Peer-Review 1: UML

Berardinelli, Genovese, Grandi, Haddou
Gruppo AM21

April 3, 2024

Contents

Valutazione del diagramma UML delle classi del gruppo AM30.

1 Lati positivi

1.A Incapsulamento e modularità I lati positivi del design del modello del gruppo AM30 hanno a che fare sicuramente con una buona separazione delle responsabilità delle classi e dunque con un discreta piattaforma per memorizzare lo stato del gioco. Il modello segue in linea di massima il principio dell'incapsulamento: i dati sono modificabili e accessibili attraverso interfacce definite che fanno parte della classe che li memorizza come attributi. Ciò promuove la modularità e riduce le dipendenze di una parte dall'altra del modello.

1.B Chiarezza dell'architettura La struttura del modello è chiara: le componenti sono ben distinte e hanno la potenzialità di essere effettivamente funzionali se consideriamo singolarmente le loro possibili implementazioni.

1.C MVC Il modello rappresentato in diagramma è ben predisposto per il pattern *Model View Controller*.

2 Lati negativi

Raccogliamo qui una serie di punti, nella speranza che risultino d'aiuto al gruppo in sede di valutazione del progetto.

2.1 Criticità del diagramma UML

2.1.A Assenza costruttori delle classi Poiché sono assenti dal diagramma tutti i costruttori delle classi, non è facile capire quali attributi vengano popolati quando viene istanziato l'oggetto e quali vengano settati a posteriori. Questo talvolta ci ha reso difficile capire quali siano le relazioni tra le classi e dove siano i metodi che gestiscono alcune dinamiche di gioco.

2.1.B Assenza di metodi getter e setter In modo analogo a come descritto nel paragrafo **A**, l'assenza di *Getter* e *Setter* rende difficile capire come siano state progettate certe dinamiche di gioco. Il *Controller*, eventualmente, si dovrà interfacciare a classi del *Model* i cui attributi sono privati affinchè il player possa giocare il suo turno ed effettuare le sue scelte.

2.1.C Posizionamento di alcuni attributi Alcuni attributi delle classi in diagramma sono posizionati esclusivamente sulle frecce che li uniscono alle classi di cui sono istanza. Comunemente, tutti gli attributi sarebbero da riportare nel box della classe, in modo da riportarne anche la visibilità. Le frecce vanno utilizzate per rappresentare il tipo di relazione fra le classi. Alcuni di questi tipi di associazione sono riportate come *Aggregation* quando in realtà sarebbero *Composition* (vedere `CardPair` e i suoi attributi e `ComplexObjective` con `Layout` eccetera). A nostro parere questo approccio dà poca visione di classe.

2.1.D Nits

1. Il *return type* di tutte le funzioni `void` è assente.
2. Il *return type* di `Table.drawCard()` e `Table.drawObjective()` è assente
3. La cardinalità di `Table.resourceCards`, `Table.goldCards`, `Table.starterCards` è formalmente 0..* o 0..n
4. Nei metodi, la sintassi per elencare gli argomenti è talvolta errata: quella corretta è `foo(Bar bar,Car car):void`
5. In `Table.getDrawableCardsByType()`, se si tratta di un'API generica, manca nei parametri del metodo il numero di carte richiesto da restituire come *Array*.

2.2 Dinamiche di gioco non gestite

2.2.A Aggiunta del player al gioco Nel diagramma non è specificato come viene aggiunto l'oggetto `Player` a `Game.players`, dato che `Game.addPlayer()` non prende un `Player` come parametro.

2.2.B Gestione del ritmo di gioco In `Game` non sono presenti attributi che salvino lo stato corrente del gioco come ad esempio il `CurrentPlayer`, o il `GameState`. In particolare si potrebbe salvare se questo si trovi in uno stato di inizializzazione, se si trovi in attesa che i player si uniscano alla partita, se i player stiano giocando, se si tratti degli ultimi turni o se il gioco sia finito.

2.2.C Gestione dell'ultimo turno Nel diagramma non è specificato come viene gestito l'ultimo turno di gioco. Non sono presenti attributi in `Game` che dopo che un player abbia raggiunto un punteggio sufficiente a far terminare la partita e/o le carte siano finite, tiene conto del numero di turni o round rimanenti, prima che il gioco termini effettivamente con `Game.endGame()` e dunque siano valutati gli obbiettivi.

2.2.D Token di gioco Non sono previsti né in `Game`, né in `Player` attributi che riguardano la serie di `Token` assegnabili dal gioco e il colore del `Token` scelto dal giocatore.

2.3 Errori di modellazione

2.3.A Valori non accettabili assegnabili Da diagramma UML, gli attributi di `ComplexObjective.Layout` possono assumere come valori `Resource.PEN`, `Resource.INKWELL` e `Resource.MANUSCRIPT` quando questo in realtà nel gioco non accade. Noi suggeriremmo di separarle in due *Enum* differenti.

