

Universidad ORT Uruguay  
Facultad de Ingeniería

# Diseño de aplicaciones 1

## Obligatorio 1

Entregado como requisito para la obtención del título de Ingeniero en  
Sistemas

Sofía Decuadra - 233397  
Juan Pablo Rodriguez Sotto - 241896  
Fernanda Secinaro - 245650

Tutores: Nicolás Fornaro, Pablo Benítez y Martín Ganon

**2021**

# Índice

<b>Índice</b>	<b>2</b>
<b>Introducción</b>	<b>4</b>
1.1 Herramientas utilizadas	4
1.2 Instalación	4
1.3 Limitaciones del sistema	5
<b>Repositorio en Github</b>	<b>6</b>
<b>Descripción general</b>	<b>7</b>
3.2 Flujo general de la aplicación	7
<b>Diseño</b>	<b>8</b>
4.1 Introducción a las clases del dominio	10
4.2 Generación de contraseñas	11
4.3 Reporte de fortaleza de una contraseña	11
4.4 Reporte de Fortaleza de todas las contraseñas del usuario	11
4.5 Reporte de Fortaleza por categoría	12
4.6 Data Breaches	12
4.6.2 Implementación de nuevos formatos	13
4.6.1 Histórico de Data Breaches	13
4.7 Sobre el código	14
4.7.1 Clean Code	14
4.7.2 TDD	14
4.7.3 Porque utilizar TDD facilitó aplicar los principios de Clean Code	15
4.7.4 TDD y el diseño adaptativo	16
4.7.5 Cobertura del código	17
4.7.6 Manejo de excepciones	17
4.8 Interfaz de Usuario	19
4.8.1 Reutilización de componentes en la interfaz de usuario	19
4.9 Encriptación de contraseñas	19
4.9.1 Algoritmo RSA	19
4.9.2 Implementación de .NET del algoritmo RSA	20
4.9.2 Clase "Encrypter"	20
4.10 Entity Framework	21
4.10.1 Code First y Fluent API	21
4.10.2 Ejemplo de uso de Fluent API: Encriptación de la contraseña maestra	21
4.10.3 Manejo de excepciones de la base de datos	22
4.10.4 Patrón Repositorio	22
4.10.5 Patrón Singleton	23
<b>Anexo 1 - Requerimientos Funcionales</b>	<b>24</b>
<b>Anexo 2 - Definiciones</b>	<b>28</b>
<b>Anexo 3 - Ejemplos de la Interfaz Gráfica</b>	<b>29</b>

<b>Anexo 4 - Diagrama de Paquetes</b>	<b>34</b>
<b>Anexo 5 - Diagrama de Clases</b>	<b>35</b>
Diagrama de la clase Encrypter	35
<b>Anexo 6 - Pruebas funcionales</b>	<b>36</b>
Video 1:	36
Video 2:	37
Video 3:	38
<b>Anexo 7 - Diagramas de secuencia</b>	<b>39</b>
Agregar contraseña:	39
Borrar tarjeta de crédito:	39
Modificar categoría:	40
<b>Anexo 8 - Modelo de tablas</b>	<b>41</b>

# 1.Introducción

El presente informe tiene como objetivo exponer, de manera clara y detallada, el trabajo realizado y la toma de decisiones de diseño al implementar una aplicación para el gestionamiento de datos.

## 1.1 Herramientas utilizadas

Para el desarrollo de este sistema se utilizó el lenguaje C# y el framework .NET en su versión 4.8. El dominio fue hecho como una librería de clases y fue construido utilizando la metodología TDD para la cual fue necesario utilizar MSTest, un framework para pruebas unitarias. Posteriormente, la interfaz de usuario fue hecha utilizando Windows Forms. La persistencia de datos fue hecha en una base de datos SQLServer de la cual se hizo uso a través de Entity Framework en su versión 6.0.

Para el versionado del proyecto, se utilizó un repositorio Git el cual fue subido a Github dentro de la organización ORT-DA1. Para trabajar con Git, seguimos la metodología Git Flow en la cual tenemos la rama Main, una rama de desarrollo Develop, ramas feature-\* y ramas release-\*.

## 1.2 Instalación

Para utilizar la aplicación, se puede hacer mediante el ejecutable "PasswordsManagerUserInterface.exe". Antes de esto, es necesario cierta configuración previa. Se debe tener instalado y corriendo una instancia de SQLServer a la cual deberá referirse mediante un connection string que deberá ser especificado en el archivo "PasswordsManagerUserInterface.exe.config" bajo la sección "connectionStrings". El nombre de la instancia de SQLServer se registra en "DataSource" mientras que el nombre de la base de datos a utilizar se coloca en "Catalog". Si se desea cargar los datos de prueba incluidos, se podrá hacer uso del archivo "Base de datos/Base con datos de prueba/DataManager.bak" dentro de la carpeta "Base De Datos" para restaurar la base de datos. Para cargar una base de datos vacía con solo las tablas, utilizar el archivo "Base de datos/Base vacía/DataManager.bak "

Para acceder a los datos de los usuarios que están cargados en la base de datos de prueba, se debe tener en cuenta la siguiente información:

<b>Username</b>	<b>Master Password</b>
andrea	andrea
cecilia	cecilia
pedro	pedro
ismael	ismael
henry	henry

### **1.3 Limitaciones del sistema**

En esta segunda etapa del obligatorio, los datos son persistidos en una base de datos. Aunque esto soluciona el problema de persistir los datos, surge otra limitación por parte de la encriptación de las contraseñas. Será explicado más adelante pero el equipo utilizó un algoritmo de encriptación asimétrica que hace uso de claves públicas y privadas las cuales para esta entrega fueron persistidas en la base de datos. Idealmente se debería encontrar otra solución para almacenar la clave privada por temas de seguridad.

## **2.Repositorio en Github**

[ORT-DA1/Decuadra-RodriguezSotto-Secinaro \(github.com\)](https://github.com/ORT-DA1/Decuadra-RodriguezSotto-Secinaro)

### 3.Descripción general

El proyecto consiste en una sencilla aplicación que permite a los usuarios el guardado de sus usuarios, con las respectivas contraseñas, para varios sitios y/o aplicaciones y la información de tarjetas de crédito.

Para su implementación, se utilizó de referencia el libro “Clean Code” y la metodología de desarrollo TDD.

#### 3.2 Flujo general de la aplicación

Al iniciar la aplicación, el usuario deberá ingresar su nombre de usuario y contraseña maestra, de no contar con uno previamente deberá crearlo<sup>1</sup>.

Una vez dentro del sistema, el usuario podrá elegir entre varias opciones como por ejemplo el “Passwords”. Elegir esta opción, lo llevará al menú de contraseñas donde podrá ver las suyas y aquellas que le compartieron así como realizar todas las funcionalidades referentes al manejo de las mismas, estas incluyen, por ejemplo, agregar, modificar, borrar, compartir o dejar de compartir <sup>2</sup>. Al agregar o modificar una contraseña, el usuario tendrá disponible para completar todos los campos que ellas requieren<sup>3</sup> y para el campo “Password” en particular, el usuario podrá elegir generar una contraseña aleatoria mediante los parámetros que la interfaz de usuario le permite elegir<sup>4</sup>. Para borrar una contraseña, basta con hacer click en la fila correspondiente a la misma y después hacer click en “Delete”.

Otras opciones del menú principal incluyen el manejo de tarjetas de crédito, con un flujo similar al de las contraseñas pero sin poder compartirlas, el reporte de contraseñas y la funcionalidad de comprobar que los datos del usuario no hayan sido comprometidos en un data breach<sup>5</sup>.

Finalmente, en el header del menú principal, el usuario tiene dos opciones, o bien puede modificar su contraseña maestra o puede hacer Logout.

---

<sup>1</sup> [RF16](#)

<sup>2</sup> [RF03 a 05 y 10 y 11](#)

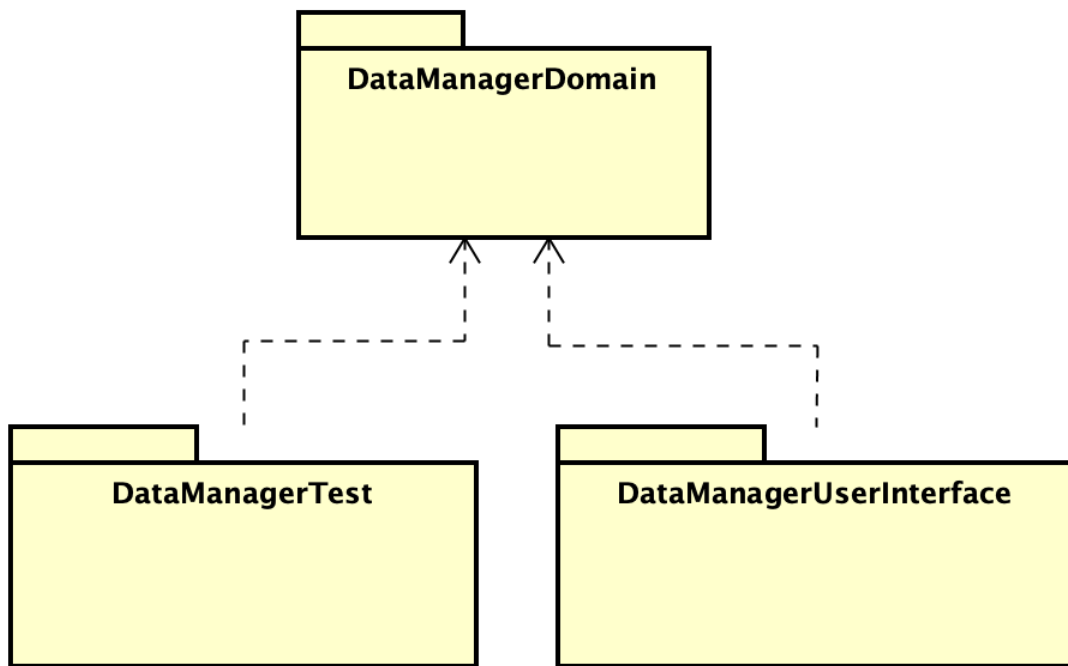
<sup>3</sup> Categoría, nombre en el sitio, nombre del sitio, contraseña y notas

<sup>4</sup> [RF06](#)

<sup>5</sup> [RF07 a 09 y 12. 17 y 18](#)

