Documento integrativo UML

Riccardo Piantoni, Matteo Rossi, Federico Pinto, Jacopo Sacramone
Gruppo 12

26 marzo 2024

Il presente allegato all'UML descrive alcune delle scelte progettuali e implementative prese, ponendo particolare attenzione ai motivi e alle conseguenze di ognuna. L'UML presentato mostra la struttura server-side del Model, orientato all'attuale ipotesi di realizzazione delle seguenti funzionalità aggiuntive:

- **Partite multiple**: la gestione di più partite sul server è stata realizzata creando le classi GameLobby e Game, che modellano le diverse istanze di una lobby di attesa e di una partita avviata;
- **Persistenza**: il salvataggio dei dati viene effettuata all'interno della funzione persistence(), che viene chiamata all'interno di ogni esecuzione di transition() nelle varie sottoclassi di GameState e cioè ad ogni cambiamento di stato della partita;
- **Chat**: la funzionalità di chat non impatta la rappresentazione del Model, poiché è stato scelto per il momento di non mantenere una cronologia dei messaggi, ma solo di gestirne l'inoltro verso i destinatari in real-time.

1 Descrizione delle classi

1.1 Package Cards

- **Card** Classe astratta che racchiude le informazioni comuni alle carte disponibili. Ogni istanza di carta è una sua sottoclasse.
  - **ObjectiveCard** Rappresenta le carte Obiettivo presenti nel gioco.
  - **PlayableCard** Classe astratta che introduce le informazioni relative agli angoli, rappresentati come coordinate relative rispetto alla posizione della carta. Solo istanze delle sue sottoclassi possono essere posizionate sul campo.
    - **GoldCard** Introduce le informazioni addizionali di una carta Oro, ovvero le risorse da possedere per giocare la carta ed eventuali condizioni da soddisfare per ottenere i punti dal piazzamento.
    - **InitialCard** Rappresenta le carte Iniziali piazzabili sul campo.
    - **ResourceCard** Rappresenta le carte Risorsa presenti nel gioco.
- **CardDeck** Un mazzo di carte generiche (Card), implementato tramite delega ad uno Stack interno avente un ridotto insieme di operazioni permesse.

1.2 Package Conditions

- **PointsCondition** Un'interfaccia comune a tutte le condizioni. Una condizione identifica un metodo di assegnamento dei punti di una determinata carta, astratto dalla carta stessa per poter riutilizzare i tipi di condizione. Ogni classe Condition deve implementare questa interfaccia, e quindi definire un metodo il quale, al momento della valutazione per la carta a cui si riferisce, conta il numero di volte in cui la condizione viene soddisfatta dai dati di un determinato InGamePlayer. Inoltre, ognuna delle classi che implementano questa interfaccia può liberamente appoggiarsi a dati interni per la valutazione della condizione, memorizzati insieme ad altri dati della carta nel file delle carte.
  - **CornerCondition** Condizione per cui ogni angolo coperto da una carta contribuisce all'attribuzione dei punti. Non necessita di dati interni, in quanto si affida solo alla posizione della carta e a ciò che le sta intorno.

- **ResourceCondition** Condizione per cui il possedimento di determinate risorse contribuisce all'attribuzione dei punti. Necessita di memorizzare il set di risorse di cui conteggiare la quantità posseduta, in modo da poterlo confrontare con le risorse dal giocatore che la piazza sul terreno.
- **PatternCondition** Condizione per cui un determinato pattern presente sul campo di gioco contribuisce all'attribuzione dei punti. Necessita di memorizzare le informazioni del pattern, rappresentato come lista di triplette di elementi: per ogni carta che compone il pattern viene memorizzata una coppia di interi come coordinate (definite come offset relativo rispetto ad una generica posizione del terreno) e una risorsa per identificare il colore della carta.

1.3 Package GameStates

- **GameState** Una classe astratta che contiene TUTTE le possibili azioni eseguibili dai giocatori in qualsiasi fase della partita. Ciascun metodo lancia una ForbiddenActionException. Ogni sottoclasse State ridefinirà solamente le azioni che sono valide nel suo contesto, inclusa per ogni State la funzione di transizione.
  - **SetupState** Svolgimento delle operazioni di preparazione della partita in fase in avvio.
  - **ChooseInitialCardState** Distribuzione delle carte iniziali ai giocatori e piazzamento secondo la loro scelta.
  - **DrawStartingHandState** Distribuzione della mano ai giocatori.
  - **ChooseObjectiveCardState** Distribuzione delle due carte obiettivo ai giocatori e selezione secondo la loro scelta.
  - **PlayerTurnPlayState** Fase di piazzamento di una carta dalla mano del giocatore corrente sul campo di gioco. I giocatori non in turno sono impossibilitati nel compiere qualsiasi azione relativa alla partita. Inoltre, imposta il counter dei turni mancanti in caso di innesco della fase finale della partita.
  - **PlayerTurnDrawState** Fase di pesca dai mazzi o dalle carte visibili da parte del giocatore corrente. E' valida la considerazione al punto precedente. Inoltre, imposta il counter dei turni mancanti in caso di innesco della fase finale della partita.
  - **VictoryCalculationState** Conteggio finale dei punti e proclamazione dei vincitori.

