SPRINT 2 DELIVERABLES

FIT3077 - CL_Tuesday04pm_Team063 - Coding With Goblins

Table of contents

1. Object-Oriented Design and Design Rationales

1.1. Create a complete UML Diagram

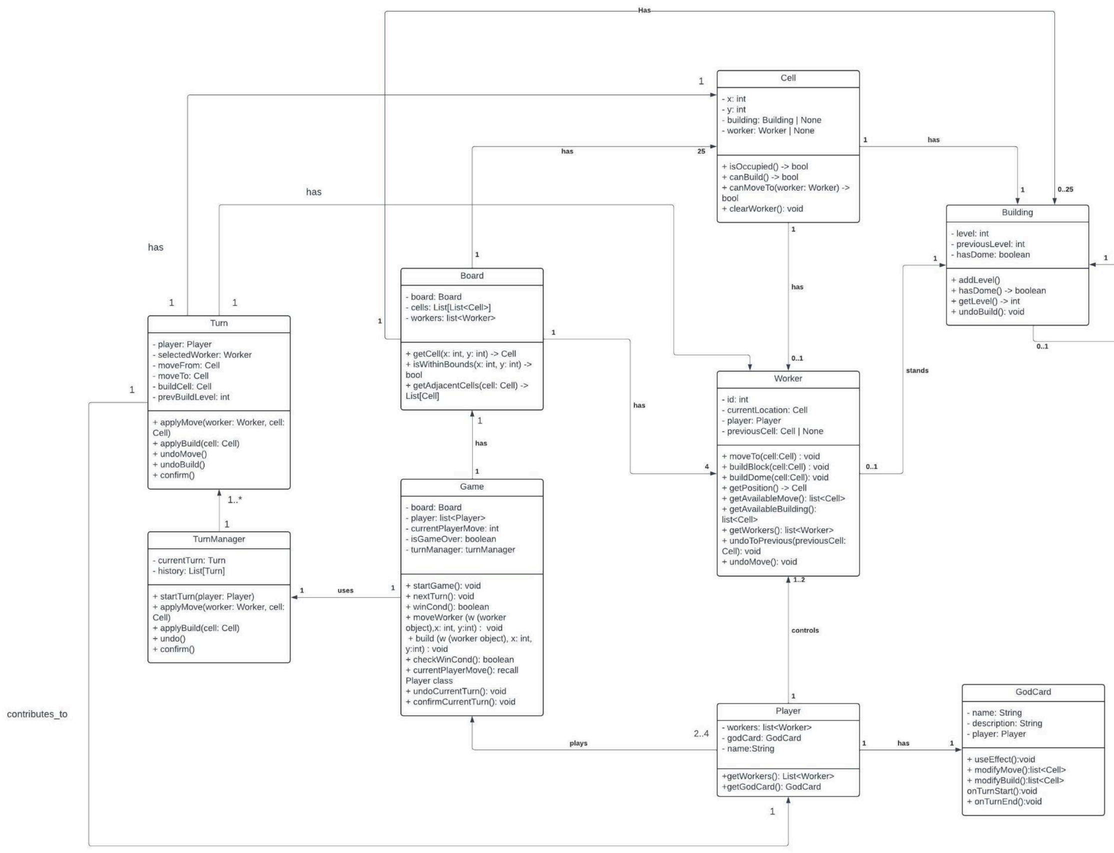


Figure 1: UML version 1.0

This is the initial UML diagram created at the beginning of Sprint 2. It reflects the foundational structure of our system as it evolved from the domain model developed during Sprint 1. At this stage, we began to define the key classes that would form the backbone of our application and explored their relationships and responsibilities.

Our focus was on identifying and implementing the core functionalities that would be essential for each class. This included not only basic operations but also utility methods that we anticipated would support the overall system workflow. We made decisions based on both current requirements and possible future needs to maintain flexibility.

In addition to the methods, we added all the basic attributes relevant to each class. These attributes were chosen based on the roles of the classes in the system and the data they needed to encapsulate. While this version is an early draft, it provided a clear visual representation of our software architecture and helped guide our initial coding efforts.

