# COMP 302

## Final Project Report

**28/12/2018**

## **Project:** Ultimate Monopoly Game

## **Group:** Euler's Disciples

**Team Members:** Arda Öztaşkın

Bulut Bulgu

Cem Ege Saygılı

Duygu Sezen Islakoğlu

Hazal Mengüaslan

Mert Lüleci

# Table of Contents

# Introduction

This final report documents our vision, all of our deliverables, progress, teamwork and any other information related to the project. In the project, we were asked to implement the Ultimate Monopoly Board Game, albeit with some differences as it is an online game. We worked on this game for more than three months, September to December. Through the project, we managed to work as a team and overcome difficulties as a unit even though we had our differences. Overall, it was a very fun project to implement and we learned many new techniques while working on it.

# Vision

We envision a next generation Ultimate Monopoly Game, with the flexibility to support varying computers, different amounts of players, different types of bots to create an authentic Monopoly experience whenever you play on whatever machine you want. Our objective is to make a Monopoly game that is suitable for all ages and at the same time be intuitive and easy-to-learn.

## Problem Statement

In the realm of board games, the normal Monopoly game is infamous for lasting hours upon hours all the while the enjoyment factor goes lower the longer it goes. Although there are different editions of Monopoly, none of those solve these particular problems. Ultimate Monopoly Game, as the name states, combine different elements from many different Monopoly editions to create the ultimate Monopoly experience. But this creates yet another problem; to play the Ultimate Monopoly Game, you need pieces and boards from many different editions of Monopoly, some of which are discontinued as of this writing which make playing this ultimate version pretty much impossible.

## Stakeholder Descriptions

Because of the fact that this is only a game meant to be played for fun, the only stakeholders are the people who are looking to have fun while sparking a rivalry between close friends which hopefully does not go beyond the game.

## Key High-Level Goals

Our key goals include having visible animations for many of the actions happening in the game so that you do not miss anything if you look away for a given time. Having network play and saving/loading is also crucial for our project because we want to transcend the Monopoly game from a mediocre board game into a perfect online game. One other goal of this project is to make this game playable with bots, so that you never miss the joy of Monopoly even when you are alone.

## Summary of System Features

The major features include:

### 1. User Interface

a. Buttons such as roll dice, end turn, buy tile.
b. Board with 120 tiles.
c. Information of all players including their name, money, location, assets.

### 2. Animation

a. Each player’s piece moves smoothly.
b. Rolling the dice has a fancy animation.

### 3. Network

a. Game can be played with 8 players from different computers.

4. **Save-Load**
    a. Players can save the current game and load them to continue.

5. **Pause-Resume**
    a. Players have the ability to pause both the animation and the game with the Pause button. Game continues as normal when the Resume button is pressed.

6. **Bot**
    a. The game can have bots which makes decisions according to their strategy.

## Importance of the Project

Through this project, we had the opportunity to study many differents of topics discussed within the COMP 302 course. We worked together as a group and collaborated so the project also gave us a chance to do some group work like the ones we will encounter later in our worklife. We also learned how to increase the readability and adaptability of our code which was a really nice bonus.

## Our criteria to success

- To meet all the requirements that the game has.
- To have efficient and bug-free code.
- To have an implementation that is easy to understand.
- To work on the project with fairly divided workload.

# Teamwork Organization & Workload Share

Through the development, we used Git to share our files fast and efficiently so that we do not lose time on pointless work. Having everyone work on different Git branches at the same time made it especially easy for us to divide the work. We decided it was best to divide the team members into three groups; Domain, GUI, Network. Ege, Hazal and Mert worked on the domain package. Ege mainly worked on tile logic, game logic and constructed the TileJSON in domain. Mert handled the card logic, worked on game logic and constructed rollthreeJSON. Hazal worked on every logic inside domain prioritizing the game logic. Bulut and Duygu worked on GUI. Bulut helped implementing the piece configurations, storage logic whereas Duygu handled the Observer and Controller implementations and the UI panels. Arda started working on network and after finishing the network he started helping the domain package and GUI, working on game logic and storage implementation along with legal actions implementation and singleton implementations. Below are the lists of all the branches we used corresponding to the members who worked on them:

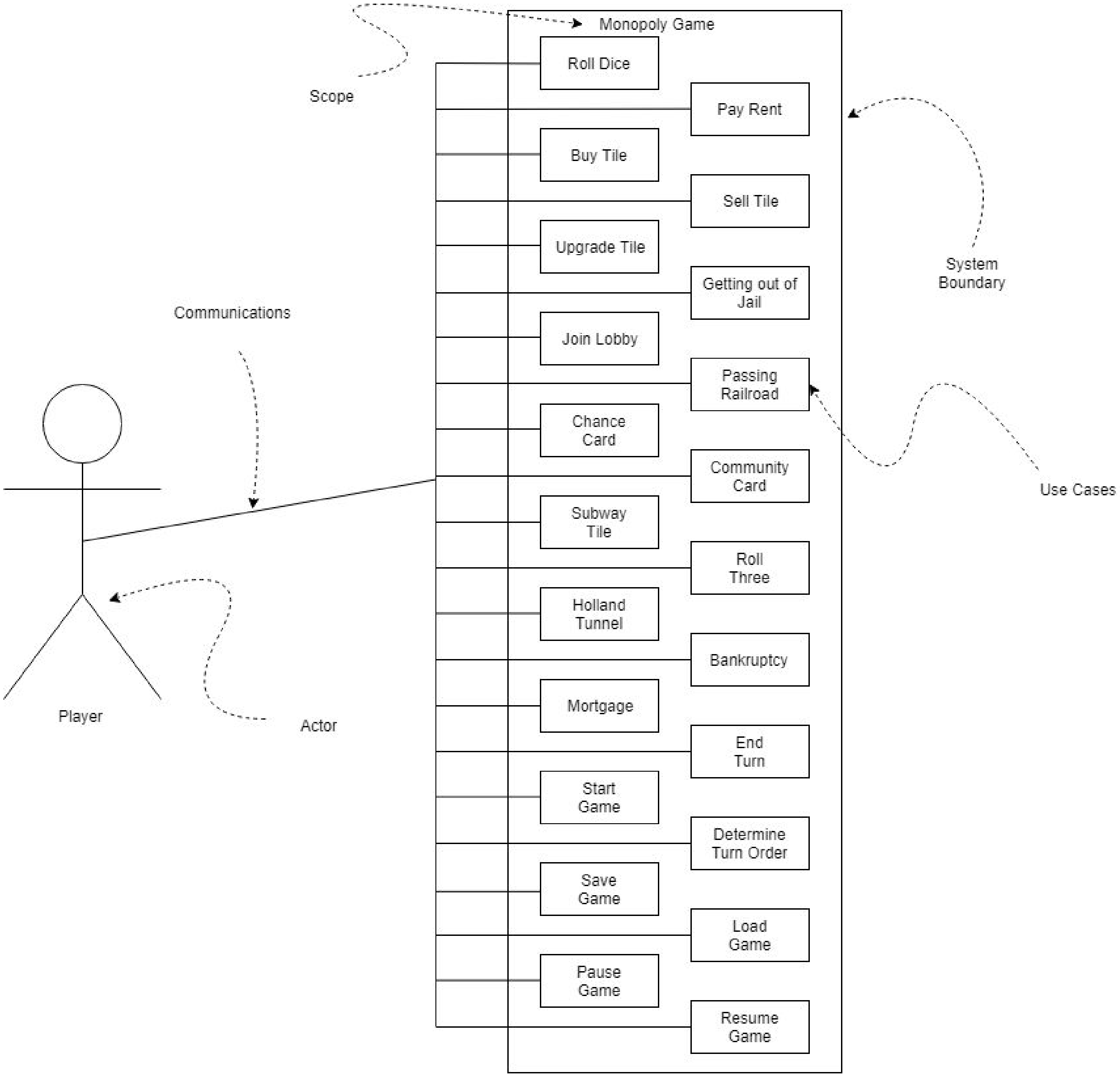
**Arda Öztaşkın** worked on the following branches: NetworkImprovement, Actions, GameLogic,Animation, SaveLoadTest, LegalActions, RollDiceLogic, StorageImplementation. **Hazal Mengüaslan** worked on the following branches: Board, Tile, Actions, TileLogic, GameLogic, LegalActions, CardObserver, RailroadMethods, BirthdayGift, Player, TileInformation, TileInformationFixed, RailroadInfo. **Cem Ege Saygılı** worked on the following branches: Board, Tile, TileLogic, GameLogic, TileInformation, TileInformationFixed, TileJSON, RailroadInfo. **Duygu Sezen Islakoğlu** worked on the following branches: PlayerPanel, RightPanel, Card, newButtonPanel, Images, Observer, CardObserver, Player. **Kadri Mert Lüleci** worked on the following branches: CardLogic, TileLogic, Card, GameLogic, RollThree, Speed. **Bulut Bulgu** worked on the following branches: SaveLoadTest, Animation, GameWindow, Images, RightPanel, NewButtonPanel.

# Deliverables

This section contains all the diagrams we made while developing our project. These diagrams include system sequence diagrams, communication diagrams, domain models among many other diagrams. After initially making diagrams in the Phase 1 of the project, we revised many of the old diagrams when we reached Phase 2 because many things changed between the two phases. The texts and diagrams documented are as follows:

- *Use Case Diagram*
- *Use Case Narratives*
- *System Sequence Diagrams*
- *Operation Contracts*
- *Package Diagram*
- *Sequence Diagrams*
- *Communication Diagrams*
- *Domain Model*
- *Class Diagram*

# Use Case Diagram



# 1) Use Case Narratives

When selecting our use case narratives, we opted in to using the narratives that are occured the most in the game, such as rolling the dice or buying a tile. This also gave us an opportunity to showcase how long narratives are made and different, various extensions are shown in the use cases. After initially making the narratives early in the project, we made some fixes depending on the feedback that we got and added some new narratives corresponding to the new features we added in the Phase 2.

# USE CASE #1: ROLLING DICE

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to see the result of dice

Other Players: Wants to see the what the current player has rolled.

Network Service: Wants to receive the results of dice to distribute.

**Preconditions:** Player has network connection, is eligible to roll dice.

**Postconditions:** Dice result is saved and displayed to players.

**Main Success Scenario:**

1. Current player presses the roll dice button.
2. Results are displayed on all players.

**Extensions** (or alternative flows):

1a. Player does not roll for designated time. *

1. Dice is rolled automatically and results are displayed as usual.

2a. Some players have spotty network connection.

1. Wait for player to come back.
2. Player come back online.

    2a. Player does not come back online in time.

    1. Disconnected player protocol commences.
3. Results are displayed on their board.

**Special Requirements:**

- Dice animation

**Frequency of Occurrence:**

- Almost every turn at least once.

# USE CASE #2: PAY RENT

**Scope:** Euler’s Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to pay rent

Receiver: Receives rent and has it displayed on screen.

Other Players: Wants to see the money changes

System: Wants to update money state

**Preconditions:** Player has network connection, player has landed on a foreign, non-mortgaged tile.

**Postconditions:** Money state is updated displayed to players.

**Main Success Scenario:**

1. Player lands on a foreign tile, has to pay rent depending on the number of buildings on that tile and the monopoly status.
2. Money transfer happens between two players.
3. Final states are displayed to players.

**Extensions** (or alternative flows):

2a. Player doesn’t have enough money to pay.

1. Player sells assets.

   1a. Player can’t sell enough asset to get enough money.

   1. Player bankrupts.
   2. Bankrupt protocol is commenced.
2. Player pays rent.
3. Final states are displayed to players.

**Frequency of Occurrence:**

- Every few turns.

## USE CASE #3: BUY TILE

**Scope:** Euler’s Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to buy tile

Other Players: Wants to see the money and asset list changes

System: Wants to update money state, asset list

**Preconditions:** Player has network connection, player has landed on a unowned tile.

**Postconditions:** Money state and asset list are updated and displayed to players.

**Main Success Scenario:**

1. Player lands on a buyable tile, player wants to buy that tile.
2. Player pays money to bank.
3. Final states are displayed to players.

**Extensions** (or alternative flows):

1a. Player does not have enough money.

1. The system does not allow the player to buy the tile.
2. The tile goes on an auction.

1b. Player does not want to buy the tile.

1. The tile goes on an auction.

# USE CASE #4: JOIN LOBBY

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to join the game

Other Players: Wants to see the lobby list change

System: Wants to update the user list

**Preconditions:** Player has network connection, the game has not started yet.

**Postconditions:** User list is updated.

**Main Success Scenario:**

1. Player pushes the button to join the online lobby.
2. Player joins the lobby.
3. Other users see the updated user list in the lobby.

**Extensions** (or alternative flows):

3a. Some players have bad network connection

1. Those player reconnect to the server.
2. Their user lists are updated.

# USE CASE #5: UPGRADE TILE

**Scope:** Euler’s Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to upgrade the tile

Other Players: Wants to see the changes applied to that tile

System: Wants to update the tiles status

**Preconditions:** Player has network connection, player has the turn.

**Postconditions:** Player’s money changes. Changes are shown on the board.

**Main Success Scenario:**

1. Player wants to upgrade its properties by buildings.
2. Player upgrades the tile.
3. Upgrade is shown on the board.

**Extensions** (or alternative flows):

2a. Player does not have enough money.

1. Player cannot upgrade the tile and is shown an error message.

2b. Hotel/house quota is reached.

1. Player cannot upgrade the tile and is shown an error message.

2c. Player doesn’t own the majority of that colored tile.

1. Player cannot upgrade the tile and is shown an error message.

**Special Requirements:**

- Building animation

**Frequency of Occurrence:**

- Every few turns.

## USE CASE #6: GETTING OUT OF JAIL

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Jailed

System: Wants to update the player's being jailed status

**Preconditions:** Player has network connection, player has the turn, the player is jailed.

**Postconditions:** Player gets out of jail.

**Main Success Scenario:**

1. Player rolls double.
2. Player gets out of jail.
3. Player's jailed status is updated.

**Extensions** (or alternative flows):

1a. Player has a get out of jail card

1. Player uses the card to get out of jail.
2. Player's being jailed status is updated.

1b. Players 3rd turn

1. Player pays to get out.

    1a. Player does not have enough money.

    1. Player sells assets

        1a. Player cannot sell assets.

        1. Player bankrupts.

    2. Player gets out of jail.

2. Player's being jailed status is updated.

1c. Player does not roll double

1. Player stays put.

# USE CASE #7: SELL BUILDING

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to sell a building and see the change in the game state

Other Players: Wants to see the building and game state change

System: Wants to update the game state correctly

**Preconditions:** Player has network connection, the tile the player is trying to sell the buildings on has a building on it.

**Postconditions:** Money and the board state is changed. The UI is updated accordingly.

**Main Success Scenario:**

1. Player tries to sell a building.
2. Player sells the building to the system.
3. Player receives money from the other player or bank, tile downgrades.

**Extensions** (or alternative flows):

2a. Player can't sell due to the even rule

1. Error pop-up appears on player's device

2b. Bank does not have enough house/hotels

1. Error pop-up appears on player's device

# USE CASE #8: PASSING RAILROAD

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Players: Wants to obey the game rules

System: Wants to update the player's current place

**Preconditions:** Player has network connection, player has the turn and player moves through a  railroad.

**Postconditions:** Player's lane changes.

**Main Success Scenario:**

1. Player moves through railroad/transit station.
2. Player's roll was even and changes lane.

**Extensions** (or alternative flows):

2a. Player's roll was odd.

1. Player does not change lane and continues on its lane.

# USE CASE #9: CHEST/CHANCE CARD

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to draw a card

System: Wants to update the card list

**Preconditions:** Player has network connection, player has the turn and player lands on a chest/chance card tile.

**Postconditions:** Cards are updated.

**Main Success Scenario:**

1. Player lands on a community chest or chance card tile.
2. Player draws a random card from the list.
3. Cards effects are applied.

**Extensions** (or alternative flows):

3a. Player decides to keep the card to use it later (if that option is available).

1. The card goes into the card list of the player.

# USE CASE #10: SUBWAY TILE

**Scope:** Euler’s Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to use the subway tile.

System: Wants to synchronize the player position for all players.

**Preconditions:** Player has network connection, player has the turn and player lands on a subway tile.

**Postconditions:** Player’s current tile changes.

**Main Success Scenario:**

1. Player lands on a subway tile.
2. In the next turn player chooses any tile they want to go, does not roll dice.
3. Player moves to the tile.

**Extensions** (or alternative flows):

2a. Player has additional dice rolls in the turn.

1. Instead of rolling a dice, player choses a tile to go to.
2. Player moves to that tile.

# USE CASE #11: ROLL THREE

**Scope:** Euler's Monopoly

**Level:** User goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to draw a Roll Three card, roll the appropriate dice and have the results displayed

Other Players: Wants to see the rolled dice and have the changes displayed

System: Wants to update

**Preconditions:** Player has network connection, player has the turn and player lands on a Roll Three tile.

**Postconditions:** Players earn money depending on their card.

**Main Success Scenario:**

1. Player lands on a Roll Three tile.
2. Draws a Roll Three card.
3. Player rolls the dice.
4. Players with matching cards earn money.

**Extensions** (or alternative flows):

4a. No one has matching cards.

1. No one earns money.

4b. The current player has a matching card.

1. The current player earns more money than the other players with matching cards.

# USE CASE #12: SELL TILE

**Scope:** Euler's Monopoly

**Level:** User goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to sell tile

Other Players: Wants to see the money and asset list changes

System: Wants to update money state, asset list

**Preconditions:** Player has network connection.

**Postconditions:** Money state and asset list are updated and displayed to players.

**Main Success Scenario:**

1. Player presses a button to sell a tile.
2. The tile is sold.
3. Player receives money from the one bought that tile.

**Extensions** (or alternative flows):

1a. There are buildings on the tile.

1. Error message pop-up is displayed.

1b. Loses majority on upgraded tiles if the tile is sold.

1. Error message pop-up is displayed.

1c. Tile is mortgaged.

1. Error message pop-up is displayed.

# USE CASE #13: HOLLAND TUNNEL

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Players: Wants the rules enforced

Network service: Wants to see the current positions of players

**Preconditions:** Player has network connection, player has the turn and player lands on a Holland Tunnel.

**Postconditions:** Player's current position changes.

**Main Success Scenario:**

1. Character lands on Holland tunnel.
2. Character moves to the other Holland Tunnel.

**Special requirements:**

- Animated movement of player.

# USE CASE #14: BANKRUPTCY

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Players: Want the rules enforced

Network service: Wants to see the current positions of players

**Preconditions:** Player doesn't have money enough money to pay.

**Postconditions:** Player list is updated. Player's assets are given to the bank.

**Main Success Scenario:**

1. Player has to pay money for whatever reason.
2. Player does not have enough money.
3. Player bankrupts and drops out of the game.

**Extensions** (or alternative flows):

2a. Player has tiles to mortgage or sell.

1. Player sells tiles until he/she has enough money to pay the rent.

**Special requirements:**

- Animated movement of player

# USE CASE #15: MORTGAGE

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Players: Wants to mortgage the tile which he/she owns.

System: Wants to update money state

**Preconditions:** Owning the tile.

**Postconditions:** Player receives half the of the price of the tile.

**Main Success Scenario:**

1. Player wants to mortgage a tile.
2. The tile is mortgaged.
3. Player receives half the price of the tile from the bank.

# USE CASE #16: END TURN

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Players: Wants to see the

System: Wants to update the current player

**Preconditions:** Player has network connection, player has come to the end of their turn.

**Postconditions:** The turn passes to the next player.

**Main Success Scenario:**

1. Player has done all he/she wanted and decides to end their turn.
2. The current player changes.

# USE CASE #17: START GAME

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to start the game

Other Players: Wants to have their game started as well

System: Wants to create the game, notify and synchronize all the players

**Preconditions:** All the players have added themselves on the lobby to the player list.
All the players have network connection.

**Postconditions:** All the players are initialized, the game is ready to determine the turn order.

**Main Success Scenario:**

1. The player hosting the game presses the start game button.
2. The server initializes the game.
3. The server notifies all the players that the game has started.
4. The players have their game initialized and ready to be played.

**Extensions** (or alternative flows):

3a. Some players doesn't receive the notification due to network errors.

1. The server starts timeout countdown for each disconnected player.
2. The server sends the notification again until the timeout.

   2a. The player doesn't receive the notification until the timeout occurs.

