PolyBoard

Progetto di Programmazione ad Oggetti

Michele Veronesi - 1187383

A.A. 2019/2020

Università degli Studi di Padova

Indice

1 Introduzione

In questo progetto abbiamo voluto implementare il classico gioco degli scacchi in C++ con una gerarchia polimorfa per le pedine (la gerarchia principale). Tuttavia, scrivendo il codice, abbiamo mantenuto uno stile che permettesse di aggiungere all'applicazione un qualsiasi gioco che preveda delle pedine su una scacchiera di dimensione $N \times M$ con $N, M \in \mathbb{N}$ numeri finiti ≥ 2 sufficientemente piccoli da essere visualizzati in uno schermo (ad esempio la dama).
In particolare, è stato seguito lo schema Model - View - Controller, i quali verranno descritti approfonditamente nel seguito.
Oltre alla gerarchia principale delle pedine, è stata utilizzata un'altra gerarchia polimorfa per i giochi.
La struttura dati utilizzata è un array visto come matrice $N \times M$, contenente puntatori polimorfi a pedine.
La grafica e l'accesso ai file sono stati realizzati tramite la libreria Qt.
Sono state implementate tutte le regole degli scacchi per una partita PvP, ad eccezione delle regole di parità, le quali richiedevano uno sforzo di scrittura di algoritmi ottimizzati che avrebbe fatto sforare il tetto massimo di ore a disposizione per il progetto. È stata inserita un'action nella GUI per dichiarare una situazione di pareggio.

1.1 Metodo di sviluppo e suddivisione del lavoro progettuale

A causa dell'emergenza sanitaria del 2020 il progetto è stato realizzato completamente a distanza con *sessioni di lavoro esclusivamente sincrone*: si è usato Zoom/Skype per comunicare durante la stesura del codice e nella discussione dell'architettura.
Come IDE abbiamo adottato Qt Creator affiancato a Visual Studio Code con il plugin live share, il quale permette di lavorare sullo stesso workspace da pc diversi contemporaneamente.
Abbiamo sviluppato l'applicazione in ambiente Ubuntu 18.04.
Come strumento di versioning ci siamo affidati a GitHub.
Sono state preferite sessioni collettive e non individuali in modo da convergere verso soluzioni condivise e velocizzare la stesura del codice, dunque non c'è stata alcuna suddivisione del carico di lavoro. Questo ha permesso di realizzare il progetto in circa 50 ore di sessioni con tutti e tre i membri del gruppo, per un totale di 150 ore (considerando le ore individuali).
Gli studenti coinvolti nello sviluppo di PolyBoard, oltre al sottoscritto, sono:

- Alberto Guarnieri - MATR. 1187119
- Diego Piola - MATR. 1193414

1.2 Dettaglio impiego ore sviluppo

È stato tenuto un log preciso degli incontri effettuati, da questo ne ricavo che le ore di sviluppo sono state impiegate nel seguente modo:

- 4 ore per la partecipazione ai tutorati (io ho partecipato a quelli live, gli altri componenti hanno visto i video di Qt realizzati dal tutor)
- 8 ore per lo sviluppo del container, il quale ha portato via parecchio tempo per la difficoltà di templatizzazione, che alla fine si è preferito non effettuare (verrà spiegato nell'apposita sezione il motivo)
- 8 ore per la gerarchia polimorfa del modello
- 8 ore per la creazione della gerarchia di pedine (e successive modifiche)
- 12 ore per la realizzazione di grafica e segnali
- 4 ore per la definizione del tipo di file da usare e l'accesso a questi
- 4 ore di test (distribuiti durante tutto lo sviluppo) e revisione finale

