Icono

Descripción generada automáticamente

**UNIVERSIDAD POLITÉCNICA INTERNACIONAL**

**TECNICAS DE PROGRAMACIÓN**

**PROYECTO 1**

SplitBuddies

**ESTUDIANTES**

MANUEL NAVIDAD VALVERDE

YODAN ESPINOZA CARBALLO

**PROFESOR ACADÉMICO**

LUIS FELIPE MORA UMAÑA

**Índice**

**1.** Introducción

**2.** Decisiones de Diseño y Arquitectura

**2.1.** Patrón Modelo-Vista-Controlador (MVC)

**2.2**. Principios SOLID

**2.3.** Patrones de Diseño Aplicados (Singleton)

**2.4**. Código Limpio y Mejores Prácticas

**3.** Desarrollo: Guía de Usuario y Ejecución

**3.1.** Requisitos del Sistema

**3.2**. Configuración y Ejecución

**3.3**. Interacciones del Sistema

**3.3.1**. Pantalla Principal (frmMain)

**3.3.2.** Gestión de Usuarios (frmUsers**)**

**3.3.3**. Gestión de Grupos (frmGroups)

**3.3.4**. Gestión de Gastos (frmExpenses)

**3.3.5.** Generación de Informes (frmReports)

**4.** Análisis de Resultados y Aprendizajes

**5**. Conclusiones

**Introducción**

El proyecto surge como una solución innovadora para la gestión de gastos compartidos, totalmente un desafío para nuestras capacidades en programcion. A medida que las relaciones sociales se vuelven más complejas, la necesidad de una herramienta que facilite el seguimiento de los gastos compartidos se vuelve cada vez más evidente. Con este objetivo en mente, nosotros, como equipo de desarrollo, hemos creado una plataforma que permite a los usuarios registrarse, formar grupos, ingresar gastos y visualizar saldos en tiempo real, eliminando así la confusión que a menudo acompaña a la gestión de finanzas compartidas.

En este documento, describiremos las decisiones técnicas que tomamos durante el desarrollo del proyecto, así como la arquitectura del sistema y su funcionamiento práctico. A lo largo del proceso, hemos aplicado conceptos fundamentales de programación orientada a objetos, patrones de diseño como el Modelo-Vista-Controlador (MVC), y principios de desarrollo de software como SOLID. Nuestro enfoque ha sido garantizar que el código no solo funcione correctamente, sino que también sea fácil de mantener y entender en el futuro. Con la esperanza de que esta herramienta no solo resuelva problemas inmediatos, sino que también se convierta en una solución sostenible y escalable, hemos trabajado arduamente para crear un producto que cumpla con las expectativas de nuestros usuarios y que, al mismo tiempo, nos brinde una valiosa experiencia de aprendizaje en el ámbito del desarrollo de software.

**Decisiones de Diseño y Arquitectura**

La arquitectura del proyecto ha sido bien pensada para no tener dolores de cabeza en el futuro por tener un código muy desordenado. Se tomaron decisiones que permiten que el sistema crezca sin fracturarse, que sea predecible y que cada parte tenga verdaderamente un propósito que sea fácil de ver.

**2.1. Patrón Modelo-Vista-Controlador (MVC)**

El patrón de diseño MVC fue implementado para dividir el sistema en 3 piezas claramente separadas: Modelo, Vista y Controlador. Al dividirlas, cada parte puede hacer lo suyo sin interferir con lo que hace la otra. El Modelo es la lógica de tu sistema, lo que realmente se necesita para que todo funcione. Estas son clases que almacenan datos, como User.cs, **Group.cs y Expense.cs**. Son clases simples, no mucha lógica, solo datos. Hay algunos otros servicios - **DataService** es uno de ellos. **Cs** que maneja el guardado y carga de datos hacia y desde archivos JSON. No se pierde nada haciéndolo de esa manera.

Y hay BalanceService.cs, que calcula la deuda de todos, lo cual puede complicarse si hay más de unas pocas facturas, pero se hace en un solo lugar. La Vista es lo que el usuario ve. Son los formularios **.H frmMain, frmUsers, frmGroups** y así sucesivamente. Todos ellos presentan información y son interactivos. Su papel es materializar datos, recibir entrada y nada más. No procesa, no guarda y no recuerda, solo muestra. El Controlador es el puente. Está en el código que está detrás de cada formulario, básicamente escuchando lo que hace el usuario (como hacer clic en un botón) y luego diciéndole al Modelo que haga su trabajo. En ese caso, obtiene los datos y los empuja a la Vista. De esa manera, todo fluye y la interfaz no se mezcla con la lógica. Tener esa separación es útil porque si deseas almacenar tus datos de una manera diferente en el futuro (como, si deseas comenzar a guardarlos en una base de datos en lugar de en un archivo JSON), no tienes que tocar tus pantallas. Solo se modifica el Modelo.

