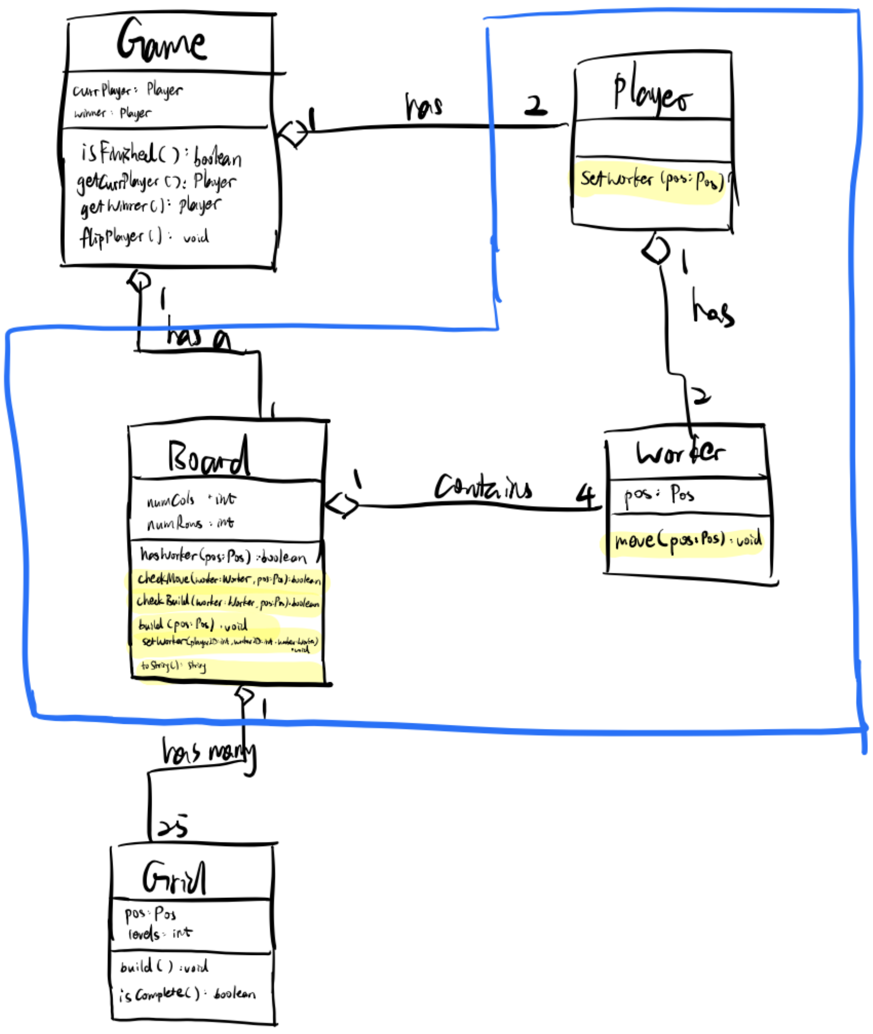
Task 3: Justification

1. Player’s interaction with the game and possible actions

The player can interact with the game in the following ways:
(1) Set the initial position of his/her workers at the beginning of the game
(2) Move his/her workers
(3) Build using his/her workers
(4) check the current board situation



The above actions are implemented and explained in Object Model as below:
(1) When setting the initial position for the worker:
- the Player class call method setWorker(Pos pos)
to initialize and set the position of the worker

- the Board class call setWorker(int playerId, int workerId, Worker worker)
  to add the worker that is initialized by the player to the board

(2) When a player wants to move his/her worker:

- the Board class call checkMove(Worker worker, Pos pos)
  to check if the move to do is a valid move;

- If the move is valid, the corresponding Worker class call move(Pos pos)
  to update the position of the worker

(3) When a player wants to build using one of his/her workers:

- the Board class call checkBuild(Worker worker, Pos pos)
  to check if that build is a valid build
- the Board class call build(Pos pos),
  which further invoke build() at the corresponding Grid class
  to actually perform the build and update the grid tower levels