   1. The server timeouts the player and notifies other players about the timeout.
   2. The players initialize the game without the disconnected players.

# USE CASE #18: DETERMINE TURN ORDER

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Players: Wants to roll to have their turn order determined

System: Wants to create and synchronize the turn order between the players

**Preconditions:** The game has started without any problems

**Postconditions:** The turn order is set and players are notified about it.

**Main Success Scenario:**

1. The server randomly generates a temporary turn order for players to roll at first
2. The server sends the turn order to the players.
3. The players receive the turn order.
4. Each player rolls on their turn.
5. Every players turn ends automatically after rolling the dice and the turn order is determined.

**Extensions** (or alternative flows):

3a. The player does not receive the turn order due to network error.

1. The server re-sends the turn order.
2. The server starts the timeout countdown.
3. The player receives the message before it times out.

   3a. The player times out

4a. The player does not roll in the determined time.

1. The player automatically throws the dice.

7a. Some players do not receive the message sent by the server.

## USE CASE #19: SAVING THE GAME

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to save the game.

System: Successfully saving the game.

**Preconditions:** The game must be paused.

**Postconditions:** The game state is saved in a file which can be used in order to start the game from the same place.

**Main Success Scenario:**

1. Player presses the save game button.
2. System shows a pop-up window to player asking the name of the file to be saved.
3. Player confirms the name by pressing a button.
4. System saves the game state into a file.

**Extensions** (or alternative flows):

2a. The file name is invalid

1. System asks for the name of the file again until a valid name is entered.
2. Player confirms the name by pressing a button.
3. System saves the game state into a file.

4a. The game is cannot be saved due to an issue in the system.

1. System shows an error pop-up to the user.

**Frequency of Occurrence:**

- Whenever a user wants to save the game.

# USE CASE #20: LOADING THE GAME

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Player: Wants to load the game.

Other Players: Want the host to load the game.

System: Load a previously saved game.

**Preconditions:** Player has entered the correct communication IP and port, and joined the lobby. There is a save file the host wants to load from.

**Postconditions:** The game is loaded from a save file.

**Main Success Scenario:**

1. Player presses the load game button in the lobby window.
2. System shows a prompt to the player, asking to select the appropriate save file.
3. System starts the game from the save file.
4. Player waits for other players to join so that the game can start.
5. Player presses the start game button and the system starts the game.

**Extensions** (or alternative flows):

2a. The file is invalid

1. The system asks the player for a file again until a valid file is selected.
2. After getting an appropriate file, scenario moves on.

2b. File cannot be loaded due to an issue.

1. The system shows an error pop-up to player.
2. The system asks the player to select the appropriate save file.

4a. One or more clients from the original save is missing.

1. The system assigns the players assigned to the missing clients to a predetermined client.

## USE CASE #21: PAUSING THE GAME

**Scope:** Euler's Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to pause the game.

Other Players: Want their game to be paused as well.

System: Wants to successfully pause the game.

**Preconditions:** The turn must be the current player's.

**Postconditions:** The system has paused the game for everyone.

**Main Success Scenario:**

1. Player presses the pause game button.
2. System pauses the animations.
3. The system pauses the game.
4. The system notifies the other players
5. The system pauses the game for all the players.

**Extensions** (or alternative flows):

4a. Due to a network error, the notification can not be sent.

1. Standard disconnection scenario occurs.

**Frequency of Occurrence:**

- Whenever the player wants to pause the game.

# USE CASE #22: RESUMING THE GAME

**Scope:** Euler’s Monopoly

**Level:** User Goal

**Primary Actor:** Current Player

**Stakeholders:**

Current Player: Wants to resume the game.

Other Players: Wants their game to be resumed as well.

System: Wants to successfully resume the game.

**Preconditions:** The game was paused by the same player who wants to resume.

**Postconditions:** The game starts from the same state where it was paused.

**Main Success Scenario:**

1. Player presses the resume game button.
2. The system notifies the systems of other players.
3. The system resumes the game from the same state as it was paused.

**Extensions** (or alternative flows):

2a. Due to a network error, the notification can not be sent.

1. Standard disconnection scenario occurs.

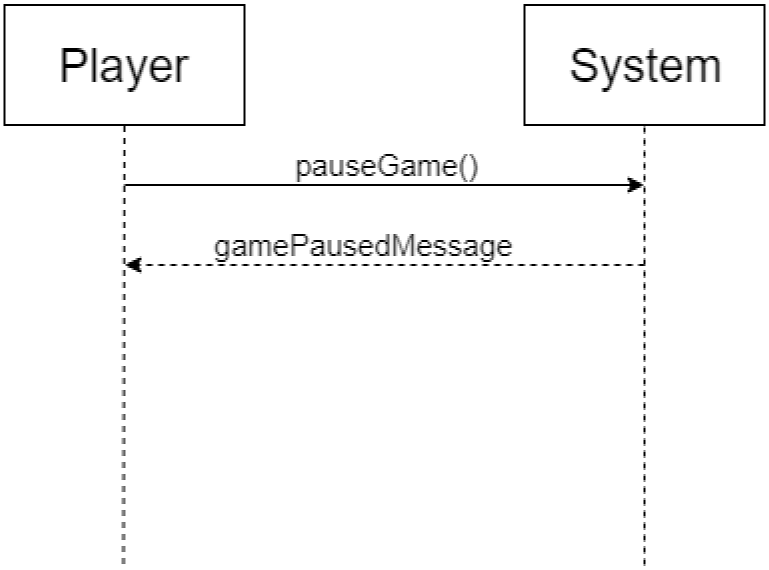
**Frequency of Occurrence:**

- Whenever the player wants to resume the game they paused.

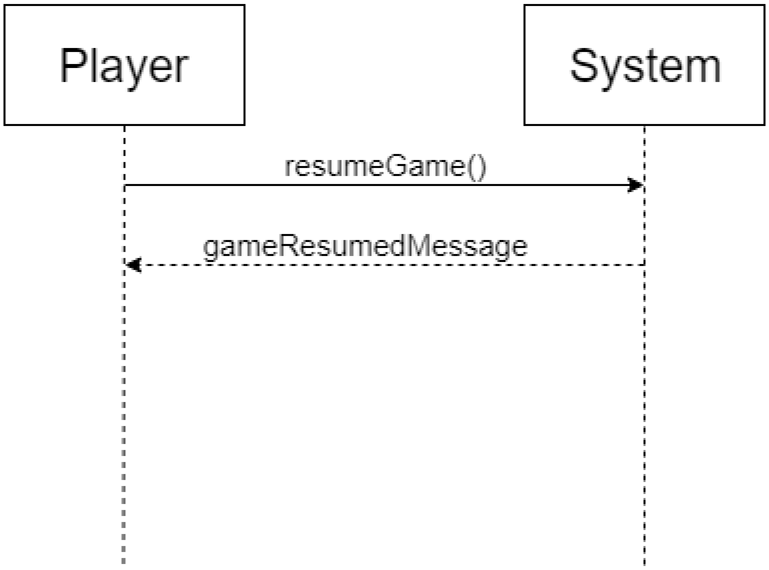
## 2) System Sequence Diagrams

These system sequence diagrams (SSD's) are designed to show a rough idea of how some selected use case narratives work behind the scenes. We intentionally selected the most important ones because through the development most of the ideas shown in these diagrams are prone to change and we did not want to continuously update our diagrams as our code changes and evolves.

