LET’S GO SPRINT 2 SANTORINI

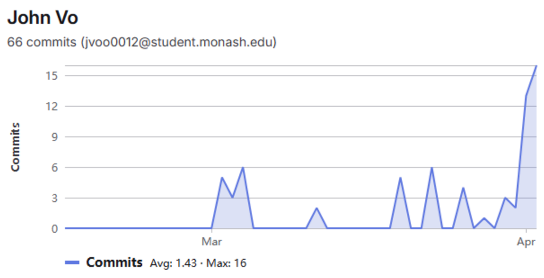
Team Name

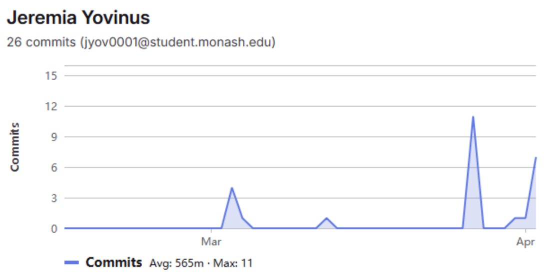
Team Name: Let’s Go

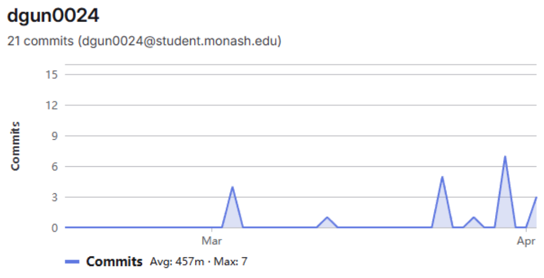
Team Membership

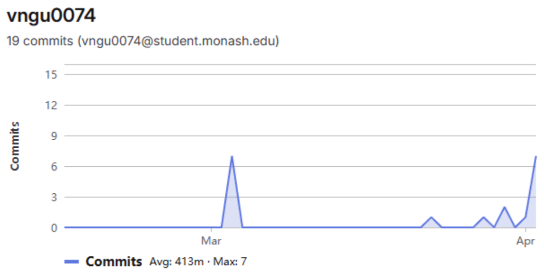
- Duleesha Gunaratne
- John Vo
- Vincent Nguyen
- Jeremia Yovinus