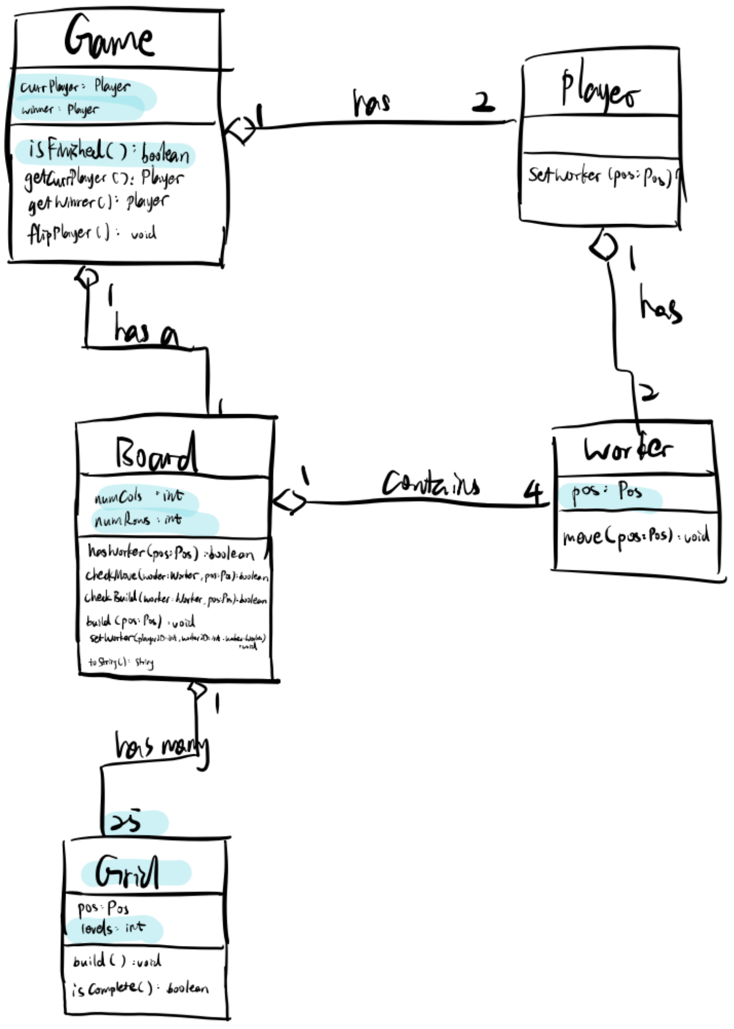
(4) When a player wants to check the current board situation:

The Board class has a override toString() method that prints the current board (with the grid's levels and workers' positions for both players in a human-readable way)

2. What state does the game need to store and where are they stored?

The worker's positions, the tower levels (we can think of the Dome of the tower as level 4 of the tower) of each grid on the board, and who is the next player needs to be stored.

Also, if the game is still going or has ended, and who is the winner if the game has ended also need to be stored.



As shown in the Object Model,

- Worker class store the current position **pos: Pos** of the worker.
- The tower levels of each grid are stored as **levels: int** in the Grid class.
- Each grid on the board is stored as a 2-D Grid array in the Board class.
- The next player is stored as **currPlayer: Player** in the Game class.

- The game status (if is finished) can be accessed by **isFinished(): boolean** function in the Game class.
- The winner of the game (if the game has finished) is stored as **winner: Player** in the Game class. This is set when the **isFinished()** function is called.

3. [UPDATED FOR HW5] How does the game determine a valid build? How does the game perform the build?

If a player has Demeter card, the build can either come from the normal build, or from the additional build from the Demeter god power.

When a player wants to perform a **normal** build, the game has the following operations:
- first, the frontend calls the API (App.class) to perform the build, given the player Id and the worker Id of the build, so the App class calls the Game class to perform the build, passing the same playerId and workerId
- next, the Game do getWorker() to get the Worker object that performs the build,
- Then the Game let the board to check if the build is a valid build, calling checkBuild(Worker worker, Pos pos) to check if the build is valid
- inside the checkBuild() function,
  - the Board calls isInBound(Pos pos) to check if the position is inside the board
  - the Worker calls getPosition() to get the current position of the worker
  - the Board calls isAdjacent() to check if the build position is adjacent to the worker position
  - the Board calls hasWorker() to check if the position is already occupied by another worker
    - inside hasWorker(), all 4 workers stored in the board calls its getPosition() function to check if they are occupying the target position
  - the Board calls getGrid(Pos pos) to get the corresponding grid of the target position
  - the grid calls isComplete() to check if the tower of the grid can be further built (if contains a doom)

