

Informe de Desarrollo - Sistema UdeATunes

Juan José Rendón González

C.C 1000415194

jose.rendon1@udea.edu.co

Curso: Informática II - Semestre 2025-2

1. Introducción y Contexto del Problema

UdeATunes representa el desafío de modelar un sistema de streaming musical completo en C++, emulando las funcionalidades básicas de plataformas comerciales como Spotify o Apple Music. El problema requiere gestionar múltiples entidades interrelacionadas: usuarios con diferentes niveles de membresía, artistas, álbumes, canciones, listas de reproducción y un sistema de publicidad inteligente. La complejidad radica en mantener la eficiencia mientras se manejan relaciones jerárquicas y se respetan restricciones específicas como la prohibición del uso de la STL y la implementación manual de todas las estructuras de datos.

El sistema debe diferenciar claramente entre usuarios estándar y premium. Los usuarios premium, que pagan una membresía de \$19.900 mensuales, disfrutan de beneficios exclusivos como listas de favoritos personalizadas hasta 10.000 canciones, capacidad de seguir listas de otros usuarios, reproducción en alta calidad (320 kbps) y ausencia de publicidad. Por contraste, los usuarios estándar reciben publicidad cada dos canciones reproducidas, con un sistema probabilístico que prioriza mensajes según su categoría (AAA, B, C).

2. Análisis del Problema y Consideraciones de Diseño

El análisis del problema reveló varios desafíos técnicos significativos. Primero, la gestión eficiente de un catálogo musical que puede escalar a miles de canciones, manteniendo las relaciones entre artistas, álbumes y canciones. Segundo, la implementación de un sistema de reproducción con estados complejos que varían según el tipo de usuario. Tercero, la creación de un sistema de listas de favoritos que permita tanto la gestión personal como el seguimiento de listas ajenas. Cuarto, la implementación de un algoritmo de publicidad que evite repeticiones consecutivas y respete las prioridades por categoría.

Las consideraciones de diseño se orientaron hacia la eficiencia y mantenibilidad. Se optó por un diseño modular con gestores especializados para diferentes dominios: GestorUsuarios para la autenticación y gestión de usuarios, GestorCatalogo para el catálogo musical, y un sistema de Reproductor independiente para la lógica de reproducción. Esta separación de responsabilidades permite un código más mantenible y testeable.

Una decisión fundamental fue el uso de un diseño entre structs y clases. Los structs se emplean para entidades simples que actúan principalmente como contenedores de datos:

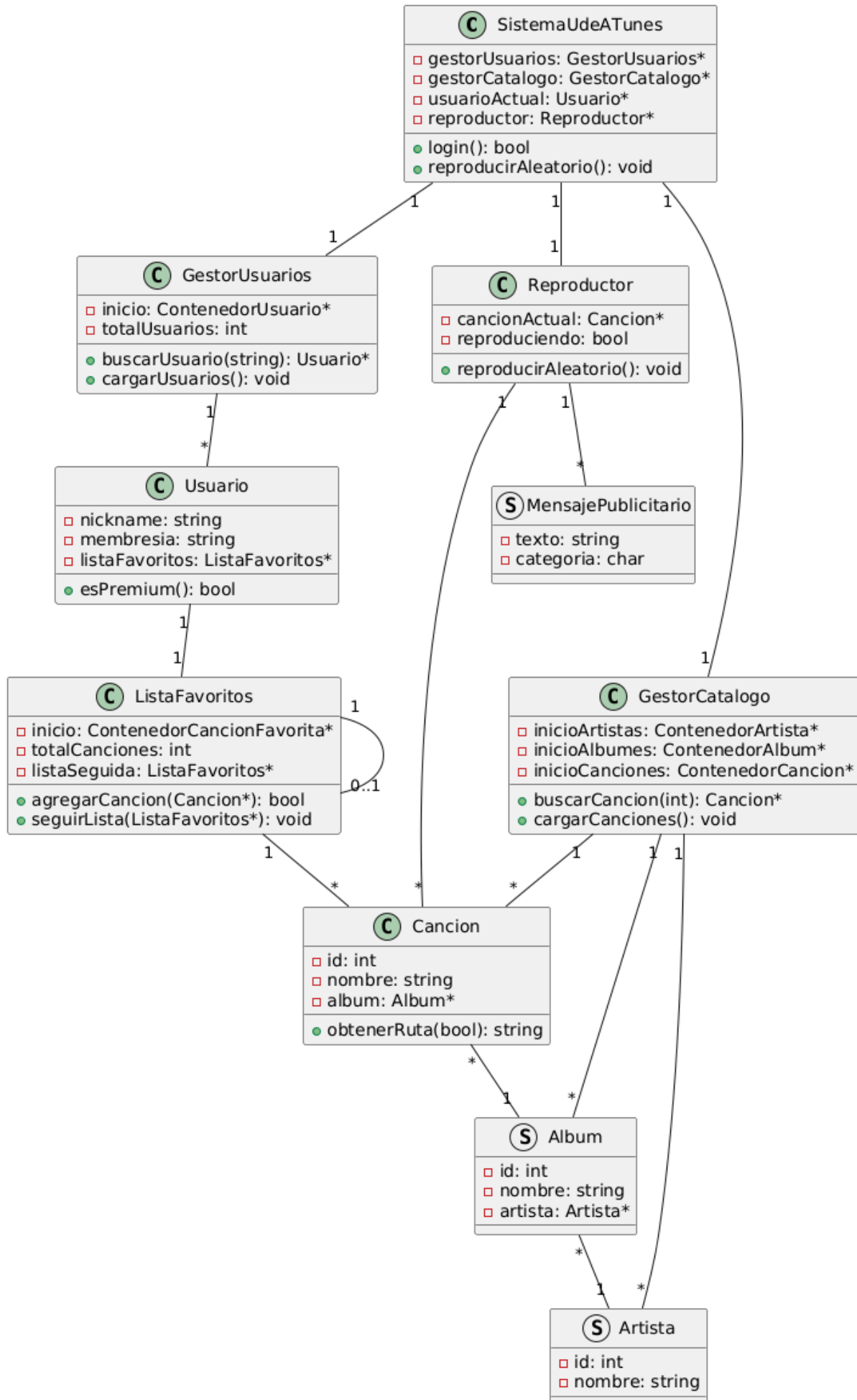
Artista, Album y MensajePublicitario. Las clases se reservan para entidades con comportamiento complejo: Usuario, Cancion, ListaFavoritos y los gestores.

3. Arquitectura del Sistema y Diagrama de Clases

La arquitectura del sistema se organiza alrededor de la clase SistemaUdeATunes, que actúa como principal punto de entrada de la aplicación. Esta clase contiene y coordina tres componentes principales: el GestorUsuarios, responsable de la autenticación y gestión de usuarios; el GestorCatalogo, que maneja todo el catálogo musical; y el Reproductor, que implementa la lógica de reproducción con sus diferentes modos.

El GestorUsuarios gestiona una lista enlazada de objetos Usuario. Cada Usuario puede contener una ListaFavoritos (exclusiva para usuarios premium), que a su vez gestiona una lista enlazada de canciones favoritas. El GestorCatalogo mantiene tres listas enlazadas independientes para Artistas, Albums y Canciones, preservando las relaciones a través de referencias entre ellas.

Las relaciones clave del sistema son: cada Canción pertenece exactamente a un Album, y cada Album pertenece exactamente a un Artista. Esta jerarquía se refleja en el formato de ID de las canciones, que utiliza 9 dígitos: 5 para el artista, 2 para el álbum y 2 para la canción específica. El Reproductor accede tanto al catálogo de canciones como a los mensajes publicitarios, adaptando su comportamiento según el tipo de usuario autenticado.



4. Lógica de Implementación de Funcionalidades

4.1 Sistema de Reproducción con Gestión de Estados Complejos

El sistema de reproducción implementa una máquina de estados que gestiona transiciones entre reproducción, pausa, modo repetir y navegación por historial. Para usuarios premium, se mantiene un historial de las últimas 4 canciones reproducidas, permitiendo la navegación hacia atrás. La implementación del modo repetir requiere un control cuidadoso del flujo de ejecución para evitar bucles infinitos mientras mantiene la capacidad de interrumpir la reproducción.

