FIT3077 Sprint 1
Team Information and Technology Prototypes

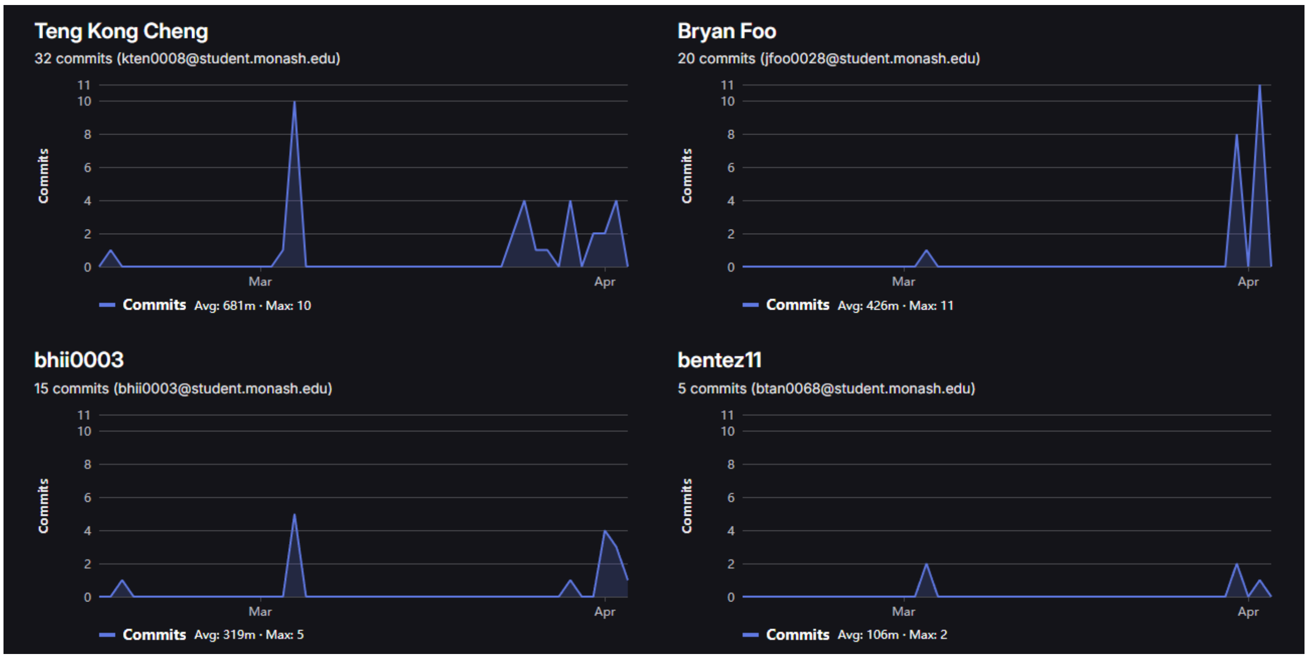
Tutorial: MA_Thursday08am

Team 013: SoftWheres

Prepared by

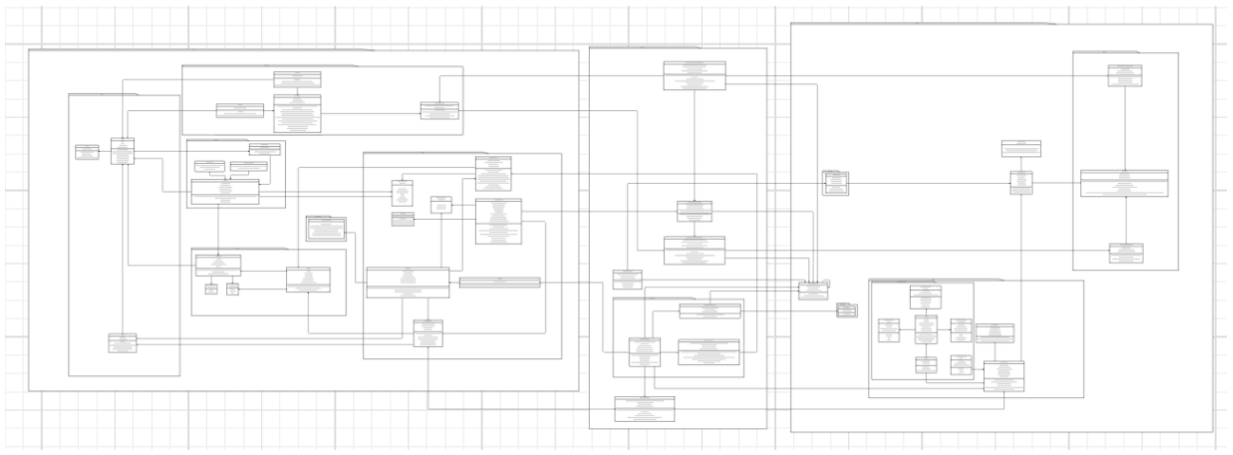
Teng Kong Cheng
Byran Hii Neng Yuan
Foo Jia Wei Bryan
Benjamin Tan En Zhe