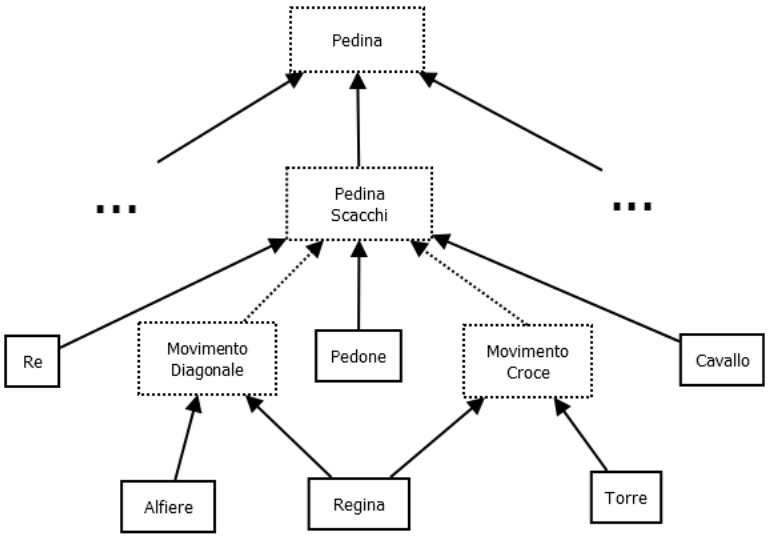
1.3 Compilazione ed esecuzione

Al fine di compilare ed eseguire correttamente l'applicazione è necessario lanciare i seguenti comandi:

1. `qmake -project QT+= widgets`
2. `qmake`
3. `make`
4. `./PolyBoard`

2 La gerarchia G di pedine

Il punto chiave del progetto è la gerarchia polimorfa che contiene le informazioni sulle pedine.



Nel diagramma sono indicate con un rettangolo tratteggiato le classi astratte; le derivazioni virtuali sono frecce tratteggiate.
Se si vuole aggiungere un gioco con pedine diverse da quelle degli scacchi, è necessario creare una nuova classe astratta che eredita da `Pedina` e creare come sottoclassi di questa una classe per ogni tipo di pedina.

- La classe base offre un'interfaccia minimale che si adatta praticamente a tutti i giochi di questo tipo:
  - `clone`: metodo per la clonazione profonda.
  - `controlloMossa`: metodo che ritorna le caselle attraversate dalla pedina per andare da un punto $A(x_1, y_1)$ di partenza ad un punto $B(x_2, y_2)$ di arrivo. Le caselle vengono tornate sotto forma di `std::list<Posizione>` dove `Posizione` è una semplice classe con campi dati interi x, y. Se la mossa non può essere fatta dalla pedina, questa ritorna una lista vuota (ad esempio provo a muovere la torre in diagonale, oppure $A == B$), altrimenti ritorna la sequenza di caselle attraversate sotto forma di coordinate unita alla coordinata di arrivo (questo per evitare che una mossa di una sola casella venga interpretata come mossa non valida).
    Questo è un metodo virtuale, visto che la pedina deve rispondere alla richiesta di spostamento secondo il suo tipo dinamico.
  - `getColore`: ritorna un tipo enum `Colore` che indica il colore della pedina, il quale identifica il giocatore. Il parametro colore è infatti messo nella classe base e può assumere colori diversi dal bianco/nero.
  - `getID`: ritorna un tipo `ID` che identifica univocamente il tipo di pedina, usato per il salvataggio dei file ed altre operazioni che richiedono di conoscere il tipo di pedina.
  - `pedinaMossa`: serve a comunicare alla pedina che è stata spostata nella scacchiera, in modo che possa cambiare i suoi dati, ad esempio il pedone sa che non è più la sua prima mossa. Infatti il metodo `controlloMossa` serve solo per interrogare la pedina su un tentativo di mossa, la quale risponde in base al suo tipo. Tuttavia non sa se sarà effettivamente mossa, dato che la traiettoria che deve essere percorsa potrebbe non essere libera, informazione che conosce solo il modello.
    Il metodo è dichiarato virtuale in quanto è sensato che le pedine sottostanti reinterpretino questa informazione passata dal model (non è il caso degli scacchi ma è utile fornire questa funzionalità per estensioni future).
  - `getPrimaMossa` che ritorna true se e solo se la pedina non è mai stata mossa. Anche questo metodo è stato reso virtuale nel caso si volesse implementare diversamente la prima mossa.

