Manual técnico del sistema "Polibanco"

CONTENIDO

PRESENTACIÓN

Este manual técnico describe el desarrollo del sistema de cajero automático "Polibanco", un proyecto que simula las operaciones básicas de un cajero automático. Se detallan las herramientas utilizadas, el diseño del sistema y las instrucciones para su uso y modificación.

RESUMEN

El sistema "Polibanco" se desarrolló utilizando el lenguaje de programación C y herramientas como Zinjai y GitHub. Permite a los usuarios realizar operaciones bancarias básicas como retiros, depósitos y transferencias. El código se gestionó a través de GitHub para facilitar el trabajo colaborativo y el control de versiones.

OBJETIVO

El objetivo principal de este proyecto es simular un cajero automático funcional que permita a los usuarios realizar operaciones bancarias básicas de forma segura y eficiente. Además, sirve como ejemplo de desarrollo de software utilizando buenas prácticas de programación y control de versiones.

FINALIDAD DEL MANUAL

Este manual tiene como finalidad proporcionar una guía completa para comprender el funcionamiento del sistema "Polibanco". Está dirigido a desarrolladores, testers y cualquier persona interesada en aprender sobre el desarrollo de aplicaciones de cajeros automáticos.

INTRODUCCIÓN

El sistema "Polibanco" es una simulación de un cajero automático que permite a los usuarios realizar operaciones bancarias básicas. Se ha desarrollado con el objetivo de proporcionar una experiencia realista y segura. Este manual describe las características del sistema, su arquitectura y las herramientas utilizadas en su desarrollo.

1. ASPECTOS TÉCNICOS

1.1. HERRAMIENTAS UTILIZADAS PARA EL DESARROLLO

1.1.1. Zinjai (Pág. 9)

Zinjai es un entorno de desarrollo integrado (IDE) que se utilizó para escribir, compilar y depurar el código fuente del sistema "Polibanco". Ofrece una interfaz intuitiva y herramientas que facilitan el desarrollo en C.

1.1.2. GitHub

GitHub es una plataforma de desarrollo colaborativo basada en la nube que utiliza el sistema de control de versiones Git. Se utilizó para almacenar el código fuente del proyecto, facilitar el trabajo en equipo, el seguimiento de cambios y el control de versiones.

1.1.3. Lenguaje C

El sistema "Polibanco" se desarrolló utilizando el lenguaje de programación C. C es un lenguaje de propósito general, eficiente y flexible, ideal para este tipo de aplicaciones que requieren un control preciso de los recursos del sistema.

1.2 DESCRIPCIÓN CÓDIGO FUENTE

- **`obtenerNumero(const char *mensaje)`**: (int) Muestra un mensaje al usuario y obtiene un número entero como entrada, validando que la entrada sea correcta.
- **`obtenerTexto(const char *mensaje, char *texto, int tamano)`**: (void) Muestra un mensaje y obtiene una cadena de texto como entrada.
- **`obtenerContrasena(const char *mensaje, char *contrasena, int tamano)`**: (void) Obtiene la contraseña del usuario sin mostrarla en la pantalla.
- **`inicializarUsuarios()`**: (void) Inicializa los datos de los usuarios con valores predeterminados.
- **`mostrarMenuPrincipal()`**: (void) Muestra el menú principal del cajero automático.
- **`realizarRetiro(int indice)`**: (void) Permite al usuario realizar un retiro de su cuenta.
- **`realizarDeposito(int indice)`**: (void) Permite al usuario realizar un depósito en su cuenta.
- **`realizarTransferencia(int indice)`**: (void) Permite al usuario transferir dinero a otra cuenta.
- **`estadoDeCuenta(int indice)`**: (void) Muestra el estado de cuenta del usuario, incluyendo los últimos movimientos.
- **`exportarEstadoDeCuenta(int indice)`**: (void) Exporta el estado de cuenta a un archivo de texto.
- **`registrarMovimiento(Usuario *usuario, const char *descripcion)`**: (void) Registra un movimiento en el historial del usuario.
- **`registrarFechaMovimiento(Usuario *usuario)`**: (void) Registra la fecha y hora del movimiento.
- **`validarMultiploDe10(int cantidad)`**: (int) Verifica si una cantidad es múltiplo de 10.
- **`verificarContrasena(int indice)`**: (int) Verifica si la contraseña ingresada por el usuario es correcta.

