**Informe Final Desafío 2**

**Integrantes:** Sara Echeverri Giraldo, Luis Miguel Alzate

**Curso:** Informática II

**Link del video:** https://youtu.be/UGh8VR9C-Jk

1. **Introducción**

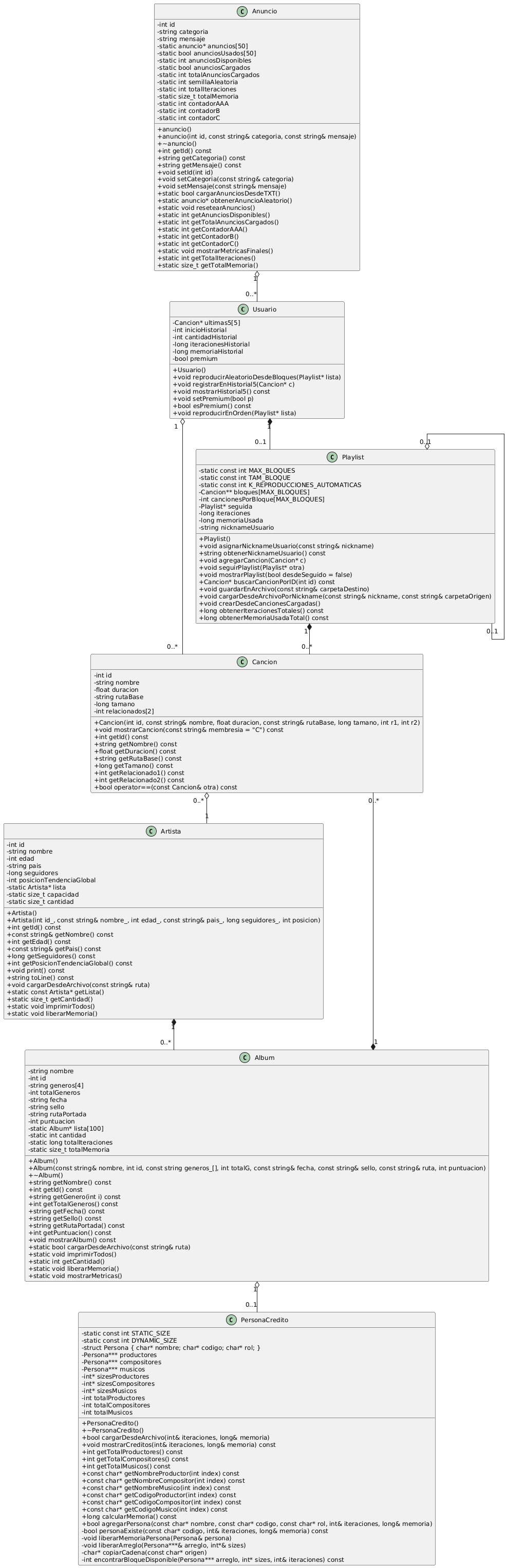
El presente informe documenta de forma detallada el desarrollo técnico del Desafío 2 de Informática II. El objetivo de este proyecto se fundamentó en el desarrollo un programa que emule el funcionamiento de un servicio de streaming, utilizando POO. Nuestra plataforma: UdeATunes, una aplicación de simulación de reproducción musical desarrollada en C++. El centro principal fue construir un sistema modular que permitiera gestionar usuarios, canciones, listas de reproducción, álbumes, artistas y anuncios, priorizando la claridad de responsabilidades, el control de recursos (memoria y operaciones) y la persistencia sencilla mediante archivos de texto.

1. **Descripción general de la arquitectura**

El sistema previamente descrito se organizó en las siguientes capas:

* Capa de Dominio: clases que representan entidades (Canción, Álbum, Artista, Anuncio).
* Capa de Negocio: clases que encapsulan el comportamiento (Playlist, Usuario).
* Capa de Orquestación: módulo de funcionalidad que gestiona menús, autenticación y flujo principal.
* Persistencia: archivos .txt por tipo de entidad (canciones, álbumes, artistas, anuncios, usuarios, playlist por usuario).

