Universidad ORT Uruguay

Licenciatura en Sistemas

Diseño de Aplicaciones I - Obligatorio 2

Micaela Felder 212322

Paula Hernandez 201886

Sara Vila 163497

Repositorio:
https://github.com/ORT-DA1/212322-201886-163497-oblig1.git

Índice

1. **Descripción general del trabajo y del sistema**

El objetivo de esta aplicación es poder llevar un control de las finanzas personales.

El sistema cuenta de distintas secciones dentro de las cuales se pueden realizar distintas acciones. Las secciones se dividen en: categoría, moneda, gasto común, gasto recurrente,presupuesto, reporte de gasto y reporte de presupuesto.

Las categorías son ingresadas por el usuario, a las mismas se les puede asignar palabras clave que van a identificarlas.Por ejemplo: una categoría podria ser entretenimiento y sus palabras clave cine, deporte, salida, entre otras. También se cuenta con la posibilidad de eliminar y modificar dichas palabras clave. Las categorías y palabras clave van a formar parte del registro de un gasto.

El sistema permite crear monedas con su respectivo nombre, símbolo y cotización. Esto también se incluye en el gasto ya que, por ejemplo, cuando una persona realiza un pago se realiza en cierta moneda. A su vez es posible modificar y eliminar las monedas.

Otra funcionalidad es la de crear gastos, existen dos tipos de gastos a crear, los gastos comunes y gastos recurrentes. Por gastos comunes entendemos que son gastos que se realizan en la vida cotidiana, como por ejemplo una compra en el super, una salida al cine, entre otros. Para ingresar un gasto de este tipo es necesario ir a la opción de agregar, en la misma se ingresa una descripción y se despliegan el resto de los datos a rellenar, la plataforma tiene una buena usabilidad por lo tanto es intuitiva, el usuario se dará cuenta cuales son los pasos a seguir.

Los gastos recurrentes son gastos fijos en una fecha preestablecida y que se dan repetidamente. El usuario ingresará un día del mes correspondiente. Por ejemplo: el día del mes en que se realizará el pago de la luz.
Este tiene varios componentes en común con el gasto común.

Otra funcionalidad es la de planificar presupuestos. Para esto el usuario indicará una fecha para crear el presupuesto, entendemos por presupuesto un monto que se estima gastar para cada categoría en un mes determinado. Por ejemplo, se planifica gastar $2000 en gastos correspondientes a la categoría “Casa” en marzo del 2020.

El usuario también tiene la opción de realizar un reporte de de gastos y de presupuestos.

¿En qué consiste?

En el reporte de gastos es posible visualizar con mayor perspectiva cuáles fueron mis gastos realizados en un mes y año en específico. Se pueden ver a partir de una tabla detallada con fecha, categoría, monto y moneda, así como de una gráfica en la que se muestra lo acumulado en cada día del mes. Otra recurso es la de guardar este reporte en un archivo en el escritorio, en un .csv, .txt o .xml. También se puede ver el total del mes en la moneda por defecto, $(pesos).

Así mismo el reporte de presupuestos presenta la posibilidad de poder consultar cuales presupuestos fueron registrados. A partir de la elección de la fecha de un presupuesto obtenemos la información de la categoría para la cual se realizó un presupuesto, el presupuesto planificado y el real y la diferencia entre ambos.

Errores conocidos:

- Al iniciar el programa desde Visual Studio el paquete **Interfaz De Usuario** no esta como proyecto de inicio ya que debido a la configuración de Git Ignore no hemos logrado dejar esa configuración establecida.

- La forma de modificar una moneda y un gasto recurrente quedó complejo, con código repetido debido a que cambiamos cada atributo con un método distinto. Hubiera sido mejor implementarlo de la misma forma que hicimos el método ModificarGastoComún, al cual se le pasa por parámetro un gasto con los nuevos atributos y en la base de datos se busca ese gasto a través del id.
  Cuando lo encuentra, se actualiza la moneda y las categorías con attach y se setean el resto de los atributos. De esta forma hemos aprendido una forma de hacerlo más simple y corta que a nivel de ventanas brinda una mejor experiencia para el usuario ya que tiene menos botones y recibe un solo parámetro lo que es mejor según lo estudiado de CleanCode.

- La clase Categoría tiene una lista pública de palabras claves y la clase Presupuesto tiene una lista pública de CategoriaMonto, ambas deberían ser privadas con métodos de agregar, eliminar, para acceder a ellas y esconderlas. Por cuestiones de tiempos no hemos llegado a modificarlas.

- Al agregar un monto a un presupuesto (a pesar de que la modificación queda hecha) no se actualiza en la ventana hasta que se modifique otro monto.

