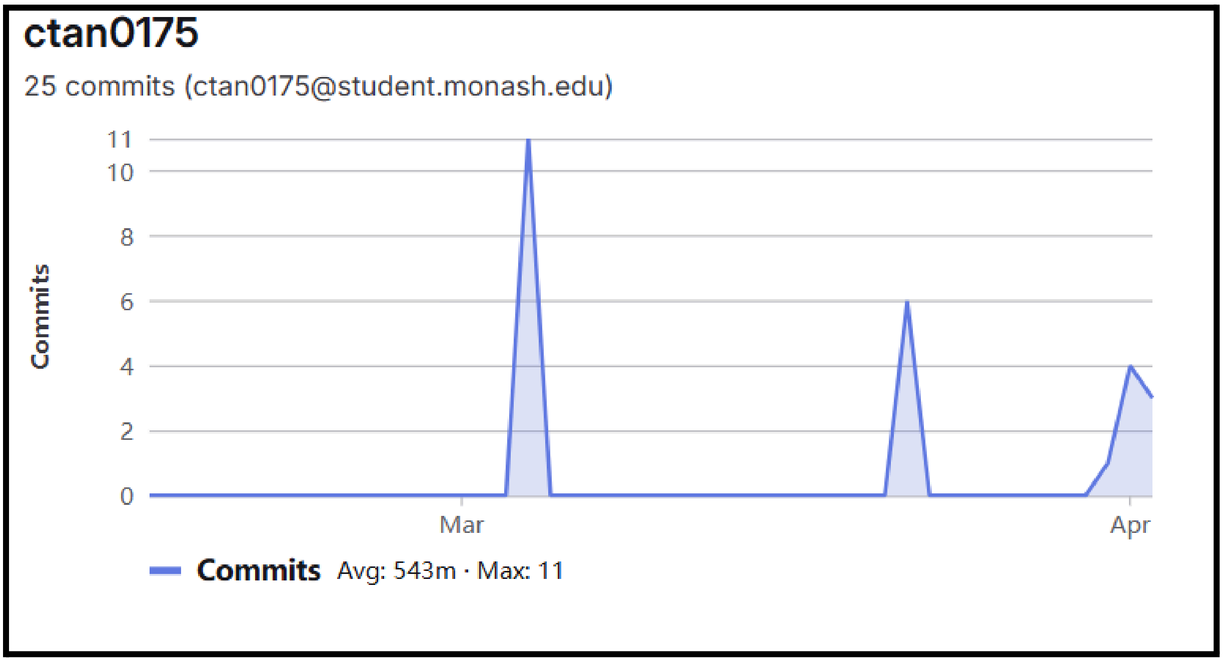
Sprint 2 Report

Team Name: SnakeByte

Team Members:

- Charlene Tan Ling Yi: +60176045058
- Reynaldi Marshen Sutanto: +601112805715
- Alvin Andrean : +601127201597
- Steven Tirtadjaja: +60129259472

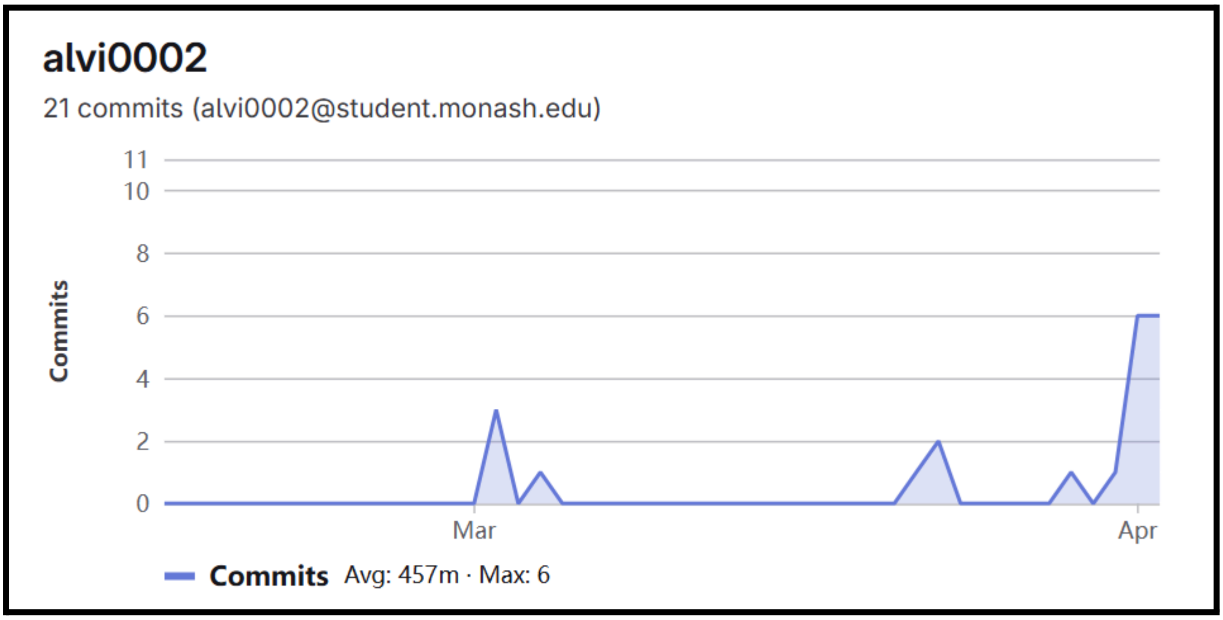
Contributor Analytics



ctan0175

25 commits (ctan0175@student.monash.edu)

