# SWE Project 1

## Team 14

Abhijith Anil - 2020101030
Jatin Agarwala - 2020111011
Pratyay Suvarnapathaki - 2020111016
Sneha Raghava Raju - 2020101125
Yash Mehan - 2020111020

## Table of Contents

# Task One: Mining the Repository

## Methodology

Given the sheer scale and complexity of the repository at hand[0], it was clear that using various kinds of software to supplement manual exploration of the codebase wasn't just a 'smart choice', but a necessity.

After exploring various options to help us make sense of the repository, we landed on IntelliJ's Diagrams plugin[1] as our primary means of exploration, primarily owing to its versatility in presenting UML diagrams (at different levels of abstraction).

The full UML for Music may be viewed here.

Subsequently, similar to our modus operandi for the Barbershop assignment, team members acted as 'advocates' for the subsystems. The job of an advocate is to greedily ensure the best interests / comprehensive representation of their respective subsystems.

- User management → Sneha
- Library management → Yash
- Lastfm integration → Jatin
- Administrator features → Abhijith

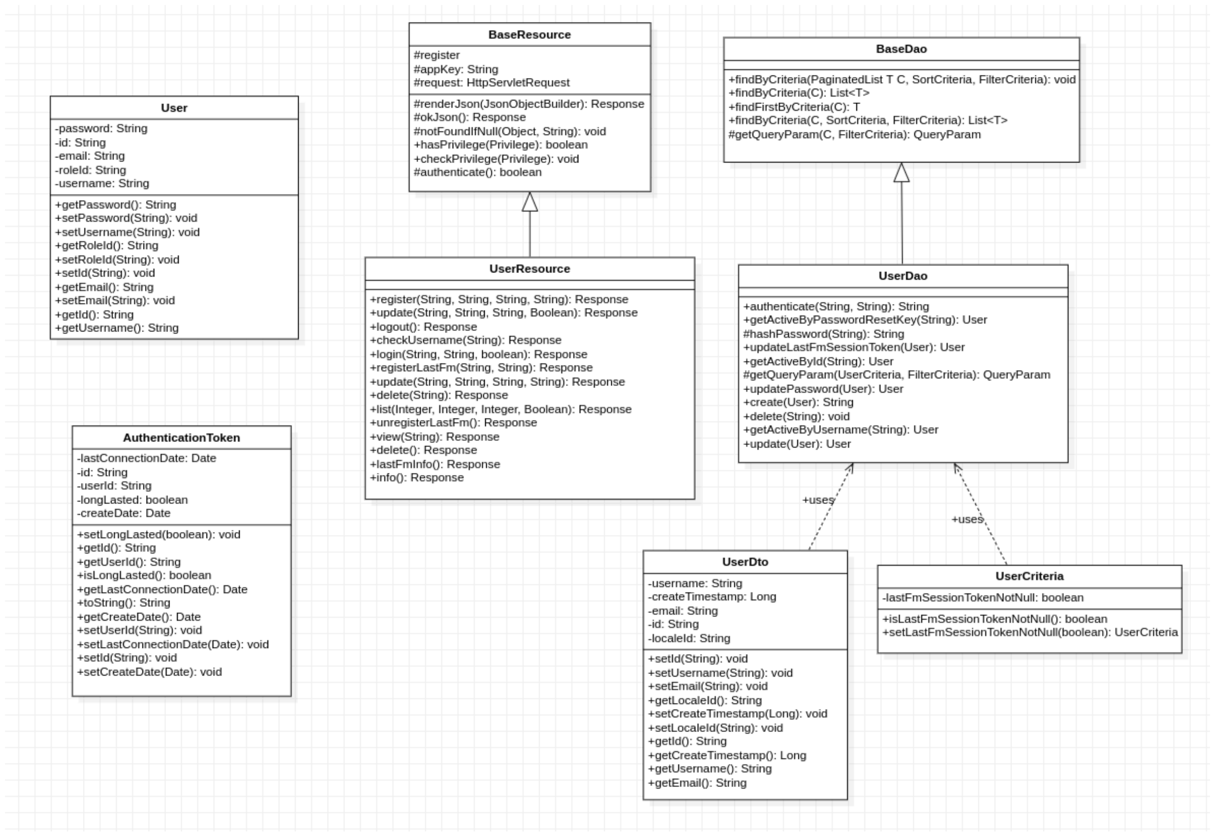
Pratyay acted as the advocate for the overall system, ensuring cohesiveness and interoperability between subsystems, enabling effective collaboration.

Each member identifies all the classes relevant to their respective subsystems. Now begins the task of digging through the mess to make sense of the individual responsibilities of the subsystem classes, their mutual relationships, the flow of information, and external interfacing.

For this step, tools we found handy were VSCode's fuzzy directory search[2] and ChatGPT[3]. The former allowed us to identify dependencies, references and method calls across classes at a deeper level than what the UML would indicate. ChatGPT was a great aid in helping us make sense of these 'snippets of code' obtained via VSCode search. We made it a point to remain aware of ChatGPT's tendency to be a 'nonsense generator'[4] of course.

Based on our refined mental models, we then proceed to create a condensed UML diagram of our respective subsystems, **not necessarily adhering to UML syntax.** The express aim here is to paint a clear and concise picture of the system. This diagram is supplemented with a comprehensive writeup detailing several attributes of each subsystem, as can be seen below.

## 1A. User Management Subsystem


BaseResource
#register
#appKey: String
#request: HttpServletRequest
#renderJson(JsonObjectBuilder): Response
#okJson(): Response
#notFoundIfNull(Object, String): void
+hasPrivilege(Privilege): boolean
+checkPrivilege(Privilege): void
#authenticate(): boolean
BaseDao
+findByCriteria(PaginatedList T C, SortCriteria, FilterCriteria): void
+findByCriteria(C): List<T>
+findFirstByCriteria(C): T
+findByCriteria(C, SortCriteria, FilterCriteria): List<T>
#getQueryParam(C, FilterCriteria): QueryParam
User
-password: String
-id: String
-email: String
-roleId: String
-username: String
+getPassword(): String
+setPassword(String): void
+setUsername(String): void
+getRoleId(): String
+setRoleId(String): void
+setId(String): void
+getEmail(): String
+setEmail(String): void
+getId(): String
+getUsername(): String
UserResource
+register(String, String, String, String): Response
+update(String, String, String, Boolean): Response
+logout(): Response
+checkUsername(String): Response
+login(String, String, boolean): Response
+registerLastFm(String, String): Response
+update(String, String, String, String): Response
+delete(String): Response
+list(Integer, Integer, Integer, Boolean): Response
+unregisterLastFm(): Response
+view(String): Response
+delete(): Response
+lastFmInfo(): Response
+info(): Response
UserDao
+authenticate(String, String): String
+getActiveByPasswordResetKey(String): User
#hashPassword(String): String
+updateLastFmSessionToken(User): User
+getActiveById(String): User
#getQueryParam(UserCriteria, FilterCriteria): QueryParam
+updatePassword(User): User
+create(User): String
+delete(String): void
+getActiveByUsername(String): User
+update(User): User
AuthenticationToken
-lastConnectionDate: Date
-id: String
-userId: String
-longLasted: boolean
-createDate: Date
+setLongLasted(boolean): void
+getId(): String
+getUserId(): String
+isLongLasted(): boolean
+getLastConnectionDate(): Date
+toString(): String
+getCreateDate(): Date
+setUserId(String): void
+setLastConnectionDate(Date): void
+setId(String): void
+setCreateDate(Date): void
+uses
+uses
UserDto
-username: String
-createTimestamp: Long
-email: String
-id: String
-localeId: String
+setId(String): void
+setUsername(String): void
+setEmail(String): void
+getLocaleId(): String
+setCreateTimestamp(Long): void
+setLocaleId(String): void
+getId(): String
+getCreateTimestamp(): Long
+getUsername(): String
+getEmail(): String
UserCriteria
-lastFmSessionTokenNotNull: boolean
+isLastFmSessionTokenNotNull(): boolean
+setLastFmSessionTokenNotNull(boolean): UserCriteria

### Functionality and Behavior

A new user can only be registered by the admin. The ‘register’ method in the UserResource class uses the checkPrivilege(Privilege) method of the BaseResource class (UserResource extends from the BaseResource class) to check if the user is an admin before proceeding.

Then an instance of the User class is created and the username, password, etc variables are set using the User methods: setUsername(username), setPassword(password), etc. Then an instance of the UserDao class is created and its create(User) method is called. This method is used to hash the password and to check if the username already exists. UserDto and UserCriteria are passed as attributes to UserDao.

User login is done by the login(String, String, boolean) method of the UserResource class. The method uses the authenticate(String, String) method of the UserDao class to authenticate the user. It also uses the AuthenticationTokenDao to create a session token.

### Classes Used

- User: The User entity has attributes like password, email, etc. and their respective getter and setter methods (setPassword, setEmail, etc.)
- BaseResource: The BaseResource class handles user authentication and privilege checking.

