

Santorini - Sprint Two

The Team Dream

CL_Tuesday04pm_Team012

Aryan Chordia

Sunny Cho

Naailah Taqui Hasan

Contributor Analytics

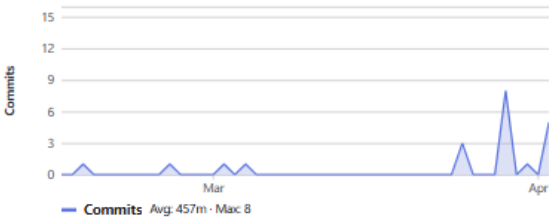
Sunny Cho

47 commits (scho0103@student.monash.edu)



Aryan Chordia

21 commits (acho0098@student.monash.edu)



Naailah Hasan

18 commits (nhas0021@student.monash.edu)



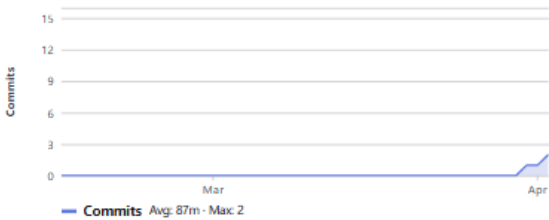
Naailah Hasan

10 commits (nhas0021@student.monash.edu)



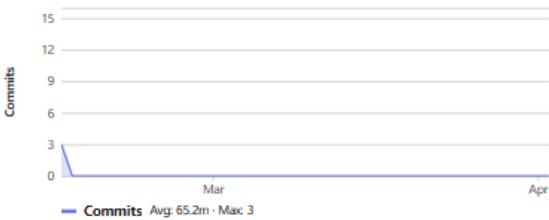
Naailah Hasan

4 commits (nhas0021@student.monash.edu)



Jordan.Nathanael

3 commits (Jordan.nathanael@monash.edu)



https://lucid.app/lucidchart/f2e285c3-2af3-48ca-a999-b54602bbefc8/edit?viewport_loc=-1835%2C-864%2C4016%2C1749%2CsXTbmbfqx~Ie&invitationId=inv_4ad3aa11-9edd-4daa-9d1e-1a8279e983e6

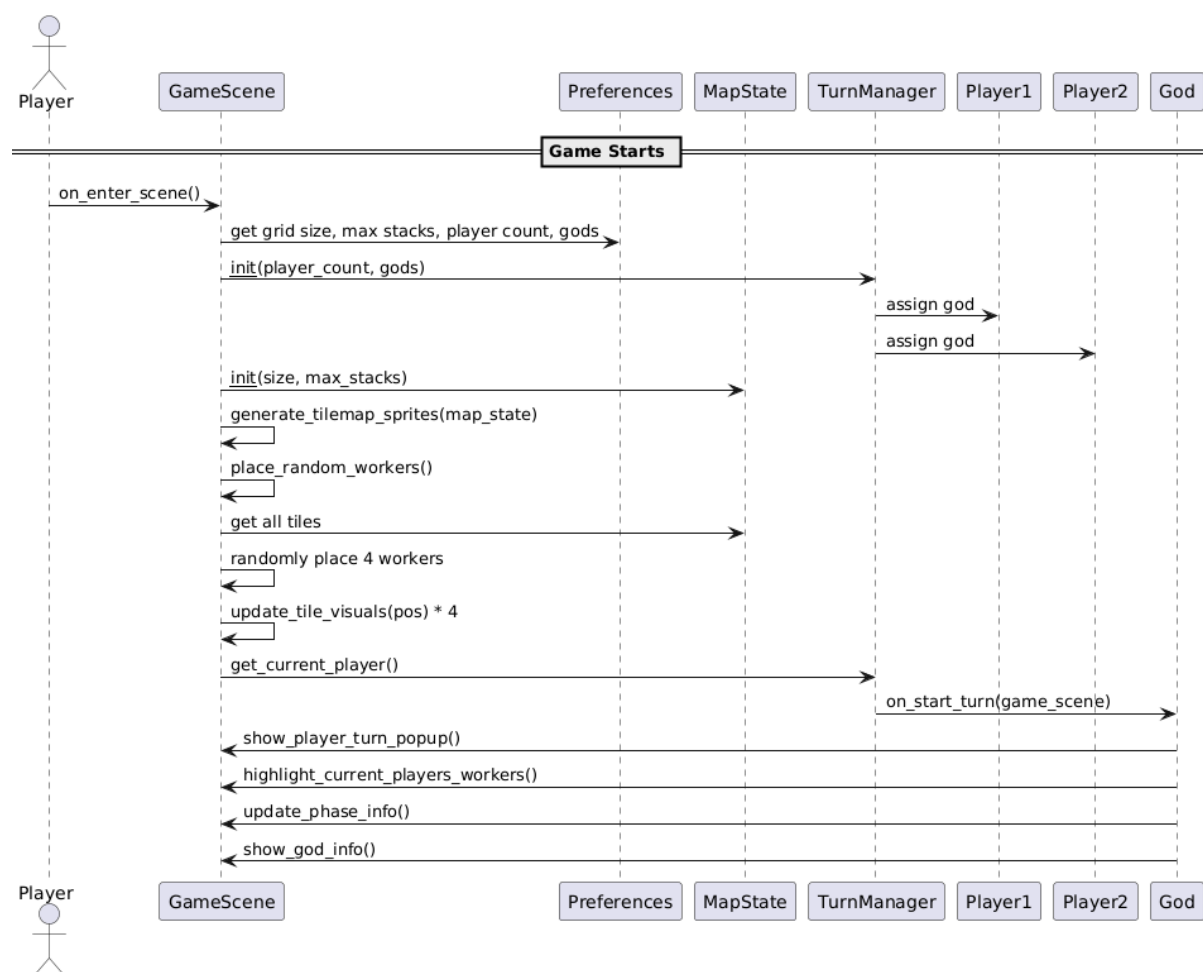
UML Sequence Diagram

These diagrams use standard UML notation to illustrate the interactions that occur when the scenario occurs. The vertical dashed lines represent the lifelines of actors and system components, whilst horizontal arrows denote method calls exchanged between them over time.

