FIT3077 Software Engineering: Architecture and Design

Sprint Three

Student ID: 33198489

Santorini Game

Table of contents

Introduction

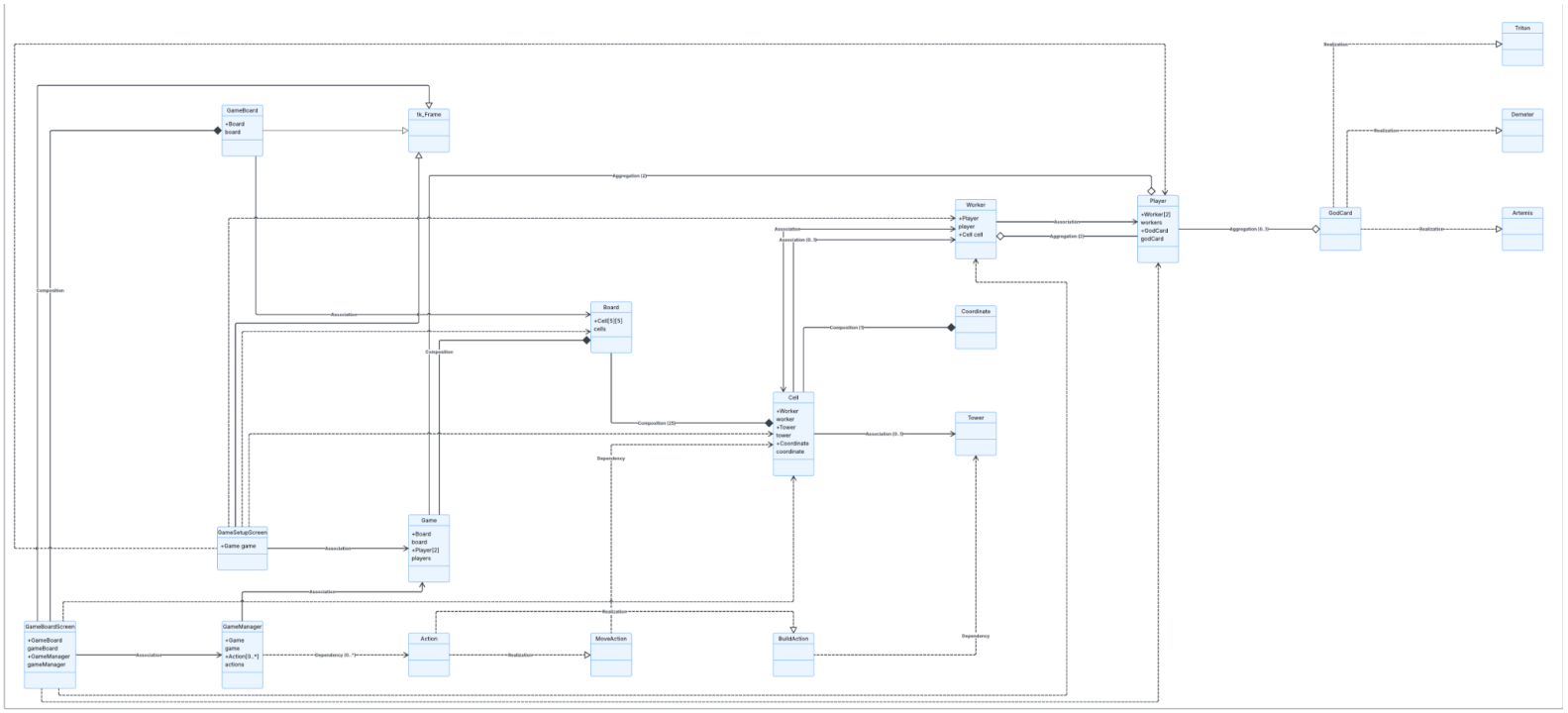
Sprint 3 extends our Santorini implementation with three major capabilities:

- Triton god power (unlimited perimeter moves in a single turn)
- 15-minute per-player chess-clock
- Hidden “ancestral” cells that reward players with bonus thinking time