- UserResource: Handles user actions like register, login, registerLastFm, etc.
- BaseDao: The BaseDao class handles sort and filtering functionality.
- UserDao: Handles user CRUD operations and authentication and password hashing.
- UserDto: Holds user data like id, email and username, and their respective getters and setters.
- UserCriteria: Checks if the user has registered with LastFm (i.e if the LastFm session token exists).

## OOP concepts

- Inheritance: The UserResource class inherits from the BaseResource class. The BaseResource class handles user authentication and privilege checking. The UserResource uses these methods to check if the user is an admin; if true, the admin can register a new user (using the register method). UserDao inherits from the BaseDao class. The BaseDao class handles the sort and filtering functionality. Classes like AlbumDao and ArtistDao, which also inherit from the BaseDao class, use the sort and filter methods to return active artists and get album information.
- Polymorphism (Method Overloading): In the UserResource class, through method overloading, there are two methods named ‘update’ and two named ‘delete’. They differ only by the number/type of the parameters.
- Abstraction: BaseDao and BaseResource are abstract classes; they can have non-abstract and abstract methods (method without body that needs to be implemented by the class that extends it). The implementation details of the Base classes are hidden, and only functionality is shown to the user.

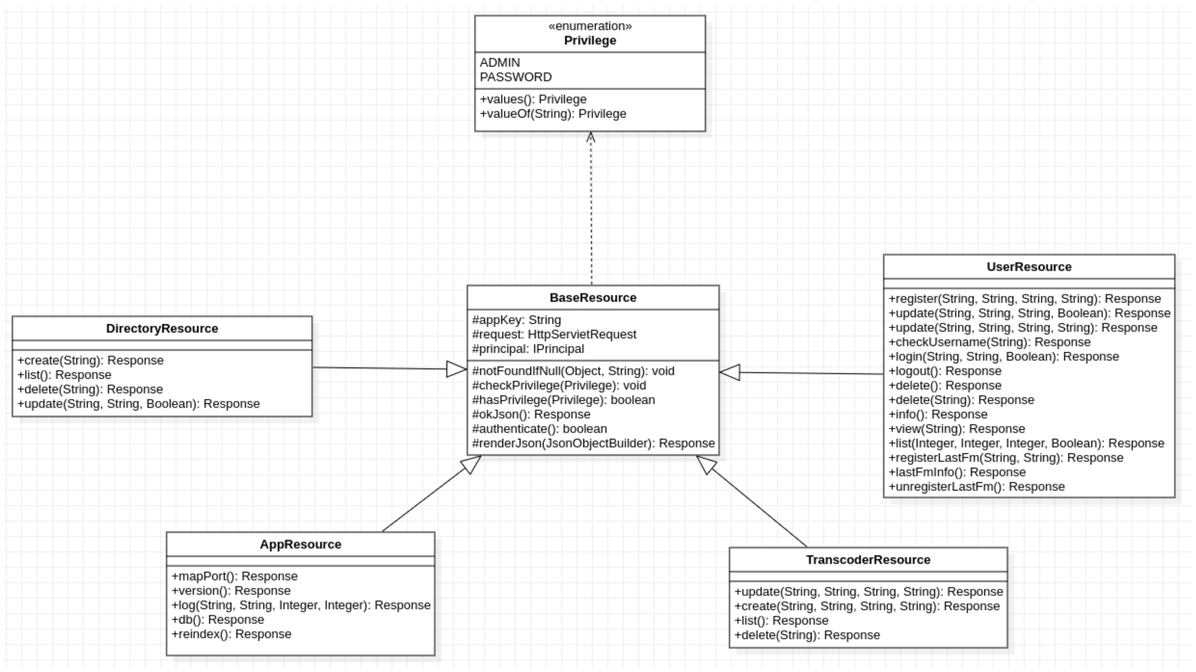
## Strengths

- The admin can view the entire song library and control the local song directory. The admin can have control over the library.
- The admin has complete control over the users. This could help control any malpractice. The admin can create, view, update and delete users.
- Users have multiple options to upload songs; import from a local directory, or an external source (eg, Youtube).

## Weaknesses

- Users can view every other user's songs, so there is no privacy; the library is shared with everyone.
- Only the admin can create new users. Users cannot create an account independently.
- Users cannot set up a directory; only the admin has this capability. Users must depend on the admin to set up local directories for song upload.

## 1B. Administrator features


«enumeration»
Privilege
ADMIN
PASSWORD
+values(): Privilege
+valueOf(String): Privilege
BaseResource
#appKey: String
#request: HttpServletRequest
#principal: IPrincipal
#notFoundIfNull(Object, String): void
#checkPrivilege(Privilege): void
#hasPrivilege(Privilege): boolean
#okJson(): Response
#authenticate(): boolean
#renderJson(JsonObjectBuilder): Response
DirectoryResource
+create(String): Response
+list(): Response
+delete(String): Response
+update(String, String, Boolean): Response
UserResource
+register(String, String, String, String): Response
+update(String, String, String, Boolean): Response
+update(String, String, String, String): Response
+checkUsername(String): Response
+login(String, String, Boolean): Response
+logout(): Response
+delete(): Response
+delete(String): Response
+info(): Response
+view(String): Response
+list(Integer, Integer, Integer, Boolean): Response
+registerLastFm(String, String): Response
+lastFmInfo(): Response
+unregisterLastFm(): Response
AppResource
+mapPort(): Response
+version(): Response
+log(String, String, Integer, Integer): Response
+db(): Response
+reindex(): Response
TranscoderResource
+update(String, String, String, String): Response
+create(String, String, String, String): Response
+list(): Response
+delete(String): Response

### Functionality and Behavior

The admin can add new users to the server, view and update the details of any user, or delete a user. The create() method in UserResource creates a new user by first checking if the current user is an admin (which is done by the checkPrivileges() method in the superclass, BaseResource), and accepts the new user's username, password and email id. The admin can use the update() method in UserResource to change the username, password, locale or email of any user.

The admin can also delete an existing user from the server. This is possible using the delete() method in UserResource, which takes the username as an input parameter and deletes the associated user account from the server.

Another aspect of the administrator system is that the admin can create and change the local directories to which the music library is saved. Creating a new directory is done using the create() method in the class DirectoryResource, which takes the path to the directory, and saves it as a new directory to store the music uploaded. The admin can change the directory information such as the location of the directory. This is realised by the update() method in DirectoryResource. The admin can also delete directories from the server, by using the delete() method in DirectoryResource, which identifies the directory using its id, and removes it from the server.

The admin has additional privileges like music reindexing, creating, updating, and deleting transcoders. The create(), update() and delete() methods in the TranscoderResource class are used to realise the latter. The music collection index can be rebuilt using the reIndex() method in the AppResource class.

## Classes Used

- Privilege: Keeps track of if the user (Principal) has admin privilege or not.
- BaseResource: Responsible for user authentication and checking if the user has admin privilege. It also generates the response json object.
- UserResource: This class handles user scenarios like creating a new user, listing and updating existing users, and deleting users from the system
- TranscoderResource: This class has methods for creating, listing and updating transcoder information by the admin.
- AppResource: This class has methods which allow the admin to retrieve application logs, map ports to the gateway, and reindex the music collection.
- DirectoryResource: The DirectoryResource class handles the admin actions like creating new directories, listing and updating the directories and deleting directories where the music library is stored.

## OOP Concepts used

- Abstraction: The BaseResource class is used to define all the common functionalities for other classes like UserResources, DirectoryResources etc.
- Inheritance: The classes DirectoryResource, UserResource, TranscoderResource and AppResource inherit from BaseResource.
- Polymorphism: Various resources have different methods with the same name. For instance, the class DirectoryResource has multiple definitions for the methods update() and delete() based on the input parameters passed, showing polymorphism through method overloading

