Desafío II

Informática II

Semestre: 2025-2

Integrantes del equipo

Ana Sofia Agudelo Blandon  
Keiner Marcelo Torres Villeros

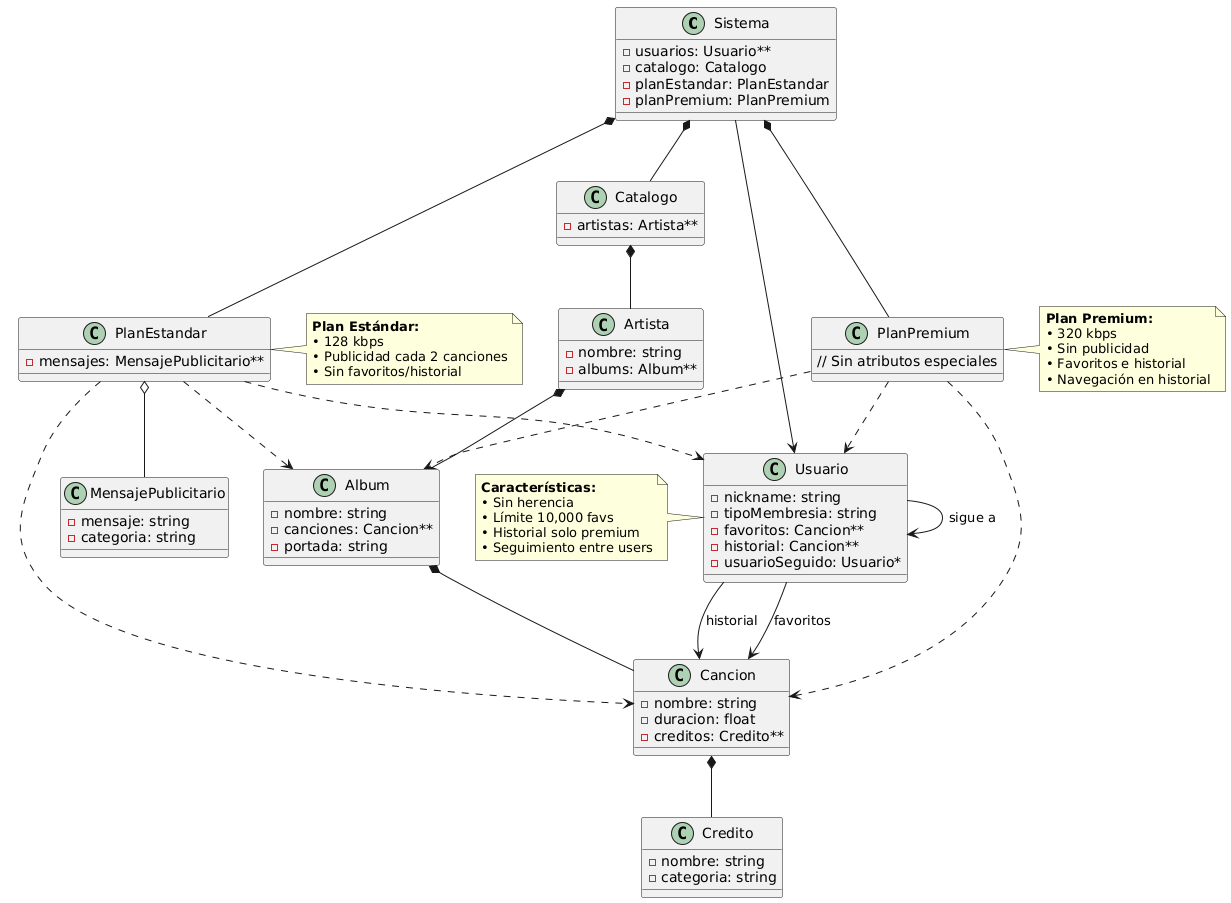
Fecha de entrega

Primera entrega: 17 de octubre

1. Análisis del problema y consideraciones para la alternativa de solución propuesta.

R// La idea detrás del desafío es crear un sistema de streaming musical llamado UdeATunes, que imite plataformas comerciales como Spotify o Apple Music. El reto consiste en modelar entidades del mundo real, como usuarios, artistas, álbumes y canciones, y entender sus complejas interrelaciones, además de incluir funciones específicas para la reproducción, gestión de listas y publicidad. La solución que proponemos busca equilibrar la complejidad funcional con la eficiencia técnica, aplicando principios sólidos de programación orientada a objetos, todo mientras se cumplen las restricciones establecidas (sin usar STL y con manejo manual de memoria). El diseño fue concebido, con fácil mantenimiento y expansión en memoria, está preparado para amoldarse a futuras actualizaciones del sistema.

1. Diagrama de clases de la solución planteada. Adicionalmente, describa en alto nivel la lógica de las tareas que usted definió para aquellos subprogramas cuya solución no sea trivial.

