Universidad ORT Uruguay

Obligatorio 2

Diseño de Aplicaciones 1

Agustín Ferrari 240503

Ivan Monjardín 239850

Francisco Rossi 219401

ORT-DA1/239850_219401_240503 (github.com)

2021

Índice:

1. Descripción General:

En este proyecto el equipo se propuso como objetivo, desarrollar código profesional, fácil de entender, consistente y extensible. Para llevar a cabo esto, se implementaron varias técnicas aprendidas durante el curso, como, Test Driven Development (TDD), recomendaciones de Clean Code, entre otras.

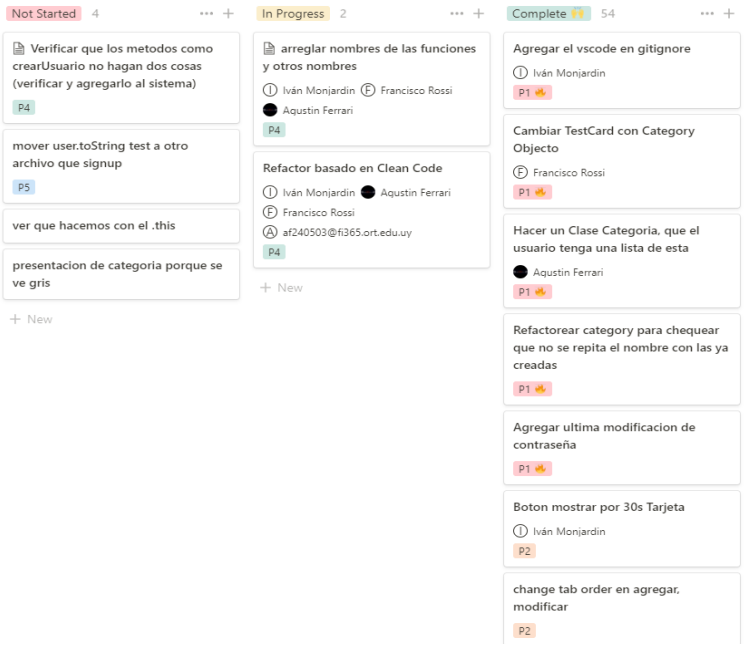
La idea fue seguir estrictamente TDD, plantear un test que no anduviera, escribir el código mínimo para que esta pasara, y luego realizar un refactor. Esto impedía generar código sin que exista una prueba para ello. Sin lugar a dudas generó un gran desafío debido a que ningún integrante del equipo, había desarrollado de esta forma antes, pero más sobre el final, pudimos apreciar ventajas de aplicar TDD.

El equipo cuenta con 3 integrantes, por lo que se decidió en conjunto, trabajar utilizando GitHub y Git Flow, creando ramas individuales para cada funcionalidad y/o refactors necesarios. Se planteó de esta manera para tener la posibilidad de avanzar sin tener que ser necesario la disponibilidad horaria de los otros integrantes todo el tiempo. Para que el código pueda ser consistente y todos estemos al tanto de los avances, se priorizó el uso de los pull request y reunirse al menos 1 o 2 veces al día para discutir y sacar dudas entre todos. Todos los pulls request fueron revisados por los otros dos integrantes y se hicieron comentarios con sugerencias o cambios necesarios, los cuales fueron arreglados antes de realizar los merge a develop.

Se establecieron algunos estándares para trabajar en el repositorio, como por ejemplo:

- El nombre de las ramas feature es "feature-NombreDeLaRama". Se utiliza camelCase y la primera letra del nombre va con mayúscula.
- Los commits se escriben en presente (Ej. “Adds modify button”).
- Para arreglar una funcionalidad se crea una rama “fix-NombreDeLaRama”
- Si se generó un problema al mergear a develop, se puede arreglar y commitear directo a develop o crear una rama Fix.
- Regla para los pull request, donde al menos dos integrantes tienen que aprobarlo.

Continuamos trabajando con la herramienta Notion como lo hicimos para la primera entrega. Nos ayudó a manejar las tareas necesarias, y a planear una mejor solución.
Cada vez que se nos presentaba una situación o se agregaba una modificación en el foro, se agregaba en la columna de Not Started y se le asignaba una prioridad del 1 al 5, siendo 1 lo más importante. Luego cada vez que algún integrante se ponía a desarrollar, elegía una tarea, la marcaba como In progress y creaba la rama para trabajar en ellas. Finalmente se pasaba a la columna de Complete y se podía avanzar con otra tarea.



A partir de la devolución del primer obligatorio, diseñamos una nueva solución para manejar la lógica del negocio. El primer paso, fue separar en distintos paquetes, las clases del dominio y sus validaciones, con la lógica de negocio (BusinessLogic). Para mejorar la cohesión, se realizaron cambios en la lógica, usando controladoras, que actuaban como expertos en cada operación de la aplicación. Estas controladoras actúan como el vínculo entre el dominio y la interfaz gráfica. También se encargan de comunicarse con el módulo de acceso a la base de datos (Repository) , para persistir los objetos.

Pudimos implementar todas las funcionalidades requeridas e incluir algunas que consideramos importantes para mejorar la usabilidad del sistema. Las decisiones de cómo y cuáles funciones implementar fueron tomadas en conjunto por todo el equipo.

Tuvimos un problema a la hora de crear la interfaz de compartir contraseñas que consitía en que a veces al deshabilitar un botón nos tiraba una IndexOutOfRangeException. Luego de intentar encontrar donde estaba el error en nuestro código nos dimos cuenta que no era un problema nuestro sino más bien del set de la Property Enabled del botón, no podemos acceder a dicho código, por lo cual la solución fue hacer un try y catch esperando la exception, esto hace que no se caiga el programa pero a veces el botón queda semi-deshabilitado, es decir, en el proceso de deshabilitarlo el código interno falla y solo se cambia el color pero al hacer hover se sigue marcando el botón por ejemplo.

Otro problema fue que, en el hashing (encriptación de MasterPassword) que utilizamos, se generan strings con algunos caracteres especiales que al generar los scripts para insertar en la base de datos, no se traducen bien. Por lo tanto, las contraseñas de los usuarios de testing Santiago y Mario, solo funcionan si se utiliza el archivo .bak para realizar el backup.

2. Descripción y justificación de diseño:

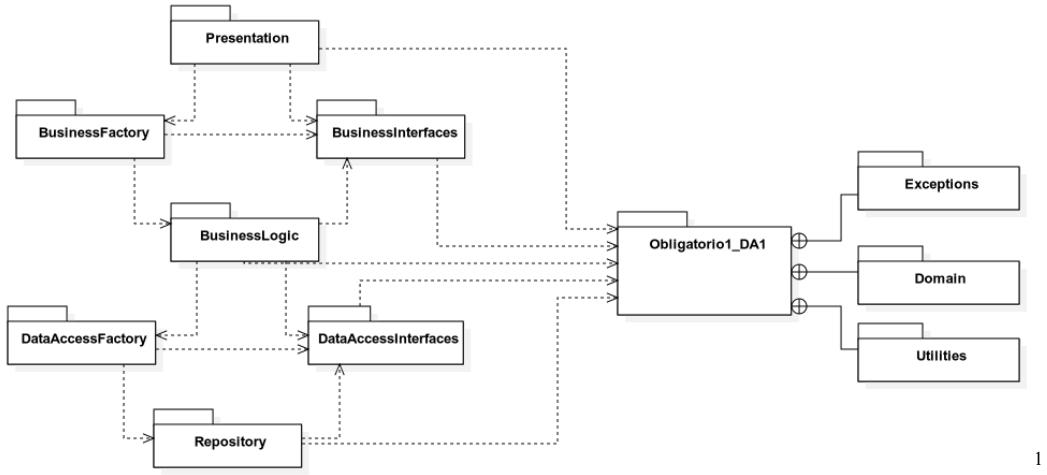
Todos los diagramas mostrados a continuación se encuentran en formato svg publicados en GitHub y adjuntos en el archivo .zip entregado en la carpeta Documentation/Diagrams.

Diagrama de Paquetes

Con respecto a la primera entrega, nuestros paquetes cambiaron en su mayoría. Esto se hizo buscando que el acoplamiento bajara y la cohesión aumentara. Agregamos varios paquetes a la solución, pero dos de los más importantes fueron BusinessLogic y Repository. El primero fue una decisión de diseño para poder utilizar controladoras encargadas de la lógica de la aplicación y el segundo para crear una capa de acceso con la base de datos (BD). En esta última, está el contexto de la aplicación y la interacción con la BD, creando allí las tablas y restricciones necesarias. También están las operaciones de búsqueda de datos, que finalmente se termina usando en la Presentation.
También se separó la parte de BusinessLogic, del paquete Obligatorio1_DA1, que contiene Domain, Exceptions y Utilities. Ya que el mismo se encargará además de ser el vínculo entre el dominio y la interfaz, de comunicarse con el paquete repositorio para persistir los datos. Dadas las reglas de referencia de .NET esto hubiese sido imposible de hacer por la referencia circular (no puede existir un ciclo entre las referencia de los paquetes de la solución) en caso de no haber creado este paquete.
Para resolver este problema y a la vez hacer la solución escalable, también decidimos utilizar el principio DIP. También utilizamos el patrón Factory, encargado del manejo de las interfaces creadas a partir de DIP. Estas decisiones de diseño fueron implementadas entre los vínculos de Presentation-BusinessLogic y BusinessLogic-Repository.