1. Screenshot of Contributors Analytics in Git



2. UML Diagram

UML Diagram (click into the link for clearer UML Diagram)



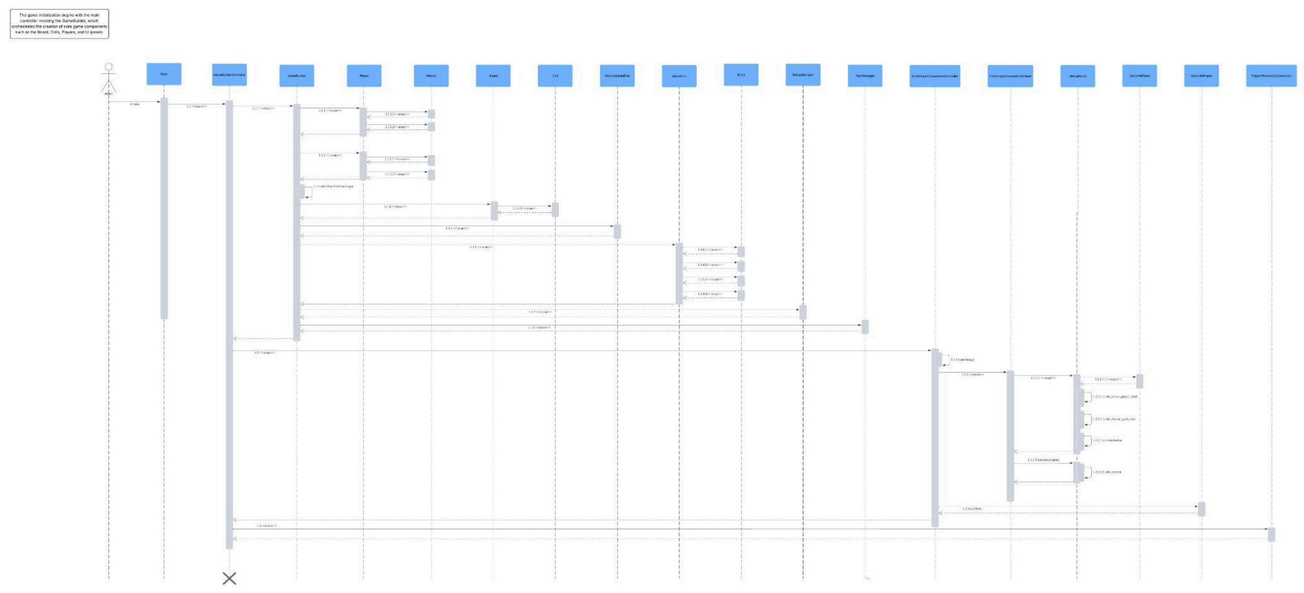
The Santorini game architecture diagram illustrates the complex structure of the game, composed of various panels and controllers that manage different aspects such as setup, gameplay, and game over states. It showcases the interaction between controllers, models, and views, including the handling of God cards, board components, and player actions to create a dynamic gaming experience. Key components include the GameBuilderController, SetupController, TurnController, and the GameState model, each facilitating different stages of the game from choosing gods to managing turns and game state progression.

