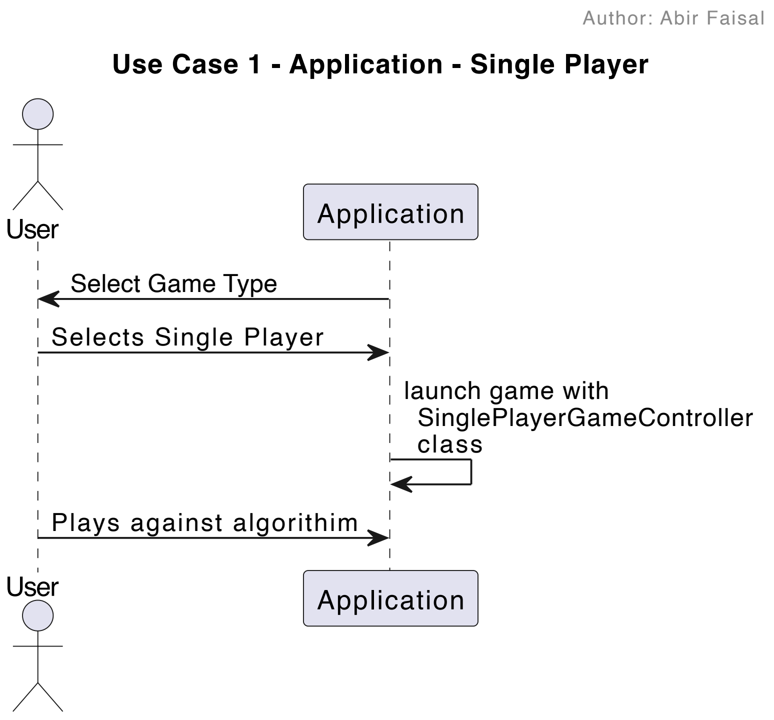
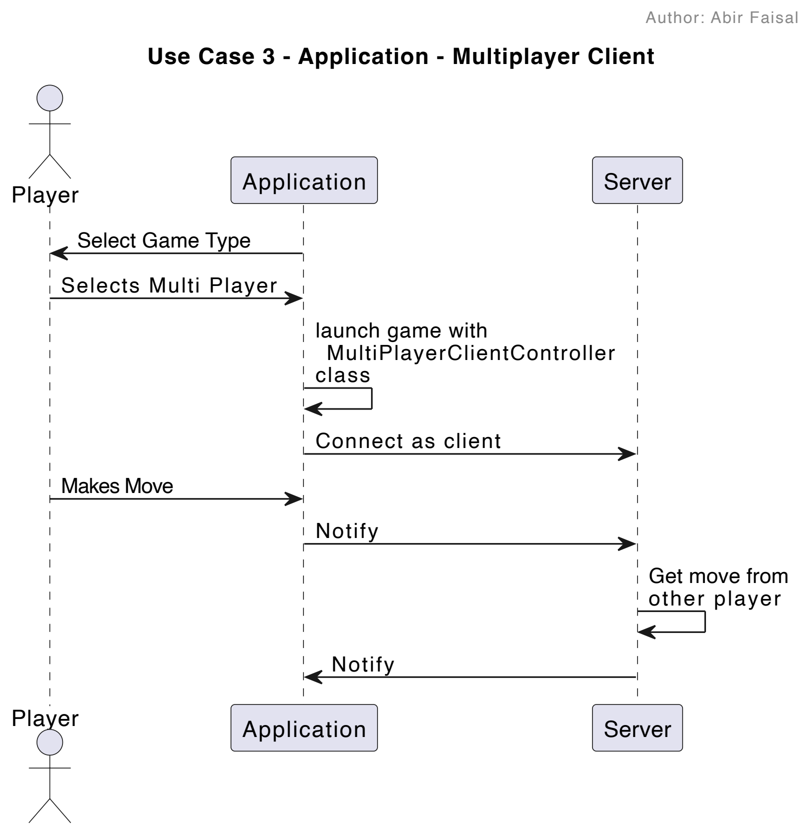
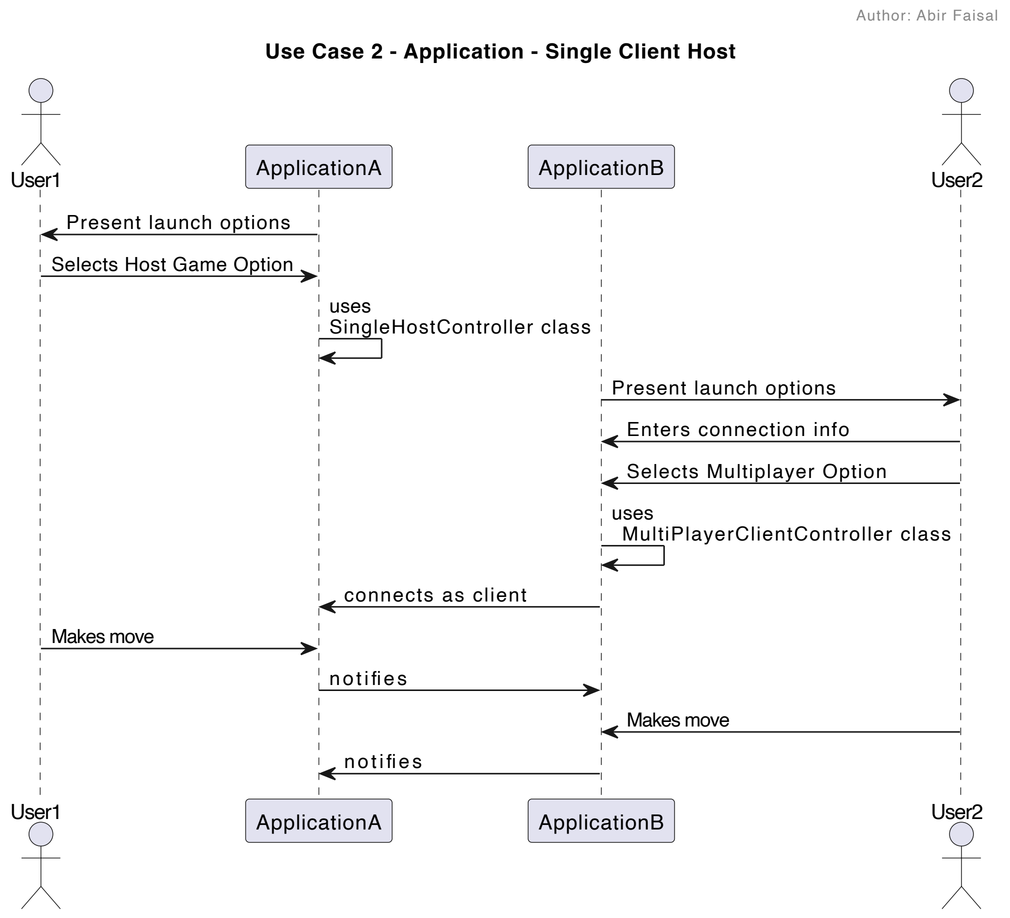
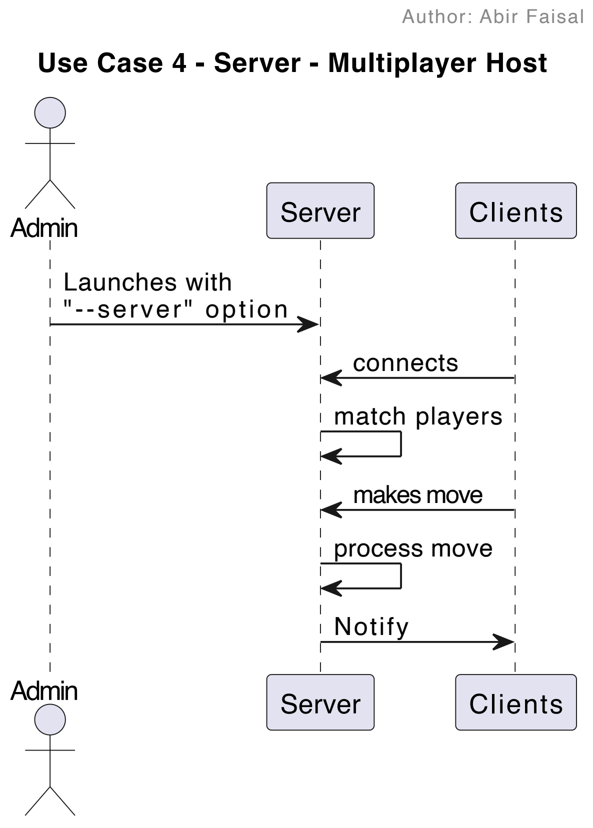
Sequence Diagrams for scenarios