Each of these diagrams represent a key gameplay feature or interaction within the Santorini system, such as setting up the board, moving a worker, building, handling turn transitions, or checking for a win condition. Each diagram traces how user input (e.g. tile clicks) is handled across UI (TileSprite, GameScene), logic (God, MapState), and state coordination (TurnManager), ensuring responsibilities are clearly divided. This helps demonstrate both the structure of our system and how user actions are translated into visual and logical changes during play.

Feature 1

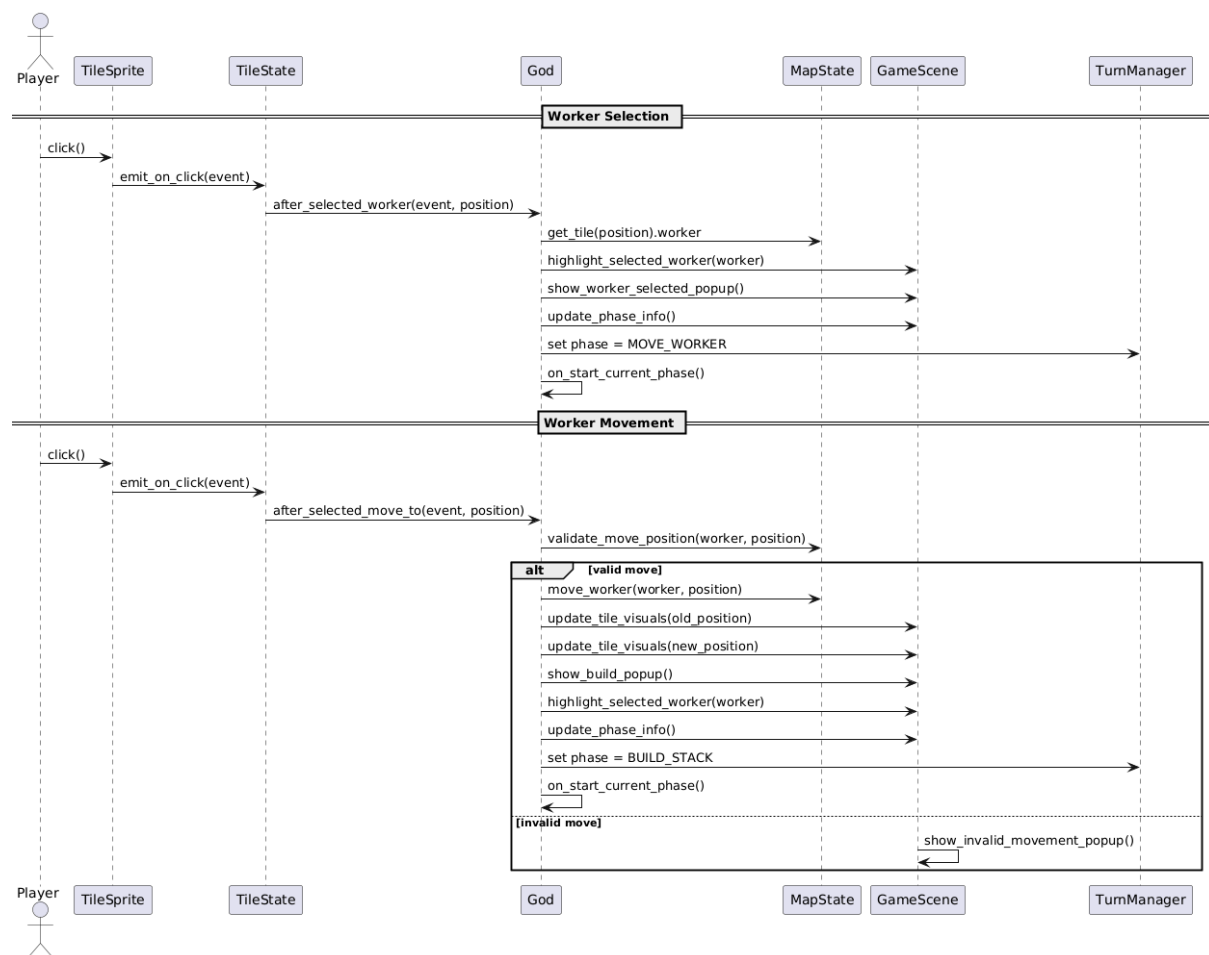
This diagram shows the setup process for the initial 5x5 board. It illustrates how player preferences are loaded, how each player is randomly assigned a god card, and how workers are randomly placed onto valid positions on the board. The sequence also includes visual



updates to ensure that the game board reflects the logical setup, providing players with immediate feedback through the user interface. This setup establishes the initial conditions for gameplay while maintaining a clear separation between logic and presentation.

