文本

描述已自动生成

**IT+ Masters Team Project M**

**2021-22**

Table of Contents

[1 Class List 2](#_Toc94481124)

[1.1 Basic Data Holding Classes 2](#_Toc94481125)

[1.2 Event Processing Classes 6](#_Toc94481127)

[1.3 Some Other Possible Classes 7](#_Toc94481128)

[2 Game Round Process Flow 8](#_Toc94481129)

[3 Story 11](#_Toc94481130)

[3.1 roles 11](#_Toc94481131)

[3.2 Story list 11](#_Toc94481132)

[3.3 Story card 12](#_Toc94481133)

# 1 Class List

## 1.1 Basic Data Holding Classes

**Classname: Board**

Description: The Board is a gloabal data class designed for easier assessment of dynamic parameters on the battleground at any time, it has a variety of datas such as the information of the playing board (e.g. the width and length of the board) as well as a static two-dimensional array called tilesCollection, which reflect the occupation states of tiles (e.g. tilesCollection[xpos][ypos] = TileInstance,  if you know the tileA position is (1,2), you can easily know whether there is an unit on it with checking tilesCollection[1][2].unit != null ). Furthermore, with Board(int width, int length) constructor function players could initialize the board by filling it with tiles. It also has a dataset Map<tile,mode> (see Tile class).

It also has two references to two Player classes which are the player and the AI.

It also has an int variable called whoseTurn to show which player is currently playing the turn.

It also has a Boolean variable called endGame to denote whether the game has come to an end.

It also has a reference to the Unit if that one is currently being selected.

Relationships: No extends.

*Possible Attributes and Methods in Board：*

*int width*

*int length*

*static Tile[] []  tilesCollection*

*Map<tile,mode> tilesMode*

*boolean endGame*

*int whoseTurn \* 1 stand for players turn while -1 stand for AI’s turn.*

*Player player1*

*Player player2(AI)*

*Unit unitSelected*

## Classname: Player

Description: The Player Class is the main character to execute functionalities. Player instances should be created when the initialize() function in the GameActor is called. The Player class has basic info about th initialize e player which include the current mana and the current health of the player, Player should also have a deck information dataset as well as a list(List<Integer> unitId) and a map(Map<Card, mode> handcardMode) which records the information of current cards at hand.

Relationships: No extends.

*Possible Attributes and Methods in Player：*

*int defaultHealth*

*int currentHealth; // Current player health*

*int mana*

*int maxMana*

*List<Integer> unitId*

*List<Card> handcard*

*Map<card, mode> handcardMode*

*Queue<Card> deck = new LinkedList<>()*

**Classname:AI**

Description:This is a class which automatically finds and executes the behaviors of a non-human opponent during the other playing turn.

Relationships: **extends Player**

*AI doesn’t need to be covered here, but we could still suggest some Possible Attributes and Methods in AI*

*It has the following functions:*

*JudgeUnitCardPriority()*

*judgeUnitSummonTile()*

*JudgeSpellCardPriority()*

*JudgeSpellTile()*

*JudgeUnitMovePriority()*

*JudgeUnitMoveTo()*

*JudgeUnitAttackPriority()*

*JudgeUnitAttackTo()*

**Classname: Tile**

Description: Tile class is purely a representation of a grid of the battle ground, it contains the x and y coordination of a point. Other than that, the tile instances are important components of many datasets.e.g. the two-dimensional array tilesCollection(see “Board” class);  the Map<tiles,mode> is generally used to check the destination state of a tile.

Relationships: No extends.

*Possible Attributes and Methods in Tile*

*List<String> tileTextures; // urls of different tile textures*

*int xpos; // x (horizontal) pixel position of the tile*

*int ypos; // y (vertical) pixel position of the tile*

*int width; // Pixel width of the tile*

*int height; // Pixel height of the tile*

*int tilex; // x (horizontal) index in the board grid (range 1-9)*

*int tiley; // y (vertical) index in the board grid (range 1-5)*

*Unit unit*

*boolean isOccupied()*

**Classname:Unit**

Description: Unit class is an important class used to define all attributes of a unit, which has some dynamic parameters including

1, a unique id which uniquely defines the unit.