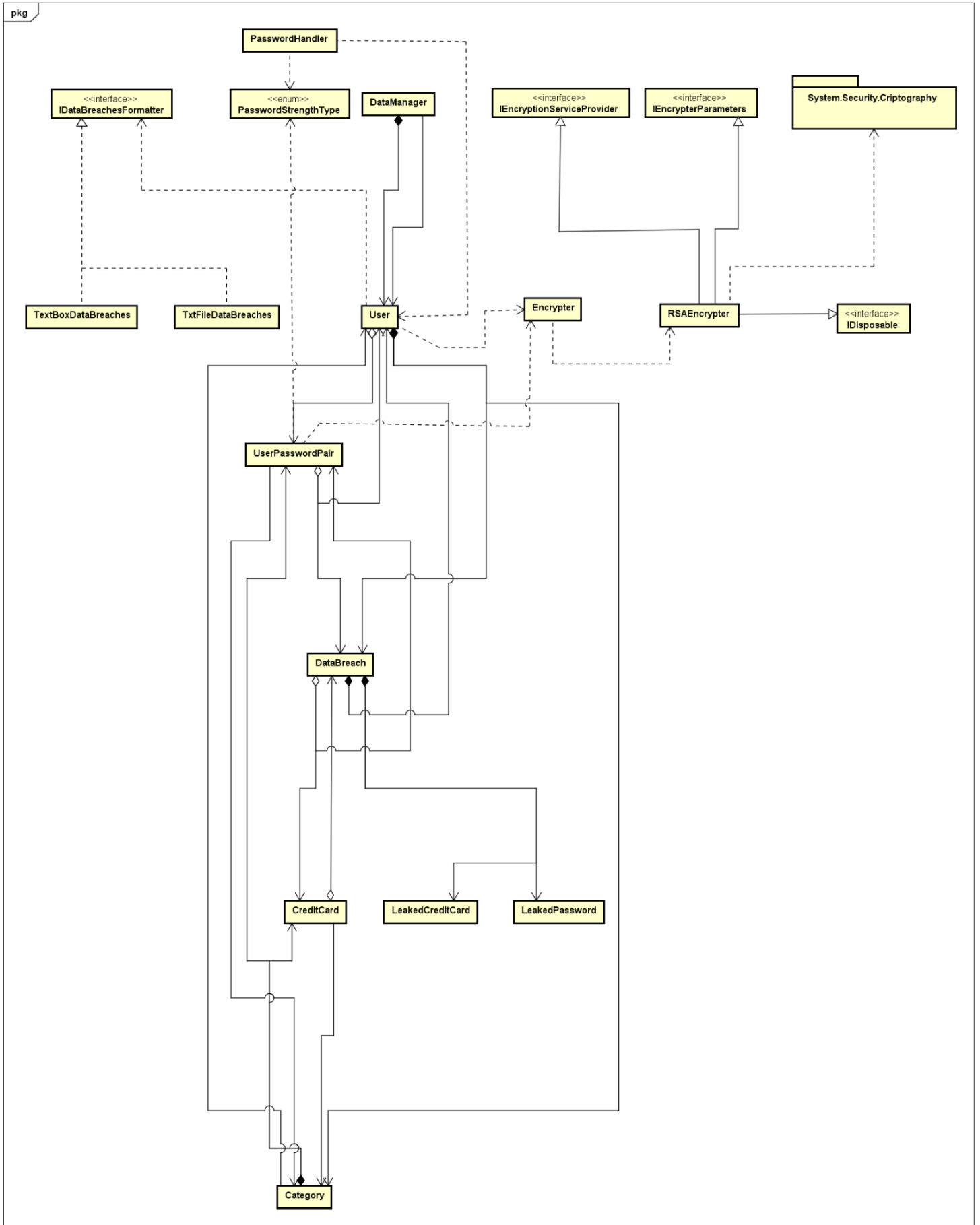
## 4. Diseño

Dentro de lo que es el desarrollo de esta aplicación, se pueden observar 3 paquetes. Los primeros dos son el dominio y sus respectivas pruebas y el tercero corresponde a la interfaz de usuario. El paquete “DataManagerDomain” contiene la librería de clases del dominio y de él dependen los otros dos paquetes, “DataManagerTest” y “DataManagerUserInterface”.



Para explicar las clases del dominio y porque existe cada una, se mostrará el diagrama completo y acto seguido se procederá a desglosarlo en componentes más pequeños que permitan observar su funcionamiento.

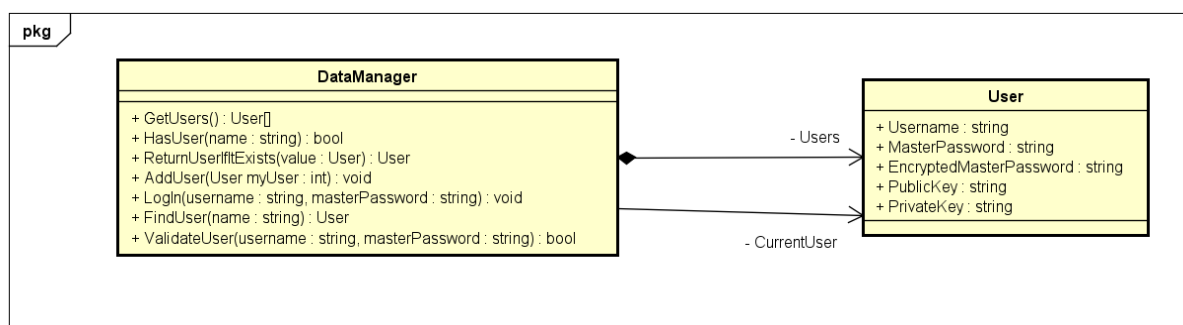




## 4.1 Introducción a las clases del dominio

Para implementar la aplicación, detectamos primeramente la necesidad de una clase “Data Manager” la cual tiene el rol de “sistema” y se encarga de llevar el registro de todos los usuarios de la aplicación así como de saber quien es el usuario que está haciendo uso de la aplicación actualmente.

Esto nos lleva a la necesidad de tener una clase “User”<sup>6</sup> para representar a cada usuario y este, a su vez, tiene la responsabilidad de mantener un registro de sus credenciales para identificarse, nombre y contraseña maestra<sup>7</sup>, así como una composición de categorías<sup>8</sup> que este tiene creadas y una lista de contraseñas que fueron compartidas con él.



Para cada categoría, creamos la clase “Category” la cual es utilizada para almacenar las contraseñas y tarjetas de crédito del usuario..

En lo referente a la representación de las contraseñas en el sistema, creamos una clase “UserPasswordPair” que se encarga de almacenar todos los datos relacionados a las contraseñas y se asegura de que los datos a guardar sean todos válidos. Es en esta clase donde se almacenan los usuarios que tienen acceso a dicha contraseña. Por lo tanto, aquí existe una composición ya que cada instancia de “UserPasswordPair” solo puede existir en una categoría del propio usuario pero cada usuario es capaz de tener varias contraseñas que le hayan compartido otros usuarios por lo tanto también tendría una relación de agregación con “User”.

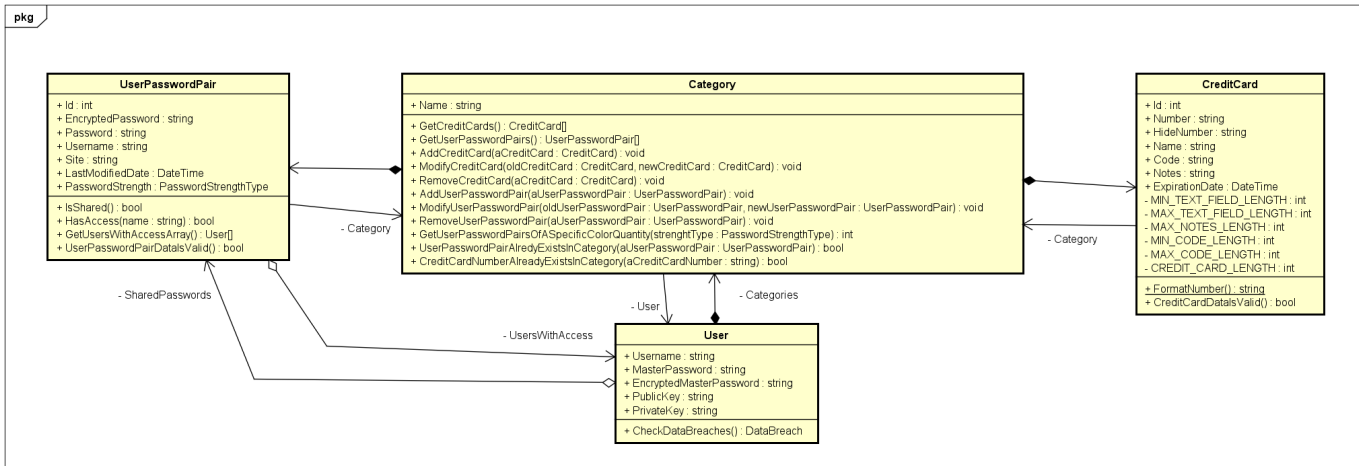
Para las tarjetas de crédito ocurre algo similar con la clase llamada “CreditCard” la cual almacena los datos de una tarjeta de crédito y se asegura de que sean válidos. En contraste con “UserPasswordPair”, esta clase se

<sup>6</sup> En relación de composición con “DataManager” ya que un usuario no puede existir por fuera de una instancia de “DataManager”

<sup>7</sup> “Name” y “Master Password”

<sup>8</sup> Una categoría debe siempre pertenecer a un usuario y solo uno

encuentra en una relación de composición con “Category” ya que solo puede existir dentro de una categoría y esta no puede ser compartida entre usuarios.



## 4.2 Generación de contraseñas

La lógica responsable de generar las contraseñas aleatorias se encuentra dentro de la clase “PasswordHandler<sup>9</sup>” y esta utiliza y depende de una estructura de datos auxiliar llamada “PasswordGenerationConditions<sup>10</sup>” como forma de recibir los parámetros en base a las cuales va a operar el algoritmo.

## 4.3 Reporte de fortaleza de una contraseña

Al igual que el método mencionado anteriormente, este método también se encuentra en la clase “PasswordHandler” y opera devolviendo un enum llamado “PasswordStrengthType” que contiene los 5 colores posibles de fortaleza de contraseña.<sup>11</sup>

## 4.4 Reporte de Fortaleza de todas las contraseñas del usuario

Cada vez que el usuario agrega una nueva contraseña, esta es analizada en base a su fortaleza y el valor resultante es guardado dentro de UserPasswordPair. Al solicitar el reporte de fortaleza, utilizando el método “GetPasswordsStrengthReport”, lo que uno obtiene es una lista de tuplas en

<sup>9</sup> Esta es una clase que contiene únicamente métodos para trabajar con strings de contraseñas.

<sup>10</sup> Los parámetros que guarda son el largo de la contraseña a generar y si debe tener alguna de las siguientes: Mayúsculas, minúsculas, números y/o símbolos.

<sup>11</sup> Red, Orange, Yellow, Light Green, Dark Green.

la cual se encuentran el color y la cantidad de contraseñas de ese color que posee el usuario.

También es posible conseguir todas las contraseñas de un color en específico mediante el método “GetUserPasswordPairsOfASpecificColor”. Este método recibe un valor de “PasswordStrengthType” y retorna todas las contraseñas del usuario que pertenecen a ese color.

## 4.5 Reporte de Fortaleza por categoría

Es posible obtener cuantas contraseñas hay, de cada color, separándolas por categoría. Esto es posible recorriendo las contraseñas del usuario y agrupándolas por “Category” y dentro de esa agrupación, separándolas por color.

## 4.6 Data Breaches

Cada instancia de “User” posee un método “CheckDataBreaches” el cual, dado una instancia de la interfaz “IDataBreachesFormatter”<sup>12</sup>, retorna una tupla conteniendo dos listas, una de contraseñas filtradas y otra con las tarjetas de crédito filtradas.

La interfaz “IDataBreachesFormatter” fue diseñada con el objetivo de utilizar el polimorfismo para poder ejecutar el método “CheckDataBreaches” independientemente de donde proviniera la información. Esto se logra haciendo que todas las clases que implementan dicha interfaz, deban tener un método “ConvertToArray” el cual devuelve la información en el formato necesario<sup>13</sup> para ser procesada por el algoritmo.

