Software-Entwicklung 2

Sommersemester 2023

Whalex App

Robin Schmidt
44783 | rs141@hdm-stuttgart.de

Marvin Helmer
44479 | mh380@hdm-stuttgart.de

https://gitlab.mi.hdm-stuttgart.de/rs141/whalex-app

1. Inhaltsverzeichnis

1. Inhaltsverzeichnis 2
2. Disclaimer 3
3. Kurzbeschreibung 3
4. Startklasse 3
5. Besonderheiten 4
5.1. Git LFS - Large File Storage 4
5.2. Dateipfade unter UNIX Systemen 5
5.3. Löschen von Sound Objekten 5
6. UML-Diagramme 6
6.1. Use-Case Diagramm 6
6.1.1. Login 6
6.1.2. Hauptanwendung 7
6.2. Klassendiagramm 8
6.3. Sequenzdiagramm 8
7. Stellungnahme 9
7.1. Architektur 9
7.2. Clean Code 10
7.3. Tests 10
7.4. GUI (JavaFX) 10
7.5. Logging/Exceptions 11
7.6. UML 11
7.7. Threads 12
7.8. Streams und Lambda-Funktionen 12
8. Bewertungsbogen 12

2. Disclaimer

Wir möchten Sie darauf hinweisen, dass die in dieser Dokumentation vorhandenen Links auf die entsprechenden Klassen in unserem GitLab Repository führen. Also haben Sie keine Scheu draufzuklicken! Die UML Diagramme sind ebenfalls anklickbar und führen Sie ebenfalls zu den entsprechenden Dateien im Repository. Ebenso werden Verweise auf andere Abschnitte der Dokumentation verwendet.

3. Kurzbeschreibung

Unsere Applikation "Whalex", aus "Whale" (engl. Wal) und "ex" als Kurzform von (engl. Relax), stellt eine Audio-Bibliothek mit integriertem Musikplayer dar, in der sich eigene Sounds sowie Playlisten anlegen und abspielen lassen. Hierbei steht das Thema der Wale natürlich im Vordergrund, da Walgesänge zum entspannen anregen und man durch sie in unvergessliche Traumwelten abschweift.

Beim Start der Anwendung ist vorerst ein Nutzerprofil zu erstellen. Nach der Registrierung sind bereits verschiedene Walgesänge auf der Benutzeroberfläche zu sehen. Es ist möglich, Audiodateien sowie Playlisten über den orangen Button hinzuzufügen und über das Kontextmenü wieder zu löschen.

Das Programm verwaltet nicht nur die eigenen Audios, sondern auch die der anderen Nutzer. Um eine übersichtlichere Auflistung aller Sounds und Playlisten zu erhalten, öffnet sich die Anwendung standardmäßig in der “Home” Ansicht. Um hingegen nur seine eigenen Elemente zu sehen, kann auf den Reiter “Profile” navigiert werden.

Durch die Suchleiste am oberen Fensterrand können die verschiedenen Elemente durchgesucht werden. So kann beispielsweise einfach nach vorkommenden Zeichen gesucht werden. Ebenfalls können nur Elemente mit bestimmten Usern, Sounds oder Playlisten angezeigt werden.

Beispiel: (user:Marvin) → Zeigt alle Elemente von “Marvin” an
(sound:Wal) → Zeigt alle Sounds an, die “Wal” enthalten
(playlist:Ocean) → Zeigt alle Playlisten an, die “Ocean” enthalten

Einzelne Sounds und sogar ganze Playlisten können entweder über den Button im Cover oder über das Kontextmenü zur “Queue” hinzugefügt werden und sind nach dem Abspielen in der “History” zu sehen.
Ebenfalls kann man über die Einstellungen das Profilbild und Passwort des Benutzers ändern und als weitere Option kann man sein eigenes Profil mit allen Abhängigkeiten löschen.

4. Startklasse

Die Klasse zum Starten unserer Anwendung heißt WhalexApp.

