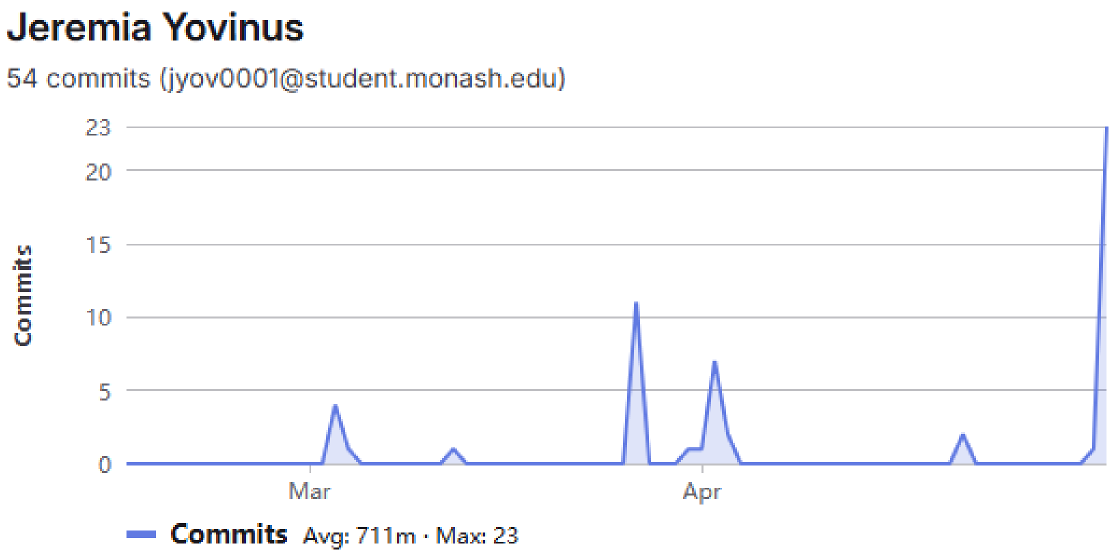
Jeremia Yovinus FIT3077 Sprint 3

Contributor Analytics



3 Implemented Extensions

Required Extension

The new god card I have chosen to introduce is Triton. Triton's power allows the player to move again immediately after moving into a perimeter space. As long as Triton continues moving into perimeter tiles, the player may keep moving indefinitely, gaining an extra move each time.