Para el obligatorio 1, el método en el que se cargaban los Data Breaches era mediante un input en el cual el usuario podrá ingresar las credenciales filtradas separadas por un salto de línea<sup>14</sup>. Con el objetivo de procesar la información dada es que se creó la clase “TextBoxDataBreaches” la cual implementa la interfaz “IDataBreachesFormatter”. Su versión de “ConvertToArray” toma el texto y realiza un split por el carácter mencionado.

Para este obligatorio, se sumó otro formato al previamente implementado en el que se lee la información de las contraseñas o tarjetas de crédito filtradas

---

<sup>12</sup> Relación de dependencia ya que esta solo se utiliza como parámetro del método

<sup>13</sup> Un array en el cual en cada posición se encuentra un dato, que puede ser una contraseña o un número de tarjeta de crédito, que fue filtrado.

<sup>14</sup> “\n”

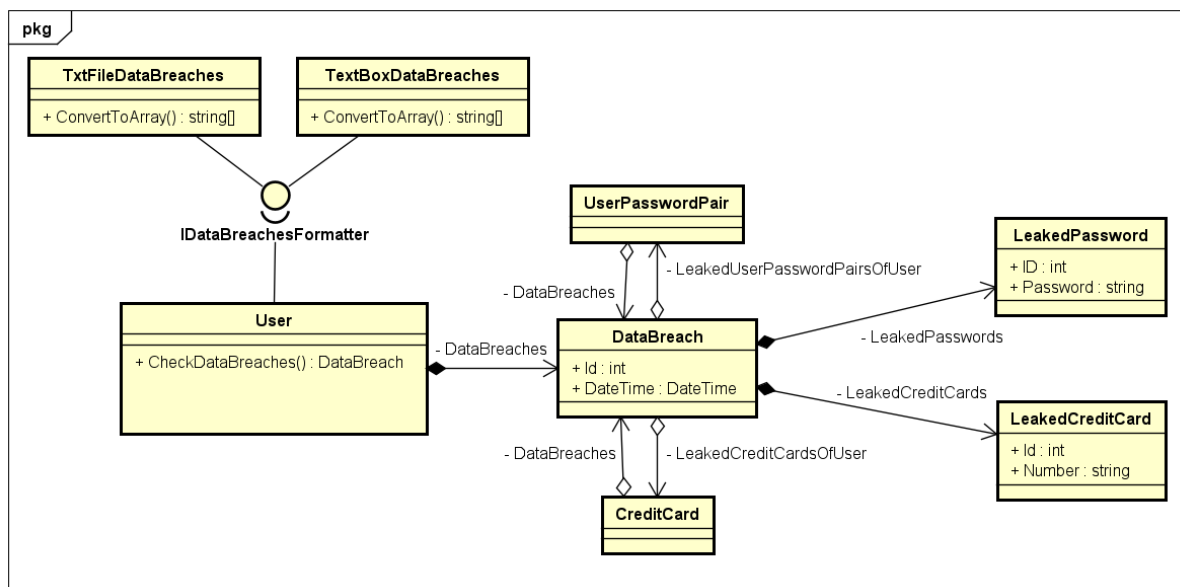
mediante un archivo de texto. Para este fin es que se implementó la clase “TxtFileDataBreaches” que a su vez implementa la interfaz previamente mencionada y su método “ConvertToArray” se limita a hacer un “split” en cada caracter “tab”<sup>15</sup>.

## 4.6.2 Implementación de nuevos formatos

Como reflexión sobre el nuevo formato introducido en el obligatorio 2, el equipo no tuvo problemas para su implementación. El haber utilizado la interfaz para el obligatorio 1 hizo que agregar otro formato más no supusiera problema alguno ya que se tomó ventaja del polimorfismo dentro de la función que chequea los Data Breaches. Se puede decir que la solución cumplió con el Open-Closed Principle ya que para agregar funcionalidad solo fue necesario agregar código pero no modificar el ya existente.

### 4.6.1 Histórico de Data Breaches

Como evolución del obligatorio 1, para esta entrega fue necesario contar con un historial de los data breaches para poder ser consultado en un futuro<sup>16</sup>. Es con este fin que se creó la clase “DataBreach” la cual guarda la fecha en la que fue creada así como el usuario que la creó y las credenciales que fueron expuestas.



<sup>15</sup> “\t”

<sup>16</sup> Por ejemplo, para saber si una contraseña fue expuesta a la hora de agregarla.

## 4.7 Sobre el código

Para mantener un estándar de calidad aceptable<sup>17</sup>, se utilizó la metodología TDD para desarrollar la aplicación. Además, se siguió la escuela de pensamiento de Robert C. Martin sobre lo que es “Código Limpio” tal cual fue descrito en su libro “Clean Code”.

### 4.7.1 Clean Code

“Clean code” es un libro escrito por Robert C. Martin y publicado en el año 2008. En él se detallan varias de las que el autor considera son buenas prácticas de desarrollo de software y provee lineamientos para todos, o casi todos los aspectos del proceso de escritura del código. El objetivo de esta escuela de pensamiento es, de forma resumida, asegurar la calidad del código escrito de forma tal de que mejore la mantenibilidad.

Para buscar lograr el propósito con el que fue escrito el libro, se ha tomado como referencia las reglas descritas en él para el diseño del código. Ejemplos de esto pueden ser observados a lo largo de todo el código, desde los nombres que revelan la intención o el tener como máximo dos niveles de indentación hasta cumplir con el “Single Responsibility Principle”<sup>18</sup> mediante refactorings y la delegación de responsabilidades a otras clases. Esto último encapsula también otro de los principios “SOLID”<sup>19</sup>, el “Open Closed Principle”<sup>20</sup>.

### 4.7.2 TDD

TDD significa “Test Driven Development” y consiste en desarrollar el código en base a pruebas. La idea fundamental es seguir un ciclo de tres fases<sup>21</sup> en los cuales no se escribe código de dominio si no es para que una prueba pase y no se escriben pruebas si no es para que algo del dominio falle.<sup>22</sup>

---

<sup>17</sup> Porcentaje de cobertura alto (mayor al 90%) y código “limpio”

<sup>18</sup> Esto quiere decir que cada función debería de cumplir con solo una “responsabilidad” y lo que no le compete debería estar en otra parte. Robert C. Martin lo describió con la frase “Gather together the things that change for the same reasons. Separate those things that change for different reasons”

<sup>19</sup> Acrónimo introducido por Robert C. Martin que detalla principios del paradigma OO que de ser seguidos, conducen al buen diseño de clases y métodos.

<sup>20</sup> Establece que una clase debería ser abierta para la extensión pero cerrada a la modificación

<sup>21</sup> Red, Green y Refactor

<sup>22</sup> Para una definición más exhaustiva de TDD, ver [anexo 2](#)

### 4.7.3 Porque utilizar TDD facilitó aplicar los principios de Clean Code

Seguir las reglas de TDD hace que los métodos resultantes sean testeables, ya que se diseñan para pasar pruebas. Tener métodos con estas características resulta en que estos no poseen ninguna lógica sobrante<sup>23</sup> y por lo tanto no se produce sobre-ingeniería. Esto ayuda a que no hayan efectos secundarios<sup>24</sup> y se siga el principio de única responsabilidad establecido en Clean Code.

A lo anterior le sumamos que el ciclo de TDD tiene como tercer etapa el refactoring. Al final del desarrollo de cada funcionalidad, nos tomamos un tiempo para realizar todos los refactorings que pudiéramos observar. Los refactorings más frecuentes que el equipo realizó fueron los que hacen énfasis en el principio mencionado anteriormente, esto desencadenó también en que se redujeron los niveles de anidación ya que cosas como for-loops y condicionales fueron extraídas a otros métodos

El ya mencionado “no escribir código si no es para que pase una prueba” llevó también a que cada clase tuviera una responsabilidad más acorde a su objetivo. Como resultado de TDD, ninguna clase tenía un atributo que no fuera utilizado y en general tampoco utiliza elementos que no se puedan conseguir de dentro de la misma clase. Esto fue hecho así en primer lugar para asegurarse que las funciones no reciban muchos parámetros, y la mayoría recibe uno o ninguno. En segundo lugar se buscó disminuir la cantidad de llamadas en cadena, también conocidas en Clean Code como “Train Wrecks”, tratando de que una clase solo llame a properties o métodos de los objetos que tiene como atributos y que lo que suceda entre medio para obtener el resultado sea responsabilidad de la clase que si tiene los atributos necesarios.

En el libro, Robert C. Martin habla de que el mejor comentario es aquel que no se hace. TDD es de gran ayuda para esto ya que logramos documentar cada clase y su comportamiento en base a las pruebas. El uso de la metodología hizo además, como ya fue mencionado anteriormente, que las funciones no tuvieran efectos secundarios y los nombres de ellas y sus variables sean reveladores lo que redujo aún más la necesidad de comentarios.

---

<sup>23</sup> Tienen la lógica mínima para cumplir su cometido

<sup>24</sup> Funcionalidad extra, no esperada al ver la firma del método

#### 4.7.4 TDD y el diseño adaptativo

El diseño adaptativo consiste de realizar un diseño original de como se espera a grandes rasgos que sea la aplicación y después permitir que este vaya evolucionando a lo largo del proceso de desarrollo como respuesta a varios factores.<sup>25</sup>

La metodología TDD es un claro ejemplo del diseño adaptativo ya que comenzamos con un conjunto de funcionalidades que queremos tener y la implementación de dichas funcionalidades se va armando a medida que progresa el desarrollo de la aplicación. Muchos de estos cambios sucedieron en las etapas de refactoring en donde se observó que una clase estaba asumiendo una responsabilidad que no le correspondía y fue necesario modificarla. Un ejemplo fue pasar las validaciones de los campos de contraseñas y tarjetas de crédito a las clases “UserPasswordPair” y “CreditCard” respectivamente en vez de dejarlas en “Category” como lo estaban inicialmente.

Originalmente el equipo optó por utilizar listas para todas las colecciones de objetos a lo largo del sistema pero conforme avanzó el proyecto y la frecuencia y uso de cada clase se fue haciendo más evidente, la elección de estructuras de datos para cada caso fue cambiando. Para guardar los usuarios, las contraseñas y las tarjetas se decidió utilizar un Dictionary el cual utiliza generics y nos permitió un tiempo de acceso y agregado<sup>26</sup> constante junto con la eliminación de la necesidad de hacer casteos<sup>27</sup> lo que consideramos que era lo ideal ya que esas eran las operaciones más frecuentes de dichas clases. Por otra parte, las categorías tienen como su operación más frecuente el listado, en específico el listado alfabético<sup>28</sup> por lo que un Dictionary no tenía ningún beneficio tangible. En contraste, una lista ordenada<sup>29</sup> nos dejaría listar las categorías en orden  $O(n)$  y al insertar igual tenemos un orden  $O(\log(n))$  promedio por lo que era una solución adecuada para nuestro caso de uso.