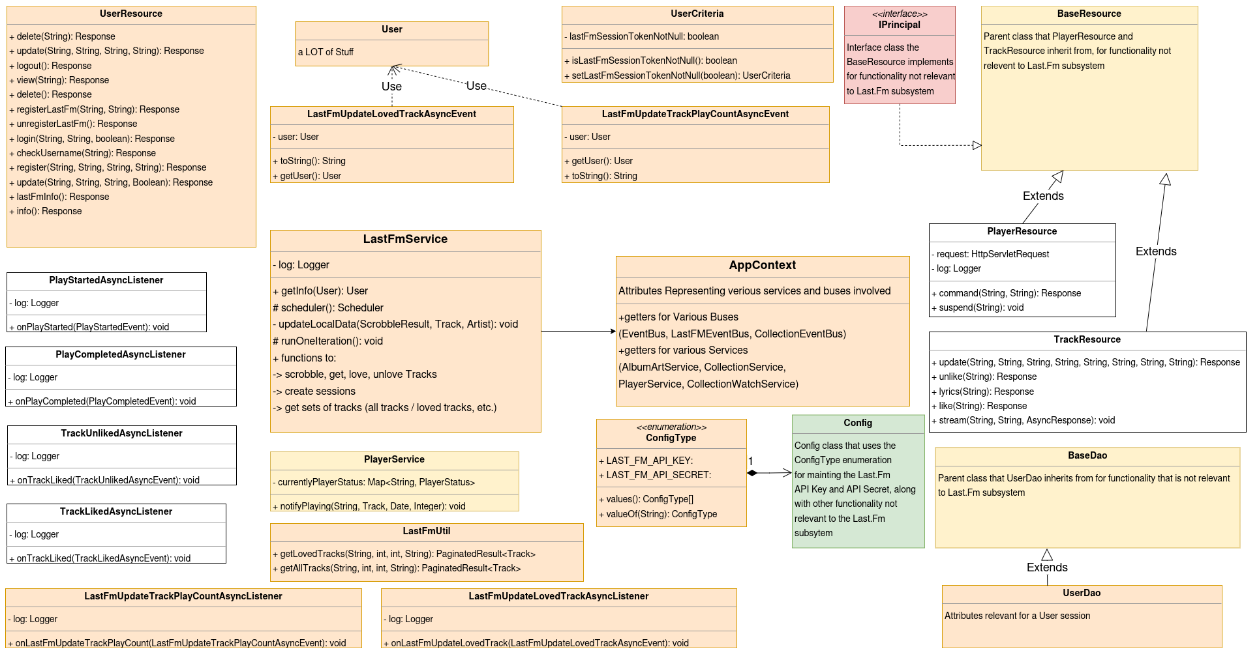
## Strengths

- The system is broken down into several different resource classes, each with a specific responsibility, which can make it easier to understand, maintain and extend the system.
- Only the admin can view the list of all users and their details and delete users from the application.
- Only the admin can change the structure of the music library and the directory in which the files are uploaded to. This adds extra security to the app.
- Allows the admin to get application logs, which will help streamline the application.

## Weaknesses

- The admin can change any user's details, which includes his password, without any prompt to the user, which creates a lack of security.
- The users are overly dependent on the admin for many aspects like creating an account and changing the music library directory.

# 1C. Last.fm Integration


UserResource
+ delete(String): Response
+ update(String, String, String, String): Response
+ logout(): Response
+ view(String): Response
+ delete(): Response
+ registerLastFm(String, String): Response
+ unregisterLastFm(): Response
+ login(String, String, boolean): Response
+ checkUsername(String): Response
+ register(String, String, String, String): Response
+ update(String, String, String, Boolean): Response
+ lastFmInfo(): Response
+ info(): Response
User
a LOT of Stuff
Use
Use
UserCriteria
- lastFmSessionTokenNotNull: boolean
+ isLastFmSessionTokenNotNull(): boolean
+ setLastFmSessionTokenNotNull(boolean): UserCriteria
<<interface>>
IPrincipal
Interface class the BaseResource implements for functionality not relevant to Last.Fm subsystem
BaseResource
Parent class that PlayerResource and TrackResource inherit from, for functionality not relevent to Last.Fm subsystem
LastFmUpdateLovedTrackAsyncEvent
- user: User
+ toString(): String
+ getUser(): User
LastFmUpdateTrackPlayCountAsyncEvent
- user: User
+ getUser(): User
+ toString(): String
Extends
Extends
PlayerResource
- request: HttpServletRequest
- log: Logger
+ command(String, String): Response
+ suspend(String): void
LastFmService
- log: Logger
+ getInfo(User): User
# scheduler(): Scheduler
- updateLocalData(ScrobbleResult, Track, Artist): void
# runOneIteration(): void
+ functions to:
-> scrobble, get, love, unlove Tracks
-> create sessions
-> get sets of tracks (all tracks / loved tracks, etc.)
AppContext
Attributes Representing verious services and buses involved
+getters for Various Buses
(EventBus, LastFMEventBus, CollectionEventBus)
+getters for various Services
(AlbumArtService, CollectionService, PlayerService, CollectionWatchService)
TrackResource
+ update(String, String, String, String, String, String, String, String): Response
+ unlike(String): Response
+ lyrics(String): Response
+ like(String): Response
+ stream(String, String, AsyncResponse): void
PlayStartedAsyncListener
- log: Logger
+ onPlayStarted(PlayStartedEvent): void
PlayCompletedAsyncListener
- log: Logger
+ onPlayCompleted(PlayCompletedEvent): void
TrackUnlikedAsyncListener
- log: Logger
+ onTrackLiked(TrackUnlikedAsyncEvent): void
TrackLikedAsyncListener
- log: Logger
+ onTrackLiked(TrackLikedAsyncEvent): void
PlayerService
- currentlyPlayerStatus: Map<String, PlayerStatus>
+ notifyPlaying(String, Track, Date, Integer): void
LastFmUtil
+ getLovedTracks(String, int, int, String): PaginatedResult<Track>
+ getAllTracks(String, int, int, String): PaginatedResult<Track>
<<enumeration>>
ConfigType
+ LAST_FM_API_KEY:
+ LAST_FM_API_SECRET:
+ values(): ConfigType[]
+ valueOf(String): ConfigType
1
Config
Config class that uses the ConfigType enumeration for mainting the Last.Fm API Key and API Secret, along with other functionality not relevant to the Last.Fm subsytem
BaseDao
Parent class that UserDao inherits from for functionality that is not relevant to Last.Fm subsystem
Extends
UserDao
Attributes relevant for a User session
LastFmUpdateTrackPlayCountAsyncListener
- log: Logger
+ onLastFmUpdateTrackPlayCount(LastFmUpdateTrackPlayCountAsyncEvent): void
LastFmUpdateLovedTrackAsyncListener
- log: Logger
+ onLastFmUpdateLovedTrack(LastFmUpdateLovedTrackAsyncEvent): void

Last.fm[5] is an online service that, according to Wikipedia, “using a music recommender system called ‘Audioscrobbler’, builds a detailed profile of each user’s musical taste by recording details of the tracks the user listens to, either from Internet radio stations, or the user’s computer or many portable music devices”. Music allows users to link their Last.fm profile, and sync their listening history with it.

## Functionality and Behavior

Users can link their Last.fm accounts to their Music accounts by going to “My Account” and providing their Last.fm username and password. After this connection is successful, Music will show the number of songs Scrobbled by Last.FM since a particular date. Linking Last.fm provides with the following functionality:

- Updating “Loved” or “Liked” tracks: Songs that you “love” or “heart” in Music will be exported to the “Loved” tracks of Last.FM (according to the logger, Music treats heart in the Music app as a “Like”, which is translated to “Love” by LastFM logger)
- Songs that you play in Music will be Scrobbled by Last.FM
- Update the Track Play Count in Last.FM based on Music listening

## Classes Used

- AppContext: It is the global application context class. It contains the EventBus lastFmEventBus. It also contains parameters for executing the primary Last.Fm services and functions like AlbumArtService, LastFmService, along with their corresponding getter functions.
- LastFmService: The main class responsible for the LastFM service, that uses the methods of the other classes. It extends AbstractScheduledService. Is

responsible for posting new events for LastFmUpdateLovedTrackAsyncEvent and LastFmUpdateTrackPlayCountAsyncEvent. It uses the lastFM username and password to create a new session. It is also responsible for other tasks like updating the “Now Playing” track, scrobbling tracks, loving and unloving tracks, and importing data of loved tracks and track play counts.
- LastFmUtil: Util class for helping the LastFmService class to carry out its functions
- AlbumArtResource: Class to search for AlbumArt from LastFm
- UserCriteria: Maintains a bool if a user is registered in last.fm to check if if the last.fm session token is null or not
- Listener classes that update activity between Last.fm and Music (such as liking/loving, play counts, etc.)
- PlayerService:  Updates the currently playing track by posting the event to the LastFmEventBus
- ConfigType: Enumerator class containing the configuration parameters: LastFm API Key and Secret
- PlayerResource: Signals Music that a track is being played. A track is considered playing if it is played more than halfway. It also posts a set of tracks played before which is useful for offline mode. Besides, it is also responsible for registering, unregistering and deleting a player
- TrackResource: Has the functionality to implement the liking and unliking of a track by posting to the TrackLikedAsyncEvent and TrackUnlikedAsyncEvent class respectively

## OOP Concepts used

Inheritance
- PlayerResource inherits from BaseResource
- TrackResource inherits from BaseResource
- BaseResource inherits from IPrincipal interface
- UserDao inherits from the BaseDao class

Association
- AppContext is associated with LastFmService
- LastFmUpdateLovedTrackAsyncEvent is associated with User
- LastFmUpdateTrackPlayCountAsyncEvent is associated with User

