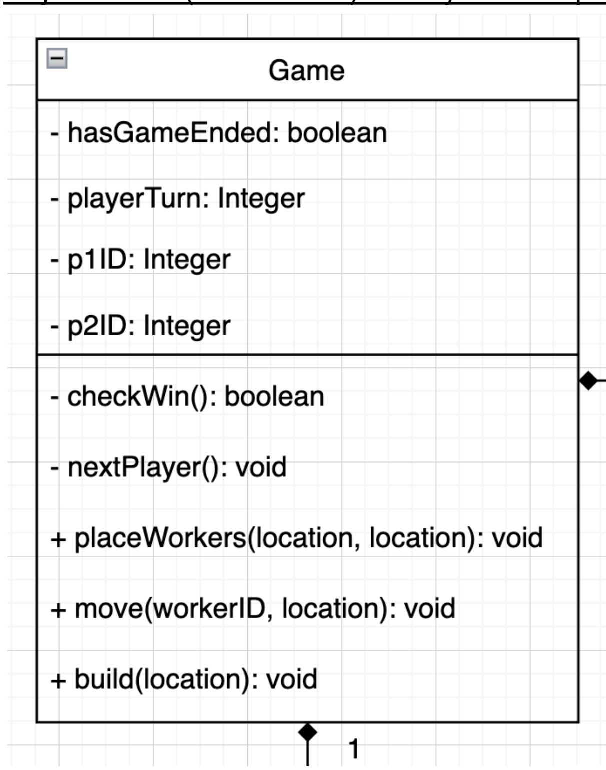
1. How can a player interact with the game? What are the possible actions?

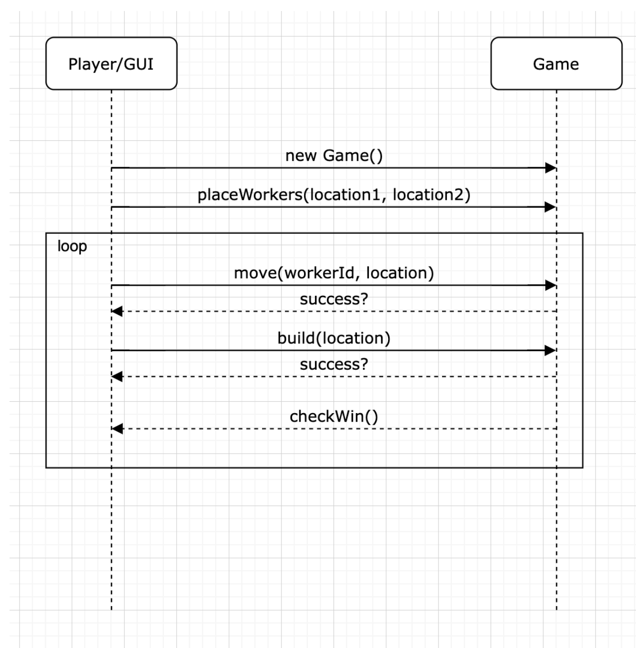
My implementation assumes that the users will be switching off as p1 and p2 when executing game methods and playing the game (order is p1, then p2, repeat).

The player may begin the game by creating a new Game class. Starting with p1, players may place their workers. For each round, a player will be able to move a worker to an unoccupied, adjacent, climbable location and then build a tower/dome at an unoccupied, adjacent location during their turn. After each turn (after the player has a successful build), the game checks if the current player has won the game. If so, the game is over, otherwise, the game will switch it to the other player's turn.

The users' defined actions exemplify the design principle of a low representational gap, since the actions are exactly what the instructions allow the user to do. Therefore, this accomplishes the design goal of understandability and change. To be specific, the methods the user can call to play the game are placeWorkers(loc1, loc2), move(workerID, loc), and build(loc). Methods like nextPlayer() and checkWin() are automatically called by the Game object, and the results are printed to the console for the user to see. When there is user input error (out-of-bounds location, etc), the method will not execute any actions or change the state, instead returning with an error message printed to console. This way, the user can call the method again with valid inputs. All of this accomplishes good abstraction, so the user does not need to know the details of the program to play the game.

<u>Object Model (Game class) and System Sequence Diagram</u>





2. What state does the game need to store? Where is it stored? Include the necessary parts of an object model to support your answer.

The game has attributes hasGameEnded and playerTurn to track the game status and current player. It also stores the grid, represented in a Grid class, with attributes gridState and occupiedFields. The grid itself is represented by a 2d matrix of tower objects. Each tower can have 1 location object (row and col of the grid) and also information about its level and if it has a dome. For efficiency, the Grid class also has attribute occupiedFields which represents locations that workers cannot move to. This includes locations of towers with domes and locations that workers are currently occupying. In addition to the grid, the game also stores the Player objects in a dictionary, where the players can be accessed by their ID. Similarly, each player stores their Worker objects in a dictionary, where they can also be accessed by their ID. For ease, a player stores all their worker objects' positions (Location objects) in a dictionary as well. Both players and workers have id attributes, and the workers store their current position. The game also can get the currentPlayer with a method call, and the winner is the result of currentPlayer when hasGameEnded is true.

The game, grid, tower, location, player, and worker classes are separated to use the design principle of high cohesion and to accomplish design goals of understandability, reuse, and change. Generally, this separation is more understandable, and if any class was changed to add features, those changes would not have drastic effects on other classes.