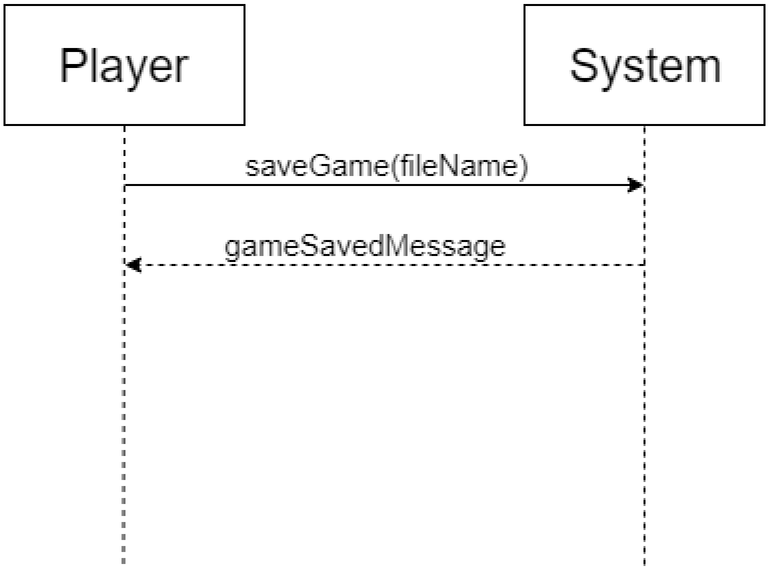
### Pausing the Game



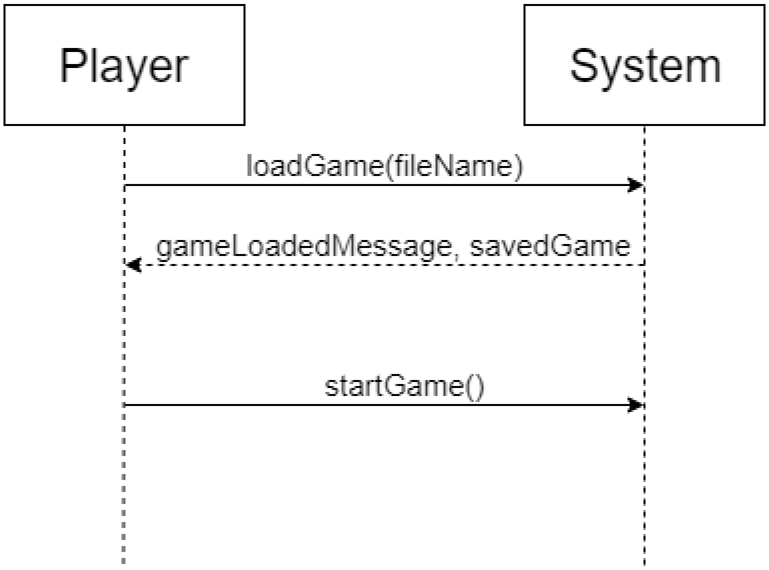
### Resuming the Game



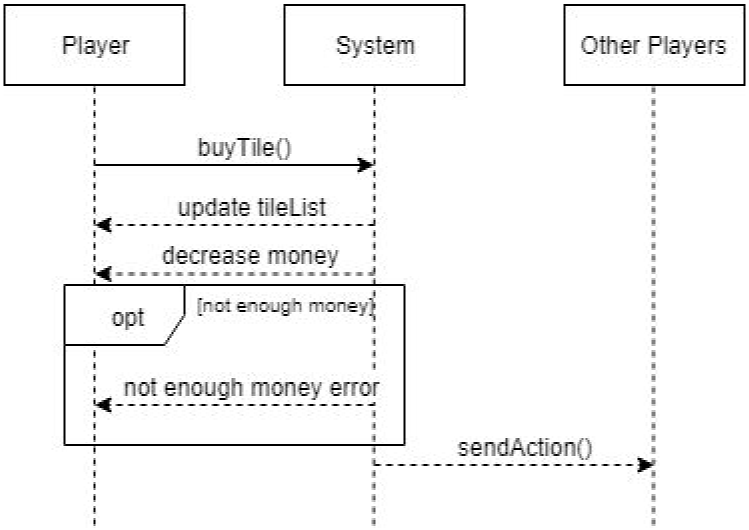
## Saving the Game



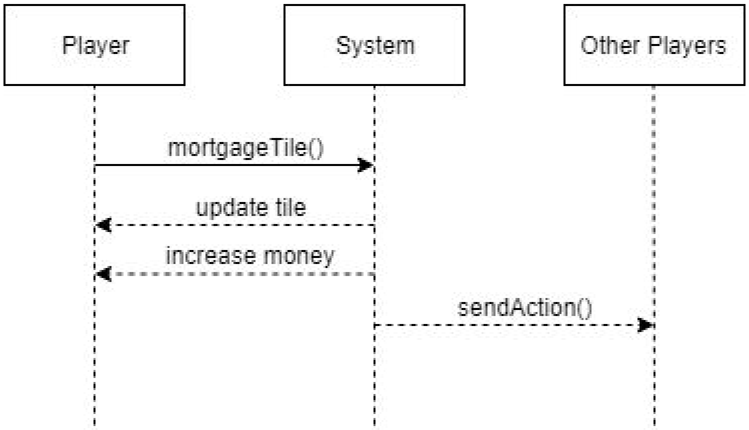
## Loading the Game



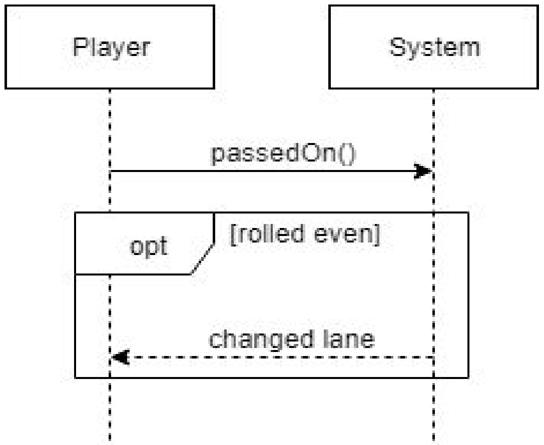
## Buy Tile



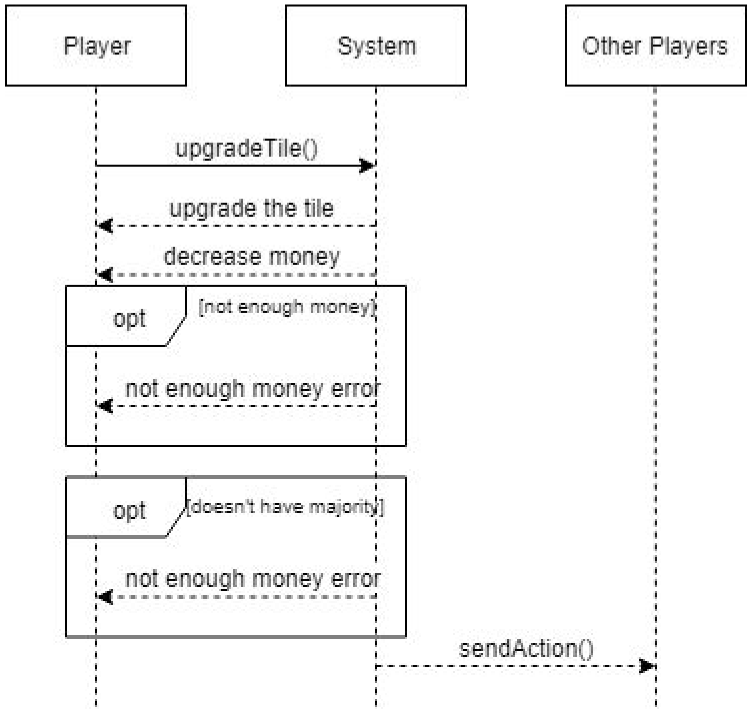
## Mortgage



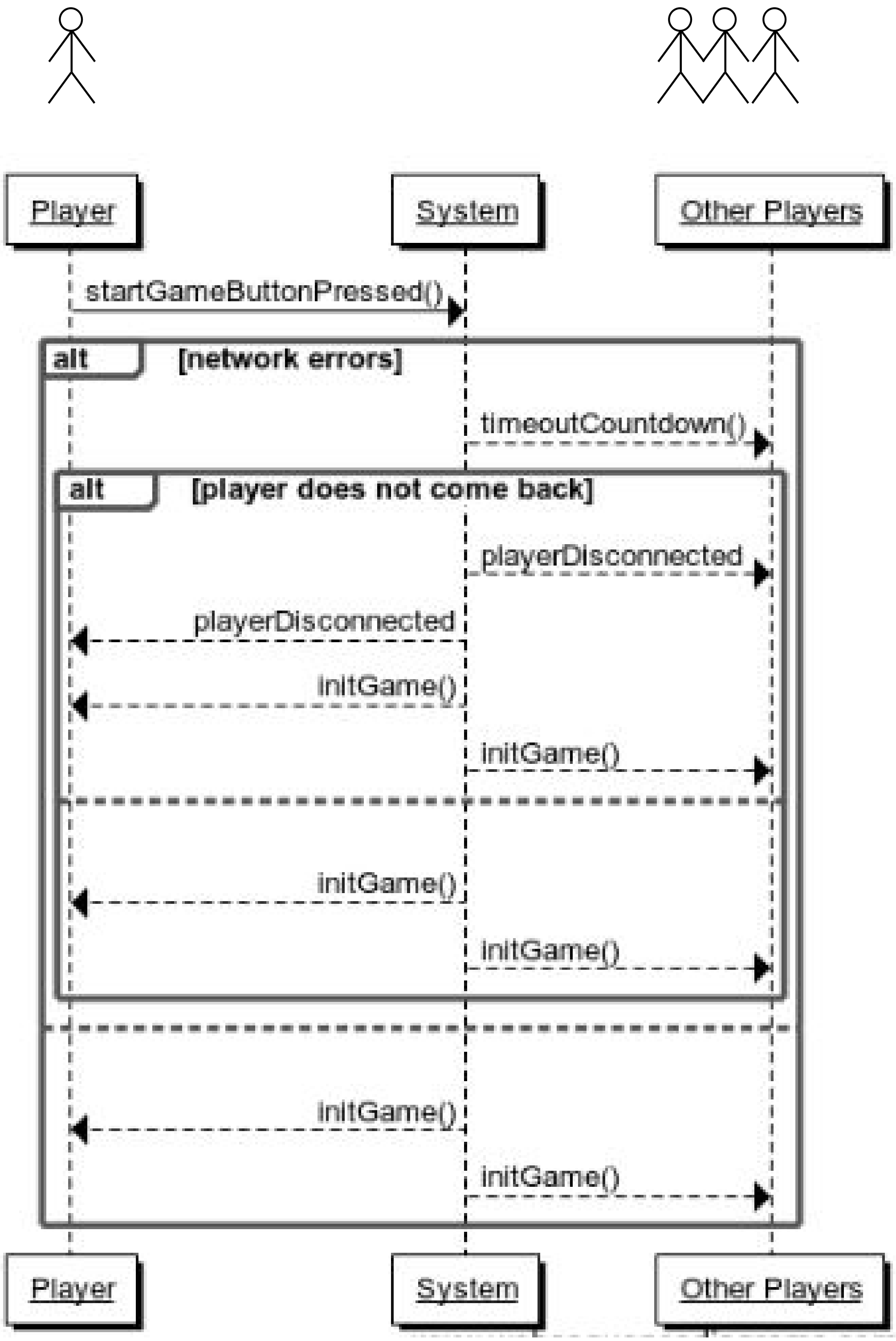
## Passing Railroad



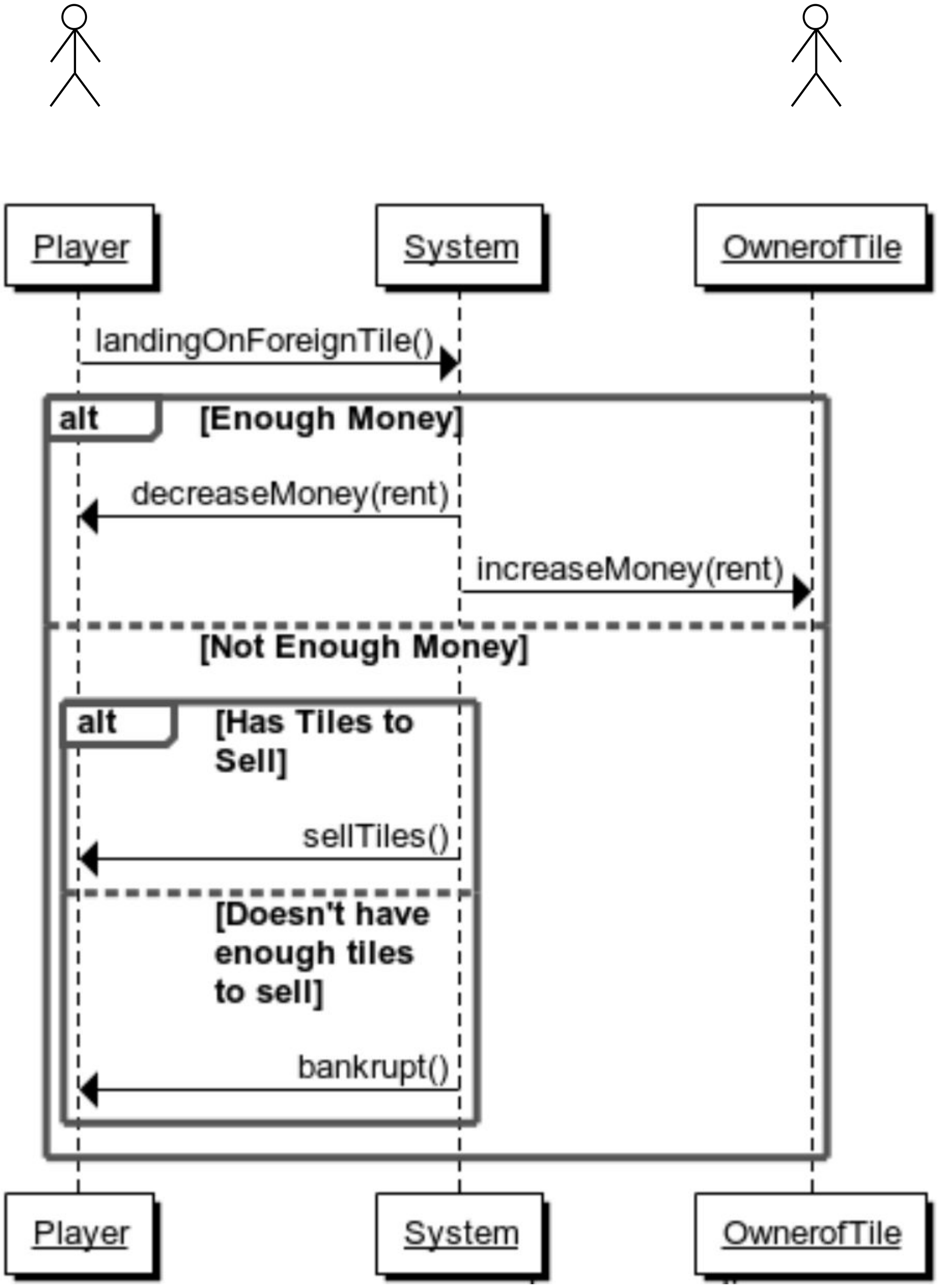
## Upgrade Tile



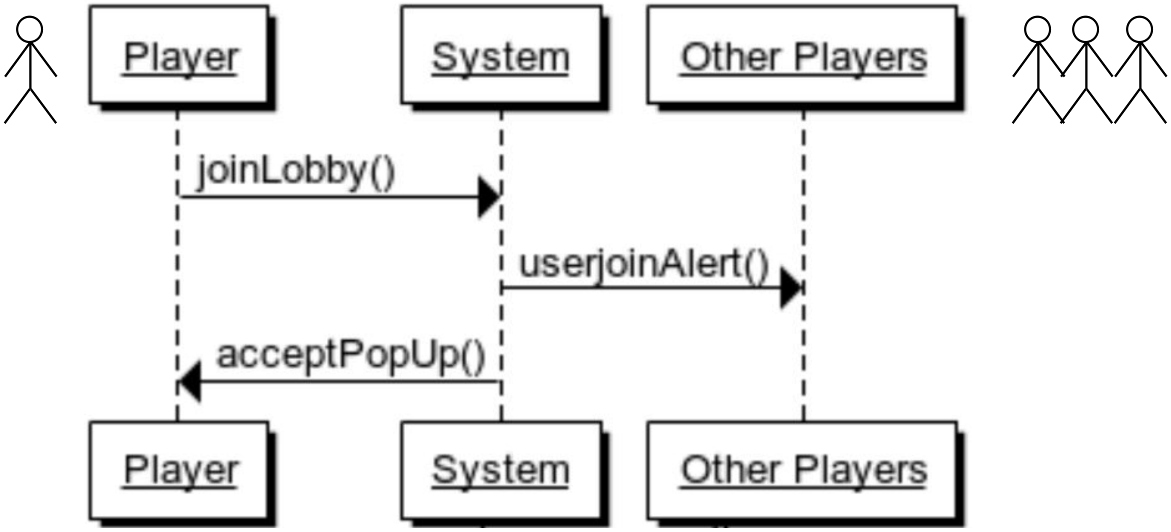
## Start Game


Player
System
Other Players
startGameButtonPressed()
alt
[network errors]
timeoutCountdown()
alt
[player does not come back]
playerDisconnected
playerDisconnected
initGame()
initGame()
initGame()
initGame()
initGame()
initGame()
Player
System
Other Players

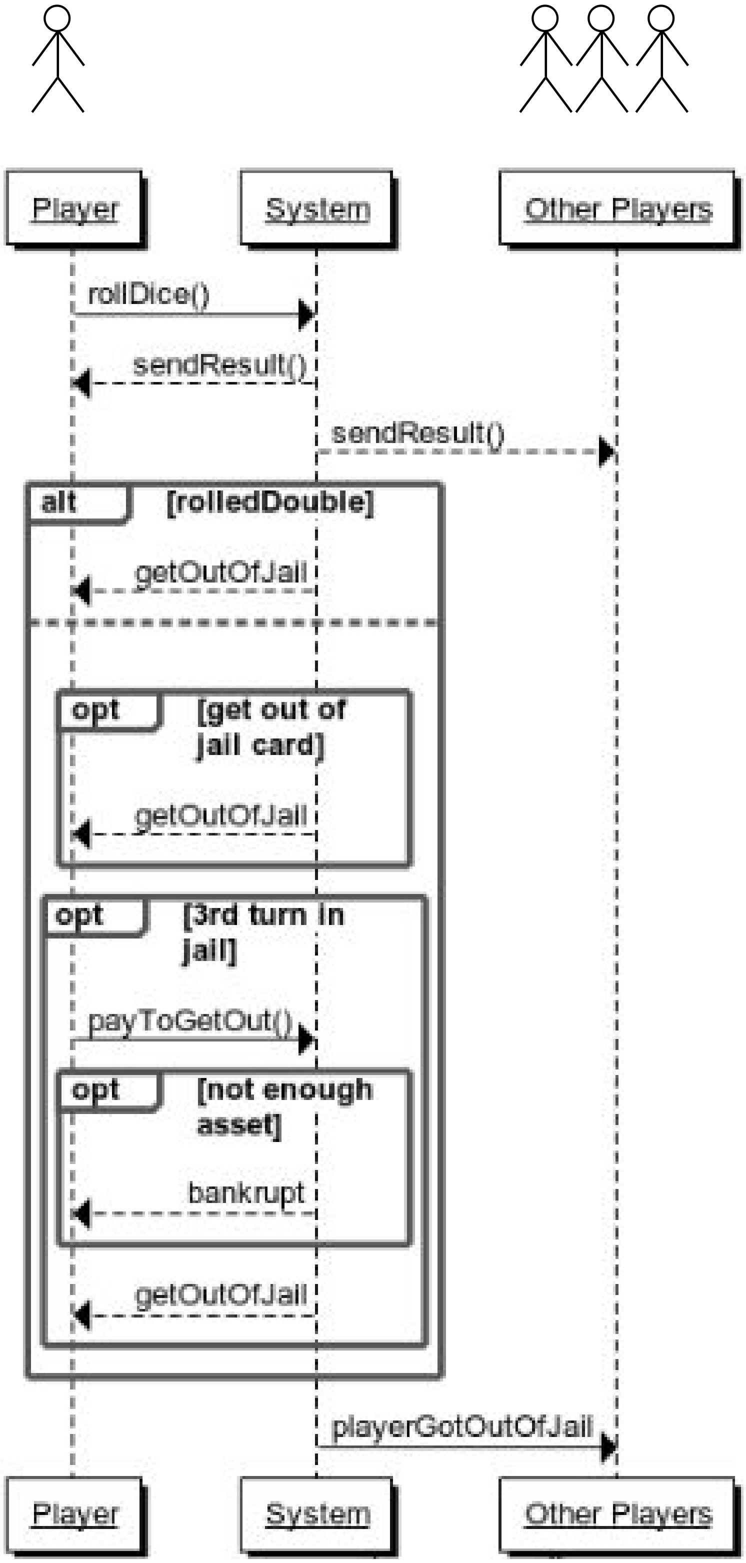
### Pay Rent


Player
System
OwnerofTile
landingOnForeignTile()
alt
[Enough Money]
decreaseMoney(rent)
increaseMoney(rent)
[Not Enough Money]
alt
[Has Tiles to Sell]
sellTiles()
[Doesn't have enough tiles to sell]
bankrupt()
Player
System
OwnerofTile

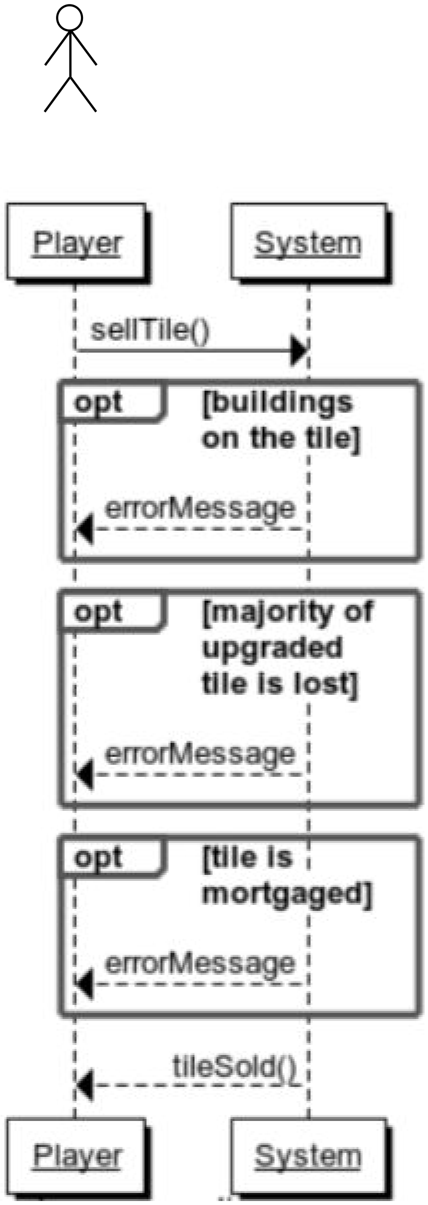
### Join Lobby


Player
System
Other Players
joinLobby()
userjoinAlert()
acceptPopUp()
Player
System
Other Players

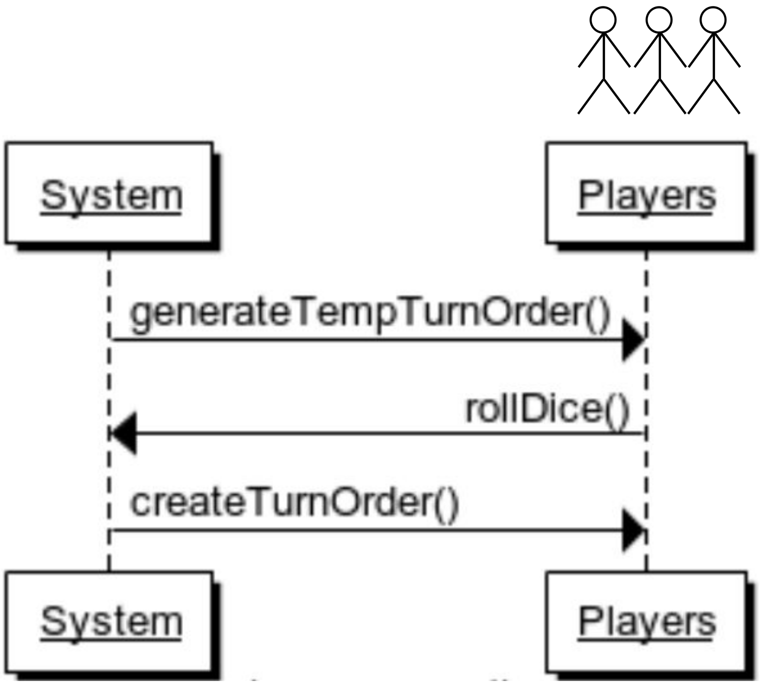
## Getting out of Jail



## Sell Tile


Player
System
sellTile()
opt
[buildings on the tile]
errorMessage
opt
[majority of upgraded tile is lost]
errorMessage
opt
[tile is mortgaged]
errorMessage
tileSold()
Player
System

### Determining Turn Order



## 3) Operation Contracts

After some discussion regarding the methods and the names we were going to use in our project, we started our work on operation contracts based on our SSD’s. When adding new SSD’s in Phase 2, we also added their corresponding operation contracts.

## Contract CO1: applyCard

Operation: applyCard()

Cross References: Use Cases: Chest/Chance Card

Preconditions: Player has a chest/chance card.

Player has network connection.

Post Conditions: Player’s money status is updated.

Player’s current position is updated.

Player’s being in jail status is updated.

## Contract CO2: rollDice

Operation: rollDice()

Cross References: Use Cases: Rolling Dice, Jail, Roll Three

Preconditions: Player has network connection, and eligible to roll dice.

Post conditions: Dice result is randomly generated. Dice result is sent to players.

## Contract CO3: changeLane

Operation: changeLane()

Cross References: Use Cases: Passing Railroad, Subway Tile,  Chest/Chance Card

Preconditions: One of the following:

Player rolls even and passes through railroad.

Player, after landing on the subway tile, decides to ‘teleport’ to a tile on a different lane.

Post Conditions: Player’s lane is updated.

## Contract CO4: endTurn

Operation: endTurn()

Cross References: Use Cases:

Preconditions: Player has network connection.
Player has played its turn.

Post Conditions: Turn is changed with updating the current player.

## Contract CO5: saveGame

Operation: saveGame()

Cross References: Use Cases: Saving the Game

Preconditions: Game is paused.

Post conditions: A save file with the specified name is created and the saved game object is written into that file.

## Contract CO6: loadGame

Operation: loadGame()

Cross References: Use Cases: Loading the Game

Preconditions: The host has selected a file to load.
The save game object was successfully transferred to the client.

Post conditions: The game state is created from the saved game object.
This object is the same for all of the players. GUI elements are created from the generated Monopoly game.

## Contract CO7: pauseGame

Operation: pauseGame()

Cross References: Use Cases: Pausing the Game

Preconditions: The player that wants to pause the game has the current turn.

Post conditions: Animation controller is notified of the pause request.
The game and animation threadsare paused.

## Contract CO8: resumeGame

Operation: resumeGame()

Cross References: Use Cases: Resuming the Game

Preconditions: The game was paused by the same player who wants to resume.

Post conditions: Animation controller is notified of the resume request.
The game and animation threads are resumed.

## Contract CO9: payRent

Operation: payRent()

Cross References: Use Cases: Pay Rent

Preconditions: Player has landed on a foreign, non-mortgaged tile and has network connection.

Post Conditions: Money states of two players are updated.

## Contract CO10: buyTile

Operation: buyTile()

Cross References: Use Cases: Buy Tile

Preconditions: Player has landed on an unowned tile and have network connection.

Post Conditions: Money state of the player is updated. Owned tiles list is changed.

## Contract CO11: requestJoin

Operation: requestJoin()

Cross References: Use Cases: Join Lobby

Preconditions: Player has network connection and the game has not started yet.

Post Conditions: A new user is created. Users lists is updated for every player.

## Contract CO12: sellTile (to bank)

Operation: sellTile()

Cross References: Use Cases: Pay Rent, Getting Out Of Jail, Sell Tile

Preconditions: Player has network connection.

Post Conditions: Money state and asset list are updated.

## Contract CO13: sellTile (to player)

Operation: sellTile()

Cross References: Use Cases: Pay Rent, Getting Out Of Jail, Sell Tile

Preconditions: Player has network connection.

Post Conditions: Money states and asset lists of two players are updated.

## Contract CO14: drawCard

Operation: drawCard()

Cross References: Use Cases: Roll Three, Chest/Chance Card

Preconditions: Player has landed on a chance, Roll Three or community tile or it is the start of the game and each player has to draw a Roll Three card.

Post Conditions: The relevant card list of each player is updated so that it contains the new card. The card deck is also updated.

## Contract CO15: buyBuilding

Operation: buyBuilding()

Cross References: Use Cases: Upgrade Tile

Preconditions: Player has turn and has network connection.

Post Conditions: Money state of the player is updated. The bank receives money for the transaction. The upgrade field of the tile is updated.

## Contract CO16: landOnSquare

Operation: landOnSquare()

Cross References: Use Cases: Pay Rent, Buy Tile, Getting out of Jail, Chest/Chance Card, Subway Tile, Roll Three, Holland Tunnel

Preconditions: Player has network connection.
Player lands on a tile.

Post Conditions: Player’s current position is updated.
Player’s money state is updated.
Player’s asset list is updated.
Player’s being in jail status is updated.

## Contract CO17: move

Operation: move()

Cross References: Use Cases:

Preconditions: Player has landed on a foreign, non-mortgaged tile and has network connection.

Post Conditions: Money states of two players are updated.

## Contract CO18: passSquare

Operation: passSquare()

Cross References: Use Cases: Passing Railroad

Preconditions: Player has network connection.
Player lands on a tile.

Post Conditions: Player’s current position is updated.

Player’s money state is updated.

Player’s asset list is updated.

## Contract CO19: teleport

Operation: teleport()

Cross References: Use Cases: Holland Tunnel, Subway Tile

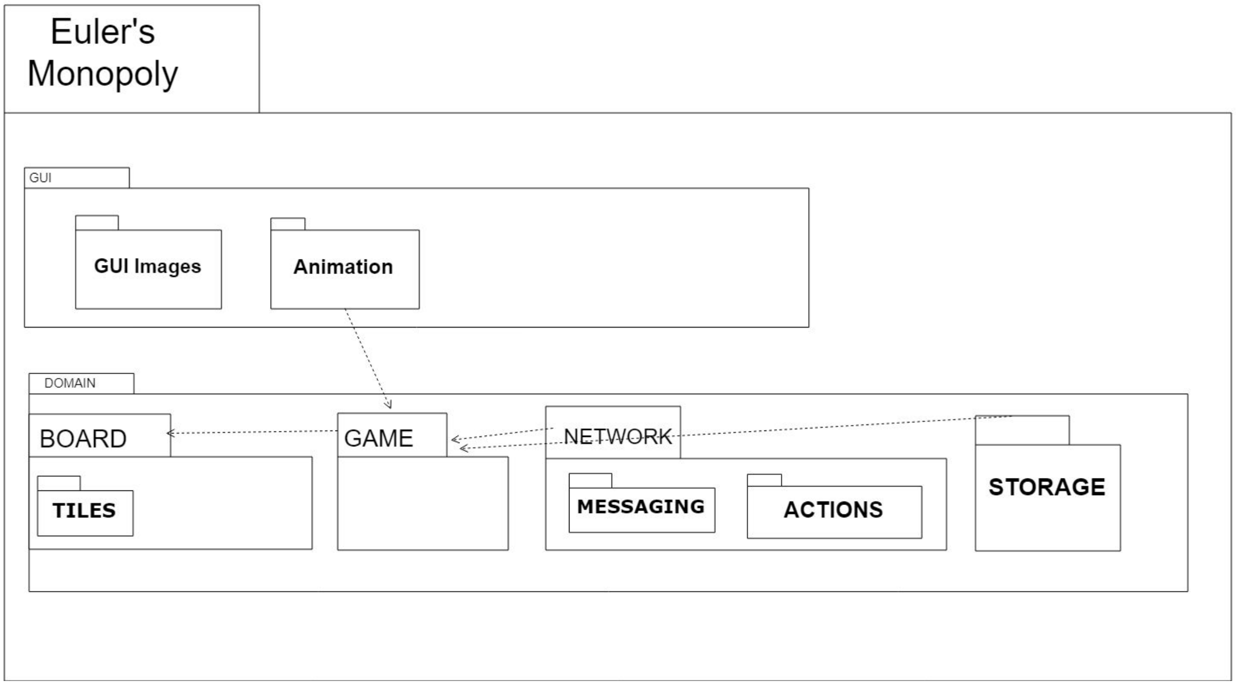
Preconditions: Player has landed on a subway or holland tile.

Player has network connection.

Post Conditions: Players current position is updated.