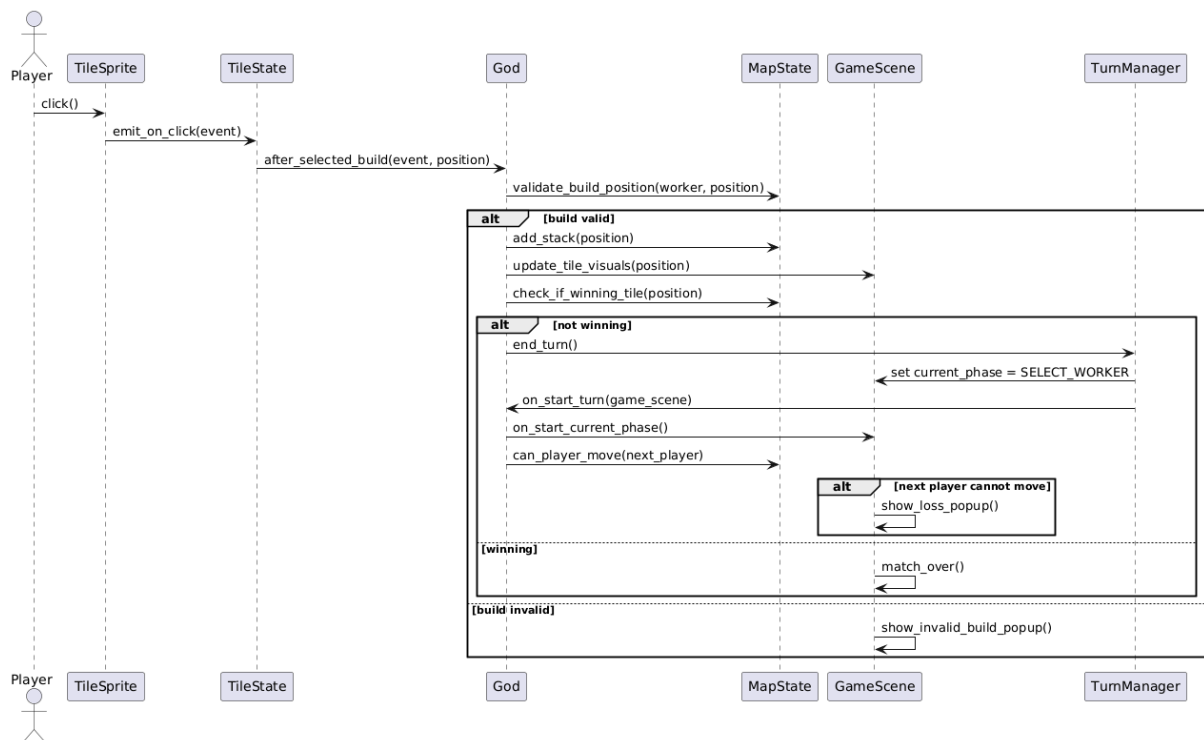
Feature 2

This diagram outlines how a player selects a worker and moves it to a valid tile. The interaction begins with a tile click, which triggers a validation check to ensure the selected worker belongs to the current player. Upon validation, the move is processed and the visual elements are updated to reflect the new state.



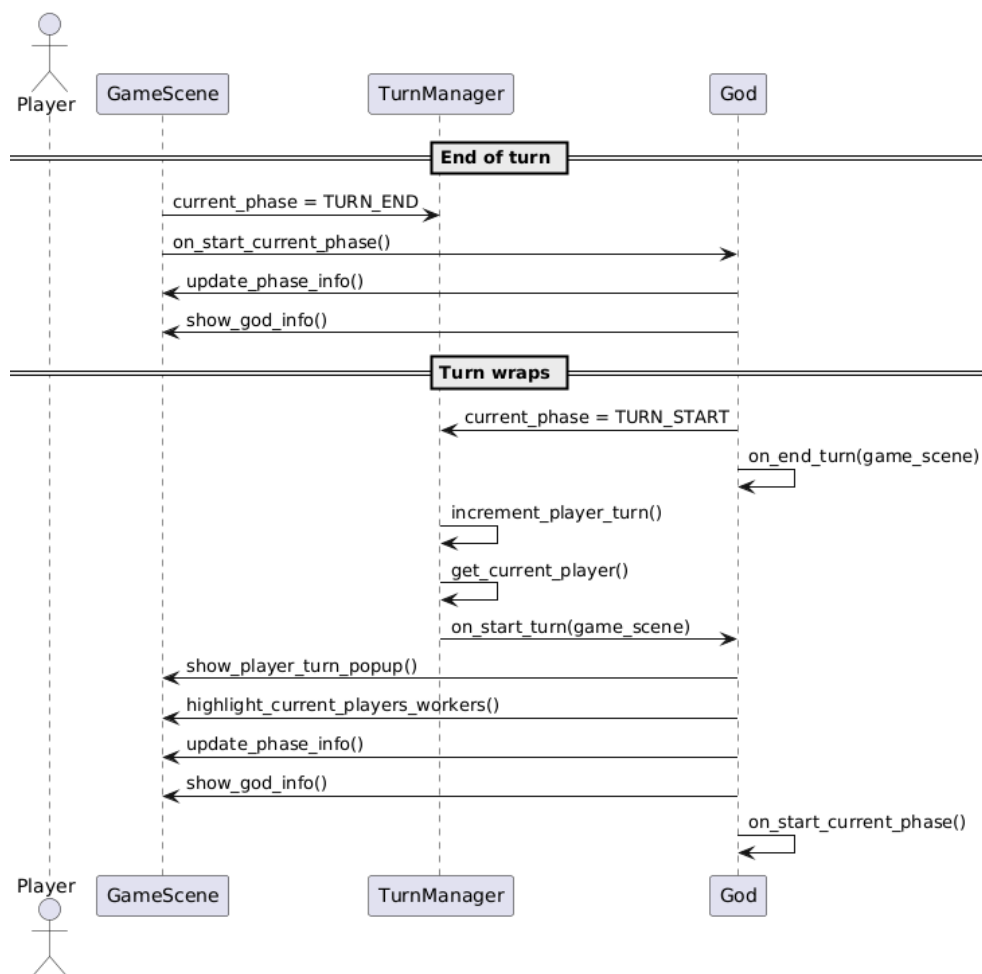
Feature 3

The following sequence diagram represents the building phase that follows a valid worker movement. When the player clicks on an adjacent tile to build, the system checks if the action is valid, ensuring the tile isn't occupied or domed. If the move results in a win condition, this is detected, otherwise the game progresses. If there is no valid build possible, the system triggers a loss condition.



