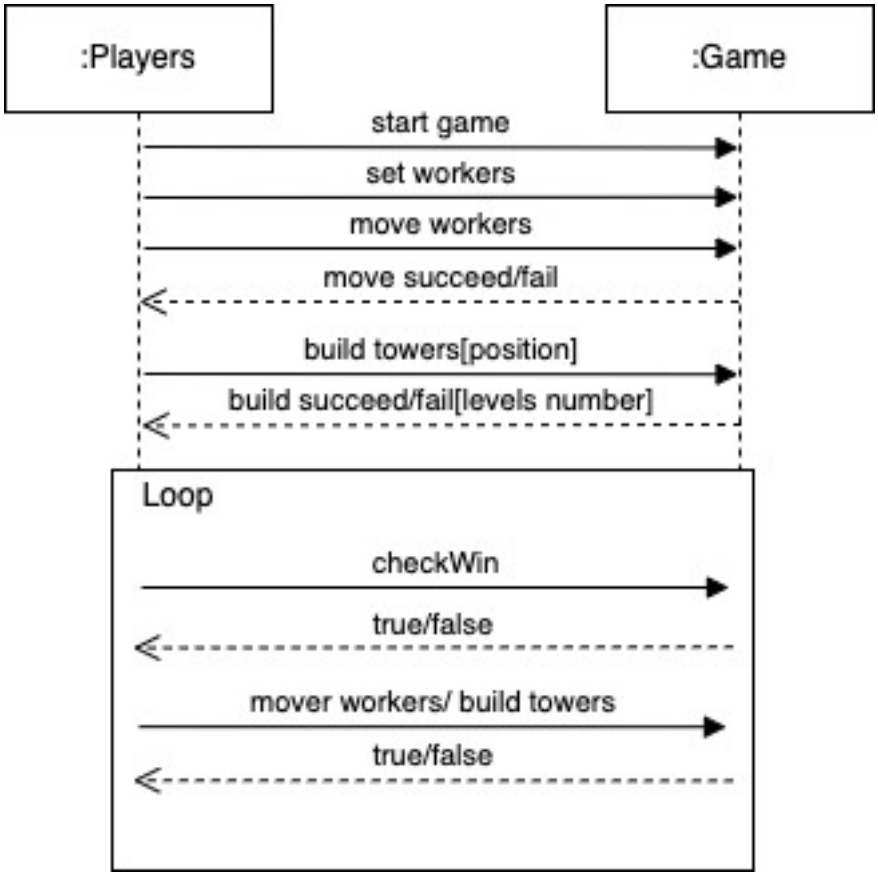
Deliverable 4: Justification. Write a short report that answers the following questions to justify your design choices. To receive full credit, each of your answers to these questions must both:

- Refer to design goals, principles, and patterns where appropriate.

- Discuss the alternatives you had considered and the trade-offs they entailed that led you to choose this particular design (essentially, your design process).

Questions:

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



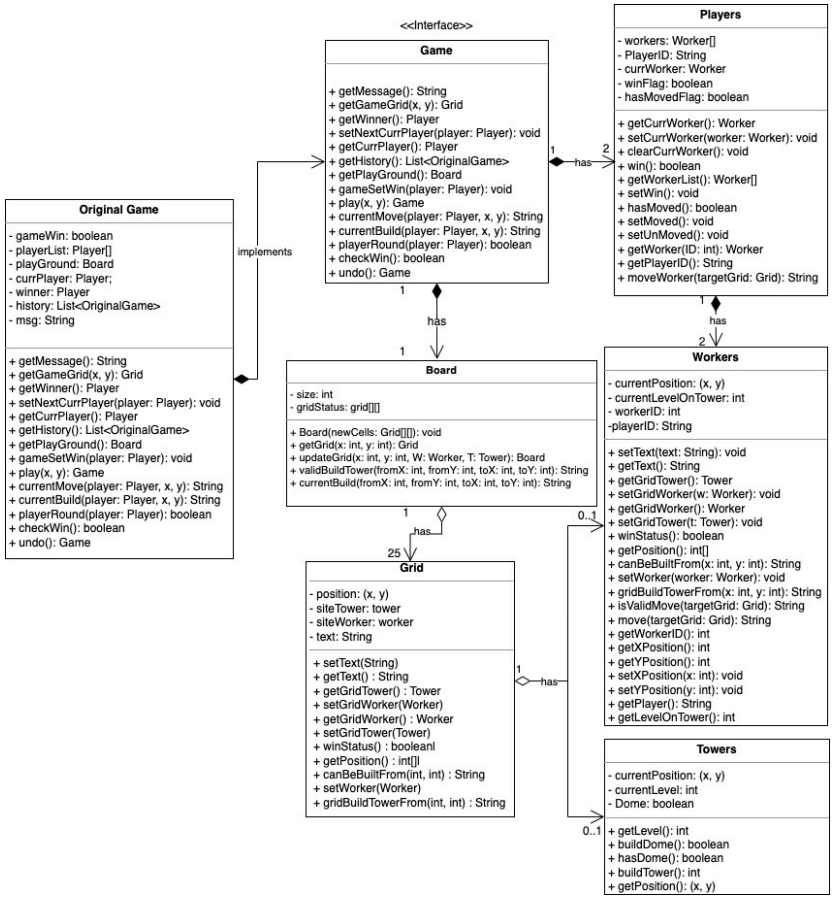
The users input attributes to currentAction functions. The action attribute can be ether build a tower, add a dome or move a worker. Different actions will trigger different boolean judgement inside this function. I will implement certain boolean inside the validAction rather than separate them into different function, since they are short. Then different parts of the game are called to modify the conditions of workers and towers.