1. Initialize the Santorini game architecture.
2. Begin with the HomeController to set up the initial game view.
    2.1. Initialize listeners for starting a new game or loading an existing game.
3. Transition to the ChallengerChooseGodController.
    3.1. Display CHALLENGER_CHOOSE_GODS_VIEW.
    3.2. Add listeners for god selection and next action.
    3.3. Handle god card selection using GodCardFactory.
4. Proceed to PlayerChooseGodController for player god selection.
    4.1. Display PLAYER_CHOOSE_GOD_VIEW.
    4.2. Add listeners for god selection and next action.
5. Use GameBuilderController to construct the game.
    5.1. Call challengerChooseGod() and playerChooseGod() methods.
    5.2. CreateGame() method initializes the game components.
6. SetupController manages board setup.
    6.1. Attach listeners for setup actions.
    6.2. Place workers and complete setup.
7. Transition to GameController for gameplay.
    7.1. SetupGame() initializes game view and components.
    7.2. StartGame() begins the turn-based gameplay.
8. TurnController processes each turn.
    8.1. Manage worker selection and actions.
    8.2. Update UI for current phase.
9. GameOverController handles game conclusion.
    9.1. Display GAME_OVER_VIEW and show win panel.
10. Throughout the process, interact with GameState model to track game progression and rules.

3. Sequence Diagram

3.1 Initializing the Board Game Sequence Diagram

Initializing the Board Game Sequence Diagram (click into the link for clearer Sequence Diagram)


The game initialization begins with the main controller invoking the GameBuilder, which orchestrates the creation of core game components such as the Board, Cells, Players, and UI panels.
Actor
Main
GameBuilderController
GameBuilder
Player
Worker
Board
Cell
SetUpManager
TurnManager
SetUpPanel
SantoriniPanel
SantoriniFrame
1 main

The game initialization sequence diagram outlines the structured setup of the Santorini game environment, coordinated by the main controller and the GameBuilder. This process ensures that all core game components—including the Board, Cells, Players, and user interface panels—are systematically created and linked before gameplay begins. The diagram captures the orchestration of setup operations, from component instantiation to establishing player roles and preparing the game view. Key participants in this flow include the MainController, GameBuilder, and various UI-related components responsible for rendering the game's startup visuals and logic.

