“Año del Bicentenario, de la consolidación de nuestra Independencia, y de la conmemoración de las heroicas batallas de Junín y Ayacucho”

Caso: Actividad Calificada 2

Facultad de Ingeniería

Curso:

Programación Orientada a Objetos II

Docente:

Alfaro Gutierrez, Gianny Romie

Grupo - Integrantes:

Reyes Dioses, Victor Daniel(2312567)

Cordova Fernandez, Jose Alberto(2310153)

Huisa Villafuana, Giancarlo Victor(2310919)

Bellatin Nadal, Rafaela(2310185)

Casas Castro, Maria Kamila (2310758)

Lima-Perú

2024-01

1. Análisis del Problema

    Identificación de 3 problemas de diseño en el código actual:

    - El código actual presenta varios problemas de diseño. En primer lugar, utiliza múltiples estructuras if-else o switch para crear los diferentes tipos de cuentas, lo que genera un código rígido, difícil de extender o mantener. Además, viola el principio de **Abierto/Cerrado (OCP)**, ya que cada vez que se agrega un nuevo tipo de cuenta, es necesario modificar el código existente en lugar de simplemente extenderlo. Por último, el sistema sufre de **alto acoplamiento**, ya que la GUI (BancoGUI) está directamente vinculada a la lógica de creación de las cuentas (CuentaAhorro), lo que complica la posibilidad de realizar cambios independientes en cada componente del sistema.

¿Por qué no escala para nuevos tipos de cuentas?

Cada vez que aparece un nuevo tipo de cuenta, toca modificar una y otra vez los bloques de if-else o switch, lo que acaba impactando varias clases al mismo tiempo. Ese vaivén de cambios eleva la probabilidad de introducir errores, dificulta las pruebas de funcionalidades nuevas y, al final, hace que mantener el sistema a largo plazo sea casi inviable.

2. Elección del Patrón de Diseño

¿Factory Method o Builder?

Elegimos `Factory Method`

Justificación (3 argumentos):

1. El patrón Factory Method ofrece una gran simplicidad y claridad al delegar la creación de cada tipo de cuenta a subclases especializadas, manteniendo el código limpio y bien organizado; además, su naturaleza escalable nos permite añadir nuevos tipos de cuenta simplemente creando nuevas fábricas sin tocar el código existente, y encaja de forma natural con interfaces gráficas como Swing, donde un combo box puede instanciar dinámicamente el objeto apropiado según la opción seleccionada.

Refactorizar Usando Factory Method:

Análisis del Problema:

El sistema bancario original utilizaba un enfoque rígido con múltiples condicionales (como `if-else` o `switch`) para determinar el tipo de cuenta que se debía crear. Esto introduce varios problemas:

- El sistema presenta una falta de flexibilidad y escalabilidad, ya que cada vez que se agrega un nuevo tipo de cuenta, se debe modificar el código existente, lo que aumenta la posibilidad de errores y la complejidad. Además, cambiar reglas de negocio, como la tasa de interés de una cuenta, implica tocar múltiples partes del código, lo que compromete la coherencia del sistema. Por último, existe un acoplamiento fuerte entre la lógica de negocio y la interfaz gráfica, ya que la creación de cuentas está directamente vinculada con la vista, lo que complica tanto las pruebas como el mantenimiento del sistema.

Identificar 3 problemas de diseño en el código actual:

1. **Acoplamiento alto entre la vista y la lógica de negocio:**
    - En el código original, la interfaz gráfica asume directamente la responsabilidad de decidir qué tipo de cuenta crear, lo que genera un fuerte acoplamiento entre la vista y la lógica de negocio y rompe el principio de separación de responsabilidades.
2. **Violación del principio abierto/cerrado (OCP):**
    - El código no está preparado para ser extendido de manera sencilla; cada vez que se añade un nuevo tipo de cuenta, como una cuenta para menores, se debe modificar la lógica de creación de cuentas. Esto va en contra del principio abierto/cerrado, que establece que el código debe ser abierto para agregar nuevas funcionalidades, pero cerrado para modificaciones en el código existente.
3. **Dependencia directa de la creación de cuentas:**
    - La vista no se limita a mostrar o recopilar datos, sino que también asume la responsabilidad de instanciar directamente los objetos de cuenta, lo que genera un acoplamiento excesivo entre la interfaz y la lógica de negocio y complica tanto el mantenimiento como la realización de pruebas.

