Documentazione relativa all'UML (Model) del gioco Codex Naturalis per peer review

Gruppo 32: Bergagna Matteo, Brun Antony, Chen Jie, Chiroli Lorenzo

Riportiamo in questo file tutte le parti relative alla logica di gioco implementate dal Model: il Controller implementa al suo interno tutta la parte sequenziale (eg. turni), invece, il resto (eg. regole) viene gestito dal Model.

Il Model è visto, "ad alto livello", come una "fotografia" dello stato di gioco che può venir modificata dal Controller mediante i metodi esposti; le View non saranno direttamente connesse al Model ma i loro dati passeranno sempre attraverso il Controller.

Controller: Per ogni partita, il controller crea un'istanza di Match: MyMatch, dopodiché sarà il costruttore di Match a creare la partita e tutti gli altri oggetti necessari.

Model: segue una descrizione delle varie classi costituenti.

ModelInterface: al suo interno troviamo una lista di metodi che saranno implementati dalla classe Match. Sono dei metodi relativi alle sei fasi della partita: LOBBY, PREPARATION, PLAYING, TERMINATING (raggiunta quando un giocatore ottiene almeno 20 punti o quando entrambi mazzi di carte si esauriscono), LAST_TURN e TERMINATED.

Match: classe che si occupa di gestire il comportamento del gioco nei vari status e nel quale vengono salvate le informazioni essenziali della partita: le carte sul tavolo, i mazzi dal quale i giocatori possono pescare, gli obiettivi comuni, i giocatori, lo stato della partita, ID del giocatore che sta giocando e il numero del turno.

Deck: i mazzi sono classificati in CardDeck e NonObjectiveCardDeck (si veda in seguito la divisione delle carte); questi vengono utilizzati da Match per popolare e mescolare le carte sul tavolo.

Player: in una partita, ad ogni giocatore viene associato un Field e un nickname univoco. Ha il compito di: inizializzare il campo, scegliere l'obbiettivo segreto (tra due carte date dal sistema), ricevere le carte pescate e posizione le carte nel proprio campo; inoltre, dopo ogni mossa, aggiorna il proprio punteggio utilizzando la pointStrategy ed il valore associati a ogni carta.

Field: è l'area di gioco; contiene tutte le informazioni relative alle carte posizionate: Field, tramite la classe CardPlaced, memorizza le coordinate, la faccia delle carte appoggiate.

Card: ogni card possiede un'ID, un value e una pointStrategy che descrive il modo in cui vengono calcolati i punteggi: si prenda in esempio una carta che restituisce 2 punti per ogni angolo che riesce a coprire; questa avrà un value di 2 e una PointStrategy "AnglesCovered" (vedi seguito) che, invocato il metodo "calculateOccurences" contenuto, ritornerà il numero di N angoli coperti. Il Player ottiene i punti che questo piazzamento ha generato mediante il prodotto 2*N.

Le carte sono genericamente divise in Card (obbiettivo e obbiettivo segreto) e NonObjectiveCard (risorsa, oro e carta iniziale). Le NonObjectiveCard sono una particolarizzazione di Card: nella nostra implementazione tutte le carte hanno una PointStrategy, dunque, non si rende necessario una classe ObjectiveCard perché tutte le sue funzionalità sono già state implementate, da Card, mediante la PointStrategy. Le carte piazzabili sono più descrittive delle carte obbiettivo.

PointStrategy: questa classe si occupa di implementare i controlli da effettuare sullo stato di gioco al fine di ritornare un “moltiplicatore di punteggio” utilizzabile dal Player per calcolarsi il punteggio dato dal piazzamento di una carta o dalla realizzazione di un dato obbiettivo; è suddivisa in nove categorie principali:

1. Empty: utilizzato per tutte quelle carte che danno un punteggio costante; non effettua alcun controllo e ritorna un coefficiente di 1.
2. AnglesCovered: ritorna il numero di angoli che una data carta riesce a coprire.
3. CountResource: ritorna il numero di risorse, eventualmente normalizzato (se una carta assicura 2 punti per ogni 3 risorse specifiche, la strategia produrrà la parte intera della divisione per 3 del numero di risorse cercato) richiesto dalla carta.
4. Diagonals: per tutte quelle carte che assegnano punti per ogni tre carte dello stesso regno collegate diagonalmente; ritorna il numero di pattern validi riscontrati nel Field.
5. AllSpecial: per le carte che necessitano il possesso di tutti e tre gli oggetti; si tratta di una particolarizzazione di CountResource.
6. LConfigurationOne: per le carte obbiettivo dove una carta del regno Pianti e due carte del regno Insetti formano una configurazione “L”.
7. LConfigurationTwo: per le carte obbiettivo dove una carta del regno Insetti e due carte del regno Pianti formano una configurazione “L” rovesciata.
8. LConfigurationThree: per le carte obbiettivo dove una carta del regno Animali e due carte del regno Insetti formano una configurazione “L” girata di 180 gradi.
9. LConfigurationFour: per le carte obbiettivo dove una carta del regno Funghi e due carte del regno Animali formano una configurazione “L” invertita.

model

model.field

Field

- fieldCards : ArrayList<CardPlaced>
- activeRes : int[7]

+ placeCardInField(NonObjectiveCard, int, int, boolean): boolean
+ getCardFromPosition(int, int): NonObjectiveCard
+ avaliableSpace(int, int): boolean
+ getFieldCards(): ArrayList<CardPlaced>
+ getActiveRes(int1 int