A medida que el código fue creciendo, en función de las pruebas y los principios de Clean Code, se fueron haciendo refactors para mejorar la solución. También se recurrió a diseñar una solución en base a los patrones y principios vistos en clase. Por ejemplo, una clase abstracta Item, que también es parte del dominio y de la cual heredan Password y CreditCard. Esto se decidió implementarlo así, para aprovechar el polimorfismo y no repetir código.

Para el manejo de errores, decidimos crear excepciones personalizadas que fueron utilizadas en todas las clases del dominio. Todas las excepciones heredan de una padre que es la ValidationException y tienen un mensaje de error escrito en español para poder ser mostrado en la interfaz gráfica. Se planteó de esta manera para que a la hora de capturarlas fuera más sencillo y seguir las recomendaciones de Clean Code.



[1]

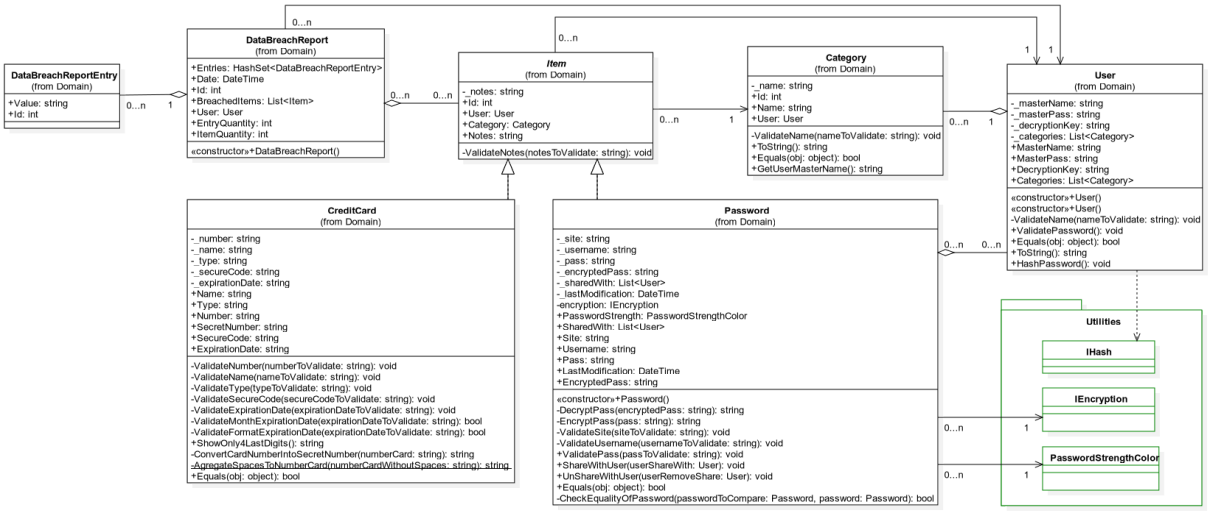
En el paquete de Utilities, se encuentran las clases que dan soporte a otras funcionalidades esenciales de otras clases. Inicialmente se creó después de un refactor donde nos dimos cuenta que más de una clase tenía que acceder a métodos de validación similares (Ej: largo de atributos). Este paquete creció en esta segunda entrega, ya que decidimos agregarles ciertas entidades. En este paquete se encuentra lo relacionado a la encriptación de contraseñas, con las librerías que usamos. También decidimos incluir lectores de data breach, ya sea por string o archivos txt. Por último se agregaron clases buscando reducir las responsabilidades de la clase Password, como lo son, la generación de passwords y calcular el color de la misma.

[1] Diagrama encontrado en Documentation/Diagrams/PACKAGE.SVG

Diagramas de clases:

Para entender más específicamente cómo diseñamos cada paquete, podemos observar los siguientes Diagramas de Clases:
Recalcamos que se pueden encontrar en mejor resolución en la carpeta documentation.

Dominio



2

Con respecto a la primera entrega hicimos algunos refactors de dominio, por ejemplo, movimos los controladores a otro paquete, separando la lógica de negocio, del dominio. Agregamos identificadores a las clases, para poder manejarlas fácilmente con EntityFramework y reducimos las responsabilidades de las clases, para que este únicamente se encarguen de los atributos y sus validaciones específicas.
Algunas de las responsabilidades fueron movidas a BusinessLogic (las que debían tratar con otros objetos de distintas clases), mientras que las demás las separamos en clases de utilities como pueden ser la de encriptación y cálculo de seguridad de contraseñas.
También se agregaron las clases DataBreachReport y DataBreachEntry que se encargan de guardar los datos de los Breaches para poder consultar el historial de los mismo en cualquier momento.

Un inconveniente que tuvimos fue que al no tener un User en Category, y utilizar el code first en la BD, no nos reconocía esta relación. Luego de consultas con distintos profesores, llegamos a la conclusión que haciendo un refactor para agregar la navegabilidad de ambos lados de la relación se resolvería.

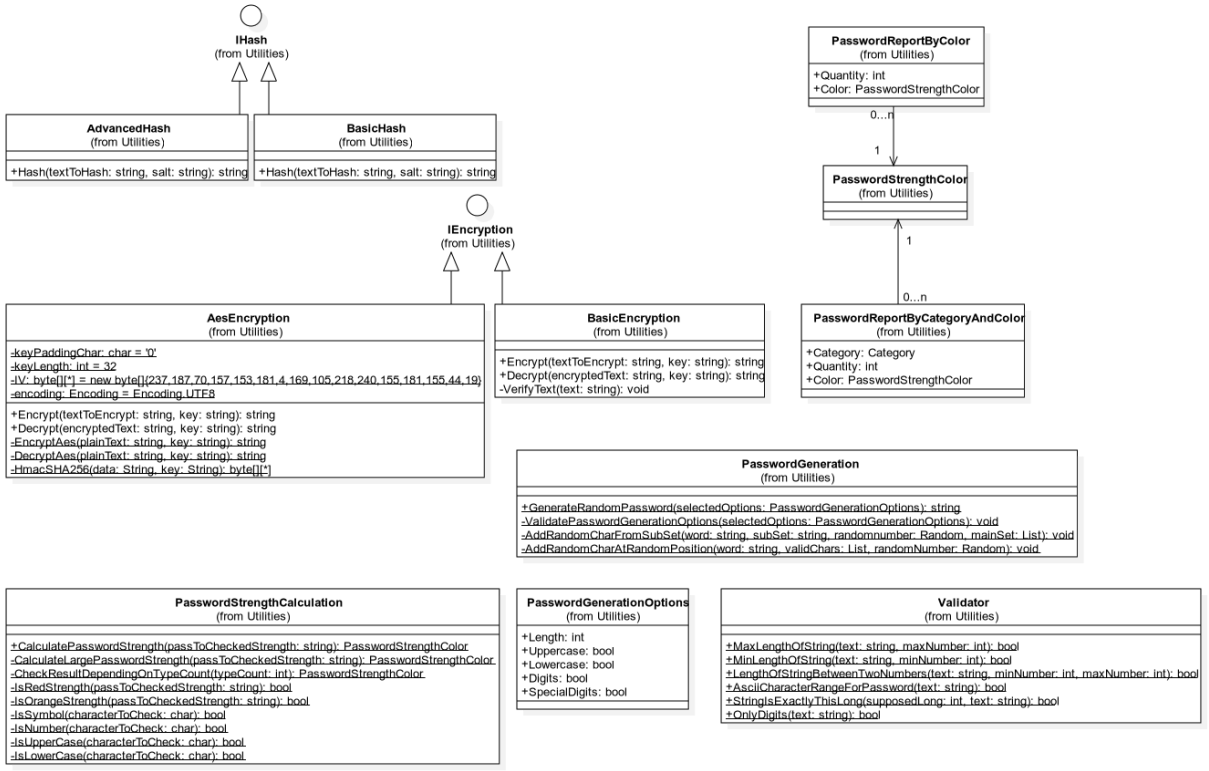
La clase abstracta Item surgió a partir de reconocer que CreditCard y Password, comparten User, Notes, Category. Fue clave darse cuenta de esto, para no repetir código y cumplir con las sugerencias de Clean Code. También aprovechamos el polimorfismo para los DataBreaches y pudimos crear una herencia de Items, Passwords y Credit cards en la base de datos.