Zum Auslieferungszustand sind noch keine Beispielobjekte vorhanden. Da die Anwendung ausschließlich auf Windows entwickelt wurde und die Dateipfade nicht plattformunabhängig funktionieren. Daher bitte zuerst Dateipfade unter UNIX Systemen beachten.
Wenn Sie nun die Anwendung mit Testdaten versehen möchten, muss die InitializeAdminUsers Klasse separat ausgeführt werden.
Anschließend können Sie die Anwendung über die main Methode in der WhalexApp Klasse starten.
Die Anmeldedaten für die beiden Testnutzer sind folgende:
User 1:
username: “Robin”
password: “123”

User 2:
username: “Marvin”
password: “123”

Beide zum Start relevanten Klassen sind unter der folgenden Struktur vorzufinden:

```
.
├──src
    ├──main
        ├──java
            └──org
                ├──example
                    └──WhalexApp.java
                    └──InitializeAdminUsers.java
```

5. Besonderheiten

5.1. Git LFS - Large File Storage

In unserem Projekt verwenden wir Git LFS, was uns die Auslagerung bestimmter Dateien (.wav, .mp3, .jpg, .png) ermöglicht. Dadurch entsteht weniger Datentraffic im Repository, was performantere Git Operationen sicherstellt. Die vollständige Dokumentation zu dem LFS von Gitlab finden sie unter: https://docs.gitlab.com/ee/topics/git/lfs/

Vor dem ersten Klonen des Projektes sollte demnach der Git LFS Client auf Ihrem Gerät installiert sein, um die vollwertigen Dateien zu laden.

5.2. Dateipfade unter UNIX Systemen

Durch die Serializer und Deserializer der einzelnen Objekte werden, nur unter Windows, valide Dateipfade gespeichert. Um dies für Ihr System anzupassen, muss der Dateipfad für die Sounds in data –> SoundSerializer (Zeile 24)

von:

```
jsonObject.addProperty("path","src\\main\\resources\\data\\sounds\\" +
decodedString.substring(decodedString.lastIndexOf('/')+1));
```

zu:

```
jsonObject.addProperty("path","src/main/resources/data/sounds/" +
decodedString.substring(decodedString.lastIndexOf('/')+1));
```

geändert werden. Anschließend können Sie wieder unter der Startklasse fortfahren.

5.3. Löschen von Sound Objekten

Unter Windows funktioniert das Löschen der Sound Objekte und deren in das Projekt kopierte Sounddatei nicht, da hier ein Java Prozess bereits die Dateien schon beim Anlegen eines Media Objektes verwendet. Somit besteht während der Anwendung immer ein Zugriff auf diese Datei, was ein Löschen aus der Applikation heraus nicht möglich macht.

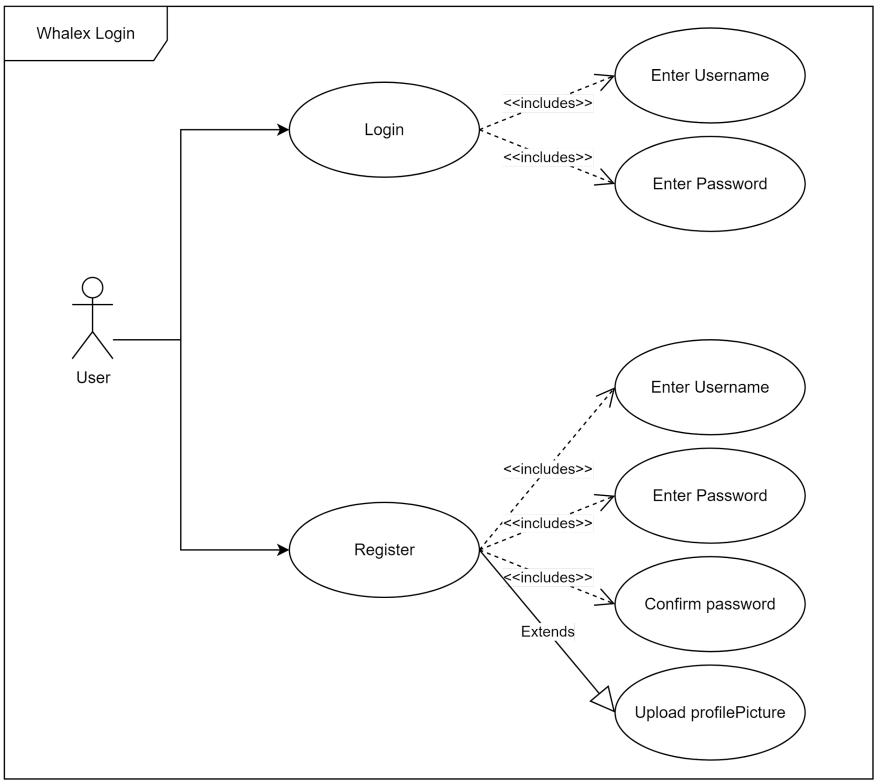
Da dieses Verhalten jedoch auch von Datei zu Datei unterschiedlich ist, haben wir 4 .wav Dateien als Testdaten hinterlegt, welche bei uns zumindest auf 3 verschiedenen Windows Systemen einwandfrei liefen.

