Documentation and Design for Constructor

Jerry Zhang, Jack Zhao, Jeffery Xu

I. Introduction

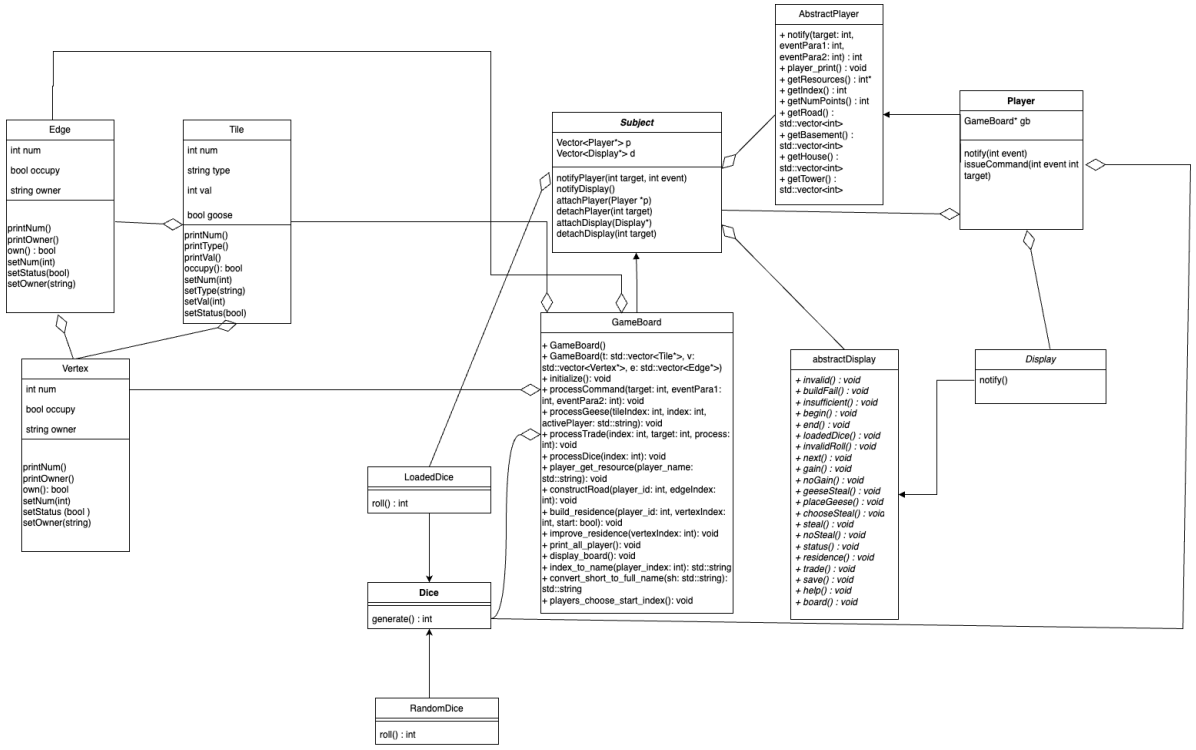
"Constructor" is a board game implemented as part of a programming project, and it's designed as a variant of the well-known game "Settlers of Catan." The game is set on a board representing the University of Waterloo, with tiles corresponding to different resources such as BRICK, ENERGY, GLASS, HEAT, and WIFI. Players, known as builders, take turns rolling dice to obtain resources based on where they've built residences. These resources can be used to build and improve residences, construct roads, and trade with other players. Special elements like geese, which can overrun tiles and inhibit resource collection, and the option to use "loaded" dice, add unique twists to the gameplay. The game's objective centers around strategic building, resource management, and trading, with the ultimate goal of achieving specific criteria detailed within the game's rules.

