Santorini – Sprint Three

Extension from Sprint Two Team:

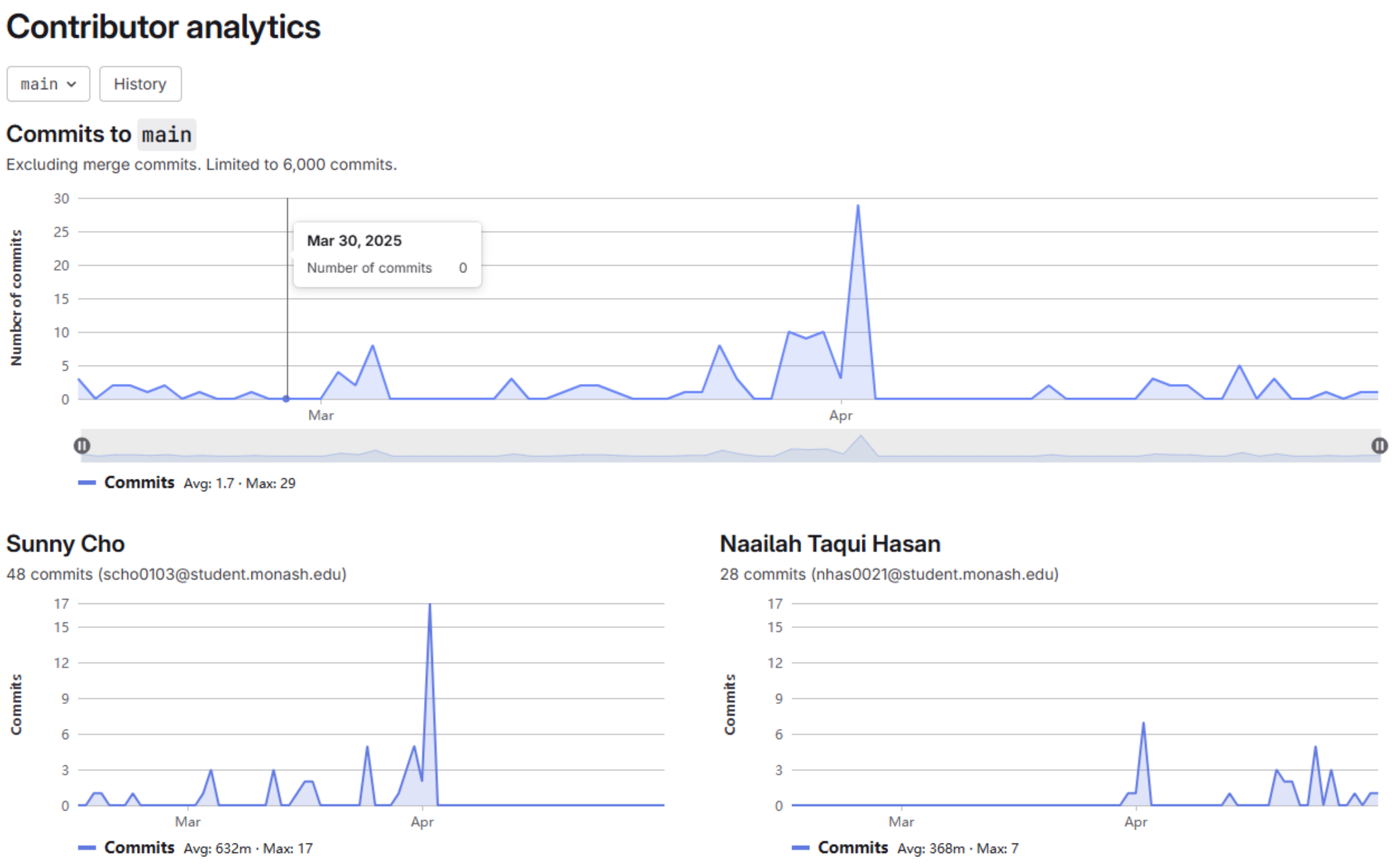
The Team Dream

CL_Tuesday04pm_Team012

Naailah Taqui Hasan

Student ID: 34150943

Contributor Analytics



Contributor analytics

Commits to `main`

Excluding merge commits. Limited to 6,000 commits.