---

[2] Diagrama encontrado en Documentation/Diagrams/DOMAIN.SVG

Utilities

En Utilities, se añadieron clases que utilizamos como soporte para funcionalidades de otras clases. Como por ejemplo Validator que valida los largos generales para distintas properties, también se encuentra PasswordReportByColor, la cual es fundamental para el reporte de fortaleza de contraseñas. Otro ejemplo es la encriptación y hash que son las encargadas de encriptar las password del usuario y hashear la master password respectivamente. Esto es para que la aplicación tenga un nivel de seguridad mayor.



[3]

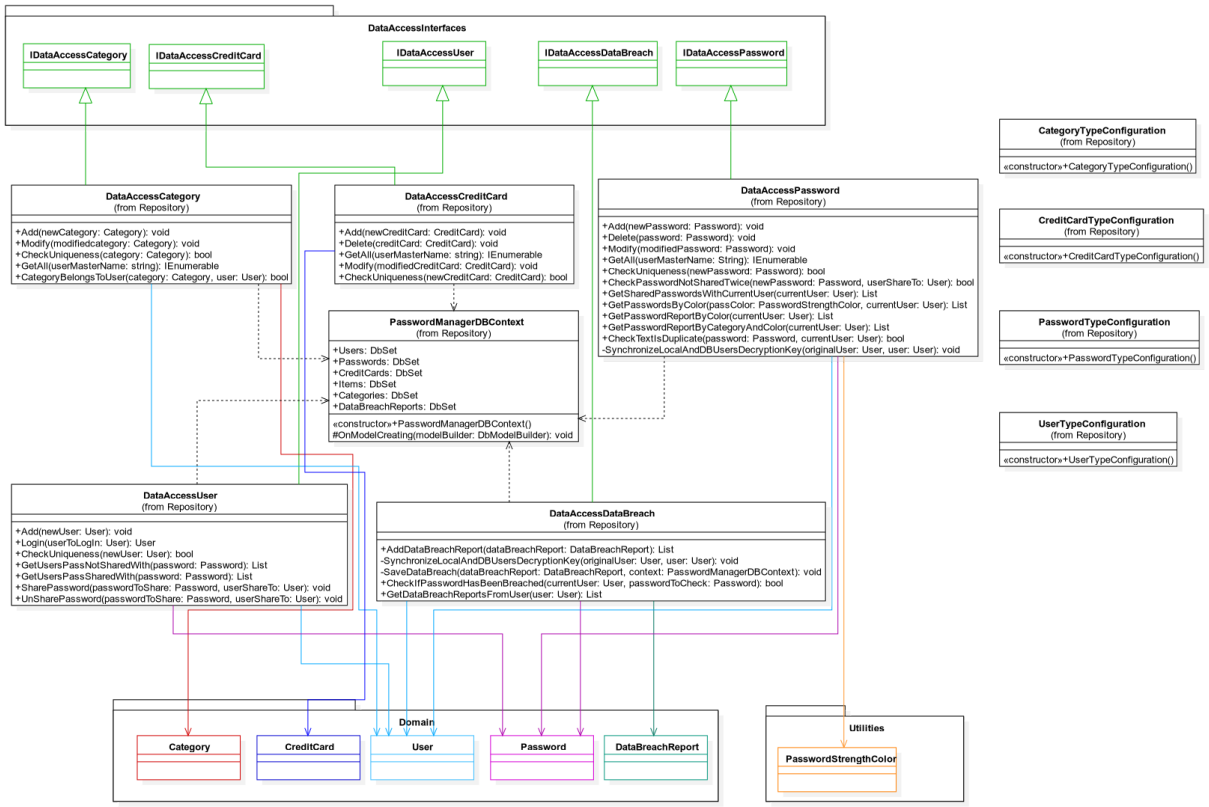
Una decisión que brindó frutos de la primera entrega, fue crear la interfaz de DataBreach, porque para esta entrega, al querer implementar otro formato (txt), la extensibilidad se hizo más sencilla.

Excepciones

Como fue mencionado anteriormente creamos una clase ValidationException y para cada excepción posible de nuestro programa, se crea una excepción con un claro mensaje de error. La property del mensaje de error se inicializa en ValidationException y se especifica en cada una. El mensaje se escribe en español ya que puede ser mostrado al usuario. Solo se agregó una nueva excepción para la encriptación. Este diagrama se encuentra en la sección de adjuntos (Diagrama Exceptions) y tambíen en github para poder referenciarlo, básicamente cuenta con todas las clases excepción heredando la excepción padre ValidationException.[4]

[3] Diagrama encontrado en Documentation/Diagrams/UTILITIES.SVG
[4] Diagrama encontrado en Documentation/Diagrams/EXCEPTIONS.SVG y en Anexo 4.1)

Repositorio



5

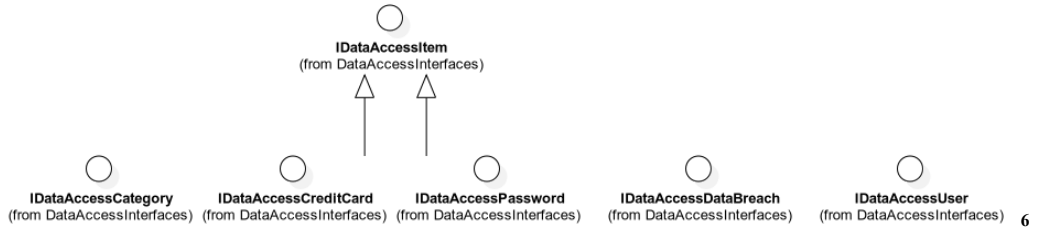
Este paquete cuenta con las configuraciones de las clases para poder hacer las migrations, y las migrations mismas y los data access para cada clase que necesitamos persistir en la base de datos, suplantando las implementación con listas de la primera entrega.
Cada data access además sigue con el contrato de las interfaces que utiliza el business logic.
De estas últimas hay que destacar que en varias ocasiones tuvimos que borrarlas, ya que nos generaban problemas con la definición de la base de datos una vez se que encadenaban muchas. Para resolver esto simplemente borramos la base de datos junto con todas las migrations y las hicimos desde cero, a estas migrations especiales decidimos denominarles “CleanMigrations”(Ver anexo 4.2 Inconsistencia Migrations).

Interfaces de Data Access y Business

Se crearon dos paquetes de interfaces, para que las clases se apeguen al contrato que estas tienen. Si en un futuro se quiere cambiar algunos detalles de la implementación, los contratos van a seguir vigentes y se van a tener que alterar la menor cantidad de clases.
Los módulos de bajo nivel dependen de interfaces, por lo que una solución diferente en el futuro, sería más fácil de implementar. Estas interfaces fueron creadas en nuevos proyectos (BusinessInterfaces y DataAccessInterfaces), y las clases que utilizan sus métodos confían en los contratos únicamente.

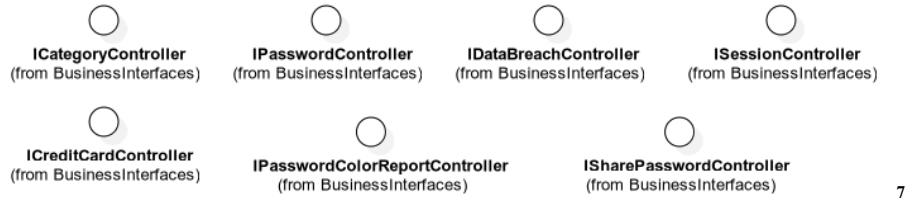
[5] Diagrama encontrado en Documentation/Diagrams/REPOSITORY.SVG

Los dos paquetes de interfaces que se crearon fueron DataAccessInterfaces:



[6]