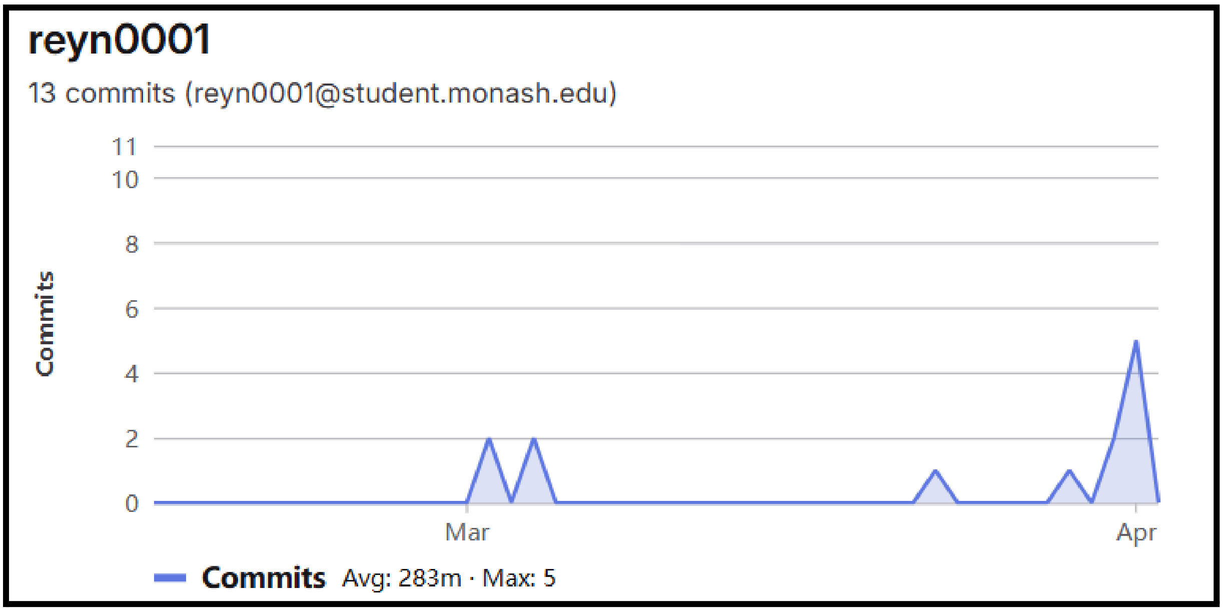
Commits Avg: 543m · Max: 11



alvi0002

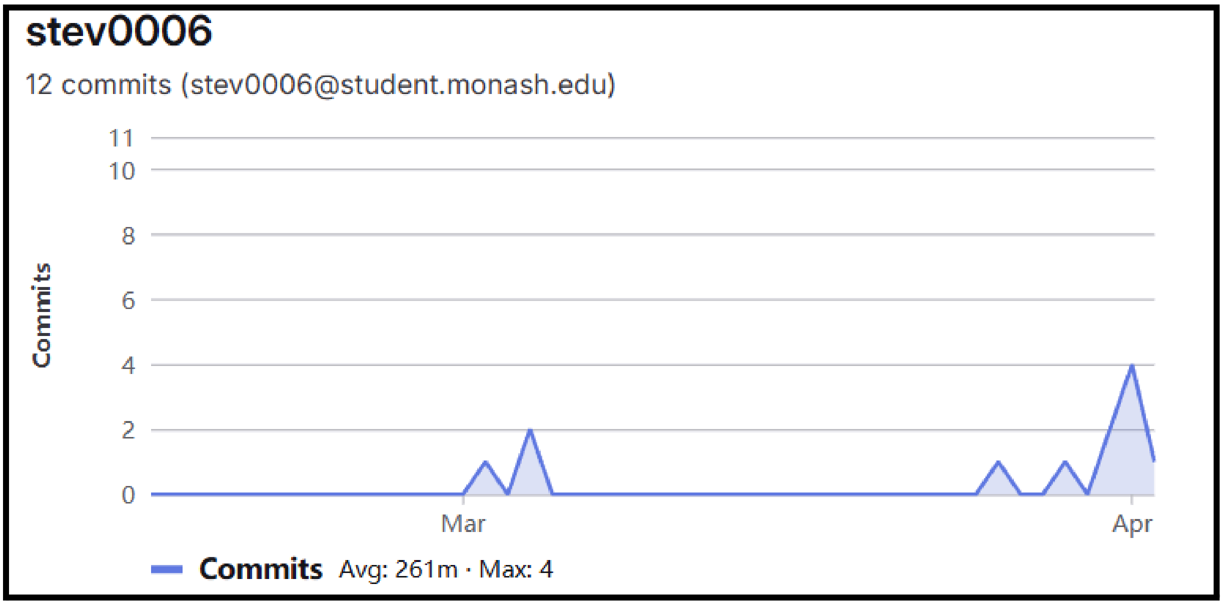
21 commits (alvi0002@student.monash.edu)

Commits Avg: 457m · Max: 6



reyn0001

13 commits (reyn0001@student.monash.edu)