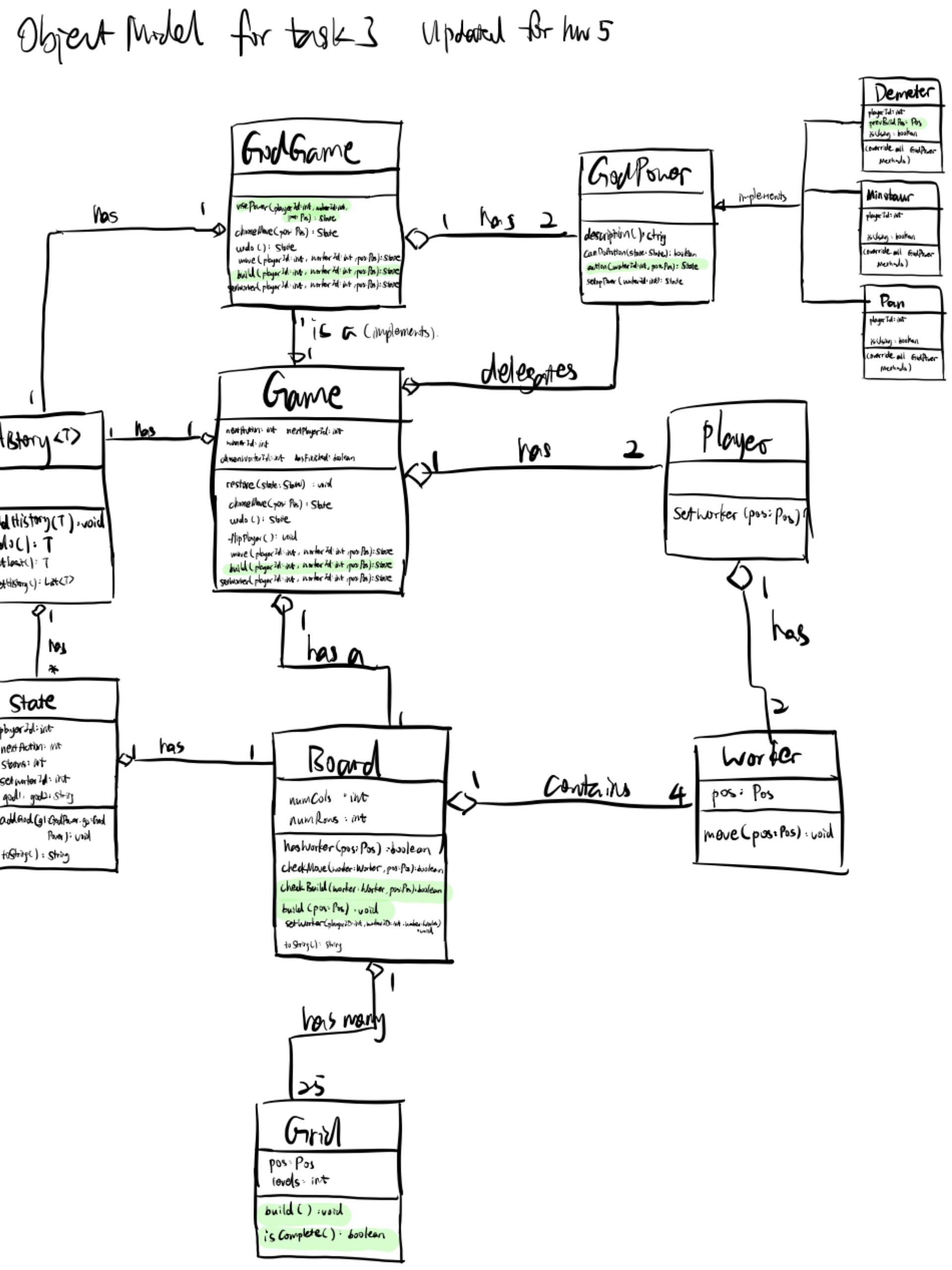
  If all of the above checks succeed, the checkboard() returns true
  Otherwise if any of the above checks fail, the function will return null, indicating the build() is failed
- Then, if checkboard() is true, the Game will call build(). Inside the build() function,
  - the Board calls getGrid(Pos pos) again to get the grid of the target position
  - The grid calls build() to finish the building operation
  - A new Board is returned to the Game
- When receiving the new Board, the game would call the Demeter God to storeInfo(pos), storing the position of this successful build into a variable called “prevBuildPos”, so that if the additional build is used, it will know if the additional build is on the same position (which is an invalid operation)
- Then the Game packs the new Board returned with the game globals into a new State and stores it into the game History
- Finally returns the new State back to the App API

