

Progetto di Ingegneria del Software

Spiegazione dell'UML per peer-review

Descrizione della classe Game

L'idea alla base della realizzazione della logica di gioco consiste nella suddivisione in quattro classi: la classe Game, la classe Round e le classi PianificationPhase ed ActionPhase.

Nella classe Game verranno creati i giocatori e la *GameTable*, e verrà istanziato un oggetto della classe Round, aggregata alla classe Controller, che tramite il metodo *startRound()* farà incominciare la partita. *startRound()* creerà due istanze, una per la classe *PianificationPhase* ed una per la classe *ActionPhase*, codificate rispettivamente con 0 ed 1. La fase corrente verrà mantenuta nell'attributo *currentPhase*: quando una fase è abilitata i giocatori non potranno richiamare metodi appartenenti all'altra fase.

Si incomincerà quindi dalla fase di Pianificazione. Viene calcolato il primo giocatore tramite *calculateFirstPlayer()*, questo dovrà innanzitutto piazzare gli studenti sulle nuvole, tramite *putStudentsOnCloud()*. In seguito, in ordine, ogni giocatore dovrà giocare una carta assistente: il giocatore 1 chiamerà il metodo *playAssistant(int player, int assistant)*, per verificare che è stato effettivamente il primo giocatore a richiamare il metodo, si farà un controllo di uguaglianza tra il player passato come parametro al metodo ed il *currentPlayer* salvato, che verrà modificato di volta in volta dal metodo privato *calculateNextPlayerPianification()*. Onde evitare che ogni player possa richiamare lo stesso metodo più volte di quanto gli è consentito fare, ogni volta che si verifica l'autenticità del giocatore, questo verrà registrato nell'array di interi *alreadyPlayed* (*forse non è necessario ma ne stiamo ancora valutando l'implementazione*).

Una volta che tutti i giocatori avranno giocato il proprio assistente, la fase di pianificazione terminerà e verrà switchata la flag di *currentPhase*: inizia ora l'*ActionPhase*. Verrà calcolato a seconda degli assistenti giocati chi sarà ad incominciare, chiamando il metodo *calculateFirstPlayer()*. Ora il giocatore in turno dovrà fare tre mosse per decidere dove spostare gli studenti presenti sulla sua entrance, per verificare che ognuno faccia precisamente tre mosse utilizzo l'array di interi *movesCounter*, facendo sì che al player 1, associato all'indice 0 dell'array, ogni volta che chiama un metodo tra *moveStudentOnTable()* e *moveStudentOnIsland()*, venga incrementato di uno il contatore: quando il suo contatore raggiunge il valore di 3, allora non potrà più chiamare metodi *move..()*. Questo vale per tutti i giocatori e, come sopra, per verificare la loro autenticità si fa un check di uguaglianza tra *playerOnTurn* e il parametro passato al metodo *move..()*. Ora il giocatore in turno dovrà muovere Madre Natura, il meccanismo è il medesimo spiegato in precedenza. Quando un giocatore ha tutte le flag dei metodi alzate, allora avrà terminato il suo turno. Il giocatore successivo viene calcolato con *calculateNextPlayerAction()*. Quando tutti i giocatori avranno terminato il loro turno, allora sarà terminata la *ActionPhase*, e di conseguenza il *Round*. Ogni volta che terminerà un Round si verificherà la presenza di un vincitore.

Se nessun giocatore ha vinto, verrà istanziato un nuovo oggetto *Round*, separato da quello precedente, così che il gioco possa proseguire come sopra anche se con dati diversi, salvati nel corso del round precedente all'interno del Model.

Stiamo considerando anche una soluzione che preveda la presenza di un valore intero (o di altro tipo), già presente nell'UML, che segnali lo stato di esecuzione della classe Round in modo tale da poter effettuare i controlli sulla possibilità di chiamare un dato metodo in un preciso istante di esecuzione. Dato un valore di *roundState* potranno essere eseguiti solo certi metodi.

Nell'UML mancano alcuni metodi che in Java stiamo implementando.

Questa è soltanto una spiegazione dell'idea implementativa plausibile secondo la quale è stato realizzato l'UML. Come è ovvio, quando avremo completato l'implementazione java sarà anche meglio definito l'UML.

Descrizione del model

Le classi principali del modello sono Game, Player e GameTable, tramite i loro metodi gestiscono le chiamate da parte del controller. GameTable è il tavolo di gioco e gestisce anche una serie di funzionalità che possono risolversi da sole, ad esempio il calcolo dell'influenza e il posizionamento/cambiamento di una torre conseguente a una dominanza da parte di un giocatore su quell'isola. Anche l'unione delle isole è in parte gestita dalla GameTable che possiede la lista di isole sul tavolo.

L'unione delle isole viene gestita decorando due Island che diventano una MergedIsland, eventuali metodi che vengono chiamati sulle MergedIsland chiamano in modo ricorsivo i metodi ottenendo il risultato richiesto.

Le classi Advanced implementano le funzionalità relative alla modalità Expert del gioco.