Project

by

Jason McCauley and Aidan Nestor

Stevens.edu

December 20, 2024

Project

Jason McCauley and Aidan Nestor
Stevens.edu

Table 1: Document Update History

Date	**Updates**
11/07/2024	JM: • Under Chapter 1, wrote developer biography paragraph • Under Chapter 2, wrote a couple paragraphs detailing the project and the problem it is intending to solve • Under Chapter 3, wrote a couple paragraphs explaining what the bot does and why it is a viable solution to the problem • Under Chapter 4, wrote two use cases that describe the main functionality of the Spotify Clean Playlist Bot. Created a table of requirements with priorities mapped to each Use Case. Created a table of Use Cases mapped to the requirements. • Under Section 5.1, created a sequence diagram of the Spotify Clean Playlist Bot in action. • Under Section 5.2, created a five-frame storyboard to demonstrate the primary task that a user performs with the bot.
11/08/2024	AN: • Under Chapter 6 Section 6.1, created the component diagram for the bot. Also added a paragraph explaining the interactions between the components. • Under Chapter 6 Section 6.2, created both activity diagrams, one for each use case.

Table 1: Document Update History

Date	Updates
12/06/2024	JM: • Updated the introduction chapter to discuss my responsibilities for the project, as well as the effort I believe is being put forth by each team member. • Created a chapter for the Agile process, which includes a section of notes from our first two meetings with Dr. Muresan. Also includes a section on issue tracking, with a screenshot of our current Kanban Board. • Created a chapter for the process reflection, which includes a section for the status of complete tasks, where we discuss which tasks on the Kanban board have been completed, and how Aidan and I have been able to work together effectively throughout the project development. Also includes a section for the status of incomplete tasks, which details why those tasks have not been completed yet. • Updated dsnManual.tex to properly render the Bibliography page. Added a citation to a website we referenced when creating our UML diagrams.
12/07/2024	AN: • Under Chapter 6, updated Activity Diagrams for Use Cases 1 and 2. Fixed errors in the flow of the diagrams. • Wrote an introduction about myself in the introduction section. • Created a List of Tables page to keep track of and reference all of the tables throughout the notebook. • Created a List of Figures page to keep track of and reference all of the figures throughout the notebook.
12/19/2024	JM: • Under Chapter 5, updated the sequence diagram to correspond with the class diagram after Dr. Muresan's feedback. Added the class diagram with code snippets, and explained how those code snippets aligned with the UML. Also added a subsection to explain the design pattern we implemented and how the code corresponds. Also added the object diagram that corresponds with the class diagram. • Under Chapter 6, added the package diagram and the composite structure diagram. • Under the Agile Process Chapter, updated the meeting notes to include the discussion between Aidan and I after Dr. Muresan provided feedback on our presentation. • Under Chapter 8, updated discussion of incomplete and complete tasks to correspond with the Kanban Board from the day of the final report submission.

Table 1: Document Update History

Date	**Updates**
12/19/2024	AN: • Under Chapter 6, created Communication Diagram, Interaction Overview Diagram, and Profile Diagram. • Under the Agile Process Chapter, added more screenshots of the Kanban board for issue tracking.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

– Jason McCauley and Aidan Nestor

My name is Jason McCauley. I am a 4/4 Software Engineering student at Stevens Institute of Technology. While I do not have any industry experience yet, I did take part in research this past summer. Working under Dr. Jacob Gissinger in the Chemical Engineer Department, I preprocessed a large dataset which encompassed the interaction between two polystyrene molecules. Utilizing relevant features, such as the position and velocity vectors of the molecules, I developed a robust machine learning model to predict whether or not a reaction would occur between them, achieving 94% accuracy. For this project, I am responsible for managing the Kanban board, whether that be creating tasks, assigning them between Aidan and me, or designating story points. Additionally, I am expected to work on the class diagram, use case diagrams, sequence diagram, state machine diagram, composite structure diagram, package diagram, and object diagram. Both Aidan and I will create the needed components of this project notebook, such as a list of figures, list of tables, a bibliography, a chapter to reflect on the status of complete and incomplete tasks, and a chapter to discuss the Agile process we adhered to. Truthfully, I would say that the amount of effort put forth by each of us has been an equal 50% up to this point, and I expect that to continue for the remainder of this project.

My name is Aidan Nestor. I am a 3/4 Software Engineering student at Stevens Institute of Technology. I don't yet have any industry experience however I have an Associates Degree in Software Engineering. For this project, I have mainly been responsible for the setup of certain parts of the Overleaf document including the list of tables, list of figures, etc. I am also responsible for certain diagrams including profile, communication, interaction, timing, deployment, component, and activity diagrams. I will assist Jason in creating the necessary components that Jason has mentioned above. I would agree that the effort put in has been an equal 50% up to this point, and I also expect for this to continue.

Chapter 2

Problem Statement

– Jason McCauley and Aidan Nestor

Music streaming has become deeply integrated into our daily lives, with people using their personal playlists across various social and professional settings. However, a significant challenge arises when listeners need to adapt their existing playlists for environments where explicit content is inappropriate or unwelcome, such as professional workplaces, family gatherings, school events, or public venues. This is a pressing problem because manually creating clean versions of playlists is an incredibly time-consuming and tedious process – users must individually search for each song's clean version, verify that it corresponds with the original track, and rebuild their playlist from scratch. For playlists containing hundreds of songs, this process can take several hours of focused effort.

The issue is particularly problematic because the consequences of not having readily available clean playlists extend beyond mere inconvenience. Employees may feel unable to play their personal playlists in their workspace due to the risk of explicit lyrics, teachers might be hesitant to play musical content in educational settings, and event organizers face the constant worry of inappropriate lyrics surfacing at family-friendly gatherings. Additionally, the manual conversion process is prone to human error, potentially leading to embarrassing situations where explicit lyrics accidentally slip through, damaging professional relationships or creating uncomfortable social situations. Typically, this forces many users to either completely avoid playing their carefully curated playlists in public settings or spend excessive time double-checking each song before playing it.

Chapter 3

Project Description

– Jason McCauley and Aidan Nestor

The Spotify Clean Playlist Bot is an automated solution that streamlines the process of creating clean versions of explicit playlists while maintaining the original listening experience. The bot leverages Spotify's Web API to analyze the existing playlist of a user, intelligently identify clean alternatives for explicit tracks, and automatically generate a new playlist with appropriate versions of each song. This solution directly addresses the core problem by reducing a process that typically takes hours into one that requires just a few clicks, while ensuring completeness and accuracy in the song-matching process.

The bot operates as an event-response system, focusing on completeness and accuracy in playlist conversion. When a user submits a playlist for conversion, the bot performs several operations: it authenticates with Spotify's API, analyzes the original playlist's tracks, searches for clean versions of each explicit track using a matching algorithm that incorporates factors like the song title, artist, and duration to ensure accurate matches, and finally creates a new playlist with the clean versions. The modular, object-oriented design allows for thorough error handling and quality assurance, such as marking songs where clean versions cannot be found and providing users with clear feedback about the conversion process. This automated approach not only saves users significant time but also reduces the likelihood of errors that often occur during manual playlist conversion, making it a practical and reliable solution for individuals and organizations that need clean music playlists for various settings.

Tagline: "Clean your playlist with just one click!"

Chapter 4

Use Cases

– Jason McCauley and Aidan Nestor

Use Case 1: Generate Clean Playlist

1. Preconditions
    - User is authenticated with Spotify
    - User has access to source playlist
2. Main Flow
    - User provides playlist ID or URL [S1]
    - Bot analyzes playlist content [S2]
    - Bot searches for clean versions of explicit tracks [S3]
    - Bot creates new playlist with clean tracks [S4]
3. Subflows (for bot)
    - User inputs playlist identifier [S1]
    - Bot retrieves playlist data and track information [S2]
    - Bot identifies explicit tracks and searches for clean alternatives [S3]
    - Bot creates new playlist with clean versions [S4]
4. Alternative Flows
    - Invalid playlist ID provided [E1]
    - Clean versions not found for some tracks [E2]
    - Authentication fails [E3]

Use Case 2: Playlist Management

1. Preconditions

- User is authenticated
- Clean playlist has been generated

2. Main Flow
   - Bot creates new playlist [S1]
   - Bot adds clean tracks to playlist [S2]
   - Bot updates playlist metadata [S3]
3. Subflows (for bot)
   - Bot creates new playlist with appropriate name [S1]
   - Bot adds tracks maintaining original order [S2]
   - Bot sets playlist description and visibility [S3]
4. Alternative Flows
   - Insufficient permissions [E1]
   - Playlist creation fails [E2]
   - Track addition fails [E3]

ID	Requirement	Priority	Use Case
R1	System must authenticate with Spotify API	Must	UC1, UC2
R2	System must be able to read existing playlists	Must	UC1
R3	System must identify explicit tracks	Must	UC1
R4	System must find clean versions of tracks	Must	UC1
R5	System must create new playlists	Must	UC2
R6	System should maintain original track order	Should	UC2
R7	System could provide progress updates	Could	UC1, UC2
R8	System would handle failed track matches	Would	UC1

Table 4.1: Table of requirements with priorities mapped to Use Cases.

