Relazione del progetto "L.A.M.A. Chess"

Marco Raggini, Angela Speranza, Lorenzo Tosi, Andrea Zavatta

Giugno 2022

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software che intendiamo proporre è una ripresentazione di uno dei giochi di strategia più antichi e famosi della storia: gli scacchi. L'obiettivo del progetto è quello di realizzare un ambiente in cui due giocatori possano sfidarsi in tempo reale. Tramite l'utilizzo di un'unica macchina, i due utenti giocano una partita: il vincitore è colui che riesce, per primo, a catturare il re avversario.

Requisiti funzionali

- La schermata iniziale conterrà un menù, grazie al quale l'utente può scegliere di iniziare una partita o visualizzare alcune impostazioni. Sarà inoltre disponibile un tutorial per coloro che desiderano ripassare le regole basilari del gioco.
- L'interfaccia principale della partita consisterà in una scacchiera 8x8, per un totale di 64 caselle, scure e chiare. Sul piano di gioco saranno disposti i vari pezzi: all'inizio della partita, questi sono allineati lungo le traverse più "esterne", di fronte ai rispettivi giocatori.
- Verrà gestito un semplice "database" locale, accessibile attraverso il menù. Questa funzionalità tiene traccia delle partite recenti, permettendo all'utente di visualizzare le sue statistiche, le mosse effettuate e gli ultimi avversari sfidati.
- Durante la partita, i giocatori muovono i pezzi: l'implementazione grafica proposta permetterà di "afferrare" i pezzi tramite l'utilizzo del mouse e di "trascinarli" su una casella, a patto che la destinazione sia valida.

- Il software dovrà gestire in modo efficiente eventi particolari, e.g.: il caso in cui il re è in scacco o addirittura in scacco matto, l'arrocco e il caso in cui la partita finisca in parità (patta).

Requisiti funzionali opzionali

- Sarà possibile personalizzare la scacchiera e le pedine secondo il gusto dell'utente.
- Verrà proposta una variante del gioco degli scacchi classico, gli scacchi "magici". Questa modalità colloca i pezzi pesanti (re e regina, alfieri, cavalli e torri) in una posizione casuale all'interno della loro traversa. La rivisitazione in questione può risultare interessante per tutti quegli scacchisti che desiderano cimentarsi in una partita un po' fuori dagli schemi.
- Il programma sarà in grado di gestire la mossa dell'en passant.
- Sarà possibile, per l'utente, giocare una partita in LAN (dunque su due macchine diverse).
- Il programma sarà in grado di gestire delle partite a tempo attraverso un timer: il primo dei due giocatori ad aver finito il tempo a sua disposizione, avrà perso la partita.

1.2 Analisi e modello del dominio

La challenge principale del nostro programma è quella di gestire tutti gli aspetti più importanti di una partita di scacchi. Il gioco degli scacchi è ricco di regole ed è potenzialmente imprevedibile, per questo motivo il principale goal del team è stato individuare i maggiori aspetti del gioco e sviscerare nel dettaglio tutte le casistiche a cui uno scacchista può andare in contro in una partita. I principali attori che costituiscono le fondamenta del programma (rappresentati nella Figura 1.1) sono:

- Il Game, nonché la classe centrale, responsabile della gestione dell'intera partita. Essa contiene e gestisce il concetto di turno, di vincitore, oltre a tenere traccia dello stato della partita.
- La Chessboard di gioco, che gestisce i pezzi, il loro stato attuale e i loro movimenti; è fondamentale in quanto costituisce un elemento dinamico i cui aggiornamenti sono il punto di partenza per ogni turno.

- Il Turno, che si occupa della gestione dei due giocatori e li connette alla scacchiera.
- Il Giocatore (User) con la sua caratterizzazione, personalizzabile in ogni match.
- Il Pezzo, che ha una posizione corrente all'interno della scacchiera e informazioni sulle posizioni valide in caso lo si voglia muovere.
- La Mossa: è uno dei concetti più importanti, poiché deve necessariamente rispettare una serie di regole che non riguardano soltanto i pezzi stessi, ma anche lo stato corrente della scacchiera.

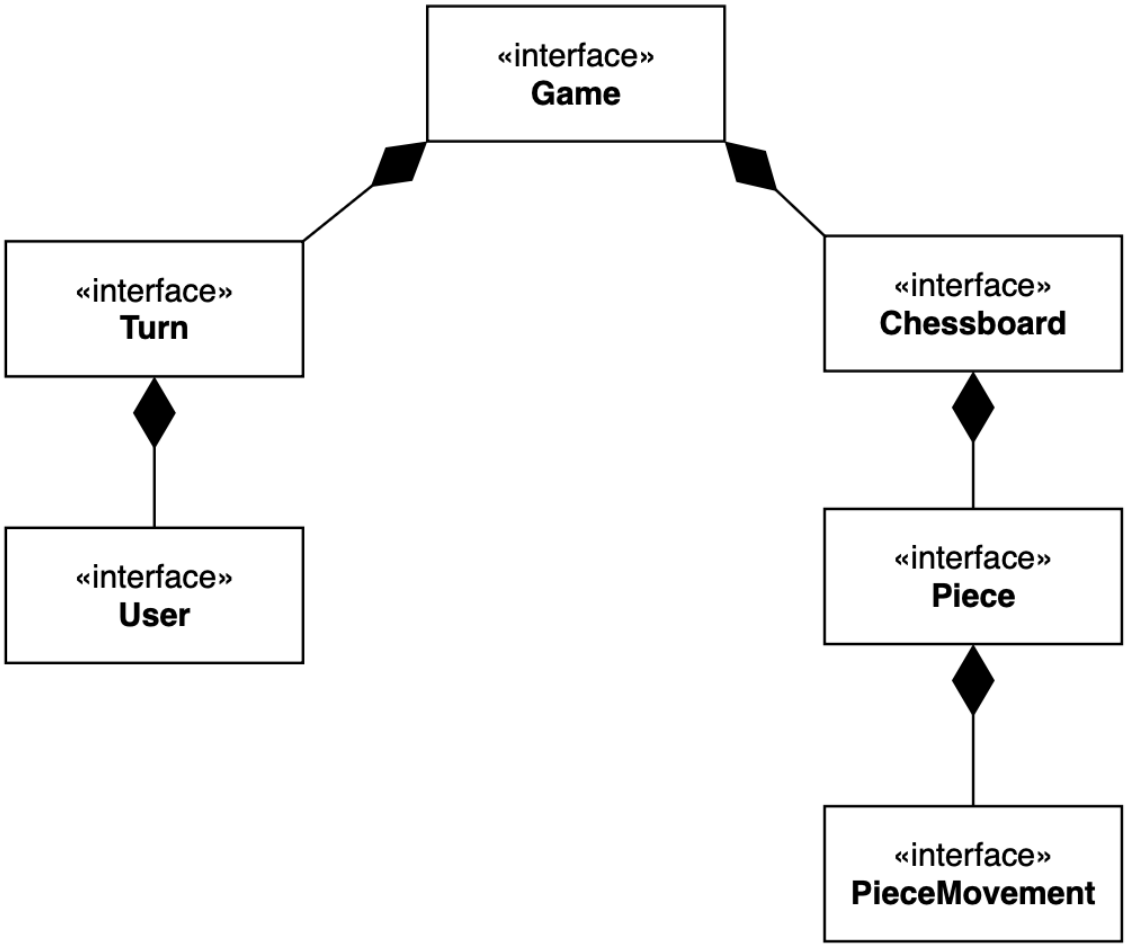

Figura 1.1: Uno schema riassuntivo delle principali entità e delle relazioni che intercorrono tra loro.

Capitolo 2

Design

2.1 Architettura

L.A.M.A. Chess sfrutta il pattern architetturale MVC. Ogni singolo aspetto di questo pattern funziona in autonomia; in particolare l'intero model si basa sulla classe Game, che coordina il funzionamento della partita e tutte le funzionalità da noi aggiunte. I componenti della view sono stati gestiti con JavaFX, tramite la creazione di file fxml. Tutti questi elementi, costituenti la vera struttura del progetto, sono stati uniti e interconnessi dal controller, nello specifico dalla classe BoardController. Quest'ultima garantisce il funzionamento dell'applicazione anche nel caso si desiderasse utilizzare una GUI diversa (non necessariamente implementata con JavaFX).

Figura 2.1: La nostra implementazione del pattern MVC.

2.2 Design dettagliato

2.2.1 Marco Raggini

MODEL

Il mio compito all'interno del model era quello di gestire la scacchiera e assemblare tutti i controlli e i pezzi creati dai miei colleghi al fine di implementare una partita funzionante.

