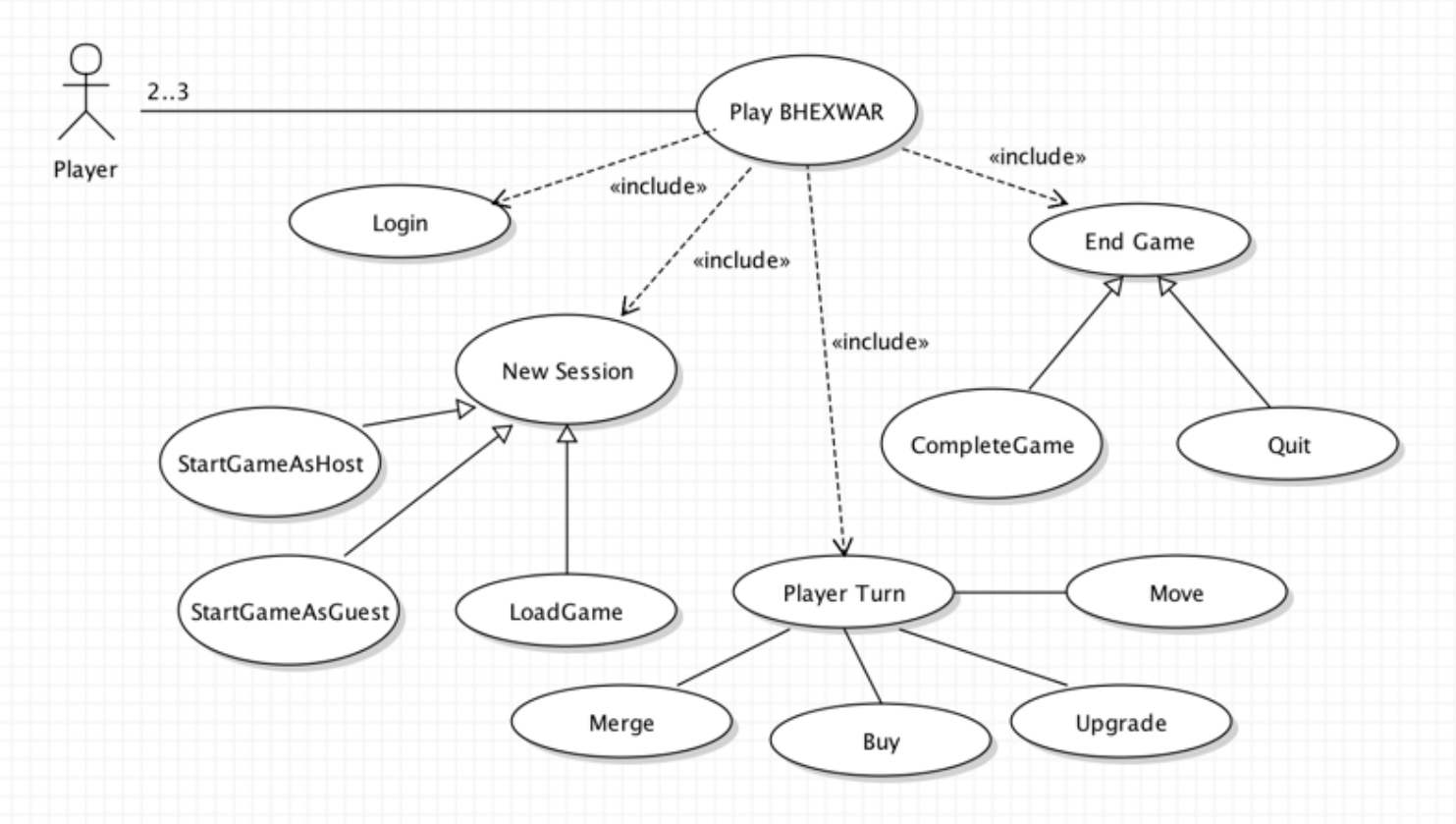
COMP-361 Project Requirements Models

1 Use Case Model



PlayBHEXWAR Use Case
Use Case: PlayBHEXWAR
Scope: Entire Game System
Level: Summary Level
Intention in Context: The intention of the Player is to have fun playing a game with at least one other player in the hopes of winning the game
Multiplicity: Many players can play BHEXWAR at a time, and each player can only run one instance of a client at a time.
Primary Actor: Player
Precondition: Server is running / not down
Main Success Scenario:
1. *Player* Login
2. *Player* starts a New Session
Step 2 can be be done and undone any number of times, but not anymore once player moves on to Step 3.

3. *Player* Plays Turns
Step 3 alternates
Step 3 is repeated until either a player quits game or a winner is determined.
4. End Game

Login Use Case

Use Case: Login
Scope: BHEXWAR
Level: User Goal
Intention in Context: The player intends to login
Multiplicity: Each user can only login into one session at any given time but many users can log in at the same time
Primary Actor: Player
Precondition: Player must be already registered
Main Success Scenario:
1. *Player* sends username and password information for *System* to validate.
2. *System* outputs Main Lobby screen(list of current games, online players, saved games, etc.) to *Player*
Extensions:
1a. Username/password invalid, *System* prompts *Player* to retry.
1b. Username does not exist and *System* creates a new user.

StartGameAsHost Use Case

Use Case: StartGameAsHost
Scope: BHEXWAR
Level: Subfunction Level
Intention in Context: The player intends to create a new game by making decisions on the map and the number of players that can participate in the game
Multiplicity: Many players can host different games on the system, and each player can only host at most one game at a time.
Primary Actor: Host Player
Secondary Actor: Guest Player
Main Success Scenario:
1. *Host player* informs *System* they want to host a new game.
2. *System* outputs the game setup lobby (with game settings, number of players, etc).
3. *Host player* inputs game settings(map settings, game length and number of players that can participate).
4. *System* enables “start game” button to the *Host player* once *Guest player(s)* are ready.
5. *Host player* requests *System* to start the game.
6. *System* outputs new game.

StartGameAsGuest Use Case

Use Case: StartGameAsGuest
Scope: BHEXWAR
Level: Subfunction Level
Intention in Context: Player joins a game that is hosted by another player
Multiplicity: Each user can only join one game at a time but many users can log in at the same time
Primary Actor: Guest Player
Secondary Actor: Host Player
Main Success Scenario:
1. *Guest player* informs *System* they want to join an existing game.
2. *System* outputs the game setup lobby (with game settings, number of players, etc).
3. If *Guest player* agrees to settings, informs *System* they are ready.
4. Once *Host player* requests game start, *System* outputs new game.
Extension:
2a. Game Lobby is full. *System* informs *Player* request was unsuccessful. Use case ends in failure.

LoadGame Use Case

Use Case: LoadGame
Scope: BHEXWAR
Level: Subfunction-level
Intention in Context: Player loads a previously saved game.
Multiplicity: Each user can only LOAD one game at a time but many users can load at the same time on the system.
Primary Actor: Player
Main Success Scenario:
1. *Player* informs *System* to load an existing game.
2. *System* outputs the selected saved game to *Player*.
Extension:
2a. *System* cannot load existing game. Outputs error message to *Player*. Use case ends in failure.

Merge Use Case

Use Case: Merge
Scope: BHEXWAR
Level: Subfunction Level
Intention in Context: Player merges two units to create a better one.
Multiplicity: Each player can only perform one instance of the merge use case at a time, but multiple instances of merge use case can exist in the system since there can exist multiple games that are playing at one time.
Primary Actor: Player
Main Success Scenario:
1. *Player* sends merge option with two units to *System.*
2. *System* outputs updated state of the game to *Player*.
Extensions:
2a. *Player* does not have resources needed to upgrade. *System* informs *Player*. Use case ends in failure.

