Nine Men's Morris Project Report - Sprint2

by Yuxiang Feng(32431813), Yifan Wang(32459238), Tianyi Liu(27936619)

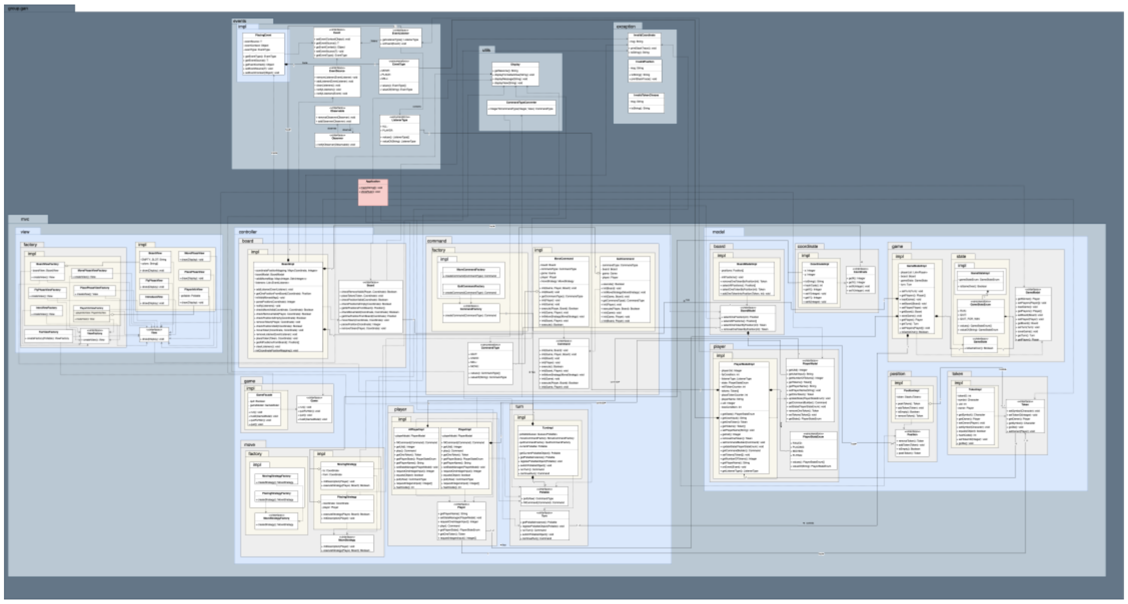
Team Name: **G**enerative **A**dversarial **N**etwork (short: GAN)

0. Project Package Tree

The following is the tree structure of the package of the project architecture, which is used to quickly understand the file structure of this project.

```
.
├── java
│   └── group
│   └── gan
│   ├── events
│   ├── exception
│   ├── mvc
│   │   ├── controller
│   │   │   ├── board
│   │   │   ├── command
│   │   │   │   └── factory
│   │   │   ├── game
│   │   │   ├── move
│   │   │   │   └── factory
│   │   │   ├── player
│   │   │   └── turn
│   │   ├── model
│   │   │   ├── board
│   │   │   ├── coordinate
│   │   │   ├── game
│   │   │   │   ├── rule
│   │   │   │   └── state
│   │   │   ├── player
│   │   │   ├── position
│   │   │   └── token
│   │   ├── view
│   │   └── factory
│   └── utils
└── resources
```

1. Class Diagram



Full Size Image and Download Link:

https://drive.google.com/file/d/1JYiMqHA57G0ExMMU5Xqp6xsg0tm7XP0J/view?usp=sharing

2. Design Rationale

This project realizes the development of full interface and the construction of MVC architecture mode. So in all the discussions that follow, almost all classes mentioned have their interfaces. Different classes communicate through interfaces, not through specific subclasses. At the same time, based on the general practice in the industry, the class name of the implementation class that implements the interface will have the suffix "Impl". For example, the implementation class of the **BoardModel** interface is **BoardModelImpl**.

2.1 Explain two key classes that you have debated as a team

We have implemented two important classes, that is, BoardModelImpl and BoardImpl. Their interfaces are BoardModel Interface and Board Interface respectively.

Through these two classes, we have completed the separation of concerns for "Board". In the traditional vision, "Board" needs to save data and interact with the outside world. But in the current design, we classify all the data storage functions of the chessboard in BoardModelImpl. At the same time, BoardModelImpl will provide meta-operations for manipulating data. For example, adding, deleting, modifying and checking operations. The implementation of complex operations and logic will be implemented by BoardImpl.

BoardModelImpl only provides the most basic meta-operations, such as querying a Position, or querying a Token on a Position. Or place a Token in a Position. These operations are basic and involve no logic.

