# Project 1 - CS6.401.Software Engineering

### Members


- Adhiraj Anil Deshmukh (2021121012)
- Arjun Muraleedharan (2020101099)
- Pratham Gupta (2020101080)
- Keyur Ganesh Chaudhari (2020101100)
- Sambasai Reddy Andem (2020101014)


## Specification 1: Mining the Repository

**Prior Assumptions:**

1. While the models for the mentioned classes do exist in the Music code with relevant attributes, we believe that the model classes by themselves do not sufficiently document the behavior of the system. Therefore, in order to represent the system as accurately as possible, we have chosen to include attributes and methods in the classes in the UML diagram that may not necessarily be bound to the classes in the code, but certainly do involve the classes in question and can be associated with these classes for the purpose of explanation.
2. For the sake of convenience and to avoid redundancy, we have opted not to include getter and setter methods in the classes in the UML diagram.
3. We have also compressed some classes and chains of methods into single classes/methods for abstraction, so as to make the function and behavior of the system clearer and easier to understand.

## 1. User Management

This is the subsystem that is involved in managing user accounts for Music, taking care of features such as authentication, user creation, and keeping track of albums and channels associated with specific users.

**Structure:** The following are the classes associated with this subsystem.

- `User`**:** Represents a user in the system.
- `AuthenticationToken`**:** Stores details of authentication tokens used to keep users logged into Music.
- `UserTrack`**:** Represents a user's relationship with a track. Users can listen to tracks on Music and like/unlike them.
- `UserAlbum`**:** Represents a user's relationship with an album. Users can rate a particular score for albums.

`UserTrack` and `UserAlbum` would come under user management since their specific purpose is to represent how the user would interact with the rest of the system. That is the reason `Playlist` has not been included even though each user can create and manage playlists, since `Playlist` represents an independent entity and is not built on some form of interaction.

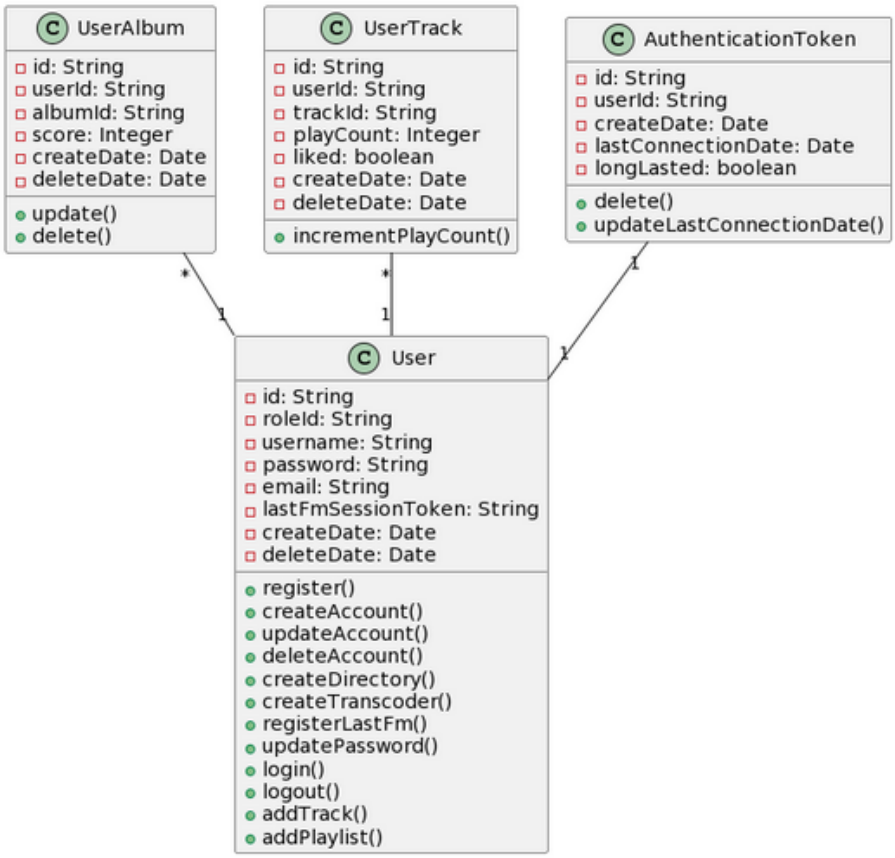
**Functionality:** This subsystem has the following features.

1. User creation, updation, deletion, and authentication
2. Maintaining records of user-album and user-track relationships

**Behavior:** The behavior of the system would be explained in terms of the above features.

1. First, the user (assuming they have an account) logs in to the system. The process of authentication is performed via the `login()` function. It would verify the username and password of the user and create an authentication token on the server, which is represented by the `AuthenticationToken` class. The authentication token would include a reference to the user, which helps in checking privileges and obtaining user details if relevant. For logging out, the authentication token would be deleted.
2. For user creation, updation, and deletion, the privileges of the user have to be that of an admin. For the sake of abstraction, we have opted to mention the relevant classes and functioning of the administrative system in a separate section. Firstly, the subsystem checks if the role of the user is "admin" via the `roleId` attribute of the user and that they have the required privileges to create/update/delete a user. Then, a user object is created with the supplied details, or in the case of updation and deletion, the user is retrieved and updated/deleted.
3. Whenever, albums and tracks are added, listened to, liked/unliked or scored by the user, records of `UserTrack` and `UserAlbum` would be created to store the relationship between the user and these tracks/albums.

Bearing the above points in mind, the following is our proposed UML diagram for the subsystem.




