

# **Obligatorio 2**

## **Diseño de Aplicaciones 1**

**Grupo: M5B**

**Link repositorio:**

[https://github.com/ORT-DA1/240926\\_221025\\_247096](https://github.com/ORT-DA1/240926_221025_247096)

**Estudiantes:**

Santiago Diaz - 240926

Agustina Disiot - 221025

Joaquín Meerhoff - 247096



# Indice

<b>Descripción General:</b>	<b>3</b>
Bugs o Errores	3
Descripción general de los paquetes:	3
Git Flow	4
Clean Code	5
<b>Descripción y Justificación de diseño</b>	<b>6</b>
Explicación general:	6
Una sola ventana	6
Diagramas de secuencia:	6
Diagramas de clases	6
Diagrama de Interfaz	7
Diagrama de Negocio:	8
Diagrama de LogicaDeNegocio:	9
Diagrama de Repositorio:	10
Diagrama de relación excepciones	11
<b>Explicación decisiones diseño</b>	<b>12</b>
Negocio	12
LogicaDeNegocio	12
Repositorio	12
Data Breach	13
Para indicarle a un usuario si se ha modificado la contraseña desde que ocurrió el dataBreach, se utiliza la última fecha de modificación de la clave, y se compara con la fecha de creación del dataBreach.	13
Interacción Interfaz Visual con Negocio y Lógica de Negocio	14
Almacenamiento de datos	15
Sugerencias de mejora de contraseñas:	16
Encriptación de contraseñas	17
<b>Pruebas</b>	<b>19</b>
Pruebas Unitarias	19
Casos de Uso/Prueba	19
<b>GRASP, SOLID y Patrones de diseño</b>	<b>20</b>
GRASP:	20

# Descripción General:

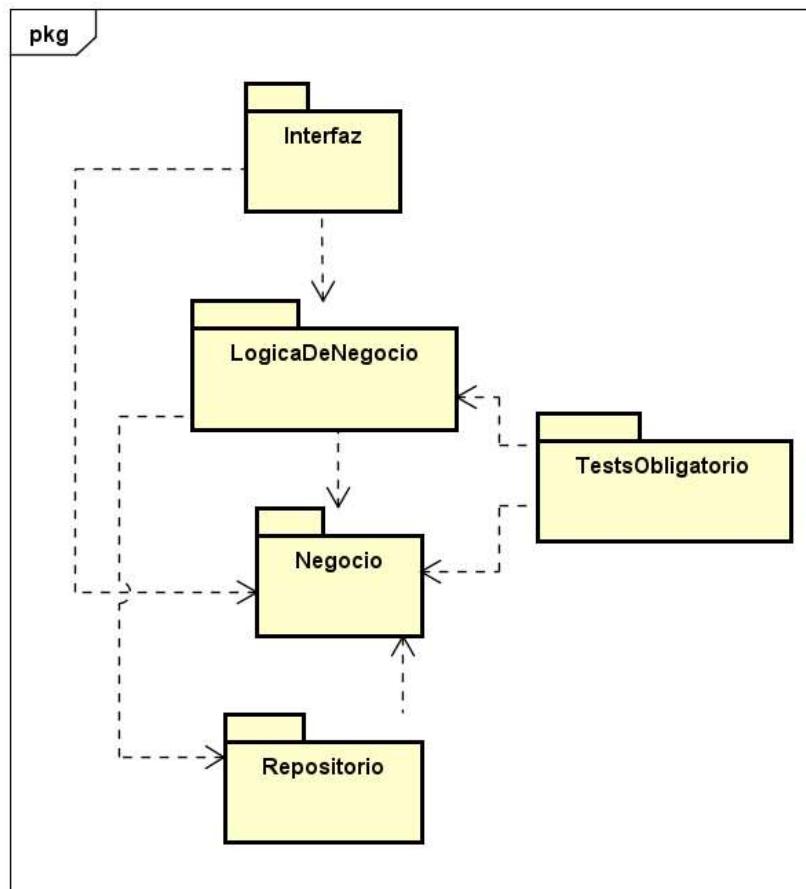
## Bugs o Errores

En este obligatorio pudimos cumplir con todos los requisitos pedidos para la aplicación, asegurándonos de no dejar ningún error de ejecución.

Algo que podría ser considerado un bug, es que en el caso de no tener instalado SQL EXPRESS, la aplicación crashea, en vez de avisar que se requiere tenerlo instalado, ya que asumimos que el usuario lo va a tener instalado.

## Descripción general de los paquetes:

Trabajamos con cinco paquetes, uno encargado de los tests, llamado TestsObligatorio, otro encargado de la lógica con el nombre “LogicaDeNegocio”, otro paquete, encargado de tener los datos, es el paquete de “Negocio”, el siguiente se llama “Repositorio” y es el que maneja la base de datos, y se encarga de tener los DataAccess y TypeConfigurations de todas las clases. El último encargado de la interfaz visual llamado “Interfaz”.



# Git Flow

Para trabajar en el repositorio de github con git, seguimos la siguientes reglas:

- En cuanto a la lógica, funcionalidad y TDD, no agregar más de una prueba por commit, para facilitar la lectura de commits en los que uno no estuvo presente.
- Ramas para cada funcionalidad, de forma que se puedan aislar posibles ingresos de errores, y además no borramos las ramas para que quede registro de lo realizado para la corrección del obligatorio. Estas ramas se juntaron por merge desde la línea comandos de git o por pull requests (Se usó este último principalmente cuando se agregaba algo que pudiera romper la aplicación.).
- Todas las ramas debían comenzar en minúscula y luego las siguientes palabras debían comenzar con mayúsculas.
- En develop no debíamos realizar commit, a no ser que fuera de un merge conflict que se estuviera resolviendo inmediatamente.