Esto fue alterado nuevamente al comenzar a utilizar Entity Framework, cuyo uso será explicado más adelante. Esto causó que dejara de ser posible

---

<sup>25</sup> Cambio de requerimientos, necesidad de hacer refactoring y mejorar la eficiencia entre otros

<sup>26</sup> En lo referente a las contraseñas y tarjetas de crédito, las funcionalidades más frecuentes que un usuario va a utilizar es guardarlas y ver una que ya guardó

<sup>27</sup> Este era un posible problema que nos encontramos cuando quisimos utilizar un hash previo a elegir usar los diccionarios

<sup>28</sup> Para cumplir con [RF13](#), es necesario mostrar las contraseñas ordenadas alfabéticamente por categoría y esto se logra mediante la obtención de las categorías ordenadas y de ahí las contraseñas de cada categoría. Lo mismo para [RF15](#)

<sup>29</sup> En C# la implementación sería SortedList<>



utilizar estructuras de datos como un Dictionary o una Hashtable y pasó a ser necesario hacer uso de estructuras que implementen la interfaz ICollection puesto que Entity Framework las puede mapear automáticamente.

Otra forma en la que se aplicó diseño adaptativo como consecuencia de TDD y que resultó en aplicar principios de Clean Code fueron los métodos para obtener información, los getters. Al crear las pruebas, se pudo observar que no eran necesarias para cada cosa todas las funcionalidades de las interfaces originales en las que los datos estaban guardados por lo que en los getters, donde fue apropiado y creando getters particulares cuando fue necesario se retornó una interfaz distinta a la original que provee un conjunto de funcionalidades más limitada que la interfaz original y era más “feliz” a la hora de manipular los datos de forma segura y de ocultar la implementación de las clases. Un ejemplo de esto es la función “GetCategories” que retorna un array con las categorías ordenadas alfabéticamente, esto impide que el usuario pueda modificar la lista donde se guardan las categorías pero además le impide saber que las categorías se guardan en una base de datos en primer lugar. Con este enfoque, el día de mañana es posible cambiar la forma de guardar categorías e igual asegurar, quizás con más cálculos de por medio<sup>30</sup> que el método “GetCategories” siga retornando un array de categorías ordenado alfabéticamente.

#### **4.7.5 Cobertura del código**

Para asegurar que no quedara ningún método sin probar, el equipo logró tener un 100% de cobertura en todos los módulos y funciones del dominio. Esto no asegura que no hay errores, ya que no existe ninguna prueba perfecta, y que no surja ningún error simplemente significa que no se encontró pero sí genera un mayor nivel de seguridad a la hora de implementar funcionalidades nuevas o de hacer refactoring y ante cualquier cambio, uno puede correr las pruebas y saber que el 100% del código está funcionando como lo hacía anteriormente.

#### **4.7.6 Manejo de excepciones**

Para controlar los casos inválidos a lo largo del desarrollo del proyecto, se hizo uso de excepciones personalizadas. Ellas fueron originalmente construidas con los tres constructores ya que Microsoft considera esto una

---

<sup>30</sup> Originalmente se utilizó una lista ordenada pero después se pasó a usar Entity Framework y las categorías eran traídas de la base de datos y ordenadas mediante el OrderBy de Linq

buena práctica pero eventualmente optamos por dejar, y por consiguiente utilizar solo el constructor con mensaje. El grupo tomó esta decisión porque los demás constructores no fueron usados nunca y lo único que hacían era bajar el porcentaje de cobertura ya que eran código abandonado.

## 4.8 Interfaz de Usuario

La interfaz de usuario para esta aplicación fue hecha utilizando Windows Forms, esto es el formato de interfaz gráfica provisto por .NET Framework. En términos del flujo de la misma, el equipo se centró en ser lo más fiel posible a los bosquejos provistos en la letra del obligatorio sin perjuicio de que en algunos casos fue necesario desviarse por cuestiones de diseño.<sup>31</sup>

### 4.8.1 Reutilización de componentes en la interfaz de usuario

Algunas de las funcionalidades de la aplicación son muy parecidas y por lo tanto la interfaz de usuario resultante es muy similar. Como forma de no repetir código sin necesidad fue que el equipo optó por hacer reutilizables esos componentes que se repetían. El claro ejemplo es la función de agregar y modificar contraseñas, categorías o tarjetas para las cuales lo único que cambia entre pantallas son las acciones de aceptar y volver atrás. Para volver estos componentes reutilizables, se creó un Form de cada una<sup>32</sup> en la cual se cargaban los datos en caso de modificarse o se extraían en caso de agregar una nueva. Los botones externos al form se colocaron en una ventana particular a cada funcionalidad.

## 4.9 Encriptación de contraseñas

Para encriptar las contraseñas del gestor<sup>33</sup>, utilizamos el algoritmo de RSA el cual está implementado dentro del namespace System.Security.Cryptography.

### 4.9.1 Algoritmo RSA

El algoritmo RSA es un algoritmo de criptografía asimétrica, esto quiere decir que funciona en base a dos claves, una pública y una privada. La clave pública es la que se utiliza para encriptar los mensajes y puede ser compartida con todos. Posteriormente, los mensajes pueden ser descryptados por el destinatario mediante el uso de la clave privada la cual debe ser mantenida en secreto. Se cree que mientras la clave privada sea secreta, la encriptación del algoritmo es segura. Esto radica en que el algoritmo se basa en la utilización de números primos grandes y actualmente

---

<sup>31</sup> Referirse al [anexo 3](#) para ver ejemplos de la interfaz de usuario

<sup>32</sup> De Contraseña, Tarjeta de Crédito y Categoría independientemente de si es para agregar o modificar

<sup>33</sup> Contraseñas maestras y contraseñas normales (UserPasswordPair)

no se tiene suficiente poder de cómputo como para poder factorizar los resultados de la operación del algoritmo.

### **4.9.2 Implementación de .NET del algoritmo RSA**

Como fue mencionado anteriormente, el namespace `System.Security.Cryptography` contiene una implementación del algoritmo. El eje central es la clase “`RSACryptoServiceProvider`” la cual se encarga de realizar la encriptación asimétrica utilizando RSA. Esta clase a su vez nos sirve para generar las claves pública y privada para cada usuario en forma de un objeto del tipo “`RSAPParameters`” el cual puede ser serializado como XML para ser luego utilizado por otra instancia de `RSACryptoServiceProvider` durante la encriptación o desencriptación.

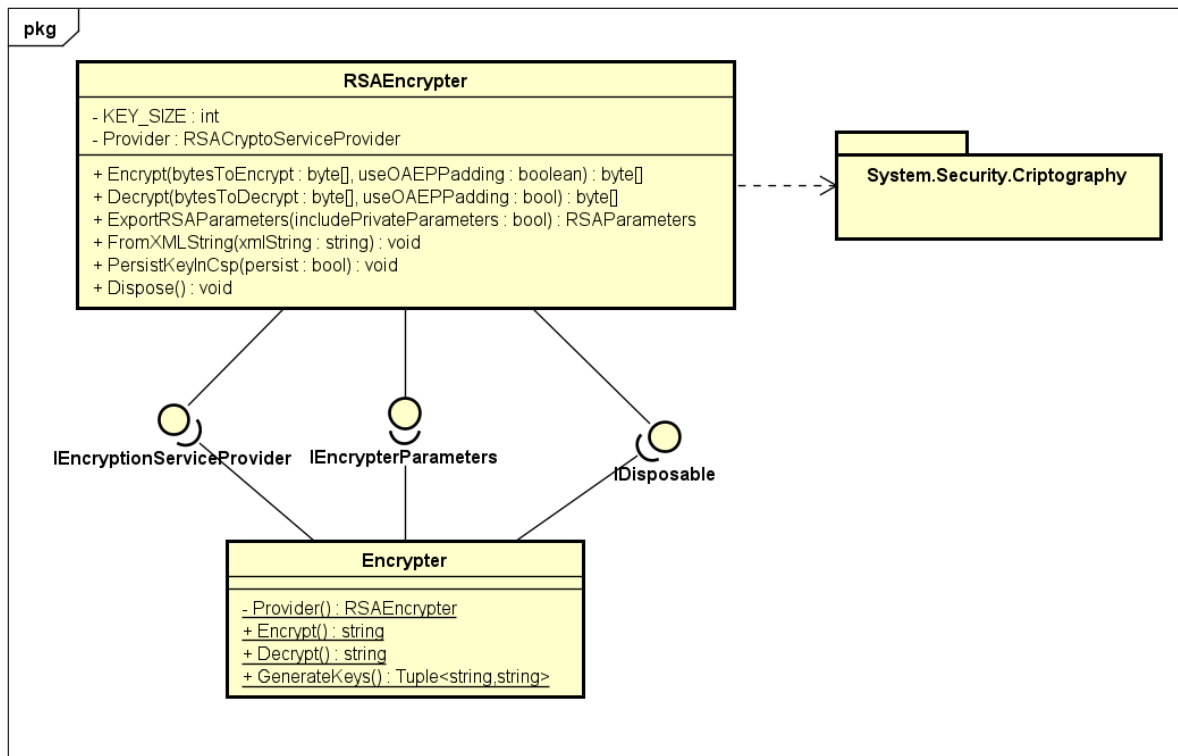
### **4.9.2 Clase “Encrypter”**

Para evitar el acoplamiento con la librería de encriptación utilizada, se hizo uso de una clase intermedia que fue llamada `Encrypter`. Esta posee los métodos públicos básicos de `Encrypt`, `Decrypt` y `GenerateKeys`. Esto ocasiona que de haber algún cambio en la librería utilizada, solo deba ser alterada la clase `Encrypter`. Para tratar de que dicha clase siga el Open-Closed Principle lo más posible, `Encrypter` posee un método “`Provider`” que retorna un objeto que debe implementar las interfaces “`IEncryptionParameters`”, “`IEncryptionServiceProvider`” y “`IDisposable`”. La primera de ellas se encarga de generar los parámetros específicos para el algoritmo de encriptación y con la segunda se añade los métodos utilizados para encriptar. La última de las 3 interfaces se utiliza para poder hacer uso del “`using`” ya que la clase que se ate a dicho contexto debe ser “`Disposable`”<sup>34</sup>. De esta forma, para implementar otro algoritmo de encriptación asimétrico, solo es necesario cambiar el retorno de `Provider()` a otro encriptador distinto de “`RSAEncrypter`”<sup>35</sup>. Mientras este implemente las interfaces mencionadas, `Encrypter` seguirá funcionando correctamente.

---

<sup>34</sup> El diagrama con los métodos específicos de cada interfaz se encuentra en el [Anexo 5](#)

<sup>35</sup> Esta es la clase que implementa los métodos de RSA y hace uso de las interfaces mencionadas anteriormente.



## 4.10 Entity Framework

La persistencia para el obligatorio 2 fue lograda utilizando Entity Framework. Este Framework es un ORM que permite mapear los objetos de nuestro dominio a tablas en la base de datos facilitando el manejo de la persistencia.

### 4.10.1 Code First y Fluent API

Entity Framework puede ser utilizado en base a 3 enfoques, Model First, Database First y Code First. Debido a la naturaleza del obligatorio, utilizamos Code First ya que el dominio de la aplicación estaba creado y debíamos generar una base de datos que se ajustara a él. Es por esto que utilizamos Fluent API para lograr la especificación más profunda de los modelos cuando las convenciones de Entity Framework no eran suficientes. Otra manera de haber logrado lo mismo hubiese sido utilizando Data Annotations pero esto hubiese significado un acoplamiento mayor del dominio con la persistencia.