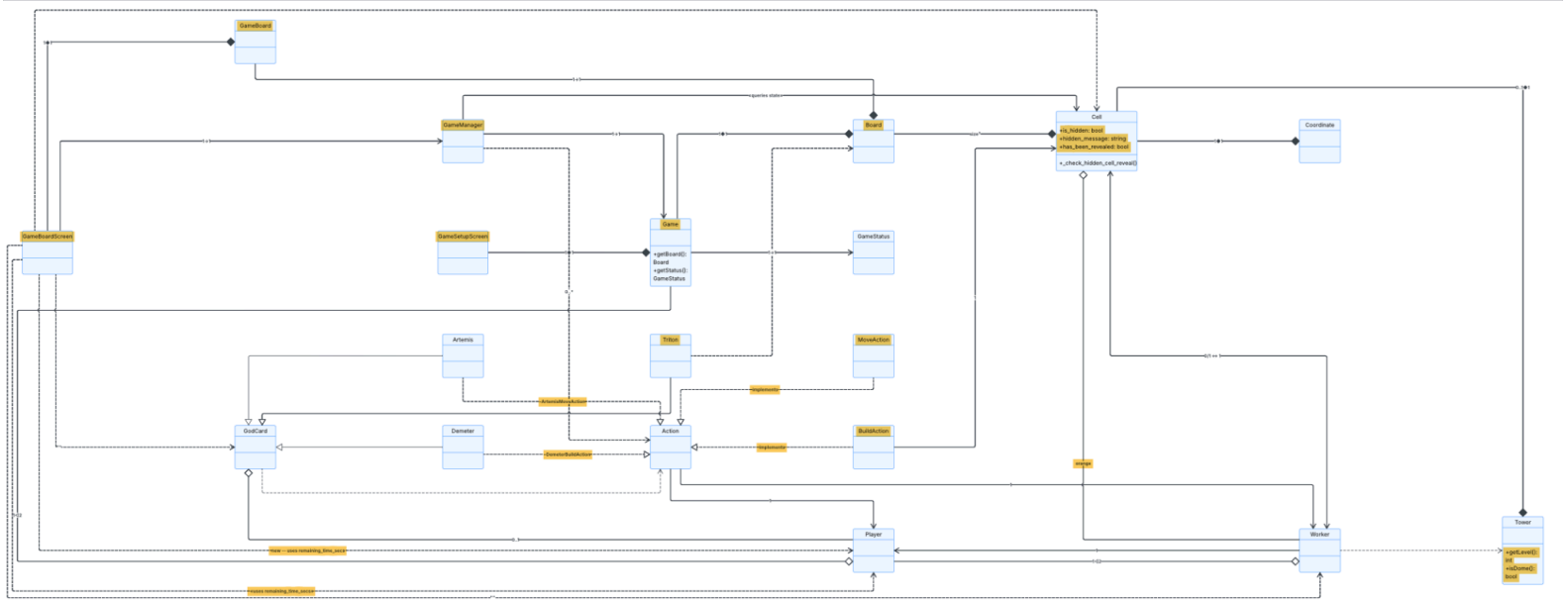
Instead of adding large numbers of new classes, we evolved the existing architecture to remain cohesive, testable, and SOLID-compliant. This document explains why each change was made, evaluates alternative designs, and demonstrates how the final architecture realises the required human value of Tradition.

Update UML Class Diagrams

UML_V1



UML_V2(Final version with updated highlights)



Class-Responsibility-Collaborator

Class: GameManager	
Responsibilities	Collaborators
Control game flow and status.	Game (to get players, update status)
Track current player and turns	Player (to get god card, apply effects)
Execute and validate actions(move/build)	Action subclasses (to execute/validate)
Check win/loss conditions	MoveAction(to move a worker), BuildAction(to build) , GodCard (just applying the god card effects)
Handle hidden cell reveals	Cell, Worker

Class: GodCard (abstract + Artemis/Demeter/Triton)	
Responsibilities	Collaborators
Define unique god powers (extra moves/builds).	Player (to fetch god card)
Validate rule exceptions for actions.	Action, MoveAction(applying move god powers) , BuildAction(applying build god powers)
Maintain internal state (e.g., first move/build location).	Cell (to check god power for Triton)
Reset god power state at end of turn.	GameManager (to evaluate effects)

Class: Action (abstract + MoveAction, BuildAction)	
Responsibilities	Collaborators
Define interface for game actions.	Player, Worker
Enforce game rules for execution.	Cell (target cell for move/build)
Validate legality of move/build.	GameManager (for execution/validation)

Class: Board	
Responsibilities	Collaborators
Manage grid and cells.	Coordinate, Cell, Worker
Provide adjacency and movement logic.	Player (for placing workers)
Place workers and hidden cells.	Tower (for buildability checks)
Return valid move/build options.	