**2.2. Principios SOLID**

El código siguió los principios SOLID para no convertirse en un desastre. Desde una sola pieza de destello S – Principio de Responsabilidad Única El S aquí significa: Cada clase debe tener una y solo una razón para cambiar. Si DataService trabaja solo en datos, BalanceService tiene solo una lógica, OneForm que quiere decir solo su interfaz. Así que, si algo sale mal, sabes exactamente dónde buscar. El Principio de Abierto/Cerrado (O) -- El sistema debe estar abierto para extensión, pero cerrado para modificación. Por ejemplo, si deseas introducir un nuevo tipo de informes, puedes escribir un nuevo servicio y agregarlo sin tocar la base de código actual.

El código es solo compatible con el Principio de Sustitución de Liskov (L), aunque L no se aplica directamente. Si la herencia se vuelve necesaria más adelante, ya está configurada. I - El principio de la segregación de interfaces no fue considerado con interfaces explícitas, sin embargo, el modularidad del sistema permite implementarlo. También satisface la parte D de DIP porque los formularios no se preocupan de cómo se almacenan los datos, solo que se almacenan, en un DataService. El sistema a su vez no está anclado a ningún almacenamiento, ya que esa instancia está disponible en todas partes.

**2.3. Patrones de Diseño Aplicados (Singleton)**

El DataService empleó el patrón Singleton. **cs class**. La razón es que solo hay una instancia de DataService para toda la aplicación. Esto es, por supuesto, crucial, porque si existen múltiples, posiblemente tendrían datos diferentes, y esto sería muy indeseable. La implementación es sencilla: el constructor es privado, hay una propiedad estática Instance donde creas la instancia cuando se hace referencia por primera vez. Todas las demás veces, simplemente devuelve la misma. Significa que todos los formularios comparten el mismo DataService, los mismos datos, y todo está sincronizado.

**2.4. Código Limpio y Mejores Prácticas**

Se ha hecho un esfuerzo por escribir un código claro, con nombres que indiquen exactamente lo que hacen. Por ejemplo, **btnManageUsers** es un botón para gestionar los usuarios, **RefreshBalances** actualiza los saldos y dgvBalances es una tabla que muestra los saldos. Nada de esto se trata de nombres confusos, todo es claro. Los métodos son pequeños y hacen una pequeña cosa. Un método no hace diez cosas. Por ejemplo, **RefreshBalances** solo actualiza la tabla, y **btnAddUser-Clic** agrega un usuario, cuando lo guarda en DataService. Así que cada parte se vuelve más fácil de probar y arreglar.

También se emplearon comentarios de estilo XML, particularmente en las clases y métodos clave. No están por todas partes, pero en un lugar donde necesitas explicar qué hace la cosa o por qué se hizo de esta manera. En términos de manejo de errores, se usaron bloques try-catch para operaciones sensibles como leer y escribir archivos. En caso de fallo, la aplicación no se cierra, sino que muestra un mensaje y está disponible para ejecutarse. Es cómo mantener el sistema de romperse de maneras no previstas.

**Desarrollo**

**Guía de Usuario y Ejecución**

Windows OS

.Net Framework 4.7.2

Los archivos de datos (users.json, groups.json, expenses.json) deben estar en un directorio de Datos junto al ejecutable.

**3.2. Configuración y Ejecución**

Hacer un fork de este repositorio o simplemente clona o descarga el código.

Abrir el archivo de solución (SplitBuddies.App.sln) con Visual Studio.

Asegurarse de que el paquete NuGet Newtonsoft.Json esté instalado.

Presionar el botón de inicio (o F5) y construye y ejecuta la aplicación.

**3.3. Interacciones del Sistema**

La interfaz principal y opera como el propio panel de control. Muestra una tabla con todos los usuarios y su cantidad absoluta, positiva si tienen que recibir dinero, negativa si debe.

**Botones:** Estos permiten al usuario navegar a otras partes del sistema:

**3.3.1. Usuarios:** frmUsersForm Abre la ventana frmUsers.

**3.3.2. Gestionar Grupos:** Lanza la ventana frmGroups.

**3.3.3. FinancesManagers:** Gestionar Gastos: Abre el formulario frmExpenses.

**3.3.4. Ver Informes:** Abre la ventana frmReports.

**3.3.5. Actualizar:** Recalcula y muestra los saldos.

**Panel de control principal**

**Características:** Registrar nuevo usuario y ver la lista de usuarios creados.

**Interacción:**

Ingresa los campos "Nombre" y "Correo Electrónico".