`AuthenticationToken` would have a one-one relationship with `User`, since a user can only have one authentication token stored at a particular time while logged in. However, since a user can like/unlike or listen to multiple tracks and create multiple albums, `UserAlbum` and `UserTrack` have many-one relationships with `User`. All of the relationships in this system are simple associations.

## 2. Administrator Features

This is the subsystem that handles the administrative features made available to certain users via their roles and privileges in the system. The system follows a role-based access control system, allowing for more security.

**Structure:** The following are the classes associated with this subsystem.

- `User`**:** Represents a user in the system.
- `Role`**:** Represents a particular role in the system. While the roles as we understand it currently are “user” and “admin”, having a dedicated class for roles opens up the possibility of adding differing degrees of role-based access control (such as allowing some advanced features for a certain kind of user, but not others).
- `Privilege`**:** Represents a particular privilege (like having admin permissions or having privileges to change passwords). Allows for greater flexibility in the mentioned role-based access control system as you can choose which roles get which privileges.

- `RolePrivilege`: Links roles and privileges.
- `Directory`: Represents a local directory where music is stored.
- `Transcoder`: Represents a transcoder entity.

The `Directory` and `Transcoder` classes do not directly play a role in the administrative system but are present merely because their features are controlled by the admins.

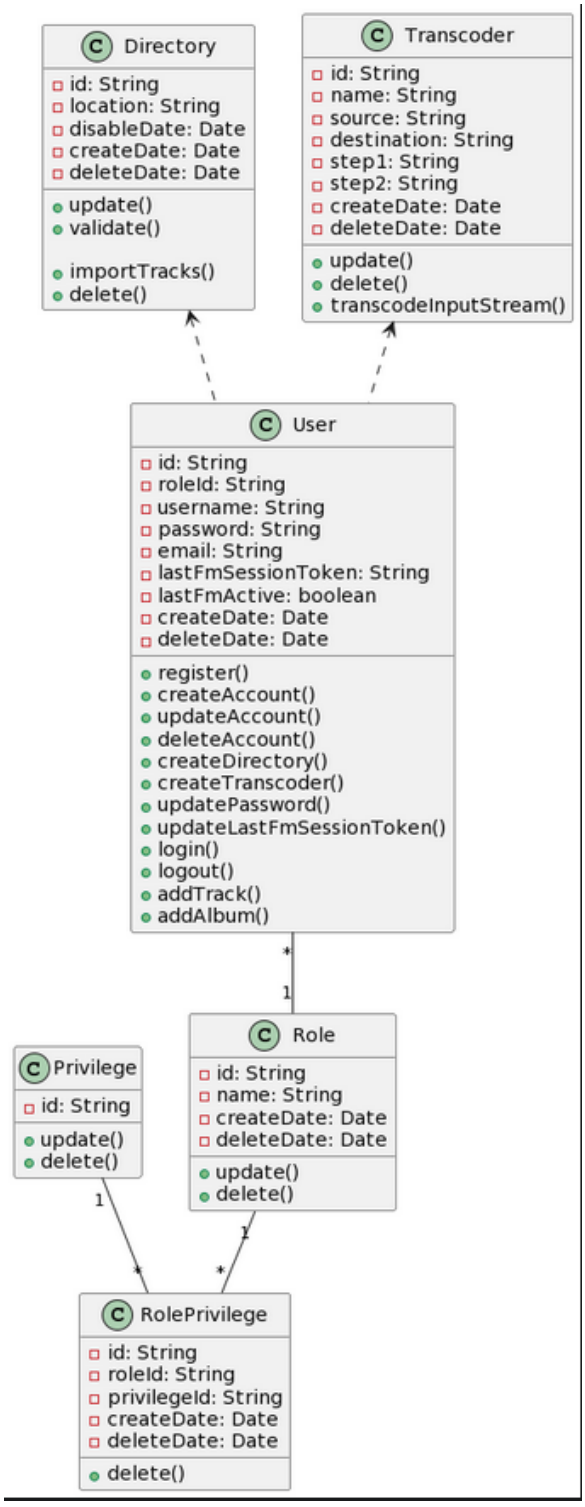
**Functionality:** The administrative subsystem permits certain users, called admins, to exercise certain privileges not available to other users.

1. They can create, update and delete user accounts.
2. They can add new directories, delete them or change the local directory to which music is stored.
3. They can add, delete or update transcoders.

**Behavior:** The behavior of the system would be explained in terms of the above features.

1. The details regarding user creation, updation, and deletion have been mentioned in the User Management section. However, it does not mention how privilege checking is done. In order to check if a user is an admin, one check if the user's role (via `roleId`) has the "ADMIN" privilege (via the `Privilege` class). The corresponding privileges for each role are obtained through the `RolePrivilege` class. The features here and the relevant methods in the `User` class can only be accessed if the admin privileges are granted to the user.
2. The creation, updation, and deletion of directories and transcoders occur in a very similar manner to that of users. First, the privileges of the user are checked, and if the user is an admin, the supplied details are validated (in the case of creation and updation). For directories, validation checks whether the supplied path is actually a valid and writable local path. The requested action is performed if the user has the required privileges. Revalidation of the directory can be done again (in the frontend via the "Rescan" option), hence the presence of a separate `validate()` method for the directory.

Bearing the above points in mind, the following is our proposed UML diagram for the subsystem.