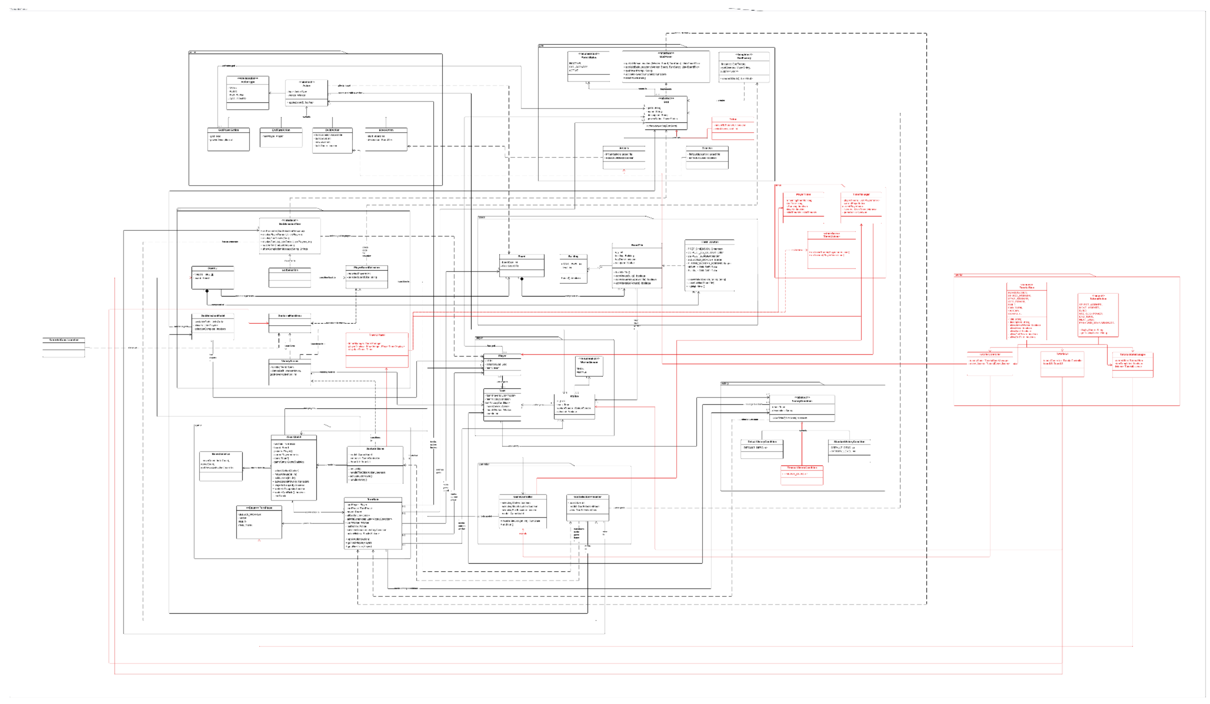
Pick-from-list Extension

I'm implementing a game timer system where each player has a designated time limit, such as 15 minutes, for their turns. Once a player completes their turn, their timer pauses, and the next player's timer begins counting down. If any player's timer reaches zero, that player loses the game. The interface will display each player's remaining time so all players can monitor their turns effectively

Self-Defined Extension

The extension I introduced is a Beginner-Friendly Mode (Tutorial Mode), which guides new players through the rules and mechanics of Santorini using tooltips, visual cues, and step-by-step prompts during gameplay. This directly supports the human value of Helpful under Benevolence, aiming to assist players who may be unfamiliar with the game. By providing clear and accessible guidance, the feature lowers the learning barrier and creates a more supportive, inclusive environment, ensuring all players, especially beginners, feel confident and welcomed while playing.

Updated Class Diagram



I updated the UML diagram to reflect three key extensions to the Santorini game. First, the Triton god was added via a new Triton class extending the existing God abstract class, integrating easily through the existing Factory and Strategy patterns. Second, a 15-minute countdown timer system was introduced using a new timer package with PlayerTimer, TimerManager, TimerListener and related UI components for real-time updates and timeout handling. Third, a beginner-friendly tutorial mode was implemented through a tutorial package, including TutorialStep, TutorialAction, TutorialController, TutorialUI, and TutorialStateManager to guide players step-by-step.

Class-Responsibility-Collaborator Cards

TutorialStep (Enum)

Responsibilities	Collaborators
Define tutorial step sequence and properties	TutorialStateManager
Store step title and description text	TutorialController
Specify which actions are allowed per step	TutorialUI
Provide step navigation (next/previous)	
Identify information-only vs interactive steps	
Calculate step numbers and total count	
Define step permissions (select/move/build/etc.)	

TutorialController

Responsibilities	Collaborators
Extend GameController with tutorial-specific restrictions	GameController (extends)
Validate actions according to current tutorial step	TutorialStateManager
Handle tile clicks with tutorial validation	TutorialStep
Manage tutorial progression and step completion	TutorialEventListener
Activate god powers with tutorial constraints	GameModel
Provide tutorial-specific status messages	TurnState
Control which actions are allowed per step	BoardTile
	Worker

TutorialStateManager

Responsibilities	Collaborators
Track current tutorial step and completion status	TutorialStep
Validate if actions are allowed in current step	TutorialAction
Handle step progression (next/previous)	TurnPhase
Complete steps based on performed actions	TutorialListener
Generate appropriate error messages	
Provide status messages for current step	
Notify listeners of step changes	

Design Rationale

Interfaces & Classes

In the updated design, several new interfaces and classes were added to support the three extensions. For the Triton god implementation, a new `Triton` class was created, extending the existing `God` hierarchy. This class was needed to encapsulate Triton's unique movement behaviour and could not be handled by existing gods like Artemis or Demeter without breaking the Single Responsibility Principle. The design uses the Strategy Pattern, allowing Triton to be interchangeable with other gods without modifying existing code.

For the timer system, the `PlayerTimer` class manages countdown logic for each player, and the `TimerManager` handles coordination between timers during turn changes. These responsibilities were kept separate from `GameController` and `SantoriniGame` to avoid coupling unrelated concerns. A `TimerListener` interface was introduced to notify UI components when timer related events occur, applying the Observer Pattern to keep the system modular and decoupled. These features required minimal changes to existing classes, only to initialise and update timer behaviour, so no major reassignment of responsibilities was needed from Sprint 2.