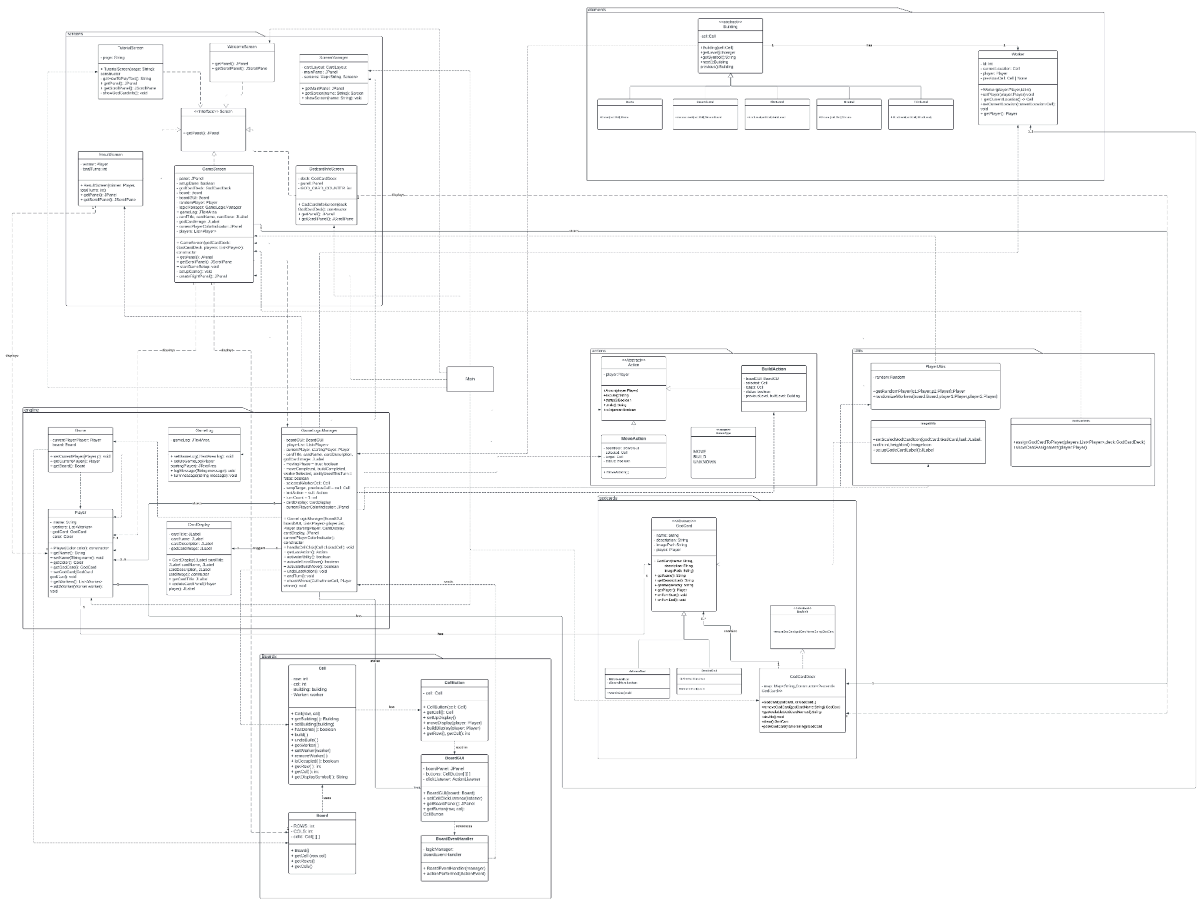


Figure 2: UML version 4.0

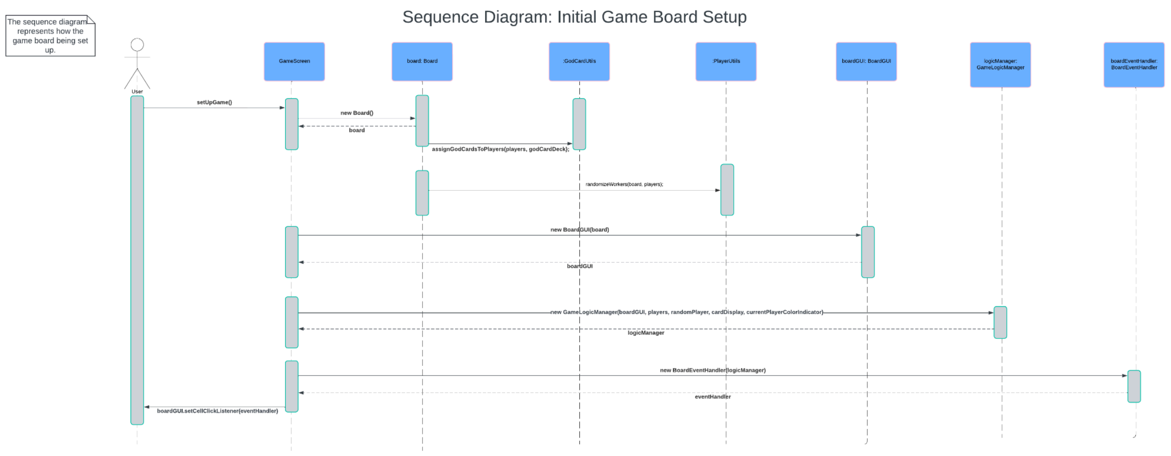
This final UML diagram represents the completed design of our Sprint 2 prototype. It includes refined class structures, expanded relationships, and integration with new components such as GameLogicManager, GodCard, and UI classes like BoardGUI and CardDisplay. Complex behaviors, phase control, and god power handling are now clearly modeled, supporting the finalized architecture and game flow implementation.

Please refer to the team Git repository for the full version of UML class diagrams and Sequence diagrams within **docs/Sprint 2/images/** route.

