Universidad de La Laguna

Grado en Ingeniería Informática

Diseño Arquitectónicos y patrones

Práctica 7. Patrón Observador

Alejandro García Pérez
alu0101441207
04/12/25

Índice

Introducción

Este proyecto implementa el **Patrón Observador** a través de **SecureBank Core**, un simulador de infraestructura bancaria crítica diseñado para entornos de alta concurrencia. El sistema se articula en torno a un motor central (TransactionEngine) que actúa como Sujeto, emitiendo eventos financieros en tiempo real a una red desacoplada de subsistemas reactivos.

La arquitectura integra diversos actores independientes que operan simultáneamente:

- **Frontend:** Simuladores móviles con gestión de sesiones y notificaciones *Push*.
- **Seguridad:** Un motor de IA (*FraudDetectorAI*) con análisis de comportamiento para bloquear fraudes.
- **Monitorización:** Un Dashboard para la visualización de tráfico y métricas.
- **Integridad:** Un *Ledger* inmutable basado en hashing criptográfico.

Gracias a la aplicación del Patrón Observador, se ha logrado una arquitectura modular y flexible. Esto conlleva, como beneficio adicional, el cumplimiento del Principio Open/Closed, ya que el sistema permite integrar nuevos observadores sin necesidad de modificar el código del motor transaccional.

Tiempo dedicado a la práctica

A continuación, se detalla la planificación temporal estimada frente al tiempo real invertido. La complejidad del proyecto ha sido superior a prácticas anteriores debido a la incorporación de persistencia de datos, programación concurrente (hilos) y lógica de seguridad avanzada.

Tarea	Tiempo estimado	Tiempo real dedicado
Diseño de la Arquitectura (Interfaces y patrón Observer)	1 hora	45 minutos
Implementación del Core (TransactionEngine y Observadores)	2 horas	2 horas
Persistencia y Datos (CSV, Repositorio, Login DNI)	3 horas	4 horas
Interfaz Gráfica Avanzada (Swing, Recibos, Dashboard)	4 horas	5 horas y 30 minutos
Motor de Seguridad (Behavioral Analytics, Hilos)	2 horas	3 horas
Refactorización OCP (Inyección de Dependencias)	1 hora	1 hora

Pruebas y Documentación	2 horas	2 horas
Tiempo total	15 horas	18 horas y 15 minutos

Tabla 1. *Tiempo estimado versus tiempo real dedicado a la práctica*

Cómo podemos ver en la Tabla 1, el tiempo real se ha ajustado bastante a la estimación. El uso de herramientas de IA generativa para la creación de datos de prueba (CSV) y componentes visuales repetitivos (Swing) permitió dedicar la mayor parte del esfuerzo a la lógica de negocio.

Diagrama de clases. Planteamiento y estructura

El diseño de la solución se fundamenta estrictamente en el Patrón de Diseño Observador. La arquitectura se divide en tres capas lógicas desacopladas:

El Sujeto (ITransactionSubject y TransactionEngine):

- **ITransactionSubject:** Es la interfaz que define las operaciones de suscripción (attach, detach) y notificación (notifyObservers)
- **TransactionEngine:** Es la implementación concreta del sujeto. Mantiene la lista de suscriptores y su única responsabilidad es procesar la lógica de negocio y emitir eventos. No tiene dependencias hacia las implementaciones concretas de las pantallas o servicios.

Los Observadores (IBankObserver):

- **Visuales:** MobilePhoneSimulator (Cliente) y BankAdminConsole (Administración).
- **Lógicos:** FraudDetectorAI (Seguridad), GeneralLedger (Integridad), AuditLogger (Registro) y NotificationService (Comunicaciones).
- Todos implementan la misma interfaz, permitiendo que el motor los trate de forma polimórfica.

Capa de Datos y Soporte:

- **BankRepository:** Encapsula el acceso a datos. Se ha refactorizado para implementar la interfaz IBankRepository, permitiendo cambiar el origen de datos (CSV a SQL, por ejemplo) sin afectar al resto del sistema, cumpliendo el Principio de Inversión de Dependencias.

Este planteamiento garantiza que el sistema sea extensible. Añadir un nuevo tipo de notificación o dispositivo no requiere modificar el código del motor, sino simplemente crear una nueva clase y suscribirla.