2. DIAGRAMAS DE MODELAMIENTO

(Aquí debes incluir los diagramas que hayas creado durante el desarrollo del sistema. Si no los tienes, te recomiendo crearlos. Son muy útiles para visualizar la estructura y el funcionamiento del sistema.)

2.1. DIAGRAMA DE CLASES

(Inserta aquí el diagrama de clases. Debe mostrar las clases que componen el sistema y sus relaciones. En este caso, podrías tener una clase "Usuario" con atributos como nombre, número de cuenta, contraseña y saldo.)

2.2. DIAGRAMA DE CASOS DE USO

(Inserta aquí el diagrama de casos de uso. Debe mostrar las interacciones entre los usuarios y el sistema. Por ejemplo, "Retirar efectivo", "Depositar efectivo", "Transferir dinero", "Consultar saldo".)

2.3. DICCIONARIO DE DATOS

(Inserta aquí el diccionario de datos. Debe describir las estructuras de datos utilizadas en el sistema, incluyendo los tipos de datos, las variables y su significado. Por ejemplo, la estructura "Usuario" con sus campos "nombre" (cadena de texto), "numero_cuenta" (entero), "contrasena" (cadena de texto), "saldo" (decimal).)

3. ASPECTO TÉCNICO DEL DESARROLLO DEL SISTEMA

3.1. MODIFICACIÓN LOCAL

Para modificar el código fuente del sistema "Polibanco", se debe utilizar Zinjai o cualquier otro editor de texto que soporte C. Se recomienda descargar el código desde el repositorio de GitHub y realizar las modificaciones localmente.

3.2. CONTROL DE VERSIONES CON GIT

3.2.1. Creación de un repositorio

Se creó un repositorio en GitHub para almacenar el código fuente del sistema "Polibanco". Esto permite llevar un control de las versiones del código y facilita el trabajo colaborativo.

3.2.2. Ramificaciones (Branching)

Se utilizaron ramas (branches) en Git para desarrollar nuevas funcionalidades o corregir errores de forma independiente, sin afectar la rama principal del proyecto. En este proyecto se crearon las siguientes ramas:

- **corrección-registrar-movimiento:** Para corregir un error en la impresión de fechas en el historial de movimientos.
- **mejora-estadocuenta:** Para mejorar la información mostrada en el estado de cuenta.
- **mejora-transferencia:** Para agregar un límite a la cantidad máxima de transferencia.
- **control-contrasena:** Para implementar un control de intentos fallidos de contraseña.
- **mejora-retiro:** Para establecer un límite en la cantidad de retiro.
- **mejora-deposito:** Para limitar la cantidad máxima de depósito.
- **exportar-archivo:** Para mejorar la función de exportar el estado de cuenta a un archivo.

3.2.3. Confirmaciones (Commits)

Cada cambio realizado en el código se registró como una confirmación (commit) en Git con un mensaje descriptivo. Esto permite llevar un historial de las modificaciones y facilita la reversión a versiones anteriores si es necesario.

3.2.4. Fusiones (Merges)

Una vez que los cambios en una rama se probaron y aprobaron, se fusionaron (merge) con la rama principal del proyecto (`main`).

3.2.5 Resolución de Conflictos

Durante el proceso de fusión, se puede presentar un conflicto si dos ramas modifican la misma línea de código. En este proyecto se presentó un conflicto al fusionar la rama `mejora-transferencia` con `main`. El conflicto se resolvió manualmente, combinando los cambios de ambas ramas y realizando pruebas para asegurar el correcto funcionamiento.