Sunny Cho

48 commits (scho0103@student.monash.edu)

Naailah Taqui Hasan

28 commits (nhas0021@student.monash.edu)

Sprint Three Extensions

Extension 1: Zeus God Power

This extension introduces the unique ability of the god Zeus. Unlike regular builders who can only build on adjacent spaces, Zeus can build beneath himself. Implementing this required modifying the build logic to allow construction on the worker's current position while ensuring the move remained valid within the game rules. If the worker builds a third block under itself, it will not count as a win condition. This added strategic variety and enriched gameplay.

Extension 2: Save and Load System

A full save-load mechanism was implemented to persist the current game state. This includes saving player data, board configuration, turn order, God assignments, and game phase into a JSON file. The load functionality reconstructs the exact game state from the file. This supports user convenience, testing, and longer gameplay sessions.

Extension 3: God Reassignment Every 5 Turns

This feature introduces mid-game god reassignment every 5 turns, adding variety and unpredictability to gameplay. It is implemented directly within the TurnManager using the same randomisation logic as the initial god assignment, without requiring a scene transition. This seamless integration preserves gameplay flow while introducing new challenges. The extension addresses the human value of stimulation: variation in life, by keeping the experience fresh and preventing repetitive gameplay.

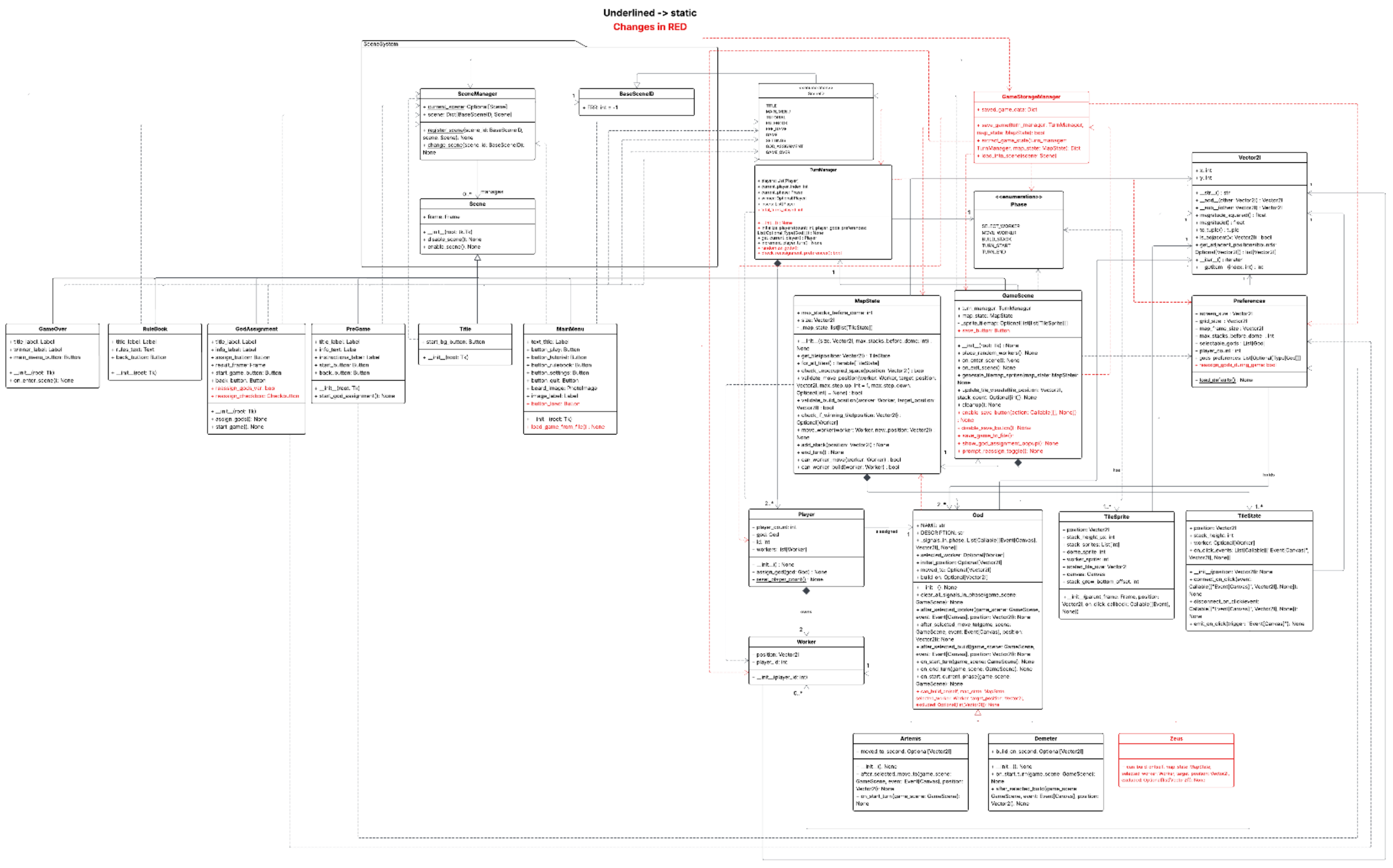
Updated Class Diagram

The updated class diagram introduces several key enhancements highlighted in red to improve game functionality and player experience:

1. **Save and Load System**:
   A new GameStorageManager class enables saving and loading of game state. It integrates with GameScene, and MainMenu through different methods, allowing full restoration of player, board, and turn data.

2. **God Reassignment Feature**:
   The GodAssignment class now includes a reassign_checkbox, giving players the option to reassign gods when loading a saved game. Logic in GameScene and TurnManager supports applying or skipping reassignment based on this preference.

3. **New God – Zeus**:
   A new Zeus class extends God, introducing unique gameplay behavior through overridden methods such as can_build_on(). In the diagram, this is shown by Zeus's custom method implementations listed in red under the Zeus class box.



[Link to LucidChart](#)

Class – Responsibility – Collaboration Diagrams

Class – Responsibility – Collaboration card for Zeus Class

Zeus is a class used to represent the God, Zeus, which inherits from the parent class of God and is a concrete class. Zeus' powers in the game include having the ability to build a block under the selected worker.

Class Name: Zeus	**Subclass of God**
Knows its name	
Knows its power description	
Allows building beneath the selected worker's current position	MapState, Worker, Vector2I

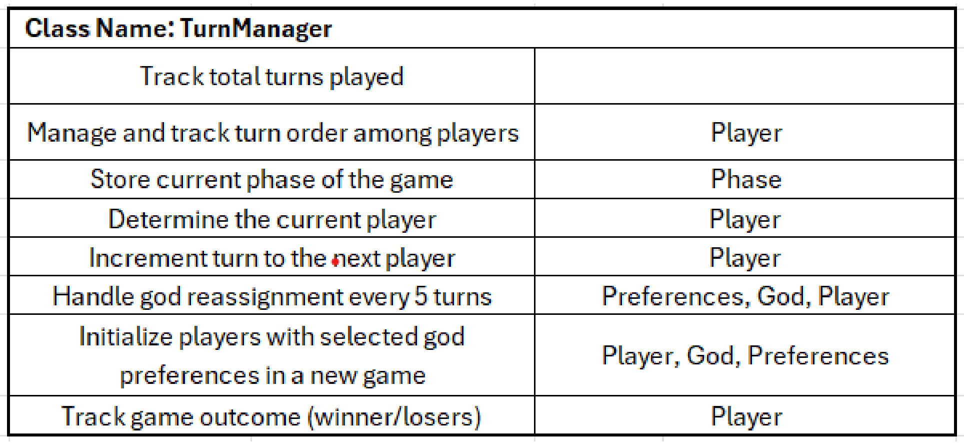
Class – Responsibility – Collaboration card for GameStorageManager Class

The GameStorageManager class is responsible for the Save-Load mechanism of the game which allows player to save the current game to a json file and reload the saved game from this file later.

Class Name: GameStorageManager	
Store the saved game loaded from a file	
Save the current game state to a .json file	TurnManager, MapState
Extract game data	TurnManager, MapState, Vector2I, Player, Worker
Load a saved game into the GameScene	GameScene
Rebuild map state	MapState, Preferences
Rebuild players' worker positions from JSON	Player, Worker
Assign respective god abilities to players during loading	Player, God, Preferences
Reconstruct the visual state of the board	GameScene
Resume game loop after loading	TurnManager, Phase, God

Class – Responsibility – Collaboration card for TurnManager Class

The TurnManager class is responsible for managing the game's turn flow, including tracking the current player, handling phase transitions, determining winners and losers, and reassigning god powers to players every five turns to introduce variation and maintain gameplay balance.