Description: Sequence diagrams for our use case scenarios



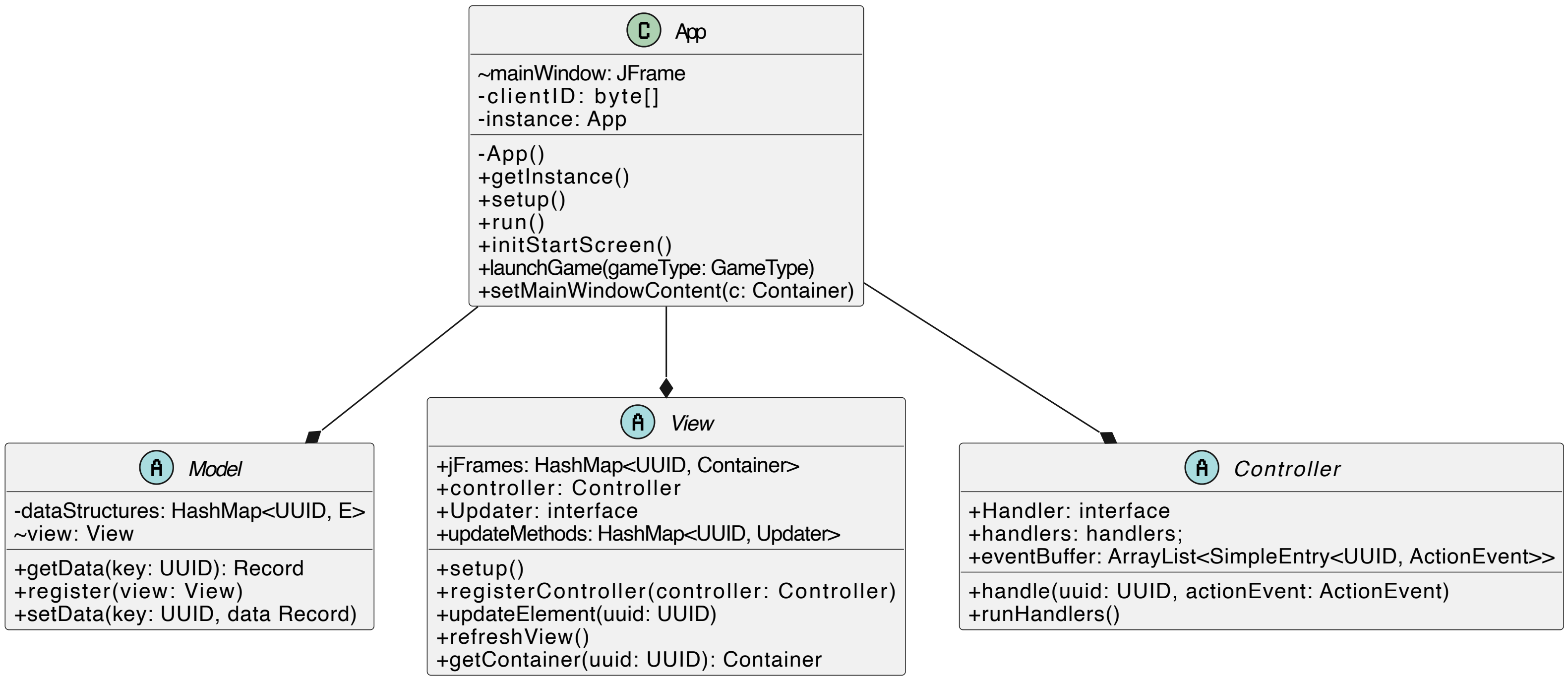






UML Diagrams

Author: Abir Faisal

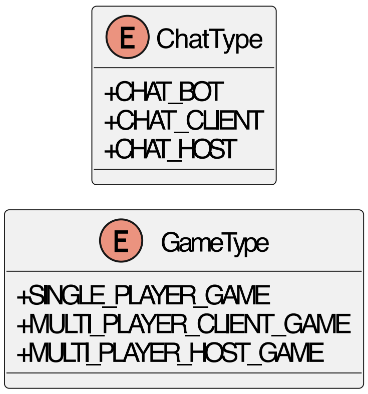

App
~mainWindow: JFrame
-clientID: byte[]
-instance: App
-App()
+getInstance()
+setup()
+run()
+initStartScreen()
+launchGame(gameType: GameType)
+setMainWindowContent(c: Container)
Model
-dataStructures: HashMap<UUID, E>
~view: View
+getData(key: UUID): Record
+register(view: View)
+setData(key: UUID, data Record)
View
+jFrames: HashMap<UUID, Container>
+controller: Controller
+Updater: interface
+updateMethods: HashMap<UUID, Updater>
+setup()
+registerController(controller: Controller)
+updateElement(uuid: UUID)
