Unit Project 1 (Team 12)

Task1

Base Classes used

BaseDao.java

This is a Java class that defines a basic Data Access Object (DAO) that provides methods for searching a collection of objects of a given type T by specified criteria C. The DAO uses a **PaginatedList data structure** to handle the results of the search, which is updated by side effects.

The BaseDao class contains four methods for searching the collection of objects, findByCriteria, findByCriteria with three arguments, findByCriteria with one argument, and findFirstByCriteria. All these methods accept an instance of the search criteria and may also accept an instance of SortCriteria and FilterCriteria.

The findByCriteria methods return a list of objects of type T m8atching the search criteria, while the findFirstByCriteria method returns only the first result that matches the search criteria. These methods utilize the **PaginatedLists** utility class to execute paginated queries using the search criteria, **SortCriteria**, and **FilterCriteria** parameters.

This is an abstract class, meaning that it defines some basic behavior for a DAO, but cannot be directly instantiated. Concrete DAOs that extend this BaseDao must implement the getQueryParam method, which is used to create the **QueryParam** instance that is required by the PaginatedLists utility class. The getQueryParam method accepts an instance of the search criteria and an instance of FilterCriteria, and returns a QueryParam instance that can be used to execute the search.

Methods:

1. **findByCriteria(PaginatedList<T> list, C criteria, SortCriteria sortCriteria, FilterCriteria filterCriteria):** This method takes a PaginatedList instance, an instance of the search criteria, an instance of SortCriteria, and an instance of FilterCriteria. It executes a paginated search using these parameters and updates the PaginatedList with the results of the search. The method does not return anything, but the results are available through the PaginatedList.
2. **findByCriteria(C criteria, SortCriteria sortCriteria, FilterCriteria filterCriteria):** This method is similar to the previous method, but it returns a list of objects of type T that match the search criteria, sorted according to the SortCriteria parameter. The method creates a QueryParam instance using the search criteria and filter criteria, and passes it to the PaginatedLists.executeQuery method, which executes the search and returns a list of objects.
3. **findByCriteria(C criteria):** This method is a simplified version of the previous method that only takes an instance of the search criteria. It calls the findByCriteria method with the search criteria and null for the SortCriteria and FilterCriteria parameters, which returns an unsorted list of objects that match the search criteria.
4. **findFirstByCriteria(C criteria):** This method is similar to the findByCriteria method, but it only returns the first object that matches the search criteria. It calls the PaginatedLists.executeQuery method with a QueryParam instance created using the search criteria, and returns the first object in the list of results.

In summary, these methods provide various ways to search a collection of objects of type T by a specified set of criteria C, and return the results in a paginated or unpaginated format. The methods utilize the PaginatedLists utility class to handle pagination and execute the search.

Paginated List

The PaginatedList class is a generic class which is used by the BaseDao class for pagination of query results. It contains four fields: limit, offset, resultCount, and resultList.

- limit represents the maximum number of results that should be returned in a single page.
- offset represents the index of the first result that should be included in the current page.
- resultCount represents the total number of records that match the query.
- resultList represents the list of records for the current page.

It also provides getter and setter methods for each field.

The PaginatedList class is constructed with two arguments: pageSize and offset. pageSize specifies the maximum number of results to be returned per page, and offset specifies the starting index of the first result to be returned.

Attributes:

- limit (int): Size of a page.
- offset (int): Offset of the page (in number of records).
- resultCount (int): Total number of records.
- resultList (List<T>): List of records of the current page.

Methods:

- PaginatedList(int pageSize, int offset): Constructor that creates a new PaginatedList with the given page size and offset.
- getResultCount(): Getter method for the resultCount attribute.
- setResultCount(int resultCount): Setter method for the resultCount attribute.
- getResultList(): Getter method for the resultList attribute.
- setResultList(List<T> resultList): Setter method for the resultList attribute.
- getLimit(): Getter method for the limit attribute.
- getOffset(): Getter method for the offset attribute.

The PaginatedList class is used to represent a page of query results. It contains information about the size of the page, the offset of the page, the total number of records that match the query, and the list of records for the current page. The constructor takes a page size and an offset as arguments, and the getter and setter methods are used to access and modify the various attributes.

<u>Paginated Lists</u>

This Java class contains a set of utility methods for creating and executing paginated database queries. It is part of a larger music-related Java package.

Attributes:

- DEFAULT_PAGE_SIZE (int): The default number of results to return per page when paginating results.
- MAX_PAGE_SIZE (int): The maximum number of results to return per page when paginating results.

Methods:

- create(Integer pageSize, Integer offset): creates a new PaginatedList object with the specified page size and offset. If pageSize is null, it will default to the DEFAULT_PAGE_SIZE attribute. If offset is null, it will default to 0. The method also checks that pageSize is not greater than MAX_PAGE_SIZE and is not 0 (in which case it defaults to 1).
- create(): creates a new PaginatedList object with default values (DEFAULT_PAGE_SIZE and an offset of 0).
- executeQuery(QueryParam queryParam): executes a non-paginated database query using the specified QueryParam object, and returns a list of the results.
- executeCountQuery(PaginatedList<E> paginatedList, QueryParam queryParam): executes a native count(*) request to count the number of results, and sets the resultCount attribute of the specified PaginatedList object accordingly.
- getNativeCountQuery(QueryParam queryParam): returns a native query to count the number of records using the specified QueryParam object. The initial query must be of the form "select xx from yy".
- executeResultQuery(PaginatedList<E> paginatedList, QueryParam queryParam): executes a paginated database query using the specified PaginatedList and QueryParam objects, and sets the resultList attribute of the PaginatedList object with the results of the current page.
- getLimitQuery(String queryString, int limit, int offset): creates a native query that limits the number of results returned based on the specified queryString, limit, and offset.
- execute(PaginatedList<E> paginatedList, QueryParam queryParam, SortCriteria sortCriteria): executes a paginated database query using the specified PaginatedList, QueryParam, and SortCriteria objects. The

method calls executeCountQuery() and executeResultQuery() internally, and sets the resultCount and resultList attributes of the PaginatedList object accordingly.

SortCriteria

This Java class represents a sort criteria for a database query. It has three attributes:

1. column: An integer representing the index of the column to sort, where the first column is at index 0.
2. asc: A boolean value indicating whether to sort in increasing order (true) or decreasing order (false).
3. sortQuery: A string representing the query to use for sorting.

The class has two constructors:

1. SortCriteria(String sortQuery): Creates a new SortCriteria instance with the given sort query.
2. SortCriteria(Integer column, Boolean asc): Creates a new SortCriteria instance with the given column and sort direction. Either of these arguments may be null, in which case the corresponding attribute will be left unset.

The class also provides two methods for accessing its attributes:

1. getSortQuery(): Returns the sort query string.
2. getColumn(): Returns the column index to sort on.
3. isAsc(): Returns a boolean indicating whether to sort in increasing order or not.

FilterCriteria

The FilterCriteria class represents the filtering criteria of a query. It contains a list of FilterColumn objects, which define the column to filter and the value to filter on.

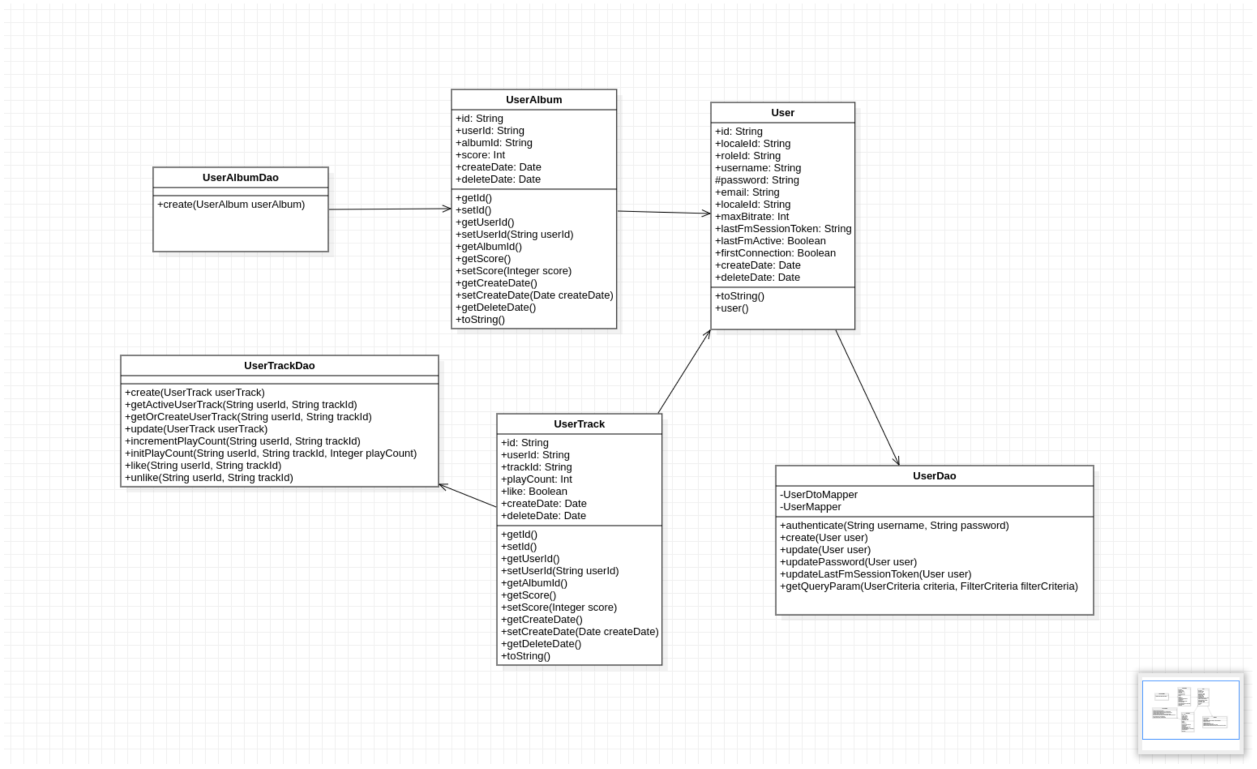
Attributes:

- filterColumnList: a list of FilterColumn objects.

Methods:

- FilterCriteria(List<FilterColumn> filterColumnList): constructor that takes a list of FilterColumn objects and initializes the filterColumnList attribute with it.
- getFilterColumnList(): returns the list of FilterColumn objects.

1. <u>**USER MANAGEMENT:**</u>



<u>User.java</u>

This is a User entity class for a music management system. It represents a user who can interact with the system.

The attributes of this class are:

- id: the unique ID of the user
- localeId: the locale ID of the user
- roleId: the ID of the user's role
- username: the username of the user
- password: the password of the user
- email: the email address of the user
- maxBitrate: the maximum bitrate in kbps the user can use, null if unlimited
- lastFmSessionToken: the session token on Last.fm of the user
- lastFmActive: a boolean that indicates whether scrobbling on Last.fm is active for the user
- firstConnection: a boolean that indicates whether the user has dismissed the first connection screen
- createDate: the creation date of the user
- deleteDate: the deletion date of the user

The methods of this class are:

- User(): the default constructor for the User class
- User(String id, String localeId, String roleId, String username, String password, String email, Integer maxBitrate, String lastFmSessionToken, boolean lastFmActive, boolean firstConnection, Date createDate, Date deleteDate): a constructor that initializes all the attributes of the User class
- Getter and setter methods for all the attributes of the class: getId(), setId(String id), getLocaleId(), setLocaleId(String localeId), getRoleId(), setRoleId(String roleId), getUsername(), setUsername(String username), getPassword(), setPassword(String password), getEmail(), setEmail(String email), getMaxBitrate(), setMaxBitrate(Integer maxBitrate), getLastFmSessionToken(), setLastFmSessionToken(String lastFmSessionToken), isLastFmActive(), setLastFmActive(boolean lastFmActive), isFirstConnection(), and setFirstConnection(boolean firstConnection).

UserAlbum.java

This is a Java class called UserAlbum that represents user information on an album entity. It contains the following attributes:

- id: a String representing the user information on an album ID.
- userId: a String representing the user ID.
- albumId: a String representing the album ID.
- score: an Integer representing the user score for this album.
- createDate: a Date representing the creation date.
- deleteDate: a Date representing the deletion date.

The class also includes the following methods:

- UserAlbum(): a constructor that initializes all attributes to null.
- UserAlbum(String id, String userId, String albumId, Integer score, Date createDate, Date deleteDate): a constructor that initializes all attributes to the values provided as arguments.
- getId(): a getter method that returns the id attribute.
- setId(String id): a setter method that sets the id attribute to the value provided as an argument.
- getUserId(): a getter method that returns the userId attribute.
- setUserId(String userId): a setter method that sets the userId attribute to the value provided as an argument.
- getAlbumId(): a getter method that returns the albumId attribute.
- setAlbumId(String albumId): a setter method that sets the albumId attribute to the value provided as an argument.
- getScore(): a getter method that returns the score attribute.
- setScore(Integer score): a setter method that sets the score attribute to the value provided as an argument.
- getCreateDate(): a getter method that returns the createDate attribute.
- setCreateDate(Date createDate): a setter method that sets the createDate attribute to the value provided as an argument.
- getDeleteDate(): a getter method that returns the deleteDate attribute.
- setDeleteDate(Date deleteDate): a setter method that sets the deleteDate attribute to the value provided as an argument.
- toString(): a method that returns a String representation of the object, using the id, userId, and albumId attributes.

UserTrack.java

The UserTrack class represents user information on a track entity, storing information such as the user ID, track ID, play count, like status, creation and deletion dates.

Attributes:

- id (String): User information on a track ID.
- userId (String): User ID.
- trackId (String): Track ID.
- playCount (Integer): Number of times this track was played.
- like (boolean): True if this track is a like.
- createDate (Date): Creation date.
- deleteDate (Date): Deletion date.

Constructor:

- UserTrack() : Empty constructor.
- UserTrack(String id, String userId, String trackId, Integer playCount, Boolean like, Date createDate, Date deleteDate): Constructor that initializes all the attributes with the given values.

Methods:

- Getter and Setter methods for all the attributes.
- toString() method that returns a string representation of the object.

UserDao.java

The UserDao class is a data access object (DAO) that provides methods for accessing and manipulating user data in a database. The class is part of a larger system for managing music and is designed to work with the database interface library (DBI).

The UserDao class extends a base DAO class called BaseDao that handles basic CRUD (create, read, update, and delete) operations for the UserDto data transfer object (DTO). The UserDto class is a simple container for user data that is used to transfer data between the database and the application.

The UserDao class provides several methods for working with user data, such as creating a new user, authenticating a user, updating user information, and retrieving user data based on certain criteria. The class also contains methods for hashing user passwords and updating the Last.fm session token for a user.

Overall, the UserDao class provides an interface between the application and the underlying database, allowing the application to store, retrieve, and manipulate user data as needed.

UserDao is used to access the User data in a database. It extends the BaseDao class and contains various methods to handle user data, such as create a new user, update an existing user's information, authenticate users by checking their username and password, etc.

Attributes:

- No public attributes are declared in the class.
- Private attributes that are being used within the class include UserDtoMapper object, which maps the UserDto object to a SQL result set, and UserMapper object, which maps the User object to a SQL result set. The class also has some constants used in SQL queries, such as the names of the table columns.

Methods:

- authenticate(String username, String password): This method takes the username and password parameters as input, and returns the ID of the authenticated user if the authentication is successful. If the user is not found or the password is incorrect, this method returns null.

- create(User user): This method creates a new user in the database. It takes a User object as input and returns the ID of the created user. It first checks whether the username is unique, and then inserts the user data into the database.
- update(User user): This method updates an existing user's information in the database. It takes a User object as input and returns the updated User object. It updates the user's locale, email, and first connection details in the database.
- updatePassword(User user): This method updates the password of an existing user in the database. It takes a User object as input and returns the updated User object. It updates the password in the database by hashing the new password using the hashPassword() method, which uses the BCrypt library to create a secure hash.
- updateLastFmSessionToken(User user): This method updates the Last.fm session token of an existing user in the database. It takes a User object as input and returns the updated User object. It updates the Last.fm session token in the database.
- getQueryParam(UserCriteria criteria, FilterCriteria filterCriteria): This method is used internally to create the SQL query to retrieve users based on a given set of criteria and filter criteria. It returns a QueryParam object, which contains the SQL query and other parameters needed for the query.
- Various other private methods are also used in the class, such as the hashPassword() method, which hashes the password using the BCrypt library.

UserAlbumDao.java

Class description: The UserAlbumDao class represents a DAO (Data Access Object) used for creating UserAlbum entities in the database. It provides a method to create a new UserAlbum entity with the given parameters.

Attributes:

- None of its own attributes, but it uses an instance of Handle class provided by JDBI library and a ThreadLocalContext object.

Methods:

- create(UserAlbum userAlbum): Method that takes a UserAlbum object as parameter, sets some default values (UUID, createDate, and score) and inserts the data into the database using a JDBI Handle instance.

Relationships with other classes:

- UserAlbum: UserAlbumDao uses this model class to create a new UserAlbum entity in the database.

<u>**UserTrackDao.java**</u>

Description: The UserTrackDao class provides a data access layer for the UserTrack model object. It allows creating, updating, and retrieving user track objects from the database. It also provides methods to update play count and like status for a given user and track.

Attributes:

- None.