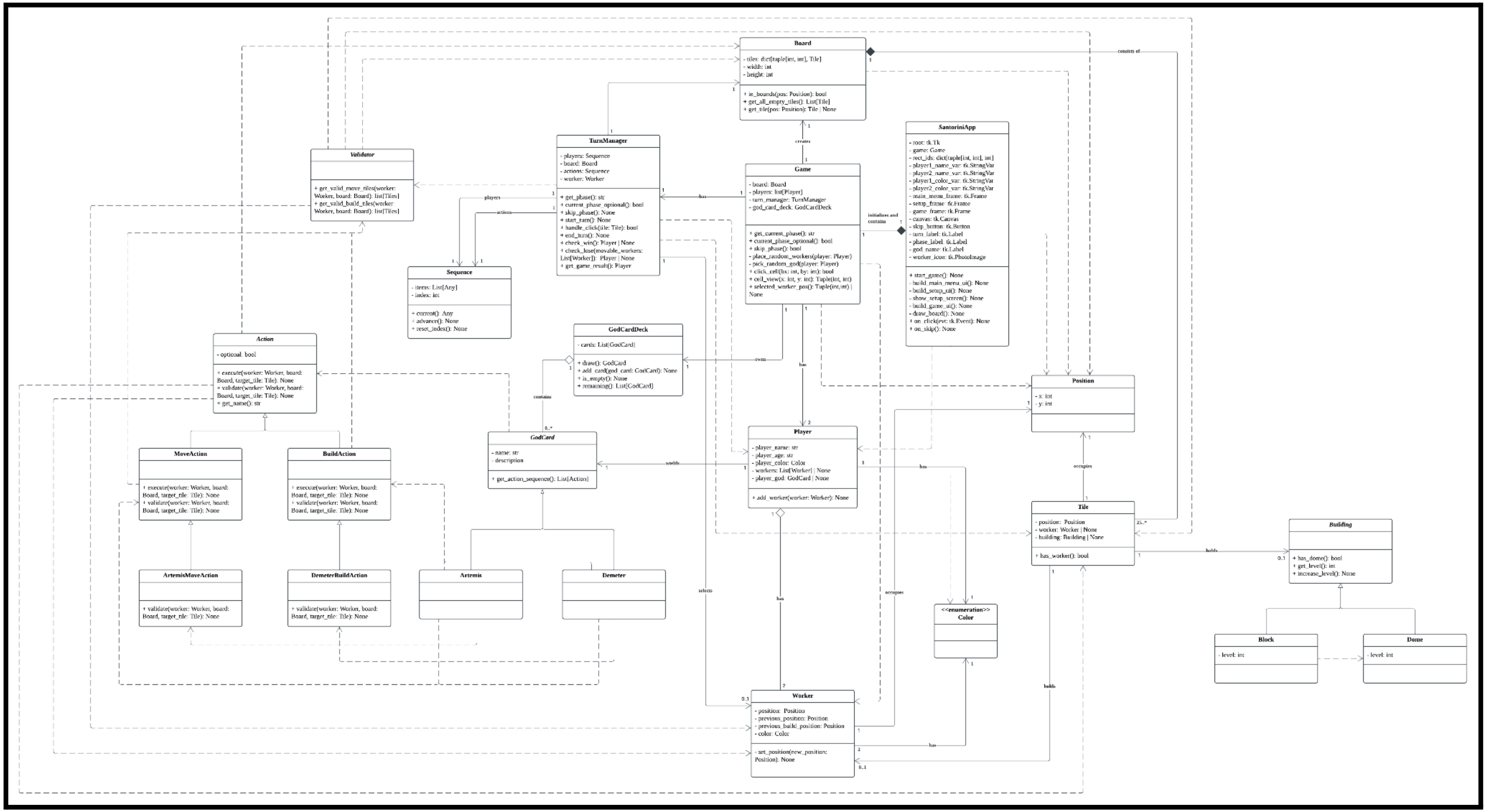


stev0006

12 commits (stev0006@student.monash.edu)

All commits can be checked on our repository. https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-groups/MA_Tuesday06pm_Team007/project/-/graphs/main?ref_type=heads