Relazione tra Chessboard e Game

La prima scelta di design che voglio presentare è l'idea di dividere concettualmente la scacchiera dall'idea di partita, costruendo la scacchiera come sola funzione quella di mettere insieme tutti i pezzi e muoverli a seconda delle loro possibili mosse, mentre, la partita usufruisce della scacchiera per poter funzionare e aggiunge tutte le componenti necessarie per giocare una partita, come ad esempio i giocatori, i turni, la vittoria o il pareggio. Per questo motivo ho creato la classe Chessboard, che rappresenta la scacchiera e comunica con i pezzi all'interno di essa e la classe Game, che gestisce la partita all'interno del metodo nextMove dove sono presenti tutti i controlli necessari.

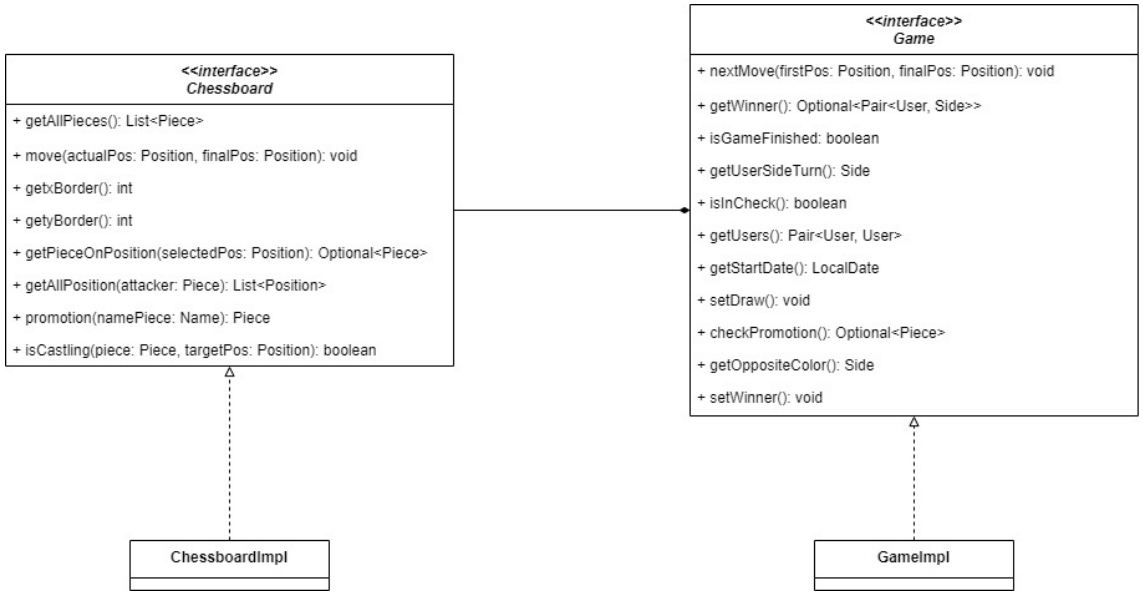

Figura 2.2: Relazione tra Chessboard e Game.

Problema 1: Un problema sorto all'interno della creazione della classe Chessboard era quello di non voler creare i pezzi all'interno della stessa, que-

sta decisione è stata presa per implementare futuri tipi diverse di scacchiere dove i pezzi possono essere posizionati in modo differente.
Soluzione 1: l'utilizzo del pattern Factory. Mediante la classe ChessboardFactory le scacchiere possono essere create tramite i metodi presenti all'interno di essa, inoltre, questa classe ha reso la creazione dei test molto più semplice tramite la possibilità di cambiare i pezzi nella scacchiera in partenza.

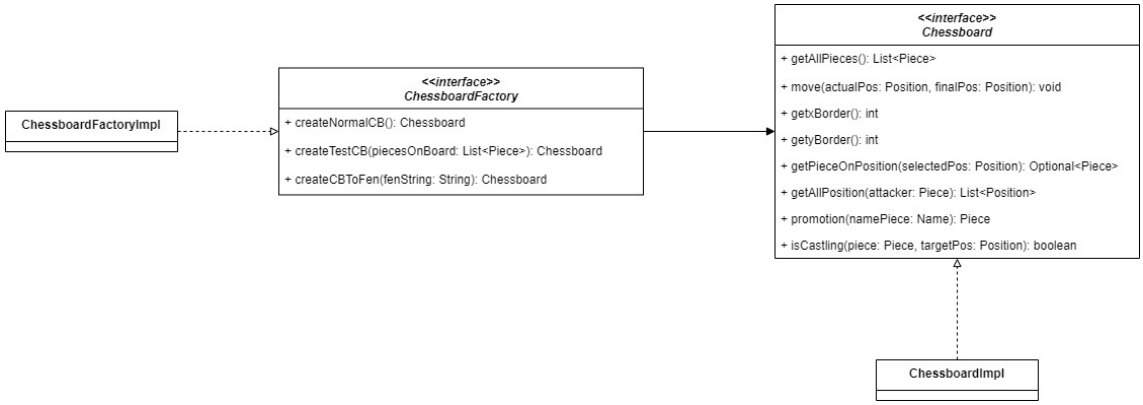

Figura 2.3: Schema della Chessboard.

CONTROLLER
Il mio compito in questo ambito è stato la creazione del controller UserHandlerController per l'interfaccia della scelta del nome e immagine utente e ho contribuito alla realizzazione di BoardController dove principalmente ho dovuto collegare le immagini dei pezzi ai relativi pezzi del model, aggiornare le caselle all'interno della scacchiera dopo ogni mossa e tutto quello che riguarda il collegamento tra le casistiche di gioco e gli effetti visivi dell'applicazione. Problema 2 Il problema principale di questa parte è stato il collegamento tra le immagini dei pezzi e i relativi pezzi del model, in quanto, ogni pezzo(Re, Regina, Cavaliere, ecc..) aveva due immagini differenti, nel caso il pezzo fosse bianco oppure nero. Soluzione 2 La creazione di una Factory ed un Enum. L'Enum è stato utilizzato per tenere traccia di ogni percorso per l'immagine richiesta, insieme a una funzione che, dato un colore restituisce l'immagine corretta. Mentre la Factory è stata utilizzata per creare, dato un qualsiasi pezzo, la sua corretta immagine da inserire poi all'interno della scacchiera.

VIEW
Il mio compito all'interno della view è stato quello di creare l'interfaccia per la scelta del nome e dell'immagine utente, inoltre ho anche creato la

visualizzazione delle caselle disponibili in cui il pezzo selezionato si può muovere.

2.2.2 Angela Speranza

Il mio ruolo all'interno del team è stato abbastanza versatile e ho avuto l'occasione di spaziare tra i diversi aspetti del pattern MVC.

MODEL
Il primo compito di cui mi sono occupata è stato quello di creare un punto di partenza per la nostra applicazione, il Launcher. Successivamente, il mio lavoro nella parte di model è stato temporaneamente messo in pausa: la mia parte, infatti, consisteva soprattutto nell'implementazione della mossa dell'arrocco e di tutte le casistiche di fine partita. Per riuscire ad implementare queste funzionalità, avevo infatti bisogno sia della scacchiera, sia che i pezzi si muovessero in modo corretto.

Castling / Castling Impl
Quella dell'arrocco è una mossa complessa, specialmente perché richiede una serie di condizioni per essere effettuata. Ho scelto di rendere pubblico un metodo booleano che potesse aiutare Marco nella gestione del Game. L'arrocco è una mossa difensiva che serve a fornire ulteriore protezione al re, con l'aiuto della torre. Il re si sposta di due case alla sua destra o alla sua sinistra, e la torre dello stesso lato passa al fianco opposto del re. Le case tra questi due pezzi devono essere necessariamente vuote e quelle che il re deve attraversare devono anche essere non minacciate dai pezzi nemici. Vi sono anche altri vincoli: il re e la torre coinvolti nell'arrocco non devono mai essere stati mossi e, infine, non ci si deve trovare in scacco.

Figura 2.4: Schema esemplificativo di una condizione favorevole all'arrocco.

Tutte queste condizioni devono essere accuratamente controllate, ed è stato dunque necessario avere un continuo e preciso accesso alla scacchiera e allo stato dei pezzi. Il mio sforzo maggiore è stato concentrato sul rendere l'implementazione dell'arrocco il più generale e astratta possibile. L'idea principale è che questa mossa non dovesse in alcun modo dipendere né dal colore del giocatore, né dal lato in cui si è scelto di arroccare. In altre parole, non ho effettuato una vera e propria distinzione tra arrocco lungo e arrocco corto: queste due tipologie di mosse, infatti, dipendono strettamente dal colore del giocatore. La soluzione alla quale sono arrivata sfrutta la costruzione intrinseca della scacchiera. Ho fatto in modo che fosse necessario considerare unicamente il re e una torre, quest'ultima identificata soltanto dalla sua coordinata x, ovvero la sua colonna di partenza. Aiutandomi con i controlli di Andrea per verificare che il re non sia in scacco e implementando in autonomia le altre verifiche, sono riuscita nell'intento iniziale. Non è stato nemmeno necessario dover specificare che l'arrocco può verificarsi soltanto una volta per ogni colore in una partita in quanto, cambiate le condizioni di partenza sopra descritte, il booleano pubblico ritornerà falso.