R// 

**Herencia Usuario → UsuarioEstandar UsuarioPremium:**  
La herencia se usa, porque los dos tipos de usuario comparten atributos y comportamiento base (login, datos personales), aunque, tienen comportamientos distintos como la publicidad, favoritos y navegación previa que se modelan en las subclases.  
**Composición Artista → Album, Album → Cancion, UsuarioPremium → ListaFavoritos:** Estas relaciones, indican la dependencia de existencia, pues un Álbum, no tiene sentido fuera del Artista que lo creo, además, una Canción pertenece solamente a un Álbum, también, la ListaFavoritos pertenece al UsuarioPremium y se destruye si el usuario se elimina.  
**Asociaciones Cancion ↔ Credito, UsuarioPremium ↔ UsuarioPremium:**Los créditos de la producción, músicos y compositores, se asocian a una canción; un usuario premium puede seguir la lista de otro, eso es una asociación con cardinalidad.  
**Dependencia Plataforma → Cancion, Plataforma → MensajePublicidad:** Plataforma utiliza canciones y mensajes para ofrecer funcionalidades, como reproducción aleatoria o para mostrar anuncios, pero no es necesario que Plataforma “cree” las canciones pues las administra, así es como modelamos la dependencia/uso.  
**Estructuras de datos arreglos dinámicos:** El enunciado prohíbe STL: por eso en el UML se muestra Tipo\*\* + contador (cantX) para indicar arreglos dinámicos manejados manualmente (new/delete), cumpliendo la restricción..  
**Notas operativas publicidad, límites:** Las notas facilitan comprobar las reglas funcionales por ejemplo, límite de 10.000 favoritos. ¿Sabes? 50 anuncios, prioridad, la publicidad aparece después de dos canciones, etc.

1. Algoritmos implementados debidamente intra-documentados. No exceda la intra-documentación. No use IA para generar la documentación.

R// **Main**

#include <iostream>

#include <string>

#include "sistema.h"

#include "usuario.h"

#include "catalogo.h"

#include "artista.h"

#include "album.h"

#include "cancion.h"

using namespace std;

**/**

**Limpia el buffer de entrada para evitar problemas con getline**

**después de leer números**

**/**

static void limpiarBuffer()

{

cin.clear();

cin.ignore(10000, '\n');

}

**/**

**Lee un número entero de la entrada estándar con validación**

**@return El entero leído**

**/**

static int leerEntero()

{

int x;

while (!(cin >> x))

{

cout << "Entrada invalida. Ingrese un numero:" << endl;

cin.clear();

limpiarBuffer();

}

limpiarBuffer();

return x;

}

**/**

**Lee una línea de texto mostrando un mensaje al usuario**

**@param mensaje Mensaje a mostrar antes de leer**

**@return La cadena leída**

**/**

static string leerLinea(const string& mensaje)

{

string s;

cout << mensaje;

getline(cin, s);

return s;

}

**/**

**Busca el artista al que pertenece un álbum en el catálogo**

**@param c Puntero al catálogo**

**@param album Álbum a buscar**

**\*@return Artista dueño del álbum o nullptr si no se encuentra**

**/**

static Artista\* buscarArtistaDeAlbum(Catalogo\* c, Album\* album)

{

if (c == nullptr || album == nullptr)

return nullptr;

// Recorre todos los artistas buscando el álbum

for (int i = 0; i < c->obtenerCantidadArtistas(); ++i)

{

Artista\* a = c->obtenerArtistaEn(i);

if (a == nullptr) continue;

for (int j = 0; j < a->obtenerCantidadAlbums(); ++j)

{

if (a->obtenerAlbumEn(j) == album)

return a;

}

}

return nullptr;

}

**Sistema**

#include "sistema.h"

#include <fstream>

#include <iostream>

#include <cstdio>

#include <sstream>

using namespace std;

**/\*\***

**\* Constructor del sistema - inicializa arreglos dinámicos**

**\*/**

Sistema::Sistema()

: listaUsuarios(nullptr),

cantidadUsuarios(0),

capacidadUsuarios(0),

rutaArchivoUsuarios("Usuarios.txt")

{

}

**/\*\***

**\* Destructor - libera memoria de usuarios**

**\*/**

Sistema::~Sistema()

{

delete[] listaUsuarios;

listaUsuarios = nullptr;

cantidadUsuarios = 0;

capacidadUsuarios = 0;

}

**/\*\***

**\* Expande la capacidad del arreglo de usuarios cuando se llena**

**\* Duplica el tamaño actual o inicializa con capacidad 2**

**\*/**

void Sistema::aumentarCapacidadUsuarios()

{

int nuevaCapacidad = (capacidadUsuarios == 0) ? 2 : capacidadUsuarios \* 2;

