Análisis del Desarrollo del Taller: Sistema de Gestión de Biblioteca

Juan David Meza Pimentel

Docente

Claudia Isabel Cáceres

Universidad Autónoma De Bucaramanga

Ingeniería Biomédica

Programación de Computadores

2025

1. Introducción y Objetivo del Proyecto

El presente documento detalla el análisis y las decisiones tomadas durante el desarrollo del proyecto "Sistema de Gestión de Biblioteca". El objetivo principal fue crear una aplicación de consola en Java para automatizar el control de libros y revistas, incluyendo el registro de ítems, gestión de préstamos, devoluciones y la generación de reportes básicos, cumpliendo con los requisitos especificados. La meta era desarrollar una solución funcional, bien estructurada y que aplicara correctamente los principios de la Programación Orientada a Objetos (POO).

2. Diseño y Arquitectura: Programación Orientada a Objetos (POO)

- ### Clase Padre publicación:
    - **Decisión:** Se creó una clase abstracta publicación para encapsular los atributos y comportamientos comunes a todos los ítems gestionables por la biblioteca.
    - **Justificación:** Esta abstracción evita la duplicación de código y establece un contrato común para todas las subclases. Los atributos comunes identificados fueron: codigo (identificador único), titulo, disponible (booleano para estado de préstamo) y diasPrestado (para llevar control del tiempo de préstamo).
    - **Métodos Comunes:** Se incluyeron métodos como prestar() y devolver() con una implementación base en Publicacion, ya que la lógica fundamental de cambiar el estado disponible y actualizar diasPrestado es similar para todos los tipos. El método mostrarInfo() y toStringParaArchivo() se declararon abstractos (o con implementación base para ser sobrescrita) para forzar a las subclases a proporcionar su representación específica.
- **Subclases para Libros: Libro, LibroFisico y LibroDigital:**
    - **Decisión:** Siguiendo el requisito, se creó una jerarquía para los libros. Primero, una clase abstracta Libro que hereda de Publicacion. Luego, las clases concretas LibroFisico y LibroDigital que heredan de Libro.
    - **Justificación:**
        - La clase Libro sirve como un nivel intermedio de abstracción para agrupar atributos comunes a todos los libros. Respondiendo a la pregunta del requisito "¿Qué dato falta?", se identificó el autor como un atributo esencial para cualquier libro y se incluyó en la clase Libro.
        - Las subclases LibroFisico y LibroDigital permiten diferenciar estos dos tipos de libros, tal como se solicitó. Para darles una distinción más allá del tipo, se añadieron atributos específicos: ubicacion (String) para LibroFisico y formato (String, ej: "PDF", "EPUB") para LibroDigital. Esto permite almacenar información relevante para cada tipo y potencialmente extender su funcionalidad de forma independiente en el futuro.
- **Subclase para Revistas: Revista:**
    - **Decisión:** Se creó la clase Revista heredando directamente de Publicacion.
    - **Justificación:** Respondiendo a la pregunta del requisito "¿qué datos debería incluir?", se consideraron atributos específicos para las revistas, como numeroEdicion (int) y

anioPublicacion (int), ya que son datos identificativos y relevantes para este tipo de publicación.

- **Encapsulamiento:**
    - **Decisión:** Todos los atributos de las clases se declararon como protected (en la clase padre para acceso de subclases) o private (en subclases para atributos propios), y se proporcionaron métodos public getters (y setters selectivos donde era apropiado y seguro).
    - **Justificación:** Esto protege la integridad de los datos de los objetos, permitiendo un acceso controlado y facilitando futuras modificaciones internas sin afectar el código que utiliza las clases.
- **Polimorfismo:**
    - **Decisión:** Los métodos mostrarInfo() y toStringParaArchivo() son ejemplos clave de polimorfismo.
    - **Justificación:** Permiten que cada subclase (LibroFisico, LibroDigital, Revista) proporcione su propia implementación para mostrar su información y para generar la cadena de texto que se guardará en el archivo. Esto simplifica el código en la clase GestionBiblioteca, ya que puede tratar a todos los objetos de Publicacion de manera uniforme (e.g., al iterar la lista para mostrar información o guardar en archivo) y Java automáticamente invoca la versión correcta del método según el tipo real del objeto.

3. Gestión de Datos: Lectura y Escritura en Archivo

- **Elección del Formato de Archivo:**
    - **Decisión:** Se optó por un archivo de texto plano (biblioteca.txt) con campos delimitados por punto y coma (;). Cada línea representa un objeto.
    - **Justificación:** Aunque el requisito mencionaba "archivo de objetos", el ejemplo proporcionado (LIBRO;001;...) y la sugerencia de usar split(";") apuntaban claramente hacia un formato de texto. Este enfoque ofrece varias ventajas:
        - **Simplicidad:** Fácil de implementar la lectura y escritura.
        - **Legibilidad:** Los datos pueden ser inspeccionados y modificados manualmente si es necesario (para depuración o correcciones rápidas).
        - **Portabilidad:** Los archivos de texto son universalmente compatibles.
        - **Cumplimiento de Sugerencias:** Se utilizó split(";") tal como se sugirió.
    - Para distinguir entre los diferentes tipos de publicaciones al leer el archivo, el primer campo de cada línea actúa como un identificador de tipo (e.g., "LIBROFISICO", "LIBRODIGITAL", "REVISTA").
- **Proceso de Carga (cargarDatosDesdeArchivo):**
    - **Decisión:** Al iniciar la aplicación, se leen los datos del archivo. Se utiliza BufferedReader para una lectura eficiente.

