Software Engineering Project 1 Bonus
Team 4

Team Members:

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

Task 1E

User Management

User Transition System

Set of Initial States (X_0) - {Logged Out}
Set of States (X) - {Logged Out, Logged In}
Set of Actions (U) - {Log In, Create User, Log Out}
Set of Observables (Y) - {Logged Out, Logged In}
Display Map (h:X->Y) -

State (X)	Observation (Y)
Logged Out	Logged Out
Logged In	Logged In

Transition Relation (f: X, U->X) -

(State, Action)	State
(Logged Out, Log In)	Logged In
(Logged Out, Create User)	Logged In
(Logged In, Log Out)	Logged Out

Validation Transition System

Set of Initial States (X_0) - {Not Verified}
Set of States (X) - {Not Verified, Verified}
Set of Actions (U) - {Verify}
Set of Observables (Y) - {Not Verified, Verified}
Display Map (h:X->Y) -

State (X)	Observation (Y)
Not Verified	Not Verified
Verified	Verified

Transition Relation (f: X, U->X) -

(State, Action)	State
(Not Verified, Verify)	Verified
(Not Verified, Verify)	Not Verified

There is a function call between the User and Validation transition systems, where the user transition system temporarily suspends its execution until it waits for the validation transition system to execute, upon which the user transition system can continue executing based on the output of the validation system.

Library Management

Playlist Transition System

Set of Initial States (X_0) - {Playlist Does Not Exist}
Set of States (X) - {Playlist Does Not Exist, Playlist Empty, Playlist Not Empty}
Set of Actions (U) - {Create, Delete, Update}
Set of Observables (Y) - {Playlist Not Playable, Playlist Playable}
Display Map (h:X->Y) -

State (X)	Observation (Y)
Playlist Does Not Exist	Playlist Not Playable
Playlist Empty	Playlist Not Playable

Playlist Not Empty	Playlist Playable

Transition Relation (f: X, U->X) -

(State, Action)	State
(Playlist Does Not Exist, Create)	Playlist Empty
(Playlist Empty, Update)	Playlist Not Empty
(Playlist Not Empty, Update)	Playlist Not Empty
(Playlist Empty, Delete)	Playlist Does Not Exist
(Playlist Not Empty, Delete)	Playlist Does Not Exist

Audio Transition System

Set of Initial States (X_0) - {Audio File Does Not Exist}
Set of States (X) - {Audio File Does Not Exist, Audio File Exists}
Set of Actions (U) - {Import, Update}
Set of Observables (Y) - {Audio File Does Not Exist, Audio File Exists}
Display Map (h:X->Y) -

State (X)	Observation (Y)
Audio File Does Not Exist	Audio File Does Not Exist
Audio File Exists	Audio File Exists

Transition Relation (f: X, U->X) -

(State, Action)	State
(Audio File Does Not Exist, Import)	Audio File Exists
(Audio File Exists, Update)	Audio File Exists

Last.fm Integration

Track Transition System

Set of Initial States (X_0) - {{Φ, {x_0, x_1, …, x_n}, ∀ i ϵ [0,n], Z, x_i is a track that is a part of the track stream}}
Set of States (X) - {{Φ, {x_0, x_1, …, x_n}}, {i, {x_0, x_1, …, x_n}}} (Contains currently playing track's index and list of tracks part of track stream)
Set of Actions (U) - {(Play, x_i), (Like, x), (Unlike, x_i)}
Set of Observables (Y) - {Music Not Playing, Music Playing}
Display Map (h:X->Y) -

State (X)	Observation (Y)
{Φ, {x_0, x_1, …, x_n}}	Music Not Playing
{i, {x_0, x_1, …, x_n}}	Music Playing

Transition Relation (f: X, U->X) -

(State, Action)	State
({Φ, {x_0, x_1, …, x_n}}, (Play, x_i))	{i, {x_0, x_1, …, x_n}}
({i, {x_0, x_1, …, x_n}}, (Play, x_j))	{j, {x_0, x_1, …, x_n}}
({Φ, {x_0, x_1, …, x_n}}, (Like, x))	{Φ, {x_0, x_1, …, x_n, x_{n+1}}}
({i, {x_0, x_1, …, x_n}}, (Like, x))	{i, {x_0, x_1, …, x_n, x_{n+1}}}
({Φ, {x_0, x_1, …, x_n}}, (Unlike, x_i))	{Φ, {x_0, x_1, …, x_{i-1}, x_{i+1}, …, x_n}}
({i, {x_0, x_1, …, x_n}}, (Unlike, x_j))	{i, {x_0, x_1, …, x_{j-1}, x_{j+1}, …, x_n}}

Administrator Features

Directory Transition System

Set of Initial States (X_0) - {{i, {x_0, x_1, …, x_n}, ∀ i ϵ [0,n], Z, x_i is a directory}}
Set of States (X) - {{Φ, {x_0, x_1, …, x_n}}, {i, {x_0, x_1, …, x_n}}} (Contains active directory and list of all directories)
Set of Actions (U) - {(Create, x), (Update, x_i), (Delete, x_i)}
Set of Observables (Y) - {No Directories, Directories Exist}
Display Map (h:X->Y) -