I considered about the control to workers from both players and the grid. They Both can get workers from certain functions, which is a kind of coupling I wanted to avoid.

But in one grid, the control to worker is more like save the worker's ID. So I saved this design for now.

I also thought about the trade-off between cohesion on the Board class and coupling on different functions. There was one way that the Player class only store workers' ID and players' ID, but soon I found out that the class of Player is barely used, and Board would have too many cohesion with building functions. Thus, I decided to put moving workers function into the Player, since the building functions are more changing the grid status, or board's character, which is more intuitive.

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.

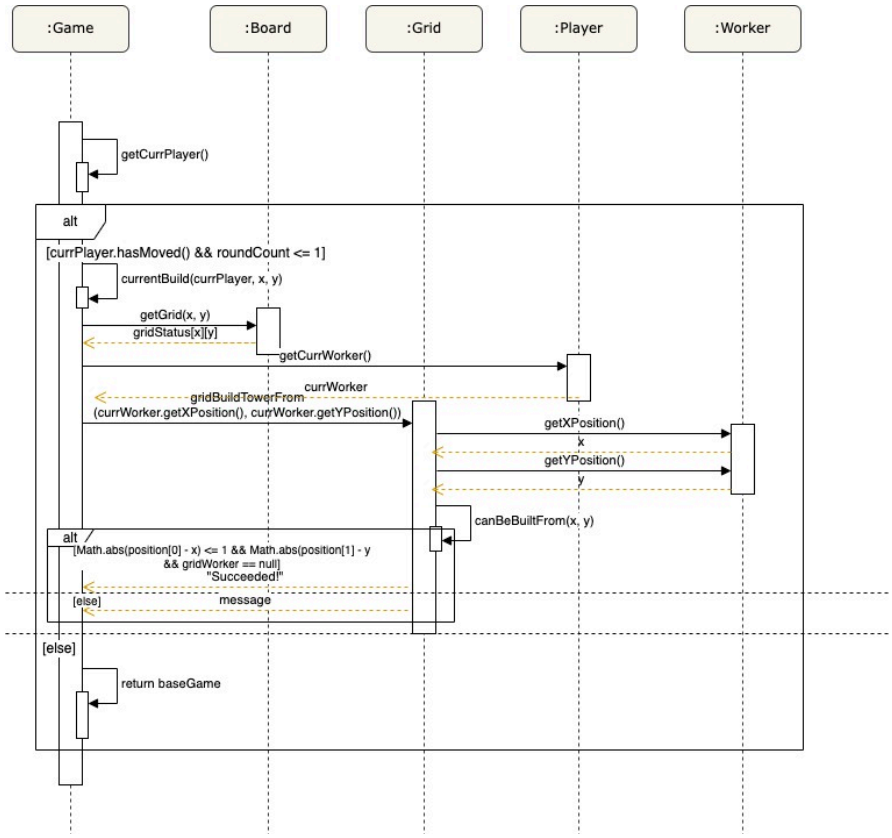


The game stored whether the game is win, the players' list, the playground(board), the winner of the game, a list of history game for undo and also the message return to frontend for informing players. They all store in the private attributes.

The god cards game needs to store a base game that wrapped in different decorators. The base game can shared the game conditions.