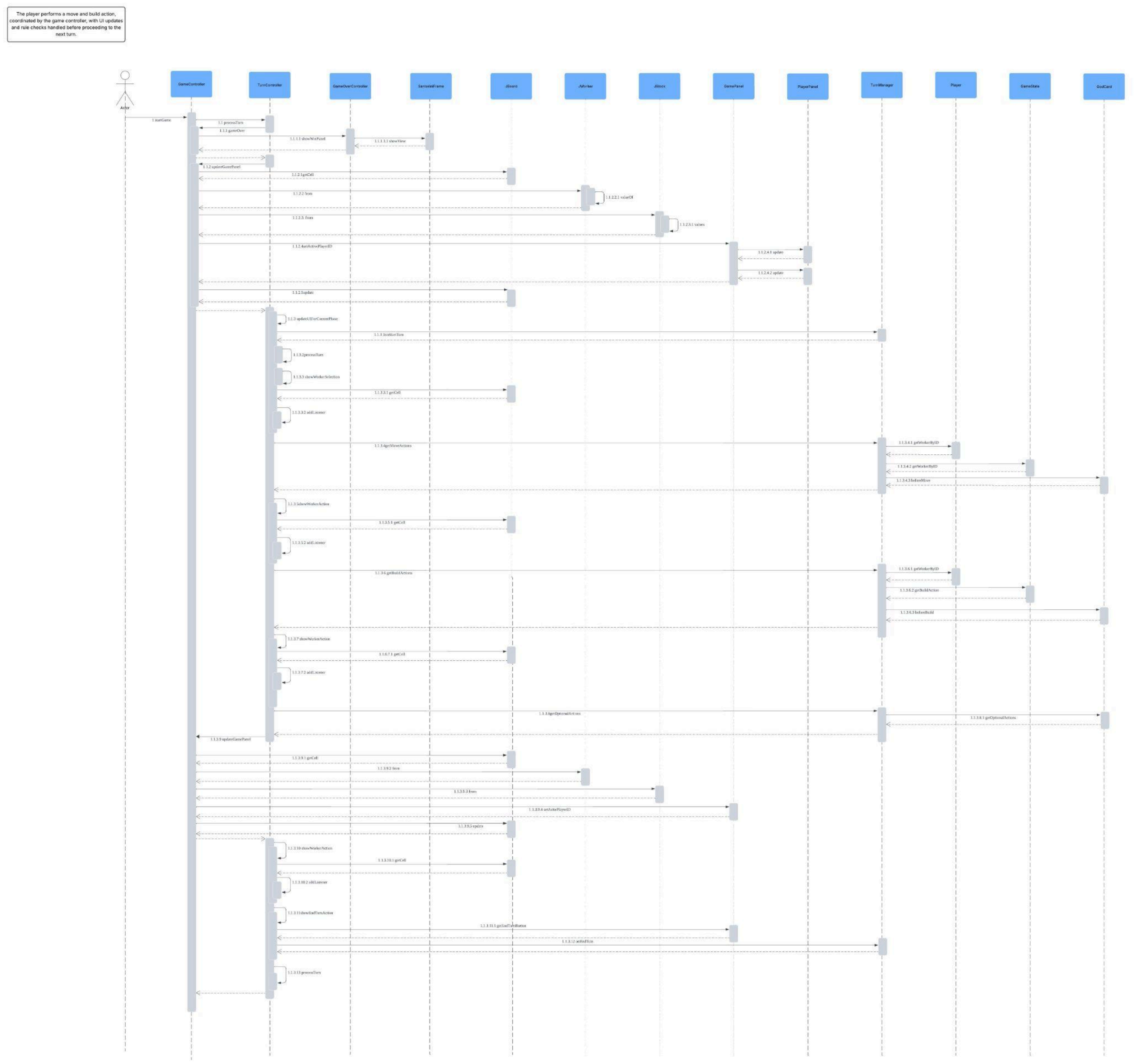
Initializing the Board Sequence:

1. Begin game initialization via the main controller.

2. Call the GameBuilder to manage the component creation process.
- Instantiate the Board used for core gameplay.
- Generate Cell objects to populate the board grid.
- Create and register Player objects for the session.
- Initialize UI panels such as SantoriniPanel, SetUpPanel, and ChallengerChooseGodPanel.
3. Execute panel creation and associate views with controllers.
4. Assign the challenger and prepare God card selection interface.
5. Confirm that all components are properly created and connected.
6. Complete the setup and signal readiness for gameplay transition.

3.2 Move & Build Sequence Diagram

Move & Build Sequence Diagram (click into the link for clearer Sequence Diagram)



The Santorini game process flowchart illustrates the core gameplay loop and the coordinated interactions between game logic and UI components. This diagram highlights the responsibilities of key components such as TurnManager, GameState, GameOverController, and UI panels like PlayerPanel, which together manage player actions, game state updates, and visual feedback. From initiating a turn to processing move and build actions, the system ensures each phase is validated, rendered, and transitioned smoothly. The structured sequence supports a responsive and rule-consistent turn-based game flow, ensuring both logic integrity and user experience.

Move & Build Sequence:

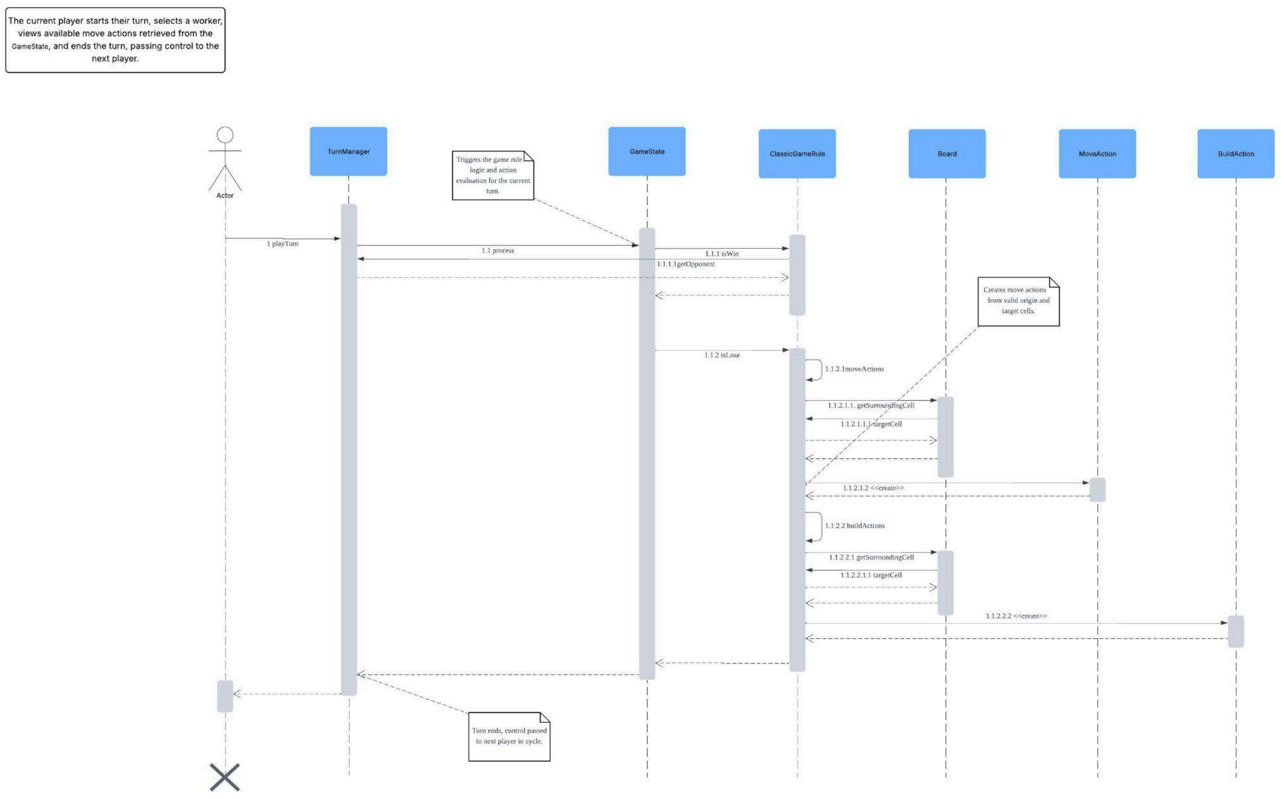
1. Initialize core components, including TurnManager, GameState, GameOverController, and PlayerPanel.
2. Start the current player's turn.

- Trigger onStartTurn to begin the phase.
- Update the UI to reflect the current game phase.

3. Process move actions.
    - Call getMoveActions to retrieve valid move options.
    - Validate using beforeMove if god powers are active.
4. Process build actions.
    - Retrieve build options via getBuildActions.
    - Validate actions with beforeBuild logic.
5. Update the game panel to reflect the new state.
    - Use updateGamePanel for rendering changes.
    - Set and update the active player ID on the UI.
6. Handle optional actions if applicable.
    - Fetch them with getOptionalActions.
    - Attach listeners for user-triggered optional actions.
7. Display available worker actions.
    - Use showWorkerAction to visualize valid operations.
    - Add interactive listeners for those actions.
8. End the player's turn.
    - Execute onEndTurn logic.
    - Refresh UI to reflect the turn change.
9. Repeat this cycle for the next player in sequence.

3.3 Game Turn Sequence Diagram

GameTurn Sequence Diagram (click into the link for clearer Sequence Diagram)


The current player starts their turn, selects a worker, views available move actions retrieved from the GameState, and ends the turn, passing control to the next player.
Actor
TurnManager
Triggers the game rule logic and action evaluation for the current turn.
GameState
ClassicGameRule
Board
MoveAction
BuildAction
1 playTurn
1.1 process
1.1.1 isWin
1.1.1.1getOpponent
Creates move actions from valid origin and target cells.
1.1.2 isLose
1.1.2.1moveActions
1.1.2.1.1. getSurroundingCell
1.1.2.1.1.1 targetCell
1.1.2.1.2 <<create>>
1.1.2.2 buildActions
1.1.2.2.1 getSurroundingCell
1.1.2.2.1.1 targetCell
1.1.2.2.2 <<create>>
Turn ends, control passed to next player in cycle.