+refreshView()
+getContainer(uuid: UUID): Container
Controller
+Handler: interface
+handlers: handlers;
+eventBuffer: ArrayList<SimpleEntry<UUID, ActionEvent>>
+handle(uuid: UUID, actionEvent: ActionEvent)
+runHandlers()

Description: The App class is implimented as a singleton pattern

It declares a Model, View, and Controller.

When the application is launched it initializes an empty window and puts whatever View type the programmer specifies into the mainWindow JFrame

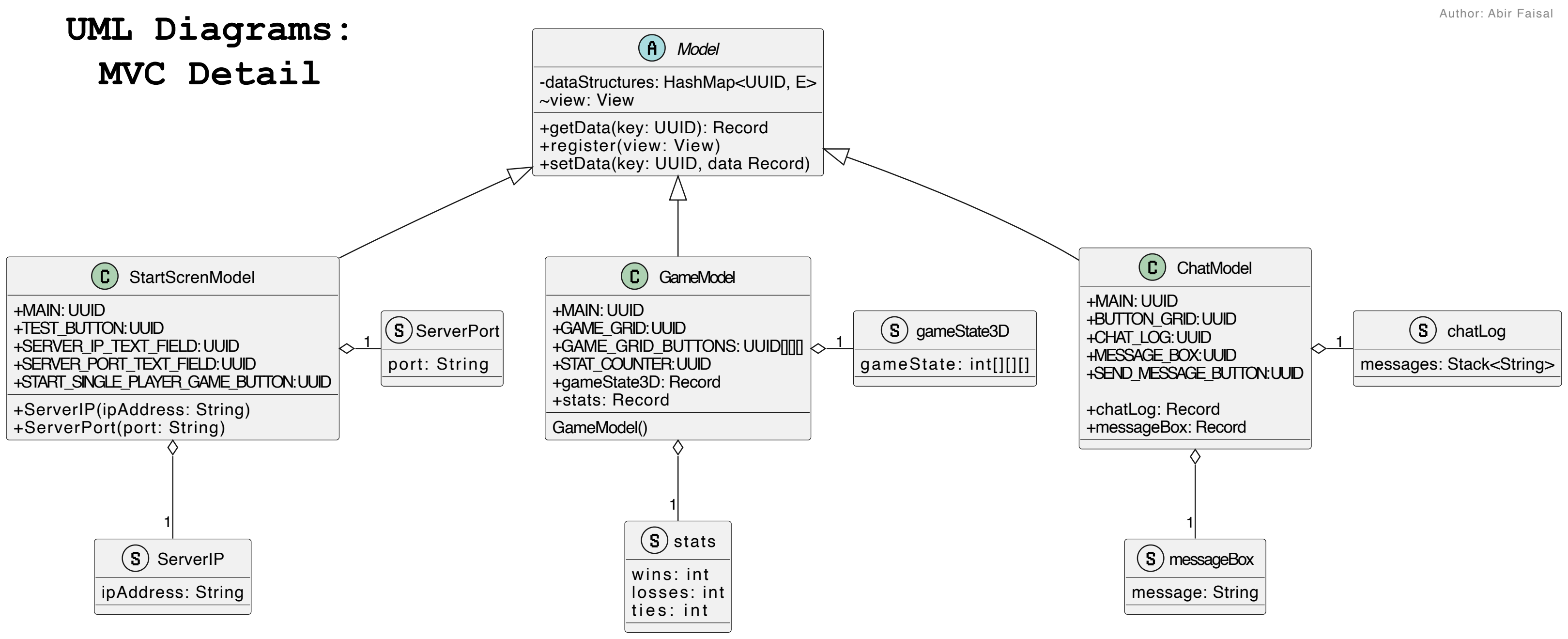
More detail on next page



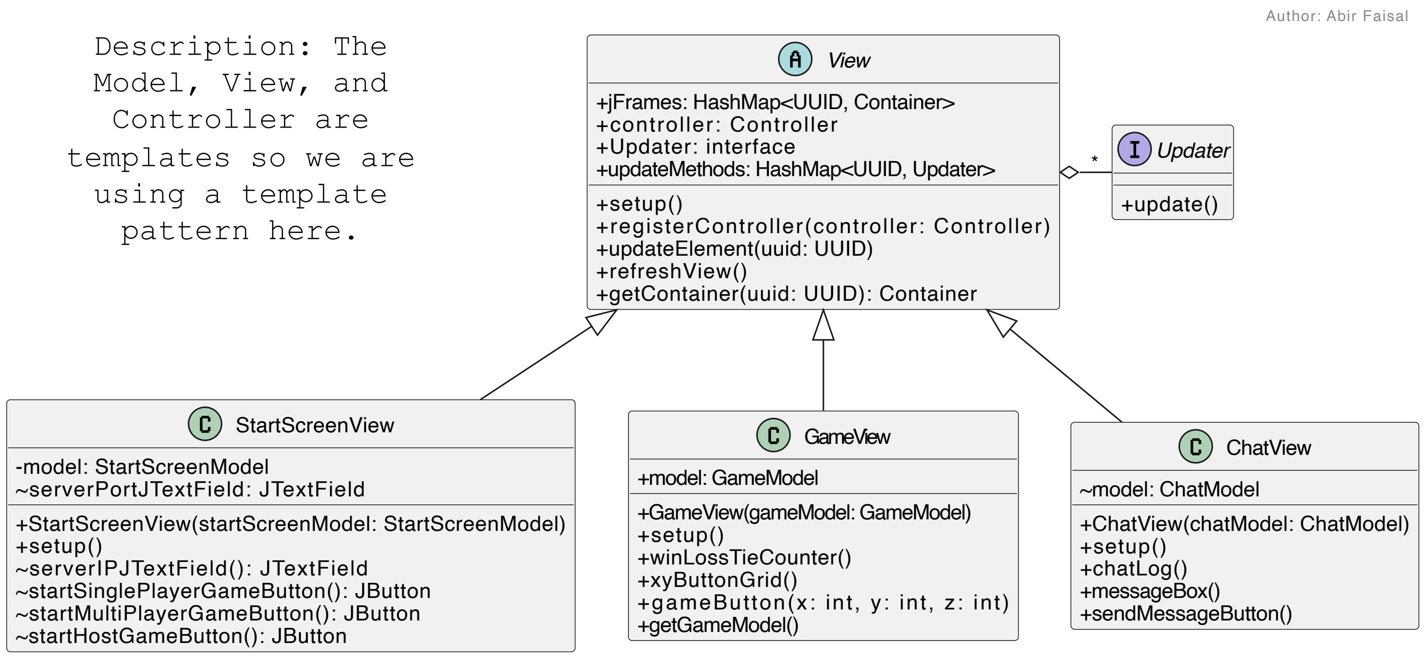
Description: These Enums are used by the launch controller to tell the App what type of Game and Chat to launch.

