COMP-361 Software Development Project

Milestone 4

Elfen Roads

Requirements Specification Models

November 24, 2021

by Béatrice Duval, Maneth Kulatunge, Alexandru Popian,
Ryan Sowa, Shalee Walsh, and Jifang Wang

Table of Contents

1 Architectural Decisions

For our system, we have decided to build two executables, a *server* and a *client*. As per typical server/client architecture, there will be one *server* instance and multiple *client* instances at run-time. The *server* will handle the main game logic, such as modifying the game based off edition and variant choice, storing the current game state (boot locations, player points and items, etc.) and communicating this state to clients, prompting the clients for the appropriate action at their turn, and executing client actions in the game. The *client* will be responsible for communicating the relevant information to the users through the GUI and verifying user actions before relaying them to the *server* to be executed. To do this, it will receive all the game state information from the *server* but will not modify it directly. It will use the information to check if an action is valid and send valid actions to the *server*, where the game information will be modified and resent to all clients. If an action is invalid, the *client* will communicate through the GUI to have a new action sent. Both the *server* and *client* will interact directly with the REST API lobby service. The *server* will communicate with the lobby service to register the game service and the lobby service will communicate when to create a new instance. The *client* will communicate with the lobby service to access the user's account and view, join, and create new games with the lobby's registered game services.

2 Requirements Specification Models

2.1 Structural Requirements

2.1.1 Environment Model

1. Server

Input Messages (sent from Client to Server)

- setGameEdition(GameEdition)
- setGameVariant(GameVariant)
- pass()
- endTurn()
- quitGame()
- Replies to Server Prompt
    - replyPlaceBid(goldValue: Integer)
    - replyMoveBoot(cards: Set{Card}, town:Town)
    - replyChooseGold()
    - replyChooseDrawTwoCards()
    - replyDrawFaceUpCard(card: Card)
    - replyDrawFaceDownCard()
    - replyCollectGoldStack()
    - replyDrawFaceUpToken(token: Token)
    - replyDrawFaceDownToken()
    - replyHideTokens(tokens:Set{Token})
    - replyPlaceToken(e: Edge, token: Token)
    - replyExchangeToken(token:Token)
    - replyExtraTokenForRoad(token:Token)
    - replyChooseTokensToKeep(tokens:Set{Token})
    - replyChooseCardsToKeep(cardsToKeep: Set{Card})
    - replySaveGame()
    - replyChooseBoot(color:Color)
- deleteSaveGame(GameIdentifier)

Outputs (sent from Server to Client)

- gameStarted()
- getGameEdition(Set{GameEdition})
- getGameVariant(Set{GameVariant})
- currentGameBoard(GameBoard)

- newPhase(GamePhase)
- endPhase(GamePhase)
- newRound(currRound: Integer)
- yourTurn()
- startOpponentTurn(Player)
- currentBid(Player, Integer, Token)
- Prompt Client
    - promptChooseBoot(color: Set{Color})
    - promptPlaceBid()
    - promptMoveBoot()
    - promptTakeGoldOrDrawTwoCards(goldEarned: Integer)
    - promptDrawCard()
    - promptDrawCardOrGoldStack()
    - promptDrawToken()
    - promptDrawFaceDownToken()
    - promptHideTokens(numTokensToHide: Integer)
    - promptPlaceToken()
    - promptExchangeToken(token:Token)
    - promptExtraTokenForRoad(token:Token)
    - promptChooseTokensToKeep(numTokensToKeep:Integer)
    - promptChooseCardsToKeep(numCardsToKeep: Integer)
    - promptSaveGame()
- gameInstanceDeleted()
- playerQuitProtocol(p: Player)
- gameOverMessage(winner: Player)

Input Messages (sent from LobbyService to Server)

- newGameInstance(SessionInformation:String)
- deleteGameInstance(GameIdentifier:String)

Outputs (sent from Server to LobbyService)

- registerGameService(GameServiceInfo:String)
    - Note: this is triggered upon Server start-up
- removeGameServiceRegistration()
    - Note: this is triggered upon Server shut down
- registerSaveGame(SaveGameInfo:String)
- deleteSaveGame(GameIdentifier:String)
- removeGameInstance()

2. Client

Input Messages (sent from Server to Client)

- gameStarted()
- getGameEdition(Set{GameEdition})
- getGameVariant(Set{GameVariant})
- currentGameBoard(GameBoard)
- newPhase(GamePhase)
- endPhase(GamePhase)
- newRound(currRound: Integer)
- yourTurn()
- startOpponentTurn(Player)
- currentBid(Player, Integer, Token)
- Prompt Client
    - promptChooseBoot(color: Set{Color})
    - promptPlaceBid()
    - promptMoveBoot()
    - promptTakeGoldOrDrawTwoCards(goldEarned: Integer)
    - promptDrawCard()
    - promptDrawCardOrGoldStack()
    - promptDrawToken()
    - promptDrawFaceDownToken()
    - promptHideTokens(numTokensToHide: Integer)
    - promptPlaceToken()
    - promptExchangeToken()
    - promptExtraTokenForRoad()
    - promptChooseTokensToKeep(numTokensToKeep:Integer)
    - promptChooseCardsToKeep(numCardsToKeep: Integer)
    - promptSaveGame()
- gameInstanceDeleted()
- playerQuitProtocol(p: Player)
- gameOverMessage(winner: Player)

Outputs (sent from Client to Server)

- setGameEdition(GameEdition)
- setGameVariant(GameVariant)
- pass()
- endTurn()
- quitGame()
- Replies to Server Prompt

    - replyPlaceBid(goldValue: Integer)
    - replyMoveBoot(cards: Set{Card}, town:Town)
    - replyChooseGold()
    - replyChooseDrawTwoCards()
    - replyDrawFaceUpCard(card: Card)
    - replyDrawFaceDownCard()
    - replyCollectGoldStack()
    - replyDrawFaceUpToken(token: Token)
    - replyDrawFaceDownToken()
    - replyHideTokens(tokens:Set{Token})
    - replyPlaceToken(e: Edge, token: Token)
    - replyExchangeToken(token:Token)
    - replyExtraTokenForRoad(token:Token)
    - replyChooseTokensToKeep(tokens:Set{Token})
    - replyChooseCardsToKeep(cardsToKeep: Set{Card})
    - replySaveGame()
    - replyChooseBoot(color:Color)
- deleteSaveGame(GameIdentifier)

Input Messages (sent from GUI to Client)

- logIn(username:String, password:String)
- logOut()
- refreshLobbyScreen()
- loadGameSession(GameService, saveGameID)
- retrieveSaveGames()
- createNewGameSession(GameService)
- launchGameSession()
- deleteUnlaunchedGameSession(sessionID)
- joinGameSession(sessionID)
- cancelJoinGameSession(sessionID)
- deleteSaveGame(GameIdentifier)
- quitGame()
- refreshLobby()
- pass()
- endTurn()
- viewTransportationChart()
- hideTransportationChart()
- Replies to prompts
    - replyChooseGameEdition(GameEdition)
    - replyChooseGameVariant(GameVariant)
    - replyPlaceBid(goldValue: Integer)

