UNIVERSIDAD DE ANTIOQUIA

1 8 0 3

Facultad de Ingeniería
Departamento de Electrónica y telecomunicaciones

INFORME DE ANÁLISIS Y DISEÑO
UdeATunes: Sistema de Gestión de Streaming Musical

Brayan Camilo Silva Porras

Hamilton Alexander Suárez Pérez

2025 - 2

Análisis del problema

Introducción

UdeATunes es una plataforma de streaming musical que permite reproducir canciones bajo un modelo de negocio diferenciado: usuarios premium pagan por acceso sin publicidad y funcionalidades extras, mientras que usuarios estándar financian el servicio a través de publicidad.

Objetivo General

Desarrollar un sistema que modele las operaciones esenciales de una plataforma de música en línea: gestión de usuarios, artistas, álbumes, canciones, reproducción aleatoria, listas personalizadas y medición de eficiencia.

Requerimientos Funcionales

Funcionalidad	**Descripción**
Carga de datos	Leer desde archivos de almacenamiento permanente
Autenticación	Ingreso a la plataforma según tipo de usuario
Reproducción aleatoria	Seleccionar y reproducir canciones al azar con controles variados
Lista de favoritos	Gestión de colecciones personales (solo premium)
Medición de recursos	Conteo de iteraciones y consumo de memoria

Cuadro 1: Funcionalidades principales del sistema

Estrategia de solución

Enfoque Arquitectónico

El sistema se basará en Programación Orientada a Objetos pura, sin el uso de herencia ni de la biblioteca estándar de plantillas (STL). Para su implementación se tendrán en cuenta los siguientes principios:

- Estructuras de datos propias con manejo de memoria dinámica.
- Composición y asociación entre clases mediante el uso de punteros.
- Encapsulación estricta de los datos, garantizando la integridad y el control del acceso a los atributos internos.

Componentes Principales

El sistema se estructura en capas, cada una con responsabilidades claramente definidas:

- **Capa de Modelos:** Clases que representan las entidades del dominio, tales como Usuario, Artista, Canción, Álbum, entre otras.
- **Capa de Servicios:** Clases de utilidad encargadas de la persistencia de datos y la medición de recursos.
- **Capa de Orquestación:** Componente central que coordina y gestiona las operaciones complejas del sistema.
- **Capa de Presentación:** Interfaz basada en consola que permite la interacción directa con el usuario.

Estructura de Clases y Relaciones

Entidades de Dominio

Se implementarán ocho clases principales:

- **Fecha:** Gestión temporal (*año, mes, día*).
- **Crédito:** Representa a los participantes en la creación de una canción (*productor, músico, compositor*).
- **Canción:** Contenido reproducible con un ID de 9 dígitos (5 del artista, 2 del álbum y 2 de la canción).
- **Álbum:** Colección de canciones con metadatos (*géneros, puntuación, portada*).
- **Artista:** Creador de contenido con catálogo propio de álbumes.
- **Usuario:** Persona registrada con nickname único y membresía (*estándar o premium*).
- **ListaFavoritos:** Colección personalizada de canciones disponible únicamente para usuarios premium.
- **Publicidad:** Mensaje promocional con categoría de prioridad (*C, B, AAA*).

Relaciones entre Clases

Se empleará **composición** (relaciones 1:N) y **asociación** (referencias mediante punteros), según los siguientes vínculos:

- **Artista** → **Álbum**: un artista puede tener múltiples álbumes.
- **Álbum** → **Canción**: un álbum contiene múltiples canciones.
- **Canción** → **Crédito**: una canción tiene múltiples créditos asociados.

- **Usuario → ListaFavoritos**: solo los usuarios premium poseen una lista de favoritos.
- **Usuario → Histórico de Reproducción**: registro de las últimas K canciones escuchadas.

No se utilizará **herencia** entre clases. La diferencia entre tipos de usuarios se implementará mediante **composición**, a través de la presencia o ausencia de una *ListaFavoritos*.

Conceptos Avanzados de POO

- **Sobrecarga de operadores:** Cada clase principal (*Fecha*, *Canción*, *Usuario*) implementará al menos dos operadores:
  - Operador de igualdad `==` para comparar identidad y evitar duplicados.
  - Operador de orden `<` para permitir ordenamiento.
  - Operador de asignación `=` para realizar copias seguras.
- **Sobrecarga de métodos:** Se implementarán métodos con el mismo nombre pero diferente tipo o número de parámetros. Ejemplo: `obtenerRuta()` con variantes para retornar audio a *128 kbps*, *320 kbps* o según la *membresía del usuario*.
- **Constructores de copia:** Todas las clases implementarán constructores de copia explícitos para evitar compartir punteros entre instancias.
- **Funciones amigas:** Validadores externos que podrán acceder a miembros privados con el fin de comparar objetos sin exponer getters públicos (por ejemplo, para verificar unicidad).
- **Plantillas genéricas:** Clase `ArregloDinamico<T>` que actuará como contenedor genérico para instancias de cualquier tipo (*Canción, Usuario, Artista*, etc.).
- **Getters y Setters:** Métodos de acceso con validación implícita en los setters (por ejemplo, rechazo de tipos de membresía inválidos).