This is a strategy pattern.

UML Diagrams: MVC Detail



Description: The Model, View, and Controller are templates so we are using a template pattern here.

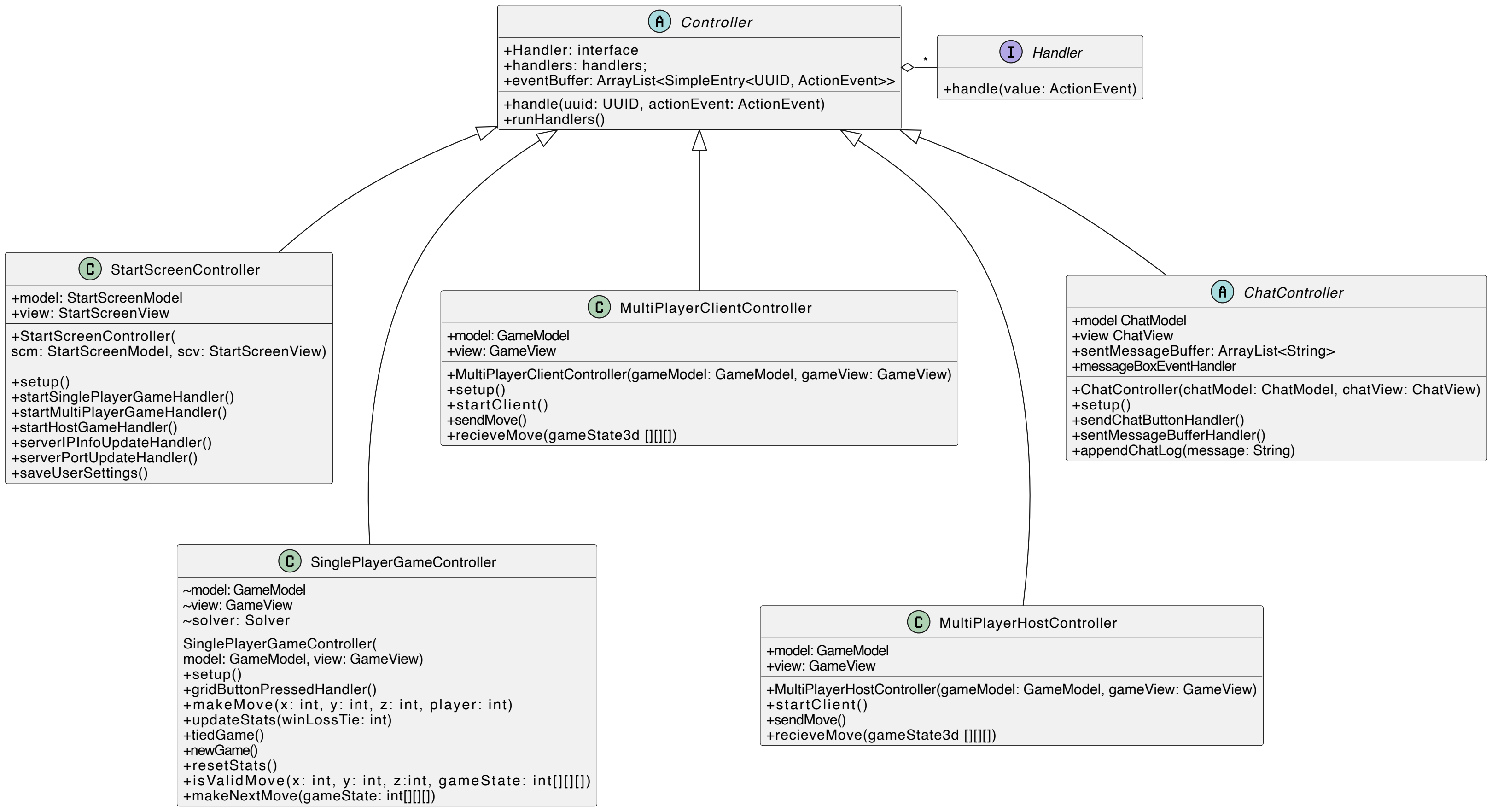

Author: Abir Faisal
View
+jFrames: HashMap<UUID, Container>
+controller: Controller
+Updater: interface
+updateMethods: HashMap<UUID, Updater>
+setup()
+registerController(controller: Controller)
+updateElement(uuid: UUID)
+refreshView()
+getContainer(uuid: UUID): Container
Updater
+update()
*
StartScreenView
-model: StartScreenModel
~serverPortJTextField: JTextField
+StartScreenView(startScreenModel: StartScreenModel)
+setup()
~serverIPJTextField(): JTextField
~startSinglePlayerGameButton(): JButton
~startMultiPlayerGameButton(): JButton
~startHostGameButton(): JButton
GameView
+model: GameModel
+GameView(gameModel: GameModel)
+setup()
+winLossTieCounter()
+xyButtonGrid()
+gameButton(x: int, y: int, z: int)
+getGameModel()
ChatView
~model: ChatModel
+ChatView(chatModel: ChatModel)
+setup()
+chatLog()
+messageBox()
+sendMessageButton()