Use Case	Requirements
UC1	R1, R2, R3, R4, R7, R8
UC2	R1, R5, R6, R7

Table 4.2: Table of Use Cases mapped to requirements.

4.1 Use Case Diagram

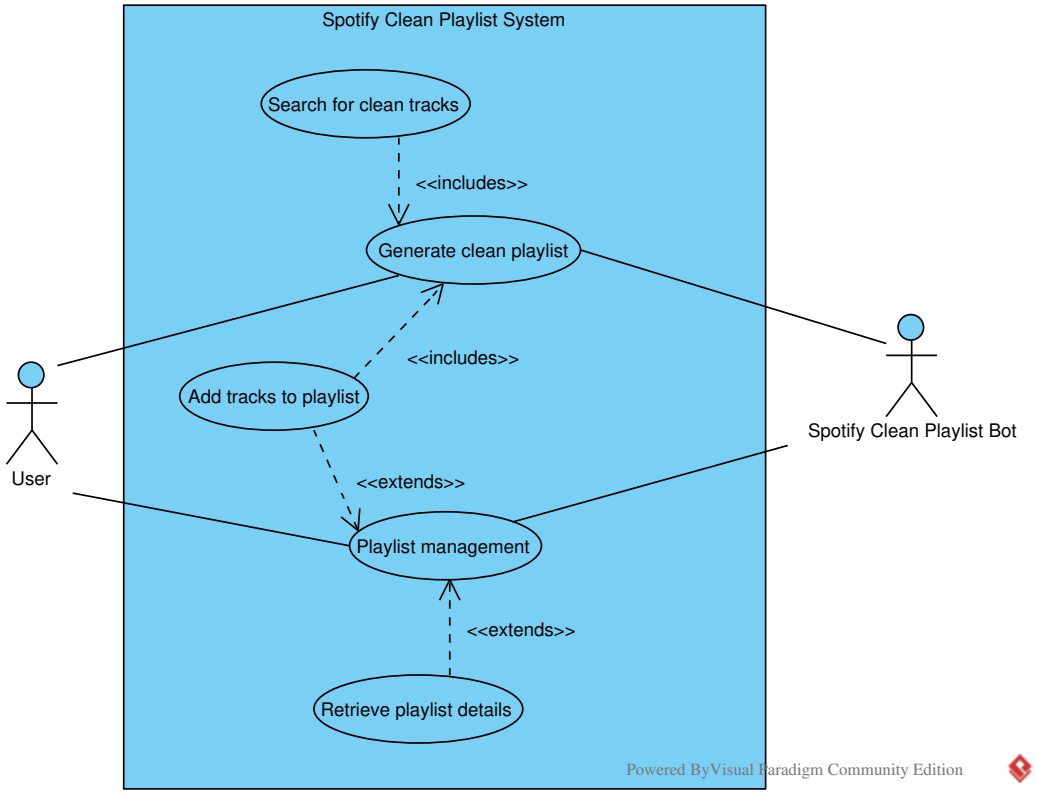

Figure 4.1: Use case diagram for the Spotify Clean Playlist Bot.

Chapter 5

Design Sketches

– Jason McCauley and Aidan Nestor

5.1 Story Board

Frame 1: Initial Interaction

- User: ”I need to create a clean version of my workout playlist”
- Bot: ”I can help you create a clean version of your playlist. Could you please share the Spotify playlist link or ID?”
- User begins interaction with the bot

Frame 2: Playlist Identification

- User: *Shares Spotify playlist link*
- Bot: ”Thanks! I’ve found your ’Workout Mix’ playlist with 25 tracks. I’ll now scan for songs with explicit lyrics and find clean versions.”
- Bot recognizes playlist and begins analysis

Frame 3: Processing

- Bot: ”Processing your playlist...
  - Found 8 explicit tracks
  - Searching for clean versions...
  - Clean versions found for 7 tracks
  - One track unavailable in clean version!”
- Bot shows real-time progress

Frame 4: Creation Confirmation

- Bot: ”I’ve created ’Clean Workout Mix’ with all available clean versions. One track (’Song A’) couldn’t be found as a clean version. Would you like the link to your new playlist?”
- User: ”Yes, please!”
- Bot ‘completes playlist creation

Frame 5: Final Result

- Bot: ”Here’s your new clean playlist: [Spotify Playlist Link]. Feel free to share it with anyone! Need anything else?”
- User receives the final clean playlist

5.2 Class Diagram

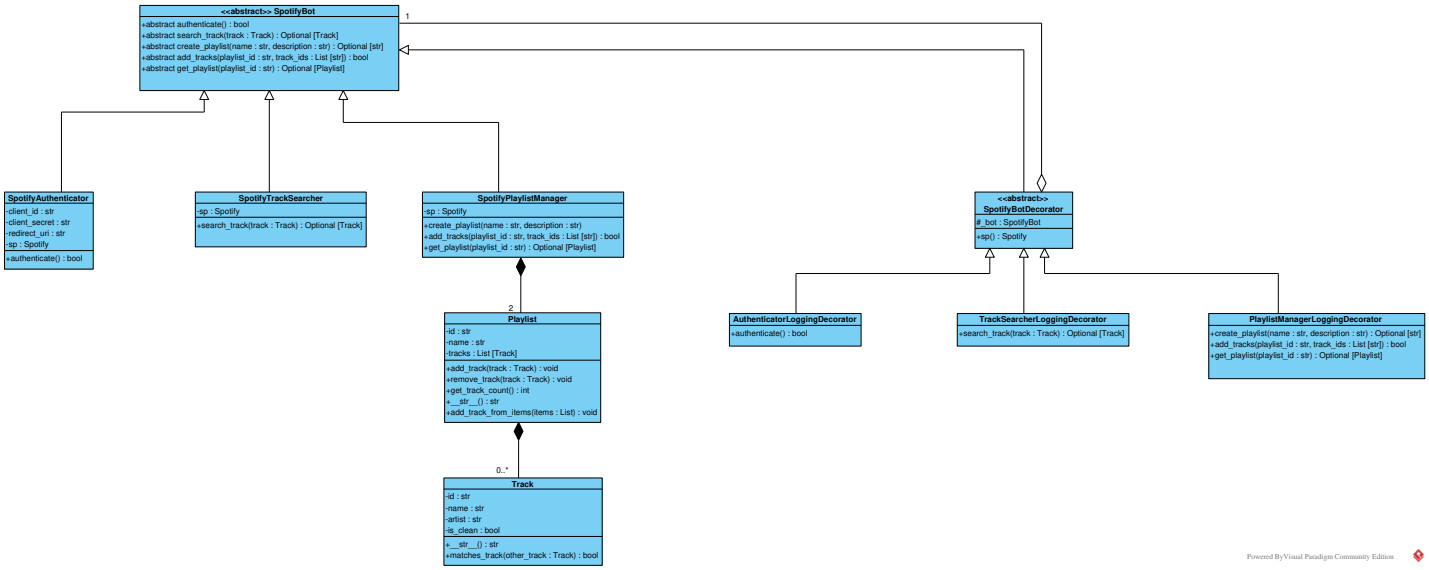

Figure 5.1: Class diagram of the Spotify Clean Playlist Bot.

```python
# base abstract SpotifyBot class
class SpotifyBot(ABC):
    @abstractmethod
    def authenticate(self) -> bool:
        pass

    @abstractmethod
    def search_track(self, track: Track) -> Optional[Track]:
        pass

    @abstractmethod
    def create_playlist(self, name: str, description: str = "") -> Optional[str]:
        pass

    @abstractmethod
    def add_tracks(self, playlist_id: str, track_ids: list[str]) -> bool:
        pass

    @abstractmethod
    def get_playlist(self, playlist_id: str) -> Optional[Playlist]:
        pass
```

Figure 5.2: Screenshot of the SpotifyBot abstract class code.