  Ha inoltre un campo dato booleano privato che indica se la pedina è stata mossa all'interno della partita. Prende il nome di `primaMossa` ed è `true` se e solo se la pedina non è mai stata toccata. Abbiamo pensato di porlo in cima alla gerarchia in quanto potrebbe essere utile in diversi giochi, tuttavia gli è stato assegnato un valore di default nel caso chi estendesse la gerarchia non avesse bisogno di questo campo. L'overhead di fatto non è rilevante in quanto il campo occupa un bit soltanto.
- La classe astratta `PedinaScacchi` cambia solamente il tipo di ritorno della `clone` con una covariante.
- Le classi `Re - Pedone - Cavallo` implementano il metodo controllo mossa secondo le regole degli scacchi.
  Nella classe pedone è stato assunto che le pedine bianche siano posizionate in basso (righe 6-7), mentre quelle nere in alto (righe 0-1) all'interno della scacchiera, in quanto possono andare in una sola direzione in base al colore. Quindi il model specializzato per gli scacchi agirà di conseguenza.
- Le classi astratte `MovimentoDiagonale - MovimentoCroce` implementano le regole per le pedine che si muovono in questo modo. In questo modo se i movimenti per queste pedine devono essere modificati basterà farlo a questo livello della gerarchia, non per ogni pedina.
- Le classi `Alfiere - Torre` ereditano rispettivamente da `MovimentoDiagonale - MovimentoCroce`.

- La classe `Regina` chiude l'ereditarietà a diamante ereditando da `MovimentoCroce` e `MovimentoDiagonale` poiché può muoversi sia a croce che in diagonale. Il suo controllo mossa chiama prima quello della superclasse `MovimentoCroce`, se ritorna una lista non vuota la ritorna a sua volta, altrimenti ritorna la lista ritornata dal metodo di `MovimentoDiagonale`.

3 Il container Scacchiera

Come già accennato, la struttura dati utilizzata è un array di dimensione fissata $N \times M$ visto come una matrice, dove N, M sono larghezza e altezza della scacchiera richiesta dallo specifico tipo di gioco (nel caso degli scacchi $N = M = 8$), parametri passati dal modello. L'array contiene puntatori polimorfi a pedine, dunque è stata implementata la *rule of three* per la gestione profonda della memoria. **Se una casella di coordinate (x, y) è vuota allora questa contiene** `nullptr`. Questa scelta implementativa ha portato all'impossibilità di templatizzazione:

- Se il container fosse templatizzato ad un generico parametro `T`, avrebbe come campo dati un `T*` che punta ad un array di `T`. Ne segue l'impossibilità di rappresentare celle vuote in quanto farlo con una pedina che rappresenta questo stato porterebbe ad una seria complicazione del codice. Inoltre la scacchiera non potrebbe fare i controlli in seguito descritti, i quali forniscono al model operazioni molto utili al fine di implementare le regole di molti giochi di questo genere.
- Viene subito in mente di istanziare il container con un'apposita classe `SmartP` che gestisca i puntatori polimorfi. Tuttavia in questo modo il container non può comunque verificare se una determinata traiettoria sia libera o meno, tratto caratterizzante del nostro container e necessario per i giochi su scacchiera. Inoltre con questa soluzione si renderebbe l'applicazione meno efficiente, in quanto ogni volta che una pedina viene mossa verrebbe creata di copia nella nuova casella e distrutta dalla vecchia. Con la soluzione corrente viene semplicemente spostato il puntatore nella nuova cella e inserito `nullptr` nella vecchia (eliminando eventualmente la pedina nella casella di arrivo, se presente).

