Software Engineering Project 1
Team 4

Team Members:

Ainesh Sannidhi 2019101067
Meghna Mishra 2019111030
Tanvi Narsapur 2019111005
Ahana Datta 2019111007
Kajal Sanklecha 2019801006

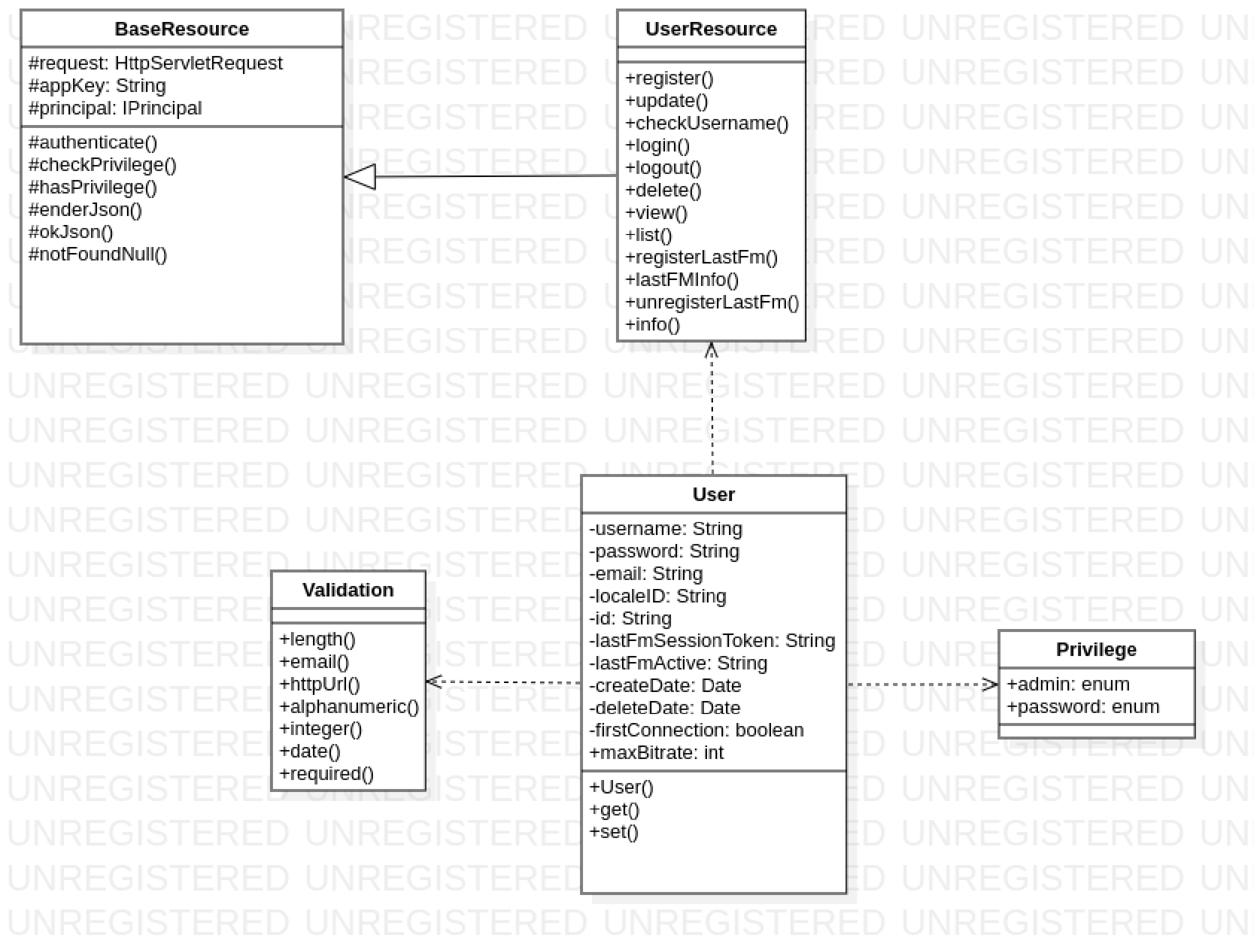
Task 1

Assumption: Since there were a large number of get and set methods for each attribute in each class, we have included one get() function and one set() function to represent all of them abstractly.