2. Descripción y justificación de diseño

a. Diagrama de paquetes

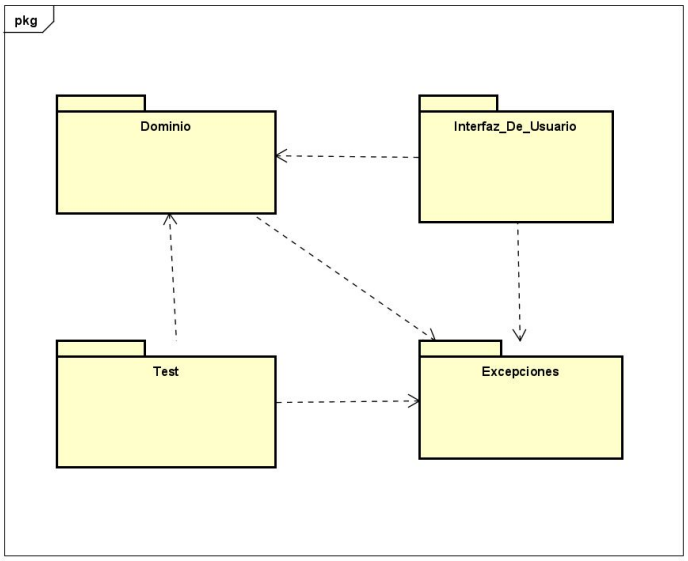
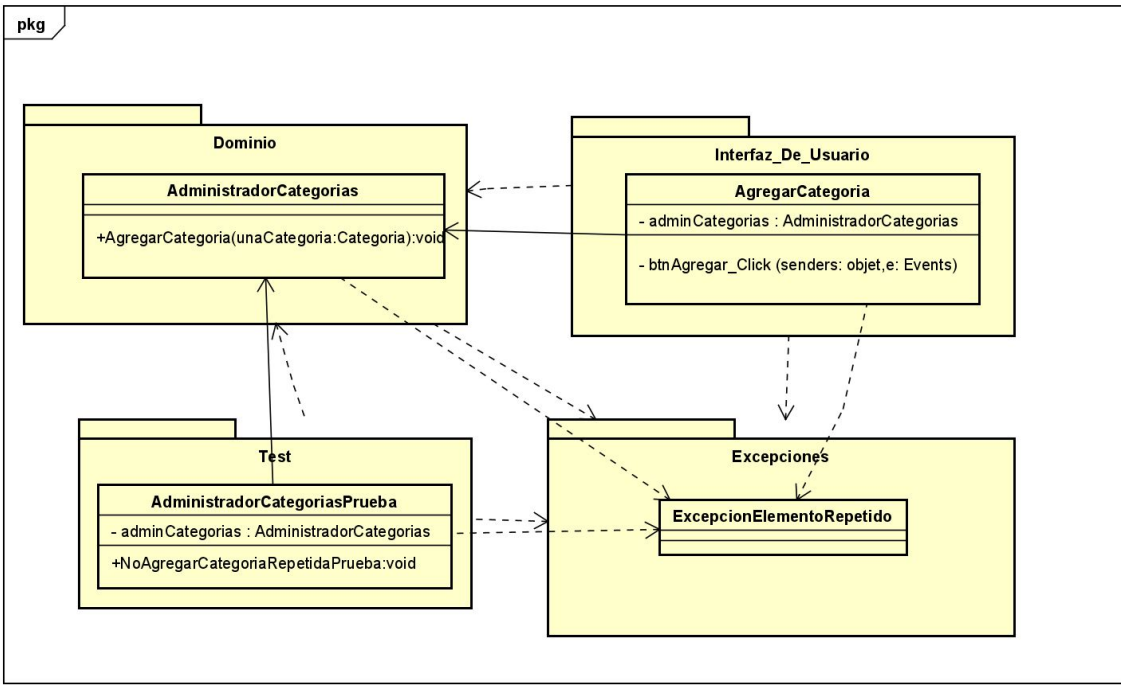
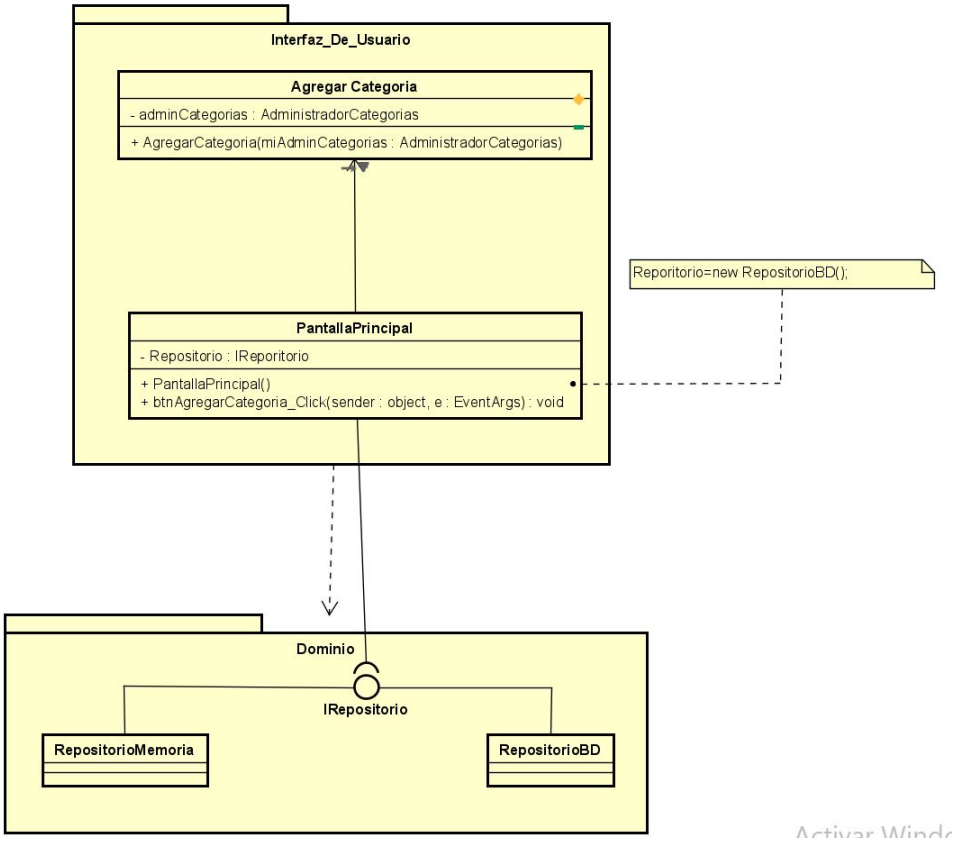


Diagrama de paquetes incluyendo relaciones entre clases.