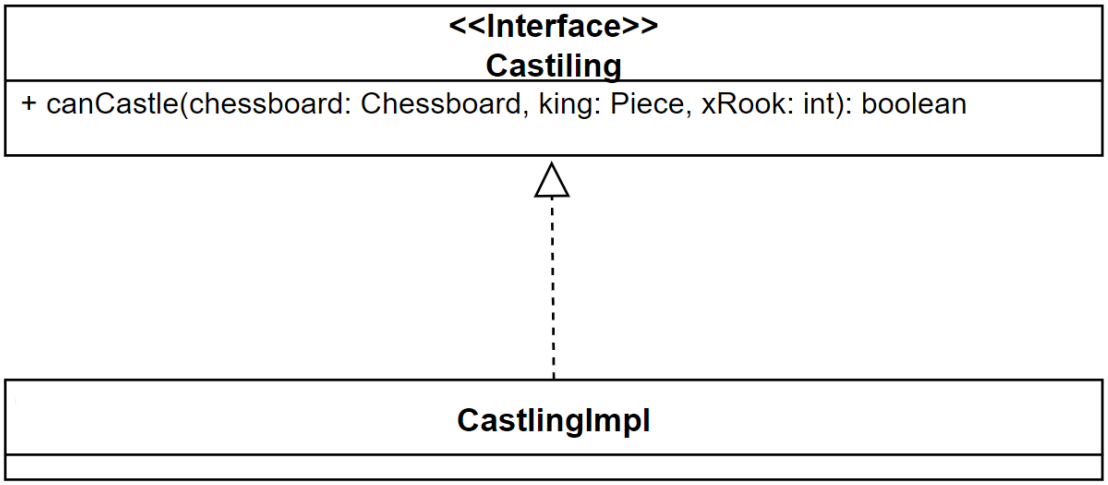

Figura 2.5: Soluzione scelta per l'arrocco.

Endgame / EndgameImpl
Il mio altro compito, secondo gli accordi, era modellare le diverse modalità con le quali una partita di scacchi può finire. Nella realtà, una partita può finire anche in maniera "forzata": questo succede nel caso di una patta concordata o un ritiro. Questi due possibili scenari non hanno bisogno di particolari controlli, per questo motivo sono stati gestiti tramite un semplice bottone nella GUI collegato al Game nel model. La mia parte, invece, si è concentrata sui seguenti casi: lo scacco matto, la patta per stallo e la patta per materiale insufficiente. In tutti i casi è stato, come per l'arrocco, assolutamente indispensabile disporre di uno stato della scacchiera sempre aggiornato. Tutte le situazioni da gestire sono state costruite a partire dalla lista dei pezzi rimanenti sulla scacchiera. Grazie ad uno strumento potente come quello delle stream, è stato necessario soltanto filtrare queste liste al fine di ottenere e verificare le condizioni necessarie al verificarsi dei diversi endgame. Nel caso della patta per materiale insufficiente, è necessario verificare che, per entrambi i colori, i pezzi rimasti in vita sulla scacchiera non riescano a dare scacco matto. La patta per stallo, invece, presuppone l'assenza di mosse disponibili per il re attaccato, oltre che per i pezzi alleati. Lo scacco matto, invece, oltre a dover imporre al re l'impossibilità di movimento, deve anche controllare che nessuno degli altri pezzi tra gli alleati possa bloccare la traiettoria del pezzo nemico che minaccia il matto. Anche in questo caso

la mia scelta è ricaduta su dei metodi booleani, molto utili da utilizzare nel Game e molto versatili nella gestione dei turni per decretare il vincitore.

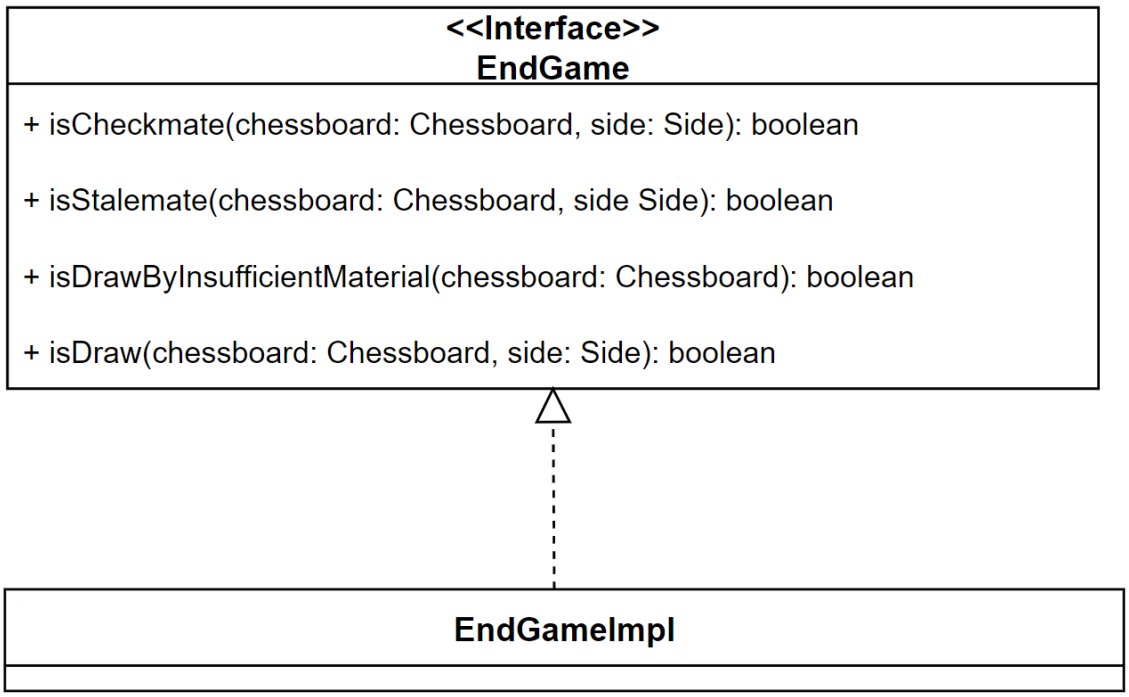

Figura 2.6: Soluzione scelta per la gestione della fine della partita.

ChessTimer / ChessTimerImpl
In questi giorni ho avuto anche modo di realizzare il Timer, una feature opzionale. Ho trovato interessante sviluppare questa classe anche per arricchire la mia conoscenza sulla programmazione Multithread introdotta nel corso dal professor Ricci. Questa funzionalità è stata costruita grazie alla classe Timer del package java.util, in particolare facendo uso del metodo "scheduleAtFixedRate". In realtà sono due i timer da gestire, uno per ogni giocatore: effettuata una mossa, quello dell'avversario deve iniziare a scorrere. Nella pratica, però, questa classe di utility mi ha permesso di gestire entrambi i clock, rappresentati con due label di JavaFX, tramite l'utilizzo di un unico Thread. A questo punto è bastato gestire il comportamento della TimerTask() richiesta come argomento del metodo per portare a termine il mio compito. È stato inoltre necessario dover gestire il fatto che, in caso uno

dei due giocatori finisca il tempo a disposizione, la partita debba terminare con un vincitore. Ho risolto questo problema utilizzando un'interfaccia funzionale, Consumer¡T¿. Ho implementato un metodo che, richiamato nel controller della scacchiera, permette di "ascoltare" l'evento "tempo scaduto" per fermare il thread del timer e ritornare il vincitore della partita.

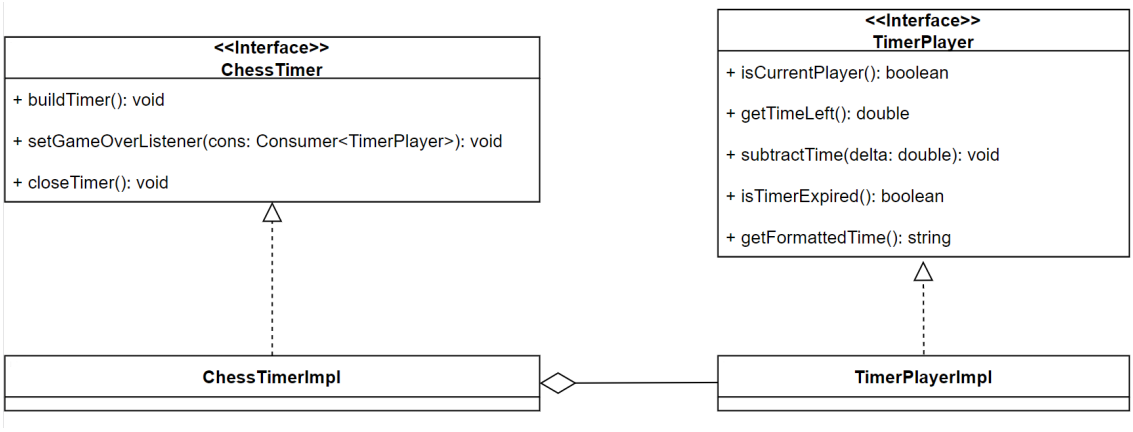

Figura 2.7: Schema per la gestione del Timer e dei due giocatori.

