Justification

1. A player can interact with the game by entering the initial positions to initiate the game and doing possible actions like moving a worker and building around that worker. As the object model shows, the player can call move and build block or dome methods in the Game class by passing the coordinate values, the player's index and the worker's index, which can both be either 0 or 1.

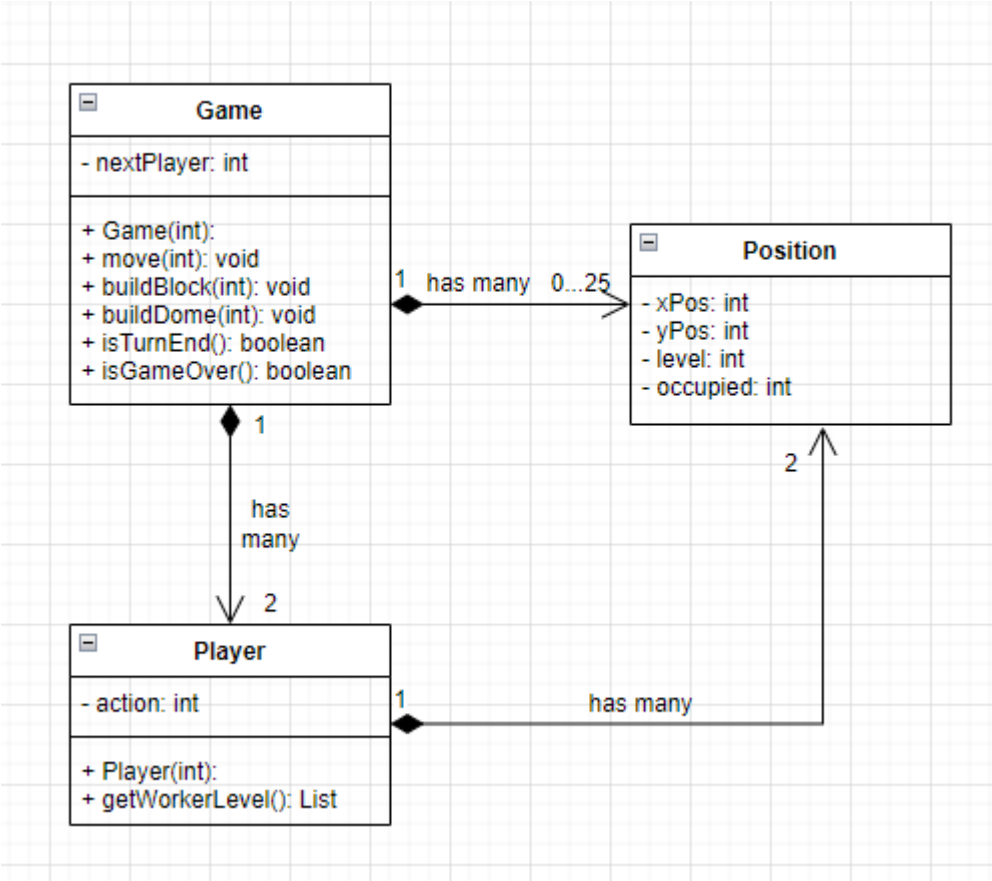

Figure 1. Object Model (some methods for validation, setter and getters are omitted)

2. The game needs to store a state indicating which player is taking the turn and a state indicating if the player has finished all the actions. As the object model shows, the first state should be stored in the Game class, which is an integer called nextPlayer indicating who's the next player. It will change

to 1 – nextPlayer to take turns between 0 and 1. The second state should be stored in the Player class, which is an integer called action indicating if the player has finished one action, two actions or zero action.