Se buscó que cada módulo tuviese una responsabilidad única y bien definida, lo que facilitó las pruebas unitarias y la depuración.

1. **Diagrama de clases**
2. **Justificación del uso de archivos .txt**

Se eligieron los archivos de texto por las siguientes razones prácticas:

* Visibilidad y trazabilidad: El contenido es legible y auditable con cualquier editor, lo que facilitó la verificación manual durante las pruebas y la evaluación.
* Simplicidad de formato: El uso de un delimitador fijo “|” permitió un análisis controlado por nosotros como programadores, sin dependencias externas.
* Portabilidad: Los archivos .txt funcionan en cualquier sistema y son fáciles de versionar y compartir.
* Gestión de errores: Los errores de formato se detectaron rápidamente y las líneas erróneas se pudieron excluir con mensajes claros.
* Fin de aprendizaje: Facilitó la comprensión del flujo de carga de datos sin capas adicionales (por ejemplo, bases de datos o serialización binaria) que dificultarían el proceso.

No obstante, en el proceso de desarrollo también se consideraron alternativas (JSON, CSV, base de datos integrada), pero se priorizó la claridad y el control total sobre el análisis y la validación.

1. **Módulos y detalles de implementación**

La implementación, la lógica y la justificación de cada decisión se explican módulo por módulo a continuación:

**5.1 Canción (modelo base)**

La responsabilidad de esta clase es representar una pista de música con atributos mínimos pedidos en el documento reglamentario y de contextualización: ID, nombre, duración, ruta base, tamaño y relaciones. En cuanto a las decisiones clave tomadas se puede decir lo siguiente: Se mantuvieron los atributos compactos y explícitos para generar rutas de archivo basadas en la pertenencia (calidad), se incluyó un operador de comparación para detectar duplicados por ID o coincidencia de atributos, utilizado por la lista de reproducción para evitar inserciones repetidas.

Por otro lado, evitamos un almacenamiento excesivo en cada objeto; los punteros de objeto se gestionan para que el mismo objeto Canción pueda ser referenciado por múltiples listas de reproducción sin duplicación, lo cual sin duda mejora la eficiencia de la memoria al no duplicar objetos y facilita la búsqueda rápida de ID dentro de los bloques.

**5.2 cargarcanciones (módulo de carga)**

La responsabilidad de este módulo es leer el archivo de la canción, analizar cada línea y crear los objetos Canción en una matriz global basada en bloques. En cuanto a las decisiones clave tomadas se puede decir lo siguiente: Se definió una estructura basada en bloques (matriz de punteros con un número limitado de bloques y tamaño por bloque), lo cual facilitó la división de la colección y limitó los costos de expansión. También se tuvo en cuenta el análisis manual de cadenas para extraer campos y convertirlos a tipos numéricos, con un contador de iteraciones para medir el costo, así como la gestión explícita de contadores de iteraciones globales y memoria estimada para monitorización.

El impacto que pudimos notar con estas estrategias fue el siguiente: Se pudo obtener carga controlada, ya que si el archivo es demasiado grande, el sistema puede detectar límites de bloque y detenerlo con un mensaje, adicional, métricas integradas para el análisis y la eficiencia de la memoria.

**5.3 Playlist (gestor de listas)**

La responsabilidad de esta clase es mantener listas de reproducción por usuario, permitir agregar/quitar canciones, seguir otra lista de reproducción y guardar/cargar lista de reproducción desde archivo. Internamente mantiene un arreglo de bloques de punteros a Canción y contadores de canciones por bloque, soporta seguir otra lista de reproducción (relación por puntero), y cuando muestra, evita duplicados usando un registro local de IDs ya impresos.

Otro punto clave fue el guardado y carga por archivo por usuario (archivo por nickname), pues facilitó la persistencia y restauración de la lista de reproducción personal. Tuvimos como puntos clave la separación clara entre lista propia y playlist seguida (no mezclar objetos, sino referenciar), evitar duplicados al agregar usando comparación por id o por atributos significativos y el manejo manual de memoria al asignar bloques (nuevo/eliminar) y ajustar métricas de memoria.