En este obligatorio, al tener muchas funcionalidades no por parte de lógica de negocio, ni de negocio, sino del funcionamiento de la base de datos, se realizaron varios cambios sin utilizar TDD, ya que las pruebas no podrían ayudar a agregar de a poco nuevas funcionalidades.

# Clean Code

A lo largo de esta entrega seguimos teniendo en cuenta las reglas de clean code para facilitar el entendimiento del código, aun así somos conscientes de que hubo un aspecto que no cumple con clean code.

Los aspectos que no cumplimos :

- Usamos el idioma español aun cuando se nos recomendó el uso del inglés ya que el primer obligatorio lo habíamos hecho en español y para ser consistentes y no tener que cambiar todo el obligatorio decidimos continuar usando el español.

Los aspectos que nos aseguramos de cumplir fueron:

- Tratar de mantener los parámetros de funciones y constructores en general en los mínimos posibles. Esto lo cumplimos creando clases “auxiliares” con el uso de contener de manera más ordenada los datos que se pasan entre objetos, ventanas, etc.

En algunos casos, no llegamos a hacer esto inmediatamente, sino realizamos refactors en otro momento cuándo nos dábamos cuenta de una forma más prolífica y que cumpliera las mismas funciones.

- Utilizar nombres legibles, por ejemplo usar contador, en vez de cnt.
- Utilizamos nombres descriptivos, por ejemplo en VerificadoraString utilizamos nombres largos pero descriptivos para los rangos de caracteres ASCII.
- No utilizamos comentarios: esto fue para que los nombres de las funciones o variables sean suficientemente claros para no requerir un comentario. Si una función comenzaba a precisar algo que la simplifique o explique, decidimos separarla en más funciones para que sea más legible.
- En el caso de nombres de variables y funciones, si una variable era privada de un objeto, utilizamos “\_” al principio para demostrarlo, luego las funciones las nombramos con mayúsculas, y se trató de mantener una consistencia en cómo se nombró a los diferentes atributos, variables, etc. Esta forma de nombrar los atributos privados fue recomendada por los profesores de tecnología antes del primer obligatorio.
- Nombres de variables redundantes: En casos de listas, se trató de no escribir por ejemplo una lista de categorías con el nombre “\_listaCategorias”, sino solo “\_categorias”.
- Para esta entrega también modificamos las pruebas para que tengan un TestInitialize y de esta forma evitar repetir código creando en un setUp los objetos necesarios para las pruebas.

# Descripción y Justificación de diseño

## Explicación general:

Para realizar los diagramas de paquetes y clases utilizamos el programa dado en clase llamado “Astah UML”. El archivo en el que se realizó esto está incluido en la carpeta de documentación del repositorio y los diagramas se van a incluir en formato png y svg en la entrega de gestión y en el repositorio para poder verlos en mejor resolución.

## Una sola ventana

Mantuvimos la idea de usar una sola ventana como uno de los principales aspectos del diseño de la interfaz con la excepción de la VentanaModificarClave que fue introducida para modificar una clave dentro de Data Branches y en Historico Data Branches. Además utilizamos otra ventana para las confirmaciones con pop ups de cancelar o confirmar o para una ventana de aviso que se creo o modiflico correctamente cierto objeto. Esto nos llevó a utilizar varios eventHandlers y delegate para poder transmitir entre paneles o de paneles a ventanas las distintas acciones que realiza el usuario. Utilizar una sola ventana nos evita tener que actualizar ventanas iguales o tener ventanas abiertas repetidas.