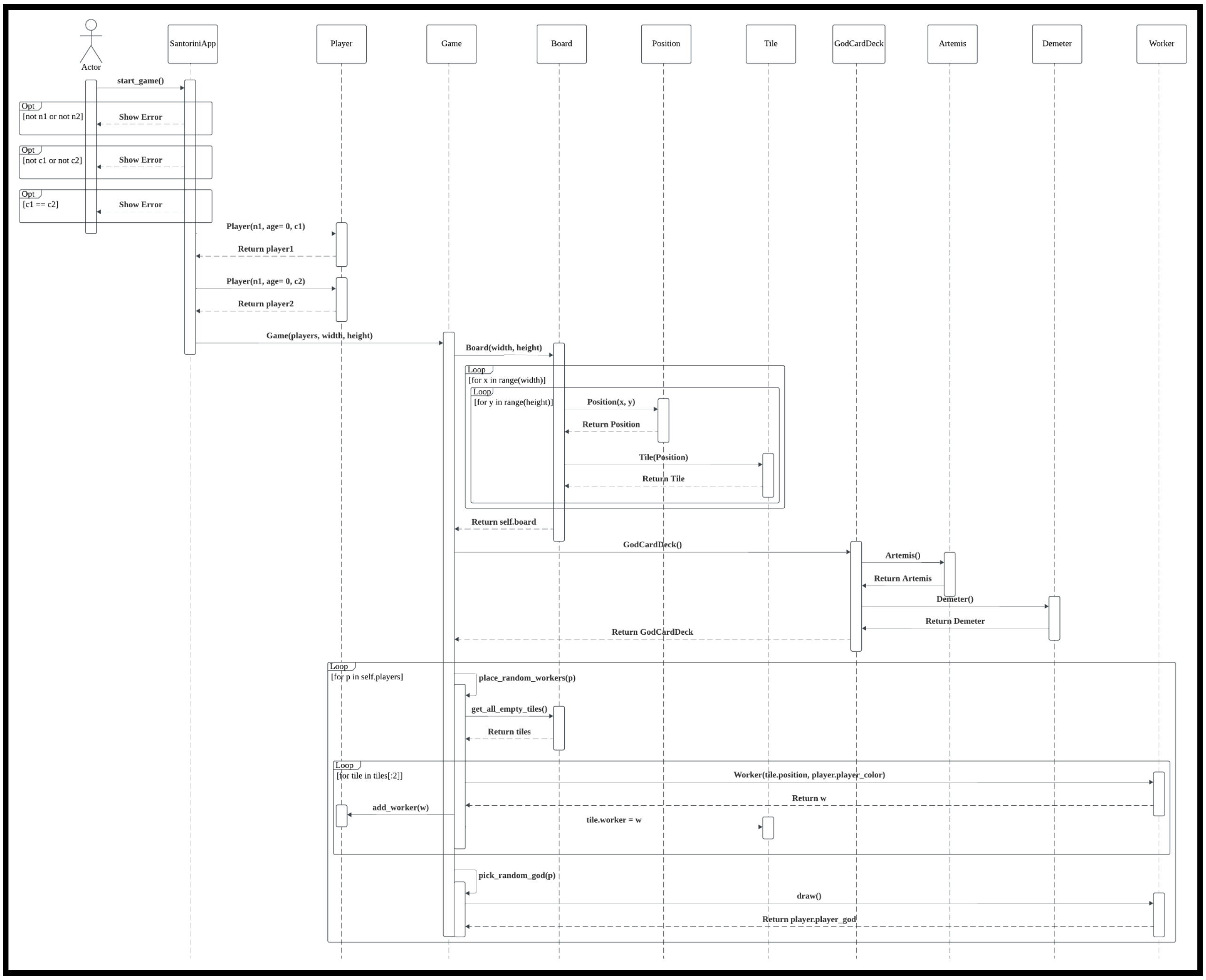
Class Diagram


Board
- tiles: dict[tuple[int, int], Tile]
- width: int
- height: int
+ in_bounds(pos: Position): bool
+ get_all_empty_tiles(): List[Tile]
+ get_tile(pos: Position): Tile | None
consists of
TurnManager
- players: Sequence
- board: Board
- actions: Sequence
- worker: Worker
+ get_phase(): str
+ current_phase_optional(): bool
+ skip_phase(): None
+ start_turn(): None
+ handle_click(tile: Tile): bool
+ end_turn(): None
+ check_win(): Player | None
+ check_lose(movable_workers: List[Worker]): Player | None
+ get_game_result(): Player
Validator
+ get_valid_move_tiles(worker: Worker, board: Board): list[Tiles]
+ get_valid_build_tiles(worker: Worker, board: Board): list[Tiles]
players
actions
has
creates
Game
- board: Board
- players: list[Player]
- turn_manager: TurnManager
- god_card_deck: GodCardDeck
+ get_current_phase(): str
+ current_phase_optional(): bool
+ skip_phase(): bool
- place_random_workers(player: Player)
- pick_random_god(player: Player)
+ click_cell(bx: int, by: int): bool
+ cell_view(x: int, y: int): Tuple(int, int)
+ selected_worker_pos(): Tuple(int,int) | None
initializes and contains
SantoriniApp
- root: tk.Tk
- game: Game
- rect_ids: dict[tuple[int, int], int]
- player1_name_var: tk.StringVar
- player2_name_var: tk.StringVar
- player1_color_var: tk.StringVar
- player2_color_var: tk.StringVar
- main_menu_frame: tk.Frame
- setup_frame: tk.Frame
- game_frame: tk.Frame
- canvas: tk.Canvas
- skip_button: tk.Button
- turn_label: tk.Label
- phase_label: tk.Label
- god_name: tk.Label
- worker_icon: tk.PhotoImage
+ start_game(): None
- build_main_menu_ui(): None
- build_setup_ui(): None
- show_setup_screen(): None
- build_game_ui(): None
- draw_board(): None
+ on_click(evt: tk.Event): None
+ on_skip(): None
Sequence
- items: List[Any]
- index: int
+ current(): Any
+ advance(): None