2.3.B Numero di carte in Layout Le cardinalità di `ComplexObjective.Layout` non tornano. Ogni carta ha un massimo di 2 `Layout.edges`, perché le carte a destra e sinistra di una carta sulla griglia di gioco sono sfalsate e sono già salvate in `Layout.corners`.

2.3.C Angoli "disabilitati" Non sono modellati gli angoli "disabilitati" delle carte, ovvero gli angoli che non sono `Resource.EMPTY` e cui non si può collegare una carta. La cardinalità di `Card.cornersFront` dovrebbe essere 0..4, e non 4, per questo motivo. Sebbene esista `Table.forbiddenPoints`, il fatto che la carta abbia un angolo disabilitato, per ora, non è salvato da nessuna parte.

2.3.D Nits

1. La cardinalità di `Player.hand` è di massimo 3.

2.4 Aree di miglioramento

Crediamo che per i seguenti aspetti di modellazione esista del margine di miglioramento:

2.4.A Riutizzabilità Gli oggetti collezionabili sono duplicati nelle *Enum* `Resource` e `Multiplier`, a nostro parere, come anticipiamo in 2.3.A, sarebbe meglio separare Risorse e Oggetti per renderli utilizzabili in modo indipendente utilizzando i generici di Java per utilizzarli in modo più flessibile.

2.4.B Ereditarietà Concettualmente le `Card` non sono tutte uguali. Ad esempio, non tutte le carte *possono* avere una `placementCondition`. Ad esempio, la `placementCondition` è opzionale per le *GoldCard*, ma una *ResourceCard* non la avrà mai. In quest'ottica, sarebbe meglio, a nostro parere, considerare opzionale un attributo che una certa categoria di carta *può* avere. Si potrebbe gestire meglio questo aspetto con un maggiore uso dell'ereditarietà.

2.4.C Opzionalità di alcuni attributi `Resource.EMPTY` non è un vero valore, gli attributi che lo usano potrebbero essere modellati in modo più elegante con un `Optional<Resource>`. O meglio, dato che UML è agnostico all'implementazione: `Resource[0..1]`.

2.4.D Ridondanza di alcuni riferimenti Il doppio riferimento a `Table` presente in `Game` e in `Player` è ridondante. La lista di `Player` si trova dentro già dentro `Game` che ha un unico `Table`, si può gestire meglio la situazione con dei *Getter* in `Game`, dato che si tratta di funzionalità probabilmente usate dal layer del *Controller*.

2.4.E Approccio talvolta poco orientato agli oggetti Pensiamo che il metodo `Table.shuffle()` ne sia un esempio. I mazzi di carte al posto che essere *Array* potrebbero essere rappresentati con un oggetto `Deck` che ha un metodo `Deck.shuffle()`. Tutto ciò che è effettivamente funzionale da solo potrebbe essere una classe a parte con i suoi metodi.

2.4.F Design patterns Non è riportato in diagramma alcun utilizzo di *creational pattern*. Pensiamo avrebbero la potenzialità di integrarsi bene nel

modello se le carte fossero meglio differenziate. Fra i modelli che potrebbero essere utilizzati vi sono il *Factory Pattern* e il *Builder Pattern*.

2.4.G Naming di alcune classi Ci permettiamo di dare alcuni suggerimenti sul naming di alcune entità o metodi.

1. La parola *Points* viene usata sia nei Set di `Coordinates` che in `Player.points`. Si potrebbe cambiare una delle due per evitare confusione

2. Il nome di `CardPair` non è veramente descrittivo di come è usata la classe. Il nome evoca una coppia di carte giocabili, ma contiene una `Card` sola.
   Vediamo, da come è usato in `Player.placeCard()`, che in realtà si tratta di una carta cui è stata scelta la `Side` da giocare. Ma allora perché `Pair`? Non si capisce, tra l'altro, dove venga chiamato il costruttore, dato che nessun metodo di Player ha come come parametro un `Side`.

3. C'è della ripetizione in alcuni nomi di metodi ed attributi. Ad esempio, per `Game.endGame()`, `Game.startGame()`, `Game.gameId` potrebbe essere più elegante usare `Game.start()`, `Game.end()`, `Game.id` eccetera.

4. I nomi delle realizzazioni di `Objective` non sono molto descrittivi. A nostro parere sarebbe meglio includere la loro natura dentro il nome. Si tratta di obiettivi di conto e di obbiettivi che hanno a che fare con la geometria del `Player.playingField`. Simple e Complex non dicono molto. Ma capiamo si tratti veramente di un nitpick.

2.5 Perplessità

1. Non è chiaro cosa faccia `Supervisor.compute()`.

2. Come mai `Card.getCorners()` è *Private*?

3 Confronto tra le architetture

3.A Scelte di design sulle responsabilità del modello L'approccio del gruppo AM30 è sicuramente diverso da quello scelto dal nostro gruppo. Alla base della differenza c'è la *design choice* che riguarda quanto "lavoro" affidare al *controller* che conterrà il *model*, vale a dire il server.

