1. Introduction

Sprint 3 significantly enhances the Santorini board game implementation through three major extensions: a competitive timer system, the Triton god card with advanced movement mechanics, and a comprehensive tutorial system that reflects the human value of helpfulness. In addition to these core features, Sprint 3 introduces extensive architectural refactoring that transforms Sprint 2’s flat, tightly coupled structure into a well-organized, modular codebase.

Sprint 2 delivered a functional **tkinter-based** implementation of Santorini, including core gameplay mechanics, two god cards (Artemis and Demeter), and basic UI functionality. Sprint 3 builds on this foundation with strategic refactoring, implementing clear separation of concerns, leveraging design patterns, and enabling scalable extensions that follow sound object-oriented design principles.

2. Class Design and Responsibilities

2.1 New or Updated Classes and Interfaces

2.1.1. Architectural Refactoring from Sprint 2

In Sprint 3, we restructured the architecture to address several design issues highlighted in the Sprint 2 feedback. One of the most impacted areas was the game logic, where certain classes, like TurnManager, had taken on too many responsibilities. For example, TurnManager was not only managing player turns but also handling user input, controlling game phases, and checking win/loss conditions, which violated the Single Responsibility Principle.

To improve modularity and maintainability, I decomposed these responsibilities into specialized components:

- TurnManager now focuses purely on sequencing turns.
- GamePhaseManager oversees the flow and transitions of game actions.
- GameInputHandler interprets and validates player input.

Additionally, the win/lose logic was extracted from TurnManager and reimplemented using the Strategy pattern. This resulted in a more flexible and testable structure with components like WinConditionStrategy and WinConditionChecker that support multiple types of victory conditions, including time-based and custom rules. More importantly, it sets a foundation for future growth by aligning the architecture with key object-oriented principles such as Open/Closed, Single Responsibility, and Separation of Concerns.

The god card system was also significantly improved. Sprint 2 relied on basic implementations, but Sprint 3 introduces a GodCardFactory to handle dynamic creation of god cards and a GodCardDeck to manage the available selection. This design provides better scalability for adding new cards and aligns with object creation and encapsulation principles. The StandardGodCard class provides a reusable base for implementing future god cards. It supports both enhanced abilities (like Artemis' movement ability) and modes where god powers are disabled altogether, falling back to the default move-and-build behavior. This allows the system to cleanly handle a variety of gameplay styles without requiring changes to the core logic.

To support alternative modes of play, such as tutorials or competitive timing, a new GameMode abstraction was introduced using the Strategy pattern. This decouples game logic from game mode configuration, allowing modes like TutorialGameMode or StandardGameMode to define their own setup, validation rules, and behavior without modifying core classes like Game or TurnManager.

2.1.2 Timer System Classes (Extension 1)

The timer system was introduced to enable competitive play. Rather than embedding timer logic into TurnManager, Sprint 3 introduces TimerManager as a standalone class that coordinates all timing functionality using the Mediator pattern. This ensures that timing logic remains modular and decoupled from turn logic. Each player's timing is encapsulated in a PlayerTimer instance. While this logic could have been managed through primitive variables within the Player class, such an approach would scatter timing responsibilities across the codebase. PlayerTimer encapsulates countdown behavior, pause/resume functionality, and expiration detection, allowing it to remain testable, reusable, and independent of player identity.

Responsibility for coordinating timing was explicitly moved out of TurnManager to maintain a clear separation of concerns. This allows different game modes to apply different timing strategies without requiring changes to the core turn-handling logic.

2.1.3 Triton God Card (Extension 2)

The Triton god card introduces advanced movement mechanics that demonstrate the extensibility of the new god card framework. While Sprint 2 included simpler god cards like Artemis and Demeter, Triton's ability, chained movement on perimeter tiles, requires a more complex, condition-based action sequence.

To support this, Sprint 3 introduces TritonMoveAction, a subclass of MoveAction, implementing chained movement using the Chain of Responsibility pattern. This adheres to the Open/Closed Principle: rather than modifying MoveAction and impacting all other god cards, Triton-specific logic is encapsulated in a dedicated class that integrates with the existing action system.

Had MoveAction been reused for this purpose, it would have required the introduction of god-specific logic, such as a new method or condition to check whether the destination tile is on the perimeter of the board—a rule unique to Triton. Embedding this logic directly into MoveAction would have broken the Open/Closed Principle by forcing a general-purpose class to accommodate special-case behavior. This not only risks introducing bugs in unrelated god cards but also increases coupling and reduces maintainability as new gods are added in the future.