2, the current tactical state of the unit, like current damage and current HP, which can be used to judge the result of attackings and spellings.

3, the current action state of the unit, like position, isMoved, isAttacked, current move/attack point, which can be utilized to judge the attacking range as well as AI tactics.

4, card Instance, which is used to judge if the unit belongs to a ranged attack unit, or used to get the default HP and default damage when buff spellings are used.

5, animations which could be used to visualize the various states of the unit.

Relationships: No extends.

*Possible Attributes and Methods in Unit*

*isMoved and isAttacked,*

*int id; // a unique id for the unit*

*UnitAnimationType animation; // animation to play [idle,death,attack,move]*

*Position position; // contains board position data*

*UnitAnimationSet animations; // contains animation data*

*ImageCorrection correction; // contains pixel position correction data*

*int currentAttack*

*int currentHealth*

*int defaultAttack*

*int defaultHealth*

*boolean isMoved*

*boolean isAttacked*

*int curMovePoints()*

*int curAttackTimes()*

**Classname:Card (abstract class)**

Description: The Card Class is a common class which is used to describe the main attributes of different cards, which include unit cards and spell cards.

Relationships: No extends.

*Possible Attributes and Methods in Card*

*Map<String, Card> CardMap*

*getCard(String cardname)*

**Classname:UnitCard / SpellCard**

Description: Extended by sub-cards to distinguish those in order to show their proper range on canvas.

Relationships: extends Card

**Classname: Sub-Cards e.g. “ComodoCharger”**

Description: Those classes represent 20 + 2(including 2 avatar cards) kinds of different cards in decks. Those sub-card classes are not only used to provide the data information as well as the procedures when playing the cards, but also contain urls of visualization materials, which include all kinds of icons and preview windows for the cards to display.

In order to ensure encapsulation and reduce coupling, sub-card classes are designed as nested singletons in the Card class and can only be accessed through a static variable of the Card class. Sub-cards classes have the command execution functionality to realize the card effect. E.g. to use a ComodoCharger card just call CardMap.getCard(“ComodoCharger”).executeCombo(), the executeCombo() function will get the List<Integer> unitId from Player instance, then calculate the biggest unitId, then add 1 to that unitId as a new id to instantiate a new unit( , unitId, position), then call the **Administrator.sendMana(Player player)** function to reduce the player's mana. Meanwhile use the playEffectAnimation(out, StaticConfFiles.f1\_summon, tile) method to place the unit on the canvas.

Relationships: extends UnitCard / SpellCard

*Possible Attributes and Methods in SpecifiesCard*

*int cardID*

*String cardName*

*int manaCost*

*MiniCard miniCard;*

*BigCard bigCard;*

*method executeCombo()*

*method setInstance()*

**Classname:Administrator**

Description:Administrator class is used to maintain various static methods which are vital components of the game logic.

Relationships: No extends.

*Possible Attributes and Methods in Administrator*

*Method:*

*Dealing(Player player, int cardNo)*

*sendMana(Player player, int mana)*

*attackCalc(Unit unit)*

*checkUnitRange(Card card)*

*checkSpellRange(Card card)*

*moveUnit(Unit unit, tile)*

*attackUnit(Unit attackingUnit, Unit targetUnit)*

*checkOperation(Player player)*

*checkVictory(Player player)*

*noInitializationResponse()*

*endGameResponse()*

*nonePlayerTurnResponse()*

## 1.2 Event Processing Classes

**Explanation on event classes:**

Sub-Event Processor classes are designed as singletons and can only be accessed through the Map<String(eventname), EventProcessor(SubEventProcessor)> eventMap.

Each sub-EventProcessor class implements the EventProcessor interface and should have its own way of overriding the processEvent().

When the user operates on the front end, it sends an event message to GameActor class. The message is used to trigger the event processors by the code eventMap.get(event).processEvent. In this way event logic is executed.

The eventMap is created and maintained in the GameActor class.

**Classname: Initialize**

Description:For detailed logic see Game Round Process flow <6.Event – Initialized>.