+ reset_index(): None
GodCardDeck
- cards: List[GodCard]
+ draw(): GodCard
+ add_card(god_card: GodCard): None
+ is_empty(): None
+ remaining(): List[GodCard]
owns
contains
Action
- optional: bool
+ execute(worker: Worker, board: Board, target_tile: Tile): None
+ validate(worker: Worker, board: Board, target_tile: Tile): None
+ get_name(): str
GodCard
- name: str
- description
+ get_action_sequence(): List[Action]
wields
MoveAction
+ execute(worker: Worker, board: Board, target_tile: Tile): None
+ validate(worker: Worker, board: Board, target_tile: Tile): None
BuildAction
+ execute(worker: Worker, board: Board, target_tile: Tile): None
+ validate(worker: Worker, board: Board, target_tile: Tile): None
ArtemisMoveAction
+ validate(worker: Worker, board: Board, target_tile: Tile): None
DemeterBuildAction
+ validate(worker: Worker, board: Board, target_tile: Tile): None
Artemis
Demeter
Position
- x: int
- y: int
Player
- player_name: str
- player_age: str
- player_color: Color
- workers: List[Worker] | None
- player_god: GodCard | None
+ add_worker(worker: Worker): None
occupies
Tile
- position: Position
- worker: Worker | None
- building: Building | None
+ has_worker(): bool
holds
Building
+ has_dome(): bool
+ get_level(): int
+ increase_level(): None
Block
- level: int
Dome
- level: int
<<enumeration>>
Color
selects
has
occupies
holds
Worker
- position: Position
- previous_position: Position
- previous_build_position: Position
- color: Color
- set_position(new_position: Position): None
1
2
0..1
0..*
25..*

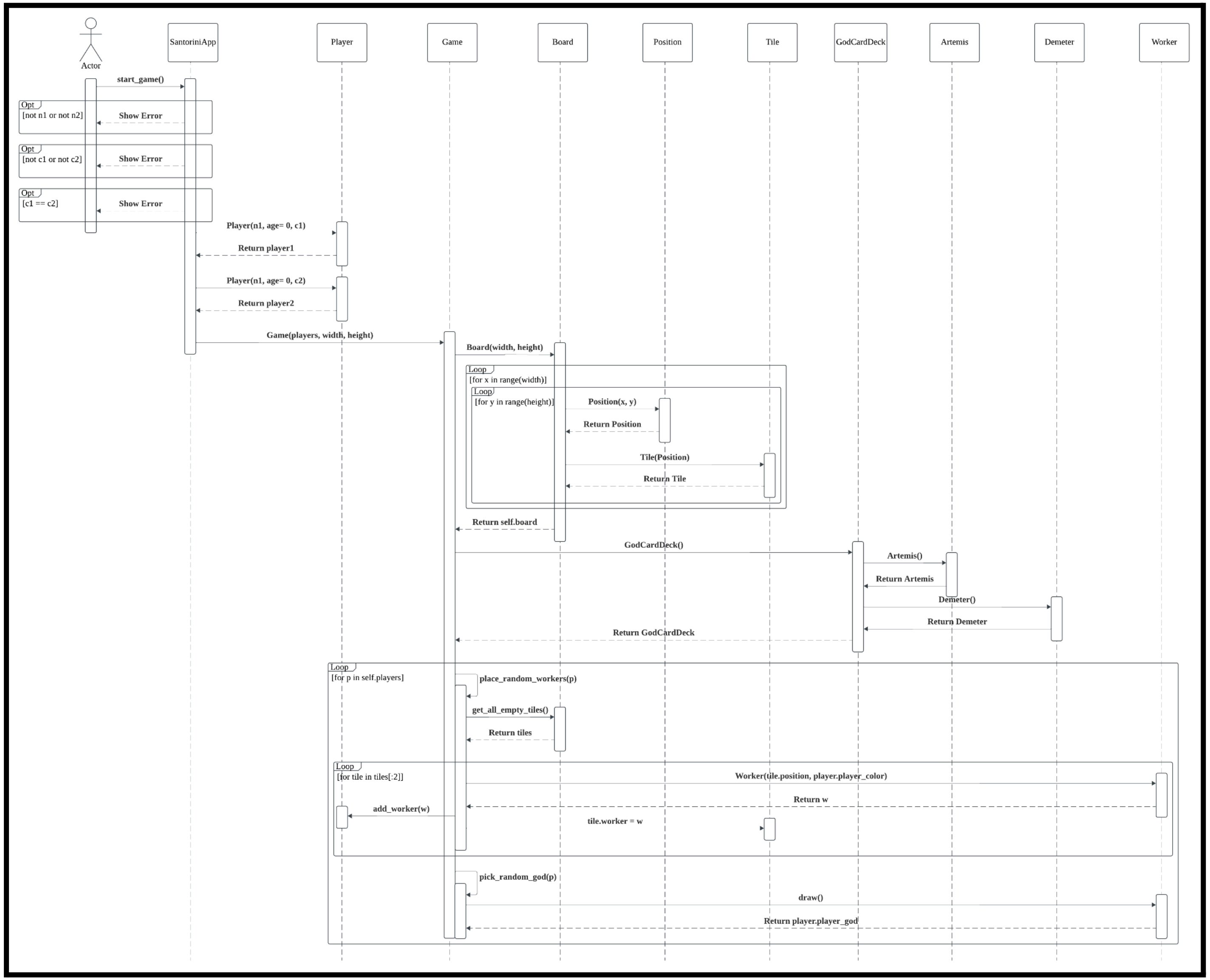
Previous draft versions can be seen inside our repository Sprint 2/uml_class_diagram/in_progress_uml/class_diagram. https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-groups/MA_Tuesday06pm_Team007/project/-/tree/main/Sprint%202/uml_class_diagram/in_progress_uml/class_diagram?ref_type=heads

