FORZA 4

Argento Simone
Ferri Francesco

Metodologie Di Programmazione 30/01/2022

Indice

1 Introduzione

Il seguente documento ha lo scopo di fornire tutte le informazioni utili alla comprensione del lavoro svolto per lo sviluppo del progetto "**Forza 4 con visualizzazione tramite GUI**". Il programma realizzato rappresenta una personale edizione in formato digitale del celebre gioco da tavolo creato dalla [1] *Milton Bradley* nel 1974.
Il gioco consiste in una sfida tra due giocatori i quali, per vincere, si pongono l'obiettivo di allineare 4 pedine del proprio colore in orizzontale, in verticale o in diagonale su di una griglia di 6 righe e 7 colonne. I giocatori, a turno, giocano un gettone del proprio colore in una colonna a loro scelta. Fisicamente, la griglia è disposta in verticale in modo tale che i gettoni vengano inseriti dall'alto in una fenditura e cadano sempre sulla riga libera più bassa. Così, si può scegliere in quale colonna giocare, ma non in quale riga. Di conseguenza non si può giocare in una colonna già piena, contenente 6 gettoni.
Il giocatore che per primo riesce ad allineare 4 pedine del proprio colore è il vincitore della partita. Nel caso in cui la griglia fosse totalmente riempita, senza che nessun giocatore riesca ad allineare 4 pedine del proprio colore, la partita è dichiarata patta.

I referenti del progetto, Argento Simone e Ferri Francesco, si sono divisi i compiti di sviluppo come segue:

- Argento Simone: gestione della partita, salvataggio della partita, recupero informazioni partita salvata, creazione e salvataggio dei giocatori, creazione matrice di gioco.
- Ferri Francesco: interfaccia grafica, gestione della User Experience, creazione matrice di gioco.

Per maggiori dettagli andare alla sezione Referenti di Sviluppo.

[1] La Milton Bradley Company è una casa editrice statunitense di giochi da tavolo e, in misura minore, di videogiochi.

2 Fasi di progetto

La realizzazione del progetto **"Forza 4 - GUI"** è frutto di un lavoro certosino caratterizzato da fasi di realizzazione precise e determinate; non si è pensato, quindi, di scrivere semplicemente del codice Java funzionante che permettesse di ricreare in digitale il famoso gioco da tavola, ma di sviluppare un programma manutenibile nel tempo e aggiornabile in maniera semplice, per l'eventuale futura aggiunta di ulteriori funzionalità.

Le fasi di lavoro possono essere così sintetizzate:

1. **Pianificazione**: sono state definite le prime idee implementative del progetto e le varie funzionalità da realizzare. All'inizio si era pensato di creare una sorta di "Roster dei Personaggi", consistente in una schermata di gioco dove l'utente avrebbe potuto scegliere il proprio personaggio personalizzandone il colore della pedina. Questa soluzione è stata successivamente scartata in quanto si è deciso utilizzare due colori fissi, assegnati in maniera casuale ai due giocatori di ogni partita, al fine di rispettare una determinata palette cromatica. Si è deciso di offrire ai giocatori la possibilità di modificare il proprio username e di visualizzare lo storico delle proprie statistiche in-game.

2. **Progettazione**: si è discusso e pianificato su carta il percorso implementativo da intraprendere. È stato dato un primo design alle varie schermate di gioco ponendo particolare attenzione sulle funzionalità da offrire in ogni finestra piuttosto che sulla rappresentazione grafica vera e propria. Parte fondamentale della fase di progettazione è stata quella di definire le linee guida delle varie classi da creare, rimarcando l'importanza di seguire per la scrittura del codice quelli che sono i principi **SOLID** dell'*OOP*. A tal proposito è stato creata una prima bozza di diagramma UML, successivamente migliorata e modificata. La sua versione finale comprende tutte le varie classi usate per il progetto e le relazioni che ci sono tra di esse. Per visionarla fare riferimento alla sezione Diagramma UML.
   Per una spiegazione dettagliata delle classi create e dei relativi metodi visitare la sezione descrizione delle classi. In questa fase sono stati definiti anche gli strumenti pratici di realizzazione. Nello sviluppo del codice ci si è avvalsi, in particolare, dell'IDE Eclipse e dello strumento di controllo versione distribuito *GitHub*[2].