1.2. Basic game functionalities

1.2.1. Initial board

Sequence Diagram:

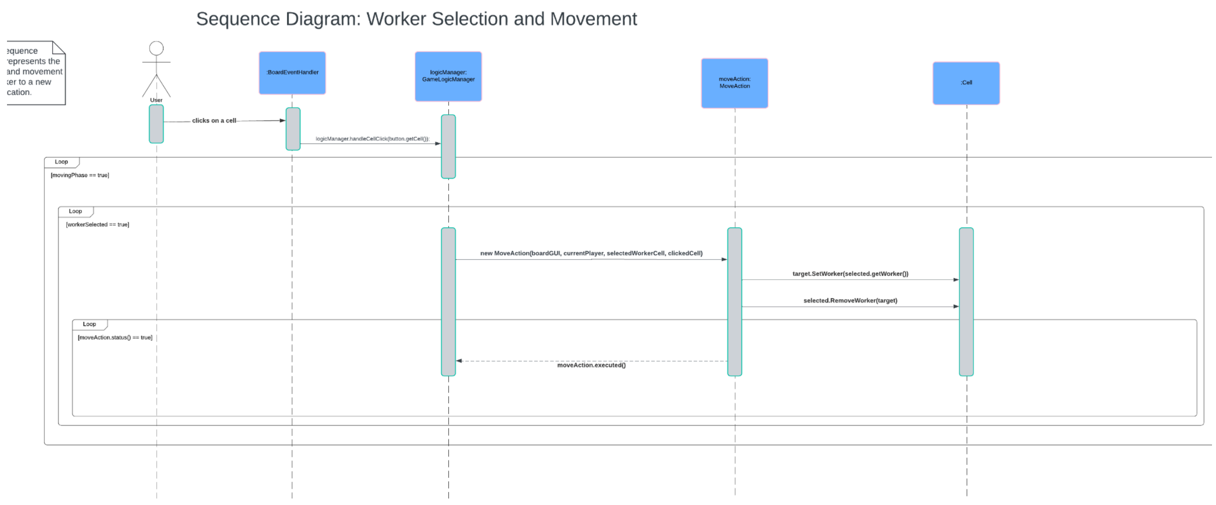

Sequence Diagram: Initial Game Board Setup
The sequence diagram represents how the game board being set up.
User
GameScreen
board: Board
:GodCardUtils
:PlayerUtils
boardGUI: BoardGUI
logicManager: GameLogicManager
boardEventHandler: BoardEventHandler
setUpGame()
new Board()
board
assignGodCardsToPlayers(players, godCardDeck);
randomizeWorkers(board, players);
new BoardGUI(board)
boardGUI
new GameLogicManager(boardGUI, players, randomPlayer, cardDisplay, currentPlayerColorIndicator)
logicManager
new BoardEventHandler(logicManager)
eventHandler
boardGUI.setCellClickListener(eventHandler)

Summary:

To display the initial board (5x5 board), we had the Board Class that stores the attributes of the board (rows and columns). To display the board, we will use the game screen class which is used to display the game to create a new board object, after that we will pass that board to BoardGUI to display it. For each row and column, it will create a new cell to display the ground of the board which the player can interact with (CellButon).

1.2.2. Selection and movement of worker

Sequence Diagram:

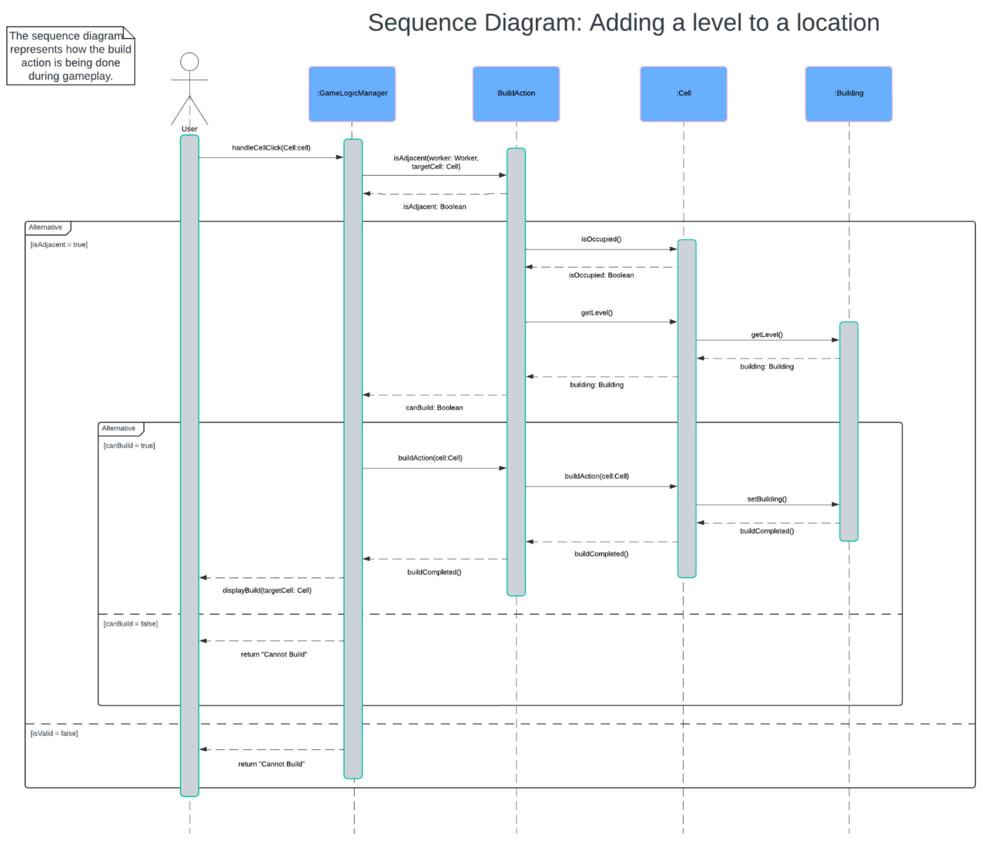

Sequence Diagram: Worker Selection and Movement
equence
represents the
and movement
ker to a new
cation.
User
:BoardEventHandler
logicManager: GameLogicManager
moveAction: MoveAction
:Cell
clicks on a cell
logicManager.handleCellClick(button.getCell());
Loop
[movingPhase == true]
Loop
[workerSelected == true]
new MoveAction(boardGUI, currentPlayer, selectedWorkerCell, clickedCell)
target.SetWorker(selected.getWorker())
selected.RemoveWorker(target)
Loop
[moveAction.status() == true]
moveAction.executed()

Summary:

We handle the Selecting and Moving action in the GameLogicManager class. So when we click on the Cell that has the player on it, it will check if that Cell has the correct player turn's worker. If yes, the player will choose the available move and it will call the MoveAction class (which extends from the Action class) to perform the move.

1.2.3. Building action of worker

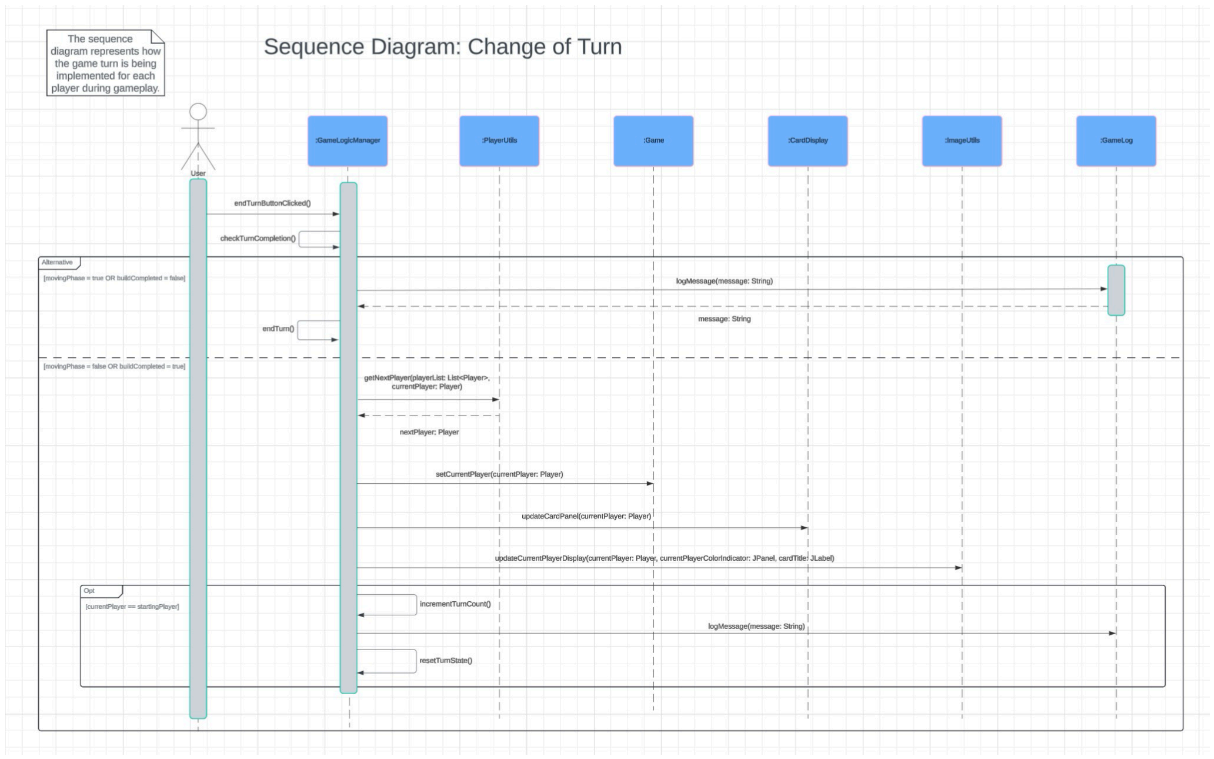
Sequence Diagram:



Summary: Same as the MoveAction, Building action will also be handled in the GameLogicManager.java. After the player performs the move, the player needs to pick that worker again, if it is not the worker that performs the move action, it will alert the player to re-pick their worker to perform the BuildingAction. it will execute the BuildingAction (which extend from the Action class)

1.2.4. Switching game turn

Sequence Diagram:


The sequence diagram represents how the game turn is being implemented for each player during gameplay.
Sequence Diagram: Change of Turn
User
:GameLogicManager
:PlayerUtils
:Game
:CardDisplay
:ImageUtils
:GameLog
endTurnButtonClicked()
checkTurnCompletion()
Alternative
[movingPhase = true OR buildCompleted = false]
logMessage(message: String)
message: String
endTurn()
[movingPhase = false OR buildCompleted = true]
getNextPlayer(playerList: List<Player>, currentPlayer: Player)
nextPlayer: Player
setCurrentPlayer(currentPlayer: Player)
updateCardPanel(currentPlayer: Player)
updateCurrentPlayerDisplay(currentPlayer: Player, currentPlayerColorIndicator: JPanel, cardTitle: JLabel)
Opt
[currentPlayer == startingPlayer]
incrementTurnCount()
logMessage(message: String)
resetTurnState()

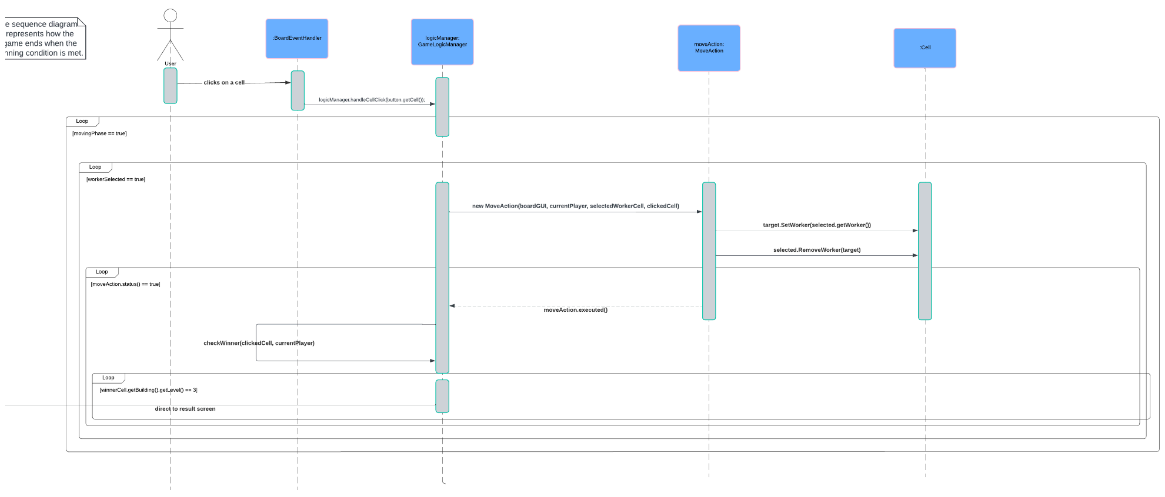
Summary:

To switch the player turn. The current player needs to click the end turn button after they complete their turn (finished build action). The “End Turn” button will call the end turn function in the GameLogicManager to check whether the player has completed the movement and building action yet. If the current player has already completed, it will switch to the next player and reset the phrase of the game.

1.2.5. Winning condition

Sequence Diagram:

Sequence Diagram: Winning The Game


e sequence diagram represents how the jame ends when the nning condition is met.
User
:BoardEventHandler
logicManager: GameLogicManager
moveAction: MoveAction
:Cell
clicks on a cell
logicManager.handleCellClick(button.getCell());
Loop
[movingPhase == true]
Loop
[workerSelected == true]
new MoveAction(boardGUI, currentPlayer, selectedWorkerCell, clickedCell)
target.SetWorker(selected.getWorker())
selected.RemoveWorker(target)
Loop
[moveAction.status() == true]
moveAction.executed()
checkWinner(clickedCell, currentPlayer)
Loop
[winnerCell.getBuilding().getLevel() == 3]
direct to result screen

Summary:

For the winning condition, we implemented the checkWinner function in the GameLogicManager class. That function will check the current location of the player's worker. If the worker is at the level 3 of the building, it will register the ResultScreen to display the winner. Each turn after the player performs the move, it will call the checkWinner function.

1.3. Design Rationales (explaining the 'Whys')

1.3.1. Board

In our Sprint 2 implementation of the Santorini game, a key architectural focus was the design of the Board class and related components. This design is critical as the board represents the game's central data structure, coordinating spatial logic, tile updates, and interaction with the GUI and game logic. The design reflects careful consideration of object-oriented principles, modularity, and extensibility.

Key Classes and Justification:

1. Board:

The Board class encapsulates the 5x5 grid structure using a two-dimensional array of Cell objects (Cell[ ][ ] grid). Each tile on the board is a Cell that manages its own state (level, dome, and occupancy). This class is responsible for initializing the grid, providing access to specific cells, and validating board positions. It was necessary to define Board as a standalone class rather than a method set, as it maintains and controls persistent state across the game. It encapsulates core functionality that is reused across multiple game phases (setup, movement, building), and its lifecycle aligns with the game session.