The tutorial system introduced the most new components. `TutorialStep` and `TutorialAction` enums defined each tutorial stage and valid actions. `TutorialController`, which extends `GameController`, enforced step specific behaviour. `TutorialStateManager` tracked progression and validated actions. These responsibilities are fundamentally different from normal gameplay and could not be added to existing game logic without breaking OOP principles. As with the timer system, tutorial components were designed as a separate module. No significant responsibilities from Sprint 2 were reassigned, demonstrating that the original design was already well structured for extensibility.

Alternative Designs

Several alternative designs were considered during development, but were either not implemented or intentionally avoided. One feasible alternative was extending support for 3 or 4 players. Our current `Team` and `Player` architecture already allows for multiple players per team, so this would have been a natural extension, but it was not included due to time and scope constraints. Another considered feature was adjustable board sizes such as 3x3 or 9x9. Our dynamic `Board` constructor already supports different dimensions, so this could be done with minimal changes, but was excluded to maintain focus on core gameplay. Additional god powers like Atlas, Hephaestus, and Pan were also considered. These could easily be added by extending the `God` class and implementing the `GodPower` interface, without needing to modify existing logic. We also considered allowing timer customisation through the UI. This would only require updating `PlayerBoardSelection` to pass user-selected values into `TimerManager`, but this feature was deprioritised.

On the other hand, some alternatives were deliberately not pursued due to complexity and architectural impact. Implementing an AI opponent would require building a separate decision-making system, as the current `Player` and `GameController` classes assume user input. Adding AI would involve introducing move evaluation logic, minimax algorithms, and god power handling for non-human players, which could not be cleanly integrated into existing classes without breaking the Single Responsibility Principle. Real-time multiplayer was also ruled out, as our current design is built for local, synchronous play. Supporting networked gameplay would require restructuring core classes to handle message passing, state sync, and latency, which would conflict with our current architecture and design goals. These options were excluded as they would demand major rewrites and compromise the clarity and maintainability of our codebase.

Design Patterns

Several design patterns were applied in the development of our extensions to improve modularity, maintainability, and extensibility. The **Strategy Pattern** was used in the god system, where each god, including the new Triton class, implements unique gameplay behaviour by extending the abstract God class. This pattern allows different gods to apply distinct movement or building rules without changing the core game logic. The game interacts with the God interface, enabling new gods to be added without modifying existing code. This supports runtime flexibility and follows the Open/Closed Principle.

The **Observer Pattern** was implemented in the timer system to handle UI updates and game responses to timer events. The TimerManager acts as the subject, managing a list of TimerListener observers, such as UI panels and the GameController. These listeners are notified through methods like onTimerStarted() and onTimeout() whenever a timer state changes. This setup keeps the timer logic in PlayerTimer separate from the components that respond to timer changes, allowing for clean, decoupled communication and easy UI integration without direct dependencies.

The **State Pattern** was applied in the tutorial system through the TutorialStep enum and TutorialStateManager. Each tutorial step acts as a distinct state that restricts player actions and defines valid transitions. For example, only worker selection is allowed in the SELECT_WORKER state, and the next state is predefined. The TutorialStateManager holds the current step and checks whether actions like move or build are valid based on the current state. This design ensures players follow a strict educational sequence and that gameplay logic remains controlled and consistent during tutorials.

The **Command Pattern** was extended in the tutorial through the TutorialAction enum, which defines allowable actions during each tutorial step. The TutorialController intercepts these actions and checks if they are valid for the current TutorialStep before allowing them to proceed. This enables fine control over gameplay during the tutorial, preventing invalid actions and guiding the user step by step. The use of these patterns across the extensions allowed us to maintain clean separation of concerns, avoid tight coupling, and ensure that new features integrated smoothly with the existing architecture.