TimerPlayer / TimerPlayerImpl
Queste classi modellano un'estensione di un oggetto già esistente, lo User di Marco. La differenza tra un TimerPlayer e un giocatore base è, come si evince facilmente, quella di avere un certo timer che scorre durante il suo turno. Questa classe si occupa appunto di gestire il tempo rimasto prima dello scadere del timer, aggiornarlo durante la durata del turno e formattare la stringa che la Label deve mostrare sulla View. MatchDuration Ho implementato anche una semplice enum per gestire le diverse durate delle partite ed evitare i cosiddetti *magic numbers*.

VIEW
Per quanto riguarda la view, il mio compito è stato quello di realizzare e curare il menù principale e un piccolo tutorial che spiegasse le regole base degli scacchi. Per realizzare queste viste ho approfondito JavaFX e ho anche trovato un significativo aiuto nel tool SceneBuilder, che ci ha permesso di spaziare con la fantasia. Infine, ho avuto anche modo, anche se in piccola parte, di interfacciarmi con il css per fornire al nostro progetto un ulteriore tocco personale.

CONTROLLER
Il mio ruolo nella costruzione del controller è strettamente connesso con quello che ho ricoperto nella view. Mi sono infatti occupata di modellare il comportamento delle viste precedentemente create. Infine, ho dato un piccolo contributo anche al controller della Chessboard aggiungendo il Timer da me implementato.

2.2.3 Lorenzo Tosi

Nella realizzazione di LAMA-CHESS mi sono principalmente occupato dei pezzi, del loro movimento e della loro creazione per quello che riguarda il lato model. Dal punto di vista della view, ho creato la scacchiera di gioco e, infine, lato controller ho sviluppato, assieme ai miei colleghi, il controller della scacchiera di gioco.

MODEL
Il mio compito in questo ambito è stato la gestione completa del pezzo.

Rappresentazione dei pezzi

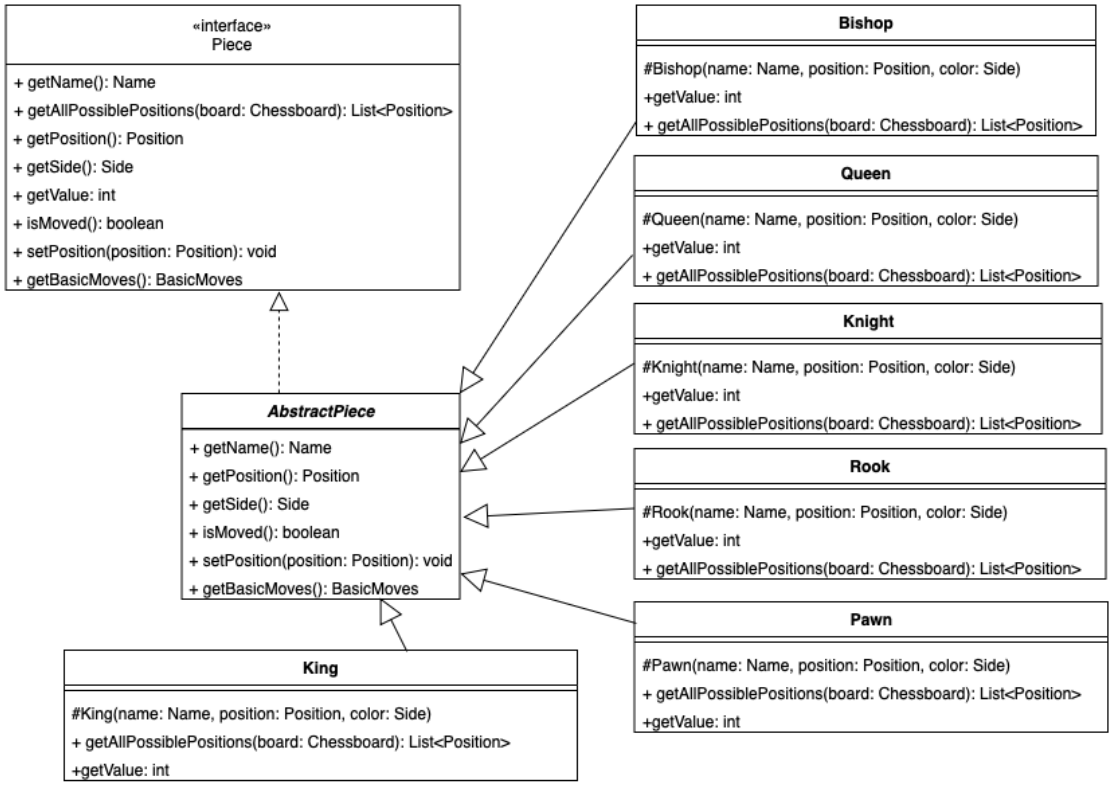

Figura 2.8: Schema del pezzo.

Piece è l'interfaccia che rappresenta il singolo pezzo degli scacchi. Essa permette di ottenere il nome del pezzo e il suo colore, gestire la sua posizione, oltre a contenere l'informazione che indica se il pezzo è stato mosso e identificare tutte le posizioni nelle quali si può muovere.

Problema 1: Ogni pezzo deve avere un certo comportamento, anche se gran parte delle caratteristiche sono comuni in ogni tipologia.
Soluzione 1: La soluzione scelta è stata quella di utilizzare il Template Method Pattern ed avere quindi una abstract class, AbstractPiece, che racchiude gli elementi comuni dei diversi pezzi. Questa soluzione ha quindi permesso

di minimizzare la duplicazione di codice raggruppando gran parte dei comportamenti in un'unica implementazione.

Problema 2: I comportamenti specifici dei diversi pezzi devono essere gestiti in modo particolare, ad esempio essi devono essere "coscienti" di tutte le posizioni nelle quali possono muoversi.
Soluzione 2: Questi comportamenti, che sono nello specifico il calcolo di tutte le posizioni possibili e la conoscenza del proprio valore, non vengono quindi implementati nell'AbstractPiece, ma nella classe specifica del pezzo.

Problema 3: Il movimento, in particolare il calcolo di tutte le posizioni possibili è principalmente di 2 tipologie. La prima itera l'intera scacchiera seguendo un comportamento fisso finché non arriva ai bordi o fino a quando non incontra un altro pezzo; la seconda consiste solo nel controllare delle posizioni prestabilite.
Soluzione 3: Per ridurre la ripetizione di codice, viene fatto uso di un'interfaccia, la PieceMovement, che gestisce queste due tipologie di calcolo, e, per ogni pezzo, viene richiamata la tipologia corretta. Fa eccezione il Pawn (Pedone) che, poiché possiede un comportamento unico, ha un'implementazione "ad-hoc" all'interno della propria classe.

Rappresentazione del legame tra la PieceMovement e il Piece
L'interfaccia PieceMovement contiene solamente due metodi in grado di produrre una lista contenente tutte le posizioni disponibili nelle quali il pezzo può muoversi.

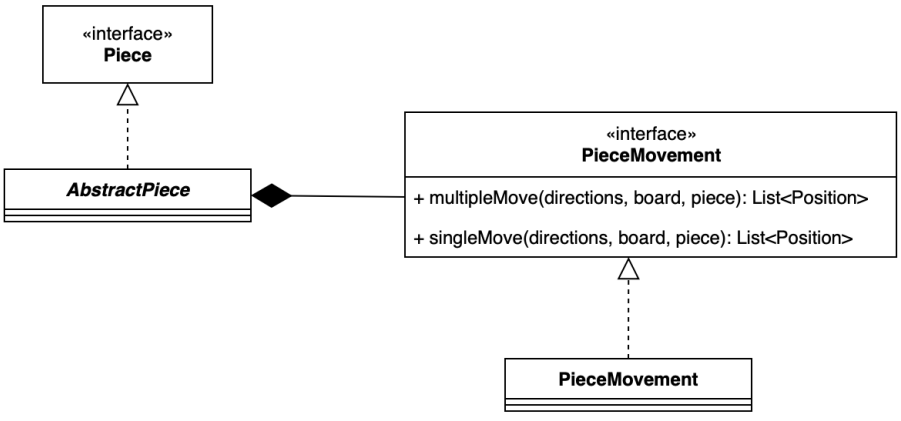

Figura 2.9: Relazione tra pezzo e movimento.

Come si può osservare, ogni pezzo ha necessariamente bisogno di un og-

getto di tipo PieceMovement, in quanto senza il movimento, esso è pressoché inutile e soprattutto inutilizzabile.