Controller Detail on next page

UML Diagrams: MVC Detail

Description: The Model, View, and Controller details continued

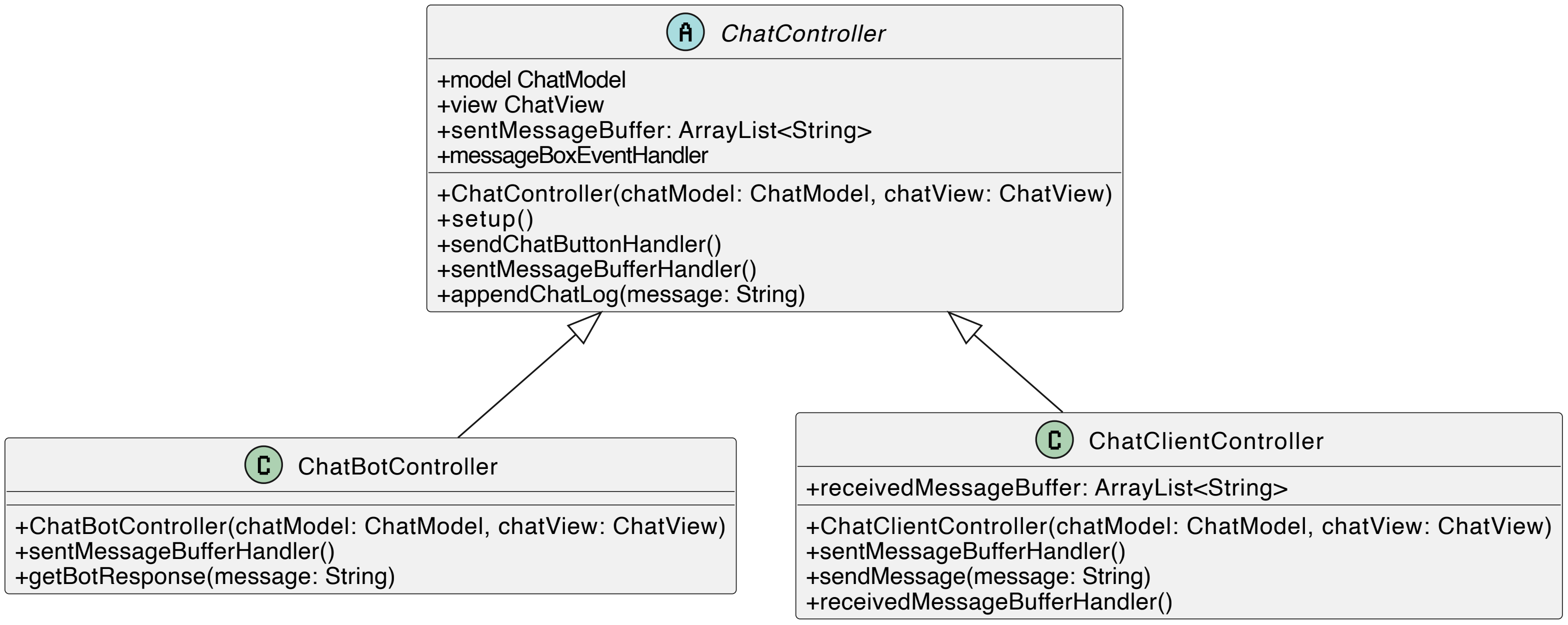
Author: Abir Faisal


Controller
+Handler: interface
+handlers: handlers;
+eventBuffer: ArrayList<SimpleEntry<UUID, ActionEvent>>
+handle(uuid: UUID, actionEvent: ActionEvent)
+runHandlers()
*
Handler
+handle(value: ActionEvent)
StartScreenController
+model: StartScreenModel
+view: StartScreenView
+StartScreenController(
scm: StartScreenModel, scv: StartScreenView)
+setup()
+startSinglePlayerGameHandler()
+startMultiPlayerGameHandler()
+startHostGameHandler()
+serverIPInfoUpdateHandler()
+serverPortUpdateHandler()
+saveUserSettings()
MultiPlayerClientController
+model: GameModel
+view: GameView
+MultiPlayerClientController(gameModel: GameModel, gameView: GameView)
+setup()
+startClient()
+sendMove()
+recieveMove(gameState3d [][][])
ChatController
+model ChatModel
+view ChatView
+sentMessageBuffer: ArrayList<String>
+messageBoxEventHandler
+ChatController(chatModel: ChatModel, chatView: ChatView)
+setup()
+sendChatButtonHandler()
+sentMessageBufferHandler()
+appendChatLog(message: String)
SinglePlayerGameController
~model: GameModel
~view: GameView
~solver: Solver
SinglePlayerGameController(
model: GameModel, view: GameView)
+setup()
+gridButtonPressedHandler()
+makeMove(x: int, y: int, z: int, player: int)
+updateStats(winLossTie: int)
+tiedGame()
+newGame()
+resetStats()
+isValidMove(x: int, y: int, z:int, gameState: int[][][])
+makeNextMove(gameState: int[][][])
MultiPlayerHostController
+model: GameModel
+view: GameView
+MultiPlayerHostController(gameModel: GameModel, gameView: GameView)
+startClient()
+sendMove()
+recieveMove(gameState3d [][][])

UML Diagrams: MVC Detail
Chat Feature

Description: The chat feature was not a part of our original design, but we thought it would show how our MVC architecture is extensible, you can just extend the MVC classes and make your own thing. It doesn't have to be a game it can be anything.