Methods:

- create(UserTrack userTrack): creates a new user / track in the database and returns the generated ID.
- getActiveUserTrack(String userId, String trackId): gets the active user / track for the given userId and trackId.
- getOrCreateUserTrack(String userId, String trackId): gets the active user / track for the given userId and trackId, or creates a new one if it does not exist.
- update(UserTrack userTrack): updates the given UserTrack object in the database and returns the updated object.
- incrementPlayCount(String userId, String trackId): increments the play count for the user / track with the given userId and trackId.
- initPlayCount(String userId, String trackId, Integer playCount): initializes the play count for the user / track with the given userId and trackId.
- like(String userId, String trackId): sets the "like" status to true for the user / track with the given userId and trackId.
- unlike(String userId, String trackId): sets the "like" status to false for the user / track with the given userId and trackId.

Relationships:

- UserTrackDao depends on the UserTrack model object to perform create and update operations.
- UserTrackDao uses the UserTrackMapper class to map the database rows to UserTrack objects.
- UserTrackDao depends on the Handle class from the JDBI library to execute SQL statements against the database.

AuthenticationToken.java

Description: The AuthenticationToken class represents an authentication token entity. It contains information related to an authentication token such as the token ID, user ID, creation date, and last connection date using the token.

Attributes:

- id: a String representing the token ID.
- userId: a String representing the user ID associated with the token.
- longLasted: a boolean indicating if the token should last longer (e.g. for a longer session).
- createDate: a Date object representing the creation date of the token.
- lastConnectionDate: a Date object representing the date of the last connection made using the token.

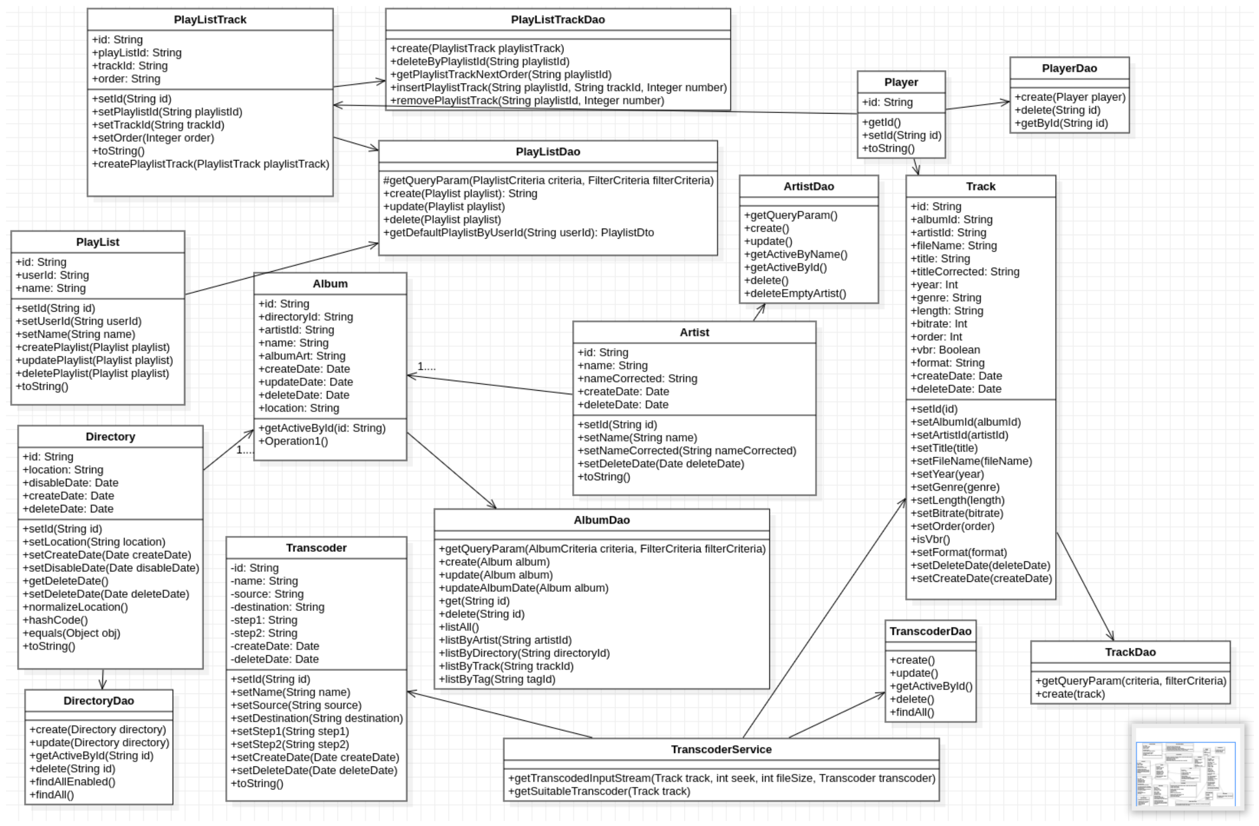
Methods:

- AuthenticationToken(): a no-argument constructor.
- AuthenticationToken(String id, String userId, boolean longLasted, Date createDate, Date lastConnectionDate): a constructor that takes in the values of the attributes and initializes them.
- getId(): a getter method for the id attribute.
- setId(String id): a setter method for the id attribute.
- getUserId(): a getter method for the userId attribute.
- setUserId(String userId): a setter method for the userId attribute.
- isLongLasted(): a getter method for the longLasted attribute.
- setLongLasted(boolean longLasted): a setter method for the longLasted attribute.

- getCreateDate(): a getter method for the createDate attribute.
- setCreateDate(Date createDate): a setter method for the createDate attribute.
- getLastConnectionDate(): a getter method for the lastConnectionDate attribute.
- setLastConnectionDate(Date lastConnectionDate): a setter method for the lastConnectionDate attribute.
- toString(): overrides the toString() method to provide a string representation of the object.

Relationships: For example, there may be a User class that contains information about the user and their credentials, and the AuthenticationToken class could be used to authenticate the user and provide access to certain features or functionality within the application.

2. <u>LIBRARY MANAGEMENT</u>



Album.java

Description: The purpose of this class is to encapsulate the data related to an album object in the application's database.The Album class represents an album entity in the system. It contains information about the album, such as its ID, name, directory ID, artist ID, album art, creation date, update date, deletion date, and location. The class also provides a method to retrieve an active album by its ID.

Attributes:

- id: String, representing the ID of the album.
- directoryId: String, representing the directory ID of the album.
- artistId: String, representing the artist ID of the album.
- name: String, representing the name of the album.
- albumArt: String, representing the ID of the album art.
- createDate: Date, representing the creation date of the album.
- updateDate: Date, representing the last update date of the album.
- deleteDate: Date, representing the deletion date of the album.
- location: String, representing the location of the album.

Methods:

- Constructor: Album(): creates an empty album object.
- Constructor: Album(id: String): creates an album object with the given ID.
- Constructor: Album(id: String, directoryId: String, artistId: String, name: String, albumArt: String, createDate: Date, updateDate: Date, deleteDate: Date, location: String): creates an album object with the given attributes.
- getActiveById(id: String): static method that retrieves an active album by its ID.
- Getter and setter methods for all attributes.

Relationships: The Album class has a many-to-one relationship with the Directory class (via the directoryId attribute) and the Artist class (via the artistId attribute). It also uses the AlbumMapper class for database mapping.

Artist.java

Description: The Artist class represents an artist entity in the music application's database. It contains information such as the artist's name, ID, name corrected, creation date, and deletion date.It represents an artist entity in a music database.It represents an artist entity in a music database.

Attributes:

- id: A string that represents the artist's ID.
- name: A string that represents the artist's name.
- nameCorrected: A string that represents the corrected name of the artist.
- createDate: A Date object that represents the date the artist was created.
- deleteDate: A Date object that represents the date the artist was deleted.

Methods:

- Artist(): A constructor that creates an instance of the Artist class with default values.
- Artist(String id, String name, String nameCorrected, Date createDate, Date deleteDate): A constructor that creates an instance of the Artist class with specific values for its attributes.
- getId(): A getter method that returns the artist's ID.
- setId(String id): A setter method that sets the artist's ID.
- getName(): A getter method that returns the artist's name.
- setName(String name): A setter method that sets the artist's name.
- getNameCorrected(): A getter method that returns the corrected name of the artist.
- setNameCorrected(String nameCorrected): A setter method that sets the corrected name of the artist.
- getCreateDate(): A getter method that returns the creation date of the artist.
- setCreateDate(Date createDate): A setter method that sets the creation date of the artist.
- getDeleteDate(): A getter method that returns the deletion date of the artist.
- setDeleteDate(Date deleteDate): A setter method that sets the deletion date of the artist.
- toString(): A method that returns a string representation of the Artist object.

<u>Directory.java</u>

Description: This is a Java class called Directory which represents a directory entity. It has attributes such as an ID, location, creation date, disable date, and delete date. It also has a method to normalize the directory location. The class provides methods to get and set these attributes, as well as methods to compare and hash the directory objects.

Attributes:

- id: A String representing the ID of the directory
- location: A String representing the location of the directory
- disableDate: A Date representing the disable date of the directory
- createDate: A Date representing the creation date of the directory
- deleteDate: A Date representing the delete date of the directory

Methods:

- Directory(): A default constructor for the class
- Directory(String id, String location, Date disableDate, Date createDate, Date deleteDate): A constructor that sets the values of the attributes
- getId(): A getter method for the id attribute
- setId(String id): A setter method for the id attribute
- getLocation(): A getter method for the location attribute
- setLocation(String location): A setter method for the location attribute
- getCreateDate(): A getter method for the createDate attribute
- setCreateDate(Date createDate): A setter method for the createDate attribute
- getDisableDate(): A getter method for the disableDate attribute
- setDisableDate(Date disableDate): A setter method for the disableDate attribute
- getDeleteDate(): A getter method for the deleteDate attribute
- setDeleteDate(Date deleteDate): A setter method for the deleteDate attribute
- normalizeLocation(): A method to normalize the directory location

- hashCode(): An overridden method to compute the hash code of the directory object
- equals(Object obj): An overridden method to compare two directory objects for equality
- toString(): An overridden method to convert the directory object to a string representation

Locale.java

Attributes:

id: a private field that holds the locale identifier.

Methods:

getId(): a public method that returns the value of the id field.

setId(String id): a public method that sets the value of the id field.

Locale(String id): a constructor that takes an id parameter and sets the id field to its value.

toString(): an overridden method that returns a string representation of the object. It uses the Objects.toStringHelper() method from the Guava library to format the output, including the id field.

Player.java

The Player class represents a music player. It has a single attribute id that identifies the player.

Attributes:

- id: a String representing the ID of the player.

Methods:

- getId(): returns the ID of the player.
- setId(String id): sets the ID of the player.
- toString(): returns a string representation of the player.

Relationships: There are no explicit relationships with other classes in this class, but it is likely that instances of this class would be associated with instances of other classes in a music application, such as a Playlist or Track.

Playlist.java

Description: The Playlist class represents a playlist entity in the system. A playlist can be a list of currently played tracks or a saved playlist with a name.

Attributes:

- id: The unique identifier of the playlist.
- userId: The ID of the user who owns the playlist.
- name: The name of the playlist.

Methods:

- Playlist(): Constructor without parameters.
- Playlist(String id): Constructor that sets the playlist ID.
- Playlist(String id, String userId): Constructor that sets the playlist ID and the user ID.
- getId(): Getter for the id attribute.
- setId(String id): Setter for the id attribute.
- getUserId(): Getter for the userId attribute.
- setUserId(String userId): Setter for the userId attribute.

- getName(): Getter for the name attribute.
- setName(String name): Setter for the name attribute.
- createPlaylist(Playlist playlist): Static method that creates a named playlist by generating a UUID and calling the create method of the PlaylistDao class.
- updatePlaylist(Playlist playlist): Static method that updates a named playlist by calling the update method of the PlaylistDao class.
- deletePlaylist(Playlist playlist): Static method that deletes a named playlist by calling the delete method of the PlaylistDao class.
- toString(): Method that returns a string representation of the playlist object.

Relationships: The Playlist class has a relationship with the PlaylistDao class, which is used to perform CRUD operations on the playlist data in the database.

PlaylistTrack.java

Description:

The PlaylistTrack class represents a relationship between a track and a playlist in the system. It contains information about the order of the track within the playlist.

Attributes:

- id: The unique identifier of the playlist track.
- playlistId: The ID of the playlist that the track belongs to.
- trackId: The ID of the track that is in the playlist.
- order: The order of the track within the playlist.

Methods:

- PlaylistTrack(): Constructor without parameters.
- PlaylistTrack(String id, String playlistId, String trackId, Integer order): Constructor that sets the ID, playlist ID, track ID, and order of the playlist track.

- getId(): Getter for the id attribute.
- setId(String id): Setter for the id attribute.
- getPlaylistId(): Getter for the playlistId attribute.
- setPlaylistId(String playlistId): Setter for the playlistId attribute.
- getTrackId(): Getter for the trackId attribute.
- setTrackId(String trackId): Setter for the trackId attribute.
- getOrder(): Getter for the order attribute.
- setOrder(Integer order): Setter for the order attribute.
- toString(): Method that returns a string representation of the playlist track object.
- createPlaylistTrack(PlaylistTrack playlistTrack): Static method that creates a playlist track by generating a UUID and calling the create method of the PlaylistTrackDao class.

Relationships:

The PlaylistTrack class has a relationship with the PlaylistTrackDao class, which is used to perform CRUD operations on the playlist track data in the database.

<u>Track.java</u>

This is a Java class named Track that models a music track entity. Here are the attributes and methods:

Attributes:

- id (String): the track ID
- albumId (String): the album ID that this track belongs to
- artistId (String): the artist ID that this track belongs to
- fileName (String): the file name of the track
- title (String): the title of the track
- titleCorrected (String): a corrected title of the track
- year (Integer): the year of the track
- genre (String): the genre of the track
- length (Integer): the length of the track in seconds
- bitrate (Integer): the bitrate of the track in kbps

- order (Integer): the order of the track in the album
- vbr (boolean): whether the track is encoded in variable bitrate (VBR)
- format (String): the format of the track
- createDate (Date): the creation date of the track
- deleteDate (Date): the deletion date of the track

Methods:

- Track() (constructor): constructs a new Track object
- Track(id, albumId, artistId, fileName, title, titleCorrected, year, genre, length, bitrate, order, vbr, format, createDate, deleteDate) (constructor): constructs a new Track object with the given attributes
- getId() (getter): gets the track ID
- setId(id) (setter): sets the track ID
- getAlbumId() (getter): gets the album ID
- setAlbumId(albumId) (setter): sets the album ID
- getArtistId() (getter): gets the artist ID
- setArtistId(artistId) (setter): sets the artist ID
- getTitle() (getter): gets the track title
- setTitle(title) (setter): sets the track title
- getFileName() (getter): gets the track file name
- setFileName(fileName) (setter): sets the track file name
- getYear() (getter): gets the track year
- setYear(year) (setter): sets the track year
- getGenre() (getter): gets the track genre
- setGenre(genre) (setter): sets the track genre
- getLength() (getter): gets the track length in seconds
- setLength(length) (setter): sets the track length in seconds
- getBitrate() (getter): gets the track bitrate in kbps
- setBitrate(bitrate) (setter): sets the track bitrate in kbps
- getOrder() (getter): gets the track order
- setOrder(order) (setter): sets the track order
- isVbr() (getter): checks if the track is encoded in variable bitrate (VBR)
- setVbr(vbr) (setter): sets whether the track is encoded in variable bitrate (VBR)
- getFormat() (getter): gets the track format
- setFormat(format) (setter): sets the track format
- getCreateDate() (getter): gets the track creation date

- setCreateDate(createDate) (setter): sets the track creation date
- getDeleteDate() (getter): gets the track deletion date
- setDeleteDate(deleteDate) (setter): sets the track deletion date

Transcoder.java

Attributes:

- id: a private String variable that represents the transcoder ID.
- name: a private String variable that represents the transcoder name.
- source: a private String variable that represents the source formats of the transcoder, separated by space.
- destination: a private String variable that represents the destination format of the transcoder.
- step1: a private String variable that represents the command for the first step of the transcoder.
- step2: a private String variable that represents the command for the second step of the transcoder.
- createDate: a private Date variable that represents the creation date of the transcoder.
- deleteDate: a private Date variable that represents the deletion date of the transcoder.

Methods:

- Transcoder(): a default constructor that creates an empty Transcoder object.
- Transcoder(String id, String name, String source, String destination, String step1, String step2, Date createDate, Date deleteDate): a constructor that takes all the variables as input and initializes the Transcoder object.
- getId(): a getter method that returns the transcoder ID.
- setId(String id): a setter method that sets the transcoder ID.
- getName(): a getter method that returns the transcoder name.
- setName(String name): a setter method that sets the transcoder name.
- getSource(): a getter method that returns the source formats of the transcoder.

- setSource(String source): a setter method that sets the source formats of the transcoder.
- getDestination(): a getter method that returns the destination format of the transcoder.
- setDestination(String destination): a setter method that sets the destination format of the transcoder.
- getStep1(): a getter method that returns the command for the first step of the transcoder.
- setStep1(String step1): a setter method that sets the command for the first step of the transcoder.
- getStep2(): a getter method that returns the command for the second step of the transcoder.
- setStep2(String step2): a setter method that sets the command for the second step of the transcoder.
- getCreateDate(): a getter method that returns the creation date of the transcoder.
- setCreateDate(Date createDate): a setter method that sets the creation date of the transcoder.
- getDeleteDate(): a getter method that returns the deletion date of the transcoder.
- setDeleteDate(Date deleteDate): a setter method that sets the deletion date of the transcoder.
- toString(): a method that returns a string representation of the Transcoder object.

