Sagrada
Dettagli Implementativi

Lambertucci Michele, Mancassola Mattia e Losavio Andrea

1 MODIFICA CONTENUTO CASELLE

L'interfaccia `DieCoord` rappresenta una casella in un certo *container* [1].

L'unico modo per cambiare il contenuto di una cella (rimuovere o inserire un dado) è passando attraverso un oggetto `DieCoord`, tramite le funzioni `void set(Die die)` e `Die get()`.

L'interfaccia ha anche la funzione `PlacementError isDieAllowed(Die die)`, che restituisce quali sarebbero gli errori di piazzamento se volessimo inserire un dado all'interno di quella certa casella.

`PlacementError` è strutturato come un `Set` di errori.

2 PLAYER STATE

Il `player state` di un giocatore, immagazzinato come attributo privato all'interno di ogni oggetto `Player`, viene stabilito dal Server ad ogni turno, e indica ≪cosa deve fare il giocatore nel turno attuale≫. In poche parole la *View* reagirà ad un cambiamento di `player state`, ad esempio chiedendo all'utente quale dado vuole spostare.

I possibili stati sono:

1. `IDLE` - Il giocatore è in attesa del proprio turno;
2. `YOUR_TURN` - Il giocatore può scegliere se attivare una toolcard, muovere un dado normalmente (da DraftPool a Board) o passare;
3. `PICK` - Il giocatore deve scegliere una casella di un certo container.
4. `VALUE` - Il giocatore deve scegliere un valore tra 1 e 6 (usato in alcune toolcards);
5. `UPDOWN` - Il giocatore deve scegliere se aumentare o diminuire il valore di un certo dado (usato in alcune toolcards);
6. `YESNO` - Il giocatore deve scegliere se continuare con la prossima mossa della toolcard (usato quando una toolcard permette di fare due movimenti);
7. `REPEAT` - Il giocatore deve ripetere l'ultima mossa che ha fatto perché non è stata giudicata valida dal Server.

Di seguito vengono indicati alcuni dettagli di alcuni stati.

2.1 Your Turn

Quando un giocatore si trova in `YOUR_TURN`, potrebbe avere disponibili tra una e tre possibili azioni da compiere tra *attivare una Toolcard*, *compiere una mossa normale* (prendere un dado dalla draftpool e metterlo all'interno di una casella della propria board) e *passare*. Non necessariamente tutte e tre queste mosse sono possibili durante il corso del gioco, e soprattutto **non si può compiere la stessa azione due volte nello stesso turno**.

Per questo motivo le azioni possibili sono immagazzinate all'interno di `Player` in un attributo di tipo `Set<PossibleAction>`, dove `PossibleAction` è un semplice `Enum` contenente dei valori che rappresentano le tre azioni: `ACTIVATE_TOOLCARD`, `PICK_DIE`, `PASS_TURN`.

Una volta che il giocatore avrà scelto cosa fare, il client invierà un messaggio corrispondente all'azione scelta al Server, che avvierà una procedura, descritta con cura in 3.

2.2 Pick State

Questo stato rappresenta una qualsiasi situazione in cui il Player deve scegliere una casella da un certo contenitore (DraftPool, RoundTracker o Board).

A differenza degli altri *states* (che sono tutti contenuti in un oggetto `PlayerState`), per poter rispondere a domande come ≪da quale contenitore devo scegliere?≫ o ≪devo scegliere una casella vuota o una casella piena?≫ è stato necessario estendere la classe `PlayerState` in `PickState`, classe atta a contenere informazioni sulla casella da selezionare.

[1] Le classi contenitore sono `Board` e `DiceContainer`, di cui l'ultima delle due è usata per implementare sia *RoundTracker* sia *DraftPool*

3 DIALOGO INTRA-TURNO TRA CLIENT E SERVER

All'interno di un singolo turno, il giocatore può compiere una serie di azioni. Abbiamo cercato di rendere il più possibile modulari queste azioni, di modo da rendere semplice aggiungere nuove *Toolcards* o modificare le regole stesse del gioco.

Ad esempio l'azione di prendere un dado dalla *DraftPool* e posizionarlo in una casella della propria *Board* è divisa in due fasi, rappresentate da due `PickState` (vedi 2.2):

1. `PickState: ActiveContainers = {DRAFTPOOL}, CellStates = {FULL}` [2]

2. `PickState: ActiveContainers = {BOARD}, CellStates = {EMPTY, NEAR}`

Il client riceverà un primo messaggio, contenente il `PickState` 1, e di conseguenza avvierà una procedura nella *View* in cui l'utente dovrà scegliere una casella `FULL` della *DraftPool*.

A questo punto il client invierà un messaggio contenente le coordinate della casella scelta dal giocatore. Il server elaborerà il messaggio e aggiornerà lo stato del Player al secondo `PickState`.

In caso il giocatore selezioni una casella inaccettabile (ad esempio avente una restrizione non compatibile con il dado che vuole piazzare), il server aggiornerà lo stato del Player a `REPEAT`. Il giocatore potrà quindi scegliere un'altra casella.

3.1 Actions

Ogni azione, che si tratti di spostare un dado, di cambiarne il valore o altro, implementa l'interfaccia `Action`.

La funzione `PlacementError check()` è una funzione che restituisce quali sarebbero gli errori di piazzamento se l'azione venisse eseguita. Internamente, ogni implementazione di questo metodo utilizza vari `DieCoord.isDieAllowed` (vedi 1).

Questa funzione ci permette di **scoprire** in maniera molto semplice **se una determinata azione possa effettivamente essere eseguita** a seconda della situazione. Ad esempio molte *Toolcards* permettono azioni che violano alcune delle regole di piazzamento, e utilizzando questa funzione è possibile fare un check e filtrare gli errori.

[2] `FULL` indica che la casella selezionata deve contenere un dado

3.2 Toolcards

Le *Toolcards* sono rappresentate come una serie di operazioni e di `PlayerState`s che vengono assegnati in ordine sequenziale al player che attiva la Toolcard. Nella nostra implementazione abbiamo, per ogni toolcard, una `Queue` di `BiFunction<ToolcardController, PlayerMove, PlayerState>`. Tralasciando il primo argomento, che è necessario per salvare informazioni tra un'operazione e la successiva [3], un'operazione prende come argomento la `PlayerMove` inviata dal giocatore (contenente ad esempio la casella selezionata) e restituirà il nuovo stato da assegnare al Player.

Questo stato può essere anche `REPEAT`, in caso che `PlayerMove` violi le regole della Toolcard.

La logica della toolcard risiederà quindi all'interno delle singole operazioni.

Le Queue stesse sono infine immagazzinate all'interno di una `static final Map<Integer, Queue<...>>`, che mappa da id della toolcard a Queue di operazioni. In questo modo, per aggiungere una toolcard basta aggiungere una nuova `Queue` di operazioni a questa mappa.

[3] Queste funzioni, chiamate da un `ToolCardController`, avranno sempre come primo argomento `this`, potendo così accedere ad una lista interna all'oggetto TCC chiamato memory, dove verranno salvate le `DieCoord` da recuperare nelle operazioni successive