Design rationale:

1. Worker class

The Worker class is an essential part of the fundamental gameplay of Santorini. It is an individual piece (1 Player has access to 2 Workers) controlled by each Player. It is a fundamental unit in gameplay that moves across the board and performs building actions per turn to eventually win the game.
Creating Worker as a class is justified because:

- Each Worker holds state: its owner, canMove and currentCell are distinct attributes that persist and evolve across turns.
- It encapsulates behavior relevant only to the game piece: future methods such as getCurrentCell, or interaction with SpecialPower would logically belong here.
- The game enforces individual behavior rules per worker. For example, only a specific worker (not both) can act each turn (i.e only workers that moved in that turn), which would be hard to track if workers were just anonymous values in a list.

Thus, Worker is a domain object, not a utility holder making it appropriate and necessary as a class.

1. 3 key relationships for Worker:
    a. Player to Worker (Aggregation)
    Cardinality: 1 Player has 2 Workers (1 to 2)
    Rational for aggregation: A Worker isn't strictly bound to the lifetime of a Player. Also, Worker has an identity beyond Player, like currentCell.

    b. Worker to Cell (Association)
    Rationale for association: Movement is location-based and must always have a valid current cell. It's an association rather than aggregation/composition since Worker uses the Cell, but doesn't own or manage it.

    c. NEED ONE MORE ON THIS BUT COULDN'T FIGURE OUT RATIONALE
2. Our decisions around inheritance for the worker class, why we didn't decide to use it:

    The Worker class does not inherit from any superclass because all workers in Santorini exhibit uniform behavior and do not require polymorphism to differentiate their roles. Each worker moves and builds using the same rules, and there are no distinct types of workers that would necessitate subclassing. Any variations in how a worker behaves—such as building twice, moving before building, or swapping places—are handled externally through SpecialPower or GodCard objects, not within the Worker class itself. This design choice aligns with the principle of composition over inheritance, as it allows external components to modify worker behavior without bloating the class hierarchy. Furthermore, introducing a superclass like GamePiece or BoardEntity would add unnecessary abstraction and risk violating the Single Responsibility Principle by grouping unrelated entities under a shared parent. By keeping Worker as a standalone concrete class and delegating special behaviors to

other parts of the system, the design remains clean, modular, and more adaptable to future extensions without being locked into an inheritance-based structure.

3. 2 sets of Worker class cardinality explanations:
    a. Cell to Worker: 0..1
    A cell can be:
    Unoccupied is 0 and Occupied by one worker is 1. A cell can never have more than one worker (enforced by game rules).
    Worker to Cell: 1
    A Worker must always be standing on exactly one Cell (even after being placed initially or after a move). This is enforced in the game's movement and building logic.

    b. Player to Worker: 1..2
    A Player always has exactly two Workers per game rules.
    Why 2? It's a fixed rule in Santorini. Why not 1..*? Because indicating extra or fewer workers might make the UML diagram complicated to understand and it's always a fixed number.
    Worker to Player: 1..1
    A worker can be assigned to only 1 player.

4. Design Patterns Consideration
    The Worker class does not directly implement any well-known design patterns, and this is an intentional and appropriate design choice in our opinion. While design patterns offer useful abstractions for managing complexity and promoting reusability, they should only be applied when the underlying problem justifies their use. Below is an expanded evaluation of three design patterns, Strategy, State, and Observer, and why they were not applied to the Worker class in our Santorini game:
    1. Strategy Pattern
    The Strategy pattern is intended to encapsulate interchangeable behaviors and algorithms, allowing an object's logic to be chosen at runtime. This is effectively used elsewhere in our system, for example, in the RandomPlacementStrategy class which provides random worker placement logic during the setup phase. However, within the Worker class itself, this pattern is not necessary. Workers do not have random but rather controlled behavior at runtime. Their actions such as moving or building are determined externally, typically through a combination of Action, SpecialPower, and validation logic. Including Strategy here would introduce unneeded abstraction, making the codebase more complex without enhancing functionality or flexibility.

    2. State Pattern
    The State pattern allows an object to alter its behavior based on its internal state, modeling transitions between states like "active", "disabled", or "waiting". This could theoretically be applied to Worker if our design included features like freezing, empowering, or restricting a worker based on turn events or power effects. However, in our implementation, such conditions are handled externally. Whether a worker is eligible to perform an action is determined by Action logic or by SpecialPower, not by the worker changing its internal state object. This approach keeps the worker's

responsibilities minimal and aligns well with the Single Responsibility Principle, ensuring the object only models ownership and position, not behavior logic.

3. Observer Pattern

The Observer pattern is used to notify multiple dependent components automatically when an object's state changes. While this is useful in scenarios involving UI updates, real-time monitoring, or event-driven architectures, it would be excessive in the context of our Worker class. For example, if every move or build required broadcasting updates to listeners like the game board, UI, or other components, it could justify using Observer. However, our design handles updates in a more direct and controlled way, actions are triggered through explicit calls and game logic flows in a turn-based, sequential manner. Introducing Observer here would add unnecessary indirection and event management complexity for a game that thrives on simple, turn-by-turn control.