BoardImpl itself does not have data, it just provides an interface to operate BoardModelImpl. For example, when a player wants to place a Token at a certain Position on the board, BoardImpl will perform the following three steps: Step 1, check whether the coordinates entered by the player are legal, for example, the player is not allowed to place a Token on a line or place the Token Placed on a position off the board. The second step is to check that the Position is empty, and the player is not allowed to place Token again on the Position that already has Token. In the third step, BoardImpl calls the method of BoardModelImpl to complete the placement of Token. These three steps must succeed or fail at the same time, so BoardImpl will combine these three operations into a function.

Similarly, BoardImpl can combine the two meta-operations provided by two BoardModelImpls, delete the Token of a certain Position and add a Token to another Position to complete the movement of the chess pieces.

At the same time, BoardModelImpl abstracts data storage. It does not need to store data according to the true model of the chessboard. For example, the 9MM chessboard is a 2D chessboard. If we use a two-dimensional array to store Position data, it will cause a lot of data waste, and we need to plan its coordinates in two-dimensional space. Since we use BoardModelImpl for data abstraction, we will use a one-dimensional array to store all Positions.

BoardModelImpl does not care how the Positions stored in the one-dimensional array are rendered in the View, because the rendering function will be managed by the BoardView.

Through the separation of concerns, the operation of data and the storage of data will be completely separated. This effectively prevents "Board" from becoming a God class, and at the same time makes each class have a clearer division of responsibilities. At the same time, because the Board itself has too much complex data and operations, we divide the Board into several different classes instead of several functions.

2.2 Explain two key relationships in your class diagram

The first is the Composition relationship between BoardModelImpl and Position. The two of them cannot exist independently, they need to exist at the same time. This is because the function of BoardModelImpl is to manage the meaningful data on the "board". Position can store Token. Therefore, BoardModelImpl needs to maintain the management of Position to achieve the function of managing "board" data. If Position is independent of Board, then the entire game will lack an effective interface for manipulating Position data. At the same time, Position itself should only complete the encapsulation of data. He should not know the relationship and coordinates of other Positions, it should not have such functions. The "board" is composed of multiple Positions. The Position completes the encapsulation of the data, and the BoardModelImpl completes the storage and operation of the data. If we define them as the Aggregation relationship, it means that the Position can be changed independently, which will cause the functions of the Position and the Board to be not clearly divided, which does not comply with the single responsibility principle. Therefore, BoardModelImpl and Position are Composition relationships.

The second is the Aggregation relationship between Position and Token. They can exist independently and operate independently of each other. This is because the information represented by Position does not depend on Token. Even without Token, Position can represent a coordinate information located in the "board". At the same time, players can place, move or mill pieces to Position. This causes the Token information in Position to be modified at any time. If we define them as Composition relationships, this obviously violates the basic logic of the game. Therefore, Position and Token can exist independently and operate independently of each other.

2.3 Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?

Based on our full interface development and design of the entire project and the realization of the MVC architecture pattern. Therefore, we decided not to use inheritance. This brings many benefits:

1. Flexibility: Interface development allows us to provide functions without changing other modules. Because even if we use a different implementation class, it must have the function of the corresponding interface. For example, in our current View implementation class, our BoardView displays the content of the game board to the player through the Terminal. If we replace BoardView with a new implementation class, it will display the chessboard through GUI rendering. The other modules of this game will not be affected in any way. Because when they want to display the chessboard, they will only request BoardView to render the chessboard. They don't pay attention to how the checkerboard is rendered. The new implementation class must have a draw() method. Another example is PlayerImpl and AIPlayerImpl, both of which implement the Plyaer interface. Therefore, when these two classes exist at the same time, Turn can run normally. Because its goal is to receive classes that implement the Player interface, they

must have specific functions for Turn to call. This design greatly satisfies the principle of dependency inversion.

2. Low coupling: Since we have adopted full-interface development, the degree of coupling between modules is greatly reduced. We can maintain and change each module without affecting other modules. It would be bad if we used inheritance. For example, if the function of a parent class is changed in subsequent development, all subclasses have to follow this change. This greatly improves the coupling between modules.

3. Composition-based instead of inheritance-based: If we want to implement multiple functions, we can place the constraints of these functions in different interfaces. Through such a flexible combination, we can assign different functions to different classes. For example, a class that implements the EventListener interface will be able to listen to events. We only need to let the class that needs to listen to the event implement the EventListener interface. We can flexibly combine multiple interfaces without worrying about any inheritance issues.

4. The division of functions is more clear: because the interface specifies which functions the class that implements the interface has. Therefore, using interface development instead of inheritance can reduce the maintenance cost between modules.

Not using inheritance makes perfect sense in our design. Because we use full interface development, different modules communicate through interfaces rather than specific implementation classes. Therefore, when we need to replace or update a certain function, we don't need to inherit the parent class, but to implement different implementation classes under the same interface.

2.4 Explain how you arrived at two sets of cardinalities

The cardinality between PositionImpl and TokenImpl is 0...1, because at most one Token can be placed on a Position. Tokens cannot overlap. At the same time, since there are 24 Positions and 18 Tokens, there must be situations where there are no Tokens on some Positions. Since only one chess piece can be placed on a Position. So 0...1 instead of 1...2.