Classes and Interfaces Explanation

We touched almost every layer. In the user-interface layer GameBoard now only draws the board and tells its parent when someone clicks; it no longer moves pieces itself. GameBoardScreen became the place that runs a whole turn—it shows clocks, pops up hidden-cell messages, and knows when Triton may move again. GameSetupScreen gained text boxes for player names plus radio buttons for board size; it also places workers by calling a helper on the board, not by looping inline.

Inside the engine we removed the Singleton from GameManager, so we can run several games in parallel during tests. This manager now holds the “template” for a turn: check whether the move is legal, perform the action, apply any god power, and finally ask the game if someone won or ran out of time. The Game object itself grew a method to see whether a player has no legal moves—a natural place because the game already knows the whole board.

Because every god will want something different, we stripped Artemis and Demeter logic out of MoveAction and BuildAction and let each GodCard answer questions like “Is this move legal for me?” That avoids the class explosion we would face if we wrote one action subclass per god. Supporting pieces also changed: Board now handles three possible board sizes and owns a list of hidden cells; Cell now knows whether it is hidden or already revealed; and Player stores its own countdown in seconds so the clock works even if we swap Tkinter for a different UI in future. Finally, Tower hides its internal numbers and refuses illegal builds, so we cannot break the board by mistake.

Core Game Logic Evolution

GameManager - Enhanced Coordination Hub

Why it needed modification:
Sprint 2's GameManager used a singleton pattern that created hidden global state and prevented parallel test execution
The turn validation logic was scattered across Action classes and GameManager, violating Single Responsibility Principle
No support for time-based gameplay or special cell interactions

Key modifications:
Removed singleton pattern: Replaced private _instance with normal instantiation, enabling parallel game instances and better testability
Centralized validation: Moved all turn validation logic into GameManager.validate_turn(), consolidating generic rules with god-card-specific restrictions
Added timer integration: Incorporated timer checks and time-based loss conditions
Hidden cell support: Added check_hidden_cell_reveal() to handle ancestral message rewards while maintaining separation of concerns

Could existing classes handle these responsibilities?

No. The Game class manages state but shouldn't handle turn flow logic. Action classes handle atomic operations but can't coordinate multi-step game sequences. GameManager's role as orchestrator is essential and justified.

Game - State Management Hub

Why it needed modification:
Required support for dynamic board sizes (4×4, 5×5, 6×6)
Needed integration with random worker placement for larger boards
Had to incorporate loss condition checking for immobile players

Key modifications:
Random worker placement: Added call to board.place_workers_randomly() in initialize_game() to support variable board sizes
Loss condition centralization: Added check_lose_condition(player) method that delegates to player.has_valid_moves(board), removing this logic from GameManager
Maintained aggregate root pattern: Continues to own Board and Player objects while providing clean interfaces

Could existing classes handle these responsibilities?
The GameManager could theoretically handle loss checking, but this would violate Single Responsibility - GameManager should coordinate turns, not determine game-ending conditions. Game is the natural owner of match-level rules.

Action Class Hierarchy - Streamlined Command Pattern

Why the abstract Action class needed enhancement:
Sprint 2 had redundant parameter passing across all Action subclasses
Each subclass duplicated player/worker initialization
The god-power-specific Action subclasses (ArtemisMoveAction, DemeterBuildAction) didn't scale

Key architectural decision:
Consolidated initialization: Moved player and worker attributes to the base Action class constructor, eliminating duplication
Eliminated god-specific Action subclasses: Removed ArtemisMoveAction and DemeterBuildAction, moving their logic to respective GodCard classes
Maintained polymorphism: Kept the is_valid() and execute() contract while simplifying the hierarchy

Alternative design considered:
We could have kept separate Action subclasses for each god power, but this would create 35+ classes (one per official god card), violating the Open/Closed Principle. Our chosen approach allows new god powers without new Action classes.

