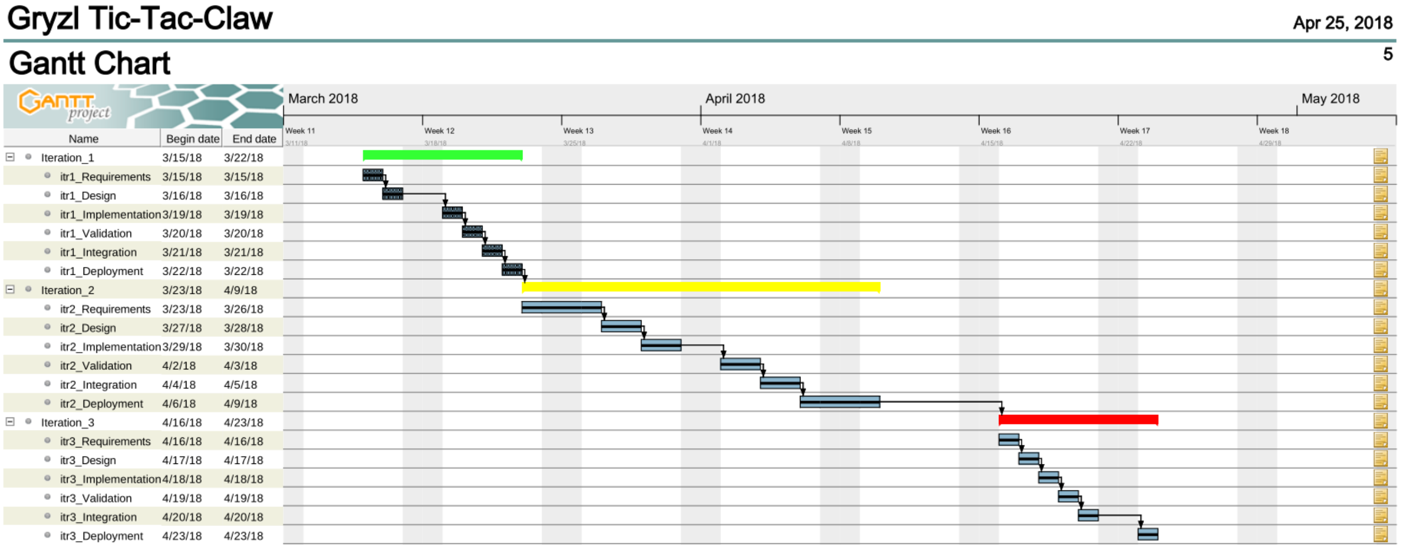
# Tic-Tac-Claw

**Outline:** We created a desktop application (a game), specifically, a rendition of “Ultimate Tic-Tac-Toe” we’ve dubbed “Tic-Tac-Claw” . The game is build using libraries such as JavaFX, Jasypt, and MySQL Connector. We also used Amazon Web Services, specifically Amazon Relational Database Services, to host our remote server for the database we used for user authentication. The project helped to demonstrate our abilities as full-stack developers - we are excited to share this game with the world and hope you enjoy.

**Vision Statement:** Tic-Tac-Claw seems like a simple concept, at heart - which is great, since it means it is very easy to get into - but it’s clear how high a skill cap this game can truly produce. Our twist on the age-old tic-tac-toe will leave players hooked and ready to find the next best strategies to rival their friends and foes alike!

## All Roles:

- Players
- Database Management System
- Game System
- Music Box



Gryzl Tic-Tac-Claw

Apr 25, 2018

Gantt Chart

5

| Name | Begin date | End date |
|---|---|---|
| Iteration_1 | 3/15/18 | 3/22/18 |
| itr1_Requirements | 3/15/18 | 3/15/18 |
| itr1_Design | 3/16/18 | 3/16/18 |
| itr1_Implementation | 3/19/18 | 3/19/18 |
| itr1_Validation | 3/20/18 | 3/20/18 |
| itr1_Integration | 3/21/18 | 3/21/18 |
| itr1_Deployment | 3/22/18 | 3/22/18 |
| Iteration_2 | 3/23/18 | 4/9/18 |
| itr2_Requirements | 3/23/18 | 3/26/18 |
| itr2_Design | 3/27/18 | 3/28/18 |
| itr2_Implementation | 3/29/18 | 3/30/18 |
| itr2_Validation | 4/2/18 | 4/3/18 |
| itr2_Integration | 4/4/18 | 4/5/18 |
| itr2_Deployment | 4/6/18 | 4/9/18 |
| Iteration_3 | 4/16/18 | 4/23/18 |
| itr3_Requirements | 4/16/18 | 4/16/18 |
| itr3_Design | 4/17/18 | 4/17/18 |
| itr3_Implementation | 4/18/18 | 4/18/18 |
| itr3_Validation | 4/19/18 | 4/19/18 |
| itr3_Integration | 4/20/18 | 4/20/18 |
| itr3_Deployment | 4/23/18 | 4/23/18 |

## Gantt Diagram with Timing:

*(See attached pages for detailed report)*

## Requirements:

- User must be able to create an account on our servers
- User must be able to play a local game with another player
- User must be able to play music
- User must be able to mute the music
- User must be able to change the volume of the music

## Business Rules (3 or more):

- To access the game system, a user must be registered on our login servers.
- All users are required to have a unique username; Any attempt to register using a name that has already been claimed will result in an invalid registration.
- Users can only stop the song if it is playing and start the song if it is already off.
- Only one user can make a move at a time.
- There are only two possible outcomes per game: One single winner, or a tie.

## Use Cases (3 or more):

### User Registers an Account

ID: UC User Registration
Scope: Tic-Tac-Claw Registration System
Level: User Goal
Actors: User/Player, Registration System

Precondition: Users has not yet registered to use the application.
Postcondition: User will be able to log in and use our application.
Main Success Scenario:

1) User will have opened the application and be at the main “Login” screen.
2) They will enter their information (Username, Password).
3) They will hit the “Register” button.
4) Their input will be sent from the form to our Registration System.