A modo de ejemplo se eligió la clase AdministradorCategorias para mostrar cómo interactúan los paquetes. El método agregar categoría hace New ExcepcionElementoRepetido, por lo tanto es dependencia. La clase AdministradorCategoriasPrueba tiene una variable del tipo AdministradorCategorias por lo tanto es una relación de asociación. A su vez en el metodo NoAgregarCategoriaRepetidaPrueba recibe una excepción por parámetro (así sucesivamente con las otras clases). Este tipo de relaciones entre clases generan la relación de dependencia entre paquetes.



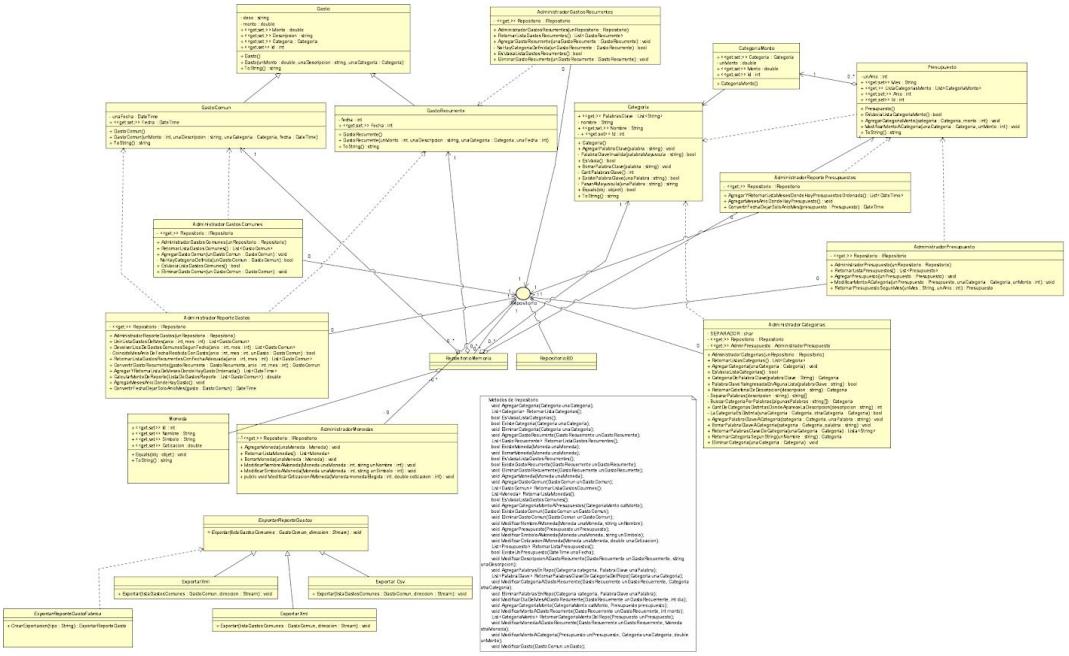
Elegimos tener estos paquetes ya que resultan los agrupamientos más distinguibles según sus responsabilidades. El dominio es quien se encarga de toda la lógica, la interfaz sólo tiene los eventos de las ventanas (sin lógica), los test solo contienen los unit test y las excepciones para manejo de errores. Identificamos que se podría haber creado un paquete para la persistencia ya que tiene una responsabilidad distinta. Este paquete no podría haber estado en un proyecto aparte, ya que quedaría con una relación de agregación con el dominio y esto genera dependencia circular.



Principio DIP (Dependencias invertidas): En este caso, la ventana que es un módulo de alto nivel, está dependiendo del dominio que es de bajo nivel. Para invertir la dependencia podríamos haber puesto la interfaz de repositorio en el paquete de la interfaz de usuario. Surgió la dificultad-discusión de dónde crear las instancias de las implementaciones.

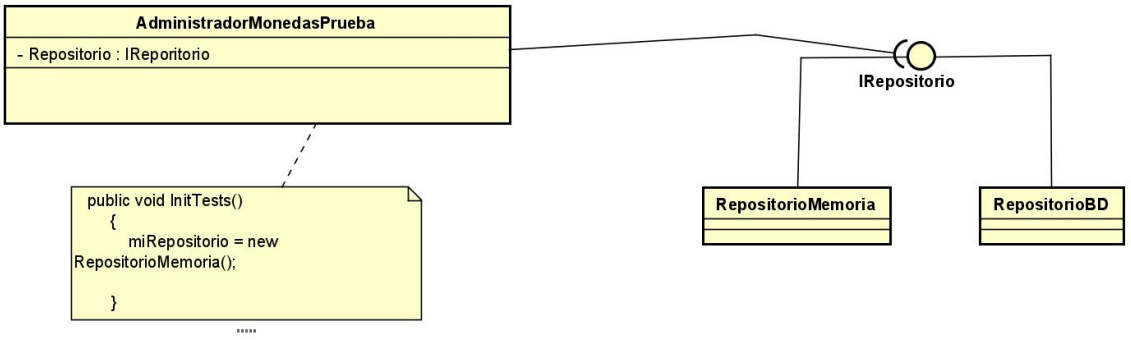
Diagramas de clases:

b.Dominio

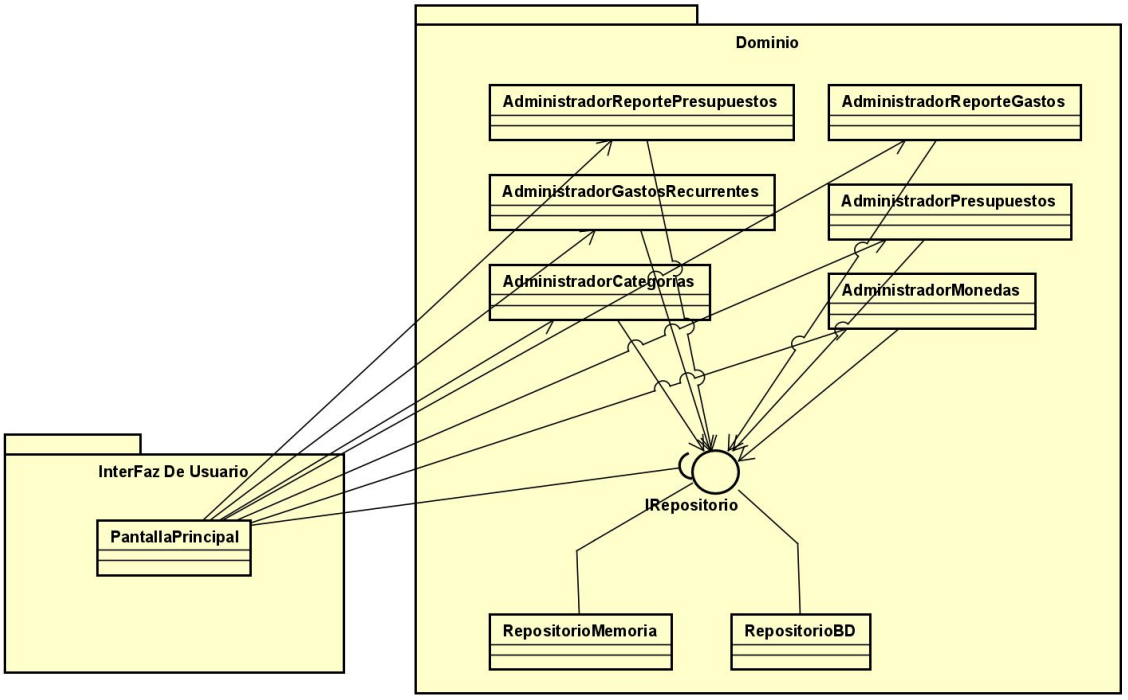


Se adjunta archivo Astah

AdministradorMonedasPrueba - IRepositorio



Para cumplir con todas las pruebas del dominio realizamos Unit Tests.
Como vimos en clase, sobre el aislamiento de pruebas, tomamos RepositorioMemoria como una Fake de la base de datos, y probamos en el dominio contra la misma. Hicimos con una abstracción que representa a ambos repositorios, implementada con una interfaz.



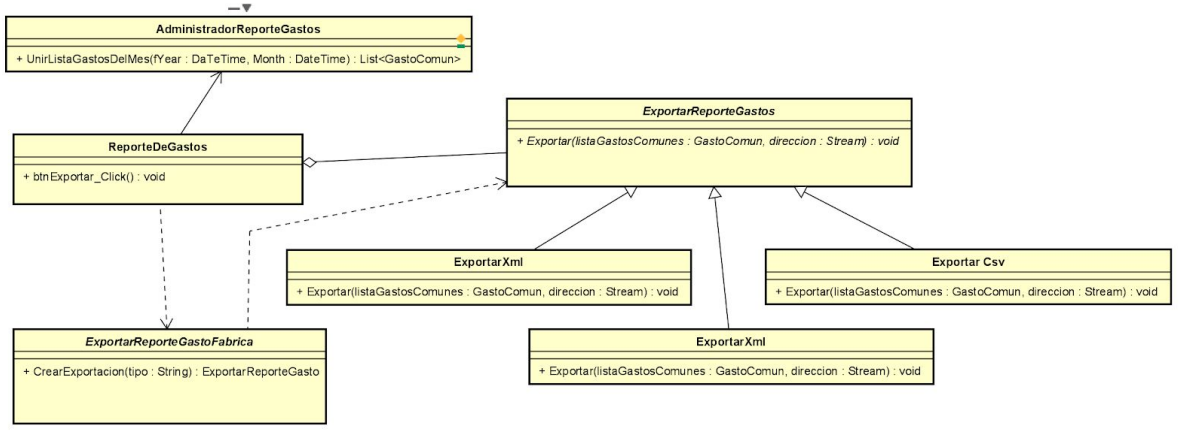
Utilizamos el Patrón Controlador:
Es un tipo de patrón Graps de estructura donde creemos administradores que son los puntos de entrada al sistema y quienes atienden los eventos de la ventana. Si bien algunos administradores contienen más de un caso de uso como por ejemplo AdministradorPresupuesto que permite agregar un presupuesto y modificar un presupuesto, La clase Administrador Reporte de Gastos se centra en el caso de uso de realizar un reporte según un mes y un año. Los Administradores son quienes realizan las validaciones y delegan la responsabilidad de agregar, eliminar, modificar etc. al repositorio.
La pantalla principal es la que crea la instancia de Repositorio en BD y se la pasa por parámetro a los administradores.
Hoy hemos aprendido que se puede utilizar el patrón Singleton, para generar esta única instancia de repositorio en BD.

Patrón Indirección:
Los administradores son clases intermediarias entre el cliente, en este caso la ventana, y el repositorio.

Patrón Fabricación pura:
Los administradores son clases nuevas que no surgen netamente de los requerimientos.

Interface Segregation Principle:
Nuestros administradores dependen de una única interface que tienen varios métodos. Esto lo podríamos haber mejorado siguiendo este principio el cual afirma que los clientes, nuestros administradores, no deberían ser forzados a depender de interfaces.

Diagrama Exportar Reporte de Gastos



Patrón Creador:
Como se puede notar en el diagrama anterior ReporteDeGastos, ventana de la UI, agrega objetos de ExportarReporteGastos, está en una relación de agregacion. Por lo tanto, en respuesta a quien debería ser el encargado de crear los objetos consideramos que debería ser ReporteDeGastos ya que,como establece el patrón creador, debe ser un objeto que esté destinado a estar conectado a objetos nuevos en cualquier momento, así apoyamos el bajo acoplamiento.En los requerimientos del sistema se pide poder crear distintos tipos de objetos a elección del usuario, estos pueden ser archivos: .csv, .txt y ,xml. Por lo tanto, para no complejizar la creación del objeto, y que la clase ReporteDeGastos no tenga demasiadas responsabilidades, llevamos a cabo una **Abstract Factory** "Exportar Reporte Gasto Fábrica" la cual facilita mediante polimorfismo la implementación de distintos tipos.