Buy Use Case

Use Case: Buy
Scope: BHEXWAR
Level: Subfunction Level
Intention in Context: Player upgrades units
Multiplicity: Each player can only perform one instance of the buy use case at a time, but multiple instances of buy use case can exist in the system since there can exist multiple games that are playing at one time.
Primary Actor: Player
Main Success Scenario:
1. *Player* requests the *System* that they want to buy a unit for a specific village.
2. *System* outputs updated state of the game to *Player*.
Extensions:
2a . *Player* does not have resources needed to buy said unit. *System* informs *Player*. Use case ends in failure.

Upgrade Use Case

Use Case: Upgrade
Scope: BHEXWAR
Level: Subfunction Level
Intention in Context: Player upgrades units
Multiplicity: Each player can only perform one instance of the upgrade use case at a time, but multiple instances of upgrade use case can exist in the system since there can exist multiple games that are playing at one time.
Primary Actor: Player
Main Success Scenario:
1. *System* outputs current state of the game to *Player*.
2. *Player* informs the *System* of the game unit they select.
3. *System* displays *Player* UI with available options.
4. *Player* sends upgrade option to *System*.
5. *System* outputs updated state of the game to *Player*.
Extensions:
3a. If unit cannot be upgraded *System* does not display upgrade option to *Player*. Use case ends in failure.
4a. *Player* does not have resources needed to upgrade. *System* informs *Player*. Use case ends in failure.

Move Use Case

Use Case: Move
Scope: BHEXWAR
Level: Subfunction Level
Intention in Context: Player moves unit from one tile to another
Multiplicity: *Player* can only move one unit at a time, but there can be multiple instances of this use case in the system.
Primary Actor: Player
Main Success Scenario:
1. *Player* sends unit move request to *System*.
2. *System* outputs updated state of the game to *Player*.
Extensions:
2a. If unit cannot move this turn, *System* informs *Player*. Use case ends in failure.
2b. If move is not legal, *System* informs *Player*. Use case ends in failure.

CompleteGame Use Case

Use Case: CompleteGame
Scope: BHEXWAR
Level: Subfunction Level (can be User Goal)
Intention in Context: Player's intention is to complete a game either by winning or by losing the game.
Multiplicity: This use case can happen once per player, but there can be multiple instances of this use case in the system since there can be multiple games playing at a time.
Primary Actor: Player
Precondition: All tiles have been captured by one player or only one player remains
Main Success Scenario:
1. *System* informs *Player* of the winner
2. *System* shows *Player* updated Players' statistics.

QuitGame Use Case

Use Case: QuitGame
Scope: BHEXWAR
Level: Subfunction Level (can be User Goal)
Intention in Context: Player intends to exit the game by forfeiting
Multiplicity: This use case can happen once per player, but there can be multiple instances of this use case in the system since there can be multiple games playing at a time.
Primary Actor: Player
Precondition:
Main Success Scenario:
1. *Player* informs *System* they wish to quit the game.
2. *System* outputs option to either "Save" or "Forfeit" to *Player*
3a.1. *Player* chooses "Save"
3a.2. *System* outputs Main Lobby screen.
3b.1 *Player* chooses " Forfeit"
3b.2 *System* outputs End Game message and sends Player to Main Lobby.

2 Architectural Decisions

Chosen architecture: Client-server

We will use centralized computing where all state, user and game information is stored on the server and part of it with the client. This allows the infrastructure to be simpler. The downfall of centralized computing is that it requires many more interactions between the clients and the server but since our game is not real-time, this is not a problem.

Client will be an executable that runs the UI and stores some game logic. It will also take care of computing the legal actions that the player can make once a unit (villager, village, tile, etc.) is selected by the Player. Client manages each unit selection that the player makes until a permanent and legal game move is made, in which case the client will forward that update to the server. That way, the server can push every game state update to all of the clients participating in the game and won't need to handle every mouse click made by the player.

Server will take in the clients' requests and will handle the computation and will modify the state of the game and the user interface.

3 Requirements Models

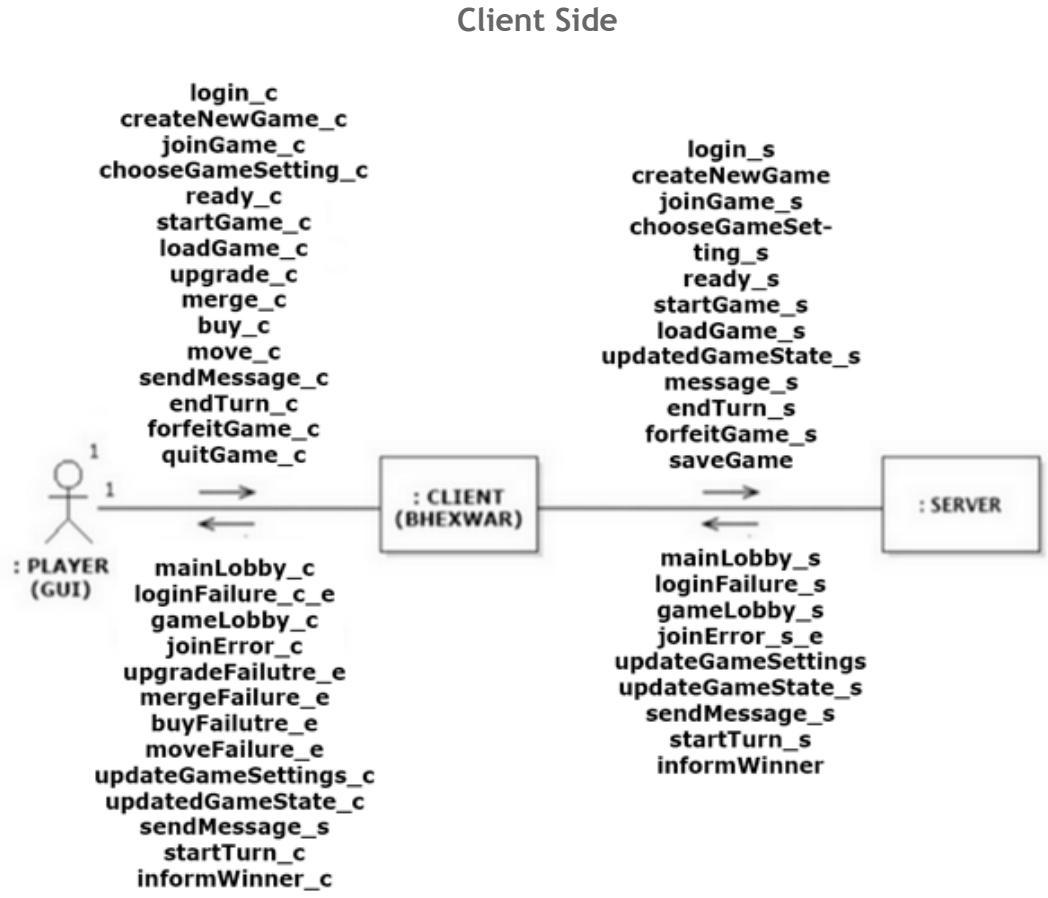
3.1 Structural Requirements (Environment Model and Concept Model)

3.1.1 Environment Model

Type Definitions:

The environment model and concept model of the game assume the existence of the following types:

- `type LoginInfo is TupleType{username: String, password: String}`
- `type VillageRank is enum {Hovel, Town, Fort}`
- `type VillagerRank is enum {Peasant, Infantry, Soldier, Knight}`
- `type Gold is Integer range 0..*`
- `type Wood is Integer range 0..*`
- `type Resources is TupleType{g: Gold, w: Wood}`



Input Messages

From Player (GUI)

- `login_c(loginInfo: LoginInfo):` sent by *Player* to communicate the login information
- `createNewGame_c()`: sent by *Player* to create a new game lobby as the host
- `joinGame_c (g: Game):` sent by *Player* to join an existing game lobby
- `chooseGameSetting_c(..):` sent by *Player* to select the configurations of the game. The parameters must clearly identify the configuration parameter that is requested to be modified and the value to which it should be set.
- `ready_c(ready: Boolean):` sent by *Player* when they are ready to start the game
- `startGame_c():` sent by *Player* to start the game
- `loadGame_c():` sent by *Player* to request the loading of a previously saved game
- `upgrade_c(u: Unit):` sent by *Player* to upgrade a villager or a village unit
- `merge_c(u1: Unit, u2: Unit):` sent by *Player* to merge 2 villagers into a single unit

- `buy_c(u: Unit, v: Village)`: sent by *Player* to buy a new villager from a village
- `move_c(x, y : Integer)`: sent by *Player* to move a villager unit to an adjacent hex tile
- `sendMessage_c(m: String, p: Player)`: sent by *Player* to send a message string to another player
- `endTurn_c()`: sent by *Player* to indicate that it has finished playing his turn
- `forfeitGame_c()`: sent by *Player* to quit the game before a winner has been determined. The Player loses by default.
- `quitGame_c()`: sent by *Player* to quit the game before it is finished. The game is automatically saved.

From Server

- `mainLobby_s(l: List of Game, p: List of User)`: sent by *Server* when the login information provided is correct
- `loginFailure_s_e()`: sent by *Server* when the login information provided is incorrect
- `gameLobby_s(l: List of User,)`: sent by *Server* when *Player* creates a new game lobby as host or requests to join an existing game lobby
- `joinError_s_e()`: sent by *Server* when the game lobby is full or when the join request is unsuccessful
- `updateGameSettings_s(..)`: sent by *Server* when the selected configurations of the game have been updated
- `updatedGameState_s(g: Game)`: sent by *Server* when the state of the game has been changed and updated
- `sendMessage_s(m: String)`: sent by *Server* when *Player* sends a message string to another player
- `startTurn_s()`: sent by *Server* to indicate that it is the *Player's* turn
- `informWinner()`: sent by *Server* when a winner has been determined

Output Messages

To Server

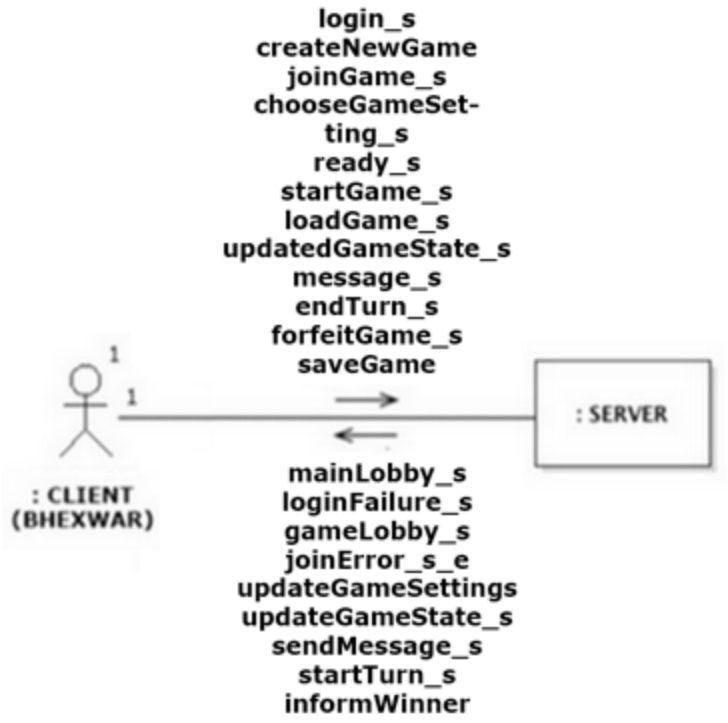
- `login_s(username: String, password: String)`: sent to *Server* to communicate the login information provided by the *Player*
- `createNewGame`: sent to *Server* when *Player* wants to create a new game lobby as the host
- `joinGame_s(g: Game)`: sent to *Server* when *Player* wants to join an existing game lobby
- `chooseGameSetting_s(..)`: sent to *Server* when *Player* selects the configurations of the game. The parameters must clearly identify the configuration parameter that is requested to be modified and the value to which it should be set.

- `ready_s(ready: bool)`: sent to *Server* when the *Player* is ready to start the game
- `startGame_s(g: Game)`: sent to *Server* to start the game
- `loadGame_s(g: Game)`: sent to *Server* when *Player* requests the loading of a previously saved game
- `updateGameState_s(g: Game)`: sent to *Server* when *Player* changed or updated the state of the game
- `message_s(s: String)`: sent to *Server* when *Player* sends a message string to another player
- `endTurn_s()`: sent to *Server* when *Player* decides to end a turn
- `forfeitGame_s()`: sent to *Server* when *Player* quits the game before a winner has been determined. The Player loses by default.
- `saveGame()`: sent to *Server* when *Player* quits the game before it is finished. The game is automatically saved.

To Player (GUI)

- `mainLobby_c(l: List of Game, p: List of Player)`: sent to *Player* the login information provided is correct
- `loginFailure_c_e()`: sent to *Player* the login information provided is incorrect
- `gameLobby_c(..)`: sent to *Player* when a new game lobby is created or when the *Player* joins an existing game lobby
- `joinError_c_e()`: sent to *Player* when the game lobby is full or when the join request is unsuccessful
- `upgradeFailure_e()`: sent to *Player* when they don't have the necessary ressources to upgrade the selected unit
- `mergeFailure_e()`: sent to *Player* when merging villager requirements were not met
- `buyFailure_e()`: sent to *Player* when villager buying requirements are not met
- `moveFailure_e()`: sent to *Player* when villager buying requirements are not met
- `updateGameSettings_c(..)`: sent to *Player* when the selected configurations of the game have been updated
- `updatedGameState_c(g: Game)`: sent to *Player* when the state of the game has been changed and updated
- `sendMessage_c(s: String)`: sent to *Player* to receive a message string sent by another player
- `startTurn_c()`: sent to *Player* to indicate that it is their turn to play
- `informWinner_c()`: sent to *Player* when a winner has been determined