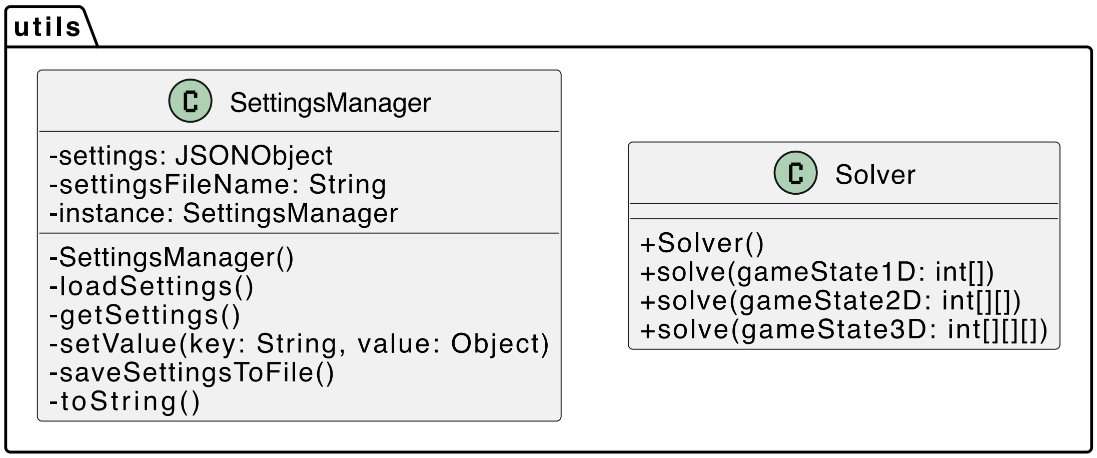
Author: Abir Faisal



A ChatController
+model ChatModel
+view ChatView
+sentMessageBuffer: ArrayList<String>
+messageBoxEventHandler
+ChatController(chatModel: ChatModel, chatView: ChatView)
+setup()
+sendChatButtonHandler()
+sentMessageBufferHandler()
+appendChatLog(message: String)
C ChatBotController
+ChatBotController(chatModel: ChatModel, chatView: ChatView)
+sentMessageBufferHandler()
+getBotResponse(message: String)
C ChatClientController
+receivedMessageBuffer: ArrayList<String>
+ChatClientController(chatModel: ChatModel, chatView: ChatView)
+sentMessageBufferHandler()
+sendMessage(message: String)
+receivedMessageBufferHandler()

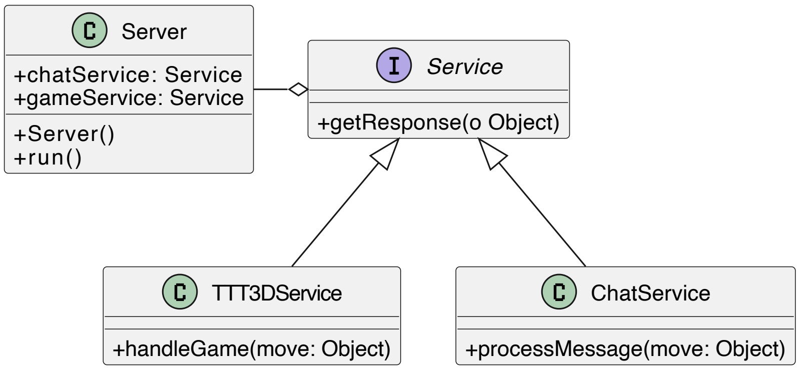
UML Diagrams: Other

Authors: Abir Faisal



utils
C SettingsManager
-settings: JSONObject
-settingsFileName: String
-instance: SettingsManager
-SettingsManager()
-loadSettings()
-getSettings()
-setValue(key: String, value: Object)
-saveSettingsToFile()
-toString()
C Solver
+Solver()
+solve(gameState1D: int[])
+solve(gameState2D: int[][])
+solve(gameState3D: int[][][])

Description: These are utilities that various parts of the program can use as needed.
The solver can check the game for a winner, and the SettingsManager loads and saves program settings from file.

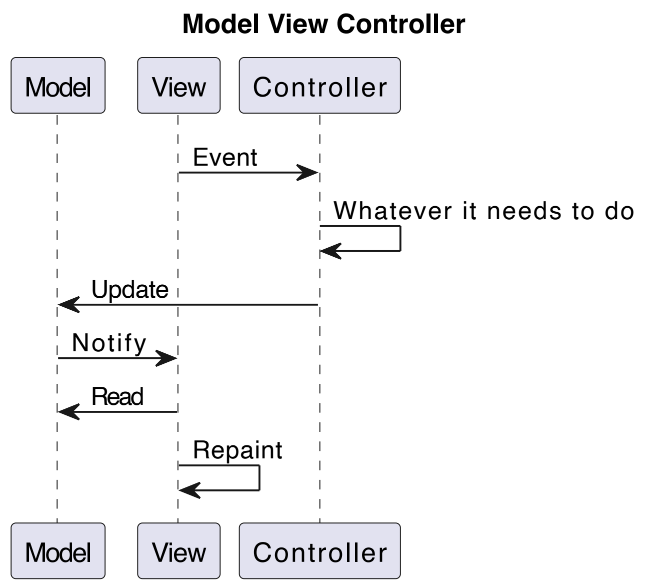


Description: The server is suppose to provide services for clients.

We didn't get to it but basically the idea was that the server could provide any service to any client as long as there exists a service handler.

Sequence Diagrams for program

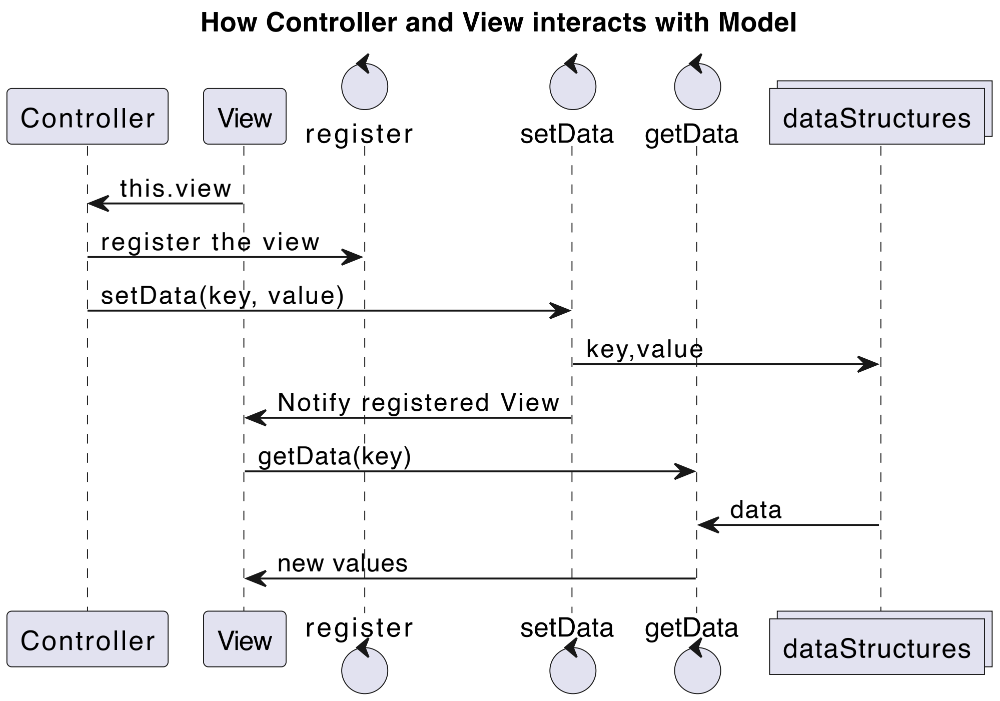
Author: Abir Faisal



Description: The MVC components interact with each other using UUIDs that are defined in models that extend the abstract Model. (Template pattern)

