* Questions 1 and 2 have remained the same (not required to be updated).
* Question 3 has been changed assuming that the player has a Demeter card and the opponent has no card.
* Additionally, the extra 2 questions have been included at the bottom (numbered 4 and 5, respectively).
* I placed the models in one place so that it is easier to read (Last time I placed it on every question and that tends to clutter things up and it's redundant)

1. How can a player interact with the game? What are the possible actions? Please include necessary parts of the Object Model to explain.

In my implementation, the way that the player interacts with the game is through the Game class. My Game class does not have much, in terms of functions, but it serves as a medium to "connect" everything; it acts as a controller for the rest of my classes (cell, worker, player). I decided that the path of a Game controller, which the user interacts with to access all of the core functionality of playing the game is the most ideal in my opinion since allows me to keep the rest of my classes fairly independent from each other and group functions that are closely connected which follows the principles of high cohesion and low coupling. In Game, you can initiate each player using initiatePlayer which keeps track of their two workers. Once these initial players are created, the game can begin asking each player to take turns moving their worker using initiateMove, and placing towers using initiateTower. The game ends when one of the players' workers has a height of 3 (when they stand on a 3 level high tower). My initiate functions check the validity of a move, so given a row and a col, it determines whether the resulting location is valid or not according to the rules (so I throw exceptions for things like out of bounds, trying to place a worker or tower on top of an existing worker location, etc.)

2. What state does the game need to store? And where should is be stored? Please include necessary parts of the Object Model to explain.

My Game stores a lot of information, mainly because it is the central point of the whole program. While it doesn't have implementations of core functionality of the cell, worker, or player class, it delegates the work to these classes. I keep track of a lot of private variables in my Game class such as currentPlayer, player1, player2, currentWorker, gameOver, winner, and the board. I use currentPlayer to keep track of whose turn it is and currentWorker is used to keep track of which worker was moved (for tower placement since we can only place towers adjacent to a recently moved worker). I also keep track of the board since this allows me to access specific values on the board. Lastly, I have my winner and gameover which helps me determine a winner and end the game. These are all located in my Game class, however I also keep track of some things in my other classes. In my cell class I keep track of the row, col, numLevels,

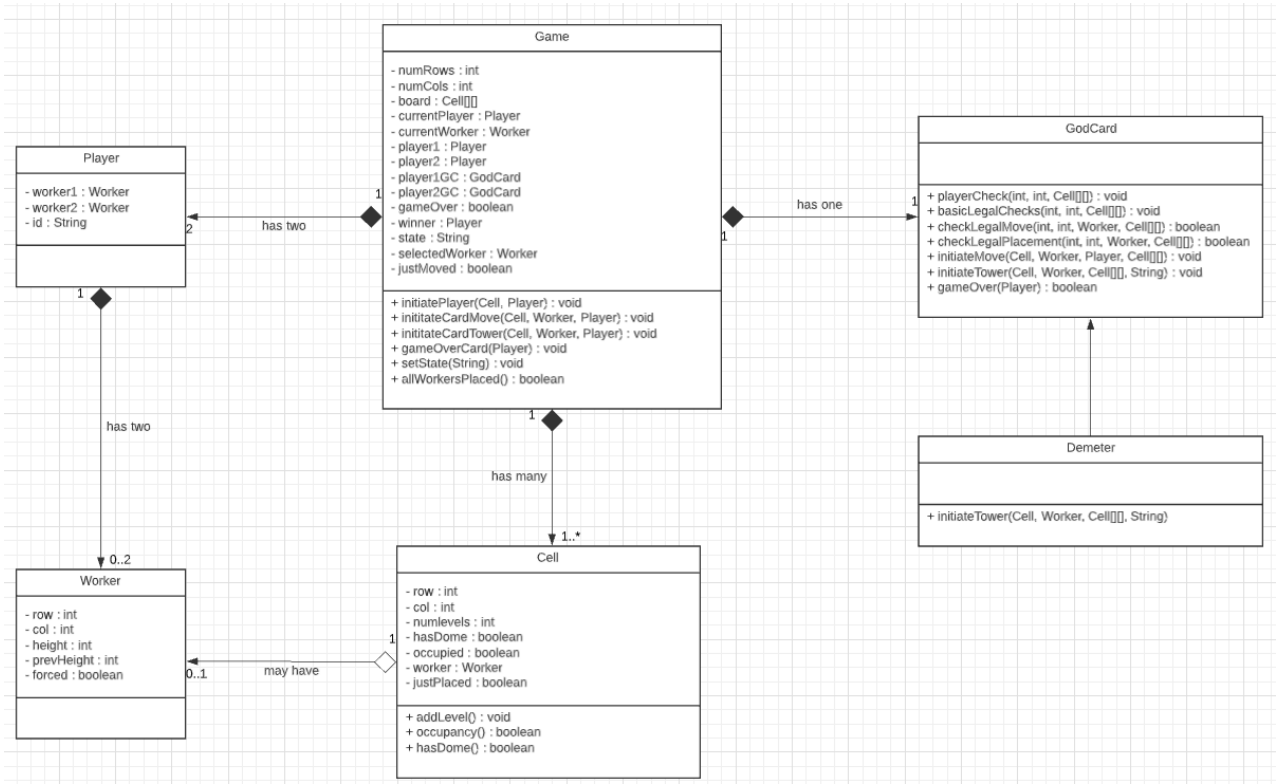
hasDome, and occupancy which are really helpful in keeping everything organized as well as keeping track of important information of a cell (especially for checking validity later on). In my worker, I keep track of row, col, and height so that I can call it later to check if the worker is in a winning position (height). And last, I store the workers in the player class so that I can keep track of which worker belongs to which player.

