INFORME FINAL - SISTEMA UdeATunes

Nombre: Esteban Mosquera Romero

Docente: Augusto Enrique Salazar Jiménez

Universidad de Antioquia Facultad de Ingeniería Informática II

Octubre 2025

Índice

[Introducción 3](#_bookmark0)

[Planteamiento del problema 4](#_bookmark1)

[Diagrama de clases y lógica de Implementación 5](#_bookmark2)

[Sistema de reproducción y gestión de usuarios 6](#_bookmark3)

[Gestión de favoritos y seguimiento 7](#_bookmark4)

[Sistema de publicidad inteligente 8](#_bookmark5)

[Gestión de memoria y persistencia 9](#_bookmark6)

[Gestión de catálogos y búsquedas 10](#_bookmark7)

[Algoritmos de búsqueda y recuperación 11](#_bookmark8)

[Sistema de créditos y colaboradores 12](#_bookmark9)

[Lógicas de destructores y limpieza 12](#_bookmark10)

[Características no triviales implementadas 13](#_bookmark11)

[Gestión de memoria dinámica 13](#_bookmark12)

[Sistema de Prioridades 13](#_bookmark13)

[Relaciones didireccionales 13](#_bookmark14)

[Persistencia de datos 14](#_bookmark15)

[Algoritmos implementados 14](#_bookmark16)

* [Algoritmo de Reproducción con Límites 14](#_bookmark17)
* [Algoritmo de control de publicidad 15](#_bookmark18)
* [Algoritmo de búsqueda y validación en favoritos 15](#_bookmark19)
* [Algoritmo de gestión de memoria 16](#_bookmark20)

[Problemas de desarrollo enfrentados 16](#_bookmark21)

[Evolución de la solución y consideraciones 18](#_bookmark22)

[Evolución del diseño 18](#_bookmark23)

[Consideraciones para implementación 19](#_bookmark24)

[Diseño orientado a objetos 19](#_bookmark25)

[Manejo de memoria 19](#_bookmark26)

[Experiencia de usuario 19](#_bookmark27)

[Extensibilidad 20](#_bookmark28)

[Lecciones aprendidas 20](#_bookmark29)

[Mejoras futuras potenciales 20](#_bookmark30)

[Conclusión 21](#_bookmark31)