As seen in the code above, the SpotifyBot is an abstract class, with each of the methods, namely, authenticate, search_track, create_playlist, add_tracks, and get_playlist also being abstract. This is made possible due to the ”pass” keyword in Python, which, in an abstract class, indicates that they will be implemented in a concrete subclass.

```python
class SpotifyAuthenticator(SpotifyBot):
    def __init__(self, client_id: str, client_secret: str, redirect_uri: str):
        self.client_id = client_id
        self.client_secret = client_secret
        self.redirect_uri = redirect_uri
        self.sp = None

    def authenticate(self) -> spotipy.Spotify:
        # set up the authentication for Spotify API
        try:
            self.sp = spotipy.Spotify(auth_manager = SpotifyOAuth(
                client_id = self.client_id,
                client_secret = self.client_secret,
                redirect_uri = self.redirect_uri,
                scope = "playlist-modify-public playlist-read-private"
            ))
            logging.info("Successfully authenticated with Spotify Web API")
            return True
        except Exception as e:
            logging.error(f"Authentication failed: {str(e)}")
            return False

    # implement other abstract methods with pass
    def search_track(self, track: Track) -> Optional[Track]:
        pass
```

Figure 5.3: Screenshot of the SpotifyAuthenticator class code, a concrete subclass that implements the functionality of SpotifyBot.

The code above is for the SpotifyAuthenticator class, which, in correspondence with the class diagram, inherits from SpotifyBot class, making it a concrete subclass. Furthermore, it provides the implementation for the authenticate method, which was abstract in the SpotifyBot parent class. While we are not providing the implementation for all of the abstract methods in this subclass, we still have to define their signatures, since we are inheriting from an abstract class. To do this, we use the ”pass” keyword again, allowing them to be implemented elsewhere. We can see this begin with the search_track method, as it is not implemented in the SpotifyAuthenticator class.

```python
class SpotifyTrackSearcher(SpotifyBot):
    def __init__(self, spotify_client: spotipy.Spotify):
        self.sp = spotify_client

    def search_track(self, track: Track) -> Optional[Track]:
        # search for clean version of a track
        try:
            query = f"{track.name} {track.artist} clean"
            results = self.sp.search(q = query, type = "track", limit = 50)

            for item in results['tracks']['items']:
                # instantiate Track object from search result
                found_track = Track(
                    id = item['id'],
                    name = item['name'],
                    artist = item['artists'][0]['name'],
                    is_clean = not item.get("explicit", False)
                )

                # check if this is the clean version of our track
                if found_track.matches_track(track) and found_track.is_clean:
                    return found_track

            return None
        except Exception as e:
            logging.error(f"Error searching for track: {str(e)}")
            return None
```

Figure 5.4: Screenshot of the SpotifyTrackSearcher class code, a concrete subclass that implements the functionality of SpotifyBot.

The code above is for the SpotifyTrackSearcher concrete class, which, again, in correspondence with the class diagram, inherits from the abstract SpotifyBot class. Additionally, it provides the implementation for the search_track method, which was also abstract in the SpotifyBot parent class. While it cannot be seen in the screenshot, all of the other abstract methods still have to be defined in this class, since they are inherited from an abstract class. Because we only want to provide the concrete implementation for the search_track method in this class, we use the ”pass” keyword on the rest of the inherited methods.

5.3 Design Pattern

```python
class SpotifyBotDecorator(SpotifyBot, ABC):
    def __init__(self, spotify_bot):
        self._bot = spotify_bot

    @property
    def sp(self):
        return self._bot.sp
```

Figure 5.5: Screenshot of the SpotifyBotDecorator abstract class code, which inherits from SpotifyBot and takes an instance of SpotifyBot as an argument upon instantiation.

The code above is for the SpotifyBotDecorator abstract class. The design pattern we chose to implement is the Decorator pattern, and we did this by creating an abstract SpotifyBotDecorator that inherits from the SpotifyBot class. Additionally, we demonstrate aggregation, as an instance of the SpotifyBot has to be passed into the SpotifyBotDecorator upon instantiation. However, we

know that you cannot create instances of abstract classes – instead, we are passing in an instance of one of the concrete subclasses of the SpotifyBot abstract class, since they retain the SpotifyBot type.

```python
# concrete decorator classes
class AuthenticatorLoggingDecorator(SpotifyBotDecorator):
    def authenticate(self) -> bool:
        print("\n--- Authentication Process ---")
        start_time = time.time()

        result = self._bot.authenticate()

        processing_time = time.time() - start_time
        print(f"Authentication {'Successful' if result else 'Failed'}")
        print(f"Processing Time: {processing_time:.2f} seconds")

        return result

    # delegate other methods to the wrapped bot
    def search_track(self, track: Track) -> Optional[Track]:
        return self._bot.search_track(track)

    def create_playlist(self, name: str, description: str = "") -> Optional[str]:
        return self._bot.create_playlist(name, description)

    def add_tracks(self, playlist_id: str, track_ids: list[str]) -> bool:
        return self._bot.add_tracks(playlist_id, track_ids)

    def get_playlist(self, playlist_id: str) -> Optional[Playlist]:
        return self._bot.get_playlist(playlist_id)
```

Figure 5.6: Screenshot of the AuthenticatorLoggingDecorator class code, a concrete subclass that implements the functionality of SpotifyBotDecorator for the SpotifyAuthenticator concrete class.

We then have the AuthenticatorLoggingDecorator, which inherits from the abstract SpotifyBotDecorator class. Because the SpotifyBotDecorator is an abstract class as well, we cannot directly create an instance of it – instead, we create an instance of the concrete subclass, which in this case, is the AuthenticatorLoggingDecorator. By inheriting from the SpotifyBotDecorator, we have access to the instance of the SpotifyBot class, which is really an instance of the SpotifyAuthenticator class that has retained the SpotifyBot type of its parent abstract class. From here, we are able to ”decorate” the SpotifyAuthenticator class by providing additional functionality to its existing methods. As seen in the screenshot above, we are decorating the SpotifyAuthenticator’s authenticate method by logging the time it takes to authenticate, as well as if the authentication succeeded or failed.

```python
class TrackSearcherLoggingDecorator(SpotifyBotDecorator):
    def search_track(self, track: Track) -> Optional[Track]:
        print(f"\n--- Searching for Clean Version ---")
        print(f"Track: {track}")
        start_time = time.time()

        result = self._bot.search_track(track)

        processing_time = time.time() - start_time
        if result:
            print(f"Found clean version: {result}")
        else:
            print("No clean version found")
        print(f"Search Time: {processing_time:.2f} seconds")

        return result

    # delegate other methods
    def authenticate(self) -> bool:
        return self._bot.authenticate()

    def create_playlist(self, name: str, description: str = "") -> Optional[str]:
        return self._bot.create_playlist(name, description)

    def add_tracks(self, playlist_id: str, track_ids: list[str]) -> bool:
        return self._bot.add_tracks(playlist_id, track_ids)
```

Figure 5.7: Screenshot of the TrackSearcherLoggingDecorator class code, a concrete subclass that implements the functionality of SpotifyBotDecorator for the SpotifyTrackSearcher concrete class.

Similarly, we have the TrackSearcherLoggingDecorator, which also inherits from the SpotifyBotDecorator abstract class. By inheriting from the SpotifyBotDecorator, we again gain access to that instance of the SpotifyBot class, which is really an instance of the SpotifyTrackSearcher class that retains the type of the parent abstract class. Like the previous example, we are now able to ”decorate” the SpotifyTrackSearcher class by providing additional functionality to its existing methods, for instance, the search_track method. As seen in the screenshot of the code above, we are decorating the SpotifyTrackSearcher’s search_track method by logging the track we are searching for a clean version of, whether or not a clean version was found, and the time it took to do so.

5.4 Object Diagram

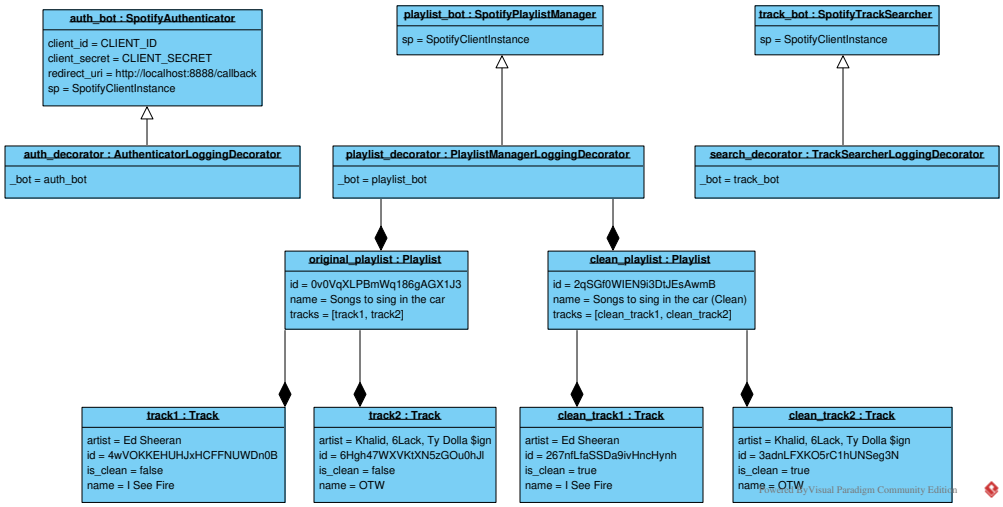

Figure 5.8: Object diagram of the Spotify Clean Playlist Bot.