MoveAction - Simplified Universal Movement

Why it needed modification:
Sprint 2 version contained Artemis-specific logic, violating Single Responsibility
Returned god-specific flags like "SECOND_MOVE", coupling it to specific powers
Lacked debugging capabilities for complex turn sequences

Key modifications:
Removed god-specific logic: All Artemis constraints moved to Artemis.can_move_to_cell() method
Eliminated special return flags: No longer returns "SECOND_MOVE"; god powers handle their own state in apply_god_power()
Enhanced debugging: Added __str__ method for clearer console output during timer and hidden-cell testing
Streamlined validation: Now handles only universal movement rules (adjacency, height difference, non-stationary)

Could existing classes handle these responsibilities?
The god-specific logic could have remained here, but this would require MoveAction to know about every god power's rules, violating Open/Closed Principle. Our refactor ensures new god powers require no changes to MoveAction.

BuildAction - Simplified Universal Build

Why it needed modification:
Sprint 2 version contained Demeter-specific branching logic
Returned god-specific flags like "SECOND_BUILD"
Duplicated player/worker initialization with other Action subclasses

Key modifications:
Removed Demeter-specific logic: All extra-build logic moved to Demeter.apply_god_power() method
Eliminated special return flags: No longer returns "SECOND_BUILD"; Demeter class manages its own state
Inherited common initialization: Now calls super().__init__() instead of manual field assignment
Enhanced debugging: Added __str__ method for development and testing support

Could existing classes handle these responsibilities?
Similar to MoveAction, keeping god-specific logic here would require BuildAction to know about every god power's building rules. The refactor maintains reusability across all god powers.

Eliminated Classes - Architectural Simplification

ArtemisMoveAction - Removed for Scalability

Why it existed in Sprint 2:
Implemented Artemis's "second move but not back to starting cell" rule
Contained duplicated validation logic from base MoveAction
Represented the "one class per god power" approach

Why removed in Sprint 3:
Scalability issue: This approach would require 35+ MoveAction subclasses for all official god cards
Code duplication: Repeated adjacency and movement validation from MoveAction
Maintenance burden: Changes to movement rules would require updates across multiple classes

Responsibility migration:
Validation logic: Moved to Artemis.can_move_to_cell() method
Execution logic: Standard MoveAction.execute() now handles all movements
State management: Artemis class now tracks original position internally

DemeterBuildAction - Removed for Consistency

Why it existed in Sprint 2:
Handled Demeter's "build twice on different cells" special rule
Duplicated build validation from BuildAction
Created inconsistent action handling across god powers

Why removed in Sprint 3:
Consistency: Matches the MoveAction refactor approach
Reduced duplication: Eliminated repeated adjacency and build validation
Simplified game flow: GameManager no longer needs to handle different action types per god

Responsibility migration:
Extra build logic: Moved to Demeter.apply_god_power() method
Build validation: Standard BuildAction now handles all builds
State tracking: Demeter class manages first build location internally

Game State and Board Management

Board - Enhanced Spatial Intelligence

Why it needed extensive modification:
Sprint 2 couldn't handle dynamic board sizes (4×4, 5×5, 6×6)
No support for random worker placement across variable grid sizes
Missing infrastructure for hidden cell mechanics (human values extension)
Inefficient adjacency calculations repeated for every query

Key modifications:
Dynamic board sizing: Constructor now accepts size parameter, supporting 4×4, 5×5, and 6×6 configurations
Random worker placement: Added place_workers_randomly() method to distribute workers across ground-level cells automatically
Hidden cell infrastructure: Added create_hidden_cells(), get_hidden_cells(), and hidden_cells_created flag for ancestral message extension
Optimized adjacency: Replaced per-query coordinate calculations with pre-computed get_adjacent_cells() method
Enhanced cell queries: Rewrote get_available_move_cells() and get_available_build_cells() to use cached adjacency data

Could existing classes handle these responsibilities?
Cell could theoretically handle its own adjacency, but this would require every cell to know about the entire board structure. Game could handle worker placement, but this would violate Single Responsibility - Game manages overall state, not spatial arrangements. Board is the natural owner of all spatial logic.