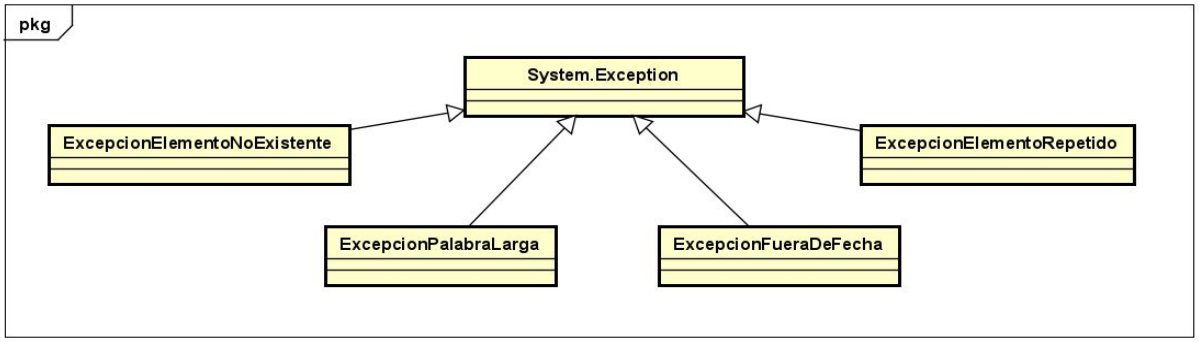
OCP:
Este principio facilitará adaptarse a nuevos requerimientos, ya que al ser extensible se podrá añadir un nuevas instancias a la fábrica abstracta y nuevos tipos en la herencia sin afectar otras funcionalidades ya existentes.Por ejemplo, para poder añadir un nuevo tipo de archivo a exportar en nuestro sistema.

Patrón Strategy:
Se define una familia de algoritmos en torno a exportar gasto, cada uno tiene su implementación y son intercambiables ya que se utiliza polimorfismo. Permite elegir la implementación en tiempos de ejecución.

Patrón Polimorfismo:
Para manejar alternativas basadas en tipos. En este caso, la clase ExportarReporteGasto es abstracta. Con un método Exportar que es abstracto también, lo que permite darle distintas implementaciones en cada clase concreta. Hay 3 clases derivadas concretas según lo requerido ExportarTXT, ExportarCSV, ExportarXml.

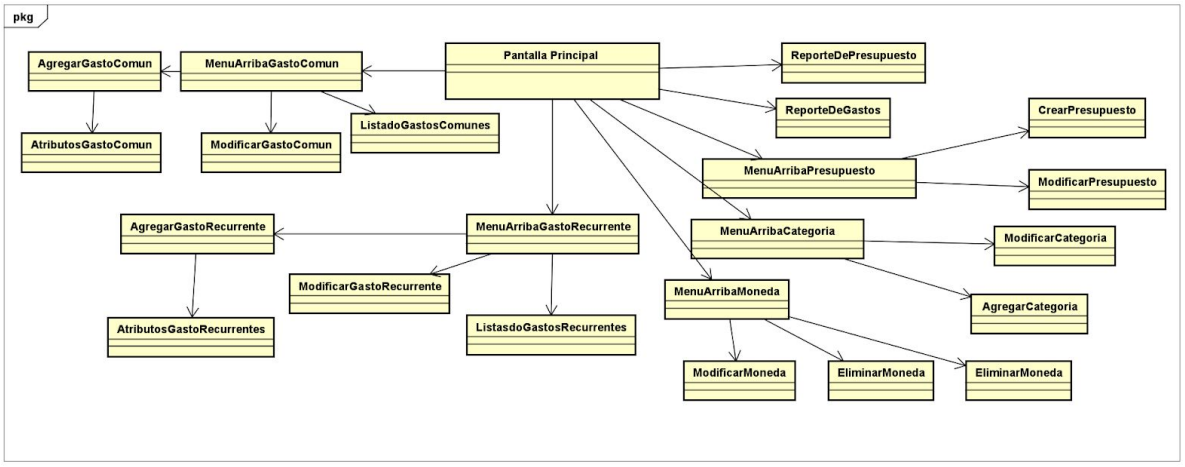
Según la selección del usuario, si desea exportar en txt, csv, xml hay que resolver la implementación en tiempo de ejecución. No es necesario preguntar por el tipo con un if/ else o sentencias switch. Se crea una variable del tipo ExportarReporteGasto y según en new que se haga (explicado anteriormente con el patrón creador) el método de exportar que se va a utilizar.

