Gestor Musical

PARTS 1 i 2 del projecte ED del curs 2023/24

Pràctica Estructures de Dades

Avui en dia existeixen moltes utilitats per a gestionar biblioteques de música. Una característica molt apreciada pels usuaris és la capacitat de fer recomanacions basades en les seves preferències. En aquesta pràctica s'implementarà un sistema senzill de reproducció i recomanació de música. Una característica important d'aquesta pràctica serà la utilització d'estructures de dades adients per a optimitzar tant l'ús de la memòria com l'execució de les operacions.

1. Introducció

Aquesta pràctica es pot catalogar dintre del conjunt d'aplicatius anomenats *Gestors Musicals* o *Music Managers*. Aquestes aplicacions tenen com a objectiu principal emmagatzemar i gestionar les col·leccions musicals dels usuaris, proporcionant determinades funcionalitats auxiliars com ara la creació de llistes de reproducció i recomanacions de cançons. Òbviament, la pràctica no consisteix en construir una aplicació comercial d'aquest tipus. Però sí que s'implementaran algunes de les funcionalitats típiques d'aquestes aplicacions, entre elles:

- Recuperació de metadades (títol, intèrpret, gènere, etc.) d'arxius mp3.
- Gestió de llistes de reproducció M3U de les cançons d'una col·lecció musical.
- Reproducció de les cançons en format mp3.
- Recomanacions musicals basada en cerques i creuament de metadades.

Dintre d'aquest domini la pràctica desenvolupada haurà de realitzar amb la suficient eficiència les tasques que s'aniran descrivint. Mostrant així la utilitat de fer servir estructures de dades complexes per augmentar el rendiment del processament de dades. A més, no només el rendiment haurà de ser suficientment bo, sinó que la implementació a baix nivell de les estructures de dades haurà de ser correcta i robusta. Això es comprovarà realitzant una sèrie de tests que avaluaran l'execució —incloent casos extrems— dintre d'un corpus de cançons donat (a banda de l'òbvia depuració i les proves que vosaltres ja heu de fer habitualment durant el desenvolupament).

1.1 Els arxius MP3 i M3U

En aquesta pràctica els arxius musicals utilitzats seran únicament els existents en format MP3. I encara que es podria fer extensiu a d'altres formats, com que l'objectiu de la pràctica no és pas crear una aplicació real, podem fer aquesta simplificació. Respecte als arxius MP3 tingueu present que es tracta d'un format de so comprimit amb pèrdues, en el qual es guarda una representació de les mostres d'àudio i una sèrie de metadades que s'emmagatzemen en una capçalera. Aquestes metadades reben el nom de tags ID3, dels quals hi existeixen diferents versions.

El conjunt de metadades ID3 present en un arxiu MP3 és completament opcional. Així doncs en cada arxiu hi poden haver només certs camps amb un contingut arbitrari. Aquests camps poden ser molt variats, alguns dels quals son: *Title* (títol de la cançó), *Artist* (intèrpret), *Album* (àlbum), *Year* (any de publicació), *Genre* (gènere musical), *ISCR*

(identificador internacional de gravació musical), etc. La llista ha crescut segons les versions, i actualment poden incloure fins i tot diferents imatges associades a la cançó. Però encara que el conjunt d'aquestes metadades és molt obert, dins aquesta pràctica només hi considerarem un subconjunt reduït d'elles per a simplificar. Cal mencionar que existeixen moltes utilitats que permeten editar (tant manualment com de forma automàtica) aquestes metadades. Dins la pràctica es recomana utilitzar alguna per a visualitzar i/o editar les que hi puguin haver dins els arxius MP3 utilitzats.

Un altre format d'arxiu rellevant relacionat amb els MP3 és el **format M3U**. Aquests arxius s'anomenen llistes de reproducció. El seu contingut bàsic és simplement una llista ordenada de cançons, descrita amb un format de text senzill, les quals estan referenciades per un URI. Un **URI (*Uniform Resource Identifier*)** és una generalització de les URL que habitualment fem servir. Fonamentalment és simplement una cadena de text que identifica un recurs, i que dins el format M3U pot ser un simple nom d'arxiu, tant amb path absolut com relatiu. Vegem aquí un exemple senzill d'una llista de reproducció M3U:

```
#EXTM3U
#EXTINF:-1, Artista 1 - Cançó 1
song1.mp3
#EXTINF:-1, Artista 2 - La Meva Primera Cançó
subdirectory1/song01.mp3
```

Fixeu-vos que el format és molt simple de processar: La primera línia es sempre la mateixa seqüència que identifica el tipus d'arxiu: #EXTM3U. Després per a cada cançó hi han dues línies de text, la primera de les quals comença amb el caràcter #, i que als efectes es pot tractar com un comentari; seguida d'una segona que és el path de l'arxiu.

Dintre d'aquesta pràctica la utilitat específica d'aquests arxius serà representar una seqüència de cançons que es reprodueixen seguint l'ordre de la llista. Per tant, no serà necessari tenir en compte totes les opcions que proporciona d'aquest format. El fet d'utilitzar un format establert és per que així es poden utilitzar eines externes per a reproduir i crear aquests arxius. A més d'exemplificar l'ús recomanable de formats oberts quan volem intercanviar dades entre sistemes.

1.2 Recursos necessaris

Degut a que l'objectiu és assolir les competències necessàries per a implementar i utilitzar estructures de dades complexes que permeten fer un processament eficient d'un conjunt d'informació, determinats aspectes de la programació es veuran simplificats artificialment. Això té importants implicacions, com ara la interfície d'ús o les funcionalitats de la mateixa. Per tant, per a determinats aspectes es donaran instruccions concretes, com ara fer servir algunes llibreries externes, utilitzar codi d'exemple o resoldre certes parts d'una determinada forma.

S'ha de tenir en consideració que el model de desenvolupament que s'haurà de seguir serà el model basat en fites, en les quals es presentaran una sèrie d'objectius que s'aniran

afegint segons el curs vagi avançant. L'entorn de desenvolupament serà amb el llenguatge de programació Python (versió 3.8) dins l'entorn Anaconda+Spyder, que els alumnes ja hi coneixen. La interfície del programari a desenvolupar serà per línia de comandes, no essent necessari incorporar una interfície d'usuari gràfica.

Cal mencionar a més que el present enunciant no és una documentació completa. Per tant caldrà consultar documentació externa, com ara els apunts de l'assignatura, la bibliografia recomanada, així com la documentació del llenguatge Python i de les llibreries utilitzades. Així i tot podeu consultar certs aspectes amb els professors o compartir experiències en els grups de l'assignatura.