    - replyMoveBoot(cards: Set{Card}, town:Town)
    - replyChooseGold()
    - replyChooseDrawTwoCards()
    - replyDrawFaceUpCard(card: Card)
    - replyDrawFaceDownCard()
    - replyCollectGoldStack()
    - replyDrawFaceUpToken(token: Token)
    - replyDrawFaceDownToken()
    - replyHideTokens(tokens:Set{Token})
    - replyPlaceToken(e: Edge, token: Token)
    - replyExchangeToken(token:Token)
    - replyExtraTokenForRoad(token:Token)
    - replyChooseTokensToKeep(tokens:Set{Token})
    - replyChooseCardsToKeep(cardsToKeep: Set{Card})
    - replySaveGame()
    - replyChooseBoot(color:Color)

Outputs (sent from Client to GUI)

- displayLogInScreen()
- displayLobbyScreen()
- returnSaveGames()
- newPlayerJoinedSession(Player)
- gameSessionCancelledMessage()
- gameStarted()
- displayBoardInformation(GameBoard)
- displayPlayerInformation(Player)
- invalidAction(m: Message)
- newPhase(GamePhase)
- endPhase(GamePhase)
- newRound(currRound: Integer)
- yourTurn()
- opponentTurnMessage(m:Message)
- endTurn()
- displayTransportationChart()
- hideTransportationChart()
- displayCurrentBid(Player, Integer, Token)
- Prompt Action
    - promptChooseGameEdition(Set{GameEdition})
    - promptChooseGameVariant(Set{GameVariant})
    - promptChooseBoot(color: Set{Color})
    - promptPlaceBid()

    - promptMoveBoot()
    - promptTakeGoldOrDrawTwoCards(goldEarned: Integer)
    - promptDrawCard()
    - promptDrawCardOrGoldStack()
    - promptDrawFaceDownToken()
    - promptDrawToken()
    - promptHideTokens(numTokensToHide: Integer)
    - promptPlaceToken()
    - promptExchangeToken()
    - promptExtraTokenForRoad()
    - promptChooseTokensToKeep(numTokensToKeep:Integer)
    - promptChooseCardsToKeep(numCardsToKeep: Integer)
    - promptSaveGame()
- gameInstanceDeleted()
- playerHasQuitMessage(p: Player)
- gameOverMessage(winner: Player)

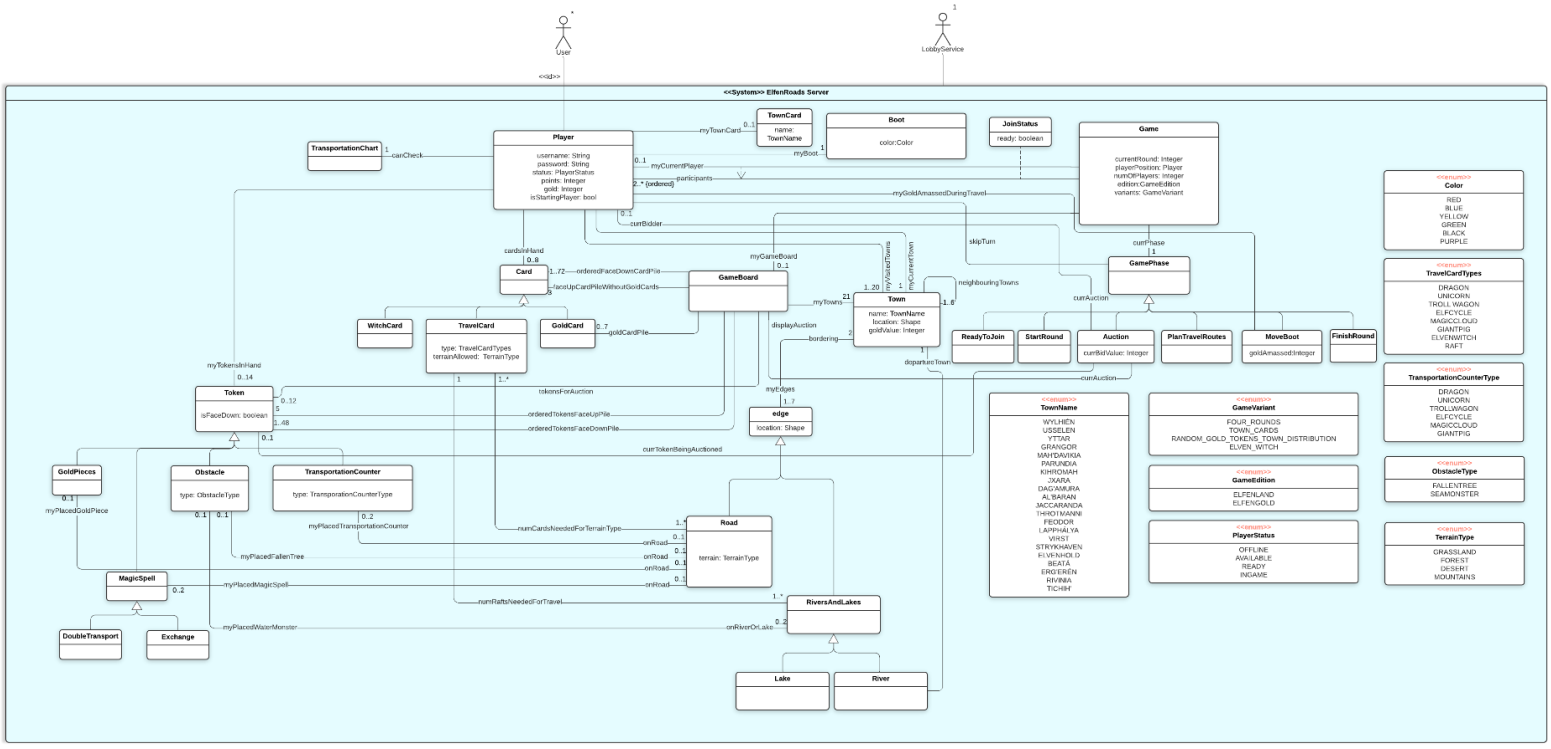
Input Messages (sent from LobbyService to Client)