CardPlaced

- placedCard : nonObjectiveCard
- x : int
- y : int
- isUp : boolean

+ getCard(): nonObjectiveCard
+ getX(): int
+ getY(): int
+ getIsUp(): boolean

model.player

Player

- nickname : string
- gameField : Field
- chosenSecretObj : Card
- colour : Colour
- points : int
- hand : ArrayList<nonObjectiveCard>
- tempSecretObj : Card[2]

+ assignStartingCard(NonObjectiveCard): boolean
+ initializeGameField(boolean): boolean
+ secretObjectiveSelection(int): boolean
+ receiveSecretObjective(Card, Card): int
+ putCardInHand(NonObjectiveCard): boolean
+ performMove(NonObjectiveCard, int, int, boolean): boolean
+ updatePointsForObjectives(Card[]): boolean
+ calculatePoints(Card, int, int): int
+ getNickname(): String
+ getField(): Field
+ getSecretObjective(): Card
+ getColour(): Colour
+ getPoints(): int
+ getHand(): ArrayList<NonObjectiveCard>
+ getInitialCard(): NonObjectiveCard
+ setPoints(int): boolean
+ setColour(Colour): boolean

1...4

«enumeration»
Colour

Red
Green
Black
Blue
Yellow

<<use>>

model.match

1

Match

- starterCardsDeck : NonObjetiveCardDeck
- objectiveCardsDeck : CardDeck
- resourceCardsDeck : NonObjetiveCardDeck
- goldCardsDeck : NonObjetiveCardDeck
- currentResourceCards : NonObjetiveCard[2]
- currentGoldCards : NonObjetiveCard[2]
- commonObjectives : Card[2]
- players : ArrayList<Player>
- matchStatus : MatchStatus
- currentPlayerID : string
- currentTurnNumber : int

<<*enumeration*>>
MatchStatus

Lobby
Preparation
Playing
Terminating
Last_Turn
Terminated

<<*Interface*>>
ModelInterface

+ enterLobbyPhase(): boolean
+ addPlayer(string): boolean
+ deletePlayer(string): boolean
+ enterPreparationPhase(): boolean
+ assignRandomColoursToPlayers(): boolean
+ assignRandomStartingInitialCardsToPlayers(): boolean
+ getInitialCardPlayer(string): int
+ createFieldPlayer(string, boolean): boolean
+ assignRandomStartingResourceCardsToPlayers(): boolean
+ assignRandomStartingGoldCardsToPlayers(): boolean
+ pickRandomCommonObjectives(): boolean
+ assignRandomStartingSecretObjectivesToPlayers(): boolean
+ getSecretsObjectivesCardsPlayer(string): ArrayList<Int>
+ receiveSecretObjectiveChoiceFromPlayer(string, int): boolean
+ randomizePlayersOrder(): boolean
+ enterPlayingPhase(): boolean
+ startTurns(): boolean
+ placeCard(int, int, int, boolean): boolean
+ drawCard(int, int): boolean
+ nextTurn(): boolean
+ setTerminating(): boolean
+ areWeTerminating(): boolean
+ isFirstPlayer(): boolean
+ setLastTurn(): boolean
+ enterTerminatedPhase(): boolean
+ addObjectivePoints(): boolean
+ getWinners(): ArrayList<string>
+ getPlayersNicknames(): ArrayList<string>
+ getCurrentResourcesCards(): ArrayList<int>
+ getCurrentGoldCards(): ArrayList<int>
+ getCommonObjectives(): ArrayList<int>
+ getPlayerResources(string): ArrayList<int>
+ getPlayerSecretObjective(string): int
+ getPlayerHand(string): ArrayList<int>
+ getPlayerField(string): ArrayList<[]>
+ getMatchStatus(): string
+ getCurrentTurnNumber(): int

model.card

Card

- id : int
- value : int
- pointStrategy : PointStrategy

+ getStrategy(): PointStrategy

Extends

NonObjectiveCard

- topLeft : CornerType
- topRight : CornerType
- bottomLeft : CornerType
- bottomRight : CornerType
- topLeftBack : CornerType
- topRightBack : CornerType
- bottomLeftBack : CornerType
- bottomRightBack : CornerType
- kingdom : ObjectType
- permRes : int[4]
- conditionCount : int[4]

+ getKingdom: ObjectType

«enumeration»
CornerType

Plant
Fungi
Animal
Insect
Quill
Inkwell
Manuscript
Empty
NonCoverable

model.card.pointstrategy

«abstract»
PointStrategy

+ calculateOccurences(Field, int, int): int

Extends

Empty
+ calculateOccurences(Field, int, int): int

AnglesCovered
+ calculateOccurences(Field, int, int): int

CountResource
- type : ObjectType
- count: int

+ calculateOccurences(Field, int, int): int

Diagonals
- type : ObjectType
- leftToRight : boolean

+ calculateOccurences(Field, int, int):int

AllSpecial
+ calculateOccurences(Field, int, int): int

LConfigurationOne
+ calculateOccurences(Field, int, int): int

LConfigurationTwo
+ calculateOccurences(Field, int, int): int

LConfigurationThree
+ calculateOccurences(Field, int, int): int

LConfigurationFour
+ calculateOccurences(Field, int, int): int

«enumeration»
ObjectType

Plant
Fungi
Animal
Insect
Quill
Inkwell
Manuscript

Deck

CardDeck
- cards: ArrayList<Card>

+ draw(): Card
+ shuffle(): boolean

DecksFromDisk
+createResourcesDeck(): ArrayList<nonObjetiveCard>
+createGoldDeck(): ArrayList<nonObjetiveCard>
+createObjectiveCardsDeck(): ArrayList<Card>
+ createStartCardsDeck(): ArrayList<nonObjetiveCard>

NonObjetiveCardDeck
- cards: ArrayList<nonObjetiveCard>

+ draw(): nonObjetiveCard
+ shuffle(): boolean