Server Side



Input Messages

From Client

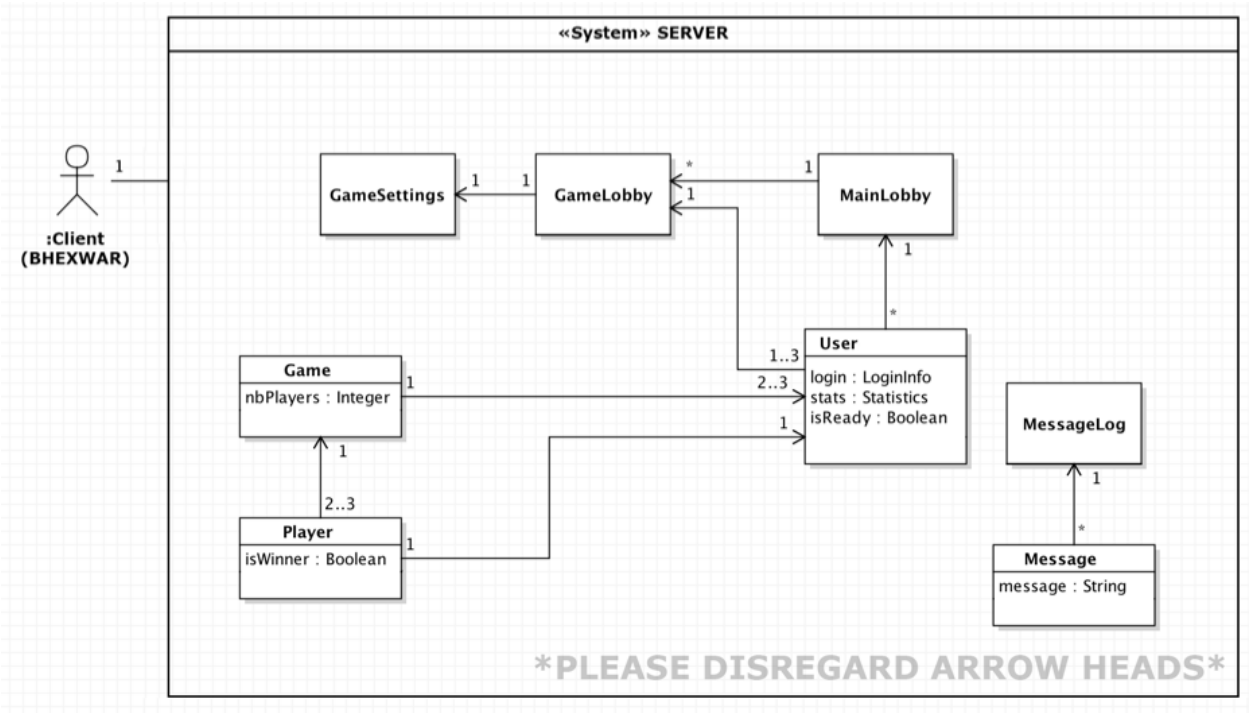
- `login_s(loginInfo: LoginInfo)`: sent by *Client* to verify credentials
- `createNewGame_s()`: sent by *Client* to create a new game lobby
- `joinGame_s(g: Game)`: sent by *Client* to join an existing game lobby
- `chooseGameSetting_s(..)`: sent by *Client* to select the configurations of the game. The parameters must clearly identify the configuration parameter that is requested to be modified and the value to which it should be set.
- `ready_s(ready: bool)`: sent by *Client* when ready to start the game
- `startGame_s()`: sent by *Client* to request to start the game
- `loadGame_s(g: Game)`: sent by *Client* to request the loading of a previously saved game
- `updateGameState_s(g: Game)`: sent by *Client when* the state of the game has been updated
- `sendMessage_s(s: String)`: sent by *Client* to send a message string
- `endTurn_s()`: sent by *Client* to indicate the end a turn
- `forfeitGame_s()`: sent by *Client* when a game is quit before a winner has been determined. This results in a loss by default.
- `saveGame(g: Game)`: sent by *Client* when a game is quit before it is finished. The game is automatically saved.