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? Please include necessary parts of an object-level Interaction Diagram and the Object Model to explain. **Assuming that the active player has the Demeter card (and the opposing player has no card).**

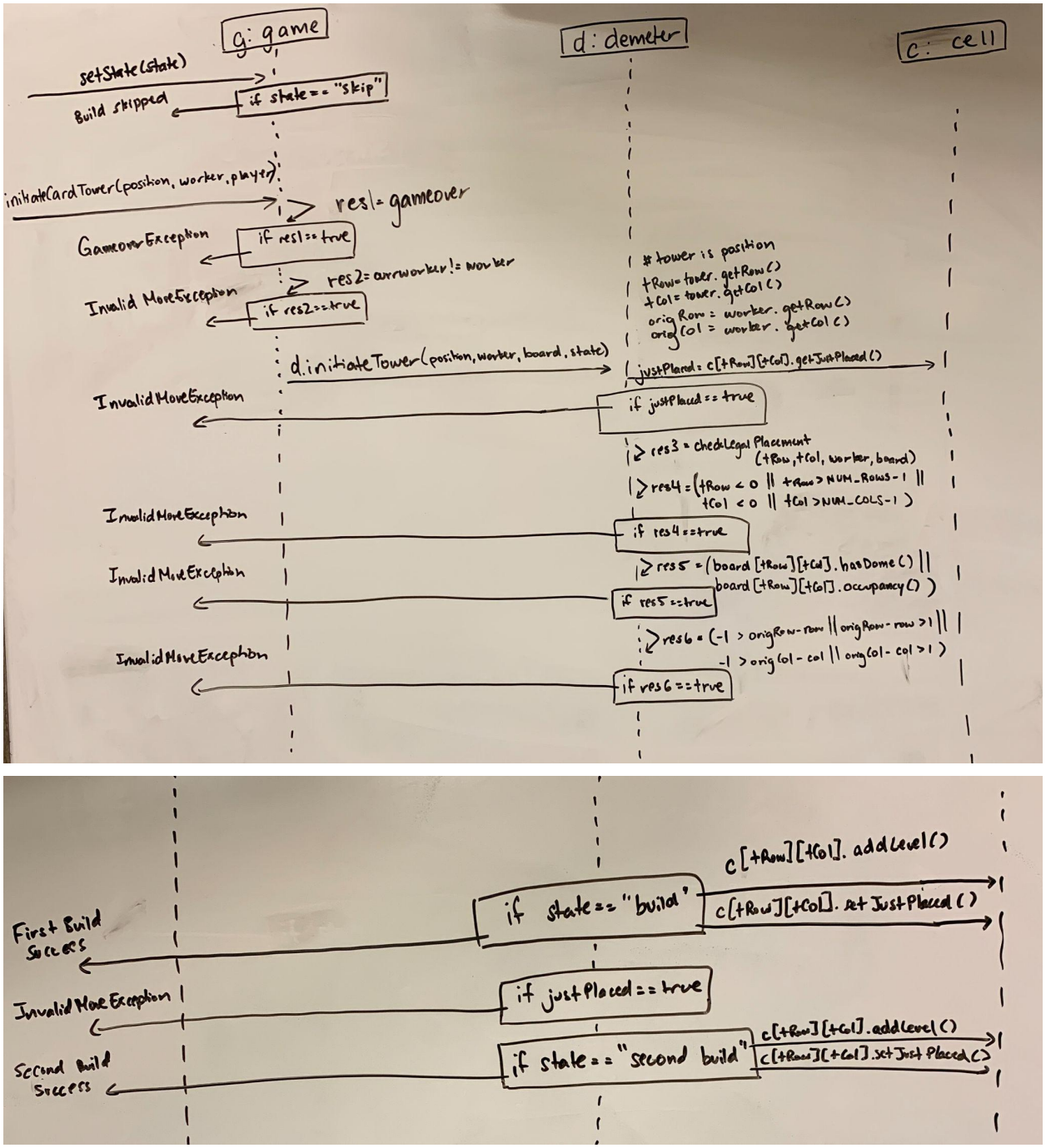
The game determines what is a valid build based on functions that check legality of a move. So checkLegalMove checks if the provided row and col are adjacent to the worker if the row and col are either the same height or their difference is less than 2. It also checks if the row and cols are in bounds and if there are workers/domes on the cell (which is checked by worker.occupancy and worker.hasDome - any height > 3).