3. **Realizzazione**: si è passati in maniera concreta allo sviluppo di codice Java e alla creazione degli elementi grafici per la GUI. Nella scrittura delle classi si è cercato di rispettare rigidamente quelle che sono le *Java Code Convention*[3]. Durante questa fase si è strutturato il programma in directory.

4. **Documentazione**: sono stati redatti tutti i documenti relativi al progetto. In questi modo risulta più facile comprendere quali sono state le decisioni prese alla base della sua realizzazione e quali sono i passaggi da seguire per la corretta esecuzione del software applicativo.

5. **Revisione**: sono state portate a termine tutte le attività di revisione finale e sono state fatte le ultime verifiche prima della consegna.

[2] GitHub
[3] Java Code Convention

3 Descrizione delle classi

In questa sezione si vogliono descrivere i ragionamenti tecnici e le scelte implementative che stanno dietro la creazione delle varie classi. L'obiettivo non è quello di creare un semplice elenco puntato che descriva l'utilizzo di ogni classe, bensì è quello di spiegare al meglio quello che è stato il processo di analisi tecnica e di progettazione del software. Inoltre, per ogni classe, verranno indicati quali sono i principi SOLID che rispecchia. Per evitare che questa porzione di documento risulti una mera ripetizione si fa presente che per avere una visione dettagliata su metodi e attributi di ogni classe è consigliabile consultare la Javadoc contenuta nella cartella *Forza4/doc/javadoc*.

Alla base dello sviluppo del programma ci sono pochi concetti chiave così riassumibili:

- lo scopo del gioco è quello di far interagire due giocatori, dotati ognuno delle proprie pedine colorate (un colore diverso per ogni giocatore), con una griglia di dimensioni 6x7 (rispettivamente altezza e larghezza) e disporre quattro pedine dello stesso colore verticalmente, orizzontalmente oppure diagonalmente. Risulta ovvio come la creazione della classi **Grid** e **Player** sia necessaria per espletare quelle che sono le funzionalità minime dell'applicativo;
- poiché il software deve dare la possibilità agli utenti di interrompere una partita, memorizzarla e recuperarla, si è pensato di creare una classe **Handler** in grado di svolgere queste attività.
- offrire un'interfaccia grafica *user friendly* che faccia capire all'utente, in pochi secondi, come muoversi all'interno dell'applicativo. Per questa ragione sono state scritte tutte le classi relative alle schermate.

Nell'elenco sottostante verranno brevemente descritte le classi principali che permettono il funzionamento logico del programma e la loro eventuale relativa attinenza a principi SOLID:

- **Player**: la classe rappresenta un giocatore. Si è pensato che ogni *Player* abbia come caratteristiche un username **univoco** e degli attributi che indicano il numero di partite vinte, perse e pareggiate. Ogni volta che un giocatore termina una partita le sue statistiche vengono aggiornate in maniera automatica. La classe rispetta il principio di *Singola Responsabilità* in quanto è stata progettata solamente per gestire un giocatore e le sue caratteristiche e impiega il polimorfismo per *overloading* sul metodo costruttore. Ciò è necessario in quanto, se un giocatore viene letto dal documento digitale sul quale è memorizzato, bisogna essere in grado di creare un oggetto *Player* con un determinato numero partite vinte, perse e pareggiate.
- **Grid**: rappresenta e gestisce la griglia di gioco. Si è pensato di dotarla di due attributi:
  1. una matrice di gioco (matrice di interi) che memorizza il numero 0 nelle celle ancora vuote, il numero 1 nelle celle riempite dalle pedine del primo giocatore e il numero 2 nelle celle riempite dalle pedine del secondo giocatore
  2. un vettore di 7 celle ognuna delle quali contiene un numero (da 0 a 6) che indica quante righe sono disponibile per ogni colonna.