c) Excepciones



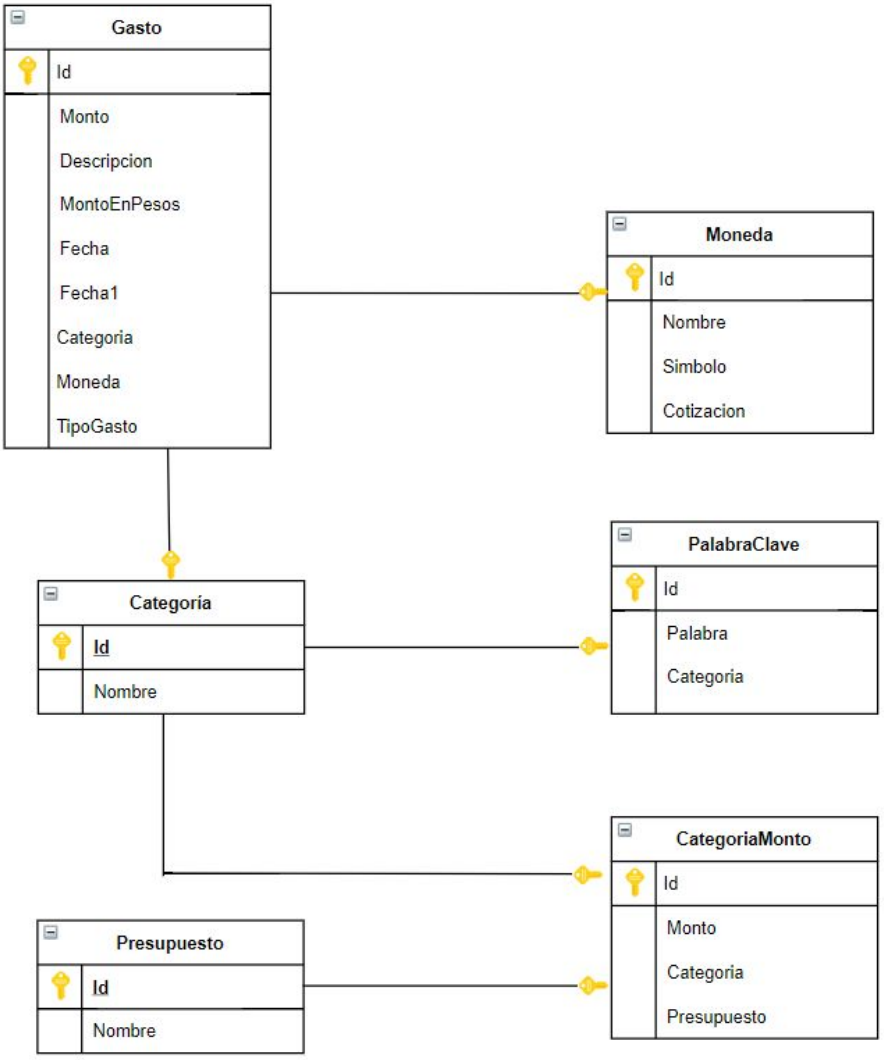
Manejo de errores: Creamos nuestras propias excepciones para las situaciones más comunes como: Elemento repetido, Elemento no existente entre otras, separándolas en clases distintas. Las mismas se usan para validar información, en las clases de dominio. Por ejemplo al agregar una categoría, si la misma ya existe, se hace new ExcepcionElementoRepetido ("Categoría ya existente") y no permite ingresar la categoría. Esta excepción es capturada en la ventana para que se le muestre el mensaje de error al usuario.

d) Interfaz de Usuario



e) Modelo de tablas de la estructura de la base de datos

La base de datos del sistema fue creada, utilizando Entity Framework, en base al dominio. En la figura a continuación se representa el modelo de la base de datos creada:



Para mapear la herencia de Gastos Comunes y Recurrentes de Gasto se utilizó el Mapeo a una tabla única. Esta nos pareció la opción más simple y de fácil acceso a los datos. Esta herencia comprende una jerarquía de clases simple. Hay solo dos subclases, que comparten la mayoría de sus atributos. Su única diferencia es el atributo Fecha, por lo que el espacio desperdiciado en la base de datos no es significante.
A su vez, no es una jerarquía grande y consideramos que Gasto no se espera que crezca en más clasificaciones además de Común y Recurrente, lo que sería una desventaja al usar una única tabla.

Estrategia de carga

Para la carga de los datos, comenzamos utilizando la estrategia Lazy Loading para la lista de Palabras clave en Categoría y la lista de CategoríasMonto en Presupuesto. Luego notamos que esto implicaría muchas consultas y quedaba acoplado el dominio a Entity Framework, por lo que decidimos cambiar la estrategia de carga a Eager Loading.

f) Diagramas de interacción[1]

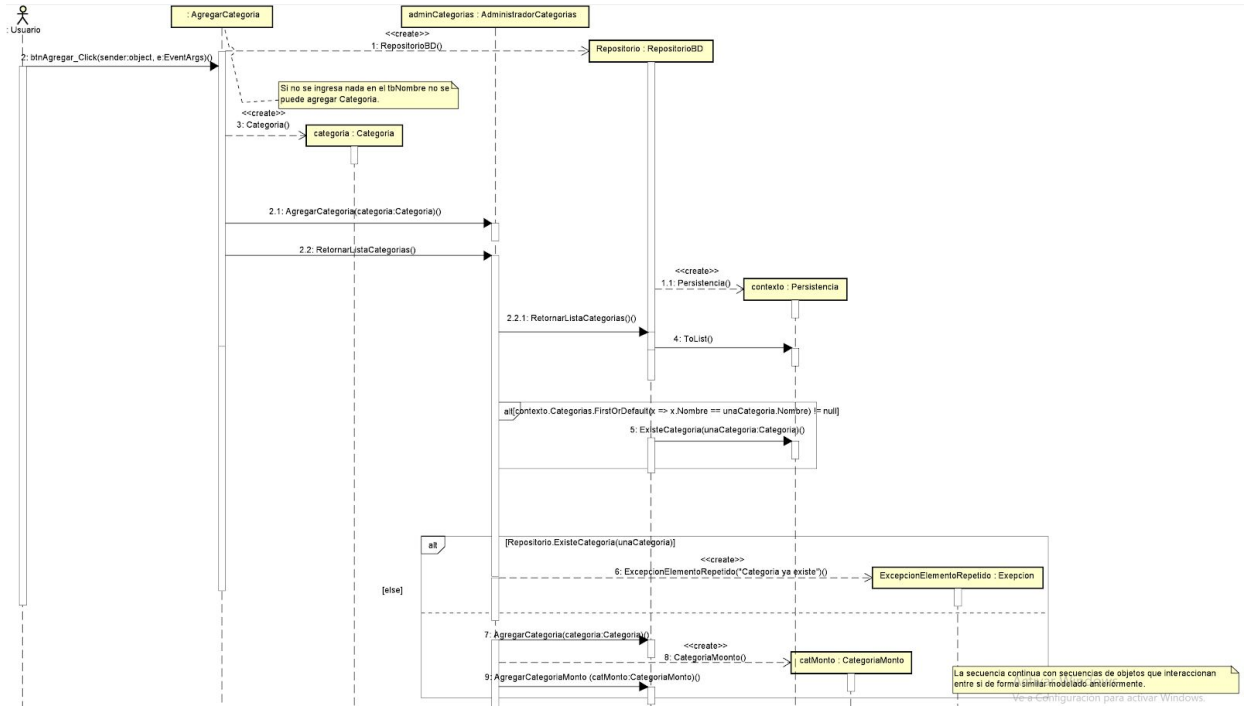
Los diagramas de secuencia que representamos a continuación son :