<u>TranscoderService.java</u>

Methods:

- getTranscodedInputStream(Track track, int seek, int fileSize, Transcoder transcoder): Returns a TranscodedInputStream for a given Track object, seek position, expected file size, and Transcoder object. The TranscodedInputStream is created by running a command in a ProcessBuilder object.

- getSuitableTranscoder(Track track): Returns a Transcoder object that is suitable for the given Track object.

Relationships:

The class uses several other classes defined in different packages, including com.google.common.io.Files, com.sismics.music.core.dao.dbi.TranscoderDao, com.sismics.music.core.model.dbi.Track, com.sismics.music.core.model.dbi.Transcoder, and com.sismics.util.io.TranscodedInputStream.

<u>AlbumDao.java</u>

This is a Java code of a DAO (Data Access Object) class named AlbumDao. This DAO class provides methods to interact with the database related to Album entities. It extends a BaseDao class which provides base functionality to the DAO classes.

Methods:

- getQueryParam(AlbumCriteria criteria, FilterCriteria filterCriteria): Returns a QueryParam object representing the SQL query used to retrieve a list of albums that meet the specified criteria. The method takes an AlbumCriteria object and a FilterCriteria object as input parameters, and returns a QueryParam object.
- create(Album album): Creates a new album in the database. The method takes an Album object as input parameter, sets the album ID, create date, and update date if necessary, and inserts the album data into the database. The method returns the ID of the created album.
- update(Album album): Updates an existing album in the database. The method takes an Album object as input parameter, and updates the corresponding record in the database with the new data. The method returns the updated Album object.
- updateAlbumDate(Album album): Updates the date of an existing album in the database. The method takes an Album object as input parameter, and

updates the corresponding record in the database with the new date. The method returns the updated Album object.

- get(String id): Retrieves an album with the specified ID from the database. The method takes a string ID as input parameter, retrieves the corresponding record from the database, and returns an Album object representing the retrieved data.
- delete(String id): Deletes an album with the specified ID from the database. The method takes a string ID as input parameter, deletes the corresponding record from the database, and returns the number of deleted rows.
- listAll(): Retrieves a list of all albums in the database. The method returns a list of Album objects representing the retrieved data.
- listByArtist(String artistId): Retrieves a list of albums with the specified artist ID from the database. The method takes a string artist ID as input parameter, retrieves the corresponding records from the database, and returns a list of Album objects representing the retrieved data.
- listByDirectory(String directoryId): Retrieves a list of albums with the specified directory ID from the database. The method takes a string directory ID as input parameter, retrieves the corresponding records from the database, and returns a list of Album objects representing the retrieved data.
- listByTrack(String trackId): Retrieves a list of albums containing the specified track ID from the database. The method takes a string track ID as input parameter, retrieves the corresponding records from the database, and returns a list of Album objects representing the retrieved data.
- listByTag(String tagId): Retrieves a list of albums with the specified tag ID from the database. The method takes a string tag ID as input parameter, retrieves the corresponding records from the database, and returns a list of Album objects representing the retrieved data.

ArtistDao.java

This is a Java class for a Data Access Object (DAO) for artists in a music application.

The ArtistDao extends the BaseDao class with generic type parameters ArtistDto and ArtistCriteria. The BaseDao provides methods to implement data access using jDBI (JDBC Data Binding and SQL Objects).

The ArtistDto class is a Data Transfer Object (DTO) for an artist and contains an artist's ID and name.

The ArtistCriteria class contains criteria that can be used to filter artists.

This class has several methods for querying and modifying artist data in the database, including:

- getQueryParam: a protected method that generates a QueryParam object for the provided ArtistCriteria and FilterCriteria parameters. The QueryParam object contains a SQL query string, a list of criteria, a map of parameter values, a list of order-by clauses, and a mapper that maps query results to ArtistDto objects.
- create: a method that creates a new artist in the database using the provided Artist object.
- update: a method that updates an artist in the database using the provided Artist object.
- getActiveByName: a method that retrieves an active artist from the database by name.
- getActiveById: a method that retrieves an active artist from the database by ID.
- delete: a method that deletes an artist from the database by ID.
- deleteEmptyArtist: a method that deletes artists from the database that don't have any associated albums or tracks.

Overall, this class provides methods for querying and modifying artist data in the database using jDBI.

<u>**AuthenticationTokenDao.java**</u>

Description:

The AuthenticationTokenDao class is a Data Access Object (DAO) class for managing authentication tokens in the database. It provides methods to retrieve, create, delete, and update authentication tokens.

Methods:

- get(String id): Gets the authentication token with the given ID.
- create(AuthenticationToken authenticationToken): Creates a new authentication token with the given information and saves it to the database. Returns the ID of the newly created token.
- delete(String authenticationTokenId): Deletes the authentication token with the given ID from the database.
- deleteOldSessionToken(String userId): Deletes all short-lived authentication tokens for the user with the given ID that were created more than a day ago.
- updateLastConnectionDate(String id): Updates the last connection date of the authentication token with the given ID.

Relationships:

AuthenticationTokenDao has a composition relationship with the AuthenticationToken class, which represents an authentication token in the system. The create method takes an instance of AuthenticationToken as input and stores it in the database. The get method returns an instance of AuthenticationToken based on the ID passed to the method.

AuthenticationTokenDao has a dependency on the ThreadLocalContext class, which provides thread-local storage for objects. The get and create methods use the ThreadLocalContext to obtain a database connection handle.

<u>DirectoryDao.java</u>

Description: This is a DAO (Data Access Object) class that provides CRUD (Create, Read, Update, Delete) operations to manipulate Directory entities in the database. The Directory entity represents a directory location in the file system.

Methods:

- create(Directory directory): String: Creates a new Directory entity in the database and returns its ID.
- update(Directory directory): Directory: Updates an existing Directory entity in the database and returns the updated entity.
- getActiveById(String id): Directory: Retrieves an active Directory entity by its ID from the database and returns it.
- delete(String id): void: Marks a Directory entity as deleted in the database.
- findAllEnabled(): List<Directory>: Retrieves a list of all enabled (not disabled or deleted) Directory entities from the database and returns it.
- findAll(): List<Directory>: Retrieves a list of all Directory entities (including disabled and deleted) from the database and returns it.

Relationships: This class has a composition relationship with the Directory entity, which is used as a parameter in the methods to create, update, and get directory entities. It also has a dependency relationship with the ThreadLocalContext class, which is used to get the database connection handle to execute SQL statements.

LocaleDao.java

This code defines a DAO class for the Locale entity.

The LocaleDao class has two methods:

- getById: retrieves a single Locale entity based on its id. It takes a single parameter id which is the identifier of the Locale. This method returns a Locale object based on the specified id value. It uses the Handle object from the jdbi library to query the database, retrieve the Locale entity, and map it to a Locale object using the mapTo method.
- findAll: retrieves all Locale entities in the database. It returns a list of all the Locale objects in the database, sorted by their id value. It also uses the

Handle object to query the database and the mapTo method to map the results to a list of Locale objects.

The LocaleDao class has a relationship with the Locale class, as it uses Locale objects to perform the database operations. The class also uses the Handle object from the jdbi library to communicate with the database. The class also has a dependency on the ThreadLocalContext class, which provides a thread-local storage mechanism for managing the database connections.

PlayerDao.java

Description: The PlayerDao class is a DAO (Data Access Object) for the Player entity, which allows the interaction with the database to perform CRUD (Create, Read, Update, Delete) operations on the Player entity.

Methods:

- create(Player player): This method creates a new Player in the database by generating a new UUID and inserting the Player's ID into the database. It takes a Player object as a parameter and returns the ID of the new Player.
- delete(String id): This method deletes a Player from the database by its ID. It takes the ID of the Player to be deleted as a parameter and does not return anything.
- getById(String id): This method retrieves a Player from the database by its ID. It takes the ID of the Player to be retrieved as a parameter and returns a Player object.

Relationships:

- PlayerDao has a composition relationship with Player as it interacts with the Player entity in the database through its methods.
- PlayerDao uses the Handle class from the JDBI library to execute SQL statements in the database.

- PlayerDao also uses the ThreadLocalContext class to retrieve the Handle instance from the current thread's context.

<u>**PlaylistDao.java**</u>

Description: The PlaylistDao class is a DAO (Data Access Object) class that provides an interface to the database for performing CRUD (Create, Read, Update, and Delete) operations on Playlist entities. It extends the BaseDao class, which provides a generic implementation of common DAO methods.

Methods:

- protected QueryParam getQueryParam(PlaylistCriteria criteria, FilterCriteria filterCriteria): This method constructs the SQL query for fetching a list of playlists from the database based on the provided criteria and filter criteria.
- public String create(Playlist playlist): This method inserts a new playlist into the database and returns the ID of the newly created playlist.
- public void update(Playlist playlist): This method updates the name of an existing playlist in the database.
- public void delete(Playlist playlist): This method deletes a playlist from the database.
- public PlaylistDto getDefaultPlaylistByUserId(String userId): This method fetches the default playlist for a given user from the database.
- private List<PlaylistDto> assembleResultList(List<Object[]> resultList): This method converts the result set from a database query to a list of PlaylistDto objects.

Relationships:

- PlaylistDao extends BaseDao and uses its methods for common database operations.
- PlaylistDao uses the PlaylistCriteria and PlaylistMapper classes to construct SQL queries and convert database records to PlaylistDto objects, respectively.

PlaylistTrackDao.java

Description: The PlaylistTrackDao class is a data access object that provides CRUD operations on the PlaylistTrack entity for persistence storage in a database.

Attributes: This class has no attributes.

Methods:

- create(PlaylistTrack playlistTrack): Inserts a new PlaylistTrack object into the database.
- deleteByPlaylistId(String playlistId): Deletes all PlaylistTrack objects with a specific playlist ID.
- getPlaylistTrackNextOrder(String playlistId): Gets the next order number for a track in a playlist, based on the existing tracks in the playlist.
- insertPlaylistTrack(String playlistId, String trackId, Integer number): Inserts a new track into a playlist at a specified position, reordering the existing tracks if necessary.
- removePlaylistTrack(String playlistId, Integer number): Removes a track from a playlist at a specified position, reordering the existing tracks if necessary.

Relationships: The PlaylistTrackDao class is dependent on the following classes:

- PlaylistTrack: A model class that represents a track in a playlist.
- ThreadLocalContext: A utility class that manages thread-local storage of a context object.
- ObjectMapper: A utility class that maps the result set of a database query to an object.
- Handle: A class that provides a connection to the database and allows executing SQL statements on that connection.
- StringMapper: A utility class that maps a single string column from the result set of a database query to a string.

TrackDao.java

This is a method for creating a new track in the database, written in Java using JDBI.It takes in a Track object and generates a unique ID for it, sets the creation date to the current time, and then inserts it into the database using JDBI.The

The TrackDao class is a data access object (DAO) for managing tracks in a music library. It extends the BaseDao class, which is a utility class for executing database queries using the jDBI library.

Methods:

- getQueryParam(criteria, filterCriteria): This method is used to construct a QueryParam object, which contains the SQL query, query parameters, sort criteria, and filter criteria to use for a database query. It takes a TrackCriteria object and a FilterCriteria object as input, and returns a QueryParam object as output.
- create(track): This method is used to create a new track in the database. It takes a Track object as input, sets its id and createDate properties, and inserts it into the t_track table using a jDBI query. It returns the ID of the new track as a string.

The TrackDao class is related to the following classes/interfaces:

- TrackCriteria: A class that encapsulates search criteria for fetching Tracks from the database.
- TrackDto: A DTO (Data Transfer Object) that represents the Track entity returned by the getQueryParam method.
- TrackMapper: A mapper class that maps a Track entity to a TrackDto entity.
- Track: A model class that represents a Track entity in the application.

<u>TranscoderDao.java</u>

Methods:

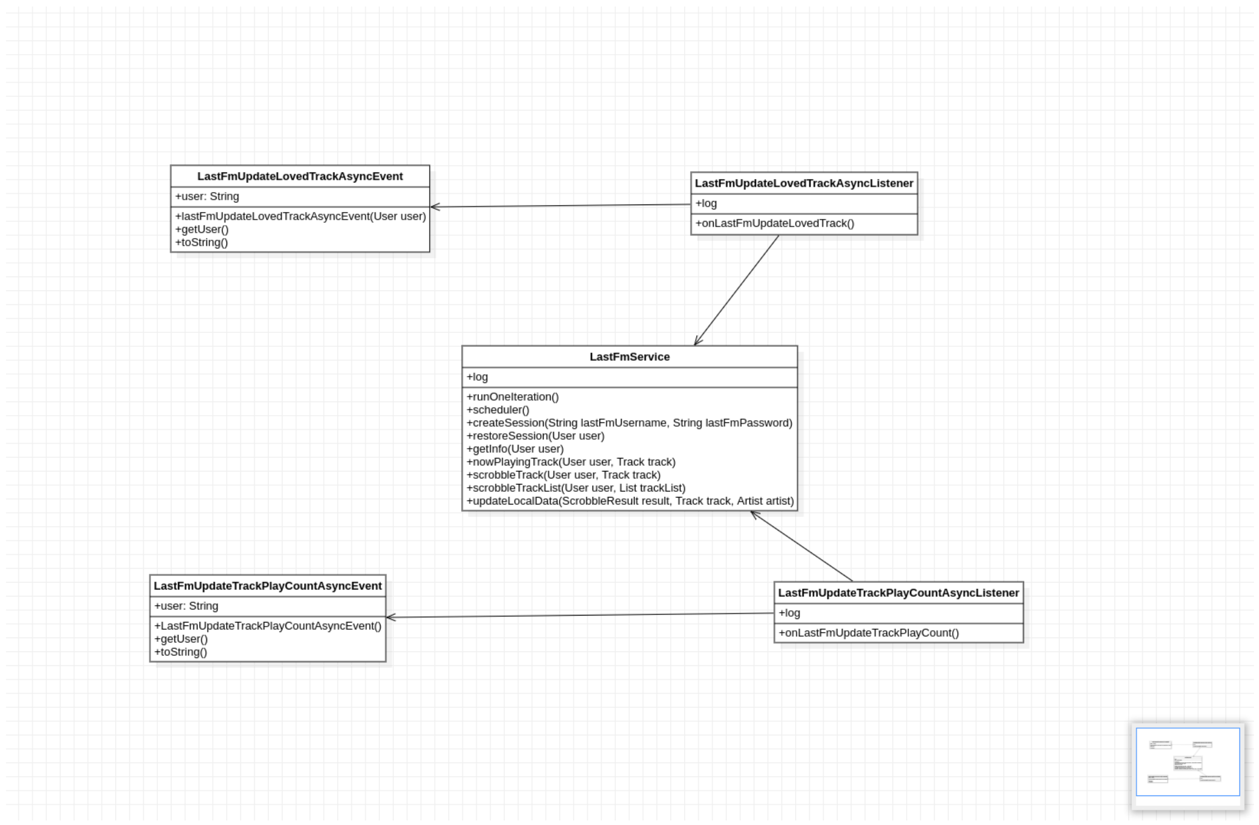
- create: creates a new Transcoder in the database with the given information.
- update: updates the information of an existing Transcoder in the database.

- getActiveById: retrieves an active Transcoder by its ID from the database. An active Transcoder is one that has not been deleted, as determined by the presence of a non-null deleteDate field in the database table.
- delete: soft-deletes a Transcoder in the database by setting its deleteDate field to the current date/time.
- findAll: retrieves a list of all active Transcoder entities from the database.

Relationships:

- The TranscoderDao class has a dependency on the ThreadLocalContext class, which provides access to the JDBI Handle object used to execute SQL statements against the database.
- The TranscoderDao class depends on the TranscoderMapper class, which maps result set columns to Java object fields for the Transcoder entity.

3. LAST.FM INTEGRATION


LastFmUpdateLovedTrackAsyncEvent
+user: String
+lastFmUpdateLovedTrackAsyncEvent(User user)
+getUser()
+toString()
LastFmUpdateLovedTrackAsyncListener
+log
+onLastFmUpdateLovedTrack()
LastFmService
+log
+runOneIteration()
+scheduler()
+createSession(String lastFmUsername, String lastFmPassword)
+restoreSession(User user)
+getInfo(User user)
+nowPlayingTrack(User user, Track track)
+scrobbleTrack(User user, Track track)
+scrobbleTrackList(User user, List trackList)
+updateLocalData(ScrobbleResult result, Track track, Artist artist)
LastFmUpdateTrackPlayCountAsyncEvent
+user: String
+LastFmUpdateTrackPlayCountAsyncEvent()
+getUser()
+toString()
LastFmUpdateTrackPlayCountAsyncListener
+log
+onLastFmUpdateTrackPlayCount()

<u>LastFmService.java</u>

This is a Java class named LastFmService from the com.sismics.music.core.service.lastfm package. It is a service that interacts with Last.fm API to update the play counts and love tracks of users, as well as retrieve information from Last.fm.

The class extends AbstractScheduledService from the com.google.common.util.concurrent package and overrides two of its methods: runOneIteration() and scheduler(). The former runs periodically and updates the play counts and loved tracks of all users that have a Last.fm session token stored in the database by creating and posting two LastFmAsyncEvent objects to the LastFmEventBus. The latter defines the schedule of the service: it runs every 24 hours with an initial delay of 23 hours.