Class Name: TurnManager	
Track total turns played	
Manage and track turn order among players	Player
Store current phase of the game	Phase
Determine the current player	Player
Increment turn to the next player	Player
Handle god reassignment every 5 turns	Preferences, God, Player
Initialize players with selected god preferences in a new game	Player, God, Preferences
Track game outcome (winner/losers)	Player

Design Rationale

New Key Classes:

Zeus:

The Zeus class was introduced as a subclass of God to implement the specific god power: "Your Worker may build a block under itself." This logic requires a unique rule for validating build positions, which differs from the standard implementation in the base God class.

Like the other subclasses of God, Zeus encapsulates a distinct gameplay mechanic that fundamentally alters a phase of the player's turn, in this case, the build phase. Each god in the game introduces specific conditions and exceptions to the default rules, making it necessary to represent them as separate classes to preserve modularity, maintainability, and clarity in the codebase.

The responsibility for enforcing Zeus's unique rule is appropriately handled within the Zeus class itself. Attempting to manage this behaviour in a shared class such as GameScene or TurnManager would lead to bloated, hard-to-maintain code with conditionals for each god's logic. Instead, by overriding the can_build_on() method, Zeus-specific logic is isolated and adheres to the Single Responsibility Principle. This also enables the use of polymorphism, allowing the game engine to interact with all gods uniformly while invoking their specialized behaviors when needed.

In conclusion, the Zeus class is necessary to ensure that the unique rules of Zeus's god power are implemented in a clean, modular, and extensible manner, without disrupting the logic or behavior of other gods or the base gameplay system.

GameStorageManager:

The GameStorageManager was introduced to encapsulate all responsibilities related to saving and loading game state. This functionality involves complex, multi-object serialization and reconstruction logic that would violate the Single Responsibility Principle (SRP) if placed inside an existing class like GameScene or TurnManager.

While GameScene and TurnManager are deeply involved in managing game flow and player turns, their primary focus is on in-game behavior and logic. Including save/load logic in those classes would tightly couple game logic with external system operations, making the code harder to maintain, extend, or test. Separating it out keeps concerns clean and supports future expansion.

This class doesn't need to be instantiated or maintain its own internal state (besides the optional saved_game_data), making it well-suited as a static utility class. It interacts with GameScene, TurnManager, MapState, and other domain classes purely through method calls.

Reassignment of Responsibilities

Unlike some redesigns that involve major refactoring, these extensions did not require reassigning responsibilities from Sprint 2 classes. Instead, the existing architecture was extended using inheritance and separation of concerns. This approach preserved the integrity of Sprint 2 designs while enabling new features to be added in a clean, maintainable way.

Alternative Designs

Zeus powers:

In the current implementation, the Zeus class is a subclass of the God base class and overrides the can_build_on() method to allow Zeus's unique ability: building a block under the worker's own current position. This method is called during the build phase within the shared game flow logic. By overriding only this method, Zeus-specific behavior is neatly encapsulated without affecting other gods or core game mechanics.

To facilitate god-specific build behaviour, the can_build_on() method was introduced in the parent God class. By default, it delegates to the existing validate_build_position() method in MapState to determine if a given build position is valid. This design provides a clear extension point for child classes, such as Zeus, to override the method and implement custom logic. Within the shared game flow, the system now calls can_build_on() instead of directly invoking the map validation function, ensuring that any god-specific rules are consistently and seamlessly enforced.

Alternative designs considered but not chosen:

- Embedding Zeus logic directly inside MapState.validate_build_position() by checking isinstance(god, Zeus)

  Isinstance() is a well-known code smell in object-oriented design. In this design, the MapState class would detect if the player's god is an instance of Zeus and allow building on the current tile accordingly. This violates Separation of Concerns as MapState should only be responsible for structural validation of the board, not god-specific logic. It also introduces tight coupling between low-level game logic and god mechanics resulting in decreased maintainability as more gods are added in the future, violating the Open/Closed Principle and undermining polymorphism.

- Passing the player as a parameter to MapState.validate_build_position() and checking the god name.

  This version avoids direct instance checks by adding a player parameter to the validation method and checking the player's god name (e.g., "Zeus"). However, it still suffers from coupling concerns as MapState must now understand god behavior. It relies on string comparison, which is more fragile and less extensible than class-based polymorphism.

- Overriding on_start_current_phase() of the God class in Zeus and handling the build phase entirely differently.