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

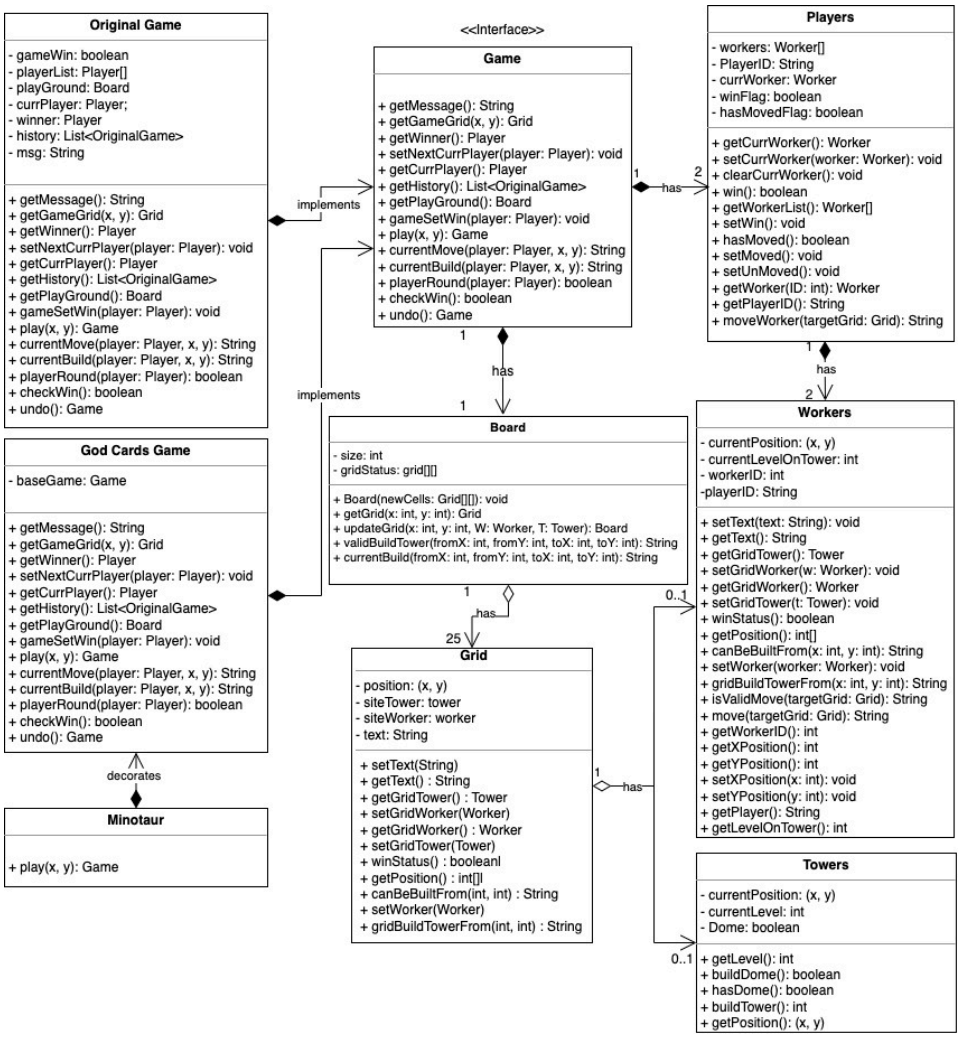
an object-level interaction diagram (using planned method names and calls) to support your answer.



With god cards, the game will first test whether it is the special condition that uses god powers, and then it will just follow what's inside the base game. If the current player has moved, and do not build for more than 2 times, the game will go into the sequences of normal game to check whether it is a valid build. We will check the grid distance and whether there is a worker. About whether there is a tower with 3 levels or with dome, I simply put them in the building function. If the grid tower cannot be built, it is simply will not change.

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 Minotaur is sufficient) with the relevant objects to illustrate.

I used the **decorator pattern** and store a map (cardsMap = new HashMap<Player, Game>(2);) in App.java, where to receive url from frontend. For the initial game, they are all OriginalGame class. After set, for example, one may become Minotaur(baseGame) so that whenever the url is play, it will call Map.get(currPlayer) and use Minotaur(baseGame).play(x, y). If the player does not choose god card, it



will be baseGame.play(x, y). In conclusion, I use baseGame as a wrapee and the GodCardsGame as the decorator. Different god cards extend GodCardsGame.

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 decorator pattern as said before, and I chose it mostly because it follows the logic that different god cards are like decorators of the original game. Template pattern and Strategy pattern, which are also allowed to implement or extend the same interface/class, might be used here but they cannot share the game status only with the original game. They might need to input all states stored in one game to another game. Thus, I think decorator pattern is better.