Sequence diagrams

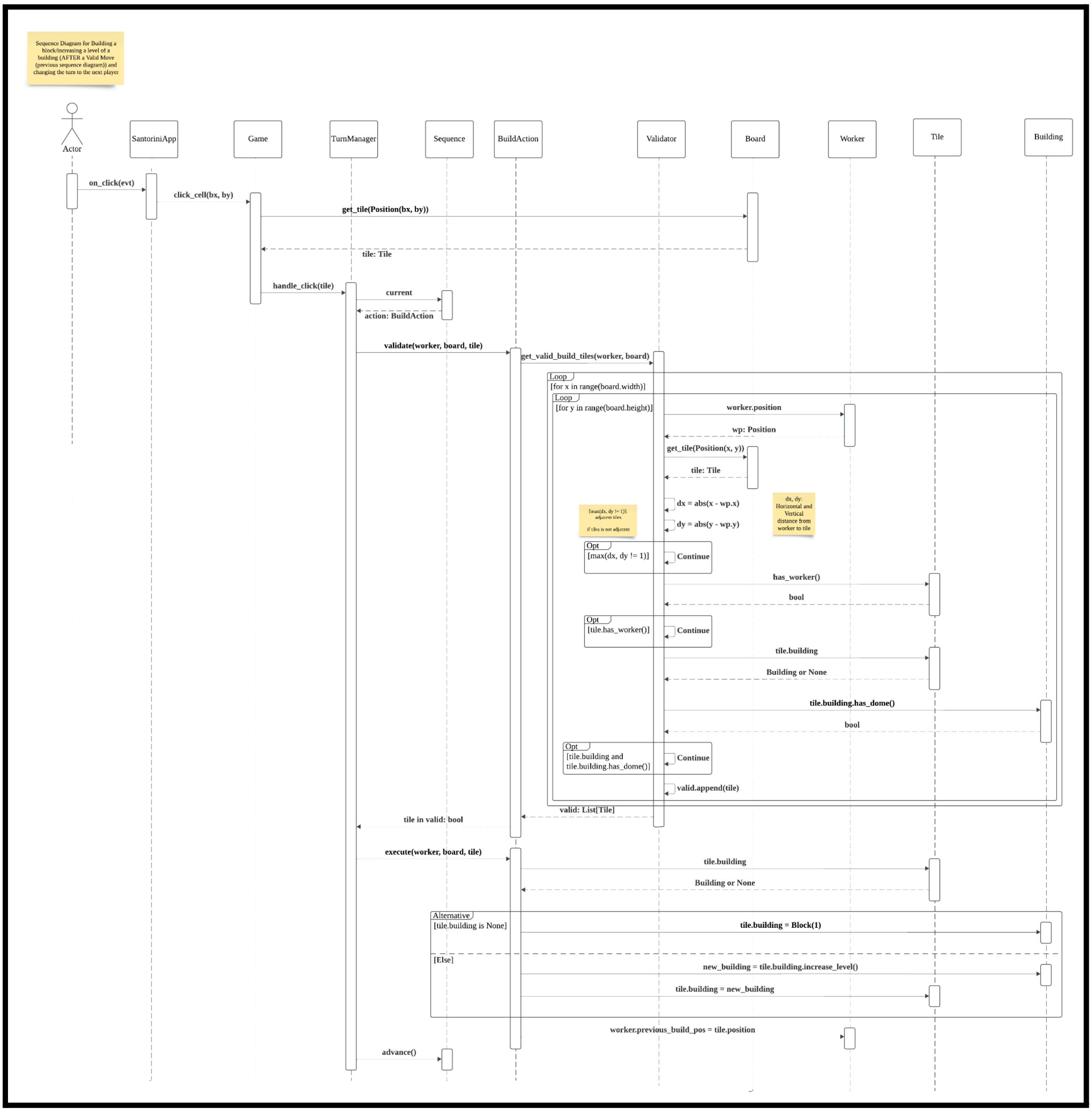
Functionality 1: Initial game setup



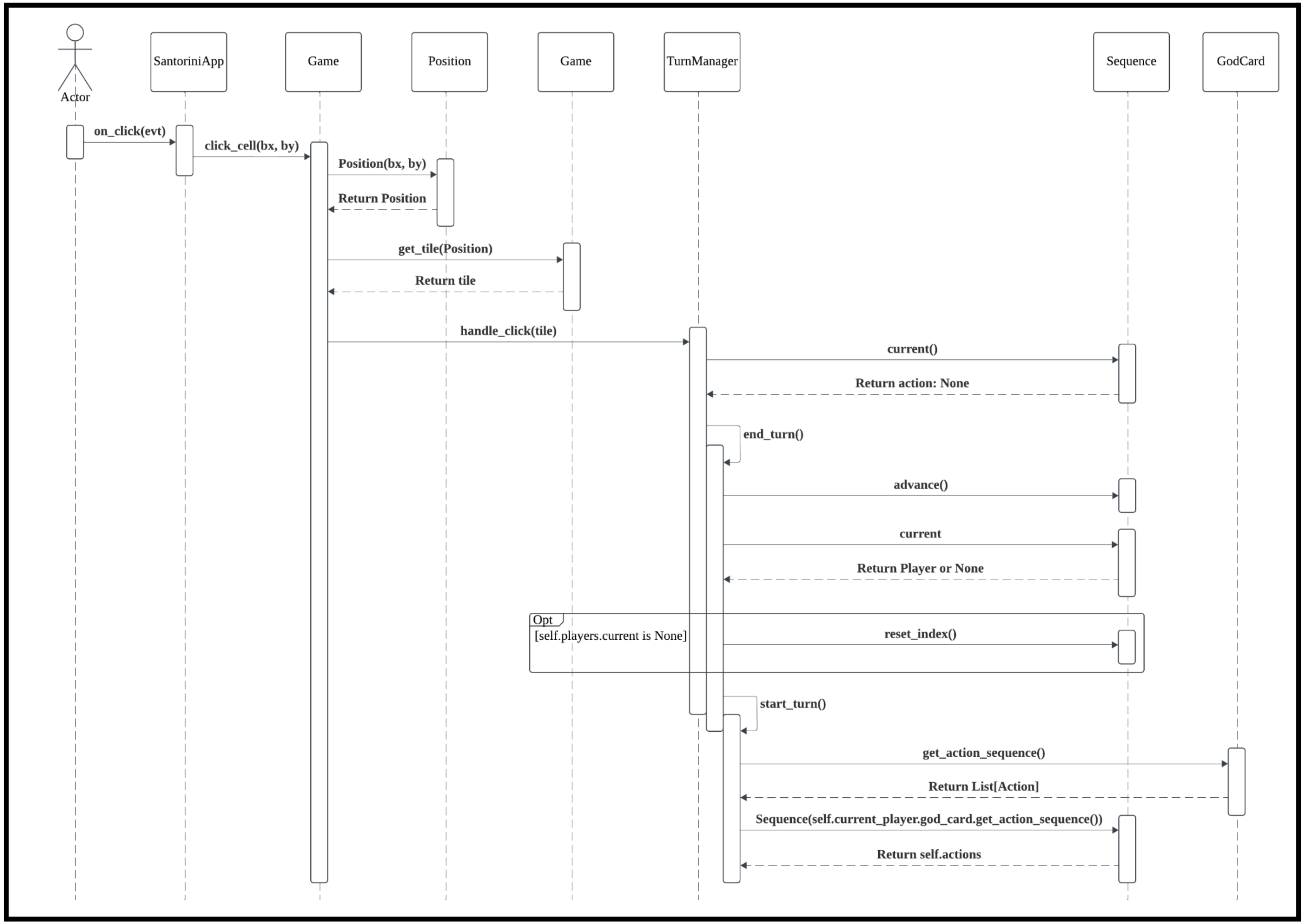
Functionality 2: Selection and movement of player worker