Cell - Comprehensive State Management

Why it needed significant enhancement:
Sprint 2 had redundant occupied boolean flag alongside worker reference
No support for hidden cell mechanics required by human values extension
Movement and building validation scattered across multiple classes
Lacked comprehensive state representation for debugging

Key modifications:
Eliminated redundancy: Removed occupied flag; occupancy now determined by worker is not None
Hidden cell mechanics: Added is_hidden, hidden_message, has_been_revealed attributes and reveal_hidden_cell() method
Centralized validation: Enhanced can_move_to() to enforce climb-by-one rule directly; improved is_available_for_build() delegation
Safer worker management: All worker placement now goes through assign_worker()/remove_worker() methods, preventing state inconsistencies
Enhanced debugging: Improved state representation for development and testing

Could existing classes handle these responsibilities?
Board could manage hidden cell state, but this would require Board to track fine-grained cell details, violating encapsulation. Worker could handle movement validation, but this would duplicate logic across all workers. Cell is the natural owner of its own state and validation rules.

Game Entities and Rules

Player - Enhanced State Container

Why it needed modification:
Required timer support for 15-minute thinking time extension
Needed ability to determine if player has any valid moves (loss condition)
Had unnecessary game mechanic methods that violated Single Responsibility

Key modifications:
Timer integration: Added remaining_time_secs field to support countdown timer extension
Loss condition support: Added has_valid_moves(board) method to check if player is immobilized
Simplified responsibilities: Removed direct move/build helper methods, keeping Player as pure data container
Enhanced state management: Player now focuses solely on storing player state, not executing game mechanics

Could existing classes handle these responsibilities?
GameManager could track timing, but this would scatter player-specific state across multiple classes. Board could check valid moves, but this would require Board to understand player-worker relationships. Player is the natural container for player-specific state and queries.

Worker - Safer Game Piece Operations

Why it needed modification:
Sprint 2 build logic had potential failure cases without proper error handling
Console output cluttered development testing
String representation insufficient for debugging complex scenarios

Key modifications:
Safer building logic: Enhanced build methods to check tower level before building, returning success/failure status
Cleaner output: Removed console print statements, keeping methods purely functional
Better debugging: Improved __str__ output to show essential worker information (id, coordinate, owner)
Robust validation: Build operations now validate tower state before attempting modifications

Could existing classes handle these responsibilities?

Cell could handle building logic, but this would require Cell to understand worker ownership and permissions. Tower could validate builds, but Worker needs to coordinate between Cell and Tower. Worker's role as the active agent remains appropriate.

Tower - Defensive Construction Rules

Why it needed modification:
Sprint 2 allowed invalid tower states through direct field access
No validation during tower creation
Build operations didn't provide success/failure feedback
Potential for inconsistent tower states across the game

Key modifications:
Constructor validation: Added safety checks preventing illegal initial states (height > 3, dome below level 3)
Encapsulated fields: Made level and dome private with getter methods, preventing external manipulation
Return status reporting: Build and dome methods now return True/False to indicate success
Consistent state guarantees: Interface changes ensure towers maintain valid states throughout game lifecycle

Could existing classes handle these responsibilities?
Cell could validate tower operations, but this would create tight coupling between Cell and Tower internals. Worker could handle tower validation during builds, but this would duplicate validation logic across all workers. Tower's responsibility for its own integrity is well-placed.

UI Modifications

These three classes now focus purely on UI sequencing (GameBoard), (GameBoardScreen) and one-time setup (GameSetupScreen), while rules, powers and core data stay in lower layers.

GameBoard - Visual representation of the Santorini game board.

Why it exists?
GameBoard is the visual layer only. It draws towers and workers, lets the user click a square or a worker, colours legal targets, and then tells a higher-level screen “the player clicked here.” Keeping display separate from rules means the UI can change (new colours, bigger tiles, drag-and-drop, etc.) without touching any game mechanics.