2. Cell:

The Cell class models the individual tiles on the board. Each instance tracks its level, dome presence, and the occupying Worker, if any. Methods like buildLevel(), placeDome(), and isOccupied() encapsulate cell-specific logic. Without a dedicated class, these properties would have to be distributed across procedural methods, leading to high coupling and reduced readability. Creating Cell as an object allows for encapsulation, data protection, and modular tile manipulation.

3. BoardGUI:

BoardGUI is responsible for rendering the visual representation of the board using Java Swing. It contains a matrix of CellButton components that reflect the underlying Cell state. This class separates visual concerns from game logic, supporting the principle of separation of concerns. It could not be reduced to methods alone, as it manages a persistent stateful component tree tied to user interaction.

Key Relationships and Their Justifications:

1. Board to Cell:

The Board aggregates Cell objects in a 2D array. This is modeled as an aggregation because cells exist independently of the board logic and can be created, replaced, or reset without destroying the board itself. This is different from composition, where the lifecycle of components is tightly bound.

2. Cell to Worker:

Each Cell may hold a reference to a Worker. This is a simple association, not an aggregation or composition, as a worker can move between cells and is not owned by any specific cell. The reference exists to allow occupancy checks and interactions.

3. BoardGUI to CellButton:

BoardGUI composes CellButton components to display the board. These buttons are created within BoardGUI and do not exist outside of it. Destroying the GUI destroys the buttons, fulfilling the criteria for composition.

Inheritance Decisions:

We did not use inheritance for the Board class or Cell because there was no meaningful generalization or subtype behavior to extract. Both classes have well-defined, single responsibilities and do not share common behavior with other parts of the system. Inheritance was deliberately avoided to maintain clarity and avoid premature generalization.

However, GUI components like CellButton inherit from JButton, which is justified as they extend GUI behavior while retaining standard Java Swing functionality.

Cardinality Justifications:

Board → Cell: 1 → 25 (Fixed Size Grid) Each Board contains exactly 25 Cell objects (5x5), instantiated at game start. The 1..1 to 25..25 cardinality ensures a fully populated grid with no null tiles.

Cell → Worker: 0..1 Each Cell may or may not have a Worker. The 0..1 cardinality reflects this transient occupancy, supporting the rules of movement and tile control.

Design Pattern Decisions:

We did not explicitly implement a design pattern in the board logic due to the simplicity and specificity of the domain. However, we considered several:

- **Singleton**: Not used because the game does not require a globally shared board instance; multiple boards (e.g., for testing) may be needed.
- **Factory Method**: Considered for creating Cell objects, but deemed unnecessary due to the simplicity of construction.
- **Observer**: Not yet implemented, but may be added in Sprint 3 to update UI components in response to board state changes.

Summary

The design of the Board component is central to supporting the core gameplay of Santorini. By using a clean, object-oriented model that separates tile state (Cell) from rendering logic (BoardGUI), we ensure strong cohesion and low coupling. The relationships between board elements are clearly defined and follow best practices in aggregation and association. The current implementation provides a flexible foundation for gameplay, and its modularity allows for scalable additions such as dynamic board resizing or special tile mechanics in future sprints.

1.3.2. GameLogicManager

The GameLogicManager class plays a central role in enforcing the rules and flow of the Santorini game. It coordinates the entire turn cycle, including selecting and moving workers, constructing buildings, switching turns, checking win conditions, and interacting with GodCard abilities. Its responsibilities span both gameplay mechanics and certain UI behaviors, making it a crucial part of the controller logic in our architecture. Instead of embedding game rules across multiple classes like Player, Worker, or Board, we created GameLogicManager to centralize these responsibilities and ensure clear modularity.

Key Classes and Justification

1. GameLogicManager:

The GameLogicManager encapsulates all gameplay phases including move and build validation, turn switching, and GodCard effect activation. It manages the state of the current player, the selected worker, the game board (BoardGUI), and coordinates between different phases via flags such as movingPhase, workerSelected, moveCompleted, and buildCompleted. This functionality is exposed through methods such as handleCellClick(), activateAbility(), undoLastAction(), and endTurn(). Without a dedicated class, these interconnected operations would be scattered across unrelated components, breaking cohesion and making the game difficult to manage or extend. This class also supports undo functionality through lastAction, referencing MoveAction or BuildAction instances.

2. TurnManager:

Although not explicitly a class in our code, the turn-switching logic is embedded within the endTurn() method of GameLogicManager, managing current and starting players. We plan to abstract this logic into a dedicated TurnManager class in a future sprint for greater SRP compliance.

3. CardDisplay:

GameLogicManager interacts with the CardDisplay class to visually update the active GodCard and the player color indicator. This connection reinforces separation between game mechanics and UI updates, aligning with MVC principles.

Key Relationships and Their Justifications

1. GameLogicManager to BoardGUI:

GameLogicManager accesses BoardGUI to manipulate the displayed board state. This is a temporary association, not aggregation or composition, as the board is independently instantiated and passed in.

2. GameLogicManager to Player:

The manager tracks the current and starting players. These references allow the class to determine valid actions based on ownership, manage player turns, and check win conditions.