Seguidament teniu una llista dels recursos necessaris per desenvolupar la pràctica:

- Anaconda: Spyder 4.1.4 + Python 3.8.3.
- Plataforma: Windows, MacOS o Linux.
- Corpus cançons: conjunt free music proporcionat dins d'un enllaç ZIP.
- Editor ID3: Mp3tag[1] (Windows i Mac) ó Puddletag (Linux).
- Reproductor MP3: VLC[2] amb suport M3U.
- python-vlc: binding de la llibreria "libvlc" per reproduir MP3 (cal instal·lar VLC).
- eyed3: llibreria per accedir als tags ID3.

Tingueu present que per a posar en marxa l'entorn de desenvolupament només cal fer una instal·lació del programari Anaconda i del VideoLAN (VLC), i després afegir els paquets de les llibreries esmentades. Per a fer això darrer cal obrir "Anaconda Prompt" i executar les següents dues comandes:

```
$ pip install python-vlc
$ pip install eyed3
```

Si el vostre computador està connectat a Internet es descarregaran els components de Python que permeten fer servir aquestes llibreries, i es configurarà l'entorn. Llavors ja es podran fer els imports corresponents. Tingueu present de tenir-ne instal·lat el VLC (VideoLAN) prèviament en el vostre sistema, o la llibreria "eyed3" donarà errors en temps d'execució.

Finalment, tingueu en compte les següents restriccions dins la pràctica:

- Encara el corpus proporcionat, hi podeu fer servir qualsevol conjunt de música en format MP3, però exclusivament en aquest format.

- Les utilitats esmentades per a editar/reproduir MP3 es proporcionen com a exemples. Hi podeu fer servir qualsevol altre que vos sigui convenient. La

[1] https://www.mp3tag.de/en/

[2] http://www.videolan.org

recomanació és disposar d'un editor de metadades dels MP3 que suporti el format ID3v2, i d'un reproductor que suporti les llistes de reproducció en format M3U.

- El corpus de música s'organitza a partir d'un subdirectori del vostre computador. Anomenarem aquest subdirectori ROOT_DIR. Dintre d'aquest hi podran haver tants subdirectoris amb d'altres subdirectoris com es vulgui.

- A la pràctica serà necessari fer un *import* de l'arxiu "cfg.py". Aquest arxiu conté la configuració del ROOT_DIR, i algunes funcions que podeu fer servir. Podeu modificar les constants (variables amb noms en majúscules) d'aquest arxiu segons les vostres necessitats, però no l'haureu de lliurar, doncs la configuració és pròpia de cada equip a on s'executa la pràctica. Per tant no afegiu ni modifiqueu cap codi dins d'aquest arxiu.

- El conjunt de cançons proporcionat pertany al *Free Music Archive*[3], i es pot fer servir lliurement. Encara que vos animen a que vosaltres feu servir els arxius MP3 que vos siguin més convenients.

- Les llistes de reproducció en format M3U només hi podran ser al directori ROOT_DIR. I les referències incloses en elles hauran de ser relatives a aquest path.

- Dintre de la pràctica, una cançó es considerarà única només des del punt de vista de l'arxiu. És a dir, que un arxiu repetit dues vegades serà considerat com a dos cançons diferents. Igualment, dos arxius amb les mateixes metadades tampoc es considerarà com la mateixa cançó. Però teniu present que dues llistes de reproducció podran contenir la mateixa cançó, la qual cosa serà certa només si apunten al mateix arxiu.

Un darrer comentari important que heu de tenir-ne en compte respecte al projecte: És imprescindible respectar totes les instruccions d'aquesta guia per a poder avaluar correctament el resultat. Això és especialment important en quant a l'execució del vostre codi dins de l'entorn d'execució i proves del Caronte. No s'acceptarà cap lliurament que no es pugui executar, i per tant, es consideraran invàlids els lliuraments que només es puguin executar dins el vostre entorn. Tingueu doncs cura de no actualitzar els elements de l'entorn de desenvolupament, i fer servir exclusivament les versions indicades.

[3] https://www.freemusicarchive.org/static

2. Codi d'exemple

2.1 Descripció

Per tal de fer una primera aproximació, aquí teniu un exemple de codi que implementa algunes de les funcionalitats necessàries de forma simple:

```
# -*- coding: utf-8 -*-
"""
test-mp3.py : Script de proves per reproduïr MP3
"""

import cfg      # Necessari per a la pràctica !!
                # Mireu el contingut de l'arxiu

import os.path
import sys
import numpy    #  installed in anaconda by default
import uuid
import eyed3    #  $ pip install eyed3
import vlc      #  $ pip install python-vlc
import time


# STEP 1: Cerca dels arxius al filesystem
print("Cercant arxius dins [" + cfg.get_root() + "]\n")
uri_file = cfg.get_one_file()  # Aquesta funció és només de proves!
if not os.path.isfile(uri_file):
    print("ERROR: Arxiu MP3 inexistent!")
    sys.exit(1)


# STEP 2: Obtenció de les metadades
metadata = eyed3.load(uri_file)
if metadata is None:
    print("ERROR: Arxiu MP3 erroni!")
    sys.exit(1)
duration  = int(numpy.ceil(metadata.info.time_secs))
title     = metadata.tag.title
artist    = metadata.tag.artist
album     = metadata.tag.album
try:
    genre = metadata.tag.genre.name
except:
    genre = "None"


# STEP 3: Generació del identificador únic
name_file = cfg.get_canonical_pathfile(uri_file)
mp3_uuid  = uuid.uuid5(uuid.NAMESPACE_URL, name_file)


# STEP 4: Reproducció
print("Reproduïnt [{}]".format(uri_file))
```

```
print(" Duració:  {} segons".format(duration))
print(" Títol:    {}".format(title))
print(" Artista:  {}".format(artist))
print(" Àlbum:    {}".format(album))
print(" Gènere:   {}".format(genre))
print(" UUID:     {}".format(mp3_uuid))
print(" Arxiu:    {}".format(name_file))

player = vlc.MediaPlayer(uri_file)

player.play()     # Nota: Crida ASYNC !!

# Poolling loop pel control de la reproducció
timeout = time.time() + duration
while True:
    if time.time() < timeout:
        try:
            time.sleep(1)
        except KeyboardInterrupt:  # STOP amb <CTRL>+<C> a la consola
            break
    else:
        break

player.stop()


# END
print("\nFinal!")
```

En aquest exemple podeu veure les següents funcions:

- Com comprovar l'existència d'un arxiu.
- Com obtenir algunes metadades d'un arxiu MP3.
- Com generar un identificar únic d'un arxiu basat en el seu URI complet.
- Com reproduir un arxiu MP3.