## 4) Package Diagram

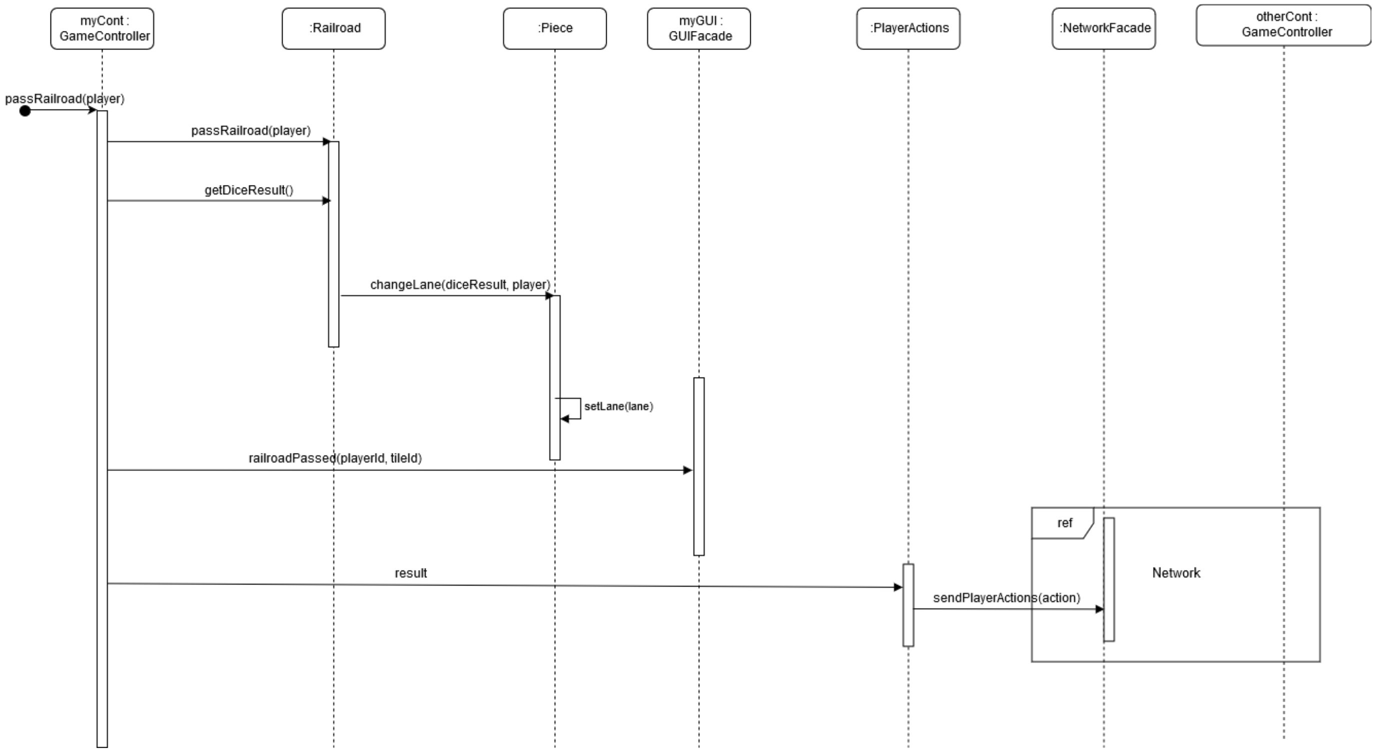
After finishing with our operation contracts, it was time to start coding. After a short discussion we all agreed on a package layout we were all comfortable with. We realized that we had made a pretty correct layout because we did not need to make any changes to our package diagram in Phase 2.


Euler's Monopoly
GUI
GUI Images
Animation
DOMAIN
BOARD
TILES
GAME
NETWORK
MESSAGING
ACTIONS
STORAGE

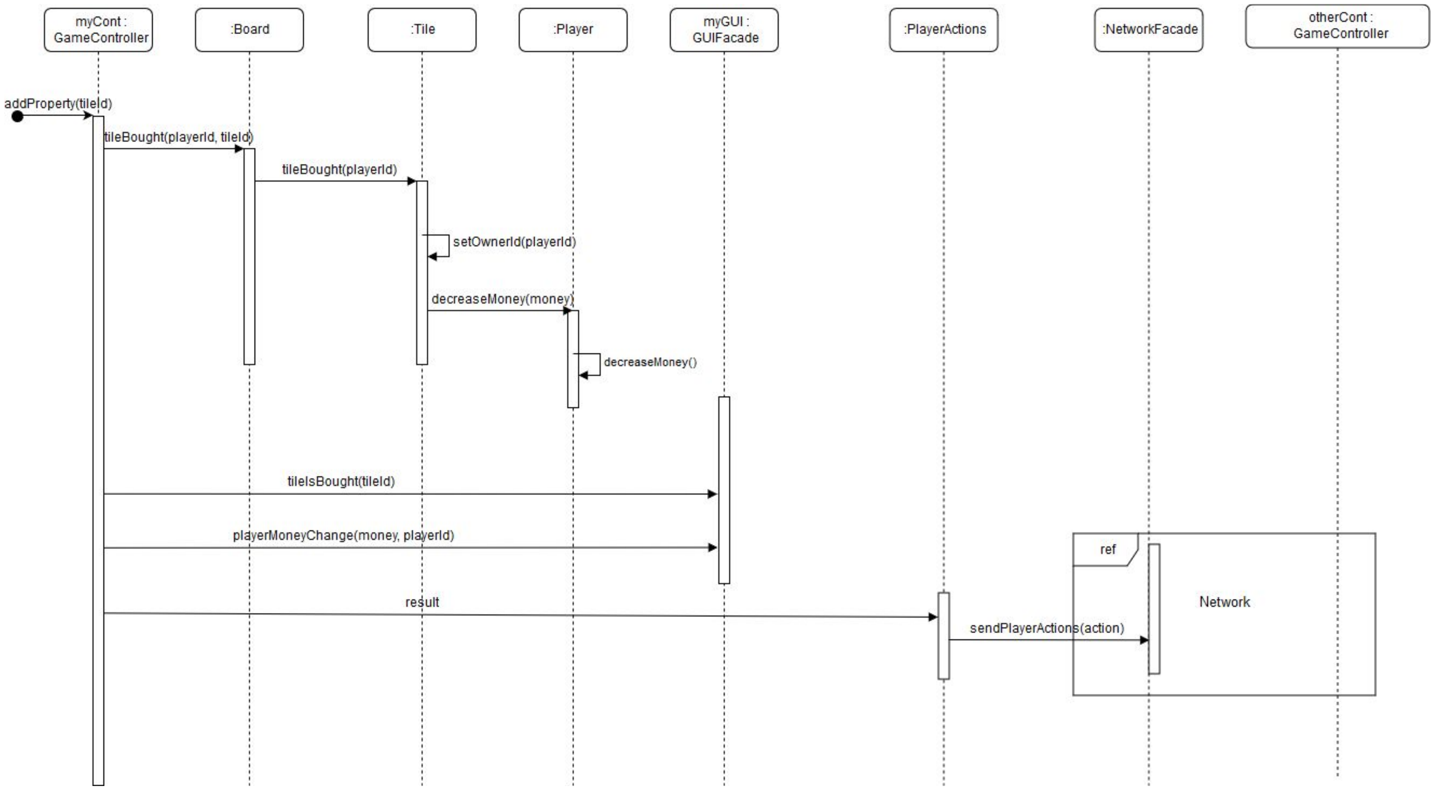
## 5) Sequence Diagrams

For sequence diagrams, we had to have started to work on the code so that we could correctly determine which methods we use and how the classes and objects interact with each other. Fortunately we had already written most of the methods, albeit mostly empty, so we did not have to waste time making the sequence diagrams. We selected the most complicated methods to write about so that we could demonstrate our diagram-making skills.

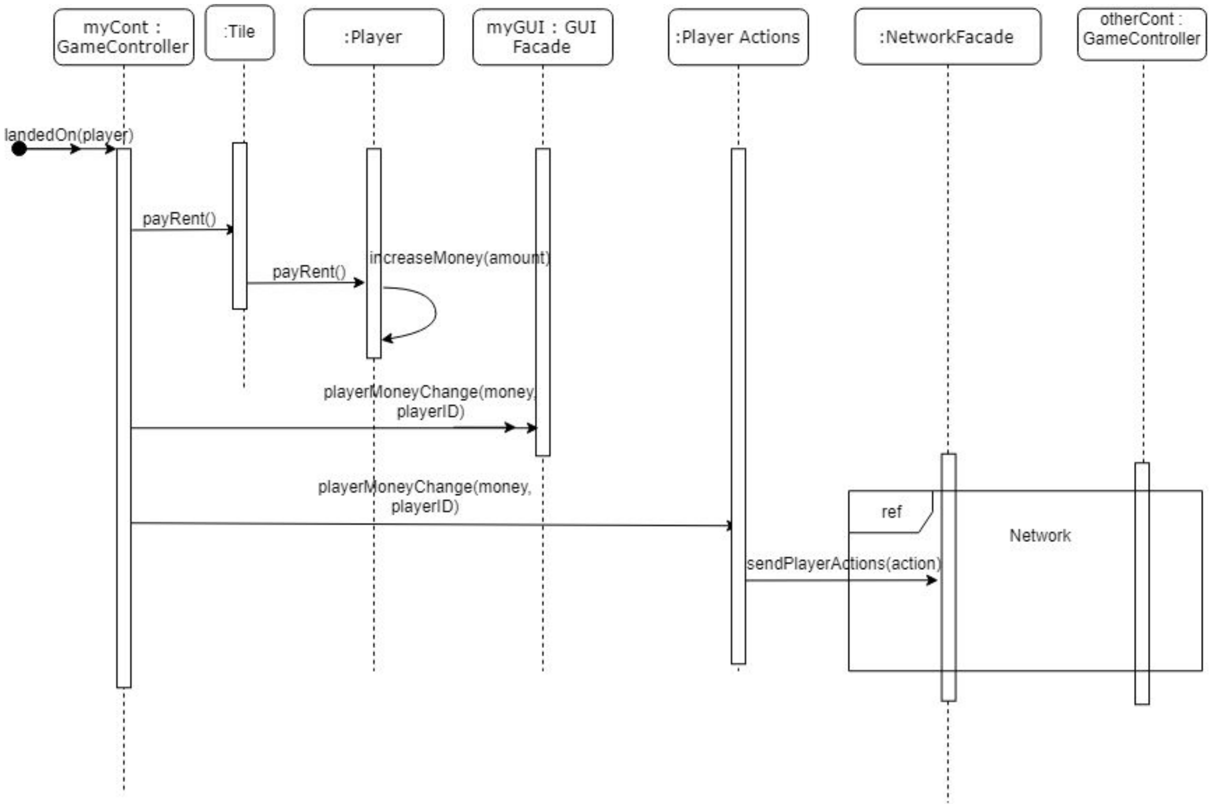
# passRailroad()



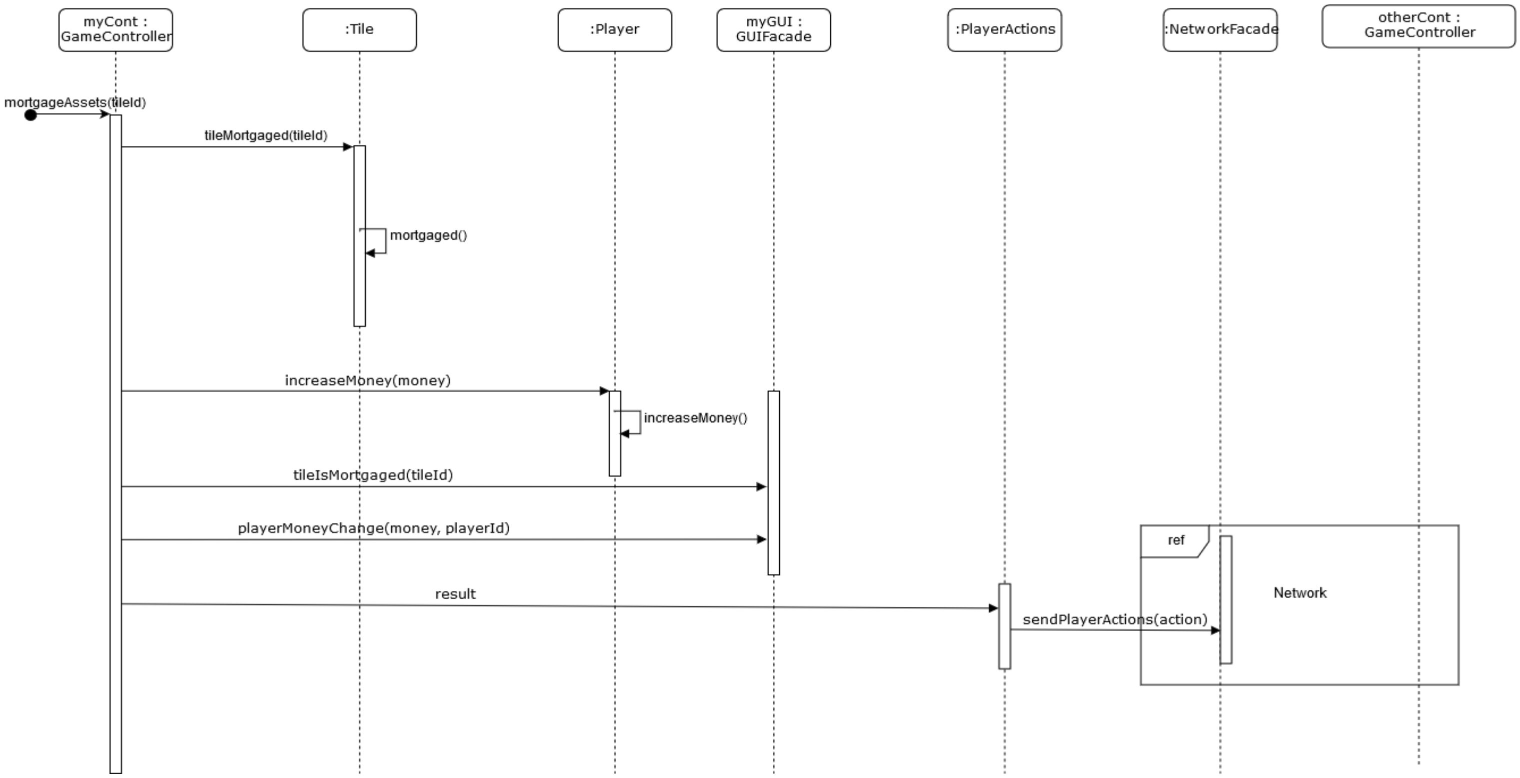
# addProperty()



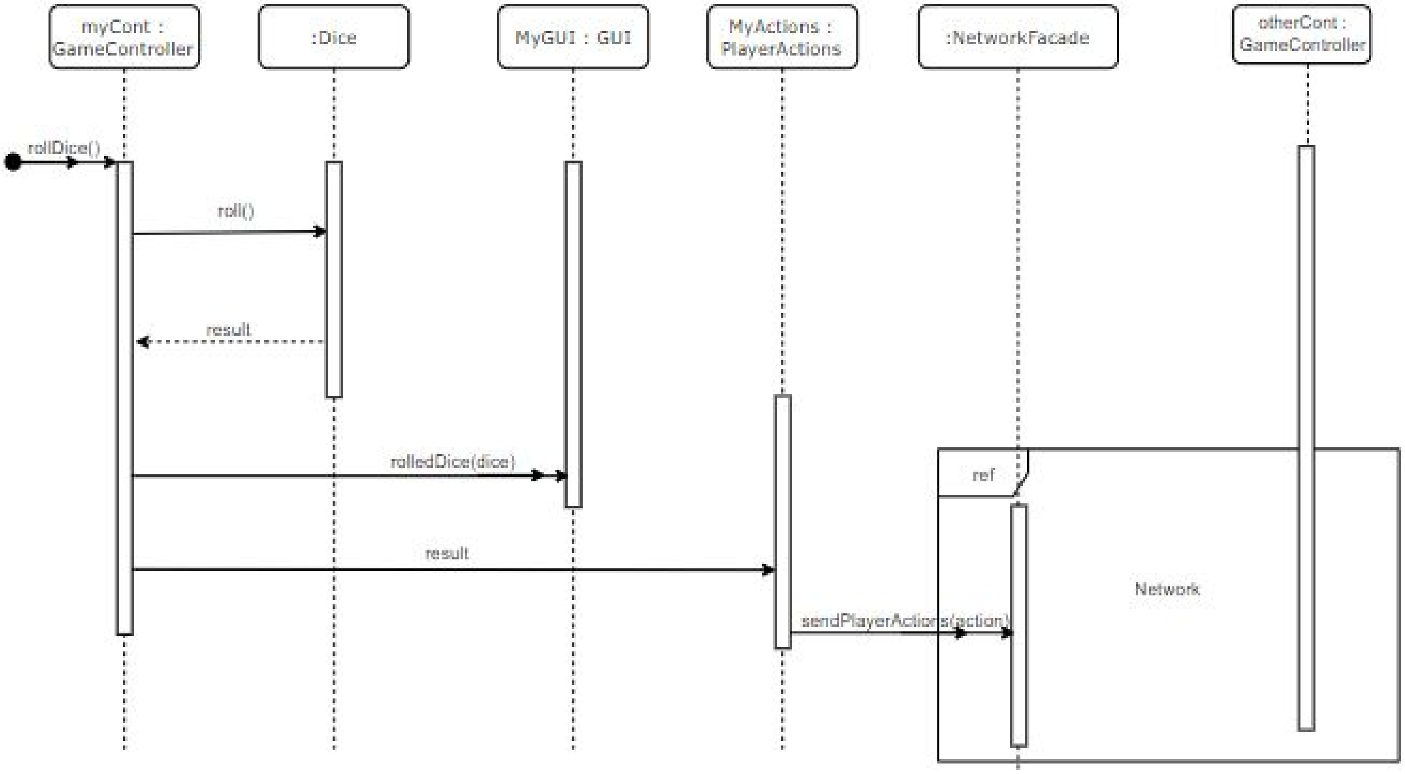
# payRent()



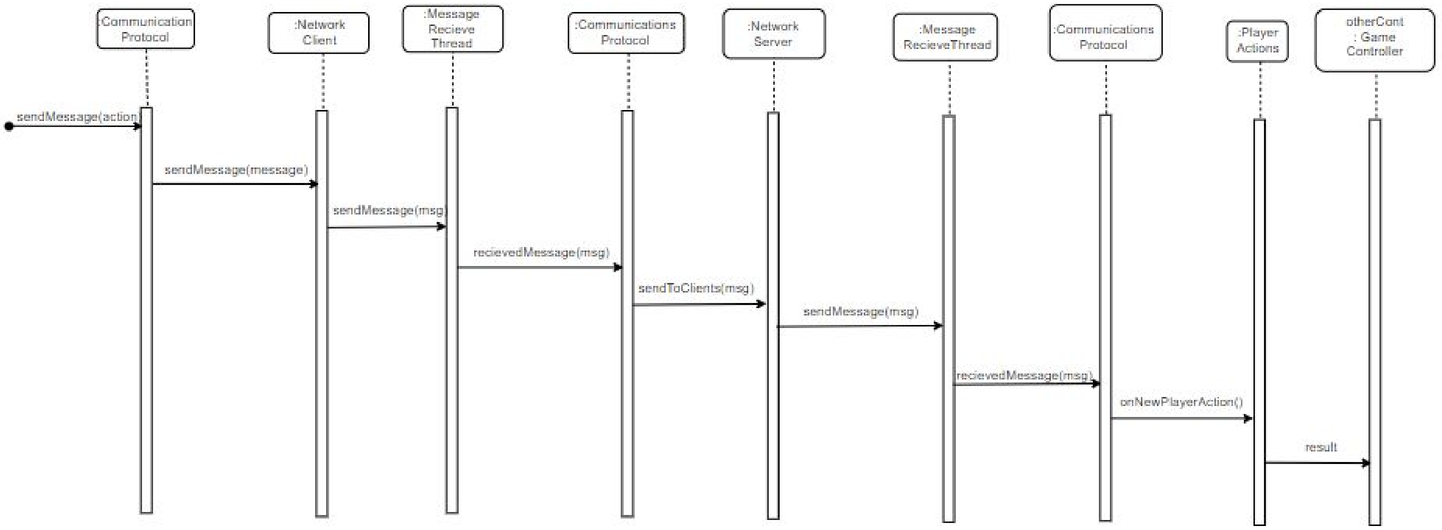
# mortgage()