User Management

Classes:

- UserResource - Used to create new users and store user data. It also contains basic user functionalities.
- Validation - Used to validate a user's login by verifying login information.
- User - Stores all user-related data. Its operations are used to retrieve all user-specific data.
- Privilege - Allows admins to use their functionalities and allows users to change their passwords.
- BaseResource - Contains basic REST resources and has authentication and admin verification checkers.



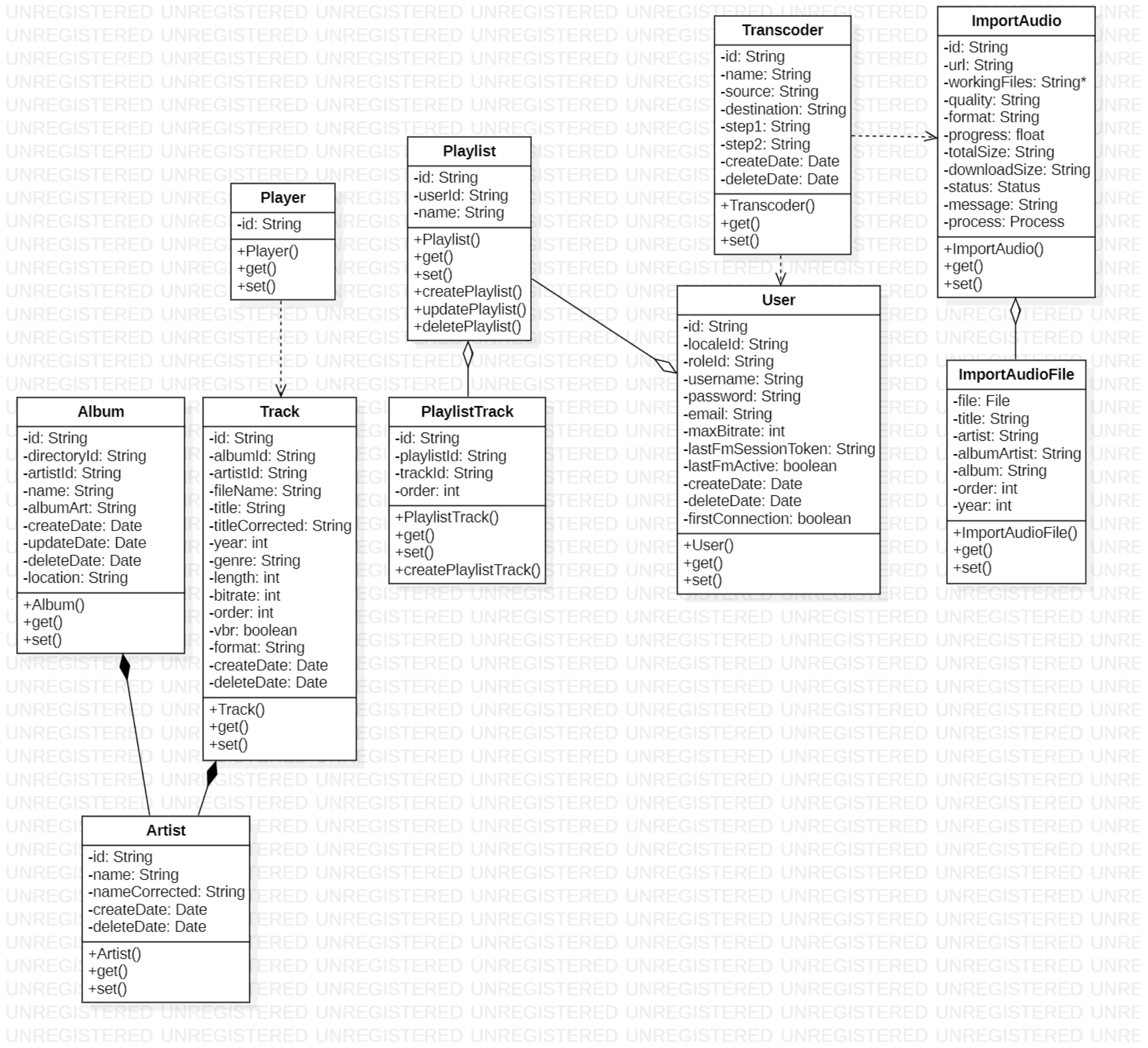
Observations:

1. The code has generally been structured very well, with well-defined purposes for each class and multiple sets of operations which could have been kept in the main user class, however has been separated into several other classes for clarity and to partially deconstruct what would have rather been an extremely large class.
2. Some of the naming conventions could have been improved. For example, UserResource has two update functions which at face value look the exact same, however going into the code there is one difference which is that one of these operations is for when the user hasn’t acknowledged the first connection yet. The same issue exists with delete().
3. Some of the classes are still extremely long with redundant operations for getting information.
4. The privilege class could have instead be kept as an attribute of User as it is just being used as a boolean in the code.

Library Management

Classes:

- Album: Contains details of the album such as directory, name, artist, dates, location, etc. and functionality to fetch and update it.
- Artist: Contains details of the artist such as name, dates of creation and deletion, and functionalities to fetch and update it.
- Player: Contains player ID and functionalities to get and update it.
- Playlist: Contains details of the playlist such as user, name, and functionalities to create, delete or update it.
- PlaylistTrack: Contains details of a track in a playlist such as IDs for itself, its playlist, its position in the playlist, and functionalities to get and update them.
- Track: Contains details of a track, including name, artist, album, etc., and functionalities to get and update them.
- User: Contains all user-related data. Its operations are used to retrieve all user-specific data.
- Transcoder: Contains details about the transcoder entity using which original data is decoded to an intermediate uncompressed format, which is then encoded into the target format, including destination and source details.
- ImportAudio: Contains details of an audio import in progress, such as size, percentage of completion, quality, URL, etc., and functionalities to get and update them.
- ImportAudioFile: Contains details of the audio file being imported, such as album, name, artist, etc. and functionalities to get and update them.