• Key Modifications:
– Stripped out game logic. The old perform_move_or_build_action() and direct worker.move_to() calls are gone; the frame no longer moves pieces itself.
– Clearer state: keeps just three UI variables—highlighted_cells, selected_cell, and the canvas dict—making repaint logic simpler and bug-free.
– Reusable helpers: _draw_tower, _draw_worker, highlight_cells(), and select_cell() give the parent screen fine-grained control without duplicating code.
– Consistent look & feel: every square starts light-blue, highlights green, selection navy, and outline indigo—easy to theme later.
By slimming GameBoard down to “draw + notify”, the program now follows Single-Responsibility more closely and any future extension (hex grid, animation, new colours) lives in one place.

GameBoardScreen (responsible for running a whole ‘Turn’ on-screen)

• Why it exists
– Shows the board, buttons and messages — and decides what step comes next (pick worker → pick move → confirm → pick build → confirm → finish turn).
– In Sprint 3 it also handles the 15-minute countdown clock, the hidden-cell “+10 s” bonus, a draw proposal, and a full-screen game-over panel. Keeping all this turn-flow logic in one class lets the deeper engine stay UI-free.

• Key modifications:
– Replaced scattered booleans with a clean TurnPhase enum so button enabling/disabling is automatic and easy to read.
– Moved all Artemis/Demeter logic into their own card classes; the screen now just looks at the returned tag and shows a pop-up.
– Added _tick_timer() plus timer labels; when a clock hits zero _handle_time_expired() ends the game.
– Added _check_loss_condition() at turn start (no valid moves → lose) and _propose_draw() for a consensual draw.

– Layout split into helper builders (_create_info_panel, _create_control_panel, etc.) and helper updaters (_update_display, _update_button_states) so each method is short and has one job.

GameSetupScreen (responsible for collecting names / board size and launching a match)

• Why it exists
– Gives players a quick wizard to type their names, choose board size, shuffle God cards and randomly assign god cards as well as assigning the hidden cell locations, and click “Start Game.” It keeps setup details out of the game loop.

• Key modifications:
– Added entry fields for custom player names and a radio-button set for 4 × 4, 5 × 5 or 6 × 6 boards; validation ensures names differ and there’s enough space for workers.
– Token colours (player.token_color) assigned here so all drawing code can pick a consistent colour later.
– Worker placement moved into _place_workers_randomly() (and internally uses the board’s data) instead of inline loops; keeps the screen short even as board shapes change.
– Wrapped the whole start routine in a try/except; any error pops a neat “Setup Error” dialog instead of crashing the app.

Alternative Designs Discussion

God Power Architecture Alternatives

Considered: Command pattern for god-specific actions
Chosen: Consolidated Action classes with god logic in cards
Rationale:

- Prevents class explosion (35+ god cards × multiple action types)
- Easier to maintain and extend
- Better adherence to Open/Closed Principle

Timer Implementation Alternatives

Considered: Separate Timer class with Observer pattern
Chosen: Timer state in Player class with UI-driven updates
Rationale:

- Simpler implementation for two-player game
- Avoids over-engineering for current requirements
- Observer pattern would add complexity without clear benefits at this scale

Separate HiddenCellManager class

Considered: First planned to give hidden-cell logic its own object, but that meant every move would ask both Board (for geometry) and HiddenCellManager (for secrets).
Chosen: Hidden cells are still just squares on the board, we folded the flags (is_hidden, hidden_message, revealed) into Cell and let Board sprinkle them at setup.
Rationale:

- This keeps all spatial concerns in one place and avoids double-lookups.

Relationships, Inheritance, and Cardinality Justification

Composition – “lives-and-dies-with” relationships

- Game 1 -> 1 Board – If a game ends, its board goes with it.
- Game 1 -> 2 Player – Exactly two players per match.
- Player 1 -> 2 Worker – Santorini’s rules fix this at two.
- Board 1 -> N Cell – N equals size × size (16, 25 or 36).
- Cell (when hidden) -> 1 hidden_message (String) – Each hidden cell owns its unique message until revealed.