Tingueu present que cal importar sempre l'arxiu "cfg.py"[4] en la vostra pràctica. Com s'indica a la secció 1.2 aquest arxiu conté la configuració del vostre entorn. Reviseu el seu contingut i feu els canvis de les constants per adaptar-les al vostre entorn. A més tingueu present que hi podeu fer servir les funcions declarades dins aquest arxiu, excepte que s'indiqui el contrari.

[4] Aquest arxiu el podeu trobar juntament amb el "test-mp3.py" dins el paquet amb el codi d'exemple de la pràctica.

3. Primer lliurament (de 2)

3.1 Objectius

L'objectiu de la primera part de la pràctica és construir l'esquelet de l'aplicació. Dintre d'aquest esquelet hi haurà una implementació simple d'algunes de les funcionalitats demanades. Principalment es vol aconseguir:

- Familiaritzar-se amb l'entorn de les llibreries i els formats dels arxius utilitzats.
- Tenir-ne una representació a la memòria de les dades d'un arxiu musical.

A partir d'aquí definim una sèrie de funcionalitats que cal implementar:

- Func1: /* llistat d'arxius mp3 */
- Func2: /* generar ID de cançons */
- Func3: /* consultar metadades cançons */
- Func4: /* reproduir cançó amb metadades */
- Func5: /* reproduir llistes de reproducció m3u */
- Func6: /* cercar cançons segons certs criteris */
- Func7: /* generar llista basada en una cerca */

Anem a descriure en detall aquestes funcionalitats i com implementar-les, tenint present que s'indiquen els tipus generals dels paràmetres de les funcions per a que vos resulti més fàcil d'entendre:

3.1.1. Func1 /* llistat d'arxius mp3 */

Aquesta funcionalitat implica obtenir una llista completa de tots els arxius MP3 presents dins la biblioteca musical. Això vol dir recórrer tots els subdirectoris a partir de ROOT_DIR per a obtenir la llista de tots els MP3 existents al filesystem. Cal tenir-ne en compte que aquesta funció es podrà cridar en qualsevol moment per a obtenir una llista «actualitzada» dels arxius presents en la biblioteca.

Tingueu present a més que, internament i dins el codi de la pràctica, el que representa un arxiu és el path relatiu a on es troba dins el filesystem. Per tant, un arxiu queda identificat pel nom de l'arxiu i els subdirectoris des del ROOT_DIR que el contenen. D'aquesta forma un exemple d'arxiu hi seria "subdir1/carpeta-A/song01.mp3"; i seria diferent de l'arxiu "subdir2/ song01.mp3", encara que l'arxiu MP3 físic fos una còpia idèntica.

Per a implementar aquesta funcionalitat hi caldrà crear una classe anomenada "MusicFiles". Aquesta classe en una primera fase haurà d'implementar aquests mètodes:

- `MusicFiles.reload_fs(path: str)`
- `MusicFiles.files_added() -> list`
- `MusicFiles.files_removed() -> list`

Respecte a quests mètodes, fixeu-vos que la funció “MusicFiles.reload_fs()” s’encarrega de mantenir a memòria una representació dels arxius que hi han al disc. Per tant, els mètodes “MusicFiles.files_added()” i “MusicFiles.files_removed()” faran referència només als canvis des de la darrera vegada que s’ha cridat al mètode que llegeix el disc.

També fixeu-vos que aquesta classe sempre treballarà amb Strings que representen directoris o arxius: i per tant les llistes que hi retornaran els mètodes seran llistes de Strings.

3.1.2. Func2 /* generar ID de cançons */

Dintre del programa per a cada arxiu MP3 es generarà un identificador únic de cançó. Aquest identificador seguirà el format estàndard UUID ó GUID de 128 bits. Això vol dir que per a identificar cançons no hi farem servir el path de l’arxiu, sinó un altre valor més compacte. L’avantatge és que podrem fer servir un identificador de mida fixa envers de l’adreça al disc de l’arxiu. Per a fer això disposarem d’una funció que generarà a partir del *path canònic* de l’arxiu un valor únic, el qual serà el ID de la cançó. Un exemple d’aquest UUID seria el següent String “e2960755-c2b7-56bc-b0b1-5d206e899647”.

Ara bé, cal tenir-ne en compte que aquesta funció, encara que té una probabilitat extremadament baixa, podria generar una col·lisió retornant el mateix identificador a partir de dos arxius diferents. Aquest cas extrem aquí no el tindrem en compte (suposarem que els UUID són realment únics en el nostre sistema), però sí que haureu de comprovar que efectivament l’identificador és lliure quan es generi un de nou. En el cas de que no fos així i ja estigui utilitzat, el que caldrà fer és no tenir-ne en compte el nou arxiu trobat. És a dir, que el segon arxiu seria com si no hi fos. Caldrà però avisar d’aquest fet traient un missatge per pantalla que indiqui que aquest arxiu no s’utilitzarà, i continuar després amb l’execució de forma habitual.

Per a implementar aquesta funcionalitat hi caldrà crear una classe anomenada “MusicID”. Aquesta classe en una primera fase haurà d’implementar aquests mètodes:

- `MusicID.generate_uuid(file: str) -> str`
- `MusicID.get_uuid(file: str) -> str`
- `MusicID.remove_uuid(uuid: str)`

Fixeu-vos que aquesta classe també utilitzarà Strings per a representar els identificadors UUID. I que internament haurà de guardar els identificadors utilitzats. Per tant, només el mètode “MusicID.generate_uuid()” haurà de comprovar que l’identificador no està essent utilitzat. Així una vegada un UUID ha estat generat una vegada, només es podria tornar a utilitzar si s’ha cridat a “MusicID.remove_uuid()”. Finalment, el mètode “MusicID.get_uuid()” retornarà el valor del UUID sense cap més comprovació que veure si ha estat generat i és actiu (és a dir, que no hi ha estat esborrat).

3.1.3. Func3 /* consultar metadades cançons */

Donat un arxiu MP3 qualsevol (de fet, donat el path relatiu de l’arxiu), cal poder llegir aquest arxiu i consultar les seves metadades. Aquestes metadades podran existir o no, i per tant, caldrà utilitzar alguna convenció predeterminada quan un camp no hi estigui definit. Així

doncs, internament les metadades es representaran a la pràctica sempre per Strings, i el valor “None” indicarà que el camp està buit. La llista de metadades que obligatòriament heu de considerar són: *title*, *artist*, *album* i *genre*. Tingueu present que el darrer camp, quan s’assigna més d’un estil a la mateixa cançó, es llisten tots al mateix String separats per comes. Així doncs una cançó que sigui “pop” i “rock” al mateix temps, tindria com a gènere “pop, rock”.