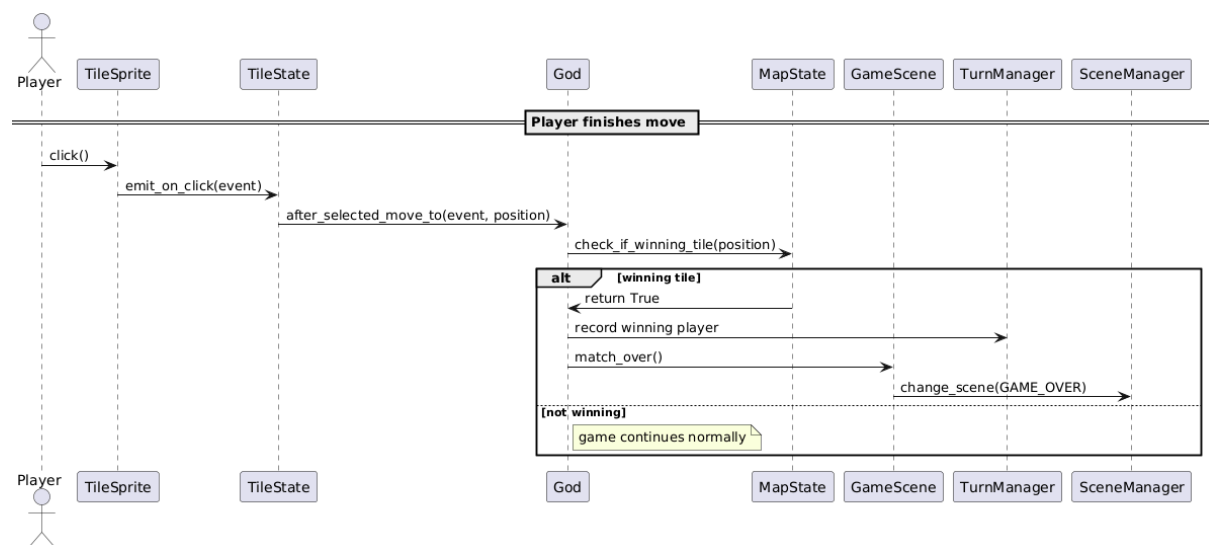
Feature 4

This diagram shows the transition from one player's turn to the next. Once a player finishes their build phase, the system resets any temporary turn-specific state, updates the active player, and prepares the next turn. The new player's god ability is activated, and interface elements are updated accordingly. The diagram shows how game flow is managed without needing to restart the board or scene, demonstrating how turn logic is modular and maintains a smooth player experience.



Feature 5

This diagram captures the final stage of a turn where a win condition may be triggered. After a move or build, the system checks if the action results in a win—such as moving onto a third-level tower. If a win is detected, the game state is updated and a transition is made to the game over scene. If not, the game continues normally. This sequence highlights how win detection is seamlessly integrated into the existing flow, ensuring the player receives immediate feedback and the game responds appropriately.



Design Rationale

Architecture Overview

In order to facilitate a robust and modular design, the codebase can be structured into two main categories; self produced modules and scripting behaviour. Essentially, a thin and lightweight game framework was produced to facilitate development. The overall strategy used was to rely on events (mouse, button, keyboard, etc) as a way to receive input from the user which was deemed acceptable due to the game being turn-based.

Three Key Classes

Scene

The Scene class (can also be called a “level”) is a base class which serves to encapsulate the data, sprites and logic of the different stages or levels of the game (modeled after a structure common in most game engines; [Unity](#), [Unreal](#), [Godot](#)). Derivatives include Main Menu, Game Scene and Settings. This avoids duplication and provides a consistent interface for all scenes, such as `enable_scene()` and `disable_scene()` ensuring that there is consistent usage. Each “scene” or “level” represents its own widgets along with their relevant event handlers; keeping this logic in its own class helps isolate it cleanly from other logic. This means that upon “entering” a scene, everything would be set up and it would work as sort of an isolated program.

It utilizes SceneManager to switch between different scenes in the game. This pattern would be impractical if scenes were merely methods, as method state does not persist cleanly across such UI transitions. Using methods instead of a class can make transitions and lifecycle management harder and lead to tight coupling and poor separation of concerns.

GameScene

A specific variant is the GameScene class is central to the game as it is responsible for managing the game’s user interface, player interaction and the visual representation of gameplay during a match. While it appears that it holds a high amount of responsibility, it also exists in other implementations as well, just not as an object but as the global namespace along with even more items that may not be related to it, such as main menus and the like.

It is composed of 3 main components:

- Map Data (MapState)
- Turn Data (TurnManager)
- Sprite and GUI

In regards to the match flow, GameScene handles the board setup, UI state updates, player information panels and visual feedback through popup windows.

Originally, GameScene contained both UI and gameplay logic however this led to a God Class with too much responsibility. Therefore in the current design, the class has been refactored to isolate responsibility; rule validation, phase progression and god powers are now within their dedicated components.

GameScene is implemented as a class as it needs to maintain persistent state across the lifecycle of a game. These states include tracking the worker, storing the sprite tile map and dynamically changing labels. Methods alone cannot preserve the state between interaction or manage UI components that depend on context.

MapState

MapState was created to encapsulate the logical representation of the game board. Its primary role is to manage and validate interactions between the grid of TileState objects, thus enabling gameplay logic such as movement, building, and win detection to be separate from the UI. Therefore, this abstraction closely follows the Single Responsibility Principle by ensuring that MapState is only responsible for the board state.

Since MapState manages several internal states that need to persist throughout the game, converting it into a class enables better encapsulation of logic, provides a consistent interface for other classes to interact with, and ensures reusability and flexibility. If implemented as standalone functions instead, it would not be possible to maintain state between calls, and the design would violate the Single Responsibility Principle (SRP).

God

In order to represent the chosen character, its name, description, and abilities, a God class was developed. To keep this open to extension, each variant or subclass of a God can override its methods, allowing it to exhibit different behaviour to a basic character which has no abilities. It also means that the basic behaviour does not need to be repeated across every variant, but instead only when it differs.

Additionally, God instances control the game as the turn manager will invert control over to the instance during key points throughout a turn (at the start, changing phases, after an action), allowing the God type to determine the strategy to be used. This allows for easy extensibility due to the nature of having the God instance / type being separate yet still being able to affect the overall state of the program, without having to understand what is going on beneath it.

Three Key Relationships

Composition between Player and Worker

The composition relationship between the Player class and the Worker class is evident in the way each player directly owns and manages two worker instances. These workers are not

created or maintained independently, they are instantiated as part of the Player's lifecycle and are intrinsically tied to it. If a Player object is removed from the game, the associated Worker instances are also discarded, meaning the workers cannot exist without the player that owns them. This tight coupling, where the creation and destruction of the Worker objects are entirely controlled by the Player, is the defining characteristic of a composition. It differs from aggregation in a way that associated objects can have independent lifetimes and may exist even if the parent object is deleted. In this case, since a worker has no context or purpose without its player, composition accurately models their relationship.