5) Registration is successful.

6) Their login information will be saved for future logins.

7) User is taken to the main menu.

Extensions:
5.a Registration is unsuccessful.

1) User has input a username that is already in use by another user.

2) User will need to enter a new username.

3) User will hit the registration button again.

4) Continue.

5.a Registration Server is Down

1) User will not get a response from the server.

2) User will exit the application.

3) User will wait a few moments to allow the server to reset.

4) User will log back on and try again.

**User Logs In**

ID: UC User Login
Scope: Tic-Tac-Claw Login System
Level: User Goal
Actors: User/Player, Login System

Precondition: User is registered to use our application.
Postcondition: User will remain logged in for the remainder of the session.
Main Success Scenario:

1) User opens our application

2) User will be prompted with a form to login.

3) User will type information (username and password) into form.

4) Input will be taken by system and passed along to our login server for proper authentication.

5) Login server will report back a login status.

6) Login status is successful.

7) User can now access the main portion of the application.

Extensions:
∗ .a Application is not responding

1) User may restart the application.

5.a Users enters invalid login information

1) User will re-enter the correct login information.

2) Login system will return successful login.

**Players Play a Game**

ID: UC Play game
Scope: Tic-Tac-Claw Game System
Level: User Goal
Actors: Players, Game System

Precondition: At least one user was able to log in.
Postcondition: The game will result in either a win for a single player, or a tie.
Main Success Scenario:

1) Player selects the "Co-op" button on the main menu.

2) Players will find themselves at the Game Board.

3) Each player will take a turn to themselves, starting with player 1.

4) This will continue until there is either a winner, or a tie.

5) Game ends and the winner is greeted by a "Congratulations" screen.

6) Player selects button to return to the main screen.

Extensions:
0. Prior to entering a game, users will click on the "How to Play" button.

1) Users will read the instructions to ensure they know how to play.

2) Users will hit the "Go Back" button to return to the home screen.

4.a User makes an incorrect move.

1) User will be notified by a pop-up window that they made an invalid move.

2) User will hit “Okay” to return to the game board.

### Player Will Change Music Settings

ID: UC Change Settings
Scope: Tic-Tac-Claw Music System
Level: User Goal
Actors: Players, Music Box

Precondition: At least one user was able to log in; music is currently playing.
Postcondition: User will have set the music to their desired level, even completely off.
Main Success Scenario:

1) User has logged in and is at the main menu.

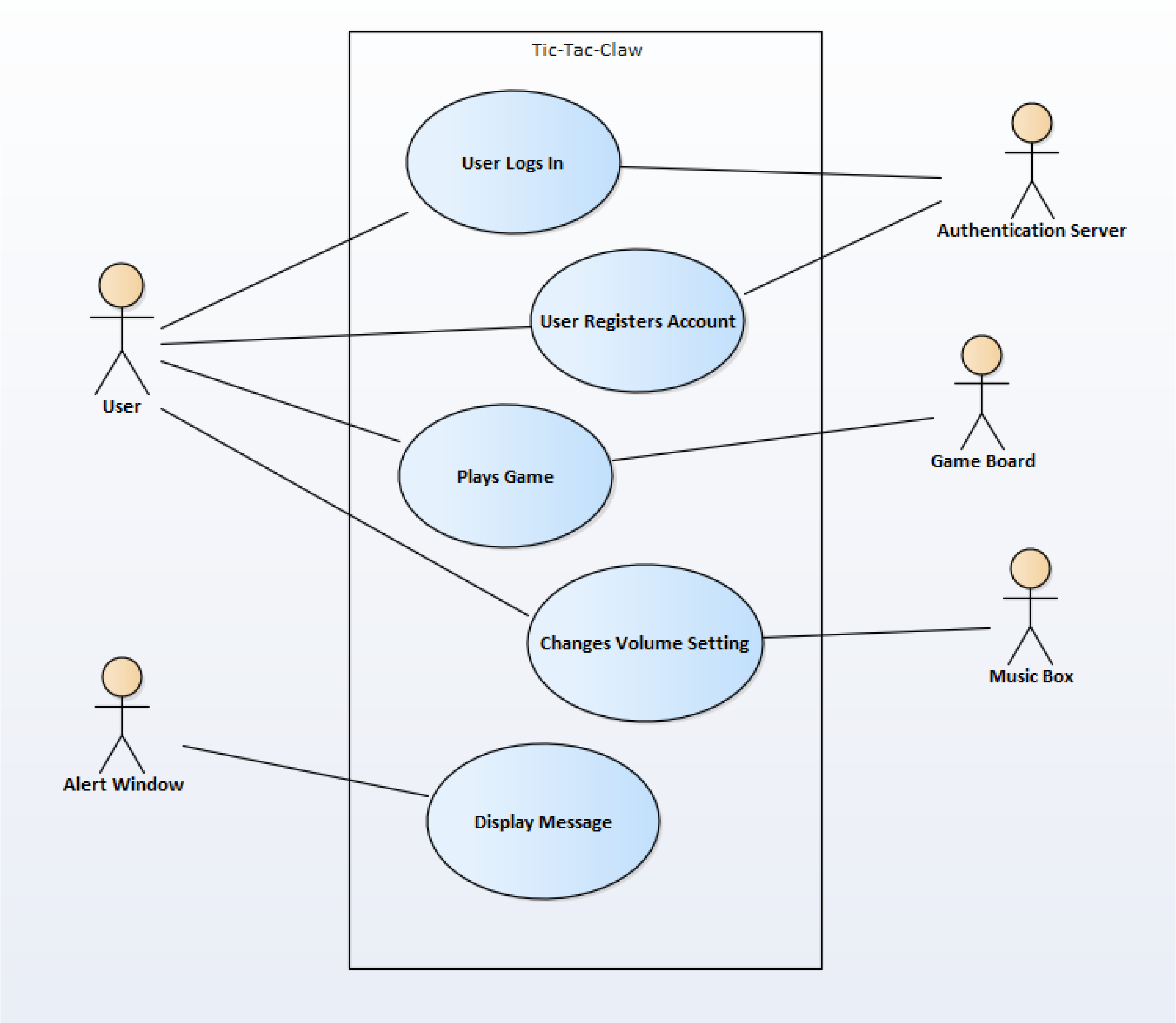
2) User will select the “Settings” button.