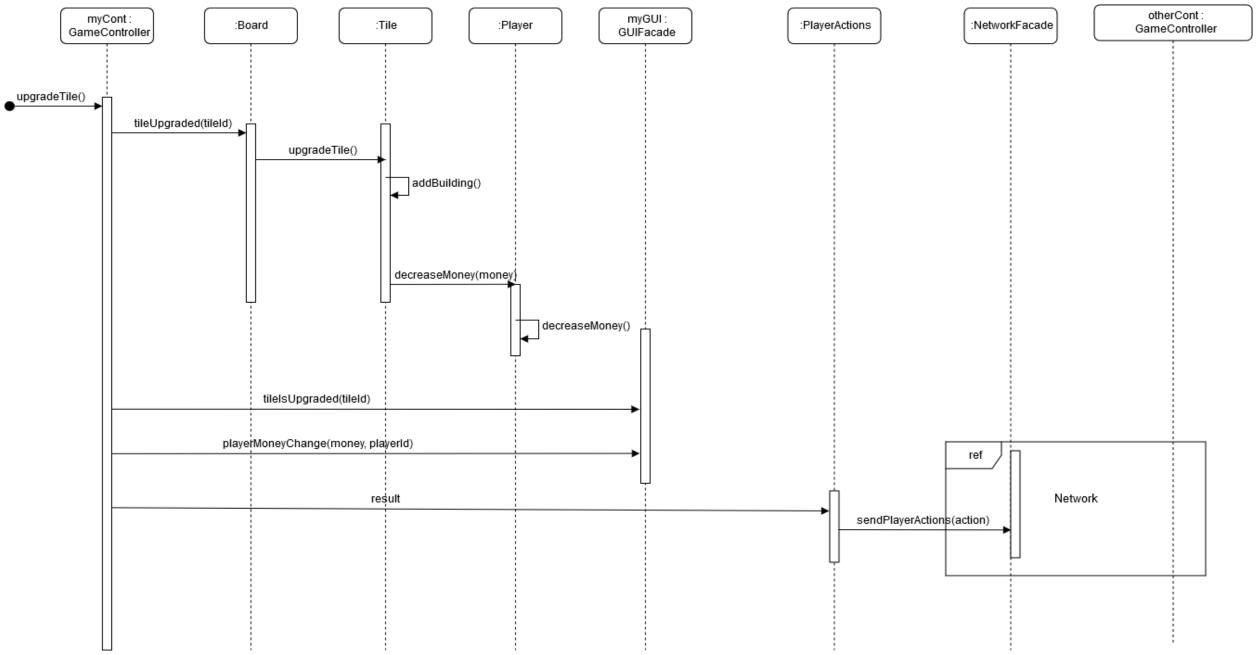
# rollDice()



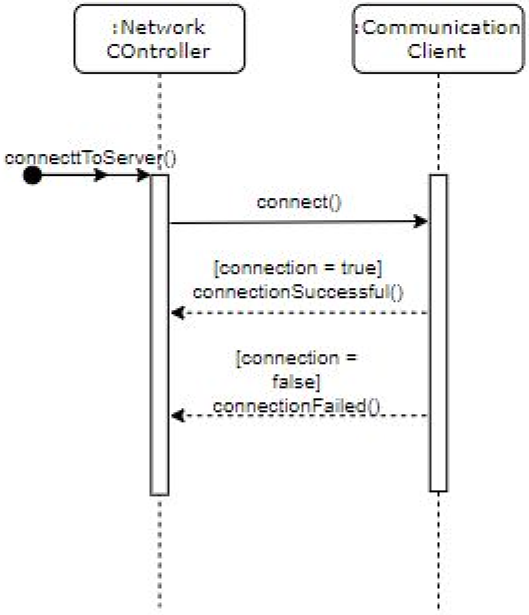
# Network



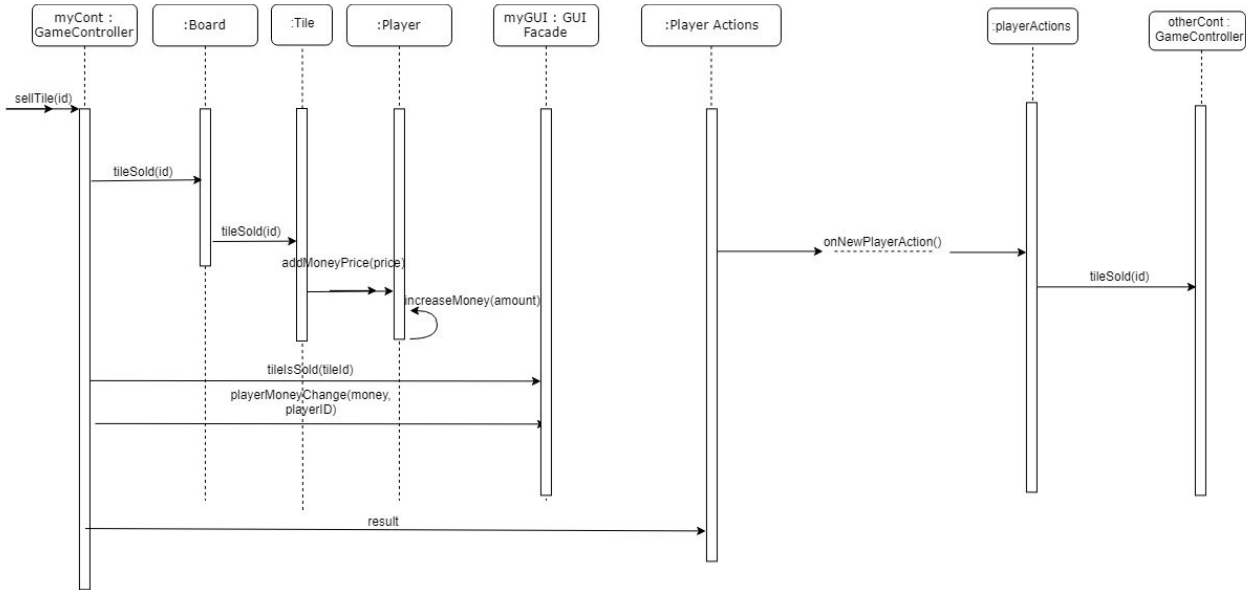
# upgradeTile()



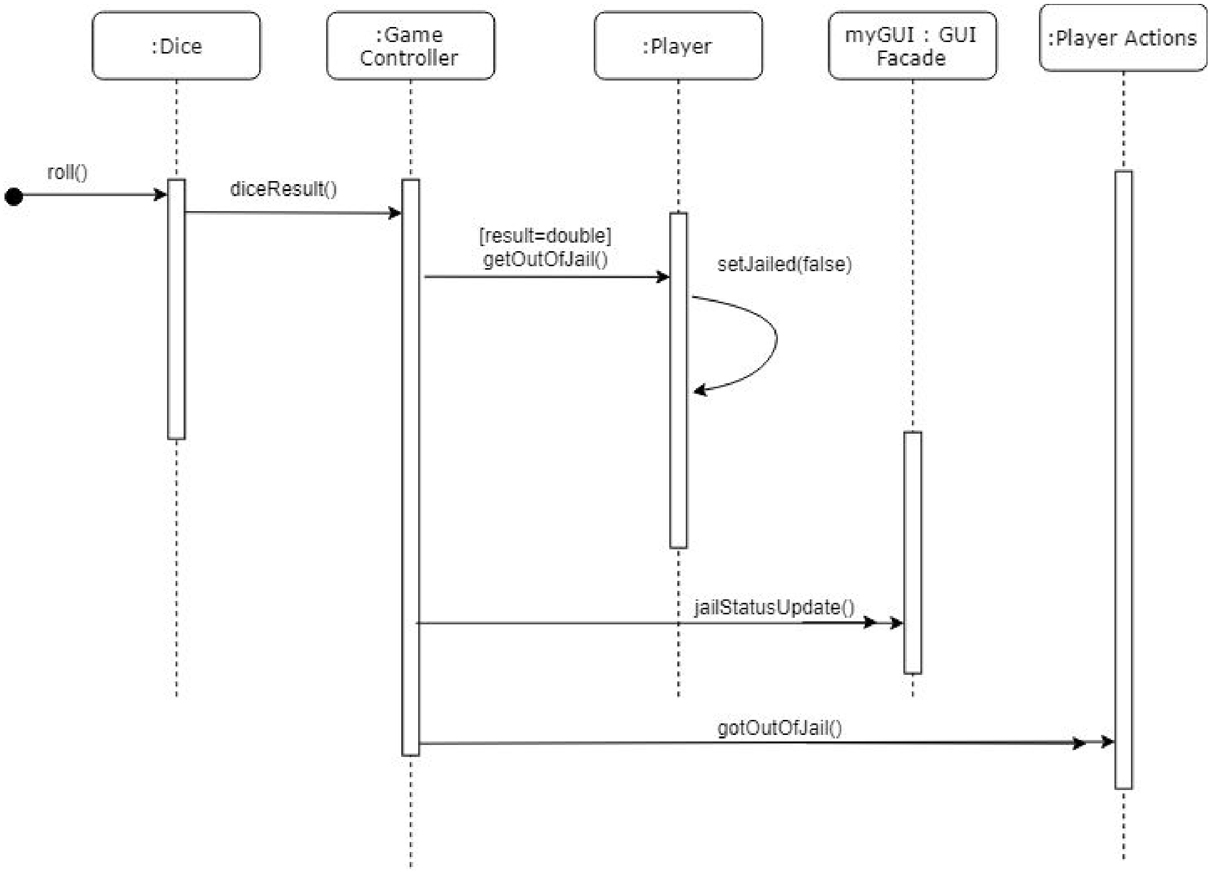
# connectToServer()


:Network COntroller
:Communication Client
connecttToServer()
connect()
[connection = true] connectionSuccessful()
[connection = false] connectionFailed()

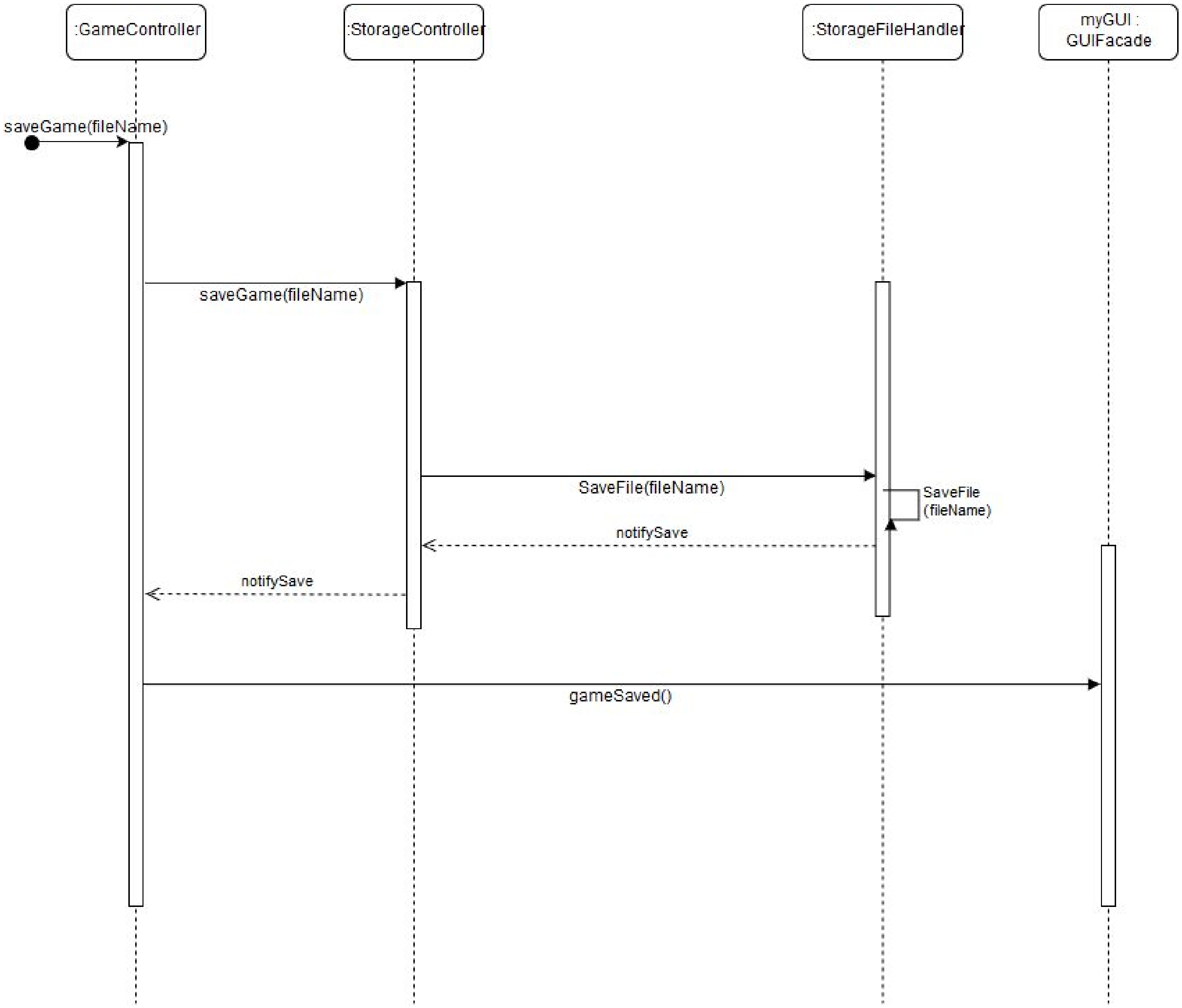
# sellTile()



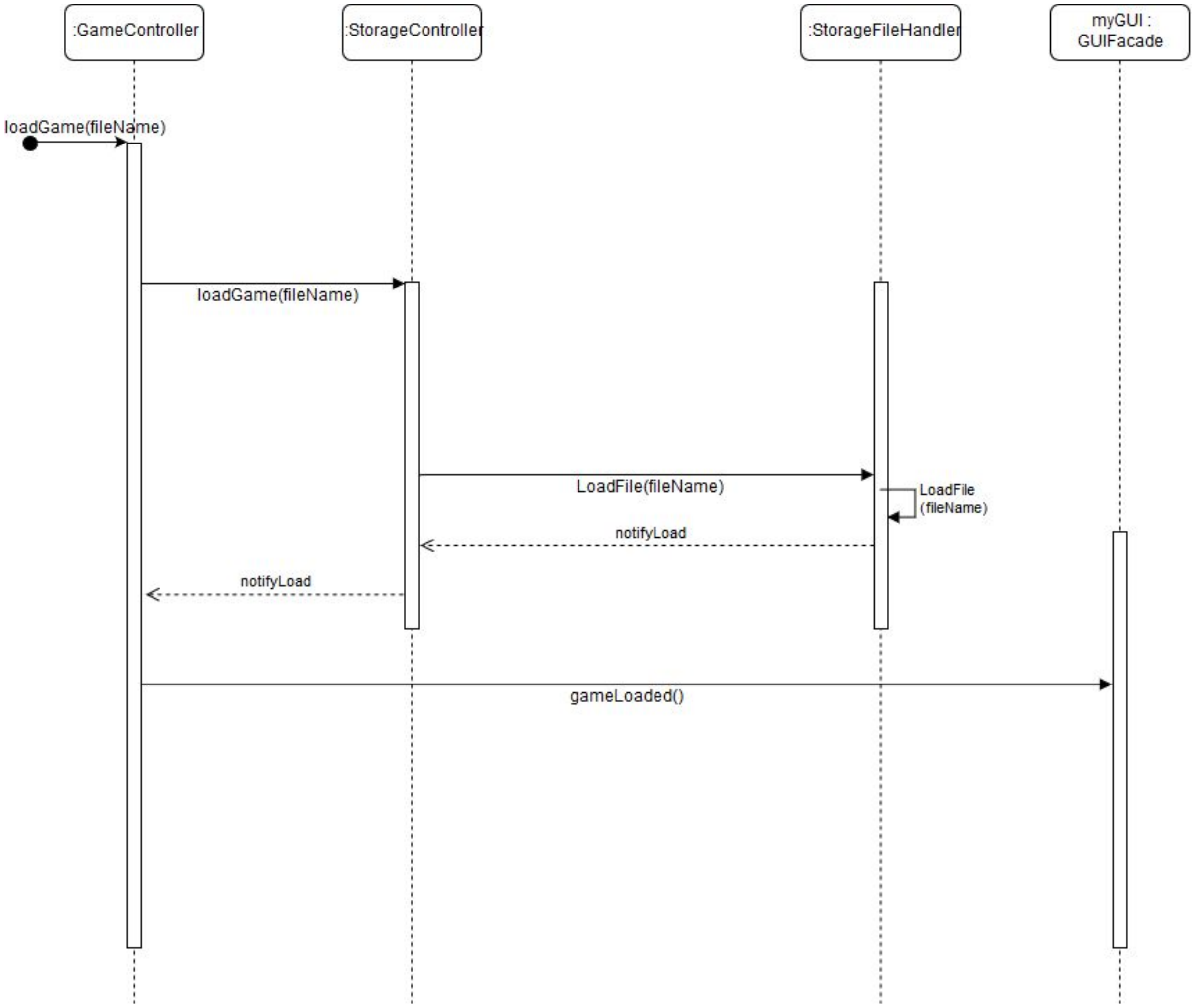
# getOutOfJail()



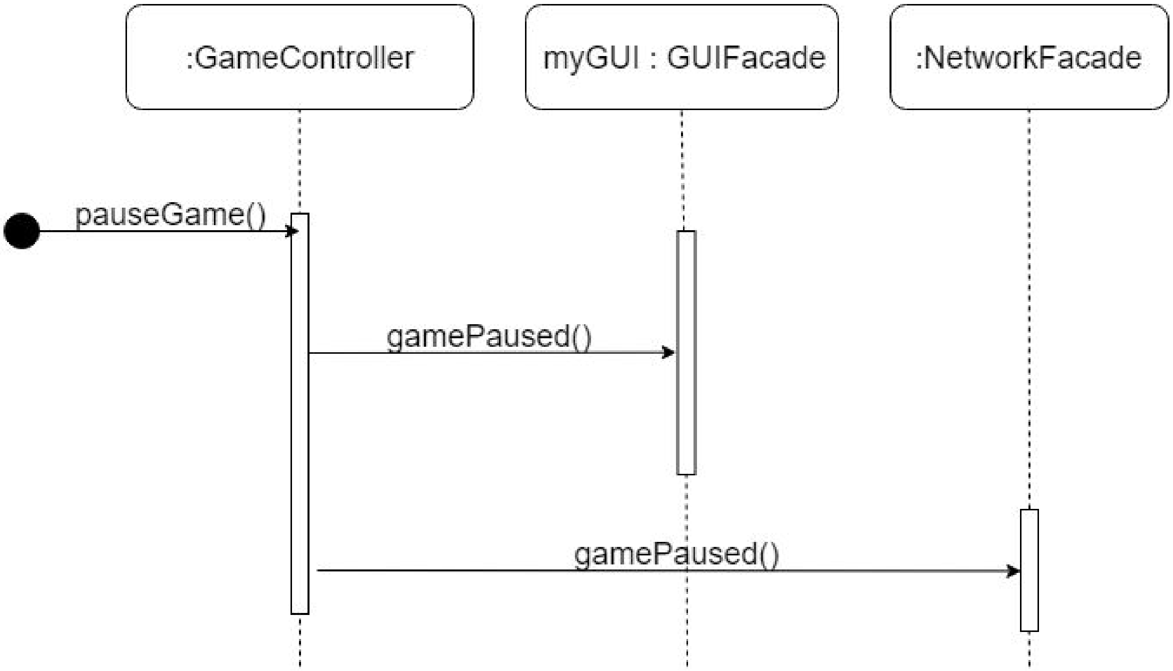
# saveGame()


:GameController
:StorageController
:StorageFileHandler
myGUI : GUIFacade
saveGame(fileName)
saveGame(fileName)
SaveFile(fileName)
SaveFile (fileName)
notifySave
notifySave
gameSaved()