  Ogni volta che l'utente sceglie una colonna, il programma in automatico posiziona la pedina sulla prima riga disponibile dal basso, purché la colonna selezionata non sia piena. Questa classe si occupa anche di controllare se un giocatore ha vinto oppure se la partita può continuare. Anche questa classe rispetta il principio di *Singola Responsabilità* poichè si occupa della semplice gestione della tabella. Applica il polimorfismo per *overloading* sul metodo costruttore per lo stesso motivo riconducibile a quello di *Player* (anche essa viene salvata sul documento digitale) e sul metodo *setAvailable*. Per *overriding* sul metodo *toString*. Ciò risulta particolarmente utile per la classe che si occupa del salvataggio su file JSON, in quanto offre una stringa già pronta per la memorizzazione. Per avere più informazioni si consiglia di nuovo la consultazione della Javadoc al fine di evitare ripetizioni inutili.
- **Handler**: classe astratta creata al fine di poter usare un Handler generico nelle varie schermate di gioco. Questa scelta ha permesso di rispettare i principi **L** e **D** (*Liskov Substitution*, *Dipendence Invertion*). Il principio di Sostituzione di Liskov è rispettato dato che una qual si voglia sottoclasse di *Handler* è in grado di interfacciarsi con qualsiasi oggetto che usi direttamente un *Handler* senza generare errori di alcun tipo. Ciò è garantito in quanto la classe astratta dichiara metodi astratti, e qualsiasi sottoclasse che la voglia estendere deve obbligatoriamente implementare i metodi astratti. Il principio di Inversione delle Dipendenze

è rispettato poiché le classi che estendono *Handler* non dipendono da un'implementazione concreta bensì da un'astrazione. La classe adotta anche polimorfismo per *overloading* su metodo *save*: uno è in grado di memorizzare una *Match*, l'altro serve per salvare un *Player*.

- **JSONHandler**: rappresenta una concreta implementazione di un gestore per i salvataggi. Estende la classe astratta *Handler* ed ha il compito di memorizzare *Player* e *Match* all'interno di un file JSON. Come attributi ha due costanti di tipo *String* che indicano il percorso e il nome dei file sui quali salvare la partite e i giocatori. La classe rispetta il principio si *Singola Responsabilità* in quanto si occupa della sola gestione dei file JSON, per eventuali ulteriori metodi di memorizzazione è necessario implementare altri Handler. Come per *Handler*, la classe adotta il polimorfismo per *overloading* sul metodo *save*.

- **Match**: la classe è stata creata in quanto si è pensato di gestire ogni singolo match come un'unità logica a sé. Si è ragionato sul fatto che ogni singolo "scontro di gioco" sia caratterizzato da due giocatori, una griglia e un contatore di turno. Questa classe fa quindi uso diretto delle classi *Grid* e *Player* mettendolo in associazione tra di loro. All'interno del codice si può notare la presenza di un altro attributo di tipo booleano, esso serve per rilevare se la partita è terminata.

Non vengono inserite in questo elenco le classi relative alle schermate. Questa scelta è dettata dal fatto che sarebbero molto simili tra loro in quanto la loro funzionalità è praticamente la stessa, ciò che le differenzia è la tipologia di schermata che devono mostrare a video.
Si vuole fare menzione della classe **Index** in quanto non è una schermata ma una classe che gestisce il passaggio fra le schermate tramite un CardLayout[4]. Nonché la classe contenente il Main.

3.1 Diagramma UML

A causa della grandezza notevole del diagramma UML per non rendere impossibile la leggibilità chiediamo di aprire il file *Forza4/doc/Class_ Diagram.png*.

[4] CardLayout

4 Descrizione delle funzionalità

L'applicativo ha le seguenti funzionalità:

- Gestione degli utenti
  - Creazione di un nuovo utente.
  - Eliminazione di un utente.
  - Modifica dell'username di un utente.
- Esperienza di gioco
  - Giocare una partita.
  - Creazione di una partita fra due giocatori precedentemente creati.
  - Riprendere una partita salvata.
- Visualizzazione statistiche dei giocatori.