Unter Linux tritt dieses Phänomen nicht auf: Die Dateien, welche bei uns das beschriebene Problem hervorrufen, können auf Linux ohne Probleme gelöscht werden.

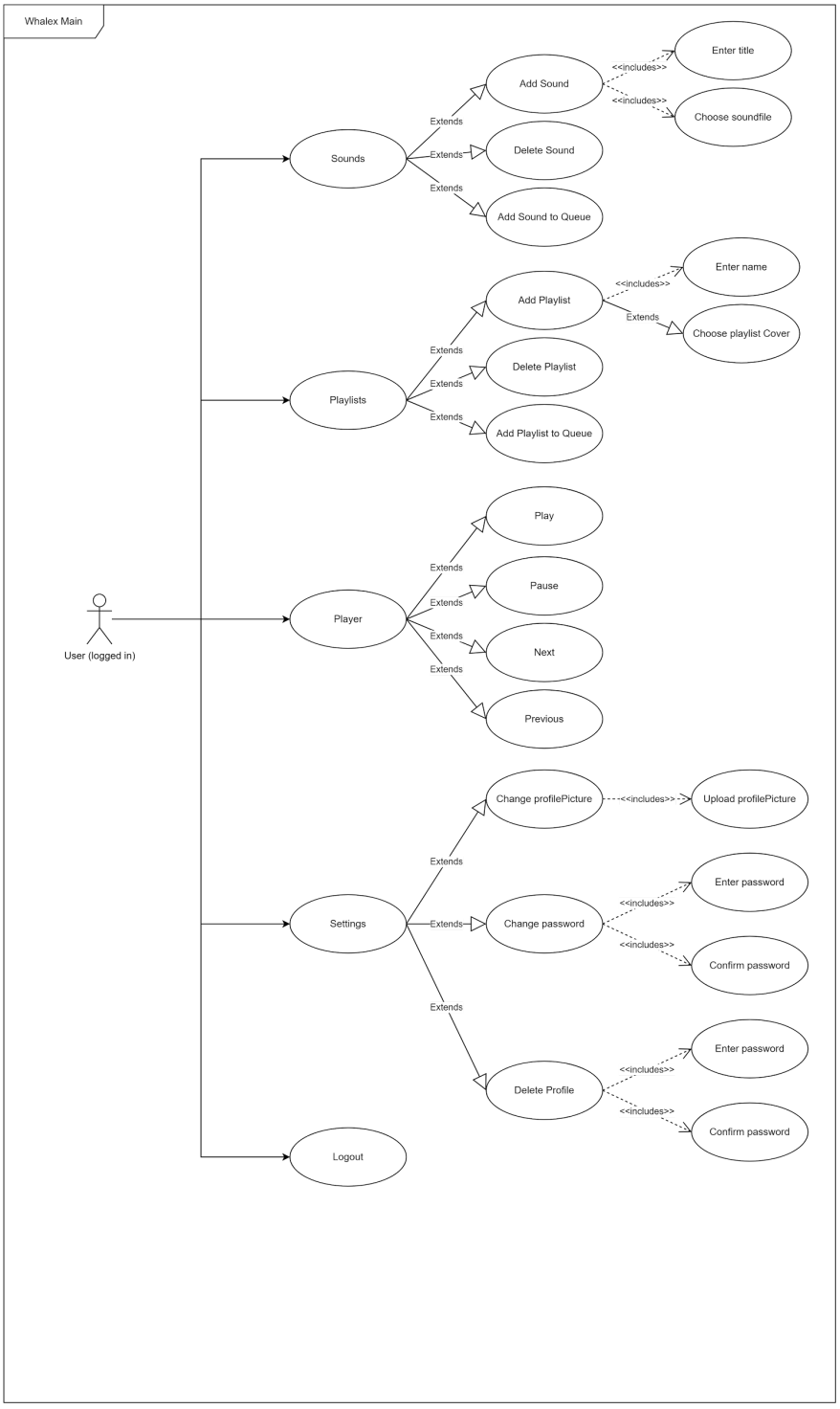
6. UML-Diagramme

6.1. Use-Case Diagramm

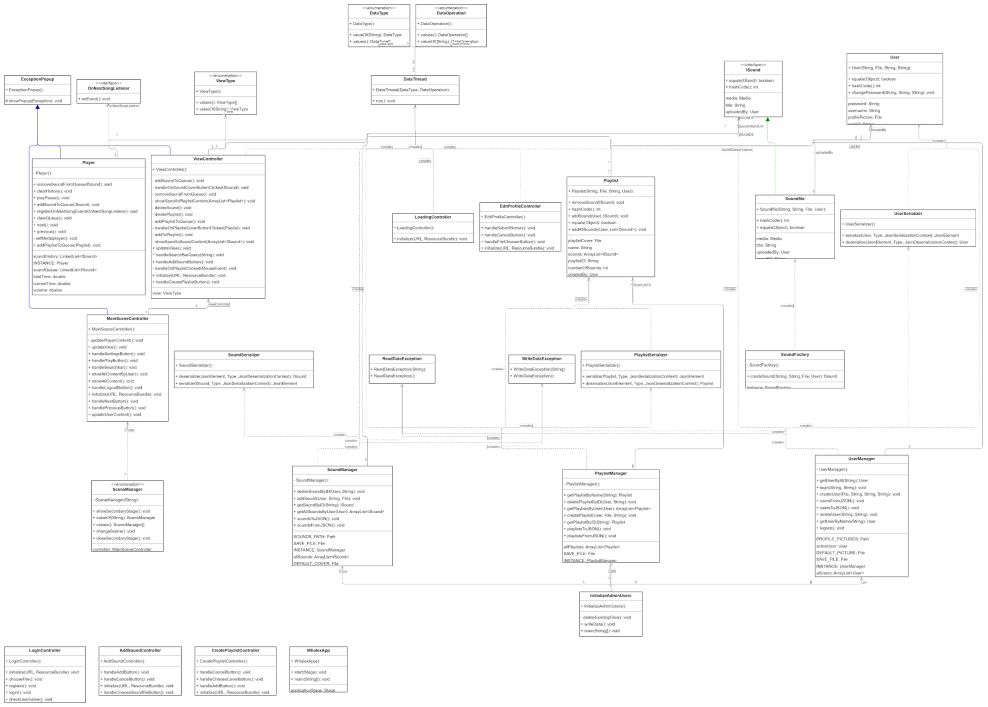
6.1.1. Login



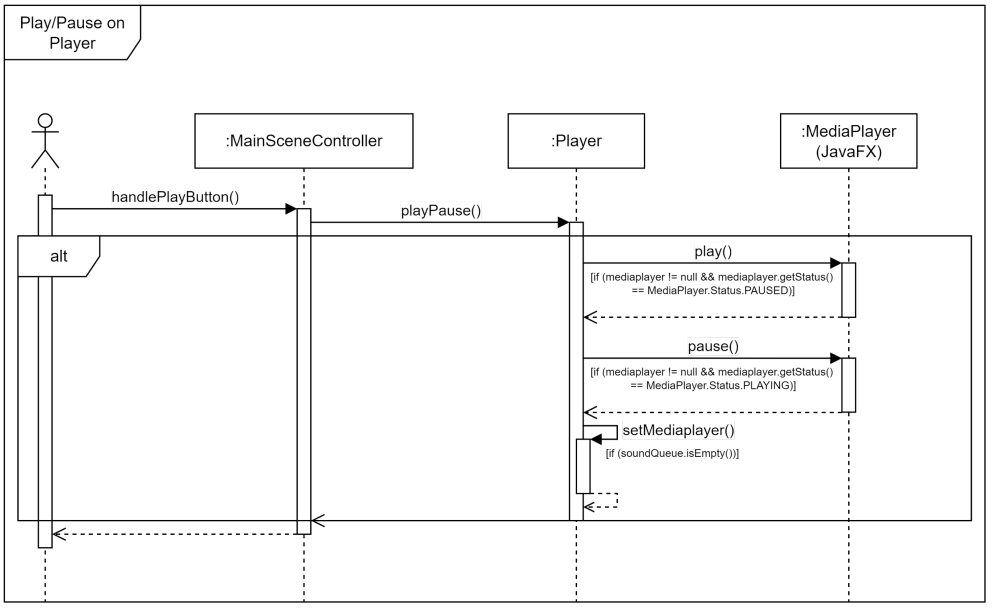
6.1.2. Hauptanwendung



6.2. Klassendiagramm



6.3. Sequenzdiagramm



7. Stellungnahme

7.1. Architektur

Singleton Access: Bei der Architektur unserer Anwendung haben wir uns bei den grundlegenden Klassen (WhalexApp, SoundFactory, UserManager, SoundManager, PlaylistManager und Player) für das Singleton Pattern entschieden. Da diese Klassen an verschiedenen Stellen des Programms verfügbar sein müssen und wir sie für den Programmablauf nur einmal benötigen. Somit wird sichergestellt, dass wir immer auf dieselbe Instanz zugreifen und das Prinzip der Objektorientierung erhalten bleibt.