Association between Scene and SceneManager

SceneManager keeps a track of the currently active scene using the `current_scene`: `Optional[Scene]` attribute, forming an association. SceneManager does not create or destroy the scenes; it simply stores references and coordinates transitions by calling their `enable_scene` or `on_enter_scene` methods. These relationships are associations rather than aggregations or compositions because the Scene instances are created and managed externally in `App.py`, and they can exist independently of the SceneManager. If SceneManager is destroyed the Scene objects can still exist. This design promotes separation of concerns and allows flexible scene management without tightly coupling the manager to the scene lifecycles.

Association between Preferences and God

The association between Preferences and God is a simple unidirectional association where Preferences maintains a list of God objects through its class-level attribute `selectable_gods`: `List[God]`. Although the Gods are instantiated and stored during configuration, Preferences does not own their life cycle. If it is reset or deleted, the God objects themselves remain unaffected. This association is not an aggregation or composition because God instances are not dependent on Preferences for their existence and can be created or managed independently, ensuring loose coupling and greater flexibility for expanding the game's functionality.

Inheritance

Inheritance has been used in the design to promote code reusability and polymorphic behaviour. A key example of this is the God superclass and its subclasses such as Artemis and Demeter. The God class encapsulates shared attributes and methods common to all gods, such as the name, description, and hooks like `on_worker_moved` and `on_build_second`. By using inheritance, each specific god can override these methods to implement unique powers without duplicating code. This approach makes it easy to add new gods in the future by simply extending the God class and overriding the necessary behavior, which enhances maintainability and scalability.

Another key use of inheritance in the design is the Scene class, which acts as a base class for various game screens such as `MainMenu`, `GodAssignment`, and `GameScene`. These

subclasses inherit common functionality from Scene, including access to the root window, layout configuration, and lifecycle methods like enabling and disabling scenes. This promotes consistency and reduces code duplication across all scenes, while still allowing each subclass to define its own specific content and behavior. For instance, MainMenu focuses on navigation, while GameScene manages gameplay elements like the board and turn updates. Inheritance here ensures all scenes follow a unified structure, making transitions seamless and simplifying integration with the SceneManager. As a result, this improves modularity and maintainability.

Cardinalities

Player and Worker ($1 \rightarrow 2$)

Each player owns exactly two workers and each worker is associated with exactly one player. This cardinality was chosen to reflect the core mechanics of the Santorini board game, where every player is required to control two distinct workers on the board. The lower and upper bounds are both fixed at 2, indicating that this is not a flexible range but a strict rule of the system. Choosing a different cardinality, such as $0..2$ or $1..*$, would have misrepresented the game's structure by allowing for too few or too many workers per player, which could break the gameplay logic or introduce invalid states.

TurnManager and Player ($1 \rightarrow 2..*$)

The cardinality between the TurnManager and Player classes is set to $1 \rightarrow 2..*$ to accurately reflect the requirements and future scalability of the game. In its current form, the game is designed to be played by a minimum of two players, meaning the TurnManager must always manage at least two Player instances. This lower bound of 2 ensures that a valid game cannot start unless there are at least two participants, aligning with the rules of the Santorini board game. The upper bound is left open-ended (*) to allow for potential future extensions of the game that may support more than two players, such as multiplayer variants or team-based modes. On the other side, each Player is associated with exactly one TurnManager, indicating that all turn-related logic—such as tracking the current player, sequencing turns, and determining win/loss conditions—is centrally managed.

Design Patterns

Framework

A scene managing system was developed to aid in encapsulating both logic and data within a logical scene much like game engines, which allowed for a large groups of widgets (components in Tcl/Tk) to be enabled and disabled at the same time, ensuring that all relevant widgets are modified correctly. This system also allowed for the different scenes to be able to be referenced via a statically typed system rather than a loosely typed string lookup.

This also included generic math/vector struct-like along with their associated functions and operators which were often used to denote position. This led to the creation of an integer vector.

In order to manage the complexity of the UI layout and transition, we encapsulated each scene into dedicated classes coordinated by shared scene management logic. While this is not strictly following the Facade pattern, it allows for independent development and reuse of components. This reduces merge conflicts and improves maintainability by enforcing consistent interfaces and data structures (e.g. clearly defining vectors as `Vector2I` rather than ambiguous tuples or lists).

Managers/Singletons

The use of the above system however limited the data that could be passed through scenes and as such **singletons** were used to facilitate cross scene communication and state. By centralising such data, it also ensured that only one copy of data must be managed (no desynced state).

Examples of this include:

- **Styles:** a set of variables from which all generic/reused styling data should reference from, which allows for easy manipulation and change that is reflected in all aspects of the design rather than requiring manual modification of each widget.
- **Settings:** similar to above, settings may be required by more than one scene.
- **Game:** The game serves as the central point of truth for the current match state. It encapsulates the board grid, players, turn logic, and rule enforcement, enabling consistent and validated manipulation of game entities. By adopting a singleton design, it ensures persistent access across different scenes without duplicating the state. This simplifies gameplay logic and UI rendering by keeping all core data in a single location.

Callback & Events/Observer

The primary method of communication was via the usage of callback driven events due to how Tkinter and Tcl also use this method. Input events will trigger an event which causes a

callback to be run, reflecting the Observer pattern where UI components notify the system of an action.

Apart from simple fields such as buttons used within the UI, the game also features clickable sprites.

Such sprites are composed of a canvas which holds various shapes, images and text. This canvas then has callbacks binded to it along with identifying information (such as the tile position), creating interactable and customizable sprites.

Executable Generation

An executable of the application has been provided, which can be run on Windows, macOS, and Linux platforms. See the README.md for the most up to date information.

Setup

To build the executable from the source code, ensure you have Python and PyInstaller installed. In order to install PyInstaller, use the command:

```
pip install pyinstaller
```

After installation, Open a terminal (or command prompt) and navigate to the directory containing your Python script. Ensure current directory is ``./code/`` (from top level):

```
./project
|-- code ← (you should be here)
|   | - App.py
|
|-- readme.md
```

Package

Below are two ways of building the project once the setup is complete. Using the build script is recommended.

Build Script (Windows)

A build script is provided in the repository (*build.bat*), running it (by double clicking on it) will package the executable and open the folder which contains the resultant executable.