De esta manera, pudimos tener control sobre crecimiento y límites; la separación por bloques hace predecible el uso de memoria y al seguir otra lista de reproducción, se puede combinar visualmente sin mezclar datos físicos, manteniendo la integridad individual.

**5.4 Usuario (control de interacción)**

La responsabilidad de esta clase es representar la sesión de un usuario (datos personales, tipo de membresía), gestionar el historial y las acciones de reproducción. En cuanto a la estructura y lógica se puede decir lo siguiente: se tienen los atributos de perfil (nickname, contraseña, país, ciudad, fecha, usuario seguido) y bandera de membresía, también una circular histórica de 5 punteros a Cancion, operado con índices inicio y cantidad; otro punto importante fueron los métodos para reproducir en orden y reproducción aleatoria desde bloques de Playlist, integración con anuncios y medición de métricas.

Se mantuvo la validación de inicio de sesión en el módulo de funcionalidades, pero proporcionamos en Usuario un método simple para validar credenciales a nivel de objeto. También el registro de memoria e iteraciones en operaciones críticas (reproducción, historial) para poder mostrar métricas al cliente y la simulación temporal con crono para representar la duración de reproducción; se empleó una espera acotada para no afectar la interacción con el menú.

El impacto que tuvieron estas decisiones se pudieron notar en que el usuario actúa como controlador ligero de reproducción, sin asumir responsabilidades de persistencia global, además de que facilitó la experiencia diferenciada para premium (calidad, opción de seguir listas de reproducción, etc).

**5.5 Álbum (catálogo de discotecas)**

La responsabilidad de esta clase es almacenar metadatos de álbumes y permitir su consulta. De la estructura y lógica podemos decir que se formó de la siguiente manera: Los campos tenidos en cuenta fueron nombre, id, géneros (máx. 4), fecha, sello, ruta de portada, puntuación, adicional, la carga se hizo desde archivo en un arreglo estático de punteros, con validación por línea y una función para mostrar un álbum, incluyendo ruta de portada. Se tuvo en cuenta limitar número de géneros y preasignar espacio para evitar estructuras complejas, contabilizando iteraciones y memoria en la carga para poder evaluar el coste de catálogo. Esto se reflejó con el hecho de que el catálogo fue fácil de mostrar y relacionar con canciones y artistas y la ruta de portada se mantuvo como cadena, lista para integrarse en futuras interfaces gráficas.

**5.6 Artista (catálogo dinámico)**

La responsabilidad de esta clase es almacenar artistas con crecimiento controlado y permitir su listado. Se utilizó un arreglo dinámico que inicia con capacidad pequeña y duplica capacidad cuando sea necesario, la carga se hizo desde archivo con validaciones y evitación de duplicados, además, se tuvo presente el crecimiento por duplicación para minimizar número de reasignaciones usando una estrategia con coste amortizado eficiente la cual fue búsqueda lineal para detectar duplicados al cargar que fue suficiente por la escalada de datos prevista. Estas decisiones permitieron un catálogo de artistas que pueda crecer con el tiempo sin reescritura extensa y mantuvo simplicidad en la gestión de memoria, con operaciones específicas de liberación.

**5.7 Anuncio (rotación y ponderación)**

La responsabilidad de esta clase es cargar anuncios, mantener una base de datos disponible y seleccionar anuncios de forma ponderada por categoría. Se trabajó mediante un arreglo estático donde están estos anuncios, bandera de usados ​​y contador de disponibles teniendo en cuenta el peso por categoría (AAA, B, C) para selección ponderada, adicional se utilizó un generador pseudoaleatorio con semilla interna para reproducibilidad para priorizar anuncios de mayor categoría y marcar anuncios como usados ​​para evitar la repetición hasta agotar la base de datos, y permitir reiniciar. Lo anterior permitió una experiencia similar a un servicio real donde aparecen anuncios con frecuencia controlada sobre repetición y balance por categoría.