La reproducción aleatoria incorpora un temporizador de 3 segundos por canción para propósitos de prueba, limitándose a 5 canciones por sesión de prueba. Este mecanismo permite probar la funcionalidad sin requerir interacción prolongada. La gestión del estado de reproducción incluye validaciones para prevenir operaciones inválidas, como intentar detener una reproducción que no está en curso o navegar a una canción anterior cuando el historial está vacío.

4.2 Algoritmo de Publicidad con Probabilidades

El sistema de publicidad implementa un algoritmo de selección que considera las categorías de mensajes. Los mensajes categoría AAA tienen triple prioridad, categoría B doble prioridad, y categoría C prioridad simple. El algoritmo calcula primero un peso total sumando las prioridades de todos los mensajes disponibles, luego genera un número aleatorio dentro de este rango y recorre la lista acumulando pesos hasta encontrar el mensaje correspondiente.

Para evitar repeticiones consecutivas, el algoritmo incorpora un mecanismo de reintento que selecciona un mensaje diferente si coincide con el último mostrado. Este mecanismo incluye un límite de intentos para prevenir bucles infinitos en casos límites. La publicidad se muestra cada dos canciones para usuarios estándar, con un contador que lleva la cuenta de las reproducciones y muestra la publicidad en los intervalos apropiados.

4.3 Sistema de Listas de Favoritos con Capacidad de Seguimiento

Las listas de favoritos implementan una funcionalidad única que permite a los usuarios premium seguir las listas de otros usuarios. Cuando un usuario sigue una lista ajena, el sistema combina las canciones propias con las de la lista seguida, presentándose como una lista unificada. Esta combinación se realiza dinámicamente cada vez que se accede a la lista, garantizando que los cambios en la lista seguida se reflejan inmediatamente.

La implementación incluye protecciones (un usuario no puede seguirse a sí mismo) y validación de la existencia y elegibilidad de los usuarios a seguir. Las listas combinadas pueden reproducirse en orden original o aleatorio. El sistema mantiene la distinción entre canciones propias y seguidas en la visualización, proporcionando transparencia sobre el origen de cada canción.

5. Algoritmos y Estructuras de Datos Implementados

5.1 Estructuras de Datos

Todas las estructuras de datos se implementaron manualmente utilizando listas enlazadas simples. Cada gestor mantiene su propia lista enlazada de elementos, con nodos que contienen punteros al elemento y al siguiente nodo. Esta implementación permite inserciones y eliminaciones eficientes mientras mantiene un uso de memoria predecible.

Para las operaciones de recorrido y búsqueda, se implementaron iteradores implícitos mediante punteros a nodos actuales. Aunque menos convenientes que los iteradores de la STL, esta aproximación proporciona el control necesario sobre el rendimiento y el uso de memoria. Las búsquedas se implementan como recorridos lineales, aceptables dado los tamaños esperados de las listas en el contexto del problema.

5.2 Algoritmo de Gestión de Memoria

El sistema implementa un esquema de gestión de memoria con destructores que liberan recursivamente toda la memoria asignada. Cada clase maneja sus recursos, garantizando que se liberen adecuadamente cuando los objetos salen de ámbito. Los gestores principales incluyen métodos 'limpiar' que liberan todas sus estructuras internas, utilizados durante la destrucción y la recarga de datos.

El cálculo del consumo de memoria considera no solo el tamaño de los objetos principales, sino también las cadenas de texto y las estructuras de nodos de las listas enlazadas. Esta aproximación proporciona una estimación que se espera sea realista sobre el uso de memoria, esencial para las métricas de eficiencia requeridas.

5.3 Algoritmo de Validación de Integridad de Datos

Durante la carga de datos, el sistema implementa un algoritmo de validación que verifica la consistencia de las relaciones jerárquicas. Para cada canción, se valida que el artista y álbum referenciados por su ID compuesto existan realmente en el catálogo. Esta validación previene referencias huérfanas y garantiza la integridad de los datos.