3) User is greeted by volume rocker, and “Stop/Start” song option.

4) User will modify settings to their liking.

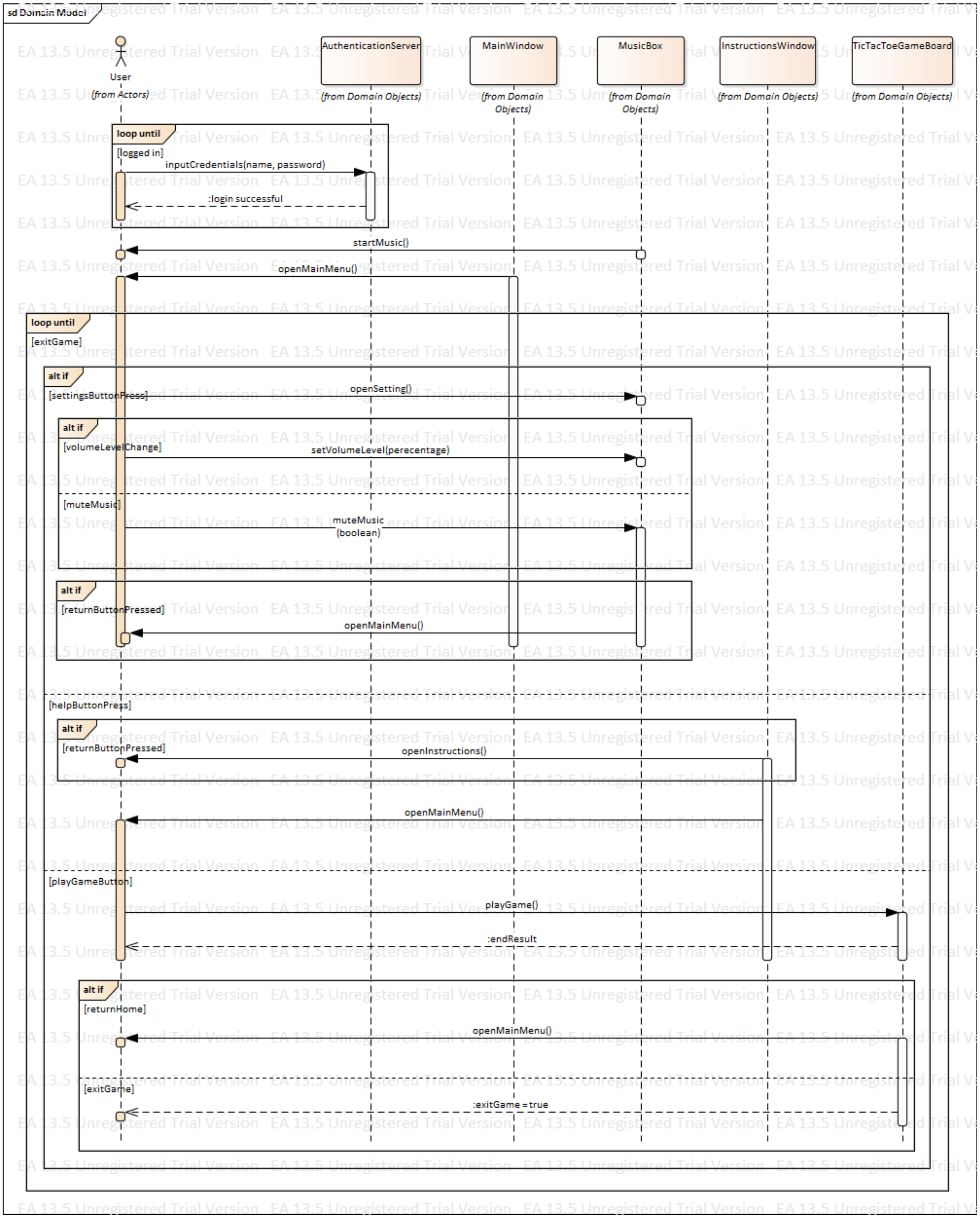
5) User will hit the “Back” button to return to the main menu.