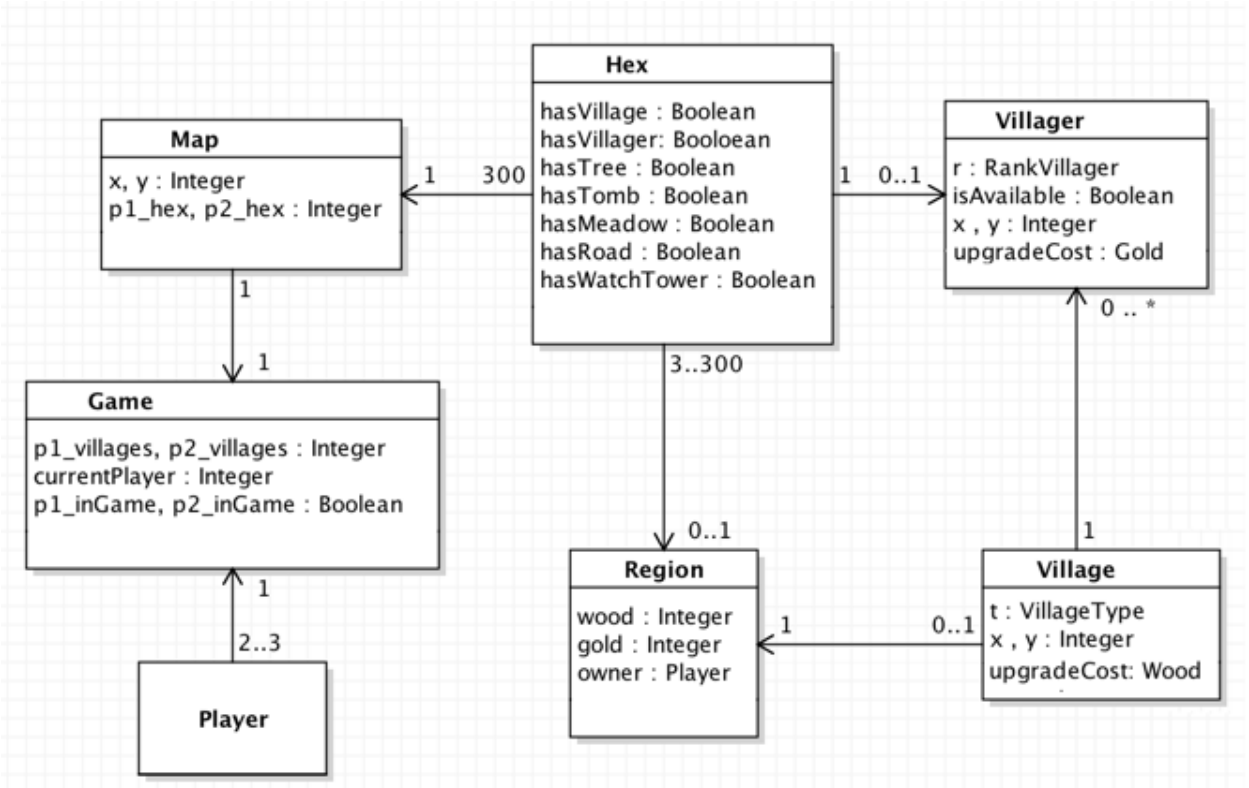
Output Messages

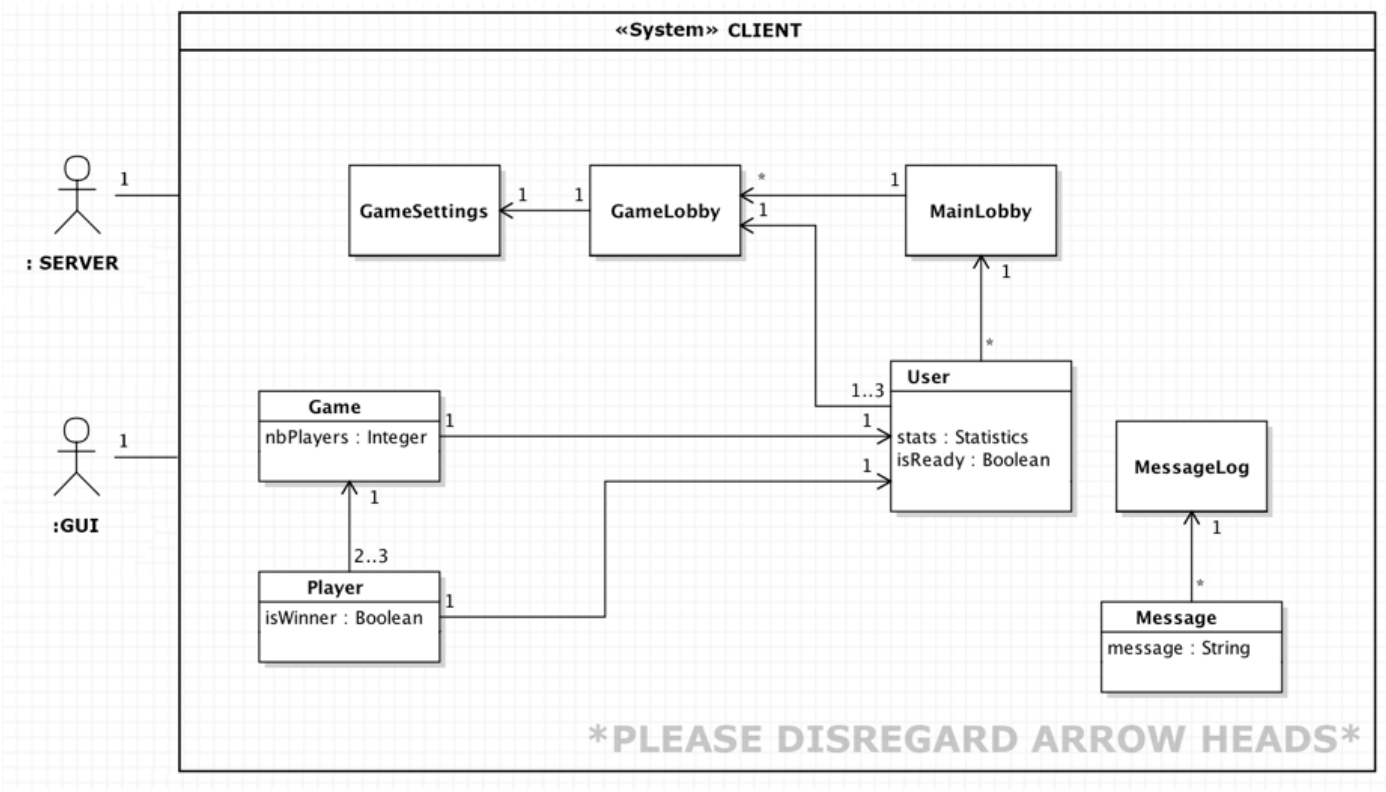
To Client

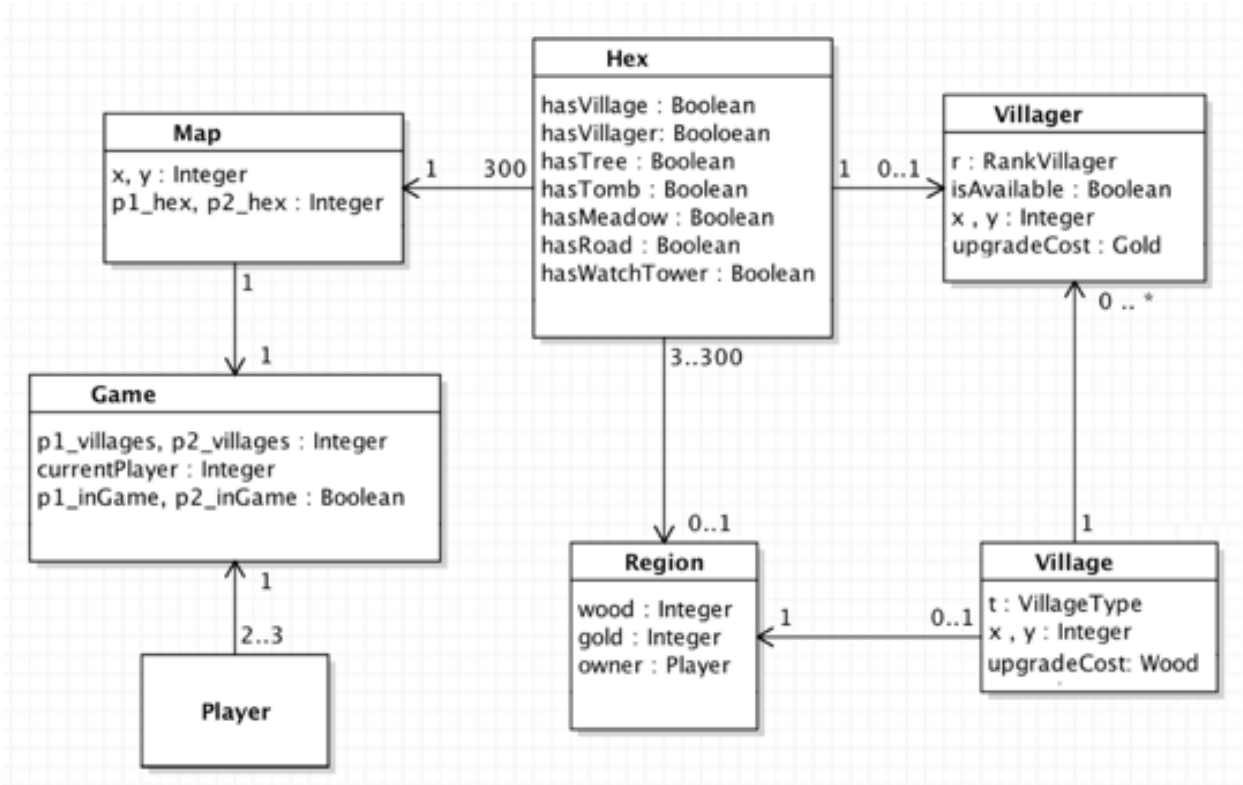
- `mainLobby_s(l: List of Game, p: List of Player)`: sent to *Client* when the login information provided is verified and correct
- `loginFailure_s(s: String)`: sent to *Client* when the login information provided is verified and incorrect
- `gameLobby_s()`: sent to *Client* when a new game lobby is created or when a request to join an existing game lobby is sent
- `joinError_s()`: sent to *Client* when the game lobby is full or when the join request is unsuccessful
- `updateGameSettings_s(..)`: sent to *Client* when the selected configurations of the game have been updated
- `updatedGameState_s(g: Game)`: sent to *Client* when the state of the game has been changed and updated
- `sendMessage_s(..)`: sent to *Client* when a message string is received from another *Client*
- `startTurn_s()`: sent to *Client* to indicate start of a new turn
- `informWinner(s: String)`: sent to *Client* when a winner has been determined

3.1.2 Concept Model






«System» CLIENT
: SERVER
:GUI
GameSettings
GameLobby
MainLobby
User
stats : Statistics
isReady : Boolean
Game
nbPlayers : Integer
Player
isWinner : Boolean
MessageLog
Message
message : String
PLEASE DISREGARD ARROW HEADS


Map
x, y : Integer
p1_hex, p2_hex : Integer
Hex
hasVillage : Boolean
hasVillager: Booloean
hasTree : Boolean
hasTomb : Boolean
hasMeadow : Boolean
hasRoad : Boolean
hasWatchTower : Boolean
Villager
r : RankVillager
isAvailable : Boolean
x , y : Integer
upgradeCost : Gold
Game
p1_villages, p2_villages : Integer
currentPlayer : Integer
p1_inGame, p2_inGame : Boolean
Player
Region
wood : Integer
gold : Integer
owner : Player
Village
t : VillageType
x , y : Integer
upgradeCost: Wood

3.2 Behavioural Requirements (Operation Model and Protocol Model)

3.2.1 Operation Model

Client Side

Operation: `Client::login_c(login: LoginInfo)`
Description: The client receives a request to verify the login information, and it redirects it to the server.
Scope: User;
Messages: `Server::{login_s};`
Post:
The operation takes the login information provided by the player and sends an output message containing this information to the server.