Contributor Analytics



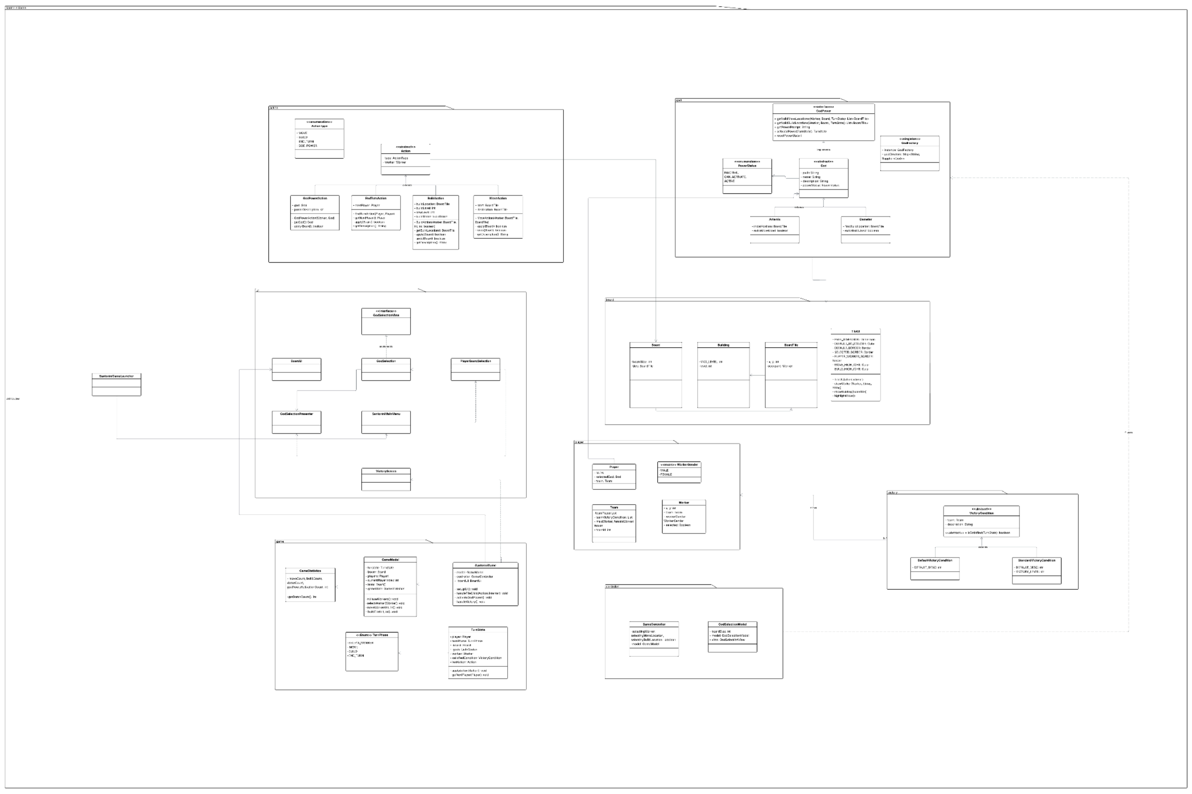






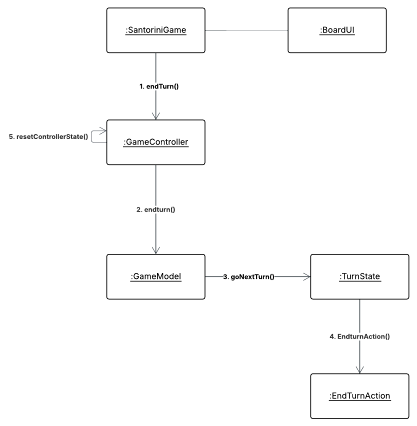
UML Diagram

The UML diagram for the Santorini game models the key components of the game using object-oriented design. It includes core entities like Player, Board, Tile, and Worker, as well as abstract classes like God, VictoryCondition, and Action, allowing for extensibility through subclasses such as Demeter, StandardVictoryCondition, and MoveAction. The diagram also separates logic into layers, including game flow (GameController, TurnState), user interface (BoardUI, VictoryScreen), and game rules (via GodPower and ActionType). This structure supports flexibility, clear responsibility division, and easier maintenance and feature expansion.

