Project 1 Report

Task 1: Mining

We identified the major classes present in the repository and have document their functionality and behaviour in 5 main sections.

DAO

UserDao : Class that exists for the purpose of accessing the user-data from the database.

UserCriteria: Class the exists to check if the user has been registered in LastFM.

UserAlbumDao: Creation of a link between the user and the album, such that a relationship is established between the two.

example -

ThreadLocalContext.get().getHandle();
handle.createStatement("insert into " +
" t_user_album(id, user_id, album_id, createdate)" +
" values(:id, :userId, :albumId, :createDate)")

This implies that we insert into the function t_user_album the values userId, albumId and the date in which the relationship was established, along with an unique ID for the same.

UserDto: Access of all the data associated with every individual user occurs in this class.

UserMapper: Correlate the user details to a map like structure for easy access from the data-base.

example -

public class UserMapper extends BaseResultSetMapper<User>

UserTrackMapper: Correlate the details of User/track relationship to a map like structure for easy access to data base.

AuthenticationTokenMapper: Map details of every log-in session to a map like structure.

PlayerMapper : Get ‘id’ of player element in current use.

LocaleDao : Get ‘id’ of locale from ‘id’ from a different object.

ArtistMapper : Extends mapper data structure to include details of artist for a particular album/track. Can also be traced on an ‘id’ basis.

AlbumMapper : Insertion of album into the map structure and creation of album map database.

PlaylistTrackMapper : Insertion of playlist/track relationship into a map structure (since every playlist will have to contain tracks) and creation of the data base for the same.

PlaylistMapper : Insertion of playlist into a map structure, and creation of the database for the same.

TrackDtoMapper : Accessing Tracks from the map structure according to the details parsed through the *data transfer* function.

DirectoryMapper : Inserting directory entries into the map structure, along with instantiation of a directory map structure.

ArtistDtoMapper : Accessing Artist from the map structure according to the details parsed through the *data transfer* function.

AlbumDtoMapper : Accessing Album from the map structure according to the details parsed through the *data transfer* function.

AlbumDto : Accessing individual album details/data points for every individual entry.

ArtistDto : Accessing individual artist details for every individual entry.

TrackDto : Accessing individual track details for every individual entry.

PlaylistDto : Accessing individual playlist details for every individual entry.

AlbumCriteria : Returning details of individual albums on the basis of data entries such as id, etc.

TrackCriteria : Returning details of individual tracks on the basis of data entries such as id, etc.

PlaylistCriteria : Returning details of individual playlists on the basis of data entries such as id, etc.

ArtistCriteria : Returning details of individual artists on the basis of data entries such as id, etc.

TranscoderDao : Creates new transcoder element.

PlaylistTrackDao : Creates new playlist/track relationship element.

ArtistDao : Creates new artist element. Along with all CRUD operations on the element.

AlbumDao : Creates new album element. Along with all CRUD operations on the element.

PlaylistDao : Creates new playlist element. Along with all CRUD operations on the element.

DirectoryDao : Creates new directory element. Along with all CRUD operations on the element.

TrackDao : Creates new track element. Along with all CRUD operations on the element.

TranscoderMapper : Uploads transcoder element to a map structure consisting of transcoders.

RolePrivilegeDao : Create privileges for certain roles (relationship). Update the roles of users/administrators.

RolePrivilegeMapper : Uploads and retrieves role privileges relationships to maps consisting of role privilege elements.

RoleMapper : Map roles to users and upload the same to a map like structure that consists of such mappings.

PrivilegeMapper : Map privileges to users on the basis of details like their ids.

Listener

LastFmUpdateTrackPlayCountAsyncListener : Update the play count of a track, consisting of only LastFM users. (Upon access of the track, if the user has lastFM, the count is updated periodically)

LastFmUpdateLovedTrackAsyncListener : Update the love count of a track, consisting of only LastFM users. (Upon access of the track, if the user has lastFM, the love is updated if the user loves the track - periodically)

DirectoryCreatedAsyncListener : Class for creating directory.