  Here, Zeus would override the full on_start_current_phase() method to manually redefine the logic of the build phase, including allowing building under self. This reimplements logic that's already correctly handled in the base class leading to repetition of code, heavily violating the DRY principle (Don't Repeat Yourself). It also reduces code reuse and increases the chance of introducing bugs or inconsistencies.

Save-Load System:

The save and load mechanism is managed by the GameStorageManager class. In the GameScene, the save button becomes enabled at the start of a player's turn, before they have taken any action. When the button is clicked, the GameScene calls its save_game() method, which in turn delegates to GameStorageManager to extract the current game state and prompt the user to save it as a .json file on their device.

For loading, when a player chooses to resume a saved game from the main menu, the selected file is stored in GameStorageManager.saved_game_data. Then, inside the on_enter_scene() method of GameScene, this saved data is checked. If present, GameStorageManager.load_into_scene() is called to restore the board layout, player states, gods, and turn information, ensuring the game resumes exactly from where it was left off.

Alternative designs considered but not chosen:

- Loading logic within GameScene

  In the initial implementation, all save and load functionality was embedded directly in the GameScene class. However, this approach led to a violation of SRP by mixing gameplay rendering and scene control with file I/O and deserialization logic. It also made the scene class harder to test and scale. Refactoring this logic into a dedicated class (GameStorageManager) helped maintain a cleaner architecture and isolate concerns.

- Temporary storage of loaded game data in Preferences

  At an earlier point, the loaded game data was temporarily stored inside the Preferences class. This was a poor fit for its role, as Preferences was originally meant for configuration settings (like grid size and UI preferences), not game state. This misuse blurred the purpose of the class, making the system less intuitive. By relocating storage handling to GameStorageManager, the responsibilities are now more logically distributed.

Human Value Feature: God reassignment every 5 turns.

The current implementation performs god reassignment by calling a randomization function, similar to the one used in the original GodAssignment scene, directly within the TurnManager when the total number of turns played reaches a multiple of 5. This ensures the reassignment happens seamlessly within the game's turn cycle without transitioning to a separate scene or interrupting the flow of gameplay. While it may seem unconventional to place this logic in the TurnManager, it aligns with its role in managing turn progression and central game state.

Alternative designs considered but not chosen:

- Switching back to the GodAssignment scene

  One alternative was to switch back to the original GodAssignment scene every time a reassignment was needed. This approach would have allowed code reuse and consistency in UI presentation. However, it was ultimately rejected because transitioning scenes mid-game could disrupt player immersion and create jarring shifts in flow. Additionally, the GodAssignment scene was designed for initial setup, not mid-game modifications, so reusing it would require significant rework.

Trade-offs with current design:

While duplicating the random god assignment logic inside TurnManager does introduce some code repetition, it was deemed acceptable to preserve a smoother in-game experience. In future iterations, this logic could be abstracted into a shared utility to reduce duplication and improve maintainability.

Design Patterns:

Singleton Pattern

As part of the Save and Load game state extension, a design pattern resembling the Singleton was applied through the implementation of the GameStorageManager class. This class is structured as a static utility that centralizes the logic for serializing and deserializing the game's state.

GameStorageManager handles the following:

- Extracting the current state from TurnManager and MapState
- Saving it to a .json file upon user request
- Reconstructing the game scene from saved data when loading

The class includes a class-level variable, saved_game_data, used to store intermediate loading data. The design allows for global access without instantiation, providing a single point of interaction for persistence logic across the application. It ensures that no unnecessary instantiations or object management is introduced. GameStorageManager doesn't need to hold internal state across game sessions. Its role is procedural: extract data, write to file, read from file, inject data back into the scene. This makes it ideal for a stateless utility class or singleton-style implementation, as it doesn't rely on persistent object attributes or lifecycle management.

This approach was chosen to maintain low coupling with gameplay classes and to keep responsibilities clearly separated. Game logic remains within GameScene, TurnManager, and MapState, while GameStorageManager is responsible solely for save/load behavior. It also simplifies future enhancements such as cloud saves or automatic backup features, without altering gameplay mechanics.

Design Pattern considered but not used:

Observer Pattern

The Observer pattern is ideal for propagating changes to multiple parts of a system in response to state updates. However, save/load actions occur at specific user-controlled points and do not require real-time reactive updates across components. Adding observer logic would increase complexity without providing meaningful benefit for this extension.