5.5 Sequence Diagram

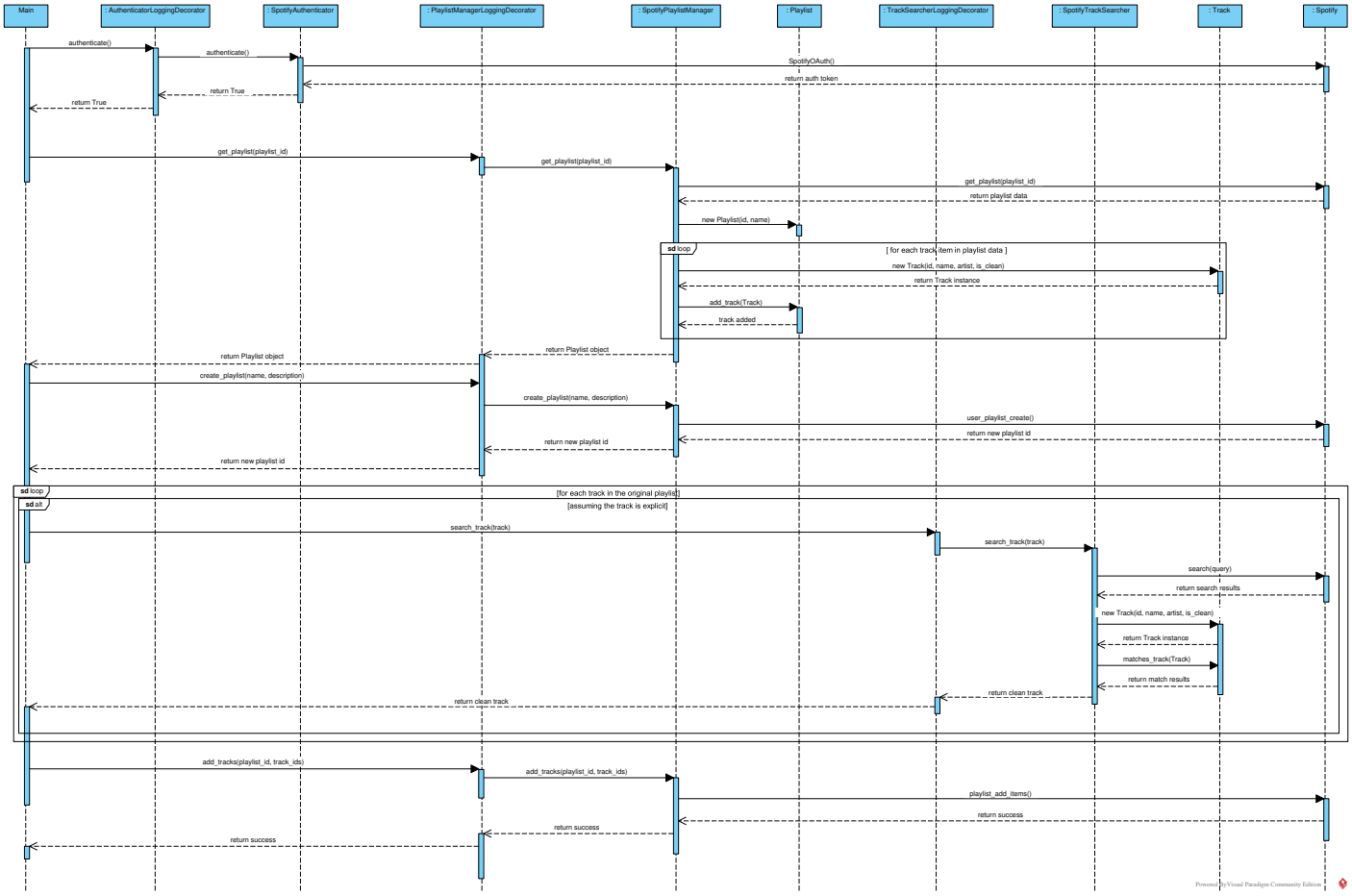

Figure 5.9: Sequence diagram of the Spotify Clean Playlist Bot in action.

5.6 State Machine Diagram

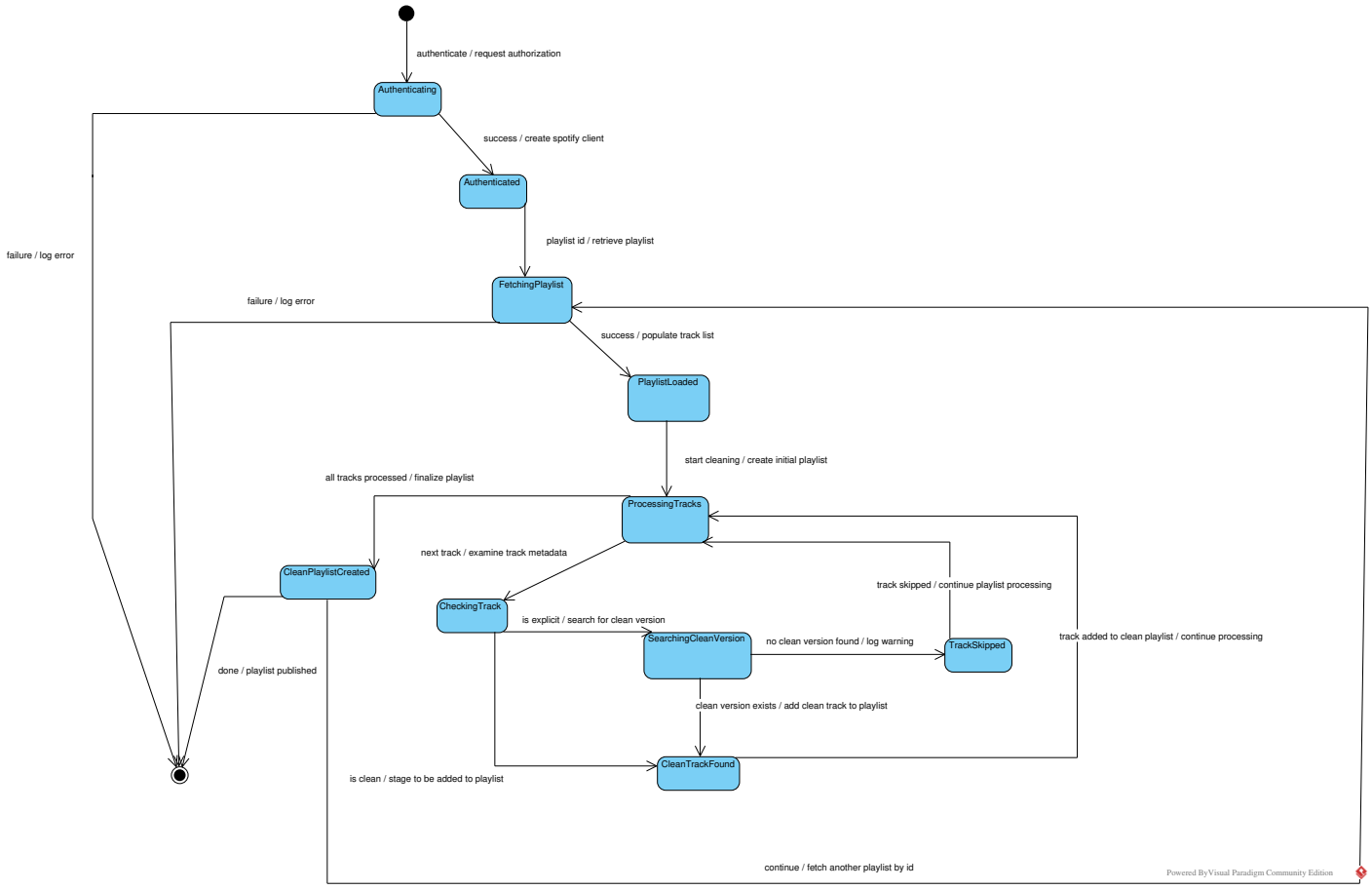

Figure 5.10: State machine diagram of the Spotify Clean Playlist Bot.

Chapter 6

Architecture Design

– Jason McCauley and Aidan Nestor

6.1 Component Diagram

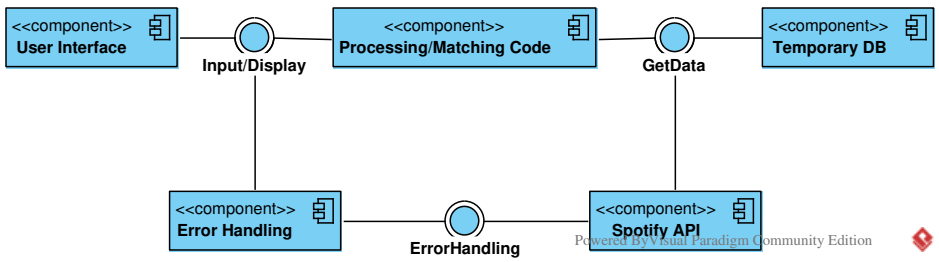

Figure 6.1: Component diagram of the Spotify Clean Playlist Bot.