Explicar por qué no escala para nuevos tipos de cuentas:

> Nuestro código actual depende de largos bloques de `if-else` o `switch` para decidir qué tipo de cuenta crear; cada vez que surge una nueva variante, tenemos que añadir un caso más a ese mazo de condicionales. El resultado es un código enredado y frágil, difícil de mantener y lleno de puntos donde pueden colarse errores, además de ir directamente contra el principio abierto/cerrado de SOLID, ya que no podemos ampliar la funcionalidad sin modificar el núcleo. Esto priva al sistema de la escalabilidad necesaria para crecer sin arriesgar su estabilidad.

Elección del Patrón: ¿Factory Method o Builder?

> **Elección: Factory Method.**
>
> Justificación para elegir Factory Method:
>
> 1. **Creación de objetos complejos:**
>
>   - El patrón Factory Method encaja perfectamente cuando tenemos que crear distintos tipos de cuentas, porque nos permite delegar la instanciación en clases concretas sin tocar el núcleo del sistema; basta con que cada nueva cuenta extienda una interfaz común (CuentaAhorro) y la fábrica, a través de su método, decida qué implementación concreta devolver según el tipo solicitado.
>
> 2. **Facilidad de adición de nuevos tipos de cuentas:**
>
>   - Con Factory Method, incorporar una nueva cuenta se vuelve muy sencillo: solo hace falta definir su propia clase concreta y actualizar la fábrica para que la reconozca, sin tener que tocar ni un solo renglón del código ya existente.
>
> 3. **Separación de responsabilidades:**
>
>   - La interfaz se limita a recoger la información del usuario y a enviarla al controlador, que a su vez invoca a la fábrica para crear la cuenta correspondiente; de este modo, cada componente asume su propia responsabilidad y el sistema gana en modularidad.

Implementación con Factory Method:

1. **Refactorizar el código aplicando el patrón elegido (Factory Method):**

   Vamos a crear una clase abstracta **`CuentaAhorro`** y luego una **fábrica** que decide qué tipo de cuenta crear, sin necesidad de modificar el código central.

   - **CuentaAhorro.java**: clase base con atributos comunes para las cuentas.
   - **CuentaAhorroFactory.java**: clase que decide qué tipo de cuenta crear.
   - **Tipos de cuentas**: cada tipo de cuenta será una clase concreta que extiende `CuentaAhorro`.

2. **Estructurar el código fuente con MVC:**

   La estructura MVC se sigue para separar claramente las responsabilidades:

   - **Modelo (`/model`)**: Contiene las clases de las cuentas y la fábrica.
   - **Vista (`/view`)**: Interfaz gráfica (Swing) que permite al usuario seleccionar el tipo de cuenta y proporcionar los datos.
   - **Controlador (`/controller`)**: Controla la lógica de creación de las cuentas utilizando la **fábrica**.

3. **Mantener una GUI simple (Swing/JavaFX):**

   La interfaz gráfica (GUI) es simple:

   - **ComboBox** para seleccionar el tipo de cuenta.
   - **JTextField** para ingresar los datos (como el titular, el saldo, etc.).
   - Un **JButton** que llama al controlador para crear la cuenta.

4. **Agregar un nuevo tipo de cuenta: Cuenta para Menores:**

   Se agrega un tipo de cuenta llamado **Cuenta para Menores**, que tiene:

   - Tasa baja.
   - Sin retiros en ATM.
   - Requiere tutor para firmar.

5. **Guardar las cuentas de ahorrBeo en un archivo `cuentas.txt`:**

   Cuando se crea una cuenta, sus detalles se guardan en un archivo `cuentas.txt` para persistencia.

7. Roles y Responsabilidades (Trabajo Colaborativo)

Rol	Responsabilidades
Analista de POO	Diseñar el diagrama de clases, definir herencia, interfaces y clases abstractas.
Desarrollador MVC	Implementar la estructura Modelo-Vista-Controlador.
Desarrollador Patrón de Diseño	Implementar el patrón de diseño que se acordó con el analista de POO.
Diseñador de GUI	Crear la interfaz gráfica con Swing (formulario, botones, validaciones).
Gestor de Archivos	Manejar la lectura/escritura en invitados.txt y validar datos.