1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

Relazione progetto

Programmazione e modellazione a oggetti
Sessione estiva anno 2023/2024
Docente: Sara Montagna

Autori

Maccaroni Tommaso: matricola 321621
Labbruzzo Angelo: matricola 319928

INDICE

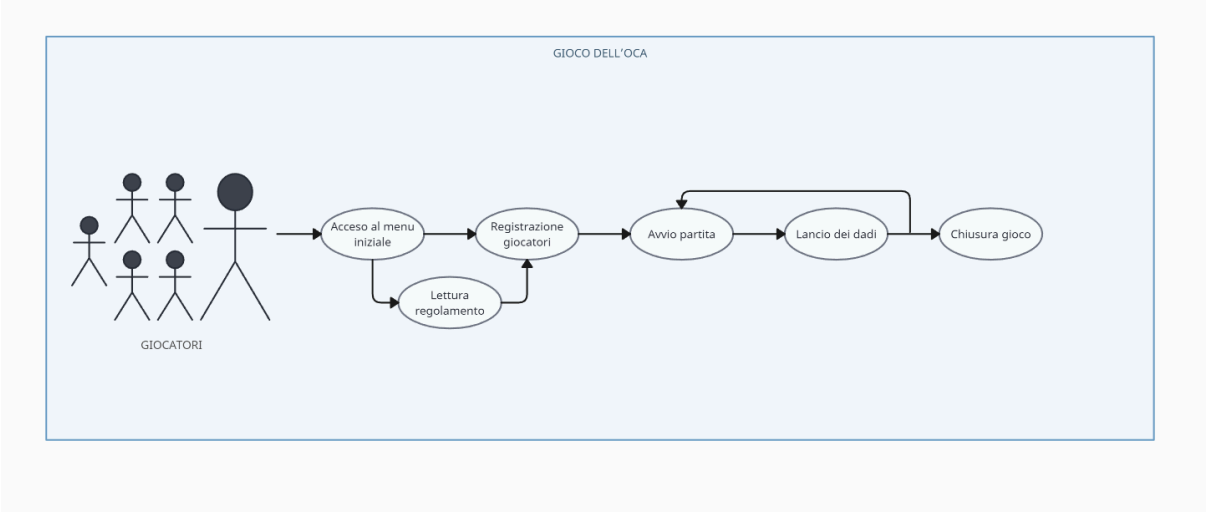
Capitolo 1

Analisi

L’applicazione ha come scopo quello di replicare il classico gioco dell’oca, in modo da crearne una versione portatile da giocare comodamente dal proprio dispositivo.
Più giocatori si sfideranno in un campo da gioco composto da 64 caselle, di cui 41 normali e 23 azione. Ogni giocatore sarà in grado di spostarsi tra esse tramite il lancio di due dadi, procedendo tante volte quanto la loro somma. Vince il primo giocatore ad arrivare precisamente all’ultima casella. Una volta terminata la partita si potrà decidere di rigiocarne una nuova, con la possibilità di cambiare giocatori, o di terminare l’applicazione.

1.1 Requisiti

1) Capacità di registrazione dei giocatori
    a) Scelta del nome che verrà utilizzato in partita
    b) Scelta del colore che avrà la pedina
2) Pagina dedicata al regolamento
3) Capacità di simulare una partita completa del gioco dell’oca
    a) Lancio dei dadi
    b) Spostamento tra caselle
    c) Esecuzione delle caselle azione