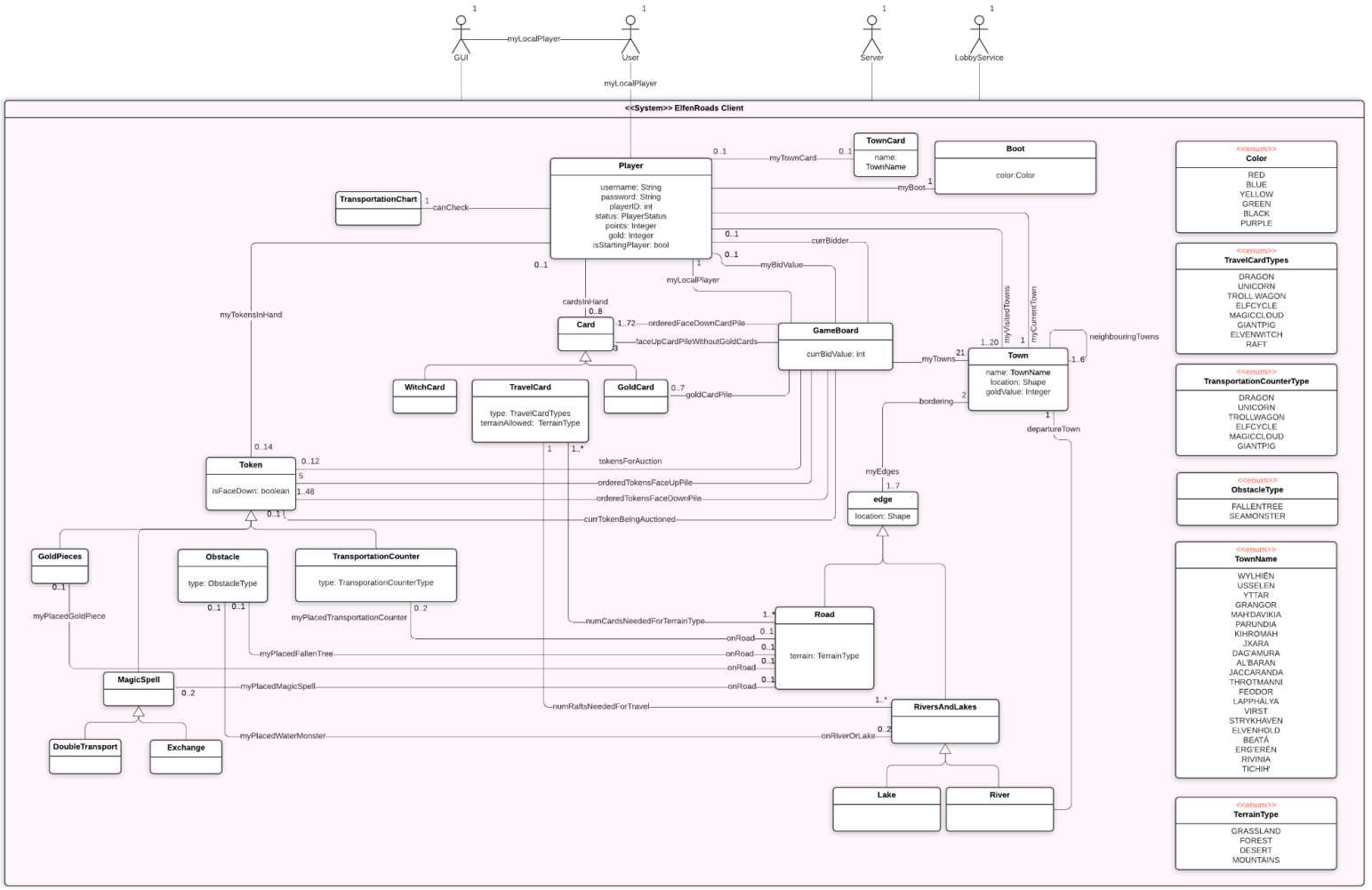
- playerAuthenticationToken(accessToken:String)
- returnGameServices(List<GameServices>)
- returnGameSessions(List<GameSessions>)
- returnSaveGames(List<SaveGames>)
- returnSessionInformation(SessionInfo:String)

Outputs (sent from Client to LobbyService)

- authenticateLogIn(username:String, password:String)
- refreshAuthentication(accessToken)
- logOut(accessToken:String)
- retrieveGameServices()
- retrieveGameSessions()
- loadGameSession(Player, GameService:String, saveGameID:String)
- createNewGameSession(Player, GameService:String)
- launchGameSession(sessionID:String)
- deleteUnlaunchedGameSession(sessionID:String)
- deleteSaveGame(GameIdentifier:String)
- joinGameSession(sessionID:String)
- retrieveSessionInformation(sessionID:String)
- cancelJoinGameSession()
- retrieveSaveGames(GameService:String)

2.1.2 Concept Model





2.2 Behavioral Requirements

2.2.1 Operation Model

1. Server

Operation: ElfenRoads:: setGameEdition(GameEdition)
Scope: Player, Game;
Post: The *setGameEdition* operation sets the *Server* game logic to play either Elfenland or Elfengold.

Operation: ElfenRoads:: setGameVariant(GameVariant)
Scope: Player, Game;
Post: The *setGameVariant* operation sets the *Server* game logic to play any variation of the game they picked

Operation: ElfenRoads:: pass()
Scope: Player, Game;
Post: The *pass* operation notifies the *Server* that the *Player* has passed on their turn and allows it to continue accordingly with the game.

Operation: ElfenRoads:: endTurn()
Scope: Player, Game, Boot;
Post: After the Player is done moving their *Boot*, the *endTurn* operation allows the *Player* to notify the *Server* to end their turn.

Operation: ElfenRoads::quitGame()
Scope: Player, Game;
Messages: Player::{currentGameBoard(GameBoard), playerQuitProtocol(p: Player), promptSaveGame}, LobbyService::{removeGameInstance}
Post: *Server* is notified that *Player* has chosen to quit the *Game*. A prompt is sent to the creator of the game instance to inquire whether they would like to save the game. All *Players* will receive the *playerQuitProtocol*. A message is sent to the LobbyService to notify the removal of a game instance.

Operation: ElfenRoads:: replyPlaceBid(goldValue: Integer)
Scope: Player, Game, Auction;
Messages: Player::{currentBid(Player, Integer, Token)}
Post: The *replyPlaceBid* operation notifies the *Server* of the *Player*'s auction bid. The bid for that *Player* and *Auction* round is then updated and sent to all *Player*s.

Operation: ElfenRoads:: replyMoveBoot(cards: Set{Card}, town:Town)
Scope: Player, Game, Town, WitchCard, Card, Obstacle, Boot;
Messages: Player::{currentGameBoard(GameBoard)}
Post: The *replyMoveBoot* operation notifies the *Server* of the *Cards* and Town the *Player* has selected to move their *Boot*.

- If one of the Cards selected is a WitchCard:
    - If the *WitchCard* is used to bypass an *Obstacle*: The gold value of the *Player* is reduced by one.
    - If the *WitchCard* is used for a Magic Flight: The gold value of the *Player* is reduced by three.

The *Boot's* position, the town pieces for the corresponding *Town*, the *Player*'s points and gold are updated and sent to all *Players*.

Operation: ElfenRoads:: replyChooseGold()
Scope: Player, Game, GameBoard;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: The Player has received the message *promptTakeGoldOrDrawTwoCards()* and has accumulated potential gold during their turn from visiting towns.
Post: The *replyChooseGold* operation notifies the *Server* that the *Player* has chosen to take gold at the end of their turn to move their boot. The *Player*'s gold value is updated correspondingly and the new GameBoard is sent to all *Players*.

Operation: ElfenRoads:: replyChooseDrawTwoCards()
Scope: Player, Game, Card, GameBoard;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: The Player has received the message *promptTakeGoldOrDrawTwoCards()* at the end of their turn to move their boot.
Post: The *replyChooseDrawTwoCards* operation notifies the *Server* that the *Player* has chosen to draw two cards. The *Player*'s card deck and the GameBoard's orderedFaceDownCardPile are updated correspondingly.