Usuario\*\* nuevoArreglo = new Usuario\*[nuevaCapacidad];

**// Copia los usuarios existentes al nuevo arreglo**

for (int i = 0; i < cantidadUsuarios; ++i)

nuevoArreglo[i] = listaUsuarios[i];

delete[] listaUsuarios;

listaUsuarios = nuevoArreglo;

capacidadUsuarios = nuevaCapacidad;

}

**/\*\***

**\* Carga todos los datos del sistema desde archivos**

**\* @param rutaArtistas Ruta al archivo de artistas**

**\* @param rutaAlbums Ruta al archivo de álbumes**

**\* @param rutaCanciones Ruta al archivo de canciones**

**\* @param rutaCreditos Ruta al archivo de créditos**

**\* @param rutaCancionCreditos Ruta al archivo de relación canción-créditos**

**\* @return true si la carga fue exitosa, false en caso contrario**

**\*/**

bool Sistema::cargarDatos(const std::string& rutaArtistas,

const std::string& rutaAlbums,

const std::string& rutaCanciones,

const std::string& rutaCreditos,

const std::string& rutaCancionCreditos)

{

bool ok = cargador.cargarTodo(

&catalogo,

rutaArtistas,

rutaAlbums,

rutaCanciones,

rutaCreditos,

rutaCancionCreditos

);

if (!ok)

{

cout << "Error: no se pudieron cargar los datos en el Sistema." << endl;

return false;

}

return true;

}

**/\*\***

**\* Reproduce una canción para un usuario según su tipo de membresía**

**\* @param nick Nickname del usuario**

**\* @param idCancion ID de la canción a reproducir**

**\*/**

void Sistema::reproducirParaUsuario(const std::string& nick, int idCancion)

{

Usuario\* u = buscarUsuarioPorNickname(nick);

if (u == nullptr)

{

cout << "Error: usuario '" << nick << "' no encontrado." << endl;

return;

}

Cancion\* c = catalogo.buscarCancionPorId(idCancion);

Album\* a = catalogo.buscarAlbumPorCancionId(idCancion);

if (c == nullptr || a == nullptr)

{

cout << "Error: cancion ID " << idCancion << " no encontrada." << endl;

return;

}

**// Selecciona el plan según el tipo de usuario**

if (u->esPremium())

{

cout << endl << "=== Reproduccion (Plan: premium) ===" << endl;

planPremium.reproducirCancion(u, c, a);

}

else

{

cout << endl << "=== Reproduccion (Plan: estandar) ===" << endl;

planEstandar.reproducirCancion(u, c, a);

}

}

**Usuario**

#include "usuario.h"

#include "cancion.h"

#include <iostream>

using namespace std;

**/\*\***

**\* Constructor de usuario - inicializa todos los atributos**

**\* @param nickname Identificador único del usuario**

**\* @param tipoMembresia "estandar" o "premium"**

**\* @param ciudad Ciudad de residencia**

**\* @param pais País de residencia**

**\* @param fechaInscripcion Fecha en formato AAAAMMDD**

**\*/**

Usuario::Usuario(const std::string& nickname, const std::string& tipoMembresia,

const std::string& ciudad, const std::string& pais, int fechaInscripcion)

: nickname(nickname),

tipoMembresia(tipoMembresia),

ciudad(ciudad),

pais(pais),

fechaInscripcion(fechaInscripcion),

favoritos(nullptr),

cantidadFavoritos(0),

capacidadFavoritos(0),

historialReproduccion(nullptr),

cantidadEnHistorial(0),

posicionActualHistorial(-1),

cancionActual(nullptr),

reproduciendo(false),

contadorCancionesReproducidas(0),

usuarioSeguido(nullptr)

{

**// Solo usuarios premium tienen historial**

if (esPremium()) {

historialReproduccion = new Cancion\*[MAX\_HISTORIAL];

for (int i = 0; i < MAX\_HISTORIAL; ++i) {

historialReproduccion[i] = nullptr;

}

}

**// Valida y corrige tipo de membresía**

if (tipoMembresia != "estandar" && tipoMembresia != "premium") {

this->tipoMembresia = "estandar";

}

**// Valida fecha**

if (!validarFecha(fechaInscripcion)) {

this->fechaInscripcion = 20240101;

}

}

**/\*\***

**\* Expande la capacidad del arreglo de favoritos**

**\* Incrementa hasta máximo 10000 canciones**

**\*/**

void Usuario::aumentarCapacidadFavoritos()

{

int nuevaCapacidad;

if (capacidadFavoritos == 0) {

nuevaCapacidad = 10;

} else {

nuevaCapacidad = capacidadFavoritos \* 2;

}

**// Límite máximo de 10000 favoritos**

if (nuevaCapacidad > 10000) {

nuevaCapacidad = 10000;

}

Cancion\*\* nuevoArreglo = new Cancion\*[nuevaCapacidad];

