CSC207 Design Document Phase 1

Table of Contents

Updated Specification

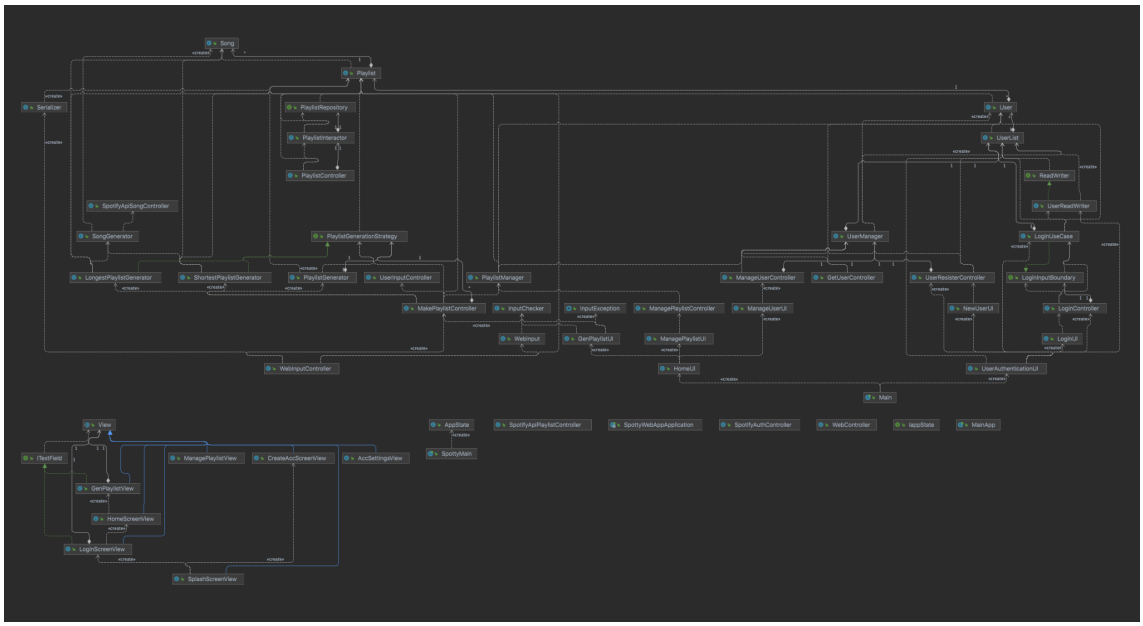
Spotify allows the functionality to create ordered playlists to sort and save songs which are then shareable with other users. This then led to people creating playlists where the title of each song creates a sentence.

Our program will generate a playlist (group of songs) based on a string input by the user. These then get assigned to the user's account and the user can also edit the contents of the playlists.

- Running the program starts a virtual server, where the user can access the webpage for the web app. The webpage has a title screen and a menu interface with 3 options. First, the user is prompted to log in to their spotify account by a button
- Login: If the user chooses to login, they are redirected through the spotify api to the spotify OAuth page where they can accept or decline to grant the application access to user information from the requested scopes. If they accept, they are rerouted back to the web app webpage.
- After logging in successfully the program prints a menu where users can choose from:
    - Generate new playlist from string
    - Manage saved playlists [works for text-based interface but not on web]
- Generate new playlist from string:
    - If the user wants to create a playlist, they are prompted to enter a string of words which the program then splices in multiple different combinations to be able to find song titles for each word or group of words. Once song titles are found for each song a playlist is created and the user can choose to add it to their spotify account or not.
- Manage saved playlists
    - Once the user has a number of saved playlist they can then edit and manage their playlists, deleting playlists they no longer want and changing the playlist name [works for text-based interface but not on web]

Class Diagram

- UI classes for the api
- Controller classes
  - On the left you can see controller classes relating to playlist generation. These include all the classes responsible for handing the user inputThey are interdependent with each other and mostly with the Playlist Generator Use case, which connects them to the Playlist entity.
  - There is a composition relationship (dependency injection) between the MakePlaylist controller and the PlaylistGenerator.
  - There are some examples of violations of clear architecture in the authentication as the login use cases have a direct association/dependency on the UserList entity. We plan to introduce interfaces to target some of these issues. In the case of UserList, our implementation of the database should rid these dependencies altogether.
- Use Case classes
  - The <<create>> dependency implies that the client class creates an instance of the supplier class. This makes sense as our use cases are responsible for instantiating our entities, however this also implies a strong coupling, which we want to avoid. We plan to refactor this in Phase 2, as our implementation of the integrated database will allow us to instantiate our objects through a creational design pattern interface (likely abstract factory), which will mean our entities and use cases will have less coupling.
- Entity classes
  - None of the entity classes are dependent on other classes, showing out dependencies point inward.
  - View classes are part of our Phase 0 UI, and have changed because of our implementation of spring, and may become obsolete in phase 2