### 4.10.2 Ejemplo de uso de Fluent API: Encriptación de la contraseña maestra

Un ejemplo práctico de en donde se hizo uso de Fluent API es a la hora de guardar las contraseñas encriptadas en la base de datos. Originalmente,

estas se encriptaban y desencriptaban en los get y set de la property “MasterPassword” pero esto llevó a que la encriptación no se viera reflejada en la base. Es por esto que se utilizó Fluent API para que Entity Framework ignorara la property “MasterPassword” y en su lugar guardara otra llamada “EncryptedMasterPassword” la cual era una property simple. Con esto, la property “MasterPassword” siguió funcionando correctamente en memoria por lo que no fue necesario hacer ningún cambio al resto del código pero a su vez se cumplió el requisito de persistir la contraseña encriptada.

Una solución similar fue empleada a la hora de persistir las contraseñas encriptadas de la clase “UserPasswordPair”.

### **4.10.3 Manejo de excepciones de la base de datos**

Al utilizar Entity Framework, introducimos un nuevo nivel donde el sistema puede fallar: la base de datos. Cada operación que se realiza a la base de datos puede generar un error de la misma y esto resulta en una excepción que debe ser controlada para que no se caiga la aplicación. Es por esto que se debería utilizar una función anterior a los “using” que se encargue de verificar que funcione correctamente. Para implementar esta función, lo más práctico es utilizar el patrón repositorio, descrito después, o de lo contrario se estaría repitiendo mucho código y se violaría Clean Code.

Al haber trabajado en memoria para el primer obligatorio, muchas de las verificaciones previas a las operaciones<sup>36</sup> ya son contempladas y lo que restaría serían las excepciones que son causadas por la estructura misma de la base de datos o la conexión a ella. La falta de tiempo y el considerar que cambiar la base de datos de la aplicación en producción sería un suceso poco frecuente hicieron que se optara por dejar esta función como posible cambio a hacer en un futuro.

### **4.10.4 Patrón Repositorio**

El patrón repositorio es una forma de abstraer el acceso a datos. Podríamos crear una interfaz “IRepositorio” de la cual podemos tener múltiples implementaciones. A la hora de interactuar con las colecciones y objetos del sistema, uno hace uso de la interfaz pero sin saber que clase concreta está utilizando de manera de desacoplar totalmente la persistencia del resto del dominio. Para este obligatorio se pudo haber implementado primero un repositorio en memoria y después cambiar la implementación usada a un

---

<sup>36</sup> INSERT, UPDATE y DELETE

repositorio que haga uso de la base de datos. Por más que esta es una solución elegante para lograr un código mantenible, no fue aplicada en este obligatorio por falta de tiempo.

#### **4.10.5 Patrón Singleton**

Otro patrón de diseño que pudo haber sido aplicado es el patrón Singleton el cual se implementa globalizando una instancia que sea usada en todos lados para poder ser llamada desde cualquier parte y no tener que estar siendo pasada como parámetro siempre. Esto podría haber sido aplicado para nuestra clase “DataManager” la cual es pasada a todos los user control y de esta manera se pudo haber mejorado la mantenibilidad. Al igual que el patrón anterior, este no fue aplicado por falta de tiempo.

## Anexo 1 - Requerimientos Funcionales

Identificador	Nombre	Descripción	Prioridad
RF01	Agregar Categoría	Un usuario debe poder registrar una categoría, estas son únicas y están identificadas por su nombre	Alta
RF02	Modificar Categoría	Un usuario debe poder modificar el nombre de una categoría existente	Media
RF03	Agregar Contraseña	Un usuario debe poder registrar una contraseña en el sistema bajo una cierta categoría y esta debe ser única en términos del sitio y usuario para el sitio. Esta debe tener, además, una sección de notas opcional.	Alta
RF04	Modificar Contraseña	Un usuario debe poder modificar todos los atributos de una contraseña ya existente.	Alta
RF05	Dar de baja Contraseña	Un usuario debe poder borrar una contraseña	Alta
RF06	Generar Contraseña	Un usuario debe poder generar una contraseña aleatoria, con parámetros elegidos desde la interfaz de usuario como qué caracteres tiene y el largo, y esta opción debe estar presente al crear la contraseña	Media



RF07	Agregar Tarjeta de Crédito	Un usuario debe poder registrar una Tarjeta de Credito en el sistema bajo una cierta categoría y esta debe ser única y debe admitir ponerle un nombre, tipo, código de seguridad, fecha de vencimiento y opcionalmente, unas notas	Alta
RF08	Modificar Tarjeta de Crédito	Un usuario debe poder modificar todos los atributos de una Tarjeta de Crédito ya existente.	Alta
RF09	Dar de baja Tarjeta de Crédito	Un usuario debe poder eliminar una Tarjeta de Crédito ya existente.	Alta
RF10	Compartir contraseñas	La aplicación deberá permitir seleccionar una contraseña y compartirla con otro usuario de la aplicación. El usuario con quien se comparte la contraseña, puede ver la información de la misma pero no puede realizar ninguna modificación sobre ella. Si el dueño original de la contraseña realiza algún cambio sobre la misma, este cambio debe ser reflejado en todos los usuarios que tienen acceso a la contraseña.	Media
RF11	Descompartir contraseña	Un usuario debe poder dejar de compartir una contraseña	Media

RF12	Chequear por data breaches	La aplicación deberá chequear que las contraseñas y las tarjetas de crédito guardadas no hayan aparecido en data breaches.	Media
RF13	Listado de Contraseñas	Un usuario debe poder ver las contraseñas que ha guardado, inicialmente de forma reducida y si lo desea, con todos los datos.	Alta
RF14	Listado de Contraseñas Compartidas	Un usuario debe poder ver las Tarjeta de Crédito que ha guardado, inicialmente de forma reducida, con el número oculto, y si lo desea, con todos los datos.	Alta
RF15	Listado de tarjetas de Crédito	Un usuario debe poder ver las contraseñas que han sido compartidas con él, inicialmente de forma reducida y si lo desea, con todos los datos.	Media
RF16	Soporte para múltiples usuarios	Se podrá manejar perfiles independientes, cada uno con sus propios datos. Para eso, cada uno tendrá un nombre de usuario y una contraseña que se ingresan al momento de crear su perfil	Alta
RF17	Reporte de Fortaleza de Contraseñas	Al ingresar al reporte, la aplicación deberá calcular la fortaleza de las contraseñas. En base a ese cálculo se elaborará una tabla indicando la cantidad de contraseñas	Media

en cada grupo		
RF18	Gráficos en Reporte de Fortaleza de Contraseñas	Se deberán mostrar un gráfico que muestre por categoría, la cantidad de contraseñas en cada grupo.

## Anexo 2 - Definiciones

TDD: El desarrollo guiado por las pruebas (TDD) es una técnica para el diseño y desarrollo de software que consiste en escribir la prueba de una funcionalidad antes de codificar.

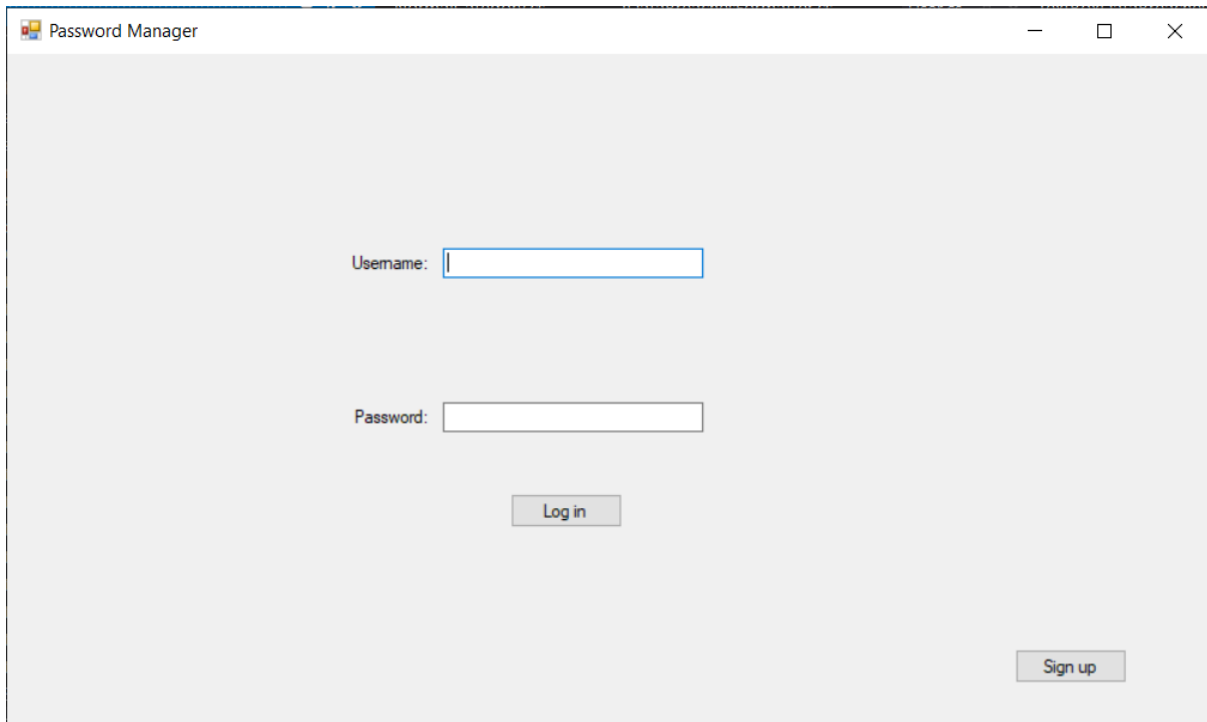
Puede ser reducido en 3 reglas:

- Nunca escribir código si no hay una prueba unitaria.
- Escribir el código mínimo y suficiente de la prueba para que sea una prueba fallida.
- Escribir solo el código mínimo y suficiente para que la prueba pase.

Fuente: Apuntes de clase - Prof. Nicolás Fornaro.