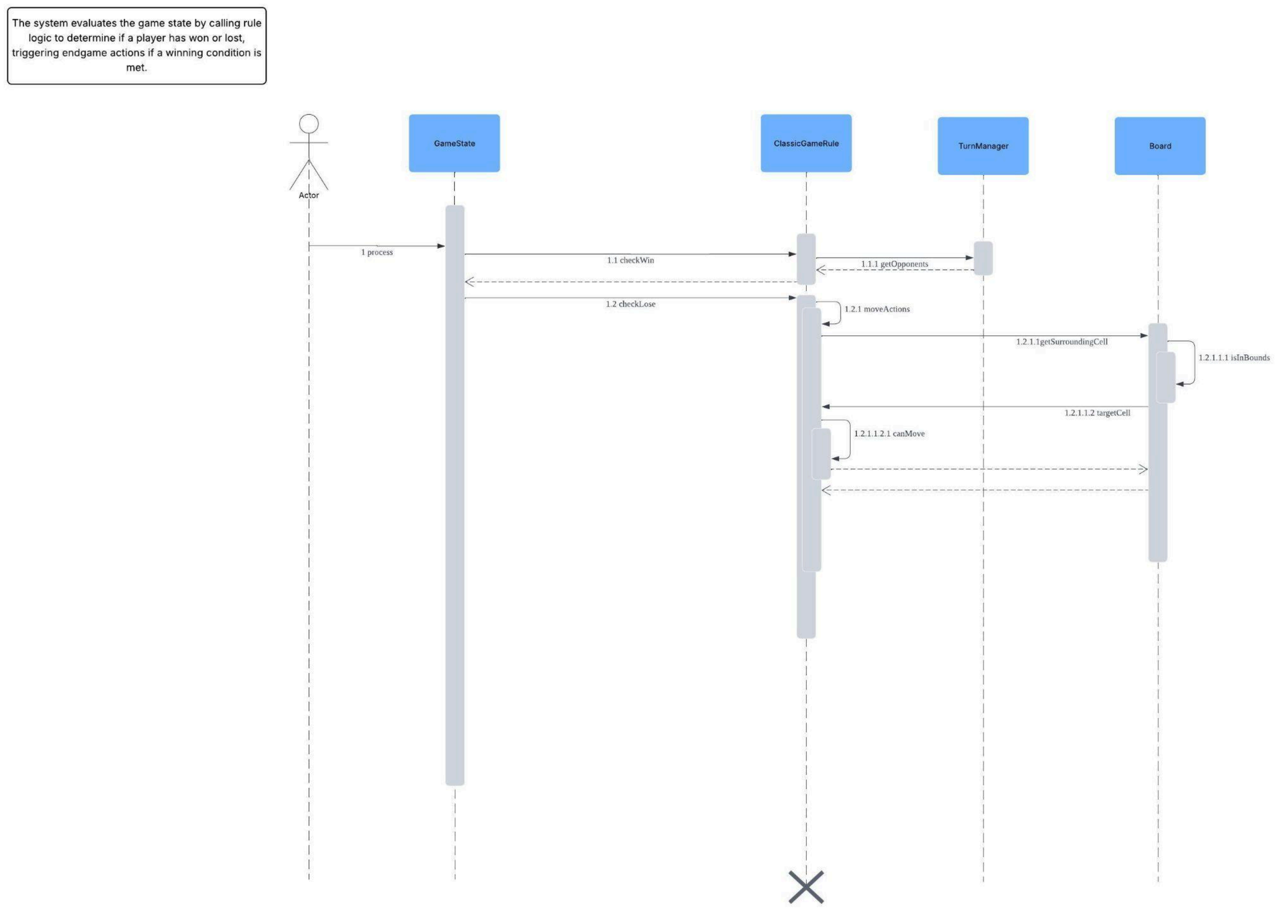
The game turn sequence diagram captures the flow of logic and interaction during a player's turn in the Santorini game. It demonstrates how the system transitions between players, evaluates game conditions using the rule engine, and generates legal move actions based on the current state of the board. Critical components such as TurnManager, GameState, ClassicGameRule, MoveAction, and BuildAction collaborate to ensure that player actions are validated, win/lose conditions are checked, and the game proceeds smoothly in a turn-based cycle.

