Peer-Review 1: UML

Letizia Maria Chiara Torre, Riccardo Speroni,
Vlad Alexandru Robu, Davide Vola

Gruppo GC20

2 aprile 2023

Valutazione del diagramma UML delle classi del gruppo GC10.

1 Lati positivi

L'utilizzo delle coordinate semplifica di molto la ricerca e il confronto delle `Tiles`, sia per quanto riguarda le carte obiettivo che per controllare se una determinata tessera nel tabellone sia `pickable` o meno guardandone i lati liberi. Degno di nota è l'utilizzo in `Personal` di un `ArrayList<Cell>`, che contiene una lista di tipi di tessere con le rispettive coordinate e che permette un confronto immediato con la `Shelf` semplicemente usando la coordinata di ogni `Cell` e vedere se in tale posizione c'è lo stesso tipo di tessera.

2 Lati negativi

Non è chiaro il senso di scrivere `tile...type` e/o `int...coord`: se si tratta di un `Set` o di una lista di oggetti basta scrivere, ad esempio, `ArrayList <Coordinate>`.

2.1 Shelf

Ci dovrebbe essere un metodo che conta gli spazi liberi di ogni colonna e ne ritorni il massimo, cosí che il giocatore non possa selezionare più tessere dal

tabellone di quante sia possibile inserirne nella `Shelf` (ad esempio, se tutte le colonne della shelf sono piene tranne per una, che ha solo una casella libera, il giocatore non potrá prelevare tre tessere dal tabellone, nemmeno se sono tutte `pickable`).

2.2 Player

`Player` non ha bisogno di righe e di colonne, perché ognuno ha già la sua mensola con le stesse informazioni, quindi sarebbe ridondante.
`pShelf`, `firstPlayer` e `nickname` non dovrebbero essere `readOnly` perché vanno inizializzate - e, nel caso della mensola, vanno modificate nel corso del gioco.

Il metodo `getPersonalGoal` dovrebbe restituire `pGoal` al posto di `Personal` (questo attributo infatti non esiste nella classe `Player`).
Il metodo `assignShelf` non è necessario, visto che ognuno ha giá la sua `Shelf` e non occorre assegnargliene una, e comunque non servirebbe a granché restituire un `int` in tale metodo.

Se `countAdjacent` serve a sapere quante tessere adiacenti ci sono nel tabellone, sarebbe meglio se fosse allora un metodo pubblico, o comunque che fosse un metodo della `Board` stessa (non è il giocatore a vedere se può prendere una determinata tessera, ma è il modello che decide o meno se una tessera è `pickable`, e di conseguenza a permettere al giocatore di selezionarla); se invece conta le tessere adiacenti nella `Shelf` allora è superfluo, anche se potrebbe servire per verificare una carta obiettivo comune, e allora in quel caso dovrebbe essere pubblico ed appartenere ad una delle classi delle carte obiettivo. In ogni caso, basta che abbia come argomento il riferimento a una specifica `Tile`, oppure una coppia di coordinate.

Manca il metodo che permette al giocatore di scegliere le tessere dal tabellone (che restituisce una lista di tessere) per metterle nella `Shelf`, e un metodo che le possa riordinare secondo i gusti del `Player`.

2.3 Bag

Se `Bag` contiene le tessere rimanenti, perché dovrebbe avere un attributo `removed` che restituisce una sola `Tile`? La lista di `unplacedTiles` ha de-

cisamente senso, perché si tiene traccia di tutte le tessere rimaste e che si possono ancora utilizzare, ma non occorre un ulteriore attributo che mi dice quali tessere sono state rimosse.

Se `pickTiles(int)` si occupa di pescare un determinato numero di tessere dalla `Bag`, allora deve restituire una lista o un `set` di `Tile`, invece che una sola.

2.4 Board

Non è chiaro il senso di `placeHolder`, probabilmente si può fare a meno di questo attributo: se infatti si riferisce a oggetti di tipo `TileSet` - costituendo quindi le caselle invalide - si può eliminare e mettere invece i `TileSet` nelle giuste coordinate di `spaces`. Non ha inoltre senso imporre che gli `spaces` siano `readOnly` se sopra devo metterci delle tessere: devo poter modificare il contenuto degli `spaces` mettendo o togliendo tessere, o settando delle tessere come `non-pickable`.

Tutti i metodi che controllano se un determinato lato sia vuoto si possono comprimere in un unico metodo, cioé `atLeastOneSideEmpty`. Il metodo che controlla che ci sia almeno un lato occupato non è necessario, ma si può tenere per testare che la matrice sia riempita a dovere, cosí come anche `printStatus`; non sono indispensabili, tuttavia, per il corretto funzionamento del gioco.

Non è chiaro perché `initSpaces` dovrebbe prendere come attributo una stringa; se si tratta di un metodo che inizializza l'intera matrice si potrebbe eliminare ed utilizzare in sua vece `refill()`.