Relationships: Implements EventProcessor

*Possible Attributes and Methods：*

*processEvent(ActorRef out, Board board, JsonNode message)*

*Initialize getInstance()*

**Classname: CardClicked**

Description: For detailed logic see Game Round Process flow <7. Event - CardClicked>.

Relationships: Implements EventProcessor

*Possible Attributes and Methods：*

*processEvent(ActorRef out, Board board, JsonNode message)*

*CardClicked getInstance()*

**Classname: TileClicked**

Description: For detailed logic see Game Round Process flow <8. Event - TileClicked>.

Relationships: Implements EventProcessor

*Possible Attributes and Methods：*

*processEvent(ActorRef out, Board board, JsonNode message)*

*TileClicked getInstance()*

**Classname: UnitMoving / UnitStopped**

Description: Play the moving animation and idle animation of the friendly units. For detailed logic see Game Round Process flow <9. Event - UnitMoving / UnitStopped>.

Relationships: Implements EventProcessor

*Possible Attributes and Methods：*

*processEvent(ActorRef out, Board board, JsonNode message)*

*UnitMoving / UnitStopped getInstance()*

**Classname: EndturnClicked**

Description: For detailed logic see Game Round Process flow <9. Event – EndTurnClicked>.

Relationships: Implements EventProcessor

*Possible Attributes and Methods：*

*processEvent(ActorRef out, Board board, JsonNode message)*

*EndturnClicked getInstance()*

**Classname: OtherClicked**

Description: For detailed logic see Game Round Process flow <10. Event – OtherClicked>.

Relationships: Implements EventProcessor

*Possible Attributes and Methods：*

*processEvent(ActorRef out, Board board, JsonNode message)*

*OtherClicked getInstance()*

**Classname: HeartBeat**

Description: For detailed logic see Game Round Process flow <11. Event – HeartBeat>.

Relationships: Implements EventProcessor

*Possible Attributes and Methods：*

*processEvent(ActorRef out, Board board, JsonNode message)*

*HeartBeat getInstance()*

**Interface: EventProcessor**

Description: In order to achieve polymorphism when assembling Map<String(eventname), EventProcessor(SubEventProcessor)> eventMap, all

event processing classes should implement this interface.

Relationships: No extends

*Possible Attributes and Methods：*

*processEvent(ActorRef out, Board board, JsonNode message)*

**Classname: GameActor**

The game actor is an Akka Actor that receives events from the user front-end UI (e.g. when the user clicks on the board) via a websocket connection. When an event arrives, the processEvent() method is called, which can be used to react to the event. The Game actor also includes an ActorRef object which can be used to issue commands to the UI to change what the user sees. The GameActor is created when the user browser creates a websocket connection to backend services (on load of the game web page).

Relationships: Extends akka.actor.AbstractActor

*Possible Attributes and Methods：*

*drawTile(ActorRef out, Tile tile, int mode)*

*drawUnit(ActorRef out, Tile tile)*

*setUnitAttack(ActorRef out, Unit unit, int attack)*

*setUnitHealth(ActorRef out, Unit unit, int health)*

*moveUnitToTile(ActorRef out, Unit unit, Tile tile)*

*moveUnitToTile(ActorRef out, Unit unit, Tile tile, boolean yfirst)*

*playUnitAnimation(ActorRef out, Unit unit, UnitAnimationType animation)*

*deleteUnit(ActorRef out, Unit unit)*

*setPlayer1Health/setPlayer2Health(ActorRef out, Player player)*

*setPlayer1Mana/setPlayer2Mana(ActorRef out, Player player)*

*drawCard(ActorRef out, Card card, int position, int mode)*

*deleteCard(ActorRef out, position)*

*playEffectAnimation(ActorRef out, EffectAnimation effect, Tile tile)*

*playProjectileAnimation(...)*

*addPlayer1Notification(...)*

## 1.3 Some Other Possible Classes

**Position**

This is a small class which contains the detailed position infos, including the tilex and tiley of the Unit.

**UnitAnimationType**

This is a class which define unit statement visualization, e.g. attack frames, idle frames

**UnitAnimationSet**

Contains Animation data.

**ImageCorrection**

Used to adjust the units and icons.

**EffectAnimation**

The EffectAnimation class isan animation for a unit to play. Each unit has a set of animation frames stored in an array.