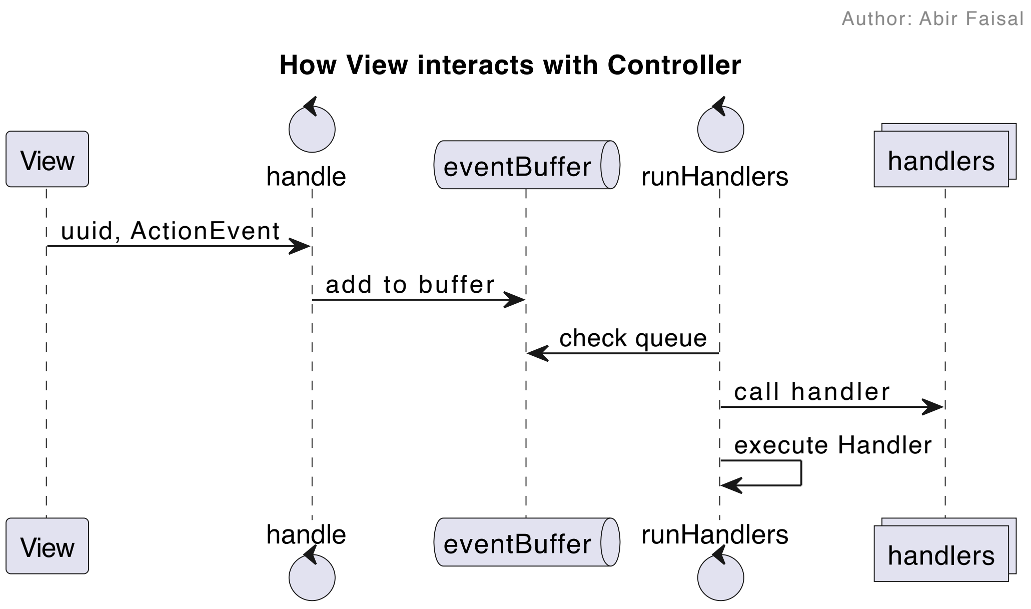
The templates contain everything needed forthe MVC to work. You just have to define the UI components in your View subclass, event handlers in your Controller subclass, and UUID constants in the Model as well as data strctures in the form of record classes.

Author: Abir Faisal



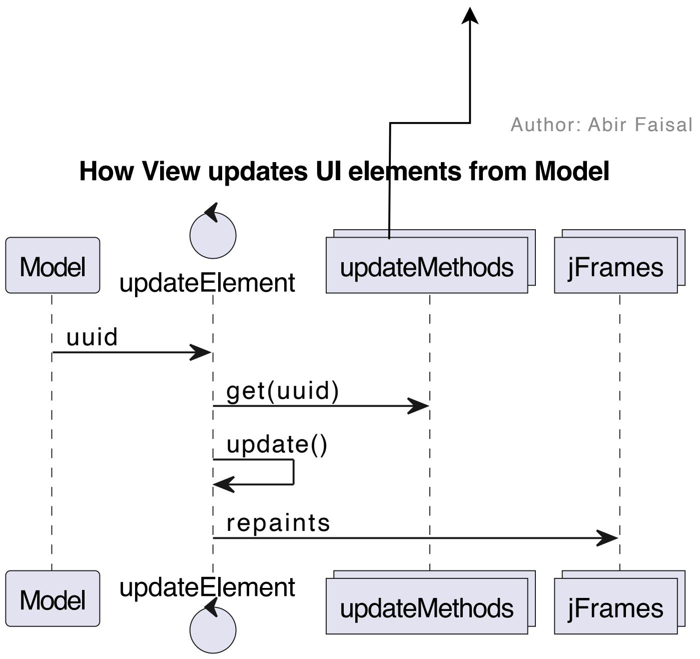
Description: A View is registered to a Model. When the model is updated by the Controller the view is notified that the model has changed. From there the view will update itself from the model.

Sequence Diagrams for program continued



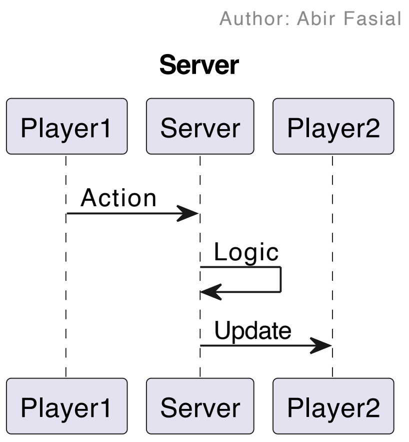
Description: When an event occurs in the view the Controller checks the handlers HashMap to see if it contains a corresponding Handler.

If so then the handler is executed.



Description: Each Swing component has an by declaring an new Updater with the Updater interface and putting it into the updateMethods hashmap.

When the model notifies the view the View calls the corresponding update method executes



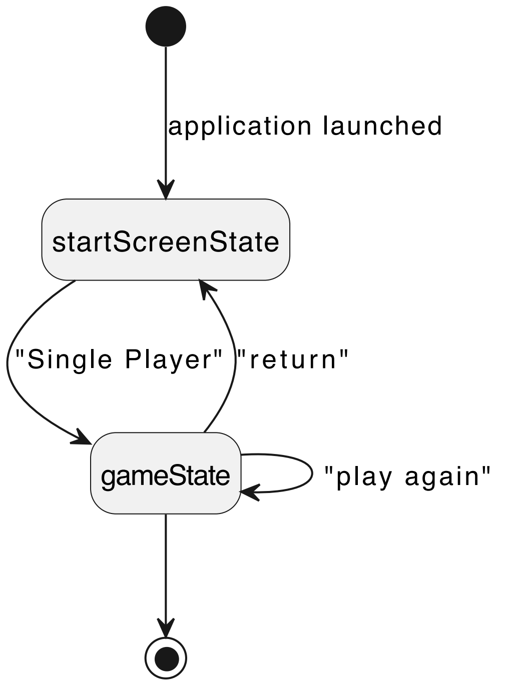
Description: Server
We didn't get to this but basically it's a server that would have players connect to it and it would mediate between them.