Operation: ElfenRoads:: replyDrawFaceUpCard(card: Card)
Scope: Player, Game, Card, GameBoard;
New: FaceUpCard;
Messages: Player::{currentGameBoard(GameBoard)}
Post: The *replyDrawFaceUpCard* operation notifies the *Server* which face-up card the *Player* has selected to draw. The *Player*'s card deck is updated correspondingly. The GameBoard's faceUpCardPileWithoutGoldCards is refilled with a card from the orderedFaceDownCardPile.

Operation: ElfenRoads:: replyDrawFaceDownCard()
Scope: Player, Game, Card, GameBoard;
Messages: Player::{currentGameBoard(GameBoard), promptDrawCardOrGoldStack}
Post: The *replyDrawFaceDownCard* operation notifies the *Server* that the *Player* has chosen to randomly select a face down card from the orderedFaceDownCardPile. The *Player*'s deck and the GameBoard's orderedFaceDownCardPile are updated correspondingly. If a Gold Card is drawn, *Player* is sent the message *promptDrawCardOrGoldStack*.

Operation: ElfenRoads:: replyCollectGoldStack()
Scope: Player, Game, GoldCard, GameBoard;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: Player has just drawn a GoldCard from the deck.
Post: The *replyCollectGoldStack* operation allows the *Player* to collect the stack of gold cards. The *Player*'s gold is increased by 1 for every Gold Card in the Gold Card stack, and the GameBoard is updated correspondingly.

Operation: ElfenRoads:: replyDrawFaceUpToken(token: Token)
Scope: Player, Game, Token, GameBoard;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: *Player* is prompted to draw a transportation token.
Post: The *replyDrawFaceUpToken* operation allows the *Player* to draw a face-up token. The *Player*'s tokens are updated correspondingly. The GameBoard's myOrderedTokensFaceUpPile is refilled from the myOrderedTokensFaceDownPile.

Operation: ElfenRoads:: replyDrawFaceDownToken()
Scope: Player, Game, Token, GameBoard;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: *Player* is prompted to draw a transportation counter.
Post: The *replyDrawFaceDownToken* operation allows the *Player* to randomly draw a face-down transport counter. The *Player*'s tokens and the GameBoard's myOrderedTokensFaceDownPile are updated correspondingly.

Operation: ElfenRoads:: replyHideTokens(tokens:Set{Token})
Scope: Player, Game, GameBoard;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: The Player is prompted to choose the appropriate amount of tokens to hide
Post: The *replyHideTokens* operation allows the *Player* to choose their tokens to hide. The *GameBoard* is updated correspondingly.

Operation: ElfenRoads::replyPlaceToken(e: Edge, token: Token)
Scope: Player, Game, Token, GameBoard, TransportationCounter, Obstacle, GoldPiece, Exchange, DoubleTransport;
Messages: Player::{currentGameBoard(GameBoard), promptExchangeToken, promptExtraTokenForRoad}
Pre: *Player* is prompted to place a token. If an *Exchange*, or *DoubleTransport* is placed, the chosen road must already contain a Transportation Counter.
Post: The *replyPlaceToken* operation allows the *Player* to place a *TransportationCounter*, *Obstacle*, *GoldPiece*, *Exchange*, or *DoubleTransport*. The *GameBoard* is updated correspondingly and the *Player*'s *Token* is removed from their tokensInHand. For tokens requiring further action, a follow up message is sent out.

Operation: ElfenRoads::replyExchangeToken(token:Token)
Scope: Player, Game, Token, Road;
Messages: Player::{currentGameBoard(GameBoard))}
Pre: *Player* is prompted to exchange a token after using an exchange counter. The chosen token is placed on another road.
Post: The *replyExchangeToken* operation allows the *Player* to choose the token they want to switch for on a given road.

Operation: ElfenRoads::replyExtraTokenForRoad(token:Token)
Scope: Player, Game, Token, Road ;
Messages: Player::{currentGameBoard(GameBoard))}
Pre: *Player* is prompted to choose a second token to place on a road after using a x2 counter with *replyPlaceToken*.
Post: The *replyExchangeToken* operation allows the *Player* to choose the token they want to add to the road previously chosen in *replyPlaceToken*.

Operation: ElfenRoads::replyChooseTokensToKeep(tokens:Set{Token})
Scope: Player, Game, Token;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: *Player* is prompted to choose the appropriate amount of tokens to keep.
Post: The *replyChooseTokensToKeep* operation allows the *Player* to choose the token(s) they want to keep at the end of the round.

Operation: ElfenRoads:: replyChooseCardsToKeep(cardsToKeep: Set{Card})
Scope: Player, Game, Card;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: *Player* is prompted to choose the appropriate number of cards to keep.
Post: The *replyChooseCardsToKeep* operation allows the *Player* to choose the cards they want to keep at the end of the round.

Operation: ElfenRoads:: replySaveGame()
Scope: Player, Game, LobbyService;
Messages: LobbyService:: {registerSaveGame(SaveGameInfo)};
Pre: *Player* is prompted to confirm that they want to save the game.
Post: The *replySaveGame* operation allows the *Player* to save the game and notifies the LobbyService.

Operation: ElfenRoads:: replyChooseBoot(color:Color)
Scope: Player, Game, Boot;
Messages: Player::{currentGameBoard(GameBoard)}
Pre: *Player* is prompted to choose a boot.
Post: The *replyChooseBoot* operation allows the *Player* to choose a boot of a specific color. The *Player*'s Boot is updated correspondingly.

Operation: ElfenRoads:: newGameInstance(SessionInformation)
Scope: LobbyService, Game;
New: Player, Game, GameBoard, Card, Token, Town, Edge ;
Messages: Player::{gameStarted}
Post: A new game with session information is created. The server will notify all players that they are now in an active game session.

Operation: ElfenRoads:: deleteGameInstance(GameIdentifier)
Scope: LobbyService, Game;
Messages: Player::{gameInstanceDeleted}
Post: A game with session information is deleted from the LobbyService due to preconditions being breached (such as a player account being deleted). The server will notify all players that the session has been deleted.

Operation: ElfenRoads:: deleteSaveGame(GameIdentifier)
Scope: Player, Game, LobbyService;
Messages: LobbyService::{deleteSaveGame(GameIdentifier)}
Pre: *Player* sending the message is the creator of the game instance.
Post: *Server* notifies the *LobbyService* to delete all information on a saved game.

2. Client

Start Game

Operation: Player::gameStarted()
Scope: Player, Game
Messages: GUI::{gameStarted}
Pre: The PlayerStatus of all participants (*Players*) is Ready.
Post: The *Player* is notified that the game session they have joined has started. The GUI is notified to replace the Lobby Screen with the main game screen.

Choosing the Game Edition

Operation: Player::getGameEdition(Set{GameEdition})
Scope: Player, Game;
Messages: GUI::{promptChooseGameEdition{Set{GameEdition})}
Post: The *getGameEdition* operation prompts the *Player* to set the *Game* edition, and in turn the *Player* prompts the GUI to select a GameEdition, namely ElfenRoads or ElfenLands.