4.1 Scelte di progetto

- E' stato scelto di memorizzare una sola partita alla volta fra due utenti perchè in fase di analisi è stato ritenuto insensato lasciare in sospeso più di una partita fra gli stessi giocatori. Le partite sono riconoscibili grazie alla denominazione:
  'USERNAME PLAYER 1' VS 'USERNAME PLAYER 2'.
- E' stato scelto di memorizzare utenti e partite tramite file JSON in maniera tale da comprendere a pieno e rafforzare le proprie competenze sull'argomento essendo stato un tema del il corso. Inoltre, in fase di analisi, è stato selezionato per la sua semplicità e la sua praticità. Si tiene a ribadire che comunque l'utilizzo della classe astratta *Handler* non preclude l'utilizzo di altre tipologie di memorizzazione.
- In fase di analisi è stato trovato utile creare la sezione **Statistiche** per mantenere uno storico del numero di partite vinte, perse e pareggiate da ogni giocatore.
- In fase di analisi si è pensato al concedere la possibilità all'utente di modificare il proprio username. Questo ha implicato un sistema di modifica automatizzato per tutte le partite salvate in cui è partecipe. In questo documento non si è voluto aggiungere del codice in quanto si ritiene che le classi scritte siano abbastanza auto esplicative poiché ricche di commenti. Un riferimento va fatto, però, per il metodo **updateMatchOBJ**.

```java
/**
     * Metodo per modificare il nome di una partita sul file JSON relativa ad un vecchio Player di cui è stato
modificato il nome
     * @param oldPlayer vecchio giocatore del quale rimuovere le partite a lui associate
     * @param newPlayer nuovo giocatore da associare alle partite relative del vecchio giocatore
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    private void updateMatchOBJ(Player oldPlayer, Player newPlayer) {
        JSONArray matches = new JSONArray();


        Map<String, Object> partite = getMatchesOBJ();

        // Se la partita ha il giocatore è da modificare
        for (Map.Entry element : partite.entrySet()) {      // MAP.Entry : è un'interfaccia per accedere a tutti gli
elementi di una Map
            if (! element.getKey().toString().contains(oldPlayer.getUsername())) {
                matches.add((JSONObject) element.getValue());
            }
            else {
                JSONObject obj = (JSONObject) element.getValue();
                if (obj.get("player1").toString().compareTo(oldPlayer.getUsername()) == 0) {

                    Match rechargeMatch = parseMatch(obj, newPlayer, 1);
                    this.removeMatchFromPlayers(oldPlayer, new Player((String) obj.get("player2")));
                    this.save(rechargeMatch);

                }


                else {

                    Match rechargeMatch = parseMatch(obj, newPlayer, 2);
                    this.removeMatchFromPlayers(oldPlayer, new Player((String) obj.get("player1")));
                    this.save(rechargeMatch);

                }

            }
        }
    }
```

Figura 1: Codice sorgente della funzione updateMatchOBJ

Il codice sottostante mostra come non è stato possibile gestire le partite salvate sul file JSON semplicemente come *Match* durante la fase di update. La scrittura del metodo si è resa necessaria in quanto, una volta modificato il nome dell'utente, non era più possibile creare delle partite, a partire dal file JSON, che contenessero l'utente. Per questo motivo le semplici stringhe contenute sul JSON hanno avuto il bisogno di essere gestiste come oggetti di tipo JSON.

In fase di realizzazione ci si è più volte soffermati a pensare se la creazione di una classe astratta *Handler* fosse la scelta più appropriata. Alla fine, dopo numerose ricerche, si è pensato che il suo utilizzo risultasse migliore rispetto a quello di un'interfaccia. In accordo con quanto riportato sulla documentazione Java e sul sito Tutorials Point, l'utilizzo della classe astratta è preferibile in quanto *"si vuole condividere codice tra classi strettamente correlate e ci si aspetta che esse abbiano molti metodi in comune"*. Non si deve garantire una comune funzionalità tra classi slegate tra loro bensì tra classi con lo stesso scopo ma in ambienti di impiego diversi. Poiché solo gli *Hanlder* devono essere in grado di gestire partite e giocatore, si è arrivati alla conclusione che la classe astratta fosse la migliore soluzione.

5 Referenti di sviluppo

Fare riferimento a Ferri Francesco per:

- CreateUser
- EditUser
- Game
- Grid
- Index
- InsertButton
- LoadMatch
- Menu
- PlayerStatsViewer
- RemoveUser
- Stats
- Tied
- UserEdited
- UsersModifier
- UsersPool
- Winned

Fare riferimento ad Argento Simone per:

- Grid
- Handler
- JSONHandler
- Match
- Player