3. GameLogicManager to GodCard:

Via the activateAbility() method, the class delegates special logic execution to the player's GodCard object. This relationship enables flexible strategy injection without tightly coupling card behavior to the game manager.

Inheritance Decisions:

We intentionally avoided using inheritance in the GameLogicManager as all core game logic is tied to one unified set of behavior in Sprint 2. However, method overriding through GodCard's useEffect(GameLogicManager) supports polymorphism in a controlled way. This setup prepares us for Sprint 3, where god powers will introduce rule variations. In contrast, components like MoveAction and BuildAction could potentially share a common interface or superclass in the future for better extensibility.

Cardinality Justifications:

GameLogicManager to Player: 1 → 2 Exactly two players are managed in every game. The manager references both but interacts with only one at a time via the currentPlayer attribute.

GameLogicManager to Cell (0..1 selectedWorkerCell) The manager keeps track of a selected worker's cell during move and build phases. This reference is null when no worker is selected.

GameLogicManager to Action (0..1 lastAction) The manager stores only the last completed action to support undo functionality. This reference is transient and cleared at the end of each turn.

Design Pattern Decisions:

- **Facade Pattern**: The GameLogicManager acts as a high-level interface that coordinates lower-level operations from MoveAction, BuildAction, BoardGUI, and Player. This encapsulation simplifies interactions for other system components.
- **Command Pattern**: While not explicitly implemented, the structure of lastAction as an instance of either MoveAction or BuildAction reflects a command-like interface. Each action encapsulates its own execution and undo logic.
- **Strategy Pattern (Planned)**: GodCard behavior is handled via delegation. The useEffect() method accepts GameLogicManager as a context, allowing the injected strategy to manipulate the game state dynamically. This will become a full strategy pattern in Sprint 3.

Design patterns such as **Observer** (for real-time GUI updates) and **Singleton** (for shared managers) were avoided in Sprint 2 to prioritize testability and simplicity.

Summary:

The GameLogicManager is the control center of our game logic. It manages player input, validates moves, enforces building rules, handles turn sequencing, and integrates god power abilities. Its cohesive design keeps the system modular and scalable while ensuring consistent rule enforcement. As we expand the game in future sprints, this manager provides a strong foundation for injecting new rule systems and UI feedback mechanisms.

1.3.3. Player

Modeling the Player class was essential in maintaining game flow and organizing game state. Our design evolution reinforced the importance of encapsulating player identity, game pieces, and unique abilities through a dedicated class.

Key Classes and Justification:

1. Player:

The Player class encapsulates a participant's identity (name, color), the list of Worker instances they control, and their assigned GodCard. It is responsible for managing player-specific state while delegating game logic to the GameLogicManager. Creating this as a class rather than scattering attributes avoids data duplication and aligns with object-oriented encapsulation. The player's state evolves through the game, justifying a persistent class instance.

2. Worker:

Each Worker is tied to a Player and represents an interactive game piece. It stores position and movement history. Creating this as a class, rather than embedding data into Player, supports modular logic such as validating movements and checking victory conditions.

3. GodCard:

This class encapsulates special abilities. While not central to Sprint 2, it is designed to allow future extension via polymorphism. Each Player holds one GodCard, reinforcing the one-to-one association. Without a class, implementing abilities would require complex conditionals in unrelated code.

Key Relationships and Their Justifications:

1. Player to Worker:

A Player aggregates two Worker objects. The workers exist independently but are grouped under a player during the game. They can be moved, reset, or removed without affecting the player's identity.

2. Player to GodCard:

Each Player has a GodCard, modeled as an aggregation. The GodCard instance provides abilities specific to that player, and though it can exist independently, it is logically owned by the player.

3. Worker to Cell:

A Worker moves between Cell instances, creating a temporary association. The Worker holds coordinates or a direct reference to its current cell. This relationship is not an ownership, reflecting the dynamic movement rules of the game.

Inheritance Decisions:

We chose not to use inheritance for the Player class, as our game does not differentiate player types (e.g., AI vs human) at this stage. The behavior and structure are identical for both participants. Instead, potential variation is deferred to composition (via GodCard) and delegation (to logic managers).

The GodCard class, however, is designed for future subclassing to support specific card behaviors via method overriding.

Cardinality Justifications:

Player → Worker: 1 → 2 Each Player controls exactly two workers throughout the game. This cardinality is fixed based on the official rules of Santorini.

Player → GodCard: 1 → 1 Each player has one assigned GodCard. It cannot be null once the game starts, and each player has exactly one ability card in play.

Design Pattern Decisions:

We did not apply full design patterns in this section but reviewed several options:

- **Strategy Pattern**: Considered for GodCard behavior, and likely to be implemented in Sprint 3.
- **State Pattern**: Not applied, as player states (e.g., active, waiting) are managed via game logic rather than polymorphic transitions.

- **Prototype Pattern**: Not used, as each Player, Worker, and GodCard is created individually and not cloned.

Summary:

Overall, our use of dedicated classes and clean relationships allows the system to be extended with minimal refactoring, aligning with object-oriented best practices and the needs of Santorini gameplay.

