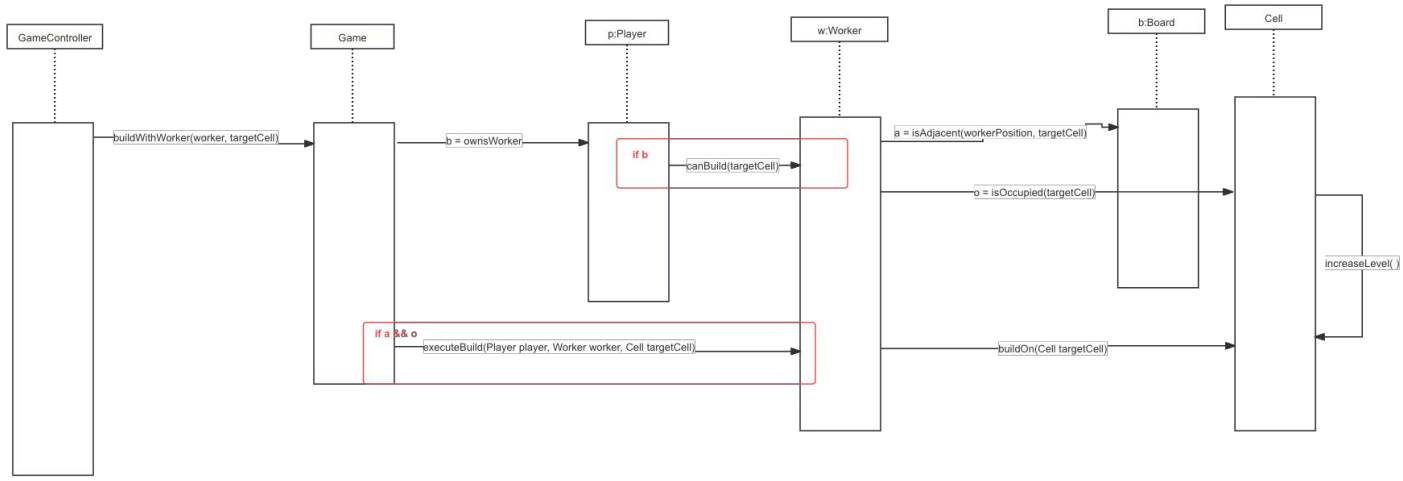
Justification for Building Action

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

- `GameController`: Handles user interaction and delegates the build request to the appropriate game entities.
- `Game`: Ensures that the current player is making a valid move and delegates build execution to the relevant worker.
- `Board`: Performs spatial validation, ensuring the build location is **within bounds**, **adjacent**, and **not occupied** by another worker or a dome.
- `Player` : Manages worker ownership and delegates build operations to the worker.
- `Worker`: Handles the core logic of the building process, ensuring the build action follows game rules.
- `Cell`: Manages state changes related to the build, such as increasing the tower height.

Each component follows the **single responsibility principle** ,ensuring that validation and execution steps are properly distributed without unnecessary coupling.

Object-level Interaction Diagram



Implementation Details

Validation Process

GameController:

1. Receives a build request from the user.
2. Calls `Game.buildWithWorker(worker, targetCell)`.

Game:

1. Ensures that the `worker` belongs to the **current player**.
2. Ensures that the **target cell is valid** by delegating validation to the `Board`.

Board

1. Checks whether `targetCell` **exists and is within board boundaries**.
2. Ensures `targetCell` is **adjacent** to the worker's current position:

   ```
   Board.isAdjacent(worker.getPosition(), targetCell)
   ```

3. Checks whether `targetCell` is **not occupied** (i.e., it does not contain a **worker** or **dome**).

Worker:

1. Ensures that `targetCell` is **not the worker's current position**.
2. Confirms that the worker is eligible to build by calling `Worker.canBuild(targetCell)`.

Final Decision:

1. If all validations pass, the game proceeds to the building execution phase.
2. Otherwise, the build action is rejected, and an error message is returned.

Building Process Flow

Once validation is complete, the following steps are executed to update the game state:

1. `GameController` initiates the build action.
2. Game calls `Player.buildWithWorker(worker, targetCell)`.
3. `Player` validates ownership and calls `Worker.build(targetCell)`.
4. `Worker` executes the build action:
    - Calls `Cell.increaseLevel()` to raise the height of `targetCell`.
    - Updates the game state to reflect the new tower height.
5. **`Game`** checks for win conditions:
    - If the worker's movement in the next turn leads to a level-3 tower, the player **wins**.
6. Turn advances:
    - The game proceeds to the next phase or the next player's turn.

Design Decisions and Justification

The design follows a **layered responsibility approach**, ensuring that each component is responsible for only what it needs to know:

1. Board - Spatial Validation

- The board knows the **physical constraints** of the game, making it responsible for checking:
  - Whether the **target cell exists**.
  - Whether the **target cell is adjacent**.
  - Whether the **target cell is occupied**.
- **Advantage:** This keeps spatial validation **encapsulated** in one place, making it easy to modify the board without affecting the rest of the system.

2. Worker - Gameplay Validation

- he worker has knowledge of:
  - The **player's rules** regarding movement and building.
  - Whether it can legally perform a build at a given location.
- **Advantage:** This ensures that each **worker instance** can independently check if it is allowed to perform an action without exposing unnecessary details.

3. Player - Worker Ownership and Delegation

- he player **manages its workers**, ensuring:
  - The worker **belongs to them** before allowing a build action.
  - The game follows a **valid turn order**.
- **Advantage:** This enforces player-based restrictions without adding unnecessary complexity to `Game`.

4. Game - High-Level Coordination

- The `Game` class is responsible for:
  - Ensuring the **current player is acting**.
  - Delegating tasks to `Player`, `Worker`, and `Board`.
  - Updating the **game state** and checking for **win conditions**.

Alternative Approaches Considered

Alternative 1: Cell-Level Building

Considered having the `Cell class` handle building validation:

Pros:

- Direct access to level and occupancy information
- Simpler coordinate calculations

Cons:

- Would require Cell to know about Worker positions, which violates the information Expert principle
- Increases coupling between Cell and Worker

Alternative 2: Game-Level Building Logic

Considered having the `Game class` handle building logic:

Pros:

- Centralized game rules and logic to Game class
- Easier to modify game rules

Cons:

- Violates Single Responsibility Principle
- Higher coupling as Game would need detailed knowledge of Workers and Cells

Final Design Choice

The final implementation adopts a distributed responsibility approach where building actions flow through multiple layers, each handling its specific concern:

1. The `GameController` handles user interaction and coordinates the building process:
2. The `Player` class acts as a security layer, ensuring only valid worker operations
3. The `Worker` class contains the core building logic, making it the central point for move validation
4. The `Cell` class handles the actual state changes

This design was chosen over alternatives because:

1. It places building validation in the `Worker` class where all necessary information is readily available - both the worker's position and the target cell's state are accessible without needing to query multiple objects.
2. The separation between validation (`canBuild`) and execution (`build`) makes the code safer and easier to debug - we can check if a build is valid without actually performing it.
3. The `Cell` class focuses purely on state management, making it simple and resistant to bugs - it doesn't need to know about game rules or worker positions.

While I could have centralized all logic in the `Game` class, this design provides a practical balance: it's easy to understand and correspond to the real game logic, maintains good separation of concerns, and is sufficiently flexible for the game's requirements.