Operation: Player::replyChooseGameEdition(GameEdition)
Scope: Player, GameEdition;
Messages: ElfenRoads::{setGameEdition}, GUI::{invalidAction, promptChooseGameEdition}
Post: The *replyChooseGameEdition* operation informs the *Player* of the *GameEdition* chosen, as detected by the GUI. In turn, the *Server* is notified of the decision. In case the action registered by the GUI is invalid, the *Player* prompts the GUI to output a invalid action message and to select a *GameEdition*.

Choosing the Game Variant

Operation: Player::getGameVariant(Set{GameVariant})
Scope: Player, Game;
Messages: GUI::{promptChooseGameEdition(Set{GameVariant})}
Post: The *getGameVariant* operation prompts the *Player* to set the *Game* variants, and in turn the *Player* prompts the GUI to select a *GameVariant*(s) depending on the *GameEdition*.

Operation: Player::replyChooseGameVariant(GameVariant)
Scope: Player, Game;
Messages: ElfenRoads::{setGameVariant}, GUI::{invalidAction, promptChooseGameVariant}
Post: The *replyChooseGameVariant* operation informs the *Player* of the *GameVariant* chosen, as detected by the GUI. In turn, the *Server* is notified of the decision. In case the action registered by the GUI is invalid, the *Player* prompts the GUI to output a invalid action message and to select a *GameVariant*.

Displaying the Game Board

Operation: Player::currentGameBoard(GameBoard)
Scope: Player, Game, GameBoard, Card*;*
Messages: GUI::{displayBoardInformation(GameBoard), displayPlayerInformation(Player)}
Post: The effect of *currentGameBoard* is to share an update of the state of the *GameBoard* with the *Player* (for example, distributing *Cards* randomly to *Players* at the beginning of the *Game*). In turn, the *Player* requests the GUI to display the general board information and the Player's board information.

Starting a New Phase

Operation: Player::newPhase(GamePhase)
Scope: Player, GamePhase;
Messages: GUI::{newPhase}
Post: The *newPhase* operation informs the *Player* that a new *GamePhase* has started. In turn, the *Player* requests the GUI to display the change of phase.

Ending a Phase

Operation: Player::endPhase(GamePhase)
Scope: Player, GamePhase;
Messages: GUI::{endPhase}
Post: The *endPhase* operation informs the *Player* that a *GamePhase* is ending. In turn, the *Player* requests the GUI to display this change of state.

New Round

Operation: Player::newRound(currRound:Integer)
Scope: Player, Game;
Messages: GUI::{newRound(currRound:Integer)}
Post: The *newRound* operation informs the *Player* that a new round has started. In turn, the *Player* requests the GUI to display the change of round.

Player Turns

Operation: Player::yourTurn()
Scope: Player, Game, GamePhase;
Messages: GUI::{yourTurn}
Post: The *yourTurn* operation informs the *Player* that it is now their turn. For all *GamePhases*, the first *Player* to have a turn is the *startingPlayer*, and the rest play in the order determined by the random set of participants stored in *Game*. In turn, the *Player* requests the GUI to display that it is now this *Player's* turn.

Operation: Player::startOpponentTurn(Player)
Scope: Player, Game;
Messages: GUI::{opponentTurnMessage(m:Message)}
Post: The *startOpponentTurn* operation informs the *Player* that it is now another *Player*'s turn. In turn, the *Player* requests the GUI to display this information.

Passing Turn

Operation: Player::pass()
Scope: Player, GameEdition, Token;
Messages: ElfenRoads::{pass}
Pre: The GUI was just prompted to place a bid, to place a *Token* to plan their move, to move their *Boot*, or any other required turn action.
Post: The *pass* operation informs the *Player* that the user has chosen to pass, as detected by the GUI. In turn, the *Server* is notified of the decision.

Saving Game

Operation: Player::promptSaveGame()
Scope: Player, Game;
Messages: GUI::{promptSaveGame}
Post: The *promptSaveGame* operation asks the *Player* to prompt the GUI with the option to save the game.

Operation: Player::replySaveGame()
Scope: Player, Game;
Messages: ElfenRoads::{replySaveGame}
Post: The *replySaveGame* operation informs the *Player* from the GUI that the choice has been made to save the game. This choice is relayed to the server.

Sharing the Current Bid Info

Operation: Player::currentBid(Player, Integer, Token)
Scope: Player, Auction;
Messages: GUI::{displayCurrentBid(Player, Integer, Token)}
Post: The *currentBid* operation shares with the *Player* information about the current *Auction*, namely the amount of the current highest bid, the corresponding bidder, and the Token that is auctioned. In turn, the *Player* requests the GUI to display the current bid information.

Choosing Boot

Operation: Player::promptChooseBoot(color: Set{Color})
Scope: Player, Game, Boot;
Messages: GUI::{promptChooseBoot(color: Set{Color})}
Post: The *promptChooseBoot* operation prompts the *Player* to choose a *Color* for their *Boot* from the available *Colors*, and in turn the *Player* prompts the GUI to choose a *Boot* color.

Operation: Player::replyChooseBoot(color:Color)
Scope: Player, Game;
Messages: ElfenRoads::{replyChooseBoot(color:Color)}
Post: The *replyChooseBoot* operation informs the *Player* of the *Boot Color* chosen, as detected by the GUI. In turn, the *Server* is notified of the decision.

Placing a Bid

Operation: Player::promptPlaceBid()
Scope: Player, Game, Token;
Messages: GUI::{promptPlaceBid}
Pre: The *Player* has not previously passed on the current *Token*.
Post: The *promptPlaceBid* operation prompts the *Player* to place a bid on a *Token*, using the information previously sent with the *currentBid* operation. In turn, the *Player* prompts the GUI to place a bid.

Operation: Player::replyPlaceBid(goldValue: Integer)
Scope: Player, Game, GameBoard, GameEdition;
Messages: ElfenRoads::{replyPlaceBid(goldValue:Integer)}, GUI::{invalidAction, promptPlaceBid}
Post: The *replyPlaceBid* operation informs the *Player* of the bid made, as detected by the GUI. In case the bid registered by the GUI is lower or equal to the current highest bid, the *Player* prompts the GUI to output a invalid action message and to place a bid. Otherwise, the *Server* is notified of the decision.

Moving Boot

Operation: Player::promptMoveBoot()
Scope: Player, Game, Boot, Town, Card;
Messages: GUI::{promptMoveBoot}
Pre: The *Player* has cards in hand left, and there is at least one valid move possible with their *Cards*.
Post: The *promptMoveBoot* operation prompts the *Player* to move their *Boot* to a *Town* using their *Cards* cardsInHand. In turn, the *Player* prompts the GUI to move the *Boot*.

Operation: Player::replyMoveBoot(cards: Set{Card}, town:Town)
Scope: Player, Game, Boot, Town, Card, GameBoard, WitchCard;
Messages: ElfenRoads::{replyMoveBoot(cards:Set{Card},town:Town), endTurn},
GUI::{invalidAction(m: Message), promptMoveBoot, endTurn}
Post: The *replyMoveBoot* operation informs the *Player* of the *Boot* move made as detected by the GUI. In the case where the move is invalid (check made using *GameBoard* and *Player* state, and checking if the Player has enough gold in case they want to use a *WitchCard*), the *Player* prompts the GUI to output a invalid action message and to move *Boot* again. In cases where the *Player* has no cards in hand left or there is no valid move possible with their *Cards,* the *Player* notifies the *Server* and the GUI that their turn has ended. Otherwise, the *Server* is notified of the valid *Boot* move.