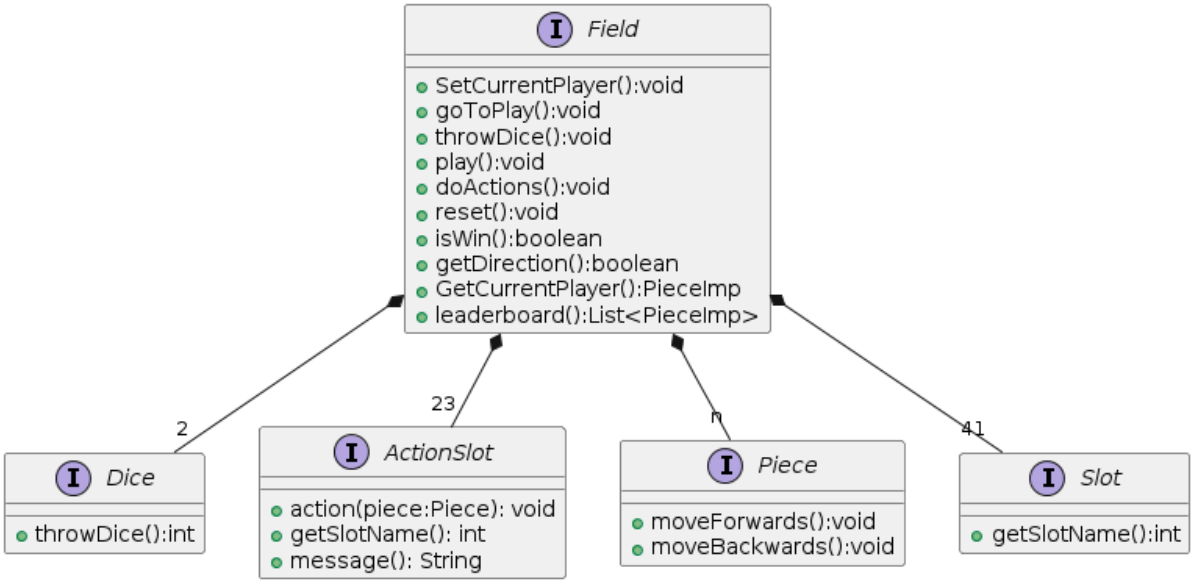


1.2 Modello del dominio

Questo progetto dovrà permettere la regolare esecuzione di una o più partite del gioco dell'oca.
Potranno partecipare da 2 fino a 6 giocatori, ognuno dei quali libero di scegliere il proprio nome utente e il colore della propria pedina tra quelli disponibili nel menù di registrazione.
Dopo che ogni giocatore avrà chiaro le regole del gioco, si potrà dare il via alla partita.
Ogni giocatore, all'inizio del proprio turno, lancerà i due dadi(Dice), muovendosi poi in base alla somma del lancio.
I giocatori tireranno i dadi nell'ordine con il quale si sono registrati.
Il campo da gioco (Field), proprio come nella versione originale, si compone di caselle normali (Slot) e caselle azione di diversi tipi (ActionSlot):

- STOP (6) → Il giocatore rimarrà fermo tanti turni quanto segnalato dalla casella
- X2 (2) → Il giocatore avanzerà del risultato del lancio moltiplicato per due
- RITIRA (5) → Il giocatore potrà ritirare i dadi una volta finito lo spostamento
- VAI A (7) → Questo tipo di caselle possono rappresentare sia un malus che un bonus, obbligando il giocatore a muoversi alla posizione designata dalla casella
- SWAP (3) → Questo tipo di casella effettua uno swap tra la pedina che vi esce sopra e una pedina casuale

La partita verrà completata quando uno dei giocatori finirà precisamente sulla casella finale, in caso contrario il giocatore si muoverà all'indietro tante volte quanti i movimenti mancanti.
Il tutto verrà gestito tramite un'interfaccia grafica (GUI).



Capitolo 2

Design

La classe principale è “FieldImpl”, che si occupa di gestire tutto ciò che avviene nel campo da gioco durante la partita. Tramite i diversi metodi, la classe permette di avviare la partita, assegnando una pedina per ogni utente registrato precedentemente, permette inoltre il lancio dei dadi e gestisce il movimento dei pezzi.
Le caselle del campo da gioco vengono tutte create partendo dall’interfaccia Slot, che viene implementata da SlotImp (casella senza effetti), per quanto riguarda invece le caselle azione, si estende la classe SlotImpl e si implementa un ulteriore interfaccia, ovvero ActionSlot, in modo tale da avere una maggiore facilità nella gestione di queste caselle.
Per la creazione degli ActionSlot abbiamo utilizzato il pattern “Factory” tramite la classe FactorySlots.
In ogni partita vengono utilizzati due dadi, creati a partire dall’interfaccia Dice, implementata da DiceImpl, che generano due numeri casuali (da 1 a 6), la cui somma indicherà il numero di caselle di cui avanzare.
La registrazione degli utenti avviene nel menù iniziale, dove ogni giocatore è in grado di inserire un proprio nickname e selezionare un colore tra quelli presenti.
L’utente è strettamente collegato alla pedina, rappresentata dalla classe PieceImp che estende la classe astratta User, rappresentate la persona fisica che decide di partecipare al gioco, e implementa l’interfaccia Piece, rappresentate la pedina sul campo da gioco.

2.1 Architettura

Per la realizzazione di questo applicativo abbiamo deciso di utilizzare un pattern architetturale chiamato MVC (Model View Controller)
I 3 elementi del pattern sono definiti dai 3 package utilizzati:

- **Model** → Si occupa della gestione totale dei dati presenti all'interno dell'applicazione, definendo il comportamento e lo stato di essa.
  Quando il Model subisce delle variazioni viene interrogato dal Controller in modo tale da riferirgli i cambiamenti avvenuti così che lui possa aggiornare la View di conseguenza. All'interno del nostro progetto, la gestione di ciò è affidata all'entità "Field" , che si occupa di gestire utenti, lanci e gli spostamenti delle pedine, tale entità viene implementata poi dalla classe "FieldImpl".