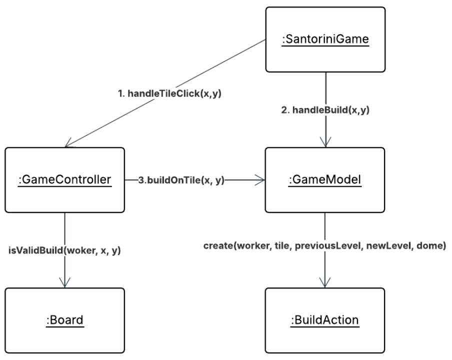


UML Collaboration Diagrams

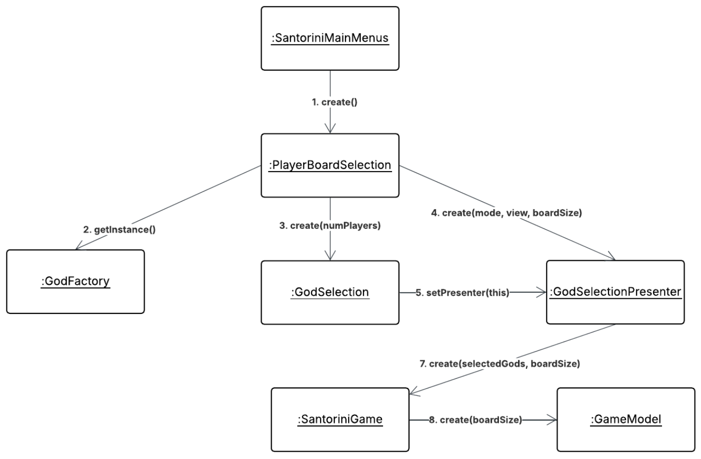
Turn Changing



Building



Game Setup



Design Rationale

Key Classes

The **Game** class, it is essential that it is a class as it manages and encapsulates all of the core components of the game like Board, Players, TurnManager and VictoryCondition. Using a class for game allows it to maintain state across turns, coordinate interactions between object and also follow object-orientated principles like encapsulation and modularity. It is not appropriate for it to be a method as methods cannot hold long-term data or manage complex behaviours, thus making it into a class makes it a more suitable and scalable design choice.

The **TurnState** class is a key component in the design as it encapsulates all the important information related to the current player's turn. It stores the current player, the turn phase (such as selecting or moving a worker), the selected worker, the board, the last action taken, all god powers in play, and all active victory conditions. This class is essential because it maintains and updates the state of the game between turns, coordinates checks for god power activations and win conditions, and ensures a consistent handling of turn-based gameplay. It would not be appropriate for this to be a method because a method cannot persist game state over time, nor can it coordinate between multiple objects or respond to different game events. By making it a class, we allow it to hold state, define clear responsibilities, and support maintainability and scalability in the overall game logic.

The **God** class; encapsulates the logic behind God cards and provides the implementation behind their God powers and the different implications regarding the God power. By having God as an abstract class, it shares functionality and methods and enforces consistent behaviour among all God classes. This enables polymorphic behaviour across the different Gods but does not duplicate code. This allows further extensibility, by making is straightforward to add specific Actions to these God classes.

Key Relationships

The relationship between **Game** and **Board** is an association. This means that the Board **Class** is linked to the **Game** Class and can exist independently. The association between the two classes allow for greater flexibility, such as reusing, replacing or injecting customer boards, this will be useful for possible extension such as adding obstacles or traps. This was chosen not to be a composition as the board as each time the game ended, the board would be destroyed as well, removing possible extensions such as saving the board progression, the association relationship keeps their lifescycles separate, supporting a more modular and reusable design.

Another key relationship is between **Action** and **Game**, and is a dependency. This means neither class contains the other as an attribute, but **Action** does make use of the **Game** class. This is because an **Action**, when executed, leads to a change in the game's state. This includes both the difference in state, e.g. moving a worker or building a level/dome, but also the resulting change to the next phase, e.g. after moving a worker, the phase is changed to the "build" phase. Because each action is self-contained and has limited lifespan, it makes little sense for **Action** to contain the **Game** instance as an attribute, and thus it is not an association.

Another key relationship is **Player** and **Worker** which is a composite relationship as the Player owns the Worker. This shows a strong relationship between the **Player** and **Worker** as workers are vital in the gameplay of players. This is implemented as a composition rather than an aggregation as the **Worker's** life-cycle is exclusive to **Player**, if **Player** is removed, **Worker** must be removed as well. Additionally, there is a strong identity relationship given that the Player is responsible for all movements of **Worker.** This also represents strong containment between the two closes, with the close relationship being a composite relationship.

Inheritance

The **God** class is designed as an abstract class rather than an interface because it needs to maintain state (power status), provide shared attributes (name, description, image path), and implement common behaviors while still requiring specific implementations for unique god powers. This approach allows concrete god classes like Artemis and Demeter to inherit substantial common functionality while only implementing their distinctive abilities, promoting code reuse and consistency across all god types. Making it an abstract class rather than a concrete class ensures that each god must properly implement its unique power activation logic through the abstract checkActivation method, while the abstract class handles the standardized aspects of god behaviors that don't vary between implementations. By inheriting from the God class, subclasses like Artemis and Demeter can focus on their unique functionality while leveraging the common code, reducing duplication and improving maintainability.