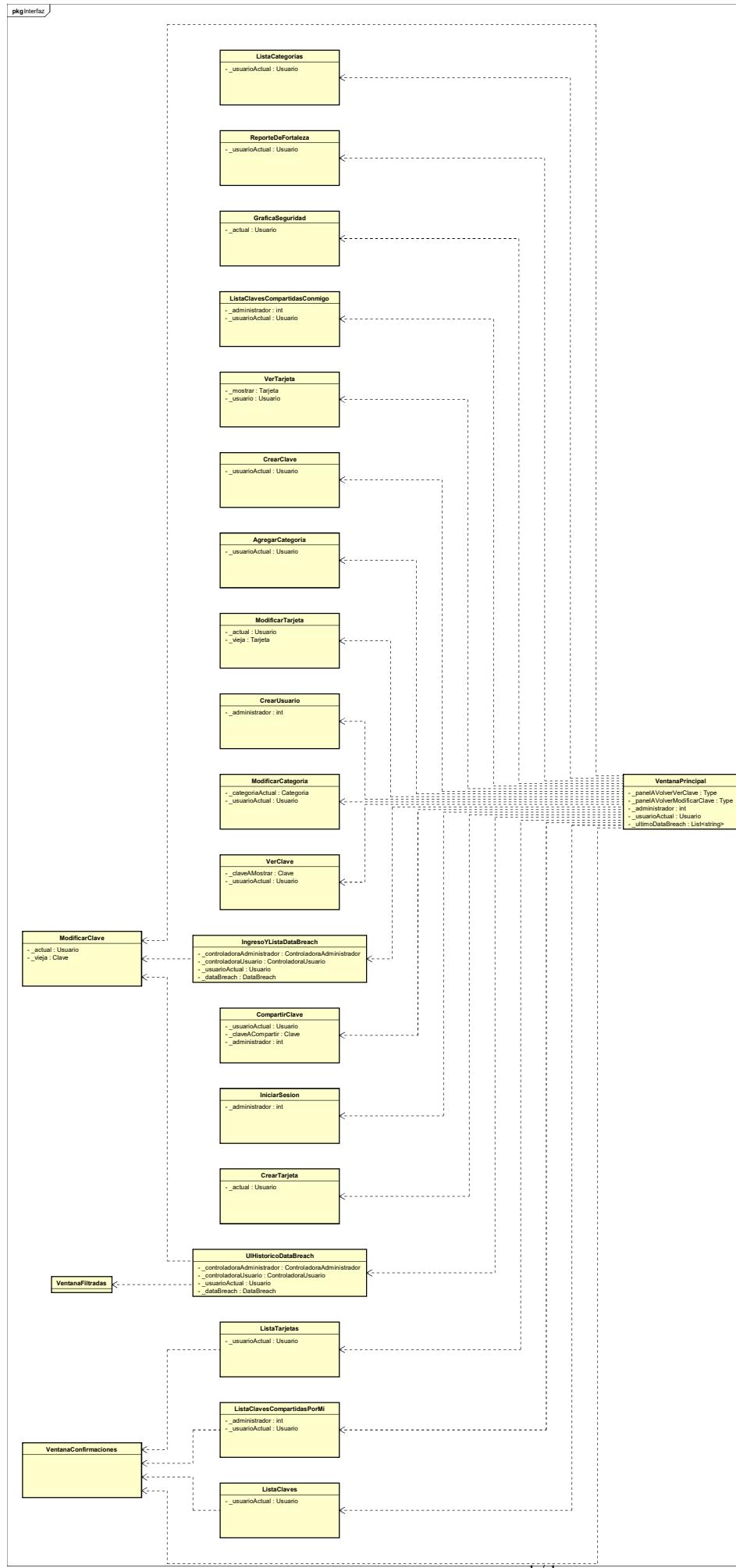
## Diagramas de secuencia:

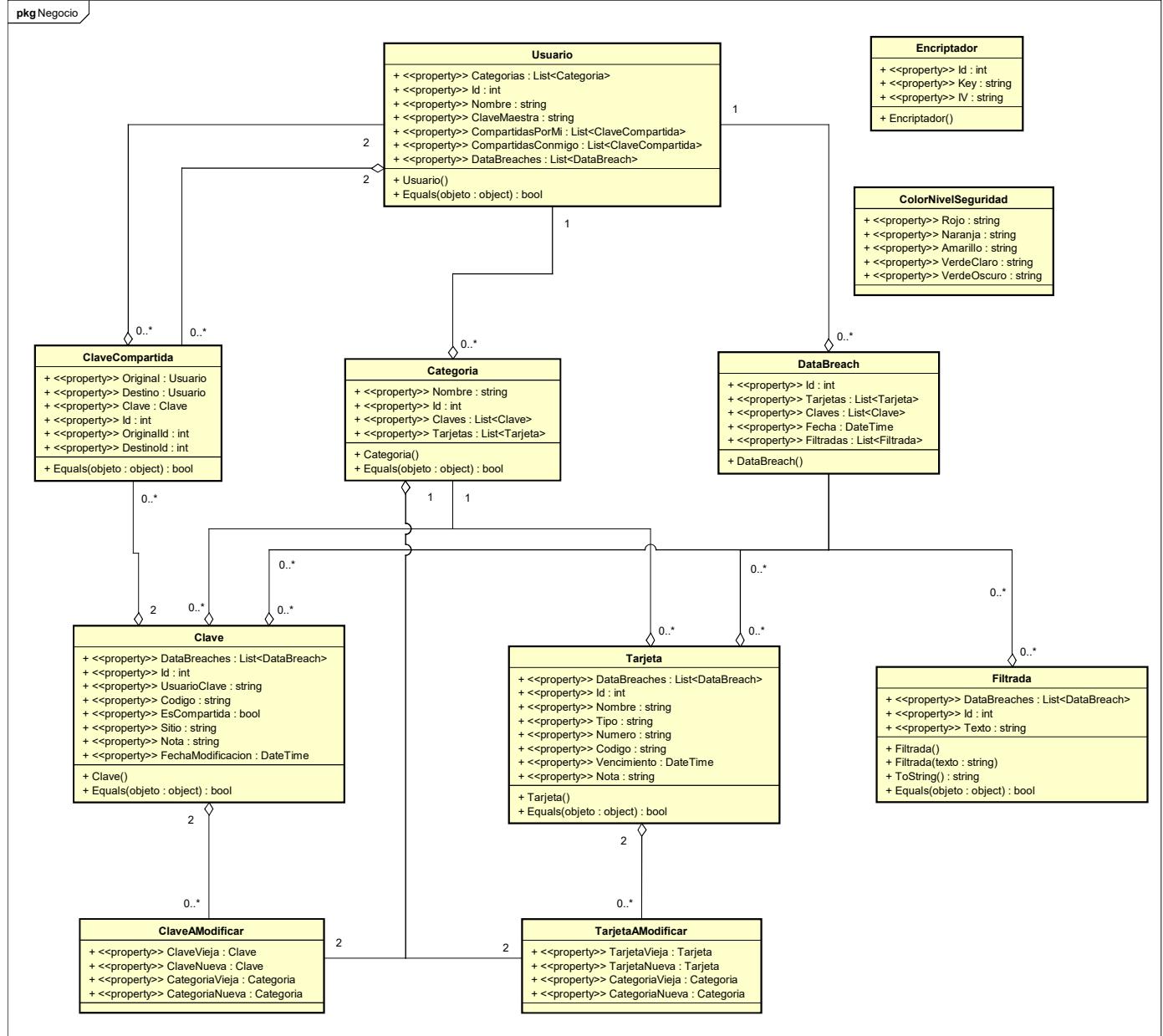
Se van encontrar 3 pdfs de diagramas de secuencia, con comentarios dentro y junto al diagrama que explican el funcionamiento de:

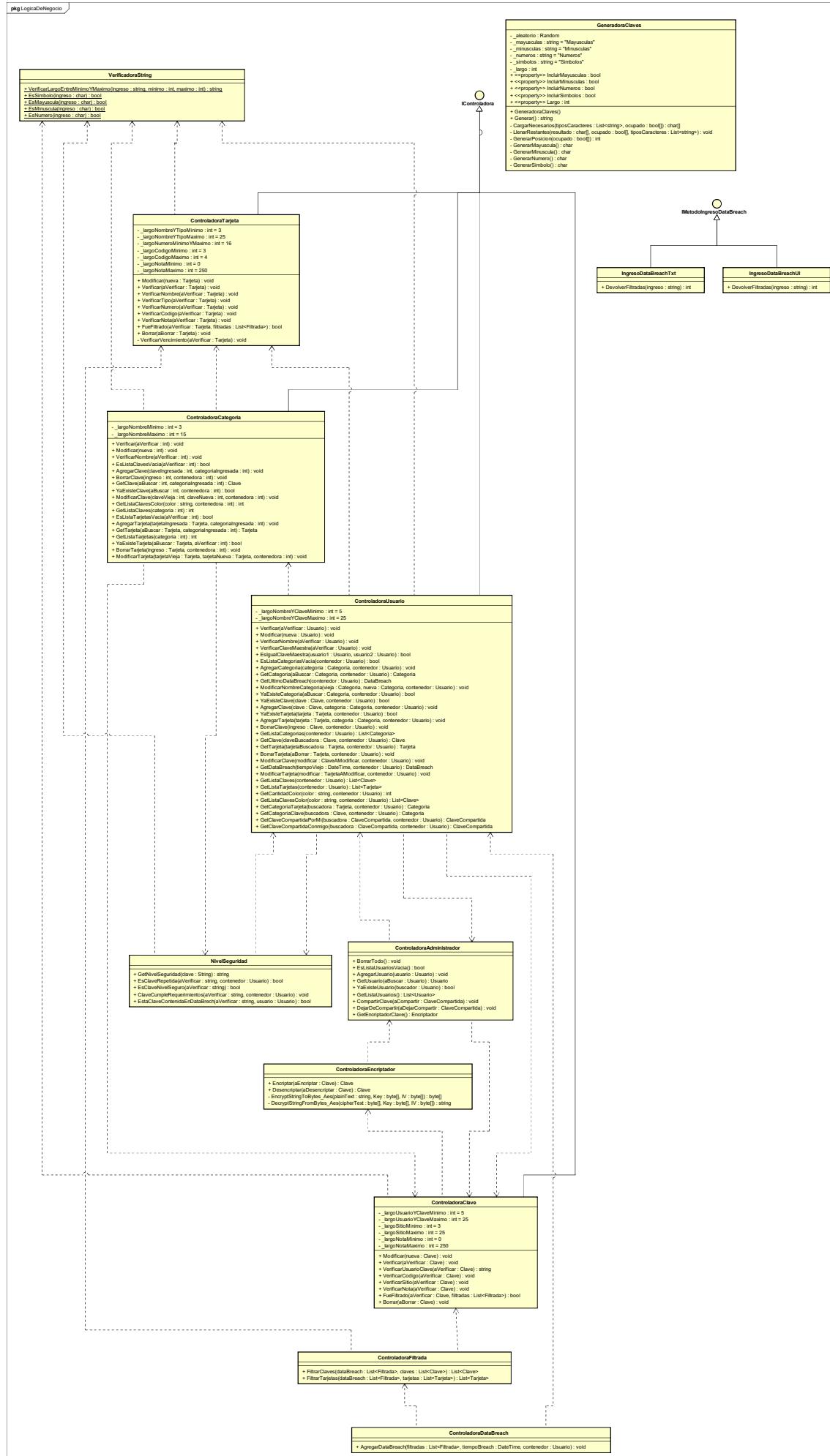
Filtrar las claves de un usuario, agregar un usuario, y modificar una tarjeta.

