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.

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.

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.