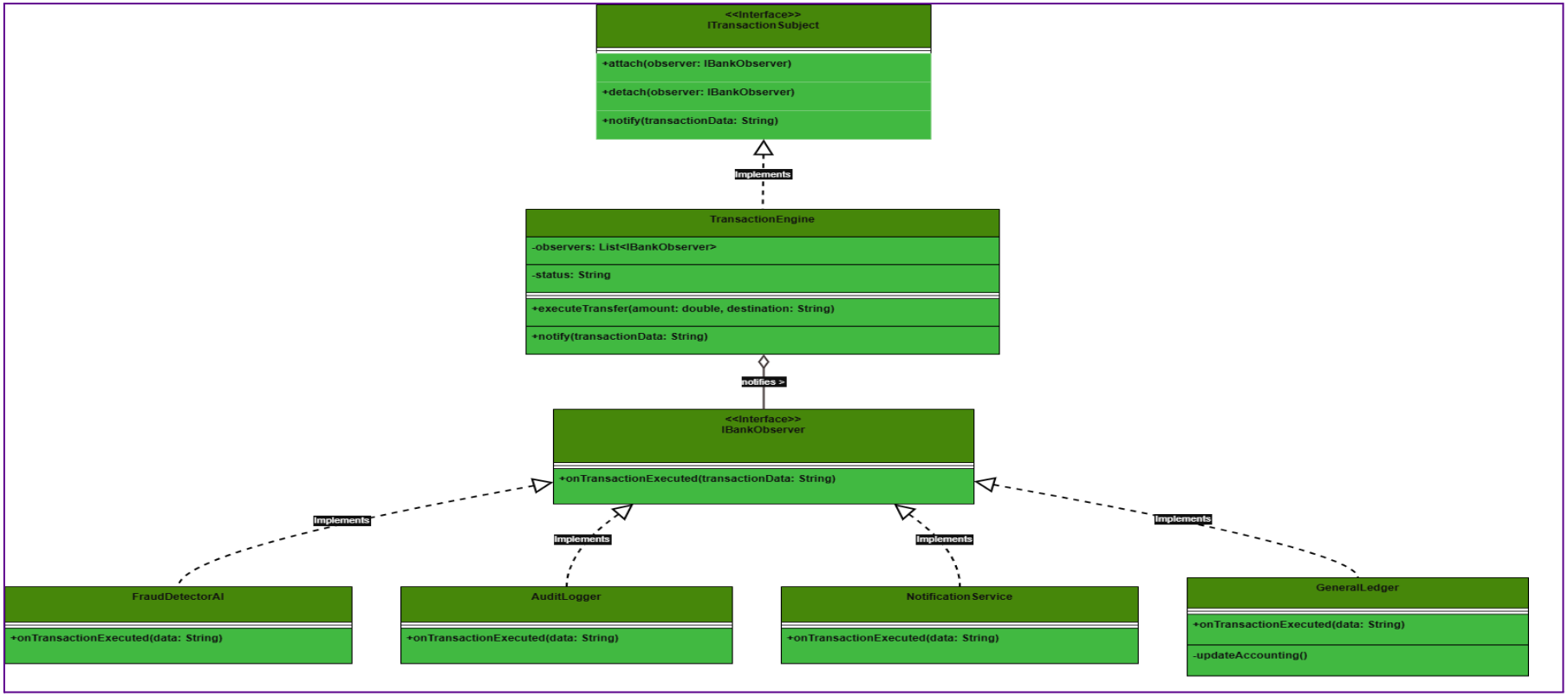

Figura 1. *Diagrama inicial del proyecto*

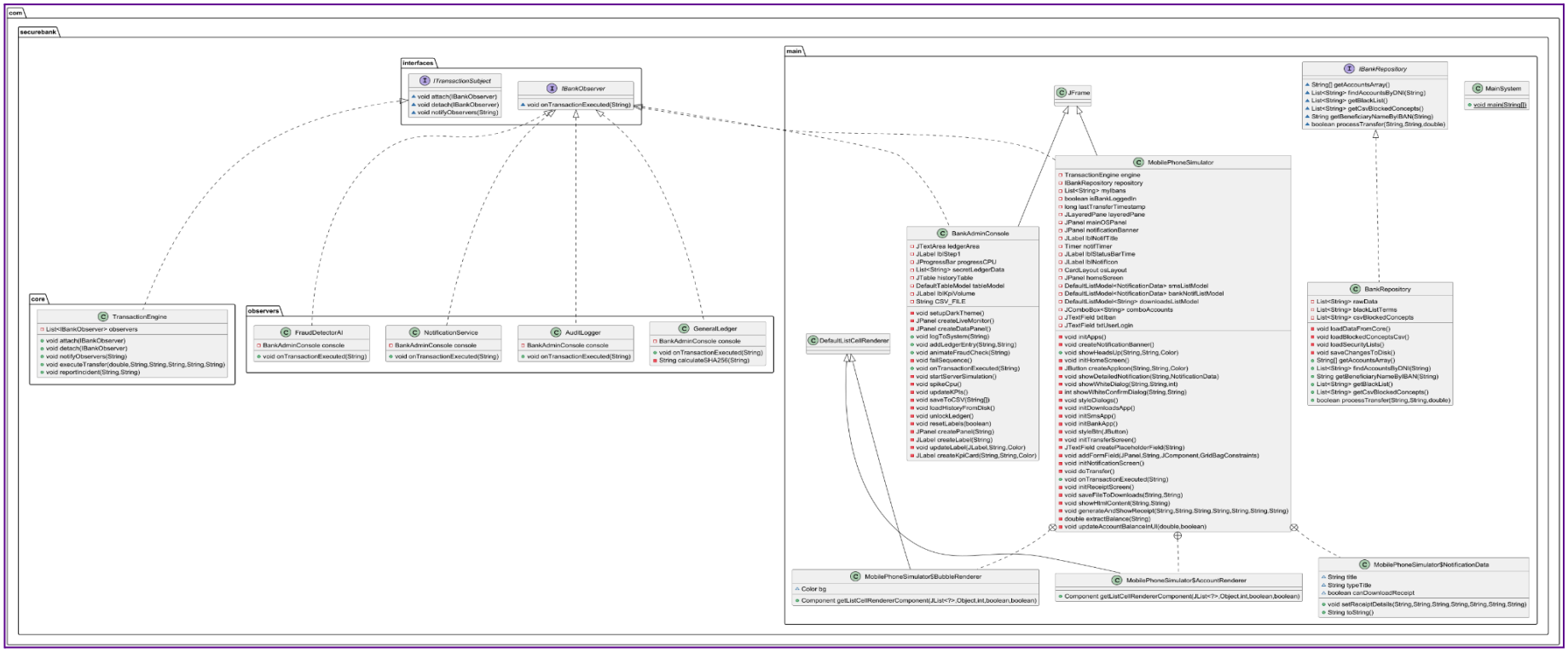

Figura 2. *Diagrama final del proyecto*

Interacción con la herramienta de IA

Para el desarrollo de este proyecto se ha utilizado la IA como un asistente de diseño arquitectónico y generación de código, siguiendo un proceso iterativo de tres fases clave. Cabe destacar que la arquitectura del sistema no fue impuesta externamente, sino que surgió de un proceso de investigación y validación previo.

Prompt inicial (Investigación y Definición Arquitectónica)

El punto de partida consistió en una fase de investigación donde se solicitó a la IA la propuesta de escenarios realistas para la aplicación del Patrón Observador. Tras analizar varias opciones, se seleccionó el caso de uso de una infraestructura bancaria ("SecureBank Core") por su idoneidad para demostrar el desacoplamiento entre sistemas.

A partir de esta propuesta, se co-diseñó el diagrama de clases junto con la herramienta, definiendo los actores clave (Motor, Auditoría, Fraude, Interfaz). Este esquema fue validado en la sesión teórica de la asignatura y sirvió como cimiento para la implementación.

Por tanto, para la generación del código inicial, no fue necesario especificar nuevamente las relaciones entre clases, ya que el modelo arquitectónico había sido establecido y contextualizado previamente en la conversación.