DirectoryDeletedAsyncListener: Class for deleting directory.

TrackLikedAsyncListener : User has the ability to like a track.

TrackUnlikedAsyncListener : User has the ability to unlike a previously liked track.

DeadEventListener : Listens for dead events across listener processes and kills them if needed.

Service

LastFmsService : Creation of last fms login sessions. Gives added LastFM functionality.

AlbumArtFilenameFilter : Filters and cleans up the names of album art.

AlbumArtimporter : Getting album art file from directory.

AlbumArtService : Imports the same into the user services.

TranscoderService : Enables Transcoder.

PlayerStatus : Checks for player status.

PlayerService : Allows the user to access player services.

I**mportAudioService** : Importing audio services from directory.

ImportAudioFile : Import independent audio files.

ImportAudio : Import audio as a group of files.

CollectionService : Create, read, update and delete and collection of tracks (directories) from an index.

CollectionWatchService : Enables methods to create and delete watch services for a directory.

CollectionVisitor : Tracks the visiting of file from a visitor.

Model

User : Creation of the user entity.

Artist : Creation of an artist entity.

Player : Creation of a player entity.

Role : Creation of a role element that can then be associated with an user element.

UserTrack : Modelling of the relationship between User and Track.

Track : Creation of the track entity.

RolePrivilege : Modelling the relationship between roles and privileges.

Locale : Modelling locale entity

Privilege : Creation of privileges

AuthenticationToken : Modelling individual AuthenticationToken element

Directory : Creation of directory entity.

Transcoder : Creation of a transcoder entity.

UserAlbum : Modelling the relationship between User and Album.

Config : Configuring basic elements common to all classes.

Util

ImageUtil : Utility class that exists for managing independent image files. CRUD operations take place for images take place in this class.

DirectoryUtil : Same as the imageUtil class, except it does the same for directories.

UserUtil : Same as the imageUtil class, except it does the same for users.

ConfigUtil : Configure base utilities for all classes.

TransactionUtil : Util class for transaction (DTO).

QueryParam : Query parameters for all classes.

SortCriteria : Sorting elements on the basis of their 'criteria' (accessed from criteria classes in DAO)

PaginatedList : Construction, definition and limitation of page elements. Stores total number of records and other meta data.

PaginatedLists : Creates a list of pages with default params or edited params depending on user usage.

NOTE: The UML diagrams have been provided in the `docs` directory.

Task 2a: Design Smells

We have identified some main design smells in the code which have been listed below. We used `SonarQube` to identify code smells and the supporting code smell has been listed in the associated design smell heading.

Deficient Encapsulation

This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required.

The class `columnindexmapper` has the `deficient encapsulation` smell because the class exposes fields (`INSTANCE`) belonging to it with public accessibility which is more than required.

In our refactored code, we have changed that accessibility to private and the method `mapResultSet` was extracted from the `map` method, making the `map` method more readable and easier to understand. The `mapResultSet` method was also made private, as it is only used within the `ColumnIndexMapper` class, making the encapsulation more secure.