Problema 4: Nel Problema 3 viene affrontata la questione relativa a come ogni singolo pezzo possegga delle istruzioni personali ed uniche per il suo movimento.
Soluzione 4: Viene fatto uso di un'enumerazione chiamata PieceDirections che al suo interno, per ogni tipologia di pezzo, contiene il set di istruzioni - sotto forma di lista - che rappresenta tutte le sue direzioni.

Creazione dei Pezzi

Problema 5: La creazione dei pezzi.
Soluzione 5: Per risolvere questo problema ho deciso di utilizzare il template Factory Method in quanto riduce notevolmente il numero di "new" nel codice. Per creare i pezzi, si è dunque vincolati ad usare questa factory in quanto ho impostato come "protetto" il costruttore di ogni singolo pezzo. Quindi la PieceFactoryImpl, classe che implementa l'interfaccia PieceFactory, contiene un solo metodo pubblico che crea un nuovo pezzo, nella posizione selezionata e con il colore scelto dal programmatore.

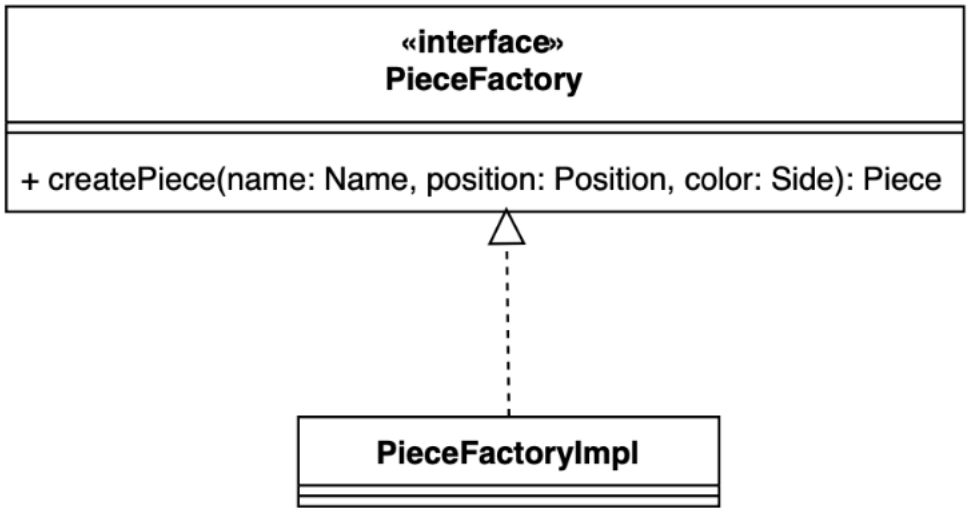

Figura 2.10: Factory dei pezzi.

La promozione del pedone

Problema 6: Gestire la promozione del pedone, ovvero quella situazione in cui un pedone arriva nell'ultima riga (quella avversaria) e viene "promosso", o meglio sostituito, con un pezzo pesante a scelta dall'utente.
Soluzione 6: Ho sviluppato una semplice interfaccia che controlla se nell'ultima riga è presente un pedone, e se è presente questo viene cambiato.

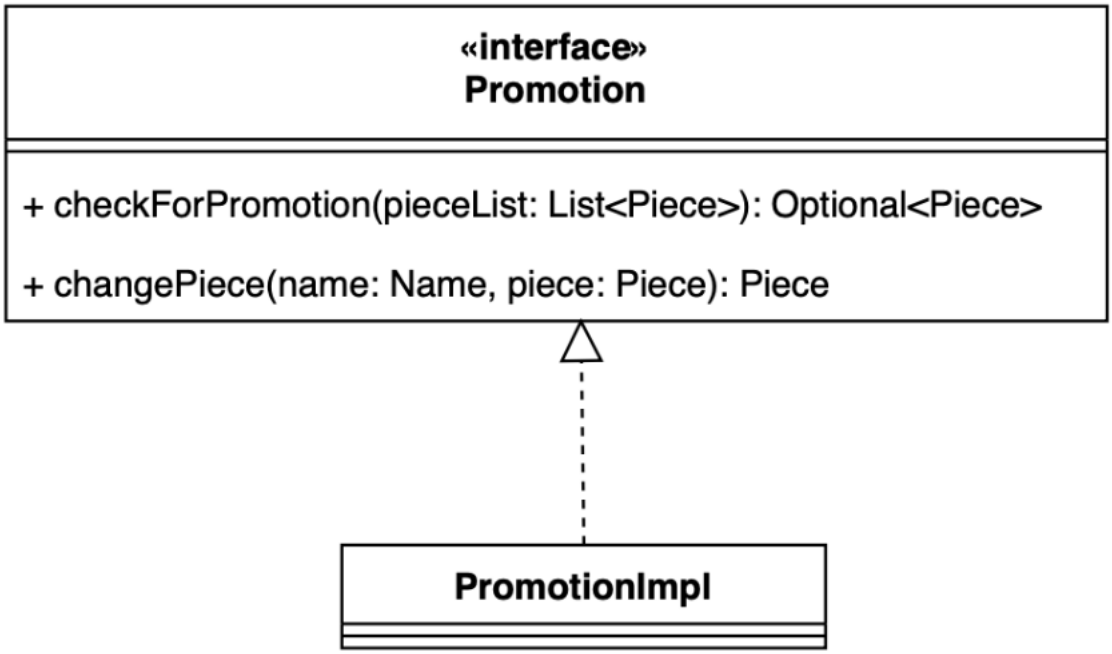

Figura 2.11: Schema della promozione.

Considerazioni finali Gli elementi necessari per la creazione del pezzo sono il nome, il colore e la posizione.

Problema 7: Come modellare il nome e il colore del pezzo.
Soluzione 7: Ho deciso di modellare il nome e il colore attraverso delle enumerazioni visto che essi sono elementi non modificabili.

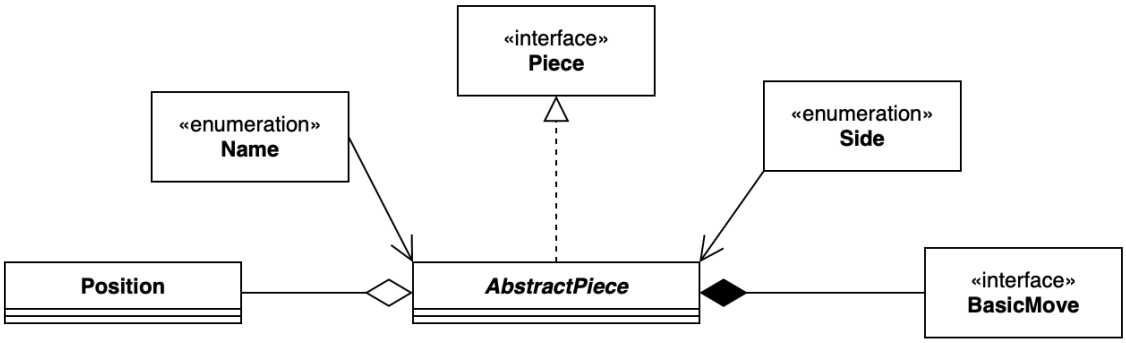

Figura 2.12: Composizione del Pezzo.

Problema 8: Come gestire la posizione del pezzo.
Soluzione 8: Ho modellato la posizione attraverso una classe Position. In questa classe ho utilizzato il pattern Static Factory Method in quanto è stato necessario, per la comodità mia e dei miei compagni, poter creare una Position sia numericamente, passando una coordinata x e una y, sia passando la notazione scacchistica che indica una certa casella (esempio "a3", "c6").

VIEW
Il mio compito in questo ambito è stato creare graficamente la schermata di gioco. Per svilupparla in JavaFX, mi sono aiutato con SceneBuilder, un programma che guida l'utente nella creazione della vista, permettendo di posizionare tutti i componenti nel posto desiderato. L'ultima parte di cui mi sono occupato è stata la creazione dell'entità GuiPiece, ovvero il "pezzo" che l'utente vede nella scacchiera, e che può muovere. Ho creato una classe che contiene tutte le informazioni necessarie per rendere la sua creazione e il suo utilizzo semplice. Graficamente il GuiPiece è rappresentato da un rettangolo al quale ho dato un'immagine per rappresentare il pezzo, il quale è trasportabile all'interno della scacchiera con il "drag-and-drop".

Figura 2.13: Composizione del GuiPiece.

CONTROLLER
Il mio compito per ciò che riguarda il controller consisteva nella gestione di una parte della classe BoardController. Qui ho implementato il drag-and-drop, ho inizializzato gli utenti, la scacchiera e il pop-up di fine partita con la descrizione dell'esito.

2.2.4 Andrea Zavatta

Nello sviluppo di L.A.M.A Chess, mi sono occupato soprattutto del salvataggio su file e del controllo delle mosse, implementando anche classi riguardanti i parsers.