# 2 Game Round Process Flow

1.

The player opens the game screen in their browser, this retrieves the HTML and JavaScript from the Web Server.

2.

To satisfy the request a GameController Java class is initialized, which when ready responds to the request with the HTML and Javascript data.

3.

The browser renders the HTML and starts the javascript initialization function, which calls openWebsocketConnection().

4.

The WebSocket connection is initialized with the web server, this triggers GameController to create a new GameActor instance to mediate the game.

5.

On as part of the constructor, the GameActor issues a first command‘actorReady’ to the front- end to let the browser know that the backend is ready.

6. **Event – Initialize:**

Response with invoking eventMap.get(“Initialize”).processEvent( *ActorRef out, Board board, JsonNode message* ), for mechanism see Class list: Explaination on event classes.

Then starts the initialization process by creating instances of all data classes: creating a board instance with **new Board(int width, int length)**, creating the player instance with **new Player/AI( Board board )**, which instantiates the Player and AI, initializes 20 +2 (including 2 avatar cards) card instances with the singleton pattern and put them in a Map so that the card instances could be get by their name, initializes two hand Lists, two deck Lists of 20 cards each.

Place the avatars on canvas by calling **executeCombo()** of the avatar card instances ( for detailed information see classlist **sub-cards**).

Then the system issues 3 cards each to the player and the AI by calling the static function **Administrator.Dealing(Player player, 3)**, which take 3 cards instances from Queue<Card> deck and add to List<Card> handcard, setting the handcard mode to 0 by changing Map<Card,Mode> handcardMode.

Then calls the **Administrator.sendMana(Player player, 1)** function to add 1 mana to both players.

7. **Event - CardClicked**:

Response with invoking eventMap.get(“CardClicked”).processEvent( *ActorRef out, Board board, JsonNode message* ), for mechanism see Class list: Explanation on event classes.

Do the checks in the beginning (do the same with the following events):

If Board doesn’t exists, execute *Adminstrator.noInitializationResponse(); return;*

If(*Board.endGame == true) execute Administrator.endGameResponse(); return;*

*If(Board.whoseTurn == -1), execute Adminstrator.nonePlayerTurnResponse(); return;*

The front-end passes “cardClicked” along with a position to the back-end, finds the card instance in the corresponding position in **List<Card> handcard**. If in there is no card instance found in the position, then check if both the **List<Card> handcard** and **Queue<Card> deck** are empty which lead to the end of game.

If card instance found, then changes its mode to 1 in **Map<Card, mode> handcardMode**, then pass **mode=1** to **drawCard(, Card card, int postition, int mode)** to activate the card in UI.

Then judge whether the parent class of this instance is UnitCard or SpellCard? If the card is a unitCard, call **Administrator.checkUnitCardRange(Card card)** to show the location where the unit can be placed; if the card is a spellCard, call **Administrator.checkSpellCardTile()** to show the location where the spell can be placed. Then send the returned hash table **Map<tile,mode>** which stores the places to the front-end, so that the range lights up.

8. **Event - TileClicked:**

Response with invoking eventMap.get(“TileClicked”).processEvent( *ActorRef out, Board board, JsonNode message* ), for mechanism see Class list: Explaination on event classes.

**(1)Check if the handcard is already selected:**

Checking the **Map<Card, mode> handcardMode**, If there is a card which mode=1 then it could possibly be put on board with this click.

Then put the tilex and tiley in the tilesCollection[tileX][tileY] to get the tile instance, passes the tile instance into the Map<tile,mode> tilesMode to get the mode of tile, if the mode is 1 then use the card by calling **executeCombo()** of the card instance ( for detailed information see classlist **sub-cards**). If the mode is not 1 then clear all modes to 0 both in Map<tile,mode> tilesMode and Map<Card, mode> handcardMode, and then clean the canvas by calling **drawCard(, Card card, int mode)** and **drawTile( ,Tile tile, int mode)** with mode=0 as the argument.

**(2)Check if an unit has already been selected:**

After finishing step (1), if no hand card is already selected, then check if an unit is already selected with checking Board.unitSelected==null. If there aren’t unit selected, then check if the tile is already occupied by checking tile.isOccupied(). If occupied and it’s the friendly unit then make the unit selected by adding to Board.unitSelected = unit.