Operation: Player::endTurn()
Scope: Player, Game;
Messages: ElfenRoads::{endTurn}, GUI::{endTurn}
Pre: The GUI was informed by the user that they want to end their turn.
Post: The *endTurn* operation informs the *Player* through the GUI to end the turn. In turn, the *Server* is notified and the decision is confirmed to the GUI

Operation: Player:viewTranportationChart()
Scope: Player, TransportationChart;
Messages: GUI::{displayTransportationChart}
Post: The *viewTransportationChart* operation informs the *Player* through the GUI to display the TransportationChart, and the TransportationChart is displayed to the GUI.

Operation: Player:hideTranportationChart()
Scope: Player, TransportationChart;
Messages: GUI::{hideTransportationChart}
Post: The *hideTransportationChart* operation informs the *Player* through the GUI to hide the TransportationChart, and the TransportationChart is hidden in the GUI.

Ending Travel in Elfengold

Operation: Player::promptTakeGoldOrDrawTwoCards(goldEarned: Integer)
Scope: Player, Game, MoveBoot, Card, GameEdition;
Messages: GUI::{promptTakeGoldOrDrawTwoCards(goldEarned: Integer)}
Pre: The Player has just ended their Boot travel in the ElfenGold *GameEdition* and has accumulated gold in their travels.
Post: The *promptTakeGoldOrDrawTwoCards* operation prompts the *Player* at the end of their travel to either take the gold they have accumulated or draw two *Cards* from the *GameBoard*, and this choice is displayed to the GUI..

Operation: Player::replyChooseGold()
Scope: Player, MoveBoot, GameBoard, Card;
Messages: ElfenRoads::{replyChooseGold}
Pre: *Player* has prompted the GUI to choose between taking gold or drawing two cards after travelling between towns.
Post: The *replyChooseGold* operation informs the *Player* of the choice made to take the gold accumulated during travel, as detected by the GUI. The *Player* notifies the *System* of the decision.

Operation: Player::replyChooseDrawTwoCards()
Scope: Player, MoveBoot, GameBoard, Card;
Messages: ElfenRoads::{replyChooseDrawTwoCards}, GUI::{promptDrawCard}
Pre: *Player* has prompted the GUI to choose between taking gold or drawing two cards after travelling between towns.
Post: The *replyChooserDrawTwoCards* operation informs the *Player* of the choice made to draw two Cards, as detected by the GUI. The *Player* notifies the *Server* of the decision and prompts the GUI to draw a *Card* twice.

Draw Card

Operation: Player:: promptDrawCard()
Scope: Player, Game, GameBoard, Card, GameEdition;
Messages: GUI::{promptDrawCard}
Post: The *promptDrawCard* operation prompts the *Player* to draw a card either from the *GameBoard*'s orderedFaceDownCardPile or its faceUpCardPileWithoutGoldCards. In turn, the *Player* prompts the GUI to draw a Card.

Operation: Player:: replyDrawFaceUpCard(card: Card)
Scope: Player, GameBoard, Card;
Messages: ElfenRoads::{replyDrawFaceUpCard(card:Card)}
Post: The *replyDrawFaceUpCard* operation notifies the *Player* of the drawn Card from the faceUpCardPileWithoutGoldCards, detected through the GUI. In turn, the *Player* notifies the Server.

Operation: Player:: replyDrawFaceDownCard()
Scope: Player, GameBoard, Card;
Messages: ElfenRoads::{replyDrawFaceDownCard}
Post: The *replyDrawFaceDownCard* operation notifies the *Player* of the GUI-detected decision to draw a Card from the *orderedFaceDownCardPile*. In turn, the *Player* notifies the Server.

Operation: Player:: promptDrawCardOrGoldStack()
Scope: Player, GameBoard, Card, GoldCard, GameEdition;
Messages: GUI::{promptDrawCardOrGoldStack}
Pre: The current *GameEdition* is ElfenGold and the *replyDrawFaceDownCard* message from the *Player* to the *Server* has chosen a GoldCard from the *orderedFaceDownCardPile*.
Post: The *promptDrawCard* operation prompts the *Player* to draw a card either from the *GameBoard*'s orderedFaceDownCardPile or its faceUpCardPileWithoutGoldCards, or to collect the gold *Cards* in the *goldCardPile*. In turn, the *Player* prompts the GUI with the choice.

Operation: Player:: replyCollectGoldStack()
Scope: Player, Game, GameBoard, GoldCard, GameEdition;
Messages: ElfenRoads::{replyCollectGoldStack()}
Pre: *Player* has received *promptDrawCardorGoldStack* and relayed the choice to the GUI.
Post: The *replyCollectGoldStack* operation notifies the *Player* the choice to collect the goldCardPile. In turn, the *Player* notifies the Server.

Draw Token (Face-Up or Face-Down)

Operation: Player:: promptDrawToken()
Scope: Player, GameBoard, Token, GameEdition;
Messages: GUI::{promptDrawToken}
Pre: The *GameBoard* has a *myOrderedTokensFaceDownPile* and a *myOrderedTokensFaceUpPile* from which the *Player* can draw *Tokens* from. The current *GameEdition* is ElfenLand.
Post: The *promptDrawToken* operation prompts the *Player* to either draw a *Token* from the *GameBoard's myOrderedTokensFaceUpPile* or *myOrderedTokensFaceDownPile*. In turn, the *Player* prompts the GUI to either choose to draw a *Token* from *myOrderedTokensFaceUpPile* or *myOrderedTokensFaceDownPile*.

Operation: Player:: replyDrawFaceUpToken(token: Token)
Scope: Player, GameBoard, Token;
Messages: ElfenRoads::{replyDrawFaceUpToken}
Post: The *replyDrawFaceUpToken* operation notifies the *Player* that the *User* chose to draw a *Token* from the *myOrderedTokensFaceUpPile*. In turn, the *Player* notifies the *Server*.

Operation: Player:: replyDrawFaceDownToken()
Scope: Player, GameBoard, Token;
Messages: GUI::{replyDrawFaceDownToken}
Post: The *replyDrawFaceDownToken* operation notifies the *Player* that the *User* chose to draw a *Token* from the *myOrderedTokensFaceDownPile*. In turn, the *Player* notifies the *Server*.

Draw Token (Only Face-Down)

Operation: Player:: promptDrawFaceDownToken()
Scope: Player, GameBoard, Token;
Messages: GUI::{promptDrawFaceDownToken}
Pre: The *GameBoard* has a myOrderedTokensFaceDownPile from which the *Player* can draw *Tokens*.
Post: The *promptDrawToken* operation prompts the *Player* to draw a *Token* from the *GameBoard'*s *myOrderedTokensDownUpPile*. In turn, the *Player* prompts the GUI to draw a Token from *myOrderedTokensFaceDownPile*.