The **Action** class is abstract because it establishes a common interface and shared functionality for all game actions while requiring specific implementations for different action types. It contains essential common state (action type, worker) and behavior (getters, basic validation) while deferring the implementation-specific logic to subclasses through abstract methods like apply() and getDescription(). This approach enables the Command pattern, allowing diverse actions (Move, Build, GodPower) to be treated uniformly in the action history and game flow, while ensuring each concrete action class implements its unique logic. The inheritance hierarchy creates a consistent way to execute, describe, and potentially undo different game actions without duplicating code.

The **VictoryCondition** class is abstract because it defines a framework for different winning conditions while requiring specific implementation of the victory checking logic. It maintains common attributes (player, description) and provides utility methods that all victory conditions need, but leaves the actual victory determination (isSatisfied method) to concrete subclasses. This approach allows the game to support multiple types of victory conditions without changing the core game flow - the game can check if any victory condition is satisfied without knowing the specific details of each condition. This follows the Open-Closed Principle, allowing new victory conditions to be added just by creating new subclasses without modifying existing code.

Cardinalities

The cardinality between the **Board** class and the **Worker** class it **1 to 2..***, meaning each board must have at least 2 worker objects present, but can support many more. By default, the game includes 2 players with 2 workers each (total 4 workers). But we do recognise the possible extension in adding the God “Bia” who can remove workers during gameplay and future extensions like the 4 player gamemode, where each player controls 2 workers.

The cardinality between the BoardTile class and the Building class is 1 to 1, meaning every board tile has exactly one building object, and every building belongs to exactly one tile. This relationship reflects the game's design where even empty tiles have a building object with level 0, representing the ground state. The design choice ensures consistency in the data model by guaranteeing that all tiles maintain a building reference, simplifying operations like checking building heights or adding new levels. When construction occurs, the existing building object's level is incremented rather than creating a new building instance, maintaining this one-to-one relationship throughout gameplay and eliminating the need for special handling of "empty" tiles since they are simply tiles with level 0 buildings.

Design Patterns

In our model of the game, we used the Observer pattern for the TurnState and God classes, where the TurnState was the publisher and the God class the subscribers. This was done because the God classes needed to know the current state of the game to determine their status (active, inactive, can be activated). However, we don't know beforehand how many and which gods will be in a game session, so we can't hard code in what instances to notify about the state. The Observer pattern allows us to handle these different cases, like Artemis and Demeter, Ares and Aphrodite and Bia, and so on, by dynamically passing the TurnState what God instances to notify. It also allows for handling cases where a player has lost the game, but it is still ongoing, by simply unsubscribing the lost player's god.

To enhance reusability and minimise repetition, we used the Command pattern to implement the actions a player can take in the Action abstract class. By having Action objects that, e.g., move a worker, we can pass them into buttons to then execute these actions. The buttons, when executed, do not need to know how to move a worker. They only need to know that when pressed, do something with this Action object. This helps separate the model of the game from its view, allowing us to add more Actions without needing to update how buttons interact with the model.

We also used the Visitor Pattern for Actions and God classes where Gods can validate actions through the Visitor Pattern and each God can impose restrictions on certain actions. This separates core Action mechanics from potential God abilities which can encapsulate its own conditional Action mechanics. This can be seen with Artemis who imposes their own logic of being able to move twice in one turn. Our Visitor Pattern allows the Artemis class to define her own conditional move actions but does not impact the MoveAction class. As we scale the game and add more Gods, we can let those specific God classes to encapsulate their own action logic without impacting the fundamental Action nature of the game.

Executable Description

The Santorini game executable runs on Windows and requires Java SDK 24 to function properly (earlier versions may be incompatible). To run the game, users simply double-click the executable file. To create the executable from source code, follow the detailed instructions provided in Sprint 1 under prototypes → john vo.