A user has a single role, but a single role could have multiple users under it, eg. There can be several admins in the system. Since users can be grouped based on their roles, `User` and `Role` have an aggregation relationship (they are still independent entities). A role can be shared by many `RolePrivilege` objects, and so can a privilege, since a role can have multiple privileges, and a single

privilege can be shared by multiple roles. The `User` class' (or specifically an admin account's) relationships with the `Directory` and `Transcoder` classes are slightly unconventional dependencies; they are of type *<<instantiate>>* since the `User` class cannot function without these classes in order to create objects of those classes. There is no conceptual or class link between them.

## 3. Last.fm Integration

This subsystem manages the Music system's integration with the Last.fm service. Music allows users to link their Last.fm profile, and sync their listening history with it in order to build a profile of the user's musical taste.

**Structure:** The following are the classes associated with this subsystem.

- `User`**:** Represents a user in the system. This class would store the Last.fm token of the user.
- `LastFmSession`**:** Represents a Last.fm session of a particular user. The session lasts for as long as the user has linked their credentials to Music, and would be destroyed when the user unlinks.
- `Track`**:** Represents a music track. Last.fm requires access to the listening history of the user and so would be related to the `Track` class.
- `UserTrack`**:** Represents a user's relationship with a track.

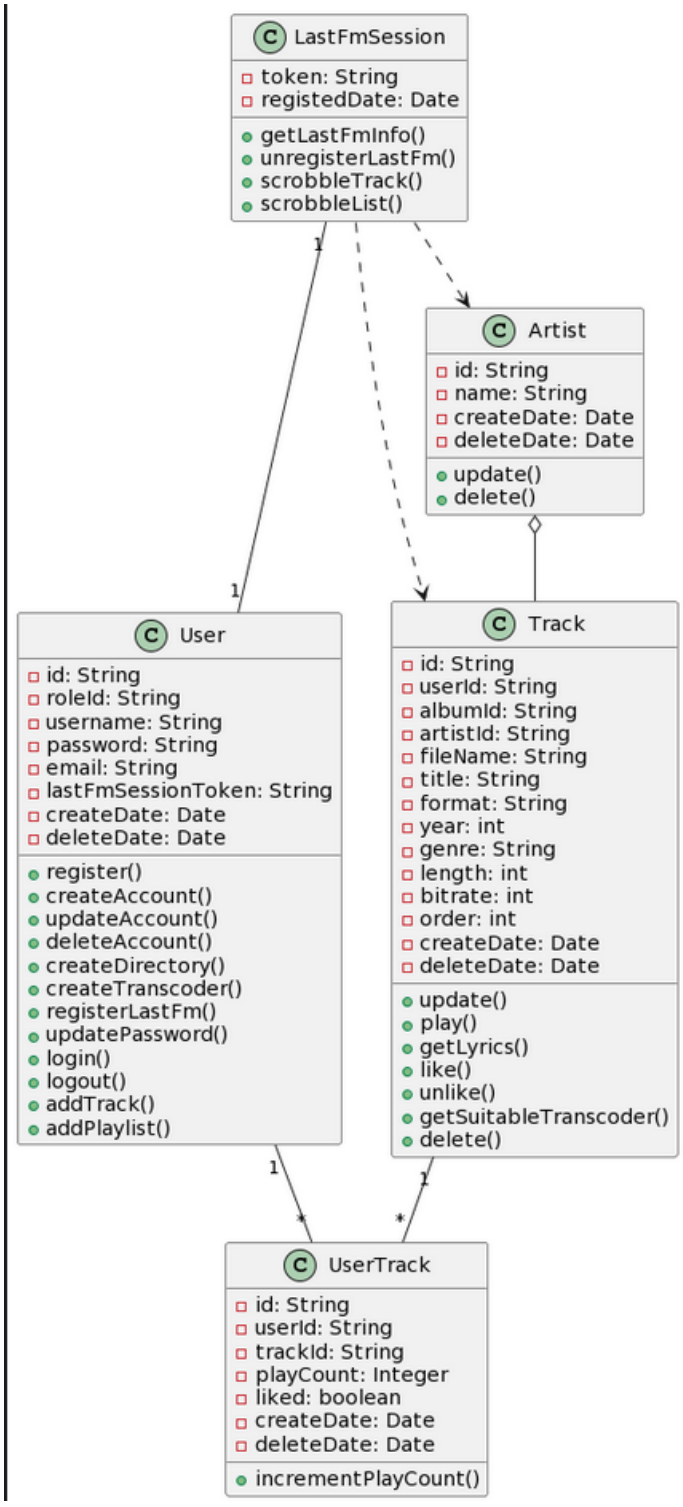
**Functionality:** The purpose of the subsystem is to do the following.

1. Enable the user to link their Last.fm account with their Music account.
2. Allow Last.fm access to users' listened-to tracks to analyze them.

**Behavior:** The behavior of the system would be explained in terms of the above features.

1. When the user enters their Last.fm credentials and submits them, a session is established with Last.fm via a token which would be stored in the `User` class. This is done via the `registerLastFm()` method. As long as this session lasts (until the user unlinks their Last.fm account from Music), the Last.fm service would be integrated with Music. The `unregisterLastFm()` method would delete this session.
2. In order to have the Last.fm service perform analysis on the tracks listened to by the user, the `scrobbleTrack()` and `scrobbleList()` (for a list of tracks) methods are present. These take in the track(s) and have Last.fm perform analysis on them. These methods require access to both the track and the artist in order to perform its analysis, hence the presence of these two classes in the subsystem.

Bearing the above points in mind, the following is our proposed UML diagram for the subsystem.