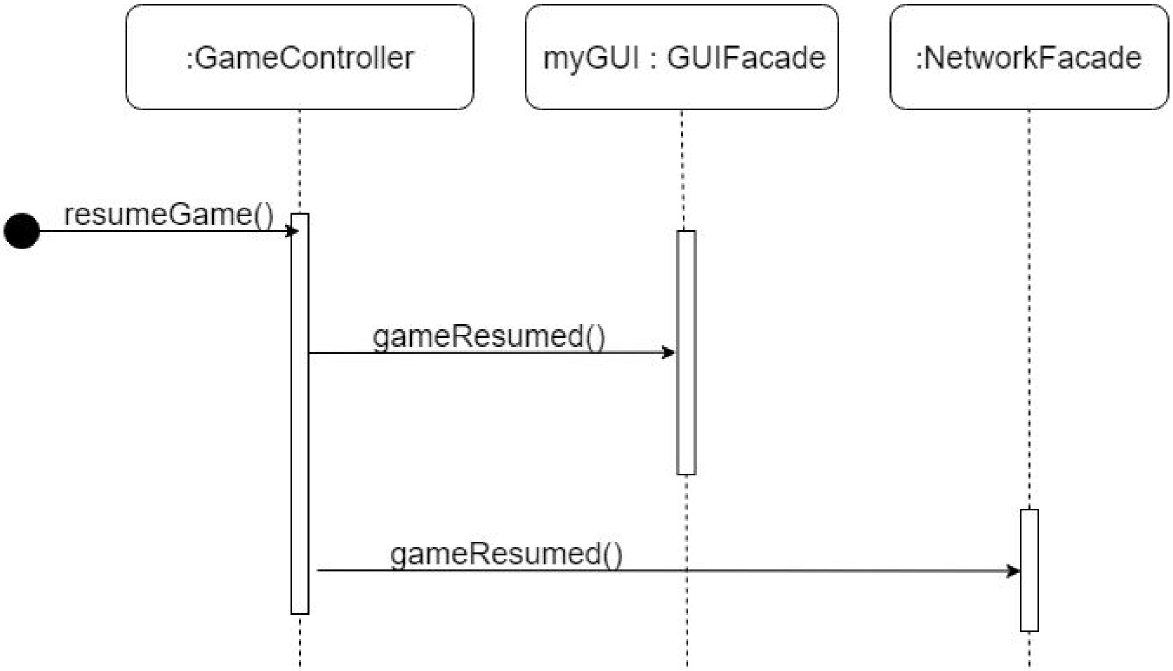
# loadGame()


:GameController
:StorageController
:StorageFileHandler
myGUI :
GUIFacade
loadGame(fileName)
loadGame(fileName)
LoadFile(fileName)
LoadFile
(fileName)
notifyLoad
notifyLoad
gameLoaded()

## pauseGame()



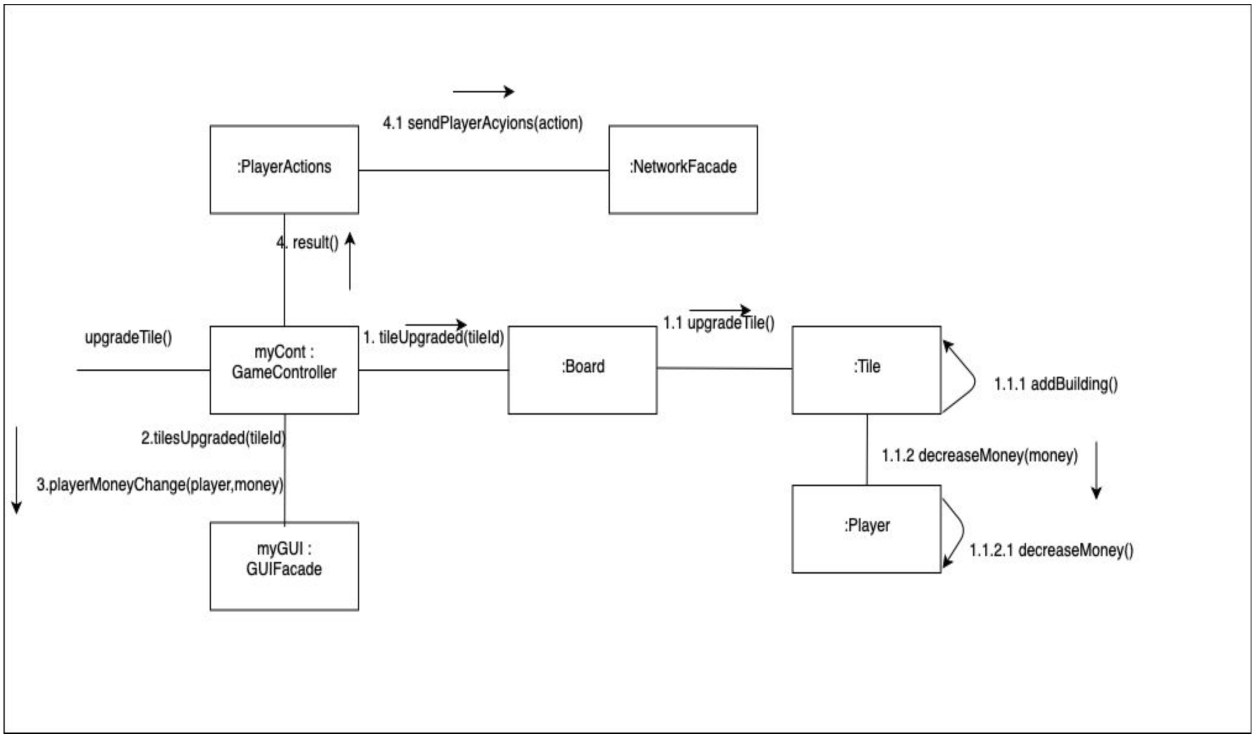
## resumeGame()



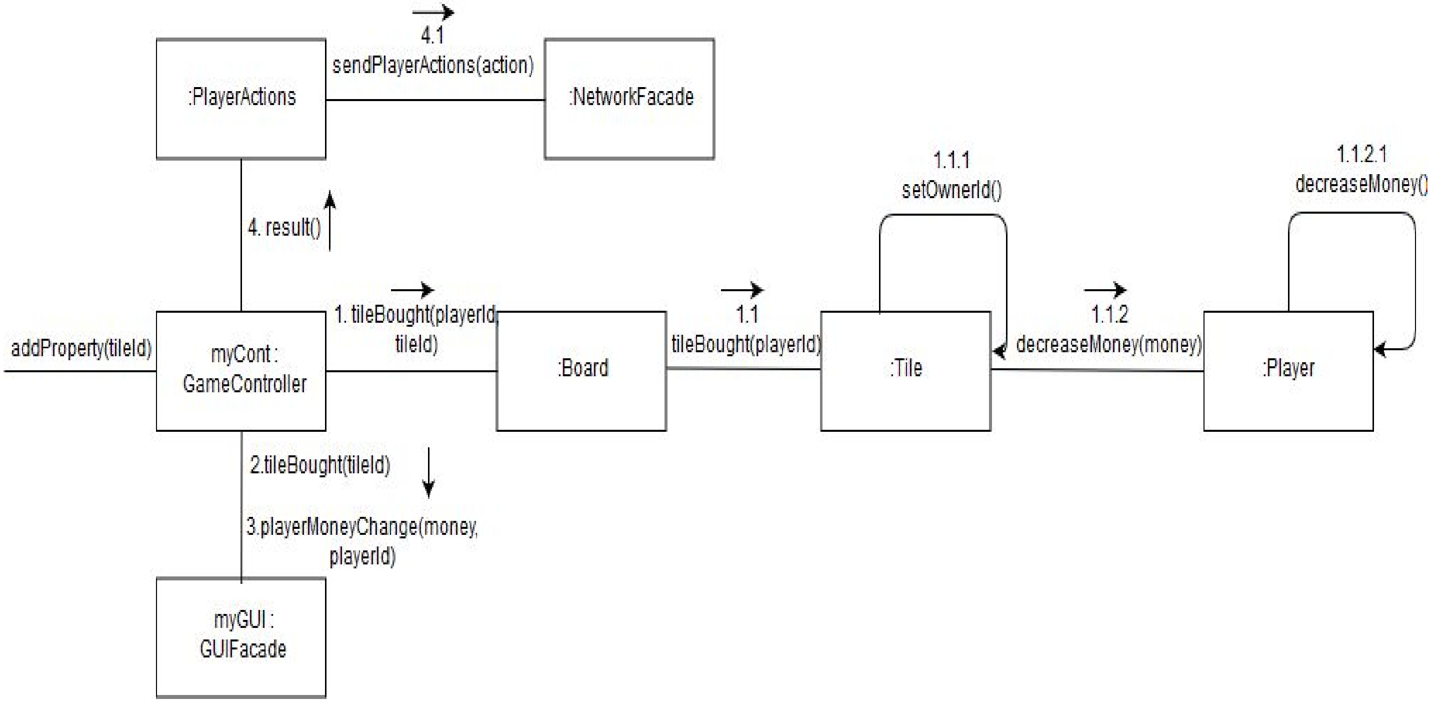
## 6) Communication Diagrams

After the sequence diagrams it was time to construct communication diagrams. These did not prove to be hard as we had already done the sequence diagrams. The important thing to note is that both have their clear advantages in how they represent the methods.

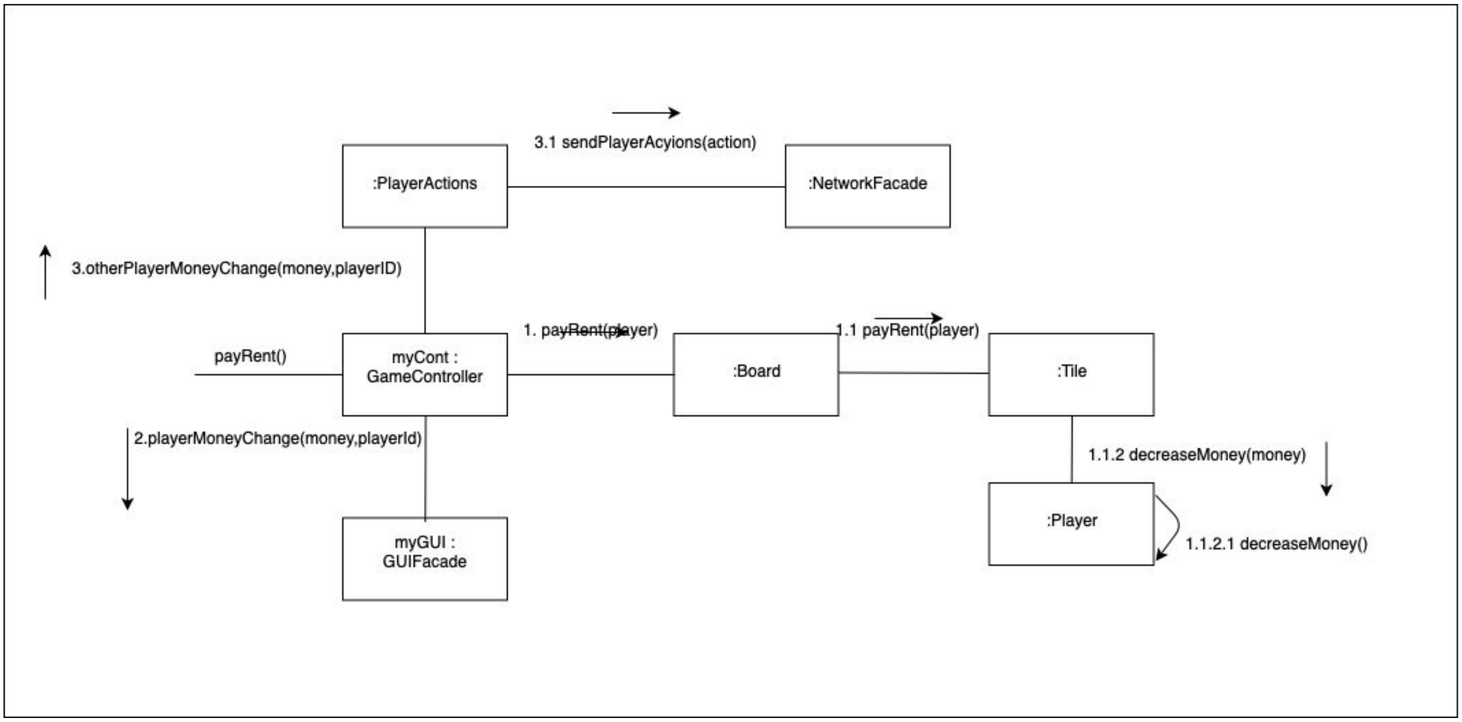
# upgradeTile()



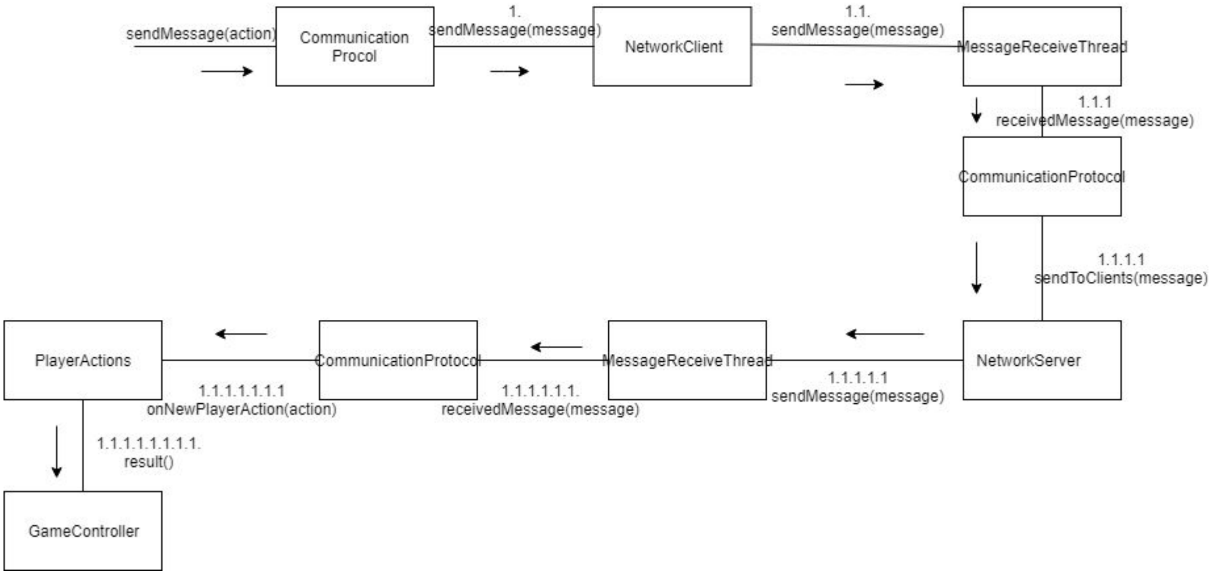
# buyTile()



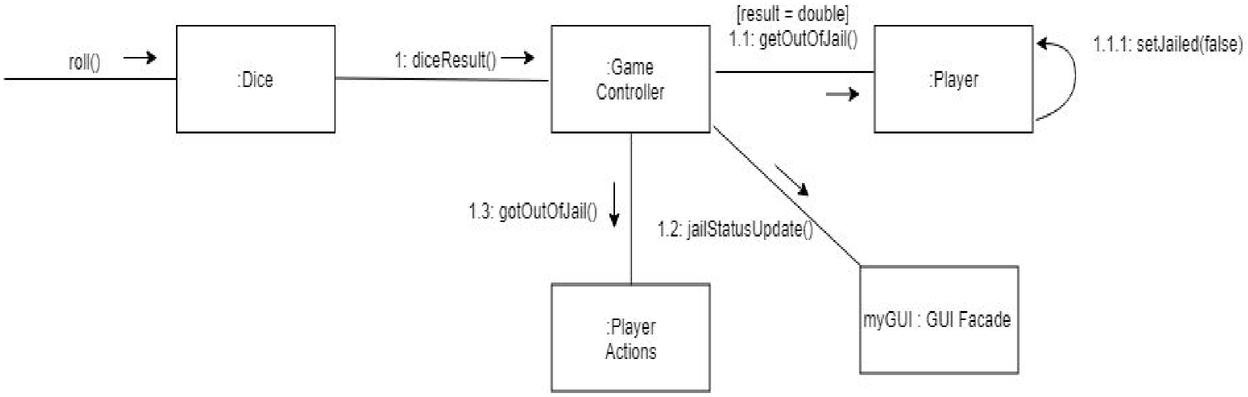
# payRent()


3.1 sendPlayerAcyions(action)
:PlayerActions
:NetworkFacade
3.otherPlayerMoneyChange(money,playerID)
payRent()
myCont :
GameController
1. payRent(player)
:Board
1.1 payRent(player)
:Tile
2.playerMoneyChange(money,playerId)
1.1.2 decreaseMoney(money)
:Player
1.1.2.1 decreaseMoney()
myGUI :
GUIFacade

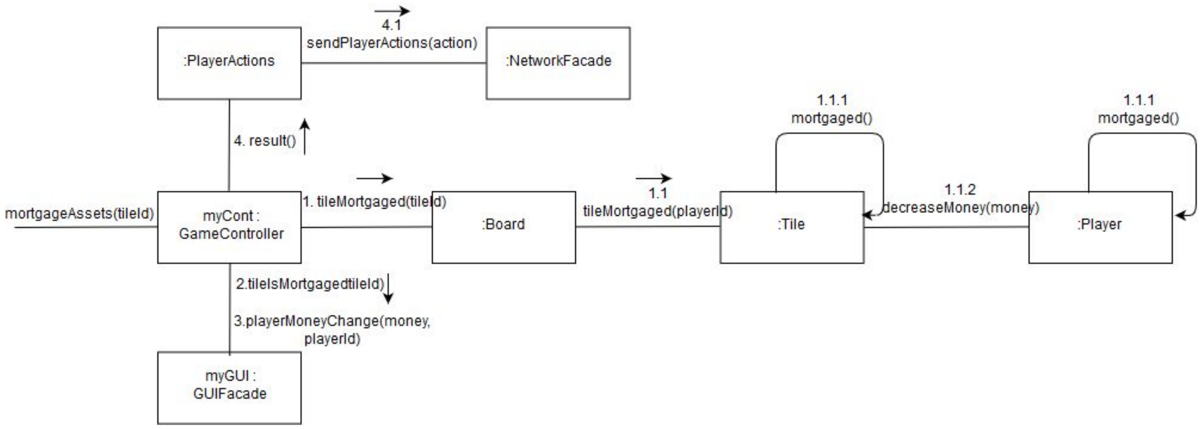
## Network


sendMessage(action)
Communication Procol
1. sendMessage(message)
NetworkClient
1.1. sendMessage(message)
MessageReceiveThread
1.1.1 receivedMessage(message)
CommunicationProtocol
1.1.1.1 sendToClients(message)
NetworkServer
1.1.1.1.1 sendMessage(message)
MessageReceiveThread
1.1.1.1.1.1. receivedMessage(message)
CommunicationProtocol
1.1.1.1.1.1.1 onNewPlayerAction(action)
PlayerActions
1.1.1.1.1.1.1.1. result()
GameController