4. REQUERIMIENTOS DEL SOFTWARE

4.1. REQUISITOS MÍNIMOS

- Sistema operativo: Windows
- IDE: Zinjai (u otro compilador de C)
- Librerías: stdio.h, stdlib.h, string.h, conio.h, time.h

BIBLIOGRAFÍA

- Documentación de Zinjai: (enlace a la documentación oficial, si existe)

- Documentación de GitHub: (enlace a la documentación oficial)

DESCRIPCIÓN DE LAS FUNCIONES

A continuación, se detallan las funciones utilizadas en el proyecto, incluyendo su tipo, parámetros, retorno y una breve descripción de su funcionalidad:

- **`obtenerNumero(const char *mensaje)`**
  - **Tipo:** `int`
  - **Parámetros:** `mensaje` (cadena de texto que se muestra al usuario)
  - **Retorno:** Un número entero ingresado por el usuario.
  - **Descripción:** Muestra un mensaje al usuario y obtiene un número entero como entrada, validando que la entrada sea correcta.
- **`obtenerTexto(const char *mensaje, char *texto, int tamano)`**
  - **Tipo:** `void`
  - **Parámetros:** `mensaje` (cadena de texto que se muestra al usuario), `texto` (puntero a un array de caracteres donde se almacenará la entrada del usuario), `tamano` (tamaño máximo del array de caracteres).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Muestra un mensaje al usuario y obtiene una cadena de texto como entrada, almacenándola en el array de caracteres proporcionado.
- **`obtenerContrasena(const char *mensaje, char *contrasena, int tamano)`**
  - **Tipo:** `void`
  - **Parámetros:** `mensaje` (cadena de texto que se muestra al usuario), `contrasena` (puntero a un array de caracteres donde se almacenará la contraseña), `tamano` (tamaño máximo del array de caracteres).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Muestra un mensaje al usuario y obtiene la contraseña como entrada sin mostrarla en la pantalla, almacenándola en el array de caracteres proporcionado.
- **`inicializarUsuarios()`**
  - **Tipo:** `void`
  - **Parámetros:** No recibe ningún parámetro.
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Inicializa los datos de los usuarios con valores predeterminados.
- **`mostrarMenuPrincipal()`**
  - **Tipo:** `void`
  - **Parámetros:** No recibe ningún parámetro.
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Muestra el menú principal del cajero automático.
- **`realizarRetiro(int indice)`**
  - **Tipo:** `void`
  - **Parámetros:** `indice` (índice del usuario en el array de usuarios).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Permite al usuario realizar un retiro de su cuenta.

- **`realizarDeposito(int indice)`**
  - **Tipo:** `void`
  - **Parámetros:** `indice` (índice del usuario en el array de usuarios).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Permite al usuario realizar un depósito en su cuenta.
- **`realizarTransferencia(int indice)`**
  - **Tipo:** `void`
  - **Parámetros:** `indice` (índice del usuario en el array de usuarios).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Permite al usuario transferir dinero a otra cuenta.
- **`estadoDeCuenta(int indice)`**
  - **Tipo:** `void`
  - **Parámetros:** `indice` (índice del usuario en el array de usuarios).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Muestra el estado de cuenta del usuario, incluyendo los últimos movimientos.
- **`exportarEstadoDeCuenta(int indice)`**
  - **Tipo:** `void`
  - **Parámetros:** `indice` (índice del usuario en el array de usuarios).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Exporta el estado de cuenta a un archivo de texto.
- **`registrarMovimiento(Usuario *usuario, const char *descripcion)`**
  - **Tipo:** `void`
  - **Parámetros:** `usuario` (puntero a la estructura del usuario), `descripcion` (cadena de texto que describe el movimiento).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Registra un movimiento en el historial del usuario.
- **`registrarFechaMovimiento(Usuario *usuario)`**
  - **Tipo:** `void`
  - **Parámetros:** `usuario` (puntero a la estructura del usuario).
  - **Retorno:** No retorna ningún valor.
  - **Descripción:** Registra la fecha y hora del movimiento.
- **`validarMultiploDe10(int cantidad)`**
  - **Tipo:** `int`
  - **Parámetros:** `cantidad` (número entero que se va a validar).
  - **Retorno:** 1 si la cantidad es múltiplo de 10, 0 en caso contrario.
  - **Descripción:** Verifica si una cantidad es múltiplo de 10.
- **`verificarContrasena(int indice)`**
  - **Tipo:** `int`
  - **Parámetros:** `indice` (índice del usuario en el array de usuarios).
  - **Retorno:** 1 si la contraseña es correcta, 0 en caso contrario.
  - **Descripción:** Verifica si la contraseña ingresada por el usuario es correcta.