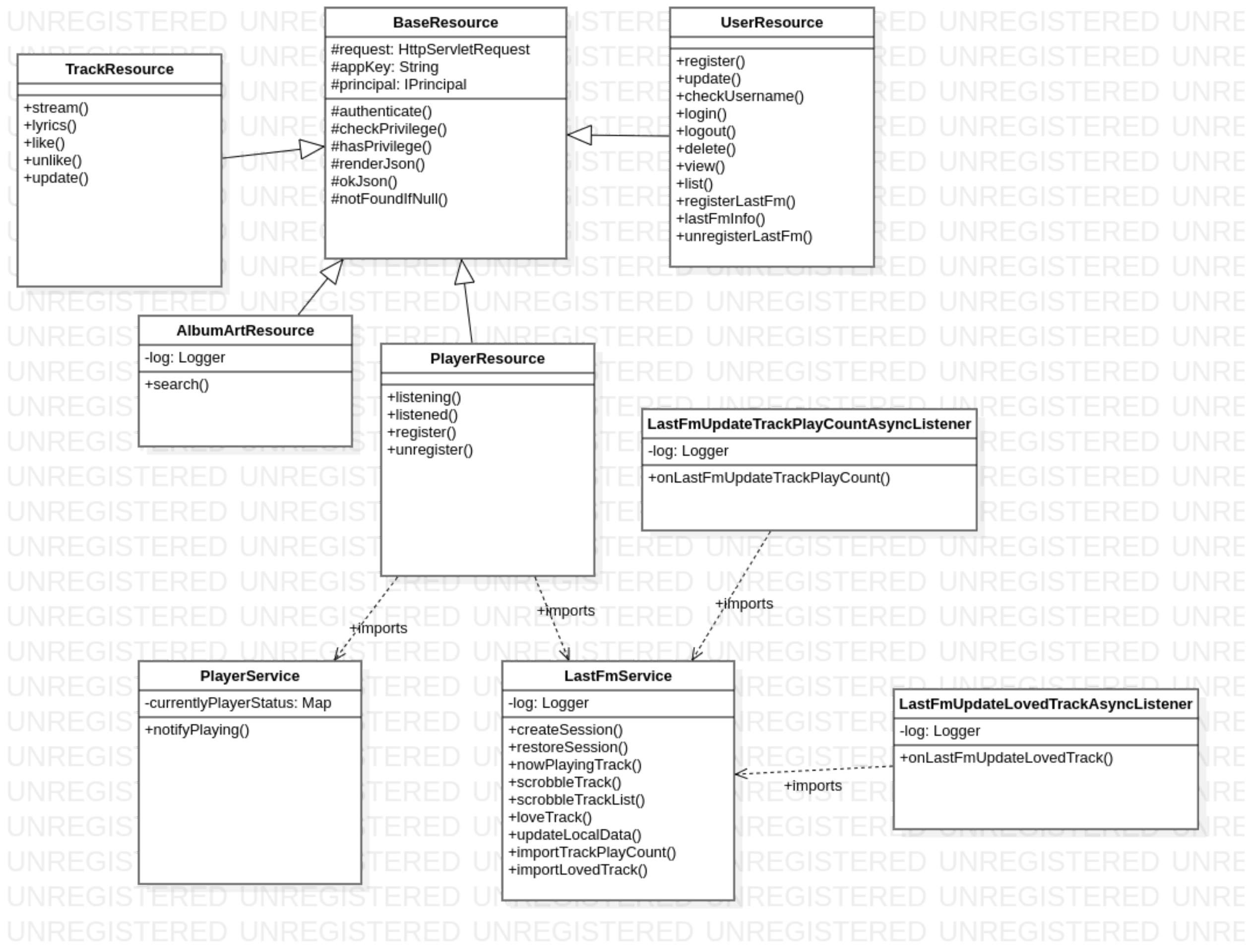
Observations:

1. The code is modular and follows consistent naming conventions, making it easy to extend. The process of importing audio files is also well constructed through the respective classes by storing the details of the files and processes.
2. The classes are long, contain many attributes and double the number of methods, making the code difficult to understand.

Last.fm Integration

Classes:

- AlbumArtResource - Useful for searching for art albums using lastfm API.
- PlayerResource - Provides functionality to access the current playing track, listening history, and registering and unregistering a player.
- TrackResource - Involved in returning a track stream, liking and unliking a track and updating tags on file.
- UserResource - handles user account-related functionalities like registering, deleting, updating user information etc.
- LastFmUpdateLovedTrackAsyncListener - Class allows the user to update the tracks liked by the user
- LastFmUpdateTrackPlayCountAsyncListener - This class updates the track play count.
- LastFmService - Involved in functionalities like creating and restoring user sessions, keeping track of liked and unliked tracks etc. It also tracks listening habits by keeping track of the song tracks, and track lists listened to and uploading them to the Last.fm account.
- PlayerService - Used the update the current playing track.