As mentioned in the User Management section, `User` would have a one-many association with `UserTrack` and so would `Track`, since one track could be interacted with by multiple users. Since `LastFmSession` uses the `Artist` and `Track` classes for scrobbling, it would have dependencies on those

classes. Since one artist can have multiple tracks, but `Artist` and `Track` are independent entities within the system, there is an aggregation relationship between them.

## 4. Library Management

This subsystem focuses on the main part of Music: playing, importing, and managing of music libraries and albums. This is what makes Music a music application.

**Structure:** The following are the classes associated with this subsystem.

- `User`**:** Represents a user in the system.
- `Track`**:** Represents a music track.
- `UserTrack`**:** Represents a user's relationship with a track.
- `Artist`**:** Represents a music artist.
- `Album`**:** Represents an album of tracks by an artist. It should be noted here that there is no specific method that creates an album. This is because an album is automatically created by the code when track information is added via the frontend and read by the backend. Therefore, it can be said that album creation is actually done via the `addTrack()` function of the User class.
- `UserAlbum`**:** Represents a user's relationship with an album. Users can rate a particular score for albums.
- `Playlist`**:** Represents a user-created music playlist.
- `PlaylistTrack`**:** This represents the relationship between a playlist and a track.
- `Directory`**:** Represents a local directory where music is stored.
- `Transcoder`**:** Represents a transcoder entity.
- `ImportResource`**:** Contains functionality for importing music from local directories or external resources.
- `PlayerResource`**:** Contains functionality for managing music that one is currently playing.
- `Queue`**:** Represents a music queue for a particular user.

**Functionality:** The purpose of the subsystem is to do the following.

1. To play, pause and stop a music stream
2. To add, update and delete track, album, artist, and playlist information
3. To import music tracks from local directories or external sources
4. To store imported music in admin-added directories

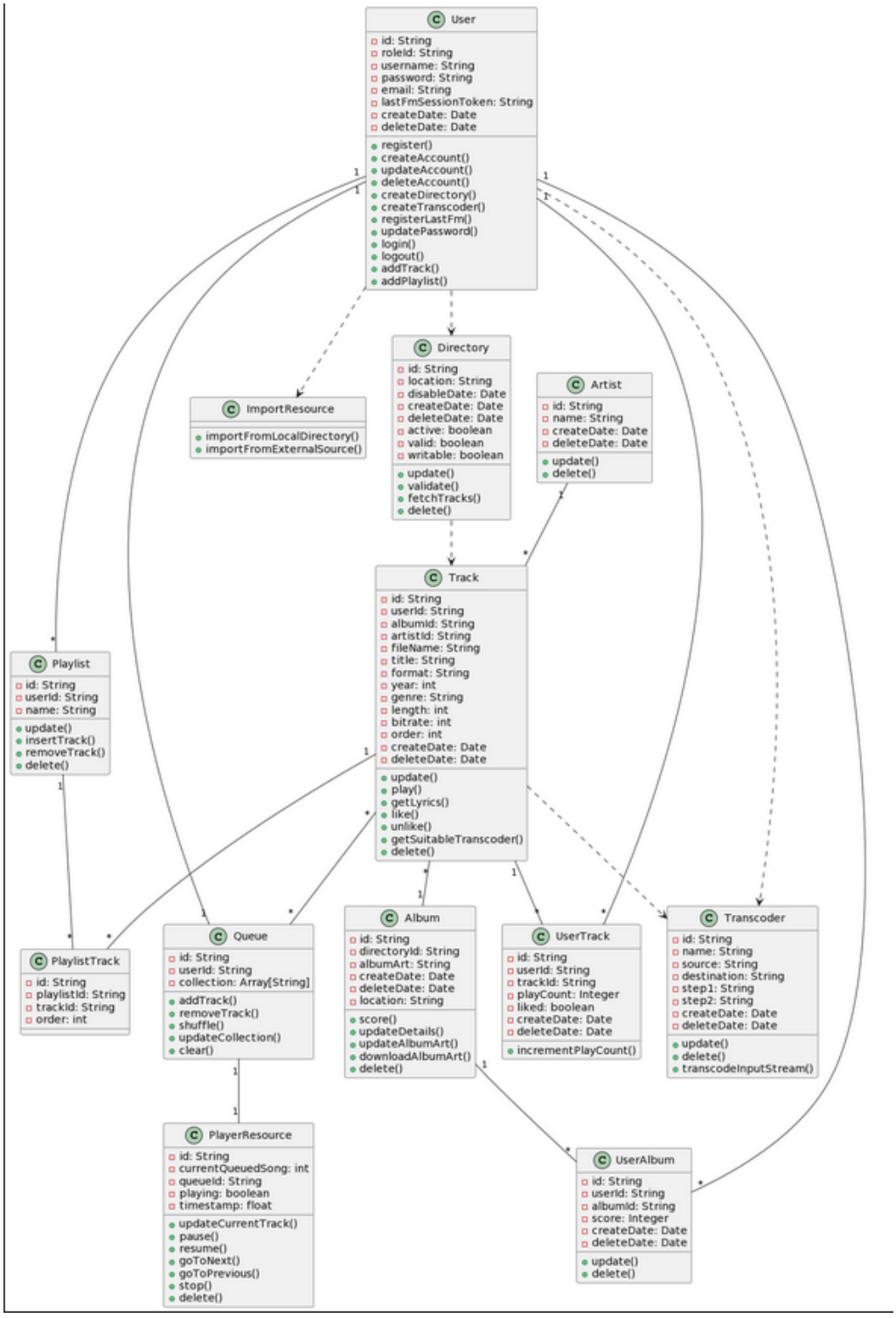
**Behavior:** The behavior of the system would be explained in terms of the above features.