The Triton class itself extends the god card framework with its own rules and logic. Although configuration-based approaches could theoretically handle Triton's behavior, its conditional chaining requires logic too specific to be cleanly expressed through flags or static rules. A dedicated subclass enables polymorphism, making Triton's unique behavior easy to manage without disrupting the overall system.

No responsibility reassignment was required for this extension; the god card system was already designed for inheritance-based extension. However, the improved factory and deck mechanisms (via GodCardFactory and GodCardDeck) make integration of new cards significantly cleaner and more maintainable.

2.1.3 Tutorial System (Extension 3)

The tutorial system is a new gameplay mode introduced in Sprint 3 to guide new players and support learning. Unlike standard gameplay, tutorials require structured scenarios, step-by-step instructions, error prevention, and context-aware feedback.

To support this, the GameMode abstraction separates different gameplay types cleanly. TutorialGameMode defines behavior tailored for teaching, while StandardGameMode maintains the original Santorini rules. This prevents the need to add tutorial-specific conditions into the core game classes, keeping the codebase clean and focused.

At the core of the tutorial system is the TutorialManager, which applies the Facade pattern to manage tutorial progression. It handles step transitions, validation checks, and coordinates with other parts of the system. Instead of packing all this logic into TutorialGameMode, the manager centralizes control, making the system easier to maintain and extend.

Tutorial steps are built using a TutorialStep hierarchy, following the Template Method pattern. Each step type—such as SelectWorkerStep or MoveWorkerStep—defines its own logic for validation and progression while sharing a common structure. This design makes it simple to add new steps or modify existing ones without rewriting the entire flow.

To keep the tutorial logic separate from how it's displayed, the system uses the Observer pattern. TutorialObserver and TutorialUIAdapter respond to tutorial events and update the interface when needed. This separation means the tutorial system doesn't depend on any specific UI implementation and can adapt to future changes in how the game is presented.

The Game class was also refactored to accept both a GameMode and a WinConditionStrategy. This makes it possible to switch between competitive play and tutorial mode without altering the core logic. Overall, the tutorial architecture promotes modularity and lays the groundwork for future learning-focused features.

2.1.4 UI Overhaul

The user interface was completely redesigned in Sprint 3 to make it more modular, maintainable, and adaptable to new features. In Sprint 2, all UI logic was grouped into a single file that handled everything from rendering screens to managing input and navigation. This made the interface difficult to scale and limited flexibility when new screens or interactions were needed.

To address this, a new screen management system was introduced. At the center of this system is a BaseScreen class that defines a consistent structure for how screens behave and interact. Specific screens—such as the main menu, setup, gameplay, tutorial selection, and game over—now each have their own class, responsible only for their individual functionality. These screens are coordinated by a ScreenManager, which handles switching between them and managing their lifecycle.

This design brings clear separation of responsibilities to the UI, making it easier to add new screens or update existing ones without risking unintended side effects. For example, introducing a new settings screen or tutorial welcome screen can now be done independently without touching unrelated parts of the interface.

To support this structure, a ResourceManager was also added. Instead of loading images, fonts, and other assets in an inconsistent or scattered manner, all resources are now managed through a centralized system. This improves performance by avoiding duplicate loads and makes it easier to update or swap out assets in the future.

2.2 CRC Cards for Extensions

The following CRC (Class-Responsibility-Collaborator) cards describe key classes introduced in Sprint 3. These classes represent the core components of each major extension (Timer system, Triton god card, and Tutorial system). Each card outlines a class's primary responsibilities and the collaborators it interacts with, helping clarify how responsibilities were distributed and how different components communicate.

This design approach helps reinforce modularity, testability, and adherence to object-oriented principles like the Single Responsibility Principle and Encapsulation. These CRCs also justify architectural choices such as separating logic into specific manager or action classes instead of overloading existing ones

.

2.2.1 Timer System (Extension 1)

2.2.1.1 TimerManager

TimerManager	
Coordinate switching of active timers between players **Manage timer lifecycle during turn transitions** **Provide aggregated timer info for UI and game logic** **Detect and notify timer expiration events**	**Player** **PlayerTimer** **TurnManager**

A central controller responsible for managing all player timers. It handles the coordination of active timers, communicates with the game's UI and logic, and ensures timers transition appropriately between turns.