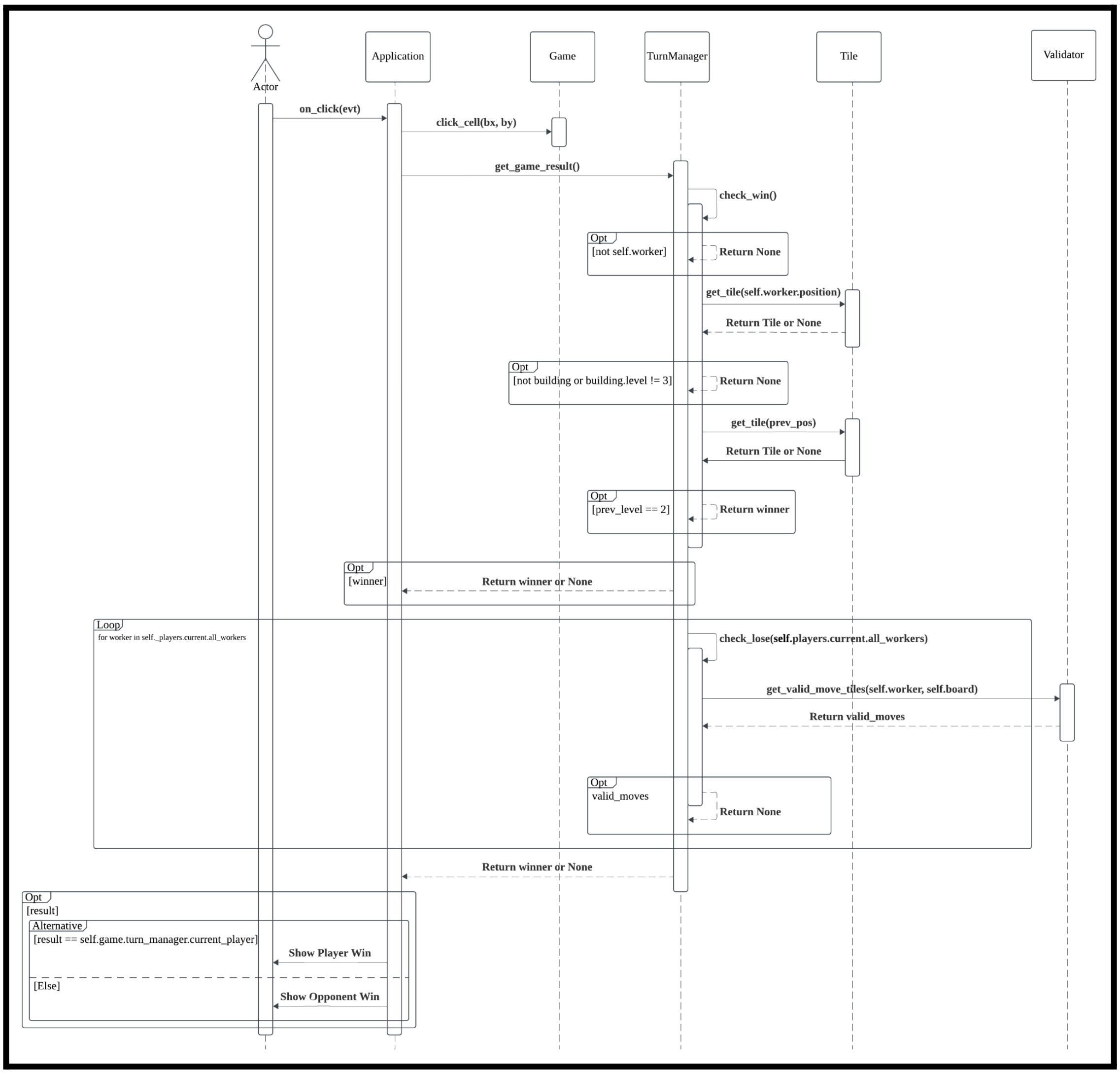
Functionality 3: Building



Functionality 4: Change of turn to the next player



Functionality 5: Winning the game



Previous draft versions can be seen inside our repository Sprint 2/uml_class_diagram/in_progress_uml/sequence_diagram. https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-groups/MA_Tuesday06pm_Team007/project/-/tree/main/Sprint%202/uml_class_diagram/in_progress_uml/sequence_diagram?ref_type=heads

Design Rationale

Key Classes

1. Board: Board class contains all operations which deal with coordinates and tile lookups. It provides methods which perform bounds checking, tile retrieval, empty‑tile gathering thus eliminating duplicate coordinate math throughout the program. Placing tile logic inside TurnManager and adding static helper functions would result in duplicate neighbor discovery and bounds tests that would create errors when modifying board shape or size.
   Alternative: Board setup can be implemented as a method in Game class, but it's better for Board to be its own class as the encapsulation allows both Board and Game class to focus on their own responsibilities.

2. Action: Action is an abstract class, which all actions implement. Move, Build, and god actions implements this action abstract. The move and build behaviors could have been embedded within TurnManager methods but this design approach would merge all validation and execution and sequencing logic into a single controller. A subclass of abstract Action contains each individual operation including moving, building and unique actions like Artemis's second move and Demeter's extra build. Each Action object uses the Command pattern to contain both validation through a validate method and execution through an execute method for the board. The process of adding new god powers through subclassing Action provides greater benefits than modifying a central controller yet requires a larger class hierarchy structure.
   Alternative: Action class can be implemented as move and build methods in a GameRule class to be imported into game class. However it limits extendability for implementation of future actions, as the turn structure becomes fixed and rigid to changes. As the extendability is limited, this would violate the open close principle.

3. TurnManager: TurnManager class orchestrates the game's turn flow. Rather than embedding turn-state, worker selection, and action sequencing inside Game or spreading it across UI callbacks, TurnManager owns the selected worker, and a Sequence object (our pipeline of Action steps, and Players). When a player clicks a tile, TurnManager drives the appropriate phase—selection, validation, execution, and advancement—without leaking UI concerns into the logic. Choosing a dedicated TurnManager class implementing as methods clarifies responsibilities (game state vs. UI wiring) and makes unit testing of turn flow straightforward, since turns can be simulated by calling handle_click with mocked tiles.
   Alternative: TurnManager could be implemented as part of Game class functionality. However this violates the Single Responsibility Principle, as Game takes care of game setup compared to TurnManager that cares of player actions for each turn and checking for win and lose conditions.

Key Relationships

1. The relationship between Player and Worker stands as the most essential one in our system. In our diagram, a player aggregates exactly two workers. The aggregation relationship proved better than loose association since Worker objects maintain essential dependencies on their Player owner through color and turn order history and previous-move records.
   Alternative: Player and Worker could be symbolized with composition relationship. However, Player and Worker can exist as separate entities as they have their own responsibilities. Player class keeps track of their god card and workers, Worker class keeps track of position on the board.

2. All Tile objects exist within the Board structure as its composition since the Board creates and controls all Tile objects and Tiles cannot exist independently from their Board. The tile collection of a board gets created automatically when the board initializes and gets destroyed when the board terminates. The strong ownership of tiles by boards proves logical since tiles become meaningless without board dimension and neighbor definitions and cross-board tile sharing would create state management complexities.
   Alternative: Originally we were thinking of displaying Board and Tile relationships as aggregation. But Tile being able to exist independently without context is illogical, as each tile needs to know their position on the board.

3. There is an association between TurnManager and Sequence. TurnManager calls on Sequence to help handle the turn. The default base turn is simple, each turn a player moves and builds. In the situation where a god card is involved, we will call on the Sequence to change the turn details. For example, a turn would now have an extra build when a player has the artemis god card.
   Alternative: We considered having the god card would directly affect the turn manager each round through an if/else. For example, if there was an Artemis god card, the code would check after moving a worker for if the player had a god effect that triggers. After that is resolved by turn manager, the game will still check at move if there’s a relevant god card. For expansion purposes, this might increase the number of if/else in the turn manager, which is not an ideal design.

Inheritance Decisions

The implementation of inheritance occurred only in the Action, Building, and GodCard hierarchies.

The Action abstract base class defines the validate and execute contracts that MoveAction and BuildAction along with ArtemisMoveAction and DemeterBuildAction subclass to change their specific behaviors. The inheritance structure enables polymorphism for

TurnManager to handle all steps uniformly through super().validate and super().execute method calls.
Alternative: We weighed the option of merging all move and build logic into one Action class but rejected it because flag-based conditional code would make the simple sequence of actions difficult to follow.

The Building class served as an abstract base to define has_dome, level and increase_level methods which Block and Dome classes implemented as concrete subclasses. Building inheritance allows easy extensions such as Atlas's ground-floor dome through subclassing Building while maintaining a common interface for all building objects.
Alternative: We could implement Building as a concrete class with methods to determine the building level and if the building has a dome or not which makes it no longer unbuildable and unmovable to. This implementation, while works for simple situations, could be difficult to change and extend if we try to implement or modify kinds of buildings.