- Agregar una categoría
- Reporte de Gastos

Ambos ejemplos reflejan cómo avanza a lo largo del tiempo el comportamiento del sistema, desde el momento en que el usuario realiza una acción sobre él, pasando por las distintas capas del dominio.

Diagrama de Secuencia: Agregar una categoría

En este diagrama se pueden ver claramente cuales son los participantes del sistema. Cual es el proceso que está por detrás de agregar una categoría con una visión global, incluso se puede visualizar que sucede ante una excepción.



[1] Se adjunta archivo Asta.

Diagrama de Secuencia: Reporte de gastos

Al igual que en el diagrama anterior podemos ver varios de los componentes que participan del sistema desde el momento en que el usuario presiona consultar.Este diagrama tiene un comportamiento distinto que el anterior ya que al consultar el sistema carga una gráfica, una tabla, tiene métodos diferentes con implementaciones diferentes. Por ejemplo, se ven representados foreach, ifs, entre otros.

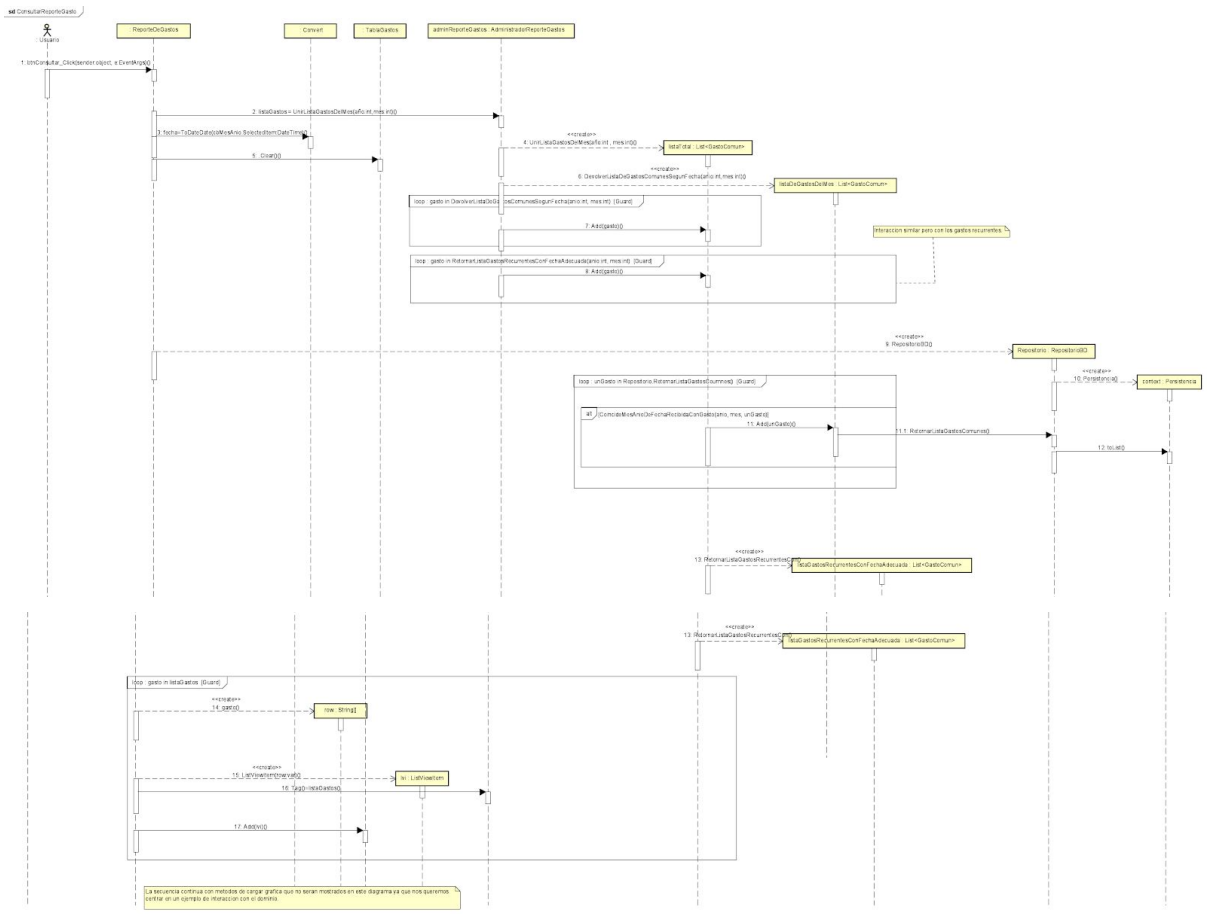
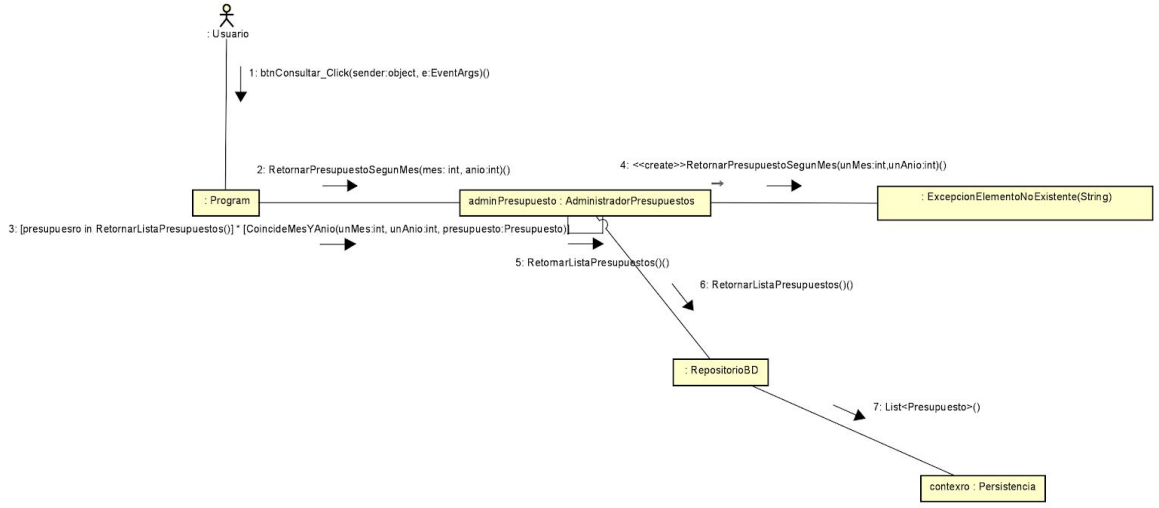