Aggregation / Shared Associations

- Board -> Worker – A worker stands on one cell at a time; the board merely keeps track of all workers for convenience.
- Cell 0/1 -> 1 Worker – Occupancy; breaks when the worker moves away.
- GameManager 1 -> many Action – Each turn creates one or more actions that vanish afterward.
- GameBoardScreen 1 -> 1 GameManager – One screen controls one manager
- GameSetupScreen 1 -> 1 Game – Setup creates a game but does not stay attached after the match window opens.
- Action 1 -> 1 Worker / 1 Cell – An action touches exactly one worker and (for builds) one cell; references are released once execute() returns.

Inheritance

- Action <- MoveAction, BuildAction – Two concrete commands share one interface.
- GodCard <- Artemis, Demeter, Triton – Three concrete strategies extend the abstract god class; more can join later with no edits elsewhere.

Design Patterns Analysis

Template Method Pattern

Definition and Purpose: The Template Method pattern defines the skeleton of an algorithm in a base class while allowing subclasses to override specific steps of the algorithm without changing its overall structure.

Implementation in Our Code: Located in logic/actions/action.py and its concrete implementations (BuildAction, MoveAction).
The abstract Action class establishes a consistent algorithmic framework for all game actions:

- Every action must implement is_valid() for validation logic
- Every action must implement execute() for execution logic
- The overall processing sequence remains constant across all action types

Why We Applied This Pattern:
Game actions need consistent behavior patterns while maintaining flexibility for different action types. For example, both move and build actions follow the same validation-then-execution sequence, but each has unique validation rules and execution mechanisms.

Benefits Achieved:

- Guarantees consistent action processing workflow
- Promotes code reuse through shared algorithmic structure
- Simplifies addition of new action types
- Reduces duplicate code between different actions

Observer Pattern

Definition and Purpose: The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all dependent objects are notified automatically.

Implementation in Our Code:

- Implemented through the hidden cell revelation system in Cell class and monitoring mechanisms in GameManager.
- When players move to hidden cells, the cell reveals its secret, and the game manager automatically responds by granting bonus time without the cell needing direct knowledge of the game manager.

Why We Applied This Pattern:
The game needed to support special events (hidden cell discoveries) without creating tight coupling between game components. The cell should focus on its state, while the game manager handles game-wide effects.

Benefits Achieved:

- Loose coupling between event sources and event handlers
- Easy extension with additional special events or cell types
- Clean separation of concerns
- Event-driven architecture for special game features

Composite Pattern

Definition and Purpose: The Composite pattern composes objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions uniformly.

Implementation in Our Code: Evident in the hierarchical relationship between Board, Cell, Tower, and Worker objects, where the board contains cells, and cells contain other game components.

Why We Applied This Pattern: The game board represents a natural hierarchy where operations can be performed at different levels (board-wide operations, cell-specific operations) while maintaining consistent interfaces.

Benefits Achieved:

- Uniform treatment of individual components and collections
- Simplified client code through consistent interfaces
- Easy traversal and manipulation of game board structure
- Natural representation of game component relationships

Human Values Integration

What to include: Explanation of your third extension's human value alignment
Key components:

Why the chosen human value (from the 58 values) is important
How it's manifested in the game (visible to players and markers)
What specific changes you made to enable this value

Chosen Human Value: Tradition

Why this value is important:
Tradition represents the passing down of wisdom, stories, and cultural knowledge from ancestors to descendants. In many cultures, ancestral wisdom is considered a source of guidance and strength during challenging times.
How it manifests in the game:

Hidden cells with ancestral messages: When players move onto hidden cells, they receive messages like "Your ancestors whisper words of encouragement" or "The wisdom of your forebears grants you clarity"
Time bonus reward: Players receive +10 seconds on their timer, representing the spiritual/mental refreshment that comes from connecting with ancestral wisdom
Visual and narrative integration: The messages explicitly reference ancestors, making the connection to tradition clear to players

What we changed to enable this value:

Cell class: Added is_hidden, hidden_message, and revelation mechanics
Board class: Added random hidden cell placement and management
GameBoardScreen: Added ancestral message display and timer bonus application
Game flow: Integrated hidden cell checks into the turn sequence

How players experience this value:
Discovery: Players explore the board knowing that ancestral wisdom might help them
Surprise and delight: Unexpected messages and time bonuses create positive emotional moments
Cultural resonance: The ancestral framing connects to universal human experiences of family wisdom and tradition
Strategic element: The time bonus has genuine gameplay impact, making tradition mechanically meaningful

Game Deviations

- Players who land on hidden cells must immediately build after receiving their bonus, representing the idea that receiving ancestral wisdom comes with the responsibility to act constructively.
- Players assigned the god power ‘Triton’ cannot build from the perimeter, they have to move again when on the perimeter.