Composition: The ConfigType class composes the Config class

## Strengths

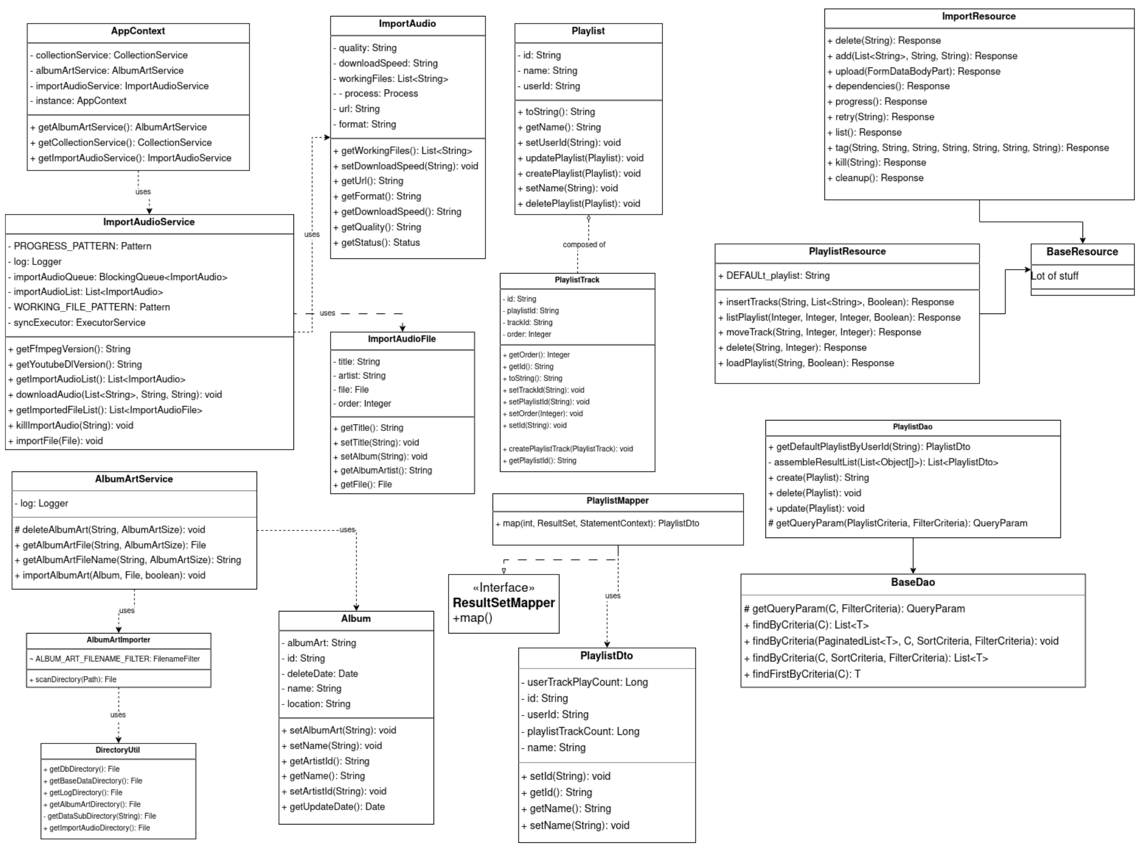
- Your Music account can be easily linked to Last.Fm account using just username and password
- Scrobbling allows you to view all your listening history from different streaming platforms in one area

- The track played counts can be updated on last.fm. This will not only allow you to check your listening history, but also use other feature of Last.fm like “current obsession” based on the insights you gain from your scrobbles
- It also accounts for offline listening, and scrobbles once you are online
- A track that you like in Music gets liked automatically in Last.Fm
- The album art of a track can be searched for online, or using Last.fm

### Weak Points

- Unliking a song from Last.Fm does not get unliked in the Music app
- The scrobbled tracks need to be uploaded to Music in order to update the listening count
- Album art needs to be searched for and set manually
- No option to import the songs directly from last.fm
- Last.Fm can scrobble from Spotify, but spotify tracks cannot be imported to Music, creating a discrepancy
- No option for incognito mode, or pausing of scrobbling without logging out or unlinking of Last.Fm account

## 1D. Library Management


AppContext
- collectionService: CollectionService
- albumArtService: AlbumArtService
- importAudioService: ImportAudioService
- instance: AppContext
+ getAlbumArtService(): AlbumArtService
+ getCollectionService(): CollectionService
+ getImportAudioService(): ImportAudioService
uses
ImportAudioService
- PROGRESS_PATTERN: Pattern
- log: Logger
- importAudioQueue: BlockingQueue<ImportAudio>
- importAudioList: List<ImportAudio>
- WORKING_FILE_PATTERN: Pattern
- syncExecutor: ExecutorService
+ getFfmpegVersion(): String
+ getYoutubeDlVersion(): String
+ getImportAudioList(): List<ImportAudio>
+ downloadAudio(List<String>, String, String): void
+ getImportedFileList(): List<ImportAudioFile>
+ killImportAudio(String): void
+ importFile(File): void
uses
uses
ImportAudio
- quality: String
- downloadSpeed: String
- workingFiles: List<String>
- - process: Process
- url: String
- format: String
+ getWorkingFiles(): List<String>
+ setDownloadSpeed(String): void
+ getUrl(): String
+ getFormat(): String
+ getDownloadSpeed(): String
+ getQuality(): String
+ getStatus(): Status
ImportAudioFile
- title: String
- artist: String
- file: File
- order: Integer
+ getTitle(): String
+ setTitle(String): void
+ setAlbum(String): void
+ getAlbumArtist(): String
+ getFile(): File
Playlist
- id: String
- name: String
- userId: String
+ toString(): String
+ getName(): String
+ setUserId(String): void
+ updatePlaylist(Playlist): void
+ createPlaylist(Playlist): void
+ setName(String): void
+ deletePlaylist(Playlist): void
composed of
PlaylistTrack
- id: String
- playlistId: String
- trackId: String
- order: Integer
+ getOrder(): Integer
+ getId(): String
+ toString(): String
+ setTrackId(String): void
+ setPlaylistId(String): void
+ setOrder(Integer): void
+ setId(String): void
+ createPlaylistTrack(PlaylistTrack): void
+ getPlaylistId(): String
ImportResource
+ delete(String): Response
+ add(List<String>, String, String): Response
+ upload(FormDataBodyPart): Response
+ dependencies(): Response
+ progress(): Response
+ retry(String): Response
+ list(): Response
+ tag(String, String, String, String, String, String, String): Response
+ kill(String): Response
+ cleanup(): Response
PlaylistResource
+ DEFAULt_playlist: String
+ insertTracks(String, List<String>, Boolean): Response
+ listPlaylist(Integer, Integer, Integer, Boolean): Response
+ moveTrack(String, Integer, Integer): Response
+ delete(String, Integer): Response
+ loadPlaylist(String, Boolean): Response
BaseResource
Lot of stuff
PlaylistDao
+ getDefaultPlaylistByUserId(String): PlaylistDto
- assembleResultList(List<Object[]>): List<PlaylistDto>
+ create(Playlist): String
+ delete(Playlist): void
+ update(Playlist): void
# getQueryParam(PlaylistCriteria, FilterCriteria): QueryParam
BaseDao
# getQueryParam(C, FilterCriteria): QueryParam
+ findByCriteria(C): List<T>
+ findByCriteria(PaginatedList<T>, C, SortCriteria, FilterCriteria): void
+ findByCriteria(C, SortCriteria, FilterCriteria): List<T>
+ findFirstByCriteria(C): T
AlbumArtService
- log: Logger
# deleteAlbumArt(String, AlbumArtSize): void
+ getAlbumArtFile(String, AlbumArtSize): File
+ getAlbumArtFileName(String, AlbumArtSize): String
+ importAlbumArt(Album, File, boolean): void
uses
uses
AlbumArtImporter
~ ALBUM_ART_FILENAME_FILTER: FilenameFilter
+ scanDirectory(Path): File
uses
DirectoryUtil
+ getDbDirectory(): File
+ getBaseDataDirectory(): File
+ getLogDirectory(): File
+ getAlbumArtDirectory(): File
- getDataSubDirectory(String): File
+ getImportAudioDirectory(): File
Album
- albumArt: String
- id: String
- deleteDate: Date
- name: String
- location: String
+ setAlbumArt(String): void
+ setName(String): void
+ getArtistId(): String
+ getName(): String
+ setArtistId(String): void
+ getUpdateDate(): Date
PlaylistMapper
+ map(int, ResultSet, StatementContext): PlaylistDto
«Interface»
ResultSetMapper
+map()
uses
PlaylistDto
- userTrackPlayCount: Long
- id: String
- userId: String
- playlistTrackCount: Long
- name: String
+ setId(String): void
+ getId(): String
+ getName(): String
+ setName(String): void

Import and Upload work in two ways: **importing** from the web, or **uploading** local files. Once the file is imported, they can be added to the library after adding metadata information. Users can also edit existing songs. They can change the metadata information, add album cover art, or add the songs to new playlists.

## Functionality and Behavior

Users click a button to add music, which gives them an option to upload music from local or import from the web. After the uploading/importing is done, the user fills in the metadata, which includes artist, album, album art etc. then 'adds' it to the collection. The facility that works in the background is youtube-DL along with ffmpeg.