This is a Java class named LastFmService that extends AbstractScheduledService. It provides a service to interact with Last.fm, a music service that provides a web services API to retrieve and interact with data from their music database.

The LastFmService provides various methods to retrieve and update data from a user's Last.fm account. It has methods to create and restore user sessions, to update now playing and to scrobble a track, and to retrieve Last.fm user information.

Attributes:

- log: a Logger instance to log messages

Methods:

- runOneIteration(): overrides the runOneIteration() method of the AbstractScheduledService class. It retrieves a list of all users that have a Last.fm session token set, and posts LastFmUpdateLovedTrackAsyncEvent and LastFmUpdateTrackPlayCountAsyncEvent events for each of them using the Last.fm event bus.
- scheduler(): overrides the scheduler() method of the AbstractScheduledService class. It returns a Scheduler instance that is used to schedule the runOneIteration() method to be executed once per day.
- createSession(String lastFmUsername, String lastFmPassword): creates a Last.fm session for a given user using the user's Last.fm username and password.
- restoreSession(User user): restores a Last.fm session for a given user using the user's stored Last.fm session token.
- getInfo(User user): retrieves the Last.fm information for a given user using the user's Last.fm session.
- nowPlayingTrack(User user, Track track): updates the currently playing track for a given user on Last.fm using the user's Last.fm session and a given track.
- scrobbleTrack(User user, Track track): scrobbles a given track for a given user on Last.fm using the user's Last.fm session.
- scrobbleTrackList(User user, List<Track> trackList): scrobbles a list of tracks for a given user on Last.fm using the user's Last.fm session.

- updateLocalData(ScrobbleResult result, Track track, Artist artist): updates the play count and love status of a given track and artist in the local database based on the result of a Last.fm scrobbling operation. This method is called by the nowPlayingTrack() and scrobbleTrack() methods after a scrobbling operation is completed.

LastFmUpdateLovedTrackAsyncEvent.java:

Attributes:

- user: A private attribute of type User, representing the user for which the Last.fm update loved tracks event is being triggered.

Methods:

- LastFmUpdateLovedTrackAsyncEvent(User user): A constructor method that takes a User object and sets it as the value of the user attribute.
- getUser(): A getter method that returns the User object stored in the user attribute.
- toString(): An overridden method from the Object class that returns a string representation of the object. In this case, it returns a string that includes the value of the user attribute.

Relationships:

- This class has a relationship with the User class, as it has an attribute of type User and takes a User object in its constructor.

LastFmUpdateLovedTrackAsyncListener.java:

Attributes:

log: a private static final Logger object for logging messages.

Methods:

onLastFmUpdateLovedTrack: a public method that takes a LastFmUpdateLovedTrackAsyncEvent object as a parameter and processes it by retrieving the User object and calling a method on a LastFmService object to import the user's loved tracks from Last.fm.

Relationships:

LastFmUpdateLovedTrackAsyncListener has a dependency on LastFmUpdateLovedTrackAsyncEvent, User, AppContext, LastFmService, TransactionUtil, and Logger.

LastFmUpdateTrackPlayCountAsyncEvent.java:

Attributes:

user: A private attribute of type User that represents the user for which the track play count needs to be updated.

Methods:

- LastFmUpdateTrackPlayCountAsyncEvent: A constructor that takes a User parameter and sets the user attribute of the class.
- getUser: A getter method that returns the user attribute of the class.
- toString: An overridden method that returns a string representation of the class using the Objects.toStringHelper method.

Relationships:

The class has a dependency on User, which is used as a parameter to the constructor and stored as an attribute.

LastFmUpdateTrackPlayCountAsyncListener.java:

Attributes:

log: a Logger object used to log information about the event processing.

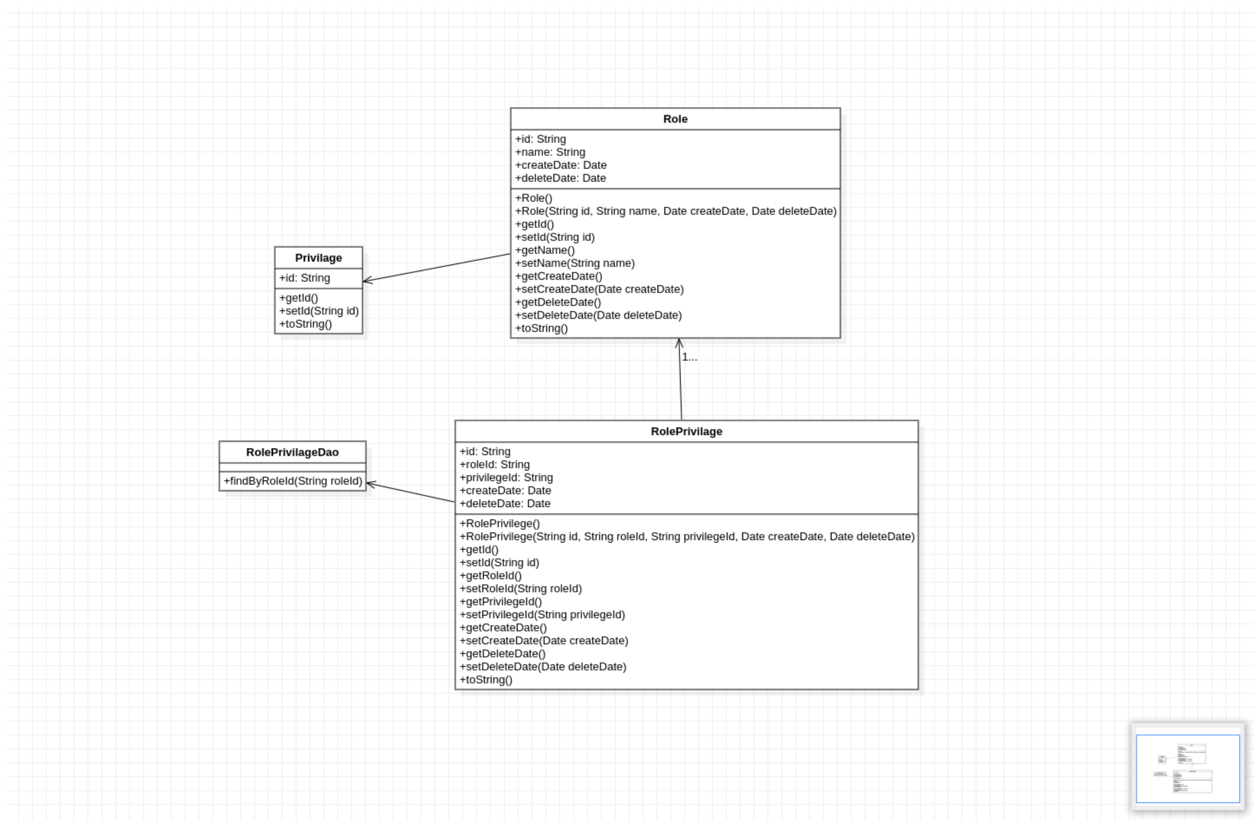
Methods:

onLastFmUpdateTrackPlayCount: a method annotated with @Subscribe that handles the LastFmUpdateTrackPlayCountAsyncEvent event. It retrieves the user from the event, calls the LastFmService.importTrackPlayCount method to import the track play count for the user, and logs the completion of the event processing.

Relationships:

- The LastFmUpdateTrackPlayCountAsyncListener class has a one-to-one relationship with the LastFmUpdateTrackPlayCountAsyncEvent class, in that it listens to this event and processes it when received.
- The LastFmUpdateTrackPlayCountAsyncListener class has a dependency on the LastFmService class, as it calls its importTrackPlayCount method to handle the event processing.
- The LastFmUpdateTrackPlayCountAsyncListener class has a dependency on the AppContext class, as it calls its getInstance method to retrieve the LastFmService instance.

4. **ADMINISTRATOR FEATURES**



<u>**Privilege.java**</u>

Description: The Privilege class represents a privilege entity, which has a unique identifier that represents the privilege (ex: "ADMIN").

Attributes:

- id (String): Privilege ID.

Methods:

- Privilege(String id): Constructor method that sets the privilege ID.
- getId(): Getter method that returns the privilege ID.
- setId(String id): Setter method that sets the privilege ID.
- toString(): Method that returns a string representation of the privilege, including its ID.

Role.java

Description: This class represents a role, which is a set of privileges that can be assigned to a user.

Attributes:

- id: String representing the role ID.
- name: String representing the role name.
- createDate: Date representing the creation date of the role.
- deleteDate: Date representing the deletion date of the role.

Methods:

- Role(): Constructor for the Role class.
- Role(String id, String name, Date createDate, Date deleteDate): Constructor for the Role class with arguments for all attributes.
- getId(): Getter for the id attribute.
- setId(String id): Setter for the id attribute.
- getName(): Getter for the name attribute.
- setName(String name): Setter for the name attribute.
- getCreateDate(): Getter for the createDate attribute.
- setCreateDate(Date createDate): Setter for the createDate attribute.
- getDeleteDate(): Getter for the deleteDate attribute.
- setDeleteDate(Date deleteDate): Setter for the deleteDate attribute.
- toString(): Returns a String representation of the Role object using Guava's Objects.toStringHelper().

RolePrivilege.java

The RolePrivilege class represents a role privilege in the system.

Attributes:

- id: the unique identifier of the role privilege.
- roleId: the ID of the role that has the privilege.
- privilegeId: the ID of the privilege that the role has.
- createDate: the date the role privilege was created.
- deleteDate: the date the role privilege was deleted.

Methods:

- RolePrivilege(): a default constructor.
- RolePrivilege(String id, String roleId, String privilegeId, Date createDate, Date deleteDate): a constructor that sets all the attributes.
- getId(): a getter for the id attribute.
- setId(String id): a setter for the id attribute.
- getRoleId(): a getter for the roleId attribute.
- setRoleId(String roleId): a setter for the roleId attribute.
- getPrivilegeId(): a getter for the privilegeId attribute.
- setPrivilegeId(String privilegeId): a setter for the privilegeId attribute.
- getCreateDate(): a getter for the createDate attribute.
- setCreateDate(Date createDate): a setter for the createDate attribute.
- getDeleteDate(): a getter for the deleteDate attribute.
- setDeleteDate(Date deleteDate): a setter for the deleteDate attribute.
- toString(): returns a string representation of the object.

Relationships: The RolePrivilege class has a many-to-one relationship with the Role class and a many-to-one relationship with the Privilege class. It also has a one-to-many relationship with the RolePrivilegeDao class, which is responsible for persisting and retrieving RolePrivilege objects in the database.

RolePrivilegeDao.java

Description: The RolePrivilegeDao class is a data access object (DAO) for retrieving privileges of a role from the database.

Attributes: The RolePrivilegeDao class has no attributes.

Methods:

- findByRoleId(String roleId): This method retrieves the set of privileges associated with the given role ID from the database. It takes in a String parameter roleId, which is the ID of the role to retrieve the privileges for, and returns a Set of String objects representing the privileges associated with the role.

Relationships: The RolePrivilegeDao class has a relationship with the Handle and ThreadLocalContext classes, which are used to interact with the database, and the Map and Set classes, which are used to store and manipulate data. It also has an implied relationship with the t_role and t_role_privilege database tables, as it is used to query data from these tables.

Points of strengths and weaknesses:

From the UML class diagram above, we can see that the system has a good separation of concerns with clear distinctions between the classes. Each of the data access object (DAO) classes provides an interface to interact with the database for their respective classes. The player class is responsible for playing the tracks and is associated with the playlist and playlist track classes.

One of the strengths of the system is that it provides a clear and consistent structure for interacting with the database. The DAO classes make it easy to perform common operations like creating, reading, updating, and deleting data. The player class also provides a convenient interface for playing tracks and managing playlists.

One potential weakness of the system is that it does not handle errors or exceptions that may occur during database interactions. If an error occurs, the system may fail without giving the user any feedback. Additionally, the system may be limited in scalability, as it relies on a single database and does not provide any mechanisms for distributing or replicating data.

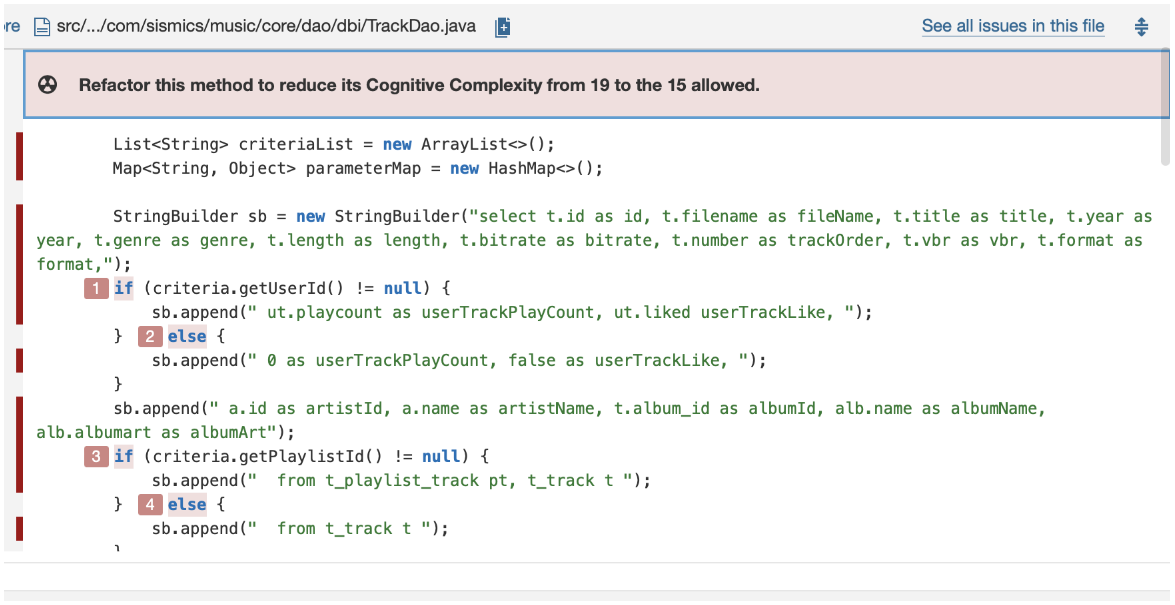
Strengths:

- The system follows a clear and organized class structure, with different domain entities separated into their own classes.
- The use of DAO classes provides a separation of concerns, making the system more modular and maintainable.
- The design is scalable, with the ability to add more domain entities in the future without affecting the existing classes.

Task 2a and 3a combined:

(First stating the Design smell and then the refactored part)

1) Insufficient Modularization (TrackDao):



re src/.../com/sismics/music/core/dao/dbi/TrackDao.java See all issues in this file

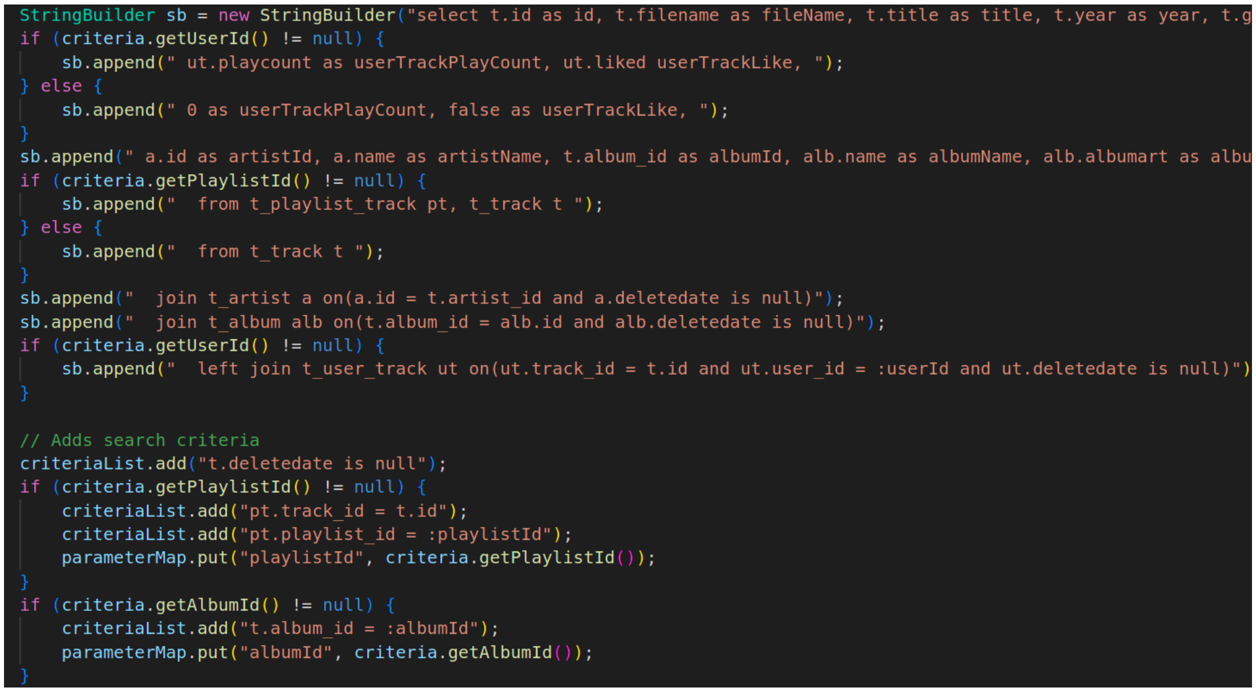
Refactor this method to reduce its Cognitive Complexity from 19 to the 15 allowed.

```
        List<String> criteriaList = new ArrayList<>();
        Map<String, Object> parameterMap = new HashMap<>();

        StringBuilder sb = new StringBuilder("select t.id as id, t.filename as fileName, t.title as title, t.year as
year, t.genre as genre, t.length as length, t.bitrate as bitrate, t.number as trackOrder, t.vbr as vbr, t.format as
format,");
     1 if (criteria.getUserId() != null) {
            sb.append(" ut.playcount as userTrackPlayCount, ut.liked userTrackLike, ");
        } 2 else {
            sb.append(" 0 as userTrackPlayCount, false as userTrackLike, ");
        }
        sb.append(" a.id as artistId, a.name as artistName, t.album_id as albumId, alb.name as albumName,
alb.albumart as albumArt");
     3 if (criteria.getPlaylistId() != null) {
            sb.append("  from t_playlist_track pt, t_track t ");
        } 4 else {
            sb.append("  from t_track t ");
```

- Sonarqube: The class TrackDao has a high cognitive complexity, which indicates a lot of if else statements, now on observation we found that it uses methods of criteria class around 19 times through if statements, which also indicates feature envy code smell.
- Bloated Method: The method getQueryParam is quite long and contains multiple responsibilities like constructing the SQL query string, adding search criteria, constructing the parameter map, etc. These responsibilities could be broken down into smaller methods, making the code more modular and maintainable.