SPRINT 2 Reflection

1. Hidden-Cell “Ancestral Blessing” Feature

Difficulty: Hard – The Board and Cell classes in Sprint 2 were built only for towers and workers, so they had no flag or storage for extra states such as “hidden” or “revealed.” I had to retrofit Boolean fields (is_hidden, has_been_revealed) and a message string into Cell, then teach Board and GameManager to coordinate reveal rules and time bonuses. Because GameManager was a singleton in Sprint 2, sharing the new reveal counter across test instances became brittle and hard to test in isolation.
Lesson / what I would change: I would have treated Cell as an extensible entity from day one—e.g., by giving it a lightweight “attributes” dictionary or by applying the Decorator pattern, so new behaviours (like “hidden cell”) could be added without touching core logic. I would also avoid a singleton GameManager; keeping it as a normal object would have removed the global-state headache.

2. Extra God Powers (Artemis, Demeter second action; Triton chain move)

Difficulty: Medium – Sprint 2 already separated “god power” logic, but actions themselves were hard-coded directly inside the UI layer. To add powers that trigger after an action, I created an Action base class plus concrete MoveAction and BuildAction classes, letting each god inspect the just-executed action and return a control string like "SECOND_MOVE". Refactoring into the Command pattern smoothed things out, but the work was bigger than it should have been.
Lesson: I would also have formalised post-action hooks in GodCard earlier so each power stayed fully within its own class rather than sprinkling checks through the GUI.

3. Turn Timer and Time Bonus Logic

Difficulty: Medium – Adding a per-player clock and the +10 seconds bonus was mostly straightforward because Player was already an independent data class. I only had to add remaining_time_secs and a small Tkinter loop in GameBoardScreen. The only pain point was again the singleton GameManager, which forced me to tuck timer cleanup into UI code rather than central game logic.
Lesson: If Sprint 2 had placed all turn-cycle responsibilities (phase control and clocks) inside GameManager, the UI could have stayed thinner and I would not have duplicated timing code in the screen.

4. Multi-Phase Turn Flow & UI Rework

Difficulty: Medium – In Sprint 2 the main screen did everything in one method chain, so extending it to handle second moves/builds, timer expiry, and draw offers meant splitting state into an explicit TurnPhase enum and many helper methods. The change was tedious but mechanical; the real blocker was coupling: UI widgets accessed game logic directly, so every small rules tweak forced UI edits.

Lesson: Had Sprint 2 enforced a cleaner Model-View-Controller split (e.g., screen talks only to a controller interface, never to Player or Board directly), I could have swapped in the new phases with far less repaint code.

5. Removing the Singleton GameManager

Difficulty: Annoying but Worth-it – The singleton saved a few lines early on but became a global bottleneck when I wanted parallel tests or multiple game sessions. Refactoring to an ordinary object touched many files that assumed GameManager.get_instance(). Once done, however, adding hidden-cell counters and timer hooks was much simpler.
Lesson: I would never introduce the singleton in Sprint 2; a single top-level “game” object passed down the call chain is clearer, testable, and future-proof.

Non-Technical Hurdles & Their Design Impact
Team-level issues—poor meeting attendance, decisions made without me, and work left for the end—left me responsible mainly for documentation. Even though, I did pair-program on the Sprint 2 code, I lacked familiarity with its structure, so every extension started with “learn how it works” before I could even code. Clearer collective ownership and early code walkthroughs would have revealed design smells (especially the singleton) sooner and spared us big rewrites in Sprint 3.

Testing

Our primary testing strategy followed an incremental implementation-and-observation cycle. After implementing each feature according to its specification, we initially tested correctness using print statements and internal state checks. Once integrated into the GUI, the features were further tested through interactive gameplay, ensuring functionality matched expected user behavior. While we did not employ formal unit testing frameworks in this sprint, testing was driven by real-time validation and user interactions.