Plan of Attack

Yiheng Ye, Linus Zhang, Mingxiao Zhang

1. Responsibilities

Linus Zhang is responsible for the implementation of the following classes:

- Player
- Game
- Dice (Fair Dice and Loaded Dice)
- View

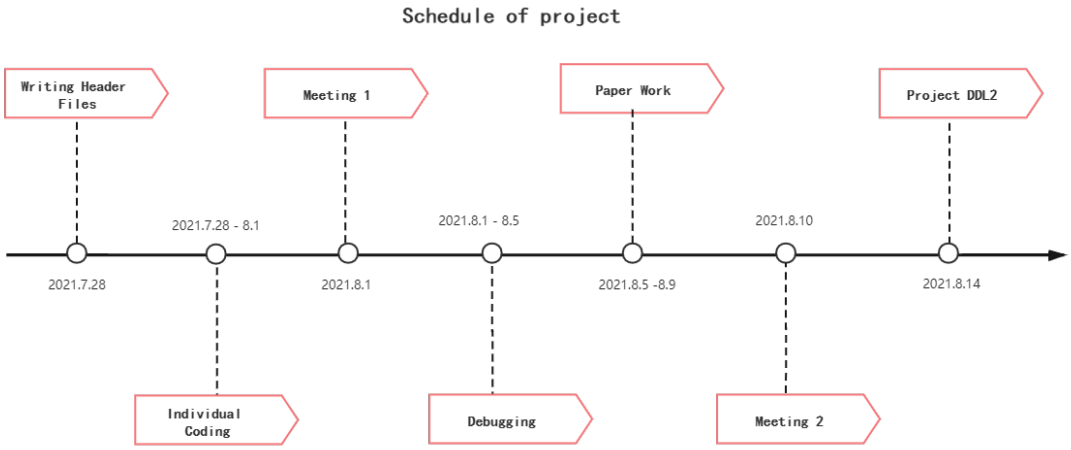
Yiheng Ye is responsible for the implementation of the following classes:

- Tile
- Geese
- Vertex
- Edge

Mingxiao Zhang is responsible for the implementation of the following classes:

- Board
- BoardFactory (RandomMode and FileMode)

2. Schedules



July 28: Writing Header Files

On July 28, three team members will together write the header files according to the UML

diagram. Necessary comments should also be included to reduce ambiguity. The header files should then be pushed to the group project private GitHub repository.

July 28 to July 31: Individual Coding

During this period, each team member will implement classes that he is responsible for. All implementations must be pushed to the group project private GitHub repository

August 1: Meeting 1

During this meeting, the team members will try to compile the entire project into an executable and then test it according to the project specification. If a bug is spotted, the input that triggers the bug should be recorded for future debugging.

August 1 to August 5: Debugging

This period is used for debugging. Three team members will together try to find the cause of bugs spotted and fix the bugs.

August 5 to August 9: Paperwork

This period is used for writing the project's final report, including analysis of the program and providing an overview of all aspects of the projects, demonstrating how each feature is implemented and how exactly the design patterns are applied. During this period, the team members should be ready to check unexpected errors or bugs.

August 10: Meeting 2

The target of this meeting is to evaluate and modify the final report to guarantee the project and the report fulfill all the requirements. At the same time, the team will submit the project for the first time, which is four days ahead of the DDL 2. If an unexpected error or crisis occurred during the last four days, the team should make emergency bug fixing. Submit the work in advance will be beneficial to any emergency happen.

3. Answers to Project 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?

The ability to choose between randomly setting up the resources of the board and reading the resources used from a file at runtime indicates that the program must have two methods to construct our gameboard. For this requirement, we shall use the factory design pattern in which we shall implement two factory classes to construct the gameboard. Both factory classes use the gameboard constructor to construct a blank board, but one fills the board with data from the file and another fills the board randomly.

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?

We would use the template method design pattern. The two types of dices have some common methods like yielding a number but have some different implementations as well. For example, the loaded dice would have to prompt the user for some input, but the fair dice would have a

random number generator inside. Therefore, we could implement an interface for their common methods and let them be derived from the dice interface. Other parts of the program using them would treat them both as the dice interface (instantiate both types of dice and store these instances both as the interface), and when the user switches the dice at the runtime, we simply switch which object to use and rest of the program remains the same.