- **Responsibilities**:
    - Coordinate switching of active timers between players
    - Manage timer lifecycle during turn transitions
    - Provide aggregated timer info for UI and game logic
    - Detect and notify timer expiration events
- **Collaborators**: Player, PlayerTimer, TurnManager

TimerManager was created to separate timing logic from TurnManager. Its responsibilities are strictly limited to timer coordination, ensuring other components remain focused on core gameplay. It serves as a central point of access for timing data, making it easier to update, extend, or disable timing in different game modes.

2.2.1.2 PlayerTimer

PlayerTimer	
Manage countdown timer for a player (start, pause, reset) **Track remaining time and expiration status**	**Player**

Manages the countdown timer for a single player. It tracks remaining time and expiration state independently of the player's game status.

- **Responsibilities**:
    - Start, pause, resume, and reset the countdown timer
    - Track remaining time and expiration status
- **Collaborators**: Player

PlayerTimer is tightly focused on timing and doesn't rely on broader game context. This independence makes it highly reusable and ensures it can be tested in isolation. It avoids burdening the Player class with unrelated timing logic.

2.2.2 Triton God Card (Extension 2)

2.2.2.1 Triton

Triton (GodCard subclass)	
Define Triton god card special ability description **Provide Triton's specific action sequence (TritonMoveAction followed by BuildAction)**	**Worker** **Board** **Tile** **Position** **ActionResult**

Defines Triton's special ability and controls the sequence of actions performed during the player's turn.

- Responsibilities:
    - Describe Triton's ability
    - Provide the action sequence (TritonMoveAction followed by BuildAction)
- Collaborators: TritonMoveAction, BuildAction

Triton's job is to declare what the god power does, not to implement movement logic itself. It connects the god card framework to Triton's unique ability in a clean and modular way.

2.2.2.2 TritonMoveAction

TritonMoveAction (MoveAction subclass)	
Execute a worker move action using Triton's special rules **Detect if the destination tile is on the board perimeter** **Chain additional optional move actions when on perimeter**	**Worker** **Board** **Tile** **Position** **ActionResult**

Handles Triton's movement ability, allowing chained moves when a worker ends on a perimeter tile. It determines if the chaining condition is met and manages the creation of follow-up move actions.

- Responsibilities:
    - Execute movement using Triton's special rule
    - Detect if the move ends on a perimeter tile
    - Trigger additional move actions if chaining conditions are met

- Collaborators: Worker, Board, Tile, Position, ActionResult

TritonMoveAction was introduced to keep Triton's logic separate from the base MoveAction, avoiding the need to insert special-case conditions that would affect other god cards. Perimeter checks and chaining logic are localized, making the behavior easier to reason about and maintain.

2.2.3 Tutorial Game Mode (Extension 3)

2.2.3.1 TutorialManager

TutorialManager	
Manage Tutorial step progression and state **Validate player actions based on current tutorial step** **Provide guidance (highlight tiles, and messages)** **Notify observers of tutorial events** **Support multiple types of tutorials with different step sequences**	**TutorialStep** **TutorialObserver** **Player** **Board** **Worker** **Tile**

Coordinates the entire tutorial system by managing step progression, validating player actions, and delivering contextual guidance. It also keeps observers (e.g., UI components) updated and supports the flexibility to run different tutorial sequences.

Responsibilities:

- Manage tutorial step progression and state
- Validate player actions based on the current tutorial step
- Provide guidance (highlight tiles and display messages)
- Notify observers of tutorial events
- Support multiple types of tutorials with different step sequences

Collaborators:

- TutorialStep
- TutorialObserver
- Player
- Board
- Worker
- Tile

TutorialManager was introduced to consolidate tutorial logic into a single, central component. By separating validation, state tracking, and observer notification from the game mode or UI, it simplifies the flow and makes the tutorial system more maintainable and extensible. It also ensures that different tutorials can be implemented or adjusted without needing to rewrite base gameplay logic.

2.2.3.2 TutorialGameMode

TutorialGameMode (GameMode subclass	
Initialize predefined board layouts and scenarios **Set up players, workers, and buildings for tutorial** **Delegate tutorial logic and validations to TutorialManager** **Manage flow and determine tutorial completion**	**TutorialManager** **Player** **Board** **Worker** **Tile** **Position** **Building**

Handles the setup and high-level control flow for tutorial mode. It prepares a controlled game environment and delegates tutorial-specific behavior to TutorialManager.