## Use Case Diagrams for All Roles:

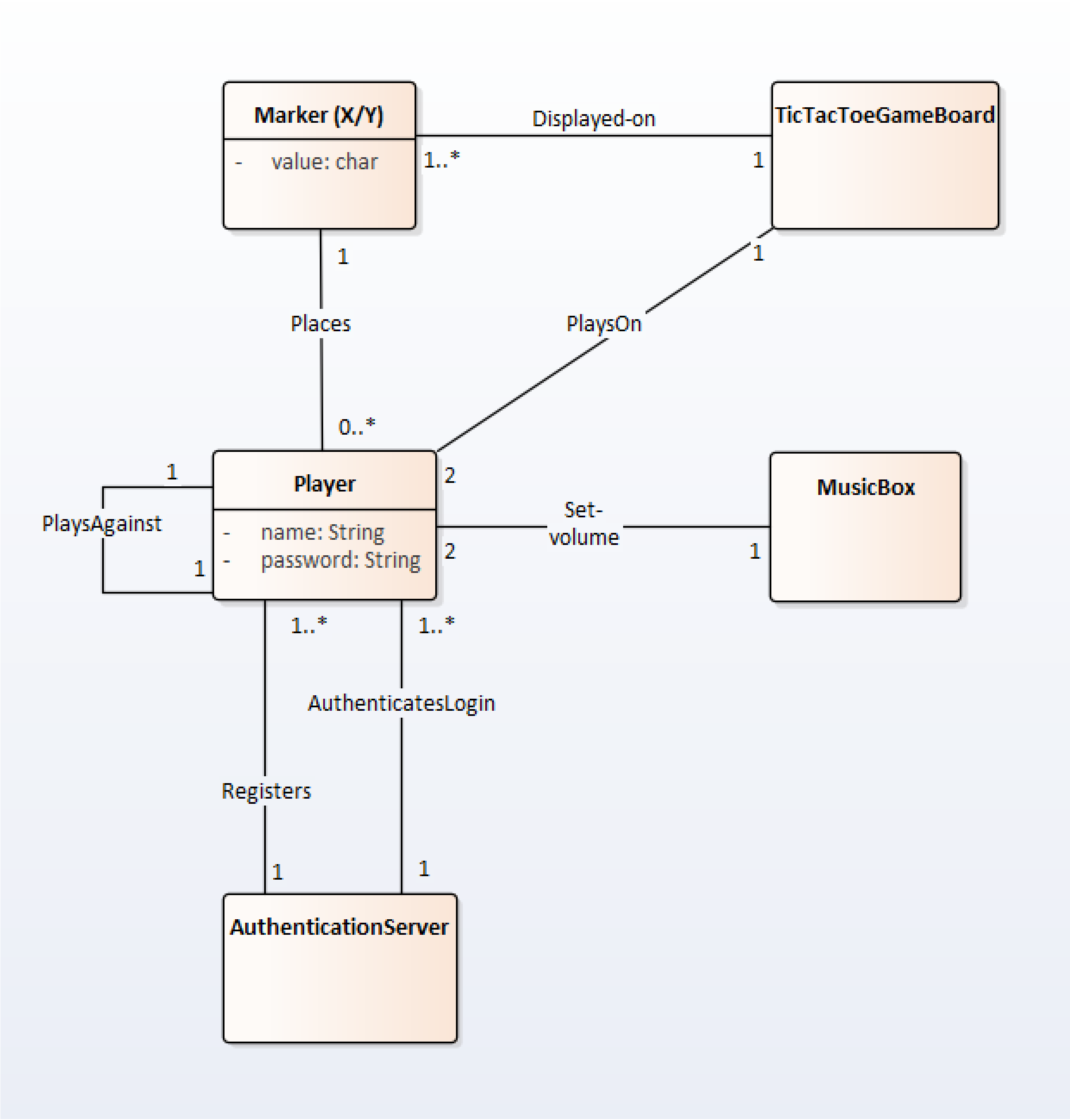


## System Sequence Diagram:

*(Source located in separate file)*



### Domain Model:



## System Operations:

inputCredentials(String, String)
displayError(String)
clickButton(Button)
displayWindow(String)
playCoopGame()
authenticateLogin()
changeVolumeLevel()
muteMusic()
exitGame()

## Operation Contracts:

### Contract C01: inputCredentials

Operation: inputCredentials(String, String)
Cross References: Use Cases: User Login, User Registration
Preconditions: The is at the main login window with the form completed.
Postconditions:

- Authentication System will have received the credentials
- Credentials used when querying database whether for login or registration

### Contract C02: displayError

Operation: displayError(String)
Cross References: Use Cases: User Login, User Registration, Users Play Game
Preconditions: User is at either the login screen, or at the game screen.
Postconditions:

- Window will appear on top of main window.
- Message passed will be displayed inside the window.
- User will have the ability to close the error window and return to where they were.

### Contract C03: playCoopGame

Operation: playCoopGame()
Cross References: Use Cases: Play Local Co-op Game
Preconditions: At least one user has logged in and has just pressed the “Co-op” button on the main menu.
Postconditions:

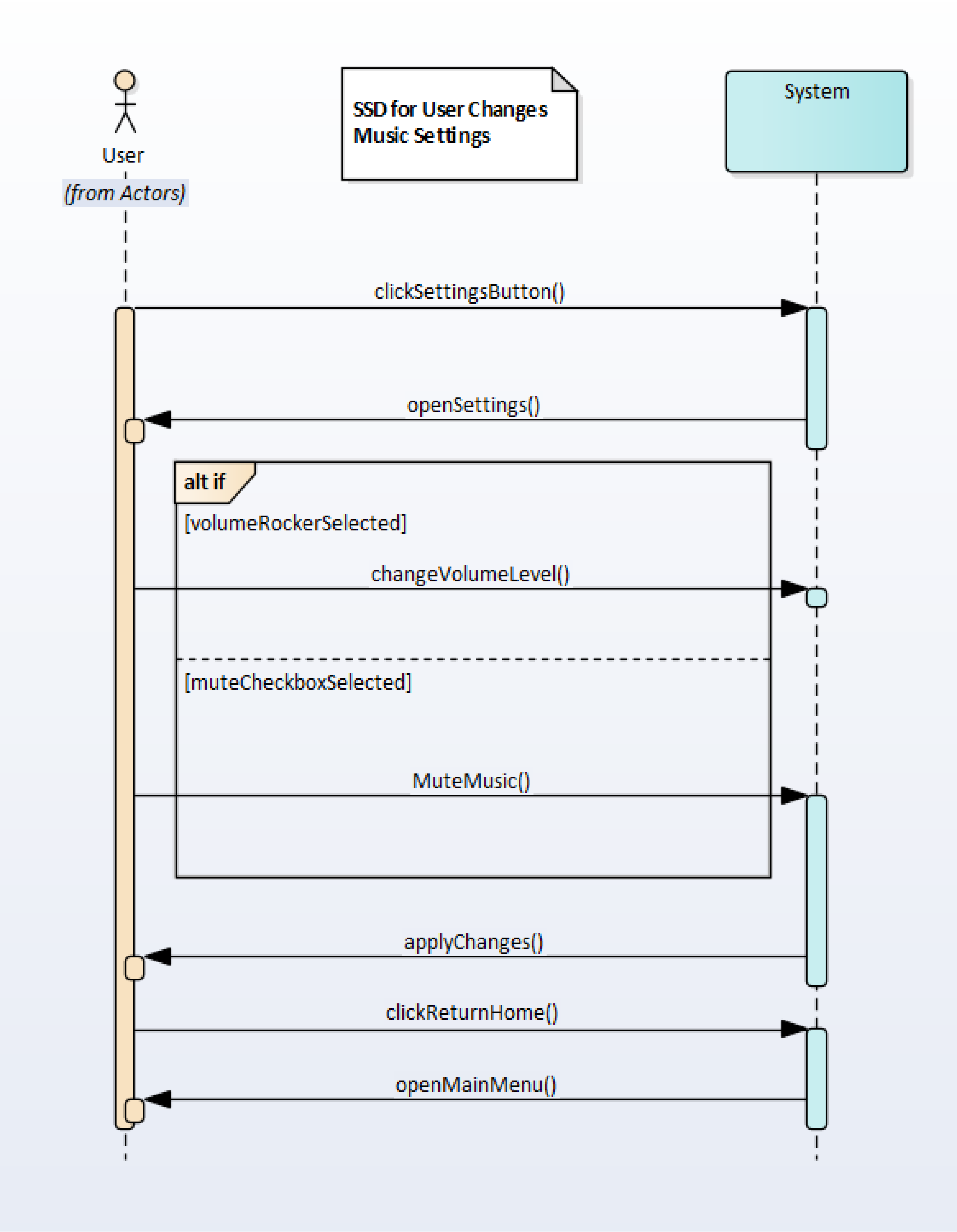
- Either a single player has won, or there was a tie
    - Incase of a win, that player will be greeted with a congratulatory message
    - In case of a tie, both users will be notified
- Users will be sent back to the main menu

