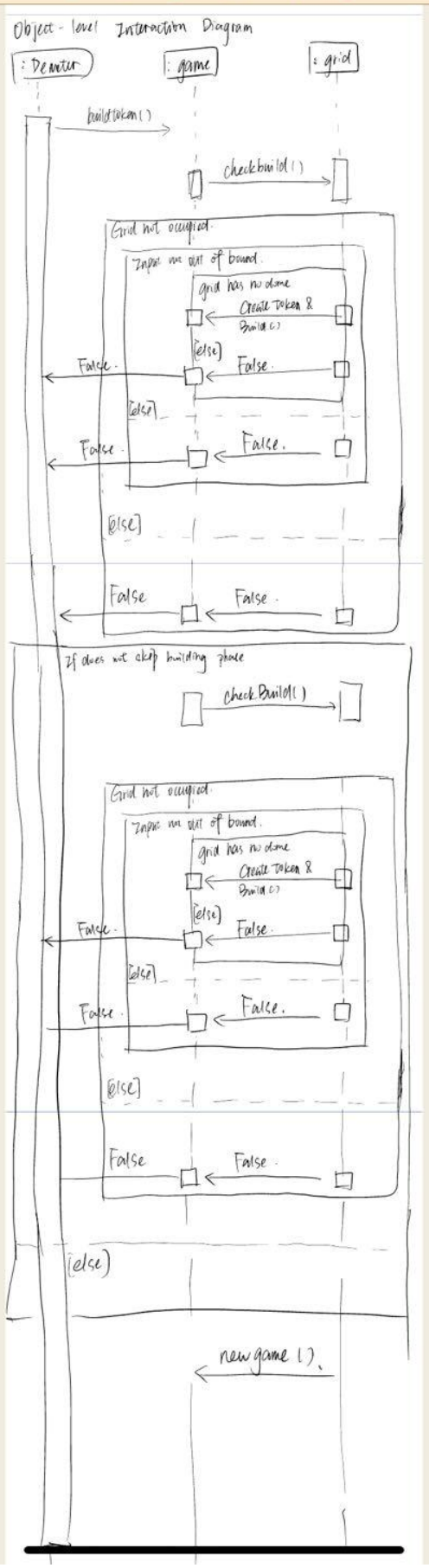
Hw3 Questions: (Some updates on Page 3)

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

1. A player could move their workers every time when it is their round, build a block or put a dome. So, their possible actions are basically moving workers to an adjacent grid, building 1 block height, and putting a dome on the 3-block-height building.

   A player interacts with the game by calling the functions in the Island board class. Several possible actions include:
   1. ***initGame(String nameA, String nameB)***: which could initialize the game and set the name of original 2 players.
   2. *chooseGod(String godA, String godB):* which is set for players to choose the god, and currently since we haven't implemented God, both strings are automatically set as Human.
   3. *pickStartPosition(int[] position):* which takes a int[] as the initial position for worker. This method will automatically repeat running until all four players' positions are settled.
   4. *chooseWorker(int[] position):* which takes an int[] as the position for the worker that the player wanted to select for future move.
   5. *RoundMove(int[] position):* which takes an int[] as the position for the worker to move to.
   6. *RoundBuild(int[] position):* which takes an int[] as the position for the worker to build at.
   7. *TakeTurn():* which swith the turn of players

2. (The object model is attached below.)
   Several states we need to store:
   1. **Player:** The score of each player needs to be stored to determine if anyone wins the game. This could be stored as a variable in the player class.
   2. **Token:** All the tokens including block, dome, tower, and worker, and their position coordinates in the 5*5 grid. This could be stored as x and y coordinates variables in the token class.
   3. **Cell:** The state of each worker and tower in the cell need to be stored. (I,E, whether some cells are occupied or not, the height of the builds, whether it is domed or not.)
   4. **Board:** The state of each grid in the island board should be stored since we need to know the original position of each worker and the status of each grid. This could be stored as a list in the state of grid in the game.



3. (The graph is attached below)