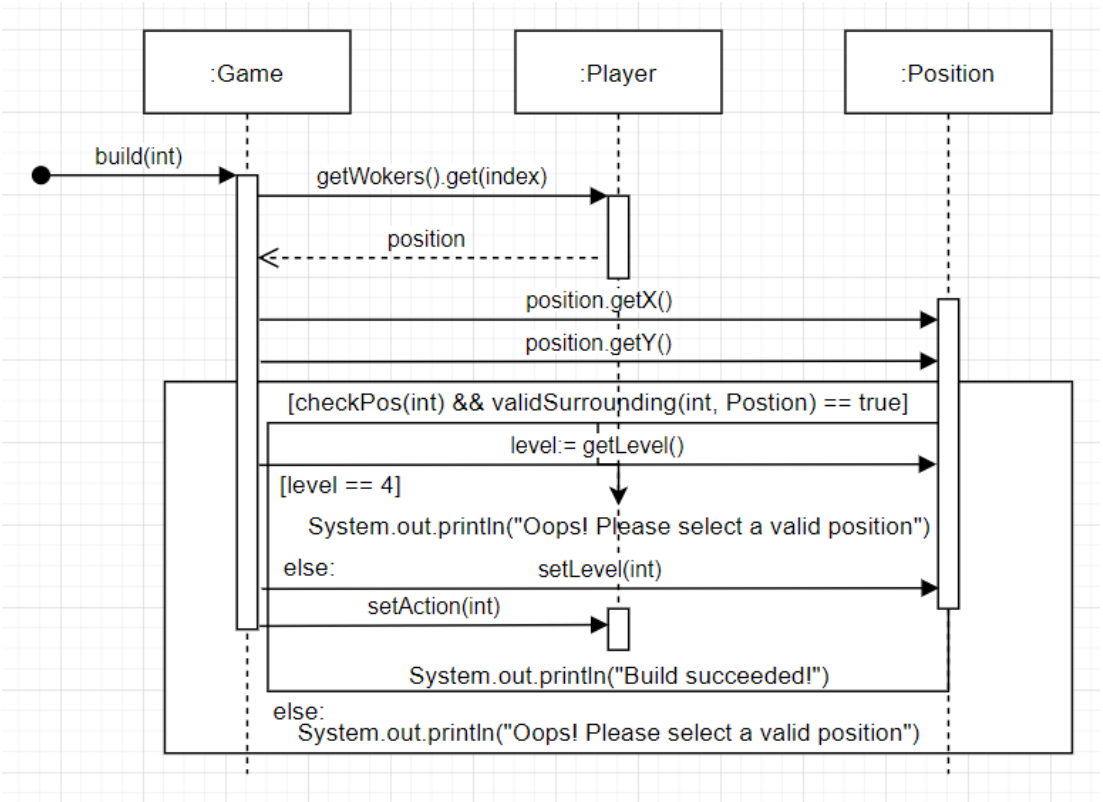

Figure 2. Interaction Diagram for a general build

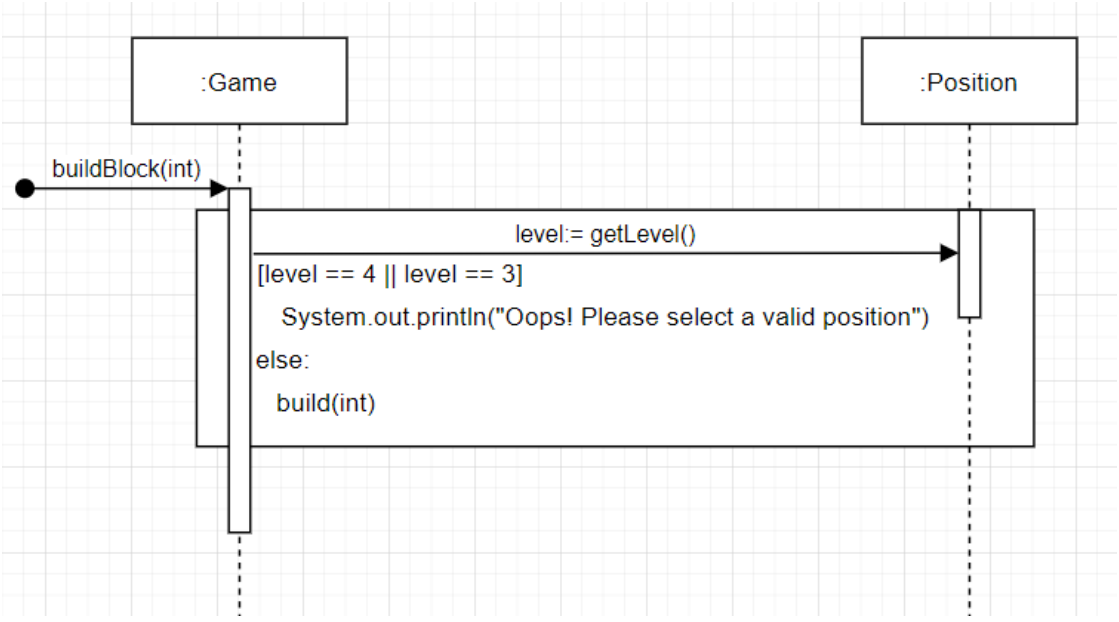

Figure 3. Interaction Diagram for build a block

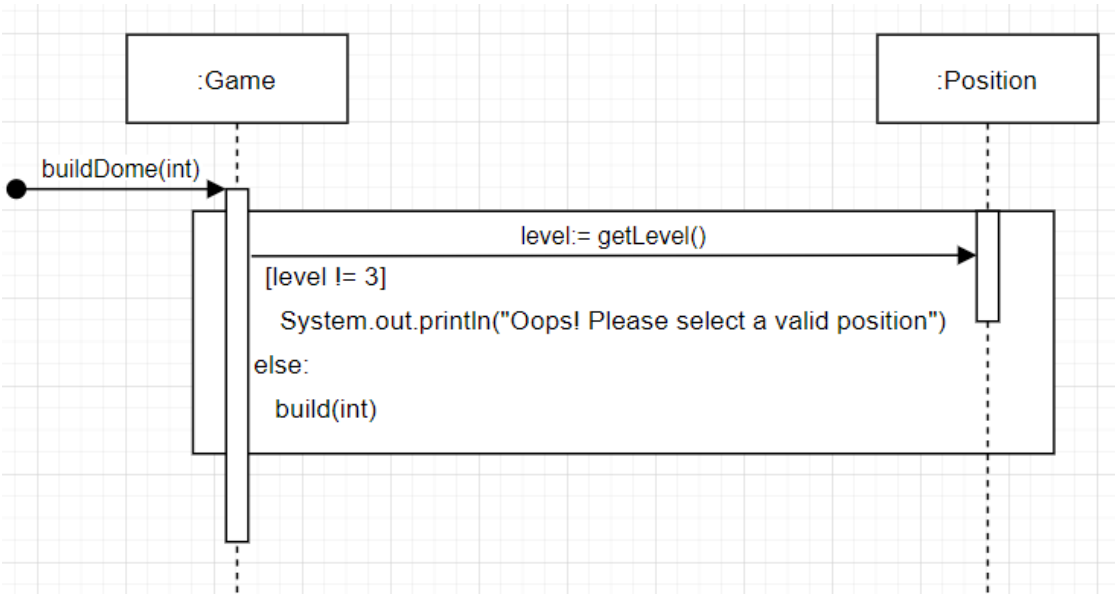

Figure 4. Interaction Diagram for build a dome

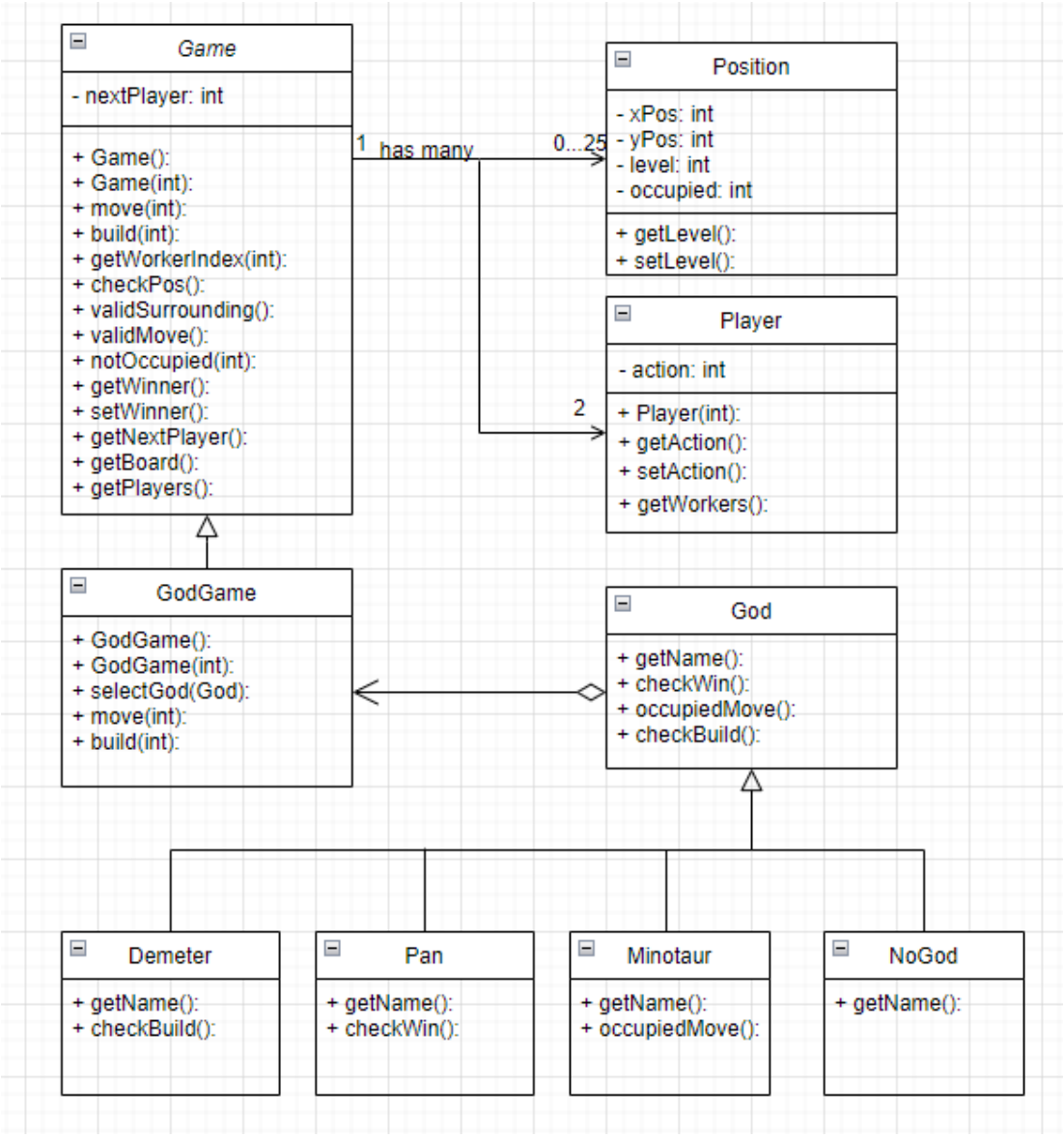

Figure. Updated object model

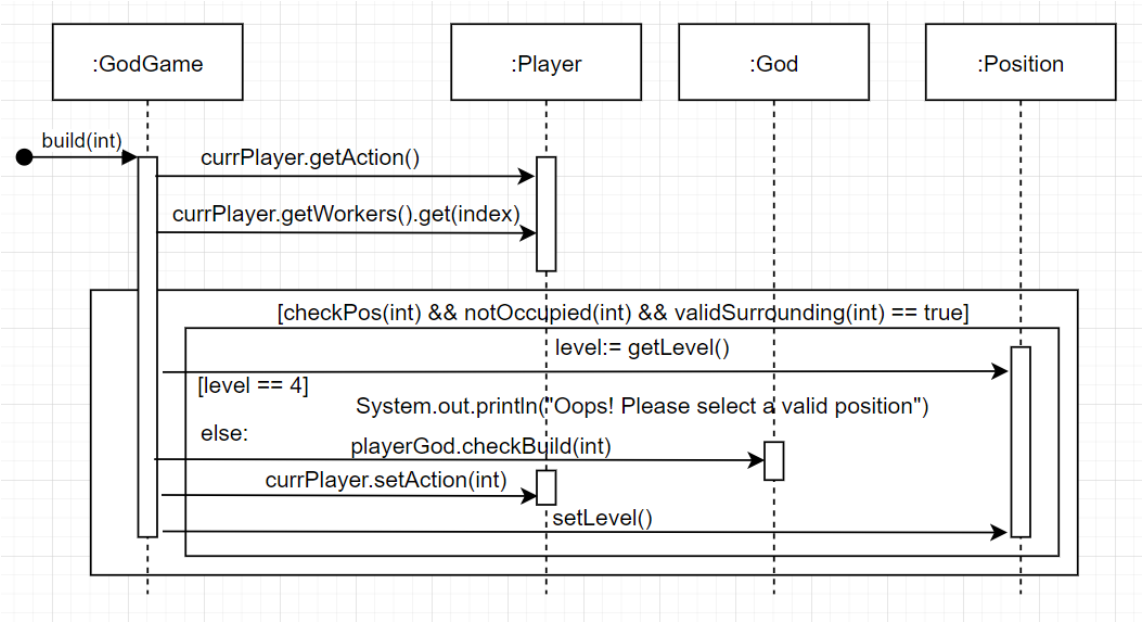

Figure. Updated interaction diagram

3. (Updated) The game determines if a build is valid or not by checking various requirements. To follow the principal of better maintaining the code and achieving high cohesion, the checking functionalities are separately written into different single methods. The code, having the worker index of the player, will get the selected worker's position and check if the given position to build is inside the board, around the selected worker and not occupied by calling checkPos() and validSurrounding() method and notOccupied(). It will also check if