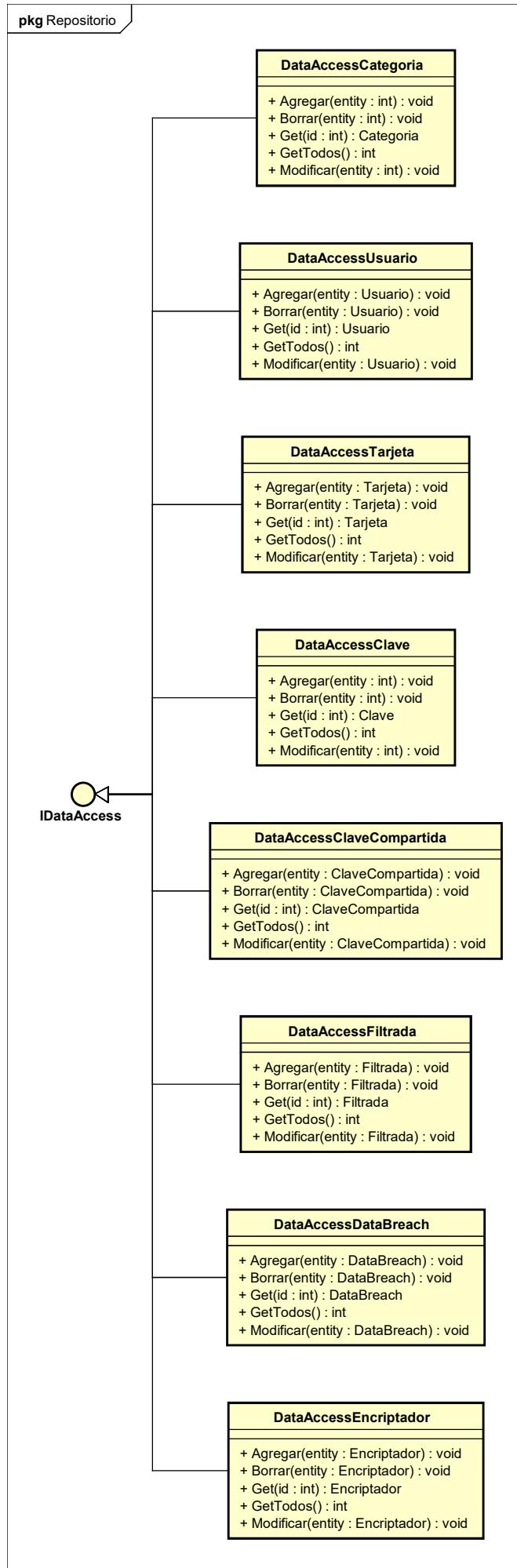
Elegimos estas funcionalidades ya que presentan diferentes implementaciones y partes del código.

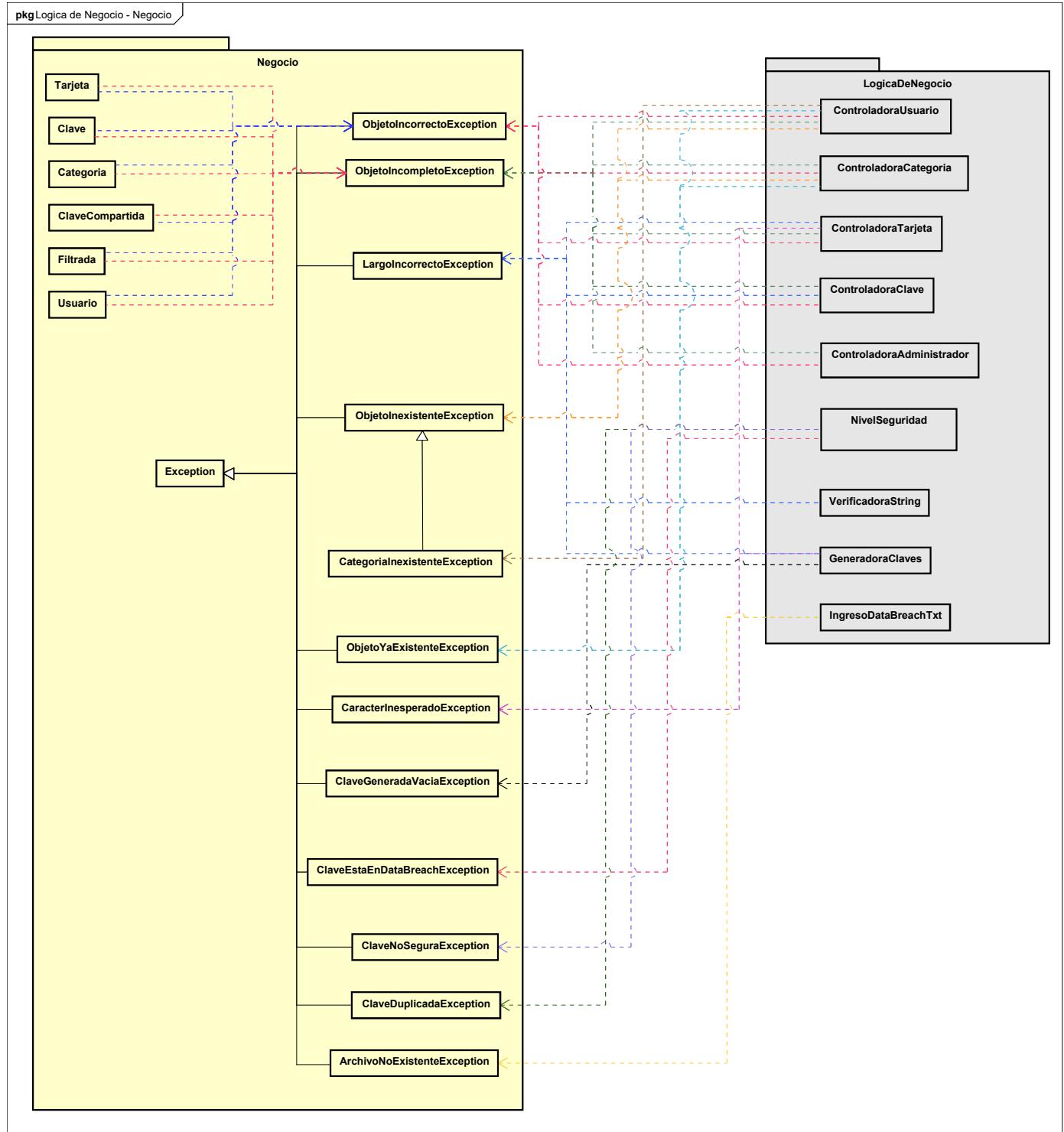
## Diagramas de clases











# Explicación decisiones diseño

Para este obligatorio hicimos un gran refactor para separar la lógica de negocio de sus clases, separando el paquete, anteriormente llamado Dominio, en Negocio y LogicaDeNegocio.