A més de les metadades caldrà guardar també de cada cançó el path de l’arxiu. Això vol dir que si utilitzem com a identificador el UUID de la cançó, llavors serà necessari emmagatzemar també no només els camps de les metadades, sinó també el path de l’arxiu original. D’aquesta forma serà possible obtenir l’adreça de l’arxiu al disc a partir del identificar únic de cançó que s’utilitza a la pràctica.

Com podeu veure, existeix doncs una relació 1:1 entre els arxius i les cançons. Els arxius hi queden identificats pels paths relatius, i les cançons pels identificadors UUID.

Per a implementar aquesta funcionalitat hi caldrà crear una classe anomenada “MusicData”. Aquesta classe en una primera fase haurà d’implementar aquests mètodes:

- `MusicData.add_song(uuid: str, file: str)`
- `MusicData.remove_song(uuid: str)`
- `MusicData.load_metadata(uuid: str)`
- `MusicData.get_title(uuid: str) -> str`
- `MusicData.get_artist(uuid: str) -> str`
- `MusicData.get_album(uuid: str) -> str`
- `MusicData.get_genre(uuid: str) -> str`

És important tenir-ne en compte que el mètode “MusicData.add_song()” només crea l’entrada de la cançó, sense afegir els camps de metadades (però sí que guarda el path de l’arxiu). Per tant, inicialment totes les metadades estaran buides fins que es crida al mètode “MusicData.load_metadata()”. A més, s’ha de tenir present que aquest mètode es podria cridar múltiples vegades, especialment si l’arxiu s’ha modificat.

3.1.4. Func4 /* reproduir cançó amb metadades */

Per a reproduir una cançó hi han dos accions que resulten necessàries. La primera és imprimir per pantalla les metadades, i la segona és cridar a un tercer per a reproduir la cançó. Dins la pràctica, la primera acció és farà consultant les metadades ja presents dins la memòria. Mentre que la segona es farà passant a un player l’adreça de l’arxiu MP3. La raó de fer aquesta dicotomia és que el procés de llegir les metadades des de l’arxiu és redundant, i per tant resulta més òptim fer-ho així. En conseqüència l’acció de reproduir utilitzarà com a paràmetre el path de l’arxiu, i l’acció d’imprimir les metadades utilitzarà el id de la cançó.

Aquestes dues accions hi seran per tant independents. Però també caldrà una funció que pugui ajuntar-les de tal forma que es facin alhora. A més, artificialment definirem que aquesta funció es pugui configurar permetent així que es pugui sel·leccionar quines accions executar. Això es farà utilitzant el paràmetre de configuració “cfg.PLAY_MODE”. La raó

d'això serà poder comprovar l'execució de la pràctica dins un test de proves sense haver de reproduir sempre les cançons.

Per a implementar aquesta funcionalitat hi caldrà crear una classe anomenada "MusicPlayer". Aquesta classe en una primera fase haurà d'implementar aquests mètodes:

- `MusicPlayer.print_song(uuid: str)`
- `MusicPlayer.play_file(file: str)`
- `MusicPlayer.play_song(uuid: str, mode: int)`

Cal tenir present que "MusicPlayer.play_file()" és una funció asíncrona, i per tant finalitza (retorna) mentre es reprodueix la cançó en background. En canvi "play_song()" és una funció síncrona, i no retorna fins que la cançó no ha finalitzat, o també si l'usuari interromp la reproducció (reviseu el codi d'exemple per a veure com estar esperant a que una cançó finalitzi, i pugueu aturar aquesta reproducció).

3.1.5. Func5 /* reproduir llistes de reproducció m3u */

Una llista de reproducció en format M3U defineix una llista ordenada d'arxius MP3. Així doncs dins la pràctica caldrà poder llegir aquests arxius simples, i fer una representació interna del seu contingut. Per tal de ser eficients, la representació interna estarà basada en els identificadors de les cançons, els quals s'obtenen a partir dels paths dels arxius. Per tant una primera acció que s'ha de fer és llegir un arxiu M3U; a més de comprovar per a cada arxiu MP3 llistat que hi existeix dins la col·lecció musical. Llavors si aquest hi és es podrà emmagatzemar el seu identificador de cançó dins una llista. Per a no complicar el vostre codi, l'algorisme de càrrega d'arxius M3U es simplifica al mínim fent només això: línia a línia es llegeix l'arxiu, i si el primer caràcter no és un "#" i el final de la línia és ".mp3", llavors s'agafa tota la línia com el path d'un arxiu MP3. Després només resta comprovar que efectivament l'arxiu existeix, i llavors s'afegeix a la llista i es continua llegint la resta de l'arxiu.

Una segona acció a realitzar ha de ser que a partir d'una col·lecció de cançons (és a dir, una llista amb identificadors UUID) es puguin reproduir en ordre les cançons de la llista (utilitzant el mètode "MusicPlayer.play_song()"). En tot cas, és important veure que per a reproduir les cançons que apareixen dins un arxiu M3U cal primer generar els identificadors de cançó (utilitzant les classes anteriors), i que només llavors es podrà cridar al reproductor per a sentir-les, doncs cal tenir-ne els UUID corresponents.

Per a implementar aquesta funcionalitat hi caldrà crear una classe anomenada "PlayList". Aquesta classe de moment haurà només d'implementar aquests mètodes:

- `PlayList.load_file(file: str)`
- `PlayList.play()`

No oblideu manipular correctament les instàncies d'aquesta classe. Cada llista de reproducció serà un nou objecte PlayList. Per tant, guardeu dins alguna estructura simple (un simple array/vector) les diferents llistes que pugueu anar creant.

3.1.6. Func6 /* cercar cançons segons certs criteris */

Una funcionalitat intrínseca d'un gestor musical és poder establir cerques dins la col·lecció musical. Per tant, l'acció de retornar les cançons que compleixen determinats criteris de cerca és fonamental. Dins la pràctica, resulta llavors necessari implementar alguns mètodes per a poder fer cerques d'arxius, encara que siguin senzilles. Per exemple, retornar la llista de les cançons que hi puguin ser d'un determinat estil, o les que pertanyen a un artista, etc. Per a ser suficientment útil, i a la vegada no massa complex, es demana que com a mínim es puguin cercar subcadenes de text dins els camps. És a dir, que cal comprovar la presència d'una cadena de text (substring) dins un String, la qual cosa es pot fer utilitzant les funcions bàsiques de la classe "str", com ara "str.find()".

D'altra banda, també ha de poder ser possible realitzar operacions lògiques bàsiques entre llistes de resultats. Així, per exemple, cal poder trobar cançons que compleixen diferents criteris. Respecte a això, només caldrà implementar els operadors AND i OR de les llistes retornades.

