FIT3077 - Sprint 2 Submission

Software Engineering: Architecture and Design
Semester 1, 2025

Moodle Team Name: *CL_Tuesday06pm_Team013*

https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-groups/CL_Tuesday06pm_Team013/project

Team: *SoftTorinis*

Class Diagram

Basic Game Functionalities

Initial Board Set-up

The 5x5 grid is initialized in the Board class, to provide increased flexibility for later iterations, the number of rows and columns were added as attributes that can later be configured based on custom implementation.

The Game class contains the initialised game method that allocates the workers to random spots and the players with random God cards.

Selection and Movement

The worker selection process occurs within the Player class, each player instant contains a list of workers as an attribute that has a select worker method to be used in each turn process in the GameManager.

The movement process and validation occurs in the getAvailableBuildCells function in the Board class, this function utilizes a getAdjacentCells function also in the Board class, that uses the canMoveTo function in the Cell class.

More clearly, the Board class finds the adjacent cells to the worker cell–this takes in a distance parameter to improve extensibility to allow for further movement and building for workers–and for each cell in that list, it calls a function in the cell class to check if it can be moved to.[1]

Level Addition and Validation

The level addition occurs in the Tower class, this class contains a level integer attribute that is changed once the update tower level method is used.[2]

The validity logic for this is similar to that of movement validity, the Board class containing a getAdjacentCells method is used for both moving and building, the difference is within the Cell class methods where for building validity the isAvailableForBuild method is used.

The initial retrieval of getting adjacent cells initially follows the *Don't Repeat Yourself* principle.

Change of Turn

[1] These design choices are further discussed in the design rationale.
[2] The usage of 'changing' the level attribute and 'updating' tower level is to allow for building removal features as per extensibility principle.

Winning the Game

A GameStatus enumeration class is used with ongoing, win, and draw flags. These are checked and updated at each turn via the GameManager class.

If a turn ends with the win or draw flags being raised at the end of the turn, the GameManager class provides the winning player to the Game class.

UML Sequence Diagram

Design Rationale

Key Classes

Board

The Board class is a key component in the design as it encapsulates all methods relating to multiple cells. This method of separating the treatment of a singular cell to the Cell class, and a group of Cells to the Board class, abstracts the board into a singular entity.

The Board class is also one of the only classes that is instantiated once, emphasising the singular entity property mentioned and causes the Board to be a representation of the game environment as a whole.

This then consequently also makes it one of the only classes (GameManager follows this as well) that fully follows the singleton design principle which is that a class is to only be instantiated once, other classes cannot follow this design principle due to the nature of the game (multiple players, cells, workers etc).

The Board class was not considered as a method as it contains the grid containing all the cells–these cells are constantly changing in the duration of the game–while also containing methods that act upon this grid (e.g getting adjacent cells). The combination of attributes and methods acting upon those attributes makes it more suitable to be constructed in a class rather than a function.

While the Board class contains multiple responsibilities of storing data, and using methods that act upon them, it also has multiple methods as well (such as getting a cell from a coordinate, and available move or build cells).

This does not break the single responsibility principle as all the mentioned attributes and methods are similar in nature; they deal with the manipulation of multiple cells as a collective. This in return makes it a very suitable class, as all the mentioned points are key components of when it is best to use a class; encapsulation, abstraction, and modelling real world entities into a class object.

GameManager

The GameManager class is key for controlling the flow of the game, this includes methods that start and end the game, as well as multiple turn functions to start and finish a turn. These methods delegate the decision making and validation to other classes like the Board and Player classes, and hence acts as an integral central system for organizing all the classes to work together to progress through the game cycle.

This crucially also maintains the GameStatus enumeration and keeps track of the current player, which are both used for the Game class that initializes the game and assigns the winner.

The GameManager is thus a great example of the separation of concerns design principle, as that is its main role; it separates the game flow concerns from other classes.

This process would be near impossible in a single function as it needs to maintain the different states mentioned, along with keeping track and updating the changes.

The GameManager class is also only instantiated once, making it also follow the singleton design pattern which makes it even more suitable for a class rather than a method.

GodCard and Action

These two classes are similar in nature as they are the main two abstract classes, and are hence reflected simultaneously in this section.

Firstly the GodCard abstract class contains a required function called activatePower, that is all Gods added into the game must have this function, but can carry their own implementation.

This is an effective method of allowing extension for a number of God cards that are not yet implemented.

This is a key class as it is one of the core functionalities of the game, it handles the functionality for all God cards, without having the requirement of knowing how each God card works; this is the core idea of an abstract class, and hence it is clear why it is a class and not a function.

If the method approach was taken, it would violate many design principles, namely the open and closed principle and DRY. As for each new God card created, there would need to be manual addition and configuration to the code each time a new card requires implementation.

This describes why the different God cards need a class, but not the significance of the claimed key class here which is an abstract GodClass; its significance is how the integration between different classes (such as Player utilising the God classes) would look like without a central abstract class, for each new God Card it would need to be added into the existing code as well, also violating the open and closed principle, and reducing maintainability to a great extent.