- **Justificación:** Cada línea se procesa, se divide usando split(";"), y se utiliza una estructura switch basada en el primer campo (tipo de publicación) para instanciar el objeto correcto (LibroFisico, LibroDigital, o Revista) con los datos correspondientes. Se incluyó manejo básico de errores para líneas malformadas o datos incorrectos (e.g., NumberFormatException).

- **Proceso de Guardado (guardarDatosEnArchivo):**
  - **Decisión:** Al salir de la aplicación (opción "Salir" del menú), todos los datos del inventario en memoria se escriben de nuevo al archivo, sobrescribiendo el contenido anterior. Se utiliza BufferedWriter.
  - **Justificación:** Esto asegura la persistencia de los cambios realizados durante la sesión (préstamos, devoluciones). Cada objeto utiliza su método toStringParaArchivo() polimórfico para generar la cadena de texto en el formato esperado.

4. Estructura de Datos Interna

- **Decisión:** Se utilizó un ArrayList<Publicacion> (nombrado inventario) para almacenar todas las publicaciones en memoria.
- **Justificación:**
  - ArrayList es una estructura de datos dinámica que permite añadir o eliminar elementos fácilmente.
  - Al ser de tipo Publicacion, puede contener objetos de cualquiera de sus subclases (LibroFisico, LibroDigital, Revista) gracias al polimorfismo. Esto simplifica la gestión de todas las publicaciones en una única colección.

5. Interfaz de Usuario y Lógica de la Aplicación (GestionBiblioteca)

- **Menú de Consola:**
  - **Decisión:** Se implementó un menú interactivo en la consola utilizando un bucle do-while y una estructura switch para procesar las opciones del usuario. Se utilizó Scanner para la entrada de datos.
  - **Justificación:** Esta es una forma estándar y sencilla de interactuar con el usuario en aplicaciones de consola, cumpliendo con el requisito de un menú con opciones específicas.
- **Funcionalidades del Menú:**
  - **Ver disponibles/prestados:** Iteran sobre el ArrayList y, usando el método isDisponible() y mostrarInfo(), presentan la información requerida.
  - **Prestar/Devolver:** Buscan la publicación por código. Si se encuentra y la operación es válida (e.g., prestar algo disponible, devolver algo prestado), se invocan los métodos prestar() o devolver() del objeto Publicacion correspondiente. Se solicita al usuario el número de días para el préstamo.
  - **Salir:** Invoca guardarDatosEnArchivo() antes de terminar la ejecución.

6. Funcionalidades Específicas

- **Conversión de Número de Días a Texto:**
    - **Decisión:** Se creó una clase Utilidades con un método estático convertirNumeroATexto(int numero).
    - **Justificación:** Esto cumple con el requisito de mostrar los días de préstamo en formato de texto (e.g., "tres días" en lugar de "3 días") al listar los libros prestados. Hacerlo un método estático en una clase de utilidad lo hace fácilmente accesible desde cualquier parte del código sin necesidad de instanciar un objeto, y mantiene esta lógica separada de las clases principales del dominio. Se implementó para números del 0 al 30, cubriendo periodos de préstamo comunes.
- **Uso de instanceof y Casting:**
    - **Decisión:** El uso de instanceof y casting explícito se minimizó en la lógica principal del menú gracias al polimorfismo (especialmente con mostrarInfo() y toStringParaArchivo()). Sin embargo, son implícitamente necesarios en el método cargarDatosDesdeArchivo() para determinar qué tipo de objeto crear a partir de los datos leídos, aunque no se vea la palabra instanceof directamente, la lógica del switch cumple una función similar de discriminación de tipo.
    - **Justificación:** Un buen diseño OO busca reducir la necesidad de instanceof y casting mediante el uso de polimorfismo, lo que lleva a un código más flexible y mantenible.

7. Conclusión

El desarrollo del Sistema de Gestión de Biblioteca se adhirió a los requisitos especificados, con un fuerte énfasis en la correcta aplicación de los principios de la Programación Orientada a Objetos. Las decisiones de diseño se tomaron buscando un equilibrio entre el cumplimiento de los requisitos, la claridad del código, la mantenibilidad y la eficiencia. La estructura de clases permite una fácil extensión futura (e.g., añadir nuevos tipos de publicaciones o funcionalidades específicas para cada tipo). La persistencia de datos mediante un archivo de texto plano resultó ser una solución adecuada y práctica para el alcance del proyecto.