Manual Build (Alternative)

Create the executable by running:



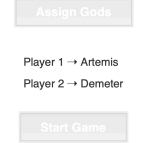



```
pyinstaller .\App.py --onefile --name Santorini --contents-directory
Resources --hidden-import=tkinter --add-data ./Assets:Assets
```

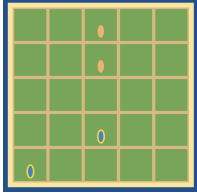

in the terminal. Here, App.py is the main Python file used to run Santorini. The generated executable will be located in the *dist* folder within the project directory. In order to run the executable, simply double click on the .exe file.

Video File Format:

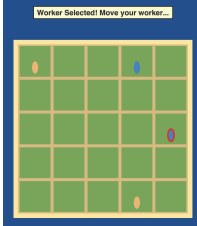


The video file, which includes a demonstration of the game and instructions for building the executable, is in MP4 (MPEG-4 Part 14) format. This format is widely supported across platforms and offers efficient compression, ensuring high-quality playback while keeping file size manageable.

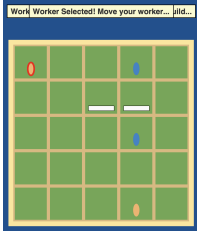
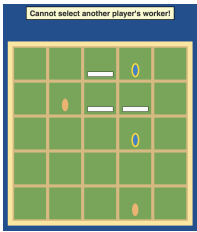


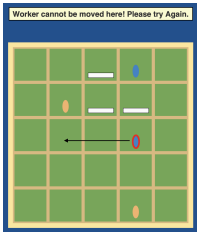


Testing Documentation

Feature: The initial game and 5x5 board setup					
ID	Description	Input	Expected Result	Actual Result	Pass/Fail
1	Each player must be assigned a unique god.	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods”.	Each player is assigned a unique god, and their assignments are displayed.	God Assignment 	Pass 
2	Each player must be able to re-assign a unique god.	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Assign Gods”	The uniquely assigned god may change to a different god, however they will still remain unique.	When “Assign Gods” is clicked multiple times, the gods remain unique. The assigned gods may change depending on randomness. God Assignment  God Assignment 	Pass 
3	Starting the game sets up a 5x5 board with 2 workers for each player.	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game”	A 5x5 board is visible with 2 workers for each player placed on tiles. The board contains no building stacks or domes.	Confirmed: board was initialised correctly with 2 workers per player.	Pass 

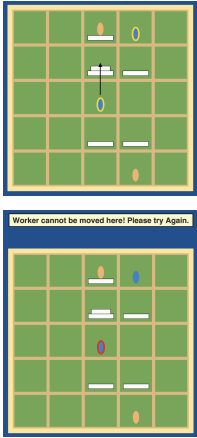
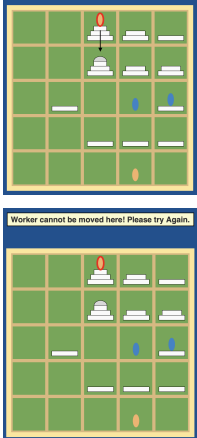
					
4	Confirm that the worker locations are randomized for each new game.	(Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game”) - Repeat multiple times and observe worker location.	The board is set up in a 5x5 grid and each time a new game is started, the 2 workers for each player are placed in random unoccupied tiles compared to previous games.	Confirmed: Each game started with workers in different positions. They were assigned to unique and valid tiles.	Pass 

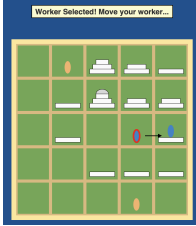
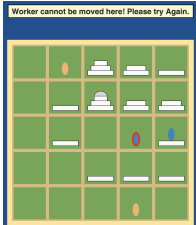
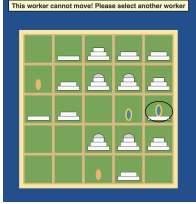
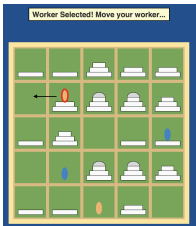
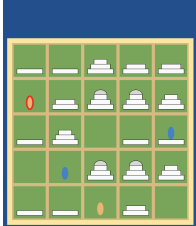
Feature: Worker selection and movement

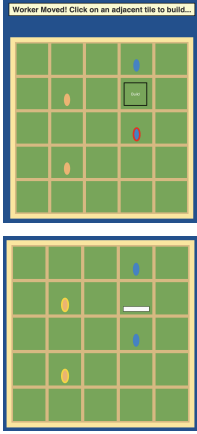

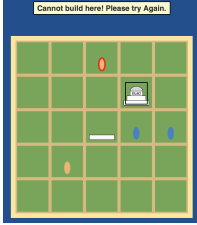


ID	Description	Input	Expected Result	Actual Result	Pass/Fail
5	Select a Player 1 worker on Player 1's turn, the selected worker should be highlighted.	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Click on a highlighted worker.	When player 1's worker is selected, it should highlight a colour that distinctly marks it out.	Confirmed: When a worker is selected, it is highlighted red and made obvious to the players. 	Pass 
6	Select a Player 2 worker on Player 2's turn, the selected worker should be highlighted.	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click	When player 2's worker is selected, it should highlight a colour that distinctly marks it out.	Confirmed: When a worker is selected, it is highlighted red and made obvious to the players.	Pass 