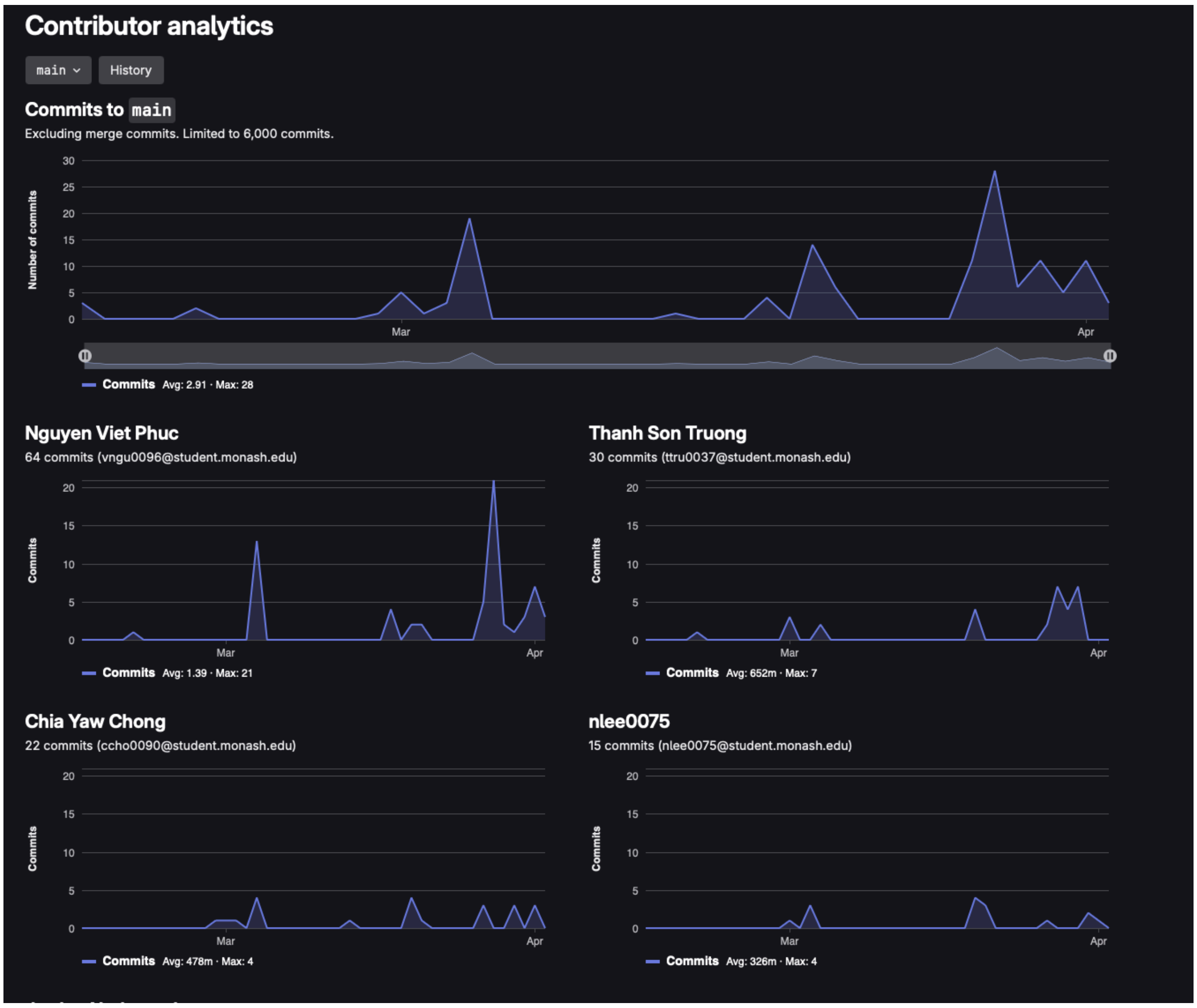
2. Tech-based Work-in-Progress (working software prototype)

Our Santorini prototype was developed in Java using Swing library and follows the architecture proposed in Sprint 1. The system includes all five required core functionalities:

- Board Setup: A dynamic 5x5 board is generated using Board and BoardGUI, with each tile represented by a Cell object
- Worker Placement: Players can place their workers on unoccupied tiles, validated through the Worker class
- Movement Logic: Implemented via MoveAction, enforcing adjacency, elevation limits and dome blocking
- Building Construction: After moving, players can build using BuildAction, domes cap buildings at level 3
- Win Condition: Detected when a worker moves onto a third-level tile, managed through GameLogicManager

Frequent GIT commits were made to track progress and contributions. The code follows the Google Java Style Guide for consistency and readability.

3. Contributor Analytics



4. Executable and Platform Details

The Santori game is a Java desktop application that runs on Windows, macOS, and Linux with JDK 24 or higher. It can be executed from the terminal or using an IDE (e.g., IntelliJ, Eclipse).

For full setup, build, and run instructions, please refer to the README file in the Git:

https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-groups/CL_Tuesday04pm_Team063/project/-/blob/main/README.md?ref_type=heads