Factory Method Pattern

This relates to the game load functionality where the gods are assigned to players after game is loaded from a file. Although god instantiation is dynamic (e.g., get_god_by_name()), a full Factory Method pattern was not implemented because the construction logic is straightforward, there are no complex dependencies, configuration variations, or runtime decisions beyond simple name-matching. A lightweight search over Preferences.gods_selectable was a simpler and more appropriate solution for the current scope of gods.

Human Value – Stimulation: Variation in Life:

The third extension, reassigning god powers every 5 turns, aligns with the human value of Stimulation, specifically defined as *“variation in life.”*

This value emphasizes the importance of novelty, surprise, and dynamic change in maintaining interest and enjoyment. In the context of a strategy game like Santorini, stimulation ensures the experience remains engaging by preventing repetitive patterns and encouraging continuous re-evaluation of tactics. Without variation, players might fall into predictable routines; this extension was designed to disrupt that.

In the game, stimulation is manifested through the mid-game reassignment of gods, where all players receive new, randomly selected powers every five full turns. These changes introduce new conditions and playstyles, forcing players to adapt to unfamiliar abilities and strategies. This variation keeps the game exciting and mentally engaging from start to finish.

The user is given the option to enable this feature during the first God Assignment Scene.

This value becomes immediately apparent when:

- The god powers suddenly change during a match.
- Players adjust their moves to accommodate newly assigned abilities.
- The user interface reflects updated god descriptions and names, signalling the shift.

The user is also given the option to enable of disable this feature when they are loading an existing game from a json file.

To implement this, changes were made to the TurnManager to track the number of total turns played. When this count reaches a multiple of five, the game triggers a god reassignment using random selection logic similar to the original GodAssignmentScene. Instead of switching scenes, the reassignment is integrated smoothly into the turn flow, maintaining player immersion.

While Stimulation (variation in life) is the central value supported, this extension also touches on:

- Excitement in life, by introducing sudden shifts in game dynamics.
- Self-Direction – “Curious” / “Choosing Own Goal”. After reassignment, players must rethink their strategy and make independent decisions based on their new powers. It fosters curiosity as they explore different combinations and playstyles.
- Hedonism – “Enjoying Life”. The randomness and variety make the game more fun and enjoyable, not just competitive. This value is supported through the light-hearted and entertaining experience this feature creates.

AI Acknowledgement:

I used ChatGPT (https://chat.openai.com/) to reword and improve the clarity of my design rationale over two iterations. The tool was used to provide a more formal and coherent version of my original written rationale, improving sentence structure, flow, and technical articulation. The output from this tool was modified by incorporating it into my own rationale draft, ensuring the ideas, structure, and technical content reflect my original understanding and intent.

Executable Generation

An executable of the application has been provided, which can be run on Windows, macOS, and Linux platforms.

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.

Sprint Two Design Reflection

Sprint 3 posed a valuable opportunity to reflect on the extensibility of the system designed during Sprint 2. The additions of Zeus, Save and Load, and God Reassignment every 5 turns each surfaced different strengths and weaknesses in the initial design. Below, I reflect on the difficulty of each extension, the design factors that influenced it, and what I would change if I were to re-architect Sprint 2 from scratch. For this purpose, I would also like to briefly discuss certain downsides to the initial design which may not have directly affected the extensions for Sprint 3.

Extension 1: Zues God Power

Difficulty: ★★☆☆☆ (Low)

Adding Zeus was relatively straightforward, thanks to the use of the God superclass and inheritance, introduced in Sprint 2. Each god subclass had a clear structure to override abilities using dynamic event binding (_signals_in_phase) and tracked selected workers and positions cleanly. Zeus's unique build-under-self behaviour fit well into this structure. However, this process revealed deeper design issues with how the God class is structured and its responsibilities.

In our current implementation, the God class is responsible not just for passive rule modification but also for actively handling the entire flow of player interaction during a turn, from input handling to move/build validation and UI state transitions. This tight coupling means gods aren't just augmenting the game logic, they're orchestrating it. This violates the Single Responsibility Principle and limits flexibility.

This design particularly affected Zeus, whose special ability (building under himself) required modifying core build validation logic. To accommodate this, we introduced a can_build_on method in the God class that internally calls the default validation function. While this helped override one specific check for Zeus, it highlighted a deeper issue: if future gods require more complex rule exceptions, we may be forced to rewrite large portions of validation logic, leading to code duplication and DRY violations.