Meanwhile, show the range that the unit could move or attack by calling Administrator.checkUnitRange(Unit unit), this method will calculate the range of the unit according to its current move/attack point, then put modes into Map<tile, mode> tilesMode and show range on the canvas using drawTIle ( ,Tile tile, int mode) .

If there is a unit already selected, put the tile in Map<Tile, mode> tilesMode to see its mode, if the mode is 1 then make a move by using method Administrator.move(), the method could select the route which isn’t blocked by enemy unit; if the mode is 2 then use Administrator.attack() which will move to an adjacent position(for remote attack unit it won’t move) to target and calculate the attack and counter attack results.

9. **Event – EndTurnClicked:**

Response with invoking eventMap.get(“EndTurnClicked”).processEvent( *ActorRef out, Board board, JsonNode message* ), for mechanism see Class list: Explanation on event classes.

If Board.whoseTurn is -1:

Flip the Board.whoseTurn to 1;

If Board.whoseTurn is 1:

Flip the Board.whoseTurn to -1, and utilize the AI logic:

AI could also utilize the eventMap.get(Event).processEvent( *ActorRef out, Board board, JsonNode message* ). However, the event message is created by algorism here instead of from front-end.

Detailed AI algorism isn’t covered here so this part is omitted.

10. **Event – OtherClicked:**

Response with invoking eventMap.get(“OtherClicked”).processEvent( *ActorRef out, Board board, JsonNode message* ), for mechanism see Class list: Explanation on event classes.

Set Modes to 0 in all data structures and call all visualization method to refresh the battle ground.

11. **Event - UnitMoving / UnitStopped:**

Response with invoking eventMap.get(“UnitMoving” / “UnitStopped”).processEvent( *ActorRef out, Board board, JsonNode message* ), for mechanism see Class list: Explaination on event classes.

Unit animation should changed in accordance with action events. When UnitMoving event arrives, response with calling playUnitAnimation( ActorRef out, UnitAnimationType animation=“move”), when UnitStopped arrives it should be playUnitAnimation( ActorRef out, UnitAnimationType animation=“idle”).

12. **Event - HeartBeat:**

Response with invoking eventMap.get(“HeartBeat”).processEvent( *ActorRef out, Board board, JsonNode message* ), for mechanism see Class list: Explaination on event classes.

Iterate to see if the game is ended. The system calls the **Administrator.checkVictory(Player player)** method to check both the players’ health and deck, if one of them is unsatisfied then write the Board.endGame to true. Meanwhile send a victory notification.

Iterate to see if there is still room to manoeuvre. The system calls the **Administrator.checkOperation(Player player)** method to iterate through the player's hand list and unitList and remaining mana to see if there is any action available, if not then pass back the parameters to the front end to highlight the end turn and notify the player.

# 3 Story

## 3.1 roles

**1. player**

**2. AI**

**3. Administrator ( the Administrator here is a virtual user rule who is responsible for keeping the logics and rules of the game running smoothly, it isn’t a human being. Besides, it has nothing to do with the Administrator class in the class list)**

## 3.2 Story list

**Administrator:**

Story name: Administrator Generates board Module

Story name: Administrator Generates the player Module

Story name: Administrator Distributes three hands Module

Story name: Administrator Updates the state of the hands Module

Story name: Administrator Sets mana value Module

Story name: Administrator Compares the selected cards with the remaining mana values Module