### Contract C04: authenticateLogin

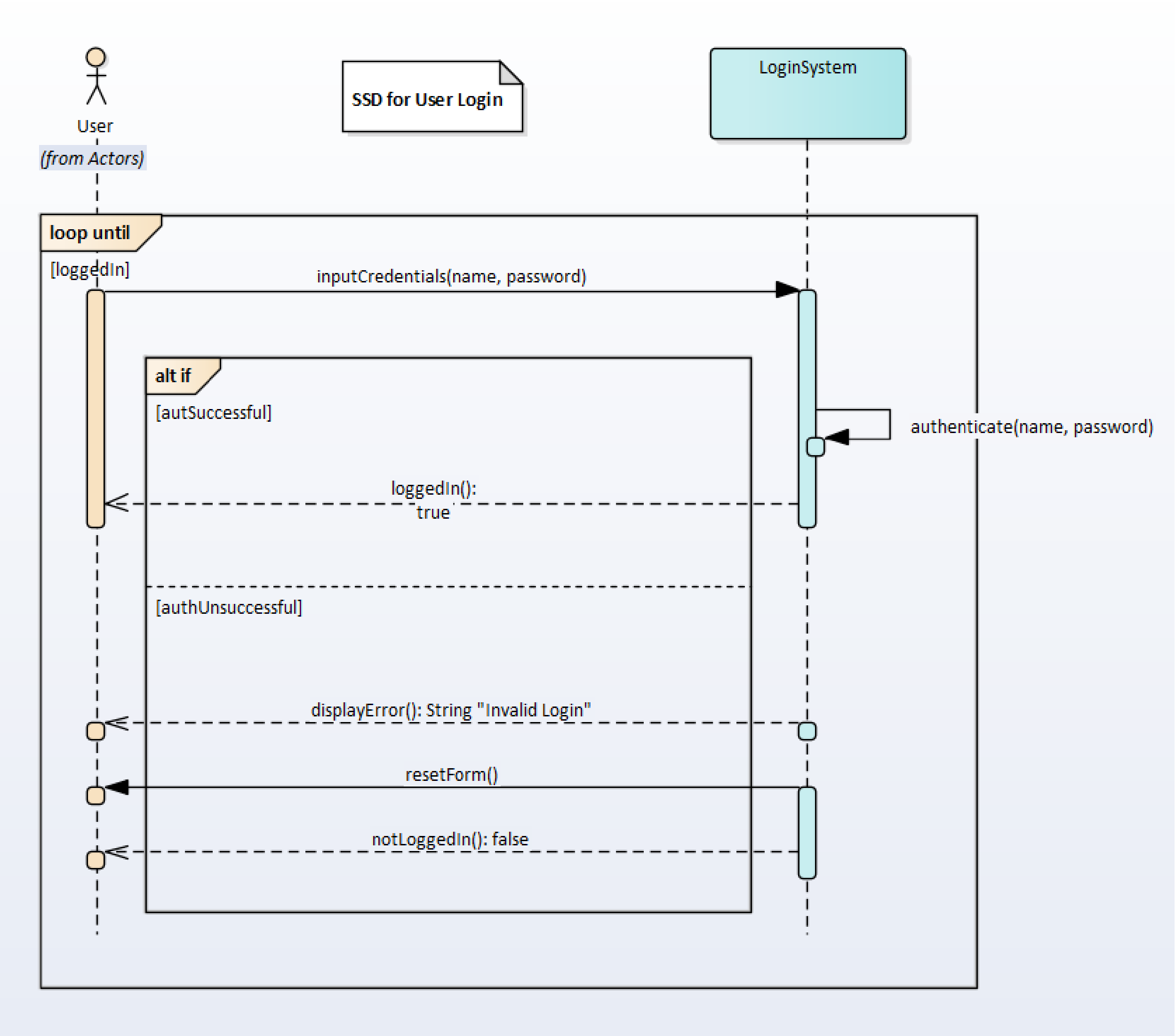
Operation: authenticate(String, String)
Cross References: User Login, User Registration
Preconditions: User has submitted their credentials into the form.
Postconditions:

- User input taken and sanitized.
- Query is made to the database: retrieve record matching name input.
- Record password (encrypted) is extracted.
- Comparing the password input to the encrypted password on the server.
- If it's a match, authentication has resulted in a pass.
- Else, result in a failure.