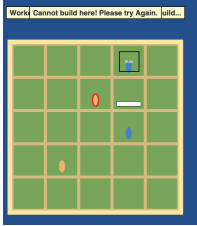
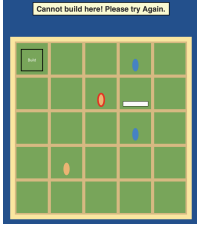
		<p>“Start Game” → Play as player 1 → Click on a highlighted worker.</p>			
7	Current player should be prevented from selecting the other player's workers.	<p>Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Click on a worker that's not highlighted.</p>	If player 1 is playing, a popup should display an appropriate message if player 1 tries to click on player 2's worker and the game should stay in SELECT WORKER phase.	<p>Confirmed: Popup is displayed and no worker is selected. Game still in SELECT WORKER phase.</p> 	Pass 
8	Try selecting a tile with no worker → should not transition to move phase	<p>Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Click on an empty tile.</p>	When an empty tile is clicked on during the worker select phase, nothing happens and the game remains in the same phase.	<p>Confirmed: Game remains in the same phase and the player is still expected to select a valid tile with a worker.</p>	Pass 
9	Attempt to move to a non-adjacent tile → should show invalid movement popup	<p>Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Click on a highlighted worker → Attempt to move the worker to a tile that is more than one tile away.</p>	When a non-adjacent tile is clicked on after selecting worker then a popup informs the player that the move is invalid.	<p>Confirmed: Player is not allowed to make a move and is informed that the move is invalid.</p> 	Pass 
10	Players should not be able to move their	<p>Open executable → Click “Play” → Click “Attain</p>	When an adjacent tile next to a non-selected	<p>Confirmed: Popup is shown and the game</p>	Pass 

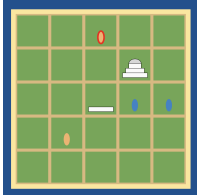




	worker which has not been selected.	God Powers" → Click "Assign Gods" → Click "Start Game" → Click on a highlighted worker → Click on adjacent tile of a worker NOT selected.	worker is clicked (assuming that tile is not adjacent to the selected worker too), then a popup displays that the move is invalid.	remains in MOVE WORKER phase. 	
11	Move to an adjacent empty tile of same height → allowed	Open executable → Click "Play" → Click "Attain God Powers" → Click "Assign Gods" → Click "Start Game" → Click on a highlighted worker → Attempt to move to an adjacent tile	When a worker moves to a tile of equal height, the move should be allowed and the worker should be displayed on the new tile.	Confirmed: Worker is moved to the adjacent tile of the same height. 	Pass ✓
12	Move to a tile one level higher → allowed	Open executable → Click "Play" → Click "Attain God Powers" → Click "Assign Gods" → Click "Start Game" → Play one full turn for each player → On player 1's turn click on a highlighted worker → Attempt to move to an adjacent tile that is exactly one level higher.	The worker should move to the desired level and the game should move to the next phase.	Confirmed: Worker is placed one level higher on the desired tile and the game continues. 	Pass ✓





13	Move to a tile two levels higher → blocked	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Play one full turn for each player → On player 1’s turn click on a highlighted worker → Attempt to move to an adjacent tile that is exactly two level higher.	The game should display a pop-up saying that the move is invalid and the game remains in the same phase.	Confirmed: Popup is displayed and the player is expected to choose another tile to move to. 	Pass ✓
14	Move to a tile with a dome → blocked	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Play until at least one tile has a dome → On Player 1’s turn, click on a highlighted worker → Attempt to move the worker to an adjacent tile with a dome.	The game should display a pop-up saying that the move is invalid and the game remains in the same phase	Confirmed: Popup is displayed and the player is expected to choose another tile to move to. 	Pass ✓
15	Move to an occupied tile → blocked	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Attempt to move worker onto an occupied tile.	The game should display a pop-up saying that the move is invalid and the game remains in the same phase	Confirmed: Popup is displayed and the player is expected to choose another tile to move to.	Pass ✓






				 	
16	Select a worker that cannot move → show warning and block	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Attempt to select a worker that has no valid adjacent tiles to move to.	When a player attempts to select a blocked worker, a warning is displayed and player is prompted to select a different worker	Confirmed: Popup is displayed and the game continues to be in the same phase, waiting for the player to select a valid worker. 	Pass ✓
17	Move down any number of levels → allowed	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Play turns until at least one worker is on a raised tile → On a player’s turn, select their worker → Attempt to move the worker to an adjacent tile that is lower in height by one or more levels.	When a player attempts to move a worker down to a tile lower in height by one or more levels, the worker is moved to that tile and the game moves on to the next phase.	Confirmed: The worker is moved and the game continues.  	Pass ✓
Feature: Building a stack.					






ID	Description	Input	Expected Result	Actual Result	Pass/Fail
18	After valid movement, attempt to build on adjacent empty tile → stack should increment.	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → On Player 1’s turn, select a worker and move it to a valid adjacent tile → After the movement, click on an adjacent empty tile to build.	When a player attempts to build on an adjacent empty tile, it stacks up the first building layer and the game continues.	Confirmed: Stack is built and the game continues. 	Pass 
19	Attempt to build on a dome → blocked	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Play until a tile contains a dome → On a player’s turn, select a worker that is adjacent to the domed tile → Attempt to build on the domed tile.	When a player attempts to build on a tile which has 3 layers and a dome, a popup with an appropriate message is displayed and the game remains in the same phase.	Confirmed: Popup is displayed and the game continues in the same phase, expecting the player to make a valid build choice. 	Pass 
20	Attempt to build on an occupied tile → blocked	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Play until a tile contains a stack → Move a worker next to a tile that has a worker standing on it → Try to build on the same	When a player attempts to build on an occupied tile, a popup with an appropriate message is displayed and the game remains in the same phase.	Confirmed: Popup is displayed and the game continues in the same phase, expecting the player to make a valid build choice.	Pass 

		tile where that worker is located.			
21	Attempt to build on non-adjacent tile → blocked	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → On Player 1’s turn, select a worker and move it to a valid adjacent tile → After the movement, click on a non neighbouring tile to build.	When a player attempts to build on a non-adjacent tile, a popup with an appropriate message is displayed and the game remains in the same phase.	Confirmed: Popup is displayed and the game continues in the same phase, expecting the player to make a valid build choice. 	Pass ✓
22	Try to build when no valid build locations → trigger loss condition	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → On Player 1’s turn, select a worker and move it to a valid adjacent tile → After the movement, no build locations available → show lost game screen	When the player moves to a tile in such a location that there are no valid build moves available, the player loses (shown through ui), the other player wins the game and screen switches to the Game Over screen.	Confirmed: Player loses the game and the other player wins. All the required screens are shown.	Pass ✓
23	Stack up to max level (3) and then build dome (level 4)	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → On Player 1’s	When a player attempts to build on a stack containing 3 levels, a dome is added to the top and the game continues.	Confirmed: A dome is built and the game continues to the next phase.	Pass ✓