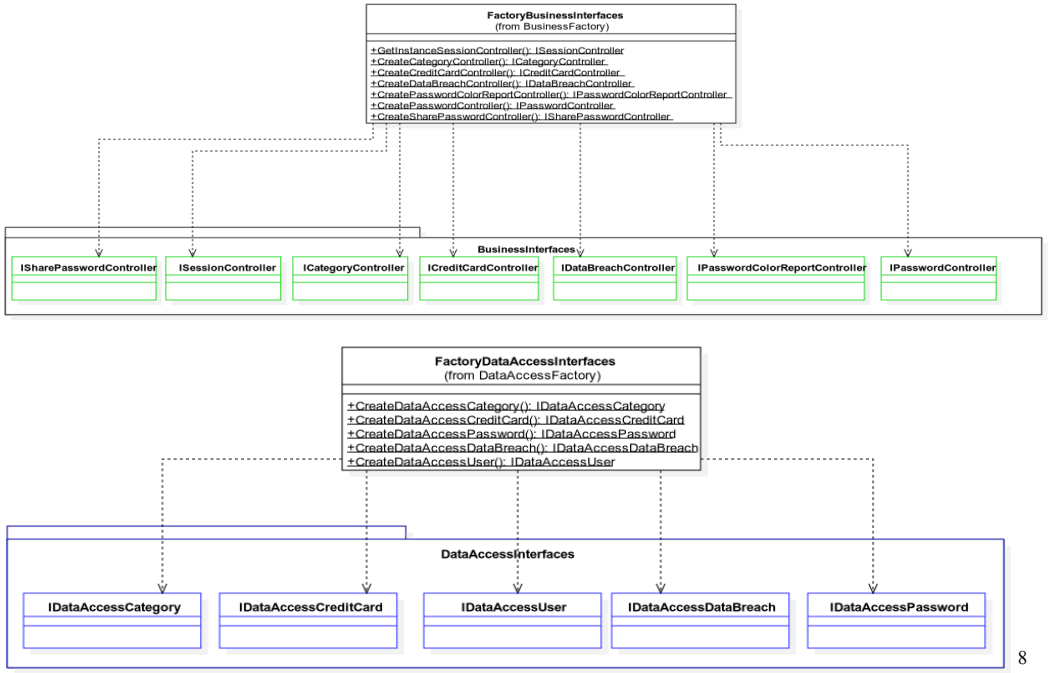
BusinessInterfaces:



[7]

Fábricas de Data Access y Business

Para esta segunda entrega decidimos diseñar una nueva solución, que incluya dos fábricas de entidades. En este caso vinculandolo con el principio DIP, permitiendo crear instancias de clases concretas que están asociadas al contrato a cumplir (interfaces). Esto genera que la futura mantenibilidad sea más fácil tanto como la extensibilidad.
Creamos dos Factory, una que maneja las interfaces de controladoras para que la pueda usar la presentación y otra que maneja los data access para que pueda operar BusinessLogic.



[8]

[6] Diagrama encontrado en Documentation/Diagrams/DATAACCESSINTERFACES.SVG
[7] Diagrama encontrado en Documentation/Diagrams/BUSINESSINTERFACES.SVG
[8] Diagrama encontrado en Documentation/Diagrams/BUSINESSFACTORY y Documentation/Diagrams/DATAACCESSFACTORY.SVG respectivamente

Presentación

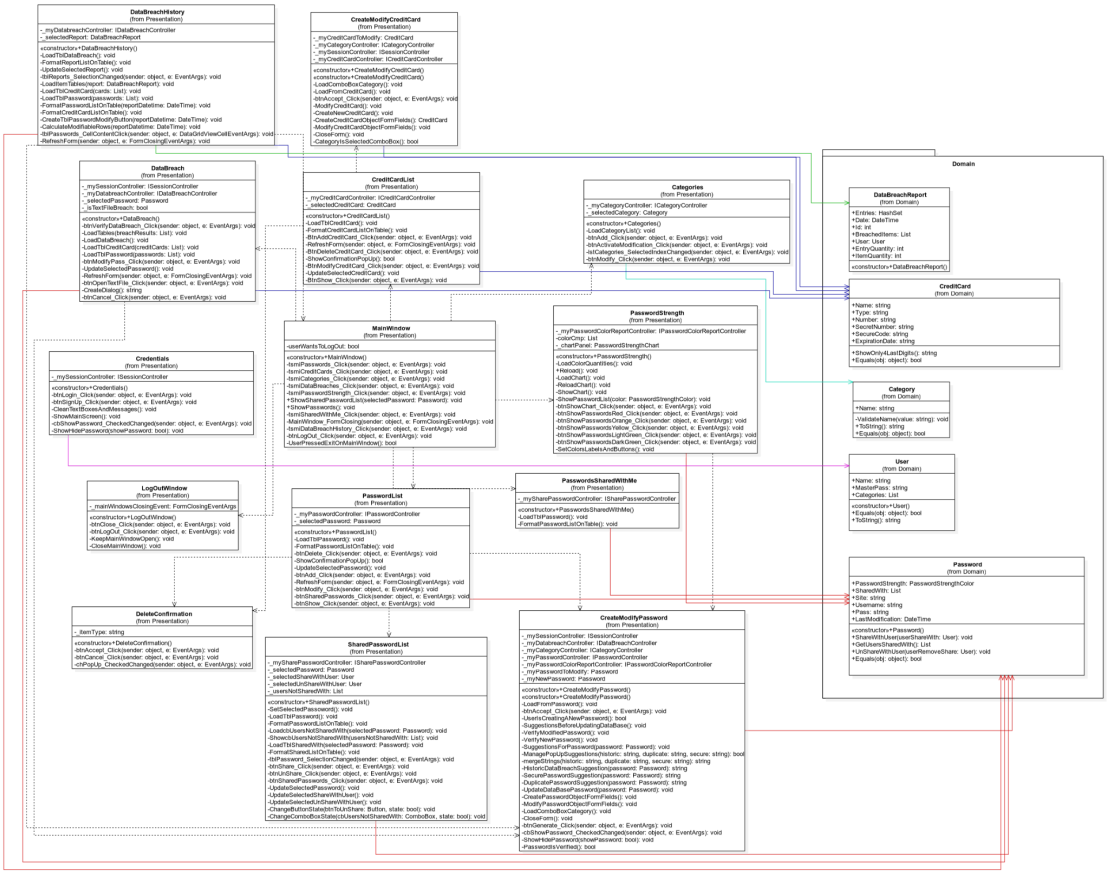
Para la presentación modelamos la interfaz gráfica usando una ventana principal donde se muestran los listados y ventanas principales. También usamos windows separadas para agregar y modificar contraseñas/tarjetas, esto nos permitió poder reutilizarlas en diferentes ventanas, donde era un requerimiento poder a acceder a la modificación directamente y le permite al usuario poder seguir viendo las listas, mientras agrega o modifica un item. Utilizamos también popups para acciones donde se necesita una confirmación, por ejemplo para eliminar un item o cerrar sesión/aplicación.

Para este diagrama se decidió agregar algunas clases del dominio simplificadas ya que nuestra intención era visualizar y comunicar de manera clara cómo se relacionan las clases de la presentación con las mismas. Además estas relaciones están marcadas con diferentes colores dependiendo de la clase para poder diferenciarlas de las de la presentación.

Se pensó incluir un objeto DTO para que en presentation se creará instancia de estos, pero como no correspondía a este curso, decidimos tomar otro camino, con las interfaces y las factories. También consideramos utilizar el patrón builder, pero este se aplica generalmente en objetos más complejos que los que tenemos nosotros o que puedan ser construidos en varias partes y usualmente no tengan tantos parámetros, por lo cual decidimos no utilizarlo.

En este diagrama se prioriza el vínculo con el dominio, y decidimos no agregar con las interfaces de business logic ya que creemos que cargaban mucho al diagrama.

Igualmente esto se puede ver fácilmente en los atributos de las ventanas.



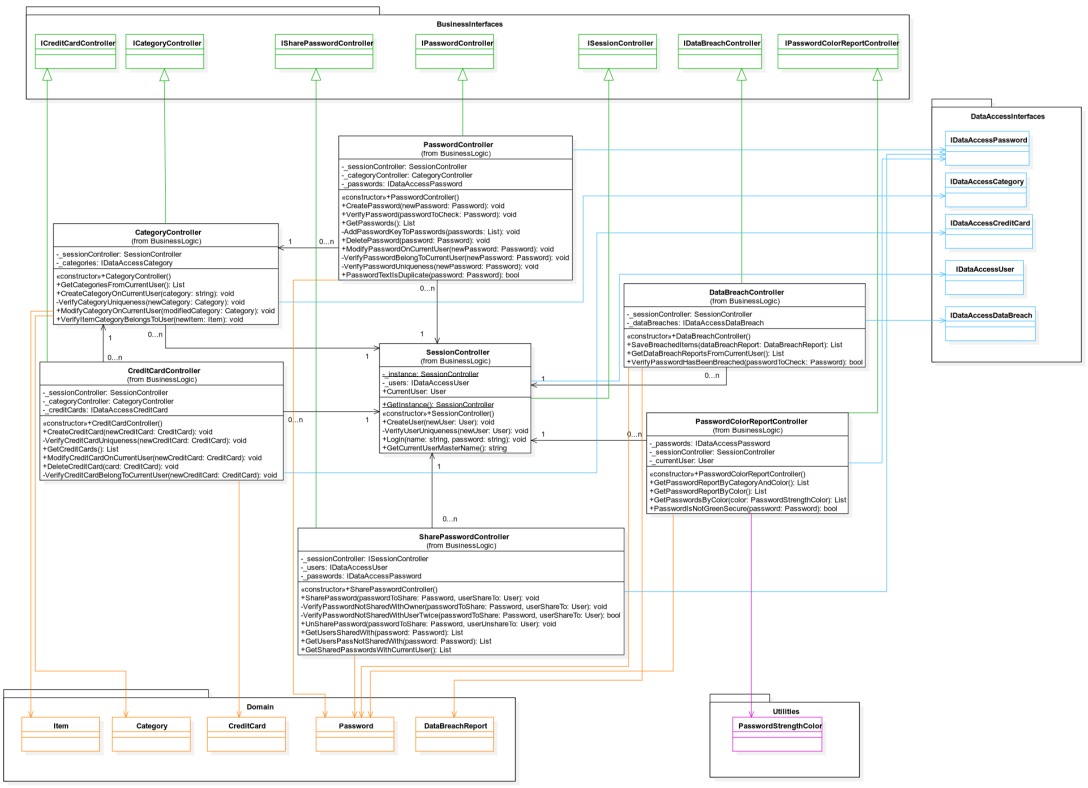
[9] Diagrama encontrado en Documentation/Diagrams/PRESENTATION.SVG