MODEL
Controllo delle mosse

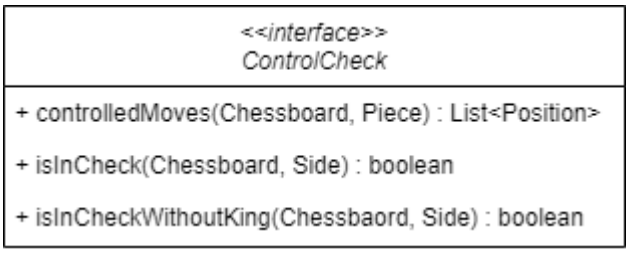

Figura 2.14: L'interfaccia ControlCheck.

Nel gioco degli scacchi il re non può muoversi in una casella attaccata da un pezzo avversario; inoltre, un pezzo che copre il re dallo scacco, non può spostarsi e di conseguenza lasciare il re libero di essere catturato da un pezzo avversario. Il mio compito consiste nel controllare che non sia ammissibile alcuna mossa che renda il re del giocatore che sta muovendo sotto scacco. Questa condizione è stata gestita attraverso l'interfaccia ControlCheck, che, oltre a fare questo controllo, espone dei metodi per verificare se il re è sotto scacco. Di seguito spiego il motivo per cui è stato inserito un metodo che potrebbe risultare non chiaro: isInCheckWithoutKing.

Problema 1: è necessario trovare un modo per ottenere le mosse disponibili del re evitando che le funzioni che gestiscono le mosse si richiamino a vicenda creando un loop.

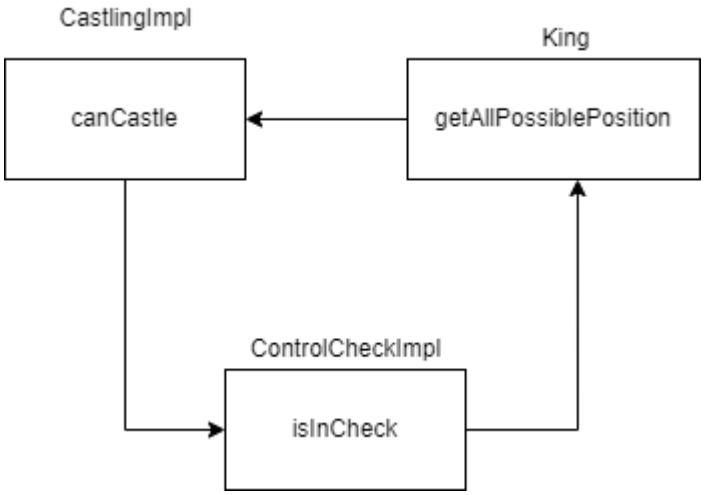

Figura 2.15: Loop da risolvere.

Soluzione 1:ho scelto di utilizzare la funzione isInCheckWithoutKing invece di IsInCheck. In questo modo non viene richiamata la funzione getAllPossiblePosition sul re, ma soltanto su tutti gli altri pezzi.

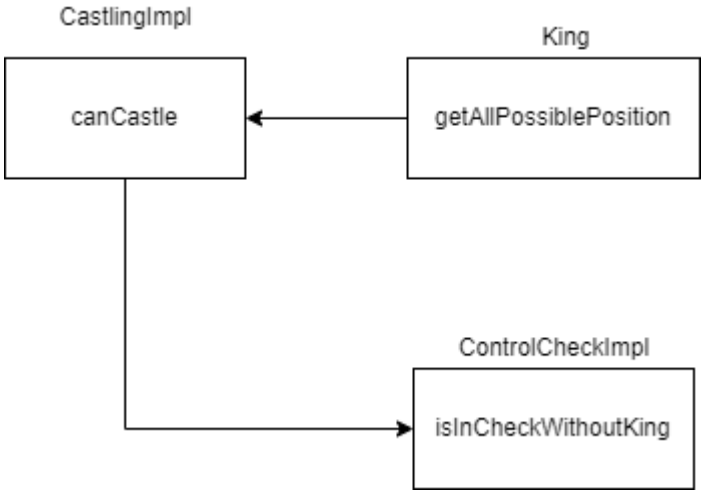

Figura 2.16: La soluzione trovata.

Parsers
Mi sono occupato anche dell'implementazione di interfacce che gestiscono la conversione in stringa di determinati oggetti e viceversa. Il mio obiettivo principale è stato quello di trovare un modo per semplificare, specialmente nei test, la gestione di una scacchiera sulla quale sono già state effettuate

delle mosse. Grazie alla notazione FEN, è possibile, attraverso una semplice stringa, ricostruire da zero una specifica posizione di gioco. Di seguito riporto il link per informarsi sulla notazione FEN (per la rappresentazione di una determinata posizione di gioco) e sulla notazione algebrica (per rappresentare le mosse):

- `https://it.wikipedia.org/wiki/Notazione_Forsyth-Edwards`
- `https://it.wikipedia.org/wiki/Notazione_algebrica#:~:text=Per%20indicare%20una%20mossa%2C%20la,un%20pedone%20muove%20in%20e4`

Fen
Si tratta di un'interfaccia che converte un oggetto di tipo Chessboard in una stringa che rappresenta il suo stato attuale. La stringa restituita contiene tutti e sei i campi di una stringa FEN tradizionale.
FenConverter
Il compito di questa interfaccia è quello di convertire una stringa Fen in un oggetto di tipo Chessboard. Dei sei campi disponibili dalla conversione FEN, questa classe tiene conto soltanto del primo, quello relativo alla posizione dei pezzi nella scacchiera, tralasciando gli altri cinque che implementano funzioni a noi inutili. L'interfaccia fenConverter è stata integrata nella classe ChessboardFactoryImpl, e ampiamente utilizzata in fase di test.
Move
Un'ulteriore interfaccia il cui compito è quello di convertire una determinata mossa in una stringa che la rappresenta.
NOTA: Le interfacce Move e Fen sono state implementate e testate, ma non si è presentata occasione per integrarle con le varie classi a livello di applicazione, in quanto trattavano di aspetti opzionali non ancora sviluppati.

Problema 2: Le implementazioni delle interfacce Fen e Move necessitano di informazioni molto specifiche per funzionare. Per rendere meglio il concetto, l'interfaccia FEN dipende strettamente dallo stato corrente della Chessboard. Nel caso della Move, invece, è necessario avere informazioni più dettagliate sul tipo di mossa (se è un movimento o una cattura), sul tipo di pezzo e sul movimento effettuato (vanno specificate la posizione di partenza e quella di arrivo).
Soluzione 2: Ho utilizzato il pattern Builder per risolvere il problema. Solo quando alla classe arrivano tutte le informazioni necessarie, si può procedere alla conversione richiesta.

Gestione file
Ho preferito creare due classi, una per la lettura e una per la scrittura su

file, rispettando così il Single-Responsibility principle. Nel momento del salvataggio viene creata un cartella di nome LAMAChess nella home directory, al cui interno viene inserito il file “database.txt” in cui vengono salvate le partite. Dunque, ho scelto di utilizzare un percorso assoluto rispetto ad un percorso relativo, poiché in questo modo non si perdono dati se si sposta il file jar dalla cartella.

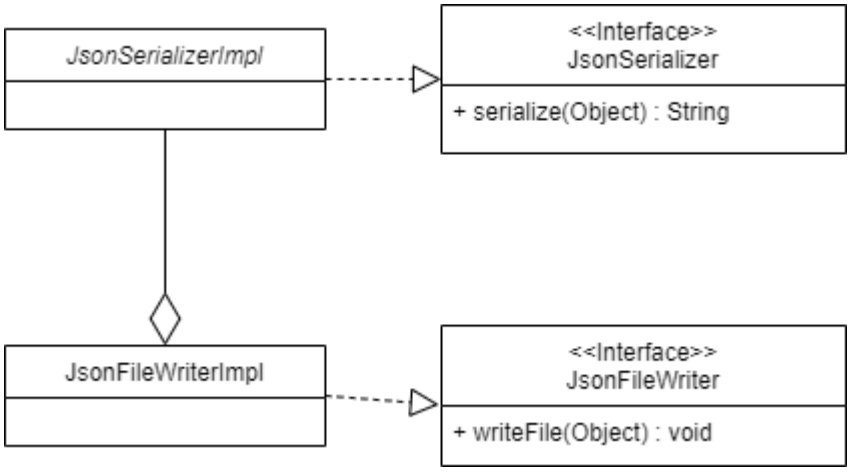

Figura 2.17: Scrittura su File.