		turn, select a worker and build on the same tile across multiple turns until it reaches level 3 → Once the tile reaches level 3, build on it again to place a dome.			
Feature: Change of turn.					
ID	Description	Input	Expected Result	Actual Result	Pass/Fail
24	Complete a valid move and build → turn changes to other player	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Complete all steps in player 1’s turn.	Once player 1 has completed all steps involved in their turn the game should return to the select worker phase for the next player.	Confirmed: The game moves on to the select worker phase for the next player.	Pass 
25	Confirm UI reflects correct current player after turn switch	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Complete all steps in player 1’s turn.	Once player 1 has completed all steps involved in their turn the UI should indicate that it is Player 2’s turn and also change the God description.	Confirm: UI reflects the turn switch and god description changes.	Pass 
26	Current player’s workers should be highlighted on the Select Worker Phase	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game”	Player 1’s workers (both of them) must be highlighted and be clearly visible with a gold border.	Confirmed: The workers are visibly highlighted on the screen	Pass 
27	Player has no valid worker to move at the start of turn → show popup and trigger lose condition	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” →	When none of the player’s workers can move, the player loses the game which is reflected through UI and the other	Confirmed: The game ends and the winner of the game is displayed.	Pass 

		Play until both workers of any player are trapped.	player wins.		
Feature: Win Detection					
ID	Description	Input	Expected Result	Actual Result	Pass/Fail
28	Move worker from level 2 to level 3 → triggers win	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Play until worker of any player moves from level 2 to 3	The game should end immediately and a screen indicating the winning player must be displayed	Confirmed: Game ends and the winner is displayed.	Pass 
29	Move worker from level 1 to level 2 → no win	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Play until worker of any player moves from level 1 to 2	The game should continue as usual and move on to the next phase.	Confirmed: The game does not end and continues to run.	Pass 
30	Confirm correct winner ID is displayed on win	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Play until Player 1 wins	The scene displaying the winner should show that Player 1 is the winning player	Confirmed: Player 1 is displayed as the winner.	Pass 
Feature: God Powers					
ID	Description	Input	Expected Result	Actual Result	Pass/Fail
31	Ensure assigned god description is displayed in the UI on each player's turn	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click	The UI should be updated to show the correct god name and description	Confirmed: When the game starts, the assigned god's name and description are clearly displayed	Pass 

		“Start Game” → Observe the god description panel during each player’s turn	corresponding to the current player’s assigned god at the start of their turn.	on the left side of the screen.	
32	God power tooltips or descriptions match their behavior	Open executable → Click “Play” → Click “Attain God Powers” → Click “Assign Gods” → Click “Start Game” → Observe the god description panel on each player’s turn → Play one turn to observe the behaviour.	The tooltip or description accurately describes the assigned god’s ability and matches its actual behaviour during gameplay.	Confirmed: The description is correct for each player’s god and behaviour matches.	Pass 
33	Artemis: Move a worker once → game should stay in MOVE phase	Assign Artemis to Player 1 → Start game → On Player 1’s turn, select a worker and move it one space → Do not perform a second move.	After the first move, the game should remain in the move phase, allowing Artemis’s second move option to trigger.	Confirmed: Game remains in MOVE phase and worker is expected to move twice.	Pass 
34	Artemis: Attempt second move to same tile as original → block	Assign Artemis to Player 1 → Start game → On Player 1’s turn, select a worker → Move to a new tile → Attempt to move back to the original tile.	The second move is blocked. A popup or feedback is shown indicating the move is not allowed.	Confirmed: Second move to initial tile is blocked.	Pass 
35	Artemis: Move a worker again to a different adjacent tile → allowed, proceed to BUILD	Assign Artemis to Player 1 → Start game → On Player 1’s turn, select a worker → Move to a new tile → Attempt to move to an adjacent tile.	The second move is successful and the game progresses to the build phase.	Confirmed: Second movement is done and the game proceeds to the next phase.	Pass 
36	Artemis: Attempt second move to invalid tile → blocked	Assign Artemis to Player 1 → Start game → On Player 1’s turn, move a worker →	The second move is blocked. The worker should remain on its	Confirmed: The worker is not moved to the invalid tile.	Pass 

		Attempt to move again to a non-adjacent tile.	previous tile.		
37	Artemis: Skip second move → proceed to build phase	Assign Artemis to Player 1 → Start game → On Player 1's turn, move a worker → Click the "Skip"	The game allows skipping the second move and proceeds to the BUILD phase normally.	Confirmed: Skip option appears on the bottom right and once clicked, skips the second move.	Pass 
38	Demeter: After valid move, build once → game should stay in BUILD phase	Assign Demeter to Player 1 → Start game → On Player 1's turn, move a worker → Build on a tile.	After the first build, the game remains in the build phase, allowing a second optional build.	Confirmed: Game remains in build phase.	Pass 
39	Demeter: Build again on a different tile → allowed, then end turn	Assign Demeter to Player 1 → Start game → On Player 1's turn, move a worker → Build on a tile → Build again on a different tile.	The second build is completed, and the game proceeds to the next player's turn.	Confirmed: Player is allowed to build twice and turn ends.	Pass 
40	Demeter: Try to build on the same tile twice → block	Assign Demeter to Player 1 → Start game → On Player 1's turn, move a worker → Build on a tile → Build again on the same tile.	The second build is blocked. Player must try to build again.	Confirmed: When the player tries to build in the same position, they are allowed to and the game remains in build phase.	Pass 
41	Demeter: Skip second build → turn should still be able to end normally	Assign Demeter to Player 1 → Start game → On Player 1's turn, move a worker → Build on a tile → Click the "Skip" button	The turn ends normally even if the second build is skipped.	Confirmed: Skip button appears at the bottom right and the player is allowed to skip the second build.	Pass 
42	Demeter: Second build attempted on dome/occupied → blocked	Assign Demeter to Player 1 → Start game → On Player 1's turn, move a worker → Build on a tile → Build again on an occupied tile.	The second build is blocked. Player must try to build again.	Confirmed: Second invalid build attempt is blocked.	Pass 