Story name: Administrator Shows the range of the unitcard available Module

Story name: Administrator Shows spellcard usable range Module

Story name: Administrator Unit Movement expression Module

Story name: Administrator Unit Attack Range Module Module

Story name: Administrator see if there is still actionable behaviour Module

**Player:**

Story name: Player Chooses a deck Module

Story name: Player Draws a card from the deck Module

Story name: Player User selects a card to compare Module

Story name: Player Selects Unit Module

Story name: Player Summons unit Module

Story name: Player uses spell Module

Story name: Player Moves unit Module

Story name: Player Checks Unit Attack Available Module

Story name: Player Unit Attack Module

Story name: Player End of Round Module

Story name: Player uses Planar Scout card Module

Story name: Player uses Rock Pulveriser card Module

Story name: Player uses Pyromancer card Module

Story name: Player uses Bloodshard Golem card Module

Story name: Player uses Blaze Hound card Module

Story name: Player uses Windshrike card Module

Story name: Player uses Hailstone Golem card Module

Story name: Player uses Serpenti card Module

Story name: Player uses Staff of Y’Kir card Module

Story name: Player uses Entropic Decay card Module

Story name: Player uses Comodo Charger card Module

Story name: Player uses Hailstone Golem card Module

Story name: Player uses Pureblade Enforcer card Module

Story name: Player uses Azure Herald card Module

Story name: Player uses Silverguard Knight card Module

Story name: Player uses Azurite Lion card Module

Story name: Player uses Fire Spitter card Module

Story name: Player uses Ironcliff Guardian card Module

Story name: Player uses Truestrike card Module

Story name: Player uses Sundrop Elixir card Module

**AI:**

Story name: AI draws Module

Story name: AI chooses the hand to play Module

Story name: AI gives priority of unit cards Module

Story name: AI summons the unit Module

Story name: AI gives priority of spell usage in playable hands Module

Story name: AI uses spell Module

Story name: AI determines the priority of unit's movement Module

Story name: AI moves unit Module

Story name: AI Unit Dynamic Path Planning Module

Story name: AI determines the priority of unit summoning among playable cards Module

Story name: AI determines the attack priority of the unit Module

Story name: AI attacks Module

Story name: AI ends turn Module

## 3.3 Story card

**Administrator:**

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件, Teams

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**Player:**

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 文本, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