Lógica de negocio

En la primera entrega, se diseñó una solución con el objetivo de extender a una base de datos en la segunda entrega. Esto nos facilitó los refactors para usar los data access en vez de las listas.
Este fue uno de los paquetes en el que hicimos un refactor más grande ya que identificamos que PasswordManager, nuestro único controlador de la primera entrega era un Bloated Controller (tenía muy poca cohesión y demasiadas responsabilidades), por lo cual decidimos separarlo en varias clases controladoras buscando que las mismas tengan mayor cohesión y dividir las responsabilidades.
Cada controladora es experta de alguna entidad, maneja su lógica y persiste los datos por medio de los DataAccess. Esto suplanta todas las listas que teníamos para el primer obligatorio. Para esto, como ya mencionamos, las controladoras utilizan interfaces de los data access y se apegan a sus contratos, para que en un futuro si se quisiera cambiar el motor de BD, la cantidad de clases a cambiar sería mínima.

Para manejar el usuario que esté utilizando la aplicación, manejamos un currentUser en la controladora de sesión, la sessionController, encargada de loguear al usuario y manejar la sesión. De esta manera el mismo solo podrá acceder a la información que le pertenezca y no podrá modificar la del resto de usuarios.
Para esta controladora utilizamos el patrón Singleton ya que queríamos mantener una única sesión activa a la vez y que de esta forma todos conozcan al currentUser y no haya inconsistencias como tener current users diferentes en la misma sesión.

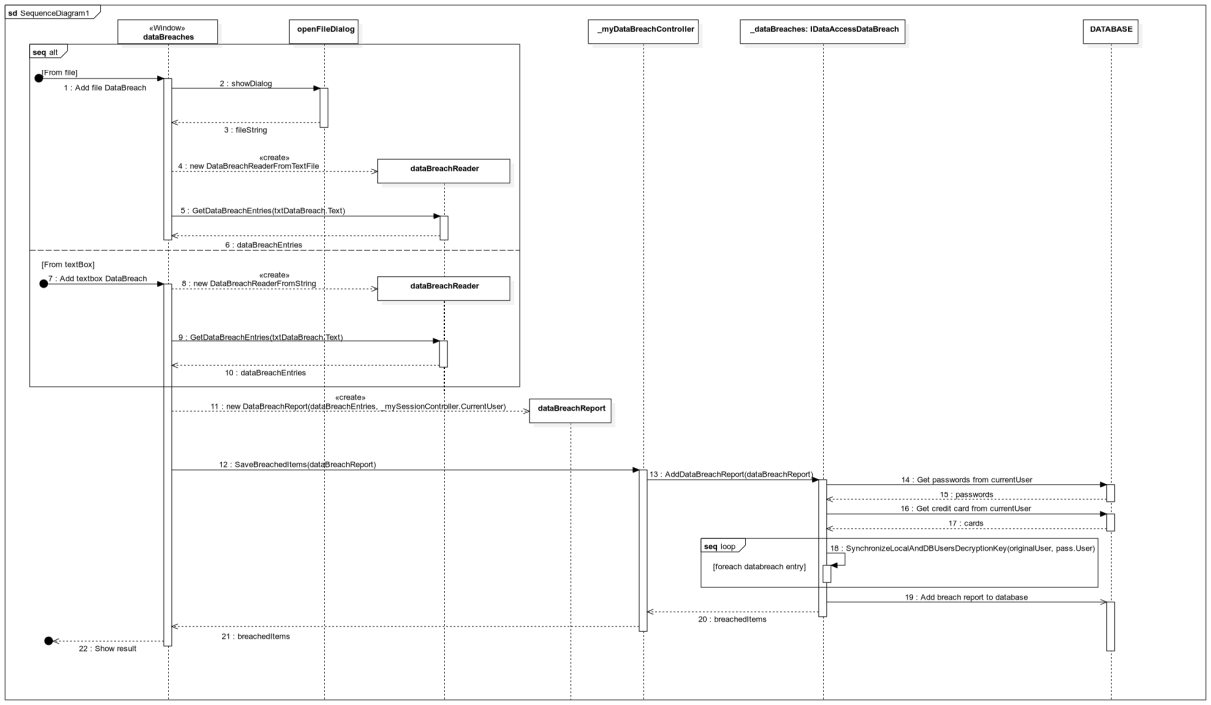
[10]



[10] Diagrama encontrado en Documentation/Diagrams/PRESENTATION.SVG

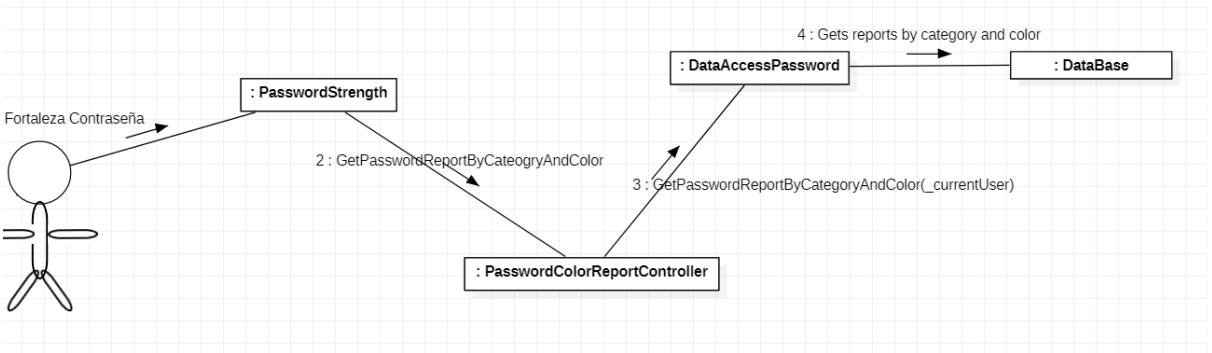
Diagramas de Interacción

En este diagrama, se quiso reflejar las clases involucradas en la acción de cargar un data breach, ya sea por medio de un archivo txt, o que el usuario ingrese un texto. La acción comienza en la interfaz gráfica y termina en la base de datos, guardando los datos breacheados.