the given position to build is buildable or not,which means whether the position has a dome. When performing the build, it will check the god power by calling playerGod.checkBuild(). If the current player has no god power on building, it will just add one to the level of the input (x, y) position and then add one to the player's action and take turns. If the current player has Demeter god, it will check the build times and build position. If it's the first build, the position will be recorded and it will add one to the level of the input (x, y) position. If it's the second build with the same position, the second build will be passed and the player's action will be added. If it's the second with a different position, it will add one to the level of the second position and the player's action will be added.

HW-5 Q1

As shown on the updated object model above, I first extend the Game class to create a subclass called GodGame, which is used to associate with God cards. In the subclass, I override move and build method to add God power to the basic functionality. Instead of using hard coded if statement, I created

a God interface as the default of checking god power, which has various checking methods just for basic game logic. By implementing the interface, I created several God classes like Demeter, Pan, Minotaur and NoGod as well for not selecting a god. In these God classes, I just need to override the basic checking method in terms of the specific God power. Since the God classes are all God, I can simply get the God as playerGod and call playerGod.check…() in each stage that needed, to either go with basic logic if selected God power is not applicable or go with specific God power if the God has the power in this stage. The God power added has nothing to do with the basic Game and Player class. It's just associating with the GodGame class. This kind of design decouples the God cards from the core game, which has low coupling and high cohesion and is much easier to maintain and extend. Having high extensibility, every time we want to add God cards, we just simply create another God class implementing God interface and adding or override the default method for the specific God power. In the GodGame class, we just simply add new method of checking God power logic. An alternative is to hard code if statements in the GodGame class. Although this way will also decouple the base game and the God power, it has very low extensibility and cohesion, which makes it hard to maintain and extend.

HW-5 Q2

In my application, I used strategy pattern for implementing God cards. The basic strategy is the God interface and the extended strategies are different God cards that implement the basic strategy. The God varies independently from the player that uses it. Using this pattern helps me solve the extensibility issue of including God cards and also makes the application more cohesive