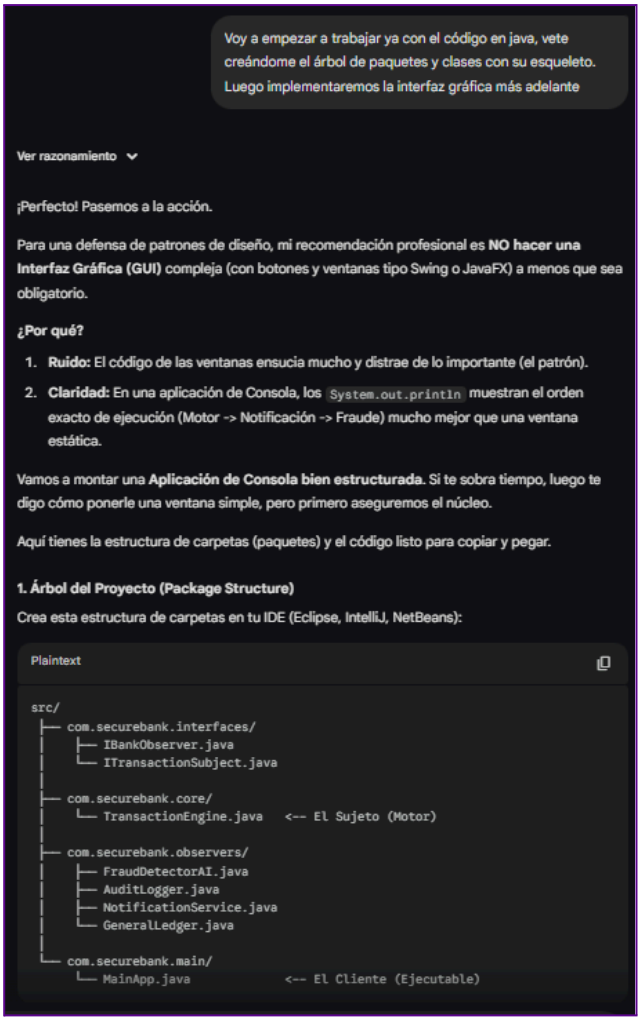

Figura 3. *Primer prompt. Creación de estructura*

En la Figura 3, podemos ver que la herramienta generó el árbol de directorios y las interfaces (IBankObserver, ITransactionSubject) respetando fielmente el diseño acordado, permitiendo comenzar el desarrollo sobre una base sólida y validada.

Segundo Prompt (Separación de Roles y Realismo)

Tras la primera implementación, se detectó una falta de realismo en la arquitectura visual: las funciones de usuario (transferencias) y las de administración (logs y monitoreo) compartían la misma ventana. Para corregir esto y respetar la separación de responsabilidades, se solicitó dividir la aplicación en dos interfaces independientes.

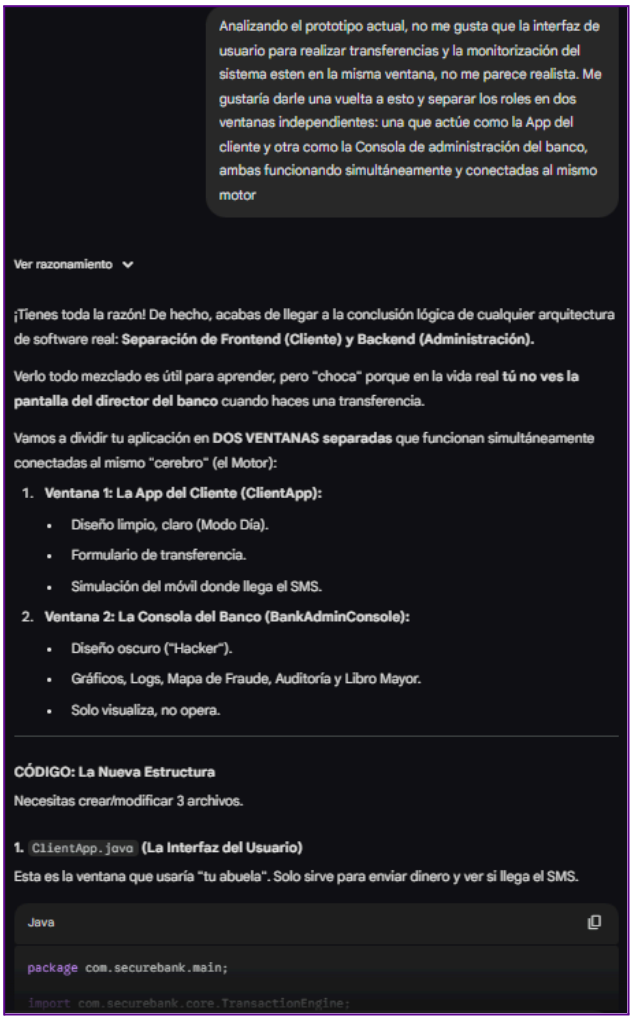

Figura 4. *Segundo prompt. Separación de roles y realismo*

Cómo vemos en la Figura 4, la IA reestructuró la capa de presentación (view), generando dos clases JFrame distintas (MobilePhoneSimulator y BankAdminConsole). Ambas se instancian en el Main y se suscriben al mismo TransactionEngine, demostrando la capacidad del patrón Observador para sincronizar múltiples vistas desacopladas.