El algoritmo de validación también verifica el formato de las rutas de archivo, asegurando que sigan la convención Linux con rutas absolutas. Para las canciones, se valida la existencia de ambas versiones (128 kbps y 320 kbps) en las ubicaciones especificadas, aunque el sistema no verifica físicamente la existencia de los archivos.

6. Problemas de Desarrollo y Soluciones Implementadas

6.1 Gestión Manual de Memoria sin STL

El requisito de evitar la STL representó uno de los desafíos más significativos. La solución involucró la implementación manual de todas las estructuras de datos, principalmente listas enlazadas para cada colección de entidades. Este enfoque requirió una atención cuidadosa a la gestión de memoria, particularmente en los destructores y en las operaciones que involucran múltiples estructuras interrelacionadas.

Se implementó un sistema de conteo de referencias implícito mediante la cuidadosa gestión de la propiedad de los objetos. Los gestores son propietarios de sus entidades y responsables de su liberación, mientras que otras partes del sistema utilizan referencias.

6.2 Coordinación entre Componentes del Sistema

La coordinación entre los diferentes gestores y componentes presentó desafíos de diseño. La solución implementada que a través de la clase `SistemaUdeATunes`, se proporciona una interfaz unificada para operaciones que involucran múltiples componentes. Por ejemplo, la reproducción de una lista de favoritos requiere coordinación entre el `GestorUsuarios` (para verificar permisos), el `GestorCatalogo` (para acceder a las canciones), y el `Reproductor` (para ejecutar la reproducción).

Este enfoque centralizado simplifica la lógica del programa y localiza la complejidad de la coordinación en una única clase. Además, facilita el testing y la evolución del sistema, ya que los cambios en las interacciones entre componentes se concentran en un lugar específico.

6.3 Sistema de Métricas de Eficiencia

La implementación del sistema de métricas requirió un enfoque intrusivo pero eficiente. Cada clase que realiza operaciones significativas incluye un contador de iteraciones que se incrementa en puntos estratégicos del código. Este enfoque proporciona una medida razonable de la complejidad computacional..

El cálculo de memoria que se consume considerando tanto el tamaño de los objetos como la memoria dinámica asociada (especialmente cadenas y arrays temporales). La implementación recorre todas las estructuras activas, sumando los tamaños de los objetos y sus componentes. Aunque es una aproximación, proporciona una estimación útil para comparar el consumo entre diferentes operaciones.

7. Evolución de la Solución

El desarrollo del sistema siguió una evolución incremental a través de cuatro fases claramente definidas. La primera fase, correspondiente al 17 de octubre, se concentró en el análisis y diseño, resultando en la definición de la arquitectura general y los formatos de archivo. Esta fase incluyó la creación del diagrama de clases y la especificación detallada de las responsabilidades de cada componente.

La segunda fase implementó el núcleo del sistema: las clases básicas de entidades y los gestores principales. Durante esta fase, se establecieron los patrones fundamentales de gestión de memoria tomando una ruta de redimensionamiento que después se abandonara

por su ineficiencia y se implementó el sistema de carga de datos. La tercera fase añadió las funcionalidades avanzadas: sistema de reproducción completo, gestión de listas de favoritos con seguimiento, y sistema de publicidad.

La fase final se dedicó a las métricas de eficiencia, optimización y pruebas exhaustivas. Esta fase incluyó la instrumentación del código para el conteo de iteraciones, la implementación del cálculo de consumo de memoria, y la refinación de la interfaz de usuario.

Las lecciones más significativas aprendidas durante el desarrollo incluyen: la importancia de establecer patrones consistentes de gestión de memoria desde el inicio, la utilidad de la separación clara de responsabilidades en sistemas complejos, y la efectividad del desarrollo incremental para gestionar la complejidad. Particularmente valiosa resultó la decisión de implementar primero la funcionalidad principal y luego añadir características avanzadas, permitiendo detectar y corregir problemas arquitectónicos temprano en el proceso..

8. Conclusión

El sistema cumple con los requisitos especificados: gestión de usuarios con diferentes niveles de membresía, catálogo musical jerárquico, sistema de reproducción con modos variados, listas de favoritos con capacidad de seguimiento, sistema de publicidad, y métricas de eficiencia. Particularmente se nota el cumplimiento de las restricciones: la implementación manual de todas las estructuras de datos sin uso de la STL, y la gestión cuidadosa de recursos.