- **View** → Ha lo scopo di gestire la parte grafica e tutto ciò che riguarda l'interazione con l'utente, informando poi il controller di eventuali avvenimenti in modo che possa operare coerentemente con le richieste dell'utente.
  Il framework che abbiamo deciso di utilizzare è Java Swing.
  Nel nostro caso la *View* è isolata dal *Controller*, siccome eseguiamo soltanto delle interrogazioni che non ne modificano lo stato.
  Essendo il nostro un programma di piccole dimensioni, abbiamo optato per utilizzare un'associazione 1:1 con soltanto una *View* e un *Controller*.

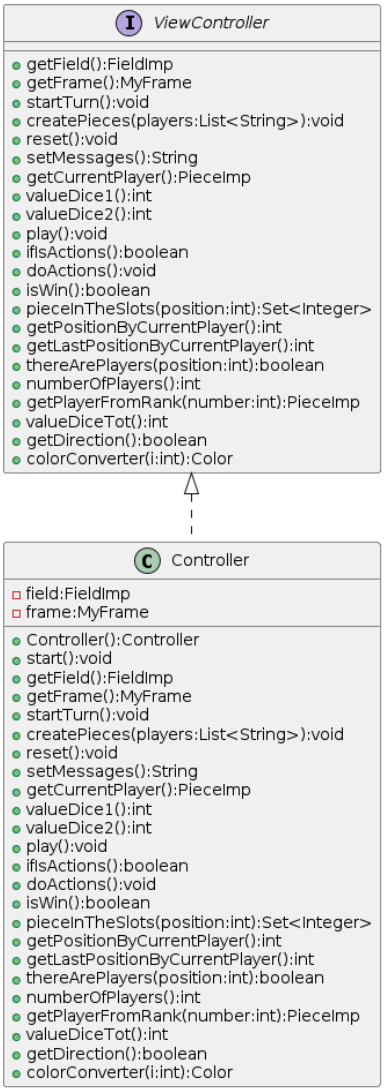
- **Controller** → Il *Controller* ha il compito di mettere in relazione e permettere la comunicazione tra *Model* e *View*, prendendo le informazioni ricevute dalla *view* in seguito all'interazione con l'utente e avvisando il Model, dopo che esso avrà effettuato le sue modifiche il *Controller* notificherà la view i cambiamenti da eseguire per rimanere coerente con *Model*.

2.2 Design dettagliato

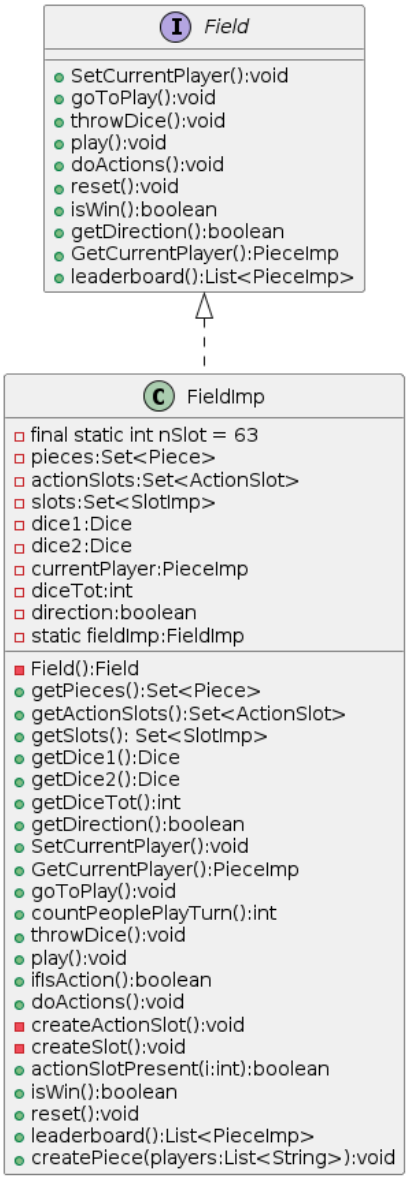
Maccaroni Tommaso
Classi e interfacce realizzate:

- Controller.java e ViewController.java
- Field.java e FieldImp.java
- Board.java
- Game.java

