Obligatorio 1

Diseño de Aplicaciones 1

Grupo: M5B

Link repositorio:
https://github.com/ORT-DA1/240926_221025_247096

Estudiantes:
Santiago Diaz - 240926
Agustina Disiot - 221025
Joaquín Meerhoff - 247096



Indice

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.

Descripción general de los paquetes:

Trabajamos con tres paquetes, uno encargado de los tests, llamado TestsObligatorio, otro encargado de la lógica con el nombre Dominio y el último encargado de la interfaz llamado Interfaz.
Mas adelante nos dimos cuenta que podíamos haberlo hecho más prolijo usando más paquetes como se explica mas abajo en la documentación, en la sección de Explicación decisiones de diseño.

Git Flow

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

- En cuanto a dominio 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 linea comandos de git y no por pull requests.
- En un principio nuestra rama de trabajo se llamaba “testing” y luego pasamos al modelo que se usa en clase, de “develop”.
- 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.

Clean Code

A lo largo de esta entrega tuvimos en cuenta las reglas de clean code para facilitar el trabajo futuro sobre el proyecto, aun así somos conscientes de algunos aspectos de diseño que no cumplimos por completo por falta de tiempo.

Los aspectos que no cumplimos están más especificados más adelante en la parte de diseño de código.

Los aspectos que nos aseguramos de cumplir fueron:

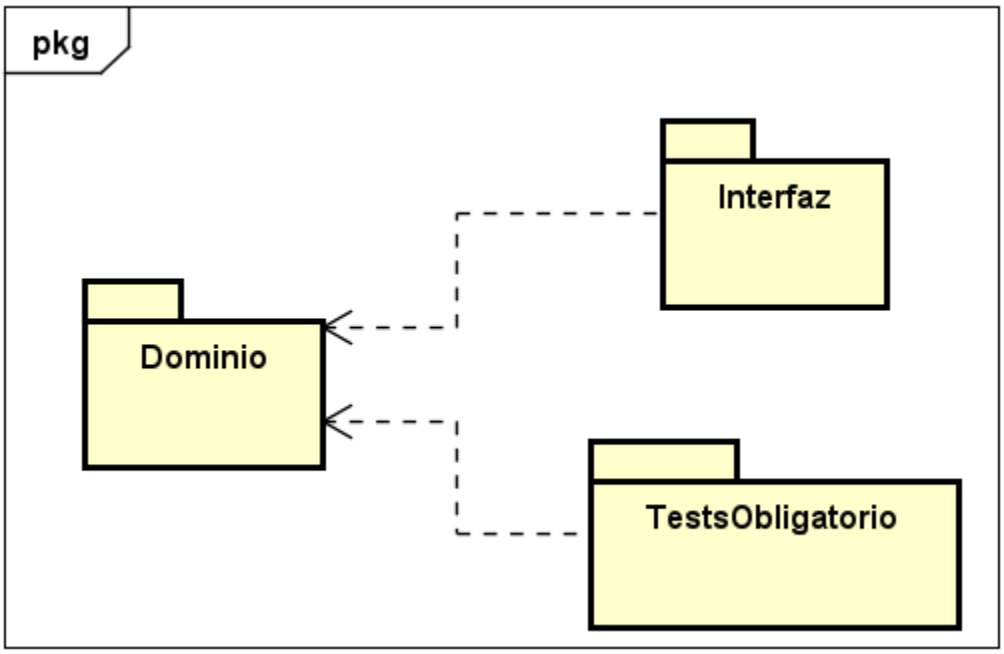
- Tratar de mantener los parámetros de funciones y constructores en general en los mínimos posibles. Esto lo cumplimos creando clases “auxiliares” con el uso de contener de manera más ordenada los datos que se pasan entre objetos, ventanas, etc.
  En algunos casos, no llegamos a hacer esto inmediatamente, sino realizamos refactors en otro momento cuándo nos dábamos cuenta de una forma más prolija 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.
- 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”.

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 incluído 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.