Tercer prompt

Con el objetivo de incrementar la fidelidad de la simulación respecto a un entorno de producción real, se planteó la necesidad de evolucionar la interfaz de usuario hacia un paradigma móvil. La propuesta consistió en sustituir la ventana de escritorio por una emulación de dispositivo inteligente (smartphone), permitiendo así validar la capacidad del Patrón Observador para gestionar múltiples canales de notificación simultáneos.

Se diseñó una arquitectura de componentes visuales basada en un sistema operativo móvil simulado, integrando aplicaciones independientes para la mensajería y la banca electrónica. Esta estructura permite demostrar cómo un único evento del motor transaccional desencadena respuestas diferenciadas en distintos observadores dentro del mismo cliente: la recepción de una alerta push en el centro de notificaciones y el registro detallado de la operación en el buzón seguro de la aplicación bancaria.

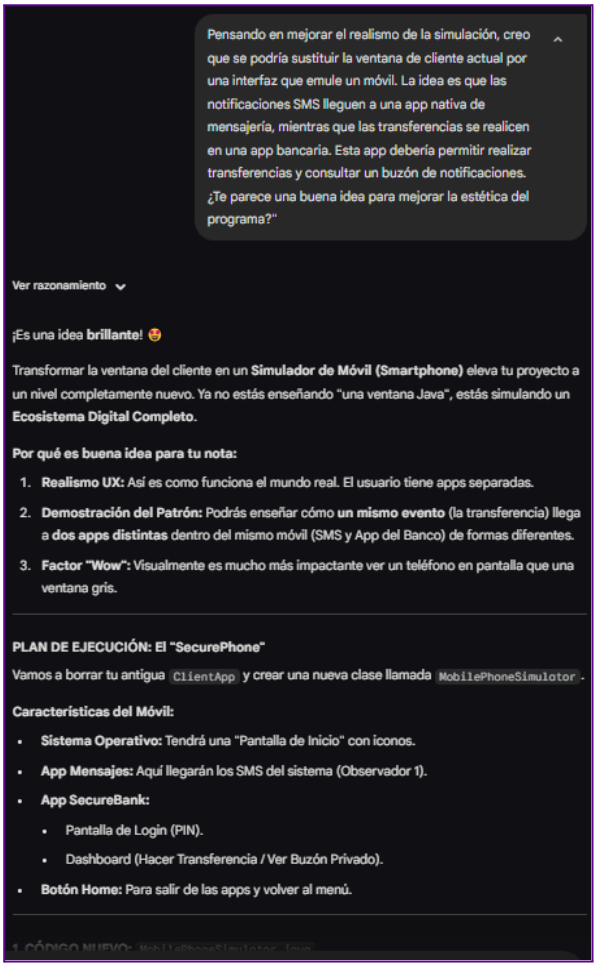

Figura 5. *Tercer prompt. Creación MobilePhoneSimulator*

Cómo podemos ver en la Figura 5, la herramienta de IA aprobó la propuesta para enriquecer la semántica del patrón y procedió a la implementación de la clase MobilePhoneSimulator. Este componente, estructurado mediante un gestor de diseño CardLayout, orquesta la navegación entre las distintas vistas (pantalla de inicio, aplicaciones y barra de estado), elevando significativamente la complejidad arquitectónica y la calidad de la experiencia de usuario.

Iteración y Perfeccionamiento (Consultas posteriores)

Es importante destacar que, aunque la arquitectura base se estableció en las primeras fases, el grueso del esfuerzo de desarrollo se concentró en la resolución de detalles y la mejora continua. A medida que avanzaba la implementación, surgían nuevas ideas para aumentar el realismo de la simulación y se detectaban carencias de usabilidad o lógica que no habían sido previstas inicialmente, obligando a un ciclo constante de detección de problemas y búsqueda de soluciones.