Interface: Für die Sounds haben wir das Interface ISound erstellt, welches durch verschiedene Implementierungen andere Ressourcentypen darstellen kann. Durch die getMedia() Methode stellen wir sicher, dass jedes ISound Objekt auch durch den verwendeten JavaFX Mediaplayer abgespielt werden kann. Zudem gibt es noch das OnNextSongListener Interface, welches als Eventtrigger für den Player dient, um nach Abspielen eines Sounds die Queue und History entsprechend zu aktualisieren.

Factories: Wir verwenden eine SoundFactory, um die gewünschte Sound Implementierung zu instanzieren. Außerdem werden im ViewController Factories verwendet, um JavaFX Interfaces zu implementieren, damit benutzerdefinierte JavaFX Elemente in der TableView angezeigt werden können.

Packages: Wir haben unsere Klassen in verschiedene Packages unterteilt, um die Berechtigungen für außenstehende Klassen einzuschränken und Funktional zusammengehörige Klassen zu schützen.

Controller: Für unsere Anwendung haben wir hauptsächlich vom Scene Builder erstellte FXML Elemente benutzt, welche alle über einen extra Controller verfügen.

Enums: Wir verwenden an mehreren Stellen unseres Programms Enums, um die falsche Verwendung von Methoden durch zuvor definierte Enum Werte zu minimieren. Zusätzlich haben wir unseren SceneManager als Enum aufgebaut und jede Szene mit Pfad zu ihrer FXML Datei darin gespeichert. Dadurch können die Szenen einfach ausgetauscht und bei Bedarf dem Controller entsprechend übergeben werden.

Vererbung: Die Klasse ExceptionPopup wird von mehreren GUI Controllern geerbt. Somit verwenden alle GUI Komponenten die gleiche Methode zum Anzeigen einer Exception als Pop-up auf der GUI Oberfläche.