Human value justification

The selected third extension, the beginner-friendly tutorial mode, aligns with the human value of “Helpful”, which falls under the category of Benevolence. This value focuses on supporting others and making experiences more accessible, which is especially important in game design to reduce learning barriers and promote inclusivity. In our game, this value is clearly expressed through a guided tutorial system that supports new players by breaking down gameplay into simple, interactive steps. Players and markers can see this value in action through visual cues such as blue highlights for selectable workers, green for valid moves, and yellow for build locations. Each tutorial step introduces one mechanic at a time, with only relevant actions enabled. Real-time feedback is provided to guide correct actions and prevent confusion. To support this functionality, we introduced a TutorialController that extends the main GameController and adds validation for tutorial-specific rules. The TutorialStateManager tracks the current tutorial step and ensures proper sequencing. The TutorialUI displays instructions, hints, and error messages, helping players understand both what to do and why. We also updated the BoardUI to support special highlighting for tutorial steps. These changes ensure that new players can safely experiment, understand core mechanics, and build confidence before entering the full game. The “Helpful” value is clearly manifested through this system, as it transforms a complex game into a supportive learning experience, making it more approachable for beginners and showing the marker how thoughtful design can promote positive, inclusive user experiences.

Reflection

First Extension: God Powers (Triton) – Very Easy
Adding Triton was straightforward due to the clean object-oriented foundation laid in Sprint 2. The abstract God class already defined key template methods like getValidMoveLocations() and activatePower(), making it easy to extend functionality. The use of the Strategy and Template Method patterns allowed us to plug in Triton's perimeter-based movement without changing existing code. The GodFactory class supported easy registration with minimal effort, and polymorphism ensured that GameModel and TurnState continued to function with no knowledge of specific god implementations. The clear separation of concerns and adherence to the Single Responsibility Principle meant no major refactoring was required. Nothing in Sprint 2 needed to be changed to make this extension easier.

Second Extension: Timer System – Moderate Difficulty
The timer system was moderately challenging because it needed real-time updates and interaction with both UI and gameplay logic. Sprint 2's familiarity with the Observer Pattern helped here, as it allowed us to introduce TimerListener, TimerManager, and PlayerTimer while keeping logic decoupled. However, timer functionality had to integrate closely with turn logic, requiring updates to GameController and GameModel, which introduced some temporal coupling. The absence of a central coordinating class made it harder to manage interactions cleanly. If we could revise Sprint 2, adding a Mediator pattern to coordinate between game logic and UI, or an Event Bus to decouple event handling, would have reduced this complexity and improved the separation of concerns.

Third Extension: Tutorial System – Quite Difficult
The tutorial system was the hardest extension due to its need to intercept and validate all user actions while enforcing a guided step-by-step flow. This was difficult because Sprint 2 lacked a Command Pattern, so actions like moving or building were not encapsulated and couldn't be easily controlled or blocked. We had to create a TutorialController that extended GameController and override multiple methods, which broke the Open/Closed Principle. Tutorial state and game state had to be tracked separately, and the lack of a State Pattern in Sprint 2 made this complex. UI feedback was also tightly coupled to game logic, making it difficult to insert tutorial-specific visuals. If we could redesign Sprint 2, we would introduce a Command Pattern for all game actions, a proper State Pattern for turn phases, and Strategy Pattern support for different game modes. Additionally, using Dependency Injection would have allowed us to substitute tutorial components more easily and reduce the coupling issues that made implementation difficult.

Executable Description

The Santorini game executable runs on Windows and requires Java SDK 24 to function correctly, as earlier versions may not be compatible. Users can run the game by simply double-clicking the .exe file. This executable is generated from the source code by first building a .jar file in IntelliJ and then converting it into a .exe using Launch4j.

To create the executable from source:

1. In IntelliJ, go to File > Project Structure > Artifacts, click the + button, and select JAR > From modules with dependencies.
2. Choose the main class, ensure libraries are set to extract, and click OK.
3. (Optional) Set your desired output directory for the .jar file.
4. Then go to Build > Build Artifacts, select your artifact, and click Build. The .jar will be created in the output folder, typically out/artifacts/PROJECT_NAME_jar.

To convert the .jar into a .exe:

1. Download and open Launch4j.
2. Under the Basic tab, set the output file to your desired .exe path and name.
3. Set the Jar field to point to your built .jar file.
4. Click the gear icon to save your configuration, then generate the .exe.

The resulting .exe can be launched directly, and the game will run on any Windows system with Java SDK 24 installed.