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

We would consider using the bridge pattern in which we separate abstractions and implementations, and different implementations would allow different game modes. Taking adding graphical display as an example, previously, methods for notification would be a mix of notification details (e.g., the text of the notification) and displaying details for a specific user display (e.g., CLI, WinForm, WPF, Qt, GTK). This would be a mess if we continue to add new displays as all stuff is mixed.

To solve such a problem, we could separate notification details and displaying details into two types of classes: notification class for abstraction, and implementor classes for specific details of different displays. The notification class would have a display implementor derived from an interface. To add a new graphical display (e.g., we have CLI and GTK for Linux GUI, but we want to have a macOS GUI), we now only have to create new display implementor classes (which should be derived from the implementor common interface) and attach it to the notification classes at runtime.

6. Suppose we wanted to add a feature to change the tiles' production once the game has begun. For example, being able to improve a tile so that multiple types of resources can be obtained from the tile, or reduce the quantity of resources produced by the tile over time. What design pattern(s) could you use to facilitate this ability?

We could use the decorator design pattern. Basically, we would implement an abstract tile interface with getter methods for the production of each resource (resource getters) and then implement the concrete basic tile class and its decorator classes. The basic tile class would have resource getters that return quantities specified in the project document and the decorators' resource getters will add or subtract resources returned by the last element in the chain of effects.

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.

Generally, we will not throw exceptions from the models in our program.

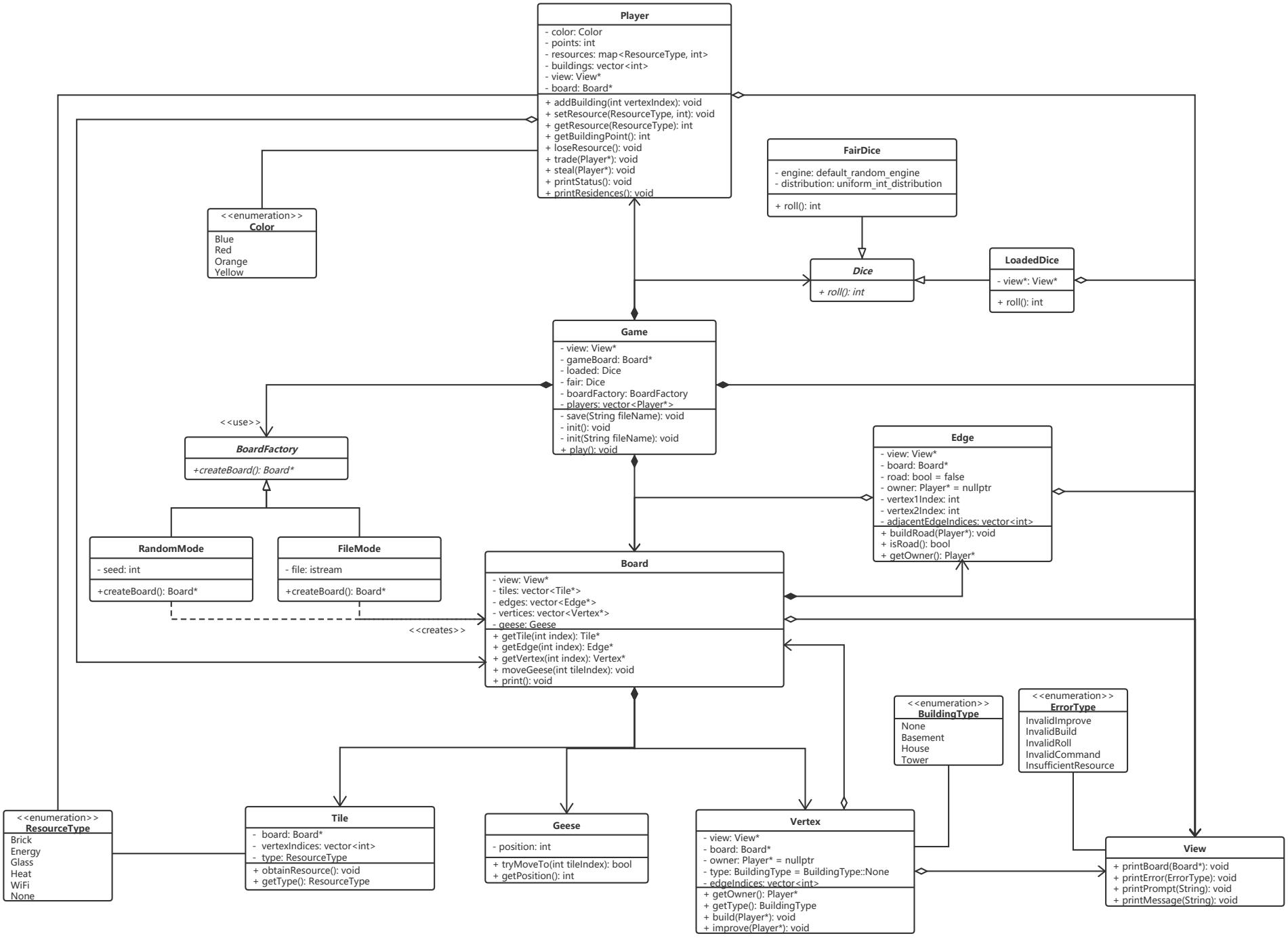
If the user input and program output are handled in one interaction class, exceptions should be thrown in other classes to notify the interaction class when the user's operation through the interaction class is illegal. Otherwise, other classes other than the interaction class will also have to handle error output directly, which is a clear breach of the Single Responsibility Principle and will cause problems when adding new user interaction methods (e.g., adding GUI to a command-line program).