El código resultante es no solo funcional sino también mantenible y extensible, con una separación clara y consistente a través de todo el programa.


Anexo clases:


A continuación se hará un anexo al diagrama de clases simplificado que se mostró anteriormente, se mostrará cada clase en su totalidad y se hace en este apartado más que todo por una cuestión de limpieza en el documento.


SistemaUdeATunes

- gestorUsuarios: GestorUsuarios*
 - gestorCatalogo: GestorCatalogo*
 - inicioMensajes: ContenedorMensaje*
 - totalMensajes: int
 - usuarioActual: Usuario*
 - reproductor: Reproductor*
 - totalIteraciones: unsigned long
 - memoriaConsumida: unsigned long
-
- SistemaUdeATunes()
 - ~SistemaUdeATunes()
 - cargarDatos(): void
 - login(): bool
 - reproducirAleatorio(): void
 - reproducirListaFavoritos(ordenAleatorio: bool): void
 - mostrarMetricas(): void
 - buscarUsuario(nickname: string): Usuario*
 - buscarCancion(id: int): Cancion*
 - getUsuarioActual(): Usuario*
 - getTotalUsuarios(): int
 - getTotalCanciones(): int
 - getTotalArtistas(): int
 - getTotalAlbumes(): int
 - getTotalMensajes(): int
 - getCanciones(): Cancion**
 - getMensajesArray(): MensajePublicitario**
 - setUsuarioActual(usuario: Usuario*): void
 - mostrarCancionesDisponibles(): void
 - agregarCancionAFavoritos(idCancion: int): bool
 - seguirListaUsuario(nicknameSeguido: string): bool
 - dejarDeSeguirLista(): void
 - mostrarMetricasEficiencia(): void
 - incrementarIteraciones(cantidad: int): void
 - resetIteraciones(): void
 - getTotalIteraciones(): unsigned long
 - calcularMemoria(): void
 - calcularMemoriaMensajes(): unsigned long
 - calcularMemoriaTodasListasFavoritos(): unsigned long
 - cargarMensajes(): void
 - guardarCambios(): void
 - limpiarMensajes(): void

 Cancion
<ul style="list-style-type: none"> □ id: int □ nombre: string □ duracion: float □ ruta128: string □ ruta320: string □ reproducciones: int □ album: Album*
<ul style="list-style-type: none"> ● Cancion() ● Cancion(id: int, nombre: string, album: Album*) ● ~Cancion() ● operator==(otra: Cancion): bool ● obtenerRuta(altaCalidad: bool): string ● incrementarReproducciones(): void ● getId(): int ● getNombre(): string ● getDuracion(): float ● getReproducciones(): int ● getAlbum(): Album* ● setDuracion(duracion: float): void ● setRuta128(ruta: string): void ● setRuta320(ruta: string): void

 GestorUsuarios
<ul style="list-style-type: none"> □ inicio: ContenedorUsuario* □ final: ContenedorUsuario* □ totalUsuarios: int □ iteraciones: unsigned long
<ul style="list-style-type: none"> ● GestorUsuarios() ● ~GestorUsuarios() ● buscarUsuario(nickname: string): Usuario* ● agregarUsuario(usuario: Usuario*): bool ● cargarUsuarios(): void ● limpiarUsuarios(): void ● getTotalUsuarios(): int ● getUsersArray(): Usuario** ● getIteraciones(): unsigned long ● incrementarIteraciones(cantidad: int): void ● resetIteraciones(): void ● calcularMemoriaUsuarios(): unsigned long

 Usuario
<ul style="list-style-type: none"> ❑ nickname: string ❑ membresia: string ❑ listaFavoritos: ListaFavoritos*
<ul style="list-style-type: none"> ● Usuario() ● Usuario(nickname: string, membresia: string) ● ~Usuario() ● esPremium(): bool ● getNickname(): string ● getMembresia(): string ● getListaFavoritos(): ListaFavoritos*