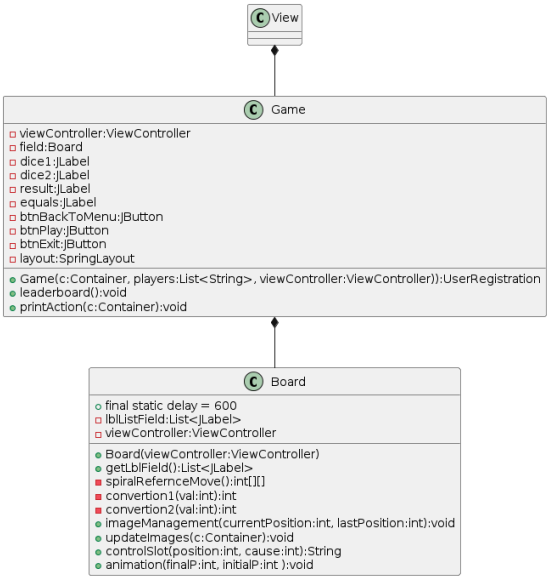
1. La classe *Controller* viene creata a partire dall'implementazione dell'interfaccia *ViewController* e si occupa di istanziare Model e View.
   Come ogni controller ha il compito di mettere in relazione queste due entità, permettendo alla View di interrogare il Model, in modo da captare i cambiamenti che avvengono al suo interno e agire di conseguenza.



2. Il fulcro del programma è la classe *FieldImp*, che si occupa di tutte le funzionalità principali per il corretto funzionamento del gioco.
   Il primo compito è quello di creare tutte le entità necessarie, come le pedine, i dadi e le caselle, sia normali che quelle azione.
   A livello interno ha il compito di gestire i turni e i cambiamenti che avvengono, fornendo i permessi necessari alle pedine che devono eseguire il turno, dando la possibilità di lanciare i dadi e fornendo indicazioni sul movimento da eseguire, inoltre si occuperà della messa in esecuzione dei vari effetti ogni qual volta una pedina si fermerà su una casella azione, oltre a questo crea la classifica e determina la fine di una partita.
   Una funzione secondaria di questa classe è quella di resettare le pedine, utile nel caso i giocatori decidano di interrompere la partita e ricominciarla.
   Essendo il campo da gioco unico durante tutta l'esecuzione dell'applicazione, ho optato di utilizzare il pattern Singleton, che rende il costruttore privato e restituibile tramite un metodo, in modo da evitare che l'utente ne crei altri.



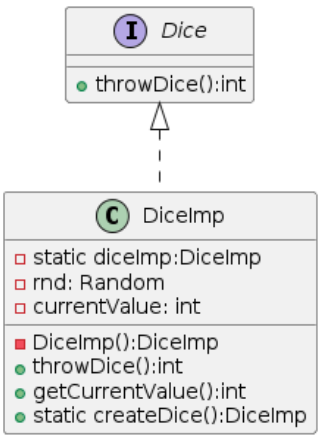
3. La classe *Board* rappresenta il campo da gioco e tutte le caselle che sono presenti all'interno di esso.
   Ho deciso di optare per una disposizione a spirale, resa possibile grazie all'ideazione di un metodo che offre dei riferimenti ad ogni casella, visto che ognuna di esse presenta un numero diverso sul campo da gioco rispetto al valore che ha all'interno della *List*.
   Al suo interno troviamo diversi metodi, il principale è quello che permette la visione del movimento sul campo delle pedine, che avviene tramite un cambio di immagini, con un delay imposto da un altro metodo per evitare uno spostamento "a teletrasporto". Un altro metodo si occupa invece di capire e gestire le diverse combinazioni di pedine che si possono verificare su una casella (da nessuna a tutte insieme) per modificare l'immagine coerentemente.
   Essendo il campo da gioco una scacchiera, ho deciso di utilizzare un layout di tipo *GridLayout* di dimensione 8X8.
4. La classe *Game* è la più significativa, dove avviene il vero e proprio gioco.
   Al suo interno troviamo infatti un'istanza della classe *Board*, che si trova sulla parte sinistra dello schermo.
   Sulla destra invece troviamo gli elementi appartenenti a questa classe, come ad esempio la classifica, che si terrà aggiornata di turno in turno per tutta la durata della partita. Nella parte inferiore troviamo invece tre bottoni, uno per annullare la partita e tornare al menu, uno per chiudere l'applicazione e il più importante, il pulsante "Gioca" che oltre a permettere il lancio dei dadi, con il risultato visibile a schermo, si occupa di interrogare il modello riguardo a ciò che avviene in modo da aggiornare la grafica di conseguenza, come ad esempio il pop-up che appare con la vittoria di un giocatore o il disabilitarsi a fine partita.
   Alla sua destra troviamo uno spazio nel quale verranno segnalati gli spostamenti effettuati durante il turno della pedina attuale.
   La classe è inoltre dotata di un metodo che, ogni volta attivata una casella azione, farà apparire un pop-up descrivente l'effetto da applicare al giocatore.
   Il layout utilizzato è uno *SpringLayout*, in modo da avere libera gestione del posizionamento degli elementi all'interno della schermata.