1.4 Package Utilities

Comune a tutti i packages del progetto.

- **Side** Indica il lato su cui la carta è giocata sul campo.
- **Color** Possibili colori selezionabili dai giocatori, compreso il gettone del giocatore iniziale.
- **Resource** Qualsiasi tipologia di risorsa che compare all'interno del gioco.
- **GenericPair** Implementazione di una coppia di oggetti generici.
- **Triplet** Implementazione di una tripletta di oggetti generici.
- **JSONParser** Implementazione personalizzata per il caricamento delle carte da gioco da file *.json*.

1.4.1 Package Exceptions

1.5 Altre classi

- **GameLobby** Una lobby per raccogliere i giocatori in attesa dell'inizio di una partita.
- **Game** Evoluzione di una GameLobby a partita iniziata, è il punto di contatto per il quale è necessario passare per interagire con il Model dall'esterno, e permette di farlo nei soli modi permessi dai suoi metodi.
- **Player** Un giocatore al di fuori di una partita.
- **InGamePlayer** Evoluzione di un Player, contiene tutti i dati e le statistiche (compreso il campo da gioco) di un giocatore all'interno di una partita, per i quali è l'unico punto di accesso che consente di modificarli provenendo dal Game.

- **Field** Astrazione logica di un campo da gioco: le carte piazzate sono identificate dalle coordinate cartesiane. La carta iniziale è posizionata in <0, 0>. Inoltre, nell'ottica di restringere l'insieme di posizioni in cui giocare le carte successive, memorizza anche le coordinate valide da regolamento, che sono di volta in volta aggiornate e comunicate ai giocatori.

2 Scelte progettuali

Per progettare la struttura del software di gioco, si è cercato di seguire il più possibile le best-practices di design e scrittura del codice. Le principali guide durante il progetto sono state:

- Il principio **DRY**, ovvero l'attivo evitare la ripetizione di codice in punti differenti (per esempio all'interno di classi affini per le quali sarebbe stato necessario implementare un'ereditarietà multipla, cercando invece soluzioni alternative per riusare il codice, aumentare la manutenibilità e facilitare la possibile aggiunta di funzionalità come carte di diverso tipo, diverse condizioni di valutazione del punteggio, nuove tipologie di pattern da riconoscere nelle carte obiettivo).
- Il conseguente principio **ETC**, secondo il quale mantenere i componenti (nel nostro caso le classi) altamente ortogonali e disaccoppiati tra loro consente di ottenere una struttura molto flessibile, e quindi di apportare successivamente cambiamenti con un impatto ridotto, meno drastico e in maniera semplificata.

2.1 Pattern utilizzati

- **"Strategy"**: per l'implementazione facilmente espandibile e disaccoppiata della valutazione delle condizioni di assegnamento dei punteggi e di giocabilità delle carte Oro, è stato attuato uno pseudopattern Strategy, con un'unica differenza: mentre nel pattern canonico la strategia può essere modificata a runtime, in questo caso la Condition è specificata nel file JSON delle carte e impostata unicamente all'avvio del server.
- **State**: il pattern State è stato implementato per la gestione delle fasi di una partita, così da facilitare il salvataggio dei dati ad ogni transizione di stato; inoltre, ha il beneficio di permettere di definire nella classe astratta GameState tutti i metodi invocabili da un client con il lancio di una ForbiddenActionException, facendo override nelle sottoclassi dei soli metodi eseguibili in quel determinato stato senza dover ripetere ogni volta la gestione dei permessi (conformemente al principio DRY).

2.2 "Fat Model"

La ferma convinzione che le responsabilità debbano essere divise, e che nessuno debba poter accedere a dati nè avere il permesso di effettuare operazioni che non gli competono, ci ha portati a togliere gran parte della logica dal Controller, delegandola ad ognuna delle classi del Model stesso. Ciò ha portato come naturale conseguenza la realizzazione di un Controller snello, il quale si limita a collegare la ricezione dei comandi tramite Socket/RMI al Model, "inoltrando" la chiamata delle azioni ai punti di accesso del Model stesso.

Nota - Visibilità dei metodi Il linguaggio Java non presenta un meccanismo agevole per definire la visibilità di metodi verso subpackages appartenenti al medesimo package (ServerModel). Nel tentativo di mantenere una suddivisione logica delle classi che costituiscono i subpackages (Cards, GameStates, ...), si è reso necessario esporre come *public* i metodi chiamati da altre classi del model. Le best-practices suggeriscono invece l'utilizzo di visibilità *protected* su metodi che non sono esposti al di fuori del package, ma in questo contesto ciò risulta essere inapplicabile.

3 Contatti

In caso siano necessari ulteriori chiarimenti, vi invitiamo a contattare:
riccardo.piantoni@mail.polimi.it.