Sprint 2 Design Helped Because:

- Gods were modular via subclasses.
- The game already supported God-specific interaction flows through signals.

What I would change:

- Refactor the God system so gods decorate or extend base rule logic, rather than replacing or duplicating it.
- Extract base movement/build validation into a separate, reusable rules engine, and allow gods to modify it via composition or strategy injection.

- Ensure the God class focuses only on rule modifications, not full control of turn flow or UI interaction.

Extension 2: Save and Load System

Difficulty: ★★★☆☆ (Moderate)

Implementing Save and Load was manageable overall, especially once it became clear that a dedicated class was needed to handle the responsibility. We introduced a new class to encapsulate the save-load functionality and applied the Singleton pattern to ensure consistent access across different parts of the game. This worked well and was easy to incorporate in the right places without disrupting existing logic.

The presence of different Scenes made it possible to include the right option (Button) for the user in the right places. With the save button in the GameScene and the Load button in the MainMenu. All that was need was attaching the functionality to these buttons.

One of the aspects that made this extension smoother was the presence of distinct scenes in the UI architecture. Having separate scenes for the MainMenu and GameScene allowed me to place the appropriate buttons in contextually relevant places, a Load button in the MainMenu and a Save button in the GameScene. This separation of interface concerns meant I didn't need to restructure any UI layouts; all that was required was attaching the save and load functionality to the corresponding buttons.

The main challenge came from the fact that Sprint 2 didn't have a centralised game state model. Critical data was scattered across TurnManager, Player, god instances, and even some UI components. Because of this, saving required us to manually gather and serialize data from multiple sources, while loading meant reconstructing objects and restoring their relationships correctly.

What I would change:

- Introduce a unified GameState abstraction early in the design to act as a single source of truth.
- Maintain strict separation between UI and logic to avoid UI-state reconstruction issues.
- Continue using the Singleton pattern where global access to core systems (like Save-Load) is needed, but ensure they remain testable and decoupled from game flow.
- Preserve scene separation for clear user interface integration points, keeping Save and Load buttons tied to the correct scene was simple and effective.

Extension 3: Human Value Feature – Stimulation

Difficulty: ★★☆☆☆ (Low)

Reassigning gods every 5 turns was a conceptually simple extension that built naturally on the existing turn logic. Once we added a turns_played counter to the TurnManager, it was easy to trigger the reassignment logic at the right time. The presence of a centralized turn manager helped here, as we could track turn progression in one place without needing to modify unrelated parts of the code.

This approach made integration relatively smooth. Because TurnManager now tracked the number of turns, checking for reassignment points was simple. Gods were already stored per-player and replacing them with newly assigned ones was a matter of updating the respective references and reinitialising their signal bindings. The design of god classes as swappable instances, combined with clear tracking of turns, made this feature feasible without introducing major structural changes.

What I would change:

- Extract the random god assignment logic into a shared utility function to eliminate code duplication as seen in TurnManager to randomize god assignment.

Critical Review and Future Strategy

The current design still exhibits notable structural limitations, particularly in the lack of clear separation between UI and game logic. Core gameplay operations remain tightly integrated with interface components, which makes the system difficult to test independently, extend cleanly, or reuse in alternative contexts. This tight coupling increases the likelihood of regressions when changes are made and complicates the implementation of new features. While patterns like Singleton were used in isolated cases, there remains a broader need to apply design patterns more systematically across the codebase, for instance, leveraging Strategy to handle god-specific rule logic, or Observer to decouple turn-based triggers from direct UI control.

Reflecting critically on the extensibility of the Sprint 2 design, it's clear that while additions were achievable, they were often implemented *around* the existing architecture rather than *within* it. This highlighted the lack of early structural foresight and the limitations of a tightly bound design. To improve future development practices, I plan to allocate more design time early in the process to envision likely extensions and structure the code accordingly. Finally, I recognise the importance of refactoring early to address code smells and duplication before they solidify into architectural constraints.

AI Acknowledgement:

I used ChatGPT (https://chat.openai.com/) to assist in structuring and refining the wording of this reflection. The tool helped me improve the clarity, tone, and alignment of the content with the marking rubric by suggesting more formal phrasing and helping articulate my critical evaluation and improvement strategies. All content generated was based on my own ideas, design decisions, and project experience, and I revised the responses to ensure they accurately reflect my understanding and intent.