## Negocio

Este paquete solo se utilizó para contener las clases del dominio con su respectivos `toString` o `Equals` además de otras clases creadas utilizadas para aumentar la cohesión como por ejemplo `ClaveAModificar`, `ClaveCompartida`, etc. También contiene la carpeta de todas las excepciones personalizadas que utilizamos.

## LogicaDeNegocio

Este es el paquete donde está todo el código de la lógica, se comunica con el Negocio que contiene todos los atributos y propiedades de las clases, y a través de métodos guarda, elimina, modifica, pide o verifica los datos llamando al paquete Repositorio que contiene los métodos de la base de datos.

Todos sus métodos fueron realizados usando TDD y fueron un refactor del paquete “Dominio” del obligatorio anterior.

Para poder fácilmente reconocer estas adiciones, realizamos cada función en una nueva rama y cada commit incluía una sola prueba. De esta forma, en caso de haber modificado algo no deseado, se puede volver atrás o reconocer el error más fácilmente.

## Repositorio

Este es el paquete que se comunica directamente con la base de datos. Acá es donde están todos los `DataAccess` y `TypeConfigurations` de todas las tablas de la base de datos con sus respectivos métodos para guardar, eliminar, obtener o modificar los distintos datos de las tablas.

# Data Breach

A partir de la implementación de databreaches en el obligatorio 1, tuvimos que reestructurar bastante código, ya que no se había hecho con la idea de guardar un historial. Movimos métodos desde la interfaz visual a la lógica de negocio, y eventualmente separamos en un par de clases/controladoras. Cada una de estas encargada de una parte de los databreaches, de forma que cada una sea una abstracción del funcionamiento.

Tenemos la interfaz `IMetodoIngresoDataBreach`, que indica que a partir de un ingreso de tipo genérico `T`, se debe devolver una lista de `Filtradas`, que son prácticamente cada `string` separado individualmente de las posibles claves o tarjetas que se pudo haber filtrado.

El resultado de ese método, luego se le pasa como parámetro a `ControladoraDataBreach`, que además recibe la fecha, y el usuario al que se le agregara el `databreach`.

Luego la controladora de filtradas se encarga de encontrar entre las tarjetas y claves del usuario, las que están contenidas en la lista de filtradas. A las tarjetas, cualquier combinación de 16 números se los considera como un posible `DataBreach`, sin importar donde y cuantos espacios tenga. Esto lo programamos de esta forma, para que si el `dataBreach` no está formateado de una manera “prolija”, aun se pueda encontrar una posible tarjeta filtrada y avisarle al usuario de este caso.

Este forma de implementación favorece la extensión con nuevos métodos de ingreso, como descargas de páginas web, u otros medios, ya que solamente requieren el extender la interfaz `IMetodoDeIngresoDataBreach` y que respeten a la misma. Luego donde sea que se utilice, debe de pasarle a la `controladoraDataBreach` su resultado y no se debe modificar código.

Para indicarle a un usuario si se ha modificado la contraseña desde que ocurrió el `dataBreach`, se utiliza la última fecha de modificación de la clave, y se compara con la fecha de creación del `dataBreach`.

## Interacción

Al iniciar la aplicación se crea una nueva “VentanaPrincipal” donde se encuentran todos los event handlers.

Al crear un usuario, tarjeta, categoría, etc se pasa la clase de Negocio que contendría al nuevo objeto y se llama al método correspondiente de LogicaDeNegocio donde por cada clase de Negocio se tiene su controladora, en el caso de una clave o tarjeta o categoría se pasa el usuario actual además del nuevo objeto en sí.

De este modo se puede utilizar en los distintos paneles los métodos de la controladora de clase para agregar, eliminar, modificar, u otras funciones directamente. En sí, la interfaz trabaja pasándose por referencia los diferentes objetos entre los paneles y ventanas.

En algunos casos, para poder pasar de un panel a otro con menor cantidad de atributos en los constructores, utilizamos delegates que permiten pasarse en los event y eventhandlers parametros utiles de panel a panel.

# Almacenamiento de datos

A diferencia de la primera entrega ahora los datos si son persistentes, es decir los datos guardados van a ser almacenados en la base de datos sin necesidad de clickear en un botón “salvar estado” o de cerrar la aplicación.