Game Turn Sequence:

1. End the current turn and pass control to the next player.
2. The new player begins their turn and selects a worker to use.
3. Trigger the game rule logic to evaluate the player's legal actions.
4. Retrieve all valid move actions using the GameState.
    - Identify legal origin and target cells.
    - Instantiate MoveAction objects from the valid cells.
5. Check if the current player satisfies the win condition using isWin.
6. Evaluate loss conditions via isLose.
7. Process the current game state to determine whether to continue or conclude the game.
8. Conclude the player's turn and prepare for the next player's actions.

3.4 Checking Win Condition Sequence Diagram

Checking Win Condition Sequence Diagram (click into the link for clearer Sequence Diagram)


The system evaluates the game state by calling rule logic to determine if a player has won or lost, triggering endgame actions if a winning condition is met.
Actor
GameState
ClassicGameRule
TurnManager
Board
1 process
1.1 checkWin
1.1.1 getOpponents
1.2 checkLose
1.2.1 moveActions
1.2.1.1getSurroundingCell
1.2.1.1.1 isInBounds
1.2.1.1.2 targetCell
1.2.1.1.2.1 canMove

The game state evaluation diagram depicts the internal logic used to assess the ongoing conditions of the Santorini game. It showcases how the GameState collaborates with ClassicGameRule, Board, and TurnManager to determine whether a player has won, lost, or must continue. This process includes checking winning and losing conditions, generating valid move actions, and evaluating the surrounding board environment to support decision-making. The interaction ensures that the game adheres to its rules and that turn transitions and outcomes are handled accurately.

Checking Win Condition Sequence:

1. Initiate the game state evaluation through the GameState component.
2. Use ClassicGameRule to apply win/loss logic.
    - Call checkWin to determine if the player has achieved a victory condition.
    - Call checkLose to determine if the player has no remaining legal actions.
3. If a win condition is met, trigger the endgame sequence.
4. If a lose condition is detected, proceed to evaluate available moves.
5. Generate move actions based on current board state.
    - Use canMove to validate the feasibility of any actions.
    - Use getSurroundingCell to retrieve adjacent cells for movement.
        - Validate cells with isInBounds.
        - Identify legal target cells for movement.

4. Design Rationale

4.1 Core Class Design Rationale

This section explains the key classes implemented in the game logic and why each is a class rather than a method. These classes encapsulate state, enforce rules, and enable modular game architecture, which would not be feasible using methods alone.

4.1.1 Block Class

Purpose: Represents a single unit of construction (e.g., level 1, 2, 3, or dome) placed during a build action.

Key Behavior:

- canStack() enforces legal stacking order.
- Prevents invalid construction through constructor-level checks.

A block carries state (BlockType) and encapsulates validation logic. If implemented as a method (e.g., buildBlock(...)), this logic would be scattered and error-prone, lacking cohesion. The class design supports future extensibility (e.g., special block types for god powers).

4.1.2 BlockType Class

Purpose: Defines construction stages in Santorini (Levels 1–3, Dome).

Key Behavior:

- Converts from level via fromLevel().
- Enforces level transitions via getNextLevel() and isDome().

Construction stages are more than constants — they involve rules and transitions. Representing them as a class makes this logic centralized and type-safe, rather than relying on integer values and conditionals spread across unrelated functions.

4.1.3 Position Class

Purpose: Represents (x, y, z) coordinates of a worker or tower block.

Key Behavior:

- Encapsulates coordinate data and methods like isAdjacent().
- Implemented as a Java record for immutability.

Position is a domain-specific concept with associated logic (e.g., adjacency checks), so it deserves its own type. Using separate integers in methods would break encapsulation and lead to repetitive, error-prone code.