When the user calls an import, it maps to `/import` route in the backend, which is handled by the `importResource` class. This class has a few other functions which handle the dependencies, progress etc. requests.

When the user calls an upload, it maps to the `/import/upload` route in the backend. That is handled by a function of the sole object of `AppContext`.

## Classes Used

- `AppContext`: The entire app is the object of this class. Any action being done in the app involves the singleton object of this class.
- `ImportResource`: REST request handle, inherits from `BaseResource`
- `ImportAudioService`: maintains the import queue and initiates the importing processes, and cleans up once imports are done. This class is where youtube-dl is called as a process. Inherits from: `AbstractExecutionThreadService`
- `ImportAudio`: when an import is in progress, this class sets and gets import status, progress, speed, audio quality, format etc.
- `AlbumArtImporter` Scans directory for importing album art.
- `AlbumArtService`: imports and deletes album art.
- `DirectoryUtil`: Utilities to gain access to the storage directories used by the app. Includes member variables which hold the path of the directories which have album art, imported music, etc.
- `CollectionService`: for managing music collections, index directories where songs are stored.
- `Playlist`: this is the class for the entity playlist
- `PlaylistTrack:` individual track of a playlist
- `PlaylistDao` Playlist housekeeping functions, like create, update, delete functions for a playlist.
- `PlaylistMapper`: Playlist result set mapper. Is an implementation of interface `ResultSetMapper`
- `PlaylistResource`: REST requests handler for playlist operations

## OOPs Concepts

Inheritance

- `ImportAudioService` inherits from `AbstractExecutionThreadService`
- `PlaylistDao` inherits from `BaseDao`
- `ImportResource` inherits from `BaseResource`

Abstraction: `PlaylistMapper` implements interface `ResultSetMapper`

Association: `AppContext` is associated with `CollectionService`

Composition: `Playlist` is composed of multiple `PlaylistTrack`s

## Strengths

- Uploading multiple songs at once: user can upload a zip instead of uploading each one at a time.
- Downloading and a song can be done in multiple formats and quality.
- Audio importing is multithreaded

## Weak Points

- After uploading a song manually, the user must enter the metadata manually.
- Provision for youtube downloading is in place, but not for Spotify

# Task Two: Analysis

## Methodology

As recommended, we used Sonarqube[6] (specifically, its Dockerized version) in order to conduct primary explorations concerning code quality. In addition to simply detecting code smells, Sonarqube also categorises them depending on their severity. It also provides insights into the quality of the codebase by providing metrics such as overall technical debt. This has been one of the main code metrics we have taken into consideration. We started out by focussing on the smells in the 'blocker' and 'critical' categories.

Subsequently, we used the Designite's[7] IntelliJ plugin[8] to identify probable design smells. Designite uses static analysis techniques to examine source code and identify potential design flaws, and violations of good coding practices.

Thus, using Sonarqube and Designite's respective outputs as starting points, we set out to manually establish correlations between the two. Diving deep into the code, we were able to figure out precise pain points and deviations from best practices. We then used cross-referenced our manually filtered smells with the following reliable online references, to land on our final selection for design smells:

- The course slides[9]
- The 'smells' repository used for Binus University, Indonesia's Code Re-engineering class[10]
- 'A Taxonomy of Design Smells' by Tushar Sharma, the creator of Designite[11]

As for code metrics, the aforementioned Sonarqube dashboard has served as an important at-a-glance measure of how well we have refactored the codebase. As for the recommended tools for code metrics, we have used PMD[12] for getting most of the metrics mentioned in Task 2B below. On exploring the metrics calculated by Checkstyle[13], we found most of the detected flaws to be pedantic regarding formatting, while missing the 'big picture' i.e. the major smells captured by PMD (which have also been discussed in class).

# 2A. Design Smells

## 2A.1: Deficient Encapsulation

Indication: One or more members is not having required protection (eg: public)

Rationale: Exposing details can lead to undesirable coupling. Each change in abstraction can cause change in dependent members

Causes: Easier testability, procedural thinking (expose data as global variables), quick fixes

Impact: Affects changeability, extensibility, reliability,...

Corresponding code smell:

| Code Smell | Description | Examples from code | Potential Fix |
|---|---|---|---|
| Unhidden Public Constructor | Utility classes, which are collections of static members, are not meant to be instantiated | 1. Class Constants in Constants.java<br>2. Class ConfigUtil in ConfigUtil.java<br>3. Class DirectoryUtil in DirectoryUtil.java | - Add a private constructor |

## 2A.2: Unutilized Abstraction

Indication: The 'speculative generality' code smell is an indication of Unutilized Abstraction. There is a class or interface that has never been used in the project.

Rationale: Sacrificing on code quality for the vague prospect of implementing some functionality in the distant future does not make sense

Causes:

- Speculative design: The developer created it based on potential future features.
- Leftover garbage: Initially the class was used, but later on, after refactoring/extending the project, there was no need for it.
- Fear of breaking the code: Fear that deleting the class may lead to bugs/breaks.

Impact: Reduces readability. It makes the code difficult to understand and maintain.

Corresponding code smell:

| Code Smell | Description | Examples from code | Potential Fix |
|---|---|---|---|
| Speculative Generality | Classes or methods that have not been used or implemented | 1. CollectionService, startup and shutdown methods in class CollectionService<br>2. ImportAudioService, startup and run methods in class ImportAudioService | - Add comment explaining why it is empty<br>- Delete the method |

| | | | |
|---|---|---|---|
| | | 3. Classes MessageUtil, UserUtil and DbiTransactionInterceptor | |

## 2A.3: Broken Modularization

Indication: The ‘data class’ code smell is an indication of Broken Modularization. There is a class with only data and no methods.

Rationale: This smell arises when data or methods that ideally should have been in a single abstraction are spread across multiple abstractions.

Causes:

- Procedural thinking in object-oriented languages : This is because procedural programming languages developers assume that data must be separated from the functions that process the data so that in OOP, developers break it into separate classes.
- Lack of knowledge of existing design : In some cases, especially in large companies, there are many classes that a developer should be able to work on apart from the part he is working on. He didn't know about this, so the developer placed the members/methods in the wrong location, which in turn caused the smell.

Impact: Leads to poor maintainability, readability and extensibility.

Corresponding code smell:

| Code Smell | Description | Examples from code | Potential Fix |
|---|---|---|---|
| Data Class | Class containing only data or parameters, and no methods | 1. Album and Artist DTOs<br>2. Mime and MimeType classes | - Refactor |

Admittedly, this smell is merely a consequence of the design patterns used in Music. Refactoring these in actually meaningful, non-trivial ways would be introduce fundamentally breaking changes to the rest of the codebase. Hence, we implement a preliminary fix wherein the above separated abstractions are combined into a single class.

## 2A.4: Unnecessary Abstraction

This smell occurs when an abstraction that is actually not needed is introduced in a software design.

Indication: The lazy class' code smell is an indication of Unnecessary Abstraction. This smell occurs where a class is not doing enough i.e. it does not have a concrete responsibility.

Causes:

- Procedural thinking in object-oriented languages : developers are not familiar with the OOP paradigm so they make mistakes when designing classes.

Impact: Over-engineering adds complexity : developers make designs that are not really needed/overkill, aka ambitious ideas.

Corresponding code smells:

| Code Smell | Description | Examples from code | Fix |
|---|---|---|---|
| Lazy Class | Class is not doing enough i.e. it does not have a concrete responsibility. | MimeType | - Remove class/merge with existing class |
| Method always returning same value | Method returns the same value on all if-else paths of a function | 1. length() method in class Validation | - Refactor |

## 2A.5: Spaghetti Code (leading to Inconsistent Abstraction)

Indication: A telltale sign of this smell is the 'conditional complexity' code smell. It arises when code becomes overly complex and difficult to understand due to its lack of structure and organization.

Cause: When abstraction is not used consistently in the code, it can become difficult to understand how the different parts of the system interact with each other, leading to a tangled mess of code that resembles a plate of spaghetti.

Impact: Can lead to maintenance problems and make it difficult to add new features or modify the code.

Corresponding code smell:

| Code Smell | Description | Examples from code | Fix |
|---|---|---|---|
| Conditional Complexity | Having too many conditional operations (if, else) makes it harder to understand, and high probability of breaking, testing also becomes difficult | 1. Function run() in the Class ImportAudioService<br>2. Function makeMosaic() in the class ImageUtil<br>3. Function doFilter() in the class TokenBasedSecurityFilter | - Remove/merge if-else conditions<br>- Extract some and turn them into methods |

### 2A.6: Cyclic-Dependent Modularization

This smell occurs when a module depends on another module, which also depends on the former.