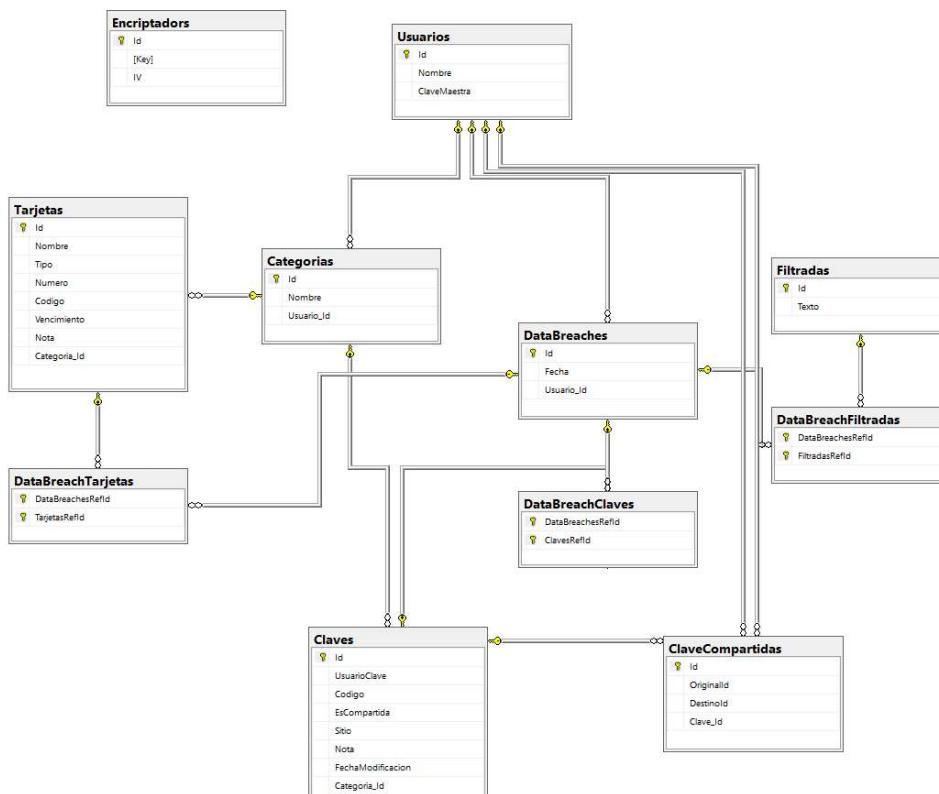
Los datos a guardar, modificar o eliminar se pasan como objetos y se guardan en la base de datos a través de los métodos de clase.

Por ejemplo, a la hora de crear un usuario se crea un objeto de clase Usuario y usando un método de la clase ControladoraAdministrador en el paquete LogicaNegocio, se agrega el usuario a la base de datos.

El único paquete que establece una conexión o que realiza los “SaveChanges” de entity framework al SQL Express es el de repositorio.

El siguiente diagrama muestra las relaciones de las tablas creadas por entity framework, y el archivo de migration que se encontrara en la rama de master.

Para poder hacer funcionar esto, se tuvo que agregar atributos como listas a los elementos de negocio,



## Sugerencias de mejora de contraseñas:

Estas sugerencias como en la letra es bastante ambiguo decidimos implementarlas como nuevos requisitos a la hora de crear o modificar una contraseña por lo que si no se cumple alguna de las tres la contraseña no será creada o modificada y se le avisará al usuario a través de un mensaje en la ventana la razon de por que no es aceptada.

Estos tres nuevos requerimientos son: El código de la contraseña no puede aparecer en un data breach conocido, el código de la contraseña no puede estar duplicado en otra contraseña de un mismo usuario y la contraseña tiene que tener un nivel de seguridad de "color verde claro" o mayor, es decir que el código la contraseña como mínimo tiene que tener un largo mayor a 14 y tener, ambas, mayúsculas y minúsculas.

Estos nuevos requerimientos fueron todos creados en el código a través de TDD al igual que todos los métodos de lógica de negocio del obligatorio.

El resto de los requerimientos para la creación y modificación de una contraseña implementados en el obligatorio 1 siguen estando presentes.

# Encriptación de contraseñas

En esta entrega las contraseñas que se desean guardar en la base de datos son encriptadas antes.

La librería que utilizamos es la de AES de microsoft que es capaz de encriptar y desencriptar string, utilizando arrays de bytes como el valor Key de un Aes, y otro array de bytes en el valor IV o vector de inicialización. Para poder utilizarlo fuera de un ejemplo y guardar los métodos de encriptación y desencriptación, creamos una tabla que guarda la key y el IV para reutilizarlos y poder seguir su uso, incluso después de cerrar el programa.

Nuestra solución encripta los códigos de las contraseñas de páginas y aplicaciones, además de la clave maestra del usuario. Ejemplo de encriptación de las claves maestras:

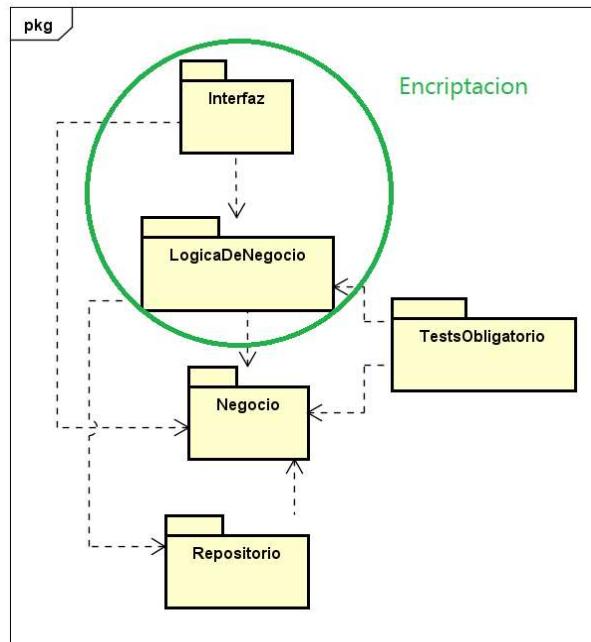
Sin encriptar y encriptadas:

	Id	Nombre	ClaveMaestra
1	60	Joaquin	JM12345
2	61	Agustina	A#45PastelPink
3	62	Santiago	12345SD
4	63	Pedro	ClaveMaestra
5	64	Roberto	12345ABCD
6	65	Agustin	1234566

