Build-Justification for Demeter Card

This document provides a clear and structured justification for how build actions are validated and performed when the active player has the **Demeter card**. It systematically addresses checks, game state updates, responsibilities, design principles, and the interaction between the base game and the Demeter-specific rules.

---

1. Validation of Build Actions and State Updates

a. Validation Checks

To determine whether a build action is valid, the following checks are performed:

1. **General Build Validation**:
   - The target cell for the build must:
     - Be adjacent to the worker's position.
     - Be unoccupied by another worker or dome.
     - Be within the board's boundaries.
     - Not exceed the maximum tower height (3).
   - **Validation Logic**:

```
public boolean isValidBuildCell(int workerX, int workerY, int buildX,
int buildY) {
    if (!isWithinBounds(buildX, buildY)) return false;
    if (isOccupied(buildX, buildY)) return false;
    if (Math.abs(workerX - buildX) > 1 || Math.abs(workerY - buildY) >
1) return false;
    return true;
}
```

2. **Demeter-Specific Validation**:
   - A second optional build is allowed but must not be on the same space as the first build.
   - **Validation Logic**:

```
if (this.extraBuildUsed && buildX == firstBuildX && buildY ==
firstBuildY) {
    throw new Exception("Second build cannot be on the same space as
the first build.");
}
```

b. State Updates

When a build action is performed, the following updates occur:

1. **Board State**:
    - The height of the tower at the target cell is incremented.
    - For domes, the height is set to 4.
2. **Demeter-Specific State**:
    - A flag `extraBuildUsed` tracks whether the second build has been performed.
3. **Game Phase**:
    - After the first build, the player can perform an optional second build.
    - After the second build or skipping it, the phase transitions to `END_TURN`.

---

2. Alignment with Interaction Diagram

The following sequence of actions matches the interaction diagram:

1. **Player Action**:
    - Player requests a build action through the `Game.build()` method.
2. **Validation**:
    - `Game` delegates validation to `Board.isValidBuildCell()` for general rules.
    - If the player has the Demeter card, `DemeterGodStrategy` enforces additional rules.
3. **State Update**:
    - `Board.build()` updates the tower height at the target cell.
    - `DemeterGodStrategy` tracks the state of the optional second build.
4. **Phase Transition**:
    - If a second build is performed or skipped, the phase transitions using `Game.setCurrentPhase()`.

---

3. Responsibility Assignment and Justification

a. Board

- **Responsibility**: Validate build positions and update tower heights.
- **Justification**:
    - The `Board` encapsulates grid-specific operations, ensuring **Single Responsibility Principle (SRP)** and **high cohesion**.

b. Game

- **Responsibility**: Orchestrate build actions and delegate logic to appropriate classes.
- **Justification**:
    - The `Game` manages high-level flow and delegates validation to `Board` and rules to `GodStrategy`, achieving **Separation of Concerns**.

c. DemeterGodStrategy

- **Responsibility**: Enforce Demeter-specific rules and track build state.
- **Justification**:
    - The **Strategy Pattern** ensures extensibility and adheres to the **Open-Closed Principle (OCP)**, allowing new God cards to implement unique rules.

Code Example

```
@Override
public boolean build(Game game, Worker worker, int x, int y) throws Exception {
    if (!game.getBoard().isValidBuildCell(worker.getX(), worker.getY(), x, y)) {
        throw new Exception("Invalid build location.");
    }
    game.getBoard().build(x, y);
    if (!this.extraBuildUsed) {
        this.extraBuildUsed = true; // Allow the second build
    } else {
        game.setCurrentPhase(GamePhase.END_TURN); // End turn
    }
    return true;
}
```

---

4. Engagement with Design Principles and Tradeoffs

Design Principles

- **Open-Closed Principle (OCP)**:
  - The `DemeterGodStrategy` extends base functionality without modifying core game logic.
- **Single Responsibility Principle (SRP)**:
  - Validation, state management, and rules are separated into distinct classes.
- **Separation of Concerns**:
  - `Board` handles grid-level logic.
  - `DemeterGodStrategy` manages Demeter-specific rules.

Tradeoffs

1. **Alternative 1: Hardcoding Demeter Rules in `Game`**:
   - **Pro**: Centralized logic.
   - **Con**: Violates SRP and makes adding new God cards difficult.
2. **Alternative 2: Tracking State in `Game`**:
   - **Pro**: Simplifies strategy classes.
   - **Con**: Adds complexity to `Game`, increasing coupling.
3. **Chosen Design: Encapsulation in Strategy**:
   - Maintains modularity and reduces coupling.

---

5. Interaction Between Base Game and Demeter Card

Base Game Logic

1. `Game.build()` delegates the build action to the current player's `GodStrategy`.
2. `Board` handles general validation and state updates.

Demeter Card Logic

1. `DemeterGodStrategy.build()` overrides the base build logic to allow a second build.
2. The optional second build is enforced by `DemeterGodStrategy`, leaving the base game unaffected.

Code Integration

```
public class Game {
    public boolean build(int x, int y) throws Exception {
        return currentPlayer.getGodStrategy().build(this, selectedWorker, x, y);
    }
}
```

Conclusion

The design:

1. Clearly validates and updates build actions with the Demeter card.
2. Matches the interaction diagram's sequence of method calls.
3. Adheres to **SRP**, **OCP**, and **Separation of Concerns**.
4. Demonstrates thoughtful tradeoffs, prioritizing modularity and extensibility.
5. Seamlessly integrates base game logic with Demeter-specific behavior using the `Strategy Pattern`.