**// Copia favoritos existentes**

for (int i = 0; i < cantidadFavoritos; ++i) {

nuevoArreglo[i] = favoritos[i];

}

delete[] favoritos;

favoritos = nuevoArreglo;

capacidadFavoritos = nuevaCapacidad;

}

**/\*\***

**\* Agrega una canción a la lista de favoritos del usuario**

**\* Solo disponible para usuarios premium**

**\* @param cancion Puntero a la canción a agregar**

**\* @return true si se agregó exitosamente, false en caso contrario**

**\*/**

bool Usuario::agregarFavorito(Cancion\* cancion)

{

if (!esPremium()) {

cout << "Error: Solo usuarios premium pueden agregar favoritos." << endl;

return false;

}

if (cancion == nullptr) {

cout << "Error: Canción inválida." << endl;

return false;

}

if (estaLlenaListaFavoritos()) {

cout << "Error: Lista de favoritos llena (máximo 10000 canciones)." << endl;

return false;

}

if (tieneFavorito(cancion)) {

cout << "Error: La canción ya está en favoritos." << endl;

return false;

}

**// Expande capacidad si es necesario**

if (cantidadFavoritos == capacidadFavoritos) {

aumentarCapacidadFavoritos();

}

favoritos[cantidadFavoritos] = cancion;

cantidadFavoritos++;

cout << "Canción '" << cancion->obtenerNombre()

<< "' agregada a favoritos." << endl;

return true;

}

**/\*\***

**\* Inicia la reproducción de una canción**

**\* Actualiza historial y contadores de publicidad**

**\* @param cancion Canción a reproducir**

**\*/**

void Usuario::iniciarReproduccion(Cancion\* cancion)

{

if (cancion == nullptr) {

cout << "Error: Canción inválida para reproducir." << endl;

return;

}

cancionActual = cancion;

reproduciendo = true;

**// Solo usuarios premium tienen historial**

if (esPremium()) {

agregarAlHistorial(cancion);

}

**// Contador para publicidad en usuarios estándar**

if (!esPremium()) {

incrementarContadorPublicidad();

}

cout << "Reproduciendo: " << cancion->obtenerNombre() << endl;

}

**Plan estándar**

#include "plan\_estandar.h"

#include <iostream>

using namespace std;

**/\*\***

**\* Constructor - inicializa arreglo de mensajes publicitarios**

**\*/**

PlanEstandar::PlanEstandar()

: listaMensajes(nullptr),

cantidadMensajes(0),

capacidadMensajes(0),

indiceActualPublicidad(0)

{

}

**/\*\***

**\* Expande capacidad del arreglo de mensajes publicitarios**

**\*/**

void PlanEstandar::aumentarCapacidadMensajes()

{

int nuevaCapacidad;

if (capacidadMensajes == 0)

{

nuevaCapacidad = 2;

}

else

{

nuevaCapacidad = capacidadMensajes \* 2;

}

MensajePublicitario\*\* nuevoArreglo = new MensajePublicitario\*[nuevaCapacidad];

for (int i = 0; i < cantidadMensajes; ++i)

{

nuevoArreglo[i] = listaMensajes[i];

}

delete[] listaMensajes;

listaMensajes = nuevoArreglo;

capacidadMensajes = nuevaCapacidad;

}

**/\*\***

**\* Reproduce una canción para usuario estándar**

**\* Incluye publicidad cada 2 canciones**

**\* @param usuario Usuario que reproduce**

**\* @param cancion Canción a reproducir**

**\* @param album Álbum de la canción**

**\*/**

void PlanEstandar::reproducirCancion(Usuario\* usuario, Cancion\* cancion, Album\* album)

{

if (usuario == nullptr || cancion == nullptr || album == nullptr)

{

cout << "Error: datos inválidos para reproducir." << endl;

return;

}

cout << "Portada: " << album->obtenerPortada() << endl;

bool altaCalidad = false;

cancion->reproducir(altaCalidad);

usuario->incrementarContadorPublicidad();

**// Muestra publicidad cada 2 canciones**

if (usuario->debeMostrarPublicidad())

{

mostrarPublicidad();

}

}

**Cancion**

#include "cancion.h"

#include <iostream>

using namespace std;

**/\*\***

**\* Constructor de canción - inicializa atributos básicos**

**\* @param id Identificador único**

**\* @param duracion Duración en segundos**

**\* @param nombre Nombre de la canción**

**\* @param ruta128 Ruta al archivo de audio 128kbps**

**\* @param ruta320 Ruta al archivo de audio 320kbps**

**\*/**

Cancion::Cancion(

int id,

float duracion,

const string& nombre,

const string& ruta128,

const string& ruta320

)

: id(id),

reproducciones(0),

duracion(duracion),

nombre(nombre),

ruta128(ruta128),

ruta320(ruta320),

creditos(nullptr),

cantidadCreditos(0),

capacidadCreditos(0)