Come gruppo, abbiamo fatto la scelta di affidare il meno possibile al *controller layer*, offrendo una serie di classi che costruissero un gioco pronto a essere giocato, attraverso i quali metodi lo stato di gioco viene interamente salvato all'interno del modello: lasciando così al server solo la gestione delle comunicazioni con i client.

L'approccio del gruppo AM30 è diverso nel fatto che sembra che questa serie di metodi e classi, che riguardano la creazione del gioco, dei player, delle carte, poiché non presenti nel diagramma del modello, saranno delegate al controller.

3.B Forbidden Points Abbiamo trovato interessante la doppia gestione delle coordinate in cui è possibile, o meno, piazzare le carte. Come grupppo, avevamo optato per un set solo di coordinate, da aggiornare ogni volta che una carta viene piazzata, scegliendo se aggiungere le posizioni attorno ad essa ad un insieme di cordinate `Placeable`.

Ci rendiamo conto però che avere un set direttamente collegato alle posizioni *forbidden* dovute agli angoli disabilitati ci consente di diminuire il numero di controlli per aggiungere le celle adiacenti a quella in cui avviene il piazzamento a quelle disponibili.

Contando di piazzare una serie di carte che arriva da due angoli opposti, se una cella è forbidden per la carta che le sta in `TOP_LEFT` corner, piazzandone una che sta nel `BOTTOM_RIGHT` corner della cella proibita, saprò immediatamente se posso aggiungere la posizione della cella proibita a quelle in cui posso piazzare una carta. Questo controllando se è già presente nel set delle *forbidden*.

Questo ci risparmia il controllare se alle posizioni libere adiacenti alla carta appena piazzata sono adiacenti carte che hanno l'angolo collegato disabilitato. Si tratta di approcci in fin dei conti equivalenti ma dal punto di vista grafico, con in mente l'implementazione della *GUI*, ci permette di differenziare l'universo delle celle non raggiungibili da quelle vietate.

3.C (Nit) GeometricObjective Fare la review di `ComplexObjective.Layout` del modello del gruppo AM30 ci ha fatto realizzare la svista di implementare la struttura dati come una griglia 3x3 di `ResourceType`. Si tratta, ovviamente, come nel caso del campo di gioco, di una griglia isometrica e dunque di al massimo 6+1 `ResourceType`.

4 Appendice

Alleghiamo in appendice il diagramma presentato dal gruppo AM30.

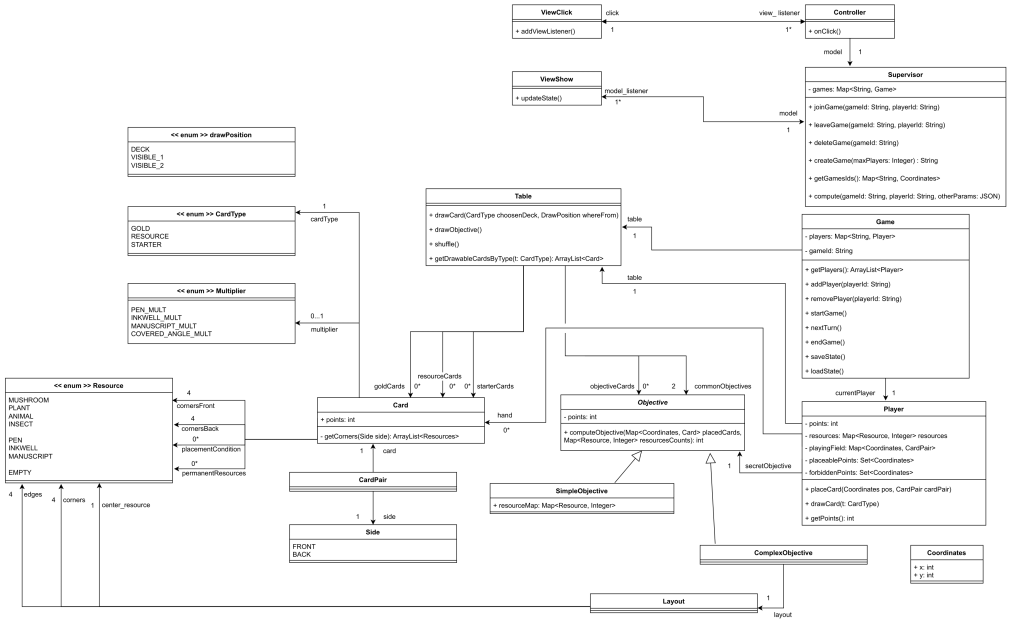

Figure 1: Diagramma UML presentato dal gruppo AM30: model