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. **2 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.
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.

2. Cell class

The Cell class represents a single tile on the Santorini game board. Each Cell holds its position (x, y), any structures built on it (Building and optionally a Dome), and may also be occupied by a Worker. The class extends JPanel, enabling it to serve as a visual and interactive element in the Swing GUI. This integration of game state and UI makes Cell a core component for both gameplay logic and interface rendering.

This class is necessary as a standalone object because it encapsulates multiple independent properties (position, occupancy, highlighting, and structural content), each of which evolves dynamically throughout the game. If this functionality were handled solely through methods or a generic board data structure, the design would become rigid, difficult to update, and hard to synchronize with the graphical interface. Additionally, since each cell must track its own state (occupied or not, highlighted or not, has dome or not), a class representation enables clean encapsulation and object-specific updates through repainting and logic validation.

1. **3 key relationships for Cell:**
    i. Cell and Building (Composition)

    The Building object is created and managed entirely within the Cell, with the constructor initializing it (this.building = new Building();).

    This is a composition relationship because the Building is owned by the Cell, and its lifecycle is tied to the Cell. If a Cell is removed, the associated building should not persist independently.

    ii. Board and Cell (Composition)

    The Board is composed of a number of Cell objects (5x5 grid in Santorini in this version), each representing a tile on the game map.

This is a classic whole-part composition, where the Board owns and manages its Cells. The Cells do not exist independently of the board. They are created during board initialization and are not reused or shared across boards.

The cardinality is typically 1 Board to many Cells, and the life cycle is tightly bound: deleting the Board should logically delete its Cells.

Composition is appropriate because a Cell is not meaningful outside the context of a specific board.

2. **Our decisions around inheritance for the cell class, why we didn't decide to use it:**

   The Cell class extends JPanel to integrate directly with the Java Swing framework. This inheritance is justified because Cell is fundamentally a visual component and must override the paintComponent(Graphics g) method to draw game elements like a Worker circle. Unlike domain-specific inheritance (between SpecialPower and its subclasses), this is a case of framework-driven inheritance: extending a GUI base class to gain rendering and event-handling capabilities. This avoids reinventing UI behavior and leverages polymorphism where it adds actual value, namely, in custom drawing and display updates.

   Within the domain model itself, Cell does not use inheritance. There is no need for different types of cells (OceanCell, UsedCell) in the base Santorini game. it would make sense to create a superclass like GameTile or BoardElement if the system had different types of board tiles with distinct rules, such as impassable tiles, trap tiles, or power-up zones. However, in the base Santorini game, every cell on the board is functionally identical in structure and rules. All cells support the same operations: holding a worker, building up levels, domes, and tracking state like occupancy or highlighting. This eliminates the need for polymorphism, rendering inheritance both unnecessary and potentially confusing.

3. **3 sets of Cell class cardinality explanations:**

   A. Cell and Building: 0..1
   A Cell may contain zero or one Building.
   This is modeled as 0..1 because:
   When the game starts, a cell has no built structure, the building level reference is null. Over time, a Building may be added or modified through game actions.

   B. Board and Cell: 4..*
   A Board may contain at least 4 Cells.
   A board is never empty, it must always have a minimum of four cells to be playable (2 workers each player). But the possibility of potential dynamical construction or deconstruction of the board grid (useful for extensions or alternate board sizes), thus not 1..25. It also aligns with composition (black diamond in UML), where the board owns its cells, meaning if the board is deleted, its cells are destroyed too.

   C. Action and Cell: 1
   An Action (MoveAction, BuildAction) must target exactly one Cell.
   This is modeled as a mandatory association with a 1-to-1 cardinality because you cannot perform an action without specifying a cell because the cell being targeted is

integral to validating and executing the action. Every gameplay action like movement, movement, and validation affects a specific cell.

4. **Design Patterns Consideration:**

   The Cell class does not explicitly implement a design pattern but indirectly aligns with the Model-View aspect of MVC (Model-View-Controller). Because it extends JPanel and contains both visual (paintComponent) and logical state (occupancy, dome, highlight), it acts as a hybrid of view and model. This was done this way due to the simplicity and small scope of the board.

   Other classic patterns like Composite or Flyweight etc are not applied here because: Composite would be used if cells contained hierarchical sub-elements (nested structures), which is not the case.

   Flyweight might reduce memory use for uniform cells, but our design prioritizes clarity and per-cell customization.

3. TurnManager class

The TurnManager class is responsible for controlling the flow of the game. It handles the sequence of phases (movement, building, god powers), player switching, turn-based UI updates, and win/lose conditions. This class is essential because it acts as a coordinator to organize interactions between multiple classes such as Player, Worker, Cell, Board, and Action.

It is appropriate to implement TurnManager as a full class rather than distributing its logic across Player, Game, or Board, because it:

- Encapsulates the rules and sequencing logic that are essential to individual player or board behavior.
- Maintains stateful data across turns (current player, last moved worker, floor counters).
- Simplifies unit testing and debugging by centralizing turn logic.

If this logic were split into methods in other classes (Player.move() or Board.processTurn()), the result would be high coupling and reduced maintainability.

1. **2 key relationships for TurnManager:**

   TurnManager and Player: Association

   This is a simple association because the TurnManager references the players in order to rotate turns, highlight current workers, and determine win/loss conditions but it does not own or create the Player objects. The Player instances may be constructed elsewhere (during game setup) and passed into the TurnManager. Additionally, removing the TurnManager does not destroy the players, and players can be reused or transferred across different contexts. Therefore, this is best modeled as a regular association, it reflects usage, not ownership.