The User Interface component is where the user will interact with the bot. This is where they will log in to their Spotify account and upload the playlist they want to make clean. This component interacts with the Processing/Matching Code component that is responsible for parsing through the explicit playlist and creating the new clean playlist. It then interacts with the User Interface to display the clean playlist. The Error Handling component interacts with the user input/display to return and display any error caused by the data the user inserts that the Processing/Matching Code returns. It also interacts with the Spotify API component to display any error that the API gives to the user. The Spotify API component interacts with the Temporary DB component to store songs/data temporarily. It also interacts with the Processing/Matching Code component as the code must grab data from the API and the code can also store data in the Temporary DB component.

6.2 Activity Diagrams for Use Cases 1 and 2

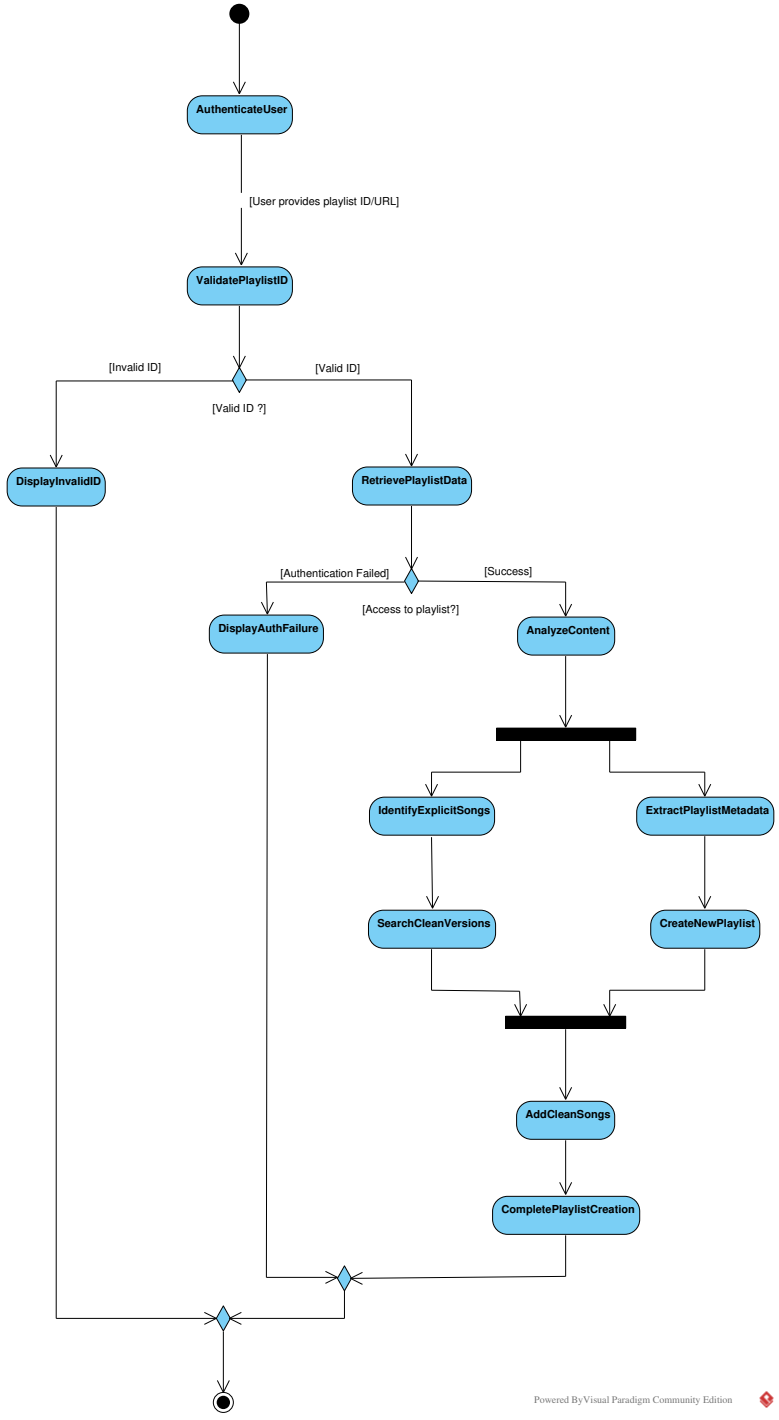

Figure 6.2: Activity diagram of Use Case 1 for the Spotify Clean Playlist Bot.

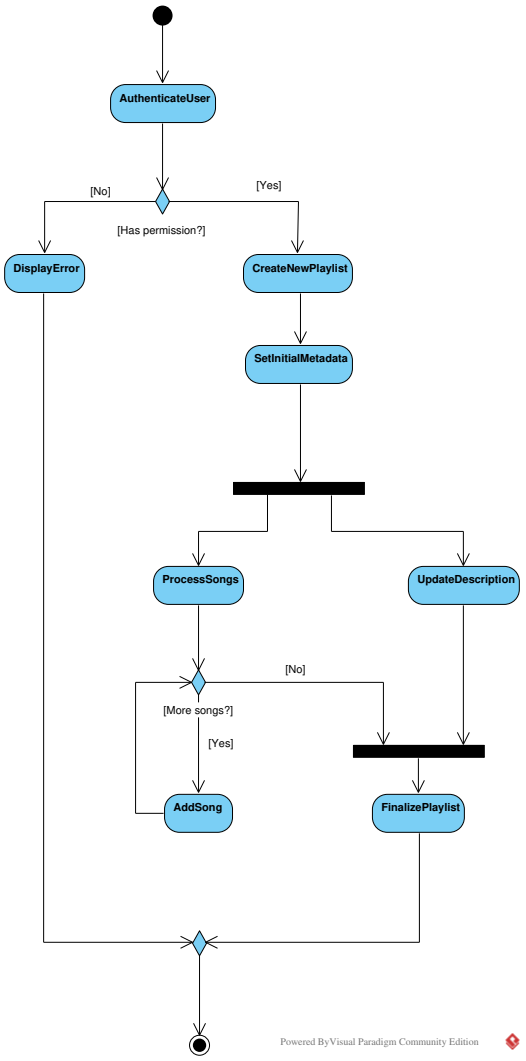

Figure 6.3: Activity diagram of Use Case 2 for the Spotify Clean Playlist Bot.

6.3 Package Diagram

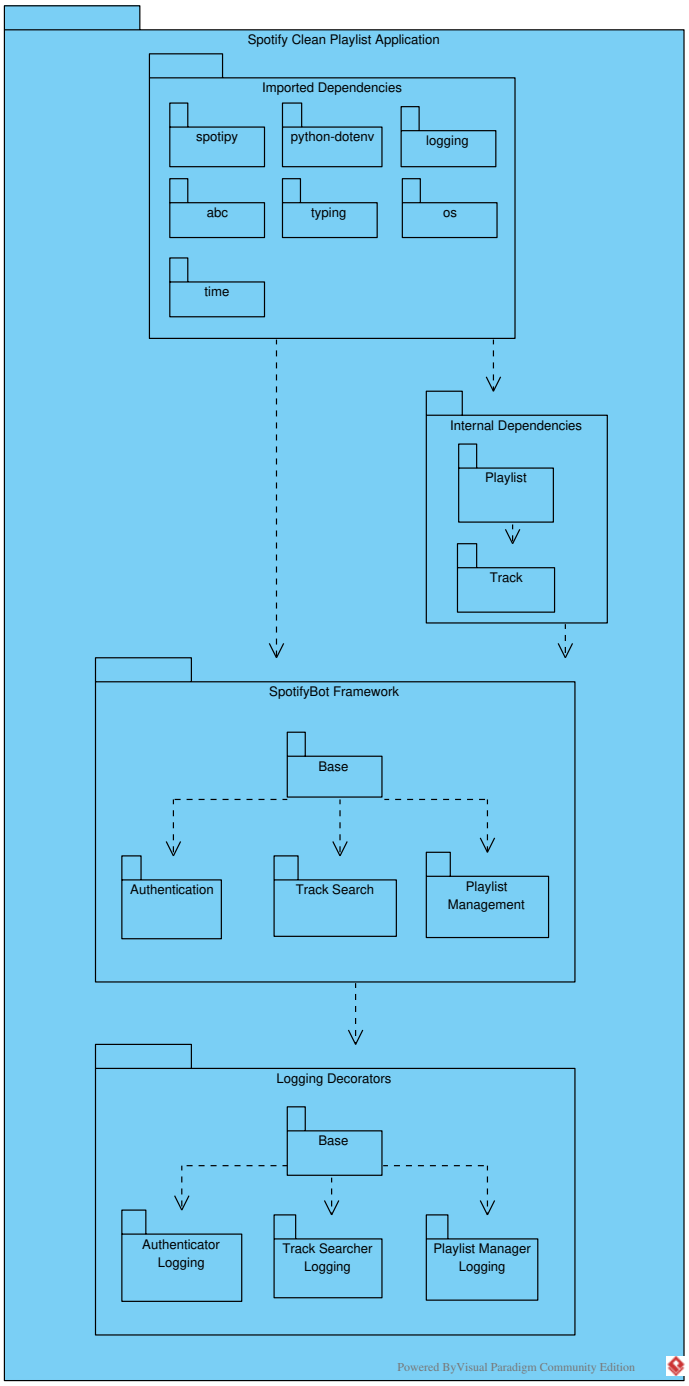

Figure 6.4: Package diagram for the Spotify Clean Playlist Bot.