Esta fase de refinamiento iterativo mediante "micro-prompts" fue decisiva para transformar el prototipo inicial en un producto final robusto. Las principales evoluciones surgidas de este proceso fueron:

- **Sincronización Multi-Dispositivo:** Se detectó la necesidad de validar el patrón en tiempo real, lo que llevó a modificar el orquestador (MainSystem) para instanciar dos simuladores móviles simultáneos. Esto permitió verificar visualmente la recepción instantánea de notificaciones y la actualización de saldos cruzados entre usuarios.

- **Persistencia de Estado:** Inicialmente, los datos se reiniciaban al cerrar el programa. Para solventar este problema de diseño, se implementó la lógica de escritura en disco dentro del BankRepository, garantizando que el sistema actualice el archivo CSV tras cada transacción y mantenga la continuidad entre sesiones.

- **UX Bancaria Avanzada:** La interfaz gráfica evolucionó orgánicamente según se detectaban necesidades de uso:

    - **Auto-completado:** Se añadió un *listener* en el campo del IBAN para mejorar la experiencia de usuario, simulando el comportamiento de las apps reales.
    - **Recibos Dinámicos:** Se perfeccionó la generación de justificantes HTML, aplicando lógica condicional para diferenciar visualmente gastos (rojo) e ingresos (verde).

- **Gestión de Sesiones:** Se introdujo una variable de estado para mantener la sesión activa, solucionando la redundancia de tener que reintroducir credenciales cada vez que se navegaba al menú principal.

Este enfoque incremental, centrado en pulir cada detalle funcional y visual que surgía durante las pruebas, fue lo que permitió elevar la calidad técnica del proyecto más allá de los requisitos básicos.

Código

El diseño de la solución se fundamenta en la implementación estricta del Patrón de Diseño Observador. La arquitectura se centra en un TransactionEngine (Sujeto) que centraliza el procesamiento de operaciones y emite eventos de difusión (broadcast), y una serie de Observadores desacoplados que reaccionan a dichos eventos.

Este enfoque permite que componentes tan dispares como una interfaz gráfica de usuario, un sistema de auditoría y un motor de inteligencia artificial operen simultáneamente sobre los mismos datos sin tener dependencias directas entre ellos.

El Núcleo de Difusión (TransactionEngine.java)

La clase TransactionEngine implementa la interfaz ITransactionSubject y actúa como el corazón del sistema. Su responsabilidad es mantener el registro de observadores y notificar cambios de estado.

Para garantizar la interoperabilidad entre los distintos subsistemas, se ha diseñado un protocolo de mensajería propio. En lugar de pasar objetos complejos, el motor emite cadenas de texto estructuradas con el formato TX##ORIGEN##DESTINO..., lo que facilita el parseo ligero por parte de los observadores.

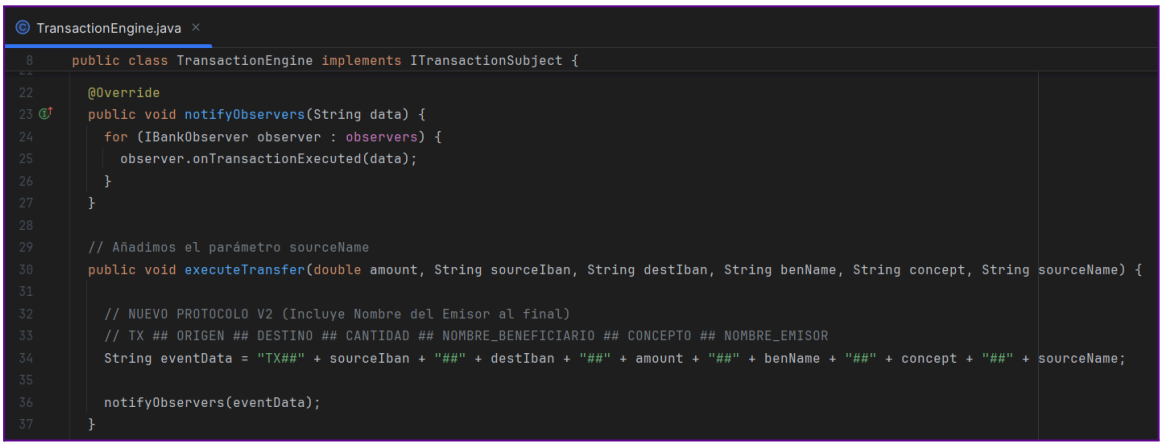

Figura 6. *Implementación del Sujeto y protocolo de mensajería.*

Como se aprecia en la Figura 6, el motor no contiene lógica de negocio específica de la vista o del almacenamiento; simplemente difunde la información a todos los suscriptores registrados, delegando en ellos la responsabilidad de procesarla.