   TurnManager and Board: Aggregation

   The relationship between TurnManager and Board is an aggregation, because the Board is a shared, externally created object that the TurnManager coordinates with but does not own. The board's lifecycle is independent of the TurnManager, for instance, other components like BoardUI also reference the board. The board is passed into the TurnManager via its constructor, reinforcing that it is an external dependency rather than a

managed subcomponent. Aggregation is appropriate here as it models a shared resource relationship, the Board is central to the game but not subordinate to the TurnManager.

TurnManager and Worker: Association
The TurnManager maintains a list of the current player's Workers during their turn for highlighting and movement purposes. However, it does not own or manage the full lifecycle of these Worker objects. The Workers are owned by the Player, and TurnManager simply uses them as part of the turn-processing logic (highlightCurrentWorkers, lastMovedWorker). Since workers exist independently of the TurnManager, this relationship is, we think, as best modeled as a simple association, the manager depends on these objects during a turn, but they are not part of its composition or aggregation.

2. **Our decisions around inheritance for the worker class, why we didn't decide to use it:**
   TurnManager does not use inheritance, and this is appropriate for several reasons:

   - It doesn't share behavior or structure with any other class. It is a coordinator/controller, not a domain entity like Player or Worker.
   - There's no need to subclass it (no AdvancedTurnManager or TimedTurnManager).
   - Inheritance would introduce unnecessary abstraction and tight coupling in a class that already depends on multiple components.

   This aligns with the Single Responsibility Principle. TurnManager governs the turn system (Can you build? Can you move? etc) and nothing else.

3. **2 sets of TurnManager class cardinality explanations:**
   TurnManager and Player: 2
   The manager holds a list of players and always needs at least one player to function.
   Justification: A game requires 2 players minimum (and maximum). The list is rotated to change the current player after each turn.

   TurnManager and Worker: 0..*
   Holds a list of the current player's workers for selection and highlighting.
   Justification of 0 (Lower Bound):
   At certain points in the game lifecycle (before the first turn starts, after game over), the TurnManager may not hold a reference to any workers. This assignment only happens when a new turn starts. Until that assignment, or if something goes wrong, the list could be empty or uninitialized.

   Justification of * (Upper Bound):
   In standard gameplay, the number of workers held is always 4 (because each player has exactly 2 workers). But to keep the model flexible and general, * is used to allow for:

   - Potential future rules with more than 2 workers per player

- Code reuse across different modes (e.g., training simulations, AI testbeds)
- Cleaner UML, since hardcoding 2 in the cardinality would be too rigid for future extensions.

Game and TurnManager Cardinality: 0..1
The Game class creates the TurnManager and passes all required components (players, board, ui) to it.
However, the TurnManager is not stored as a field inside Game — it's a local variable within startGame():
TurnManager gm = new TurnManager(players, board, ui);

Because the Game does not retain a persistent reference to the TurnManager, this is a transient association, not aggregation or composition. Since TurnManager is instantiated and used only inside the startGame() method, the Game holds no ongoing reference to it. If we were to extend the game logic (pause/resume, replays, turn history), we could store TurnManager as a field, which would change this cardinality to 1.

But the TurnManager does not know about the Game class at all — there is no reference to Game inside TurnManager. Therefore, from TurnManager to Game, the cardinality is not applicable. This is a unidirectional association from Game to TurnManager.

4. **Design Patterns Consideration**

Mediator Pattern
The TurnManager class strongly applies the Mediator Pattern by acting as the central coordinator for all game components. Rather than allowing Player, Worker, Board, Cell, Action, and the UI to interact with each other directly, the TurnManager manages and sequences these interactions. For example, it determines which player's turn it is, highlights valid Cells for movement or building, passes control to Action objects for execution, updates the game state, checks win or loss conditions, and communicates relevant changes to the user interface. This centralized control flow reduces tight coupling across components, making the system easier to modify and extend. If a new god power or turn rule is introduced, it can be integrated by updating the TurnManager without needing to change the internal behavior of individual classes like Player or Board. By encapsulating the control logic in one place, the TurnManager enhances the modularity, maintainability, and clarity of the entire system, exemplifying the Mediator Pattern in practice.

3 General key relationships in our class diagram:

Board and Cell: Composition
This relationship is a composition because Cells are fully owned and controlled by the Board. They are created as part of the board's initialization and have no independent lifecycle outside of it. If the Board is deleted or reset, all its Cells are destroyed with it. This is a whole-part relationship: a Cell cannot exist meaningfully without a Board. Therefore, composition is appropriate. It expresses a strong lifecycle dependency.

Player and Worker: Aggregation
This relationship is an aggregation because while a Player owns 2 Workers during gameplay, the Workers are not tightly bound to the Player's lifecycle. For example, even if a player is removed from the game, the Workers might still exist momentarily during cleanup or win condition checks. They also maintain their own identity (currentCell, owner, etc) and interact independently with the board. Therefore, aggregation accurately models the "has-a" relationship without implying destruction of Workers when a Player is removed.
Action and Cell: Association
This is a simple association, not an aggregation or composition, because Action instances do not own or manage the Cell they affect. The Cell exists independently of the Action, and multiple actions can affect the same cell over the course of the game. The Action merely references a Cell as a target during execution (e.g., for moving or building). The association is temporary, contextual, and unidirectional, which makes aggregation or composition inappropriate.