But in our program, we shall use the MVC design model, in which the user-input handling (controller) and program output (view) are separated. If a user feeds an illegal operation to the models through the controller, the models should not throw an exception as the controller does not provide output functions. Instead, the model class will directly notify the view through function calls.

But we should try to catch exceptions from std objects and functions in our program.

Exceptions may occur from `fstream` during the reading and writing of the gameboard storage file, since the disk space may be full, the file may be used by another program simultaneously, or the program may not have sufficient privilege. And an exception may also occur when making a new memory allocation since the memory may not be enough.

We need to catch these exceptions to prevent the program from crashing and to provide useful prompts to the user.


Player
- color: Color
- points: int
- resources: map<ResourceType, int>
- buildings: vector<int>
- view: View*
- board: Board*
+ addBuilding(int vertexIndex): void
+ setResource(ResourceType, int): void
+ getResource(ResourceType): int
+ getBuildingPoint(): int
+ loseResource(): void
+ trade(Player*): void
+ steal(Player*): void
+ printStatus(): void
+ printResidences(): void
<<enumeration>>
Color
Blue
Red
Orange
Yellow
FairDice
- engine: default_random_engine
- distribution: uniform_int_distribution
+ roll(): int
Dice
+ roll(): int
LoadedDice
- view*: View*
+ roll(): int
Game
- view: View*
- gameBoard: Board*
- loaded: Dice
- fair: Dice
- boardFactory: BoardFactory
- players: vector<Player*>
- save(String fileName): void
- init(): void
- init(String fileName): void
+ play(): void
<<use>>
BoardFactory
+createBoard(): Board*
RandomMode
- seed: int
+createBoard(): Board*
FileMode
- file: istream
+createBoard(): Board*
<<creates>>
Edge
- view: View*
- board: Board*
- road: bool = false
- owner: Player* = nullptr
- vertex1Index: int
- vertex2Index: int
- adjacentEdgeIndices: vector<int>
+ buildRoad(Player*): void
+ isRoad(): bool
+ getOwner(): Player*
Board
- view: View*
- tiles: vector<Tile*>
- edges: vector<Edge*>
- vertices: vector<Vertex*>
- geese: Geese
+ getTile(int index): Tile*
+ getEdge(int index): Edge*
+ getVertex(int index): Vertex*
+ moveGeese(int tileIndex): void
+ print(): void
<<enumeration>>
BuildingType
None
Basement
House
Tower
<<enumeration>>
ErrorType
InvalidImprove
InvalidBuild
InvalidRoll
InvalidCommand
InsufficientResource
<<enumeration>>
ResourceType
Brick
Energy
Glass
Heat
WiFi
None
Tile
- board: Board*
- vertexIndices: vector<int>
- type: ResourceType
+ obtainResource(): void
+ getType(): ResourceType
Geese
- position: int
+ tryMoveTo(int tileIndex): bool
+ getPosition(): int
Vertex
- view: View*
- board: Board*
- owner: Player* = nullptr
- type: BuildingType = BuildingType::None
- edgeIndices: vector<int>
+ getOwner(): Player*
+ getType(): BuildingType
+ build(Player*): void
+ improve(Player*): void
View
+ printBoard(Board*): void
+ printError(ErrorType): void
+ printPrompt(String): void
+ printMessage(String): void