Major design decisions

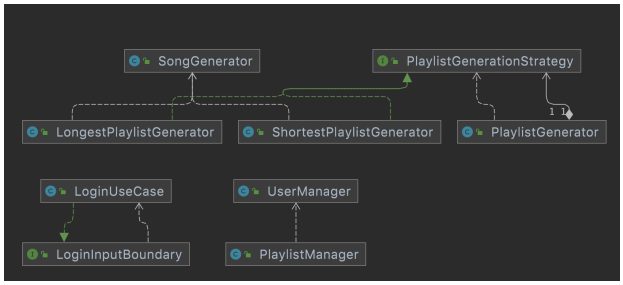
1. **Implement a Web App:** From phase 0 of the project, the principal change was refactoring and packaging our code as a web app. We altered the program and gradle to be compatible with Spring Boot, allowing us to make a spring based web app implemented in Java. This allows for an integrated database and seamless connection to the API
2. **Replace the decision tree to index shifting in an arraylist:.** In Phase 0 we intended to use a decision tree to be able to split the string into different combinations and search spotify (or our testing csv file) for each resulting combination. This way the longest possible string of words will be created as a Song object from the beginning of the list and then the next longest string will be selected. This version is simpler as we try to find the longest string that correlates to a song name from the beginning of the sentence and then move to the next words in the string. This means we are less likely to find songs for all of the words, however due to spotify's immense song variety this shouldn't adversely affect the scope of strings the program can take. Additionally although we want to create the longest song titles possible we also want to be able to keep the playlist in the correct order and the decision tree makes this complex. Once we search through depth first according to the length of each string combination we will end up at a different layer of the tree for each spliced string, and we can't search through breadth first as we would have to rule out all of the branches that group words together that are not consecutive within the original string. This alternative method is significantly simpler and shouldn't affect the functionality.
3. **Strategy Implementation for playlist generator:** The user will now get an option to create the longest or shortest playlist (according to song name) using a strategy design pattern to separate both algorithms. This was an appropriate time for the strategy design pattern since we wanted to implement two versions of a searching algorithm that would in the end build playlist objects that would be essentially interchangeable.
4. **Graphical UI Controllers:** As we decided to move to a web based interface, we naturally had to adjust some of our controllers. For instance, we removed the username parameter for the controller responsible for handling user input to create a playlist.
5. **Spotify controllers for API authentication:** We originally intended to store user objects however once we considered the functionality of the spotify API, and that we would need to authenticate through OAuth, we could request the user to allow the app access to User Information. We no longer need to implement an authentication system directly within our code, although we still intend to store the user ID through spotify purely so we have the ability to implement some of our planned extensions