The refactored code is provided below:

```
package com.sismics.music.core.util.dbi;

import org.skife.jdbi.v2.StatementContext;
import org.skife.jdbi.v2.exceptions.ResultSetException;
import org.skife.jdbi.v2.tweak.ResultSetMapper;
```

```
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

/**
 * Maps the results set to an indexed array.
 *
 * @author jtremeaux
 */
private class ColumnIndexMapper implements ResultSetMapper<Object[]> {
    /**
     * An instance of ColumnIndexMapper.
     */
    public static final ColumnIndexMapper INSTANCE = new ColumnIndexMapper();

    /**
     * Maps the result set to an indexed array.
     *
     * @param index The row number
     * @param r The result set
     * @param ctx The statement context
     * @return The indexed array
     */
    public Object[] map(int index, ResultSet r, StatementContext ctx) {
        ResultSetMetaData m;
        try {
            m = r.getMetaData();
        } catch (SQLException e) {
            throw new ResultSetException("Unable to obtain metadata from result set", e, ctx);
        }

        Object[] row = null;
        try {
            row = mapResultSet(r, m);
        } catch (SQLException e) {
            throw new ResultSetException("Unable to access specific metadata from " +
                    "result set metadata", e, ctx);
        }
        return row;
    }

    /**
     * Maps the result set to an indexed array.
     *
     * @param r The result set
     * @param m The result set meta data
     * @return The indexed array
     * @throws SQLException If the result set cannot be accessed
     */
    private Object[] mapResultSet(ResultSet r, ResultSetMetaData m) throws SQLException {
        Object[] row = new Object[m.getColumnCount()];
        for (int i = 1; i <= m.getColumnCount(); i++) {
            row[i - 1] = r.getObject(i);
        }
        return row;
    }
}
```

NOTE: This code has also been updated in the actual repository.

Insufficient Modularization

This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both.

The method `readTrackMetadata` in the class `CollectionService` had the `longMethod` smell according to SonarQube as its cognitive complexity is too high and should be reduced. This same smell was found in methods pertaining to classes `TrackDao`, `CollectionWatchService`, `ImportAudioService`, `ImageUtil`, and `ResourceUtil`.

Using this, we identified certain classes that had a bloated interface (large number of public methods). Some of these classes are:

1. `User`: 27 public methods
2. `Album`: 23 public methods
3. `Track`: 33 public methods
4. `ImportAudio`: 21 public methods
5. `TrackDto`: 34 public methods

To refactor this classes and improve modularization, one approach would be to separate the these classes into smaller, more focused classes that each have a single responsibility.

For example, for the `TrackDto` class, we could have a class for basic track information (id, title, filename, year, genre, length, bitrate), a class for user track information (userTrackPlayCount, userTrackLike), and a class for album and artist information (artistId, artistName, albumId, albumName, albumArt).

This would result in the following code:

```
package com.sismics.music.core.dao.dbi.dto;

public class BasicTrackInfoDto {
    private String id;
    private String fileName;
    private String title;
    private Integer year;
    private String genre;
    private Integer length;
    private Integer bitrate;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
```

```
    public String getFileName() {
        return fileName;
    }

    public void setFileName(String fileName) {
        this.fileName = fileName;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public Integer getYear() {
        return year;
    }

    public void setYear(Integer year) {
        this.year = year;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }

    public Integer getLength() {
        return length;
    }

    public void setLength(Integer length) {
        this.length = length;
    }

    public Integer getBitrate() {
        return bitrate;
    }

    public void setBitrate(Integer bitrate) {
        this.bitrate = bitrate;
    }
}

public class UserTrackInfoDto {
    private Integer userTrackPlayCount;
    private boolean userTrackLike;

    public Integer getUserTrackPlayCount() {
        return userTrackPlayCount;
    }

    public void setUserTrackPlayCount(Integer userTrackPlayCount) {
        this.userTrackPlayCount = userTrackPlayCount;
    }
```

```
    public boolean isUserTrackLike() {
        return userTrackLike;
    }

    public void setUserTrackLike(boolean userTrackLike) {
        this.userTrackLike = userTrackLike;
    }
}

public class AlbumAndArtistInfoDto {
    private String artistId;
    private String artistName;
    private String albumId;
    private String albumName;
    private String albumArt;

    public String getArtistId() {
        return artistId;
    }

    public void setArtistId(String artistId) {
        this.artistId = artistId;
    }

    public String getArtistName() {
        return artistName;
    }

    public void setArtistName(String artistName) {
        this.artistName = artistName;
    }

    public String getAlbumId() {
        return albumId;
    }

    public void setAlbumId(String albumId) {
        this.albumId = albumId;
    }

    public String getAlbumName() {
        return albumName;
    }

    public void setAlbumName(String albumName) {
        this.albumName = albumName;
    }

    public String getAlbumArt() {
        return albumArt;
    }

    public void setAlbumArt(String albumArt) {
        this.albumArt = albumArt;
    }
}
```