The game performs the build when I call initiateCardTower which does a series of initial checks (such as gameover or if the build is adjacent to the worker that was just moved, since these stay consistent no matter which God card/no card the player has). Then we call the initiateTower function from the interface and use the Demeter god's overridden initiateTower function. The inititateTower goes through a series of legality checks and then when all legality checks are satisfied. If it is the first build, we place a tower like we normally would, anywhere adjacent to the worker that was just moved. The user would then be prompted to click "build again" or "skip" from a UI pop up which updates the state accordingly. If they select "build again" then the state is updated to "second build" and we can place the second tower (following the usual legality checks) but with the additional condition that we cannot place the tower on the same place as the first build. If they select "skip" the state is updated and the turn is changed to the other player.

Object Model



Interaction Diagram



4. How does your design help solve the extensibility issue of including god cards? Please write a paragraph (including considered alternatives and using the course's

design vocabulary) and embed an updated object model (only Demeter is sufficient) with the relevant objects to illustrate.

By following the strategy pattern, I am able to have the game call one function, and the rest of that default function's implementation is overridden by the god card. So in this case, instead of the base game (no god card), we have Demeter and when we call initiateCardTower, we automatically use Demeter's intiateTower instead of the base game. This is useful since it significantly increases how easy it is to extend the game with more god cards. All I would have to do is override the default move/build/win methods and then most of the work would be done. The game doesn't need to run a specific function depending on which god card the player has either which allows us to avoid using a series of if statements to check.

Alternatively, I could have used the template pattern since the idea of using default is similar to how abstract classes have a default implementation that we can also override. However, I decided against this since it did not make sense to me for godcards to know the state of the game. While god cards should know a lot about the game, I believe that the game's state (in terms of all the variables like the current player, or the worker that was just selected) should stay in the game class, not an abstract class like god card. Also I think the tradeoff of code duplication for an, arguably, easier to implement, and just as extensible design pattern is worth it.

5. What design pattern(s) did you use in your application and why did you use them? If you didn't use any design pattern, why not?

I used the strategy pattern in my implementation of god cards. The reason was because the strategy pattern allows me to avoid using many if conditionals that checks if a player had a certain type of godcard, since this would destroy the implementation, in terms of extensibility. By having concrete strategies, I can easily add more god cards to my game, without breaking the core functionality of the game (and the game can be played completely independent of the cards as well). My context was the game, it stored the game state and all of the necessary variables to keep the game running smoothly. My strategy, or interface, was the god card interface which had default functions that I could override with my concrete strategies. My concrete strategies were my god cards (Pan, Demeter, Minotaur). Although strategy pattern is not ideal super ideal for a small number of concrete strategies since it may complicate implementation, it is a very efficient and effective design pattern when there are a lot of strategies and since it is easy to implement new strategies once the pattern is set up, I believed that it was the ideal design pattern to follow for extensibility and ease of use.