The cardinality of PlayerImpl and PlayerModelImpl is 1. This is because the data of one PlayerImpl should only be saved and managed by one PlayerModelImpl. PlayerImpl belongs to controller, it depends on PlayerModelImpl to manage data, if PlayerModelImpl is lost, then PlayerImpl will lose data. Therefore the cardinality between them must be 1, not 0...1.

2.5 Explain why you have applied a particular design pattern for your software architecture

We have used factory design pattern in our architecture and are not going to use singleton design pattern and prototype design pattern.

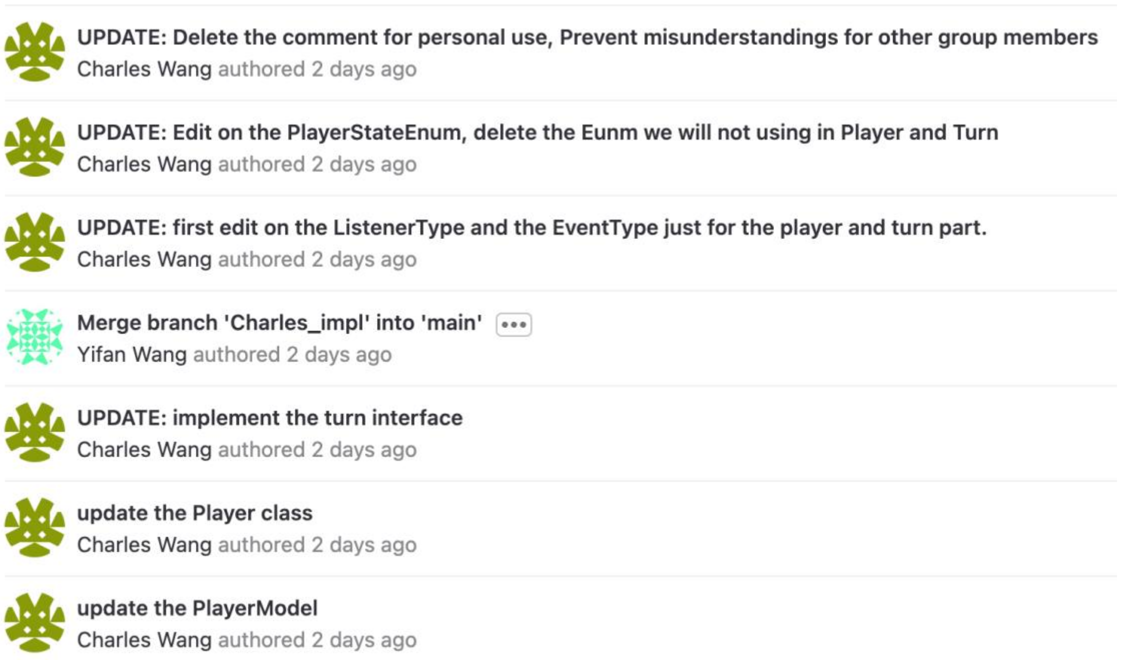
The factory design pattern is used because, in our architecture, we will create View, Command and Strategy many times. Using the factory design pattern can reduce the coupling between "user" classes and them. At the same time, doing so can perfectly comply with the single responsibility principle and the opening and closing principle. We can put the creation of specific classes in a separate factory, making the creation of classes easier to maintain. At the same time, we can complete the replacement of the creation without changing the existing framework. For example, for our current BoardView, it is responsible for rendering the chessboard to the Terminal. If we implement a new implementation class, its function is to render the chessboard into the GUI. Then we only need to modify the implementation class generated in the factory. The rest of the frame will not be affected in any way.

The reason for not using the singleton design pattern is that the singleton design pattern violates the single responsibility principle. And this will make them face the problems of multiple reuse and multiple class influences. This is very likely to cause problems with the running security and stability of the program.

The reason for not using the prototype design pattern is that our class does not involve the cloning function. Not only that, but we have many classes that need to be able to be compared. For example, between different players, even if they have the same name and the same data, they should be different players. Because they have different uids. We should not allow clones to appear on the basis of existing frameworks, which will lead to instability during the operation of the framework.

3. Special Instructions

Due to one of our members working on two different computers of his, two different identities appeared in their git records. Charles Wang in the Commits record is the same person as Yifan Wang.



UPDATE: Delete the comment for personal use, Prevent misunderstandings for other group members
Charles Wang authored 2 days ago

UPDATE: Edit on the PlayerStateEnum, delete the Eunm we will not using in Player and Turn
Charles Wang authored 2 days ago

UPDATE: first edit on the ListenerType and the EventType just for the player and turn part.
Charles Wang authored 2 days ago

Merge branch 'Charles_impl' into 'main' •••
Yifan Wang authored 2 days ago

UPDATE: implement the turn interface
Charles Wang authored 2 days ago

update the Player class
Charles Wang authored 2 days ago

update the PlayerModel
Charles Wang authored 2 days ago