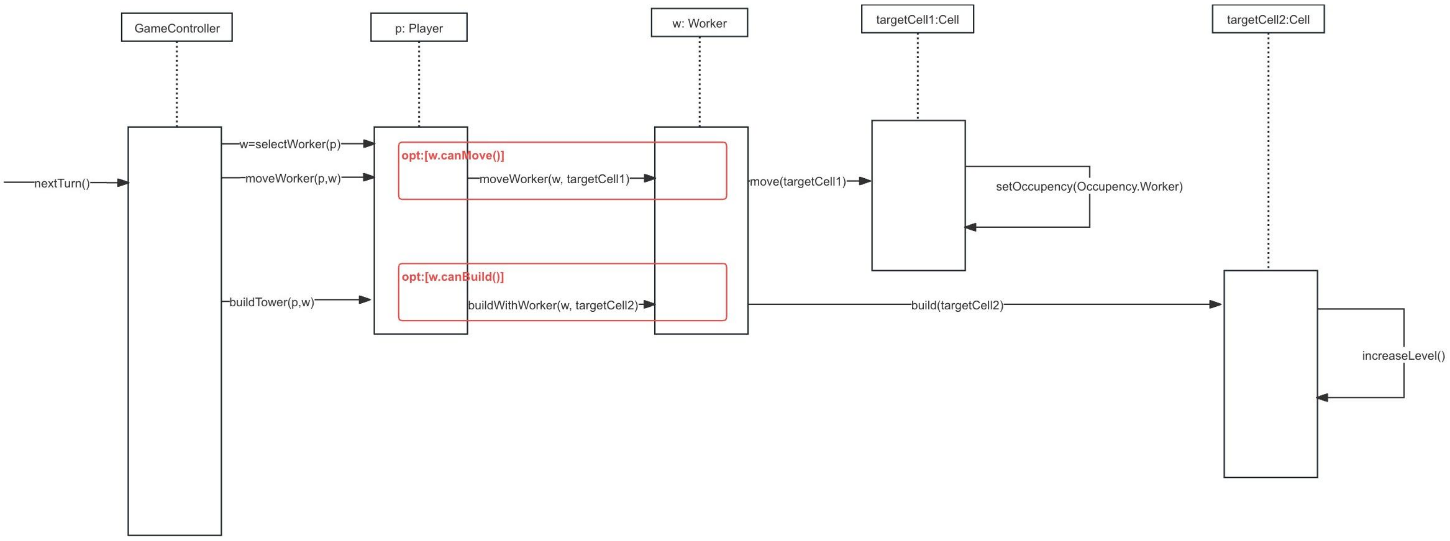
Justification for Building Action

The building action in Santorini involves several key components working together:

1. **GameController:** Handles user input and coordinates the building process
2. **Player:** Acts as a facade for worker operations
3. **Worker:** Contains the core building logic and validation
4. **Cell:** Manages the actual state changes for building

Object-level Interaction Diagram



Implementation Details

Validation Process

The building validation is primarily handled by the `Worker.canBuild( )` method, which checks:

- Target cell must be adjacent (checks `dx` and `dy` ≤ 1)
- Target cell must not be occupied (uses Cell's `isOccupied( )`)
- Target must not be the worker's current position (`dx` != 0 || `dy` != 0)
- Target cell must exist (not null)
- Worker must have a position (not null)

Building Process Flow

- `GameController.buildTower( )` initiates the building process and handles user interaction
- `Player.buildWithWorker( )` verifies worker ownership and delegates to the worker
- `Worker.build( )` validates and triggers the building action
- `Cell.increaseLevel( )` performs the actual state change

Design Decisions and Justification

Responsibility Distribution

The implementation follows several key design principles:

- **Information Expert:** Worker class handles build validation since it has the necessary information about position and movement rules
- **Low Coupling:** Components interact through well-defined interfaces without exposing internal details
- **High Cohesion:** Each class has clear, focused responsibilities:
    - Worker: Building logic and validation
    - Cell: State management
    - Player: Worker ownership, control and delegation
    - GameController: User interaction and coordination

Layered Design

The implementation uses a layered approach that separates concerns:

- UI Layer (`GameController`)
- Domain Layer (`Player`, `Worker`)
- Data Layer (`Cell`)

This separation allows for better maintainability and potential future modifications.

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.