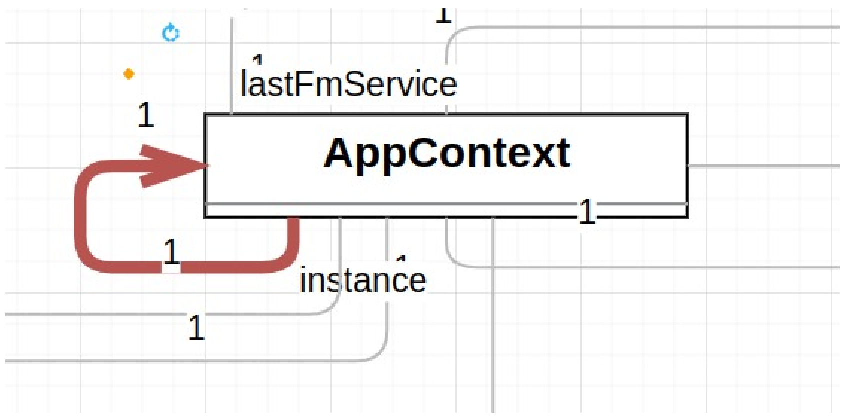
Indication: Upon making the dependency graph of the abstractions, there is a cycle detected. This is said to violate acyclic modularisation technique[14].

Causes:

- As the codebase grows, many modules start to become interdependent
- Can be because of lack of clear division of roles of modules

Impact: Makes it difficult to understand, make changes and maintain software, and can lead to issues like tight coupling.

Instance in code:



## 2A. 7: ...And The Rest

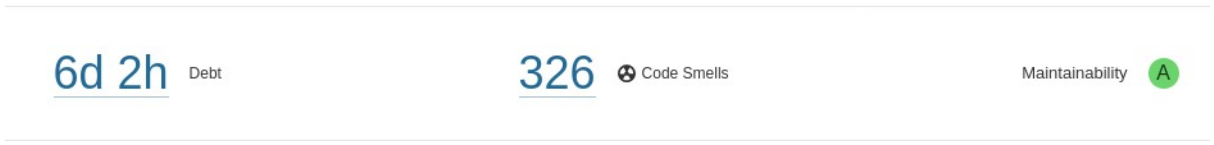
Upon looking through Snoarqube, we identified some additional code smells that don't quite fit into one of our design smells per se, but are nevertheless vitally detrimental to overall code quality (based on their impact on the metrics).

| Code Smell | Description | Examples from code | Fix |
|---|---|---|---|
| Conditional Complexity | Having too many conditional operations (if, else) makes it harder to understand, and high probability of breaking, testing also becomes difficult | 4. Function run() in the Class ImportAudioService<br>5. Function makeMosaic() in the class ImageUtil<br>6. Function doFilter() in the class TokenBasedSecurityFilter | - Remove/merge if-else conditions<br>- Extract some and turn them into methods |
| Duplicated Code | Same code in more than one place | 1. Lines 130-144 are duplicate of 180-194 in PlaylistResource.java<br>2. Lines 39-76 and 84-120 in Role.java are duplicate of 63-100 and 180-216 in Transcoder.java and 45-83 and 109-145 in Artist.java<br>3. Lines 51-88 in UserAlbum.java are duplicate of 57-94 in UserTrack.java | - Put the duplicate code into a method and use it in all places |
| Unnecessary Bloaters | Expressions which always evaluate to 'true' | 1. Line 249 of LastFmService.java<br>2. Line 288 of LastFmService.java | - Remove them |
| Too many comments | Too many comments or blocks of comments | 1. Lines 29-35 in DirectoryDao.java<br>2. Lines 165-170 in CollectionService.java<br>3. Lines 25-27 in DirectoryUtil.java | - Remove them |
| Naming convention not followed | Constant name not following regex '^[A-Z][A-Z0-9](_[A-Z0-9]+)$' | 1. String DEFAULt_playlist at line 37 in PlaylistResource.java | - Rename |
| Poor Performance Code | replaceAll calls java.util.regex.Pattern.compile() everytime, | 1. Line 143 in Directory.java<br>2. Line 44 in LocaleUtil.java | - Replace them |

| | | | |
|---|---|---|---|
| | having a significant performance cost | | |
| Improper if-else | ‘if’ is placed on the same line as the closing }, making code hard to read | 2. Line 34 in DirectoryUtil.java | - Put in next line |
| Generic Exceptions | In several places, generic RTEs of type Exception have been thrown instead of exceptions that actually indicate what specifically went wrong. | 1. UserDao.java : L68, L76<br>2. CollectionReindexAsyncListener.java : L31<br>3. DirectoryCreatedAsyncListener.java : L32<br>4. DirectoryDeletedAsyncListener.java : L32<br>...and a lot more | - Replace with specific and extensible Exception types |

## 2B. Code Metrics

### 2B. 1: SonarQube Metrics


6d 2h Debt

326 Code Smells

Maintainability A

The aforementioned Sonarqube dashboard has served as an important at-a-glance measure of how well we have refactored the codebase. Most crucially, it indicates the overall **technical debt** in the codebase, which is an important metric that has been extensively discussed in class, but which none of the other tools (checkstyle, PMD, etc. seem to capture). The other metric is the total number of **code smells** present.

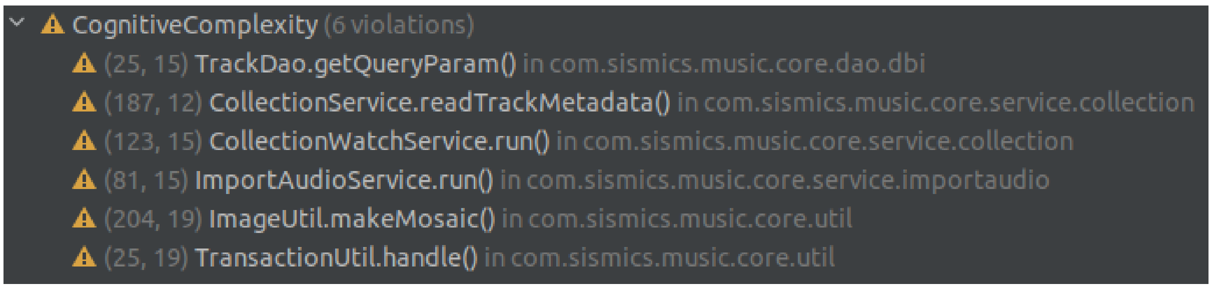
### 2B.2 : Project/Package Lines Of Code (PLOC)

Summarizes Lines Of Code (LOC) metric values for methods of all classes in project/package (exclude test scope).

- Calculated Metrics Value: 11387

### 2B. 3: Cognitive Complexity

Cognitive Complexity is a measure of how difficult a unit of code is to intuitively understand, and is one of the metrics calculated by PMD.

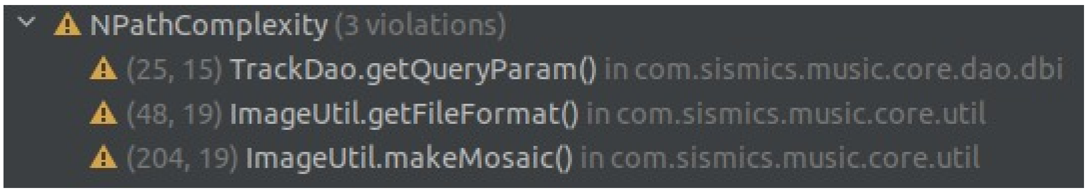


```
CognitiveComplexity (6 violations)
    (25, 15) TrackDao.getQueryParam() in com.sismics.music.core.dao.dbi
    (187, 12) CollectionService.readTrackMetadata() in com.sismics.music.core.service.collection
    (123, 15) CollectionWatchService.run() in com.sismics.music.core.service.collection
    (81, 15) ImportAudioService.run() in com.sismics.music.core.service.importaudio
    (204, 19) ImageUtil.makeMosaic() in com.sismics.music.core.util
    (25, 19) TransactionUtil.handle() in com.sismics.music.core.util
```

## 2B. 4: NPathComplexity

The NPath complexity of a method is the number of acyclic execution paths through that method. While cyclomatic complexity counts the number of decision points in a method, NPath counts the number of full paths from the beginning to the end of the block of the method. That metric grows exponentially, as it multiplies the complexity of statements in the same block. A threshold of 200 is generally considered the point where measures should be taken to reduce complexity and increase readability.

- `TrackDao.getQueryParam()`
- `ImageUtil.getFileFormat()`
- `ImageUtil.makeMosaic()`



```
NPathComplexity (3 violations)
    (25, 15) TrackDao.getQueryParam() in com.sismics.music.core.dao.dbi
    (48, 19) ImageUtil.getFileFormat() in com.sismics.music.core.util
    (204, 19) ImageUtil.makeMosaic() in com.sismics.music.core.util
```

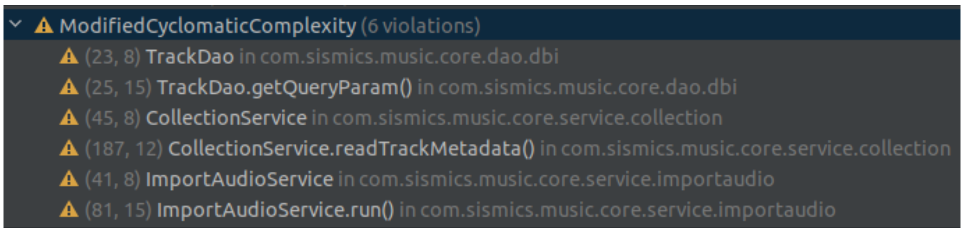
## 2B.5: CyclomaticComplexity

The complexity of methods directly affects maintenance costs and readability. Concentrating too much decisional logic in a single method makes its behaviour hard to read and change.