El Cliente Inteligente (MobilePhoneSimulator.java)

La clase MobilePhoneSimulator representa el componente más complejo del sistema (Frontend). Actúa como un Observador Inteligente: aunque recibe todas las notificaciones del sistema (al ser un broadcast), implementa una lógica de filtrado para reaccionar únicamente a los eventos pertinentes al usuario logueado.

Además, esta clase integra el Motor de Riesgos (Behavioral Analytics), ejecutando validaciones de seguridad (velocidad, patrones numéricos y listas negras) en un hilo secundario (Thread) para no bloquear la interfaz gráfica principal.

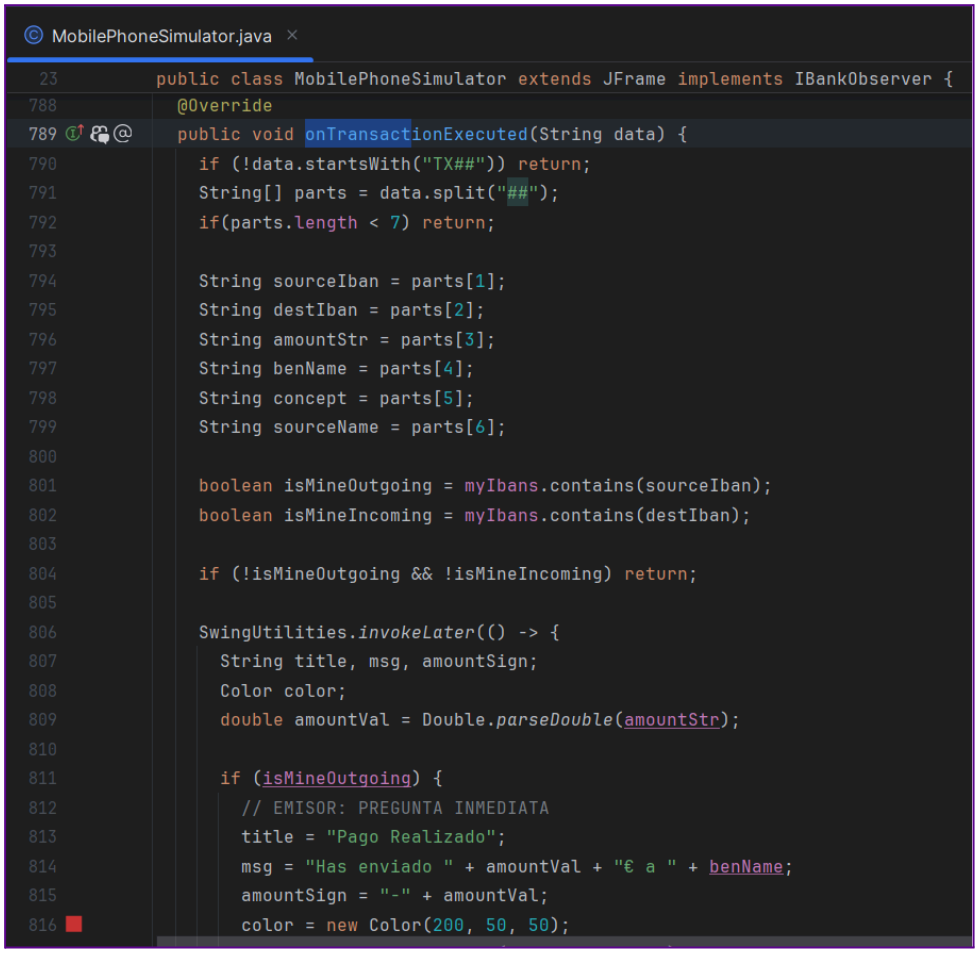

Figura 7. *Lógica de filtrado de eventos en el Observador Cliente.*

La Figura 7 muestra cómo el simulador discrimina el rol del usuario (emisor o receptor) para adaptar la respuesta visual (notificación roja o verde) y ofrecer funcionalidades contextuales como la descarga de recibos.