On the other hand, when a player wants to perform a **Demeter GodPower** build (the additional build), the game has the following operations:

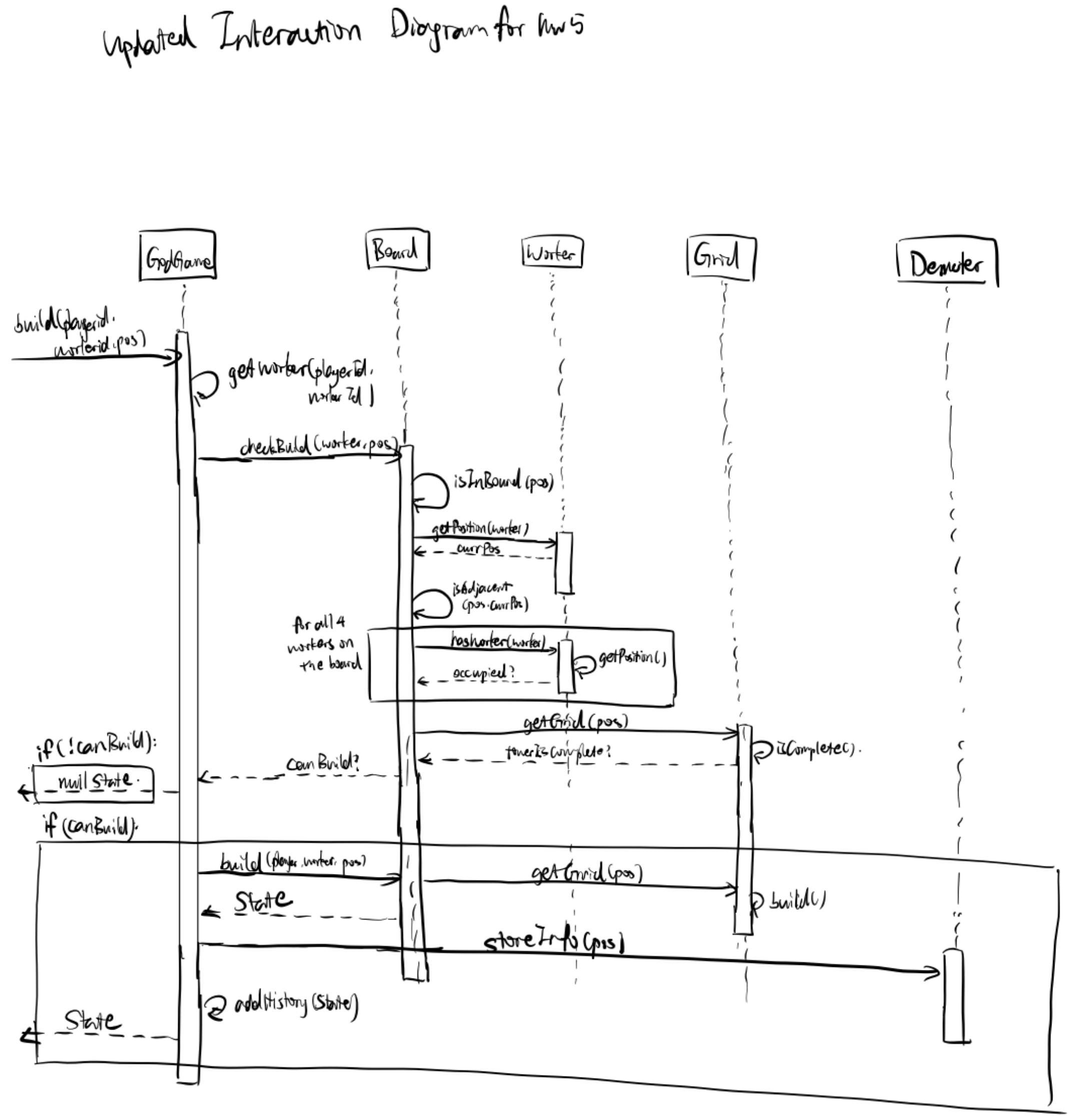
- In the god mode, the game would be a GodGame, and the GodGame would call usePower(playerId, workerId, pos) function to use the god power. Specifically, in this case, it would be the additional build
- Inside the usePower(), the GodGame would let the GodPower Demeter to perform the action. So the demeter would call action(workerId, pos) to perform the additional build;
- Inside the action(workerId, pos), the Demeter object first checks if (pos == prevBuildPos), since the additional build cannot store
- Then, if the operation is valid, since delegation is used here, the Demeter contains a Game reference inside it, it just calls the game.build() and perform the checks and returns the newly built State
- After that the GodPower stores the State into its history
- Finally return the new State back to the App API