Il container offre al modello i seguenti metodi nella sua interfaccia:

- `Pedina* operator[](pos)`: offre l'accesso all'elemento in posizione `pos` in tempo $\Theta(1)$ ritornando il puntatore alla pedina se la casella è piena, `nullptr` se è vuota. Se la posizione passata è out of bound si provoca una undefined behaviour, in quanto anche i container della std non effettuano questo genere di controlli.
- `bool isInBound(pos)`: ritorna `true` se e solo se `pos` indica una posizione valida all'interno della scacchiera.
- `bool insert(pedina, pos)`: inserisce la pedina passata nella posizione indicata solo se la casella corrispondente a quest'ultima è vuota, effettuando una copia profonda. In tal caso ritorna `true`, altrimenti se la casella era già occupata non effettua modifiche e ritorna `false`
- `void remove(pos)`: elimina, se presente, la pedina nella casella indicata dal parametro, altrimenti non effettua modifiche.
- `bool move(posFrom, posTo, bool force)`: innanzitutto controlla se `posTo` e `posFrom` sono in bound mediante l'apposito metodo. Se lo sono allora controlla il booleano `force`, se è `true` svuota la casella di arrivo con l'apposito metodo `remove`. Infine sposta il puntatore. Se la casella di destinazione non era vuota ritorna `false`, altrimenti se lo spostamento va a buon fine ritorna `true`
- `bool isFree(pos)`: ritorna true se e solo se la casella `pos` è in bound ed è vuota
- `Posizione find(Pedina*)`: ritorna la posizione del puntatore `p` in tempo lineare se esiste, altrimenti solleva l'apposita eccezione
- `bool traiettoriaLibera(list<Posizione> L)`: L deve essere non vuota, se il check fallisce ritorna `false` (se qui L è vuota qualcosa è andato storto prima). Dunque itera la lista e controlla cella per cella se queste sono vuote con l' apposito metodo, ritornando `true` se e solo se lo sono tutte. L'ultima casella non viene controllata per il semplice motivo che rappresenta la casella di arrivo, che potrebbe essere occupata

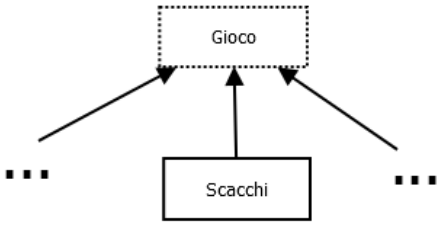
La scacchiera inoltre prevede 3 eccezioni:

- `BOARD_TO_SMALL`: sollevata quando si costruisce una scacchiera con parametri $N, M < 2$ (la scacchiera deve essere almeno 2×2 per definirsi tale);
- `ERR_CPY`: sollevata quando si tenta di eseguire l'assegnazione tra due scacchiere di dimensione diversa, in quanto non si saprebbe dove mettere le pedine;
- `ELEMENT_NOT_FOUND`: sollevata quando si cerca una pedina che non esiste all'interno della scacchiera con il metodo `find`, che non saprebbe quale `Posizione` ritornare.

3.1 Iteratori del container

Gli iteratori offerti sono `iterator` e `const_iterator`, i quali sono una versione ridotta di quelli presenti in `std::vector`, offrendo un'interfaccia minimale per scorrere la scacchiera con una scansione *row major*.
L'unica precisazione dev'essere fatta sul parametro puntatore privato dell'iteratore costante:
questo ha tipo `const Pedina* const*`, ovvero è un *puntatore non costante a puntatore costante a Pedina costante*; in pratica, scorrendo la scacchiera con il `const_iterator` non si può modificare né l'oggetto pedina (ovviamente) né il puntatore all'interno dell'array, in quanto questa operazione rappresenta lo spostamento delle pedine.

4 Il model Gioco - Scacchi



Per realizzare il modello è stata realizzata una gerarchia polimorfa contenente le classi per i vari giochi. In particolare c'è una classe base da cui far ereditare una classe per gioco, le quali dovranno contenere le informazioni riguardo il turno e su come costruire la scacchiera adatta. In generale conoscono le regole del gioco a cui si sta giocando.