Per a implementar aquesta funcionalitat hi caldrà crear una classe anomenada "SearchMetadata". Aquesta classe en una primera fase haurà d'implementar aquests mètodes:

- `SearchMetadata.title(sub: str) -> list`
- `SearchMetadata.artist(sub: str) -> list`
- `SearchMetadata.album(sub: str) -> list`
- `SearchMetadata.genre(sub: str) -> list`
- `SearchMetadata.and_operator(list1: list, list2: list) -> list`
- `SearchMetadata.or_operator(list1: list, list2: list) -> list`

Cal prestar atenció a que les llistes retornades hi poden ser buides. I també, que els valors han de ser sempre identificadors de cançons (UUID). Fixeu-vos també que aquestes "lists" no són realment un objecte PlayList.

3.1.7. Func7 /* generar llista basada en una cerca */

Com que la funcionalitat anterior és en realitat independent de les llistes de reproducció, però a la vegada a nivell semàntic no hi ha diferencia entre una llista d'identificadors de cançons (resultat d'una cerca) i una llista de reproducció (objecte PlayList), cal implementar una darrera funcionalitat que solucioni aquest fet. Per tant, una nova funcionalitat a implementar dins la primera fase de la pràctica serà afegir a la classe "PlayList" els mètodes necessaris per a poder afegir i treure elements, de tal forma que es pugui emmagatzemar el resultat d'una cerca en una llista de reproducció. S'entén que això pugui semblar artificial a priori, però en la segona part de la pràctica això tindrà més sentit.

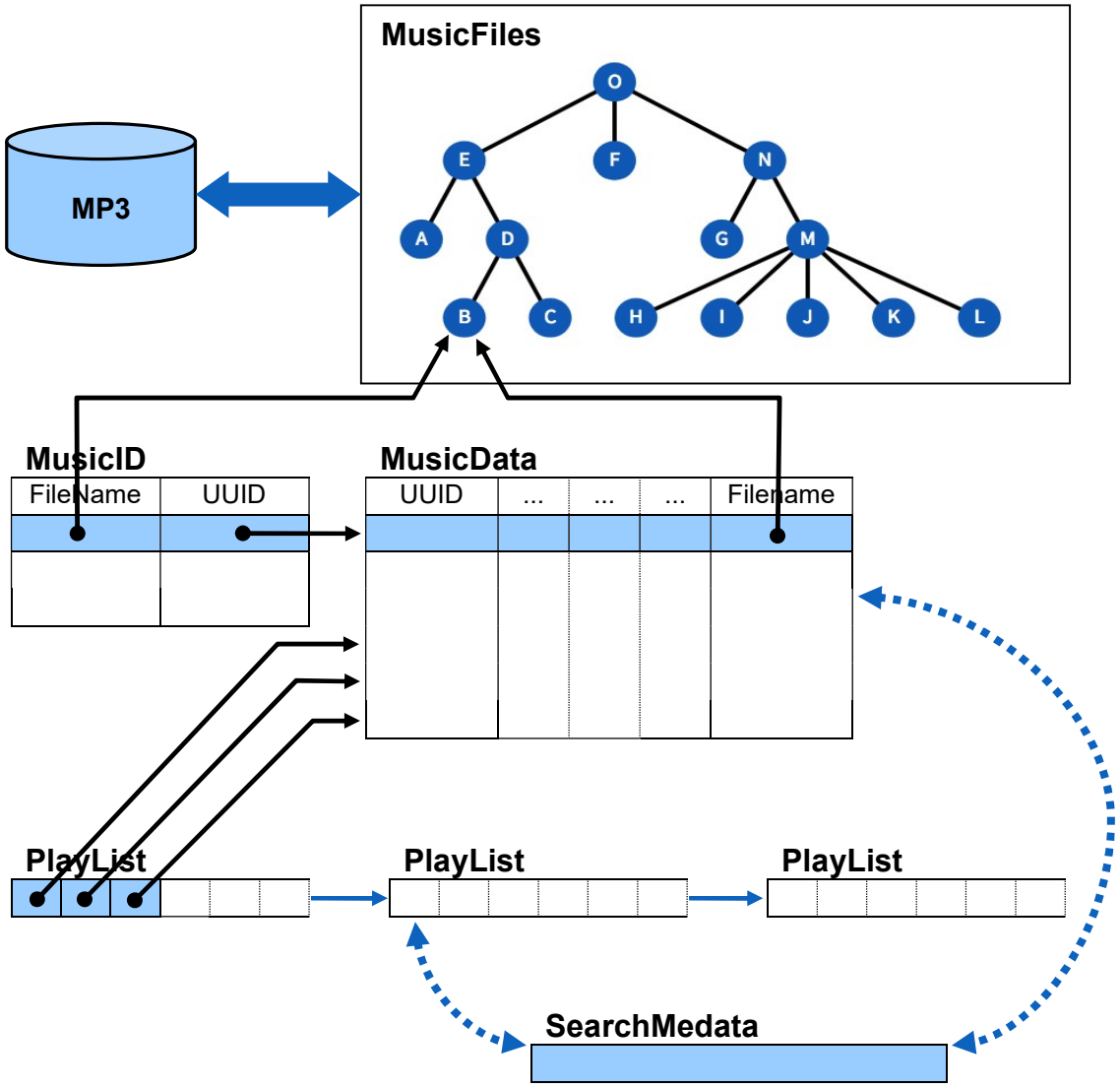
De moment, simplement caldrà ampliar la classe "PlayList" amb els següents mètodes:

- `PlayList.add_song_at_end(uuid: str)`
- `PlayList.remove_first_song()`
- `PlayList.remove_last_song()`

Com es pot veure, aquests mètodes no consumeixen temps fent cerques dels elements que hi guarden, i per tant són ràpids en la seva execució. Així doncs, les úniques operacions que es podran fer seran les més simples possibles: afegir un element al final de la llista, i eliminar un element, que podrà ser el darrer o el primer de la llista.

3.2 Diagrama

Seguidament teniu un diagrama no formal que descriu els elements rellevants de la primera part de la pràctica per a que hi pugueu veure gràficament les relacions entre les diferents estructures de dades:



3.3 Lliurament

Per a fer el lliurament del projecte caldrà fer un upload dels fitxers de la vostra pràctica dins el Caronte. La llista dels arxius necessaris son (fixeu-vos que “cfg.py” no està inclòs):

- `p1_main.py`
- `MusicFiles.py ; MusicID.py ; MusicData.py ; PlayList.py ; SearchMedata.py`
- `Docum.PDF`

El darrer document consisteix en omplir el següent template i generar un arxiu PDF amb les vostres respostes. Això servirà com a documentació bàsica del vostre treball.