1. Initially, when a track is played, the music file stored in the local directory would be retrieved (with the help of `fetchTracks()` in the `Directory` class) and converted to a media stream by a suitable

transcoder (found using `getSuitableTranscoder()`. The track would then be added to a created `Queue` object. A player controlled by `PlayerResource` would then be able to control this media stream via dedicated functions, and resume, stop or pause the stream, or skip ahead or back. As the user chooses to add more tracks or play a playlist/album instead, the queue would get updated accordingly, and can be shuffled or modified via `Queue`.

2. The user would be able to add/update/delete tracks, albums, artists and playlists via dedicated methods in the `User` class. Users can listen to or like/unlike tracks (the status of which is recorded by `UserTrack`) and rate a score for albums (recorded by `UserAlbum`). Moreover, metadata of tracks, playlists and albums can be updated (including album art).
3. When the user adds a track, it makes of `ImportResource`, which contains dedicated methods to import music from a local directory or an external source (where the YouTube URL would be supplied). This would be done via `addTrack()`, which would then create a record of the track accordingly. The imported track would be added to a dedicated directory via the `addTrack()` method of the `Directory` class.

Based on the above, the following is our proposed UML diagram of this subsystem.



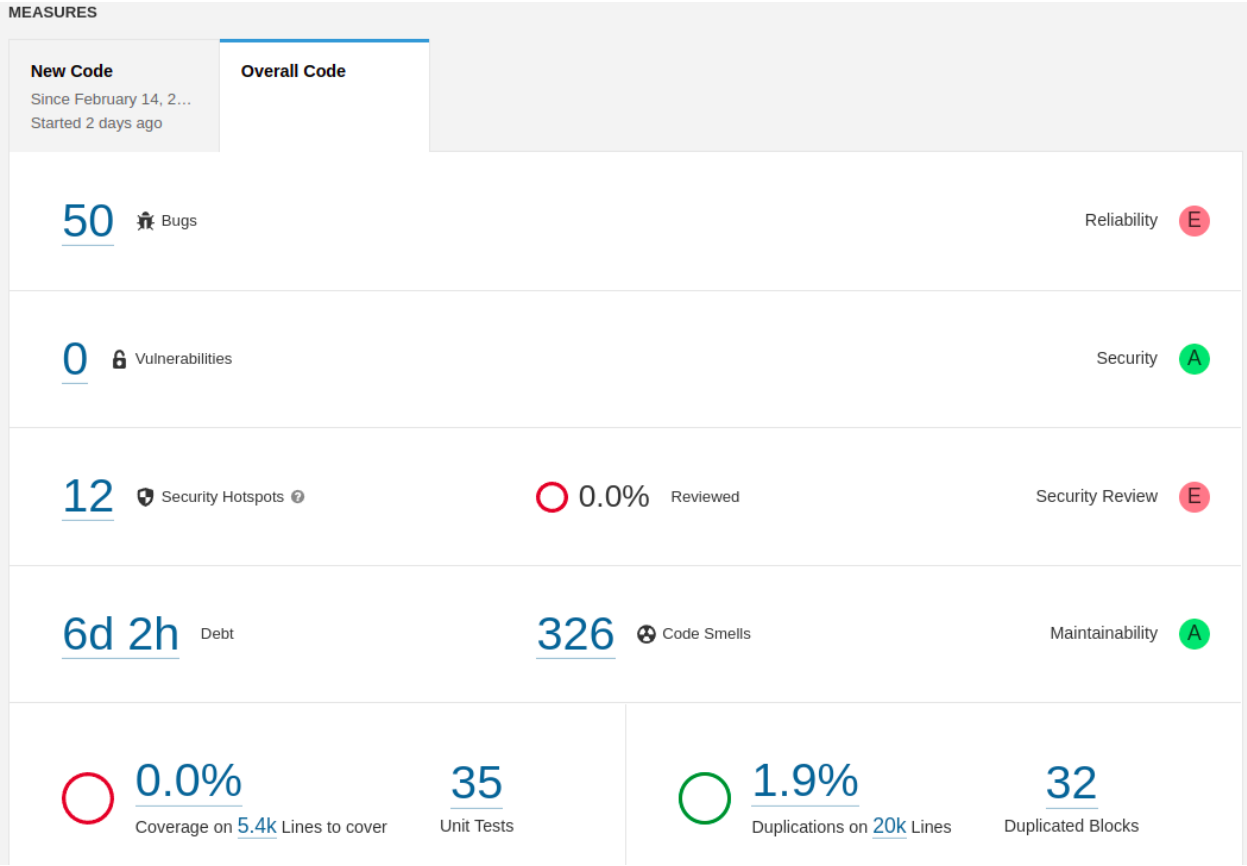

The relationships between `User`, `Album`, and `UserAlbum` have already been explained. The relationships between `Playlist`, `Track`, and `PlaylistTrack` would be similar to them. A track is a part of at most one album, hence a one-many relationship between `Track` and `Album`. A user can have multiple playlists, so there would be a one-many relationship between `User` and `Playlist`. The

dependency relationships of `Transcoder` and `Directory` have been explained in the Administrator Features section. Since one artist can have multiple tracks, but `Artist` and `Track` are independent entities within the system, there is an aggregation relationship between them. The same goes for `Track` and `Album`. However, it does not make sense for a playlist to exist without its user, so `Playlist` and `User` would have a composite relationship. A queue contains multiple tracks, and the same track can be present in multiple queues, so they have a many-many relationship. One queue is linked to one user, so they have a one-one relationship, and similarly for players and queues. The `play()` method would depend on a transcoder to obtain the media stream for the track, so `Track` depends on `Transcoder`. `User` would depend on `ImportResource` to import the track. Moreover, `Directory` would depend on `Track`, since it has a method to fetch tracks from the specified location, and this would require access to the `Track` class.