- La classe base astratta `Gioco` ha come campi dati protetti il container `Scacchiera` e un parametro di tipo `Colore` che indica il giocatore di cui è il turno.
  Nella sua interfaccia pubblica prevede i seguenti *metodi astratti* (chiamate polimorfe):
  - `clone()`: il classico metodo di copia profonda necessario in una gerarchia polimorfa
  - `bool mossa(posIniziale, posFinale)`: effettua la corretta procedura per muovere una pedina da un punto A a un punto B secondo le regole del gioco corrente (quindi in base al tipo dinamico del model); ritorna `true` se la mossa è riuscita, quindi tutte le regole sono state rispettate, `false` se qualcosa non va nei parametri posizione passati
  - `cambioTurno`: serve per passare il turno al giocatore successivo una volta che quello corrente ha effettuato una mossa valida
  - `bool controlloVincitore`: ritorna `true` se e solo se il giocatore corrente ha vinto
  - `ID* getIdPedina(pos)`: metodo che effettua una chiamata polimorfa sulla pedina in posizione pos, se esiste, chiedendole il suo ID. Se tale pedina non esiste ritorna `nullptr`. Metodo necessario per il salvataggio della partita.
  - `tipoGioco()`: ritorna un identificatore del tipo di gioco a cui si sta giocando, necessario al controller per il salvataggio dei file

La classe `Scacchi`, come ci si aspetta, implementa i metodi astratti della superclasse secondo le regole degli scacchi. Inoltre aggiunge alcuni metodi necessari solamente per tale gioco, ovvero quelli necessari alla promozione del pedone a torre/alfiere/regina/cavallo nell'interfaccia pubblica, usati dal controller, mentre aggiunge i controlli di `scaccoAlRe` e `arrocco` (usati da `mossa`) e `scaccoMatto` (usato da `controlloVincitore`) nella parte privata. Questi metodi non verranno descritti nel dettaglio in quanto non utili nella visione globale che si vuole fornire.

5 La view

Abbiamo mantenuto la GUI dell'applicazione più pulita possibile, inserendo solamente una menu bar in alto per avviare nuove partite, accedere ai file, terminare una partita in corso. Queste azioni dovrebbero essere comuni a tutti i giochi che verranno implementati, dunque la `MainWindow` non dovrebbe essere estesa dal punto di vista grafico.
La scacchiera è rappresentata tramite una nostra estensione dei `QPushButton`, denominata `ChessButton`, la quale aggiunge un parametro `Posizione` che rappresenta la posizione del bottone all'interno della scacchiera. È quest'ultimo ad essere inviato tramite l'apposito segnale al controller per segnalare la selezione di una determinata cella.
Da notare che la scacchiera viene costruita della dimensione specificata dal model, il quale conosce su che scacchiera si gioca un determinato gioco.

5.1 Avvio di una nuova partita

Ritengo necessario fare una nota sul processo di avvio di una nuova partita poiché fa parte dell'ottica di estensibilità dell'applicazione.

In particolare è nella finestra `selettore_gioco` che andranno aggiunti i nuovi giochi. Infatti quando viene cliccata la action `NuovaPartita` parte uno scambio di segnali tra view e controller che porta alla creazione di un nuovo model in base al tipo di gioco scelto dall'utente che andrà a rimpiazzare quello già esistente, se esiste.

La GUI non richiede particolari indicazioni per essere utilizzata.

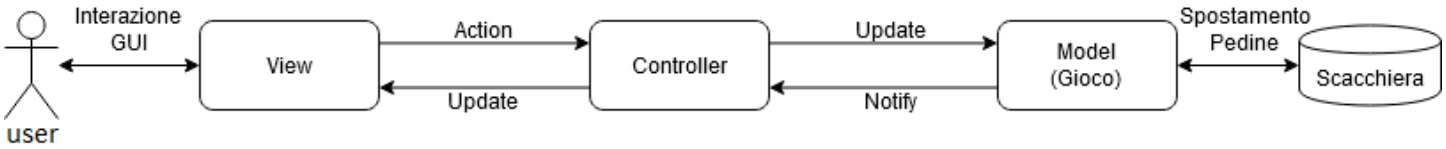
Infine c'è un selettore per la promozione del pedone, il quale è necessario per chiedere all'utente cosa deve diventare un pedone quando arriva all'ultima riga della scacchiera.