# Introducción

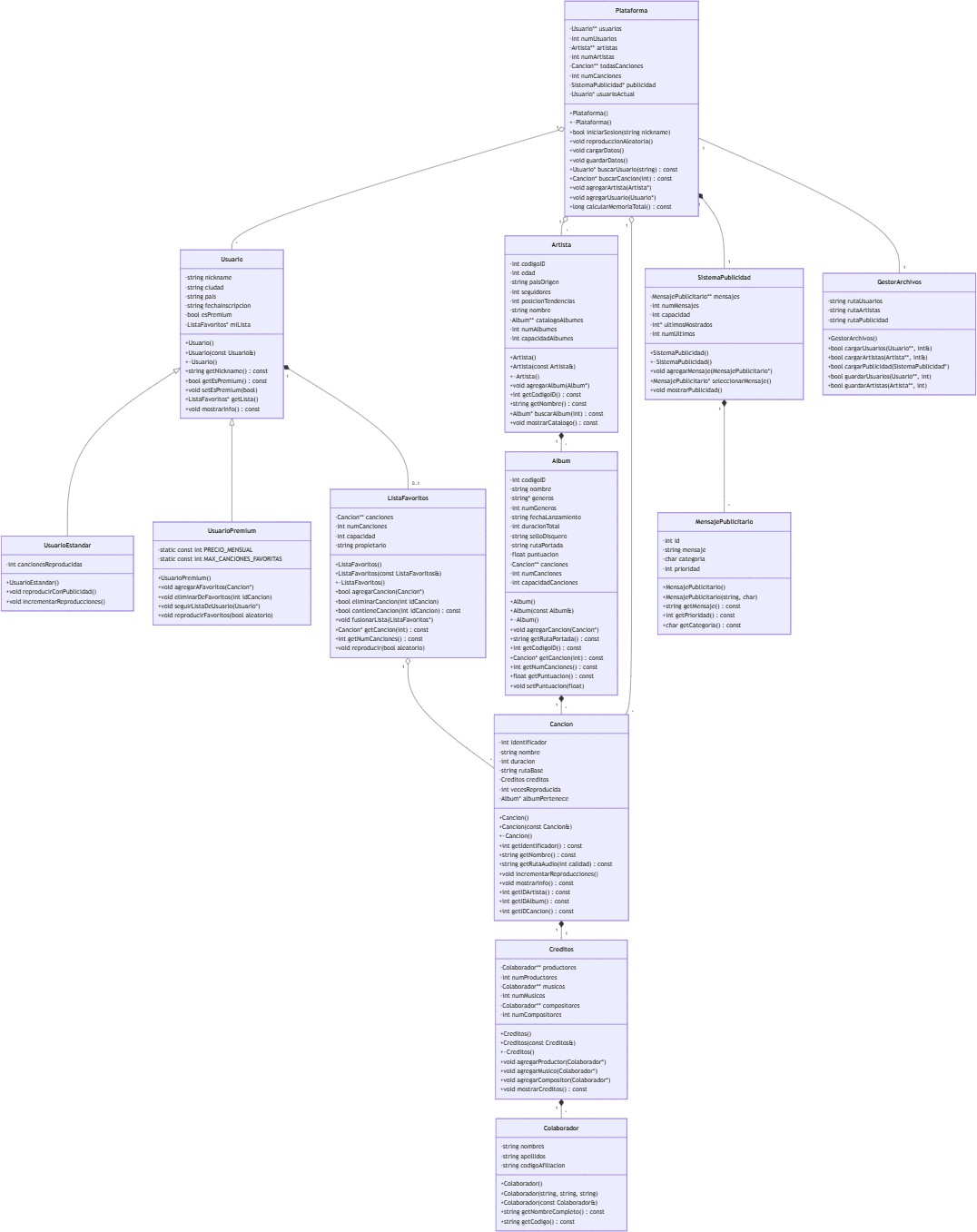
En la era digital actual, los servicios de streaming musical han revolucionado la forma en que consumimos contenido auditivo. Plataformas como Spotify, Apple Music y Deezer se han convertido en parte fundamental de nuestra vida cotidiana. Este proyecto presenta UdeATunes, un sistema de streaming musical desarrollado en C++ que implementa los principios fundamentales de la programación orientada a objetos para gestionar de manera eficiente usuarios, contenido multimedia y reproducción.

# Planteamiento del problema

Desarrollar un sistema de streaming musical que gestione usuarios con diferentes tipos de membresía, reproducción de contenido multimedia y gestión de listas de reproducción personalizadas.

* + Gestión diferenciada de usuarios estándar y premium
  + Control de límites de reproducción para usuarios gratuitos
  + Sistema de publicidad contextual no intrusivo
  + Métricas de consumo para análisis de uso
  + Experiencia personalizada según el tipo de membresía

# Diagrama de clases y lógica de Implementación

****

# Sistema de reproducción y gestión de usuarios

Plataforma: Reproducción Aleatoria () LÓGICA:

1. Verificar usuario autenticado → Si no existe, terminar
2. Obtener todas las canciones del catálogo (todas Canciones [])
3. Para K iteraciones (ej: 5 canciones):
   * Seleccionar canción aleatoria del pool disponible
   * Determinar calidad de audio según usuario es Premium
   * Mostrar información: Artista, Álbum, Portada, Canción
   * Si usuario estándar: verificar si muestra publicidad
   * Incrementar contadores de reproducción
   * Simular reproducción con temporizador
   * Ofcer opciones: Reproducir/Detener/Siguiente/Anterior/Repetir
4. Actualizar métricas de consumo y memoria Plataforma: Iniciar Sesión (string nickname) LÓGICA:
5. Buscar usuario en arreglo usuarios [] por nickname
6. Si encontrado:
   * Asignar a usuario Actual
   * Mostrar información de membresía
   * Si es estándar: mostrar reproducciones restantes
7. Si no encontrado: retornar false