II. Overview

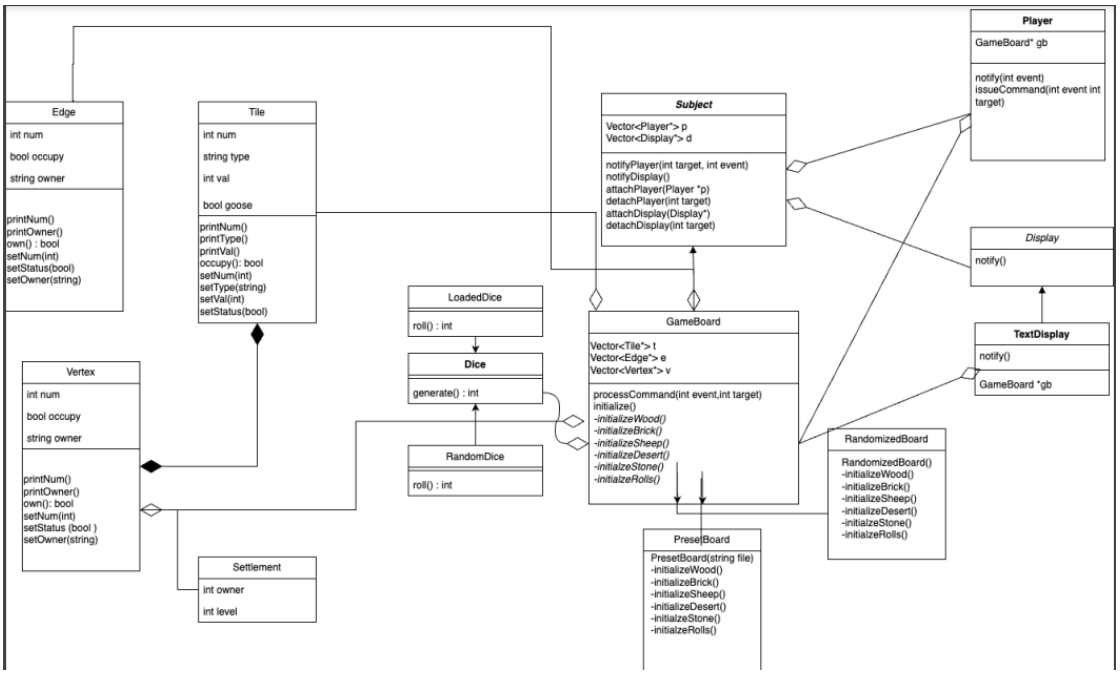
The structure of the "Constructor" is segmented into distinct classes and modules that represent the game's essential entities, such as players, board, dice, and display. The board's layout, formed by tiles, vertices, and edges, is defined through specialized classes, and game mechanics like resource gathering, trading, and building are encapsulated within dedicated methods. Observer pattern is utilized separately for gameplay interactions and . The user interface is structured to interpret command-line inputs, translating them into in-game actions. Careful attention is given to error handling and validation, reinforcing the integrity of the game state.

III. UML

This is the new UML.



This is the original UML.



There are some differences between the original UML. We created abstract classes for `Player`. (`Display` is now called `AbstractDisplay` and its subclass `TextDisplay` is now called `Display` instead). Other than that, we deleted RandomizedBoard and PresetBoard. In our previous planning, we are going to decide which board to use when receiving different commands. In addition, we decided not to use `Settlement` class. And the relationship between class `Tile`, `Vertex`, and `Edge` is different from that before.

IV. Design

Design Overview: As previously stated, the core of the programme is based on the **Observer Pattern.** The program can be divided into three clusters: the **Subject/GameBoard**, the **Player** and the **Display.**

I. Subject/GameBoard:

Class `subject` , as name suggests, acts as the subject of the Observer pattern. It owns and maintains the observers, the `AbstractPlayer` and the `AbstractDisplay`, parent of `Player` and `Display`, respectively.

Notify Methods:

`notifyPlayers(int target, int eventPara1, int eventPara2)` is probably one of the most important function in the programme. It sends requests and commands to the Player objects residing in a `vector` in class `Subject`. Target identifies the player who is impacted, eventPara1 and 2 identifies the detailed request from the player.

`setInput(int input)` is equally important. It allows players to quickly exchange simple information with the `Subject` and its much more notable subclass `GameBoard`. It allows players to send a single integer to the board quickly to respond to certain requests.

II. Board Design: The board's design is captured through three main classes: `board`, `gameBoard`, and `const`.

`board`: This class represents the main structure of the game board, including the layout of tiles, vertices, and edges. It handles the initialization of the board, placement of residences, and roads, and implements functionalities related to gameplay, such as resource allocation based on dice rolls. Key aspects include:

· Tiles: Organized into various resource types, tiles are initialized and positioned based on the game's specific configuration.

· Vertices and Edges: Vertices represent possible building locations, and edges define the connections between them, facilitating road construction.

· Residences and Roads: Methods within the class handle the placement, validation, and upgrading of residences. Roads are managed similarly, with careful adherence to adjacency rules.

In addition, there are some requirements for building the residences. For example, a residence can only be built on an empty vertex when there is an adjacent road and no residences on adjacent vertices. In order to better check the requirements, the adjacent vertices and edges are stored as vectors in each tile, vertex, and edge.

`gameBoard`: Extending the functionality of the board class, gameBoard is responsible for managing the overall state of the game, including the current player's turn, trading between players, and validating building placements. It acts as a central hub for interacting with the game's core mechanics. Key aspects include:

· Turn Management: Responsible for transitioning between players, it maintains the turn order and handles actions specific to each turn phase.

· Trading: Implements the complete trading functionality, including proposing, accepting, or rejecting trades, and transferring resources accordingly.

· Building Validations: Ensures that building actions adhere to the game's rules, checking valid locations, adjacency, and resource requirements.

`const`: This class encapsulates constant values and utility functions used throughout the game. It aids in maintaining consistency and provides a centralized location for managing constants related to the board and gameplay.

III. Dice Mechanics: The required in-game choice of dice type is done using **factory pattern**, assigning a concrete type of dice to a dice object everytime a new type of dice is chosen. When the player decides to roll, the virtual `generate()` will generate the number in the desired way.

Dice rolling is an essential game mechanic, elegantly handled by the classes `dice`, `RandomDice`, and `loadedDice`.

IV. Players Design: Players, referred to as builders, are represented by the `abstractPlayer` skelton and `player` classes.