- The presence of a long method can indicate the Large Method design smell, which may indicate that this method is doing too much and could be split into smaller methods that have a single responsibility. This can make the code easier to understand, maintain, and test.
- Another design smell that could be present in this code is the Broken Modularization design smell, as this class may be tightly coupled with the database and SQL queries, and it may not be easy to reuse this code in other parts of the application or to replace the underlying data storage technology.
- To improve this code, it might be a good idea to separate the code that constructs the SQL query from the code that adds search criteria and sorting. This could make the code more modular, easier to test and maintain, and also more reusable across different parts of the application.
- From the code smells, and from the strong indication of coupling, we came to an obvious conclusion that this is a case of Insufficient Modularization design smell. Insufficient Modularization can lead to a lack of scalability and flexibility in the codebase, making it challenging to maintain and extend over time.
- We suggest these conditions to be checked in criteria class itself, to counter these smells.

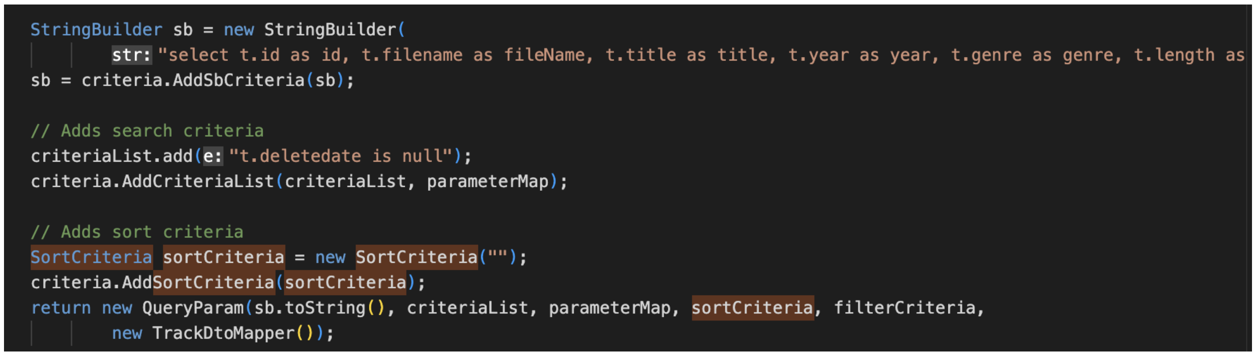
Before refactoring:



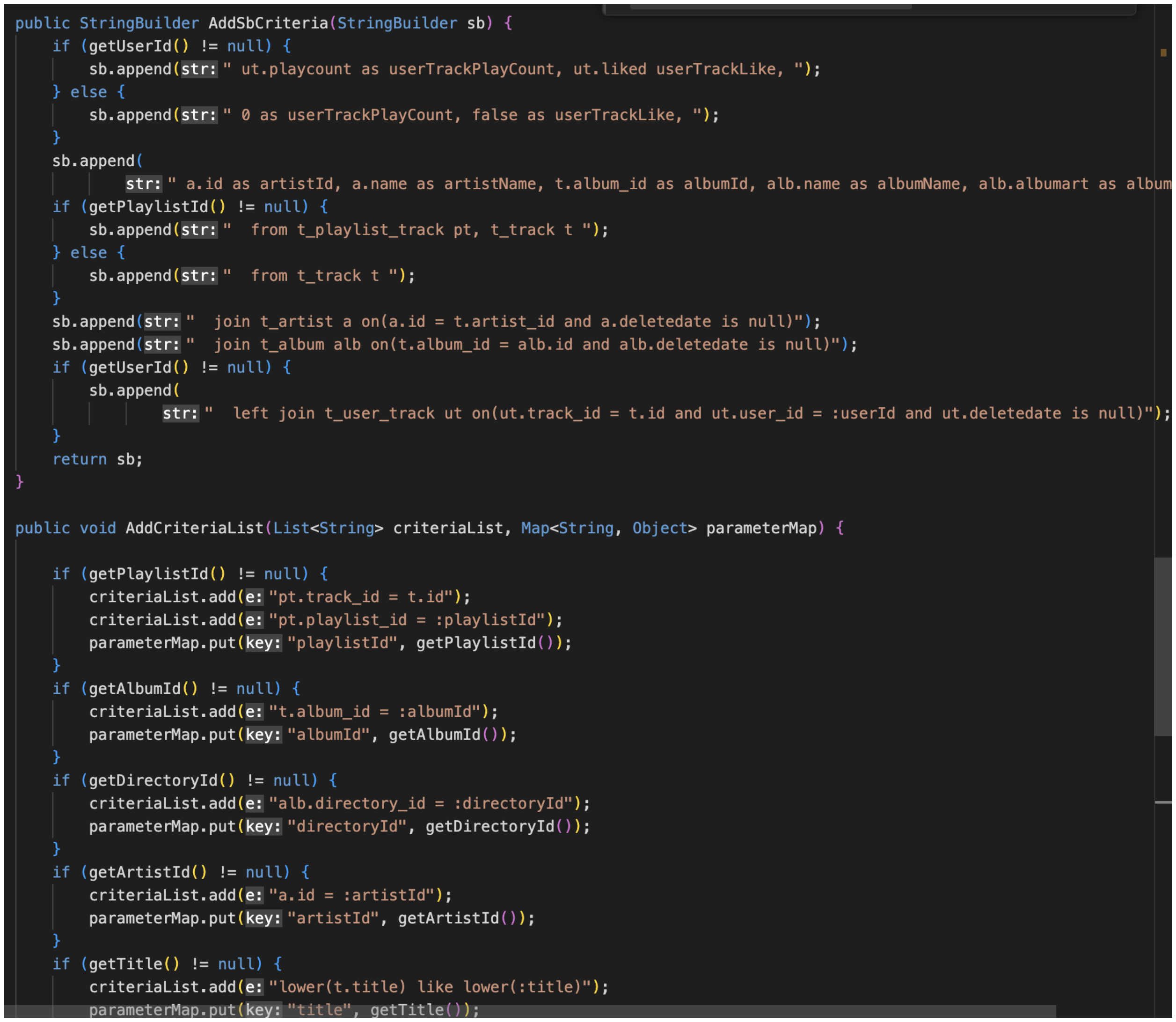
```
StringBuilder sb = new StringBuilder("select t.id as id, t.filename as fileName, t.title as title, t.year as year, t.g
if (criteria.getUserId() != null) {
    sb.append(" ut.playcount as userTrackPlayCount, ut.liked userTrackLike, ");
} else {
    sb.append(" 0 as userTrackPlayCount, false as userTrackLike, ");
}
sb.append(" a.id as artistId, a.name as artistName, t.album_id as albumId, alb.name as albumName, alb.albumart as albu
if (criteria.getPlaylistId() != null) {
    sb.append("  from t_playlist_track pt, t_track t ");
} else {
    sb.append("  from t_track t ");
}
sb.append("  join t_artist a on(a.id = t.artist_id and a.deletedate is null)");
sb.append("  join t_album alb on(t.album_id = alb.id and alb.deletedate is null)");
if (criteria.getUserId() != null) {
    sb.append("  left join t_user_track ut on(ut.track_id = t.id and ut.user_id = :userId and ut.deletedate is null)")
}

// Adds search criteria
criteriaList.add("t.deletedate is null");
if (criteria.getPlaylistId() != null) {
    criteriaList.add("pt.track_id = t.id");
    criteriaList.add("pt.playlist_id = :playlistId");
    parameterMap.put("playlistId", criteria.getPlaylistId());
}
if (criteria.getAlbumId() != null) {
    criteriaList.add("t.album_id = :albumId");
    parameterMap.put("albumId", criteria.getAlbumId());
}
```

<u>After refactoring:</u>

Code has been refactored so that all the attributes of the class TrackCriteria can be accessed directly in that class, methods AddSbCriteria(), AddCRiteriaList() and AddSortCriteria() are added to TrackCriteria we can be called by TrackDao for their respective operations in build queries.



```java
StringBuilder sb = new StringBuilder(
        str: "select t.id as id, t.filename as fileName, t.title as title, t.year as year, t.genre as genre, t.length as
sb = criteria.AddSbCriteria(sb);

// Adds search criteria
criteriaList.add(e: "t.deletedate is null");
criteria.AddCriteriaList(criteriaList, parameterMap);

// Adds sort criteria
SortCriteria sortCriteria = new SortCriteria("");
criteria.AddSortCriteria(sortCriteria);
return new QueryParam(sb.toString(), criteriaList, parameterMap, sortCriteria, filterCriteria,
        new TrackDtoMapper());
```

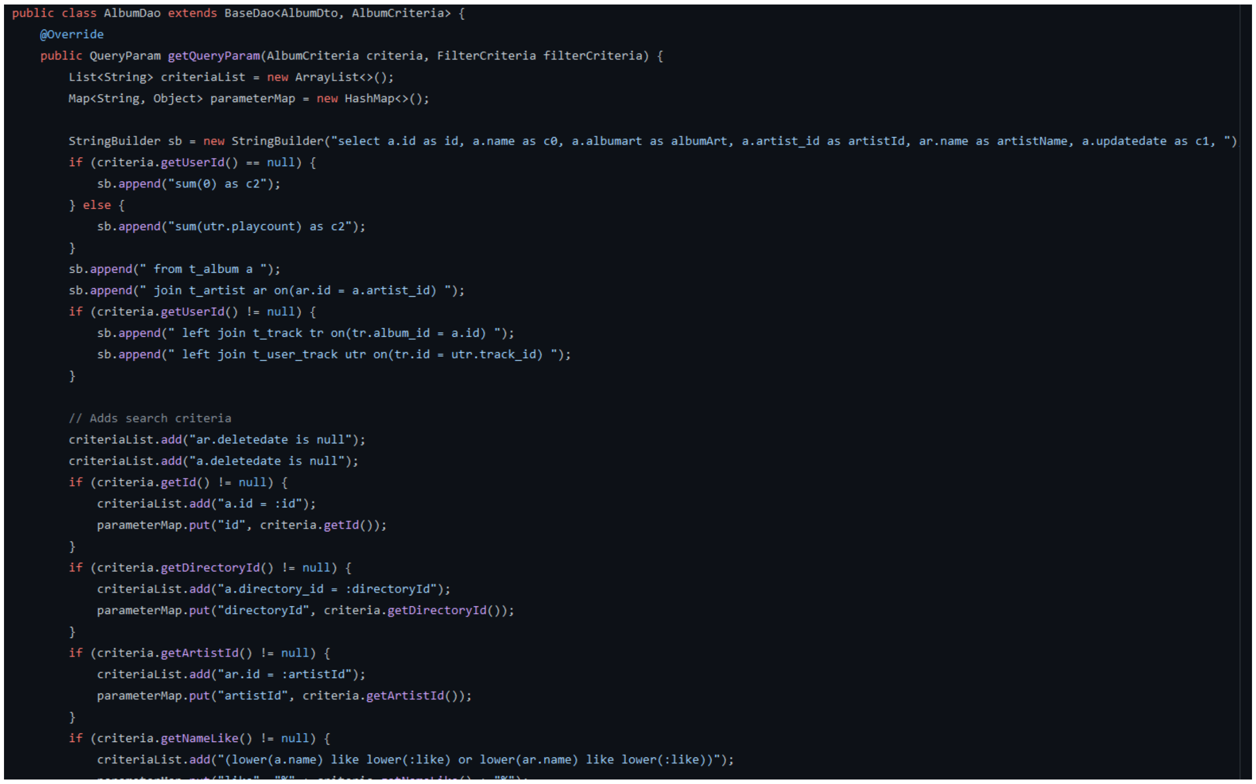


```java
public StringBuilder AddSbCriteria(StringBuilder sb) {
    if (getUserId() != null) {
        sb.append(str: " ut.playcount as userTrackPlayCount, ut.liked userTrackLike, ");
    } else {
        sb.append(str: " 0 as userTrackPlayCount, false as userTrackLike, ");
    }
    sb.append(
            str: " a.id as artistId, a.name as artistName, t.album_id as albumId, alb.name as albumName, alb.albumart as album
    if (getPlaylistId() != null) {
        sb.append(str: "  from t_playlist_track pt, t_track t ");
    } else {
        sb.append(str: "  from t_track t ");
    }
    sb.append(str: "  join t_artist a on(a.id = t.artist_id and a.deletedate is null)");
    sb.append(str: "  join t_album alb on(t.album_id = alb.id and alb.deletedate is null)");
    if (getUserId() != null) {
        sb.append(
                str: "  left join t_user_track ut on(ut.track_id = t.id and ut.user_id = :userId and ut.deletedate is null)");
    }
    return sb;
}

public void AddCriteriaList(List<String> criteriaList, Map<String, Object> parameterMap) {

    if (getPlaylistId() != null) {
        criteriaList.add(e: "pt.track_id = t.id");
        criteriaList.add(e: "pt.playlist_id = :playlistId");
        parameterMap.put(key: "playlistId", getPlaylistId());
    }
    if (getAlbumId() != null) {
        criteriaList.add(e: "t.album_id = :albumId");
        parameterMap.put(key: "albumId", getAlbumId());
    }
    if (getDirectoryId() != null) {
        criteriaList.add(e: "alb.directory_id = :directoryId");
        parameterMap.put(key: "directoryId", getDirectoryId());
    }
    if (getArtistId() != null) {
        criteriaList.add(e: "a.id = :artistId");
        parameterMap.put(key: "artistId", getArtistId());
    }
    if (getTitle() != null) {
        criteriaList.add(e: "lower(t.title) like lower(:title)");
        parameterMap.put(key: "title", getTitle());
```

2) Insufficient Modularization (AlbumDao):

- Sonarqube: The class AlbumDao has a high cognitive complexity, which indicates a lot of if else statements, now on observation we found that it uses methods of criteria class around 14 times through if statements, which also indicates feature envy code smell.
- Bloated Method: The method getQueryParam is quite long and contains multiple responsibilities like constructing the SQL query string, adding search criteria, constructing the parameter map, etc. These responsibilities could be broken down into smaller methods, making the code more modular and maintainable.
- The presence of a long method can indicate the Large Method design smell, which may indicate that this method is doing too much and could be split into smaller methods that have a single responsibility. This can make the code easier to understand, maintain, and test.
- Another design smell that could be present in this code is the Broken Modularization design smell, as this class may be tightly coupled with the database and SQL queries, and it may not be easy to reuse this code in other parts of the application or to replace the underlying data storage technology.
- To improve this code, it might be a good idea to separate the code that constructs the SQL query from the code that adds search criteria and sorting. This could make the code more modular, easier to test and maintain, and also more reusable across different parts of the application.
- From the code smells, and from the strong indication of coupling, we came to an obvious conclusion that this is a case of Insufficient Modularization design smell. Insufficient Modularization can lead to a lack of scalability and flexibility in the codebase, making it challenging to maintain and extend over time.
- We suggest these conditions to be checked in criteria class itself, to counter these smells.