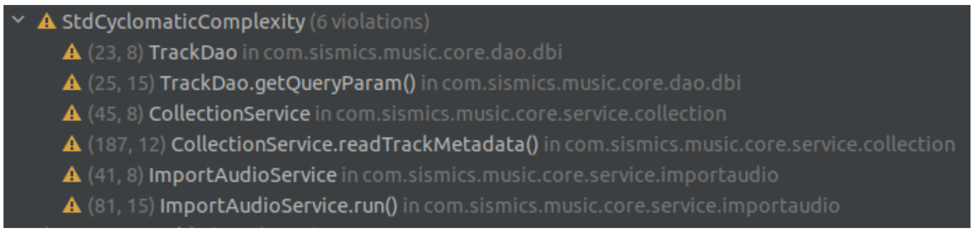
Cyclomatic complexity assesses the complexity of a method by counting the number of decision points in a method, plus one for the method entry.

- `TrackDao.getQueryParam()`
- `CollectionService.readTrackMetadata()`
- `ImportAudioService.run()`
- `ImageUtil.getFileFormat()`
- `ImageUtil.makeMosaic()`

PMD for IntelliJ lists two kinds of cyclomatic complexity: Standard and Modified



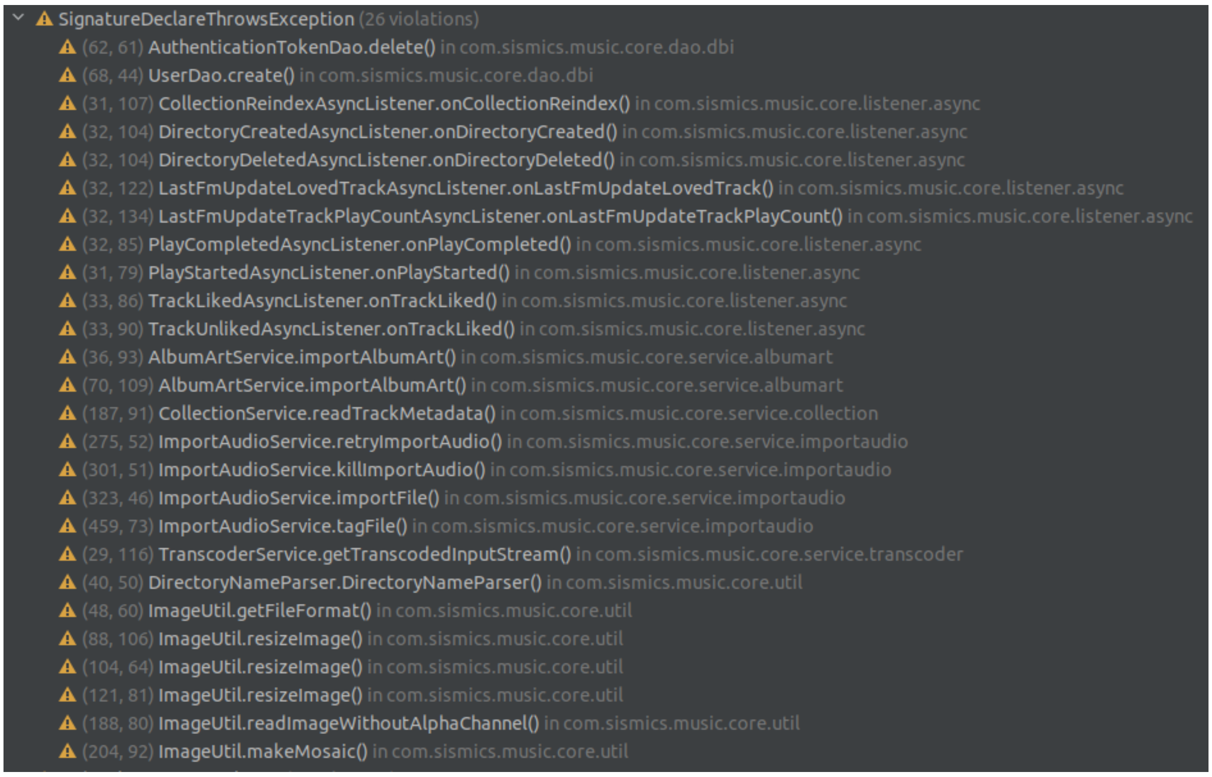
```
⚠ ModifiedCyclomaticComplexity (6 violations)
    ⚠ (23, 8) TrackDao in com.sismics.music.core.dao.dbi
    ⚠ (25, 15) TrackDao.getQueryParam() in com.sismics.music.core.dao.dbi
    ⚠ (45, 8) CollectionService in com.sismics.music.core.service.collection
    ⚠ (187, 12) CollectionService.readTrackMetadata() in com.sismics.music.core.service.collection
    ⚠ (41, 8) ImportAudioService in com.sismics.music.core.service.importaudio
    ⚠ (81, 15) ImportAudioService.run() in com.sismics.music.core.service.importaudio
```



```
⚠ StdCyclomaticComplexity (6 violations)
    ⚠ (23, 8) TrackDao in com.sismics.music.core.dao.dbi
    ⚠ (25, 15) TrackDao.getQueryParam() in com.sismics.music.core.dao.dbi
    ⚠ (45, 8) CollectionService in com.sismics.music.core.service.collection
    ⚠ (187, 12) CollectionService.readTrackMetadata() in com.sismics.music.core.service.collection
    ⚠ (41, 8) ImportAudioService in com.sismics.music.core.service.importaudio
    ⚠ (81, 15) ImportAudioService.run() in com.sismics.music.core.service.importaudio
```

## 2B. 6: SignatureDeclareThrowsException

A method/constructor shouldn't explicitly throw java.lang.Exception, since it is unclear which exceptions that can be thrown from the methods. It might be difficult to document and understand the vague interfaces.



```
⚠ SignatureDeclareThrowsException (26 violations)
    ⚠ (62, 61) AuthenticationTokenDao.delete() in com.sismics.music.core.dao.dbi
    ⚠ (68, 44) UserDao.create() in com.sismics.music.core.dao.dbi
    ⚠ (31, 107) CollectionReindexAsyncListener.onCollectionReindex() in com.sismics.music.core.listener.async
    ⚠ (32, 104) DirectoryCreatedAsyncListener.onDirectoryCreated() in com.sismics.music.core.listener.async
    ⚠ (32, 104) DirectoryDeletedAsyncListener.onDirectoryDeleted() in com.sismics.music.core.listener.async
    ⚠ (32, 122) LastFmUpdateLovedTrackAsyncListener.onLastFmUpdateLovedTrack() in com.sismics.music.core.listener.async
    ⚠ (32, 134) LastFmUpdateTrackPlayCountAsyncListener.onLastFmUpdateTrackPlayCount() in com.sismics.music.core.listener.async
    ⚠ (32, 85) PlayCompletedAsyncListener.onPlayCompleted() in com.sismics.music.core.listener.async
    ⚠ (31, 79) PlayStartedAsyncListener.onPlayStarted() in com.sismics.music.core.listener.async
    ⚠ (33, 86) TrackLikedAsyncListener.onTrackLiked() in com.sismics.music.core.listener.async
    ⚠ (33, 90) TrackUnlikedAsyncListener.onTrackLiked() in com.sismics.music.core.listener.async
    ⚠ (36, 93) AlbumArtService.importAlbumArt() in com.sismics.music.core.service.albumart
    ⚠ (70, 109) AlbumArtService.importAlbumArt() in com.sismics.music.core.service.albumart
    ⚠ (187, 91) CollectionService.readTrackMetadata() in com.sismics.music.core.service.collection
    ⚠ (275, 52) ImportAudioService.retryImportAudio() in com.sismics.music.core.service.importaudio
    ⚠ (301, 51) ImportAudioService.killImportAudio() in com.sismics.music.core.service.importaudio
    ⚠ (323, 46) ImportAudioService.importFile() in com.sismics.music.core.service.importaudio
    ⚠ (459, 73) ImportAudioService.tagFile() in com.sismics.music.core.service.importaudio
    ⚠ (29, 116) TranscoderService.getTranscodedInputStream() in com.sismics.music.core.service.transcoder
    ⚠ (40, 50) DirectoryNameParser.DirectoryNameParser() in com.sismics.music.core.util
    ⚠ (48, 60) ImageUtil.getFileFormat() in com.sismics.music.core.util
    ⚠ (88, 106) ImageUtil.resizeImage() in com.sismics.music.core.util
    ⚠ (104, 64) ImageUtil.resizeImage() in com.sismics.music.core.util
    ⚠ (121, 81) ImageUtil.resizeImage() in com.sismics.music.core.util
    ⚠ (188, 80) ImageUtil.readImageWithoutAlphaChannel() in com.sismics.music.core.util
    ⚠ (204, 92) ImageUtil.makeMosaic() in com.sismics.music.core.util
```

# Task Three: Refactoring

## 3A. Refactoring Design Smells

### 3A. 1: Deficient Encapsulation

Added private constructors to the following classes:

- Class `Constants` in Constants.java
- Class `ConfigUtil` in ConfigUtil.java
- Class `DirectoryUtil` in DirectoryUtil.java

Which now throws an Unsupported Operation Exception because those classes are utility classes and their objects should not be initialised.