Documentació Projecte E.D. - Fase I	
Autors	*Noms i NIUs*
Corpus MP3 proves	*Stock (si/no) / Altres (si/no), indicar nº de cançons i directoris*
Qualificació “grade”	*Valor obtingut amb l’execució del test de proves*
Implementació utilitzada per a les estructures	*Per a cada Classe implementada, explicar el conjunt de dades utilitzat per a emmagatzemar la informació*
Cerques realitzades	*Llista de les cerques de prova realitzades*
Anotacions	*Expliqueu aquí qualsevol comentari rellevant respecte a la vostra implementació*
Diagrama	*Feu in diagrama explicant l’arquitectura de la vostra implementació*

4. Segon lliurament (de 2)

4.1 Objectius

L'objectiu de la segona part de la pràctica consisteix en millorar l'aplicació que s'està desenvolupant, incorporant noves funcionalitats i a la vegada optimitzant el seu comportament. Així doncs, si fins ara a la primera part s'ha construït un esquelet funcional amb les funcionalitats bàsiques de l'aplicació, ara l'objectiu és obtenir una versió més usable de l'aplicació completa. Ara bé, degut a que òbviament es tracta del projecte d'una pràctica i no d'un producte comercial, els components referents a la interfície d'usuari (GUI) de l'aplicació no es desenvoluparan. Per tant, el que resta realitzar és completar i ampliar les funcionalitats, a la vegada que es garanteix que el rendiment sigui el necessari pel domini del problema plantejat.

En aquest sentit les funcionalitats, o canvis, que ara s'introdueixen dins el projecte es basen en tres pilars diferents i que són:

- Ampliació i millora de les interfícies de programació de les classes implementades.
- Implementació de noves funcionalitats.
- Optimització del rendiment adaptant-lo al domini del problema.

A partir d'això es defineixen les següents funcionalitats a implementar:

- Func2.1: /* iteradors i helpers */
- Func2.2: /* constructors i extensions */
- Func2.3: /* classes GrafHash i ElementData */
- Func2.4: /* reimplementació optimitzada classe MusicData */
- Func2.5: /* xarxa de llistes de cançons */
- Func2.6: /* generador recomanacions de cançons */

De forma anàloga a la part precedent del projecte, anem seguidament a descriure en detall aquestes funcionalitats i com implementar-les, concentrant-nos en els detalls més rellevants per a que resulti més entenedor:

4.1.1. Func2.1 /* iteradors i helpers */

Fins ara, cadascuna de les classes que s'havien d'implementar només tenien definides un conjunt reduït de les funciones necessàries. Això donava llibertat per a modificar el disseny i adaptar-lo a les vostres necessitats; a la vegada que vos forçava a trobar solucions als problemes típics relacionats amb l'acoblament entre classes. Però en aquest moment, el disseny ha de ser més estricte en quant a la implementació de les classes del projecte. Per

tant, la primera de les funcionalitats a implementar està relacionada amb definir una interfície comuna més extensa de les diferents classes. En un desenvolupament real això permetria que classes desenvolupades per diferents grups de treball puguin acoblar-se fàcilment, i també que diferents implementacions puguin ser intercanviables dintre de la mateixa aplicació. Per tant ara definirem que per a totes les classes que implementeu cal que cadascuna implementi el mètode:

- `Class.__repr__()`

I addicionalment per a les classes que continguin elements o que siguin elements pròpiament també cal implementar les funcions rellevants de la següent llista:

- `Class.__len__()`
- `Class.__iter__()`
- `Class.__hash__()`
- `Class.__eq__()`
- `Class.__ne__()`
- `Class.__lt__()`

Tingueu present que aquestes funcions auxiliars tenen sentit quan estem manipulant elements o estructures que emmagatzemen elements. Per tant, heu de determinar quines són les classes que necessiten implementar aquestes funcions, i quines funcions en concret. Òbviament no totes les classes del projecte necessiten totes aquestes funcions, però per d'altres resulta imprescindible. Documenteu perquè heu decidit implementar aquestes funcions o perquè no ho heu fet amb cadascuna de les classes.

4.1.2. Func2.2 /* constructors i extensions */

En la primera part del projecte algunes de les parts de les classes dissenyades no hi estaven definides. Ara per conveniència, i en base a les proves que s'havien de fer pel lliurament de la primera part, es defineix explícitament que:

- `MusicPlayer.__init__(obj_music_data: MusicData)`
- `SearchMetadata.__init__(obj_music_data: MusicData)`
- `PlayList.__init__(obj_music_id: MusicID, obj_music_player: MusicPlayer)`

I també que ***totes*** les vostres classes continguin la definició tancada dels seus atributs amb:

- `__slots__ = []`

Això darrer resulta útil perquè les classes del projecte no seran reutilitzades i per tant no necessiten poder ser ampliades afegint més atributs. Mentre que sí que és útil accelerar l'execució definint l'espai que ocupa cada instància.

També caldrà afegir les següents funcions per millorar l'acoblament entre les classes:

- `MusicData.get_filename(uuid: str) -> str`
- `MusicData.get_duration(uuid: str) -> int`

El nom de les funcions vos ha de servir per a determinar la seva funcionalitat.

4.1.3. Func2.3 /* classes GrafHash i ElementData */

Degut a que l'objectiu és conèixer i utilitzar, —però també implementar estructures de baix nivell— dintre del projecte, envers de fer servir sempre estructures predefinides de Python, també té sentit utilitzar algunes estructures pròpies. Per tant, per implementar aquesta funcionalitat caldrà reutilitzar la implementació de la classe *GrafHash* vista a classe, juntament amb una nova classe anomenada *ElementData*. La utilitat d'utilitzar d'aquestes dues classes hi està descrita en el apartat 4.1.4.

En quant a la implementació d'aquestes dues noves classes, cal dir primer que la relació entre les dues és que la classe *GrafHash* guarda objectes de tipus *Vertex*, i que dins aquesta el 'valor' que es guarda és un atribut únic. Per tant, utilitzarem la classe *ElementData* com a un magatzem per a guardar allà tots els atributs que hi necessitem. Llavors podrem utilitzar les instàncies d'aquesta nova classe com els valors que es guardaran dins *Vertex*. Així es per mantenir l'abstracció de la classe *GrafHash* i fer que els canvis que se li hauran de fer internament siguin reduïts.