Hide Tokens

Operation: Player:: promptHideTokens(numTokensToHide: Integer)
Scope: Player, Token, GameBoard;
Messages: GUI::{promptHideTokens(numTokensToHide: Integer}
Pre: The *Player* must have more than numTokensToHide in hand. Otherwise, all *Tokens* will be hidden.
Post: The *promptHideTokens* operation prompts the *Player* to choose from their tokens in hand numTokensToHide *Tokens* to hide from other *Players*. In turn, the *Player* prompts the GUI to select a set of *Tokens* to hide.

Operation: Player:: replyHideTokens(tokens:Set{Token})
Scope: Player, Token;
Messages: ElfenRoads::{replyHideTokens}
Post: The *replyHideTokens* operation notifies the *Player* of the selected *Tokens* to hide from other *Players*. In turn, the *Player* informs the *Server* of the decision.

Place Token

Operation: Player:: promptPlaceToken()
Scope: Player, Token, Road;
Messages: GUI::{promptPlaceToken}
Pre: The *Player* must have at least one *Token* in hand.
Post: The *promptPlaceToken* operation prompts the *Player* to place a *Token* from *myTokensInHand* on a *Road*, which in turn prompts the GUI to complete this action.

Operation: Player:: replyPlaceToken(e: Edge, token: Token)
Scope: Player, Token, Edge, Road, RiversAndLakes, GameBoard, MagicSpell;
Messages: ElfenRoads::{replyPlaceToken}, GUI::{InvalidAction, promptPlaceToken}
Post: The *replyPlaceToken* operation notifies the *Player* of the GUI detected choice of *Token* and *Road*.

- If the placed Token is an Exchange *MagicSpell* and the Edge is a Road:
    - If there is at least one *TransportationCounter* on another Road of the GameBoard: the *Player* notifies the Server that an ExchangeToken was used on the Road.
    - Otherwise, the *Player* notifies the GUI that the move is invalid and prompts the GUI to place a *Token*.
    - If the *Player* has at least one *TransportationCounter* left in hand: the *Player* notifies the Server that a DoubleTransport was placed on the Road.
    - Otherwise, the *Player* notifies the GUI that the move is invalid and prompts the GUI to place a *Token*.
- Else if the *Token* cannot be placed on that *Road*, the *Player* notifies the GUI that the move is invalid and prompts the GUI to place a *Token* again.
- Else, the *Player* notifies the *Server* of this move.

Operation: Player:: promptExchangeToken()
Scope: Player, TransportationCounter, Road;
Messages: GUI::{promptExchangeToken}
Post: The *promptExchangeToken* operation prompts the *Player* to exchange the Transportation Counter that is on the Road with a Transportation Counter on another Road, which in turn prompts the GUI to complete this action.

Operation: Player:: replyExchangeToken(token:Token)
Scope: Player, TransportationCounter, Road;
Messages: Elfenroads::{replyExchangeToken}, GUI::{invalidAction, promptExchangeToken}
Post: The *replyExchangeToken* operation notifies the Player of the Token the User wants to Exchange. The chosen Token must be a valid Token that has been placed on another road, or else the GUI is notified to display an InvalidAction message and prompt to choose a Token to exchange. Otherwise, the Server is notified of the decision.

Operation: Player:: promptExtraTokenForRoad()
Scope: Player, TransportationCounter, Road;
Messages: GUI::{promptExtraTokenForRoad}
Post: The *promptExtraTokenForRoad* operation prompts the *Player* to add an extra Transportation Counter, which in turn prompts the GUI to complete this action.

Operation: Player:: replyExtraTokenForRoad(token:Token)
Scope: Player, TransportationCounter, Road,Token;
Messages: Elfenroads::{replyExtraTokenForRoad}, GUI::{invalidAction, promptExtraTokenForRoad}
Post: The *replyExtraTokenForRoad* operation notifies the Player of the Token the User wants to add. The extra Token from the player's hand should not be the same type as the first Token placed on the road, or else the GUI is notified to display an invalidAction message and prompt to choose an extra token for the road. Otherwise, the Server is notified of the decision.

Choose Cards to Keep

Operation: Player:: promptChooseCardsToKeep(numCardsToKeep: Integer)
Scope: Player, Card, GameBoard;
Messages: ElfenRoads::{replyChooseCardsToKeep(cardsToKeep: Set{Card})},
GUI:{promptChooseCardsToKeep}
Post: The *promptChooseCardsToKeep* operation prompts the Player to choose numCardsToKeep travel cards to keep, which in turn prompts the GUI to perform this action.

- If the *Player* has less than numCardsToKeep travel cards, then this step is skipped and cardsInHand is returned in *replyChooseCardsToKeep*.

Operation: Player:: replyChooseCardsToKeep(cardsToKeep: Set{Card})
Scope: Player, CardDeck, GameBoard;
Messages: ElfenRoads::{replyChooseCardsToKeep}
Post: The replyChooseCardsToKeep operation informs the Player of the chosen travel cards to keep, which in turn informs the Server.

Choose Tokens to Keep

Operation: ElfenRoads:: promptChooseTokensToKeep(numTokensToKeep: Integer)
Scope: Player, Token, GameBoard;
Messages: ElfenRoads::{replyChooseTokensToKeep(tokens)}, GUI:{promptChooseTokensToKeep}
Post: The *promptChooseTokensToKeep* operation prompts the Player to choose tokens to keep, which in turn prompts the GUI to perform this action.

- If the *Player* has less than numTokensToKeep tokens, then this step is skipped and myTokensInHand is returned in *replyChooseTokensToKeep*.

Operation: Player:: replyChooseTokensToKeep(tokens: Set{Token})
Scope: Player, Token, GameBoard;
Messages: ElfenRoads::{replyChooseTokensToKeep}
Post: The replyChooseTokensToKeep operation informs the Player of the chosen tokens to keep, which in turn informs the Server.

Game Ending

Operation: Player:: gameInstanceDeleted()
Scope: Player, Game;
Messages: LobbyService::{retrieveGameServices, retrieveGameSessions}, GUI::{gameInstanceDeleted, displayLobbyScreen}
Post: *Player* has been notified that the game instance has been deleted. *Player* notifies the GUI to display this information, along with the lobby screen with the latest information from the LobbyService.

Operation: Player:: gameOverMessage(winner: Player)
Scope: Player, Game, TownCard, Town;
Messages: LobbyService::{retrieveGameServices, retrieveGameSessions},
GUI::{gameOverMessage(winner: Player), displayLobbyScreen}
Post: *Player* has been notified that the game is over and who the winner is. If the GameVariant TOWN_CARDS is being played, the *Server* decides on the winner based on the Players' number of points minus the number of Towns they are away from their *TownCard*. *Player* notifies the GUI to display this information, along with the lobby screen with the latest information from the LobbyService.

Operation: Player:: playerQuitProtocol(p: Player)
Scope: Player, Game;
Messages: LobbyService::{retrieveGameServices, retrieveGameSessions},
GUI::{playerHasQuitMessage(p: Player), displayLobbyScreen}
Post: *Player* has been notified that the game instance is over because a *Player* has quit. *Player* notifies the GUI to display this information, including which *Player* quit, along with the lobby screen with the latest information from the LobbyService.