State Diagrams:

Description: General Application State Diagrams

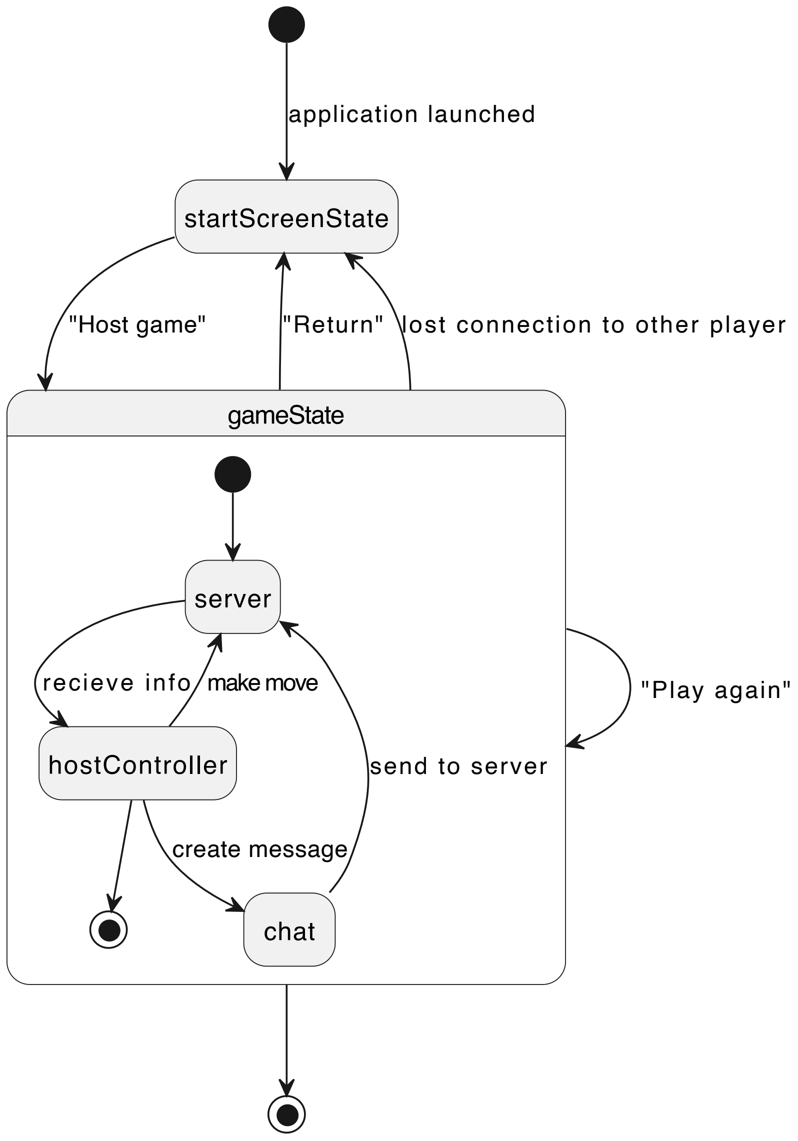
Author: Bryan Barreto

Single Player



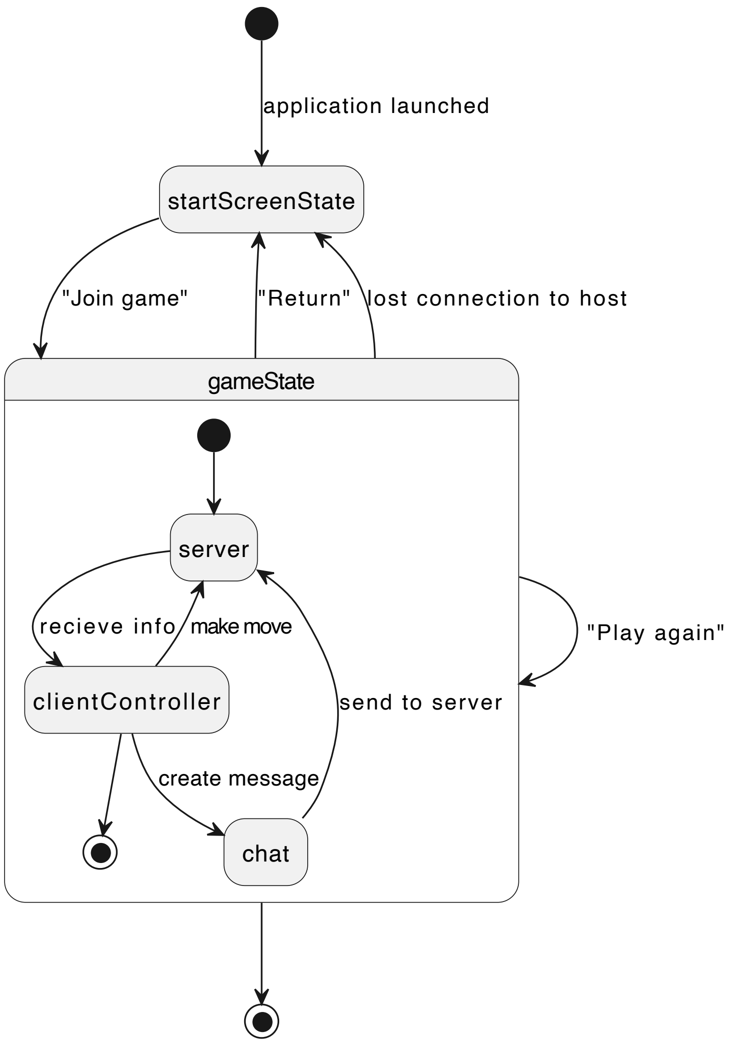
Author: Bryan Barreto

Multi Player Host



Author: Bryan Barreto

Multi Player Join


application launched
startScreenState
"Join game"
"Return"
lost connection to host
gameState
server
recieve info
make move
clientController
send to server
create message
chat
"Play again"