### 3A. 2: Unutilized Abstraction

Commented in the following methods to explain why they are empty:

- `Startup and Shutdown in CollectionService`
- `Startup and run in ImportAudioService`

Removed the following classes:

- `MessageUtil`
- `UserUtil`
- `DbiTransactionInterceptor`

These classes were never being used, their objects were never created, nor were they inherited. This is unused code.

### 3A. 3: Broken Modularization

Classes which were very related were combined, to have related member variables and members together. Thus combined `MimeType` and `MimeTypeUtil` into one class `MimeType` that has the data from `MimeType` and the methods from `MimeTypeUtil` .

### 3A. 4: Unnecessary Abstraction

Merged the data and methods in MimeType into the ImportAudioService class, because that's the only place they are being used. It is easier to understand code which has methods and variables declared in the place they are being used/called.

## 3A. 5: Spaghetti Code (Inconsistent Abstraction)

Created the following functions:

- `utilRun()` in `collectionWatchService.java`
- `utilMakeMosaic()` and `utilBufferImageSize3()` and `utilBufferImageSize4()` in `ImageUtil.java`
- `utilList()` in `ResourceUtil.java`
- `utilConsumeQuoteDollar` and `utilConsumeCommitSlash()` in `SQLSplitter.java`
- `utilDoFilter()` in `RequestContextFilter.java`
- `utilStream()` and `utilUpdateYear()` and `utilUpdateOrder` in `TrackResource.java`

## 3A. 6: Cyclically-Dependent Modularization

- In class `AppContext` there is a variable `instance` of `AppContext` class.
- This is misclassification as a smell because the dependency graph around this class forms a cycle. All the other classes that were flagged with this design smell had a dependency to AppContext. It ensures that all the classes use the same instance of `AppContext`, and hence this is desirable behaviour, and not a smell.
- Thus, there was no reason to edit this smell

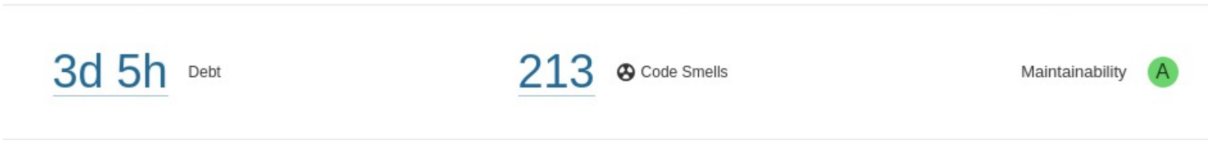
## 3A. 7: Miscellaneous Smells

Following are some code smells that played a role in reducing the technical debt further. Since these were trivial and quite commonplace, we omit mentioning every one of them to avoid verbosity.

- Removing conditional statements that were always true or redundant
- Adding explanations to why some tests were ignored as per the authors
- Refactoring methods so that they do not return the same value every time
- Renaming variables to fit convention
- Renaming variables that had the same name as another variable in a parent class
- Removing empty lines
- Removing blocks of unnecessary comments
- Removing unnecessary typecasting
- Removing trailing whitespaces
- Defining and throwing a dedicated exception instead of using a generic one

# 3B Code Metrics

## 3B. 1: Sonarqube Metrics


3d 5h Debt
213 Code Smells
Maintainability A

We are pleased to report a massive decrease in technical debt, down to 3d5h from the original 6d2h. Since many of our 'miscellaneous' code smells focus on making meaningful refactorings that are in line with code smells that contribute to a large amount of technical debt, we observe a massive improvement in this metric.

Consequently, the number of code smells detected by Sonarqube is also down to 213 from the original 326.

## 3B. 2: Lines of Code:

- Original: 11387
- Refactored: 11376

Lines of code is simply a statistic and not a meaningful metric on its own. We have kept it to show the change in the number before and after refactoring. While we have removed many lines of unnecessary code and comment blocks, we have also added methods and classes to reduce the code's complexity, causing the lines of code to decrease by only 11 lines.

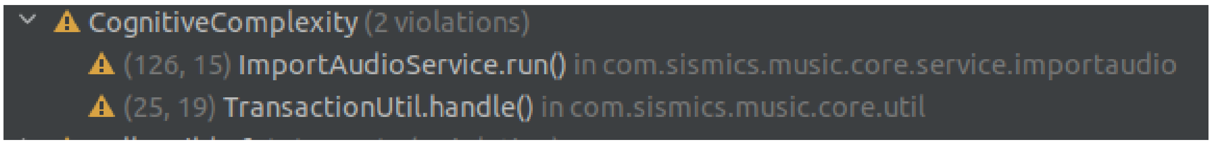
## 3B. 3: Cognitive Complexity

We have broken down complex code blocks, and spaghetti statements into simpler, easier to understand lines of code by breaking them down into separate methods. We have also removed unnecessary conditions, and made the code easier to understand overall.

We report a drop in cognitive complexity from 6 to 2 violations, as reported by PMD.

Original: 6 violations

Refactored:

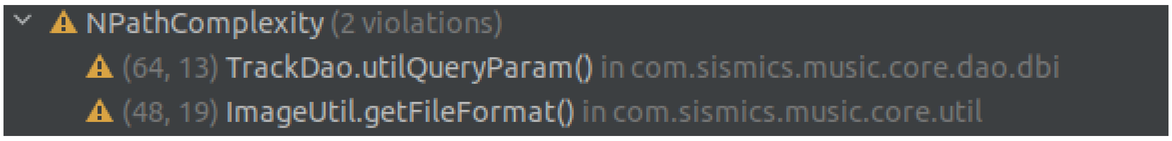

```
CognitiveComplexity (2 violations)
    (126, 15) ImportAudioService.run() in com.sismics.music.core.service.importaudio
    (25, 19) TransactionUtil.handle() in com.sismics.music.core.util
```

### 3B. 4: NPath Complexity

We were able to reduce the NPath Complexity for the methods that had violations. However, only 1 out of the 3 methods' complexity went below the threshold of 200. This was ImageUtil.MakeMosaic(), which was broken down into multiple methods.

Original: 3 violations

Refactored: 2 violations



```
⚠ NPathComplexity (2 violations)
    ⚠ (64, 13) TrackDao.utilQueryParam() in com.sismics.music.core.dao.dbi
    ⚠ (48, 19) ImageUtil.getFileFormat() in com.sismics.music.core.util
```

### 3B. 5: Cyclomatic Complexity

By removing redundant if statements and branching points, cyclomatic complexity for both, standard and modified have come down:
Original: 6 violations
Refactored: 0 violations

### 3B. 6: SignatureDeclareThrowsException

We have replaced the default exception that was thrown with specific exceptions that we wrote for each case. This was done in all the files, decreasing the number of violations to zero.

Original: 26 violations
Refactored : 0 violations

### Appendix: A Note on Coupling and Halstead Metrics

A notable point in our Tasks 2B and 3B is the omission of 'Coupling Between Objects' and 'Halstead Metrics'. We justify this choice based on the facts that:

- Halstead Metrics focus on deriving their various indicators from the number of operators and operands in the codebase. In our refactorings, we are not only deleting classes, but also adding a significant amount of code in other places. Computing the Halstead metrics here would not indicate anything meaningful.
- Coupling Between Objects won't be a very good metric to report either because any improvement we would make would be eclipsed by the tight coupling between the various Dao and Dto classes.

# 4 References

[0]: Music by Sismics

[1]: IntelliJ’s diagrams plugin

[2]: VSCode Search and File Navigation Features

[3]: ChatGPT

[4]: A. Narayanan and S. Kapoor: ‘ChatGPT is a bullshit generator. But it can still be amazingly useful’

[5]: Last.fm

[6]: Sonarqube

[7]: Designite

[8]: Designite’s IntelliJ Plugin

[9]: CS6401 Lecture Slides

[10]: Binus University COMP6106: ‘Smells’ Repository

[11]: T. Sharma: ‘A Taxonomy of Software Smells’

[12]: PMD

[13]: Checkstyle

[14]: US Patent US8146058B2