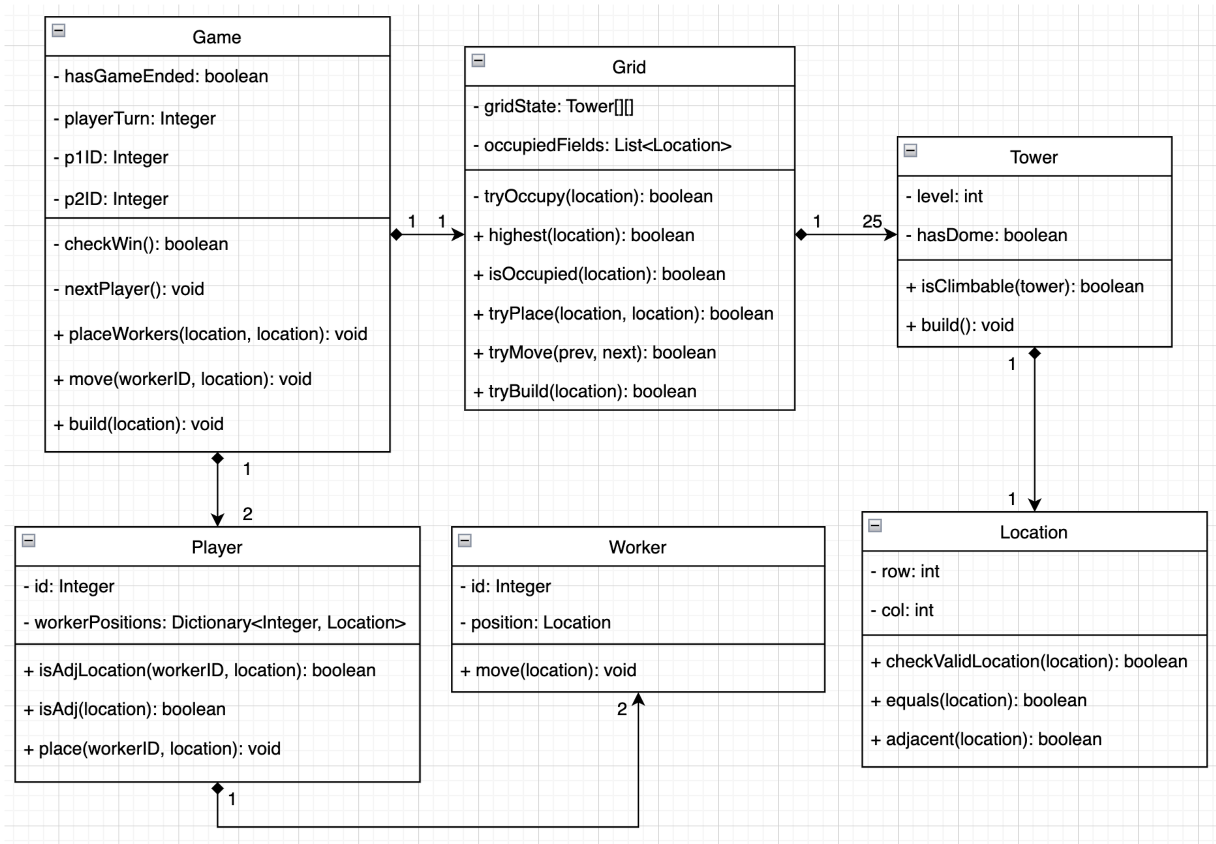
I made the design choice to implement a Player dictionary for the Game and a Worker dictionary for each Player, for the design goal of extensibility. This way, implementing more than 2 players or more than 2 workers in the future is much easier.

Additionally, I chose to design occupiedFields as one attribute instead of splitting into separate domeFields and workerFields, since they have similar functionality where a worker cannot move to that field. This design choice was made for the design goal of understandability and reuse, since there will be less attributes cluttering the Grid class and the isOccupied function can be used in many different contexts for the game.

Compared to the domain model, there is no Dome Class in the object model. This was a decision since the dome is not a significant enough concept to warrant an extra class. Instead, domes are implemented as fields for the Tower object.

Although the Location and Tower objects are closely linked, they were implemented as separate objects because locations can be used by most other classes, while towers are mainly only used in the Grid class. This is also more intuitive for the user, which accomplishes design for understandability.

Object Model parts (All Object Classes)



3. How does the game determine what is a valid build (either a normal block or a dome) and how does the game perform the build? Include the necessary parts of an object-level interaction diagram (using planned method names and calls) to support your answer.

First, the game does some beginning checks: if the game is over and if the location is invalid. If either of these is true, the build method will return with an error message.

To determine a valid build, the game uses the player method isAdj(location) to determine if location is an adjacent location to either worker of the player. If not, the method returns with an error message. Otherwise, the game calls the grid method tryBuild(location). The grid object's tryBuild method uses isOccupied(location) to check if the location is not occupied. Occupied means the location does not have a domed tower or a worker currently occupying it. If it's occupied, the tryBuild will return false.

Otherwise, the build will be performed. The tower method build() is called, which increases the level of the tower at that location by 1. The exception is when the level is already 3; in that case, the tower's hasDome attribute is turned to true. Afterwards, the tryBuild method checks if a dome has been added to the tower with tower method isDomed(), and if so, it adds the location to occupiedFields since it is now occupied.

Back in the game's build method, if tryBuild(location) returns false, the method returns with an error message. Otherwise, the build was successful. After, the build method calls checkWin() to see if the current player has won on this turn, and if not, it switches to the other player's turn. Else, it logs a winning message to the console.

I decided to incorporate the dome into the tower class, since the dome is part of the tower. Therefore, the design goal of understandability is accomplished because keeping them together is more intuitive. In addition, I decided to put the build method in the Tower class, since this incorporates the design principle of low coupling, as building a block/dome does not depend on attributes in other classes such as Grid or Location. The other necessary methods (player.isAdj(), grid.tryBuild(), grid.isOccupied(), tower.isDomed(), etc) were implemented in their respective classes for the same reason and that they're in the class most suited for their purpose. This accomplishes the design goals of reuse and change with a more modular approach.

Object-level Interaction Diagram