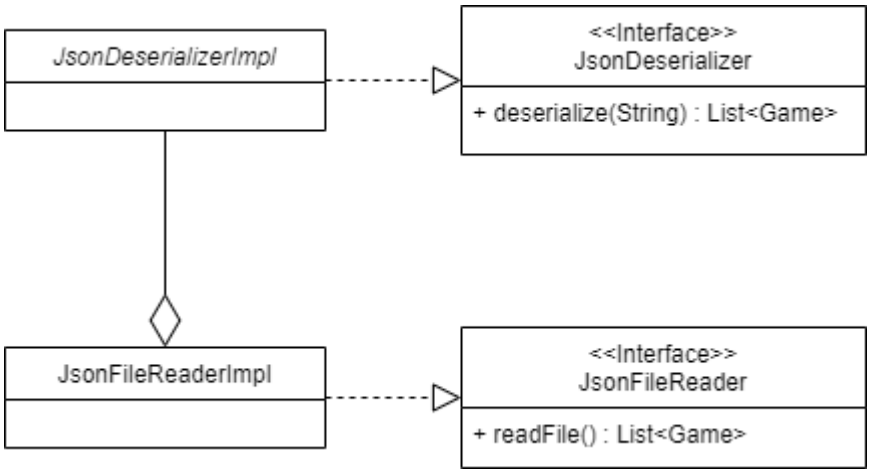

Figura 2.18: Lettura da File.

CONTROLLER

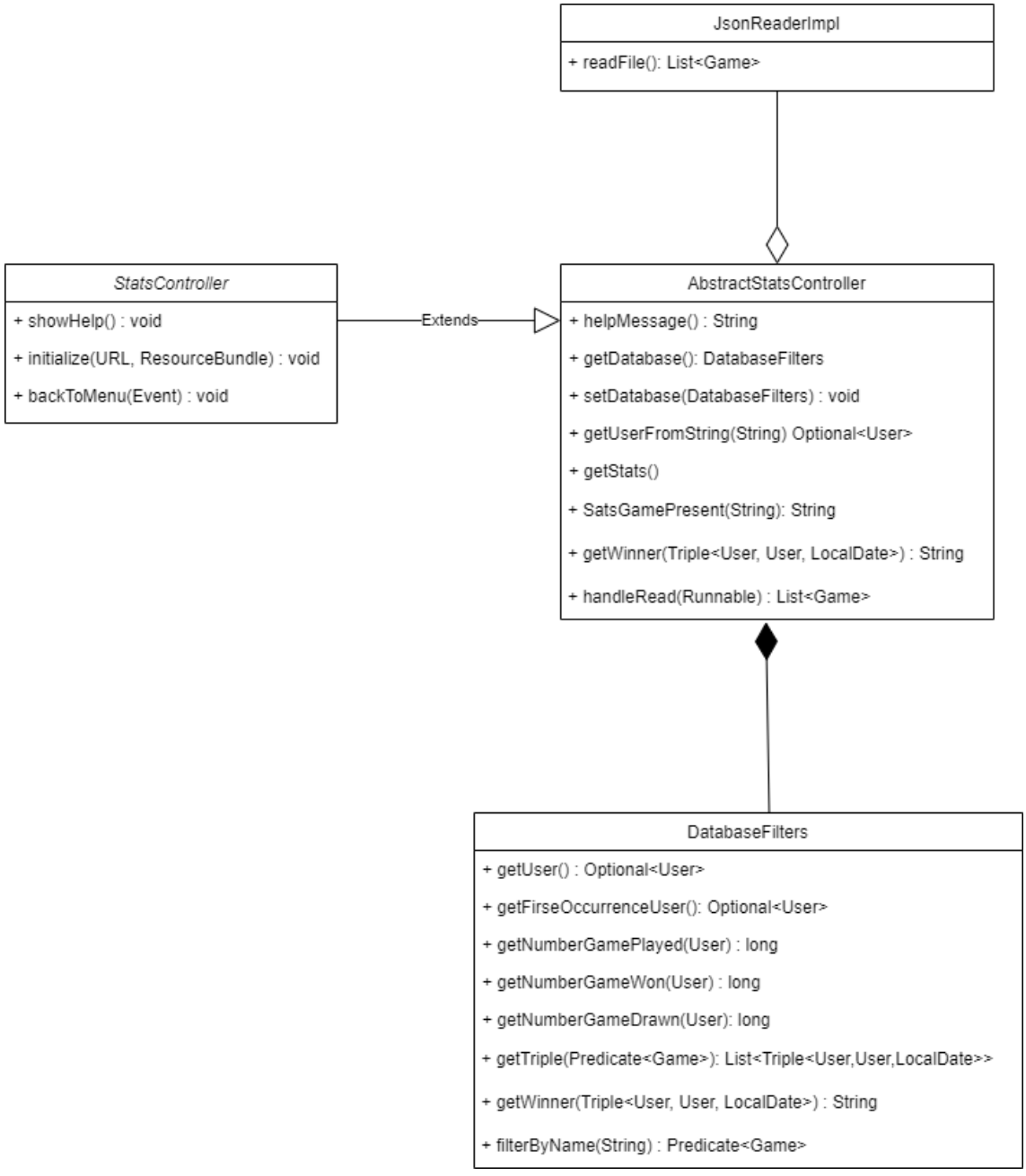

Figura 2.19: Controller.

Ho creato una classe astratta contenente funzioni che ritornano le stringhe da visualizzare. In questo modo è possibile astrarre al massimo la parte di view dal controller e rendere indipendenti tutte le parti del pattern MVC. Nel caso si desideri fare modifiche alla vista, si può creare un controller estendendo da AbstractStatsController e, in questa classe, implementare le funzioni desiderate. AbstractStatsController sfrutta anche la classe DatabaseFilters: quest'ultima permette di estrarre la stringa richiesta tramite l'utilizzo di filtri di ricerca e funzioni per il calcolo delle statistiche. Infine si è creata l'implementazione della classe StatsController che gestisce la corrispondente parte nella vista.

VIEW
Per quanto riguarda la view gli elementi principali da me sviluppati sono: una tableView che visualizza la lista di partite giocate finora, una textField che filtra le partite in base al nome del giocatore richiesto e una textArea per visualizzare le statistiche. Il mio principale obiettivo per ciò che riguarda la view è stato quello di renderla facile da usare e ridimensionabile.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il funzionamento delle classi del model è stato testato con l'ausilio della libreria JUnit 5. Di grande utilità sono stati anche gli IDE utilizzati, Eclipse ed IntelliJ, che con i loro strumenti per il debugging sono stati fondamentali nell'individuazione e nella risoluzione dei problemi incontrati durante lo sviluppo dell'applicazione.

3.1.1 Marco Raggini

Classi di testing realizzate:

- chessboard.ChessboardTest
- game.test.GameTest

3.1.2 Angela Speranza

Classi di testing realizzate:

- castling.CastlingTest
- endgame.EndGameTest

3.1.3 Lorenzo Tosi

Classi di testing realizzate:

- Pieces.test.*

- position.test.PositionTest
- promotion.PromotionTest

3.1.4 Andrea Zavatta

Classi di testing realizzate:

- FenBuilderTest
- FenConverterTest
- MoveBuilderTest
- ControlCheckTest
- IOTest

3.2 Metodologia di lavoro

3.2.1 Marco Raggini

Ho realizzato in autonomia:

- model.game.*
- model.board.Chessboard
- model.board.ChessboardFactory
- model.tuple.Pair
- model.user.*
- controller.user.*
- controller.GuiPieceFactory
- controller.UserHandlerController
- controller.utils.PieceImagePath
- UserHandler.fxml

Ho inoltre contribuito a:

- controller.BoardController
- controller.ControllerUtils

3.2.2 Angela Speranza

In autonomia ho realizzato le seguenti classi:

- application.Laucher
- application.Start
- controller.MenuController
- controller.TutorialController
- model.board.Castling
- model.board.CastlingImpl
- model.board.EndGame
- model.board.EndGameImpl
- timer.*
- resources.tutorial.* (FXML)
- MainMenu.fxml
- Tutorial.fxml

In collaborazione con i miei colleghi ho realizzato:

- controller.BoardController
- controller.ControllerUtils
- resources.stylesheets.style (css)

3.2.3 Lorenzo Tosi

Ho realizzato in autonomia:

- model.move.*
- model.pieces.*
- model.piece.utils.*
- model.promotion.*

- view.GuiPiece
- controller.BoardControllerUtils
- controller.utils.ColorSettings
- Board.fxml

Ho inoltre contribuito a:

- controller.BoardController
- controller.ControllerUtils

3.2.4 Andrea Zavatta

Classi implementate in autonomia:

- FenBuilder
- FenConverterImpl
- MoveBuilder
- JsonSerializerImpl
- JsonDeserializerImpl
- JsonFileReaderImpl
- JsonFileWriterImp
- StatsController
- Stats.fxml
- ControlCheckImpl
- Triple
- KingNotFoundException
- IllegalMoveException.

Classi implementate in collaborazione con i colleghi:

- ControllerUtils.