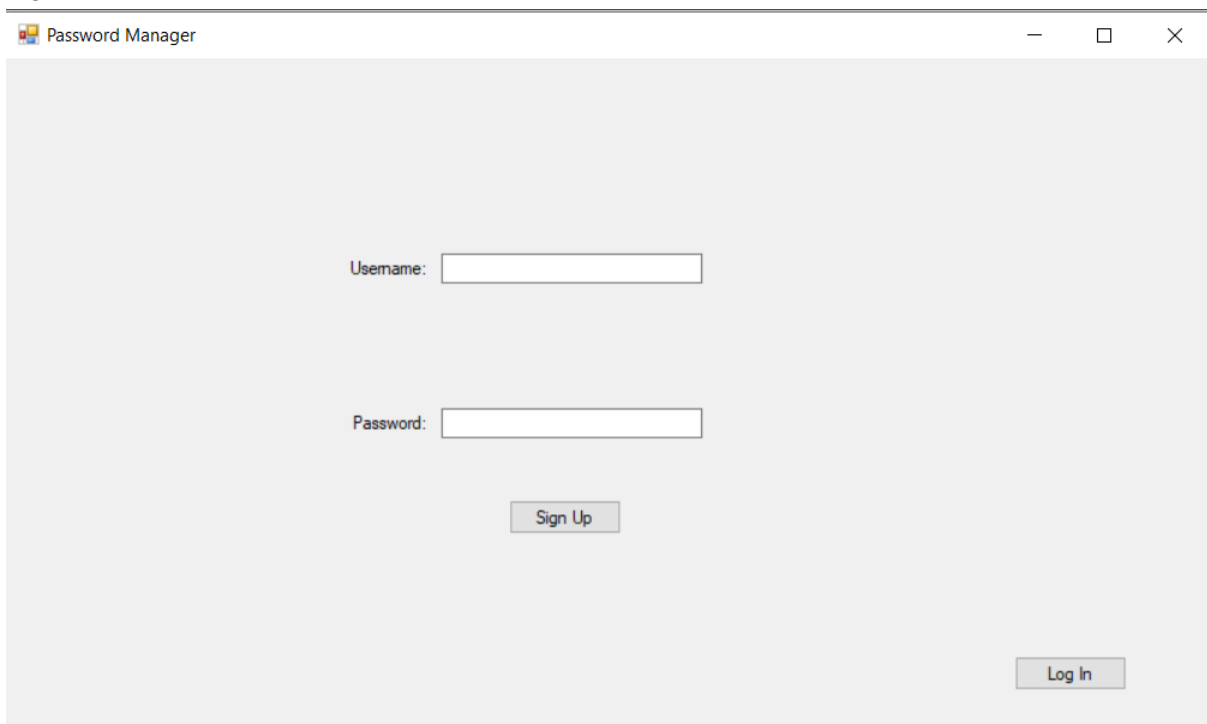
## Anexo 3 - Ejemplos de la Interfaz Gráfica

Login:



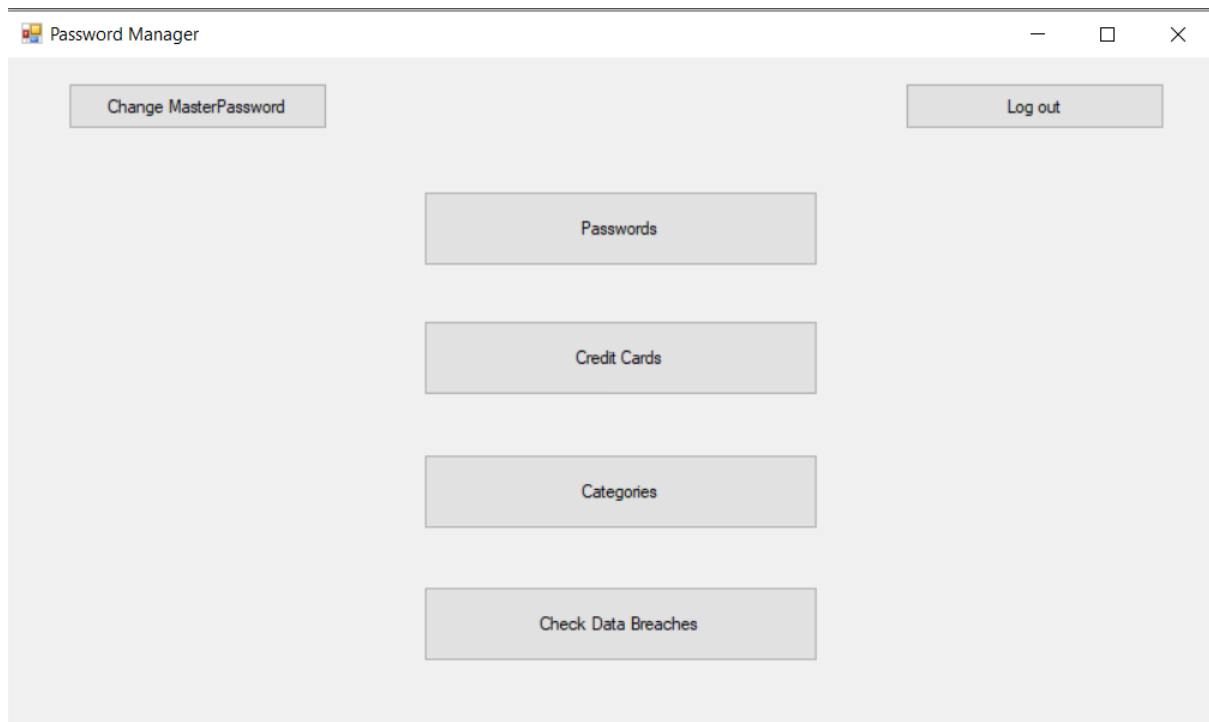
A screenshot of a web application window titled "Password Manager". The window has a light gray background and a standard window control bar at the top with minimize, maximize, and close buttons. The main content area contains two input fields: "Username:" followed by a text box with a blue border, and "Password:" followed by a text box. Below the password field is a "Log in" button. In the bottom right corner, there is a "Sign up" button.

Sign in:



A screenshot of a web application window titled "Password Manager". The window has a light gray background and a standard window control bar at the top with minimize, maximize, and close buttons. The main content area contains two input fields: "Username:" followed by a text box, and "Password:" followed by a text box. Below the password field is a "Sign Up" button. In the bottom right corner, there is a "Log In" button.

## Menú principal:

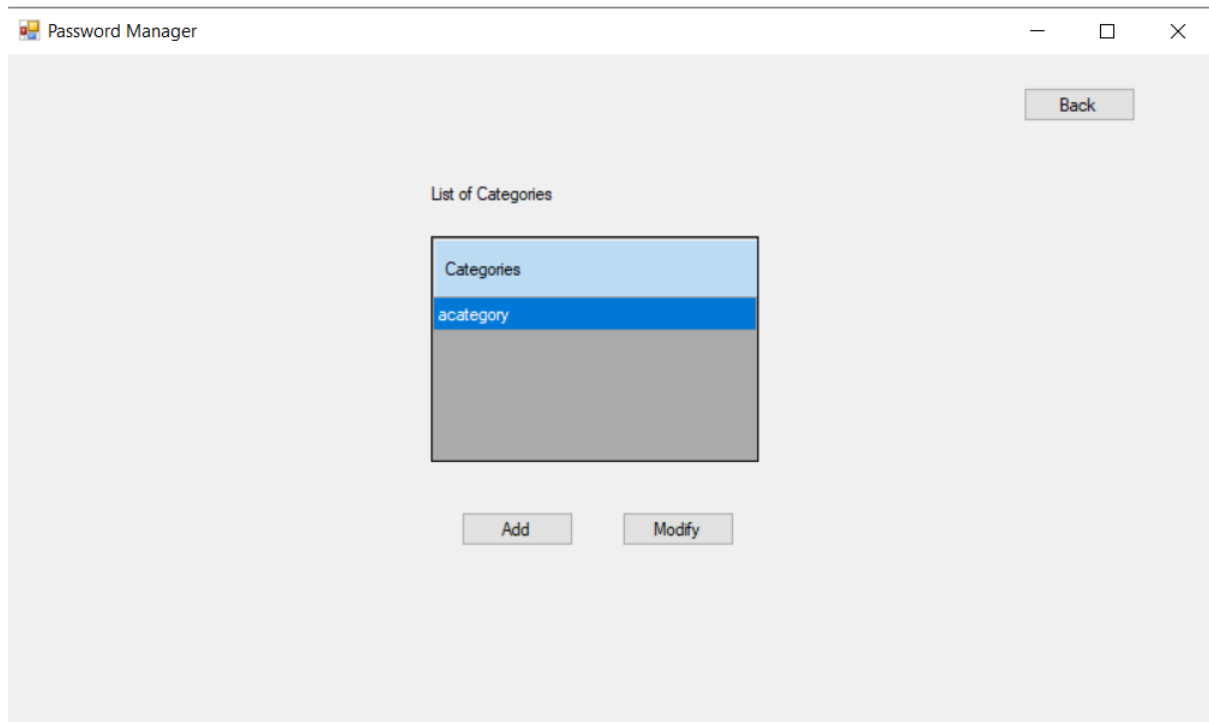


## Agregar una nueva contraseña:

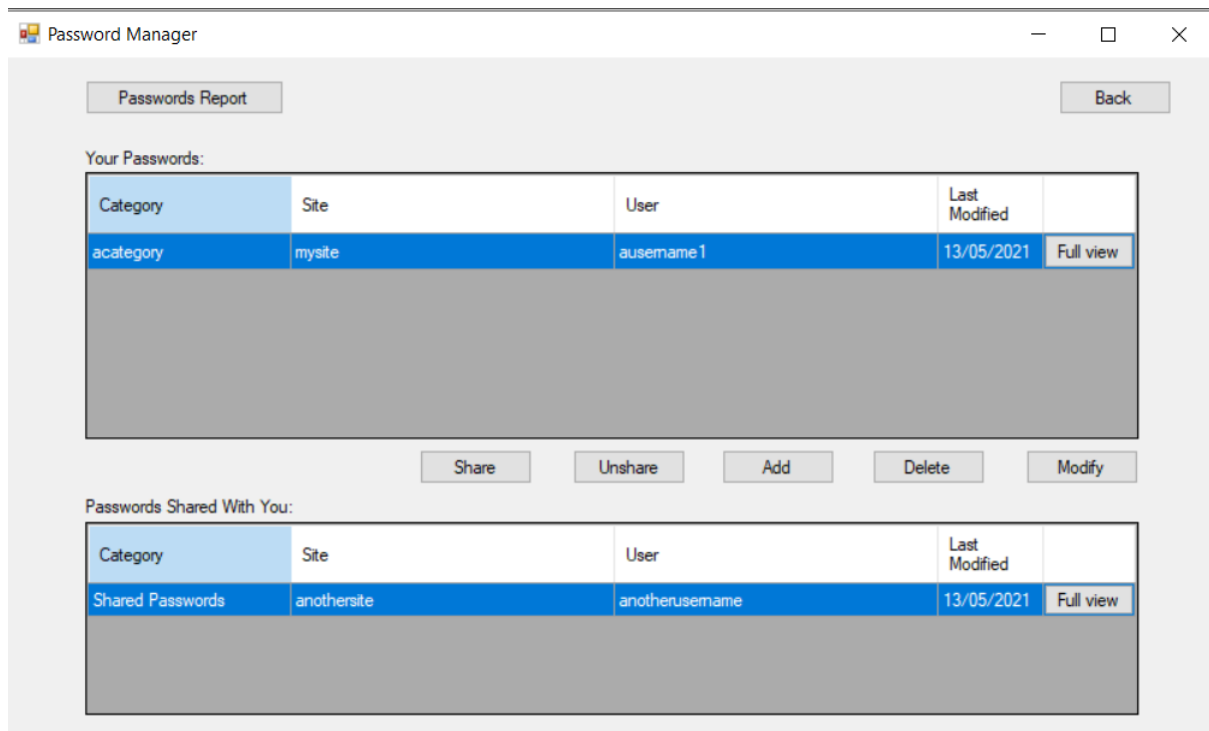
The screenshot shows the "Add New Password" form in the Password Manager application. The window title is "Password Manager". At the top right is a button labeled "Back". The form is titled "Password". It contains the following fields and controls:

- Category:** A dropdown menu with "acategory" selected.
- Site:** A text input field containing "mySite".
- Username:** A text input field containing "aUsername1".
- Password:** A text input field containing "i=4hyjH54k(c26XReB". To its right is a checkbox labeled "Show" which is checked.
- Length:** A spinner control set to "18".
- Character Options:** Four checkboxes, all checked: "Upper Case", "Lower Case", "Digits", and "Symbols".
- Generate:** A button located below the character options.
- Notes:** A text area labeled "Notes" containing the text "This are some notes.".
- Accept:** A button at the bottom center of the form.