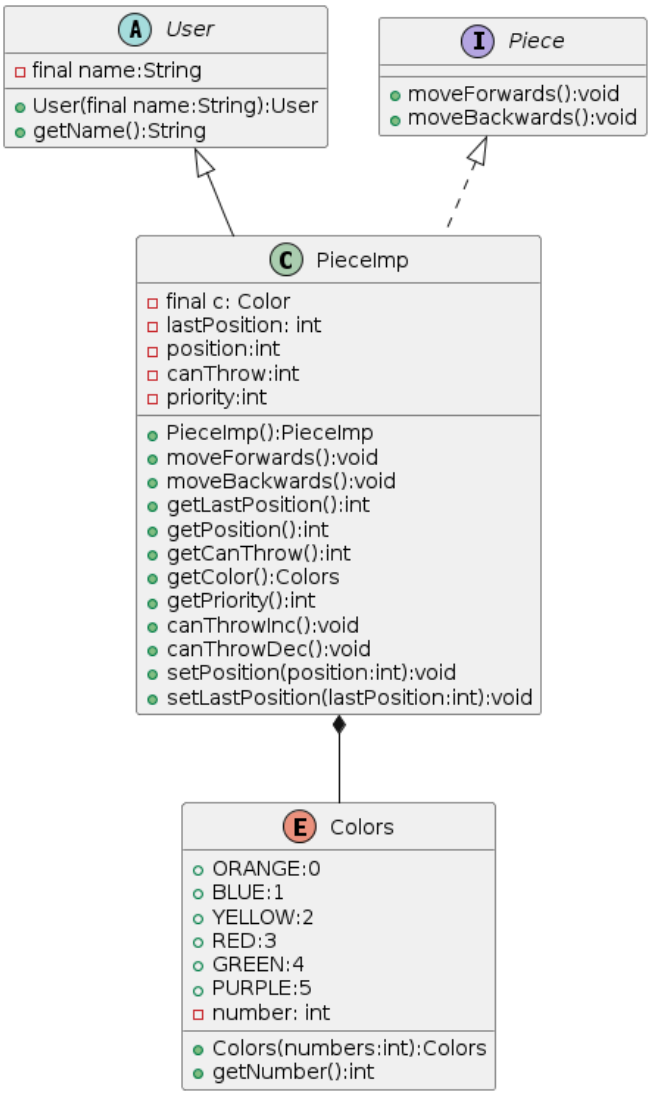
Labbruzzo Angelo
Classi e interfacce realizzate:

- Package model.dice
  - Dice
  - DiceImpl
- Package model.pieces
  - Piece
  - PieceImp
  - User
  - Colors
- Package model.slots
  - SlotImp
  - Slot
  - ActionSlot
  - FactorySlots
  - DoubleResultActionSlot
  - GoToActionSlot
  - RerollActionSlot
  - StopActionSlot
  - SwapActionSlot
- Package view
  - MyFrame
  - Menu
  - UserRegistration
  - Regulation