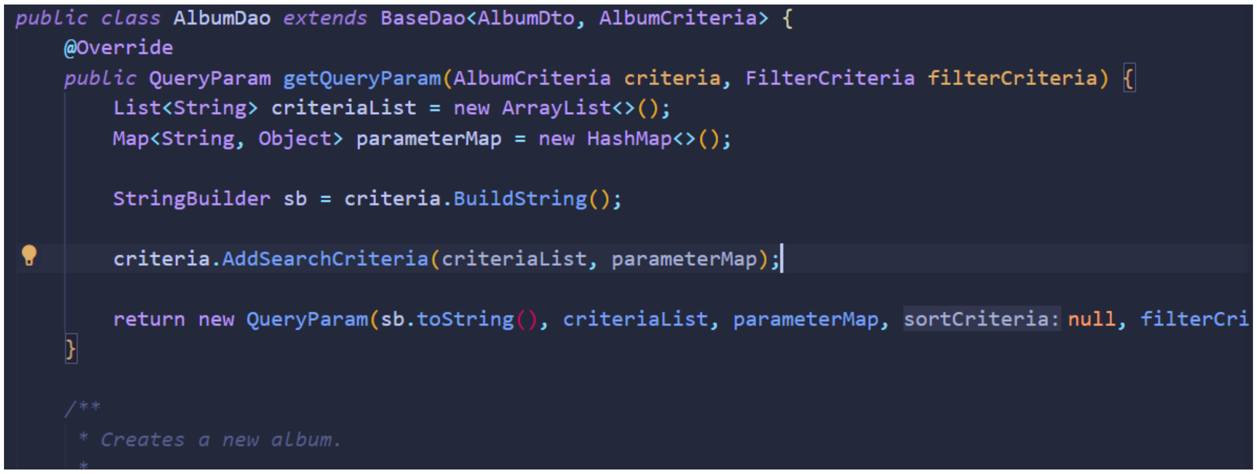
Before Refactoring:



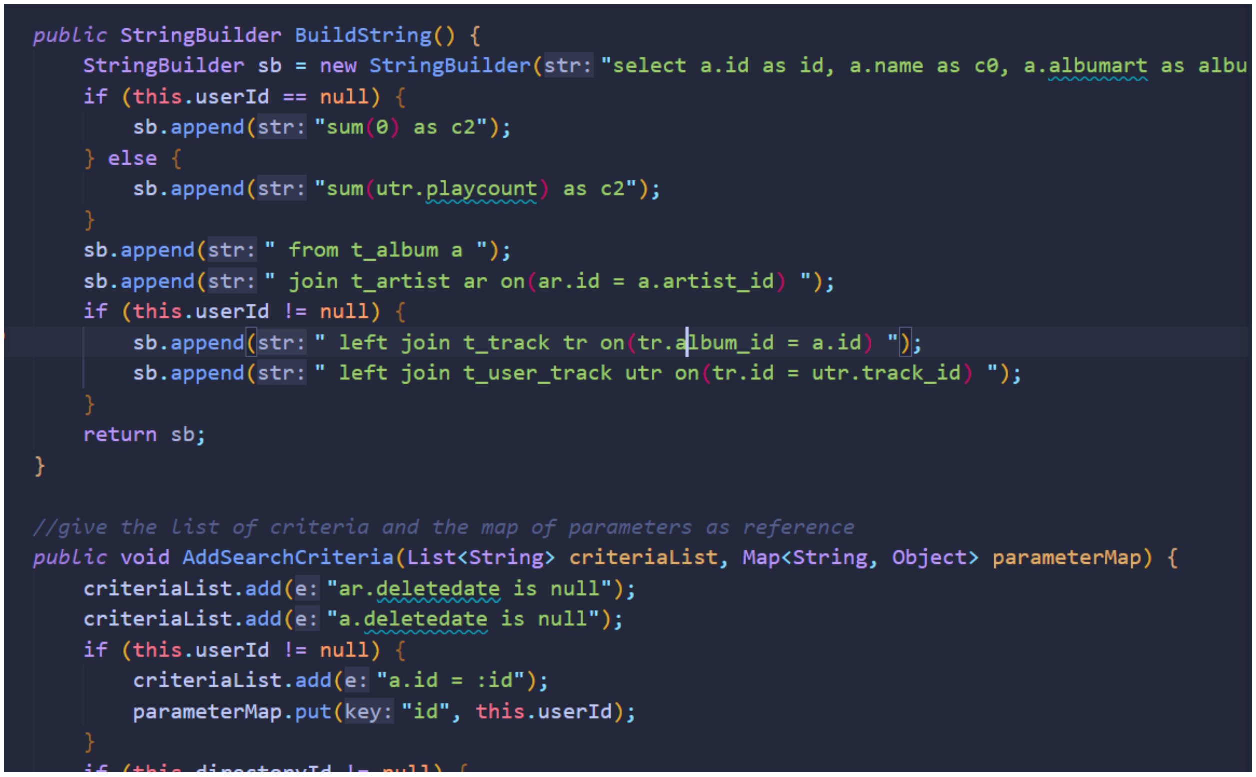
```
public class AlbumDao extends BaseDao<AlbumDto, AlbumCriteria> {
    @Override
    public QueryParam getQueryParam(AlbumCriteria criteria, FilterCriteria filterCriteria) {
        List<String> criteriaList = new ArrayList<>();
        Map<String, Object> parameterMap = new HashMap<>();

        StringBuilder sb = new StringBuilder("select a.id as id, a.name as c0, a.albumart as albumArt, a.artist_id as artistId, ar.name as artistName, a.updatedate as c1, ")
        if (criteria.getUserId() == null) {
            sb.append("sum(0) as c2");
        } else {
            sb.append("sum(utr.playcount) as c2");
        }
        sb.append(" from t_album a ");
        sb.append(" join t_artist ar on(ar.id = a.artist_id) ");
        if (criteria.getUserId() != null) {
            sb.append(" left join t_track tr on(tr.album_id = a.id) ");
            sb.append(" left join t_user_track utr on(tr.id = utr.track_id) ");
        }

        // Adds search criteria
        criteriaList.add("ar.deletedate is null");
        criteriaList.add("a.deletedate is null");
        if (criteria.getId() != null) {
            criteriaList.add("a.id = :id");
            parameterMap.put("id", criteria.getId());
        }
        if (criteria.getDirectoryId() != null) {
            criteriaList.add("a.directory_id = :directoryId");
            parameterMap.put("directoryId", criteria.getDirectoryId());
        }
        if (criteria.getArtistId() != null) {
            criteriaList.add("ar.id = :artistId");
            parameterMap.put("artistId", criteria.getArtistId());
        }
        if (criteria.getNameLike() != null) {
            criteriaList.add("(lower(a.name) like lower(:like) or lower(ar.name) like lower(:like))");
```

After refactoring:

Code has been refactored so that all the attributes of the class AlbumCriteria can be accessed directly in that class, methods BuildString(), AddSearchCriteria() are added to AlbumCriteria we can be called by AlbumDao for their respective operations in build queries.



```
public class AlbumDao extends BaseDao<AlbumDto, AlbumCriteria> {
    @Override
    public QueryParam getQueryParam(AlbumCriteria criteria, FilterCriteria filterCriteria) {
        List<String> criteriaList = new ArrayList<>();
        Map<String, Object> parameterMap = new HashMap<>();

        StringBuilder sb = criteria.BuildString();

        criteria.AddSearchCriteria(criteriaList, parameterMap);

        return new QueryParam(sb.toString(), criteriaList, parameterMap, sortCriteria: null, filterCri
    }

    /**
     * Creates a new album.
     *
```



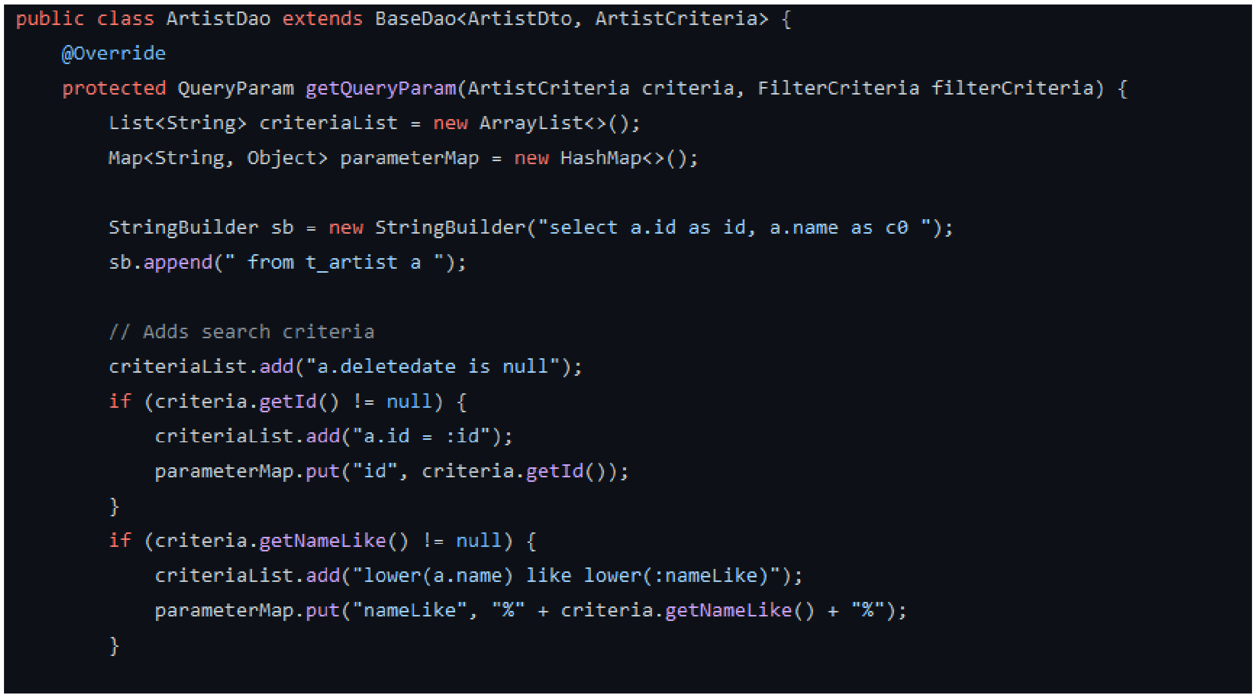
```
public StringBuilder BuildString() {
    StringBuilder sb = new StringBuilder(str: "select a.id as id, a.name as c0, a.albumart as albu
    if (this.userId == null) {
        sb.append(str: "sum(0) as c2");
    } else {
        sb.append(str: "sum(utr.playcount) as c2");
    }
    sb.append(str: " from t_album a ");
    sb.append(str: " join t_artist ar on(ar.id = a.artist_id) ");
    if (this.userId != null) {
        sb.append(str: " left join t_track tr on(tr.album_id = a.id) ");
        sb.append(str: " left join t_user_track utr on(tr.id = utr.track_id) ");
    }
    return sb;
}

//give the list of criteria and the map of parameters as reference
public void AddSearchCriteria(List<String> criteriaList, Map<String, Object> parameterMap) {
    criteriaList.add(e: "ar.deletedate is null");
    criteriaList.add(e: "a.deletedate is null");
    if (this.userId != null) {
        criteriaList.add(e: "a.id = :id");
        parameterMap.put(key: "id", this.userId);
    }
    if (this.directoryId != null) {
```

3) Insufficient Modularization(ArtistDao):

- Sonarqube: The class ArtistDao has a high cognitive complexity, which indicates a lot of if else statements, now on observation we found that it uses methods of criteria class around 12 times through if statements, which also indicates feature envy code smell.
- Bloated Method: The method getQueryParam is quite long and contains multiple responsibilities like constructing the SQL query string, adding search criteria, constructing the parameter map, etc. These responsibilities could be broken down into smaller methods, making the code more modular and maintainable.
- The presence of a long method can indicate the Large Method design smell, which may indicate that this method is doing too much and could be split into smaller methods that have a single responsibility. This can make the code easier to understand, maintain, and test.
- Another design smell that could be present in this code is the Broken Modularization design smell, as this class may be tightly coupled with the database and SQL queries, and it may not be easy to reuse this code in other parts of the application or to replace the underlying data storage technology.
- To improve this code, it might be a good idea to separate the code that constructs the SQL query from the code that adds search criteria and sorting. This could make the code more modular, easier to test and maintain, and also more reusable across different parts of the application.

- From the code smells, and from the strong indication of coupling, we came to an obvious conclusion that this is a case of Insufficient Modularization design smell. Insufficient Modularization can lead to a lack of scalability and flexibility in the codebase, making it challenging to maintain and extend over time.
- We suggest these conditions to be checked in criteria class itself, to counter these smells.

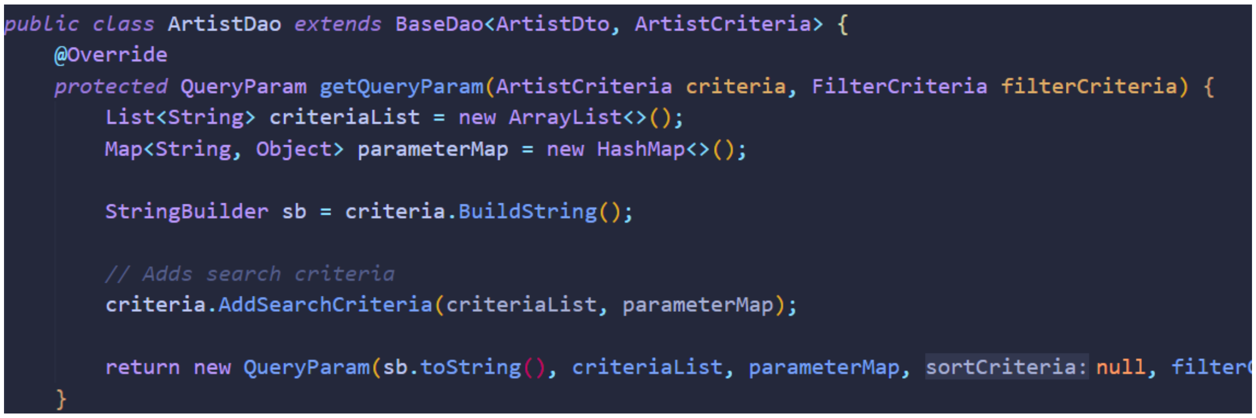
Before Refactoring:



```java
public class ArtistDao extends BaseDao<ArtistDto, ArtistCriteria> {
    @Override
    protected QueryParam getQueryParam(ArtistCriteria criteria, FilterCriteria filterCriteria) {
        List<String> criteriaList = new ArrayList<>();
        Map<String, Object> parameterMap = new HashMap<>();

        StringBuilder sb = new StringBuilder("select a.id as id, a.name as c0 ");
        sb.append(" from t_artist a ");

        // Adds search criteria
        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() + "%");
        }
```

After refactoring:

Code has been refactored so that all the attributes of the class ArtistCriteria can be accessed directly in that class, methods BuildString(), AddSearchCriteria() are added to ArtistCriteria we can be called by ArtistDao for their respective operations in build queries.

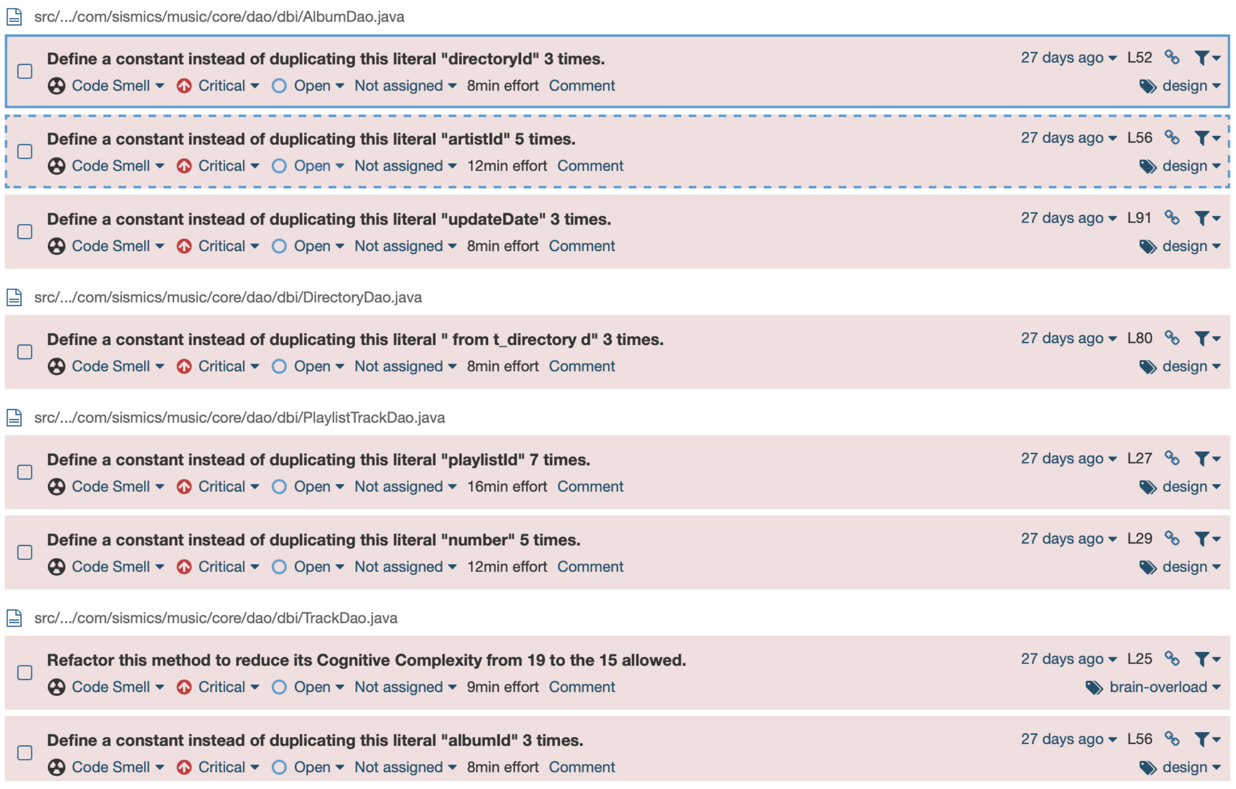


```java
public class ArtistDao extends BaseDao<ArtistDto, ArtistCriteria> {
    @Override
    protected QueryParam getQueryParam(ArtistCriteria criteria, FilterCriteria filterCriteria) {
        List<String> criteriaList = new ArrayList<>();
        Map<String, Object> parameterMap = new HashMap<>();

        StringBuilder sb = criteria.BuildString();

        // Adds search criteria
        criteria.AddSearchCriteria(criteriaList, parameterMap);

        return new QueryParam(sb.toString(), criteriaList, parameterMap, sortCriteria: null, filterC
    }
```

4) Lack of Abstraction:

- The "lack of abstraction" design smell occurs when a software design does not have enough levels of abstraction to effectively represent the problem domain or implement the required functionality. This can lead to code that is difficult to understand, modify, and maintain.
- Abstraction is a fundamental concept in software design that involves hiding implementation details and exposing only the necessary information to the user. This can help to simplify complex systems and make them more manageable.
- In the case of a lack of abstraction design smell, there may be too much low-level detail exposed to the user, making it difficult to see the big picture or understand how the system works as a whole. Alternatively, there may be too much complexity hidden away in the implementation, making it difficult to modify or extend the system.
- We found multiple cases of this design smell and we mentioned one below
- In the UserDao.java file, there are instances where the string literal has been used multiple times, hence we can initialize variables and hide the exact implementation details and use these variables in place of them.