Log in and Lobby Screen Set Up

Operation:Player::logIn(username, password)
Scope:
Messages: LobbyService::{authenticateLogIn(username, password), retrieveGameServices, retrieveGameSessions}, GUI::{displayLobbyScreen, invalidAction(m:Message)}
Pre: Player has sent *displayLogInScreen()* to GUI
Post: Player sends the incoming log in credentials to the LobbyService in order to validate the log in attempt and receive an accessToken. Upon successfully receiving the accessToken, the Player will retrieve the available Game Services and Game Sessions from the LobbyService and display them to the GUI. If the authentication is unsuccessful, the GUI will be notified to display the invalid action message.

Operation: Player::playerAuthenticationToken(accessToken)
Scope: Player
Messages: LobbyService::{refreshAuthentication(accessToken)}
Pre: Player must have sent a successful *authenticateLogIn* operation or time-triggered *refreshAuthentication*
Post: The LobbyService-generated accessToken is stored so that the Player can verify their identity when communicating with other actors. A countdown begins to send a time-triggered *refreshAuthentication* message when the accessToken will time out.

Operation: Player::returnGameServices(List<GameServices>)
Scope: Player, LobbyService;
Pre: Player has sent a *retrieveGameServices()* request to the LobbyService
Post: This updates the available GameServices that will be shown when the *displayLobbyScreen()* message is sent to the GUI.

Operation: Player::returnGameSessions(List<GameSessions>)
Scope: Player, LobbyService;
Pre: Player has sent a *retrieveGameSessions()* request to the LobbyService
Post: This updates the available GameSessions that will be shown when the *displayLobbyScreen()* message is sent to the GUI.

Operation: Player::refreshLobbyScreen()
Scope: Player, LobbyService
Messages: Lobby::{retrieveGameServices(), retrieveGameSessions()}, GUI::{displayLobbyScreen()}
Post: Player updates the information on the Lobby Screen by retrieving the available Game Services and Game Sessions from the LobbyService and displaying them to the GUI.

Log Out

Operation: Player::logOut()
Scope: Player, LobbyService
Messages: LobbyService::{logOut(accessToken)}, GUI::{displayLogInScreen}
Pre: Player must have valid accessToken
Post: The Player is notified from the GUI to log out and sends a message to the Lobby to revoke the accessToken.

Lobby

Operation: Player::retrieveSaveGames()
Scope: Player, LobbyService
Messages: LobbyService::{retrieveSaveGames(GameService)}, GUI::{returnSaveGames()}
Pre: *Player* has previously retrieved the GameServices registered with the LobbyService.
Post: The *Player* is notified from the GUI to retrieve the saved games of the authorized user. Player retrieves the user's savegames of all GameServices registered with the LobbyService and displays them to the GUI.

Operation: Player::returnSaveGames(List<SaveGames>)
Scope: Player, LobbyService
Pre: Player has sent a *retrieveSaveGames(GameService)* request to the LobbyService
Post: This updates the available SaveGames that will be shown when the *displaySaveGames* message is sent to the GUI.

Operation: Player::createNewGameSession(GameService)
Scope: Player, LobbyService
Messages: LobbyService::{createNewGameSession(Player,GameService), retrieveGameSessions(), retrieveGameServices(), retrieveSessionInformation(sessionID)}, GUI::{displayLobbyScreen()}
Post: The Player is notified from the GUI the choice to create a new game instance for a particular Game Service and notifies the LobbyService to do so. Player retrieves updated lobby information and notifies the GUI to display it. It also sets a clock on a time triggered output message to update the session information regularly.

Operation: Player::loadGameSession(GameService, saveGameID)
Scope: Player, LobbyService
Messages: LobbyService::{loadGameSession(Player, GameService, saveGameID), retrieveGameSessions(), retrieveGameServices(), retrieveSessionInformation(sessionID)}, GUI::{displayLobbyScreen()}
Post: The Player is notified from the GUI the choice to load a game from a previously saved instance and notifies the LobbyService to do so. Player retrieves updated lobby information and notifies the GUI to display it. It also sets a clock on a time triggered output message to update the session information regularly.

Operation: Player::launchGameSession()
Scope: Player, LobbyService
Messages: LobbyService::{retrieveSessionInformation(sessionID), launchGameSession(sessionID)}, GUI::{invalidAction(m: Message)}
Pre: Player must be the creator of the game session.
Post: The GUI informs the Player that they are ready to launch the session and the Player confirms with the updated session information that the minimum number of required players have joined. If not, an invalidAction message is sent to the GUI. Otherwise, a message is sent to the LobbyService to launch the game.

Operation: Player::deleteUnlaunchedGameSession(sessionID)
Scope: Player, LobbyService
Messages: LobbyService::{deleteUnlaunhcedGameSession(sessionID), retrieveGameSessions(), retrieveGameServices(), retrieveSessionInformation(sessionID)}, GUI::{gameSessionCancelledMessage(), displayLobbyScreen()}
Pre: Player must be the creator of the game session and game session must not have been launched.
Post: The GUI informs the Player to delete a previously saved game session. The Player notifies the LobbyService to do so and notifies the GUI to display *gameSessionCancelledMessage*. It updates the lobby information and informs the GUI to display the updated lobby.

Operation: Player::joinGameSession(sessionID)
Scope: Player, LobbyService
Messages: LobbyService::{joinGameSession(sessionID), retrieveGameSessions(), retrieveGameServices(), retrieveSessionInformation(sessionID)}, GUI::{displayLobbyScreen(), invalidAction(m:Message)}
Pre: Player must not have already joined another session.
Post: The GUI informs the Player of the decision to join an available game session and Player notifies the LobbyService. If unable to join, the Player notifies the GUI to display an invalidAction message. In either case, Player informs the GUI to display updated lobby screen information. It also sets a clock on time triggered output messages to update the session information regularly.

Operation: Player::returnSessionInformation(SessionInfo)
Scope: Player, LobbyService
Messages: GUI::{newPlayerJoinedSession(Player), gameSessionCancelledMessage()}
Pre: Player must have joined the game session
Post: Player compares updated Session Information with previously stored information. If there are any updates (a player has joined or the game session has been cancelled), it will inform the GUI to display the information.

Operation: Player::cancelJoinGameSession(sessionID)
Scope: Player, LobbyService
Messages: LobbyService::{cancelJoinGameSession(), retrieveGameSessions(), retrieveGameServices()}, GUI::{displayLobbyScreen()}
Pre: Player must have already joined a session.
Post: The GUI informs the Player of the decision to cancel joining an available game session and Player notifies the LobbyService. Player informs the GUI to display updated lobby screen information.

Operation: Player::deleteSaveGame(GameIdentifier)
Scope: Player, LobbyService, Game
Messages: ElfenRoads::{deleteSaveGame(GameIdentifier)
Post: Player notifies Server to delete all information on a saved game.

Operation: Player::refreshLobby()
Scope: Player, LobbyService
Messages: LobbyService::{retrieveGameSessions(), retrieveGameServices()}, GUI::{displayLobbyScreen()}
Post: Player retrieves updated information from the LobbyService and informs GUI to display it.

Operation: Player::quitGame()
Scope: Player, Game, LobbyService
Messages: ElfenRoads::{quitGame()}
Pre: Player must have joined a game and the game has been launched.
Post: The Player notifies the Server of the choice to quit the game, as detected through the GUI.