Cumplimiento de Requisitos

Requisito: Carga y Actualización de Datos

Se implementará la clase **GestorDatos**, responsable de la gestión de persistencia. Sus funciones principales son:

- Leer los archivos de almacenamiento permanente al iniciar el programa.
- Mantener los datos cargados en memoria durante la sesión.
- Escribir los cambios realizados al finalizar la ejecución (persistencia).

Los archivos manejados serán: `usuarios`, `artistas`, `álbumes`, `canciones`, `créditos`, `publicidad` y `favoritos`.

Requisito: Ingreso a la Plataforma

Se incluirá una funcionalidad de **autenticación** que permitirá:

- Validar la existencia del `nickname` ingresado.
- Distinguir entre usuarios **estándar** y **premium**.
- Mostrar un menú contextual según el tipo de membresía.
- Mantener la sesión activa durante el uso del sistema.

Requisito: Reproducción Aleatoria

Se desarrollará la clase **Reproductor**, encargada de la lógica de reproducción. Sus funcionalidades incluyen:

- Seleccionar una canción aleatoria entre todas las disponibles.
- Mostrar la ruta del archivo de audio según la calidad (*128 o 320 kbps*).
- Mostrar la ruta de la portada del álbum correspondiente.
- Gestionar los controles de reproducción: **siguiente**, **anterior**, **detener** y **repetir**.
- Validar la lógica de uso (por ejemplo: no hay "anterior" al inicio, no se repite si no hay canción en reproducción, etc.).
- Mostrar **publicidad** cada 2 canciones para usuarios estándar, rotando los anuncios para evitar repeticiones consecutivas.
- Implementar un **temporizador automático de 3 segundos** entre canciones (modo de testing).

Requisito: Mi Lista de Favoritos (Usuarios Premium)

Se implementará una funcionalidad exclusiva para usuarios premium que permitirá:

- **Editar:** agregar o eliminar canciones por ID, evitando duplicados (máximo 10,000 canciones).
- **Seguir:** agregar la lista de otro usuario premium a la propia.
- **Ejecutar:** reproducir las canciones en orden o de forma aleatoria, con retroceso permitido hasta 6 canciones previas.

Requisito: Medición de Recursos

Se desarrollarán métricas globales para evaluar el rendimiento del sistema:

- **Iteraciones:** contador estático que se incrementa en los bucles principales y se reinicia por funcionalidad.
- **Memoria:** acumulador que suma los bytes asignados en constructores y resta en destructores. Su cálculo es de complejidad $O(1)$ en cada operación.
- Las métricas se mostrarán al finalizar cada funcionalidad ejecutada.

Consideraciones de Eficiencia

Estructuras de Datos

Se emplearán **arreglos dinámicos propios** (sin uso de STL), diseñados para optimizar el rendimiento y la gestión de memoria. Sus características principales son:

- **Capacidad inicial:** 10 elementos.
- **Factor de crecimiento:** 1.5x, reduciendo la fragmentación de memoria.
- **Minimización de realocaciones:** el incremento progresivo de capacidad evita operaciones costosas de copia.

Algoritmos Optimizados

Para maximizar la eficiencia del sistema, se implementarán algoritmos con complejidades ajustadas a cada tipo de operación:

- **Búsqueda de canciones:** Implementación de una **tabla hash casera** que utiliza el **ID de 9 dígitos** como índice directo, logrando una complejidad de $O(1)$.
- **Búsqueda de usuarios:** Los usuarios se almacenarán en un **array ordenado por** `nickname`, permitiendo la aplicación de **búsqueda binaria** con complejidad $O(\log n)$.
- **Selección de publicidad:** Se precalcularán los **pesos probabilísticos** por categoría —C=1, B=2, AAA=3—, de modo que la selección se realice en tiempo constante $O(1)$.
- **Evitar copias innecesarias:** Los parámetros de solo lectura se pasarán por **referencia constante**, mientras que los punteros se usarán únicamente cuando sea necesario modificar o almacenar datos.

Restricciones Lógicas Validadas

El sistema garantizará la consistencia de los datos mediante validaciones lógicas internas:

- **Nicknames únicos:** validados mediante el operador `==` de la clase `Usuario`.
- **IDs de canción únicos:** comprobados con el operador `==` de la clase `Canción`.
- **IDs de álbum únicos dentro de cada artista.**
- **Sin duplicados en la lista de favoritos.**
- **Límite máximo de 10,000 canciones** por lista de favoritos.
- **ListaFavoritos** disponible únicamente para usuarios **premium**.
- **Calidad de audio:** 320 kbps para usuarios premium y 128 kbps para usuarios estándar.