Haz clic en "Agregar Usuario".

Los datos se validan y el nuevo usuario se almacena en users.json, y se agrega dinámicamente a la tabla de "Usuarios Registrados".

**Característica:** Se puede crear un grupo con un Nombre y una Imagen (Opcional) y se puede agregar una lista de usuarios de todos los usuarios en el sistema.

**Interacción:**

Ingresa un nombre para el grupo.

Incluye las casillas para cada usuario que estará en el grupo en el CheckedListBox.

Haz clic en "Crear Grupo".

El sistema almacena el nuevo grupo en groups.json y analiza la tabla de "Grupos Existentes".

**Interacción:**

Elige el Grupo del ComboBox de Grupo. Esto ordenará automáticamente las listas "Pagado por" y "Participantes".

Ingresa la información del gasto: título/nombre del gasto, cantidad, fecha, etc.

Indica para quién pagaste.

Audita a todos los que participaron en ese gasto.

Haz clic en "Agregar Gasto".

El sistema reduce el costo en el archivo funded.json, y tendrás un saldo diferente en la pantalla principal.

**Características:** Proporciona una vista de informe básica con la posibilidad de usar un rango de fechas.

**Interacción:**

Elige tanto "Fecha de Inicio" como "Fecha de Fin" de estos controles DateTimePicker.

Haz clic en "Generar Informe".

La tabla con resultados mostrará todos los gastos ingresados entre estas dos fechas.

**4. Análisis de Resultados y Aprendizajes**

El desarrollo de este proyecto nos ha permitido obtener una comprensión más profunda de la gestión de proyectos de software, así como de la implementación de patrones de diseño y principios de programación. A lo largo del proceso, se ha podido observar cómo la aplicación del patrón Modelo-Vista-Controlador (MVC) ha facilitado la separación de preocupaciones, lo que ha resultado en un código más limpio y mantenible.

Uno de los aprendizajes más significativos ha sido la importancia de seguir los principios SOLID. Al aplicar el Principio de Responsabilidad Única, se ha logrado que cada clase tenga un propósito claro, lo que ha simplificado la identificación y resolución de errores. Además, el uso del patrón Singleton para el DataService ha garantizado que todos los componentes de la aplicación trabajen con una única fuente de datos, evitando inconsistencias y facilitando la sincronización de la información.

En el proceso nos dimos cuenta de la importancia de la documentación y los comentarios en el código ya que nos facilita la ruta que llevamos y hacia donde deseamos dirigirnos. Además, se nos hizo evidente que un código bien documentado no solo nos ayuda a entender la lógica detrás de las implementaciones, sino que también permite a tomar decisiones en el futuro. La implementación de bloques try-catch para el manejo de errores ha sido fundamental para mejorar la robustez de la aplicación, asegurando que los fallos no interrumpan la experiencia de quien la vaya a utilizar.

En resumen, el desarrollo de este proyecto ha sido todo un reto para nosotros como estudiantes ya que ha sido algo sumamente nuevo, ha sido el trabajo más complicado hasta ahora en la UPI, pero esto nos ha permitido aprender aplicar conceptos teóricos en un entorno práctico, mejorando tanto las habilidades técnicas como la capacidad de trabajar en equipo y gestionar un proyecto de software de manera efectiva.

**5. Conclusiones**

El proyecto ha demostrado ser una solución efectiva para la gestión de gastos compartidos, abordando un problema común en la vida cotidiana de muchas personas. A través de la implementación de todo lo aprendido en clase, se han logrado aplicar las bases que necesitamos para un fututo trabajo en programación . Y técnicamente hablando hemos aplicado lo necesario para cubrir con la capacidad de registrar gastos, crear grupos y generar informes ha proporcionado a los usuarios una herramienta valiosa para mantener un control claro sobre sus finanzas compartidas.

Desde un punto de vista técnico, la aplicación de patrones de diseño y principios de programación ha sido fundamental para el éxito del proyecto. La estructura modular y la separación de responsabilidades han permitido que el código sea más fácil de mantener y escalar. Además, la experiencia adquirida en la gestión de errores y la documentación ha sido invaluable, preparando a los desarrolladores para futuros proyectos

En conclusión, el proyecto no solo ha cumplido con su objetivo de facilitar la gestión de gastos compartidos, sino que también ha servido como un excelente ejercicio de aprendizaje en el ámbito del desarrollo de software. La combinación de teoría y práctica nos ha permitido adquirir habilidades que serán útiles en su futura carrera profesional, que nota la importancia de proyectos prácticos en la educación universitaria.