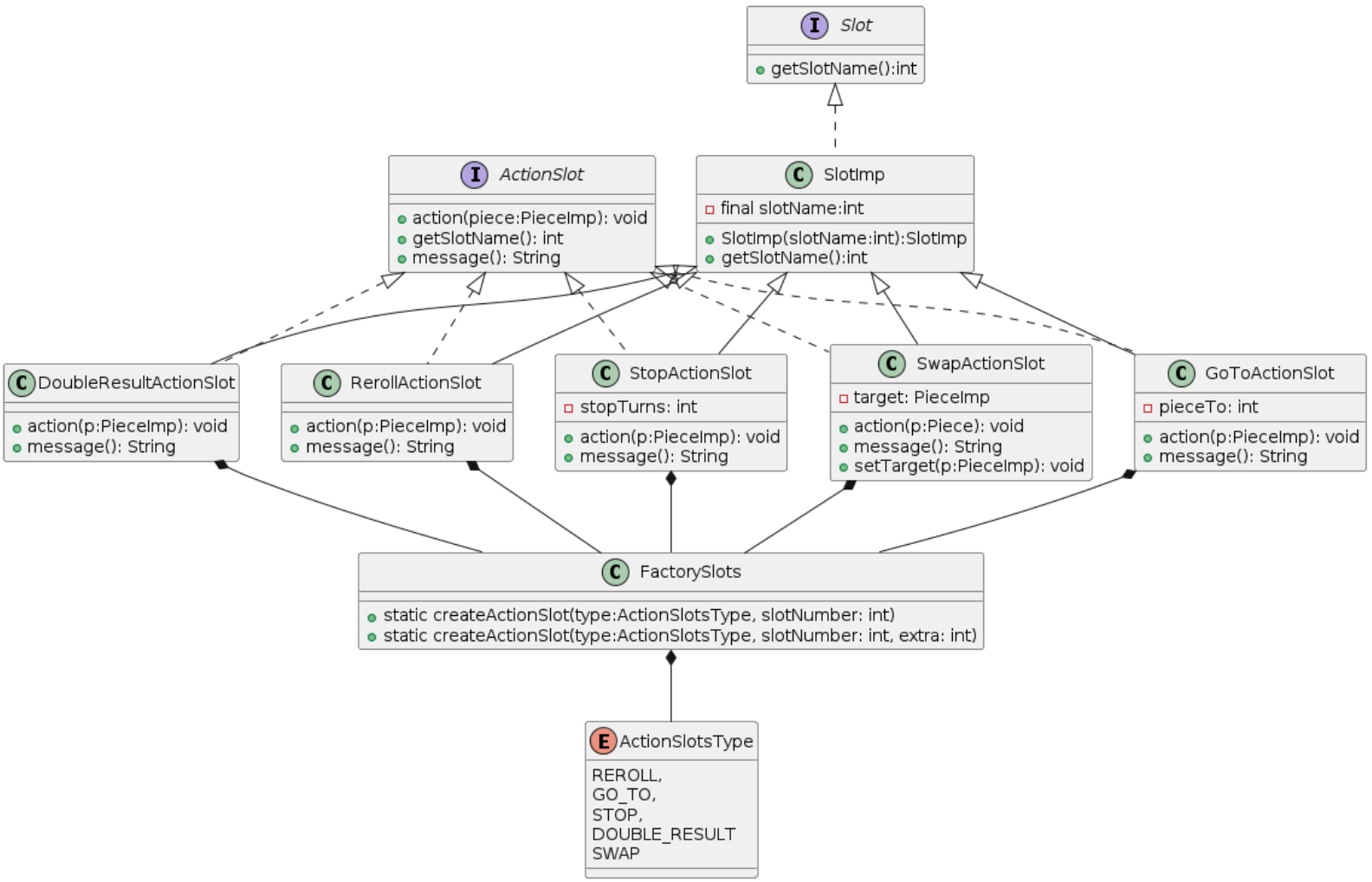
1. All'interno del package *model.dice* troviamo soltanto l'interfaccia *Dice* e la classe implementante *DiceImp*.
   Il dado è un oggetto molto semplice all'interno del progetto, è dotato di una funzione che genera un numero casuale da 1 a 6, effettuato tramite un oggetto di tipo *Random*.
   Per ogni partita vengono istanziati due dadi che devono essere unici, pertanto ho deciso di utilizzare il pattern Singleton, rendendo privato il costruttore di *DiceImp*, e restituendo la sua istanza tramite il metodo *createDice*.



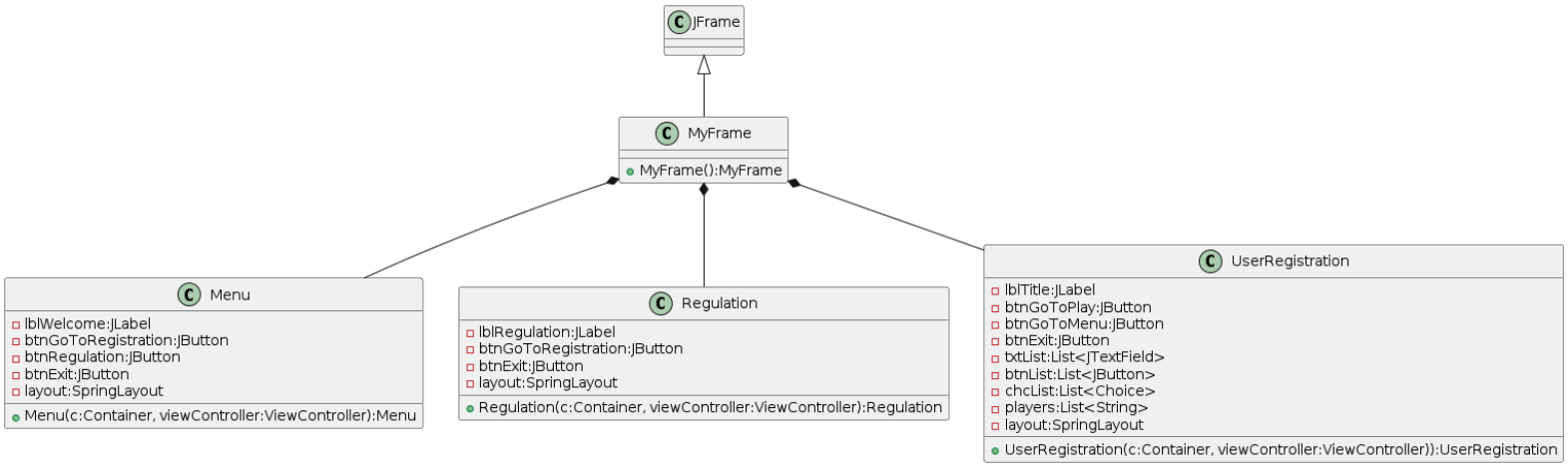
2. Il package *model.pieces* ha lo scopo di gestire tutto quello che riguarda la creazione e gestione del “pezzo” e la sua diretta relazione con lo “user”.
   L’interfaccia *Piece* presenta le caratteristiche che avrà la pedina in gioco, e viene implementata dalla classe *PieceImp.*
   All’interno di questa classe troviamo vari metodi utilizzati per il funzionamento delle pedine, come i permessi che vengono tolti e ricevuti quando necessario, ad esempio a inizio turno o quando si capita su uno stop, e due metodi, moveForwards() e moveBackwards(), che permettono rispettivamente il movimento in avanti e indietro in base a cosa è richiesto al momento.
   La classe *User* invece è una classe astratta, ovvero non istanziabile, che contiene soltanto l’attributo *“name”* e un metodo per la sua restituzione.
   Colors è invece un’enumerazione, contenente 6 diversi colori che andranno applicati sulle pedine al momento della registrazione degli utenti, ognuno dei quali sarà in grado di selezionare il preferito tra blu, rosso, verde, giallo, arancione e viola.