Diagrama de Comunicación: Modificar un presupuesto

Este diagrama nos permite ver con mayor claridad cuáles son los participantes en el sistema a partir de que el usuario presiona el botón modificar.



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

Dominio	806	42.87%	1074	57.13%
AdministradorCategorias	6	3.92%	147	96.08%
AdministradorGastosComunes	0	0.00%	33	100.00%
AdministradorGastosRecurre...	7	12.96%	47	87.04%
AdministradorMonedas	0	0.00%	27	100.00%
AdministradorPresupuesto	9	14.06%	55	85.94%
AdministradorReporteGastos	20	11.49%	154	88.51%
AdministradorReportePresup...	0	0.00%	28	100.00%
Categoria	5	10.00%	45	90.00%
CategoriaMonto	0	0.00%	13	100.00%
ExportarCsv	7	16.67%	35	83.33%
ExportarReporteGastoFabrica	2	15.38%	11	84.62%
ExportarTxt	0	0.00%	25	100.00%
ExportarXML	6	100.00%	0	0.00%
Gasto	0	0.00%	45	100.00%
GastoComun	0	0.00%	26	100.00%
GastoRecuerrente	0	0.00%	20	100.00%
Moneda	2	4.17%	46	95.83%
PalabraClave	1	5.26%	18	94.74%
Persistencia	22	100.00%	0	0.00%
Persistencia.<>c	6	100.00%	0	0.00%
Presupuesto	1	2.08%	47	97.92%
RepositorioBD	700	100.00%	0	0.00%
RepositorioBD.<>c__Display...	4	100.00%	0	0.00%
RepositorioMemoria	8	3.08%	252	96.92%

94.8 %

Para realizar las pruebas unitarias continuamos utilizando Microsoft.VisualStudio.TestTools.UnitTesting.

Cada clase del dominio fue generada a partir de un Test.Nos ayudó a poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.
Se usaron dos métodos de testeo:
[TestClass] para indicar que es una clase de tipo Test.
[TestInitialize] Indica que lo anotado se debe ejecutar antes que todos los métodos [TestMethod]. A su vez [ExpectedException] para probar las excepciones.

Hay clases que bajaron el porcentaje de cobertura de código debido a nuestra decisión de utilizar una abstracción que representa a dos repositorios, lo que entendemos como un Fake de la base de datos. Esto nos bajó el promedio total de cobertura de código, pero sin considerarlo, llegamos a un promedio de 94.8% de cobertura.

4. Cambios respecto a la primer versión

En esta segunda instancia, continuamos basándonos en clean code, codificando en modalidad TDD. Se diseña procurando favorecer patrones y principios de diseño aprendidos a lo largo del curso.
En cuanto a requerimientos, uno de los cambios más significativos fue la implementación de la persistencia de datos. Como consecuencia se incorporó un Id a todas las clases que persistimos. También hubo que modificar la estructura del sistema, añadiendo un Fake de la base de datos: una abstracción que representa dos repositorios, uno que se encarga de manejar los datos a persistir, y otro en memoria.
Para la nueva funcionalidad de alta, baja y la modificación de monedas para asignar a los gastos, se incorporan las clases Moneda y AdministradorMoneda,.
También se agrega la funcionalidad de exportar gastos, y las clases necesarias para realizarlo.

Instalación

Para instalar el programa se requiere tener instalado Microsoft SQL server Management Studio y SQL Server Express 2019.

- importar la base de datos (archivos BAK ubicados en la carpeta llamada “Datos”) al Management Studio
- Abrir el archivo ejecutable de la carpeta Release.

En caso de que no se conecte, copiar y pegar el connection String siguiente en los AppConfig de cada paquete:
"Server=.\SQLEXPRESS;Database=MisFinanzasDB;Integrated Security=True"
providerName="System.Data.SqlClient"