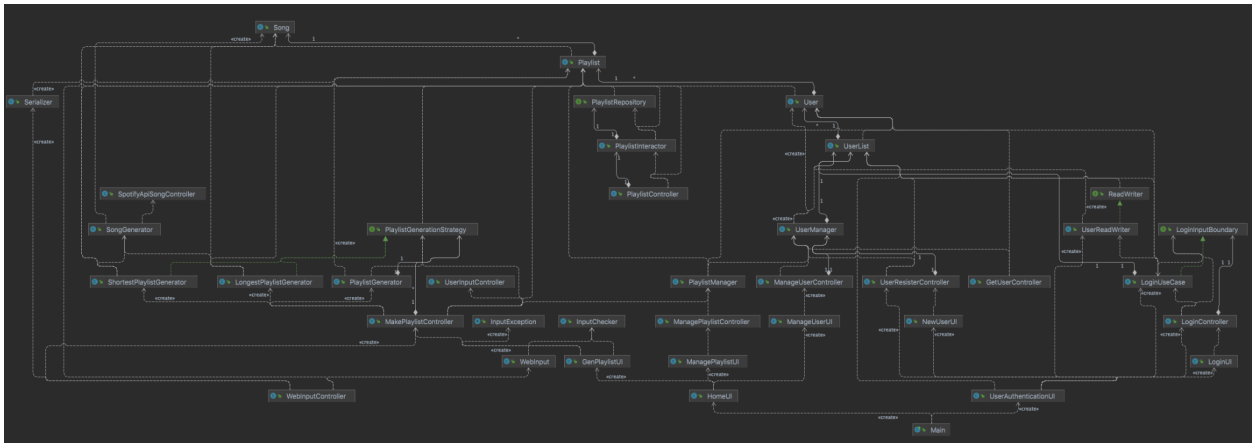
Clean Architecture

Firstly, we organized our code in packages that represent the layers of the Clean Architecture. This means that we are always (or mostly) adhering to the principles, as we will be working in packages that literally represent the layers. The notable packages are Entities, UseCases, and Controllers. Our cli package acts as the “views” layer in our text-based interface version, and our “resources” directory is where we have the GUI “views” layer. This means we have placed abstractions of the objects in the inner layers and implementation of our interfaces in the outer layers, and that our dependencies point inwards. Some of our layers currently have some unexpected dependencies and this is probably due to us missing interface layers in between. For example, we intend to implement a database during phase 1 but thus far are still using data serialisation and there is a dependency in our serialisation class.

As we are implementing the cli alongside the api (as a backup) we have shown the program is independent of UI and We planned to integrate a database on Spring Boot for this phase however so far we are still using data serialisation, which shows our program is independent of a database.

Thus far in our UML diagram majority of the dependencies do point inward through the layers, although due to some classes that are a work in progress (as part of the database), there are some dependencies and aggregations that point outward. This will likely be fixed once the database is established, as some of the classes such as UserList were used for authentication purposes before authentication was handled by spotify, however are still being partially used for serialisation. This will be solved within phase 2 by introducing more interfaces between the layers to ensure that there are less dependencies





SOLID Design Principles

Single Responsibility Principle: "Classes should have a single reason to change".

- PlaylistGenerator has the single responsibility to generate playlists, not depending on what type of song objects (web based or UI based) are being used to generate playlists.
- Classes were created in such ways to have a "single responsibility", so that they are not doing more than they need to be.
- SpotifyApiController being split into two smaller classes more specific to their responsibilities as controllers for playlists and songs. To implement in phase 2: Make a spotifyAPI facade.

Open/Closed Principle: "High level modules shouldn't depend on low-level modules, but rather depend on abstractions" and "Abstractions shouldn't depend on details, details should depend on abstractions"

- Even while integrating the database functionality, we simply added new methods to our existing entity and use case classes and did not require any changes within our classes.

Liskov Substitution Principle: "If S is a subtype of T, then objects of Type S may be substituted for objects of type T, without altering any of the desired properties of the program"

- PlaylistGenerationStrategy interface implemented by shortestPlaylistGenerator and longestPlaylistGenerator. The playlistGenerationStrategy class specifies two methods, generatePlaylist and generatePlaylistWeb, which are both implemented in ways specific to generation of shorter playlists, and longer playlists. These classes could hypothetically replace generatePlaylist with no change in functionality.

Interface Segregation Principle: "Keep interfaces small so that users don't end up depending on things they don't need"

- We made sure that the classes/interfaces we've created contain methods that are grouped together by relevance, and as a result, the user is not forced to implement any irrelevant methods. (i.e. SpotifyApiController was split into SpotifyPlaylistController and SpotifySongController to group methods together and separate them in such a way that makes more sense).

Dependency Inversion Principle:

- There are several places where Dependency Inversion Principle is used:

    - Playlists are generated in the Playlistgenerator class, which uses the interface PlaylistGenerationStrategy when generating playlists, which creates that abstraction layer between the classes.
    - The controller class, UserReadWriter, implements the ReadWriter interface.

- In our code, most of the high level classes are dependent on the low level classes. However, we didn't introduce many abstraction layers (i.e. interfaces) to separate the low and high level classes. This is one of the design issues that we want to solve in Phase 2.

Refactoring

Throughout phase 1, there were several refactorings done to improve the code's readability.