Then, we can combine these smaller classes to create a `TrackDto` class that contains all the information:

```
package com.sismics.music.core.dao.dbi.dto;

public class TrackDto {
    private BasicTrackInfoDto basicTrackInfo;
    private UserTrackInfoDto userTrackInfo;
    private AlbumAndArtistInfoDto albumAndArtistInfo;
}
```

NOTE: The above refactoring is not done in the actual repository as all instances of `TrackDto` objects would need be changed and the tests would need to be rewritten according to the new implementation. Hence, this refactored version has been provided in the `docs/Refactored_Files` directory. Also, the other classes can be broken down in a similar manner.

Cyclically-dependent Modularization

This smell arises when two or more abstractions depend on each other directly or indirectly.

We detected the `cyclically-dependent modularization` smell in the classes `LastFmService`, `AppContext`, `CollectionService`, `CollectionVisitor`, `CollectionWatchService`, `ImportAudioService`, `PlayerService`.

These classes use methods declared in each other's classes and hence all files need to be changed if one is changed. The classes should be contained within their own packages which allows for more organized and modularized code. This also promotes a more single-responsibility approach.

Unutilized Abstraction

This smell arises when an abstraction is left unused (either not directly used or not reachable).

SonarQube showed that some methods were declared but not used. An example of this is the private `assembleResultList` method that was unused in `ArtistDao` and `PlaylistDao`. There were some useless assignments too which were suggested to be removed by SonarQube in classes like `UserMapper` and `UserAlbumMapper`.

This supported our finding that the class `UserUtil` was declared but was unused. This leads to the smell of `unutilized abstraction`. We propose to therefore remove this class.

Task 2b: Code Metrics

We used `Checkstyle` to compute Code Metrics.

Boolean Expression Complexity

Boolean Expression Complexity is a code metric that measures the Complexity of boolean expressions in code. It counts the number of operators and operands in a boolean expression and indicates how difficult the expression is to read and understand. The higher the value of this metric, the more complex the expression is. This metric is useful in identifying code that is difficult to maintain and understand and can be used as a guide for refactoring such code to make it more readable and maintainable.

- **Implications**:
  Boolean Expression Complexity can have several implications for the maintainability of code. Complex boolean expressions can be difficult to read and understand, leading to potential errors as developers misunderstand the code. This can lead to increased debugging time, as well as a potential decrease in the quality of the code. In addition, complex boolean expressions can make it difficult to extend code, as new features may require changes to the expression to be implemented. This can lead to higher development costs and a potential decrease in the quality of the code.
  Finally, complex boolean expressions can lead to code that is difficult to test, as tests may need to be written to cover all possible combinations of the expression. This can lead to an increase in the amount of time spent on testing and a potential decrease in the quality of the code.
- **Effect of Refactoring Process:**
  While Boolean Expression Complexity is an important indicator for refactoring, it did not affect our refactoring as this metric indicates code smells rather than design smells.
- **Analysis:** The Total Boolean Expression Complexity of the code is `48`

Class Data Abstraction Coupling

Class Data Abstraction Coupling is a code metric that measures the coupling between classes in an object-oriented system. It looks at the number of classes each class references and the number of different types of classes referenced by each class. This metric is useful in detecting tight coupling between classes, which can lead to code that is difficult to maintain and understand. It is also useful in identifying classes which are overly reliant on other classes and can be used to guide refactoring efforts to make the code more modular and maintainable.

- **Implications**:
  Class Data Abstraction Coupling can have many implications for the maintainability of

code. Tightly coupled classes can be difficult to maintain, as changes to one class may require changes to multiple other classes to keep the system consistent. This can lead to an increase in the amount of time spent on maintenance and debugging, as well as an increase in the Complexity of the code.
In addition, overly reliant classes can lead to code that is difficult to extend, as new features may require changes to multiple classes to be implemented. This can lead to higher development costs and a potential decrease in the quality of the code.
Finally, class data abstraction coupling can lead to difficult code to test, as tests may need to be written for multiple classes to properly cover the system. This can lead to an increase in the amount of time spent on testing and a potential decrease in the quality of the code.