Moreover, having a key abstract class such as the GodCard ensures that all new God Cards have a distinct, clear function that is required, significantly improving maintainability as it prevents errors such as different naming conventions.

The rationale behind the GodCard class also applies to the Action class, although less significant in the latter due to less possible actions other than move and build being created compared to different God card classes. There is admittedly less certainty on different types of actions that can be performed in the future, but having it as an abstract class allows for their possibility with clear integration to the rest of the code.

Key Relationships

Board to Cell

This is a key relationship in the class diagram as it is integral to the whole game design, it is a composition relationship that highlights the dependence of the cells on the board, as in a cell cannot exist without a board.

The nature of providing ownership of all cells to the Board class is the core mechanism for allowing key functionalities in the flow of the game; the methods within the Board class utilize the cells that they own.

This relationship is also representative of a clear real world entity modelling system, a square (cell) on a physical Santorini board cannot exist on its own, and a Santorini board is not a board without the cells, while the board itself has its own attributes (e.g dimensions of a physical board) and methods (e.g telling someone a coordinate on the board to identify a single cell), while each cell in the board also has its own attributes and functionalities.

The relationship is a great example of following the composition over inheritance design principle, increasing flexibility, encapsulation and separation. The relationship advantages and importance can be compared to that of a Tree class and its NodeCells, an important relationship that allows for significant improvement of design.

Player to Worker

The Player to Worker relationship is a composition relationship that separates the player from their set of workers, and allows for multiple workers to have their own functionality and attributes.

The reasoning is similar to that mentioned previously; it follows composition over inheritance principle, separates functionality, and models real world principles that apply perfectly to this design.

More specifically is the constant usage of this relationship throughout the game, the separation is crucial for the flow of the game as the player can interact on a higher level with different classes, and the worker is concerned with more lower level methods and functionality.

It is a composition not aggregation to highlight the dependency of the worker on the player; a worker instance cannot exist without its player, further reflecting the real world aspect of a physical Santorini game.

GameManager to Game

This is an association relationship between the GameManager class and the Game class, it represents an important interaction that is integral to the flow of the game. This specific interaction allows for the Game to act as a data storage class, where the Game can be instantiated and utilise the GameManager for the operations and functionalities.

The association relationship models this interaction, and was selected as opposed to other possible interactions such as aggregation and composition simply because there is no ownership or multiple instances of any of the two classes. The relationship yet again highlights separation of concerns and maintainability.

This relationship also allows for multiple instances of the Game to be made, meaning multiple games can run at the same time, while not strictly relevant to this Santorini game development, this design makes it a lot easier to scale–for instance hosting a santorini game server–in contrast to having one class that controls all aspects. While not absolutely relevant, and not the integral reason for this relationship, it is good practice and follows design principles such as scalability principle.

Inheritance Decisions

Design principles such as “composition over inheritance” generally discourage the usage of inheritance.

However, its usage was believed to be justified in two cases of our game architecture, namely the GodCard and Action class, both are abstract classes that allow for subclasses to inherit their properties. In both instances, they are a clear usage of utilising polymorphism and following the open/closed principle.[3]

Cardinalities

Board to Cell

The cardinality is 1 for the Board and 25 for the Cell. This is due to each cell being restricted to just one board, while the board can contain multiple cells. It is set to 25 due to this implementation of the Santorini game being a 5x5 grid.

Worker to Player

The cardinality is 2 for the worker and 1 for the player. This represents a player containing two workers, while a worker is dependent and restricted to just one player.

[3] More detailed justification of why inheritance was used with those two classes was provided in the “GodCard and Action” portion of the document, while justification of avoiding inheritance is also present in the document (“Key relationships”, and “Key classes”)

Design Patterns

There are a number of design patterns within our design[4], the singleton design pattern is seen in the GameManager, Game and Board classes. This was utilised as multiple instances of these key classes can introduce a lot of complexity and increase fragility of the whole game. A single instance of these classes create a more error prone game design, thus allowing extensibility with significantly less overhead.

The command pattern is also present in the design, through the Action class. This is command execution that acts as a stand-alone object, leading to abstraction of the action execution, and thus increases simplicity when used as a parameter in other parts of the code.

The strategy pattern is also clear in the way the God powers are utilised by the player, the GodCard abstract class acts as the strategy interface, while the God Cards (e.g Demetres, Artemis) act as the concrete strategy, while the player acts as the context, unaware of exact implementation yet utilising their functionalities. This leads to extensibility (addition of God cards), and the abstraction of the God powers significantly reduces overall complexity.

The GameManager class also showcases the Facade pattern, this class acts as an access point to the rest of the game (‘Complex Subsystem’), this in return further simplifies the flow of the game, as a lot of classes (Player/Cell/Board etc) act together within a system, that the client (Game) is unaware of as Game interacts with GameManager.

[4] A lot of design patterns and naming conventions are used, this section uses ‘Refactoring Guru’ as a resource for naming, and a general reference.

Video Demonstration

Commit Analytics

AI Acknowledgment

References

Refactoring.Guru (2014-2015) *Design patterns*. https://refactoring.guru/design-patterns.