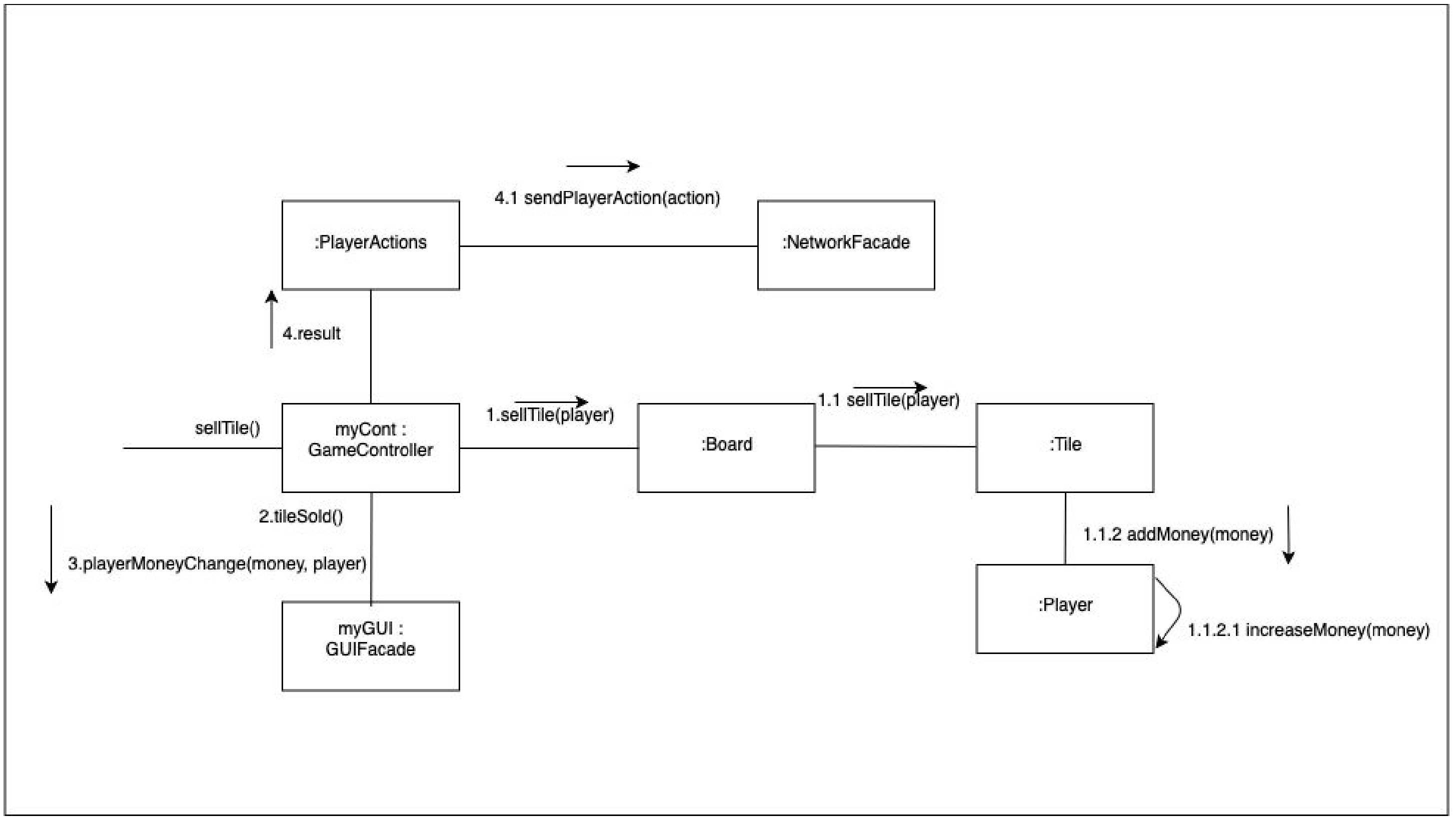
## getOutOfJail()



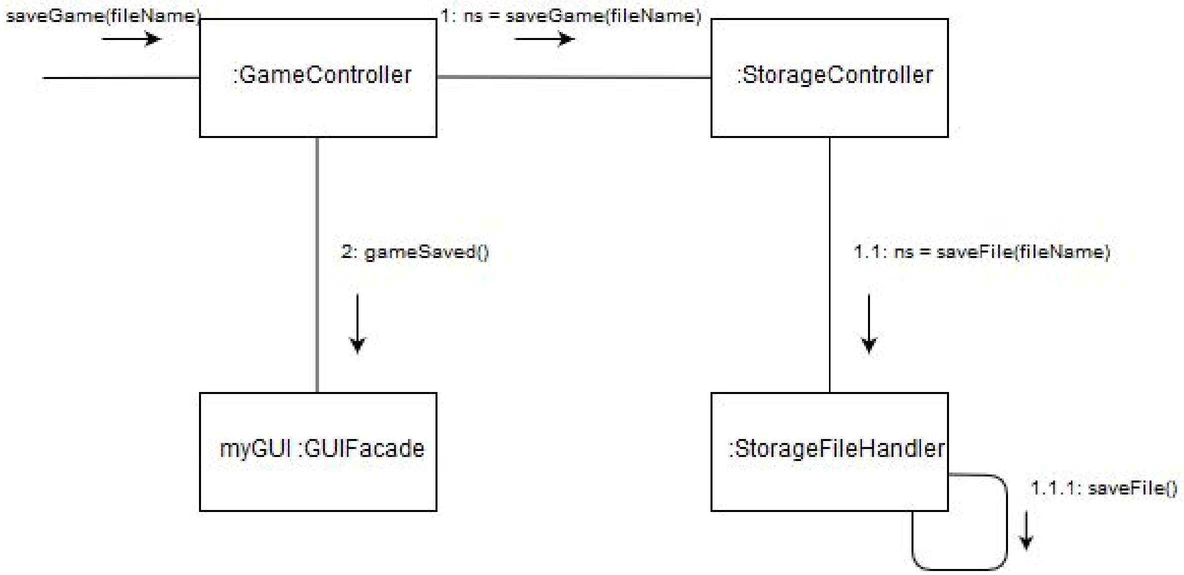
## mortgage()



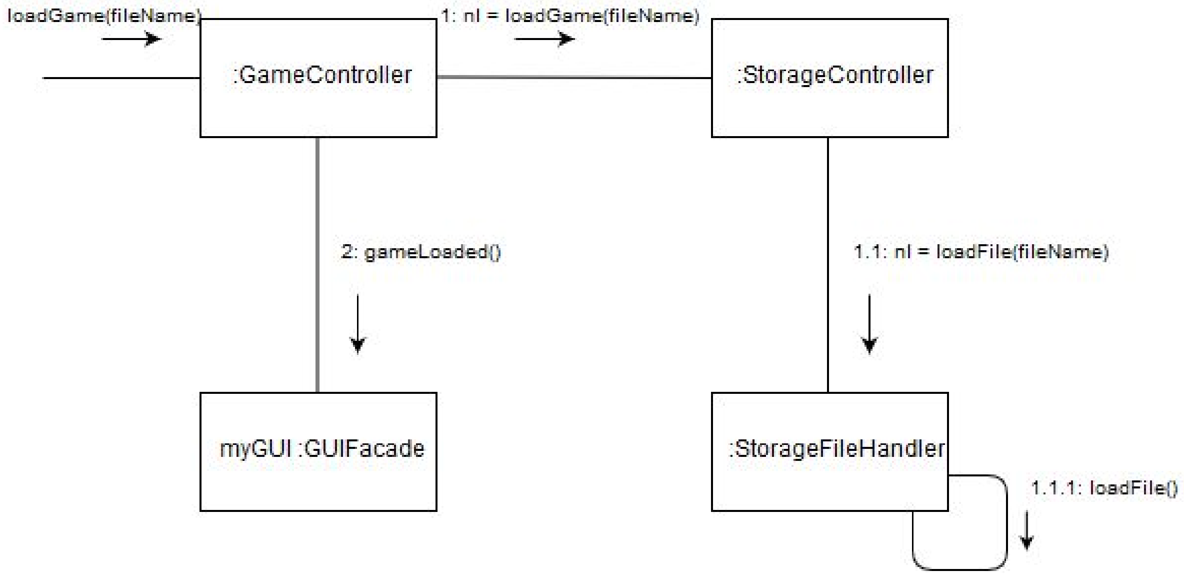
## sellTile()



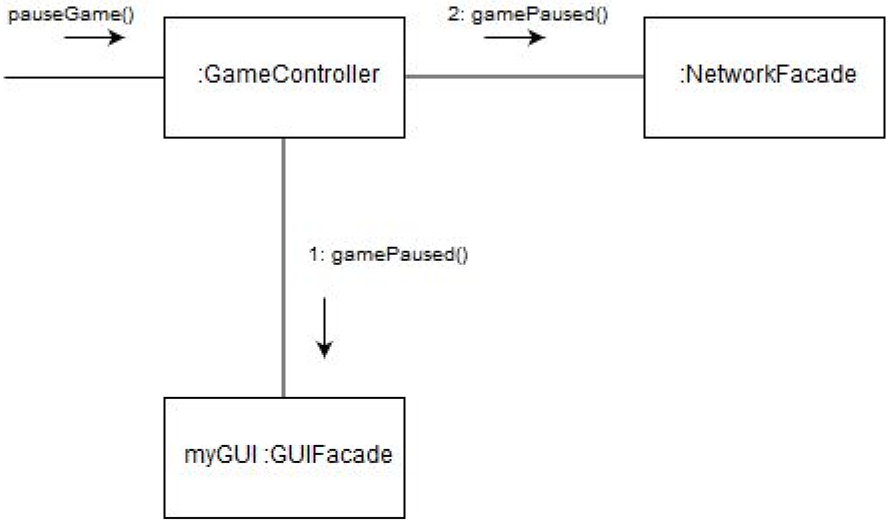
## saveGame()



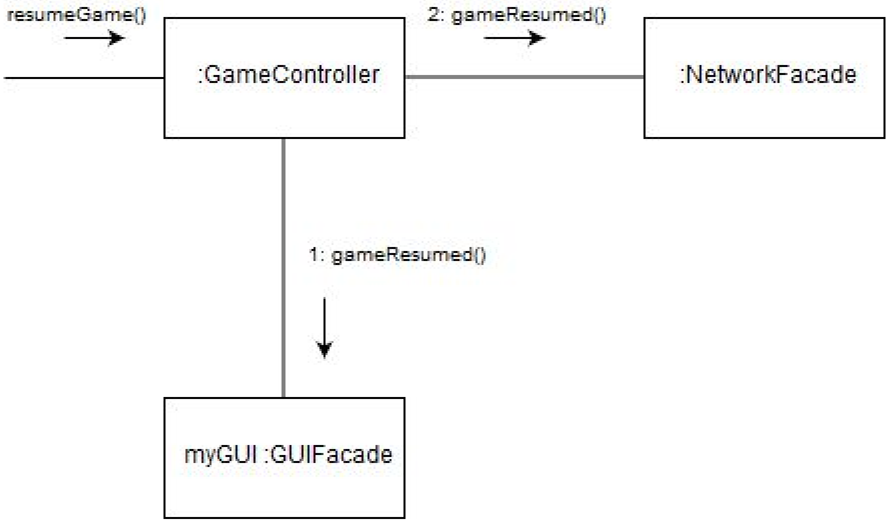
## loadGame()



## pauseGame()

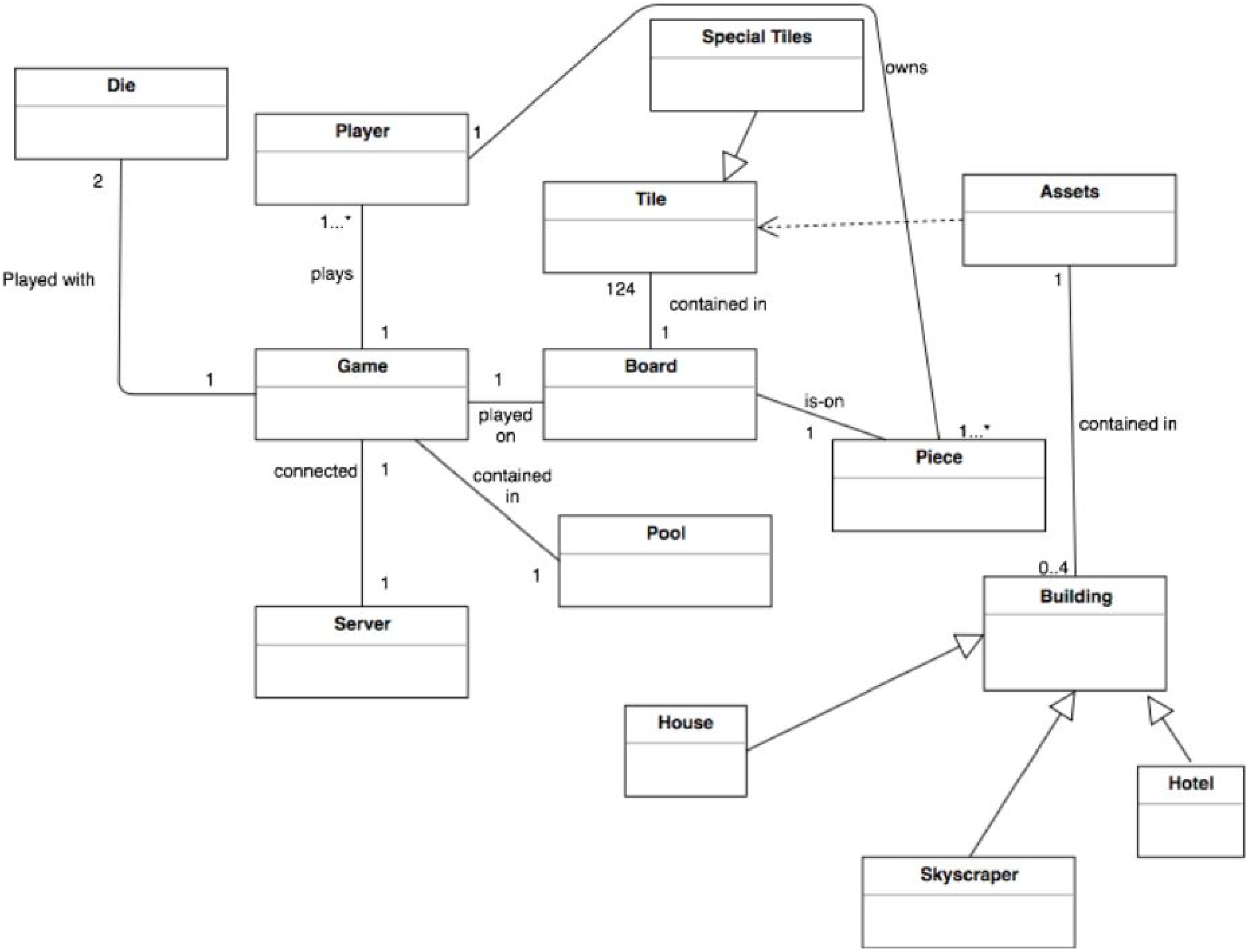


## resumeGame()



# 7) Domain Model

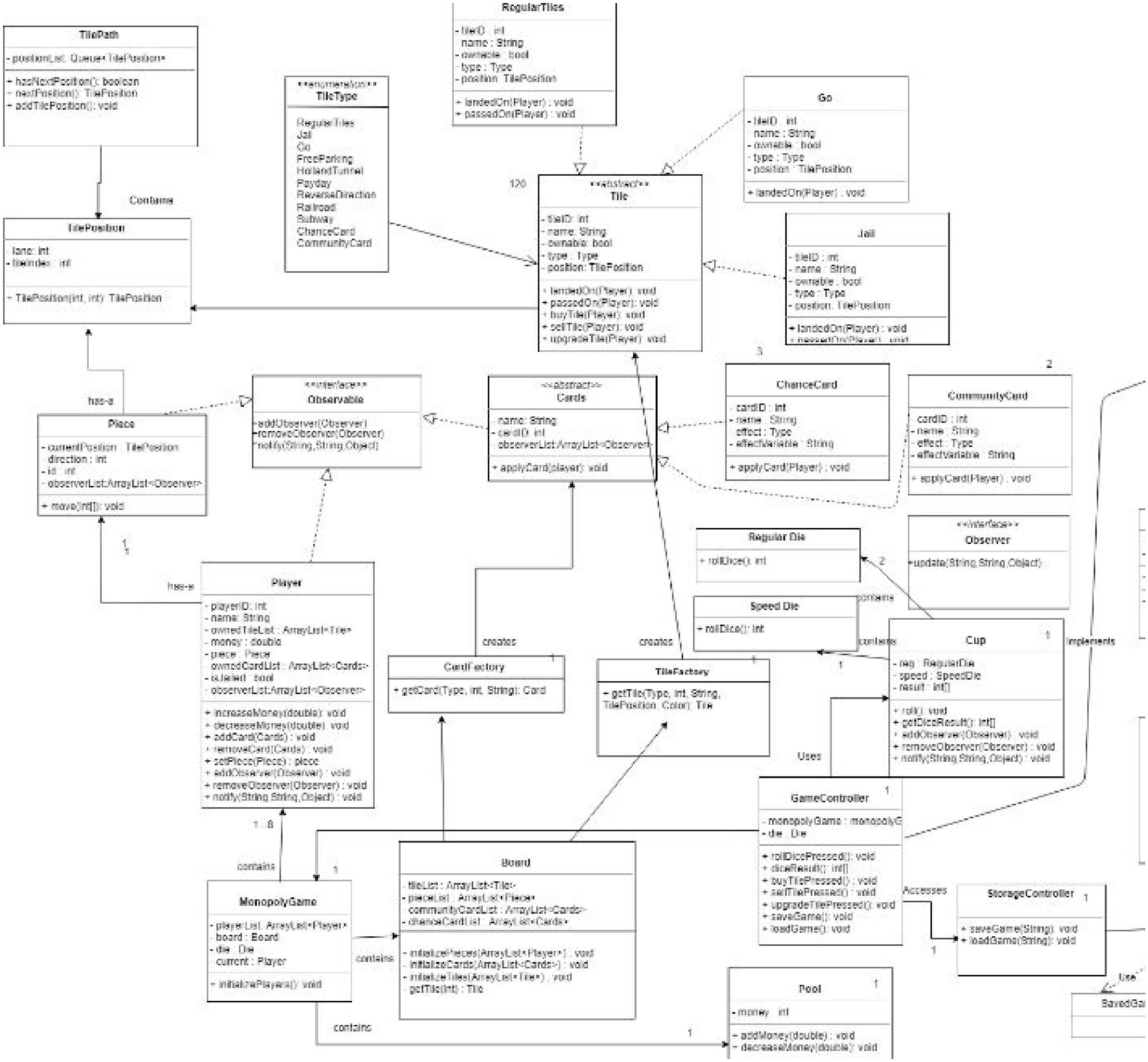
Just like before, after first making a domain model in Phase 1 we had to revise our model because we added new functionalities and features in Phase 2. The new domain model lets us see the class hierarchy in a concise way and we used this model to correct any wrongly implemented classes in the code.


Die
Player
Special Tiles
owns
Tile
Assets
Game
Board
Piece
Pool
Server
Building
House
Hotel
Skyscraper
Played with
2
1
plays
1...*
1
1
124
contained in
1
1
played on
1
is-on
1
1...*
contained in
1
0..4
connected
1
1
contained in
1

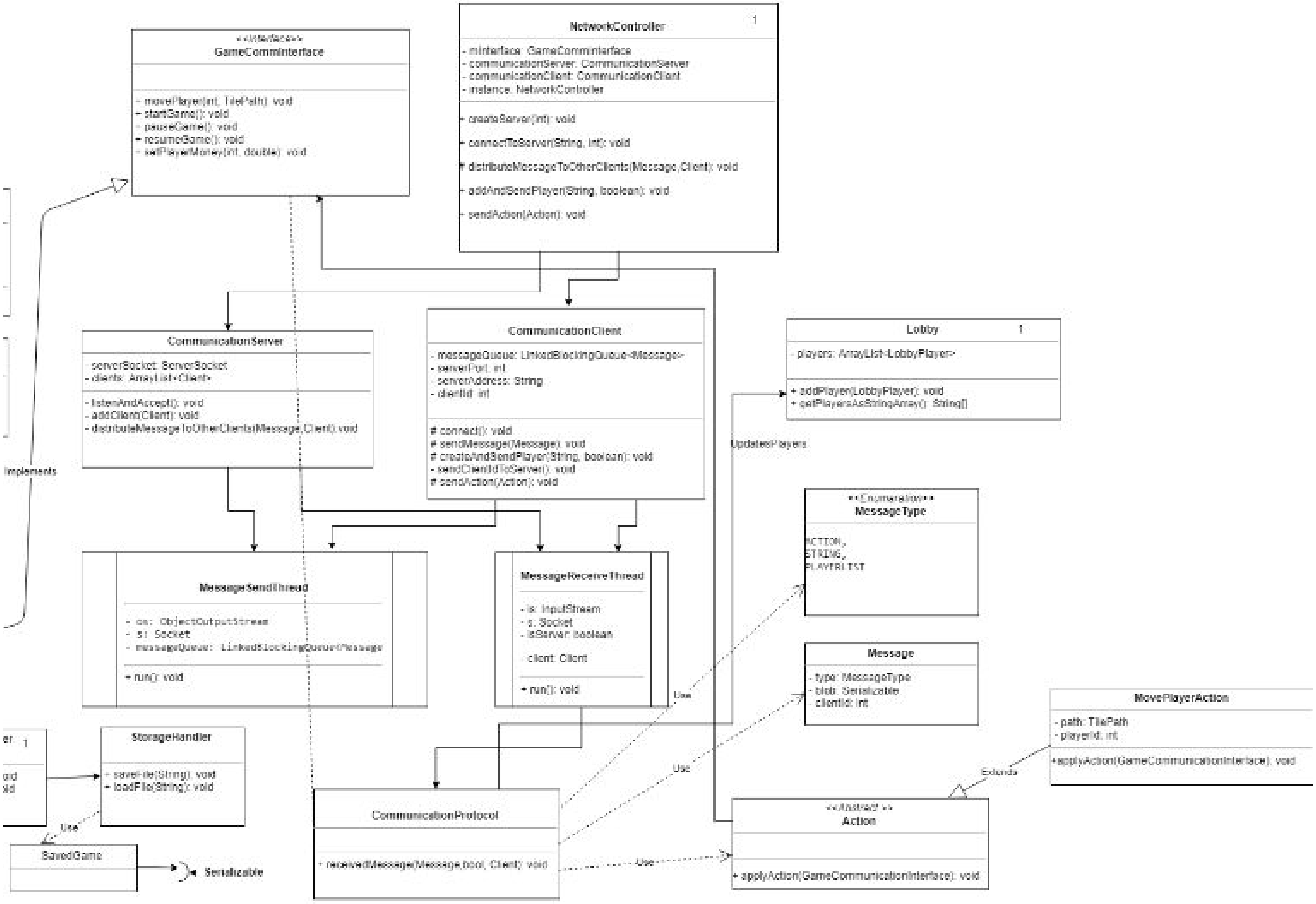
## 8) Class Diagram

This diagram, just like the domain model, was constructed earlier in the project but had to be updated to match our new features. In this diagram, we note the fields and methods of the classes so that when we start to implement them, we have a way of quickly checking if it was the correct way or not. Because of the sheer size of the image, we have decided to split it into two parts for this report, although originally they are connected.