**AI:**

图形用户界面, 应用程序, Teams

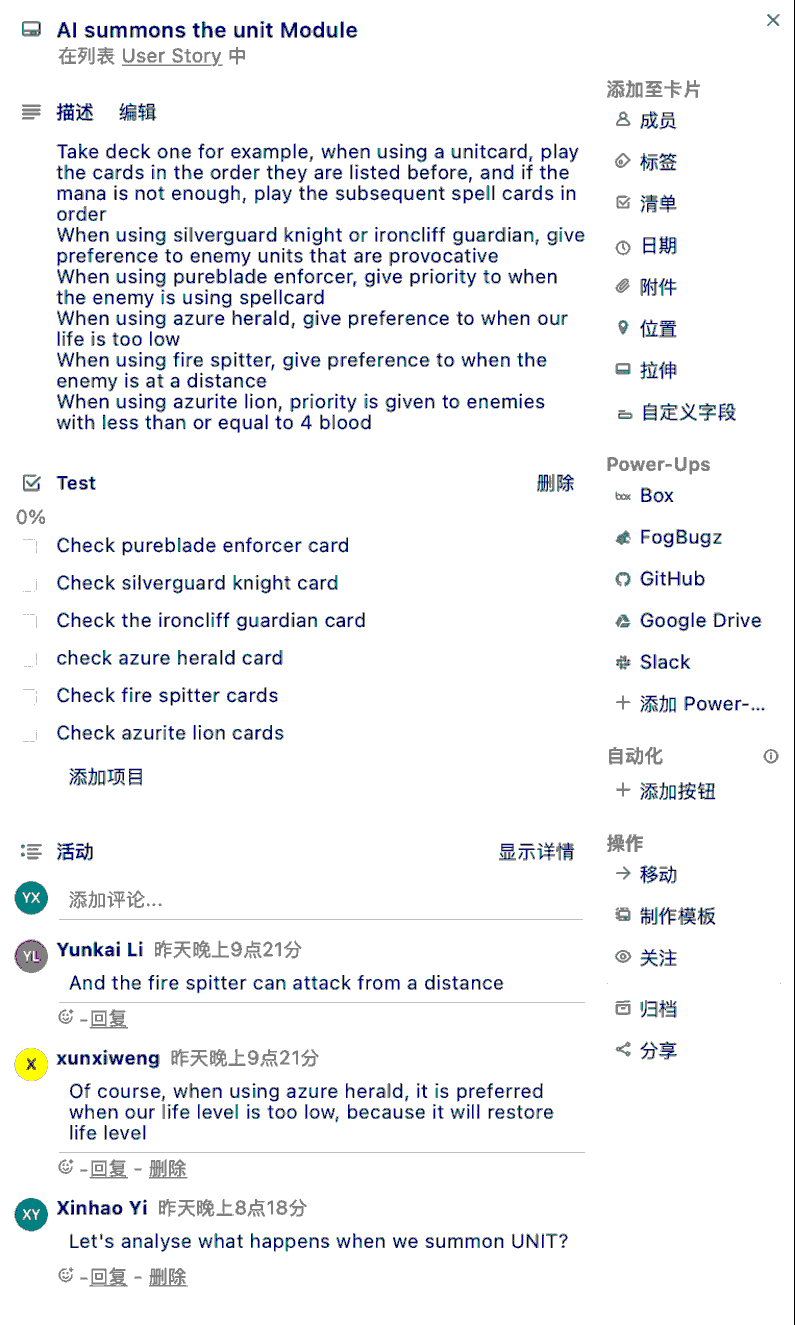
描述已自动生成

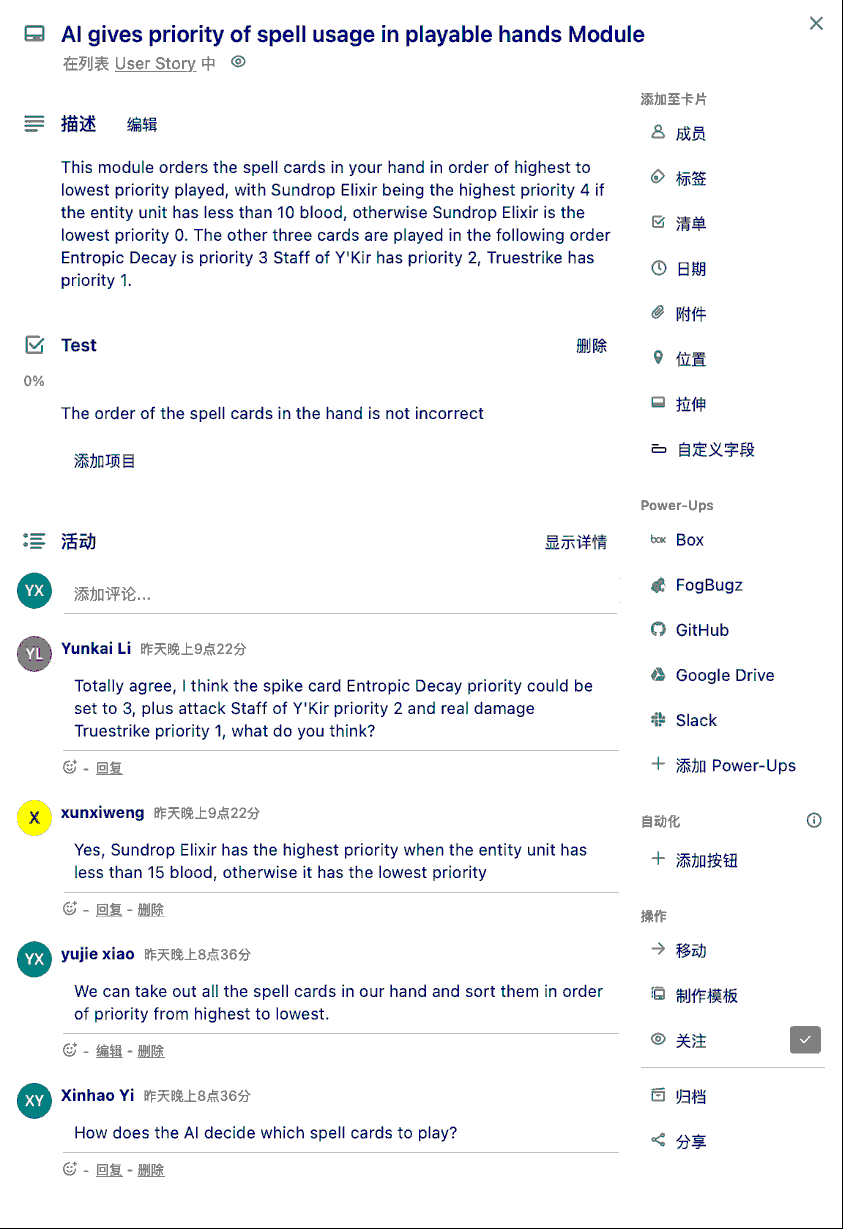
图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 文本, 应用程序, Teams