1. Problem: There were many print statements in the text-based command line interface.
   Solution: Private helper methods were created which contain only the print statements. These helper methods were called instead in the main method. Moreover, these helper methods could be changed independently without affecting other methods.
2. Problem: There were many methods in the use case classes (especially the UserManager class) which had similar names and functions.
   Solution: First, we renamed the names of the methods to specify what those methods were doing. We also created a few private helper classes when some methods share the same chunk of code. Moreover, some methods are overloaded, for example, in the PlaylistManager class, the removePlaylist method is overloaded, which can either take in a Playlist object and a String as input, or an integer and a String as input. The functionality is the same, however.
3. Problem: There were many methods that were missing javadoc or had outdated javadoc.
   Solution: Updated these javadoc to represent the functionalities of those methods.
4. Problem: The SpotifyApiController has low cohesion: it contains methods which deal with songs and playlists.
   Solution: The SpotifyApiController is splitted into two new controllers: SpotifySongController and SpotifyPlaylistController. These controllers have higher cohesion because they manage songs and playlists only respectively.
5. Problem: There were unused methods and classes.
   Solution: Deleted unused methods and classes.
6. Problem: The text-based command line uses UserManager and PlaylistManager to retrieve data of the users and playlists, but calling methods in these manager classes only makes changes to that instance of the manager.
   Solution: For the UserManager and PlaylistManager use cases, we changed these classes into singleton classes, which means that there is one and only one instance of these managers. This means that these managers each have one "copy" of their attributes, so we cannot create different manager classes which may have different attributes and not sync up with one another.

Packaging Strategies and Code Organization

Our packaging strategy is grouping classes in the same layer of Clean Architecture into one package.

The following are the packages in this project:

- api
    - This package is responsible for using the Spotify Web API.
    - The classes in the api package use classes in cli, Controllers and Entities package. (This might be a violation of Clean Architecture. See Existing Design Issues section for more details.)
- cli
    - This package includes all the UI classes responsible for the text-based command line interface. Notable classes include the HomeUI (the main menu of the command line interface), GenPlaylistUI, ManagePlaylistUI, and ManageUserUI. The last three UIs are different option menus of the command line interface, which the user can access from the main menu.
    - The classes in the cli package use classes in the Controllers package.
- Controllers
    - This package includes all the controllers. Notable ones include ManagePlaylistController, ManageUserController and MakePlaylistController, which are used by the UI (class in the cli and api) to access playlist and user entities, and to make a playlist.
    - These controllers use classes in the UseCases package.
- UseCases
    - This package includes all the use cases. Notable ones include ShortestPlaylistGenerator and LongestPlaylistGenerator, which are responsible for generating a short or long playlist; Song generator is used by the two playlist generators mentioned above, to find a song from an existing collection of songs; UserManager and PlaylistManager are used to manage users and playlists.
    - These controllers use classes in the Entities package.
- Entities
    - There are four entity classes in this package, namely Song, Playlist, User and UserList.
    - These classes are for data storage and contain basic methods for manipulation (e.g. changing attribute values).

Design Patterns

- Dependency Injection
    - To Prevent hard-coding the Song type of assigned values within an instance of the Playlist class we have used the SongGenerator class to add created Song objects into the Playlist class. this allows for us to subclass song type and potentially extend functionality to include podcasts from Spotify, and additionally this allows for dependency inversion
- Strategy - Longest vs shortest playlist generator
    - We implemented two different methods of playlist generation as options to the user. For this we implemented a strategy design pattern interface to create two algorithm choices to decide if the want a longer or shorter playlist (by number of songs)
- Singleton
    - The UserManager and PlaylistManager classes in the UseCases package are both singleton classes.
    - Singleton design pattern is used here because we want one and only one manager for users and playlists each. Moreover, this design provides global access to these managers, which means that other use cases to call methods of these classes. In addition, it allows lazy initialization, which means we call, and initialize, these manager classes only when we need them.
    - One possible downside is that since singleton classes can be accessed globally, we must pay extra attention as to not call this use case class in higher level classes.
- We intend to implement an extension of the Abstract Factory method as well, as part of the integrated database functionality within Spring Boot

Existing Design Issues (Code Smells)

- Unused imports and imports that violate clean architecture
    - Some high-level classes import low-level classes which possibly violates the principles of clean architecture.
    - For example, the WebInputController class in the api package imports Playlist from the Entities package. However, WebInputController doesn't change any attributes of any entities; it just displays information.
    - We plan to refactor our code and look through all the import statements to minimize coupling between classes.

- Long methods
    - For example, in SongGenerator, ShortestPlaylistGenerator and LongestPlaylistGenerator, the methods for generating songs and playlists are quite long, since they involve a complex algorithm to generate our wanted songs and playlists.
    - We plan to reduce the number of long methods by making more private helper methods, whenever possible.