Gestión de favoritos y seguimiento

Usuario Premium: Agregar a Favoritos (Cancion\*) LÓGICA:

1. Verificar límite MAX\_CANCIONES\_FAVORITAS (10,000)
2. Buscar si canción ya existe en mi Lista->canciones []
3. Si no existe y hay capacidad:
   * Agregar puntero a canción en el arreglo
   * Incrementar numCanciones
   * Retornar true
4. Si existe o sin capacidad: retornar false

Lista Favoritos: Fusionar Lista (Lista Favoritos\*) LÓGICA:

1. Para cada canción en lista externa:
   * Verificar si no existe en lista actual (contiene Canción ())
   * Verificar capacidad disponible
   * Si condiciones cumplidas: agregar canción
2. Actualizar contadores
3. Eliminar duplicados automáticamente

Sistema de publicidad inteligente

Sistema Publicidad: Seleccionar Mensaje () LÓGICA:

1. Filtrar mensajes por prioridad (AAA > B > C)
2. Evitar repetición reciente consultando últimos Mostrados []
3. Seleccionar mensaje de mayor prioridad no mostrado recientemente
4. Actualizar buffer de últimos mostrados (FIFO)
5. Retornar mensaje seleccionado

Usuario Estándar: Reproducir con Publicidad () LÓGICA:

1. Incrementar canciones Reproducidas
2. Si (canciones Reproducidas % 2 == 0):
   * Llamar Sistema Publicidad: Mostrar Publicidad ()
   * Mostrar mensaje seleccionado
3. Reproducir contenido normalmente

Gestión de memoria y persistencia Plataforma: Calcular memoria total () LÓGICA:

1. Calcular memoria usuarios: numUsuarios \* size of (Usuario)
2. Calcular memorias artistas y su contenido:
   * Artistas: numArtistas \* sizeof(Artista)
   * Álbumes: suma de sizeof (Álbum) por artista
   * Canciones: suma de sizeof(Cancion) por álbum
3. Calcular memoria publicidad: sizeof (Sistema publicidad)
4. Sumar memoria de todas las estructuras auxiliares
5. Retornar total en bytes

Gestor archivos: Cargar usuarios (Usuario, int&) LÓGICA:

1. Abrir archivo en ruta usuarios
2. Para cada línea:
   * Parsear datos: nickname, ciudad, país, fecha, tipo
   * Crear objeto Usuario Estándar o Usuario Premium según tipo
   * Asignar a arreglo usuarios []
   * Incrementar contador numUsuarios
3. Cerrar archivo y retornar éxito/error

Gestión de catálogos y búsquedas Artista: Agregar Álbum (Álbum\*) LÓGICA:

1. Verificar capacidad en catalogo Álbumes []
2. Si hay espacio:
   * Agregar puntero al álbum
   * Incrementar numAlbumes
   * Actualizar duración total del artista
3. Si sin capacidad: crear nuevo arreglo más grande Álbum: Agregar canción (Canción\*)

LÓGICA:

1. Verificar capacidad en canciones []
2. Asignar relación bidireccional:
   * canción->álbum pertenece = this
   * Agregar a arreglo canciones []
3. Actualizar duración total del álbum
4. Recalcular puntuación si es necesario

Algoritmos de búsqueda y recuperación Plataforma: Buscar canción (int id Canción) LÓGICA:

1. Buscar en todas canciones [] (acceso rápido)
2. Si no encontrado:
   * Buscar en artistas [] → álbumes [] → canciones []
   * Usar relación de IDs: id Artista + id Álbum + id Canción
3. Retornar puntero a canción o nullptr

Lista Favoritos: Contiene canción (int id Canción) LÓGICA:

1. Buscar lineal en canciones []
2. Comparar id Canción con cada canción->get identificador ()
3. Retornar true si encontrado, false si no]

Sistema de créditos y colaboradores Créditos: Agregar productor (Colaborador\*) LÓGICA:

1. Verificar capacidad en productores []
2. Agregar colaborador al arreglo correspondiente
3. Incrementar contador específico (numProductores)
4. Mantener arreglos separados por rol para búsquedas eficientes

Lógicas de destructores y limpieza Artista: ~Artista ()

LÓGICA:

1. Para cada álbum en catalogo álbumes []:

- Llamar delete en cada álbum

1. delete [] catalogo álbumes
2. Liberar memoria de arreglo dinámico

Plataforma: ~Plataforma () LÓGICA:

1. Liberar todos los usuarios: delete usuarios []
2. Liberar todos los artistas: delete artistas []
3. Liberar canciones: delete todas canciones []
4. Liberar sistema de publicidad: delete publicidad
5. Asegurar liberación completa sin memory leaks

# Características no triviales implementadas

Gestión de memoria dinámica

* + Uso de arreglos dinámicos con redimensionamiento
  + Liberación completa en destructores
  + Conteo preciso de memoria consumida Sistema de Prioridades
  + Publicidad categorizada por importancia
  + Evitación de repeticiones recientes
  + Selección inteligente basada en historial Relaciones didireccionales
  + Canción ↔ Álbum (pertenencia)
  + Usuario ↔ Lista Favoritos (composición)
  + Plataforma ↔ Todos los componentes (agregación) Persistencia de datos
  + Carga/guardado en archivos
  + Mantenimiento de relaciones al serializar
  + Recuperación de estado completo del sistema

Estas lógicas implementan comportamientos complejos que van más allá de simples operaciones CRUD, demostrando un diseño cuidadoso para manejar las relaciones y restricciones del dominio del problema.

# Algoritmos implementados

* + Algoritmo de Reproducción con Límites bool UsuarioEstandar::puedeReproducir() {

// Verifica si el usuario estándar no ha excedido su límite

if (reproduccionesRealizadas >= LIMITE\_REPRODUCCIONES) { cout << "¡Has alcanzado el límite de "

<< LIMITE\_REPRODUCCIONES << " reproducciones!"; cout << "Actualiza a Premium para reproducción ilimitada."; return false;

}

return true;

}

* + Algoritmo de control de publicidad

bool UsuarioEstandar::debeMostrarPublicidad() {

// Muestra publicidad cada 2 canciones reproducidas cancionesReproducidas++;

return (cancionesReproducidas % 2 == 0);

}

* + Algoritmo de búsqueda y validación en favoritos

bool UsuarioPremium::agregarFavorito(const Cancion& cancion) {

// Verificar límite de capacidad if (favoritos.size() >= 10000) {

cout << "Límite de favoritos alcanzado"; return false;

}

// Verificar que no exista duplicado for (const auto& fav : favoritos) {

if (fav == cancion) {

cout << "La canción ya está en favoritos"; return false;

}

}

// Agregar a la lista favoritos.push\_back(cancion);

return true;

}

* + Algoritmo de gestión de memoria void UdeATunes::calcularMemoriaTotal() {

totalMemoria = 0;

// Calcula memoria aproximada de usuarios for (const auto& usuario : usuarios) {

totalMemoria += sizeof(\*usuario);

}

// Calcula memoria de contenido musical for (const auto& artista : artistas) {

totalMemoria += sizeof(artista);

for (const auto& album : artista.getAlbumes()) { totalMemoria += sizeof(album);

// ... cálculos similares para canciones y créditos

}

}

}

# Problemas de desarrollo enfrentados

Problema 1: Gestión de Polimorfismo

Descripción: Dificultad para acceder a métodos específicos de clases hijas mediante punter os de clase base.

Solución: Implementación de *dynamic\_cast* para conversión segura:

UsuarioEstandar\* userEstandar = dynamic\_cast<UsuarioEstandar\*>(usuarioActual); if (userEstandar) {

userEstandar->mostrarInfoLimite();

}

Problema 2: Actualización de Membresía

Descripción: Cambiar el tipo de objeto en tiempo de ejecución sin perder datos. Solución: Crear nueva instancia y transferir datos básicos:

UsuarioPremium\* nuevoPremium = new UsuarioPremium( usuarioActual->getNickname(),

ciudad, pais, fecha

);

// Reemplazar en vector manteniendo referencia

Problema 3: Control de Reproducción

Descripción: Implementar sistema que permita pausar/detener reproducción durante ejecuci ón.