6.4 Composite Structure Diagram

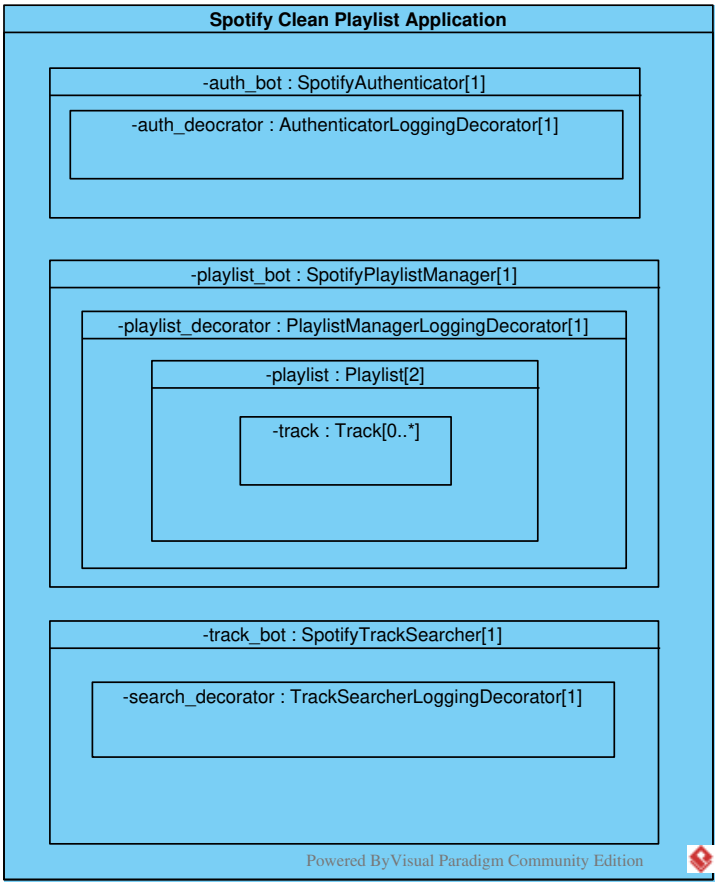

Figure 6.5: Composite structure diagram for the Spotify Clean Playlist Bot.

6.5 Profile Diagram

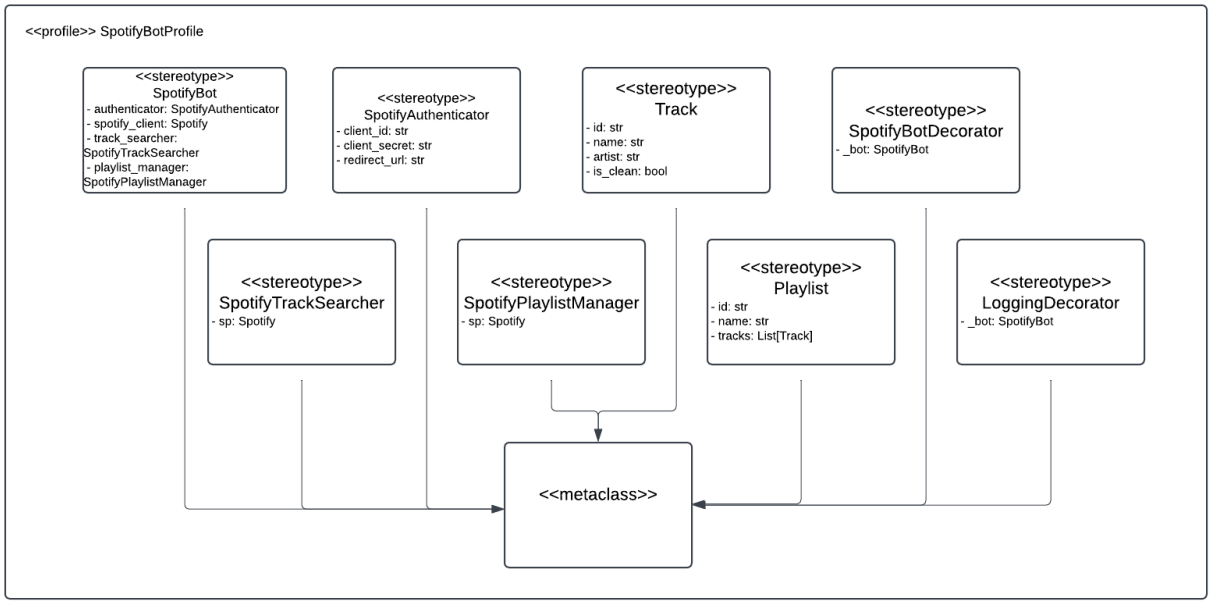

Figure 6.6: Profile diagram for the Spotify Clean Playlist Bot.

6.6 Interaction Overview Diagram

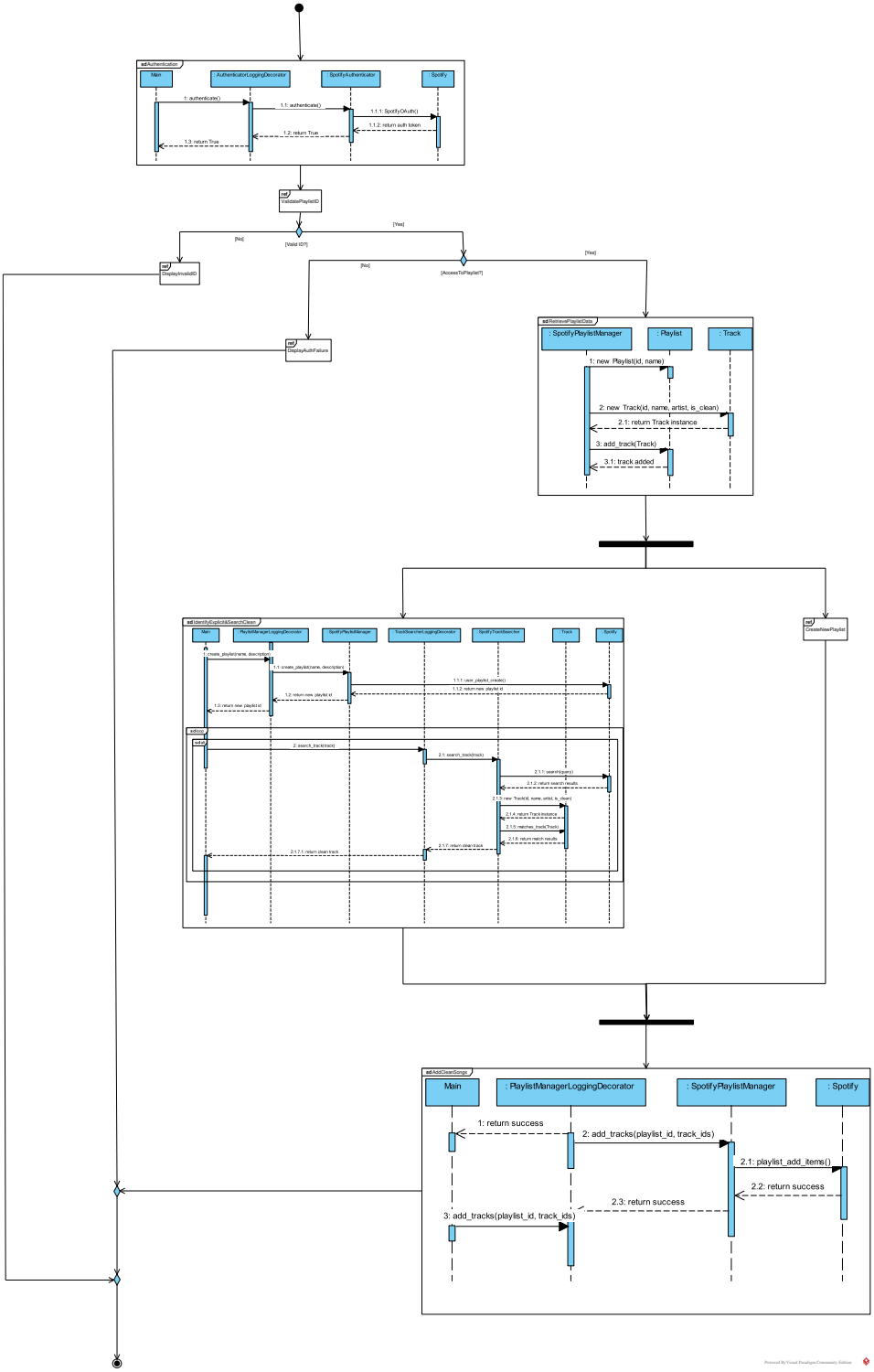

Figure 6.7: Interaction Overview diagram for the Spotify Clean Playlist Bot.