# Specification 2: Analysis

## 2a. Design Smells

### Sonarqube Analysis:




## 1. Duplicate Abstraction

Classes `DirectoryCreatedAsyncEvent` and `DirectoryDeletedAsyncEvent` have the same definitions but have been utilized differently by the `DirectoryResource` and the `DirectoryCreatedAsyncListener`

## 2. Missing Hierarchy

The `DirectoryCreatedAsyncListener` and `DirectoryDeletedAsyncListener` classes contain very similar codes for processing directory events. We can simplify this code by extracting the common code into a new method and calling this method from both classes.

## 3. Insufficient Modularization

Certain functionality has not been included as class methods which can be refactored to make the code more modular.

## 4. Missing Abstraction

Missing abstraction is when clumps of data or encoded strings are used instead of creating an abstraction. In the `getQueryParams()` method in trackDao.java, the addition of search criteria has been done in a manner that utilizes the same fragment of code over and over again, differing only with respect to the parameter supplied. If we want to add a new parameter to the search criteria, then we would have to copy it again. This would pose an inconvenience if the code fragment were bigger.

```
criteriaList.add("a.deletedate is null");

if (criteria.getId() != null) {
    criteriaList.add("a.id = :id");
    parameterMap.put("id", criteria.getId());
}

if (criteria.getNameLike() != null) {
    criteriaList.add("lower(a.name) like lower(:nameLike)");
    parameterMap.put("nameLike", "%" + criteria.getNameLike() + "%");
}
```

We can remove this smell by separating the logic for the addition of search criteria and the logic that supplies the parameters to this code. The same idea can be applied to TrackDao.java, which also contains the same design smell.

## 5. Unutilized Abstraction

This smell arises when an abstraction is left unused (either not directly used or not reachable). This can lead to unnecessary complexity and reduced maintainability. In the file TrackDtoMapper.java, for instance, the abstractions offered by the methods of the class `PlaylistTrackMapper` are not utilized anywhere, even though an object of the class is being declared. If the provided abstraction is not being utilized, then it serves no purpose and is redundant.

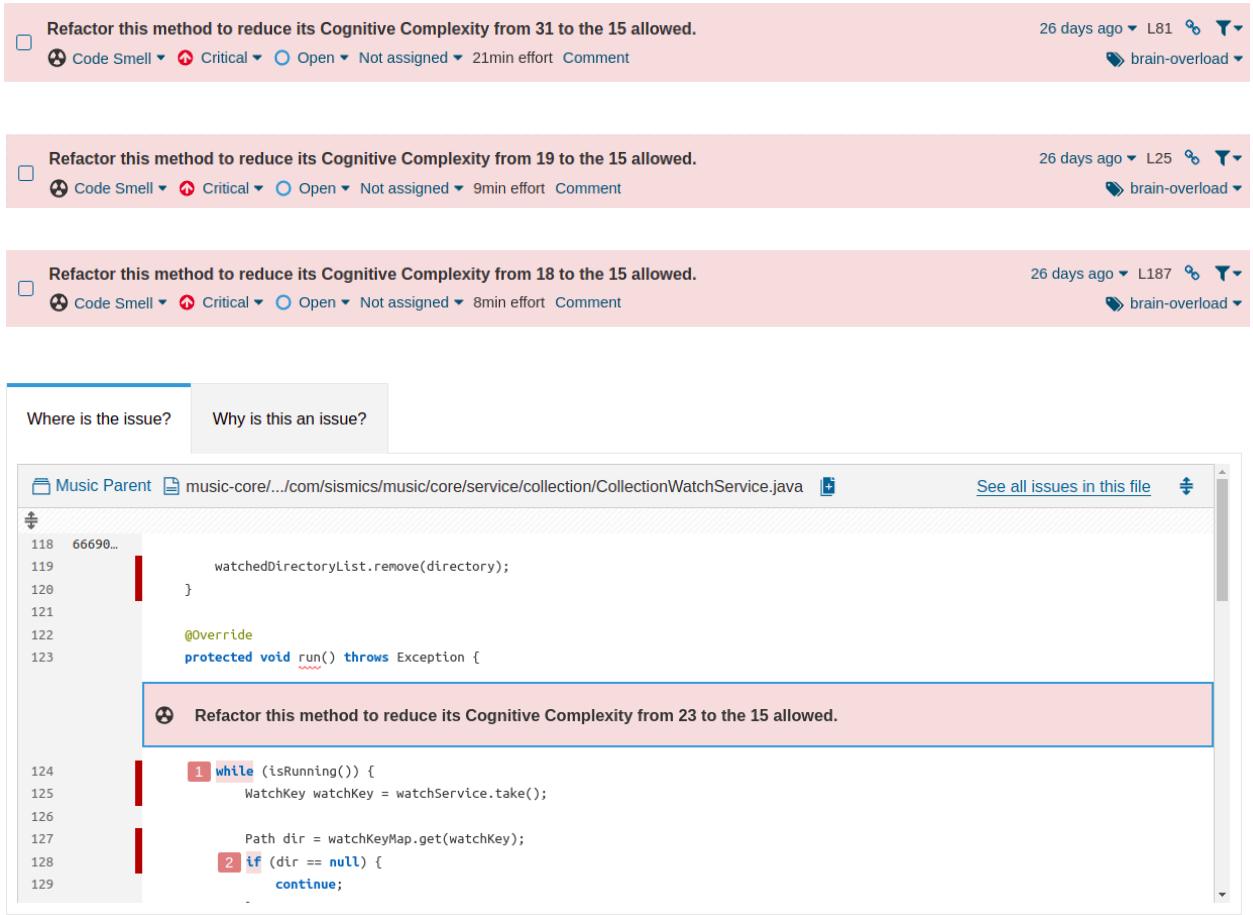
## 6. Circular Dependency

Circular dependency occurs when two or more components depend on each other directly or indirectly. In the following files and classes, there are circular dependencies between two packages:

1. The com.sismics.music.core.event.async package depends on the com.sismics.music.core.service.lastfm package through the `LastFmUpdateLovedTrackAsyncEvent` and `LastFmUpdateTrackPlayCountAsyncEvent` classes.
2. The com.sismics.music.core.service.lastfm package depends on the com.sismics.music.core.dao.dbi package through the `ArtistDao`, `TrackDao`, `UserDao`, and `UserTrackDao` classes.
3. The com.sismics.music.core.dao.dbi package depends on the com.sismics.music.core.service.lastfm package through the `LastFmService` class.

So, the com.sismics.music.core.event.async package depends on com.sismics.music.core.service.lastfm package which in turn depends on com.sismics.music.core.dao.dbi package and then back to com.sismics.music.core.event.async package, forming a circular dependency.

This can lead to problems in the application's architecture, making it difficult to understand, maintain, and test. It can also make it hard to modify or add new features to the codebase.



## 2b. Code Metrics

We used the IntelliJ plugin MetricTree to compute the following code metrics for the project:

1. **Halstead Metrics**
   The Halstead complexity metric is used to measure the complexity of a software program without running the program itself. This metric is a static testing method where measurable software properties are identified and evaluated. The source code is analyzed and broken down into a sequence of tokens. The tokens are then classified and counted as operators or operands. The operators and operands are classified and counted as follows:
   *Number of distinct operators: n1*
   *Number of distinct operands: n2*
   *Total number of operators: N1*
   *Total number of operands: N2*

   a. *Halstead Vocabulary: n = n1 + n2 =* ***7352***
   b. *Halstead Length: N = N1 + N2 =* ***18740***
   c. *Halstead Difficulty: D = (n1 / 2) * (N2 / n2) =* ***4917.0544***
   d. *Halstead Volume: V = N * log2(n) =* ***115989.9467***
   e. *Halstead Effort: E = V * D =* ***10106503.8309***
   f. *Halstead Errors: Er = ( E ^ (2/3) ) / 3000 =* ***53.003***

2. **Attribute Hiding Factor (AHF)**
   AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration.

   a. Software quality correlation: AHF is a measure of the use of the information-hiding concept supported by the encapsulation mechanism. Information hiding allows coping with complexity by turning complex components into black boxes. AHF should be used as much as possible. Ideally, all attributes would be hidden, thus being only accessed by the corresponding class methods. Very low values of AHF should trigger the designers' attention. In general, as AHF increases, the complexity of the program decreases.

   - Allowed Range: **[67%, 100%]**
   - Calculated metric: **90.5%**

3. **Attribute Inheritance Factor (AIF)**
   The Attribute Inheritance Factor is defined as a quotient between the sum of inherited attributes in all classes of the system under consideration and the total number of available attributes (locally defined plus inherited) for all classes.

   a. Software quality correlation: At first sight, we might be tempted to think that inheritance should be used extensively. However, the composition of several inheritance relations builds

a directed acyclic graph (inheritance hierarchy tree), whose depth and width make understandability and testability fade away quickly.

- Allowed Range: **[37.4%, 75.7%]**
- Calculated metric: **15.8%**

4. **Coupling Factor (CF)**
   The coupling Factor (CF) measures the coupling between classes excluding coupling due to inheritance. It is the ratio between the number of actually coupled pairs of classes in scope (e.g., package) and the possible number of coupled pairs of classes.

   a. Software quality correlation: The client-supplier relation, represented by Cc ⇒ Cs, means that Cc (client class) contains at least one non-inheritance reference to a feature (method or attribute) of class Cs (supplier class). The CF numerator represents the actual number of coupling not imputable to inheritance. It is desirable that classes communicate with as few other classes as possible, and that they exchange as little information as possible. Coupling relations increase complexity, reduce encapsulation and potential reuse, and limit understandability and maintainability. Also, coupling in software systems has a strong negative impact on software quality, and therefore should be kept to a minimum during the design phase. However, for a given application, classes must cooperate to deliver some kind of functionality. Therefore, CF is expected to be lower bound.

   b. Allowed Range: **[0%, 24.3%]**

   c. Calculated metric: **3.12%**

5. **Method Hiding Factor (MHF)**
   MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system. The invisibility of a method is the percentage of the total classes from which this method is not visible.

   a. Software quality correlation: The number of visible methods is a measure of the class functionality. Increasing the overall functionality will reduce MHF. In order to implement this functionality, a top-down approach must be adopted, where the abstract interface (visible methods) should only be the tip of the iceberg. In other words, the implementation of the class interface should be a step-wise decomposition process, where more and more details are added. This decomposition will use hidden methods, thus obtaining the above-mentioned information-hiding benefits and favoring an MHF increase. This apparent contradiction is reconciled by considering MHF to have values within an interval. A very low MHF value would indicate an insufficiently abstracted implementation. Conversely, a high MHF value would indicate very little functionality. The best thing for MHF is to be within an interval.