3. Il package *model.slots* ha lo scopo di gestire tutto quello che riguarda la creazione delle caselle che compongono il campo da gioco.
   Come spiegato prima esistono due diversi tipi di caselle, ovvero “normali” e “azione”. Le caselle normali sono gestite dalla classe *SlotImp*, che implementa l'interfaccia *Slot* e contiene al suo interno soltanto il numero dello slot, mentre le classi riguardanti le caselle azione implementano tutte l’interfaccia *ActionSlot*.
   Per la loro creazione ho utilizzato il pattern Factory tramite una classe chiamata FactorySlots, nella quale è presente un metodo statico che serve a istanziare le caselle, questo metodo viene poi richiamato nella Field.
   I diversi tipi di slot azione sono rappresentati dalle seguenti classi:
   - DoubleResultActionSlot → Raddoppia il risultato ottenuto dal precedente lancio dei dadi
   - GoToActionSlot → Sposta la pedina nella casella rappresentata dall’immagine
   - RerollActionSlot → Permette al giocatore di ritirare i dadi
   - StopActionSlot → Ferma il giocatore tante volte quanto imposto dalla casella
   - SwapActionSlot → Effettua uno scambio tra il giocatore e un altro casuale



4. Il package *view* contiene tutto ciò che riguarda la gestione dell'interfaccia grafica, che permette l'interazione con l'utente e la regolare esecuzione della partita.
   Nella classe *MyFrame* vengono date delle impostazioni base per la schermata, come la grandezza.
   La classe *Menu* rappresenta invece la prima schermata alla quale i giocatori si trovano davanti. Qui si possono visualizzare diversi pulsanti, uno per andare nella schermata di registrazione utenti, uno per leggere il regolamento e uno per chiudere l'applicazione, oltre a un messaggio di benvenuto.
   Per il posizionamento dei pulsanti e della scritta ho optato per un layout di tipo *SpringLayout*, in modo da avere completo controllo sulla loro posizione.
   Nella classe *Regulation* troviamo invece il regolamento del gioco, oltre ai bottoni per andare alla registrazione utenti o uscire dall'applicazione. Per la descrizione del regolamento ho deciso di utilizzare direttamente un'immagine, in modo tale da avere una resa grafica migliore senza dover attuare formattazioni che sarebbero risultate complicate e confuse. Come nella classe precedente ho utilizzato un layout di tipo SpringLayout.
   Nella classe *UserRegistration* abbiamo invece 6 campi di testo, nei quali ogni giocatore può inserire il proprio nome utente, e affianco ad ognuno di essi troviamo un menù a tendina dove tramite delle *Choice* si può scegliere il proprio colore, una volta fatto il pulsante "+" permette di confermare i dati inseriti, in seguito a un rapido controllo dei nomi e dei colori per verificare che non ci siano ripetizioni.
   Dopo che tutti i giocatori sono pronti si può scegliere di dare il via alla partita premendo sul pulsante "Inizia a giocare", eventualmente si può anche scegliere di tornare al menu iniziale o di chiudere l'applicazione.
   Il layout utilizzato è ancora una volta uno *SpringLayout*.