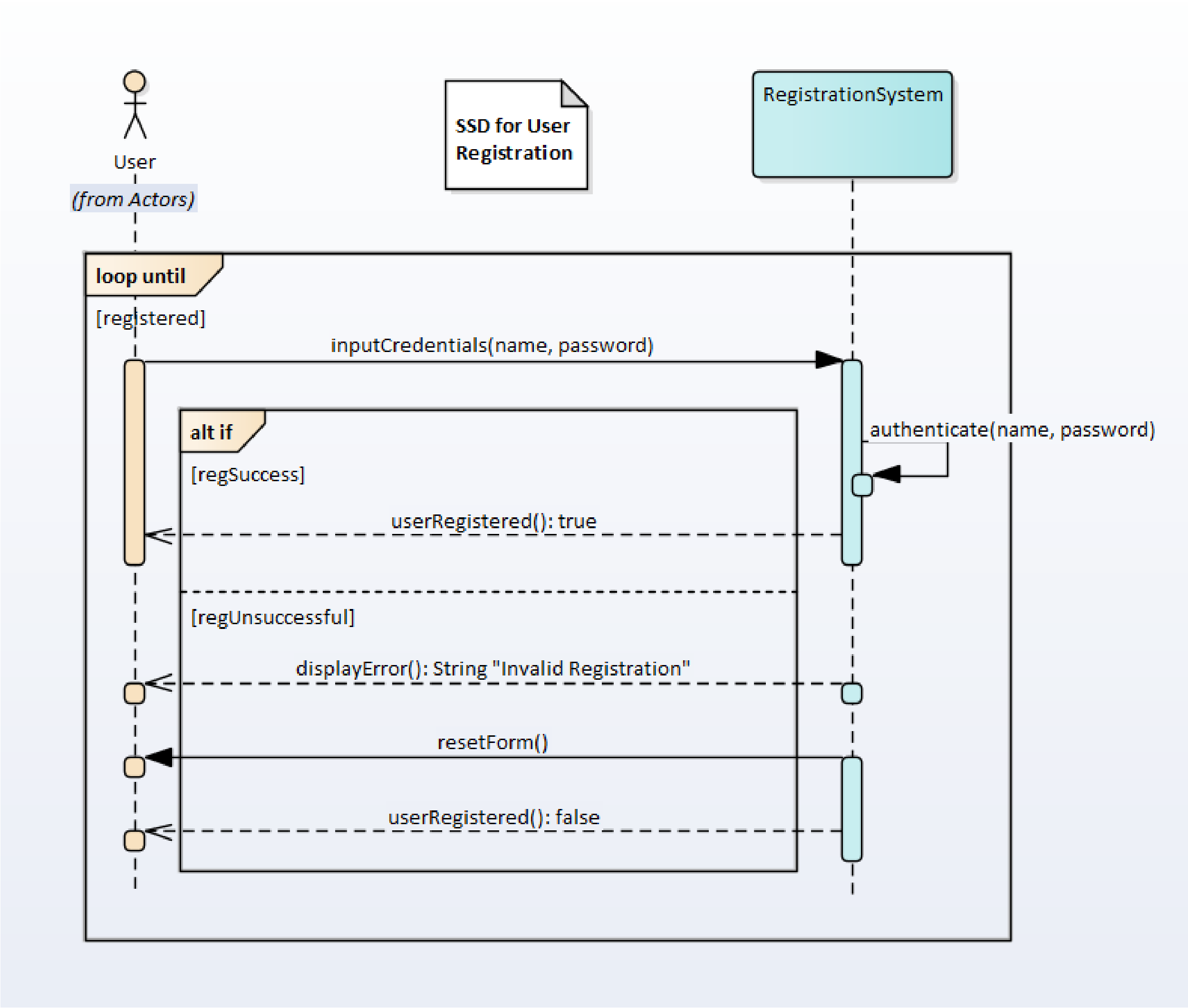
## Sequence Diagrams (for 3 Use Cases):



## Sequence Diagrams (for 3 Use Cases) Continued:



## Sequence Diagrams (for 3 Use Cases) Continued:



## Justification of GRASPs:

1) **Information Expert** - We made use of this GRASP pattern using our SinglePlayer class. Can be seen in the Domain Model that it is interacting with many different aspects of the game. We needed to have this be the main source of logic for the game, so it made sense that all the states and details about the current game being played are tracked within this class.

2) **Creator** - We have Controllers who take the position of creating new windows whenever prompted to, by a button or other action. This is something that can be extremely easy to delegate when using JavaFX since it is almost built that way.

3) **Low Coupling** - We wanted to make sure we broke apart as much of the code as possible - at one point our SinglePlayer class had over 1,200 lines of code. We wanted to make sure we were able to delegate the responsibility out to separate classes, so we could make sure we lowered the coupling on that class in specific. Also making sure we separated a lot of the windows and controllers out when possible to ensure the lowest coupling.

4) **High Cohesion** - We wanted to make sure we were able to get a smooth transition in our code when classes would talk to one another. This was a bit more difficult to do since we had to delegate between a graphical user interface, a database management system, and the logic of the game itself. We did, however, make it work very well. Because of this we were able to find a steady balance between our classes and methods, so our code is much easier to understand, highly reusable, and way easier to maintain that how we had it.

5) **Controller** - This is almost a given. We have many controller classes that can delegate a lot of the responsibilities of the window panes within our GUI. This makes it easy to represent the current state of our overall system/application. We want to make sure that the controller is completely separated from the front-end, so we made it just so.

6) **Polymorphism** - We wanted to take advantage of the awesome capabilities an object-oriented programming language like Java could provide us with, one of the more notable aspects of that being polymorphism. We created an AlertBox class that is decorated by other types of pop-up windows that will fit the context in which they are called. Implementing this design pattern (decorator) would not have been possible without the use of polymorphism.

7) **Pure Fabrication** - Because of the nature of our program being a high cohesion design, we wanted to make sure we were able to reuse various aspects of our code and avoid any sort of unwanted overhead. We were able to do things like creating our own database interface that works with multiple parts of our program together.

8) **Indirection** - This is, again, very intuitive with JavaFX. We know that SinglePlayer also displays this sort of behavior, specifically when displaying its various windows depending on the context of the situation. Handling alerts, match results, etc.