Operation: `Client::createNewGame_c()`
Description: The client receives a request to create a new game lobby.
Scope: User, MainLobby;
Messages: `Server::{createNewGame_s};`
Post:
The operation takes in a request from the player to create a new game lobby and sends an output message containing this information to the server.

Operation: `Client::joinGame_c(g: Game)`
Description: The client receives a request to join a game lobby.
Scope: MainLobby;
Messages: `Server::{joinGame_s};`
Post:
The operation takes a request from the player to join a game and sends an output message containing this information to the server.

Operation: `Client::chooseGameSetting_c(..)`
Description: The client receives a request to set the game settings in a game lobby.
Scope: GameLobby, GameSetting;
Messages: `Server::{chooseGameSetting_s};`
Post:
The operation takes the game configurations provided by the player and sends an output message containing this information to the server.

Operation: `Client::ready_c()`
Description: The client receives a notification whenever the ready option is toggled.
Scope: User, GameLobby;
Messages: `Server::{ready_s};`
Post:
The operation allows the player to toggle the ready toggle on and off and sends an output message containing this information to the server.

Operation: `Client::startGame_c()`
Description: The client receives a request to start the game in the game lobby.
Scope: GameLobby;
Messages: `Server::{startGame_s};`
Post:
The operation takes a request from the player to start a game and sends an output message containing this information to the server.

Operation: `Client::loadGame_c(g: Game)`
Description: The client receives a request to load and play an previously saved game.
Scope: Game, MainLobby;
Messages: `Server::{loadGame_s};`
Post:
The operation takes a request from the player to load a previously saved game and sends an output message containing this information to the server.

Operation: `Client::upgrade_c(vr: Villager, v: Village)`
Description: The client receives a request to upgrade a village or villager.
Scope: Game, Village, Villager;
Messages: `Player::{upgradeFailure_e}; Server::{updatedGameState_s};`
Post:
The operation takes a request from the player to upgrade a villager or a village. If the player has enough resources to upgrade the selected villager or village, the operation sends an output message containing this information to the server. Else, the operation informs the player that the upgrade failed.

Operation: `Client::merge_c(v1: Villager, v2: Villager)`
Description: The client receives a request to merge two villagers.
Scope: Game, Villager, Village;
New: newVillager: Villager;
Messages: `Player::{mergeFailure_e}; Server::{updatedGameState_s};`
Post:
The operation takes a request from the player to merge two villagers. If the merge is valid (i.e., peasant and peasant is a valid merge while soldier with infantry is not) and the operation sends an output message containing this information to the server. Else, the operation informs the player that the merge failed.

Operation: `Client::buy_c(v: Villager, v: Village)`
Description: The client receives a request to buy a villager from a village.
Scope: Game, Villager, Village, Hex;
New: newVgr: Villager;
Messages: `Player::{buyFailure_e}; Server::{updatedGameState_s};`
Post:
The operation takes a request from the player to buy a villager from a village. If the player has enough resources to buy the villager, the operation sends an output message containing this information to the server. Else, the operation informs the player that the buy failed.

Operation: `Client::move_c(x, y: Integer, v: Villager)`
Description: The client receives a request to move a villager.
Scope: Game, Villager, Hex;
Messages: `Player::{moveFailure_e}; Server::{updatedGameState_s};`
Post:
The operation takes a request from the player to move a villager to an adjacent hex tile. If the move is legal, the operation sends an output message containing this information to the server. Else, the operation informs the player that the move failed.

Operation: `Client::sendMessage_c(m: String)`
Description: The client receives a request to send a message.
Scope: Game, Message;
New: newMessage: Message;
Messages: `Server::{sendMessage_s};`
Post:
The operation takes a message string from the player and sends an output message containing this information to the server.

Operation: `Client::endTurn_c();`
Description: The client receives a request from a *Player* to end a turn.
Scope: Game
Messages: `Server::{saveGame, endTurn_s};`
Post:
The operation takes a request from the player to end their turn and sends an output message containing this information to the server. The operation also sends the current game state to the server.

Operation: `Client::forfeitGame_c()`
Description: The client receives a request to forfeit the current game.
Scope: Game;
Messages: `Server::{forfeitGame_s};`
Post:
The operation takes a request from the player to forfeit the game. The game results in a loss for the player and the operation sends an output message containing this information to the server.

Operation: `Client::quitGame_c()`
Description: The client receives a request to quit the current game.
Scope: Game;
Messages: `Server::{saveGame}`
Post:
The operation takes a request from the player to save and quit the game and sends an output message requesting to save the current state of the game to the server.

Operation: `Client::mainLobby_s(l: List of Game, p: List of Player)`
Description: The client receives a request to send the main lobby to the player.
Scope: MainLobby, User;
New: newMainLobby: MainLobby;
Messages: `Player::{mainLobby_c};`
Pre: successful login.
Post:
The operation sends the the Main Lobby to the player.

Operation: `Client::loginFailure_s_e()`
Description: The client receives a login failure message from the server and sends it to the Player.
Scope: User;
Messages: `Player::{loginFailure_c_e};`
Post:
The operation sends a login failed message to the player and prompts the player to try again.

Operation: `Client::gameLobby_s(l: List of Players, isReady: Boolean, settings: GameSettings)`
Description: The client receives the GameLobby Settings.
Scope: MainLobby, GameSettings, GameLobby;
Messages: `Player::{gameLobby_c};`
Post:
The operation sends the game lobby to the player.

Operation: `Client::joinError_s_e()`
Description: The client receives a join error message from the server and forwards it to the player.
Scope: User;
Messages: `Player::{joinError_c_e}`
Post:
The operation sends an error message to the player if they failed to join the selected game lobby.

Operation: `Client::updateGameSettings(..)`
Description: The client receives the updated Game Settings
Scope: GameLobby, GameSettings;
Messages: `Player::{updateGameSettings_c}`
Post:
The operation takes the game configurations and sends an output message containing this information to the player.

Operation: `Client::updateGameState_s(g: Game)`
Description: The client receives the updated Game State.
Scope: Game, Player, Village, Villager, Hex, Region, Map;
Messages: `Player::{updateGameState_c};`
Post:
The operation takes the game state and sends an output message containing this information to the player.

Operation: `Client::sendMessage_s(m: String)`
Description: The client receives a request to send a message.
Scope: Message, MessageLog;
New: newMessage: Message;
Messages: `Player::{sendMessage_c};`
Post:
The operation takes the message string and sends an output message containing this information to the player. The message is appended to the message log.

Operation: `Client::startTurn_s();`
Description: The server informs the Client of the new play turn.
Scope: Game
Messages: `Player::{startTurn_c};`
Post:
The operation informs the player that their turn has started.