6 Il controller

Il controller dell'applicazione contiene due campi dati puntatori, uno di tipo `Gioco*` per il modello e l'altro di tipo `MainWindow*` per la vista. Da questa architettura ne deriva che view e model non possono comunicare direttamente, rendendoli completamente indipendenti l'uno dall'altro. Nella sua interfaccia pubblica prevede esclusivamente un metodo costruttore con un unico parametro per la parentela di Qt, visto che sarà agganciato ad una `QApplication`. Di default, quando viene creato, inizializza una nuova finestra principale ma non il model. Quest'ultimo rimane `nullptr` finché non viene avviata una nuova partita o caricato un file che rappresenti lo stato di una partita in corso.

7 Funzionamento globale

L'architettura del funzionamento è la seguente



L'utente esegue azioni sulla view (i.e. preme pulsanti della scacchiera). I pulsanti della scacchiera notificano la loro pressione al controller tramite un apposito segnale con parametro `Posizione`, indicante la loro posizione all'interno del `gridLayout`. Il controller raccoglie questa posizione e, una volta che ha sufficienti posizioni per eseguire una mossa (negli scacchi ne servono due) effettua una chiamata polimorfa al model per chiedergli di che gioco si tratta. Questo passaggio è necessario in quanto ogni gioco prevede dei metodi propri, dunque nel controller servirà un metodo turno per ogni tipo di gioco (ad esempio negli scacchi è necessario implementare il metodo per promuovere il pedone, cosa non necessaria in altri giochi). Questo metodo polimorfo ritorna un ID univoco che rappresenta il gioco all'interno dell'applicazione, incapsulato in un `enum TipoGioco`.
Di conseguenza viene passato il controllo a `mossaScacchi`, il quale effettua la chiamata a `mossa` del model per gli scacchi. Questo metodo fa diversi controlli secondo le regole degli scacchi, il più interessante e unico degno di nota è la chiamata polimorfa nella gerarchia di pedine al metodo virtuale `controlloMossa`, il quale ritorna una lista di posizioni attraversate dalla pedina selezionata, se questa può fare la mossa richiesta.
Infine, se la mossa è legale, allora il model aggiorna la scacchiera muovendo il puntatore della pedina selezionata ritornando `true`, altrimenti non effettua modifiche alla struttura dati e ritorna `false`.
Il controller, che riceve questo booleano di ritorno, sa se deve aggiornare la view per rappresentare la nuova situazione della scacchiera o mostrare un messaggio di "mossa non valida", dunque aggiorna la view.

8 Accesso ai file

L'applicazione usa i file con estensione `.json` per salvare lo stato di una partita in corso per poi poterla ricaricare in seguito. In particolare vengono usate le classi per questo tipo di file del framework Qt.
Vengono quindi memorizzate le seguenti informazioni:

- tipo di gioco in corso mediante un intero, rappresentante l'enum, apposito identificatore dei giochi all'interno di PolyBoard;
- un array contenente tutte le pedine, ogni elemento di questo contiene:
  - un array per l'ID, il quale contiene:
    - un char identificante la pedina *internamente* al gioco in corso
    - un intero rappresentante il colore, convertito dall'apposito enum che identifica il colore *globalmente* nell'applicazione
    - un booleano rappresentante lo stato di prima mossa
  - un array contenente la posizione, il quale contiene il valore x e y in prima e seconda posizione rispettivamente

Al caricamento di un file si effettua l'operazione inversa, costruendo un model del tipo specificato nel file senza pedine, le quali vengono inserite manualmente dal controller secondo i valori letti dal file.

In particolare nelle azioni di accesso ai file, sia in lettura che in scrittura, viene sfruttato il polimorfismo delle gerarchie di pedine e dei giochi:

- nel salvataggio vengono interrogate tutte le pedine presenti sulla scacchiera chiedendo il loro ID, le quali rispondono in base al loro tipo dinamico grazie al late binding. Per ogni cella che restituisce un id non nullo (dunque contiene una pedina) viene aggiunto elemento all'array delle pedine per il json file
- al caricamento di una partita salvata si crea il model del tipo dinamico indicato nel file. Poi si chiede a quest'ultimo, con una chiamata polimorfa, di convertire l'id passato ad un tipo pedina e di inserirlo in una determinata posizione. Questo interpreta l'id passato in base al suo tipo dinamico (infatti le pedine sono identificate univocamente solo all'interno del gioco), creando una pedina con i giusti parametri e del giusto tipo, infine la inserisce nella `Scacchiera`.