9) **Do Not Talk to Strangers -** This one was very simple to implement using our login system. We didn't want the information to carry its course throughout multiple different classes or anything like that, so we simply take the user input directly to the ConnectToServer class and use it then and there. No need to pass it through unnecessary layers.

## Where did we use a design pattern and why?

1) Command: We implemented this design pattern by capturing the user's mouse click on the ExitBox's exitDisplay() function to determine if the game should close or not. This design pattern allows us to perform an action based on an event that will happen in the future.

2) Singleton: We implemented this design pattern by only having one instance of the MediaBox object and ensuring that it is thread safe. We used this design pattern because we only wanted one instance of MediaBox and if we had multiple instances, there could be multiple MediaBox's playing music at once. As a result, the best way to make sure there was one instance of MediaBox was to make it a Singleton.

3) Decorator: We implemented this design pattern by creating several controllers such as the LoginScreenController, HomeScreenController, and SettingScreenController to add additional behavior to our MasterWindow class. The MasterWindow class is responsible for keeping track of the current Stage and Scene that is being displayed and the controllers use this class to access these attributes and further specialize them. For example, the HomeScreenController will specialize in displaying the HomeScreen and all its buttons.

4) Iterator: We implemented this design pattern by creating an iterator called "WinBoardIterator" that will iterate through the list of tiles that have been won so far. Creating the iterator allowed a simplified version of traversing through the 2D array and reduced redundant code use.

5) Prototype: We used this design pattern with the AlertBox class to create a generic alert popup window that we can then duplicate and add specialization for all our future popup windows. For example, we created ExitBox by adding two buttons to AlertBox.

6) Lazy Initialization: We implemented this design pattern by not instantiating the MediaBox object until the user successfully logged into the game. As a result, we delayed the creation of the object so that the login screen would finish faster and improve overall performance getting into the game.

7) Mediator: We implemented this design pattern in the controllers, such as the LoginScreenController's loginAction() and registerAction(), to encapsulate how an object interacts. These specific functions will be called on an ActionEvent that will determine how the program will change its current running behavior. For example, the loginAction() will verify that it is valid in the database, change the screen to the Home Screen, and begin playing the MediaBox.

## What was covered by tests and how? (Use Cases)

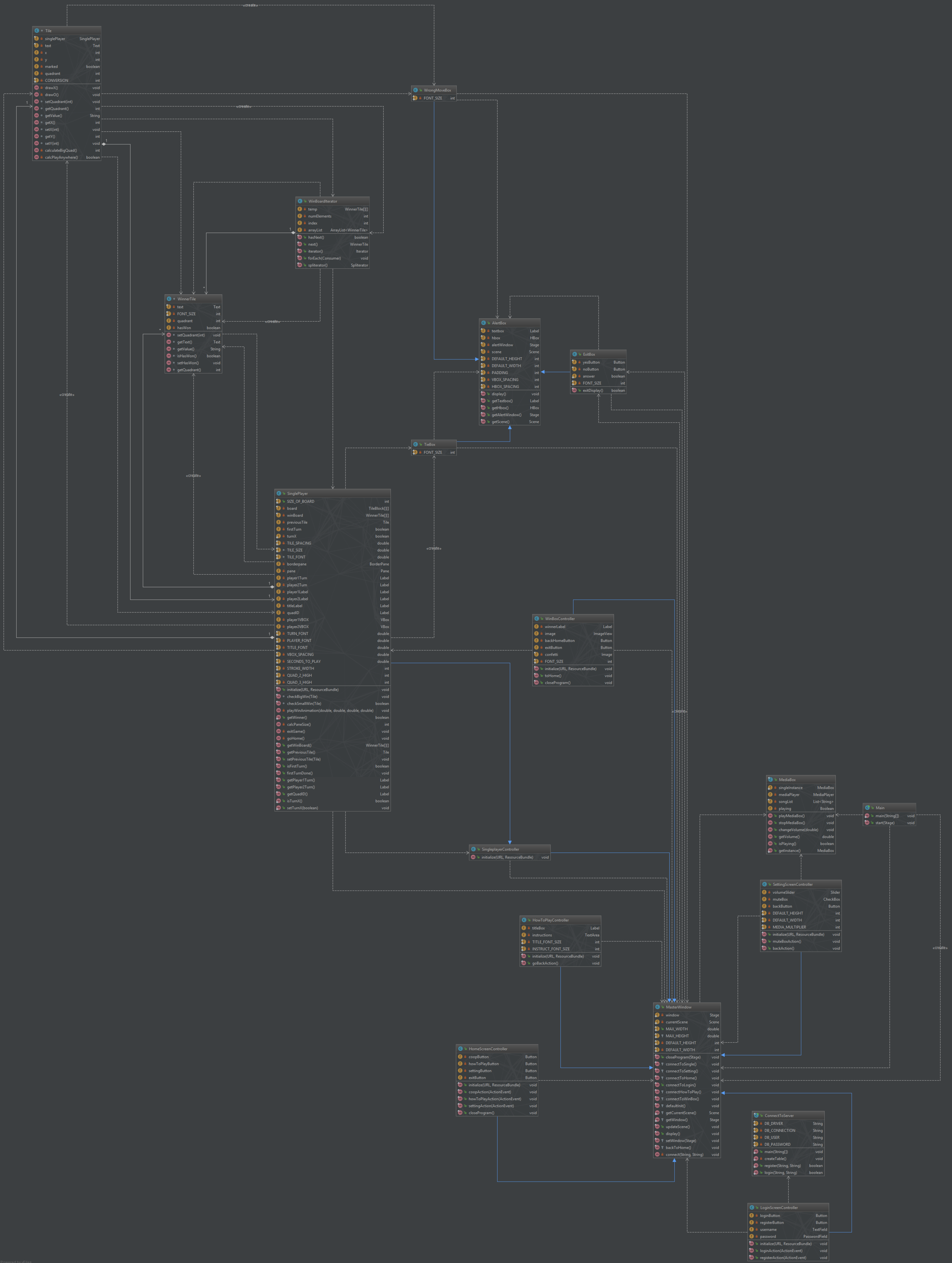
- The MediaBox is tested to verify that it is a singleton by comparing two instances of MediaBox to verify that it is the same instance.
- The MediaBox is tested to verify that the MediaPlayer is playing by running the program, play the MediaBox, and testing to see if it is playing.