11

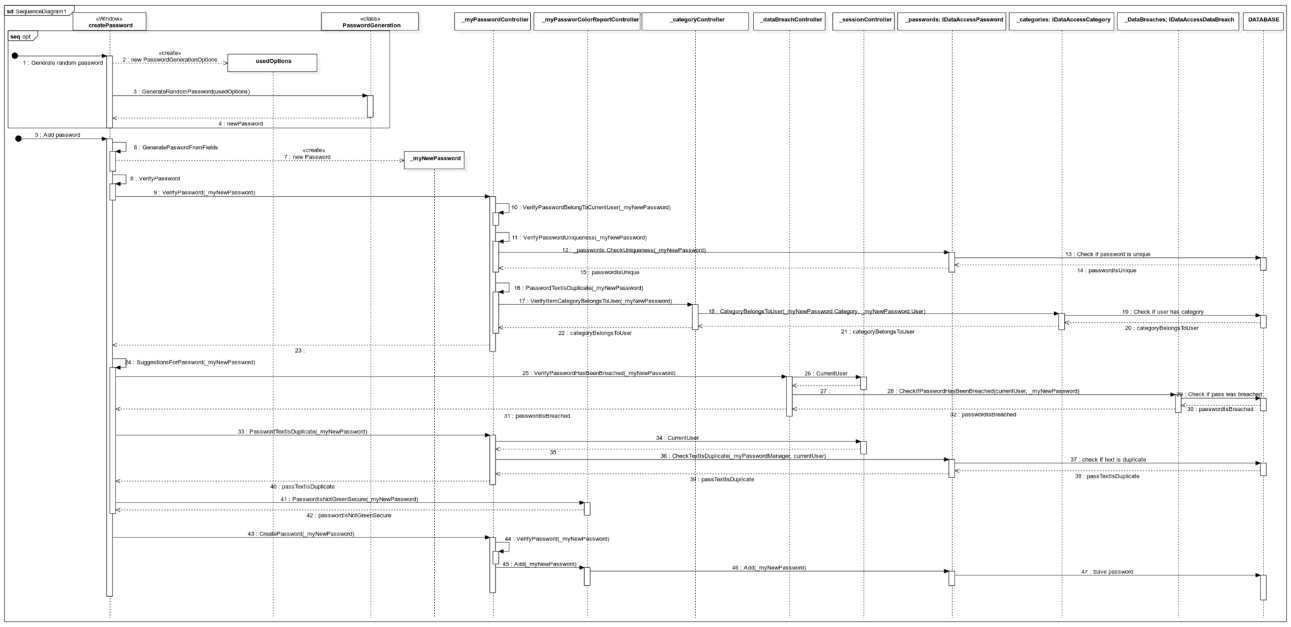
Este diagrama representa cómo se carga la gráfica con los colores de las contraseñas para ese usuario. Desde el menú el usuario elige la opción de Fortaleza de contraseña y luego desde la presentación ya se comunica con la controladora específica para resolver la consulta, llevando esta hacia la base de datos donde trae los datos necesarios.



12

[11] Diagrama encontrado en Documentation/Diagrams/ADDDATABREACH.SVG
[12] Diagrama encontrado en Documentation/Diagrams/PASSWORDSTRENGTH.PNG

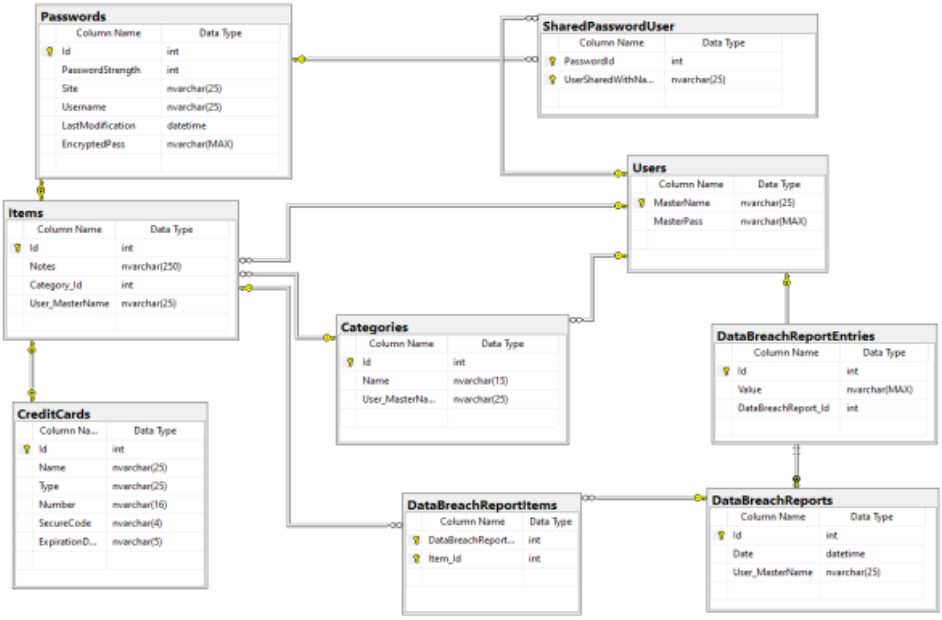
Este diagrama quiere reflejar la acción de agregar una password al sistema. En esta acción interactúan varios participantes. Para reducir la complejidad del programa se evitó mostrar la encriptación, es una acción separa del objetivo final del diagrama.



13

Modelo de tablas de la estructura de la base de datos

Este diagrama se creó a partir de la base de datos creada con Entity Framework y metodología code-first. A partir de esas definiciones se crearon las siguientes tablas y relaciones.



14

[13] Diagrama encontrado en Documentation/Diagrams/ADDPASSWORD.SVG
[14] Diagrama encontrado en Documentation/Diagrams/DATABASETABLES.PNG

Diseño y Mecanismos generales

Algunas de las decisiones de diseño ya fueron comentadas anteriormente, sin embargo queríamos resaltar otros puntos relevantes.

Estándares de codificación

Al inicio del proyecto entendimos que era necesario establecer en grupo estándares de codificación para poder llegar a una solución clara y con buena calidad de código.
Algunos estándares establecidos fueron:

- El código y commits se escribe en inglés
- La interfaz gráfica se hace en español
- El nombre del atributo privado de las clases es _nombreAtributo
- No utilizar el "this." para referirse a atributos al menos que sea:
    - Para evitar errores de compilación
    - Para ayudar a la comprensión del código
    - Para referirnos a una property adentro de una misma clase
- Los métodos con nombres autogenerados de la interfaz gráfica, no se cambia la primera letra a mayúscula.
- Todos los componentes del mismo tipo deben comenzar con el mismo prefijo. Ejemplo, los botones con btn, las etiquetas con lbl etc.

Se decidió grupalmente eliminar un estándar que teníamos en la primera entrega, que indicaba que, cuando pasabamos PasswordManager como parámetro, utilizábamos el prefijo p.

Patrones y Principios

Cuando se empezó a pensar en un diseño para esta nueva entrega, se trató de implementar nuevos patrones y principios que se dictaron en el curso. La idea fue analizar dónde se podría aplicar cada uno de ellos sin forzar su uso en sitios donde no aporten a la solución.

Principio DIP:

Utilizamos este principio, para que la aplicación sea extensible y mantenible, para que en futuro, con nuevos cambios, no se tenga que modificar gran parte del código. Por ejemplo, implementar otro motor de BD, o se tenga que cambiar a otra interfaz gráfica.

Principio OCP:

Se realizaron varios refactors desde el primer obligatorio para conseguir una solución que esté abierta a nuevas modificaciones pero que no resulten en cambios sustanciales en el código, buscando cumplir con el patrón de abierto-cerrado. Ejemplos van desde la separación de clases que tenían muchas responsabilidades, hasta la implementación del principio en la encriptación y hasheo de contraseñas el cual explicaremos más adelante.

Patrón Singleton:

Utilizamos este patrón en la controladora de SessionController. Ya que necesitábamos que solo pueda existir una instancia de la misma porque es la encargada de manejar la sesión del usuario que ingresa a la aplicación.

Patron Factory:

El propósito de esta implementación, fue desacoplar Presentation con BusinessLogic y también BusinessLogic con la capa de Repository. Estas "fábricas" intervienen en el medio de estas interacciones, y crean una instancia de la entidad que se quiere usar en un módulo de más alto nivel, lo que reduce el acoplamiento, ya que deriva la creación de la clase concreta a otro paquete.

Patron Strategy:

Buscando la extensibilidad de la solución y reducir el acoplamiento de la solución, se aplicó el patrón strategy en la encriptación de contraseñas. Como vamos a explicar más adelante, se creó una interfaz que todas las implementaciones de encriptación utilizan, la clases del dominio solo utilizan los métodos de la Interfaz y no de alguna clase concreta.

Password Strength Report

Para la funcionalidad del reporte de fortaleza de contraseña, la interfaz gráfica precisaba tener la cantidad de contraseñas por color y categoría, y por color únicamente (para mostrar en la gráfica). Por lo tanto para cada caso se creó un struct que contenga la información necesaria. La idea detrás de esta decisión fue que la presentación solo tenga que realizar un pedido de información a la controladora y no, por ejemplo, realizar 5 pedidos de cantidad de contraseñas (uno por cada color).

Confirmación de eliminación

Para la eliminación de Items, se implementó un pop-up de confirmación de la acción, en caso de que la acción no fuera intencional, teniendo en cuenta las heurísticas de diseño de interfaz. Dado que el usuario de la aplicación, puede encontrar estos pop-up molestos, se incorporó un checkbox que pregunta si no quiere que se muestre otra vez la ventana de confirmación. Para resolver esta situación se agregó dos variables de proyecto a Presentation de tipo bool, que tienen alcance (scope) a cada User y guardan la elección para eliminación de CreditCard y de Password por separado. Decidimos resetear estas variables cuando se cierre la aplicación.

