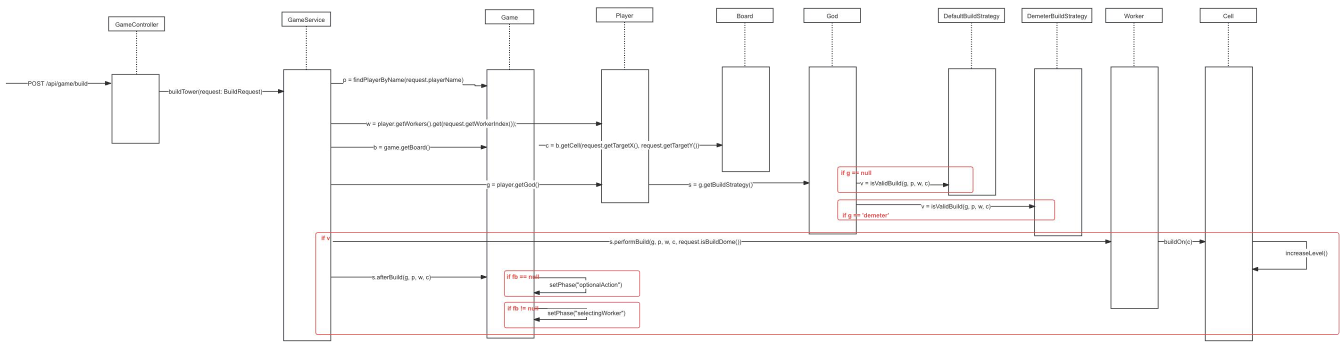
Justification for Building Action

The building action in *Santorini* is a structured process that involves multiple components, each responsible for a specific part of validation and execution:

- **GameService**: Acts as the central coordinator for user-triggered actions. It identifies the current player and worker, retrieves the appropriate `BuildStrategy` from the player's assigned `God`, and delegates both validation and execution to this strategy.
- **BuildStrategy (Interface)**: Defines the contract for validating and performing build actions. God-specific strategies (e.g., `DemeterBuildStrategy`) implement this interface to customize the behavior, including whether a second build is allowed.
- **DemeterBuildStrategy**: Extends `DefaultBuildStrategy`. It tracks the cell built on during the first build to ensure the second build occurs on a **different** cell. It manages optional flow control between builds using local state and overrides methods such as `afterBuild()` and `allowsSecondBuild()`.
- **Game**: Maintains high-level state such as turn order and phase transitions. It delegates build execution and win condition checking to appropriate strategy instances, keeping the game orchestration decoupled from god-specific logic.
- **Player**: Owns workers and a god card. It ensures worker ownership and serves as a secure interface between game logic and worker actions.
- **Worker**: It performs state-mutating actions like `buildOn(targetCell)` after receiving validated commands from the strategy layer.
- **Cell**: Manages state changes related to the build, such as increasing the tower height.

Object-level Interaction Diagram


GameController
GameService
Game
Player
Board
God
DefaultBuildStrategy
DemeterBuildStrategy
Worker
Cell
POST /api/game/build
buildTower(request: BuildRequest)
p = findPlayerByName(request.playerName)
w = player.getWorkers().get(request.getWorkerIndex());
b = game.getBoard()
c = b.getCell(request.getTargetX(), request.getTargetY())
g = player.getGod()
s = g.getBuildStrategy()
if g == null
v = isValidBuild(g, p, w, c)
v = isValidBuild(g, p, w, c)
if g == 'demeter'
if v
s.performBuild(g, p, w, c, request.isBuildDome())
buildOn(c)
increaseLevel()
s.afterBuild(g, p, w, c)
if fb == null
setPhase("optionalAction")
if fb != null
setPhase("selectingWorker")

Implementation Details
1. Validation Process and Building Process Flow

Validation Process (with Demeter)

GameService

Receives a build request from the frontend.

Locates the active `Player` and `Worker` using names and indices.

Delegates validation to the player's `God`'s `BuildStrategy` via `isValidBuild(game, player, worker, cell)`.

DemeterBuildStrategy

If this is the **first build** of the turn:

- Validates spatial rules using inherited `DefaultBuildStrategy` logic: `Board.isAdjacent()` & Target cell is not occupied or domed
- Saves `firstBuild` cell.

If this is a **second optional build**:

- Validates **different cell** from `firstBuild`.
- Applies same spatial and occupancy rules.

Result

If `isValidBuild(...)` passes, `GameService` proceeds to `performBuild(...)`.

If not, an exception is raised and the build is blocked.

Building Process Flow

GameService

Delegates the actual build via:

```
strategy.performBuild(game, player, worker, targetCell,
request.isBuildDome());

strategy.afterBuild(game, player, worker, targetCell);
```

DemeterBuildStrategy

If this was the **first build**:

- Saves `firstBuild = targetCell`
- Sets `game.setPhase("optionalAction")` to allow second build

If this was the **second build**:

- Calls `resetTurnState()` to clear `firstBuild`
- Calls `game.nextTurn()` and `game.setPhase("selectingWorker")`

Cell

`Cell.increaseLevel()` is called to modify board state

Occupancy is updated if applicable

2. Design Decisions and Justification

Component	Responsibility	Justification
GameService	Delegates to appropriate strategies based on the player’s god	Follows the Strategy Pattern; supports open/closed principle for god card extension

DemeterBuildStrategy	Implements god-specific build behavior with internal state firstBuild	Applies the Template Method Pattern: extends base behavior with override, local state allows Demeter-specific rule
Worker	Now just a build performer, no longer validates build	Reduced responsibility simplifies code, aligns with "Information Expert" and "Low Coupling" principles
Game	No longer checks detailed build logic, delegates to strategy	Promotes separation of concerns, allowing Game to focus on orchestration not validation
Cell	Purely handles structural state change	Maintains SRP, allows modular testing and avoids logic entanglement with gameplay

3. Alternative Approaches Considered

Alternative1: Game-Level Building Logic

Considered having the `Game` class handle building logic:

Pros:

- Centralized game rules and logic in one place
- Easier to modify core game mechanics

Cons:

- Violates *Single Responsibility Principle*
- ncreases coupling, as `Game` would need deep access to `Worker`, `Cell`, and `Board` internals

Alternative2: Unified God Interface without Strategy Separation

Considered simplifying the architecture by having a single `God` interface encapsulate all god power behaviors (build, move, win conditions), rather than separating them into `BuildStrategy`, `MoveStrategy`, and `WinConditionStrategy`.

Pros:

- Reduces the number of classes and interfaces

- Easier to understand and implement simple gods with limited variation

Cons:

- Violates the Interface Segregation Principle
- Makes it harder to mix and match behaviors (e.g., one god reuses default move logic but overrides build logic)
- Less extensible: adding new god cards may require rewriting large parts of the monolithic `God` class