Responsibilities:

- Initialize predefined board layouts and scenarios
- Set up players, workers, and buildings for tutorial
- Delegate tutorial logic and validations to TutorialManager
- Manage flow and determine tutorial completion

Collaborators:

- TutorialManager
- Player
- Board
- Worker
- Tile
- Position
- Building

TutorialGameMode focuses on preparing the tutorial environment and connecting it to core game mechanics without mixing in the logic of how tutorials behave. By offloading progression and validation to TutorialManager, this class stays focused on scenario setup and lifecycle control. This structure helps isolate tutorial behavior from competitive modes and enables easier updates or testing of learning content.

2.3 Alternative Designs Considered

2.3.1 Timer System (Extension 1)

2.3.1.1 Embedded Timer in Existing Classes

An early idea was to build the timer logic directly into existing classes. Each Player would track fields like remaining_time and is_paused, while TurnManager would handle things like ticking down the clock and switching between player timers during turn transitions.

Why It Was Considered:

- It keeps the overall structure simpler by reducing the number of new classes.
- The flow between gameplay and timing becomes more direct, with fewer layers of delegation.
- It may also use slightly less memory since it avoids creating separate timer objects.

Why It Was Rejected:

- It breaks the Single Responsibility Principle (SRP). For example, Player would now have to manage both its gameplay state and time tracking, two very different concerns.
- This tight coupling makes the code harder to test and reason about. Verifying timer behavior would require setting up the entire game state.
- It doesn't scale well for future needs. For example, tutorial mode might need custom timer behavior, or animations might temporarily pause the clock; handling those cases would require messy conditional logic spread throughout the codebase.
- Any new timer-related feature would likely require editing core classes, which increases the chance of unintended bugs in unrelated systems.

This approach can work in small, throwaway projects or prototypes where the gameplay is simple, there's only one timer type, and the system isn't expected to grow much over time.

2.3.2 Triton God Card (Extension 2)

2.3.1.1 Decorator-Based Abilities

One idea was to use the Decorator approach for god card actions. In this design, god abilities like Triton's could be implemented as decorators that wrap around base actions, so TritonMoveAction would effectively be a MoveActionDecorator, extending movement behavior by adding chained moves.

Why It Was Considered:

- It supports flexible composition. You could combine multiple decorators to build complex behaviors without modifying existing logic.
- It encourages reuse—smaller god card abilities (like minor movement tweaks) could be layered without duplicating code.

- It fits nicely in games where abilities are modular and stackable, like in RPGs or card battlers.

Why It Was Rejected:

- Triton's ability isn't a small extension—it fundamentally replaces the logic of movement. It introduces conditional flow, perimeter detection, and chaining, which are hard to express cleanly using simple decorators.
- Using a decorator here would force convoluted logic inside the wrapper just to intercept and override behavior—defeating the point of clean layering.
- It also doesn't offer the reuse benefit in this case, since Triton's rules are too specific and unlikely to apply to other cards.

Why I Chose My Current Approach Instead:
Triton's behavior is better modeled as a standalone subclass (TritonMoveAction) that extends MoveAction directly. This keeps the chaining logic isolated, easy to understand, and fully decoupled from other god cards. It respects the Open/Closed Principle—new abilities can be added through subclassing rather than modifying existing code—and avoids bloating the base action or adding unnecessary abstraction layers.

The decorator model is well-suited to games where abilities are small, incremental, and meant to be combined.

2.3.3 Tutorial Game Mode (Extension 3)

2.3.1.1 Embedded Tutorial Logic in Game Classes

One option was to build the tutorial logic directly into the core game classes like Game, Board, or TurnManager. This would mean using flags and conditionals inside those classes to guide the tutorial, validate player actions, and track progress.

Why It Was Considered:

- It keeps tutorial logic physically close to the game mechanics it interacts with, which can make it easier to wire up specific rules quickly.
- It reduces the number of files and classes, which can simplify things in the very short term—especially for small projects or early prototypes.
- Execution flow might seem more straightforward without delegation to separate managers.