9 Classi ed enumeratori ausiliari

Nei paragrafi precedenti sono stati menzionati diversi tipi enumeratore e classi ausiliarie. Ritengo necessario fare chiarezza a riguardo.

- `TipoGioco`: enumeratore che serve ad identificare i giochi all'interno dell'applicazione. Al momento contiene solamente il valore `chess`, qui dovrà essere aggiunto un valore univoco per ogni gioco aggiunto (ad esempio `tictactoe` per il tris o `checkers` per la dama). Nel file inoltre c'è una funzione che traduce l'enum in stringa, che potrebbe essere utile ad esempio per creare il path alla cartella contenente le immagini relative.
- `Colore`: enumeratore, identifica globalmente i colori assegnati alle pedine. Contiene già dei valori con svariati colori, al momento vengono usati solo bianco e nero, tuttavia questo rende possibile cambiare colori agli scacchi o creare nuovi giochi con pedine diverse dal bianco e nero. Anche qui troviamo una funzione che traduce l'enum in stringa.
- `ID`: classe necessaria ad identificare univocamente i tipi di pedina. Punto chiave del funzionamento del progetto, è protagonista di una chiamata polimorfa nella gerarchie di pedine, le quali costruiscono un ID di ritorno in base a 3 parametri:
  - `tipo_pedina`: valore di tipo char assegnato in base al tipo dinamico nella pedina. Negli scacchi assume questi valori:
    - Re (KING): `K`
    - Regina (QUEEN): `Q`
    - Alfiere (BISHOP): `B`
    - Torre (ROCK): `R`
    - Cavallo (KNIGHT): `N`
    - Pedone (PAWN): `P`

    NB: questo campo identifica la pedina solamente all'interno del suo gioco, ad esempio potrebbe esserci un gioco diverso dagli scacchi che usa un char già utilizzato sopra. La conversione al giusto tipo viene fatta in base al tipo dinamico del model con una chiamata polimorfa al metodo `idToPedina`.
  - `colore`: inizializzato in base al campo colore della pedina, del tipo enumeratore sopra descritto
  - `primaMossa`: campo di tipo booleano. È inizializzato `true` $\iff$ la pedina non è stata mossa nella partita.

  Questa classe, oltre ad essere fondamentale per salvare la partita su file, ha evitato dei `dynamic_cast` per fare l'arrocco e per promuovere il pedone.
- `Posizione`: sistema di coordinate su un piano cartesiano a due dimensioni. Implementa operazioni aritmetiche di base utilizzate per fare conti tra queste nel model degli scacchi.

10 La versione Alpha in CLI

```
  0  1  2  3  4  5  6  7
0 R0 N0 B0 Q0 K0 B0 N0 R0
1 P0 P0 P0 P0 P0 P0 P0 P0
2
3
4
5
6 P1 P1 P1 P1 P1 P1 P1 P1
7 R1 N1 B1 Q1 K1 B1 N1 R1
Giocatore corrente: 1
Inserisci posizione iniziale:
2 6
Inserisci posizione finale:
2 4
Pedina mossa

  0  1  2  3  4  5  6  7
0 R0 N0 B0 Q0 K0 B0 N0 R0
1 P0 P0 P0 P0 P0 P0 P0 P0
2
3
4     P1
5
6 P1 P1    P1 P1 P1 P1 P1
7 R1 N1 B1 Q1 K1 B1 N1 R1
Giocatore corrente: 0
Inserisci posizione iniziale:
```

La prima versione dell'applicazione è nata per essere eseguita nella CLI e slegata completamente dalla libreria Qt (non implementava ancora i file). Le gerarchie del model e delle pedine sono rimaste invariate.
La realizzazione della prima versione funzionante in CLI ci ha richiesto approssimativamente 30 ore di lavoro collettivo (quindi poco più di 90 ore considerando le ore individualmente).
Questo ci ha permesso di ultimare i test e trovare i bug nel model, nel container e nella gerarchia di pedine prima di avviare lo sviluppo dell'altra parte consistente del progetto, ovvero view e controller con libreria Qt.