 GestorCatalogo
<ul style="list-style-type: none"> ❑ inicioArtistas: ContenedorArtista* ❑ inicioAlbumes: ContenedorAlbum* ❑ inicioCanciones: ContenedorCancion* ❑ totalArtistas: int ❑ totalAlbumes: int ❑ totalCanciones: int ❑ iteraciones: unsigned long
<ul style="list-style-type: none"> ● GestorCatalogo() ● ~GestorCatalogo() ● buscarArtista(id: int): Artista* ● buscarAlbum(id: int): Album* ● buscarCancion(id: int): Cancion* ● cargarArtistas(): void ● cargarAlbumes(): void ● cargarCanciones(): void ● guardarCanciones(): void ● getTotalArtistas(): int ● getTotalAlbumes(): int ● getTotalCanciones(): int ● getCancionesArray(): Cancion** ● getIteraciones(): unsigned long ● incrementarIteraciones(cantidad: int): void ● resetIteraciones(): void ● calcularMemoriaArtistas(): unsigned long ● calcularMemoriaAlbumes(): unsigned long ● calcularMemoriaCanciones(): unsigned long ● limpiarCatalogo(): void ■ limpiarArtistas(): void ■ limpiarAlbumes(): void ■ limpiarCanciones(): void ■ agregarArtista(artista: Artista*): bool ■ agregarAlbum(album: Album*): bool ■ agregarCancion(cancion: Cancion*): bool

C ListaFavoritos

- inicio: ContenedorCancionFavorita*
- totalCanciones: int
- usuario: Usuario*
- listaSeguida: ListaFavoritos*
- iteraciones: unsigned long

- ListaFavoritos(usuario: Usuario*)
- ~ListaFavoritos()
- agregarCancion(cancion: Cancion*): bool
- eliminarCancion(idCancion: int): bool
- contieneCancion(idCancion: int): bool
- getCancionesConSeguidas(ordenAleatorio: bool): Cancion**
- getTotalCancionesVisibles(): int
- getTotalCancionesSeguidas(): int
- seguirLista(otraLista: ListaFavoritos*): void
- dejarDeSeguirLista(): void
- mostrarLista(): void
- getTotalCanciones(): int
- getUsuario(): Usuario*
- getCancionesArray(): Cancion**
- estaSiguiendoLista(): bool
- getIteraciones(): unsigned long
- resetIteraciones(): void
- incrementarIteraciones(cantidad: int): void
- calcularMemoria(): unsigned long

C Reproductor

- TAMANO_HISTORIAL: int
- historial: Cancion*[]
- cantidadHistorial: int
- indiceUltima: int
- todasLasCanciones: Cancion**
- todosLosMensajes: MensajePublicitario**
- totalCanciones: int
- totalMensajes: int
- usuarioActual: Usuario*
- cancionActual: Cancion*
- indiceActual: int
- reproduciendo: bool
- modoRepetir: bool
- contadorCancionesReproducidas: int
- reproduccionListaFavoritos: bool
- generadorAleatorio: mt19937
- iteraciones: unsigned long

- Reproductor(canciones: (Cancion*)*, totalCanc: int, mensajes: MensajePublicitario**, totalMsg: int, usuario: Usuario*)
- actualizarDatos(canciones: (Cancion*)*, totalCanc: int, mensajes: MensajePublicitario**, totalMsg: int, usuario: Usuario*): void
- reproducirAleatorio(): void
- reproducirListaFavoritos(cancionesLista: Cancion**, totalCancionesLista: int, ordenAleatorio: bool): void
- siguienteCancion(): void
- cancionAnterior(): void
- detenerReproduccion(): void
- toggleRepetir(): void
- getIteraciones(): unsigned long
- resetIteraciones(): void
- incrementarIteraciones(cantidad: int): void
- estaReproduciendo(): bool
- getCancionActual(): Cancion*
- mostrarInterfazReproduccion(): void
- obtenerMensajeAleatorio(): MensajePublicitario*
- generarNumeroAleatorio(maximo: int): int
- agregarAlHistorial(cancion: Cancion*): void
- obtenerCancionAnterior(): Cancion*
- retrocederEnHistorial(): bool