## Listar Categorías:



## Listar Contraseñas::



:

## Reporte de fortaleza de contraseñas

Password Manager

— □ ×

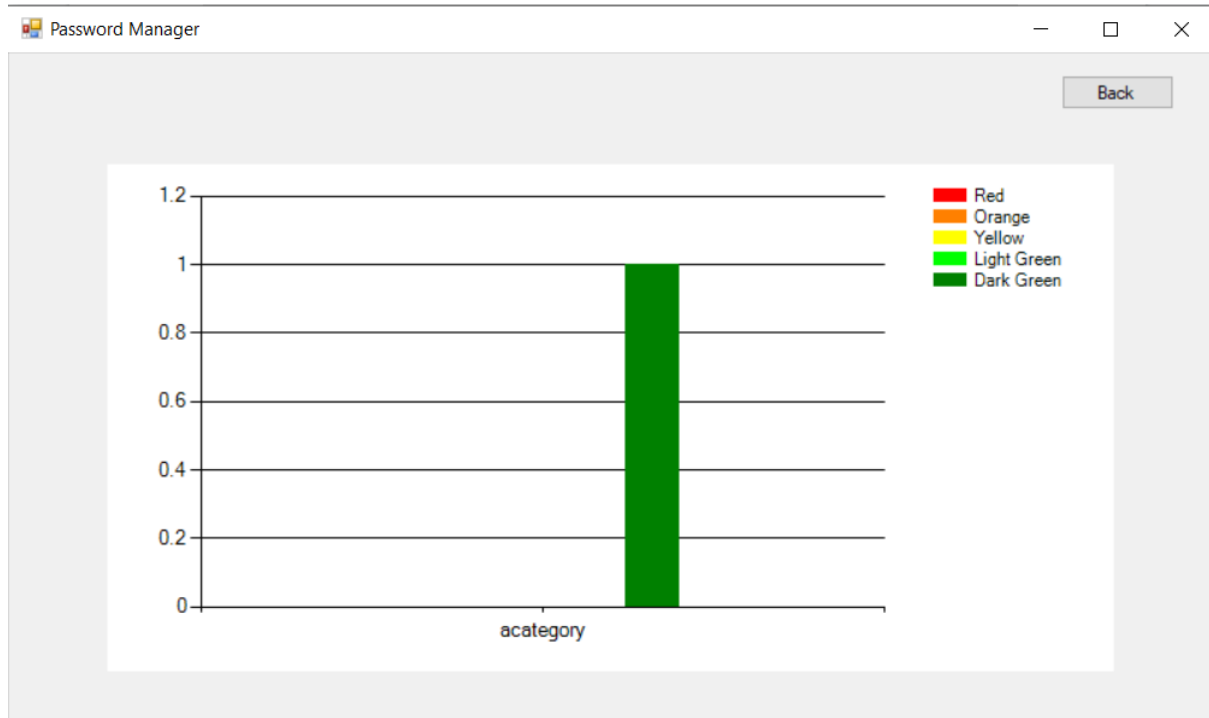
Back

Group	Quantity of Passwords	
Red	0	See List
Orange	0	See List
Yellow	0	See List
LightGreen	0	See List
DarkGreen	1	See List

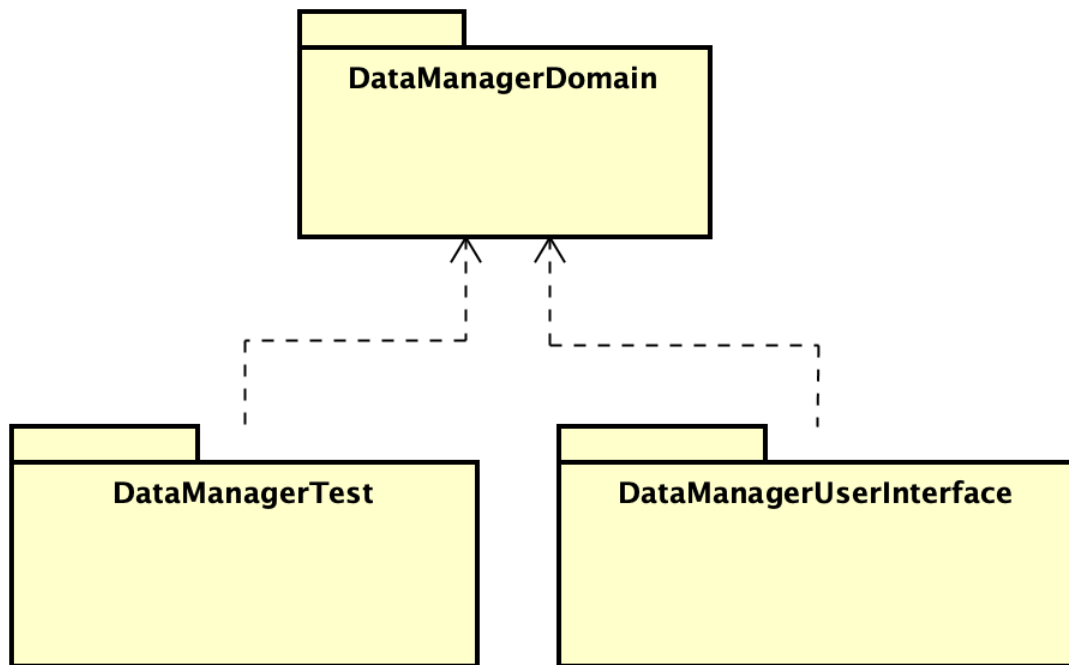
See Chart



## Gráfica de fortalezas por categoría:



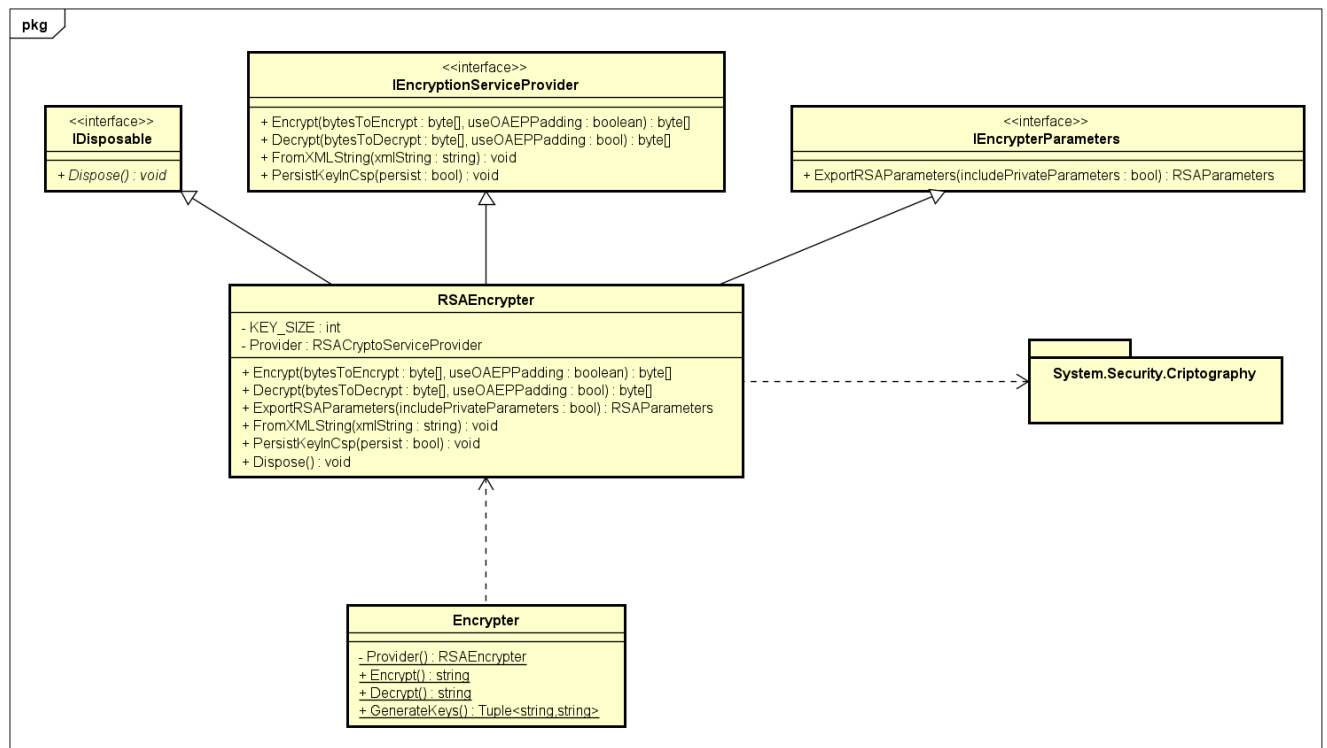
## Anexo 4 - Diagrama de Paquetes



## Anexo 5 - Diagrama de Clases

El diagrama era demasiado grande para incluirse en este pdf y que fuera legible, es por esto que se incluyó el archivo .asta en la carpeta de documentación.

### Diagrama de la clase Encrypter



## Anexo 6 - Pruebas funcionales

### Video 1:

<https://www.youtube.com/watch?v=Bh0cSq22Gso>

Añadir Par Usuario-Contraseña

Casos probados (en el formulario para añadir):

Categoría	Usuario	Contraseña	Sitio	Notas	Resultado esperado
Null	username0	myPass	mySite0		Error - Category Null
Business	username1	myPass			Error - Invalid Site length
Business	username1	myPass	si		Error - Invalid Site length
Business	username1	myPass	mySite11111 11111111111 11111111111 11111		Site trimmed to 25
Business	username1	myPass	mySite1		User-Password Pair added
Personal		myPass	mySite2		Error - Invalid Username length
Business	us	myPass	mySite2		Error - Invalid Username length
Business	username11 11111111111 11111111111 11111111111 11111	myPass	mySite2		Username trimmed to 25
Business	username1	myPass	mySite2		User-Password Pair added
University	username3		mySite3		Error - Invalid Password length
University	username3	Pass	mySite3		Error - Invalid Password length
University	username3	Password11 11111111111 11111111111 11111	mySite3		Password trimmed to 25
University	username3	Password11 11111111111 1111	mySite3		User-Password Pair added
Business	username4	myPass	mySite4	aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaa	Notes trimmed to 250

## Video 2:

<https://www.youtube.com/watch?v=V12pnlPJPKc&feature=youtu.be>

Añadir un Par Usuario-Contraseña y luego otro que tenga el mismo sitio y nombre de usuario que el primero.