Operation: `Client::informWinner()`
Description: The client receives a message that declares a winner.
Scope: Game, MainLobby, User;
Messages: `Player::{informWinner_c};`
Post:
The operation informs the player about the determined winner and updates all involved players their user account statistics.

Server Side

Operation: `Server::login_s(login: LoginInfo)`
Description: The server receive a request to verify the login information.
Scope: User, MainLobby;
Messages: `Client::{mainLobby_s, loginFailure_s};`
Post:

The operation verifies the login information provided by the client. If the login information corresponds to an entry of user accounts in the database, the user will be added to the main lobby and an output message with the main lobby is sent to the client. If the username provided does not exist in the database yet, a new account is automatically registered with the given login information. If the username is already registered in the database, but the provided password is wrong, then the operation sends an error message about the failed login to the client.

Operation: `Server::createNewGame()`
Description: The server receives a request to create a new game lobby.
Scope: GameLobby, MainLobby, User;
New: newGameLobby: GameLobby;
Messages: `Client::{gameLobby_s};`
Post:

The operation creates a new game lobby, updates the list of current game lobbies in the Main Lobby and sends an output message containing the new game lobby to the client.

Operation: `Server::joinGame_s(g: Game)`
Description: The server receives a request to join a game lobby.
Scope: GameLobby, MainLobby, User;
Messages: `Client::{gameLobby_s, joinError_s};`
Post:

If the game lobby is not full, the operation adds a player to an existing game lobby. Else, the operation sends an output error message, notifying the client that the joining was unsuccessful.

Operation: `Server::chooseGameSetting_s(..)`
Description: The server receives a request to change game lobby settings.
Scope: GameLobby, GameSettings;
New: newGameSettings: GameSettings;
Messages: `Client::{updateGameSettings_s};`
Post:

The operation updates the settings of Game Lobby and returns the updated Game Lobby to the client.

Operation: `Server::ready_s()`
Description: The server receives a notification from the client that its user is ready or not to start the game.
Scope: GameLobby, User
Messages: `Client::{updateGameSettings_s};`
Post:
The operation toggles the ready toggle on and off and sends an output message containing this information to the client.

Operation: `Server::startGame_s()`
Description: A request is sent to the server to create and start a new game.
Scope: Game, GameLobby, Player
New: newGame: Game, newPlayer1: Player, newMap: Map, newnewMessageLog: MessageLog;
Messages: `Client::{updateGameState_s};`
Pre: all players are ready
Post:
The operation creates a new game instance with everyone in the game lobby. A new player instance is created for each user. A new map instance as well as a new message log are created for the game.

Operation: `Server::loadGame_s(g: Game)`
Description: A request is sent to the server to load and play an existing game.
Scope: Game, Player
New: newPlayer: Player;
Messages: `Client::{updateGameState_s};`
Post:
The operation loads a previously saved game and creates a new player instance for the user.

Operation: `Server::updateGameSettings_s(..)`
Description:
Scope: Game, Player, User
Messages: `Client::{updatedGameSettings_s};`
Post:
The operation takes the new game state sent by the client and updates the current game state with the updated game state.

Operation: `Server::sendMessage_s(s: String)`
Description: A request is sent to the server to send a message string.
Scope: Message, MessageLog
New: newMessage: Message, newMessageLog: MessageLog;
Messages: `Client::{sendMessage_s};`
Post:
The operation sends a message string to the client.

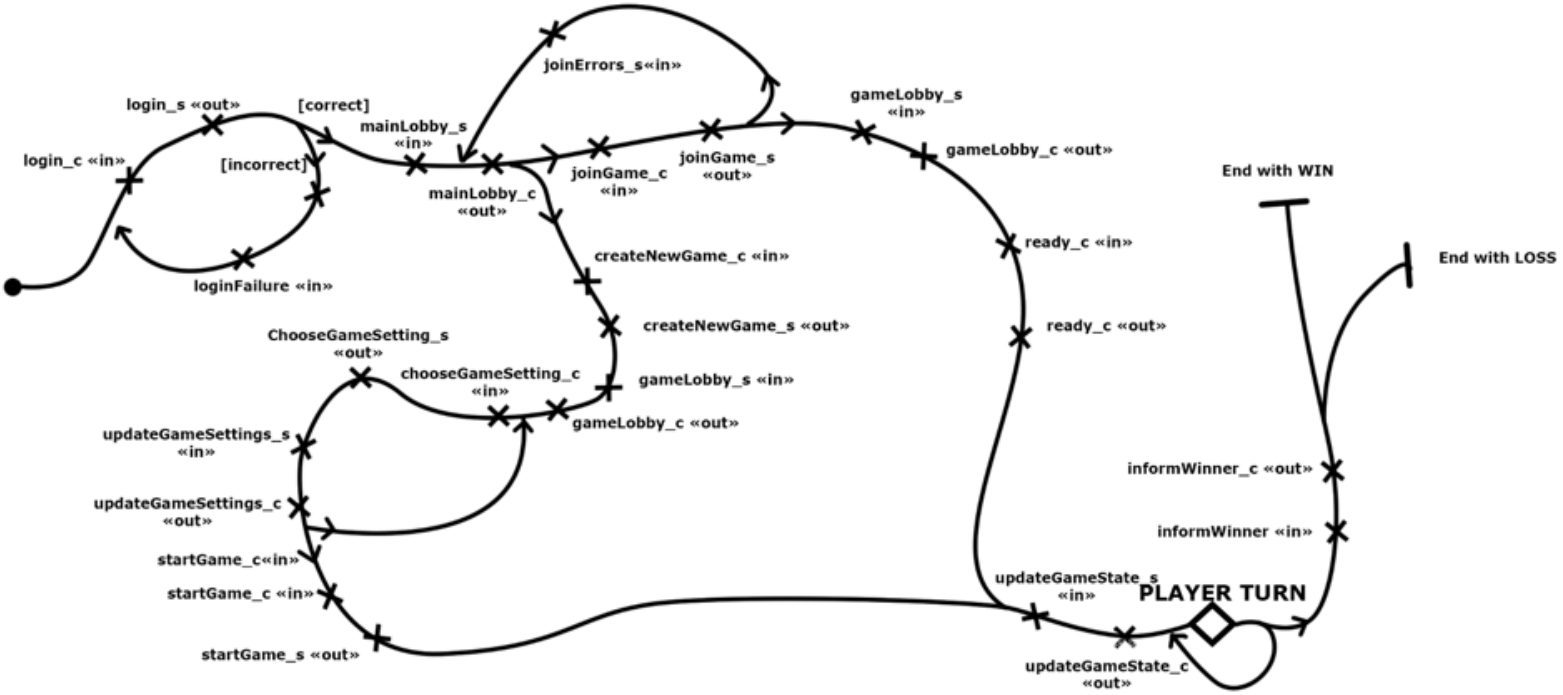
Operation: `Server::endTurn_s();`
Description: The Server receives a request to end a turn.
Scope: Game, Player
Messages: `Client::{startTurn_s, informWinner_s};`
Post:
The operation ends the turn and sends an output message containing this information to the server. If a winner is determined after the end of the turn, the operation informs the client of the winner. Else, the operation informs the client the start of a turn.