Abstrakte Klassen: Die Klasse ExceptionPopup ist außerdem auch eine abstrakte Klasse, da diese durch die Vererbung nur eine Hilfsmethode für die GUI Controller bereitstellt und daher nicht instanziiert werden muss.

7.2. Clean Code

Das Thema Clean Code haben wir in unserem Projekt über eine konkrete Datenkapselung durch passende Access Modifier realisiert.
Wir verwenden aussagekräftige Variablen und Methodennamen und haben an den meisten Stellen JavaDoc Kommentare geschrieben.
Unsere Getter Methoden liefern größtenteils ein Read-Only Objekte, indem wir Kopien von Collections erstellen und diese zurückgeben. Ausnahmen sind hier die Getter für Queue und History, da hier die Collections direkt als Referenzen an GUI Elemente übergeben werden, um die Elemente der TableViews durch Verwendung der JavaFX Observable Lists entsprechend zu aktualisieren.
Statische Methoden werden nur als Getter für die Instanzen des Singleton-Patterns und für das aktuelle Stage Objekt verwendet.

7.3. Tests

In unseren JUnit Tests haben wir die JUnit Jupiter Library verwendet, um unsere 3 Entitäten, die User, Sounds und Playlisten und deren Manager zu testen. Wir haben sowohl negative Tests als auch positive Tests implementiert. Jeder Test kann vollständig unabhängig ablaufen. Alle dadurch erstellten Objekte werden im Anschluss durch eine entsprechende Aufräum-Methode beseitigt. Ebenfalls wird getestet, ob bei den negativen Tests die richtigen Exceptions geworfen werden und ob bei den positiven Tests die Methoden ohne eine solche Exception durchlaufen werden.

Zu beachten ist hierbei das Phänomen beim Löschen von Sound Objekten, welches auch in den Tests auftreten kann und daher die Aufräum-Methode der Sounds ausgeschaltet ist. Die Folge ist, dass bei jedem Testdurchlauf der Resource Ordner unter den Tests mit Sounddateien zugemüllt wird und diese bestenfalls nach Ausführung der Tests manuell gelöscht werden sollte.

7.4. GUI (JavaFX)

Unsere Anwendung besteht aus 2 Stages und 7 verschiedenen Szenen, wovon jede Szene einen eigenen Controller zur Steuerung der GUI Elemente enthält. Der Szenenwechsel wird in Kombination mit den Randbedingungen (Fenstergröße, Resizable,...) für jede Stage im SceneManager gesteuert. Wir arbeiten in der MainScene mit einer komplexen Verschaltung der Nodes, da hier die View Szene in die Mitte der MainScene geladen wird. Zudem verfügt die View über mehrere Tabellen, welche alle durch Verwendung von Factories benutzerdefinierte Inhalte wie die SoundCover oder Profilbilder der Ersteller beinhalten. Außerdem wurde vollständig das MVC Pattern umgesetzt, sodass das Verwenden von mehreren Fenstern sowie die Kommunikation zwischen verschiedenen Szenen über Veränderung am Modell überall entsprechend angezeigt werden kann.

7.5. Logging/Exceptions

In unserer Anwendung haben wir durch Änderungen in der Logger Konfiguration die Ausgabe der Logs ebenfalls in 2 verschiedene Logfiles geschrieben. Diese sind auf das Info- bzw. Debuglevel konfiguriert. Zu finden sind diese Logs unter:

```
.
├──logs
        └──debugLogger.log
        └──infoLogger.log
```

Für das Logging in unserer Anwendung benutzen wir die folgenden 5 States:

- trace: Wurde nur an wenigen Stellen eingesetzt, vor allem um den Überblick über den Programmablauf nicht zu verlieren.
- debug: Wurde hauptsächlich an Stellen eingesetzt, die nicht direkt relevant für den Nutzer sind, dennoch nützlich, um das Verhalten der Anwendung nachzuvollziehen. (Vor Allem für Entwickler)
- info: Wurde als Ausgabe von Nutzer relevanten Informationen genutzt, welche auch für einen Endanwender einfach zu verstehen sind.
- error: Wurde benutzt, um auf einen Fehler in der Anwendung hinzuweisen. Oft im Zusammenhang mit einer Exception.
- fatal: Wurde genutzt, um äußerst Fehlerhaftes Verhalten des Programms zu protokollieren. Die Anwendung wird hierbei meist sofort beendet.