3.3 Note di sviluppo

3.3.1 Marco Raggini

- JavaFx: framework utilizzato per lo sviluppo dell'interfaccia della scelta utente.
- Stream: utilizzate molto spesso all'interno del codice per visitare le strutture basi.
- Lambda: utilizzate, insieme agli stream, per semplificare il codice.
- Optional: utilizzati per trovare un pezzo data una posizione e per il vincitore, nel caso del pareggio.
- Generici per la classe Pair.

3.3.2 Angela Speranza

In questa sezione riporto un elenco alcune feature avanzate che ho trovato utili per la mia parte.

- JavaFX: è il framework selezionato utilizzato per la creazione delle viste del menù principale e del tutorial.
- Stream: un potente strumento, utile per l'esplorazione e la fruizione di informazione ricavate dalle principali strutture dati utilizzate nel nostro codice.
- Lambda expressions: utilizzate, insieme agli stream, per semplificare e abbellire il codice.

3.3.3 Lorenzo Tosi

- JavaFX framework utilizzato per lo sviluppo della schermata di gioco.
- Stream utilizzati per la visitazione di strutture dati.
- Lambda utilizzate assieme agli stream per semplificare il codice.
- Optional utilizzato per evitare l'utilizzo di null nel codice, spesso per controllare la presenza di un pezzo.

3.3.4 Andrea Zavatta

- Optional: utilizzati come ritorno delle funzioni in tutte le situazioni in cui la funzione poteva ritornare null.
- Stream: utilizzati in larga parte all'interno del codice, soprattutto per il controllo delle mosse.
- Lambda: utilizzate per semplificare il codice nell'utilizzo di functional interfaces.
- Generici: utilizzati per creare la classe Triple.
- JavaFX: framework utilizzato per la creazione della View delle statistiche.
- Jackson: libreria utilizzata per scrivere e leggere in formato json nei file.
    1. Jackson Module Parameter names: per supportare l'introspezione dei nomi dei parametri dei metodi e dei costruttori, senza dover aggiungere l'annotazione esplicita del nome della proprietà.
    2. Jackson datatype: per deserializzare la data.
    3. Jackson Databind: per scrivere il formato json ben formattato su file.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Marco Raggini

Questa è stata per me la mia prima esperienza all'interno di un team. Durante il progetto ho avuto modo di sbagliare tanto e imparare dai miei errori e dagli errori dei miei colleghi, sono molto soddisfatto dal lavoro svolto in quanto all'inizio sembrava veramente difficile finire un progetto di medie dimensioni. Questo progetto mi ha permesso inoltre di capire meglio il funzionamento di git, ma soprattutto mi ha dato la possibilità di imparare a lavorare in un team di sviluppo, skill fondamentale per il mondo del lavoro. Sono soddisfatto del codice da me scritto e credo di aver contribuito in maniera attiva allo sviluppo del progetto dando consigli e confrontandomi con i colleghi. Ammetto però di avere commesso alcuni errori nell'utilizzo di git e in certe situazioni aver calcolato male i tempi di sviluppo, questo mi ha portato a fine progetto a non aver avuto il tempo di rivedere al meglio la correttezza e l'ottimizzazione del codice.

4.1.2 Angela Speranza

Mi ritengo piuttosto soddisfatta del lavoro da me svolto. Per molti miei colleghi, come anche per me, questa è stata la primissima occasione per toccare con mano cosa vuol dire sviluppare software in team. Nel mio caso, però, si tratta anche di una delle prime esperienze anche per quanto riguarda lo sviluppo di un vero e proprio progetto di programmazione: non ho alcuna esperienza di programmazione antecedente all'università e devo dire che, sebbene all'inizio fossi abbastanza intimorita dalla quantità di lavoro che ci aspettava, il lavoro di squadra mi ha permesso di riuscire nel mio dovere di

componente. Ho imparato che i problemi sono inevitabili e possono derivare dalle cause più disparate. Nel nostro caso, la comunicazione è stata la vera chiave ed il più importante punto di forza. Non è stato fondamentale nemmeno l'essere sempre sulla stessa lunghezza d'onda, poiché, grazie ad una attenta analisi della situazione, siamo sempre riusciti a sbrogliare la matassa, con pazienza. Da questa esperienza ho imparato anche che lavorare per me stessa, quando sono in team, significa anche e soprattutto lavorare per gli altri. La programmazione ad oggetti è un oceano di cui mi sembra di aver toccato solo qualche goccia: spero in futuro di riuscire sia ad approfondire la mia conoscenza tecnica, sia a migliorare le mie skills da programmatrice.

4.1.3 Lorenzo Tosi

Questa è stata la mia prima esperienza nella realizzazione di un software, specialmente in gruppo. È stata un'esperienza molto stimolante. Ho imparato le basi del lavoro di gruppo e ho rafforzato la mia conoscenza di Git. Lavorare ad un progetto come questo mi ha fatto rendere conto di quanto sia importante la parte di progettazione e di suddivisone dei compiti. Personalmente penso in generale, sia stato svolto un buon lavoro di progettazione; tuttavia, a parer mio, non abbiamo svolto un lavoro eccellente nella divisione dei lavori, in quanto abbiamo sovrastimato alcune parti. Partecipare alla progettazione del software è stato molto interessante, e penso di aver fatto un buon lavoro per quanto riguarda la mia parte di model. Sono infatti molto soddisfatto di ciò che ho sviluppato e penso che il mio contributo al gruppo sia stato utile al raggiungimento dell'obiettivo finale. È stato anche molto stimolante la risoluzione di problemi di carattere generale, in quanto ogni partecipante ha proposto la propria idea e siamo arrivati ad una soluzione prendendo le parti più interessanti di ogni idea e combinandole. Penso che il mio "tallone d'Achille" sia il controller; anche se è una classe (BoardController) nella quale ho lavorato assieme ai miei colleghi, penso di non aver speso abbastanza tempo nella sua progettazione e che avrei potuto alleggerire la classe scorporandola in altre sottoclassi. In conclusione, grazie a questo progetto ho imparato molto e mi sono reso conto dei miei punti di forza e delle mie debolezze.

4.1.4 Andrea Zavatta

Per quanto riguarda la mia esperienza, questo progetto è stato il primo software di medie dimensioni. Ho imparato a lavorare in gruppo e sfruttare tool importanti come Git, ma soprattutto ho avuto l'occasione di apprendere nuove skills da sviluppatore, apprendendo in questo percorso buone pratiche di

programmazione, sfruttando anche pattern noti. Sono soddisfatto di essermi occupato della gestione del salvataggio su file e quindi di approfondire in dettaglio la libreria Jackson, soprattutto perché ho gestito in autonomia argomenti non trattati a lezione. La risoluzione dei problemi non è stata banale: in fase di progettazione ho analizzato anche le performance di alcune classi che avrebbero potuto dare problemi di tempistiche all'interno dell'applicazione. Questa fase non è stata facile: in molte situazioni ho dovuto capire come fare in modo che le funzioni non si richiamassero a vicenda e quindi come strutturare il codice in modo che questo non accadesse. Soprattutto per quanto riguarda questo punto, ammetto che non avevo potuto prevedere alcune situazioni a causa della poca esperienza. Questo mi ha fatto spendere tempo aggiuntivo nella fase di scrittura del codice, ma adesso ho compreso l'importanza della fase di progettazione. Penso di avere dato un buon aiuto al gruppo per la creazione di questo progetto, e mi piacerebbe continuare a lavorare per rilasciare altre versioni. In particolare, per quanto riguarda la mia parte, vorrei integrare il progetto con un database vero e proprio.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Angela Speranza

Nel mio caso, le difficoltà che ho incontrato non sono state poche, anche se quasi mai invalidanti. La prima è stata relativa alla divisione del lavoro: ho compreso che l'analisi che precede la realizzazione del progetto vero e proprio è molto più profonda di quanto sembri e per questo motivo alcuni compiti non sono stati divisi equamente. In secondo luogo, ho seguito con entusiasmo le lezioni relative ai pattern di programmazione più importanti proposti dal programma di studio. Ritengo che abbiano un'utilità fondamentale nella programmazione di qualità e mi è dispiaciuto non essere riuscita a metterne in pratica molti. Non sono poi mancate difficoltà, anche se lievi, nell'interfacciarsi con strumenti nuovi come Git, Gradle e JavaFX.

Appendice A

Guida utente

L'applicativo, appena avviato, presenta un menù piuttosto intuitivo. Per utilizzare l'applicazione si può tranquillamente fare uso del puntatore del mouse, anche per lo spostamento dei pezzi: quest'ultimo, infatti, è stato implementato tramite la modalità drag-and-drop.

Appendice B

Esercitazioni di laboratorio

B.0.1 Marco Raggini

- Laboratorio 6: https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p135531
- Laboratorio 7: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p136847
- Laboratorio 9: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p139151
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p140294

B.0.2 Lorenzo Tosi

- Laboratorio 7: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p139367
- Laboratorio 8: https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p139359
- Laboratorio 9: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p139356
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p140271