Diagrama de paquetes



Diagramas de clases

Diagrama de Interfaz

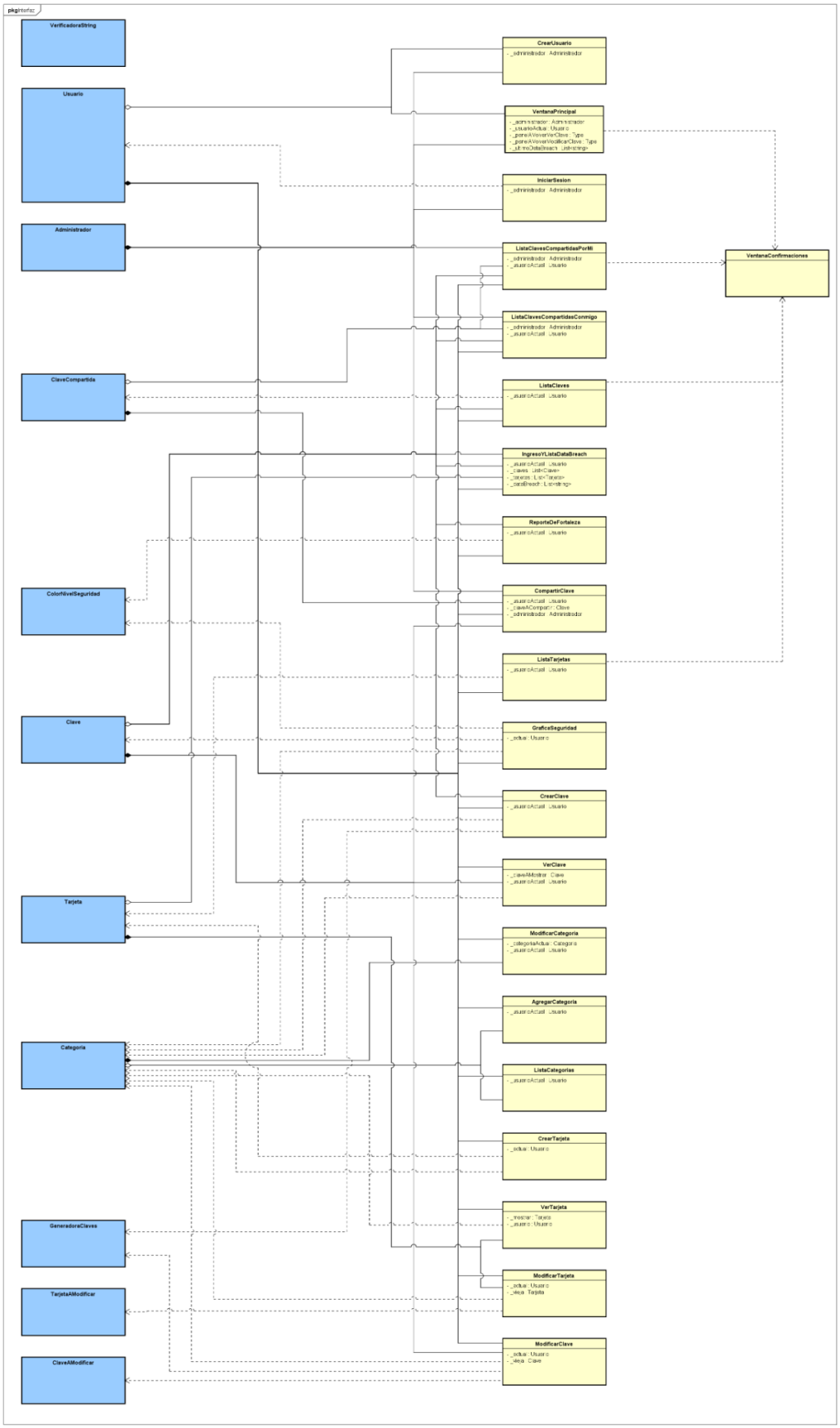


Diagrama de Dominio:

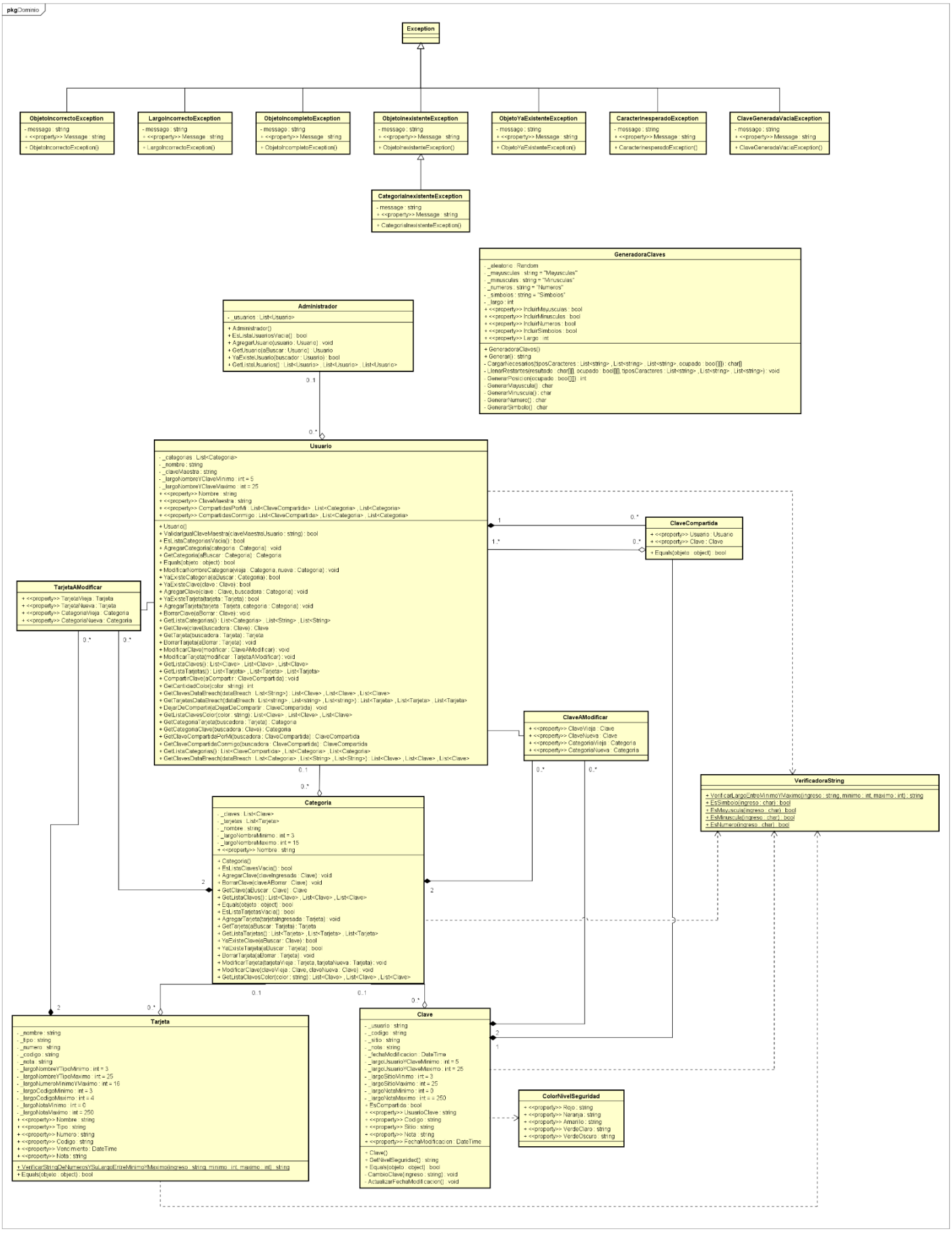