   b. Allowed Range: **[9.5%, 36.9%]**

   c. Calculated metric: **8.36%**

6. **Method Inheritance Factor (MIF)**
   The Method Inheritance Factor is defined as a quotient between the sum of inherited methods in all classes of the system under consideration and the total number of available methods (locally defined and including those inherited) for all classes.
   a. Software quality correlation: At first sight, we might be tempted to think that inheritance should be used extensively. However, the composition of several inheritance relations builds a directed acyclic graph (inheritance hierarchy tree), whose depth and width make understandability and testability fade away quickly.
   b. Allowed range: **[60.9%, 84.4%]**
   c. Calculated metric: **11.35%**
7. **Polymorphism Factor (PF)**
   The Polymorphism Factor is defined as the quotient between the actual number of different possible polymorphic situations, and the maximum number of possible distinct polymorphic situations for a given class.
   a. Software quality correlation: The PF metric represents the actual number of possible different polymorphic situations. A given message sent to class can be bound, statistically or dynamically, to a named method implementation which may have as many shapes as the number of times this same method is overridden (in class descendants). Polymorphism arises from inheritance. Binding (usually at run time) a common message call to one of several classes (in the same hierarchy) is supposed to reduce complexity and to allow refinement of the class hierarchy without side effects. On the other hand, to debug such a hierarchy, by tracing the control flow, this same polymorphism would make the job harder. Therefore, polymorphism ought to be bounded within a certain range.
   b. Allowed Range: **[1.7%, 15.1%]**
   c. Calculated metric: **82.55%**
8. **Statistics**
   a. Lines of Code = 11387
   b. Non-commenting Source statements = 4400
   c. Abstract Classes = 5
   d. Concrete Classes = 176
   e. Interfaces = 1
   f. Static Classes = 2

# Specification 3: Refactoring

## 3a. Mitigating Design Smells

1. **Duplicate Abstraction**: A single class `DirectoryAsyncEvent` has been created to substitute two duplicate classes `DirectoryCreatedAsyncEvent` and `DirectoryDeletedAsyncEvent`.

   Similarly

   1. `LastFmLovedTrackAsyncEvent`
   2. `PlayEvent`
   3. `TrackAsyncEvent`

   classes have been constructed to substitute duplicate classes

2. **Missing Hierarchy**: An abstract class `DirectoryAsyncListener` has been created that has been extended to `DirectoryAsyncListener` and `DirectoryAsyncListener` since the two classes previously had very similar implementations. A Consumer functional interface has been implemented to enable this.

   Similarly

   1. `LastFmLovedTrackAsyncListener`
   2. `PlayAsyncListener`
   3. `TrackAsyncListener`

   abstract classes have been created to hold most of the repetitive information of the two children listeners using *consumer* functional interfaces.

3. **Insufficient Modularization:** Methods have been introduced in the listener classes to make the code more modular, thus enhancing metrics.

4. **Missing Abstraction:** In this refactored version, we have extracted the logic for adding search criteria to the `addSearchCriteria()` method, which takes the `ArtistCriteria`, the list of criteria, and the map of parameter values as arguments. This method is responsible for adding the search criteria to the list and the corresponding parameter values to the map. We have also extracted the logic for adding a single criterion to the `addCriteria()` method, which takes the criterion string, the criterion value, the parameter map, and the list of criteria as arguments. This method is responsible for checking if the criterion value is not null and adding the criterion to the list and the parameter value to the map if it is not null.

   By doing this, we have removed the duplication of the code for adding search criteria and made the code more readable and easier to maintain.

5. **Unutilized Abstraction:**

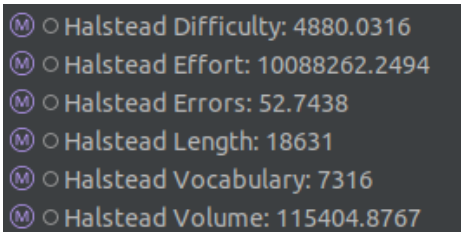
   - Identify unused abstractions: You need to identify the interfaces and abstract classes that are not being used in the codebase. You can use static analysis tools or code reviews to identify them.

- Remove unused abstractions: Once you have identified the unused abstractions, you can remove them from the codebase. Removing unused abstractions will simplify the codebase and make it easier to understand and maintain.
- Refactor code that uses unused abstractions: In some cases, the code that uses unused abstractions may need to be refactored. For example, if you remove an interface that is used by a class, you may need to modify the class to remove the references to the interface.

## 3b. Code Metrics on Refactoring

| METRIC | ORIGINAL | R1: Duplicate abstraction in events | R2: Missing Hierarchy in listeners | R3: Insufficient Modularization | R4 and R5 |
|---|---|---|---|---|---|
| AHF | 90.5 | 90.44 | 90.22 | 90.40 | 90.39 |
| AIF | 15.8 | 15.9 | 16.19 | 19.03 | 19.04 |
| CF | 3.12 | 3.12 | 3.11 | 3.12 | 3.14 |
| MHF | 8.36 | 8.67 | 9.31 | 10.45 | 10.64 |
| MIF | 11.35 | 11.55 | 12.02 | 12.40 | 12.03 |
| PF | 82.55 | 80.79 | 75.79 | 75.79 | 75.15 |

Finally, the Halstead Metrics have also been reduced from the original after the refactoring process as follows:




We see that our refactoring is able to enhance code metrics by a significant amount and push metrics like MHF (that was previously not accepted) into the accepted range.