All graphs see next page.

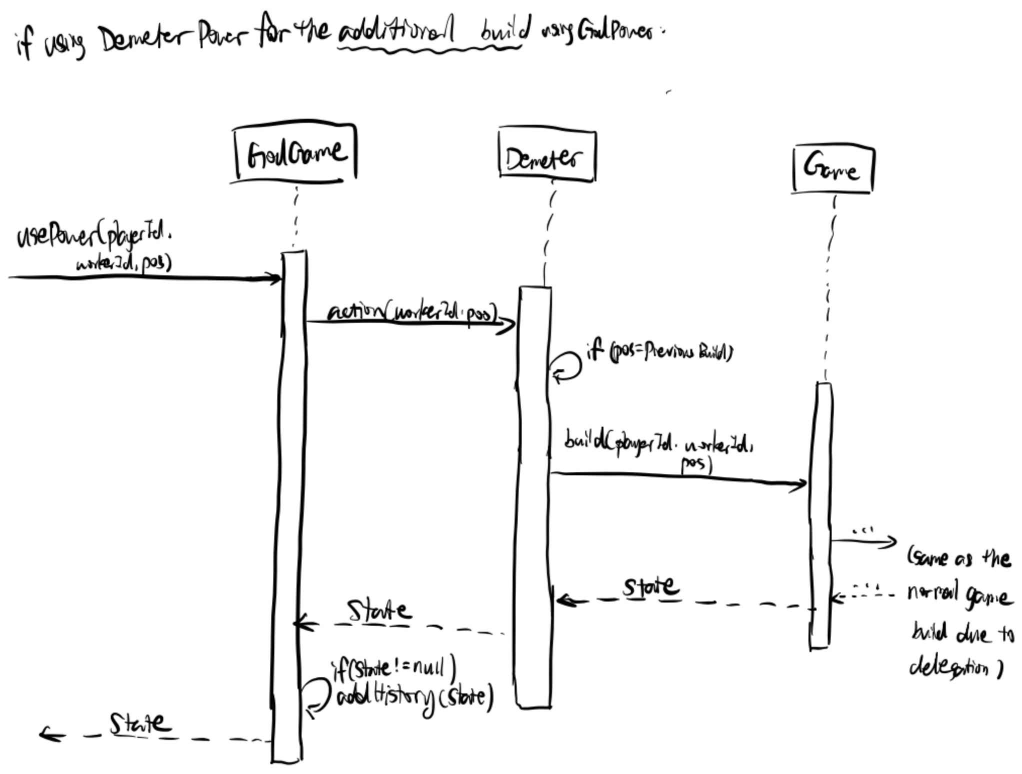
Picture 1: the updated object model (the related operations are highlighted)