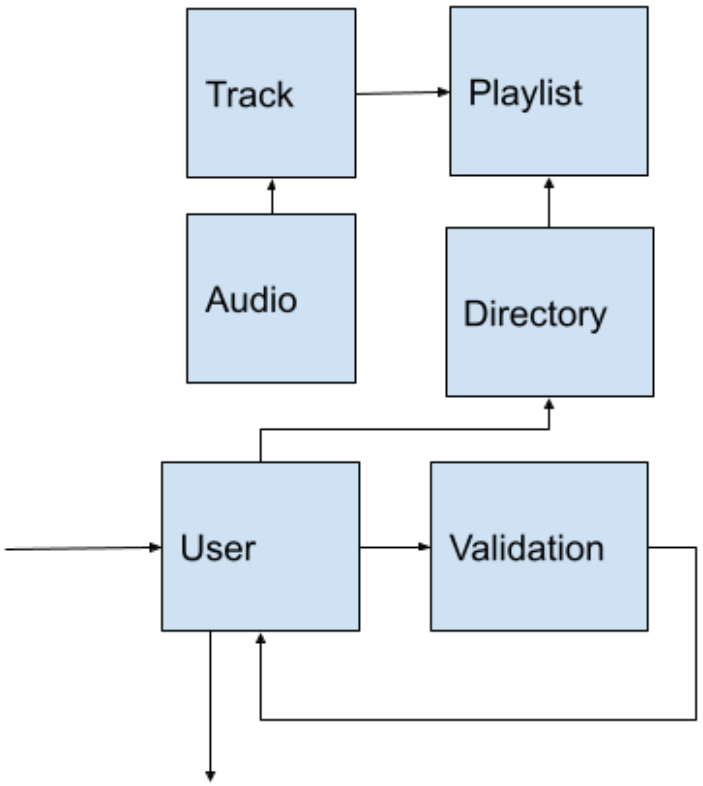
State (X)	Observation (Y)

{Φ, {}}	No Directories
{i, {x_0, x_1, …, x_n}}	Directories Exist

Transition Relation (f: X, U->X) -

(State, Action)	State
({Φ, {x_0, x_1, …, x_n}}, (Create, x))	{Φ, {x_0, x_1, …, x_n, x_{n+1}}}
({i, {x_0, x_1, …, x_n}}, (Create, x))	{i, {x_0, x_1, …, x_n, x_{n+1}}}
({Φ, {x_0, x_1, …, x_n}}, (Update, x_i))	{i, {x_0, x_1, …, x_n}}
({i, {x_0, x_1, …, x_n}}, (Update, x_j))	{j, {x_0, x_1, …, x_n}}
({Φ, {x_0, x_1, …, x_n}}, (Delete, x_i))	{Φ, {x_0, x_1, …, x_{i-1}, x_{i+1}, …, x_n}}
({i, {x_0, x_1, …, x_n}}, (Delete, x_j))	{i, {x_0, x_1, …, x_{j-1}, x_{j+1}, …, x_n}}

Diagram Form of Connections Between Transition Systems:



This is a rudimentary high-level representation of the connections between the various transition systems. I have not represented the multiple validation protocols that each subsystem uses to avoid redundancy and only represented for the main use case, for user login validation. Initially, the user logs in to the system from the starting home page or creates their own user. After this, the information is passed through the validation system which then goes back to the user system and updates the state to 'Logged In'. Now the user can interact with the rest of the system. The user can create directories nad change the active directory. Each directory will have access to playlists and these playlists subsequently contain the music tracks. Tracks can be uploaded using the audio transition system's actions. Playlist allows the user to select and update the currently playing track. Finally, the user can also log out of the application and be taken back to the starting home page.

Task 3E

Automatic refactoring should serve the main purpose of improving the design, structure, readability and maintainability of source code and decreasing its complexity, by identifying and addressing design smells with consideration for code metrics. The tool in question should suggest ways in which the code can be refactored automatically with minimal user intervention.

The following broad steps can be followed:

1. Analysis of code
2. Identification of design smells
3. Evaluation of code metrics
4. Prioritization of design smells
5. Suggesting and executing refactoring

Analysis of code

The first and foremost step which serves as a necessity for any further steps is the analysis of code. The entire source code in itself might make sense to a human reading it carefully, but for a machine, the code must be broken down to its elemental components and organized to be understood. This would encapsulate the structure as well as the flow of the program in a hierarchical manner.

The code should first be broken down into the smallest tokens of the code such as variables, operators and keywords. To establish the structure, these tokens would be built into a syntax tree through parsing of the source code.

We can then go one step further to build an Abstract Syntax Tree (AST). An AST would represent the syntactic structure of the code, and have nodes comprising the various elements of the program such as function calls, variable declarations, etc, in accordance with the rules of whatever programming language is being used.

To understand the flow of the program, we can construct a control flow graph. The graph would represent the flow of control between various statements and branches of the code. This can be done by slicing the program into smaller, relevant parts, to identify which part affects which variable and output.

Identification of design smells

There are many design smells which we must try to identify such as long methods, complex flow of control, redundancies, etc. In order to identify this, our system must incorporate a model which can be rule-based, machine learning based, or a combination of both, out of which the third option would be best suited. On top of certain rule based approaches, the model can be trained on collected data in order to identify design smells with reasonable accuracy.

Evaluation of code metrics

Multiple code metrics such as lines of code, sizes of classes, cyclomatic complexity, cohesion and coupling should be determined by scanning the codebase and applying appropriate calculation.

Prioritization of design smells

Based on the code metrics calculated, we must then determine the priority of each design smell. This is done by finding out which sections of the codebase are in most need of refactoring by looking at the distribution of code metrics. For example, a class with high coupling, low cohesion, high size and high complexity would be an extremely problematic class.

Suggesting and executing refactoring

Now that the design smells have been identified and prioritized, ways of refactoring can be suggested through a combination of rule-based and machine learning based approaches. On top of certain rules, the model can be trained on data which indicates what kind of design smells would require which kind of refactoring. Based on the selection of the user, the tool can then perform the refactoring by methods such as splitting or introducing new classes, redefining code structure, resizing functions, etc.