We also introduced a GodCard hierarchy as an abstract base that stores the card's name and description and declares get_action_sequence(). Each concrete subclass—presently Artemis and Demeter—implements that single method to return the exact list of Action objects that realize its special power. Because TurnManager only ever asks a card for its action sequence, adding a new god now means writing one new subclass and nothing more: the rest of the engine remains unchanged.
Alternative: We tried implementing a GodCard concrete class with each god being an instance instead. However this design is hard to implement and is not flexible enough as it's hard to extend functionality to other god cards in the future.

Cardinality Choices

The model incorporates multiple cardinality decisions which follow the rules of Santorini.

Every Player in the game must possess precisely two Workers. The add_worker method in Player class triggers an exception when users attempt to add more than two workers while setup allows zero workers for random game placement. Before placement each player starts with zero workers while the game duration requires exactly two workers and after removal there are no workers. We decided to maintain the core game rules by not allowing workers in different quantities than two because it would deviate from the original design.

Another example is how Tile objects can either have a building on it, or not. This is seen by the 0..1 cardinality of their relationship. A tile with no building can be treated as level 0 and can be built on and moved to by workers. Levels beyond that are handled by the Building class handles the responsibility of interaction such as building level that affects movement rules.

Design Patterns

Our implementation leverages the Command pattern in its Action class hierarchy. Every Action contains both the move or build operation together with validation rules and an option flag. TurnManager remains independent from action details through this design which enables action reuse and extension as well as independent testing.

The Sequence class functions as a Pipeline through its TurnSequence implementation to manage a linear progression of Action commands that proceed when players perform clicks. The pattern presents a step-by-step sequence that defines turn operations along with optional steps which allows easy insertion of god power execution. The UI shows the current phase directly (“Move,” “Build,” or “Skip Action”) while the skip button becomes available or disabled according to the current phase.

The combined selection of design choices produces a system where grid management, action logic, turn flow and god powers remain within distinct modular areas. The design becomes simple to expand with new gods by adding new Action subclasses or pipeline steps while remaining easy to maintain because each class handles one specific responsibility.

Executable

Santorini.exe is a single-file, self-contained desktop application that launches our Santorini board-game prototype. It is built with PyInstaller in one-file mode, so the binary already contains:

- The Python 3.10 runtime
- all project modules
- Tk/Tcl libraries for Tkinter
- The only runtime asset – assets/worker.png

No IDE, package manager, or separate Python installation is required.

The executable Santorini.exe has been fully built and verified on Windows 11 (x64). Because no team member has access to Apple hardware, the application has not been tested on macOS

How to run the pre-built executable:

1. Unzip the executable submission archive.
2. Double-click Santorini.exe
3. The main menu appears; click Start New Game to begin.

How to recreate the executable from source code:

1. Go to the prototype folder (e.g., PS> cd "C:\...\project\Sprint 2\prototype")
2. Install PyInstaller
    - pip install pyinstaller
3. Build the one-file executable
    - pyinstaller .\application.py --name Santorini --onefile --windowed --add-data "assets*;assets"

    - Flags:
        - --onefile, resulting executable is only one file
        - --windowed, suppress the console window on GUI launch
        - --add-data, bundle everything in the assets directory (contains the worker icon)

Testing

Test ID	Description	Steps taken	Expected outcome	Actual outcome	Expected behavior?
1	Exit button	1) Open application 2) Click exit button	Application close	Application close	Yes
2	Set up players	1) Open application 2) Start game 3) Enter players as details as P1 Red P2 Blue	Visually, player 1 is red and player 2 is blue	Visually, player 1 is red and player 2 is blue	Yes
3	Play a round	1) Open application 2) Start game 3) Enter players as details as P1 Red P2 Blue 4) Take P1 turn and select a worker 5) Have P1 move, build and use their god ability 6) Take P2 turn and select a worker 7) have P2 move, build and use their god ability	Both player takes their turns fine	Both player takes their turns fine	Yes
4	Invalid movement (dome)	1) Build a dome on a level 3 building 2) Get a worker up to the 2nd level 3) Try to click on domed building during movement	Worker is unable to move	Worker is unable to move	Yes
5	Invalid movement (worker)	1) Try to move out of bounds 2) Try to move too far 3) Try to jump to the 3rd floor from no building	Worker is unable to move	Worker is unable to move	Yes

6	Invalid build (dome)	1) Build a dome on a level 3 building 2) Try to build on the dome	Unable to build	Unable to build	Yes
7	Invalid build (too far)	1) Try to build more than 1 square away from the worker	Unable to build	Unable to build	Yes
8	God card skip	1) Play the game till a god card activates 2) Skip the ability	Skip the ability	Skip the ability	Yes
9	Wrong worker clicked	1) Try to click on the opposing player's workers during your turn	Unable to take control of opposing player's worker	Unable to take control of opposing player's worker	Yes
10	Valid movement (down from high level)	1) Get worker to 2nd or 1st floor 2) Hop off the building	Capable of jumping down	Capable of jumping down	Yes
11	Artemis card movement	1) Move with a worker 2) Move with the same worker	Moves a second time	Moves a second time	Yes
12	Demeter card build	1) Build with a worker 2) Build with the same worker	Builds a second time	Builds a second time	Yes
13	Artemis card movement incorrect movement	1) Move with a worker 2) Try to move back to the original space	Unable to move back	Unable to move back	Yes
14	Demeter card build incorrect build	1) Build with a worker 2) Try to select the other worker and build using that worker	Unable to build	Unable to build	Yes
15	Unusual characters	1) Try the follow string for name " ;'s"	White space is trimmed, no crash	White space is trimmed, no crash	Yes

16	Win condition	1) Player moves a worker up to a undome level 3 building	Game end and winner is annouced	Game end and winner is annouced	Yes
17	Lose condition	1) Player move workers into the corner 2) Play the game and ensure workers cannot move anymore by building around them	Game end and winner is annouced	Game end and winner is annouced	Yes

Video Demonstration File Format: MP4