraerse en un método estático que se encarga de todo. Pensamos en incluir esta funcionalidad en UserPassword o en Password, pero nos parece que le agregaba demasiada responsabilidad a las mismas, por lo tanto terminamos creando una clase PasswordSecurityFlagger que encapsula la funcionalidad principal al respecto.

Al implementar la funcionalidad de esta clase se nos presentó el problema de que teníamos muchas condiciones para los distintos casos. Muchas de las funciones que aprobaban estas condiciones podrán ser reutilizadas en los distintos casos de los colores. Realizando el código, quedaban muchas funciones parecidas repetidas y switch statements que rápidamente notamos que podrían ser reemplazados por un polimorfismo.

El polimorfismo implementado se da con una herencia de una clase ColorClassifier, la misma tiene las funciones de aprobación de condiciones las cuales sus hijos pueden utilizar para aprobar sus criterios. Luego cada hijo hace un override de la función MeetsColorCriteria en donde se define qué es lo necesario para que una contraseña se considere de su color:

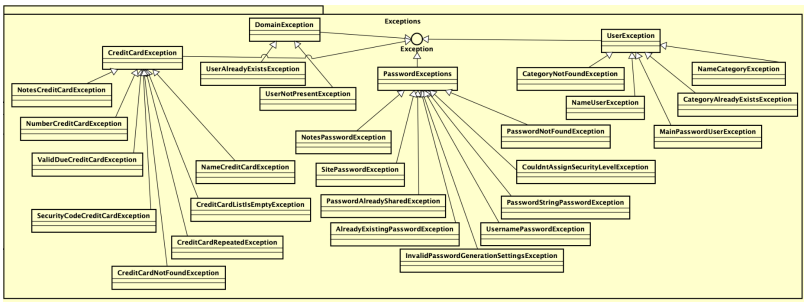


Luego el flagger puede hacer uso de ColorClassifier y mientras se asegure de que sea la subclase adecuada, tendrá su funcionalidad.

La UI nunca accede a las funcionalidades de PasswordSecurityFlagger ya que basta con recorrer la lista de contraseñas en busca de aquellas con el nivel de seguridad requerido. Esto se hace mediante UserPasswords que posee las funciones necesarias. El uso del flagger en si se da a la hora de modificar la contraseña o de agregarla. Accediendo a las funciones de GetPasswordsWithSecurityLevell y GetAmountOfPasswordsWithSecurityLevelAndCategory podemos armar la información necesaria para la UI y la gráfica:

Manejo de Excepciones

Nuestra aplicación implementa una gran cantidad de excepciones personalizadas, todas heredan de Exception:



Además, las tenemos divididas en excepciones por funcionalidad. Por ejemplo, CreditCardException son excepciones lanzadas en las funcionalidades relacionadas a tarjetas de crédito.

Luego la interfaz es la que atrapa las excepciones lanzadas por el domain y muestra el mensaje pertinente al usuario:

```
try
{
    DialogResult dialogResultDeleteCreditCard = MessageBox.Show("Are you sure you want to delete " +
                                                _selectedCreditCard.Name + " Credit Card?", "Confirmation",
                                                    MessageBoxButtons.YesNo, MessageBoxIcon.Warning);
    if (dialogResultDeleteCreditCard == DialogResult.Yes)
    {
        _currentUser.UserCreditCards.RemoveCreditCard(_selectedCreditCard);
        CreditCardLoad();
    }
}
catch (CreditCardException creditCardException)
{
    MessageBox.Show(creditCardException.Message, "ERROR",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Testing

Como fue mencionado al principio del documento, la funcionalidad de la aplicación fue realizada en su totalidad con la técnica de desarrollo TDD. En la misma aplicamos los conceptos de las tres fases de TDD y desarrollamos los tests antes de hacer la funcionalidad.

El testing resultó en una totalidad de 177 tests distintos, divididos en las distintas funcionalidades de la aplicación. Todos los tests dan verdes y poseen la siguiente cobertura de código:

obligatorioda1.exe	167	14,25 %	1005	85,75 %
{ } Domain	38	4,87 %	743	95,13 %
{ } Domain.DataBreachesTra...	0	0,00 %	4	100,00 %
{ } Domain.Exceptions	117	69,23 %	52	30,77 %
{ } Domain.PasswordGenera...	5	4,46 %	107	95,54 %
{ } Domain.PasswordSecurit...	7	6,60 %	99	93,40 %

La cobertura del ejecutable da en 85.75% debido a que una gran parte del código no testeado es de excepciones, como se puede ver en la imagen (30.77% de cobertura para Domain.Exceptions) todos los paquetes relevantes se encuentran con una cobertura por encima del 93%.

Luego, realizamos videos del testeo funcional del alta, baja y listado de contraseñas, para ver los videos acceder a los siguientes links:

https://www.youtube.com/watch?v=AVOspNODjGM

También se realizó un video testeando toda la app, el cual no era necesario pero lo incorporamos al documento de todas maneras:

https://www.youtube.com/watch?v=UuMX1ayaA8E