## Logical Lines Report:

```
github.com/AlDanial/cloc v 1.76 T=1.00 s (36.0 files/s, 3779.0 lines/s)
-------------------------------------------------------------------------
Language          files       blank       comment     code
-------------------------------------------------------------------------
HTML              10          100         10          1681
Java              22          450         582         1468
XML               9           61          75          738
CSS               8           15          1           484
Maven             1           6           3           152
-------------------------------------------------------------------------
SUM:              50          632         671         4523
-------------------------------------------------------------------------
```

## Approximate Hours Spent:

Brandon Mork: 148 Hours
Mario Lopez: 107 Hours


«create»
Tile
singlePlayer SinglePlayer
text Text
x int
y int
marked boolean
quadrant int
CONVERSION int
drawX() void
drawO() void
setQuadrant(int) void
getQuadrant() int
getValue() String
getX() int
setX(int) void
getY() int
setY(int) void
calculateBigQuad() int
calcPlayAnywhere() boolean
«create»
1
1
WrongMoveBox
FONT_SIZE int
WinBoardIterator
temp WinnerTile[][]
numElements int
index int
arrayList ArrayList<WinnerTile>
hasNext() boolean
next() WinnerTile
iterator() Iterator
forEach(Consumer) void
spliterator() Spliterator
1
*
WinnerTile
text Text
FONT_SIZE int
quadrant int
hasWon boolean
setQuadrant(int) void
getText() Text
getValue() String
isHasWon() boolean
setHasWon() void
getQuadrant() int
«create»
*
«create»
AlertBox
textbox Label
hbox HBox
alertWindow Stage
scene Scene
DEFAULT_HEIGHT int
DEFAULT_WIDTH int
PADDING int
VBOX_SPACING int
HBOX_SPACING int
display() void
getTextbox() Label
getHbox() HBox
getAlertWindow() Stage
getScene() Scene
ExitBox
yesButton Button
noButton Button
answer boolean
FONT_SIZE int
exitDisplay() boolean
«create»
TieBox
FONT_SIZE int
«create»
SinglePlayer
SIZE_OF_BOARD int
board TileBlock[][]
winBoard WinnerTile[][]
previousTile Tile
firstTurn boolean
turnX boolean
TILE_SPACING double
TILE_SIZE double
TILE_FONT double
borderpane BorderPane
pane Pane
player1Turn Label
player2Turn Label
player1Label Label
player2Label Label
titleLabel Label
quadID Label
player1VBOX VBox
player2VBOX VBox
TURN_FONT double
PLAYER_FONT double
TITLE_FONT double
VBOX_SPACING double
SECONDS_TO_PLAY double
STROKE_WIDTH int
QUAD_2_HIGH int
QUAD_3_HIGH int
initialize(URL, ResourceBundle) void
checkBigWin(Tile) void
checkSmallWin(Tile) boolean
playWinAnimation(double, double, double, double) void
getWinner() boolean
calcPaneSize() int
exitGame() void
goHome() void
getWinBoard() WinnerTile[][]
getPreviousTile() Tile
setPreviousTile(Tile) void
isFirstTurn() boolean
firstTurnDone() void
getPlayer1Turn() Label
getPlayer2Turn() Label
getQuadID() Label
isTurnX() boolean
setTurnX(boolean) void
1
1
1
1
WinBoxController
winnerLabel Label
image ImageView
backHomeButton Button
exitButton Button
confetti Image
FONT_SIZE int
initialize(URL, ResourceBundle) void
toHome() void
closeProgram() void
«create»
MediaBox
singleInstance MediaBox
mediaPlayer MediaPlayer
songList List<String>
playing Boolean
playMediaBox() void
stopMediaBox() void
changeVolume(double) void
getVolume() double
isPlaying() boolean
getInstance() MediaBox
Main
main(String[]) void
start(Stage) void
SingleplayerController
initialize(URL, ResourceBundle) void
SettingScreenController
volumeSlider Slider
muteBox CheckBox
backButton Button
DEFAULT_HEIGHT int
DEFAULT_WIDTH int
MEDIA_MULTIPLIER int
initialize(URL, ResourceBundle) void
muteBoxAction() void
backAction() void
HowToPlayController
titleBox Label
instructions TextArea
TITLE_FONT_SIZE int
INSTRUCT_FONT_SIZE int
initialize(URL, ResourceBundle) void
goBackAction() void
«create»
MasterWindow
window Stage
currentScene Scene
MAX_WIDTH double
MAX_HEIGHT double
DEFAULT_HEIGHT int
DEFAULT_WIDTH int
closeProgram(Stage) void
connectToSingle() void
connectToSetting() void
connectToHome() void
connectToLogin() void
connectHowToPlay() void
connectToWinBox() void
defaultInit() void
getCurrentScene() Scene
getWindow() Stage
updateScene() void
display() void
setWindow(Stage) void
backToHome() void
connect(String, String) void
HomeScreenController
coopButton Button
howToPlayButton Button
settingButton Button
exitButton Button
initialize(URL, ResourceBundle) void
coopAction(ActionEvent) void
howToPlayAction(ActionEvent) void
settingAction(ActionEvent) void
closeProgram() void
ConnectToServer
DB_DRIVER String
DB_CONNECTION String
DB_USER String
DB_PASSWORD String
main(String[]) void
createTable() void
register(String, String) boolean
login(String, String) boolean
LoginScreenController
loginButton Button
registerButton Button
username TextField
password PasswordField
initialize(URL, ResourceBundle) void
loginAction(ActionEvent) void
registerAction(ActionEvent) void