# Domain Class Diagram


TilePath
- positionList: Queue<TilePosition>
+ hasNextPosition(): boolean
+ nextPosition(): TilePosition
+ addTilePosition(): void
Contains
TilePosition
- lane: int
- tileIndex : int
+ TilePosition(int, int): TilePosition
<<enumeration>>
TileType
RegularTiles
Jail
Go
FreeParking
HollandTunnel
Payday
ReverseDirection
Railroad
Subway
ChanceCard
CommunityCard
RegularTiles
- tileID : int
name : String
- ownable : bool
- type : Type
- position: TilePosition
+ landedOn(Player) : void
+ passedOn(Player) : void
120
<<abstract>>
Tile
- tileID: int
- name: String
- ownable: bool
- type : Type
- position: TilePosition
+ landedOn(Player): void
+ passedOn(Player): void
+ buyTile(Player): void
+ sellTile(Player): void
+ upgradeTile(Player): void
Go
- tileID : int
name : String
- ownable : bool
- type : Type
- position : TilePosition
+ landedOn(Player) : void
Jail
- tileID : int
- name : String
- ownable : bool
- type : Type
- position: TilePosition
+ landedOn(Player) : void
+ passedOn(Player) : void
has-a
Piece
- currentPosition : TilePosition
direction : int
- id : int
- observerList:ArrayList<Observer>
+ move(int[]): void
<<interface>>
Observable
+addObserver(Observer)
+removeObserver(Observer)
+notify(String,String,Object)
<<abstract>>
Cards
- name: String
- cardID: int
observerList:ArrayList<Observer>
+ applyCard(player): void
3
ChanceCard
- cardID : int
- name : String
effect : Type
- effectVariable : String
+ applyCard(Player) : void
2
CommunityCard
- cardID : int
- name : String
- effect : Type
- effectVariable : String
+ applyCard(Player) : void
1
1
has-a
Player
- playerID: int
- name: String
- ownedTileList : ArrayList<Tile>
- money : double
- piece : Piece
ownedCardList : ArrayList<Cards>
- isJailed : bool
- observerList:ArrayList<Observer>
+ increaseMoney(double): void
+ decreaseMoney(double): void
+ addCard(Cards) : void
+ removeCard(Cards) : void
+ setPiece(Piece) : piece
+ addObserver(Observer) : void
+ removeObserver(Observer) : void
+ notify(String,String,Object) : void
creates
1
CardFactory
+ getCard(Type, int, String): Card
creates
1
TileFactory
+ getTile(Type, int, String, TilePosition, Color): Tile
Regular Die
+ rollDice(): int
2
contains
Speed Die
+ rollDice(): int
contains
1
<<interface>>
Observer
+update(String,String,Object)
Cup
1
- reg : RegularDie
- speed : SpeedDie
- result : int[]
+ roll(): void
+ getDiceResult(): int[]
+ addObserver(Observer) : void
+ removeObserver(Observer) : void
+ notify(String,String,Object) : void
implements
Uses
1..8
contains
1
MonopolyGame
- playerList: ArrayList<Player>
- board : Board
- die : Die
current : Player
+ initializePlayers(): void
contains
Board
- tileList : ArrayList<Tile>
- pieceList : ArrayList<Piece>
communityCardList : ArrayList<Cards>
- chanceCardList : ArrayList<Cards>
- initializePieces(ArrayList<Player>) : void
- initializeCards(ArrayList<Cards>) : void
- initializeTiles(ArrayList<Tile>) : void
- getTile(int) : Tile
GameController
1
- monopolyGame : monopolyG
- die : Die
+ rollDicePressed(): void
+ diceResult(): int[]
+ buyTilePressed() : void
+ sellTilePressed() : void
+ upgradeTilePressed(): void
+ saveGame(): void
+ loadGame(): void
Accesses
1
StorageController
1
+ saveGame(String): void
+ loadGame(String): void
Use
SavedGa
contains
1
Pool
1
- money : int
+ addMoney(double) : void
+ decreaseMoney(double): void

# Network Class Diagram



<<Interface>>
GameCommInterface
- movePlayer(int, TilePath): void
+ startGame(): void
- pauseGame(): void
+ resumeGame(): void
- setPlayerMoney(int, double): void
NetworkController
1
- mInterface: GameCommInterface
- communicationServer: CommunicationServer
- communicationClient: CommunicationClient
- instance: NetworkController
+ createServer(int): void
+ connectToServer(String, int): void
# distributeMessageToOtherClients(Message,Client): void
+ addAndSendPlayer(String, boolean): void
+ sendAction(Action): void
CommunicationServer
- serverSocket: ServerSocket
- clients: ArrayList<Client>
- listenAndAccept(): void
- addClient(Client): void
- distributeMessageToOtherClients(Message,Client):void
CommunicationClient
- messageQueue: LinkedBlockingQueue<Message>
- serverPort: int
- serverAddress: String
- clientId: int
# connect(): void
# sendMessage(Message): void
# createAndSendPlayer(String, boolean): void
- sendClientIdToServer(): void
# sendAction(Action): void
Lobby
1
- players: ArrayList<LobbyPlayer>
+ addPlayer(LobbyPlayer): void
+ getPlayersAsStringArray(): String[]
UpdatesPlayers
Implements
MessageSendThread
- os: ObjectOutputStream
- s: Socket
- messageQueue: LinkedBlockingQueue<Message
+ run(): void
MessageReceiveThread
- is: InputStream
- s: Socket
- isServer: boolean
- client: Client
+ run(): void
<<Enumeration>>
MessageType
ACTION,
STRING,
PLAYERLIST
Message
- type: MessageType
- blob: Serializable
- clientId: int
MovePlayerAction
- path: TilePath
- playerId: int
+applyAction(GameCommunicationInterface): void
Extends
er 1
oid
oid
StorageHandler
+ saveFile(String): void
+ loadFile(String): void
Use
SavedGame
Serializable
Use
Use
CommunicationProtocol
+ receivedMessage(Message,bool, Client): void
Use
<<Abstract>>
Action
+ applyAction(GameCommunicationInterface): void

# Testing

## Introduction

After we were done with the most crucial parts of the code, we decided to do some testing so as to not continue building on some erroneous code from before. Fortunately, we did not have to replace any code because we were careful in our implementation due to our extensive use of the diagrams. We selected some integral classes and used three different types of testing in order to test them; glassbox testing, blackbox testing and repOK testing.

## Test Objectives

With the tests, our objective was to find any faulty code that is not apparent and fix them before it becomes too engraved in the code. Identifying the errors early on is beneficial in this way; you do not need to change as much as when you test for errors at the last possible time.

## Scope

As our scope, we selected some of the most used classes in our code. We decided to not include everything in our tests because some methods were trivial and we do not have time to test everything for this project. The classes we have selected are as follows:

1) Board
2) Tile
3) Cup
4) Player

## Test Strategy

After the class selection was done, we started work on what to test and how to test it. When selecting what to test, we tried to emphasize the more complex methods, not methods such as getters and setters. We have done three different type of tests as said in the introduction. RepOK tests were used in testing whether we construct our objects the way they are intended. For blackbox testing, the important part thing was that the methods return the expected output. If there were any errors in the output, glassbox testing was needed to find the specific code where the error occurs.

## Methods Tested

These are the methods we selected to test as we thought they were the most important ones:

### Tests for Board

1.1 testInitializeTiles() - 2 Tests

First we initialized the tiles using initializeTiles() method, then by calling the tileList field of Board we see that the tiles were initialized correctly.

1.2 testInitializePieces() - 2 Tests

We created two distinct players and a player list where we then put the players in. (We needed to initialized players to be able to create the corresponding pieces. Upon creating the piece list and initializing the pieces, being able to call the pieces using get() method from the piece list we created showed that our piece initialization works well.

1.3 testInitializeCards() - 2 Tests

After calling the initializeCards() method from board class we checked the existence of the cards inside the list by calling isEmpty on the communityCardList and chanceCardList inside board class.

1.4 testGetTile() - 2 Tests

We initialized the tiles to be able to acquire the tile we wanted to reach. When we call the getID of the getTile() method from Board we see that we get the id correct.

1.5 testBoard() - 2 Tests

Two players and their corresponding playerlist were created to be able to test the methods regarding playerlist inside our board class. When we try to call such methods such as initializeTiles() initializeCard() etc. from board we see that their initialization suffices.

## Tests for Tile

2.1 testLandedOn() - 2 Tests

We created a test tile typed jail and a player. We then set the player's position to that created jail tile. Upon testing whether the player isJailed(), by calling isJailed() method we noticed that the player is jailed and we also got the the position correct when we call getCurrentPosition() method on the player therefore we saw that he teleport() method works fine.

2.2 testTeleport() - 2 Tests

We created a test tile typed jail and a player. And teleported the player to that jail tile using teleport() method. Upon testing whether the player isJailed(), by calling isJailed() method we noticed that the player is jailed and also got the the position correct when we call getCurrentPosition on player  therefore teleport() method works fine.

## Tests for Cup

3.1 testRoll() - Multiple Tests

For this test, we wanted to see whether the dice rolls returned an integer and also be within a certain range. Because of the randomness included in this method, we opted to test it 1000(!) times to minimize the randomness.

3.2 testAdd() - 2 Tests

In this test, the observability of the dice is tested. By creating a testClass in Cup which implements Observer, we try to add testClass objects to the observerList of the dice.

3.3 testRemove() - 2 Tests

Just like adding an Observer, with this test we can see if we actually get to remove the Observers from the observerList.

## Tests for Player

4.1 testAddProperty() - 2 Tests

To be able to check the addProperty() method we required to have a player and some tiles (in particular in our case two tiles tile1 and tile2). After the instantiation of all these we applied the addProperty() method on player and added tile1 and tile2 to player's tile list. Upon checking player's tile list with getOwnedTileList() we see that player had both tile1 and tile2.

4.2 testRemoveProperty() - 2 Tests

Opposite of the adding property tests, we tried removing the tiles this time with removeProperty(). Both times it worked and getOwnedTileList() returned correct.

4.3 testIncreaseMoney() - 2 Tests

For this one, we used increaseMoney() on a player and then checked their money, which returned correctly.

4.4 testDecreaseMoney() - 2 Tests

For this test, just like increasing money, we used decreaseMoney() and observed the results, which were correct.

4.5 testAddCard() - 2 Tests

With this test, we wanted to see if adding a card to a Player's inventory worked or not. To do this, after initializing the players we call addCard() and check whether the list is updated or not.

4.6 testRemoveCard() - 2 Tests

Like the above, after adding the cards we tried to remove them for this test, which was successful because Player's cardList returned as expected.

# DISCUSSION OF DESIGN ALTERNATIVES AND DESIGN PATTERNS AND PRINCIPLES

As Euler's Disciples team, we were aware of the fact that the design patterns and principles are one of the major outcomes of the lecture therefore we focused on them and tried to optimize our implementation with those. Our goal was to choose most appropriate ones for particular situations. We forced ourselves to think about how we can benefit from the advantages of them and discussed many times their pros and cons with the discussion of correct implementation of them. In this text, our choices with their reasons will be stated.

First of all we needed to determine the responsibilities, roles and collaborations. For the **Creator** pattern, we were aware the fact that containers should creates the items contained in them. For example, in our design MonopolyGame creates Player and Card objects. Another pattern which also seems trivial to us is **Information Expert**. Whenever we assign responsibility to a class, we consider the classes which have required information as candidates. For instance, we need to update the "owner" attribute of the tile, we need to reach Tile object. Board fulfills the responsibility of giving reference to the Tile object since it has necessary information.

We considered **High cohesion** principle to avoid having a class which has too many different kinds of responsibilities. Division of the responsibilities was crucial to us to manage classes easily. **Low coupling** is our another concern in order to reduce dependencies. For instance for payRent() method, Player's decreaseMoney() method needs to access tile to get the price. If Player has a method called getPrice, it will cause higher coupling.

We have **Controller** for low impact of changes. Our controller represents the game and communicate with GUI. In this way, we also hide the details. We applied **Factory** pattern for cards and tiles since there are different types of tiles and cards such as RegularTile with Railroad and RollThreeCards with ChangeCards. With the helper objects, we seperated the responsibilities. Card class is an abstract class and has subclasses with different properties. Additionally, GameController, NetworkController, CardFactory and TileFactory are instances of **Singleton** objects since we need to access them from everywhere and they need to be created as a single object.

Moreover, **Observer** pattern is heavily used in our project. GameFrame class is observer of many classes such as MonopolyGame, Player, Cup, Card since GameFrame needs to be updated when a player's money, assets, location change. For the simulation of the players on board, we have PieceGraphics class in UI, which will change the position when corresponding Piece object's location changes. So observable classes implements Observer interface and notify their observers whenever a change occurs. In addition, for different bot behaviors, we use **Strategy** pattern. One of the three behaviors will need to be selected and we will choose the best one in particular time.

Our design has landedOn() method which is a great example of **Polymorphism**. It is an abstract method and each tile has it. Since actions that will be taken after landing on each tile are not same, we override this method whenever it is needed. For example, landedOn() method in a regular tile, e.g. avenues, and landedOn() method in railroad is slightly different because instead of paying rent, player changes lane when landed on railroad.

# Supplementary Specification

This part of our final report is used to define and describe the non-functional requirements of our project. These requirements, together with the use cases, show the full requirements of the project.

## *Functionality*

### -Security

Random events will be performed by the server and the communication between the server and the players will be encrypted in order to minimize the cheating risk. By using a dedicated server instead of a peer to peer model, we will be protecting the players from sharing their IP with the other players, which could be problematic if any of the users had malicious intentions.

### -Online Play

The game will support multiple users over multiple devices via a network connection. Short connection failures will not affect the player or the game in a problematic manner.

### -Action logging

The game server will log the actions taken the players which allows for players to view previous actions and disconnected systems to catch up with the action.

### *Usability*

#### -Ease of use for disabled gamers

The game will use a color scheme that allows the color blind players to enjoy the game without any setbacks. Not having any inputs that require precise button presses allows for players with motor dysfunctions to enjoy our game as well.

### *Reliability*

#### -Recoverability

Our action logging feature coupled with reconnect feature will ensure minimal disruptions from network problems.

### *Performance*

Game runs fast even on slow computers because it is highly optimized. As a low-weight Java game, our game will run without any performance issues on every device we support.

### *Supportability*

#### -Adaptability

Our game can run on most of the common commercial OSs as it runs on JVM which is available on over 2 billion devices around the world.

#### -Configurability

As not every computer is created equally, some players could require higher or lower graphical resolution options. Depending on the course of the development, we could support such changes.

Some networks do not allow for its users to connect to certain IPs or support DNS resolution to some domains. Some people might also want to disguise their internet activity online.

### *Implementation constraints*

Due to only being allowed to use Java Swing libraries for graphics, we will not be able to create fancy 3D graphics or animations.

### *Legal Issues*

We are probably not allowed to release and monetise our clearly superior game to the public due to copyright issues.

# Glossary

This last part of the final report is the glossary; words and phrases encountered in the implementation of the game and their meanings. The usefulness of this glossary comes from the fact that not everyone understands the same things when they hear the phrases mentioned here; everyone must refer here to be on the same page for the definitions.

| Name | Description |
|---|---|
| Player | The ones who play the game, the actors; human or AI. |
| Character | Indicates where each player are on the board. |
| Lobby | Online room where the players wait for everyone to get ready. |
| Dice | Two 6-sided standard non-weighted dice are used. |
| Speed Die | A special 6-sided die which have different sides not containing numbers. |
| Assets | Everything a player owns; tiles, houses, hotels, etc. |
| Money | The currency used in the game to make all transactions. |
| Tile | The spaces on the board which the player's character uses to move around; they can be anything ranging from tiles to railroad stations to special tiles. |
| Owned Tile | A tile which the current player owns. |
| Foreign Tile | A tile which is owned by anyone but the current player. |
| Unowned Tile | A tile which has no owners. |
| House | If the majority rule is satisfied on any color of land the player owns, they can decide to build houses to increase the rent they recieve. |
| Hotel | Only buildable after already building 4 houses on each tile you own on a specific color. Replaces all the houses. |
| Skyscrapers | Only buildable after already building a hotel on each tile you own on a specific color. Requires the player to have monopoly. |

| | |
|---|---|
| Majority Ownership | The player has a majority ownership whenever they own all but one tile of a specific color. |
| Monopoly | Occurs whenever a player owns all the tiles of a specific color. |
| Buying | Buying refers to the use of Monopoly money to earn assets; the seller gains the money whether its the bank or a player. |
| Selling | Gaining money by giving up one of your assets. |
| Rent | Whenever a character steps on an owned tile which are not theirs, they have to pay a rent (money) to the owner according to the number of property built on the tile. |
| Bank | The non-playable character which gives the initial money and  takes money when players buy unowned tiles. |
| Bankruptcy | Happens when a player loses all their money as well as their assets; also means that the player has lost the game. |
| Jail | The tile the character moves into whenever they are sent to the jail; be it from a special card or rolling double thrice. |
| Railroad / Transit Stations | Acts just like an owned tile; the player has to pay a rent if the tile is owned by someone else, depending on their dice rolls. Also allows the moving from one lane to another. |
| Community Chest Tile | This tile makes the player draw a card from the community chest card pile and play according to the card. |
| Chance Tile | Similar to the community chest tile, the player draws a card from the chance action cards and plays according to the card. |
| Roll Three | Upon landing on this tile, everyone gains money depending on how much the rolled dice and the roll-three cards (which are received at the start) match. |
| Tax Refund | Any money paid to the bank, except from buying tiles, are placed in a “pool” which the player collects if he/she lands on a tax refund tile. |
| Subway | This tile allows the player to move to any tile they want in their next turn. |
| Pay Corners | These tiles have the player receive money from the bank. |
| Holland Tunnels | This tile allows the player to instantly move to any other Holland Tunnel tile, bypassing everything between them. |

| | |
|---|---|
| Utilities | These tiles are just like properties; the player can buy them and if they land on an owned utility they have to pay a rent. |
| Mortgaging | The player can opt to receive half the value on any tile they own, temporarily disabling the tile and the tile does not activate until the player returns the money they received. |
| Rolling Double | If the player rolls the same number on the two standard dice, they get to roll again. Rolling double thrice sends the player to the jail. |
| Rolling Triple | If the player rolls triple 1’s, 2’s or 3’s they can choose which tile they want to move. They do not roll again after this. |