{

}

**/\*\***

**\* Reproduce la canción en la calidad especificada**

**\* @param altaCalidad true para 320kbps, false para 128kbps**

**\* @param rutaPortadaAlbum Ruta a la imagen de portada (opcional)**

**\*/**

void Cancion::reproducir(bool altaCalidad, const std::string& rutaPortadaAlbum)

{

cout << "Portada: " << rutaPortadaAlbum << endl;

reproducir(altaCalidad);

}

Plan premium

#include "plan\_premium.h"

#include <iostream>

using namespace std;

**/\*\***

**\* Constructor del plan premium - inicializa el objeto**

**\*/**

PlanPremium::PlanPremium()

{

}

**/\*\***

**\* Destructor del plan premium**

**\*/**

PlanPremium::~PlanPremium()

{

}

**/\*\***

**\* Reproduce una canción para usuario premium**

**\* Incluye portada del álbum y calidad alta (320 kbps)**

**\* @param usuario Usuario que reproduce la canción**

**\* @param cancion Canción a reproducir**

**\* @param album Álbum al que pertenece la canción**

**\*/**

void PlanPremium::reproducirCancion(Usuario\* usuario, Cancion\* cancion, Album\* album)

{

if (usuario == nullptr || cancion == nullptr || album == nullptr)

{

cout << "Error: datos invalidos para reproducir en PlanPremium." << endl;

return;

}

**// Reproduce con portada y calidad alta**

cancion->reproducir(true, album->obtenerPortada());

**// Notificar al usuario (agrega al historial si es premium)**

usuario->iniciarReproduccion(cancion);

}

**/\*\***

**\* Agrega una canción a los favoritos del usuario premium**

**\* @param usuario Usuario que agrega a favoritos**

**\* @param cancion Canción a agregar**

**\* @return true si se agregó exitosamente, false en caso contrario**

**\*/**

bool PlanPremium::agregarAFavoritos(Usuario\* usuario, Cancion\* cancion)

{

if (usuario == nullptr || cancion == nullptr)

{

return false;

}

return usuario->agregarFavorito(cancion);

}

**/\*\***

**\* Retrocede a la canción anterior en el historial de reproducción**

**\* @param usuario Usuario que navega el historial**

**\* @return Canción anterior o nullptr si no hay más**

**\*/**

Cancion\* PlanPremium::retroceder(Usuario\* usuario)

{

if (usuario == nullptr)

{

return nullptr;

}

return usuario->obtenerCancionAnterior();

}

**/\*\***

**\* Avanza a la siguiente canción en el historial de reproducción**

**\* @param usuario Usuario que navega el historial**

**\* @return Siguiente canción o nullptr si no hay más**

**\*/**

Cancion\* PlanPremium::avanzar(Usuario\* usuario)

{

if (usuario == nullptr)

{

return nullptr;

}

return usuario->obtenerCancionSiguiente();

}

**/\*\***

**\* Obtiene la calidad de audio del plan premium**

**\* @return 320 (kbps) para calidad alta**

**\*/**

int PlanPremium::getCalidad() const

{

return 320;

}

**/\*\***

**\* Obtiene el nombre identificador del plan**

**\* @return "premium" como string constante**

**\*/**

const char\* PlanPremium::getNombrePlan() const

{

return "premium";

}

Catalogo

#include "catalogo.h"

using namespace std;

**/\*\***

**\* Constructor del catálogo - inicializa arreglo de artistas**

**\*/**

Catalogo::Catalogo()

: listaArtistas(nullptr),

cantidadArtistas(0),

capacidadArtistas(0)

{

}

**/\*\***

**\* Destructor del catálogo - libera memoria de artistas y sus contenidos**

**\*/**

Catalogo::~Catalogo()

{

for (int i = 0; i < cantidadArtistas; ++i)

{

if (listaArtistas[i] != nullptr)

{

delete listaArtistas[i];

listaArtistas[i] = nullptr;

}

}

delete[] listaArtistas;

listaArtistas = nullptr;

cantidadArtistas = 0;

capacidadArtistas = 0;

}

**/\*\***

**\* Expande la capacidad del arreglo de artistas**

**\* Duplica el tamaño actual o inicializa con capacidad 2**

**\*/**

void Catalogo::aumentarCapacidadArtistas()

{

int nuevaCapacidad;

if (capacidadArtistas == 0)

{

nuevaCapacidad = 2;

}

else

{

nuevaCapacidad = capacidadArtistas \* 2;

}

Artista\*\* nuevoArreglo = new Artista\*[nuevaCapacidad];

**// Copia artistas existentes al nuevo arreglo**

for (int i = 0; i < cantidadArtistas; ++i)

{

nuevoArreglo[i] = listaArtistas[i];