Respecte a la classe *ElementData* com hem dit ha de ser un simple contenidor de dades, però amb dues característiques importants. La primera és que els seus elements interns han de ser tots *"properties"* que representaran els tipus de dades que té que gestionar la classe *MusicData*. I la segona característica és que els objectes d'aquesta classe han de ser *hashables* i *comparables*. Respecte a com calcular el hash, aquest ha d'estar basat en l'únic atribut que hi pot ser invariable, que hi serà el 'filename'. Finalment aquesta classe haurà de tenir el següent constructor:

- ```
  ElementData.__init__( \
        title="",artist="",album="",genre="",duration=0,filename="")
  ```

En quant a la classe *GrafHash*, hi podeu fer tots els canvis que considereu necessaris per adaptar la solució vista a classe. Però mireu de documentar aquests canvis com a mínim comentant-los dins el codi. Recordeu aplicar el mateixos criteris de la funcionalitat 2.1 a la classe interna *Vertex* de la classe *GrafHash*. A més, caldrà que també incorporeu aquestes funcions:

- `GrafHash.insert_vertex(key, e: ElementData)`
- `GrafHash.get(key) -> ElementData`
- `GrafHash.__contains__(key)`
- `GrafHash.__getitem__(key)`
  ```

- `GrafHash.__delitem__(key)`

Fixeu-vos que la primera funció ‘insert_vertex(key,e)’ té el propòsit d’inserir un nou element dins el graf com a un node. I que la segona funció ‘get(key)’ serveix per a retornar aquest element. Les altres funcions proporcionen funcionalitats similars seguint les convencions de la sintaxis del llenguatge Python.

4.1.4. Func2.4 /* reimplementació classe MusicData */

La classe *MusicData* és la part central del conjunt de dades gestionat per l’aplicació. Per tant, la reimplementació d’aquesta classe tindrà un impacte important en el comportament del programa resultant. És per això que cal definir tota una sèrie de canvis en ella per a poder aconseguir un rendiment adequat i unes funcionalitats que tinguin realment utilitat dintre del domini del problema que s’ha presentat.

Els canvis que cal realitzar es resumeixen en:

- Fer servir la nova classe *ElementData* per desar les metadades de les cançons.
- Utilitzar la classe *GrafHash* per a desar la informació de les cançons.

Respecte al segon canvi, aquest vol dir que totes les dades que ha de gestionar aquesta classe han d’estar guardades dintre d’un graf utilitzant la classe *GrafHash*. En aquest graf els nodes són les cançons, identificades pel UUID, els quals guarden les metadades de cada cançó. Mentre que les arestes representen relacions entre cançons, les quals es defineixen més endavant en el punt 4.1.5.

I respecte al primer canvi, aquest vol dir que cal modificar la classe *GrafHash* per adaptar-la a les especificacions demanades. Bàsicament això implica, dins el nostre domini del problema, que la classe interna *Vertex* s’ha d’utilitzar per manipular parelles de UUID i ElementData. Aquí el primer funcionarà com la ‘key’ que identifica el node; mentre que el segon serà una instància de la classe que emmagatzema les metadades de la cançó.

A més per a poder treballar més fàcilment amb la classe *MusicData* caldrà implementar un iterador que retorni els UUID de totes les cançons emmagatzemades dins aquest objecte:

- `MusicData.__iter__()`

És important veure que el que retorna l’iterador hi seran les keys dels nodes del graf.

Finalment comentar que no cal mantenir l’antiga implementació de la classe *MusicData*, doncs ara la seva funcionalitat és diferent. Però sí cal mantenir totes les funcions ja implementades en la primera part del projecte en la classe *MusicData*.

4.1.5. Func2.5 /* xarxa de llistes de cançons */

Les arestes del graf de la nova versió de la classe *MusicData* representen el número de vegades que dues cançons surten de forma consecutiva dins una Playlist. Per tant es tracta d'un graf dirigit en el qual el node de sortida indica que la cançó precedeix a la cançó del node d'arribada dins una Playlist, i on llavors el pes de l'aresta és el número de vegades que això succeeix.

A partir d'això cal implementar la següent funció dins la classe *MusicData*:

- `MusicData.read_playlist(obj_playlist: PlayList)`

Aquesta funció s'encarrega d'afegir les arestes al graf segons les cançons existents en la Playlist que es passa com a paràmetre. Fixeu-vos que inicialment dins el graf els nodes no estan relacionats entre ells, i que només s'afegeix una aresta si dues cançons són consecutives dins una Playlist. Llavors, quan una aresta es torni a repetir el que caldrà fer és incrementar el valor del pes de l'aresta.

És important comentar que dins una instància de la classe PlayList no és possible que una cançó hi estigui repetida. Aquesta restricció és important que hi sigui, perquè sense ella hi seria possible que una cançó hi estigues precedida d'ella mateixa (és a dir, que hi estigués repetida just després), i llavors la seva representació dins el graf seria utilitzant arestes que surten i entren dins el mateix node. Però això per simplicitat no hi està suportat dins la classe GrafHash. Llavors per aquesta raó la classe PlayList només pot contenir cançons sense repeticions. Cal tenir això present alhora d'implementar les funcions que hi fiquen cançons dins la llista.

A part d'això també resulta necessari actualitzar la classe PlayList de tal forma que ella mateixa sigui capaç de llegir una llista de cançons no des d'un arxiu, sinó també des d'una llista de Python, tal i com les que genera la classe SearchMetadata. Per tant, també caldrà implementar aquest mètode:

- `PlayList.read_list(p_llista: list)`

A on el paràmetre 'p_llista' és una simple llista Python d'identificadors UUID.

4.1.6. Func2.6 /* generador recomanacions cançons */

Una funcionalitat nova, però molt interessant del programari d'aquest projecte consisteix en fer recomanacions a l'usuari basades en el contingut present dins la col·lecció musical. Aquestes recomanacions podrien estar basades en moltes variables, però en aquest cas només utilitzarem les relacions basades en llistes de reproducció. Per tant, per a recomanar

unes determinades cançons el que es planteja és un recorregut del graf de *MusicData* per tal de trobar candidats.

En base a això cal implementar en primer lloc una sèrie de funcions de cerca molt bàsiques:

- `MusicData.get_song_rank(uuid: str) -> int`
- `MusicData.get_next_songs(uuid: str) -> iterator (uuid, value)`
- `MusicData.get_previous_songs(uuid: str) -> iterator (uuid, value)`
- `MusicData.get_song_distance(uuid1:str,uuid2: str) -> (nodes,value)`

La primera funció retorna el que definirem com a "ranking" d'una cançó, que serà la suma de tots els pesos de totes les arestes que entren i surten d'un node.

La segona funció és un iterador sobre els nodes als quals hi arriben arestes des del node que es passa com a paràmetre. Aquest iterador retorna una tupla indicant el node posterior i el pes de l'aresta que el connecta amb el node precedent (que és l'indicat).

La tercera funció és una funció similar a l'anterior, però el que retorna són les cançons que precedeixen a la que es passa com a paràmetre. És a dir, és un iterador sobre la llista de les cançons que tenen arestes que finalitzen en el node indicat. El valor de retorn és una tupla exactament igual que en el cas anterior.

La quarta funció calcula la distància entre dues cançons. Si no hi estan connectades, llavors retornarà un valor de "(0,0)" (és a dir, nodes=0 i value=0). Però si existeix un camí entre totes dues, llavors serà el camí més curt entre elles. El que retorna aquesta funció és una tupla a on el primer valor és el número d'arestes existent al camí; i el segon valor és la suma dels pesos de totes les arestes del camí.

Com es pot veure totes aquestes funcions són funcions de cerca naturals dintre d'un graf, i no representen directament cap dada estrictament relaciona amb cançons, excepte pel nom que els hi hem donat. Per tant, per a poder fer una recomanació, ara afegirem una funció avançada específica del nostre domini del problema. I com que aquesta funció no és específica dels grafs, llavors la hi delegarem a la classe *SearchMetadata* que hi està especialitzada en fer cerques. Així doncs caldrà implementar dins aquesta classe la funció:

- `SearchMetadata.get_similar(uuid: str, max_list: int) -> list`

Aquesta funció ha d'implementar un determinat algorisme de recomanació basat en els següents tres criteris:

1) La llista retornada estarà ordenada en base a la *"semblança entre cançons"* en ordre decreixent.
2) La longitud màxima d'elements de la llista retornada queda limitada pel paràmetre 'max_list'.
3) La *"semblança entre cançons"* es calcula segons el següent algorisme:
    - Tenim cançó A i cançó B, llavors calculem distànciaA_B i distànciaB_A

- A partir de distànciaA_B calculem “(value / nodes) * (rank_A / 2)”
- A partir de distànciaB_A calculem “(value / nodes) * (rank_B / 2)”
- I sumem tots dos resultats.
- *Nota: Cal tenir en compte el cas de que la distància sigui cero!*

Això expressat en format pseudocodi seria:

```
AB_value, AB_nodes = get_song_distance(A,B);
BA_value, BA_nodes = get_song_distance(B,A);
AB = 0; BA = 0;
if (AB_nodes != 0) then
      AB = (AB_value / AB_nodes) * (get_song_rank(A) / 2);
if (BA_nodes != 0) then
      BA = (BA_value / BA_nodes) * (get_song_rank(B) / 2);
semblanca(A,B) = AB + BA ;
```

Finalment, i com a funció auxiliar per exemplificar el funcionament de la cerca de recomanacions, es demana implementar també la següent funció:

- `SearchMetadata.get_topfive() -> list`

Aquesta funció realitza una combinació entre les recomanacions i el rànquing d’una cançó de tal forma que és capaç de retornar les 5 “millors” cançons de tota la col·lecció. Això es calcula en base a un algorisme de cerca que realitza les següents passes:

1) Primer cerca les 5 cançons amb major ranking de tota la col·lecció.
2) Per aquestes 5 cançons fa la cerca de les 5 més similars de cadascuna.
3) Calcula el resultat de la unió de totes aquestes cançons sense repetició, les quals hi seran 25 cançons o menys.
4) Llavors calcula totes les *“semblances entre cançons”* d’aquest subconjunt, i *“ordena”* el resultat. Aquesta ordenació es fa sobre la *“semblança individual”*, que és la suma de tots els resultats de semblança a on la cançó és el primer dels dos paràmetres.
5) Les 5 *“semblances individuals”* amb major resultat serà la llista del top five.

Això expressat en format pseudocodi seria:

```
top_5_list = /* select 5 with greater get_song_rank(X) */

for i in top_5_list do
      simil_5_lists[i] = get_similar(i,5);

top_25_list = /* join(top_5_list & simil_5_lists[_]) */

for i in top_25_list do
      for j in (top_25_list.without(i)) do
            idv_semblanca[i] += semblanca(i,j);

for i in idv_semblanca do
      top_five = /* select 5 greater values of i */
```

Com es pot entendre aquests algorismes de recomanacions són inventats i no aportarien un resultat vàlid en un entorn real. Però mostren el tipus de cerques que caldria realitzar dins l'univers d'elements per a trobar resultats. La qual cosa és necessari que es pugui fer ràpidament per a que sigui pràctic en un programa real dins el nostre domini del problema. De fet, si utilitzéssim aquesta implementació en un programari real hi veuríem que se'n fan recomanacions de forma eficient; només que els resultats pot ser no ens semblarien coherents. En conseqüència només caldria utilitzar algorismes reals per a obtenir resultats amb sentit, mentre que la implementació ja hi estaria disponible.

4.2 Diagrama

L'estructura del programari internament no hi canvia, per tant, el diagrama que defineix la solució continua essent el mateix que a la primera part del projecte. Ara bé, si heu fet canvis en la vostra estructura caldria indicar-los a la documentació.

4.3 Lliurament

Per a fer el lliurament d'aquesta part del projecte caldrà fer un upload dels fitxers de la vostra pràctica seguint les mateixes instruccions de com s'ha fet amb la primera part (en aquest curs mitjançant un formulari GDocs). La llista d'arxius per aquesta segon part hi serà (fixeu-vos que de nou "cfg.py" no està inclòs):

- `P2_main.py`
- `MusicFiles.py ; MusicID.py ; PlayList.py ; SearchMedata.py ; MusicData.py ; GrafHash.py ; ElementData.py`
- `Docum.PDF`

Igualment que amb la part anterior recordeu afegir un arxiu PDF amb la documentació del vostre projecte, indicant com a mínim la següent informació en aquesta ocasió:

Documentació Projecte E.D. - Fase II	
Autors	*Noms i NIUs*
Qualificació "grade"	*Valor obtingut amb l'execució del test de proves.*
Canvis introduïts respecte a la primera part	*Per a les classes que hi hagin estat modificades, cal indicar quines parts han estat modificades i perquè.*

Anàlisis del rendiment	*Comenteu aquí els problemes de rendiment que vos hagueu pogut trobar en la generació de les recomanacions de cançons. Indiqueu possibles solucions o alternatives per a millorar el temps de resposta.*
Proves i test	*Descriviu com heu realitzat els tests del vostre codi. Si heu utilitzat un Corpus de cançons addicionals, digueu com és. Si heu codificat un joc de proves propi comenteu-lo.*
Anotacions	*Expliqueu aquí qualsevol comentari rellevant respecte a la vostra implementació*
Diagrama	*Copieu aquí el diagrama de l'arquitectura de la vostra implementació (de la part I).* *Però si hi han canvis significatius, llavors cal incorporar-los modificant el diagrama i indicant els canvis.*