Why It Was Rejected:

- Embedding tutorial logic directly into gameplay classes mixes responsibilities and breaks separation of concerns. Core systems like Game or TurnManager would end up cluttered with tutorial-specific code.
- This makes the code harder to read, test, and extend—especially if changes to the game rules or tutorial are needed independently.

- It would also make adding more complex or varied tutorial scenarios difficult without introducing even more branching logic in core classes.

For very small-scale learning games or prototypes, where the tutorial is deeply tied to a fixed set of mechanics and long-term maintainability isn't a concern, embedding logic like this could be a faster short-term solution.

3. Relationships, Inheritance, and Cardinality

This section highlights the key structural relationships and inheritance decisions introduced through Sprint 3 extensions. The focus is on how these designs support extensibility, maintainability, and separation of concerns.

3.1 Inheritance

Tutorial Step Class
The TutorialStep abstract class is extended by multiple concrete steps (e.g., MoveStep, BuildStep, CompletionStep). Each subclass defines step-specific validation while reusing shared progression behavior.

- **Justification:** This inheritance avoids code duplication and allows each tutorial step to handle its own rules independently while remaining part of a structured sequence.

GameMode Class
GameMode is an abstract base class implemented by StandardGameMode and TutorialGameMode.

- **Justification:** This separation makes it easy to introduce new modes like tutorials or competitive variants without changing core game logic.

3.1 Key Relationships and Cardinalities

TutorialManager → TutorialStep

- Cardinality: 1 to 1..*
- Type: Composition
- Explanation: Each TutorialManager owns a sequence of TutorialStep instances. The steps are entirely dependent on the manager's lifecycle and do not exist independently.

Game → GameMode

- Cardinality: 1 to 1
- Type: Association

- Explanation: Each game instance uses a single GameMode during runtime, allowing the system to switch between tutorial and standard play while keeping game logic decoupled.

TutorialManager → TutorialObserver

- Cardinality: 1 to many
- Type: Aggregation
- Explanation: The TutorialManager sends updates to observer components (e.g., UI). These observers are not owned by the manager and can be reused or swapped independently.

TimerManager → PlayerTimer

- Cardinality: 1 to 0..*
- Type: Association
- Explanation: TimerManager manages access to multiple PlayerTimer instances but does not own their lifecycle. Timers can exist independently or be reused in different contexts.

TurnManager → TimerManager

- Cardinality: 1 to 1
- Type: Association
- Explanation: TurnManager interacts with TimerManager to control the active timer during turns. However, it does not create or manage the timer's lifecycle.

Player → PlayerTimer

- Cardinality: 1 to 1
- Type: Composition
- Explanation: Each Player owns its own PlayerTimer, which exists solely for that player and is destroyed alongside it. This ensures that timer behavior is isolated per player without external dependencies.

4. Design Patterns

4.1 Design Patterns Used

Strategy Pattern
Implemented in the GameMode hierarchy (StandardGameMode, TutorialGameMode) and the WinConditionStrategy hierarchy (StandardWinCondition, TimerWinCondition, CompositeWinCondition). This pattern was essential for supporting different gameplay configurations without modifying core logic.

For example, tutorial mode uses a different board setup, validation logic, and victory condition compared to standard mode. Without Strategy, these differences would have required conditional logic scattered throughout Game and TurnManager, violating the Open/Closed Principle.

Benefits: Enabled clean runtime switching between modes, maintained separation of educational and competitive logic, and provided extensibility for future variants (e.g., custom or timed modes).

Observer Pattern
Used in the tutorial system via the TutorialObserver interface and its concrete implementation TutorialUIAdapter. TutorialManager acts as the subject, notifying all observers about state changes such as step completion or hint updates. This was vital to avoid tightly coupling tutorial logic with UI rendering and to adhere to the Dependency Inversion Principle.

Benefits: Supported multiple independent UI components (messages, highlights, progress bars), allowed easy integration of new observers, and decoupled core logic from presentation.

Template Method Pattern
Realized through the TutorialStep abstract class and its specific subclasses (MoveWorkerStep, BuildStep, etc.). Each step shares a common lifecycle—initialization, validation, feedback, completion—while allowing step-specific customization. This structure prevented duplication and enforced consistency across steps.

Benefits: Enabled flexible step implementation with a unified interface, simplified creation of new tutorial steps, and centralized shared logic for maintainability.

Mediator Pattern
Used in TimerManager to handle communication between PlayerTimer instances and TurnManager. Without a central coordinator, managing turn transitions, timeouts, and pausing logic would require excessive cross-referencing and conditionals.