`player`: Extending `abstractPlayer`, `player` implements the specific behaviors and interactions for individual players. It manages player-specific states, including their current resources, buildings, and unique actions.

V. Display Design: The game's visual representation is handled by the `abstractDisplay` and `display`.

`abstractDisplay`: defines the interface for displaying the game's state. `abstractDisplay` sets the groundwork for different display implementations, ensuring flexibility in visual representation.

`display`: implements the `abstractDisplay` interface. `display` provides the specific visualization of the game board, players' status, and ongoing actions. It translates the game state into a textual or graphical (Ascii) representation for the players to view. It also handles the input players need to play the game. This would allow us to utilize as many assets as possible from other modules when we decide to turn this into a game of video graphics.

V. Resilience to Changes

The design of the "Constructor" project exhibits significant resilience to change, accommodating potential alterations or expansions to the program specification. Several key factors contribute to this adaptability:

I. Modular Structure: By encapsulating specific functionalities within distinct classes and methods, the design facilitates isolated adjustments. For example, changes to dice mechanics can be confined to the dice and related classes without impacting other parts of the code.

II. Polymorphism and Inheritance: Leveraging object-oriented principles like polymorphism and inheritance, the design supports variations in game elements. Subclasses can be created to introduce new types of dice, players, or display methods, preserving existing code and minimizing disruption.

III. Centralized Constants: The const class centralizes constant values and utility functions, providing a single point of control for game parameters. Changes to resource values, building costs, or other constants can be made here, propagating consistently throughout the program.

VI. Answers to Questions

1. You have to implement the ability to choose between randomly setting up the resources of the board and reading the resources used from a file at runtime. What design pattern could you use to implement this feature? Did you use this design pattern? Why or why not?

We latter discovered that is hard to use template pattern for map generation. Since in the randomization generation you start with types of tiles, in the loaded generation you start with positions of tiles. These different approaches make it difficult to have a united way to generate such maps using template pattern.

2. You must be able to switch between loaded and fair dice at run-time. What design pattern could you use to implement this feature? Did you use this design pattern? Why or why not?

The Factory pattern could be used to create dice objects, with subclasses representing fair dice and loaded dice. The appropriate factory method could be called at runtime to create the desired type of dice.

3. We have defined the game of Constructor to have a specific board layout and size. Suppose we wanted to have different game modes (e.g. hexagonal tiles, a graphical display, different sized board

for a different numbers of players). What design pattern would you consider using for all of these ideas?

The Observer pattern is suitable for managing different displays. For different game boards, given that the board layout remains identical and only the resource locations change. For different sized board we can use the factory pattern for initializing.

4. At the moment, all Constructor players are humans. However, it would be nice to be able to allow a human player to quit and be replaced by a computer player. If we wanted to ensure that all player types always followed a legal sequence of actions during their turn, what design pattern would you use? Explain your choice.

We can use template method for that. Template method pattern is exactly for the case where a same sequence must be done but its steps can be done differently. This ensures that regardless of how the player chooses to perform their actions, they will always follow a sequence that adheres to the game rules. It also provides a clear and consistent framework for managing player turns, making the code easier to understand and maintain.

5. What design pattern would you use to allow the dynamic change of computer players, so that your game could support computer players that used different strategies, and were progressively more advanced/smarter/aggressive?

We can use the factory pattern and assign different strategies for different subclasses and produce corresponding actions accordingly. The Factory pattern enables the creation of players without specifying the exact class of object that will be created. This could be used to create various types of computer players with varying strategies.

7. Did you use any exceptions in your project? If so, where did you use them and why? If not, give an example of a place that it would make sense to use exceptions in your project and explain why you didn’t use them.

Exceptions could be used to handle illegal player commands, allowing the game to catch these errors at runtime and handle them in a way that allows it to continue running smoothly. If exceptions were not used, it is possible that the project used other error handling strategies, such as error return codes or conditional checks before performing potentially failing operations.

VIII. Extra Credit Features

All our programs are done by using stack memory. Therefore, there is no delete statement and no memory leak. Moreover, all the information, including tiles, vertices, edges, and players is stored using vectors.

IX. Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Working on the "Constructor" project reinforced the importance of clear communication, especially over the parameter and uses for functions that one write for other collaborators, and cohesive design in team-based software development. It highlighted the value of modular programming in managing large codebases, demonstrating how well-structured, independent components enhance code readability, maintainability, and

flexibility. The project also underscored the strategic use of design patterns, such as the Observer Design Pattern, to handle complex interactions effectively.

2. What would you have done differently if you had the chance to start over?

If given the chance to start over, I would focus on more thorough initial planning, emphasizing clear architecture and design patterns. Investing in automated testing early on and prioritizing documentation could streamline development and enhance code reliability and maintainability. These refinements would aim to create a more efficient and robust development process.

X. Conclusion

The "Constructor" project is a well-designed virtual board game that encapsulates complex mechanics and interactions. Leveraging object-oriented principles and the Observer Design Pattern, it achieves flexibility, dynamism, and resilience to change. The thoughtful architecture and engaging gameplay make it a standout example of software design, offering a rich and immersive gaming experience.