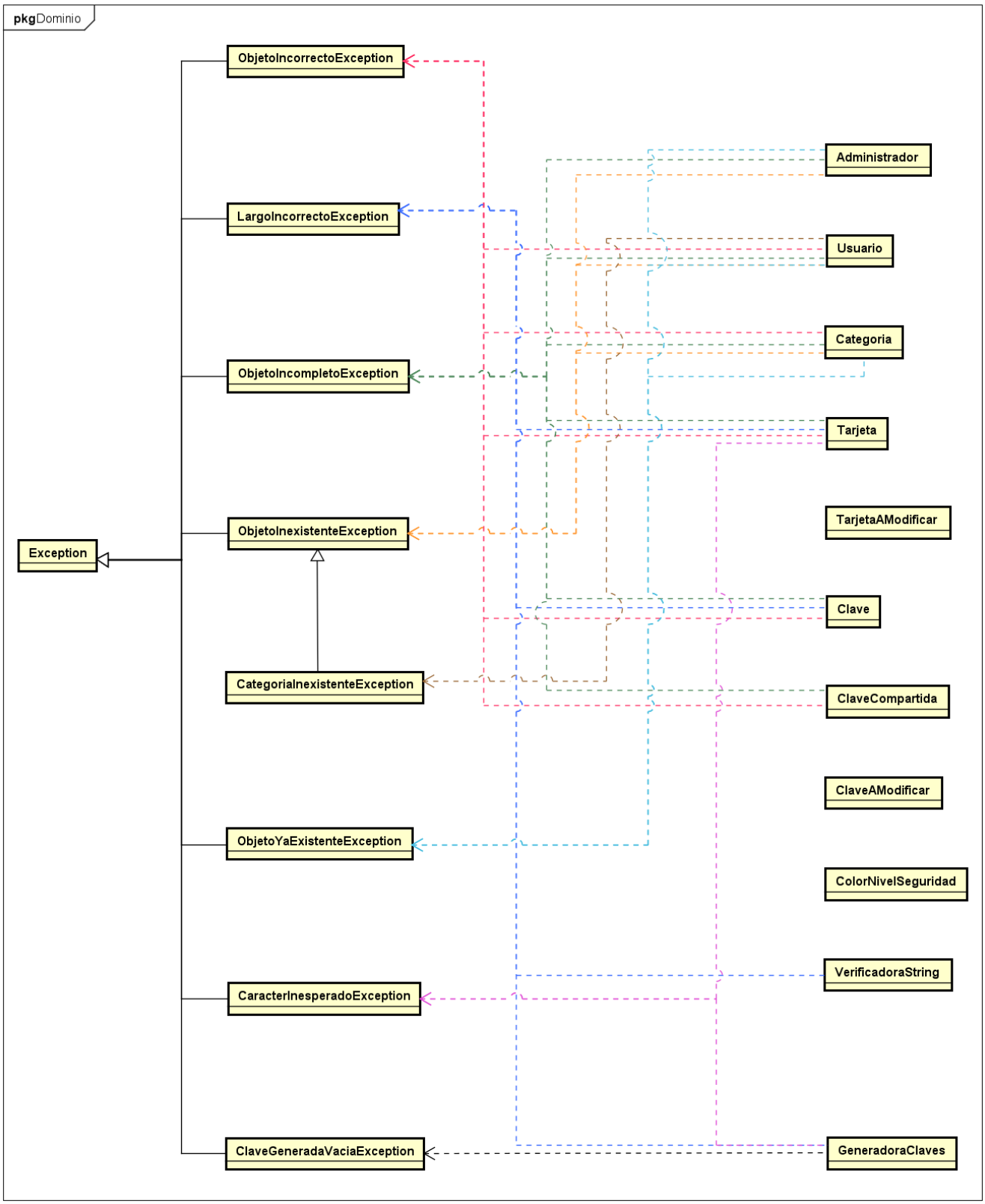