Analysis: We found this smell from sonarqube, it gave various instances of duplicate literals giving the low level implementation details, it is quite understandable from the picture provided below.


src/.../com/sismics/music/core/dao/dbi/AlbumDao.java

Define a constant instead of duplicating this literal "directoryId" 3 times. 27 days ago L52
Code Smell Critical Open Not assigned 8min effort Comment design

Define a constant instead of duplicating this literal "artistId" 5 times. 27 days ago L56
Code Smell Critical Open Not assigned 12min effort Comment design

Define a constant instead of duplicating this literal "updateDate" 3 times. 27 days ago L91
Code Smell Critical Open Not assigned 8min effort Comment design

src/.../com/sismics/music/core/dao/dbi/DirectoryDao.java

Define a constant instead of duplicating this literal " from t_directory d" 3 times. 27 days ago L80
Code Smell Critical Open Not assigned 8min effort Comment design

src/.../com/sismics/music/core/dao/dbi/PlaylistTrackDao.java

Define a constant instead of duplicating this literal "playlistId" 7 times. 27 days ago L27
Code Smell Critical Open Not assigned 16min effort Comment design

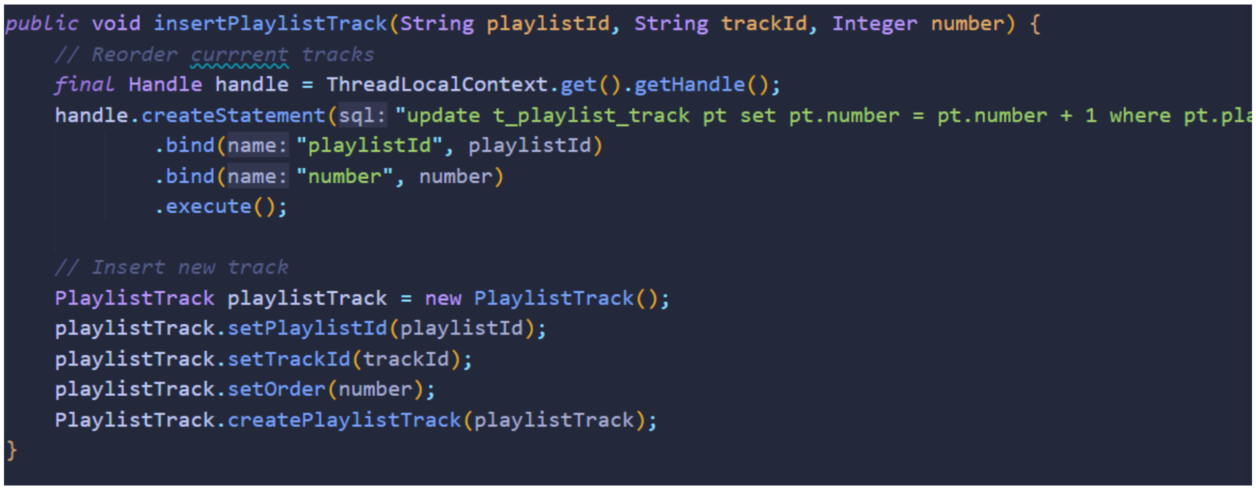
Define a constant instead of duplicating this literal "number" 5 times. 27 days ago L29
Code Smell Critical Open Not assigned 12min effort Comment design

src/.../com/sismics/music/core/dao/dbi/TrackDao.java

Refactor this method to reduce its Cognitive Complexity from 19 to the 15 allowed. 27 days ago L25
Code Smell Critical Open Not assigned 9min effort Comment brain-overload

Define a constant instead of duplicating this literal "albumId" 3 times. 27 days ago L56
Code Smell Critical Open Not assigned 8min effort Comment design

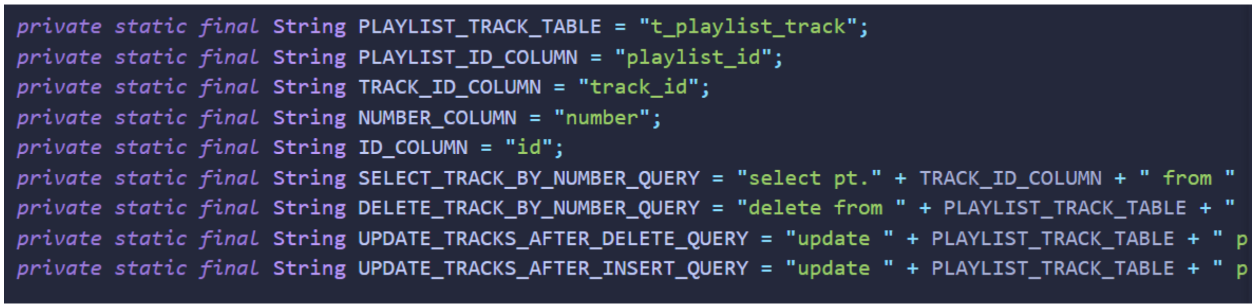
For example, in playlistrackdao.java low level implementation makes it difficult to read and understand, and also to modify.



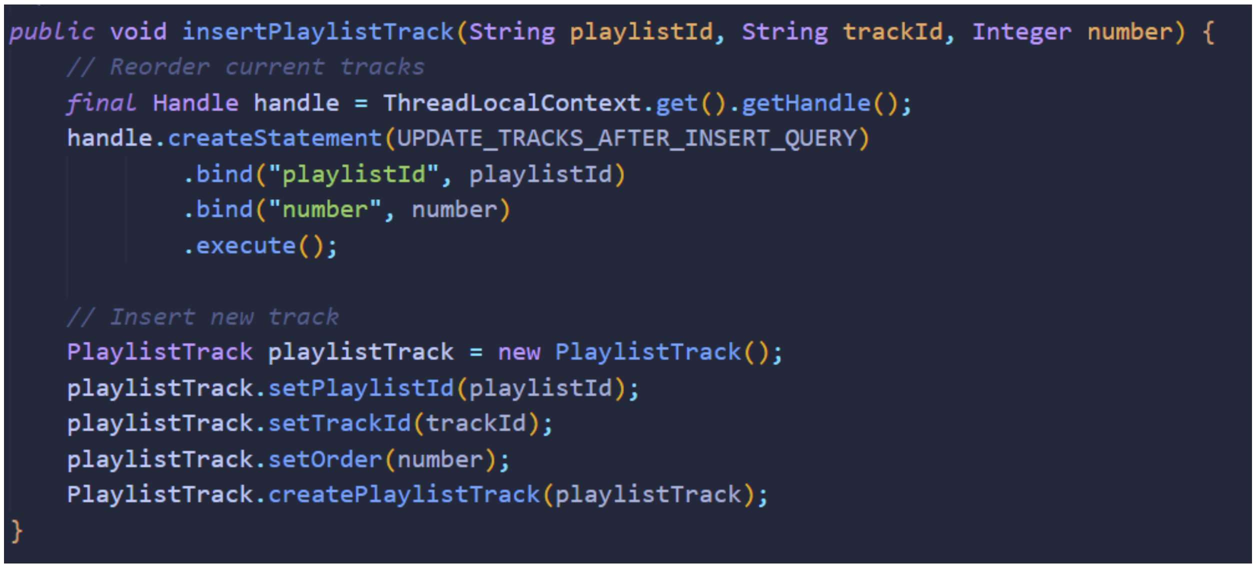
```
public void insertPlaylistTrack(String playlistId, String trackId, Integer number) {
    // Reorder currrent tracks
    final Handle handle = ThreadLocalContext.get().getHandle();
    handle.createStatement(sql: "update t_playlist_track pt set pt.number = pt.number + 1 where pt.pla
            .bind(name: "playlistId", playlistId)
            .bind(name: "number", number)
            .execute();

    // Insert new track
    PlaylistTrack playlistTrack = new PlaylistTrack();
    playlistTrack.setPlaylistId(playlistId);
    playlistTrack.setTrackId(trackId);
    playlistTrack.setOrder(number);
    PlaylistTrack.createPlaylistTrack(playlistTrack);
}
```

Refactored code:

String literals are replaced by constants which hide the low level implementation details as well as make the code more readable.



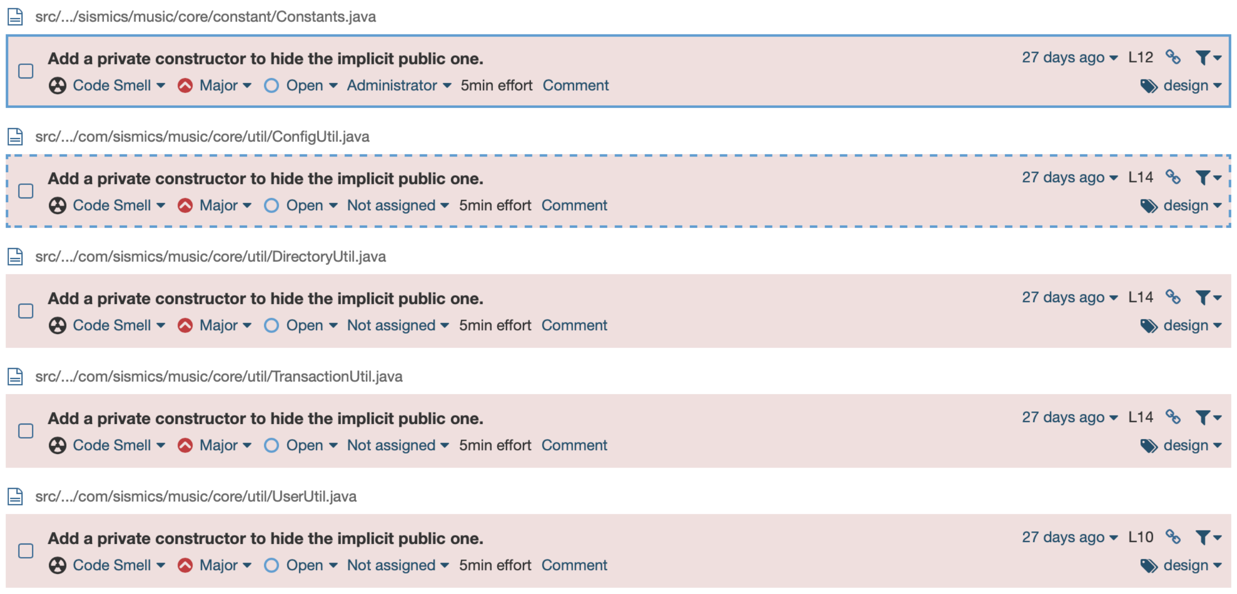
```
private static final String PLAYLIST_TRACK_TABLE = "t_playlist_track";
private static final String PLAYLIST_ID_COLUMN = "playlist_id";
private static final String TRACK_ID_COLUMN = "track_id";
private static final String NUMBER_COLUMN = "number";
private static final String ID_COLUMN = "id";
private static final String SELECT_TRACK_BY_NUMBER_QUERY = "select pt." + TRACK_ID_COLUMN + " from "
private static final String DELETE_TRACK_BY_NUMBER_QUERY = "delete from " + PLAYLIST_TRACK_TABLE + "
private static final String UPDATE_TRACKS_AFTER_DELETE_QUERY = "update " + PLAYLIST_TRACK_TABLE + " p
private static final String UPDATE_TRACKS_AFTER_INSERT_QUERY = "update " + PLAYLIST_TRACK_TABLE + " p
```



```
public void insertPlaylistTrack(String playlistId, String trackId, Integer number) {
    // Reorder current tracks
    final Handle handle = ThreadLocalContext.get().getHandle();
    handle.createStatement(UPDATE_TRACKS_AFTER_INSERT_QUERY)
            .bind("playlistId", playlistId)
            .bind("number", number)
            .execute();

    // Insert new track
    PlaylistTrack playlistTrack = new PlaylistTrack();
    playlistTrack.setPlaylistId(playlistId);
    playlistTrack.setTrackId(trackId);
    playlistTrack.setOrder(number);
    PlaylistTrack.createPlaylistTrack(playlistTrack);
}
```

5) Insufficient Encapsulation

- Insufficient encapsulation is a design smell that occurs when an object or module exposes its internal details, data, or behavior in a way that is not necessary or safe. This can lead to a number of problems, including increased coupling, reduced flexibility, and difficulty maintaining the code.
- Encapsulation is a key principle of object-oriented design that involves hiding the internal details of an object or module and exposing only a well-defined interface or set of behaviors. This helps to promote modular design, reduce coupling, and improve code maintainability.



src/.../sismics/music/core/constant/Constants.java

Add a private constructor to hide the implicit public one. 27 days ago L12
Code Smell Major Open Administrator 5min effort Comment design

src/.../com/sismics/music/core/util/ConfigUtil.java

Add a private constructor to hide the implicit public one. 27 days ago L14
Code Smell Major Open Not assigned 5min effort Comment design

src/.../com/sismics/music/core/util/DirectoryUtil.java

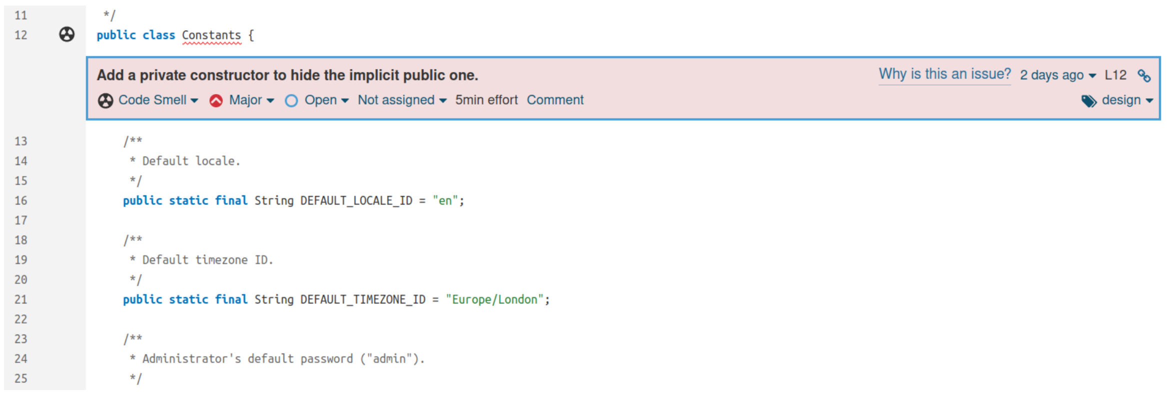
Add a private constructor to hide the implicit public one. 27 days ago L14
Code Smell Major Open Not assigned 5min effort Comment design

src/.../com/sismics/music/core/util/TransactionUtil.java

Add a private constructor to hide the implicit public one. 27 days ago L14
Code Smell Major Open Not assigned 5min effort Comment design

src/.../com/sismics/music/core/util/UserUtil.java

Add a private constructor to hide the implicit public one. 27 days ago L10
Code Smell Major Open Not assigned 5min effort Comment design

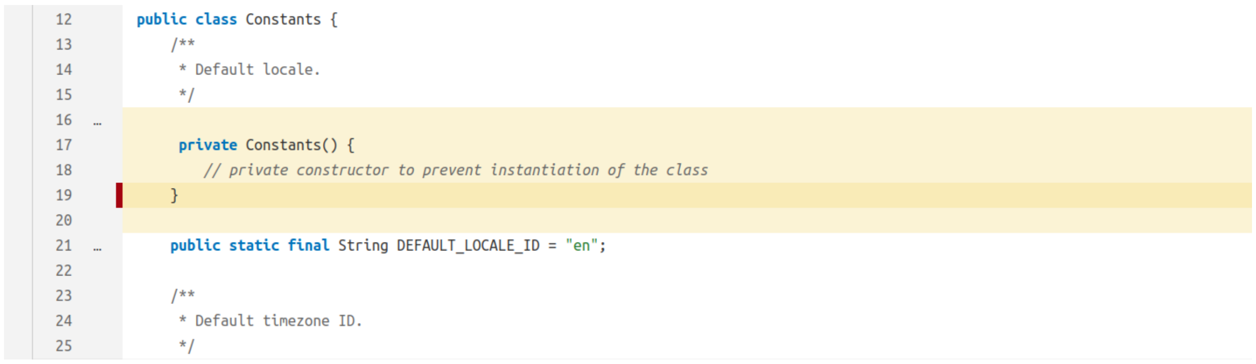


```
 */
public class Constants {
```

Add a private constructor to hide the implicit public one. Why is this an issue? 2 days ago L12
Code Smell Major Open Not assigned 5min effort Comment design

```
    /**
     * Default locale.
     */
    public static final String DEFAULT_LOCALE_ID = "en";

    /**
     * Default timezone ID.
     */
    public static final String DEFAULT_TIMEZONE_ID = "Europe/London";

    /**
     * Administrator's default password ("admin").
     */
```

Post-refactoring:

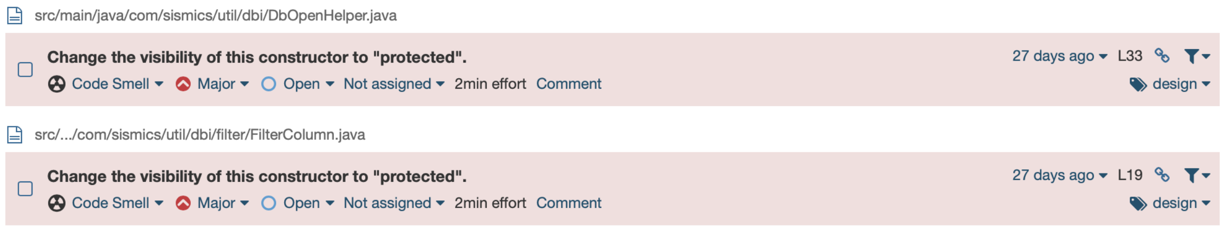

```
public class Constants {
    /**
     * Default locale.
     */

     private Constants() {
        // private constructor to prevent instantiation of the class
    }

    public static final String DEFAULT_LOCALE_ID = "en";

    /**
     * Default timezone ID.
     */
```

Prior to refactoring, the default access modifier of the class Constants was in use. This implied that any class could access the constructor method, retrieving details of objects of the Constants class.
To prevent instantiation of the class, the access modifier of the constructor is now modified to private, fixing insufficient encapsulation and ensuring a level of security to the details of the objects meant to remain constant.
The said smell is detected in multiple locations of the codebase, and has been rectified at several instances.