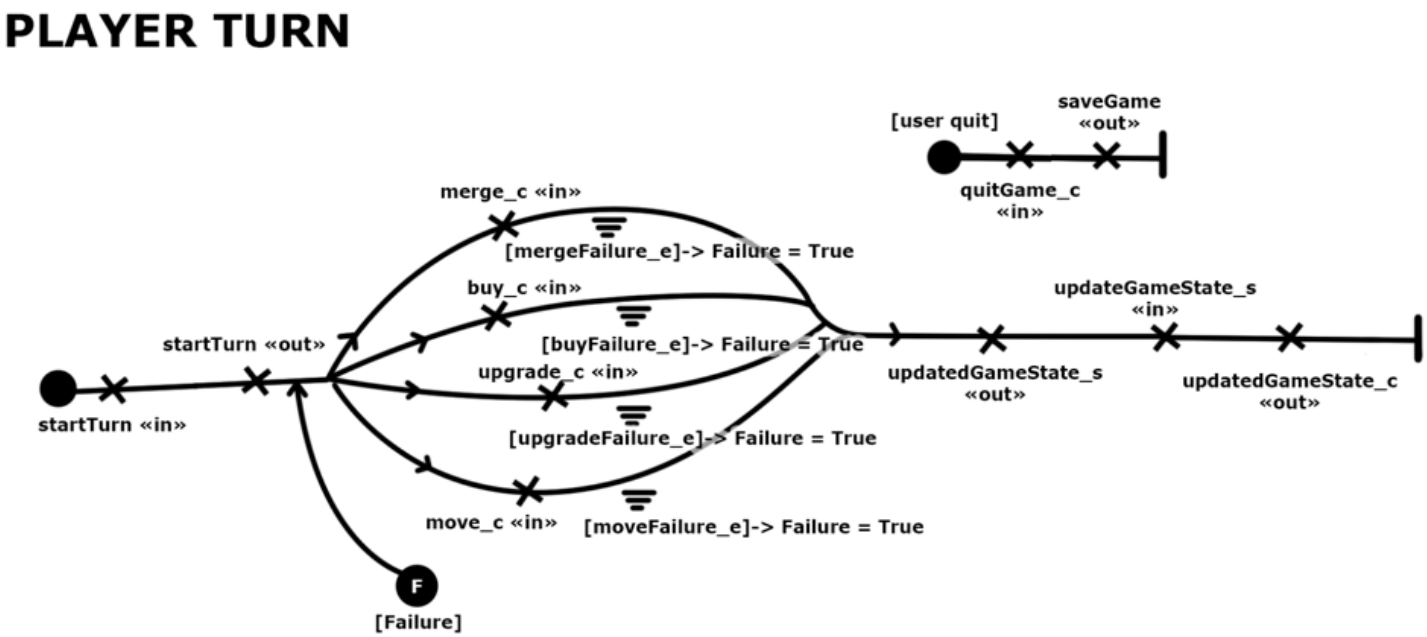
Operation: `Server::forfeitGame_s()`
Description: A request is sent to server to end the game before a winner has been determined. The result is a loss by default.
Scope: Game, Player, User
Messages: `Client::{mainLobby_s};`
Post:
The operation forfeits the game. The game results in a loss and updates the win/lose stats of the user account. All units are turned into neutral territory. The operation updates the game state and sends an output message containing this information to the client.

Operation: `Server::saveGame(g: Game)`
Description: A request is sent to server to save a game.
Scope: Game, User
Messages: `Client::{mainLobby_s};`
Post:
The operation saves the game and sends an output message containing the main lobby to the client.

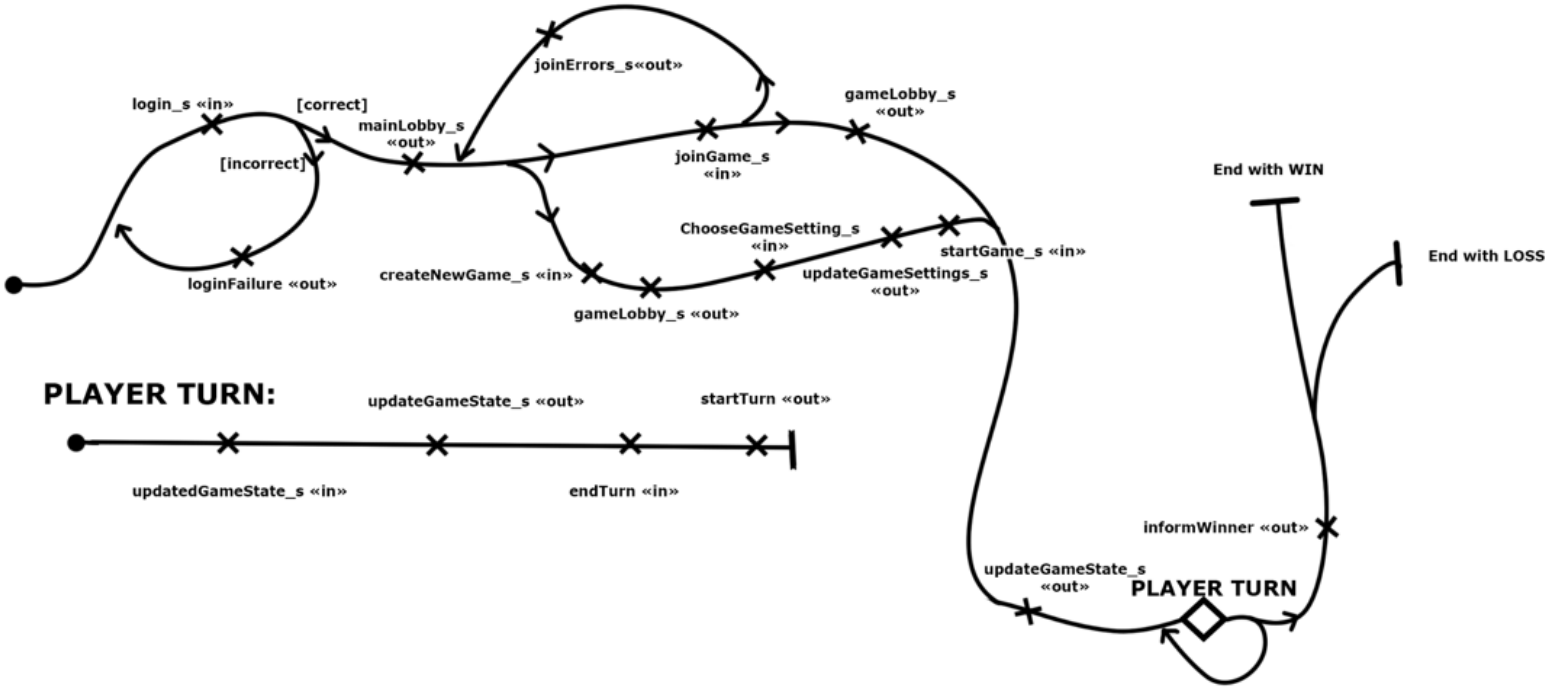
3.2.2 Protocol Model

CLIENT SIDE





SERVER SIDE


login_s «in»
[correct]
[incorrect]
mainLobby_s «out»
joinErrors_s«out»
joinGame_s «in»
gameLobby_s «out»
loginFailure «out»
createNewGame_s «in»
gameLobby_s «out»
ChooseGameSetting_s «in»
updateGameSettings_s «out»
startGame_s «in»
End with WIN
End with LOSS
informWinner «out»
updateGameState_s «out»
PLAYER TURN
PLAYER TURN:
updateGameState_s «out»
startTurn «out»
updatedGameState_s «in»
endTurn «in»