Diagrama de relación excepciones en dominio



Explicación decisiones diseño

Dominio

El dominio, el paquete donde está contenido todo el código de la lógica, lo diseñamos a partir de los más chico a lo más grande.

En un principio lo pensamos de la siguiente manera: Un administrador contiene una lista de usuarios, cada usuario contiene una lista de categorías, y cada categoría contiene una lista de tarjetas y/o contraseñas (en la clase llamadas Clave) con sus respectivos atributos.

Usando TDD comenzamos pidiendo datos de las clases más chicas desde las más grandes, y una vez que comenzamos el TDD de una clase más chica, no volvíamos a la más grande hasta que las por debajo tengan sus métodos.

Ejemplo orden:

Administrador -> Usuario -> Categoría -> Contraseña -> Administrador …

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.

En nuestro diseño encontramos varios errores que al momento de darnos cuenta no nos daba el tiempo suficiente para cambiarlos.
Algunos de estos errores fueron:

- Usar el set para verificar: en el mismo set llamamos a la función que verifica el parámetro en vez de crear el objeto y luego llamar a la función verificadora aparte.
- Separar responsabilidades: En general hicimos pocos objetos con demasiados usos, deberíamos haber hecho paquetes de verificación o por lo menos separar en diferentes clases los métodos objetos.
- No usar excepciones más ,específicas las excepciones específicas heredarían de las excepciones que si realizamos.

Una sola ventana

Uno de los principales aspectos del diseño de la interfaz que tuvimos en cuenta al crearla fue el mantener toda la interfaz en una sola ventana. No utilizamos otra ventana a no ser para las confirmaciones con pop ups de cancelar o confirmar. 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.

Data Breach

En la sección de data breach, tomamos en cuenta la posibilidad de futuras adiciones a la funcionalidad para no tener que reescribir el código de cero. Por este motivo, separamos en dos funciones la búsqueda de contraseñas o tarjetas contenidas en data breaches.

Actualmente, la clase Usuario contiene la función “GetClavesDataBreach” y “GetTarjetasDataBreach” que reciben una lista de strings, y comparan si cada contraseña y/o tarjeta del usuario están contenidas en la lista del data breach.
En el caso de las contraseñas simplemente se usa Linq y se devuelve una lista con punteros a los objetos de tipo Clave que cumplan la condición de su atributo código estar contenido en la lista de strings.

En el lado de las tarjetas, se elimina de los strings todos los que no están compuestos solamente por números. Esto se realiza con la clase VerificadoraString y su método EsNumero que verifica que su valor ASCII esté dentro del rango de los números.
Además, un aspecto que tuvimos en cuenta desde un principio fue que uno no puede estar seguro como en un data breach en internet se vaya a formatear los números de las tarjetas. Por este motivo si al quitarle los espacios, una posible tarjeta filtrada contiene solamente números y es de largo 16, se agrega a la lista de las que se tienen que controlar.

En la interfaz uno puede modificar una contraseña filtrada y cuándo se termina de modificar aún se tiene el data breach que se haya escrito en el inputbox, para evitar tener que copiar y pegar el mismo contenido.

Las tarjetas no pueden ser modificadas por la interfaz ya que una tarjeta filtrada en la realidad lo único que se podría hacer es hablar con el banco para que se cancele o que por lo menos no pueda seguir siendo utilizada.

Interacción interfaz con dominio

Las ventana principal al ser creada crea un nuevo administrador (en un futuro si se trabaja con persistencia, aquí es donde controlaremos eso).

Al crear un usuario, tarjeta, categoría, etc se pasa la clase que contendría al nuevo objeto, en el caso de una clave o tarjeta o categoría, se pasa un usuario, en el caso de un usuario el administrador, etc.

De este modo se puede utilizar en los distintos paneles los métodos de la 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

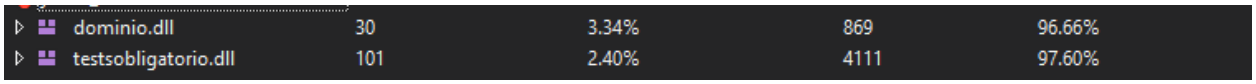
En esta primera instancia del obligatorio los datos no son persistentes, es decir que al cerrar la aplicación se borran los datos. Actualmente los datos a guardar, modificar o eliminar se pasan como objetos y se guardan a través de los métodos de clase.

Por ejemplo a la hora de crear un usuario se crea un objeto de clase Usuario y usando un método de la clase Administrador, se agrega el usuario a una lista.

Pruebas

Pruebas Unitarias

La totalidad de nuestro código fue hecho a través de tdd llegando a una cobertura de pruebas unitarias de 96,66%.



dominio.dll	30	3.34%	869	96.66%
testsobligatorio.dll	101	2.40%	4111	97.60%

Casos de Uso/Prueba

Video de las pruebas de contraseña (agregar, modificar, eliminar, etc):
https://youtu.be/AoSrtzf2x44

Anexo: Casos de uso
https://docs.google.com/document/d/15YyfX9Y23iC5ObhM1GwqnQNl_bUCV06g4Lxivm2Vhdc/edit?usp=sharing