- Divergent Change
    - Since we didn't use many interfaces to separate the different layers defined by Clean Architecture, high-level classes depend directly on low-level classes instead of an interface. This means that a single change in low-level classes might require more changes in several high-level classes.
    - We plan to solve this issue by introducing more interfaces in between clean architecture layers.

Progress Report

Open Questions

How do you work SpringBoot?

- Some features work, some that would make life a lot easier just do not work. Workarounds for the spring boot features like Model and View class and HTML references that should work have been found.
- There is also limited clear documentation of Spring Boot, using Java and gradle.

PostgreSQL: postgresql servers do not set up correctly with spring, although we have set up our dependencies clearly, and our application properties file to set up the data sources through spring is correctly configured.

What has worked well

Our backend code is pretty SOLID, except for the database and using the SpringBoot framework.

Outstanding issues

- Our song search and playlist generation algorithms are working as they should be, however, because of how spotify's search algorithm works (top search results are based off of the user's listening habits, sometimes the top results may not even contain the word we are searching or contain it in a way like: attempt to search for a song just called "hi", instead spotify returns a song called "high hopes"), sometimes the returned songs are not exactly matching the word we are asking to search, or even not returning anything because a song simply named, "the", does not exist in the top 50 recommended songs for the currently logged in user. In phase 2, we hope to manipulate the search algorithm better in order for the returned playlist to be as close to (hopefully identical to) the inputted string.

E.g. output for "Baby welcome to my house" using the option to generate a long playlist:

0) Baby, Baby, by: Madison Beer. 1) Welcome, Welcome, by: Hey Rosetta!. 2) 3) 4) House, House, by: Far Caspian.

Spotify was unable to provide us with songs for “to” and “my”, while debugging it was evident that Spotify just didn’t think that the currently logged in user would be interested in the songs “to” and “my”.

Table of work done by group members

Member	25th Oct to 1st Nov	1st Nov to 8th Nov	8th Nov to 15th Nov
Max	HomeUI, playlistGeneratorUI, refactored phase 0 UI.	WebControllers implemented to handle form submission via POST requests, front end to back end data processing, Spring boot configuration.	Serializer class implemented to ensure data persistence, display information from server side to client via html get requests, Spring boot config.
June	Generate playlist algo	Generate playlist strategy	Reading week, so I was away but helped a little with the spotify api algo
Tyler	ManagePlaylistUI and related controllers	ManageUserUI and related controllers	SpotifyAPI Further Testing (esp Use Cases) Design Document
Minori	UserAuthentication UI, Login UI, NewUser UI and related controllers and use cases	Continued working on UserAuthentication UI, Login UI, NewUser UI and related controllers and use cases	Worked on html & css,
Sara	Worked on setting up the external database	Set up the Web app, there was no longer a need for the old database code as we decided to have it integrated within spring, which completely changed the platform we set up the database on, and how it was set up	Worked on resolving issues with configuration of the database, design document

Danika	Created authorization code and authorizationcodeuri classes in order to be able to connect to the spotify api.	still working on authorization to be able to access the spotify API, kind of struggling implementing for java and springboot.	Figured out authorization, user can authorize access to their spotify data. Implemented new methods within songGenerator and playlistGenerator to generate entities based on results from the spotify database. Works with springboot.

What each member plans on working on in phase 2:

Member	Plans
Max	- Tidy up the GUI (HTML, CSS, Spring boot web controllers) - Extend GUI features to process user instructions for managing playlists and saving those changes server side. - Add tests for the web controllers.
June	Add a bunch of test cases. Fix the spotify api issue.
Tyler	- Possibly extending the feature so that the text-based command line UI can be used directly to get data from Spotify
Minori	- Add testing - Help out with the GUI - Add interfaces - Create a csv based class that will store users and their associated serialized playlists in the .csv file. - Would allow for easier management of specific user playlists.

	- Contingency in case the database is not implemented.
Sara	- Finish implementing the database, - implement extensions that use the database, such as displaying the last 10 created playlists on the homepage, integrate profile pictures and playlist pictures to make the GUI more appealing - Add testing - Refactor our files, javadocs and dependencies and add README
Danika	- Work on implementing the feature to add the generated playlists to the user's spotify account so that they can actually listen to them. - Implement extensions to how the playlists are generated, base them off of song features available in the spotify database, i.e. popularity or whether a song is explicitly or not. - Figure out how to better manipulate the spotify search algorithm.