6.7 Communication Diagram

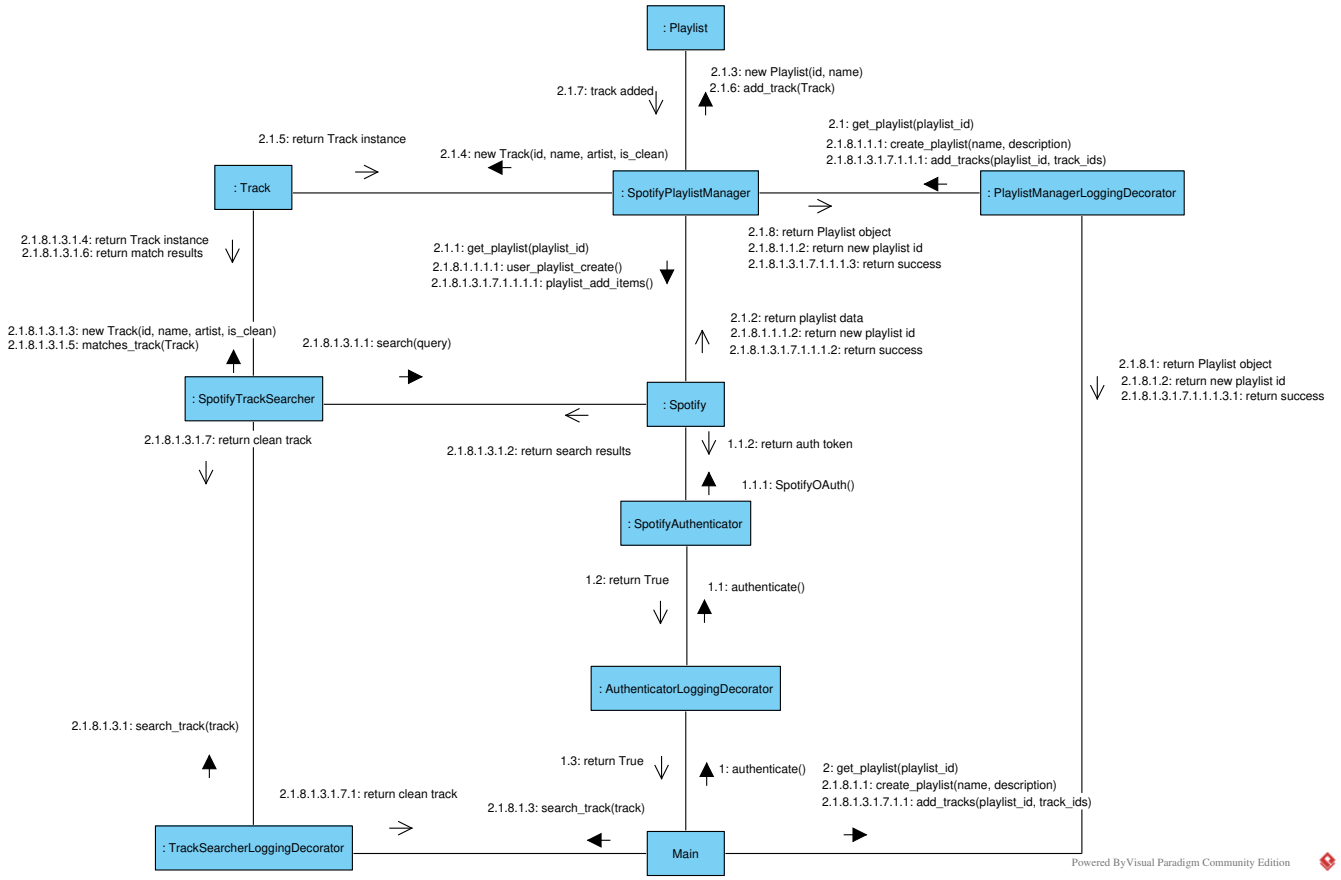

Figure 6.8: Communication diagram for the Spotify Clean Playlist Bot.

Chapter 7

Agile Process

– Jason McCauley and Aidan Nestor

7.1 Meeting Notes

During our first meeting with Dr. Muresan on December 3, 2024, we were able to discuss our project with him at a more personal level. So far, he likes the direction that we are headed in, but suggested a few minor corrections in our current diagramming. For instance, there should be a diamond at the final junction of our activity diagrams, rather than a black bar. Additionally, we should have references to the specific use cases throughout the project notebook. These suggestions will certainly be taken into account, and we will work to implement them for the final submission. Dr. Muresan also suggested that we create physical CRC cards for the next meeting, rather than on Visual Paradigm, as they will be easier to discuss and interact with.

During our second meeting with Dr. Muresan on December 5, 2024, we were able to discuss our project in much more depth, beginning with our physical CRC cards. The logic behind our program appears to be sound, however, his biggest gripe was with our SpotifyBot class, which currently comes across as a ”God class”. Furthermore, he suggested separating its functionality into separate classes, for instance, one class to authenticate the Spotify API, another class to search for tracks, and another class to create the playlist. I also got the chance at the end of class to briefly discuss the design pattern that I planned on implementing, the decorator pattern. I wanted this to add functionality to the SpotifyBot, logging information as the program runs, such as the list of explicit songs, songs that did not have a clean version, and the run time. Originally, I made this decorator concrete, but he made me realize that the decorator should be abstract, with concrete subclasses to implement the specific logging. As we work towards presenting our diagrams and code next week, and eventually submitting the final report, these suggestions will certainly be kept in mind and implemented. Also, it is worth mentioning that he liked our current Agile process, keeping track of our issues, assigning them to users, and designating story points through a Kanban Board.

Our third meeting occurred in a Discord call after our final project presentations were graded. We saw the feedback provided by Dr. Muresan on the original iterations of the use case diagram, class diagram, and state machine diagram and made sure we were on the same page with how their designs could be improved. With these critiques in mind, we went back to the drawing board and made adjustments – for instance, making the use case diagram less of an activity diagram, and only

using a few clear use cases for the actors. Furthermore, improving the state machine diagram to allow the user to fetch another playlist without having to exit the program. We also had to make changes to our initial class diagram, as it did not describe a decorator pattern. Correspondingly, we had to adjust our sequence diagram. After making these changes, we were able to continue with the last of the diagrams in our project, ensuring that they better aligned with what Dr. Muresan expected.

7.2 Issue Tracking

Regarding issue tracking, we decided to use the Kanban Board built into our GitHub repository, allowing us to create tasks, assign them to people, and designate story points in a single easy-to-access location. Our goal, of course, as we work towards the final report submission, is to have as many of these tasks moved into the ”Done” column.