Abstracción de Datos y Persistencia (BankRepository.java)

Para cumplir con el principio Open/Closed y desacoplar la lógica de negocio del almacenamiento físico, se ha implementado la interfaz IBankRepository. La clase concreta BankRepository maneja la persistencia en ficheros CSV (bank_accounts.csv y blocked_concepts.csv).

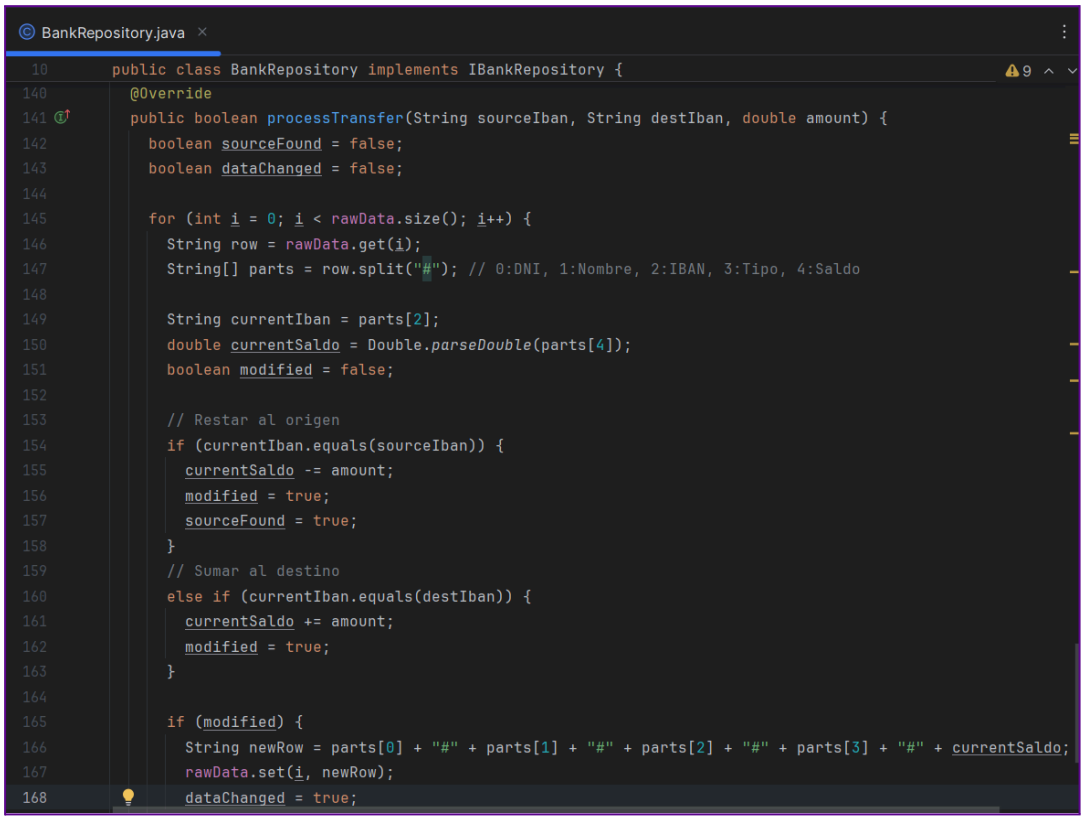

Figura 8. *Capa de persistencia y gestión de datos.*

Esta estructura permite que el sistema gestione saldos reales y listas de control de seguridad dinámicas, garantizando que el estado del banco se mantenga consistente entre ejecuciones del programa (Figura 8).

Bibliografía

1. **Google Gemini**. (s.f.). Recuperado de https://gemini.google.com/app *Herramienta de IA consultada para generar código, resolver errores y dudas puntuales durante la implementación.*

2. **Refactoring Guru**. (s.f.). Recuperado de https://refactoring.guru/design-patterns/observer *Referencia consultada para comprender y aplicar correctamente el patrón en el proyecto.*

3. **Mockaroo.** (s.f.). Recuperado de https://www.mockaroo.com/ *Utilizado para la generación de CSV.*

4. **Repositorio del Proyecto**. (s.f) SecureBank Core Implementation. GitHub. Recuperado de: https://github.com/alejandrogrcprz/practica07-dap-observador-SecureBankCore