



Estas exigencias plantean la necesidad de un manejo eficiente de la memoria (con punteros y estructuras propias) y de un diseño cohesionado y escalable.

```
=====
      U d e A T u n e s
      USUARIO PREMIUM
=====

Bienvenido: checho
Ciudad: Medellin, Colombia
Calidad de audio: 320 kbps (HD)
=====

--- REPRODUCCION ---
1. Reproduccion aleatoria
   (Sin anuncios, controles avanzados)

--- MIS FAVORITOS ---
2. Ver mi lista de favoritos
3. Agregar cancion a favoritos
4. Eliminar cancion de favoritos
5. Seguir lista de otro usuario
6. Reproducir mis favoritos

--- BUSQUEDA ---
7. Buscar cancion por ID
8. Ver todos los artistas
9. Ver todos los albumes
10. Ver todas las canciones

--- MI CUENTA ---
11. Ver mi informacion

12. Cerrar sesion
0. Salir de la aplicacion
=====
```

Fig. 2. Menú de usuario Premium..

## V. PLANTEAMIENTO DE LA SOLUCIÓN

La solución propuesta modela el sistema mediante clases especializadas y relaciones claras. La clase **Aplicacion** actúa como núcleo organizador: contiene punteros (composición UML) a los cinco gestores principales (*GestorUsuarios*, *GestorCanciones*, *GestorArtistas*, *GestorAlbumes*, *GestorPublicidades*). Cada *GestorX* gestiona dinámicamente sus objetos asociados.

Por ejemplo, *GestorUsuarios* crea y almacena objetos **Usuario**; *GestorCanciones* almacena objetos **Cancion**; y así sucesivamente. Estas relaciones «todo-parte» se implementan con composición o agregación: cuando el gestor *posee* objetos que él mismo crea, la relación es composición fuerte (los objetos se destruyen junto con el gestor); si el gestor solo referencia objetos creados externamente, se emplea agregación vía punteros.

En el diseño se emplean punteros en lugar de contenedores STL, lo que refleja agregación: por ejemplo, un **Usuario** mantiene punteros a sus **Canciones** favoritas, o un **Album** conoce punteros a sus **Canciones**. El **Reproductor** es un

componente funcional que interactúa con varios módulos: utiliza el *GestorPublicidades* para obtener anuncios, el *GestorCanciones* para acceder a las pistas de audio, y mantiene punteros al usuario activo.

La encapsulación se asegura declarando atributos privados y ofreciendo métodos públicos (getters/setters) para acceder a ellos. Gracias a esta separación por clases, el sistema evita redundancias: por ejemplo, el uso de punteros en agregación permite referenciar el mismo objeto **Artista** desde múltiples **Canciones** sin duplicarlo

## VI. RESULTADOS

El diseño resultante cumple con los objetivos: la organización modular del código facilita su entendimiento y mantenimiento. Cada gestor de clases implementa operaciones (carga, búsqueda, modificación) sin interferir con otros, respetando la ocultación de información. La interfaz pública de cada clase ofrece funciones claras para el usuario o la aplicación, logrando reutilización de código: las clases se pueden usar en contextos distintos sin cambios internos. La solución basa la robustez en la composición: por ejemplo, al destruir un *GestorCanciones* se libera toda la memoria de sus **Canciones** asociadas (garantizando manejo consistente de la memoria). En términos de eficiencia, el uso de punteros y estructuras propias permitió medir y controlar el consumo de recursos según lo requerido (sin sobrecarga de contenedores genéricos).

```
=====
      U d e A T u n e s
      USUARIO ESTANDAR (GRATIS)
=====

Bienvenido: anibal
Ciudad: medellin, colombia
Calidad de audio: 128 kbps
=====

--- REPRODUCCION ---
1. Reproduccion aleatoria
   (Con publicidad cada 2 canciones)

--- BUSQUEDA ---
2. Buscar cancion por ID
3. Ver todos los artistas
4. Ver todos los albumes
5. Ver todas las canciones

--- MI CUENTA ---
6. Ver mi informacion
7. Hazte Premium! ($19,900/mes)

8. Cerrar sesion
0. Salir de la aplicacion
=====

Opcion: █
```

Fig. 3. Menú de usuario Estándar.