}

delete[] listaArtistas;

listaArtistas = nuevoArreglo;

capacidadArtistas = nuevaCapacidad;

}

**/\*\***

**\* Agrega un artista al catálogo**

**\* @param artista Puntero al artista a agregar**

**\*/**

void Catalogo::agregarArtista(Artista\* artista)

{

if (cantidadArtistas == capacidadArtistas)

{

aumentarCapacidadArtistas();

}

listaArtistas[cantidadArtistas] = artista;

cantidadArtistas = cantidadArtistas + 1;

}

**/\*\***

**\* Busca una canción por su ID en todo el catálogo**

**\* @param idCancion ID de la canción a buscar**

**\* @return Puntero a la canción encontrada o nullptr si no existe**

**\*/**

Cancion\* Catalogo::buscarCancionPorId(int idCancion) const

{

**// Recorre artistas, álbumes y canciones**

for (int i = 0; i < cantidadArtistas; ++i)

{

Artista\* artistaActual = listaArtistas[i];

if (artistaActual == nullptr)

{

continue;

}

for (int j = 0; j < artistaActual->obtenerCantidadAlbums(); ++j)

{

Album\* albumActual = artistaActual->obtenerAlbumEn(j);

if (albumActual == nullptr)

{

continue;

}

for (int k = 0; k < albumActual->obtenerCantidadCanciones(); ++k)

{

Cancion\* cancionActual = albumActual->obtenerCancionEn(k);

if (cancionActual != nullptr)

{

if (cancionActual->obtenerId() == idCancion)

{

return cancionActual;

}

}

}

}

}

return nullptr;

}

**/\*\***

**\* Busca el álbum al que pertenece una canción por ID de canción**

**\* @param idCancion ID de la canción**

**\* @return Puntero al álbum encontrado o nullptr si no existe**

**\*/**

Album\* Catalogo::buscarAlbumPorCancionId(int idCancion) const

{

for (int i = 0; i < cantidadArtistas; ++i)

{

Artista\* artistaActual = listaArtistas[i];

if (artistaActual == nullptr)

{

continue;

}

for (int j = 0; j < artistaActual->obtenerCantidadAlbums(); ++j)

{

Album\* albumActual = artistaActual->obtenerAlbumEn(j);

if (albumActual == nullptr)

{

continue;

}

for (int k = 0; k < albumActual->obtenerCantidadCanciones(); ++k)

{

Cancion\* cancionActual = albumActual->obtenerCancionEn(k);

if (cancionActual != nullptr)

{

if (cancionActual->obtenerId() == idCancion)

{

return albumActual;

}

}

}

}

}

return nullptr;

}

Álbum

#include "album.h"

using namespace std;

**/\*\***

**\* Constructor de álbum - inicializa atributos y arreglo de canciones**

**\* @param id Identificador único del álbum**

**\* @param nombre Nombre del álbum**

**\* @param fechaAAAAMMDD Fecha de lanzamiento en formato AAAAMMDD**

**\* @param selloDisquero Sello discográfico**

**\* @param rutaPortada Ruta a la imagen de portada**

**\* @param puntuación Puntuación del álbum**

**\* @param genero1-género4 Géneros musicales del álbum**

**\*/**

Album::Album(int id, const std::string& nombre, int fechaAAAAMMDD,

const std::string& selloDisquero, const std::string& rutaPortada,

int puntuacion, const std::string& genero1,

const std::string& genero2, const std::string& genero3, const std::string& genero4)

: id(id),

nombre(nombre),

fechaAAAAMMDD(fechaAAAAMMDD),

selloDisquero(selloDisquero),

rutaPortada(rutaPortada),

duracionTotal(0.0f),

puntuacion(puntuacion),

genero1(genero1),

genero2(genero2),

genero3(genero3),

genero4(genero4),

canciones(nullptr),

cantidadCanciones(0),

capacidadCanciones(0)

{

}

**/\*\***

**\* Destructor de álbum - libera memoria de canciones**

**\*/**

Album::~Album()

{

for (int i = 0; i < cantidadCanciones; ++i)

{

if (canciones[i] != nullptr)

{

delete canciones[i];

canciones[i] = nullptr;

}

}

delete[] canciones;

canciones = nullptr;

cantidadCanciones = 0;

capacidadCanciones = 0;

duracionTotal = 0.0f;

}

**/\*\***

**\* Expande la capacidad del arreglo de canciones**

**\* Duplica el tamaño actual o inicializa con capacidad 2**

**\*/**

void Album::aumentarCapacidadCanciones()

{

int nuevaCapacidad;

if (capacidadCanciones == 0)

{

nuevaCapacidad = 2;

}

else

{

nuevaCapacidad = capacidadCanciones \* 2;

}

Cancion\*\* nuevoArreglo = new Cancion\*[nuevaCapacidad];

for (int i = 0; i < cantidadCanciones; ++i)