- **Effect of Refactoring Process:**
  While Class Data Abstraction Coupling is an important indicator for refactoring, it did not affect our refactoring as the design smells we were able to identify did not improve the Coupling in the code.
- **Analysis**: The Total Class Data Abstraction Coupling of the code is `146`

Class Fan Out Complexity

Class Fan-Out Complexity (CFO) is a software metric that measures the number of other classes that a particular class is dependent on. It measures the number of incoming dependencies to a given class, indicating the degree to which the class is coupled to other classes in the system. In other words, the CFO metric counts the number of unique classes that a particular class uses or calls, either directly or indirectly.

- **Implications**:
  - Code Coupling: High CFO can lead to tight coupling between classes, which makes it harder to maintain or modify the code. When classes are tightly coupled, changes in one class can have a cascading effect on other classes. This can make it difficult to change the codebase without introducing unintended consequences.
  - Testing: High CFO can make testing more challenging. If a class depends on many other classes, it can be difficult to test it in isolation. This means that tests for that class may need to involve multiple other classes, which can make tests more complex and time-consuming.
  - Design: High CFO can indicate a problem with the design of the code. It can indicate that the class is trying to do too much, and should be broken down into smaller, more focused classes. This can improve the overall structure of the code and make it easier to maintain and modify.

    - Dependency management: High CFO can make it more challenging to manage dependencies between classes. It can be difficult to keep track of which classes are dependent on others, and changes to one class can impact multiple other classes. This can make it more challenging to manage the evolution of the code over time.

- **Effect of Refactoring Process:**
  While trying to backtrack the class hierarchy of the system we found that the CFO of a class and its parent class was identical in several hierarchy levels. This indicated the Cyclically dependent Modularization smell in the codebase.
- **Analysis**: The Total Class Fan Out Complexity of the code is `482`

Cyclomatic Complexity

Cyclomatic Complexity is a software metric that measures the Complexity of a program by counting the number of linearly independent paths through the source code. This metric is useful in identifying complex code, as well as areas of code that may need to be refactored to make it more maintainable.

- **Implications:**
  Cyclomatic Complexity can have several implications for a program. Higher values of this metric can indicate that the code is more complex and difficult to maintain, as changes to one part of the program may have a cascading effect on other parts of the code. This can lead to increased debugging time, as well as a potential decrease in the quality of the code.
  In addition, complex code can make it difficult to extend the program, as new features may require changes to multiple parts of the code to be implemented. This can lead to higher development costs and a potential decrease in the quality of the code.
  Finally, complex code can lead to code that is difficult to test, as tests may need to be written to cover all possible paths through the code. This can lead to an increase in the amount of time spent on testing and a potential decrease in the quality of the code.
- **Effect of Refactoring Process:**
  While trying to find design smells, files with high Cyclomatic Complexity caught our attention. That was one indication to detect the insufficient modularization we detected.
- **Analysis**: The Total Class Fan Out Complexity of the code is `1012`

Java NCSS

Java NCSS (Non-Commenting Source Statements) is a code metric that measures the size of a program by counting the number of source statements in the code. Java NCSS is calculated by counting the number of executable and non-commenting source statements in the code, such as if-statements, for-statements, while-statements, and switch statements. This metric does not consider comments, which are ignored in the calculation. It is useful in determining the Complexity of a program and can be used as a guide to refactoring code to make it more maintainable.