Name	Type	Scope	Value
DontShowAgainPopUpPassword	bool	User	False
DontShowAgainPopUpCreditCard	bool	User	False

Datos de Prueba

Se incluyen un base de datos de prueba con los siguientes usuarios, que tienen tarjetas, contraseñas, categorías precargados (Usuario, contraseña):

- Juana, Juana (incluye 3 DataBreaches)
- Laura, Laura
- Pedro, Pedro
- Mario, Mario
- Santiago, Santiago

Persistencia de los datos

Para almacenar los datos se creó una base de datos usando SQL Server y Entity Framework code-first. El módulo de interacción con esta, es repository, donde se definieron los data access, que se encargan de comunicarse con la base de datos para guardar las entidades y hacer las consultas pertinentes.
El criterio que tomamos para crear las migrations fue hacer la menor cantidad de cambios en el dominio ya que usamos code-first y pensamos que deberíamos poder crear la base de datos sin cambiar nada del dominio. El único cambio que tuvimos que hacer, ya mencionado anteriormente fue agregar la navegabilidad de Category a User ya que sino era imposible poder agregarla sin usar una query de SQL cruda. También tuvimos que agregar Ids a las clases para poder manejarlas en la base de datos sin tener problemas como intentar cambiar una primary key.

Historico DataBreach

Para poder guardar el histórico de DataBreach creamos una clase de dominio que contenga la información que nos interesa guardar, también se reutilizó la interfaz que habíamos creado en el primero obligatorio buscando que este sea más extensible, esto nos facilitó implementar la lectura a partir de archivos txt.

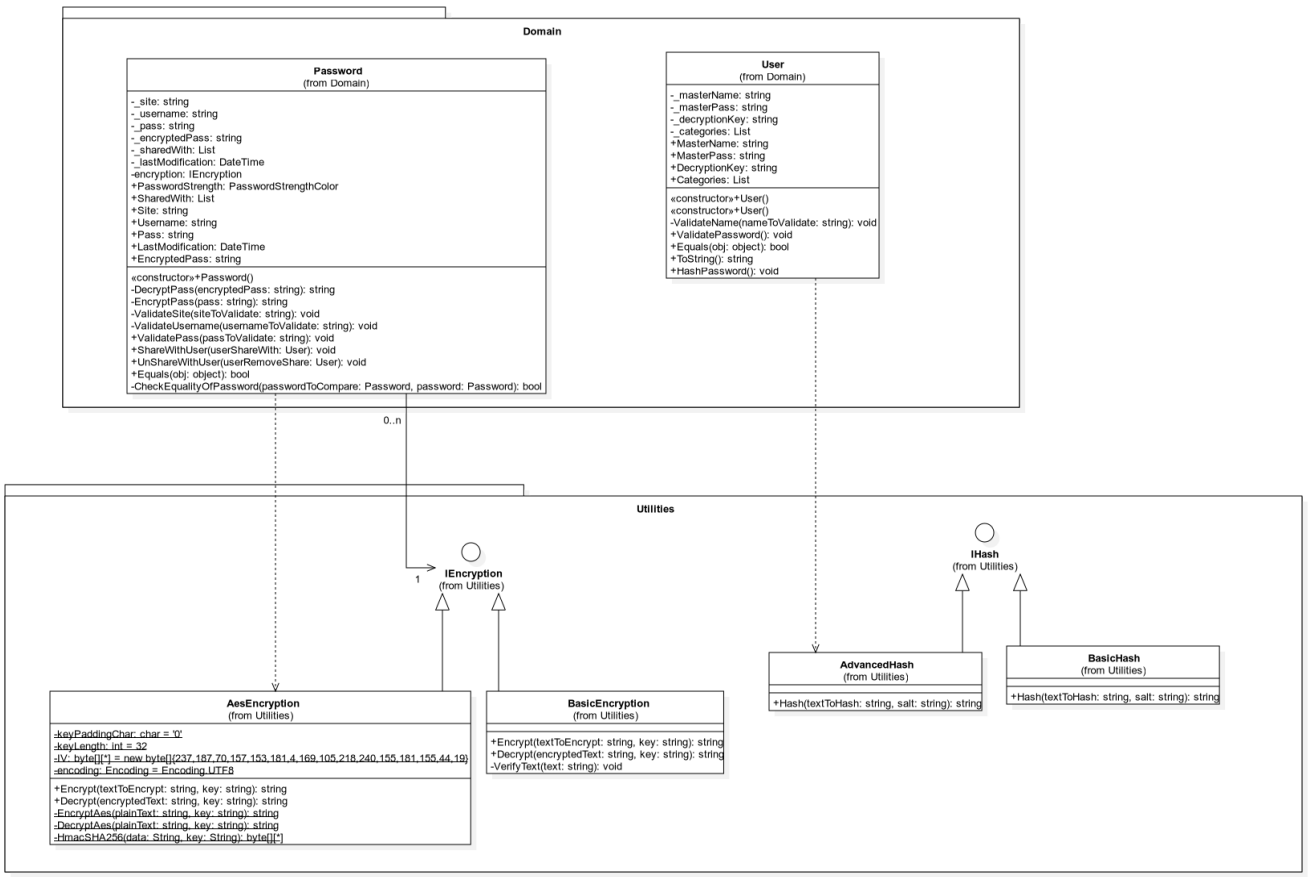
Sugerencias al crear/modificar contraseñas

Al crear o modificar una contraseña, la aplicación le muestra sugerencias para posibles cambios al usuario. La contraseña puede haber estado breacheada, no es demasiado segura (seguridad menor a verde o verde claro) o ya se usó en el sistema. Si el usuario lo desea puede crearla de todas formas, si no puede cambiarla primero.

Encriptación

Para este requerimiento , se buscó una solución extensible y que no acople los algoritmos de encriptación al código que ya tenemos. Se creó una interfaz que cualquier algoritmo de encriptación tiene que implementar y el dominio solo utiliza los métodos de la interfaz, haciendo fácil la transición a otro algoritmo. Esto permitió, adecuar el futuro algoritmo al resto del código y no adecuar el resto del código a algún algoritmo específico que elijamos.

Para los tests se creó una clase abstracta y para testear un algoritmo concreto se tiene que heredar esa clase y especificar qué implementación testear.
Inicialmente se utilizó una clase “BasicEncryption” como mock para continuar el desarrollo sin tener que implementar ningún algoritmo complejo de encriptación .
Cambiar de algoritmo requiere mínimos cambios a nivel del resto de clases del dominio y el uso de la interfaz y de tests genéricos, asegura que cualquier algoritmo que pase los tests va a funcionar. Por lo tanto, la solución se encuentra abierta a la extensión (agregar nuevas encriptaciones es sencillo) pero cerrada a la modificación (solo requiere mínima modificación si se desea cambiar el algoritmo elegido).



15

Se decidió encriptar/desencriptar las contraseñas del usuario usando la contraseña maestra del mismo y guardando en la propiedad DecryptionKey en caso que, en el futuro, se quiera utilizar otro campo para desencriptar.
Para las contraseñas maestras se decidió hashear de forma que no se puede volver al texto original, la única forma de loguearse es tener la contraseña original, hashearla y comparar con la de la BD.
Por seguridad, nunca se suben las contraseñas sin encriptar/hashear a la base de datos.
Para los algoritmos finales se utilizaron:

Encriptación de contraseñas de usuarios: Librería Cryptography, algoritmo AES
Hash de contraseñas maestras: Librería Cryptography, algoritmo SHA256

[15] Diagrama encontrado en Documentation/Diagrams/ENCRYPTION.SVG

3. Cobertura de pruebas unitarias con su debido análisis y justificaciones.

Base de datos de Testing

Utilizamos una base de datos, destinada solamente para el testing. Esto se decidió de esta manera, para poder probar casos bordes y que no se viera afectado los datos que inicialmente cargamos a la aplicación. Incluímos algunas técnicas para no repetir código, como el TestCleanUp, para que cada prueba siga siendo independiente y a su vez, cada corrida de pruebas no depende de los datos en sí.

Test Coverage

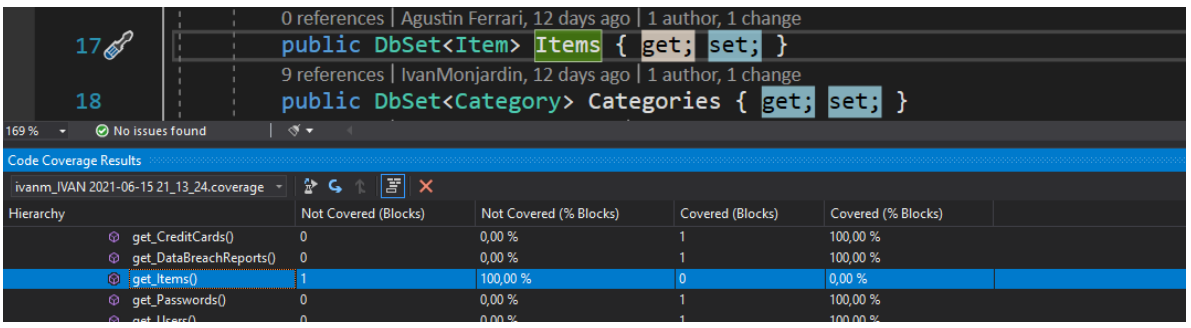
Conseguimos una cobertura de pruebas unitarias del 100% en los paquetes obligatorio1_DA1, businesslogic y dataccessfactory.