1. **Módulo de orquestación: funcionalidades.cpp**

***Responsabilidad:*** coordinar carga inicial, autenticación, menús y flujo de acciones del usuario.

***Clave de tareas:***

* Cargar datos iniciales (canciones, álbumes, artistas, anuncios).
* Gestionar inicio de sesión: pedir nickname y contraseña, validar contra usuarios.txt, instanciar Usuario.
* Mostrar menús adaptados a tipo de cuenta (estándar vs premium).
* Ejecutar opciones del usuario (editar lista de reproducción, reproducir, ver álbumes, seguir usuarios, etc.).
* Persistir cambios en archivos de lista de reproducción por usuario cuando corresponda.
* Mostrar métricas finales y liberar recursos al salir.

***Decisiones clave:***

* Mantener la lógica de presentación y control en este módulo para evitar mezclarla con el dominio.
* Evitar uso de estructuras diferentes en este archivo: trabaja con arreglos y punteros para coherencia con el resto del proyecto.
* Proporcionar mensajes claros y redirecciones automáticas (por ejemplo, si no existe una lista de reproducción al intentar reproducir, redirigir a edición, etc).

***Impacto:*** centraliza la interacción y permite modificar flujos sin tocar la lógica interna de las clases.

1. **Medición y control de recursos**

En todo el proyecto se llevan dos métricas principales:

* Contador de iteraciones: incrementado en operaciones de parseo, búsqueda y recorridos. Sirve para comparar costos entre operaciones y para detectar puntos de optimización.
* Estimación de memoria: calculada sumando tamaños de objetos y capacidades relevantes de cadenas. Da una aproximación del consumo total en cada módulo.

Estas métricas se usan luego de funcionalidades y al terminar el programa, ofreciendo información útil para ajustes de rendimiento.

1. **Pruebas, validaciones y manejo de errores**

Se definieron y aplicaron pruebas prácticas:

* Carga de datos: archivos con líneas válidas e inválidas para verificar que se omiten errores y se logreen avisos.
* Manipulación de listas de reproducción: agregar canciones válidas, intento de agregar duplicadas, eliminación inexistente (redirección a agregar).
* Reproducción: modos orden y aleatorio, verificación de anuncios cada dos canciones, historial correcto de las últimas 5 canciones.
* Gestión de memoria: pruebas con cargas grandes para verificar que no hubiesen fugas y liberar recursos al cerrar.
* Manejo de errores: verificación de apertura de archivos, validación de conversiones numéricas, comprobación de punteros nulos y límites de índices.

1. **Problemas encontrados y soluciones**

Durante el desarrollo del proyecto surgieron varios desafíos técnicos que requirieron análisis y ajustes en la implementación:

* Uso de la librería <chrono>: uno de los principales retos fue implementar la simulación del tiempo de reproducción de las canciones. El manejo de esta librería fue complejo, ya que era un recurso nuevo para nosotros y requería comprender conceptos como medición de intervalos, relojes de alta resolución y diferencias temporales. Sin embargo, a pesar de la dificultad inicial, logramos integrarla correctamente para representar pausas reales durante la reproducción de cada canción. Esto permitió dar una sensación más realista al proceso sin interferir con el flujo general del programa.
* Gestión manual de memoria: otro reto importante fue el control de memoria con punteros y arreglos dinámicos, especialmente al trabajar sin estructuras predefinidas. Se resolvió aplicando una disciplina estricta en la creación y eliminación de objetos, así como verificaciones antes de liberar memoria para evitar fugas o accesos inválidos.
* Interdependencia entre módulos: al tener muchas clases que dependen entre sí, fue necesario organizar cuidadosamente los encabezados y las declaraciones externas para evitar errores de compilación y referencias cruzadas. Esto se solucionó separando las responsabilidades por archivo y utilizando funciones de carga global en los casos necesarios.

1. **Conclusiones**

El proyecto implementó un modelo consistente de una plataforma de streaming en C++. Cada módulo se diseñó con una responsabilidad clara: Usuario gestiona la interacción y reproducción, Playlist las colecciones personales, Álbum y Artista los catálogos, y Anuncio la rotación publicitaria. La elección de archivos .txt garantizó transparencia y facilidad de prueba en el desarrollo inicial. Además, el control explícito de memoria y la medición de iteraciones ofrecen una base sólida para analizar el rendimiento y priorizar mejoras.