Etapas de Desarrollo

El desarrollo del sistema se organizará en siete etapas, distribuidas por semanas, con el fin de garantizar una construcción progresiva, modular y verificable.

Etapa 1: Implementación de Modelos (Semana 1)

Se crearán las clases **Fecha**, **Crédito**, **Canción**, **Álbum**, **Artista**, **Usuario**, **ListaFavoritos** y **Publicidad**, cada una con las siguientes características:

- Constructores paramétricos y de copia.
- Destructores que gestionen la liberación de memoria dinámica.
- Métodos **getters** y **setters** con validaciones internas.
- Sobrecarga de operadores: `==`, `<` y `=`.

Etapa 2: Estructuras de Datos Propias (Semana 1)

Se desarrollarán las estructuras de soporte para la gestión interna de información:

- **ArregloDinamico<T>**: plantilla genérica con capacidad de crecimiento dinámico.
- **Funciones amigas** para la comparación entre objetos sin necesidad de exponer sus datos privados.

Etapa 3: Persistencia (Semanas 1–2)

Implementación de la clase **GestorDatos**, encargada de la lectura y escritura de información persistente:

- Definir formato de archivos (`CSV` o delimitado manualmente).
- Leer los datos al iniciar el programa.
- Escribir los cambios realizados al finalizar la sesión.
- Validar la integridad de los datos almacenados.

Etapa 4: Reproducción (Semana 2)

Desarrollo de la clase **Reproductor**, responsable del manejo de la lógica de reproducción musical:

- Seleccionar canciones de manera aleatoria.
- Gestionar controles y validaciones (*siguiente, anterior, detener, repetir*).
- Mostrar las rutas de los archivos de audio y de las portadas de álbum.
- Administrar la publicidad en usuarios estándar.

Etapa 5: Orquestación y Menú (Semanas 2–3)

Creación de la clase **Plataforma**, la cual actuará como núcleo del sistema:

- Coordinar todos los componentes implementados.
- Gestionar el proceso de autenticación.
- Proporcionar menús según el tipo de usuario (*estándar o premium*).
- Ejecutar las funcionalidades principales del sistema.

Etapa 6: Medición de Recursos (Semana 3)

Implementación de los mecanismos de evaluación del rendimiento:

- Contador global de iteraciones.
- Acumulador de memoria asignada y liberada.
- Presentación de métricas al finalizar cada operación.

Etapa 7: Testing y Refinamiento (Semana 3)

Etapa final dedicada a la validación y mejora del sistema:

- Verificar la lógica completa de reproducción.
- Comprobar la unicidad de todos los datos.
- Probar casos límite y condiciones excepcionales.
- Optimizar el desempeño según los resultados obtenidos.

Formatos de Archivo

Los archivos de persistencia seguirán un **formato de texto estructurado**, en el cual cada campo estará separado por el carácter `|`. A continuación, se detallan los archivos utilizados por el sistema:

- **usuarios.txt:** `nickname | tipoMembresia | ciudad | país | año-mes-día`
- **artistas.txt:** `id | edad | país | seguidores | posicionTendencias`
- **albumes.txt:** `idArtista | idAlbum | nombre | géneros | año-mes-día | duracionTotal | sello | rutaPortada | puntuacion`
- **canciones.txt:** `idCancion | nombre | duracion | ruta128 | ruta320 | reproducciones`
- **creditos.txt:** `idCancion | nombre | apellido | codigoAfiliacion | tipo`
- **publicidad.txt:** `id | mensaje | categoría | ultVez`

- **favoritos.txt:** `nicknameUsuario | idCancion1 | idCancion2 | ...` *(hasta 10,000 canciones)*

Cada archivo será interpretado por la clase **GestorDatos**, que se encargará de:

- Leer los registros línea por línea al iniciar la aplicación.
- Parsear los campos mediante separadores `|`.
- Crear instancias dinámicas de los objetos correspondientes.
- Guardar los cambios realizados al finalizar la sesión.

Conclusión

UdeATunes se desarrollará como un sistema modular, escalable y eficiente, construido bajo los principios de la **Programación Orientada a Objetos pura**, haciendo uso de *sobrecarga*, *plantillas* y *funciones amigas* para maximizar la reutilización y cohesión del código.

La arquitectura propuesta garantiza una separación clara de responsabilidades mediante capas especializadas, lo que permite gestionar la complejidad de los datos sin comprometer el rendimiento.

Además, el enfoque sin dependencias externas favorece la **optimización de memoria**, el **control preciso de los recursos** y la **eficiencia en las iteraciones**, asegurando un sistema robusto, extensible y alineado con las buenas prácticas del desarrollo moderno en C++.