2.5 Game

L'attributo booleano `winner` non dovrebbe essere in `Game`, ma nella classe `Player` (questo gli garantirebbe punti in più alla fine del gioco); se si vuole tenere un attributo `winner` nella classe `Game`, questo dovrebbe essere di tipo `Player`, così da poter identificare quale giocatore dovrá avere un tot di punti in più per aver riempito per primo la propria mensola.

Non è chiara la funzione degli attributi `playersFirstShared` e `playersSecondShared`:

potrebbero rappresentare quale giocatore ha preso la prima carta obiettivo comune e quale la seconda, ma in tal caso dovrebbero restituire un solo `Player` e non una `ArrayList`.
Lo stesso vale per i metodi `getter` omonimi.

`getWinner` dovrebbe restituire un `Player` e non un valore booleano, visto che serve a sapere chi è il vincitore, e non se ci sia stato o meno un vincitore. Se peró si sta considerando il caso in cui nessuno abbia vinto, basta solo restituire `null` al posto di `Player`.

2.6 Tile

Può darsi che l'utilizzo di una stringa per identificare l'immagine sulla tessera non sia la scelta più elegante rispetto a un semplice `int`, ma funziona lo stesso.

Come sono create le tessere? Sono generate automaticamente da un costruttore?

Inoltre, qual è il motivo di avere oltre all'attributo `isSettable` anche il metodo `isSettable()` se già esiste `setIsSettable` che sembra avere la medesima funzione?

2.7 TileSet

Forse il nome di questo oggetto potrebbe essere fuorviante, nonostante la funzione abbia senso. Potrebbe essere più immediato utilizzare `TileUnusable`.

2.8 Cell

Visto che esiste questa classe, non avrebbe più senso utilizzarla nella `Board` e nella `Shelf` ogni volta che bisogna definire le caselle, o spazi, al posto di `Tile[][]`? In questo modo avremmo anche le coordinate della specifica tessera oltre al tipo, e si potrebbe scrivere `type: Tile` al posto di `TileType` in `Cell`, così da unire due classi insieme.

2.9 Shared

Avrebbe decisamente senso imporre a tutte le sottoclassi di questa `abstract` di avere un solo metodo per il controllo del pattern con lo stesso nome di quella in `Shared` (`checkPattern(Shelf)`), cosí da fare override del metodo.

Di dubbia utilitá invece sono gli attributi `ROWS` e `COLUMNS`, visto che sono informazioni già contenute nella `Shelf`.

In generale, si potrebbe dire che tutte le sottoclassi di questa potrebbero essere di gran lunga semplificate se si fossero aggregate tra di loro: ad esempio, la condizione che ci sia una 'X' formata da tessere dello stesso tipo non è altro che il controllo di due diagonali speculari, cosí come il controllo che ci sia un quadrato di tessere uguali si potrebbe vedere come una coppia di colonne di altezza due. Notare questi ulteriori pattern potrebbe semplificare il codice e permettere maggior scalabilità al programma, quindi il nostro consiglio è di rivedere questa parte (che sappiamo bene essere la più difficile da implementare).

Attenzione: ci sono alcuni metodi nell'UML che dovrebbero essere `public` e che invece sono `private`. Meglio cambiare la visibilità.

2.9.1 VerticalHoriziontal

Questa intera classe non sembra avere granché senso; sembra un costruttore di pattern, ma se giá ogni classe di `Pattern` ha dentro i suoi metodi per fare quello che serve allora questa non ha grande rilevanza: gli attributi `type`, `length`, `countMatch`, `countOccurrency` e `pairTrack` possono essere integrati nella superclasse astratta (è presumibile che comunque tutti i pattern ne abbiano bisogno in un certo senso). In alternativa possono essere messi in un'ulteriore classe posta tra l'astratta `Shared` e le classi di pattern che hanno bisogno di tali attributi per funzionare.
Lo stesso discorso si può fare per `checkSameType`, che alla fine non è altro che il metodo `checkPattern` di `Shared` su cui avrebbe senso fare override.

`printList` può servire per eseguire test sul funzionamento della classe ma non è indispensabile ai fini del buon funzionamento del gioco.

Non è chiaro il funzionamento né il senso di nessuno dei metodi `delete` di questa classe.

`runPattern(Shelf)` sembrerebbe un metodo per vedere se il pattern è applicabile alla `Shelf`, o se ha dato un esito positivo, ma in quest'ultimo caso sarebbe superfluo, visto che già `checkPattern` di `Shared` ritorna un `boolean` con la medesima funzione.

2.10 Coordinate

Il metodo che controlla che le coordinate siano uguali non è necessario, visto che `equals` è una funzione già esistente nella libreria di Java.

3 Confronto tra le architetture

L'utilizzo di una classe coordinate potrebbe tornare utile anche al nostro programma, visto che, assegnando ad ogni casella una coppia `(x, y)` ci si risparmierebbe di scorrere la matrice della `Shelf` o della `Board`.