Benefits: Centralized and simplified timer coordination, reduced coupling between timers and gameplay logic, and isolated policies like pause conditions and timer resets.

Chain of Responsibility Pattern
Applied in TritonMoveAction to allow chained movement actions when the worker ends a move on the perimeter. The chain dynamically decides whether another move is valid and should be triggered. This allowed conditional action sequencing without bloating the base MoveAction logic.

Benefits: Cleanly handled optional, repeated movement, avoided special cases in shared logic, and laid a foundation for future gods with similar conditional abilities.

Facade Pattern
TutorialManager acts as a facade over the tutorial subsystem, encapsulating step progression, observer notifications, and UI guidance logic. This shielded the rest of the game from internal tutorial complexity.

Benefits: Offered a single access point for tutorial control, simplified integration with `TutorialGameMode`, and maintained encapsulation for better maintainability.

4.2 Alternative Patterns Considered (Not Used)

Factory Method Pattern
Not applied because all tutorial steps are statically defined and known at compile time. Using a factory would have added unnecessary complexity without runtime benefit. This can be applied in systems that require dynamic tutorial generation, runtime configuration, or user-defined sequences.

Command Pattern
Omitted since Santorini does not require undo or redo functionality. Actions are executed once and committed, and the current `Action` hierarchy suffices for encapsulating game behavior. This can be applied in games that allow action rollback.

Decorator Pattern
Rejected for Triton because its ability isn't a lightweight enhancement of movement—it's a fundamental shift in behavior. Wrapping standard `MoveAction` with a decorator wouldn't accommodate perimeter-check logic and conditional chaining effectively. This can be applied in systems where multiple abilities stack or augment a base action, like in card games.

5. Human Values Alignment: Helpful (Benevolence)

Why Helpfulness Is Important

Helpfulness fosters inclusion, learning, and accessibility—core needs in both education and gaming. In a strategic game like *Santorini*, the depth and nuance of its mechanics can intimidate new players, especially those unfamiliar with turn-based strategy. Without built-in learning support, newcomers are left to trial-and-error, leading to frustration and early disengagement.

By embedding helpfulness into the game's design, we reduce barriers to entry and empower players to grow their competence through guided, supportive interactions. It also ensures the game respects players with varying skill levels, creating a more welcoming and educational experience overall.

How Helpfulness Is Manifested in the Tutorial Extension

The **Tutorial System**, introduced in Sprint 3, directly embodies helpfulness through its structure, feedback mechanisms, and player support features:

- **Progressive Skill Building:** The tutorial introduces game concepts incrementally, starting from basic movement and building, up to more strategic elements like reaching level 3 or avoiding traps. Each tutorial step builds on the last, scaffolding knowledge in a structured and accessible way.
- **Real-Time Visual Feedback:** The system highlights valid tiles for movement and building based on the current step, offering players immediate guidance. This reduces guesswork and helps players internalize game rules visually.
- **Context-Sensitive Instructions:** Tutorial messages are dynamically tailored to the current game state. For instance, players are instructed to “Select the worker closest to the dome,” with visual cues reinforcing the instruction. This ensures players aren’t overwhelmed with abstract information but instead receive precise, situational guidance.
- **Safe Learning Environment:** Tutorial mode disables competitive win/loss conditions, creating a stress-free space to learn. Players can focus on understanding mechanics without fear of failure or pressure from an opponent.

How Markers and Players Experience This Value

- **For Players:** The tutorial creates an accessible entry point into the game. By scaffolding complexity and offering real-time help, it gives new/beginner players confidence and a clear path to mastering the rules. The system allows players to engage deeply with the game without feeling overwhelmed.
- **For Markers:** The helpfulness is clearly visible during evaluation. Markers can observe visual feedback (highlighted tiles, instructional text), track player progression through steps, and see how the tutorial adapts to player actions. These features demonstrate that the extension was purposefully built to reduce cognitive load and support learning.

Changes Made to Enable Helpfulness

To embed helpfulness into the game, several systems were introduced or extended:

- **TutorialManager** orchestrates progression through steps, manages observers, and triggers updates.
- **TutorialStep hierarchy** defines learning goals, validations, and feedback for each stage.
- **Tile highlighting and instructions text** systems were added to visually and verbally guide the player.