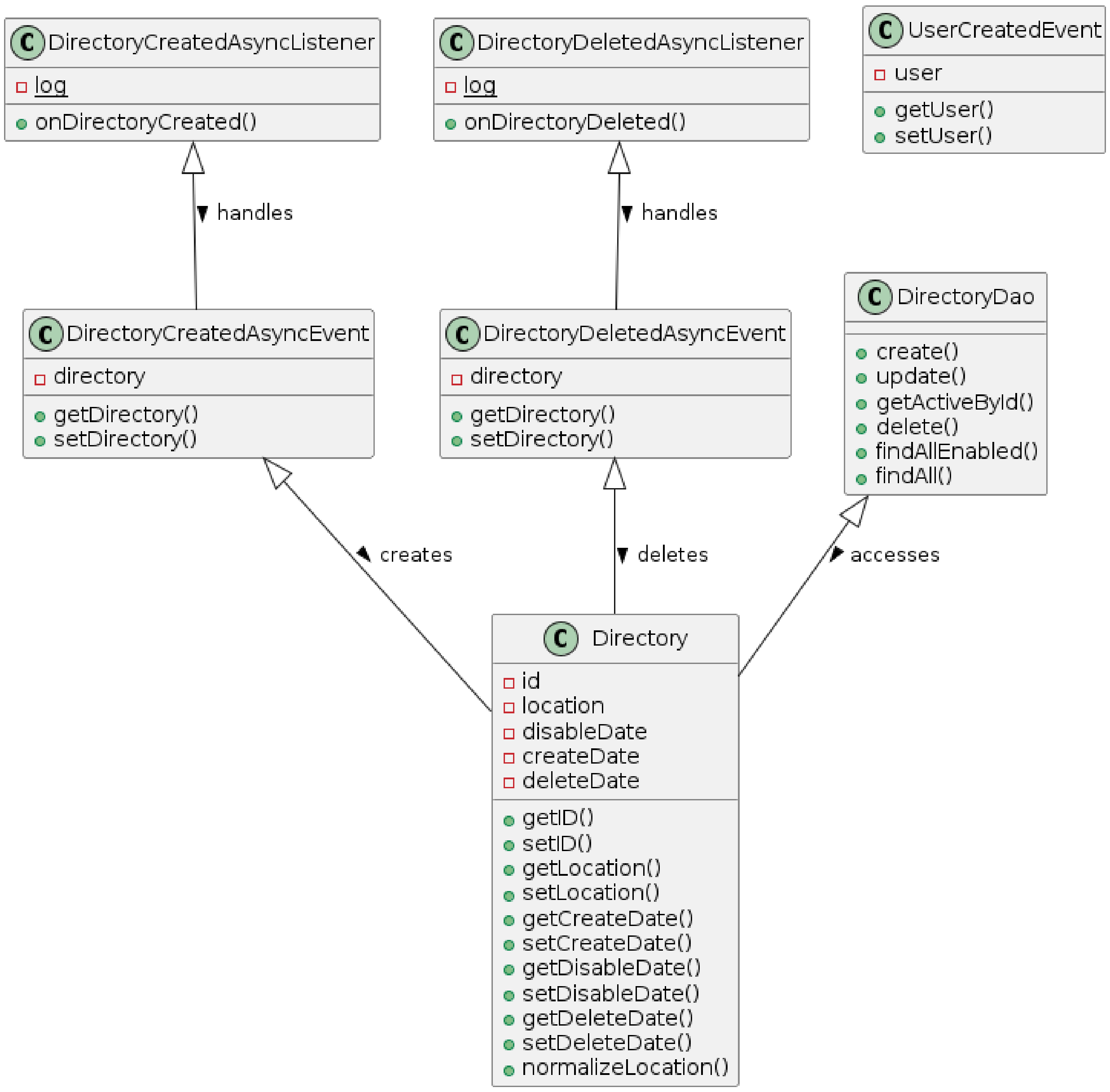
Observations:

1. Provides accurate metadata about soundtracks using Last.fm, including data about artist, album, length, genre, etc for good classification and recommendation of tracks. Code involves methods structured for reusability.
2. Multiple instances of inheritance are seen, where composition should generally be preferred over inheritance.

Administrator Features

Classes:

- DirectoryCreatedAsyncListener : New directory created listener. Used to process the event where a new directory can be made. Here, one can index new directories, watch a new directory and update the scores.
- DirectoryDeletedAsyncListener : Directory deleted listener. Used to get a logger and process the Delete directory event. Used to monitor deleting of directory
- UserCreatedEvent : Event raised after the creation of a user. Used to create and access user accounts
- DirectoryCreatedAsyncEvent : New directory created event. A class object is created when a new directory is created.
- DirectoryDeletedAsyncEvent : Directory deleted event. A class object is created when a new directory is deleted.
- Directory: Directory entity. Stores directory information, getters and setters for all the information.
- DirectoryDao: Creates, updates and deletes new directories. Get active directory, enable directories and all directories.