	Id	Nombre	ClaveMaestra
1	7	Joaquin	d5M/3gKo+UkyMJCdyLVyuw==
2	8	Agustina	No0kp34Fn/+ASG/S6TFoFA==
3	9	Santiago	Bvbqiy95PDGqS1zMUIdpWw==
4	10	Pedro	SBL5NqFw6PSyN5GcnSPeRg==
5	11	Roberto	gqpj5y5mI2cE0mPy6lcYZA==

Instrucciones de la libreria utilizada:

(<https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.aes?view=net-5.0>)



En la imagen anterior muestra un círculo a qué altura realizamos la encriptación, o sea en la interfaz y lógica de negocio. En un principio pensábamos que sería la manera más “realista”, ya que el sistema nunca se enteraría del código y la encriptación ocurriría inmediatamente después de leer lo escrito por el usuario. Ahora consideramos que quizás una mejor solución sería hacer un paquete dedicado a la encriptación y que solamente se encripta en la lógica de negocio, pero por temas de tiempo, no pudimos hacer este cambio. Otro cambio que no logramos realizar pero que creemos debería hacerse por temas de seguridad, es no utilizar la misma encriptación para todos los usuarios y claves, ya que si se rompe esta, se filtraría toda la información. Intentamos realizar esto, pero el hacerlo dificultaba el compartir claves, ya que si el método de desencriptación era la clave maestra del usuario, no se podría desencriptar las claves compartidas a noser que cada vez que se comparta, se duplique una clave encriptada por la claveMaestra del otro usuario.

La solución a la que llegamos para encriptar respeta el principio de abierto cerrado en parte, ya que se puede modificar el método de encriptación, la library o plugin que se utilice para encriptar y no se debería de tener que modificar más código de lo necesario. Esto significa que estaríamos dejándolo con bajo acoplamiento a la implementación.

De todos modos, hay un cierto grado de acoplamiento por la forma en que realizamos nuestra implementación. De ser posible, se debería modificar las pruebas TDD para que se realicen con la encriptación, y decidir un punto en la lógica de negocio en específico para encriptar todas las claves, ya sea a la altura de la controladora de usuario, o abajo en la controladora de clave.

# Pruebas

## Pruebas Unitarias

La totalidad de nuestro código fue hecho a través de tdd llegando a una cobertura de pruebas unitarias de 83%. Pero esto es porque el paquete Repositorio solo tiene un 56% de cobertura ya que no se prueban ninguno de los Type Configurations ni las Migrations además no se usan todos los métodos en todos los DataAccess.

Aun así la cobertura de la Lógica de Negocio es de 94.74% que es donde están todos nuestros métodos de funcionalidad.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
└ Santi_DESKTOP-QNG5ACI 2021-0...	1190	16.54%	6006	83.46%
└ logicadenegocio.dll	73	5.26%	1316	94.74%
└ negocio.dll	35	10.87%	287	89.13%
└ repositorio.dll	976	43.73%	1256	56.27%
└ testsobligatorio.dll	106	3.26%	3147	96.74%

## Casos de Uso/Prueba

En esta sección escribimos pruebas que se podrían realizar a partir de un sistema cargado con las claves que se mencionan en el documento del anexo.

Anexo: Casos de uso

<https://docs.google.com/document/d/1kKT7cfR9ikZJUfXrX6Y9f3FomjJIwQ2yDgikToBjL7A/edit?usp=sharing>

Además en el repositorio y en la entrega, se debería encontrar un zip que contenga backups de la base de datos que realizando los pasos mostrados en clase, se debería poder levantar un backup de la base datos y tener guardadas a las claves, tarjetas, categorías, usuarios ,etc.

Para poder crear estos casos, utilizamos la rama “casosDePrueba” que no sera mergeada a master ni develop, ya que su uso primero borra toda la base de datos y luego la llena con los casos de prueba.

Este zip que se menciona contiene dentro una carpeta “Completo” y otra “Vacia”, cada una conteniendo un backup. No se puede renombrar el backup antes de cargar la base de datos.

# GRASP, SOLID y Patrones de diseño

## GRASP:

**Information expert:** En nuestro obligatorio aplicamos este concepto al tener para las clases de negocio una simetría por el lado de lógica de negocio, al tener una controladora de dicha clase. Esto está presente en por ejemplo: la controladora de clave, tarjeta, databreach, etc.

**Creator:** Si una clase del negocio contiene listas, su controladora se encarga de crear y agregar a la instancia nuevos elementos. Por ejemplo, las categorías reciben nuevos elementos en la controladoraCategoria.

**Alta cohesión:** Dividimos responsabilidades por controladoras, y funciones, como databreaches, filtradas, etc, para evitar la dificultad de entender el programa.

**Polimorfismo:** Tenemos los ingresos de databreaches y los dataaccess para que facilite en un futuro la creación de nuevos métodos al programador

**Pure Fabrication:** Creamos clases artificiales como generadora clave y verificadora string que sirven para aumentar la cohesión del código. Estas le brindan servicios comunes a las otras o sirven para abstraer funcionamiento para facilitar la lectura del código.

**Protected variations:** Al haber utilizado IDataAccess, la interfaz para todos los accesos a la base de datos, si en algún futuro se quisiera agregar nuevos métodos, se puede agregar la interfaz para forzar su implementación en cada una de las que la extienden. Además, se tiene un template fácil para agregar nuevas clases con acceso a base de datos.