## VII. ANÁLISIS DE LOS RESULTADOS

El empleo de punteros para implementar las relaciones tuvo impacto positivo en la robustez y escalabilidad. La agregación (uso de punteros a objetos existentes) evita copias innecesarias y garantiza referencias compartidas: como señala la documentación de POO, “la clase *A* incluye punteros a objetos de clase *B* creados fuera de *A*, objetos cuyo ciclo de vida no es gestionado por *A*”. Esto reduce la duplicación de memoria y permite que varias clases accedan a un mismo objeto de modelo (por ejemplo, varias listas de reproducción referencian las mismas instancias de **Cancion** sin copias). La modularidad logra asimismo aislar fallos: debido al encapsulamiento de cada clase, cambios en la implementación interna (por ejemplo, en la estrategia de selección de publicidad) no afectan a otras (como los *Gestores* de usuarios o artista), preservando la integridad del estado global. Además, al controlar estrictamente el acceso a los datos internos (con atributos privados), se protege el sistema contra modificaciones indebidas. Como menciona la literatura, este control mejora la integridad y seguridad del sistema: un objeto sólo puede alterarse a través de sus métodos públicos, evitando estados inválidos. En conjunto, la separación en módulos independientes y el uso de composición refuerza la escalabilidad: se pueden agregar nuevas funciones (por ejemplo, nuevos tipos de listas de reproducción) sin modificar las clases ya establecidas.

## VIII. DISCUSIÓN

La implementación actual del sistema UdeATunes incorpora decisiones de diseño explícitas que merecen análisis. En particular, se eligió manejar dinámicamente los datos mediante **punteros crudos** y gestionar manualmente la memoria, renunciando al uso de la STL de C++. Esto otorga un control muy fino de los recursos, permitiendo optimizaciones específicas (por ejemplo, ajustar exactamente la asignación de memoria) en contextos donde el desempeño es crítico. Sin embargo, esta libertad conlleva riesgos importantes: se incrementa la complejidad y la carga de trabajo del desarrollador, quien debe gestionar explícitamente la liberación de memoria para evitar fugas o punteros colgantes.

Adicionalmente, la jerarquía de clases está centrada en una clase “Aplicación” que coordina los subsistemas. Si bien esta organización facilita un punto central de inicialización y control, concentra múltiples responsabilidades en un solo componente. Como advierte la literatura de arquitectura, un diseño que concentra demasiada lógica en una clase puede degenerar en el antipatrón del “**Objeto Dios**”, dificultando el mantenimiento y la escalabilidad del sistema.

En resumen, las decisiones de usar punteros crudos y una clase Aplicación como núcleo proveen flexibilidad y coherencia estructural, pero exigen cuidadosa ingeniería para mitigar sus inconvenientes.

## IX. CONCLUSIÓN

En conclusión, el diseño del sistema UdeATunes logró organizar jerárquicamente las responsabilidades, con la clase *Aplicacion* como núcleo coordinador, y clases gestoras especializadas que interactúan mediante punteros. Se comprobaron los principios fundamentales de la POO: **encapsulamiento** para ocultar detalles internos, **modularidad** en la división del problema y **composición** para las relaciones todo-parte. El uso de punteros en C++ (en lugar de contenedores STL) permitió un control exhaustivo de la memoria dinámica tal como lo exigía el enunciado. Como resultado, el sistema es claro y extensible, con código reutilizable en distintos módulos. Entre las limitaciones identificadas figuran la complejidad añadida de gestionar manualmente la memoria (requisito educativo) y la falta de interfaz gráfica (se implementó en consola). En términos de aprendizaje, este proyecto reafirma la importancia de diseñar clases coherentes y de explotar mecanismos como la composición y la abstracción para desarrollar software robusto y eficiente.

## X. REFERENCIAS

- ESPECIFICACIÓN DEL DESAFÍO UDEATUNES (DESAFÍO II 2025-2), DEPARTAMENTO DE INFORMÁTICA, UDEA.
- WIKIPEDIA – *ENCAPSULAMIENTO (INFORMÁTICA)* (CONSULTADO 2025) [ES.WIKIPEDIA.ORG](https://es.wikipedia.org).
- C. LÓPEZ, *¿QUÉ ES LA PROGRAMACIÓN ORIENTADA A OBJETOS Y CUÁLES SON SUS PRINCIPIOS FUNDAMENTALES?*, CODERSLINK (2023) [CODERSLINK.COM](https://coderslink.com).
- J. M. MEGINO, *AGREGACIÓN VS COMPOSICIÓN EN DIAGRAMAS DE CLASES UML*, BLOG SEAS (2013) [SEAS.ESEAS.ES](https://seas.esseas.es).
- CURSO DE PROGRAMACIÓN 2 (UDEA): NOTAS SOBRE RELACIONES UML Y PUNTEROS EN C++ [PERTUSA.GITBOOKS.IO](https://pertusa.gitbooks.io).
- E. GAMMA ET AL., *DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE*, ADDISON-WESLEY (1994).
- G. BOOCH, *THE UNIFIED MODELING LANGUAGE USER GUIDE*, PEARSON (2005).
- L. J. AGUILAR, *PROGRAMACIÓN ORIENTADA A OBJETOS CON C++*, DÍAZ DE SANTOS (2010).