Hierarchy	Not Covered (% Lines)	Covered (% Lines)	Not Covered (Lines)	Covered (Lines)
ivanm_IVAN 2021-06-15 21_06_18.cover...	10,03 %	89,97 %	345	3094
businesslogic.dll	0,00 %	100,00 %	0	220
dataaccessfactory.dll	0,00 %	100,00 %	0	15
obligatorio1_da1.dll	0,00 %	100,00 %	0	729
repository.dll	33,44 %	66,56 %	219	436
unittestobligatorio1.dll	6,92 %	93,08 %	126	1694

No solo realizamos pruebas para llegar al 100% pero también para casos bordes o pruebas que nos parecieron relevantes implementar. También utilizamos DataTestMethod para probar diferentes valores sin tener que repetir código, como cada DataTestMethod solo se cuenta como un test, la cantidad total de test no necesariamente refleja el alcance o la cantidad de valores probados.

En repository no se cubrió al 100% debido a que las migration se generan automáticamente y no se genera a partir de TDD. En este paquete, tampoco se pudo probar la property autogenerada del get Item, que si se usa para crear las migrations. Esto hizo que el promedio general del test coverage bajara considerablemente.

repository.dll	33,44 %	66,56 %	219	436
Repository	0,23 %	99,77 %	1	432
Repository.Migrations	98,20 %	1,80 %	218	4



Last Modified

Para cumplir el requerimiento de que que al crear o modificar las contraseñas, estas tengan una fecha de cuando fue esta última modificación, se usó System.DateTime. Este es el encargado de setear el atributo de password, con la hora configurada en la computadora que está ejecutando la aplicación. Para poder utilizar TDD y comprobar que efectivamente podía cambiar esta fecha de una día para otra, nos vimos forzados a dejar el set de la property public, y poner una fecha antigua para que la comparación fuera efectiva.

Testing funcional

Este testing funcional refleja la aplicación al momento de la primera entrega. Decidimos dejar esta sección, debido a que creemos que refleja las acciones básicas de las aplicación.
Para el testing funcional, decidimos testear cada elemento de la presentación, probando todas las posibles combinaciones de botones, valores límite o vacíos en el caso de que haya que ingresarlos. También probamos que los mensajes de las excepciones realizados se mostrarán correctamente, ya que algunos eran muy largos por lo que tuvimos que acortarlos y manejar el tamaño de las ventanas para que se muestren de la forma esperada.

A continuación dejamos evidencia de los test que hicimos en alta, baja, modificar y listado de contraseñas, donde comenzamos testeando la parte del listado, viendo que ocurre en caso de utilizar las funcionalidades sin haber ingresado ninguna contraseña, luego agregando una y chequeando que no se puedan dejar campos vacíos. Seguimos con la modificación de la contraseña, donde modificamos uno de los atributos que la identifican, es decir el par (Site, Username), junto con otros y luego modificamos ambos atributos identificadores. Para terminar testeamos la parte de eliminar contraseña, sin haber seleccionado ninguna, luego seleccionandola y aceptando el popup, y por último utilizando la función no mostrar más del popup de confirmación.

A continuación dejamos el link al video. Cada parte está dividida por timestamps para permitir encontrarlas fácilmente.
https://www.youtube.com/watch?v=uiWTPmrKEJc

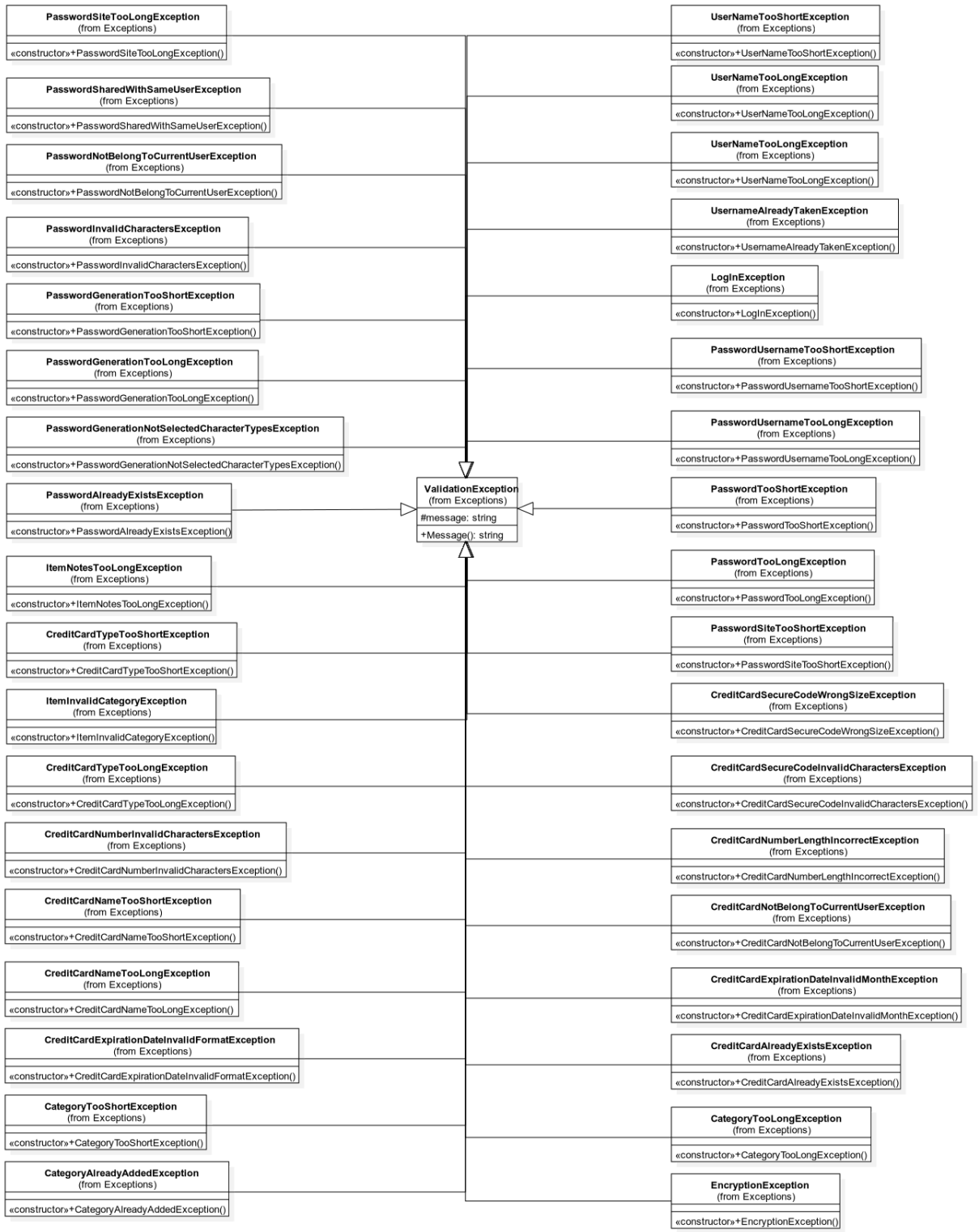
Instalación:

En el directorio principal, se encuentra la carpeta DataBaseScripts donde se encuentran dos backups de las base de datos, una de ella se encuentra sin datos (solo las tablas) y la otra tiene datos ya precargados. Se recomienda utilizar el Usuario Juana, ya que tiene varios ítems para probar todas las funcionalidades.

El ejecutable (.exe) se encuentra dentro de la carpeta Release dentro de Release.zip.
La única librería que se utilizó, fue para encriptación. Está documentado en la sección de encriptación. Para poder correrlo correctamente, se debe configurar el connection string. Para esto se debe entrar en la carpeta donde se encuentra el ejecutable y editar el archivo “Presentation.exe.config” suplantando el connection string por la conexión propia con la base de datos.

4. Anexo

1. Diagrama de Exceptions



[16] Diagrama encontrado en Documentation/Diagrams/EXCEPTIONS.SVG

2. Inconsistencia en Migrations

En la derecha se muestra la base de datos habiendo actualizado varias migrations secuencialmente, en la derecha, habiendo borrado las migrations anteriores y aplicando todos los cambios en una sola migration. Debido a esta inconsistencia, hubo momentos, donde se detectaron errores en las migrations y se borraron para poder aplicarlas todas juntas.