11 Estensibilità del codice

Di seguito verranno illustrate le parti di codice da modificare per aggiungere un qualunque gioco che preveda dei tipi di pedine su una scacchiera almeno 2×2 (non necessariamente quadrata).

11.1 Gerarchia di pedine

Come prima cosa è necessario far derivare una classe astratta da `Pedina` nella gerarchia di pedine, la quale sarà la classe base per tutte le pedine del gioco che si sta aggiungendo. Questa serve per cambiare il tipo di ritorno della `clone` con una covariante e ad aggiungere funzionalità comuni a tutte le pedine di quel gioco (se non già presenti in pedina). Sarà inoltre necessario definire un charset che identifichi univocamente le pedine internamente al gioco e fare l'override del metodo `getId`, necessario per effettuare il salvataggio della partita sui file. Le classi delle pedine implementano `controlloMossa` secondo le regole di movimento della pedina. Se più pedine prevedono gli stessi movimenti è bene fare una superclasse comune astratta a queste, come fatto negli scacchi con `MovimentoDiagonale` e `MovimentoCroce`.

11.2 Modello (gerarchia di giochi)

Successivamente è necessario far ereditare una classe da `Gioco` nella gerarchia di modelli, la quale dovrà implementare tutte le regole di quel gioco. Questa deve inizializzare la sottoclasse astratta con i parametri altezza e larghezza per la scacchiera, oltre che dirgli qual è il colore del giocatore per il primo turno. Dovrà quindi fare un override di `mossa, controlloVincitore, cambioTurno`, oltre che definire un nuovo valore identificativo nell'enum `TipoGioco`, il quale dovrà essere ritornato nell'override del metodo `tipoGioco`.
È obbligatorio inoltre fare un override del metodo protetto `idToPedina` che crea un nuovo oggetto pedina a partire da un id. Questo metodo viene usato dal metodo pubblico `inserisciPedina`, il quale è usato al caricamento di una partita da file e riceve come parametri l'ID e la posizione letta.

11.3 View

Per estendere la vista sarà necessario eseguire alcune operazioni:

- Innanzitutto bisogna aggiungere un `QPushButton` alla finestra di selezione gioco all'avvio di una nuova partita e, di conseguenza, tutti i segnali e slot necessari a far arrivare l'informazione al controller.
- Serve poi aggiungere nel metodo `aggiungiPedina` informazioni per creare il path corretto alla cartella contenente le immagini delle pedine aggiunte per il gioco.
- La classe `ChessButton` può essere riutilizzata per tutti i giochi in quanto non fa particolari cose, se non aggiungere una posizione relativa alla scacchiera per il bottone. Se necessario estendere questa classe (non dovrebbe esserlo).
- Infine è necessario aggiungere parti grafiche proprie di quel gioco, ad esempio la finestra di promozione del pedone per gli scacchi.

Infine è possibile creare un nuovo file `.css` per dare uno stile diverso ai bottoni della scacchiera.

11.4 Controller

Qui è necessario innanzitutto aggiungere un metodo per la mossa relativa a quel gioco e chiamarlo all'interno del metodo `raccogliPosizione`. Il metodo della mossa deve ricordarsi di deallocare i parametri `Posizione` raccolti una volta terminata (che sia andata a buon fine o meno).
Se sono stati aggiunti metodi nella view specifici per quel gioco è necessario fare i corrispondenti del controller ed usarli nel metodo prima creato.

Non sono necessarie altre operazioni, tutte le componenti dell'applicazione si adatteranno se le regole del gioco sono state implementate correttamente.
La scacchiera non necessita di essere modificata, visto che i metodi più comuni sono già stati implementati. Se fosse necessario aggiungere metodi si può fare nel relativo model iterando il container.