Observations:

1. Strengths
    a. Handles administrator privileges for giving extra rights for handling directory structures and other rights other than a normal user.
    b. Administrators can create normal users for the application.
2. Weaknesses
    a. DAO classes are many, so DAO should be an abstract class or interface so that there can be a consistent structure for all of them.
    b. Async Listener can be a single class with instances in Create and Delete of Directories.

Task 2a -

Identifying design smells.

1. Long method: Some classes contain the code smells like - ‘Refactor this method to reduce its Cognitive Complexity from 19 to the 15 allowed’. This makes the methodology difficult to understand and maintain. It could be simplified to enhance understandability.
2. Redundancy: Many code files contain the code smell - ‘Define a constant instead of duplicating this literal “---” n times’. This makes it difficult to update the code, and care must be taken to ensure that every occurrence of the literal is modified. Instead of this, we can define a single constant which reduces the redundancy as well.
3. Broken modularisation: For the criteria classes - AlbumCriteria, ArtistCriteria, PlaylistCriteria, TrackCriteria, and UserCriteria, an abstract class can be created which includes the common functionalities instead of distinctly mentioning the methods in each class. This would also be beneficial to add some common functionality to all the criteria classes.
4. Imperative abstraction: Some classes have been constructed by converting an operation into a class. UserAlbumDao, UserUtil, AlbumArtFilenameFilter and AlbumArtImporter are some examples. This can be avoided by directly including individual methods at required locations.
5. Tight coupling: Inheritance gives rise to tight coupling (interdependence of classes). In the given codebase, there is a series of inheritance observed in the following classes (each class inherits from the class on its left) -
   JerseyTest, BaseJerseyTest, BaseMusicTest, TestAlbumArtResource.
6. Broken Modularisation: The dto classes - AlbumDto, ArtistDto, PlaylistDto, TrackDto, and UserDto contain some functionalities in common. So instead of including these common methods in all the classes individually, we can create an abstract class from which all the classes can inherit common functionalities.

Task 2b -

Using CodeMR, we have calculated many different metrics and generated multiple graphs analysing the different classes and their respective metric values. Some of the code metrics reported are:

- Total lines of code - 6947
- Complexity: 89.2% with low complexity, 10.8% with low-medium complexity
- Size: 34.9% with low size, 60.6% with low-medium size, 4.5% with the medium-high size
- Class lines of code: 34.9% with low CLOC, 64.6% with low-medium CLOC, 4.5% with medium-high CLOC
- Access to foreign data: 86.1% with low ATFD, 13.9% with low medium ATFD

The implications of these code metrics include indicating the amount and percentage of problematic classes in the overall model and revealing the departments in which the code structure has room for improvement. We get to know which classes are problematic because of reasons such as high complexity or high coupling. Further, metrics such as lines of code, number of methods and parameters, etc., give insight into the overall size and intricacies of the code.

They guide our process for refactoring in such a way that we aim to reduce the problematic code metric values and devise an approach to refactoring that betters the overall metrics of the model.

Task 3b -

Using CodeMR, we have calculated the code metrics and compared them with the original codebase.cs after refactoring the code.

- Total lines of code - 6901
- Complexity: 93.5% with low complexity, 6.5% with low-medium complexity
- Size: 35.2% with low size, 60.3% with low-medium size, 4.5% with the medium-high size
- Class lines of code (CLOC): 35.2% with low CLOC, 60.3% with low-medium CLOC, 4.5% with medium-high CLOC
- Access to foreign data: 86% with low ATFD, 14% with low medium ATFD

The above metrics improved after refactoring the codebase based on the design smells mentioned in Task 2a. However, a few other metrics not mentioned above marginally worsened. This could be because of the addition of classes and methodologies as a part of refactoring. The results we obtained matched our expectations because they probably happened due to the introduction of new classes and methods.