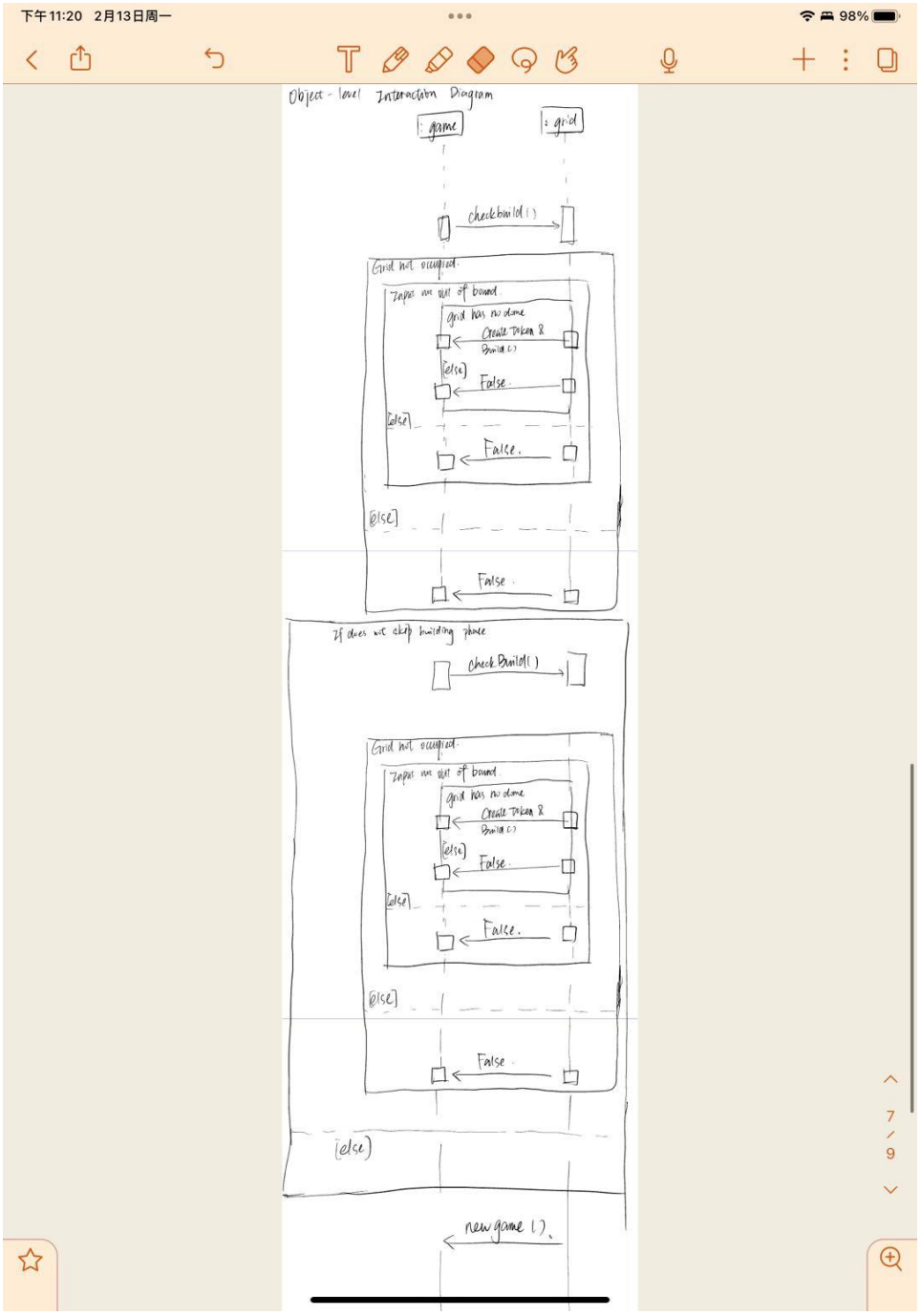
   In a build phase, we need to determine the following steps to determine if it is a valid build:

   a. It is not occupied (the grid is not occupied by another worker).

   b. The coordinate of the chosen grid is in the 5*5 bound.

   c. The coordinate of the chosen grid does not have a dome.

   If all of them are fulfilled, it is considered as a valid build.



Hw5 updates:

Models

Following the game rules, the models corresponding to the real-world objects are Board, Cell, Player and Worker. Their associations are listed as follows:

- 1 Board-> (has 5*5) Cell
- 1 Player -> (has 2) Worker
- 1 Worker -> (occupies 1) Cell

Almost all models above follow the "has-a" modeling technique. With the composition design principle, it is easy to add/change multiple behaviors with dependency injection (/using getters and setters) and also reduce different model coupling.

Design Pattern

Each model has its own inner state to keep track of and also has to interact with others. Thus, a Module Pattern should be used to design the base game models. Almost all attributes mentioned above should be labeled as private and each module has to expose some public methods for the interactions.
To implement God cards, Template Method Pattern would be a good choice. God cards, from the real-world perspective, only change some behaviors of the worker's move and build actions (or change the game-winning rules) based on the game's core functionalities. With much duplicate logic, we can extra the core logic into an <<abstract class>> God, and revise each god's specific power in their subclasses.

- Why use Template Method Pattern:
    - Concrete gods only override certain parts of the non-god (Muggle) game logic which would be less affected by changes happening to other parts of the logic
    - Design for reuse -- pull the massive duplicate code into a superclass
    - Design for understandability -- turn the monolithic non-god behaviors into a series of individual actions in which only some particular steps are extended/modified
    - Design for extensibility -- introduce new gods without having to change many contexts
- Downsides:
    - Some too-specific methods in the subclasses need definition in the superclass, which could make it looks heavy

Extensibility

Gods

In this game, all the gods do not alter too many things but just some certain behaviors with respect to either move or build action. The non-god logic (Muggle.java) is essentially followed/inherited by each god, and minor differences are implemented. For the future implementation of more gods, they could simply extends the abstract God class, inherit base methods and add some extra steps/modify the original methods if necessary.

However, the downside of suppressing a default method vis subclass is not much obvious as each god follows the move and builds core functionalities (e.g. setCurPostion(), addLevel(), computeMovableCells() and computeBuildableCells( )...). So, using Template Method Pattern could be a reasonable choice for implementing god cards. Also, Strategy Pattern could be a candidate. However, with an interface instead of a class, much duplicate code has to be written in each subtype (concrete god class), which is not good for readability and extensibility.

Changes I made for god:
Basically just extend all cards on the god class, add phases for additional moves and builds, and add additional if-else statements for them, E.g.: for the minotaur, I basically do all the functionality in minotaur's own class, by computing the position he will push opponent to, check if it's legal and then set the move for both workers.

GUI

The implementation of GUI and base game is separated by introducing a controller (Game.java) for low coupling and high cohesion between View and Model. In this way, the whole program is much easier to understand, which further leads to good evolvability -- both UI and core logic are easier to change. This also achieves the

design goal of low representational gap. Although the code and dependencies look heavy in Game, it actually does not do much work itself, instead it delegates the work to other objects. In this game, Game is a relative small interface that served as a mediator, which is quite stable. With this decoupling, UI can be easily changed without knowing anything about the domain logic design. In the same way, changes made to the domain logic only affects the controller instead of UI.

Also, the application of controller supports reuse because it serves as an interface to the core logic and supports extensibility.