{

nuevoArreglo[i] = canciones[i];

}

delete[] canciones;

canciones = nuevoArreglo;

capacidadCanciones = nuevaCapacidad;

}

**/\*\***

**\* Recalcula la duración total del álbum sumando todas las canciones**

**\*/**

void Album::recalcularDuracionTotal()

{

float total = 0.0f;

for (int i = 0; i < cantidadCanciones; ++i)

{

if (canciones[i] != nullptr)

{

total = total + canciones[i]->obtenerDuracion();

}

}

duracionTotal = total;

}

**/\*\***

**\* Agrega una canción al álbum y recalcula la duración total**

**\* @param c Puntero a la canción a agregar**

**\*/**

void Album::agregarCancion(Cancion\* c)

{

if (cantidadCanciones == capacidadCanciones)

{

aumentarCapacidadCanciones();

}

canciones[cantidadCanciones] = c;

cantidadCanciones = cantidadCanciones + 1;

recalcularDuracionTotal();

}

Artista

#include "artista.h"

using namespace std;

**/\*\***

**\* Constructor de artista - inicializa atributos y arreglo de álbumes**

**\* @param id Identificador único del artista**

**\* @param edad Edad del artista**

**\* @param nombre Nombre del artista**

**\* @param paisOrigen País de origen del artista**

**\*/**

Artista::Artista(int id, int edad, const string& nombre, const string& paisOrigen)

: id(id),

edad(edad),

seguidores(0),

posicion(0),

nombre(nombre),

paisOrigen(paisOrigen),

albums(nullptr),

cantidadAlbums(0),

capacidadAlbums(0)

{

}

**/\*\***

**\* Destructor de artista - libera memoria de álbumes**

**\*/**

Artista::~Artista()

{

for (int i = 0; i < cantidadAlbums; ++i)

{

if (albums[i] != nullptr)

{

delete albums[i];

albums[i] = nullptr;

}

}

delete[] albums;

albums = nullptr;

cantidadAlbums = 0;

capacidadAlbums = 0;

}

**/\*\***

**\* Expande la capacidad del arreglo de álbumes**

**\* Duplica el tamaño actual o inicializa con capacidad 2**

**\*/**

void Artista::aumentarCapacidadAlbums()

{

int nuevaCapacidad;

if (capacidadAlbums == 0)

{

nuevaCapacidad = 2;

}

else

{

nuevaCapacidad = capacidadAlbums \* 2;

}

Album\*\* nuevoArreglo = new Album\*[nuevaCapacidad];

for (int i = 0; i < cantidadAlbums; ++i)

{

nuevoArreglo[i] = albums[i];

}

delete[] albums;

albums = nuevoArreglo;

capacidadAlbums = nuevaCapacidad;

}

**/\*\***

**\* Agrega un álbum al artista**

**\* @param a Puntero al álbum a agregar**

**\*/**

void Artista::agregarAlbum(Album\* a)

{

if (cantidadAlbums == capacidadAlbums)

{

aumentarCapacidadAlbums();

}

albums[cantidadAlbums] = a;

cantidadAlbums = cantidadAlbums + 1;

}

Crédito

#include "credito.h"

using namespace std;

**/\*\***

**\* Constructor de crédito - inicializa información del colaborador**

**\* @param id Identificador único del crédito**

**\* @param nombre Nombre del colaborador**

**\* @param apellido Apellido del colaborador**

**\* @param categoría Rol o categoría (ej: productor, compositor)**

**\* @param** afiliación Empresa o estudio asociado

\*/

Credito::Credito(int id, const std::string& nombre, const std::string& apellido,

const std::string& categoria, const std::string& afiliacion)

: id(id),

nombre(nombre),

apellido(apellido),

categoria(categoria),

afiliacion(afiliacion)

{

}

**/\*\***

**\* Destructor de crédito**

**\*/**

Credito::~Credito()

{

}

**/\*\***

**\* Obtiene el ID del crédito**

**\* @return ID numérico del crédito**

**\*/**

int Credito::getId() const

{

return id;

}

/\*\*

**\* Obtiene el nombre del colaborador**

**\* @return Nombre como string constante**

**\*/**

const string& Credito::getNombre() const

{

return nombre;

}

**/\*\***

**\* Obtiene la categoría/rol del colaborador**

**\* @return Categoría como string constante**

**\*/**

const std::string& Credito::getCategoria() const

{

return categoria;

}

#include "mensaje\_publicitario.h"

#include <iostream>

using namespace std;

**/\*\***

**\* Incrementa el contador de veces que se ha mostrado el mensaje**

**\*/**

void MensajePublicitario::incrementarContador() {

vecesMostrado++;

