Design document 1: UML Diagram

Due to the size and complexity of our program, instead of a UML diagram, we've decided to give a verbal description of how data flows through our program.

User input

Any time a user interacts with the program, their input is taken and passed to the CommandController which will call the executeCommand method for the appropriate command based on the user's input.

Songs

The SongManager class looks for songs in a file and stores them. There's then a class called Program which stores an instance of SongManager among other things, which can return its SongManager whenever songs need to be accessed.

Playlists

The playlist and its subclasses are the container of music that the music player accesses. When program requests any song, the songs will be delivered as a permanent playlist form(i.e. Albums, favourites, user created playlists) or a temporary package just for this delivery(i.e. CurrentPlaying, program created custom Playlist). The permanent playlist are stored and managed by the PlaylistManager.

Design document 2: Design Patterns

- The Command design pattern was used to handle all commands in the program (e.g. there are commands for logging in and out, commands for playing and pausing music, etc…). The classes involved in the implementation of this design pattern are Command, CommandController, and all of the classes that inherit from the abstract Command class.
- The Singleton design pattern was used to handle persistence for playlists, to ensure that there could only be a single point of access to the persistence service. The class that implements this pattern is the PersistenceService class.
- The Adapter design pattern was used to modify and adapt the MP3 playing class that we decided to use in our project to add functionality that was necessary for our project, but was absent from the original MP3 playing class. The class that implements this pattern is the MusicPlayer class.
- The Dependency Injection pattern was used in Program, the parameters and variables that are relating songs and playlists are all sent to corresponding managers. This way we are noft hard-coding our high level code. This way we can let multiple people work in parallel and prevent someone unintentionally breaking other's code. The class that implements this pattern is Program.

Design document 3: Design Decisions

As for design improvements, we created an interface for UI displaying classes to implement and moved the previous UI-related methods to a new set of classes which all implement the aforementioned interface. This was done to make our code adhere to the Dependency-Inversion principle.
As for new features, we began working on a button-based graphical user interface as opposed to a text-based interface, and we also implemented persistence for playlists so that playlists could be saved even after the program terminates.
As for the access and storage of entities, we designed Song entity to access the song file outside the program while having basic information easily accessible by the program. The SongManager is tasked to store songs and manage them.The playlist is a uniform format that the program would accept when music is requested. Some playlist types are designed to be permanent, once they are created, they are meant to stay.(i.e. Albums, favourites, user created playlist). Some playlist types are meant to fulfil temporary functions and are created by the program to fulfil storage tasks for now(i.e. CurrentPlayling, History and program created playlist). PlaylistManager is tasked to store all the permanent playlist and answer calls from the program regarding playlist operations. By defining two fundamental entities that are very important program functions, we allow us to have the flexibility to add more functionality without breaking the code.

Design document 4: Universal Design

Principle 1: Equitable Use
Right now, our program depends on the user's ability to see the user interface and interact with it using a mouse and keyboard. This means that users who have vision impairments or who have disabilities preventing them from using a mouse and keyboard would have a significantly harder time using our program.
Principle 2: Flexibility in Use
We have a text-based UI and a graphical button-based UI implemented, which gives the user a choice between two different ways to interact with the program. The loginGUI also provides users the choice to view or hide their passwords, considering that users might forget what they had typed in. After the users login successfully, they will be directed to the Main page. The MainGUI provides the users to play and pause the music, as well as mutating the order of the currently playing playlist (shuffle and repeat).
Principle 3: Simple and Intuitive Use
We could improve on this by giving more prompts based on when something is done in the program. For example, prompting users that music is currently playing / paused,
Principle 4: Perceptible Information
Similar to the first principle, the program is much more difficult to use for users who have visual impairments so the addition of features to help accommodate such users would help improve our program's coverage of this principle.
Principle 5: Tolerance for Error

In the case that a command was incorrectly typed (for the text-based UI) the program simply tells the user that there was an issue and the user can freely try again.

Principle 6: Low Physical Effort

Might not be applicable for us. The physical effort required is no more than that required to use a computer.

Principle 7: Size and Space for Approach and Use

Again, might not be applicable for us. Physical things like this depend more on the peripherals that the end user is using to interact with the program.