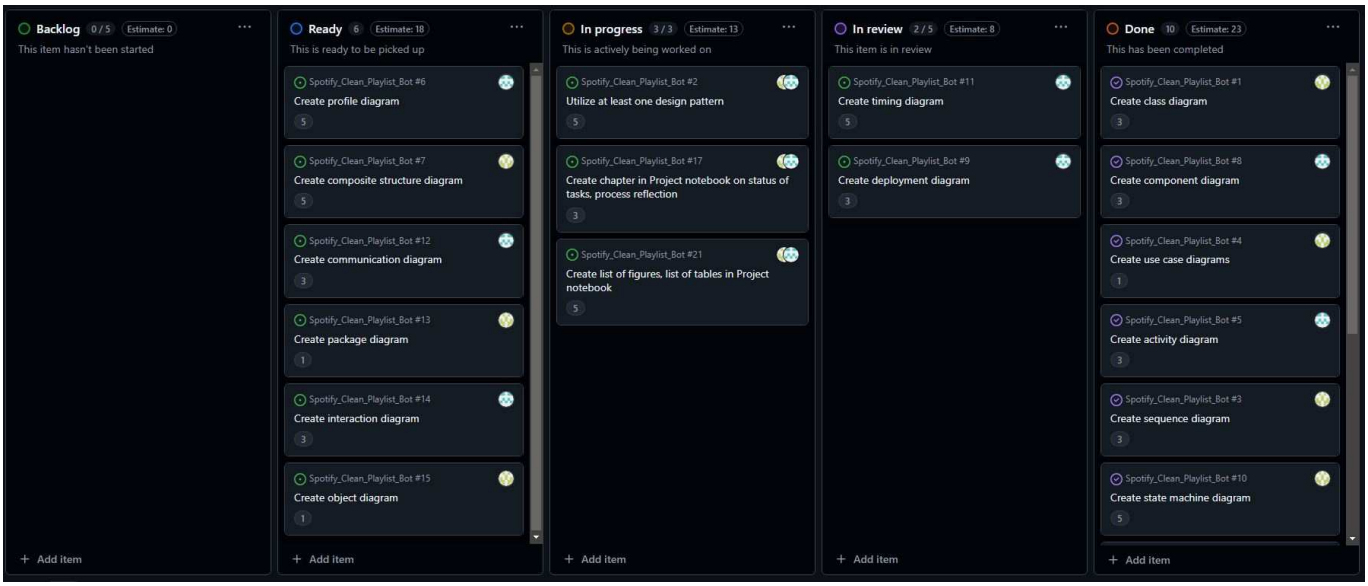

Figure 7.1: Kanban Board, as of December 7, 2024. Each task is assigned to at least one group member, and story points are designated to estimate the overall effort required to complete each task.

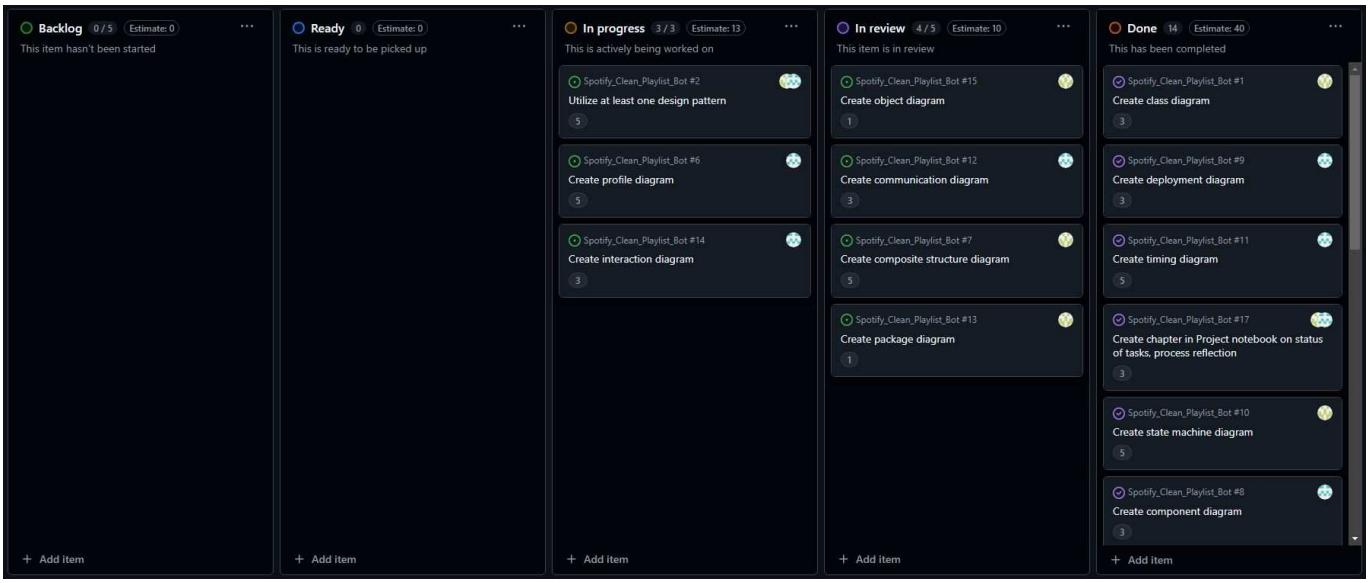

Figure 7.2: Kanban Board, as of December 12, 2024. As seen in the image above, every task had been picked up at this point. A decent amount of tasks were completed, with the rest either being in progress or review.

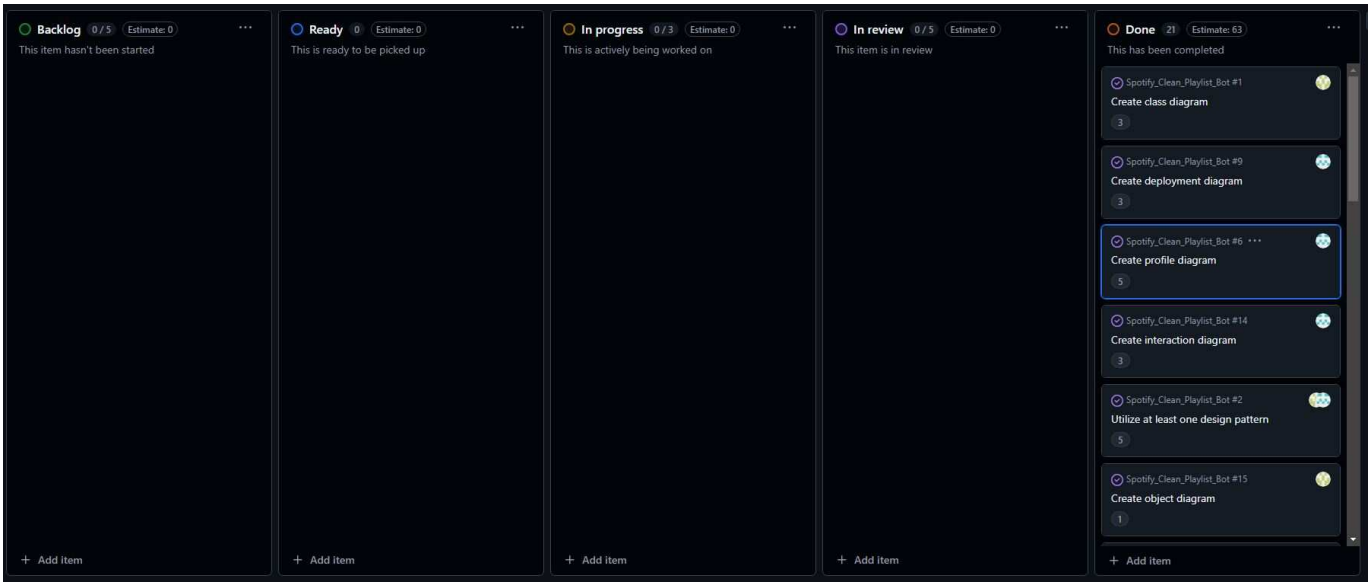

Figure 7.3: Kanban Board, as of December 19, 2024, the day of the final report submission. As seen in the image above, all tasks have been successfully completed.

Chapter 8

Process Reflection

– Jason McCauley and Aidan Nestor

8.1 Status of Complete Tasks

Regarding the currently completed tasks, Aidan and I successfully completed the introduction chapter in the project notebook describing our responsibilities and the level of effort provided. Additionally, we created a list of figures, a list of tables, a Bibliography with a current list of references, and an Index chapter at the end of the notebook. We also created a chapter on the Agile process we followed, and while this is considered done for the moment, we will have to regularly update it to demonstrate the progress being made to our tasks on the Kanban Board.

While we originally created the class diagram, component diagram, use case diagrams, activity diagrams, sequence diagram, timing diagram, and state machine diagrams for the presentation, we had to rework many of their components to better align with the suggestions and feedback of Dr. Muresan. From there, we were then able to develop the rest of our diagrams, namely the profile diagram, interaction diagram, communication diagram, composite structure diagram, package diagram, deployment diagram, and object diagram. At the culmination of this project, with all tasks being complete, Aidan and I did a wonderful job evenly distributing the work, whether it be setting up the Overleaf notebook, or creating UML diagrams. Furthermore, we did a great job asking each other questions and communicating our ideas to ensure we were always on the same page throughout the development process.

8.2 Status of Incomplete Tasks

At the submission of this project report, there are no incomplete tasks. Additionally, this can be seen in the screenshot of the Kanban Board from December 19, 2024 in the previous chapter. All tasks, whether it be creating the different UML diagrams, or setting up parts of the Overleaf notebook, such as the different chapters, the list of tables and figures, were completed evenly and in a timely manner.

Bibliography

[1] *Learning Guides. FREE Learning Resources: UML, Agile, TOGAF, PMBOK, BPMN*. Visual Paradigm. URL: https://www.visual-paradigm.com/guide/ (visited on 12/07/2024).

Index