}

**/\*\***

**\* Muestra el mensaje publicitario formateado en consola**

**\* Incluye bordes decorativos y categoría**

**\*/**

void MensajePublicitario::mostrarMensaje() const

{

cout << "\n+----------------------------------------+" << endl;

cout << "| MENSAJE PUBLICITARIO |" << endl;

cout << "| Categoria: " << categoria;

int espacios = 26 - (int)categoria.length();

if (espacios < 0) espacios = 0;

for (int s = 0; s < espacios; ++s) cout << ' ';

cout << "|" << endl;

cout << "+----------------------------------------+" << endl;

cout << mensaje << endl;

cout << "+----------------------------------------+" << endl;

cout << "| FIN MENSAJE PUBLICITARIO |" << endl;

cout << "+----------------------------------------+" << endl;

}

1. Problemas de desarrollo que afrontó.

R// problema de fragmentación y perdida de memoria en usuario

Solución: Implementar destructores robustos y verificar con valgrind.

Problema con la lógica compleja para retroceder/avanzar

Solución: Implementar un sistema de posicionamiento con límites claros.

Problema con la coordinación de carga de artistas, álbumes, canciones y créditos.

Solución: Clase CargarCatalogo especializada en la carga secuencial.

Problema con la asignación correcta de prioridades

Solución: Mapeo explícito de categorías a prioridades numéricas.

Problema con los ciclos de referencia y gestión de memoria

Solución: Usar referencias en lugar de propiedad, con cleanup en destructor.

1. Evolución de la solución y consideraciones para tener en cuenta en la implementación.

R//

**1. Patrón Strategy para Planes**

cpp

// En lugar de herencia de usuarios, usamos composición de planes

class PlanEstandar {

void reproducirCancion(Usuario\*, Cancion\*, Album\*);

};

class PlanPremium {

void reproducirCancion(Usuario\*, Cancion\*, Album\*);

};

**Ventaja: Mayor flexibilidad para agregar nuevos planes sin modificar usuarios.**

**2. Gestión Centralizada de Memoria**

cpp

class Sistema {

Usuario\*\* listaUsuarios;

~Sistema() {

delete[] listaUsuarios; // Cleanup centralizado

}

};

**3. Sistema de Carga Modular**

cpp

class CargarCatalogo {

bool cargarTodo(Catalogo\*, const string&, const string&, ...);

};

Beneficio: Separación de concerns - el sistema no sabe cómo se cargan los datos.

**Lecciones Aprendidas**

**1. Sobre Gestión de Memoria**

*// MAL: Posible memory leak*

*Artista\*\* artistas = new Artista\*[capacidad];*

*// BIEN: Destructor robusto*

*~Catalogo() {*

*for (int i = 0; i < cantidadArtistas; ++i) {*

*delete listaArtistas[i]; // Libera cada artista*

*}*

*delete[] listaArtistas; // Libera el arreglo*

*}*

**2. Sobre Diseño de Interfaces**

*// Diseño cohesivo - métodos relacionados juntos*

*class Cancion {*

*void reproducir(bool altaCalidad);*

*void reproducir(bool altaCalidad, const string& portada);*

*// Sobrecarga para diferentes escenarios*

*};*

**3. Sobre Validación de Datos**

*// Validación temprana en constructores*

*Usuario::Usuario(const string& nickname, const string& tipoMembresia, ...) {*

*if (tipoMembresia != "estandar" && tipoMembresia != "premium") {*

*this->tipoMembresia = "estandar"; // Valor por defecto seguro*

*}*

*}*

**Consideraciones para Implementación Futura**

**1. Mejoras de Rendimiento**

*// Actual: Búsqueda lineal O(n)*

*Cancion\* Catalogo::buscarCancionPorId(int idCancion) const {*

*for (int i = 0; i < cantidadArtistas; ++i) {*

*for (int j = 0; j < artista->obtenerCantidadAlbums(); ++j) {*

*// ... triple loop anidado*

*}*

*}*

*}*

*// Futuro: HashMap para búsqueda O(1)*

*std::unordered\_map<int, Cancion\*> mapaCanciones;*

**2. Extensibilidad**

*// Fácil agregar nuevos planes*

*class PlanFamiliar : public PlanBase {*

*// Nueva funcionalidad sin afectar existente*

*};*

**3. Manejo de Errores Más Robusto**

*// Actual: cout para errores*

*cout << "Error: usuario no encontrado." << endl;*

*// Futuro: Sistema de excepciones*

throw UsuarioNoEncontradoException(nickname);

**4. Persistencia Mejorada**

*// Actual: Archivos de texto*

*void Sistema::registrarUsuarioEnArchivo(...) {*

*ofstream out(rutaArchivoUsuarios.c\_str(), ios::app);*

*out << nombreUsuario << ';' << password << ';' ...;*

*}*

*// Futuro: Base de datos SQL*

*// INSERT INTO usuarios VALUES (...);*