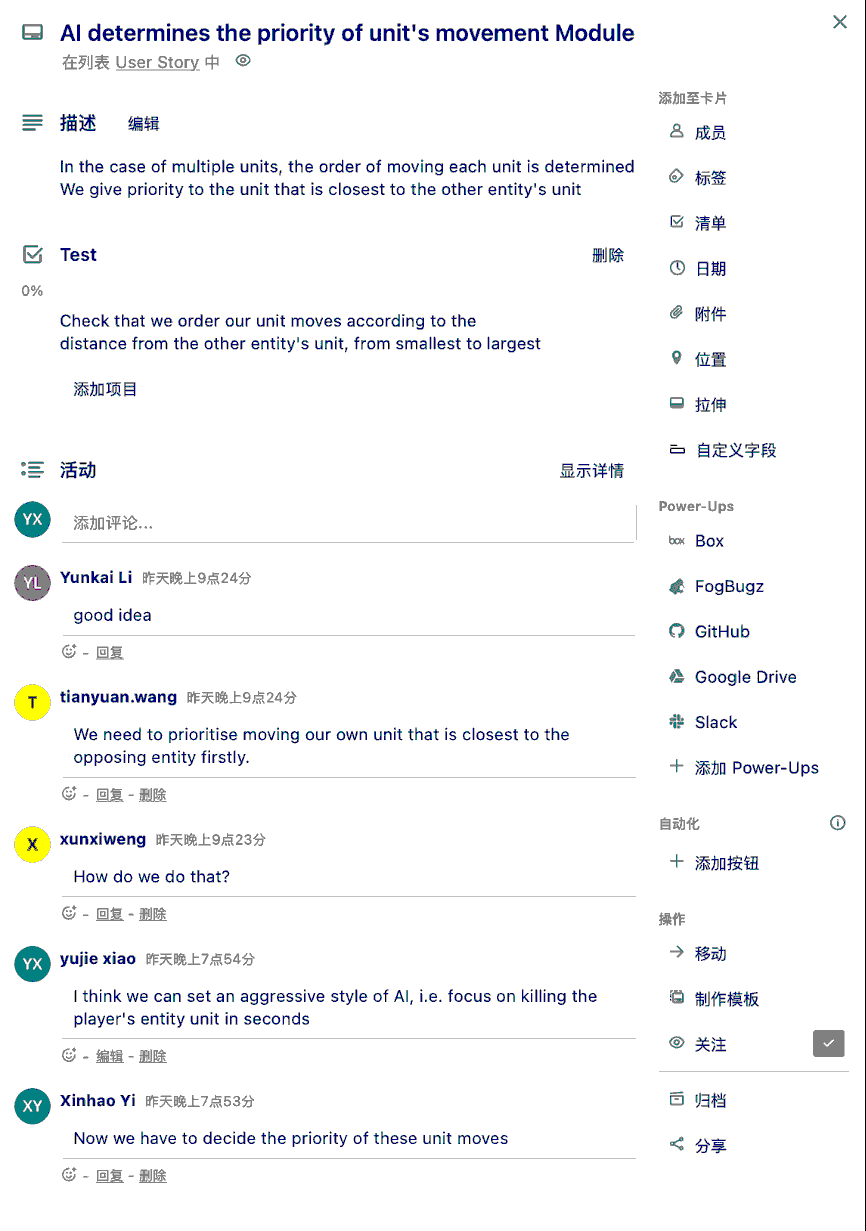
描述已自动生成





图形用户界面, 文本, 应用程序, Teams

描述已自动生成



图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件, Teams

描述已自动生成

图形用户界面, 应用程序, 电子邮件, Teams

描述已自动生成

图形用户界面, 应用程序, Teams

描述已自动生成