6) Missing encapsulation:

If the principle of encapsulation is completely fulfilled, the data and methods that operate on particular data are ideally clustered into a single unit, such as a class.
During missing encapsulation, the principle is violated, and the internal state oif some object is directly exposed to other classes.

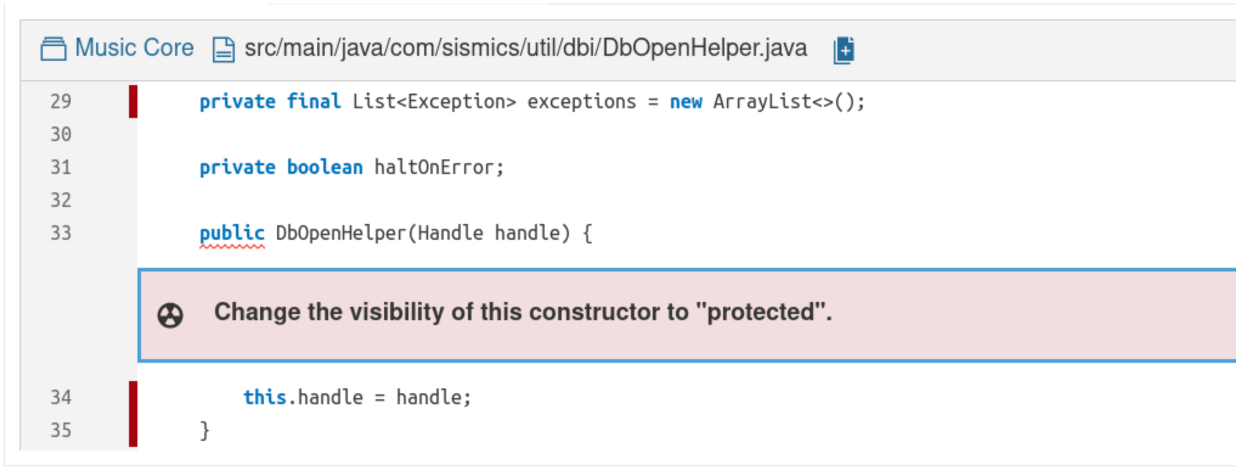
Initially, the constructor of the DbOpenHelper class is public. The constructor may thus be accessed by any class, irrespective of whether it is inherited from the DbOpenHelper class.


src/main/java/com/sismics/util/dbi/DbOpenHelper.java

Change the visibility of this constructor to "protected". 27 days ago L33
Code Smell Major Open Not assigned 2min effort Comment design

src/.../com/sismics/util/dbi/filter/FilterColumn.java

Change the visibility of this constructor to "protected". 27 days ago L19
Code Smell Major Open Not assigned 2min effort Comment design


Music Core src/main/java/com/sismics/util/dbi/DbOpenHelper.java

```
    private final List<Exception> exceptions = new ArrayList<>();

    private boolean haltOnError;

    public DbOpenHelper(Handle handle) {
```

Change the visibility of this constructor to "protected".

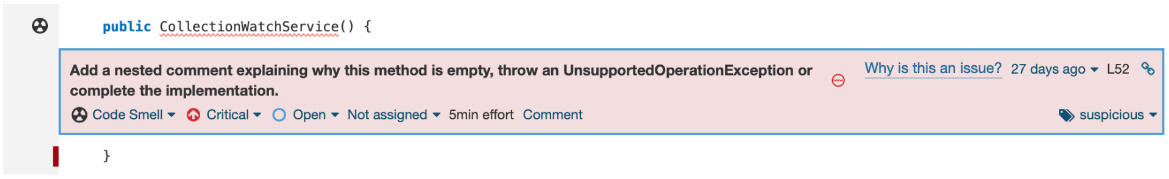
```
        this.handle = handle;
    }
```

Post-refactoring, the access modifier of the constructor is changed to protected, implying that the constructor method may only be run by classes inherited by DbOpenHelper itself.
The said smell is detected in multiple locations of the codebase, and has been rectified at several instances.

7) Empty method:

Empty methods code smells are in every file, and we refactored them.

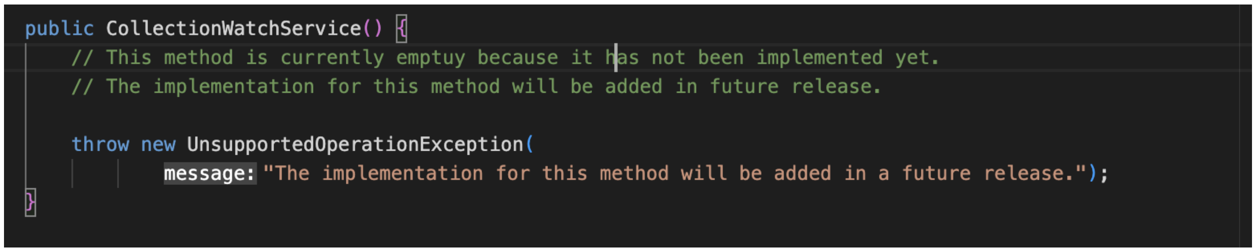
Before refactoring:


```
    public CollectionWatchService() {
```

Add a nested comment explaining why this method is empty, throw an UnsupportedOperationException or complete the implementation.
Why is this an issue? 27 days ago L52
Code Smell Critical Open Not assigned 5min effort Comment suspicious

```
    }
```

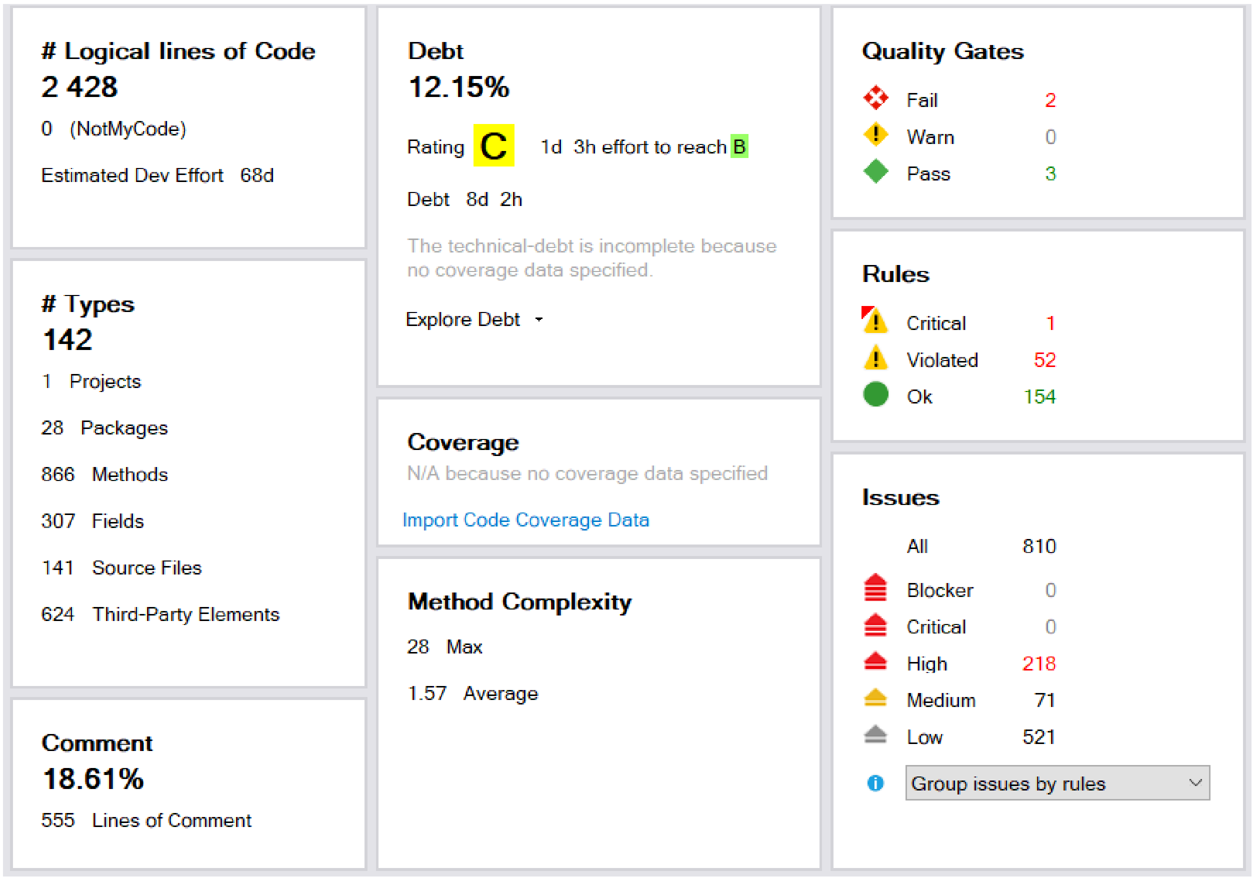
After refactoring:
The smell can be removed by defining a dedicated exception instead of using a generic one.
This helps for clarity of code.


```
public CollectionWatchService() {
    // This method is currently emptuy because it has not been implemented yet.
    // The implementation for this method will be added in future release.

    throw new UnsupportedOperationException(
            message: "The implementation for this method will be added in a future release.");
}
```

TASK 2B & 3B :

JArchitect Metrics

Before refactoring:



# Logical lines of Code
2 428
0 (NotMyCode)
Estimated Dev Effort 68d

# Types
142
1 Projects
28 Packages
866 Methods
307 Fields
141 Source Files
624 Third-Party Elements

Comment
18.61%
555 Lines of Comment

Debt
12.15%
Rating C 1d 3h effort to reach B
Debt 8d 2h
The technical-debt is incomplete because no coverage data specified.
Explore Debt

Coverage
N/A because no coverage data specified
Import Code Coverage Data

Method Complexity
28 Max
1.57 Average

Quality Gates

Fail	2
Warn	0
Pass	3

Rules

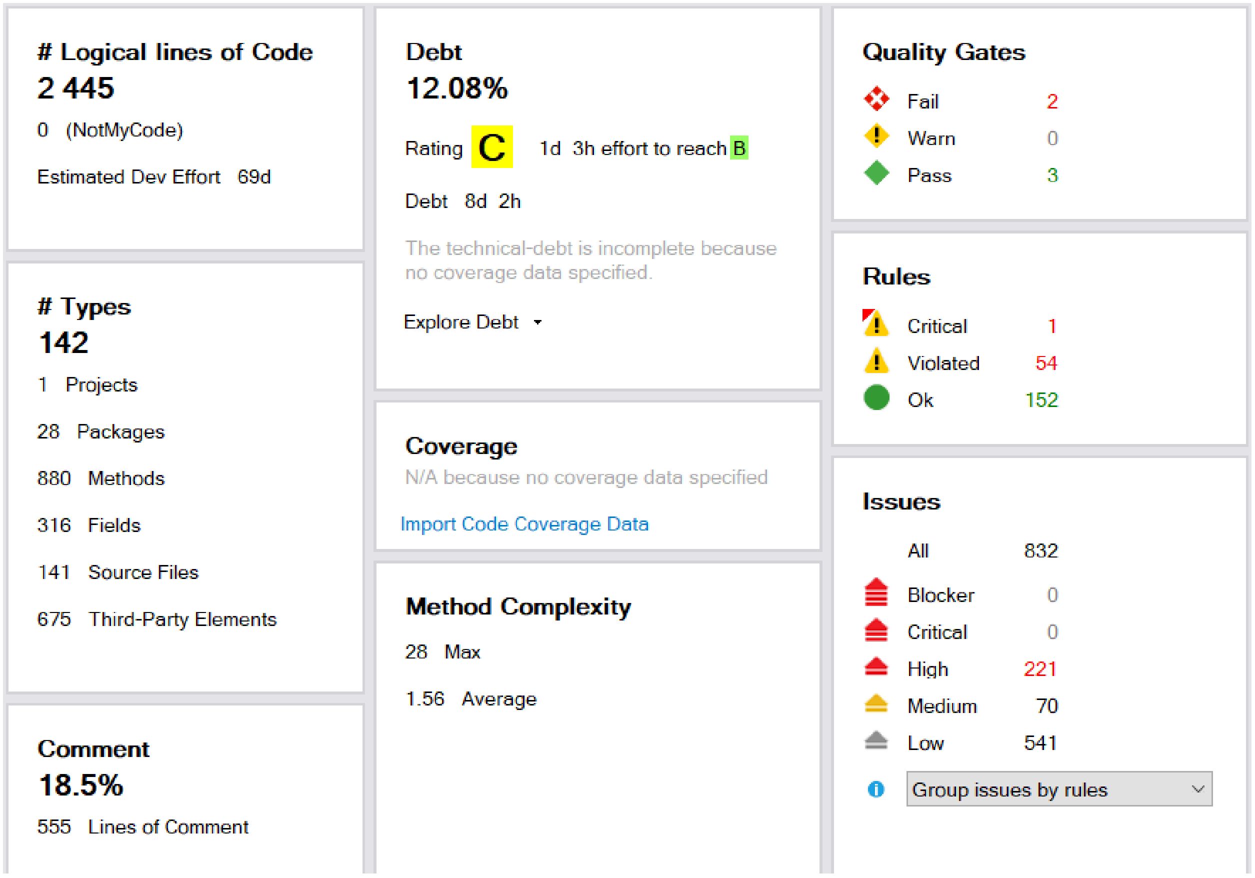
Critical	1
Violated	52
Ok	154

Issues

All	810
Blocker	0
Critical	0
High	218
Medium	71
Low	521

Group issues by rules

After refactoring:



Logical lines of Code
2 445
0 (NotMyCode)
Estimated Dev Effort 69d

Types
142
1 Projects
28 Packages
880 Methods
316 Fields
141 Source Files
675 Third-Party Elements

Comment
18.5%
555 Lines of Comment

Debt
12.08%
Rating C 1d 3h effort to reach B
Debt 8d 2h
The technical-debt is incomplete because no coverage data specified.
Explore Debt

Coverage
N/A because no coverage data specified
Import Code Coverage Data

Method Complexity
28 Max
1.56 Average

Quality Gates

Fail	2
Warn	0
Pass	3

Rules

Critical	1
Violated	54
Ok	152

Issues

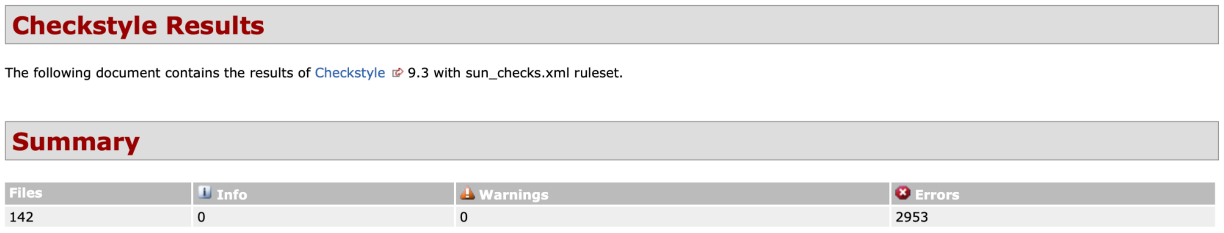
All	832
Blocker	0
Critical	0
High	221
Medium	70
Low	541

Group issues by rules

According to this, technical debt after refactoring decreased from 12.15% to 12.08%

CHECK STYLE METRICS:

Before refactoring:



Checkstyle Results

The following document contains the results of Checkstyle 9.3 with sun_checks.xml ruleset.

Summary

Files	Info	Warnings	Errors
142	0	0	2953

After refactoring:

Checkstyle Results

The following document contains the results of Checkstyle 9.3 with sun_checks.xml ruleset.

Summary

Files	Info	Warnings	Errors
142	0	0	2943