Casos probados (en el formulario para añadir):

Categoría	Usuario	Contraseña	Sitio	Notas	Resultado esperado
Business	username1	myPass	mysite1		User-Password Pair added
University	username1	myPass	mysite1		Error - UserPassword Pair already exists
University	USERNAME1	myPass	MYSITE1		Error - UserPassword Pair already exists
University	USERNAME1	myPass	MY SITE 1		Error - UserPassword Pair already exists

Agregar otro Par Usuario-Contraseña

Caso probado (en el formulario para añadir):

Categoría	Usuario	Contraseña	Sitio	Notas	Resultado esperado
Business	username1	myPass	mysite2		User-Password Pair added

Modificar el par (username1, mysite2)

Casos probados (en el formulario para modificar):

Categoría	Usuario	Contraseña	Sitio	Notas	Resultado esperado
Business	username1	myPass	mysite1		Error - UserPassword Pair already exists
Business	username1	myPass	mysiteModified		User-Password Pair modified

Eliminar Par Usuario-Contraseña y volverlo a añadir

Caso probado (en el formulario para añadir):

Categoría	Usuario	Contraseña	Sitio	Notas	Resultado esperado
Business	username1	myPass	mysite1		User-Password Pair added

## Video 3:

<https://www.youtube.com/watch?v=TgO9cAx2HaQ>

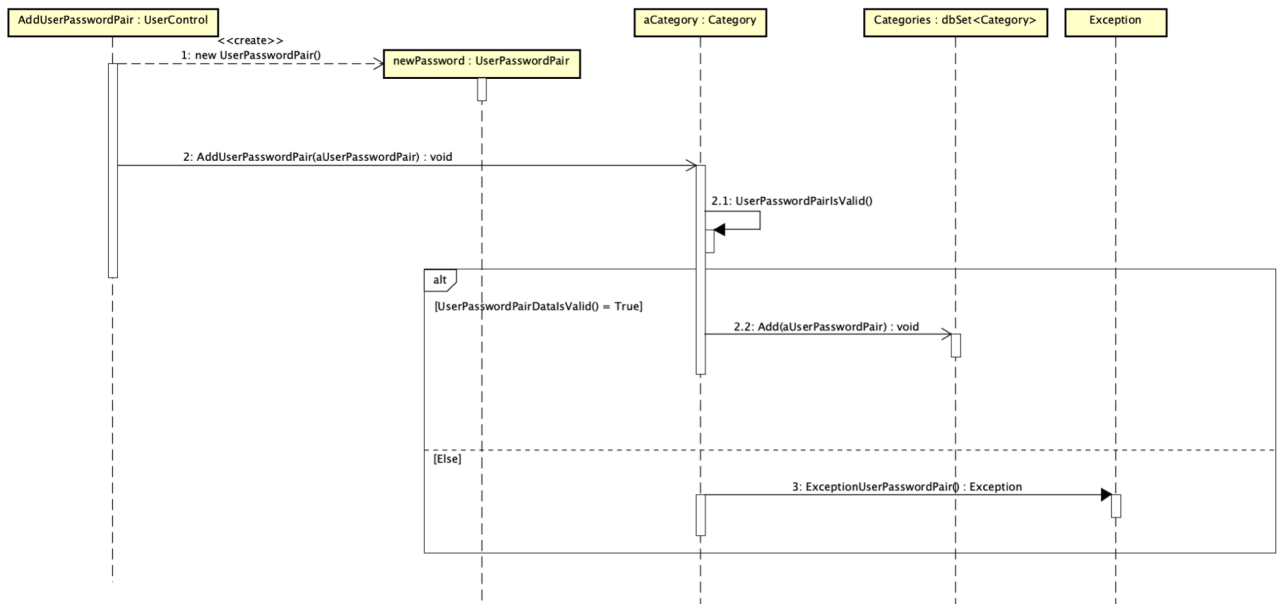
Generar contraseña y ver su fortaleza:

Casos probados (en el formulario para añadir):

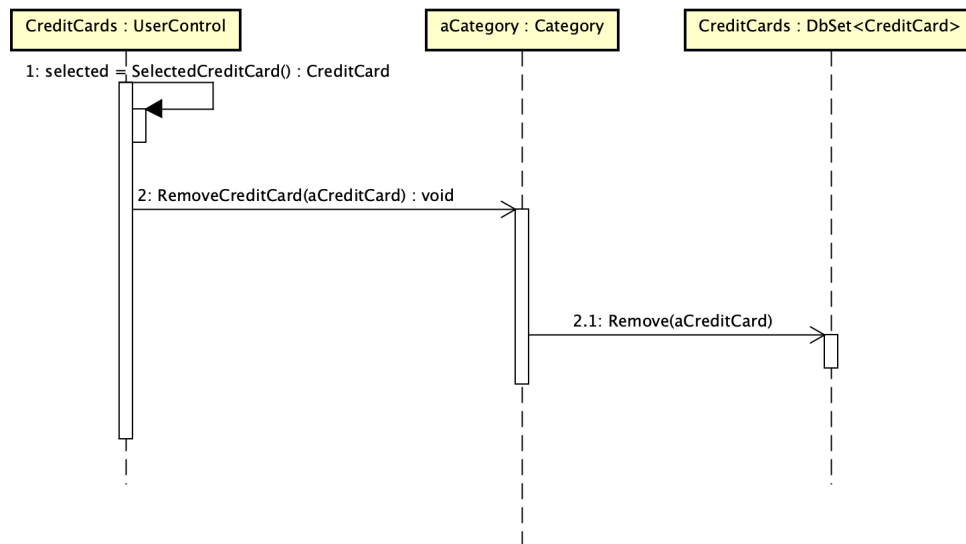
<b>Categoría</b>	<b>Usuario</b>	<b>Contraseña</b>	<b>Sitio</b>	<b>Notas</b>	<b>Resultado esperado</b>
Business	username1	Generate - Length 6	site1		Fortaleza - Red
Business	username2	Generate - ContainsBetween8And14Characters	site2		Fortaleza - Orange
Business	username3	Generate - JustUpperOrLowerCase - Length >14	site3		Fortaleza - Yellow
Business	username4	Generate - JustUpperOrLowerCaseAndSymbolsOrNumbers - Length >14	site4		Fortaleza - Yellow
Business	username5	Generate - JustSymbolsOrNumbers - Length >14	site5		Fortaleza - Yellow
Business	username6	Generate - JustUpperAndLowerCase - Length >14	site6		Fortaleza - Light Green
Business	username7	Generate - JustUpperAndLowerCaseAndSymbolsOrNumbers - Length >14	site7		Fortaleza - Light Green
Business	username8	Generate - UpperAndLowerCaseSymbolsAndNumbers - Length >14	site8		Fortaleza - Dark Green

## Anexo 7 - Diagramas de secuencia

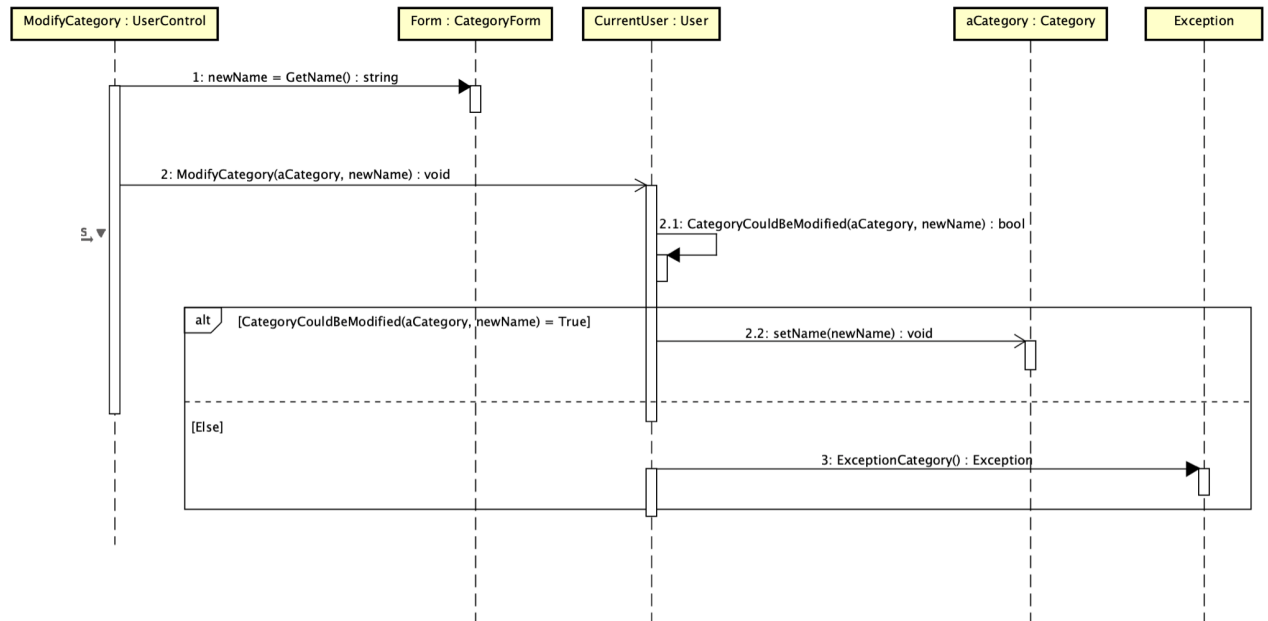
Agregar contraseña:



Borrar tarjeta de crédito:



## Modificar categoría:





## Anexo 8 - Modelo de tablas

Tabla -> Users

Columna	Tipo	Key / Referencia
Username	nvarchar(128) not null	PRIMARY
EncryptedMasterPassword	nvarchar(max) nullable	-
PublicKey	nvarchar(max) nullable	-
PrivateKey	nvarchar(max) nullable	-

Tabla -> Categories

Columna	Tipo	Key / Referencia
Id	int not null	PRIMARY
Name	nvarchar(max) nullable	-
User_Username	nvarchar(max) nullable	Foreign key -> Users

Tabla -> UserPasswordPairs

Columna	Tipo	Key / Referencia
Id	int not null	PRIMARY
EncryptedPassword	nvarchar(max) nullable	-
Username	nvarchar(max) nullable	-
Site	nvarchar(max) nullable	-
Notes	nvarchar(max) nullable	-
PasswordStrength	int not null	-
LastModified	datetime2(7) not null	-
Category_Id	int nullable	Foreign Key -> Categories

Tabla -> CreditCards

Type	Tipo	Key / Referencia
Id	int not null	PRIMARY
Number	nvarchar(max) nullable	-
HideNumber	nvarchar(max) nullable	-
Type	nvarchar(max) nullable	-
Name	nvarchar(max) nullable	-
Code	nvarchar(max) nullable	-
Notes	nvarchar(max) nullable	-
ExpirationDate	datetime not null	-
Category_Id	int nullable	Foreign Key -> Categories

Tabla -> SharedPasswords

Columna	Tipo	Key / Referencia
User_Username	nvarchar(128) not null	PRIMARY, Foreign Key -> Users
UserPasswordPair_Id	int not null	PRIMARY, Foreign Key -> UserPasswordPairs

Tabla -> DataBreaches

Columna	Tipo	Key / Referencia
Id	int not null	PRIMARY
DateTime	datetime not null	-
User_Username	nvarchar(128) nullable	Foreign Key -> Users

Tabla -> LeakedCreditCards

Columna	Tipo	Key / Referencia
Id	int not null	PRIMARY
Number	nvarchar(max) nullable	-
DataBreach_Id	int nullable	Foreign Key -> DataBreaches

Tabla -> LeakedPasswords

Columna	Tipo	Key / Referencia
Id	int not null	PRIMARY
Password	nvarchar(max) nullable	-
DataBreach_Id	int nullable	Foreign Key -> DataBreaches

Tabla -> UserPasswordPairDataBreaches

Columna	Tipo	Key / Referencia
UserPasswordPair_Id	int not null	PRIMARY, Foreign Key -> UserPasswordPairs
DataBreach_Id	int not null	PRIMARY, Foreign Key -> DataBreaches

Tabla -> CreditCardDataBreaches

Columna	Tipo	Key / Referencia
CreditCard_Id	int not null	PRIMARY, Foreign Key -> UserPasswordPairs
DataBreach_Id	int not null	PRIMARY, Foreign Key -> DataBreaches