Object Model for task 3 Updated for hw 5
GodGame
usePower(playerId: int, workerId: int, pos: Pos): State
chooseMove(pos: Pos): State
undo(): State
move(playerId: int, workerId: int, pos: Pos): State
build(playerId: int, workerId: int, pos: Pos): State
setWorker(playerId: int, workerId: int, pos: Pos): State
GodPower
description(): String
canDoAction(state: State): boolean
action(workerId: int, pos: Pos): State
setupPower(workerId: int): State
implements
Demeter
playerId: int
prevBuildPos: Pos
isUsing: boolean
(override all GodPower methods)
Minotaur
playerId: int
isUsing: boolean
(override all GodPower methods)
Pan
playerId: int
isUsing: boolean
(override all GodPower methods)
has
is a (implements)
delegates
Game
nextAction: int nextPlayerId: int
winnerId: int
chosenWorkerId: int hasFinished: boolean
restore(state: State): void
chooseMove(pos: Pos): State
undo(): State
flipPlayer(): void
move(playerId: int, workerId: int, pos: Pos): State
build(playerId: int, workerId: int, pos: Pos): State
setWorker(playerId: int, workerId: int, pos: Pos): State
History<T>
addHistory(T): void
undo(): T
getLast(): T
getHistory(): List<T>
Player
setWorker(pos: Pos)
has a
has many
contains
State
playerId: int
nextAction: int
Stones: int
setWorkerId: int
god1, god2: String
addGod(g1: GodPower, g2: GodPower): void
toString(): String
Board
numCols: int
numRows: int
hasWorker(pos: Pos): boolean
checkMove(worker: Worker, pos: Pos): boolean
checkBuild(worker: Worker, pos: Pos): boolean
build(pos: Pos): void
setWorker(playerID: int, workerID: int, worker: Worker): void
toString(): String
Worker
pos: Pos
move(pos: Pos): void
Grid
pos: Pos
levels: int
build(): void
isComplete(): boolean
1
2
4
*
25

Picture 2: the interaction diagram for the normal build



Picture 3: the interaction diagram for the Demeter's additional build (The actions in the Game delegation are omitted because they are the same as Picture 2)


if using Demeter Power for the additional build using GodPower.
GodGame
Demeter
Game
usePower(playerId, workerId, pos)
action(workerId, pos)
if (pos=Previous Build)
build(playerId, workerId, pos)
...
(Same as the normal game build due to delegation)
State
State
if(State != null) addHistory(State)
State

Additional questions for hw5:

1. How does your design helps 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.

Object model: see *Picture 1* of the previous pages.

The application has a Game interface, that the frontend App would call to perform the functionalities. The interface has two implementations: GameImpl that for the normal game and GodGame that for the god mode. The GodGame has a Game object inside it to perform delegation and increase code reusability. The GodGame also has 2 GodPower objects that helps to perform god operations. The GodPower is an interface that supports methods like:
- canDoAction() that helps to determine if a god power can be used in the current game situation
- action() that actually performs the god action (the demeter additional build in this case)
- setupPower() that sets the game environment to get ready to use the god power

The GodPower interface can be implemented by all kinds of gods: in the current application, Demeter, Minotaur and Pan all implements the GodPower interface and perform their god powers. In the future if more gods need to be added, one can just write another class that implements the GodPower interface then it would be successfully included in the game.

2.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?

Both decorator pattern and strategy pattern are used in my application.

Decorator pattern is used because this pattern helps to extend functionality of an existing class. In the application, there's a Game interface, that the frontend App would call to perform the functionalities. The interface has two implementations: GameImpl that for the normal game and GodGame that for the god mode. The GodGame has a Game object inside it to perform delegation and increase code reusability. Therefore, in this case, the GodGame serves as a Decorator class for the Game class that performs additional God operations.

Strategy pattern is also used in the application for setting the correct god power. In the god mode, when the players have chosen their god, the game need to be set up with the corresponding god powers. The GodGame includes two GodPower objects that can be implemented with different gods. And when setting the GodGame, the constructor will take two strings as the names for the two gods, and the constructor would new the corresponding GodPower (Demeter, Pan, Minotaur, etc) respectively based on the god name's parameters.