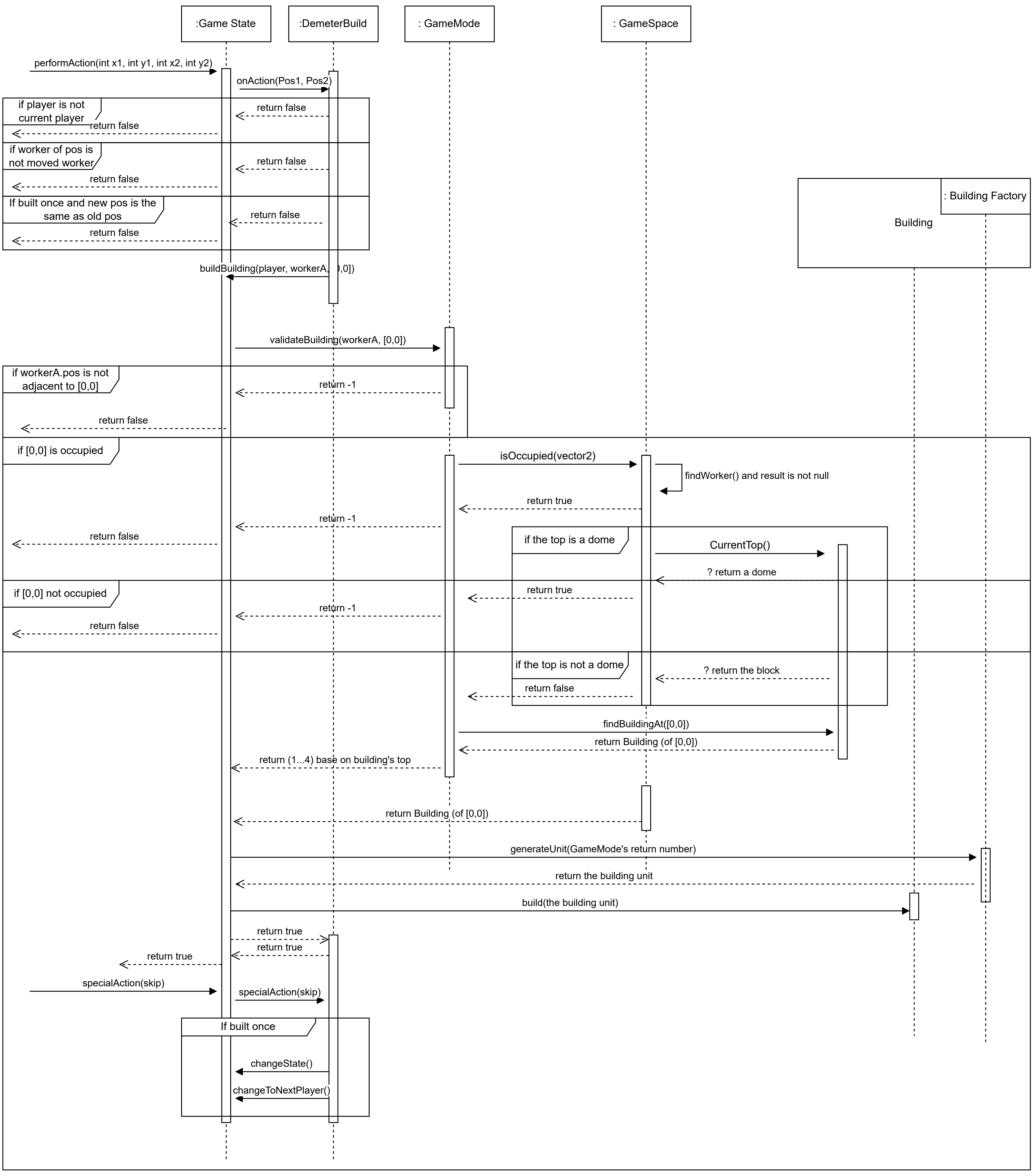

Progress of validation:

0. Game State ask current state to perform action
1. Demeter Build (as current state) Check if the player is current player
2. Demeter Build Check if the worker is the last worker used
3. Demeter Build Check if built before and if new build pos is the same of last build pos
4. Game State asked Game Mode for rule related validation.
5. Game Mode Check if the position is adjacent to worker
6. Game Mode Check if the position is occupied by asking Game Board
7. Game Check if any worker stays on the position or there's a dome at top of building, then return to Game Mode
8. Base on adjacent check and occupy check, Game Mode return the validation result to Game State

Progress of build:

1. build will only happen if all above validation is true
2. Demeter Build called build method in Game State
2. Game State will look for the Building object from the Game Board base on position info
3. Game State will ask Building Factory to create a Building Unit base on validation result of Game Mode
4. Game State will call build function of Building, stacking the new Building Unit onto this Building object
5. Game State return true to tell the Demeter Build the build is successful proceeded.
6. Demeter Build call Game State to change character if condition matched
7. Demeter Build call Game State to change state if condition matched
8. Demeter Build return true to Game State to Mark action is success

Progress of skip:

1. Game State ask current state to special action
2. Demeter Build (as current state) check if the special action info is skip
3. Demeter Build check if build is performed once
4. If condition matched, Demeter Build change character and state by calling Game State's method

Reasoning of this design:

The design involves delegation, inheritance and factory.

Demeter is an implementation of IState. When Game State is asked to perform anything
The current IState will actually execute it. So we could easily have many customizable card by making their own state

Game Mode is an Interface that is swappable. Which means the validation of rules (expect player check) can be varied
In this case, we use the GodCardMode implementation of game mode
It will return the God Card state when Game State ask for next state
When we don't want God Card, we just use Minimal Mode

Inheritance is used between some IState
The Game Mode tells you what the next state shall be base on current State
If all God State need an individual check that will be very redundant
Thus, we let God State inherit normal state like move & build
So we only need check for move and build

Building factory serves similar purposes. The current game design doesn't have any limitation on block number
But if we want to apply that later, we don't want to rewrite the Game Class. Instead, we swap a Building factory
Also, using an independent factory to handle "dome or block", "level of block" separates low level and high level

Trade off:

For the inheritance used we kind of have some redundant codes
And we are actually using instance of for state checking
This seems not very good, but it's dynamic and at least better than string comparison
We can possibly store the state's as variable in Game State and use object comparison
But that means Game State will need specific logic for God Cards which is not what we want

The coupling of current solution is actually not very low.
By storing Board and Factory reference in Game Mode, Game State need to access them through Mode
Plus, Game State need to Get Building from Board as an extra step
If we combined Game State and Game Mode into a single Game Class, coupling will be lower
But I still think the extensibility is more important in this scenario, as the original game is an extensible card game
Also, if we do further, combing Game Board into Game Class, this will make the Game Class a God Class, which is more risky