4.1.4 Cell Class

Purpose: Represents one square on the board.

Key Behavior:

- Maintains occupant state.
- Supports tower stacking via buildBlock().
- Combines position and tower height.

Cells manage both data (e.g., occupancy, tower height) and logic (e.g., building rules), so combining these in a single object ensures cohesion and avoids complex, error-prone procedural code.

4.1.5 Board Class

Purpose: Represents the 5x5 grid of the Santorini game.

Key Behavior:

- Provides getCell(), getSurroundingCell(), and getUnoccupiedCells().
- Enforces bounds and occupancy rules.

A 2D array can store state but not behavior. A class allows encapsulation of neighbor logic, cell access, and search patterns, which would otherwise require duplicative logic in multiple places.

4.1.6 Abstract Action Class

Purpose: Base class for any action that modifies the game state (e.g., move, build).

Key Behavior:

- Defines execute(GameState) as abstract.
- Tracks phase transitions and whether the action is active.

Actions must track internal state (target cell, involved worker, transition phase). Methods can't store state across invocations. Classes also allow dynamic behavior changes (e.g., through inheritance for god powers).

4.1.7 MoveAction Class

Purpose: Moves a worker from one cell to another.

Key Behavior:

- Validates and updates worker and cell positions.
- Transitions game from MOVE to BUILD phase.

Encapsulating movement as an object allows it to be queued, reversed (for undo), or modified (by god powers). This flexibility is crucial in a turn-based game engine.

4.1.8 BuildAction Class

Purpose: Handles construction after a worker has moved.

Key Behavior:

- Places a block in a target cell.
- Transitions the game from BUILD to END_TURN.

As with MoveAction, we may need to store state (which cell, which worker), track effects, or extend behavior (e.g., by a god power allowing double build). These are not possible with stateless methods.

4.1.9 TurnManager Class

Purpose: Controls the flow of a player's turn through phases: SELECT → MOVE → BUILD → END.

Key Behavior:

- Validates action order.
- Triggers GodCard hooks (if enabled).
- Updates current phase.

Turn management involves stateful logic that spans multiple steps in a sequence. Encapsulating this in a class ensures clear ownership of the turn state, keeps validation logic organized, and allows easy expansion for advanced rules or god powers.

4.2 Relationship Justification

4.2.1 Key Relationships

Relationship	Type	Justification
Game → Board	Composition	A Board cannot exist independently; each Game owns its own board.
Game → Player	Aggregation	Players can exist outside the game context (e.g., across matches), so they are not destroyed with it.
Player → Worker	Composition	Workers are directly tied to the player and cease to exist if the player does.

4.2.2 Key Cardinality Decisions

Relationship	Cardinality	Justification
Player → Worker	1 → 2	Players always control exactly two workers. It's a fixed Santorini rule.
GodCard → Player	0..1 → 1	A player may or may not be assigned a god card. This enables both standard and god-powered play.

4.2.3 Inheritance Design Decisions

- Used In:
    - AbstractAction → MoveAction and BuildAction
        - These share the same structure (execute, targetCell, etc.)
    - Block → Level1/2/3/Dome (optional if polymorphic behavior is needed)
    - GodPower → SpecificGodPower (PanPower, AthenaPower)

Justification:
Inheritance provides reusability and flexibility for action logic and god behavior. It reduces code duplication and supports dynamic behavior changes.

4.3 Design Patterns Used or Considered

Not Used, But Considered:

Design Pattern	**Why It Was *Not* Used**
Observer	Could be used for event updates (e.g., UI reacts to game state), but our architecture doesn't push updates.
Factory	Could help instantiate GodCard dynamically, but current implementation hardcodes a few cards.
Strategy	A candidate for handling god behavior per player, but simpler inheritance was more practical.

Patterns We Implicitly Follow:

- Command Pattern via Action classes (MoveAction, BuildAction): encapsulate requests with logic and undo capability.
- State Machine logic in TurnManager: defines transitions from SELECT → MOVE → BUILD → END.

5. Executable

From the source code, go into Santorini directory, then go into src directory, then open Main.java file and run it.