- **Implications**:
  - Code Complexity: NCSS gives an indication of how complex the Java code is, which in turn can impact how easy it is to maintain and modify the code. Generally speaking, the more complex the code, the harder it is to understand and modify.
  - Code Maintainability: Code with a high NCSS value can be more difficult to maintain over time. This is because it is more likely to contain bugs and may require more effort to update when changes are needed.
  - Code Quality: NCSS can be used as a metric for evaluating the quality of Java code. Generally speaking, code with a lower NCSS value is considered to be of higher quality because it is simpler and easier to understand.
  - Code Reviews: NCSS can be used as a tool for conducting code reviews. By analyzing the NCSS value of a code base, developers can identify areas of the code that may require more attention or could benefit from refactoring.
  - Development Efficiency: Reducing the NCSS value of a code base can improve development efficiency. By simplifying the code, developers can reduce the time and effort required to maintain and modify it over time.
- **Effect of Refactoring Process:** While trying to backtrack the execution of the code, we detected some files that were not being run. From the NCSS count we got the indication of Unused Abstraction in the code.
- **Analysis**: The Total Class Fan Out Complexity of the code is `3794`

N Path Complexity

N path complexity is a software metric that is used to evaluate the Complexity of a program by analyzing the number of unique execution paths that can be taken during the execution of the program. It is based on the premise that the more execution paths a program has, the more complex it is and the greater the likelihood of errors or bugs.

- **Implications**:

    - Code complexity: N path complexity is directly related to the Complexity of a program's control flow. As the number of paths increases, so does the Complexity of the code. Programs with a high N path complexity are typically more difficult to understand and modify than those with a low N path complexity.
    - Testability: N path complexity can impact the testability of a program. Programs with a high N path complexity may be more difficult to test thoroughly because more potential paths need to be tested. This can result in reduced test coverage and increase the risk of undetected defects.
    - Maintainability: Programs with a high N path complexity can be more difficult to maintain over time. This is because the Complexity makes it harder to understand and modify the code. Changes to the program may also require testing of all possible execution paths, which can increase the effort required to maintain the code.
    - Code quality: N path complexity can be used as a measure of code quality. Programs with a low N path complexity are generally considered to be of higher quality because they are simpler and easier to understand and modify.
    - Optimization: N path complexity can be used to identify areas of a program that could benefit from optimization. Programs with a high N path complexity may have redundant code or decision points that could be eliminated or simplified to reduce the number of execution paths.
- **Effect of Refactoring Process:** Unusually high N path complexity in files were another indication for cyclically dependent modularization.
- **Analysis**: The Total Class Fan Out Complexity of the code is `16053`

Task 3a: Design Smells (Refactoring)

The refactored version of the code has been mentioned in Task 2a and also provided in the repo.

Task 3b: Code Metrics (Remeasurement)

These are the code metrics after remeasurement.

Post Refactoring Analysis

1. Boolean Expression Complexity: `48`
2. Class Data Abstraction Coupling: `146`
3. Class Fan Out Complexity: `482`
4. Cyclomatic Complexity: `1013`
5. Java NCSS: `3797`
6. N Path Complexity: `16049`

Observations

- We observe that the Boolean Expression Complexity, Class Data Abstraction Coupling and Class Fan Out Complexity are unchanged and that the other three metrics, i.e., the Cyclomatic Complexity, NCSS and the N Path Complexity, have increased after refactoring.
  Thus, the trend of these metrics being greater than or equal to before refactoring is common to all the metrics.
- This trend was expected as the actual code refactoring was to add the encapsulation that the code lacked. As this was a security design smell, refactoring it was not intended to make the code more readable. Hence, the metrics worsening is justified.

Work Distribution

- Task 1 : Mining
  - Identify classes - Balaji, Dayitva, Aryan
  - UML with OOPs - Rutvij, Samyak, Dayitva
  - Documentation - Balaji, Dayitva
- Task 2 : Analysis
  - Code Smells - Aryan, Samyak, Rutvij
  - Design smells - Dayitva, Samyak, Rutvij
  - Code metrics and documentation - Aryan, Rutvij
- Task 3 : Refactoring
  - Refactor Design smells - Dayitva, Samyak, Aryan
  - Remeasure Code metrics and documentation - Rutvij