Capitolo 3

Sviluppo

3.1 Metodologia di lavoro

Dopo esserci concordati sulla specifica da sviluppare, abbiamo cercato di analizzarla al meglio per deciderne la struttura. Abbiamo individuato le entità principali che avrebbero caratterizzato il nostro progetto e le eventuali interfacce che avremmo potuto utilizzare.
Per la realizzazione della struttura abbiamo ideato un UML che abbiamo utilizzato come guida per capire cosa saremmo andati a sviluppare, l'UML è stato poi aggiornato nel corso dello sviluppo con l'aggiunta di metodi e classi.
Il passo successivo è stato quello di iniziare a scrivere il codice, assegnando precedentemente a ognuno di noi determinate classi su cui lavorare.
Seppur inizialmente abbiamo lavorato in maniera indipendente, cercando di focalizzarci prima sulle classi si reggevano in piedi da sole, abbiamo avuto frequenti scambi, sia per avere nuove idee che per tenere traccia del progresso fatto da ognuno.
Per quanto riguarda le classi "core", comprendenti funzionalità necessarie al corretto svolgimento dell'applicativo, abbiamo lavorato insieme, andando a implementare le singole classi sviluppate in precedenza.
Prima di passare all'implementazione grafica, abbiamo sviluppato una base in grado di funzionare regolarmente, permettendo quindi una partita completa anche senza di essa, il tutto accompagnato da opportuni test.
Dopo esserci assicurati che il programma funzionasse correttamente e implementasse tutte le funzionalità necessarie, abbiamo iniziato a pensare alle schermate di cui l'utente avrebbe fatto uso, passando quindi all'implementazione grafica, durante la quale ognuno di noi ha scritto e creato diversi pannelli indipendenti.
Una volta finite le schermate, abbiamo lavorato insieme per unirle tra loro e avere un'implementazione completa del gioco finale.
Una volta terminata la prima versione, abbiamo cercato di analizzarla al meglio per capire alcuni errori e come migliorarla, effettuando diversi cambiamenti, più o meno importanti, sia a livello di modello che della view.

3.2 Testing

Per il testing automatizzato abbiamo utilizzato il framework di unit testing JUnit 5, andando ad effettuare diversi test man mano che procedevamo con lo sviluppo (Test Driven Development).
Questa metodologia ci ha permesso di verificare il corretto funzionamento, indipendente, delle classi sviluppate, andando ad eseguire un nuovo test per ogni aggiunta, permettendo quindi di capire subito il segmento di codice che avrebbe eventualmente causato l'errore, ovvero l'ultimo inserito.
I test ci sono stati utili sia per quanto riguarda il corretto funzionamento dei singoli metodi presenti all'interno del modello, che per i singoli pannelli della view, testati prima singolarmente, per verificare che fossero coerenti con ciò che avevamo pensato in fase di progettazione, e poi tutti insieme una volta terminato il tutto.
Tutti i test si trovano all'interno del package *test*, e sono i seguenti:

- ModelTest.java
- ViewTest.java

3.3 Note di sviluppo

In questa sezione verranno mostrate le funzionalità avanzate di Java utilizzate per la realizzazione delle classi implementate.

Maccaroni Tommaso

- Collection → Utilizzate per avere una collezione di pedine, utenti e caselle.
- Stream → Utilizzati per la ricerca all'interno delle collection. Ve ne sono molti esempi nella classe *Controller.java*.
- Optional → Implementato per evitare l'uso della keyboard "null" riguardante il *Field*.
- Lambda Expressions → Utilizzate per definire filtri di ricerca degli stream.
- Factory → Utilizzata nel field per richiamare la creazione delle caselle azione

Per quanto riguarda la creazione del campo da gioco, ovvero del posizionamento iniziale delle caselle all'interno del *Field*, ho utilizzato un algoritmo chiamato *spiralReferenceMove()* che restituisce una matrice corrispondente alle caselle e la loro effettiva posizione.
Attraverso l'annidamento di 4 cicli *for* (uno per ogni direzione) all'interno di un *while* permette la conversione da numero della casella alla posizione effettiva nell'array, passando per una matrice, in modo da rendere coerente l'andamento delle pedine sul campo da gioco.
Ad esempio la casella 54 corrisponde alla posizione [6][6] nella matrice, l'algoritmo permette di riportare la posizione della matrice all'interno di un array, utilizzato dalle pedine per muoversi.

Labbruzzo Angelo

- ArrayList → Utilizzati nella view per tenere traccia degli utenti creati.
- Stream → Utilizzati per ricercare all'interno delle liste.
- Lambda Expressions → Vengono utilizzate per definire filtri di ricerca degli stream, vi è un esempio nella classe *UserRegistration.java*.
- Factory → Pattern utilizzato per la creazione delle caselle azione

Non è presente nessun algoritmo degno di nota in quanto le classi progettate da me utilizzano principalmente metodi semplici o già presenti nelle classi base di Java.