Exceptions werden immer dann geworfen, wenn eine von uns unbeabsichtigte Aktion ausgeführt wird. Hierzu haben wir vermehrt die IllegalArgument oder IllegalStateException verwendet. Zusätzlich haben wir noch die beiden Custom Exceptions: ReadDataException und WriteDataException geschrieben, welche bei fehlerhaften Lesen bzw. Schreiben der JSON Dateien geworfen werden. Nahezu alle für die Benutzereingabe relevanten Exceptions werden entweder als Error Text oder über die ExceptionPopup Klasse direkt im GUI angezeigt.

7.6. UML

Unsere UML Diagramme wurden mit dem Onlinetool draw.io erstellt. Wir haben 2 Use Case Diagramme, da die Verknüpfung des Login mit der Hauptanwendung erst Probleme machte, da der User für nahezu alle Aktionen in der Anwendung erst einmal eingeloggt sein muss. Daher haben wir diese separiert, um die Funktionen der beiden Szenen einzeln aufzuzeigen.
Als Sequenzdiagramm haben wir den Ablauf der Play/Pause Aktion unseres Players abgebildet. Hierbei sind die Bedingungen innerhalb der eckigen Klammern für jeden Methodenaufruf einzeln zu verstehen.
Das Klassendiagramm haben wir aus unserem IntelliJ Projekt in draw.io exportiert. Dieses stellt alle Klassen mit korrekten Sichtbarkeiten der Attribute

und Methoden sowie den Konstruktoren und den Beziehungen der Klassen untereinander dar.
Zu finden sind unsere Diagramme ebenfalls als svg Datei unter: https://gitlab.mi.hdm-stuttgart.de/rs141/whalex-app/-/tree/master/Dokumentation/Diagramme

7.7. Threads

In unserer Anwendung verwenden wir vermehrt die Klasse DataThread, welche von der Klasse Thread erbt und somit wie ein normaler Thread von mehreren Stellen im Modell gestartet wird. Innerhalb der Klasse werden in den verschiedenen Manager Klassen synchronisierte Methoden zum Speichern und Schreiben in die JSON Dateien aufgerufen. Ebenfalls verwenden wir JavaFX kompatible Threads, beispielsweise innerhalb des LoadingControllers zur Anzeige der ProgressBar oder im MainSceneController zum Aktualisieren der Player Anzeige.

7.8. Streams und Lambda-Funktionen

Streams werden in unserer Anwendung für das Durchsuchen bzw. Filtern von den Collections in den User-,Sound- und Playlist Managern genutzt. Da diese Collections im Anwendungsfall riesig werden können, haben wir dafür parallel arbeitende Streams benutzt, welche durch geeignete Terminal Operationen ohne Race Conditions ablaufen können. Meist wird mithilfe eines Streams auch nur überprüft, ob ein Element in der Collection vorhanden ist. Dies wird durch die equals und hashCode Implementierung der Entitäten (User, Sounds, Playlisten) sichergestellt, somit kann es hierbei auch keine Komplikationen bei den parallelen Streams geben.
Überall dort, wo Lambda Funktionen angewandt werden können, werden diese auch verwendet.

8. Bewertungsbogen

Vorname	Nachname	Kürzel	Matrikelnummer	Projekt	Arc.	Clean Code	Doku	Tests	GUI	Logging/Except.	UML	Threads	Streams	Profiling	Summe - Projekt	Kommentar	Projekt-Note
Robin	Schmidt	rs141	44783	Whalex App	3	3	3	3	3	3	3	3	3	3	30,00		1,00
Marvin	Helmer	mh380	44479	Whalex App	3	3	3	3	3	3	3	3	3	3	30,00		1,00