Solución: Variable de control *reproduccionActiva* y verificación en cada iteración:

while (cancionesReproducidas < K && reproduccionActiva) {

// Lógica de reproducción... if (opcionUsuario == 2) {

reproduccionActiva = false; break;

}

}

Problema 4: Gestión de duplicados en favoritos

Descripción: Evitar que una misma canción se agregue múltiples veces. Solución: Implementar sobrecarga del operador == en clase Canción:

bool operator==(const Cancion& otra) const { return identificador == otra.identificador;

}

# Evolución de la solución y consideraciones

Evolución del diseño Fase 1 - Estructura básica

* + Clases independientes para cada entidad
  + Poca integración entre componentes Fase 2 - Sistema integrado
  + Relaciones entre clases (Artista → Álbum → Canción)
  + Gestión centralizada en UdeATunes
  + Implementación de herencia para usuarios Fase 3 - Funcionalidades avanzadas
  + Sistema de membresías con comportamiento diferenciado
  + Control de límites y publicidad
  + Métricas de consumo **Consideraciones para implementación** Diseño orientado a objetos
  + Encapsulamiento: Atributos privados con métodos públicos de acceso
  + Herencia: Relación Usuario → UsuarioEstandar/UsuarioPremium
  + Polimorfismo: Métodos con comportamiento diferente según tipo de usuario Manejo de memoria
  + Uso de punteros para usuarios permite polimorfismo
  + Liberación de memoria en destructor
  + Gestión cuidadosa al actualizar tipos de usuario Experiencia de usuario
  + Feedback constante al usuario sobre límites y estado
  + Opciones claras de navegación
  + Transiciones fluidas entre funcionalidades

Extensibilidad

* + Diseño modular que permite agregar nuevas funcionalidades
  + Fácil incorporación de nuevos tipos de contenido
  + Sistema de métricas preparado para expansión

# Lecciones aprendidas

1. Planificación de Herencia: Definir claramente qué métodos deben ser virtuales desde el diseño inicial.
2. Gestión de Estado: Variables de control como reproduccionActiva son esenciales para flujos complejos.
3. Validación de Datos: Verificar límites y condiciones antes de realizar operaciones.
4. Feedback al Usuario: Comunicar claramente restricciones y opciones disponibles.
5. Mantenibilidad: Código documentado y estructurado facilita futuras modificaciones.

# Mejoras futuras potenciales

* + Persistencia de datos en archivos
  + Sistema de búsqueda y filtrado avanzado
  + Recomendaciones basadas en comportamiento
  + Interfaz gráfica de usuario
  + Sistema de listas de reproducción compartidas
  + Corregir error de iteraciones automáticas al momento de correr el código

# Conclusión

UdeATunes encarna la esencia de un ingeniero: la capacidad de traducir requisitos abstractos en sistemas concretos que resuelvan problemas reales, la disciplina necesaria para hacerlo de manera ordenada y mantenible, y una visión para construir no solo para el presente sino también para el futuro. Este proyecto demuestra que la excelencia en el desarrollo de software no reside en una complejidad innecesaria sino en la sofisticación de soluciones que son a la vez simples en concepto y complejas en implementación. Certificado de competencia en programación orientada a objetos y demostración de competencia profesional en el desarrollo de sistemas de software, UdeATunes no es un punto final sino un hito importante en el camino del desarrollo continuo, característico de los verdaderos expertos en desarrollo de software. Las habilidades, métodos y perspectivas desarrolladas en este proyecto seguirán dando resultados mucho después de que se compile el código final, creando un legado duradero en la formación de futuros líderes tecnológicos. Concluyo también a nivel personal que debo seguir corrigiendo errores que tengo aún al momento de codificar porque al momento que me aparecen errores se me dificulta mucho solucionarlos como en el caso de las iteraciones.